Data structures and Algorithms

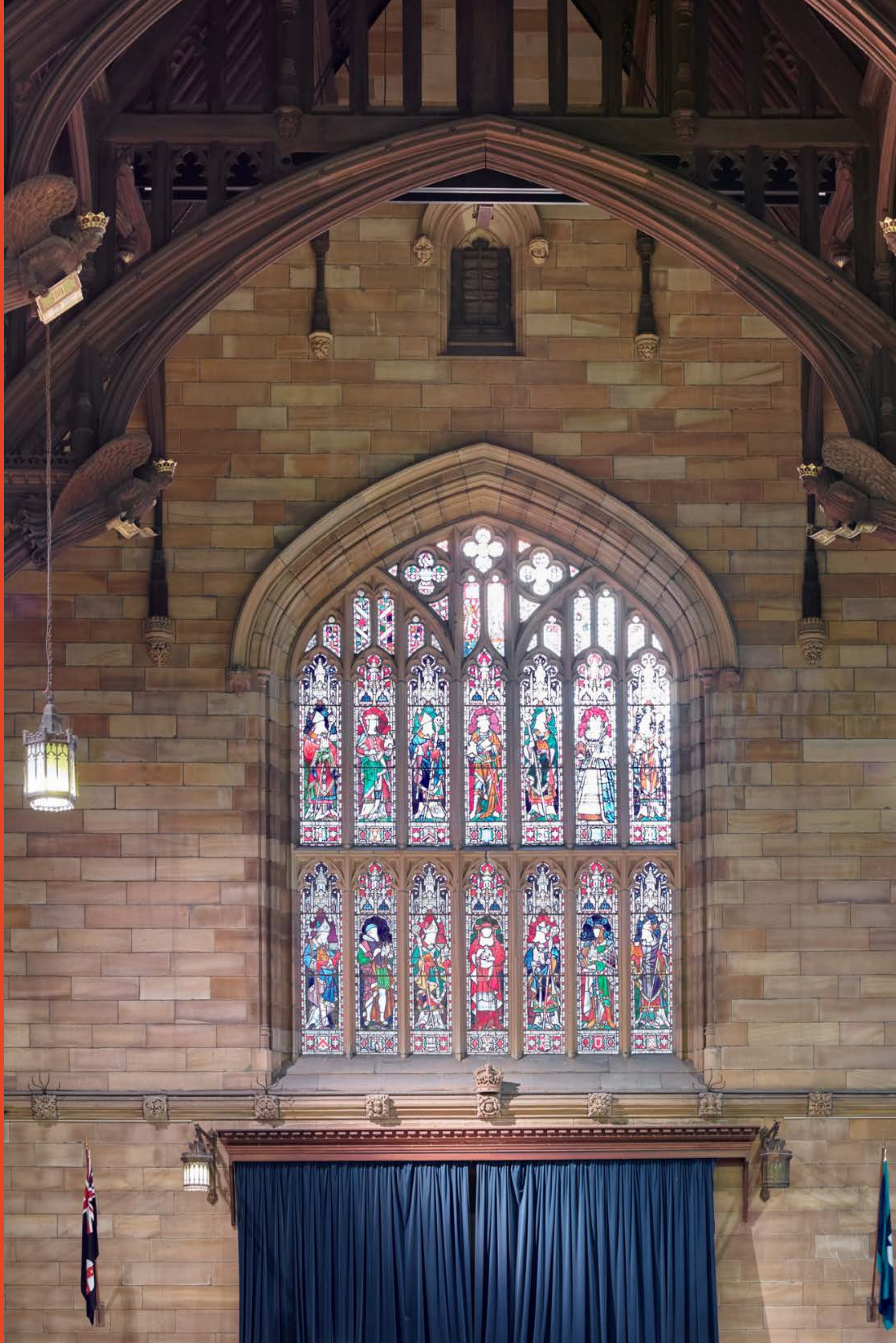
Lecture 11: Divide and Conquer [GT 3.1, 8, 9, 11]

Lecturer: Dr. Karlos Ishac

Co-ordinator: Dr. Ravihansa Rajapakse
School of Computer Science

Some content is taken from the textbook publisher Wiley and previous Co-ordinator Dr. Andre van Renssen.

THE UNIVERSITY OF SYDNEY



Divide and Conquer

Divide and Conquer algorithms can normally be broken into these three parts:

1. Divide If it is a base case, solve directly, otherwise break up the problem into several parts.

2. Recur/Delegate Recursively solve each part [each sub-problem].

3. Conquer Combine the solutions of each part into the overall solution.

Divide and Conquer

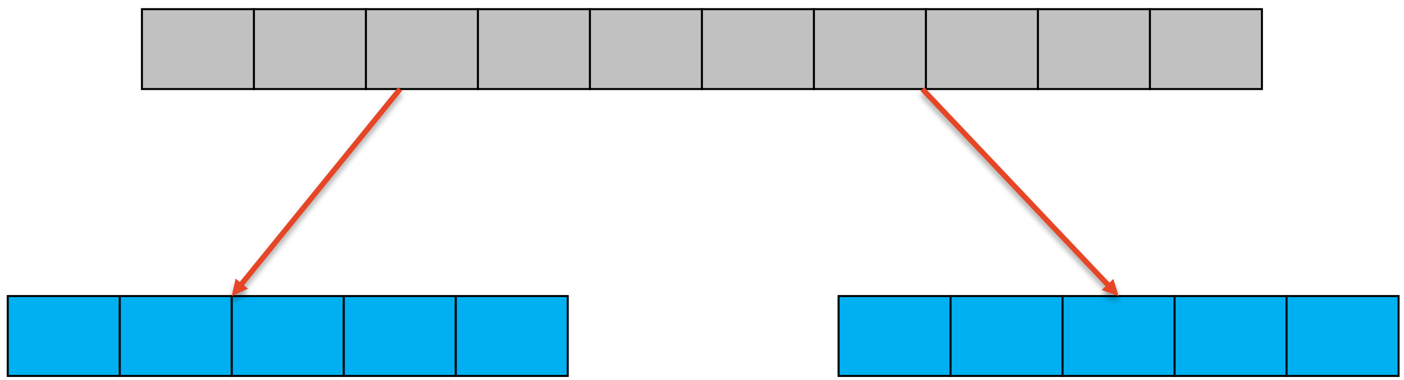
1. Divide If it is a base case, solve directly, otherwise break up the problem into several parts.

Typical base case:

Subproblem of constant size (usually 0 or 1 elements) for which you can compute the solution explicitly

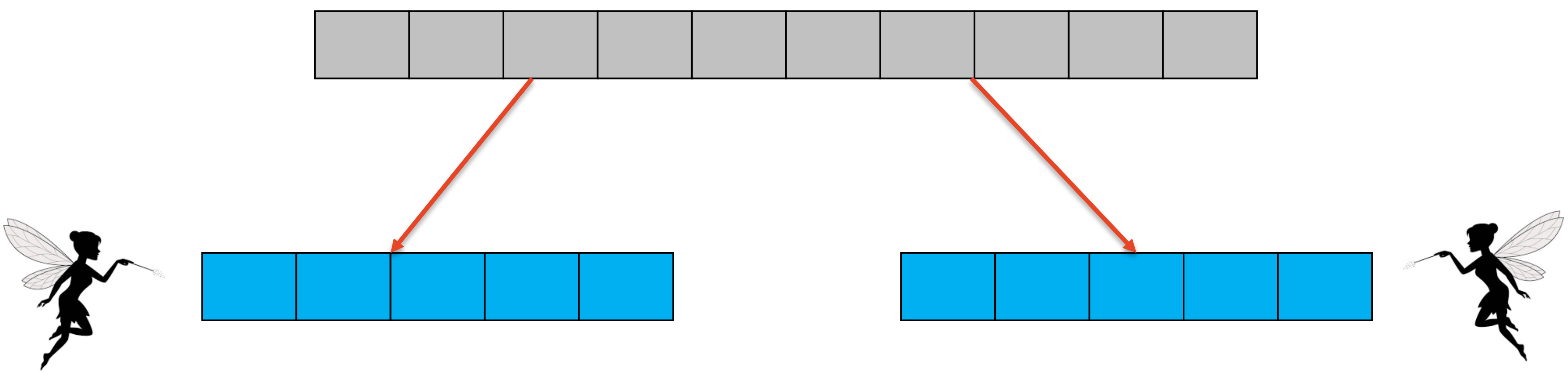
Divide and Conquer

1. Divide If it is a base case, solve directly, otherwise break up the problem into several parts.



Divide and Conquer

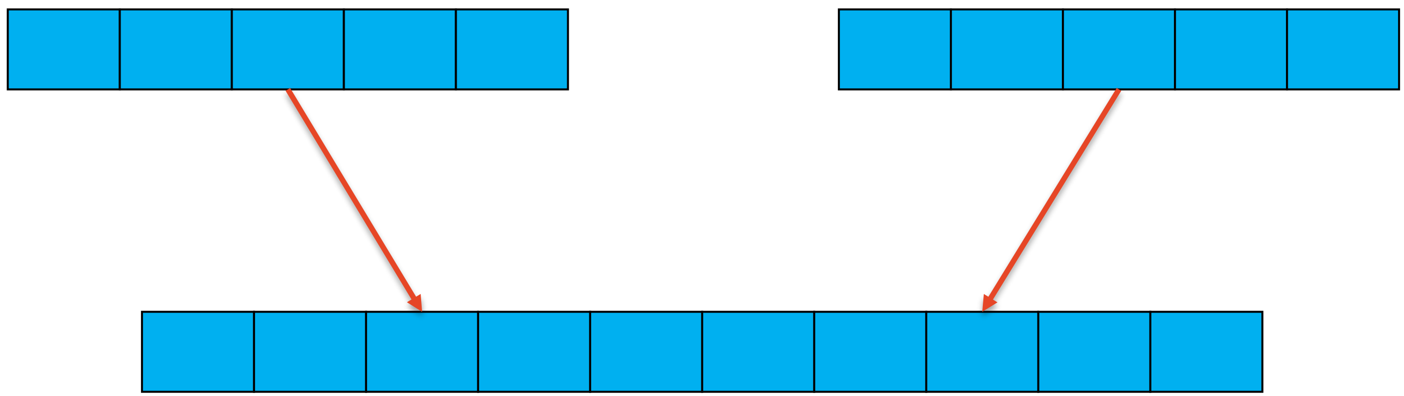
2. Recur/Delegate Recursively solve each part [each sub-problem].



The sub-problems are solved by the Recursion Fairy (similar to induction hypothesis), so we don't have to worry about them.

