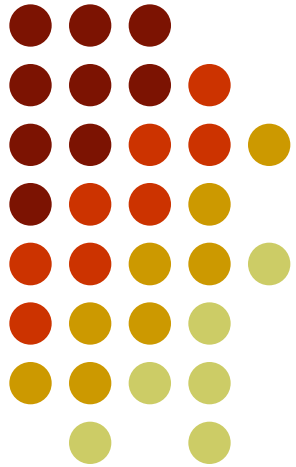
CSE 5311 Advanced Algorithms

Submitted By:

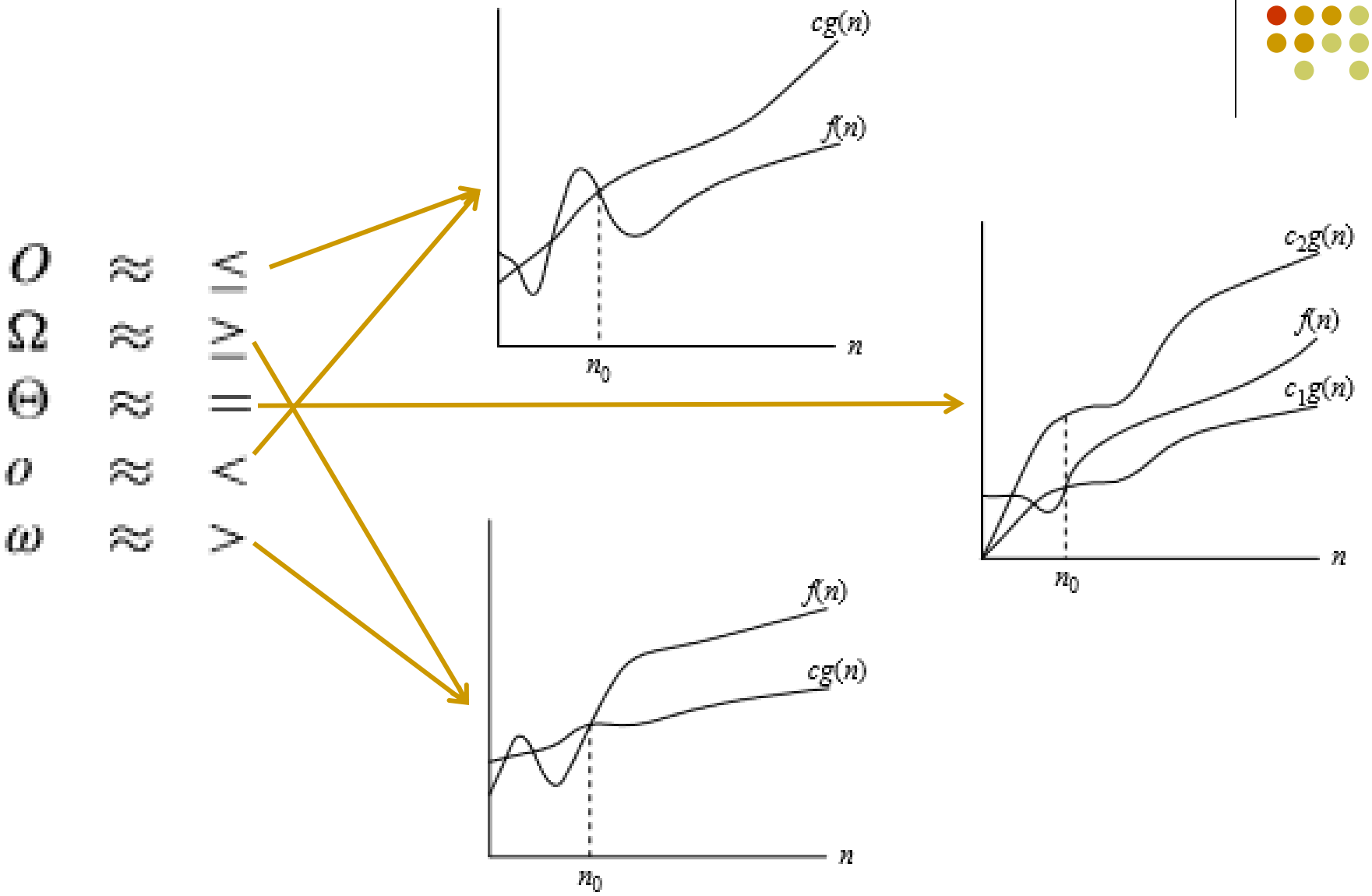
Abhishek Hemrajani & Pinky Dewani



Topics Covered This Week:

- ✓ Divide-and-Conquer
- ✓ Median Finding Algorithm (Description and Analysis)
- ✓ Binary Search Tree

Growth of Functions (reminder)



Divide-and-Conquer

Recall the divide-and-conquer paradigm, which we used for merge sort:

Divide the problem into a number of subproblems that are smaller instances of the same problem.

Conquer the subproblems by solving them recursively.

Base case: If the subproblems are small enough, just solve them by brute force.

Combine the subproblem solutions to give a solution to the original problem.

Use a recurrence to characterize the running time of a divide-and-conquer algorithm. Solving the recurrence gives us the asymptotic running time.

Analyzing Divide-and-Conquer

A *recurrence* is a function is defined in terms of

- one or more base cases, and
- itself, with smaller arguments.

