# COMP2123/2823/9123
# Data structures and Algorithms
## Lecture 8: Divide and Conquer
## [GT 8]

Dr. Julian Mestre
School of Computer Science

*Some content is taken from material provided by the textbook publisher Wiley.*

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 Recursively solve each part [each sub-problem].

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:
    if e equals x:
        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]

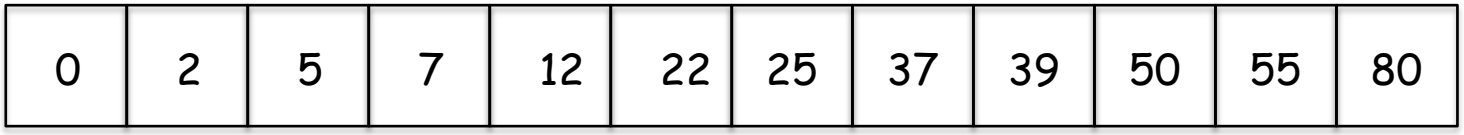
1. If the array is empty, then "No"
2. Compare x to the middle element, namely A[⌈n/2⌉]
3. If this middle element is x, then "Yes"
4. When the middle element is not x: if A[⌈n/2⌉]>x, then recursively search A[0 to ⌈n/2⌉-1]
5. if A[⌈n/2⌉]<x, then recursively search A[⌈n/2⌉+1 to n]

