MODULE-II

Contents

LECTURE 9:

1. EXHAUSTIVE SEARCH .

For discrete problems in which no efficient solution method is known, it might be necessary to test each possibility sequentially in order to determine if it is the solution. Such exhaustive examination of all possibilities is known as exhaustive search, complete search or direct search. Exhaustive search is simply a brute force approach to combinatorial problems (Minimization or maximization of optimization problems and constraint satisfaction problems). Reason to choose brute-force / exhaustive search approach as an important algorithm design strategy

1. First, unlike some of the other strategies, brute force is applicable to a very wide variety of problems. In fact, it seems to be the only general approach for which it is more difficult to point out problems it cannot tackle.

2. Second, 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.

3. Third, 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.

4. Fourth, even if too inefficient in general, a brute-force algorithm can still be useful for solving small-size instances of a problem. Exhaustive Search is applied to the important problems like

- Traveling Salesman Problem
- Knapsack Problem
- Assignment Problem.

1.1 TRAVELLING SALESMAN PROBLEM

The traveling salesman problem (TSP) is one of the combinatorial problems.

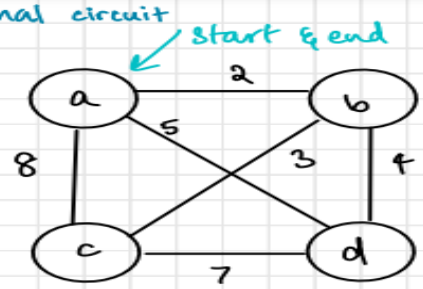
The problem asks to find the shortest tour through a given set of n cities that visits each city exactly once before returning to the city where it started.

The problem can be conveniently modelled by a weighted graph, with the graph's vertices representing the cities and the edge weights specifying the distances.

Then the problem can be stated as the problem of finding the shortest Hamiltonian circuit of the graph. (A Hamiltonian circuit is defined as a cycle that passes through all the vertices of the graph exactly once).

A Hamiltonian circuit can also be defined as a sequence of n + 1 adjacent vertices vi_0, vi_1, . . . , vi_{n-1}, vi_0, where the first vertex of the sequence is the same as the last one and all the other n − 1 vertices are distinct. All circuits start and end at one particular vertex. Following Excercise presents a small instance of the problem and its solution by this method.

Q: Find optimal circuit



3! = 6 permutations

$a \to b \to c \to d \to a = 2+3+7+5 = 17$

$a \to b \to d \to c \to a = 2+4+7+8 = 21$

$a \to c \to b \to d \to a = 8+3+4+5 = 20$

$a \to c \to d \to b \to a = 8+7+4+2 = 21$

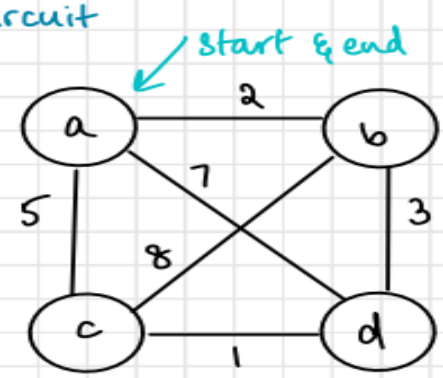
$a \to d \to b \to c \to a = 5+4+3+8 = 20$

$a \to d \to c \to b \to a = 5+7+3+2 = 17$

Optimal paths:

$a \to b \to c \to d \to a$
$a \to d \to c \to b \to a$

Q: Find circuit



$a \to b \to c \to d \to a = 2+8+1+7 = 18$

$a \to b \to d \to c \to a = 2+3+1+5 = 11$

$a \to c \to b \to d \to a = 5+8+3+7 = 23$

$a \to c \to d \to b \to a = 5+1+3+2 = 11$

$a \to d \to b \to c \to a = 7+3+8+5 = 23$

$a \to d \to c \to b \to a = 7+1+8+2 = 18$

Optimal solution

$a \to b \to d \to c \to a$
$a \to c \to d \to b \to a$

Time efficiency

• We can get all the tours by generating all the permutations of n − 1 intermediate cities from a particular city.. i.e. (n - 1)!

• Consider two intermediate vertices, say, b and c, and then only permutations in which b precedes c. (This trick implicitly defines a tour's direction.)

• An inspection of above exercises n reveals three pairs of tours that differ only by their direction. Hence, we could cut the number of vertex permutations by half because cycle total lengths in both directions are same.

• The total number of permutations needed is still $\frac{1}{2}$ (n − 1)!, which makes the exhaustive search approach impractical for large n. It is useful for very small values of n.

Review Questions:

1. Explain Travelling Salesman Problem.
2. Write the total number of permutations needed in TSP problem.
3. What is the Time complexity of TSP.

LECTURE 10:

1.2 KNAPSACK PROBLEM:

Given *n* items of known weights $w_1, w_2, \ldots, w_n$ and values $v_1, v_2, \ldots, v_n$ and a knapsack of capacity *W*, find the most valuable subset of the items that fit into the knapsack.

Real time examples:

 A Thief who wants to steal the most valuable loot that fits into his knapsack,

 A transport plane that has to deliver the most valuable set of items to a remote location without exceeding the plane's capacity.

The exhaustive-search approach to this problem leads to generating all the subsets of the set of *n* items given, computing the total weight of each subset in order to identify feasible subsets (i.e.,the ones with the total weight not exceeding the knapsack capacity), and finding a subset of the largest value among them.

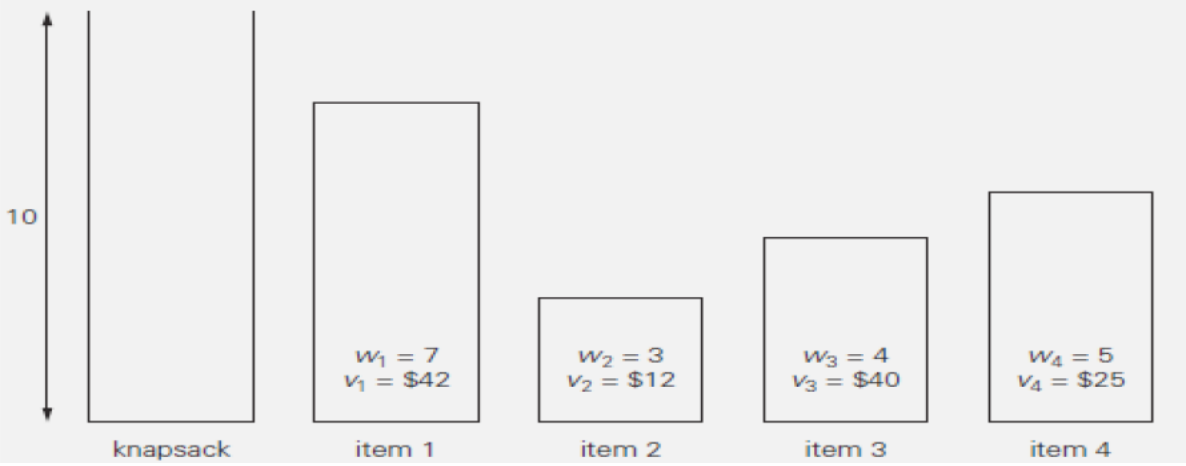


Figure 1.2.1 Instance of the knapsack problem

Q: Knapsack capacity W = 16

Item	Weight	Value
1	2	20
2	5	30
3	10	50
4	5	10

	Subset	Weight	Value
1	{}	0	0
2	{1}	2	20
3	{2}	5	30
4	{3}	10	50
5	{4}	5	10
6	{1,2}	7	50
7	{1,3}	12	70
8	{1,4}	7	30
9	{2,3}	15	80
10	{2,4}	10	40
11	{3,4}	15	60
12	{1,2,3}	17	not feasible
13	{1,2,4}	12	60
14	{1,3,4}	17	not feasible
15	{2,3,4}	20	not feasible
16	{1,2,3,4}	22	not feasible

optimal

- Exhaustive search: $\Omega(2^n)$

Fig 1.2.2 knapsack problem's solution by exhaustive search. The information about the optimal selection

Time efficiency: As given in the example, the solution to the instance of Figure 1.2.1 is given in Figure 1.2.2. Since the *number of subsets of an n-element set is* 2^n, the exhaustive search leads to a $\Omega(2^n)$ algorithm, no matter how efficiently individual subsets are generated.

