Divide-and-Conquer Algorithms

Antonio Carzaniga

Faculty of Informatics
Università della Svizzera italiana

March 25, 2021

Outline

- Merging (or set union)
- Searching
- Sorting
- Multiplying
- Computing the *median*

Merging (Set Union)

Merging (Set Union)

- **Input:** sequences $A = \langle a_1, a_2, \ldots, a_n \rangle$ and $B = \langle b_1, b_2, \ldots, b_m \rangle$

  Output: a sequence $X = \langle x_1, x_2, \ldots, x_\ell \rangle$ such that

Merging (Set Union)

- **Input:** sequences $A = \langle a_1, a_2, \ldots, a_n \rangle$ and $B = \langle b_1, b_2, \ldots, b_m \rangle$

  Output: a sequence $X = \langle x_1, x_2, \ldots, x_\ell \rangle$ such that

  - every element of A appears once in X
  - every element of B appears once in X
  - every element of X appears in A or in B or in both

- **Input:** sequences $A = \langle a_1, a_2, \ldots, a_n \rangle$ and $B = \langle b_1, b_2, \ldots, b_m \rangle$

  Output: a sequence $X = \langle x_1, x_2, \ldots, x_\ell \rangle$ such that

  - every element of A appears once in X
  - every element of B appears once in X
  - every element of X appears in A or in B or in both

- **Example:**

  $A = \langle 34, 7, 11, 31, 14, 51, 8, 21, 10 \rangle$

  $B = \langle 51, 21, 14, 15, 27, 31, 2 \rangle$

  $X =$

- **Input:** sequences $A = \langle a_1, a_2, \ldots, a_n \rangle$ and $B = \langle b_1, b_2, \ldots, b_m \rangle$

  Output: a sequence $X = \langle x_1, x_2, \ldots, x_\ell \rangle$ such that

  - every element of A appears once in X
  - every element of B appears once in X
  - every element of X appears in A or in B or in both

- **Example:**

  $A = \langle 34, 7, 11, 31, 14, 51, 8, 21, 10 \rangle$

  $B = \langle 51, 21, 14, 15, 27, 31, 2 \rangle$

  $X = \langle 34, 7, 11, 31, 14, 51, 8, 21, 10, 15, 27, 2 \rangle$

A Simple Merge Algorithm

- Algorithm strategy

- Algorithm strategy
  - iterate through every position i, first through A, and then B
  - output a_i if a_i is not in $\langle a_1, a_2, \ldots, a_{i-1} \rangle$
  - output b_i if b_i is not in $\langle a_1, a_2, \ldots, a_n, b_1, b_2, \ldots b_{i-1} \rangle$

A Simple Merge Algorithm

- Algorithm strategy
  - iterate through every position i, first through A, and then B
  - output a_i if a_i is not in $\langle a_1, a_2, \ldots, a_{i-1} \rangle$
  - output b_i if b_i is not in $\langle a_1, a_2, \ldots, a_n, b_1, b_2, \ldots b_{i-1} \rangle$

```
MergeSimple(A, B)
for i = 1 to length(A)
    if not Find(A[1 . . i − 1], A[i])
        output A[i]
for i = 1 to length(B)
    if not Find(A, B[i]) and not Find(B[1 . . i − 1], B[i])
        output B[i]
```

```
MergeSimple(A, B)
for i = 1 to length(A)
        if not Find(A[1 . . i − 1], A[i])
                output A[i]
for i = 1 to length(B)
        if not Find(A, B[i]) and not Find(B[1 . . i − 1], B[i])
                output B[i]
```

Complexity

MergeSimple(A, B)

```
for i = 1 to length(A)
        if not Find(A[1 .. i − 1], A[i])
                output A[i]
for i = 1 to length(B)
        if not Find(A, B[i]) and not Find(B[1 .. i − 1], B[i])
                output B[i]
```

let $n = length(A) + length(B)$

$$T(n) = \sum_{i=1}^{length(A)} T_{\mathbf{Find}}(i) + \sum_{i=1}^{length(B)} \left(T_{\mathbf{Find}}(i) + T_{\mathbf{Find}}(length(A))\right)$$

Complexity

```
MergeSimple(A, B)
for i = 1 to length(A)
    if not Find(A[1 .. i − 1], A[i])
        output A[i]
for i = 1 to length(B)
    if not Find(A, B[i]) and not Find(B[1 .. i − 1], B[i])
        output B[i]
```

let $n = length(A) + length(B)$

$$T(n) = \sum_{i=1}^{length(A)} T_{\textbf{Find}}(i) + \sum_{i=1}^{length(B)} \left(T_{\textbf{Find}}(i) + T_{\textbf{Find}}(length(A))\right)$$

$$T(n) = \sum_{i=1}^{n} T_{\textbf{Find}}(i)$$

- **Input:** a sequence A and a value *key*

  Output: true if A contains *key*, or false otherwise

- **Input:** a sequence A and a value *key*

  Output: true if A contains *key*, or false otherwise

```
Find(A, key)
for i = 1 to length(A)
    if A[i] == key
        return true
return false
```

```
Find(A, begin, end, key)
for i = begin to end
    if A[i] == key
        return true
return false
```

Searching

- **Input:** a sequence A and a value *key*

  Output: true if A contains *key*, or false otherwise

```
Find(A, key)
    for i = 1 to length(A)
        if A[i] == key
            return true
    return false
```

```
Find(A, begin, end, key)
    for i = begin to end
        if A[i] == key
            return true
    return false
```

- The complexity of **Find** is

- **Input:** a sequence *A* and a value *key*
  Output: true if *A* contains *key*, or false otherwise

```
Find(A, key)
for i = 1 to length(A)
    if A[i] == key
        return true
return false
```

```
Find(A, begin, end, key)
for i = begin to end
    if A[i] == key
        return true
return false
```

- The complexity of **Find** is

$$T(n) = O(n)$$

- **Input:** a sequence A and a value *key*
  Output: true if A contains *key*, or false otherwise

Searching in a List

- **Input:** a sequence *A* and a value *key*

  Output: true if *A* contains *key*, or false otherwise

```
FindInList(A, key)
    item = first(A)
    while item ≠ last(A)
        if value(item) == key
            return true
        item = next(item)
    return false
```

Searching in a List

- **Input:** a sequence A and a value *key*

  Output: true if A contains *key*, or false otherwise

```
FindInList(A, key)
item = first(A)
while item ≠ last(A)
        if value(item) == key
                return true
        item = next(item)
return false
```

- The complexity of **FindInList** is

- **Input:** a sequence A and a value *key*
  Output: true if A contains *key*, or false otherwise

```
FindInList(A, key)
item = first(A)
while item ≠ last(A)
    if value(item) == key
        return true
    item = next(item)
return false
```

- The complexity of **FindInList** is

$$T(n) = O(n)$$

Complexity of MergeSimple

```
MergeSimple(A, B)
for i = 1 to length(A)
        if not Find(A[1 .. i − 1], A[i])
                output A[i]
for i = 1 to length(B)
        if not Find(A, B[i]) and not Find(B[1 .. i − 1], B[i])
                output B[i]
```