Divide and Conquer

3. Conquer Combine the solutions of each part into the overall solution.



Searching Sorted Array

Given A sorted sequence S of n numbers $a_0, a_1, \ldots, a_{n-1}$ stored in an array A[0, 1, …, n - 1].

Problem Given a number x, is x in S?

0	2	5	7	12	22	25	37	39	50	55	80

Searching: Naïve Approach

Problem Given a number x, is x in S?

Idea Check every element in turn to see if it is equal to x.

```
for e in S do
    if e equals x then
        return "Yes"
return "No"
```

Found an element equal to x in S

There was no element equal to x in S

0	2	5	7	12	22	25	37	39	50	55	80

Running Time O(n)

Binary Search in sorted A[0 to n-1]

1. If the array is empty, then return "No"
2. Compare x to the middle element, namely $A[\lfloor n/2 \rfloor]$
3. If this middle element is x, then return "Yes"
4. When the middle element is not x: if $A[\lfloor n/2 \rfloor] > x$, then recursively search $A[0 \text{ to } \lfloor n/2 \rfloor - 1]$
5. if $A[\lfloor n/2 \rfloor] < x$, then recursively search $A[\lfloor n/2 \rfloor + 1 \text{ to } n-1]$

0	2	5	7	12	22	25	37	39	50	55	80

Binary Search Pseudocode

```
def binary_search(A, left, right, x):
    # A is sorted and left <= right
    # looking for x in A[left:right]

    if left = right then
        return “unsuccessful”

    mid = floor((left + right) / 2)
    if A[mid] < x then
        return binary_search(A, mid + 1, right, x)
    else if A[mid] > x then
        return binary_search(A, left, mid, x)
    else
        return mid
```

Heads up: pseudocode textbook uses indexing from 1 to n, not 0 to n-1

Binary Search

– Example, search for x=5

0	2	5	7	12	22	25	37	39	50	55	80

Binary Search

– Example, search for x=5

0	2	5	7	12	22	25	37	39	50	55	80

A[6]

Binary Search

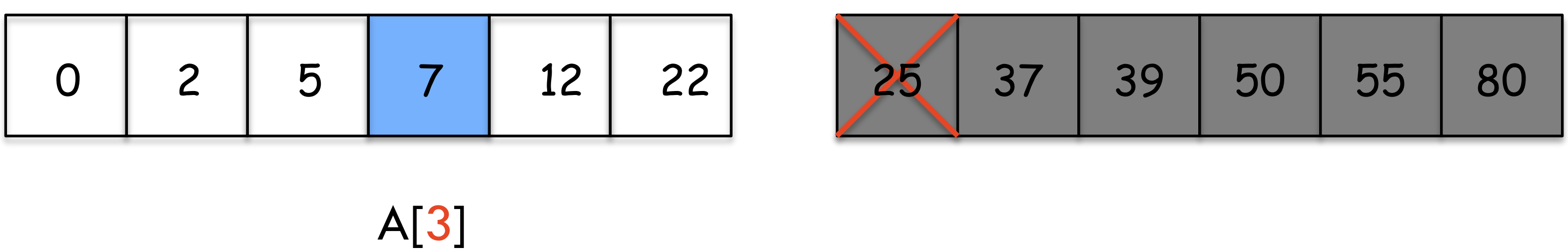
– Example, search for x=5

0	2	5	7	12	22	25	37	39	50	55	80

A[6] $= 25 > 5 = x$

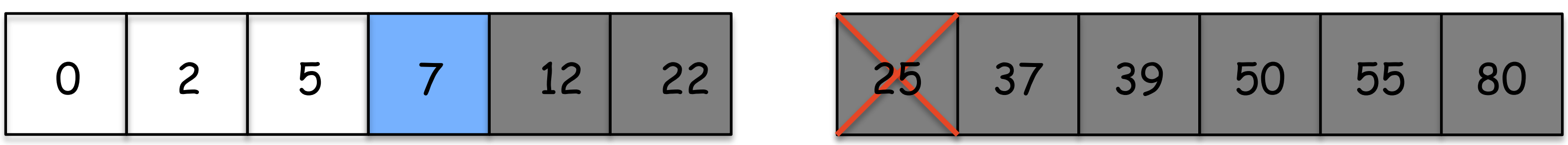
Binary Search

– Example, search for x=5



Binary Search

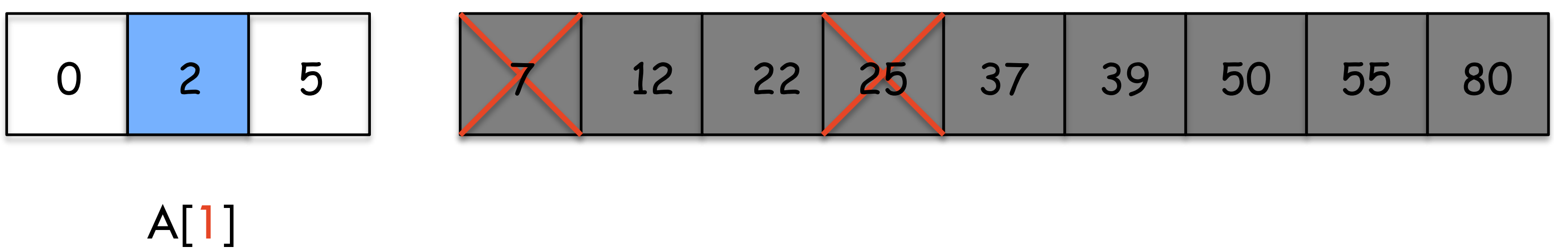
– Example, search for x=5



A[3] $= 7 > 5 = x$

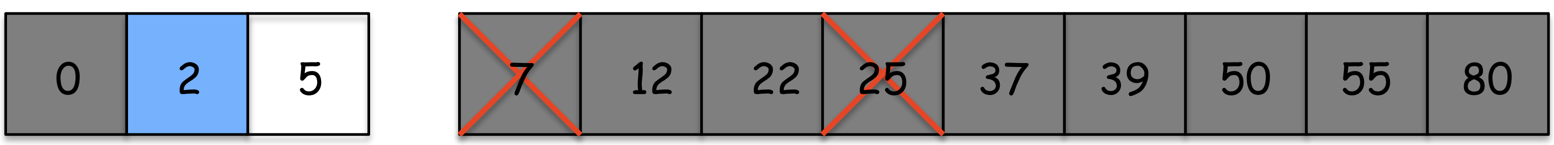
Binary Search

– Example, search for x=5



Binary Search

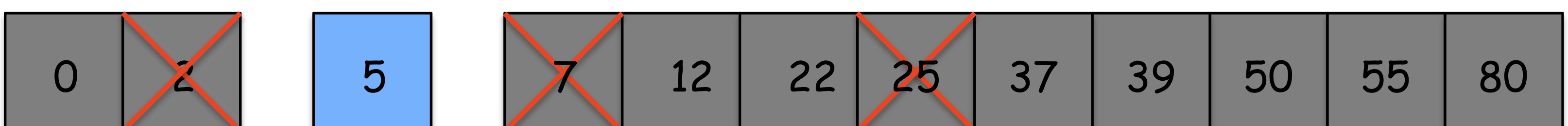
– Example, search for x=5



$A[1] = 2 < 5 = x$

Binary Search

– Example, search for x=5



A[2]

Recurrence formula

An easy way to analyze the time complexity of a divide-and-conquer algorithm is to define and solve a recurrence

Let T(n) be the running time of the algorithm, we need to find out:

– Divide step cost in terms of n
– Recur step(s) cost in terms of T(smaller values)
– Conquer step cost in terms of n

Together with information about the base case, we can set up a recurrence for T(n) and then solve it.

$$T(n) = \begin{cases} \text{“Recur” + “Divide and Conquer”} & \text{for } n > 1 \\ \text{“Base case” (typically } O(1)\text{)} & \text{for } n = 1 \end{cases}$$

Binary search on an array complexity analysis

Divide step (find middle and compare to x) takes $O(1)$

Recur step (solve left or right subproblem) takes $T(n/2)$

Conquer step (return answer from recursion) takes $O(1)$

Now we can set up the recurrence for $T(n)$:

$$T(n) = \begin{cases} T(n/2) + O(1) & \text{for } n > 1 \\ O(1) & \text{for } n = 1 \end{cases}$$

This solves to $T(n) = O(\log n)$, since we can only halve the input $O(\log n)$ times before reaching a base case

Binary search on a linked list complexity analysis

Divide step (find middle and compare to x) takes $O(n)$

Recur step (solve left or right subproblem) takes $T(n/2)$

Conquer step (return answer from recursion) takes $O(1)$

Now we can set up the recurrence for $T(n)$:

$$T(n) = \begin{cases} T(n/2) + O(n) & \text{for } n > 1 \\ O(1) & \text{for } n = 1 \end{cases}$$

This solves to $T(n) = O(n)$, since to access the next index we end up with $n/2 + n/4 + n/8 + \ldots$

Merge-Sort

1. Divide the array into two halves.
2. Recur recursively sort each half.
3. Conquer two sorted halves to make a single sorted array.

1	12	5	16	19	7	23	6	13	20

1	12	5	16	19	7	23	6	13	20	Divide
1	5	12	16	19	6	7	13	20	23	Recur

1	5	6	7	12	13	16	19	20	23	Conquer

Merge-Sort pseudocode

```
def merge_sort(S):
    # base case
    if |S| < 2 then
        return S

    # divide
    mid ← ⌊|S|/2⌋
    left ← S[:mid]    # doesn't include S[mid]
    right ← S[mid:]   # includes S[mid]


    # recur
    sorted_left ← merge_sort(left)
    sorted_right ← merge_sort(right)

    # conquer
    return merge(sorted_left, sorted_right)
```

How?

Merge

Input Two sorted lists.

Output A new merged sorted list.

To merge, we use:

- O(n) comparisons.
- An array to store our results.