Note: Exhaustive search of both the traveling salesman and knapsack problems leads to extremely inefficient algorithms on every input. In fact, these two problems are the best-known examples of ***NP-hard problems***. **No polynomial-time** algorithm is known for any *NP*-hard problem. Moreover, most computer scientists believe that such algorithms do not exist. some sophisticated approaches like **backtracking** and **branch-and-bound** enable us to solve some instances but not all instances of these in less than exponential time. Alternatively, we can use one of many **approximation algorithms.**

Review Questions:

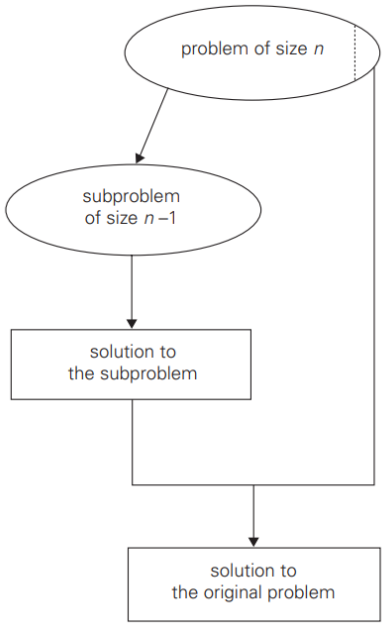
1. What is the objective of the knapsack problem?
2. Define the terms "weight" and "value" as used in the context of the knapsack problem.
3. Explain why the exhaustive search approach for solving the knapsack problem is inefficient.
4. What is the time complexity of the exhaustive search approach for the knapsack problem?
5. Let's consider the Knapsack problem with the following inputs:
   Total capacity Item# 1 2 3 4 5
   7 Kgs Weight (Kg) 3 1 3 4 2
   Value (USD) 2 2 4 5 3
   1. Solve the problem using exhaustive search

LECTURE 11:

2. DECREASE AND CONQUER APPROACH:

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. Once such a relationship is established, it can be exploited either top down or bottom up. The bottom-up variation is usually implemented iteratively, starting with a solution to the smallest instance of the problem; it is called sometimes the incremental approach. There are three major variations of decrease-and-conquer:

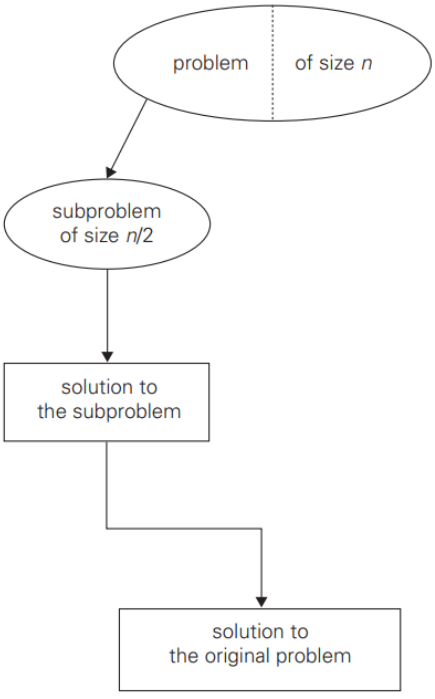
- **Decrease by a constant**



In the decrease-by-a-constant variation, the size of an instance is reduced by the same constant on each iteration of the algorithm. Typically, this constant is equal to one (Figure 4.1), although other constant size reductions do happen occasionally.

- **Decrease by a constant factor**

The decrease-by-a-constant-factor technique suggests reducing a problem instance by the same constant factor on each iteration of the algorithm. In most applications, this constant factor is equal to two.



- **Variable size decrease**

Finally, in the variable-size-decrease variety of decrease-and-conquer, the size-reduction pattern varies from one iteration of an algorithm to another. Euclid's algorithm for computing the greatest common divisor provides a good example of such a situation.

2.1 INSERTION SORT

Insertion sort is an efficient algorithm for sorting arrays. It follows the decrease-by-one technique, exploiting the smaller sorted subarray to insert elements in their correct positions. The algorithm iterates through the array, comparing each element with the sorted subarray from right to left. When an element smaller or equal to the current one is found, it shifts the elements to the right to create space and inserts the element. This process continues until the entire array is sorted

Algorithm:

ALGORITHM *InsertionSort*$(A[0..n-1])$
//Sorts a given array by insertion sort
//Input: An array $A[0..n-1]$ of n orderable elements
//Output: Array $A[0..n-1]$ sorted in nondecreasing order
for $i \leftarrow 1$ **to** $n-1$ **do**
    $v \leftarrow A[i]$
    $j \leftarrow i-1$
    while $j \geq 0$ **and** $A[j] > v$ **do**
        $A[j+1] \leftarrow A[j]$
        $j \leftarrow j-1$
    $A[j+1] \leftarrow v$

Example Problem:

89 |	**45**	68	90	29	34	17
45	89 |	**68**	90	29	34	17
45	68	89 |	**90**	29	34	17
45	68	89	90 |	**29**	34	17
29	45	68	89	90 |	**34**	17
29	34	45	68	89	90 |	**17**
17	29	34	45	68	89	90

Analysis:

In the best case, the comparison A[j]>v is executed only once on every iteration of the outer loop. It happens if and only if A[i − 1]≤ A[i] for every i = 1, . . . , n − 1, i.e., if the input array is already sorted in nondecreasing order.

$$C_{best}(n) = \sum_{i=1}^{n-1} 1 = n - 1 \in \Theta(n).$$

The number of key comparisons in this algorithm obviously depends on the nature of the input. In the worst case, A[j]> v is executed the largest number of times, i.e., for every j = i − 1, . . . , 0. Since v = A[i], it happens if and only if A[j]>A[i] for j = i − 1, . . . , 0. Thus, for the worst-case input, we get A[0]>A[1] (for i = 1), A[1]>A[2] (for i = 2), . . . , A[n − 2]>A[n − 1] (for i = n − 1). In other words, the worst-case input is an array of strictly decreasing values. The number of key comparisons for such an input is

$$C_{worst}(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1 = \sum_{i=1}^{n-1} i = \frac{(n-1)n}{2} \in \Theta(n^2).$$

It shows that on randomly ordered arrays, insertion sort makes on average half as many comparisons as on decreasing arrays, i.e.,

$$C_{avg}(n) \approx \frac{n^2}{4} \in \Theta(n^2).$$

Review Questions:

1. What is the fundamental idea behind the decrease-and-conquer technique?
2. Describe the three major variations of decrease-and-conquer with examples.
3. Explain how insertion sort works and what technique it employs.
4. What is the key difference between decrease by a constant and decrease by a constant factor in the decrease-and-conquer approach?

