BLM5106- Advanced Algorithm Analysis and Design

H. İrem Türkmen

Introduction to the Design and Analysis of Algorithms, **Anany Levitin**
http://ocw.mit.edu, **Design and Analysis of Algorithms**
http://web.stanford.edu/class/archive/cs/cs161/cs161.1176/, **Design and Analysis of Algorithms**

Brute force

- ***Brute force*** is a straightforward approach to solving a problem, usually directly based on the problem statement and definitions of the concepts involved.
- Exponentiation problem: compute *an* for a nonzero number *a* and a nonnegative integer *n.*

$$a^n = \underbrace{a * \cdots * a}_{n \text{ times}}$$

Brute force

- For some important problems—e.g., sorting, searching, matrix multiplication, string matching— the brute-force approach yields reasonable algorithms of at least some practical value with no limitation on instance size.
- The expense of designing a more efficient algorithm may be unjustifiable if only a few instances of a problem need to be solved and a brute-force algorithm can solve those instances with acceptable speed.
- Even if too inefficient in general, a brute-force algorithm can still be useful for solving small-size instances of a problem.

Brute force

- Basic matrix multiplication   n^3
- Bubble Sort n^2
- Sequential Search

ALGORITHM *SequentialSearch2*$(A[0..n], K)$

//Implements sequential search with a search key as a sentinel
//Input: An array A of n elements and a search key K
//Output: The index of the first element in $A[0..n-1]$ whose value is
//   equal to K or -1 if no such element is found
$A[n] \leftarrow K$
$i \leftarrow 0$
while $A[i] \neq K$ **do**
    $i \leftarrow i + 1$
if $i < n$ **return** i
else return -1

Selection Sort

- Selection Sort

ALGORITHM *SelectionSort*$(A[0..n-1])$

//Sorts a given array by selection sort
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Array $A[0..n-1]$ sorted in nondecreasing order
for $i \leftarrow 0$ **to** $n-2$ **do**
    $min \leftarrow i$
    for $j \leftarrow i+1$ **to** $n-1$ **do**
        if $A[j] < A[min]$ $min \leftarrow j$
    swap $A[i]$ and $A[min]$

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} (n-1-i) = \frac{(n-1)n}{2}.$$

String Matching

- Brute Force String Matching

ALGORITHM *BruteForceStringMatch*($T[0..n-1]$, $P[0..m-1]$)

```
//Implements brute-force string matching
//Input: An array T[0..n − 1] of n characters representing a text and
//         an array P[0..m − 1] of m characters representing a pattern
//Output: The index of the first character in the text that starts a
//         matching substring or −1 if the search is unsuccessful
for i ← 0 to n − m do
    j ← 0
    while j < m and P[j] = T[i + j] do
        j ← j + 1
    if j = m return i
return −1
```

Are they Brute force ?

- The algorithm computes a sum

$$\sum_{i=1}^{n} i^2$$

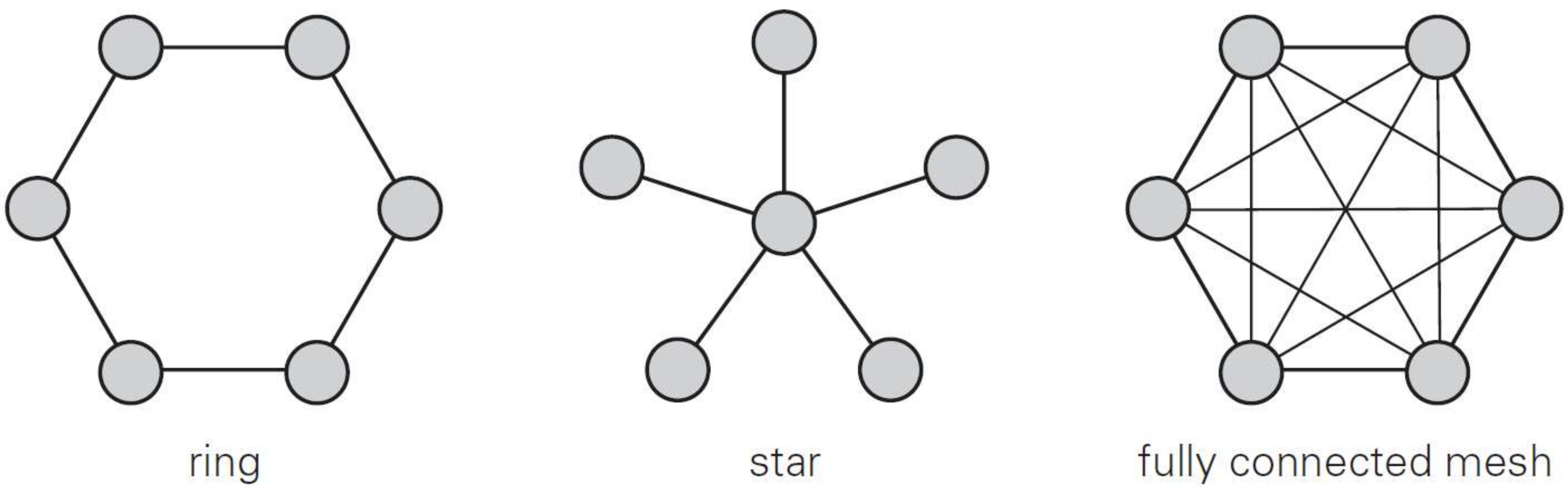
- The algorithm computes a range of the given array

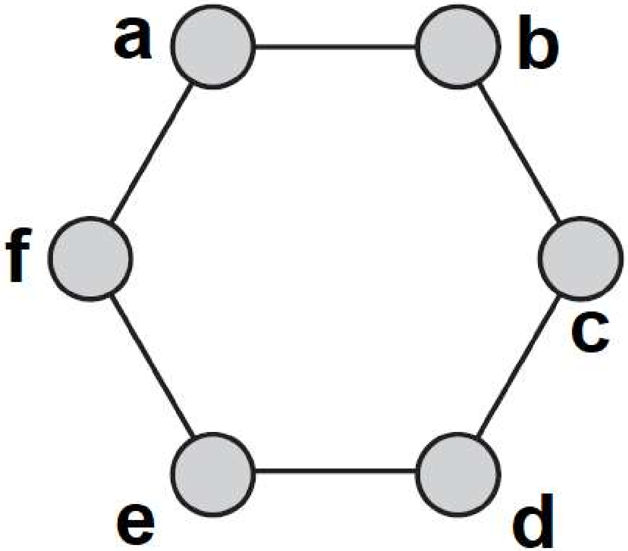
$$range \leftarrow maxval - minval$$

- The algorithm checks whether a given matrix is symmetric

Design Brute force solutions

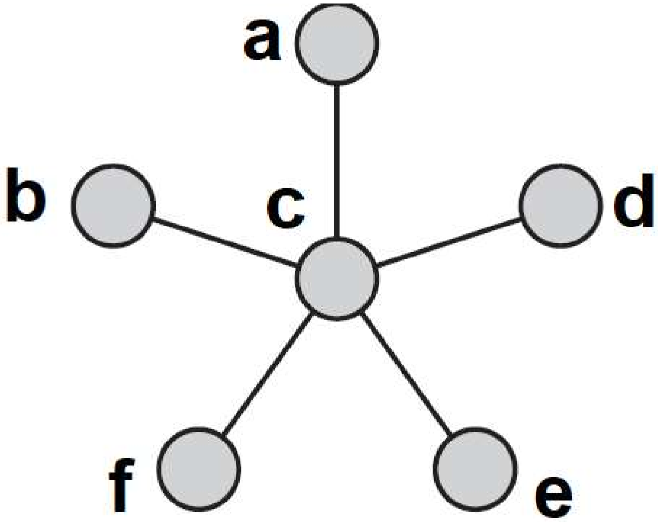
- For an adjacency matrix of a graph modeling, design a brute-force algorithm to determine which of these three topologies is given.





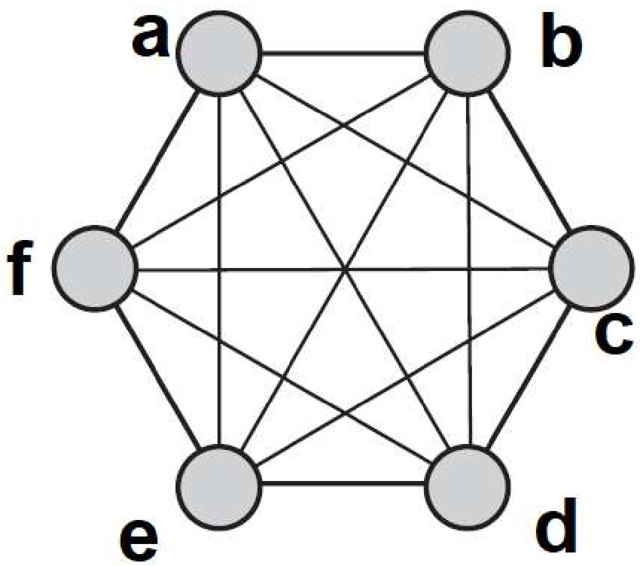
ring

$$\begin{array}{cccccc} a & b & c & d & e & f \end{array}$$
$$\begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$



star

$$\begin{array}{c} \begin{array}{cccccc} a & b & c & d & e & f \end{array} \\ \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix} \end{array}$$



fully connected mesh

$$\begin{array}{c} \begin{array}{cccccc} a & b & c & d & e & f \end{array} \\ \begin{bmatrix} 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 \end{bmatrix} \end{array}$$

Exhaustive search

- ***Exhaustive search*** is simply a brute-force approach to combinatorial problems.
- It suggests generating each and every element of the problem domain, selecting those of them that satisfy all the constraints, and then finding a desired element.

- **Knapsack Problem**
- **Assignment Problem**
- **Traveling Salesman Problem**

Assignment Problem

- There are *n* people who need to be assigned to execute *n* jobs, one person per job.
- The cost that would accrue if the i_{th} person is assigned to the j_{th} job is a known quantity $C[i, j]$ for each pair. The problem is to find an assignment with the minimum total cost.

	Job 1	**Job 2**	**Job 3**	**Job 4**
Person 1	9	2	7	8
Person 2	6	4	3	7
Person 3	5	8	1	8
Person 4	7	6	9	4

Assignment Problem

- Describe n-tuples $<j_1, \ldots, j_n>$
- For the cost matrix above, 2, 3, 4, 1 indicates the assignment of Person 1 to Job 2, Person 2 to Job 3, Person 3 to Job 4, and Person 4 to Job 1.