Examples

- $$T(n) = \begin{cases} 1 & \text{if } n = 1\,, \\ T(n-1) + 1 & \text{if } n > 1\,. \end{cases}$$

  Solution: $T(n) = n$.

- $$T(n) = \begin{cases} 1 & \text{if } n = 1\,, \\ 2T(n/2) + n & \text{if } n \geq 1\,. \end{cases}$$

  Solution: $T(n) = n \lg n + n$.

- $$T(n) = \begin{cases} 0 & \text{if } n = 2\,, \\ T(\sqrt{n}) + 1 & \text{if } n > 2\,. \end{cases}$$

  Solution: $T(n) = \lg \lg n$.

Solving Recurrence

Master method

Used for many divide-and-conquer recurrences of the form

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

where $a \geq 1$, $b > 1$, and $f(n) > 0$.

Based on the ***master theorem*** (Theorem 4.1).

Compare $n^{\log_b a}$ vs. $f(n)$:

Case 1: $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$.

Case 2: $f(n) = \Theta(n^{\log_b a} \lg^k n)$, where $k \geq 0$.

Case 3: $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$ and $f(n)$ satisfies the regularity condition $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n.

Solving Recurrence

Substitution method

1. Guess the solution.
2. Use induction to find the constants and show that the solution works.

Recursion Trees

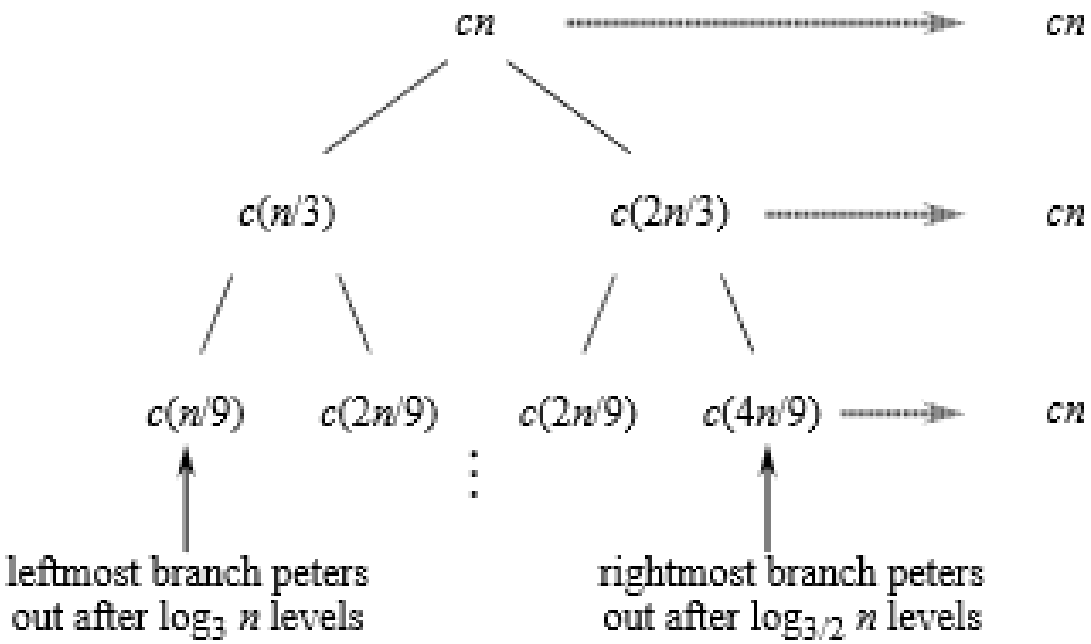
Use to generate a guess. Then verify by substitution method.

Example

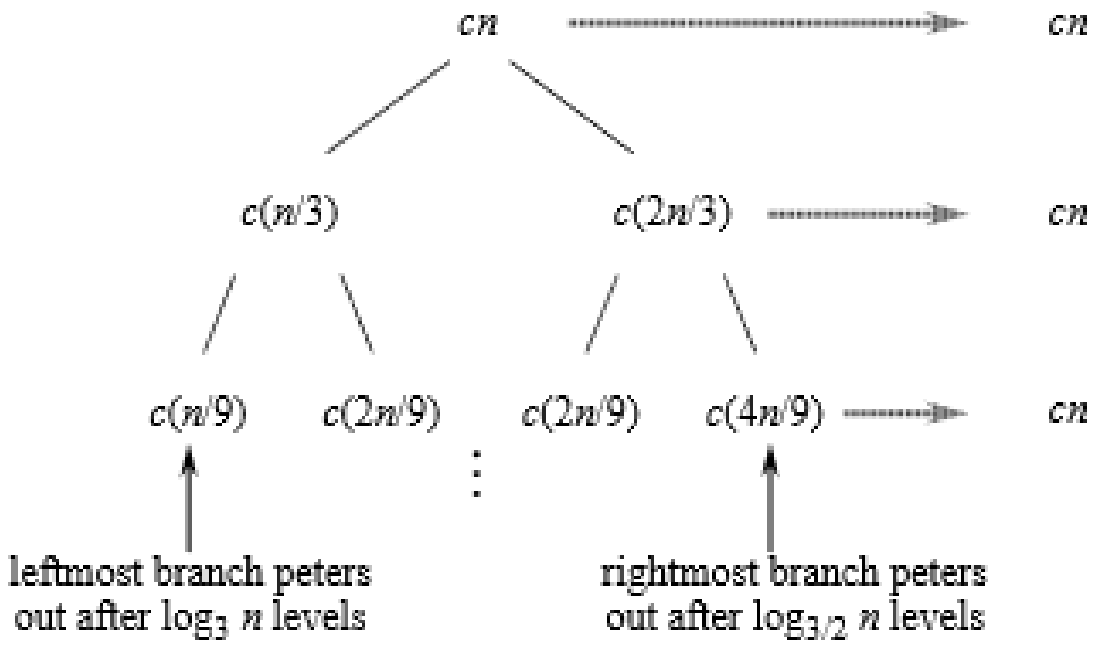
$T(n) = T(n/3) + T(2n/3) + \Theta(n)$.

