Exercises for Algorithms and Data Structures

Antonio Carzaniga
Faculty of Informatics
USI
(Università della Svizzera italiana)

Edition 3.06
April 2024
(with some solutions)

►**Exercise 1 (m06).** Answer the following questions on the big-oh notation.

Question 1: Explain what $g(n) = O(f(n))$ means. *(5')*

Question 2: Explain why it is meaningless to state that "the running time of algorithm A is *at least* $O(n^2)$." *(5')*

Question 3: Given two functions $f = \Omega(\log n)$ and $g = O(n)$, consider the following statements. For each statement, write whether it is true or false. For each false statement, write two functions f and g that show a counter-example. *(5')*

- $g(n) = O(f(n))$
- $f(n) = O(g(n))$
- $f(n) = \Omega(\log(g(n)))$
- $f(n) = \Theta(\log(g(n)))$
- $f(n) + g(n) = \Omega(\log n)$

Question 4: For each one of the following statements, write two functions f and g that satisfy the given condition. *(5')*

- $f(n) = O(g^2(n))$
- $f(n) = \omega(g(n))$
- $f(n) = \omega(\log(g(n)))$
- $f(n) = \Omega(f(n)g(n))$
- $f(n) = \Theta(g(n)) + \Omega(g^2(n))$

►**Exercise 2 (m06).** Illustrate the execution of the *merge-sort* algorithm on the array

$$A = \langle 3, 13, 89, 34, 21, 44, 99, 56, 9 \rangle$$

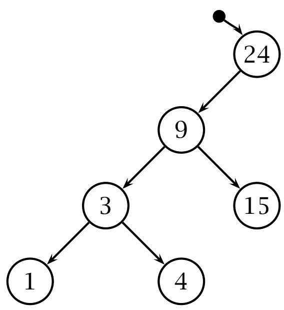
For each fundamental iteration or recursion of the algorithm, write the content of the array. Assume the algorithm performs an in-place sort. *(20')*

►**Exercise 3 (m06).** Consider the array $A = \langle 29, 18, 10, 15, 20, 9, 5, 13, 2, 4, 15 \rangle$.

Question 1: Does A satisfy the *max-heap* property? If not, fix it by swapping two elements. *(5')*

Question 2: Using array A (possibly corrected), illustrate the execution of the *heap-extract-max* algorithm, which extracts the max element and then rearranges the array to satisfy the *max-heap* property. For each iteration or recursion of the algorithm, write the content of the array A. *(15')*

►**Exercise 4 (m06).** Consider the following binary search tree (BST).



Question 1: List all the possible insertion orders (i.e., permutations) of the keys that could have produced this BST. *(5')*

Question 2: Draw the same BST after the insertion of keys: 6, 45, 32, 98, 55, and 69, in this order. *(5')*

Question 3: Draw the BST resulting from the deletion of keys 9 and 45 from the BST resulting from question 2. *(5')*

Question 4: Write at least three insertion orders (permutations) of the keys remaining in the BST after question 3 that would produce a balanced tree (i.e., a minimum-height tree). *(5')*

►**Exercise 5 (m06).** Consider a hash table that stores integer keys. The keys are 32-bit unsigned values, and are always a power of 2. Give the minimum table size t and the hash function $h(x)$ that takes a key x and produces a number between 1 and t, such that no collision occurs. *(10')*

►**Exercise 6 (m06).** Explain why the time complexity of searching for elements in a hash table, where conflicts are resolved by chaining, decreases as its load factor α decreases. Recall that α is defined as the ratio between the total number of elements stored in the hash table and the number of slots in the table.

►**Exercise 7 (f06).** The binary string below is the title of a song encoded using Huffman codes.

0011000101111101100111011101100000100111010010101

Given the letter frequencies listed in the table below, build the Huffman codes and use them to decode the title. In cases where there are multiple "greedy" choices, the codes are assembled by combining the first letters (or groups of letters) from left to right, in the order given in the table. Also, the codes are assigned by labeling the left and right branches of the prefix/code tree with '0' and '1', respectively.

letter	a	h	v	w	' '	e	t	l	o
frequency	1	1	1	1	2	2	2	3	3

(20')

►**Exercise 8 (f06).** You wish to create a database of stars. For each star, the database will store several megabytes of data. Considering that your database will store billions of stars, choose the data structure that will provide the best performance. With this data structure you should be able to find, insert, and delete stars. Justify your choice. *(10')*

►**Exercise 9 (f06).** You are given a set of persons P and their friendship relation R. That is, $(a, b) \in R$ if and only if a is a friend of b. You must find a way to introduce person x to person y through a chain of friends. Model this problem with a graph and describe a strategy to solve the problem. *(10')*

►**Exercise 10 (f06).** Answer the following questions

Question 1: Explain what $f(n) = \Omega(g(n))$ means. *(5')*

Question 2: Explain what kind of problems are in the P complexity class. *(5')*

Question 3: Explain what kind of problems are in the NP complexity class. *(5')*

Question 4: Explain what it means for problem A to be *polynomially-reducible* to problem B. *(5')*

Question 5: Write *true*, *false*, or *unknown* depending on whether the assertions below are true, false, or we do not know. *(5')*

- $P \subseteq NP$
- $NP \subseteq P$
- $n! = O(n^{100})$
- $\sqrt{n} = \Omega(\log n)$
- $3n^2 + \frac{1}{n} + 4 = \Theta(n^2)$

Question 6: Consider the following *exact-change problem*. Given a collection of n values $V = \{v_1, v_2, \ldots, v_n\}$ representing coins and bills in a cash register, and given a value x, output 1 if there exists a subset of V whose total value is equal to x, or 0 otherwise. Is the exact-change problem in NP? Justify your answer. *(5')*

▸**Exercise 11 (f06).** A thief robbing a gourmet store finds n pieces of precious cheeses. For each piece i, v_i designates its value and w_i designates its weight. Considering that W is the maximum weight the thief can carry, and considering that the thief may take any fraction of each piece, you must find the quantity of each piece the thief must take to maximize the value of the robbery. *(20')*

Question 1: Write an algorithm that solves the problem using a *greedy* or *dynamic programming* strategy. Analyze the complexity of your solution.

Question 2: Prove that the problem has an *optimal substructure*, meaning that an optimal solution to a problem instance X contains within it some optimal solutions to sub-problems $Y \subseteq X$.

Question 3: Show the *greedy choice property* also holds for some greedy-choice strategy. Recall that the greedy-choice property holds if and only if every greedy choice according to the given strategy is contained in an optimal solution.

▸**Exercise 12 (f06).** You are in front of a stack of pancakes of different diameter. Unfortunately, you cannot eat them unless they are sorted according to their size, with the biggest one at the bottom. To sort them, you are given a spatula that you can use to split the stack in two parts and then flip the top part of the stack. Write the an algorithm SORT-PANCAKES(P) that sorts the stack P using only spatula-flip operations. The array P stores the pancakes top-to-bottom, thus $P[1]$ is the size of the pancake at the top of the stack, while $P[P.length]$ is the size of the pancake at the bottom of the stack. Your algorithm must indicate a spatula flip with the spatula inserted at position i with SPATULA-FLIP(P, i), which flips all the elements in $P[1 \dots i]$. *(20')*

▸**Exercise 13 (f06).** Explain what it means for a hash function to be perfect for a given set of keys. Consider the hash function $h(x) = x \mod m$ that maps an integer x to a table entry in $\{0, 1, \dots m-1\}$. Find an $m \leq 12$ such that h is a perfect hash function on the set of keys $\{0, 6, 9, 12, 22, 31\}$. *(10')*

▸**Exercise 14 (f06).** Draw the binary search tree obtained when the keys $1, 2, 3, 4, 5, 6, 7$ are inserted in the given order into an initially empty tree. What is the problem of the tree you get? Why is it a problem? How could you modify the insertion algorithm to solve this problem. Justify your answer. *(10')*

▸**Exercise 15 (f06).** Consider the following array:

$$A = \langle 4, 33, 6, 90, 33, 32, 31, 91, 90, 89, 50, 33 \rangle$$

Question 1: Is A a *min-heap*? Justify your answer by briefly explaining the *min-heap* property. *(10')*

Question 2: If A is a *min-heap*, then extract the minimum value and then rearrange the array with the *min-heapify* procedure. In doing that, show the array at every iteration of *min-heapify*. If A is not a *min-heap*, then rearrange it to satisfy the *min-heap* property. *(10')*

▸**Exercise 16 (f06).** Write the pseudo-code of the *insertion-sort* algorithm. Illustrate the execution of the algorithm on the array $A = \langle 3, 13, 89, 34, 21, 44, 99, 56, 9 \rangle$, writing the intermediate values of A at each iteration of the algorithm. *(20')*

▸**Exercise 17 (f06).** Encode the following sentence with a Huffman code

Common sense is the collection of prejudices acquired by age eighteen

Write the complete construction of the code. *(20')*

▸**Exercise 18 (f06).** Consider the *text* and *query* strings:
text: `It ain't over till it's over.`
query: `over`
Use the Boyer-Moore string-matching algorithm to search for the query in the text. For each character comparison performed by the algorithm, write the current *shift* and highlight the character position considered in the query string. Assume that indexes start from 0. The following table shows the first comparison as an example. Fill the rest of the table. *(10')*

n.	*shift*	I	t		a	i	n	'	t		o	v	e	r		t	i	l	l		i	t	'	s		o	v	e	r	.
1	0	o	v	e	$\underline{r}$																									
2																														
...	...															...														

▸**Exercise 19 (f06).** Briefly answer the following questions

Question 1: What does $f(n) = \Theta(g(n))$ mean? *(5')*

Question 2: What kind of problems are in the P class? Give an example of a problem in P. *(5')*

Question 3: What kind of problems are in the NP class? Give an example of a problem in NP. *(5')*

Question 4: What does it mean for a problem A to be *reducible* to a problem B? *(5')*

▸**Exercise 20 (f06).** For each of the following assertions, write "true," "false," or "?" depending on whether the assertion is true, false, or it may be either true or false. *(10')*

Question 1: $P \subseteq NP$

Question 2: The *knapsack* problem is in P

Question 3: The *minimal spanning tree* problem is in NP

Question 4: $n! = O(n^{100})$

Question 5: $\sqrt{n} = \Omega(\log(n))$

Question 6: *insertion-sort* performs like *quicksort* on an almost sorted sequence

▸**Exercise 21 (f06).** An application must read a long sequence of numbers given in no particular order, and perform many searches on that sequence. How would you implement that application to minimize the overall time-complexity? Write exactly what algorithms you would use, and in what sequence. In particular, write the high-level structure of a *read* function, to read and store the sequence, and a *find* function too look up a number in the sequence. *(10')*

▸**Exercise 22 (m07).** For each statement below, write whether it is true or false. For each false statement, write a counter-example. *(10')*

- $f(n) = \Theta(n) \wedge g(n) = \Omega(n) \Rightarrow f(n)g(n) = \Omega(n^2)$
- $f(n) = \Theta(1) \Rightarrow n^{f(n)} = O(n)$
- $f(n) = \Omega(n) \wedge g(n) = O(n^2) \Rightarrow g(n)/f(n) = O(n)$
- $f(n) = O(n^2) \wedge g(n) = O(n) \Rightarrow f(g(n)) = O(n^3)$
- $f(n) = O(\log n) \Rightarrow 2^{f(n)} = O(n)$
- $f = \Omega(\log n) \Rightarrow 2^{f(n)} = \Omega(n)$

▸**Exercise 23 (m07).** Write tight asymptotic bounds for each one of the following definitions of $f(n)$. *(10')*

- $g(n) = \Omega(n) \wedge f(n) = g(n)^2 + n^3 \Rightarrow f(n) =$
- $g(n) = O(n^2) \wedge f(n) = n \log(g(n)) \Rightarrow f(n) =$
- $g(n) = \Omega(\sqrt{n}) \wedge f(n) = g(n + 2^{16}) \Rightarrow f(n) =$
- $g(n) = \Theta(n) \wedge f(n) = 1 + 1/\sqrt{g(n)} \Rightarrow f(n) =$
- $g(n) = O(n) \wedge f(n) = 1 + 1/\sqrt{g(n)} \Rightarrow f(n) =$
- $g(n) = O(n) \wedge f(n) = g(g(n)) \Rightarrow f(n) =$

▸**Exercise 24 (m07).** Write the ternary-search trie (TST) that represents a dictionary of the strings: "gnu" "emacs" "gpg" "else" "gnome" "go" "eps2eps" "expr" "exec" "google" "elif" "email" "exit" "epstopdf" *(10')*

▸**Exercise 25 (m07).** Answer the following questions.

Question 1: A hash table with chaining is implemented through a table of K slots. What is the expected number of steps for a search operation over a set of $N = K/2$ keys? Briefly justify your answers.

Question 2: What are the worst-case, average-case, and best-case complexities of *insertion-sort*, *bubble-sort*, *merge-sort*, and *quicksort*? *(5')*

►**Exercise 26 (m07).** Write the pseudo code of the in-place *insertion-sort* algorithm, and illustrate its execution on the array

$$A = \langle 7, 17, 89, 74, 21, 7, 43, 9, 26, 10 \rangle$$

Do that by writing the content of the array at each main (outer) iteration of the algorithm. *(20')*

►**Exercise 27 (m07).** Consider a binary tree containing N integer keys whose values are all less than K, and the following FIND-PRIME algorithm that operates on this tree.

```
FIND-PRIME(T)
x = TREE-MIN(T)
while x ≠ NIL
    x = TREE-SUCCESSOR(x)
    if IS-PRIME(x.key)
        return x
return x
```

```
IS-PRIME(n)
i = 2
while i · i ≤ n
    if i divides n
        return FALSE
    i = i + 1
return TRUE
```

Hint: these are the relevant binary-tree algorithms.

```
TREE-SUCCESSOR(x)
if x.right ≠ NIL
return TREE-MINIMUM(x.right)
y = x.parent
while y ≠ NIL and x == y.right
    x = y
    y = y.parent
return y
```

```
TREE-MINIMUM(x)
while x.left ≠ NIL
    x = x.left
return x
```

Write the time complexity of FIND-PRIME. Justify your answer. *(10')*

►**Exercise 28 (m07).** Consider the following *max-heap*

$$H = \langle 37, 12, 30, 10, 3, 9, 20, 3, 7, 1, 1, 7, 5 \rangle$$

Write the exact output of the following EXTRACT-ALL algorithm run on H

```
EXTRACT-ALL(H)
while H.heap-size > 0
    HEAP-EXTRACT-MAX(H)
    for i = 1 to H.heap-size
        output H[i]
    output "." END-OF-LINE
```

```
HEAP-EXTRACT-MAX(H)
if H.heap-size > 0
    k = H[1]
    H[1] = H[H.heap-size]
    H.heap-size = H.heap-size − 1
    MAX-HEAPIFY(H)
return k
```

(20')

►**Exercise 29 (m07).** Develop an efficient in-place algorithm called PARTITION-EVEN-ODD(A) that partitions an array A in *even* and *odd* numbers. The algorithm must terminate with A containing all its *even* elements preceding all its *odd* elements. For example, with $A = \langle 7, 17, 74, 21, 7, 9, 26, 10 \rangle$, the result might be $A = \langle 74, 10, 26, 17, 7, 21, 9, 7 \rangle$. PARTITION-EVEN-ODD must be an *in-place* algorithm, which means that it may use only a constant memory space in addition to A. In practice, this means that you may not use another temporary array.

Question 1: Write the pseudo-code for PARTITION-EVEN-ODD. *(20')*

Question 2: Characterize the complexity of PARTITION-EVEN-ODD. Briefly justify your answer. *(10')*

Question 3: Formalize the correctness of the partition problem as stated above, and prove that PARTITION-EVEN-ODD is correct using a loop-invariant. *(20')*

Question 4: If the complexity of your algorithm is not already linear in the size of the array, write a new algorithm PARTITION-EVEN-ODD-OPTIMAL with complexity $O(N)$ (with $N = |A|$). *(20')*

►**Exercise 30 (f07).** The following matrix represents a directed graph over vertices $a, b, c, \ldots, \ell$. Rows and columns represent the source and destination of edges, respectively.

	a	b	c	d	e	f	g	h	i	j	k	ℓ
a					1	1						
b										1		
c								1			1	
d			1									
e		1								1		
f		1								1		
g			1	1								
h											1	1
i			1				1					
j												
k												1
ℓ												

Sort the vertices in a *reverse topological order* using the *depth-first search* algorithm. (**Hint:** if you order the vertices from left to right in reverse topological order, then all edges go from right to left.) Justify your answer by showing the relevant data maintained by the depth-first search algorithm, and by explaining how that can be used to produce a reverse topological order. *(15')*

►**Exercise 31 (f07).** Answer the following questions on the complexity classes P an NP. Justify your answers.

Question 1: P $\subseteq$ NP? *(5')*

Question 2: A problem Q is in P and there is a polynomial-time reduction from Q to Q'. What can we say about Q'? Is $Q' \in$ P? Is $Q' \in$ NP? *(5')*

Question 3: Let Q be a problem defined as follows. *Input:* a set of numbers $A = \{a_1, a_2, \ldots, a_N\}$ and a number x; *Output:* 1 if and only if there are two values $a_i, a_k \in A$ such that $a_i + a_k = x$. Is Q in NP? Is Q in P? *(5')*

►**Exercise 32 (f07).** Consider the *subset-sum* problem: given a set of numbers $A = \{a_1, a_2, \ldots, a_n\}$ and a number x, output TRUE if there is a subset of numbers in A that add up to x, otherwise output FALSE. Formally, $\exists S \subseteq A$ such that $\sum_{y \in S} y = x$. Write a dynamic-programming algorithm to solve the subset-sum problem and informally analyze its complexity. *(20')*

►**Exercise 33 (f07).** Explain the idea of *dynamic programming* using the shortest-path problem as an example. (The shortest path problem amounts to finding the shortest path in a given graph $G = (V, E)$ between two given vertices a and b.) *(15')*

►**Exercise 34 (f07).** Consider an initially empty B-Tree with minimum degree $t = 3$. Draw the B-Tree after the insertion of the keys $27, 33, 39, 1, 3, 10, 7, 200, 23, 21, 20$, and then after the additional insertion of the keys $15, 18, 19, 13, 34, 200, 100, 50, 51$. *(10')*

►**Exercise 35 (f07).** There are three containers whose sizes are 10 pints, 7 pints, and 4 pints, respectively. The 7-pint and 4-pint containers start out full of water, but the 10-pint container is initially empty. Only one type of operation is allowed: pouring the contents of one container into another, stopping only when the source container is empty, or the destination container is full. Is there a sequence of pourings that leaves exactly two pints in either the 7-pint or the 4-pint container?

Question 1: Model this as a graph problem: give a precise definition of the graph involved (type of the graph, labels on vertices, meaning of an edge). Provide the set of all reachable vertices, identify the initial vertex and the goal vertices. (**Hint:** all vertices that satisfy the condition imposed by the problem are reachable, so you don't have to draw a graph.)

Question 2: State the specific question about this graph that needs to be answered?

Question 3: What algorithm should be applied to solve the problem? Justify your answer. *(15')*

▸**Exercise 36 (f07).** Write an algorithm called MOVETOROOT(x, k) that, given a binary tree rooted at node x and a key k, moves the node containing k to the root position and returns that node if k is in the tree. If k is not in the tree, the algorithm must return x (the original root) without modifying the tree. Use the typical notation whereby $x.key$ is the key stored at node x, $x.left$ and $x.right$ are the left and right children of x, respectively, and $x.parent$ is x's parent node. *(15')*

▸**Exercise 37 (f07).** Given a sequence of numbers $A = \langle a_1, a_2, \ldots, a_n \rangle$, an *increasing subsequence* is a sequence $a_{i_1}, a_{i_2}, \ldots, a_{i_k}$ of elements of A such that $1 \le i_1 < i_2 < \ldots < i_k \le n$, and such that $a_{i_1} < a_{i_2} < \ldots < a_{i_k}$. You must find the *longest increasing subsequence*. Solve the problem using dynamic programming.

Question 1: Define the *subproblem structure* and the solution of each subproblem. *(5')*

Question 2: Write an iterative algorithm that solves the problem. Illustrate the execution of the algorithm on the sequence $A = \langle 2, 4, 5, 6, 7, 9 \rangle$. *(10')*

Question 3: Write a recursive algorithm that solves the problem. Draw a tree of recursive calls for the algorithm execution on the sequence $A = \langle 1, 2, 3, 4, 5 \rangle$. *(10')*

Question 4: Compare the time complexities of the iterative and recursive algorithms. *(5')*

▸**Exercise 38 (f07).** One way to implement a *disjoint-set* data structure is to represent each set by a linked list. The first node in each linked list serves as the representative of its set. Each node contains a key, a pointer to the next node, and a pointer back to the representative node. Each list maintains the pointers *head*, to the representative, and *tail*, to the last node in the list.

Question 1: Write the pseudo-code and analyze the time complexity for the following operations:

- MAKE-SET(x): creates a new set whose only member is x.
- UNION(x, y): returns the representative of the union of the sets that contain x and y.
- FIND-SET(x): returns a pointer to the representative of the set containing x.

Note that x and y are nodes. *(15')*

Question 2: Illustrate the linked list representation of the following sets:

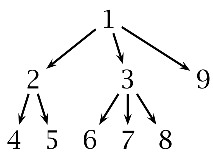
- $\{c, a, d, b\}$
- $\{e, g, f\}$
- UNION(d, g)

(5')

▸**Exercise 39 (r07).** Write an algorithm that takes a set of (x, y) coordinates representing points on a plane, and outputs the coordinates of two points with the maximal distance. The signature of the algorithm is MAXIMAL-DISTANCE(X, Y), where X and Y are two arrays of the same length representing the x and y coordinates of each point, respectively. Also, write the asymptotic complexity of MAXIMAL-DISTANCE. Briefly justify your answer. *(10')*

▸**Exercise 40 (r07).** A *directed tree* is represented as follows: for each vertex v, $v.first\text{-}child$ is either the first element in a list of child-vertices, or NIL if v is a leaf. For each vertex v, $v.next\text{-}sibling$ is the next element in the list of v's siblings, or NIL if v is the last element in the list. For example, the arrays on the left represent the tree on the right:

v	1	2	3	4	5	6	7	8	9
first-child	2	4	6	NIL	NIL	NIL	NIL	NIL	NIL
next-sibling	NIL	3	9	5	NIL	7	8	NIL	NIL



Question 1: Write two algorithms, MAX-DEPTH$(root)$ and MIN-DEPTH$(root)$, that, given a tree, return the maximal and minimal depth of any leaf vertex, respectively. (E.g., the results for the example tree above are 2 and 1, respectively.) *(15')*

Question 2: Write an algorithm DEPTH-FIRST-ORDER$(root)$ that, given a tree, prints the vertices in depth-first visitation order, such that a vertices is always preceded by all its children (e.g., the result for the example tree above is $4, 5, 2, 6, 7, 8, 3, 9, 1$). *(10')*

Question 3: Analyze the complexity of MAX-DEPTH, MIN-DEPTH and DEPTH-FIRST-ORDER. *(5')*

►**Exercise 41 (r07).** Write an algorithm called IN-PLACE-SORT(A) that takes an array of numbers, and sorts the array *in-place*. That is, using only a constant amount of extra memory. Also, give an informal analysis of the asymptotic complexity of your algorithm. *(10')*

►**Exercise 42 (r07).** Given a sequence $A = \langle a_1, \ldots, a_n \rangle$ of numbers, the *zero-sum-subsequence* problem amounts to deciding whether A contains a sequence of consecutive elements $a_i, a_{i+1}, \ldots, a_k$, with $1 \leq i \leq k \leq n$, such that $a_i + a_{i+1} + \cdots + a_k = 0$. Model this as a dynamic-programming problem and write a dynamic-programming algorithm ZERO-SUM-SEQUENCE(A) that, given an array A, returns TRUE if A contains a zero-sum subsequence, or FALSE otherwise. Also, give an informal analysis of the complexity of ZERO-SUM-SEQUENCE. *(30')*

►**Exercise 43 (r07).** Give an example of a randomized algorithm derived from a deterministic algorithm. Explain why there is an advantage in using the randomized variant. *(10')*

►**Exercise 44 (r07).** Implement a TERNARY-TREE-SEARCH(x, k) algorithm that takes the root of a ternary tree and returns the node containing key k. A ternary tree is conceptually identical to a binary tree, except that each node x has two keys, $x.key_1$ and $x.key_2$, and three links to child nodes, $x.left$, $x.center$, and $x.right$, such that the left, center, and right subtrees contains keys that are, respectively, less than $x.key_1$, between $x.key_1$ and $x.key_2$, and greater than $x.key_2$. Assume there are no duplicate keys. Also, assuming the tree is balanced, what is the asymptotic complexity of the algorithm? *(10')*

►**Exercise 45 (r07).** Answer the following questions. Briefly justify your answers.

Question 1: A hash table that uses chaining has M slots and holds N keys. What is the expected complexity of a search operation? *(5')*

Question 2: The asymptotic complexity of algorithm A is $\Omega(N \log N)$, while that of B is $\Theta(N^2)$. Can we compare the two algorithms? If so, which one is asymptotically faster? *(5')*

Question 3: What is the difference between "Las Vegas" and "Monte Carlo" randomized algorithms? *(5')*

Question 4: What is the main difference between the Knuth-Morris-Pratt algorithm and Boyer-Moore string-matching algorithms in terms of complexity? Which one as the best worst-case complexity? *(5')*

►**Exercise 46 (f08).** Consider *quick-sort* as an in-place sorting algorithm.

Question 1: Write the pseudo-code using only *swap* operations to modify the input array. *(10')*

Question 2: Apply the algorithm of question 1 to the array $A = \langle 8, 2, 12, 17, 4, 8, 7, 1, 12 \rangle$. Write the content of the array after each swap operation. *(10')*

►**Exercise 47 (f08).** Consider this *minimal vertex cover* problem: given a graph $G = (V, E)$, find a minimal set of vertices S such that for every edge $(u, v) \in E$, u or v (or both) are in S.

Question 1: Model *minimal vertex cover* as a dynamic-programming problem. Write the pseudo-code of a dynamic-programming solution. *(15')*

Question 2: Do you think that your model of *minimal vertex cover* admits a greedy choice? Try at least one meaningful greedy strategy. Show that it does not work, giving a counter-example graph for which the strategy produces the wrong result. (**Hint:** one meaningful strategy is to choose a maximum-degree vertex first. The degree of a vertex is the number of its incident edges.) *(5')*

►**Exercise 48 (f08).** The graph $G = (V, E)$ represents a social network in which each vertex represents a person, and an edge $(u, v) \in E$ represents the fact that u and v know each other. Your problem is to organize the largest party in which nobody knows each other. This is also called the *maximal independent set* problem. Formally, given a graph $G = (V, E)$, find a set of vertices S of maximal size in which no two vertices are adjacent. (I.e., for all $u \in S$ and $v \in S$, $(u, v) \notin E$.)

Question 1: Formulate a decision variant of *maximal independent set*. Say whether the problem is in NP, and briefly explain what that means. *(10')*

Question 2: Write a verification algorithm for the *maximal independent set* problem. This algorithm, called TESTINDEPENDENTSET(G, S), takes a graph G represented through its adjacency matrix, and a set S of vertices, and returns TRUE if S is a valid independent set for G. *(10')*

▸**Exercise 49 (f08).** A *Hamilton cycle* is a cycle in a graph that touches every vertex exactly once. Formally, in $G = (V, E)$, an ordering of *all* vertices $H = v_1, v_2, \ldots, v_n$ forms a Hamilton cycle if $(v_n, v_1) \in E$, and $(v_i, v_{i+1}) \in E$ for all i between 1 and $n - 1$. Deciding whether a given graph is *Hamiltonian* (has a Hamilton cycle) is a well known NP-complete problem.

Question 1: Write a verification algorithm for the *Hamiltonian graph* problem. This algorithm, called TESTHAMILTONCYCLE(G, H), takes a graph G represented through adjacency lists, and an array of vertices H, and returns TRUE if H is a valid Hamilton cycle in G. *(10')*

Question 2: Give the asymptotic complexity of your implementation of TESTHAMILTONCYCLE. *(5')*

Question 3: Explain what it means for a problem to be NP-complete. *(5')*

▸**Exercise 50 (f08).** Consider using a b-tree with minimum degree $t = 2$ as an in-memory data structure to implement dynamic sets.

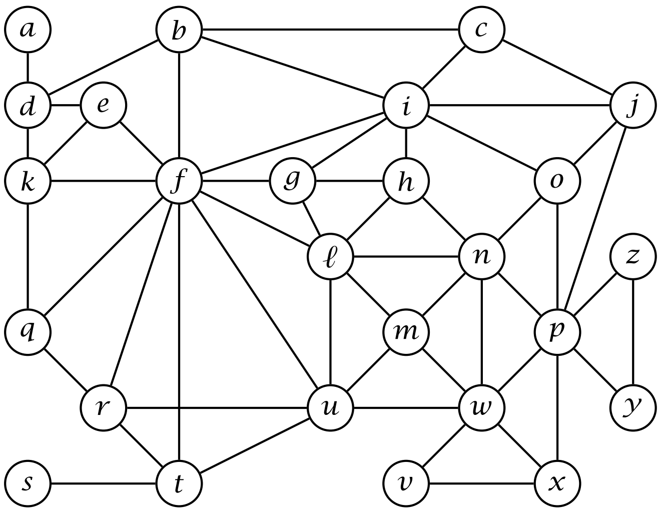
Question 1: Compare this data structure with a red-black tree. Is this data structure better, worse, or the same as a red-black tree in terms of time complexity? Briefly justify your answer. In particular, characterize the complexity of insertion and search. *(10')*

Question 2: Write an iterative (i.e., non-recursive) *search* algorithm for this degree-2 b-tree. Remember that the data structure is *in-memory*, so there is no need to perform any disk read/write operation. *(10')*

Question 3: Write the data structure after the insertion of keys $10, 3, 8, 21, 15, 4, 6, 19, 28, 31$, in this order, and then after the insertion of keys $25, 33, 7, 1, 23, 35, 24, 11, 2, 5$. *(10')*

Question 4: Write the insertion algorithm for this degree-2 b-tree. (**Hint:** since the minimum degree is fixed at 2, the insertion algorithm may be implemented in a simpler fashion without all the loops of the full b-tree insertion.) *(15')*

▸**Exercise 51 (f08).** Consider a breadth-first search (BFS) on the following graph, starting from vertex a.



Write the two vectors π (previous) and d (distance), resulting from the BFS algorithm. *(10')*

▸**Exercise 52 (r08).** Write a sorting algorithm that runs with in time $O(n \log n)$ in the average case (on an input array of size n). Also, characterize the best- and worst-case complexity of your solution. *(20')*

▸**Exercise 53 (r08).** The following algorithms take an array A of integers. For each algorithm, write the asymptotic, best- and worst-case complexities as functions of the size of the input $n = |A|$. Your characterizations should be as tight as possible. Justify your answers by writing a short explanation of what each algorithm does. *(20')*

```
ALGORITHM-I(A)
for i = |A| downto 2
    s = TRUE
    for j = 2 to i
        if A[j − 1] > A[j]
            swap A[j − 1] ↔ A[j]
            s = FALSE
    if s == TRUE
        return
```

```
ALGORITHM-II(A)
i = 1
j = |A|
while i < j
    if A[i] > A[j]
        swap A[i] ↔ A[i + 1]
        if i + 1 < j
            swap A[i] ↔ A[j]
        i = i + 1
    else j = j − 1
```

►**Exercise 54 (r08).** The following algorithms take a binary search tree T containing n keys. For each algorithm, write the asymptotic, best- and worst-case complexities as functions of n. Your characterizations should be as tight as possible. Justify your answers by writing a short explanation of what each algorithm does. *(20')*

```
ALGORITHM-III(T, k)
if T == NIL
    return FALSE
if T.key == k
    return TRUE
if ALGORITHM-III(T.left)
    return TRUE
else return ALGORITHM-III(T.right)
```

```
ALGORITHM-IV(T, k1, k2)
if T == NIL
    return 0
if k1 > k2
    swap k1 ↔ k2
r = 0
if T.key < k2
    r = r + ALGORITHM-IV(T.right, k1, k2)
if T.key > k1
    r = r + ALGORITHM-IV(T.left, k1, k2)
if T.key < k2 and T.key > k1
    r = r + 1
return r
```

►**Exercise 55 (r08).** Answer the following questions on complexity theory. Justify your answers. All problems are decision problems. (*Hint:* answers are not limited to “yes” or “no.”) *(20')*

Question 1: An algorithm A solves a problem P of size n in time $O(n^3)$. Is P in NP?

Question 2: An algorithm A solves a problem P of size n in time $\Omega(n \log n)$. Is P in P? Is it in NP?

Question 3: A problem P in NP can be polynomially reduced into a problem Q. Is Q in P? Is Q in NP?

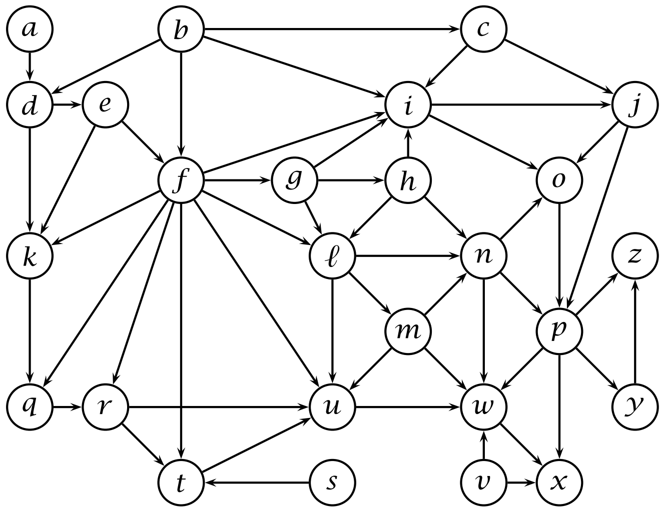
Question 4: A problem P can be polynomially reduced into a problem Q in NP. Is P in P? Is P NP-hard?

Question 5: A problem P of size n does not admit to any algorithmic solution with complexity $O(2^n)$. Is P in P? Is P in NP?

Question 6: An algorithm A takes an instance of a problem P of size n and a "certificate" of size $O(n^c)$, for some constant c, and *verifies* in time $O(n^2)$ that the solution to given problem is affirmative. Is P in P? Is P in NP? Is P NP-complete?

►**Exercise 56 (r08).** Write an algorithm TSTCOUNTGREATER(T, s) that takes the root T of a ternary-search trie (TST) and a string s, and returns the number of strings stored in the trie that are lexicographically greater than s. Given a node T, $T.left$, $T.middle$, and $T.right$ are the left, middle, and right subtrees, respectively; $T.value$ is the value stored in T. The TST uses the special character '#' as the string terminator. Given two characters a and b, the relation $a < b$ defines the lexicographical order, and the terminator character is *less than* every other character. (**Hint:** first write an algorithm that, given a tree (node) counts *all* the strings stored in that tree.) *(20')*

►**Exercise 57 (r08).** Consider a depth-first search (DFS) on the following graph.



Write the three vectors π, d, and f that, for each vertex represent the *previous* vertex in the depth-first forest, the *discovery* time, and the *finish* time, respectively. Whenever necessary, iterate through vertexes in alphabetic order. *(20')*

►**Exercise 58 (r08).** Consider the following algorithm:

```
ALGO-A(X)
d = ∞
for i = 1 to X.length − 1
    for j = i + 1 to X.length
        if |X[i] − X[j]| < d
            d = |X[i] − X[j]|
return d
```

Question 1: Interpreting X as an array of coordinates of points on the x-axis, explain concisely what algorithm ALGO-A does, and give a tight asymptotic bound for the complexity of ALGO-A. *(5')*

Question 2: Write an algorithm BETTER-A(X) that is functionally equivalent to ALGO-A(X), but with a better asymptotic complexity. *(15')*

►**Exercise 59 (r08).** A set of keys is stored in a *max-heap* H and in a *binary search tree* T. Which data structure offers the most efficient algorithm to output all the keys in descending order? Or are the two equivalent? Write both algorithms. Your algorithms may change the data structures. *(20')*

►**Exercise 60 (r08).** Answer the following questions. Briefly justify your answers. *(10')*

Question 1: Let A be an array of numbers sorted in descending order. Does A represent a max-heap (with $A.\mathit{heap\text{-}size} = A.\mathit{length}$)?

Question 2: A hash table has T slots and uses chaining to resolve collisions. What are the worst-case and average-case complexities of a search operation when the hash table contains N keys?

Question 3: A hash table with 9 slots, uses chaining to resolve collision, and uses the hash function $h(k) = k \mod 9$ (slots are numbered $0, \dots, 8$). Draw the hash table after the insertion of keys 5, 28, 19, 15, 20, 33, 12, 17, and 10.

Question 4: Is the operation of deletion in a binary search tree *commutative* in the sense that deleting x and then y from a binary search tree leaves the same tree as deleting y and then x? Argue why it is, or give a counter-example.

►**Exercise 61 (m09).** Draw a binary search tree containing keys $8, 27, 13, 15, 32, 20, 12, 50, 29, 11$, inserted in this order. Then, add keys $14, 18, 30, 31$, in this order, and again draw the tree. Then delete keys 29 and 27, in this order, and again draw the tree. *(10')*

►**Exercise 62 (m09).** Consider a *max-heap* containing keys $8, 27, 13, 15, 32, 20, 12, 50, 29, 11$, inserted in this order in an initially empty heap. Write the content of the array that stores the heap. Then, insert keys 43 and 51, and again write the content of the array. Then, extract the maximum value three times, and again write the content of the array. In all three cases, write the heap as an array. *(10')*

►**Exercise 63 (m09).** Consider a *min-heap* H and the following algorithm.

```
BST-FROM-MIN-HEAP(H)
T = NEW-EMPTY-TREE()
for i = 1 to H.heap-length
     TREE-INSERT(T, H[i])    // binary-search-tree insertion
return T
```

Prove that BST-From-Min-Heap does not always produce minimum-height binary trees. *(10')*

►**Exercise 64 (m09).** Consider an array A containing n numbers and satisfying the *min-heap* property. Write an algorithm Min-Heap-Fast-Search(A, k) that finds k in A with a time complexity that is better than linear in n whenever at most $\sqrt{n}$ of the values in A are less than k. *(20')*

►**Exercise 65 (m09).** Write an algorithm B-Tree-Top-K(R, k) that, given the root R of a b-tree of minimum degree t, and an integer k, outputs the largest k keys in the b-tree. You may assume that the entire b-tree resides in main memory, so no disk access is required. Recall that a node x in a b-tree has the following properties: $x.n$ is the number of keys, $X.\mathit{key}[1] \le x.\mathit{key}[2] \le \dots x.\mathit{key}[x.n]$ are the keys, $x.\mathit{leaf}$ tells whether x is a leaf, and $x.c[1], x.c[2], \dots, x.c[x.n+1]$ are the pointers to x's children. *(30')*

►**Exercise 66 (m09).** Your computer has a special machine instruction called Sort-Five(A, i) that, given an array A and a position i, sorts in-place and in a single step the elements $A[i \dots i+5]$ (or $A[i \dots |A|]$ if $|A| < i+5$). Write an in-place sorting algorithm called Sort-With-Sort-Five that uses only Sort-Five to modify the array A. Also, analyze the complexity of Sort-With-Sort-Five. *(20')*

►**Exercise 67 (m09).** For each of the following statements, briefly argue why they are true, or show a counter-example. *(10')*

Question 1: $f(n) = O(n!) \Longrightarrow \log(f(n)) = O(n \log n)$

Question 2: $f(n) = \Theta(f(n/2))$

Question 3: $f(n) + g(n) = \Theta(\min(f(n), g(n)))$

Question 4: $f(n)g(n) = O(\max(f(n), g(n)))$

Question 5: $f(g(n)) = \Omega(\min(f(n), g(n)))$

►Exercise 68 (m09). Analyze the complexity of the following algorithm. *(10')*

```
SHUFFLE-A-BIT(A)
i = 1
j = A.length
if j > i
    while j > i
        p = CHOOSE-UNIFORMLY({0, 1})
        if p == 1
            swap A[i] ↔ A[j]
        j = j − 1
        i = i + 1
    SHUFFLE-A-BIT(A[1 ... j])
    SHUFFLE-A-BIT(A[i ... A.length])
```

►Exercise 69 (f09). Answer the following questions. For each question, write "yes" when the answer is always true, "no" when it is always false, "undefined" when it can be true or false. *(10')*

Question 1: Algorithm A solves decision problem X in time $O(n \log n)$. Is X in NP?

Question 2: Is X in P?

Question 3: Decision problem X in P can be polynomially reduced to problem Y. Is there a polynomial-time algorithm to solve Y?

Question 4: Decision problem X can be polynomially reduced to a problem Y for which there is a polynomial-time verification algorithm. Is X in NP?

Question 5: Is X in P?

Question 6: An NP-hard decision problem X can be polynomially reduced to problem Y. Is Y in NP?

Question 7: Is Y NP-hard?

Question 8: Algorithm A solves decision problem X in time $\Theta(2^n)$. Is X in NP?

Question 9: Is X in P?

►Exercise 70 (f09). Write a minimal character-based binary code for the following sentence:

in theory, there is no difference between theory and practice; in practice, there is.

The code must map each character, including spaces and punctuation marks, to a binary string so that the total length of the encoded sentence is minimal. Use a Huffman code and show the derivation of the code. *(20')*

►Exercise 71 (f09). The following matrix represents a directed graph over 12 vertices labeled $a, b, \ldots, \ell$. Rows and columns represent the source and destination of edges, respectively. For example, the value 1 in row a and column f indicates an edge from a to f.

	a	b	c	d	e	f	g	h	i	j	k	ℓ
a						1						
b								1	1	1		
c									1		1	
d	1		1		1							1
e							1			1	1	
f					1					1	1	1
g		1										
h		1		1					1	1		1
i								1				
j		1					1	1				
k	1							1		1		
ℓ									1		1	

Run a *breadth-first search* on the graph starting from vertex a. Using the table below, write the two vectors π (previous) and d (distance) at each main iteration of the BFS algorithm. Write the pair π, d in each cell; for each iteration, write only the values that change. Also, write the complete BFS tree after the termination of the algorithm. *(20')*

a	b	c	d	e	f	g	h	i	j	k	ℓ
$a, 0$	$-, \infty$	$-, \infty$	$-, \infty$	$-, \infty$	$-, \infty$	$-, \infty$	$-, \infty$	$-, \infty$	$-, \infty$	$-, \infty$	$-, \infty$

▸**Exercise 72 (f09).** A *graph coloring* associates a color with each vertex of a graph so that adjacent vertices have different colors. Write a greedy algorithm that tries to color a given graph with the least number of colors. This is a well known and difficult problem for which, most likely, there is no perfect greedy strategy. So, you should use a *reasonable* strategy, and it is okay if your algorithm does not return the absolute best coloring. The result must be a *color* array, where $v.color$ is a number representing the color of vertex v. Write the algorithm, analyze its complexity, and also show an example in which the algorithm does not achieve the best possible result. *(20')*

▸**Exercise 73 (f09).** Given an array A and a positive integer k, the *selection* problem amounts to finding the largest element $x \in A$ such that at most k elements of A are less than or equal to x, or NIL if no such element exists. A simple way to implement it is as follows:

```
SimpleSelection(A, k)
if k > A.length
      return NIL
else sort A in ascending order
      return A[k]
```

Write another algorithm that solves the selection problem without first sorting A. (**Hint:** use a divide-and-conquer strategy that "divides" A using one of its elements.) Also, illustrate the execution of the algorithm on the following input by writing its state at each main iteration or recursion.

$$A = \langle 29, 28, 35, 20, 9, 33, 8, 9, 11, 6, 21, 28, 18, 36, 1 \rangle \qquad k = 6$$

(20')

▸**Exercise 74 (f09).** Consider the following *maximum-value contiguous subsequence* problem: given a sequence of numbers $A = \langle a_1, a_2, \ldots, a_n \rangle$, find two positions i and j, with $1 \le i \le j \le n$, such that the sum $a_i + a_{i+1} + \cdots + a_j$ is maximal.

