

# **COMMONWEALTH OF AUSTRALIA**

## **Copyright Regulations 1969**

### **WARNING**

This material has been reproduced and communicated to you by or on behalf of the University of Sydney pursuant to Part VB of the Copyright Act 1968 (**the Act**). The material in this communication may be subject to copyright under the Act. Any further copying or communication of this material by you may be the subject of copyright protection under the Act.

**Do not remove this notice.**

**COMP2123**

**Data structures and Algorithms**

**Lecture 11: Divide and Conquer**

[GT 11 and 9]

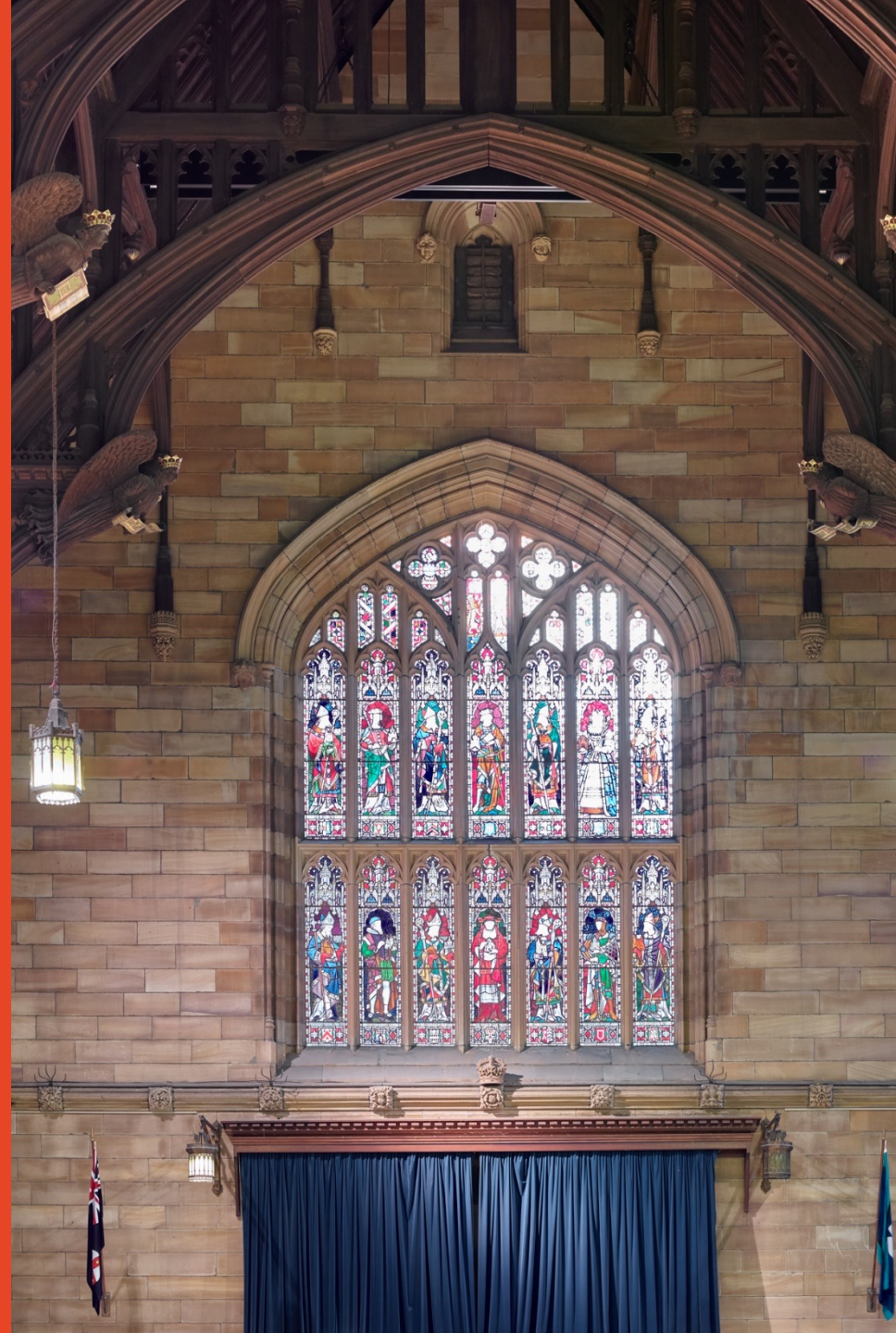
A/Prof Julian Mestre

School of Computer Science

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



THE UNIVERSITY OF  
**SYDNEY**



# Divide and Conquer

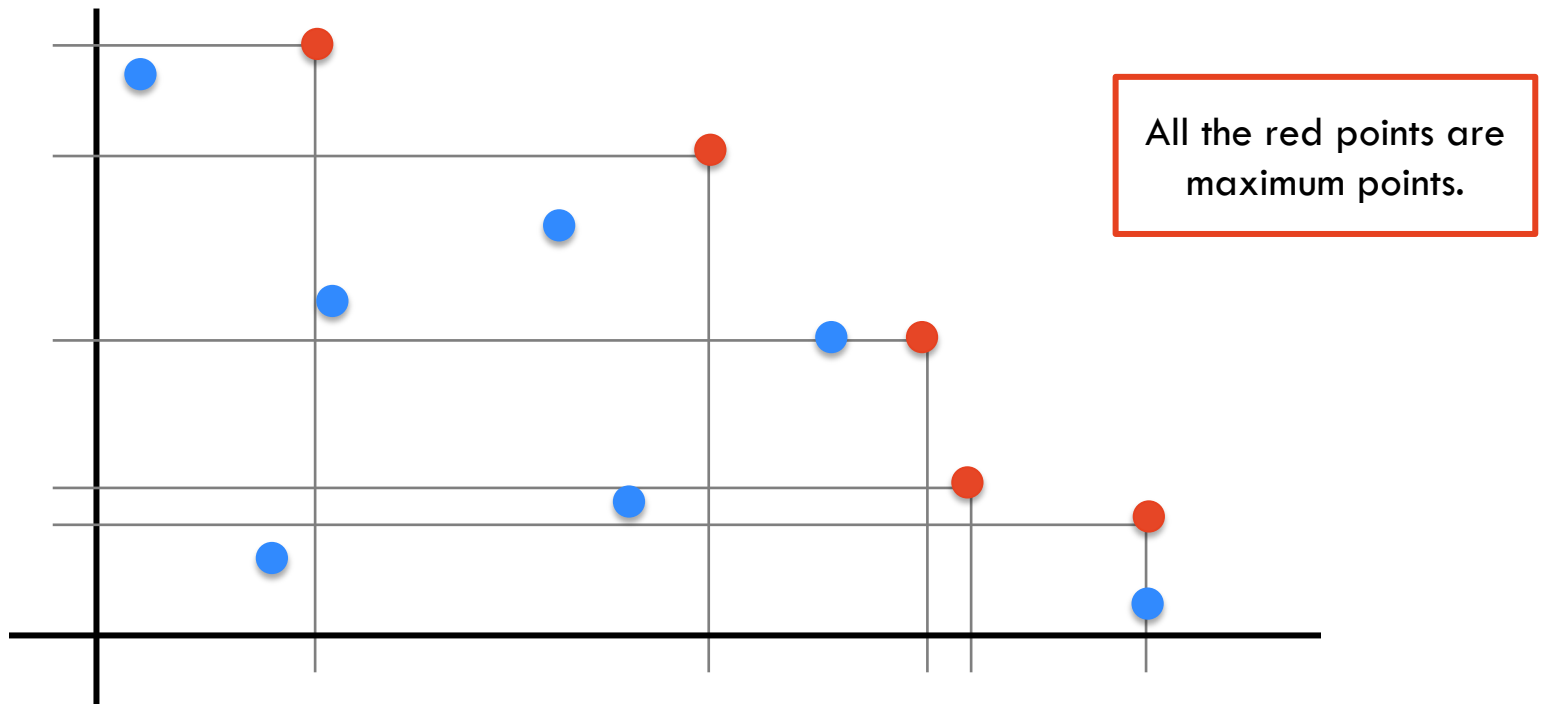
**Divide and Conquer algorithms** can normally be broken into these three parts:

1. **Divide** If it is a base case, solve directly, otherwise break up the problem into several parts.
2. **Recur/Delegate** Recursively solve each part [each sub-problem].
3. **Conquer** Combine the solutions of each part into the overall solution.

# 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.



# 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  $\neq$  p and q.x  $\geq$  p.x and q.y  $\geq$  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  $S$ :

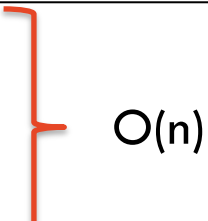
```
maximaSet ← empty list
for p in S do
    if p is a maximum point in S then
        add p to the maximaSet
return maximaSet
```

# 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  $S$ :

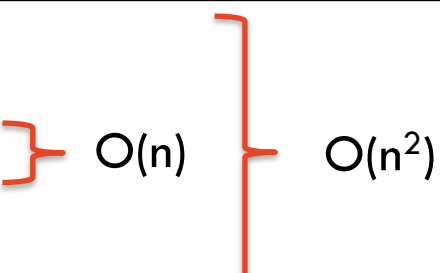
```
for q in S do
    if q ≠ p and q.x ≥ p.x and q.y ≥ p.y then
        return "No"
return "Yes"
```



$O(n)$

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
```

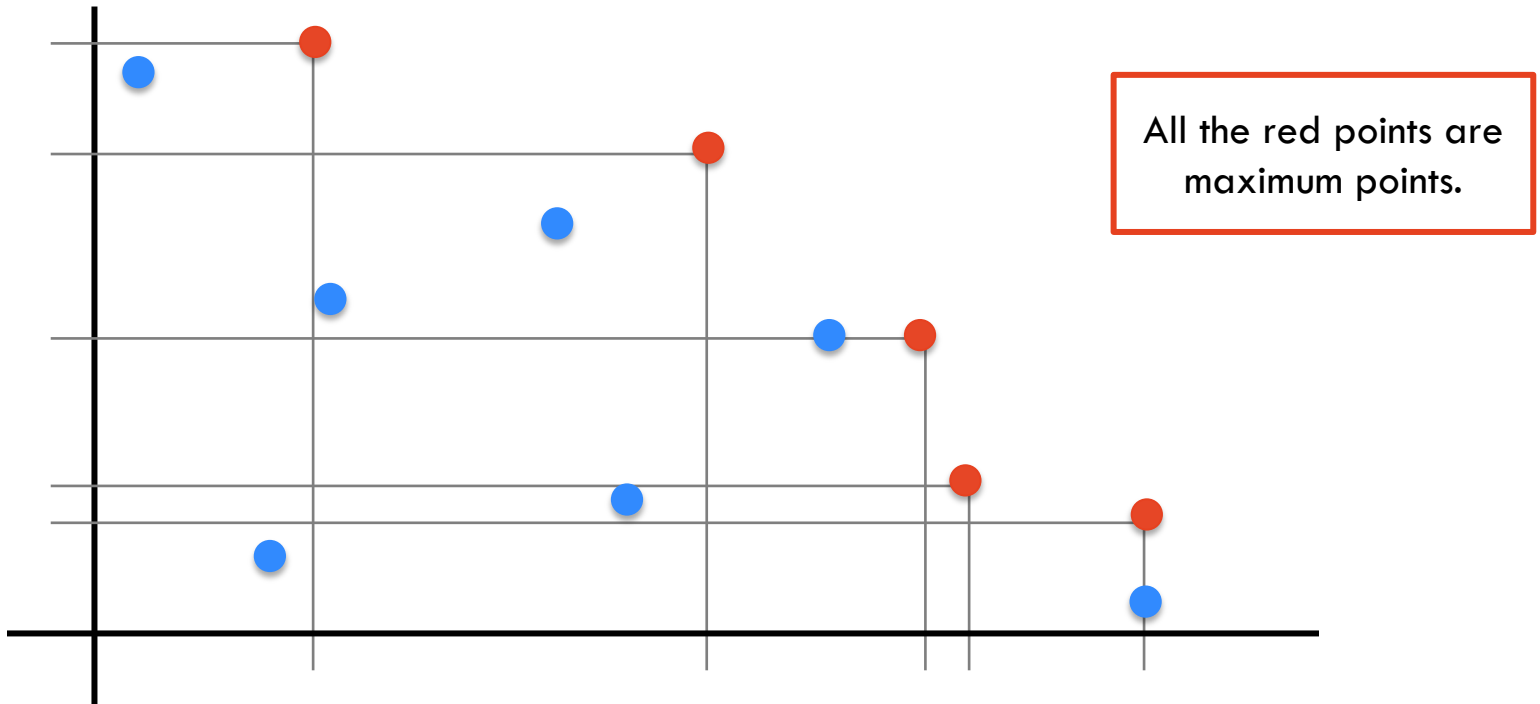


$O(n)$   $O(n^2)$

# Maxima-Set

**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.





# 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.

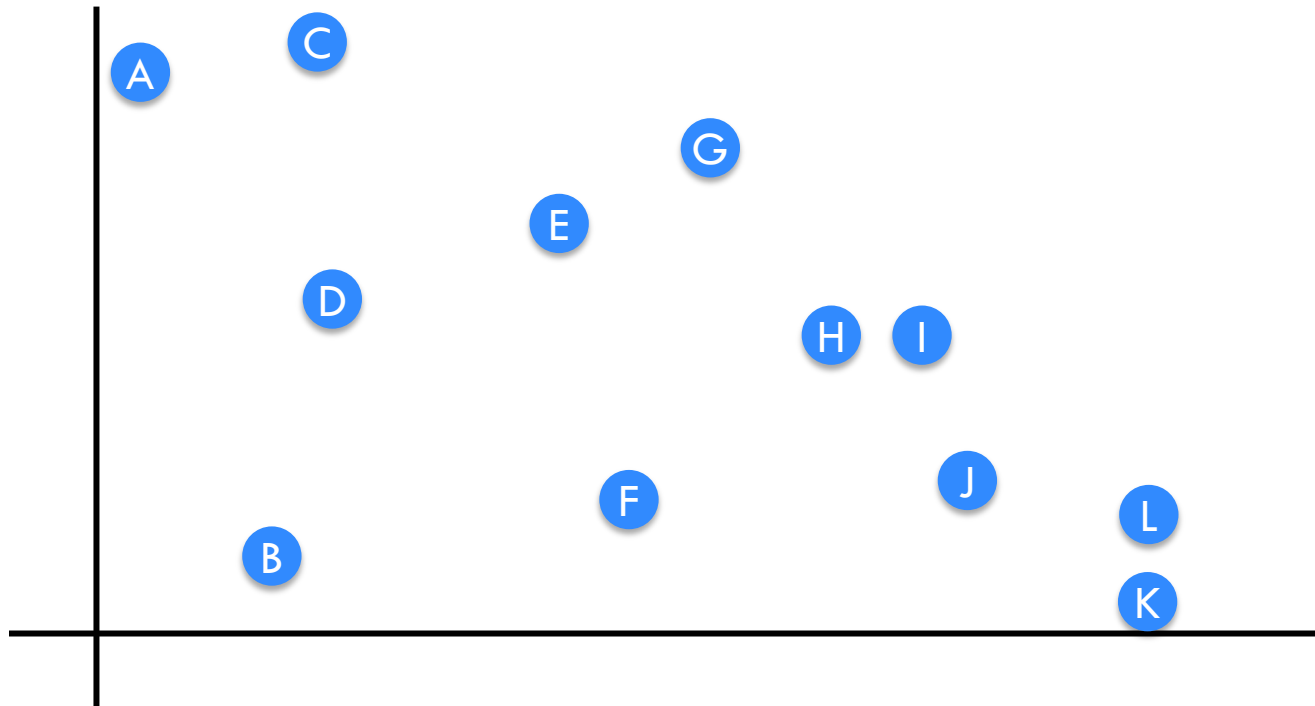
**Divide** sorted array into two halves.

**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.

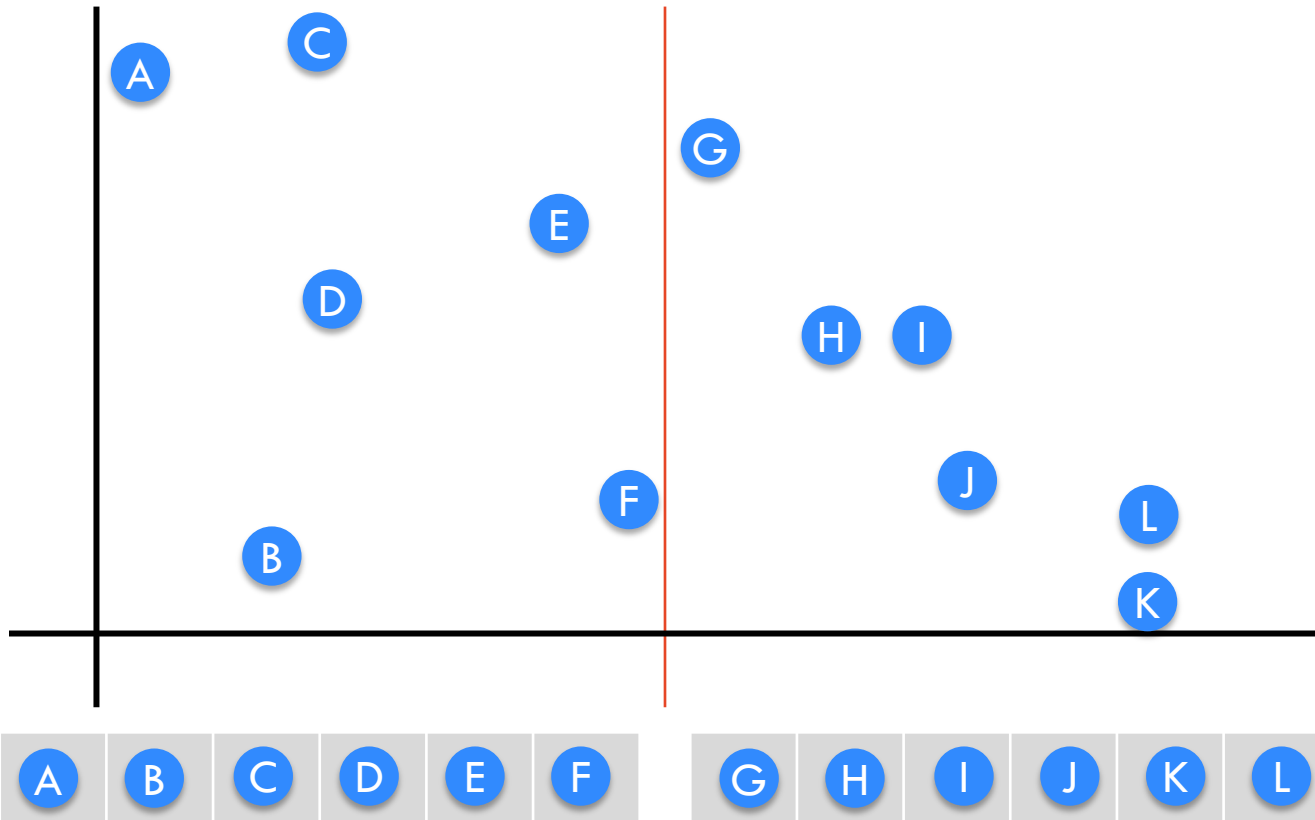


**Sorted Points**



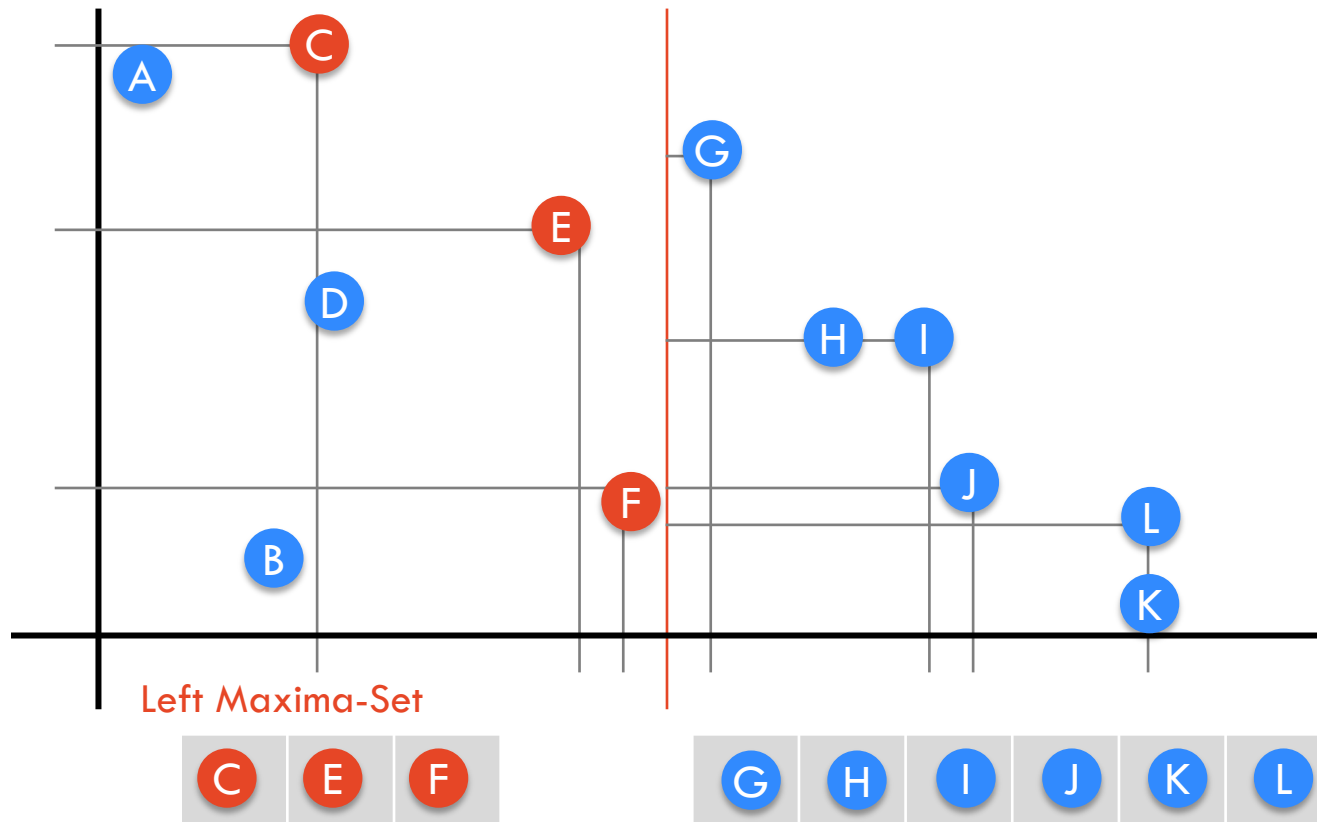
# Maxima-Set

Divide array into two halves.



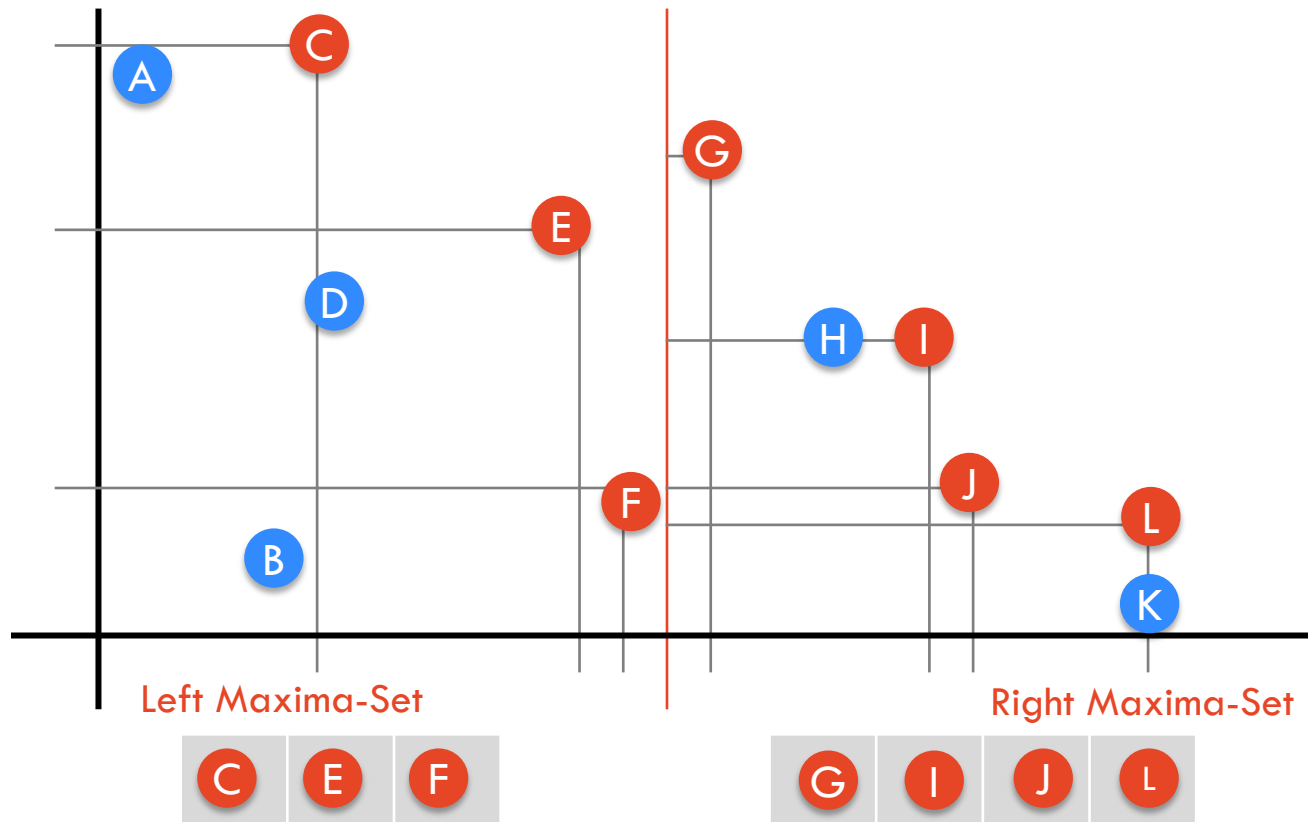
# Maxima-Set

**Recur** recursively find the Maxima-Set of each half.



# Maxima-Set

**Recur** recursively find the Maxima-Set of each half.



# Maxima-Set

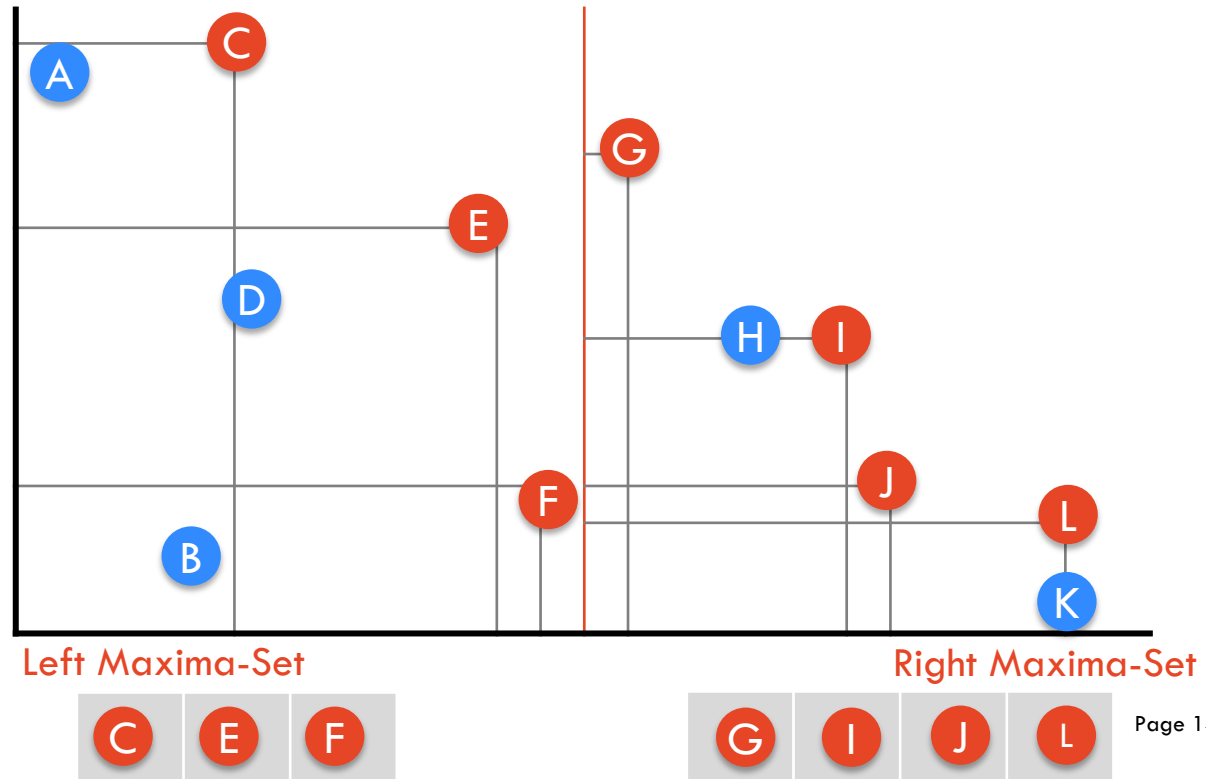