For upper bound, rewrite as $T(n) \leq T(n/3) + T(2n/3) + cn$; for lower bound, as $T(n) \geq T(n/3) + T(2n/3) + cn$.

By summing across each level, the recursion tree shows the cost at each level of recursion (minus the costs of recursive calls, which appear in subtrees):



Recursion Trees Example



- There are $\log_3 n$ full levels, and after $\log_{3/2} n$ levels, the problem size is down to 1.
- Each level contributes $\leq cn$.
- Lower bound guess: $\geq dn \log_3 n = \Omega(n \lg n)$ for some positive constant d.
- Upper bound guess: $\leq dn \log_{3/2} n = O(n \lg n)$ for some positive constant d.
- Then *prove* by substitution.

Recursion Trees Example

1. ***Upper bound:***

   Guess: $T(n) \le d\,n \lg n$.

   Substitution:

$$
\begin{aligned}
T(n) &\le T(n/3) + T(2n/3) + cn \\
&\le d(n/3)\lg(n/3) + d(2n/3)\lg(2n/3) + cn \\
&= (d(n/3)\lg n - d(n/3)\lg 3) \\
&\qquad + (d(2n/3)\lg n - d(2n/3)\lg(3/2)) + cn \\
&= d\,n\lg n - d((n/3)\lg 3 + (2n/3)\lg(3/2)) + cn \\
&= d\,n\lg n - d((n/3)\lg 3 + (2n/3)\lg 3 - (2n/3)\lg 2) + cn \\
&= d\,n\lg n - d\,n(\lg 3 - 2/3) + cn \\
&\le d\,n\lg n \qquad \text{if } -d\,n(\lg 3 - 2/3) + cn \le 0\,, \\
&\qquad\qquad\qquad\qquad\qquad\qquad d \ge \frac{c}{\lg 3 - 2/3}\,.
\end{aligned}
$$

   Therefore, $T(n) = O(n \lg n)$.

Recursion Trees Example

2. ***Lower bound:***

   Guess: $T(n) \geq d\, n \lg n$.

   Substitution: Same as for the upper bound, but replacing $\leq$ by $\geq$. End up needing

   $$0 < d \leq \frac{c}{\lg 3 - 2/3}\ .$$

   Therefore, $T(n) = \Omega(n \lg n)$.

Since $T(n) = O(n \lg n)$ and $T(n) = \Omega(n \lg n)$, we conclude that $T(n) = \Theta(n \lg n)$. ■

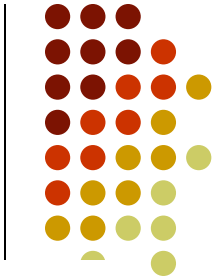
Median Finding Algorithm

Given a set of "n" numbers we can say that,

- ***Mean:*** Average of the "n" numbers
- ***Median:*** Having sorted the "n" numbers, the value which lies in the middle of the list such that half the numbers are higher than it and half the numbers are lower than it.

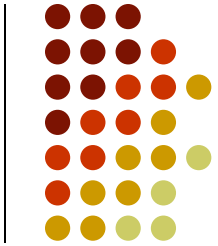
  Thus the problem of finding the median can be generalized to finding the i^{th} largest number where $i = n/2$

Order Statistics



- ***i th order statistic*** is the ith smallest element of a set of n elements.
- The ***minimum*** is the first order statistic ($i = 1$).
- The ***maximum*** is the nth order statistic ($i = n$).
- A ***median*** is the "halfway point" of the set.
- When n is odd, the median is unique, at $i = (n + 1)/2$.
- When n is even, there are two medians:
  - The ***lower median***, at $i = n/2$, and
  - The ***upper median***, at $i = n/2 + 1$.
  - We mean lower median when we use the phrase "the median."

The Selection Problem



Input: A set A of n distinct numbers and a number i, with $1 \le i \le n$.

Output: The element $x \in A$ that is larger than exactly $i - 1$ other elements in A.
In other words, the ith smallest element of A.

We can easily solve the selection problem in $O(n \lg n)$ time:

- Sort the numbers using an $O(n \lg n)$-time algorithm, such as heapsort or merge sort.
- Then return the ith element in the sorted array.

There are faster algorithms, however.