Question 1: Write an algorithm to solve the problem and analyze its complexity. *(10')*

Question 2: If you have not already done so for question 1, write an algorithm that solves the maximum-value contiguous subsequence problem in time $O(n)$. (**Hint:** one such algorithm uses dynamic-programming.) *(20')*

▸**Exercise 75 (m10).** Consider the following intuitive definition of the *size* of a binary search (sub)tree t: $size(t) = 0$ if t is NIL, or $size(t) = 1 + size(t.left) + size(t.right)$ otherwise. For each node t in a tree, let attribute $t.size$ denote the size of the subtree rooted at t.

Question 1: Prove that, if for each node t in a tree T, $\max\{size(t.left), size(t.right)\} \le \frac{2}{3} size(t)$, then the height of T is $O(\log n)$, where $n = size(T)$. *(10')*

Question 2: Write the rotation procedures ROTATE-LEFT(t) and ROTATE-RIGHT(t) that return the left- and right rotation of tree t maintaining the correct *size* attributes. *(10')*

Question 3: Write an algorithm called SELECTION(T, i) that, given a tree T where each node t carries its size in $t.size$, returns the i-th key in T. *(10')*

Question 4: A tree T is *perfectly balanced* when $\max\{size(t.left), size(t.right)\} = \lfloor size(t)/2 \rfloor$ for all nodes $t \in T$. Write an algorithm called BALANCE(T) that, using the rotation procedures defined in question 2, balances T perfectly. (**Hint:** the essential operation is to move the median value of a subtree to the root of that subtree.) *(30')*

►**Exercise 76 (m10).** Write the *heap-sort* algorithm and illustrate its execution on the following sequence.

$$A = \langle 1, 1, 24, 8, 3, 36, 34, 23, 4, 30 \rangle$$

Assuming the sequence A is stored in an array passed to the algorithm, for each main iteration (or recursion) of the algorithm, write the content of the array. *(10')*

►**Exercise 77 (m10).** A radix tree is used to represent a dictionary of words defined over the alphabet of the 26 letters of the English language. Assume that letters from A to Z are represented as numbers from 1 to 26. For each node x of the tree, $x.links$ is the array of links to other nodes, and $x.value$ is a Boolean value that is true when x represents a word in the dictionary. Write an algorithm PRINT-RADIX-TREE(T) that outputs all the words in the dictionary rooted at T. *(10')*

►**Exercise 78 (m10).** Consider the following algorithm that takes an array A of length $A.length$:

```
ALGO-X(A)
for i = 3 to A.length
    for j = 2 to i − 1
        for k = 1 to j − 1
            if |A[i] − A[j]| == |A[j] − A[k]|
            or |A[i] − A[k]| == |A[k] − A[j]|
            or |A[k] − A[i]| == |A[i] − A[j]|
                return TRUE
return FALSE
```

Write an algorithm BETTER-ALGO-X(A) equivalent to ALGO-X(A) (for all A) but with a strictly better asymptotic complexity than ALGO-X(A). *(20')*

►**Exercise 79 (m10).** For each of the following statements, write whether it is correct or not. Justify your answer by briefly arguing why it is correct, or otherwise by giving a counter example. *(10')*

Question 1: If $f(n) = O(g^2(n))$ then $f(n) = \Omega(g(n))$.

Question 2: If $f(n) = \Theta(2^n)$ then $f(n) = \Theta(3^n)$.

Question 3: If $f(n) = O(n^3)$ then $\log(f(n)) = O(\log n)$.

Question 4: $f(n) = \Theta(f(2n))$

Question 5: $f(2n) = \Omega(f(n))$

►**Exercise 80 (m10).** Write an algorithm PARTITION(A, k) that, given an array A of numbers and a value k, changes A in-place by only swapping two of its elements at a time so that all elements that are less then or equal to k precede all other elements. *(10')*

►**Exercise 81 (f10).** Consider an initially empty B-Tree with minimum degree $t = 2$.

Question 1: Draw the tree after the insertion of keys 81, 56, 16, 31, 50, 71, 58, 83, 0, and 60 in this order. *(10')*

Question 2: Can a different insertion order produce a different tree? If so, write the same set of keys in a different order and the corresponding B-Tree. If not, explain why. *(10')*

►**Exercise 82 (f10).** Consider the following decision problem. Given a set of integers A, output 1 if some of the numbers in A add up to a multiple of 10, or 0 otherwise.

Question 1: Is this problem in NP? If it is, then write a corresponding verification algorithm. If not, explain why not. *(5')*

Question 2: Is this problem in P? If it is, then write a polynomial-time solution algorithm. Otherwise, argue why not. (*Hint:* consider the input values modulo 10. That is, for each input value, consider the remainder of its division by 10.) *(15')*

►**Exercise 83 (f10).** The following greedy algorithm is intended to find the shortest path between vertices u and v in a graph $G = (V, E, w)$, where $w(x, y)$ is the length of edge $(x, y) \in E$.

```
GREEDY-SHORTEST-PATH(G = (V, E, w), u, v)
Visited = {u}                          // this is a set
path = ⟨u⟩                             // this is a sequence
while path not empty
      x = last vertex in path
      if x == v
            return path
      y = vertex y ∈ Adj[x] such that y ∉ Visited and w(x, y) is minimal
                                       // y is x's closest neighbor not already visited
      if y == UNDEFINED                // all neighbors of x have already been visited
            path = path − ⟨x⟩          // removes the last element y from path
      else Visited = Visited ∪ {y}
            path = path + ⟨y⟩          // append y to path
return UNDEFINED                       // there is no path between u and v
```

Does this algorithm find the shortest path always, sometimes, or never? If it always works, then explain its correctness by defining a suitable invariant for the main loop, or explain why the greedy choice is correct. If it works sometimes (but not always) show a positive example and a negative example, and briefly explain why the greedy choice does not work. If it is never correct, show an example and briefly explain why the greedy choice does not work. *(20')*

►**Exercise 84 (f10).** Write the quick-sort algorithm as a deterministic in-place algorithm, and then apply it to the array

$$\langle 50, 47, 92, 78, 76, 7, 60, 36, 59, 30, 50, 43\rangle$$

Show the application of the algorithm by writing the content of the array after each main iteration or recursion. *(20')*

►**Exercise 85 (f10).** Consider an undirected graph G of n vertices represented by its adjacency matrix A. Write an algorithm called IS-CYCLIC(A) that, given the adjacency matrix A, returns TRUE if G contains a cycle, or FALSE if G is acyclic. Also, give a precise analysis of the complexity of your algorithm. *(20')*

►**Exercise 86 (f10).** A palindrome is a sequence of characters that is identical when read left-to-right and right-to-left. For example, the word "racecar" is a palindrome, as is the phrase "rats live on no evil star." Write an algorithm called LONGEST-PALINDROME(T) that, given an array of characters T, prints the longest palindrome in T, or any one of them if there are more than one. For example, if T is the text "radar radiations" then your algorithm should output "dar rad". Also, give a precise analysis of the complexity of your algorithm. *(20')*

►**Exercise 87 (r10).** Write an algorithm called OCCURRENCES that, given an array of numbers A, prints all the distinct values in A each followed by its number of occurrences. For example, if $A = \langle 28, 1, 0, 1, 0, 3, 4, 0, 0, 3\rangle$, the algorithm should output the following five lines (here separated by a semicolon) "28 1; 1 2; 0 4; 3 2; 4 1". The algorithm may modify the content of A, but may not use any other memory. Each distinct value must be printed exactly once. Values may be printed in any order. The complexity of the algorithm must be $o(n^2)$, that is, strictly lower than $O(n^2)$. *(20')*

►**Exercise 88 (r10).** The following algorithm takes an array of line segments. Each line segment s is defined by its two end-points $s.a$ and $s.b$, each defined by their Cartesian coordinates $(s.a.x, s.a.y)$ and $(s.b.x, s.b.y)$, respectively, and ordered such that either $s.a.x < s.b.x$ or $s.a.x = s.b.x$ and $s.a.y < s.b.y$. That is, $s.b$ is never to the left of $s.a$, and if $s.a$ and $s.b$ have the same x coordinates, then $s.a$ is below $s.b$.

```
EQUALS(p, q)
   // tests whether p and q are the same point
if p.x == q.x and p.y == q.y
      return TRUE
else return FALSE
```

```
ALGO-X(A)
for i = 1 to A.length
    for j = 1 to A.length
        if EQUALS(A[i].b, A[j].a)
            for k = 1 to A.length
                if EQUALS(A[j].b, A[k].b) and EQUALS(A[i].a, A[k].a)
                    return TRUE
return FALSE
```

Question 1: Analyze the asymptotic complexity of ALGO-X *(10')*

Question 2: Write an algorithm ALGO-Y that does exactly what ALGO-X does but with a better asymptotic complexity. Also, write the asymptotic complexity of ALGO-Y. *(20')*

►**Exercise 89 (r10).** Write an algorithm called TREE-TO-VINE that, given a binary search tree T, returns the same tree changed into a *vine*, that is, a tree containing exactly the same nodes but restructured so that no node has a left child (i.e., the returned tree looks like a linked list). The algorithm must not destroy or create nodes or use any additional memory other than what is already in the tree, and therefore must operate through a sequence of *rotations*. Write explicitly all the rotation algorithms used in TREE-TO-VINE. Also, analyze the complexity of TREE-TO-VINE. *(15')*

►**Exercise 90 (r10).** We say that a binary tree T is *perfectly balanced* if, for each node n in T, the number of keys in the left and right subtrees of n differ at most by 1. Write an algorithm called IS-PERFECTLY-BALANCED that, given a binary tree T returns TRUE if T is perfectly balanced, and FALSE otherwise. Also, analyze the complexity of IS-PERFECTLY-BALANCED. *(15')*

►**Exercise 91 (r10).** Two graphs G and H are *isomorphic* if there exists a *bijection* $f : V(G) \to V(H)$ between the vertexes of G and H (i.e., a one-to-one correspondence) such that any two vertices u and v in G are adjacent (in G) if and only if $f(u)$ and $f(v)$ are adjacent in H. The *graph-isomorphism* problem is the problem of deciding whether two given graphs are isomorphic.

Question 1: Is graph isomorphism in NP? If so, explain why and write a verification procedure. If not, argue why not. *(10')*

Question 2: Consider the following algorithm to solve the graph-isomorphism problem:

```
ISOMORPHIC(G, H)
if |V(G)| ≠ |V(H)|
    return FALSE
A = V(G) sorted by degree    // A is a sequence of the vertices of G
B = V(H) sorted by degree    // B is a sequence of the vertices of H
for i = 1 to |V(G)|
    if degree(A[i]) ≠ degree(B[i])
        return FALSE
return TRUE
```

Is ISOMORPHIC correct? If so, explain at a high level what the algorithm does and informally but precisely why it works. If not, show a counter-example. *(10')*

►**Exercise 92 (r10).** Write an algorithm HEAP-PRINT-IN-ORDER(H) that takes a min heap H containing unique elements (no element appears twice in H) and prints the elements of H in increasing order. The algorithm must not modify H and may only use a constant amount of additional memory. Also, analyze the complexity of HEAP-PRINT-IN-ORDER. *(20')*

►**Exercise 93 (m11).** Write an algorithm BST-RANGE-WEIGHT(T, a, b) that takes a well balanced binary search tree T (or more specifically the root T of such a tree) and two keys a and b, with $a \le b$, and returns the number of keys in T that are between a and b. Assuming there are $o(n)$ such keys, then the algorithm should have a complexity of $o(n)$, that is, strictly better than linear in the size of the tree. Analyze the complexity of BST-RANGE-WEIGHT. *(10')*

►**Exercise 94 (m11).** Let (a, b) represent an interval (or range) of values x such that $a \le x \le b$. Consider an array $X = \langle a_1, b_1, a_2, b_2, \ldots, a_n, b_n \rangle$ of $2n$ numbers representing n intervals (a_i, b_i), where $a_i = X[2i-1]$ and $b_i = X[2i]$ and $a_i \leq b_i$. Write an algorithm called SIMPLIFY-INTERVALS(X) that takes an array X representing n intervals, and simplifies X in-place. The "simplification" of a set of intervals X is a minimal set of intervals representing the *union* of all the intervals in X. Notice that the union of two disjoint intervals can not be simplified, but the union of two partially overlapping intervals can be simplified into a single interval. For example, a correct solution for the simplification of $X = \langle 3, 7, 1, 5, 10, 12, 6, 8 \rangle$ is $X = \langle 10, 12, 1, 8 \rangle$. An array X can be shrunk by setting its length (effectively removing elements at the end of the array). In this example, $X.length$ should be 4 after the execution of the simplification algorithm. Analyze the complexity of SIMPLIFY-INTERVALS. *(30')*

►**Exercise 95 (m11).** Write an algorithm SIMPLIFY-INTERVALS-FAST(X) that solves Exercise 94 with a complexity of $O(n \log n)$. *(20')*

►**Exercise 96 (m11).** Consider the following algorithm:

```
ALGO-X(A, k)
i = 1
while i ≤ A.length
      if A[i] == k
            ALGO-Y(A, i)
      else i = i + 1
```

```
ALGO-Y(A, i)
while i < A.length
      A[i] = A[i + 1]
      i = i + 1
A.length = A.length − 1 // discards last element
```

Analyze the complexity of ALGO-X and write an algorithm called BETTER-ALGO-X that does exactly the same thing, but with a strictly better asymptotic complexity. Analyze the complexity of BETTER-ALGO-X. *(20')*

►**Exercise 97 (m11).** Write an in-place partition algorithm called MODULO-PARTITION(A) that takes an array A of n numbers and changes A in such a way that (1) the final content of A is a permutation of the initial content of A, and (2) all the values that are equivalent to $0 \mod 10$ precede all the values equivalent to $1 \mod 10$, which precede all the values equivalent to $2 \mod 10$, etc. Being an in-place algorithm, MODULO-PARTITION must not allocate more than a constant amount of memory. For example, for an input array $A = \langle 7, 62, 5, 57, 12, 39, 5, 8, 16, 48 \rangle$, a correct result would be $A = \langle 12, 62, 5, 5, 16, 57, 7, 8, 48, 39 \rangle$. Analyze the complexity of MODULO-PARTITION. *(30')*

►**Exercise 98 (m11).** Write the *merge sort* algorithm and analyze its complexity. *(10')*

►**Exercise 99 (f11).** Write an algorithm called LONGEST-REPEATED-SUBSTRING(T) that takes a string T representing some text, and finds the longest string that occurs at least twice in T. The algorithm returns three numbers $begin_1$, end_1, and $begin_2$, where $begin_1 \leq end_1$ represent the first and last position of the *longest* substring of T that also occurs starting at another position $begin_2 \neq begin_1$ in T. If no such substring exist, then the algorithm returns "None." Analyze the time and space complexity of your algorithm. *(20')*

►**Exercise 100 (f11).** Answer the following questions on complexity theory. Recall that SAT is the Boolean satisfiability problem, which is a well-known NP-complete problem.

Question 1: A decision problem Q is polynomially-reducible to SAT. Can we say for sure that Q is NP-complete? *(2')*

Question 2: SAT is polynomially-reducible to a decision problem Q. Can we say for sure that Q is NP-complete? *(2')*

Question 3: A decision problem Q is polynomially reducible to a problem Q' and Q' is polynomially reducible to SAT. Can we say for sure that Q is in NP? *(2')*

Question 4: An algorithm A solves every instance of a decision problem Q of size n in $O(n^3)$ time. Also, Q is polynomially reducible to another problem Q'. Can we say for sure that Q' is in NP? *(2')*

Question 5: A decision problem Q is polynomially reducible to another decision problem Q', and an algorithm A solves Q' with complexity $O(n \log n)$. Can we say for sure that Q is in NP? *(2')*

Question 6: Consider the following decision problem Q: given a graph G, output 1 if G is connected (i.e., there exists a path between each pair of vertices) or 0 otherwise. Is Q in P? If so, outline an algorithm that proves it, if not argue why not. *(10')*

Question 7: Consider the following decision problem Q: given a graph G and an integer k, output 1 if G contains a cycle of size k. Is Q in NP? If so, outline an algorithm that proves it, if not argue why not. *(10')*

►**Exercise 101 (f11).** Consider an initially empty B-tree with minimum degree $t = 3$. Draw the B-tree after the insertion of the keys 84, 13, 36, 91, 98, 14, 81, 95, 12, 63, 31, and then after the additional insertion of the keys 65, 62, 187, 188, 57, 127, 6, 195, 25. *(10')*

►**Exercise 102 (f11).** Write an algorithm B-TREE-RANGE(T, k_1, k_2) that takes a B-tree T and two keys $k_1 \leq k_2$, and prints all the keys in T between k_1 and k_2 (inclusive). *(20')*

►**Exercise 103 (f11).** Write an algorithm called FIND-TRIANGLE(G) that takes a graph represented by its *adjacency list* G and returns true if G contains a triangle. A triangle in a graph G is a triple of vertices u, v, w such that all three edges (u, v), (v, w), and (u, w) are in G. Analyze the complexity of FIND-TRIANGLE. *(15')*

►**Exercise 104 (f11).** Write an algorithm MIN-HEAP-INSERT(H, k) that inserts a key k in a min-heap H. Also, illustrate the algorithm by writing the content of the array H after the insertion of keys 84, 13, 36, 91, 98, 14, 81, 95, 12, 63, 31, and then after the additional insertion of the key 15. *(15')*

►**Exercise 105 (m12).** Implement a priority queue by writing two algorithms:

- ENQUEUE(Q, x, p) enqueues an object x with priority p, and
- DEQUEUE(Q) extracts and returns an object from the queue.

The behavior of ENQUEUE and DEQUEUE is such that, if a call ENQUEUE(Q, x_1, p_1) is followed (not necessarily immediately) by another call ENQUEUE(Q, x_2, p_2), then x_1 is dequeued before x_2 unless $p_2 > p_1$. Implement ENQUEUE and DEQUEUE such that their complexity is $o(n)$ for a queue of n elements (i.e., strictly less than linear). *(20')*

►**Exercise 106 (m12).** Write an algorithm called MAX-HEAP-MERGE-NEW(H_1, H_2) that takes two max-heaps H_1 and H_2, and returns a new max-heap that contains all the elements of H_1 and H_2. MAX-HEAP-MERGE-NEW must create a *new* max heap, therefore it must allocate a new heap H and somehow copy all the elements from H_1 and H_2 into H without modifying H_1 and H_2. Also, analyze the complexity of MAX-HEAP-MERGE-NEW. *(20')*

►**Exercise 107 (m12).** Write an algorithm called BST-MERGE-INPLACE(T_1, T_2) that takes two binary-search trees T_1 and T_2, and returns a new binary-search tree by merging all the elements of T_1 and T_2. BST-MERGE-INPLACE is *in-place* in the sense that it must rearrange the nodes of T_1 and T_2 in a single binary-search tree without creating any new node. Also, analyze the complexity of BST-MERGE-INPLACE. *(20')*

►**Exercise 108 (m12).** Let A be an array of points in the 2D Euclidean space, each with its Cartesian coordinates $A[i].x$ and $A[i].y$. Write an algorithm MINIMUM-BOUNDING-RECTANGLE(A) that, given an array A of n points, in $O(n)$ time returns the smallest axis-aligned rectangle that contains all the points in A. MINIMUM-BOUNDING-RECTANGLE must return a pair of points corresponding to the bottom-left and top-right corners of the rectangle, respectively. *(10')*

►**Exercise 109 (m12).** Let A be an array of points in the 2D Euclidean space, each with its Cartesian coordinates $A[i].x$ and $A[i].y$. Write an algorithm LARGEST-CLUSTER(A, ℓ) that, given an array A of points and a length ℓ, returns the maximum number of points in A that are contained in a square of size ℓ. Also, analyze the complexity of LARGEST-CLUSTER. *(30')*

►**Exercise 110 (m12).** Consider the following algorithm that takes an array of numbers:

```
ALGO-X(A)
i = 1
j = 1
m = 0
c = 0
while i ≤ |A|
      if A[i] == A[j]
            c = c + 1
      j = j + 1
      if j > |A|
            if c > m
                  m = c
            c = 0
            i = i + 1
            j = i
return m
```

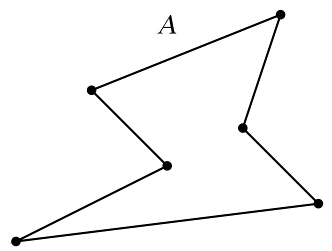
Question 1: Analyze the complexity of ALGO-X. *(5')*

Question 2: Write an algorithm that does exactly the same thing as ALGO-X but with a strictly better asymptotic time complexity. *(15')*

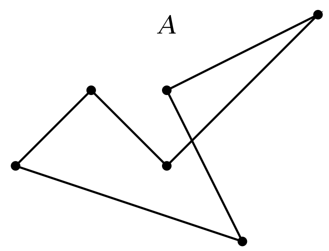
►**Exercise 111 (f12).** Write a THREE-WAY-MERGE(A, B, C) algorithm that merges three sorted sequences into a single sorted sequence, and use it to implement a THREE-WAY-MERGE-SORT(L) algorithm. Also, analyze the complexity of THREE-WAY-MERGE-SORT. *(20')*

►**Exercise 112 (f12).** Write an algorithm IS-SIMPLE-POLYGON(A) that takes a sequence A of 2D points, where each point $A[i]$ is defined by its Cartesian coordinates $A[i].x$ and $A[i].y$, and returns TRUE if A defines a simple polygon, or FALSE otherwise. Also, analyze the complexity of IS-SIMPLE-POLYGON. A polygon is *simple* if its line segments do not intersect.

Example:



IS-SIMPLE-POLYGON(A) = TRUE

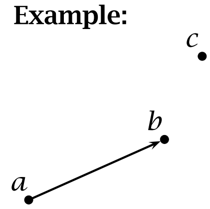


IS-SIMPLE-POLYGON(A) = FALSE

Hint: Use the following DIRECTION-ABC algorithm to determine whether a point c is *on the left side*, *collinear*, or *on the right side* of a segment ab:

```
DIRECTION-ABC(a, b, c)
d = (b.x − a.x)(c.y − a.y) − (b.y − a.y)(c.x − a.x)
if d > 0
      return LEFT
elseif d == 0
      return CO-LINEAR
else return RIGHT
```

Example:



DIRECTION-ABC(a, b, c) = LEFT

(20')

►**Exercise 113 (f12).** Implement a dictionary that supports *longest prefix matching*. Specifically, write the following algorithms:

- BUILD-DICTIONARY(W) takes a list W of n strings and builds the dictionary.
- LONGEST-PREFIX(k) takes a string k and returns the longest prefix of k found in the dictionary, or NULL if none exists. The time complexity of LONGEST-PREFIX(k) must be $o(n)$, that is, sublinear in the size n of the dictionary.

For example, assuming the dictionary was built with strings, “luna”, “lunatic”, “a”, “al”, “algo”, “an”, “anto”, then if k is “algorithms”, then Longest-Prefix(k) should return “algo”, or if k is “anarchy” then Longest-Prefix(k) should return “an”, or if k is “lugano” then Longest-Prefix(k) should return NULL. *(20')*

►**Exercise 114 (f12).** Consider the following decision problem: given a set S of character strings, with characters of a fixed alphabet (e.g., the Roman alphabet), and given an integer k, return TRUE if there are at least k strings in S that have a common substring.

Question 1: Is the problem in NP? Write an algorithm that proves it is, or argue the opposite. *(5')*

Question 2: Is the problem in P? Write an algorithm that proves it is, or argue the opposite. *(15')*

►**Exercise 115 (f12).** Draw a red-black tree containing the following set of keys, clearly indicating the color of each node.

$$\{8, 7, 7, 35, 23, 35, 13, 7, 23, 18, 3, 19, 22\}$$

(10')

►**Exercise 116 (f12).** Consider the following algorithm Algo-X that takes an array A of n numbers:

```
Algo-X(A)
  return Algo-XR(A, 0, 1, 2)

Algo-XR(A, t, i, r)
  while i ≤ A.length
      if r == 0
          if A[i] == t
              return TRUE
      else if Algo-XR(A, t − A[i], i + 1, r − 1)
              return TRUE
      i = i + 1
  return FALSE
```

Analyze the complexity of Algo-X and then write an algorithm Better-Algo-X that does exactly the same thing but with a strictly better time complexity. *(30')*

►**Exercise 117 (r12).** A Eulerian cycle in a graph is a cycle that goes through each edge exactly once. As it turns out, a graph contains a Eulerian cycle if (1) it is connected, and (2) all its vertexes have even degree. Write an algorithm Eulerian(G) that takes a graph G represented as an adjacency matrix, and returns TRUE when G contains a Eulerian cycle. *(10')*

►**Exercise 118 (r12).** Consider a social network system that, for each user u, stores u's friends in a list *friends*(u). Implement an algorithm Top-Three-Friends-Of-Friends(u) that, given a user u, recommends the three other users that are not already among u's friends but are among the friends of most of u's friends. Also, analyze the complexity of the Top-Three-Friends-Of-Friends algorithm. *(20')*

►**Exercise 119 (r12).** Consider the following algorithm:

```
Algo-X(A)
  for i = 3 to A.length
      for j = 2 to i − 1
          for k = 1 to j − 1
              x = A[i]
              y = A[j]
              z = A[k]
              if x > y
                  swap x ↔ y
              if y > z
                  swap y ↔ z
                  if x > y
                      swap x ↔ y
              if y − x == z − y
                  return TRUE
  return FALSE
```

Analyze the complexity of ALGO-X and write an algorithm called BETTER-ALGO-X(A) that does the same as ALGO-X(A) but with a strictly better asymptotic time complexity and with the same space complexity. *(20')*

►**Exercise 120 (r12).** The weather service stores the daily temperature measurements for each city as vectors of real numbers.

Question 1: Write an algorithm called HOT-DAYS(A, t) that takes an array A of daily temperature measurements for a city and a temperature t, and returns the maximum number of consecutive days with a recorded temperature above t. Also, analyze the complexity of HOT-DAYS(A, t). *(5')*

Question 2: Now imagine that a particular analysis would call the HOT-DAYS algorithm several times with the same series A of temperature measurements (but with different temperature values) and therefore it would be more efficient to somehow index or precompute the results. To do that, write the following two algorithms:

- A preprocessing algorithm called HOT-DAYS-INIT(A) that takes the series of temperature measurements A and creates an auxiliary data structure X (an index of some sort).
- An algorithm called HOT-DAYS-FAST(X, t) that takes the index X and a temperature t and returns the maximum number of consecutive days with a temperature above t. HOT-DAYS-FAST must run in *sub-linear time* in the size of A.

Also, analyze the complexity of HOT-DAYS-INIT and HOT-DAYS-FAST. *(25')*

►**Exercise 121 (r12).** Consider the following decision problem: given a sequence A of numbers and given an integer k, return TRUE if A contains either an increasing or a decreasing subsequence of length k. The elements of the subsequence must maintain their order in A but do not have to be contiguous.

Question 1: Is the problem in NP? Write an algorithm that proves it is, or argue the opposite. *(10')*

Question 2: Is the problem in P? Write an algorithm that proves it is, or argue the opposite. *(20')*

►**Exercise 122 (r12).** Write an algorithm HEAP-DELETE(H, i) that, given a max-heap H, deletes the element at position i from H. *(10')*

►**Exercise 123 (m13).** Write an algorithm MAX-CLUSTER(A, d) that takes an array A of numbers (not necessarily integers) and a number d, and prints a maximal set of numbers in A that differ by at most d. The output can be given in any order. Your algorithm must have a complexity that is strictly better than $O(n^2)$. For example, with

$$A = \langle 7, 15, 16, 3, 10, 43, 8, 1, 29, 13, 4.5, 28 \rangle \quad d = 5$$

MAX-CLUSTER(A, d) would output $7, 3, 4.5, 8$ (or the same numbers in any other order) since those numbers differ by at most 5 and there is no larger set of numbers in A that differ by at most 5. Also, analyze the complexity of MAX-CLUSTER. *(20')*

►**Exercise 124 (m13).** Consider the following algorithm that takes a non-empty array of numbers

```
Algo-X(A)
B = make a copy of A
i = 1
while i ≤ B.length
      j = i + 1
      while j ≤ B.length
            if B[j] == B[i]
                  i = i + 1
                  swap B[i] ↔ B[j]
            j = j + 1
      i = i + 1
q = B[1]
n = 1
m = 1
for i = 2 to B.length
      if B[i] == q
            n = n + 1
            if n > m
                  m = m + 1
      else q = B[i]
            n = 1
return m
```

Question 1: Briefly explain what Algo-X does, and analyze the complexity of Algo-X. *(10')*

Question 2: Write an algorithm called Better-Algo-X that is functionally identical to Algo-X but with a strictly better complexity. Analyze the complexity of Better-Algo-X. *(10')*

►**Exercise 125 (m13).** Write the *heap-sort* algorithm and then illustrate how *heap-sort* processes the following array in-place:

$$A = \langle 33, 28, 23, 48, 32, 46, 40, 12, 21, 41, 14, 37, 38, 0, 25 \rangle$$

In particular, show the content of the array at each main iteration of the algorithm. *(20')*

►**Exercise 126 (m13).** Write an algorithm BST-Print-Longest-Path(T) that, given a binary search tree T, outputs the sequence of nodes (values) of the path from the root to any node of maximal depth. Also, analyze the complexity of BST-Print-Longest-Path. *(30')*

►**Exercise 127 (m13).** Consider insertion in a binary search tree.

Question 1: Write a valid insertion algorithm BST-Insert. *(10')*

Question 2: Illustrate how BST-Insert works by drawing the binary search tree resulting from the insertion of the following keys in this order:

$$33, 28, 23, 48, 32, 46, 40, 12, 21, 41, 14, 37, 38, 0, 25$$

Also, if the resulting tree is not already of minimal depth, write an alternative insertion order that would result in a tree of minimal depth. *(10')*

Question 3: Write an algorithm Best-BST-Insert-Order(A) that takes an array of numbers A and outputs the elements of A in an order that, if used with BST-Insert would lead to a binary search tree of minimal depth. *(10')*

►**Exercise 128 (f13).** Write an algorithm called Find-Negative-Cycle that, given a weighted directed graph $G = (V, E)$, with weight function $w : E \to \mathbb{R}$, finds and outputs a negative-weight cycle in G if one such cycle exists. Also, analyze the complexity of Find-Negative-Cycle. *(20')*

►**Exercise 129 (f13).** Consider a text composed of n lines of up to 80 characters each. The text is stored in an array T where each line $T[i]$ is an array of characters containing words separated by a single space.

Question 1: Write an algorithm SORT-LINES-BY-WORD-COUNT(T) that, with a worst-case complexity of $O(n)$, sorts the lines in T in non-decreasing order of the number of words in the line. (**Hint:** lines have at most 80 characters, so the number of words in a line is also limited.) *(20')*

Question 2: If you did not already do that for exercise 1, write an *in-place* variant of the SORT-LINES-BY-WORD-COUNT algorithm. This algorithm, called SORT-LINES-BY-WORD-COUNT-IN-PLACE, must also have a $O(n)$ complexity to sort the set of lines, and may use only a constant amount of extra space to do that. *(20')*

►**Exercise 130 (f13).** Consider a weighted undirected graph $G = (V, E)$ representing a group of programmers and their affinity for team work, such that the weight $w(e)$ of an edge $e = (u, v)$ is a number representing the ability of programmers u and v to work together on the same project. Write an algorithm BEST-TEAM-OF-THREE that outputs the best team of three programmers. The value of a team is considered to be the lowest affinity level between any two members of the team. So, the best team is the group of programmers for which the lowest affinity level between members of the group is maximal. *(20')*

►**Exercise 131 (f13).** Write an algorithm MAXIMAL-NON-ADJACENT-SUM(A) that, given a sequence of numbers $A = \langle a_1, a_2, ..., a_n \rangle$, computes, with worst-case complexity $O(n)$, the maximal sum of non-adjacent elements in A. A subsequence of non-adjacent elements may include a_i or a_{i+1} but not both, for all i. For example, with $A = \langle 2, 9, 6, 2, 6, 8, 5 \rangle$, MAXIMAL-NON-ADJACENT-SUM(A) should return 20. (**Hint:** use a dynamic programming algorithm that scans the input once.) *(20')*

►**Exercise 132 (f13).** Consider a trie rooted at node T that represents a set of character strings. For simplicity, assume that characters are from the Roman alphabet and that the letters of the alphabet are encoded with numeric values between 1 and 26. Write an algorithm PRINT-TRIE(T) that prints all the strings stored in the trie. *(20')*

►**Exercise 133 (r13).** Write an algorithm PRINT-IN-THREE-COLUMNS(A) that takes an array of words A and prints all the words in A, in the given order left-to-right and top-to-bottom, such that the words are left-aligned in three columns. Words must be separated by at least one space horizontally, but in order to align words, the algorithm might have to print more spaces between words. For example, if A contains the words *exam, algorithms, asymptotic, complexity, graph, greedy, lugano, np, quicksort, retake, september,* then the output should be

```
exam       algorithms asymptotic
complexity graph      greedy
lugano     np         quicksort
retake     september
```

(20')

►**Exercise 134 (r13).** Consider a binary search tree.

Question 1: Write an algorithm BST-MEDIAN(T) that takes the root T of a binary search tree and returns the median element contained in the tree. Also analyze the complexity of BST-MEDIAN(T). Can you do better? *(10')*

Question 2: Assume now that the tree is balanced and also that each node t has an attribute $t.weight$ corresponding to the total number of nodes in the subtree rooted at t (including t itself). Write an algorithm BETTER-BST-MEDIAN(T) that improves on the complexity of BST-MEDIAN. Analyze the complexity of BETTER-BST-MEDIAN. *(10')*

►**Exercise 135 (r13).** Consider the following decision problem. Given a set of strings S, a number w, and a number k, output *YES* when there are at least k strings in S that share a common substring of length w, or *NO* otherwise. For example, if S contains the strings *exam, algorithms, asymptotic, complexity, graph, greedy, lugano, np, quicksort, retake, september, theory, practice, programming, math, art, truth, justice,* with $w = 2$ and $k = 3$ the output should be *YES*, because the 3 strings *graph, greedy,* and *programming* share a common substring "gr" of length 2. The output should also be *YES* for $w = 3$ and $k = 3$ and for $w = 2$ and $k = 4$, but it should be *NO* for $w = 3$ and $k = 4$.

Question 1: Is this problem in NP? Write an algorithm that proves it is, or argue that it is not. *(10')*

Question 2: Is this problem in P? Write an algorithm that proves it is, or argue that it is not. (**Hint:** a string of length ℓ has $O(\ell^2)$ sub-strings of any length.) *(20')*

▸**Exercise 136 (r13).** Consider the following sorting problem: you must reorder the elements of an array of numbers in-place so that odd numbers are in odd positions while even numbers are in even positions. If there are more even elements than odd ones in A (or vice-versa) then those additional elements will be grouped at the end of the array. For example, with an initial sequence

$$A = \langle 50, 47, 92, 78, 76, 7, 60, 36, 59, 30, 50, 43\rangle$$

the result could be this:

$$A = \langle 47, 50, 7, 78, 59, 76, 43, 92, 36, 60, 30, 50\rangle$$

Question 1: Write an algorithm called ALTERNATE-EVEN-ODD(A) that sorts A in place as explained above. Also, analyze the complexity of ALTERNATE-EVEN-ODD. (You might want to consider question 2 before you start solving this problem.) *(20')*

Question 2: If you have not done so already, write a variant of ALTERNATE-EVEN-ODD that runs in $O(n)$ steps for an array A of n elements. *(10')*

▸**Exercise 137 (r13).** Write an algorithm called FOUR-CYCLE(G) that takes a directed graph represented with its adjacency matrix G, and that returns *true* if and only if G contains a 4-cycle. A 4-cycle is a sequence of four distinct vertexes a, b, c, d such that there is an arc from a to b, from b to c, from c to d, and from d to a. Also, analyze the complexity of FOUR-CYCLE(G). *(20')*

▸**Exercise 138 (m14).** Write an algorithm FIND-EQUAL-DISTANCE(A) that takes an array A of numbers, and returns four distinct elements a, b, c, d of A such that $a - b = c - d$, or NIL if no such elements exist. FIND-EQUAL-DISTANCE must run in $O(n^2 \log n)$ time. *(20')*

▸**Exercise 139 (m14).** Consider the following algorithm that takes an array of numbers:

```
ALGO-X(A)
i = 1
while i < A.length
    if A[i] > A[i + 1]
        swap A[i] ↔ A[i + 1]
    p = i
    q = i + 1
    for j = i + 2 to A.length
        if A[j] < A[p]
            p = j
        else if A[j] > A[q]
                q = j
    swap A[i] ↔ A[p]
    swap A[i + 1] ↔ A[q]
    i = i + 2
```

Question 1: Explain what ALGO-X does and analyze its complexity. *(5')*

Question 2: Write an algorithm BETTER-ALGO-X that is functionally equivalent to ALGO-X but with a strictly better time complexity. *(15')*

▸**Exercise 140 (m14).** Consider the following definition of the height of a node t in a binary tree:

$$height(t) = \begin{cases} 0 & \text{if } t == \text{NIL} \\ 1 + \max\{height(t.left), height(t.right)\} & \text{otherwise.} \end{cases}$$

Question 1: Write an algorithm HEIGHT(t) that computes the height of a node t. Also, analyze the complexity of your HEIGHT algorithm when t is the root of a tree of n nodes. *(5')*

Question 2: Consider now a binary search tree in which each node t has an attribute $t.height$ that denotes the height of that node. Write a constant-time rotation algorithm LEFT-ROTATE(t) that performs a left rotation around node t and also updates the *height* attributes as needed. *(5')*

►Exercise 141 (m14). Consider the following classic insertion algorithm for a binary search tree:

```
BST-INSERT(t, k)
if t == NIL
    return NEW-NODE(k)
else if k ≤ t.key
        t.left = BST-INSERT(t.left, k)
    else t.right = BST-INSERT(t.right, k)
return t
```

Write an algorithm SORT-FOR-BALANCED-BST(A) that takes an array of numbers A, and prints the elements of A so that, if passed to BST-INSERT, the resulting BST would be of minimal height. Also, analyze the complexity of your solution. *(20')*

►Exercise 142 (m14). Consider the array of numbers:

$$A = \langle 69, 36, 68, 18, 36, 36, 50, 9, 36, 36, 18, 18, 8, 10 \rangle$$

Question 1: Does A satisfy the *max-heap* property? If not, fix it by swapping two elements. *(5')*

Question 2: Write an algorithm MAX-HEAP-INSERT(H, k) that inserts a key k in a max-heap H. *(10')*

Question 3: Illustrate the behavior of MAX-HEAP-INSERT by applying it to array A (possibly corrected). In particular, write the content of the array after the insertion of each of the following keys, in this order: $69, 50, 60, 70$. *(5')*

►Exercise 143 (m14). Consider the following algorithm that takes an array of numbers:

```
ALGO-Y(A)
a = 0
for i = 1 to A.length - 1
    for j = i + 1 to A.length
        x = 0
        for k = i to j
            if A[k] is even:
                x = x + 1
            else x = x - 1
        if x == 0 and j - i > a
            a = j - i
return a
```

Question 1: Explain what ALGO-Y does and analyze its complexity. *(5')*

Question 2: Write an algorithm BETTER-ALGO-Y that is functionally equivalent to ALGO-Y but with a strictly better time complexity. Also analyze the time complexity of BETTER-ALGO-Y. *(10')*

Question 3: If you have not already done so for question 2, write a BETTER-ALGO-Y that is functionally equivalent to ALGO-Y but that runs in time $O(n)$. *(15')*

►Exercise 144 (f14). Write an algorithm THREE-WAY-PARTITION(A, v) that takes an array A of n numbers, and partitions A *in-place* in three parts, some of which might be empty, so that the left part $A[1 \dots p-1]$ contains all the elements less than v, the middle part $A[p \dots q-1]$ contains all the elements equal to v, and the right part $A[q \dots n]$ contains all the elements greater than v. THREE-WAY-PARTITION must return the positions p and q and must run in time $O(n)$. *(20')*

►Exercise 145 (f14). A DNA strand is a sequence of nucleotides, and can be represented as a string over the alphabet $\Sigma = \{\mathsf{A}, \mathsf{C}, \mathsf{G}, \mathsf{T}\}$. Consider the problem of determining whether two DNA strands s_1 and s_2 are k-related in the sense that they share a sub-sequence of at least k nucleotides.

Question 1: Is the problem in NP? Write an algorithm that proves it is, or argue that it is not. *(10')*

Question 2: Is the problem in P? Write an algorithm that proves it is, or argue that it is not. *(20')*

►**Exercise 146 (f14).** Consider the following algorithm that takes an array of numbers:

```
ALGO-X(A)
y = −∞
i = 1
j = 1
x = 0
while i ≤ A.length
    x = x + A[j]
    if x > y
        y = x
    if j == A.length
        i = i + 1
        j = i
        x = 0
    else j = j + 1
return y
```

Question 1: Explain what ALGO-X does and analyze its complexity. *(10')*

Question 2: Write an algorithm BETTER-ALGO-X that is functionally equivalent to ALGO-X but with a strictly better time complexity. *(20')*

►**Exercise 147 (f14).** Write an algorithm MAXIMAL-CONNECTED-SUBGRAPH(G) that takes an undirected graph $G = (V, E)$ and prints the vertices of a maximal connected subgraph of G. *(20')*

►**Exercise 148 (f14).** A system collects the positions of cars along a highway that connects two cities, A and B. The positions are grouped by direction in two arrays, A and B. Thus A contains the distances in kilometers from city A of the cars traveling towards city A. Write an algorithm CONGESTION(A) that takes the array A and prints a list of congested sections of the highway. A congested interval is a contiguous stretch of highway of 1km or more in which the density of cars is more than 50 cars per kilometer. CONGESTION(A) must run in $O(n \log n)$ time. *(20')*

►**Exercise 149 (r14).** The following matrix represents a directed graph over vertices $a, b, c, \ldots, \ell$. Rows and columns represent the source and destination of edges, respectively.

	a	b	c	d	e	f	g	h	i	j	k	ℓ
a			1									
b	1		1								1	
c				1	1							
d	1						1	1				
e				1		1					1	
f				1			1					
g												
h									1	1		
i												
j								1	1			
k												1
ℓ											1	

Write the graph and the *DFS numbering* of the vertexes using the DFS algorithm. Every iteration through vertexes or adjacent edges is performed in alphabetic order. (*Hint:* the DFS numbering of a vertex v is a pair of numbers representing the "time" at which DFS discovers v and the time DFS leaves v.) *(20')*

►**Exercise 150 (r14).** Consider an array A of n numbers that is initially sorted, in ascending order, and then modified so that k of its elements are decreased in value.