LECTURE 12:

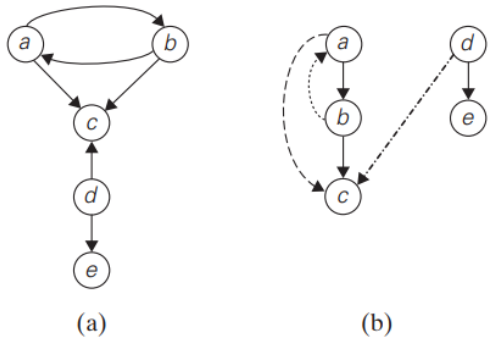
2.2 TOPOLOGICAL SORTING.

A directed graph, or digraph for short, is a graph with directions specified for all its edges. The **adjacency matrix** and **adjacency lists** are still two principal means of representing a digraph. There are only two notable differences between undirected and directed graphs in representing them:

(1) the adjacency matrix of a directed graph does not have to be symmetric;

(2) an edge in a directed graph has just one (not two) corresponding node in the digraph's adjacency lists.

Depth-first search and breadth-first search are principal traversal algorithms for traversing digraphs as well, but the structure of corresponding forests can be more complex than for undirected graphs. Thus, even for the simple example of Figure, the depth-first search forest exhibits all four types of edges possible in a DFS forest of a directed graph: *tree edges* (ab, bc, de), *back edges* (ba) from vertices to their ancestors, *forward edges* (ac) from vertices to their descendants in the tree other than their children, and *cross edges* (dc), which are none of the aforementioned types.

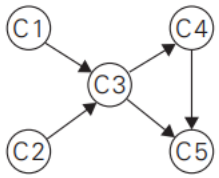


(a) Digraph. (b) DFS forest of the digraph for the DFS traversal started at a.

Note that a back edge in a DFS forest of a directed graph can connect a vertex to its parent. Whether or not it is the case, the presence of a back edge indicates that the digraph has a directed cycle. A directed cycle in a digraph is a sequence of three or more of its vertices that starts and ends with the same vertex and in which every vertex is connected to its immediate predecessor by an edge directed from the predecessor to the successor. For example, a, b, a is a directed cycle in the digraph in Figure. Conversely, if a DFS forest of a digraph has no back edges, the digraph is a DAG, an acronym for **directed acyclic graph**.

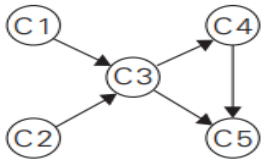
- **Topological sorting example:**

Consider a set of five required courses {C1, C2, C3, C4, C5} a part-time student has to take in some degree program. The courses can be taken in any order as long as the following course prerequisites are met: C1 and C2 have no prerequisites, C3 requires C1 and C2, C4 requires C3, and C5 requires C3 and C4. The student can take only one course per term. In which order should the student take the courses? The situation can be modeled by a digraph in which vertices represent courses and directed edges indicate prerequisite requirements.



Digraph representing the prerequisite structure of five courses.

In terms of this digraph, the question is whether we can list its vertices in such an order that for every edge in the graph, the vertex where the edge starts is listed before the vertex where the edge ends. (Can you find such an ordering of this digraph's vertices?) This problem is called topological sorting. It can be posed for an arbitrary digraph, but it is easy to see that the problem cannot have a solution if a digraph has a directed cycle. Thus, for topological sorting to be possible, a digraph in question must be a dag. It turns out that being a dag is not only necessary but also sufficient for topological sorting to be possible; i.e., if a digraph has no directed cycles, the topological sorting problem for it has a solution. Moreover, there are two efficient algorithms that both verify whether a digraph is a dag and, if it is, produce an ordering of vertices that solves the topological sorting problem.



(a)

$C5_1$
$C4_2$
$C3_3$
$C1_4$ $C2_5$

(b)

The popping-off order:
C5, C4, C3, C1, C2
The topologically sorted list:
C2 C1→C3→C4→C5

(c)

(a) Digraph for which the topological sorting problem needs to be solved. (b) DFS traversal stack with the subscript numbers indicating the popping-off order. (c) Solution to the problem.

The first algorithm is a simple application of depth-first search: perform a DFS traversal and note the order in which vertices become dead-ends (i.e., popped off the traversal stack). Reversing this order yields a solution to the topological sorting problem, provided, of course, no back edge has been encountered during the traversal. If a back edge has been encountered, the digraph is not a dag, and topological sorting of its vertices is impossible. Figure illustrates an application of this algorithm to the digraph in above Figure. Note that in Figure-c, we have drawn the edges of the digraph, and they all point from left to right as the problem's statement requires. It is a convenient way to check visually the correctness of a solution to an instance of the topological sorting problem.

The second algorithm is based on a direct implementation of the decrease-(by one)-and-conquer technique: repeatedly, identify in a remaining digraph a source, which is a vertex with no incoming edges, and delete it along with all the edges outgoing from it. The order in which the vertices are deleted yields a solution to the topological sorting problem. The application of this algorithm to the same digraph representing the five courses is given in Figure below.

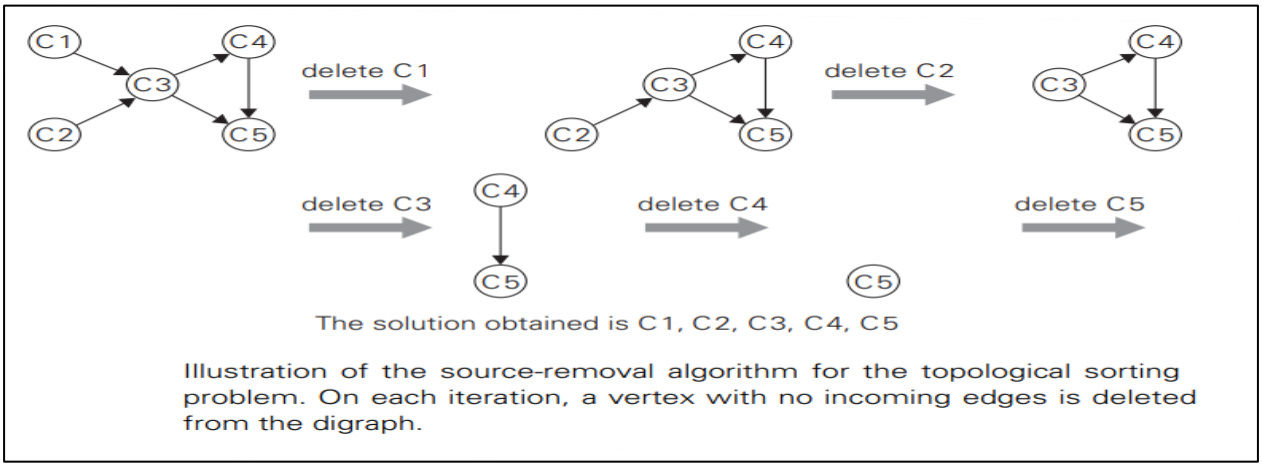


Illustration of the source-removal algorithm for the topological sorting problem. On each iteration, a vertex with no incoming edges is deleted from the digraph.