1	5	12	16	19

6	7	13	20	23

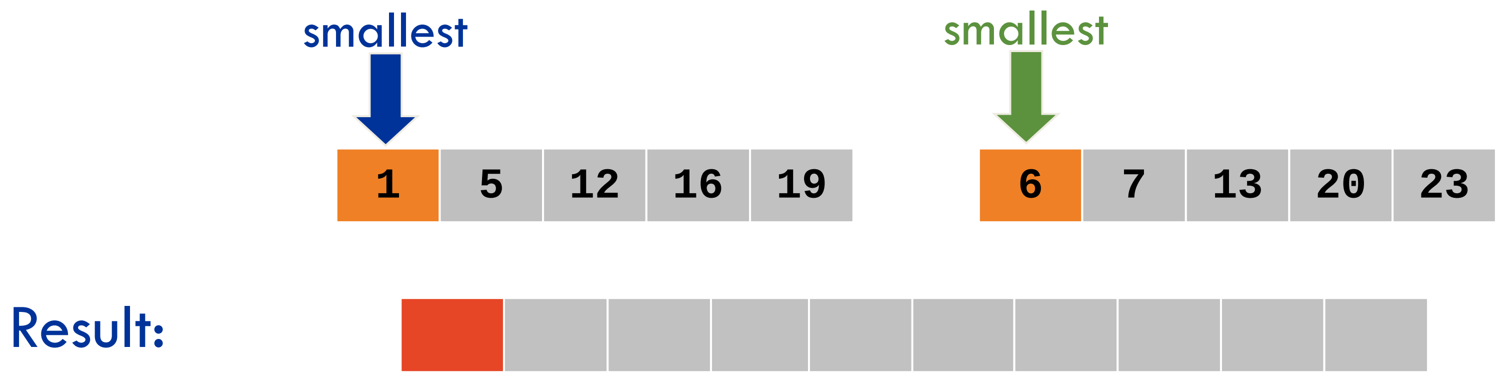
Result:

1	5	6	7	12					

Merge

Merge Algorithm

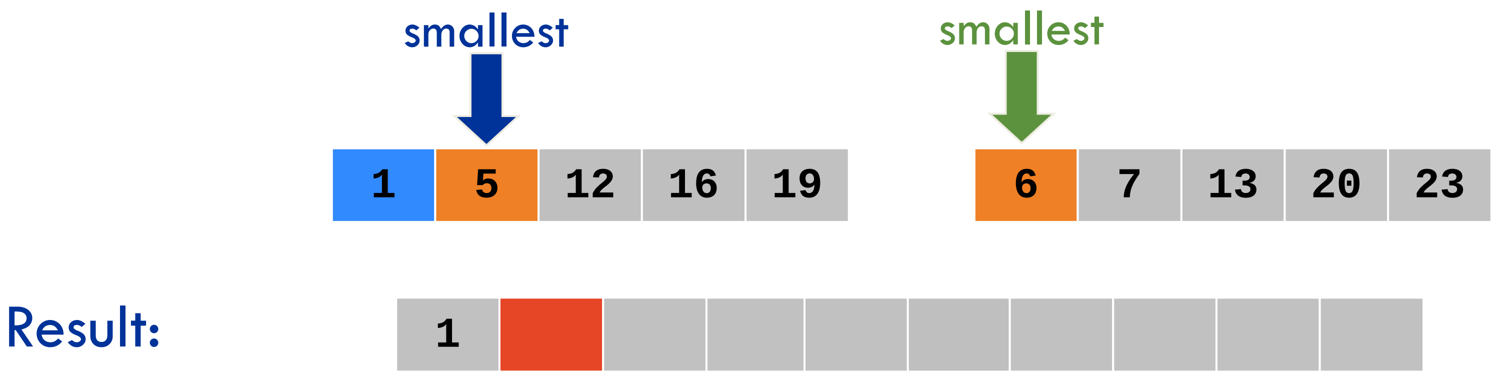
- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into the resultant array.
- Repeat until done.



Merge

Merge Algorithm

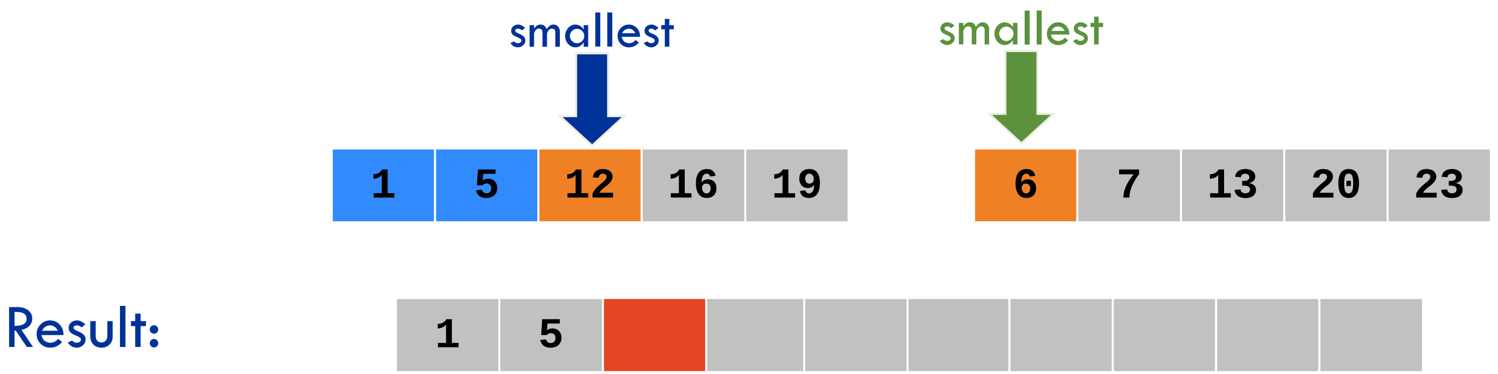
- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into the resultant array.
- Repeat until done.



Merge

Merge Algorithm

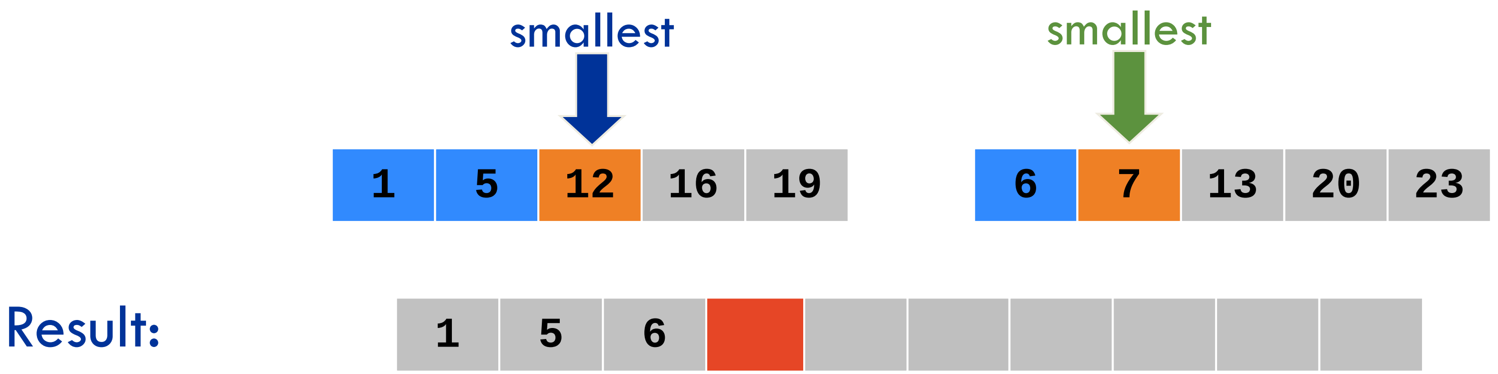
- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into the resultant array.
- Repeat until done.



Merge

Merge Algorithm

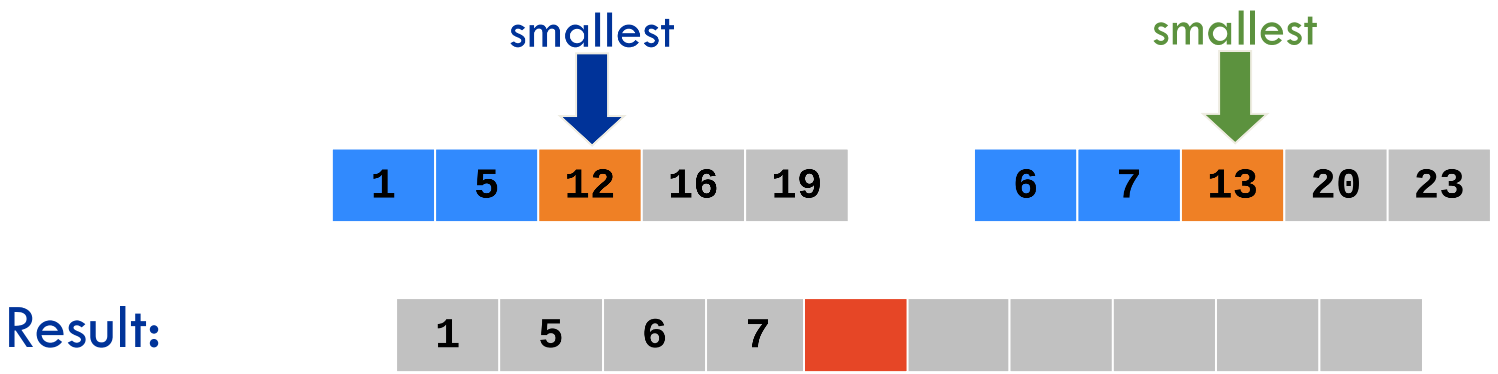
- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into the resultant array.
- Repeat until done.



Merge

Merge Algorithm

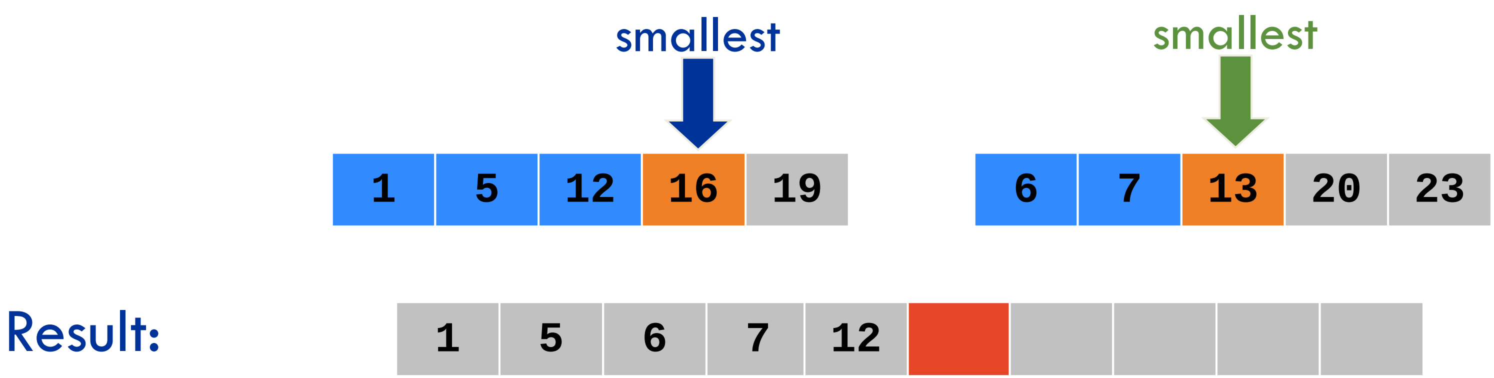
- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into the resultant array.
- Repeat until done.