## Conquer

1. Find the highest point  $p$  in the Right MS

$$p = \text{G}$$

### 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  $p$



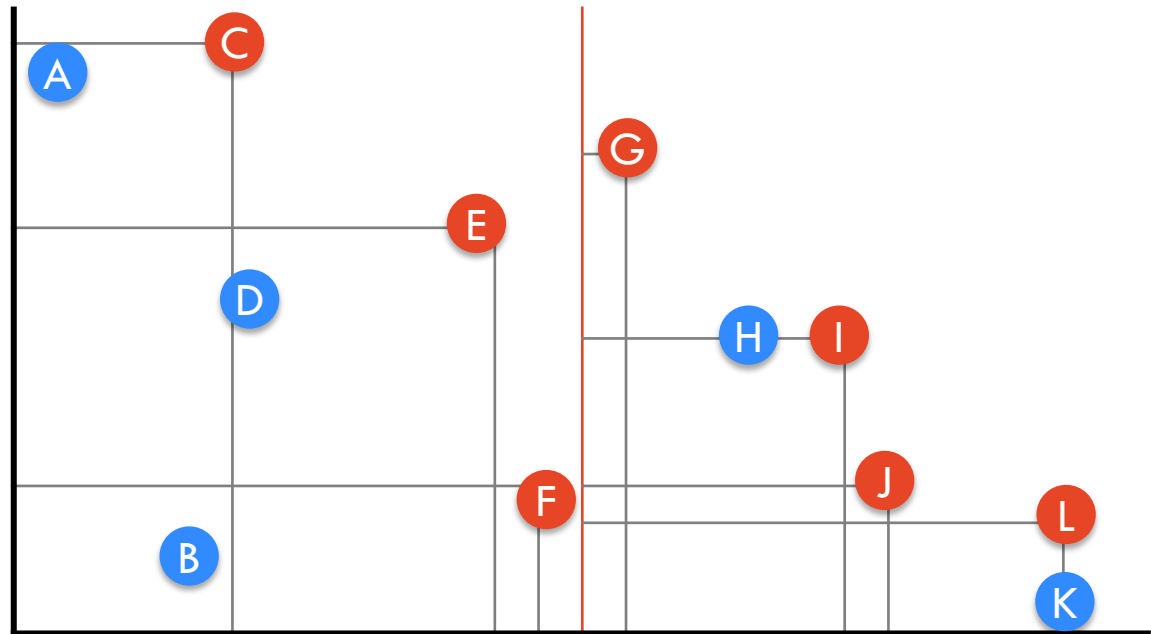
# Maxima-Set

## 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**

$$p = \text{G}$$

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

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**

$O(n)$

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

**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  $p$

# Integer multiplication

**Given** two  $n$ -digit integers  $x$  and  $y$

**Problem** compute the product  $x \cdot y$

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  $x \cdot y$

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