	Job 1	Job 2	Job 3	Job 4
Person 1	9	2	7	8
Person 2	6	4	3	7
Person 3	5	8	1	8
Person 4	7	6	9	4

Exhaustive-search approach

- Generate all the permutations of integers 1, 2, . . . , *n*,
- Compute the total cost of each assignment
- Select the one with the smallest sum.

$$C = \begin{bmatrix} 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{bmatrix}$$

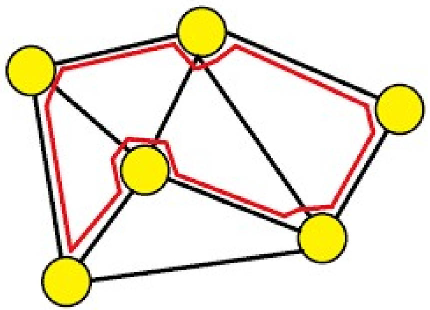
$\langle 1, 2, 3, 4 \rangle$	cost = 9 + 4 + 1 + 4 = 18	
$\langle 1, 2, 4, 3 \rangle$	cost = 9 + 4 + 8 + 9 = 30	
$\langle 1, 3, 2, 4 \rangle$	cost = 9 + 3 + 8 + 4 = 24	
$\langle 1, 3, 4, 2 \rangle$	cost = 9 + 3 + 8 + 6 = 26	etc.
$\langle 1, 4, 2, 3 \rangle$	cost = 9 + 7 + 8 + 9 = 33	
$\langle 1, 4, 3, 2 \rangle$	cost = 9 + 7 + 1 + 6 = 23	

- Number of permutations are considered for the general case of the assignment problem: $n!$
- Exhaustive search is impractical for all but very small instances of the problem

Exhaustive-search approach

- How exhaustive search can be applied to the sorting problem?

- How exhaustive search can be applied to the Hamiltonian circuit problem?



Decrease-and-Conquer

- The *decrease-and-conquer* technique is based on exploiting the relationship between a solution to a given instance of a problem and a solution to its smaller instance.
- Top-down :recursive   Bottom-up:iterative
- There are three major variations of decrease-and-conquer:

  decrease by a constant

  decrease by a constant factor

  variable size decrease

Decrease-(by one)-and-conquer technique

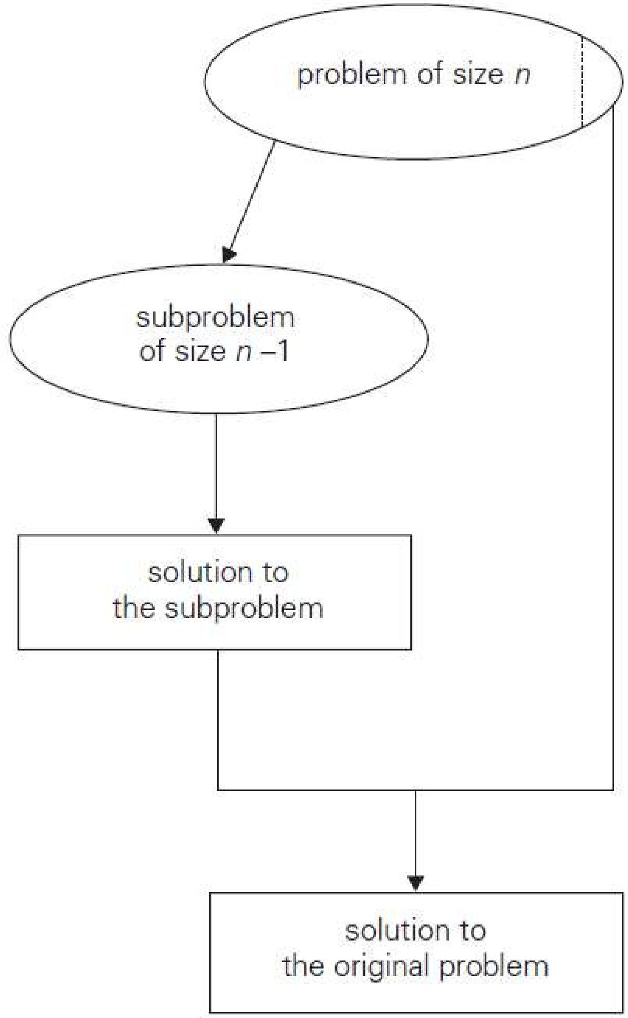
- Decrease by a constant

Top-down: recursive

$$a^n = a^{n-1} \cdot a \quad f(n) = a^n$$

$$f(n) = \begin{cases} f(n-1) \cdot a & \text{if } n > 0, \\ 1 & \text{if } n = 0, \end{cases}$$

Bottom up: a*a*a*a*a*a*a ..

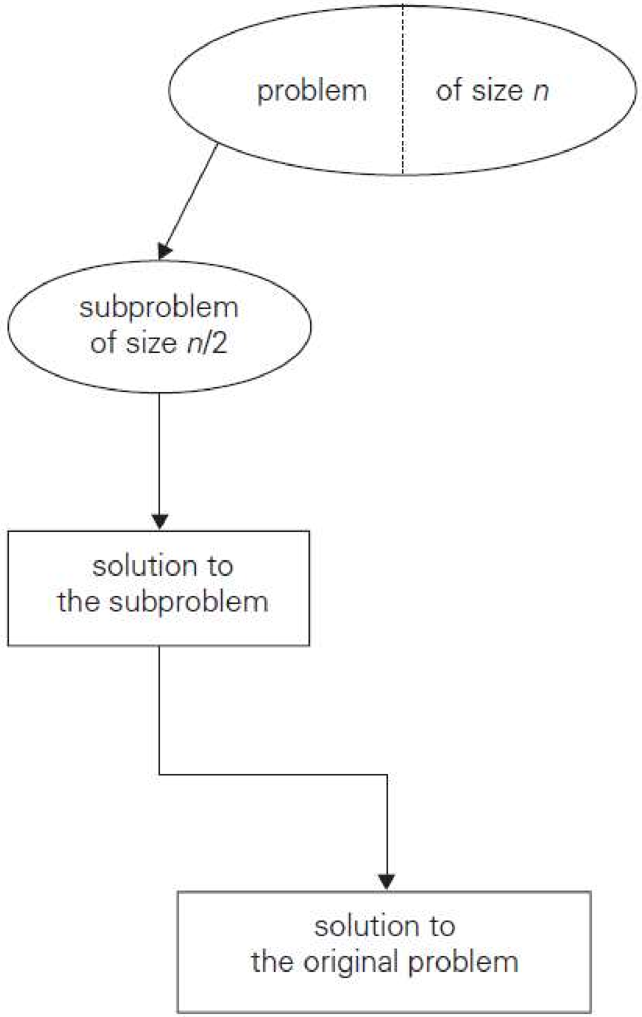


Decrease-(by half)-and-conquer technique

- Decrease by a constant factor

$$a^n = (a^{n/2})^2$$

$$a^n = \begin{cases} (a^{n/2})^2 & \text{if } n \text{ is even and positive,} \\ (a^{(n-1)/2})^2 \cdot a & \text{if } n \text{ is odd,} \\ 1 & \text{if } n = 0. \end{cases}$$



Variable-size-decrease

- Size-reduction pattern varies from one iteration of an algorithm to another
- Euclid's algorithm for computing the greatest common divisor
- gcd(u; v) = gcd(v; u mod v)

  gcd(180; 146)
  = gcd(146; 34)
  = gcd(34; 10)
  = gcd(10; 4)
  = gcd(4; 2)
  = gcd(2; 0)
  = 2

$\textsc{Euclid}(u, v)$ { inputs are integers ≥ 0}
  if $(v = 0)$ then
    return(u)
  else
    return$(\textsc{Euclid}(v, u \bmod v))$

Binary Search (Decrease by a constant factor)

ALGORITHM *BinarySearch*$(A[0..n-1], K)$

//Implements nonrecursive binary search
//Input: An array $A[0..n-1]$ sorted in ascending order and
// a search key K
//Output: An index of the array's element that is equal to K
// or -1 if there is no such element
$l \leftarrow 0;\quad r \leftarrow n-1$
while $l \le r$ **do**
  $m \leftarrow \lfloor (l+r)/2 \rfloor$
  if $K = A[m]$ **return** m
  else if $K < A[m]$ $r \leftarrow m-1$
  else $l \leftarrow m+1$
return -1

$$\begin{array}{ccc} & K & \\ & \updownarrow & \\ \underbrace{A[0]\ldots A[m-1]}_{\text{search here if } K<A[m]} & A[m] & \underbrace{A[m+1]\ldots A[n-1]}_{\text{search here if } K>A[m]} \end{array}$$

$$C_{worst}(n) = \lfloor \log_2 n \rfloor + 1 = \lceil \log_2(n+1) \rceil \qquad \Theta(\log n)$$

Binary Search

What is the largest number of key comparisons made by binary search in searching for a key in the following array?

3	14	27	31	39	42	55	70	74	81	85	93	98

Binary Search

What is the largest number of key comparisons made by binary search in searching for a key in the following array?

3	14	27	31	39	42	55	70	74	81	85	93	98

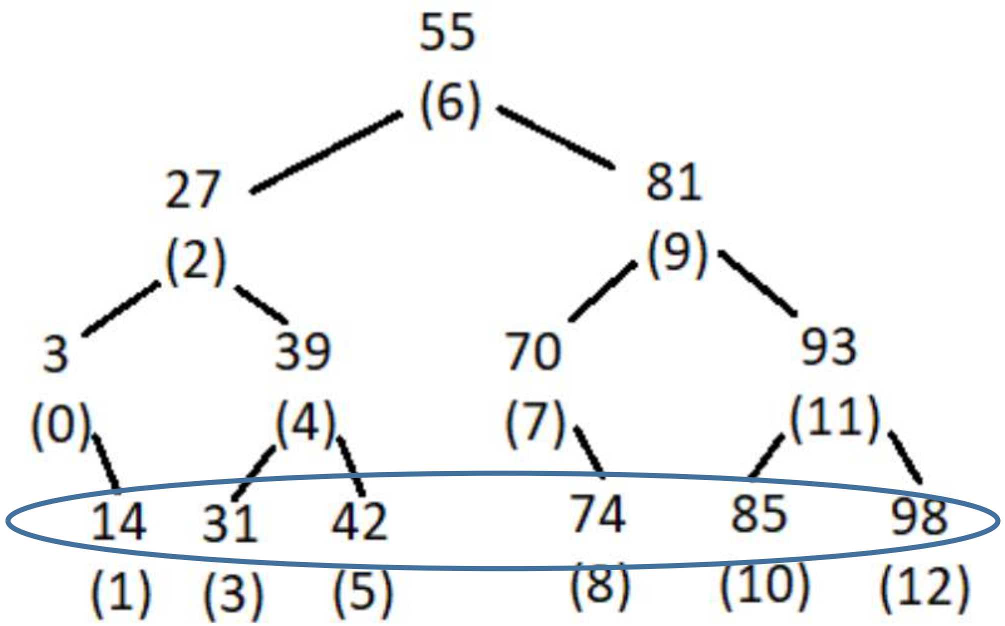
$$C_worst(n) = \lfloor log_2 n \rfloor + 1 = \lceil log_2(n+1) \rceil$$

$$\begin{aligned} C_worst(13) &= \lfloor log_2 13 \rfloor + 1 \\ &= \lfloor 3.7 \rfloor + 1 \\ &= 3 + 1 \\ &= 4 \end{aligned}$$