Merge

Merge Algorithm

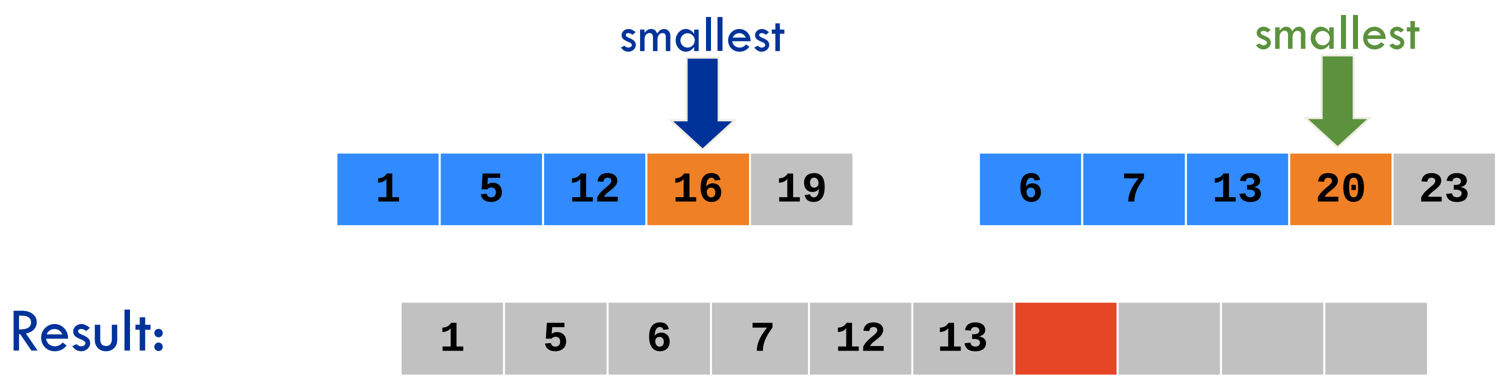
– Keep track of smallest element in each sorted half.
– Insert smallest of two elements into the resultant array.
– Repeat until done.



Merge

Merge Algorithm

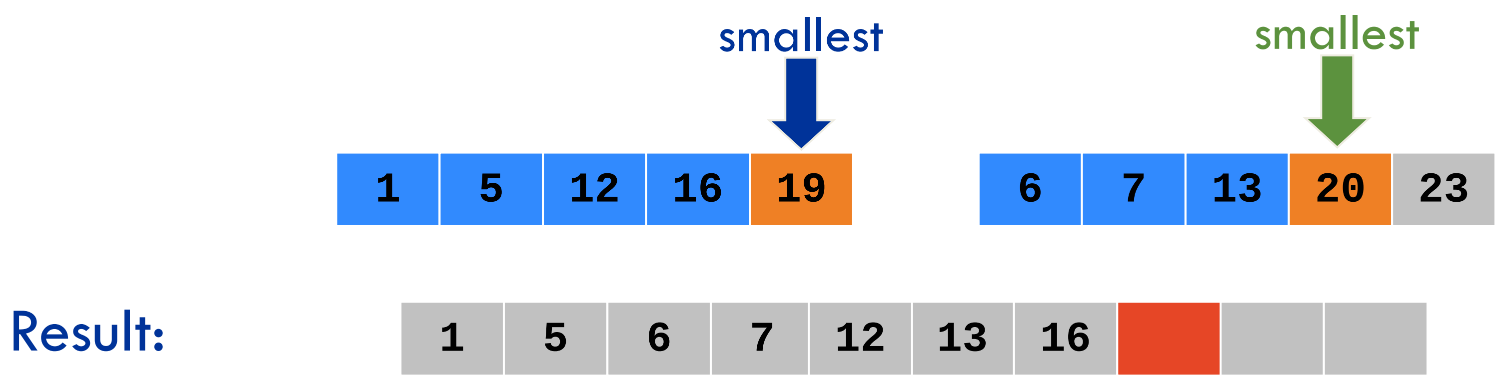
- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into the resultant array.
- Repeat until done.



Merge

Merge Algorithm

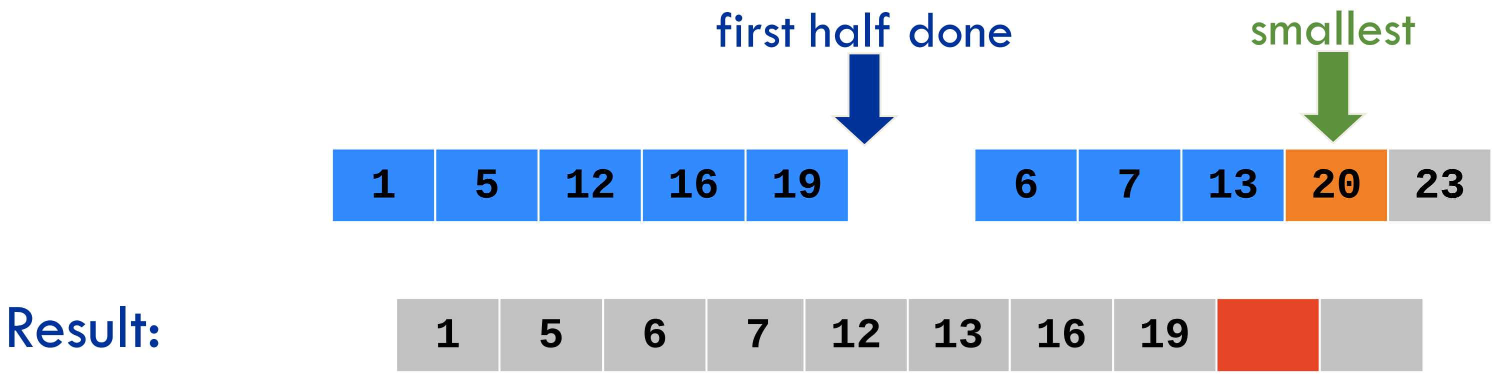
- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into the resultant array.
- Repeat until done.



Merge

Merge Algorithm

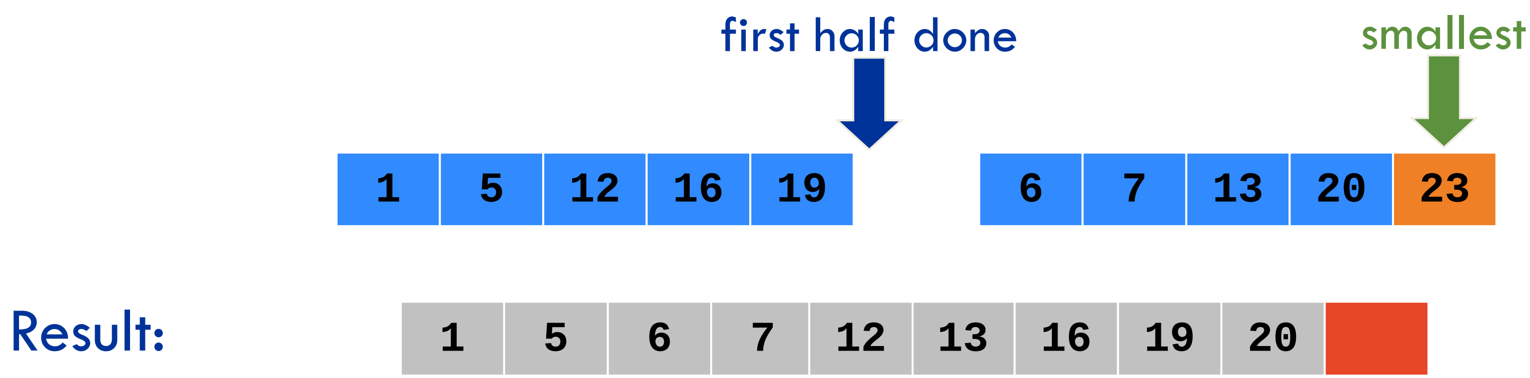
- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into the resultant array.
- Repeat until done.



Merge

Merge Algorithm

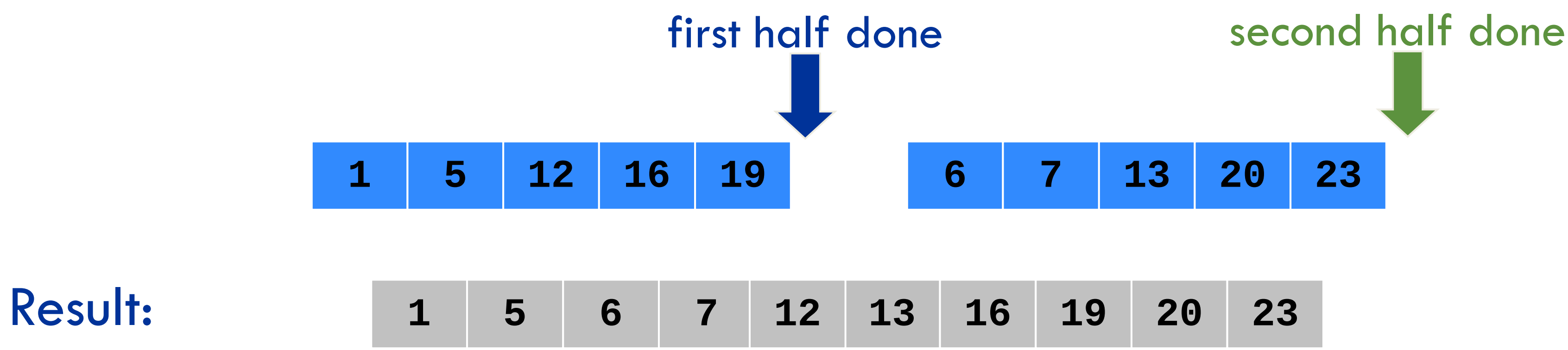
- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into the resultant array.
- Repeat until done.



Merge

Merge Algorithm

- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into the resultant array.
- Repeat until done.



Merge: Implementation

```
def merge(L, R):
    result ← array of length (|L| + |R|)
    l, r ← 0, 0
    while l + r < |result| do
        index ← l + r
        if r ≥ |R| or (l < |L| and L[l] < R[r]) then
            result[index] ← L[l]
            l ← l + 1
        else
            result[index] ← R[r]
            r ← r + 1
    return result
```

Merge-Sort

1. Divide array into two halves.
2. Recur Recursively sort each half.
3. Conquer Merge two sorted halves to make a sorted whole.