Question 1: Write an algorithm that sorts A *in-place* in time $O(kn)$. *(10')*

Question 2: Write an algorithm that sorts A in time $O(n + k \log k)$ but not necessarily in-place. *(20')*

►**Exercise 151 (r14).** Consider the decision version of the well-known *vertex cover* problem: given a graph $G = (V, E)$ and an integer k, output 1 if G contains a vertex cover of size k. A vertex cover is a set of vertexes $S \subseteq V$ such that, for each edge $(u, v) \in E$, either vertex u is in S or vertex v is in S. Write an algorithm that proves that vertex cover is in NP. *(20')*

►**Exercise 152 (r14).** Write an algorithm that transforms a min-heap H into a max-heap *in-place*. *(10')*

►**Exercise 153 (r14).** We say that two words x and y are *linked* to each other if they differ by a single letter, or more specifically by one edit operation, meaning an insertion, a deletion, or a change in a single character. For example, "fun" and "pun" are linked, as are "flower" and "lower", "port" and "post", "canton" and "cannon", and "cat" and "cast".

Question 1: Write an algorithm LINKED(x, y) that takes two words x and y and, in linear time, returns TRUE if x and y are linked to each other, or FALSE otherwise. *(10')*

Question 2: Write an algorithm WORD-CHAIN(W, x, y) that takes an array of words W and two words x and y, and outputs a minimal sequence of words $x, w_1, w_2, \ldots, y$ that starts with x and ends with y where $w_1, w_2, \ldots$ are all words from W, and each word in the sequence is linked to the words adjacent to it. For example, if W is a dictionary of English words, and x and y are "first" and "last", respectively, then the output might be: *first fist list last.* *(30')*

►**Exercise 154 (m15).** Write an algorithm MAX-HEAP-INSERT(H, k) that inserts a new value k in a max-heap H. Briefly analyze the complexity of your solution. *(10')*

►**Exercise 155 (m15).** Consider an algorithm FIND-ELEMENTS-AT-DISTANCE(A, k) that takes an array A of n integers sorted in non decreasing order and returns TRUE if and only if A contains two elements a_i and a_j such that $a_i - a_j = k$.

Question 1: Write a version of the FIND-ELEMENTS-AT-DISTANCE algorithm that runs in $O(n \log n)$ time. Briefly analyze the complexity of your solution. *(10')*

Question 2: Write a version of the FIND-ELEMENTS-AT-DISTANCE algorithm that runs in $O(n)$ time. Briefly analyze the complexity of your solution. *(20')*

►**Exercise 156 (m15).** Write an algorithm PARTITION-PRIMES-COMPOSITES(A) that takes an array A of n integers such that $1 < A[i] \leq m$ for all i, and partitions A in-place so that all primes precede all composites in A. Analyze the complexity of your solution as a function of n and m. Recall that an integer greater than 1 is *prime* if it is divisible by only two positive integers (itself and 1) or otherwise it is *composite.* *(20')*

►**Exercise 157 (m15).** Consider the following classic insertion algorithm for a binary search tree:

```
BST-INSERT(t, k)
if t == NIL
    return NEW-NODE(k)
else if k ≤ t.key
        t.left = BST-INSERT(t.left, k)
    else t.right = BST-INSERT(t.right, k)
return t
```

Write an algorithm SORT-FOR-BALANCED-BST(A) that takes an array of numbers A, and prints the elements of A in a new order so that, if the printed sequence is passed to BST-INSERT, the resulting BST would be of minimal height. Also, analyze the complexity of your solution. *(20')*

►**Exercise 158 (m15).** Consider a game in which, given a multiset of positive numbers A (possibly with repeated values) a player can simplify A by removing, one at a time, an element a_k if there are two other elements a_i, a_j such that $a_i + a_j = a_k$.

Question 1: Write an algorithm called MINIMAL-SIMPLIFIED-SUBSET(A) that, given a multiset A of n numbers, returns a minimal simplified subset $X \subseteq A$. The result X is *minimal* in the sense that no smaller set can be obtained with a sequence of simplifications starting from A. For example, with $A = \{7, 89, 11, 88, 106, 4, 28, 71, 17\}$, a valid result would be $X = \{7, 89, 4, 71, 17\}$. Briefly analyze the complexity of your solution. *(10')*

Question 2: Write a MINIMAL-SIMPLIFIED-SUBSET(A) algorithm that runs in $O(n^2)$. If you have already done so for exercise 1, then simply say so. *(20')*

►**Exercise 159 (m15).** Consider the following algorithm that takes an integer n as input:

```
ALGORITHM-X(n)
c = 0
a = n
while a > 1
    b = 1
    while b ≤ a^2
        c = c + 1
        b = 2b
    a = a/2
return c
```

Write the complexity of ALGORITHM-X as a function of n. Justify your answer. *(10')*

►**Exercise 160 (f15).** Write an algorithm FIND-CYCLE(G) that, given a directed graph G, returns TRUE if and only if G contains a cycle. You may assume the representation of your choice for G. *(20')*

►**Exercise 161 (f15).** A breadth-first search over a graph G returns a vector π that represents the resulting breadth-first tree, where the parent $\pi[v]$ of a vertex v is the next-hop from v on the tree towards the source of the breadth-first search.

Question 1: Write an algorithm BFS-FIRST-COMMON-ANCESTOR(π, u, v) that finds the first common ancestor of two given nodes in the breadth-first tree, or NULL if u and v are not connected in G. The complexity of BFS-FIRST-COMMON-ANCESTOR must be $O(n)$. Briefly analyze the space complexity of your solution. *(10')*

Question 2: Write an algorithm BFS-FIRST-COMMON-ANCESTOR-2(π, D, u, v) that is also given the distance vector D resulting from the same breadth first search. BFS-FIRST-COMMON-ANCESTOR-2 must be functionally equivalent to BFS-FIRST-COMMON-ANCESTOR (as defined in Exercise 1) but with space complexity $O(1)$. *(20')*

►**Exercise 162 (f15).** Consider the height and the black height of a red-black tree.

Question 1: What are the minimum and maximum heights of a red-black tree containing 10 keys? Exemplify your answers by drawing a minimal and a maximal tree. Clearly identify each node as red or black. *(10')*

Question 2: What are the minimum and maximum *black* heights of a red-black tree containing 10 keys? Exemplify your answers by drawing a minimal and a maximal tree. Clearly identify each node as red or black. *(10')*

►**Exercise 163 (f15).** Consider an algorithm BST-FIND-SUM(T, v) that, given a binary search tree T containing n distinct numeric keys, and given a target value v, finds and returns two nodes in T whose keys add up to v. The algorithm returns NULL if no such keys exist in T. BST-FIND-SUM may not modify the tree, and may only use a constant amount of memory.

Question 1: Write BST-FIND-SUM. You may use the basic algorithms that operate on binary search trees (BST-MIN, BST-SUCCESSOR, BST-SEARCH, etc.) without defining them explicitly. *(10')*

Question 2: Write a variant of BST-FIND-SUM(T, v) that works in $O(n)$ time. If your solution to Exercise 1 already has this complexity bound, then simply say so. *(20')*

►**Exercise 164 (f15).** Consider this decision problem: given a set of integers $X = \{x_1, x_2, \ldots, x_n\}$, and an integer k, return 1 if there are k elements in X that are pairwise relatively prime, or return 0 otherwise. Two integers are relatively prime if their only common divisor is 1. For example, for $X = \{5, 6, 10, 14, 18, 21, 49\}$ and $k = 3$, the result is 1, since the 3 elements $5, 18, 49$ are pairwise relatively prime (5 and 18 have no common divisor other than 1, and the same is true for 5 and 49, and 18 and 49). However, for the same set $X = \{5, 6, 10, 14, 18, 21, 49\}$ and $k = 4$, the solution is 0, since no four elements from X are all pairwise relatively prime.

Question 1: Is this problem in NP? Write an algorithm that proves it is, or argue that it is not. *(20')*

Question 2: **(BONUS)** Is this problem NP-hard? Prove it. *(60')*

▸**Exercise 165 (r15).** You are given a square matrix $M \in \mathbf{R}^{n\times n}$ whose elements are sorted both row-wise and column-wise. In other words, rows and columns are non-decreasing sequences. Formally, for every element $m_{i,j} \in M$, $(j < n \Rightarrow m_{i,j} \le m_{i,j+1}) \wedge (i < n \Rightarrow m_{i,j} \le m_{i+1,j})$. Write an algorithm SEARCH-IN-SORTED-MATRIX(M, x) that returns TRUE if $x \in M$ or FALSE otherwise. The time complexity of SEARCH-IN-SORTED-MATRIX must be $O(n \log n)$. Justify that your solution has such a complexity. *(20')*

▸**Exercise 166 (r15).** Consider the following algorithm that takes an array A of positive integers:

```
ALGO-X(A)
B = copy of A
i = 1
x = 1
while i ≤ A.length
    B[i] = B[i] − 1
    if B[i] == 0
        B[i] = A[i]
        i = i + 1
    else x = x + 1
        i = 1
return x
```

Question 1: Briefly explain what ALGO-X does and analyze the complexity of ALGO-X. *(10')*

Question 2: Write an algorithm called BETTER-ALGO-X that is functionally identical to ALGO-X but with a strictly better complexity. Analyze the complexity of BETTER-ALGO-X. *(10')*

▸**Exercise 167 (r15).** Consider the following algorithm that takes an array A of numbers:

```
ALGO-Y(A)
i = 2
j = 1
x = −∞
while i ≤ A.length
    if |A[i] − A[j]| > x
        x = |A[i] − A[j]|
    j = j + 1
    if j == i
        i = i + 1
        j = 1
return x
```

Question 1: Briefly explain what ALGO-Y does and analyze the complexity of ALGO-Y. *(10')*

Question 2: Write an algorithm called BETTER-ALGO-Y that is functionally identical to ALGO-Y but with a complexity $O(n)$. *(10')*

▸**Exercise 168 (r15).** Write an algorithm BTREE-LOWER-BOUND(T, k) that, given a B-tree T and a value k, returns the least key v in T such that $k \le v$, or NULL if no such key exist. Also, analyze the complexity of BTREE-LOWER-BOUND. Recall that a node x in a B-tree has the following properties: $x.n$ is the number of keys, $X.key[1] \le x.key[2] \le \dots x.key[x.n]$ are the keys, $x.leaf$ tells whether x is a leaf, and $x.c[1], x.c[2], \dots, x.c[x.n+1]$ are the pointers to x's children. *(20')*

▸**Exercise 169 (r15).** Write an algorithm BST-LEAST-DIFFERENCE(T) that, given a binary search tree T containing numeric keys, returns in $O(n)$ time the minimal distance between any two keys in the tree. *(20')*

▸**Exercise 170 (r15).** A connected component of an undirected graph G is a maximal set of vertices that are connected to each other (directly or indirectly). Thus the vertices of a graph can be partitioned into connected components. Write an algorithm CONNECTED-COMPONENTS(G) that, given an undirected graph G, returns the number of connected components in G. Also, analyze the complexity of CONNECTED-COMPONENTS. *(20')*

►**Exercise 171 (m16).** Rank the following functions in decreasing order of growth by indicating their rank next to the function, as in the first line (n^{n^n} is the fastest growing function). If any two functions f_i and f_j are such that $f_i = \Theta(f_j)$, then rank them at the same level. *(10')*

function	*rank*
$f_0(n) = n^{n^n}$	1
$f_1(n) = \log^2(n)$	
$f_2(n) = n!$	
$f_3(n) = \log(n^2)$	
$f_4(n) = n$	
$f_5(n) = \log(n!)$	
$f_6(n) = \log\log n$	
$f_7(n) = n\log n$	
$f_8(n) = \sqrt{n^3}$	
$f_9(n) = 2^n$	

Hint: as a reminder, consider the following mathematical definitions and facts: (definition of factorial) $n! = 1 \cdot 2 \cdot 3 \cdots (n-1) \cdot n$; (facts about the logarithm) $\log(ab) = \log a + \log b$, and therefore $\log(a^k) = k\log a$.

►**Exercise 172 (m16).** Write an algorithm called MINIMAL-COVERING-SQUARE(P) that takes a sequence P of n points in the 2D Euclidean plane, each defined by its Cartesian coordinates $P[i].x$ and $P[i].y$, and returns the area of a minimal axis-aligned square that covers all points in P. An axis-aligned square is one in which the sides are parallel to X and Y axes. MINIMAL-COVERING-SQUARE must run in time $O(n)$. *(10')*

►**Exercise 173 (m16).** A sequence of numbers is called *unimodal* if it is first strictly increasing and then strictly decreasing. For example, the sequence $1, 5, 19, 17, 12, 8, 5, 3, 2$ is unimodal, while the sequence $1, 5, 3, 7, 4, 2$ is not. Write an algorithm UNIMODAL-FIND-MAXIMUM(A) that finds the maximum of a unimodal sequence A of n numbers in time $O(\log n)$. *(20')*

►**Exercise 174 (m16).** Consider the following algorithm ALGO-X(A, k) that takes an array A of n objects and an integer k:

```
ALGO-X(A, k)
l = -∞
r = +∞
for i = 1 to A.length - k
      for j = i + 1 to A.length
            if ALGO-Y(A, i, j) ≥ k
                  if r - l > j - i
                        l = i
                        r = j
return l, r
```

```
ALGO-Y(A, a, b)
m = 1
for i = a to b
      c = 1
      for j = i + 1 to b
            if A[i] == A[j]
                  c = c + 1
      if c > m
            m = c
return m
```

Question 1: Explain what ALGO-X(A, k) does and analyze its complexity. Do not simply paraphrase the code. Instead, explain the high level semantics, independent of the code. *(10')*

Question 2: Write an algorithm BETTER-ALGO-X(A, k) with the same functionality as ALGO-X(A, k), but with a strictly better complexity. Also, analyze the complexity of BETTER-ALGO-X(A, k). *(20')*

►**Exercise 175 (m16).** An algorithm THREE-WAY-PARTITION(A, *begin*, *end*) chooses a pivot element from the sub-array $A[\mathit{begin} \ldots \mathit{end} - 1]$, and partitions that sub-array in-place into three parts (two of which might be empty): $A[begin \ldots q_1 - 1]$ containing all the elements less than the pivot, $A[q_1 \ldots q_2 - 1]$ containing all the elements equal to the pivot, and $A[q_2 \ldots end - 1]$ containing all elements greater than the pivot.

Question 1: Write a THREE-WAY-PARTITION(A, *begin*, *end*) algorithm that runs in time $O(n)$, where $n = end - begin$, and that returns the partition boundaries q_1, q_2. You may assume that $begin < end$. *(20')*

Question 2: Use the THREE-WAY-PARTITION algorithm to write a better variant of the classic quick-sort algorithm. Also, describe in which cases this variant would perform significantly better than the classic algorithm. *(10')*

►**Exercise 176 (m16).** The following algorithm SUM(A, s) takes an array A of n numbers and a number s. Describe what SUM(A, s) does at a high level and analyze its complexity in the best and worst cases. Justify your answer by clearly describing the best- and worst-case input, as well as the behavior of the algorithm in each case. *(20')*

```
SUM(A, s)
return SUM-R(A, s, 1, A.length)
```

```
SUM-R(A, s, b, e)
if b > e and s == 0
      return TRUE
elseif b ≤ e and SUM-R(A, s, b + 1, e)
      return TRUE
elseif b ≤ e and SUM-R(A, s − A[b], b + 1, e)
      return TRUE
else return FALSE
```

►**Exercise 177 (f16).** Big Brother tracks a set of m cell-phone users by recording every cell antenna the user connects to. In particular, for each user u_i, Big Brother stores a time-ordered sequence $S_i = (t_1, a_1), (t_2, a_2), \ldots$ that records that user u_i was connected to antenna a_1 starting at time t_1, and later switched to antenna a_2 at time $t_2 > t_1$, and so on. Write an algorithm called GROUP-OF-K($S_1, S_2, \ldots, S_m, k$) that finds whether there is a time t^* when a group of at least k users are connected to the same antenna. In this case, GROUP-OF-K must output the time t^* and the antenna a^*. Otherwise, GROUP-OF-K must output NULL. GROUP-OF-K must run in time $O(n \log m)$ where n is the total number of entries in all the sequences, so $n = |S_1| + |S_2| + \cdots + |S_m|$. You may use common data structures and algorithms without specifying those algorithms completely. *(20')*

►**Exercise 178 (f16).** Consider the following algorithm that takes an array A of integers:

```
ALGO-X(A)
i = 1
j = A.length + 1
while i < j
      if A[i] ≡ 0 mod 2
      // A[i] is even
            j = j − 1
            v = A[i]
            ALGO-Y(A, i, j)
            A[j] = v
      else i = i + 1
return j
```

```
ALGO-Y(A, p, q)
while p < q
      A[p] = A[p + 1]
      p = p + 1
```

Question 1: Briefly explain what ALGO-X does and analyze the complexity of ALGO-X. *(10')*

Question 2: Write an algorithm BETTER-ALGO-X that is functionally identical to ALGO-X but with a strictly better complexity. Also briefly analyze the complexity of BETTER-ALGO-X. *(10')*

►**Exercise 179 (f16).** Write an algorithm BTREE-PRINT-RANGE(T, a, b) that, given a B-tree T and two values $a < b$, prints all the keys k in T that are between a and b, that is, $a < k < b$. Recall that a node x in a B-tree has the following properties: $x.n$ is the number of keys, $x.key[1] \le x.key[2] \le \ldots x.key[x.n]$ are the keys, $x.leaf$ tells whether x is a leaf, and $x.c[1], x.c[2], \ldots, x.c[x.n+1]$ are the pointers to x's children. *(20')*

►**Exercise 180 (f16).** Consider the following decision problem: given a weighted graph G and a number k, where $w(e)$ is the weight of an edge $e = (u, v) \in E(G)$, return TRUE if and only if there are at least two nodes u and v at distance $d(u, v) = k$. Is the problem in NP? Write an algorithm that proves it is, or argue the opposite. Is the problem in P? Write an algorithm that proves it is, or argue the opposite. Recall that the distance $d(u, v)$ in a graph is the minimal length of any path connecting u and v. *(20')*

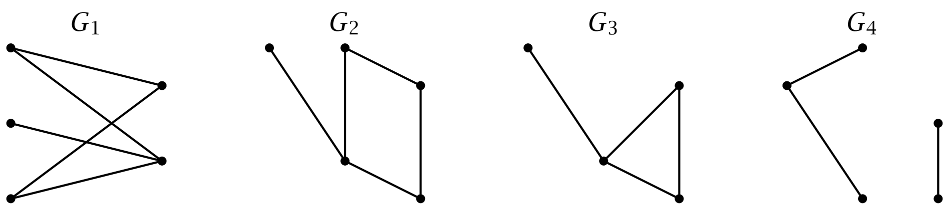
►**Exercise 181 (f16).** A highway traffic app sends the coordinates of each vehicle to a server that reports on congested sections of highway. Consider the highway as a straight line in which each position is identified by a single x coordinate. Write an algorithm MOST-CONGESTED-SEGMENT(A, ℓ) that, given an array A of vehicle positions and a length ℓ, outputs the position of a maximally congested highway segment of length at most ℓ. A segment of highway between positions x and $x + \ell$ is considered maximally congested if there are no other segments of length at most ℓ with more vehicles. Coordinates as well as the length ℓ are real numbers, not necessarily integers; ℓ is positive (it is a distance). *(20')*

►**Exercise 182 (f16).** Consider the following decision problem: given a graph G represented as an adjacency matrix G, and an integer k, return TRUE if and only if there are at least k nodes $v_1, v_2, \ldots, v_k$ in G that form a fully connected sub-graph of G, meaning that for every pair $i, j \in 1, \ldots, k$, edge (v_i, v_j) is in G. Is the problem in NP? Write an algorithm that proves it is, or argue the opposite. *(20')*

►**Exercise 183 (r16).** Write an algorithm MAX-HEAP-TOP-THREE(H) that takes a heap H and prints the three highest values stored in the heap. The algorithm must run in $O(1)$ time, may not allocate more than a constant amount of memory, and may not modify the heap in any way. If the heap contains less than three values, then MAX-HEAP-TOP-THREE must print whatever elements exist. *(20')*

►**Exercise 184 (r16).** Let P be a sequence of points representing an alpine road where, for each point $p \in P$, $p.x$ is the distance from the beginning of the road and $p.y$ is the elevation (meters above sea level). Write an algorithm LONGEST-STRETCH(P, h) that takes a sequence of points P and an altitude range (difference) h, and returns the maximal length of a stretch of road that remains within an altitude range of at most h. For example, if $h = 0$, the algorithm must return the maximal length of road that is absolutely flat (that is, contiguous points at the same elevation). Analyze the complexity of your solutions showing a worst-case input. *(20')*

►**Exercise 185 (r16).** An undirected graph G is *bipartite* when its vertices can be partitioned into two sets V_A, V_B such that each edge in G connects a vertex in V_A with a vertex in V_B. In other words, no two vertices in V_A are adjacent, and no two vertices in V_B are adjacent. To exemplify, see the graphs below.



bipartite | *bipartite* (same as G_1) | *not bipartite* | *bipartite*

Write an algorithm IS-BIPARTITE(G) that takes an undirected graph G and outputs TRUE if and only if G is bipartite. (**Hint:** you may use a simple BFS in which you keep track of which vertex is in which partition.) *(20')*

►**Exercise 186 (r16).** Algorithm IS-GOOD(x) classifies a number x as "good" or "not good" in constant time $O(1)$.

Question 1: Write an algorithm GOOD-ARE-ADJACENT(A) that takes a sequence of numbers and, using algorithm IS-GOOD, returns TRUE if all the "good" numbers in A are adjacent, or FALSE otherwise. GOOD-ARE-ADJACENT(A) must not change the input sequence A in any way, may allocate only a constant amount of memory, and must run in time $O(n)$. *(10')*

Question 2: Write an algorithm MAKE-GOOD-ADJACENT(A) that takes a sequence of numbers A and changes A in-place so that all "good" numbers are adjacent. MAKE-GOOD-ADJACENT may allocate only a constant amount of memory and must run in time $O(n)$. *(10')*

►**Exercise 187 (r16).** Consider the following decision problem: given a sequence of numbers A and an integer k, returns TRUE if A contains at least k identical values, or FALSE otherwise. Is the problem in NP? Write an algorithm that proves it is, or argue the opposite. Is the problem in P? Write an algorithm that proves it is, or argue the opposite. *(20')*

►**Exercise 188 (r16).** Write an algorithm MAXIMAL-COMMON-SUBSTRING(X, Y) that, given strings X and Y, returns the maximal length of a common substring of X and Y. For example, with X = "BDDBADCDCCDCBAD" and Y = "DDCBCDAABAAC", the output should be 3, since there is a 3-character common substring ("DCB") but no 4-character common substring. Analyze the complexity of your solution. *(20')*

►**Exercise 189 (m17).** We say that a node in a binary tree is *unbalanced* when the number of nodes in its left subtree is more than twice the number of nodes in its right subtree plus one, or vice-versa. Write an algorithm BST-COUNT-UNBALANCED-NODES(t) that takes a binary search tree t (the root), and returns the number of unbalanced nodes in the tree. Analyze the complexity of BST-COUNT-UNBALANCED-NODES(t). (**Hint:** an algorithm can return multiple values. For example, the statement **return** x, y returns a pair of values, and if F() returns a pair of values, you can read them with a, b = F().) *(20')*

►**Exercise 190 (m17).** Consider the following algorithm that takes an array A of numbers:

```
ALGO-X(A)
x = 0
for i = 1 to A.length − 1
    for j = i + 1 to A.length
        if ALGO-Y(A, i, j) and A[j] − A[i] > x
            x = A[j] − A[i]
return x
```

```
ALGO-Y(A, i, j)
for k = i to j − 1
    if A[k] > A[k + 1]
        return FALSE
return TRUE
```

Question 1: Briefly explain what ALGO-X does and analyze the complexity of ALGO-X by describing a worst-case input. *(10')*

Question 2: Write an algorithm LINEAR-ALGO-X(A) that is equivalent to ALGO-X but runs in linear time. *(20')*

►**Exercise 191 (m17).** Let P be an array of points on a plane, each with its Cartesian coordinates $P[i].x$ and $P[i].y$.

Question 1: Write an algorithm FIND-SQUARE(P) that returns TRUE if and only if there are four points in P that form a square. Briefly analyze the complexity of your solution. *(10')*

Question 2: Write an algorithm FIND-SQUARE(P) that solves the problem of Exercise 1 in time $O(n^2 \log n)$. If your solution for Exercise 1 already does that, then simply say so. *(20')*

►**Exercise 192 (m17).** Implement a priority queue based on a heap. You must implement the following algorithms:

- INITIALIZE(Q) creates an empty queue. The complexity of INITIALIZE must be $O(1)$.
- ENQUEUE(Q, obj, p) adds an object *obj* with priority p to a queue Q. The complexity of ENQUEUE must be $O(\log n)$.
- DEQUEUE(Q) extracts and returns an object from a queue Q. The returned object must be among the objects in the queue that were inserted with the lowest priority. The complexity of DEQUEUE must be $O(\log n)$.

(**Hint:** Consider Q as an object to which you can add attributes. For example, you may write $Q.A$ = new array, and then later write $Q.A[i]$.) *(20')*

▶**Exercise 193 (m17).** Implement an algorithm MAXIMAL-DISTANCE(A) that takes an array A of numbers and returns the maximal distance between any two distinct elements in A, or 0 if A contains less than two elements. MAXIMAL-DISTANCE(A) must run in time $O(n)$. *(10')*

▶**Exercise 194 (m17).** The *height* of a binary tree is the maximal number of nodes on a branch from the root to a leaf node. In other words, it is the maximal number of nodes traversed by a simple path starting at the root. Implement an algorithm BST-HEIGHT(t) that returns the height of a binary search tree rooted at node t. BST-HEIGHT(t) must run in time $O(n)$. *(10')*

▶**Exercise 195 (f17).** Consider the following decision problem: given a graph $G = (V, E)$ where the edges are weighted by a weight function $w : E \to \mathbb{R}$, and given a number t, output *true* if there is a set of non-adjacent edges $S = \{e_1, e_2, \ldots, e_k\}$ of total weight greater or equal to t, so $\sum w(e_i) \geq t$; or output *false* otherwise. For example, the vertices could represent people, say the students in the Algorithms class, and an edge $e = (u, v)$ with weight $w(e)$ could represent the affinity of the couple (u, v). The question is then, given an affinity value t, tell whether the students in the Algorithms class can form monogamous couples of total affinity value at least t. Argue whether this decision problem is in NP or not, and if it is, then write an algorithm that proves it. *(20')*

▶**Exercise 196 (f17).** Consider the following game: you are given a set of n valuable objects placed on a 2D plane with non-negative x, y coordinates. In practice, you are given three arrays X, Y, V, such that $X[i]$, $Y[i]$, and $V[i]$ are the x and y coordinates and the *value* of object i, respectively. You start from position 0, 0, and can only move horizontally to the right (increasing your x coordinate) or vertically upward (increasing your y coordinate). Your goal is to reach and collect valuable objects. Write an algorithm MAXIMAL-GAME-VALUE(X, Y, V) that returns the maximal total value you can achieve in a given game. *(30')*

▶**Exercise 197 (f17).** Write an algorithm MAXIMAL-SUBSTRING(S) that takes an array S of strings, and returns a string x of maximal length such that x is a substring of every string $S[i]$. Also, analyze the complexity of MAXIMAL-SUBSTRING as a function of the size $n = |S|$ of the input array, and the maximal size m of any string in S. *(20')*

▶**Exercise 198 (f17).** Consider the following algorithm that takes an array A of numbers:

```
ALGO-X(A)
x = 0
y = 0
for i = 1 to A.length
    k = 1
    for j = i + 1 to A.length
        if A[i] == A[j]
            k = k + 1
    if x < k
        x = k
        y = A[i]
return y
```

Question 1: Briefly explain what ALGO-X does and analyze the complexity of ALGO-X by describing a worst-case input. *(10')*

Question 2: Write an algorithm BETTER-ALGO-X that does the same as ALGO-X but with a strictly better time complexity. Also analyze the complexity of BETTER-ALGO-X. *(10')*

▶**Exercise 199 (f17).** Write an algorithm GRAPH-DEGREE(G) that takes an undirected graph represented by its adjacency matrix G and computes the *degree* of G. The degree of a graph is the maximal degree of any vertex of G. The degree of a vertex v is the number of edges that are adjacent to v. Also analyze the complexity of GRAPH-DEGREE(G). *(15')*

▶**Exercise 200 (f17).** Write an algorithm FIND-3-CYCLE(G) that takes an undirected graph represented as an adjacency list, and returns TRUE if G contains a cycle of length 3, or FALSE otherwise. Also, analyze the complexity of FIND-3-CYCLE(G). *(15')*

►**Exercise 201 (r17).** Write an algorithm Longest-Common-Prefix(S) that takes an array of strings S, and returns the maximal length of a string that is a prefix of at least two strings in S. Also, analyze the complexity of your solution as a function of the size n of the input array S, and the maximal size m of any string in S. For example, with S = ["ciao", "lugano", "bella"] the result is 0, because the only common prefix is the empty string, while with S = ["professor", "prefers", "to", "teach","programming"] the result is 3 because "pro" is a prefix of at least two strings. *(20')*

►**Exercise 202 (r17).** Write an algorithm Longest-K-Common-Prefix(S, k) that takes an array of strings S and an integer k, and returns the maximal length of a string that is a prefix of at least k strings in S. Also, analyze the complexity of your solution as a function of k, the size n of the input array S, and the maximal size m of any string in S. For example, with S = ["algorithms", "and", "data", "structures"] and $k = 3$, the result is 0, because the only common prefix common to at least three strings is the empty string. While with S = ["professor", "prefers", "to", "teach","programming"] and $k = 3$, the result is 2 because the longest prefix common to at least three strings is "pr". *(20')*

►**Exercise 203 (r17).** Consider the following decision problem: given a directed and weighted graph G (with weighted arcs), output TRUE if and only if G contains a path of length 3 and of negative total weight; otherwise output FALSE. Is the problem in NP? Write an algorithm that proves it is, or argue the opposite. Is the problem in P? Write an algorithm that proves it is, or argue the opposite. *(20')*

►**Exercise 204 (r17).** Given a collection A of numbers and a number x, the *upper bound* of x in A is the minimal value $a \in A$ such that $x \leq a$, or NULL if no such value exists. For example, given $A = [7, 20, 1, 3, 4, 3, 31, 50, 9, 11]$, the upper bound of $x = 15$ is 20, while the upper bound of $x = 9$ is 9 and the upper bound of $x = 51$ is NULL.

Question 1: Write an algorithm Upper-Bound(A, x) that returns the upper bound of x in an array A. Also analyze the complexity of Upper-Bound. *(20')*

Question 2: Write an algorithm Upper-Bound-Sorted(A, x) that returns the upper bound of x in a sorted array A in time $o(n)$. Analyze the complexity of Upper-Bound-Sorted. *(20')*

Question 3: Write an algorithm Upper-Bound-BST(T, x) that returns the upper bound of x in a binary search tree T. Analyze the complexity of Upper-Bound-BST. *(20')*

►**Exercise 205 (m18).** Write an algorithm Sum-Of-Three(A, s) that takes an array A of n numbers and a number s, and in $O(n^2)$ time decides whether A contains three distinct elements that add up to s. That is, Sum-Of-Three(A, s) returns TRUE if there are three indexes $1 \leq i < j < k \leq n$ such that $A[i] + A[j] + A[k] = s$, or FALSE otherwise. Analyze the complexity of your solution and briefly explain the algorithm by commenting on its non-obvious parts. *(20')*

►**Exercise 206 (m18).** The following algorithm takes an array A of numbers, and a number x:

```
Algo-X(A, x)
i = A.length
j = 1
while i > 0
    if j == i
        j = 1
        i = i − 1
    elseif A[i] − A[j] > x or A[j] − A[i] > x
        return TRUE
    else j = j + 1
return FALSE
```

Question 1: Briefly explain what Algo-X does and analyze the complexity of Algo-X by describing a worst-case input. *(10')*

Question 2: Write an algorithm Better-Algo-X(A, x) that is functionally equivalent to Algo-X but with a strictly better time complexity. Analyze the complexity of your solution and briefly explain the algorithm by commenting on its non-obvious parts. *(10')*

►**Exercise 207 (m18).** Consider the following algorithm that takes an array A of numbers, and an integer k:

```
ALGO-S(A, k)
for i = 1 to A.length
    if ALGO-R(A, A[i]) == k
        return A[i]
return NULL
```

```
ALGO-R(A, y)
c = 0
for i = 1 to A.length
    if A[i] < y
        c = c + 1
return c
```

Question 1: Briefly explain what ALGO-S does and analyze the complexity of ALGO-S by describing a worst-case input. *(10')*

Question 2: Write an algorithm BETTER-ALGO-S(A, k) that is functionally equivalent to ALGO-S(A, k) but with a strictly better complexity. Analyze the complexity of your solution and briefly explain the algorithm by commenting on its non-obvious parts. *(10')*

►**Exercise 208 (m18).** An array A of n numbers contains only four values, possibly repeated many times. Write an algorithm SORT-SPECIAL(A) that sorts A in-place and in time $O(n)$. Analyze the complexity of your solution and briefly explain the algorithm by commenting on its non-obvious parts. *(20')*

►**Exercise 209 (m18).** Write an algorithm HEAP-PROPERTIES(A) that takes an array A of n numbers and in $O(n)$ time returns one of four values: -1, if A satisfies the min-heap property; 1, if A satisfies the max-heap property; 2, if A satisfies both the max-heap and min-heap properties; 0, if A does not satisfy either the max-heap or min-heap properties. Analyze the complexity of your solution and briefly explain the algorithm by commenting on its non-obvious parts. *(20')*

►**Exercise 210 (m18).** You are given a constant-time decision algorithm COMPATIBLE(x, y) that, given two objects x and y tells whether x and y are compatible. The relation expressed by the COMPATIBLE algorithm is *symmetric*, meaning that COMPATIBLE(x, y) implies COMPATIBLE(y, x), and *transitive*, meaning that COMPATIBLE(x, y) and COMPATIBLE(y, z) imply COMPATIBLE(x, z). In other words, it is an *equivalence* relation.
Write an algorithm MAX-COMPATIBLE-PAIRING(A) that takes an array of n objects, and in $O(n^2)$ time, returns the maximum number of compatible pairs that can be formed from the objects in A. A compatible pair is a pair of distinct compatible elements, that is, a pair of indexes $1 \le i < j \le n$ such that COMPATIBLE$(A[i], A[j])$ == TRUE. Each element (index) may appear in only one pair. Analyze the complexity of your solution and briefly explain the algorithm by commenting on its non-obvious parts. *(20')*

►**Exercise 211 (f18).** Consider an infinite chessboard in which the rows and columns are numbered with corresponding integers in their natural order $(\ldots -3, -2, -1, 0, 1, 2, 3, \ldots)$. You are given two arrays W and B of positions of white and black queens, respectively, such that $W[i].row$ and $W[i].col$ are the row and column of the i-th white queen, and correspondingly $B[i].row$ and $B[i].col$ are the row and column of the i-th black queen.
Write an algorithm WHITE-ATTACKS-BLACK(W, B) that takes the two arrays of white and black queens, and returns TRUE if and only if there is a white queen that attacks a black queen. The complexity of your solution must be $o(n^2)$, meaning strictly less than quadratic. (Recall that a queen in row i and column j attacks all positions in row i, all positions in column j, and all positions in the two 45-degree diagonals that pass through the square in row i and column j.) *(20')*

►**Exercise 212 (f18).** We say that a node in a binary search tree is *full* if it has both a left and a right child.

Question 1: Write an algorithm called COUNT-FULL-NODES(t) that takes a binary search tree rooted at node t, and returns the number of full nodes in the tree. Analyze the complexity of your solution. *(10')*

Question 2: Write an algorithm called NO-FULL-NODES(t) that takes a binary search tree rooted at node t, and changes the tree in-place, using only rotations, so that the tree does not contain any full node. Analyze the complexity of your solution. *(20')*

▶**Exercise 213 (f18).** Consider the following decision problem: given two arrays A and B, both containing n numbers, output TRUE if and only if there is a number k and a permutation A' of A such that $A'[i] + B[i] = k$ for all positions $i \in \{1, \ldots, n\}$.

Question 1: Is the problem in NP? Write an algorithm that proves it is, or argue otherwise. *(10')*

Question 2: Is the problem in P? Write an algorithm that proves it is, or argue otherwise. *(20')*

▶**Exercise 214 (f18).** Write an algorithm called MINIMAL-CONTIGUOUS-SUM(A) that takes an array A of numbers, and outputs the value of the minimal contiguous sub-sequence sum in time $O(n)$. A contiguous sub-sequence sum is the sum of some contiguous elements of A. For example, if A is the sequence

$$-1, 2, -2, -4, 1, -2, 5 - 2 - 3, 1, 2, -1$$

then the minimal contiguous sub-sequence sum is -7, which is the sum of elements $-2, -4, 1, -2$. *(20')*

▶**Exercise 215 (f18).** Write an algorithm called HAS-CYCLE(G) that takes a directed graph G represented as an adjacency list, and returns TRUE whenever G contains one or more cycles. You can denote the adjacency list of a vertex v in G as $G.Adj[v]$. Your solution must have a polynomial and possibly linear complexity. Briefly analyze the complexity of your solution. *(20')*

▶**Exercise 216 (r18).** A DNA sequence S is an array of characters (a string) where each character $S[i]$ is one of 'A', 'C', 'G', or 'T'. Write an algorithm DNA-PERMUTATION-SUBSTRING(S, X) that takes a large DNA sequence S and a smaller sequence X, and in linear time returns TRUE if and only if S contains a contiguous subsequence (a substring) that is a permutation of X. For example, DNA-PERMUTATION-SUBSTRING("GCCATCAGTGACGAAGCT", "TAGG") would return TRUE, because the long sequence contains the contiguous subsequence "AGTG", which is a permutation of the sequence "TAGG". *(30')*

▶**Exercise 217 (r18).** Consider the following algorithm that takes a non-empty array A of numbers:

```
ALGO-X(A)
n = A.length
let B be an array of size n
for i = 1 to n
    B[i] = 0
m = 1
x = A[1]
for i = 1 to n
    if B[i] == 0
        B[i] = 1
        for j = i + 1 to n
            if A[i] == A[j]
                B[i] = B[i] + 1
                B[j] = 1
        if m < B[i] or (m == B[i] and x > A[i])
            x = A[i]
            m = B[i]
return x
```

Question 1: Briefly describe what ALGO-X does and analyze the complexity of ALGO-X. *(10')*

Question 2: Write an algorithm called BETTER-ALGO-X that does exactly the same thing, but with a strictly better asymptotic complexity. Analyze the complexity of BETTER-ALGO-X. *(20')*

▶**Exercise 218 (r18).** Consider the problem of comparing two binary search trees.

Question 1: Write an algorithm BST-EQUALS(t_1, t_2) that takes the roots t_1 and t_2 of two binary search trees and returns TRUE if and only if the tree rooted t_1 is exactly the same as the tree rooted at t_2, meaning that the two trees have nodes with the same keys connected in exactly the same way. Also, analyze the complexity of your solution. *(10')*

Question 2: Write an algorithm BST-EQUAL-KEYS(t_1, t_2) that takes the roots t_1 and t_2 of two binary search trees and returns TRUE if and only if the tree rooted t_1 contains exactly the same keys as the tree rooted at t_2. *(20')*

►**Exercise 219 (r18).** Consider an infinite chessboard in which the rows and columns are numbered with corresponding integers in their natural order ($\ldots - 3, -2, -1, 0, 1, 2, 3, \ldots$). Write an algorithm KNIGHT-DISTANCE(r_1, c_1, r_2, c_2) that takes two positions on the chessboard, identified by the respective row and column numbers, and returns the minimal number of hops it would take a knight to go from the first position to the second position. Also, analyze the complexity of your solution. *Hints:* a knight moves in a single hop by two squares horizontally and by one square vertically, or vice-versa. Notice that what matters is the *distance*, not the absolute positions, so consider computing the distance between any position (r, c) and the $(0, 0)$ position. Consider a dynamic-programming solution. Also notice that the problem has symmetries that can greatly simplify the solution. For example, the distance from $(0, 0)$ to position (a, b) is the same as to position (b, a). *(30')*

►**Exercise 220 (r18b).** Consider a directed graph G of 20 vertexes, numbered from 1 to 20, and defined by the following adjacency list

$v \to adj(v)$
$1 \to 2$
$2 \to 8\ 9$
$3 \to 2\ 4\ 5\ 6$
$4 \to 10\ 11\ 12\ 13\ 14\ 15\ 5\ 9$
$5 \to 18\ 7$
$6 \to 5\ 7$
$7 \to 18\ 19\ 4$
$8 \to 9$
$9 \to 10$
$10 \to 11$
$11 \to 12\ 14$
$12 \to 14$
$13 \to 14\ 17\ 20$
$15 \to 13\ 16\ 5$
$16 \to 13\ 17\ 5$
$17 \to 18\ 19$
$18 \to 19$
$20 \to 14\ 17$

(**Hint:** draw the graph and use the drawing to answer the following questions.)

Question 1: Compute a depth-first search on G. Write the three vectors P, D, and F that, for each vertex, hold the *previous vertex* in the depth-first forest, the *discovery time*, and the *finish time*, respectively. Whenever necessary, iterate through vertexes in numeric order. *(20')*

Question 2: Compute a breadth-first search on G starting from vertex 1. Write the two vectors P and D that, for each vertex, hold the *previous vertex* in the breadth-first tree and the *distance*, respectively. Whenever necessary, iterate through vertexes in numeric order. *(20')*

►**Exercise 221 (r18b).** Consider the following decision problem: given a sequence A of numbers and given an integer k, return TRUE if and only if A contains either an increasing or a decreasing subsequence of length k. The elements of the subsequence must maintain their order in A but do not have to be contiguous. For example, $A = [4, 5, 3, 8, 3, 9]$ contains an increasing sequence of length $k = 4$ (4, 5, 8, 9), but neither an increasing or decreasing sequence of length $k = 5$.

Question 1: Is the problem in NP? Write an algorithm that proves it is, or argue the opposite. *(10')*

Question 2: Is the problem in P? Write an algorithm that proves it is, or argue the opposite. *(20')*

►**Exercise 222 (r18b).** Given a sequence of numbers $A = \langle a_1, a_2, \ldots, a_n \rangle$, we define a *maximal contiguous subsequence* as a contiguous subsequence of numbers in A, starting at position i and ending at position j with $1 \le i \le j \le n$, whose sum is maximal.

Question 1: Write an algorithm MCS-VALUE(A) that, given a sequence A, returns the sum of a maximal contiguous subsequence in A. Also, analyze the complexity of your solution. *(10')*

Question 2: Write an algorithm MCS-VALUE-LINEAR(A) that, given a sequence A, returns the sum of a maximal contiguous subsequence in A with $O(n)$ complexity. *(20')*

►**Exercise 223 (r18b).** Analyze the following algorithms that take an array A of integers. First, briefly describe what the algorithm does, and then analyze the best- and worst-case complexity as functions of the size of the input $n = |A|$. Your characterizations should be as tight as possible. Briefly justify your answers.