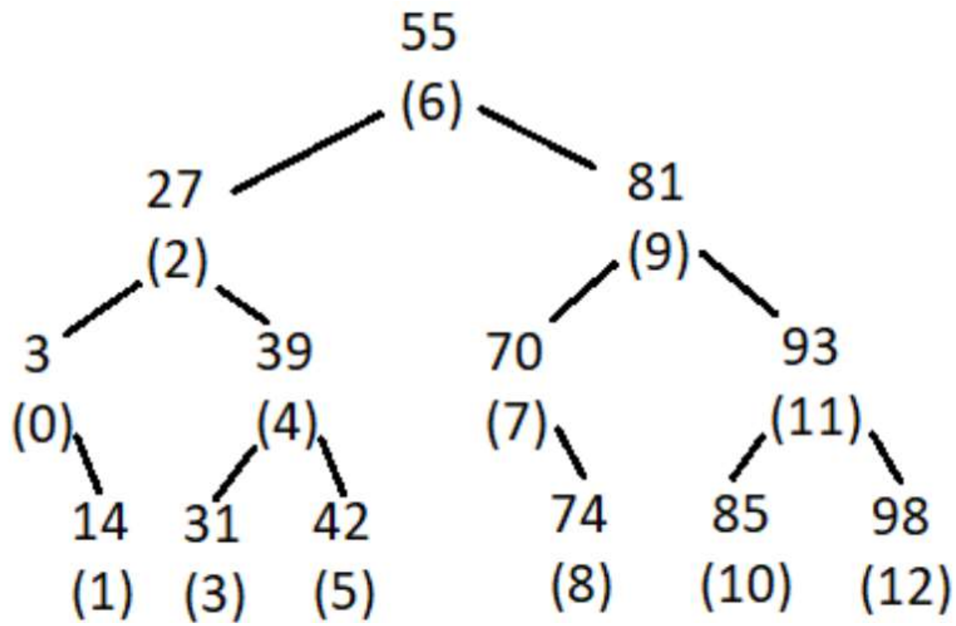
Binary Search Tree

3	14	27	31	39	42	55	70	74	81	85	93	98

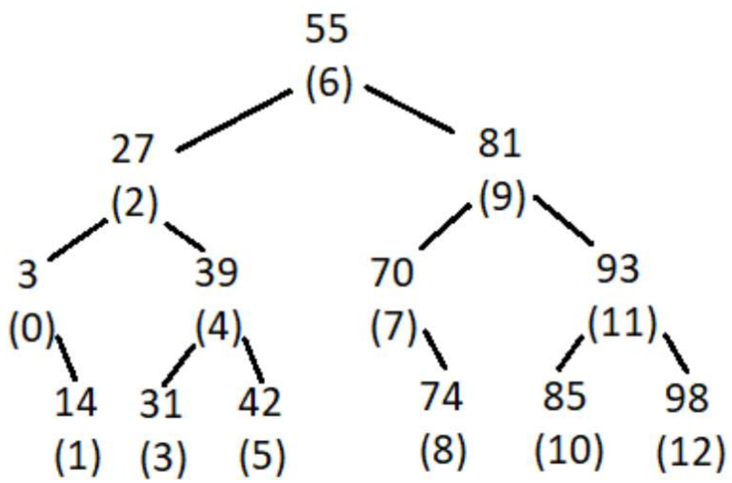
List all the keys of this array that will require the largest number of key comparisons when searched for by binary search.



Find the average number of key comparisons made by binary search in a successful search in this array. Assume that each key is searched for with the same probability.



5 (root node, first level), we will make 1 comparison
27 or 81 (second level), we will make 2 comparisons
$3, 39, 70$ or 93 (third level), we will make 3 comparisons
$14, 31, 42, 74, 85$ or 98 (fourth level, leaves), we will make 4 comparisons



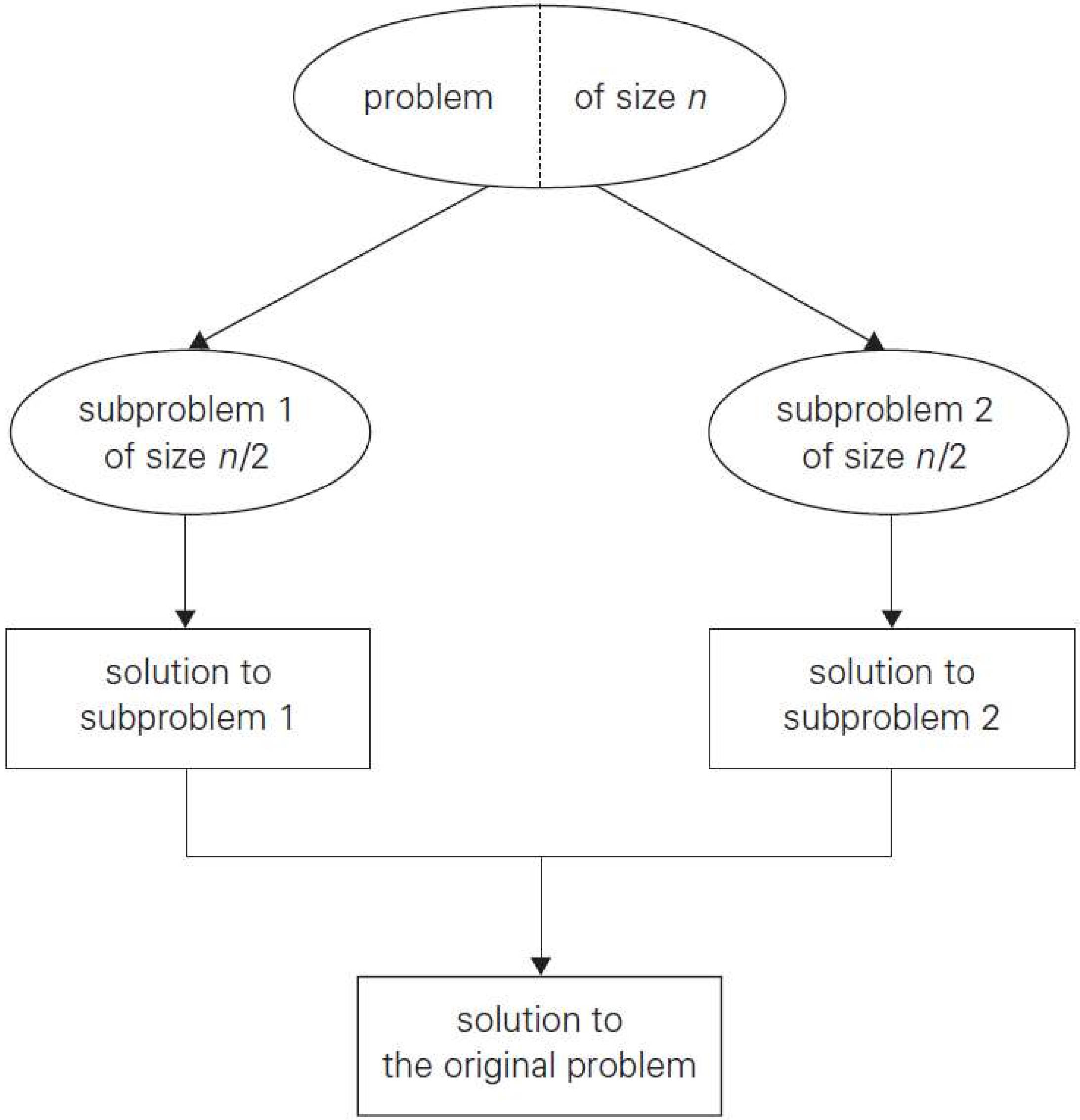
$$C_{avg}^{yes} = \frac{1 \cdot 1 + 2 \cdot 2 + 4 \cdot 3 + 6 \cdot 4}{13} = \frac{41}{13} = 3.15$$

Equation for average number of key comparisons? ($n=2^k$)

$$C_{avg}(n) = \sum_{i=1}^{\log(n)} \frac{i2^{i-1}}{n} = \frac{1}{n} \sum_{i=1}^{\log(n)} i2^{i-1} \approx \log_2 n$$

Divide-and-conquer algorithms

- A problem is divided into several subproblems of the same type, ideally of about equal size.
- The subproblems are solved (typically recursively, though sometimes a different algorithm is employed, especially when subproblems become small enough).
- If necessary, the solutions to the subproblems are combined to get a solution to the original problem.
- Cooley–Tukey Fast Fourier Transform (FFT) algorithm, Quick Sort, Convex Hull..


problem
of size n
subproblem 1
of size $n/2$
subproblem 2
of size $n/2$
solution to
subproblem 1
solution to
subproblem 2
solution to
the original problem

Divide-and-conquer algorithms

- Computing the sum of *n* numbers $a_0, \ldots, a_{n-1}$
- Divide the problem into two instances of the same problem: to compute the sum of the first $n/2$ numbers and to compute the sum of the remaining $n/2$ numbers. (if $n = 1$, return a_0,)
- Once each of these two sums is computed by applying the same method recursively, add their values to get the sum

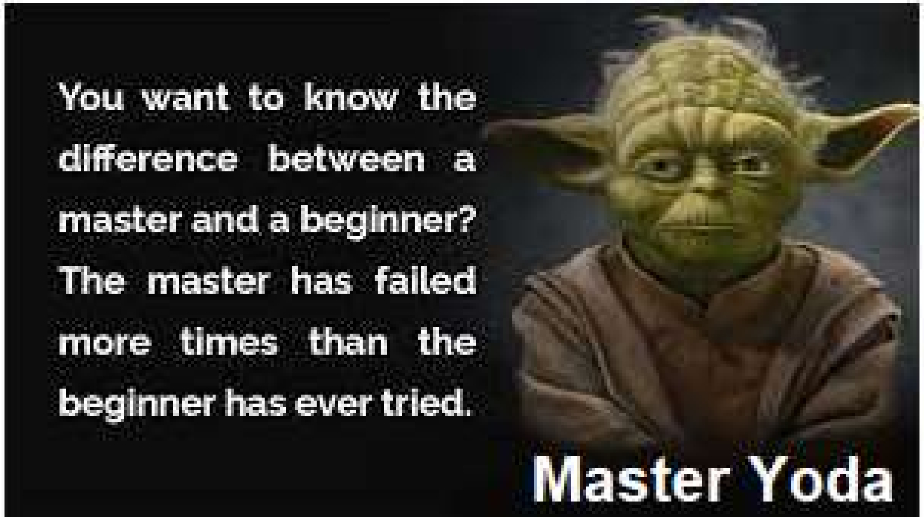
$$a_0 + \cdots + a_{n-1} = (a_0 + \cdots + a_{\lfloor n/2 \rfloor - 1}) + (a_{\lfloor n/2 \rfloor} + \cdots + a_{n-1})$$

- Is this efficient? What if you perform parallel computing?

