A TEXT BOOK OF

DESIGN AND ANALYSIS OF ALGORITHMS

FOR

Semester – IV

SECOND YEAR (S.Y.) B. TECH COURSE IN COMPUTER ENGINEERING

Strictly According to New Revised Credit System Syllabus of Dr. Babasaheb Ambedkar Technological University (BATU), Lonere, (Dist. Raigad) Maharashtra,

(w.e.f June 2018)

Mrs. VAISHALI S. TIDAKE
ME. (CSE-IT),
Associate Professor,
Computer Engineering Deptt.,
NDMVPS's K.B.T College of Engineering,
NASHIK.

Dr. VAISHALI S. PAWAR
ME. (CE) Ph. D
Professor & Head,
Computer Engineering Deptt.,
NDMVPS's K.B.T College of Engineering,
NASHIK.

AJITKUMAR S. SHITOLE
M. Tech. (COEP),
Associate Professor,
Computer Engineering Deptt.,
International Institute of Inform. Technology, (I^2IT)
Hinjewadi, PUNE.

Price ₹ 100.00

NIRALI PRAKASHAN
ADVANCEMENT OF KNOWLEDGE

N0996

CONTENTS

Unit I

INTRODUCTION TO ALGORITHI

1.1 INTRODUCTION

- Algorithms are fundamental to computer science and software engineering. In the recent years a number of significant advances in the field of algorithms have been made. These advances have ranged from the development of faster algorithms such as Fast Fourier Transforms, to the astonishing discovery of certain natural problems for which all algorithms are inefficient.
- These results have considerable interest in the study of algorithms, and the area of algorithm design and analysis has blossomed into a field of intense interest. The intent of this unit is to bring together the fundamental results in this area, so the unifying principles and underlying concepts of algorithm design may be more easily understood.

Algorithm

- An algorithm is named from the ninth-century Persian mathematician Abu Jafar Mohammed ibn Musa-al-Khowarizmi.
- It is simply a set of rules for carrying out some task, either by hand or, more usually, on a machine. In other words, an algorithm is used by a computer to solve a problem.

1.2 THE ROLE OF ALGORITHMS IN COMPUTING

- The real world performance of any software depends on two things:
  1. The algorithm chosen, and
  2. The suitability and efficiency of various layers of implementation.
- Good algorithm design is therefore crucial for the performance of all software systems. Moreover, the study of algorithms provides insight into the intrinsic nature of the problem as well as possible solution techniques independent of programming language, paradigm, computer hardware, and any other implementation aspect.
- The study of algorithms introduces formal techniques to support the design and analysis of algorithms, focusing on both the underlying mathematical theory and practical considerations of efficiency. Topics include asymptotic complexity bounds and techniques of analysis.

1.3 WHAT ARE ALGORITHMS ?

- For solving any problem there may be one or r methods. We have to formulate some step by step act to get the solution of the problem. These steps nothing but an algorithm.
- A full and exact definition of an algorithm is complic and may be confusing initially. However all the esse ideas about an algorithm can be encapsulated in single statement that an "Algorithm is a set of rule carrying out some task, either by hand or more usuall a machine". This set of rules is idea behind a comp program. This idea is independent of implementation.
- An algorithm remains same whether the program Pascal running on a cray in New York or is in I running on a Macintosh in Kathmandu or a FORT program running on Param – 10000 in India! Th because before any computer program, there comes concept of an algorithm, which specifies the step formulate the method to solve that particular problen there may be numerous methods to solve a parti problem, naturally there may be more than one pos algorithms for the solution of the problem.

Algorithms - What are They?

- An algorithm has to solve general specified problem algorithmic problem is specified by describing the s input data it must work on and what desired prope the output it produces must have.

Fig. 1.1

Let us Redefine the Term Algorithm:

- An algorithm is a well-defined computational proce that transforms inputs into outputs, achieving the de input-output relationship. A computational problem specification of the desired input-output relationship instance of a problem is all the inputs needed to com a solution to the problem.
- A correct algorithm halts with the correct output for e input instance. We can then say that algorithm solves problem.

in rather more detail, an algorithm is a finite and definite procedure for solving a problem. The finiteness is important. The definiteness is also important. We cannot accept as algorithms those methods which involve making inspired guesses, like finding a clever substitution for an integral.

Hence, an algorithm is finite ordered set of unambiguous and effective steps which are when followed, accomplish a particular task, by accepting zero or more input quantities and generate at least one output.

PROPERTIES OF ALGORITHMS

owing are the Properties of an Algorithm :

Input : An algorithm is supplied with zero or more external quantities.

Output : An algorithm must produce one or more results / outputs.

Unambiguous Steps / Definiteness : Each step in an algorithm must be clear and unambiguous. This helps the one who is following the steps to take a definite action.

Finiteness : An algorithm must halt. Hence it must have finite number of steps.

Effectiveness : Every instruction must be sufficiently basic. An algorithm is an ordered finite set of unambiguous and effective steps that produces result and terminates.

EXPRESSING ALGORITHM

An algorithm is nothing but a sequence of steps to be followed to perform some task. There are different ways to express an algorithm like natural language, pseudocode, flowchart, etc.

For example, let us write an algorithm to find largest of two numbers.

1. Start
2. Accept two numbers from the user.
3. If the first number is greater than the second number, then output the first number, otherwise output the second number.
4. Stop

FLOWCHART

Flowchart is a way to express an algorithm. It's a pictorial representation of an algorithm. It uses following pictorial notations:

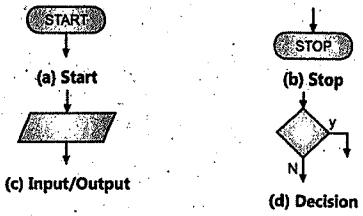


Fig. 1.2 : Notations used for flowchart

Flowchart for Finding Largest of Two Numbers is Shown Below.

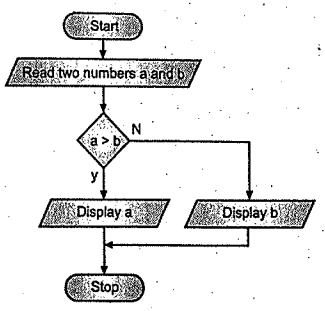


Fig. 1.3 : Sample flowchart

1.7 ALGORITHM DESIGN TECHNIQUES

- An algorithmic strategy is sometimes also called as design technique or design paradigm. It is a general approach to solve problems algorithmically that is applicable to a variety of problems for different areas of computing. Creation of an algorithm is an art which may never be fully automated. They help you to devise new and useful algorithms easily. There are five algorithmic strategies:
  1. Divide and conquer
  2. Backtracking
  3. Branch and bound
  4. Dynamic programming
  5. Greedy technique.

Each of These Techniques is Briefly Described Below:

1. Divide and Conquer

- This approach is suitable for problems where the subproblems are of same type to original problem. It is a common approach to solve a problem.
- It partitions the problem into smaller parts, find solutions for the parts, and then combine the solutions for the parts into a solution for the whole.
- When this approach is combined with recursion, it yields efficient solutions.

2. Backtracking

- This approach does not follow fixed rules of computation. It follows trial and error to solve a problem. The principle idea is to construct solutions one component at a time and evaluate such partially constructed solutions as follows:
- If a partially constructed solution can be developed further without violating the problem's constraints, it is done by taking the first remaining legitimate option for the next component. If there is no legitimate option for the next component, no alternatives for any remaining component need to be considered. In this case, the algorithm backtracks to replace the last component of the partially constructed solution with its next option.

3. Branch and Bound

- We know that the graph can be searched using two ways: Depth First Search (DFS) or Breadth First Search (BFS). In the branch and bound approach, at each node we calculate a bound on the possible value of solutions.
- Calculation of the bounds is combined with either DFS or BFS. The calculated bound is used to decide which node should be expanded first. The nodes which can be expanded (explored) are termed as live nodes.

4. Dynamic Programming

- Using recursive approach, if a problem of size n is partitioned into n problems of size n-1, then the algorithm will have an exponential growth. In such case dynamic programming gives a more efficient algorithm. It is a tabular technique.
- It calculates the solution to all subproblems. Once a subproblem is solved, the answer is stored in a table and never recalculated again. Calculations start from the small subproblems and proceed towards the larger subproblems.

5. Greedy Approach

- This approach builds a solution to a problem in steps. At each iteration, it adds a part of the solution. Part of the solution to be added next is decided using a greedy rule.
- The rule is greedy because among all the parts of a solution that might be chosen, the best one is selected.
- A greedy approach may or may not give optimal solution. If it is not optimal, it is near to optimal and hence good enough for many applications.

1.8 DESIGN OF ALGORITHMS

- The various issues can arise in the design of all algorithms. The design approach is dependent on the model Choose.
- Same problem can be solved by using more than one algorithm. The algorithm and decided by their effectiveness.

e.g. Traveling salesman problem. The model selected suggest the following as a possible algorithm.

Problem Description :

1. "n" cities are arbitrarily numbered from 1 to n.
2. Every tour corresponds to a permutation of numbers 1, 2, 3, ... n – 1. Base city 'n' is always included first and last.
3. There is one to one correspondence between a permutation and particular tour. Hence permutation is tour.
4. Three important functions are function permutation, tour formation and cost calculation.

Algorithms for Exhaustive Travelling Salesman :

To solve an n-city travelling salesman problem by conside all permutation on fist n-1 integer. Choose a tour with cost MIN.

Input : number of cities n, cost matrix C

```
1.  Initialize
2.  Tour ← 0
3.  Min ← ∞
4.  Step 1 : Generate permutations
5.  for i ← 1 to (n - 1) do
6.  Get new permutation
7.  P ← K + V permutation of integers 1 to n-1
8.  Construct a new tour
9.  Construct the tour T(p) corresponding to P
10. Compute cost as cost (T(P))
11. Compare
12. if cost (T(P)) < MIN then
13. Tour ← T (P)
14. MIN ← Cost (T(P))
15. End
16. End
```

1.9 PERFORMANCE ANALYSIS OF ALGORITHM

- We have studied methods to compute time compl The total time taken by the algorithm or progra calculated using the sum of the time taken by each c executable statement in an algorithm or a program. required by each statement depends on:
  - Time required for executing it once.
  - Number of times the statement is executed.
- Product of above two gives time required for particular statement. First compute execution time executable statements. Summation of all is the total required for that algorithm or program. Generally, we sum the frequency count of all the statements, w a polynomial.
- In analysis, we are interested in the order of magnitu an algorithm i.e., we are interested in only statements, which have the greatest frequency coun function is linear – that is, if it contains no loops - th efficiency is a function of the number of instructic contains. In this case, its efficiency depends on the s of computer and is generally not a factor in the o efficiency of the program.
- On the other hand, functions containing loop vary v in their efficiency. The study of algorithm effic therefore focuses on loops. As we study specific exan we shall generally discuss the algorithm's efficiency function of the number of elements to be processec general format is:

$$f(n) = \text{efficiency}$$

A program consists of many statements made up of various constructs like for loop, while loop, do-while (repeat) loop, recursive function calls. These constructs influence performance analysis of the algorithm. Remaining constructs like conditional statement (if, if-else, switch, etc.) do not repeat the statement execution. Hence mainly loops and recursion are considered for performance analysis.

10 ANALYSIS OF ALGORITHMS

Algorithms heavily depend on the organization of data. There can be several organizations of data and/or algorithms for given problem. The question is which of the algorithms is the best?

We can compare one algorithm with other and choose the best. This is called as analysis of algorithms. Analysis involves measuring the performance of an algorithm.

rformance is Measured in Terms of :

- **Programmers Time Complexity :** Very rarely taken into account as is paid only once.
- **Time Complexity :** The amount of time taken by algorithm to perform the intended task.
- **Space Complexity :** The amount of memory needed to perform the task. Space complexity is computed by considering the data and their sizes.

is very convenient to classify algorithms based on the ative amount of time and relative amount of space they quire and specify the growth of time space requirements as unction of the input size.

mplexity of Algorithms :

Algorithms are measured in terms of time and space complexity. The Time Complexity of an algorithm is a measure of how much time is required to execute an algorithm for a given number of inputs.

The time complexity of an algorithm is measured by its rate of growth relative to standard functions. Standard functions are given in Table 1.1.

Table 1.1 : Standard Time Complexity Functions

Function	Name
c	Constant
log N	Logarithmic
$\log_2 N$	Log-squared
N	Linear
N^2	Quadratic
N^3	Cubic
C^N	Exponential

Space complexity is similar to time complexity. The Space Complexity of an algorithm is a measure of how much storage is required by the algorithm. It is possible to make an algorithm to use more space and less time.

- Typically, computer scientists are interested in minimizing the time complexity of algorithms. The economics of storage versus the speed of computers are the principal factors determining the focus on time complexity.
- Memory has decreased in cost at an exponential rate for the past 25 years whereas the cost of central processing unit time has not decreased at that rate. The bottleneck is execution time. Hence, computer scientists focus on the execution time of algorithms.
- An algorithm can be characterized by a timing function T(N). T(N) is a measure of how much time is required to execute an algorithm given N values. For example, the timing function for a sort operation specifies the time required to sort N values. The timing function for an algorithm that solves a system of linear equations specifies the time required solving N linear equations.
- If we say that an algorithm is $O(N^2)$, pronounced, oh of N squared, then what we mean is that the timing function for the algorithm will grow no faster than the square of the number of values it processes.

Let us redefine the complexities :

- **Time Complexity :** Running time of the program as a function of the size of input.
- **Space Complexity :** Amount of memory required during the program execution.

Let us study both of them in detail.

1.10.1 Space Complexity

- Space complexity specifies the amount of computer memory required during the program execution, as a function of the input size.
- Space complexity measurement, that is, measurement of space requirement of an algorithm can be done at two different times :

1. Compile Time and
2. Run Time

1. Compile Time Space Complexity :

Compile time space complexity is defined as the storage requirement of a program at compile time. This storage requirement can be computed during compile time. The storage needed by the program at compile time can be determined by summing up the storage size of each variable using declaration statements. For example, space complexity of a recursive function of calculating factorial of number 'n' depends upon the number n itself.

The space complexity = Space needed at compile time. This includes memory requirement before execution starts.

2. Runtime Space Complexity :

If program is recursive or uses dynamic variables or dynamic data structure, then there is a need to determine space complexity at runtime. Generally this dynamic storage size is dependent on some parameters used in a program. It is difficult to estimate memory requirement accurately, as it is also determined by efficiency of compiler. The memory requirement is summation of the program space, data space and stack space.

- **Program Space :** This is memory occupied by program itself.
- **Data Space :** This is memory occupied by data members such as constants and variables.
- **Stack Space :** This is stack memory needed to save functions runtime environment while another function is called. This cannot be accurately estimated, since it depends on the runtime call stack, which can depend on the programs' data set. This memory space is crucially important for recursive functions.

1.10.2 Time Complexity

Time complexity specifies the maximum time required for the execution of program as a function of the input size. Time Complexity T(P) is the time taken by program P and is the sum of the compile and execution time. This is system dependent. Let us count the number of algorithm steps. Algorithm step is defined as a syntactically or semantically meaningful segment of a program. We can determine the number of steps needed by a program to solve a particular problem instance in one of the two ways :

1. Introduce a new variable, say count, into the program. This is a global variable with initial value 0. Statements to increment count are introduced in the program. This is done so that each time the statement in original program is executed, count is incremented by step count of that statement.

   We measure the run time of an algorithm by counting the number of steps.
2. Compute number of times each statement will be executed manually. Number of times the statement is executed is its frequency count. Sum frequency counts of all statements. This sum is the number of steps needed to solve given problem.

1.11 TYPES OF ALGORITHM'S ANALYSIS

There are three types of doing analysis of algorithms:

1. Best case
2. Worst case
3. Average case

1. The Best Case complexity of the algorithm is the function defined by the minimum number of steps taken on any instance of size n.

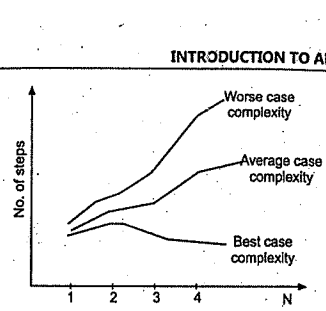


Fig. 1.4

2. The Worst Case complexity of the algorithm is function defined by the maximum number of steps on any instance of size n.
3. The Average-Case complexity of the algorithm is function defined by an average number of steps tak any instance of size n.

Each of these complexities defines a numerical function vs. size.

1.11.1 Analyzing Algorithms

Analyzing algorithm includes :

1. Proof of correctness of an algorithm
2. Cost of an algorithm

Given an Algorithm to be Analyzed :

- The first task is to determine which operations employed and what their relative costs are. operations include integer operations (+, –, arithmetic on floating point numbers, compari assigning values to variables and executing proce calls. These operations need fixed amount of time, a are called as bounded by constant. Total time fo comparison of two strings will depend upon their len while the time for each character comparison is bou by a constant.
- The second task is to determine a sufficient numb data set which causes the algorithm to exhibit all po patterns of behavior.

Analysis of Computing Time :

- **Priori Analysis** (Theoretical) (Performance analysis).
- **Posterior Analysis** (Empirical) (Performance mea ment).
- In a priori analysis, we obtain a function which bound algorithms computing time. In a posterior analysis collect actual statistics about the algorithms consum of time and space, while it is executing. This requires statement's frequency count (number of times state will be executed) and time for one execution.
- Product of these two numbers is the total time. Sinc time per execution depends on both the machine l used and the programming language with its com Order of magnitude of algorithm refers to the su frequencies of all its statements. Determining the orc magnitude of an algorithm is very important

producing an algorithm which is faster by an order of magnitude is significant accomplishment.

A priori analysis is mainly concerned with order of magnitude determination. We need to formalize the ideas that an algorithm has running time or storage requirements that are never 'more than', 'always greater than' or 'exactly' same amount. Recall that we use T(n) for the running time or time complexity and S(n) for the storage requirements or space complexity of an algorithm on an instance of size n. The phrase never more than is upper bound on T(n) or S(n).

ficiency of an Algorithm :

There may exists more than one algorithm for one problem. When comparing two different algorithms that solve the same problem, we will often find that one algorithm is of an order of magnitude more efficient than the other. In this case, it only makes sense that we should be able to recognize and choose the more efficient algorithm.

symptotic Notation :

Ignoring machine and language dependent factors, consider only order of magnitude of frequency-counts for executing statements. The notations are :

- O(n), big-O-notation gives the upper bound on the time complexity value for the algorithm. We shall study details of big-O-notation in further topics.
- Ω(n), omega notation gives minimum amount (lower bound) of time an algorithm needs.
- θ(n), theta notation, when best and worst case requires same time, θ(n) is used.

.12 ORDER OF GROWTH

Table 1.2 : Time Measure of Efficicnecy

Efficicency	Big-O	Iterations	Estimated Time
Logarithmic	$O(\log_2 n)$	14	Microseconds
Linear	O(n)	10,000	0.1 seconds
Linear Logarithmic	$O(n(\log_2 n))$	140,000	2 seconds
Quadratic	$O(n^2)$	$10{,}000^2$	15 – 20 minutes
Polynomial	$O(n^k)$	$10{,}000^k$	Hours
Exponential	$O(C^n)$	$2^{10{,}000}$	Intractable
Factorial	O(n!)	10,000!	Intractable

Assumes instruction speed of 1 micro second and 10 instructions in loop. The magnitude of their efficiency for a problem containing 10,000 elements shows that the linear solution requires a fraction of second while the quadratic solution requires upto 20 minutes. Looking at the problem from the other end, if we are using a computer that executes a million instructions per second and the loop contains ten instructions, then we would spend 0.00001 for each iteration of the loop. The above Table 1.2 also contains an estimate of the time needed to solve the problem given different efficiencies.

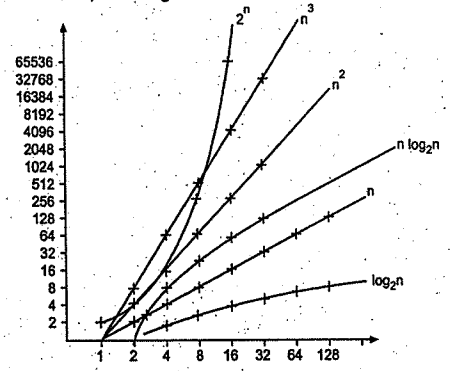


Fig. 1.5 : Rate of growth of common computer time functions

1.13 ASYMPTOTIC NOTATION

- A priori analysis of computing time ignores all the factors which are machine or language dependent and concentrates on determining order of magnitude of algorithm.
- We need to formalize the idea that an algorithm has running time or storage requirements that are "never more than," "always greater than," or "exactly" some amount. We use T(n) for the running time or time complexity and S(n) for the storage requirements or space complexity of an algorithm on an instance of size n. And recall that the time and space may vary for different instances of the same size, For example, it may take less time to sort an already sorted list.
- The phrase "never more than" expresses an upper bound on T(n) or S(n). There are several ways you could choose to express this.:
  - The algorithm never takes more than some function of n.
  - The algorithm has running time that is always less than some function of n.
  - The algorithm's running time is in the order of some function of n.
  - The algorithm's time complexity is big-O of some function of n.

You can make similar statements about space complexity.

The phrase "always greater than" expresses a lower bound on T(n) or S(n).

There are several ways you could choose to express this :

- The algorithm always takes at least (some function of n) operations.
- The algorithm has running time that is always greater than some function of n.
- The algorithm's time complexity is big-Omega (Ω) of some function of n.

You can make similar statements about space complexity.

There are certain technical features which must be placed on this function referred to above. We won't try to state all of them. Let's give the function a name, call it f(n).

We are familiar with the common functions that will be useful complexity functions. We have listed them in section 1.9 in Table 1.1.

- The constant function $f(n) = c$;
- The logarithmic function $f(n) = lg\ n$;
- The linear function $f(n) = n$;
- The linear-log function $f(n) = n\ lg\ n$;
- The quadratic function $f(n) = n^2$;
- The cubic function $f(n) = n^3$;
- The general power function $f(n) = n^k$;
- The exponential function $f(n) = 2^n$;
- The factorial function $f(n) = n!$;
- The function $f(n) = n^n$.

1.13.1 Big O-notation (O)

f(n) = O(g(n)) if there exist two +ve constants c and n_0 such that $|f(n)| \le cg(n)|$ for all values $n \ge n_0$.

When we say that an algorithm has computing time O(g(n)), we mean that if the algorithm is run on same computer on the same type of data, but increasing value of n, the resulting time will always be less than some constant time |g(n)|.

If an algorithm has k-statements whose orders of magnitude are $C_1 n^{m1} + C_2 n^{m2} + \ldots\ldots + C_k n^{mk}$, order of algorithm is $O(n^m)$, where, $m = \max(m_i)$, $1 \le i \le k$.

O-notation is used to denote upper bounds. We write

$$T(n) = O(f(n))$$

to denote that T(n) never takes more than roughly f(n) operations. This is stated in English as :

"T(n) is in the order of f(n)", or "T(n) is big-O of f(n)".

We need to make this informal notion more precise.

Definition 1 : Let T(n) be the time complexity of an algorithm and let f(n) be a proper complexity function. We write

$$T(n) = O(f(n))$$

if and only if there exists a natural number N > 0 and a constant c > 0 such that

$$T(n) \le c\ f(n);\ \forall n > N$$

This can be interpreted as follows :

"T grows slower than a constant time f for all sufficiently la n",

or

"The graph of T lies below the graph of a constant time f all sufficiently large n".

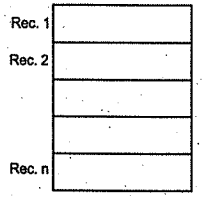


Fig. 1.6

Consider an algorithm of linear search. In this example search a particular record in a file of 'n' records, there a possibilities :

- The record is present at 1st position (Best case).
- The record is present at last position (Worst case).
- The record is somewhere in between 1st and record (Average case).
- Record is absent (Worst case).

This algorithm will take maximum of 'n' number comparisons to search the record, if the record is at n^{th} p or if the record is absent. So the maximum time taken by algorithm is say 'n comparisons'.

Time complexity can be said as O(n).

Now consider a case if the record is present at the 1st loca (Best case). Here minimum time taken by the algorithm is t for one comparison. Therefore the lower bound of algorithm can be stated as Ω(1) which is constant.

1.13.2 Omega Notation (Ω)

f(n) = Ω(g(n)) if there exist positive constants c and n_0 s that for all $n > n_0$.

$$f(n) \ge cg(n)$$

In some cases the time for an algorithm, f(n) will be such f(n) = Ω(g(n)) and f(n) = O(g(n)). For such circumstances use another notation i.e. θ notation.

Ω-notation is used to denote lower bounds. We write

$$T(n) = \Omega(f(n))$$

to denote that T(n) always takes at least roughly operations. This is stated in English as

"T(n) is Ω of f (n)".

We need to make this informal notion more precise. This be interpreted as follows:

"T grows faster than a constant time f for all sufficiently la n", or

"The graph of T lies above the graph of a constant time f all sufficiently large n".

1.13.3 Theta Notation (θ)

$f(n) = \theta(g(n))$ if there exists positive constants c_1, c_2 and n_0 such that

$$c_1 g(n) \leq f(n) \leq c_2 g(n)$$

If $f(n) = \theta(g(n))$, then g(n) is both upper and lower bound on f(n). This means that the worst and the best cases require the same amount of time within a constant factor.

θ-notation is used to denote both upper and lower bounds. We write

$$T(n) = \theta(f(n))$$

to denote that T(n) always takes roughly f(n) operations. This is stated in English as

"T(n) is θ of f(n)"

We need to make this informal notion more precise.

Example : Consider the algorithm of finding a maximum number from a list of 'n' unsorted numbers. In this algorithm if the maximum number is present at 1st location or at last location, our algorithm is going to perform 'n' comparisons always. So each time our algorithm takes same maximum time and same minimum time i.e. 'n' comparisons. This means that lower bound and upper bound of time complexity is same, so this can be represented by asymptotic notation θ. So this algorithm takes

(1) O(n)
(2) Ω (n)

and hence (3) θ (n)

Definition : Let T(n) be the time complexity of an algorithm and let f(n) be a proper complexity function. We write

$$T(n) = \theta(f(n))$$

if and only if there exists a natural number $N > 0$ and constants c, $d > 0$ such that

$$d.f(n) \leq T(n) \leq c.f(n) \quad \forall\ n_0 > N$$

This can be interpreted as follows:

"T grows at the same rate as some constant time f for all sufficiently large n", or

"The graph of T lies between the graphs of d.f and c.f for all sufficiently large n".

```
/*  Algorithm to find maximum from n numbers */
        max (A, n, j)
        {   i = 1;
            max = A(i)
            for i = 2 to n do
                begin
                    if A(i) > max then
                        max = A(i);
                        j = i;
                end if
            end for
        }
```

Above algorithm has computing time both O(n) and Ω(n), since the for loop always makes n iterations. Thus, we say that its time complexity is θ(n).

The algorithm to search a number in an array of n elements has computing time as O(n) and Ω(1), the worst case and the best case respectively.

1.14 RECURSION

Using recursion,

```
1. function factorialR (int n)
2. {
3.     if (n = 1)
4.         return 1;
5.     else
6.         return factorialR(n * factorialR(n-1));
7. }
```

Here if n = 1, then only statements 3 and 4 are executed, which requires constant time, say a.

But if n > 1, then time required for n depends on time required for (n – 1). Hence, this can be stated by recurrence equation as

$$T(n) = \begin{cases} a & \text{, if } n = 1 \\ T(n-1) + b & \text{, otherwise} \end{cases}$$

Solving the above recurrence equation gives O(n).

1.15 ITERATIVE METHOD

- For recursive function calls, the flow of control in the algorithm can be observed to find the recurrence equation. This recurrence equation can be solved to perform analysis. For example, factorial of a number can be computed using iteration or recursive function.

Using iterations,

```
1. function factorial(int n)
2. {
3.     int fact = 1, i;
4.
5.     for i = 1 to n
6.     {
7.         fact = fact * i;
8.         i = i + 1;
9.     }
10.    return (fact);
11. }
```


- Here statement 7 and 8 are executed n times. Statement 5 is executed (n + 1) times.

Statement 3 and 10 are executed once. If each statement requires constant time C, then the algorithm requires

$$c + (n+1)c + nc + nc + c = O(n)$$

1.16 RECURRENCES RELATION

To solve any problem, we can use different algorithms. But to decide which algorithm is efficient, we analyze the algorithms. The last step of analysis is to solve recurrence equations. We can solve the recurrence equation by calculating some initial values and by guessing its general form depending on the regularity of calculated values. Finally we can prove the obtained general form by mathematical induction.

Some Important Formulae for Analyzing Algorithm Efficiency :

$$\log x^y = y \log x$$
$$\log xy = \log x + \log y$$
$$\log (x/y) = \log x - \log y$$
$$\log_a x = \log_a {}^b \log_b x$$
$$a \log_b x = x \log_b a$$

lg x represents the logarithm to the base 2.

ln x represents the logarithm to the base e = 2.718.

Permutation p(n)= n!

$$\text{Combination } c(n, k) = \frac{n!}{k!(n-k)!}$$

$$\sum_{i=1}^{n} 1 = n \qquad 1 + 1 + 1 \ldots\ldots n \text{ times}$$

$$\sum_{i=1}^{n} i^k \approx \frac{n^{k+1}}{k+1} \qquad 1^k + 2^k + 3^k \ldots\ldots + n^k$$

$$\sum_{i=1}^{n} 2^i = 2^{n+1} - 1 \qquad 2^1 + 2^2 + 2^3 + \ldots\ldots + 2^n$$

$$\sum_{i=1}^{n} \frac{1}{i} \approx \ln n + r, \text{ where } r \approx 0.5772 \qquad 1 + \frac{1}{2} + \frac{1}{3} + \ldots\ldots + \frac{1}{n}$$

$$\sum_{i=1}^{n} \lg i \approx n \lg n \qquad \lg 1 + \lg 2 + \ldots\ldots + \lg n$$

$$\sum_{i=l}^{u} c x_i = c \sum_{i=l}^{u} x_i$$

$$\sum_{i=l}^{u} (x_i \pm y_i) = \sum_{i=l}^{u} x_i \pm \sum_{i=l}^{u} y_i$$

$$\sum_{i=l}^{u} 1 = u - l + 1$$

$$\sum_{i=1}^{n} i = 1 + 2 + \cdots + n = \frac{n(n+1)}{2} \in \theta(n^2)$$

where, $l \leq u$ (l is the lower bound and u is the upper bound.)

1.16.1 Homogeneous Linear Recurrences

- Instead of using the guessing technique, there is r powerful technique available which uses the characte equation. Let us see the technique of the characte equation to solve homogeneous linear recurrences. homogeneous linear recurrences with cons coefficients have the following form :

$$a_0 t_n + a_1 t_{n-1} + \cdots + a_k t_{n-k} = 0 \quad \ldots$$

- Here we want to solve for values of t_n, t_{n-1}, ..., t_{n-k}. Tha we want to find out values of t_i on k where, $0 \leq i \leq k-$ $1 \leq i \leq k$. These values are called as initial conditions.
- Equation (1.1) is linear, homogeneous and it constant coefficients.
- It is linear because it contains the following terms of t t_{n-1}, ..., t_{n-k}. It does not contain $t_{n-1} * t_{n-k}$ or t^2_{n-1} etc.
- The equation (1.1) is homogeneous because the lir combination of t_i is equal to zero.
- The equation (1.1) also uses constant coefficients a_1, ..., a_k.
- Any linear combination of solutions of an equation i solution of that equation. So t_n is also solution equation (1.1).

$$a_0 x^k + a_1 x^{k-1} + \ldots + a_k = 0 \quad \ldots (1$$

- This is the equation of degree k in x. It is called as characteristic equation. The following is the characteris polynomial in x:

$$p(x) = a_0 x^k + a_1 x^{k-1} + \ldots + a_k$$

- Because p(x) is a polynomial of degree k, it has k roo The k roots may be all distinct or they may not be. these roots r_i are the only solutions for the equati $p(x) = 0$ and this polynomial p(x) can be expressed as product of k terms involving roots r_i. For any root value $p(r_i) = 0$ means that $x = r_i$ is a solution of the characteris equation (1.2). So if all the roots r_i of the characteris equation are distinct, then for the constants c_1, c_2, ..., the solution for the recurrence is

$$t_n = \sum_{i=1}^{k} c_i r_i^n \quad \ldots (1.$$

- The constants c_1, c_2, ..., c_k can be obtained by solving linear equations using k initial conditions.
- If all the roots are not distinct, then the solution chang slightly and it also uses the multiplicity of each root.

To Summarize, the Homogeneous Linear Recurrences with Constant Coefficients can be Solved using the Following Steps :

(1) First rewrite the recurrence in the following form:

$a_0 t_n + a_1 t_{n-1} + \dots + a_k t_{n-k} = 0$

(2) Then form the characteristic polynomial of degree k using constants as:

$a_0 x^k + a_1 x^{k-1} + \dots + a_k$

(3) Find the k roots of the characteristic polynomial.

(a) If k roots are distinct, then the general solution is of the following form:

$$t_n = c_1 r_1^n + c_2 r_2^n + \dots + c_n r_k^n$$

(b) If k roots are not distinct, then suppose root r_1 has multiplicity m_1 and root r_2 has multiplicity m_2, then $m_1 + m_2 = k$ and the general solution is of the following form:

$$t_n = c_{11} r_1^n + c_{12} n r_1^n + c_{13} n^2 r_1^n + \dots + c_{c1,m1} n^{(m_1-1)} r_1^n + c_{21} r_2^n + c_{22} n r_2^n + c_{23} n^2 r_2^n + \dots + c_{2,m2} n^{(m_2-1)} r_2^n$$

(4) Now we can solve k equations using k initial conditions to find values of constants c_i.

(5) After putting the constant values in the equation of t_n and simplifying it, we obtain the solution for the recurrence.

Let us Solve Some Recurrence Equations which are Homogeneous and Linear.

SOLVED EXAMPLES

Example 1.1 : *Solve the following recurrence:*

$$t_n = \begin{cases} n & , \text{ if } n = 0 \text{ or } n = 1 \\ 5t_{n-1} - 6t_{n-2} & , \text{ otherwise} \end{cases}$$

Solution : First rewrite the recurrence as :

$t_n - 5t_{n-1} + 6t_{n-2} = 0$

Here degree of polynomial k = 2, coefficients are $a_0 = 1$,

$a_1 = -5, a_2 = 6$

The characteristic polynomial is

$a_0 x^k + a_1 x^{k-1} + a_2 x^{k-2} = x^2 - 5x + 6 = (x-3)(x-2)$

So roots of the characteristic polynomial are

$r_1 = 3$, and $r_2 = 2$

The general solution is of the following form:

$t_n = c_1 r_1^n + c_2 r_2^n$

$\therefore \quad t_n = c_1 (3)^n + c_2 (2)^n$

The initial conditions are n = 0 and n = 1.

$t_n = n$, if n = 0, 1 .... (given)

For n = 0

$c_1 (3)^0 + c_2 (2)^0 = 0$

i.e. $c_1 + c_2 = 0$ ...(1)

For n = 1,

$c_1 (3)^1 + c_2 (2)^1 = 1$

i.e. $3c_1 + 2c_2 = 1$ ... (2)

By multiplying equation (1) by 2, we get

$2c_1 + 2c_2 = 0$ ... (3)

By doing subtraction of equation (2) and equation (3), we get,

$c_1 = 1$

and $c_2 = -c_1$

$\therefore \quad c_2 = -1$

Therefore $t_n = 3^n - 2^n$

Example 1.2 : *Solve the following recurrence*

$$t_n = \begin{cases} 9n^2 - 15n + 106 & , \text{ if } n = 0, 1 \text{ or } 2 \\ 5t_{n-1} - 8t_{n-2} + 4t_{n-3} & , \text{ otherwise} \end{cases}$$

Solution : First rewrite the recurrence as

$t_n - 5t_{n-1} + 8t_{n-2} - 4t_{n-3} = 0$

Here k = 3, $a_0 = 1, a_1 = -5, a_2 = 8, a_3 = -4$

The characteristic equation is

$x^3 - 5x^2 + 8x - 4$

$= (x-1)(x-2)^2$

So the roots of the characteristic equation are $r_1 = 1$ and $r_2 = 2$ with multiplicity 2.

The general solution is of the form

$t_n = c_1 r_1^n + c_2 r_2^n + c_3 n r_2^n$

$\therefore \quad t_n = c_1 (1)^n + c_2 (2)^n + c_3 n (2)^n$

The initial conditions are

$n = 0, 1, 2$

$t_n = 9n^2 - 15n + 106$, if n = 0, 1, 2 ... (given)

For n = 0, we get

$c_1 + c_2 + 0 = 106$ ...(1)

For n = 1, we get

$c_1 + 2c_2 + 2c_3 = 100$ ... (2)

For n = 2, we get

$c_1 + 4c_2 + 8c_3 = 112$ ... (3)

Solving these three equations, we get

$c_1 = 136, c_2 = -30,$

$c_3 = 12$

Therefore

$t_n = 136(1)^n - 30(2)^n + 12n(2)^n$

Example 1.3 : *Solve the following recurrence of fibonacci series*

$$f_n = \begin{cases} n & , \text{ if } n = 0 \text{ or } n = 1 \\ f_{n-1} + f_{n-2} & , \text{ otherwise} \end{cases}$$

Solution : First rewrite the recurrence

$f_n - f_{n-1} - f_{n-2} = 0$

Here, k = 2, $a_0 = 1, a_1 = -1, a_2 = -1$

The characteristic polynomial is $x^2 - x - 1$

The roots of the characteristic polynomial are

$r_1 = \frac{1+\sqrt{5}}{2}$ and $r_2 = \frac{1-\sqrt{5}}{2}$

The general solution is of the form

$f_n = c_1 r_1^n + c_2 r_2^n$

$\therefore \quad f_n = c_1 \left(\frac{1+\sqrt{5}}{2}\right)^n + c_2 \left(\frac{1-\sqrt{5}}{2}\right)^n$

The initial conditions are

$f_n = n$, if n = 0 or 1 .... (given)

For n = 0, we get

$c_1 + c_2 = 0$ ... (1)

For n = 1, we get

$r_1 c_1 + r_2 c_2 = 1$ ... (2)

From equation (1), we get

$c_1 = -c_2$.

Putting it in equation (2), we get

$-r_1 c_2 + r_2 c_2 = 1$

$\therefore \quad c_2 (r_2 - r_1) = 1$

$\therefore \quad c_2 = \frac{1}{r_2 - r_1}$

$= \frac{1}{\left(\frac{1-\sqrt{5}}{2}\right) - \left(\frac{1+\sqrt{5}}{2}\right)}$

$= \frac{2}{1-\sqrt{5}-1-\sqrt{5}}$

$= \frac{2}{-2\sqrt{5}}$

$= -\frac{1}{\sqrt{5}}$

$\therefore \quad c_2 = -\frac{1}{\sqrt{5}}$ and $c_1 = \frac{1}{\sqrt{5}}$

Therefore

$f_n = \frac{1}{\sqrt{5}} r_1^n - \frac{1}{\sqrt{5}} r_2^n$

$\therefore \quad f_n = \frac{1}{\sqrt{5}} \left[\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n \right]$

Example 1.4 : *Solve the following recurrence*

$$t_n = \begin{cases} n & , \text{ if } n = 0 \text{ or } n = 1 \\ 2t_{n-1} - 2t_{n-2} & , \text{ otherwise} \end{cases}$$

Solution : Rewrite the recurrence as $t_n - 2t_{n-1} + 2t_{n-2} = 0$

Here k = 2, $a_0 = 1, a_1 = -2, a_2 = 2$

The characteristic polynomial is

$x^2 - 2x + 2$

The roots of the characteristic polynomial are

$r_1 = 1 + \frac{\sqrt{-4}}{2}$ and $r_2 = 1 - \frac{\sqrt{-4}}{2}$

The general solution is of the form

$t_n = c_1 r_1^n + c_2 r_2^n$

The initial conditions are n = 0 or 1.

$t_n = n$, if n = 0 or 1 ... (giv

For n = 0, we get

$c_1 + c_2 = 0$ ...

For n = 1, we get

$c_1 r_1 + c_2 r_2 = 1$ ...

From equation (1), we get

$c_1 = -c_2$

Putting it in equation (2), we get

$-c_2 r_1 + c_2 r_2 = 1$

$\therefore \quad c_2 (r_2 - r_1) = 1$

$\therefore \quad c_2 = \frac{1}{r_2 - r_1}$

$= \frac{1}{\left(1 - \frac{\sqrt{-4}}{2}\right) - \left(1 + \frac{\sqrt{-4}}{2}\right)}$

$= \frac{2}{2 - \sqrt{-4} - 2 - \sqrt{-4}}$

$= \frac{2}{-2\sqrt{-4}}$

$= -\frac{1}{\sqrt{-4}}$

$\therefore \quad c_2 = -\frac{1}{\sqrt{-4}}$

and $c_1 = \frac{1}{\sqrt{-4}}$

Therefore $t_n = \frac{1}{\sqrt{-4}} r_1^n - \frac{1}{\sqrt{-4}} r_2^n$

$\therefore \quad t_n = \frac{1}{\sqrt{-4}} \left[\left(1 + \frac{\sqrt{-4}}{2}\right)^n - \left(1 - \frac{\sqrt{-4}}{2}\right)^n \right]$

Example 1.5 : *Solve the following recurrence*

$$t_n = \begin{cases} n & \text{if } n = 0 \\ 5 & \text{if } n = 1 \\ 3t_{n-1} + 4t_{n-2} & \text{otherwise} \end{cases}$$

Solution : Rewrite the recurrence as

$t_n - 3t_{n-1} - 4t_{n-2} = 0$

Here, $k = 2$,

$a_0 = 1, a_1 = -3, a_2 = -4$

The characteristic polynomial is

$$x^2 - 3x - 4 = (x-4)(x+1)$$

So the roots of the characteristic polynomial are $r_1 = 4$ and $r_2 = -1$

The general solution is of the form

$$t_n = c_1 r_1^n + c_2 r_2^n$$

$$\therefore \quad t_n = c_1 4^n + c_2 (-1)^n$$

The initial conditions are n = 0 or 1.

For $n = 0$, we get

$$c_1 + c_2 = 0 \quad \ldots(1)$$

For $n = 1$, we get

$$4c_1 - c_2 = 5 \quad \ldots(2)$$

Solving equation (1) and (2), we get $5c_1 = 5$

$$\therefore \quad c_1 = 1 \text{ and } c_2 = -1$$

Therefore $t_n = 4^n - (-1)^n$

1.16.2 Inhomogeneous Recurrences

We have seen in the last section how we can solve the homogeneous linear recurrences with constant coefficients, which equate to zero. But if recurrences are inhomogeneous, then we have to use different methods to solve them.

Case 1 : See following recurrence which does not equate to zero.

$$a_0 t_n + a_1 t_{n-1} + \ldots + a_k t_{n-k} = b^n p(n) \quad \ldots(1.4)$$

Here, b is a constant and p(n) is a polynomial in n of degree d. For such recurrences, the characteristic polynomial is of the form

$$(a_0 x^k + a_1 x^{k-1} + \ldots + a_k)(x-b)^{d+1}$$

After getting this characteristic polynomial, remaining steps are same as that for solving the homogeneous recurrence. Because the initial conditions are not specified, they are obtained from the recurrence.

Case 2 : Consider the recurrence of the following form:

$$a_0 t_n + a_1 t_{n-1} + \ldots + a_k t_{n-k} = b_1^n p_1(n) + b_2^n p_2(n) + \ldots$$

Here, b_1, b_2, ... are distinct constants and $p_1(n)$, $p_2(n)$, ... are polynomials in n of degree d_1, d_2, ... respectively. For such recurrences, the characteristic polynomial is of the form:

$$(a_0 x^k + a_1 x^{k-1} + \ldots + a_k)(x-b_1)^{d_1+1}(x-b_2)^{d_2+1} \ldots$$

After getting this polynomial, remaining steps are same as before.

Let us solve some examples of inhomogeneous recurrences of the case 1 form.

Example 1.6 : *Solve the following recurrence, which gives the number of movements of a ring required in the towers of Hanoi problem:*

$$t_n = \begin{cases} 0 & \text{if } n = 0 \\ 2t(n-1) + 1 & \text{otherwise} \end{cases}$$

Solution : Rewrite the recurrence as:

$t(n) - 2t(n-1) = 1$

where, b = 1, and p(n) = 1 which is a polynomial of degree 0.

Here, $k = 1$, $a_0 = 1$, $a_1 = -2$

The characteristic polynomial is of the form

$$(a_0 x^k + a_1 x^{k-1} + \ldots + a_k)(x-b)^{d+1}$$

Here, $(x-2)(x-1)$

So the roots of the characteristic polynomial are $r_1 = 2$, $r_2 = 1$.

The general solution is of the form

$$t(n) = c_1(2)^n + c_2(1)^n$$

For two constants c_1 and c_2, we can use the initial conditions for n = 0 and 1.

We have $t(0) = 0$ (given)

And $t(1) = 2t(0) + 1 = 1$

So,

$$c_1 + c_2 = 0 \quad \ldots(1)$$

$$2c_1 + c_2 = 1 \quad \ldots(2)$$

By subtracting equation (1) from (2), we get

$$c_1 = 1$$

and $c_2 = -1$

Therefore the general solution is

$$t(n) = 2^n - 1^n$$

Thus $t_n \in \theta(2^n)$

Example 1.7 : *Solve the following recurrence*

$$t_n = \begin{cases} n+1 & \text{if } n = 0 \text{ or } 1 \\ 3t_{n-1} - 2t_{n-2} + 3 \times 2^{n-2} & \text{otherwise} \end{cases}$$

Express your answer as simply as possible using the θ notation.

Solution : Rewrite the recurrence as

$t_n - 3t_{n-1} + 2t_{n-2} = 3 \times 2^{n-2}$

i.e. $t_n - 3t_{n-1} + 2t_{n-2} = (3/4)2^n$

Here, $k = 2$, $a_0 = 1$, $a_1 = -3$, $a_2 = 2$ and $b = 2$, $p(n) = (3/4)$ which is a polynomial of degree 0.

The characteristic polynomial is

$$= (x^2 - 3x + 2)(x-2)$$

$$= (x-2)(x-1)(x-2)$$

$$= (x-1)(x-2)^2$$

The roots of the characteristic polynomial are $r_1 = 1$ with multiplicity 1 and $r_2 = 2$ with multiplicity 2.

The general solution is of the form

$$t_n = c_1 1^n + c_2 2^n + c_3 n 2^n$$

$$\therefore \quad t_n = c_1 + (c_2 + c_3 n)2^n$$

Thus, $t_n \in \theta(n2^n)$.

We can use the initial condition to find out the constants.

Example 1.8 : *Solve following recurrence:*

$$t_n = 2t_{n-1} + n \quad \ldots(1)$$

Express your answer as simply as possible in the θ notation.

Solution : Part A :

Rewrite the recurrence as

$t_n - 2t_{n-1} = n$

Here, $k = 1$, $a_0 = 1$, $a_1 = -2$ and $b = 1$, $p(n) = n$ which is a polynomial of degree 1.

The characteristic polynomial is

$$(x-2)(x-1)^2$$

The roots of the characteristic polynomial are $r_1 = 2$ with multiplicity 1 and $r_2 = 1$ with multiplicity 2. The general solution is of the form

$$t_n = c_1(2)^n + c_2(1)^n + c_3 n(1)^n \quad \ldots(2)$$

Part B : Provided that $t_0 \geq 0$.

