



**MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY**

**(AUTONOMOUS INSTITUTION – UGC, GOVT. OF INDIA)**



**Department of CSE**

**(Emerging Technologies)**

**(Data Science, Cyber Security and Internet of Things)**



**DESIGN AND ANALYSIS OF  
ALGORITHMS  
(R20A0505)**

**LECTURE NOTES**

# DESIGN AND ANALYSIS OF ALGORITHMS



## LECTURE NOTES

**B.TECH (R-20 Regulation)  
(II YEAR – II SEM)  
(2022-23)**

## **DEPARTMENT OF CSE (EMERGING TECHNOLOGIES)**

**(Data Science, Cyber Security and Internet of Things)**



## **MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY**

**(Autonomous Institution – UGC, Govt. of India)**

Recognized under 2(f) and 12 (B) of UGC ACT 1956

(Affiliated to JNTUH, Hyderabad, Approved by AICTE - Accredited by NBA & NAAC – 'A' Grade - ISO 9001:2015 Certified)  
Maisammaguda, Dhulapally (Post Via. Hakimpet), Secunderabad – 500100, Telangana State, India

## **Department of Computer Science and Engineering**

# **(EMERGING TECHNOLOGIES)**

### **Vision**

- ❖ To be at the forefront of Emerging Technologies and to evolve as a Centre of Excellence in Research, Learning and Consultancy to foster the students into globally competent professionals useful to the Society.

### **Mission**

**The department of CSE (Emerging Technologies) is committed to:**

- ❖ To offer highest Professional and Academic Standards in terms of Personal growth and satisfaction.
- ❖ Make the society as the hub of emerging technologies and thereby capture opportunities in new age technologies.
- ❖ To create a benchmark in the areas of Research, Education and Public Outreach.
- ❖ To provide students a platform where independent learning and scientific study are encouraged with emphasis on latest engineering techniques.

### **Quality Policy**

- ❖ To pursue continual improvement of teaching learning process of Undergraduate and Post Graduate programs in Engineering & Management vigorously.
- ❖ To provide state of art infrastructure and expertise to impart the quality education and research environment to students for a complete learning experiences.
- ❖ Developing students with a disciplined and integrated personality.
- ❖ To offer quality relevant and cost effective programmes to produce engineers as per requirements of the industry need.

**For more information: [www.mrcet.ac.in](http://www.mrcet.ac.in)**

## INDEX

S. No	Unit	Topic	Page no
1	I	Introduction to Algorithms	5
2	I	Divide and Conquer	14
3	II	Disjoint Sets	26
4	II	Greedy Method	38
5	III	Dynamic Programming	51
6	IV	Back Tracking	63
7	V	Branch and Bound	73
8	V	NP-Hard and NP-Complete Problems	93

**MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY****II Year B. Tech CSE-Emerging Technologies-Cyber Security - II Sem****L T/P/D C****3 -/-/ 3****(R20A0505) DESIGN AND ANALYSIS OF ALGORITHMS****COURSE OBJECTIVES:**

1. To analyze performance of algorithms.
2. To choose the appropriate data structure and algorithm design method for a specified application.
3. To understand how the choice of data structures and algorithm design methods impacts the performance of programs.
4. To solve problems using algorithm design methods such as the greedy method, divide and conquer, dynamic programming, backtracking and branch and bound.
5. To understand the differences between tractable and intractable problems and to introduce P and NP classes.

**UNIT-I** Introduction: Algorithms, Pseudo code for expressing algorithms, performance analysis- Space complexity, Time Complexity, Asymptotic notation- Big oh notation, omega notation, theta notation and little oh notation. Divide and Conquer: General method. Applications- Binary search, Quick sort, merge sort, Strassen's matrix multiplication.

**UNIT-II** Disjoint set operations, Union and Find algorithms, AND/OR graphs, Connected components, Bi-connected components.

Greedy method: General method, applications- Job sequencing with deadlines, Knapsack problem, Spanning trees, Minimum cost spanning trees, Single source shortest path problem.

**UNIT-III** Dynamic Programming: General method, applications- Matrix chained multiplication, Optimal binary search trees, 0/1 Knapsack problem, All pairs shortest path problem, Traveling sales person problem, Reliability design.

**UNIT-IV** Backtracking: General method, Applications- n-queue problem, Sum of subsets problem, Graph coloring, Hamiltonian cycles.

**UNIT-V** Branch and Bound: General method, applications- Travelling sales person problem, 0/1 Knapsack problem- LC branch and Bound solution, FIFO branch and Bound solution.

NP-Hard and NP-Complete Problems: Basic concepts, Non deterministic algorithms, NP-Hard and NPComplete classes, NP-Hard problems, Cook's theorem.

**TEXT BOOKS:**

1. Fundamentals of Computer Algorithms, Ellis Horowitz, SartajSahni and Rajasekharan, Universities press
2. Design and Analysis of Algorithms, P. h. Dave, 2nd edition, Pearson Education.

**REFERENCES:**

1. Introduction to the Design And Analysis of Algorithms A Levitin Pearson Education
2. Algorithm Design foundations Analysis and Internet examples, M.T.Goodrich and R Tomassia John Wiley and sons
3. Design and Analysis of Algorithms, S. Sridhar, Oxford Univ.Press
4. Design and Analysis of Algorithms, Aho , Ulman and Hopcraft , Pearson Education.
5. Foundations of Algorithms, R. Neapolitan and K. Naimipour , 4th edition

## UNIT-I

Introduction: Algorithms, Pseudo code for expressing algorithms, performance analysis- Space complexity, Time Complexity, Asymptotic notation- Big oh notation, omega notation, theta notation and little oh notation.

Divide and Conquer: General method. Applications- Binary search, Quick sort, merge sort, Strassen's matrix multiplication.

### INTRODUCTION TO ALGORITHM

#### What is an Algorithm?

**Algorithm** is a set of steps to complete a task.

**For example,**

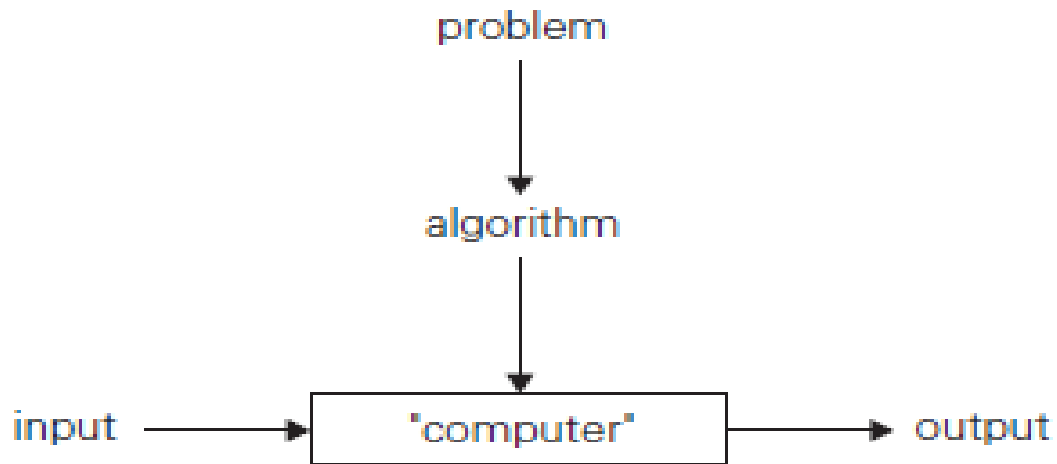
**Task: to make a cup of tea.**

Algorithm:

- add water and milk to the kettle,
- boil it, add tea leaves,
- Add sugar, and then serve it in cup.

**"a set of steps to accomplish or complete a task that is described precisely enough that a computer can run it ".**

- An algorithm is a finite set of instructions that, if followed, accomplishes a particular task. In addition, all algorithms must satisfy the following criteria:
- Input. Zero or more quantities are externally supplied.
- Output. At least one quantity is produced.
- Definiteness. Each instruction is clear and unambiguous.
- Finiteness. The algorithm terminates after a finite number of steps.
- Effectiveness. Every instruction must be very basic enough and must be feasible.
- Algorithms that are definite and effective are also called *computational procedures*.
- A program is the expression of an algorithm in a programming language



### **PSEUDOCODE:**

- Algorithm can be represented in Text mode and Graphic mode
- Graphical representation is called Flowchart
- Text mode most often represented in close to any High level language such as C, Pascal Pseudocode
- **Pseudocode: High-level description of an algorithm.**
  - ❑ More structured than plain English.
  - ❑ Less detailed than a program.
  - ❑ Preferred notation for describing algorithms.
  - ❑ Hides program design issues.
- **Example of Pseudocode:**
- To find the max element of an array

```
Algorithm arrayMax(A, n)  
Input array A of n integers  
Output maximum element of A  
currentMax  $\leftarrow$  A[0]  
for i  $\leftarrow$  1 to n - 1 do  
  if A[i] > currentMax then  
    currentMax  $\leftarrow$  A[i]  
return currentMax
```

## **RULES FOR PSEUDOCODE**

1. Write only one stmt per line Each stmt in your pseudocode should express just one action for the computer. If task list is properly drawn, then in most cases each task will correspond to one line of pseudocode.

2. Capitalize initial keyword In the example above, READ and WRITE are in caps. There are just a few keywords we will use: READ, WRITE, IF, ELSE, ENDIF, WHILE, ENDWHILE, REPEAT, UNTIL

3. Indent to show hierarchy We will use a particular indentation pattern in each of the design structures:

SEQUENCE: keep statements that are “stacked” in sequence all starting in the same column.

SELECTION: indent the statements that fall inside the selection structure, but not the keywords that form the selection.

LOOPING: indent the statements that fall inside the loop, but not the keywords that form the loop

4. Keep stmts language independent Resist the urge to write in whatever language you are most comfortable with. In the long run, you will save time! There may be special features available in the language you plan to eventually write the program in; if you are SURE it will be written in that language, then you can use the features. If not, then avoid using the special features.

## **PERFORMANCE ANALYSIS:**

- What are the Criteria for judging algorithms that have a more direct relationship to performance?
- computing time and storage requirements.

**Performance evaluation** can be loosely divided into two major phases:

- a priori estimates and
- a posteriori testing.
- The space complexity of an algorithm is the amount of memory it needs to run to completion.
- The time complexity of an algorithm is the amount of computer time it needs to run to completion.

### **Space Complexity:**

- Space Complexity Example:

Algorithm abc(a,b,c)

```
{  
    return a+b++*c+(a+b-c)/(a+b) +4.0;  
}
```

The Space needed by each of these algorithms is seen to be the sum of the following component.



1. A fixed part that is independent of the characteristics (eg: number, size) of the inputs and outputs. The part typically includes the instruction space (ie. Space for the code), space for simple variable and fixed-size component variables (also called aggregate) space for constants, and so on.

2. A variable part that consists of the space needed by component variables whose size is dependent on the particular problem instance being solved, the space needed by referenced variables (to the extent that it depends on instance characteristics), and the recursion stack space.

The space requirement  $s(p)$  of any algorithm  $p$  may therefore be written as,  
 $S(P) = c + S_p(\text{Instance characteristics})$   
Where 'c' is a constant.

### Example 2:

Algorithm sum(a,n)

```
{  
s=0.0;  
for I=1 to n do  
s= s+a[I];  
return s;  
}
```

- The problem instances for this algorithm are characterized by  $n$ , the number of elements to be summed. The space needed by 'n' is one word, since it is of type integer.
- The space needed by 'a' is the space needed by variables of type array of floating point numbers.
- This is at least 'n' words, since 'a' must be large enough to hold the 'n' elements to be summed.
- So, we obtain  $S_{\text{sum}}(n) \geq (n+s)$  [n for a[], one each for n, I a & s]

### Time Complexity:

- The time  $T(p)$  taken by a program  $P$  is the sum of the compile time and the run time (execution time)
- The compile time does not depend on the instance characteristics. Also we may assume that a compiled program will be run several times without recompilation. This run time is denoted by  $t_p(\text{instance characteristics})$ .
- The number of steps any problem statement is assigned depends on the kind of statement.
- For example, comments à 0 steps.  
Assignment statements is 1 steps.

[Which does not involve any calls to other algorithms]

Interactive statement such as for, while & repeat-until à Control part of the statement.

We introduce a variable, count into the program statement to increment count with initial value 0. Statement to increment count by the appropriate amount are introduced into the program.

This is done so that each time a statement in the original program is executed count is incremented by the step count of that statement.

**Algorithm:**

Algorithm sum(a,n)

```

{
s= 0.0;
count = count+1;
for I=1 to n do
{
count =count+1;
s=s+a[I];
count=count+1;
}
count=count+1;
count=count+1;
return s;
}

```

- 1. If the count is zero to start with, then it will be  $2n+3$  on termination. So each invocation of sum execute a total of  $2n+3$  steps.
- 2. The second method to determine the step count of an algorithm is to build a table in which we list the total number of steps contributed by each statement.
- ☐ First determine the number of steps per execution (s/e) of the statement and the total number of times (ie., frequency) each statement is executed.
- ☐ By combining these two quantities, the total contribution of all statements, the step count for the entire algorithm is obtained.

<i>Statement</i>	<i>Steps per execution</i>	<i>Frequency</i>	<i>Total</i>
1. Algorithm Sum(a,n)	0	-	0
2. {	0	-	0
3. S=0.0;	1	1	1
4. for I=1 to n do	1	$n+1$	$n+1$
5. s=s+a[I];	1	$n$	$n$
6. return s;	1	1	1
7. }	0	-	0
<b>Total</b>			<b><math>2n+3</math></b>

We usually consider one algorithm to be more efficient than another if its worst-case running time has a smaller order of growth.

### **Complexity of Algorithms**

The complexity of an algorithm  $M$  is the function  $f(n)$  which gives the running time and/or storage space requirement of the algorithm in terms of the size 'n' of the input data. Mostly, the storage space required by an algorithm is simply a multiple of the data size 'n'.

Complexity shall refer to the running time of the algorithm.

The function  $f(n)$ , gives the running time of an algorithm, depends not only on the size 'n' of the input data but also on the particular data. The complexity function  $f(n)$  for certain cases are:

1. **Best Case** : The minimum possible value of  $f(n)$  is called the best case.
2. **Average Case** : The expected value of  $f(n)$ .
3. **Worst Case** : The maximum value of  $f(n)$  for any key possible input.

### **ASYMPTOTIC NOTATION**

#### **□ Formal way notation to speak about functions and classify them**

The following notations are commonly use notations in performance analysis and used to characterize the complexity of an algorithm:

1. Big-OH ( $O$ ),
2. Big-OMEGA ( $\Omega$ ),
3. Big-THETA ( $\Theta$ ) and
4. Little-OH ( $o$ )

#### **Asymptotic Analysis of Algorithms:**

Our approach is based on the *asymptotic complexity* measure. This means that we don't try to count the exact number of steps of a program, but how that number grows with the size of the input to the program. That gives us a measure that will work for different operating systems, compilers and CPUs. The asymptotic complexity is written using big-O notation.

- It is a way to describe the characteristics of a function in the limit.
- It describes the rate of growth of functions.
- Focus on what's important by abstracting away low-order terms and constant factors.
- It is a way to compare "sizes" of functions:

$$O \approx \leq$$

$$\begin{aligned}\Omega &\approx \geq \\ \Theta &\approx = \\ o &\approx < \\ \omega &\approx >\end{aligned}$$

Time complexity	Name	Example
$O(1)$	Constant	Adding an element to the front of a linked list
$O(\log n)$	Logarithmic	Finding an element in a sorted array
$O(n)$	Linear	Finding an element in an unsorted array
$O(n \log n)$	Linear	Logarithmic Sorting n items by 'divide-and-conquer'-Mergesort
$O(n^2)$	Quadratic	Shortest path between two nodes in a graph
$O(n^3)$	Cubic	Matrix Multiplication
$O(2^n)$	Exponential	The Towers of Hanoi problem

**Big 'oh':** the function  $f(n)=O(g(n))$  iff there exist positive constants  $c$  and  $n_0$  such that  $f(n) \leq c \cdot g(n)$  for all  $n, n \geq n_0$ .

**Omega:** the function  $f(n)=\Omega(g(n))$  iff there exist positive constants  $c$  and  $n_0$  such that  $f(n) \geq c \cdot g(n)$  for all  $n, n \geq n_0$ .

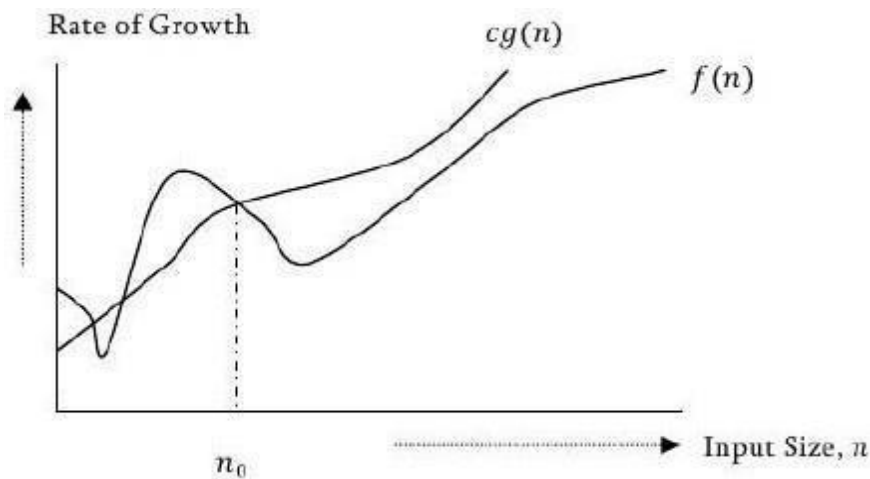
**Theta:** the function  $f(n)=\Theta(g(n))$  iff there exist positive constants  $c_1, c_2$  and  $n_0$  such that  $c_1 g(n) \leq f(n) \leq c_2 g(n)$  for all  $n, n \geq n_0$

### **Big-O Notation**

This notation gives the tight upper bound of the given function. Generally we represent it as  $f(n) = O(g(n))$ . That means, at larger values of  $n$ , the upper bound of  $f(n)$  is  $g(n)$ . For example, if  $f(n) = n^4 + 100n^2 + 10n + 50$  is the given algorithm, then  $n^4$  is  $g(n)$ . That means  $g(n)$  gives the maximum rate of growth for  $f(n)$  at larger values of  $n$ .

**O —notation** defined as  $O(g(n)) = \{f(n): \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq c \cdot g(n) \text{ for all } n \geq n_0\}$ .  $g(n)$  is an asymptotic tight upper bound for  $f(n)$ . Our objective is to give some rate of growth  $g(n)$  which is greater than given algorithm's rate of growth  $f(n)$ .

In general, we do not consider lower values of  $n$ . That means the rate of growth at lower values of  $n$  is not important. In the below figure,  $n_0$  is the point from which we consider the rate of growths for a given algorithm. Below  $n_0$  the rate of growths may be different.



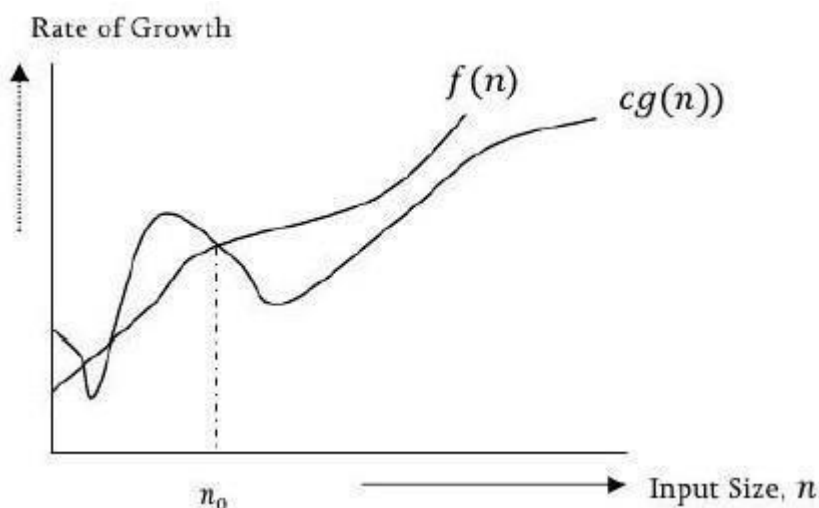
Note Analyze the algorithms at larger values of  $n$  only What this means is, below  $n_0$  we do not care for rates of growth.

### **Omega— $\Omega$ notation**

Similar to above discussion, this notation gives the tighter lower bound of the given algorithm and we represent it as  $f(n) = \Omega(g(n))$ . That means, at larger values of  $n$ , the tighter lower bound of  $f(n)$  is  $g$

For example, if  $f(n) = 100n^2 + 10n + 50$ ,  $g(n)$  is  $\Omega(n^2)$ .

The  $\Omega$  notation as be defined as  $\Omega(g(n)) = \{f(n): \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0\}$ .  $g(n)$  is an asymptotic lower bound for  $f(n)$ .  $\Omega(g(n))$  is the set of functions with smaller or same order of growth as  $f(n)$ .

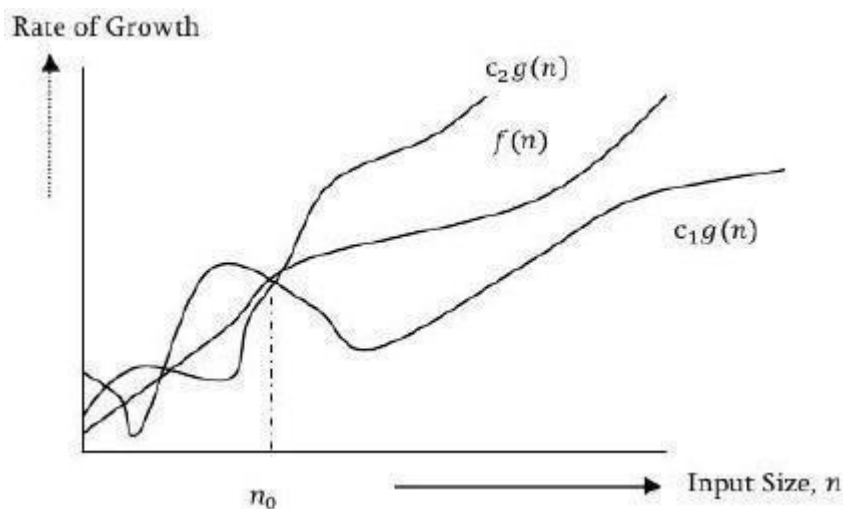


### **Theta- $\Theta$ notation**

This notation decides whether the upper and lower bounds of a given function are same or not. The average running time of algorithm is always between lower bound and upper bound.

If the upper bound (O) and lower bound ( $\Omega$ ) gives the same result then  $\Theta$  notation will also have the same rate of growth. As an example, let us assume that  $f(n) = 10n + n$  is the expression. Then, its tight upper bound  $g(n)$  is  $O(n)$ . The rate of growth in best case is  $g(n) = O(n)$ . In this case, rate of growths in best case and worst are same. As a result, the average case will also be same.

None: For a given function (algorithm), if the rate of growths (bounds) for O and  $\Omega$  are not same then the rate of growth  $\Theta$  case may not be same.



Now consider the definition of  $\Theta$  notation It is defined as  $\Theta(g(n)) = \{f(n) : \text{there exist positive constants } C_1, C_2 \text{ and } n_0 \text{ such that } C_1g(n) \leq f(n) \leq C_2g(n) \text{ for all } n \geq n_0\}$ .  $g(n)$  is an asymptotic tight bound for  $f(n)$ .  $\Theta(g(n))$  is the set of functions with the same order of growth as  $g(n)$ .

### Important Notes

For analysis (best case, worst case and average) we try to give upper bound (O) and lower bound ( $\Omega$ ) and average running time ( $\Theta$ ). From the above examples, it should also be clear that, for a given function (algorithm) getting upper bound (O) and lower bound ( $\Omega$ ) and average running time ( $\Theta$ ) may not be possible always.

For example, if we are discussing the best case of an algorithm, then we try to give upper bound (O) and lower bound ( $\Omega$ ) and average running time ( $\Theta$ ).

In the remaining chapters we generally concentrate on upper bound (O) because knowing lower bound ( $\Omega$ ) of an algorithm is of no practical importance and we use  $\Theta$  notation if upper bound (O) and lower bound ( $\Omega$ ) are same.

### **Little Oh Notation**

The little Oh is denoted as o. It is defined as : Let,  $f(n)$  and  $g(n)$  be the non negative functions then

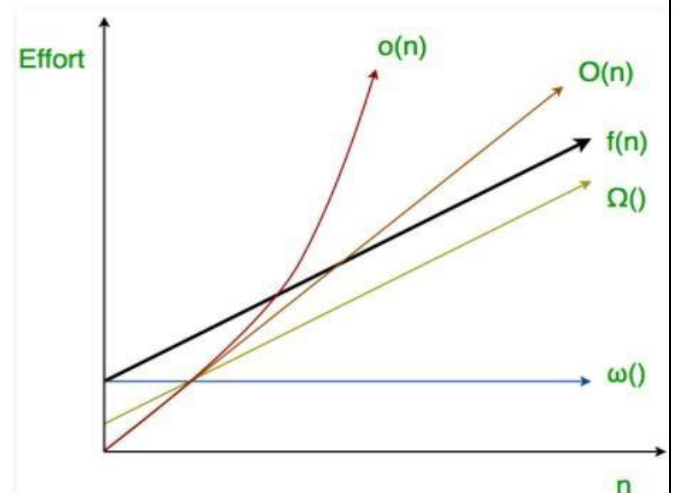
### Little o asymptotic notation

Big-O is used as a tight upper-bound on the growth of an algorithm's effort (this effort is described by the function  $f(n)$ ), even though, as written, it can also be a loose upper-bound. "Little-o" ( $o()$ ) notation is used to describe an upper-bound that cannot be tight.

**Definition :** Let  $f(n)$  and  $g(n)$  be functions that map positive integers to positive real numbers. We say that  $f(n)$  is  $o(g(n))$  (or  $f(n) \in o(g(n))$ ) if for **any real** constant  $c > 0$ , there exists an integer constant  $n_0 \geq 1$  such that  $0 \leq f(n) < c \cdot g(n)$ .

Thus, little  $o()$  means **loose upper-bound** of  $f(n)$ . Little  $o$  is a rough estimate of the maximum order of growth whereas Big-O may be the actual order of growth.

In mathematical relation,  
 $f(n) = o(g(n))$  means  
 $\lim_{n \rightarrow \infty} f(n)/g(n) = 0$



## DIVIDE AND CONQUER

### General Method

In divide and conquer method, a given problem is,

- i) Divided into smaller subproblems.
- ii) These subproblems are solved independently.
- iii) Combining all the solutions of subproblems into a solution of the whole.

If the subproblems are large enough then divide and conquer is reapplied. The generated subproblems are usually of some type as the original problem.

Hence recursive algorithms are used in divide and conquer strategy.

```

Algorithm DAndC(P)
{
if small(P) then return
S(P)else{
divide P into smaller instances P1,P2,P3...Pk;
apply DAndC to each of these subprograms; // means DAndC(P1), DAndC(P2).....
DAndC(Pk)
return combine(DAndC(P1), DAndC(P2)..... DAndC(Pk));
}
}
//P→Problem
//Here small(P)→ Boolean value function. If it is true, then the function S is
//invoked

```

### Time Complexity of DAndC algorithm:

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

$a, b$  □ constants.

This is called the **general divide and-conquer recurrence**.

### Example for GENERAL METHOD:

As an example, let us consider the problem of computing the sum of  $n$  numbers  $a_0, \dots a_{n-1}$ .