Master Theorem

- An instance of size ***n*** can be divided into ***b*** instances of size *n/b*, with ***a*** of them needing to be solved (*a* and *b* are constants; $a \geq 1$ and $b > 1$)
- *f (n)* is a function that accounts for the time spent on dividing an instance of size *n* into instances of size *n/b* and combining their solutions
- Recurrence for the running time *T (n)*:

$$T(n) = aT(n/b) + f(n)$$



Master Theorem

- The order of growth of its solution *T (n)* depends on the values of the constants *a* and *b* and the order of growth of the function *f (n).*

$$T(n) = aT(n/b) + f(n)$$

If $f(n) \in \Theta(n^d)$ where $d \geq 0$

$$T(n) \in \begin{cases} \Theta(n^d) & \text{if } a < b^d, \\ \Theta(n^d \log n) & \text{if } a = b^d, \\ \Theta(n^{\log_b a}) & \text{if } a > b^d. \end{cases}$$

Analogous results hold for the O and Ω notations, too

Analyze computing the sum of *n* numbers by Master Teorem

$$a_0 + \cdots + a_{n-1} = (a_0 + \cdots + a_{\lfloor n/2 \rfloor - 1}) + (a_{\lfloor n/2 \rfloor} + \cdots + a_{n-1})$$

$$T(n) = aT(n/b) + f(n) \qquad f(n) \in \Theta(n^d)$$

$$n = 2^k \qquad A(n) = 2A(n/2) + 1.$$

Recurrence relation

$$a = 2,\ b = 2,\ \text{and } d = 0;\ a > b^d,$$

$$A(n) \in \Theta(n^{\log_b a}) = \Theta(n^{\log_2 2}) = \Theta(n).$$

Merge Sort

MergeSort (A[0..n-1])

if n>1

    copy $A[0..\lfloor n/2 \rfloor - 1]$ to $B[0..\lfloor n/2 \rfloor - 1]$

    copy $A[\lfloor n/2 \rfloor .. n - 1]$ to $C[0..\lceil n/2 \rceil - 1]$

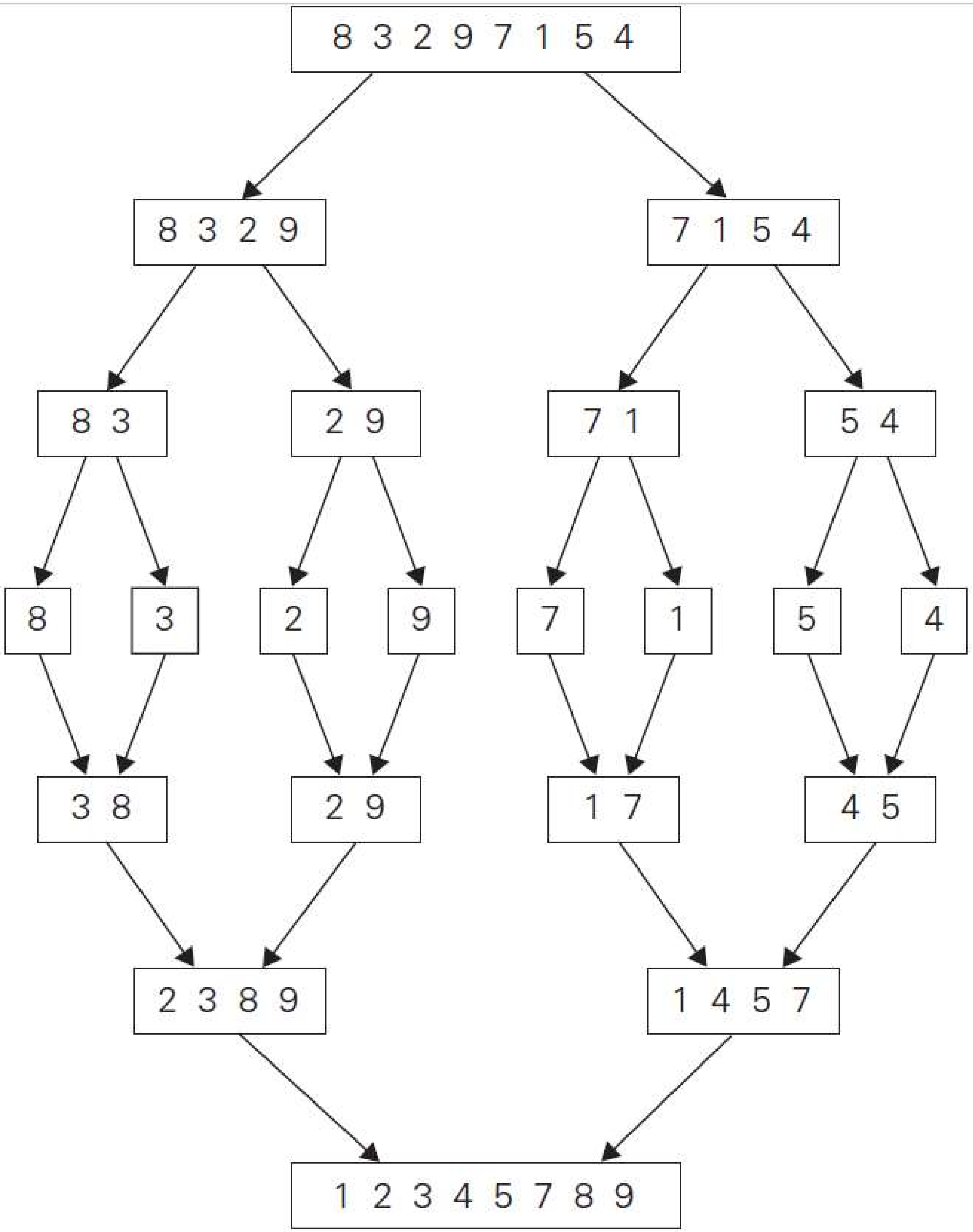
    MergeSort ($B[0..\lfloor n/2 \rfloor - 1]$)

    MergeSort ($C[0..\lceil n/2 \rceil - 1]$)

    Merge (B,C,A)

ALGORITHM $Merge(B[0..p-1], C[0..q-1], A[0..p+q-1])$

//Merges two sorted arrays into one sorted array
//Input: Arrays $B[0..p-1]$ and $C[0..q-1]$ both sorted
//Output: Sorted array $A[0..p+q-1]$ of the elements of B and C
$i \leftarrow 0$; $j \leftarrow 0$; $k \leftarrow 0$
while $i < p$ **and** $j < q$ **do**
  if $B[i] \leq C[j]$
    $A[k] \leftarrow B[i]$; $i \leftarrow i + 1$
  else $A[k] \leftarrow C[j]$; $j \leftarrow j + 1$
  $k \leftarrow k + 1$
if $i = p$
  copy $C[j..q-1]$ to $A[k..p+q-1]$
else copy $B[i..p-1]$ to $A[k..p+q-1]$


8 3 2 9 7 1 5 4
8 3 2 9
7 1 5 4
8 3
2 9
7 1
5 4
8
3
2
9
7
1
5
4
3 8
2 9
1 7
4 5
2 3 8 9
1 4 5 7
1 2 3 4 5 7 8 9

What about complexity?

Assuming for simplicity that n is a power of 2,

$$C(n) = 2C(n/2) + C_{merge}(n) \quad \text{for } n > 1, \quad C(1) = 0.$$

for the worst case, $C_{merge}(n) = n - 1$,

$$C_{worst}(n) = 2C_{worst}(n/2) + \underbrace{n - 1} \quad \text{for } n > 1, \quad C_{worst}(1) = 0.$$

1 3 5 7

2 4 6 8

Apply master teorem

$$C(n) = 2C(n/2) + C_{merge}(n) \quad \text{for } n > 1, \quad C(1) = 0.$$

$$\text{for the worst case, } C_{merge}(n) = n - 1,$$

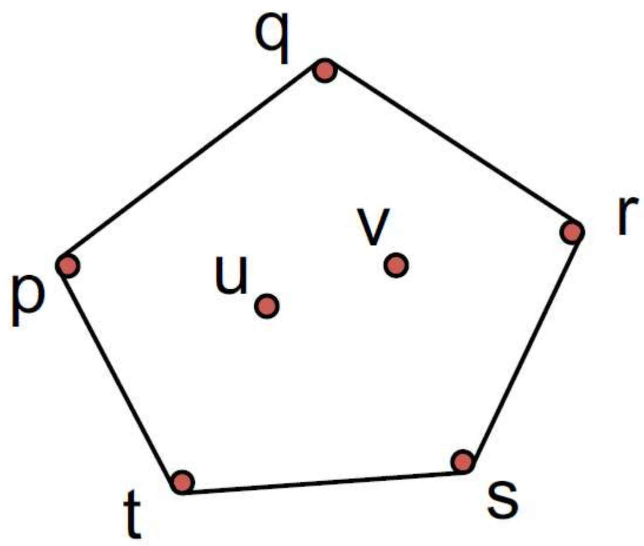
$$\text{If } f(n) \in \Theta(n^d) \text{ where } d \geq 0$$

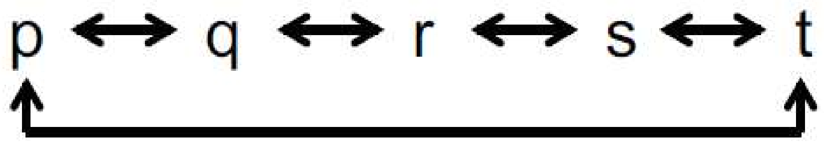
$$T(n) \in \begin{cases} \Theta(n^d) & \text{if } a < b^d, \\ \Theta(n^d \log n) & \text{if } a = b^d, \\ \Theta(n^{\log_b a}) & \text{if } a > b^d. \end{cases}$$

a= 2 b=2 d=1 -> nlogn

Convex Hull

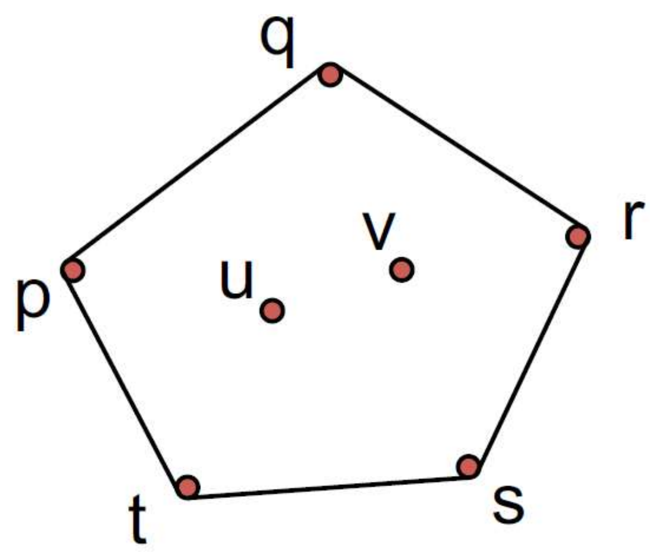
- Given n points in plane $S = \{(x_i,y_i) | i=1,2,...,n\}$
- Assume no two have same x coordinate, no two have same y coordinate, and no three in a line for convenience.
- Convex Hull CH(S): smallest polygon containing all points in S.
- CH(S) represented by the sequence of points on the boundary in order clockwise as doubly linked list





Brute force for Convex Hull

- Test each line segment to see if it makes up an edge of the convex hull
- If the rest of the points are on one side of the segment, the segment is on the convex hull.