For three constants c_1, c_2, c_3, we can use three initial conditions n = 0, 1, 2. We have

$$t_1 = 1 + 2t_0$$

$$t_2 = 2 + 2t_1 = 2 + 2(1 + 2t_0)$$

$$= 2 + 2 + 4t_0$$

$$\therefore \quad t_2 = 4 + 4t_0$$

Therefore n = 0, 1, 2, we get

$$c_1 + c_2 + 0 = t_0 \quad \ldots(3)$$

$$2c_1 + c_2 + c_3 = 1 + 2t_0 \quad \ldots(4)$$

$$4c_1 + 2c_2 + 2c_3 = 4 + 4t_0 \quad \ldots(5)$$

Doing multiplication of equation (4) by 2, we get

$$4c_1 + 2c_2 + 2c_3 = 2 + 4t_0$$

Subtracting it from equation (5), we get

$$-c_2 = 2$$

$$\therefore \quad c_2 = -2$$

and $c_1 = t_0 + 2$

Putting in equation (4), we get

$$2(t_0 + 2) - 2 + c_3 = 1 + 2t_0$$

$$\therefore \quad c_3 = 1 + 2t_0 - 2t_0 - 4 + 2$$

$$\therefore \quad c_3 = -1$$

Therefore the general solution is

$$t_n = (t_0+2)2^n - 2(1)^n - n(1)^n$$

$$\therefore \quad t_n = (t_0+2)2^n - 2 - n$$

Thus, $t_n \in \theta(2^n)$.

Alternative to Part B :

From equation (2), find out t_{n-1} and put in equation (1

$$\therefore \quad t_n = c_1 2^n + c_2 + c_3 n$$

and $t_{n-1} = c_1 2^{n-1} + c_2 + c_3(n-1)$

$$\therefore \quad n = t_n - 2t_{n-1} \quad \ldots(\text{gi}$$

$$= (c_1 2^n + c_2 + c_3 n) - 2(c_1 2^{n-1} + c_2 + c_3 n - c_3)$$

$$= c_1 2^n - 2c_1 2^n 2^{-1} + c_2 - 2c_2 + c_3 n - 2c_3 n + 2c_3$$

$$= -c_2 - c_3 n + 2c_3$$

$$\therefore \quad n = (2c_3 - c_2) - c_3 n$$

$$\therefore \quad 2c_3 - c_2 = 0 \text{ and } -c_3 = 1$$

$$\therefore \quad c_3 = -1 \text{ and } c_2 = -2$$

$$\therefore \quad t_n = c_1 2^n - 2 - n$$

Thus $t_n \in \theta(2^n)$

Example 1.9 : *Solve the following recurrence*

$$t_n = \begin{cases} 1 & \text{if } n = 0 \\ 4t_{n-1} - 2^n & \text{otherwise} \end{cases}$$

Express your answer as simply as possible in the θ notation.

Solution : Rewrite the recurrence as

$$t_n - 4t_{n-1} = -2^n \quad \ldots$$

Here, $k = 1$, $a_0 = 1$, $a_1 = -4$ and $b = 2$, $p(n) = -1$ which a polynomial of degree 0.

The characteristic polynomial is

$$(x-4)(x-2)$$

The roots of the characteristic polynomial are $r_1 = 4$, $r_2 =$

The general solution is of the form

$$t_n = c_1 4^n + c_2 2^n$$

Putting t_n and t_{n-1} in equation (1), we get

$$-2^n = t_n - 4t_{n-1}$$

$$= (c_1 4^n + c_2 2^n) - 4(c_1 4^{n-1} + c_2 2^{n-1})$$

$$= c_1 4^n + c_2 2^n - 4c_1 4^{n-1} - 4c_2 2^{n-1}$$

$$= c_2 2^n - 4c_2 2^n 2^{-1}$$

$$= c_2 2^n - 2c_2 2^n$$

$$\therefore \quad -2^n = -c_2 2^n$$

$$\therefore \quad c_2 = 1$$

$$\therefore \quad t_n = c_1 4^n + 2^n$$

For $n = 0$, we get

$$c_1 + 1 = t_0 = 1$$

$$\therefore \quad c_1 = 0$$

$$\therefore \quad t_n = 2^n$$

But $t_1 = 4t_0 - 2$

$\therefore$ For $n = 1$, we get

$\therefore \quad 4c_1 + 2 = 4t_0 - 2$

$\therefore \quad 4c_1 = 4t_0 - 4$

$\therefore \quad c_1 = t_0 - 1$

$\therefore \quad t_n = (t_0 - 1)\, 4^n + 2^n$

Thus $t_n \in \theta(4^n)$ only if $t_0 > 1$

Let us solve one example of the inhomogeneous recurrence of the case 2 form.

Example 1.10 : *Solve the following recurrence exactly:*

$$t_n = \begin{cases} n & \text{if } n = 0 \\ 2t_{n-1} + n + 2^n & \text{otherwise} \end{cases}$$

Express your answer as simply as possible in the θ notation.

Solution : Rewrite the recurrence as

$$t_n - 2t_{n-1} = n + 2^n$$

Here $k = 1$, $a_0 = 1$, $a_1 = -2$ and R.H.S. consists of two terms:

1) $b_1 = 1$ and $p_1(n) = n$ which is a polynomial of degree $d_1 = 1$.
2) $b_2 = 2$ and $p_2(n) = 1$ which is a polynomial of degree $d_2 = 0$.

Hence the characteristic polynomial is of the form

$$(x-2)(x-1)^2 (x-2)^1 = (x-1)^2 (x-2)^2$$

The roots of the characteristic polynomial are $r_1 = 1$ with multiplicity 2 and $r_2 = 2$ with multiplicity 2.

The general solution is of the form

$$t_n = c_1 1^n + c_2 n 1^n + c_3 2^n + c_4 n 2^n$$

We have to find whether value of $c_4 > 0$ or not to find out the exact order of t_n.

Putting t_n and t_{n-1} in the original recurrence, we get

$$\begin{aligned} n + 2^n &= (c_1 + c_2 n + c_3 2^n + c_4 n 2^n) - 2(c_1 + c_2(n-1) \\ &\quad + c_3 2^{n-1} + c_4 (n-1)\, 2^{n-1}) \\ &= c_1 + c_2 n + c_3 2^n + c_4 n 2^n - 2c_1 - 2c_2 n \\ &\quad + 2c_2 - 2c_3 2^{n-1} - 2c_4 n 2^{n-1} + 2c_4 2^{n-1} \\ &= -c_1 - c_2 n + 2c_2 + 2c_4 2^{n-1} \\ &= (2c_2 - c_1) - c_2 n + c_4 2^n \end{aligned}$$

$\therefore \quad c_4 = 1, -c_2 = 1$

$\therefore \quad c_2 = -1$

and $2c_2 - c_1 = 0$

$\therefore \quad c_1 = 2c_2$

$\therefore \quad c_1 = -2$

For $n = 0$, from the general solution we get

$c_1 + 0 + c_3 + 0 = 0$

$\therefore \quad c_3 = 2$

Therefore the general solution is

$$t_n = -2.1^n - n.1^n + 2.2^n + n.2^n$$

$\therefore \quad t_n = -2 - n + 2^{n+1} + n2^n$

Thus $t_n \in \theta(n2^n)$

Example 1.11 : *Solve the following recurrence exactly:*

$$T(n) = \begin{cases} a & \text{if } n = 0 \text{ or } n = 1 \\ T(n-1) + T(n-2) + c & \text{otherwise} \end{cases}$$

Express your answer as simply as possible using the θ notation and the golden ratio $\phi = (1 + \sqrt{5})/2$.

Solution : Rewrite the recurrence as

$$T(n) - T(n-1) - T(n-2) = c$$

Here $k = 2$, $a_0 = 1$, $a_1 = -1$, $a_2 = -1$ and $b = 1$, $p(n) =$ (which is a polynomial of degree 0.

The characteristic polynomial is

$(x^2 - x - 1)(x - 1)$ whose roots are

$$r_1 = \frac{1+\sqrt{5}}{2},\ r_2 = \frac{1-\sqrt{5}}{2},\ r_3 = 1$$

Therefore the general solution is of the form

$$T(n) = c_1 r_1^n + c_2 r_2^n + c_3 r_3^n$$

$$\therefore \quad T(n) = c_1 \left(\frac{1+\sqrt{5}}{2}\right)^n + c_2 \left(\frac{1-\sqrt{5}}{2}\right)^n + c_3 . 1^n$$

Thus $T(n) \in \theta(\phi^n)$, provided $c_1 \neq 0$.

Example 1.12 : *Solve the following recurrence exactly*

$$T_n = \begin{cases} a & \text{if } n = 0 \text{ or } n = 1 \\ T(n-1) + T(n-2) + cn & \text{otherwise} \end{cases}$$

Express your answer as simply as possible using the θ notation and the golden ratio $\phi = (1 + \sqrt{5})/2$.

Solution :

Rewrite the recurrence as

$$T(n) - T(n-1) - T(n-2) = cn$$

Here, $k = 2$, $a_0 = 1$, $a_1 = -1$, $a_2 = -1$ and $b = 1$, $p(n) = cn$ which is a polynomial of degree 1.

The characteristic polynomial is of the form

$$(x^2 - x - 1)(x-1)^2$$

whose roots are $r_1 = \frac{1+\sqrt{5}}{2}$, $r_2 = \frac{1-\sqrt{5}}{2}$, both with multiplicity 1 and $r_3 = 1$ with multiplicity 2. The general solution is of the form

$$\therefore \quad T(n) = c_1 \left(\frac{1+\sqrt{5}}{2}\right)^n + c_2 \left(\frac{1-\sqrt{5}}{2}\right)^n + c_3 1^n + c_4 n 1^n$$

$$\therefore \quad T(n) = c_1 \phi^n + c_2 \left(\frac{1-\sqrt{5}}{2}\right)^n + c_3 + nc_4$$

Thus $T(n) \in \theta(\phi^n)$, provided $c_1 \neq 0$.

Let us see some problems which are somewhat different.

Example 1.13 : *Solve the following recurrence*

$$t_n - 2t_{n-1} = 3^n \quad \ldots (1)$$

Express your answer as simply as possible using the θ notation.

Solution : Part A :

Here $b = 3$, $p(n) = 1$ that is a polynomial with degree 0.

Let us find homogeneous recurrence from equation (1).

By multiplying with 3, we get

$$3t_n - 6t_{n-1} = 3^{n+1}$$

By replacing n with $n - 1$, we get

$$3t_{n-1} - 6t_{n-2} = 3^n \quad \ldots (2)$$

Subtracting equation (2) from equation (1), we get

$$t_n - 5t_{n-1} + 6t_{n-2} = 0 \quad \ldots (3)$$

Equation (3) is a characteristic equation with

$k = 2$, $a_0 = 1$

$a_1 = -5$, $a_2 = 6$.

The characteristic polynomial is

$$= x^2 - 5x + 6$$

$$= (x-2)(x-3)$$

So the roots of the characteristic polynomial are $r_1 = 2$ and $r_2 = 3$.

The general solution is of the from

$$t_n = c_1 2^n + c_2 3^n \quad \ldots (4)$$

Part B : The initial conditions can be taken as t_0 and t_1.

From equation (1)

$$t_1 = 2t_0 + 3$$

Putting the initial conditions in equation (4), for $n = 0$ we get

$$c_1 + c_2 = t_0$$

and for $n = 1$, we get

$$2c_1 + 3c_2 = t_1$$

i.e. $2c_1 + 3c_2 = 2t_0 + 3$

$\therefore \quad 2(t_0 - c_2) + 3c_2 = 2t_0 + 3$

$\therefore 2t_0 - 2c_2 + 3c_2 = 2t_0 + 3$

$\therefore \quad c_2 = 3$

and $c_1 = t_0 - 3$

Therefore

$$t_n = (t_0 - 3)2^n + 3(3)^n$$

i.e. $t_n = (t_0 - 3)2^n + 3^{n+1}$

Thus $t_n \in \theta(3^n)$

Alternative to Part B :

From equation (4), find out t_{n-1} and put in equation (1)

$$\therefore (c_1 2^n + c_2 3^n) - 2(c_1 2^{n-1} + c_2 3^{n-1}) = 3^n$$

$$\therefore c_1 2^n + c_2 3^n - 2c_1 2^{n-1} - 2c_2 3^{n-1} = 3^n$$

$$\therefore c_2 3^n (1 - 2.3^{-1}) = 3^n$$

$$\therefore \quad c_2 (1 - 2/3) = 1$$

$$\therefore \quad c_2 (1/3) = 1$$

$$\therefore \quad c_2 = 3$$

for $n = 0$, equation (4) gives

$$c_1 + c_2 = t_0$$

$$\therefore \quad c_1 = t_0 - 3$$

Therefore the general solution is

$$t_n = (t_0 - 3)\, 2^n + 3(3)^n$$

$$\therefore \quad t_n = (t_0 - 3)\, 2^n + 3^{n+1}$$

Thus $t_n \in \theta(3)^n$

Example 1.14 : *Solve the following recurrence*

$$t_n - 2t_{n-1} = (n+5)\, 3^n,\ n \geq 1 \quad \ldots$$

Express the answer as simply as possible using the θ notation.

Solution : The given recurrence is

$$t_n - 2t_{n-1} = (n+5)\, 3^n$$

Replace n by n–1 and then multiply by – 6, we get

$$-6t_{n-1} + 12t_{n-2} = -6(n+4)\, 3^{n-1} \quad \ldots$$

Replace n by n-2 in equation (1) and then multiply it by we get

$$9t_{n-2} - 18t_{n-3} = 9(n+3)\, 3^{n-2} \quad \ldots$$

By adding equation (1), equation (2), and equation (3), get

$$\begin{aligned} t_n - 8t_{n-1} + 21t_{n-2} - 18t_{n-3} &= (n+5)\, 3^n - 6(n+4)3^{n-1} \\ &\quad + 9(n+3)3^{n-2} \quad \ldots \end{aligned}$$

Let us simplify R.H.S. of equation (4).

$$\begin{aligned} \text{R.H.S.} &= (n+5)\, 3^n - 6(n+4)\, 3^{n-1} + 9(n+3)3^n \\ &= 3^n [(n+5) - 6(n+4)3^{-1} + 9(n+3)3^{-2} \\ &= 3^n [(n+5) - 2(n+4) + (n+3)] \\ &= 3^n [n + 5 - 2n - 8 + n + 3] \\ &= 3^n (0) \\ &= 0 \end{aligned}$$

Therefore,

$$t_n - 8t_{n-1} + 21t_{n-2} - 18t_{n-3} = 0$$

Here, $k = 3$, $a_0 = 1$, $a_1 = -8$, $a_2 = 21$, $a_3 = -18$

The characteristic polynomial is,

$$x^3 - 8x^2 + 21x - 18$$

$$= (x-2)(x-3)^2$$

The roots of the characteristic polynomial are $r_1 = 2$ and $r_2 =$ with multiplicity 2.

The general solution is of the form

$$t_n = c_1 2^n + c_2 3^n + c_3 n 3^n \quad \ldots ($$

For three constants c_1, c_2, c_3 we can use the 3 initial conditions.

From equation (1), we have

$$t_1 = 2t_0 + (1 + 5)\,3$$

$$\therefore \quad t_1 = 2t_0 + 18$$

Also $$t_2 = 2t_1 + (2 + 5)\,3^2$$

$$\therefore \quad t_2 = 2(2t_0 + 18) + 63$$

$$\therefore \quad t_2 = 4t_0 + 99$$

Now putting the initial conditions

$n = 0, 1, 2$ in equation (5), we get

$$c_1 + c_2 + 0 = t_0$$

$$2c_1 + 3c_2 + 3c_3 = 2t_0 + 18$$

$$4c_1 + 9c_2 + 18c_3 = 4t_0 + 99$$

By solving these equations, we get

$$c_1 = t_0 - 9,\ c_2 = 9,\ c_3 = 3$$

Therefore the general solution is

$$t_n = (t_0 - 9)2^n + 9(3)^n + 3n3^n$$

$$\therefore \quad t_n = (t_0 - 9)2^n + (3)^{n+2} + n3^{n+1}$$

$$\therefore \quad t_n = (t_0 - 9)2^n + (n + 3)3^{n+1}$$

Thus $t_n \in \theta(n3^n)$

1.16.3 Change of Variable

There are some complicated recurrences which can be solved only by making a change of variable. In all our examples n is a power of 2, so we can replace n by 2^i. So $i = lg\ n$.

So the Steps to be Followed to Solve Such Recurrences are:

(1) First rewrite the recurrence by replacing n by 2^i. Then we obtain a new recurrence t_i.

(2) The new recurrence t_i can be written in the form of inhomogeneous recurrences.

(3) Then we can obtain the characteristic polynomial and its roots.

(4) Write the general solution for t_i.

(5) Replace i by log n to obtain the general solution for t(n).

(6) Solve the recurrence for constants.

(7) Express the answer in the θ notation.

Let us solve some examples of this kind.

Example 1.15 : *Solve the following recurrence exactly for n as power of 2*

$$T(n) = \begin{cases} 1 & , \text{ if } n = 1 \\ 4T(n/2) + n & , \text{ otherwise} \end{cases}$$

Express your answer as simply as possible in the θ notation.

Solution : Rewrite the recurrence by replacing n by 2^i. So $n/2 = 2^i/2 = 2^{i-1}$. So the original recurrence T(n) which is a function of T(n/2) is now redefined as t_i which is a function of t_{i-1}. The new recurrence t_i is

$$t_i = T(n) = T(2^i)$$

$$= 4T(2^{i-1}) + 2^i$$

$$= 4t_{i-1} + 2^i$$

which can be written as

$$t_i - 4t_{i-1} = 2^i$$

Here $k = 1, a_0 = 1, a_1 = -4$, and $b = 2$, $p(i) = 1$ which is a polynomial of degree 0.

The characteristic polynomial is ,

$$(x - 4)(x - 2)$$

whose roots are $r_1 = 4, r_2 = 2$.

The general solution is of the form

$$t_i = c_1 4^i + c_2 2^i$$

But $$n = 2^i$$

$$\therefore \quad i = lg\ n$$

$$\therefore \quad T(n) = t_i = t\ lg\ n$$

$$= c_1 4^{lg\,n} + c_2 2^{lg\,n}$$

$$= c_1 (2^2)^{lg\,n} + c_2 n$$

$$= c_1 (2^{lg\,n})^2 + c_2 n$$

$$= c_1 n^2 + c_2 n$$

Putting T(n) and T(n/2) in the original recurrence, we get

$$n = T(n) - 4\,T(n/2)$$

$$= (c_1 n^2 + c_2 n) - 4\,[(n/2)^2 c_1 + c_2\, n/2]$$

$$= c_1 n^2 + c_2 n - c_1 n^2 - 2c_2 n$$

$$= -\,c_2 n$$

$$\therefore \quad -\,c_2 = 1$$

$$\therefore \quad c_2 = -1$$

But for $n = 1$, we have

$$c_1 + c_2 = 1$$

$$\therefore \quad c_1 = 1 - c_2 = 2$$

Therefore

$$T(n) = 2n^2 - n$$

Thus $T(n) \in \theta(n^2 \mid n \text{ is power of 2})$

Example 1.16 : *Solve the following recurrence exactly*

$$T(n) = 4\,T(n/2) + n^2$$

when n is a power of 2, $n \geq 2$.

Express your answer as simply as possible in the θ notation.

Solution : Rewrite the recurrence by replacing n by 2^i. So $n/2 = 2^i/2 = 2^{i-1}$.

The original recurrence T(n) which is defined as a function of T(n/2) is now redefined as t_i which is a function of t_{i-1}. The new recurrence t_i is

$$t_i = T(2^i) = 4\,T(2^{i-1}) + (2^i)^2$$

$$\therefore \quad t_i = 4t_{i-1} + 4^i$$

which can be written as

$$\therefore \quad t_i - 4t_{i-1} = 4^i$$

Here, $k = 1, a_0 = 1, a_1 = -4$, and $b = 4$, $p(i) = 1$ which is a polynomial of degree 0.

So the characteristic polynomial is

$$(x - 4)(x - 4) = (x - 4)^2$$

Whose roots are $r_1 = 4$ with multiplicity 2.

The general solution is of the form

$$t_i = c_1 4^i + c_2 i\, 4^i$$

But $$n = 2^i$$

$$\therefore \quad i = lg\ n$$

So the original recurrence is

$$T(n) = t_i = t_{lg\ n}$$

Therefore the general solution becomes

$$T(n) = c_1 4^{lg\,n} + c_2 . lg\ n\ .\ 4^{lg\,n}$$

$$= c_1 (2^2)^{lg\,n} + c_2\, lg\ n\ (2^2)^{lg\,n}$$

$$= c_1 (2^{lg\,n})^2 + c_2\, lg\ n\ (2^{lg\,n})^2$$

$$= c_1 n^2 + c_2\, lg\ n\ .\ n^2$$

Putting T(n) and T(n/2) in the original recurrence, we get

$$n^2 = T(n) - 4\,T(n/2)$$

$$= (c_1 n^2 + c_2 n^2\, lg\ n) - 4[c_1\, n^2\, (n/2)^2 + c_2 (n/2)^2\, .\, lg\ (n/2)]$$

$$= (c_1 n^2 + c_2 n^2\, lg\ n) - 4[c_1\, n^2/4 + c_2 n^2/4\, .\, lg\ n/2]$$

$$= c_1 n^2 + c_2 n^2\, lg\ n - c_1 n^2 - c_2 n^2\, lg\ (n/2)$$

$$= c_2 n^2\, [lg\ n - lg\ (n/2)]$$

$$= c_2 n^2\, [lg\ (n \div n/2)]$$

$[\because lg\ x - lg\ y = log\ (x \div y)]$

$$= c_2 n^2\, lg\ (n \times 2/n)$$

$$= c_2 n^2\, lg_2^2$$

$$= c_2 n^2\, .\, 1$$

$$= c_2 n^2$$

$$\therefore \quad c_2 = 1$$

Thus $$T(n) = c_1 n^2 + n^2\, lg\ n$$

Therefore regardless of the c_1 value,

$T(n) \in \theta(n^2\ lg\ n \mid n \text{ is a power of 2})$

Example 1.17 : *Solve the following recurrence exactly:*

$$T(n) = 2T(n/2) + n\ lg\ n$$

when, n is a power of 2, $n \geq 2$.

Express your answer as simply as possible in the θ notation.

Solution : Rewrite the recurrence by replacing n by 2^i. So $n/2 = 2^i/2 = 2^{i-1}$. The original recurrence T(n) which is a function of T(n/2) can be redefined as t_i which is a function of t_{i-1}.

The new recurrence t_i is

$$t_i = T(n) = T(2^i)$$

$$= 2T(2^{i-1}) + 2^i\, lg\ 2^i$$

$$= 2t_{i-1} + i2^i$$

which can be written as

$$t_i - 2t_{i-1} = i2^i$$

Here, $k = 1, a_0 = 1, a_1 = -2$, and $b = 2$, $p(i) = i$ which polynomial of degree 1.

The characteristic polynomial is

$$(x - 2)(x - 2)^2 = (x - 2)^3$$

whose roots are $r_1 = 2$ with multiplicity 3.

The general solution is

$$t_i = c_1 2^i + c_2 i 2^i + c_3 i^2 2^i$$

But $$n = 2^i$$

$$\therefore \quad i = lg\ n$$

Hence the original recurrence

$$T(n) = t_i = t_{lg\,n}$$

$$\therefore \quad T(n) = c_1 2^{lg\,n} + c_2 lg\ n\ .\ 2^{lg\,n} + c_3 (lg\ n)^2 2^{lg\,n}$$

$$\therefore \quad T(n) = c_1 n + c_2 n\ lg\ n + c_3 n\ lg^2\ n$$

Let us put T(n) and T(n/2) in the original recurrence.

$$\therefore \quad n\ lg\ n = T(n) - 2\,T(n/2)$$

$$= (c_1 n + c_2 n\ lg\ n + c_3 n\ lg^2\ n) - 2[c_1 n/2 + c_2 n/2\ lg\ (n/2) + c_3\, n/2\ lg^2(n/2)]$$

$$= c_1 n + c_2 n\ lg\ n + c_3 n\ lg^2\ n - c_1 n - c_2 n\ lg\ (n/2) - c_3\, n\ lg^2(n/2)$$

$$= c_2 n[lg\ n - lg\ (n/2)] + c_3 n\ [lg^2\ n - lg^2\ (n/2)]$$

$$= c_2 n\ [lg(n \div n/2)] + c_3 n\ [(lg\ n + lg\ (n/2)) \times (lg\ n - lg(n/2))]$$

$[\because (a^2 - b^2) = (a + b)(a -$

$$= c_2 n\ [lg(n \times 2/n)] + c_3 n\ [(lg(n \times n/2)) \times (lg\ (n \div n/2)]$$

$$= c_2 n\ lg_2^2 + c_3 n\ [(lg\ (n^2/2))\ (lg\ (n \times 2/n$$

$$= c_2 n\ .\ 1 + c_3 n\ [(lg\ n_2 - lg_2^2)\ (lg_2^2)]$$

$$= c_2 n + c_3 n\ [(2\ lg\ n - 1)\ .\ 1]$$

$$= c_2 n + 2c_3 n\ lgn - c_3 n$$

$$= c_2 n - c_3 n + 2c_3 n\ lg\ n$$

$$= (c_2 - c_3)\, n + 2c_3 n\ lg\ n$$

$$\therefore \quad 2c_3 = 1$$

$$\therefore \quad c_3 = 1/2$$

and $$c_2 - c_3 = 0$$

$$\therefore \quad c_2 = c_3 = 1/2$$

Therefore, $$T(n) = c_1 n + \frac{n\ lg\ n}{2} + \frac{n\ lg^2 n}{2}$$

Thus $T(n) \in \theta(n\ lg^2\ n \mid n \text{ is a power of 2})$ regardless of the ini conditions.

Example 1.18 : *Solve the following recurrence exactly.*

$$t_n = \begin{cases} 1 & \text{if } n = 1 \\ 3T(n/2) + n & \text{if n is a power of 2 } n > 1 \end{cases}$$

Express your answer as simply as possible in the θ notation.

Solution : We change a variable n by 2^i so the new recurrence t_i is defined as $t_i = T(2^i)$

Therefore,

$$n/2 = 2^i / 2 = 2^{i-1}$$

So the original recurrence T(n) which is defined as a function of T(n/2) is now redefined as t_i which is a function of t_{i-1}.

The new recurrence t_i is

$$t_i = T(2^i) = 3T(2^{i-1}) + 2^i$$
$$= 3t_{i-1} + 2^i$$
$$\therefore \quad t_i - 3t_{i-1} = 2^i$$

This is an inhomogeneous recurrence where k = 1, $a_0 = 1$, $a_1 = -3$, and b = 2, p(i) = 1 which is a polynomial of degree 0.

The characteristic polynomial is

$$(x-3)(x-2)$$

whose roots are $r_1 = 3$, $r_2 = 2$.

The general solution is of the form

$$t_i = c_1 3^i + c_2 2^i$$

But $n = 2^i$

$$\therefore \quad i = lg\, n$$

$\therefore$ The original recurrence is

$$T(n) = T(2^i) = t_i$$
$$\therefore \quad T(n) = t_{lg\, n}$$

Therefore the general solution is

$$T(n) = c_1 3^{lg\, n} + c_2 2^{lg\, n}$$
$$\therefore \quad T(n) = c_1 n^{lg\, 3} + c_2 n$$

where, n is a power of 2.

Solving for n = 1 and n = 2, we get

$$c_1 + c_2 = 1 \qquad [\because 1^{log\, 3} = 1] \qquad \ldots (1)$$
$$c_1 2^{lg\, 3} + 2c_2 = T(2)$$
$$= 3T(1) + 2 = 3 \times 1 + 2 = 5$$
$$\therefore \quad 3c_1 + 2c_2 = 5 \qquad \ldots (2)$$

Multiply by 2 to equation (1), we get

$$2c_1 + 2c_2 = 2$$

Subtracting it from equation (2), we get

$$c_1 = 3 \quad \text{and} \quad c_2 = -2$$

Therefore the general solution is of the form

$$T(n) = 3n^{lg\, 3} - 2n$$

where n is a power of 2.

Thus $T(n) \in \theta(n^{lg\, 3})$ [n is a power of 2)

[Note that $(1/2)^{lg\, 3} = 1/3$]

Example 1.19 : *Solve the following recurrence used for the analysis of divide-and-conquer algorithms:*

The constants $n_0 \geq 1$, $l \geq 1$, $b \geq 2$ and $k \geq 0$ are integers, whereas c is a strictly positive real number. Let T: $N \to R^+$ be an eventually non-decreasing function such that

$$T(n) = lT(n/b) + cn^k, \; n > n_0$$

where, n/n_0 is an exact power of b, that is, $n \in \{bn_0, b^2 n_0, b^3 n_0, \ldots\}$.

Solution : Here n is a power of b. So we can replace n with $b^i n_0$

$$\therefore \quad n = b^i n_0$$

So the new recurrence t_i is

$$t_i = T(b^i n_0)$$
$$= lT(b^i n_0 / b) + c(b^i n_0)^k$$
$$= lT(b^{i-1} n_0) + cn_0^k b^{ik}$$
$$= lt_{i-1} + cn_0^k b^{ik}$$

The original recurrence T(n) is a function of T(n/b) which is redefined as t_i which is a function of t_{i-1}.

The new recurrence can be written as

$$t_i - lt_{i-1} = cn_0^k (b^k)^i$$

where, k = 1, $a_0 = 1$, $a_1 = -l$. R.H.S. is of the form $a^i p(i)$ where $a = b^k$ and $p(i) = cn_0^k$ which is a polynomial of degree 0.

The characteristic polynpmial is

$$(x-l)(x-b^k)$$

General solution is of the form

$$t_i = c_1 l^i + c_2 (b^k)^i$$

Now $n = b^i n_0$

$$b^i = n/n_0$$
$$\therefore \quad i = \log_b (n/n_0)$$

Also for any positive value v,

$$v^i = v^{\log_b (n/n_0)} = (n/n_0)^{\log_b v}$$
$$\therefore \quad T(n) = c_1 l^{\log_b (n/n_0)} + c_2 (b^k)^{\log_b (n/n_0)}$$
$$= c_1 (n/n_0)^{\log_b l} + c_2 (b^{\log_b (n/n_0)})^k$$
$$= \left(\frac{c_1}{n_0^{\log_b l}}\right) n^{\log_b l} + c_2 \left[(n/n_0)^{\log_b b}\right]^k$$
$$= \left(\frac{c_1}{n_0^{\log_b l}}\right) n^{\log_b l} + c_2 (n/n_0)^k$$
$$= c_{11} n^{\log_b l} + (c_2/n_0^k) n^k$$
$$\therefore \quad T(n) = c_{11} n^{\log_b l} + c_{22} n^k \qquad \ldots (1)$$

Here the constant terms are replaced by c_{11} and c_{22} respectively. Putting T (n) and T(n/b) in the original recurrence, we get

$$cn^k = T(n) - lT(n/b)$$
$$= (c_{11} n^{\log_b l} + c_2 n^k) - l\left[c_{11}\left(\frac{n}{b}\right)^{\log_b l} + c_{22}\left(\frac{n}{b}\right)^k\right]$$

$$= c_{11} n^{\log_b l} + c_{22} n^k$$
$$= \frac{l c_{11}}{l} n^{\log_b l} - lc_{22}\left(\frac{n^k}{b^k}\right)$$
$$= c_{22} n^k - lc_{22}\left(\frac{n^k}{b^k}\right)$$
$$= \left(1 - \frac{l}{b^k}\right) c_{22} n^k$$
$$\therefore \quad \left(1 - \frac{l}{b^k}\right) c_{22} = c$$
$$c_{22} = \frac{c}{1 - l/b^k}$$

Now we have to find out which term in the general solution of T(n) is dominant. Consider three cases to compare l and b:

- **Case 1 :** if $l < b^k$, then $(l / b^k) < 1$ means $(1 - l / b^k)$ is positive and c is also positive (given). Therefore the term $c_{22} > 0$ and $k > \log_b l$. Therefore the term $c_{22} n^k$ is dominant in T(n). Therefore $T(n) \in \theta(n^k (n/n_0)$ is a power of b).
- **Case 2 :** if $l > b^k$, then $l / b^k > 1$ means $(1 - l / b^k)$ is negative but c is positive. So c_{22} is negative that is $c_{22} < 0$ and $\log_b l > k$. It is given that $\log_b l$: $N \to R^+$. So T(n) is positive, but c_{22} is negative, means the first term is positive and c_{11} is positive. Therefore form first term $c_{11} n^{\log_b l}$ is dominant and $T(n) \in \theta(n^{\log_b l})$.
- **Case 3 :** if $l = b^k$, then $l/b^k = 1$ means $(1 - l/b^k) = 0$. So there is divide by zero problem. Hence equation (1) does not give the general solution for this case.

  The characteristic polynomial becomes $(x - b^k)^2$

  whose root are $r_1 = b^k$ with multiplicity 2. The general solution is

  $$t_i = c_3 (b^k)^i + c_4 i (b^k)^i$$

  But $i = \log_b (n / n_0)$

  $$\therefore \quad T(n) = c_3 (b^k)^{\log_b (n/n_0)} + c_4 \cdot \log_b (n/n_0) (b^k)^{\log_b (n,n_0)}$$
  $$= c_3 (b^{\log_b (n/n_0)})^k + c_4 \cdot \log_b (n/n_0) (b^{\log_b (n,n_0)})^k$$
  $$= c_3 \left(\frac{n}{n_0}\right)^k + c_4 \log_b \left(\frac{n}{n_0}\right) \cdot \left(\frac{n}{n_0}\right)^k$$
  $$= \left(\frac{c_3}{n_0^k}\right) n^k + \left(\frac{c_4}{n_0^k}\right) n^k \log_b \left(\frac{n}{n_0}\right)$$
  $$\therefore \quad T(n) = c_{33} n^k + c_{44} n^k \log_b \left(\frac{n}{n_0}\right)$$

Putting T(n) and T(n/b) in the original recurrence, we get

$$cn^k = T(n) - l\, T(n/b)$$
$$= [C_{33} n^k + c_{44} n^k \log_b \left(\frac{n}{n_0}\right)] - b^k [c_{33}\left(\frac{n}{b}\right)^k + c_{44}\left(\frac{n}{b}\right)^k \log_b \left(\frac{n}{n_{0b}}\right)]$$
$$= c_{33} n^k + c_{44} n^k \log_b \left(\frac{n}{n_0}\right) - c_{33} n^k - c_{44} n^k \log_b \left(\frac{n}{n_{0b}}\right)$$
$$= c_{44} n^k [\log_b \left(\frac{n}{n_0}\right) - \log_b \left(\frac{n}{n_{0b}}\right)]$$
$$= c_{44} n^k$$
$$\therefore \quad c_{44} = c \text{ which is positive.}$$

Therefore the second term $cn^k \log_b (n/n_0)$ becon dominant. Thus

$$T(n) \in \theta(n^k \log n)$$

Finally, we have

$$T(n) \in \begin{cases} \theta(n^k) & \text{if } l < b^k \\ \theta(n \log_b l) & \text{if } l > b^k \\ \theta(n^k \log n) & \text{if } l = b^k \end{cases}$$

1.16.4 Range Transformations

In the previous section, we have used change of a variable solve the recurrence. By making change of a variable, perform domain transformation. We can also perform rar transformation to solve the recurrence. In some cases, we h to perform both domain transformation and rar transformation. Let us solve one problem of this kind.

Example 1.20 : *Solve the following recurrence exactly when a power of 2.*

$$T(n) = \begin{cases} 1/3 & \text{if } n = 1 \\ nT^2(n/2) & \text{otherwise} \end{cases}$$

Solution : Rewrite the recurrence by making a change variable. Replace n by 2^i. So i = lg n. The new recurrence t_i is

$$t_i = T(2^i) = 2^i T^2 (2^i / 2)$$
$$= 2^i T^2 (2^{i-1})$$
$$= 2^i t^2_{i-1} \qquad \ldots$$

Now we have to perform range transformation. Let $u_i = lg$ again we get new recurrence

$$u_i = lg\, t_i = lg(2^i t^2_{i-1})$$
$$= lg\,(2^i) + 2lg(t_{i-1})$$
$$= i + 2\, u_{i-1}$$

which can be written as

$$u_i - 2u_{i-1} = i \qquad \ldots$$

where, k =1, $a_0 = 1$, $a_1 = -2$.

R.H.S. gives b = 1, p (i) = i which is a polynomial of degree 1

The characteristic polynomial is

$$(x-2)(x-1)^2$$

whose roots are $r_1 = 2$ with multiplicity 1 and $r_2 = 1$ with multiplicity 2. The general solution is

$$u_i = c_1 2^i + c_2 1^i + c_3 i 1^i$$
$$= c_1 2^i + c_2 + c_3 i$$

Putting u_i and u_{i-1} in equation (2), we get

$$i = u_i - 2u_{i-1}$$
$$= (c_1 2^i + c_2 + c_3 i) - 2[c_1 2^{i-1} + c_2 + c_3(i-1)]$$
$$= c_1 2^i + c_2 + c_3 i - c_1 2^i - 2c_2 - 2c_3 i + 2c_3$$
$$= -c_2 - c_3 i + 2c_3$$
$$\therefore \quad i = (2c_3 - c_2) - c_3 i$$
$$\therefore \quad -c_3 = 1$$
$$\therefore \quad c_3 = -1$$
$$\text{and} \quad 2c_3 - c_2 = 0$$
$$\therefore \quad c_2 = 2c_3 = -2$$

Therefore we have

$$u_i = c_1 2^i - 2 - i$$
$$\text{But} \quad u_i = lg\, t_i$$
$$\therefore \quad t_i = 2^{u_i} = 2^{c_1 2^i - 2 - i}$$
$$\text{and} \quad T(n) = t_i = t_{lg\, n}$$
$$= 2^{c_1 2^{lg\, n} - 2 - lg\, n} \qquad [\because i = lg\, n]$$
$$= 2^{c_1 n} \cdot 2^{-2} \cdot 2^{-lg\, n}$$
$$= \frac{2^{c_1 n}}{2^2\, 2^{lg\, n}}$$
$$\therefore \quad T(n) = \frac{2^{c_1 n}}{4n}$$

1.16.5 Asymptotic Recurrences

In the analysis of algorithms, to find the running time of the divide and conquer algorithms the recurrence of the following form is used :

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

where, $a \geq 1$ and $b > 1$ are constants. f (n) is positive and is of the exact order of n. Such recurrences are called as an asymptotic recurrences. This recurrence actually describes that the problem of size n is divided into 'a' subproblems, where each subproblem is of size n / b. The cost of dividing the problem and combining the results of all the subproblems is described by the function f (n).

Generally to find the solution of an asymptotic recurrence, method is to sandwich its function between two simpler recurrences. After solving these two simpler recurrences, their solutions are compared. If both have the same asymptotic solution, then the asymptotic recurrence must have the same solution. The master theorem is available to solve such asymptotic recurrences which gives the solution directly.

1.16.6 The Substitution Method

To solve recurrence T(n), this method repeatedly makes substitution for each occurrence of the function T in the right hand side until all such occurrences disappear.

Let us see an example.

Example 1.21 : *Solve the following recurrence using the substitution method.*

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

Solution :

$$T(n) = 2T(n/2) + n$$
$$= 2[2T(n/4) + n/2] + n$$
$$= 4T(n/4) + 2n$$
$$= 4[2T(n/8) + n/4] + 2n$$
$$= 8T(n/8) + 3n$$
$$= 8[2T(n/16) + n/8] + 3n$$
$$= 16T(n/16) + 4n$$

In general,

$$T(n) = 2^i T(n/2^i) + i\,n, \text{ for any } \log_2 n \geq i \geq 1.$$

Putting $i = \log_2 n$, we get

$$\therefore \quad T(n) = 2^{\log_2 n} . T(n/2^{\log_2 n}) + n \log_2 n$$
$$\therefore \quad T(n) = nT(1) + n \log_2 n$$
$$\text{But} \quad T(1) = 2$$
$$\therefore \quad T(n) = 2n + n \log_2 n$$
$$\text{Thus} \quad T(n) \in \theta(n \log_2 n)$$

Example 1.22 : *Solve the following recurrence when n is power of 2 :*

$$T(n) = \begin{cases} T(1) & ; \quad \text{if } n = 1 \\ T(n/2) + c & ; \quad \text{if } n > 1 \end{cases}$$

Solution :

$$T(n) = T(n/2) + c$$
$$= [T(n/4) + c] + c$$
$$= T(n/4) + 2c$$
$$= [T(n/8) + c] + 2c$$
$$= T(n/8) + 3c$$

.
.
.

In general, $T(n) = T(n/2^i) + ic$

Putting $i = \log_2 n$, we get

$$T(n) = T(n/2^{\log_2 n}) + c \log_2 n$$
$$= T(1) + c \log_2 n$$

As T (1) is assumed to be a constant,

$$T(n) \in \theta(\log n)$$

1.17 RECURSION TREE

In recurrence tree method to solve recurrences, each subproblem is represented by a node. Each node is expanded further. First levelvise cost is computed by adding cost at each level separately. Then total cost is obtained by adding all the levelvise costs.

Example 1.23 : *Use a recursion tree to find the asymptotic upper bound on the recurrence*

$$t(n) = 2T\left(\frac{n}{2}\right) + cn$$

Verify your answer using the substitution method.

Solution :

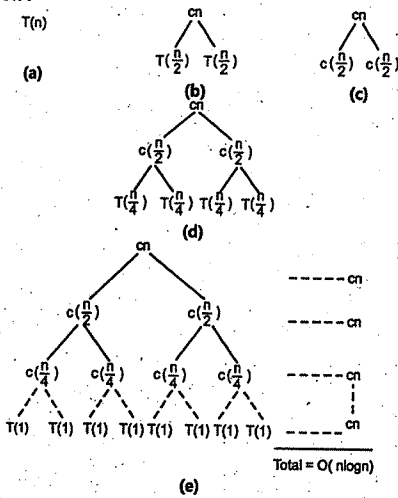


Fig. 1.7 : Construction of recursion tree for $T(n) = 2T\left(\frac{n}{2}\right) + cn$

Addition of cost of each level in Fig. 1.7 (e) is cn. The complete tree will have $\log_2 n$ levels. Hence total cost is $n \log_2 n$ which is an asymptotic upper bound for the given recurrence.

This can be verified using the substitution method.

$$T(n) = 2T\left(\frac{n}{2}\right) + cn$$
$$= 2\left[2T\left(\frac{n}{4}\right) + C\frac{n}{2}\right] + cn$$
$$= 4T\left(\frac{n}{4}\right) + 2cn$$
$$= 8T\left(\frac{n}{8}\right) + 3cn$$
$$= 2^k T\left(\frac{n}{2^i}\right) + icn$$

Let $2^i = n \quad i = \log_2 n$

$$T(n) = n\,T(1) + cn \log_2 n$$
$$\approx O(n \log n)$$

1.18 MASTER THEOREM

1.18.1 Formulation and Solving Recurrence Equations Using Master Theorem

The master theorem used to solve recurrence equations stated as follows:

Master Theorem:

Let $a \geq 1$ and $b > 1$ be constants, let f(n) be a function, and T(n) be defined on the non-negative integers by recurrence

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

where n/b is either $\lceil n/b \rceil$ or $\lfloor n/b \rfloor$. Then T(n) can be asymptotically bounded as follows:

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

Case 2 : If $f(n) = \theta(n^{\log_b a})$, then $T(n) = \theta(n^{\log_b a} \lg n)$.

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

Example 1.24: *Give asymptotic bounds for the following recurrences using the master theorem.*

1) $T(n) = 16\,T(n/4) + n$

Solution: Here $a = 16$, $b = 4$, $f(n) = n$

$$n^{\log_b a} = n^{\log_4 16} = n^{\log_4 4^2} = n^{2 \log_4 4} = n^2$$
$$\therefore \quad n^{\log_b a} = \theta(n^2)$$

As $f(n) = O(n^{\log_4 16 - \epsilon})$ for $\epsilon = 1$, case 1 of master theorem can be applied.

Hence the solution

$$T(n) = \theta(n^{\log_4 16}) = \theta(n^2)$$

2) $T(n) = T(3n/4) + 1$

Solution: Here $a = 1$, $b = 4/3$, $f(n) = 1$

$$n^{\log_b a} = n^{\log_{4/3} 1} = n^0 = 1$$

As $f(n) = 1$ and $\theta(n^{\log_b a}) = 1$, hence case 2 can be applied here. Hence

$$T(n) = \theta(n^{\log_b a} \lg n)$$
$$\therefore \quad T(n) = \theta(\lg n) \qquad \because n^{\log_b a} = 1$$

3) $T(n) = 16\,T(n/4) + n^2$

Solution: Here $a = 16$, $b = 4$, $f(n) = n^2$

$$n^{\log_b a} = n^{\log_4 16} = n^2$$
$$\therefore \quad n^{\log_b a} = \theta(n^2)$$

and $\quad f(n) = \theta(n^2)$

$\therefore$ Case 2 of master theorem can applied.

Hence $T(n) = \theta(n^{\log_b a} \lg n)$

$\therefore T(n) = \theta(n^2 \lg n)$

4) $T(n) = 16\,T(n/4) + n^3$

Solution: Here $a = 16, b = 4, f(n) = n^3$

$n^{\log_b a} = n^{\log_4 16} = n^2$

$\therefore n^{\log_b a} = O(n^2)$

Since $f(n) = \Omega(n^{\log_4 16 + \epsilon})$ for $\epsilon \approx 0.2$

Case 3 applies.

Let $a\,f(n/b) = 16f(n/4)$

$= 16(n/4)^3)$

$= 16(n^3/64)$

and $cf(n) = (3/4)n^3$ for $c = 3/4$

For sufficiently large n,

$af(n/b) \le cf(n)$

Hence case 3 can be applied here. Hence solution to the recurrence is

$T(n) = \theta(f(n))$

$\therefore T(n) = \theta(n^3)$.

5) $T(n) = 2T(n/3) + n \lg n$

Solution: Here $a = 2, b = 3, f(n) = n \lg n$

$n^{\log_b a} = n^{\log_3 2} = O(n^{0.631})$

As $f(n) = \Omega(n^{\log_3 2 + \epsilon})$ for $\epsilon \approx 0.3$, case 3 can be applied here.

$af(n/b) = 2(n/3) \lg (n/3)$ and

$cf(n) = 2/3 n \lg n$ for $c = 2/3$

For sufficiently large n, $af(n/b) \le cf(n)$. Hence solution for the given recurrence according to case 3 is

$T(n) = \theta(f(n)) = \theta(n \lg n)$

Example 1.25: *Use the master method to show that the solution to the binary search recurrence $T(n) = T(n/2) + \theta(1)$ is*

Solution: Here $a = 1, b = 2, f(n) = \theta(1)$.

$n^{\log_b a} = n^{\log_2 1} = n^0 = \theta(1)$ which is equal to $f(n)$.

Hence case 2 can be applied here. So the solution to binary search recurrence is

$T(n) = \theta(n^{\log_b a} \lg n) = \theta(\lg n)$ as $n^{\log_b a} = \theta(1)$

Example 1.26: *Can the master method be applied to the recurrence $T(n) = 4T(n/4) + n \lg n$? Why or why not ?*

Solution: Here $a = 4, b = 4, f(n) = n \lg n$.

$n^{\log_b a} = n^{\log_4 4} = n$.

But case 3 cannot be applied here.

$$\frac{f(n)}{n^{\log_b a}} = \frac{n \lg n}{n} = \lg n$$

which is asymptotically n^c for $c > 0$.

Hence the recurrence does not follow case 2 nor case 3. It falls between case 2 and case 3.

Example 1.27: *Can the master method be applied to the recurrence $T(n) = 4T(n/2) + n^2 \lg n$? Give the reason.*

Ans.: Here $a = 4, b = 2, f(n) = n^2 \lg n$

$n^{\log_b a} = n^{\log_2 4} = n^2$

But the ratio

$$\frac{f(n)}{n^{\log_b a}} = \frac{n^2 \lg n}{n^2} = \lg n$$

which is asymptotically less than n^c for $c > 0$. Hence the recurrence falls between case 2 and case 3. Hence the master method cannot be applied to solve it.

1.19 HEAP AND HEAP SORT INTRODUCTION

We have studied binary search trees. In practice Binary Search Tree(BST) are rarely used to sort data. In the case when there is a fixed amount of data, and sorting does not need to take place until all the data is collected, then the data can be placed in an array and sorted using the quick sort algorithm. On the other hand, when data must be simultaneously inserted and sorted, then there is a data structure which in practice works more efficiently than Binary Search Trees namely a heap.

