SREYAS INSTITUTE OF ENGINEERING & TECHNOLOGY

(Approved by A.I.C.T.E.New Delhi., Affiliated to JNTUH Hyderabad)

Nagole, Hyderabad- 500 068.

Department of Computer Science & Engineering

LECTURE NOTES

ON

DESIGN AND ANALYSIS OF ALGORITHMS

CONTENTS

CHAPTER 4:BACKTRACKING AND BRANCH AND BOUND

Backtracking
The General Method
The 8 Queens Problem
Sum Of Subsets Problem
Graph Coloring
Hamiltonian Cycles
Branch And Bound
The General Method
0/1 Knapsack Problem
Least Cost Branch And Bound Solution
First In First Out Branch And Bound Solution
Travelling Salesperson Problem

CHAPTER 5: NP-HARD AND NP-COMPLETE PROBLEMS

5. Basic Concepts
Non-Deterministic Algorithms
The Classes NP - Hard And NP
NP Hard Problems
Clique Decision Problem
Chromatic Number Decision Problem
Cook's Theorem

Unit-1

Introduction

ALGORITHM:

Algorithm was first time proposed a purshian mathematician Al-Chwarizmi in 825 AD. According to web star dictionary, algorithm is a special method to represent the procedure to solve given problem.

OR

An Algorithm is any well-defined computational procedure that takes some value or set of values as Input and produces a set of values or some value as output. Thus algorithm is a sequence of computational steps that transforms the input into the output.

Formal Definition:

An Algorithm is a finite set of instructions that, if followed, accomplishes a particular task. In addition, all algorithms should satisfy the following criteria.

1. **Input**. Zero or more quantities are externally supplied.
2. **Output**. At least one quantity is produced.
3. **Definiteness**. Each instruction is clear and unambiguous.
4. **Finiteness**. If we trace out the instructions of an algorithm, then for all cases, the algorithm terminates after a finite number of steps.
5. **Effectiveness**. Every instruction must very basic so that it can be carried out, in principle, by a person using only pencil & paper.

Areas of study of Algorithm:

- *How to device or design an algorithm–* It includes the study of various design techniques and helps in writing algorithms using the existing design techniques like divide and conquer.
- *How to validate an algorithm–* After the algorithm is written it is necessary to check the correctness of the algorithm i.e for each input correct output is produced, known as algorithm validation. The second phase is writing a program known as program proving or program verification.
- *How to analysis an algorithm–*It is known as analysis of algorithms or performance analysis, refers to the task of calculating time and space complexity of the algorithm.
- How to test a program – It consists of two phases . 1. debugging is detection and correction of errors. 2. Profiling or performance measurement is the actual amount of time required by the program to compute the result.

Algorithm Specification:

Algorithm can be described in three ways.
   1. Natural language like English:

2. Graphic representation called flowchart:

This method will work well when the algorithm is small& simple.

3. Pseudo-code Method:

In this method, we should typically describe algorithms as program, which resembles language like Pascal &algol.

Pseudo-Code for writing Algorithms:

1. Comments begin with // and continue until the end of line.
2. Blocks are indicated with matching braces {and}.
3. An identifier begins with a letter. The data types of variables are not explicitly declared.
4. Compound data types can be formed with records. Here is an example,

    Node. Record
    {
    data type – 1 data-1; .
    data type – n data – n;
    node * link;
    }

    Here link is a pointer to the record type node. Individual data items of a record can be accessed with → and period.
5. Assignment of values to variables is done using the assignment statement.

    <Variable>:= <expression>;

6. There are two Boolean values TRUE and FALSE.

    Logical Operators   AND, OR, NOT

    Relational Operators   <, <=,>,>=, =, !=

7. The following looping statements are employed.

    For, while and repeat-until

While Loop:

    While < condition >do{
        <statement-1>
                    .   .
        <statement-n>
    }

For Loop:

    For variable: = value-1 to value-2 step step do
    {
      <statement-1>
            .
            .
      <statement-n>

}
 One step is a key word, other Step is used for increment or decrement.

repeat-until:

        repeat{
                <statement-1>
                      .
                      .
                <statement-n>
        }until<condition>

8. A conditional statement has the following forms.
   (1) If <condition> then <statement>

   (2) If <condition> then <statement-1>

   Else <statement-2>

   Case statement:

   Case
   {   **:**<condition-1>**:**<statement-1>
                   .
                   .
           :<condition-n>**:**<statement-n>
           :else:<statement-n+1>
   }
9. Input and output are done using the instructions read & write.
10. There is only one type of procedure:
    Algorithm, the heading takes the form,

    Algorithm Name (<Parameter list>)

    As an example, the following algorithm fields & returns the maximum of 'n' given numbers:

    Algorithm Max(A,n)
     // A is an array of size n
    {
       Result := A[1];
       for I:= 2 to n do
        if A[I] > Result then
          Result :=A[I];
        return Result;
       }
    In this algorithm (named Max), A & n are procedure parameters. Result & I are
    Local variables.

