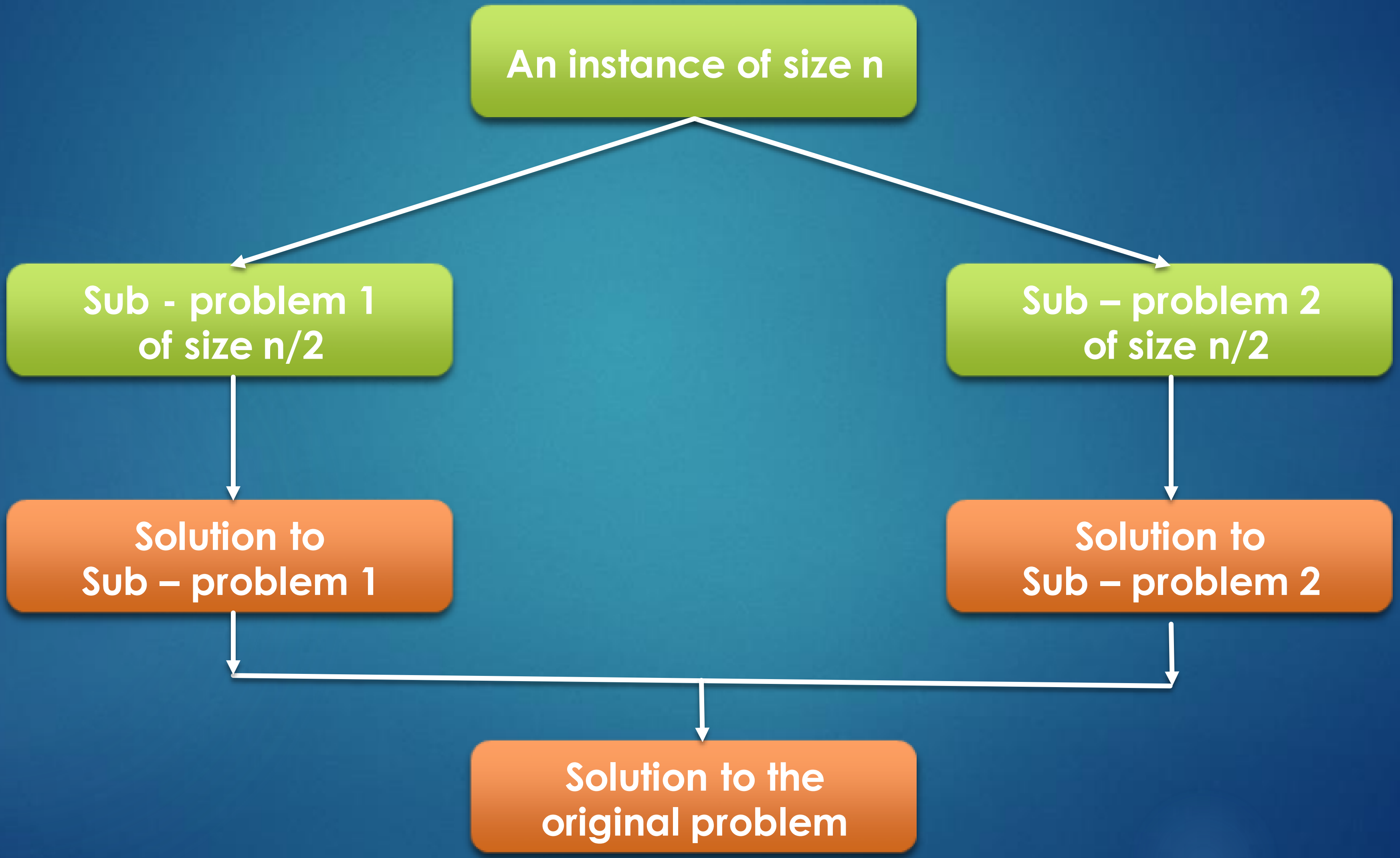
Divide and Conquer

UNIT 2

Divide and Conquer – The IDEA

- Divide and Conquer is one of the most well – known algorithm design strategies.
- The principle underlying Divide and Conquer strategy can be stated as follows:
  1. Divide the given instance of the problem into two or more smaller instances.
  2. Solve the smaller instances recursively.
  3. Combine the solutions of the smaller instances and obtain the solution for the original instance.