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

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

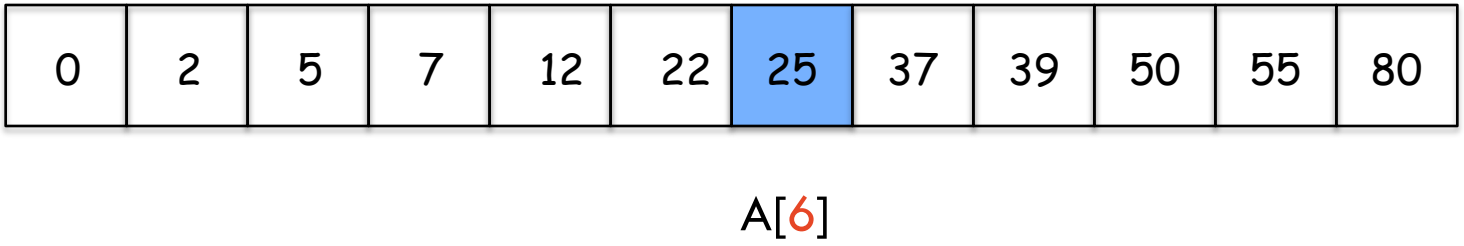
# Binary Search

- Example, search for x=5



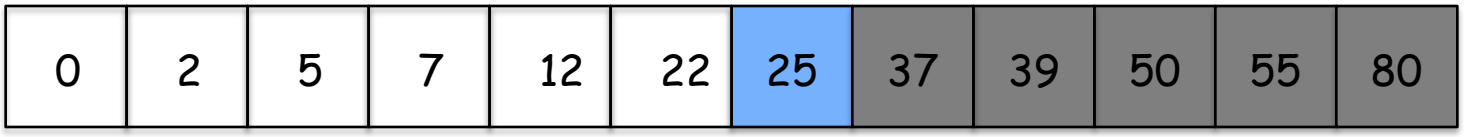
# Binary Search

- Example, search for x=5



# Binary Search

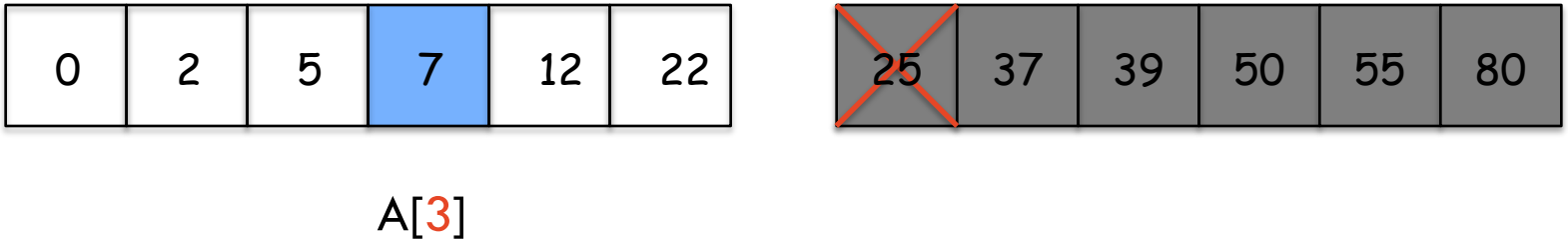
- Example, search for x=5




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

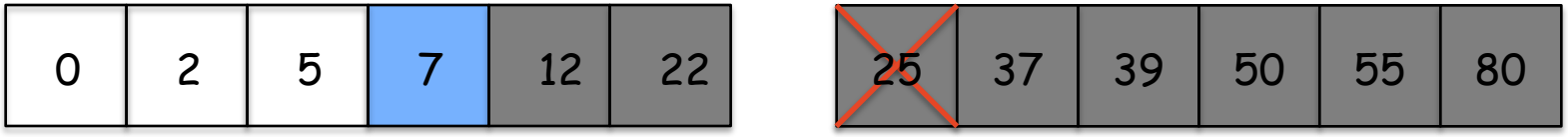
# Binary Search

- Example, search for x=5



# Binary Search

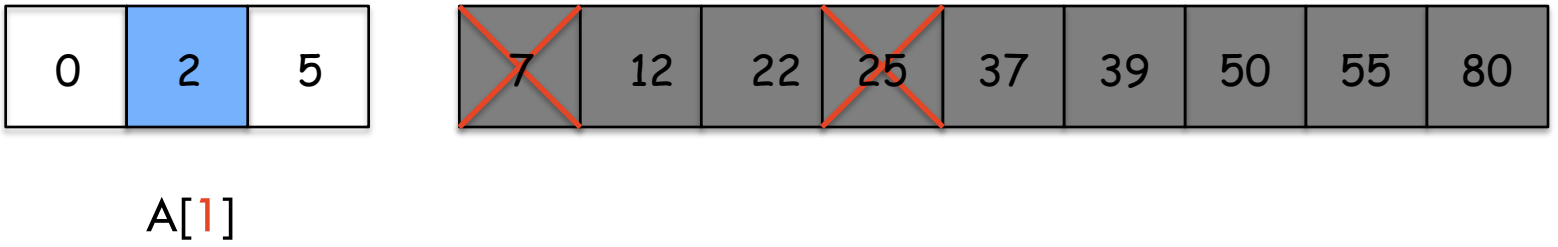
- Example, search for x=5




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

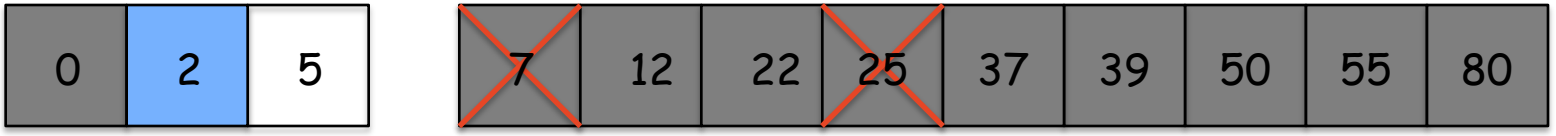
# Binary Search

- Example, search for x=5



# Binary Search

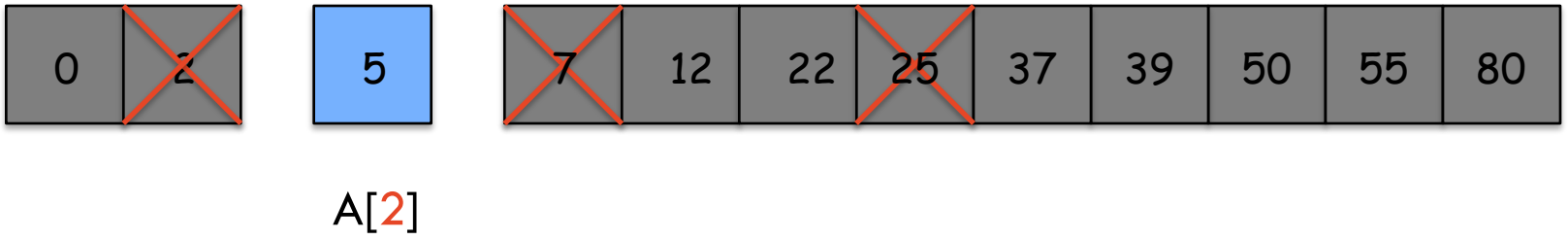
- Example, search for x=5




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

# Binary Search

- Example, search for x=5



# Binary search correctness

Invariant: If $x$ is in $A$ before the divide step, then $x$ is in $A$ after the divide step

- if $A[\lceil n/2 \rceil] > x$, then $x$ must be $A[0 \text{ to } \lceil n/2 \rceil - 1]$
- if $A[\lceil n/2 \rceil] < x$, then $x$ must be in $A[\lceil n/2 \rceil + 1 \text{ to } n]$

Every divide step leads to a smaller array. Thus, if $x$ in not in $A$ then eventually we reach the empty array and return "No"

# Recurrence formula

An easy way to analyze the time complexity of 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.

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

# Binary search on an 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 $O(1)$

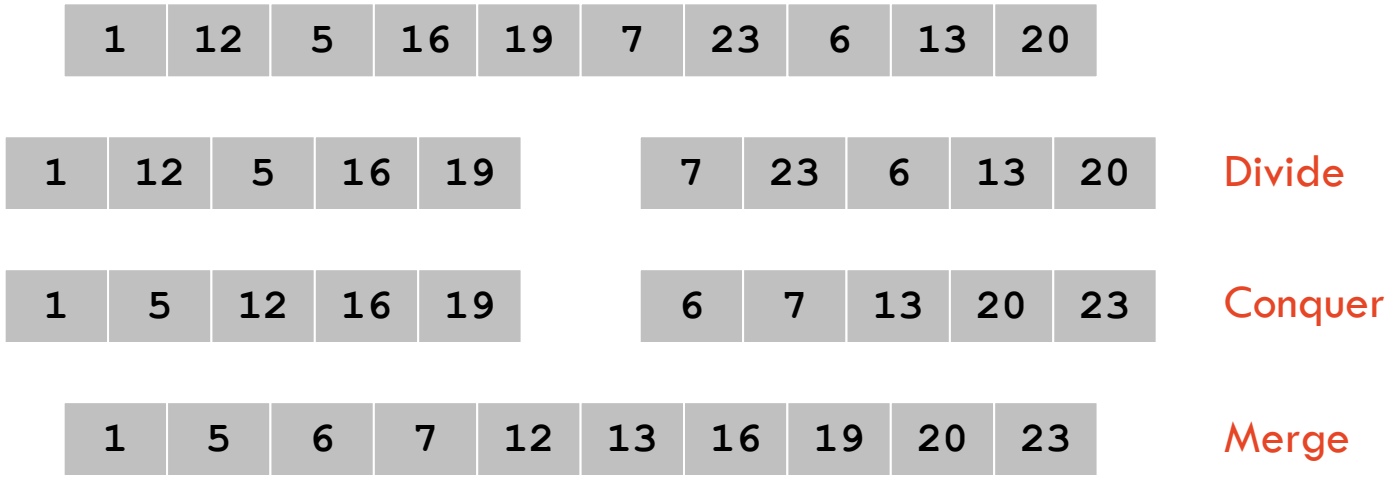
Now we can set up the recurrence for $T(n)$:

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

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

# 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.



# Merge-Sort pseudocode

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

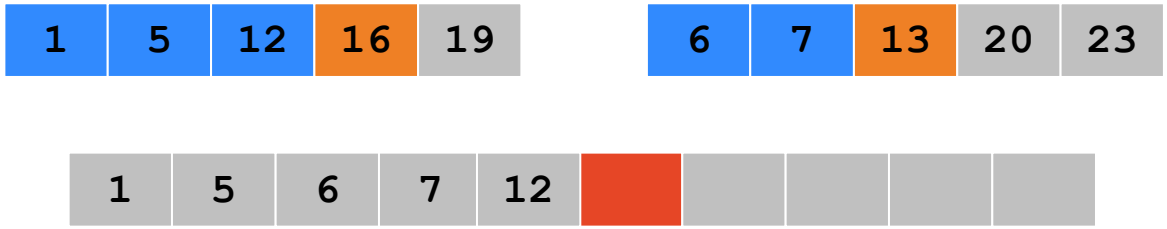
# Merge

**Input** Two sorted lists.

**Output** A new merged sorted list.

To merge, we use:

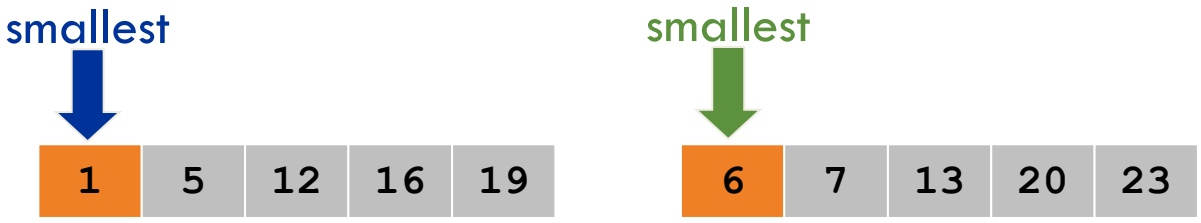
- O(n) comparisons.
- An array to store our results.