Performance Analysis.

There are many Criteria to judge an algorithm.
- Is it correct?
- Is it readable?
- How it works

Performance evaluation can be divided into two major phases.

1. Performance Analysis (machine independent)

- space complexity: The space complexity of an algorithm is the amount of memory it needs to run for completion.

- time complexity: The time complexity of an algorithm is the amount of computer time it needs to run to completion.

2 .Performance Measurement (machine dependent).

Space Complexity:

The Space Complexity of any algorithm P is given by $S(P)=C+S_P(I)$, C is constant.

1.Fixed Space Requirements (C)
Independent of the characteristics of the inputs and outputs
- It includes instruction space
- space for simple variables, fixed-size structured variable, constants
2. Variable Space Requirements ($S_P(I)$)
depend on the instance characteristic I
- number, size, values of inputs and outputs associated with I
- recursive stack space, formal parameters, local variables, return address

Examples:
*Program 1 :Simple arithmetic function
Algorithmabc(a, b, c)
{
   return a + b + b * c + (a + b - c) / (a + b) + 4.00;
}

$S_P(I)=0$

Hence**S(P)=Constant**

Program 2: Iterative function for sum a list of numbers
Algorithm sum(list[ ], n)
{
tempsum = 0;
  for i = 0 ton do
tempsum += list [i];
  return tempsum;

}

In the above example list[] is dependent on n. Hence $S_P(I)=n$. The remaining variables are i,n, tempsum each requires one location.

Hence **S(P)=3+n**

***Program 3:** Recursive function for sum a list of numbers
Algorithmrsum(list[ ], n)
{

If (n<=0) then

return 0.0

else
 return rsum(list, n-1) + list[n];

 }

In the above example the recursion stack space includes space for formal parameters local variables and return address. Each call to rsum requires 3 locations i.e for list[],n and return address .As the length of recursion is n+1.

S(P)>=3(n+1)

Time complexity:

$$T(P)=C+T_P(I)$$

It is combination of-Compile time (C)
independent of instance characteristics
-run (execution) time T_P
dependent of instance characteristics
Time complexity is calculated in terms of *program step* as it is difficult to know the complexities of individual operations.
Definition: A*program step* is a syntactically or semantically meaningful program segment whose execution time is independent of the instance characteristics.

Program steps are considered for different statements as : for comment zero steps . assignment statement is considered as one step. Iterative statements such as "for, while and until-repeat" statements, we consider the step counts based on the expression .

Methods to compute the step count:
1) Introduce variable count into programs
2) Tabular method
   – Determine the total number of steps contributed by each statement
     step per execution × frequency
   – add up the contribution of all statements

Program 1.with count statements

```
Algorithm sum( list[ ], n)
{
tempsum := 0; count++; /* for assignment */
   for i := 1 to n do {
count++;          /*for the for loop */
tempsum := tempsum + list[i]; count++; /* for assignment */
   }
count++;       /* last execution of for */
   return tempsum;
count++;       /* for return */
```

Hence **T(n)=2n+3**

Program :Recursive sum

```
Algorithmrsum( list[ ], n)
{
      count++;      /*for if conditional */
      if (n<=0) {
                count++; /* for return */
      return 0.0 }

   else

   returnrsum(list, n-1) + list[n];

      count++;/*for return and rsum invocation*/

}
```

T(n)=2n+2

Program for matrix addition

```
Algorithm add( a[ ][MAX_SIZE], b[ ][MAX_SIZE],
               c[ ][MAX_SIZE], rows, cols )
{
   for i := 1 to rows do {
count++; /* for i for loop */
      for j := 1 to cols do {
count++; /* for j for loop */
        c[i][j] := a[i][j] + b[i][j];
count++; /* for assignment statement */
      }
count++;   /* last time of j for loop */
   }
```

count++;   /* last time of i for loop */
}

T(n)=2rows*cols+2*rows+1

II Tabular method.

Complexity is determined by using a table which includes steps per execution(s/e) i.e amount by which count changes as a result of execution of the statement.

Frequency – number of times a statement is executed.

Statement	s/e	Frequency	Total steps
Algorithm sum(list[ ], n)	0	-	0
{	0	-	0
tempsum := 0;	1	1	1
for i := 0 ton do	1	n+1	n+1
tempsum := tempsum + list [i];	1	n	n
return tempsum;	1	1	1
}	0	0	0
Total			2n+3

Statement	s/e	Frequency n=0 n>0		Total steps n=0 n>0	
Algorithmrsum(list[ ], n)	0	-	-	0	0
{	0	-	-	0	0
If (n<=0) then	1	1	1	1	1
return 0.0;	1	1	0	1	0
else	0	0	0	0	0
return rsum(list, n-1) + list[n];	1+x	0	1	0	1+x
}	0	0	0	0	0
Total				2	2+x

Statement	s/e	Frequency	Total steps
Algorithm add(a,b,c,m,n)	0	-	0
{	0	-	0
for i:=1 to m do	1	m+1	m+1
for j:=1 to n do	1	m(n+1)	mn+m
c[i,j]:=a[i,j]+b[i,j];	1	mn	mn
}	0	-	0
Total			2mn+2m+1

Complexity ofAlgorithms

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 thealgorithm.