Complexity of MergeSimple

```
MergeSimple(A, B)
for i = 1 to length(A)
        if not Find(A[1 .. i − 1], A[i])
                output A[i]
for i = 1 to length(B)
        if not Find(A, B[i]) and not Find(B[1 .. i − 1], B[i])
                output B[i]
```

$$T(n) = \sum_{i=1}^{n} T_{\mathbf{Find}}(i)$$

Complexity of MergeSimple

```
MergeSimple(A, B)
for i = 1 to length(A)
    if not Find(A[1 .. i − 1], A[i])
        output A[i]
for i = 1 to length(B)
    if not Find(A, B[i]) and not Find(B[1 .. i − 1], B[i])
        output B[i]
```

$$T(n) = \sum_{i=1}^{n} T_{\mathbf{Find}}(i)$$

$$T(n) = \sum_{i=1}^{n} O(i) =$$

Complexity of MergeSimple

```
MergeSimple(A, B)
for i = 1 to length(A)
        if not Find(A[1 .. i − 1], A[i])
                output A[i]
for i = 1 to length(B)
        if not Find(A, B[i]) and not Find(B[1 .. i − 1], B[i])
                output B[i]
```

$$T(n) = \sum_{i=1}^{n} T_{\mathbf{Find}}(i)$$

$$T(n) = \sum_{i=1}^{n} O(i) = O\left(\frac{n(n+1)}{2}\right) =$$

Complexity of MergeSimple

```
MergeSimple(A, B)
for i = 1 to length(A)
        if not Find(A[1 .. i − 1], A[i])
                output A[i]
for i = 1 to length(B)
        if not Find(A, B[i]) and not Find(B[1 .. i − 1], B[i])
                output B[i]
```

$$T(n) = \sum_{i=1}^{n} T_{\mathbf{Find}}(i)$$

$$T(n) = \sum_{i=1}^{n} O(i) = O\left(\frac{n(n+1)}{2}\right) = O(n^2)$$

Searching (2)

- **Input:** a *sorted* sequence *A* and a value *key*

  Output: true if *A* contains *key*, or false otherwise

Searching (2)

- **Input:** a *sorted* sequence *A* and a value *key*

  Output: true if *A* contains *key*, or false otherwise

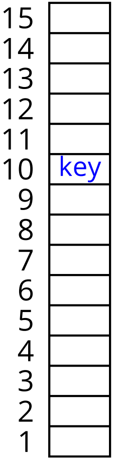
```
BinarySearch(A, key)
first = 1
last = length(A)
while first ≤ last
    middle = ⌈(first + last)/2⌉
    if A[middle] == key
        return true
    elseif first = last
        return false
    elseif A[middle] > key
        last = middle − 1
    else first = middle + 1
return false
```

Binary Search

```
BinarySearch(A, key)
first = 1
last = length(A)
while first ≤ last
        middle = ⌈(first + last)/2⌉
        if A[middle] == key
                return true
        elseif first = last
                return false
        elseif A[middle] > key
                last = middle − 1
        else first = middle + 1
return false
```

Binary Search

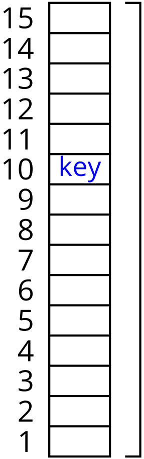
```
BinarySearch(A, key)
first = 1
last = length(A)
while first ≤ last
        middle = ⌈(first + last)/2⌉
        if A[middle] == key
                return true
        elseif first = last
                return false
        elseif A[middle] > key
                last = middle − 1
        else first = middle + 1
return false
```



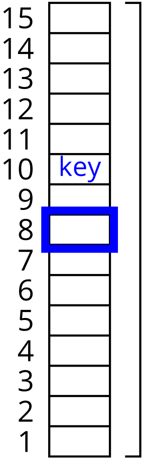
Binary Search

BinarySearch(A, key)

```
first = 1
last = length(A)
while first ≤ last
    middle = ⌈(first + last)/2⌉
    if A[middle] == key
        return true
    elseif first = last
        return false
    elseif A[middle] > key
        last = middle − 1
    else first = middle + 1
return false
```

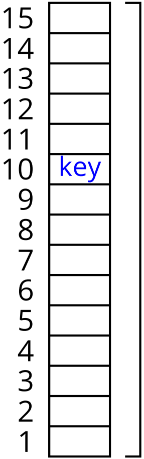


```
BinarySearch(A, key)
first = 1
last = length(A)
while first ≤ last
    middle = ⌈(first + last)/2⌉
    if A[middle] == key
        return true
    elseif first == last
        return false
    elseif A[middle] > key
        last = middle − 1
    else first = middle + 1
return false
```



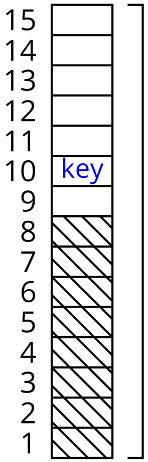
Binary Search

```
BinarySearch(A, key)
    first = 1
    last = length(A)
    while first ≤ last
        middle = ⌈(first + last)/2⌉
        if A[middle] == key
            return true
        elseif first = last
            return false
        elseif A[middle] > key
            last = middle − 1
        else first = middle + 1
    return false
```



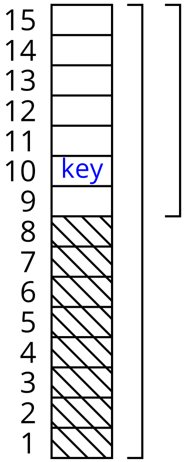
Binary Search

```
BinarySearch(A, key)
first = 1
last = length(A)
while first ≤ last
        middle = ⌈(first + last)/2⌉
        if A[middle] == key
                return true
        elseif first = last
                return false
        elseif A[middle] > key
                last = middle − 1
        else first = middle + 1
return false
```



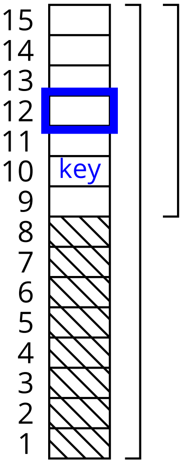
Binary Search

```
BinarySearch(A, key)
first = 1
last = length(A)
while first ≤ last
    middle = ⌈(first + last)/2⌉
    if A[middle] == key
        return true
    elseif first = last
        return false
    elseif A[middle] > key
        last = middle − 1
    else first = middle + 1
return false
```



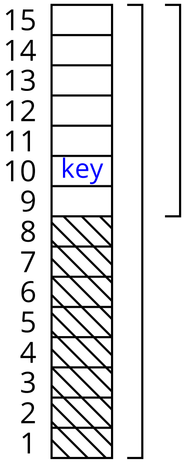
Binary Search

```
BinarySearch(A, key)
first = 1
last = length(A)
while first ≤ last
        middle = ⌈(first + last)/2⌉
        if A[middle] == key
                return true
        elseif first = last
                return false
        elseif A[middle] > key
                last = middle − 1
        else first = middle + 1
return false
```