How to test?

- The straight line through two points (x_1, y_1), (x_2, y_2) in the coordinate plane can be defined by the equation;

$$ax + by = c,$$

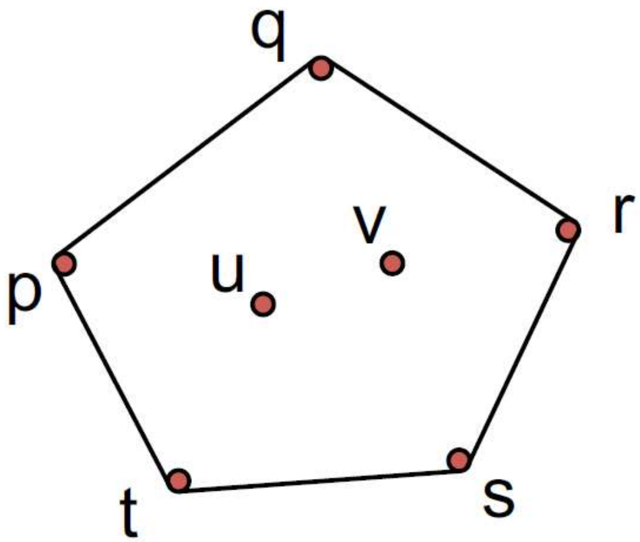
$$a = y_2 - y_1, b = x_1 - x_2, c = x_1y_2 - y_1x_2$$

- Such a line divides the plane into two half-planes: for all the points in one of them, $ax + by > c$, while for all the points in the other, $ax + by < c$.
- To check whether certain points lie on the same side of the line, we can simply check whether the expression $ax + by - c$ has the same sign for each of these points.

Brute force for Convex Hull

- $O(n^2)$ edges, $O(n)$ tests $\Rightarrow$ $O(n^3)$ complexity

- Can we do better?



Divide and Conquer Convex Hull

- Sort points by x coord (once and for all, O(nlog n))
- For input set S of points:

  Divide into left half A and right half B by x coords

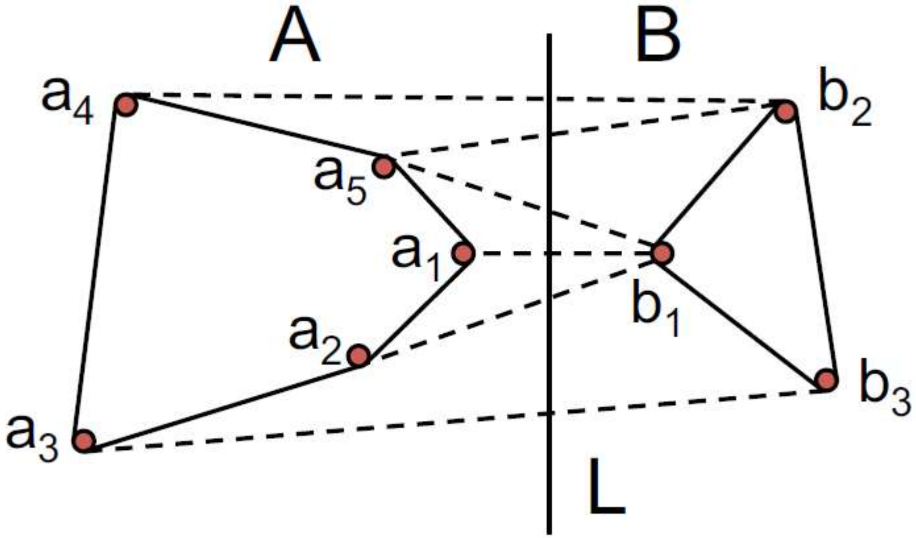
  Compute CH(A) and CH(B)

  Combine CH's of two halves (merge step)

- Divide until ..?

Merge Step

- Find upper tangent (a_i, b_j). In example, (a_4, b_2)
- Find lower tangent (a_k, b_m). In example, (a_3, b_3)



- How?

Find upper tangent

- Assume a_i maximizes x within CH(A)(a_1 , a_2 ,..., a_p).
- b_j minimizes x within CH(B) (b_1,b_2,...,b_q) L is the vertical line separating A and B.
- Define y(i,j) as y-coordinate of intersection between L and segment (a_i,b_j).
- Claim:(a_i,b_j) is uppertangent iff it maximizes y(i,j). If y(i,j) is not maximum, there will be points on both sides of (a_i,b_j) and it cannot be a tangent.
- Obvious O(n^2) algorithm looks at all a_i, b_j pairs.

According to Master Teorem

$$T(n) = aT(n/b) + f(n)$$

If $f(n) \in \Theta(n^d)$ where $d \geq 0$

$$T(n) \in \begin{cases} \Theta(n^d) & \text{if } a < b^d, \\ \Theta(n^d \log n) & \text{if } a = b^d, \\ \Theta(n^{\log_b a}) & \text{if } a > b^d. \end{cases}$$

$$T(n) = aT(\frac{n}{b}) + [\underbrace{\text{work for merge}}_{\text{F(n)}}]$$

- a=2
- b=2
- F(n)=n^2 d=2

$$T(n) = 2T(n/2) + \Theta(n^2) = \Theta(n^2)$$

Can we perform better?

```
i = 1
j = 1
while (y(i, j+1) > y(i, j) or y(i-1, j) > y(i, j))
    if (y(i, j+1) > y(i, j))  ▷ move right finger clockwise
        j = j + 1 (mod q)
    else
        i = i - 1 (mod p)  ▷ move left finger anti-clockwise
    return (a_i, b_j) as upper tangent
```

Similarly for lower tangent.

$$T(n) = aT(n/b) + f(n)$$

If $f(n) \in \Theta(n^d)$ where $d \geq 0$

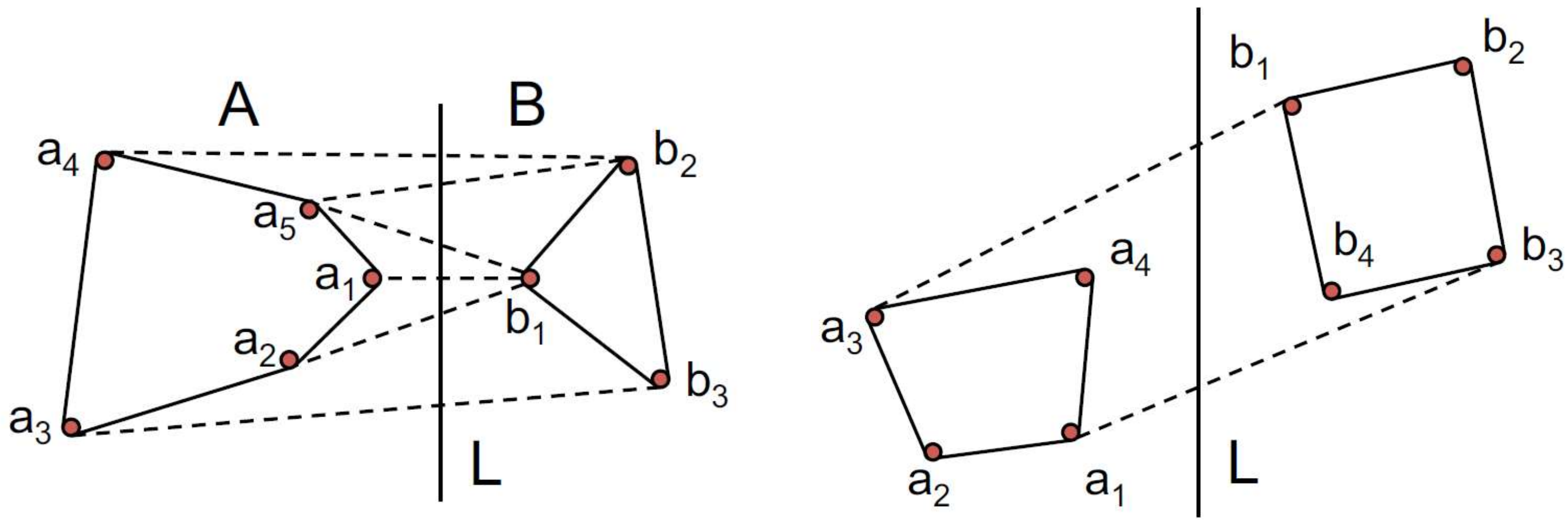
$$T(n) \in \begin{cases} \Theta(n^d) & \text{if } a < b^d, \\ \Theta(n^d \log n) & \text{if } a = b^d, \\ \Theta(n^{\log_b a}) & \text{if } a > b^d. \end{cases}$$

$$T(n) = aT(\frac{n}{b}) + [\text{work for merge}]$$

- a=2
- b=2
- F(n)=n   d=1

$$T(n) = 2T(\frac{n}{2}) + \Theta(n) = \Theta(n \log n)$$

- Can we just take highest points?



Median Finding

Given set of n numbers, define $rank(x)$ as number of numbers in the set that are $\leq x$.
Find element of rank $\lfloor \frac{n+1}{2} \rfloor$ (lower median) and $\lceil \frac{n+1}{2} \rceil$ (upper median).