If  $n > 1$ , we can divide the problem into two instances of the same problem. They are sum of the first  $\lfloor n/2 \rfloor$  numbers

Compute the sum of the 1<sup>st</sup>  $\lfloor n/2 \rfloor$  numbers, and then compute the sum of another  $n/2$  numbers. Combine the answers of two  $n/2$  numbers sum.

i.e.,

$$a_0 + \dots + a_{n-1} = (a_0 + \dots + a_{n/2}) + (a_{n/2} + \dots + a_{n-1})$$

Assuming that size  $n$  is a power of  $b$ , to simplify our analysis, we get the following recurrence for the running time  $T(n)$ .

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

This is called the general **divide and-conquer recurrence**.

$f(n)$  □ is a function that accounts for the time spent on dividing the problem into smaller ones and on combining their solutions. (For the summation example,  $a = b = 2$  and  $f(n) = 1$ .)

### Advantages of DAndC:

The time spent on executing the problem using DAndC is smaller than other method.

This technique is ideally suited for parallel computation.

This approach provides an efficient algorithm in computer science.

### Master Theorem for Divide and Conquer

In all efficient divide and conquer algorithms we will divide the problem into subproblems, each of which is some part of the original problem, and then perform some additional work to compute the final answer. As an example, if we consider merge sort [for details, refer Sorting chapter], it operates on two problems, each of which is half the size of the original, and then uses  $O(n)$  additional work for merging. This gives the running time equation:



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

The following theorem can be used to determine the running time of divide and conquer algorithms. For a given program or algorithm, first we try to find the recurrence relation for the problem. If the recurrence is of below form then we directly give the answer without fully solving it.

If the recurrence is of the form  $T(n) = aT\left(\frac{n}{b}\right) + \Theta(n^k \log^p n)$ , where  $a \geq 1$ ,  $b > 1$ ,  $k \geq 0$  and  $p$  is a real number, then we can directly give the answer as:

- 1) If  $a > b^k$ , then  $T(n) = \Theta(n^{\log_b a})$
- 2) If  $a = b^k$ 
  - a. If  $p > -1$ , then  $T(n) = \Theta(n^{\log_b a} \log^{p+1} n)$
  - b. If  $p = -1$ , then  $T(n) = \Theta(n^{\log_b a} \log \log n)$
  - c. If  $p < -1$ , then  $T(n) = \Theta(n^{\log_b a})$
- 3) If  $a < b^k$ 
  - a. If  $p \geq 0$ , then  $T(n) = \Theta(n^k \log^p n)$
  - b. If  $p < 0$ , then  $T(n) = O(n^k)$

#### Applications of Divide and conquer rule or algorithm:

- Binary search,
- Quick sort,
- Merge sort,
- Strassen's matrix multiplication.

#### Binary search or Half-interval search algorithm:

1. This algorithm finds the position of a specified input value (the search "key") within an array sorted by key value.
2. In each step, the algorithm compares the search key value with the key value of the middle element of the array.
3. If the keys match, then a matching element has been found and its index, or position, is returned.
4. Otherwise, if the search key is less than the middle element's key, then the algorithm repeats its action on the sub-array to the **left** of the middle element or, if the search key is greater, then the algorithm repeats on sub array to the **right** of the middle element.
5. If the search element is less than the minimum position element or greater than the maximum position element then this algorithm returns not found.

```

// A recursive binary search function. It returns
// location of x in given array arr[l..r] is present,
// otherwise -1
int binarySearch(int arr[], int l, int r, int x)
{
    if (r >= l) {
        int mid = l + (r - l) / 2;

        // If the element is present at the middle
        // itself
        if (arr[mid] == x)
            return mid;

        // If element is smaller than mid, then
        // it can only be present in left subarray
        if (arr[mid] > x)
            return binarySearch(arr, l, mid - 1, x);

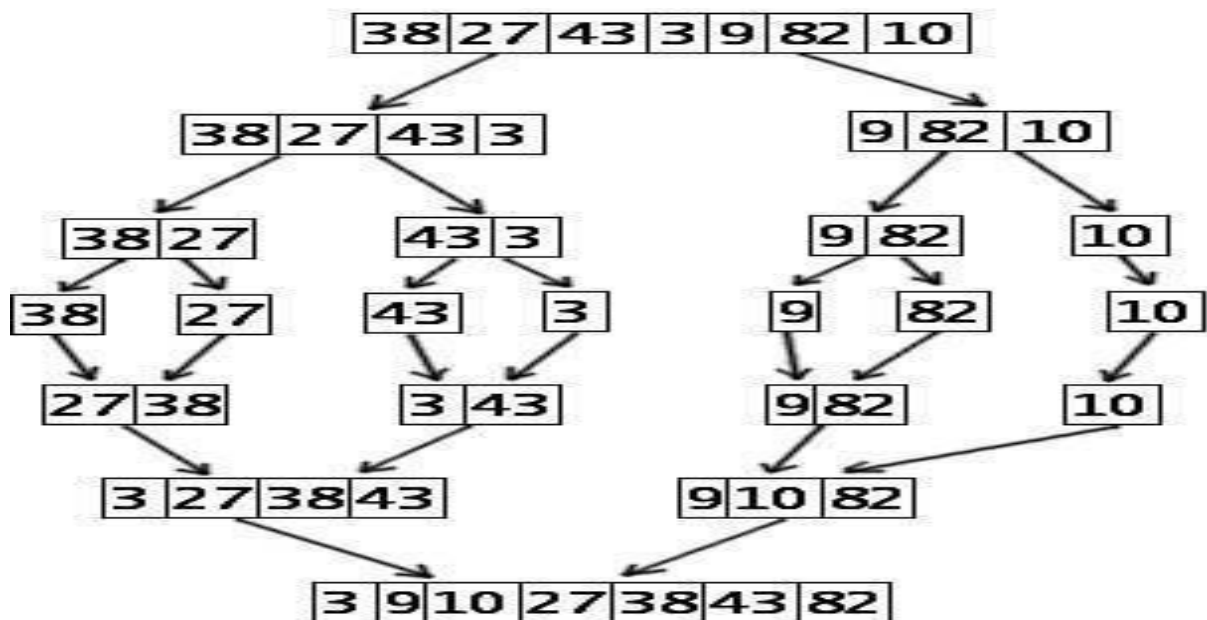
        // Else the element can only be present
        // in right subarray
        return binarySearch(arr, mid + 1, r, x);
    }

    // We reach here when element is not
    // present in array
    return -1;
}

```

### Merge Sort:

The merge sort splits the list to be sorted into two equal halves, and places them in separate arrays. This sorting method is an example of the DIVIDE-AND-CONQUER paradigm i.e. it breaks the data into two halves and then sorts the two half data sets recursively, and finally merges them to obtain the complete sorted list. The merge sort is a comparison sort and has an algorithmic complexity of  $O(n \log n)$ . Elementary implementations of the merge sort make use of two arrays - one for each half of the data set. The following image depicts the complete procedure of merge sort.



#### Advantages of Merge Sort:

1. Marginally faster than the heap sort for larger sets
2. Merge Sort always does lesser number of comparisons than Quick Sort. Worst case for merge sort does about 39% less comparisons against quick sort's average case.
3. Merge sort is often the best choice for sorting a linked list because the slow random-access performance of a linked list makes some other algorithms (such as quick sort) perform poorly, and others (such as heap sort) completely impossible.

#### Program for Merge sort:

```

#include<stdio.h>
#include<conio.h>
int n;
void main(){
int i,low,high,z,y;
int a[10];
void mergesort(int a[10],int low,int high);
void display(int a[10]);
clrscr();
printf("\n \t\t mergesort \n");
printf("\n enter the length of the list:");
scanf("%d",&n);
printf("\n enter the list elements");
for(i=0;i<n;i++)
scanf("%d",&a[i]);
low=0;
high=n-1;
mergesort(a,low,high);
display(a);
getch();
}
void mergesort(int a[10],int low, int high)

```

```

{
int mid;
void combine(int a[10],int low, int mid, int high);
if(low<high)
{
mid=(low+high)/2;
mergesort(a,low,mid);
mergesort(a,mid+1,high);
combine(a,low,mid,high);
}
}
void combine(int a[10], int low, int mid, int high){
int i,j,k;
int temp[10];
k=low;
i=low;
j=mid+1;
while(i<=mid&& j<=high){
if(a[i]<=a[j])
{
temp[k]=a[i];
i++;
k++;
}
else
{
temp[k]=a[j];
j++;
k++;
}
}
while(i<=mid){
temp[k]=a[i];
i++;
k++;
}

while(j<=high){
temp[k]=a[j];
j++;
k++;
}
for(k=low;k<=high;k++)
a[k]=temp[k];
}
void display(int a[10]){
int i;
printf("\n \n the sorted array is \n");
for(i=0;i<n;i++)
printf("%d \t",a[i]);}

```

### Algorithm for Merge sort:

```
Algorithm mergesort(low, high)
{
  if(low<high) then      // Dividing Problem into Sub-problems and
  {                      this "mid" is for finding where to split the set.
    mid=(low+high)/2;

    mergesort(low,mid);
    mergesort(mid+1,high); //Solve the sub-problems
    Merge(low,mid,high);  // Combine the solution
  }
}

void Merge(low, mid,high){
  k=low;
  i=low;
  j=mid+1;
  while(i<=mid&&j<=high) do{
    if(a[i]<=a[j]) then
    {
      temp[k]=a[i];
      i++;
      k++;
    }
    else
    {
      temp[k]=a[j];
      j++;
      k++;
    }
  }
  while(i<=mid) do{
    temp[k]=a[i];
    i++;
    k++;
  }

  while(j<=high) do{
    temp[k]=a[j];
    j++;
    k++;
  }
  For k=low to high do
  a[k]=temp[k];
}
For k:=low to high do a[k]=temp[k];
}
```

### Computing Time for Merge sort:

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

The time for the merging operation is proportional to  $n$ , then computing time for merge sort is described by using recurrence relation.

Here  $c, a$  are Constants.

If  $n$  is power of 2,  $n=2^k$

Form recurrence relation

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

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

$$4T(n/4) + 2cn$$

$$2^2 T(n/4) + 2cn$$

$$2^3 T(n/8) + 3cn$$

$$2^4 T(n/16) + 4cn$$

$$2^k T(1) + kcn$$

$$an + cn(\log n)$$

By representing it by in the form of Asymptotic notation  $O$  is

$$T(n) = O(n \log n)$$

### Quick Sort

Quick Sort is an algorithm based on the DIVIDE-AND-CONQUER paradigm that selects a pivot element and reorders the given list in such a way that all elements smaller to it are on one side and those bigger than it are on the other. Then the sub lists are recursively sorted until the list gets completely sorted. The time complexity of this algorithm is  $O(n \log n)$ .

- Auxiliary space used in the average case for implementing recursive function calls is  $O(\log n)$  and hence proves to be a bit space costly, especially when it comes to large data sets.
- Its worst case has a time complexity of  $O(n^2)$  which can prove very fatal for large data sets. Competitive sorting algorithms

### Quick sort program

```
#include<stdio.h>
#include<conio.h>
int n,j,i;
void main(){
int i,low,high,z,y;
int a[10],kk;
void quick(int a[10],int low,int high);
int n;
clrscr();
printf("\n \t mergesort \n");
printf("\n enter the length of the list:");
scanf("%d",&n);
printf("\n enter the list elements");
for(i=0;i<n;i++)
scanf("%d",&a[i]);
low=0;
high=n-1;
quick(a,low,high);
printf("\n sorted array is:");
for(i=0;i<n;i++)
printf(" %d",a[i]);
getch();
}

int partition(int a[10], int low, int high){
int i=low,j=high;
int temp;
int mid=(low+high)/2;
int pivot=a[mid];
while(i<=j)
{
while(a[i]<=pivot)
i++;
```

```

while(a[j]>pivot)
j--;
if(i<=j){
    temp=a[i];
    a[i]=a[j];
    a[j]=temp;
    i++;
    j--;
}
return j;
}
void quick(int a[10],int low, int high)
{
int m=partition(a,low,high);
if(low<m)
quick(a,low,m);
if(m+1<high)
quick(a,m+1,high);
}

```

#### Algorithm for Quick sort

```

Algorithm quickSort (a, low, high) {
If(high>low) then{
m=partition(a,low,high);
if(low<m) then quick(a,low,m);
if(m+1<high) then quick(a,m+1,high);
}}

Algorithm partition(a, low, high){
i=low,j=high;
mid=(low+high)/2;
pivot=a[mid];
while(i<=j) do { while(a[i]<=pivot)
    i++;
    while(a[j]>pivot)
    j--;
    if(i<=j){ temp=a[i];
    a[i]=a[j];
    a[j]=temp;
    i++;
    j--;
}}
return j;
}

```

Name	Time Complexity			Space Complexity
	Best case	Average Case	Worst Case	
<b>Bubble</b>	O(n)	-	O(n <sup>2</sup> )	O(n)
<b>Insertion</b>	O(n)	O(n <sup>2</sup> )	O(n <sup>2</sup> )	O(n)
<b>Selection</b>	O(n <sup>2</sup> )	O(n <sup>2</sup> )	O(n <sup>2</sup> )	O(n)



<b>Quick</b>	$O(\log n)$	$O(n \log n)$	$O(n^2)$	$O(n + \log n)$
<b>Merge</b>	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(2n)$
<b>Heap</b>	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$

### Comparison between Merge and Quick Sort:

- Both follows Divide and Conquer rule.
- Statistically both merge sort and quick sort have the same average case time i.e.,  $O(n \log n)$ .
- Merge Sort Requires additional memory. The pros of merge sort are: it is a stable sort, and there is no worst case (means average case and worst case time complexity is same).
- Quick sort is often implemented in place thus saving the performance and memory by not creating extra storage space.
- But in Quick sort, the performance falls on already sorted/almost sorted list if the pivot is not randomized. Thus why the worst case time is  $O(n^2)$ .

### Randomized Sorting Algorithm: (Random quick sort)

- While sorting the array  $a[p:q]$  instead of picking  $a[m]$ , pick a random element (from among  $a[p]$ ,  $a[p+1]$ ,  $a[p+2]$ --- $a[q]$ ) as the partition elements.
- The resultant randomized algorithm works on any input and runs in an expected  $O(n \log n)$  times.

Algorithm for Random Quick sort
<pre> Algorithm RquickSort (a, p, q) {   If(high&gt;low) then{     If((q-p)&gt;5) then       Interchange(a, Random() mod (q-p+1)+p, p);       m=partition(a,p, q+1);       quick(a, p, m-1);       quick(a,m+1,q);     }} </pre>

# Strassen's Matrix Multiplication

## Divide and Conquer

Following is simple Divide and Conquer method to multiply two square matrices.

- 1) Divide matrices A and B in 4 sub-matrices of size  $N/2 \times N/2$  as shown in the below diagram.
- 2) Calculate following values recursively.  $ae + bg$ ,  $af + bh$ ,  $ce + dg$  and  $cf + dh$ .

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} ae + bg & af + bh \\ ce + dg & cf + dh \end{bmatrix}$$

A                      B                      C

A, B and C are square matrices of size  $N \times N$   
 $a, b, c$  and  $d$  are submatrices of A, of size  $N/2 \times N/2$   
 $e, f, g$  and  $h$  are submatrices of B, of size  $N/2 \times N/2$

In the above method, we do 8 multiplications for matrices of size  $N/2 \times N/2$  and 4 additions. Addition of two matrices takes  $O(N^2)$  time. So the time complexity can be written as

$$T(N) = 8T(N/2) + O(N^2)$$

From [Master's Theorem](#), time complexity of above method is  $O(N^3)$  which is unfortunately same as the above naive method.

## Simple Divide and Conquer also leads to $O(N^3)$ , can there be a better way?

In the above divide and conquer method, the main component for high time complexity is 8 recursive calls. The idea of **Strassen's method** is to reduce the number of recursive calls to 7. Strassen's method is similar to above simple divide and conquer method in the sense that this method also divide matrices to sub-matrices of size  $N/2 \times N/2$  as shown in the above diagram, but in Strassen's method, the four sub-matrices of result are calculated using following formulae.

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} p5 + p4 - p2 + p6 & p1 + p2 \\ p3 + p4 & p1 + p5 - p3 - p7 \end{bmatrix}$$

A                      B                      C

A, B and C are square matrices of size  $N \times N$   
 $a, b, c$  and  $d$  are submatrices of A, of size  $N/2 \times N/2$   
 $e, f, g$  and  $h$  are submatrices of B, of size  $N/2 \times N/2$   
 $p1, p2, p3, p4, p5, p6$  and  $p7$  are submatrices of size  $N/2 \times N/2$

$p1 = a(f - h)$   
 $p3 = (c + d)e$   
 $p5 = (a + d)(e + h)$   
 $p7 = (a - c)(e + f)$

$p2 = (a + b)h$   
 $p4 = d(g - e)$   
 $p6 = (b - d)(g + h)$

The  $A \times B$  can be calculated using above seven multiplications.  
 Following are values of four sub-matrices of result C

## UNIT- II:

Disjoint set operations, Union and Find algorithms, AND/OR graphs, Connected components, Bi-connected components. Greedy method: General method, applications- Job sequencing with deadlines, Knapsack problem, Spanning trees, Minimum cost spanning trees, Single source shortest path problem.

**Disjoint Sets:** If  $S_i$  and  $S_j$ ,  $i \neq j$  are two sets, then there is no element that is in both  $S_i$  and  $S_j$ .

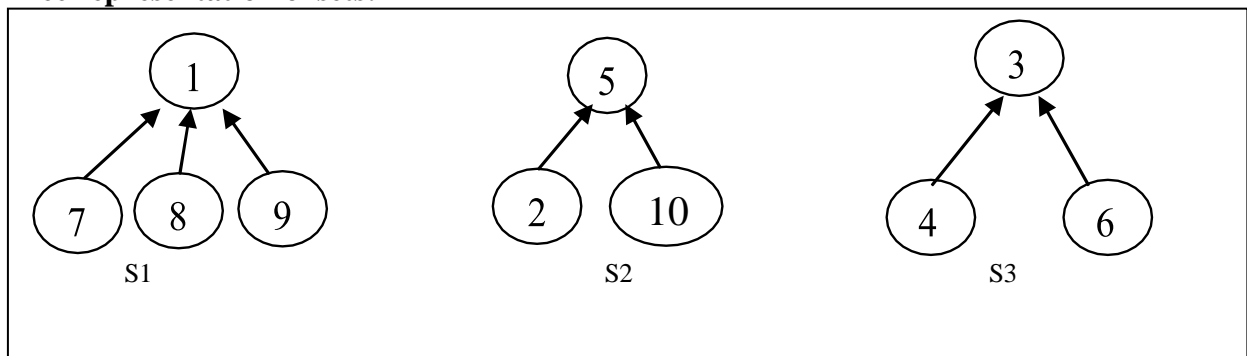
For example:  $n=10$  elements can be partitioned into three disjoint sets,

$S_1 = \{1, 7, 8, 9\}$

$S_2 = \{2, 5, 10\}$

$S_3 = \{3, 4, 6\}$

**Tree representation of sets:**

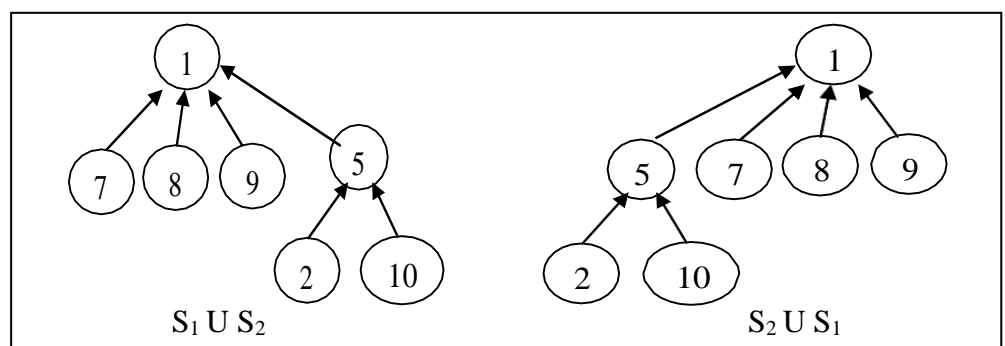


**Disjoint set Operations:**

- ☐ Disjoint set Union
- ☐ Find(i)

**Disjoint set Union:** Means Combination of two disjoint sets elements. Form above example  $S_1 \cup S_2 = \{1, 7, 8, 9, 5, 2, 10\}$

For  $S_1 \cup S_2$  tree representation, simply make one of the tree is a subtree of the other.



**Find:** Given element  $i$ , find the set containing  $i$ .

Form above example:

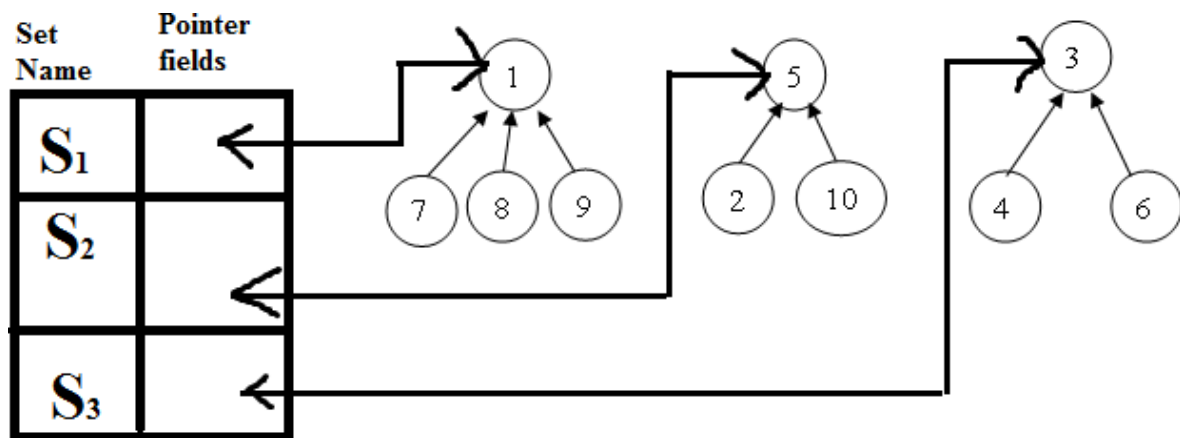
Find(4)  $\rightarrow$   $S_3$

Find(1)  $\rightarrow$   $S_1$

Find(10)  $\rightarrow$   $S_2$

## Data representation of sets:

Tress can be accomplished easily if, with each set name, we keep a pointer to the root of the tree representing that set.



For presenting the union and find algorithms, we ignore the set names and identify sets just by the roots of the trees representing them.

**For example:** if we determine that element 'i' is in a tree with root 'j' has a pointer to entry 'k' in the set name table, then the set name is just **name[k]**

For unite (**adding or combine**) to a particular set we use FindPointer function.

**Example:** If you wish to unite to  $S_i$  and  $S_j$  then we wish to unite the tree with roots FindPointer ( $S_i$ ) and FindPointer ( $S_j$ )

FindPointer is a function that takes a set name and determines the root of the tree that represents it.

For determining operations:

Find(i) □ 1<sup>st</sup> determine the root of the tree and find its pointer to entry in setname table.

Union(i, j) □ Means union of two trees whose roots are i and j.

If set contains numbers 1 through n, we represents tree node P[1:n]. n Maximum number of elements

Each node represent in array

i	1	2	3	4	5	6	7	8	9	10
P	-1	5	-1	3	-1	3	1	1	1	5

find(i) by following the indices, starting at i until we reach a node with parent  
 Example: Find(6) start at 6 and then moves to 6's parent. Since P[3] is negative, we reached the root.

Algorithm for finding Union(i, j):	Algorithm for find(i)
Algorithm Simple union(i, j) { P[i]:=j; // Accomplishes the union }	Algorithm SimpleFind(i) { While(P[i]≥0) do i:=P[i]; return i; }

If n numbers of roots are there then the above algorithms are not useful for union and find.

For union of n trees □                      Union(1,2), Union(2,3), Union(3,4),.....Union(n-1,n).

For Find i in n trees □                      Find(1), Find(2),....Find(n).

Time taken for the union (simple union) is  $O(1)$  (constant). For the n-1 unions  $O(n)$ .

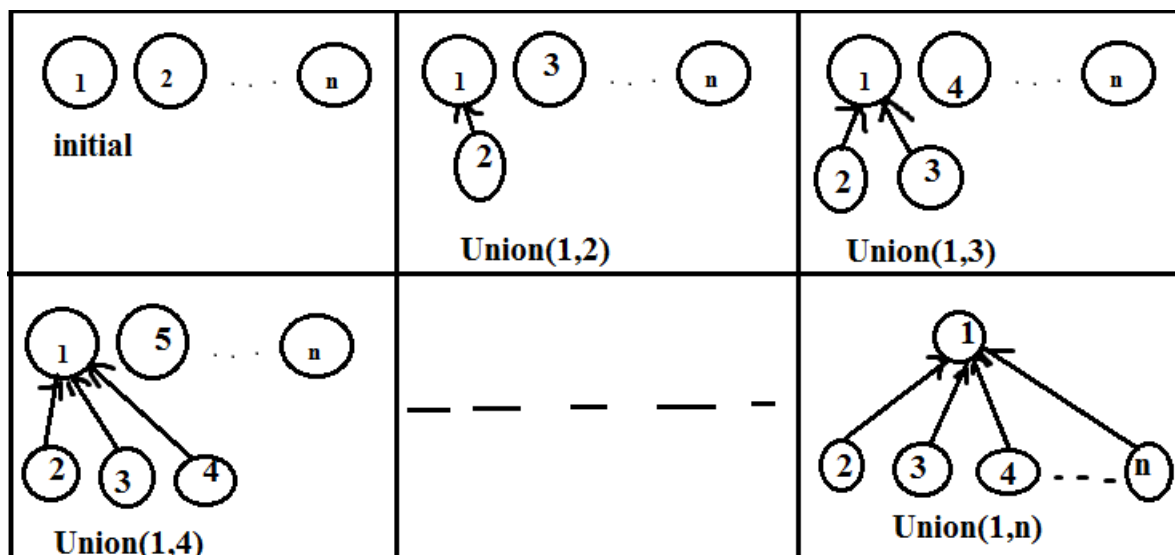
Time taken for the find for an element at level i of a tree is □                       $O(i)$ .

For n finds □                       $O(n^2)$ .

To improve the performance of our union and find algorithms by avoiding the creation of degenerate trees. For this we use a weighting rule for union(i, j)

### Weighting rule for Union(i, j):

If the number of nodes in the tree with root 'i' is less than the tree with root 'j', then make 'j' the parent of 'i'; otherwise make 'i' the parent of 'j'.



**Tree obtained using the weighting rule**

### Algorithm for weightedUnion(i, j)

```
Algorithm WeightedUnion(i,j)
//Union sets with roots i and j, i≠j
// The weighting rule, p[i]= -count[i] and p[j]= -count[j].
{
temp := p[i]+p[j];
if (p[i]>p[j]) then
{ // i has fewer nodes.
P[i]:=j;
P[j]:=temp;
}
else
{ // j has fewer or equal nodes.
P[j] := i;
P[i] := temp;
}
}
```

For implementing the weighting rule, we need to know how many nodes there are in every tree.

For this we maintain a count field in the root of every tree. i□

root node

count[i]□ number of nodes in the tree.

Time required for this above algorithm is  $O(1)$  + time for remaining unchanged is determined by using **Lemma**.

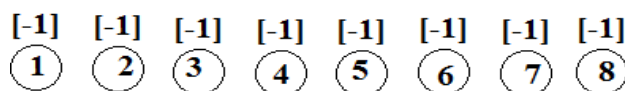
**Lemma:**-Let T be a tree with **m** nodes created as a result of a sequence of unions each performed using Weighted Union. The height of T is no greater than  $\lceil \log_2 m \rceil + 1$ .

**Collapsing rule:** If 'j' is a node on the path from 'i' to its root and  $p[i] \neq \text{root}[i]$ , then set  $p[j]$  to  $\text{root}[i]$ .