A **Heap** is a complete binary tree T whose nodes satisfy the following ordering property:

For every node M of T

M.Data≥max {C.data where C is a child of M}

Following Fig. 1.8 shows a heap.

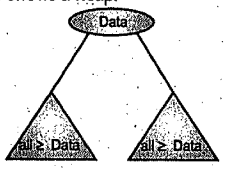


Fig. 1.8 : A Min-Heap

The structure shown above in Fig. 1.8 is called as **Min-Heap**. The property of a heap, the key value is lesser than the keys of the subtrees, can be reversed to create a **Max-Heap.**

That is we can create a maximum heap in which the key value in a node is greater than the key values in all of its subtrees. Generally, whenever the term heap is used by itself, it refers to a Max-Heap as shown in Fig. 1.9.

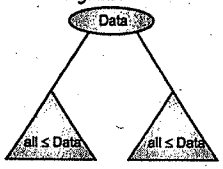


Fig. 1.9 : A max-heap

1.19.1 Definition

Heap Trees (or just **Heaps**) are a form of binary tree in which each node is greater than or equal to both of its children. Thus, the largest element in the entire tree is always the root of the tree.

Heaps are based on the notion of a **Complete Tree.**

Formally, a binary heap tree must satisfy two properties:

1. Structure property, and
2. Heap-order property.

1. Structure Property:

A binary tree is completely full if it is of height, h and has $2^{h+1} - 1$ nodes.

A binary tree of height, h, is **Complete** iff

(a) It is empty, or

(b) Its left subtree is complete of height $h - 1$ and its right subtree is completely full of height $h - 2$ or,

(c) Its left subtree is completely full of height $h - 2$ and its right subtree is completely full of height $h - 1$.

A complete tree is filled from the left:

- All the leaves are on the same level or Two adjacent ones
- All nodes at the lowest level are as far to the left as possible.

2. Heap Property:

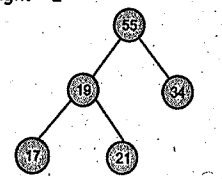
- A binary tree has the heap property iff
  (a) It is empty or
  (b) The key in the root is larger than in either child and both subtrees have the heap property.
- Although it is not necessary, it is often quite useful to define the heap as both **Full** or **Dense**. Fullness and density are qualities that both apply to binary trees and most trees in general as well. In a **Full Binary Tree,** all of the nodes with less than two children are in the bottom two levels.
- A binary tree is **Dense** when it is a full tree such that, in the second last level, all nodes with two children will be to the left of a node with one child, and a node with only one child will come to the left of the nodes with no children.
- The final requirement for density is that there is no more than one node in the second to last level that has only one child, and that one child must be the node's left child. Density is also sometimes called **Leftness** (because all the nodes in the last level are shifted to the left).
- In addition, the heap is usually defined so that only the **Largest** element (that is, the root) will be removed at a time. This makes the heap useful for scheduling and prioritizing. In fact, one of the two main uses of the heap is as a **Priority Queue,** which helps systems decide what to do next.
- Implementing and programming this structure is not as difficult as it was with a normal binary search tree because the denseness and fullness allow us to conveniently represent the heap with an **array**. In a **0-index** (the first element has the index 0), a node at the has a parent node at $(n - 1)/2$, rounded down. Tl appeal of the heap: it is fast, efficient, and minimal storage space.
- Other than as a priority queue, the heap has o important usage: Heap Sort. **Heap Sort** is one fastest sorting algorithms, achieving speed as Quicksort and mergesort. The advantages of heap that it does not use recursion and that heap so just fast for any data order. That is, there is basi worst case scenario.

1.19.2 An Example of Heap

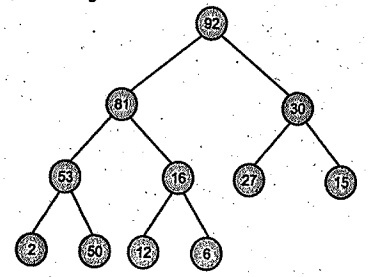
To complete our understanding, let us look at few exan Max-Heap shown in Fig. 1.10.

(1) Heap with one element (2) Heap with height

(3) Heap with height = 2



(4) Heap with height = 3



(5) An example of Min-Heap

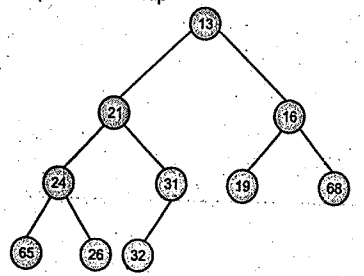
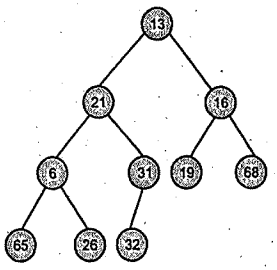


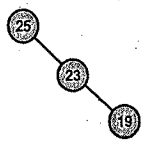
Fig. 1.10 : Sample Heaps

Let us consider now Fig. 1.11 which shows the binary trees which are not heaps.

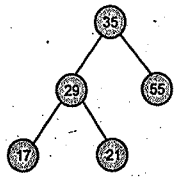
(1) Not nearly complete



(3) Not a Min-Heap: Min not at root



(2) Root not largest



(4) Subtree with root node 21 is not a heap

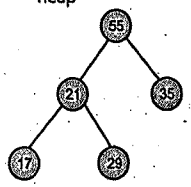


Fig. 1.11 : Invalid heaps

1.20 HEAP OPERATIONS

Three Basic Heap Operations are:

1. **Insert:** To insert an element into the heap.
2. **Delete:** To delete max (or min) element from heap.
3. **Build Heap:** To create a heap from scratch. This can be done by successive inserts.

Heap is generally not traversed, searched or printed. To implement the insert and delete operations, we need two other operations: ReheapUp and ReheapDown.

More advanced operations include Merge( ), which merges two heaps.

1.20.1 ReheapUp

If we have a nearly complete binary tree with N elements whose first N–1 elements satisfy the order property of heaps, but the last element does not. That is, the structure would be a heap if the last element were not there. The **ReheapUp** operations repairs the structure, so that it is a heap by lifting the last element up the tree until that element reaches at proper position in the tree. This restructuring can be graphically viewed in Fig. 1.12.

We can note in Fig. 1.12, that the last node in the heap was out of order. After the reheap, it is in its correct location and the heap has been extended one node.

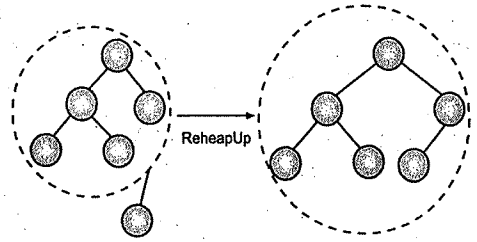


Fig. 1.12 : ReheapUp Operation

As heap is a complete or nearly complete tree, the node must be placed in the last leaf level at the first leftmost empty position as in Fig. 1.13. If the new node's key is larger than that of its parent, it is lifted up the tree by exchanging the child and parent keys and data. The data eventually move to the correct position in the heap by repeatedly exchanging child-parent keys and data. In brief, the ReheapUp repairs a broken heap by lifting the last element up the tree until it reaches the correct location in the heap.

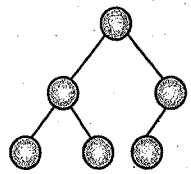


Fig. 1.13 : A heap

Let us consider the following example.

Fig. 1.14 shows a tree which is not a heap.

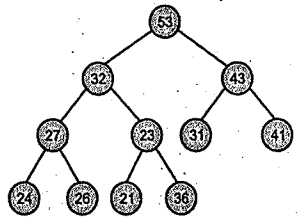


Fig. 1.14 : A tree, not a heap

Here, 36 is greater than its parent, 23; hence is an invalid heap. We therefore exchange 36 and 23 and call ReheapUp to test its current position in the heap. We get the tree shown in Fig. 1.15.

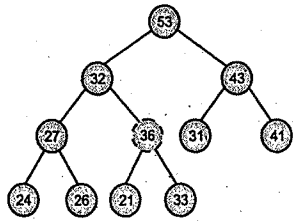


Fig. 1.15 : 36 Moved up

Once again, 36 is greater than its parent, 32. Therefore, we again exchange the data, and find that when ReheapUp is called, the node is placed at correct position and hence, the operation stops. We yield heap as shown in Fig. 1.16.

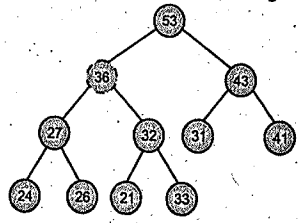


Fig. 1.16 : A heap

1.20.2 ReheapDown

When we have nearly a complete binary tree that satisfies the heap order property except in the root position, we need operation **ReheapDown**. Such situation occurs when the root is deleted from the tree, leaving two disjointed heaps. To correct such situation we move data in the last tree node to the root. Obviously such action disturbs the tree's heap properties. To restore the heap, we need an operation that will sink the root down until the heap ordering property is satisfied.

We call this operation as **ReheapDown.** Fig. 1.17 shows pictorially a ReheapDown operation.

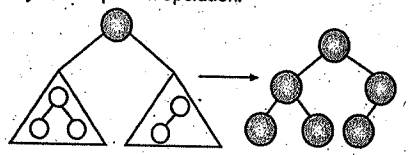


Fig. 1.17 : ReheapDown

Let us consider a broken heap in Fig. 1.18.

Here, when we start, the root 21 is smaller than its subtrees. We examine them and select the larger of the two to exchange with the root, here the 43.

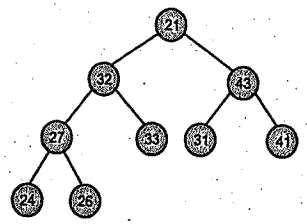


Fig. 1.18 : Original tree, not a heap

Having made the exchange in Fig. 1.19, we see that if we are finished and see that if 21 is smaller than their keys. Once again we exchange 21 with the larger subtrees, 41.

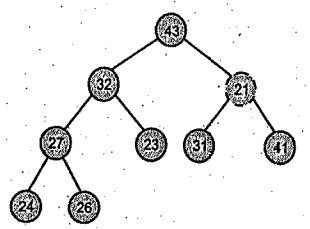


Fig. 1.19 : Root moved down (right)

After moving down 21, we again move it further down a yield tree as in Fig. 1.20.

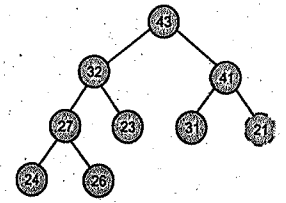


Fig. 1.20 : Moved down again yielding a heap

From Fig. 1.20 we see that we have reached a leaf and should stop.

1.20.3 Data Structure for Heap: An Implementation of Heap using Array

By the property of being complete, a heap can be viewed continuous array, instead of a dynamic tree structure. As t heap tree is complete or nearly complete, the relationsh between a node and its children is fixed and can be calculat as:

(1) For a node at index i, its children are at followir locations:
   - Left child at 2i + 1 and
   - Right child at 2i + 2 location.

(2) The parent of a node of index i is at location $\lfloor (i-1)/2 \rfloor$

(3) Given the index for a left child, j, its right sibling, if any, found at j + 1. Conversely, for a right child at j, its le sibling is at index j – 1.

(4) Given the value, size of complete heap, n, the location first leaf is at $\lfloor n/2 \rfloor$. Given the location of the first le element, the location of the last non-leaf element is 1 les

Let us consider a heap tree at Fig. 1.21.

Heap tree in Fig. 1.21 is shown in its logical form. Physic representation of heap tree of Fig. 1.21 is shown in Fig. 1.22.

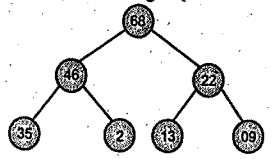


Fig. 1.21 : A heap tree

Using the rules stated, we represent the tree in array as in Fig. 1.22.

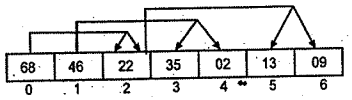


Fig. 1.22 : A heap in an array

1.20.4 BuildHeap

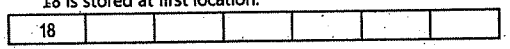
- There are two ways to build a heap. We can start with an empty array and insert elements into the array one at a time, or given an array of data that are not a heap, we can rearrange the elements in the array to form a heap.
- Given a filled array of elements in random order, to build the heap we need to rearrange the data, so that each node in the heap is greater than its children. We begin by dividing the array into two parts, the left being a heap and the right being data to be inserted into the heap.
- Initially, the first node, the root is the only node in the heap and rest of the array are data to be inserted. To insert a node into the heap, we follow the parent path up the heap, swapping nodes that are out of order.
- If the nodes are in proper order, then the process of insertion stops and the next node is selected and inserted and so on till last element. This process is called as **Heapify.**

Let us consider following data.

18, 29, 33, 42, 55, 66, 88.

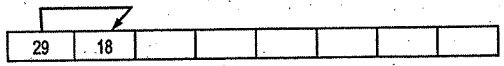
Following Steps Illustrate Heap Building Operation:

(i) Consider 18, to be inserted initially into an empty array. 18 is stored at first location.

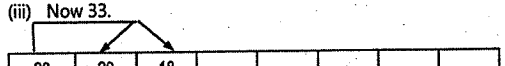


(ii) Now insert 29.

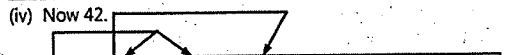
Here, 29 is greater than 18.



(iii) Now 33.

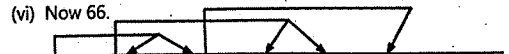


(iv) Now 42.

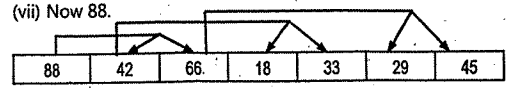


(v) Now 55.

(vi) Now 66.



(vii) Now 88.



Each iteration of the insertion algorithm uses reheap up to insert the next element into the heap and keeps growing on right side of array by one position. If all the elements are already stored randomly in an array then it works as in the following Fig. 1.23.

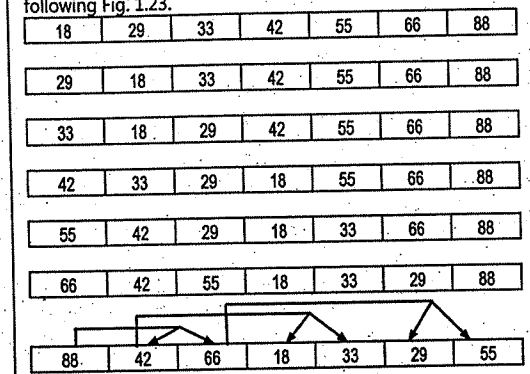


Fig. 1.23 : Building a heap

1.20.5 Insert

A node can be inserted in a heap which have been already built; if there is an empty location in the array. To insert a node we need to search the first empty leaf in the array. We find it immediately after the last node in the tree. To insert a node, we move the new data to the first empty leaf and ReheapUp.

Let us consider the heap already built as in Fig. 1.24.

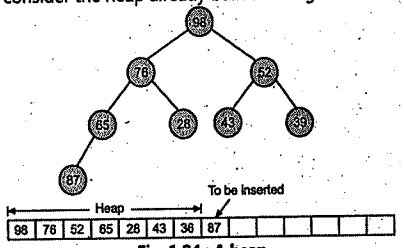


Fig. 1.24 : A heap

A heap in Fig. 1.24 has 7 elements present in it. Let us consider that 87 is to be inserted. Initially 87 is stored at last empty location as the first empty leaf of heap array. Then we heapify to store it in proper position. Resultant heap is shown in Fig. 1.25.

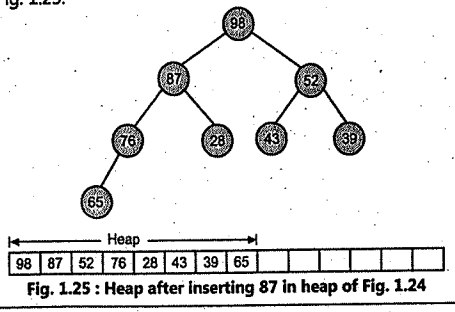


Fig. 1.25 : Heap after inserting 87 in heap of Fig. 1.24

1.20.6 Delete

- While deleting a node from a heap, the most common and meaningful logic is to delete the root. The heap is thus left without a root. To reconstruct the heap, we move the data in the last heap node to the root and ReheapDown.
- Let us consider the heap tree shown in Fig. 1.26. The data at the top of the heap is returned by Delete Operation.

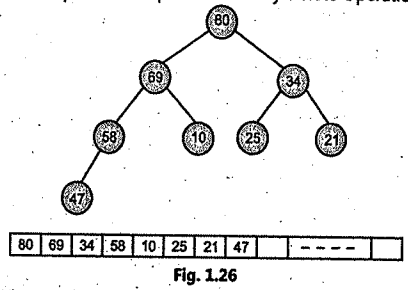


Fig. 1.26

- When Delete operation is performed for heap tree in Fig. 1.26, it returns the element at root, 80. Now the tree is to be ReheapDown again to reconstruct a heap. The reconstructed heap is shown in Fig. 1.27.

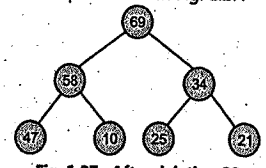


Fig. 1.27 : After deleting 80

1.21 PRIORITY QUEUE

- We have studied the data structure, queue, which works on principle first in first out. Queue is implemented using linear lists. In many applications, we want to prioritize one element over the others. The heap is an excellent structure to use for a **Priority Queue.**
- An event enters the queue, it is assigned a priority number that decides its position in a queue. Element is assigned a priority number even though the new event can enter the heap in only one place at any given time, the first empty leaf. Once the element enters the queue, it is quickly taken to its proper position.
- The highest priority element is taken to the top of the heap and becomes the next element to be processed. If element is of low priority, it remains relatively at lower position in the heap, waiting for its turn.
- **A Heap can be Used as a Priority Queue** the highest queue item is at the root and is trivially extracted. But, if the root is deleted, we are left with two sub-trees and we must efficiently re-create a single tree with the heap property. The value of the heap structure is that we can both extract the highest priority item and insert a new one in **O(log n)** time.

1.21.1 Heap as a Priority Queue

Let's start with the heap given in Fig. 1.28.

A deletion will remove the T at the root.

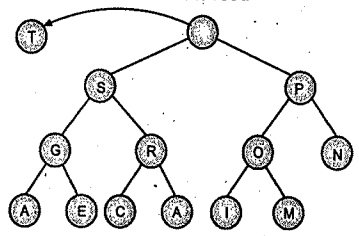


Fig. 1.28

To work out how we are going to maintain the heap proper use the fact that a complete tree is filled from the left. So th the position which must become empty is the one occupi by the M.

Put it in the vacant root position.

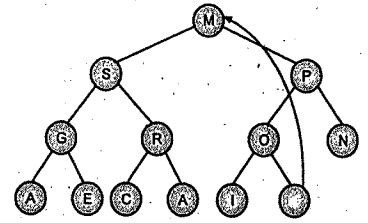


Fig. 1.29

This has violated the condition that the root must be great than each of its children.

So interchange the M with the larger of its children.

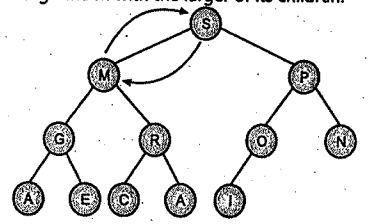


Fig. 1.30

The left structure has now lost the heap property.

So again interchange the M with the larger of its children a shown in Fig. 1.31.

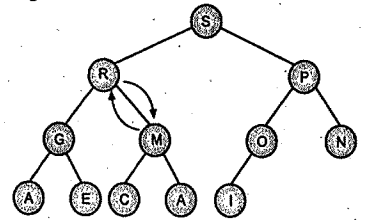


Fig. 1.31

This tree is now a heap again, so we are finished.

We need to make at most *h* interchanges of a root of a subtree with one of its children to fully restore the heap property. Thus, deletion from a heap requires **O(n)** or **O(log n)**.

Addition to Heap:

To add an item to heap, we follow the reverse procedure.

Place it in the next leaf position and move it **up**.

X is added as a leaf as shown in Fig. 1.32.

Again, we require **O(n)** or **O(log n)** exchanges.

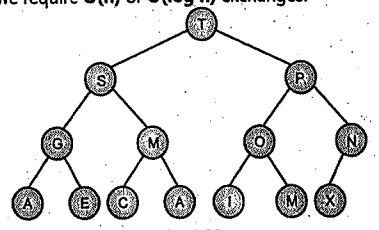


Fig. 1.32

1.21.2 Storage of Complete Trees

The properties of a complete tree lead to a very efficient storage mechanism using n sequential locations in an array.

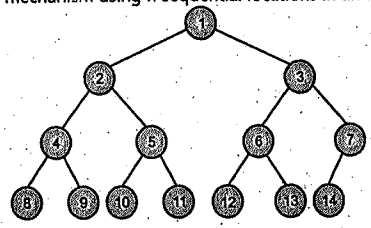


Fig. 1.33

If we number the nodes from 1 at the root and place:

- The left child of node k at position 2k.
- The right child of node k at position 2k + 1.

Then the 'fill from the left' nature of the complete tree ensures that the heap can be stored in consecutive locations in an array.

Viewed as an array, we can see that the n^{th} node is always in index position n.

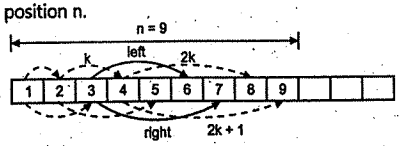


Fig. 1.34

The code for extracting the highest priority item from a heap is naturally, recursive. Once we've extracted the root (highest priority) item and swapped the last item into its place, we simply call MoveDown recursively until we get to the bottom of the tree.

1.21.3 Applications of Priority Queue

There are several applications of priority queue. Some of them are given below:

- **Operating Systems:**
  - CPU Scheduling
  - I/O Scheduling
  - Process Scheduling
- **Event Simulation:**

  Customers arriving to a bank, waiting in a queue for railway reservation etc.
- **Selection Problem:**
  - This problem can be solved in O(N log N) time using priority queue.
  - Consider the problem of determining the k^{th} element in an unsorted list. There are two solutions to the problem. We could first sort the list and select the element at location k, or we could create a heap and delete k – 1 elements from it, leaving the desired element at the top. Let us go for the second solution, using a heap.
  - After building the heap, we can simply select the top element and discard it. But a better solution would be to place the deleted element at the end of the heap and reduce the heap size by one. After processing the k^{th} element, the temporarily removed elements can be reinserted into the heap.
  - Let us consider the heap in Fig. 1.35 (a). Suppose we want to know the fourth largest element in a list.

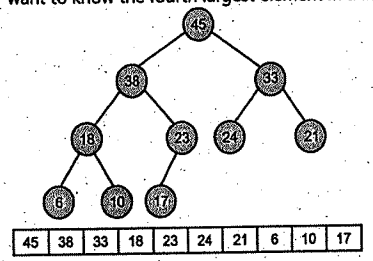


Fig. 1.35 (a)

  - In Fig. 1.35 (a), for the heap of the fourth largest element in a list, after deleting three times, we have the fourth largest element 24 at the tip of the heap. After selecting 24, we reheap to store the heap so that it is complete as shown in Fig. 1.35 (b).

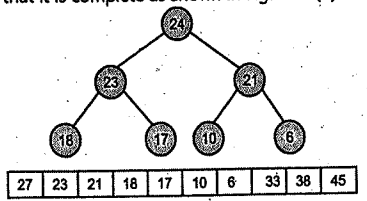


Fig. 1.35 (b)

1.21.4 Heap Sort

In heap sort, data is represented in the form of heap. Let us assume that n data elements are already stored in an array A of size n.

The Basic Idea of Heap Sort is to:

- Organize the entire collection of data elements as a binary tree stored in an array indexed from 1 to N, where for any node at index i, its two children, if exist, will be stored at index of 2*i, and 2*i + 1.
- Divide the binary tree into two parts, the top part in which data elements are in their original order the bottom part in which data elements are heap order, where each node is in higher order than children, if any.
- Start the bottom part with the half of the array, which contains only leaf nodes. Of course, it is in heap order, because leaf nodes have no child.
- Move the last node from the top part to the bottom part, compare its order with its children, and swap its location with its highest order child, if its order is lower than any child. Repeat the comparison and swapping to ensure the bottom part is in heap order again with this new node added.
- Repeat step 4 until the top part is empty. At this time, the bottom part becomes complete heap tree.
- Now divide the array into two sections, the left section which contains a complete heap tree, and the right section which contains sorted data elements.
- Swap the root node with the last node of the heap tree in the left section, and move it to the right section. Since the left section with the new root node may not be a heap tree any more, we need to repeat step 4 and 5 to ensure the left section is in heap order again.
- Repeat step 7 until the left section is empty.

Procedure Heap Sort:

```
HeapSort (Array A, int n)
begin
   Heap (A, n); // construct max heap from existing
elements
   // Now remove root value and put it at the end of the
array
   for p = n to 2 step -1 do
   begin
      swap (a[p], A[1]);
      AlterHeap(A, 1, p-1);
   end
end
```

Procedure Heap which Constructs Max Heap:

```
Heap(Array A, int n)
begin
   for p = (int) (n/2) to 1 step -1 do
   begin
      AlterHeap(A,p,n);
   end
end
```

Procedure AlterHeap:

```
AlterHeap(Array A, int p, int n)
begin
   num = A[p];
   q = 2p; //left child of p
   while (q<=n) do
   begin
/* Compare left and right child elements at positions q
and q + 1, respectively. If right child element is bigger,
then store position of right child in q */
   if num >= A[q] then
   break; // because position for num is found
pos = (int) (q/2)
   A[pos] = a[q];
   q = 2q
   end
   A[pos] = num;
end
```

Here in procedure HeapSort, call to Heap procedure tal O(n) time. And call to AlterHeap procedure is made n tim where each call takes O(log n) time. So the worst-case ti taken is O(n log n). Space requirement is A[1 ... n] and f temporary and index variables.

MULTIPLE CHOICE QUESTIONS (MCQ's)

1. Time complexity is
   (a) Space required by program.
   (b) Amount of machine time necessary for running program.
   (c) Time required for programmer to code.
   (d) all of the above.
2. The worst case complexity is (for instance of size n)
   (a) A function defined by maximum number of ste taken.
   (b) A function defined by average number of ste taken.
   (c) A function defined by a minimum number of ste taken.
   (d) All of the above.
3. The best case complexity (for instance of size n) is
   (a) a function defined by maximum number of ste taken.
   (b) a function defined by average number of ste taken.
   (c) a function defined by a minimum number of ste taken.
   (d) all of the above.

4. What is the time complexity of linear search algorithm over an array of n elements?
   (a) $O(\log_2 n)$ (b) $O(n)$
   (c) $O(n \log_2 n)$ (d) $O(n^2)$
5. What is the time taken by binary search algorithm to search a key in a sorted array of n elements?
   (a) $O(\log_2 n)$ (b) $O(n)$
   (c) $O(n \log_2 n)$ (d) $O(n^2)$
6. What is the time required to search an element in a linked list of length n?
   (a) $O(\log_2 n)$ (b) $O(n)$
   (c) $O(1)$ (d) $O(n^2)$
7. What is the worst case time required to search a given element in a sorted linked list of length n?
   (a) $O(1)$ (b) $O(\log_2 n)$
   (c) $O(n)$ (d) $O(n \log_2 n)$
8. Consider a linked list of n elements which is pointed by an external pointer. What is the time taken to delete the element which is successor of the element pointed to by a given pointer.
   (a) $O(1)$ (b) $O(\log_2 n)$
   (c) $O(n)$ (d) $O(n \log_2 n)$
9. Consider a linked list of n elements. What is the time taken to insert an element after an element pointed by some pointer?
   (a) $O(1)$ (b) $O(\log_2 n)$
   (c) $O(n)$ (d) $O(n \log_2 n)$
10. Consider a linked list implementation of a queue with two pointers: front and rear. What is the time needed to insert element in a queue of length n?
    (a) $O(1)$ (b) $O(\log_2 n)$
    (c) $O(n)$ (d) $O(n \log_2 n)$
11. What is the time required to insert an element in a stack with linked implementation?
    (a) $O(1)$ (b) $O(\log_2 n)$
    (c) $O(n)$ (d) $O(n \log_2 n)$
12. The time required to search on element in a binary search tree having n element is:
    (a) $O(1)$ (b) $O(\log_2 n)$
    (c) $O(n)$ (d) $O(n \log_2 n)$
13. Consider that n elements are to be sorted. What is the worst case time complexity of Bubble sort?
    (a) $O(1)$ (b) $O(\log_2 n)$
    (c) $O(n)$ (d) $O(n^2)$
14. Consider that n elements are to be sorted. What is the worst case time complexity of Shell sort?
    (a) $O(1)$ (b) $O(\log_2 n)$
    (c) $O(n^{1.2})$ (d) $O(n^2)$
15. Example(s) of $O(N^2)$ algorithm is (are)
    (a) Initializing all the elements in a two-dimensional array to zero.
    (b) Printing out all the elements in a two-dimensional array.
    (c) Searching for the smallest element in an unsorted two-dimensional array.
    (d) All of the above.

ANSWERS

1. (b)	2. (a)	3. (c)	4. (b)	5. (a)
6. (b)	7. (c)	8. (a)	9. (a)	10. (a)
11. (a)	12. (b)	13. (d)	14. (c)	15. (d)

EXERCISE

1. What is the formal definition of an algorithm?
2. What is an algorithm? Write essential properties and the performance measures of an algorithm.
3. Find the time complexity of the following programs:

```
for i:=1 to n do
    for j:=i+1 to n do
        for k:=j+1 to n do
    x:=x+1;
```

```
i = 1;
do {
    x++;
    if (i==10)
    break;
    i++;
    }
while (i<=n);
```

4. Write short note on the following:
   (a) Space and time complexity of an algorithm
   (b) Big 'O' notation.
   (c) Characteristics of Algorithm.
5. What do you mean by 'Analysis of Algorithm'?
6. Compare the growth rate of the following functions. Which one is having the highest rate? $n!$, 2^n, n^2, $n \log n$, n^3.
7. Define asymptotic notations. Explain their significance in analyzing algorithms.
8. What is the use of recurrence relations? Give a recurrence relation for sequential search.
9. Solve the recurrence equation: $T(n) = 2T(n/2) + n^2$
10. Explain different recurrence relation solving methods with suitable examples.

✠ ✠ ✠

Unit II

DIVIDE AND CONQUE

2.1 DIVIDE AND CONQUER

- One of the popular algorithm strategies is the divide and conquer. A common approach for solving a problem is to partition the problem into smaller subproblems, find solutions for the subproblems, and then combine the solution of subproblems into a solution for the whole.
- This approach, especially when used recursively, often yields efficient solutions to problems in which the subproblems are smaller versions of the original problem. The subproblems are independently solved. This description raises a question:

By Which Methods are the Subproblems to be Independently Solved?

Often the subproblems are still relatively large and the divide-conquer strategy may possibly be reapplied. Also often the subproblems resulting from a divide-and-conquer design are of the same type as the original problem. For those cases the reapplication of the divide conquer principle is naturally expressed by a recursive procedure.

This again raises a question:

Upto What Limit Should We Reapply Divide-Conquer Process?

The process of splitting the input into distinct subsets continues till these smaller subproblems which are of the same kind are small enough to be solved without splitting further.

2.1.1 Peculiar Characteristics and Use

- Divide and conquer is probably the best known general algorithm design technique. Quite a few very efficient algorithms such as quicksort, fast Fourier transform are specific implementations of this general strategy. Divide and conquer algorithms are designed to be executed on a single processor computer.
- The divide and conquer technique is ideally suited for parallel computations, in which each subproblem can be solved simultaneously by different processors.
- Well suited for problems in which the data set of n-elements can be divided into smaller subsets of data elements and each data set can be independently processed.
- If the subsets are of the same kind as that of original one, recursively divide and conquer is applied till the subset is small enough, so that it can be solved without furth splitting.
- This approach is not suitable for data elements which a not properly subdivided and if the subtasks cannot independently processed.

2.1.2 The General Method

Given a function to complete n inputs, the divide and conqu strategy suggests splitting the inputs into k distinct subse $1 \le k \le n$ yielding k subproblems. These k subproblems are be solved by suitable method. These sub-solutions should combined to yield a solution of the whole. To ea subproblem the divide and conquer is applied till t subproblem is small enough to be solved without furth subdivision.

Control Abstraction for Divide and Conquer :

- Control abstraction is a procedure which mirrors the w an actual problem based on the said strategy w look. It formally means a procedure whose flow of conti is clear, but whose primary operations are specified other procedures whose precise meaning is l undefined.
- Let the n inputs to be processed are stored in an array [1, n] which is global. Let function DandC be a functic which is initially invoked as DandC(1, n). DandC (i, j) solv a problem instance defined by inputs A [i, j].

Algorithm DandC(p, q)

```
Global n, A[1, n]
integer p, q, k
{1 ≤ p ≤ q ≤ n}
begin
if small (p, q)
    then return S (p, q)
else
begin
    Divide problem p into smaller subproblems say p1, p2, ...
pk
    Apply DandC to each of these pi
    return combine (DandC (p1), DandC (p2), ..., DandC
(pk)); /* k ≥ 1 */
    end
end
```

Here small (p, q) is a function which finds whether the input size is small enough that the answer can be computed without further splitting, and then accordingly a Boolean value will be returned by the small (p, q) function. If the input is found to be small enough, then the processing on it will be performed by procedure S. Otherwise the problem p is further divided into subproblems $p_1, p_2, ---, p_k$ and DandC is applied to each $p_1, p_2, ---, p_k$. Combine is a function which collects the results of subproblems $p_1, p_2, ---, p_k$ and combines them to form the result of problem P.

2.1.3 Computing Time of Algorithm DandC

Computing time of DandC is described by the following recurrence relation

$$T(n) = \begin{cases} g(n) & \text{if n is small} \\ T(n1) + T(n2) + \text{-------} + T(nk) + f(n) & \text{otherwise} \end{cases}$$

The complexity of many divide and conquer algorithms is given by recurrence of the form

$$T(n) = \begin{cases} T(1) & \text{if } n = 1 \\ aT(n/b) + f(n) & \text{if } n > 1 \end{cases}$$

where, a and b are known constants which are positive. We assume that T(1) is known and n is a power of b (i.e. $n = b^k$). This recurrence actually describes that the problem of size n is divided into 'a' subproblems, where each subproblem is of size n/b. The cost of dividing the problem and combining the results of all the subproblems is described by the function f(n). One of the methods for solving any such recurrence relation is called the **Substitution Method.** This method repeatedly makes substitution for each occurrence and the function T in the right hand side until all such occurrences disappear.

Let us solve the following recurrence using the substitution method,

$$T(n) = 2\,T(n/2) + cn$$

$$\therefore \quad T(n) = 2\,[2T(n/4) + cn/2] + cn$$

$$= 4T\,(n/4) + 2cn$$

$$= 4[2T(n/8) + cn/4] + 2cn$$

$$= 8T(n/8) + 3cn$$

$$= 2^i T(n/2^i) + icn$$

Putting $i = \log_2 n$, we get

$$T(n) = 2\log_2 n\, T(n/2 \log_2 n) + cn \log_2 n$$

$$= nT(1) + cn \log_2 n$$

If T(1) is assumed to be a constant, then

$$T(n) \in \theta\,(n \log n).$$

2.2 BINARY SEARCH

- Sequential search is not suitable for larger lists. It requires n comparisons in worst case. It is like sequentially searching the friend name in the list and the friend name is **Zahir**. What if the list is name wise sorted in ascending order? Obviously linear search through directory is not an efficient method. Hence we have a better method when the data is sorted.
- Let us consider a typical game played by kids. You are asked to guess the number remembered by your friend in the range of 1 to 100. You have to guess by asking him minimum number of questions. You are of course not allowed to ask a question "which number you have remembered?". The most easiest approach is start asking him: Is it 1? No, then is it 2? And so on till you get the answer "Yes".
- What if the number your friend remembered is 99? Obviously this approach is not the efficient one. The solution to this problem is asking him a question is it greater than 50? If the answer is "Yes", then the range to be searched is 51 to 100 which is half of the previous range. If the answer is "No", still the range is 1 to 49 which is half of the original. You may continue doing so. Ask him "Is it greater than 75?" "Is it greater than 87?" and so on till you guess the number. Surely the second approach reduces the total number of questions asked on an average.
- This method is called **Binary Search,** as we divided the list to be searched every time in two lists and search in only one of the lists. Consider that the list is sorted in ascending order. In binary search algorithm, to search a particular element, it is first compared with the element at middle position if found the search is successful. Else if middle position value is greater than target, the search will continue in the first half of the list else will be searched in the second half of the list. The same process is repeated for one of the half of the list, till list reduces to the list of size one.
- The effectiveness of the binary search algorithm lies in its continual halving of the list to be searched. For an ordered list of 50,000 keys the worst case efficiency is a mere 16 accesses. (In case you do not believe this dramatic increase in efficiency as the list gets larger, try plugging "50,000" into a hand-held calculator and count how many times you must halve the display number to reduce it to 1). The same file that would have necessitated an average wait of 2 minutes using a sequential search will permit a virtually instantaneous response when the binary search strategy is used. In more precise algebraic terms, the halving method yields a worst-case search efficiency of $\log_2 n$.

Implementation of Binary Search:

Here, we have shown only the function Binary_Search.

Iterative Version:

```
int Binary_Search(int list[ ], int n, int x)
{
    int first=0, last = n-1, mid;
    while (first<=last)    /* Iterate while first <= last */
    {
        mid=(first + last)/2;        /* Calculate mid */
        if (list[mid]==x)            /* Found */
            return mid;
        else if (x<list[mid])        /* Not found, look in
                                        the upper half of the list */
            last=mid-1;
        else
            first=mid+1;             /* Look in lower half */
    }
    return -1;                       /* Return "not found" */
}
```

We can also implement binary search in the following recursive way:

Recursive Version:

```
int Binary_Search(int list[ ], int first, int last, int x)
{
    int mid;

    if (first < last)
    {
        mid = (first + last) / 2;
        if (list[mid] == x)
            return mid;
        else if (x < list[mid])
            return Binary_Search(list, first, mid-1, x);
        else
            return Binary_Search(list, mid + 1, last, x);
    }
    return -1;
}
```

- Although this is a more direct implementation of the above description, it uses needless stack space, and is much slower on most systems. Also, this form of recursion is called 'Tail Recursion', which is the most wasteful form of recursion. Recursion is a powerful tool, which mu used with care.
- Binary search requires **O(log(n))** as it halves the list si each step. It is a large improvement over linear searcl a list with 10 million entries, linear search will nee million key comparisons, whereas binary search will just about 24.
- Time complexity of binary search can be written recurrence relation as,

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

- The most popular and easiest way to solve recurr relation is repeatedly make substitutions for occurrence of the function T in right hand side unt such occurrences disappear.

Therefore, $T(n) = T(n/2) + c$

$$= T(n/4) + 2c$$

$$= T(n/8) + 3c$$

$$= T(n/2^k) + kc$$

$$= T(n/n) + kc = T(1) + kc$$

where, $2^k = n$, hence $k = \log_2 n$

$\therefore \quad T(n) = T(1) + c \log_2 n$

$$T(n) = O(\log_2 n)$$

Although binary search is already very good, at times it car slightly improved, using the Fibonacci search.

2.3 MERGE SORT

- We have studied quicksort which is one of the sev classic algorithms for sorting which follows the divide conquer strategy. Let us study one more algorithm tha merge sort.
- Merge sort is the most common algorithm used external sorting. Merging is the process of combining or more sorted files into the third sorted file. We can merging of two sorted lists to understand the techniqu
- A file (or sub-file) is divided into two files, f1 and f2. Th two files are then compared, one pair of records at a tir and merged. This is done by writing them on t separate new files M1 and M2. Elements that do not p off are simply rewritten into the new files.
- The records in M1 and M2 are now blocked with t records in each segment. The two blocks (that is, records), one from M1 and one from M2, are merged a written onto the original files f1 and f2.

- The length of the segments in each of f1 and f2 is now increased to 4, the merge process is applied again, and the new files are written to M1 and M2. The process is continued until one of the two files f1 or f2 is empty.
- Let us implement the merge sort technique for two arrays instead of working on the files. Let us write a routine that accepts two sorted arrays A and B containing elements n1 and n2 respectively and merges them into a third array C containing n3 elements as follows:

Algorithm MergeSort(List L, int n)

```
begin
   if (n=1)then
          return (L);
   else
   begin
          split L into two halves A and B
          return (Merge(MergeSort(A,n/2),
MergeSort(B,n/2)));
   end
end
```

The algorithm to merge two sorted lists A and B into C is given below:

Algorithm Merge(A, B, C, n1, n2, n3)

```
begin
   i=j=k=1;
   while (i<n1 and j<n2)
   begin
          if (A[i]<B[j])
          begin
                C[k]=A[i]
                i=i+1
          end
          else
          begin
                C[k]=B[j]
                j=j+1
          end
          k=k+1
   end
   while (i<=n1)
   begin
          C[k] = A[i]
          i=i+1
          k=k+1
   end
   while (j<=n2)
   begin
       C[k]=B[j]
       k=k+1; j=j+1;
   end
end
```

The algorithm Merge Sort illustrates well all the facets of divide and conquer strategy. When the number of elements to be sorted are greater than one, Merge Sort separates list into two sub-instances, solves each of these recursively, and then combines the two sorted halves to obtain the solution by calling the algorithm Merge.

Let T = {3, 1, 4, 1, 5, 9, 2, 6, 3, 5, 8, 9}

T spited into two halves:

A = {3, 1, 4, 1, 5, 9}, B = {2, 6, 3, 5, 8, 9}

Each A and B are recursively sorted again by calling Merge Sort for each and are sorted:

A = {1, 1, 3, 4, 5, 9}, B = {2, 3, 5, 6, 8, 9}

Now call to merge gives T as:

T = {1, 1, 2, 3, 3, 4, 5, 5, 6, 8, 9, 9}s

Time complexity of merge sort is **O(n log n).**

When merge sort is used for files as described above, each merge operation requires reading and writing of two files, both of which are on the average about n/2 records long. Thus, the total number of blocks read or written in a merge operation is approximately 2n/c, where, c is the number of records in a segment. The number of segments accessed for the whole operation is $O((n(\log_2 n))/c)$, which amounts to $O(\log_2 n)$ passes through the entire original file.

2.3.1 Analysis of Merge Sort

The merge sort algorithm has the nice property that its time complexity is O(n log n) even in the worst case. If the time for the merging operation is proportional to n, then the computing time for merge sort is described by the recurrence relation.

$$T(n) = \begin{cases} a & \text{if } n = 1 \\ 2T(n/2) + cn & \text{if } n > 1 \end{cases}$$

Here, a and c are constants. When n is a power of 2, $n = 2^i$. We can solve this recurrence by the substitution method as shown below:

$$\begin{aligned} T(n) &= 2T(n/2) + cn \\ &= 2[2T(n/4) + cn/2] + cn \\ &= 4T(n/4) + 2cn \\ &= 4[2T(n/8) + cn/4] + 2cn \\ &= 8T(n/8) + 3cn \\ &= 8[2T(n/16) + cn/8] + 3cn \\ &= 16T(n/16) + 4cn \\ &\vdots \\ &= 2^iT(n/2^i) + icn \end{aligned}$$

$$\begin{aligned} &= nT(n/n) + cn \log_2 n \\ &= nT(1) + cn \log_2 n \\ &= an + cn \log_2 n \end{aligned}$$

If $2^i < n \le 2^{i+1}$, then $T(n) \le T(2^{i+1})$.

Therefore, T(n) = O(n log n).

Thus, the time complexity for merge sort is O(n log n) even in the worst case.

2.4 QUICK SORT

- As the name suggests, the quick sort method is the fastest one. The quick sort is an in-place, divide-and-conquer, massively recursive sort. The algorithm is simple in theory, but very difficult to put into code.
- The purpose of the quick sort is to move a data item in the correct direction just enough for it to reach its final place in the array. The method, therefore, reduces unnecessary swaps, and move an item a great distance in one move.
- A pivot item near the middle of the array is chosen, and then items on either side are moved, so that the data items on one side of the pivot are smaller than the pivot, whereas those on the other side are larger. The middle (pivot) item is now in its correct position.
- The procedure is then applied recursively to the two parts of the array, on either side of the pivot, until the whole of numbers are sorted.

The Recursive Algorithm Consists of Four Steps:

1. If there is one element in the array to be sorted, return immediately.
2. Pick an element in the array to serve as a "pivot" point. (Usually the left-most element in the array is used.)
3. Split the array into two parts - one with elements smaller than the pivot and the other with elements larger than the pivot.
4. Recursively repeat the algorithm for both halves of the original array.

- The purpose of the quick sort is to move a data item in the correct direction just enough for it to reach its final place in the array. The method, therefore, reduces unnecessary swaps, and moves an item a great distance in one move.
- In quick sort, the division into two subarrays is made so that the sorted subarrays do not need to be merged later. This is accomplished by rearranging the elements in A [1:n] such that A [i] < A [j] for all i between 1 to m and all j between m + 1 and n for some m, $1 \le m \le n$. Thus, the elements in A [1: m] and A [m + 1: n] can be independently sorted. No merge is needed.
- The rearrangement of elements is accomplished picking some element of array A [ ], say t = A [5] and reordering the other elements, so that all elem appearing before t in A [1: n] are less than or equal and all element appearing after t are greater than or e to t. This rearrangement is called as **Partitioning**.

Let us assume that m represents first position in a partitior

Algorithm Partition(A, m, p)

```
begin
   v = A[m], i = m, j = p
   do
   begin
   do i = i + 1 while (A(i)≤v);     //find first element
                                     greater than pivot
   do j = j - 1 while (A(j)≥v);     // find first element le
                                     than pivot
      if i < j exchange (A(i), A(j))
   end
   while (i≤j);
   A(m) = A(j), A(j) = v            // place pivot at its correc
                                     position
   return(j);
end
/* Quick sort algorithm */
```

Algorithm qsort(p, q)

```
/* p and q are start and end positions of a partition */
begin
   if (p<q) then
   begin
      j = q + 1
      m = partition (A, p, j)       //pivot has taken its
                                     correct position
      qsort (p, m - 1) // sort left partition of pivot
      qsort (m + 1, q) // sort right partition of pivot
   end
end
```

- The partition algorithm take care of partitioning. It ta three arguments. The first argument is an array A wh contains all the elements. The second argument m a the third argument p denote the starting and end positions of a partition to be rearranged respectively. H we are using the first element of a partition A[m] a pivot element v.

- Actually any element can be used as a pivot element. But in practice, the first element is used generally. The algorithm will rearrange the elements A [m], A [m+1], ..., A [p] such that the pivot element will be at position j. All the elements at positions m to j – 1 are smaller than the pivot element, that is, A [u] < A [j] for all m ≤ u < j. All the elements between j + 1 to p are greater than or equal to the pivot element, that is, A [u] ≥ A [j] for all j < u ≤ p.

2.4.1 Analysis of Quicksort

- Now, let us see the efficiency of quick sort. On the first pass, every element in the array is compared to the pivot, so there are n comparisons. The array is then divided into two parts each of size (n/2) approximately. (We assume that the array is divided into approximately one half each time). For each of these **subarrays**, (n/2) comparisons are made and four **subarrays** of size (n/4) are formed. So at each level, the number of **subarrays** doubles. It will take up $\log_2 n$ divisions if we are dividing the array approximately one half each time. Therefore, quick sort is **O(n log n)** on the average.
- If the original array is sorted and array[left] is chosen as a pivot, the quick sort turns out to be $O(n^2)$. Therefore, when we choose array[left] as pivot, quick sort works best for files that are completely unsorted and worst for files which are completely sorted. In the case of nearly sorted arrays choose a random element as a pivot value.

Let us analyze again using another method.