Note that the solution obtained by the source-removal algorithm is different from the one obtained by the DFS-based algorithm. Both of them are correct, of course; the topological sorting problem may have several alternative solutions.

Review Questions:

1. What is a directed cycle in a digraph, and how does its presence relate to the concept of a DAG?
2. Using the example of course prerequisites, explain how topological sorting can be applied to solve a scheduling problem.
3. Describe the two efficient algorithms used for verifying if a digraph is a DAG and performing topological sorting.
4. Compare and contrast the DFS-based algorithm and the source-removal algorithm for performing topological sorting. What are their main differences and similarities?

LECTURE 13:

3.DIVIDE and CONQUER

A divide and conquer algorithm works by recursively breaking down a problem into two or more sub-problems of the same (or related) type (divide), until these become simple enough to be solved directly (conquer). Divide-and-conquer algorithms work according to the following general plan:

1. A problem is divided into several subproblems of the same type, ideally of about equal size.

2. The subproblems are solved (typically recursively, though sometimes a different algorithm is employed, especially when subproblems become small enough).

3. If necessary, the solutions to the subproblems are combined to get a solution to the original problem.

The divide-and-conquer technique as shown in Figure 2.9, which depicts the case of dividing a problem into two smaller subproblems, then the subproblems solved separately. Finally solution to the original problem is done by combining the solutions of subproblems.

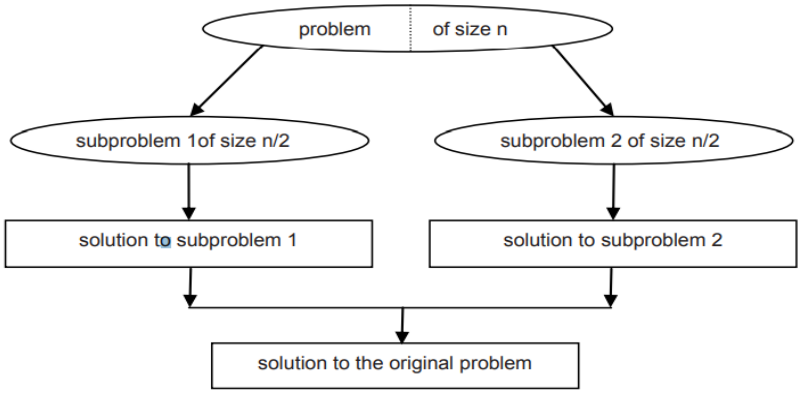


FIGURE 2.9 Divide-and-conquer technique.

Divide and conquer methodology can be easily applied on the following problem.

1. Merge sort
2. Quick sort
3. Binary search

In the most typical case of divide-and-conquer a problem's instance of size n is divided into two instances of size $n/2$. More generally, an 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; $a \geq 1$ and $b > 1$.) Assuming that size n is a power of b to simplify our analysis, we get the following recurrence for the running time $T(n)$:

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

where $f(n)$ 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. (For the sum example above, $a = b = 2$ and $f(n) = 1$.) Recurrence (5.1) is called the ***general divide-and-conquer recurrence***.

The efficiency analysis of many divide-and-conquer algorithms is greatly simplified by the following theorem.

Master Theorem If $f(n) \in \Theta(n^d)$ where $d \geq 0$ in recurrence (5.1), then

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

Analogous results hold for the O and Ω notations, too.

For example, the recurrence for the number of additions $A(n)$ made by the divide-and-conquer sum-computation algorithm (see above) on inputs of size $n = 2^k$ is

$$A(n) = 2A(n/2) + 1.$$

Thus, for this example, $a = 2$, $b = 2$, and $d = 0$; hence, since $a > b^d$,

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

Review Questions:

1. Explain the basic principle of divide and conquer algorithms.
2. What are the three main steps involved in the divide-and-conquer technique?
3. Describe the general form of a divide-and-conquer recurrence relation.
4. What are the key components of the recurrence relation T(n) = aT(n/b) + f(n)?

3.1MERGE SORT

- It uses DIVIDE AND CONQUER method
- Mergesort is a perfect example of a successful application of the divide-and conquer technique.
- It sorts a given array A[0…n − 1] by dividing it into two halves A[0….(n/2)−1] and A[(n/2)…..n−1], sorting each of them recursively, and then merging the two smaller sorted arrays into a single sorted one.

ALGORITHM *Mergesort(A[0..n − 1])*

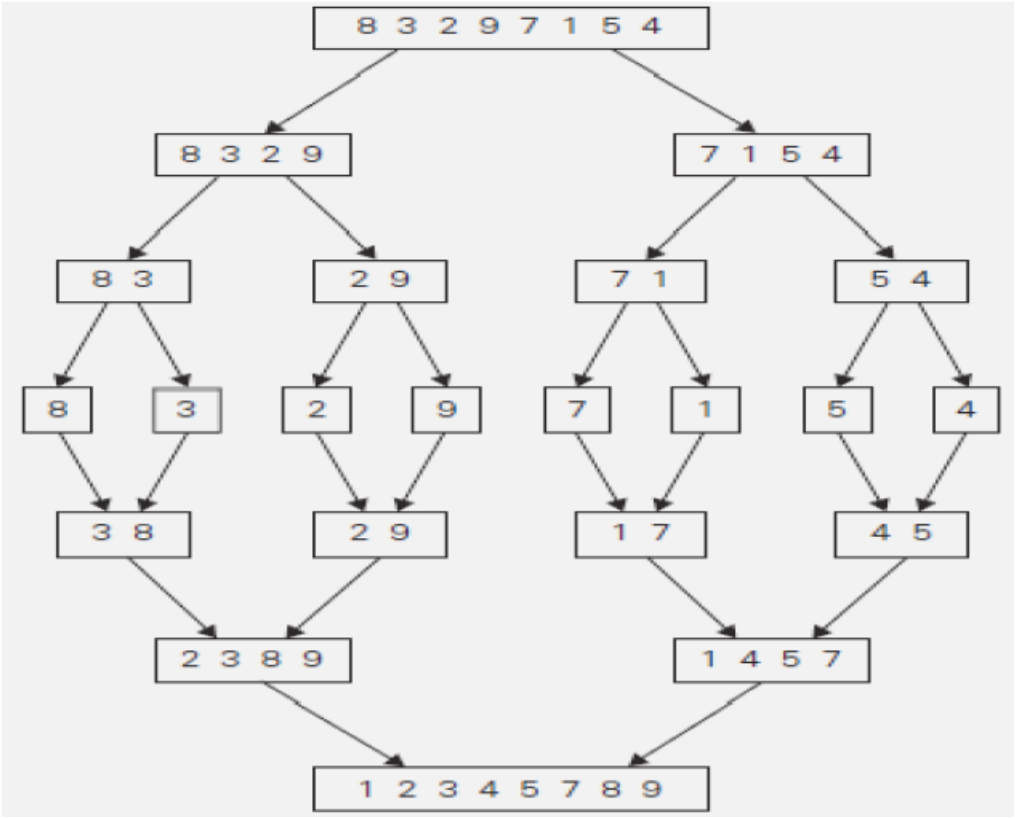
//Sorts array $A[0..n-1]$ by recursive mergesort
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Array $A[0..n-1]$ sorted in nondecreasing order
if $n > 1$
copy $A[0..\lfloor n/2 \rfloor - 1]$ to $B[0..\lfloor n/2 \rfloor - 1]$
copy $A[\lfloor n/2 \rfloor..n-1]$ to $C[0..\lceil n/2 \rceil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

Merge(B, C, A) //see below

The ***merging*** of two sorted arrays can be done as follows. Two pointers (array indices) are initialized to point to the first elements of the arrays being merged. The elements pointed to are compared, and the smaller of them is added to a new array being constructed; after that, the index of the smaller element is incremented to point to its immediate successor in the array it was copied from. This operation is repeated until one of the two given arrays is exhausted, and then the remaining elements of the other array are copied to the end of the new array.

