Exercises for Algorithms and Data Structures

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

Edition 2.5
February 2018
(with some solutions)

▶**Exercise 1.** 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.** Write an algorithm called FIND-LARGEST that finds the largest number in an array using a divide-and-conquer strategy. Also, write the time complexity of your algorithm in terms of big-oh notation. Briefly justify your complexity analysis. *(20')*

▶**Exercise 3.** Illustrate the execution of the *merge-sort* algorithm on the array

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

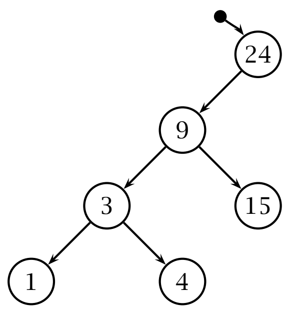
For each fundamental iteration or recursion of the algorithm, write the content of the array. Assume the algorithm performs an in-place sort. *(20')*

▶**Exercise 4.** 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 5.** 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 6.** Implement a function that returns the successor of a node in a binary search tree (the BST stores integer keys). A successor of a node n is defined as the smallest key x in the BST such that x is bigger than the value of n, or *null* if that does not exist. You may assume that the BST does not contain duplicate keys. The signature of the function you have to implement and the interface of the *TreeNode* class, which implements the BST, are given below. Note that *getLeft()*, *getRight()*, and *getParent()* return *null* if the node does not have a left, a right child, or is the *root*, respectively. *(10')*

```
interface TreeNode {
    int getValue();
    TreeNode getLeft();
    TreeNode getRight();
    TreeNode getParent();
}

/* Returns -1 if no successor exists */
int successor(TreeNode x) {
```

►**Exercise 7.** 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 8.** 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 9.** 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 10.** 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 11.** 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 12.** 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 13.** 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 14.** 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 15.** 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 16.** 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, for input $A = \langle 7, 17, 74, 21, 7, 9, 26, 10 \rangle$, 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 17.** 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 18.** Consider the *text* and *pattern* strings:
text: `momify my mom please`
pattern: `mom`
Use the Boyer-Moore string-matching algorithm to search for the pattern in the text. For each character comparison performed by the algorithm, write the current *shift* and highlight the character position considered in the pattern 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*	m	o	m	i	f	y		m	y		m	o	m		p	l	e	a	s	e
1	0	m	o	$\underline{m}$																	
2																					
…	…										…										

►**Exercise 19.** 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 20.** 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 21.** 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 *exact change problem* characterized as follows. *Input:* a multiset of values $V = \{v_1, v_2, \ldots, v_n\}$ representing coins and bills in a cash register; 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 22.** 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 robber can carry, and considering that the robber may take any fraction of each piece, you must find the quantity of each piece the robber must take to maximize the value of the robbery. *(20')*

Question 1: Devise an algorithm that solves the problem using a *greedy* or *dynamic programming* strategy.

Question 2: Prove the problem exhibits an *optimal substructure*. Moreover, if you used a greedy strategy, show that the *greedy choice property* holds for your algorithm. (**Hint**: the *greedy-choice* property holds if and only if every greedy choice is contained in an optimal solution; the optimization problem exhibits an *optimal substructure* if and only if an optimal solution to the problem contains within it optimal solutions to subproblems.)

Question 3: Compute the time complexity of your solution.

```
/* Outputs the quantity of each piece taken */
float[] knapSack(int[] v, int[] w, int W) {
```

►**Exercise 23.** 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 pseudo-code of a function `sortPancakes` that sorts the stack. The i-th element of array `pancakes` contains the diameter of the i-th pancake, counting from the bottom. The `sortPancakes` algorithm can modify the stack only through the `spatulaFlip` function whose interface is specified below.
(**Hint:** Notice that you can move a pancake at position x to position y, without modifying the positions of the order of the other pancakes, using a sequence of spatula flips.) *(20')*

```
/* Flips over the stack of pancakes from position pos and returns the result */
int[] spatulaFlip(int pos, int[] pancakes);

int[] sortPancakes(int[] pancakes) {
```

►**Exercise 24.** 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 25.** Answer the following questions on the complexity classes **P** an **NP**. Justify your answers.