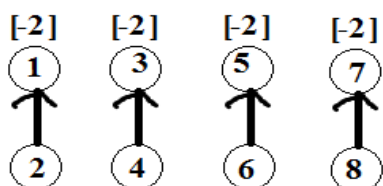
Algorithm for Collapsing find.
<pre> Algorithm CollapsingFind(i) //Find the root of the tree containing element i. //collapsing rule to collapse all nodes from i to the root. {   r:=i;   while(p[r]&gt;0) do r := p[r]; //Find the root.   While(i ≠ r) do // Collapse nodes from i to root r.   {     s:=p[i];     p[i]:=r;     i:=s;   }   return r; } </pre>

Collapsing find algorithm is used to perform find operation on the tree created by Weighted Union.

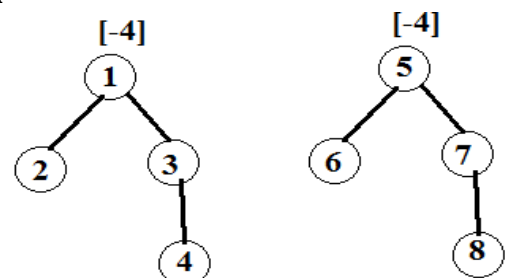
**For example: Tree created by using Weighted Union**



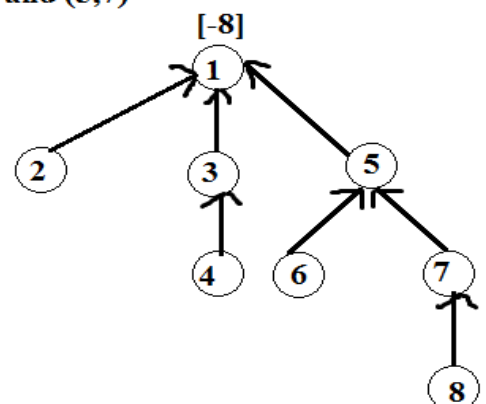
(a) initial height -1 tree



(b) Height -2 trees following Union(1,2),(3,4),(5,6),(7,8)



(c) Height -3 trees following Union (1,3) and (5,7)



(d) Height -4 tree Following Union(1,5)

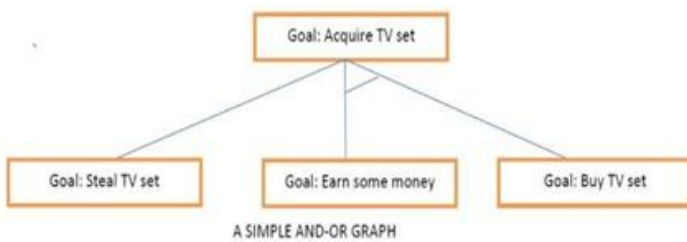
Now process the following eight finds: Find(8), Find(8), ..... Find(8)

If SimpleFind is used, each Find(8) requires going up three parent link fields for a total of 24 moves to process all eight finds. When CollapsingFind is used the first Find(8) requires going up three links and then resetting two links. Total 13 moves requires for process all eight finds.

## AND-OR GRAPHS

The AND-OR GRAPH (or tree) is useful for representing the solution of problems that can be solved by decomposing them into a set of smaller problems, all of which must then be solved. This decomposition, or reduction, generates arcs that we call AND arcs. One AND arc may point to any number of successor nodes, all of which must be solved in order for the arc to point to a solution. Just as in an OR graph, several arcs may emerge from a single node, indicating a variety of ways in which the original problem might be solved. This is why the structure is called not simply an AND-graph but rather an AND-OR graph (which also happens to be an AND-OR tree)

### EXAMPLE FOR AND-OR GRAPH



### ALGORITHM:

1. Let  $G$  be a graph with only starting node  $INIT$ .
2. Repeat the followings until  $INIT$  is labeled SOLVED or  $h(INIT) > FUTILITY$ 
  - a) Select an unexpanded node from the most promising path from  $INIT$  (call it  $NODE$ )
  - b) Generate successors of  $NODE$ . If there are none, set  $h(NODE) = FUTILITY$  (i.e.,  $NODE$  is unsolvable); otherwise for each  $SUCCESSOR$  that is not an ancestor of  $NODE$  do the following:
    - i. Add  $SUCCESSOR$  to  $G$ .
    - ii. If  $SUCCESSOR$  is a terminal node, label it SOLVED and set  $h(SUCCESSOR) = 0$ .
    - iii. If  $SUCCESSOR$  is not a terminal node, compute its  $h$
  - c) Propagate the newly discovered information up the graph by doing the following: let  $S$  be set of SOLVED nodes or nodes whose  $h$  values have been changed and need to have values propagated back to their parents. Initialize  $S$  to  $Node$ . Until  $S$  is empty repeat the followings:
    - i. Remove a node from  $S$  and call it  $CURRENT$ .
    - ii. Compute the cost of each of the arcs emerging from  $CURRENT$ . Assign minimum cost of its successors as its  $h$ .
    - iii. Mark the best path out of  $CURRENT$  by marking the arc that had the minimum cost in step ii
    - iv. Mark  $CURRENT$  as SOLVED if all of the nodes connected to it through new labeled arc have been labeled SOLVED
    - v. If  $CURRENT$  has been labeled SOLVED or its cost was just changed, propagate its new cost back up through the graph So add all of the ancestors of  $CURRENT$  to  $S$ .

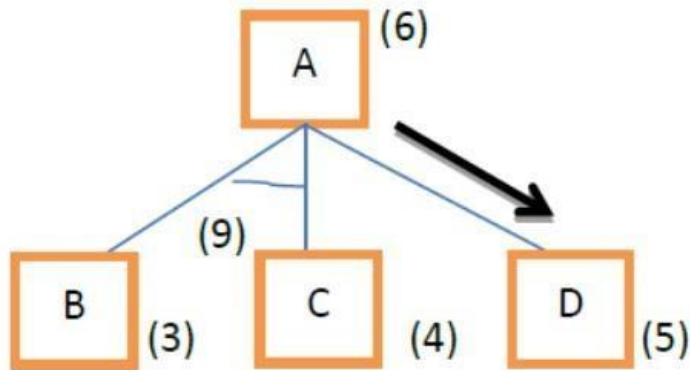


### EXAMPLE: 1

#### STEP 1:

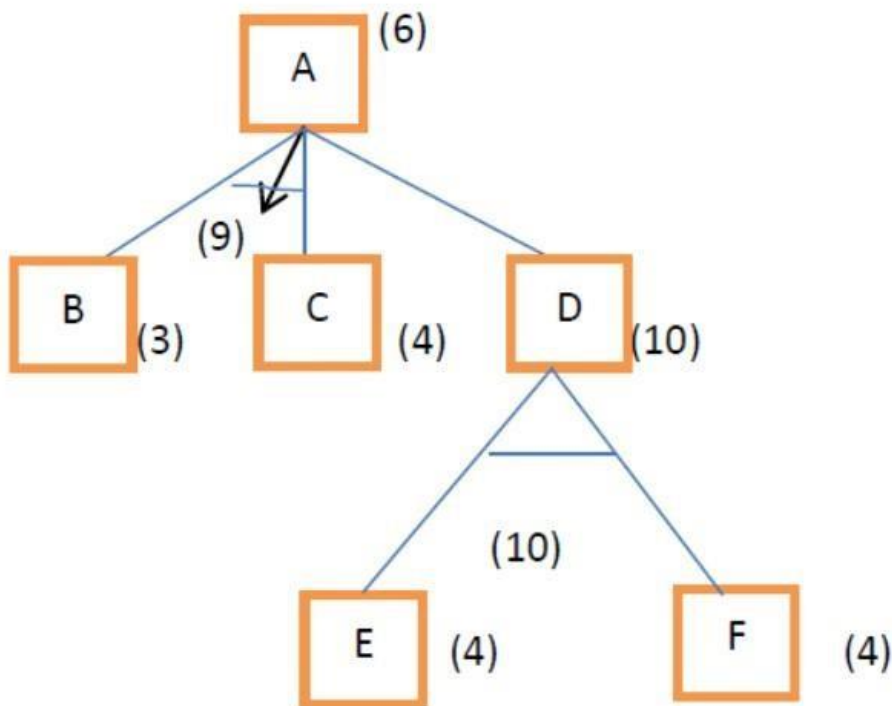
A is the only node, it is at the end of the current best path. It is expanded, yielding nodes B, C, D. The arc to D is labeled as the most promising one emerging from A, since it costs 6 compared to B and C, Which costs 9.

#### STEP 2:



Node B is chosen for expansion. This process produces one new arc, the AND arc to E and F, with a combined cost estimate of 10. so we update the  $f^*$  value of D to 10. Going back one more level, we see that this makes the AND arc B-C better than the arc to D, so it is labeled as the current best path.

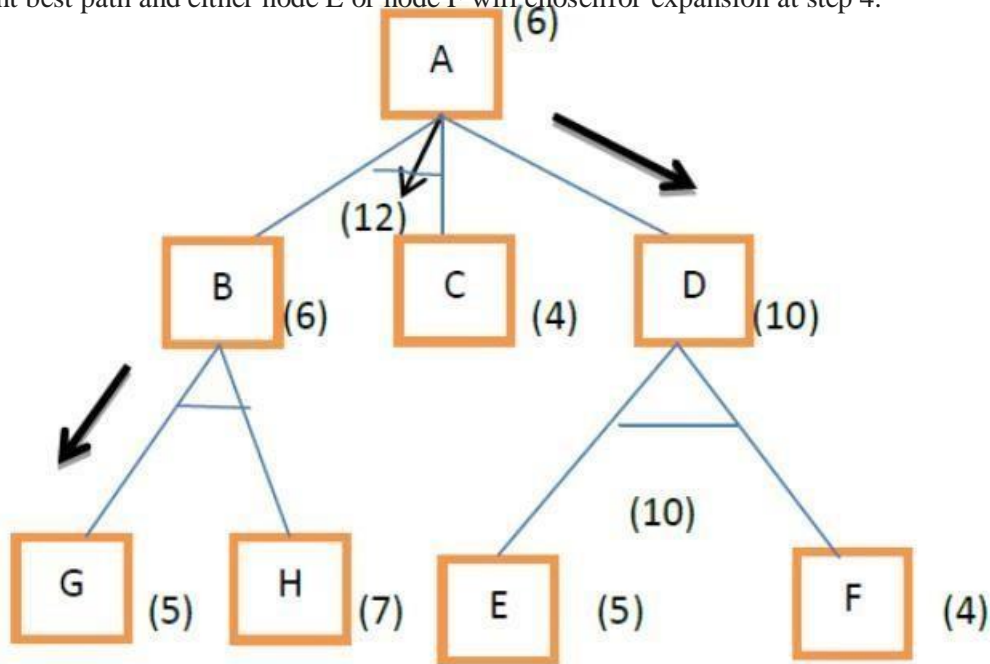
#### STEP 3:



we traverse the arc from A and discover the unexpanded nodes B and C. If we going to find a solution along this path, we will have to expand both B and C eventually, so let's choose to

explore B first. This generates two new arcs, the ones to G and to H. Propagating their  $f^*$  values backward, we update  $f^*$  of B to 6 (since that is the best we think we can do, which we can achieve by going through G). This requires updating the cost of the AND arc B-C to  $12(6+4+2)$ . After doing that, the arc to D is again the better path from A, so we record that as the current best path and either node E or node F will chosen for expansion at step 4.

STEP4:



### Connected Component:

Connected component of a graph can be obtained by using BFST (Breadth first search and traversal) and DFST (Dept first search and traversal). It is also called the spanning tree.

### BFST (Breadth first search and traversal):

- ☐ In BFS we start at a vertex V mark it as reached (visited).
- ☐ The vertex V is at this time said to be unexplored (not yet discovered).
- ☐ A vertex is said to been explored (discovered) by visiting all vertices adjacent from it. All unvisited vertices adjacent from V are visited next.
- ☐ The first vertex on this list is the next to be explored. Exploration continues until no unexplored vertex is left. These operations can be performed by using Queue.

**Algorithm for BFS to convert undirected graph G to Connected component or spanningtree.**

Algorithm BFS(v)

// a bfs of G is begin at vertex v

// for any node I, visited[i]=1 if I has already been visited.

// the graph G, and array visited[] are global

{

U:=v; // q is a queue of unexplored vertices.

Visited[v]:=1;

Repeat{

For all vertices w adjacent from U doIf

(visited[w]=0) then

{

Add w to q; // w is unexplored

Visited[w]:=1;

}

If q is empty then return; // No unexplored vertex.

Delete U from q; //Get 1<sup>st</sup> unexplored vertex.

} Until(false)

}

Maximum Time complexity and space complexity of G(n,e), nodes are in adjacency list.

$T(n, e) = \theta(n+e)$

$S(n, e) = \theta(n)$

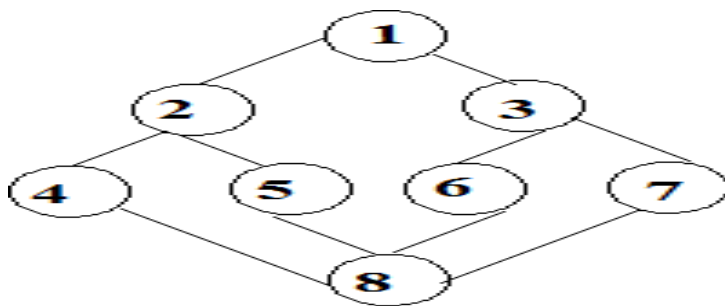
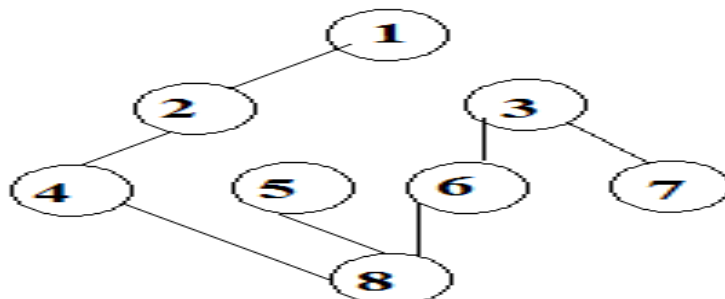
If nodes are in adjacency matrix then

$T(n, e) = \theta(n^2)$

$S(n, e) = \theta(n)$

**DFST(Dept first search and traversal).:**

- ☐ Dfs different from bfs
- ☐ The exploration of a vertex  $v$  is suspended (stopped) as soon as a new vertex is reached.
- ☐ In this the exploration of the new vertex (example  $v$ ) begins; this new vertex has been explored, the exploration of  $v$  continues.
- ☐ Note: exploration start at the new vertex which is not visited in other vertex exploring and choose nearest path for exploring next or adjacent vertex.

**Undirected Graph G****DFS(1) Spanning tree**

### Algorithm for DFS to convert undirected graph G to Connected component or spanningtree.

```
Algorithm dFS(v)
// a Dfs of G is begin at vertex v
// initially an array visited[] is set to zero.
//this algorithm visits all vertices reachable from v.
// the graph G, and array visited[] are global
{
  Visited[v]:=1;
  For each vertex w adjacent from v do
  {
    If (visited[w]=0) then DFS(w);
  }
  Add w to q; // w is unexplored
  Visited[w]:=1;
}
```

Maximum Time complexity and space complexity of  $G(n,e)$ , nodes are in adjacency list.

$T(n, e)=\theta(n+e)$

$S(n, e)=\theta(n)$

If nodes are in adjacency matrix then

$T(n, e)=\theta(n^2)$

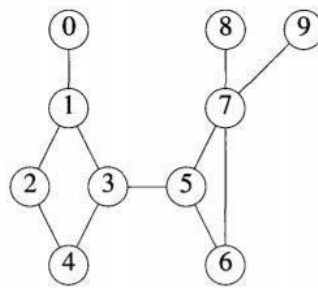
$S(n, e)=\theta(n)$

A **biconnected component** of  $G$  is a maximal set of edges such that any two edges in the set lie on a common simple cycle

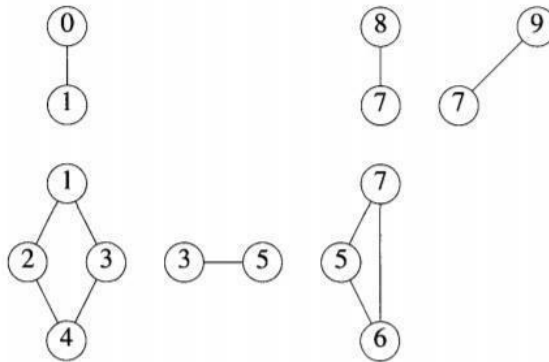
A *biconnected component* of a connected undirected graph is a *maximal biconnected subgraph*,  $H$ , of  $G$ . By maximal, we mean that  $G$  contains no other subgraph that is both biconnected and properly contains  $H$ . For example, the graph of Figure (a) contains the six biconnected components shown in Figure (b). The biconnected graph of Figure however, contains just one biconnected component: the whole graph. It is easy to verify that two biconnected components of the same graph have no more than one vertex in common. This means that no edge can be in two or more biconnected components of a graph. Hence, the biconnected components of  $G$  partition the edges of  $G$ .

We can find the biconnected components of a connected undirected graph,  $G$ , by using any depth first spanning tree of  $G$ . For example, the function call  $dfs(3)$  applied to the graph of Figure (a) produces the spanning tree of Figure (a). We have redrawn the tree in Figure (b) to better reveal its tree structure. The numbers outside the vertices in either figure give the sequence in which the vertices are visited during the depth first search. We call this number the *depth first number*, or *dfn*, of the vertex. For example,  $dfn(3) = 0$ ,  $dfn(0) = 4$ , and  $dfn(9) = 8$ . Notice that vertex 3, which is an ancestor of both vertices 0 and 9, has a lower *dfn* than either of these vertices. Generally, if  $u$  and  $v$  are two vertices, and  $u$  is an ancestor of  $v$  in the depth first spanning tree, then  $dfn(u) < dfn(v)$ .

The broken lines in Figure (b) represent nontree edges. A nontree edge  $(u, v)$  is a *back edge* iff either  $u$  is an ancestor of  $v$  or  $v$  is an ancestor of  $u$ . From the definition of depth first search, it follows that all nontree edges are back edges. This means that the root of a depth first spanning tree is an articulation point iff it has at least two children. In addition, any other vertex  $u$  is an articulation point iff it has at least one child  $w$  such



(a) Connected graph



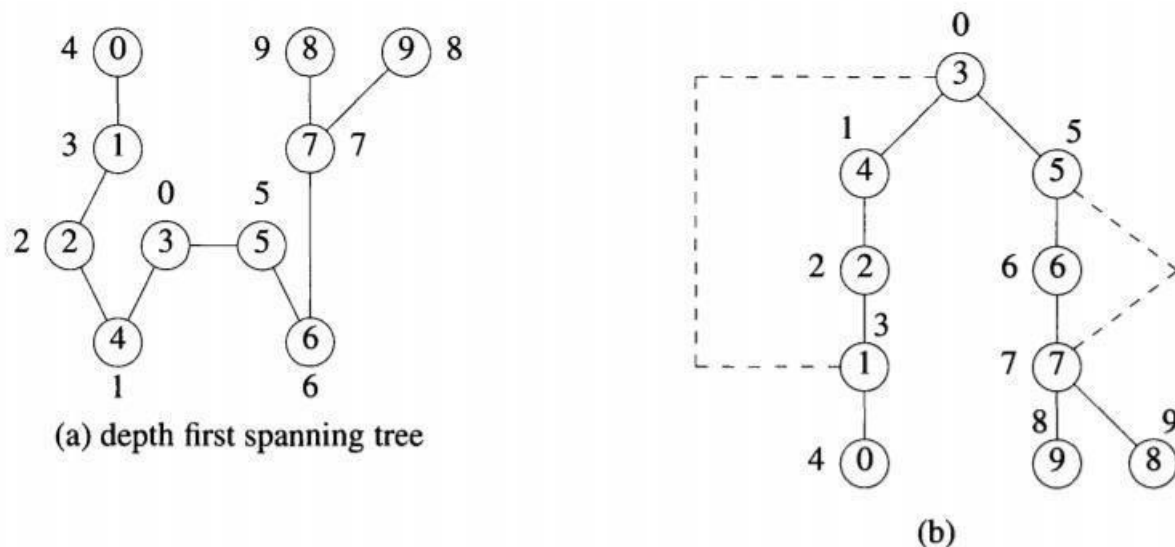
(b) Biconnected components

**Figure** A connected graph and its biconnected components

that we cannot reach an ancestor of  $u$  using a path that consists of only  $w$ , descendants of  $w$ , and a single back edge. These observations lead us to define a value,  $low$ , for each vertex of  $G$  such that  $low(u)$  is the lowest depth first number that we can reach from  $u$  using a path of descendants followed by at most one back edge:

$$low(u) = \min\{dfn(u), \min\{low(w) \mid w \text{ is a child of } u\}, \min\{dfn(w) \mid (u, w) \text{ is a back edge}\}\}$$

Therefore, we can say that  $u$  is an articulation point *iff*  $u$  is either the root of the spanning tree and has two or more children, or  $u$  is not the root and  $u$  has a child  $w$  such that  $low(w) \geq dfn(u)$ . Figure shows the  $dfn$  and  $low$  values for each vertex of the spanning tree of Figure (b). From this table we can conclude that vertex 1 is an



**Figure 6.20:** Depth first spanning tree of Figure 6.19(a)

articulation point since it has a child 0 such that  $low(0) = 4 \geq dfn(1) = 3$ . Vertex 7 is also an articulation point since  $low(8) = 9 \geq dfn(7) = 7$ , as is vertex 5 since  $low(6) = 5 \geq dfn(5) = 5$ . Finally, we note that the root, vertex 3, is an articulation point because it has more than one child.

Vertex	0	1	2	3	4	5	6	7	8	9
<i>dfn</i>	4	3	2	0	1	5	6	7	9	8
<i>low</i>	4	3	0	0	0	5	5	7	9	8

**Figure** : *dfn* and *low* values for *dfs* spanning tree with *root* = 3

### Greedy Method:

The greedy method is perhaps (maybe or possible) the most straight forward design technique, used to determine a feasible solution that may or may not be optimal.

**Feasible solution:-** Most problems have  $n$  inputs and its solution contains a subset of inputs that satisfies a given constraint(condition). Any subset that satisfies the constraint is called feasible solution.

**Optimal solution:** To find a feasible solution that either maximizes or minimizes a given objective function. A feasible solution that does this is called optimal solution.

The greedy method suggests that an algorithm works in stages, considering one input at a time. At each stage, a decision is made regarding whether a particular input is in an optimal solution.

Greedy algorithms neither postpone nor revise the decisions (ie., no back tracking).

**Example:** Kruskal's minimal spanning tree. Select an edge from a sorted list, check, decide, and never visit it again.

**Application of Greedy Method:**

- ☐ Job sequencing with deadline
- ☐ 0/1 knapsack problem
- ☐ Minimum cost spanning trees
- ☐ Single source shortest path problem.

Algorithm for Greedy method
<pre>Algorithm Greedy(a,n) //a[1:n] contains the n inputs. { Solution :=0; For i=1 to n do { X:=select(a); If Feasible(solution, x) then Solution :=Union(solution,x); } Return solution; }</pre>

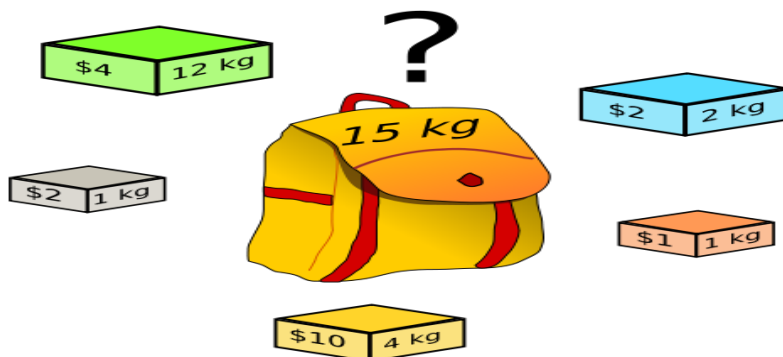
Selection ☐ Function, that selects an input from a[] and removes it. The selected input's value is assigned to x.

Feasible ☐ Boolean-valued function that determines whether x can be included into the solution vector.

Union ☐ function that combines x with solution and updates the objective function.

## Knapsack problem

The **knapsack problem** or **rucksack (bag) problem** is a problem in combinatorial optimization: Given a set of items, each with a mass and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible



### There are two versions of the problems

1. 0/1 knapsack problem
2. Fractional Knapsack problem
  - a. Bounded Knapsack problem.
  - b. Unbounded Knapsack problem.



## Solutions to knapsack problems

- ❑ **Brute-force approach**:- Solve the problem with a straight forward algorithm
- ❑ **Greedy Algorithm**:- Keep taking most valuable items until maximum weight is reached or taking the largest value of each item by calculating  $v_i = \text{value}_i / \text{Size}_i$
- ❑ **Dynamic Programming**:- Solve each sub problem once and store their solutions in an array.

### 0/1 knapsack problem:

Let there be  $n$  items,  $z_1$  to  $z_n$  where  $z_i$  has a value  $v_i$  and weight  $w_i$ . The maximum weight that we can carry in the bag is  $W$ . It is common to assume that all values and weights are nonnegative. To simplify the representation, we also assume that the items are listed in increasing order of weight.

$$\text{Maximize } \sum_{i=1}^n v_i x_i \quad \text{subject to } \sum_{i=1}^n w_i x_i \leq W, \quad x_i \in \{0, 1\}$$

Maximize the sum of the values of the items in the knapsack so that the sum of the weights must be less than the knapsack's capacity.

#### Greedy algorithm for knapsack

```
Algorithm GreedyKnapsack(m,n)
// p[1:n] and [1:n] contain the profits and weights respectively
// if the n-objects ordered such that  $p[i]/w[i] \geq p[i+1]/w[i+1]$ , m size of knapsack and
x[1:n] the solution vector
{
  For i:=1 to n do x[i]:=0.0
  U:=m;
  For i:=1 to n do
  {
    if(w[i]>U) then break;
    x[i]:=1.0;
    U:=U-w[i];
  }
  If(i<=n) then x[i]:=U/w[i];
}
```

**Ex:** - Consider 3 objects whose profits and weights are defined as  $(P_1, P_2, P_3) = (25, 24, 15)$   
 $(W_1, W_2, W_3) = (18, 15, 10)$

$n=3$  number of objects

$m=20$  Bag capacity

Consider a knapsack of capacity 20. Determine the optimum strategy for placing the objects in to the knapsack. The problem can be solved by the greedy approach where in the inputs are arranged according to selection process (greedy strategy) and solve the problem in stages. The various greedy strategies for the problem could be as follows.

$(x_1, x_2, x_3)$	$\sum x_i w_i$	$\sum x_i p_i$
$(1, 2/15, 0)$	$18x_1 + \frac{2}{15}x_{15} = 20$	$25x_1 + \frac{2}{15}x_{24} = 28.2$
$(0, 2/3, 1)$	$\frac{2}{3}x_{15} + 10x_1 = 20$	$\frac{2}{3}x_{24} + 15x_1 = 31$

$(0, 1, \frac{1}{2})$	$1x_{15} + \frac{1}{2}x_{10} = 20$	$1x_{24} + \frac{1}{2}x_{15} = 31.5$
$(\frac{1}{2}, \frac{1}{3}, \frac{1}{4})$	$\frac{1}{2}x_{18} + \frac{1}{3}x_{15} + \frac{1}{4}x_{10} = 16.5$	$\frac{1}{2}x_{25} + \frac{1}{3}x_{24} + \frac{1}{4}x_{15} = 12.5 + 8 + 3.75 = 24.25$

**Analysis:** - If we do not consider the time considered for sorting the inputs then all of the three greedy strategies complexity will be  $O(n)$ .