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 casesare:

1. Best Case   : The minimum possible value of f(n) is called the bestcase.

2. Average Case   : The average value off(n).

3. Worst Case   : The maximum value of f(n) for any key possibleinput.

The field of computer science, which studies efficiency of algorithms, is known as analysis ofalgorithms.

Algorithms can be evaluated by a variety of criteria. Most often we shall be interested in the rate of growth of the time or space required to solve larger and larger instances of a problem. We will associate with the problem an integer, called the size of the problem, which is a measure of the quantity of inputdata.Rate ofGrowth:

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

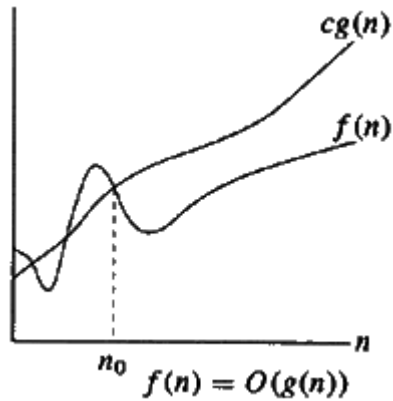## Asymptotic notation

Big oh notation:O

The function f(n)=O(g(n)) (read as "f of n is big oh of g of n") iff there exist positive constants c and n_0 such that **f(n)≤C*g(n)** for all n, n≥0

The value g(n)is the upper bound value of f(n).

Example:

3n+2=O(n) as

3n+2 ≤4n for all n≥2

$cg(n)$

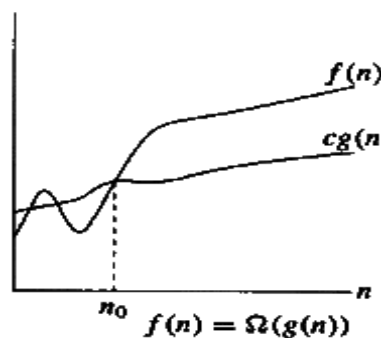$f(n)$

n_0 $f(n) = O(g(n))$

n

Omega notation:Ω
The function f(n)=Ω (g(n)) (read as "f of n is Omega of g of n") iff there exist positive
constants c and n_0 such that **f(n)$\geq$C*g(n)** for all n, n$\geq$0
The value g(n) is the lower bound value of f(n).
Example:
3n+2=Ω (n) as
3n+2 $\geq$3n for all n$\geq$1

$f(n)$

$cg(n$

n_0 $f(n) = \Omega(g(n))$

n

Theta notation:θ
The function f(n)= θ (g(n)) (read as "f of n is theta of g of n") iff there exist positive
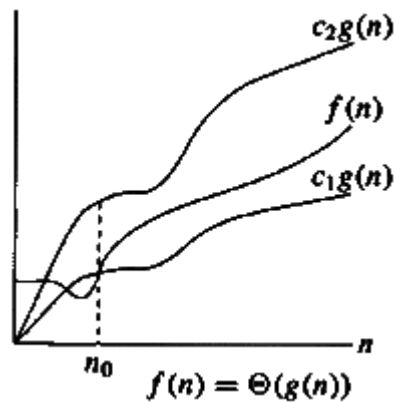constants c1, c2 and n_0 such that **C1*g(n) $\leq$f(n)$\leq$C2*g(n)** for all n, n$\geq$0
Example:
3n+2=θ (n) as
3n+2 $\geq$3n for all n$\geq$2
3n+2 $\leq$3n for all n$\geq$2
Here c1=3 and c2=4 and n_0=2

$$f(n) = \Theta(g(n))$$

Little oh: o
The function f(n)=o(g(n)) (read as "f of n is little oh of g of n") iff
Lim f(n)/g(n)=0   for all n, n≥0
n→~
Example:
$3n+2=o(n^2)$ as

Lim $((3n+2)/n^2)$=0
n→~

Little Omega:ω
The function f(n)=ω (g(n)) (read as "f of n is little ohomega of g of n") iff

Lim g(n)/f(n)=0   for all n, n≥0
n→~
Example:
$3n+2=o(n^2)$ as

Lim $(n^2/(3n+2)$ =0
n→~

AnalyzingAlgorithms
Suppose 'M' is an algorithm, and suppose 'n' is the size of the input data. Clearly the complexity f(n) of M increases as n increases. It is usually the rate of increase of f(n) we want to examine. This is usually done by comparing f(n) with some standard functions. The most common computing timesare:

$O(1), O(\log_2 n), O(n), O(n. \log_2 n), O(n^2), O(n^3), O(2^n),$ n! and n^n

Numerical Comparison of DifferentAlgorithms
The execution time for six of the typical functions is givenbelow:

| N	log2n	n*log2n	n^2	n^3	2^n

1	0	0	1	1	2
2	1	2	4	8	4
4	2	8	16	64	16
8	3	24	64	512	256
16	4	64	256	4096	65,536
32	5	160	1024	32,768	4,294,967,296
64	6	384	4096	2,62,144	Note1
128	7	896	16,384	2,097,152	Note2
256	8	2048	65,536	1,677,216	????????

Note1: The value here is approximately the number of machine instructions executed by a 1 gigaflop computer in 5000years.

Note 2: The value here is about 500 billion times the age of the universe in nanoseconds, assuming a universe age of 20 billionyears.

Graph of log n, n, n log n, n^2, n^3, 2^n, n! andn^n

Graph of $\log_2 n$, n, $n \log_2 n$, n^2, n^3, 2^n, $n!$, and n^n
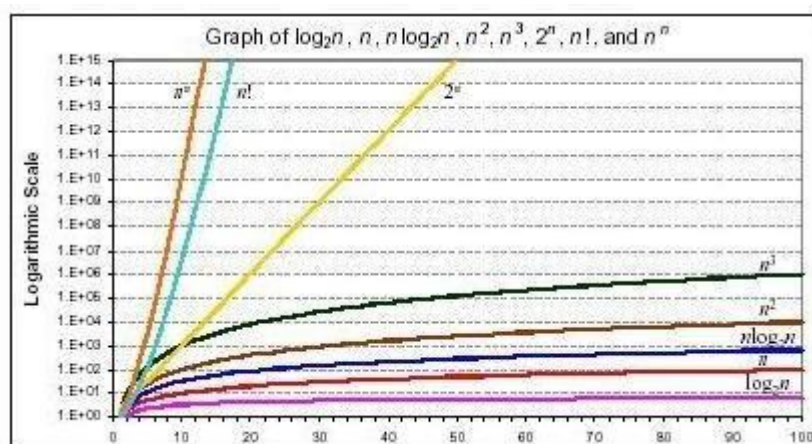
One way to compare the function f(n) with these standard function is to use the functional 'O' notation, suppose f(n) and g(n) are functions defined on the positive integers with the property that f(n) is bounded by some multiple g(n) for almost all 'n'.Then,f(n) =O(g(n)) Which is read as "f(n) is of order g(n)". For example, the order of complexityfor:

- Linear search is O(n)

- Binary search is O (logn)

- Bubble sort is O(n^2)

- Merge sort is O (n logn)

Probabilistic analysis of algorithms is an approach to estimate the computational complexity of an algorithm or a computational problem. It starts from an assumption about a probabilistic distribution of the set of all possible inputs. This assumption is then used to design an efficient algorithm or to derive the complexity of a known algorithm.

DIVIDE AND CONQUER

General method:

Given a function to compute on 'n' inputs the divide-and-conquer strategy suggests splitting the inputs into 'k' distinct subsets, 1<k<=n, yielding 'k' sub problems.

These sub problems must be solved, and then a method must be found to combine sub solutions into a solution of the whole.

If the sub problems are still relatively large, then the divide-and-conquer strategy can possibly be reapplied.Often the sub problems resulting from a divide-and-conquer design are of the same type as the original problem.For those cases the re application of the divide-and-conquer principle is naturally expressed by a recursive algorithm.DAndC(Algorithm) is initially invoked as DandC(P), where 'p' is the problem to be solved.Small(P) is a Boolean-valued function that determines whether the i/p size is small enough that the answer can be computed without splitting.If this so, the function 'S' is invoked.Otherwise, the problem P is divided into smaller sub problems.These sub problems P1, P2 …P_k are solved by recursive application of DAndC.Combine is a function that determines the solution to p using the solutions to the 'k' sub problems.If the size of 'p' is n and the sizes of the 'k' sub problems are n1, n2 ….nk, respectively, then the computing time of DAndC is described by the recurrence relation.

$$T(n)= \{ \; g(n) \qquad\qquad\qquad\qquad n \; small$$

$$T(n1)+T(n2)+……………+T(n_k)+f(n); \; otherwise.$$

Where T(n) is the time for DAndC on any i/p of size 'n'.

g(n) is the time of compute the answer directly for small i/ps.

f(n) is the time for dividing P & combining the solution to

sub problems.

```
Algorithm DAndC(P)
  {
  if small(P) then return S(P);
  else
  {
  divide P into smaller instances
        P1, P2… Pk, k>=1;

   Apply DAndC to each of these sub problems;
  return combine (DAndC(P1), DAndC(P2),…….,DAndC(Pk));
   }
  }
```

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

$$T(n) \; = T(1) \qquad\qquad n=1$$

$$= aT(n/b)+f(n) \quad n>1$$

Where a & b are known constants.