Divide and Conquer – The IDEA



General Divide and Conquer Recurrence

- In the most typical cases of Divide and Conquer, a problem's instance of size ***n*** can be divided into ***b*** instances of size ***n/b***, with ***a*** of them needing to be solved.
- Here a and b are constants; a >= 1 and b >= 1.
- Assuming that size n is a power of b, we get the following recurrence for the running time:

$$T(n) = a * T(n/b) + f(n)$$

- f(n) is a function that accounts for the time spent on dividing the problem and combining the solutions.

Master Theorem

- For the recurrence:

$$T(n) = a * T(n/b) + f(n)$$

- If $f(n) \in \Theta(n^d)$, where $d >= 0$ in the recurrence relation, then:

If $a < b^d$, $T(n) \in \Theta(n^d)$

If $a = b^d$, $T(n) \in \Theta(n^d \log n)$

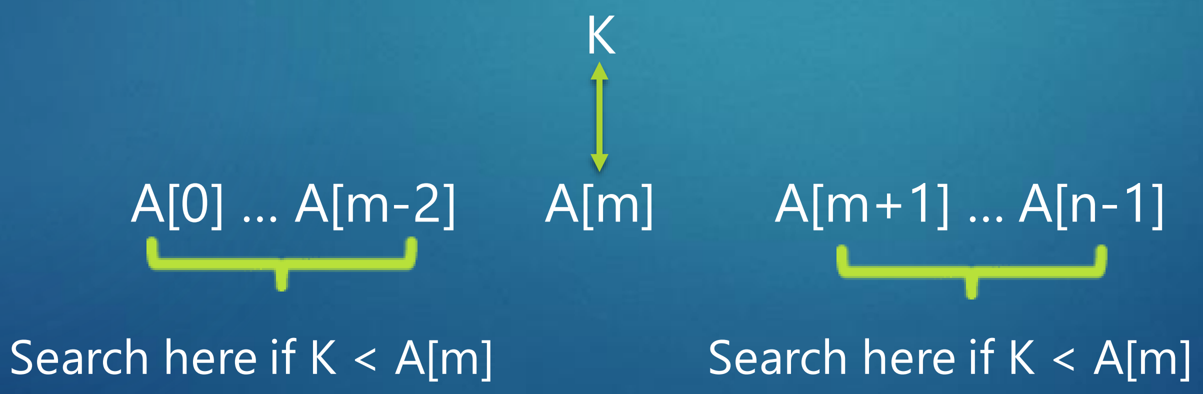
If $a > b^d$, $T(n) \in \Theta(n^{\log_b a})$

- Analogous results hold for O and Ω as well!

Binary Search

Binary Search - IDEA

- Binary Search is a remarkably efficient algorithm for searching in a sorted array.
- It works by comparing the search key K with the array's middle element A[m].
- If they match, the algorithm stops.
- Otherwise, the same operation is repeated recursively for the first half of the array if K < A[m] and for the second half if K > A[m].