Question 1: Describe and analyze the following ALGO-X *(10')*

```
ALGO-X(A)
for i = |A| downto 2
    s = TRUE
    for j = 2 to i
        if A[j − 1] > A[j]
            swap A[j − 1] ↔ A[j]
            s = FALSE
    if s == TRUE
        return
```

Question 2: Describe and analyze the following ALGO-Y *(10')*

```
ALGO-Y(A)
i = 1
j = |A|
while i < j
    if A[i] > A[j]
        swap A[i] ↔ A[i + 1]
        if i + 1 < j
            swap A[i] ↔ A[j]
        i = i + 1
    else j = j − 1
```

►**Exercise 224 (m19).** Write an algorithm PARTITION-ZERO(A) that takes an array of numbers A and, in $O(n)$ time, rearranges the elements of A in-place so that all the negative elements of A precede all the elements equal to zero that precede all the positive elements. For example, with an initial array $A = [2, 5, 0, -1, 3, -7, 0, 3, -1, 10]$, a valid (but not unique) result of PARTITION-ZERO(A) would be the permuted array $A = [-1, -7, -1, 0, 0, 2, 5, 3, 3, 10]$. *(20')*

►**Exercise 225 (m19).** Implement a priority queue. Given two objects x and y, you can test whether x has a higher priority than y by testing the condition $x > y$. Briefly describe the data structure (data and meta-data) and then write three algorithms: PQ-INIT(n) creates, initializes, and returns a priority queue Q of maximal size n; PQ-ENQUEUE(Q, x) enqueues an object x into queue Q; PQ-DEQUEUE(Q) extracts and returns an object x such that there is no other object y in Q such that $y > x$. Both PQ-ENQUEUE and PQ-DEQUEUE must have a complexity $O(\log n)$. *(30')*

►**Exercise 226 (m19).** Consider the following algorithm ALGO-X(A, B) that takes two arrays of numbers

```
ALGO-X(A, B)
C = copy of array B
n = C.length
for i = 1 to A.length
    j = 1
    while j ≤ n
        if A[i] == C[j]
            swap C[j] ↔ C[n]
            n = n − 1
        else j = j + 1
if n == 0
    return TRUE
else return FALSE
```

Question 1: Briefly explain what ALGO-X does, and analyze its complexity by also describing a worst-case input. *(10')*

Question 2: Write a an algorithm BETTER-ALGO-X that is functionally identical to ALGO-X but with a strictly better time complexity. *(20')*

►**Exercise 227 (m19).** Consider the following algorithm QUESTIONABLE-SORT(A) that takes an array of numbers A and intends to sort it in-place.

```
QUESTIONABLE-SORT(A)
for i = 1 to A.length − 1
    for j = i + 1 to A.length
        if A[i] > A[j]
            swap A[i] ↔ A[j]
```

Question 1: Is QUESTIONABLE-SORT correct? If so, explain how the algorithm works. If not, show a counter-example. *(10')*

Question 2: Write an algorithm BETTER-SORT that sorts in-place with a strictly better average-case complexity than QUESTIONABLE-SORT. *(10')*

►**Exercise 228 (m19).** Write an algorithm LOWER-BOUND(A, x) that takes a sorted array A of numbers and, in $O(\log n)$ time returns the least (smallest) number a_i in A such that $a_i \geq x$. If no such value exists, LOWER-BOUND(A, x) must return a "not-found" error. *(20')*

►**Exercise 229 (f19).** Write an algorithm CONTAINS-SQUARE(A) that takes an $\ell \times \ell$ matrix A of numbers, and returns TRUE if and only if A contains a square pattern of equal numbers, that is, a set of equal elements $A_{x,y}$ whose positions, interpreted as points with Cartesian coordinates (x, y), lay on the perimeter of a square. A square pattern consists of at least four elements, so a single number is not a valid square pattern. For example, the following matrix contains a square pattern consisting of elements with value 3. Notice in fact that there are two such square patterns.

$$\begin{bmatrix} 7 & 8 & 3 & 8 & 8 & 3 \\ 7 & 8 & 3 & 3 & 3 & 3 \\ 1 & 3 & 3 & 5 & 8 & 3 \\ 7 & 6 & 3 & 5 & 3 & 3 \\ 0 & 4 & 3 & 3 & 3 & 3 \\ 9 & 9 & 1 & 3 & 7 & 3 \end{bmatrix}$$

Also, analyze the complexity of your solution as a function of $n = \ell^2$. *(20')*

►**Exercise 230 (f19).** Write an algorithm MIN-HEAP-CHANGE(H, i, x) that takes a min-heap H of size n, an index i, and a value x, and changes the value $H[i]$ to x, possibly adjusting the heap so as to maintain the min-heap property. MIN-HEAP-CHANGE must run in $O(\log n)$ time. Analyze the complexity of your solution. *(20')*

►**Exercise 231 (f19).** Write an algorithm BST-SUBSET(T_1, T_2) that takes two binary search trees T_1 and T_2 (the roots) and returns TRUE if and only if T_1 contains a subset of the keys in T_2. Your solution must run in time $O(n)$, where n is the total size of the two input trees. Analyze the complexity of your solution. *(20')*

►**Exercise 232 (f19).** Consider the following decision problem: given a graph $G = (V, E)$ and an integer k, return TRUE if G contains a cycle of length k, or otherwise FALSE. Is this problem in NP? Show a proof of your answer. *(20')*

►**Exercise 233 (f19).** Consider the following decision problem: given a graph $G = (V, E)$, return TRUE if G contains a cycle of length 4, or otherwise FALSE. Is this problem in P? Show a proof of your answer. *(20')*

►**Exercise 234 (f19).** Write an algorithm SUMS-ONE-TWO-THREE(n) that takes an integer n and, in time $O(n)$, returns the number of possible ways to write n as a sum of 1, 2, and 3. For example, SUMS-ONE-TWO-THREE(4) must return 7 because there are 7 ways to write 4 as a sum of ones, twos, and threes ($1+1+1+1$, $1+1+2$, $1+2+1$, $2+1+1$, $2+2$, $1+3$, $3+1$). Analyze the complexity of your solution. *Hint:* use dynamic programming. *(20')*

►**Exercise 235 (r19).** Write an algorithm TWO-PRIMES(n) that takes a number n and returns TRUE if and only if n is the sum of two primes. For example, TWO-PRIMES(12) returns TRUE because $12 = 5 + 7$, and 5 and 7 are primes, but TWO-PRIMES(11) returns FALSE, because it can not be expressed as the sum of two primes. Analyze the complexity of your solution as a function of n.
Recall that a prime p is a positive integer that can not be written as the product of two positive integers smaller than p. Thus $2, 3, 5, 7, 11, \ldots$ are primes, but 1 and 4 are not. *(20')*

►**Exercise 236 (r19).** Consider the following algorithm ALGO-X(A) that takes a non-empty array A of objects each with two numeric attributes: *weight* and *category*.

```
ALGO-X(A)
c = A[1].category
w = -∞
for i = 1 to A.length
    t = 0
    for j = 1 to A.length
        if A[j].category == A[i].category
            t = t + A[j].weight
    if t > w or (t == w and c > A[i].category)
        c = A[i].category
        w = t
return c
```

Question 1: Describe at a high-level what ALGO-X does, and analyze its complexity. *(10')*

Question 2: Write an algorithm BETTER-ALGO-X that is functionally equivalent to ALGO-X but with a strictly better time complexity. *(20')*

►**Exercise 237 (r19).** Consider a min-heap represented internally as an array H with an additional attribute $H.heap\text{-}size$ representing the number of elements in the heap.

Question 1: Write an algorithm MIN-HEAP-INSERT(H, x) that takes a valid min-heap H and inserts a new value x in H. Analyze the complexity of your solution. *(10')*

Question 2: Write an algorithm MIN-HEAP-DEPTH(H) that computes the depth of a given min-heap in $O(\log n)$ time. *(10')*

►**Exercise 238 (r19).** Consider the following algorithm ALGO-Y(A) that takes an array A of numbers.

```
ALGO-Y(A)
m = −∞
for i = 1 to A.length − 1
    for j = i + 1 to A.length
        if A[i] + A[j] > m
            m = A[i] + A[j]
return m
```

Question 1: Describe at a high-level what ALGO-Y does and analyze its complexity. *(10')*

Question 2: Write an algorithm BETTER-ALGO-Y that is functionally equivalent to ALGO-Y and that runs in $O(n)$ time. *(20')*

►**Exercise 239 (r19).** Consider the following decision problem: given an array A of numbers, a number m, and an integer k, output TRUE if A contains k distinct elements $A[i_1], A[i_2], \ldots, A[i_k]$ such that $A[i_1] + A[i_2] + \cdots + A[i_k] \geq m$, or FALSE otherwise. Is this problem in P? Show a proof of your answer. *(20')*

►**Exercise 240 (m20).** Given a number k, a step-k sequence of length ℓ is a sequence of ℓ numbers $a_1, a_2, \ldots, a_\ell$ such that either $a_i = a_{i+1} + k$ for all pairs of adjacent elements a_i, a_{i+1}, or $a_i + k = a_{i+1}$ for all pairs of adjacent elements a_i, a_{i+1}. For example, the sequence $2, 3.5, 5, 6.5, 8$ is a step-1.5 sequence, and $7, 4, 1, -2$ is a step-3 sequence. *(20')*
Write a python function called maximal_step_k_length(A,k) that takes a sequence of numbers A, and a number k, and returns the maximal length ℓ such that there is at least one contiguous sequence of elements in A that form a step-k sequence. You solution must have a time complexity $O(n)$, where n is the length of A.
For example, maximal_step_k_length([2,4,5,6,8,6,4,2,0,2,4,6,10,3,1],2) must return 5.

►**Exercise 241 (m20).** Your sport watch is equipped with an altitude sensor that, every second, measures your altitude in meters. Given an array $A = [a_1, a_2, \ldots, a_n]$ of n consecutive altitude measurements, you want to determine whether you had a high-power run. A high-power run occurs when there is a certain total altitude gain over a period of time, where the total altitude gain is the sum of all altitude gains (positive altitude variations) over that period. For example, the sequence of measurements $10, 10, 12, 11, 10, 11, 12$ corresponds to a total altitude gain of 4 meters ($10, 12$ and then $10, 11, 12$). *(30')*
Write a Python function called high_power_run(A,h,t) that takes a vector A of altitude measurements taken consecutively every second, an altitude gain h, and a time limit t, and returns True if A indicates a steep climb of at least h meters in at most t seconds, or False otherwise. Your solution must have a complexity $O(n)$. For example, high_power_run([10,6,1,3,2,1,3,4,6,5,6,4,3,4],6,5) must return True, because the measurements $1, 3, 4, 6, 5, 6$ indicate a total gain of 6 meters in 5 seconds. However, high_power_run([10,6,1,3,2,1,3,4,6,5,6,4,3,4],6,4) must return False, because there is no total gain of at least 6 meters in 4 seconds.

►**Exercise 242 (m20).** An array $A = [a_1, a_2, \ldots, a_n]$ of numbers is said to be in "peak" order if $a_i \geq a_{i-1}$ for all $1 < i \leq (n+1)/2$, and $a_j \geq a_{j+1}$ for all $(n+1)/2 \leq j < n$. In essence, A is in peak order when its first half is in ascending order while the second half is in descending order. Write a Python function called peak_order(A) that takes an array of numbers A and reorders its elements into a peak order. peak_order(A) must change the array A *in-place*, and must run in $O(n \log n)$ time. *(20')*

►**Exercise 243 (m20).** A *left-rotation* of an array A is defined as a permutation of A such that every element is shifted by one position to the left except for the first element that is moved to the last position. For example, with $A = [1, 2, 3, 4, 5, 6, 7, 8, 9]$, a *left-rotation* would change A into $A = [2, 3, 4, 5, 6, 7, 8, 9, 1]$.

Question 1: Write an algorithm rotate(A,k) that takes an array A and performs k left-rotations on A. The complexity of your algorithm must be $O(n)$, which means that the complexity must not depend on k. *(10')*

Question 2: Write a function rotate_inplace(A,k) that takes an array A and, in $O(n)$ steps, performs k left-rotations *in-place*. In-place means that rotate_inplace(A,k) may not use more than a constant *(30')* amount of extra memory. If your implementation of rotate(A,k) is already in-place, then you may use it directly to implement rotate_inplace(A,k).

►**Exercise 244 (m20).** Write a function is_sorted(A) that returns True if A is sorted in either ascending or descending order. Analyze the complexity of is_sorted(A). *(10')*

►**Exercise 245 (f20).** Given a set of integers A, define $C(A)$ as the set of all the subsets of A that contain at most one number whose decimal representation ends in the same digit. So, for example, if $A = \{7, 31, 17, 20\}$ then $C(A)$ contains $\{7\}, \{31\}$, and $\{7, 31, 20\}$, but does not contain the set $\{7, 20, 17\}$ because $\{7, 20, 17\}$ contains more than one element whose decimal representation ends in the same digit (7).

Question 1: Write a function count_C(A) that takes an array of distinct integers A and returns the size of $C(A)$. count_C(A) must run in linear time and must allocate only a constant amount of memory. *Hint:* the decimal representation of a number a ends in digit d when $a \equiv d \mod 10$, that is, when the remainder of the integer division of a by 10 is d, which you can check in python with the condition a % 10 == d. *(20')*
For example, print_C([7, 31, 17, 20]) must return 11.

Question 2: Write a function print_C(A) that prints $C(A)$, with each set in $C(A)$ on a separate line. print_C(A) must have a linear complexity in the size of $C(A)$, that is, it must be linear in the size of its output and therefore minimal. *(20')*
For example, print_C([7, 31, 17, 20]) should output the following lines (in any order):

```
7
17
31
31 7
31 17
20
20 7
20 17
20 31
20 31 7
20 31 17
```

►**Exercise 246 (f20).** Consider the following number-matching game. A pair of numbers a and b is worth 3 point if $a = b$; 5 if $a \neq b$ but a divides b exactly or vice-versa b divides a; 9 points if $a = b^2$ or $b = a^2$; and 1 point otherwise. Notice that if $a = b^2$, it is also the case that b divides a, but the value is still 9 points.
The game starts with two lists of numbers, A and B, from which you can remove any number of elements, resulting in two new sub-sequences $A' = [a_1, a_2, \ldots, a_\ell]$ and $B' = [b_1, b_2, \ldots, b_\ell]$. The score is the total value of all the pairs $(a_1, b_1), (a_2, b_2), \ldots (a_\ell, b_\ell)$. Notice that if A' and B' are not of the same size ℓ, the total score is still the same as the score of two lists trimmed to the smaller size ℓ.
For example, the initial score with $A = [4, 9, 5, 100]$ and $B = [1, 2, 2, 10, 3]$ is 16, but you can remove the second element from A and the first element from B to obtain $A' = [4, 5, 100]$ and $B' = [2, 2, 10, 3]$ with a score of 19.

Question 1: Write an algorithm MAXIMAL-SCORE(A, B) that computes the maximal score achievable at the number-matching game with input sequences A and B. Analyze the complexity of your solution. *(10')*

Question 2: Write a Python function maximal_score(A,B) that takes two arrays of integers A and B, and returns the maximal score achievable at the number-matching game. *(20')*

►**Exercise 247 (f20).** Consider the following decision problem. Given an undirected graph $G = (V, E)$, and a number k, output 1 if G contains a subgraph H that is a *tree* of size k, or 0 otherwise. Recall that a subgraph $H = (V_H, E_H)$ is defined by a subset $V_H \subseteq V$ and by all the edges $E_h \subseteq E$ that connect vertices in V_H. In other words, a subgraph can be obtained by removing a set of vertices *(20')*

and all the edges adjacent to them. Recall also that a tree over n vertices is a connected graph with no cycles, and therefore with $n-1$ edges.
Is this problem in NP? Show a proof of your answer in a text file called `ex3.txt`.

▶**Exercise 248 (f20).** Consider the following algorithm ALGO-X(P) that takes a sequence of n distinct 2D points $P = [(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)]$ each represented by its Cartesian coordinates, such that $P[i].x$ and $P[i].y$ are the coordinates of point $P[i]$, respectively.

```
ALGO-X(P = [(x1, y1), (x2, y2), ..., (xn, yn)])
n = P.length
for i = 1 to n
    for j = 1 to n
        if j ≠ i
            a_x = P[j].x − P[i].x
            a_y = P[j].y − P[i].y
            for k = j + 1 to n
                if k ≠ i
                    b_x = P[k].x − P[i].x
                    b_y = P[k].y − P[i].y
                    if a_x b_x + a_y b_y == 0
                        return TRUE
return FALSE
```

Question 1: Describe what ALGO-X does and analyze its complexity. Give a high-level, conceptual description of the functionality expressed by the algorithm. Do not simply paraphrase the pseudocode. *Hint:* recall from basic linear algebra that the dot-product of two vectors a and b relates to the angle between a and b. In particular, $a \cdot b = 0$ means that a and b are orthogonal, that is, they form a right angle. *(10')*

Question 2: Write an algorithm called BETTER-ALGO-X that does exactly the same thing as ALGO-X but with a strictly better time complexity. Analyze the complexity of your solution. *(20')*

▶**Exercise 249 (r20).** You are given an array A of objects. The objects are opaque, meaning that you do not know their structure. An equivalence relation exists between objects, that can be checked in constant-time with an algorithm EQUALS(x, y). No other relation exists, in particular there are no order relations between the objects. Write an algorithm CLUSTER(A) that changes A in-place so that equal objects are contiguous. Also, analyze the worst-case and best-case complexities of your solution. *(20')*
As an example, imagine that objects are letters with the usual case-insensitive equality relation (but without a lexicographical or any other ordering relation). Then, given an input

$$A = [\mathsf{A}, \mathsf{n}, \mathsf{t}, \mathsf{o}, \mathsf{n}, \mathsf{i}, \mathsf{o}, \mathsf{C}, \mathsf{a}, \mathsf{r}, \mathsf{z}, \mathsf{a}, \mathsf{n}, \mathsf{i}, \mathsf{g}, \mathsf{a}]$$

CLUSTER(A) could change A as follows

$$A = [\mathsf{C}, \mathsf{i}, \mathsf{i}, \mathsf{a}, \mathsf{A}, \mathsf{a}, \mathsf{a}, \mathsf{o}, \mathsf{o}, \mathsf{r}, \mathsf{z}, \mathsf{t}, \mathsf{n}, \mathsf{n}, \mathsf{n}, \mathsf{g}].$$

Notice that no particular order is required. The only requirement is that equal objects be contiguous in A. Notice also that the algorithm must be *in-place*. In practice this means that you may not use any additional data structure to store the elements of A.

▶**Exercise 250 (r20).** An array M holds a set of measurements of temperature and humidity in a forest. $M[i].time$ is the time of measurement i, $M[i].temperature$ is the temperature, and $M[i].humidity$ is the humidity. Measurements in M are time-ordered, so for $i < j$, $M[i].time < M[j].time$. A series of measurements $M[i], M[i+1], \ldots, M[j]$ (with $i < j$) indicates a fire danger when the temperature is monotonically increasing, so $M[i].temperature < M[i+1].temperature < \cdots < M[j].temperature$, and the humidity is monotonically decreasing, so $M[i].humidity > M[i+1].humidity > \cdots > M[j].humidity$.

Question 1: Write an algorithm MAXIMAL-DANGER-PERIOD(M) that finds the maximal duration of any fire-danger period in M, that is, the maximal interval $M[j].time - M[i].time$ $(i > j)$ such that the measurements between i and j indicate a fire danger. The result should be 0 if there are no fire-danger periods in M. Also, analyze the best and worst-case complexity of your solution. *(10')*

Question 2: Write a Python function `max_danger_linear(M)` that finds the maximal duration of any fire-danger period in $O(n)$ time. You may assume that the input array `M` contains objects with numeric attributes `time`, `temperature`, and `humidity`. *(20')*

►**Exercise 251 (r20).** Consider the following algorithm ALGO-X(A, B, k) that takes two arrays of numbers A and B and an integer k:

```
ALGO-X(A, B, k)
for i = 1 to A.length − k + 1
    d = 0
    j = 1
    while j + k − 1 ≤ B.length
        if d == k
            return TRUE
        elseif A[i + d] == B[j + d]
            d = d + 1
        else d = 0
            j = j + 1
return FALSE
```

Question 1: Describe what ALGO-X does and analyze its complexity. Do not just paraphrase the code. Explain the behavior of the algorithm at a high-level. *(10')*

Question 2: Consider the following algorithm: *(20')*

```
ALGO-Y(A, B)
if ALGO-X(A, B, 1)
    return FALSE
else return TRUE
```

Write an algorithm BETTER-ALGO-Y(A, B) that is exactly equivalent to ALGO-Y(A, B) and that runs in $O(n \log n)$ time, where n is the combined length of A and B.

►**Exercise 252 (r20).** Consider the following decision problem: Given an undirected graph G and an integer k, return TRUE if and only if G contains at least k vertices that are all reachable from each other. Answer the following questions about this problem in a text file called `ex4.txt`.

Question 1: Is this problem in NP? Show a proof of your answer. *(10')*

Question 2: Is this problem in P? Show a proof of your answer. *(20')*

Question 3: Can this problem be solved in linear time? Show a proof of your answer. *(10')*

►**Exercise 253 (m21).** Consider the following algorithm ALGO-X(A, k) that takes a sequence A of n numbers and a positive integer k:

```
ALGO-X(A, k)
B = ALGO-Y(A, 1, A.length + 1)
c = 0
for i = 1 to B.length
    if i ≤ k
        c = c + B[i]
    else return c
return c
```

```
ALGO-Y(A, i, j)
D = empty sequence
if j − i == 1
    append A[i] to D
elseif j − i > 1
    k = ⌊(i + j)/2⌋
    B = ALGO-Y(A, i, k)
    C = ALGO-Y(A, k, j)
    b = 1
    c = 1
    while b ≤ k − i or c ≤ j − k
        if c > j − k or (b ≤ k − i and B[b] < C[c])
            append B[b] to D
            b = b + 1
        else append C[c] to D
            c = c + 1
return D
```

Question 1: Explain what ALGO-X does. Do not simply paraphrase the code. Instead, explain the high-level semantics, independent of the code. *(5')*

Question 2: Analyze the complexity of ALGO-X. Is there a difference between the best- and worst-case complexity? If so, describe a best-case and a worst-case input of size n, as well as the behavior of the algorithm in each case. *(5')*

Question 3: Write an algorithm called BETTER-ALGO-X that does exactly the same thing as ALGO-X, but with a strictly better complexity in the average case. Analyze the complexity of BETTER-ALGO-X. Notice that if ALGO-X modifies the content of the input array A, then BETTER-ALGO-X must do the same. Otherwise, if ALGO-X does not modify A, then BETTER-ALGO-X must not modify A. *(20')*

►**Exercise 254 (m21).** Consider the following algorithm ALGO-X(A, x) that takes a sorted sequence A of n numbers and a positive number x.

```
ALGO-X(A, x)
for i = 1 to A.length
    if ALGO-Y(A, i, A.length + 1, A[i] + x)
        return TRUE
return FALSE
```

```
ALGO-Y(A, i, j, x)
while j > i
    k = ⌊(i + j)/2⌋
    if x < A[k]
        j = k
    elseif x > A[k]
        i = k + 1
    else return TRUE
return FALSE
```

Question 1: Explain what ALGO-X does. Do not simply paraphrase the code. Instead, explain the high-level semantics, independent of the code. *(5')*

Question 2: Analyze the complexity of ALGO-X. Is there a difference between the best- and worst-case complexity? If so, describe a best-case and a worst-case input of size n, as well as the behavior of the algorithm in each case. *(5')*

Question 3: Write an algorithm called BETTER-ALGO-X that does exactly the same thing as ALGO-X, but with a strictly better complexity in the worst case. Analyze the complexity of BETTER-ALGO-X, showing a best-case and a worst-case input. Notice that if ALGO-X modifies the content of the input array A, then BETTER-ALGO-X must do the same. Otherwise, if ALGO-X does not modify A, then BETTER-ALGO-X must not modify A. *(20')*

►**Exercise 255 (m21).** Given a sequence of $2n$ numbers $A = x_1, y_1, x_2, y_2, \dots, x_n, y_n$ representing the Cartesian coordinates of n points in the plane, $p_1 = (x_1, y_1), p_2 = (x_2, y_2), \dots p_n = (x_n, y_n)$, consider the line segments p_i–p_j defined by pairs of distinct points in A. You may assume that no two points in A are identical. That is, $i \neq j$ implies $p_i \neq p_j$.

Question 1: Write two Python functions, `count_vertical(A)` and `count_horizontal(A)`, that given the sequence A structured as above, return the number of vertical and horizontal segments in A, respectively. Also, write an analysis of the complexity of your solution. *(10')*

Question 2: Write a Python function `intersection(A)` that returns `True` if A contains at least one vertical segment that intersects at least one horizontal segment, or `False` otherwise. Also, write an analysis of the complexity of your solution, in particular describing a worst-case input. *(20')*
Two segments intersect when they have at least one point in common. For example, the vertical segment $(1,7)$-$(1,0)$ intersects the horizontal segment $(0,1)$-$(10,1)$. Similarly, the vertical segment $(1,7)$-$(1,0)$ intersects the horizontal segment $(1,0)$-$(3,0)$. However, the vertical segment $(1,7)$-$(1,0)$ does not intersect the horizontal segment $(0,10)$-$(10,10)$. Therefore, as an example, `intersection([9,3,5,6,0,9,3,2,6,7,7,9,3,5,1,8,8,4,9,0])` must return `False`, since the set of points $(9,3)$, $(5,6)$, $(0,9)$, $(3,2)$, $(6,7)$, $(7,9)$, $(3,5)$, $(1,8)$, $(8,4)$, $(9,0)$ do not define intersecting vertical and horizontal segments. Instead, with the sequence of points $(5,1)$, $(9,0)$, $(2,3)$, $(2,2)$, $(9,2)$, $(5,4)$, $(0,3)$, $(7,2)$, $(8,6)$, $(4,2)$, `intersection` must return `True`, since horizontal segment $(2,2)$-$(9,2)$ intersects vertical segment $(5,1)$-$(5,4)$; and with the sequence $(2,6)$, $(8,6)$, $(3,6)$, $(7,5)$, $(5,3)$, $(1,6)$, $(7,1)$, $(5,0)$, $(8,8)$, $(5,6)$, the result must be `True` because horizontal segment $(2,6)$-$(8,6)$ intersects vertical segment $(8,6)$-$(8,8)$.

►**Exercise 256 (m21).** Given a sequence of numbers $A = a_1, a_2, a_3, \dots, a_n$, we say that a subsequence $a_i, a_{i+1}, \dots, a_j$ of length $j - i + 1 \geq 2$ is strictly increasing if $a_i < a_{i+1} < \cdots < a_j$, or strictly decreasing if $a_i > a_{i+1} > \cdots > a_j$. *(30')*
Write a Python function `increasing_or_decreasing(A)` that, given a sequence of numbers A, in time $O(n)$ returns the string `'increasing'` if A contains a strictly increasing subsequence that is longer than any strictly decreasing subsequence in A; or vice-versa the result is `'decreasing'` if A contains a strictly decreasing subsequence that is longer than any strictly increasing subsequence in A. If there are no strictly increasing or strictly decreasing subsequences, then the return value must be the string `'flat'`. If there are strictly increasing and strictly decreasing subsequences, but the maximal sequences of the two kinds are of equal length, then the return value must be `'equal'`. Also, write an analysis of the complexity of your solution.
You may use the following examples to test your code:

```
>>> increasing_or_decreasing([1])
'flat'
>>> increasing_or_decreasing([1,1,1,1,1])
'flat'
>>> increasing_or_decreasing([1,2,1,2,1])
'equal'
>>> increasing_or_decreasing([1,2,1,2,10,1])
'increasing'
>>> increasing_or_decreasing([1,2,3,2,8,10,1,0])
'equal'
>>> increasing_or_decreasing([1,20,11,10,1,0])
'decreasing'
```

►**Exercise 257 (f21).** Consider the following algorithm ALGO-X(A) that takes a sequence A of n numbers.

ALGO-X(A)

```
for i = 2 to A.length
    j = i − 1
    a = remainder of the integer division A[i]/4
    s = TRUE
    while j > 0
        b = remainder of the integer division A[j]/4
        if a < b
            swap A[j] ↔ A[j + 1]
            j = j − 1
        else j = 0
```

Question 1: Explain what ALGO-X does. Do not simply paraphrase the code. Instead, explain the high-level semantics, independent of the code. Also, analyze the complexity of ALGO-X. *(5')*

Question 2: Write an algorithm called LINEAR-ALGO-X that does exactly the same thing as ALGO-X, but with a $O(n)$ time complexity. Notice that if ALGO-X modifies the content of the input array A, then LINEAR-ALGO-X must do the same. Otherwise, if ALGO-X does not modify A, then LINEAR-ALGO-X must not modify A. *(25')*

►**Exercise 258 (f21).** You are given a set of n persons represented by the set $P = \{1, 2, \dots, n\}$, and a symmetric relation *knows* $\subseteq P \times P$ represented as a Boolean function KNOWS(p, q), with $p, q \in P$, such that KNOWS(p, q) = TRUE (and KNOWS(q, p) = TRUE) if persons p and q have met at least once, or KNOWS(p, q) = KNOWS(q, p) = FALSE otherwise. We are only interested in the relation p *knows* q between two *distinct* persons $p \neq q$, so KNOWS(p, p) is always FALSE, by definition.

Question 1: With P and the KNOWS function, you are also given two positive integers k and ℓ, and with that you must decide whether there are at least k persons that have each met at least ℓ other persons. Is this decision problem in P? Write an algorithm that proves it is, or argue otherwise. *(10')*

Question 2: With P and the KNOWS function, you are also given a positive integers k, and with that you must decide whether there are at least k persons that have never met each other. Is this decision problem in NP? Write an algorithm that proves it is, or argue otherwise. *(20')*

►**Exercise 259 (f21).** Consider the following algorithm ALGO-Y(A, k) that takes a sequence A of n distinct numbers, and a positive integer k.

```
ALGO-Y(A, k)
for i = 1 to A.length − 1
    for j = i + 1 to A.length
        if A[j] · A[j] == A[i]
            k = k − 1
            if k == 0
                return TRUE
return FALSE
```

Question 1: Explain what ALGO-Y does. Do not simply paraphrase the code. Instead, explain the high-level semantics, independent of the code. Also, analyze the complexity of ALGO-Y. *(5')*

Question 2: Write an algorithm called BETTER-ALGO-Y that does exactly the same thing as ALGO-Y, but with a strictly better time complexity. Analyze the complexity of BETTER-ALGO-Y. Notice that if ALGO-Y modifies the content of the input array A, then BETTER-ALGO-Y must do the same. Otherwise, if ALGO-Y does not modify A, then BETTER-ALGO-Y must not modify A. *(25')*

►**Exercise 260 (f21).** Given two sequences A and B, a mirror sequence for A and B is a contiguous subsequence of A that also appears in reverse as a contiguous subsequence of B.

Question 1: Write a Python function `longest_mirror_seq(A,B)` that, given two sequences A and B of total length n, returns the maximal length of a mirror subsequence for A and B. Also analyze the complexity of your solution as a function of n. *(10')*
For example, with `A=[3,7,4,5,7]` and `B=[3,7,5,4,3]`, `longest_mirror_seq(A,B)` must return 3, because the sequence $4, 5, 7$ in A mirrors the sequence $7, 5, 4$ in B, and that sequence is maximal in length.

Question 2: Write a Python function `longest_mirror_seq2(A,B)` that returns the maximal length of a mirror subsequence for A and B in time $O(n^2)$. If your solution for Question 1 already satisfies this complexity requirement, then simply say so. *(20')*

►**Exercise 261 (r21).** You are given three sequences of numbers, $A = a_1, \dots, a_n$, $B = b_1, \dots, b_n$, and $C = c_1, \dots, c_n$, containing the precise daily measurements of the high temperature in three locations, L_A, L_B, and L_C, respectively. The measurements in A, B, and C are for the same sequence of n consecutive days. Write an algorithm COUNT-INVERSIONS(A, B, C) that, in time $O(n)$, returns the number of inversions in the given sequences. An inversion occurs when the ranking of the three locations in terms of their temperatures changes from one day to the next. For example, there is an inversion if one day the temperature at location L_A is higher than the temperature in *(20')*

L_C but the temperature in L_C is instead higher the next day. Notice that if one day the ranking changes from the previous day, you must count *one* inversion for that day, no matter how the ranking changes. You may assume that the temperatures are always different at the tree locations, that is, $a_i \neq b_i, a_i \neq c_i, c_i \neq b_i$ for all i.

▸**Exercise 262 (r21).** Write a linear-time algorithm AT-MOST-THREE-VALUES(A) that returns TRUE *(20')* if and only if the input sequence A contains at most three distinct values, or FALSE otherwise. For example, $A = [2, \text{“xyz”}, 2, -1, \text{“xyz”}, 2]$ contains six elements but only three distinct values, so in this case AT-MOST-THREE-VALUES(A) would return TRUE. As you can see from this example, the input array may contain values of different types (strings and numbers) that therefore compare not-equal.

▸**Exercise 263 (r21).** You are given a sequence $A = a_1, a_2, \dots, a_n$ of n numbers representing measurements collected at regular intervals at times $t = 1, 2, \dots, n$. Therefore, A defines n points on a chart with Cartesian coordinates $(1, a_1), (2, a_2), \dots, (n, a_n)$, respectively. Consider the following algorithm ALGO-X operating on sequence A:

```
ALGO-X(A)
for i = 1 to A.length
    for j = i + 1 to A.length
        if ALGO-Y(A, i, j)
            return TRUE
return FALSE
```

```
ALGO-Y(A, i, j)
p = NIL
r = NIL
for k = 1 to A.length
    if k ≠ i and k ≠ j
        if p == NIL
            p = k
        elseif r == NIL
            r = (A[k] − A[p])/(k − p)
        elseif r ≠ (A[k] − A[p])/(k − p)
            return FALSE
return TRUE
```

Question 1: Briefly explain what ALGO-X does and analyze the complexity of ALGO-X by describing a worst-case input. *(10')*

Question 2: Write an algorithm BETTER-ALGO-X that does the same as ALGO-X but with a strictly better time complexity. Notice that, if ALGO-X modifies its input, then BETTER-ALGO-X should also modify its input in the same way. Conversely, if ALGO-X does not modify its input, then BETTER-ALGO-X should not do that either. Also analyze the complexity of BETTER-ALGO-X. *(20')*

Question 3: Write an algorithm LINEAR-ALGO-X that does the same as ALGO-X with a $O(n)$ time complexity. If your solution for Question 2 is a valid solution for this question, then simply say so. *(20')*

▸**Exercise 264 (r21).** Consider a directed graph $G = (V, A)$ representing a set of software components (e.g., functions or methods) and their direct dependencies, such that, for two software components $u, v \in V$, there is an arc $(u, v) \in A$ from vertex u to vertex v, if u directly uses v (e.g., u invokes v). Let $d(v)$ be the number of unique components that directly or indirectly use v. Write an algorithm MAX-DEPENDENCIES($G = (V, Adj)$) that, given the adjacency-list representation of graph G, returns the maximum value of $d(v)$ for any component v in G. Also, analyze the complexity of your solution. *(30')*

▸**Exercise 265 (m22).** Write an algorithm MAX-HEAP-INSERT(H, x) that inserts a value x in a max-heap H. Also, write the content of H (as an array) after the insertion of each of the following values, in the given order, starting from an empty max-heap: *(20')*

$$3, 7, 3, 2, 9, 5, 9, 8, 5, 2, 9, 4, 7, 3, 9$$

▸**Exercise 266 (m22).** The following algorithm ALGO-X(A) takes an array A of n numbers.

```
ALGO-X(A)
for i = 1 to A.length
    s = 0
    for j = 1 to A.length
        if i ≠ j
            s = s + A[j]
    if A[i] == s
        return TRUE
return FALSE
```

Question 1: Explain what ALGO-X does. Do not simply paraphrase the code. Instead, explain the high-level semantics of the algorithm independent of the code. *(5')*

Question 2: Analyze the complexity of ALGO-X. Is there a difference between the best and worst-case complexity? If so, describe a best and a worst-case input of size n, as well as the behavior of the algorithm in each case. *(5')*

Question 3: Write an algorithm called BETTER-ALGO-X that does exactly the same thing as ALGO-X in $O(n)$ time. *(10')*

►**Exercise 267 (m22).** The following algorithm ALGO-Y(A, r, c) operates on an $r \times c$ matrix of $n = rc$ elements, where r and c are the numbers of rows and columns of the matrix, and the matrix is stored row-wise in the given array A. This means that the first c elements of A are the c elements of the first row of the matrix, the following c elements of A are the c elements of the second row of the matrix, and so on.

```
ALGO-Y(A, r, c)
for i = 1 to rc
    for j = i + 1 to rc
        if A[i] == A[j]
            a = ⌊(i − 1)/c⌋ // integer division
            b = ⌊(j − 1)/c⌋ // integer division
            if a == b or a == b − 1
                if i − ac == j − bc or i − ac == j − bc + 1 or i − ac == j − bc − 1
                    return TRUE
return FALSE
```

Question 1: Explain what ALGO-Y does. Do not simply paraphrase the code. Instead, explain the high-level semantics of the algorithm independent of the code. *(5')*

Question 2: Analyze the complexity of ALGO-Y. Is there a difference between the best and worst-case complexity? If so, describe a best and a worst-case input of size n, as well as the behavior of the algorithm in each case. *(5')*

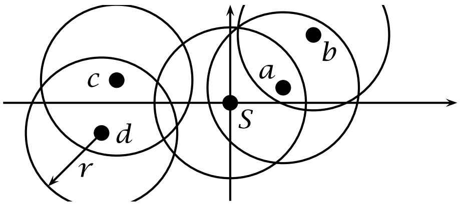
Question 3: Write an algorithm called BETTER-ALGO-Y that does exactly the same thing as ALGO-Y, but with a strictly better complexity in the worst case. Analyze the complexity of BETTER-ALGO-Y. *(20')*

►**Exercise 268 (m22).** Write an algorithm FIND-AVG-POINT(A) that takes an array of $n \geq 2$ numbers, and returns a position i where the values in A cross the average between the first and last element. More specifically, letting $m = (A[n] + A[1])/2$, FIND-AVG-POINT(A) must return an index i such that $A[i] \leq m \leq A[i+1]$ or $A[i] \geq m \geq A[i+1]$. FIND-AVG-POINT(A) must have a worst-case time complexity of $o(n)$, meaning strictly better than linear time. Also, analyze the complexity of FIND-AVG-POINT. (*Hint:* interpret the values in A as a series of points with coordinates $(i, A[i])$ connected by line segments. FIND-AVG-POINT(A) must return a position i where the segment crosses or touches the horizontal line at level m.) *(20')*

►**Exercise 269 (m22).** We say that an array A is in "e-top" order when $A[i] \leq A[j]$ for all i, j such that i is odd and j is even. Write an algorithm SORT-E-TOP(A) that sorts an array A in e-top order with an average-case time complexity of $O(n)$. You may want to use standard, well-known algorithms. However, you must explicitly write their pseudo-code. *(30')*

▶**Exercise 270 (f22).** Write an algorithm BST-COUNT-IN-RANGE(T, a, b) that, given the root t of a binary search tree and two values a and b, returns the number of keys in the tree that are between a and b. Also, analyze the best and worst-case complexity of your solution. *(20')*

▶**Exercise 271 (f22).** Some sensors equipped with a radio transmitter/receiver are deployed over a flat region. The location of each sensor is identified by its Cartesian coordinates (x, y). The sensors are supposed to send data to a central station located at coordinates $(0, 0)$, which is also equipped with the same radio transmitter/receiver. All transmitters/receivers have an effective range r, meaning that two radios can communicate if and only if their distance is at most r. However, the sensors and the base station establish a network, such that two devices that are not within direct radio communication can still communicate indirectly through one or more other devices that act as relay stations. See the example below.



We have four sensors, a, b, c, d, and a base station S. The circles represent the range of each radio. Sensor a can communicate with the base station directly, and sensor b can also communicate with the base station through a acting as a relay. Sensors c and d can communicate with each other but not with S.

Question 1: Write an algorithm CHECK-CONNECTIVITY(X, Y, r) that, given the coordinates of all the sensors stored in arrays X and Y, such that sensor i is located at coordinates $(X[i], Y[i])$, and given the communication range r, returns TRUE if all sensors can transmit their data to the base station, or FALSE if one or more sensors can not do that. Also, analyze the complexity of your solution. *(20')*

Question 2: Write an algorithm MINIMAL-CONNECTIVITY-RANGE(X, Y, t) that, given the coordinates of all the sensors stored in arrays X and Y, and given a precision threshold t, returns the minimal radio range r that would guarantee full connectivity. The resulting radius r may be an approximation of the actual minimal radius $\overline{r}$ up to a threshold t, meaning that $|r - \overline{r}| \le t$. *Hint:* you can use the CHECK-CONNECTIVITY algorithm of Question 1. You may use CHECK-CONNECTIVITY even if you did not write that algorithm correctly or at all. Analyze the complexity of MINIMAL-CONNECTIVITY-RANGE. *(20')*

▶**Exercise 272 (f22).** Given an array A of n numbers, we say that A contains a pair of value v if there are two elements $a_i, a_j \in A$ $(i \neq j)$ such that $a_i + a_j = v$. Now, given a positive integer k, you must decide whether the elements of A can form at least k pairs of the same value v. Notice that an element a_i may appear in at most one pair. For simplicity, you may assume that the values in A are distinct, that is, $i \neq j$ implies that $A[i] \neq A[j]$.
For example, for $k = 3$ and $A = [8, 3, 6, 10, 9, 14, 13, 20, 4, 5, 12]$, the answer is "yes", because we can form three pairs, such as $(10, 4), (8, 6), (9, 5)$, of the same value 14. For $k = 4$ and the same array A, the answer is still "yes", since A contains 4 pairs of equal value, such as $(8, 10), (6, 12), (13, 5), (14, 4)$. However, For $k = 5$ the answer is "no".

Question 1: Is this problem in NP? Write an algorithm that proves it, or argue the opposite. *(10')*

Question 2: Is the problem in P? Write an algorithm that proves it, or argue the opposite. *(20')*

▶**Exercise 273 (f22).** Consider the following algorithm ALGO-X(A, B) operating on two arrays of numbers A and B of total length $A.length + B.length = n$:

```
ALGO-X(A, B)
C = [FALSE] * A.length        // array of A.length Boolean values all initially FALSE
for j = 1 to B.length
    i = 1
    while i ≤ A.length and (C[i] == TRUE or A[i] ≠ B[j])
        i = i + 1
    if i ≤ A.length
        C[i] = TRUE
    else return FALSE
for i = 1 to A.length
    if C[i] == FALSE
        return FALSE
return TRUE
```

Question 1: Explain what ALGO-X does. Do not simply paraphrase the code. Instead, explain the high-level semantics, independent of the code. Also, analyze the best and worst-case complexity of ALGO-X. *(10')*

Question 2: Write an algorithm called BETTER-ALGO-X that does exactly the same thing as ALGO-X, but with a strictly better worst-case time complexity and equal or better best-case complexity. Analyze the complexity of BETTER-ALGO-X. Notice that if ALGO-X modifies the content of the input arrays A and B, then BETTER-ALGO-X must do the same. Otherwise, if ALGO-X does not modify A and B, then BETTER-ALGO-X must not modify A and B. *(20')*

►**Exercise 274 (f22b).** Write an algorithm BST-COUNT-OUTSIDE-RANGE(T, a, b) that, given the root T of a binary search tree and two values a and b, returns the number of keys in the tree that are outside of the interval $[a, b]$. Your solution must have a best-case complexity of $O(1)$. Also, analyze the worst-case complexity of your solution. *(30')*

►**Exercise 275 (f22b).** A social network N is defined by a set of users U and by a constant-time function $F(u_1, u_2)$ that tells whether users u_1 and u_2 are "friends". We say that a social network can be covered by a social circle of diameter $D \geq 1$ when, for all pairs of users a and b, either $F(a, b)$ or there is a chain $u_1, u_2, \ldots, u_k$ of $k < D$ other users such that $F(a, u_1), F(u_1, u_2), \ldots, F(u_k, b)$. Given a social network $N = (U, F)$ and a number d, consider the problem of determining whether the social network can be covered by a social circle of diameter d.