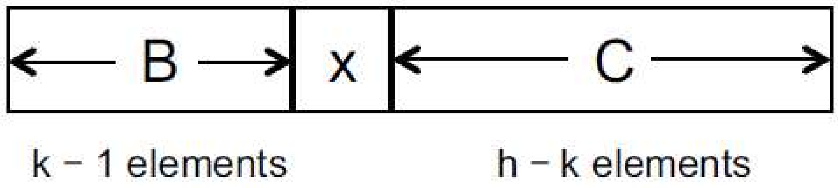
Clearly, sorting works in time $\Theta(n \log n)$.

Can we do better?

Median Finding – Divide and Conquer

SELECT(S, i)

Pick $x \in S$ ▷ cleverly
Compute $k = rank(x)$
$B = \{y \in S | y < x\}$
$C = \{y \in S | y > x\}$
if $k = i$
    return x
else if $k > i$
    return Select(B, i)
else if $k < i$
    return Select$(C, i - k)$



i : rank that you want to find
K : rank of pivot

Median Finding – Divide and Conquer

- How to select x?
- Best case?
- Worst case?
- Can we apply Master Theorem?

$$T(n) = aT(n/b) + f(n)$$

If $f(n) \in \Theta(n^d)$ where $d \geq 0$

$$T(n) \in \begin{cases} \Theta(n^d) & \text{if } a < b^d, \\ \Theta(n^d \log n) & \text{if } a = b^d, \\ \Theta(n^{\log_b a}) & \text{if } a > b^d. \end{cases}$$

Recall the Master Theorem

- Master Theorem doesn't apply here
- Lets pretend we know the problem size!! Assume we obtained a balanced partitioning:

In our case:

- $T(n) \leq T\left(\frac{n}{2}\right) + O(n)$
- So a = 1, b = 2, d = 1
- $T(n) \leq O\left(n^d\right) = O(n)$

- What if len(B) is about 7n/10

In our case:

- $T(n) \leq T\left(\frac{7n}{10}\right) + O(n)$
- So a = 1, b = 10/7, d = 1
- $T(n) \leq O(n^d) = O(n)$

$$T(n) = aT(n/b) + f(n)$$

$$\text{If } f(n) \in \Theta(n^d) \text{ where } d \geq 0$$

$$T(n) \in \begin{cases} \Theta(n^d) & \text{if } a < b^d, \\ \Theta(n^d \log n) & \text{if } a = b^d, \\ \Theta(n^{\log_b a}) & \text{if } a > b^d. \end{cases}$$

Still O(n) ☺

- What about the worst case?

- What about the worst case? What if the selected pivot is always the biggest (or smallest) one?

In our case:

- $T(n) \leq T(n-1) + O(n)$
- So a = 1, b = n/(n -1) , d = 1
- $T(n) \leq O(n)$ still?
- NO!!! b needs to be independent of n for the master thm to work.

- What about the worst case?

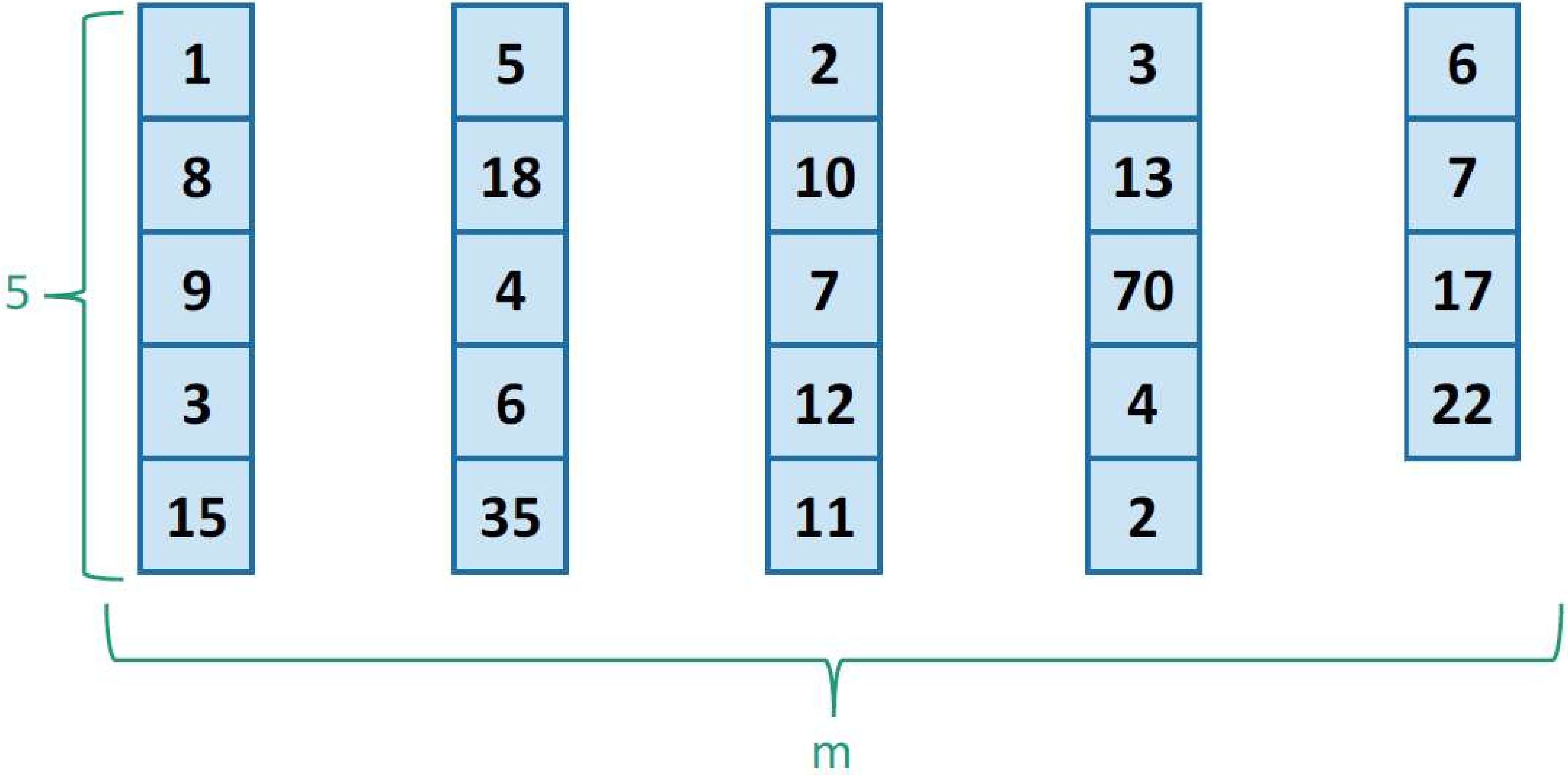
Actual running time is $O(n^2)$

- Why?

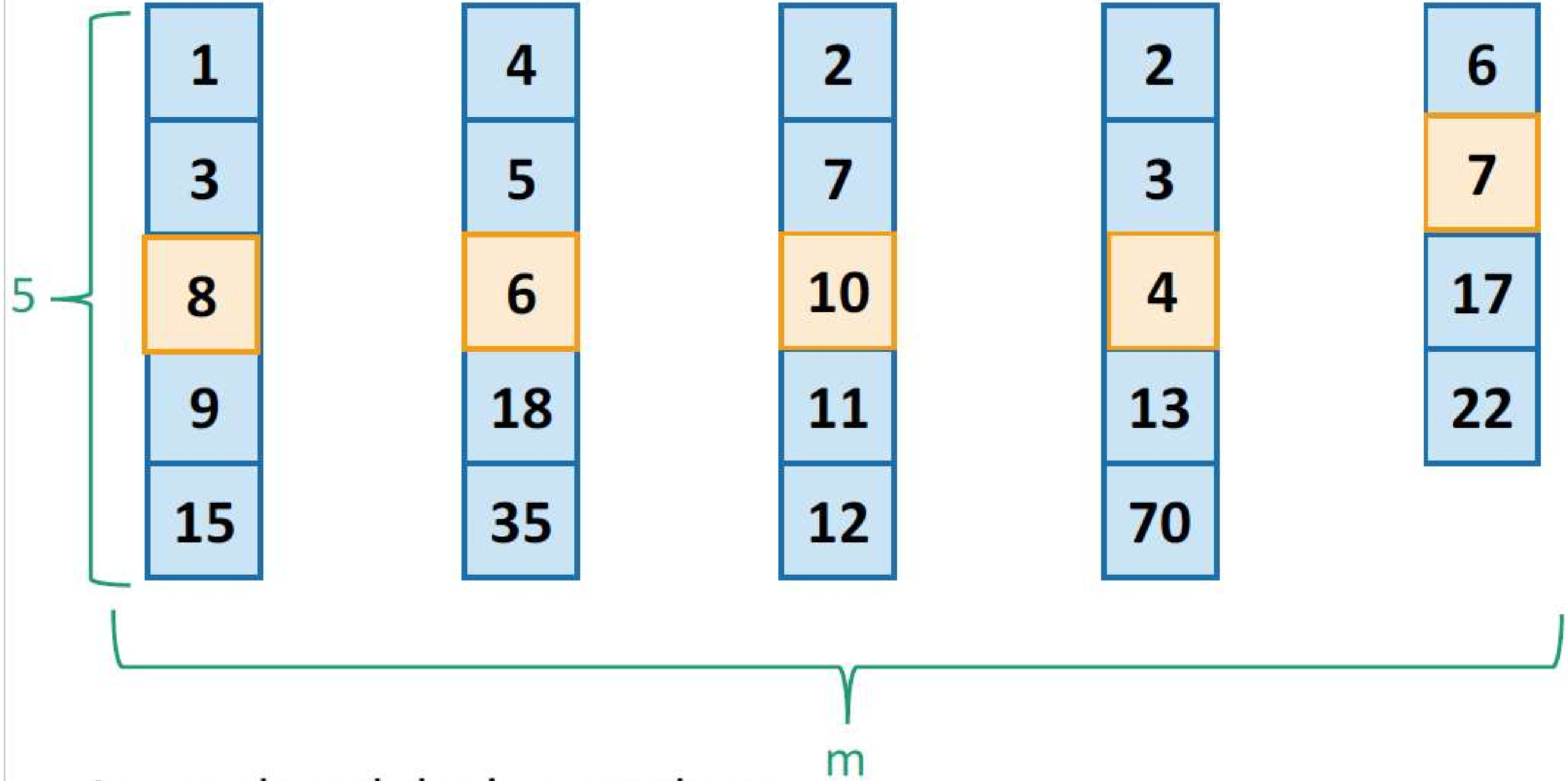
Picking x Cleverly

Need to pick x so $rank(x)$ is not extreme.