### **Job Sequence with Deadline:**

There is a set of  $n$ -jobs. For any job  $i$ ,  $d_i$  is an integer deadline  $d_i \geq 0$  and profit  $P_i > 0$ , the profit  $P_i$  is earned iff the job is completed by its deadline.

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.

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  $J$  is the sum of the profits of the jobs in  $J$ , i.e.,  $\sum_{i \in J} P_i$

An optimal solution is a feasible solution with maximum value.

The problem involves identification of a subset of jobs which can be completed by its deadline. Therefore the problem suits the subset methodology and can be solved by the greedy method.

**Ex:** - Obtain the optimal sequence for the following jobs.

$$(P_1, P_2, P_3, P_4) = \begin{matrix} j_1 & j_2 & j_3 & j_4 \\ (100, 10, 15, 27) \end{matrix}$$

$$(d_1, d_2, d_3, d_4) = (2, 1, 2, 1)$$

$n=4$

Feasible solution	Processing sequence	Value
$j_1 j_2$ (1, 2)	(2,1)	100+10=110
(1,3)	(1,3) or (3,1)	100+15=115
(1,4)	(4,1)	100+27=127
(2,3)	(2,3)	10+15=25
(3,4)	(4,3)	15+27=42
(1)	(1)	100
(2)	(2)	10
(3)	(3)	15
(4)	(4)	27

In the example solution '3' is the optimal. In this solution only jobs 1&4 are processed and the value is 127. These jobs must be processed in the order  $j_4$  followed by  $j_1$ . the process of job 4 begins at time 0 and ends at time 1. And the processing of job 1 begins at time 1 and ends at time2. Therefore both the jobs are completed within their deadlines. The optimization measure for determining the next job to be selected in to the solution is according to the profit. The next job to include is that which increases  $\sum p_i$  the most, subject to the constraint that the resulting "j" is the feasible solution. Therefore the greedy strategy is to consider the jobs in decreasing order of profits.

The greedy algorithm is used to obtain an optimal solution.

We must formulate an optimization measure to determine how the next job is chosen.

```

algorithm js(d, j, n)
//d→ dead line, j→subset of jobs ,n→ total number of jobs
// d[i]≥1 1 ≤ i ≤ n are the dead lines,
// the jobs are ordered such that p[1]≥p[2]≥...≥p[n]
//j[i] is the ith job in the optimal solution 1 ≤ i ≤ k, k→ subset range
{
d[0]=j[0]=0;
j[1]=1;
k=1;
for i=2 to n do{
r=k;
while((d[j[r]]>d[i]) and [d[j[r]]≠r)) do
r=r-1;
if((d[j[r]]≤d[i]) and (d[i]> r)) then
{
for q:=k to (r+1) setp-1 do j[q+1]= j[q];
j[r+1]=i;
k=k+1;
}
}
return k;
}

```

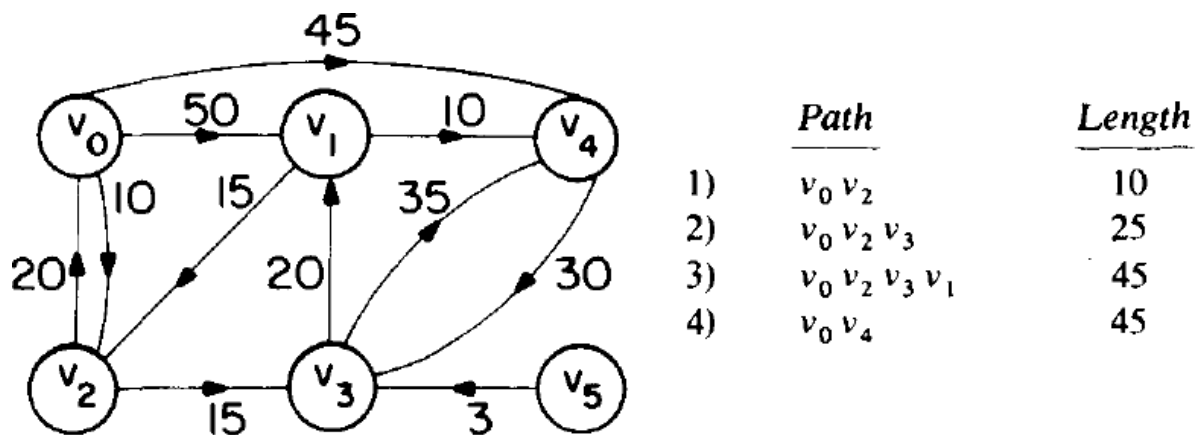
Note: The size of sub set j must be less than equal to maximum deadline in given list.

### **Single Source Shortest Paths:**

- Graphs can be used to represent the highway structure of a state or country with vertices representing cities and edges representing sections of highway.
- The edges have assigned weights which may be either the distance between the 2 cities connected by the edge or the average time to drive along that section of highway.
- For example if A motorist wishing to drive from city A to B then we must answer the following questions
  - Is there a path from A to B
  - If there is more than one path from A to B which is the shortest path
- The length of a path is defined to be the sum of the weights of the edges on that path.

Given a directed graph  $G(V,E)$  with weight edge  $w(u,v)$ . e have to find a shortest path from source vertex  $S \in v$  to every other vertex  $v1 \in V-S$ .

- To find SSSP for directed graphs  $G(V,E)$  there are two different algorithms.
  - Bellman-Ford Algorithm
  - Dijkstra's algorithm
- Bellman-Ford Algorithm:- allow -ve weight edges in input graph. This algorithm either finds a shortest path form source vertex  $S \in V$  to other vertex  $v \in V$  or detect a -ve weight cycles in  $G$ , hence no solution. If there is no negative weight cycles are reachable form source vertex  $S \in V$  to every other vertex  $v \in V$
- Dijkstra's algorithm:- allows only +ve weight edges in the input graph and finds a shortest path from source vertex  $S \in V$  to every other vertex  $v \in V$ .



### Graph and shortest paths from $v_0$ to all destinations

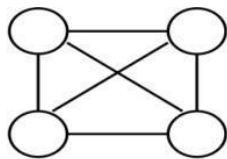
- Consider the above directed graph, if node 1 is the source vertex, then shortest path from 1 to 2 is 1,4,5,2. The length is  $10+15+20=45$ .
- To formulate a greedy based algorithm to generate the shortest paths, we must conceive of a multistage solution to the problem and also of an optimization measure.
- This is possible by building the shortest paths one by one.
- As an optimization measure we can use the sum of the lengths of all paths so far generated.
- If we have already constructed 'i' shortest paths, then using this optimization measure, the next path to be constructed should be the next shortest minimum length path.
- The greedy way to generate the shortest paths from  $V_0$  to the remaining vertices is to generate these paths in non-decreasing order of path length.
- For this 1<sup>st</sup>, a shortest path of the nearest vertex is generated. Then a shortest path to the 2<sup>nd</sup> nearest vertex is generated and so on.

### Algorithm for finding Shortest Path

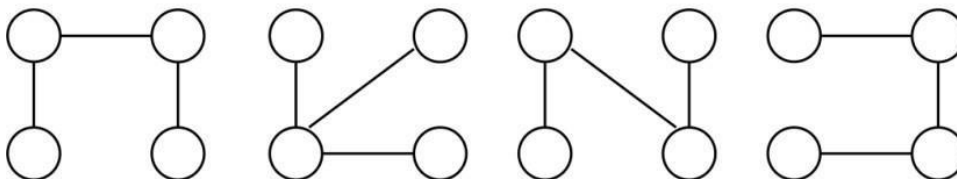
```
Algorithm ShortestPath(v, cost, dist, n)
//dist[j],  $1 \leq j \leq n$ , is set to the length of the shortest path from vertex v to vertex j in graph g
with n-vertices.
// dist[v] is zero
{
  for i=1 to n do{
    s[i]=false;
    dist[i]=cost[v,i];
  }
  s[v]=true;
  dist[v]:=0.0; // put v in s
  for num=2 to n do{
    // determine n-1 paths from v
    choose u from among those vertices not in s such that dist[u] is minimum.
    s[u]=true; // put u in s
    for (each w adjacent to u with s[w]=false) do
      if(dist[w]>(dist[u]+cost[u, w])) then
        dist[w]=dist[u]+cost[u, w];
  }
}
```

**SPANNING TREE**: - A Sub graph 'n' of o graph 'G' is called as a spanning tree if

- (i) It includes all the vertices of 'G'
- (ii) It is a tree



A connected,  
undirected graph

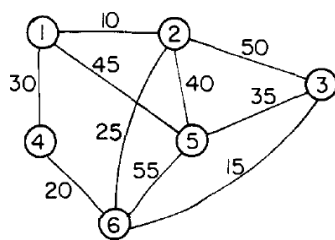


Four of the spanning trees of the graph

**Minimum cost spanning tree:** For a given graph 'G' there can be more than one spanning tree. If weights are assigned to the edges of 'G' then the spanning tree which has the minimum cost of edges is called as minimal spanning tree.

The greedy method suggests that a minimum cost spanning tree can be obtained by contacting the tree edge by edge. The next edge to be included in the tree is the edge that results in a minimum increase in the some of the costs of the edges included so far.

There are two basic algorithms for finding minimum-cost spanning trees, and both are greedy Algorithms



Edge	Cost	Spanning tree
(1,2)	10	
(2,6)	25	
(3,6)	15	
(6,4)	20	
(1,4)	reject	
(3,5)	35	

Stages in Prim's Algorithm

□

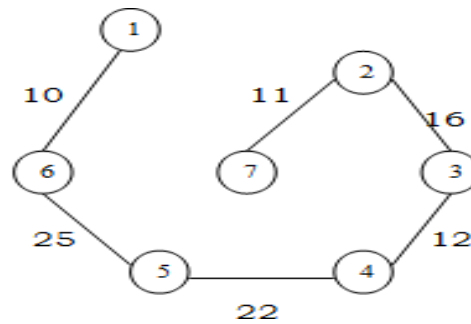
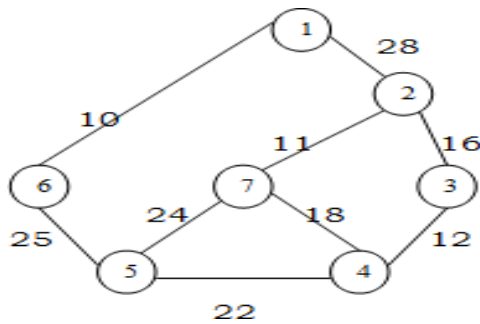
- Prim's Algorithm
- Kruskal's Algorithm

**Prim's Algorithm:** Start with any *one node* in the spanning tree, and repeatedly add the cheapest edge, and the node it leads to, for which the node is not already in the spanning tree.

### PRIM'S ALGORITHM: -

- i) Select an edge with minimum cost and include in to the spanning tree.
- ii) Among all the edges which are adjacent with the selected edge, select the onewithminimum cost.
- iii) Repeat step 2 until 'n' vertices and (n-1) edges are been included. And the subgraphobtained does not contain any cycles.

**Notes:** - At every state a decision is made about an edge of minimum cost to be included into thespanning tree. From the edges which are adjacent to the last edge included in the spanning tree i.e. at every stage the sub-graph obtained is a tree.



#### **Prim's minimum spanning tree algorithm**

```
Algorithm Prim (E, cost, n,t)
// E is the set of edges in G. Cost (1:n, 1:n) is the
// Cost adjacency matrix of an n vertex graph such that
// Cost (i,j) is either a positive real no. or  $\infty$  if no edge (i,j) exists.
//A minimum spanning tree is computed and
//Stored in the array T(1:n-1, 2).
//(t (i, 1), + t(i,2)) is an edge in the minimum cost spanning tree. The final cost is returned
{
    Let (k, l) be an edge with min cost
    in EMin cost: = Cost (x,l);
    T(1,1):= k; + (1,2):= l;
    for i:= 1 to n do//initialize
        near
        if (cost (i,l)<cost (i,k) then n east
        (i): l;else near (i): = k;
        near (k): = near (l): =
        0;for i: = 2 to n-1 do
    { //find n-2 additional edges for t
    let j be an index such that near (i)0 & cost (j, near (i)) is
    minimum;t(i,1): = j + (i,2): = near(j);
    min cost: = Min cost + cost (j, near
    (j));near (j): = 0;
    for k:=1 to n do // update near ()
    if ((near (k) 0) and (cost {k, near (k)) > cost
    (k,j)))then near Z(k): = ji
    }
    return mincost;
}
```

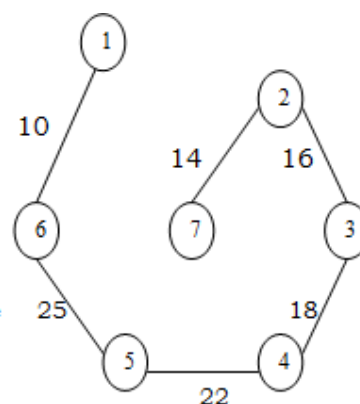
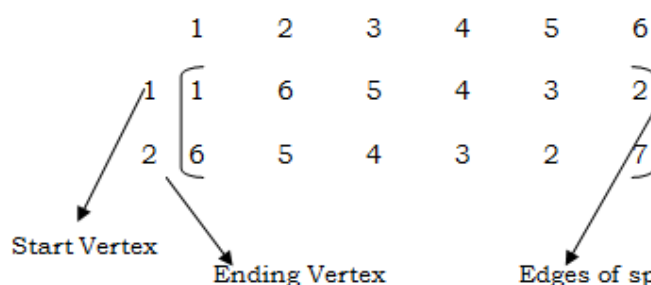
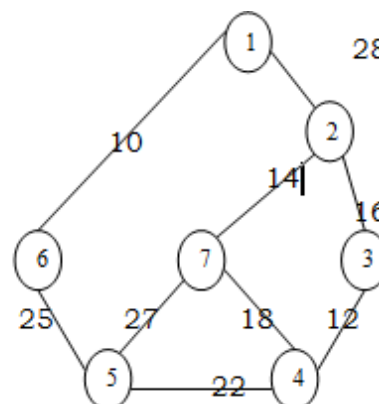


The algorithm takes four arguments E: set of edges, cost is nxn adjacency matrix cost of (i,j)= +ve integer, if an edge exists between i&j otherwise infinity. 'n' is no/: of vertices. 't' is a (n-1):2matrix which consists of the edges of spanning tree.

$E = \{ (1,2), (1,6), (2,3), (3,4), (4,5), (4,7), (5,6), (5,7), (2,7) \}$

$n = \{ 1,2,3,4,5,6,7 \}$

Cost	1	2	3	4	5	6	7
1	$\alpha$	28	$\alpha$	$\alpha$	$\alpha$	10	$\alpha$
2	28	$\alpha$	16	$\alpha$	$\alpha$	$\alpha$	14
3	$\alpha$	10	$\alpha$	12	$\alpha$	$\alpha$	$\alpha$
4	$\alpha$	$\alpha$	12	$\alpha$	22	$\alpha$	18
5	$\alpha$	$\alpha$	$\alpha$	22	$\alpha$	25	24
6	10	$\alpha$	$\alpha$	$\alpha$	25	$\alpha$	$\alpha$
7	$\alpha$	14	$\alpha$	18	24	$\alpha$	$\alpha$



- The algorithm will start with a tree that includes only minimum cost edge of G. Then edges are added to this tree one by one.
- The next edge (i,j) to be added is such that i is a vertex which is already included in the tree and j is a vertex not yet included in the tree and cost of i,j is minimum among all edges adjacent to 'i'.
- With each vertex 'j' next yet included in the tree, we assign a value near 'j'. The value near 'j' represents a vertex in the tree such that cost (j, near (j)) is minimum among all choices for near (j)
- We define near (j):= 0 for all the vertices 'j' that are already in the tree.
- The next edge to include is defined by the vertex 'j' such that  $\min(\text{near}(j)) = 0$  and cost of (j, near (j)) is minimum.

#### Analysis: -

The time required by the prince algorithm is directly proportional to the no/: of vertices. If a graph 'G' has 'n' vertices then the time required by prim's algorithm is  $O(n^2)$

**Kruskal's Algorithm:** Start with *no* nodes or edges in the spanning tree, and repeatedly add the cheapest edge that does not create a cycle.

In Kruskal's algorithm for determining the spanning tree we arrange the edges in the increasing order of cost.

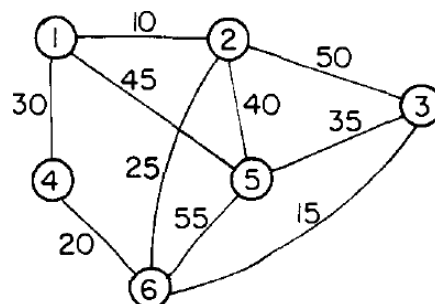
- i) All the edges are considered one by one in that order and deleted from the graph and are included in to the spanning tree.
- ii) At every stage an edge is included; the sub-graph at a stage need not be a tree. Infact it is a forest.
- iii) At the end if we include 'n' vertices and n-1 edges without forming cycles then we get a single connected component without any cycles i.e. a tree with minimum cost.

At every stage, as we include an edge in to the spanning tree, we get disconnected trees represented by various sets. While including an edge in to the spanning tree we need to check it does not form cycle. Inclusion of an edge (i,j) will form a cycle if i,j both are in same set. Otherwise the edge can be included into the spanning tree.



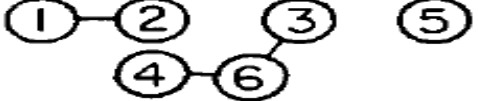
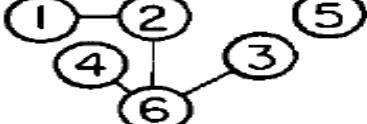
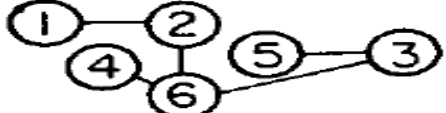
#### Kruskal minimum spanning tree algorithm

```

Algorithm Kruskal (E, cost, n,t)
//E is the set of edges in G. 'G' has 'n' vertices
//Cost {u,v} is the cost of edge (u,v) t is the set
//of edges in the minimum cost spanning tree
//The final cost is returned
{ construct a heap out of the edge costs using heapify;
  for i:= 1 to n do parent (i):= -1 // place in different sets
//each vertex is in different set      {1} {1}
  {3} i:= 0; min cost:= 0.0;
  While (i<n-1) and (heap not empty))do
  {
Delete a minimum cost edge (u,v) from the heaps; and reheapify using
adjust;j:= find (u); k:=find (v);
if (j k) then
{ i:= 1+1;
+ (i,1)=u; + (i, 2)=v;
mincost:=
mincost+cost(u,v); Union
(j,k);
}
}
if (i=n-1) then write ("No
spanningtree");else return
mincost;
}
  
```



Consider the above graph of , Using Kruskal's method the edges of this graph are considered for inclusion in the minimum cost spanning tree in the order (1, 2), (3, 6), (4, 6), (2, 6), (1, 4), (3, 5), (2, 5), (1, 5), (2, 3), and (5, 6). This corresponds to the cost sequence 10, 15, 20, 25, 30, 35, 40, 45, 50, 55. The first four edges are included in T. The next edge to be considered is (1, 4). This edge connects two vertices already connected in T and so it is rejected. Next, the edge (3, 5) is selected and that completes the spanning tree.

<u>Edge</u>	<u>Cost</u>	<u>Spanning Forest</u>
(1,2)	10	
(3,6)	15	
(4,6)	20	
(2,6)	25	
(1,4)	30	(reject)
(3,5)	35	

### Stages in Kruskal's algorithm

**Analysis:** - If the no/: of edges in the graph is given by  $|E|$  then the time for Kruskalsalgorithm is given by  $O(|E| \log |E|)$ .

### **UNIT-III**

Dynamic Programming: General method, applications- Matrix chained multiplication, Optimal binarysearch trees, 0/1 Knapsack problem, All pairs shortest path problem, Traveling sales person problem, Reliability design.

#### **Dynamic Programming**

Dynamic programming is a name, coined by Richard Bellman in 1955. Dynamic programming, as greedy method, is a powerful algorithm design technique that can be used when the solution to the problem may be viewed as the result of a sequence of decisions. In the greedy method we make irrevocable decisions one at a time, using a greedy criterion. However, in dynamic programming we examine the decision sequence to see whether an optimal decision sequence contains optimal decision subsequence.

When optimal decision sequences contain optimal decision subsequences, we can establish recurrence equations, called *dynamic-programming recurrence equations*, that enable us to solve the problem in an efficient way.

Dynamic programming is based on the principle of optimality (also coined by Bellman). The principle of optimality states that no matter whatever the initial state and initial decision are, the remaining decision sequence must constitute an optimal decision sequence with regard to the state resulting from the first decision. The principle implies that an optimal decision sequence is comprised of optimal decision subsequences. Since the principle of optimality may not hold for some formulations of some problems, it is necessary to verify that it does hold for the problem being solved. Dynamic programming cannot be applied when this principle does not hold.

The steps in a dynamic programming solution are:

- Verify that the principle of optimality holds
- Set up the dynamic-programming recurrence equations
- Solve the dynamic-programming recurrence equations for the value of the optimal solution.
- Perform a trace back step in which the solution itself is constructed.

## Matrix Chain Multiplication using Dynamic Programming

**Matrix chain multiplication problem:** Determine the optimal parenthesization of a product of  $n$  matrices.

Matrix chain multiplication (or Matrix Chain Ordering Problem, MCOP) is an optimization problem that to find the most efficient way to multiply a given sequence of matrices. The problem is not actually to perform the multiplications but merely to decide the sequence of the matrix multiplications involved.

The matrix multiplication is associative as no matter how the product is parenthesized, the result obtained will remain the same. For example, for four matrices  $A$ ,  $B$ ,  $C$ , and  $D$ , we would have:

$$((AB)C)D = ((A(BC))D) = (AB)(CD) = A((BC)D) = A(B(CD))$$

However, the order in which the product is parenthesized affects the number of simple arithmetic operations needed to compute the product or the efficiency. For example, if  $A$  is a  $10 \times 30$  matrix,  $B$  is a  $30 \times 5$  matrix, and  $C$  is a  $5 \times 60$  matrix, then computing  $(AB)C$  needs  $(10 \times 30 \times 5) + (10 \times 5 \times 60) = 1500 + 3000 = 4500$  operations while computing  $A(BC)$  needs  $(30 \times 5 \times 60) + (10 \times 30 \times 60) = 9000 + 18000 = 27000$  operations. Clearly, the first method is more efficient.

The idea is to break the problem into a set of related subproblems that group the given matrix to yield the lowest total cost.

Following is the recursive algorithm to find the minimum cost:

- Take the sequence of matrices and separate it into two subsequences.
- Find the minimum cost of multiplying out each subsequence.
- Add these costs together, and add in the price of multiplying the two result matrices.
- Do this for each possible position at which the sequence of matrices can be split, and take the minimum over all of them.

For example, if we have four matrices  $ABCD$ , we compute the cost required to find each of  $(A)(BCD)$ ,  $(AB)(CD)$ , and  $(ABC)(D)$ , making recursive calls to find the minimum cost to compute  $ABC$ ,  $AB$ ,  $CD$ , and  $BCD$  and then choose the best one. Better still, this yields the minimum cost and demonstrates the best way of doing the multiplication.

## All pairs shortest paths

In the all pairs shortest path problem, we are to find a shortest path between every pair of vertices in a directed graph  $G$ . That is, for every pair of vertices  $(i, j)$ , we are to find a shortest path from  $i$  to  $j$  as well as one from  $j$  to  $i$ . These two paths are the same when  $G$  is undirected.

When no edge has a negative length, the all-pairs shortest path problem may be solved by using Dijkstra's greedy single source algorithm  $n$  times, once with each of the  $n$  vertices as the source vertex.

The all pairs shortest path problem is to determine a matrix  $A$  such that  $A(i, j)$  is the length of a shortest path from  $i$  to  $j$ . The matrix  $A$  can be obtained by solving  $n$  single-source problems using the algorithm shortest Paths. Since each application of this procedure requires  $O(n^2)$  time, the matrix  $A$  can be obtained in  $O(n^3)$  time.

The dynamic programming solution, called Floyd's algorithm, runs in  $O(n^3)$  time. Floyd's algorithm works even when the graph has negative length edges (provided there are no negative length cycles).

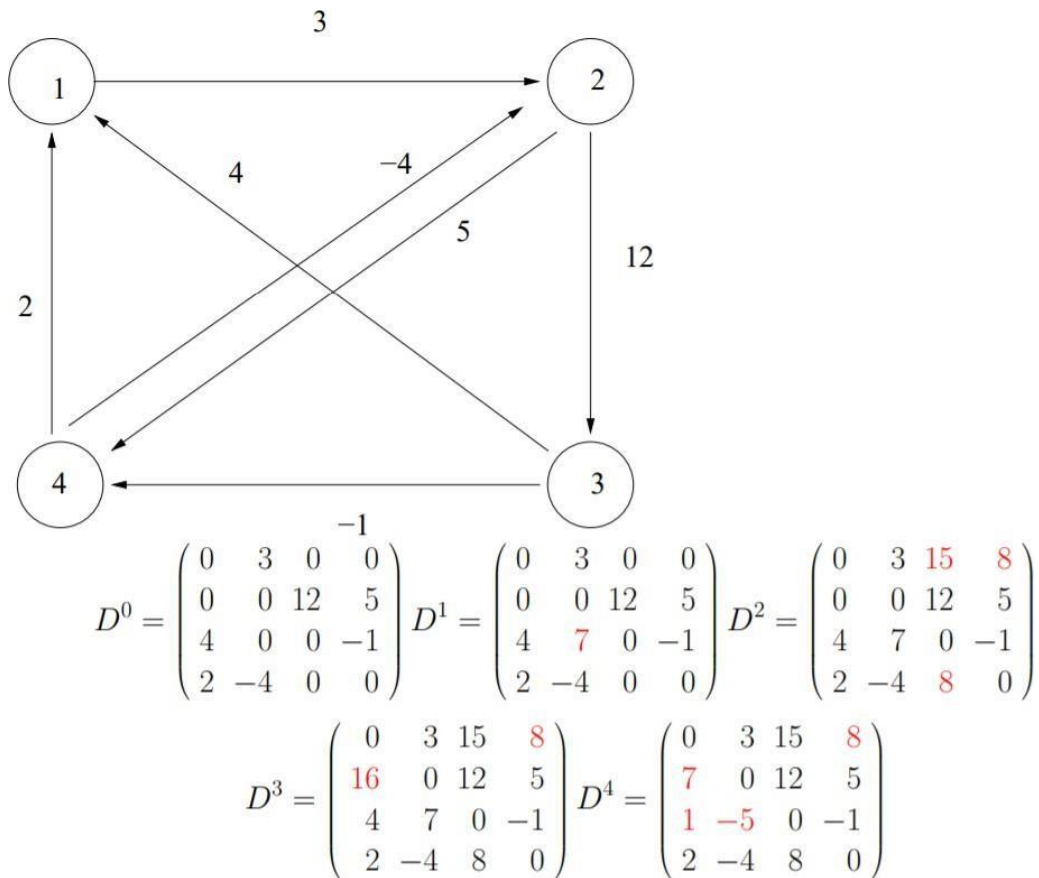
The shortest  $i$  to  $j$  path in  $G$ ,  $i \neq j$  originates at vertex  $i$  and goes through some intermediate vertices (possibly none) and terminates at vertex  $j$ . If  $k$  is an intermediate vertex on this shortest path, then the subpaths from  $i$  to  $k$  and from  $k$  to  $j$  must be shortest paths from  $i$  to  $k$  and  $k$  to  $j$ , respectively. Otherwise, the  $i$  to  $j$  path is not of minimum length. So, the principle of optimality holds. Let  $A^k(i, j)$  represent the length of a shortest path from  $i$  to  $j$  going through no vertex of index greater than  $k$ , we obtain:

$$A^k(i, j) = \{ \min_{1 \leq k < n} \{ A^{k-1}(i, k) + A^{k-1}(k, j) \}, c(i, j) \}$$

### Algorithm All Paths (Cost, A, n)

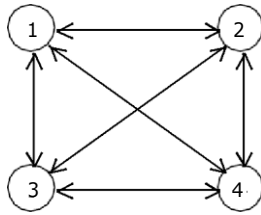
```
// cost [1:n, 1:n] is the cost adjacency matrix of a graph which
// n vertices; A [I, j] is the cost of a shortest path from vertex
// i to vertex j. cost [i, i] = 0.0, for  $1 \leq i \leq n$ .
{
    for i := 1 to n do
        for j := 1 to n do
            A [i, j] := cost [i, j]; // copy cost into A.
        for k := 1 to n do
            for i := 1 to n do
                for j := 1 to n do
                    A [i, j] := min (A [i, j], A [i, k] + A [k, j]);
    }
```

**Complexity Analysis:** A Dynamic programming algorithm based on this recurrence involves in calculating  $n+1$  matrices, each of size  $n \times n$ . Therefore, the algorithm has a complexity of  $O(n^3)$ .