We assume that T(1) is known & 'n' is a power of b(i.e., $n=b^k$)

One of the methods for solving any such recurrence relation is called the substitution method.This method repeatedly makes substitution for each occurrence of the function. T is the right-hand side until all such occurrences disappear.
Example:

1) Consider the case in which a=2 and b=2. Let T(1)=2 & f(n)=n.
   We have,

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

$$= 2[2T(n/2/2)+n/2]+n$$

$$= [4T(n/4)+n]+n$$

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

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

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

$$= 8T(n/8)+n+2n$$

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

$$*$$
$$*$$

- In general, we see that $T(n)=2^iT(n/2^i)+in$., for any $\log_2 n >=i>=1$.
$T(n) =2^{\log n} T(n/2^{\log n}) + n \log n$

Corresponding to the choice of $i=\log_2 n$

Thus, $T(n) = 2^{\log n} T(n/2^{\log n}) + n \log n$

$$= n. T(n/n) + n \log n$$

$$= n. T(1) + n \log n \qquad [\text{since, } \log 1=0, 2^0=1]$$

$$= 2n + n \log n$$

$$T(n)= n\log n+2n.$$

The recurrence using the substitution method,it can be shown as

$$\mathbf{T(n)=n^{\log_b a}[T(1)+u(n)]}$$

h(n)	u(n)
$O(n^r), r<0$	$O(1)$

$((\log n)^i), i \geq 0$	$((\log n)^{i+1}/(i+1))$
$\Omega(n^r), r > 0$	$(h(n))$

Applications of Divide and conquer rule or algorithm:
   Binary search, Quick sort, Merge sort, Strassen's matrix multiplication.

BINARY SEARCH
 Given a list of n elements arranged in increasing order. The problem is to determine whether a given element is present in the list or not. If x is present then determine the position of x, otherwise position is zero.

Divide and conquer is used to solve the problem. The value Small(p) is true if n=1. S(P)= i, if x=a[i], a[] is an array otherwise S(P)=0.If P has more than one element then it can be divided into sub-problems. Choose an index j and compare x with a_j then there 3 possibilities (i). X=a[j] (ii) x<a[j] (x is searched in the list a[1]…a[j-1])
(iii) x>a[j] (x is searched in the list a[j+1]…a[n]).
And the same procedure is applied repeatedly until the solution is found or solution is zero.

```
    Algorithm Binsearch(a,n,x)
        // Given an array a[1:n] of elements in non-decreasing
        //order, n>=0,determine whether 'x' is present and
        // if so, return 'j' such that x=a[j]; else return 0.
        {
        low:=1; high:=n;
        while (low<=high) do
        {
        mid:=[(low+high)/2];
        if (x<a[mid]) then high;
        else if(x>a[mid]) then
    low:=mid+1;
        else return mid;
          }
        return 0;
        }
```

Algorithm, describes this binary search method, where Binsrch has 4 inputssa[], I , n& x.It is initially invoked as Binsrch (a,1,n,x)A non-recursive version of Binsrch is given below.
This Binsearch has 3 i/psa,n, & x.The while loop continues processing as long as there are more elements left to check.At the conclusion of the procedure 0 is returned if x is not present, or 'j' is returned, such that a[j]=x.We observe that low & high are integer Variables such that each time through the loop either x is found or low is increased by at least one or high is decreased at least one.
Thus we have 2 sequences of integers approaching each other and eventually low becomes > than high & causes termination in a finite no. of steps if 'x' is not present.
Example:

> 1) Let us select the 14 entries.
>   -15,-6,0,7,9,23,54,82,101,112,125,131,142,151.

Place them in a[1:14], and simulate the steps Binsearch goes through as it searches for different values of 'x'.

Only the variables, low, high & mid need to be traced as we simulate the algorithm.

We try the following values for x: 151, -14 and 9.

for 2 successful searches & 1 unsuccessful search.

Table. Shows the traces of Binsearch on these 3 steps.

X=151	low	high	mid
	1	147	
	8	14	11
	12	14	13
	14	14	14
			Found

x=-14	low	high	mid
	1	14	7
	1	6	3
	1	2	1
	2	2	2
	2	1	Not found

x=9	low	high	mid
	1	14	7
	1	6	3
	4	6	5
			Found

Theorem: Algorithm Binsearch(a,n,x) works correctly.

Proof: We assume that all statements work as expected and that comparisons such as x>a[mid] are appropriately carried out.