Binary Search

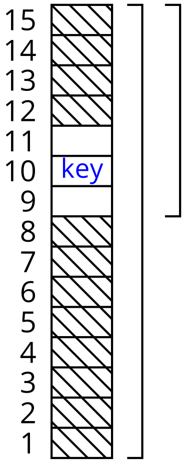
```
BinarySearch(A, key)
first = 1
last = length(A)
while first ≤ last
    middle = ⌈(first + last)/2⌉
    if A[middle] == key
        return true
    elseif first = last
        return false
    elseif A[middle] > key
        last = middle − 1
    else first = middle + 1
return false
```



Binary Search

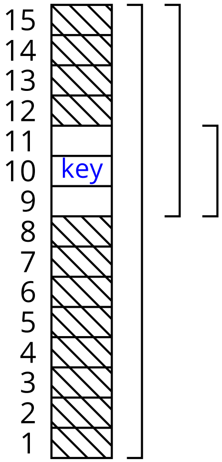
BinarySearch(A, key)

```
first = 1
last = length(A)
while first ≤ last
    middle = ⌈(first + last)/2⌉
    if A[middle] == key
        return true
    elseif first = last
        return false
    elseif A[middle] > key
        last = middle − 1
    else first = middle + 1
return false
```



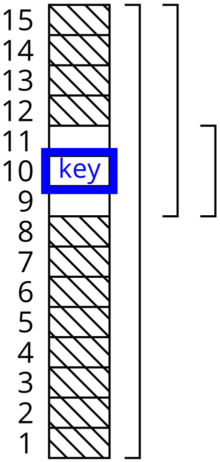
Binary Search

```
BinarySearch(A, key)
first = 1
last = length(A)
while first ≤ last
    middle = ⌈(first + last)/2⌉
    if A[middle] == key
        return true
    elseif first = last
        return false
    elseif A[middle] > key
        last = middle − 1
    else first = middle + 1
return false
```



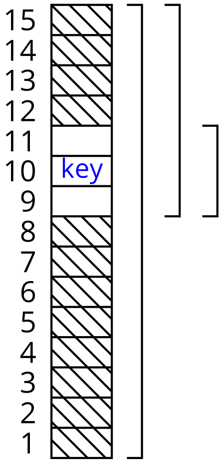
Binary Search

```
BinarySearch(A, key)
    first = 1
    last = length(A)
    while first ≤ last
        middle = ⌈(first + last)/2⌉
        if A[middle] == key
            return true
        elseif first = last
            return false
        elseif A[middle] > key
            last = middle − 1
        else first = middle + 1
    return false
```



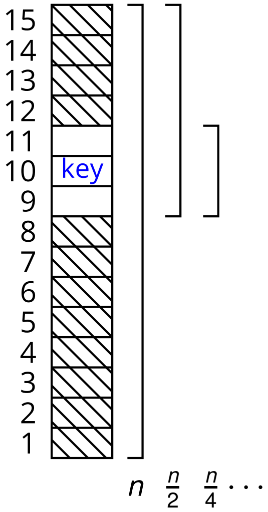
Binary Search

```
BinarySearch(A, key)
first = 1
last = length(A)
while first ≤ last
        middle = ⌈(first + last)/2⌉
        if A[middle] == key
                return true
        elseif first = last
                return false
        elseif A[middle] > key
                last = middle − 1
        else first = middle + 1
return false
```



Binary Search

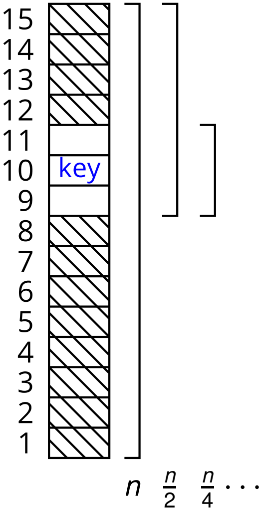
```
BinarySearch(A, key)
first = 1
last = length(A)
while first ≤ last
      middle = ⌈(first + last)/2⌉
      if A[middle] == key
            return true
      elseif first = last
            return false
      elseif A[middle] > key
            last = middle − 1
      else first = middle + 1
return false
```



Binary Search

BinarySearch(A, key)

```
first = 1
last = length(A)
while first ≤ last
        middle = ⌈(first + last)/2⌉
        if A[middle] == key
                return true
        elseif first = last
                return false
        elseif A[middle] > key
                last = middle − 1
        else first = middle + 1
return false
```



$$T(n) = O(\log n)$$

- A slightly different problem:

  Input: two *sorted* sequences $A = \langle a_1, a_2, \ldots, a_n \rangle$ and $B = \langle b_1, b_2, \ldots, b_m \rangle$, where $a_1 \le a_2 \le \ldots \le a_n$ and $b_1 \le b_2 \le \ldots \le b_m$

  Output: a sequence $X = \langle x_1, x_2, \ldots, x_\ell \rangle$ such that

  - every element of A appears once in X
  - every element of B appears once in X
  - every element of X appears in A or in B or in both

A Better Merge Algorithm

```
MergeSimple2(A, B)
for i = 1 to length(A)
        if not BinarySearch(A[1 . . i − 1], A[i])
                output A[i]
for i = 1 to length(B)
        if not BinarySearch(A, B[i])
        and not BinarySearch(B[1 . . i − 1], B[i])
                output B[i]
```

A Better Merge Algorithm

```
MergeSimple2(A, B)
for i = 1 to length(A)
    if not BinarySearch(A[1 .. i − 1], A[i])
        output A[i]
for i = 1 to length(B)
    if not BinarySearch(A, B[i])
    and not BinarySearch(B[1 .. i − 1], B[i])
        output B[i]
```

$$T(n) = \sum_{i=1}^{n} O(\log i) =$$

A Better Merge Algorithm

```
MergeSimple2(A, B)
for i = 1 to length(A)
    if not BinarySearch(A[1 .. i − 1], A[i])
        output A[i]
for i = 1 to length(B)
    if not BinarySearch(A, B[i])
    and not BinarySearch(B[1 .. i − 1], B[i])
        output B[i]
```

$$T(n) = \sum_{i=1}^{n} O(\log i) = O(n \log n)$$

```
MergeSimple2(A, B)
for i = 1 to length(A)
    if not BinarySearch(A[1 .. i − 1], A[i])
        output A[i]
for i = 1 to length(B)
    if not BinarySearch(A, B[i])
    and not BinarySearch(B[1 .. i − 1], B[i])
        output B[i]
```

$$T(n) = \sum_{i=1}^{n} O(\log i) = O(n \log n)$$

Better than $O(n^2)$, but can we do even better than $O(n \log n)$?