First we need to learn about partitioning:

Partion **is to divide an array** ***A*** **into two sub-arrays** ***A*****[***p***…***q***-1] and A[***q***+1…***r***],**
such that, ***A*****[***p***…***q***-1]** $\le$ ***A*****[***q***]** $\le$ ***A*****[***q***+1..***r***]**

Partitioning

Partition subarray $A[p \,..\, r]$ by the following procedure:

PARTITION(A, p, r)

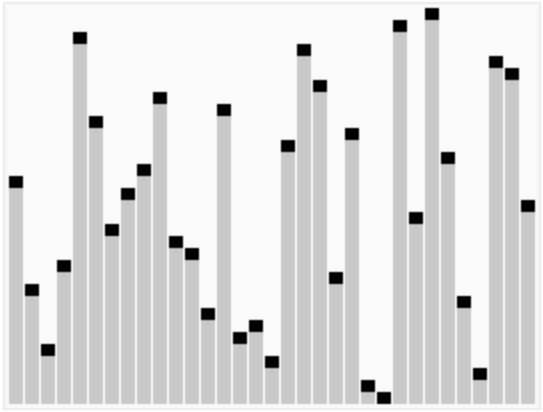
```
x = A[r]
i = p − 1
for j = p to r − 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

As the procedure executes, the array is partitioned into four regions, some of which may be empty:

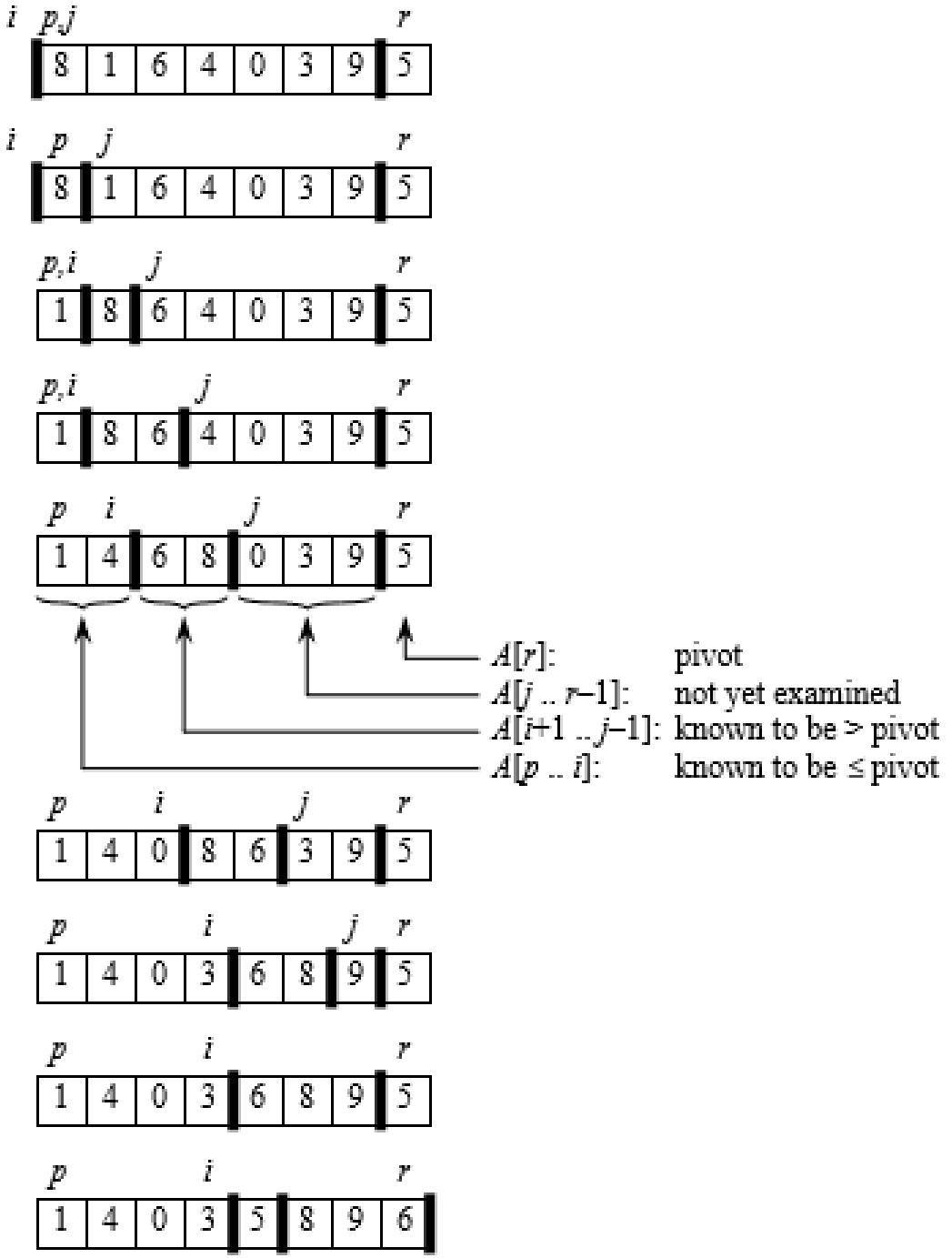
Partitioning

PARTITION(A, p, r)

$x = A[r]$
$i = p - 1$
for $j = p$ **to** $r - 1$
  if $A[j] \le x$
    $i = i + 1$
    exchange $A[i]$ with $A[j]$
exchange $A[i + 1]$ with $A[r]$
return $i + 1$



Source: https://en.wikipedia.org/wiki/Quickselect#/media/File:Selecting_quickselect_frames.gif



Now, SELECT