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 $A[k..p+q-1]$

The operation of the algorithm on the list 8, 3, 2, 9, 7, 1, 5, 4 is illustrated in Figure 2.10.



The recurrence relation for the number of key comparisons C(n) is

$C(n) = 2C(n/2) + C_{merge}(n)$ for $n > 1$,

$C(1) = 0$.

In the worst case, $C_{merge}(n) = n - 1$, and we have the recurrence

$C_{worst}(n) = 2C_{worst}(n/2) + n - 1$ for $n > 1$,

$C_{worst}(1) = 0$.

By Master Theorem, $C_{worst}(n) \in \Theta(n \log n)$

the exact solution to the worst-case recurrence for $n = 2^k$

$C_{worst}(n) = n \log_2 n - n + 1$.

For large n, the number of comparisons made by this algorithm in the average case turns out to be about 0.25n less and hence is also in Θ (n log n).

First, the algorithm can be implemented bottom up by merging pairs of the array's elements, then merging the sorted pairs, and so on. This avoids the time and space overhead of using a stack to handle recursive calls. Second, we can divide a list to be sorted in more than two parts, sort each recursively, and then merge them together. This scheme, which is particularly useful for sorting files residing on secondary memory devices, is called multiway mergesort.

Review Questions:

1. **Describe how mergesort utilizes the divide-and-conquer method to sort an array.**
2. **Explain the process of merging two sorted arrays in mergesort.**
3. **What is the recurrence relation used to analyze the number of key comparisons in mergesort?**
4. **How is the worst-case number of key comparisons calculated using the recurrence relation?**
5. **According to the Master Theorem, what is the time complexity (big theta notation) of mergesort in the worst case scenario?**

LECTURE 14:

3.2QUICK SORT

It uses divide and conquer method. Quicksort is a perfect example of a successful application of the divide-and conquer technique.C.A.R. Hoare, the prominent British computer scientist who invented quicksort.

Unlike mergesort, which divides its input elements according to their position in the array, quicksort divides them according to their value.

A partition is an arrangement of the array's elements so that all the elements to the left of some element A[s] are less than or equal to A[s], and all the elements to the right of A[s] are greater than or equal to it:

$$\underbrace{A[0] \dots A[s-1]}_{\text{all are} \le A[s]} \quad A[s] \quad \underbrace{A[s+1] \dots A[n-1]}_{\text{all are} \ge A[s]}$$

Obviously, after a partition is achieved, A[s] will be in its final position in the sorted array, and we can continue sorting the two subarrays to the left and to the right of A[s] independently.

difference with mergesort: there, the division of the problem into two subproblems is immediate and the entire work happens in combining their solutions; here, the entire work happens in the division stage, with no work required to combine the solutions to the subproblems.

In this type of sorting technique, we will be given n elements in the array A, and the first element is made as **"Pivot"** element. The position next to pivot element as i and the last position as j. Then following steps have to be done.

- **Steps:**

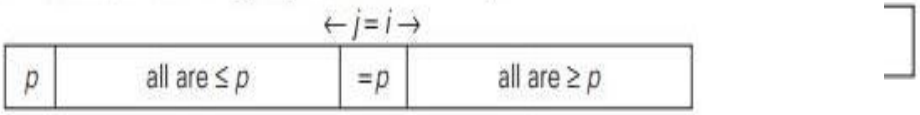
  1. we will now scan the subarray from both ends, comparing the subarray's elements to the pivot.

  2. The left-to-right scan, denoted below by index pointer i, starts with the second element. Since we want elements smaller than the pivot to be in the left part of the subarray, this scan skips over elements that are smaller than the pivot and stops upon encountering the first element greater than or equal to the pivot.

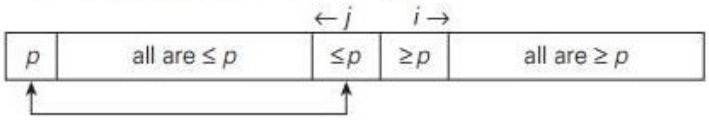
  3. The right-to-left scan, denoted below by index pointer j, starts with the last element of the subarray. Since we want elements larger than the pivot to be in the right part of the subarray, this scan skips over elements that are larger than the pivot and stops on encountering the first element smaller than or equal to the pivot.

  4. After both scans stop, three situations may arise, depending on whether or not the scanning indices have crossed. If scanning indices i and j have not crossed, i.e., i < j, we simply exchange A[i] and A[j] and resume the scans by incrementing i and decrementing j, respectively:

Finally, if the scanning indices stop while pointing to the same element, i.e., $i = j$, the value they are pointing to must be equal to p (why?). Thus, we have the subarray partitioned, with the split position $s = i = j$:



If the scanning indices have crossed over, i.e., $i > j$, we will have partitioned the subarray after exchanging the pivot with $A[j]$:



We can combine the last case with the case of crossed-over indices ($i > j$) by exchanging the pivot with $A[j]$ whenever $i \geq j$.

Here is pseudocode implementing this partitioning procedure.

Here is pseudocode of quicksort: call *Quicksort(A*[0..$n - 1$]*)* where As a partition algorithm use the *HoarePartition*

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

```
//Sorts a subarray by quicksort
//Input: Subarray of array 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 ←Partition(A[l..r])  //s is a split position
    Quicksort(A[l..s − 1])
    Quicksort(A[s + 1..r])
```

ALGORITHM *HoarePartition(A[l..r])*
//Partitions a subarray by Hoare's algorithm, using the first element as a pivot
//Input: Subarray of array $A[0..n-1]$, defined by its left and right indices l and r $(l<r)$
//Output: 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** $A[j] \leq p$
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