Initially low =1, high= n,n>=0, and a[1]<=a[2]<=……..<=a[n].
If n=0, the while loop is not entered and is returned. Otherwise we observe that each time thro' the loop the possible elements to be checked of or equality with x and a[low], a[low+1],……..,a[mid],……a[high]. If x=a[mid], then the algorithm terminates successfully. Otherwise, the range is narrowed by either increasing low to (mid+1) or decreasing high to (mid-1). Clearly, this narrowing of the range does not affect the outcome of the search. If low becomes > than high, then 'x' is not present & hence the loop is exited.

The complexity of binary search is **successful searches** is
Worst case is O(log n) or θ(log n)
Average case is O(log n) or θ(log n)
Best case is O(1) or θ(1)

Unsuccessful searches is: θ(log n) for all cases.

MergeSort

Merge sort algorithm is a classic example of divide and conquer. To sort an array, recursively, sort its left and right halves separately and then merge them. The time complexity of merge sort in the *best case, worst case* and *average case* is O(n log n) and the number of comparisons used is nearlyoptimal.

This strategy is so simple, and so efficient but the problem here is that there seems to be no easy way to merge two adjacent sorted arrays together in place (The result must be build up in a separatearray).The fundamental operation in this algorithm is merging two sorted lists. Because the lists are sorted, this can be done in one pass through the input, if the output is put in a thirdlist.

Algorithm MERGESORT (low,high)

```
// a (low : high) is a global array to besorted.
{
        if (low <high)
        {
mid := (low +high)/2;//finds where to split theset
MERGESORT(low, mid); //sortonesubset
MERGESORT(mid+1, high); //sort the other subset
                MERGE(low,mid,high); // combine theresults
        }

}
```

Algorithm MERGE (low, mid,high)

```
// a (low : high) is a global array containing two sortedsubsets
// in a (low : mid) and in a (mid + 1 :high).
// The objective is to merge these sorted sets into singlesorted
// set residing in a (low : high). An auxiliary array B isused.
{
        h :=low; i := low; j:= mid + 1;
         while ((h ≤mid) and (J ≤high))do
        {
                if (a[h] ≤a[j])then
                {
b[i] :=a[h];   h:=h+1;

}
else
{

                        b[i] :=a[j]; j := j +1;


                }
                i := i +1;
        }
      if (h > mid)then
                for k := j to highdo
                {
                        b[i]        :=        a[k];        i        :=        i        +1
for k := h to middo
```

```
                {
                        b[i] := a[K]; i := i +l;
                }
        for k := low to highdo
                a[k] :=b[k];
 }
```
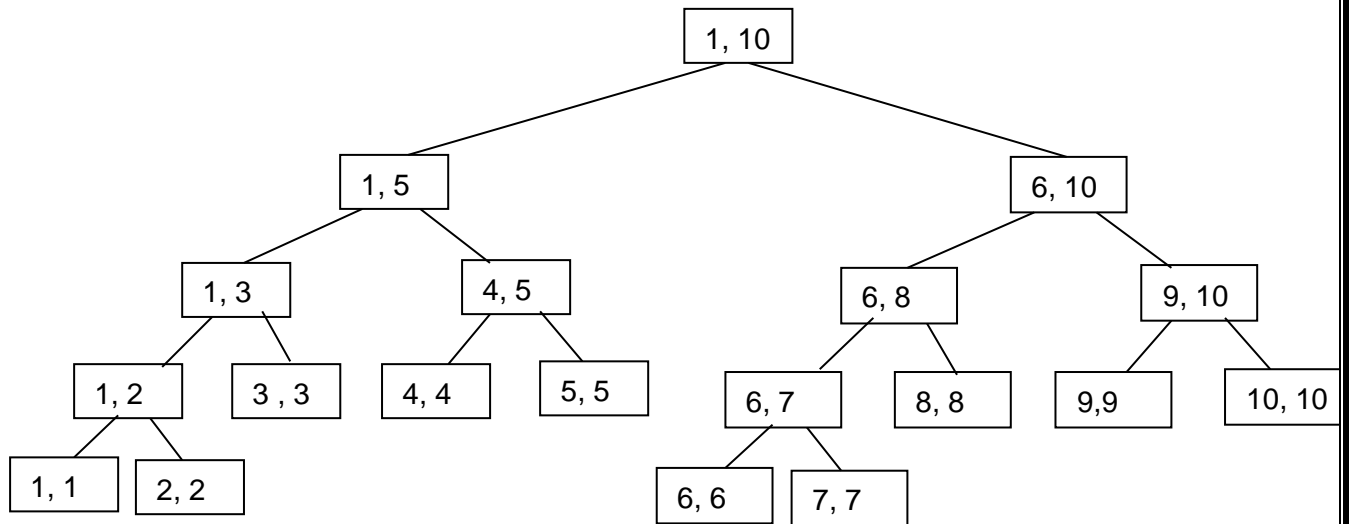
Example

Tree call of Merge sort:

A[1:10]={310,285,179,652,351,423,861,254,450,520}

Tree call of Merge sort (1, 10)

Analysis of MergeSort

      We will assume that 'n' is a power of 2, so that we always split into even halves, so we solve for the case $n =2^k$.