```
SELECT(A,p, r, i)
if p == r
    return A[p]
q = PARTITION(A, p, r)
k = q – p + 1
if i == k    // pivot value is the answer
    return A[q]
elseif  i  <  k
    return SELECT(A, p, q-1, i)
else
    return SELECT(A, q+1, r, i-k)
```

SELECT returns the i-th smallest element within the array elements to the left and the array elements to the right (i.e. A[left] <= A[i] <= A[right])

Median Finding in Linear Time

1. Divide the n elements into groups of 5. Get $\lceil n/5 \rceil$ groups: $\lfloor n/5 \rfloor$ groups with exactly 5 elements and, if 5 does not divide n, one group with the remaining $n \bmod 5$ elements.
2. Find the median of each of the $\lceil n/5 \rceil$ groups:
   - Run insertion sort on each group. Takes $O(1)$ time per group since each group has ≤ 5 elements.
   - Then just pick the median from each group, in $O(1)$ time.
3. Find the median x of the $\lceil n/5 \rceil$ medians by a recursive call to SELECT. (If $\lceil n/5 \rceil$ is even, then follow our convention and find the lower median.)
4. Using the modified version of PARTITION that takes the pivot element as input, partition the input array around x. Let x be the kth element of the array after partitioning, so that there are $k-1$ elements on the low side of the partition and $n-k$ elements on the high side.
5. Now there are three possibilities:
   - If $i = k$, just return x.
   - If $i < k$, return the ith smallest element on the low side of the partition by making a recursive call to SELECT.
   - If $i > k$, return the $(i-k)$th smallest element on the high side of the partition by making a recursive call to SELECT.

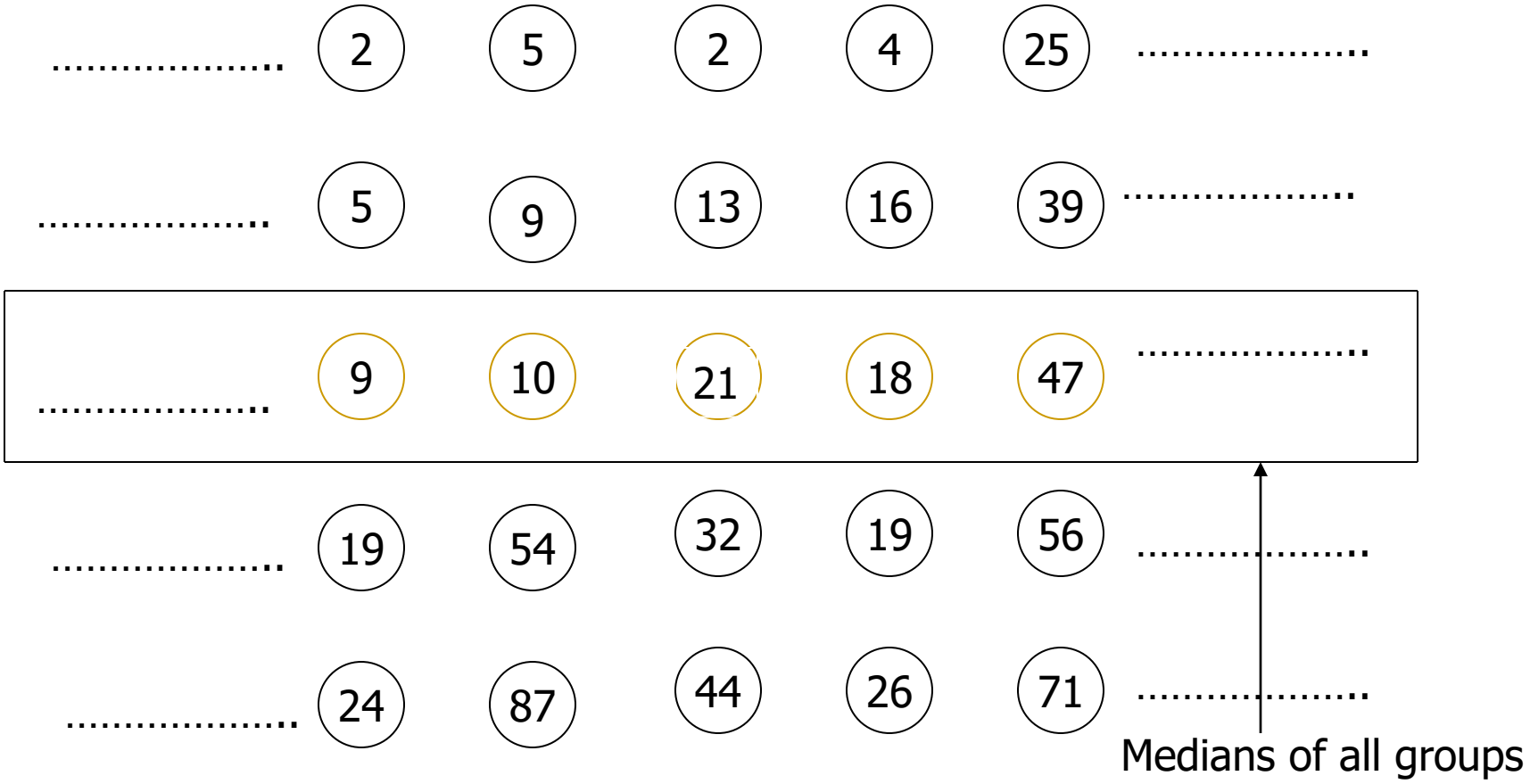
Example:

(……..2,5,9,19,24,54,5,87,9,10,44,32,21,13,24, 18,26,16,19,25,39,47,56,71,91,61,44,28…… …) is a set of “n” numbers

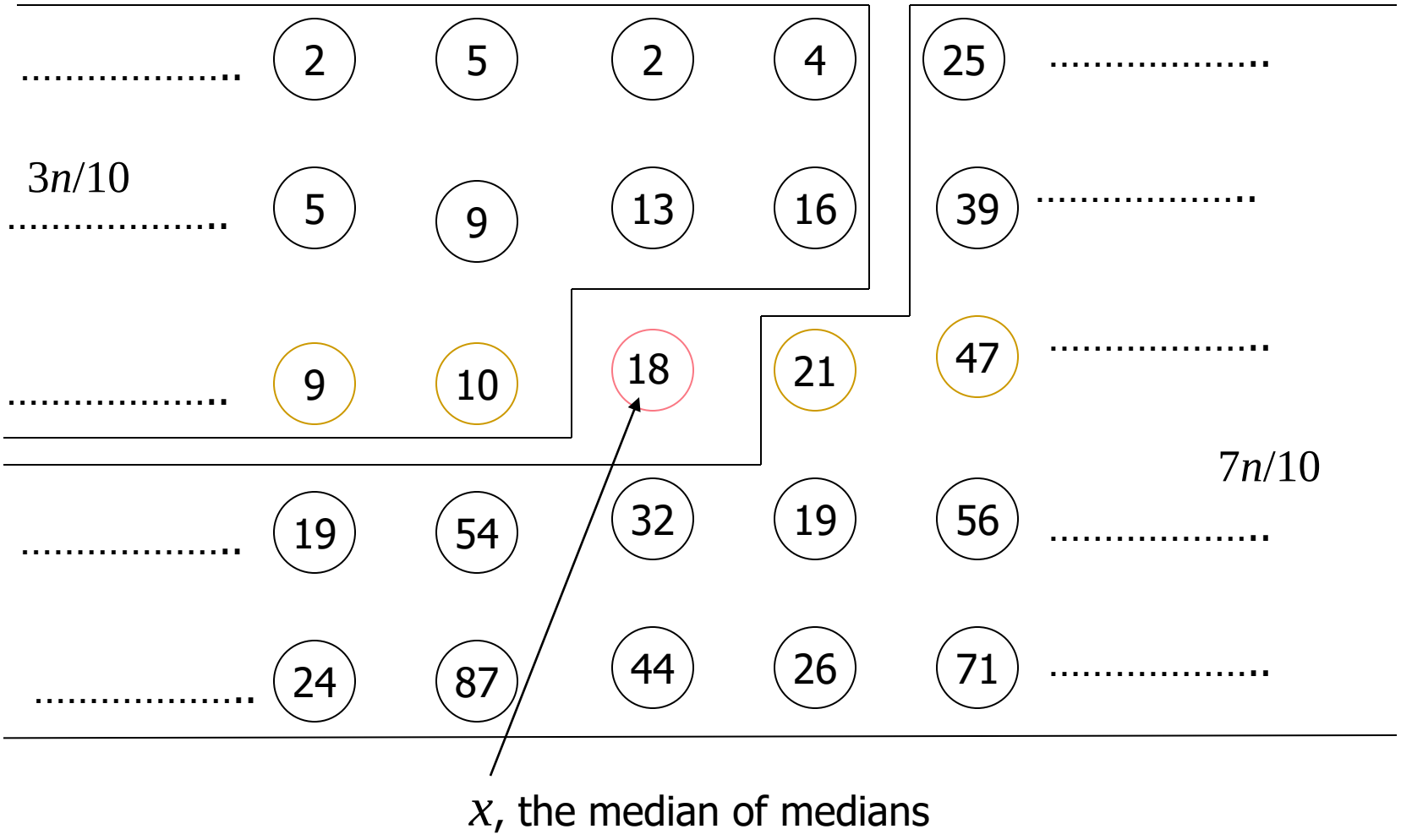
Step1: Group numbers in sets of 5 (Vertically)