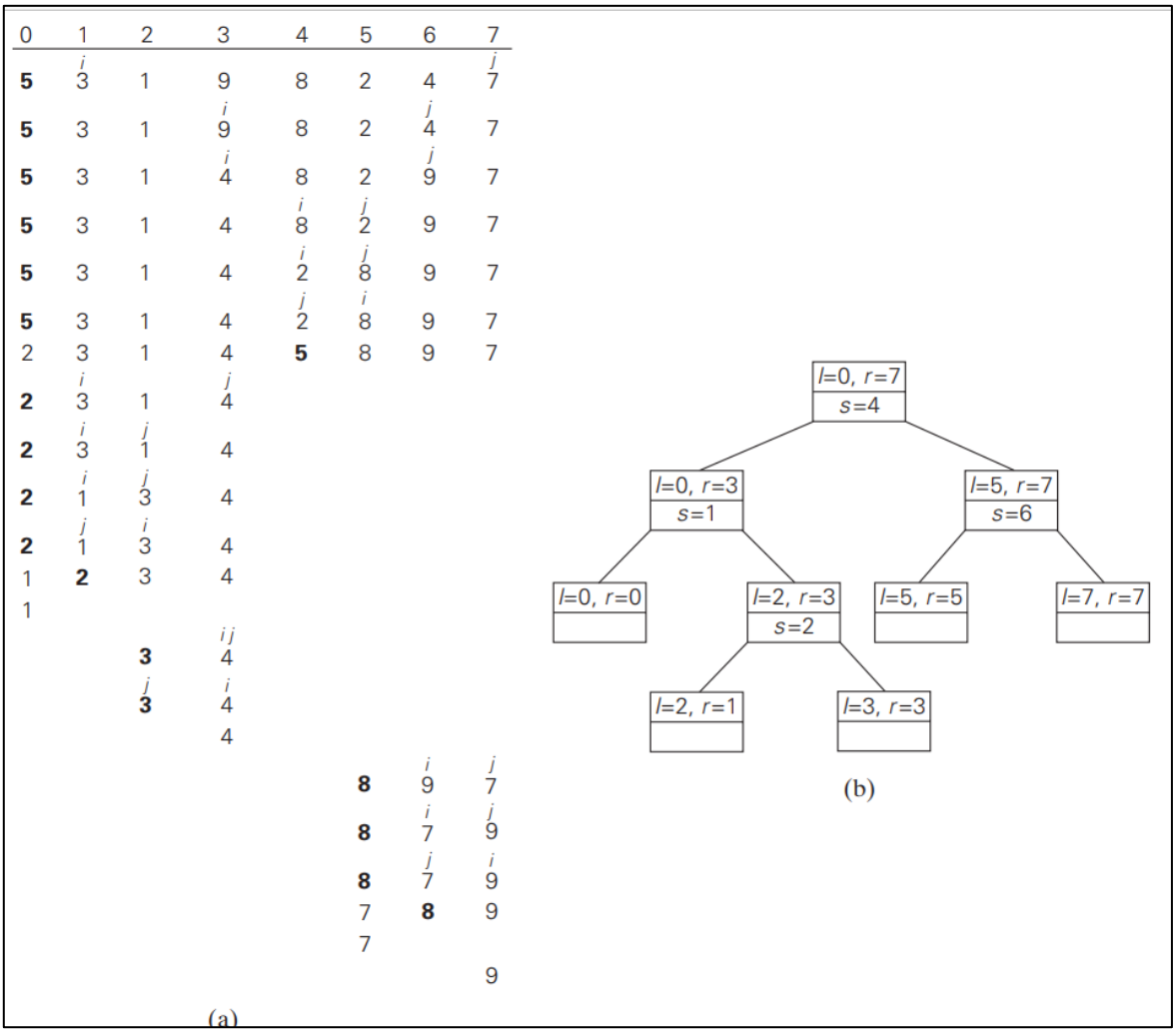


Fig: Example of quicksort operation. (a) Array's transformations with pivots shown in bold. (b) Tree of recursive calls to Quicksort with input values l and r of subarray bounds and split position s of a partition obtained.

Analysis:

The number of key comparisons in the best case satisfies the recurrence

$C_{best}(n) = 2C_{best}(n/2) + n$ for $n > 1$,

$C_{best}(1) = 0$.

By Master Theorem, $C_{best}(n) \in \Theta(n \log_2 n)$; solving it exactly for $n = 2^k$ yields

$C_{best}(n) = n \log_2 n$.

The total number of key comparisonsin worst case made will be equal to

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

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

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

$$C_{avg}(n) \approx 2n \ln n \approx 1.39n \log_2 n.$$

Review Questions

1. Describe the steps involved in the Quicksort algorithm using the divide-and-conquer technique.
2. What is the significance of choosing a "pivot" element in Quicksort?
3. what is the time complexity of Quicksort in the best case scenario?
4. Calculate the total number of key comparisons made in the worst-case scenario for Quicksort. How does this relate to the time complexity?

LECTURE 15:

3.4BINARY TREE TRAVERSALS AND RELATED PROPERTIES:

- Here, we see how the divide-and-conquer technique can be applied to binary trees.
- 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 T_L and T_R called, respectively, the left and right subtree of the root. Binary tree is a special case of an ordered tree .
- Since the definition itself divides a binary tree into two smaller structures of the same type, the left subtree and the right subtree, many problems about binary trees can be solved by applying the divide-and-conquer technique.
- As an example, let us consider a recursive algorithm for computing the height of a binary tree.
- Recall that the height is defined as the length of the longest path from the root to a leaf.

- Hence, it can be computed as the maximum of the heights of the root's left and right subtrees plus 1.

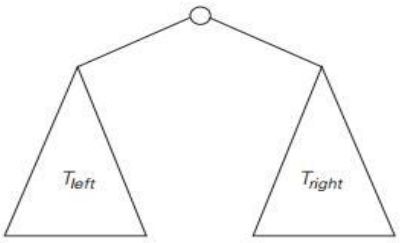


FIGURE 5.4 Standard representation of a binary tree.

Note: that it is convenient to define the height of the empty tree as −1.

Thus, we have the following recursive algorithm.

ALGORITHM *Height(T)*

//Computes recursively the height of a binary tree

//Input: A binary tree *T*

//Output: The height of *T*

if *T* = ∅

return −1

else

return max{*Height*(T_{left}), *Height*(T_{right})} + 1

We measure the problem's instance size by the number of nodes n(T) in a given binary tree T . Obviously, the number of comparisons made to compute the maximum of two numbers and the number of additions A(n(T)) made by the algorithm are the same.

We have the following recurrence relation for A(n(T)):

$A(n(T)) = A(n(T_{left})) + A(n(T_{right})) + 1$ for $n(T) > 0$,

$A(0) = 0$.

The extra nodes (shown by little squares in Figure 5.5) are called ***external***; the original nodes (shown by little circles) are called ***internal***.

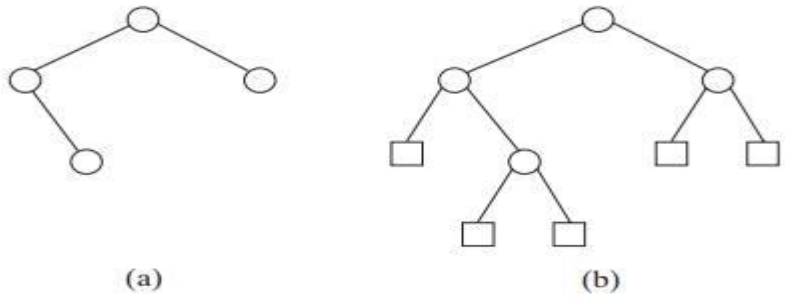


FIGURE 5.5 Binary tree (on the left) and its extension (on the right). Internal nodes are shown as circles; external nodes are shown as squares.

The number of external nodes ***x*** is always 1 more than the number of internal nodes ***n***:

$$x = n + 1. \qquad \textbf{(5.2)}$$

To prove this equality, consider the total number of nodes, both internal and external. Since every node, except the root, is one of the two children of an internal node, we have the equation

$$2n + 1 = x + n,$$

which immediately implies equality (5.2).

Note that equality (5.2) also applies to any nonempty ***full binary tree***, in which, by definition, every node has either zero or two children:

Returning to algorithm *Height*, the number of comparisons to check whether the tree is empty is

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

and the number of additions is

$$A(n) = n.$$

The most important divide-and-conquer algorithms for binary trees are the three classic traversals: preorder, inorder, and postorder. All three traversals visit nodes of a binary tree recursively, i.e., by visiting the tree's root and its left and right subtrees. They differ only by the timing of the root's visit:

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).