# Merge

## Merge Algorithm

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

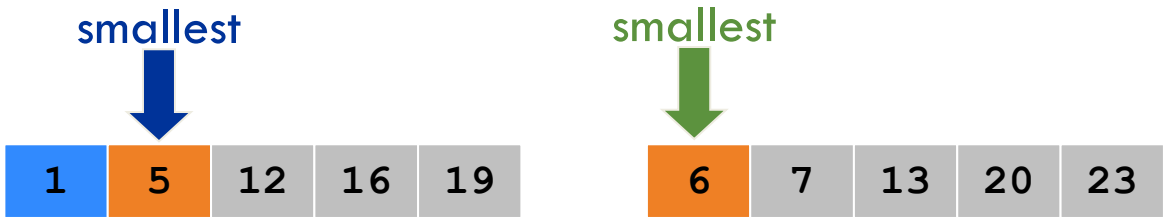



Result:

# Merge

## Merge Algorithm

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

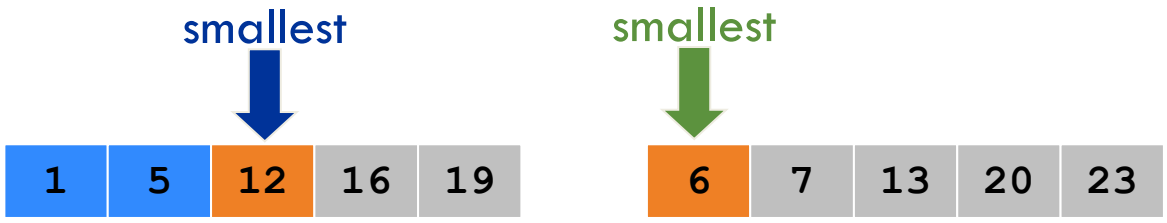



Result:

# Merge

## Merge Algorithm

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

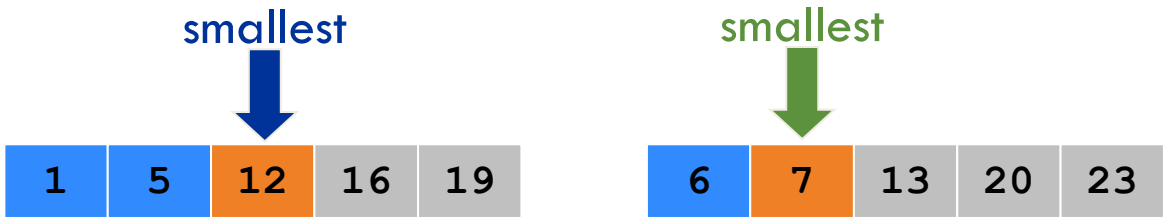



Result:

# Merge

## Merge Algorithm

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

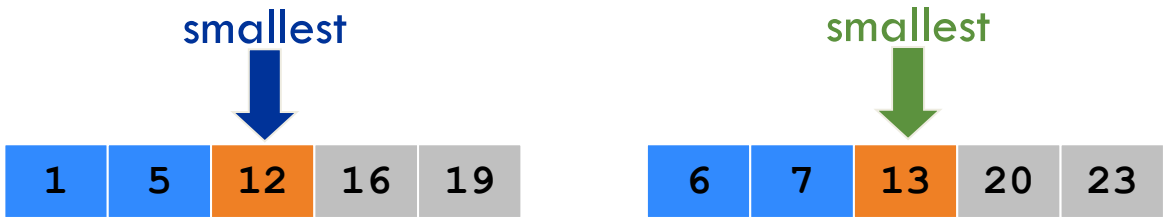



Result:

# Merge

## Merge Algorithm

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

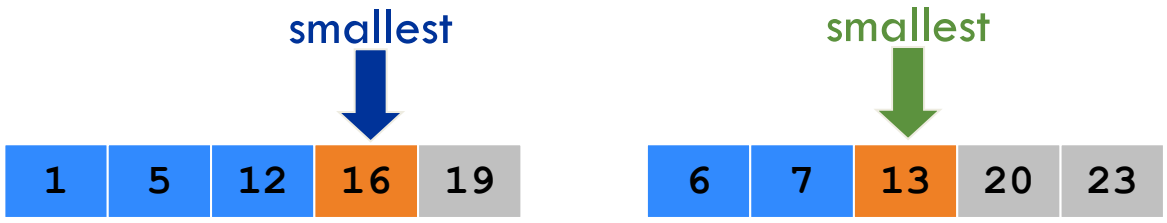



Result:

# Merge

## Merge Algorithm

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

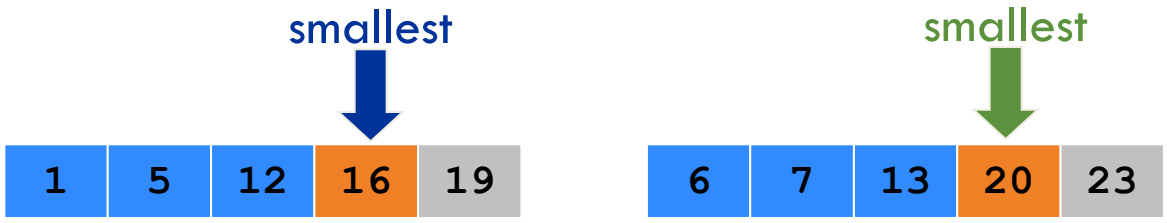



Result:

# Merge

## Merge Algorithm

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




Result:

# Merge

## Merge Algorithm

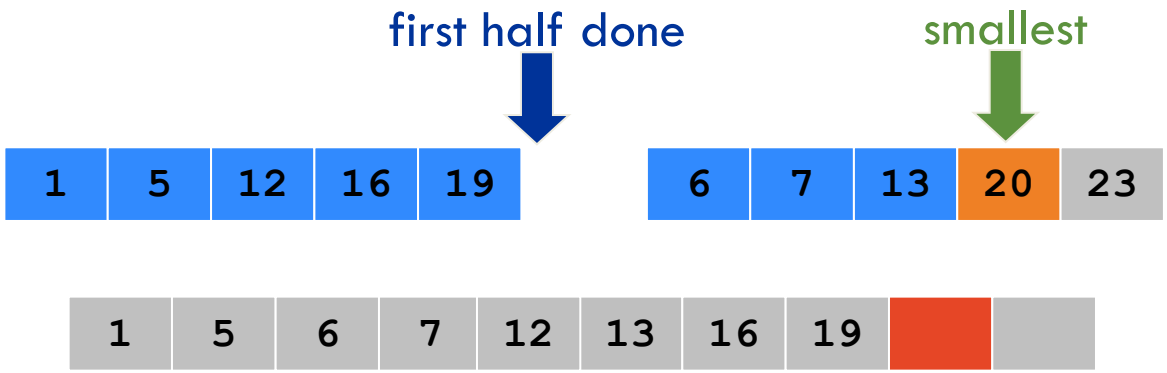
- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into the resultant array.
- Repeat until done.