.....	2	54	44	4	25	.....
.....	5	5	32	18	39	.....
.....	9	87	21	26	47	.....
.....	19	9	13	16	56	.....
.....	24	10	2	19	71	.....

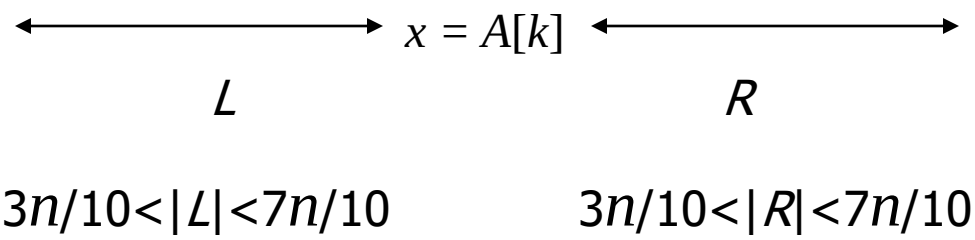
Step2: Find Median of each group



Step3: Find the MedianOfMedians



Step4: Partition original data around the MOM



Step5: If $i = k$, then return $A[k]$ as median else

If $i < k$ -1, then return SELECT(A, p, k-1, i) else

return SELECT(A, k, r, i - k)

Time Analysis

Step	***Task***	***Complexity***
1	Group into sets of 5	$O(n)$
2	Find Median of each group	$O(1)$
3	Find MOM	$T(n/5)$
4	Partition around MOM	$O(n)$
5	Condition	$T(7n/10)$ {Worst Case}

Time Analysis

Thus, we get the recurrence

$$T(n) \leq \begin{cases} O(1) & \text{if } n < 140\,, \\ T(\lceil n/5 \rceil) + T(7n/10 + 6) + O(n) & \text{if } n \geq 140\,. \end{cases}$$

- ***Inductive hypothesis:*** $T(n) \leq cn$ for some constant c and all $n > 0$.
- Substitute the inductive hypothesis in the right-hand side of the recurrence:

$$\begin{aligned} T(n) &\leq c\lceil n/5 \rceil + c(7n/10 + 6) + an \\ &\leq cn/5 + c + 7cn/10 + 6c + an \\ &= 9cn/10 + 7c + an \\ &= cn + (-cn/10 + 7c + an)\,. \end{aligned}$$

- This last quantity is $\leq cn$ if

$$\begin{aligned} -cn/10 + 7c + an &\leq 0 \\ cn/10 - 7c &\geq an \\ cn - 70c &\geq 10an \\ c(n - 70) &\geq 10an \\ c &\geq 10a(n/(n - 70))\,. \end{aligned}$$

Time Analysis

$$T(n) \le \begin{cases} O(1) & \text{if } n < 140\,, \\ T(\lceil n/5 \rceil) + T(7n/10 + 6) + O(n) & \text{if } n \ge 140\,. \end{cases}$$

$$c \;\ge\; 10a(n/(n-70))\,.$$

- Because we assumed that $n \ge 140$, we have $n/(n-70) \le 2$.
- Thus, $20a \ge 10a(n/(n-70))$, so choosing $c \ge 20a$ gives $c \ge 10a(n/(n-70))$, which in turn gives us the condition we need to show that $T(n) \le cn$.
- We conclude that $T(n) = O(n)$, so that SELECT runs in linear time in all cases.