- In analyzing qsort( ), we count only number of element comparisons C(n). We make following assumption:

  n elements to be sorted are distinct and i/p distribution is such that partition element v = A[m] in the call to partition (A, m, p) has an equal probability of being i^{th} smallest element, $1 \le i \le (p - m)$ in A (m, p – 1).

Worst Case:

At level one only one call to partition is made with n elements in call of partition, at level two at most two calls are made with elements (n – 1) etc.

$$C(n) = O(n^2)$$

Average Case C_A (n):

(n + 1) number of element comparisons are required by partition on its first call. The partition element has equal probability of being i^{th} smallest element in array.

$$\therefore \quad C_A(n) = (n+1) + \frac{1}{n} \sum_{1 \le k \le n} C_A(k-1) + C_A(n-k)$$

Multiply by n

$$n\,C_A(n) = n(n+1) + \sum_{1 < k < n} C_A(k-1) + C_A(n-k)$$

$$= n(n+1) + C_A(0) + C_A(1) + \ldots\ldots + C_A(n-1) + C_A(n-1) + C_A(n-2) + \ldots\ldots + C_A(0)$$

$$= n(n+1) + 2\,[C_A(0) + C_A(1) + \ldots\ldots + C_A(n-1)] \quad \ldots (2.1)$$

Replace n by n – 1, we get

$$(n-1)\,C_A(n-1) = (n-1)\,n + 2\,[C_A(0) + C_A(1) + \ldots\ldots + C_A(n-2)] \quad \ldots (2.2)$$

Subtracting (2.2) from (2.1):

$$n\,C_A(n) - (n-1)\,C_A(n-1) = n(n+1) - n(n-1) + 2\,C_A(n-1)$$

$$n\,C_A(n) = n^2 + n - n^2 + n + 2\,C_A(n-1) + (n-1)\,C_A(n-1)$$

$$= 2n + C_A(n-1)\,(2 + n - 1)$$

$$n\,C_A(n) = 2n + (n+1)\,C_A(n-1)$$

.... divide by n(n+1)

$$\frac{C_A(n)}{n+1} = \frac{2}{(n+1)} + \frac{1}{n}\,C_A(n-1) \quad \ldots (2.3)$$

$$\frac{C_A(n)}{n+1} = \frac{2n + (n+1)\,C_A(n-1)}{(n+1)n}$$

Use equation (2.3) for $C_A(n-1)$, $C_A(n-2)$, ...

$$\frac{C_A(n)}{n+1} = \left[\frac{C_A(n-1)}{n}\right] + \frac{2}{n+1}$$

$$= \left[\frac{nC_A(n-2) + 2n - 2}{n(n-1)}\right] + \frac{2}{n+1}$$

$$= \frac{C_A(n-2)}{n-1} + \frac{2}{n} + \frac{2}{n+1}$$

$$= \frac{C_A(n-3)}{n-2} + \frac{2}{n-1} + \frac{2}{n} + \frac{2}{n+1} \quad \ldots (2.4)$$

$$= \frac{C_A(1)}{2} + 2 \sum_{3 \le k \le n+1} \frac{1}{k}$$

But $C_A(0) = C_A(1) = 0$.

$$= 2 \sum_{3 \le k \le n+1} \frac{1}{k}$$

$$= 2 \sum_{3 \le k \le n+1} \frac{1}{k} \int_{2}^{n+1} \frac{1}{x}\,\partial x$$

$$= 2\,(\log_e(n+1) - \log_e 2)$$

$$\therefore \quad \frac{C_A(n)}{n+1} = 2\,(\log_e(n+1) - \log_e 2)$$

$$\therefore \quad C_A(n) = 2(n+1) * [\log(n+1) - \log_e 2]$$

$$= (2n+2)\log(n+1) - 2n\log_e 2 - 2\log_e 2$$

$$= 2n\log(n+1) + 2\log(n+1) - 2n\log_e 2 - 2\log_e 2$$

$$= O(n \log n)$$

2.4.2 Quick Sort Program Implementation

```
void q_sort(int array[], int left, int right)
  {
int pivot, l_hold, r_hold;

  l_hold = left;
  r_hold = right;
  pivot = array[left];
  while (left<right)
    {
      while ((array[right]>=pivot)&&(left<right))
        right--;
      if (left!=right)
        {
          array[left] = array[right];
          left++;
        }
      while ((array[left]<=pivot)&&(left<right))
          left++;
      if (left!=right)
        {
          array[right] = array[left];
          right--;
        }
    }
  array[left] = pivot;
  pivot = left;
  left = l_hold;
  right = r_hold;
  if (left<pivot)
    q_sort(array, left, pivot-1);
  if (right>pivot)
    q_sort(array, pivot+1, right);
}

void QuickSort(int array[ ], int array_size)
  {
  q_sort(array, 0, array_size - 1);
}
```

2.5 STRASSEN'S MATRIX MULTIPLICATION

Let A, B, C are three matrices of size n × n each. M multiplication is performed as ,

$$C_{ij} = \sum_{k=1}^{n} A_{ik}\,B_{kj} \text{ for } 1 \le i \le n.\ i \le j \le n$$

Thus time taken to compute C_{ij} is θ(n). But matrix C cont n^2 elements. Hence, total time required to multiply matrices is $\theta(n^3)$.

Strassen has discovered the following algorithm for m multiplication.

Let,

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$$

$$B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

Consider the following operations :

$$m_1 = (A_{11} + A_{22})\,(B_{11} + B_{22})$$
$$m_2 = (A_{21} + A_{22})\,B_{11}$$
$$m_3 = A_{11}\,(B_{12} - B_{22})$$
$$m_4 = A_{22}\,(B_{21} - B_{11})$$
$$m_5 = (A_{11} + A_{12})\,B_{22}$$
$$m_6 = (A_{12} - A_{11})\,(B_{11} + B_{12})$$
$$m_7 = (A_{12} - A_{22})\,(B_{21} + B_{22})$$

All the above operations involve only one multiplication ea

If C = AB, then elements of matrix C can be computed as:

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

This computation of 2 × 2 matrix involves 7 multiplicati and 10 additions/subtractions.

If each element of A and B is replaced by n × n matrix, tl the algorithm computes multiplication of two 2n × matrices using 7 matrix multiplications and 10 ma additions/subtractions.

Assume $n = 2^k$. If $n \ne 2^k$, then extra rows and columns of ze can be added to a matrix to make $n = 2^k$. As Strassen's ma multiplication requires 7 matrix multiplications and also ma additions/subtractions require $\theta(n^2)$, the time taken Strassen's matrix multiplication is given by

$$T(n) = \begin{cases} a & , \ n \le 2 \\ 7T(n/2) + bn^2 & , \ n > 2 \end{cases} \quad \ldots (2$$

where a, b are constants. Solving equation (2.5), we get

$$T(n) = O(n^{\log_2 7}) \approx O(n^{2.81})$$

Thus it takes $O(n^{2.81})$ time to multiply two n × n matrices usi Stassen's algorithm.

MULTIPLE CHOICE QUESTIONS (MCQ's)

1. Which of the following algorithm design technique is used in quick sort algorithm?
   (a) Dynamic programming
   (b) Back tracking
   (c) Divide and conquer
   (d) Greedy
2. Algorithms like merge sort, quick sort and binary search are based on
   (a) Greedy algorithm
   (b) Divide and Conquer algorithm
   (c) Hash table
   (d) Parsing
3. Step(s) in Divide and conquer process that takes a recursive approach is said to be
   (a) Conquer/Solve (b) Merge/Combine
   (c) Divide/Break (d) Both B and C
4. In Divide and Conquer process, breaking problem into smaller sub-problems is responsibility of
   (a) Divide/Break (b) Sorting/Divide
   (c) Conquer/Solve (d) Merge/Combine
5. Quick Sort can be categorized into which of the following?
   (a) Brute force technique
   (b) Divide and conquer
   (c) Greedy algorithm
   (d) Dynamic programming
6. What is the best case complexity of Quick Sort?
   (a) O(nlogn) (b) O(logn)
   (c) O(n) (d) $O(n^2)$
7. The given array is arr = {2,3,4,1,6}. What are the pivots that are returned as a result of subsequent partitioning?
   (a) 1 and 3 (b) 3 and 1
   (c) 2 and 6 (d) 6 and 2

ANSWERS

1. (c)	2. (b)	3. (c)	4. (a)	5. (b)
6. (a)	7. (a)			

EXERCISE

1. Explain divide and conquer strategy.
2. Compare quick sort and merge sort methods.
3. Give typical applications in which the divide-conquer is the best suitable algorithmic strategy.
4. Write the control abstraction of divide and conquer strategy. Prove that divide and conquer strategy algorithms are inherently recursive. Describe in brief: Time complexity of quick sort algorithm.
5. Devise a binary search algorithm which splits the set into two sets of sizes: one third and two third. How you will compare this algorithm with binary search?
6. Write the recurrence relation for quick sort.
7. What do you mean by Strassen's Matrix Multiplication?

Unit III

GREEDY ALGORITHM

3.1 GREEDY STRATEGY

- A Greedy Method is any algorithm that follows the problem solving heuristic of making the locally optimal choice at each stage with the hope of finding the optimum solution.
- For example, applying the greedy strategy to the travelling salesman problem yields the following algorithm: "At each stage visit the unvisited city nearest to the current city". In general, greedy algorithms are used for optimization problems.
- Often we are looking at optimization problems whose performance is exponential. A feasible solution for which the optimization function has the best possible value is called an *optimal solution*.
- In the greedy method we attempt to construct an optimal solution in sequence of choice. At each choice we make a decision that appears to be the best at that time.
- A decision made at one choice is not changed in a later choice, so each decision should assure feasibility. A greedy method could be, at each choice increase the total amount of change constructed as much as possible.
- A greedy method is optimal for some change systems. In normal circumstance to find a solution all the combinations are required. If such combinations are more, then the greedy algorithm reduces useless combinations.

The Optimal Solution is as Follows :

- The Greedy algorithms are the simplest one and are usually the most straight forward. As the name suggests, they are shortsighted in their approach, taking decisions on the basis of information immediately at hand without worrying about the effect these decisions may have in the future.
- Thus, they are easy to invent, easy to implement, and they work efficiently. However, since the world is rarely that simple, many problems cannot be solved correctly by such a crude approach.
- The Greedy approach suggests constructing a solution obtained so far, until a complete solution to the problem is reached. This is the central point of this technique the choice made must be :
  - **Feasible**, i.e. it has to satisfy the problem constraints.
  - **Locally Optimal**, i.e. it has to be the best local choice among all feasible choices available on that step.
  - **Irrevocable**, i.e. once made, it cannot be changed subsequent steps of the algorithm.

3.1.1 The General Method

- The greedy algorithm suggests that one can devise algorithm which works in stages, considering one inpu a time. At each stage, decision is made regarding whet or not a particular input is in optimal solution. Any sut of input which satisfies given constraints is called feasible solution. A feasible solution that maximizes minimizes a given objective is called an optimal solutio
- There is usually an obvious way to determine a feas solution but not necessarily an optimal solution. method works in stages, considering one input at a ti At each stage, a decision is made regarding whethe particular input is an optimal solution.
  - This is done by considering the inputs in an or determined by some selection procedure. If inclusion of the next input into the parti constructed optimal solution will result in infeasible solution, then this input is not added to partial solution. Otherwise, it is added.
  - The selection procedure itself is based on so optimization measure. We need to find a feasi solution that either maximizes or minimizes a gi objective function. The measure may be object function. Following is the control abstraction greedy method.

3.1.2 Control Abstraction

- In the control abstraction given here, the function sele an input from an array a[ ] and removes it. The select input value is assigned to x. Feasible is a Boolean valu function that determines whether x can be included ir the solution vector.
- The function union combines x with the solution a updates the objective function. The function Gree describes the essential way that a greedy algorithm v look, once a particular problem is chosen and t functions select, feasible and union are prope implemented.

Algorithm Greedy(a, n)

```
{ a[1: n] contains n inputs }
begin
    solution = nil
    for i = 1 to n do
    begin
        x = Select (a)
        if Feasible (solution, x) then
            solution = Union (solution, x)
    end
    return solution
end
```

Elements of the Greedy Strategy

To decide whether a problem can be solved using a greedy strategy, the following elements of the greedy strategy should be considered :

- Greedy-choice property
- Optimal substructure

Greedy-Choice Property

- A problem exhibits the greedy-choice property if a globally optimal solution can be arrived at by making a locally optimal greedy choice. That is, we make that choice which seems best at that time, without considering results from subproblems. When the dynamic programming makes choice at each step, it considers solutions to the subproblems.
- So it proceeds from smaller subproblems to larger subproblems in a bottom-up fashion. But when the greedy algorithm makes choice at each step, it uses that choice which looks best at that time and then solves the problem. So it never depends on future solutions. Thus it proceeds in a top-down manner and reduces each problem instance to a smaller one.
- Many times it is possible to design an efficient algorithm by making greedy choices quickly. This can be achieved by using appropriate data structure or by preprocessing the input.

Optimal Substructure

- When an optimal solution to the problem contains optimal solutions to subproblems within it, then it is said that a problem exhibits optimal substructure.
- This property tells us that the given problem can be solved using dynamic programming and greedy method.

3.1.3 Peculiar Characteristics and Use

Commonly, greedy algorithms and the problems they can solve are characterized by most or all of the following features.

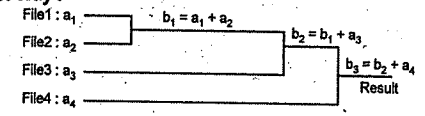
- Greedy algorithms are both intuitively appealing and simple.
- Greedy algorithms are typically used to solve optimization problem.
- We have some problem to solve in an optimal way. To construct the solution of our problem, we have a set (or list) of candidates: the coins that are available, the edges of a graph that may be used to build a path, the set of jobs to be scheduled, or whatever.
- As the algorithm proceeds, we accumulate two other sets. One contains candidates that have already been considered and chosen, while the other contains candidates that have been considered and rejected.
- There is a function that checks whether a particular set of candidates provides a solution to our problem, ignoring questions of optimality for the time being. For instance, do the coins we have chosen add up to the amount to be paid? Do the selected edges provide a path to the node we wish to reach? Have all the jobs been scheduled?
- A second function checks whether a set of candidates is feasible, that is, whether or not it is possible to complete the set by adding further candidates, so as to obtain at least one solution to our problem. Here too, we are not for the time being concerned with optimality. We usually expect the problem to have at least one solution that can be obtained using candidates from the set initially available.
- Yet another function, the selection function, indicates at any time which of the remaining candidates, that have neither been chosen nor rejected, is the most promising.
- Finally an objective function gives the value of a solution we have found : the number of coins we used to make change, the length of the path we constructed, the time needed to process all the jobs in the schedule, or whatever other value we are trying to optimize. Unlike the three functions mentioned previously, the objective function does not appear explicitly in the greedy algorithm.

3.2 OPTIMAL MERGE PATTERNS

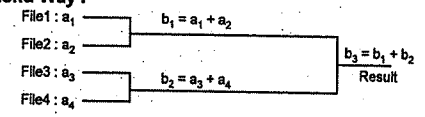
- When we want to merge many sorted files, then there are many ways. If the files have x and y records respectively, then the merged file can be obtained by x + y movements of records, that is, the time required is O(x + y) and the merged file contains (x + y) records.

- Suppose we want to merge four sorted files having a_1, a_2, a_3, a_4 records respectively. Then to obtain merged file, pairing can be done in different ways.

First Way :



Second Way :



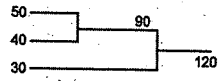
Like this there are many ways to merge n sorted files. Let us see an example.

SOLVED EXAMPLES

***Example 3.1** : Merge three sorted files having 50, 40, 30 records using different ways :*

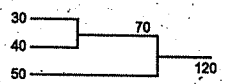
Solution :

First Way :



It means that after merging 50 and 40 records, we get a file with 90 records i.e. 90 records are moved to the merged file. Then this file with 90 records is merged with a file having 30 records, to get result having 120 records obtained after 120 record movements. So total record movements required are 90 + 120 = 210.

Second Way :



Here after merging 30 and 40 records, 70 records are obtained which require 70 record movements. Then the final file is obtained with 70 + 50 = 120 records which involved 120 record movements. So total record movements are 70 + 120 = 190.

So second way of merging requires less record movements, and therefore it is faster than the first way. Now we can determine the greedy method to obtain an optimal merge pattern. To merge n sorted files, the selection criteria is that at each step the two smallest size files should be merged together.

***Example 3.2** : Consider the five sorted files of sizes (28, 15, 20, 9, 3). By applying the greedy rule, the following merge pattern will be generated.*

Solution :

- merge 3 and 9 records to get a_1 = 12 records
- merge a_1 and 15 records to get a_2 = 27 records
- merge 20 records and a_2 to get a_3 = 47 records
- merge 28 records with a_3 to get a_4 = 75 records

3.2.1 Two-Way Merge Pattern and Binary Merge Tre

- **Two-Way Merge Pattern :**

  The merge pattern obtained using the above greedy is referred to as a two-way merge pattern because at e merge step two files are merged.

- **Binary Merge Tree :**

  The two way merge patterns can be represented by bin merge trees. The leaf nodes represent the initial f before merging. Each internal node represents a obtained by merging two files of its children. At e node position, the number of records in the file is sho Binary merge tree for the above example is as sho below (Fig. 3.1):

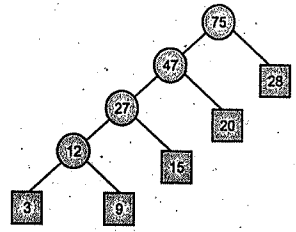


Fig. 3.1 : Binary merge tree

This binary merge tree represents an optimal mer pattern for the given files. Note few more points abo this tree:

- A node at level i is at a distance of i – 1 from the ro
- The weighted external path length of a binary mer tree is given by the total number of record moves that tree. It is given by

$$\sum_{i=1}^{n} d_i l_i$$

where, d_i is a distance of an external node for some file and l_i is the number of records in file x_i.

3.2.2 Optimal Two-Way Merge Pattern

- It corresponds to a binary merge tree with minimu weighted external path length. Let us see an algorith Merge Tree which generates an Optimal two way mer pattern. Algorithm uses the following data structure.

```
struct treenode
{
    int weight;
    struct treenode *lchild,*rchild;
} list[];
```

- Each node in the tree has three fields: weight (number records in the file), pointers to left and right childr respectively.

Algorithm Merge Tree(n)

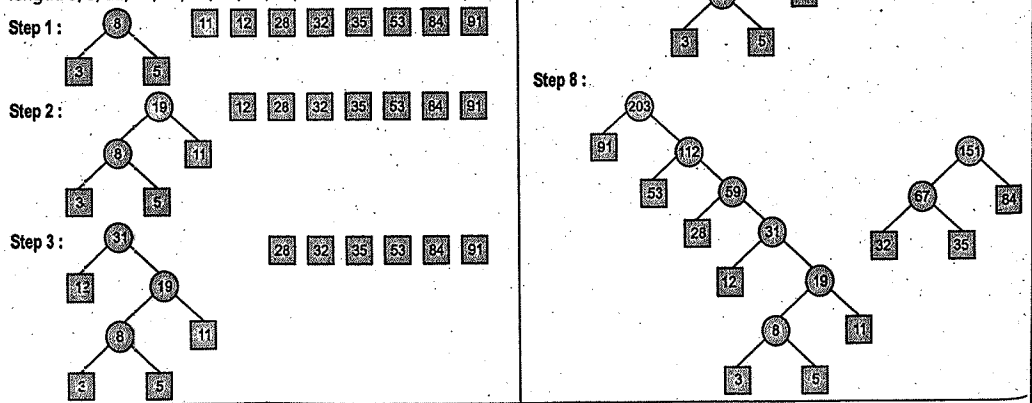
```
begin
    for k = 1 to n – 1
    begin
        m = new treenode;
        (m → lchild) = Smallest (list);
        (m → rchild) = Smallest (list);
        (m → weight) = ((m → lchild) → weight) +
        ((m → rchild) → weight)
        InsertNode (list, m);
    end
    return Smallest(list);
end
```

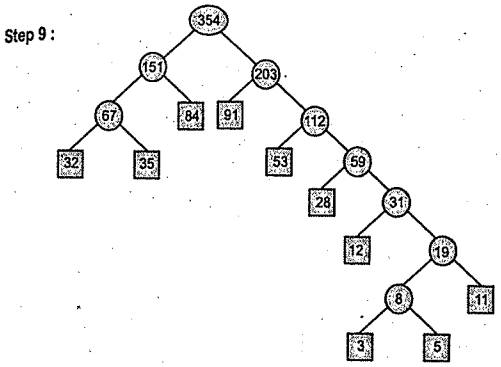
The algorithm uses the greedy rule. It requires n – 1 iterations to merge n files. Input to algorithm is a list in which nodes for n files are stored. These are external nodes of a merge tree. The function Smallest(list) will find and return a tree whose root has smallest weight in the list and removes it from the list.

In each iteration, the two trees having smallest weights are removed from a list, a new tree is created with these two trees as children, their weights are added to get weight of a new root, and this new root is inserted in a list. The function InsertNode(list, m) inserts tree with root m in a list. All the nodes created by the algorithm represent internal nodes in a merge tree.

Example 3.3 : *Find an optimal binary merge pattern for ten files whose lengths are 28, 32, 12, 5, 84, 53, 91, 35, 3, 11.*

Solution : Let us construct an optimal binary merge pattern step by step. First arrange the files in the increasing order of length. 3, 5, 11, 12, 28, 32, 35, 53, 84, 91.





3.2.3 Analysis of Merge Tree Algorithm

The algorithm consists of a for loop which is executed n – 1 times. Now depending on the way the list is maintained, the time complexity of the algorithm differs.

Case 1 : If the list of trees is maintained in the increasing order of weights, then the function Smallest( ) requires O(1) time, but the function InsertNode( ) requires O(n) time. Hence the total time required by the algorithm is $O(n^2)$.

Case 2 : If the list is maintained as a minheap, then the function Smallest( ) and InsertNode( ) require O(log n) time. Hence total computing time is O(n log n).

We have to Show that the Algorithm MergeTree Generates an Optimal Two-Way Merge Tree for n Files (n ≥ 1).

Proof : We can prove the theorem by mathematical induction. For n = 1, the statement is true, because the solution has only one file which is optimal. The induction hypothesis is assume that the statement is true for k, $1 \le k < n$. Now we have to show that the statement is true for n. We can assume that $q_1 \le q_2 \le \ldots \le q_n$. In an iteration, suppose q_1 and q_2 are returned by the smallest( ) which give a subtree T, as shown below in Fig. 3.2 :

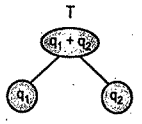


Fig. 3.2

Let T' be an optimal two way merge tree for $(q_1, q_2, \ldots q_n)$. If the children of p are not q_1 and q_2, then we can exchange the present children of p with q_1 and q_2 without increasing the weighted external path length of T'. Hence, T is also a subtree in an optimal merge tree. If T in T' is replaced by an external node with weight $q_1 + q_2$, then the resulting tree T'' is an optimal merge tree for $(q_1 + q_2, q_3, \ldots, q_n)$. From the induction hypothesis, we know that T is replaced by an external no with value $q_1 + q_2$, the algorithm MergeTree tries to find optimal merge tree for $(q_1 + q_2, q_3, \ldots, q_n)$. Hence the algorit MergeTree generates an optimal merge tree for n fi $((q_1, q_2, q_3, \ldots, q_n)$.

3.3 HUFFMAN CODES : AN APPLICATION OF BINARY TREES WITH MINIMAL WEIGHTED EXTERNAL PATH LENGTH

- Huffman code is an optimal set of codes for messag $M_1, M_2, \ldots, M_n$. Each code represents a binary string whi is used as a message for transmission. The receiving e decodes this message using a decode tree. The deco tree generally interprets zero and one as left and rig branch respectively. The leaf nodes of the decode tr represent messages. For example, consider the followi tree shown in Fig. 3.3.

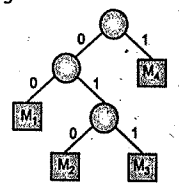


Fig. 3.3

- This decode tree corresponds to codes 00, 010, 011, 1 f messages M_1, M_2, M_3, M_4 respectively. These codes a called as Huffman codes. The number of bits in the co depends on the distance of leaf node for that messag More number of bits causes more time consumption f decoding the message and the decode time increases.
- In general, for message M_i if q_i is a relative frequency f transmission and d_i is a distance of its leaf node from tl root, then expected decode time is $\sum_{i=1}^{n+1} q_i d_i$.
- Hence, if decode tree has minimal weighted external pa length, then expected decode time is minimal, which turn minimizes message length expected.

Algorithm to Compute Huffman Codes :

Let $w_1, w_2, \ldots, w_t$ be the weights of the leaves and it is require to construct an optimal binary tree. The following algorith gives the required optimal binary tree.

Step 1 : Arrange the weights in increasing order.

Step 2 : Consider two leaves with the minimum weights and w_2. Replace these two leaves and their parent node whos weight is $w_1 + w_2$.

Step 3 : Repeat the step 2 for the weights $(w_1 + w_2), w_3, w$ $\ldots, w_t$ until no weight remains. Keep the weights in ascendir order always.

Step 4 : The tree obtained in this way is an optimal tree for given weights.

Step 5 : Assign 0 and 1 to left and right branch respectively.

Step 6 : Sequence of 0's and 1's from a root to a leaf gives Huffman code for that leaf.

3.3.1 Huffman Trees

- If we want to represent some message, then every character in the message can be assigned some sequence of bits called codeword. Obviously every character has unique codeword. In fixed-length encoding, n-bits can be used to encode 2^n characters. For example, 2-bits are sufficient to encode 4 characters. 3-bits are sufficient to encode 8 characters. But even if we want to represent 5, 6 or 7 characters, then also we have to use 3-bits.
- Generally, it is preferred to use less bits for more frequent characters and more bits for less frequent characters. This is referred as variable-length encoding. But its problem is how to know when codeword of one character ends, because length of codeword's of different characters may vary.
- Solution is use of prefix codes. In a prefix code, no codeword is a prefix of another codeword. Algorithm to construct a Huffman tree and to compute Huffman codes.
- Initially every character is represented as a leaf which is also root of its tree. Every node in the tree carries its weight in the tree. The root of a tree gives total weight of that tree. Initial weight of a leaf is the frequency of a character it represents.

Huffman Tree is Constructed using the Following Method.

Let $W_1, W_2, ..., W_n$ be the frequencies of n characters.

Step 1 : Create n trees for n characters and assign weight of n^{th} tree equal to frequency of n^{th} character.

Step 2 : Obtain two trees with the smallest and second smallest weights, say W_1 and W_2. Remove W_1 and W_2 from the list of trees. Create a new node having W_1 as left child; W_2 as right child, and weight $W_1 + W_2$. Add this new tree to the list of trees.

Step 3 : Repeat step 2 for the weights $(W_1 + W_2)$, W_3, W_4 ..., W_n until a single tree is obtained.

The tree obtained is called a Huffman tree. To compute Huffman codes, use the following steps:

Step 4 : Label 0 and 1 to every left and right branch of a Huffman tree respectively.

Step 5 : Travel from the root to a leaf. The sequence of bits obtained represents Huffman code for that leaf.

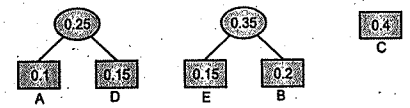
Example 3.4 : *Construct a Huffman code for the following data:*

Character	A	B	C	D	E
Probability	0.1	0.2	0.4	0.15	0.15

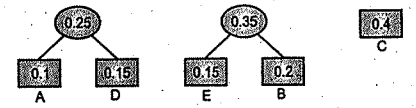
Solution : Consider the weights in the increasing order.

0.1(A), 0.15(D), 0.15(E), 0.2(B), 0.4(C)

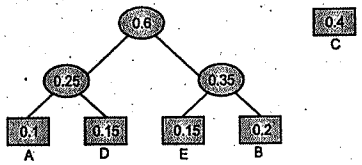
Step 1 :



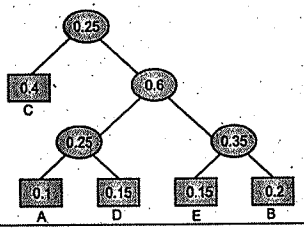
Step 2 :



Step 3 :



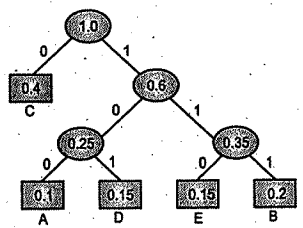
Step 4 :



Example 3.5 : *Obtain Huffman code for the message ABCDE using data of example 3.4.*

Solution :

Step 1 : First label the tree obtained in example 3.4 with 0 and 1 to all the left and right branches respectively. The labelled tree is shown below:



Step 2 : Travel the tree from the root to each leaf and get the following Huffman codes.

Character	Huffman Code
A	1 0 0
B	1 1 1
C	0 0 0
D	1 0 1
E	1 1 0

Huffman code for the message ABCDE is 1001110101110.

Example 3.6 : *Decode the message from bit string 1110100 using data of example 3.4.*

Solution : Travel the Huffman tree from root and follow branches by 111. Leaf B is visited. Then again start from root and follow 0. Leaf C is visited. Again start from root and follow 100. Leaf A is visited. Hence the decoded message is BCA. In the example 2, number of bits in Huffman code for A, B, C, D, E are 3, 3, 1, 3, 3 respectively. Hence the expected number of bits per character is ,

$$3 \times 0.1 + 3 \times 0.2 + 1 \times 0.4 + 3 \times 0.15 + 3 \times 0.15$$

$$= 0.3 + 0.6 + 0.4 + 0.45 + 0.45 = 2.20$$

Hence, to represent 5 characters, 2.20 bits are required per character. But to represent 5 characters using fixed-length encoding, 3 bits are required per character.

Hence, $((3 - 2.20)/3) \times 100 = (0.8/3) \times 100 = 80/3 = 26.67\%$. This shows that Huffman's code saves memory, in this example by 26.67%. Hence it achieves compression.

3.4 KNAPSACK PROBLEM

- We are given n objects and a knapsack or a bag. Each object has a positive weight w_i and a positive profit p_i for $i = 1$ to n.
- The maximum capacity of knapsack is M, that is knapsack can carry a weight not exceeding M. Our aim is to fill up the knapsack such that the profit is maximized while satisfying the constraint that the knapsack will not carry total weight more than M.
- We assume that the objects can be taken into parts, that is, some fraction x_i of total weight w_i. In this case, object i contributes $x_i w_i$ to the total weight and $x_i p_i$ to the profit.
- Hence, our aim is to fill up the knapsack such that

$$\sum_{i=1}^{n} x_i p_i \text{ is maximum subject to the condition}$$

$$\sum_{i=1}^{n} x_i w_i \le M$$

where, $0 \le x_i \le 1$

- We shall use a greedy algorithm to solve the problem terms of control abstraction, a feasible solution is a $(x_1, x_2, ..., x_n)$ that satisfies the above constraints for fil a knapsack.

  In an optimal solution

$$\sum_{i=1}^{n} x_i w_i = M \text{ and } \sum_{i=1}^{n} x_i p_i \text{ is maximum.}$$

- Since, we are working on greedy algorithm, our strat will be to select each object in some suitable order, to as large fraction as possible of the selected objects and stop when the knapsack is full. Here is the algorithm.
- Let w[1, n] and p[1, n] be the arrays representing weig and profits of n objects respectively. M is the maxim capacity of the knapsack.

Algorithm :

```
Knapsack-Greedy (w[ ], p[ ],M)
begin
    for i = 1 to n do x[i] = 0  // initialize
weight = 0, profit = 0
while (weight < M) do
begin
    i = next object with highest profit/weight ratio
    if (weight+w[i] ≤ M) then
    begin
        x[i] = 1
        weight = weight+ w[i]
        profit = profit + p[i]
    end
    else
    begin
    x[i] = (M - weight)/w[i]
    profit = profit + p[i] * x[i]
    weight = M
    end
end
return x[ ]  // solution
end
```

Example 3.7 : *Find feasible solutions for the following knapsac instance :*

Let $n = 5$, $M = 100$, $w = \{10, 20, 30, 40, 50\}$,

$P = \{20, 30, 66, 40, 60\}$

Solution :

Case 1 : Let us choose the objects in the decreasing order of profits. We choose first object 3 having weight 30, then object 5 having weight 50. But now total weight = 30 + 50 = 80. So we have to fill the knapsack with partial weight of object 4, which is equal to

$$\frac{\text{Max. allowed weight} - \text{Current weight}}{\text{Weight of object 4}}$$

$$= \frac{100 - 80}{40} = \frac{20}{40} = \frac{1}{2}$$

So total weight of knapsack = $30 + 50 + \frac{40}{2} = 100$.

Here, we have used whole object 3, whole object 5 and half fraction of object 4. So total profit = profit of object 5 + profit of object 3 + half the profit of object 4 = 66 + 60 + 40/2 = 146. Hence total profit earned is 146 if we select objects profit wise.

Case 2 : Let us choose the objects in the increasing order of weights. So first we choose object 1 having weight 10, then object 2 having weight 20, then object 3 having weight 30 then object 4 having weight 40. So total weight = 10 + 20 + 30 + 40 = 100. So all the objects are used as a whole. So total profit is equal to addition of profits of objects 1, 2, 3, 4 respectively.

$\therefore$ Total profit = 20 + 30 + 66 + 40 = 156.

Hence total profit is 156 if we choose objects weight wise.

Case 3 : Let us choose objects in the order such that the object with maximum profit per unit of weight is used. So profit/weight ratios

$$= \{20/10, 30/20, 66/30, 40/40, 60/50\}$$

$$= \{2, 3/2, 22/10, 1, 6/5\}$$

$$= \{2, 1.5, 2.2, 1, 1.2\}$$

Because object 3 gives maximum profit per unit, it is selected first and its weight is 30. Then object 1 having weight 10 is selected. Then object 2 with weight 20 is selected. Now total weight = 30 + 10 + 20 = 60. So object 5, with weight 50 is selected partially. So fraction of object 5 selected is equal to

$$\frac{\text{Max. weight allowed} - \text{Current weight}}{\text{Weigth of object 5}}$$

$$= \frac{100 - 60}{50} = \frac{40}{50} = \frac{4}{5}$$

So total weight = 30 + 10 + 20 + 50 × (4/5) = 100

Here, objects 3, 1, 2 are used as a whole and 4/5 fraction of object 5 is used. So total profit = 66 + 20 + 30 + 60 × (4/5) = 164

Hence total profit is 164 if we choose objects in the order of profit per unit.

Conclusion : If we observe the profits of three cases, then the case 3 gives the maximum profit. Case 3 actually uses the Knapsack-Greedy algorithm. So the solution obtained is surely optimal.

Example 3.8 : *Find an optimal solution to the knapsack instance with objects n = 7, maximum capacity of knapsack M = 15, profits* $(p_1, p_2, ..., p_7) = (10, 5, 15, 7, 6, 18, 3)$ *and weights* $(w_1, w_2, ..., w_7) = (2, 3, 5, 7, 1, 4, 1)$.

Solution : Using the Knapsack-Greedy algorithm, we can directly select the objects such that object with maximum profit per unit of weight is used. So maximum profit/unit

$$= \left(\frac{10}{2}, \frac{5}{3}, \frac{15}{5}, \frac{7}{7}, \frac{6}{1}, \frac{18}{4}, \frac{3}{1}\right)$$

$$= (5, 1.67, 3, 1, 6, 4.5, 3)$$

So the objects will be selected in the following order: object 5, object 1, object 6, object 7, object 3. Now total weight = 1 + 2 + 4 + 1 + 5 = 13. So next object to be selected is object 2 with weight 3. So fraction of object 2 to be used is equal to

$$\frac{\text{Max. weight allowed} - \text{Current weight}}{\text{Weigth of object 2}}$$

$$= \frac{15 - 13}{3} = \frac{2}{3}$$

$\therefore$ Total weight = 13 + 3 × (2/3) = 15

$\therefore$ Total profit = Addition of whole profits of objects 5, 1, 6, 7, 3 and 2/3 fraction of profit of objects2.

$\therefore$ Total profit = 6 + 10 + 18 + 3 + 15 + 5 × (2/3) = 52 + 10/3 = 55.33.

Hence an optimal solution gives the total profit = 55.33.

Example 3.9 : *Find an optimal solution to the knapsack instance n = 3, m = 20,* $(p_1, p_2, p_3) = (25, 24, 15)$ *and* $(w_1, w_2, w_3) = (18, 15, 10)$.

Solution : Applying the Knapsack-Greedy algorithm, the optimal solution can be obtained if we select objects in the order of profit per unit. For all objects,

$$\text{Profit/unit} = \frac{15}{18}, \frac{24}{15}, \frac{15}{10}$$

$$= (1.39, 1.6, 1.5)$$

So object 2 with weight 15 is selected. Next object 3 with weight 10 is selected partially. So fraction of object 3 to be used is equal to

$$= \frac{\text{Max. weight allowed} - \text{Current weight}}{\text{Weight of object 3}} = \frac{20 - 15}{20} = \frac{5}{10}$$

= 0.5

So 0.5 fraction of object 3 profit will contribute to the optimal solution.

$\therefore$ Total profit = Profit of object 2 + 0.5 fraction profit of object 3 = 24 + 0.5 × 15 = 31.5

Hence the optimal solution gives the total profit = 31.5

3.5 AN ACTIVITY-SELECTION PROBLEM

- Consider an activity of conducting football matches, but there is only one ground. Only when first match ends, next match can start. Consider a multiplex movie theater having multiple screens where one screen is to be shared by multiple movies.
- Only when first movie on a screen finishes, next movie can start on that screen. For example, movie 1 requires 1 pm – 4 pm, movie 2 requires 4 pm – 7 pm, movie 3 requires 7 pm – 10 pm, movie 4 requires 2 pm – 5 pm, movie 5 requires 5 pm – 8 pm time slots.
- So screen 1 can show movie {1, 2, 3} or movies {4, 5} or movies {1, 5}. Various such combinations are possible according to start time and finish time of movies.
- In general, this situation arises when a common resource is to be used exclusively by various activities, then how the activities should be scheduled is an important issue.
- Let a set of n activities $A = \{A_1, A_2, ... A_n\}$ wants to share a common resource exclusively (one at a time). Each activity A_i has start time ST_i and finish time FT_i, where $0 \le ST_i < FT_i < \infty$. Any selected activity A_i is performed in the half-open time interval $[ST_i, FT_i)$.

Compatible Activities: Two activities A_1 and A_2 are said to be compatible if $ST_1 \ge FT_2$ or $ST_2 \ge FT_1$. In other words, the interval $[ST_1, FT_1)$ of activity A_1 and the interval $[ST_2, FT_2)$ of activity A_2 do not overlap.

Activity-Selection Problem is to select a subset of maximum size having mutually compatible activities. In movie-selection problem, movies {1, 2, 3} form largest subset of compatible movies among various possible subsets.

Suppose we have $A = \{A_1, A_2, ... A_n\}$ as a set of activities. Consider a subset of activities in A denoted by A_{ij} where each activity A_k in the subset starts only when A_i finishes and A_k finishes before A_j starts. That is

$$A_{ij} = \{A_k \mid (A_k \in A) \text{ and } (FT_i \le ST_k < FT_k \le ST_j)\}$$

Assumptions:

1. Assume that there are two fictitious activities: A_0 represents first activity with $FT_0 = 0$ and A_{n+1} represents last activity with $ST_{n+1} = \infty$. Thus entire problem can be represented as $A = A_{0,n+1}$ where $0 \le i, j \le (n+1)$.
2. For simplicity, assume that all the activities are arranged according to their finish time in the ascending order. Thus for all the activities in set A, we have

$$FT_0 \le FT_1 \le FT_2 \le ... \le FT_n < FT_{n+1}$$

and

$$A_{ij} = \begin{cases} \phi & \text{, if } i \ge j \\ \{A_k \mid (A_k \in A) \text{ and } (FT_i \le ST_k \le FT_k \le ST_j)\} & \text{, otherwise} \end{cases}$$

i.e. A_{ij} is an empty set if $i \ge j$. To solve the activity selection problem using greedy method, following theorem is useful.

Theorem: Consider any non-empty subproblem A_{ij}. Let a_e an activity in A_{ij} which has earliest finish time

$$FT_e = \min \{FT_k \mid A_k \in A_{ij}\}$$

Then

- Activity a_e is used in some maximum size subset mutually compatible activities of A_{ij}.
- The subproblem $A_{ie} = \phi$ so that selecting a_e leaves subproblem A_{ej} as the only one that may be non-empty

This theorem plays very important role to solve the activ selection problem using greedy approach. It tells how particular activity A_k should be selected among many activit lying between i and j in subset A_{ij}. It tells how to select t activity which has earliest (smallest) finish time. So selection of an activity is done in a greedy manner so t more number of remaining activities can be considered. A after selecting earliest finish time activity A_e, the subprobl A_{ie} is always empty. Only subproblem A_{ej} has to be solved get the solution.

Let us write greedy algorithm for activity selection probl using recursion. It takes as input : array ST[n] having start ti of n activities, array FT[n] having finish time of all the activiti start i and end j of subproblem A_{ij} to be solved. Assume t all the activities are arranged in the ascending order of fini time FT.

```
Algorithm ActivitySelectionR(ST, FT, i, j)
begin
    // Initialization
    e = i + 1
    /* Search earliest finish time activity between i
    and j whose start time
    is greater than or equal to finish time of Ai */
    while (e < j) and (ST [e] < FT[i] )
        e = e + 1
    /* If no such activity Ae found, return empty se
    otherwise add Ae to the
    solution and continue to find next activity */
    if (e < j)
        return {Ae} U ActivitySelectionR(ST, FT, e,
    else
        return NIL
end
```

Initially the algorithm is called as ActivitySelectionR(ST, FT, n+1). Each recursive call searches suitable activity betweer and j using while loop. Hence each recursive call requires θ(time.

Let us consider the iterative version of activity selection problem using greedy method. Again the assumption is that all the activities are stored in the ascending order of finish time FT.

```
Algorithm ActivitySelectionI (ST, FT)
begin
    /* Add first activity with earliest finish time in
    solution */
    solution = {A1}
    i = 1
    /* Search next suitable activity whose start time
    is greater than or equal to finish time of last
    activity Ai */
    for e = 2 to n
        if ( ST[e] > FT[i] )
        begin
            solution = solution U {Ae}
                        //Add Ae to solution
            i = e       // Remember Ae as last activity
        endif
    endfor
    return solution
end
```

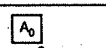
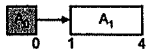
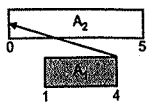
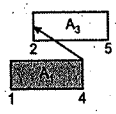
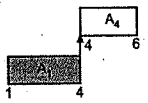
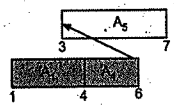
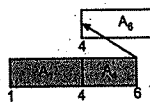
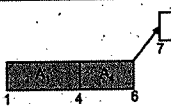
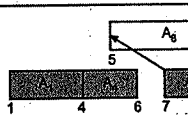
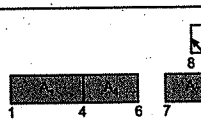
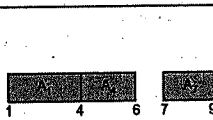
Initially the algorithm will be called as ActivitySelectionI(ST, FT). It will search the solution in the set $A = \{A_1, A_2, ..., A_n\}$. As there are maximum n activities, the algorithm requires $\theta(n)$ time.

Example 3.10 : *Consider the set of activities given in the ascending order of finish time.*

Activity	A_1	A_2	A_3	A_4	A_5	A_6	A_7	A_8	A_9	A_{10}
Start time	1	0	2	4	3	4	7	5	8	10
Finish time	4	5	5	6	7	7	9	10	11	12

Find maximum size subset of mutually compatible activities.

Solution:

Step 1		A_0 is first fictitious activity
Step 2		Select A_1 as activity having earliest FT
Step 3		Reject A_2 as $ST_2 < FT_1$ (arrow points left)
		Reject A_3 as $ST_3 < FT_1$ (arrow points left)
		Select A_4 as $ST_4 \geq FT_1$ (Arrow points up)
Step 4		Reject A_5 as $ST_5 < FT_4$ (arrow points left)
		Reject A_6 as $ST_6 < FT_4$ (arrow points left)
		Select A_7 as $ST_7 \geq FT_4$ (arrow points right)
Step 5		Reject A_8 as $ST_8 < FT_7$ (arrow points left)
		Reject A_9 as $ST_9 < FT_7$ (arrow points left)
		Select A_{10} as $ST_{10} \geq FT_7$ (arrow points right)
Step 6		Solution={A_1, A_4, A_7, A_{10}}

Step 1 : Consider a fictitious activity A_0 with $FT_0 = 0$ at the beginning.

Step 2 : Select activity having earliest FT. As A_1 has earliest FT, it is selected first. $FT_1 = 4$.

Step 3 : Consider A_2 with $ST_2 = 0$.

As $ST_2 < FT_1$, reject A_2.

- Consider A_3 with $ST_3 = 2$.

As $ST_3 < FT_1$, reject A_3.

- Consider A_4 with $ST_4 = 4$.

As $ST_4 \geq FT_1$, select A_4. $FT_4 = 6$.

Step 4 : Consider A_5 with $ST_5 = 3$.

As $ST_5 < FT_4$, reject A_5.

- Consider A_6 with $ST_6 = 6$.

As $ST_6 < FT_4$, reject A_6.

- Consider A_7 with $ST_7 = 7$

As $ST_7 \geq FT_4$, select A_7. $FT_7 = 9$.

Step 5 : Consider A_8 with $ST_8 = 5$.

As $ST_8 < FT_7$, reject it.

- Consider A_9 with $ST_9 = 8$.

As $ST_9 < FT_7$, reject it.

- Consider A_{10} with $ST_{10} = 10$.

As $ST_{10} > FT_7$, select it.

Step 6 : Thus maximum size subset of mutually compatible activities is $\{A_1, A_4, A_7, A_{10}\}$.

3.6 JOB SEQUENCING WITH DEADLINES

Problem Statement : We are given a set of n jobs. Associated with job i is an integer deadline $d_i \geq 0$ and profit $p_i \geq 0$. For any job i, profit p_i is earned if the job is completed by its deadline. In order to complete a job one has to process the job on a machine for one unit of time. Only one machine is available for processing jobs and machine can process only one job at a time. Assume that each job requires one unit time for completion.

Solution : A feasible solution for this problem is a subset J of jobs such that each job in this subset can be completed by its deadline. The value of a feasible solution is the sum of the profits of the jobs in J or $\sum_{i \in J} p_i$. An optimal solution is feasible solution having set of jobs completed within their deadline giving maximum profit.

Example 3.11 : *Let n = 4, profits $(p_1, p_2, p_3, p_4) = (100, 10, 15, 27)$ and deadlines $(d_1, d_2, d_3, d_4) = (2, 1, 2, 1)$*

Solution :

Feasible Solution	Processing Sequence	Profit
(1)	1	100
(2)	2	10
(3)	3	15
(4)	4	27
(1, 2)	2 – 1	110
(1, 3)	1 – 3 or 3 – 1	115
(1, 4)	4 – 1	127
(2, 3)	2 – 3	25
(3, 4)	4 – 3	42

In feasible solution (1, 2), the deadline of job 1 is 2, hen can be scheduled at time t = 1 or t = 2. But job 2 has deac 1. Hence, it should complete at time t = 1. Hence proces sequence is 2 – 1 (job 2 followed by job 1). In feasible solu (1, 3), deadline of jobs 1 and 3 is 2. Hence either job car completed at t = 1 or t = 2. Hence processing sequenc 1 – 3 or 3 – 1.

In feasible solution (1, 4), deadline of job 1 is 2. Hence, it be completed at t = 1 or t = 2. But deadline of job 4 i Hence it should be completed at t = 1. Hence proces sequence is 4 – 1. Similar is the case for (2, 3) and (3, 4). pair (2, 4) is not present in the feasible solution, because t the jobs 2 and 4 have deadline 1 and we cannot comp both jobs at same time t = 1, as each job requires unit time Optimal solution is 4 – 1 because it gives maximum profit 1

The next job to include will be the one that increases $\sum_{i \in J} p_i$ most subject to the constraint that the resulting J is a feas solution. This requires us to consider jobs in decreasing or of Pi's (profits). To determine whether or not a given J i feasible solution, one way would be to try out all poss permutations of jobs in J and check if the jobs in J can processed in any one of these permutations without violat the deadlines.