      For n = 1, the time to merge sort is constant, which we will be denote by 1. Otherwise, the time to merge sort 'n' numbers is equal to the time to do two recursive merge sorts of size n/2, plus the time to merge, which is linear. The equation says thisexactly:

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

      This is a standard recurrence relation, which can be solved several ways. We will solve by substituting recurrence relation continually on the right–handside.

We have, $T(n) = 2T(n/2) +n$
Since we can substitute n/2 into this mainequation

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

Wehave,

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

Again, by substituting n/4 into the main equation, we seethat

$$4T(n/4) \quad = \quad 4\,(2T(n/8)) + n/4$$
$$= \quad 8\,T(n/8) + n$$

So wehave,

$$T(n/4) \quad = \quad 2\,T(n/8) + n$$
$$T(n) \quad = \quad 8\,T(n/8) + 3n$$

Continuing in this manner, weobtain:

$$T(n) \quad = \quad 2^k\,T(n/2^k) + K.n$$

As $n = 2^k$, $K = \log_2 n$, substituting this in the aboveequation

$T(n) = 2^{\log n} T(n/2^{\log n}) + \log n * n$

$= nT(1) + n \log n$

$\quad = n + n \log n$

Representing in O-notation $T(n) = O(n \log n)$.

We have assumed that $n = 2^k$. The analysis can be refined to handle cases when 'n' is not a power of 2. The answer turns out to be almostidentical.

Although merge sort's running time is O(n log n), it is hardly ever used for main memory sorts. The main problem is that merging two sorted lists requires linear extra memory and the additional work spent copying to the temporary array and back, throughout the algorithm, has the effect of slowing down the sort considerably. *The Best and worst case time complexity of Merge sort is O(n logn).*

Strassen's MatrixMultiplication:

The matrix multiplication of algorithm due to Strassens is the most dramatic example of divide and conquer technique(1969).

Let A and B be two n×n Matrices. The product matrix C=AB is also a n×n matrix whose i, j^{th} element is formed by taking elements in the ith row of A and j^{th} column of B and multiplying them to get

The usual wayC(i, j)=

Here $1 \le i$ & $j \le n$ means i and j are in between 1 and n.

To compute C(i, j) using this formula, we need n multiplications.

The divide and conquer strategy suggests another way to compute the product of two n×n matrices. For Simplicity assume n is a power of 2 that is $n=2^k$, k is a nonnegative integer. If n is not power of two then enough rows and columns of zeros can be added to both A and B, so that resulting dimensions are a power of two.

To multiply two n x n matrices A and B, yielding result matrix 'C' as follows:
Let A and B be two n×n Matrices. Imagine that A & B are each partitioned into four square sub matrices. Each sub matrix having dimensions n/2×n/2.
The product of AB can be computed by using previous formula.
If AB is product of 2×2 matrices then

$$=$$

Then c_{ij} can be found by the usual matrix multiplication algorithm,

$C_{11} = A_{11} .B_{11} + A_{12} .B_{21}$

$C_{12} = A_{11} .B_{12} + A_{12} .B_{22}$

$C_{21} = A_{21} .B_{11} + A_{22} .B_{21}$

$C_{22} = A_{21} .B_{12} + A_{22} .B_{22}$

This leads to a divide–and–conquer algorithm, which performs nxn matrix multiplication by partitioning the matrices into quarters and performing eight (n/2)x(n/2) matrix multiplications and four (n/2)x(n/2) matrix additions.

$T(1) = 1$
$T(n) = 8T(n/2)$

Which leads to $T(n) = O(n^3)$, where n is the power of 2.
Strassens insight was to find an alternative method for calculating the C_{ij}, requiring seven (n/2) x (n/2) matrix multiplications and eighteen (n/2) x (n/2) matrix additions and subtractions:

$P = (A_{11} + A_{22}) (B_{11} + B_{22})$

$Q = (A_{21} + A_{22})B_{11}$

$R = A_{11} (B_{12} - B_{22})$

$S = A_{22} (B_{21} - B_{11})$

$T = (A_{11} + A_{12})B_{22}$

$U = (A_{21} - A_{11}) (B_{11} + B_{12})$

$V = (A_{12} - A_{22}) (B_{21} + B_{22})$

$C_{11} = P + S - T + V$

$C_{12} = R + T$

$C_{21} = Q + S$

$C_{22} = P + R - Q + U.$

This method is used recursively to perform the seven (n/2) x (n/2) matrix multiplications, then the recurrence equation for the number of scalar multiplications performedis:

$$T(1) \quad = \quad 1$$
$$T(n) \quad = \quad 7T(n/2)$$

Solving this for the case of $n = 2^k$ iseasy:

$$T(2^k) \quad = \quad 7T(2^{k-1})$$

$$= \quad 7^2 T(2^{k-}$$
$$= \quad \text{- - - - --}$$
$$= \quad \text{- - - - --}$$
$$= \quad 7^i T(2^{k-i})$$

