Divide-and-Conquer Algorithms

Antonio Carzaniga

Faculty of Informatics
Università della Svizzera italiana

March 6, 2018

Outline

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

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 (a set) $X = \langle x_1, x_2, \ldots, x_\ell \rangle$ 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 (a set) $X = \langle x_1, x_2, \ldots, x_\ell \rangle$ 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 (a set) $X = \langle x_1, x_2, \ldots, x_\ell \rangle$ 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 (a set) $X = \langle x_1, x_2, \ldots, x_\ell \rangle$ 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

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 $\langle a_1, a_2, \ldots, a_n, b_1, b_2, \ldots b_{i-1} \rangle$

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 $n = \mathit{length}(A) + \mathit{length}(B)$

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

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 $n = \mathit{length}(A) + \mathit{length}(B)$

$$T(n) = \sum_{i=1}^{\mathit{length}(A)} T_{\textsc{Find}}(i) + \sum_{i=1}^{\mathit{length}(B)} \big(T_{\textsc{Find}}(i) + T_{\textsc{Find}}(\mathit{length}(A))\big)$$

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

- *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
```

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
```

- 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
```

- The complexity of **FIND** is

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

Searching on a List

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

  Output: TRUE if *A* contains *key*, or FALSE otherwise

Searching on 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 on 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]
```

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

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]
```

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

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

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]
```

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

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

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]
```

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

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

- *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

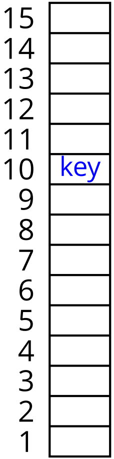
```
BinarySearch(A, key)
first = 1
last = length(A)
while first ≤ last
        middle = ⌈(first + last)/2⌉
        if A[middle] == key
                return TRUE
        elseif first = last
                return FALSE
        elseif A[middle] > key
                last = middle − 1
        else first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

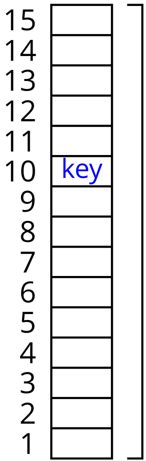
Binary Search

```
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
```



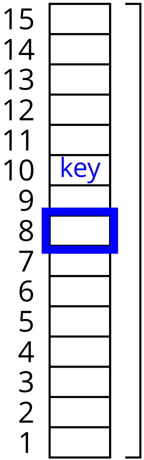
Binary Search

```
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
```



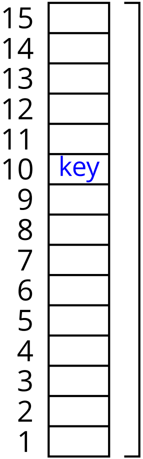
Binary Search

```
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
```



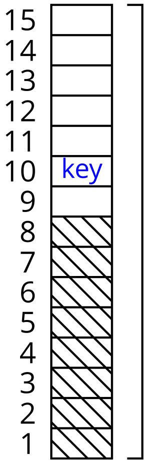
Binary Search

```
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
```



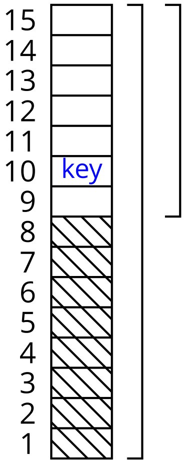
Binary Search

```
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
```

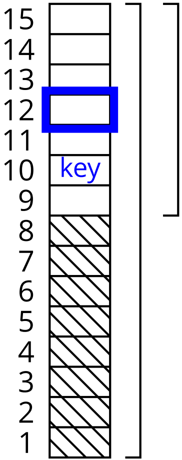


Binary Search

```
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
```

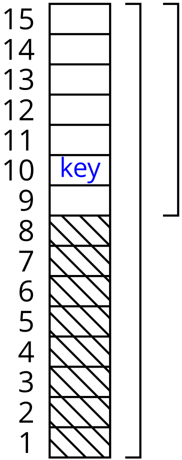


```
BINARYSEARCH(A, key)
first = 1
last = length(A)
while first ≤ last
        middle = ⌈(first + last)/2⌉
        if A[middle] == key
                return TRUE
        elseif first = last
                return FALSE
        elseif A[middle] > key
                last = middle − 1
        else first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



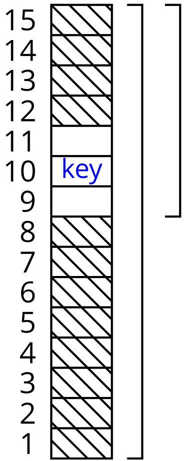
Binary Search