Binary Search - Algorithm

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

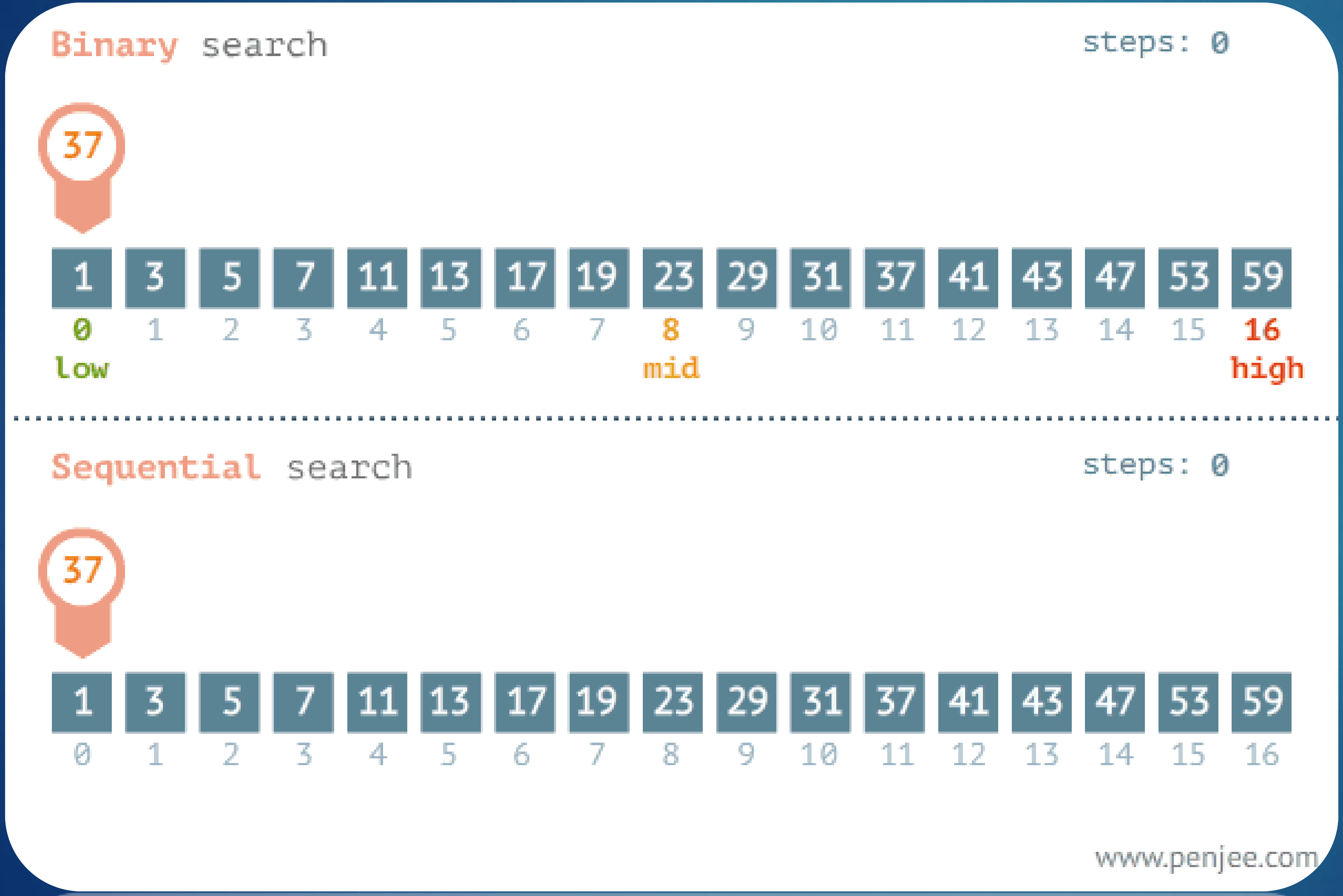
//Implements nonrecursive binary search
//Input: An array $A[0..n-1]$ sorted in ascending order and
//   a search key K
//Output: An index of the array's element that is equal to K
//   or -1 if there is no such element
$l \leftarrow 0;\ r \leftarrow n-1$
while $l \le r$ **do**
    $m \leftarrow \lfloor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

Binary Search - Example

Search Key K = 70

index	0	1	2	3	4	5	6	7	8	9	10	11	12
value	3	14	27	31	39	42	55	70	74	81	85	93	98
iteration 1	l						m						r
iteration 2								l		m			r
iteration 3								l,m	r				

Binary Search Vs Linear Search



Binary Search – Analysis: WORST CASE

The basic operation is the comparison of the search key with an element of the array.

The number of comparisons made are given by the following recurrence:

$$C_{worst}(n) = C_{worst}(\lfloor n/2 \rfloor) + 1 \quad \text{for } n > 1, \; C_{worst}(1) = 1.$$

- For the initial condition $C_{worst}(1) = 1$, we obtain:

$$C_{worst}(2^k) = k + 1 = \log_2 n + 1.$$

- For any arbitrary positive integer, n:

$$C_{worst}(n) = \lfloor \log_2 n \rfloor + 1$$

Binary Search – Analysis: AVERAGE CASE

$$C_{avg}(n) \approx \log_2 n.$$

Merge Sort

Merge Sort - IDEA

- Split array A[0..n-1] into about equal halves and make copies of each half in arrays B and C
- Sort arrays B and C recursively
- Merge sorted arrays B and C into array A as follows:
  - Repeat the following until no elements remain in one of the arrays:
    - ✓ compare the first elements in the remaining unprocessed portions of the arrays
    - ✓ copy the smaller of the two into A, while incrementing the index indicating the unprocessed portion of that array
  - Once all elements in one of the arrays are processed, copy the remaining unprocessed elements from the other array into A.