Actually, the permutation in which jobs are ordered is in increasing order of deadlines.

Example 3.12 : *Obtain optimal solutions for the following jo*

(a) Sorted profits = (p_1, p_4, p_3, p_2) = (100, 27, 15, 10)

Deadlines $d = (d_1, d_4, d_3, d_2)$ = (2, 1, 2, 1)

Solution: n = 4

Job, No.	1	4	3	2
p	100	27	15	10
d	2	1	2	1

(1) Initially

J | 1 | | i = 1

d 2

As deadline [1] = 2, it can be placed at position 1 or 2. position 1 is empty, put J [1] = 1.

(2)

J | 4 | 1 | | i = 2

d 1 2

As deadline [4] = 1, it is placed at position J[1] and job is shifted at next position.

(3) i = 3

Reject job 3, as deadline[3] = 2 and places 1 as well as are already occupied by suitable jobs which cannot shifted (by Rule 1).

(4) i = 4

Reject job 2, as deadline[2] = 1 and place 1 is already occupied by suitable job which cannot be shifted (by Rule 1).

Hence, valid job sequence is (4, 1) with profit 100 + 27 = 127.

(b) n = 7, p = (30, 20, 18, 6, 5, 3, 1)
d = (2, 4, 3, 1, 3, 1, 2)

Solution :

i	1	2	3	4	5	6	7
p	30	20	18	6	5	3	1
d	2	4	3	1	3	1	2

(1) Initially i = 1

J	1			
d	2			

As place 1 is empty and deadline [1] = 2, it can be put at J[1].

(2) i = 2

J	1	2		
d	2	4		

As place 2 is empty and deadline [2] = 4 greater than deadline[1], it can be put at J[2].

(3) i = 3

J	1	3	2	
d	2	3	4	

As deadline of J[2] = 4 is greater than deadline of J[3], hence J[2] is shifted next and J[3] is inserted in between.

(4) i = 4

J	4	1	3	2	
d	1	2	3	4	

As deadline of J[4] = 1 which can be placed only at position 1. Hence all other jobs 1, 2, 3 are shifted without violating their deadlines.

(5) Reject jobs i = 5, 6, 7 as their deadlines are 3, 1, 2 respectively. But their positions are already occupied by suitable jobs which cannot be shifted.

Hence job sequence is (4, 1, 3, 2) with profit = 30 + 20 + 18 + 6 = 74

(c) n = 5, p = (20, 15, 10 ,5, 1)
d = (2, 2, 1, 3, 3)

Solution :

i	1	2	3	4	5
p	20	15	10	5	1
d	2	2	1	3	3

(1) Initially i = 1

J	1			
d	2			

As deadline of J[1] = 2, it can be placed at empty position 1.

(2) i = 2

J	1	2		
d	2	2		

As deadline of J[2] = 2, it is placed at empty position 2. As deadline of J[1] and J[2] is same, their sequence is not changed.

(3) Reject job i = 3, because deadline of J[3] = 1 so it should be placed at position 1. But jobs J[1] and J[2] cannot be shifted now, as they have taken proper positions.

(4) i = 4

J	1	2	4	
d	2	2	3	

As deadline of J[4] = 3 and place 3 is empty. Hence put job 4 at position 3.

(5) Reject job i = 5. As deadline of J[5] = 3 which is already occupied by J[4] whose deadline is also 3. Hence J[4] cannot be shifted.

Hence job sequence is (1, 2, 4) with profit = (20 + 15 + 5) = 40.

Let us see an algorithm to construct an optimal set of jobs which can be completed by their deadlines.

Algorithm Greedy Job(d, J, n)

```
begin
    J = {1}
    for i = 2 to n do
        if (all jobs in J ∪ {i} can be completed by
        their deadlines)
        then J = J ∪ {i}
    end for
    end
```

Job Sequencing Rule :

In the job sequencing with deadlines algorithm, profit p_i is earned iff job i is completed by its deadline. This can be stated as follows :

For a job i, it can be scheduled at time t, where t is the largest integer such that $i \le t \le deadline_i$ and job to be executed at time t is not yet decided. .... Rule (1)

Let us see a greedy algorithm for sequencing unit time jobs with deadlines and profits. Consider jobs in decreasing order of profit.

Algorithm Job Sequence(deadline, Job, n)

```
begin
    deadline[0] = Job [0] = 0
    //start with job1 and add it to optimal solution
array job.
    job[1] = 1
    last = 1 //Initially no. of jobs in optimal solution= 1
    // consider remaining jobs
    for next = 2 to n do
    begin
        current = last
        while (deadline[Job[current]]>deadline[next])
            and deadline[Job[current]] ≠ current))
        begin
            current = current - 1
        end while
    /*  if deadline[Job[current]] = current,
        then we cannot shift current at next position
        (by Rule 1 stated above).
        Hence "next" job is not feasible. So reject it
        and consider next job. */
    /*  If "next" job is feasible, then find position to
        insert it in Job array */
        if ((deadline [Job[current]] ≤ deadline [next])
        and (deadline[next] > current))
        begin
            /* shift all jobs from (current + 1) to last
            by one position */
            for (m=last;m ≥current+1; m --)
            Job [m+1] = Job [m]
            /* Insert "next" job after "current" job in
            optimal solution array
            Job */
            Job[Current + 1] = next
            last = last + 1    // increase no. of jobs in
            optimal solution
        end if
    end for
    return last
end
```

In the algorithm Job sequence, the jobs are considered in the decreasing order of profits. n is the number of jobs. So p[1] ≥ p[2] ≥ ..... ≥ p[n]. Time deadlines of jobs are in the array d[ ] and d[i] ≥ 1 for 1 ≤ i ≤ n. At termination of algorithm, the optimal solution will be in array J[ ] and k is the number of jobs in optimal set. d[J[i]] ≤ d[J [i + 1]] for 1 ≤ i ≤ k, th deadline of job J[i] in the optimal solution is less than or e to the deadline of next job in that solution.

Example 3.13 : *n = 6*

i	1	2	3	4	5	
p[i]	20	15	10	7	5	
d[i]	3	1	1	3	1	

Solution :

1. J | 1 | | | i = 1
   d(i) 3
2. J | 2 | 1 | | i = 2
   d(i) 1 3
   As deadline [1] = 3, it is shifted next. Also dea [2] = 1, it is placed at position 1.
3. Reject i = 3, because deadline [2] = 1, hence we ca shift job 2 (by Rule 1)
4. J | 2 | 1 | 4 | | i = 4
   d(i) 1 3 3
5. Reject 5 as deadline [5] = 1 and place 1 is alr occupied by job 2.
6. Reject 6, as deadline[6] = 3 and place 3 is alr occupied by job 4.

3.6.1 Analysis of Job Sequence Algorithm

The time complexity of this algorithm can be measured u two parameters: total number of jobs n and the numbe jobs k included in the solution J. The total time complexity be given as O(kn). But k ≤ n, so the worst-case compu time is $O(n^2)$. The algorithm requires space for arrays d[ ] J[ ], with few temporary variables.

3.6.2 'C' Code for Job Sequencing with Deadlines

'n' jobs, associated with job 'i' is an integer deadline d_i > and p_i > 0. To complete a job, one has to process the job machine for one unit of time. Only one machine is availabl feasible solution is a subset J of jobs such that each job in subset can be completed in deadline. Value of opt solution is sum of profits of jobs in J (i.e. Σp_i where i belc to J).

```
/* d[i]>= 1, where 1 <= i<=n are the deadlines an
   >= 1. The jobs are ordered such that
   p[1]>=p[2]>=p[n] */
/* J[i] is the ith job in optimal solution 1<=i<=k.
   Also, at termination
   d[J[i]]<=d[J[i + 1]], 1<=i<=k */
int JobSequence (int d[ ], int J[ ], int n)
```

```
{
int k, l, r, q;
    d[0] = J[0] = 0; /* Initialize */
    J[1] = 1;        /* Include job 1 */
    k = 1;
for (i=2; i<=n; i++)
{
/*Consider jobs in decreasing order of p[i]. Find
position for i and check feasibility of insertion */
    r=k;
        while ((d[J[r]]>d[i])&&(d[J[r]]!=r))
        r--;
    if ((d[J[r]]<=d[i])&&(d[i]>r))
    {
    /* insert i into J[ ]*/
        for (q=k;q>=(r+1);q--)
            J[q+1]=J[q];
        J[r+1]=i;
        k++;
    }
}
return(k);
}
```

Does the Algorithm Guarantee an Optimal Solution ?

We have to prove that "Greedy method described always guarantees an optimal solution to the problem.

Proof : Let (p_i, d_i), $1 \le i \le n$, define any instance of job sequencing problem. Let I be the set of jobs selected by greedy method. Let J be the set of jobs in optimal solution. Assume I<>J. If J<I, then J is not optimal. Also, if I < J, not possible. So, there exist jobs a and b such that a belongs to I and a does not belong to J, b belongs to J and b does not belong to I.

Let a be highest profit job such that a belongs to I and a does not belong to J. It follows from greedy method that $p_a \ge p_b$ for all b∈J but b does not belong to I. Consider feasible solutions S_i and S_j for I and J. Let i be job such that $i \in I$ and $i \in J$. Let i be scheduled from t to t + 1 in S_i and t' to t' + 1 in S_j. If t < t', then we can interchange the job scheduled in [t', t'+1]. The resulting schedule is also feasible.

If t'<t, then a similar transformation can be made in S_j. Thus, we can obtain S_i and S_j. Consider the interval $[t_a, t_{a+1}]$ in S_i in which a is scheduled. Let b be scheduled in S_j in this interval. From choice of a, $p_a \ge p_b$. Scheduling a from t_a to t_{a+1} in S_j and discarding job b gives feasible set J' = J-{b}∪{a} . Clearly, J' has profit value no less than J and differs from I in one less job than J does. Thus, I is optimal.

3.7 MINIMUM COST SPANNING TREES

- A tree is a connected graph with no cycles. A spanning tree is a subgraph of G that has all vertices of G and is a tree. A minimum spanning tree of a weighted graph G is the spanning tree of G whose edges sum to minimum weight.
- Fig. 3.4 shows a graph, one of its spanning trees and a minimum spanning tree. Minimum spanning trees are useful in many applications such as in finding the least amount of wire needed to connect group of computers, houses or cities.
- A minimum spanning tree minimizes the total length over all possible spanning trees.

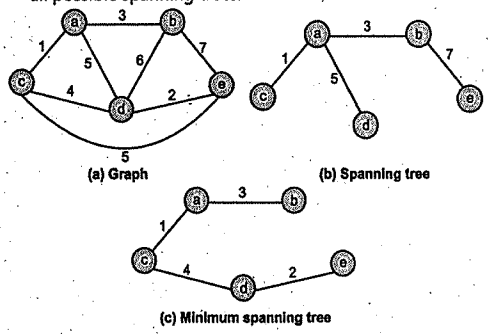


Fig. 3.4 : A Graph and its spanning trees

- The problem requires us to find a minimum spanning tree, and we would like to find the one efficiently. In theory, we could enumerate all the spanning trees of a weighted graph and simply choose the tree of least weight, but if the graph is a complicated one, this is not an easy and efficient way to get it. In this section, we shall study two better ways discovered in the 1950's by J. B. Kruskal and R. C. Prim.
- Both the algorithms produce a minimum spanning tree by adding an edge at a time at each stage making the best choice of next edge. Both are greedy algorithms. These two methods are popularly used to compute minimum spanning tree of a graph. There can be more than one minimum spanning tree of a graph.

The two methods are :

1. Prim's Algorithm.
2. Kruskal's Algorithm.

3.7.1 Connected Components

- An undirected graph is connected if there is at least one path between every pair of vertices in the graph. A connected component of a graph is a maximal connected subgraph; that is, every vertex in a connected component is reachable from vertices in the component.

- Consider graph G_1 drawn as Fig. 3.5. In this undirected graph, there is only one connected component, the graph G_1 itself.

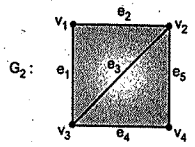


Fig. 3.5

- If we delete edges e_4 and e_5 from graph G_1, we get a graph G_2 with two connected components: ($\{v_1, v_2, v_3\}$, $\{e_1, e_2, e_3\}$) and ($\{v_4\}$, 0)

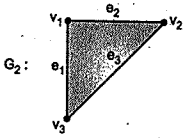


Fig. 3.6

3.7.2 Prim's Algorithm

All vertices of any connected graph are included in minimum cost spanning tree of a graph G. Prim's algorithm starts from one vertex and grows the rest of the tree one vertex at a time, by adding associated edge. This algorithm builds a tree by iteratively adding edges until a minimal spanning tree is obtained i.e. when all nodes are added. At each iteration, it adds to the current tree a minimum weight edge that does not complete a cycle. Let G = (V, E) be an original graph.

Let T be a spanning tree. T = (A, B) where, initially A and B are empty sets. Let us select an arbitrary vertex i from V and add it to set A. Now A = {i}. At each step prim's algorithm looks for the shortest possible edge <u, v> such that $u \in A$ and $v \in V - A$. It then adds v to A ($A = A \cup \{v\}$) and adds edge <u, v> to B. In this way, the edges in B form at any instant a minimum spanning tree for the vertices in A. We continue thus as long as $A \ne V$. To illustrate the algorithm, let us consider the graph in Fig. 3.7.

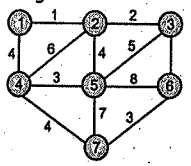


Fig. 3.7 : A Weighted graph

Let us select node 1 as the starting node. Following steps show the edge of minimum weight selected and set of vertices A.

Table 3.1

Step No.	Edge <u,v>	Set A
Initial	–	{1}
1	<1,2>	{1, 2}
2	<2,3>	{1, 2, 3}
3	<1,4>	{1, 2, 3, 4}
4	<4,5>	{1, 2, 3, 4, 5}
5	<4,7>	{1, 2, 3, 4, 5, 7}
6	<7,6>	{1, 2, 3, 4, 5, 7, 6}

When the algorithm stops, B contains the chosen ec
B = {<1,2>, <2,3>, <1,4>, <4,5>, <4,7>, <7,6>}.

The resultant spanning tree is drawn in Fig. 3.8 of we
= 17.

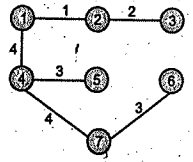


Fig. 3.8 : Minimum spanning tree

Following is an informal statement of algorithm :

Here, G is a graph and T is a spanning tree to be computed

```
(1) Let G = (V, E) and T = (A, B)
    A = {i} and B = ϕ
(2) Let i ∈ V, i is a start vertex
(3) A = A ∪ {i}
(4) While A ≠ V do
    begin
        find edge <u,v> ∈ E of minimum length
        such that u ∈ A and v ∈ V - A
        A = A ∪ {v} and
        B = B ∪ {<u,v>}
    end
(5) Stop
```

Implementation :

To obtain a simple implementation in any programmi language say 'C', suppose the vertices of G are number from 1 to n, so that V = {1, 2, ..., n}. Let matrix M gives t length of each edge and M[i][j] = ∞ if the edge <i,j> ∉ E i edge <i,j> does not exist. Let us use two arrays: Nearest[ ] a Min_Dist[ ].

Let T = (A, B) be the minimum spanning tree where initially and B are empty. For each vertex i ∈ V – A, Nearest[i] giv the vertex in A that is nearest to i. And for each vert i ∈ V – A the Min_Dist[i] gives the distance from i to th nearest vertex. For a vertex i ∈ A, we set Min_Dist[i] = -

In this way we can find out whether a vertex is in A or not. The set A, arbitrarily init alizes to {1}.

Following is the pseudo 'C' code for the same :

```
Prim (M[n,[n])
{
    /* Let A = {1} and B = ϕ */
    for(i=2;i≤n;i++)
    {
        Nearest[i] = 1;
        Min_Dist[i] = M[i,1];
    }
for(i=1;i≤n-1;i++)
    {
        minimum = -999;
        for(j=2;j≤n;j++)
        {
            if(0≤Min_Dist[j]<minimum)
                {
                    minimum = Min_Dist[j];
                    k = j;
                }
        }
        Union (B,Nearest[k],k)
/* Union is function which adds edge <Nearest(k),k> in B */
/* In brief union does B = B U <Nearest[k], k> */
        Min_Dist[k] = -1;
        for(j=2;j≤n;j++)
        {
            if(M[j][k]<Min_dist[j])
                {
                    Min_Dist[j] = M[j][k];
                    Nearest[j] = k;
                }
        }
    }
    return (B);
}
```

Here the array B contains all edges of minimum spanning tree. The computing time of algorithm Prim is $O(n^2)$, where n is the number of vertices in the graph G.

3.7.3 Kruskal's Algorithm

- We studied Prim's algorithm to find minimum spanning tree. Another way to construct a minimum spanning tree for G is to start with a graph T = (V, ϕ) consisting of the n vertices of G and having no edges. Each vertex is therefore, a connected component itself. As the algorithm proceeds, we shall always have a set of connected components, and in each connected component we shall have selected edges that form a spanning tree.
- As algorithm progresses, we add edge to T by examining edges from E. If the edge connects two vertices in two different connected components, then we add the edge to T. In other words, if the edge do not form a cycle in a T, then only it is added. If an edge joins two vertices of two different connected components, we add it to T. Consequently, the two connected components now form only one component.
- If it forms a cycle i.e. if the edge connects two vertices in the same component, then we discard the edge. At the end of the algorithm only one connected component remains, so T is then a minimum spanning tree for all the vertices of G. To build bigger and bigger component, we examine the edges of G in increasing order of their associated weights.
- To illustrate the method, consider the graph in Fig. 3.9. Let us arrange edges in the increasing order of their weights: <1,2>, <2,3>, <4,5>, <6,7>, <1,4>, <2,5>, <4,7>, <3,5>, <2,4>, <3,6>, <5,7>, and <5,6>.

Selection and addition of edges in step by step manner is shown in the following table 3.2 :

Table 3.2

Step No.	Edge Considered	Action	Connected Component
Initial	–	–	{1} {2} {3} {4} {5} {6} {7}
1	<1,2>	Add	{1,2} {3} {4} {5} {6} {7}
2	<2,3>	Add	{1,2,3} {4} {5} {6} {7}
3	<4,5>	Add	{1,2,3} {4,5} {6} {7}
4	<6,7>	Add	{1,2,3} {4,5} {6,7}
5	<1,4>	Add	{1,2,3,4,5} {6,7}
6	<2,5>	Rejected	{1,2,3,4,5} {6,7}
7	<4,7>	Add	{1,2,3,4,5,6,7}

When the algorithm stops, T contains the chosen edges <1,2>, <2,3>, <4,5>, <6,7>, <1,4> and <4,7>. This minimum spanning tree has weight as 17 and is drawn in Fig. 3.8.

In brief the algorithm can be stated as ;

Algorithm :

```
(1) Let G = (V, E) and T = (A, B)
(2) A = V and B = ∧, |A| = n and |B| = 0
(3) while (|B| ≤ n - 1) do
        begin
            find edge <u,v> of minimum length and add
            it to B only if addition
            of edge <u,v> does not complete a cycle in
            T.
        end
(4) stop
```


The graph T initially consists of the vertices of G and no edges. At each iteration, we add an edge <u,v> to T having minimum weight that does not complete a cycle in T. When T has (n – 1) edges, we stop.

Implementation :

To implement the algorithm, we have to handle a certain number of sets, which include vertices of each connected component. Two operations have to be carried out :

(i) Member (x) : It tells us which connected component the vertex x is member of.

(ii) Merge (u, v) : To merge two connected components u and v.

Algorithm :

```
Kruskal (G,T)
    begin
/* sort E in increasing order of weights */
/* G = (V, E) and T = (A, B) A = V and E = ∧ */
/* n = length (V) */
/* Initialize n sets, each containing a different
   element of v */
    while (|B|<n-1)do
        begin
            e = <u,v> the shortest edge not yet considered
            delete e from E
            U = Member (u)
            V = Member (v)
            if (U≠V)
                {
                    Merge (U,V)
                    Union (B,u,v)  //Add edge<u,v> to
                                   set B
                }
        end
    return (T)
end
```

Analysis of Kruskal's Algorithm :

Kruskal's algorithm considers the edges in the ascending order of weights. So weights of edges can be stored in the form of minheap. Then getting an edge with smallest weight takes O(1) time and deletion takes O(log n) time where n is the number of edges in a graph G. So total computing time of algorithm Kruskal is O(n log n) where n is the number of edges in a graph G.

3.8 DIJKSTRA'S ALGORITHM FOR SHORTEST PATH

- A weighted graph is a graph in which values are assig to the edges and the length of a path in a weighted gr is the sum of the weight of the edges in the p Let w (i, j) denotes the weight of an edge (i, j). In weigh graph, we often need to find a shortest path. The shor path between two given vertices is a path hav minimum length.
- This problem can be solved by a greedy algorithm Edsger W. Dijkstra often called as Dijkstra's Algorithm.
- Consider a directed graph G = (A, B). Each edge ha non-negative length. One of the nodes is the sou vertex. Suppose, we want to determine a shortest p from source vertex a to destination vertex Z.
- Let us use two sets of vertices visited and unvisited. Le denote set of visited vertices which contains th vertices that have been already chosen and mini distance from source is already known for every verte V. The U set contains all other vertices whose mini distance from the source is not yet known.
- Let an array Dist holds the shortest distance and ar Path holds the shortest path between source and each the vertex. At each step Dist[i] shows shortest dista between a and i and Path[i] shows shortest path betwe a and i.
- The basic idea of the algorithm is to determine minimum cost from i to one vertex at each iteration, sa mark j as visited and recalculate cost from i to each unvisited vertices going through j.

The Algorithm :

Initially a is the only vertex in V. At each step we add tc another vertex for which the shortest path from a has be determined. The array Dist is initialized by setting Dist[i] to t weight of the edge from a to i if it exists and to ∞ if not. determine which vertex to add to V at each step, we appl criteria of choosing the vertex j with the smallest distar recorded in Dist, such that j is not visited one. When we ad to V (set of visited vertices), we must update the entries Dist by checking, for each vertex k not in V, whether a pa through j and then directly to k is shorter than the previou recorded distance of k. That is, we replace Dist[k] by Dist[j] weight of edge from j to k if the latter quantity is lesser. Her is current selected vertex and let k be a vertex whose distar is updated. If distance is updated then update path also. Th path[k] becomes path of j followed by k.

In brief,

```
if Dist[k]>(Dist[j]+weight<j,k>)
    then
        Dist[k] = Dist[j] + w<j,k>
    and
        Path[k] = Path[j] U(k)
```

Here is an algorithm.

Algorithm :

Let G = (A, B) be a graph of n vertices where A = Set of vertices, B = Set of edges.

Let w(n, n) is a weight matrix and w(a, t) denotes weight of edge <a, t>.

(1) Initially, let V = {a} and U = V - {a}.
Here U is unvisited and V is set of visited vertices.
(2) Let Dist[t] = w (a, t) for every vertex t ∈ U.
(3) Select the vertex x in U that has the smallest index w.r.t. V.
(4) If x is the destination vertex where we want to reach from vertex a, then stop. Otherwise, change, V = V union {x} and U = U - {x}.
(5) For every vertex t in U, compute its distance with respect to V as
Dist[t] = min (Dist[t], Dist[t] + w (x, t))
(6) Repeat steps (3), (4) and (5).
(7) Stop.

Let us consider graph in Fig. 3.9.

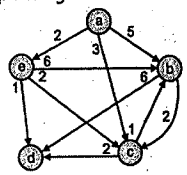


Fig. 3.9 : Directed weighted graph

(1) Initial Step :

The set V = {a}, a as source vertex.

and U = {b, c, d, e}: unvisited vertices

Dist[ ] = {–, 5, 3, ∞, 2}

can also be written as

Dist.	b	c	d	e
	5	3	∞	2

The above Dist[ ] array represents the current shortest distance between a and other vertices.

Path = {ab, ac, –, ae}.

(2) Now the distance to vertex e is the shortest, so e is added to set V.

V = {a, e}, let us update Dist array now.

Dist.	b	c	d	e
	5	3	6	2

Here, the distance of vertex d is updated to 6. Hence, path is also updated for vertex d by path of current selected vertex i.e. path of e.

Path = {–, ab, ac, aed, ae}

(3) Now the distance to vertex c among unvisited vertices is minimum, hence c is current selected vertex which gets to V.

∴ V = {a, e, c}

Let us update Dist array now.

Dist.	b	c	d	e
	4	3	5	2

Here the shortest distance between source a to b and d are updated as,

Dist[b] = min {5, Dist[c] + w (c,b)}
= min {5, 3 + 1}
= 4.

and Dist[d] = min {6, Dist[c] + w (c,d)}
= min {6, 3 + 2}
= 5

As shortest distance of b and d are updated, their respective paths are also updated as:

Path = {–, acb, ac, acd, ae}

Path vector can also be shown as follows :

Path	b	c	d	e
	acb	ac	acd	ae

(4) Now b is the vertex which has shortest distance and is unvisited.

Hence V = {a, e, c, b}

Dist	b	c	d	e
	4	3	5	2

Here, none of the shortest distance is updated hence path also remains unchanged.

Path	b	c	d	e
	acb	ac	acd	ae

(5) Now d is next selected vertex. And final distance and path vectors are same as above. Hence, the shortest distance between a and {b, c, d, e} are {4, 3, 5, 2} respectively. Also shortest path between a and {b, c, d, e} are {acb, ac, acd, ae} respectively.

The final two steps, add vertices b and d to V and yield the paths and distances shown in the following Fig. 3.10.

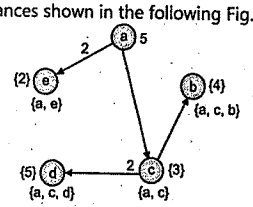


Fig. 3.10 : Shortest paths and distances

Implementation of Algorithm :

To implement algorithm in 'C', let us use adjacency matrix implementation as it facilitates random access to all the vertices of graph. Moreover, by storing the weights in the matrix, we can use the matrix to give weights as well as adjacencies. We shall place a special large value 9999 (to represent ∞) in positions for which the corresponding edge does not exist.

Pseudo C++ Function:

```
Short_Dist (int n, cost[ ][ ], Dist[ ])
    {
        int i,k,j,min;
        int final[ ];
    for (j=1;j<n;j++)
        {
            final[j] = 0;
            Dist[j] = cost[0][j];
        }
    for (i=1;i<n;i++)
        {
            min = 9999;
            for (k=1;k<n;k++)
                {
                    if (!final[k])
                        if (Dist[k]<min)
                            {
                                j = k;
                                min = Dist[k];
                            }
                }
            final[j] = 1;
            for (k=1;k<n;k++)
                if (!final[k])
                    if (min+cost[j][k]<Dist[k])
                        Dist[k] = min + Dist[j][k];
        }
    }
```

The time complexity of this algorithm with n vertices is O It can be reduced to O ((n + |E|) log n) where, E is numbe edges, if red-black trees are used.

MULTIPLE CHOICE QUESTIONS (MCQ's)

1. Which of the following standard algorithms is nc Greedy algorithm?
   (a) Dijkstra's shortest path algcrithm
   (b) Prim's algorithm
   (c) Kruskal algorithm
   (d) Huffman Coding
   (e) Bellmen Ford Shortest path algorithm
2. What is the time complexity of Huffman Coding?
   (a) O(N) (b) O(NlogN)
   (c) $O(N(\log N)^2)$ (d) $O(N^2)$
3. Which of the following is true about Kruskal and P MST algorithms? Assume that Prim is implemented adjacency list representation using Binary Heap a Kruskal is implemented using union by rank.
   (a) Worst case time complexity of both algorithms same.
   (b) Worst case time complexity of Kruskal is better th Prim
   (c) Worst case time complexity of Prim is better th Kruskal
4. Dijkstra's algorithm is also called the ..... shortest pa problem.
   (a) Multiple source (b) Single source
   (c) Single destination (d) Multiple destination
5. ..... solves the problem of finding the shortest pa from a point in a graph to a destination.
   (a) Kruskal's algorithm (b) Prim's algorithm
   (c) Dijkstra algorithm (d) Bellman ford algorithm
6. The result of prim's algorithm is a total time bound .....
   (a) O(logn) (b) O(m+n logn)
   (c) O(mn) (d) O(m logn)
7. The ..... process updates the costs of all t vertices V, connected to a vertex U, if we could impro the best estimate of the shortest path to V by includi (U,V) in the path to V.
   (a) Relaxation (b) Improvement
   (c) Shortening (d) Costing

ANSWERS

1. (e)	2. (b)	3. (a)	4. (b)	5. (c)
6. (b)	7. (a)			

EXERCISE

1. Explain the greedy strategy.
2. Write control abstraction for greedy strategy.
3. Write a greedy algorithm for knapsack problem.
4. Write a greedy algorithm for activity selection problem using recursion. What is the time complexity of the algorithm?
5. Write a greedy algorithm for activity selection problem without using recursion. Discuss time complexity for the same.
6. Write a greedy algorithm for job sequencing in unit time with deadlines and profits.
7. Consider the following instances of the Knapsack Problem : n = 3, m = 20, (p1,p2,p3) = (24,25,15) and (w1,w2,w3) = (18,15,20) Find the feasible Solutions.
8. With respect to greedy method define the following terms and explain briefly their significance.
   (i) Feasible solution (ii) Optimal solution
   (iii) Subset Paradigm.
9. Write an algorithm of minimum spanning tree and single source shortest path.

✠ ✠ ✠

Unit IV

DYNAMIC PROGRAMMING

4.1 DYNAMIC PROGRAMMING

- Dynamic programming has evolved into a major paradigm of algorithm design in computer science. However, its name is something of a mystery to many people. The name was coined in 1957 by Richard Bellman to describe a type of optimum control problem. Actually, the name originally described the problem more than the technique of solution. The sense in which programming is meant is "a series of choices". Like the programming of the radio station.
- The word dynamic conveys the idea that choices may depend on the current state, rather than being decided ahead of time. A radio show in which listener's phone in their requests might be said to be "dynamically programmed" in contrast with the usual format where the selections of songs are decided before the show begins. Bellman described a method of solution for "dynamic programming" problems, which has become the inspiration for many computer algorithms.
- The main feature of this method was that it replaced an exponential time computation by a polynomial time computation. That continues to be a common feature of dynamic programming algorithms.

4.1.1 The General Method

- Dynamic programming is an algorithm design method that can be used when the solution to a problem may be viewed as the result of a sequence of decisions.
- Like greedy method for many problems, it is not possible to make stepwise decisions (based only on local information) in such a manner that the sequence of decisions made is optimal.
- One way to solve such problems, is to try out all possible decision sequences. We could enumerate all decision sequences and then pick out the best. Dynamic programming often drastically reduces the amount of enumeration by avoiding the enumeration of some decision sequences that cannot possibly be optimal.
- In dynamic programming, an optimal sequence of decisions is arrived at by making explicit appeal to the principle of optimality.

4.1.2 Steps to Develop a Dynamic Programming Algorithm

There are Four Steps to Develop Algorithm are as Follows

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution, whic is the most creative work.
3. Compute the value of an optimal solution in a bottom-u fashion. We can also use recursive method.
4. Construct an optimal solution from computed informatio by making use of computed results.

Generic Problem Structure:

$$t_n = \begin{cases} \text{constant value} & \text{if trivial (p)} \\ \text{combine } f(p_1)\ f(p_2)\ \ldots\ f(p_n) & \text{otherwise} \end{cases}$$

4.2 COMPONENTS OF DYNAMIC PROGRAMMING

A Dynamic Programming Solution has Three Components

1. Formulate the answer as a recurrence relation or recursiv algorithm.
2. Show that the number of different instances of you recurrence is bounded by a polynomial.
3. Specify an order of evaluation for the recurrence.

To decide whether a problem can be solved using dynami programming method, the following three elements o dynamic programming should be considered:

1. Optimal substructure
2. Overlapping subproblems
3. Memorization

1. Optimal Substructure

- A problem exhibits optimal substructure if an optima solution to the problem contains within it optima solutions to subproblems. If a problem exhibit optimal substructure, then it means that dynami programming (and greedy method) might apply. A the optimal solution for the problem is built from the optimal solution of the subproblems, it is necessar to consider those subproblems which have ar optimal solution.
- The execution time of a dynamic programmin algorithm depends on the product of two factors: the overall number of subproblems and how man choices we look at for each subproblem.

- Dynamic programming uses optimal substructure in a bottom-up fashion. It first finds optimal solutions to subproblems. When the subproblems are solved, then it finds an optimal solution to the problem.

2. Overlapping Subproblems

- When a recursive algorithm revisits the same problem over and over again, then it is said that the optimization problem has overlapping subproblems. This is beneficial for dynamic programming. It solves each subproblem once and stores the answer in a table. This answer can be searched in constant time when required.
- This is contradictory to divide-and-conquer strategy where a new problem is generated at each step of recursion.

3. Memorization

- Generally dynamic programming maintains a table for solutions of subproblems. But it uses the control structure similar to the recursive algorithm.
- In a memorized recursive algorithm, an entry is maintained in a table for solution of each subproblem. Initially all entries contain a special value which indicates that entry is not yet used. For each subproblem which is encountered for the first time, its solution is computed and stored in the table.
- Next time for that subproblem, its entry is searched and value is used. This can be implemented using hashing.

4.2.1 Generic Dynamic Programming

Algorithm for generic dynamic programming is given below:

Algorithm:

```
function solve(p)
begin
  if known (p) then
    return (lookup (p))
  else
    x = compute (p)
    save(p, x)
    return x
end

function compute(p)
begin
  if trivial(p)
    then return(trivial_sol(p))
  else
    divide p into subproblems p1, p2, ... pn
    S1 = solve(p1), .....
    Sn = solve (pn)
    return (combine(S1, S2, ..., Sn))
end
```

Table 4.1

solve + compute	Recursive definitions
lookup	Maintains a table to record (Problem p, solution-for-p)
trivial	Gives a solution (always a constant value) for p
trivial_sol	Judges whether we can use trivial to compute p
divide	It depends on the problem structure
combine	It depends on the problem structure

4.3 PECULIAR CHARACTERISTICS AND USE OF DYNAMIC PROGRAMMING

- Solution to a problem is viewed as a result of a sequence of decisions.
- Avoids enumeration of some decision sequences that cannot be possibly optimal.
- An optimal sequence of decisions is arrived at by making an explicit appeal to the 'principle of optimality'.
- Principle of optimality states that: An optimal sequence of decisions has property that whatever the initial state and decision are, the remaining decisions must constitute an optimal decision sequence with regard to the state resulting from the first decision.
- In the greedy method only one decision sequence is ever generated. In dynamic programming, many decision sequences may be generated.
- However, sequences containing suboptimal sequences cannot be optimal and so will not be generated.
- Two approaches for dynamic programming:
  Let $(x_1, x_2, ..., x_n)$ be variables.
  1. **Forward Approach:** Decision x_i is made in terms of optimal decision sequences for $x_{i+1}, ..., x_n$.
  2. **Backward Approach:** Decision x_i is made in terms of optimal decision sequences for $x_1, x_2, ..., x_{i-1}$.
- Dynamic programming is a technique for solving problems with overlapping subproblems. Typically these subproblems arise from a recurrence relating a solution to a given problem with solutions to its smaller subproblems of the same type.
- Rather than solving overlapping subproblems again and again, dynamic programming suggests solving each of the smaller subproblem only once and recording the results in a table from which we can obtain a solution to the original problem.
- Applicability of dynamic programming to an optimization problem requires the problem to satisfy the principle of optimality: an optimal solution to any of its instances must be made-up of optimal solutions to its sub-instances.

4.3 Limitations of Dynamic Programming

- Dynamic programming can be applied to any problem that observes the principle of optimality. Roughly stated, this means that partial solutions can be optimally extended with regard to the *state* after the partial solution instead of the partial solution itself.
- For example, to decide whether to extend an approximate string matching by a substitution, insertion, or deletion, we do not need to know exactly which sequence of operations was performed to date.
- In fact, there may be several different edit sequences that achieve a cost of C on the first p characters of pattern P and t characters of string T. Future decisions will be made based on the consequences of previous decisions, not the actual decisions themselves.
- Problems do not satisfy the principle of optimality if the actual operations matter, as opposed to just the cost of the operations. Consider a form of edit distance where we are not allowed to use combinations of operations in certain particular order properly formulated, however, most combinatorial problems respect the principle of optimality.
- The biggest limitation on using dynamic programming is the number of partial solutions we must keep track of. For all of the examples we will see, the partial solutions can be completely described by specifying the stopping places in the input. This is because the combinatorial objects being worked on (strings, numerical sequences, and polygons) all have an implicit order defined upon their elements. This order cannot be scrambled without completely changing the problem.
- Once the order is fixed, there are relatively few possible stopping places or states, so we get efficient algorithms. If the objects are not firmly ordered, however, we have an exponential number of possible partial solutions and are doomed to need an infeasible amount of memory.

4.4 COMPARISON OF DIVIDE-AND-CONQUER AND DYNAMIC PROGRAMMING TECHNIQUES

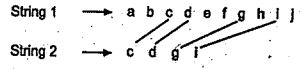
- Both divide-and-conquer and dynamic programming work in a bottom-up fashion. They divide large problem into subproblem. When the subproblems are solved, then solution to the problem is obtained by combining results of subproblems.
- In dynamic programming, subproblems are overlapping. When a recursive algorithm revisits the same problem over and over again, then it is said that the optimizati problem has overlapping subproblems. This is benefic for dynamic programming. It solves each subproble once and stores the answer in a table. This answer can searched in constant time when required. This contradictory to divide-and-conquer strategy where new problem is generated at each step of recursion.

4.5 LONGEST COMMON SUBSEQUENCE (LCS)

- LCS is the problem of finding the longest subsequen common to all sequences in a set of sequences.
- It differs from the longest common substring problem Unlike substrings, subsequences are not required occupy consecutive positions within the origin sequences.

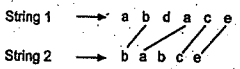
e.g.

1.



c d g i, d g i, g i are common subsequences occurred i string 1 and string 2, but longest common subsequence c d g i whose length is 4.

2.



b a c e is one solution to find LCS

a b c e is another solution to find LCS

To find LCS matching characters need not be continuous, bu should be appeared sequentially one after another.

LCS problem can be solved using recursion as well as dynami programming.

4.5.1 LCS using Recursion

LCS problem can be solved using recursion where stack i applied implicitly.

LCS Recursive Algorithm

```
int LCS(i, j)
{
    if (A[i] == '\0' || B[j] == '\0')
        return 0;
    else if (A[i] == B[j])
        return 1 + LCS (i + 1, j + 1);
    else
        return max (LCS (i + 1, j), LCS (i, j + 1));
}
```

When LCS problem is solved using recursion, some subproblems may be repeatedly solved which leads to time complexity 2^n in worst case, where n is the length of the string. Recursive solution can be improved with the help of memorization, time complexity reduces from 2^n to $m \times n$, where m is length of first string and n is length of second string.

4.5.2 LCS using Dynamic Programming

Dynamic programming follows bottom up approach to solved LCS.

LCS Dynamic Programming Algorithm

```
if (A[i] = = B[j])
    LCS[i, j] = 1 + LCS [i - 1, j - 1]
else
    LCS[i, j] = max (LCS[i - 1, j], LCS [i, j - 1])
```

A → | b | d | (1 2) B → | a | b | c | d | (1 2 3 4)

m = length of A = 2

n = length of B = 4

Construct a matrix of order (m+1) × (n+1) 0th row and 0th column are reserved to initialize with zero

		0	1	2	3	4
	0	0	0	0	0	0
b	1	0				
d	2	0				

Row indicates characters of first string i.e. A column indicates characters of second string i.e. B. Fill up all entries row wise. If i^{th} and j^{th} characters are same then add 1 into the value which is at position [i – 1, j – 1] and store at position [i, j]. If i^{th} and j^{th} characters are not same then find maximum value from the positions [i – 1, j] and [i, j – 1] and store that max value at [i, j] position.

		a	b	c	d
	0	0	0	0	0
b →	1	0	1	1	1
d →	2	0	1	1	2
			b		d

So the size of the longest common subsequence is 2. To find LCS, start from 2 m a matrix and trace back until we get 0.

Whenever we trace back from 2 to 0 and we move diagonally, note down the respective character to find longest common subsequence.

∴ LCS is "bd" of length 2.

Time complexity of LCS using dynamic programming is

$O(m \times n)$

where, m = number of characters in first string

n = number of characters in second string

Let us solve one more example.

Find LCS for str 1 → s t o n e and

str 2 → l o n g e s t

		l	o	n	g	e	s	t
	0	0	0	0	0	0	0	0
s	1	0	0	0	0	0	1	1
t	2	0	0	0	0	0	1	2
o	3	0	0	1	1	1	1	2
n	4	0	0	1	2	2	2	2
e	5	0	0	1	2	2	3	3
				o	n		e	

∴ LCS is "one" of length 3.

4.6 MATRIX MULTIPLICATION

- We are given a sequence (chain) $A_1 A_2 A_3 \dots A_n$ of n matrices, and we wish to find the product $A_1 A_2 A_3 \dots A_n$.
- We can evaluate the product using the standard algorithm for multiplying pairs of matrices as a sub-routine once we have parenthesized it to resolve all ambiguities in how the matrices are multiplied together.
- A product of matrices is fully parenthesized if it is either a single matrix or the product of two fully parenthesized matrix products, surrounded by parenthesis. Matrix multiplication is associative, so all parenthesizations yield the same product. For example, if the chain of matrices is $(A_1 A_2 A_3 A_4)$, then the product $A_1 A_2 A_3 A_4$ can be fully parenthesized in the following different ways :

(1) $(((A_1, A_2), A_3), A_4)$
(2) $(A_1 (A_2 A_3) A_4)$
(3) $(A_1 ((A_2 A_3) A_4))$
(4) $(A_1 (A_2 (A_3 A_4)))$
(5) $((A_1 A_2) (A_3 A_4))$

The way of parenthesizing a chain of matrices have a dramatic impact on the cost of evaluating the product.

When we multiply 2 matrices A_1 and A_2 that do not have same number of rows and columns, we can multiply A_1 and A_2 if and only if number of columns of A_1 is equal to the number of rows of A_2. For example, if A_1 is of size 2 × 3 and A_2 is of size 3 × 4.

- ➢ The resultant matrix is 2 × 4. It will have 2 rows and 4 columns.
- ➢ So the total number of multiplications is 2 × 3 × 4.
- ➢ If we want to multiply A_1 of size M × N and A_2 of size N × P, the resultant matrix will be of size M × P and it will take M × N × P multiplications.

Chained Matrix Multiplication Problem :

- Given a sequence (chain) $A_1 A_2 A_3 \dots A_n$ of n matrices. Find the product $A_1 A_2 A_3 \dots A_n$ by full parenthesizing the matrices to minimize the number of multiplications.

For example, consider matrices

A_1 of size 5 × 3
A_2 of size 3 × 4
A_3 of size 4 × 6
A_4 of size 6 × 5

Different parenthesizing gives different number of multiplications :

(1) $(A_1 ((A_2 A_3) A_4))$ takes (3.4.6) + (3.6.5) + (5.3.5) = 237 multiplications.
(2) $(A_1 (A_2 (A_3 A_4)))$ takes 255 multiplications.
(3) $((A_1 A_2) (A_3 A_4))$ takes 280 multiplications.
(4) $(((A_1 A_2) A_3) A_4)$ takes 330 multiplications.
(5) $((A_1 (A_2 A_3) A_4))$ takes 312 multiplications.

- In case of four matrices, there are only five ways to parenthesize the matrices. But with 'n' number of matrices the number of ways to parenthesize them grows exponentially. So we don't try for all the possibilities. The solution is to divide the problem into subproblems. Here, we use the principle of optimality which is said to apply if an optimal solution to an instance of a problem always contains optimal solutions to all substances. If $(A_1 ((A_2 A_3) A_4))$ is the optimal order then we know that $(A_2 A_3) A_4$ is the optimal order for $A_2 A_3 A_4$.
- We need to find the best way to parenthesize the chain of matrices to minimize the number of scalar multiplications. This can be done by dividing the problem into subproblems and then finding the optimal solution to the subproblem.
- Suppose we have matrix-chain $A_1 \dots A_n$. This chain is divided into subproblems $A_1 \dots A_k$ and $A_{k+1} \dots A_n$. But the problem is to find the value of k. We can find k by looking at the optimal solution of each of the subproblems.

Consider the example of 4 matrices of size.

$A_1 : 5 \times 3$
$A_2 : 3 \times 4$
$A_3 : 4 \times 6$
$A_4 : 6 \times 2$

- First we look at the possible chains of length two i.e. we see how many values of k are there for chains of length two ? Then see the possible chains of length three, then chains of length four etc. Here, we need some kind of organization to help us find the optimal way to make chains of length three out of chains of length two.

We keep the dimensions of the matrices in one dimensional array.

index	0	1	2	3	4
d	5	3	4	6	2

Each matrix A_i has dimensions d[i – 1] and d[i]. We keep the number of multiplications for chains of length one, two, three and four in a two dimensional array called N.

- N[1][1] : Holds the number of multiplications to multi from A_1 to A_1. This is 0 as it is a chain of length one. [i] where i = 1 to n is always zero.
- N[1][2] : Holds the minimum number of multiplications multiply A_1 to A_2. This is a chain of length 2. Add N[2] N[3] [4] to the table, which are the chains of length two
- N[1][3] : Holds the minimum number of multiplications one chain of length 3, $N_{1,3}$. This can be either $(A_1 A_2) A_3$ $A_1 (A_2 A_3)$. Here we have two possibilities and we m decide on the minimum. We use the term k to sh where we decide to put the parenthesis.

$$N_{i,j} = \min \{N_{i,k} + N_{k+1,j} + d_{i-1} * d_k * d_j\}$$

where, $i \le k < j$

For example, consider the chains of length 3, $N_{1,3}$ a $N_{2,4}$. For $N_{1,3}$ there are two possibilities $(A_1 A_2) A_3$ $A_1 (A_2 A_3)$. We have entries in two dimensional array $(A_1 A_2)$ and $(A_2 A_3)$ both. So we compare both of them a take the minimum one.

Pseudocode for Chained Matrix Multiplication :

```
int chained_mult (int n, int d[ ], index P[ ] [ ])
{
    int i, j, k, dia;
    int N[ ] [ ] = new int[1 to n] [1 to n];
    for (i=1;i<=n;i++)
            N[i] [i] = 0
    for (dia=1;dia<=n-1;dia++)
        for (i=1;i<=n-dia;i++)
            {
                j = i + dia;
                N[i] [j] = min (N[i] [k] + N[k+1] [j] + d[i-1] *
                    d[k] * d[j]); for i ≤ k < j
                P[i] [j] = value of k which minimizes N[i][j]
            }
return N[1] [n];
}
```

- In this algorithm, we compute N with three nested f loops. The outer loop is executed n times. The inner loo is executed at most n times. So the total running time $O(n^3)$. This algorithm is developed by Godbole in 197 There are still better algorithms for chained matr multiplication with complexities as follows :

$O(n^2)$ → developed by Yao in 1982 and

O(n log n) → developed by Hu and Shing in 1984.