An Even Better Merge Algorithm

- *Intuition:* A and B are *sorted*
  e.g.
  $A = \langle 3, 7, 12, 13, 34, 37, 70, 75, 80 \rangle$
  $B = \langle 1, 5, 6, 7, 34, 35, 40, 41, 43 \rangle$

An Even Better Merge Algorithm

- *Intuition:* A and B are *sorted*
  e.g.
  $A = \langle 3, 7, 12, 13, 34, 37, 70, 75, 80 \rangle$
  $B = \langle 1, 5, 6, 7, 34, 35, 40, 41, 43 \rangle$

  so just like in **BinarySearch** I can avoid looking for an element x if the *first* element I see is $y > x$

An Even Better Merge Algorithm

- *Intuition:* A and B are *sorted*
  e.g.
  $A = \langle 3, 7, 12, 13, 34, 37, 70, 75, 80 \rangle$
  $B = \langle 1, 5, 6, 7, 34, 35, 40, 41, 43 \rangle$

  so just like in **BinarySearch** I can avoid looking for an element x if the *first* element I see is $y > x$

- High-level algorithm strategy
  - step through every position i of A and every position j of B
  - output a_i and advance i if $a_i \leq b_j$ or if j is beyond the end of B
  - output b_j and advance j if $a_i \geq b_j$ or if i is beyond the end of A

Merge Algorithm

A	3	7	12	13	34	37	70	75	80

B	1	5	6	7	34	35	40	41	43

Merge Algorithm

$i = 1$

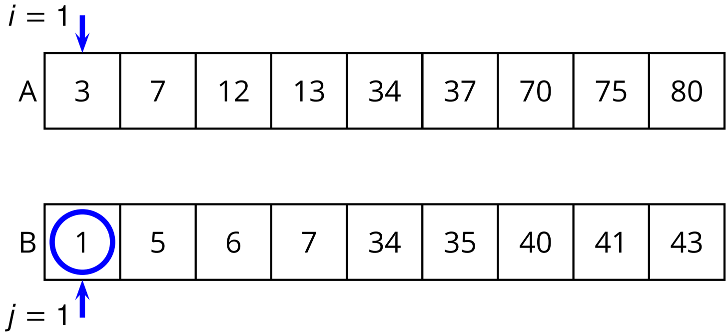
A	3	7	12	13	34	37	70	75	80

B	1	5	6	7	34	35	40	41	43

$j = 1$

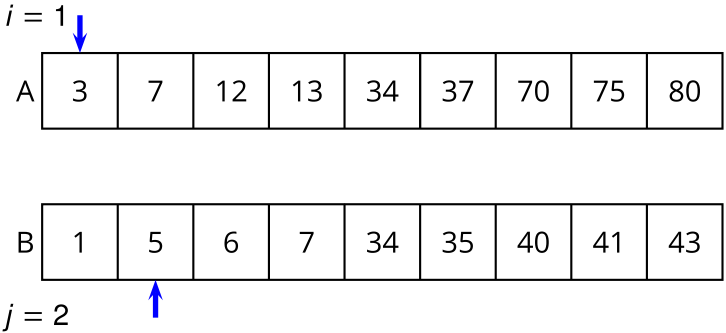
Output:

Merge Algorithm



Output:

Merge Algorithm



Output: 1

Merge Algorithm

$i = 1$

A	3	7	12	13	34	37	70	75	80
B	**1**	**5**	**6**	**7**	**34**	**35**	**40**	**41**	**43**

$j = 2$

Output: 1

$i = 2$

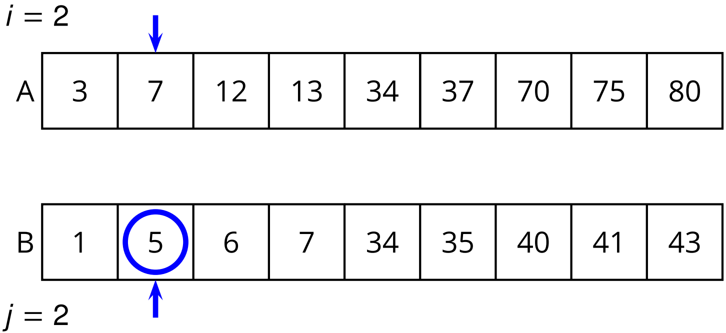
A	3	7	12	13	34	37	70	75	80

B	1	5	6	7	34	35	40	41	43

$j = 2$

Output: 1 3

Merge Algorithm



Output: 1 3

Merge Algorithm

$i = 2$

A	3	7	12	13	34	37	70	75	80

B	1	5	6	7	34	35	40	41	43

$j = 3$

Output: 1 3 5

Merge Algorithm

$i = 2$

A	3	7	12	13	34	37	70	75	80

B	1	5	6	7	34	35	40	41	43

$j = 3$

Output: 1 3 5

Merge Algorithm

$i = 2$

A	3	7	12	13	34	37	70	75	80

B	1	5	6	7	34	35	40	41	43

$j = 4$

Output: 1 3 5 6

Merge Algorithm

$i = 2$

A	3	7	12	13	34	37	70	75	80

B	1	5	6	7	34	35	40	41	43

$j = 4$

Output: 1 3 5 6

$i = 3$

A	3	7	12	13	34	37	70	75	80

B	1	5	6	7	34	35	40	41	43

$j = 5$

Output: 1 3 5 6 7

$i = 3$

A	3	7	12	13	34	37	70	75	80

B	1	5	6	7	34	35	40	41	43

$j = 5$

Output: 1 3 5 6 7

Merge Algorithm

$i = 4$

A	3	7	12	13	34	37	70	75	80

B	1	5	6	7	34	35	40	41	43

$j = 5$

Output: 1 3 5 6 7 12

$i = 4$

A	3	7	12	13	34	37	70	75	80

B	1	5	6	7	34	35	40	41	43

$j = 5$

Output: 1 3 5 6 7 12

i = 5

A	3	7	12	13	34	37	70	75	80

B	1	5	6	7	34	35	40	41	43

j = 5

Output: 1 3 5 6 7 12 13

Merge Algorithm

$i = 5$

A	3	7	12	13	34	37	70	75	80

B	1	5	6	7	34	35	40	41	43

$j = 5$

Output: 1 3 5 6 7 12 13...

Merge Algorithm (2)

```
Merge(A, B)
i, j = 1
X = ∅
while i ≤ length(A) or j ≤ length(B)
        if i > length(A)
                X = X ∘ B[j]          // appends B[j] to X
                j = j + 1
        elseif j > length(B)
                X = X ∘ A[i]
                i = i + 1
        elseif A[i] < B[j]
                X = X ∘ A[i]
                i = i + 1
        else X = X ∘ B[j]
                j = j + 1
return X
```

Merge Algorithm (2)

```
Merge(A, B)
i, j = 1
X = ∅
while i ≤ length(A) or j ≤ length(B)
    if i > length(A)
        X = X ∘ B[j]          // appends B[j] to X
        j = j + 1
    elseif j > length(B)
        X = X ∘ A[i]
        i = i + 1
    elseif A[i] < B[j]
        X = X ∘ A[i]
        i = i + 1
    else X = X ∘ B[j]
        j = j + 1
return X
```

- This algorithm is incorrect! (Exercise: fix it)

Complexity of Merge

Merge(A, B)

$i, j = 1$
$X = \emptyset$
while $i \leq \mathit{length}(A)$ **or** $j \leq \mathit{length}(B)$
    if $i \leq \mathit{length}(A)$ **and** $(j > \mathit{length}(B)$ **or** $A[i] < B[j])$
        $X = X \circ A[i]$
        $i = i + 1$
    else $X = X \circ B[j]$
        $j = j + 1$
return X

Complexity of Merge

Merge(A, B)

$i, j = 1$
$X = \varnothing$
while $i \leq \mathit{length}(A)$ **or** $j \leq \mathit{length}(B)$
    if $i \leq \mathit{length}(A)$ **and** $(j > \mathit{length}(B)$ **or** $A[i] < B[j])$
        $X = X \circ A[i]$
        $i = i + 1$
    else $X = X \circ B[j]$
        $j = j + 1$
return X

$$\boxed{T(n) = \Theta(n)}$$

Complexity of Merge

```
Merge(A, B)
i, j = 1
X = ∅
while i ≤ length(A) or j ≤ length(B)
    if i ≤ length(A) and (j > length(B) or A[i] < B[j])
        X = X ∘ A[i]
        i = i + 1
    else X = X ∘ B[j]
        j = j + 1
return X
```

$$\boxed{T(n) = \Theta(n)}$$

- Can we do better?