Merge Sort - Algorithm

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

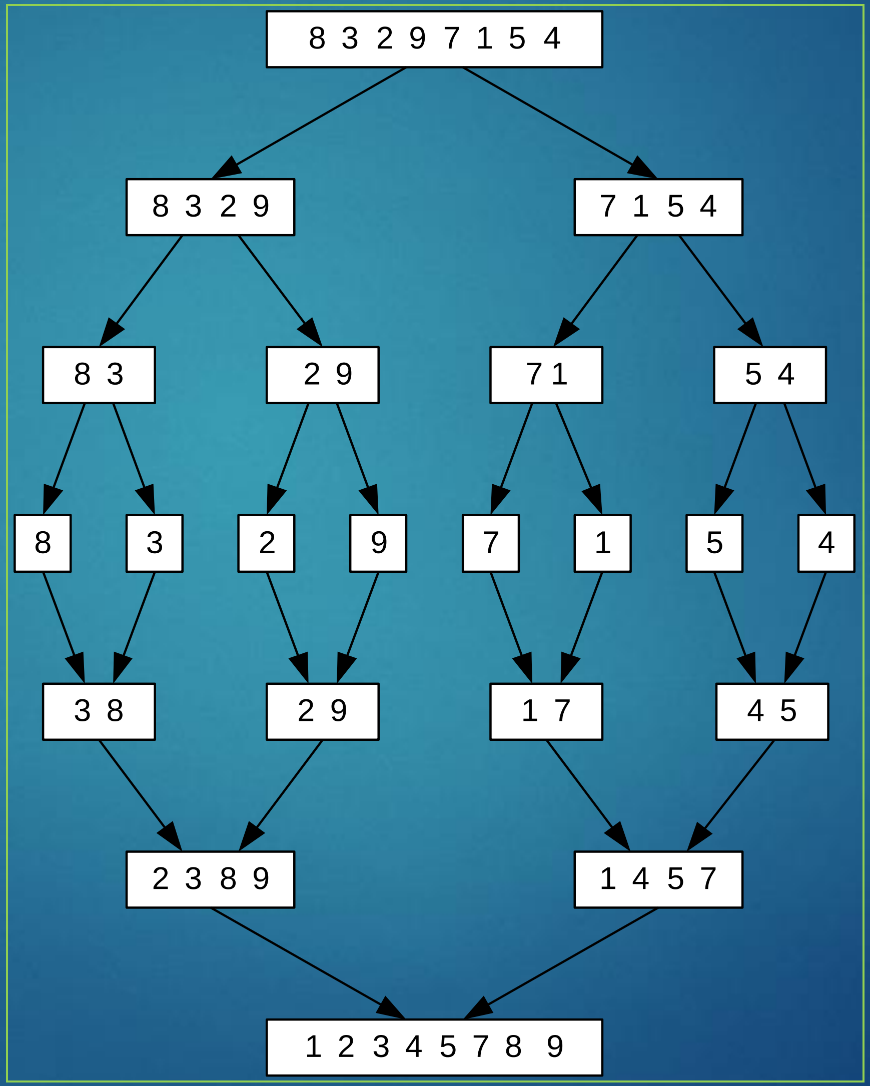
//Sorts array $A[0..n-1]$ by recursive mergesort
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Array $A[0..n-1]$ sorted in nondecreasing order
if $n > 1$
    copy $A[0..\lfloor n/2 \rfloor - 1]$ to $B[0..\lfloor n/2 \rfloor - 1]$
    copy $A[\lfloor n/2 \rfloor..n-1]$ to $C[0..\lceil n/2 \rceil - 1]$
    Mergesort$(B[0..\lfloor n/2 \rfloor - 1])$
    Mergesort$(C[0..\lceil n/2 \rceil - 1])$
    Merge(B, C, A)

Merge Sort - Algorithm

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 $B[i] \le C[j]$
    $A[k] \leftarrow B[i];\ i \leftarrow i+1$
  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 $A[k..p+q-1]$

Merge Sort - Example



Merge Sort - Analysis

- Assuming for simplicity that n is a power of 2, the recurrence relation for the number of key comparisons C(n) is:

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

- The number of key comparisons performed during the merging stage in the worst case is:

$$C_{merge}(n) = n - 1$$

- Using the above equation:

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

- Applying Master Theorem to the above equation:

$$C_{worst}(n) \in \Theta(n \log n)$$

Quick Sort

Quick Sort - IDEA

- Select a *pivot* (partitioning element) – here, the first element
- Rearrange the list so that all the elements in the first *s* positions are smaller than or equal to the pivot and all the elements in the remaining *n-s* positions are larger than or equal to the pivot (see next slide for an algorithm)