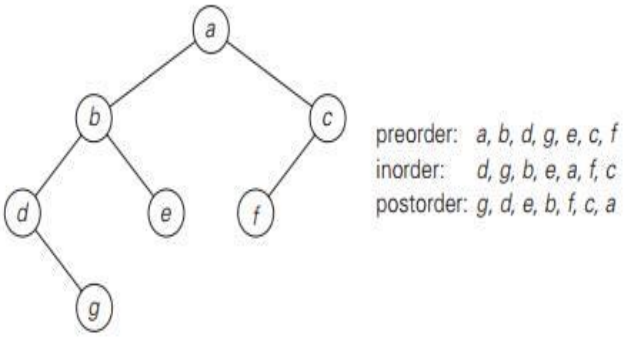


FIGURE 5.6 Binary tree and its traversals.

These traversals are illustrated in Figure 5.6.

As to their efficiency analysis, it is identical to the above analysis of the *Height* algorithm because a recursive call is made for each node of an extended binary tree.

Finally,not all questions about binary trees require traversals of both left and right subtrees. we should note that, For example, the search and insert operations for a binary search tree require processing only one of the two subtrees. Accordingly, we considered them not as applications of divide-and conquer but rather as examples of the variable-size-decrease technique.

Review Questions

1. Define a binary tree and explain the structure of a binary tree including its nodes, root, and subtrees.
2. How does the divide-and-conquer technique apply to solving problems related to binary trees?
3. Describe the recursive algorithm for computing the height of a binary tree. How is the height defined?
4. What is the recurrence relation used to analyze the number of additions in computing the height of a binary tree?

LECTURE 16:

3.3 MULTIPLICATION OF LARGE INTEGERS

Some applications like modern cryptography require manipulation of integers that are over 100 decimal digits long. Since such integers are too long to fit in a single word of a modern computer, they require special treatment. In the conventional pen-and-pencil algorithm for multiplying two n-

digit integers, each of the n digits of the first number is multiplied by each of the n digits of the second number for the total of n^2 digit multiplications. The divide-and-conquer method does the above multiplication in less than n^2 digit multiplications.

Example: $23 * 14 = (2 \cdot 10^1 + 3 \cdot 10^0) * (1 \cdot 10^1 + 4 \cdot 10^0)$

$= (2 * 1)10^2 + (2 * 4 + 3 * 1)10^1 + (3 * 4)10^0$

$= 2 \cdot 10^2 + 11 \cdot 10^1 + 12 \cdot 10^0$

$= 3 \cdot 10^2 + 2 \cdot 10^1 + 2 \cdot 10^0$

$= 322$

The term $(2 * 1 + 3 * 4)$ computed as $2 * 4 + 3 * 1 = (2 + 3) * (1 + 4) - (2 * 1) - (3 * 4)$. Here $(2 * 1)$ and $(3 * 4)$ are already computed used. So only one multiplication only we have to do.

For any pair of two-digit numbers $a = a_1a_0$ and $b = b_1b_0$, their product c can be computed by the formula $c = a * b = c_2 10^2 + c_1 10^1 + c_0$, where

$c_2 = a_1 * b_1$ is the product of their first digits,

$c_0 = a_0 * b_0$ is the product of their second digits,

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

Now we apply this trick to multiplying two n-digit integers a and b where n is a positive even number. Let us divide both numbers in the middle to take advantage of the divide-and conquer technique.

We denote the first half of the a's digits by a_1 and the second half by a_0; for b, the notations are b_1 and b_0, respectively. In these notations, $a = a_1a_0$ implies that $a = a_1 10^{n/2} + a_0$ and $b = b_1b_0$ implies that $b = b_1 10^{n/2} + b_0$. Therefore, taking advantage of the same trick we used for two-digit numbers, we get

$C = a * b$

$= (a_1 10^{n/2} + a_0) * (b_1 10^{n/2} + b_0)$

$= (a_1 * b_1)10^n + (a_1 * b_0 + a_0 * b_1)10^{n/2} + (a_0 * b_0)$

$= c_2 10^n + c_1 10^{n/2} + c_0$, 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,

$c_1 = (a_1 + a_0) * (b_1 + b_0) - (c_2 + c_0)$

If n/2 is even, we can apply the same method for computing the products c_2, c_0, and c_1.

Thus, if n is a power of 2, we have a recursive algorithm for computing the product of two n-digit

integers. In its pure form, the recursion is stopped when n becomes 1. It can also be stopped when we deem n small enough to multiply the numbers of that size directly.

The multiplication of n-digit numbers requires three multiplications of n/2-digit numbers, the recurrence for the number of multiplications M(n) is

$$\mathbf{M(n) = 3M(n/2) \text{ for } n > 1,}$$

$$\mathbf{M(1) = 1.}$$

Solving it by backward substitutions for $n = 2^k$ yields.

$$\mathbf{M(2^k) = 3M(2^{k-1}) = 3[3M(2^{k-2})] = 3^2M(2^{k-2}) = \ldots = 3^iM(2^{k-i}) = \ldots = 3^kM(2^{k-k}) = 3^k} \text{ (Since } \mathbf{k = \log_2 n)}$$

$$\mathbf{M(n) = 3^{\log_2 n} = n^{\log_2 3} \approx n^{1.585}}$$

(On the last step, we took advantage of the following property of logarithms: $a^{\log_b c} = c^{\log_b a}$.) Let A(n) be the number of digit additions and subtractions executed by the above algorithm in multiplying two n-digit decimal integers. Besides 3A(n/2) of these operations needed to compute the three products of n/2-digit numbers, the above formulas require five additions and one subtraction. Hence, we have the recurrence

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

$$\mathbf{A(1) = 1}$$

By using Master Theorem, we obtain $\mathbf{A(n) \in \Theta(n^{\log_2 3})}$,

which means that the total number of additions and subtractions have the same asymptotic order of growth as the number of multiplications.

Example: For instance: a = 2345, b = 6137, i.e., n=4.

Then $C = a * b = (23*10^2+45)*(61*10^2+37)$

$$C = a * b = (a_1 10^{n/2} + a_0) * (b_1 10^{n/2} + b_0)$$
$$= (a_1 * b_1)10^n + (a_1 * b_0 + a_0 * b_1)10^{n/2} + (a_0 * b_0)$$
$$= (23 * 61)10^4 + (23 * 37 + 45 * 61)10^2 + (45 * 37)$$
$$= 1403 \cdot 10^4 + 3596 \cdot 10^2 + 1665$$
$$= 14391265$$

3.4 STRASSEN'S MATRIX MULTIPLICATION

The Strassen's Matrix Multiplication find the product C of two 2 × 2 matrices A and B with just seven multiplications as opposed to the eight required by the brute-force algorithm.

$$\begin{bmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} * \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix}$$
$$= \begin{bmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{bmatrix},$$

where

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

Thus, to multiply two 2 × 2 matrices, Strassen's algorithm makes 7 multiplications and 18 additions/subtractions, whereas the brute-force algorithm requires 8 multiplications and 4 additions. These numbers should not lead us to multiplying 2 × 2 matrices by Strassen's algorithm. Its importance stems from its asymptotic superiority as matrix order n goes to infinity. Let A and B be two n × n matrices where n is a power of 2. (If n is not a power of 2, matrices can be padded with rows and columns of zeros.) We can divide A, B, and their product C into four n/2 × n/2 submatrices each 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 value C00 can be computed either as A00 * B00 + A01 * B10 or as M1 + M4 − M5 + M7 where M1, M4, M5, and M7 are found by Strassen's formulas, with the numbers replaced by the corresponding submatrices. The seven products of n/2 × n/2 matrices are computed recursively by Strassen's matrix multiplication algorithm.