Question 1: $\mathbf{P} \subseteq \mathbf{NP}$? *(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 $Q' \in \mathbf{P}$? Is $Q' \in \mathbf{NP}$? *(5')*

Question 3: Let Q be a problem defined as follows. *Input:* a set of numbers $A = \{a_1, a_2, \dots, a_N\}$ 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 26.** Consider the *subset-sum* problem: given a set of numbers $A = \{a_1, a_2, \dots, a_n\}$ 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 27.** 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 28.** 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 29.** 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 30.** 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 31.** 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 32.** 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 33.** 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 $\{0, 1, \ldots m-1\}$. Find an $m \le 12$ such that h is a perfect hash function on the set of keys $\{0, 6, 9, 12, 22, 31\}$. *(10')*

▶**Exercise 34.** 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 35.** 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 36.** 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 37.** 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 38.** 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 39.** 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 40.** 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: $\sqrt{n} = \Omega(\log(n))$

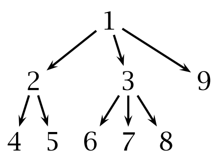
Question 6: *insertion-sort* performs like *quicksort* on an almost sorted sequence

►**Exercise 41.** 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 42.** 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 43.** A *directed tree* is represented as follows: for each vertex v, v.*first-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-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 44.** 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 45.** 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 subsequence of consecutive elements $a_i, a_{i+1}, \ldots, a_k$, with $1 \le i \le k \le n$, 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 46.** 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 47.** 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 48.** 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 49.** A ternary search trie (TST) is used to implement a dictionary of strings. Write the TST corresponding to the following set of strings: "doc" "fun" "algo" "cat" "dog" "data" "car" "led" "function". Assume the strings are inserted in the given order. Use '#' as the terminator character. *(10')*

►**Exercise 50.** The following declarations define a ternary search trie in C and Java, respectively:

```
struct TST {
    char value;
    struct TST * higher;
    struct TST * lower;
    struct TST * equal;
};
void print(const struct TST * t);
```

```
public class TST {
    byte value;
    TST higher;
    TST lower;
    TST equal;
    void print() {/* ...  */}
};
```

The TST represents a dictionary of byte strings. The `print` method must output all the strings stored in the given TST, in alphabetical order. Assume the terminator value is 0. Write an implementation of the `print` method, either in C or in Java. You may assume that the TST contains strings of up to 100 characters. (**Hint**: store the output strings in a static array of characters.) *(20')*

►**Exercise 51.** 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 52.** 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 53.** 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 54.** 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 $H = v_1, v_2, \dots, v_n$ 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 55.** Consider using a b-tree with minimum degree $t = 2$ as an in-memory data structure to implement dynamic sets.

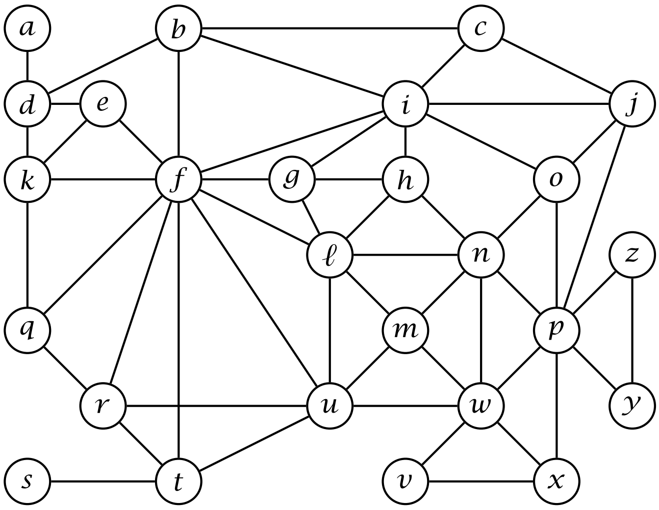
Question 1: Compare this data structure with a red-black tree. Is this data structure better, worse, or the same as a red-black tree in terms of time complexity? Briefly justify your answer. In particular, characterize the complexity of insertion and search. *(10')*

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 56.** 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 57.** 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 58.** 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 59.** 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 60.** 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**?