Result:

# Merge

## Merge Algorithm

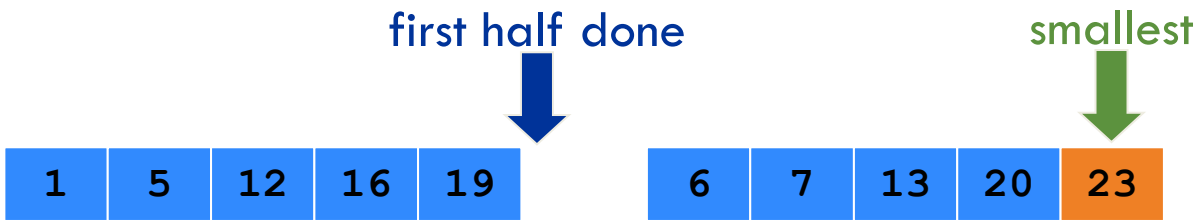
- Keep track of smallest element in each sorted half.
- Insert smallest of two elements into the resultant array.
- Repeat until done.



# Merge

## Merge Algorithm

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

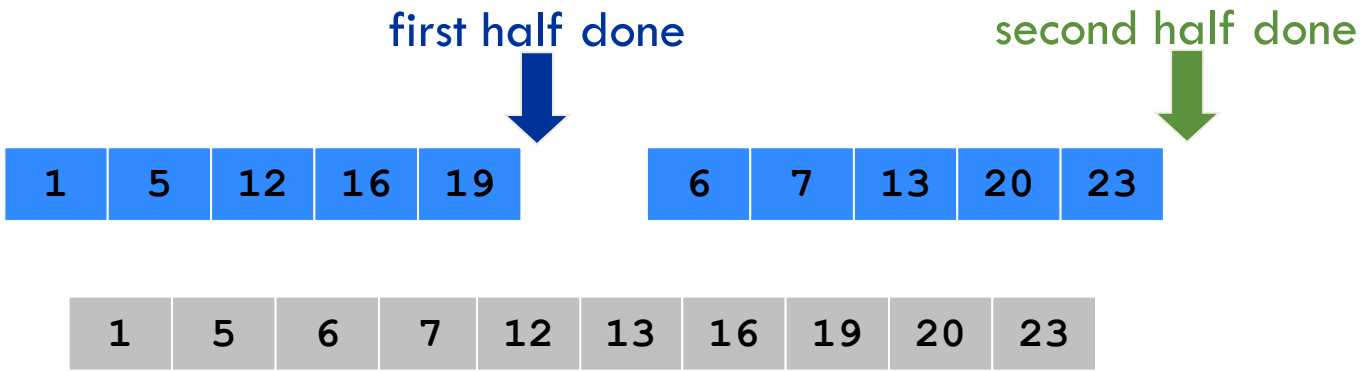



Result:

# Merge

## Merge Algorithm

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

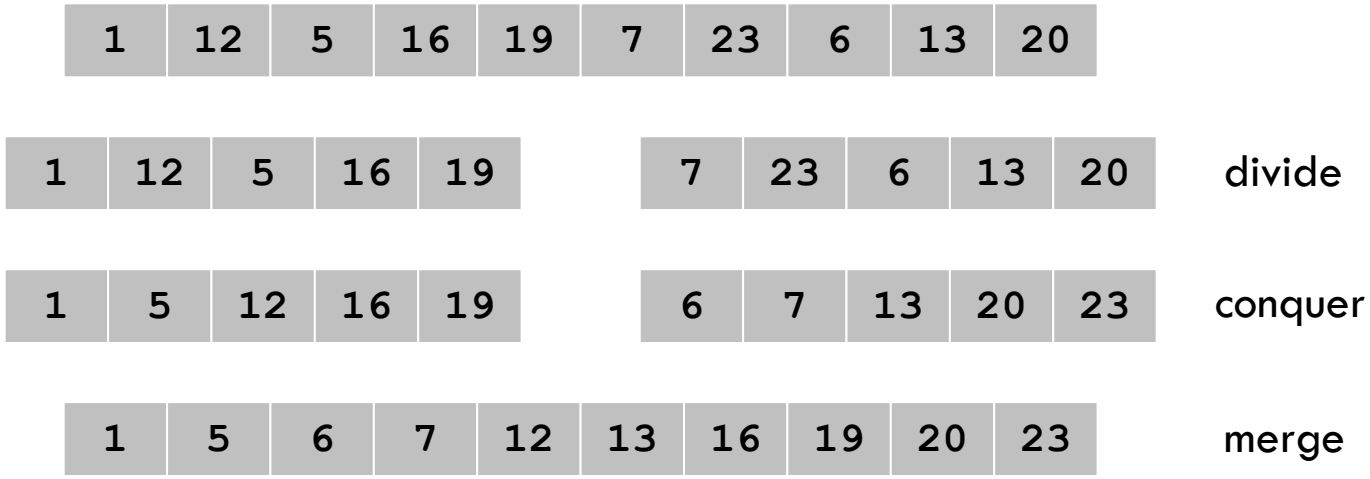


# Merge: Implementation

```
def merge(L, R):
   result ← array of length (|L| + |R|)
   l, r ← 0, 0
   while l + r < |result| do
      index = l + r
      if r ≥ len(R) or (l < len(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. Conquer Recursively sort each half.
3. Merge Merge two sorted halves to make a sorted whole.



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

# Solving recurrences by unrolling

General strategy:

- Analyze first few levels
- Identify the pattern for a generic level
- Sum up over all levels

To verify the solution, we can substitute guess into the recurrence and prove it formally using induction

For Merge sort this method yields $T(n) = O(n \log n)$

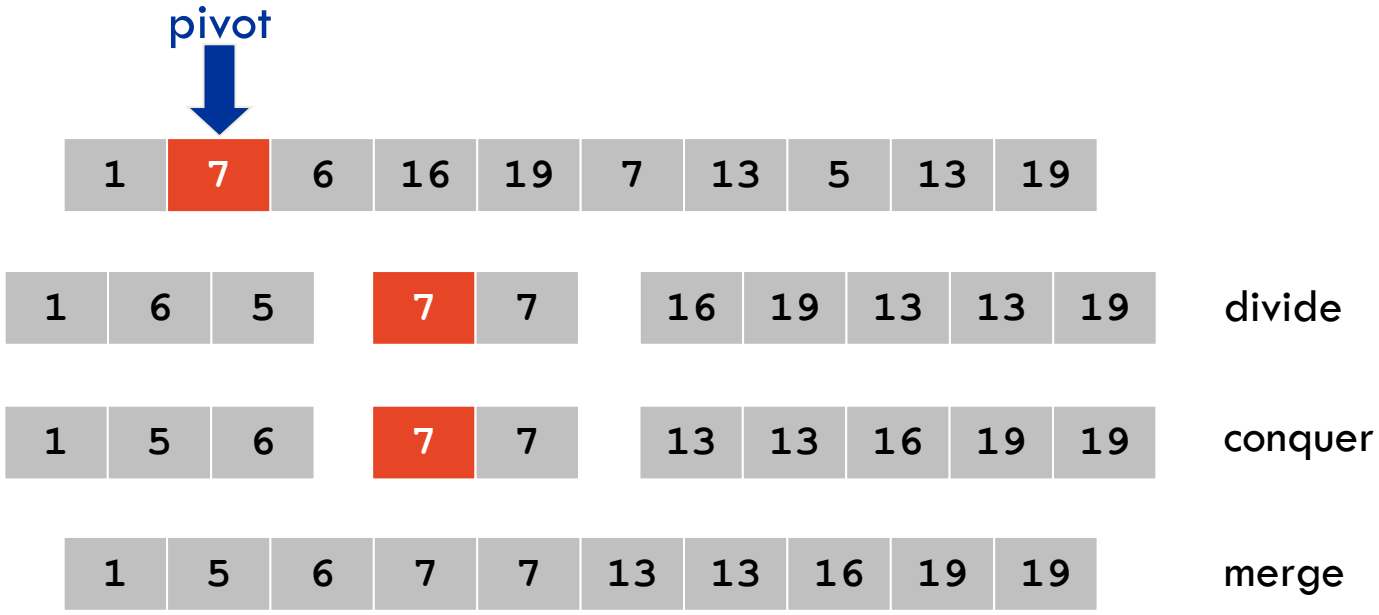
There is a "Master theorem" (see textbook) that can handle most recurrences of interest, but unrolling is enough for our purposes

# 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



# 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 $E[T(n)] = O(n \log n)$

(details available on the textbook but not examinable)

# Interlude: Comparison sorting lower bound

So far we've seen many sorting algorithms. Some runs in $O(n^2)$ time while others run in $O(n \log n)$ time.

These algorithms work by performing pair-wise comparisons between elements of the sequence we are trying to sort

Such algorithms can be viewed as a decision tree where:

- each internal node compares two indices of the input array
- each external node corresponds to a permutation of $\{1, \ldots, n\}$

The height of the decision tree is lower bound on the running time of the algorithm, since it only counts number of comparisons

# Interlude: Comparison sorting lower bound

Fact: Comparison-based sorting algorithms take $\Omega(n \log n)$ time

Proof:

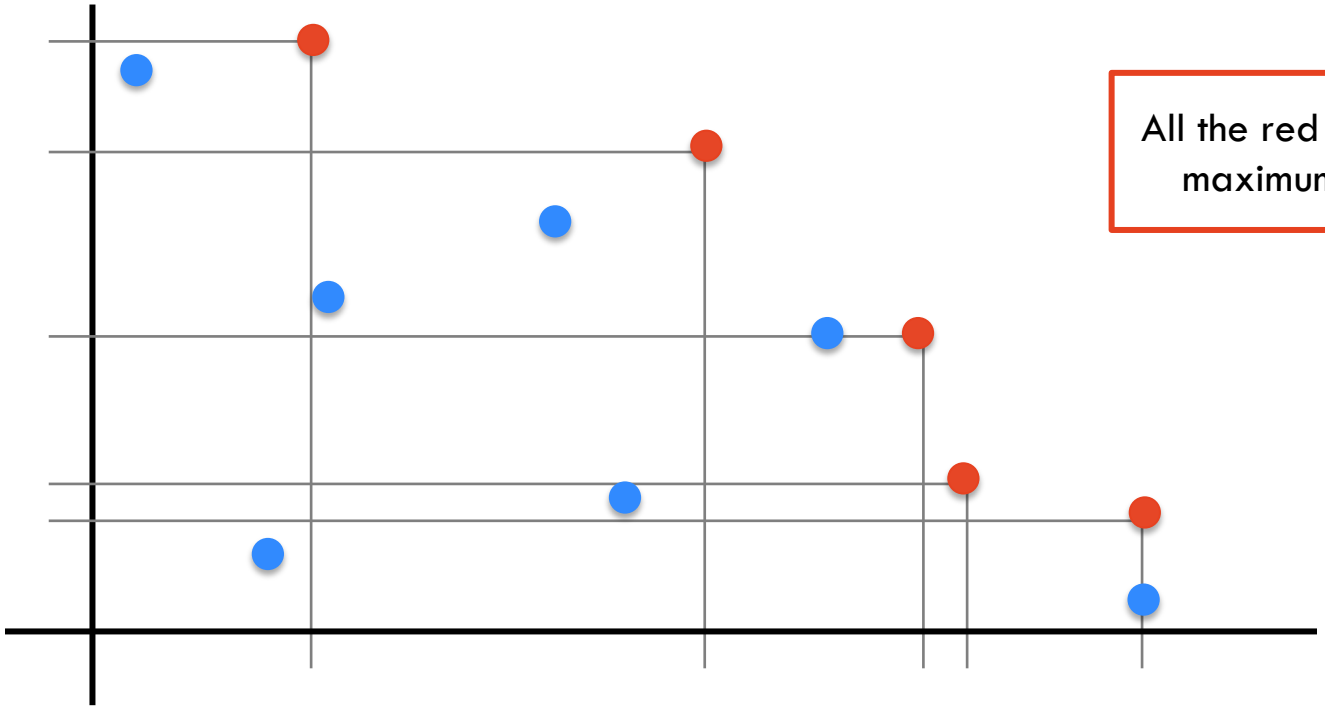
The decision tree associated with a comparison-based sorting algorithm is binary and has $n!$ external nodes. Thus the height is $\log n!$ which is $\Omega(n \log n)$

$$\log n! = \log (n * (n-1) * \ldots * 1)$$
$$= \log n + \log(n-1) + \ldots + \log 1$$
$$> (\log n/2)^{n/2}$$
$$= \Omega(n \log n)$$

# 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:
    if q ≠ p and q.x ≥ p.x and q.y ≥ p.y :
        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:
    if q ≠ p and q.x ≥ p.x and q.y ≥ p.y :
        return "No"
return "Yes"
```

Naïve algorithm to find the maxima-set of S:

```
maximaSet ← empty list
for p in S:
    if p is a maximum point in S:
        add p to the maximaSet
return the 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:
    if q ≠ p and q.x ≥ p.x and q.y ≥ p.y :
        return "No"
return "Yes"
```

O(n)

Naïve algorithm to find the maxima-set of S:

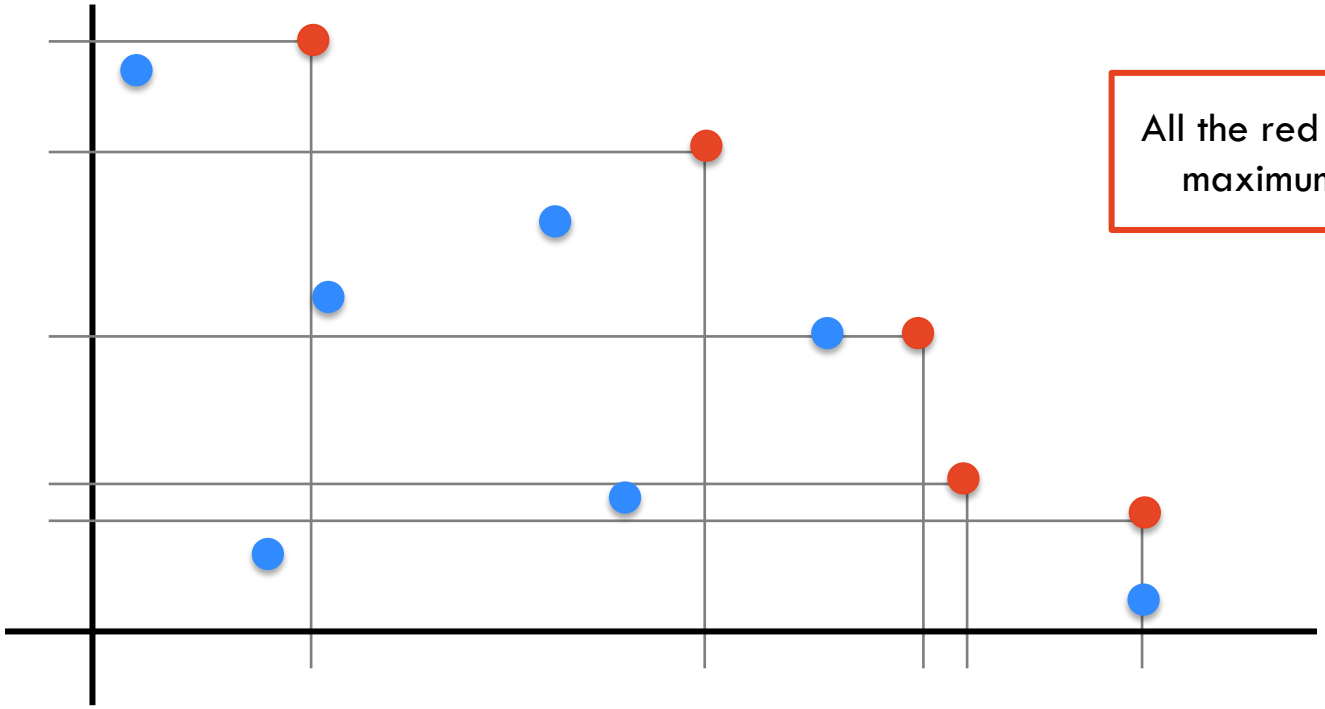
```
maximaSet ← empty list
for p in S:
    if p is a maximum point in S:
        add p to the maximaSet
return the maximaSet
```

O(n) inside the loop; O(n²) overall

# Maxima-Set

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

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.

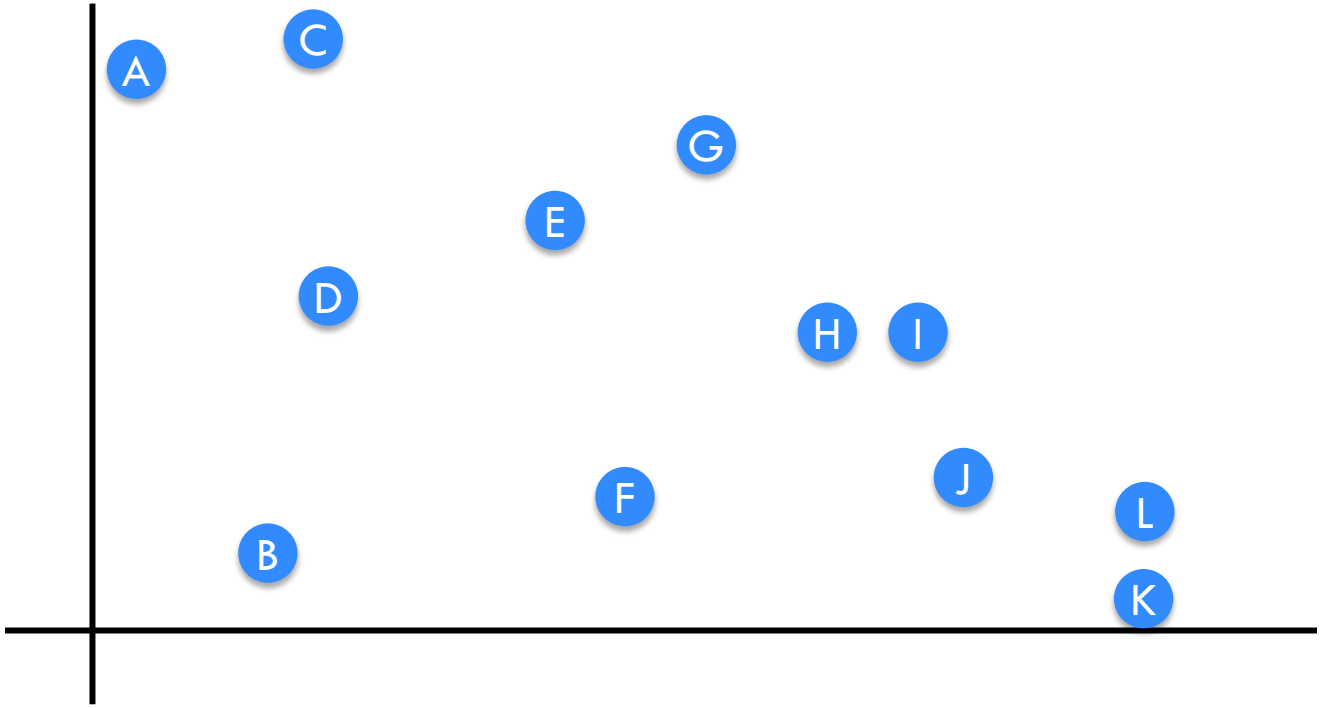
Divide sorted array into two halves.

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.