## TRAVELLING SALESPERSON PROBLEM

For the following graph find minimum cost tour for the traveling sales person problem:



Let us start the tour from vertex 1:

$$g(1, V - \{1\}) = \min_{2 \leq k \leq n} \{c_{1k} + g(k, V - \{1, K\})\} \quad - \quad (1)$$

More generally writing:

$$g(i, s) = \min \{c_{ij} + g(j, s - \{j\})\} \quad - \quad (2)$$

Clearly,  $g(i, T) = c_{i1}$ ,  $1 \leq i \leq n$ . So,

$$g(2, T) = C_{21} = 5$$

$$g(3, T) = C_{31} = 6$$

$$g(4, T) = C_{41} = 8$$

Using equation – (2) we obtain:

$$g(1, \{2, 3, 4\}) = \min \{c_{12} + g(2, \{3, 4\}), c_{13} + g(3, \{2, 4\}), c_{14} + g(4, \{2, 3\})\}$$

$$g(2, \{3, 4\}) = \min \{c_{23} + g(3, \{4\}), c_{24} + g(4, \{3\})\} \\ = \min \{9 + g(3, \{4\}), 10 + g(4, \{3\})\}$$

$$g(3, \{4\}) = \min \{c_{34} + g(4, T)\} = 12 + 8 = 20$$

$$g(4, \{3\}) = \min \{c_{43} + g(3, T)\} = 9 + 6 = 15$$

Therefore,  $g(2, \{3, 4\}) = \min \{9 + 20, 10 + 15\} = \min \{29, 25\} = 25$

$$g(3, \{2, 4\}) = \min \{(c_{32} + g(2, \{4\})), (c_{34} + g(4, \{2\}))\}$$

$$g(2, \{4\}) = \min \{c_{24} + g(4, T)\} = 10 + 8 = 18$$

$$g(4, \{2\}) = \min \{c_{42} + g(2, T)\} = 8 + 5 = 13$$

Therefore,  $g(3, \{2, 4\}) = \min \{13 + 18, 12 + 13\} = \min \{31, 25\} = 25$

$$g(4, \{2, 3\}) = \min \{c_{42} + g(2, \{3\}), c_{43} + g(3, \{2\})\}$$

$$g(2, \{3\}) = \min \{c_{23} + g(3, T)\} = 9 + 6 = 15$$

$$g(3, \{2\}) = \min \{c_{32} + g(2, T)\} = 13 + 5 = 18$$



Therefore,  $g(4, \{2, 3\}) = \min \{8 + 15, 9 + 18\} = \min \{23, 27\} = 23$

$g(1, \{2, 3, 4\}) = \min \{c_{12} + g(2, \{3, 4\}), c_{13} + g(3, \{2, 4\}), c_{14} + g(4, \{2, 3\})\} = \min \{10 + 25, 15 + 25, 20 + 23\} = \min \{35, 40, 43\} = 35$

The optimal tour for the graph has length = 35 The

optimal tour is: 1, 2, 4, 3, 1.

## OPTIMAL BINARY SEARCH TREE

Let us assume that the given set of identifiers is  $\{a_1, \dots, a_n\}$  with  $a_1 < a_2 < \dots < a_n$ . Let  $p(i)$  be the probability with which we search for  $a_i$ . Let  $q(i)$  be the probability that the identifier  $x$  being searched for is such that  $a_i < x < a_{i+1}$ ,  $0 \leq i \leq n$  (assume  $a_0 = -\infty$  and  $a_{n+1} = +\infty$ ). We have to arrange the identifiers in a binary search tree in a way that minimizes the expected total access time.

In a binary search tree, the number of comparisons needed to access an element at depth 'd' is  $d + 1$ , so if ' $a_i$ ' is placed at depth ' $d_i$ ', then we want to minimize:

$$\sum_{i=1}^n P_i (1 + d_i) .$$

Let  $P(i)$  be the probability with which we shall be searching for ' $a_i$ '. Let  $Q(i)$  be the probability of an un-successful search. Every internal node represents a point where a successful search may terminate. Every external node represents a point where an unsuccessful search may terminate.

The expected cost contribution for the internal node for ' $a_i$ ' is:

$$P(i) * level(a_i) .$$

Unsuccessful search terminate with  $I=0$  (i.e at an external node). Hence the cost contribution for this node is:

$$Q(i) * level((E_i) - 1)$$

The expected cost of binary search tree is:

$$\sum_{i=1}^n P(i) * level(a_i) + \sum_{i=1}^n Q(i) * level((E_i) - 1)$$

Given a fixed set of identifiers, we wish to create a binary search tree organization. We may expect different binary search trees for the same identifier set to have different performance characteristics.

The computation of each of these  $c(i, j)$ 's requires us to find the minimum of  $m$  quantities. Hence, each such  $c(i, j)$  can be computed in time  $O(m)$ . The total time for all  $c(i, j)$ 's with  $j - i = m$  is therefore  $O(nm - m^2)$ .

The total time to evaluate all the  $c(i, j)$ 's and  $r(i, j)$ 's is therefore:

$$\sim (nm - m^2) = O(n^3) \\ ) 1 < m < n$$

### Example 1:

Let  $n = 4$ , and  $(a_1, a_2, a_3, a_4) = (\text{do}, \text{if}, \text{need}, \text{while})$  Let  $P(1:4) = (3, 3, 1, 1)$  and  $Q(0:4) = (2, 3, 1, 1, 1)$

### Solution:

Table for recording  $W(i, j)$ ,  $C(i, j)$  and  $R(i, j)$ :

Column Row	0	1	2	3	4
0	2, 0, 0	3, 0, 0	1, 0, 0	1, 0, 0	1, 0, 0
1	8, 8, 1	7, 7, 2	3, 3, 3	3, 3, 4	
2	12, 19, 1	9, 12, 2	5, 8, 3		
3	14, 25, 2	11, 19, 2			
4	16, 32, 2				

This computation is carried out row-wise from row 0 to row 4. Initially,  $W(i, i) = Q(i)$  and  $C(i, i) = 0$  and  $R(i, i) = 0$ ,  $0 \leq i < 4$ .

Solving for  $C(0, n)$ :

**First**, computing all  $C(i, j)$  such that  $j - i = 1$ ;  $j = i + 1$  and as  $0 \leq i < 4$ ;  $i = 0, 1, 2$  and  $3$ ;  $i < k \leq j$ . Start with  $i = 0$ ; so  $j = 1$ ; as  $i < k \leq j$ , so the possible value for  $k = 1$

$$W(0, 1) = P(1) + Q(1) + W(0, 0) = 3 + 3 + 2 = 8$$

$$C(0, 1) = W(0, 1) + \min \{C(0, 0) + C(1, 1)\} = 8$$

$$R(0, 1) = 1 \text{ (value of 'K' that is minimum in the above equation).}$$

Next with  $i = 1$ ; so  $j = 2$ ; as  $i < k \leq j$ , so the possible value for  $k = 2$

$$W(1, 2) = P(2) + Q(2) + W(1, 1) = 3 + 1 + 3 = 7$$

$$C(1, 2) = W(1, 2) + \min \{C(1, 1) + C(2, 2)\} = 7$$

$$R(1, 2) = 2$$

Next with  $i = 2$ ; so  $j = 3$ ; as  $i < k \leq j$ , so the possible value for  $k = 3$

$$W(2, 3) = P(3) + Q(3) + W(2, 2) = 1 + 1 + 1 = 3$$

$$C(2, 3) = W(2, 3) + \min \{C(2, 2) + C(3, 3)\} = 3 + [(0 + 0)] = 3$$

$$R(2, 3) = 3$$

Next with  $i = 3$ ; so  $j = 4$ ; as  $i < k \leq j$ , so the possible value for  $k = 4$

$$W(3, 4) = P(4) + Q(4) + W(3, 3) = 1 + 1 + 1 = 3$$

$$C(3, 4) = W(3, 4) + \min \{ [C(3, 3) + C(4, 4)] \} = 3 + [(0 + 0)] = 3$$

$$ft(3, 4) = 4$$

**Second**, Computing all  $C(i, j)$  such that  $j - i = 2$ ;  $j = i + 2$  and as  $0 \leq i < 3$ ;  $i = 0, 1, 2$ ;  $i < k \leq J$ . Start with  $i = 0$ ; so  $j = 2$ ; as  $i < k \leq J$ , so the possible values for  $k = 1$  and  $2$ .

$$W(0, 2) = P(2) + Q(2) + W(0, 1) = 3 + 1 + 8 = 12$$

$$C(0, 2) = W(0, 2) + \min \{ (C(0, 0) + C(1, 2)), (C(0, 1) + C(2, 2)) \} = 12$$

$$+ \min \{ (0 + 7, 8 + 0) \} = 19$$

$$ft(0, 2) = 1$$

Next, with  $i = 1$ ; so  $j = 3$ ; as  $i < k \leq j$ , so the possible value for  $k = 2$  and  $3$ .

$$W(1, 3) = P(3) + Q(3) + W(1, 2) = 1 + 1 + 7 = 9$$

$$C(1, 3) = W(1, 3) + \min \{ [C(1, 1) + C(2, 3)], [C(1, 2) + C(3, 3)] \}$$

$$= W(1, 3) + \min \{ (0 + 3), (7 + 0) \} = 9 + 3 = 12$$

$$ft(1, 3) = 2$$

Next, with  $i = 2$ ; so  $j = 4$ ; as  $i < k \leq j$ , so the possible value for  $k = 3$  and  $4$ .

$$W(2, 4) = P(4) + Q(4) + W(2, 3) = 1 + 1 + 3 = 5$$

$$C(2, 4) = W(2, 4) + \min \{ [C(2, 2) + C(3, 4)], [C(2, 3) + C(4, 4)] \}$$

$$= 5 + \min \{ (0 + 3), (3 + 0) \} = 5 + 3 = 8$$

$$ft(2, 4) = 3$$

**Third**, Computing all  $C(i, j)$  such that  $J - i = 3$ ;  $j = i + 3$  and as  $0 \leq i < 2$ ;  $i = 0, 1$ ;  $i < k \leq J$ . Start with  $i = 0$ ; so  $j = 3$ ; as  $i < k \leq j$ , so the possible values for  $k = 1, 2$  and  $3$ .

$$W(0, 3) = P(3) + Q(3) + W(0, 2) = 1 + 1 + 12 = 14$$

$$C(0, 3) = W(0, 3) + \min \{ [C(0, 0) + C(1, 3)], [C(0, 1) + C(2, 3)], [C(0, 2) + C(3, 3)] \}$$

$$14 + \min \{ (0 + 12), (8 + 3), (19 + 0) \} = 14 + 11 = 25$$

$$ft(0, 3)$$

Start with  $i = 1$ ; so  $j = 4$ ; as  $i < k \leq j$ , so the possible values for  $k = 2, 3$  and  $4$ .

$$W(1, 4) = P(4) + Q(4) + W(1, 3) = 1 + 1 + 9 = 11 = W(2, 4)$$

$$C(1, 4) = W(1, 4) + \min \{ [C(1, 1) + C(2, 4)], [C(1, 3) + C(4, 4)] \} + 8 = 19$$

$$ft(1, 4) = 11 + \min \{ (0 + 8), (7 + 3), (12 + 0) \} = 11 + 2 = 13$$

**Fourth,** Computing all  $C(i, j)$  such that  $j - i = 4$ ;  $j = i + 4$  and as  $0 \leq i < 1$ ;  $i = 0$ ;  $i < k \leq J$ .

Start with  $i = 0$ ; so  $j = 4$ ; as  $i < k \leq j$ , so the possible values for  $k = 1, 2, 3$  and  $4$ .

$$W(0, 4) = P(4) + Q(4) + W(0, 3) = 1 + 1 + 14 = 16$$

$$C(0, 4) = W(0, 4) + \min \{ [C(0, 0) + C(1, 4)], [C(0, 1) + C(2, 4)], [C(0, 2) + C(3, 4)], [C(0, 3) + C(4, 4)] \}$$

$$= 16 + \min [0 + 19, 8 + 8, 19 + 3, 25 + 0] = 16 + 16 = 32$$

$$ft(0, 4) = 2$$

From the table we see that  $C(0, 4) = 32$  is the minimum cost of a binary search tree for  $(a_1, a_2, a_3, a_4)$ . The root of the tree 'T04' is 'a2'.

Hence the left sub tree is 'T01' and right sub tree is T24. The root of 'T01' is 'a1' and the root of 'T24' is a3.

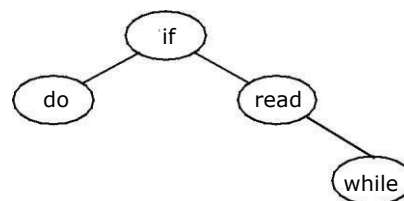
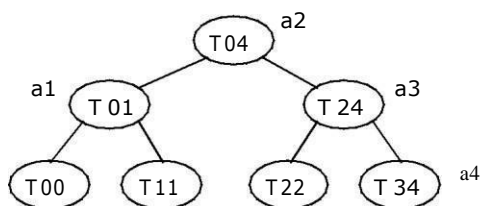
The left and right sub trees for 'T01' are 'T00' and 'T11' respectively. The root of T01 is 'a1'

The left and right sub trees for T24 are T22 and T34 respectively.

The root of T24 is 'a3'.

The root of T22 is null

The root of T34 is a4.



---

## 0/1 – KNAPSACK

We are given  $n$  objects and a knapsack. Each object  $i$  has a positive weight  $w_i$  and a positive value  $V_i$ . The knapsack can carry a weight not exceeding  $W$ . Fill the knapsack so that the value of objects in the knapsack is optimized.

A solution to the knapsack problem can be obtained by making a sequence of decisions on the variables  $x_1, x_2, \dots, x_n$ . A decision on variable  $x_i$  involves determining which of the values 0 or 1 is to be assigned to it. Let us assume that

decisions on the  $x_i$  are made in the order  $x_n, x_{n-1}, \dots, x_1$ . Following a decision on  $x_n$ , we may be in one of two possible states: the capacity remaining in  $m - w_n$  and a profit of  $p_n$  has accrued. It is clear that the remaining decisions  $x_{n-1}, \dots, x_1$  must be optimal with respect to the problem state resulting from the decision on  $x_n$ . Otherwise,  $x_n, \dots, x_1$  will not be optimal. Hence, the principle of optimality holds.

$$F_n(m) = \max \{f_{n-1}(m), f_{n-1}(m - w_n) + p_n\} \quad \text{--} \quad 1$$

For arbitrary  $f_i(y)$ ,  $i > 0$ , this equation generalizes to:

$$F_i(y) = \max \{f_{i-1}(y), f_{i-1}(y - w_i) + p_i\} \quad \text{--} \quad 2$$

Equation-2 can be solved for  $f_n(m)$  by beginning with the knowledge  $f_0(y) = 0$  for all  $y$  and  $f_i(y) = -\infty$ ,  $y < 0$ . Then  $f_1, f_2, \dots, f_n$  can be successively computed using equation-2.

When the  $w_i$ 's are integer, we need to compute  $f_i(y)$  for integer  $y$ ,  $0 \leq y \leq m$ . Since  $f_i(y) = -\infty$  for  $y < 0$ , these function values need not be computed explicitly. Since each  $f_i$  can be computed from  $f_{i-1}$  in  $\Theta(m)$  time, it takes  $\Theta(mn)$  time to compute  $f_n$ . When the  $w_i$ 's are real numbers,  $f_i(y)$  is needed for real numbers  $y$  such that  $0 < y \leq m$ . So,  $f_i$  cannot be explicitly computed for all  $y$  in this range. Even when the  $w_i$ 's are integer, the explicit  $\Theta(mn)$  computation of  $f_n$  may not be the most efficient computation. So, we explore **an alternative method for both cases**.

The  $f_i(y)$  is an ascending step function; i.e., there are a finite number of  $y$ 's,  $0 = y_1 < y_2 < \dots < y_k$ , such that  $f_i(y_1) < f_i(y_2) < \dots < f_i(y_k)$ ;  $f_i(y) = -\infty$ ,  $y < y_1$ ;  $f_i(y) = f_i(y_k)$ ,  $y \geq y_k$ ; and  $f_i(y) = f_i(y_j)$ ,  $y_j \leq y < y_{j+1}$ . So, we need to compute only  $f_i(y_j)$ ,  $1 \leq j \leq k$ . We use the ordered set  $S^i = \{(f(y_j), y_j) \mid 1 \leq j \leq k\}$  to represent  $f_i(y)$ . Each number of  $S^i$  is a pair  $(P, W)$ , where  $P = f_i(y_j)$  and  $W = y_j$ . Notice that  $S^0 = \{(0, 0)\}$ . We can compute  $S^{i+1}$  from  $S^i$  by first computing:

$$S^{i+1} = \{(P, W) \mid (P - p_i, W - w_i) \in S^i\}$$

Now,  $S^{i+1}$  can be computed by merging the pairs in  $S^i$  and  $S^{i+1}$  together. Note that if  $S^{i+1}$  contains two pairs  $(P_j, W_j)$  and  $(P_k, W_k)$  with the property that  $P_j \leq P_k$  and  $W_j > W_k$ , then the pair  $(P_j, W_j)$  can be discarded because of equation-2. Discarding or purging rules such as this one are also known as dominance rules. Dominated tuples get purged. In the above,  $(P_k, W_k)$  dominates  $(P_j, W_j)$ .

## Reliability Design

The problem is to design a system that is composed of several devices connected in series. Let  $r_i$  be the reliability of device  $D_i$  (that is  $r_i$  is the probability that device  $i$  will function properly) then the reliability of the entire system is  $\prod r_i$ . Even if the individual devices are very reliable (the  $r_i$ 's are very close to one), the reliability of the system may not be very good. For example, if  $n = 10$  and  $r_i = 0.99$ ,  $1 \leq i \leq 10$ , then  $\prod r_i = .904$ . Hence, it is desirable to duplicate devices. Multiple copies of the same device type are connected in parallel.

If stage  $i$  contains  $m_i$  copies of device  $D_i$ . Then the probability that all  $m_i$  have a malfunction is  $(1 - r_i)^{m_i}$ . Hence the reliability of stage  $i$  becomes  $1 - (1 - r_i)^{m_i}$ .

**EXAMPLE** Consider a 4-stage system with device types  $d_1, d_2, d_3, d_4$ . Let  $c_1 = 30$ ,  $c_2 = 15$ ,  $c_3 = 20$ ,  $c_4 = 10$ . The cost of the system is  $C \leq 135$ . The reliabilities of the devices are:

$$r_1 = 0.9, r_2 = 0.8, r_3 = 0.5, r_4 = 0.6$$

Subject to the budget constraint, the allowable maximum number of devices at different stages are:

$$u_1 = \lfloor (165 - 75)/30 \rfloor = 3$$

$$u_2 = \lfloor (150 - 75)/15 \rfloor = 5$$

$$u_3 = \lfloor (155 - 75)/20 \rfloor = 4$$

$$u_4 = \lfloor (145 - 75)/10 \rfloor = 7$$

We use  $S^i$  to represent the set of all undominated tuples  $(f_i(x), x)$  that may result from the various decision sequences for  $m_1, m_2, \dots, m_i$ . Beginning with  $S^0 = \{(1, 0)\}$ , we can construct  $S^i$  from  $S^{i-1}$  by trying out all possible values for  $m_i$  and combining the resulting tuples together.  $S_j^i$  represents all tuples obtainable from  $S^{i-1}$  by choosing  $m_i = j$ . Some tuples below are shown in bold. These are the entries that are used for finding the solution through backtracing as explained later.

$$S_1^1 = \{(\mathbf{0.9}, \mathbf{30})\}, S_2^1 = \{(0.9, 30), (.99, 60)\}, S_3^1 = \{(0.9, 30), (.99, 60), (0.999, 90)\}.$$

$$S^1 = \{(\mathbf{0.9}, \mathbf{30}), (.99, 60), (0.999, 90)\}.$$

$$S_1^2 = \{(0.72, 45), (0.792, 75), (0.7992, 105)\}.$$

$$S_2^2 = \{(\mathbf{0.864}, \mathbf{60}), (0.9504, 90), (0.95904, 120)\}.$$

$$S_3^2 = \{(0.8928, 75), (0.98208, 105), (0.991008, 135)\}.$$

$$S_4^2 = \{(0.89856, 90), (0.988416, 120)\}.$$

$$S_5^2 = \{(0.899712, 105), (0.9896832, 135)\}.$$

Removing dominated tuples, we get

$$S^2 = \{(0.72, 45), (\mathbf{0.864}, \mathbf{60}), (0.8928, 75), (0.9504, 90), (0.98208, 105), (0.988416, 120), (0.991008, 135)\}.$$

$$S_1^3 = \{(0.36, 65), (0.432, 80), (0.4464, 95), (0.4752, 110), (0.49104, 125)\}.$$

$$S_2^3 = \{(0.54, 85), (\mathbf{0.648}, \mathbf{100}), (0.6696, 115), (0.7128, 130)\}.$$

$$S_3^3 = \{(0.63, 105), (0.756, 120), (0.7812, 135)\}.$$

$$S_4^3 = \{(0.675, 125)\}.$$

Removing dominated tuples, we get

$$S^3 = \{(0.36, 65), (0.432, 80), (0.54, 85), (\mathbf{0.648}, \mathbf{100}), (0.6696, 115), (0.756, 120), (0.7812, 135)\}.$$

$$S_1^4 = \{(0.216, 75), (0.2592, 90), (0.324, 95), (0.3888, 110), (0.40176, 125), (0.4536, 130)\}.$$

$$S_2^4 = \{(0.3024, 85), (0.36288, 100), (0.4536, 105), (0.54432, 120), (0.562464, 135)\}.$$

$$S_3^4 = \{(0.33696, 95), (0.404352, 110), (0.50544, 115), (\mathbf{0.606528, 130})\}.$$

$$S_4^4 = \{(0.350784, 105), (0.4209408, 120), (0.526176, 125)\}.$$

$$S_5^4 = \{(0.3563136, 115), (0.4275763, 130), (0.5344704, 135)\}.$$

$$S_6^4 = \{(0.3585254, 125)\}.$$

$$S_7^4 = \{(0.3594101, 135)\}.$$

Removing dominated tuples, we get

$$S^4 = \{(0.216, 75), (0.3024, 85), (0.33696, 95), (0.36288, 100), (0.4536, 105), (0.54432, 120), (\mathbf{0.606528, 130})\}.$$

We find that 0.606528 is the highest reliability attainable within the given budget at a cost of 130. (0.606528, 130) appears in  $S_3^4$  and it does not appear in  $S^3$ . So  $m_4 = 3$ . (0.606528, 130) has come from (0.648, 100) in  $S^3$ . (0.648, 100) appears in  $S_2^3$  and it does not appear in  $S^2$ . So  $m_3 = 2$ . (0.648, 100) has come from (0.864, 60). (0.864, 60) appears in  $S_2^2$  and it does not appear in  $S^1$ . So  $m_2 = 2$ . (0.864, 60) has come from (0.9, 30) in  $S^1$ . (0.9, 30) appears in  $S_1^1$ . So  $m_1 = 1$ . So the optimal design of the machine is as shown in Figure 4.11.

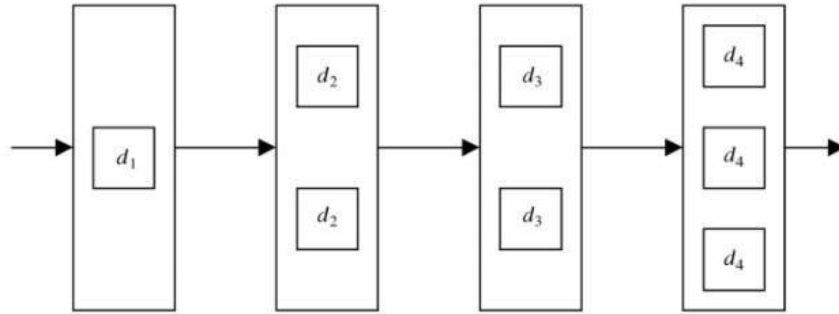


FIGURE 4.11 | Designed machine with highest reliability.

The reliability of the machine is  $0.9 \times (1 - 0.2 \times 0.2) \times (1 - 0.5 \times 0.5) \times (1 - 0.4 \times 0.4 \times 0.4) = 0.9 \times 0.96 \times 0.75 \times 0.936 = 0.606528$  and the cost of the machine is  $1 \times 30 + 2 \times 15 + 2 \times 20 + 3 \times 10 = 130$ . The machine will have reliability 0.216 without redundant devices.



## UNIT IV:

Backtracking: General method, Applications- n-queue problem, Sum of subsets problem, Graph coloring, Hamiltonian cycles.

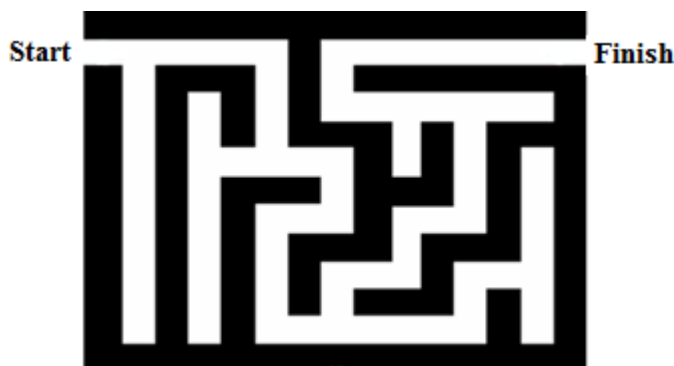
### **Backtracking (General method)**

Many problems are difficult to solve algorithmically. Backtracking makes it possible to solve at least some large instances of difficult combinatorial problems.

Suppose you have to make a series of decisions among various choices, where

- You don't have enough information to know what to choose
- Each decision leads to a new set of choices.
- Some sequence of choices ( more than one choices) may be a solution to your problem.

Backtracking is a methodical (Logical) way of trying out various sequences of decisions, until you find one that “works”



Given a maze, find a path from start to finish.

- In maze, at each intersection, you have to decide between 3 or fewer choices:
  - ✓ Go straight
  - ✓ Go left
  - ✓ Go right
- You don't have enough information to choose correctly
- Each choice leads to another set of choices.
- One or more sequences of choices may or may not lead to a solution.
- Many types of maze problem can be solved with backtracking.

Sorting the array of integers in  $a[1:n]$  is a problem whose solution is expressible by an  $n$ -tuple

$x_i$  is the index in 'a' of the  $i^{\text{th}}$  smallest element.

The criterion function 'P' is the inequality  $a[x_i] \leq a[x_{i+1}]$  for  $1 \leq i \leq n$

$S_i$  is finite and includes the integers 1 through  $n$ .