1	12	5	16	19	7	23	6	13	20

1	12	5	16	19	7	23	6	13	20	divide
1	5	12	16	19	6	7	13	20	23	recur

1	5	6	7	12	13	16	19	20	23	conquer

Merge sort complexity analysis

Divide step (find middle and split) takes $O(n)$

Recur step (solve left and right subproblem) takes $2\ T(n/2)$

Conquer step (merge subarrays) takes $O(n)$

Now we can set up the recurrence for $T(n)$:

$$T(n) = \begin{cases} 2\ T(n/2) + O(n) & \text{for } n > 1 \\ O(1) & \text{for } n = 1 \end{cases}$$

This solves to $T(n) = O(n \log n)$

Some recurrence formulas with solutions

Recurrence	Solution
$T(n) = 2\,T(n/2) + O(n)$	$T(n) = O(n \log n)$
$T(n) = 2\,T(n/2) + O(\log n)$	$T(n) = O(n)$
$T(n) = 2\,T(n/2) + O(1)$	$T(n) = O(n)$
$T(n) = T(n/2) + O(n)$	$T(n) = O(n)$
$T(n) = T(n/2) + O(1)$	$T(n) = O(\log n)$
$T(n) = T(n-1) + O(n)$	$T(n) = O(n^2)$
$T(n) = T(n-1) + O(1)$	$T(n) = O(n)$

Quick sort

1. Divide Choose a random element from the list as the pivot
   Partition the elements into 3 lists:
   (i) less than, (ii) equal to and (iii) greater than the pivot
2. Recur Recursively sort the less than and greater than lists
3. Conquer Join the sorted 3 lists together

pivot

1	7	6	16	19	7	13	5	13	19

1	6	5	7	7	16	19	13	13	19	divide
1	5	6	7	7	13	13	16	19	19	recur
1	5	6	7	7	13	13	16	19	19	conquer

Quick sort complexity analysis

Divide step (pick pivot and split) takes $O(n)$

Recur step (solve left and right subproblem) takes $T(n_L) + T(n_R)$

Conquer step (merge subarrays) takes $O(n)$

Now we can set up the recurrence for $T(n)$:

$$E[T(n)] = \begin{cases} E[T(n_L) + T(n_R)] + O(n) & \text{for } n > 1 \\ O(1) & \text{for } n = 1 \end{cases}$$

This solves to $E[T(n)] = O(n \log n)$ expected time

(details available on the textbook but not examinable)

Remember

Important:

Simply using Merge-Sort in your algorithm doesn't make your algorithm a divide and conquer algorithm.

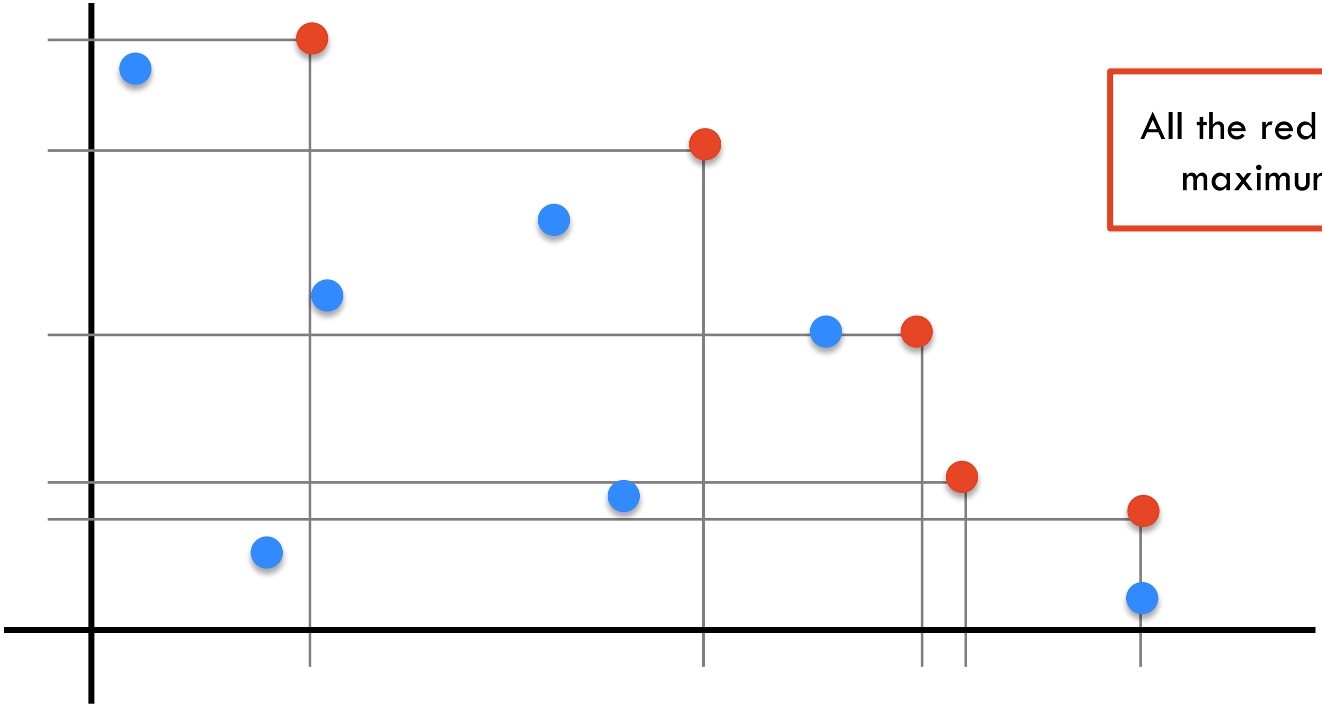
Example:

A greedy algorithm first sorts the input in some way and then processes the items one by one in that order. Using Merge-Sort for the sorting step doesn't change the fact that the algorithm computes the solution in a greedy way.

Maxima-Set (Pareto frontier)

Definition A point is maximum in a set if all other points in the set have either a smaller x- or smaller y-coordinate.

Problem Given a set S of n distinct points in the plane (2D), find the set of all maximum points.



All the red points are maximum points.

Maxima-Set: Naïve Solution

Idea: Check every point (one at a time) to see if it is a maximum point in the set S.

To check if point p is a maximum point in S:

```
for q in S do
    if q ≠ p and q.x ≥ p.x and q.y ≥ p.y then
        return "No"
return "Yes"
```

There is a point q that dominates p

Maxima-Set: Naïve Solution

Idea: Check every point (one at a time) to see if it is a maximum point in the set S.

To check if point p is a maximum point in S:

```
for q in S do
    if q ≠ p and q.x ≥ p.x and q.y ≥ p.y then
        return "No"
return "Yes"
```

Naïve algorithm to find the maxima-set of S:

```
maximaSet ← empty list
for p in S do
    if p is a maximum point in S then
        add p to the maximaSet
return maximaSet
```

Maxima-Set: Naïve Solution

Idea: Check every point (one at a time) to see if it is a maximum point in the set S.

To check if point p is a maximum point in S:

```
for q in S do
    if q ≠ p and q.x ≥ p.x and q.y ≥ p.y then
        return "No"
return "Yes"
```

$O(n)$

Naïve algorithm to find the maxima-set of S:

```
maximaSet ← empty list
for p in S do
    if p is a maximum point in S then
        add p to the maximaSet
return maximaSet
```

$O(n)$ $O(n^2)$

Maxima-Set

Preprocessing Sort the points by increasing x coordinate and store them in an array. Note: we only do this once. Break ties in x by sorting by increasing y coordinate.

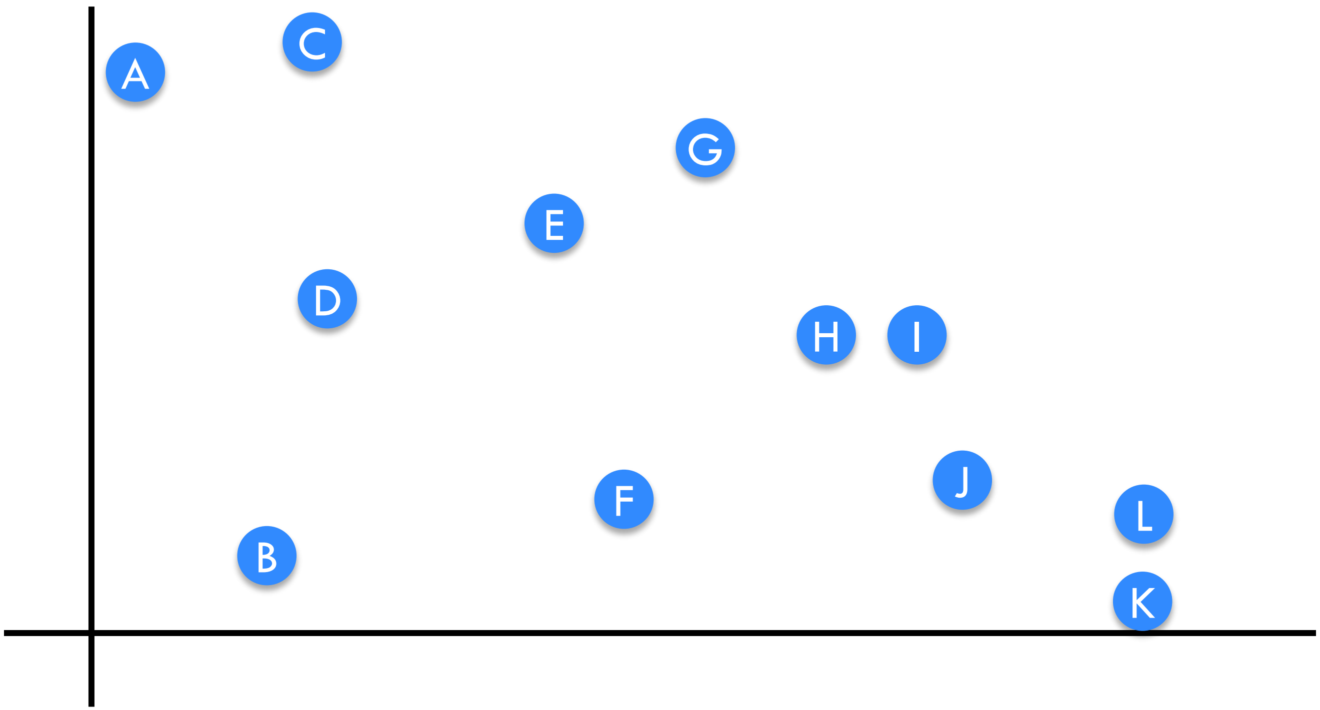
Divide sorted array into two halves.

Recur recursively find the MS of each half.

Conquer compute the MS of the union of Left and Right MS

Maxima-Set

Preprocessing Sort the points by increasing x coordinate and store them in an array. Note: we only do this once. Break ties in x by sorting by increasing y coordinate.