The asymptotic efficiency of Strassen's matrix multiplication algorithm If M(n) is the number of multiplications made by Strassen's algorithm in multiplying two n×n matrices, where n is a power of 2, The recurrence relation is $M(n) = 7M(n/2)$ for $n > 1$, $M(1)=1$.

Since $n = 2^k$, $M(2^k) = 7M(2^{k-1}) = 7[7M(2^{k-2})] = 7^2M(2^{k-2})$=…….= $7^iM(2^{k-i}) = \ldots\ldots$ = $7^kM(2^{k-k}) = 7^kM(2^0) = 7^kM(1) = 7^k(1)$ (Since M(1)=1) $M(2^k) = 7^k$. Since $k = \log_2 n$,

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

which is smaller than n 3 required by the brute-force algorithm. Since this savings in the number of multiplications was achieved at the expense of making extra additions, we must check the number of additions A(n) made by Strassen's algorithm. To multiply two matrices of order n>1, the algorithm needs to multiply seven matrices of order n/2 and make 18 additions/subtractions of matrices of size n/2; when n = 1, no additions are made since two numbers are simply multiplied. These observations yield the following recurrence relation:

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

$$A(1) = 0.$$

By closed-form solution to this recurrence and the Master Theorem, $A(n) \in \Theta(n^{\log_2 7})$. which is a better efficiency class than $\Theta(n^3)$ of the brute-force method.

Example: Multiply the following two matrices by Strassen's matrix multiplication algorithm.

$$A = \begin{bmatrix} 1 & 0 & 2 & 1 \\ 4 & 1 & 1 & 0 \\ 0 & 1 & 3 & 0 \\ 5 & 0 & 2 & 1 \end{bmatrix} \qquad B = \begin{bmatrix} 0 & 1 & 0 & 1 \\ 2 & 1 & 0 & 4 \\ 2 & 0 & 1 & 1 \\ 1 & 3 & 5 & 0 \end{bmatrix}$$

Answer:

$$C = \begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix} = \begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} \times \begin{bmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{bmatrix}$$

Where $A_{00} = \begin{bmatrix} 1 & 0 \\ 4 & 1 \end{bmatrix}$ $A_{01} = \begin{bmatrix} 2 & 1 \\ 1 & 0 \end{bmatrix}$ $A_{10} = \begin{bmatrix} 0 & 1 \\ 5 & 0 \end{bmatrix}$ $A_{11} = \begin{bmatrix} 3 & 0 \\ 2 & 1 \end{bmatrix}$

$B_{00} = \begin{bmatrix} 0 & 1 \\ 2 & 1 \end{bmatrix}$ $B_{01} = \begin{bmatrix} 0 & 1 \\ 0 & 4 \end{bmatrix}$ $B_{10} = \begin{bmatrix} 2 & 0 \\ 1 & 3 \end{bmatrix}$ $B_{11} = \begin{bmatrix} 1 & 1 \\ 5 & 0 \end{bmatrix}$

$$M_1 = (A_{00}+A_{11})*(B_{00}+B_{11}) = \left(\begin{bmatrix} 1 & 0 \\ 4 & 1 \end{bmatrix} + \begin{bmatrix} 3 & 0 \\ 2 & 1 \end{bmatrix}\right) * \left(\begin{bmatrix} 0 & 1 \\ 2 & 1 \end{bmatrix} + \begin{bmatrix} 1 & 1 \\ 5 & 0 \end{bmatrix}\right) = \begin{bmatrix} 4 & 0 \\ 6 & 2 \end{bmatrix} * \begin{bmatrix} 1 & 2 \\ 7 & 1 \end{bmatrix} = \begin{bmatrix} 4 & 8 \\ 20 & 14 \end{bmatrix}$$

Similarly apply Strassen's matrix multiplication algorithm to find the following.

$M_2 = \begin{bmatrix} 2 & 4 \\ 2 & 8 \end{bmatrix}$, $M_3 = \begin{bmatrix} -1 & 0 \\ -9 & 4 \end{bmatrix}$, $M_4 = \begin{bmatrix} 6 & -3 \\ 3 & 0 \end{bmatrix}$, $M_5 = \begin{bmatrix} 8 & 3 \\ 10 & 5 \end{bmatrix}$, $M_6 = \begin{bmatrix} 2 & -3 \\ -2 & -3 \end{bmatrix}$, $M_7 = \begin{bmatrix} 3 & 2 \\ -9 & -4 \end{bmatrix}$

$C_{00} = \begin{bmatrix} 5 & 4 \\ 4 & 5 \end{bmatrix}$, $C_{01} = \begin{bmatrix} -7 & 3 \\ 1 & 9 \end{bmatrix}$, $C_{10} = \begin{bmatrix} 8 & 1 \\ 5 & 8 \end{bmatrix}$, $C_{11} = \begin{bmatrix} 3 & 7 \\ 7 & 7 \end{bmatrix}$

$$C = \begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix} = \begin{bmatrix} 5 & 4 & 7 & 3 \\ 4 & 5 & 1 & 9 \\ 8 & 1 & 3 & 7 \\ 5 & 8 & 7 & 7 \end{bmatrix}$$

Review Questions:

1. How does divide-and-conquer optimize the multiplication of large integers compared to conventional methods?
2. Explain the recursive algorithm used to multiply large integers using divide-and-conquer.

3. Describe Strassen's Matrix Multiplication algorithm and how it reduces the number of multiplications compared to the brute-force method.
4. Compare the asymptotic efficiency of Strassen's matrix multiplication with the brute-force method for large matrix sizes n

QUESTION BANK- MODULE II

1. Explain Divide And Conquer Method?
2. Explain Merge Sort with suitable example.
3. Discuss Quick Sort
4. Explain Strassen's Algorithm
5. Explain in detail about Travelling Salesman Problem using exhaustive search.
6. Explain in detail about knapsack problem.
7. Apply the Quick sort to the list E , X , A , M , P , L , E .
8. Compute 2011 * 1130 using divide and conquer algorithm.
9. Describe exhaustive search in detail 5. Explain in detail quick sorting method.
10. Provide a complete analysis of quick sort with example.
11. Explain in detail merge sort. Illustrate the algorithm with a numeric example. Provide complete analysis of the same.
12. Explain Topological Sorting?
13. Explain Binary Tree Travesals and Related Properties?
14. Apply strassen's algorithm to compute .

$$\begin{vmatrix} 1 & 2 & 1 & 1 \\ 0 & 3 & 2 & 4 \\ 0 & 1 & 1 & 1 \\ 5 & 0 & 1 & 0 \end{vmatrix} * \begin{Vmatrix} 2 & 1 & 0 & 2 \\ 1 & 2 & 1 & 1 \\ 0 & 3 & 2 & 1 \\ 4 & 0 & 0 & 4 \end{Vmatrix}$$

15. Sort the list E,X,A,M,P,L,E in alphabetical order using MergeSort. Also discuss the efficiency.
16. Apply Quick sort to sort the list Q,U,E,S,T,I,O,N.
17. Consider the numbers given below.Show how the partition algorithm of quicksort will place 106 in its correct place. Show all the steps.
    106,117,129,114,141,91,84,63,42