$m_i$  size of set  $S_i$

$m = m_1 m_2 m_3 \dots m_n$   $n$  tuples that possible candidates for satisfying the function P.

With brute force approach would be to form all these  $n$ -tuples, evaluate (judge) each one with P and save those which yield the optimum.

By using backtrack algorithm; yield the same answer with far fewer than 'm' trails.

Many of the problems we solve using backtracking requires that all the solutions satisfy a complex set of constraints.

For any problem these constraints can be divided into two categories:

- Explicit constraints.
- Implicit constraints.

**Explicit constraints:** Explicit constraints are rules that restrict each  $x_i$  to take on values only from a given set.

Example:  $x_i \geq 0$  or  $s_i = \{\text{all non negative real numbers}\}$

$X_i = 0$  or  $1$  or  $S_i = \{0, 1\}$

$l_i \leq x_i \leq u_i$  or  $s_i = \{a: l_i \leq a \leq u_i\}$

The explicit constraint depends on the particular instance  $I$  of the problem being solved. All tuples that satisfy the explicit constraints define a possible solution space for  $I$ .

### **Implicit Constraints:**

The implicit constraints are rules that determine which of the tuples in the solution space of  $I$  satisfy the criterion function. Thus implicit constraints describe the way in which the  $x_i$  must relate to each other.

### **Applications of Backtracking:**

- N Queens Problem
- Sum of subsets problem
- Graph coloring
- Hamiltonian cycles.

### **N-Queens Problem:**

It is a classic combinatorial problem. The eight queen's puzzle is the problem of placing eight queens puzzle is the problem of placing eight queens on an  $8 \times 8$  chessboard so that no two queens attack each other. That is so that no two of them are on the same row, column, or diagonal.

The 8-queens puzzle is an example of the more general n-queens problem of placing n queens on an  $n \times n$  chessboard.

	1	2	3	4	5	6	7	8
1				Q				
2						Q		
3								Q
4		Q						
5							Q	
6	Q							
7			Q					
8					Q			

**One solution to the 8-queens problem**

Here queens can also be numbered 1 through 8

Each queen must be on a different row

Assume queen 'i' is to be placed on row 'i'

All solutions to the 8-queens problem can therefore be represented as s-tuples  $(x_1, x_2, x_3, \dots, x_8)$

$x_i$  = the column on which queen 'i' is placed

$s_i \in \{1, 2, 3, 4, 5, 6, 7, 8\}, 1 \leq i \leq 8$

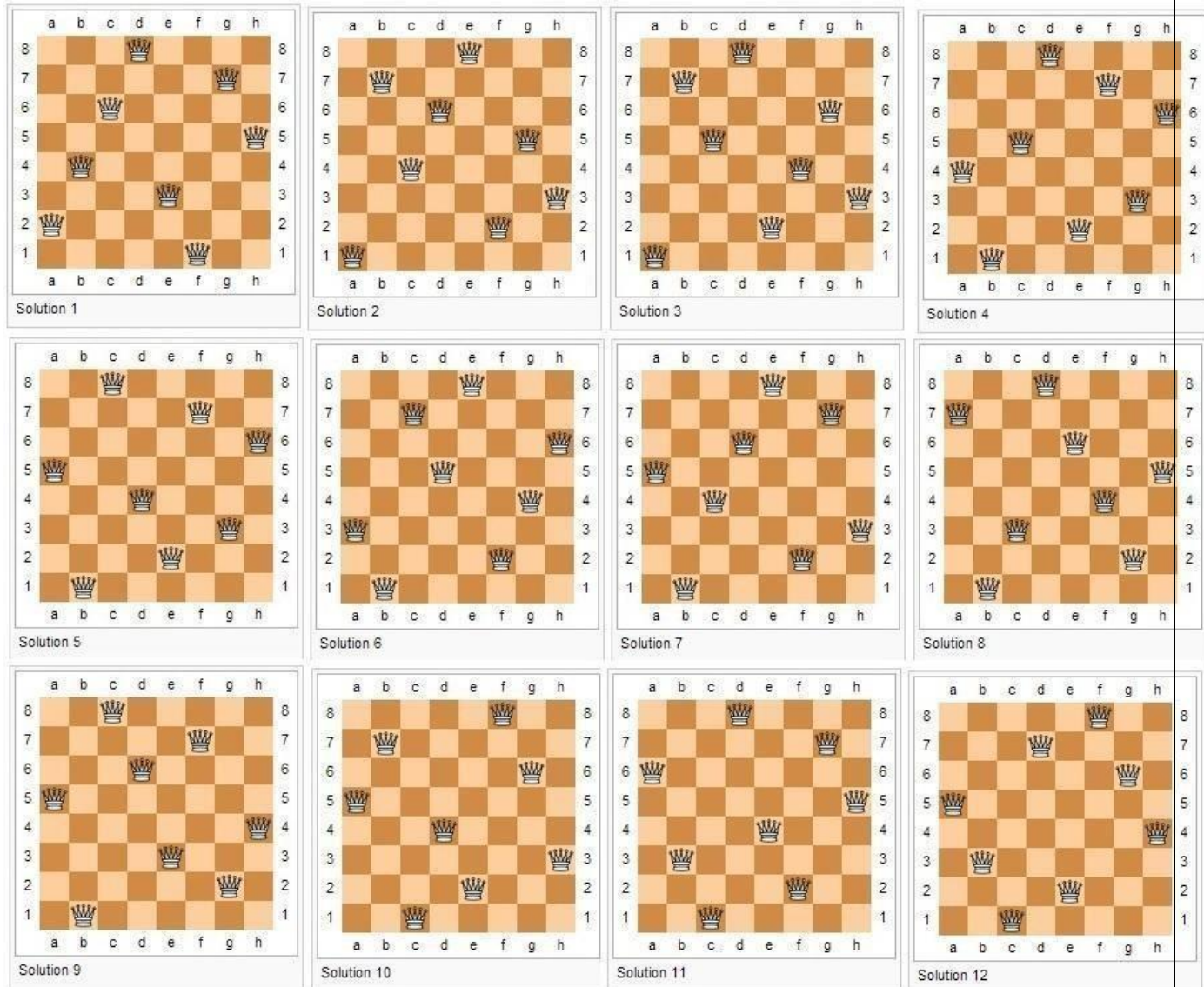
Therefore the solution space consists of  $8^8$  s-tuples.

The implicit constraints for this problem are that no two  $x_i$ 's can be the same column and no two queens can be on the same diagonal.

By these two constraints the size of solution space reduces from  $8^8$  tuples to  $8!$  Tuples.

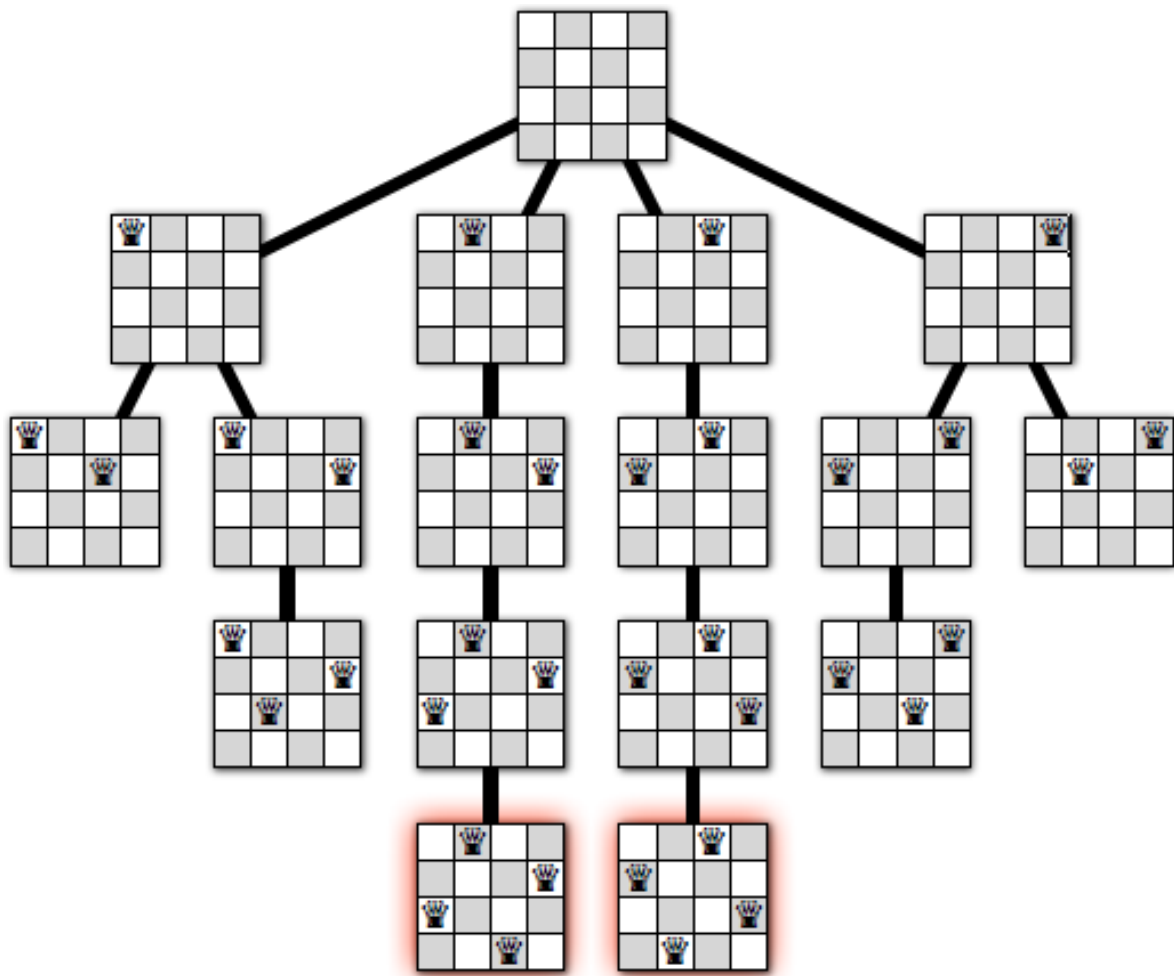
Form example  $s_i(4, 6, 8, 2, 7, 1, 3, 5)$

In the same way for n-queens are to be placed on an  $n \times n$  chessboard, the solution space consists of all  $n!$  Permutations of n-tuples (1,2, ---n).



Some solution to the 8-Queens problem

Algorithm for new queen be placed	All solutions to the n-queens problem
<pre> Algorithm Place(k,i) //Return true if a queen can be placed in kth row &amp; ith column //Other wise return false { for j:=1 to k-1 do if(x[j]=i or Abs(x[j]-i)=Abs(j-k)) then return false return true } </pre>	<pre> Algorithm NQueens(k, n) // its prints all possible placements of n- queens on an n×n chessboard. { for i:=1 to n do{ if Place(k,i) then { X[k]:=i; if(k==n) then write (x[1:n]); else NQueens(k+1, n); } }} </pre>



The complete recursion tree for our algorithm for the 4 queens problem.

### **Sum of Subsets Problem:**

Given positive numbers  $w_i$   $1 \leq i \leq n$ , &  $m$ , here sum of subsets problem is finding all subsets of  $w_i$  whose sums are  $m$ .

**Definition:** Given  $n$  distinct +ve numbers (usually called weights), desire (want) to find all combinations of these numbers whose sums are  $m$ . this is called sum of subsets problem. To formulate this problem by using either fixed sized tuples or variable sized tuples. Backtracking solution uses the fixed size tuple strategy.

### **For example:**

If  $n=4$  ( $w_1, w_2, w_3, w_4$ )=(11,13,24,7) and  $m=31$ .

Then desired subsets are (11, 13, 7) & (24, 7).

The two solutions are described by the vectors (1, 2, 4) and (3, 4).

In general all solution are  $k$ -tuples  $(x_1, x_2, x_3, \dots, x_k)$   $1 \leq k \leq n$ , different solutions may have different sized tuples.

- Explicit constraints requires  $x_i \in \{j / j \text{ is an integer } 1 \leq j \leq n\}$
- Implicit constraints requires:  
No two be the same & that the sum of the corresponding  $w_i$ 's be  $m$   
i.e., (1, 2, 4) & (1, 4, 2) represents the same. Another constraint is  $x_i < x_{i+1}$   $1 \leq i \leq k$

$W_i$  □ weight of item  $i$

M □ Capacity of bag (subset)

X<sub>i</sub> □ the element of the solution vector is either one or zero.

X<sub>i</sub> value depending on whether the weight w<sub>i</sub> is included or not.

If X<sub>i</sub>=1 then w<sub>i</sub> is chosen.

If X<sub>i</sub>=0 then w<sub>i</sub> is not chosen.

$$\underbrace{\sum_{i=1}^k W(i)X(i)}_{\text{Total sum till now}} + \underbrace{\sum_{i=k+1}^n W(i)}_{\text{Still there}} \geq M$$

The above equation specifies that x<sub>1</sub>, x<sub>2</sub>, x<sub>3</sub>, ... x<sub>k</sub> cannot lead to an answer node if this condition is not satisfied.

$$\sum_{i=1}^k W(i)X(i) + W(k+1) > M$$

The equation cannot lead to solution.

$$B_k(X(1), \dots, X(k)) = \text{true iff} \left( \sum_{i=1}^k W(i)X(i) + \sum_{i=k+1}^n W(i) \geq M \text{ and } \sum_{i=1}^k W(i)X(i) + W(k+1) \leq M \right)$$

$$s = \sum_{j=1}^{k-1} W(j)X(j). \quad \text{and} \quad r = \sum_{j=k}^n W(j)$$

Recursive backtracking algorithm for sum of subsets problem

Algorithm SumOfSub(s, k, r)

{

$$//s = \sum_{j=1}^{k-1} W(j)X(j). \quad \text{and} \quad r = \sum_{j=k}^n W(j)$$

X[k]=1

If (S+w[k]=m) then write(x[1: ]); // subset found.

Else if (S+w[k] + w{k+1} ≤ M)

Then SumOfSub(S+w[k], k+1, r-w[k]);

if ((S+r - w{k} ≥ M) and (S+w[k+1] ≤ M) ) then

{

X[k]=0;

SumOfSub(S, k+1, r-w[k]);

}

}

## Graph Coloring:

Let  $G$  be a undirected graph and ' $m$ ' be a given +ve integer. The graph coloring problem is assigning colors to the vertices of an undirected graph with the restriction that no two adjacent vertices are assigned the same color yet only ' $m$ ' colors are used.

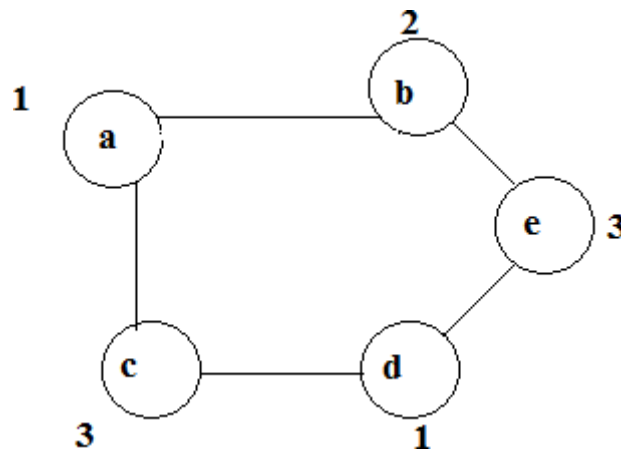
The optimization version calls for coloring a graph using the minimum number of coloring.

The decision version, known as  $K$ -coloring asks whether a graph is colourable using at most  $k$ -colors.

Note that, if ' $d$ ' is the degree of the given graph then it can be colored with ' $d+1$ ' colors.

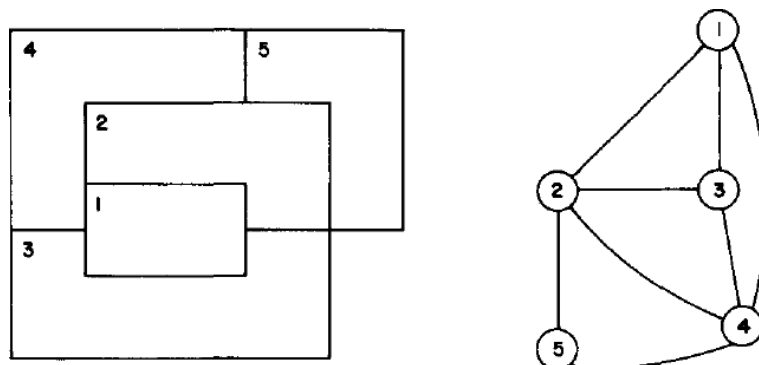
The  $m$ -colorability optimization problem asks for the smallest integer ' $m$ ' for which the graph  $G$  can be colored. This integer is referred as "**Chromatic number**" of the graph.

### Example



- Above graph can be colored with 3 colors 1, 2, & 3.
- The color of each node is indicated next to it.
- 3-colors are needed to color this graph and hence this graph' Chromatic Number is 3.
- A graph is said to be planar iff it can be drawn in a plane (flat) in such a way that no two edges cross each other.
- **M-Colorability decision problem** is the 4-color problem for planar graphs.
- Given any map, can the regions be colored in such a way that no two adjacent regions have the same color yet only 4-colors are needed?
- To solve this problem, graphs are very useful, because a map can easily be transformed into a graph.
- Each region of the map becomes a node, and if two regions are adjacent, then the corresponding nodes are joined by an edge.

### o Example:



o

**A map and its planar graph representation**

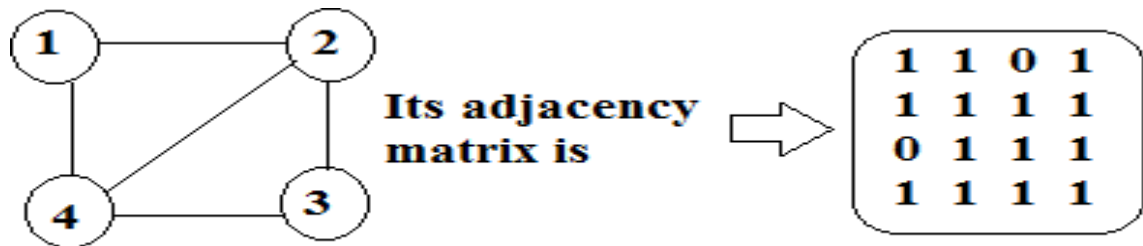
The above map requires 4 colors.

➤ Many years, it was known that 5-colors were required to color this map.

➤ After several hundred years, this problem was solved by a group of mathematicians with the help of a computer. They show that 4-colors are sufficient.

Suppose we represent a graph by its adjacency matrix  $G[1:n, 1:n]$

Ex:



Here  $G[i, j]=1$  if  $(i, j)$  is an edge of  $G$ , and  $G[i, j]=0$  otherwise.

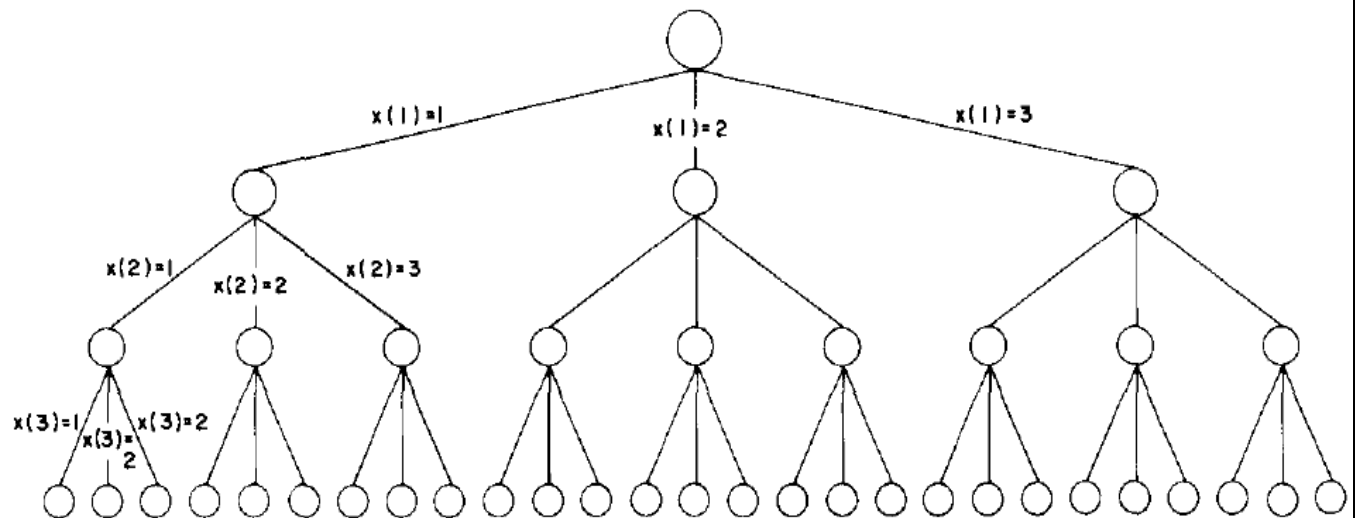
Colors are represented by the integers 1, 2, ...,  $m$  and the solutions are given by the  $n$ -tuple  $(x_1, x_2, \dots, x_n)$

$x_i$  Color of node  $i$ .

State Space Tree for

$n=3$  nodes

$m=3$  colors



**State space tree for M Coloring when  $n = 3$  and  $m = 3$**

1<sup>st</sup> node coloured in 3-ways

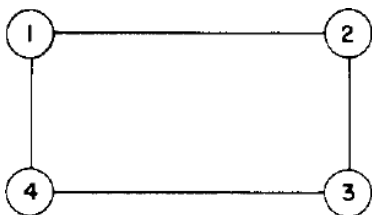
2<sup>nd</sup> node coloured in 3-ways

3<sup>rd</sup> node coloured in 3-ways

So we can colour in the graph in 27 possibilities of colouring.

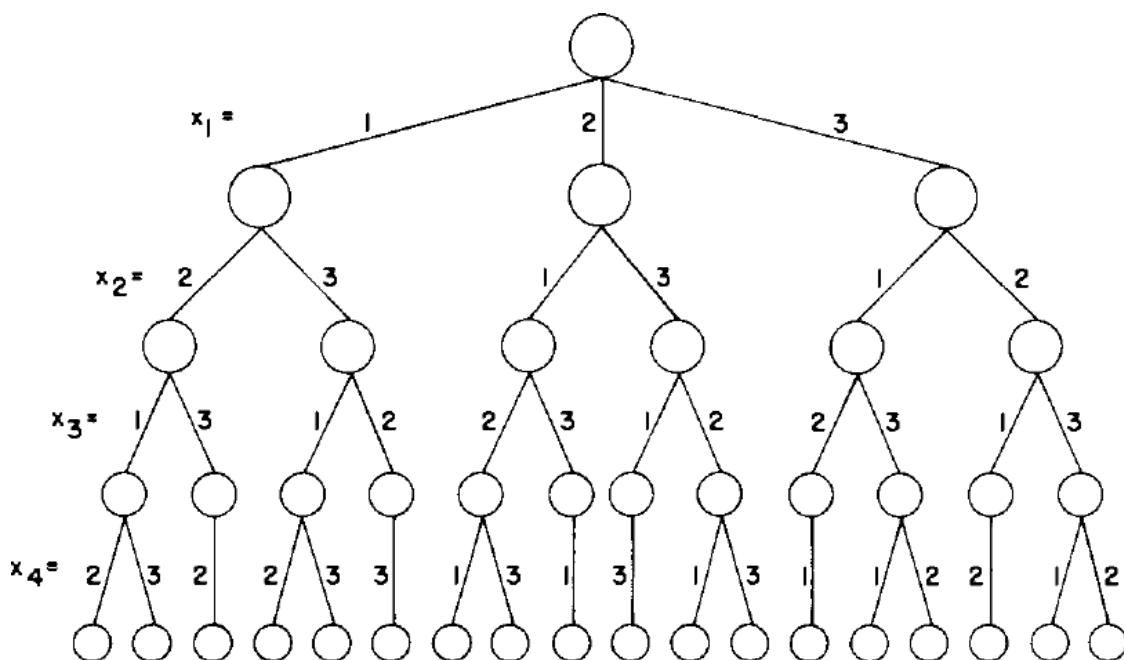


Finding all m-coloring of a graph	Getting next color
<pre> Algorithm mColoring(k){ // g(1:n, 1:n) boolean adjacency matrix. // k index (node) of the next vertex to color. repeat{ nextvalue(k); // assign to x[k] a legal color. if(x[k]=0) then return; // no new color possible if(k=n) then write(x[1: n]; else mcoloring(k+1); } until(false) } </pre>	<pre> Algorithm NextValue(k){ //x[1],x[2],---x[k-1] have been assigned integer values in the range [1, m] repeat { x[k]=(x[k]+1)mod (m+1); //next highest color if(x[k]=0) then return; // all colors have been used. for j=1 to n do { if ((g[k,j]≠0) and (x[k]=x[j])) then break; } if(j=n+1) then return; //new color found } until(false) } </pre>



Adjacency matrix is

$$\begin{pmatrix} 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{pmatrix}$$



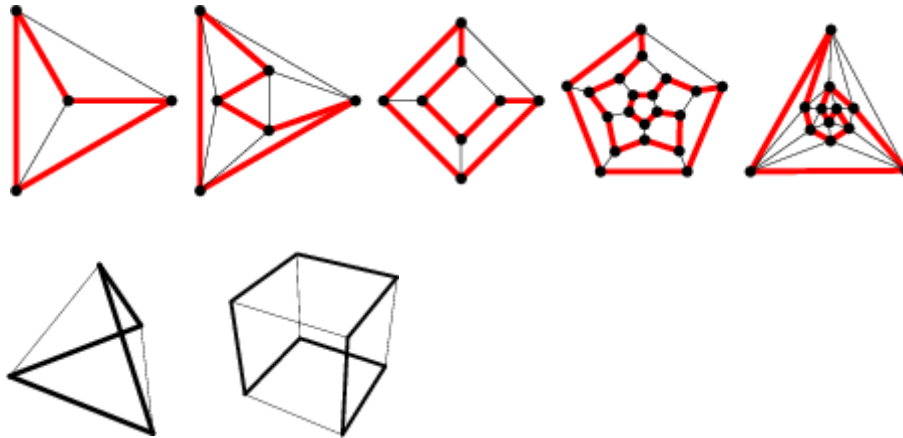
A 4 node graph and all possible 3 colorings



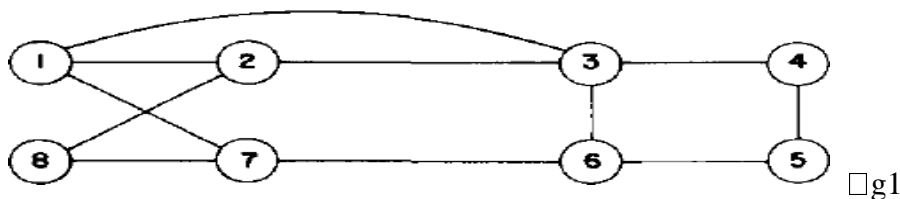
## Hamiltonian Cycles:

- **Def:** Let  $G=(V, E)$  be a connected graph with  $n$  vertices. A Hamiltonian cycle is a round trip path along  $n$ -edges of  $G$  that visits every vertex once & returns to its starting position.
- It is also called the Hamiltonian circuit.
- Hamiltonian circuit is a graph cycle (i.e., closed loop) through a graph that visits each node exactly once.
- A graph possessing a Hamiltonian cycle is said to be Hamiltonian graph.