- The array P computed in the above algorithm remembers values of k which minimizes corresponding value of N[i][j] and stores in P[i][j]. These P values can be used to find parenthesization order of matrices recursively for A_1 to A_n denoted as A1...n as follows:

$$A_{1...n} = (A_{1...P[1, n]} A_{P[1, n]+1 ... n})$$

Function to print matrix chain denoting optimal solution is as follows:

```
function DispMatrixChain( P, i , j )

  if i = j
  {
      output "A"
      output Value of i
  }
  else
  {
      output "("
      DispMatrixChain( P, P[i,j] )
      DispMatrixChain( P, P[i,j]+1, j )
      output ")"
  }
```

Consider the following example :

Matrix	A_1	A_2	A_3	A_4	A_5	A_6
Dimensions	30 × 35	35 × 15	15 × 5	5 × 10	10 × 20	20 × 25

∴ 1-D matrix dimension array:

Index	0	1	2	3	4	5	6
d	30	35	15	5	10	20	25

2-D array N and P (upper value denotes N[i][j] and lower value denotes P[i][j]):

Index	j=1	j=2	j=3	j=4	j=5	j=6
i=1	0	15750 k = 1	7875 k = 1	9375 k = 3	11875 k = 3	15125 k = 3
i=2		0	2625 k = 2	4375 k = 3	7125 k = 3	10500 k = 3
i=3			0	750 k = 3	2500 k = 3	5375 k = 3
i=4				0	1000 k = 4	3500 k = 5
i=5					0	5000 k = 5
i=6						0

Initialization:

Number of matrices n = 6

N[1] [1] = 0, N[2][2] = 0, N [3][3] = 0,

N[4][4] = 0, N[5][5] = 0, N[6][6] = 0

Step 1: Compute values for $i \le k < j$

For i = 1, j = 2, k = 1

N[1] [2] = N[1] [1] + N[2] [2] + $d_0 \times d_1 \times d_2$

= 0 + 0 + 30 × 35 × 15

= 15750

For i = 2, j = 3, k = 2

N[2] [3] = N[2] [2] + N[2] [3] + $d_1 \times d_2 \times d_3$

= 0 + 0 + 35 × 15 × 5

= 2625

For i = 3, j = 4, k = 3

N[3] [4] = N[3] [3] + N[4] [4] + $d_2 \times d_3 \times d_4$

= 0 + 0 + 15 × 5 × 10

= 750

For i = 4, j = 5, k = 4

N[4] [5] = N[4] [4] + N[5] [5] + $d_3 \times d_4 \times d_5$

= 0 + 0 + 5 × 10 × 20

= 1000

For i = 5, j = 6, k = 5

N[5] [6] = N[5] [5] + N[6] [6] + $d_4 \times d_5 \times d_6$

= 0 + 0 + 10 × 20 × 25

= 5000

Step 2: Compute values for $i \le k < j$

For i = 1, j = 3, 1 ≤ k < 3

N[1] [3] = min {N[1] [1] + N[2] [3] + $d_0 \times d_1 \times d_3$, N[1] [2] + N[3] [3] + $d_0 \times d_2 \times d_3$}

= min {[0 + 2625 + 30 × 35 × 5], [15750 + 0 + 30 × 15 × 5]}

= min {7875, 18000}

= 7875 for k=1

For i = 2, j = 4, 2 ≤ k < 4

N[2] [4] = min {N[2] [2] + N[3] [4] + $d_1 \times d_2 \times d_4$, N[2] [3] + N[4] [4] + $d_1 \times d_3 \times d_4$}

= min {[0 + 750 + 35 × 15 × 10], [2625 + 0 + 35 × 5 × 10]}

= min {6000, 4375}

= 4375 for k=3

For i = 3, j = 5, 3 ≤ k < 5

N[3] [5] = min {N[3] [3] + N[4] [5] + $d_2 \times d_3 \times d_5$, N[3] [4] + N[5] [5] + $d_2 \times d_4 \times d_5$}

= min {[0 + 1000 + 15 × 5 × 20], [750 + 0 + 15 × 10 × 20]}

= min {2500, 3750}

= 2500 for k=3

For i = 4, j = 6, 4 ≤ k < 6

N[4] [6] = min {N[4] [4] + N[5] [6] + $d_3 \times d_4 \times d_6$, N[4] [5] + N[6] [6] + $d_3 \times d_5 \times d_6$}

= min {[0 + 5000 + 5 × 10 × 25], [000 + 0 + 5 × 20 × 25]}

= min {6250, 3500}

= 3500 for k=5

Step 3: Compute values for $i \le k < j$

For i = 1, j = 4, 1 ≤ k < 4

N[1] [4] = min {N[1] [1] + N[2] [4] + $d_0 \times d_1 \times d_4$, N[1] [2] + N[3] [4] + $d_0 \times d_2 \times d_4$, N[1] [3] + N[4] [4] + $d_0 \times d_3 \times d_4$}

= min {[0 + 4375 + 30 × 35 × 10], [15750 + 750 + 30 × 15 × 10]}

= min {14875, 21000, 9375}

= 9375 for k=3

For i = 2, j = 5, 2 ≤ k < 5

N[2] [5] = min {N[2] [2] + N[3] [5] + $d_1 \times d_2 \times d_5$, N[2] [3] + N[4] [5] + $d_1 \times d_3 \times d_5$, N[2] [4] + N[5] [5] + $d_1 \times d_4 \times d_5$}

= min {[0 + 2500 + 35 × 15 × 20], [2625 + 1000 + 35 × 5 × 20], [4375 + 0 + 35 × 10 × 20]}

= min {13000, 7125, 11375}

= 7125 for k=3

For i = 3, j = 6, 3 ≤ k < 6

N[3] [6] = min {N[3] [3] + N[4] [6] + $d_2 \times d_3 \times d_6$, N[3] [4] + N[5] [6] + $d_2 \times d_4 \times d_6$, N[3] [5] + N[5] [6] + $d_2 \times d_5 \times d_6$}

= min {[0 + 3500 + 15 × 5 × 25], [750 + 5000 + 15 × 10 × 25], [2500 + 0 + 15 × 20 × 25],

= min {5375, 9500, 10000}

= 5375 for k=3

Step 4: Compute values for $i \le k < j$

For i = 1, j = 5, 1 ≤ k < 5

N[1] [5] = min {N[1] [1] + N[2] [5] + $d_0 \times d_1 \times d_5$, N[1] [2] + N[3] [5] + $d_0 \times d_2 \times d_5$, N[1] [3] + N[4] [5] + $d_0 \times d_3 \times d_5$, N[1] [4] + N[5] [5] + $d_0 \times d_4 \times d_5$}

= min {[0 + 7125 + 30 × 35 × 20], [15750 + 2500 + 30 × 5 × 20], [7875 + 1000 + 30 × 5 × 20], [9375 + 0 + 30 × 10 × 20],

= min {28125, 27250, 11875, 15375}

= 11875 for k=3

For i = 2, j = 6, 2 ≤ k < 6

N[2] [6] = min {N[2] [2] + N[3] [6] + $d_1 \times d_2 \times d_6$, N[2] [3] + N[4] [6] + $d_1 \times d_3 \times d_6$, N[2] [4] + N[5] [6] + $d_1 \times d_4 \times d_6$, N[2] [5] + N[6] [6] + $d_1 \times d_5 \times d_6$}

= min {[0 + 5375 + 35 × 15 × 25], [2625 + 3500 + 35 × 5 × 25], [4375 + 5000 + 35 × 10 × 25]} [7125 + 0 + 35 × 20 × 25]}

= min {18500, 10500, 18125, 24625]}

= 10500 for k=3

Step 5: Compute values for $i \le k < j$

For i = 1, j = 6, 1 ≤ k < 6

N[1] [6] = min {N[1] [1] + N[2] [6] + $d_0 \times d_1 \times d_6$, N[1] [2] + N[3] [6] + $d_0 \times d_2 \times d_6$, N[1] [3] + N[4] [6] + $d_0 \times d_3 \times d_6$, N[1] [4] + N[5] [6] + $d_0 \times d_4 \times d_6$, N[1] [5] + N[6] [6] + $d_0 \times d_5 \times d_6$}

= min {[0 + 10500 + 30 × 35 × 25], [15750 + 5375 + 30 × 5 × 25], [7875 + 3500 + 30 × 5 × 25], [9375 + 5000 + 30 × 10 × 25], [11875 + 0 + 30 × 20 × 25]}

= min {36750, 32375, 15125, 21875, 26875}

= 15125 for k=3

Matrix multiplication chain can be obtained using $A_{1...n} = (A_{1...P[1, n]} A_{P[1, n]+1 ... n})$ as follows:

$A_{1...6} = (A_{1...P[1, 6]} A_{P[1, 6]+1 ... 6})$

$= (A_{1...3} A_{4...6})$ as P[1, 6]

$= ((A_{1...P[1, 3]} A_{P[1, 3]+1 ... 3})(A_{4...P[4, 6]} A_{P[4, 6]+1 ... 6}))$

$= ((A_{1...1} A_{2...3})(A_{4...5} A_{6...6}))$ as P[1, 3]=1, P[4, 6]

$= ((A_1(A_{2...P[1, 3]} A_{P[2, 3]+1 ... 3}))((A_{4...P[4, 5]} A_{P[4, 5]+1 ... 5})A_6))$

$= ((A_1(A_{2...2} A_{3...3}))((A_{4...4} A_{5...5})A_6))$ as P[2, 3]=2, P[4, 5]

$= ((A_1(A_2 A_3))((A_4 A_5)A_6))$

Hence optimal matrix multiplication chain for given instance $((A_1(A_2 A_3))((A_4 A_5)A_6))$.

4.7 BELLMAN FORD SHORTEST PATH

It is used to find length of shortest path from source vertex to any other destination vertex for directed graph with negative edges.

Dijkstra's shortest path algorithm will not work properly for graph with negative edges.

Bellman Ford algorithm works on the principle – relaxation of all edges n – 1 times, where n is total number of nodes of a graph. Relaxation is applied n – 1 times because longest path from source to any destination vertex consists of maximum n – 1 edges.

Relaxation from u to v is given below

if (d[u] + c (u, v) < d[v])

d[v] = d[u] + c(u, v)

Let us consider graph to apply Bellman Ford algorithm

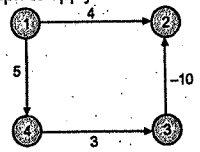


Fig. 4.1

Total number of vertices = n = 4

Apply relaxation n – 1 = 4 – 1 = 3 times to find length of shortest path from (1) to any destination source vertex is 1

Initialization

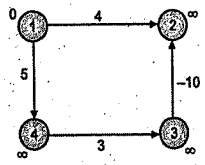


Fig. 4.2

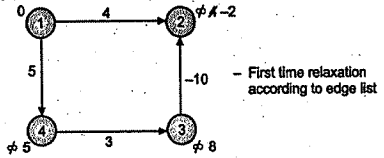
edge list = {(1, 2), (1,4), (4, 3), (3, 2)}

= Relax edge (1, 2) u = 1, V = 2

d[1] = 0, c(1, 2) = 4, d[2] = ∞

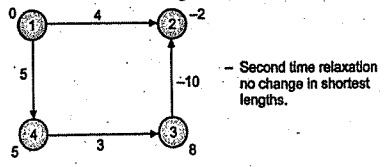
if (d[u] + c(u, v) < d[v]) ⇒ if (0 + 4 < ∞)

d[v] = d[u] – c (u, v) d[2] = 4



– First time relaxation according to edge list

Fig. 4.3



– Second time relaxation no change in shortest lengths.

Fig. 4.4

For third time relaxation we will also not get any change. Therefore length of shortest path from 1st vertex to any other destination vertex is as follows :

d [0 –2 8 5]

1 2 3 4

Time complexity of Bellman Ford.

As every edge is relaxed n – 1 times.

O(|E| |V| – 1)

O(|V| |E|)

If both V and E are same

$O(n^2)$

It graph is complete graph.

Time complexity = $O(n^3)$

Drawback of Bellman ford algorithm is that it does not work for graph which consists of negative edge cycle.

E.g.

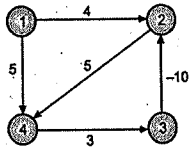


Fig. 4.5

The above graph consists of cycle 2 – 4 – 3 with negative edge.

Bellman ford algorithm fails to work on graph with negative edge cycle.

Bellman and Ford Algorithm to Complete Shortest Path

Algorithm BellmanFord (V, Cost, dist, n)

```
//Single Source / all destinations shortest paths with -ve
edge costs.
{
   for i=1 to n
   {   dist[i] = cost[v, i];
   }  for K =2 to n - 1
   {
      For each u such that u! = v & u has at least one
      incoming edge
      {
      for each <i, u> in the graph
      {
      if (dist[u]> dist[i] + cost[i u])
         dist[u] = dist[i] + cost[i, u];
      }
   }
}
```


4.8 FLOYD-WARSHALL ALGORITHM

- In this section, we are going to study an algorithm known as Floyd-Warshall algorithm. It uses dynamic programming approach. Let G(V, E) is a directed graph. Assume that negative-weight edges are allowed in the graph. The algorithm considers the intermediate vertices of a shortest path. An intermediate vertex of a simple path P = {v_1, v_2, ..., v_m} is any vertex of P other than v_1 or v_m. The intermediate vertex ∈ {v_2, v_3, ..., v_{m-1}}.
- For any pair of vertices i, j ∈ V, let P is a minimum-weight path among (i, j) and its intermediate vertices ∈ {1, 2, ...,k}.
  - If k is not an intermediate vertex of path p, then shortest path from i to j using intermediate vertices {1, 2, ..., k – 1} is same as shortest path from i to j using intermediate vertices {1, 2, ..., k}.
  - If k is an intermediate vertex of path P, then P is made up of two shortest paths : First path from i to k and second path from k to j.
  - Both these paths are shortest paths which use intermediate vertices from {1, 2, ..., k – 1}.
  - To compute all pairs shortest paths, use the following steps
  - Represent the graph using matrix W which follows the following recurrence

$$w_{ij} = \begin{cases} 0 & , \text{ if } i = j \\ \text{weight}(i\ j) & , \text{ if } i \neq j \text{ and edge } <i\ j> \in E \\ \infty & , \text{ if } i \neq j \text{ and edge } <i\ j> \notin E \end{cases}$$

  - Let D(n) = d_{ij}(n), where d_{ij}(k) denotes the weight of a shortest path from vertex i to vertex j for which all intermediate vertices are in the set {1, 2, ..., k}. When k = 0, $d_{ij}(0) = w_{ij}$.

$$d_{ij}(k) = \begin{cases} w_{ij} & , \text{ if } k = 0 \\ \min[d_{ij}(k-1)\ d_{ik}(k-1)\ d_{kj}(k-1)] & , \text{ if } k \geq 1 \end{cases}$$

- The algorithm All Pairs Shortest Paths takes two inputs : the matrix W[n, n] and number of vertices n. It returns matrix D(n) of shortest-path lengths as output.

The algorithm is as given below :

Algorithm AllPairsShortestPaths (W, n)

```
begin
    D(0) = W
    for k = 1 to n
        for i = 1 to n
            for j = 1 to n
                d_ij(k) = min [d_ij(k-1), d_ik(k-1) + d
                kj(k-1)]
            end for
        end for
    end for
    return D(n)
end
```

Analysis of All Pairs Shortest Paths Algorithm :

Computing time of this algorithm depends on the compu time of d_{ij}(k) which is constant, that is, O(1). Hence compu time of algorithm is $O(n^3)$.

4.9 APPLICATION OF DYNAMIC PROGRAMMING

- Dynamic programming amounts to break down optimization problem into simpler sub-problems, storing the solution to each sub-problem so that sub-problem is only solved once. This is why dyn programming has its name.
- The flexibility of the dynamic programming technique reality appreciated by means of publicity to a huge of applications. To recognize and contribute to the sli side research on optimization problems such as classical superior preventing trouble and r appropriate useful resource allocation in the pr framework of approximate dynamic programn complex decision making problems below uncertainty reliability analysis, useful resource allocation, org collection manipulation and hazard control.
- Dynamic Programming can be applied in variety of a such as follows:
  - Information theory.
  - Control theory.
  - Bioinformatics.
  - Operations research.
  - Computer science - Theory, Graphics, Arti Intelligence, etc.

MULTIPLE CHOICE QUESTIONS (MCQ s)

1. Dynamic programming is based on the principle of:
   (a) Optimality (b) Heruistics
   (c) Regularity (d) None of the above
2. The following strategies complexity function genera in the form of recurrence relation:
   (a) Dynamic (b) Divide and conquer
   (c) a and b both (d) None of the a and b
3. Which of the following methods can be used to s the Knapsack problem?
   (a) Brute force algorithm
   (b) Recursion
   (c) Dynamic programming
   (d) All of the mentioned

4. Which of the following is/are property/properties of a dynamic programming problem?
   (a) Optimal substructure
   (b) Overlapping subproblems
   (c) Greedy approach
   (d) Both optimal substructure and overlapping subproblems
5. If an optimal solution can be created for a problem by constructing optimal solutions for its subproblems, the problem possesses _____ property.
   (a) Overlapping subproblems
   (b) Optimal substructure
   (c) Memorization
   (d) Greedy
6. If a problem can be broken into subproblems which are reused several times, the problem possesses _____ property.
   (a) Overlapping subproblems
   (b) Optimal substructure
   (c) Memorization
   (d) Greedy
7. If a problem can be solved by combining optimal solutions to non-overlapping problems, the strategy is called _____
   (a) Dynamic programming
   (b) Greedy
   (c) Divide and conquer
   (d) Recursion
8. In dynamic programming, the technique of storing the previously calculated values is called _____
   (a) Saving value property
   (b) Storing value property
   (c) Memorization
   (d) Mapping
9. When a top-down approach of dynamic programming is applied to a problem, it usually _____
   (a) Decreases both, the time complexity and the space complexity
   (b) Decreases the time complexity and increases the space complexity
   (c) Increases the time complexity and decreases the space complexity
   (d) Increases both, the time complexity and the space complexity
10. Which of the following problems is NOT solved using dynamic programming?
    (a) 0/1 knapsack problem
    (b) Matrix chain multiplication problem
    (c) Edit distance problem
    (d) Fractional knapsack problem
11. Which of the following problems should be solved using dynamic programming?
    (a) Mergesort
    (b) Binary search
    (c) Longest common subsequence
    (d) Quick sort

ANSWERS

1. (a)	2. (c)	3. (d)	4. (d)	5. (b)
6. (a)	7. (c)	8. (c)	9. (b)	10. (d)
11. (c)				

EXERCISE

1. Explain dynamic programming approach.
2. State the principle of optimality and explain in brief.
3. Construct OBST for (a_1, a_2, a_3) = (int, char, float).
   Let p(1: 3) = (0.5, 0.1, 0.05) and
   q(0: 3) = (0.15, 0.1, 0.05, 0.05). Also compute w(i, j), r(i, j,) and c(i, j).
4. Generate sets S^i, $0 \le i \le 4$ when weights (w_1, w_2, w_3, w_4) = (1, 11, 21, 23) and profits (p_1, p_2, p_3, p_4) = (11, 21, 31, 33).
5. Write control abstraction for following algorithm strategies.
   (a) Divide and Conquer
   (b) Greedy Method
   (c) Dynamic programming
6. Compare Greedy and Dynamic strategies.
7. Find an optimal parenthesization of matrix chain multiplication for sequence of following dimensions: 5, 10, 3, 12, 5, 50, 6.
8. What do you mean by Longest Common Subsequence?
9. Explain Bellman Ford algorithm with suitable example.
10. Write a short note on applications of dynamic programming.

✠ ✠ ✠

Unit V

BACKTRACKI

5.1 INTRODUCTION

- We studied three algorithm strategies till now. Backtracking is a more intelligent variation of these approaches. The principle idea is to construct solutions one component at a time and evaluate such partially constructed solutions as follows: If a partially constructed solution can be developed further without violating the problem's constraints, it is done by taking the first remaining legitimate option for the next component. If there is no legitimate option for the next component, no alternatives for any remaining component need to be considered. In this case, the algorithm *backtracks* to replace the last component of the partially constructed solution with its next option.
- In brief, the task is to determine algorithm to find solutions to specific problem not by following a fixed rule of computation, but by trial and error. The common pattern is to decompose the trial and error process into partial tasks. Often these tasks are most naturally expressed in recursive terms and consists of the exploration of a finite number of subtasks.
- It is convenient to implement this kind of processing by constructing a tree of choices made, called the state-space-tree. Its root represents an initial state before the search for solution begins. The nodes of the first level in the tree represent the choices made for the first component of a solution, the nodes of the second level represent the choices for the second component and so on. A node in a state space-tree is said to be promising if it corresponds to a partially constructed solution that may still lead to a complete solution; otherwise it is called non-promising.
- Leaves represent either non-promising dead ends or complete solutions found by the algorithm. In the majority of cases, a state space tree for backtracking algorithm is constructed as depth first search. If the current node is promising, its child is generated by adding the first remaining legitimate option, for the next component of a solution, and the processing moves to this child. If the current node turns out to be non-promising, the algorithm backtracks to the node's parent to consider the next possible option for its last component, if there is no such option, it backtracks one more level up and so on. Finally, if the algorithm reac complete solution to the problem, it either stops (one solution is required) or backtracks to cor searching for other possible solutions.

5.2 THE GENERAL METHOD

- In the backtrack method, the desired solutic expressible as an n-tuple $(x_1, x_2, ..., x_n)$, where the chosen from some finite set S_i.
- Often the problem to be solved calls for finding vector that maximizes (or minimizes or satisfie criterion function $P(x_1, x_2, ..., x_n)$. Sometimes it seel vectors that satisfy P.

The Control Abstraction :

- We assume that all answer nodes are to be found anc just one. Let $(x_1, x_2, ..., x_i)$ be a path from the root of space tree to a node.
- Let $T(x_1, x_2, ..., x_i)$ be the set of all possible values for such that $(x_1, x_2, ..., x_{i+1})$ is also a path to a problem sta $T(x_1, x_2, x_3, ..., x_n) = \theta$.

1. Recursive Backtrack Algorithm

```
Algorithm Backtrack (k)
    x[1], x[2], ..., x[k - 1] solution vectors
    begin
        for (each x[k] ∈ T(x[1], ..., x[k - 1])) do
            begin
                if (B_k(x[1], x[2], ..., x[k]) ≠ 0) then
                    begin
                        if (x[1], x[2], ..., x[k] is a p
                        to an answer node)
                            then print (x[1: k]);
                        if (k<n) then Backtrack (k +
                    end
            end
    end
```

2. Iterative Backtrack Method

```
Algorithm IBacktrack (n)
    begin
        k = 1
        while (k≠0) do
            begin
                if (there remains an untried x[k] ∈ T
                (x[1], x[2], ..., x[k - 1]) and
```

```
B[k] (x[1], ..., x[k]) is a path to an
        answer node)
then print (x[1:k]);
    k = k + 1;
    else k = k - 1;
  end
end
```

5.3 PECULIAR CHARACTERISTICS AND USE

- Backtracking is an algorithm design technique for solving problems in which the number of choices grows at least exponentially with their instance size. This technique constructs a solution one component at a time, trying to terminate the process as soon as one can ascertain that no solution can be obtained as a result of the choices already made.
- Backtracking employs a state space tree as its principle mechanism, which is a rooted tree whose nodes represent partially constructed solutions.
- The backtracking algorithm has as its virtue the ability to yield the same answer for fewer than m trials. Its basic idea is to build-up the solution vector one component at a time and to use modified criterion functions $p_i(x_1, ..., x_i)$ and sometimes called bounding functions to test whether the vector being formed has any chance of success.
- The major advantage of this method is this : if it is realized that the partial vector $(x_1, x_2, ..., x_i)$ can in no way lead to an optimal solution, then $x_1, ..., x_n$ possible test vectors can be ignored entirely.

5.3.1 State Space Tree

- In all the problems, we are going to find solution for given n weights $(w_1, w_2, w_3, ..., w_n)$. There are two ways to state the solution : for example, first way is to form a solution array as (w_1, w_2, w_4) or form an array of indices (1, 2, 4). Here solution will be of size k, $(1 \le k \le n)$. In general, all solutions are k-tuples $(x_1, x_2, ..., x_k)$. Different solutions may have different size tuples. Second way to state the solution is fixed-size n-tuple $(x_1, x_2, ..., x_n)$ where $x_i = 0$ /1, $1 \le i \le n$. If object i belongs to solution, then $x_i = 1$, otherwise $x_i = 0$.
- In backtracking approach and branch-and-bound approach the problem solutions are systematically searched using a tree organization of solution spaces. Each node in this tree defines a problem state. All paths from the root to other nodes define the state space of the problem. All the states s for which the path from the root to s defines a tuple in the solution space are called solution states.
- Solution states s for which the path from the root to s defines a tuple that is a member of the set of solutions of the problem are called answer states. The tree organization of the solution space is called as the state space tree.
- The tree which is independent of the problem instance being solved is called static tree. The tree which depends on the problem instance being solved is called dynamic tree. For given problem, when a state space tree is generated it is easy to determine problem states, then determine which problem states are solution states, then determine which solution states are answer states.
- Generation of problem states can be done in two ways :
  1. Backtracking
  2. Branch-and-bound.

  Both backtracking and branch-and-bound approaches start construction of nodes from the root of tree. A node which has been constructed, but whose children are not yet constructed is called a live node. A live node whose children are being constructed is called an E-node. A node which is not to be expanded or all of whose children are constructed is called a dead node. As the construction of tree proceeds, E-node goes on changing. To kill live nodes without generating their children, bounding functions are used. In all the topics ahead, our constraints are that in solutions x_i is always an integer, $1 \le i \le n$. No two x_i's are same. Multiple instances represent the same subset for example, (2, 3) is same as (3, 2).

5.4 N-QUEENS PROBLEM

- The n-queens problem is to place n queens on an n × n chessboard, so that no two queens attack each other by being in the same row or in the same column or on the same diagonal.
- For n = 1, the problem has a trivial solution, and it is easy to see that there is no solution for n = 2 and n = 3. So let us consider the 4-queens problem and 8-queens problem to solve them by the backtracking technique.

5.4.1 Four-Queens Problem

- In a Four-Queens problem, we have to place 4 queens on the chessboard such that no two queens attack each other by being in the same row or in the same column or on the same diagonal.
- The row of each queen can be fixed so that only column needs to be decided.
- Let the queen 1, 2, 3 and 4 be placed in the row 1, 2, 3, and 4 respectively as shown in Fig. 5.1. Since each of the four queens has to be placed in its own row, all we need to do is to assign a column for each queen on the board presented in Fig. 5.1.

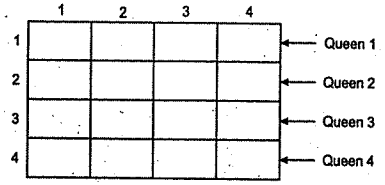


Fig. 5.1 : Chessboard for Four-Queens problem

- We start with the empty board and then place queen 1 in the first possible position that is Board [1] [1] as shown in Fig. 5.2 (a).

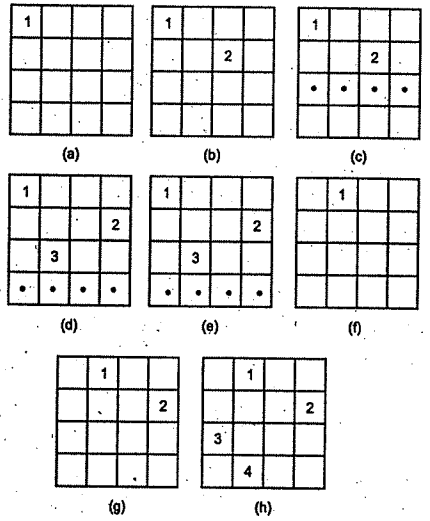


Fig. 5.2 ; Backtrack solution for Four-Queens problem

- Fig. 5.2 shows backtrack solution to the Four-Queens problem. We started with an empty board and one queen has been placed as shown in Fig. 5.2(a).
- Now we place queen 2, after trying unsuccessfully columns 1 and 2 in the first acceptable position for it, which is square (2, 3), the square in row 2 and column 3 (Fig. 5.2(b)). This proves to be a dead end because there is no acceptable position for queen 3 (Fig. 5.2(c)). So, the algorithm backtracks and puts queen 2 in the next possible position at (2, 4) (Fig. 5.2 (d)).
- Then queen 3 is placed at (3, 2), which proves to be another dead end (Fig. 5.2(e)). The algorithm then backtracks and puts queen 1 at position (1, 2), queen 2 at (2, 4), queen 3 at (3, 1) and queen 4 at (4, 3), which is a solution to the problem as shown in Fig. 5.2(f), Fig. 5.2(g) and Fig. 5.2(h) respectively.
- The state space tree to this search is given below :

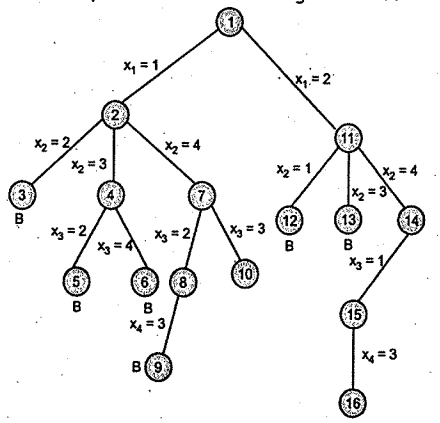


Fig. 5.3 : State space tree for Four-Queens problems

- Nodes are numbered level wise. Node at wh backtracking occurs is represented with B. Because rows which queens are to be placed are already known, we j have to find out column. x_i in the above state space t denotes column number i. If other solutions need to found, the algorithm can simply resume its operation which it stopped.

5.4.2 Eight-Queens Problem

- In an Eight-Queens problem, we have to place 8 quee on the chessboard such that no two queens attack ea other by being in the same row or in the same column on the same diagonal.
- The row of each queen can be fixed so that only colun needs to be decided. That is, queens 1 to 8 are placed the rows 1 to 8 respectively as shown in Fig. 5.4. Sin each of the eight queens has to be placed in its own ro all we need to do is to assign a column for each queen the board as presented in Fig. 5.4.

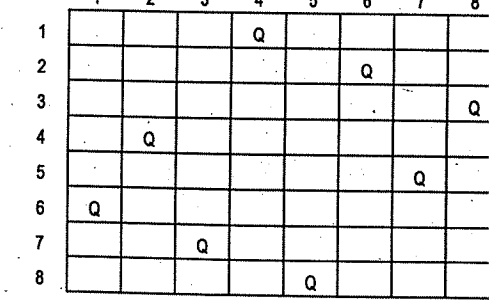


Fig. 5.4 : One possible solution to the Eight-Queens problem

Algorithm for N-Queens Problem

- Let us generalize the problem and consider an n × n chess board and try to find all the ways to place n non-attacking queens. We observed from 4-queens problem that we can let $(x_1, ..., x_n)$ represent a solution in which x_i is the column of the i^{th} row where the i^{th} queen is placed. The x_i's will all be distinct since no two queens can be placed in the same column. Now how do we test whether two queens are on the same diagonal?
- If we imagine the chessboard squares being numbered as the indices of the two dimensional array a[1 : n, 1 : n], then we observe that every element on the same diagonal that runs from the upper left to the lower right has the same row-column value.
- For example, in Fig. 5.4 consider the queen at position a[4, 2]. The squares that are diagonal to this queen (running from the upper left to the lower right) are a[3, 1], a[5, 3], a[6, 4], a[7, 5] and a[8, 6]. All these squares have a difference of 2 between row and column values i.e. for all, (row – column) is same.
- Also, every element on the same diagonal that goes from the upper right to the lower left has the same (row + column) value. Suppose two queens are placed at position (i, j) and (k, l). Then by the above rules, they are on the same diagonal only if (i – j) = (k – l) or (i + j) = (k + l).
  The first equation implies :
  $$j - l = i - k$$
  The second implies :
  $$j - l = k - i$$
- Therefore, two queens are on the same diagonal if and only if |j – l| = |i – k|.
- PlaceQueen(k, i) algorithm returns a boolean value that is true if the k^{th} queen can be placed in column i. It tests both whether i is distinct from all previous values x[1], ..., x[k – 1] and whether there is no other queen on the same diagonal. Its computing time is (k – 1).
- Using algorithm PlaceQueen we can define the general backtracking method as given by algorithm discussed in control abstraction and give a precise solution to the n-queens problem. The array x[ ] is global. The algorithm PlaceQueen is invoked by NQueens algorithm.

Algorithm PlaceQueen(k ,i)

```
begin
    for j = 1 to k - 1
    begin
    if (x[j]=i)          // Two queens are on the same
                         column
    or (abs(x[j] - i) = abs (j - k) )   // same diagonal
        then
            return false;
    end for
    return true
end
```

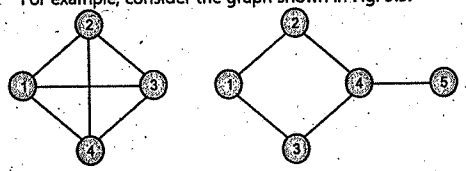
- Let us see an algorithm which gives all possible solutions to place n queens on an n × n chessboard in non-attacking positions. The algorithm is called for the first time as : NQueens (1, n). Here is the algorithm to plane n queens on a n × n chessboard.

Algorithm NQueens(k, n)

```
begin
    for i = 1 to n      // as there are n columns
    begin
        if PlaceQueen (k, i) then
        begin
            x [k] = i
            if (k = n) then
                print solution x [1 .. n ]
            else
                NQueens (k + 1, n)
            end if
        end if
    end for
end
```

5.5 HAMILTONIAN CYCLES

- In a connected graph, a path which starts at a vertex, visits every vertex of graph once and returns to its starting position is called as a Hamiltonian cycle. It is suggested by Sir William Hamilton. Let G = (V, E) be a connected graph of n vertices.
- Let $v_1, v_2, ..., v_n$ be its vertices. Then $v_1, v_2, ..., v_{n+1}$ is called as a Hamiltonian cycle if it begins at v_1, visits other vertices in the order $v_1, v_2, ..., v_{n+1}$ once such that E contains an edge $<v_i, v_{i+1}>$ for $1 \le i \le n$ where each v_i is distinct, except v_1 and v_{n+1} which represent the same vertex.
- For example, consider the graph shown in Fig. 5.5:



Graph 1 Graph 2

Fig. 5.5

- Graph 1 contains a Hamiltonian cycle 1, 2, 3, 4, 1. Graphs 2 has no Hamiltonian cycle. Let us see an algorithm which gives all possible Hamiltonian cycles in a graph. Let G[n, n] is an adjacency matrix for graph.
  G[i, j] = 1 if an edge $<i, j> \in E$.
  G[i, j] = 0 if an edge $<i, j> \notin E$.
- Let an array x[1 ... n] denotes a Hamiltonian cycle where x[i] denotes the i^{th} vertex visited in the path. Algorithm HamiltonianCycle is called with vertex 1 initially as HamiltonianCycle (1).

Algorithm HamiltonianCycle(k)

```
begin
    repeat
    {   //find next vertex to be visited and store at
        position x[k].
        FindNextVertex (k);
        //if no new vertex is available, return
        if (x[k]=0)
            return
        //if all the vertices visited, then print
        Hamiltonian cycle.
    k = n
    displayHCycle x[1..n]
        // continue to find next vertex in the path
        HamiltonianCycle (k+1)
    } until (false);
end
```

- Algorithm FindNextVertex(k) searches a vertex which is connected to vertex k – 1 and which is unvisited (means does not appear in x[1 .. k–1].
- If no such vertex is found, then x[k] = 0. If k = n, then it is checked whether x[n] and x[1] are connected or not. Initially array x[ ] is set to 0.

Algorithm FindNextVertex(k)

```
begin
    repeat
    {
    //Find next vertex
        x[k] = (x[k] +1) mod (n+1)
    //If no new vertex available, return
        if (x[k]=0)
            return
    //If new vertex available, check whether it is
    connected to previous vertex
    if(G(x[k - 1], x[k]) ≠ 0)
    begin
    //check whether vertex k is already visited
        for p = 1 to k - 1
            if (x[p]=x[k])
                break;
        end for
        /* if vertex k is unvisited, and k = n, then
        check whether it is connected to first
        vertex */
        if (p=k)        // means vertex k is unvisited
        if ((k=n)and(G[x[k]x[1]]≠0))
            return
    } until (false)
end
```

We have already seen travelling salesperson problem whic a Hamiltonian cycle. If all the edges have cost m and there n edges in Hamiltonian cycle, then its cost is mn.

5.6 SUM OF SUBSETS PROBLEM

Given n distinct weights $(w_1, w_2, ..., w_n)$, finding combinations of these weights whose sums are M, is ca the sum of subsets problem. We will consider fixed-solution $(x_1, x_2, ..., x_n)$ where $x_i = 1$ if object i is includec solution, otherwise $x_i = 0$. Consider the following example.

SOLVED EXAMPLES

Example 5.1 : *Let $(w_1, w_2, w_3, w_4) = (10, 5, 7, 8)$ are weig then find all combinations of these weights whose sums M = 15.*

Solution : The variable-sized solutions are (10, 5, 0, 0) and 0, 7, 8). The solution space can be stated using BFS to show possible solutions using tree organization, as given below : Here the nodes are numbered as in breadth-first-search. the tree, root has n children. From the left, first child leads the subtree which defines all subsets containing weig $(w_1, ..., w_n)$. Second child of root leads to the subtree wh defines all subsets containing weights $(w_2, ..., w_n)$, but not w_1 Third child leads to the subtree which defines all subs containing weights $(w_3, ..., w_n)$, but not w_1 and w_2, and so on fixed-size solution tuples are used, then solutions (1, 1, 0, 0) and (0, 0, 1, 1). In the first solution, $w_1 + w_2 = 10$ = 15, hence (w_1, w_2) are included in the solution. In the seco solution, $w_3 + w_4 = 7 + 8 = 15$, hence (w_3, w_4) are included the solution.

As n = 4, there are total $2^4 = 16$ possible solution tuples. paths from the root to a leaf node define the solution spa The tree organization is shown below in Fig. 5.7.

Here each node at level k leads to two edges : left ed labelled $x_i = 1$ which denotes inclusion of w_i, and right ed labelled $x_i = 0$ which denotes that w_i is not included solution.

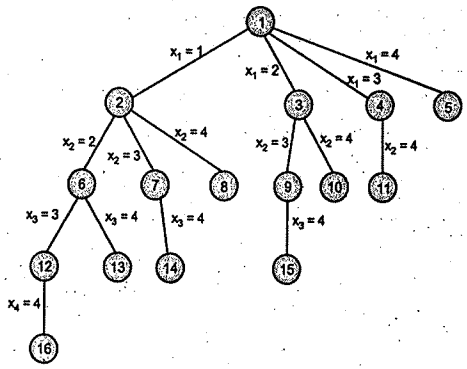


Fig. 5.6 : Solution space

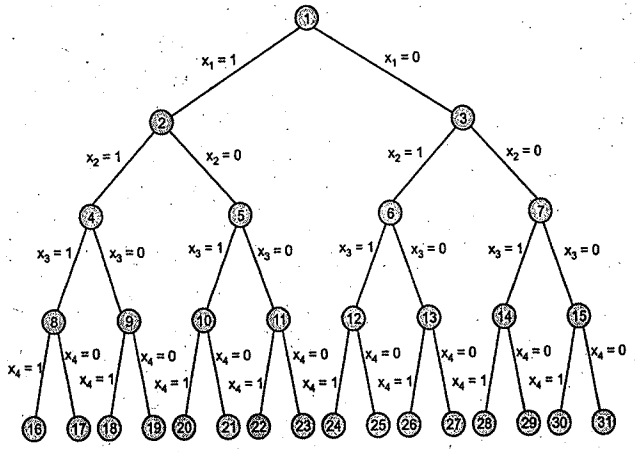


Fig. 5.7 : Solution space for n = 4

In backtracking algorithm, we need a bounding function to kill live nodes which do not lead to answer nodes. For the sum of subsets problem. The bounding function can be defined as,

$B_k(x_1, ..., x_k)$ = true iff

$$\sum_{i=1}^{k} w_i x_i + \sum_{i=k+1}^{n} w_i \geq M \text{ and}$$

$$\sum_{i=1}^{k} w_i x_i + w_{k+1} \leq M.$$

In simple words, if $x_k = 1$, then $\sum_{i=1}^{k} w_i x_i + \sum_{i=k+1}^{n} w_i > M$ ... (5.1)

We will use this result in the algorithm. The algorithm SumOfSubsets(a, k, b) has three inputs : Value of 'a' denotes $\sum_{i=1}^{k-1} w_i x_i$, object k to be considered next, value of 'b' denotes $\sum_{i=k}^{n} w_i$

When algorithm is called with SubOfSubsets(a, k, b), then $[x_1, ..., x_{k-1}]$ have already been determined. Assume that weights w[1, ..., n] are in increasing order. Also the algorithm assumes that $w_1 \leq M$ and $\sum_{i=1}^{n} w_i \geq M$.

Algorithm SumOfSubsets(a, k, b)

```
begin
   //Generate left child of state space tree considering
   xk = 1
   x[k] = 1
   if (a + w[k] = M)
   then //subset is found so display it. Obviously x[k+1
   ---, n] contains zeroes.
        Display x[1..k]
   else if (a + w[k] + w[k+1] ≤ M)
        then  //include kth object and check for (k+1)th
                object
              SumOfSubsets (a + w[k], k+1, b - w[k]);
        end if
   end if
   //Generate right child of state space tree considering
   xk = 0
   if(a+b - w[k] ≥ M) and (a+w[k+1] < M)
   then
        x[k] = 0
        SumOfSubsets (a, k+1, b - w[k]);
   end if
end
```

Initially the algorithm is called as SumOfSubsets (0, 1, $\sum_{i=1}^{n} w_i$).

Let us solve few examples for sum of subsets problem.

Example 5.2 : *Given n = 4 weights (w_1, w_2, w_3, w_4) = (7, 11, 13, 24). Find all possible subsets whose sums are M = 31 using sum of subsets algorithm.*

Solution : The state space tree is shown below in Fig. 5.8. Ea node shows a, k, b values.

So there are two possible subsets for which sum is M : (1, 1, 0) and (1, 0, 0, 1).

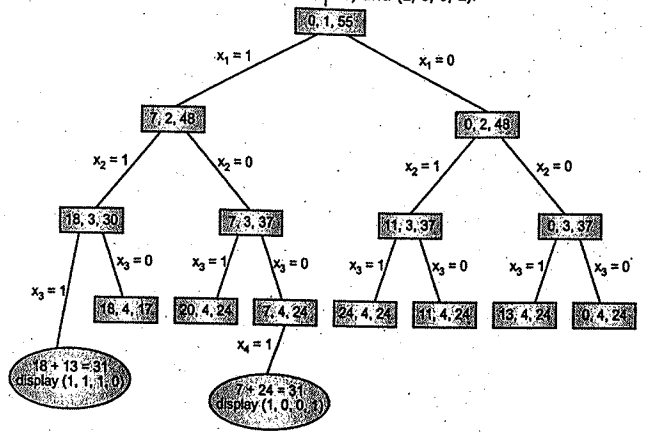


Fig. 5.8 : State space tree

Example 5.3 : *Given n = 6 weights*

w = {5, 10, 12, 13, 15, 18 } and

M = 30.

Find all possible subsets for which sum = M using Sum of Subsets algorithm.

Draw the generated partial state space tree.

Solution : The state space tree is shown below in Fig. 5.9. there are total 3 solutions :

P denotes (1, 1, 0, 0, 1, 0).

Q denotes (1, 0, 1, 1, 0, 0).

R denotes (0, 0, 1, 0, 0, 1).

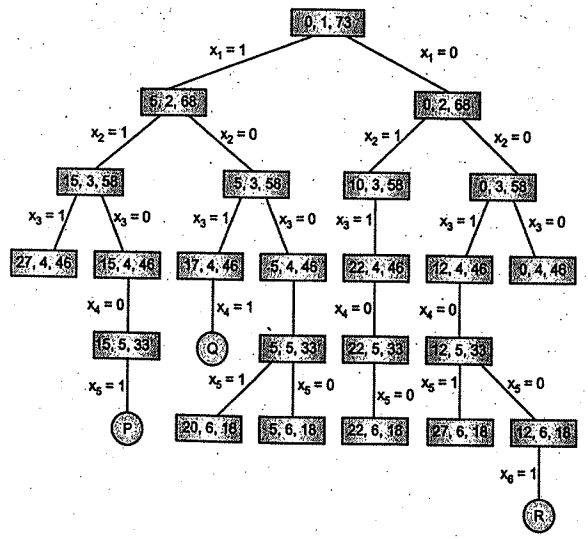


Fig. 5.9 : State Space Tree

Example 5.4 : *Given n = 7 weights w = {5, 7, 10, 12, 15, 18, 20} and M = 35. Find all possible subsets for which sum is M using SumOfSubsets algorithm.*

Draw the generated state space tree partially.

Solution : The state space tree is shown in Fig. 5.10.

Hence there are total 3 solutions :

P denotes (1, 0, 1, 0, 0, 0, 1)

Q denotes (1, 0, 0, 1, 0, 1, 0)

R denotes (0, 0, 0, 0, 1, 0, 1)

In general, a full state space tree for n weights contains $2^n - 1$ internal nodes from which calls could be made.

The tree may have 2^n leaf nodes which do not generate calls.

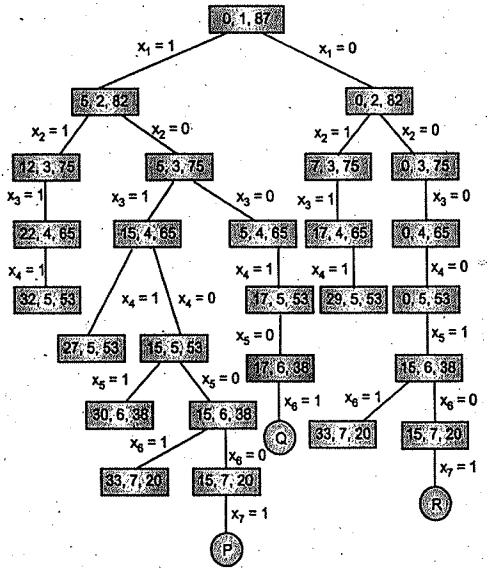


Fig. 5.10 : State space tree

5.7 GRAPH COLOURING PROBLEM

- Given a graph, it is always possible to colour the nodes of a graph such that no two adjacent nodes have same colour. It is always desired that the number of colours required are minimum. This is a graph colouring problem.

Graph 1 has 4 nodes and requires only 2 colours.

Graph 2 has 5 nodes and requires only 2 colours.

Graph 3 has 4 nodes and requires only 4 colours.

Graph 4 has 5 nodes and requires only 3 colours.

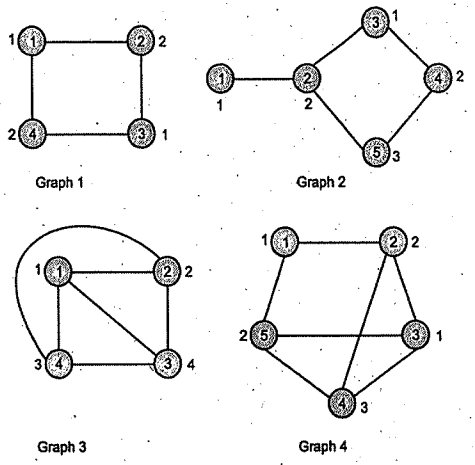


Fig. 5.11

For example, consider the graphs shown below :

- If a graph G can be coloured with minimum m colors, then m is called as the chromatic number of the graph. Finding the minimum value of m is called the m-colourability optimization problem. Application of graph coloring is to colour all the regions of a map such that no two adjacent regions are of same colour. Each region can be considered as a node and an edge represents two adjacent regions. Let us see the algorithm to find all possible ways to colour a graph G using m colours.
- Let a graph is represented as an adjacency matrix G[n, n]. G[p, q] = 1 denotes that there is an edge <p, q> in graph. G[p, q] = 0 denotes that there is no edge <p, q>. Let n denotes the number of vertices in G. Let there are m colours denoted as 1, 2, ..., m. The graph coloring solution is given by an array x[1 : n] where x_i denotes the colour of node i in solution. Initially solution x[1 : n] is set to 0 for all elements. The algorithm starts from vertex 1.

Algorithm mColouring(k)

```
begin
    repeat
    {
        // First find the next possible colour for
           node k.
        // Otherwise set its colour to 0.
        FindNextColour(k);
        // If colour of node k is 0, then stop because
        // New colour is not available.
        if (x[k] = 0)
            then return
        // If all the nodes are coloured, then display
           solution.
        if (k = n)
            display solution x [1 .. n]
        // else continue process for next node k+1
        else
            mColoring (k+1)
        endif
    } until (false)
end
```

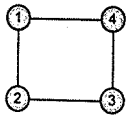

- This algorithm will print all possible coloring solutions x[1 ... n]. Algorithm FindNextColour will assign the next available colour to node k, else assigns 0. The algorithm is given below. The algorithm starts from vertex 1. That is, it will be called as mColoring (1) initially.