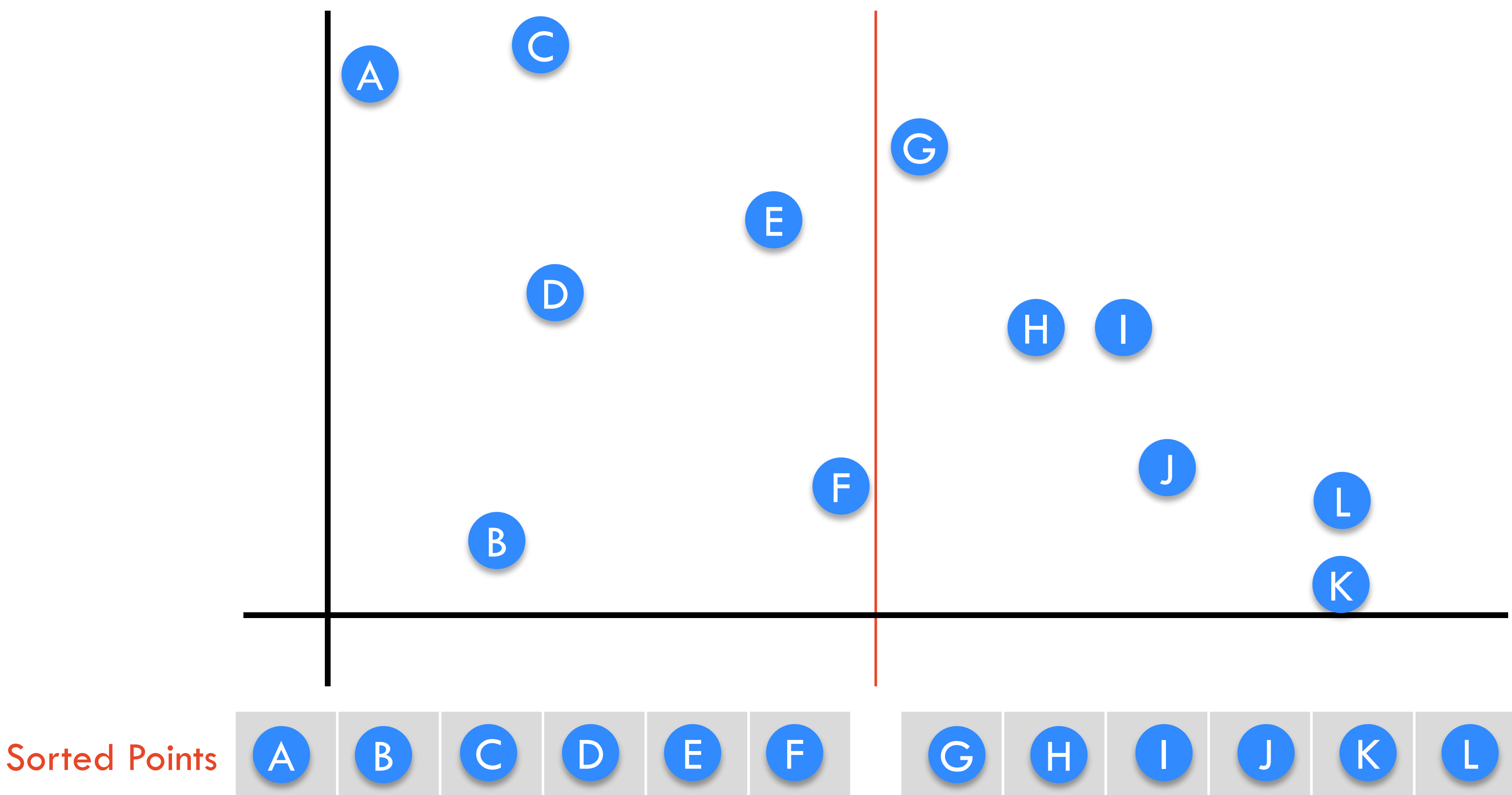


Sorted Points

A	B	C	D	E	F	G	H	I	J	K	L

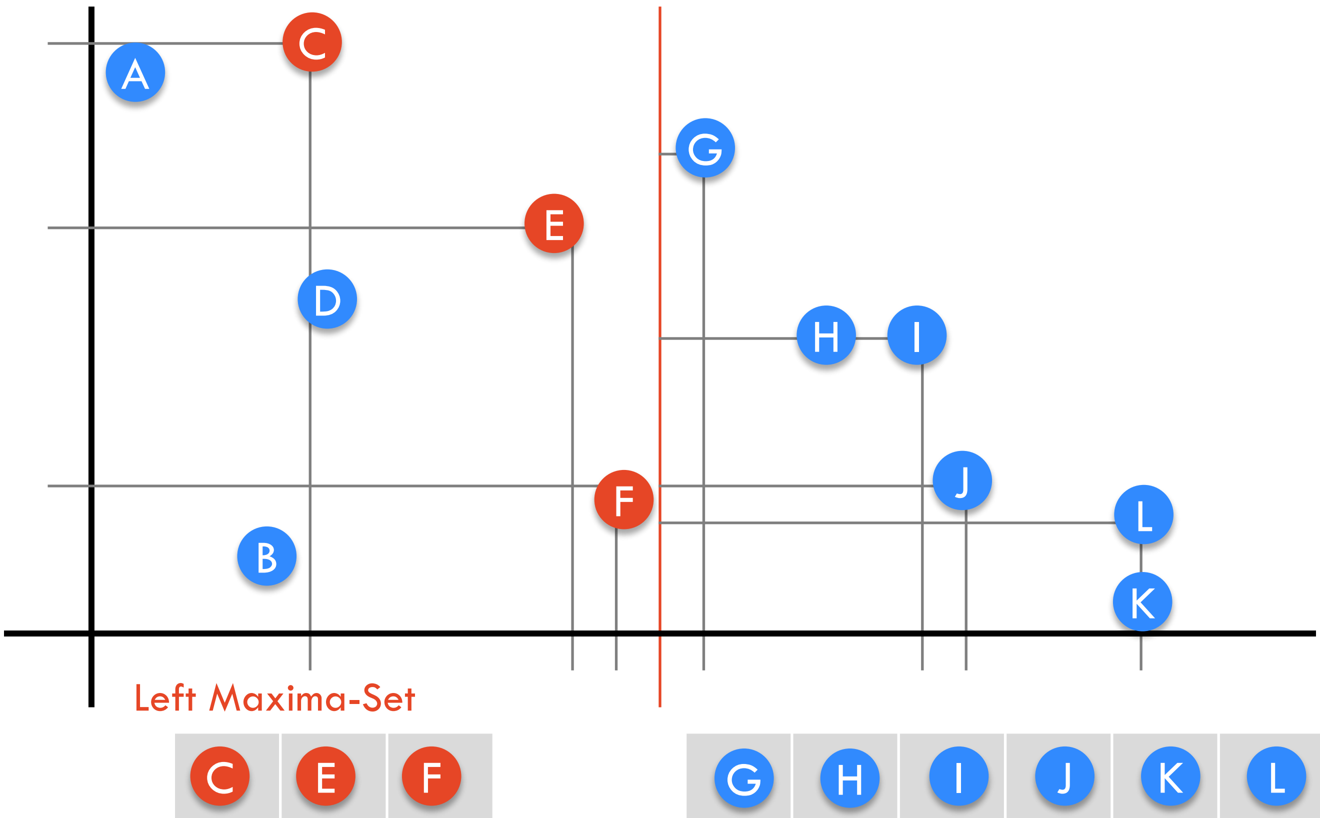
Maxima-Set

Divide array into two halves.



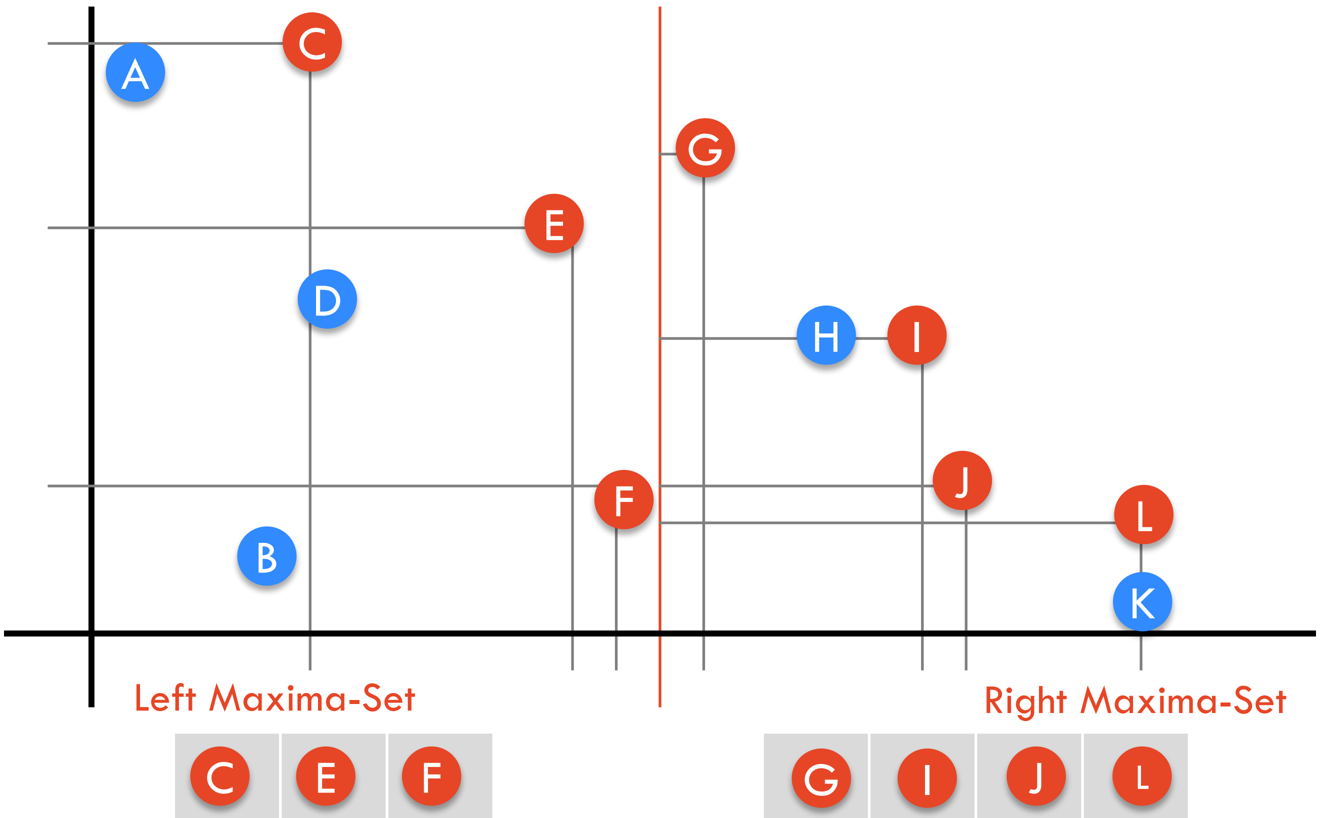
Maxima-Set

Recur recursively find the Maxima-Set of each half.