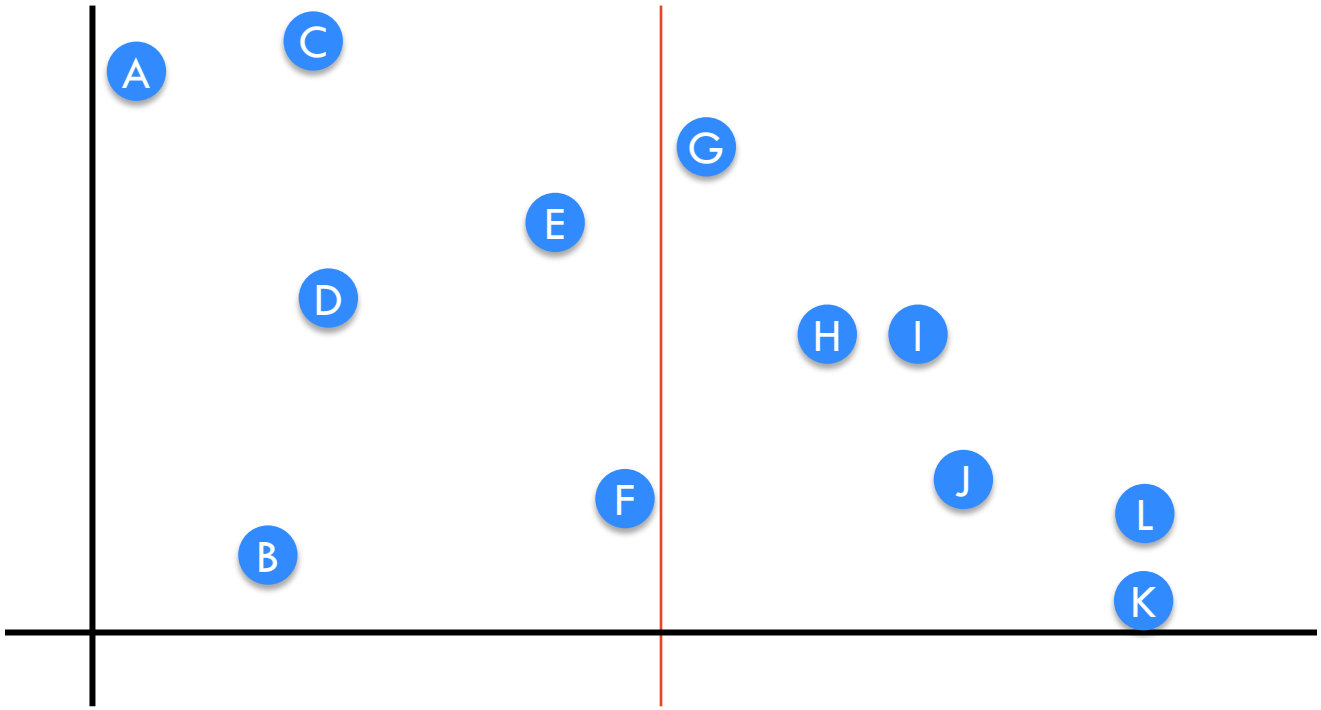



Sorted Points

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

# Maxima-Set

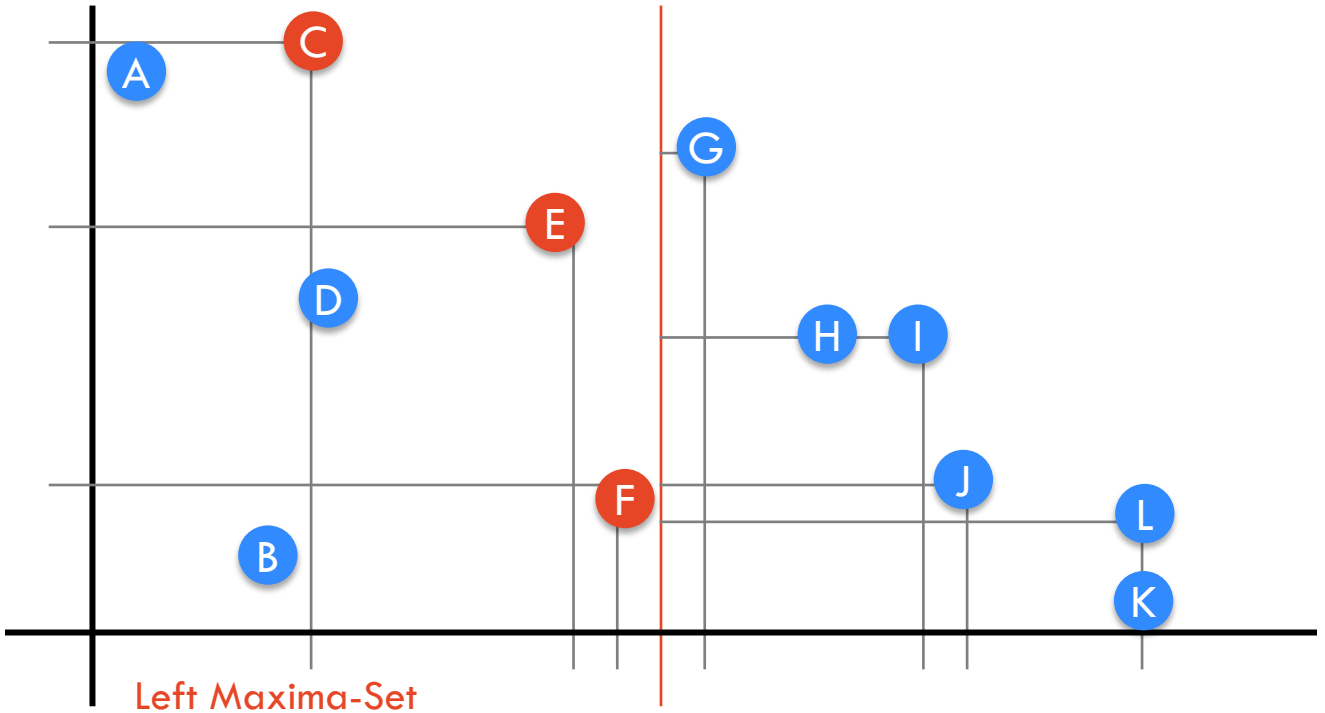
Divide array into two halves.