- Why 140? We could have used any integer strictly greater than 70.
  - Observe that for $n > 70$, the fraction $n/(n-70)$ decreases as n increases.
  - We picked $n \ge 140$ so that the fraction would be ≤ 2, which is an easy constant to work with.

Why sets of 5?

Assuming sets of 3,

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

$T(1) = 1$

Assume $T(n) \leq Cn$ (For it to be linear time)

L.H.S = Cn

R.H.S = $C_1 n + Cn/3 + 2Cn/3 = (C_1 + C)\, n$

Hence, L.H.S = R.H.S if $\boldsymbol{C_1 = 0}$

Thus this is invalid assumption. Hence we do not use groups of 3!

Assuming sets of 7,

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

$T(1) = 1$

Assume $T(n) \leq Cn$ (For it to be linear time)

L.H.S = Cn

R.H.S = $C_1 n + Cn/7 + 5Cn/7 = (C_1 + 6C/7)\, n$

Hence, L.H.S = R.H.S if $\boldsymbol{C = 7C_1}$

However, constant factor of O(n) term increases to a sub-optimal value as size of set increases!

In fact, any odd group size > 5 works in linear time.

Dynamic Sets

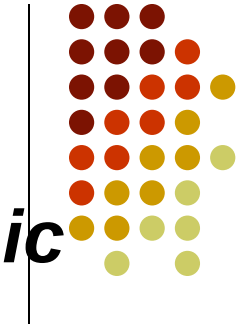
- ***Sets,*** fundamental elements
- If they can be manipulated by algorithms we call them ***Dynamic Sets***

  Example: Customer Database, Yellow Pages
- ***Maintenance of Dynamic Sets:***

  Given a Set "S" and a value "x", some typical operations that we perform on dynamic sets:

Query	Modifying
✓Find (S, x) ✓Minimum (S) ✓Maximum (S) ✓Successor (S, x) ✓Predessor (S,x)	✓Insert (X, S) ✓Delete (X,S)

Dynamic Sets



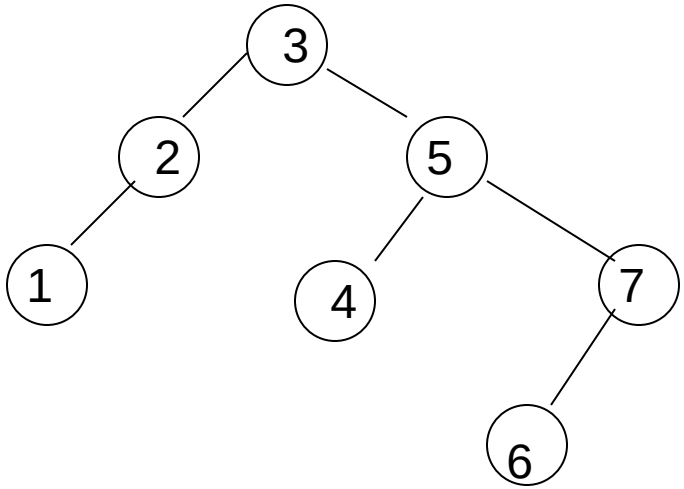
Asymptotic running times of some Dynamic Set operations:

Operation	Unordered Array	Sorted Array	Unordered List	Goal
Insert (X, S)	O (1)	O (n)	O (1)	O (logn)
Delete (X,S)	O (n)	O (n)	O (n)	O (logn)
Find (X, S)	O (n)	O (n)	O (n)	O (logn)

Binary Search Tree

A Binary Search Tree is a data structure such that at any node, the values in the left subtree are smaller than the node and the values in the right subtree are greater than the node.