Maxima-Set

Recur recursively find the Maxima-Set of each half.



Maxima-Set

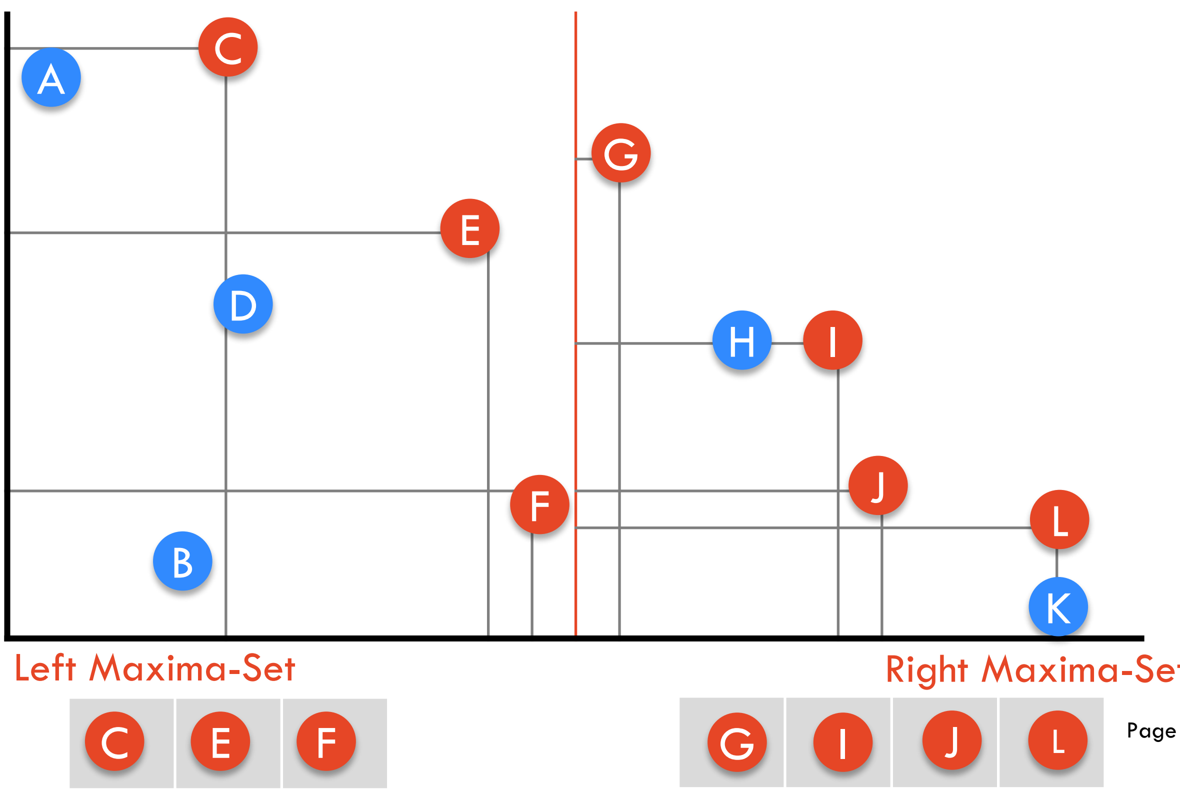
Conquer

1. Find the highest point p in the Right MS

$p =$ G

Observations:

1. Every point in MS of the whole is in Left MS or Right MS
2. Every point in Right MS is in MS of the whole
3. Every point in Left MS is either in MS of the whole or is dominated by p

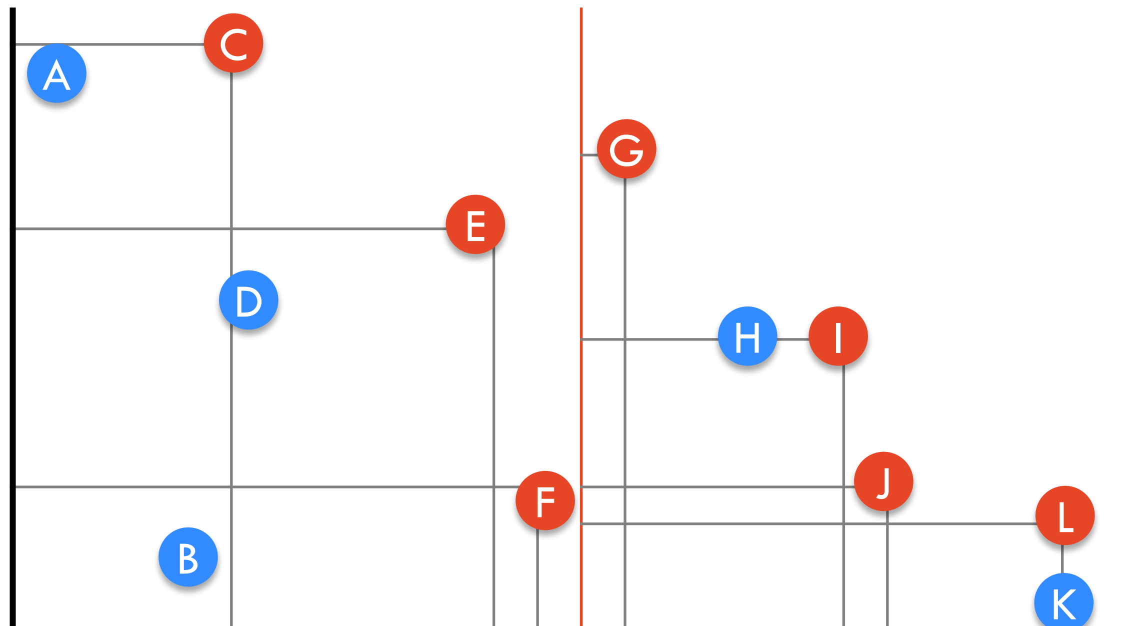


Maxima-Set

Conquer

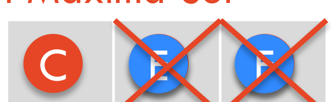
1. Find the highest point *p* in the Right MS
2. Compare every point *q* in the Left MS to this point.
   If q.y > p.y, add q to the Merged MS
3. Add every point in the Right MS to the Merged MS

p = G

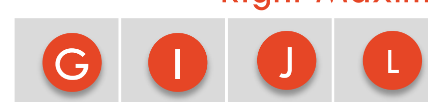


Merged Maxima-Set

Left Maxima-Set



Right Maxima-Set

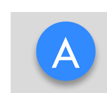


Maxima-Set

Base case a single point.

The MS of a single point is the point itself.

Maxima-Set



Maxima-Set: Analysis

Preprocessing Sort the points by increasing x coordinate and store them in an array. Note: we only do this once. Break ties in x by sorting by increasing y coordinate. $O(n \log n)$

Divide sorted array into two halves. $O(n)$

Recur recursively find the MS of each half. $2T(^n/_2)$

Conquer compute the MS of the union of Left and Right MS $O(n)$

1. Find the highest point *p* in the Right MS
2. Compare every point *q* in the Left MS to this point.
   If q.y > p.y, add q to the Merged MS
3. Add every point in the Right MS to the Merged MS

$$T(n) = 2T(^n/_2) + O(n) = O(n \log n)$$

Overall Running Time: pre-processing + $T(n) = O(n \log n)$

Integer multiplication

Given two n-digit integers x and y

Problem compute the product x y

While this seems like recreational mathematics, it does have real applications: Public key encryption is based on manipulating integers with thousands of bits.

Integer multiplication: Naïve approach

Given two n-digit integers x and y

Problem compute the product $x\ y$

Suppose we wanted to do it by hand. We assume that two digits can be multiplied or added in constant time

In primary school we all learn an algorithm for this problem that performs $\Theta(n^2)$ operations

Integer multiplication: Divide and conquer

Let $x = x_1 \, 2^{n/2} + x_0$ and $y = y_1 \, 2^{n/2} + y_0$

Then $x \, y = x_1 \, y_1 \, 2^n + x_1 \, y_0 \, 2^{n/2} + x_0 \, y_1 \, 2^{n/2} + x_0 \, y_0$

We can compute the product of two n-digit numbers by making 4 recursive calls on n/2-digit numbers and then combining the solutions to the subproblems.

Integer multiplication: Divide and conquer

```
def multiply(x, y):
  // x and y are positive integers represented in binary
  if x = 0 or y = 0 then return 0
  if x = 1 then return y
  if y = 1 then return x

  // recursive case
  let x_1 and x_0 be such that x = x_1 2^{n/2} + x_0
  let y_1 and y_0 be such that y = y_1 2^{n/2} + y_0

  return multiply(x_1, y_1) 2^n +
       (multiply(x_1, y_0) + multiply(x_0, y_1)) 2^{n/2} +
       multiply(x_0, y_0)
```

Integer multiplication: Complexity analysis

Recall $x\,y = x_1\,y_1\,2^n + (x_1\,y_0 + x_0\,y_1)\,2^{n/2} + x_0\,y_0$

Divide step (produce halves) takes $O(n)$

Recur step (solve subproblems) takes $4\,T(n/2)$

Conquer step (add up results) takes $O(n)$

$$T(n) = \begin{cases} 4\,T(n/2) + O(n) & \text{for } n > 1 \\ O(1) & \text{for } n = 1 \end{cases}$$

This solves to $T(n) = O(n^2)$. No better than naïve!!!

Integer multiplication: Divide and conquer v2.0

Let $x = x_1\ 2^{n/2} + x_0$ and $y = y_1\ 2^{n/2} + y_0$

$$x\,y = x_1\,y_1\ 2^n + (x_1\,y_0 + x_0\,y_1)\ 2^{n/2} + x_0\ y_0$$
$$(x_1 + x_0)\ (y_1 + y_0) = x_1\,y_1 + x_1\,y_0 + x_0\,y_1 + x_0\ y_0$$

We can compute the product of two n-digit numbers by making 3 recursive calls on $n/2$-digit numbers and then combining the solutions to the subproblems.