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

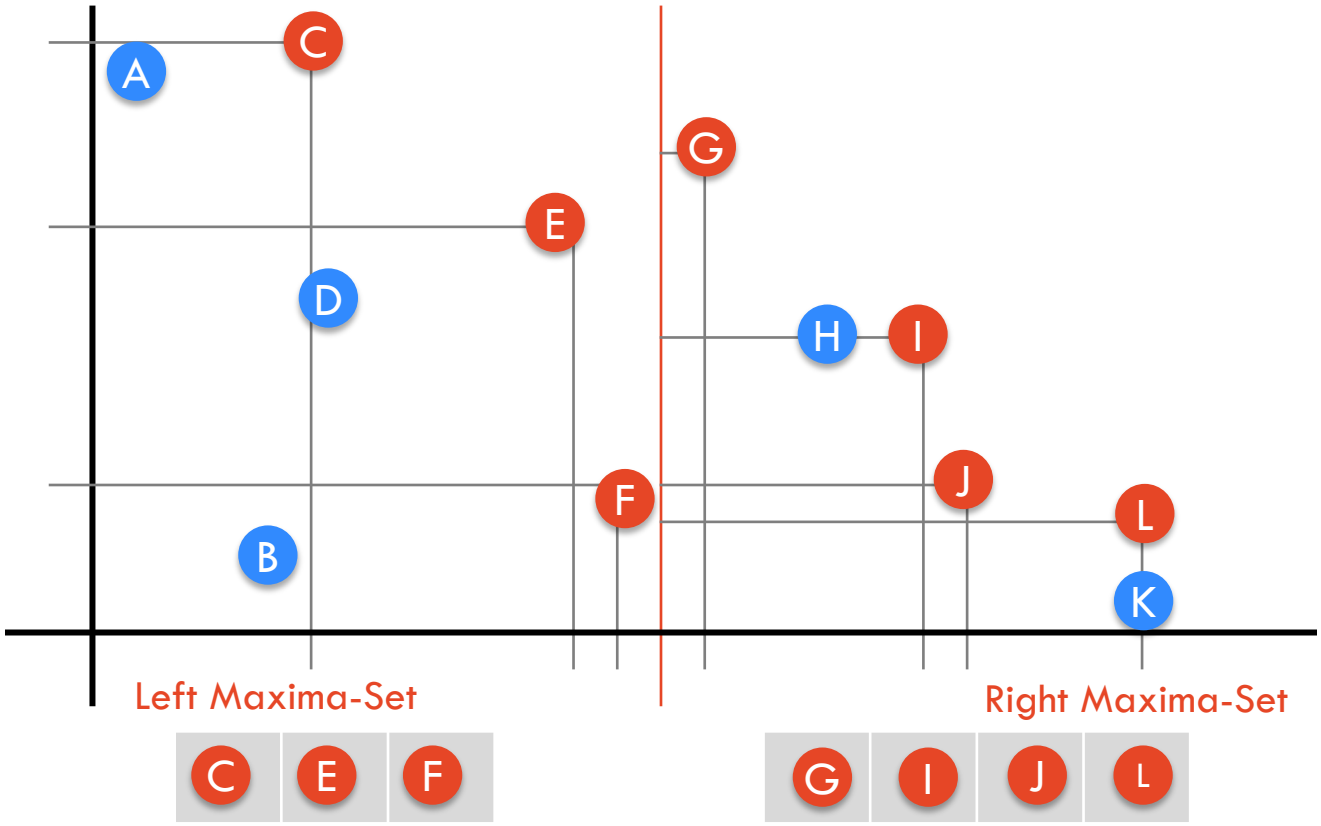
# Maxima-Set

Recur recursively find the Maxima-Set of each half.



# Maxima-Set

Recur recursively find the Maxima-Set of each half.



# Maxima-Set

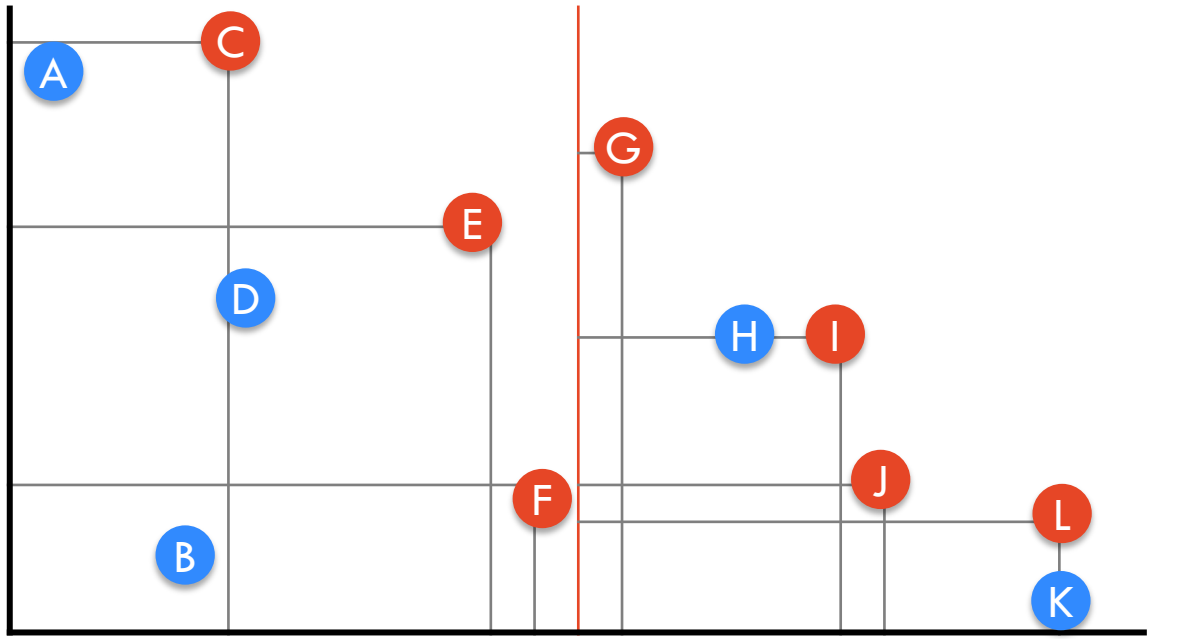
## Conquer

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

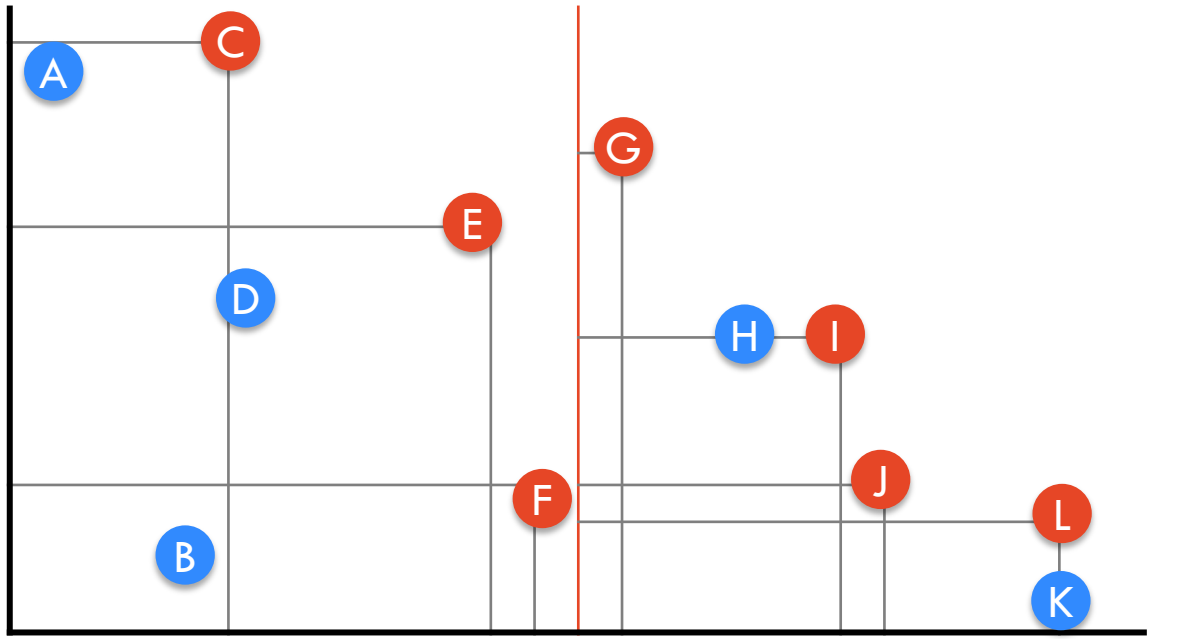


# Maxima-Set

## 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

C G I J L

C G I J L

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

# Maxima-Set: Correctness

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 MS of union of Left/Right MS

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

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