Complexity of Merge

```
Merge(A, B)
i, j = 1
X = ∅
while i ≤ length(A) or j ≤ length(B)
        if i ≤ length(A) and (j > length(B) or A[i] < B[j])
                X = X ∘ A[i]
                i = i + 1
        else X = X ∘ B[j]
                j = j + 1
return X
```

$$\boxed{T(n) = \Theta(n)}$$

- Can we do better? No!

Complexity of Merge

```
Merge(A, B)
i, j = 1
X = ∅
while i ≤ length(A) or j ≤ length(B)
        if i ≤ length(A) and (j > length(B) or A[i] < B[j])
                X = X ∘ A[i]
                i = i + 1
        else X = X ∘ B[j]
                j = j + 1
return X
```

$$\boxed{T(n) = \Theta(n)}$$

- Can we do better? No!
  - we have to output $n = \mathit{length}(A) + \mathit{length}(B)$ elements

- So now we have a *linear-complexity* merge procedure
  - merges two *sorted* sequences
  - *produces a sorted sequence*

Using Merge

- So now we have a *linear-complexity* merge procedure
  - merges two *sorted* sequences
  - *produces a sorted sequence*

- Perhaps we could use it to implement a sort algorithm

- So now we have a *linear-complexity* merge procedure
  - merges two *sorted* sequences
  - *produces a sorted sequence*

- Perhaps we could use it to implement a sort algorithm

- Idea
  - use a variant of **Merge** that outputs *all* elements of its input sequences
    - i.e., without removing duplicates
  - assume that two parts, $A_L \circ A_R = A$, and that A_L and A_R are sorted

- So now we have a *linear-complexity* merge procedure
  - merges two *sorted* sequences
  - *produces a sorted sequence*

- Perhaps we could use it to implement a sort algorithm

- Idea
  - use a variant of **Merge** that outputs *all* elements of its input sequences
    - i.e., without removing duplicates
  - assume that two parts, $A_L \circ A_R = A$, and that A_L and A_R are sorted
  - use **Merge** to combine A_L and A_R into a sorted sequence

- So now we have a *linear-complexity* merge procedure
  - merges two *sorted* sequences
  - *produces a sorted sequence*

- Perhaps we could use it to implement a sort algorithm

- Idea
  - use a variant of **Merge** that outputs *all* elements of its input sequences
    - i.e., without removing duplicates
  - assume that two parts, $A_L \circ A_R = A$, and that A_L and A_R are sorted
  - use **Merge** to combine A_L and A_R into a sorted sequence
  - this suggests a recursive algorithm

Merge Sort

MergeSort(A)

```
if length(A) == 1
        return A
m = ⌊length(A)/2⌋
A_L = MergeSort(A[1 .. m])
A_R = MergeSort(A[m + 1 .. length(A)])
return Merge(A_L, A_R)
```

Merge Sort

```
MergeSort(A)
if length(A) == 1
        return A
m = ⌊length(A)/2⌋
A_L = MergeSort(A[1 . . m])
A_R = MergeSort(A[m + 1 . . length(A)])
return Merge(A_L, A_R)
```

- The complexity of **MergeSort** is

Merge Sort

MergeSort(A)

```
if length(A) == 1
        return A
m = ⌊length(A)/2⌋
A_L = MergeSort(A[1 .. m])
A_R = MergeSort(A[m + 1 .. length(A)])
return Merge(A_L, A_R)
```

- The complexity of **MergeSort** is

$$T(n) = 2T(n/2) + O(n)$$

Merge Sort

MergeSort(A)

```
if length(A) == 1
        return A
m = ⌊length(A)/2⌋
A_L = MergeSort(A[1 .. m])
A_R = MergeSort(A[m + 1 .. length(A)])
return Merge(A_L, A_R)
```

- The complexity of **MergeSort** is

$$T(n) = 2T(n/2) + O(n)$$

$$\boxed{T(n) = O(n \log n)}$$

- **MergeSort** exemplifies the ***divide and conquer*** strategy

- **MergeSort** exemplifies the ***divide and conquer*** strategy
- *General strategy:* given a problem P on input data A
  - ***divide*** the input A into parts $A_1, A_2, \ldots, A_k$ with $|A_i| < |A| = n$
  - ***solve*** problem P for the individual k parts
  - ***combine*** the partial solutions to obtain the solution for A

- **MergeSort** exemplifies the ***divide and conquer*** strategy
- *General strategy:* given a problem P on input data A
  - ***divide*** the input A into parts $A_1, A_2, \ldots, A_k$ with $|A_i| < |A| = n$
  - ***solve*** problem P for the individual k parts
  - ***combine*** the partial solutions to obtain the solution for A
- Complexity analysis

$$T(n) = T_{\text{divide}} + \sum_{i=1}^{k} T(|A_i|) + T_{\text{combine}}$$

we will analyze this formula another time…

A Divide-and-Conquer Merge

MergeR(A, B)

```
if length(A) == 0
        return B
if length(B) == 0
        return A
if A[1] < B[1]
        return A[1] ∘ MergeR(A[2 . . length(A)], B)
else return B[1] ∘ MergeR(A, B[2 . . length(B)])
```

A Divide-and-Conquer Merge

```
MergeR(A, B)
if length(A) == 0
        return B
if length(B) == 0
        return A
if A[1] < B[1]
        return A[1] ∘ MergeR(A[2 . . length(A)], B)
else return B[1] ∘ MergeR(A, B[2 . . length(B)])
```

- Again, this algorithm is a bit incorrect (Exercise: Fix it.)