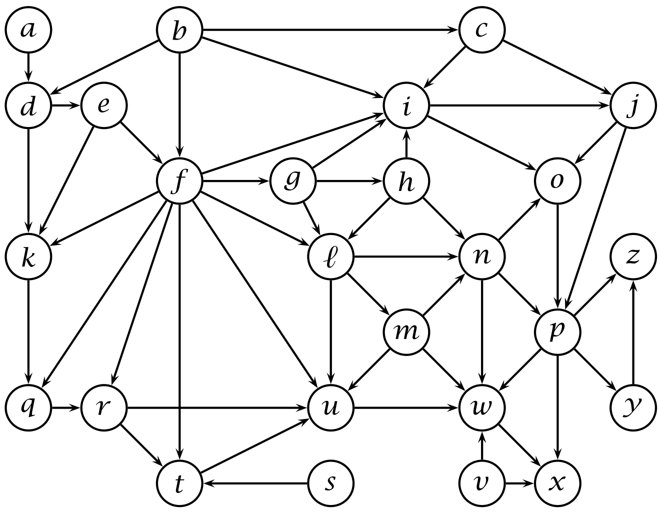
Question 4: A problem P can be polynomially reduced into a problem Q in **NP**. Is P in **P**? Is P **NP-hard**?

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 61.** 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 62.** 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 63.** Write an implementation of a *radix tree* in Java. The tree must store 32-bit integer keys. Each node in the tree must contain at most 256 links or keys, so each node would cover at most 8 bits of the key. You must implement a class called `RadixTree` with two methods, `void insert(int k)` and `boolean find(int k)`. You must also specify every other class you might use. For example, you would probably want to define a class `RadixTreeNode` to represent nodes in the tree. *(20')*

►**Exercise 64.** Answer the following questions about *red-black trees*.

Question 1: Describe the structure and the properties of a red-black tree. *(5')*

Question 2: Write an algorithm RB-TREE-SEARCH(T, k) that, given a red-black tree T and a key k, returns TRUE if T contains key k, or FALSE otherwise. *(5')*

Question 3: Let h be the height of a red-black tree containing n keys, prove that

$$h \leq 2\log(n+1)$$

Hint: first give an outline of the proof. Even if you can not give a complete proof, try to explain informally how the red-black tree property limits the height of a red-black tree. *(10')*

►**Exercise 65.** Sort the following functions in ascending order of asymptotic growth rate:

$f_1(n) = 3^n$	$f_2(n) = n^{1/3}$	$f_3(n) = \log^2 n$
$f_4(n) = n^{\log n}$	$f_5(n) = n^3$	$f_6(n) = 4^{\log n}$
$f_7(n) = n^2\sqrt{n}$	$f_8(n) = 2^{2n}$	$f_9(n) = \sqrt{\log n}$

That is, write the sequence of sorted indexes $a_1, a_2, \ldots, a_9$ such that for all indexes a_i, a_j with $i < j$, $f_{a_i}(n) = O(f_{a_j}(n))$. (Notice that $\log n$ means $\log_2 n$.) *(10')*

►**Exercise 66.** 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 67.** The following defines a *ternary search trie* (TST) for character strings, in Java and in pseudo-code notation:

```
class TSTNode {
    char c;                 x.c           character at node x
    boolean have_key;       x.have-key    TRUE if node x represents a key
    TSTNode left;           x.left        left child of node x
    TSTNode middle;         x.middle      middle child of node x
    TSTNode right;          x.right       right child of node x
}
```

Write an algorithm, `void TSTPrint(TSTNode t)` in Java or TST-PRINT(x) in pseudo-code that, given the root of a TST, prints all its keys in alphabetical order. *(20')*

►**Exercise 68.** 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 69.** 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 $0, \ldots, 8$). 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 70.** 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 71.** 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 72.** 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 73.** 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 74.** 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. (*Reminder:* a node x in a b-tree has the following properties: $x.n$ is the number of keys, $X.key[1] \leq x.key[2] \leq \ldots x.key[x.n]$ 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.) *(30')*