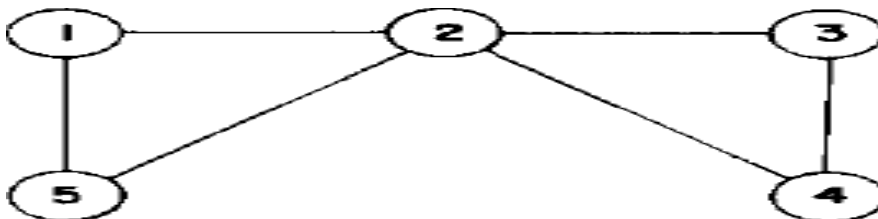
Example:



- In graph  $G$ , Hamiltonian cycle begins at some vertex  $v_1 \in G$  and the vertices of  $G$  are visited in the order  $v_1, v_2, \dots, v_{n+1}$ , then the edges  $(v_i, v_{i+1})$  are in  $E$ ,  $1 \leq i \leq n$ .



The above graph contains Hamiltonian cycle: 1,2,8,7,6,5,4,3,1



The above graph contains no Hamiltonian cycles.

- There is no known easy way to determine whether a given graph contains a Hamiltonian cycle.
- By using backtracking method, it can be possible
  - Backtracking algorithm, that finds all the Hamiltonian cycles in a graph.
  - The graph may be directed or undirected. Only distinct cycles are output.
  - From graph  $g_1$  backtracking solution vector =  $\{1, 2, 8, 7, 6, 5, 4, 3, 1\}$
  - The backtracking solution vector  $(x_1, x_2, \dots, x_n)$   
 $x_i \square i^{\text{th}}$  visited vertex of proposed cycle.

- By using backtracking we need to determine how to compute the set of possible vertices for  $x_k$  if  $x_1, x_2, x_3, \dots, x_{k-1}$  have already been chosen.

If  $k=1$  then  $x_1$  can be any of the  $n$ -vertices.

By using “NextValue” algorithm the recursive backtracking scheme to find all Hamiltonian cycles.

This algorithm is started by 1<sup>st</sup> initializing the adjacency matrix  $G[1:n, 1:n]$  then setting  $x[2:n]$  to zero &  $x[1]$  to 1, and then executing Hamiltonian (2)

Generating Next Vertex	Finding all Hamiltonian Cycles
<pre> Algorithm NextValue(k) { // x[1: k-1] is path of k-1 distinct vertices. // if x[k]=0, then no vertex has yet been assigned to x[k] Repeat{ X[k]=(x[k]+1) mod (n+1); //Next vertex If(x[k]=0) then return; If(G[x[k-1], x[k]]≠0) then { For j:=1 to k-1 do if(x[j]=x[k]) then break; //Check for distinctness If(j=k) then //if true , then vertex is distinct If((k&lt;n) or (k=n) and G[x[n], x[1]]≠0)) Then return ; } } Until (false); } </pre>	<pre> Algorithm Hamiltonian(k) { Repeat{ NextValue(k); //assign a legal next value to x[k] If(x[k]=0) then return; If(k=n) then write(x[1:n]); Else Hamiltonian(k+1); } until(false) } </pre>

## UNIT-V

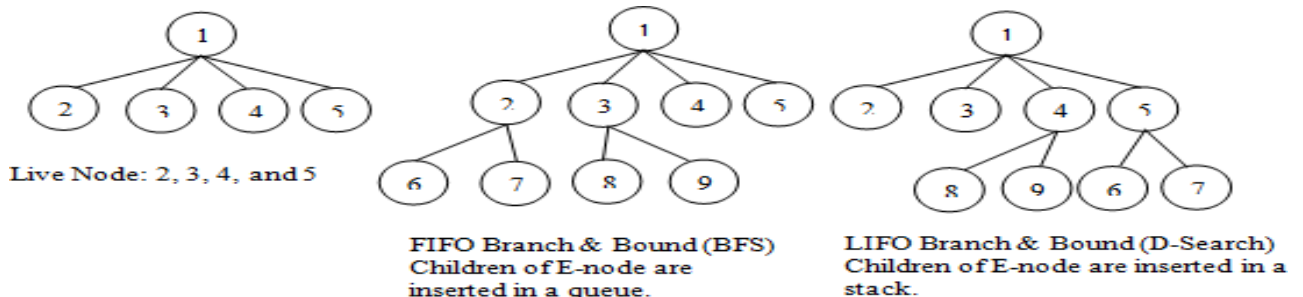
**Branch and Bound: General method, applications- Travelling sales person problem, 0/1 Knapsack problem- LC branch and Bound solution, FIFO branch and Bound solution.**

**NP-Hard and NP-Complete Problems: Basic concepts, Non deterministic algorithms, NP-Hard and NP Complete classes, NP-Hard problems, Cook's theorem.**

### Branch & Bound

Branch & Bound (B & B) is general algorithm (or Systematic method) for finding optimal solution of various optimization problems, especially in discrete and combinatorial optimization.

- The B&B strategy is very similar to backtracking in that a state space tree is used to solve a problem.
- The differences are that the B&B method
  - ✓ Does not limit us to any particular way of traversing the tree.
  - ✓ It is used only for optimization problem
  - ✓ It is applicable to a wide variety of discrete combinatorial problem.
- B&B is rather general optimization technique that applies where the greedy method & dynamic programming fail.
- It is much slower, indeed (truly), it often (rapidly) leads to exponential time complexities in the worst case.
- The term B&B refers to all state space search methods in which all children of the “E-node” are generated before any other “live node” can become the “E-node”
- ✓ **Live node** □ is a node that has been generated but whose children have not yet been generated.
- ✓ **E-node** □ is a live node whose children are currently being explored.
- ✓ **Dead node** □ is a generated node that is not to be expanded or explored any further. All children of a dead node have already been expanded.



- Two graph search strategies, BFS & D-search (DFS) in which the exploration of a new node cannot begin until the node currently being explored is fully explored.
- Both BFS & D-search (DFS) generalized to B&B strategies.
- ✓ **BFS** □ Like state space search will be called FIFO (First In First Out) search as the list of live nodes is “First-in-first-out” list (or queue).
- ✓ **D-search (DFS)** □ Like state space search will be called LIFO (Last In First Out) search as the list of live nodes is a “last-in-first-out” list (or stack).
- In backtracking, bounding function are used to help avoid the generation of sub-trees that do not contain an answer node.
- We will use 3-types of search strategies in branch and bound
  - 1) FIFO (First In First Out) search
  - 2) LIFO (Last In First Out) search

### 3) LC (Least Count) search

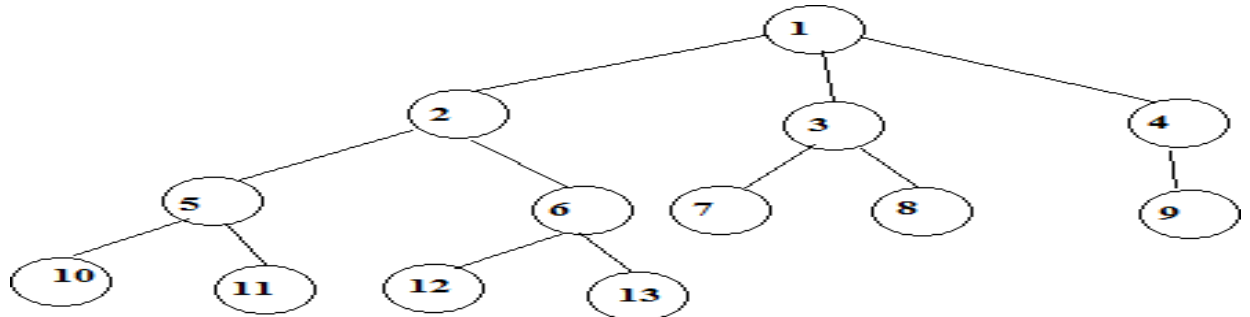
#### FIFO B&B:

FIFO Branch & Bound is a BFS.

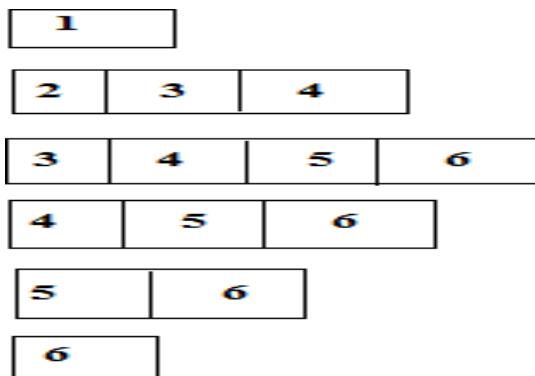
In this, children of E-Node (or Live nodes) are inserted in a queue.

Implementation of list of live nodes as a queue

- ✓ Least() □ Removes the head of the Queue
- ✓ Add() □ Adds the node to the end of the Queue



Assume that node '12' is an answer node in FIFO search, 1<sup>st</sup> we take E-node has '1'



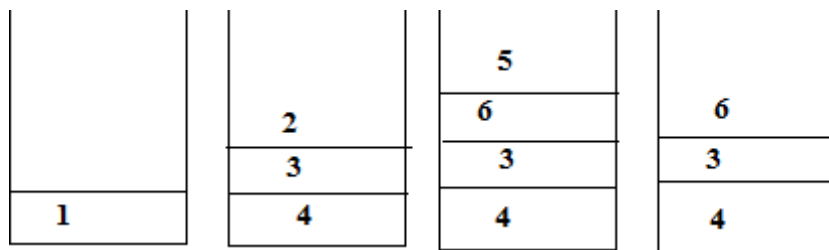
#### LIFO B&B:

LIFO Branch & Bound is a D-search (or DFS).

In this children of E-node (live nodes) are inserted in a stack

Implementation of List of live nodes as a stack

- ✓ Least() □ Removes the top of the stack
- ✓ ADD() □ Adds the node to the top of the stack.



#### Least Cost (LC) Search:

The selection rule for the next E-node in FIFO or LIFO branch and bound is sometimes "blind". i.e., the selection rule does not give any preference to a node that has a very good chance of getting the search to an answer node quickly.

The search for an answer node can often be speeded by using an "intelligent" ranking function. It is also called an approximate cost function " $\hat{C}$ ".

Expanded node (E-node) is the live node with the best  $\hat{C}$  value.

Branching: A set of solutions, which is represented by a node, can be partitioned into mutually (jointly or commonly) exclusive (special) sets. Each subset in the partition is represented by a child of the original node.

Lower bounding: An algorithm is available for calculating a lower bound on the cost of any solution in a given subset.

Each node X in the search tree is associated with a cost:  $\hat{C}(X)$

$C$ =cost of reaching the current node, X(E-node) from the root + The cost of reaching an answer node from X.

$$\hat{C} = g(X) + h(X).$$

### Example:

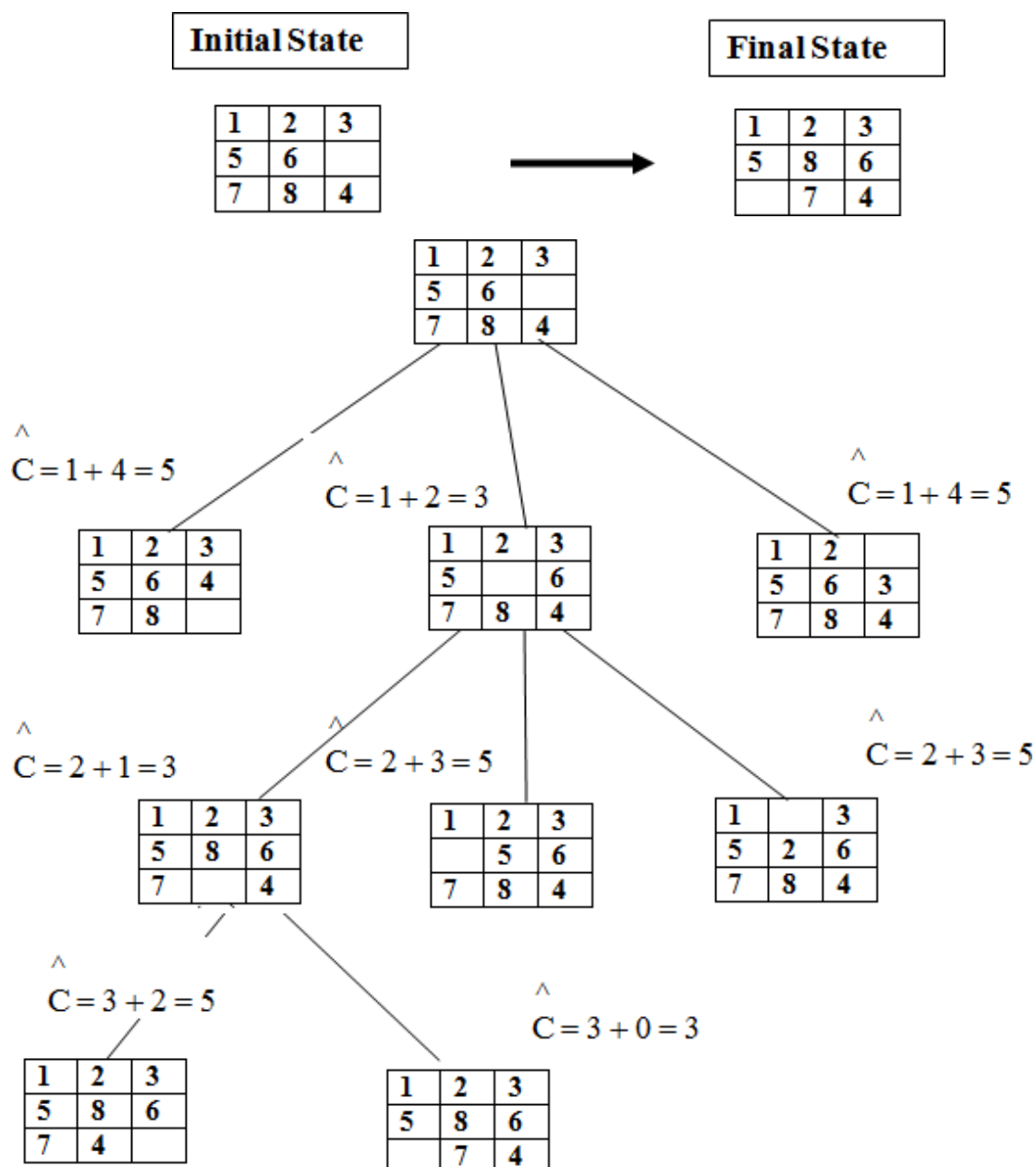
8-puzzle

Cost function:  $\hat{C} = g(x) + h(x)$

where  $h(x)$  = the number of misplaced tiles

and  $g(x)$  = the number of moves so far

Assumption: move one tile in any direction cost 1.



Note: In case of tie, choose the leftmost node.

### Travelling Salesman Problem:

Def:- Find a tour of minimum cost starting from a node S going through other nodes only once and returning to the starting point S.

Time Complexity of TSP for Dynamic Programming algorithm is  $O(n^2 2^n)$

B&B algorithms for this problem, the worst case complexity will not be any better than  $O(n^2 2^n)$  but good bounding functions will enable these B&B algorithms to solve some problem instances in much less time than required by the dynamic programming algorithm.

Let  $G=(V,E)$  be a directed graph defining an instance of TSP.

Let  $C_{ij}$  = cost of edge  $\langle i, j \rangle$

$C_{ij} = \infty$  if  $\langle i, j \rangle \notin E$

$|V|=n$  = total number of vertices.

Assume that every tour starts & ends at vertex 1.

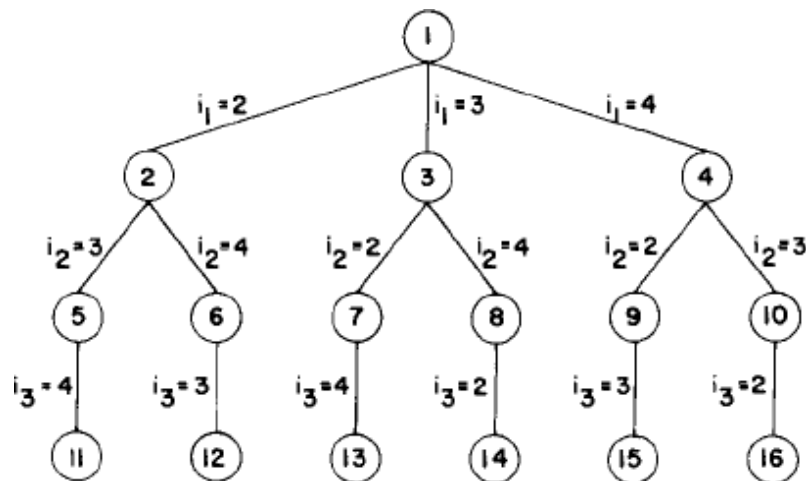
Solution Space  $S = \{1, \Pi, 1 / \Pi \text{ is a permutation of } (2, 3, 4, \dots, n)\}$  then  $|S|=(n-1)!$

The size of S reduced by restricting S

So that  $(1, i_1, i_2, \dots, i_{n-1}, 1) \in S$  iff  $\langle i_j, i_{j+1} \rangle \in E, 0 \leq j \leq n-1, i_0 = i_n = 1$

S can be organized into "State space tree".

Consider the following Example



State space tree for the travelling salesperson problem with  $n=4$  and  $i_0=i_4=1$

The above diagram shows tree organization of a complete graph with  $|V|=4$ .

Each leaf node 'L' is a solution node and represents the tour defined by the path from the root to L.

Node 12 represents the tour.

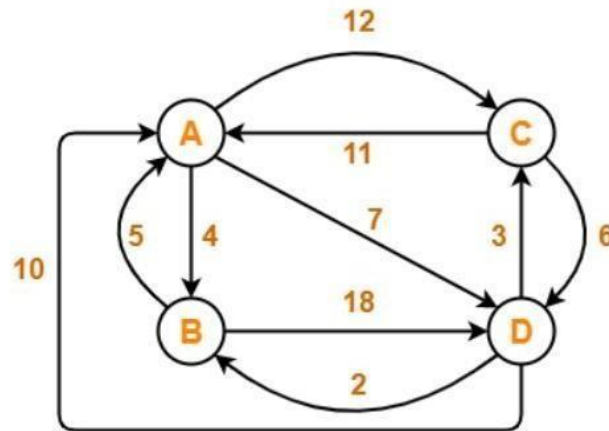
$i_0=1, i_1=2, i_2=4, i_3=3, i_4=1$

Node 14 represents the tour.

$i_0=1, i_1=3, i_2=4, i_3=2, i_4=1$ .

### TSP is solved by using LC Branch & Bound:

Solve Travelling Salesman Problem using Branch and Bound Algorithm in the following graph-



Write the initial cost matrix and reduce it-

	A	B	C	D
A	$\infty$	4	12	7
B	5	$\infty$	$\infty$	18
C	11	$\infty$	$\infty$	6
D	10	2	3	$\infty$

#### Rules

- To reduce a matrix, perform the row reduction and column reduction of the matrix separately.
- A row or a column is said to be reduced if it contains at least one entry '0' in it.

## Row Reduction-

Consider the rows of above matrix one by one.

If the row already contains an entry '0', then-

- There is no need to reduce that row.

If the row does not contains an entry '0', then-

- Reduce that particular row.
- Select the least value element from that row.
- Subtract that element from each element of that row.
- This will create an entry '0' in that row, thus reducing that row.

Following this, we have-

- Reduce the elements of row-1 by 4.
- Reduce the elements of row-2 by 5.
- Reduce the elements of row-3 by 6.
- Reduce the elements of row-4 by 2.

Performing this, we obtain the following row-reduced matrix-

	A	B	C	D
A	$\infty$	0	8	3
B	0	$\infty$	$\infty$	13
C	5	$\infty$	$\infty$	0
D	8	0	1	$\infty$

## Column Reduction-

Consider the columns of above row-reduced matrix one by one.

If the column already contains an entry '0', then-

- There is no need to reduce that column.

If the column does not contains an entry '0', then-



- Reduce that particular column.
- Select the least value element from that column.
- Subtract that element from each element of that column.
- This will create an entry '0' in that column, thus reducing that column.

Following this, we have-

- There is no need to reduce column-1.
- There is no need to reduce column-2.
- Reduce the elements of column-3 by 1.
- There is no need to reduce column-4.

Performing this, we obtain the following column-reduced matrix-

	A	B	C	D
A	$\infty$	0	7	3
B	0	$\infty$	$\infty$	13
C	5	$\infty$	$\infty$	0
D	8	0	0	$\infty$

Finally, the initial distance matrix is completely reduced.

Now, we calculate the cost of node-1 by adding all the reduction elements.

Cost(1)

= Sum of all reduction elements

= 4 + 5 + 6 + 2 + 1

= 18

## **Step-02:**

- We consider all other vertices one by one.
- We select the best vertex where we can land upon to minimize the tour cost.

## Choosing To Go To Vertex-B: Node-2 (Path A → B)

- From the reduced matrix of step-01,  $M[A,B] = 0$
- Set row-A and column-B to  $\infty$
- Set  $M[B,A] = \infty$

Now, resulting cost matrix is-

	A	B	C	D
A	$\infty$	$\infty$	$\infty$	$\infty$
B	$\infty$	$\infty$	$\infty$	13
C	5	$\infty$	$\infty$	0
D	8	$\infty$	0	$\infty$

Now,

- We reduce this matrix.
- Then, we find out the cost of node-02.

### Row Reduction-

- We can not reduce row-1 as all its elements are  $\infty$ .
- Reduce all the elements of row-2 by 13.
- There is no need to reduce row-3.
- There is no need to reduce row-4.

---

Performing this, we obtain the following row-reduced matrix-

	A	B	C	D
A	$\infty$	$\infty$	$\infty$	$\infty$
B	$\infty$	$\infty$	$\infty$	0
C	5	$\infty$	$\infty$	0
D	8	$\infty$	0	$\infty$

### Column Reduction-

- Reduce the elements of column-1 by 5.
- We can not reduce column-2 as all its elements are  $\infty$ .
- There is no need to reduce column-3.
- There is no need to reduce column-4.

Performing this, we obtain the following column-reduced matrix-

	A	B	C	D
A	$\infty$	$\infty$	$\infty$	$\infty$
B	$\infty$	$\infty$	$\infty$	0
C	0	$\infty$	$\infty$	0
D	3	$\infty$	0	$\infty$

Finally, the matrix is completely reduced.

Now, we calculate the cost of node-2.

Cost(2)

= Cost(1) + Sum of reduction elements + M[A,B]

= 18 + (13 + 5) + 0

= 36

### Choosing To Go To Vertex-C: Node-3 (Path A → C)

- From the reduced matrix of step-01,  $M[A,C] = 7$
- Set row-A and column-C to  $\infty$
- Set  $M[C,A] = \infty$

Now, resulting cost matrix is-

	A	B	C	D
A	$\infty$	$\infty$	$\infty$	$\infty$
B	0	$\infty$	$\infty$	13
C	$\infty$	$\infty$	$\infty$	0
D	8	0	$\infty$	$\infty$

Now,

- We reduce this matrix.
- Then, we find out the cost of node-03.

### Row Reduction-

- We can not reduce row-1 as all its elements are  $\infty$ .
- There is no need to reduce row-2.
- There is no need to reduce row-3.
- There is no need to reduce row-4.

Thus, the matrix is already row-reduced.

### Column Reduction-

- There is no need to reduce column-1.
- There is no need to reduce column-2.
- We can not reduce column-3 as all its elements are  $\infty$ .
- There is no need to reduce column-4.

Thus, the matrix is already column reduced.

Finally, the matrix is completely reduced.

Now, we calculate the cost of node-3.

Cost(3)

= Cost(1) + Sum of reduction elements + M[A,C]

= 18 + 0 + 7

= 25

### Choosing To Go To Vertex-D: Node-4 (Path A → D).

- From the reduced matrix of step-01,  $M[A,D] = 3$
- Set row-A and column-D to  $\infty$
- Set  $M[D,A] = \infty$

Now, resulting cost matrix is-

	A	B	C	D
A	$\infty$	$\infty$	$\infty$	$\infty$
B	0	$\infty$	$\infty$	$\infty$
C	5	$\infty$	$\infty$	$\infty$
D	$\infty$	0	0	$\infty$

Now,

- We reduce this matrix.
- Then, we find out the cost of node-04.

### **Row Reduction-**

- We can not reduce row-1 as all its elements are  $\infty$ .
- There is no need to reduce row-2.
- Reduce all the elements of row-3 by 5.
- There is no need to reduce row-4.

Performing this, we obtain the following row-reduced matrix-

	A	B	C	D
A	$\infty$	$\infty$	$\infty$	$\infty$
B	0	$\infty$	$\infty$	$\infty$
C	0	$\infty$	$\infty$	$\infty$
D	$\infty$	0	0	$\infty$

### **Column Reduction-**

- There is no need to reduce column-1.
- There is no need to reduce column-2.
- There is no need to reduce column-3.
- We can not reduce column-4 as all its elements are  $\infty$ .

Thus, the matrix is already column-reduced.

Finally, the matrix is completely reduced.

Now, we calculate the cost of node-4.

Cost(4)

= Cost(1) + Sum of reduction elements + M[A,D]

= 18 + 5 + 3

= 26

Thus, we have-

- Cost(2) = 36 (for Path A  $\rightarrow$  B)
- Cost(3) = 25 (for Path A  $\rightarrow$  C)

- $\text{Cost}(4) = 26$  (for Path  $A \rightarrow D$ )

We choose the node with the lowest cost.

Since cost for node-3 is lowest, so we prefer to visit node-3.

Thus, we choose node-3 i.e. path  $A \rightarrow C$ .

### **Step-03:**

We explore the vertices B and D from node-3.

We now start from the cost matrix at node-3 which is-

	A	B	C	D
A	$\infty$	$\infty$	$\infty$	$\infty$
B	0	$\infty$	$\infty$	13
C	$\infty$	$\infty$	$\infty$	0
D	8	0	$\infty$	$\infty$

$$\text{Cost}(3) = 25$$

### **Choosing To Go To Vertex-B: Node-5 (Path $A \rightarrow C \rightarrow B$ )**

- From the reduced matrix of step-02,  $M[C,B] = \infty$
- Set row-C and column-B to  $\infty$
- Set  $M[B,A] = \infty$

Now, resulting cost matrix is-

	A	B	C	D
A	$\infty$	$\infty$	$\infty$	$\infty$
B	$\infty$	$\infty$	$\infty$	13
C	$\infty$	$\infty$	$\infty$	$\infty$
D	8	$\infty$	$\infty$	$\infty$

Now,

- We reduce this matrix.
- Then, we find out the cost of node-5.

### Row Reduction-

- We can not reduce row-1 as all its elements are  $\infty$ .
- Reduce all the elements of row-2 by 13.
- We can not reduce row-3 as all its elements are  $\infty$ .
- Reduce all the elements of row-4 by 8.

Performing this, we obtain the following row-reduced matrix-

	A	B	C	D
A	$\infty$	$\infty$	$\infty$	$\infty$
B	$\infty$	$\infty$	$\infty$	0
C	$\infty$	$\infty$	$\infty$	$\infty$
D	0	$\infty$	$\infty$	$\infty$

### Column Reduction-

- There is no need to reduce column-1.
- We can not reduce column-2 as all its elements are  $\infty$ .
- We can not reduce column-3 as all its elements are  $\infty$ .
- There is no need to reduce column-4.

Thus, the matrix is already column reduced.

Finally, the matrix is completely reduced.

Now, we calculate the cost of node-5.



$$= \text{cost}(3) + \text{Sum of reduction elements} + M[C,B]$$

$$= 25 + (13 + 8) + \infty$$

$$= \infty$$

### **Choosing To Go To Vertex-D: Node-6 (Path A → C → D)**

- From the reduced matrix of step-02,  $M[C,D] = \infty$
- Set row-C and column-D to  $\infty$
- Set  $M[D,A] = \infty$

Now, resulting cost matrix is-

	A	B	C	D
A	$\infty$	$\infty$	$\infty$	$\infty$
B	0	$\infty$	$\infty$	$\infty$
C	$\infty$	$\infty$	$\infty$	$\infty$
D	$\infty$	0	$\infty$	$\infty$

Now,

- We reduce this matrix.
- Then, we find out the cost of node-6.