- Exchange the pivot with the last element in the first (i.e., ≤) subarray — the pivot is now in its final position
- Sort the two subarrays recursively

Quick Sort - Algorithm

ALGORITHM *Quicksort*$(A[l..r])$

//Sorts a subarray by quicksort

//Input: A subarray $A[l..r]$ of $A[0..n-1]$, defined by its left and right indices

// l and r

//Output: Subarray $A[l..r]$ sorted in nondecreasing order

if $l < r$

$s \leftarrow Partition(A[l..r])$ //s is a split position

$Quicksort(A[l..s-1])$

$Quicksort(A[s+1..r])$

Quick Sort - Algorithm

Algorithm *Partition*$(A[l..r])$
//Partitions a subarray by using its first element as a pivot
//Input: A subarray $A[l..r]$ of $A[0..n-1]$, defined by its left and right
// indices l and r $(l < r)$
//Output: A partition of $A[l..r]$, with the split position returned as
// this function's value
$p \leftarrow A[l]$
$i \leftarrow l$; $j \leftarrow r + 1$
repeat
  repeat $i \leftarrow i + 1$ **until** $A[i] \geq p$
  repeat $j \leftarrow j - 1$ **until** $A[j] \cdot p$
  swap$(A[i], A[j])$
until $i \geq j$
swap$(A[i], A[j])$ //undo last swap when $i \geq j$
swap$(A[l], A[j])$
return j

Quick Sort - Example

5 3 1 9 8 2 4 7

2 3 1 4 5 8 9 7

1 2 3 4 5 7 8 9

1 2 3 4 5 7 8 9

1 2 3 4 5 7 8 9

1 2 3 4 5 7 8 9

Quick Sort – Analysis: BEST CASE

- The number of comparisons in the best case satisfies the recurrence:

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

- According to Master Theorem,

$$C_{best}(n) \in \Theta(n \log_2 n)$$

Quick Sort – Analysis: WORST CASE

- The number of comparisons in the best case satisfies the recurrence:

$$C_{worst}(n) = (n+1) + n + \cdots + 3 = \frac{(n+1)(n+2)}{2} - 3 \in \Theta(n^2).$$

Quick Sort – Analysis: AVERAGE CASE

- Let $C_{avg}(n)$ be the number of key comparisons made by Quick Sort on a randomly ordered array of size n.

$$C_{avg}(n) = \frac{1}{n}\sum_{s=0}^{n-1}[(n+1) + C_{avg}(s) + C_{avg}(n-1-s)] \quad \text{for } n > 1,$$

$$C_{avg}(0) = 0, \quad C_{avg}(1) = 0.$$

- The solution for the above recurrence is:

$$C_{avg}(n) \approx 2n \ln n \approx 1.38 n \log_2 n.$$

Multiplication of Large Integers

Multiplication of large integers - IDEA

- Let the two numbers being multiplied be a and b.
- a and b are n – digit integers, where n is a positive even number.
- Let the first half of a's digits be a_1 and second half be a_0.
- Similarly, let the first half of b's digits be b_1 and second half be b_0.
- In these notations, $a = a_1a_0$ implies $a = a_1*10^{n/2} + a_0$ and $b = b_1b_0$ implies $b = b_1*10^{n/2} + b^0$.

Multiplication of large integers - IDEA

$$\begin{aligned}
c = a * b &= (a_1 10^{n/2} + a_0) * (b_1 10^{n/2} + b_0) \\
&= (a_1 * b_1)10^n + (a_1 * b_0 + a_0 * b_1)10^{n/2} + (a_0 * b_0) \\
&= c_2 10^n + c_1 10^{n/2} + c_0,
\end{aligned}$$

where

$c_2 = a_1 * b_1$ is the product of their first halves,
$c_0 = a_0 * b_0$ is the product of their second halves,
$c_1 = (a_1 + a_0) * (b_1 + b_0) - (c_2 + c_0)$ is the product of the sum of the a's halves and the sum of the b's halves minus the sum of c_2 and c_0.