►**Exercise 75.** 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 $A[i \ldots i+5]$ (or $A[i \ldots |A|]$ 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 76.** 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 77.** Characterize the complexity of the following algorithm. Briefly justify your answer. *(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 78.** 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 79.** 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 80.** 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 81.** 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 82.** 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 83.** 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 84.** 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, $\max\{size(t.left), size(t.right)\} \leq \frac{2}{3} size(t)$, 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 85.** 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 86.** 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 87.** 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 88.** 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 89.** 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 90.** 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 91.** 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 92.** 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 93.** 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 94.** 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 95.** 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 96.** 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 97.** 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 98.** 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 99.** 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 100.** 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 101.** 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 102.** 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 $a \leq b$, 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 103.** Let (a, b) represent an interval (or range) of values x such that $a \leq x \leq b$. Consider an array $X = \langle a_1, b_1, a_2, b_2, \dots, a_n, b_n \rangle$ 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 104.** Write an algorithm SIMPLIFY-INTERVALS-FAST(X) that solves exercise 103 with a complexity of $O(n \log n)$. If your solution for exercise 103 already has an $O(n \log n)$ complexity, then simply say so. *(20')*

►**Exercise 105.** 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 106.** 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 107.** Write the *merge sort* algorithm and analyze its complexity. *(10')*