A Divide-and-Conquer Merge

```
MergeR(A, B)
if length(A) == 0
        return B
if length(B) == 0
        return A
if A[1] < B[1]
        return A[1] ∘ MergeR(A[2 . . length(A)], B)
else return B[1] ∘ MergeR(A, B[2 . . length(B)])
```

- Again, this algorithm is a bit incorrect (Exercise: Fix it.)
- The complexity of **MergeR** is

A Divide-and-Conquer Merge

```
MergeR(A, B)
if length(A) == 0
        return B
if length(B) == 0
        return A
if A[1] < B[1]
        return A[1] ∘ MergeR(A[2 . . length(A)], B)
else return B[1] ∘ MergeR(A, B[2 . . length(B)])
```

- Again, this algorithm is a bit incorrect (Exercise: Fix it.)
- The complexity of **MergeR** is

$$T(n) = C_1 + T(n-1)$$

A Divide-and-Conquer Merge

```
MergeR(A, B)
if length(A) == 0
        return B
if length(B) == 0
        return A
if A[1] < B[1]
        return A[1] ∘ MergeR(A[2 . . length(A)], B)
else return B[1] ∘ MergeR(A, B[2 . . length(B)])
```

- Again, this algorithm is a bit incorrect (Exercise: Fix it.)
- The complexity of **MergeR** is

$$T(n) = C_1 + T(n-1) = C_1 n$$

A Divide-and-Conquer Merge

```
MergeR(A, B)
if length(A) == 0
        return B
if length(B) == 0
        return A
if A[1] < B[1]
        return A[1] ∘ MergeR(A[2 . . length(A)], B)
else return B[1] ∘ MergeR(A, B[2 . . length(B)])
```

- Again, this algorithm is a bit incorrect (Exercise: Fix it.)
- The complexity of **MergeR** is

$$T(n) = C_1 + T(n-1) = C_1 n = O(n)$$

- Can we do better?

A Divide-and-Conquer Merge

```
MergeR(A, B)
if length(A) == 0
        return B
if length(B) == 0
        return A
if A[1] < B[1]
        return A[1] ∘ MergeR(A[2 . . length(A)], B)
else return B[1] ∘ MergeR(A, B[2 . . length(B)])
```

- Again, this algorithm is a bit incorrect (Exercise: Fix it.)
- The complexity of **MergeR** is

$$T(n) = C_1 + T(n-1) = C_1 n = O(n)$$

- Can we do better? No! (We knew that already)

Divide-and-Conquer Multiplication

- Going back to multiplication…

- Going back to multiplication...

$x = \boxed{X_L}\,\boxed{X_R}$ and $y = \boxed{Y_L}\,\boxed{Y_R}$

- Going back to multiplication...

  $x = \boxed{\ X_L\ }\boxed{\ X_R\ }$ and $y = \boxed{\ Y_L\ }\boxed{\ Y_R\ }$

  which means $x = 2^{\ell/2}x_L + x_R$ and $y = 2^{\ell/2}y_L + y_R$, so...

$$\begin{aligned} xy &= (2^{\ell/2}x_L + x_R)(2^{\ell/2}y_L + y_R) \\ &= 2^{\ell}x_Ly_L + 2^{\ell/2}(x_Ly_R + x_Ry_L) + x_Ry_R \end{aligned}$$

  we reduced the problem of multiplying two numbers of ℓ bits into the problem of multiplying *four* numbers of $\ell/2$ bits...

- Going back to multiplication...

  $x = \boxed{X_L}\,\boxed{X_R}$ and $y = \boxed{Y_L}\,\boxed{Y_R}$

  which means $x = 2^{\ell/2}x_L + x_R$ and $y = 2^{\ell/2}y_L + y_R$, so...

$$\begin{aligned} xy &= (2^{\ell/2}x_L + x_R)(2^{\ell/2}y_L + y_R) \\ &= 2^{\ell}x_Ly_L + 2^{\ell/2}(x_Ly_R + x_Ry_L) + x_Ry_R \end{aligned}$$

  we reduced the problem of multiplying two numbers of ℓ bits into the problem of multiplying *four* numbers of $\ell/2$ bits...

$$T(\ell) = 4T(\ell/2) + O(\ell)$$

- Going back to multiplication…

  $x = \boxed{X_L}\boxed{X_R}$ and $y = \boxed{Y_L}\boxed{Y_R}$

  which means $x = 2^{\ell/2}x_L + x_R$ and $y = 2^{\ell/2}y_L + y_R$, so…

$$\begin{aligned} xy &= (2^{\ell/2}x_L + x_R)(2^{\ell/2}y_L + y_R) \\ &= 2^{\ell}x_Ly_L + 2^{\ell/2}(x_Ly_R + x_Ry_L) + x_Ry_R \end{aligned}$$

  we reduced the problem of multiplying two numbers of ℓ bits into the problem of multiplying *four* numbers of $\ell/2$ bits…

$$T(\ell) = 4T(\ell/2) + O(\ell)$$

$$T(\ell) = \Theta(\ell^2)$$

Divide-and-Conquer Multiplication (2)

Divide-and-Conquer Multiplication (2)

- Again, we have

$$\begin{aligned} xy &= (2^{\ell/2}x_L + x_R)(2^{\ell/2}y_L + y_R) \\ &= 2^{\ell}x_L y_L + 2^{\ell/2}(x_L y_R + x_R y_L) + x_R y_R \end{aligned}$$

- Again, we have

$$\begin{aligned} xy &= (2^{\ell/2}x_L + x_R)(2^{\ell/2}y_L + y_R) \\ &= 2^{\ell}x_Ly_L + 2^{\ell/2}(x_Ly_R + x_Ry_L) + x_Ry_R \end{aligned}$$

  but notice that $x_Ly_R + x_Ry_L = (x_L + x_R)(y_R + y_L) - x_Ly_L - x_Ry_R$, so

- Again, we have

$$\begin{aligned} xy &= (2^{\ell/2}x_L + x_R)(2^{\ell/2}y_L + y_R) \\ &= 2^{\ell}x_Ly_L + 2^{\ell/2}(x_Ly_R + x_Ry_L) + x_Ry_R \end{aligned}$$

but notice that $x_Ly_R + x_Ry_L = (x_L + x_R)(y_R + y_L) - x_Ly_L - x_Ry_R$, so

$$xy = 2^{\ell}x_Ly_L + 2^{\ell/2}((x_L + x_R)(y_R + y_L) - x_Ly_L - x_Ry_R) + x_Ry_R$$

Divide-and-Conquer Multiplication (2)

- Again, we have

$$\begin{aligned} xy &= (2^{\ell/2}x_L + x_R)(2^{\ell/2}y_L + y_R) \\ &= 2^{\ell}x_Ly_L + 2^{\ell/2}(x_Ly_R + x_Ry_L) + x_Ry_R \end{aligned}$$

  but notice that $x_Ly_R + x_Ry_L = (x_L + x_R)(y_R + y_L) - x_Ly_L - x_Ry_R$, so

$$xy = 2^{\ell}x_Ly_L + 2^{\ell/2}((x_L + x_R)(y_R + y_L) - x_Ly_L - x_Ry_R) + x_Ry_R$$

  Only 3 multiplications: x_Ly_L, $(x_L + x_R)(y_R + y_L)$, and x_Ry_R

- Again, we have

$$\begin{aligned} xy &= (2^{\ell/2}x_L + x_R)(2^{\ell/2}y_L + y_R) \\ &= 2^{\ell}x_Ly_L + 2^{\ell/2}(x_Ly_R + x_Ry_L) + x_Ry_R \end{aligned}$$

  but notice that $x_Ly_R + x_Ry_L = (x_L + x_R)(y_R + y_L) - x_Ly_L - x_Ry_R$, so

$$xy = 2^{\ell}x_Ly_L + 2^{\ell/2}((x_L + x_R)(y_R + y_L) - x_Ly_L - x_Ry_R) + x_Ry_R$$