- Arrange S into columns of size 5 ($\lceil \frac{n}{5} \rceil$ cols)
- Sort each column
- Find "median of medians" as x

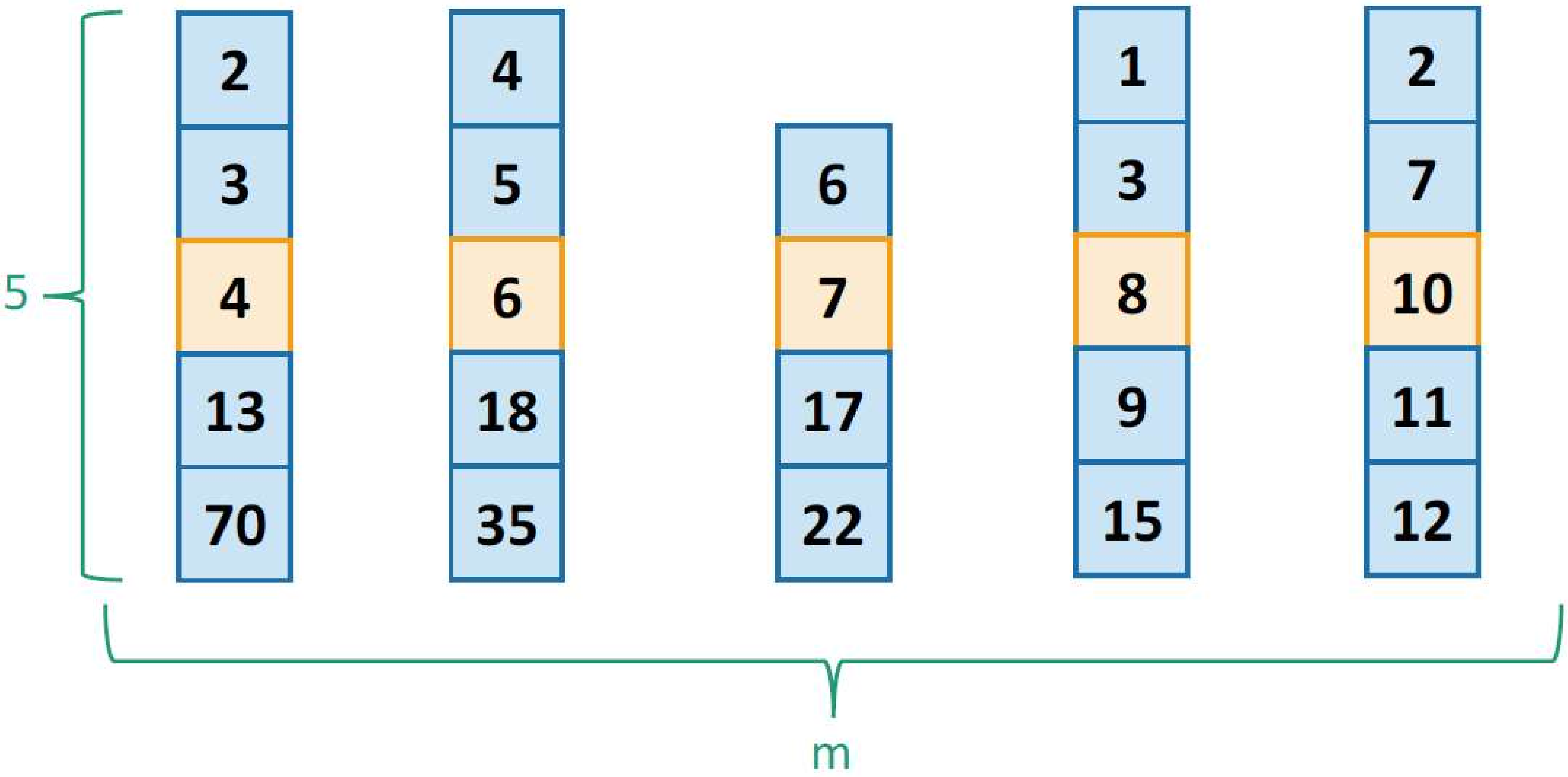


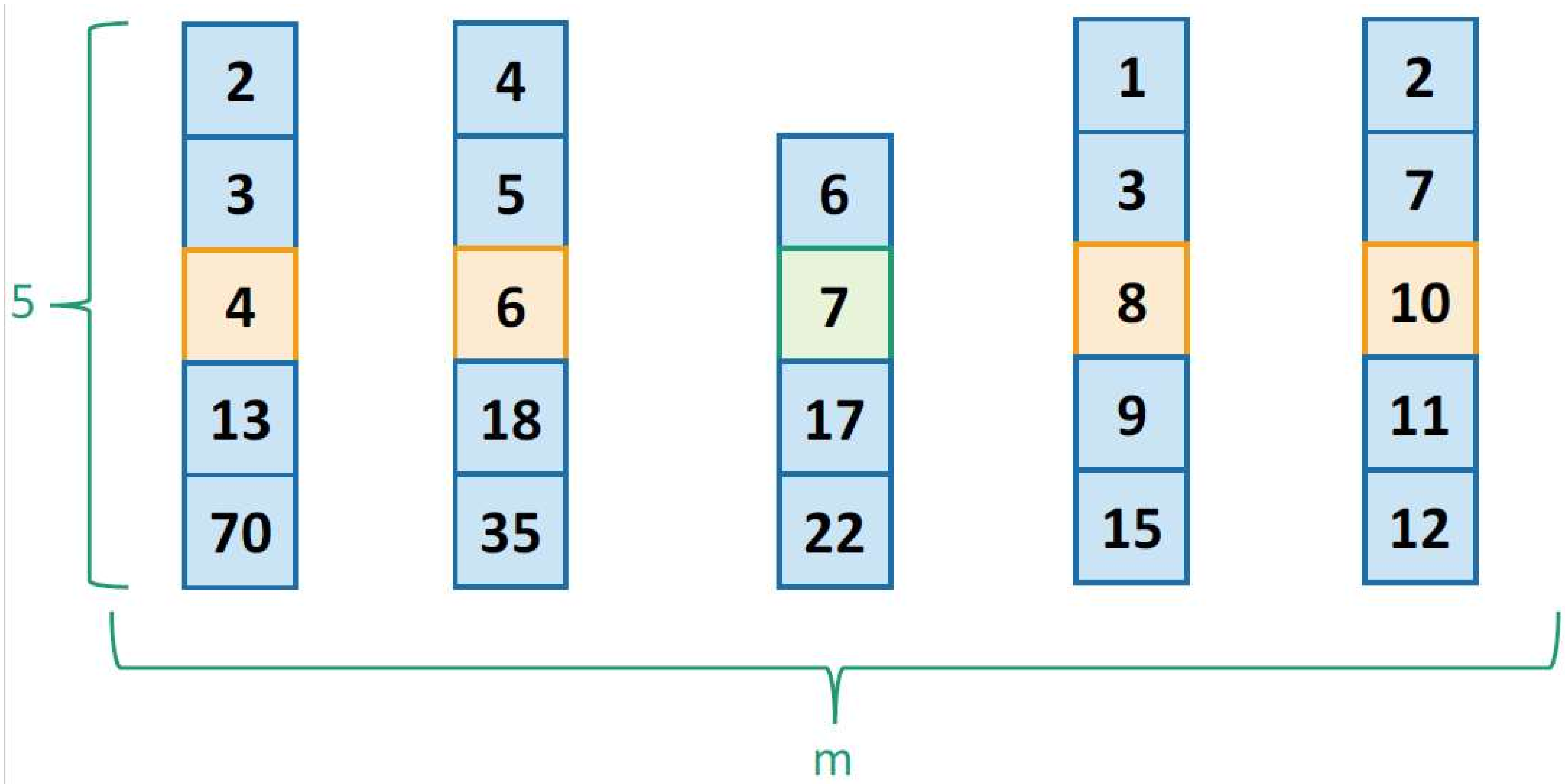
1 8 9 3 15 5 18 4 6 35 2 10 7 12 11 3 13 10 70 4 2 6 7 17 22

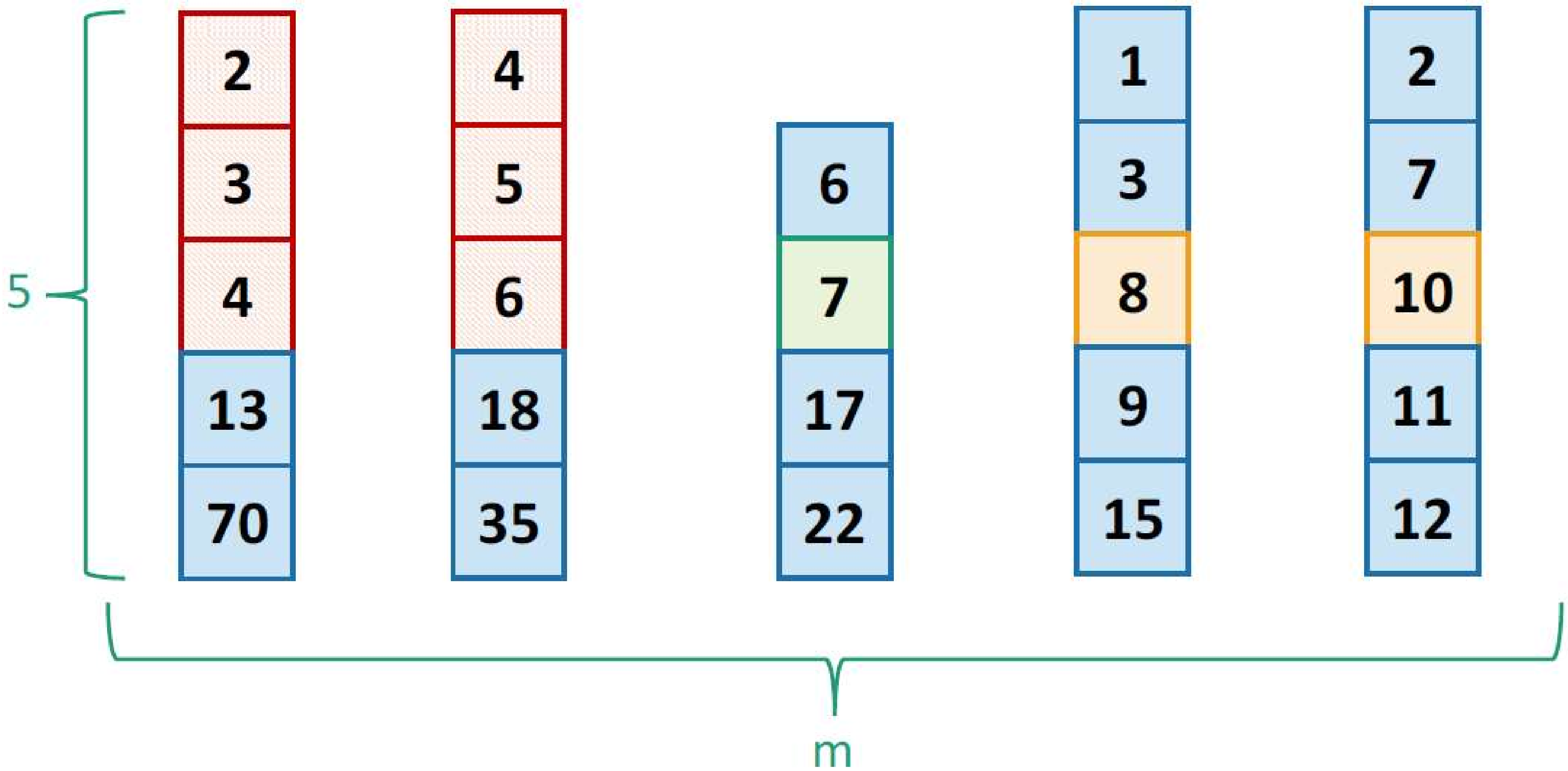


In our head, let's sort them.