►**Exercise 108.** 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 109.** Answer the following questions on complexity theory. Reminder: 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 110.** 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 111.** 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 112.** 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 113.** 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 114.** 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 115.** 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 116.** 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 117.** 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 118.** 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 119.** 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
```

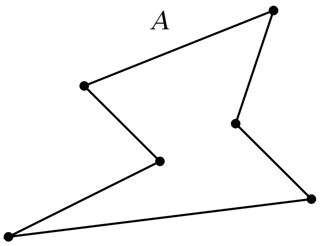
Question 1: Analyze the complexity of ALGO-X. *(5')*

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

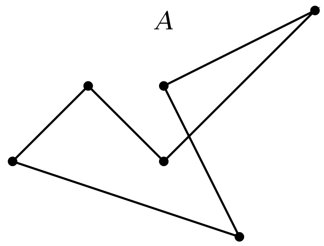
►**Exercise 120.** Write a THREE-WAY-MERGE(A, B, C) algorithm that merges three sorted sequences into a single sorted sequence, and use it to implement a THREE-WAY-MERGE-SORT(L) algorithm. Also, analyze the complexity of THREE-WAY-MERGE-SORT. *(20')*

►**Exercise 121.** 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

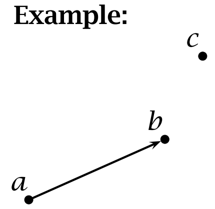


IS-SIMPLE-POLYGON(A) = FALSE

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 122.** 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 123.** 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 124.** 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 125.** 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 126.** 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 127.** 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 128.** 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 129.** 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 130.** 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 131.** Write an algorithm Heap-Delete(H, i) that, given a max-heap H, deletes the element at position i from H. *(10')*

►**Exercise 132.** 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 133.** 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 134.** 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 135.** 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 136.** 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 137.** 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 138.** 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 139.** 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 140.** Write an algorithm MAXIMAL-NON-ADJACENT-SEQUENCE-WEIGHT(A) that, given a sequence of numbers $A = \langle a_1, a_2, \ldots, a_n \rangle$, computes, with worst-case complexity $O(n)$, the maximal weight of any sub-sequence of non-adjacent elements in A. A sub-sequence 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-SEQUENCE-WEIGHT(A) should return 20. (**Hint:** use a dynamic programming algorithm that scans the input sequence once.) *(20')*

►**Exercise 141.** 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 142.** 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 143.** 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 144.** 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 sub-string 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 145.** 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 146.** 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 147.** 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 148.** 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 149.** 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 150.** 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 151.** 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 152.** 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 153.** 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 $A[1 \ldots p-1]$ contains all the elements less than v, the middle part $A[p \ldots q-1]$ contains all the elements equal to v, and the right part $A[q \ldots n]$ 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 154.