  Only 3 multiplications: x_Ly_L, $(x_L + x_R)(y_R + y_L)$, and x_Ry_R

$$T(\ell) = 3T(\ell/2) + O(\ell)$$

- Again, we have

$$\begin{aligned} xy &= (2^{\ell/2}x_L + x_R)(2^{\ell/2}y_L + y_R) \\ &= 2^{\ell}x_Ly_L + 2^{\ell/2}(x_Ly_R + x_Ry_L) + x_Ry_R \end{aligned}$$

but notice that $x_Ly_R + x_Ry_L = (x_L + x_R)(y_R + y_L) - x_Ly_L - x_Ry_R$, so

$$xy = 2^{\ell}x_Ly_L + 2^{\ell/2}((x_L + x_R)(y_R + y_L) - x_Ly_L - x_Ry_R) + x_Ry_R$$

Only 3 multiplications: x_Ly_L, $(x_L + x_R)(y_R + y_L)$, and x_Ry_R

$$T(\ell) = 3T(\ell/2) + O(\ell)$$

which, as we will see, leads to a much better complexity

$$T(\ell) = O(\ell^{\log_2 3}) = O(\ell^{1.59})$$

- The *median* of a sequence A is a value $m \in A$ such that half the values in A are smaller than m and half are bigger than m

- The *median* of a sequence A is a value $m \in A$ such that half the values in A are smaller than m and half are bigger than m
  - e.g., what is the median of $A = \langle 2, 36, 5, 21, 8, 13, 11, 20, 5, 4, 1 \rangle$?

Computing the Median

- The *median* of a sequence A is a value $m \in A$ such that half the values in A are smaller than m and half are bigger than m
  - e.g., what is the median of $A = \langle 2, 36, 5, 21, 8, 13, 11, 20, 5, 4, 1 \rangle$?

- Idea: first sort, then pick the element in the middle

SimpleMedian(A)
```
1  X = MergeSort(A)
2  return X[⌊length(A)/2⌋]
```

Computing the Median

- The *median* of a sequence A is a value $m \in A$ such that half the values in A are smaller than m and half are bigger than m
  - e.g., what is the median of $A = \langle 2, 36, 5, 21, 8, 13, 11, 20, 5, 4, 1 \rangle$?

- Idea: first sort, then pick the element in the middle

SimpleMedian(A)

```
1  X = MergeSort(A)
2  return X[⌊length(A)/2⌋]
```

- Is it correct?

Computing the Median

- The *median* of a sequence A is a value $m \in A$ such that half the values in A are smaller than m and half are bigger than m
  - e.g., what is the median of $A = \langle 2, 36, 5, 21, 8, 13, 11, 20, 5, 4, 1 \rangle$?

- Idea: first sort, then pick the element in the middle

SimpleMedian(A)

$X = $ **MergeSort**(A)
return $X[\lfloor \mathit{length}(A)/2 \rfloor]$

- Is it correct? Yes

- The *median* of a sequence A is a value $m \in A$ such that half the values in A are smaller than m and half are bigger than m
  - e.g., what is the median of $A = \langle 2, 36, 5, 21, 8, 13, 11, 20, 5, 4, 1 \rangle$?

- Idea: first sort, then pick the element in the middle

SimpleMedian(A)
1 $X = $ **MergeSort**(A)
2 **return** $X[\lfloor \mathit{length}(A)/2 \rfloor]$

- Is it correct? Yes

- How long does it take?

- The *median* of a sequence A is a value $m \in A$ such that half the values in A are smaller than m and half are bigger than m
  - e.g., what is the median of $A = \langle 2, 36, 5, 21, 8, 13, 11, 20, 5, 4, 1 \rangle$?

- Idea: first sort, then pick the element in the middle

```
SimpleMedian(A)
1  X = MergeSort(A)
2  return X[⌊length(A)/2⌋]
```

- Is it correct? Yes

- How long does it take? $T(n) = T_{\textbf{MergeSort}}(n) = O(n \log n)$

- The *median* of a sequence A is a value $m \in A$ such that half the values in A are smaller than m and half are bigger than m
  - e.g., what is the median of $A = \langle 2, 36, 5, 21, 8, 13, 11, 20, 5, 4, 1 \rangle$?

- Idea: first sort, then pick the element in the middle

```
SimpleMedian(A)
1  X = MergeSort(A)
2  return X[⌊length(A)/2⌋]
```

- Is it correct? Yes

- How long does it take? $T(n) = T_{\mathbf{MergeSort}}(n) = O(n \log n)$

- Can we do better?

- The *median* of a sequence A is a value $m \in A$ such that half the values in A are smaller than m and half are bigger than m
  - e.g., what is the median of $A = \langle 2, 36, 5, 21, 8, 13, 11, 20, 5, 4, 1 \rangle$?
- Idea: first sort, then pick the element in the middle

```
SimpleMedian(A)
1   X = MergeSort(A)
2   return X[⌊length(A)/2⌋]
```

- Is it correct? Yes
- How long does it take? $T(n) = T_{\textbf{MergeSort}}(n) = O(n \log n)$
- Can we do better? Let's try *divide-and-conquer*...

Computing the Median (2)

- The *median* of a sequence A is a value $m \in A$ such that half the values in A are less than or equal to m

- The *median* of a sequence A is a value $m \in A$ such that half the values in A are less than or equal to m

- Generalizating, the ***k-smallest*** element of a sequence A is a value $v \in A$ such that exactly k elements of A are less than or equal to v

Computing the Median (2)

- The *median* of a sequence A is a value $m \in A$ such that half the values in A are less than or equal to m

- Generalizating, the ***k-smallest*** element of a sequence A is a value $v \in A$ such that exactly k elements of A are less than or equal to v

  E.g.,

  - for $k = 1$, the minimum of A

Computing the Median (2)

- The *median* of a sequence A is a value $m \in A$ such that half the values in A are less than or equal to m

- Generalizating, the ***k-smallest*** element of a sequence A is a value $v \in A$ such that exactly k elements of A are less than or equal to v

  E.g.,

  - for $k = 1$, the minimum of A
  - for $k = \lfloor |A|/2 \rfloor$, the median of A

- The *median* of a sequence A is a value $m \in A$ such that half the values in A are less than or equal to m

- Generalizating, the ***k-smallest*** element of a sequence A is a value $v \in A$ such that exactly k elements of A are less than or equal to v

  E.g.,

  - for $k = 1$, the minimum of A
  - for $k = \lfloor |A|/2 \rfloor$, the median of A
  - what is the *6th smallest* element of $A = \langle 2, 36, 5, 21, 8, 13, 11, 20, 5, 4, 1 \rangle$?