Question 1: Is this problem in NP? Write an algorithm that proves it, or argue the opposite. *(10')*

Question 2: Is the problem in P? Write an algorithm that proves it, or argue the opposite. *(20')*

►**Exercise 276 (f22b).** Consider the following algorithm ALGO-X(A, B) operating on two arrays of numbers A and B of total length $A.length + B.length = n$:

```
ALGO-X(A, B)
for ℓ = A.length downto 1
    for j = 1 to B.length
        for i = 1 to A.length − ℓ + 1
            s = 0
            for k = i to i + ℓ − 1
                s = s + A[k]
            if s == B[j]
                return ℓ
return 0
```

Question 1: Explain what ALGO-X does. Do not simply paraphrase the code. Instead, explain the high-level semantics, independent of the code. Also, analyze the best and worst-case complexity of ALGO-X. *(10')*

Question 2: Write an algorithm called BETTER-ALGO-X that does exactly the same thing as ALGO-X, but with a strictly better worst-case time complexity and equal or better best-case complexity. Analyze the complexity of BETTER-ALGO-X. Notice that if ALGO-X modifies the content of the input *(20')* arrays A and B, then BETTER-ALGO-X must do the same. Otherwise, if ALGO-X does not modify A and B, then BETTER-ALGO-X must not modify A and B.

▸**Exercise 277 (f22b).** Consider the following algorithm that takes an array A of numbers:

```
ALGO-Y(A)
B = [NIL] * A.length        // empty array of size A.length
ℓ = 0
for i = 1 to A.length
    k = 1
    for j = i + 1 to A.length
        if A[i] == A[j]
            k = k + 1
    if ℓ == 0 or B[ℓ] < k
        ℓ = 1
        B[ℓ] = A[i]
    elseif B[ℓ] == k
        ℓ = ℓ + 1
        B[ℓ] = A[i]
sort the first ℓ elements of B
for i = 1 to ℓ
    print B[i]
```

Question 1: Briefly explain what ALGO-Y does and analyze the complexity of ALGO-Y by describing a worst-case input. Do not simply paraphrase the code. Instead, explain the high-level semantics, independent of the code. *(10')*

Question 2: Write an algorithm BETTER-ALGO-Y that does the same as ALGO-Y but with a strictly better time complexity. Also analyze the complexity of BETTER-ALGO-Y. Notice that if ALGO-Y modifies the content of the input array A, then BETTER-ALGO-Y must do the same. Otherwise, if ALGO-Y does not modify A, then BETTER-ALGO-Y must not modify A either. *(20')*

▸**Exercise 278 (f22c).** Write an algorithm BST-COUNT-OUTSIDE-RANGE(T, a, b) that, given the root T of a binary search tree and two values a and b, returns the number of keys in the tree that are outside of the interval $[a, b]$. Your solution must have a best-case complexity of $O(1)$. Also, analyze the worst-case complexity of your solution. *(30')*

▸**Exercise 279 (f22c).** A social network N is defined by a set of users U and by a constant-time function $F(u_1, u_2)$ that tells whether users u_1 and u_2 are "friends". We say that a social network can be covered by a social circle of diameter $D \geq 1$ when, for all pairs of users a and b, either $F(a, b)$ or there is a chain $u_1, u_2, \ldots, u_k$ of $k < D$ other users such that $F(a, u_1), F(u_1, u_2), \ldots, F(u_k, b)$. Given a social network $N = (U, F)$ and a number d, consider the problem of determining whether the social network can be covered by a social circle of diameter d.

Question 1: Is this problem in NP? Write an algorithm that proves it, or argue the opposite. *(10')*

Question 2: Is the problem in P? Write an algorithm that proves it, or argue the opposite. *(20')*

▸**Exercise 280 (f22c).** Consider the following algorithm ALGO-X(A, B) operating on two arrays of numbers A and B of total length $A.length + B.length = n$:

```
ALGO-X(A, B)
for ℓ = A.length downto 1
    for j = 1 to B.length
        for i = 1 to A.length − ℓ + 1
            s = 0
            for k = i to i + ℓ − 1
                s = s + A[k]
            if s == B[j]
                return ℓ
return 0
```

Question 1: Explain what ALGO-X does. Do not simply paraphrase the code. Instead, explain the high-level semantics, independent of the code. Also, analyze the best and worst-case complexity of ALGO-X. *(10')*

Question 2: Write an algorithm called BETTER-ALGO-X that does exactly the same thing as ALGO-X, but with a strictly better worst-case time complexity and equal or better best-case complexity. Analyze the complexity of BETTER-ALGO-X. Notice that if ALGO-X modifies the content of the input arrays A and B, then BETTER-ALGO-X must do the same. Otherwise, if ALGO-X does not modify A and B, then BETTER-ALGO-X must not modify A and B. *(20')*

►**Exercise 281 (f22c).** Consider the following algorithm that takes an array A of numbers:

```
ALGO-Y(A)
B = [NIL] * A.length        // empty array of size A.length
ℓ = 0
m = 0
for i = 1 to A.length
    k = 1
    for j = i + 1 to A.length
        if A[i] == A[j]
            k = k + 1
    if m < k
        ℓ = 1
        m = k
        B[ℓ] = A[i]
    elseif m == k
        ℓ = ℓ + 1
        B[ℓ] = A[i]
sort the first ℓ elements of B
for i = 1 to ℓ
    print B[i]
```

Question 1: Briefly explain what ALGO-Y does and analyze the complexity of ALGO-Y by describing a worst-case input. Do not simply paraphrase the code. Instead, explain the high-level semantics, independent of the code. *(10')*

Question 2: Write an algorithm BETTER-ALGO-Y that does the same as ALGO-Y but with a strictly better time complexity. Also analyze the complexity of BETTER-ALGO-Y. Notice that if ALGO-Y modifies the content of the input array A, then BETTER-ALGO-Y must do the same. Otherwise, if ALGO-Y does not modify A, then BETTER-ALGO-Y must not modify A either. *(20')*

►**Exercise 282 (r22).** Let two numbers a, b define an interval, that is, the set of all numbers x such that $a \le x \le b$ or $b \le x \le a$. Write an algorithm COMPARE-INTERVALS(a_1, b_1, a_2, b_2) that compares the two intervals, I_1 defined by a_1 and b_1, and I_2 defined by a_2 and b_2. The algorithm should return "disjoint" if the two intervals are disjoint, meaning that there are no numbers that are in both I_1 and I_2; or "1 equals 2" if the two intervals are identical, meaning that all the numbers in I_1 are also in I_2 and vice-versa; or "1 covers 2" if all the numbers in I_2 are also in I_1 but not vice-versa; or "2 covers 1" if all the numbers in I_1 are also in I_2 but not vice-versa; or "partial" if more than one number is in both I_1 and I_2, but there are also numbers in I_1 that are not in I_2 and vice-versa; or "touch" if there is exactly one number that is in both I_1 and I_2, and there are also other numbers in I_1 that are not in I_2 and vice-versa. For example, COMPARE-INTERVALS$(-2.3, 2, 0, -7)$ must return "partial", because the interval $[-2.3, 0]$ is in both intervals $[-2.3, 2]$ and $[-7, 0]$, but there are also other elements in both; and COMPARE-INTERVALS$(5.5, 6.6, 7, 5.2)$ must return "2 covers 1", because the first interval, $[5.5, 6.6]$ is completely contained in the second interval $[5.2, 7]$, and the second interval has other numbers that are not in the first. *(30')*

►**Exercise 283 (r22).** Given an array A of $2n$ numbers, a *pairing* over A is a set of n pairs formed from the elements of A, such that each element $A[i]$ appears in exactly one pair. For example, given the array $A = [1, 0, 3, 7, 3, 2]$, a valid pairing could be $(1, 3), (3, 7), (2, 0)$.

Consider the following decision problem. Given an array A of $2n$ numbers, output "yes" if there exists a *uniform* pairing over A, meaning a pairing in which all the pairs have the same total value. The total value of a pair is simply the sum of its two elements. For example, the pairing given above is not uniform, since the total values of its three pairs are 4, 10, and 2, respectively.

Question 1: Is the problem in NP? Write an algorithm that proves it, or argue the opposite. *(10')*

Question 2: Is the problem in P? Write an algorithm that proves it, or argue the opposite. *(20')*

►**Exercise 284 (r22).** A *leaf* in a binary search tree T is a node that has no children.

Question 1: Write an algorithm AT-MOST-K-LEAVES(T, k) that, given the root of a binary search tree T and a non-negative integer k, returns TRUE if T has at most k leaves, or otherwise FALSE. Also, analyze the complexity of AT-MOST-K-LEAVES(T, k). *(10')*

Question 2: Write an algorithm AT-MOST-K-LEAVES-ITR(T, k) that is functionally identical to algorithm AT-MOST-K-LEAVES(T, k) but does not use recursion either directly or indirectly. If your implementation of AT-MOST-K-LEAVES(T, k) does not use recursion, just say so. *(20')*

►**Exercise 285 (r22).** Consider the following algorithm ALGO-X(A, B) operating on two arrays of numbers A and B of equal length $A.length = B.length = n$:

```
ALGO-X(A, B)
x = 0
for i = 1 to A.length
    k = ALGO-Y(A, B, i)
    if k > x
        x = k
return x
```

```
ALGO-Y(A, B, i)
k = i
while A[k] > B[k]
    k = k + 1
return k − i
```

Question 1: Explain what ALGO-X does. Do not simply paraphrase the code. Instead, explain the high-level semantics independent of the code. Also, analyze the best and worst-case complexity of ALGO-X. *(10')*

Question 2: Write an algorithm called BETTER-ALGO-X that does exactly the same thing as ALGO-X, but with a strictly better time complexity. Analyze the complexity of BETTER-ALGO-X. Notice that if ALGO-X modifies the content of the input arrays A and B, then BETTER-ALGO-X must do the same. Otherwise, if ALGO-X does not modify A and B, then BETTER-ALGO-X must not modify A and B. *(20')*

►**Exercise 286 (m23).** Write an algorithm MOUNTAIN-SORT(A) that, given an array A of n numbers, sorts A in-place such that the left half of A is increasing and the right half is decreasing. More specifically, the values from $A[1]$ to $A[\lfloor n/2 \rfloor]$ are increasing and the values from $A[\lfloor n/2 \rfloor]$ to $A[n]$ are decreasing. Notice that the left and right subsequences share the element in the middle position $A[\lfloor n/2 \rfloor]$. Notice also that the resulting order is not unique. For example, for $A = [8, 2, 5, -12, 2, 11, -15, -8, -1, 12]$, MOUNTAIN-SORT$(A)$ might result in $A = [-12, -8, -1, 1, 12, 11, 8, 5, 2, -15]$. *(30')*

You must detail every algorithm you use in your solution. So, if you want to, say, sort the input array or any part of it, you must explicitly write the sorting algorithm. Also, analyze the complexity of your solution.

►**Exercise 287 (m23).** Consider the following algorithm that takes an array A of n numbers.

```
ALGO-X(A)
n = A.length
x = 0
for i = 1 to n
    j = 1
    while j ≤ n and (i == j or A[i] ≠ A[j])
        j = j + 1
    if j > n
        x = x + 1
return x
```

Question 1: Explain what ALGO-X does. Do not simply paraphrase the code. Instead, explain the high-level semantics of the algorithm independent of the code. *(5')*

Question 2: Analyze the complexity of ALGO-X. Is there a difference between the best and worst-case complexity? If so, describe a best and a worst-case input of size n, as well as the behavior of the algorithm in each case. *(5')*

Question 3: Write an algorithm called BETTER-ALGO-X that does exactly the same thing as ALGO-X with a strictly better time complexity. *(10')*

►**Exercise 288 (m23).** An accounting system models a revenue transaction t as an object with two attributes, $t.date$ and $t.amount$ representing the date and amount of the transaction, respectively. Dates are represented as numbers of days since a reference initial date, such that $t_2.date - t_1.date$ is the number of days between transactions t_1 and t_2. Amounts are positive numbers. With that, consider the following ALGO-Y(T) that takes an array T of transactions:

```
ALGO-Y(T)
x = 0
for i = 1 to T.length
    l = T[i].amount
    r = T[i].amount
    for j = 1 to T.length
        if i ≠ j
            if T[j].date ≤ T[i].date and T[i].date − T[j].date ≤ 10
                l = l + T[j].amount
            if T[j].date ≥ T[i].date and T[j].date − T[i].date ≤ 10
                r = r + T[j].amount
    if x < r
        x = r
    if x < l
        x = l
return x
```

Question 1: Explain what ALGO-Y does. Do not simply paraphrase the code. Instead, explain the high-level semantics of the algorithm independent of the code. *(5')*

Question 2: Analyze the complexity of ALGO-Y. Is there a difference between the best and worst-case complexity? If so, describe a best and a worst-case input of size n, as well as the behavior of the algorithm in each case. *(5')*

Question 3: Write an algorithm called BETTER-ALGO-Y that does exactly the same thing as ALGO-Y, but with a strictly better complexity in the worst case. Analyze the complexity of BETTER-ALGO-Y. *(20')*

►**Exercise 289 (m23).** Consider the following array

$$H = [3, 5, 8, 6, 10, 9, 5, 6, 7, 20, 11, 17, 6, 9, 10]$$

Question 1: Does H contain a valid *min heap?* If so, extract the minimum value, rearranging H again as a minheap, and then write the resulting content of the array. If not, turn H into a min heap by applying a minimal number of swap operations, and write the resulting content of the array. Justify your answer. *(5')*

Question 2: Write an algorithm MIN-HEAP-ADD(H, x) that adds a new value x into a min heap H. *(10')*

Question 3: Execute MIN-HEAP-ADD(H, 4) using the algorithm you wrote as a solution to Question 2. In this case, the input H contains the min-heap resulting from your solution to Question 1. Illustrate the execution of MIN-HEAP-ADD(H, 4) by writing the full content of the array H at the beginning of each iteration of the algorithm, as well as at the end of the algorithm. *(5')*

►**Exercise 290 (m23).** Write an algorithm SQUARE-ROOT(n) that, given a non-negative integer n, returns $\lfloor \sqrt{n} \rfloor$. SQUARE-ROOT(n) may only use the basic arithmetic operations of addition, subtraction, multiplication and division (integer), and must run in $O(\log n)$ time. *(20')*

▶**Exercise 291 (f23).** An array A of n numbers is sorted. Some elements are then set to 0. Write an algorithm RE-SORT(A) that takes such an array A and sorts it in-place and in time $O(n)$. *(30')*

▶**Exercise 292 (f23).** Consider the following game: you start with two decks of n playing cards each (shuffled). At each round, you remove one or two cards as follows. If the two cards at the top of the two decks have the same suit or the same numeric value, you may remove both of them at no cost. If the two cards have different suits and numbers, or if you do not choose to remove both of them, you must choose to remove one of the two cards at a cost corresponding to its numeric value. If one of the decks is empty, you have no choice: you must remove the card on the remaining deck at the cost of its numeric value. The game ends when both decks are empty.
Now consider the following decision problem: given the two initial shuffled decks A and B and a maximal cost c, decide whether it is possible to play a game with a total cost less than c. A and B are arrays of cards; the functions $suit(x)$ and $value(x)$ return, in $O(1)$ time, the suit and numeric value of a card x, respectively. For example, $suit(A[i])$ returns the suit of the the i-th card on the A deck.

Question 1: Is this problem in NP? Show a proof of your answer. *(10')*
Hint: a decision problem is in NP when an example that shows that the answer is "yes" can be verified in polynomial time. Here, a sequence of game choices can be such an example.

Question 2: Is this problem in P? Show a proof of your answer. *(20')*
Hint: consider a dynamic-programming approach to find the minimal cost of a game.

▶**Exercise 293 (f23).** Consider the following algorithm that takes two strings A and B. You may assume that characters have numeric codes between 0 and m for some relatively small constant m. For example, ASCII characters are encoded by numbers between 0 and 127.

```
ALGO-X(A, B)
V = [] // empty array
for i = 1 to B.length
    append 0 to V
for i = 1 to A.length
    x = FALSE
    j = 1
    while j ≤ B.length and x == FALSE
        if A[i] == B[j] and V[j] == 0
            x = TRUE
            V[j] = 1
        else j = j + 1
for j = 1 to B.length
    if V[j] == 0
        return FALSE
return TRUE
```

Question 1: Explain what ALGO-X does. Do not simply paraphrase the code. Instead, explain the high-level semantics, independent of the code. Also, analyze the complexity of ALGO-X. *(10')*

Question 2: Write an algorithm called BETTER-ALGO-X that does exactly the same thing as ALGO-X, but with a strictly better complexity. Analyze the complexity of BETTER-ALGO-X. Notice that if ALGO-X modifies the content of the input strings, then BETTER-ALGO-X must do the same. Otherwise, BETTER-ALGO-X must not modify A and B. *(20')*
Bonus: extra points if your BETTER-ALGO-X runs in linear time. *(5')*

▶**Exercise 294 (f23).** Write an algorithm MINIMAL-ADDITIONAL-EDGES(G) that takes an undirected graph G and returns the minimal number of edges that must be added to G to make it connected. *(30')*

▶**Exercise 295 (r23).** Write an algorithm BST-ROOT-CHANGE(t, x) that takes a non-empty binary search tree t and changes the key of the root node (meaning t) to x without creating any new nodes. In other words , BST-ROOT-CHANGE(t, x) must somehow rearrange the nodes of the BST. BST-ROOT-CHANGE(t, x) must then return the new root, which can be the same as the old one. You must detail every algorithm you use. Also, analyze the complexity of your solution as a function of the size n and the height h of the tree. *(30')*

►**Exercise 296 (r23).** We want to cover a set of n numbers with a set of k intervals such that the total length of the intervals is minimal. An interval $[a, b]$, defined by two numbers $a \le b$, covers all the numbers between a and b, including a and b. For example, $[3, 7]$ and $[6, 10.5]$ cover all the numbers in $A = [3, 5, 7, 9]$. However, their total length $(7-3) + (10.5-6) = 8.5$ is not minimal. The minimal length is instead 4. In general, we want to have k intervals $[a_1, b_1], [a_2, b_2], \ldots, [a_k, b_k]$ such that, for each number x in A, there is at least one interval $[a_i, b_i]$ such that $a_i \le x \le b_i$, and the total length $\sum(b_i - a_i)$ is minimal.

Question 1: Write an algorithm MINIMAL-K-INTERVAL-COVER-LENGTH(A, k) that, given an array A of numbers and a positive integer k, returns the minimal total length of k intervals that cover every number in A. Also, analyze the complexity of your solution. *(10')*

Question 2: Write an algorithm MINIMAL-K-INTERVAL-COVER-LENGTH(A, k) that runs in $O(n \log n)$ time. If this is already the case for your solution to Question 1, then just say so. *(20')*

►**Exercise 297 (r23).** Consider the following algorithm ALGO-X(A, k) that takes an array A of numbers, and a positive integer k.

```
ALGO-X(A, k)
n = A.length
for i = 1 to n
    x = 0
    y = 0
    r = 0
    for j = 1 to n
        if A[j] < A[i]
            x = x + 1
            r = r + A[j]
        elseif A[j] == A[i]
            y = y + 1
    if x ≤ k and x + y ≥ k
        return r + A[i](k − x)
return NIL
```

Question 1: Explain what ALGO-X does. Do not just paraphrase the code. Instead, explain the high-level semantics, independent of the code. Also, analyze the complexity of ALGO-X. *(10')*

Question 2: Write an algorithm called BETTER-ALGO-X that does exactly the same thing as ALGO-X, but with a strictly better time complexity. Analyze the complexity of BETTER-ALGO-X. Notice that if ALGO-X modifies the content of the input array, then BETTER-ALGO-X must do the same. Otherwise, BETTER-ALGO-X must not modify A. *(20')*

►**Exercise 298 (r23).** Consider the following algorithm ALGO-Y(A) that takes an array A of numbers.

```
ALGO-Y(A)
B = MERGE-SORT(A)
n = A.length
x = 1
for i = 2 to n
    if B[i] ≠ B[i − 1]
        x = x + 1
if x > 3
    return TRUE
else return FALSE
```

Question 1: Explain what ALGO-Y does. Do not simply paraphrase the code. Instead, explain the high-level semantics, independent of the code. Also, analyze the complexity of ALGO-Y. *(10')*

Question 2: Write an algorithm called BETTER-ALGO-Y that does exactly the same thing as ALGO-Y, but with a strictly better time complexity. Analyze the complexity of BETTER-ALGO-Y. Notice that if *(20')*

ALGO-Y modifies the content of the input array, then BETTER-ALGO-Y must do the same. Otherwise, BETTER-ALGO-Y must not modify A.

Solutions

WARNING: solutions are sparse, meaning that many are missing, and many are only sketched at a high level—and many may be incorrect! Please, consider contributing your solutions, including alternative solutions, and please report any error you might find to the author (Antonio Carzaniga <antonio.carzaniga@usi.ch>).

▷ *Solution 10.6*

Yes, the exact-change problem is in NP. There is in fact a verification algorithm that, given an instance of the problem (V, x) and a “witness” set S that shows that the solution is 1, can check in polynomial time that S indeed proves that the solution is 1. Below is such an algorithm:

```
EXACT-CHANGE-VERIFY(V, x, S)
t = 0
for v ∈ S
    if v ∉ V
        return FALSE
    t = t + v
if t == x
    return TRUE
else return FALSE
```

▷ *Solution 52*

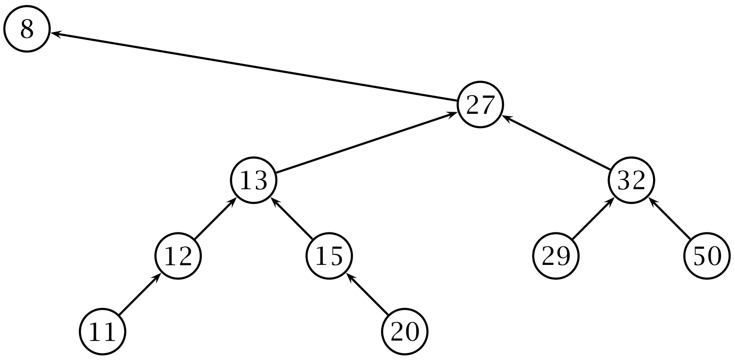
Quick-sort. Best-case is $O(n \log n)$, worst-case is $O(n^2)$.

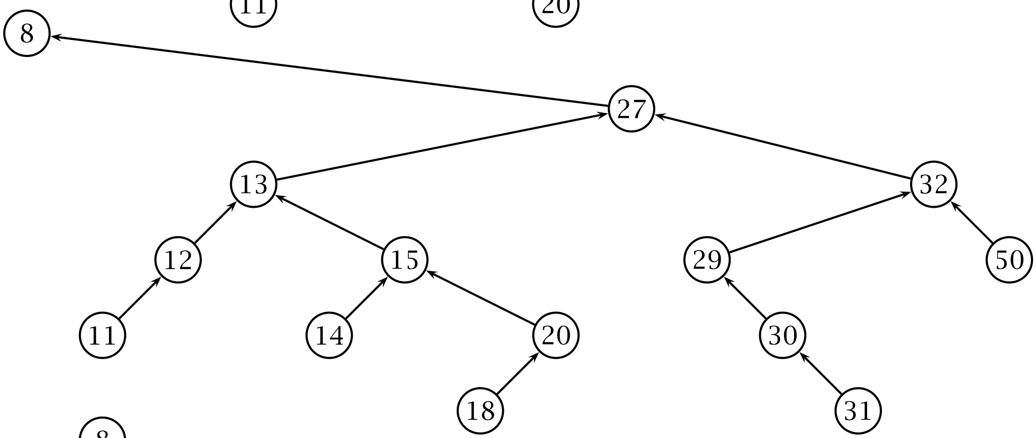
▷ *Solution 53*

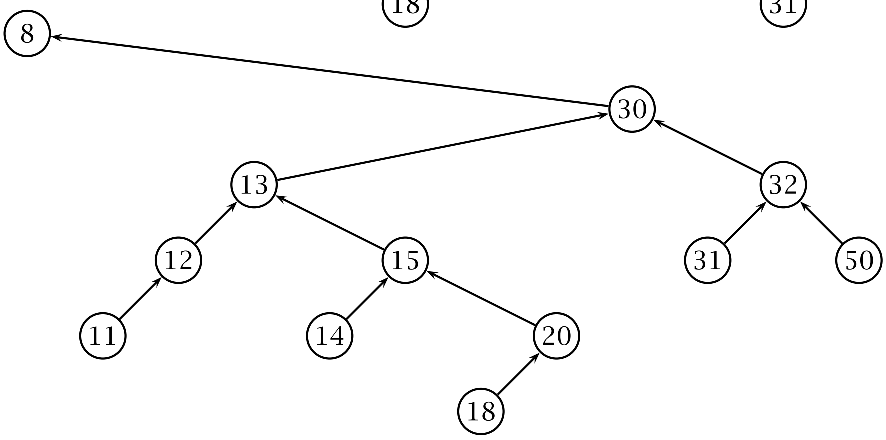
ALGORITHM-I sorts the input array in-place. In the best case, the algorithm terminates in the first execution of the outer loop, with the condition $s ==$ TRUE. This is the case when the inner loop does not swap a single element of the array, meaning that the array is already sorted. So, the best-case complexity is $O(n)$. Conversely, the worst case is when each iteration of the outer loop swaps at least one element. This happens when the array is sorted in reverse order. So, the worst-case complexity is $O(n^2)$.

ALGORITHM-II sorts the input array in-place so that the value $v = A[0]$, that is the element originally at position 0, ends up in position q, and every other element less than v ends up somewhere in $A[1 \ldots q-1]$, that is to the left of q, and every other element less than or equal to v ends up somewhere in $A[q+1 \ldots |A|]$. In other words, ALGORITHM-II partitions the input array in-place using the first element as the “pivot”. The loop closes the gap between i and j, which are initially the first and last position in the array, respectively. Each iteration either moves i to the right or j to the left, so each iteration reduces the gap by one. Therefore, in any case—worst case is the same as the best case—the complexity is $O(n)$.

▷ *Solution 61*





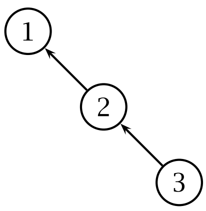


▷ *Solution 62*

a) 50 32 20 29 15 13 12 8 27 11
b) 51 43 50 29 32 20 12 8 27 11 15 13
c) 32 29 20 27 15 13 12 8 11

▷ *Solution 63*

Proof: Let $H = [1, 2, 3]$, then T would look like this:



▷ *Solution 67.1*

True. $O(n!)$ is at most $Kn!$ so $\log(Kn!) = \log K + \log 1 + \log 2 + \cdots + \log n \leq n \log n$

▷ *Solution 67.2*

False. as a counter example, let $f(n) = \sqrt{n}$

▷ *Solution 67.3*
False. Counter-example: $f(n) = 1$ and $g(n) = n$.

▷ *Solution 67.4*
False. Counter example: $f(n) = n$ and $g(n) = n$

▷ *Solution 67.5*
False. Counter example: $f(n) = \sqrt{n}$ and $g(n) = \sqrt{(}n)$

▷ *Solution 68*
SHUFFLE-A-BIT has the same common structure as a best-case run of QUICK-SORT. There is an initial linear phase, and then there are two recursions on arrays of size $n/2$. This results in $\log n$ levels of recursion, each having a total cost of $O(n)$. Therefore the complexity is $n \log n$.

▷ *Solution 69.1*
yes.

▷ *Solution 69.2*
yes.

▷ *Solution 69.3*
undefined.

▷ *Solution 69.4*
yes.

▷ *Solution 69.5*
undefined.

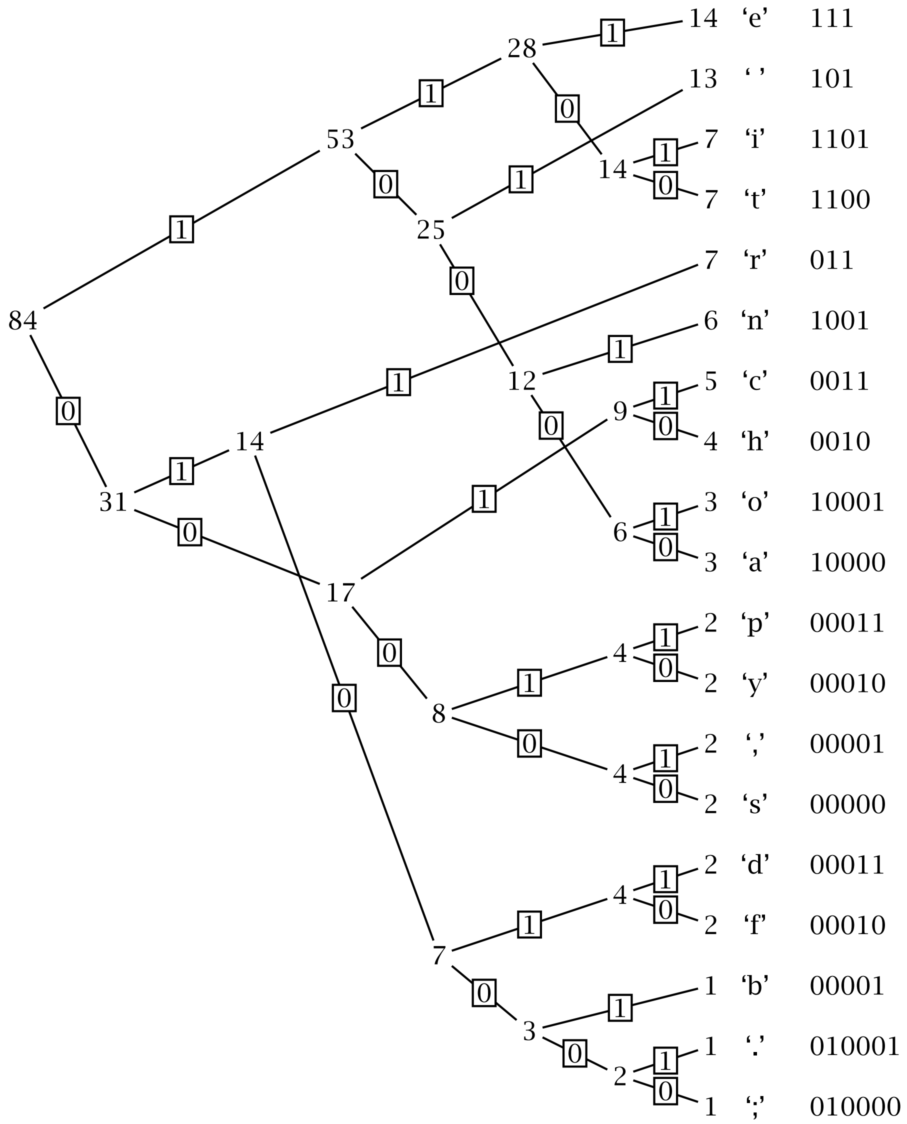
▷ *Solution 69.6*
undefined.

▷ *Solution 69.7*
yes.

▷ *Solution 69.8*
undefined.

▷ *Solution 69.9*
undefined.

▷ *Solution 70*
First figure out the frequencies and sort the characters by frequency. Then we proceed with the derivation:



▷ *Solution 72*

```
IsColorValid(G = (V, E), v)
  for each u adjacent to v
      if color[u] = color[v]
          return FALSE
  return TRUE
```

```
Color(G = (V, E))
  for each v ∈ V
      color[v] = 0
  for each v ∈ V
      color[v] = 1
      while IsColorValid(G = (V, E), v) = FALSE
          color[v] = color[v] + 1
  return color
```

▷ *Solution 78*

Given an array A of number, ALGO-X(A) returns TRUE if and only if there are three numbers $x \le y \le z \in A$ such that $y - x = z - y$. ALGO-X does that by testing each triple of distinct elements of A. There are $\binom{n}{3} = n(n-1)(n-2)/3!$ such triples, so the complexity is $\Theta(n^3)$.

A better way to do the same thing is as follows:

```
Better-Algo-X(A)
sort A
for i = 1 to A.length − 2
    for j = i + 2 to A.length
        m = (A[i] + A[j])/2
        if Binary-Search(A[i + 1 ... j − 1], m)
            return TRUE
return FALSE
```

In essence, after sorting the numbers, this algorithm tests each pair of non-adjacent numbers and then looks for the median using a binary search. There are $O(n^2)$ pairs of non-adjacent numbers in A, and binary-search costs $O(\log n)$, so the complexity is $O(n^2 \log n)$.

▷ *Solution 89*

```
Tree-To-Vine(t)
 1  if t == NIL
 2      return (NIL
 3  while t.left ≠ NIL
 4      t = BST-Right-Rotate(t)
 5  root = t
 6  while t.right ≠ NIL
 7      while t.right.left ≠ NIL
 8          t.right = BST-Right-Rotate(t.right)
 9      t = t.right
10  return root
```

The best-case complexity is $\Theta(n)$, which corresponds to the case of BST that is already a vine. The general worst-case complexity is certainly $O(n^2)$, since the outer loop (line 6) can run for at most n iterations, and similarly the inner loop (line 7) can also run for at most n iterations. However, it is not immediately obvious that the quadratic complexity is "tight". In fact, the complexity is $\Theta(n)$ also in the worst case. To see why, consider what happens to an edge e in the tree. If e is a left edge—that is, an edge connecting a parent node to a left child node—then at some point the algorithm will rotate e with a right rotation of the parent node, transforming e into a right edge that the algorithm will then simply traverse once. In other words, a left edge will involve two steps: a right rotation plus a traversal. If e is a right edge, then e might be immediately traversed, or it might be transformed into a left edge due to a rotation of another edge right above and to the right of e, which means that at some point e will be treated like any other left edge, so rotated and then traversed. In any case, every edge induces a constant-time process, which means that the algorithm linear, since there are $n - 1$ edges in the tree.

▷ *Solution 90*

```
Is-Perfectly-Balanced(t)
if t == NIL
    return (TRUE, 0)
(balanced_l, weight_l) = Is-Perfectly-Balanced(t.left)
(balanced_r, weight_r) = Is-Perfectly-Balanced(t.right)
if balanced_l and balanced_l and |weight_r − weight_l| ≤ 1
    return (TRUE, weight_l + weight_r + 1)
else return (FALSE, weight_l + weight_r + 1)
```

▷ *Solution 92*

We are not allowed to modify H, and we are not allowed to create a copy of H that we can then sort. So, we must print the elements in order, by simply reading H. We know that each number is unique in H, so the idea is this: we start from the minimum value in H, which happens to be in the first position of H, print that value, and then look for the second-smallest number, which we can simply find with a linear scan. We then proceed with the third-smallest, and so on, which again we can find with a linear scan. Notice that we can use a linear scan to find the i-th smallest element by simply considering only those elements in H that are greater than the smallest element we found in the previous $(i - 1)$ scan.

In pseudo-code:

```
HEAP-PRINT-IN-ORDER(H)
m = H[0]
print m
for i = 2 to H.length
    x = ∞
    for j = 2 to H.length
        if H[j] > m and H[j] < x
            x = H[j]
    m = x
    print m
```

It is easy to see that the complexity of HEAP-PRINT-IN-ORDER is $\Theta(n^2)$.

▷ *Solution 94*

One way to proceed is to try to progressively merge all pairs of intervals.

```
SIMPLIFY-INTERVALS(X)
i = 1
while i + 3 ≤ X.length
    j = i + 2
    if X[i + 1] < X[j] o X[i] > X[j + 1]
        j = j + 2
    else if X[j] < X[i]
            X[i] = X[j]
        if X[j + 1] > X[i + 1]
            X[i + 1] = X[j + 1]
            X[j] = X[X.length]
            X[j + 1] = X[X.length − 1]
            X.length = X.length − 2
    i = i + 2
```

```
ALGO-Y(A, i)
while i < A.length
    A[i] = A[i + 1]
    i = i + 1
A.length = A.length − 1 // discards last
```

▷ *Solution 96*

ALGO-X removes every element equal to k from array A. with a complexity of $\Theta(n^2)$.
Consider as a worst-case input an array A in which all n values are equal to k. In this case, ALGO-X would iterate over lines 3 and 4 (always with i equal to 1). In each iteration, ALGO-X would then invoke ALGO-Y (again with i equal to 1), which would then iterate over the length of the array, effectively removing the i-th element by shifting every subsequent element to the left by 1 position, and then by cutting the length of the array by 1.
So, ALGO-Y would run for n iterations the first time, then $n-1$ the second time, then $n-2$, and so on, until the array is completely empty. The complexity is therefore $n+(n-1)+\ldots+2+1 = \Theta(n^2)$.
A better way to remove every element equal to k from an array A is as follows.

```
BETTER-ALGO-X(A, k)
j = 1
for i = 1 to A.length
    if A[i] ≠ k
        A[j] = A[i]
        j = j + 1
A.length = j − 1
```

▷ *Solution 100.1*

No.

▷ *Solution 100.2*

No. SAT is NP-complete, meaning that every problem in NP can be reduced to SAT (polynomially), so if SAT can then be reduced to Q, then all problems in NP can be reduced to Q, which makes Q an NP-hard problem. However, to say that Q is NP-complete, we also have to know that Q is itself in NP.

▷ *Solution 100.3*

Yes. SAT is in NP, and therefore there is a polynomial-time *verification* algorithm for SAT. Since Q (and Q') can be transformed into SAT, that means that one can implement a polynomial verification algorithm also for Q (and Q'), by first transforming the Q instance into an instance of SAT, and then running the verification algorithm for SAT. Thus both Q and Q' are polynomially verifiable.

▷ *Solution 100.4*

No. We can say that Q is not more complex than Q', but we can not say much about Q'.

▷ *Solution 100.5*

Yes. Q is polynomially solvable, since it can be transformed into Q' and then solved in polynomial time through algorithm A. This means that Q is in P and therefore also in NP.

▷ *Solution 100.6*

Yes. Q is in P, since it can be easily solved through a breadth-first search.

▷ *Solution 116*

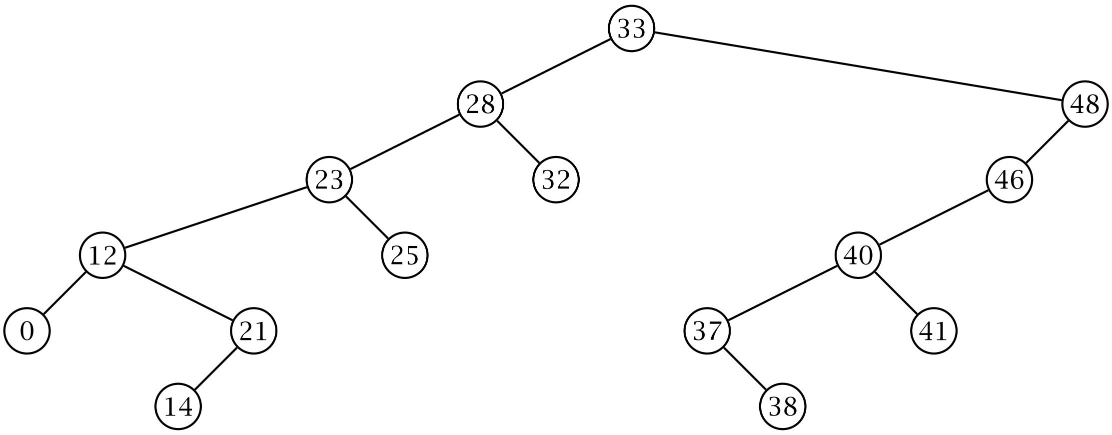
We must first of all understand what ALGO-XR(A, t, i, r) does. It is useful to interpret t as a target value, i as an index into A, and r as a count of remaining elements. If r is zero, ALGO-XR(A, t, i, r) simply checks that $A[i]$ equals the target t. Otherwise, ALGO-XR(A, t, i, r) tries the target $t - A[i]$. Effectively, this means that ALGO-XR(A, t, i, r) returns true of there are exactly $r + 1$ distinct elements in A starting at position i whose sum is the target t. Therefore, ALGO-X(A), which is equivalent to ALGO-XR$(A, 0, 1, 2)$, returns true if there are exactly three distinct elements in A whose sum is 0.
The worst-case input, which determines the complexity of ALGO-XR and therefore ALGO-X, is such that the execution of each call ALGO-XR goes through the loop without ever returning TRUE. This is the case, for example, with any sequence A of n positive numbers. In this case, the algorithm effectively checks every r-tuple, in A. So, the complexity of ALGO-XR is $T(n) = \binom{n}{r+1} = O(n^k)$, and for ALGO-X that is $T(n) = \binom{n}{3} = O(n^3)$.
We can improve ALGO-X by checking, for every pair of elements $A[i], A[j]$ ($1 \le i < j \le A.\mathit{length}$), whether A contains a third element $A[k]$, with $k > j$, such that $A[i] + A[j] + A[k] = 0$. This amounts to finding the value $-(A[i] + A[j])$ in the subsequence $A[j + 1 \ldots A.\mathit{length}]$. And that search operation can be sped-up by first sorting the array and using binary search.

BETTER-ALGO-X(A)

```
sort A
for i = 1 to A.length − 2
    for j = 1 to A.length − 1
        run a binary search of −(A[i] + A[j]) within A[j + 1 ... A.length]
        if −(A[i] + A[j]) ∈ A[j + 1 ... A.length]
            return TRUE
return FALSE
```

Here the complexity is $\Theta n^2 \log n$.

▷ *Solution 127.2*



Optimal sequence: 32, 21, 25, 40, 37, 46, 41, 12, 23, 48, 14, 33, 38, 0, 28.

▷ *Solution 129.1*

```
SORT-LINES-BY-WORD-COUNT(T)
X = array of 40 empty lists
for i = 1 to T.length
    c = WORD-COUNT(T[i])
    append T[i] to X[c]
i = 1
for c = 1 to 40
    for l ∈ X[c]
        T[i] = l
        i = i + 1
```

```
WORD-COUNT(l)
count = 0
for i = 1 to l.length
    if l[i] == ' ' and (i == 1 or l[i − 1] ≠ ' ')
        count = count + 1
return count
```

▷ *Solution 129.2*

```
SORT-LINES-BY-WORD-COUNT(T)
j = 1
c = 1
while j ≤ T.length and c ≤ 40
    for i = j to T.length
        if WORD-COUNT(T[i]) == c
            swap T[j] ↔ T[i]
            j = j + 1
    c = c + 1
```

```
WORD-COUNT(l)
count = 0
for i = 1 to l.length
    if l[i] == ' ' and (i == 1 or l[i − 1] ≠ ' ')
        count = count + 1
return count
```

▷ *Solution 131*

We start by modeling MAXIMAL-NON-ADJACENT-SUM as a classic recursive dynamic-programming algorithm. Given a sequence $A = a_1, a_2, \ldots, a_n$, there are two cases:

(i) the maximal sequence includes a_1, and therefore does not include a_2 and instead includes the maximal sequence for the remaining subsequence $a_3 \ldots a_n$;

(ii) the maximal sequence does not include a_1 and therefore is the same as the maximal sequence for the subsequence starting at a_2.