** 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 155.** 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 156.** 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 157.** 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 158.** 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 159.** 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 160.** 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 161.** Write an algorithm that transforms a min-heap H into a max-heap *in-place*. *(10')*

▶**Exercise 162.** 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 lienar 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 163.** 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 164.** 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 165.** 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. Reminder: 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 166.** 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 167.** 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 168.** 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 169.** 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 170.** 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 171.** 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 172.** 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 173.** 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 174.** 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 175.** 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 176.** 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 177.** 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. (**Reminder:** 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 178.** 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 179.** 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 180.** 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 181.** 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 182.** 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 183.** 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 exactly 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 184.** An algorithm THREE-WAY-PARTITION$(A, \mathit{begin}, \mathit{end})$ chooses a pivot element from the sub-array $A[\mathit{begin} \dots \mathit{end} - 1]$, and partitions that sub-array in-place into three parts (two of which might be empty): $A[\mathit{begin} \dots q_1 - 1]$ containing all the elements less than the pivot, $A[q_1 \dots q_2 - 1]$ containing all the elements equal to the pivot, and $A[q_2 \dots \mathit{end} - 1]$ containing all elements greater than the pivot.

Question 1: Write a THREE-WAY-PARTITION$(A, \mathit{begin}, \mathit{end})$ algorithm that runs in time $O(n)$, where $n = \mathit{end} - \mathit{begin}$, and that returns the partition boundaries q_1, q_2. You may assume that $\mathit{begin} < \mathit{end}$. *(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 185.** 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 186.** 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 $S_i = (t_1, a_1), (t_2, a_2), \dots$ 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$(S_1, S_2, \dots, S_m, k)$ 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 187.** 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 188.** 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, $a < k < b$. (**Reminder:** 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 189.** 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. (**Reminder:** the distance $d(u, v)$ in a graph is the minimal length of any path connecting u and v.) *(20')*

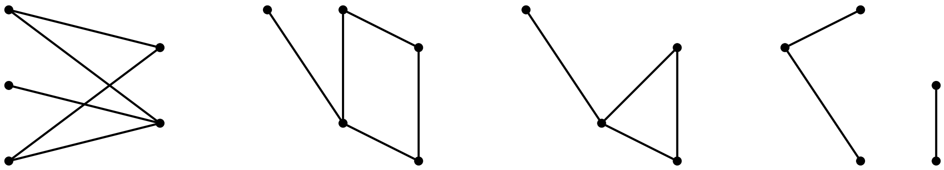