# Integer multiplication: Correctness

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

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

Straight forward application of induction to prove that  $\text{multiply}(x, y) = x y$

# Integer multiplication: Complexity analysis

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

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

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

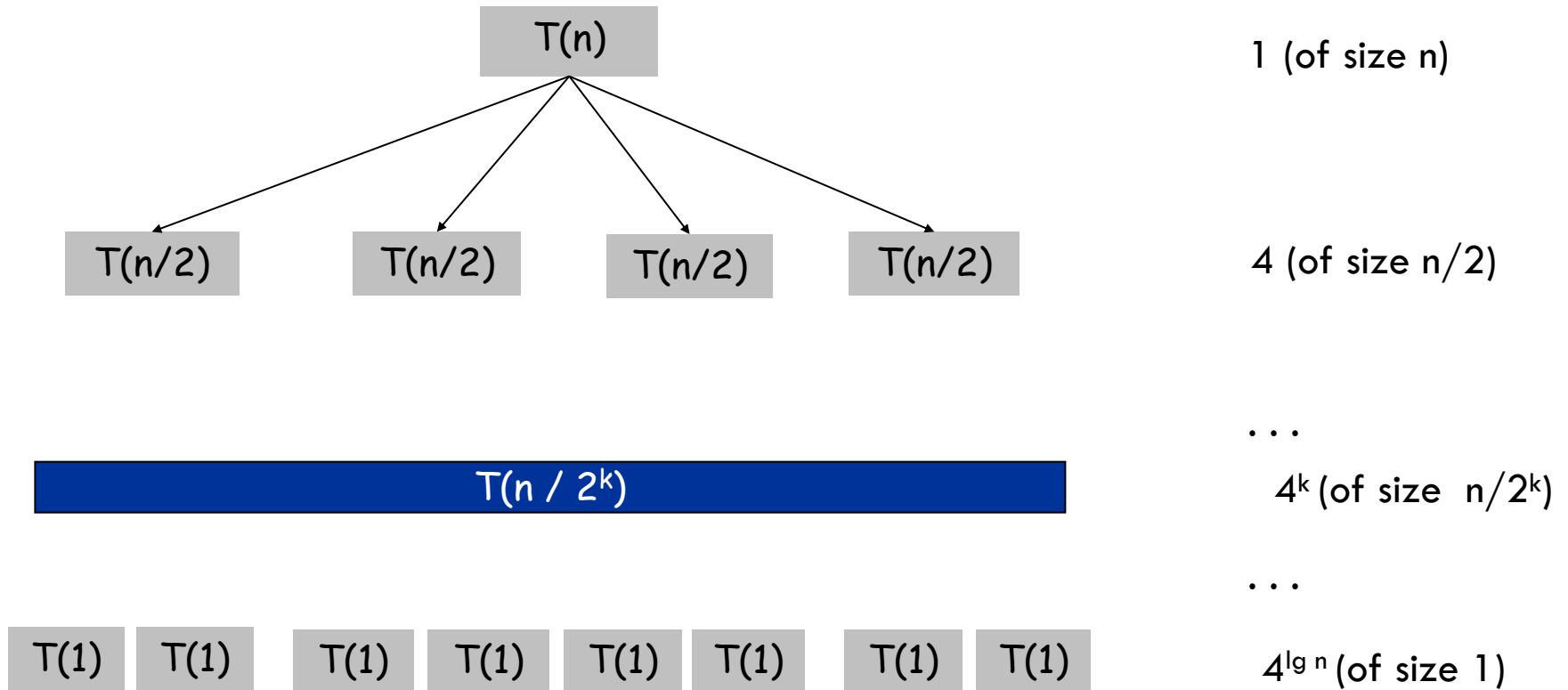
**Conquer step** (add up results) takes  $O(n)$

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

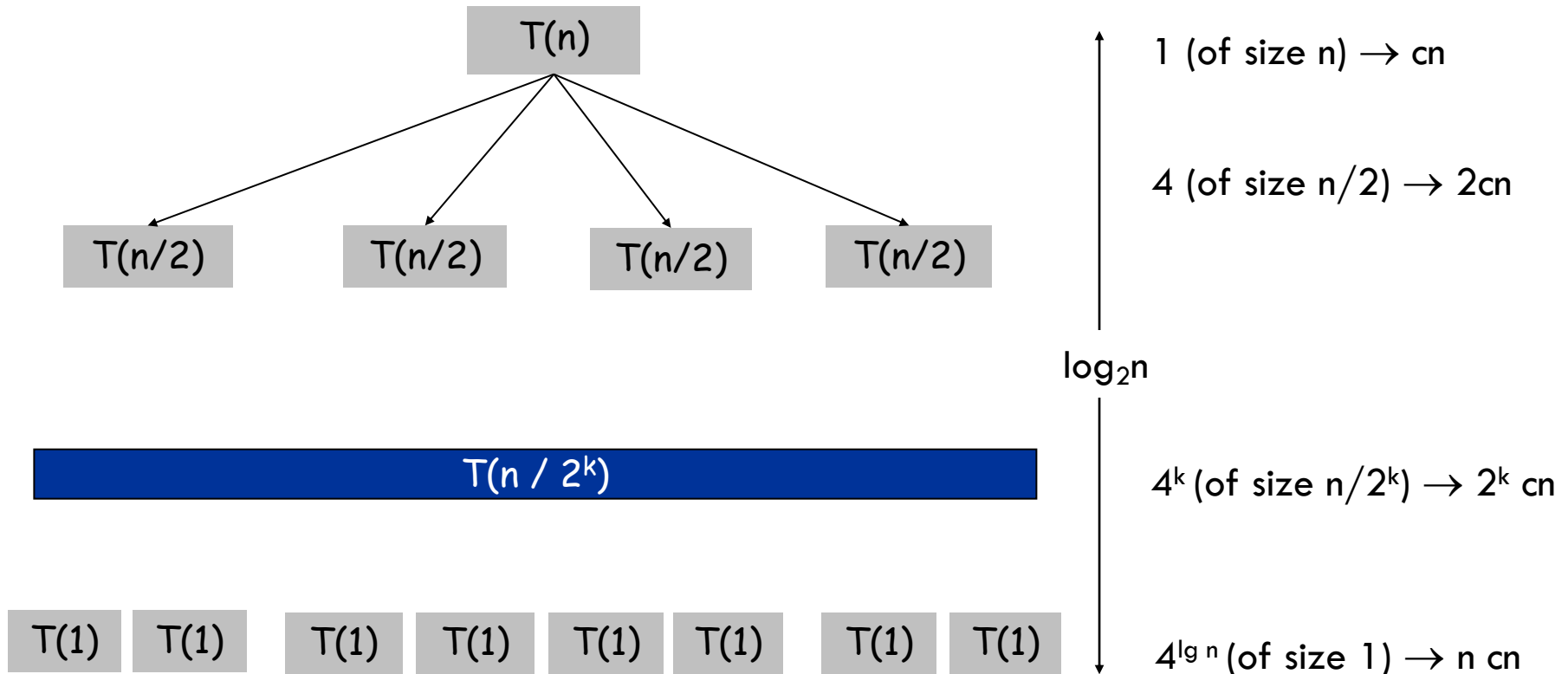
This solves to  $T(n) = O(n^2)$ . No better than naïve!!!



# Proof by unrolling: $T(n) = 4 T(n/2) + O(n)$



# Proof by unrolling: $T(n) = 4 T(n/2) + O(n)$



# 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  $x_1$  and  $x_0$  be such that  $x = x_1 2^{n/2} + x_0$   
    let  $y_1$  and  $y_0$  be such that  $y = y_1 2^{n/2} + y_0$   
  
    first_term  $\leftarrow$  multiply( $x_1$ ,  $y_1$ )  
    last_term  $\leftarrow$  multiply( $x_0$ ,  $y_0$ )  
    other_term  $\leftarrow$  multiply( $x_1 + x_0$ ,  $y_1 + y_0$ )  
  
    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  $3 T(n/2)$

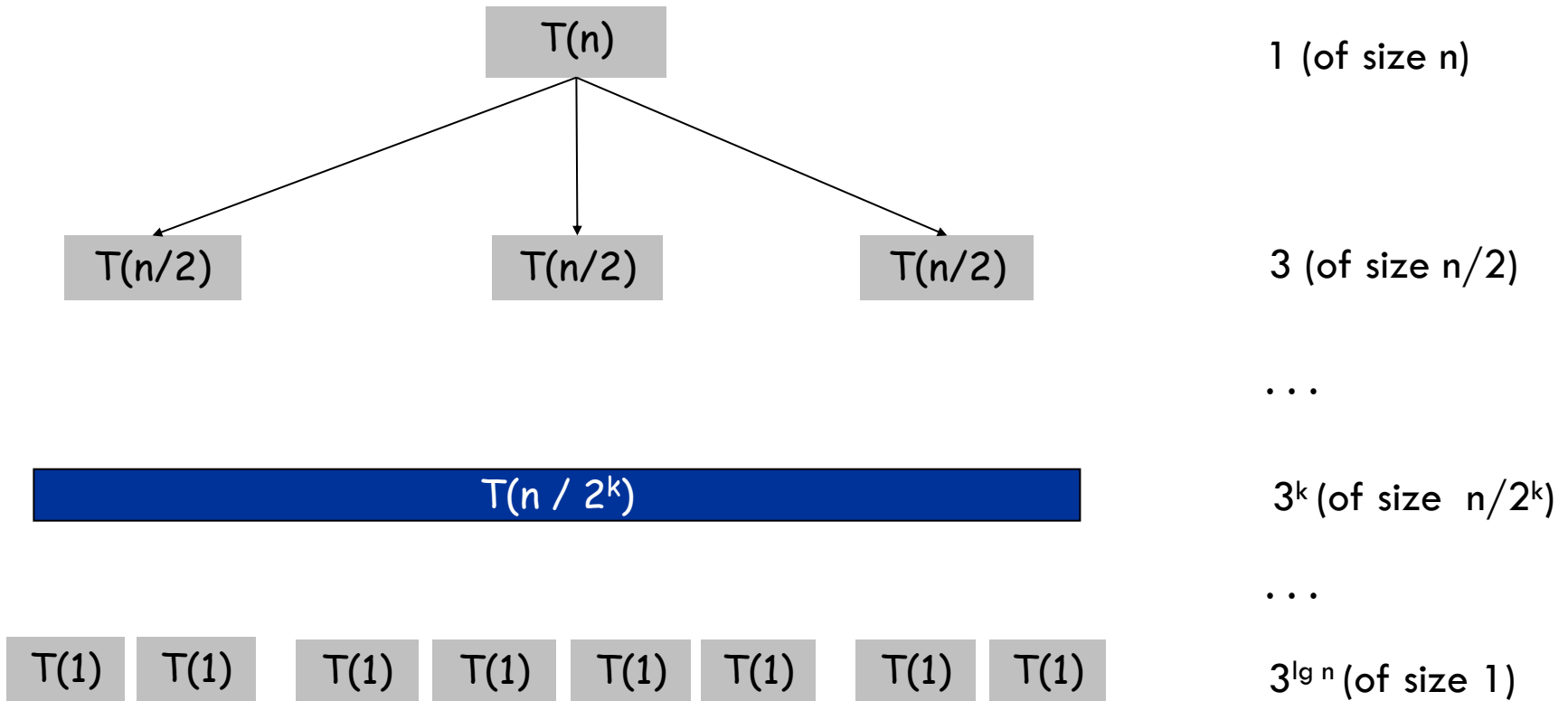
Conquer step (add up results) takes  $O(n)$

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

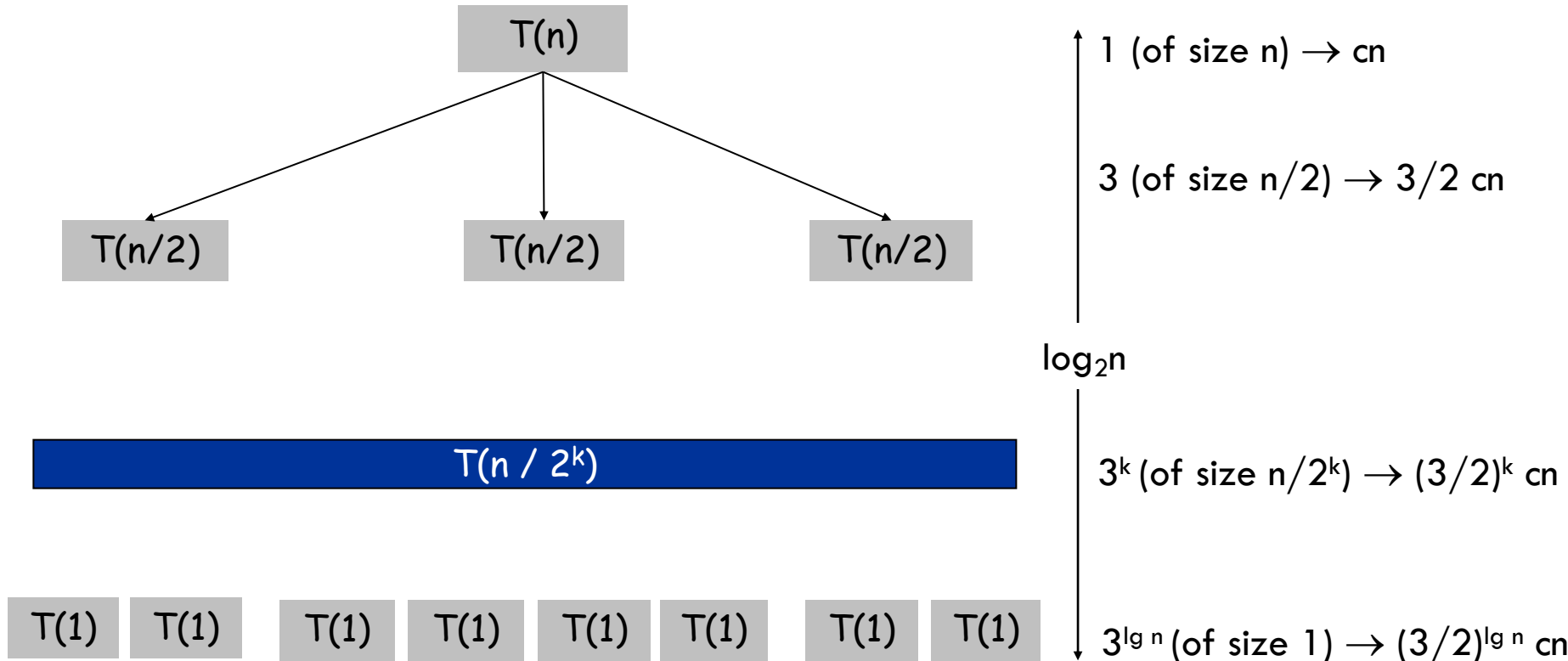
This solves to  $T(n) = O(n^{\log_2 3})$ , where  $\log_2 3 \approx 1.6$

Better than naïve!!!

# Proof by unrolling: $T(n) = 3 T(n/2) + O(n)$



# Proof by unrolling: $T(n) = 3 T(n/2) + O(n)$



## Geometric series facts

Let  $r$  be a positive real and  $k$  a positive integer then