Integer multiplication: Divide and conquer

```
def multiply(x, y):
  // base case
  ⋮
  // recursive case
  let x₁ and x₀ be such that x = x₁ 2^(n/2) + x₀
  let y₁ and y₀ be such that y = y₁ 2^(n/2) + y₀

  first_term ← multiply(x₁, y₁)
  last_term ← multiply(x₀, y₀)
  other_term ← multiply(x₁ + x₀, y₁ + y₀)

  return first_term 2^n +
       (other_term - first_term - last_term) 2^(n/2) +
       last_term
```

Integer multiplication: Complexity analysis

Divide step (produce halves) takes $O(n)$

Recur step (solve subproblems) takes $3\ T(n/2)$

Conquer step (add up results) takes $O(n)$

$$T(n) = \begin{cases} 3\ T(n/2) + O(n) & \text{for } n > 1 \\ O(1) & \text{for } n = 1 \end{cases}$$

This solves to $T(n) = O(n^{\log_2 3})$, where $\log_2 3 \approx 1.6$

Better than naïve!!!

Logarithms facts

Base exchange rule:

$$\log_a x = (\log_b x)/(\log_b a)$$

Product rule:

$$\log_a (xy) = (\log_a x) + (\log_a y)$$

Power rule:

$$\log_a x^b = b \log_a x$$

Master Theorem

Let f(n) and T(n) be defined as follows:

$$T(n) = \begin{cases} a\,T(n/b) + f(n) & \text{for } n \geq d \\ c & \text{for } n < d \end{cases}$$

Depending on a, b and f(n) the recurrence solves to:

1. if $f(n) = O(n^{\log_b a - \varepsilon})$ for $\varepsilon > 0$ then $T(n) = \Theta(n^{\log_b a})$,
2. if $f(n) = \Theta(n^{\log_b a} \log^k n)$ for $k \geq 0$ then $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$,
3. if $f(n) = \Omega(n^{\log_b a + \varepsilon})$ and $a\,f(n/b) \leq \delta\, f(n)$ for $\varepsilon > 0$ and $\delta < 1$ then $T(n) = \Theta(f(n))$,

Note: If f(n) is given as big-O, you can only conclude T(n) as big-O (not Θ).

Note: You should be able to solve all recurrences in this class using unrolling, but if you are comfortable using the Master Theorem, go for it.

The Master Theorem

$$T(n) = \begin{cases} a\,T(n/b) + f(n) & \text{for } n \geq d \\ c & \text{for } n < d \end{cases}$$

Examples

1. $T(n) = 8T(n/2) + n^2$

   $a=8$, $b=2$, $f(n)=n^2$, $\log_b(a) \rightarrow \log_2(8) = 3$; so $f(n) = O(n^{\log_b a - \varepsilon})$ (case 1)

   $T(n) \in \Theta(n^3)$

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

   $a=2$, $b=2$, $f(n)=n$, $\log_b(a) \rightarrow \log_2(2) = 1$; so $f(n) = \Theta(n^{\log_b a} \log^k n)$ (case 2 with $k=0$)

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

3. $T(n) = 2T(n/2) + O(n^2)$

   $a=2$, $b=2$, $f(n)=n^2$, $\log_b(a) \rightarrow \log_2(2) = 1$; so $f(n) = \Omega(n^{\log\ a + \varepsilon})$ (case 3)

   $T(n) \in O(n^2)$

Selection

Given an unsorted array A holding n numbers and an integer k, find the kth smallest number in A

Trivial solution: Sort the elements and return kth element

Can we do better than $O(n \log n)$?

Yes, with divide and conquer!

First attempt

1. Divide find the median ($\lfloor n/2 \rfloor$th element for simplicity) and split array on the halves, $\leq$ and $>$ than the median
2. Recur if k $\leq \lfloor n/2 \rfloor$ find kth element on smaller half
   if k $> \lfloor n/2 \rfloor$ find (k-$\lfloor n/2 \rfloor$)th element on larger half
3. Conquer return value of the recursive call

1	12	5	16	19	7	23	6	13	20

k = 6
n = 10

1	12	5	7	6

12

16	19	23	13	20

Divide

16	19	23	13	20

Recur

k = 1

13

Conquer

Selection time complexity

Divide step (find median and split) takes at least O(n)

Recur step (solve left or right subproblem) takes T(n/2)

Conquer step (return recursive result) takes O(1)

If we could compute the median in O(n) time then:

$$T(n) = \begin{cases} T(n/2) + O(n) & \text{for } n > 1 \\ O(1) & \text{for } n = 1 \end{cases}$$

This solves to T(n) = O(n) but only if we can solve the median problem, which is in fact a special case of selection with $k = \lfloor n/2 \rfloor$

Second attempt: Approximating the median

We don't need the exact median. Suppose we could find in $O(n)$ time an element x in A such that

$$|A| \,/\, 3 \le \text{rank}(A, x) \le 2\, |A| \,/\, 3$$

Then we get the recurrence

$$T(n) = \begin{cases} T(2n/3) + O(n) & \text{for } n > 1 \\ O(1) & \text{for } n = 1 \end{cases}$$

Which again solves to $T(n) = O(n)$

To approximate the median we can use a recursive call!

Median of 3 medians

Consider the following procedure

- Partition A into $|A| / 3$ groups of 3
- For each group find the median (brute force)
- Let x be the median of the medians (computed recursively)

We claim that x has the desired property

$$|A| / 3 \leq \text{rank}(A, x) \leq 2|A| / 3$$

Half of the groups have a median that is smaller/larger than x, and each group has two elements smaller/larger than x, thus

elements smaller than $x > 2 (|A| / 6) = |A| / 3$
elements greater than $x > 2 (|A| / 6) = |A| / 3$

Median of 3 medians

Let x be the median of the medians, then

$$|A| / 3 \le \text{rank}(A, x) \le 2|A| / 3$$

1	12	5	16	19	7	23	6	13

1	12	5	16	19	7	23	6	13
1	5	12	7	16	19	6	13	23
1	5	12	6	13	23	7	16	19

elements smaller than x > 2 (|A| / 6) = |A| / 3
elements greater than x > 2 (|A| / 6) = |A| / 3

Median of 3 median time complexity

We don't need the exact median. With a recursive call on $n/3$ elements, we can find x in A such that

$$|A|/3 < \text{rank}(A, x) < 2|A|/3$$

Then we get the recurrence

$$T(n) = T(2\,n \,/\, 3) + T(n \,/\, 3) + O(n)$$

Which solves to $T(n) = O(n \log n)$

No better than sorting!

Median of 5 medians

We don't need the exact median. With a recursive call on $n/5$ elements, we can find x in A such that

$$3|A|/10 < \text{rank}(A, x) < 7|A|/10$$

Then we get the recurrence

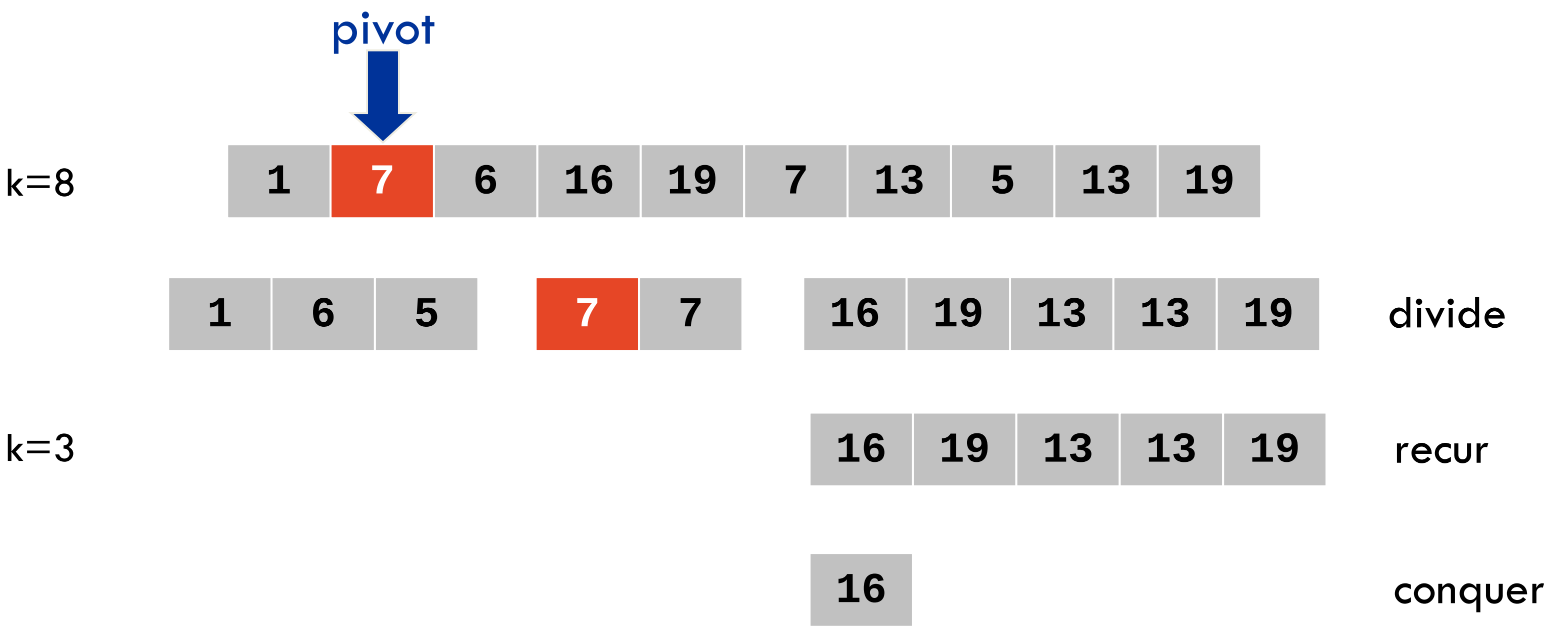
$$T(n) = T(7n / 10) + T(n / 5) + O(n)$$

Which solves to $T(n) = O(n)$

Asymptotically faster than sorting!

Quick selection

1. Divide Choose a random element from the list as the pivot
   Partition the elements into 3 lists:
   (i) less than, (ii) equal to and (iii) greater than the pivot
2. Recur Recursively select right element from correct list
3. Conquer Return solution to recursive problem



Quick selection complexity analysis

Divide step (pick pivot and split) takes $O(n)$

Recur step (solve left and right subproblem) takes $T(n')$

Conquer step (return solution) takes $O(1)$

Now we can set up the recurrence for $T(n)$:

$$E[T(n)] = \begin{cases} E[T(n')] + O(n) & \text{for } n > 1 \\ O(1) & \text{for } n = 1 \end{cases}$$

This solves to $E[T(n)] = O(n)$

(details available on the textbook but not examinable)