Multiplication of large integers - Analysis

- $M(n) = 3M(n/2)$ for $n > 1$, $M(1) = 1$
- Solving it by backward substitutions for $n = 2^k$ yields:

$$M(2^k) = 3M(2^{k-1}) = 3[3M(2^{k-2})] = 3^2 M(2^{k-2})$$
$$= \cdots = 3^i M(2^{k-i}) = \cdots = 3^k M(2^{k-k}) = 3^k.$$

- Since $k = \log_2 n$: $M(n) = 3^{\log_2 n} = n^{\log_2 3} = n^{1.585}$
- The number of additions is given by:

$$A(n) = 3A(n/2) + cn \text{ for } n > 1,\ A(1) = 1$$

$$A(n) \text{ belongs to } \Theta(n^{\log_2 3})$$

Example

$2135 * 4014$

$= (21*10^2 + 35) * (40*10^2 + 14)$

$= (21*40)*10^4 + c1*10^2 + 35*14$

where $c1 = (21+35)*(40+14) - 21*40 - 35*14$, and

$21*40 = (2*10 + 1) * (4*10 + 0)$

$= (2*4)*10^2 + c2*10 + 1*0$

where $c2 = (2+1)*(4+0) - 2*4 - 1*0$, etc.

Strassen’s Matrix Multiplication

Strassen's Matrix Multiplication

- This algorithm was published by V Strassen in 1969.
- The principal insight of the algorithm lies in the discovery that we can find product of two 2 – by – 2 matrices A and B with seven multiplications as opposed to the eight required by the Brute – Force algorithm.
- This is accomplished by the following formulae:

Strassen's Matrix Multiplication

$$\begin{bmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} * \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix}$$

$$= \begin{bmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{bmatrix},$$

Strassen's Matrix Multiplication

$$m_1 = (a_{00} + a_{11}) * (b_{00} + b_{11})$$
$$m_2 = (a_{10} + a_{11}) * b_{00}$$
$$m_3 = a_{00} * (b_{01} - b_{11})$$
$$m_4 = a_{11} * (b_{10} - b_{00})$$
$$m_5 = (a_{00} + a_{01}) * b_{11}$$
$$m_6 = (a_{10} - a_{00}) * (b_{00} + b_{01})$$
$$m_7 = (a_{01} - a_{11}) * (b_{10} + b_{11}).$$

Strassen's Matrix Multiplication – General Formula

- For any two matrices A and B of size n – by – n, we can divide A, B and the product C as follows:

$$\left[\begin{array}{c|c} C_{00} & C_{01} \\ \hline C_{10} & C_{11} \end{array}\right] = \left[\begin{array}{c|c} A_{00} & A_{01} \\ \hline A_{10} & A_{11} \end{array}\right] * \left[\begin{array}{c|c} B_{00} & B_{01} \\ \hline B_{10} & B_{11} \end{array}\right]$$

- The sub – matrices can be treated as numbers to get the correct product.

Strassen's Matrix Multiplication – Analysis

- If M(n) is the number of multiplications made by Strassen's algorithm in multiplying two matrices n – by – n, we get the following recurrence relation for it:

$$M(n) = 7M(n/2) \quad \text{for } n > 1, \qquad M(1) = 1.$$

Since $n = 2^k$,

$$M(2^k) = 7M(2^{k-1}) = 7[7M(2^{k-2})] = 7^2 M(2^{k-2}) = \cdots$$
$$= 7^i M(2^{k-i}) \cdots = 7^k M(2^{k-k}) = 7^k.$$

Since $k = \log_2 n$,

$$M(n) = 7^{\log_2 n} = n^{\log_2 7} \approx n^{2.807},$$

Strassen's Matrix Multiplication – Analysis

- The number of additions are given by the following recurrence:

$$A(n) = 7A(n/2) + 18(n/2)^2 \quad \text{for } n > 1, \qquad A(1) = 0.$$

- According to Master's Theorem, A(n) belongs to $\Theta(n^{\log_2 7})$

Binary Tree Traversals and Other Properties

What is a Binary Tree?

- A *binary tree* **T** is defined as a finite set of nodes that is either empty or consists of a root and two disjoint binary trees $\mathbf{T_L}$ and $\mathbf{T_R}$ called as the left and right subtree of the root.

- The definition itself divides the Binary Tree into two smaller structures and hence many problems concerning the binary trees can be solved using the Divide – And – Conquer technique.

- The binary tree is a Divide – And – Conquer ready structure ☺

Height of a Binary Tree

- Height of a binary tree = Length of the longest path from root to leaf.