Thus the maximal solution is the best of these two. Let $OPT(a_1, a_2, \ldots, a_n)$ denote the maximal weight of non-adjacent elements from a sequence $a_1, a_2, \ldots, a_n$. With this, the algorithm is as follows:

$$OPT(a_1, a_2, a_3, \ldots, a_n) = \max\{a_1 + OPT(a_3, \ldots, a_n), OPT(a_2, \ldots, a_n)\}$$

Now we just have to write this simple, recursive dynamic-programming solution as a single iteration. This can be done by remembering only two values in each iteration, namely the optimal value for the previous two elements in the sequence. We can perform this iteration in either direction, so here we do it in increasing order, left-to-right. Therefore, for each element a_i, we must remember the two previous optimal values $OPT(a_1, \ldots, a_{i-1})$ and $OPT(a_1, \ldots, a_{i-2})$. The full algorithm is as follows:

```
MAXIMAL-NON-ADJACENT-SUM(A)
p = 0
q = 0
r = 0
for i = 1 to A.length
    r = max{A[i] + p, q}
    p = q
    q = r
return r
```

▷ *Solution 146.1*
ALGO-X returns the maximal sum of any contiguous subsequence of A.

▷ *Solution 146.2*
Dynamic programming: with i going from left to right, let $x(i)$ be the value of the maximal contiguous sequence *ending at position* i. So, $x(1) = A[1]$, $x(i) = \max\{A[i] + x(i-1), A[i]\}$.

▷ *Solution 154*

```
MAX-HEAP-INSERT(H, k)
H.heap-size = H.heap-size + 1
H[H.heap-size] = k
i = H.heap-size
while i > 1 and H[i] > H[⌊i/2⌋]
    swap H[i] ↔ H[⌊i/2⌋]
    i = ⌊i/2⌋
```

The complexity is $\Theta(\log n)$.

▷ *Solution 155.1*

```
FIND-ELEMENTS-AT-DISTANCE(A, k)
for i = 1 to A.length
    if BINARY-SEARCH(A[i + 1 .. A.length], k + A[i])
        return TRUE
return FALSE
```

The complexity is $\Theta(n \log n)$, since for each of the n elements, we perform a binary search that runs in $\Theta(\log n)$.

▷ *Solution 155.2*

```
FIND-ELEMENTS-AT-DISTANCE(A, k)
i = 1
j = 2
while j ≤ A.length
    if A[j] − A[i] < k
        j = j + 1
    elseif A[j] − A[i] > k
        i = i + 1
    else return TRUE
return FALSE
```

In each iteration of the loop we either increase j or i by one (or we return). Also, the loop is such that $j \geq i$, so in at most $\Theta(n)$ iterations we push j beyond *A.length*. Thus the complexity is $\Theta(n)$.

▷ *Solution 156*

```
IS-PRIME(x)
i = 2
while i * i < x
    if i divides x
        return TRUE
    i = i + 1
return FALSE
```

```
PARTITION-PRIMES-COMPOSITES(A)
i = 1
j = A.length
while i < j
    if IS-PRIME(A[j])
        swap A[j] ↔ A[i]
        i = i + 1
    elseif not IS-PRIME(A[i])
        swap A[j] ↔ A[i]
        j = j − 1
    else i = i + 1
        j = j − 1
```

IS-PRIME runs in $\Theta(\sqrt{m})$, while PARTITION-PRIMES-COMPOSITES requires $\Theta(n)$ basic operations and $\Theta(n)$ invocations of IS-PRIME. The complexity is therefore $\Theta(n\sqrt{m})$.

▷ *Solution 157*

In this exercise, randomization or rotations cannot be used to balance the height of the BST. So, input sequence A must be pre-sorted so that, inserting elements in the tree in the new order, the resulting BST has still minimal height, $O(\log n)$, even using the classic insertion algorithm (that could potentially result in unbalanced trees). Intuitively, this is possible by inserting elements in this order: $median(1, n)$, $median(1, \frac{n}{2})$, $median(\frac{n}{2}, n)$, $median(1, \frac{n}{4})$, $median(\frac{n}{4}, \frac{n}{2})$, $median(\frac{n}{2}, \frac{3n}{4})$, $median(\frac{3n}{4}, n)$. Or, equivalently, $median(1, n)$, $median(1, \frac{n}{2})$, $median(1, \frac{n}{4})$, $median(\frac{n}{4}, \frac{n}{2})$, $median(\frac{n}{2}, n)$, $median(\frac{n}{2}, \frac{3n}{4})$, $median(\frac{3n}{4}, n)$. The input array can be sorted in this order by using the functions below:

```
SORT-FOR-BALANCED-BST(A)
sort A in non-descending order
PRINT-R(A, 1, A.length)
```

```
PRINT-R(A, i, j)
if i ≤ j
    m = ⌊(i + j)/2⌋
    print A[m]
    PRINT-R(A, i, m − 1)
    PRINT-R(A, m + 1, j)
```

PRINT-R runs in $O(n)$, since it simply prints one element—the median element, since the input is sorted—and then recurses on the left and side parts by excluding the element it just printed. In the end, PRINT-R runs (recursively) exactly once for each element of the array. So, the complexity of PRINT-R is $O(n)$ and the dominating cost for SORT-FOR-BALANCED-BST is the cost of sorting, which can be done in $O(n \log n)$.

▷ *Solution 158.1*

```
MINIMAL-SIMPLIFIED-SEQUENCE(A)
X = ∅
sort A in non-decreasing order
for i = A.length downto 3
    for j = A.length downto 3
        if BINARY-SEARCH(A[1...j − 1], A[i] − A[j]) ≠ TRUE
            X = X ∪ {A[i]}
return X
```

Hey, is the solutions above incorrect? An alternative solution is below:

```
MINIMAL-SIMPLIFIED-SEQUENCE(A)
X = ∅
sort A in non-decreasing order
for i = 1 to A.length − 1
    for j = i + 1 to A.length
        i = BINARY-SEARCH(A[j + 1 ... A.length], A[i] + A[j])
        if i > 0
            X = X ∪ {A[i]}
return X
```

The complexity is $\Theta(n^2 \log n)$.

▷ *Solution 158.2*

```
MINIMAL-SIMPLIFIED-SEQUENCE(A)
B = array of A.length zeroes
sort A in non-decreasing order
for i = 1 to A.length − 2
    j = i + 1
    k = i + 2
    while k ≤ A.length
        if A[k] − A[j] < A[i]
            k = k + 1
        elseif A[k] − A[j] > A[i]
            j = j + 1
        else B[k] = 1
            k = k + 1
X = ∅
for i = 1 to A.length
    if B[i] == 0
        X = X ∪ {A[i]}
return X
```

▷ *Solution 159*

The algorithm consists of two nested loops. The outer loop takes variable a from n to 1 by dividing a in half at every iteration. Therefore, the values of a are $n, n/2, n/4, n/8 \ldots$. That is, at iteration i of the outer loop, $a = n/2^i$. The outer loop terminates when $n/2^i \leq 1$, that is, it runs for $\lceil \log n \rceil$ iterations.

The inner loop takes variable b from 1 to a^2 by doubling b at every iteration. Therefore the values of b are $1, 2, 4, \ldots$, that is, $b = 2^j$ at the j-th iteration of the inner loop. Therefore the inner loop runs for $2 \log a$ iterations.

Altogether, the complexity is

$$\begin{aligned} T(n) &= \sum_{i=1}^{\lceil \log n \rceil} 2 \log (n/2^i) \\ &= \Theta(\log^2 n). \end{aligned}$$

▷ *Solution 160*

```
FIND-CYCLE(G)
N = array of size |V(G)|  // visited
P = array of size |V(G)|  // previous
for v ∈ V(G)
    N[v] = FALSE
    P[v] = NULL
for v ∈ V(G)
    if not N[v]
        N[v] = TRUE
        if FIND-CYCLE-R(N, P, v)
            return TRUE
return FALSE
```

```
FIND-CYCLE-R(N, P, v)
for w ∈ v.Adj
    if N[w]
        u = P[v]
        while u ≠ NULL
            if u == w
                return TRUE
            u = P[u]
    else N[w] = TRUE
        P[w] = v
        if FIND-CYCLE-R(N, P, w)
            return TRUE
return FALSE
```

▷ *Solution 161.1*

```
BFS-FIRST-COMMON-ANCESTOR(π, u, v)
S = array of size |π|
for i = 1 to |π|
    S[i] = 0
while u ≠ NULL or v ≠ NULL
    if u ≠ NULL
        if S[u] == 1
            return u
        else S[u] = 1
            u = π[u]
    if v ≠ NULL
        if S[v] == 1
            return v
        else S[v] = 1
            v = π[v]
return NULL
```

The time complexity is $\Theta(n)$. The space complexity is $\Theta(n)$.

▷ *Solution 161.2*

```
BFS-FIRST-COMMON-ANCESTOR-2(π, D, u, v)
if D[u] == ∞ or D[v] == ∞
    return NULL
while u ≠ v
    if D[u] > D[v]
        u = π[u]
    else v = π[v]
return u
```

The time complexity is $\Theta(n)$.

▷ *Solution 163.1*

```
BST-FIND-SUM(T, v)
t1 = BST-MIN(T)
while t1 ≠ NULL
    t2 = BST-SEARCH(T, v − t.key)
    if t2 ≠ NULL
        return t1, t2
    else
    else t1 = BST-SUCCESSOR(t1)
return NULL
```

The time complexity is $\Theta(n^2)$.

▷ *Solution 163.2*

```
BST-LOWER-BOUND(t, v)
// rightmost element whose key is ≤ v, or NULL
while t ≠ NULL
    if v < t.key
        t = t.left
    elseif t.right ≠ NULL and t.right.key < v
        t = t.right
    else return t
return NULL
```

```
BST-FIND-SUM(T, v)
t1 = BST-LOWER-BOUND(T, v/2)
t2 = BST-SUCCESSOR(t1)
while t1 ≠ NULL and t2 ≠ NULL
    if t1 + t2 = v
        return t1, t2
    elseif t1 + t2 < v
        t2 = BST-SUCCESSOR(t2)
    else t1 = BST-PREDECESSOR(t1)
return NULL
```

The time complexity is $\Theta(n)$.

▷ *Solution 164.1*

```
VERIFY-K-PAIRWISE-RELATIVELY-PRIME(X, k, S)
if S ⊈ X or |S| < k
    return FALSE
for i = 1 to |S| − 1
    for j = i + 1 to |S|
        if GCD(S[i], S[j]) > 1
            return FALSE
return TRUE
```

```
GCD(a, b)
while a ≠ b
    if a > b
        a = a % b
    else b = b % a
return a
```

The time complexity is $O(k \log n + k^2 \log m)$, where m is the maximum value in X.

▷ *Solution 166.1*

ALGO-X computes the product of all the elements of the input array A. ALGO-X effectively counts all the combinations in $\{0,\dots,A[1]\} \times \{0,\dots,A[2]\} \times \{0,\dots,A[3]\} \times \cdots \times \{0,\dots,A[n]\}$. Therefore, the complexity is $\Theta(A[1]A[2]\cdots A[n])$ or $O(m^n)$ where $m = \max A[i]$.

▷ *Solution 166.2*

BETTER-ALGO-X(A)

```
x = 1
for i = 1 to A.length
    x = x · A[i]
return x
```

The complexity of BETTER-ALGO-X is $\Theta(n)$.

▷ *Solution 171*

function	*rank*
$f_0(n) = n^{n^n}$	1
$f_1(n) = \log^2(n)$	7
$f_2(n) = n!$	2
$f_3(n) = \log(n^2)$	8
$f_4(n) = n$	6
$f_5(n) = \log(n!)$	5
$f_6(n) = \log\log n$	9
$f_7(n) = n\log n$	5
$f_8(n) = \sqrt{n^3}$	4
$f_9(n) = 2^n$	3

▷ *Solution 172*

MINIMAL-COVERING-SQUARE(P)

```
if P.length == 0
    return 0
left = P[1].x
right = P[1].x
top = P[1].y
bottom = P[1].y
for i = 2 to P.length
    if P[i].x > right
        right = P[i].x
    elseif P[i].x < left
        left = P[i].x
    if P[i].y > top
        top = P[i].y
    elseif P[i].y < bottom
        bottom = P[i].y
if right − left > top − bottom
    return (right − left)^2
else return (top − bottom)^2
```

▷ *Solution 173*

```
UNIMODAL-FIND-MAXIMUM(A)
l = 1
h = A.length
while l < h − 1
    m = ⌊(l + h)/2⌋
    if A[m − 1] > A[m]
        h = m
    elseif A[m + 1] > A[m]
        l = m
    else return A[m]
error "A is not a unimodal sequence"
```

▷ *Solution 174.1*

BETTER-ALGO-X(A, k) returns the beginning and ending position of a minimal subsequence of A that contains at least k equal elements. The complexity of BETTER-ALGO-X is $\Theta(n^4)$. In essence, this is because there are four nested loops.

▷ *Solution 174.2*

Notice that any minimal sequence $P[i], P[i+1], \ldots, P[j]$ that contains at least k equal elements contains *exactly* k elements equal to the first and last element of the sequence. Otherwise, the sequence $P[i], \ldots, P[j-1]$ would be a smaller sequence that still contains at least k equal elements. So, we just have to find a sequence that starts and ends with the same element x, and contains exactly k elements equal to x, including the first and last element:

```
BETTER-ALGO-X(A, k)
l = −∞
r = +∞
for i = 1 to A.length
    c = 1
    j = i + 1
    while c < k and j ≤ min(A.length, i + r − l)
        if A[i] == A[j]
            c = c + 1
        j = j + 1
    if c == k and r − l > j − i
        l = i
        r = j
return l, r
```

The complexity of BETTER-ALGO-X is $O(n^2)$.

▷ *Solution 175.1*

```
THREE-WAY-PARTITION(A, begin, end)
q1 = begin
q2 = q1 + 1
for i = q1 + 1 to end − 1
    if A[i] ≤ A[q1]
        swap A[i] ↔ A[q2]
        if A[q2] < A[q1]
            swap A[q2] ↔ A[q1]
            q1 = q1 + 1
        q2 = q2 + 1
return q1, q2
```

▷ *Solution 175.2*

```
QUICK-SORT(A)
QUICK-SORT-R(A, 1, A.length + 1)
```

```
QUICK-SORT-R(A, begin, end)
if begin < end
    q1, q2 = THREE-WAY-PARTITION(A, begin, end)
    QUICK-SORT-R(A, begin, q1)
    QUICK-SORT-R(A, q2, end)
```

This variant would be much more efficient with sequences often-repeated elements. In the extreme case of a sequence with n identical numbers, this variant would terminate in time $O(n)$, while the classic algorithm would run in time $O(n^2)$.

▷ *Solution 176*

S(A, s) returns TRUE if there is a subset of the elements in A that add up to s. This is also known as the *subset-sum* problem.
The best-case complexity is $O(n)$. An example of a best-case input (of size n) is with any array A and with $s = 0$. In this case, the algorithm recurses n times in line 3, only then to return TRUE from line 2 of the last recursion, and then unrolling all the recursions out of line 3 to ultimately return TRUE out of line 4.
The worst-case complexity is $O(2^n)$. A worst-case input (of size n) is one that leads to a FALSE result. An example would be an array A of positive numbers with $s < 0$. In this case, every invocation recurses twice, except for the base case. Each recursion reduces the size of the input range by 1, so the recursion tree amounts to a full binary tree with n levels, which leads to a complexity of $O(2^n)$.

▷ *Solution 177*

```
GROUP-OF-K(S_1, S_2, ..., S_m, k)
H = empty min-heap (sorted by time)
for i = 1 to m
    t, a = S_i[1]
    MIN-HEAP-INSERT(H, (t, a, i, 1)) (sorted by t)
C = dictionary mapping antennas to integers (hash map)
while H is not empty
    t, a, i, j = MIN-HEAP-EXTRACT-MIN(H)
    if j > 1
        t', a' = S_i[j - 1]
        C[a'] = C[a'] - 1
    if a ∈ C
        C[a] = C[a] + 1
    else C[a] = 1
    if C[a] ≥ k
        return t, a
    if j ≤ S_i.length
        t, a = S_i[j + 1]
        MIN-HEAP-INSERT(H, (t, a, i, j + 1)) (sorted by t)
return NULL
```

▷ *Solution 178.1*

ALGO-X(A) sorts the elements of A in-place so that all odd numbers precede all even numbers. I other words, ALGO-X(A) *partitions* A in two parts, $A[1 \dots j-1]$ and $A[j \dots A.length]$ so that $A[1 \dots j-1]$ contains only odd numbers and $A[j \dots A.length]$ contains only even numbers. One of the two parts might be empty. The complexity of ALGO-X is $\Theta(n^2)$.

▷ *Solution 178.2*

```
BETTER-ALGO-X(A)
i = 1
j = A.length + 1
while i < j
    if A[i] ≡ 0 mod 2 // A[i] is even
        j = j − 1
        swap A[i] ↔ A[j]
    else i = i + 1
return j
```

▷ *Solution 179*

```
BTREE-PRINT-RANGE(T, a, b)
if not T.leaf and T.key[1] > a
    BTREE-PRINT-RANGE(T.c[1], a, b)
for i = 1 to T.n
    if T.key[i] ≥ b
        return
    if T.key[i] > a
        print T.key[i]
    if not T.leaf
        if i == T.n or T.key[i + 1] > a
            BTREE-PRINT-RANGE(T.c[i + 1], a, b)
```

▷ *Solution 180*

The problem is in P, and therefore it is also in NP. This is a polynomial-time *solution* algorithm that proves it:

```
ALGO(G, k)
for v ∈ V(G)
    D_v = DIJKSTRA(G, v)
    // D_v is the distance vector resulting from Dijkstra
    for u ∈ V(G)
        if D_v[u] == k
            return TRUE
return FALSE
```

▷ *Solution 181*

```
MOST-CONGESTED-SEGMENT(A, ℓ)
sort A
i = 1
j = 1
x = NULL
m = 0
while j < A.length
    if A[j] − A[i] ≤ ℓ
        if m < j − i + 1
            x = A[i]
            m = j − i + 1
        j = j + 1
    else i = i + 1
return x
```

▷ *Solution 182*

The problem is in NP because a TRUE answer can be verified in polynomial time with a “certificate” consisting of a set of nodes $C = \{v_1, v_2, \ldots, v_\ell\}$

```
VERIFY(G, k, C = {v_1, v_2, ..., v_ℓ})
if |C| < k
    return FALSE
for all pairs u, v ∈ C
    if G[u][v] ≠ 1
        return FALSE
return TRUE
```

▷ *Solution 183*

```
MAX-HEAP-TOP-THREE(H)
if H.length < 4
    for i = 1 to H.length
        print H[i]
else print H[1]
    if H[2] > H[3]
        i = 2
        j = 3
    else i = 3
        j = 2
if H.length ≥ 2i + 1 and H[j] < H[2i + 1]
    j = 2i + 1
if H.length ≥ 2i and H[j] < H[2i]
    j = 2i
print H[i]
print H[j]
```

▷ *Solution 184*

```
LONGEST-STRETCH(P, h)
ℓ = 0
i = 1
while i < P.length
    a = P[i].y
    b = P[i].y
    j = i + 1
    while b − a < h
        if P[j].y > b
            b = P[j].y
        elseif P[j].y < a
            a = P[j].y
        if b − a < h
            if P[j].x − P[i].x > ℓ
                ℓ = P[j].x − P[i].x
        else j = j + 1
    i = i + 1
return ℓ
```

LONGEST-STRETCH(P, h) runs in $O(n^2)$ in the worst case. For example, a completely flat road would be a worst-case input.

▷ *Solution 185*

```
Is-Bipartite(G)
for v ∈ V(G)
    C[v] = GREEN // can be in either V_A or V_B
for v ∈ V(G)
    if C[v] == GREEN
        C[v] = RED
        Q = {v} // queue containing v
        while Q is not empty
            u = Dequeue(Q)
            for all w adjacent to u:
                if C[w] == GREEN
                    if C[u] == RED
                        C[w] = BLUE
                    else C[w] = RED
                    Enqueue(Q, w)
                elseif C[u] == C[w]
                    return FALSE
return TRUE
```

▷ *Solution 186.1*

```
Good-Are-Adjacent(A)
i = 1
while i < j and not Is-Good(A[i])
    i = i + 1
while i < j and not Is-Good(A[j])
    j = j − 1
while i < j
    if not Is-Good(A[i])
    else i = i + 1
return TRUE
```

▷ *Solution 186.2*

```
Make-Good-Adjacent(A)
i = 1
while i < j and not Is-Good(A[i])
    i = i + 1
while i < j and not Is-Good(A[j])
    j = j − 1
while i < j
    if not Is-Good(A[i])
        swap A[i] ↔ A[j]
        j = j − 1
    i = i + 1
return TRUE
```

▷ *Solution 187*

The problem is in P, and therefore also in NP. This is an algorithm that solves the problem in $O(n \log n)$ time.

```
GROUP-OF-EQUALS(A, k)
B = SORT(A)
i = 1
j = 1
while j < A.length
    if A[i] == A[j]
        j = j + 1
        if j − i == k
            return TRUE
    else i = j
return FALSE
```

▷ *Solution 188*

A simple, brute-force solution is to check each combination of positions in the two strings

```
MAXIMAL-COMMON-SUBSTRING(X, Y)
m = 0
for i = 1 to A.length
    for j = 1 to B.length
        ℓ = 0
        while i + ℓ ≤ A.length and j + ℓ ≤ B.length and A[i + ℓ] == B[j + ℓ]
            ℓ = ℓ + 1
        if ℓ > m
            m = ℓ
return m
```

The complexity of MAXIMAL-COMMON-SUBSTRING is $O(n^3)$.

▷ *Solution 189*

```
BST-COUNT-UNBALANCED-NODES(t)
if t == NULL
    return 0, 0
U_L, Tot_L = BST-COUNT-UNBALANCED-NODES(t.left)
U_R, Tot_R = BST-COUNT-UNBALANCED-NODES(t.right)
U = U_L + U_R
if Tot_L > 2Tot_R + 1 or Tot_R > 2Tot_L + 1
    U = U + 1
return U, (Tot_L + Tot_R + 1)
```

The complexity is $\Theta(n)$.

▷ *Solution 190.1*

ALGO-X returns the maximal difference between two values in an increasing sequence of elements in A. The complexity is $\Theta(n^3)$.

▷ *Solution 190.2*

```
LINEAR-ALGO-X(A)
x = 0
i = 0
j = 1
while j ≤ A.length
    if A[j] > A[j − 1]
        if A[j] − A[i] > x
            x = A[j] − A[i]
    else i = j
    j = j + 1
return x
```

▷ *Solution 191.1*

A naïve solution for FIND-SQUARE is to test all quadruples of points p_i, p_j, p_k, p_ℓ, and determine whether p_i, p_j, p_k, p_ℓ form a square.

```
FIND-SQUARE(P)
for i = 1 to P.length
    for j = 1 to P.length
        for k = 1 to P.length
            for ℓ = 1 to P.length
                d_x = P[j].x − P[i].x
                d_y = P[j].y − P[i].y
                if P[k].x == P[j].x + d_y and P[k].y == P[j].y − d_x
                and P[ℓ].x == P[i].x + d_y and P[ℓ].y == P[i].y − d_x
                    return TRUE
return FALSE
```

▷ *Solution 191.2*

Here the idea is to test all segments defined by two distinct points, and then to try to find the other corners of a square, which we can do with a binary search.

```
ORDER-2D(p_1, p_2)
if p_1.x < p_2.x
    return TRUE
elseif p_1.x > p_2.x
    return FALSE
elseif p_1.y < p_2.y
    return TRUE
else return FALSE
```

```
BINARY-SEARCH-2D(P, x, y)
i = 1
j = P.length
while i ≤ j
    m = ⌊(i + j)/2⌋
    if ORDER-2D(v, P[m])
        j = m − 1
    elseif P[m].x == x and P[m].y == y
        return TRUE
    else i = m + 1
return FALSE
```

```
FIND-SQUARE(P)
sort P using ORDER-2D as a comparison between pairs of points
for i = 1 to P.length
    for j = 1 to P.length
        d_x = P[j].x − P[i].x
        d_y = P[j].y − P[i].y
        if BINARY-SEARCH-2D(P, P[i].x + d_y, P[i].y − d_x)
        and BINARY-SEARCH-2D(P, P[j].x + d_y, P[j].y − d_x)
            return TRUE
return FALSE
```

▷ *Solution 192*

```
INITIALIZE(Q)
Q.A = new empty array
Q.P = new empty array
```

```
ENQUEUE(Q, obj, p)
append obj to array Q.A
append p to array Q.P
i = Q.P.length
j = ⌊i/2⌋
while i > 1 and Q.P[i] < Q.P[j]
    swap Q.P[i] ↔ Q.P[j]
    swap Q.A[i] ↔ Q.A[j]
    i = j
    j = ⌊i/2⌋
```

```
DEQUEUE(Q)
ℓ = A.P.length
if ℓ < 1
    error "empty queue"
x = Q.A[1]
swap Q.P[1] = Q.P[ℓ]
swap Q.A[1] = Q.A[ℓ]
remove last element from Q.P
remove last element from Q.A
ℓ = ℓ − 1
i = 1
while 2i ≤ ℓ and Q.P[i] > Q.P[2i]
or 2i + 1 ≤ ℓ and Q.P[i] > Q.P[2i + 1]
    if 2i + 1 ≤ ℓ and Q.P[2i + 1] > Q.P[2i]
        j = 2i + 1
    else j = 2i
        swap Q.P[i] = Q.P[j]
        swap Q.A[i] = Q.A[j]
        i = j
return x
```

▷ *Solution 193*

```
MAXIMAL-DISTANCE(A)
if A.length < 2
    return 0
min = A[1]
max = A[1]
for i = 2 to A.length
    if A[i] > max
        max = A[i]
    elseif A[i] < min
        min = A[i]
return max − min
```

▷ *Solution 194*

```
BST-HEIGHT(t)
if t == NULL
    return 0
return 1 + max(BST-HEIGHT(t.left), BST-HEIGHT(t.right))
```

▷ *Solution 195*

The problem, which is the well-known *matching* problem in graph theory, is definitely in NP. This is a possible verification algorithm:

```
VERIFY-MATCHING(G = (V, E, w), t, S)
X = ∅
for e = (u, v) ∈ S
    if u ∈ X or v ∈ X
        return FALSE
    X = X ∪ {u, v}
    weight = weight + w(e)
if weight ≥ t
    return TRUE
else return FALSE
```

▷ *Solution 196*

We can use a dynamic programming approach. Let P_i be the maximal value of the objects you can collect by reaching object i. Now, since you can reach P_i only by increasing your x and y coordinates, then that means that the maximal total value P_i is the value of object i plus the maximal total value when you reach any one of the objects from which you can then reach object i. This means all the objects with coordinates less than those of i. So:

$$P_i = V[i] + \max_{j \mid X[j] \leq X[i] \wedge Y[j] \leq Y[i]} P[j]$$

The global maximal game value is then $\max P_i$.

Now, the formula for P_i gives us a very simple recursive algorithm. This is inefficient, but it can be made very efficient with memoization.

```
MAXIMAL-GAME-VALUE(X, Y, V)
P = array of n = |V| elements initialized to P[i] = NULL
m = −∞
for i = 1 to V.length
    if m < MAXIMAL-VALUE-P(P, X, Y, V, i)
        m = MAXIMAL-VALUE-P(P, X, Y, V, i)
return m
```

```
MAXIMAL-VALUE-P(P, X, Y, V, i)
if P[i] == NULL
    P[i] = V[i]
    for j = 1 to V.length
        if j ≠ i and X[j] ≤ X[i] and Y[j] ≤ Y[i]
            if P[i] < V[i] + MAXIMAL-VALUE-P(P, X, Y, V, j)
                P[i] = V[i] + MAXIMAL-VALUE-P(P, X, Y, V, j)
return P[i]
```

▷ *Solution 197*

We are not required to be particularly efficient, so we can write a simple algorithm.

```
MAXIMAL-SUBSTRING(S)
A = NULL
for i = 1 to |S|
    X = ∅
    for j = 1 to |S[i]| − 1
        for k = j + 1 to |S[i]|
            X = X ∪ {S[i][j...k]}
    if A == NULL
        A = X
    else A = A ∩ X
        if A == ∅
            return “”
return longest string in A or “” if A == NULL
```

▷ *Solution 198.1*

ALGO-X returns the *mode* of A, meaning an element that occurs in A with maximal frequency (count). The complexity is $\Theta(n^2)$. Any input is the worst-case input.

▷ *Solution 198.2*

```
BETTER-ALGO-X(A)
B = copy of A
sort B
if |S| == 0
    return 0
x = B[1]
m = 1
c = 1
for i = 2 to |S|
    if B[i] == B[i − 1]
        c = c + 1
        if c > m
            m = c
            x = B[i]
    else c = 1
return x
```

▷ *Solution 199*

```
GRAPH-DEGREE(G)
n = |V(G)|
for i = 1 to n
    d = 0
    for j = 1 to n
        if G[i, j] == 1
            d = d + 1
    if d > m
        m = d
return m
```

▷ *Solution 200*

We don't have complexity constraints, so the algorithm can be simple:

```
FIND-3-CYCLE(G)
for u ∈ V(G)
    for v ∈ Adj[u]
        for w ∈ Adj[v]
            for x ∈ Adj[w]
                if x == u
                    return TRUE
return FALSE
```

The complexity is $\Theta(n\Delta^3)$, where Δ is the degree. Now, consider the full bipartite graph of $n/2$ plus $n/2$ nodes. In this case, the complexity is $\Theta(n^4)$.

▷ *Solution 201*

Here's an obvious $O(mn^2)$ solution:

```
LONGEST-COMMON-PREFIX(S)
m = 0
for i = 2 to S.length
    for j = 1 to i − 1
        ℓ = COMMON-PREFIX-LENGTH(S[i], S[j])
        if ℓ > m
            ℓ = m
return m
```

```
COMMON-PREFIX-LENGTH(a, b)
for i = 1 to min(a.length, b.length)
    if a[i] ≠ b[i]
        return i − 1
return min(a.length, b.length)
```

▷ *Solution 202*

Notice that if we sort the array S in lexicographical order, then any k strings with a common prefix will be contiguous in the sorted array.

```
LONGEST-K-COMMON-PREFIX(S, k)
sort S in lexicographical order
m = 0
for i = k to S.length
    ℓ = COMMON-PREFIX-LENGTH(S[i], S[i − k])
    if ℓ > m
        ℓ = m
return m
```

```
COMMON-PREFIX-LENGTH(a, b)
for i = 1 to min(a.length, b.length)
    if a[i] ≠ b[i]
        return i − 1
return min(a.length, b.length)
```

This complexity is $O(m \log n)$.

▷ *Solution 203*

The problem is in P and therefore it is also in NP. This is an algorithm that solves the problem in polynomial time:

```
NEGATIVE-THREE-CYCLE(G = (V, E))
for u ∈ V
    for v ∈ Adj[u]
        for w ∈ Adj[v]
            if w == u and weight(u, v) + weight(v, w) + weight(w, u) < 0
                return TRUE
return FALSE
```

▷ *Solution 204.1*

```
UPPER-BOUND(A, x)
u = UNDEFINED
d = UNDEFINED
for a ∈ A
    if x ≤ a
        if u == UNDEFINED or d > a − x
            d = a − x
            u = a
return u
```

The complexity is $\Theta(n)$.

▷ *Solution 204.2*

```
UPPER-BOUND-SORTED(A, x)
i = 1
j = A.length
if A[j] < x
    return UNDEFINED
elseif A[i] ≥ x
    return A[i]
while i < j
    m = ⌊i + j/2⌋
    if A[m] == x
        return A[m]
    elseif A[m] < x
        i = m
    else j = m
return A[j]
```

The complexity is $\Theta(\log n)$.

▷ *Solution 204.3*

```
UPPER-BOUND-BST(T, x)
while T ≠ NULL
    if T.key < x
        T = T.right
    else while T.left ≠ NULL and T.left.key ≥ x
            T = T.left
        return T.key
return UNDEFINED
```

The complexity is $\Theta(h)$ where h is the height of the input tree.

▷ *Solution 205*

```
SUM-OF-THREE(A, s)
B = MERGE-SORT(A)
for i = 1 to A.length
    j = 1
    k = A.length
    while j < k
        if j == i
            j = j + 1
        elseif k == i
            k = k - 1
        elseif B[i] + B[j] + B[k] == s
            return TRUE
        elseif B[i] + B[j] + B[k] > s
            k = k - 1
        else j = j + 1
return FALSE
```

▷ *Solution 206.1*

ALGO-X returns TRUE if and only if A contains two distinct numbers whose absolute difference is greater than x.

The complexity of ALGO-X is $\Theta(n^2)$. The worst case is when the algorithm reaches the last return statement in line 10. In this case, the loop amounts to an iteration over all pairs of distinct positions $j < i$. In fact, the loop starts with j and i at beginning and end of the array, respectively. Then the loop moves j forward by one step at a time until j reaches i, at which point j restarts from the beginning and i moves by one position to the left.

▷ *Solution 206.2*

```
BETTER-ALGO-X(A, x)
if A.length < 1
    return FALSE
low = A[1]
high = A[1]
for i = 2 to A.length
    if A[i] < low
        low = A[i]
    elseif A[i] > high
        high = A[i]
    if high - low > x
        return TRUE
return FALSE
```

BETTER-ALGO-X scans the array once looking for the maximum and minimum values, and exits as soon as it finds that the difference between the current (partial) maximum and minimum is greater than x.

▷ *Solution 207.1*

ALGO-S returns the value x in A such that there are exactly k elements less than x in A, or NULL if no such element exists.

The complexity of ALGO-S is $\Theta(n^2)$. The worst case is when the algorithm returns NULL. In this case, ALGO-S loops for exactly n times, each one costing the complexity of ALGO-R running on A, which is also n iterations.

▷ *Solution 207.2*

```
BETTER-ALGO-S(A, k)
B = MERGE-SORT(A)
if k ≥ 0 and k < B.length and B[k + 1] > B[k]
    return B[k + 1]
else return NULL
```

▷ *Solution 208*

```
SORT-SPECIAL(A)
q = A.length
while q > 1
    for i = 1 to q − 1
        if A[i] > A[q]
            swap A[i] ↔ A[q]
    i = 1
    while i < q
        if A[i] == A[q]
            swap A[i] ↔ A[q − 1]
            q = q − 1
        else i = i + 1
    q = q − 1
```

The idea here is to use a partitioning scan for each of the values in A, starting from the highest one and then down to the lowest one. More specifically, we first look for the maximum value v over the whole array. Then we partition A using v as the pivot. As a result, all the values equal to v are packed at the end of the array, and all the lower values are packed before that, up to position q. Now we repeat the process, only considering the sub-array from position 1 to q.

Each iteration consists of a linear scan to look for the maximum, plus another linear scan to perform the partitioning. And since there are at most 4 distinct values, we have at most 4 iterations. The complexity is therefore $O(n)$.

▷ *Solution 209*

```
HEAP-PROPERTIES(A)
max = 1
min = 1
for i = A.length downto 2
    p = ⌊i/2⌋
    if A[i] < A[p]
        min = 0
    elseif A[i] > A[p]
        max = 0
if min == 1
    if max == 1
        return 2
    else return −1
else if max == 1
        return 1
    else return 0
```

▷ *Solution 210*

We can find and then group, and thereby count, pairs of compatible objects as follows:

```
MAX-COMPATIBLE-PAIRING(A)
c = 0
i = 1
while i < A.length
    j = i + 1
    while j ≤ A.length and COMPATIBLE(A[i], A[j]) == FALSE
        j = j + 1
    if j ≤ A.length // we found a compatible pair (i, j)
        swap A[i + 1] ↔ A[j]
        i = i + 2
        c = c + 1
    else i = i + 1
return c
```

Another option is not to group but rather to simply count the pairs of equivalent elements. However, to avoid counting elements multiple times, we must somehow mark elements as being part of pair.

```
MAX-COMPATIBLE-PAIRING(A)
c = 0
B = [0, 0, ..., 0] // array of n Boolean values
i = 1
while i < A.length
    j = i + 1
    while j ≤ A.length and (B[j] == 1 or COMPATIBLE(A[i], A[j]) == FALSE)
        j = i + 1
    if j ≤ A.length // we found a compatible pair (i, j)
        B[j] = 1
        i = i + 1
        c = c + 1
    else i = i + 1
return c
```

▷ *Solution 211*

A queen in row i and column j attacks all the squares in row x and column y such that $x = i$, $y = j$, $x + y = i + j$, or $x - y = i - j$. Therefore, we can simply create an index for each of these four conditions for all the white queens, and then check whether a black queen is in any one of these indexes. The following solution creates indexes using sorted arrays. Another solution would be to use hash tables.

```
WHITE-ATTACKS-BLACK(W, B)
R = array of W.length integers
C = array of W.length integers
D1 = array of W.length integers
D2 = array of W.length integers
for i = 1 to W.length
    R[i] = W[i].row
    C[i] = W[i].col
    D1[i] = W[i].row − W[i].col
    D2[i] = W[i].row + W[i].col
sort R
sort C
sort D1
sort D2
for i = 1 to B.length
    if BINARY-SEARCH(R, B[i].row)
        return TRUE
    if BINARY-SEARCH(C, B[i].col)
        return TRUE
    if BINARY-SEARCH(D1, B[i].row − B[i].col)
        return TRUE
    if BINARY-SEARCH(D2, B[i].row + B[i].col)
        return TRUE
return FALSE
```

The complexity is $O(n \log n)$, since the sorting phase for the white queens takes $O(n \log n)$, and the lookup phase for the black queens also takes $O(n \log n)$.

▷ *Solution 212.1*

A very simple solution is the following recursive algorithm:

```
COUNT-FULL-NODES(t)
if t == NULL
    return 0
if t.left ≠ NULL and t.right ≠ NULL
    return 1 + COUNT-FULL-NODES(t.left) + COUNT-FULL-NODES(t.right)
else return COUNT-FULL-NODES(t.left) + COUNT-FULL-NODES(t.right)
```

The algorithm is equivalent to a walk of the tree, so it visits each node exactly once. The complexity is therefore $\Theta(n)$.

▷ *Solution 212.2*

The idea here is to walk through the nodes that are not full, and to rotate those that are full until they have a single child. For example, right rotate until they have only a right child. This leads to the following recursive solution.

```
NO-FULL-NODES(t)
if t == NULL
    return t
if t.left == NULL
    t.right = NO-FULL-NODES(t.right)
    return t
elseif t.right == NULL
    t.left = NO-FULL-NODES(t.left)
    return t
while t.left ≠ NULL
    t = RIGHT-ROTATE(t)
t.right = NO-FULL-NODES(t.right)
return t
```

The algorithm is also a tree walk for all non-full nodes. For each node t that is full, the algorithm performs some right rotations so as to move all the nodes in the subtree rooted at t from the left to the right side of t. This process iterates through those nodes at most once. So, in total, each node is touched either once or twice by the algorithm. The complexity is therefore $\Theta(n)$.

▷ *Solution 213.1*

The problem is in NP, since given a permutation Π of the indexes, it is easy to show (in polynomial time) that the answer is indeed TRUE with the following algorithm:

```
VERIFY-PERM-SUM-K(A, B, Π)
if A.length ≠ B.length
    return FALSE
n = A.length
A′ = array of n NULL values
for i = 1 to n
    A′[Π(i)] = A[i]
for i = 1 to n
    if A′[i] == NULL
        return FALSE
k = A′[1] + B[1]
for i = 2 to n
    if A′[i] + B[i] ≠ k
        return FALSE
return TRUE
```

This algorithm first checks that Π indeed defines a permutation on A, then it checks that the permutation satisfies the condition of the problem.
This question can also be immediately answered by solving the following exercise question, that is, showing that the problem is in P.

▷ *Solution 213.2*

The problem is in P, since it is easy to show that if A' exists, then it is also possible to sort A in increasing order, and correspondingly B in decreasing order so that the condition is satisfied. So, the following algorithm *solves* the problem, in polynomial time.

```
PERM-SUM-K(A, B)
if A.length ≠ B.length
    return FALSE
A′ = sort A
B′ = sort B in reverse order
k = A′[1] + B′[1]
for i = 2 to n
    if A′[i] + B′[i] ≠ k
        return FALSE
return TRUE
```

▷ *Solution 214*

There is a simple dynamic-programming solution. Let m_i be the minimal contiguous sub-sequence sum *ending* at position i. Then, we can obtain the minimal contiguous subsequence ending at $i+1$ by either connecting to the minimal contiguous subsequence ending at i, or by starting and ending a singleton subsequence at $i+1$. So, $m_{i+1} = \min\{m_i + A[i], A[i]\}$. In the case of the first element $A[1]$, the value of m_1 is simply $A[1]$. Then, from there we can compute all the other values, and remember the minimal value.

```
Minimal-Contiguous-Sum(A)
m = A[1]
p = A[1]
for i = 2 to A.length
    if p > 0
        p = A[i]
    else p = p + A[i]
    if p < m
        m = p
return m
```

▷ *Solution 215*

The idea is to find whether there is any path that can turn onto itself. We can do that using a simple depth-first search. We mark a node v as *visited* when we find v for the first time, and then we mark v as *finished* when we have explored all the nodes reachable from v. We have a cycle when, from the current node v we hit a neighbor u that is visited but not yet finished. This is because this means that v is reachable from u, and v is reachable from u (it is one of its neighbors), so we have a cycle.

```
Has-Cycle(G)
visited = ∅
finished = ∅
for all v ∈ V(G)
    if DFS-Find-Cycle(G, v)
        return TRUE
return FALSE
```

```
DFS-Find-Cycle(G, v)
global variable visited
global variable finished
if v ∈ visited
    if v ∉ finished
        return TRUE
    else return FALSE
visited = visited ∪ {v}
for all u ∈ G.Adj[v]
    if DFS-Find-Cycle(G, u)
        return TRUE
finished = finished ∪ {v}
return FALSE
```

▷ *Solution 216*

A DNA sequence S_1 is a permutation of another DNA sequence S_2 when S_1 contains exactly the same number of A's, the same number of C's, the same number of G's, and the same number of T's as S_2. So, it is easy to check that S_1 is a permutation of S_2 by counting and comparing the numbers of A's, C's, G's, and T's, respectively, in both sequences.
So, let $m = |X|$ be the length of the X sequence, then a basic idea would be to look at each substring $S[i, \dots, i+m-1]$ in S, and check whether $S[i, \dots, i+m-1]$ is a permutation of X.

```
DNA-Permutation-Substring(S, X)
x_A = number of A's in X
x_C = number of C's in X
x_G = number of G's in X
x_T = number of T's in X
for i = 1 to S.length − X.length + 1
    s_A = number of A's in S[i,...,i + X.length − 1]
    s_C = number of C's in S[i,...,i + X.length − 1]
    s_G = number of G's in S[i,...,i + X.length − 1]
    s_T = number of T's in S[i,...,i + X.length − 1]
    if x_A == s_A and x_C == s_C and x_G == s_G and x_T == s_T
        return TRUE
return FALSE
```

However, the complexity of this algorithm is $\Theta(\ell m)$, where $\ell = |S|$ and $m = |X|$. Instead, we want $O(n)$ where $n = \ell + m$.