Example: Given S = {3, 2, 5, 4, 7, 6, 1}, the Binary Search Tree is:

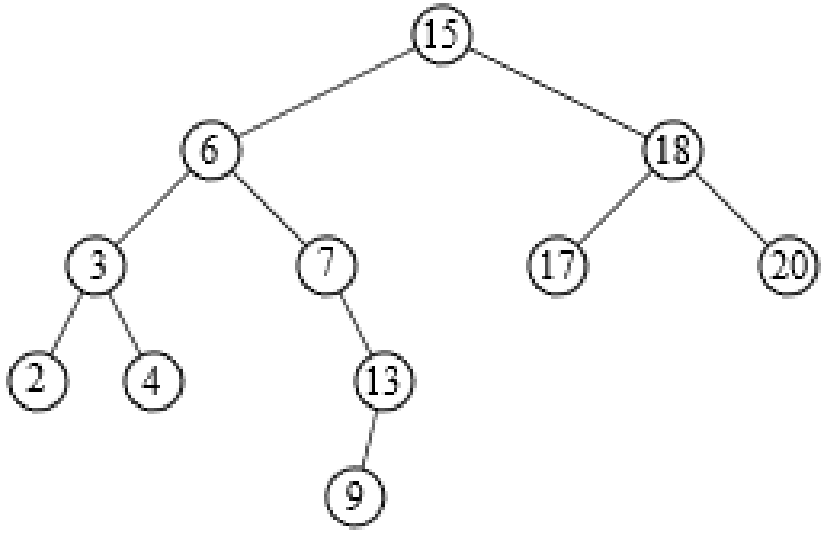


Binary Search Tree

Basic operations performed on the Binary Search Tree are of the order of the path from root to leaf or $O(h)$

- For complete binary tree with n nodes:
  Worst case, $\Theta(\lg n)$
- For linear chain of n nodes:
  Worst case, $\Theta(n)$

BST, Some Properties



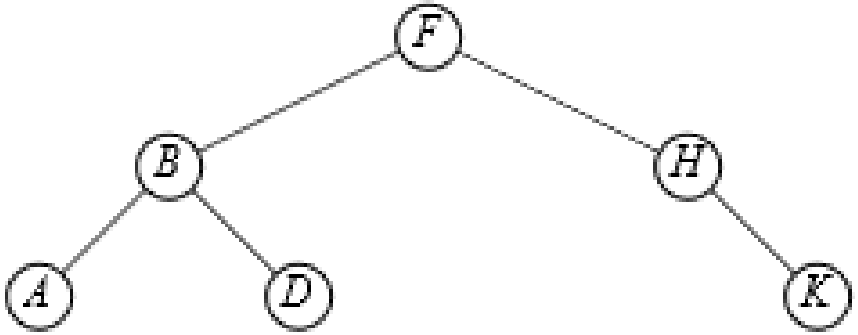
- $T.root$ points to the root of tree T.
- Each node contains the attributes
  - *key* (and possibly other satellite data).
  - *left*: points to left child.
  - *right*: points to right child.
  - p: points to parent. $T.root.p = \text{NIL}$.
- Stored keys must satisfy the ***binary-search-tree property***.
  - If y is in left subtree of x, then $y.key \leq x.key$.
  - If y is in right subtree of x, then $y.key \geq x.key$.

Sorting using Inorder Tree Walk

- BST can be traced/sorted using:

```
INORDER-TREE-WALK(x)
  if x ≠ NIL
      INORDER-TREE-WALK(x.left)
      print key[x]
      INORDER-TREE-WALK(x.right)
```

- Example:
  Executing Inorder Tree Walk on the tree below gives,
  ABDFHK



Sorting using Inorder Tree Walk

```
INORDER-TREE-WALK(x)
  if x ≠ NIL
      INORDER-TREE-WALK(x.left)
      print key[x]
      INORDER-TREE-WALK(x.right)
```

- Time
  Intuitively, the walk takes $\Theta(n)$ time for a tree with n nodes, because we visit and print each node once
- Also solving, $T(n) = T(k) + T(n\text{-}k\text{-}1) + O(1)$, provides $\Theta(n)$. Proof is in the book

Note:

- ✓ B Tree and B^+ Tree are optimizations of Binary Search Tree to give guaranteed $O(\lg n)$ performance

Home Work:

Exercises

- ✓ 3.1-2, 3.1-3, 3.1-4, 3.2-4
- ✓ 4-4-1, 4.4-9
- ✓ 7.1-1, 7.1-3
- ✓ 9.2-3, 9.3-1, 9.3-5
- ✓ 12.1-1, 12.1-2