►**Exercise 190.** A highway traffic app sends the coordinates of each vehicle to a central server that reports on congested sections of highway. For simplicity, consider the highway as a straight line in which each position is identified by a single x coordinate. Write an algorithm MOST-CONGESTED-SEGMENT(A, 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 191.** 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 $v_1, v_2, \dots, v_k$ in G that form a fully connected sub-graph of G, meaning that for every pair $i, j \in 1, \dots, k$, 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 192.** 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 193.** Let P be a sequence of points representing an alpine road, where each point $p \in P$ is defined by two coordinates $p.x$ and $p.y$ where $p.x$ is the distance from the beginning of the road and $p.y$ is the elevation (meters above see 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 194.** 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.



G_1 — *bipartite*; G_2 — *bipartite* (same as G_1); G_3 — *not bipartite*; G_4 — *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 195.** 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 196.** 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 197.** Write an algorithm MAXIMAL-COMMON-SUBSTRING(X, Y) that takes two strings X and Y, and returns the maximal length of a common substring of X and Y. For example, if 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 198.** 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 199.** 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 200.** 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 201.** 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 202.** 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 203.** 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 204.** 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 205.** 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 206.** 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 207.** 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 208.** 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 209.** 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 210.** 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 211.** 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 212.** 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 213.** 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')*

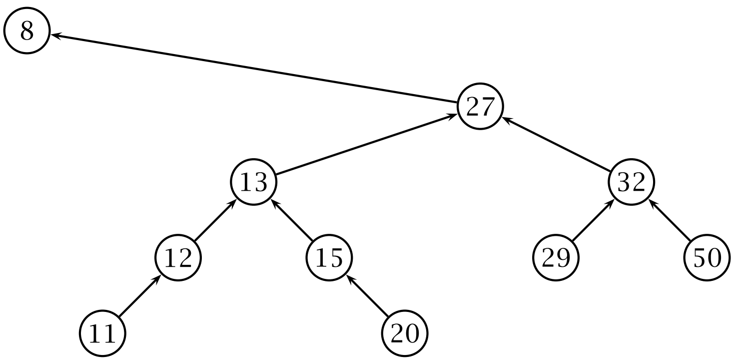
Solutions

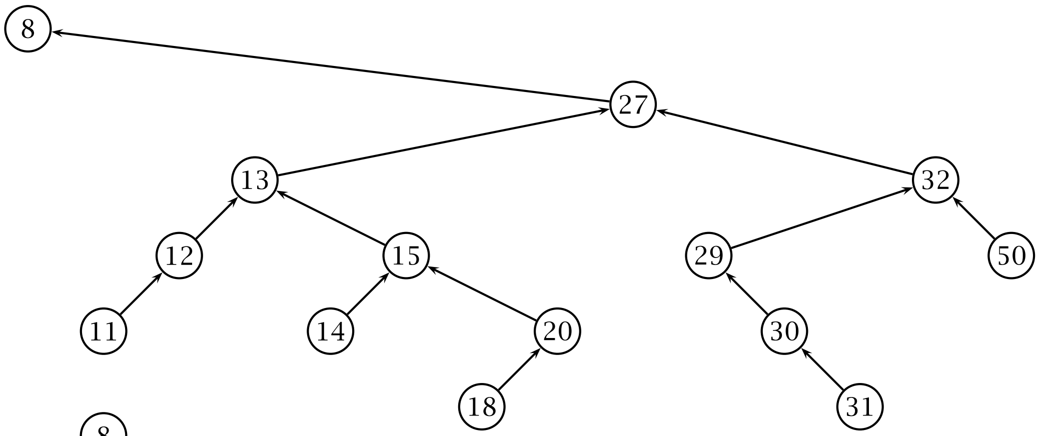
WARNING: solutions are very sparse, meaning that many are missing, most of the solutions are only sketched at a high level, and many may be incorrect! Please, consider contributing your solutions, including alternative solutions, and please report any error you might find to the author (Antonio Carzaniga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

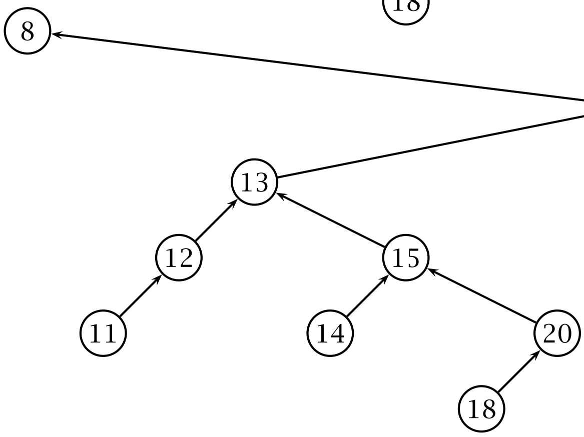
▷ *Solution 57*
Quick-sort. Best-case is $O(n \log n)$, worst-case is $O(n^2)$.

▷ *Solution 58*
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 $s == \text{TRUE}$. 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 $O(n^2)$.
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 $A[1 \dots q-1]$, that is to the left of q, and every other element less than or equal to v ends up somewhere in $A[q+1 \dots |A|]$. 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 70*

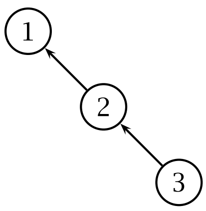






▷ *Solution 71*
a) 50 32 20 29 15 13 12 8 27 11
b) 51 43 50 29 32 20 12 8 27 11 15 13
c) 32 29 20 27 15 13 12 8 11

▷ *Solution 72*
Proof: Let $H = [1, 2, 3]$, then T would look like this:



▷ *Solution 78.1*
yes.

▷ *Solution 78.2*
yes.

▷ *Solution 78.3*
undefined.

▷ *Solution 78.4*
yes.

▷ *Solution 78.5*
undefined.

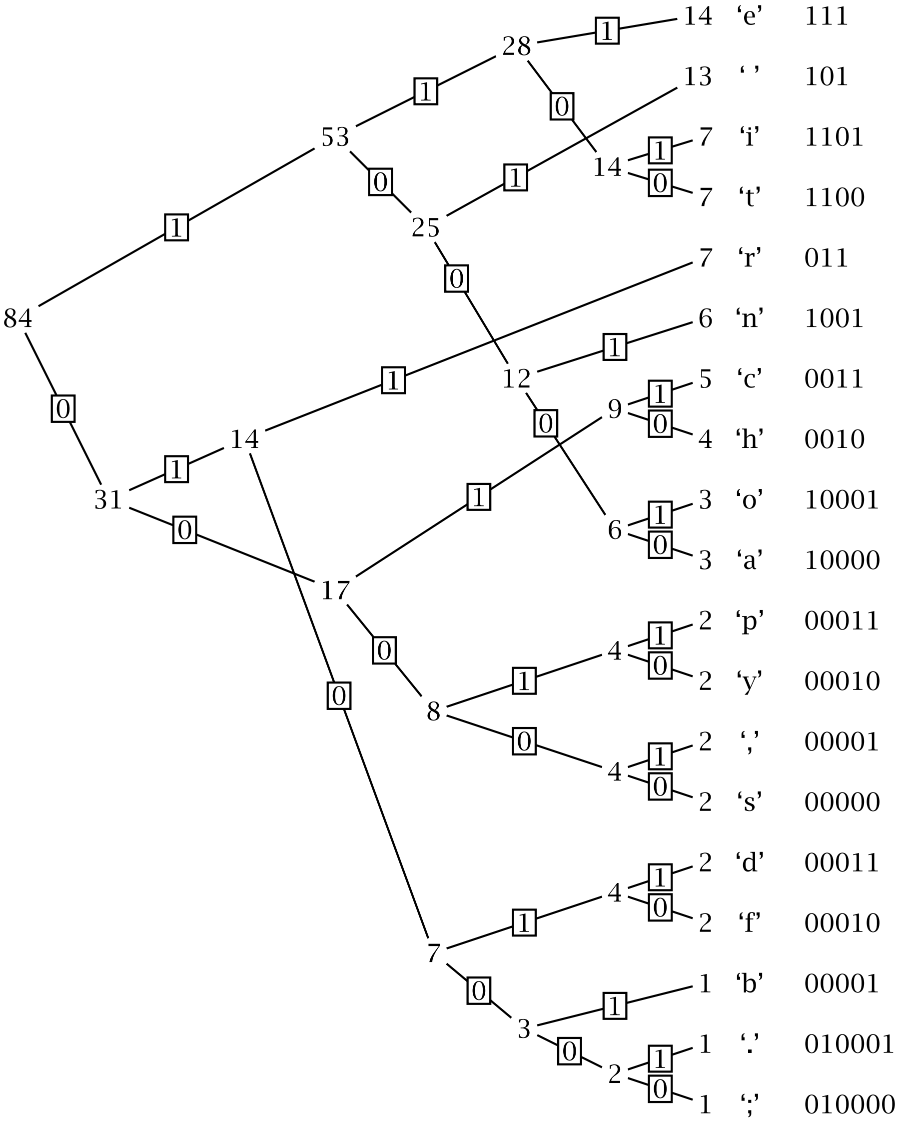
▷ *Solution 78.6*
undefined.

▷ *Solution 78.7*
yes.

▷ *Solution 78.8*
undefined.

▷ *Solution 78.9*
undefined.

▷ *Solution 79*
First figure out the frequencies and sort the characters by frequency. Then we proceed with the derivation:



▷ *Solution 81*
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 87*

Given an array A of number, ALGO-X(A) returns TRUE if and only if there are three numbers $x \leq y \leq z \in A$ 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 99*