Algorithm FindNextColour(k)

```
begin
    repeat
    {
    //Get next highest colour in x[k]
    x[k] = (x[k] + 1) mod(m+1)
    //if no colour available, return
    if (x[k]=0)
        return
    //Check whether the new colour is not same as
    colour of adjacent nodes
    for p = 1 to n
    begin
        if G[k,p] = 0 and x[k] = x[p]
            break
    end for
    if (p=n+1)
        return //      new colour found
    //Otherwise continue to find next colour
    } until (false)
end
```

Computing time of this algorithm is O(mn) for all legal colours.

A graph with 4-nodes and all possible 2 coloring is shown in Fig. 5.12 and all possible 3 colouring is shown in Fig. 5.13.



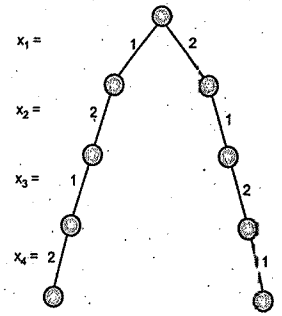


Fig. 5.12 : Possible 2-coloring for graph G

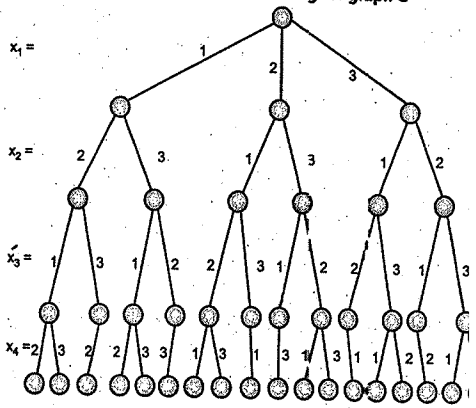


Fig. 5.13 : Possible 3-colouring for graph G

So using 2 colours, there are only 2 possible ways and using colours, there are 18 possible coloring ways for the giv graph.

- The number of internal nodes in the state space tree $\sum_{i=0}^{n-1} m^i$, which is upper bound on the computing time mColoring algorithm. For each internal noc FindNextColour algorithm is called which requires O(m time.
- Hence, the total time is bounded by,

$$\sum_{i=0}^{n-1} m^{i+1} n = \sum_{i=0}^{n} m^i n == O(nm^n)$$

5.8 BRANCH AND BOUND APPROACH

5.8.1 General Strategy

- We have seen different algorithmic strategies. In this un we are going to see branch-and-bound approach. W know that the graph can be searched using two way Depth First Search (DFS) or Breadth First Search (BFS).

- In the branch and bound approach, at each node we calculate a bound on the possible value of solutions. Calculation of the bounds is combined with either DFS or BFS. The calculated bound is used to decide which node should be expanded first. The nodes which can be expanded (explored) are termed as live nodes.

5.8.2 General Characteristics

The Branch-and-Bound Terminology Works in Two Ways:

1. If the live nodes are explored in the reverse order of their creation like DFS, then it is termed as Last In First Out (LIFO) search. It uses data structure stack.
2. If the live nodes are searched in the order of their creation like BFS, then it is termed as First In First Out (FIFO) search. It uses data structure queue.

- Actually BFS and DFS are special cases of Least Cost (LC) search. Branch-and-bound uses a priority list to store nodes that have been generated but not yet explored. A heap can better represent a priority list. It uses bounding functions to avoid generation of sub trees that do not contain solution node (an answer node).
- The technique is sufficiently powerful and is often used in practical applications. The behavior of this technique depends on the bound value. With a better bound, we may get an optimal solution more quickly. Sometimes, we may have to spend time at each node.

5.9 LEAST COST (LC) SEARCH

- BFS and DFS are special cases of LC search. In BFS, the live nodes are considered in FIFO order. In DFS, the live nodes are considered in LIFO order. But in LC search, that node is considered immediately which has a very good chance of getting the search to a solution node quickly. This quick search can be done using proper **Ranking Function** $\hat{c}(.)$ for live nodes.
- The ranking function helps to select next live node. The ranking function considers the additional efforts needed to reach an answer node from the live node. The number of levels from the nearest answer node can be considered as **Ranking**.

5.10 BOUNDING

- Actually BFS, DFS and LC search differ only in the sequence in which they visit nodes in the tree. In branch-and-bound approach, all the children of a live node are generated before another live node is considered.
- Let a cost function $\hat{c}(.)$ is such that $\hat{c}(x) \le c(x)$ and it provides lower bounds on solutions which can be obtained from x.
- Let u(x) is upper bound on minimum cost of solution, then all live nodes for which $c(x) \ge \hat{c}(x) > u(x)$ will be killed because they exceed upper bound. Each time a new answer node is found, the upper bound value should be updated.

5.11 0/1 KNAPSACK PROBLEM

- We already know the 0/1 knapsack problem. It is to maximize $\sum p_i x_i$ such that $\sum p_i x_i \le M$. In other words,

$$\text{minimise } -\sum_{i=1}^{n} p_i x_i \text{ and } \sum_{i=1}^{n} w_i x_i \le M \text{ and}$$

$$x_i = 0/1, \text{ for } 1 \le i \le n$$

- Every leaf node in the state space tree is an answer node for which $\sum_{i=1}^{n} w_i x_i \le M$. For such answer nodes, $c(x) = \sum_{i=1}^{n} p_i x_i$. All the remaining leaf nodes are obviously infeasible and for them $c(x) = \infty$. For all non-leaf nodes,

$$c(x) = \min[c(\text{Lchild}(x)), c(\text{Rchild}(x))]$$

For every node x, $\hat{c}(x) \le c(x) \le u(x)$.

- Let x be a node at level j for $1 \le j \le n+1$. For each node $1 \le i \le j$,

$$c(x) \le -\sum_{1 \le i \le j} p_i x_i \text{ and } u(x) = -\sum_{1 \le i \le j} p_i x_i. \text{ When new}$$

answer node is found and $a = -\sum_{1 \le i \le j} p_i x_i$, then we can update upper bound as

$$u(x) = \text{UpperBound}\left(a, \sum_{1 \le i \le j} w_i x_i, j-1, M\right)$$

- The algorithm UpperBound takes four input parameters: Current weight cw, current profit cp, current object k, maximum capacity of knapsack M. The output of the function is modified upper bound. An array w[1 ..., n] holds weights. An array p[1 ... n] holds profits.

Algorithm UpperBound(cw, cp, k, M)

```
begin
   np = cp;
   nw = cw;
   for i=k+1 to n
   begin
      if (nw+w[i]≤M)
        then
           nw = nw + w[i];
           np = np - p[i];
      end if
   end for
   return np
end
```


- Algorithm BBKnapsack takes three inputs: object k, current weight cw, current profit cp. Assume that $(p[i]/w[i]) \ge (p[i+1])$. Let tw is the final weight of knapsack and tp is the final maximum profit. The final solution will be available in an array x[1 ... n].
- If object k is in knapsack, x[k] = 1, otherwise x[k] = 0. Upper bound for a feasible left child of a node N is same as that for N. But for right child, upper bound should be generated. The path tx[ ] is the path from first node to the current node. Hence current weight $cw = \sum_{i=1}^{k-1} w_i tx_i$ and current profit $cp = \sum_{i=1}^{k-1} p_i tx_i$

The path of current node is stored in x[ ] if needed.

Algorithm BBKnapsack(k, cw, cp)

```
begin
   // Generation of left child
   if (cw+w[k]≤M)
   begin
      tx[k] = 1
      if (k<n)
         BBKnapsack (k + 1, cw+w[k], cp +p[k]);
      end if
   if ((cp+p[k]>tp)and(k=n))
   begin
      tp = cp + p[k]
      tw = cw + w[k]
      for j=1 to k
         x [j] = tx [j]
      end for
   end if
   end if
// Generation of right child
if (UpperBound(cw, cp, k, M)≥tp)
   begin
      tx[k] = 0
      if (k<n)
         BBKnapsack (k+1, cw, cp);
      end if
      if ((cp>tp)and(k=n))
      begin
         tp = cp
         tw = cw
         for j=1 to k
            x [j] = tx [j]
         end for
      end if
end
```

Initially the algorithm is called as BBKnapsack (1, 0, 0).

5.11.1 LC Branch-and-Bound

Let us solve one example:

Example 5.5: *Consider the knapsack instance n = 4, pro* $(p_1, p_2, p_3, p_4) = (10, 10, 12, 18)$, *weights* $(w_1, w_2, w_3, w_4) = (2$ *6, 9), maximum capacity M = 15. Solve the problem using branch-and-bound approach.*

Solution: LCBB will generate the following tree in Fig. 5.14.

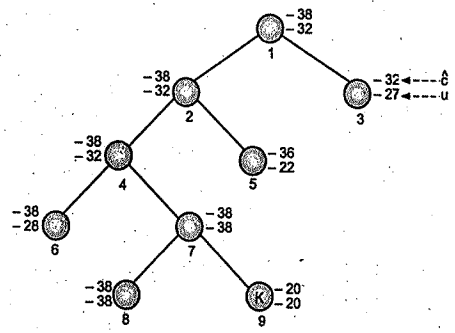


Fig. 5.14: LCBB Tree

For root node 1, $\hat{c}(1) = -38$, $u(1) = -32$

They are calculated as

$$u(1) = \text{UpperBound}(0, 0, 0, 15) = -10-10-12 = -32$$

$$\hat{c}(1) = -10-10-12-\frac{3}{9} \times 18 = -38$$

Node 1 is not a solution node, hence answer = 0 a upperbound (UB) = – 32. The E-node 1 is expanded whi generates children 2 and 3. $u(2) = -32$, $\hat{c}(2) = -38$, $u(3) = -$ $\hat{c}(3) = -32$. Add 2 and 3 to the list of live nodes. Next E-no is 2 which generates 4 and 5, add them to live nodes list. N E-node 4 generates 6 and 7 add them to live nodes list. No 7 changes UB to – 38, generates nodes 8 and 9, and ad them to live nodes list. Node 8 is a solution node. Node 9 h $\hat{c}(9) >$ UB and hence killed. Nodes 6 and 8 have least $\hat{c}$. F both, $\hat{c} \ge$ UB, hence the search terminates with node 8 as answer node. Optimal solution is $(x_4, x_3, x_2, x_1) = (1, 0, 1, 1)$. Though LCBB finds solution usi less number of nodes, observe that it has to maintain hea And insertions and deletions in heap are expensive.

Hence FIFOBB can be used which uses FIFO queue whi requires θ (1) time for insertions and deletions.

5.11.2 FIFO Branch-and-Bound

Example 5.6: *Let us solve the above example 5.5 using FIFOBB:*

Solution: The FIFOBB tree is shown below in Fig. 5.15.

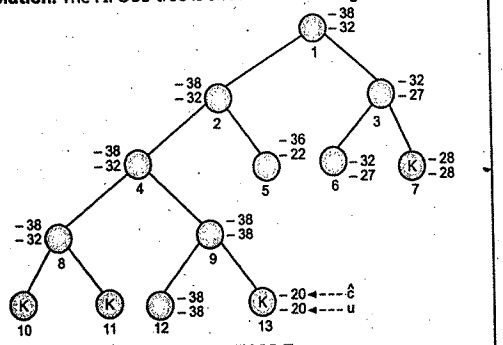


Fig. 5.15: FIFOBB Tree

- Initially queue is empty and root node 1 has u(1) = – 32, $\hat{c}(1)$ = – 38. Node 1 is not a solution node, it generates nodes 2 and 3, adds them to queue. UB remains same. Next node is 2 which generates nodes 4 and 5, and adds to queue. Then node 3 generates 6, 7 and adds 6 to queen. Node 7 is killed because $\hat{c}(7)$ > UB. Node 4 generates 8, 9 and adds to queue, changes u(9) = – 38, UB = – 38. Then nodes 5, 6 are considered. But their $\hat{c}$ > UB, hence not expanded. Node 8 generates nodes 10, 11, kills node 10 because it is infeasible, kills node 11 because $\hat{c}(11)$ > UB. Node 9 generates nodes 12, 13, adds 12 to queue, kills 13 because $\hat{c}(13)$ > UB, changes UB = – 38, answer = 12. Now there is only one node 12 in queue, but it has no children, hence the search terminates.
- We have solved the 0/1 knapsack problem using different approaches: greedy approach, dynamic programming, backtracking, LCBB, FIFOBB. Which approach is the best?
- We can decide it by writing programs, run those using different data, and obtain computing times. But still the results will vary due to effectiveness of bounding functions.

5.12 TRAVELLING SALESPERSON PROBLEM

The travelling salesperson problem is to find a minimum cost tour in a directed graph which visits each vertex in a graph exactly once.

Applications of Travelling Salesperson Problem:

- Consider a graph in which a vertex represents city and an edge represents distance between two cities. If a sales person wants to visit all the cities for selling items starting and ending from home city, then the route taken by a sales person is a tour and he is interested in finding a minimum length tour.
- A company wants to use a robot to pack a produced item in a box at each machine after every hour. There are many machines. If a machine denotes vertex of a graph, then time required to pack an item is same at each machine. But the time required to move from one machine to other is cost of the edge which connects those two machines. So the route taken by a robot is tour and company is interested in finding a tour which requires minimum time.
- Using dynamic programming approach, the problem can be solved in $O(n^2 2^n)$ time, but it requires $O(n2^n)$ space. But if the problem instance is considered while solving, then lesser time is required if proper bounding functions are selected.
- Let us solve the problem using branch-and-bound approach. Let G = (V, E) be a directed graph. Let c_{ij} denotes the cost of edge <i, j>. $c_{ij} = \infty$ if edge <i, j> $\notin$ E. Let the graph has n vertices, hence |V| = n.
- Assume that the tour of salesperson starts and ends at vertex 1. So the solution space S is given by path $(i_0, i_1, \ldots, i_{n-1}, i_n)$ where $i_0 = i_n = 1$. Obviously this is possible iff there is an edge $\langle i_k, i_{k+1} \rangle \in E(1 <= k < n)$ for each adjacent pair of vertices in path S. The solution space S can be represented as a state space tree.

Example 5.7: *Consider the following graph G in Fig. 5.16 (a). This is a graph with n = 4. For this graph, state space tree is as shown below in Fig. 5.16 (b).*

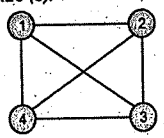


Fig. 5.16 (a) : Graph G

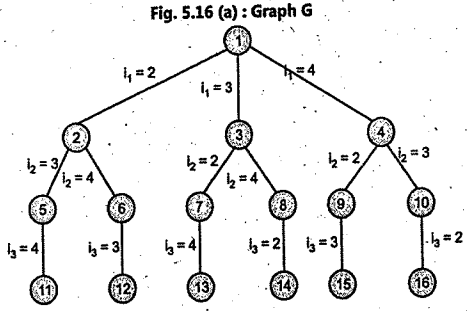


Fig. 5.16 (b) : State space tree for graph G

Here each leaf node is a solution node which defines a path from root to that node. For example, node 12 defines a tour $(i_0, i_1, i_2, i_3, i_4) = (1, 2, 4, 3, 1)$.

We have already seen FIFOBB and LCBB techniques. Let us use LCBB here. We can use either static space tree or dynamic space tree. Let us first solve the problem using static space tree by LCBB search.

5.12.1 LCBB Using Static State Space Tree

It requires c(.), $\hat{c}$(.) and u(.).

Definition of cost function c(.):

Let cost function c(.) is defined in the following way:

- If R is a leaf node, then c(R) is a length of path from root to node R.
- If R is a non-leaf node, then c(R) is a cost of minimum-cost leaf in the subtree of R.

Definition of $\hat{c}$(.) function:

$\hat{c}$(R) denotes the lower bound of node R. It can be obtained by using a **Reduced Cost Matrix.** A matrix is reduced iff every row and column is reduced. A row (column) is **Reduced** iff it contains at least one zero value and all the remaining entries in that row (column) are non-negative values.

We can represent graph G as a cost matrix M. To find $\hat{c}$( ), do the following:

1. Let k be the minimum value in row i.
2. Subtract k from each value in row i.
3. Repeat steps 1 and 2 for all rows and all columns until we obtain a reduced matrix.
4. The total of all k values used for subtraction, say L, is $\hat{c}$(.) for root node in state space tree.

Actually it tells the lowest value for all tours in graph G.

Let us solve one example:

Example 5.8: *Consider the following cost matrix in Fig. 5.17 (a) which defines an instance of traveling salesperson.*

∞	20	30	10	11
15	∞	16	4	2
3	5	∞	2	4
19	6	18	∞	3
16	4	7	16	∞

Fig. 5.17 (a) : Original cost matrix

∞	10	20	0	1
13	∞	14	2	0
1	3	∞	0	2
16	3	15	∞	0
12	0	3	12	∞

Fig. 5.17 (b) : Cost matrix with reduced rows

∞	10	17	0	1
12	∞	11	2	0
0	3	∞	0	2
15	3	12	∞	0
11	0	0	12	∞

Fig. 5.17 (c) : Cost matrix with reduced columns

Fig. 5.17 (b) shows cost matrix with reduced rows whi obtained by subtracting lowest value in each row from al elements in that row. If values 10, 2, 2, 3 and 4 are subtra from rows 1, 2, 3, 4, 5 respectively, we get matri Fig. 5.17 (b).

- It is not yet reduced matrix, so reduce columns. Sub values 1, 3 from columns 1 and 3 respectively, we reduced matrix of Fig. 5.17 (c).
- Total amount subtracted is 10 + 2 + 2 + 3 + 4 + 1 = 25. This is lower bound of root.

In this way, we have to find reduced cost matrix for each n in the state space tree. Let A is parent node of child B in tree and M is a reduced cost matrix of node A. Let A and B connected by an edge <x, y> from A to B. If B is a leaf n then $\hat{c}$(.)= c( ).

If B is a non-leaf node, then compute its reduced cost ma as follows:

- Make all elements in row x and column y as ∞, beca we don't want to consider edges <x, k> or <k, y>, tha remaining edges going from vertex x and remain edges coming to vertex y.
- Change M[y, 1] = ∞ to prevent use of edge <y, 1>.
- Except the rows and columns containing only ∞, all remaining rows and columns should be reduced. Let l the total amount subtracted to obtain reduced cost ma in this step.
- Compute $\hat{c}$(B) = $\hat{c}$(A) + M [x, y] + L

Computation of u( ):

For all the nodes A in state space tree, upper bound u(A) =

As we have defined c( ), $\hat{c}$( ) and u( ), we can apply LC algorithm now. Consider the same cost matrix again:

∞	20	30	10	11
15	∞	16	4	2
3	5	∞	2	4
19	6	18	∞	3
16	4	7	16	∞

(a) Initial cost matrix

∞	10	17	0	1
12	∞	11	2	0
0	3	∞	0	2
15	3	12	∞	0
11	0	0	12	∞

(b) Reduced cost matrix with L = 25

Fig. 5.18

This is reduced cost matrix of root node of state space t with L = 25, hence $\hat{c}$ ( ) = 25 and upper = ∞. Using this, can generate reduced cost-matrix of children of root no We obtain the following state space tree in Fig. 5.19 for t given travelling salesperson instance with n = 5.

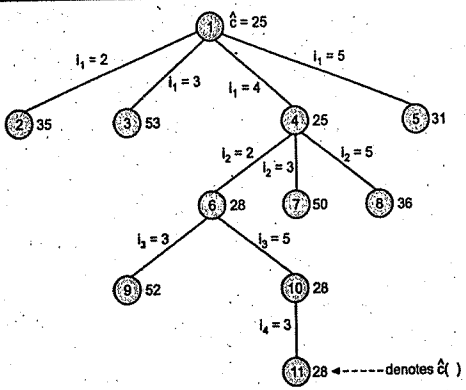


Fig. 5.19: State Space Tree

Reduced cost matrix should be computed for each node generated to find its $\hat{c}$ value. For all the non-leaf nodes except root, the reduced cost matrices are as shown below in Fig. 5.20.

∞	∞	∞	∞	∞
∞	∞	11	2	0
0	∞	∞	0	2
15	∞	12	∞	0
11	∞	0	12	∞

(a) Node 2, edge <1,2>

∞	∞	∞	∞	∞
1	∞	∞	2	0
∞	3	∞	0	2
4	3	∞	∞	0
0	0	∞	12	∞

(b) Node 3, edge <1,3>

∞	∞	∞	∞	∞
12	∞	11	∞	0
0	3	∞	∞	2
∞	3	12	∞	0
11	0	0	∞	∞

(c) Node 4, edge <1,4>

∞	∞	∞	∞	∞
10	∞	9	0	∞
0	3	∞	0	∞
12	0	9	∞	∞
∞	0	0	12	∞

(d) Node 5, edge <1,5>

∞	∞	∞	∞	∞
∞	∞	11	∞	0
0	∞	∞	∞	2
∞	∞	∞	∞	∞
11	∞	0	∞	∞

(e) Node 6, edge <4,2>

∞	∞	∞	∞	∞
1	∞	∞	∞	0
∞	1	∞	∞	0
∞	∞	∞	∞	∞
0	0	∞	∞	∞

(f) Node 7, edge <4,3>

∞	∞	∞	∞	∞
1	∞	0	∞	∞
0	3	∞	∞	∞
∞	∞	∞	∞	∞
∞	0	0	∞	∞

(g) Node 8, edge <4,5>

∞	∞	∞	∞	∞
∞	∞	∞	∞	∞
∞	∞	∞	∞	0
∞	∞	∞	∞	∞
0	∞	∞	∞	∞

(h) Node 9, edge <2,3>

∞	∞	∞	∞	∞
∞	∞	∞	∞	∞
0	∞	∞	∞	∞
∞	∞	∞	∞	∞
∞	∞	0	∞	∞

(i) Node 10, edge <2, 5>

∞	∞	∞	∞	∞
∞	∞	∞	∞	∞
∞	∞	∞	∞	∞
∞	∞	∞	∞	∞
∞	∞	∞	∞	∞

(j) Node 11, edge <5,3>

Fig. 5.20

Let us see how reduced cost matrix of node 4 is obtained. Node 1 and 4 denote edge <1, 4>.

1. First all the elements of row 1 and column 4 of node 1 matrix are set to ∞.
2. Set M[4, 1] = ∞. We obtain

∞	∞	∞	∞	∞
12	∞	11	∞	0
0	3	∞	∞	2
∞	3	12	∞	0
11	0	0	∞	∞

3. It is already a reduced matrix, hence L = 0.
4. Lower bound of node 4 is

$$\hat{c}(4) = \hat{c}(1) + M[1, 4] + L = 25 + 0 + 0 = 25$$

Note that M is a reduced cost matrix of parent node 1.

Similarly,

$$\hat{c}(11) = \hat{c}(10) + M[5, 3] + L = 28 + 0 + 0 = 28$$

In this way, we can find $\hat{c}$ values of all the nodes.

Root node 1 has 4 children 2, 3, 4, 5 having $\hat{c}$ values 35, 53, 25, 31 respectively. Since node 4 has lowest $\hat{c}$ value, it will be expanded further to have children 6, 7, 8. Again node 6 has lowest $\hat{c}$ value among nodes 2, 3, 6, 7, 8, 5. Hence node 6 will be expanded further to have children 9 and 10. Again node 10 has lowest $\hat{c}$ value, hence it is expanded to have child node 11. Among all leaf nodes node 11 has lowest $\hat{c}$ value = 28, hence upper is modified to 28. Tour length of node 11 is 28.

Now among the remaining children, node 5 has the lowest $\hat{c}$ value = 31, but 3 > upper. Hence LCBB search terminates here. The minimum cost tour is 1, 4, 2, 5, 3, 1.

5.12.2 LCBB Using Dynamic State Space Tree

- In this approach, a state space tree is generated which is dynamic binary tree, in which a left branch represents inclusion of an edge and a right branch represents exclusion of an edge. Depending on the problem to be solved, the order in which edges are considered differs.

  The method to compute $\hat{c}$ is same as in the previous section.

- If an edge <x, y> and right subtree represent all paths not including edge<x, y> at each node, that edge should be selected which has the highest probability to be in a minimum cost tour. One common approach for edge selection is to select that edge which will result in a right subtree having highest $\hat{c}$ value. We can achieve this by selecting an edge with reduced cost 0.

Example 5.9: *Consider again the same problem instance shown in Fig. 5.21 (a):*

∞	20	30	10	11
15	∞	16	4	2
3	5	∞	2	4
19	6	18	∞	3
16	4	7	16	∞

Fig. 5.21 (a) : Initial Cost Matrix

∞	10	17	0	1
12	∞	11	2	0
0	3	∞	0	2
15	3	12	∞	0
11	0	0	12	∞

Fig. 5.21 (b) : Reduced Cost Matrix with L = 25

Because total amount of subtraction L = 25, all tours in the graph will have minimum length 25. Hence $\hat{c}$ of root = 25. The dynamic state space tree is shown below in Fig. 5.22.

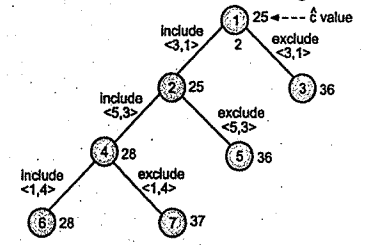


Fig. 5.22: Dynamic State Space Tree

At root node, we have to select an edge which results in a right child having highest $\hat{c}$ value. This can be done by selecting an edge with reduced cost 0. If the next edge is one of <1, 4>, <2, 5>, <3, 1>, <3, 4>, <4, 5>, <5, 2>, <5, 3>, then the resultant cost matrix B will have ∞ at that B[a, b] position for edge <a, b>. Matrix B should be converted to reduced cost matrix, which will increase reduced cost of right child.

- If an edge <1, 4> is included, then B[1, 4] = ∞, reduce row 1 by subtracting 1, column 4 remains reduced. Hence $\hat{c}$ of right child will increase by 1.
- If edge <3, 1> is included, then B[3, 1] = ∞, reduce column 1 by subtracting 11, row 3 remains reduced. Hence $\hat{c}$ of right child will increase by 11.
- In this way if the included edge is one of <1, 4>, <2, 5>, <3, 1>, <3, 4>, <4, 5>, <5, 2>, <5, 3>, then $\hat{c}$ of right child will increase by 1, 2, 11, 0, 3, 3, 11 respectively general, if M is the reduced matrix of parent A, inclusion of edge <a, b> where M[a, b] = 0 will increas value of right child by

$$\Delta = \min_{key}\{M(x, k)\} + \min_{key}\{M(k, y)\}$$

which should be subtracted from row x and column y reduce them.

- Here edges <3, 1> and <5, 3> both will increase $\hat{c}$ right child by 11. Let us select <3, 1>. Hence row 3 column 1 will have all ∞. B[1, 3] = ∞ to avoid cycle. resultant matrix B is shown in Fig. 5.23 (a) below. Like we can find reduced cost matrix when edge <3, 1> excluded as shown in Fig. 5.23 (b).

∞	10	∞	0	1
∞	∞	11	2	0
∞	∞	∞	∞	∞
∞	3	12	∞	0
∞	0	0	12	∞

Fig. 5.23 (a) : Node 2, Edge <3, 1> Included, L = 0

∞	10	17	0	1		∞	10	17	0	1
12	∞	11	2	0		1	∞	11	2	0
∞	3	∞	0	2	reduced	∞	3	∞	0	2
15	3	12	∞	0	matrix	4	3	12	∞	0
11	0	0	12	∞	=====>	0	0	0	12	∞

Fig. 5.23 (b) : Reduced Cost Matrix for Node3, Edge <3, 1> Excluded, L = 11

- Node 2 has the lowest $\hat{c}$ value among nodes 2 and hence node 2 will be expanded next. E-node = node For selection of next edge, consider following order edges having reduced cost 0 in Fig. 5.23 (a): edge <1, <2, 5>, <4, 5>, <5, 2>, <5, 3>. For these edges Δ = (1 = 3), 2, 3, 3, 11 respectively. So next edge will be <5, which has maximum Δ.
- Fig. 5.23 (c) is obtained when <5, 3> included, row 5 column 3 changed to ∞. M [3, 5] = ∞. Also M[1, 5] = ∞ <3, 1>, <5, 3> are already included and inclusion of ec <1, 5> will create cycle.]

∞	10	∞	0	∞		∞	7	∞	0	∞
∞	∞	∞	2	0		∞	∞	∞	2	0
∞	∞	∞	∞	∞	reduced	∞	∞	∞	∞	∞
∞	3	∞	∞	0	matrix	∞	0	∞	∞	0
∞	∞	∞	∞	∞	=====>	∞	∞	∞	∞	∞

Fig. 5.23 (c) : Reduced Cost Matrix for Node 4, Edge <5, 3> Included, L = 3, $\hat{c}$ (4) = 25 + 3 = 28

∞	10	∞	0	1		∞	10	∞	0	1
∞	∞	11	2	0		∞	∞	0	2	0
∞	∞	∞	∞	∞	reduced	∞	∞	∞	∞	∞
∞	3	12	∞	0	matrix	∞	3	1	∞	0
∞	0	∞	12	∞	=====>	∞	0	∞	12	∞

M[5, 3] = ∞

Fig. 5.23 (d) : Reduced Cost Matrix for Node 5, Edge <5. 3>

Excluded, L = 11, $\hat{c}$ (5) = 25 + 11 = 36

- Again node 4 has lowest $\hat{c}$ value, hence it will be expanded next. E-node = node 4. Consider the edges in the following order from Fig. 5.23 (c): edge <1, 4>, <2, 5>, <4, 2>, <4, 4>. They give Δ = (7 + 2 = 9), 2, 7, 0 respectively. As edge <1, 4> has maximum Δ, it will be selected next.
- Fig. 5.23 (e) shows reduced cost matrix when edge <1, 4> is included. M[4, 1] = ∞, row 1 and column 4 set to ∞. So now path includes <3, 1>, <5, 3>, <1, 4>. Also M[4, 5] = ∞, as inclusion of edge <4, 5> will form cycle.

∞	∞	∞	∞	∞
∞	∞	∞	∞	0
∞	∞	∞	∞	∞
∞	0	∞	∞	∞
∞	∞	∞	∞	∞

Fig. 5.23 (e) : Reduced Cost Matrix for Node 6, Edge <1, 4>

Included, L = 0, $\hat{c}$ (6) = 28 + 0 = 28

∞	7	∞	∞	∞		∞	0	∞	∞	∞
∞	∞	∞	2	0		∞	∞	∞	0	0
∞	∞	∞	∞	∞	reduced	∞	∞	∞	∞	∞
∞	0	∞	∞	0	matrix	∞	0	∞	∞	∞
∞	∞	∞	∞	∞	=====>	∞	∞	∞	∞	∞

Fig. 5.23 (f) : Reduced Cost Matrix for Node 7, Edge <1, 4>

Excluded, L = 7 + 2 = 9, $\hat{c}$ (7) = 28 + 9 = 37

- Node 6 has lowest $\hat{c}$ value. Hence it will be expanded next. E-node = Node 6. Consider edges in the following order from Fig. 5.23 (e) <2, 5>, <4, 2>. They give Δ = 0, 0 respectively. So both edges can be selected directly. Hence edges in path are <3, 1>, <5, 3>, <1, 4>, <4, 2>, <2, 5>, which give path 1, 4, 2, 5, 3, 1 with length 28. Hence upper is modified to 28. In the remaining nodes, node 3 and node 5 have smaller $\hat{c}$ value = 36. Let node 3 becomes next E-node, whose $\hat{c}$ value = 36 > upper. Hence LCBB search terminates here.
- We can conclude that the LCBB search using dynamic state space tree works better for large subtrees. When a small subtree is reached, then it is evaluated without using the bounding functions.

Let us Solve Few Examples :

Example 5.10: *Consider the following cost matrix for a traveling salesperson instance. Compute reduced cost matrix and draw the dynamic state space tree obtained using LCBB. Also write the reduced cost matrices of all the nodes.*

∞	7	3	12	8
3	∞	6	14	9
5	8	∞	6	18
9	3	5	∞	11
18	14	9	8	∞

Solution: Subtract 3, 3, 5, 3, 8 from the row 1, 2, 3, 4, 5 respectively. We get first matrix in Fig. 5.24 (a).

∞	4	0	9	5		∞	4	0	9	0
0	∞	3	11	6		0	∞	3	11	1
0	3	∞	1	13	Reduced	0	3	∞	1	8
6	0	2	∞	8	column 5	6	0	2	∞	3
10	6	1	0	∞	=====>	10	6	1	0	∞

Fig. 5.24 (a) : Reduced Cost Matrix of Root Node 1,

$\hat{c}$ (1) = 3 + 3 + 5 + 3 + 8 + 5 = 27

- Initially upper = ∞.
- Consider edges with reduced cost = 0 in the following order: <1, 3>, <1, 5>, <2, 1>, <3,1>, <4,2>, <5, 4>. They give Δ = 1, 1, 1, 1, (2 + 3 = 5), (1 + 1 = 2) respectively. Hence edge <4, 2> giving maximum Δ is selected next. Reduced cost matrix when <4, 2> is included is shown in Fig. 5.24 (b). It has M[4, 2] = ∞, row 4 and column 2 is set to ∞. Also M[2, 4] = ∞ to avoid cycle.

∞	∞	0	9	0
0	∞	3	∞	1
0	∞	∞	1	8
∞	∞	∞	∞	∞
10	∞	1	0	∞

Fig. 5.24 (b) : Node 2, Edge <4, 2> Included, L = 0,

$\hat{c}$ (2) = 27 + 0 = 27

The dynamic state space tree is as shown below in Fig. 5.24 (c).

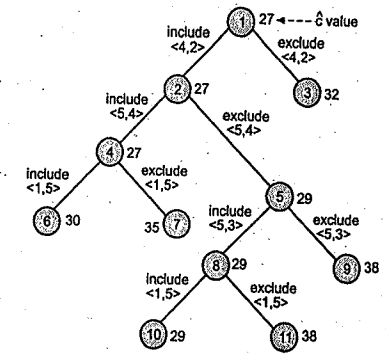


Fig. 5.24 (c): Dynamic State Space Tree

∞	4	0	9	0		∞	1	0	9	0
0	∞	3	11	1	Reduced	0	∞	3	11	1
0	3	∞	1	8	row 4 and	0	0	∞	1	8
6	∞	2	∞	3	column 2	4	∞	0	∞	1
10	6	1	0	∞	=====>	10	3	1	0	∞

Fig. 5.24 (d) : Node 3, Edge <4, 2> Excluded. L = 2+3 = 5,

$\hat{c}$ (3) = 27 + 5 = 32

- As node 2 has lowest $\hat{c}$ value, next E-node = node 2. Consider the edges from Fig. 5.24 (b) in the order: <1, 3>, <1, 5>, <2, 1>, <3, 1>, <5, 4>. They give Δ = 1, 1, 1, 1, (1 + 1 = 2) respectively. As edge <5, 4> has maximum Δ, it is selected next. Reduced cost matrix when edge <5, 4> is selected is as shown in Fig. 5.24 (e). It has M[4, 5]= ∞, row 5 and column 4 set to ∞. So now path includes <4, 2>, <5, 4> edges. As inclusion of edge <2, 5> creates cycle, set M[2, 5] = ∞.

∞	∞	0	∞	0
0	∞	3	∞	∞
0	∞	∞	∞	8
∞	∞	∞	∞	∞
∞	∞	∞	∞	∞

Fig. 5.24 (e) : Node 4, Edge <5, 4> Included,

L = 0, $\hat{c}$ (4) = 27 + 0 = 27

∞	∞	0	9	0		∞	∞	0	8	0
0	∞	3	∞	1	Reduced	0	∞	3	∞	1
0	∞	∞	1	8	row 5	0	∞	∞	0	8
∞	∞	∞	∞	∞	column 4	∞	∞	∞	∞	∞
10	∞	1	∞	∞	=====>	9	∞	0	∞	∞

M [5, 4] = ∞

Fig. 5.24 (f) : Node 5, Edge <5, 4> Excluded, L = 2,

$\hat{c}$ (5) = 27 + 2 = 29

- As node 4 has lowest $\hat{c}$ value, it is next expanded. E-r = node 4. From Fig. 5.24 (e), consider edges with redu cost 0 in the following order: <1. 3>, <1, 5>, <2, 1>, 1>. They give Δ = 3, 8, 3, 8 respectively. Edges <1, 5> <3, 1> give maximum Δ.
- Let us select <1, 5> as next edge. Hence reduced matrix when edge <1, 5> is included, is as shown Fig. 5.24 (g). Set M[5, 1] = ∞. Set row 1 and colu 5 to ∞. So now path includes <4, 2>, <5, 4>, <1, 5> inclusion of an edge <2, 1> creates cycle. Set M[2, 1] =

∞	∞	∞	∞	∞		∞	∞	∞	∞	∞
∞	∞	3	∞	∞		∞	∞	0	∞	∞
0	∞	∞	∞	∞	Reduced	0	∞	∞	∞	∞
∞	∞	∞	∞	∞	matrix	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	=====>	∞	∞	∞	∞	∞

Fig. 5.24 (g) : Node 6, Edge <1, 5> Included, L = 3,

$\hat{c}$ (6) = 27 + 3 = 30

∞	∞	0	∞	∞		∞	∞	0	∞	∞
0	∞	3	∞	∞		0	∞	3	∞	∞
0	∞	∞	∞	8	Reduced	0	∞	∞	∞	0
∞	∞	∞	∞	∞	column 5	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	=====>	∞	∞	∞	∞	∞

M[1, 5] = ∞

Fig. 5.24 (h) : Node 7, Edge <1, 5> Excluded, L = 8,

$\hat{c}$ (7) = 27 + 8 = 35

- As node 5 has lowest $\hat{c}$ value, it is expanded next. E-no = node 5. form Fig. 5.24 (f), consider edges with reduc cost 0 in the order: <1, 3>, <1, 5>, <2, 1>, <3, 1>, <3, 4 <5, 3>. They give Δ = 0, 1, 1, 0, 8, 9 respectively. As ed <5, 3> gives maximum Δ, it is selected next.
- Reduced cost matrix when edge <5, 3> is included, is shown in Fig. 5.24 (i) below. Set M[3, 5] = ∞. Set row and column 3 to ∞. Now path includes <4, 2>, <5, 3>.

∞	∞	∞	8	0
0	∞	∞	∞	1
0	∞	∞	0	∞
∞	∞	∞	∞	∞
∞	∞	∞	∞	∞

Fig. 5.24 (i) : Node 8, Edge <5, 3> Excluded, L = 8,

$\hat{c}$(8) = 29 + 0 = 29

∞	∞	0	8	0
0	∞	3	∞	1
0	∞	∞	0	8
∞	∞	∞	∞	∞
9	∞	∞	∞	∞

Reduced row 5 =====>

∞	∞	0	8	0
0	∞	3	∞	1
0	∞	∞	0	8
∞	∞	∞	∞	∞
0	∞	∞	∞	∞

M [5, 3] = ∞

Fig. 5.24 (j) : Node 9, Edge <5, 3> Excluded, L = 9, $\hat{c}(9) = 29 + 9 = 38$

- Among nodes 6, 7, 8, 9, 3 only node 8 has lowest $\hat{c}$ value, hence E-node = node 8. From Fig. 5.24 (i), consider edges with reduced cost 0 in the order: <1, 5>, <2, 1>, <3, 1>, <3, 4>. They give Δ = (8 + 1 = 9), 1, 0, 8 respectively. As edge <1, 5> gives maximum Δ. It is selected next.
- Reduced cost matrix when edge <1, 5> included is shown in Fig. 5.24 (k). Set row 1 and column 5 to ∞. Also set M [5, 1] = ∞ to avoid cycle. Now path contains edges <4, 2>, <5, 3>, <1, 5>. As inclusion of edge <3, 1> creates cycle {<5,3>, <3, 1>, <1, 5>}, set M[3, 1] = ∞.

∞	∞	∞	∞	∞
0	∞	∞	∞	∞
∞	∞	∞	0	∞
∞	∞	∞	∞	∞
∞	∞	∞	∞	∞

Fig. 5.24 (k) : Node 10, Edge <1, 5> Included, L = 0, $\hat{c}(10) = 27 + 0 = 29$

∞	∞	∞	8	∞
0	∞	∞	∞	1
0	∞	∞	0	8
∞	∞	∞	∞	∞
∞	∞	∞	∞	∞

Reduced row 1 and column 5 =====>

∞	∞	∞	0	∞
0	∞	∞	∞	0
0	∞	∞	0	∞
∞	∞	∞	∞	∞
∞	∞	∞	∞	∞

M[1, 5] = ∞

Fig. 5.24 (l) : Node 11, Edge <1, 5> Included, L = 8 +1 = 9, $\hat{c}(4) = 29 + 9 = 38$

- Among nodes 6, 7, 3, 10, 11, 9 node 10 has lowest $\hat{c}$ value, hence next E-node = node 10. From the reduce cost matrix of node 10 in Fig. 5.24 (k), if is clear that only 2 edges are remained: <2, 1> and <3, 4>. They can be selected directly.
- Now path contains edges <4, 2>, <5, 3>, <1, 5>, <2, 1>, <3, 4> which give tour of 1, 5, 3, 4, 2, 1 of length 29.

Hence upper is modified to 29. In the remaining nodes, node 6 has lowest $\hat{c}$ value. But $\hat{c}(6) = 30 >$ upper. Hence LCBB terminates here. The minimum-cost tour is 1, 5, 3, 4, 2, 1 of length 29.

Example 5.11: *Reduce the following cost matrix which gives a traveling salesperson instance. Using LCBB, generate dynamic state space tree, find reduced cost matrices for all the nodes in the tree, find the minimum-cost tour.*

∞	11	10	9	6
8	∞	7	3	4
8	4	∞	4	8
11	10	5	∞	5
6	9	5	5	∞

Solution: Subtract 6, 3, 4, 5, 5 from rows 1, 2, 3, 4, 5 respectively. We get first matrix in Fig. 5.25.

∞	5	4	3	0
5	∞	4	0	1
4	0	∞	0	4
6	5	0	∞	0
1	4	0	0	∞

Reduced column 1 =====>

∞	5	4	3	0
4	∞	4	0	1
3	0	∞	0	4
5	5	0	∞	0
0	4	0	0	∞

Fig. 5.25 : Reduced Cost Matrix of Root Node 1, $\hat{c}(1) = 6 + 3 + 4 + 5 + 5 + 1 = 24$

- From second matrix of Fig. 5.25 consider edges with reduced cost 0 in the order: <1, 5>, <2, 4>, <3, 2>, <3, 4>, <4, 3>, <4, 5>, <5, 1>, <5, 3>, <5, 4>. They give Δ = 3, 1, 4, 0, 0, 0, 3, 0, 0 respectively. Edge <3, 2> gives maximum Δ, hence it is selected next. The dynamic state space tree for this example is as shown below in Fig. 5.26.

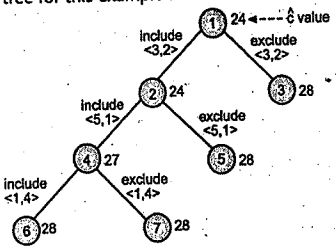


Fig. 5.26 : Dynamic State Space Tree

- Reduced cost matrix for node 2 when edge <3, 2> included is shown in Fig. 5.27 (a). Set row 3 and column 2 to ∞. Set M[2, 3] = ∞ to avoid creation of cycle.

∞	∞	4	3	0
4	∞	∞	0	1
∞	∞	∞	∞	∞
5	∞	0	∞	0
0	∞	0	0	∞

Fig. 5.27 (a) : Node 2, Edge <3, 2> Included, L = 0, $\hat{c}(2) = 24 + 0 = 24$

∞	5	4	3	0
4	∞	4	0	1
3	∞	∞	0	4
5	5	0	∞	0
0	4	0	0	∞

Reduced column 2 =====>

∞	1	4	3	0
4	∞	4	0	1
3	∞	∞	0	4
5	1	0	∞	0
0	0	0	0	∞

M [3, 2] = ∞

Fig. 5.27 (b) : Node 3, Edge <3, 2> Included, L = 4, $\hat{c}(3) = 24 + 4 = 28$

- As node 2 has lowest $\hat{c}$ value, it is expanded next. E-node = node 2. From Fig. 5.27 (a), consider edges with reduced cost 0 in the order: <1, 5>, <2, 4>, <4, 3>, <4, 5>, <5, 1>, <5, 3>, <5, 4>. They give Δ = 3, 1, 0, 0, 4, 0, 0 respectively.
- As edge <5, 1> fives maximum Δ, it is selected next. Reduced cost matrix when edge <5, 1> included is as shown in Fig. 5.27 (c), row 5 and column 1 is set to ∞. Also M[1, 5] = ∞ to avoid cycle, now path includes edges <3, 2>, <5, 1>.

∞	∞	4	3	∞
∞	∞	∞	0	1
∞	∞	∞	∞	∞
∞	∞	0	∞	0
∞	∞	∞	∞	∞

Reduced row 1 =====>

∞	∞	1	0	∞
∞	∞	∞	0	1
∞	∞	∞	∞	∞
∞	∞	0	∞	0
∞	∞	∞	∞	∞

Fig. 5.27 (c) : Node 4, Edge <5, 1> Included, L = 3, $\hat{c}(4) = 24 + 3 = 27$

∞	∞	4	3	0
4	∞	∞	0	1
∞	∞	∞	∞	∞
5	∞	0	∞	0
∞	∞	0	0	∞

Reduced column 1 =====>

∞	∞	4	3	0
0	∞	∞	0	1
∞	∞	∞	∞	∞
1	∞	0	∞	0
∞	∞	0	0	∞

Fig. 5.27 (d) : Node 5, Edge <5, 1> Excluded, L = 4, $\hat{c}(5) = 24 + 4 = 28$

- As node 4 has lowest value among nodes 4, 5, 3, E-node = node 4. From Fig. 5.27 (c) reduced matrix, consider edges with reduced cost 0 in the order: <1, 4>, <2, 4>, <4, 3>, <4, 5>. They give Δ = 1, 0, 1, 0 respectively. As edges <1, 4> and <4, 3> both give maximum Δ, let us select <1, 4> as next edge.
- Reduced cost matrix when edge <1, 4> included is shown in Fig. 5.27 (e). Row 1 and column 4 is set to ∞. Also set M[4, 1] = ∞ to avoid cycle. Now path contains edges <3, 2>, <5, 1>, <1, 4>. As inclusion of edge <4, 5> will create cycle {<1, 4>, <4, 5>, <5, 1>}, set M[4, 5] = ∞

∞	∞	∞	∞	∞
∞	∞	∞	∞	1
∞	∞	∞	∞	∞
∞	∞	0	∞	∞
∞	∞	∞	∞	∞

Reduced row 2 =====>

∞	∞	∞	∞	∞
∞	∞	∞	∞	0
∞	∞	∞	∞	∞
∞	∞	0	∞	∞
∞	∞	∞	∞	∞

Fig. 5.27 (e) : Node 6, Edge <1, 4> Included, L = 1, $\hat{c}(6) = 27 + 1 = 28$

∞	∞	1	∞	∞
∞	∞	∞	0	1
∞	∞	∞	∞	∞
∞	∞	0	∞	0
∞	∞	∞	∞	∞

Reduced row 1 =====>

∞	∞	0	∞	∞
∞	∞	∞	0	1
∞	∞	∞	∞	∞
∞	∞	0	∞	0
∞	∞	∞	∞	∞

M[1, 4] = ∞

Fig. 5.27 (f) : Node 7, Edge <1, 4> Excluded, L = 1, $\hat{c}(7) = 27 + 1 = 28$

- As all the nodes 6, 7, 5, 3 have same $\hat{c}$ value, let next node = node 6. Now only two edges are remained: <2, 1> and <4, 3>. They can be selected directly. Now path contains edges <3, 2>, <5, 1>, <1, 4>, <2, 5>, <4, 3>. Hence upper is modified to 28. Among the remaining nodes, all have same $\hat{c}$ value, which is equal to upper. Hence LCBB terminates here. Minimum-cost tour is 1, 4, 2, 5, 1 of length 28.