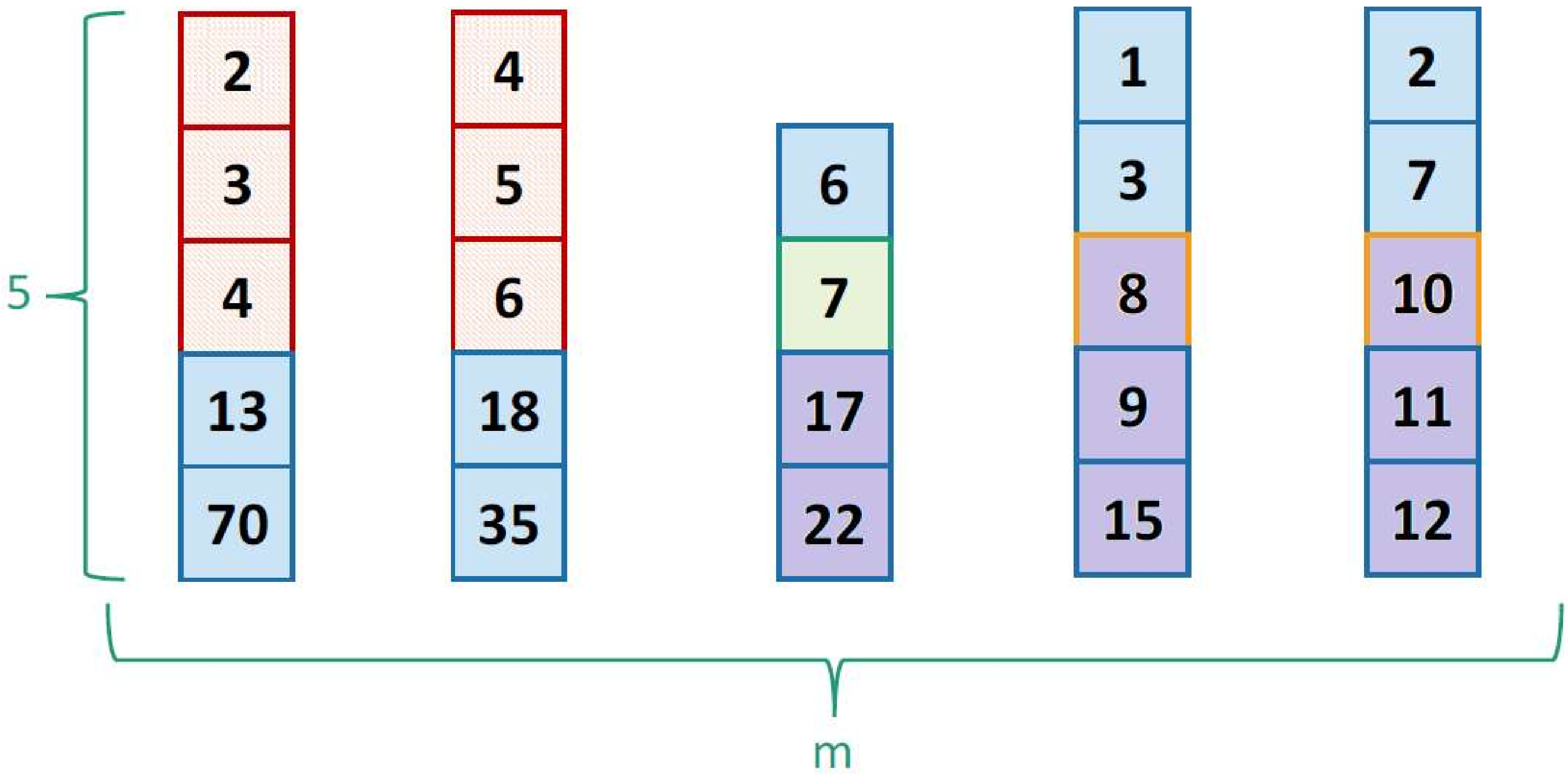
Then find medians.


5
2
3
4
13
70
4
5
6
18
35
6
7
17
22
1
3
8
9
15
2
7
10
11
12
m


2
3
4
13
70
4
5
6
18
35
6
7
17
22
1
3
8
9
15
2
7
10
11
12
5
m



5
2
3
4
13
70
4
5
6
18
35
6
7
17
22
1
3
8
9
15
2
7
10
11
12
m



Can i find median of median
without sorting?

Median Finding – Divide and Conquer

SELECT(S, i)

Pick $x \in S$ ▷ cleverly
Compute $k = rank(x)$
$B = \{y \in S | y < x\}$
$C = \{y \in S | y > x\}$
if $k = i$
    return x
else if $k > i$
    return Select(B, i)
else if $k < i$
    return Select$(C, i - k)$

i : rank that you want to find
K : rank of pivot

Median Finding – Divide and Conquer

SELECT(S, i)

Pick $x \in S \triangleright$ cleverly
Compute $k = rank(x)$
$B = \{y \in S | y < x\}$
$C = \{y \in S | y > x\}$
if $k = i$
    return x
else if $k > i$
    return Select(B, i)
else if $k < i$
    return Select$(C, i - k)$

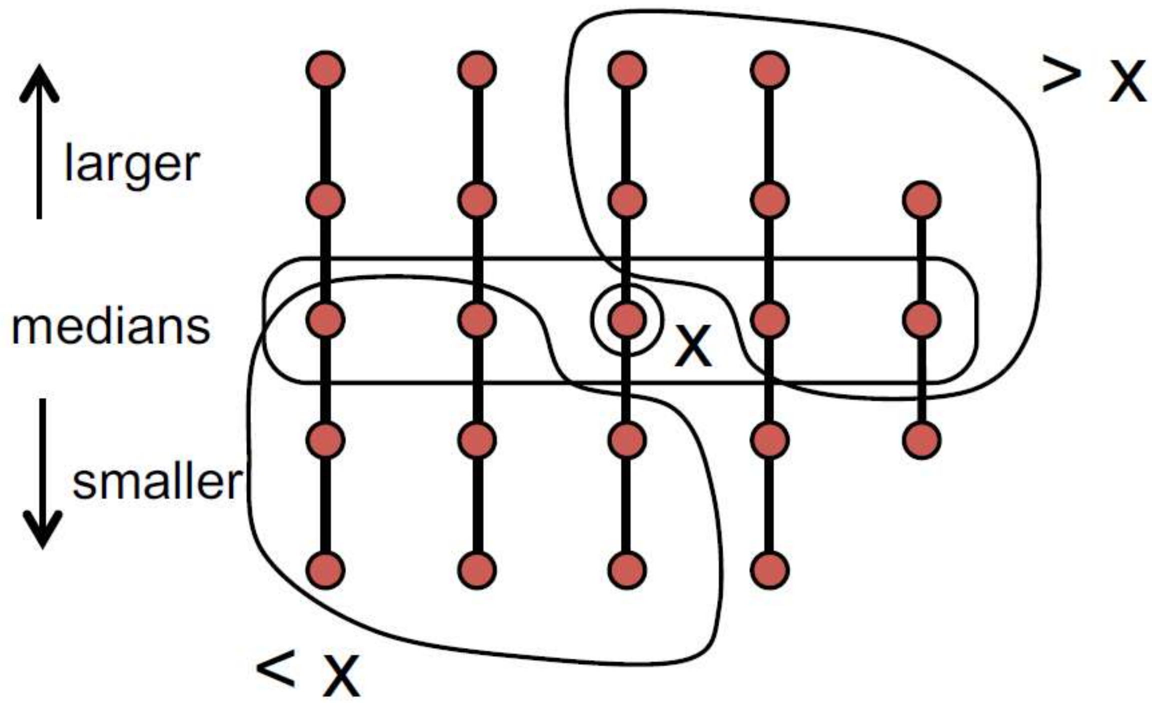
If len(**S**) <= 50:

- **S** = MergeSort(**S**)
- Return **S**[**i**]

i : rank that you want to find
K : rank of pivot

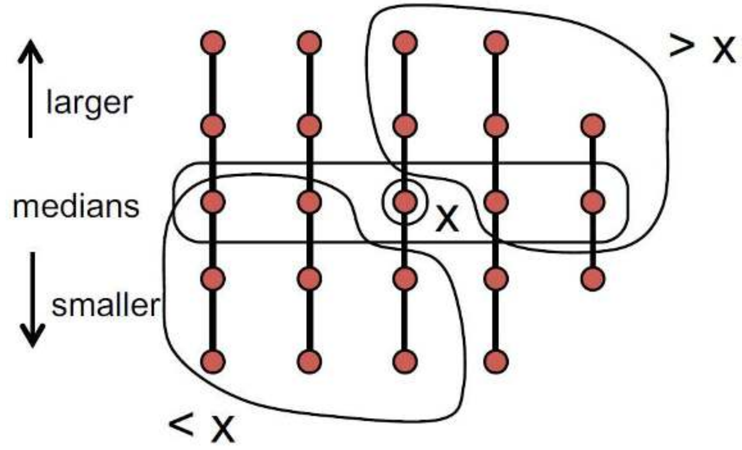
What about complexity?

- What are sizes of subgroups?



(bigger elements on top)

What about complexity?



3n/10 -6

At lease $3(\lceil \frac{n}{10} \rceil - 2)$ elements are $> x$

Recurrence:

Some reasonable constant

To partition columns
To sort n/5 columns
To divide array (smaller and bigger ones)

$$T(n) = \begin{cases} O(1), & \text{for } n \leq 140 \\ T(\lceil \frac{n}{5} \rceil) + T(\frac{7n}{10} + 6) + \Theta(n), & \text{for } n > 140 \end{cases}$$

Median of medians

Discard one half (at most)