We can do this with the following algorithm:

```
DNA-PERMUTATION-SUBSTRING(S, X)
// the following can be computed in time linear in |X|
x_A = number of A's in X
x_C = number of C's in X
x_G = number of G's in X
x_T = number of T's in X
// the following can be computed in time linear in |S|
let A, C, G, T be arrays of length n + 1
such that A[i] is the number of A's in the first i − 1 symbols of S
and correspondingly for C, G, and T arrays
for i = m + 1 to n + 1
    if x_A == A[i] − A[i − m] and x_C == C[i] − C[i − m]
    and x_G == G[i] − G[i − m] and x_T == T[i] − T[i − m]
        return TRUE
return FALSE
```

More in detail:

```
DNA-PERMUTATION-SUBSTRING(S, X)
n = S.length
m = X.length
x_A = 0
x_C = 0
x_G = 0
x_T = 0
for i = 1 to m
    if X[i] == 'A'
        x_a = x_a + 1
    elseif X[i] == 'C'
        x_C = x_C + 1
    elseif X[i] == 'G'
        x_G = x_G + 1
    elseif X[i] == 'T'
        x_T = x_T + 1
let A, C, G, T be arrays of length S.length + 1
A[1] = 0
C[1] = 0
G[1] = 0
T[1] = 0
for i = 2 to S.length + 1
    A[i] = A[i - 1]
    C[i] = C[i - 1]
    G[i] = G[i - 1]
    T[i] = T[i - 1]
    if S[i - 1] == 'A'
        A[i] = A[i] + 1
    elseif S[i - 1] == 'C'
        C[i] = C[i] + 1
    elseif S[i - 1] == 'G'
        G[i] = G[i] + 1
    elseif S[i - 1] == 'T'
        T[i] = T[i] + 1
for i = m + 1 to S.length + 1
    if x_A == A[i] - A[i - m] and x_C == C[i] - C[i - m]
    and x_G == G[i] - G[i - m] and x_T == T[i] - T[i - m]
        return TRUE
return FALSE
```

▷ *Solution 217.1*

ALGO-X returns the lowest, most common number in A. That is, the number x such that no other number appears more often than x in A, and if there is other number y that appear exactly the same number of times as x, then $x < y$. The complexity of ALGO-X is $\Theta(n^2)$.

▷ *Solution 217.2*

```
BETTER-ALGO-X(A)
B = copy of A sorted in ascending order
x = B[1]
m = 1
i = 2
while i < A.length
    j = i + 1
    while B[i] == B[j]
        j = j + 1
    if j − i > m
        m = j − 1
        x = B[i]
    i = j
return x
```

The complexity of BETTER-ALGO-X is $\Theta(n \log n)$, since the loop amounts to a linear scan of the sorted array B, so the dominating complexity is the time needed to sort the array at the beginning, which can be done in $\Theta(n \log n)$.

▷ *Solution 218.1*

```
BST-EQUALS(t1, t2)
if t1 == NIL and t2 == NIL
    return TRUE
elseif t1 == NIL or t2 == NIL
    return FALSE
if t1.key == t2.key and BST-EQUALS(t1.left, t2.left) and BST-EQUALS(t1.right, t2.right)
    return TRUE
else return FALSE
```

The algorithm amounts to a parallel walk of both trees. The complexity is therefore $O(n)$.

▷ *Solution 218.2*
We iterate through all the keys *in order* in both trees, and check them one by one.

```
BST-MIN-NODE(t)
if t == NIL
    return NIL
while t.left ≠ NIL
    t = t.left
return t
```

```
BST-SUCCESSOR(t)
if t == NIL
    return NIL
elseif t.right ≠ NIL
    return BST-MIN-NODE(t.right
else while t.parent ≠ NIL
        if t == t.parentleft
            return t.parent
        else t = t.parent
return NIL
```

```
BST-EQUAL-KEYS(t1, t2)
t1 = BST-MIN-NODE(t1)
t2 = BST-MIN-NODE(t2)
while t1 ≠ NIL and t2 ≠ NIL
    if t1.key ≠ t2.key
        return FALSE
    t1 = BST-SUCCESSOR(t1)
    t2 = BST-SUCCESSOR(t2)
if t1 == NIL and t2 == NIL
    return TRUE
else return FALSE
```

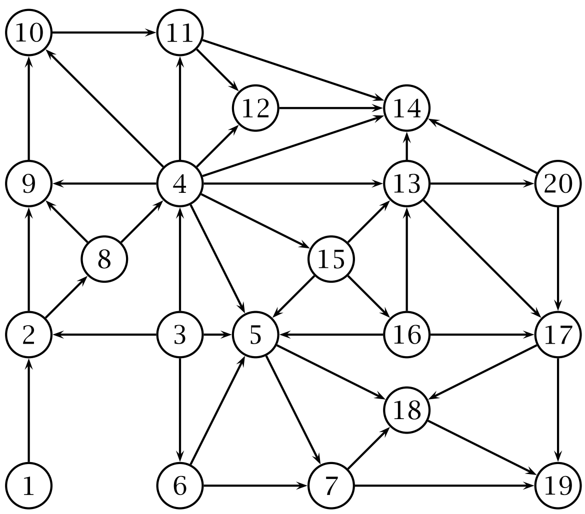
The algorithm amounts to a parallel walk of both trees. The complexity is therefore $O(n)$.

▷ *Solution 219*

```
KNIGHT-DISTANCE(r1, c1, r2, c2)
return KNIGHT-DISTANCE-ORIGIN(r1 − r2, c1 − c2)
```

```
KNIGHT-DISTANCE-ORIGIN(x, y)
M = global "memoization" matrix/dictionary initialized as follows:
M[0,0] = 0
M[0,1] = 3
M[0,2] = 2
M[1,1] = 2
M[1,2] = 1
M[2,2] = 4
any other value in M is initially NULL
if x < 0
    x = −x
if y < 0
    y = −y
if x > y
    swap x and y
if M[x, y] ≠ NULL
    return M[x, y]
d1 = KNIGHT-DISTANCE-ORIGIN(x − 2, y − 1) + 1
d2 = KNIGHT-DISTANCE-ORIGIN(x − 1, y − 2) + 1
M[x, y] = min(d1, d2)
return min(d1, d2)
```

▷ *Solution 220.1*



▷ *Solution 223.1*

ALGO-X sorts the input array in-place. In the best case, the algorithm terminates in the first execution of the outer loop, with the condition $s == \text{TRUE}$. This is the case when the inner loop does not swap a single element of the array, meaning that the array is already sorted. So, the best-case complexity is $O(n)$. Conversely, the worst case is when each iteration of the outer loop swaps at least one element. This happens when the array is sorted in reverse order. So, the worst-case complexity is $O(n^2)$.

▷ *Solution 223.2*

ALGO-Y sorts the input array in-place so that the value $v = A[0]$, that is the element originally at position 0, ends up in position q, and every other element less than v ends up somewhere in $A[1 \ldots q-1]$, that is to the left of q, and every other element less than or equal to v ends up somewhere in $A[q+1 \ldots |A|]$. In other words, ALGO-Y partitions the input array in-place using the first element as the "pivot". The loop closes the gap between i and j, which are initially the first and last position in the array, respectively. Each iteration either moves i to the right or j to the left, so each iteration reduces the gap by one. Therefore, in any case—worst case is the same as the best case—the complexity is $O(n)$.

▷ *Solution 224*

```
PARTITION-ZERO(A)
j = 1
for i = 1 to A.length
    if A[i] < 0
        swap A[i] ↔ A[j]
        j = j + 1
for i = j to A.length
    if A[i] == 0
        swap A[i] ↔ A[j]
        j = j + 1
```

▷ *Solution 225*

We can use a binary heap to implement a priority queue. In particular, we can use a max-heap where the heap property is based on the comparison between object priorities. Therefore we use an array Q as the base data structure, and we also keep track of the queue size $Q.size$, meaning the number of elements in the queue. Note that this is not the allocated size of the array $Q.size$.

```
PQ-INIT(n)
Q = new array of size n
Q.size = 0
Q.maxsize = n
return Q
```

```
PQ-ENQUEUE(Q, x)
if Q.size == Q.maxsize
    return "error: queue overflow"
Q[Q.size] = x
i = Q.size
Q.size = Q.size + 1
while i > 1 and Q[i] > Q[⌊i/2⌋]
    swap Q[i] ↔ Q[⌊i/2⌋]n
    i = ⌊i/2⌋
```

```
PQ-DEQUEUE(Q)
if Q.size == 0
    return "error: empty queue"
x = Q[1]
Q[1] = Q[Q.size]
Q.size = Q.size − 1
i = 1
while 2i ≤ Q.size
    j = i
    m = Q[i]
    if Q[2i] > m
        j = 2i
        m = Q[2i]
    if 2i + 1 ≤ Q.size and Q[2i + 1] > m
        j = 2i + 1
        m = Q[2i + 1]
    if j > i
        swap Q[i] ↔ Q[j]
        i = j
    else return x
return x
```

▷ *Solution 226.1*

ALGO-X returns TRUE if and only if B contains a subset of the elements in A. The complexity is $\Theta(n^2)$. A worst case input is one in which $A.length = B.length = n/2$ and none of the elements of A are contained in B. In this case, the outer loop (line 3) runs for $n/2$ iterations, and the inner loop also runs for $n/2$ times for each of the iterations of the outer loop.

▷ *Solution 226.2*

A different strategy is to first sort both vectors, and then to use an algorithm similar to a merge as below.

```
BETTER-ALGO-X(A, B)
C = sorted copy of array A
D = sorted copy of array B
n = D.length
j = 1
i = 1
while i ≤ C.length and j ≤ D.length
    if C[i] < D[j]
        i = i + 1
    elseif D[i] > D[j]
        j = j + 1
    else n = n − 1
        j = j + 1
        i = i + 1
if n == 0
    return TRUE
else return FALSE
```

▷ *Solution 227.1*

QUESTIONABLE-SORT is correct. It is also known as *selection-sort*. Effectively, the inner loop (j-loop) finds and leaves in position i a *minimal* element in $A[i \ldots n]$.

▷ *Solution 227.2*

Quick-sort or heap-sort would work. Here's quick-sort:

```
BETTER-SORT(A)
QUICK-SORT(A, 1, A.length)

QUICK-SORT(A, b, e)
if e − b > 0
    q = PARTITION(A, b, e)
    QUICK-SORT(A, b, q − 1)
    QUICK-SORT(A, q + 1, e)
```

```
PARTITION(A, b, e)
q = random position in [b, ..., e]
swap A[q] ↔ A[e]
q = b
i = b
for i = b to e
    if A[i] ≤ A[e]
        swap A[q] ↔ A[i]
        q = q + 1
return q − 1
```

▷ *Solution 228*

We can use a binary search.

```
LOWER-BOUND(A, x)
if x > A[A.length]
    return error: “not-found”
l = 1
r = A.length
while l < r
    m = ⌊(l + r)/2⌋
    if A[m] ≥ x
        r = m
    else l = m + 1
return A[r]
```

▷ *Solution 229*

```
CONTAINS-SQUARE(A)
ℓ = A.size
for i = 1 to ℓ − 1
    for j = 1 to ℓ − 1
        d = 1
        while i + d ≤ ℓ and j + d ≤ ℓ
            if IS-SQUARE(A, i, j, d)
                return TRUE
            d = d + 1
return FALSE
```

```
IS-SQUARE(A, i, j, d)
for k = 1 to d
    if A[i + k][j] ≠ A[i][j]
        return FALSE
    if A[i + k][j + d] ≠ A[i][j]
        return FALSE
    if A[i][j + k] ≠ A[i][j]
        return FALSE
    if A[i + d][j + k] ≠ A[i][j]
        return FALSE
return TRUE
```

The complexity of IS-SQUARE(A, i, j, d) is $\Theta(d)$. The complexity of CONTAINS-SQUARE(A) is therefore $O(n^4)$.

▷ *Solution 230*

```
MIN-HEAP-CHANGE(H, i, x)
if x < H[i]
    H[i] = x
    while i > 1 and H[⌊i/2⌋] > x
        swap H[⌊i/2⌋] ↔ H[i]
        i = ⌊i/2⌋
elseif x > H[i]
    H[i] = x
    j = MIN-OF-THREE(H, i)
    while i ≠ j
        swap H[i] ↔ H[j]
        j = MIN-OF-THREE(H, i)
```

```
MIN-OF-THREE(H, i)
m = H[i]
j = i
if 2i ≤ H.heap-size and H[2i] < m
    j = 2i
    m = H[2i]
if 2i + 1 ≤ H.heap-size and H[2i + 1] < m
    j = 2i + 1
return j
```

The complexity is $\Theta(\log n)$, since in the worst case we would start from a leaf and go all the way up to the root, or we would start from the root and go all the way down to a leaf.

▷ *Solution 231*

```
BST-Subset(T1, T2)
a = Min(T1)
b = Min(T2)
while b ≠ NIL and a ≠ NIL
    if a.key < b.key
        return FALSE
    elseif a.key > b.key
        b = Next(b)
    else a = Next(a)
if a == NIL
    return TRUE
else return FALSE
```

```
Min(t)
if t == NIL
    return NIL
while t.left ≠ NIL
    t = t.left
return t
```

```
Next(t)
if t == NIL
    return NIL
if t.right ≠ NIL
    return Min(t.right)
while t.parent ≠ NIL
and t == t.parent.right
    t = t.parent
return t.parent
```

The complexity is $\Theta(n)$, since we use Min and Next to effectively iterate over each tree as in a tree walk.

▷ *Solution 232*

This is a classic NP problem. We prove that by showing a polynomial-time verification algorithm. In particular, we show an algorithm Verify-Cycle(G, k, C) that takes an instance of the problem, that is, a graph G and a cycle length k, and a witness cycle C, and verifies that C is indeed a cycle in G of length k.

```
Verify-Cycle(G, k, C)
if C.length ≠ k
    return FALSE
for i = 1 to C.length − 1
    for j = i + 1 to C.length
        if C[i] == C[j]
            return FALSE
    if not Find-Neighbor(G, C[i], C[i + 1])
        return FALSE
if not Find-Neighbor(G, C[C.length], C[1])
    return FALSE
return TRUE
```

```
Find-Neighbor(G, u, v)
Adj = adjacency list of G
for w ∈ Adj[u]
    if w == v
        return TRUE
return FALSE
```

The complexity of Verify-Cycle is $O(n^2)$.

▷ *Solution 233*

This problem is in P. We prove that by showing an algorithm Find-Four-Cycle(G) that solves the problem in polynomial-time. In particular, the complexity of Find-Four-Cycle(G) is $O(n^4)$.

```
Find-Four-Cycle(G)
for a ∈ V(G)
    for b ∈ V(G)
        if b ≠ a and Find-Neighbor(G, a, b)
            for c ∈ V(G)
                if c ≠ b and c ≠ a and Find-Neighbor(G, b, c)
                    for d ∈ V(G)
                        if d ≠ c and d ≠ b and d ≠ a
                        and Find-Neighbor(G, c, d) and Find-Neighbor(G, d, a)
                            return TRUE
return FALSE
```

```
FIND-NEIGHBOR(G, u, v)
Adj = adjacency list of G
for w ∈ Adj[u]
    if w == v
        return TRUE
return FALSE
```

▷ *Solution 234*

Let DP(n) be the number of ways one can express n as a sum of ones, twos, and threes. Then we can immediately write a dynamic-programming recurrence as follows:

$$\mathrm{DP}(n) = \mathrm{DP}(n-1) + \mathrm{DP}(n-2) + \mathrm{DP}(n-3)$$

This is because n can be obtained by adding 1 to all the DP($n-1$) ways one can obtain $n-1$, or by adding 2 to all the DP($n-2$) ways one can obtain $n-2$, or by adding 3 to all the DP($n-3$) ways one can obtain $n-3$. We could then write SUMS-ONE-TWO-THREE(n) recursively as follows:

```
SUMS-ONE-TWO-THREE(n)
if n ≤ 0
    return 0
elseif n == 1
    return 1
elseif n == 2
    return 2 // 1 + 1, 2
elseif n == 3
    return 4 // 1 + 1 + 1, 2 + 1, 1 + 2, 3
else return SUMS-ONE-TWO-THREE(n − 1)
    +SUMS-ONE-TWO-THREE(n − 2)
    +SUMS-ONE-TWO-THREE(n − 3)
```

However, the complexity of this solution is most definitely not $O(n)$, since it looks a lot like the recursive version of FIBONACCI, and in fact we can use the same idea to make it efficient. The idea is to compute DP(i) from left to right, starting from the base cases:

```
SUMS-ONE-TWO-THREE(n)
if n ≤ 0
    return 0
elseif n == 1
    return 1
elseif n == 2
    return 2 // 1 + 1, 2
elseif n == 3
    return 4 // 1 + 1 + 1, 2 + 1, 1 + 2, 3
else a = 1
    b = 2
    c = 4
    r = a + b + c
    for i = 5 to n
        a = b
        b = c
        c = r
        r = a + b + c
    return r
```

▷ *Solution 235*

The most straightforward solution is one that simply tries all the partitions of $n = a + b$ into two integers a and b greater than 1 Since $n = a + b = b + a$, we can limit the search to $a \le b$.

```
TWO-PRIMES(n)
a = 2
while a ≤ n − a
      if IS-PRIME(a) and IS-PRIME(n − a)
            return TRUE
      a = a + 1
return FALSE
```

```
IS-PRIME(n)
i = 2
while i² ≤ n
      if n is divisible by i
            return TRUE
      i = i + 1
return FALSE
```

The main loop of TWO-PRIME runs for at most $n/2$ iterations, each costing $O(\sqrt{n})$, since the complexity of IS-PRIME(n) is $O(\sqrt{n})$. So, the overall complexity of TWO-PRIME is $O(n\sqrt{n})$.

▷ *Solution 236.1*

ALGO-X(A) returns the lowest category corresponding to the objects in A with a maximal total weight. This is the category c such that there is no other category $k < c$ such that the sum of all the objects in A with category k is higher than the sum of all the objects in A with category c. The complexity of ALGO-X(A) is $\Theta(n^2)$, since the algorithm consists of two nested loops over exactly n.

▷ *Solution 236.2*

We can create a copy of A that is sorted by category, which would allow us to compute the total weight for each category in a single linear pass.

```
BETTER-ALGO-X(A)
B = copy of A
sort B by category
w = −∞
t = B[1].weight
for i = 2 to B.length
      if B[i].category == B[i − 1].category
            t = t + B[i].weight
      else if t > w
                  c = B[i − 1].category
                  w = t
                  t = B[i].weight
if t > w
      c = B[B.length].category
return c
```

▷ *Solution 237.1*

```
MIN-HEAP-INSERT(H, x)
H.heap-size = H.heap-size + 1
i = H.heap-size
H[i] = x
while i > 1 and H[i] < H[⌊i/2⌋]
      swap H[i] ↔ H[⌊i/2⌋]
      i = ⌊i/2⌋
```

▷ *Solution 237.2*

```
MIN-HEAP-DEPTH(H)
i = 1
d = 0
while 2i ≤ H.heap-size
      i = 2i
      d = d + 1
return d
```

▷ *Solution 238.1*

ALGO-Y(A) returns the maximal sum of any pair of distinct elements in the input array. If there are less than 2 elements in the array, then the result is $-\infty$. The complexity of ALGO-Y(A) is $\Theta(n^2)$, since the algorithm iterates over all the $n(n-1)/2$ pairs of elements.

▷ *Solution 238.2*

The maximal sum of any two pairs of elements is simply the sum of the two highest values in A. So, we can simply find those two elements and then return their sum:

```
BETTER-ALGO-Y(A)
if A.length < 2
    return −∞
i = 1
for k = 2 to A.length
    if A[k] > A[i]
        i = k
if i == 1
    j = 2
else j = 1
for k = 1 to A.length
    if k ≠ i and A[k] > A[j]
        j = k
return A[i] + A[k]
```

▷ *Solution 239*

The problem is in P. As a proof, we show an algorithm MIN-K-SUM(A, m, k) that solves the problem in polynomial time. In fact, this is a greedy problem. In particular, we can answer the question by adding up the highest k values in A. If their total sum is greater or equal than m, then the result is clearly true. Otherwise, the result is clearly false, since there can not be another element that yields a larger sum.

```
MIN-K-SUM(A, m, k)
if k > A.length
    return FALSE
B = copy of A
sort B in descending order
s = 0
for i = 1 to k
    s = s + B[i]
if s ≥ m
    return TRUE
else return FALSE
```

The complexity of MIN-K-SUM is $O(n \log n)$.

▷ *Solution 240*

```
def maximal_step_k_length(A,k):
    m = 0
    j = 0
    for i in range(1,len(A)):
        if A[i] == A[i - 1] + k:
            if i - j + 1 > m:
                m = i - j + 1
        else:
            j = i
    i = 1
    j = 0
```

```
    for i in range(1,len(A)):
        if A[i] + k == A[i - 1]:
            if i - j + 1 > m:
                m = i - j + 1
        else:
            j = i
    return m
```

▷ *Solution 241*

```
def high_power_run(A,h,t):
    j = 0
    h_cur = 0 # value of sliding window
    for i in range(1, len(A)):
        if A[i] > A[i-1]:
            h_cur += A[i] - A[i-1]
        if i > j + t:
            j += 1
            if A[j] > A[j-1]:
                h_cur -= A[j] - A[j-1]
        if h_cur >= h:
            print(i, j)
            return True
    return False
```

▷ *Solution 242*

```
def peak_order(A):
    A.sort()
    i = len(A)//2
    j = len(A)-1
    while i < j:
        A[i], A[j] = A[j], A[i]
        i += 1
        j -= 1
```

▷ *Solution 243.1*

```
def rotate(A,k):
    n = len(A)
    k = k % n
    if k == 0:
        return;
    i = 0
    start = 0
    start_value = A[start]
    prev = 0
    curr = k
    while i < n:
        A[prev] = A[curr]
        i = i + 1
        prev = curr
        curr = (curr + k) % n

        if curr == start:
            A[prev] = start_value
```

```
            i = i + 1
            start = start + 1
            start_value = A[start]
            prev = start
            curr = (start + k) % n

def rotate_inplace(A,k):
    return rotate(A,k)
```

▷ *Solution 244*

```
def is_sorted(A):
    d = 0
    for i in range(1,len(A)):
        if A[i] > A[i-1]:
            if d < 0:
                return False
            d = 1
        elif A[i] < A[i-1]:
            if d > 0:
                return False
            d = -1
    return True
```

▷ *Solution 245.1*

```
def count_C(A):
    D = [1]*10
    for a in A:
        D[a % 10] += 1
    c = 1
    for d in D:
        c *= d
    return c - 1
```

▷ *Solution 245.2*

```
def print_C_r(D, S, i):
    if i == 10:
        if len(S) > 0:
            for s in S:
                print(s, end=' ')
            print()
    else:
        print_C_r(D, S, i+1)
        for d in D[i]:
            S.append(d)
            print_C_r(D, S, i+1)
            del S[-1]

def print_C(A):
    D = []
    for i in range(10):
        D.append([])

    for a in A:
```

```
        D[a % 10].append(a)

    print_C_r(D, [], 0)
```

▷ *Solution 246.1*
We first define a SCORE function to implement the rules for two numbers.

```
SCORE(a, b)
if a == b
    return 3
elseif a == b^2 or b == a^2
    return 9
elseif a divides b or b divides a
    return 5
else return 1
```

Then we compute the maximal score of two sequences with classic dynamic programming.

```
MAXIMAL-SCORE(A, B)
if A.length == 0 or B.length == 0
    return 0
return max{MAXIMAL-SCORE(A[2...], B[2...]) + SCORE(A[1], B[1]),
           MAXIMAL-SCORE(A[2...], B),
           MAXIMAL-SCORE(A, B[2...])}
```

This pure recursive solution is inefficient, but can be made efficient with memoization. In fact, in essence, this solution amounts to exploring a two-dimensional space of sub-problems, each defined by the length of the suffixes of A and B that are considered in the sub-problem. And again, this can be done implicitly through memoization, or explicitly by creating the matrix of sub-problems and by filling that matrix iteratively. As a further exercise, you might consider writing that solution. In any case, with memoization, the complexity is $O(n^2)$, where n is the total length of A and B.

▷ *Solution 246.2*
Here we simply implement the algorithm described in Question 1, with memoization.

```
def score(a,b):
    if a == b:
        return 3
    elif a == b*b or b == a*a:
        return 9
    elif a % b == 0 or b % a == 0:
        return 5
    else:
        return 1

def max3(x,y,z):
    if x < y:
        x = y
    if x < z:
        x = z
    return x

def DP(A,B,i,j,M):
    if i >= len(A) or j >= len(B):
        return 0
    if ((i,j) in M):    # if we already solved this problem, return the "memoized" solution
        return M[(i,j)]
    res = max3(DP(A,B,i+1,j+1,M) + score(A[i],B[j]),
```

```
            DP(A,B,i+1,j,M),
            DP(A,B,i,j+1,M))
    M[(i,j)] = res        # "memoize" the solution
    return res;

def maximal_score(A,B):
    return DP(A,B,0,0,{})
```

▷ *Solution 247*

The problem is in NP. We can prove that by showing a verification algorithm that, given $G = (V, E)$ and a number k, and a subset V_H, checks first of all that $|V_H| = k$ and then checks that the subgraph $H = (V_H, E_H)$ defined by V_H is indeed a tree.

```
VERIFICATION(G = (V, E), k, V_H)
if |V_H| ≠ k
    return 0
H = empty graph // We now build the subgraph H
for v ∈ V
    if v ∈ V_H
        V(H) = V(H) ∪ {v}
for e = (u, v) ∈ E
    if v ∈ V_H and u ∈ V_H
        E(H) = E(H) ∪ {(u, v)}
s = any vertex from V_H
D, P = BFS(H, s) // We check that H is connected using BFS
for v ∈ V_H
    if D[v] == ∞
        return 0
if |E(H)| == k − 1 // Lastly, we check that H has k − 1 edges
    return 1
else return 0
```

▷ *Solution 248.1*

ALGO-X considers all triples of distinct points $p_i, p_j, p_k \in P$, and returns true when vector $a = p_i p_j$ and vector $b = p_i p_k$ are orthogonal. This means that p_i, p_j, p_k form a right triangle. Thus ALGO-X returns TRUE if and only if P contains a right triangle. The complexity is $O(n^3)$, since there are $O(n^3)$ triples of points in P.

▷ *Solution 248.2*

```
BETTER-ALGO-X(P = [(x_1, y_1), (x_2, y_2), ..., (x_n, y_n)])
n = P.length
for i = 1 to n
    A = empty sequence
    for j = 1 to n
        if j ≠ i
            a_x = P[j].x − P[i].x
            a_y = P[j].y − P[i].y
            θ = angle of vector a = (a_x, a_y)
            add θ to A
    sort A
    for θ ∈ A
        if BINARY-SEARCH(A, θ + π/2) or BINARY-SEARCH(A, θ − π/2)
            return TRUE
return FALSE
```

The main idea is to consider, for each point $p_i \in P$, every other point $p_j \neq p_i$ and the direction θ_{ij} of the vector $a = p_i p_j$. The direction can be defined as the angle θ_{ij} between vector $a = p_i p_j$ and

the X-axis (or any fixed axis). In time $O(n \log n)$ we can build a sorted array A that contains all the angles. Then, for each angle $\theta \in A$, we can check with a binary search whether A also contains one of the two orthogonal angles $\theta_{ij} \pm \pi/2$. So again, the total cost for p_i is $O(n \log n)$. If we do this for every p_i, then the overall cost is $O(n^2 \log n)$.
Notice that the sorted array A is effectively a dictionary. So, a similar complexity can be obtained using any other type of efficient dictionary data structure.

▷ *Solution 249*
The most direct solution is conceptually similar to selection-sort. For each element $A[i]$, we iterate through every $A[j]$ in $A[i+1] \dots A[n]$, and we swap $A[j]$ close to $A[i]$.

```
CLUSTER(A)
i = 1
while i < A.length
    for j = i + 1 to A.length
        if EQUAL(A[i], A[j])
            i = i + 1
            swap A[i] ↔ A[j]
    i = i + 1
```

The worst case is when all objects are different, so the algorithm runs the two nested loops with a total of $n + (n-1) + (n-2) + \cdots + 2 + 1 = \Theta(n^2)$ iterations. Conversely, the best case is when all objects are equivalent, so the algorithm terminates after the first inner loop, with a complexity of $\Theta(n)$.

▷ *Solution 250.1*
Observe that danger periods do not overlap. Further, if a danger period is measured up to position $j-1$ (possibly empty), then the period can be extended to j if $M[j].\mathit{temperature} > M[j-1].\mathit{temperature}$ and $M[j].\mathit{humidity} < M[j-1].\mathit{humidity}$. Otherwise, the period ends at $j-1$ and a new one may start at j. This suggests a simple linear scan:

```
MAXIMAL-DANGER-PERIOD(A)
i = 1
m = 0
for j = 2 to M.length
    if M[j].temperature > M[j-1].temperature and M[j].humidity < M[j-1].humidity
        if M[j].time - M[i].time > m
            m = M[j].time - M[i].time
    else i = j
return m
```

The complexity is $O(n)$, since the algorithm amounts to a loop through the entire array M. The worst case is the same as the best case, since the loop is unconditional.

▷ *Solution 250.2*
We can simply translate the solution for Question 1 into a Python function:

```python
def max_danger_linear(M):
    i = 0
    m = 0
    for j in range(1,len(M)):
        if M[j].temperature > M[j-1].temperature and M[j].humidity < M[j-1].humidity:
            if M[j].time - M[i].time > m:
                m = M[j].time - M[i].time
        else:
            i = j
    return m
```

▷ *Solution 251.1*

ALGO-X(A, B, k) returns TRUE if and only if A and B contain a common subsequence of numbers of length k. More specifically, if there is a sequence $A[i], A[i+1], \ldots A[i+k]$ that is equal to $B[j], B[j+1], \ldots B[j+k]$ for some i and j.

▷ *Solution 251.2*

ALGO-Y(A, B) returns TRUE if and only if A and B are completely disjoint, meaning that there are no common elements in A and B. We can decide that in various ways. One is to merge the two sequences and return false as soon as we find equal elements.

```
BETTER-ALGO-Y(A, B)
X = sorted copy of A
Y = sorted copy of B
i = 1
j = 1
while i ≤ X.length and j ≤ Y.length
      if X[i] == Y[j]
            return FALSE
      elseif X[i] < Y[j]
            i = i + 1
      else j = j + 1
return TRUE
```

The complexity is $O(n \log n)$ because that is the cost of sorting A and B. The rest is a linear scan of the sorted sequences.

▷ *Solution 252.1*

This decision problem amounts to finding a group of nodes that are all mutually connected. Such a group is also called a *connected component* of the graph. This problem can be solved in polynomial time with a breadth-first search. So, not only the problem is in NP, it is also in P, and in fact it can be solved in *linear* time with the algorithm below. So we have an affirmative answer and a constructive proof for all the three questions of this exercise.

```
CONNECTED-COMPONENT(G, k)
S = ∅
for v ∈ V(G)
      if v ∉ S
            c = 1
            S = S ∪ {v}
            Q = queue containing only v
            while Q is not empty
                  v = pop first element from Q
                  for u ∈ Adj(v)
                        if u ∉ S
                              S = S ∪ {u}
                              add u to Q
                              c = c + 1
            if c ≥ k
                  return TRUE
return FALSE
```

▷ *Solution 253.1*

ALGO-X returns the sum of the top-k elements of A.

▷ *Solution 253.2*

The complexity is $\Theta(n \log n)$. The algorithm uses *merge-sort* as the main subroutine, plus a linear scan that is at most $\Theta(n)$. So the dominating complexity is the complexity of *merge-sort*, which is $\Theta(n \log n)$ and is the same in the worst and best case.

▷ *Solution 253.3*

We can use the same idea of the classic divide-and-conquer *k-selection* algorithm for order statistics: we partition using a chosen pivot, then recurse, at most once.

```
BETTER-ALGO-X(A, k)
if k ≥ A.length
    return SUM(A)
v = random value in A
L = empty sequence
M = empty sequence
R = empty sequence
for i = 1 to A.length
    if A[i] < v
        append A[i] to L
    elseif A[i] > v
        append A[i] to R
    else append A[i] to M
if k < L.length
    return BETTER-ALGO-X(L, k)
if k − L.length ≤ M.length
    return SUM(L) + (k − L.length) ∗ v
return SUM(L) + M.length ∗ v
        + BETTER-ALGO-X(R, k − L.length − M.length)
```

```
SUM(A)
s = 0
for i = 1 to A.length
    s = s + 1
return s
```

The algorithm is really the same as *k-selection*, so the complexity analysis is the same: the worst case is quadratic, but the average and most common case is linear.

▷ *Solution 254.1*

ALGO-X returns TRUE if and only if there are two distinct elements $A[i]$ and $[j]$ at distance x from each other, meaning $A[i] - A[j] = x$ (with $i \neq j$), or FALSE otherwise.

▷ *Solution 254.2*

ALGO-X essentially invokes a binary search (ALGO-Y) for each element of $A[i]$ in the remainder of the array. The best-case complexity is constant, which corresponds to an input array of size n in which the first element is $A[1] = y$, and there is an element $A[\lfloor n/2 \rfloor + 1] = y + x$. The worst-case complexity is instead $\Theta(n \log n)$, which corresponds to an input array that contains no to elements at distance x, for example, $A = [2, 4, 6, 8, 10, \ldots, 2n], x = 1$.

▷ *Solution 254.3*

Since A is sorted, we can find two elements $A[i]$ and $A[j]$ at distance $A[j] - A[i] = x$ with a linear scan. Again, since A is sorted, we simply advance the index of the higher (further) element when the distance is less than x (so as to increase the distance), or we advancing the base index i when the distance is higher than x (so as to decrease the distance):

```
BETTER-ALGO-X(A, k)
i = 0
j = 1
while j < A.length
    if A[j] < A[i] + x
        j = j + 1
    elseif A[j] > A[i] + x
        i = i + 1
    else return TRUE
return FALSE
```

The best-case complexity is constant, for example with $A = [1, 2, \ldots, n], x = 1$. The worst-case complexity is when we don't find two elements at distance x. For example, $A = [2, 4, \ldots, 2n], x = 1$.

▷ *Solution 255.1*

```
def count_vertical(A):
```

```
    #
    # Complexity: \Theta(n^2), since we go through all the pairs of
    # points.
    #
    n = len(A)//2
    c = 0
    for i in range(n):
        for j in range(i + 1, n):
            if A[2*i] == A[2*j]:
                c = c + 1
    return c

def count_horizontal(A):
    #
    # Complexity: \Theta(n^2), since we go through all the pairs of
    # points.
    #
    n = len(A)//2
    c = 0
    for i in range(n):
        for j in range(i + 1, n):
            if A[2*i+1] == A[2*j+1]:
                c = c + 1
    return c
```

▷ *Solution 255.2*

```
def intersection(A):
    #
    # Complexity: \Theta(n^4). Consider in fact the worst-case input:
    # A = [0,1,0,2,0,3,0,4,0,5,...,0,n].  In this case, we go through
    # the n(n-1)/2 vertical segments, and for each one of them we go
    # through each of the same n(n-1)/2 pairs of points looking for
    # intersecting horizontal segments.
    #
    n = len(A)//2
    for v1 in range(n):
        for v2 in range(v1+1,n):
            if A[2*v1] == A[2*v2]:
                x = A[2*v1]
                y1 = A[2*v1+1]
                y2 = A[2*v2+1]
                for h1 in range(n):
                    for h2 in range(h1+1,n):
                        if A[2*h1+1] == A[2*h2+1]:
                            y = A[2*h1+1]
                            x1 = A[2*h1]
                            x2 = A[2*h2]
                            if ((y >= y1 and y <= y2) or (y >= y2 and y <= y1)) \
                              and ((x >= x1 and x <= x2) or (x >= x2 and y <= x1)):
                                return True
    return False
```

▷ *Solution 256*

```
def increasing_or_decreasing(A):
```

```
#
# Complexity: \Theta(n). There are two loops of length n.
#
inc = 0
j = 0
for i in range(1,len(A)):
    if A[i] > A[i-1]:
        if i - j > inc:
            inc = i - j
    else:
        j = i
dec = 0
j = 0
for i in range(1,len(A)):
    if A[i] < A[i-1]:
        if i - j > dec:
            dec = i - j
    else:
        j = i
if inc > dec:
    return 'increasing'
elif dec > inc:
    return 'decreasing'
elif inc == 0:
    return 'flat'
else:
    return 'equal'
```

▷ *Solution 257.1*

ALGO-X sorts the input array so that all numbers that are equivalent to $0 \mod 4$ precede those that are equivalent to $1 \mod 4$ that precede those that are equivalent to $2 \mod 4$, and then those that are equivalent to $3 \mod 4$. The algorithm is essentially equivalent to insertion-sort, with this special ordering relation ($\mod 4$). The complexity is therefore $\Theta(n^2)$.

▷ *Solution 257.2*

LINEAR-ALGO-X(A)

```
m = 0
i = 1
while i < A.length
    j = A.length
    while i < j
        if A[i] == m mod 4
            i = i + 1
        elseif A[j] ≠ m mod 4
            j = j − 1
        else swap A[i] ↔ A[j]
            i = i + 1
            j = j − 1
    m = m + 1
```

▷ *Solution 258.1*

In essence, we are given a graph over the vertex set P, with an edge (p, q) whenever KNOWS(p, q) = TRUE. In this case, the decision problem asks whether there are at least k vertices each with at least ℓ neighbors. The problem is in P, since the question can be answered with a simple scan of the graph as follows:

```
SOLUTION(P, KNOWS, k, ℓ)
c = 0
for all p ∈ P
    d = 0
    for all q ∈ P
        if p ≠ q and KNOWS(p, q)
            d = d + 1
    if d ≥ ℓ
        c = c + 1
        if c ≥ k
            return TRUE
return FALSE
```

▷ *Solution 258.2*

Again we are given a graph over the vertex set P, with an edge (p,q) whenever KNOWS(p,q) = TRUE, and we are asked whether there is a set S of at least k vertices such that no two vertices in S are adjacent, meaning that no two persons in S have met each other. That is, for all $p,q \in S$, KNOWS(p,q) = FALSE. This definition immediately suggests a *verification* algorithm that proves that the problem is in NP. The verification algorithm takes the set S as a proof of a *true* answer.

```
VERIFICATION(P, KNOWS, k, S)
if |S| < k
    return FALSE
for all p ∈ P
    d = 0
    for all q ∈ P
        if p ≠ q and KNOWS(p, q)
            return FALSE
return TRUE
```

▷ *Solution 259.1*

ALGO-Y checks whether there are at least k pairs of distinct elements a_i, a_j in A (with $i \neq j$) such that $a_i = (a_j)^2$. The complexity is $\Theta(n^2)$.

▷ *Solution 259.2*

A straightforward $O(n \log n)$ solution is to sort the array and then, for each value, look for its square using binary search.

Another idea—just a bit more involved, but also more elegant (at least in the humble opinion of your teacher)—is to also sort the array, but then proceed with two linear scans. The scans use two indexes $i < j$ that move (linearly) to the right. The only problem is that we also need to consider *negative* numbers. For example, $a_i = 4, a_j = -2$ would be counted as a valid pair, but the linear right-ward scan would miss this case. However, we can simply have two scans: one for the positive numbers a_j where j moves to the right, and one for the negative numbers a_j where j moves to the left.

```
BETTER-ALGO-Y(A, k)
B = sorted copy of A
z = 1
while B[z] < 0      // we find the first non-negative number
    z = z + 1
i = z
j = i − 1           // here we consider negative numbers B[j] (if any)
while j > 0 and i ≤ B.length
    if B[j] · B[j] < B[i]
        j = j − 1
    elseif B[j] · B[j] > B[i]
        i = i + 1
    else k = k − 1
        if k == 0
            return TRUE
        i = i + 1
        j = j − 1
i = z
j = i
while i ≤ B.length
    if B[j] · B[j] < B[i]
        j = j + 1
    elseif B[j] · B[j] > B[i]
        i = i + 1
    else k = k − 1
        if k == 0
            return TRUE
        i = i + 1
        j = j + 1
return FALSE
```

▷ *Solution 260.1*

One way to solve this problem is to try every alignment between sequence A and the reverse of sequence B. For example, if $A = [3, 7, 4, 5, 7]$ and $B = [3, 7, 5, 4, 3]$, then we would try the following alignments:

A					3	7	4	5	7
B rev	3	4	5	7	3				
A				3	7	4	5	7	
B rev	3	4	5	7	3				
A			3	7	4	5	7		
B rev	3	4	5	7	3				
A		3	7	4	5	7			
B rev	3	4	5	7	3				
A	3	7	4	5	7				
B rev	3	4	5	7	3				
A	3	7	4	5	7				
B rev	3	4	5	7	3				
A	3	7	4	5	7				
B rev	3	4	5	7	3				
A	3	7	4	5	7				
B rev		3	4	5	7	3			
A	3	7	4	5	7				
B rev			3	4	5	7	3		
A	3	7	4	5	7				
B rev				3	4	5	7	3	
A	3	7	4	5	7				
B rev					3	4	5	7	3

and for each alignment, we then look for the longest contiguous sequence of equal values, which we can do easily in linear time. With this idea, it is also easy to figure out that we don't need to actually reverse the second sequence, and instead we simply need to go through its positions backwards. In fact, the more natural thing to do with the above alignments, as in the code below (next page), is to iterate forward in B and backwards in A. In any case, the complexity is $\Theta(n^2)$, since there are $O(n)$ alignments, and each one requires a linear $O(n)$ scan.

```
def longest_mirror(A,B):
    m = 0
    for p in range(1, len(A)+1): # p is a prefix length for A
        l = 0
        i = 0
        while i < p and i < len(B):
            if A[p − 1 − i] == B[i]:
                l = l + 1
                if l > m:
                    m = l
            else:
                l = 0
            i = i + 1
    for p in range(1, len(B)): # p is a suffix length for B
        l = 0
        i = p
        while i < len(B) and len(A) − 1 − i + p >= 0:
            if A[len(A) − 1 − i + p] == B[i]:
                l = l + 1
                if l > m:
                    m = l
            else:
                l = 0
            i = i + 1
    return m
```

Another way to solve the problem—again, more elegant, according to your teacher—is with dynamic programming. Below is the code. However, you should try to figure it out!

```
def longest_mirror(A,B):
    DP = [0]*len(A)
    m = 0
    for j in range(len(B)-1,-1,-1):
        for i in range(len(A)-1,-1,-1):
            if A[i] == B[j]:
                if i > 0:
                    DP[i] = DP[i-1] + 1
                else:
                    DP[i] = 1
                if DP[i] > m:
                    m = DP[i]
            else:
                DP[i] = 0
    return m
```

The complexity of this dynamic programming solution is also $\Theta(n^2)$, since we have two, fixed loops over A and B, which in the worst case can be of size $n/2$ each.