### **Row Reduction-**

- We can not reduce row-1 as all its elements are  $\infty$ .
- There is no need to reduce row-2.
- We can not reduce row-3 as all its elements are  $\infty$ .
- We can not reduce row-4 as all its elements are  $\infty$ .

Thus, the matrix is already row reduced.

### **Column Reduction-**

- There is no need to reduce column-1.
- We can not reduce column-2 as all its elements are  $\infty$ .
- We can not reduce column-3 as all its elements are  $\infty$ .
- We can not reduce column-4 as all its elements are  $\infty$ .

Thus, the matrix is already column reduced.

Finally, the matrix is completely reduced.

Now, we calculate the cost of node-6.

Cost(6)

= cost(3) + Sum of reduction elements + M[C,D]

= 25 + 0 + 0

= 25

**Thus, we have-**

- Cost(5) =  $\infty$  (for Path A  $\rightarrow$  C  $\rightarrow$  B)
- Cost(6) = 25 (for Path A  $\rightarrow$  C  $\rightarrow$  D)

We choose the node with the lowest cost.

Since cost for node-6 is lowest, so we prefer to visit node-6.

Thus, we choose node-6 i.e. path **C  $\rightarrow$  D**.

### **Step-04:**

We explore vertex B from node-6.

We start with the cost matrix at node-6 which is-

	A	B	C	D
A	$\infty$	$\infty$	$\infty$	$\infty$
B	0	$\infty$	$\infty$	$\infty$
C	$\infty$	$\infty$	$\infty$	$\infty$
D	$\infty$	0	$\infty$	$\infty$

Cost(6) = 25

### Choosing To Go To Vertex-B: Node-7 (Path A $\rightarrow$ C $\rightarrow$ D $\rightarrow$ B).

- From the reduced matrix of step-03,  $M[D,B] = 0$
- Set row-D and column-B to  $\infty$
- Set  $M[B,A] = \infty$

Now, resulting cost matrix is-

	A	B	C	D
A	$\infty$	$\infty$	$\infty$	$\infty$
B	$\infty$	$\infty$	$\infty$	$\infty$
C	$\infty$	$\infty$	$\infty$	$\infty$
D	$\infty$	$\infty$	$\infty$	$\infty$

Now,

- We reduce this matrix.
- Then, we find out the cost of node-7.

### Row Reduction-

- We can not reduce row-1 as all its elements are  $\infty$ .
- We can not reduce row-2 as all its elements are  $\infty$ .
- We can not reduce row-3 as all its elements are  $\infty$ .
- We can not reduce row-4 as all its elements are  $\infty$ .

## Column Reduction-

- We can not reduce column-1 as all its elements are  $\infty$ .
- We can not reduce column-2 as all its elements are  $\infty$ .
- We can not reduce column-3 as all its elements are  $\infty$ .
- We can not reduce column-4 as all its elements are  $\infty$ .

Thus, the matrix is already column reduced.

Finally, the matrix is completely reduced.

All the entries have become  $\infty$ .

Now, we calculate the cost of node-7.

Cost(7)

= cost(6) + Sum of reduction elements +  $M[D,B]$

= 25 + 0 + 0

= 25

Thus,

- Optimal path is: **A  $\rightarrow$  C  $\rightarrow$  D  $\rightarrow$  B  $\rightarrow$  A**
- Cost of Optimal path = **25 units**

### O/1 Knapsack Problem

**What is Knapsack Problem:** Knapsack problem is a problem in combinatorial optimization, Given a set of items, each with a mass & a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit & the total value is as large as possible.

**O-1 Knapsack Problem can formulate as.** Let there be n items, **Z<sub>1</sub> to Z<sub>n</sub>** where **Z<sub>i</sub>** has value **P<sub>i</sub>** & weight **w<sub>i</sub>**. The maximum weight that can carry in the bag is m.

All values and weights are non negative.

Maximize the sum of the values of the items in the knapsack, so that sum of the weights must be less than the knapsack's capacity m.

The formula can be stated as

$$\begin{aligned} & \textbf{maximize} \quad \sum_{1 \leq i \leq n} p_i x_i \\ & \textbf{subject to} \quad \sum_{1 \leq i \leq n} w_i x_i \leq M \\ & X_i = 0 \text{ or } 1 \quad 1 \leq i \leq n \end{aligned}$$

**To solve 0/1 knapsack problem using B&B:**

➤ Knapsack is a maximization problem

- Replace the objective function  $\sum p_i x_i$  by the function  $-\sum p_i x_i$  to make it into a minimization problem
- The modified knapsack problem is stated as

$$\begin{aligned} & \textbf{Minimize} \quad -\sum_{i=1}^n p_i x_i \\ & \textbf{subject to} \quad \sum_{i=1}^n w_i x_i < m, \\ & \quad \quad \quad x_i \in \{0, 1\}, 1 \leq i \leq n \end{aligned}$$

□ Fixed tuple size solution space:

○ Every leaf node in state space tree represents an answer for which

$$\sum_{1 \leq i \leq n} w_i x_i \leq m \quad \text{is an answer node; other leaf nodes are infeasible}$$

○ For optimal solution, define

$$c(x) = -\sum_{1 \leq i \leq n} p_i x_i \quad \text{for every answer node } x$$

□ For infeasible leaf nodes,  $c(x) = \infty$

□ For non leaf nodes

$$c(x) = \min\{c(\text{lchild}(x)), c(\text{rchild}(x))\}$$

□ Define two functions  $\hat{c}(x)$  and  $u(x)$  such that for every node  $x$ ,

$$\hat{c}(x) \leq c(x) \leq u(x)$$

- Computing  $c^*(\cdot)$  and  $u(\cdot)$

Let  $x$  be a node at level  $j$ ,  $1 \leq j \leq n + 1$

Cost of assignment:  $-\sum_{1 \leq i < j} p_i x_i$

$c(x) \leq -\sum_{1 \leq i < j} p_i x_i$

We can use  $u(x) = -\sum_{1 \leq i < j} p_i x_i$

Using  $q = -\sum_{1 \leq i < j} p_i x_i$ , an improved upper bound function  $u(x)$  is

$$u(x) = \text{ubound}(q, \sum_{1 \leq i < j} w_i x_i, j - 1, m)$$

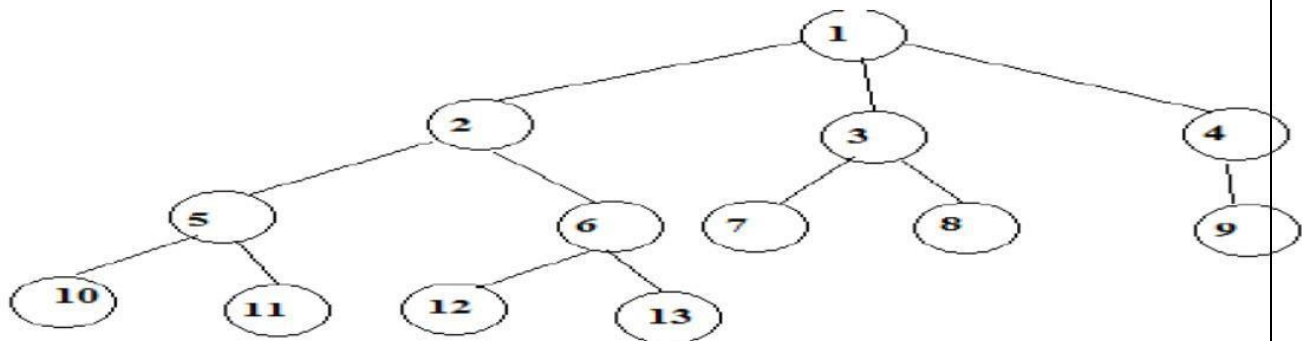
```

Algorithm ubound ( cp, cw, k, m )
{
// Input:  cp: Current profit total
// Input:  cw: Current weight total
// Input:  k:  Index of last removed item
// Input:  m:  Knapsack capacity
b=cp; c=cw;
for i:=k+1 to n do{
    if(c+w[i] ≤ m) then {
        c:=c+w[i]; b=b+p[i];
    }
}
return b;
}

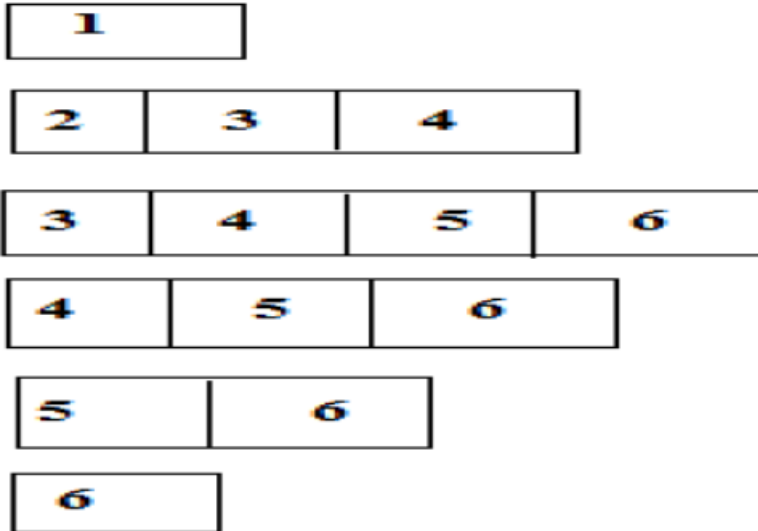
```

### FIFO Branch and bound

- FIFO Branch and Bound is a BFS.
- In FIFO Branch and Bound , children of E-Node (or Live nodes) are inserted in a queue.
- Implementation of list of live nodes as a queue
  - Least()** Removes the head of the Queue
  - Add()** Adds the node to the end of the Queue



Assume that node '12' is an answer node in FIFO search, 1<sup>st</sup> we take E-node has '1'



FIFO stands for First In First Out, it follows the BFS technique. As like BFS, In this FIFO Branch and Bound reach a node to find upper bound and lower bound values and next it reaches another node in the same level of state space tree. And next reaches the node which is presented in the queue. FIFO Branch and Bound applications are knapsack problem and travelling salesman problem. This FIFO is one of the techniques of LC- search. When implementing a FIFO branch and bound algorithm, here reach each and every node in the state space tree. Here can find out more than one solution for the problem, if the problem has the multiple paths of solution. Unlike LCBB, The FIFO BB needs more memory to show each node in the state space tree. The main principle of the FIFO is reaching every node which is generated by solution process in the state space tree.

### LCBB

For speeding up the search process here need to intelligent ranking function for live nodes. Each time, the next E- node is selected on the basis of this ranking function. For this ranking function additional computation (normally cost) is needed to reach the answer node from the live node. LC-search is a kind of search in which least cost involved for reaching to answer node. At each E-node the probability of being an answer node is checked. BFS and D-search are special cases of LC search. An LC search with bounding functions is known as LC Branch and Bound search. The applications of LC Branch and Bound are 0/1 Knapsack problem and Travelling salesman problem.

The search mainly based on the state space tree. To find upper bound value and lower bound value at the each node to get the ranking value to identify which node is least cost.

### Basic concepts:

**NP**, Nondeterministic Polynomial time

The problems has best algorithms for their solutions have “Computing times”, that cluster into two groups

Group 1	Group 2
<ul style="list-style-type: none"> <li>&gt; Problems with solution time bound by a polynomial of a small degree.</li> <li>&gt; It also called “Tractable Algorithms”</li> </ul>	<ul style="list-style-type: none"> <li>&gt; Problems with solution times not bound by polynomial (simply non polynomial )</li> <li>&gt; These are hard or intractable problems</li> </ul>

<ul style="list-style-type: none"> <li>&gt; Most Searching &amp; Sorting algorithms are polynomial time algorithms</li> <li>&gt; <b>Ex:</b> Ordered Search (<b><math>O(\log n)</math></b>), Polynomial evaluation <b><math>O(n)</math></b>  Sorting <b><math>O(n \cdot \log n)</math></b></li> </ul>	<ul style="list-style-type: none"> <li>&gt; None of the problems in this group has been solved by any polynomial time algorithm</li> <li>&gt; <b>Ex:</b> Traveling Sales Person <b><math>O(n^2 \cdot 2^n)</math></b>  Knapsack <b><math>O(2^{n/2})</math></b></li> </ul>
--	--

No one has been able to develop a polynomial time algorithm for any problem in the 2<sup>nd</sup> group (i.e., group 2)

So, it is compulsory and finding algorithms whose computing times are greater than polynomial very quickly because such vast amounts of time to execute that even moderate size problems cannot be solved.

### **Theory of NP-Completeness:**

Show that many of the problems with no polynomial time algorithms are computationally related.

There are two classes of non-polynomial time problems

1. NP-Hard
2. NP-Complete



**NP Complete Problem:** A problem that is NP-Complete can be solved in polynomial time if and only if (iff) all other NP-Complete problems can also be solved in polynomial time.

**NP-Hard:** Problem can be solved in polynomial time then all NP-Complete problems can be solved in polynomial time.

All NP-Complete problems are NP-Hard but some NP-Hard problems are not known to be NP-Complete.

### Non deterministic Algorithms:

Algorithms with the property that the result of every operation is uniquely defined are termed as deterministic algorithms. Such algorithms agree with the way programs are executed on a computer.

Algorithms which contain operations whose outcomes are not uniquely defined but are limited to a specified set of possibilities. Such algorithms are called nondeterministic algorithms.

The machine executing such operations is allowed to choose any one of these outcomes subject to a termination condition to be defined later.

To specify nondeterministic algorithms, there are 3 new functions.

Choice(S) arbitrarily chooses one of the elements of sets S

Failure () Signals an Unsuccessful completion

Success () Signals a successful completion.

### **Example for Non Deterministic algorithms:**

<pre> <b>Algorithm Search(x){</b> //Problem is to search an element x  //output J, such that A[J]=x; or J=0 if x is not in A J:=Choice(1,n); if( A[J]:=x) then {                 Write(J);                 Success();             } else{                 write(0);                 failure();             }         }     </pre>	<p>Whenever there is a set of choices that leads to a successful completion then one such set of choices is always made and the algorithm terminates.</p> <p>A Nondeterministic algorithm terminates unsuccessfully if and only if (iff) there exists no set of choices leading to a successful signal.</p>

Nondeterministic Knapsack algorithm	
<b>Algorithm DKP</b> (p, w, n, m, r, x){ W:=0; P:=0; for i:=1 to n do{ x[i]:=choice(0, 1); W:=W+x[i]*w[i]; P:=P+x[i]*p[i]; } if((W>m) or (P<r)) then Failure(); else Success(); }	p. given Profits w. given Weights n. Number of elements (number of p or w) m. Weight of bag limit P. Final Profit W. Final weight

### The Classes NP-Hard & NP-Complete:

For measuring the complexity of an algorithm, we use the input length as the parameter. For example, An algorithm A is of polynomial complexity  $p()$  such that the computing time of A is  $O(p(n))$  for every input of size n.

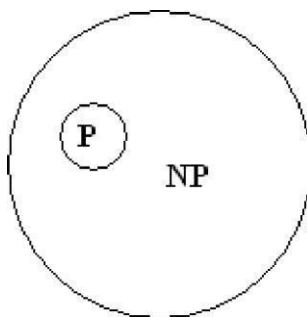
**Decision problem/ Decision algorithm:** Any problem for which the answer is either zero or one is decision problem. Any algorithm for a decision problem is termed a decision algorithm.

**Optimization problem/ Optimization algorithm:** Any problem that involves the identification of an optimal (either minimum or maximum) value of a given cost function is known as an optimization problem. An optimization algorithm is used to solve an optimization problem.

**P** is the set of all decision problems solvable by deterministic algorithms in polynomial time.

**NP** is the set of all decision problems solvable by nondeterministic algorithms in polynomial time.

Since deterministic algorithms are just a special case of nondeterministic, by this we concluded that **P  $\subseteq$  NP**



Commonly believed relationship between P & NP

The most famous unsolvable problems in Computer Science is Whether  $P=NP$  or  $P \neq NP$   
In considering this problem, s.cook formulated the following question.

If there any single problem in NP, such that if we showed it to be in 'P' then that would imply that  $P=NP$ .

Cook answered this question with

**Theorem:** Satisfiability is in P if and only if (iff)  $P=NP$

-)Notation of Reducibility

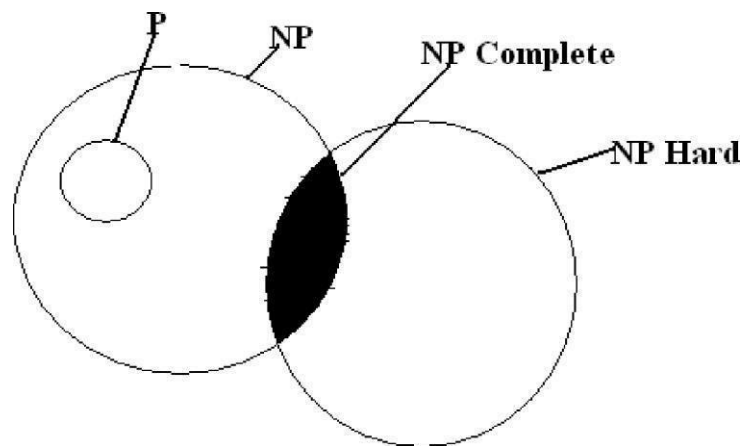
Let  $L_1$  and  $L_2$  be problems, Problem  $L_1$  reduces to  $L_2$  (written  $L_1 \alpha L_2$ ) iff there is a way to solve  $L_1$  by a deterministic polynomial time algorithm using a deterministic algorithm that solves  $L_2$  in polynomial time

This implies that, if we have a polynomial time algorithm for  $L_2$ , Then we can solve  $L_1$  in polynomial time.

Here  $\alpha$  is a transitive relation i.e.,  $L_1 \alpha L_2$  and  $L_2 \alpha L_3$  then  $L_1 \alpha L_3$

A problem  $L$  is NP-Hard if and only if (iff) satisfiability reduces to  $L$  i.e., **Satisfiability  $\alpha L$**

A problem  $L$  is NP-Complete if and only if (iff)  $L$  is NP-Hard and  $L \in NP$



Commonly believed relationship among P, NP, NP-Complete and NP-Hard

Most natural problems in NP are either in P or NP-complete.

**Examples of NP-complete problems:**

- > Packing problems: SET-PACKING, INDEPENDENT-SET.
- > Covering problems: SET-COVER, VERTEX-COVER.
- > Sequencing problems: HAMILTONIAN-CYCLE, TSP.
- > Partitioning problems: 3-COLOR, CLIQUE.
- > Constraint satisfaction problems: SAT, 3-SAT.
- > Numerical problems: SUBSET-SUM, PARTITION, KNAPSACK.

*The MIN VERTEX-COVER problem is NP-hard.*

*Proof. It is NP-hard since its decision version is NP-complete.  $\square$*

*There are two problems closely related to the MIN VERTEX-COVER problem.*

*The first one is the MAX INDEPENDENT SET problem: Given a graph  $G = (V, E)$ , find an independent set with maximum cardinality. Here, an independent set is a subset of vertices such that no edge exists between two vertices in the subset. A subset of vertices is an independent set if and only if its complement is a vertex cover. In fact, from the definition, every edge*

*has to have at least one endpoint in the complement of an independent set, which means that the complement of an independent set must be a vertex cover. Conversely, if the complement of a vertex subset  $I$  is a vertex cover, then every edge has an endpoint not in  $I$  and hence  $I$  is independent. Furthermore, it is easy to see that A vertex subset  $I$  is the maximum independent set if and only if the complement of  $I$  is the minimum vertex cover.*

*The second one is the MAX CLIQUE problem: Given a graph  $G = (V, E)$ , find a clique with maximum size. Here, a clique is a complete subgraph and its size is the number of vertices in the clique. Let  $\bar{G}$  be the complementary graph of  $G$ , that is,  $\bar{G} = (V, \bar{E})$  where  $\bar{E}$  is the complement of  $E$ . Then a subgraph on a vertex subset  $I$  is a clique in  $G$  if and only if  $I$  is an independent set in  $\bar{G}$ . Thus, a subgraph on a vertex subset  $I$  is a maximum clique if and only if  $I$  is a maximum independent set in  $\bar{G}$ .*

*From their relationship, we already see that the MIN VERTEX COVER problem, the MAX INDEPENDENT SET problem and the MAX CLIQUE problem have the same complexity in term of computing exact optimal solutions. However, it may be interesting to point out that they have different computational complexities in term of computing approximation solutions.*

*The KNAPSACK problem is NP-hard.*

*Proof.* For each instance of the SUBSUM problem, which consists of  $n + 1$  positive integers  $a_1, a_2, \dots, a_n$  and  $L$ , consider the following KNAPSACK problem:

$$\begin{array}{ll} \max & a_1x_1 + a_2x_2 + \dots + a_nx_n \\ \text{subject to} & a_1x_1 + a_2x_2 + \dots + a_nx_n \leq L. \end{array}$$

Clearly, there exists a subset  $N_1$  of  $[n]$  such that  $\sum_{i \in N_1} a_i = L$  if and only if above corresponding KNAPSACK problem has an optimal solution with objective function value  $L$ . Therefore, if the KNAPSACK problem is polynomial-time solvable, so is the SUBSUM problem.  $\square$

Let  $opt(k, S)$  be the objective function value of an optimal solution of the following problem:

$$\begin{array}{ll} \max & c_1x_1 + c_2x_2 + \dots + c_kx_k \\ \text{subject to} & a_1x_1 + a_2x_2 + \dots + a_kx_k \\ & x_1, x_2, \dots, x_k \in \{0, 1\}. \end{array}$$

Then

$$opt(k, S) = \max(opt(k-1, S), c_k + opt(k-1, S - a_k)).$$

This recursive formula gives a dynamic programming to solve the KNAPSACK problem within  $O(nS)$  time. This is a pseudopolynomial-time algorithm, not a polynomial-time algorithm because the input size of  $S$  is  $\lceil \log_2 S \rceil$ , not  $S$ .

An optimization problem is said to have PTAS (polynomial-time approximation scheme) if for any  $\varepsilon > 0$ , there is a polynomial-time  $(1 + \varepsilon)$ -approximation for the problem. The KNAPSACK problem has a PTAS. To construct a PTAS, we need to design another pseudopolynomial-time algorithm for the KNAPSACK problem.

Let  $c(i, j)$  denote a subset of index set  $\{1, \dots, i\}$  such that

- (a)  $\sum_{k \in c(i, j)} c_k = j$  and
- (b)  $\sum_{k \in c(i, j)} s_k = \min\{\sum_{k \in I} s_k \mid \sum_{k \in I} c_k = j, I \subseteq \{1, \dots, i\}\}$ .

If no index subset satisfies (a), then we say that  $c(i, j)$  is undefined, or write  $c(i, j) = nil$ . Clearly,  $opt = \max\{j \mid c(n, j) \neq nil \text{ and } \sum_{k \in c(i, j)} s_k \leq S\}$ . Therefore, it suffices to compute all  $c(i, j)$ . The following algorithm is designed with this idea.



## The 2nd Exact Algorithm for Knapsack

Initially, compute  $c(1, j)$  for  $j = 0, \dots, c_{sum}$  by setting

$$c(1, j) := \begin{cases} \emptyset & \text{if } j = 0, \\ \{1\} & \text{if } j = c_1, \\ nil & \text{otherwise,} \end{cases}$$

where  $c_{sum} = \sum_{i=1}^n c_i$ .

Next, compute  $c(i, j)$  for  $i \geq 2$  and  $j = 0, \dots, c_{sum}$ .

**for**  $i = 2$  **to**  $n$  **do**

**for**  $j = 0$  **to**  $c_{sum}$  **do**

**case 1**  $[c(i-1, j-c_i) = nil]$

            set  $c(i, j) = c(i-1, j)$

**case 2**  $[c(i-1, j-c_i) \neq nil]$

            and  $[c(i-1, j) = nil]$

            set  $c(i, j) = c(i-1, j-c_i) \cup \{i\}$

**case 3**  $[c(i-1, j-c_i) \neq nil]$

            and  $[c(i-1, j) \neq nil]$

**if**  $[\sum_{k \in c(i-1, j)} s_k > \sum_{k \in c(i-1, j-c_i)} s_k + s_i]$

**then**  $c(i, j) := c(i-1, j-c_i) \cup \{i\}$

**else**  $c(i, j) := c(i-1, j);$

Finally, set  $opt = \max\{j \mid c(n, j) \neq nil \text{ and } \sum_{k \in c(i, j)} s_k \leq S\}$ .

This algorithm computes the exact optimal solution for KNAPSACK with running time  $O(n^3 M \log(MS))$  where  $M = \max_{1 \leq k \leq n} c_k$ , because the algorithm contains two loops, the outside loop runs in  $O(n)$  time, the inside loop runs in  $O(nM)$  time, and the central part runs in  $O(n \log(MS))$  time. This is a pseudopolynomial-time algorithm because the input size of  $M$  is  $\log_2 M$ , the running time is not a polynomial with respect to input size.

*The TRAVELING SALESMAN problem is NP-hard.*

*Proof.* A polynomial-time many-one reduction has been constructed from the HAMILTONIAN CYCLE problem to the TRAVELING SALESMAN problem.

□

The LONGEST PATH problem is a maximization problem as follows: Given a graph  $G = (V, E)$  with positive edge length  $c : E \rightarrow R^+$ , and two vertices  $s$  and  $t$ , find a longest simple path between  $s$  and  $t$ .

*The LONGEST PATH problem is NP-hard.*

*Proof.* We will construct a polynomial-time many-one reduction from the HAMILTONIAN CYCLE problem to the decision version of the LONGEST PATH problem as follows: *Given a graph  $G = (V, E)$  with positive edge length  $c : E \rightarrow \mathbb{R}^+$ , two vertices  $s$  and  $t$ , and an integer  $K > 0$ , is there a simple path between  $s$  and  $t$  with length at least  $K$ ?*

Let graph  $G = (V, E)$  be an input of the HAMILTONIAN CYCLE problem. Choose a vertex  $u \in V$ . We make a copy of  $u$  by adding a new vertex  $u'$  and connecting  $u'$  to all neighbors of  $u$ . Add two new edges  $(u, s)$  and  $(u', t)$ . Obtained graph is denoted by  $f(G)$ . Let  $K = |V| + 2$ . We show that  $G$  contains a Hamiltonian cycle if and only if  $f(G)$  contains a simple path between  $s$  and  $t$  with length at most  $K$ .

**Cook's Theorem:** States that satisfiability is in P if and only if  $P=NP$  If

$P=NP$  then satisfiability is in P

If satisfiability is in P, then  $P=NP$

To do this

> A-) Any polynomial time nondeterministic decision algorithm.

I-) Input of that algorithm

Then formula  $Q(A, I)$ , Such that  $Q$  is satisfiable iff 'A' has a successful termination with Input **I**.

> If the length of 'I' is 'n' and the time complexity of A is  $p(n)$  for some polynomial  $p()$  then length of Q is  $O(p^3(n) \log n) = O(p^4(n))$

The time needed to construct Q is also  $O(p^3(n) \log n)$ .

> A deterministic algorithm 'Z' to determine the outcome of 'A' on any input 'I'  
Algorithm Z computes 'Q' and then uses a deterministic algorithm for the satisfiability problem to determine whether 'Q' is satisfiable.

> If  $O(q(m))$  is the time needed to determine whether a formula of length 'm' is satisfiable then the complexity of 'Z' is  $O(p^3(n) \log n + q(p^3(n) \log n))$ .

> If satisfiability is 'p', then 'q(m)' is a polynomial function of 'm' and the complexity of 'Z' becomes ' $O(r(n))$ ' for some polynomial 'r()'.

> Hence, if satisfiability is in **p**, then for every nondeterministic algorithm A in **NP**, we can obtain a deterministic Z in **p**.

By this we show that satisfiability is in **p** then  $P=NP$