```
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
```



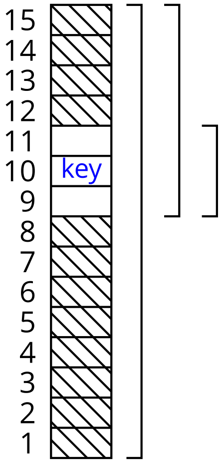
Binary Search

```
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
```



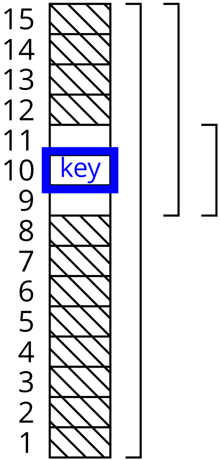
Binary Search

```
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

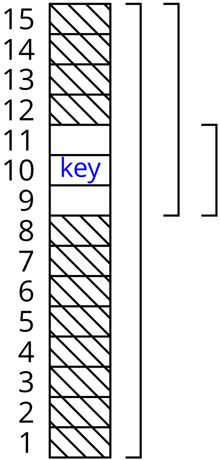
```
BINARYSEARCH(A, key)
first = 1
last = length(A)
while first ≤ last
        middle = ⌈(first + last)/2⌉
        if A[middle] == key
                return TRUE
        elseif first = last
                return FALSE
        elseif A[middle] > key
                last = middle − 1
        else first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



Binary Search

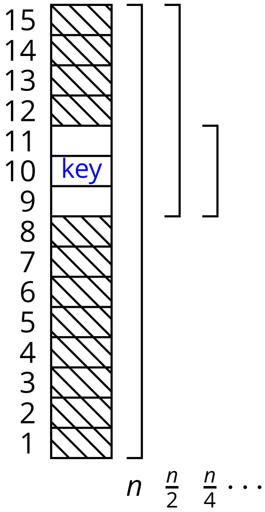
BinarySearch(*A*, *key*)

```
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
```



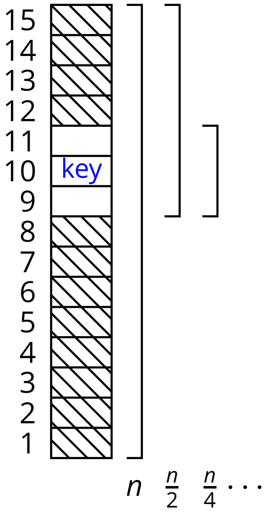
Binary Search

```
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]
```

```
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]
```

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

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]
```

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

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

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

- *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

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

$i = 1$

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

$i = 1$

A	3	7	12	13	34	37	70	75	80

B	1	5	6	7	34	35	40	41	43

$j = 2$

Output: 1

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

MERGE Algorithm

$i = 2$

A	3	7	12	13	34	37	70	75	80

B	1	5	6	7	34	35	40	41	43

$j = 2$

Output: 1 3

$i = 2$

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

$i = 2$

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

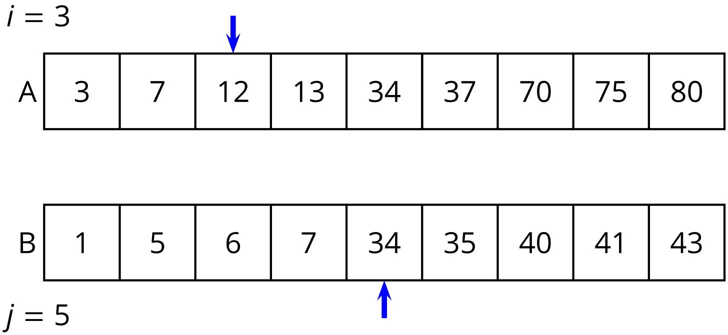
$i = 2$

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

MERGE Algorithm



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

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

$i = 5$

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

$i = 5$

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)

```
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
```

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
```

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

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*

- 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

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

- 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

```
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

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

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

- 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

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

- 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 $A = \langle 2, 36, 5, 21, 8, 13, 11, 20, 5, 4, 1 \rangle$?

- 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[⌊length(A)/2⌋]
```

- 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[⌊length(A)/2⌋]
```

- Is it correct? Yes

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)

$X = \textsc{MergeSort}(A)$

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

- Is it correct? Yes

- How long does it take?

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? Yes

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

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? Yes

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

- 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)
X = MERGESORT(A)
return X[⌊length(A)/2⌋]
```

- Is it correct? Yes

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

- 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

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

- 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[random(1 . . . |A|)]
    A_L, A_v, A_R = ∅
    for i = 1 to |A|
        if A[i] < v
            A_L = A_L ∪ A[i]
        elseif A[i] == v
            A_v = A_v ∪ A[i]
        else A_R = A_R ∪ A[i]
    if k ≤ |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
```