- The *median* of a sequence A is a value $m \in A$ such that half the values in A are less than or equal to m

- Generalizating, the ***k-smallest*** element of a sequence A is a value $v \in A$ such that exactly k elements of A are less than or equal to v

  E.g.,

  - for $k = 1$, the minimum of A
  - for $k = \lfloor |A|/2 \rfloor$, the median of A
  - what is the *6th smallest* element of $A = \langle 2, 36, 5, 21, 8, 13, 11, 20, 5, 4, 1 \rangle$?

    the 6th smallest element of A—a.k.a. $select(A, 6)$—is 8

k-Smallest Element

- Idea: we split the sequence A in three parts based on a *chosen value* $v \in A$
  - A_L contains the set of elements that are *less than* v
  - A_v contains the set of elements that are *equal to* v
  - A_R contains the set of elements that are *greater then* v

- Idea: we split the sequence A in three parts based on a *chosen value* $v \in A$
  - A_L contains the set of elements that are *less than* v
  - A_v contains the set of elements that are *equal to* v
  - A_R contains the set of elements that are *greater then* v

E.g., $A = \langle 2, 36, 5, 21, 8, 13, 11, 20, 5, 4, 1 \rangle$
and we must compute the 7th smallest value in A

- Idea: we split the sequence A in three parts based on a *chosen value* $v \in A$
  - A_L contains the set of elements that are *less than* v
  - A_v contains the set of elements that are *equal to* v
  - A_R contains the set of elements that are *greater then* v

E.g., $A = \langle 2, 36, 5, 21, 8, 13, 11, 20, 5, 4, 1 \rangle$
and we must compute the 7th smallest value in A

we pick a splitting value, say $v = 5$

- Idea: we split the sequence A in three parts based on a *chosen value* $v \in A$
  - A_L contains the set of elements that are *less than* v
  - A_v contains the set of elements that are *equal to* v
  - A_R contains the set of elements that are *greater then* v

E.g., $A = \langle 2, 36, 5, 21, 8, 13, 11, 20, 5, 4, 1 \rangle$
and we must compute the 7th smallest value in A

we pick a splitting value, say $v = 5$

$$A_L = \langle 2, 4, 1 \rangle$$

- Idea: we split the sequence A in three parts based on a *chosen value* $v \in A$
  - A_L contains the set of elements that are *less than* v
  - A_v contains the set of elements that are *equal to* v
  - A_R contains the set of elements that are *greater then* v

E.g., $A = \langle 2, 36, 5, 21, 8, 13, 11, 20, 5, 4, 1 \rangle$
and we must compute the 7th smallest value in A

we pick a splitting value, say $v = 5$

$$A_L = \langle 2, 4, 1 \rangle \quad A_v = \langle 5, 5 \rangle$$

- Idea: we split the sequence A in three parts based on a *chosen value* $v \in A$
  - A_L contains the set of elements that are *less than* v
  - A_v contains the set of elements that are *equal to* v
  - A_R contains the set of elements that are *greater then* v

E.g., $A = \langle 2, 36, 5, 21, 8, 13, 11, 20, 5, 4, 1 \rangle$
and we must compute the 7th smallest value in A

we pick a splitting value, say $v = 5$

$$A_L = \langle 2, 4, 1 \rangle \quad A_v = \langle 5, 5 \rangle \quad A_R = \langle 36, 21, 8, 13, 11, 20 \rangle$$

- Idea: we split the sequence A in three parts based on a *chosen value* $v \in A$
  - A_L contains the set of elements that are *less than v*
  - A_v contains the set of elements that are *equal to v*
  - A_R contains the set of elements that are *greater then v*

E.g., $A = \langle 2, 36, 5, 21, 8, 13, 11, 20, 5, 4, 1 \rangle$
and we must compute the 7th smallest value in A

we pick a splitting value, say $v = 5$

$$A_L = \langle 2, 4, 1 \rangle \quad A_v = \langle 5, 5 \rangle \quad A_R = \langle 36, 21, 8, 13, 11, 20 \rangle$$

Now, where is the 7th smallest value of A?

- Idea: we split the sequence A in three parts based on a *chosen value* $v \in A$
  - A_L contains the set of elements that are *less than* v
  - A_v contains the set of elements that are *equal to* v
  - A_R contains the set of elements that are *greater then* v

E.g., $A = \langle 2, 36, 5, 21, 8, 13, 11, 20, 5, 4, 1 \rangle$
and we must compute the 7th smallest value in A

we pick a splitting value, say $v = 5$

$$A_L = \langle 2, 4, 1 \rangle \quad A_v = \langle 5, 5 \rangle \quad A_R = \langle 36, 21, 8, 13, 11, 20 \rangle$$

Now, where is the 7th smallest value of A?

It is the 2nd smallest value of A_R

k-Smallest Element (2)

We use $select(A, k)$ to denote the k-smallest element of A

$$select(A,k) = \begin{cases} select(A_L, k) & \text{if } k \leq |A_L| \\ v & \text{if } |A_L| < k \leq |A_L| + |A_v| \\ select(A_R, k - |A_L| - |A_v|) & \text{if } k > |A_L| + |A_v| \end{cases}$$

We use $select(A, k)$ to denote the k-smallest element of A

$$select(A, k) = \begin{cases} select(A_L, k) & \text{if } k \leq |A_L| \\ v & \text{if } |A_L| < k \leq |A_L| + |A_v| \\ select(A_R, k - |A_L| - |A_v|) & \text{if } k > |A_L| + |A_v| \end{cases}$$

- Computing A_L, A_v, and A_R takes $O(n)$ steps

k-Smallest Element (2)

We use $select(A, k)$ to denote the k-smallest element of A

$$select(A,k) = \begin{cases} select(A_L, k) & \text{if } k \leq |A_L| \\ v & \text{if } |A_L| < k \leq |A_L| + |A_v| \\ select(A_R, k - |A_L| - |A_v|) & \text{if } k > |A_L| + |A_v| \end{cases}$$

- Computing A_L, A_v, and A_R takes $O(n)$ steps
- How do we pick v?

We use $select(A, k)$ to denote the k-smallest element of A

$$select(A, k) = \begin{cases} select(A_L, k) & \text{if } k \leq |A_L| \\ v & \text{if } |A_L| < k \leq |A_L| + |A_v| \\ select(A_R, k - |A_L| - |A_v|) & \text{if } k > |A_L| + |A_v| \end{cases}$$

- Computing A_L, A_v, and A_R takes $O(n)$ steps
- How do we pick v?
- Ideally, we should pick v so as to obtain $|A_L| \approx |A_R| \approx |A|/2$
  - so, ideally we should pick $v = median(A)$, but…

We use $select(A, k)$ to denote the k-smallest element of A

$$select(A, k) = \begin{cases} select(A_L, k) & \text{if } k \leq |A_L| \\ v & \text{if } |A_L| < k \leq |A_L| + |A_v| \\ select(A_R, k - |A_L| - |A_v|) & \text{if } k > |A_L| + |A_v| \end{cases}$$

- Computing A_L, A_v, and A_R takes $O(n)$ steps
- How do we pick v?
- Ideally, we should pick v so as to obtain $|A_L| \approx |A_R| \approx |A|/2$
  - so, ideally we should pick $v = median(A)$, but...
- We pick *a random element of A*

Selection Algorithm

Selection(A, k)

$v = A[\text{random}(1 \ldots |A|)]$
$A_L, A_v, A_R = \emptyset$
for $i = 1$ **to** $|A|$
    if $A[i] < v$
        $A_L = A_L \cup A[i]$
    elseif $A[i] == v$
        $A_v = A_v \cup A[i]$
    else $A_R = A_R \cup A[i]$
if $k \leq |A_L|$
    return Selection(A_L, k)
elseif $k > |A_L| + |A_v|$
    return Selection$(A_R, k - |A_L| - |A_v|)$
else return v