▷ *Solution 260.2*
See the solution for Question 1.

▷ *Solution 261*
The description of the algorithm already gives us a solution: for each day $i \in \{1, 2, \dots, n\}$ we compute the ranking between a_i, b_i, and c_i and check whether the ranking is different from that of the previous day.

```
COUNT-INVERSIONS(A, B, C)
r_prev = "null"
c = 0
for i = 1 to A.length
    if A[i] < B[i]
        if B[i] < C[i]
            r = "abc"
        elseif A[i] < C[i]
            r = "acb"
        else r = "cab"
    else if A[i] < C[i]
            r = "bac"
        elseif B[i] < C[i]
            r = "bca"
        else r = "cba"
    if r ≠ r_prev
        c = c + 1
    r_prev = r
return c
```

▷ *Solution 262*
We scan the input array A and store an array of the unique values contained in A. We return FALSE as soon as we find a fourth unique value. Below are two variants of this algorithm. The first one is completely self-contained. The second one uses an auxiliary FIND procedure.

```
AT-MOST-THREE-VALUES(A)
V = empty array
for i = 1 to A.length
    j = 1
    while j ≤ V.length and A[i] ≠ V[j]
        j = j + 1
    if j > 3
        return FALSE
    if j > V.length
        append A[i] to V
return TRUE
```

```
AT-MOST-THREE-VALUES(A)
V = empty array
for i = 1 to A.length
    if not FIND(V, A[i])
        if V.length < 3
            append A[i] to V
        else return FALSE
return TRUE
```

```
FIND(A, x)
for i = 1 to A.length
    if A[k] == x
        return TRUE
return FALSE
```

▷ *Solution 263.1*

ALGO-X checks whether all except at most two points in the time series given by A are on the same line. The worst-case complexity is $\Theta(n^3)$. This worst case corresponds to an input A in which all but the last two points lay on the same line. For example, $A = 1, 1, \ldots, 1, 2, 3$ would be a worst-case input.

▷ *Solution 263.2*

Our goal is to check that at least $n - 2$ points are on the same straight line. To check that a set of points are on the same line, we can take one reference point p, and then check that every other point q defines the same line (passing through p and q), meaning a line with the same slope $r = (y_q - y_p)/(x_q - x_p)$.

In this case, however, we must allow for at most two exceptions, and one of the exceptions could be our chosen reference point p. If the reference point p is one of the points on the line, then $n-3$ slope values out of the $n-1$ we compute—each defined by one of the $n-1$ remaining points—will be identical. If on the other hand p is one of the points that does not lay on the line, then most slope values will be different.

Since at most two points are not on the line, we just need to try at most three reference points. Any three points would do, so we use the first three.

```
BETTER-ALGO-X(A)
if A.length ≤ 4
    return TRUE
for i = 1 to 3
    if CHECK-LINE(A, i)
        return TRUE
return FALSE
```

```
CHECK-LINE(A, i)
V = empty array
C = empty array
for j = 1 to A.length
    if i ≠ j
        r = (A[k] − A[i])/(k − i)
        k = 1
        while k < V.length and V[k] ≠ r
            k = k + 1
        if k ≤ V.length
            C[k] = C[k] + 1
            if C[k] ≥ A.length − 3
                return TRUE
        elseif V.length ≥ 3
            return FALSE
        else append r to V
            append 1 to C
return FALSE
```

The core of the algorithm is in the CHECK-LINE(A, i) procedure, which uses point i as a reference point and then checks that all other points except possibly two of them form a line with the same slope. To implement CHECK-LINE we effectively maintain a *map* of at most three entries *slope* → *count* that associates a slope value with a count of points. We implement the map with two arrays, V and C, such that $V[k] \to C[k]$.

When we find a fourth slope value, we know that the result is FALSE. Conversely, when we find that one slope value has a count of $n - 3$, then the answer is TRUE.

The complexity of BETTER-ALGO-X is the complexity of CHECK-LINE, which is linear in the size of A.

▷ *Solution 263.3*

The solution for Question 2 has a linear complexity and is therefore also a solution for this question.

▷ *Solution 264*

In essence, $d(v)$ is the size of the set of nodes reachable from v by following the arcs of G in reverse. So, we first build the reverse adjacency list, meaning the adjacency list of the graph obtained by flipping the direction of all the arcs of G. With that graph, we then run a breadth-first search for each vertex v, where we count the nodes we reach. We then return the maximal count.

```
MAX-DEPENDENCIES(G = (V, Adj))
RAdj = array of n empty lists (n = |V|)
for all v ∈ V
    for all u ∈ Adj[v]
        append v to RAdj[u]
m = 0
for all v ∈ V
    d = 0
    S = {v}
    Q = queue containing v
    while Q is not empty
        u = dequeue node from Q
        for all w ∈ RAdj[u]
            if w ∉ S
                S = S ∪ {w}
                d = d + 1
                enqueue w into Q
    if d > m
        m = d
return m
```

The complexity of MAX-DEPENDENCIES is the complexity of a breadth-first search done for each vertex, so $\Theta(n(n+m))$, where n and m are the numbers of vertexes and arcs in G, respectively.

▷ *Solution 265*

```
MAX-HEAP-INSERT(H, x)
H.heap-size = H.heap-size + 1
i = H.heap-size
H[i] = x
while i > 1 and H[i] > H[⌊i/2⌋]
    swap H[i] ↔ H[⌊i/2⌋]
    i = ⌊i/2⌋
```

[3]
[7, 3]
[7, 3, 3]
[7, 3, 3, 2]
[9, 7, 3, 2, 3]
[9, 7, 5, 2, 3, 3]
[9, 7, 9, 2, 3, 3, 5]
[9, 8, 9, 7, 3, 3, 5, 2]
[9, 8, 9, 7, 3, 3, 5, 2, 5]
[9, 8, 9, 7, 3, 3, 5, 2, 5, 2]
[9, 9, 9, 7, 8, 3, 5, 2, 5, 2, 3]
[9, 9, 9, 7, 8, 4, 5, 2, 5, 2, 3, 3]
[9, 9, 9, 7, 8, 7, 5, 2, 5, 2, 3, 3, 4]
[9, 9, 9, 7, 8, 7, 5, 2, 5, 2, 3, 3, 4, 3]
[9, 9, 9, 7, 8, 7, 9, 2, 5, 2, 3, 3, 4, 3, 5]

▷ *Solution 266.1*
ALGO-X checks whether A contains an element $A[i]$ that is equal to the sum of all other elements in A.

▷ *Solution 266.2*
The worst-case complexity is $\Theta(n^2)$. In such a case, the algorithm goes through each one of the n elements, computes the sum of all the other $n-1$ elements in n steps, and then returns FALSE. The best-case complexity is instead $\Theta(n)$, which happens when the first element equals the sum of all other elements, which the algorithm computes in $\Theta(n)$ steps.

▷ *Solution 266.3*

If there is an element x such that the sum of every other element is x, then the total sum of all elements must be $2x$. So, we can simply compute the total sum s, in $\Theta(n)$ time, and then look for $s/2$ in A, also in $\Theta(n)$ time.

```
BETTER-ALGO-X(A)
s = 0
for i = 1 to A.length
    s = s + A[i]
for i = 1 to A.length
    if A[i] == s/2
        return TRUE
return FALSE
```

▷ *Solution 267.1*

ALGO-Y checks whether any two adjacent positions in the matrix contain equal elements. Adjacent means different positions whose column and row indexes differ by at most one.

▷ *Solution 267.2*

The complexity is $\Theta(n^2)$. The worst case is when there are no two equal elements, so the two loops go through all the $\binom{n}{2}$ pairs of elements, only to return FALSE at the end. Conversely, the best-case complexity is $O(1)$, which happens when the first two elements of the first row of the matrix are equal.

▷ *Solution 267.3*

For each element i, j in the matrix, which we denote here as $M_{i,j}$, there are at most 6 neighbors, namely $M_{i,j\pm1}$, $M_{i\pm1,j}$, and $M_{i\pm1,j\pm1}$. We can therefore scan all those pairs of adjacent positions in $\Theta(n)$ time. (Recall that the size of the matrix is $rc = n$.)

```
BETTER-ALGO-Y(A, r, c)
for i = 1 to r − 1
    for j = 1 to c
        if A[ic + j + 1] == A[(i + 1)c + j + 1] // M_{i,j} == M_{i+1,j}
            return TRUE
for i = 1 to r
    for j = 1 to c − 1
        if A[ic + j + 1] == A[ic + j + 2] // M_{i,j} == M_{i,j+1}
            return TRUE
for i = 1 to r − 1
    for j = 1 to c − 1
        if A[ic + j + 1] == A[(i + 1)c + j + 2] // M_{i,j} == M_{i+1,j+1}
            return TRUE
        if A[(i + 1)c + j + 1] == A[ic + j + 2] // M_{i+1,j} == M_{i,j+1}
            return TRUE
return FALSE
```

▷ *Solution 268*

The average $m = (A[n]+A[1])/2$ is such that either $m = A[1] = A[n]$, in which case the algorithm can immediately return $i = 1$ or $i = n$, or $A[1] < m < A[n]$ or $A[1] > m > A[n]$. In both these latter cases, we can proceed with a binary search. We just have to make sure that we run the binary search consistently with the specific relative order between $A[1]$ and $A[n]$.

```
FIND-AVG-POINT(A)
r = A.length
if A[1] == A[r]
    return 1
m = (A[r] + A[1])/2
ℓ = 1
while ℓ + 1 < r
    c = ⌊(ℓ + r + 1)/2⌋
    if A[c] > m
        if A[ℓ] > m
            ℓ = c
        else r = c
    elseif A[c] < m
        if A[ℓ] < m
            ℓ = c
        else r = c
    else return c
return ℓ
```

▷ *Solution 269*

The e-top order requires that all the elements in the even positions are less than or equal to all the elements in the odd positions. Since there are about $n/2$ even positions and $n/2$ odd positions in the array—more specifically, there are exactly $n/2$ even and $n/2$ odd positions if n is itself even, or $(n-1)/2$ even and $(n+1)/2$ odd positions if n is odd—the e-top order is equivalent to partitioning the array by the *median* value $m \in A$.

```
SORT-E-TOP(A)
n = A.length
if n is even
    k = n/2
else k = (n + 1)/2
m = SELECTION(A, k)
i = 1
j = 2
while i ≤ n or j ≤ n
    if A[i] ≤ m
        i = i + 2
    elseif A[j] > m
        j = j + 2
    else swap A[i] ↔ A[j]
        i = i + 2
        j = j + 2
```

```
SELECTION(A, k)
n = A.length
L = empty array
M = empty array
R = empty array
v = pick an element at random from A
for i = 1 to n
    if A[i] < v
        append A[i] to L
    elseif A[i] > v
        append A[i] to R
    else append A[i] to M
if k ≤ L.length
    return SELECTION(L, k)
elseif k ≤ L.length + M.length
    return v
else return SELECTION(R, k − L.length − M.length)
```

▷ *Solution 270*

```
BST-COUNT-IN-RANGE(T, a, b)
if T == NIL
    return 0
if a ≤ T.key and b ≥ T.key
    return 1 + BST-COUNT-IN-RANGE(T.left, a, b) + BST-COUNT-IN-RANGE(T.right, a, b)
if b < T.key
    return BST-COUNT-IN-RANGE(T.left, a, b)
else return BST-COUNT-IN-RANGE(T.right, a, b)
```

In the worst case, we have to count all the nodes in T. So the complexity is $\Theta(n)$. In the best case, the root key $T.key$ is the minimum (and therefore $T.left$ is NIL) and the given range $[a, b]$ is to the left of that key, so the algorithm terminates immediately after one recursion into $T.left$, and therefore the complexity is $O(1)$. Same thing in the other direction.

▷ *Solution 271.1*

We start from the base-station position, and we discover all nodes within radius r of that position, then we do the same from every discovered node until we do not discover more nodes. If this process discovers all nodes, then we return TRUE. Otherwise, we return FALSE.

```
CHECK-CONNECTIVITY(X, Y, r)
n = X.length
D = [FALSE] * n        // D[i] indicates whether sensor i was discovered
Q = empty queue
enqueue coordinates (0, 0) into Q
while Q is not empty
    (x, y) = dequeue coordinates from Q
    for i = 1 to n
        if D[i] == FALSE and (X[i] − x)^2 + (Y[i] − y)^2 ≤ r
            D[i] = TRUE
            enqueue coordinates (X[i], Y[i]) into Q
for i = 1 to n
    if D[i] == FALSE
        return FALSE
return TRUE
```

The worst-case complexity is $\Theta(n^2)$, since that is what the algorithm costs when all sensors are discovered. In this case, all nodes are added to the queue Q and therefore processed by the main loop exactly once. With each iteration of the main loop, we consider a sensor at some coordinates (x, y), and we then scan the entire set of sensors to see which other sensors are within range of the sensor at (x, y). Thus the complexity is $\Theta(n^2)$.

▷ *Solution 271.2*

The minimal connectivity range is at most equal to the maximal distance of any sensor from the base station. This is because, with that range, each sensor would be directly connected to the base station. We set that distance as r_{max}, and then use CHECK-CONNECTIVITY(X, Y, r) to perform a binary search on the result r. The binary search works because, by definition, CHECK-CONNECTIVITY(X, Y, r) returns TRUE for any value r greater or equal to the minimal connectivity range $\overline{r}$, and FALSE for any value r less than the minimal connectivity range $\overline{r}$.

```
MINIMAL-CONNECTIVITY-RANGE(X, Y, t)
r_max = 0
for i = 1 to X.length        // note that X.length = Y.length = n
    if sqrt((X[i])^2 + (Y[i])^2) > r_max
        r_max = sqrt((X[i])^2 + (Y[i])^2)
r_min = 0
while r_max − r_min > t
    r = (r_max + r_min)/2
    if CHECK-CONNECTIVITY(X, Y, r)
        r_max = r
    else r_min = r
return (r_max + r_min)/2
```

The complexity is given by the numeric values of the maximal distance r_{max} and the threshold t. More specifically, starting from a range r_{max}, we divide by 2 repeatedly until we reach t. Therefore, the while-loop runs for $\log_2 (r_{max}/t)$ iterations. At each iteration, we invoke CHECK-CONNECTIVITY, which costs us $\Theta(n^2)$. Therefore, the complexity is $\Theta(n^2 \log (r_{max}/t))$

▷ *Solution 272.1*

The problem is in NP. We prove that by showing an algorithm that *verifies* a certificate for a "yes" answer in polynomial time. As a certificate, we give the verification algorithm $2k$ indexes $I = [i_1, i_2, \ldots, i_{2k-1}, i_{2k}]$ that define k pairs.

```
VERIFY(k, A, I)
if I.length ≠ 2k
    return FALSE
for p = 1 to k − 1
    for q = p + 1 to 2k
        if I[p] == I[q]
            return FALSE
for p = 1 to k − 1
    if A[I[2p + 1]] + A[I[2p + 2]] ≠ A[I[1]] + A[I[2]]
        return FALSE
return TRUE
```

▷ *Solution 272.2*

The problem is in P. We prove that by showing an algorithm that *solves* the problem in polynomial time. The main idea of this algorithm is to compute the values of all pairs, and then to find repeated values. We find repeated values by sorting the array of values, and then by counting consecutive equal values.

```
SOLVE(k, A)
B = empty array
for i = 1 to A.length − 1
    for j = i + 1 to A.length
        append A[j] + A[i] to B
sort B
j = 1
for i = 2 to B.length
    if B[i] ≠ B[j]
        j = i
    if i − j + 1 ≥ k
        return TRUE
return FALSE
```

▷ *Solution 273.1*

ALGO-X returns TRUE if A and B contain exactly the same elements, in any order, or FALSE otherwise. The worst-case is when A contains distinct elements, and B contains exactly the same elements, but in reverse order. In this case the complexity is $\Theta(n^2)$. In the best case, the length of A is a small fixed value (independent of the total length n) and the algorithm returns FALSE after the first iteration of the inner while-loop. In this case, the complexity if $O(1)$.

▷ *Solution 273.2*

We must compare the sequences as multi-sets. We do that by first sorting the two sequences, so that we can then compare the sequences element-by-element in each position.

```
BETTER-ALGO-X(A, B)
if A.length ≠ B.length
    return FALSE
C = sorted copy of A
D = sorted copy of B
for i = 1 to A.length
    if C[i] ≠ D[j]
        return FALSE
return TRUE
```

▷ *Solution 274*

```
BST-COUNT-OUTSIDE-RANGE(T, a, b)
return BST-COUNT-IN-RANGE(T, −∞, a) + BST-COUNT-IN-RANGE(T, b, ∞)
```

```
BST-Count-In-Range(T, a, b)
if T == NIL
    return 0
if a < T.key and b > T.key
    return 1 + BST-Count-In-Range(T.left, a, b) + BST-Count-In-Range(T.right, a, b)
if b ≤ T.key
    return BST-Count-In-Range(T.left, a, b)
else return BST-Count-In-Range(T.right, a, b)
```

In the worst case, we have to count all the nodes in T. So the complexity is $\Theta(n)$. In the best case, the root key $T.key$ is the minimum (and therefore $T.left$ is NIL), $T.right.key$ is the maximum (and therefore $T.right.right$ is NIL), and the minimum and the maximum keys are both in the interval $[a, b]$. In this case, the first call to BST-Count-In-Range$(T, -\infty, a)$ recurses to $T.left$ so the algorithm terminates immediately after one recursion into $T.left$ and returns immediately (since $T.left$ is NIL), and the second call BST-Count-In-Range(T, b, ∞) recurses only to $T.right$ and then to $T.right.right$.

▷ *Solution 275.1*
The problem is in NP. We prove that by showing an algorithm that *verifies* a certificate for a "yes" answer in polynomial time. As a certificate, we give two users a and b for which we verify that their distance in the social network graph is greater than n.

```
Verify(U, F, n, a, b)
D = BFS(U, F, a)
...
```

▷ *Solution 275.2*
The problem is in P. We prove that by showing an algorithm that *solves* the problem in polynomial time.

```
Solve(U, F, n)
for u ∈ U
    D = BFS(U, F, u)
    for v ∈ U
        if D[v] > n
            return TRUE
```

▷ *Solution 276.1*
Algo-X returns the maximal length of any contiguous subsequence of A whose total value is equal to an element of B. If no such sequence exists, the result is 0.
The worst-case is when the result is 0, which happens when the algorithm iterates through its four nested loops. So, intuitively, the complexity is $O(n^4)$. As it turns out, a slightly more involved analysis shows that this intuitive upper bound is also tight.
The algorithm effectively iterates over all the pairs (S_A, b) where S_A is a contiguous subsequence of A, and b is an element of B. For each of the $n_A - \ell + 1$ subsequences of length ℓ, the algorithm computes the sum of the subsequence in ℓ steps. So, the overall cost of the computations of all the subsequences is $1(n_A) + 2(n_A - 1) + 3(n_A - 2) + \cdots + (n_A - 1)(2) + (n_A)(1)$. Intuitively, this sum is $O((n_A)^3)$ and $\Omega((n_A)^2)$. It is also possible to show, for example using a geometric argument, that the $O((n_A)^3)$ upper bound is also tight, so the overall complexity for the sums of all the subsequences is $\Theta((n_A)^3)$. And the algorithm runs this for every element of B, so the overall complexity is $\Theta(n_B(n_A)^3)$. And since we can choose $n_A = n/2$ and $n_B = n/2$, the total complexity is $\Theta(n^4)$.

▷ *Solution 276.2*
We can get an immediate improvement of a factor of n by simply computing the sums of the various subsequences incrementally. That is, once you have a sum of a subsequence of length ℓ starting at position i, you can get the next sequence starting at position $i + 1$ in constant time by adding the element at position $i + \ell$ and subtracting the element at position i.

```
BETTER-ALGO-X(A, B)
for ℓ = A.length downto 1
    for j = 1 to B.length
        s = 0
        for k = 1 to ℓ
            s = s + A[k]
        if s == B[j]
            return ℓ
        for i = 2 to A.length − ℓ + 1
            s = s − A[i] + A[i + ℓ − 1]
            if s == B[j]
                return ℓ
return 0
```

▷ *Solution 277.1*

ALGO-Y prints in ascending order all the elements of A whose occurrence count is maximal. The complexity is $\Theta(n^2)$. Any input is the worst-case input. This is because the two nested loops that determine the $\Theta(n^2)$ are executed to completion in all cases.

▷ *Solution 277.2*

```
BETTER-ALGO-Y(A)
B = copy of A
sort B
if B.length == 0
    return 0
m = 1
c = 1
for i = 2 to B.length
    if B[i] == B[i − 1]
        c = c + 1
        if c > m
            m = c
    else c = 1
c = 1
for i = 1 to B.length
    if i > 1 and B[i] == B[i − 1]
        c = c + 1
    else c = 1
    if c == m
        print B[i]
```

The complexity of BETTER-ALGO-Y is dominated by the complexity of sorting the input array A, so $\Theta(n \log n)$. The rest of the algorithm amounts to two linear scans, so $\Theta(n)$.

▷ *Solution 278*

```
BST-COUNT-OUTSIDE-RANGE(T, a, b)
return BST-COUNT-IN-RANGE(T, −∞, a) + BST-COUNT-IN-RANGE(T, b, ∞)
```

```
BST-COUNT-IN-RANGE(T, a, b)
if T == NIL
    return 0
if a < T.key and b > T.key
    return 1 + BST-COUNT-IN-RANGE(T.left, a, b) + BST-COUNT-IN-RANGE(T.right, a, b)
if b ≤ T.key
    return BST-COUNT-IN-RANGE(T.left, a, b)
else return BST-COUNT-IN-RANGE(T.right, a, b)
```

In the worst case, we have to count all the nodes in T. So the complexity is $\Theta(n)$. In the best case, the root key $T.key$ is the minimum (and therefore $T.left$ is NIL), $T.right.key$ is the maximum (and therefore $T.right.right$ is NIL), and the minimum and the maximum keys are both in the interval $[a, b]$. In this case, the first call to BST-COUNT-IN-RANGE$(T, -\infty, a)$ recurses to $T.left$ so the algorithm terminates immediately after one recursion into $T.left$ and returns immediately (since $T.left$ is NIL), and the second call BST-COUNT-IN-RANGE(T, b, ∞) recurses only to $T.right$ and then to $T.right.right$.

▷ *Solution 279.1*

The problem is in NP. We prove that by showing an algorithm that *verifies* a certificate for a "yes" answer in polynomial time. As a certificate, we give two users a and b for which we verify that their distance in the social network graph is greater than n.

```
VERIFY(U, F, n, a, b)
D = BFS(U, F, a)
...
```

▷ *Solution 279.2*

The problem is in P. We prove that by showing an algorithm that *solves* the problem in polynomial time.

```
SOLVE(U, F, n)
for u ∈ U
    D = BFS(U, F, u)
    for v ∈ U
        if D[v] > n
            return TRUE
```

▷ *Solution 280.1*

ALGO-X returns the maximal length of any contiguous subsequence of A whose total value is equal to an element of B. If no such sequence exists, the result is 0.

The worst-case is when the result is 0, which happens when the algorithm iterates through its four nested loops. So, intuitively, the complexity is $O(n^4)$. As it turns out, a slightly more involved analysis shows that this intuitive upper bound is also tight.

The algorithm effectively iterates over all the pairs (S_A, b) where S_A is a contiguous subsequence of A, and b is an element of B. For each of the $n_A - \ell + 1$ subsequences of length ℓ, the algorithm computes the sum of the subsequence in ℓ steps. So, the overall cost of the computations of all the subsequences is $1(n_A) + 2(n_A - 1) + 3(n_A - 2) + \cdots + (n_A - 1)(2) + (n_A)(1)$. Intuitively, this sum is $O((n_A)^3)$ and $\Omega((n_A)^2)$. It is also possible to show, for example using a geometric argument, that the $O((n_A)^3)$ upper bound is also tight, so the overall complexity for the sums of all the subsequences is $\Theta((n_A)^3)$. And the algorithm runs this for every element of B, so the overall complexity is $\Theta(n_B(n_A)^3)$. And since we can choose $n_A = n/2$ and $n_B = n/2$, the total complexity is $\Theta(n^4)$.

▷ *Solution 280.2*

We can get an immediate improvement of a factor of n by simply computing the sums of the various subsequences incrementally. That is, once you have a sum of a subsequence of length ℓ starting at position i, you can get the next sequence starting at position $i + 1$ in constant time by adding the element at position $i + \ell$ and subtracting the element at position i.

```
BETTER-ALGO-X(A, B)
for ℓ = A.length downto 1
    for j = 1 to B.length
        s = 0
        for k = 1 to ℓ
            s = s + A[k]
        if s == B[j]
            return ℓ
        for i = 2 to A.length − ℓ + 1
            s = s − A[i] + A[i + ℓ − 1]
            if s == B[j]
                return ℓ
return 0
```

▷ *Solution 281.1*

ALGO-Y prints in ascending order all the elements of A whose occurrence count is maximal. The complexity is $\Theta(n^2)$. Any input is the worst-case input. This is because the two nested loops that determine the $\Theta(n^2)$ are executed to completion in all cases.

▷ *Solution 281.2*

```
BETTER-ALGO-Y(A)
B = copy of A
sort B
if B.length == 0
    return 0
m = 1
c = 1
for i = 2 to B.length
    if B[i] == B[i − 1]
        c = c + 1
        if c > m
            m = c
    else c = 1
c = 1
for i = 1 to B.length
    if i > 1 and B[i] == B[i − 1]
        c = c + 1
    else c = 1
    if c == m
        print B[i]
```

The complexity of BETTER-ALGO-Y is dominated by the complexity of sorting the input array A, so $\Theta(n \log n)$. The rest of the algorithm amounts to two linear scans, so $\Theta(n)$.

▷ *Solution 282*

```
Compare-Intervals(a1, b1, a2, b2)
if a1 > b1
    swap a1 ↔ b1
if a2 > b2
    swap a2 ↔ b2
if a2 > b1 or a1 > b2
    return "disjoint"
if a1 == a2 and b1 == b2
    return "1 equals 2"
if a1 < a2
    if b2 ≤ b1
        return "1 covers 2"
    elseif a2 < b1
        return "partial"
    else return "touch"
elseif b2 ≤ b1
    return "2 covers 1"
elseif b2 > a1
    return "partial"
else return "touch"
```

The complexity is constant, $O(1)$.

▷ *Solution 283.1*

The problem is in P, as we show in the answer for Question 2. Therefore, the problem is also in NP. We can also prove that the problem is in NP by showing an algorithm that *verifies* a given pairing. In particular, we give a "witness" pairing as an array $P = [(i_1, j_1), (i_2, j_2), \ldots, (i_n, j_n)]$ of n pairs of indexes into A.

```
Verify-Uniform-Pairing(A, P)
n = A.length/2
if P.length ≠ n
    return FALSE
v = A[P[1][1]] + A[P[1][2]]
for i = 2 to n
    if A[P[i][1]] + A[P[i][2]] ≠ v
        return FALSE
I = empty array
for i = 1 to n
    append P.[i][1] to I
    append P.[i][2] to I
sort I
for i = 1 to 2n
    if I[i] ≠ i
        return FALSE
return TRUE
```

▷ *Solution 283.2*

The problem is in P. We prove that by showing an algorithm that *solves* the problem in polynomial time.

```
Has-Uniform-Pairing(A)
B = sorted copy of A
v = B[1] + B[A.length]
i = 2
j = A.length − 1
while j > i
    if B[i] + B[j] ≠ v
        return FALSE
    i = i + 1
    j = j − 1
return TRUE
```

▷ *Solution 284.1*

```
At-Most-K-Leaves(T, k)
if Count-Leaves(T) ≤ k
    return TRUE
else return FALSE
```

```
Count-Leaves(T)
if T == NIL
    return 0
if T.left == NIL and T.right == NIL
    return 1
return Count-Leaves(T.left) + Count-Leaves(T.right)
```

The complexity is COUNT-LEAVES is $\Theta(n)$, since the algorithm performs a full walk of the tree. AT-MOST-K-LEAVES simply calls COUNT-LEAVES, so its complexity is also Θn.

▷ *Solution 284.2*
We can perform the same walk through the tree without using recursion, by simply using a breadth-first search on the tree.

```
At-Most-K-Leaves-Itr(T, k)
Q = empty queue
ℓ = 0
if T ≠ NIL
    enqueue T into Q
while Q is not empty
    t = dequeue from Q
    if t.left == NIL and t.right == NIL
        ℓ = ℓ + 1
        if ℓ > k
            return FALSE
    else if t.left ≠ NIL
            enqueue t.left into Q
        if t.right ≠ NIL
            enqueue t.right into Q
return TRUE
```

▷ *Solution 285.1*
ALGO-X returns the maximal length of any contiguous sub-sequence of indexes $1 \le i \le n$ such that $A[i] > B[i]$. Interpreting A and B as data at times $1, 2, \ldots, n$, then ALGO-X returns the maximal interval (length) where the A curve is greater than the B curve.
The complexity is $\Theta(n^2)$, since ALGO-X calls ALGO-Y for each value of $1 \le i \le n$, and ALGO-Y always runs for $n - i$ steps.

▷ *Solution 285.2*

Better-Algo-X(A, B)

```
ℓ = 0
j = 1
for i = 1 to A.length
    if A[i] > B[i]
        if i − j + 1 > ℓ
            ℓ = i − j + 1
    else j = i + 1
return ℓ
```

The complexity of Better-Algo-X is $\Theta(n)$, since the algorithm simply scans A and B once.

▷ *Solution 286*

In essence, the resulting order must be such that the value in the middle position $A[\lfloor n/2 \rfloor]$ is maximal, and that the subsequence to the left of $A[\lfloor n/2 \rfloor]$ is increasing while the subsequence on the right is decreasing. Other than that, the two sides don't need to be balanced or otherwise correlated in any way. Also, there are no complexity constraints. So, we can develop a very simple solution based on Insertion-Sort. The idea here is to first sort the whole sequence, with Insertion-Sort, and then to invert the right half of A.

Mountain-Sort(A)

```
n = A.length
for i = 2 to n
    j = i
    while j > 1 and A[j − 1] > A[j]
        swap A[j − 1] ↔ A[j]
        j = j − 1
i = ⌊n/2⌋
j = n
while i < j
    swap A[i] ↔ A[j]
    i = i + 1
    j = j − 1
```

The complexity is $\Theta(n^2)$, which is the complexity of sorting the array.
Another approach could be to first put the maximal value in the middle, and then sort the left half in increasing order and the right half in decreasing order.

▷ *Solution 287.1*

Algo-X returns the number of unique values in A

▷ *Solution 287.2*

The worst-case complexity is $\Theta(n^2)$. This is a case in which the algorithm must check that $A[i]$ is not equal to any other value $A[j]$ (with $i \neq j$). In the best case, the algorithm goes through each value $A[i]$ but then does not run the inner loop more than a constant amount of times. This is the case, for example, in which A contains n copies of the same value. In this case, the inner loop terminates immediately for every $A[i]$, and therefore the complexity is $\Theta(n)$.

▷ *Solution 287.3*

We first sort A, and then go through the sorted data B, counting how many elements $B[i]$ are different from their adjacent elements $B[i-1]$ and $B[i+1]$.

Better-Algo-X(A)

```
B = copy of A sorted in ascending order
x = 0
for i = 1 to B.length
    if (i == 1 or B[i] ≠ B[i − 1]) and (i == n or B[i] ≠ B[i + 1])
        x = x + 1
return x
```

▷ *Solution 288.1*
ALGO-Y returns the highest total sales in a period of ten days.

▷ *Solution 288.2*
The complexity is $\Theta(n^2)$. The nested loops perform complete iterations over T without any short-cut. So, the complexity is the same also in the best case.

▷ *Solution 288.3*
We first sort the set of transactions by date, and then we simply scan the set of transactions maintaining the total (net) gain for a window of transactions that are all within 10 days of each other.

```
BETTER-ALGO-Y(T)
S = copy of T sorted by date
i = 1
j = 1
v = 0
m = 0
while j ≤ S.length
    if S[j].date − S[i].date ≤ 10
        v = v + S[j].amount
        j = j + 1
        if m < v
            m = v
    else v = v − S[i].amount
        i = i + 1
return m
```

After sorting T, at a cost of $\Theta(n \log n)$, BETTER-ALGO-Y performs a linear scan of the sorted array. The overall complexity is therefore $\Theta(n \log n)$.

▷ *Solution 289.1*
H is not a valid min heap because the value $H[3] = 8$ should be less than or equal to both the values $H[6] = 9$ and $H[7] = 5$. So, $H[7] < H[3]$ violates the min-heap property. Similarly, $H[13] = 6 < H[6] = 9$ also violate the same property. A simple fix is to swap those two pairs of values: $H[7] \leftrightarrow H[3]$ and $H[13] \leftrightarrow H[6]$. The resulting content of the array is:

$$H = [3, 5, 5, 6, 10, 6, 8, 6, 7, 20, 11, 17, 9, 9, 10]$$

▷ *Solution 289.2*

```
MIN-HEAP-ADD(H)
append x to H
i = H.length
while i > 1 and H[i] < H[⌊i/2⌋]
    swap H[i] ↔ H[⌊i/2⌋]
    i = ⌊i/2⌋
```

▷ *Solution 289.3*

$$H = [3, 5, 5, 6, 10, 6, 8, 6, 7, 20, 11, 17, 9, 9, 10, 4]$$
$$H = [3, 5, 5, 6, 10, 6, 8, 6, 7, 20, 11, 17, 9, 9, 4, 10]$$
$$H = [3, 5, 5, 6, 10, 6, 4, 6, 7, 20, 11, 17, 9, 9, 8, 10]$$
$$H = [3, 5, 4, 6, 10, 6, 5, 6, 7, 20, 11, 17, 9, 9, 8, 10]$$

▷ *Solution 290*
We can compute the square root using a straightforward binary search.

```
Square-Root(n)
h = n + 1
l = 0
while l + 1 < h
    m = ⌊(l + h)/2⌋
    if m · m > n
        h = m
    elseif m · m < n
        l = m
    else return m
return m
```

▷ *Solution 291*

The input consists of a sorted sub-sequence of negative numbers (possibly empty) followed by a sorted sub-sequence of positive numbers (possibly empty), possibly with zeroes within and between the first and second sequence. So, all we have to do is pack the first subsequence of negative numbers towards the left side of A, then pack the subsequence of positive numbers towards the right side of A, and then set to 0 all the positions that are left in the middle.

```
Re-Sort(A)
n = A.length
i = 1
i_base = i
while i ≤ n and A[i] ≤ 0
    if A[i] < 0
        A[i_base] = A[i]
        i_base = i_base + 1
    i = i + 1
j = n
j_base = j
while j ≥ i
    if A[j] > 0
        A[j_base] = A[j]
        j_base = j_base − 1
    j = j + 1
while i_base ≤ j_base
    A[i_base] = 0
    i_base = i_base + 1
```

▷ *Solution 292.1*

The problem is in NP, since it is easy to play the game following a given set of choices S that serve as a “witness” for a TRUE answer.

```
VERIFY(A, B, c, S)
n = A.length // assume A.length == B.length
i = 1
j = 1
k = 1
t = 0 // total cost of the game
while i ≤ n or j ≤ n
    if i ≤ n and j ≤ n
        if S[k] == DISCARD-BOTH
            if suit(A[i]) ≠ suit(B[j]) and value(A[i]) ≠ value(B[j])
                return FALSE
            i = i + 1
            j = j + 1
        elseif S[k] == DISCARD-A // discard from A
            t = t + value(A[i])
            i = i + 1
        else t = t + value(B[j]) // discard from B
            j = j + 1
        k = k + 1
    elseif i < n
        t = t + value(A[i])
        i = i + 1
    else t = t + value(B[j])
        j = j + 1
if t < c
    return TRUE
else return FALSE
```

▷ *Solution 292.2*

The problem is in P. We can decide by checking that the minimal cost of a game is less than the given cost limit c. We find the minimal cost of a game with a dynamic programming algorithm. The dynamic-programming solution is simply a coding of all the possible choices in the game.

```
SOLVE(A, B, c)
if DP(A, 1, B, 1) < c
    return TRUE
else return FALSE
```

```
DP(A, i, B, j)
if i > A.length and j > B.length
    return 0
if i == A.length
    t = 0
    while j ≤ B.length
        t = t + valueB[j]
        j = j + 1
    return t
if j == B.length
    t = 0
    while i ≤ A.length
        t = t + valueA[i]
        i = i + 1
    return t
t = min{DP(A, i + 1, B, j) + value(A[i]), DP(A, i, B, j + 1) + value(B[j])}
if suit(A[i]) == suit(B[j]) or value(A[i]) == value(B[j])
    t = min{t, DP(A, i + 1, B, j + 1)}
return t
```

Now, this solution is not really polynomial, since the combination of all possible choices given by the multiple recursion of the DP function leads to an exponential complexity. However, the algorithm can be readily turned into a polynomial one by using memoization, which is left as an exercise for the reader...

▷ *Solution 293.1*
ALGO-X returns TRUE if and only if the characters of B are a subset of those of A, considering also their multiplicity. So, for example, B =“aac” is a subset of A =“aabbcc”. The worst-case is when B does not contain any of the characters of A. For example, A =“aaa...” and B =“bbb...”. In fact, the outer loop (over A) is fixed, and the inner loop (over B) can only terminate when $A[i] == B[j]$ for some i and j. So, in the worst case, the complexity is $\Theta(n^2)$.

▷ *Solution 293.2*
We must compare the sequences as multi-sets. We can do that by first sorting the two sequences, so that we can then compare them element-by-element as if we were performing a *merge* of the two (sorted) sequences.

```
BETTER-ALGO-X(A, B)
C = sorted copy of A
D = sorted copy of B
j = 1
for i = 1 to A.length
    if j > B.length
        return TRUE
    elseif C[i] == D[j]
        j = j + 1
    elseif C[i] > D[j]
        return FALSE
if j ≤ B.length
    return FALSE
else return TRUE
```

The main body of this algorithm runs in $O(n)$ time, so the overall complexity is $\Theta(n \log n)$ for sorting A and B.
Since the values of the characters in A and B are numbers from a fixed and small range, we can also develop an $O(n)$ solution:

```
BETTER-ALGO-X-LINEAR(A, B)
n = A.length
C = []
D = []
for i = 1 to m // m is the size of the alphabet
    append 0 to C
    append 0 to D
for i = 1 to n
    C[A[i]] = C[A[i]] + 1
    D[B[i]] = D[B[i]] + 1
for i = 1 to m
    if C[i] < D[j]
        return FALSE
return TRUE
```

▷ *Solution 294*
Notice that G can be seen as the union of c connected components, with $1 \le c \le n$, where each connected component is a maximal set of vertexes that form a connected subgraph of G. We can then connect those components by adding $c - 1$ edges to form a spanning tree of the connected components.

In practice, we can start from any vertex v_0, then visit all the vertexes reachable from v_0, directly or indirectly using BFS, then find the first vertex v_1 that was not already visited, and therefore implicitly count an additional edge (v_0, v_1), and again visit all the vertexes reachable from v_1 (BFS); then again find the next non-visited vertex v_2, implicitly count an additional edge (v_1, v_2), and so on until we visited every vertex in G.

```
MINIMAL-ADDITIONAL-EDGES(G = (V, E))
Visited = ∅ // vertexes that were already visited
c = 0 // number of connected components
while Visited ≠ V
    u = any vertex that is not in Visited // must exist, since Visited ≠ V
    c = c + 1 // we now run a BFS starting from u
    Q = empty queue
    enqueue u in Q
    Visited = Visited ∪ {u}
    while Q is not empty
        v = dequeue vertex from Q
        for w ∈ Adj(v)
            if w ∉ Visited
                enqueue w in Q
                Visited = Visited ∪ {w}
return c − 1
```

▷ *Solution 295*

A simple way of changing a key is to delete the current one and then insert the new one. However, here we need to do that without creating any new node. That can be done simply by recycling the node we delete, so as to then use it for the following insertion.

```
BST-ROOT-CHANGE(t, x)
if t.left == NIL and t.right == NIL
    t.key = x
    return t
elseif t.left == NIL
    r = t.right
    r.parent = NIL
    t.right = NIL
elseif t.right == NIL
    r = t.left
    r.parent = NIL
    t.left = NIL
else r = t
    t = t.right
    while t.left ≠ NIL
        t = t.left
    if t == t.parent.left
        t.parent.left = t.right
    else t.parent.right = t.right
    t.right = NIL
    r.key = t.key
t.key = x
BST-INSERT(r, t)
return r
```

```
BST-INSERT(r, t)
while TRUE
    if t.key ≤ r.key
        if r.left == NIL
            r.left = t
            return
        r = r.left
    else if r.right == NIL
            r.right = t
            return
        r = r.right
```

The complexity is $\Theta(h)$. This is because the while-loop in the third deletion case goes at most through h iterations, and so does the insertion loop.

▷ *Solution 296.1*

For $k = 1$, the minimal cover is the interval that goes from the minimum to the maximum values of A. From this single interval, we can build two intervals of minimal total length by removing the largest interval that does not contain any number, that is, the largest gap between any two numbers in A. And then again, we can obtain three minimal intervals by removing the second-largest gap, and so on. Thus in general we can obtain k minimal intervals by removing the $k-1$ largest gaps. And since all we care about is the length, we don't need to keep track of which intervals (although that wouldn't be difficult either) and instead we can simply compute the total length and then subtract the top $k-1$ gap lengths.

```
MINIMAL-K-INTERVAL-COVER-LENGTH(A, k)
n = A.length
B = sorted copy of A
G = array of n − 1 numbers
for i = 1 to n − 1
    G[i] = B[i + 1] − B[i]
sort G in decreasing order
ℓ = B[n] − B[1]
for i = 1 to min(k, n − 1)
    ℓ = ℓ − G[i]
return ℓ
```

The complexity is $\Theta(n \log n)$, which is the complexity of the sorting of A and G. The rest of the algorithm is $O(n)$.

▷ *Solution 296.2*
See the solution to Question 1.

▷ *Solution 297.1*
ALGO-X returns the sum of the k smallest values in A. If A contains less than k values, then the result is NIL. The complexity is $\Theta(n^2)$, since there are two nested loops that are run without shortcuts in the worst case of $k > n$.

▷ *Solution 297.2*
It is easy enough to sort the array and then add up the first k elements. Or return NIL if the length n of the array is less than k.

```
BETTER-ALGO-X(A, k)
n = A.length
if k > n
    return NIL
B = sorted copy of A
s = 0
for i = 1 to k
    s = s + B[i]
return s
```

The complexity is $\Theta(n \log n)$, which is the complexity of sorting A.

▷ *Solution 298.1*
ALGO-Y returns TRUE if the input array A contains more than 3 distinct values, or FALSE otherwise. The complexity is $\Theta(n \log n)$, due to the sorting algorithm.

▷ *Solution 298.2*
We can simply scan the input array and build an array of at most three elements to store the first three distinct values.

Better-Algo-Y(A)

```
V = array of 3 elements
k = 0
for i = 1 to A.length
    j = 1
    while j ≤ k and V[j] ≠ A[i]
        j = j + 1
    if j > k
        if k > 3
            return TRUE
        else V[j] = A[i]
            k = k + 1
return FALSE
```

The complexity is $\Theta(n)$, since the main loop goes through the entire array, while the inner loop goes through at most k iterations, where $k \le 3$.