ALGORITHM *Height*(T)

//Computes recursively the height of a binary tree
//Input: A binary tree T
//Output: The height of T
if $T = \varnothing$ **return** -1
else return $\max\{Height(T_L), Height(T_R)\} + 1$

Height of a Binary Tree – Analysis

- The measure of input's size is the number of nodes in the given binary tree. Let us represent this number as $n(T)$.

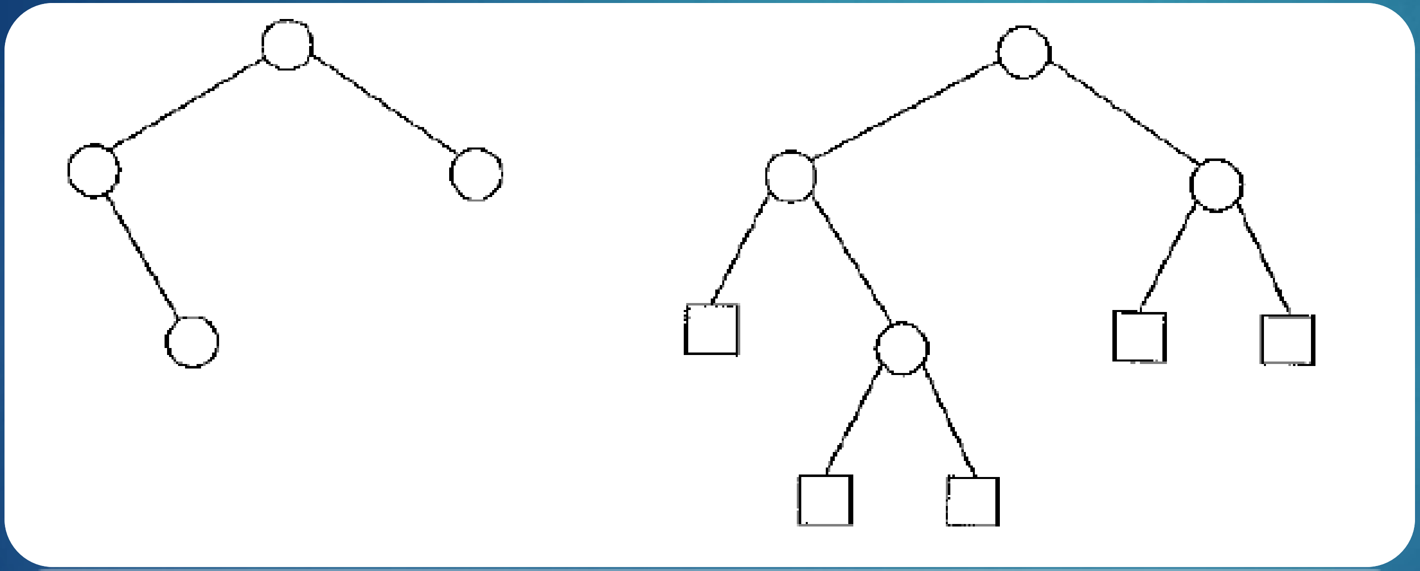
- Basic Operation: Addition

- The recurrence relation is setup as follows:

$$A(n(T)) = A(n(TL)) + A(n(TR)) + 1, \quad \text{for } n(T) > 0$$
$$A(0) = 0$$

Height of a Binary Tree – Analysis

- In the analysis of tree algorithms, the tree is extended by replacing empty subtrees by special nodes called external nodes.



Height of a Binary Tree – Analysis

- x – Number of external nodes
- n – Number of internal nodes

$$x = n + 1$$

- The number of comparisons to check whether a tree is empty or not:

$$C(n) = n + x = 2n + 1$$

- The number of additions is:

$$A(n) = n$$

Binary Tree Traversals

- The three classic traversals for a binary tree are inorder, preorder and postorder traversals.

- In the preorder traversal, the root is visited before the left and right subtrees are visited (in that order).
- In the inorder traversal, the root is visited after visiting its left subtree but before visiting the right subtree.
- In the postorder traversal, the root is visited after visiting the left and right subtrees (in that order).

Binary Tree Traversals

Algorithm Inorder(T)

if $T \neq \varnothing$

  Inorder(T_{left})

  print(root of T)

  Inorder(T_{right})

Algorithm Preorder(T)

if $T \neq \varnothing$

  print(root of T)

  Preorder(T_{left})

  Preorder(T_{right})

Algorithm Postorder(T)

if $T \neq \varnothing$

  Postorder(T_{left})

  Postorder(T_{right})

  print(root of T)