5.13 15-PUZZLE PROBLEM

The 15 puzzle problem consists of 15 numbered titles on a 4 × 4 square frame with a capacity of 16 tiles. Initial arrangement of the titles is shown in Fig. 5.28 (a). We have to convert it into the goal arrangement shown on Fig. 5.28 (b).

1	2	4	15
3		5	12
7	8	11	14
6	9	10	13

Fig. 5.28 (a) : Initial arrangement

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

Fig. 5.28 (b) : Goal arrangement

To convert initial arrangement into a goal arrangement, we have to follow certain legal moves. A legal move is one in which a tile adjacent to the Empty Spot (ES) is moved to ES. Each legal move is followed by other legal moves. Each move creates new arrangement of the tiles. These arrangements are called the states of the puzzle. The initial and goal arrangements are called initial and goal states.

A state is reachable from the initial state iff there is a sequence of legal moves from the initial state to this state. There are total 16 ! different arrangements of the tiles.

Let us number the frame positions from 1 to 16. Let position (i) be the position number in the initial state of the tile numbered i. For example, in above Fig. 5.28 (a), position (16) denotes the position of 13. For any state, let Less (i) denotes the number of tiles j < i and position (j) > position (i). For example, in Fig. 5.28 (a),

Less (1) = 0

Less (3) = 1, because in the range (1, 2), position (2) > position (3) only.

Less (2) = 0

Less (4) = 1, because in the range (1, 3) only position (2) > position (4).

Less (5) = less (6) = 0

Less (7) = 1

Less (8) = Less (9) = Less (10) = 0

Less (11) = 3 because in the range (1, 10) positions of 8, 9, 10 are > positions (11).

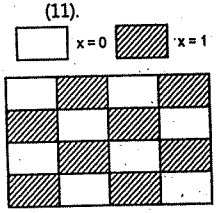


Fig. 5.28 (c)

In the initial state, if empty spot is on the shaded position of Fig. 5.28 (c), then x = 1, other wise x = 0. The following theorem can be used.

Theorem : The goal state of Fig. 5.28 (b) is reachable from the initial state iff $\sum_{i=1}^{16}$ Less (i) + x is even.

To check whether the goal state is in the state space of the initial state, above theorem can be used.

- For an intelligent reach to reach the goal position, we use a cost functions :

$$\hat{c} = f(x) + \hat{g}(x)$$

where f(x) is the length if the path from the root to node x and $\hat{g}(x)$ is an estimate of the length of a shortest path from x to a goal node in the subtree of root x. Let $\hat{g}(x)$ = number of non-blank tiles not in their goal position. $\hat{c}(x)$ is a lower bound on the value of c(x).

The portion of state space tree is as shown below.

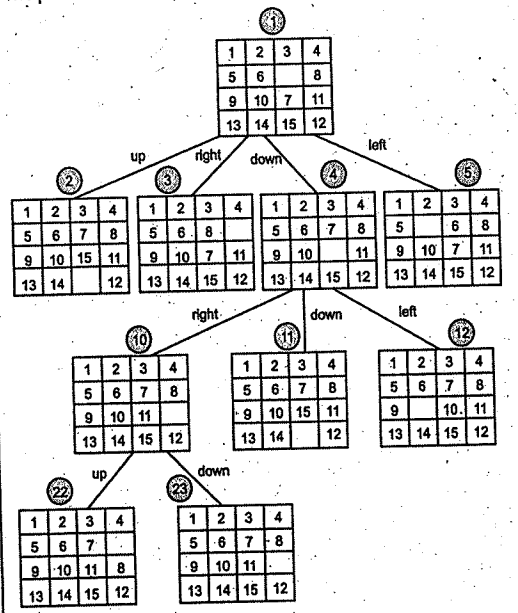


Fig. 5.29

Draw the tree using the following steps :

- Initially node 1 is the only live node. Using the 4 possible legal moves, we get nodes 2, 3, 4, 5.

  For node 2, $\hat{c}(2) = 1 + 4 = 5$

  For node 3, $\hat{c}(3) = 1 + 4 = 5$

  For node 4, $\hat{c}(4) = 1 + 2 = 3$

  For node 5, $\hat{c}(5) = 1 + 4 = 5$

- Among nodes 2, 3, 4, 5 value of $\hat{c}(4)$ is minimum. Hence node 4 is expanded next. E-node = node 4.

- From node 4 on legal moves, we get nodes 10, 11, 12.

  $\hat{c}(10) = 2 + 1 = 3$

  $\hat{c}(11) = 2 + 3 = 5$

  $\hat{c}(12) = 2 + 3 = 5$

- As node 10 has lowest $\hat{c}(10) = 3$, it is expanded next. E-node = node 10.

- Node 23 is the goal state. In this way, we can solved 15-puzzle problem using LC search.

5.14 COMPARISONS BETWEEN BACKTRACKING AND BRANCH AND BOUND

Generally branch and bound algorithms are used to solve optimization problems. As the algorithm progresses, a tree of sub-problems is formed. The original problem forms a root. It calculates an upper and lower bounds for each problem. At each node

- If the bounds match, it is considered a feasible solution to that particular sub-problem.
- If the bounds do not match, partition the sub-problem of that node into two sub-problems represented by two children of that node.

If a node has solution better than the best feasible solution, it is remembered as the best feasible solution, else that section of the tree is trimmed.

The Branch and Bound Strategy Uses Two Tools:

1. **Branching:** To create tree for problem and child for sub-problem
2. **Bounding:** To find upper and lower bounds

Generally branch and bound algorithms are used to solve optimization problems whereas backtracking strategy is effective to solve decision problems. It is not designed for optimization problems, where we also have a cost function f(x) for minimization or maximization.

SOLVED EXERCISE

I. Long Question Answer:

1. Explain search strategies in branch and bound.

Ans.: There are four search strategies in branch and bound.

(1) LIFO

(2) FIFO

(3) LC

(4) Heuristic

(1) LIFO search is also termed as DFS (Depth First search). It explores live nodes in the reverse order of their creation. It uses stack data structure.

(2) FIFO search is also termed as BFS (Breadth First Search). It explores live nodes in the order of their creation. It uses queue data structure.

(3) DFS and BFS are special cases of LC search. LC search explores that node immediately which has a very good chance of getting the search to a solution node quickly. It uses priority queue as data structure.

(4) Heuristic search.

2. Explain heuristic search in brief.

Ans.: Heuristic search consists of the following characterist It helps to designs an algorithm with provably good run tin and provably good or optimal solution quality. But there is proof that it will always satisfy both these characteristics.

(a) Use of heuristic search is very common in real wo implementations.

(b) A heuristic search might not find the best solution alwa but it is guaranteed to find a good solution in reasonal time.

(c) It is useful to solve problems which require an infin time for solving by other methods.

(d) A classic example of heuristic search is travelli salesperson problem.

- The heuristic search performs the following steps:

  (a) It generates a possible solution which can either be point in the problem state or a path from the init state.

  (b) It tests to see if this possible solution is a real soluti by comparing the state reached with set of gc states

  (c) If the possible solution is a real solution, them retur otherwise repeat step (a) again.

3. Write control abstraction for LC search.

Ans.: It uses the following structure:

```
typedef struct ListNode
{
    ListNode *next, *parent;
    float cost;
} *T, *x, *E-node;
```

Algorithm LCSearch(t)

```
begin
    // Search t for an answer node.
    if *t is an answer node then
    begin
        output *t
        return
    endif
    E-node = t // Set E-node.
    //Initialize list of live nodes to empty
    LiveNodes = 0
    repeat
```

```
{
    for (each child x of E-node)
    begin
        if x is an answer node then
        begin
            output the path (x, t)
            return
        endif
        //Add x to list of live nodes
        LiveNodes = LiveNodes ∪ x
        // Set E-node as parent of node x
        x → parent = E-node
    end for

    if (there are no more live nodes) then
    begin
        print "No answer node"
        return
    endif
    E-node = Least ( )
}
until (false);
end
```

Function Least( ) searches LiveNodes list and returns a node having least cost $\hat{c}$.

MULTIPLE CHOICE QUESTIONS (MCQ's)

1. In Backtracking method, backtrack to previous move is possible with the help of
   (a) Queue (b) Stack
   (c) Dqueue (d) All of the above
2. State space is used in:
   (a) Back track (b) Greedy
   (c) Dynamic (d) None of the above
3. Which of the following is not a backtracking algorithm?
   (a) Knight tour problem
   (b) N queen problem
   (c) Tower of hanoi
   (d) M coloring problem
4. Match the following with respect to algorithm paradigms:

List-I	List-II
(a) Merge sort	i. Dynamic Programming
(b) Huffman coding	ii. Greedy approach
(c) Optimal polygon triangulation	iii. Divide and conquer
(d) Subset sum problem	iv. Back tracking

   Codes:
   a b c d
   (a) iii i ii iv (b) ii i iv iii
   (c) ii i iii iv (d) iii ii i iv
5. From the following which is not return optimal solution
   (a) Back track (b) Branch and Bound
   (c) Dynamic (d) Greedy
6. The term ____ refers to all state space search methods in which all children of the nodes are generated before any other live node can become the E-node.
   (a) Back track (b) Branch and Bound
   (c) Depth First Search (d) Breadth First Search
7. Graph coloring is the example of ____ algorithmic strategy.
   (a) Back track (b) Branch and Bound
   (c) Dynamic (d) Greedy
8. The method which stops the execution, if it finds the solution. Otherwise it starts from the top.
   (a) Back track (b) Branch and Bound
   (c) Dynamic (d) Greedy
9. Name the node which has been generated but none of its children nodes have been generated in state space tree of back tracking method.
   (a) Dead node (b) Live node
   (c) E-node (d) State node

ANSWERS

1. (b)	2. (a)	3. (c)	4. (d)	5. (a)
6. (b)	7. (a)	8. (a)	9. (b)	

EXERCISE

1. Write a short note on Branch and bound approach.
2. Explain in brief:
   (a) FIFO Search
   (b) LIFO Search
   (c) LC Search
3. List and explain various search strategies of branch and bound approach.
4. Write control abstraction for LC search.
5. Explain the term: Bounding.
6. Define reduced cost matrix. Explain how we can obtain reduced cost matrix.
7. Define minimum cost tour.
8. Consider the travelling sales person instance defined by the cost matrix:

$$\begin{bmatrix} \infty & 20 & 30 & 10 & 11 \\ 15 & \infty & 16 & 4 & 2 \\ 3 & 5 & \infty & 2 & 4 \\ 19 & 6 & 18 & \infty & 3 \\ 16 & 4 & 7 & 16 & \infty \end{bmatrix}$$

   (a) Obtain the reduced cost matrix.
   (b) Using the space tree formulation, obtain the tree that will be generated by LCBB. Label each node by its value. Write out the reduced matrices corresponding to each of nodes.
   (c) What is path and minimum cost of tour for travelling sales person in above problem?
9. Devise LC branch and bound algorithm for travelling salesman problem.
10. (a) Describe following w.r.t Branch and Bound:
    (i) The method.
    (ii) Least-Cost search (LC-Search).
    (iii) Control-abstraction for LC-search.
    (iv) Bounding.
    (b) Write LCBB algorithm for knapsack problem using the fixed tuple size formulation and the state space tree.
11. Develop an algorithm based on the FIFO approach to solve the knapsack problem.
12. Develop the LCBB algorithm for the 0/1 Knapsack problem using a fixed tuple size formulation. State the assumptions made. The algorithm should include procedures to compute lower and upper bounds, creating a new node and output the answer.
13. Describe in brief "Branch and Bound strategy".
14. Consider the travelling salesman instance defined by the following cost matrix.

$$\begin{bmatrix} \infty & 7 & 3 & 12 & 8 \\ 3 & \infty & 6 & 14 & 9 \\ 5 & 8 & \infty & 6 & 18 \\ 9 & 3 & 5 & \infty & 11 \\ 18 & 14 & 9 & 8 & \infty \end{bmatrix}$$

    (a) Obtain reduced cost matrix.
    (b) Obtain the portion of the state space tree by LCBB. Label each node by its $\hat{c}$ value.
    (c) Solve the above to compute the shortest tour?
    (d) What do you mean by heuristic search?
15. Explain how branch and bound can be used to solve Knapsack problem?
16. State TRUE or FALSE.
    (i) Bounding functions are used to help avoid the generation of sub-trees that do not contain an answer node.
    (ii) In branch and bound terminology, Breadth First Search is called First In First Out.
    (iii) The worst case complexity of Travelling sales person problem using branch and bound is $\Theta(n^2 2^n)$.
    (iv) The worst case complexity of travelling sales person problem using branch and bound method is same as that using dynamic programming.
17. Explain in brief: Branch and Bound method.
18. Write a short note on branch and bound method.
19. Explain how Branch and Bound method can be used to solve Least Cost search.
20. Write an upper bound function for 0/1 Knapsack problem.
21. Explain backtracking strategy.

22. Write a short note on: State space tree.
23. List the problems which can be solved by using backtracking.
24. Let w = {6, 8, 11, 13, 16, 19, 21} and M = 36. Find all possible subsets having sum = M. Draw state space tree generated.
25. Discuss in brief 8-queens problem.
26. Write backtracking algorithm (non-recursive) for the following problem and also mention bounding function if any:
    (i) 8 queen's problem,
    (ii) Graph coloring problem.
27. Solve the sum of subset problem using backtracking algorithmic strategy for the following data: N=4, (w1, w2, w3, w4) = (11, 13, 24, 7) and M = 31

✠ ✠ ✠

Unit VI

TRI

6.1 INTRODUCTION

- The main purpose of trees is fast information retrieval and update.
- In database programs where most information is stored on disks or tapes, the time penalty for accessing secondary storage can be reduced by the proper choice of data structures. B-trees (Bayer) are one such approach. A B-tree operates closely with secondary storage and can be tuned to reduce the impediments imposed by this storage.
- One important property of B-tree is the size of each node which can be made as large as the size of the block of the disk. The number of keys in one node can vary depending on the sizes of the keys organization of data and size of the block.
- Block size varies for each system. It can be 512 bytes, 4 KB or more block size is the size of each node of a B-tree.

6.2 B-TREE

Multiway Search Trees :

- Multiway search tree is a general search tree in which each node contains one or more keys (i.e. data fields).
- A multiway search trees of degree n may be defined as a generated search tree in which each node has at the most n subtrees and contains number of keys one less than the no. of subtree it has.

Fig. 6.1 (a) is multiway search tree of degree 4 (i.e. node has atmost 4 subtrees).

Full Nodes : The nodes which have maximum no. of keys 3 and maximum no. of subtrees 4 are called as full nodes.

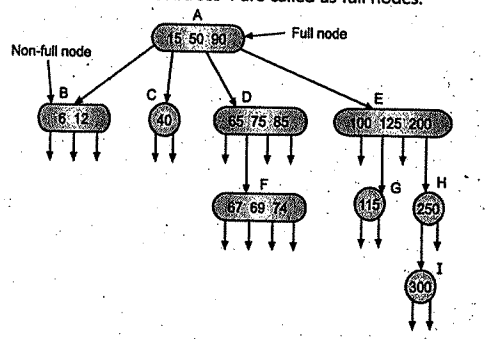


Fig. 6.1 : Multiway search tree

Full nodes are A, D, E, F.

Semileaf : A semileaf is defined as a node with atleast empty subtree.

Top-Down Multiway Search Tree :

- If a multiway search tree satisfies a condition that any full node is a leaf it is called is topdown multiway sea tree.
- In addition, in a topdown multiway search tree, e semileaf must be either full or a leaf.

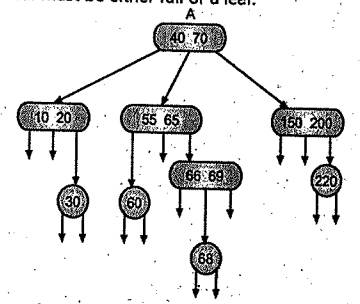


Fig. 6.2 : Topdown multiway search tree

- If all the semileaf nodes of multiway search tree are at t same level, it is called as balanced. This implies that semileaves are leaves.

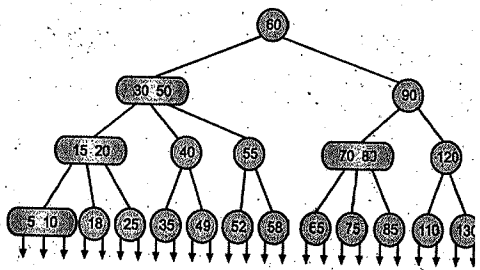


Fig. 6.3 : Balanced multiway search tree

- This tree is not top down multiway search tree becaus every non full node is not leaf node. If all the leaves are the same level, then it is called as balanced or B-trees.

B–Trees :

- Order of the tree = Maximum number of subtree attached to any node. A balanced multiway search tree order n in which each non root node contains at least n/ keys is called as B-tree of order n.
- B tree of order n is called as (n – 1) – n or n – (n – 1) tre Thus 3 – 4 tree is a B-tree of order 4 same as that of 4 – B trees are not allowed to grow at their leaves instea they grow at the root.

ertion into B-Tress :

Inserting a key in a leaf that still has some room.

Consider tree of order 5. Insert 7 key.

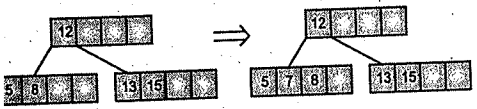


Fig. 6.4

2. Inserting a key in a leaf that is full. Insert 6 key.

Fig. 6.5

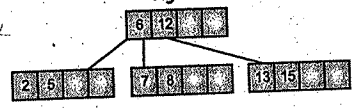


Fig. 6.6

A special case arises if the root of the B-tree is full. Insert 13 key.

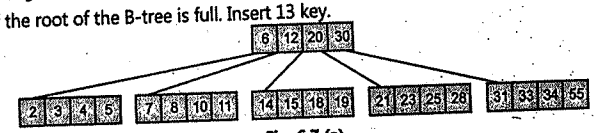


Fig. 6.7 (a)

nsert 13

13 will go into 3rd leaf

13 14 15 18 19

Median key

15 will go into root which is already full.

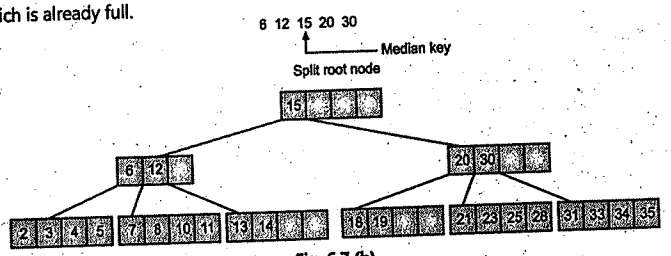
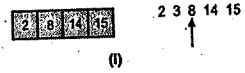


Fig. 6.7 (b)

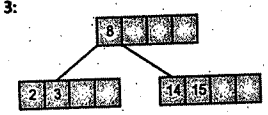
Example : 8,14,2,15,3,1,16,6,5,27,37,18,25,7,13,20,22,23,24. Build B-tree of order 5.

Solution.:

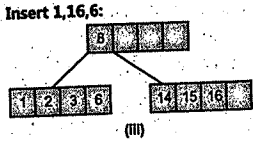
Insert 8,14,2,15:



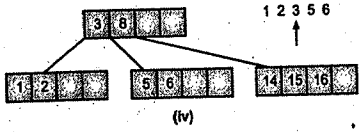
Insert 3:



Insert 1,16,6:



Insert 5:



Insert 27:

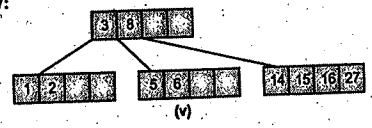


Fig. 6.8

Insert 37 :

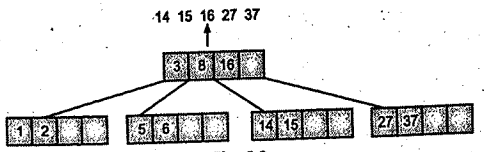


Fig. 6.9

Insert 18, 25 :

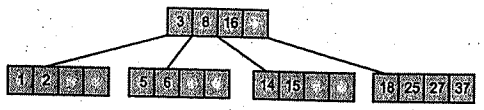


Fig. 6.10

Insert 7, 13 :

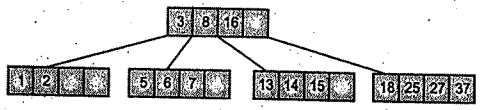


Fig. 6.11

Insert 20 :

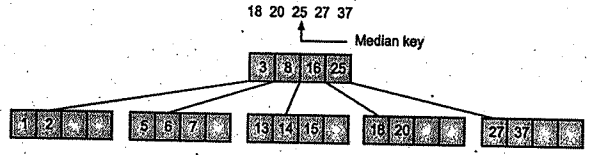


Fig. 6.12

Insert 22, 23 :

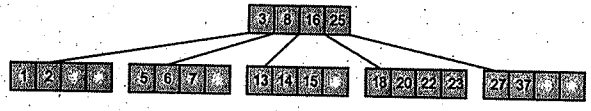


Fig. 6.13

Insert 24 :

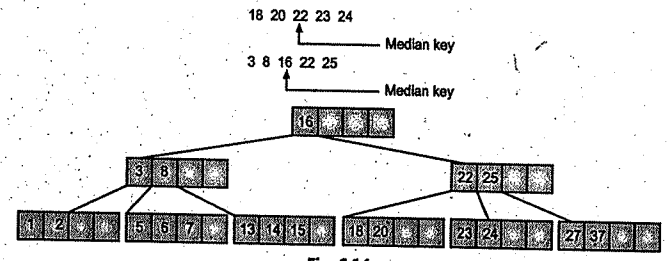


Fig. 6.14

Deletion Operation : Delete 6 from Fig. 6.14.

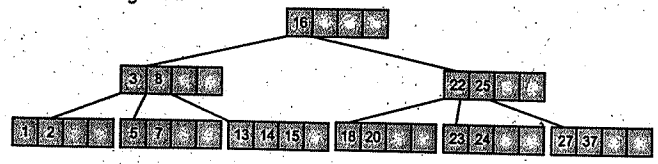


Fig. 6.15

Delete 7 :

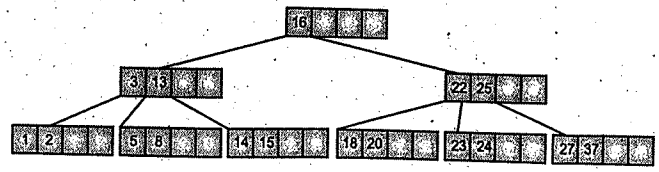


Fig. 6.16

Delete 8 :

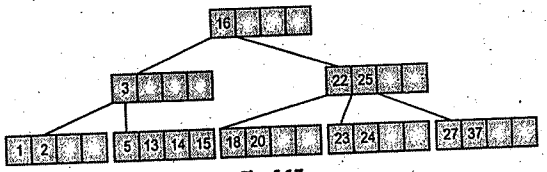


Fig. 6.17

Delete 8 : contd.

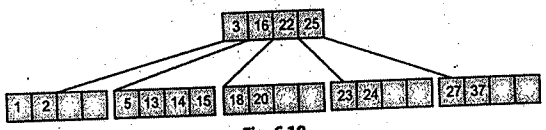


Fig. 6.18

Deleting a key from a non-leaf. The key to be deleted is replaced by its immediate successor (or the predecessor) which can only be found in a leaf.

Delete 16 :

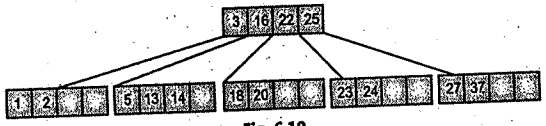


Fig. 6.19

B-trees according to their definition are guaranteed to be at least 50% full, so it may happen that 50% of space is wasted.

6.3 RED-BLACK TREES (RBT)

A red-black tree is a binary tree representation of 2-3-4 tree. The child pointers of node in a red-black tree are of 2 types : read and black. If the child pointer was present in the original 2-3-4 tree it is black pointer, other wise it is red-pointer.

Node Structure :

```
struct red_black_tree
{  enum color {red, black};
   int data;
   struct red_black_tree *left, *right, *parent;
}
```

The red node has a red pointer from its parent and a black node has a black pointer from its parent. The root node is black node.

——— Black pointer

– – – – Red pointer

Transforming a 2-node into one red black node :

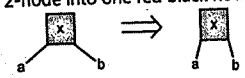


Fig. 6.20

Transforming a 3-node into 2 red black nodes:

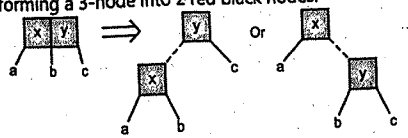


Fig. 6.21

Transforming a 4-node into 3 red black tree nodes:

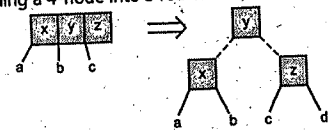


Fig. 6.22

Red-Black tree representation of 2-3-4 trees:

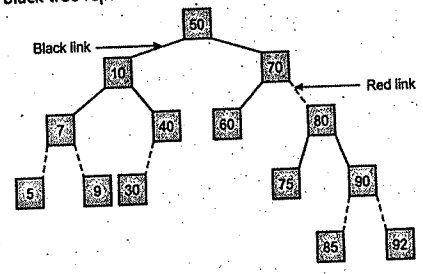


Fig. 6.23

Red-Black Tree Satisfies the Following Properties :

- It is a binary search tree.
- Every root to external node path has the same number of black links.
- No root to external node path has two or more consecutive red pointers.

6.3.1 Red-Black Trees Insertion

- New node is always red node.
- Root node is always black node.

- Every path from root to leaf has same number of black nodes.
- No two successive nodes are red nodes.

40, 60, 50, 20, 30, 45, 35, 48

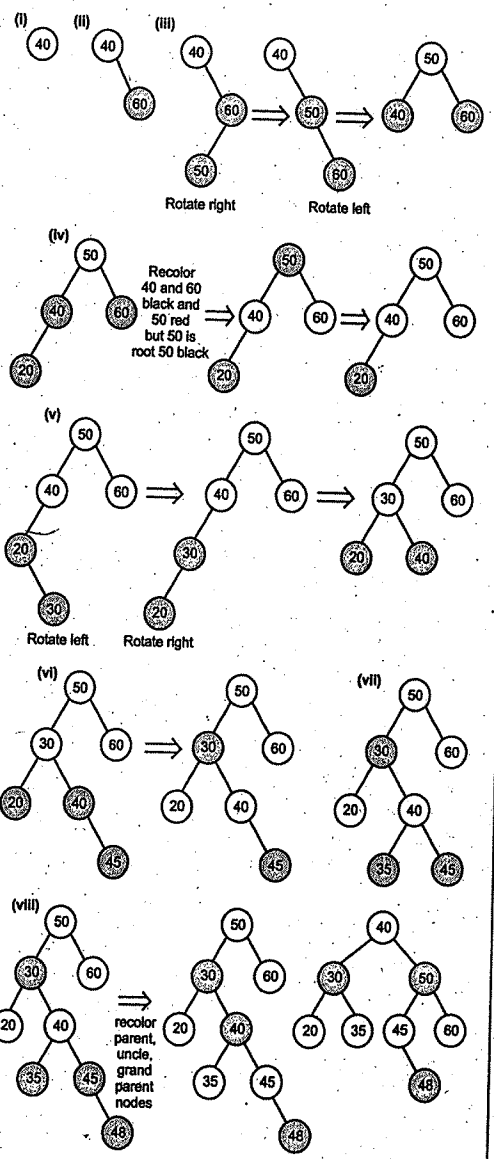


Fig. 6.24

6.3.2 Deletion from Red-Black Trees

This operation is similar to deletion operation from B Search Trees (BST) but with maintaining properties of Black Trees.

1. Delete 48 from Fig. 6.24 leaf node is red.

   Delete 48 without further adjustment.

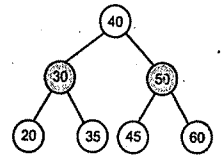


Fig. 6.25

(2) Delete 60 – Black leaf node from Fig. 6.24.

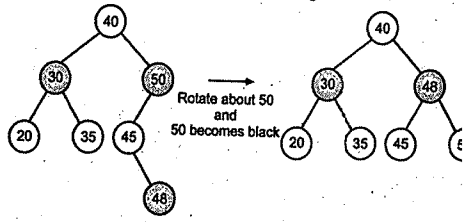


Fig. 6.26

Red-Black Trees Deletion :

Delete 50 internal red-node from Fig. 6.24.

Replace 50 by its inorder predecessor.

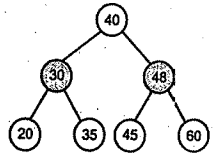


Fig. 6.27

Delete 40 – internal black node from Fig. 6.24.

Delete 40 – Internal | root black node from Fig. 6.24.

40 is replaced by its inorder successor i.e. 45 and becomes black node, attached to left of 50.

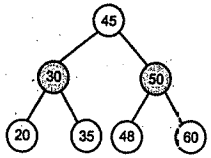


Fig. 6.28

4 ALGORITHMS AND THEIR CLASSES

We have solved many problems using different algorithms. For each algorithm we also computed its run time. The algorithms are classified into two groups depending on their computing time:

1. **P Class:** This group consists of all the algorithms whose computing times are polynomial time, that is, there computing time is bounded by polynomials of small degree. For example, insertion sort requires $O(n^2)$ time. Merge sort and quick sort require O(n log n) time. Binary search tree requires O(log n) time for searching. Polynomial evaluation requires O(n) time. All these algorithms have polynomial computing time.
2. **NP Class:** This group consists of all the algorithms whose computing times are non-deterministic polynomial time. For example, computing time of the travelling salesperson problem is $O(n^2 2^n)$ using dynamic programming. The knapsack problem takes $O(2^{n/2})$ time. These are called non-polynomial algorithms.

Generally, problems which can be solved using polynomial time algorithms are called as **Tractable (Easy)**. Problems that can be solved using super polynomial time algorithms are called **Intractable (Hard)**. Problems which require exponential time are termed as intractable. But in reality, the exponential function works better for smaller values than the polynomial functions.

The NP class problems again can be classified into two groups:

1. NP-Complete (Non-deterministic Polynomial time Complete) Problems
2. NP-Hard Problems

No NP-complete or NP-hard problem is polynomially solvable. All NP-Complete problems are NP-hard but some NP-hard problems are not NP-Complete.

The relationship between P and NP is shown in Fig. 6.29.

Since deterministic algorithms are special cases of non-deterministic algorithms. $P \subseteq NP$. Also all the NP-complete problems are NP-hard problems, but some NP-hard problems are not NP-complete.

For example, the halting problem is a NP-hard problem, but not NP-complete.

Given a deterministic algorithm A and input I, the halting problem is to determine that if algorithm A is run with input I, whether it will stop or enter in an infinite loop.

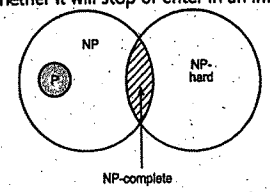


Fig. 6.29: Relation between P and NP

6.4.1 Definitions

- **Decision Problem:** Any problem having the answer either zero or one is called a decision problem. Only a decision problem can be NP-complete.
- **Decision Algorithm:** It is an algorithm which solves a decision problem. It gives output 1 on successful completion. On unsuccessful termination, it gives output 0.
- **Optimization Problem:** Any problem that involves the identification of an optimal value for a given cost function is known as an optimization problem. The optimal value can be either maximum or minimum. An optimization problem may be NP-hard.
- **Optimization Algorithm:** It is an algorithm which solves an optimization problem.
- **Polynomial Complexity:** Generally the complexity of an algorithm is measured in terms of input size. If there exists a polynomial p(n) such that the computing time of A is O(p(n)) for every input of length n, then the corresponding algorithm is said to have polynomial complexity.
- **Satisfiability Problem:** The satisfiability problem is to find out whether a given expression is true for some assignment of truth values to the variable in that expression.
- **Clique:** It is a maximal complete subgraph for a given graph. The size of the clique is the number of vertices in it.
- **Reducibility:** Let L_1 and L_2 are two problems. Problem L_2 can be solved in polynomial time using a deterministic algorithm. If the same algorithm can solve problem L_1 in deterministic polynomial time, then it is termed as L_1 reduces to L_2 ($L_1 \propto L_2$).
- **NP-Hard Problem:** A problem L is said to be NP-hard iff satisfiability $\propto$ L (satisfiability reduces to L).
- **NP-Complete Problem:** A problem L is said to be NP-complete iff L is NP-hard and $L \in NP$.

6.5 NON-DETERMINISTIC POLYNOMIAL TIME (NP) DECISION PROBLEMS

- To specify non-deterministic algorithms, we can use the following functions:
  - **Choice (Set S):** It returns one of the values in set S arbitrarily.
  - **Success ( ):** It denotes a successful completion of an algorithm.
  - **Failure ( ):** It denotes an unsuccessful termination of an algorithm.

All the above functions have O(1) computing time.

- The Choice function can return any value in the set S in any order. Hence depending on the order in which values from input are selected affects the successful termination of a non-deterministic algorithm. If there exists at least one set of choices which result in successful completion and if it is made, then the algorithm completes successfully.
- If there is no set of choices which can result successful completion of an algorithm, then the algorithm always terminates unsuccessfully. A non-deterministic machine can execute a non-deterministic algorithm. But practically such non-deterministic machines do not exist.
- Let us study some non-deterministic decision algorithms. A decision problem gives answer either 0 or 1. A non-deterministic decision algorithm does a successful completion iff it gives output 1. On unsuccessful termination, it gives output 0. Remember that this non-deterministic machine is not present practically.
- Before we proceed, let us define the complexity of a non-deterministic algorithm having input of size n as the minimum number of steps required for successful completion if there exists a sequence of choices leading to such completion. If the input is of size $n \geq n_0$, then the complexity is O(f(n)) in case of successful completion (n_0 is a constant). If no such sequence of choices exists, then the complexity of algorithm is O(1). Let us study some non-deterministic decision algorithms.

6.5.1 Non-Deterministic Search Algorithm

Suppose unordered input list L[1 : n] is an array of size n, $n \geq 1$. We want to search an element x in array L [ ]. If element x is found, its position should be output, otherwise output is 0. The algorithm is as given below:

Algorithm NSearch(L, n, x)

```
begin
    p = Choice (1, n) // Returns index between 1 to n
    // if x is found at position p, print p, else print 0
    if (L[p]=x)
    then
        print p;
        Success ( );
    end if
    print 0;
    Failure ( );
end
```

The computing time of this algorithm is O(1) in both the cases of successful search and unsuccessful search. Any deterministic search algorithm requires O(n) time.

6.5.2 Non-Deterministic 0/1 Knapsack Algorithm

In the 0/1 knapsack decision problem, array P[1: n] stores profits and array w[1 : n] stores weights of n input objects. M is the maximum capacity of knapsack and R is the minimum profit required. P_i's, w_i's, M and R all are non-negative numbers. We have to find array x[i] = 0/1 such that $\sum w_i x_i \leq M$ and $\sum p_i x_i \geq R$, for $1 \leq i \leq n$. Because it's a decision problem, the successful termination is possible only if output of this algorithm is 1.

The algorithm computes set of choices in array x[1 : n].

Algorithm NKnapsack(n, P, w, M, R, x)

```
begin
    // Initialize total profit TP and total weight TW to zero.
    TP = 0
    TW = 0
    for i = 1 to n
    begin
        x[i] = Choice (0, 1);
        TW = TW + w[i] * x [i]
        TP = TP + P[i] * x[i]
    end for
    if ((TP<R) OR (TW>M))
    then
        Failure ( );
    else
        print x [1: n]
        Success ( );
    end if
end
```

If any set of choices can lead to successful completion, the only array x[ ] is printed. The computing time of this algorith is O(n).

6.5.3 Non-Deterministic Clique Problem

- A clique is a maximal complete subgraph of a given grap G(V, E). A clique of size x in a graph is its comple subgraph of x vertices. The size of the clique is th number of vertices in it. The clique decision problem is t find out whether graph G contains a clique of size at lea x for $x \leq n$.

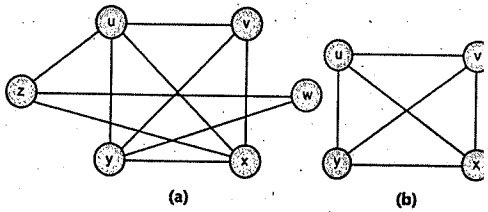


Fig. 6.30

- The graph G in Fig. 6.30 (a) has 6 vertices. A clique of siz 4 in graph G is its complete subgraph of 4 vertices. So set V' = {u, v, x, y} forms a clique of size 4 a shown in Fig. 6.30 (b) for graph G.

The algorithm takes as input: Graph G, total number o vertices n and value x. It first computes a set S having distinct vertices. Then it checks whether the set S forms complete subgraph of G. If not, then the algorithn terminates unsuccessfully, otherwise it complete successfully.

Algorithm NClique(G, n, x)

```
begin
  //Initialize set S to empty set
  S = 0
  //compute set S
  for i = 1 to x
  begin
     y = Choice (1, n)
     if (y ∈ S) then Failure ( );
     //otherwise add y to set S
     S = S U {y}
  end for

  //check whether S forms clique
  for each pair (i, j) of vertices in S such that i∈S, j∈S,
i ≠ j
  begin
     if edge <i, j> ∉ E
        then Failure( );
  end for
  Success ( );
end
```

The time complexity of this algorithm is $O(n + x^2) = O(n^2)$, because $x \leq n$.

6.5.4 Non-Deterministic Sorting Algorithm

Let P[1: n] is an input array to be sorted in ascending order. The sequence in which Choice(1, n) returns indices is used to create output array Q[ ] from P[ ]. Finally if Q[ ] is sorted, then algorithm completes successfully, otherwise it terminates unsuccessfully.

Algorithm NSort(P, n)

```
begin
  //Initialize array Q to 0
  for i = 1 to n
     Q[i] = 0
  for i = 1 to n
  begin
     x = Choice (1, n)
     if Q[x] = 0
        then
        Q[x] = P[i]
     else
        Failure( );
     end if
  end for
// check whether Q is sorted
  for i = 1 to n - 1
     if (Q[i] > Q[i+1]) then
        Failure ( );
     end if
  end for
  Success ( );
end
```

The computing time of this algorithm is O(n + n + n) = O(n). Whereas the deterministic algorithms have minimum complexity O(n log n).

6.5.5 Non-Deterministic Satisfiability Problem

- Let $x_1, x_2, ..., x_n$ represent Boolean variables which can take value true or false. Negation of x_i is represented as $\bar{x}_i$. Using variables and/or their negations, we can construct an expression in propositional calculus using Boolean AND, OR operations. The satisfiability problem is to find out whether an expression is true for some assignment of truth values to the variables in that expression.
- Let $E(x_1, x_2, ..., x_n)$ is an expression. The following algorithm uses a set of choices for all the variables $x_1, ..., x_n$. The algorithm completes successfully, if an expression E evaluates to true, otherwise algorithm terminates unsuccessfully. Such a polynomial time non-deterministic algorithm is as follows:

Algorithm NSatisfiability(E, n)

```
begin
  //Let x1, x2, ..., xn are variables used in E
  for i = 1 to n
     x[i] = Choice(true, false);
  end for
  //Evaluate expression E for these x's
  If E (x1, ---, xn) is true
  then
     Success ( )
  else
     Failure ( )
  end if
end
```

- The for loop computes set of choices for x_i's in O(n) time. The expression E can be deterministically evaluated in time O(f(n)). Then computing time of algorithm is O(n) + O(f(n)). This algorithm tells that satisfiability is in NP.

6.6 POLYNOMIAL-TIME REDUCTION

To be NP-Complete, a decision problem must belong to NP and it must be possible to polynomially reduce any other problem in NP to that problem.

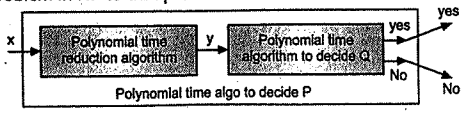


Fig. 6.31

6.7 NP-COMPLETENESS

Definition:

A decision problem X is NP-Complete if

$X \in$ NP and

$Y \leq_p X$ for every problem $Y \in$ NP.

We have already shown that some problems are NP-Complete. If we want to show other problems to be NP-Complete, then we can use the following theorem.

Theorem 6.1: Let X be a NP-Complete problem. Consider a decision problem Z ∈ NP such that $X \leq_p Z$, then Z is also NP-Complete.

≤ denotes "less than or equal to". $X \leq_p Z$ denotes that X is less harder than Z by a polynomial factor p.

Formal Definition of NP-Complete

A language L is NPC if

- L ∈ NP and
- $L' \leq p$ L for every L' ∈ NP

If the second condition is satisfied but condition one is not necessarily satisfied then it is said that L is NP-hard.

6.7.1 Satisfiability Problem

- Let $x_1, x_2, ..., x_n$ be Boolean variables. Negation of x_i is $\bar{x}_i$. A literal is a variable or its negation. An expression consists of one or more literals. A Boolean expression is said to be in Conjunctive Normal Form (CNF) if it is the product of sums of literals. In prepositional calculus, ∧ is termed as conjunction and ∨ is termed as disjunction. For example, $(x_1 + \bar{x}_2)$ $(x_2 + \bar{x}_1)$ is in CNF, which is represented as $(x_1 + \bar{x}_2) \wedge (x_2 + \bar{x}_1)$ in prepositional calculus.
- A Boolean expression is in k-CNF for some positive integer k if it is composed of clauses, each of which contains at most k literals. For CNF, a clause is either a literal or a disjunction of literals. For example, $(p + \bar{q})$ $(p + q + \bar{r})$ $\bar{p}$ r is in 3-CNF because second clause contains maximum 3 literals. (p + qr) $(\bar{p} + \bar{q}$ $(p + \bar{r}))$ is not in CNF because first clause contains conjunction qr.

Definition :

- A Boolean expression is satisfiable if it is true for some assignment of truth values to its variables. The problem of deciding whether a given Boolean expression is satisfiable or not, is denoted as SAT.
- For example, consider a Boolean expression $(a \vee b) \Rightarrow (a \wedge b)$. If we assign a = b = true, then the given expression becomes true. Hence it is satisfiable. The Boolean expression $(a \vee b) \wedge \bar{a} \wedge \bar{b}$ is not satisfiable because for all possible assignments of truth values to a and b, the given expression remains false.
- Whenever we want to find out whether given Boolean expression is satisfiable or not, we can try all the possible combinations of input variables. But it is not practical if number of input variables n is large enough. Because for n input variables, there are 2^n possible truth values combinations. To solve this problem no effic algorithm is available.
- We know, that every problem in NP reduces satisfiability and also to CNF-satisfiability. Also si satisfiability is in NP, and the construction of Q as (formula shows that satisfiability ∝ CNF-satisfiability. Al CNF-satisfiability is NP. This all together implies CNF-satisfiability is NP-Complete.

Theorem 6.2: CNF-satisfiability (SAT-CNF) is NP-Complete. We can apply theorem 7.5.1 to prove the NP-completenes other problems.

Theorem 6.3: SAT is NP-complete.

Proof: We know that SAT is in NP. If we show that SAT-CNI SAT, then we can apply theorem 4.1. We know that Bool expression in CNF is a special case of general Bool expression. Given a Boolean expression, we can tell whethe is in CNF or not. Hence any algorithm which can solve S efficiently can be used to solve SAT-CNF. Hence SAT is a NP-complete.

6.7.2 Circuit – SAT

Circuit-satisfiability problem is the 1st NP. It states that give Boolean combinations circuit composed of AND, OR and N gates. We wish to find, whether there is any set of Boole inputs to this circuit that causes its output to be 1.

Tree of NP Complete Problem :

SAT :

- The formula satisfiability problem is an NP compl problem.
- It is the problem of finding whether a Boolean formu (not a circuit) is satisfiable.
- CNF-SAT or 3-SAT or 3-CNF :
  - Exactly 2 literals in each clause then it is 2-CNF.
  - Exactly 3 literals in each clause then it is 3-CNF.

3-CNF

A Boolean formula is in 3 conjunctive normal form i.e. 3-CN if each clause has exactly 3 literals. (Variable or negation variable).

3-CNF-SAT

- It is to find whether a given Boolean formula ϕ in 3-CNF is satisfiable.
- Example: $(x_1 \vee \neg x_2 \vee \neg x_1) \wedge (x_3 \vee x_2 \vee x_4) \wedge (\neg x_1 \vee \neg x_3 \vee \neg x_4)$

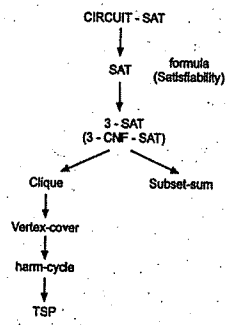


Fig. 6.32 : Tree of NP-Complete problems

3-CNF-SAT Problem Statement

Satisfiability of Boolean formula in 3-Conjunctive Normal fori is NP complete.

MULTIPLE CHOICE QUESTIONS (MCQ's)

1. What is the special property of red-black trees and what root should always be?
   (a) A color which is either red or black and root should always be black color only
   (b) Height of the tree
   (c) Pointer to next node
   (d) A color which is either green or black
2. Why do we impose restrictions like
   - Root property is black
   - Every leaf is black
   - Children of red node are black
   - All leaves have same black
   (a) To get logarithm time complexity
   (b) To get linear time complexity
   (c) To get exponential time complexity
   (d) To get constant time complexity
3. What are the operations that could be performed in O(logn) time complexity by red-black tree?
   (a) insertion, deletion, finding predecessor, successor
   (b) only insertion
   (c) only finding predecessor, successor
   (d) for sorting
4. When it would be optimal to prefer Red-black trees over AVL trees?
   (a) when there are more insertions or deletions
   (b) when more search is needed
   (c) when tree must be balanced
   (d) when log(nodes) time complexity is needed
5. When to choose Red-Black tree, AVL tree and B-trees?
   (a) many inserts, many searches and when managing more items respectively
   (b) many searches, when managing more items respectively and many inserts respectively
   (c) sorting, sorting and retrieval respectively
   (d) retrieval, sorting and retrieval respectively
6. State true or false: B+ trees are not always balanced trees.
   (a) True (b) False
7. What are the leaf nodes in a B+ tree?
   (a) The topmost nodes
   (b) The bottommost nodes
   (c) The nodes in between the top and bottom nodes
   (d) None of the mentioned
8. Leaves of which of the following trees are at the same level?
   (a) Binary tree (b) B-tree
   (c) AVL-tree (d) Expression tree
9. What does NP stands for in complexity classes' theory?
   (a) Non polynomial
   (b) Non-deterministic polynomial
   (c) Both (a) and (b)
   (d) None of the mentioned
10. The hardest of NP problems can be:
    (a) NP-complete (b) NP-hard
    (c) P (d) None of the mentioned
11. Travelling sales man problem belongs to which of the class?
    (a) P (b) NP
    (c) Linear (d) None of the mentioned
12. A problem which is both _____ and _____ is said to be NP complete.
    (a) NP, P (b) NP, NP hard
    (c) P, P complete (d) None of the mentioned

ANSWERS

1. (a)	2. (a)	3. (a)	4. (a)	5. (a)
6. (b)	7. (b)	8. (b)	9. (b)	10. (b)
11. (b)	12. (b)			

EXERCISE

1. What do you mean by Red Black trees, B trees?
2. Explain insertion and deletion operations of Red Black and B trees.
3. What are the characteristics of red Black trees?
4. Differentiate between by Red Black trees, B trees.
5. Write a short note on:
   (a) P complexity class.
   (b) NP complexity class.
6. Explain the following terms:
   (a) Computational complexity.
   (b) Optimization problem.
   (c) Decision problem.
   (d) Stable algorithm.
   (e) Tractability and intractability.
7. Explain deterministic and non-deterministic algorithms.
8. When do you claim that a problem is NP hard ?
9. Explain polynomial time reduction w.r.t. suitable example.

NOTES