```
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 101*

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 $\Theta(n^2)$.

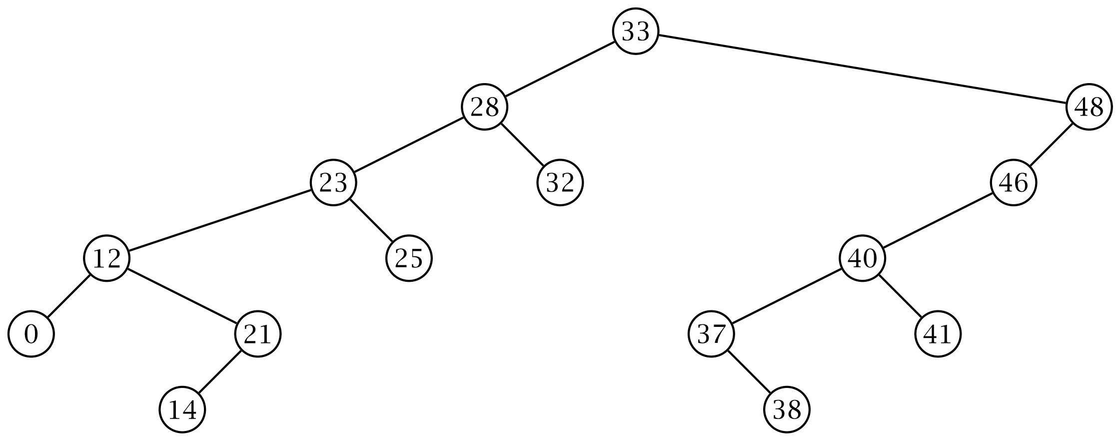
▷ *Solution 105*

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 136.2*



Optimal sequence: 32, 21, 25, 40, 37, 46, 41, 12, 23, 48, 14, 33, 38, 0, 28.

▷ *Solution 140*

We can first start by modeling MAXIMAL-NON-ADJACENT-SEQUENCE-WEIGHT 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 remebering 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 $OPT(a_1, \dots, a_{i-1})$ and $OPT(a_1, \dots, a_{i-2})$. The full algorithm is as follows:

```
MAXIMAL-NON-ADJACENT-SEQUENCE-WEIGHT(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 155.1*

min, max, min−1, max−1, min−2, max−2, ...

▷ *Solution 155.2*

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 163*

```
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 164.1*

```
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 164.2*

```
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 165*

```
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 = 1
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 166*

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 167.1*

```
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 167.2*

```
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 168*

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 169*

```
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 170.1*

```
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 170.2*

```
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 172.1*

```
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 172.2*

```
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 173.1*

```
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 180*

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 181*

```
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 182*

```
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 183.1*

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 183.2*

Notice that any minimal sequence $P[i], P[i+1], \dots, P[j]$ that contains at least k equal elements contains *exactly* k elements equal to the first and last element of the sequence. Otherwise, $P[i], \dots, P[j-1]$ 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 184.1*

```
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 184.2*

```
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 185*

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 186*

```
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 187.1*

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, $A[1 \ldots j-1]$ and $A[j \ldots A.length]$ so that $A[1 \ldots j-1]$ contains only odd numbers and $A[j \ldots A.length]$ contains only even numbers. One of the two parts might be empty. The complexity of ALGO-X is $\Theta(n^2)$.

▷ *Solution 187.2*

```
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 188*

```
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 189*

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 190*

```
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 191*

The problem is in NP because a TRUE answer can be verified in polynomial time with a “certificate” consisting of a set of nodes $C = \{v_1, v_2, \dots, v_\ell\}$

```
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 192*

```
MAX-HEAP-TOP-THREE(H)
if H.length < 4
    for i = 1 to H.length
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 193*

```
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 194*

```
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[v] == RED
                    C[w] = BLUE
                else C[w] = RED
                ENQUEUE(Q, w)
            else if C[v] ≠ C[w]
                    return FALSE
return TRUE
```

▷ *Solution 195.1*

```
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 195.2*

```
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 196*

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 197*

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 198*

```
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 199.1*

ALGO-X returns the maximal difference between two values in an increasing sequence of elements in A. The complexity is $\Theta(n^3)$.

▷ *Solution 199.2*

```
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 200.1*

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 200.2*

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
elseif p_1.y < p_2.y
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 201*

```
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 202*

```
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 203*

```
BST-HEIGHT(t)
if t == NULL
    return 0
return 1 + max(BST-HEIGHT(t.left), BST-HEIGHT(t.right))
```

▷ *Solution 204*

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 205*

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:

$$P_i = V[i] + \max_{j \mid X[j] \le X[i] \wedge Y[j] \le Y[i]} P[j]$$

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 206*

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 207.1*

ALGO-X returns *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 207.2*

```
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 208*

```
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 209*

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 210*

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 211*

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 212*

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 213.1*

```
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 213.2*

```
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 213.3*

```
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.