$$1 + r + r^2 + \dots + r^k = (r^{k+1} - 1)/(r-1)$$

Consequently if  $r > 1$  then

$$1 + r + r^2 + \dots + r^k < r^{k+1} / (r-1)$$

and if  $r < 1$  then

$$1 + r + r^2 + \dots + r^k < 1 / (1-r)$$



# 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  $\Theta$ ).

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  $f(n) = O(n^{\log_b a - \epsilon})$  (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  $f(n) = \Omega(n^{\log_b a + \epsilon})$  (case 3)

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

# Selection

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

Trivial solution: Sort the elements and return  $k$ th 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  $k$ th 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

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

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

$$\# \text{ elements smaller than } x > 2(|A| / 6) = |A| / 3$$

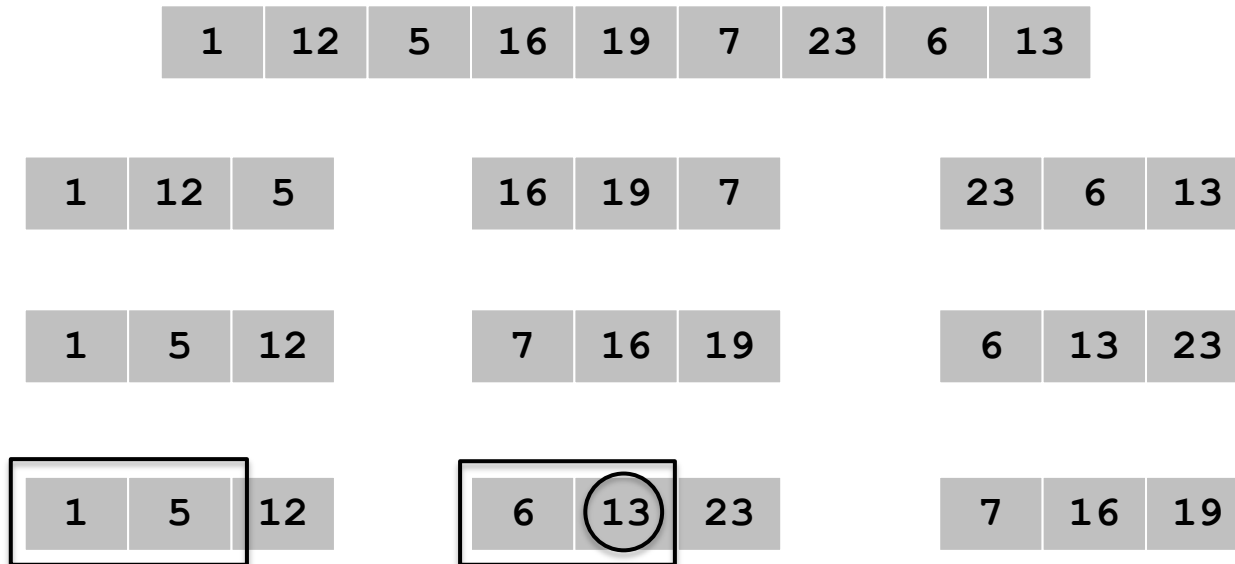
$$\# \text{ elements greater than } x > 2(|A| / 6) = |A| / 3$$



## Median of 3 medians

Let  $x$  be the median of the medians, then

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



# 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

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

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

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

Then we get the recurrence

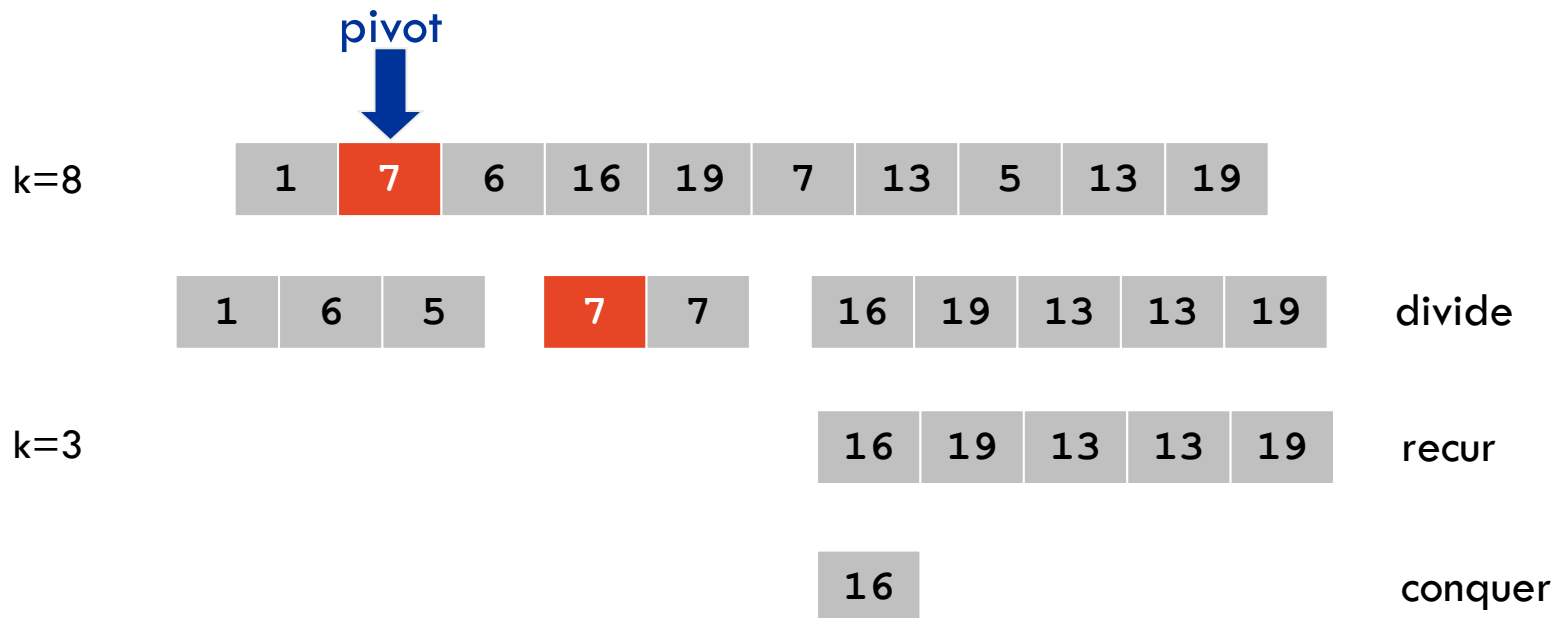
$$T(n) = T(7n/10) + T(n/5) + O(n)$$

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)