Put i =k

$$= 7^k T(2^0)$$

As k is the power of 2

That is, $T(n) \quad = \quad 7^{\log_2 n}$

$$= \quad n^{\log_2 7}$$

$$= O(n^{\log_2 7}) = O(n^{2.81})$$

So, concluding that Strassen's algorithm is asymptotically more efficient than the standard algorithm. In practice, the overhead of managing the many small matrices does not pay off until 'n' revolves thehundreds.

QuickSort

The main reason for the slowness of Algorithms in which all comparisons and exchanges between keys in a sequence $w_1, w_2, \ldots, w_n$ take place between adjacent pairs. In this way it takes a relatively long time for a key that is badly out of place to work its way into its proper position in the sortedsequence.

Hoare his devised a very efficient way of implementing this idea in the early 1960's that improves the $O(n^2)$ behavior of the algorithm with an expected performance that is O(n logn).In essence, the quick sort algorithm partitions the original array by rearranging it into two groups. The first group contains those elements less than some arbitrary chosen value taken from the set, and the second group contains those elements greater than or equal to the chosenvalue.

The chosen value is known as the *pivot element*. Once the array has been rearranged in this way with respect to the pivot, the very same partitioning is recursively applied to each of the two subsets. When all the subsets have been partitioned and rearranged, the original array issorted.

The function partition() makes use of two pointers 'i' and 'j' which are moved toward

each other in the followingfashion:

Repeatedly increase the pointer 'i' until a[i] >=pivot.

Repeatedly decrease the pointer 'j' until a[j] <=pivot.

If j > i, interchange a[j] witha[i]

Repeat the steps 1, 2 and 3 till the 'i' pointer crosses the 'j' pointer. If 'i' pointer crosses 'j' pointer, the position for pivot is found and place pivot element in 'j' pointerposition.

The program uses a recursive function quicksort(). The algorithm of quick sort function sorts all elements in an array 'a' between positions 'low' and'high'.

It terminates when the condition low >= high is satisfied. This condition will be satisfied only when the array is completelysorted.Here we choose the first element as the 'pivot'. So, pivot = x[low]. Now it calls the partition function to find the proper position j of the element x[low] i.e. pivot. Then we will have two sub-arrays x[low], x[low+1], ... x[j-1] and x[j+1], x[j+2], . ..x[high].It calls itself recursively to sort the left sub-array x[low], x[low+1], ... x[j-1] between positions low and j-1 (where j is returned by the partitionfunction).It calls itself recursively to sort the right sub-array x[j+1], x[j+2], ... x[high] between positions j+1 andhigh.

Algorithm

AlgorithmQUICKSORT(low,high)

// sorts the elements a(low), ... , a(high) which reside in the global array A(1 :n) into
//ascending order a (n + 1) is considered to be defined and must be greater than all
//elements in a(1 : n); A(n + 1) = α*/
{

    If(low < high) then
    {

        j := PARTITION(a, low,high+1);

                // J is the position of the partitioningelement
        QUICKSORT(low, j –1);
        QUICKSORT(j + 1 ,high);
    }

}

Algorithm PARTITION(a, m,p)

{

    V :=a(m); i :=m; j:=p;
    // a (m) is thepartitionelement
    do
    {

        repeat
        i := i +1;
      until (a(i)≥v);
        repeat
        j := j –1;
        until (a(j)≤v);
        if (i < j) then INTERCHANGE(a, i,j)
    } while (i ≥j);
    a[m] :=a[j];a[j]:=V;
    returnj;

}

Algorithm INTERCHANGE(a, i,j)

```
{
     p:= a[i];
     a[i]:=a[j];
     a[j]:=p;
}
```

Example

Select first element as the pivot element. Move 'i' pointer from left to right in search of an element larger than pivot. Move the 'j' pointer from right to left in search of an element smaller than pivot. If such elements are found, the elements are swapped. This process continues till the 'i' pointer crosses the 'j' pointer. If 'i' pointer crosses 'j' pointer, the position for pivot is found and interchange pivot and element at 'j' position.

Let us consider the following example with 13 elements to analyze quicksort:

_1_	_2_	_3_	_4_	_5_	_6_	_7_	_8_	_9_	_10_	_11_	_12_	_13_	_Remarks_
38	08	16	06	79	57	24	56	02	58	04	70	45	
pivot				I						j			swap i &j
				04						79			
					i			j					swap i &j
					02			57					
						j	i						
(24	08	16	06	04	02)	**38**	(56	57	58	79	70	45)	swap pivot
pivot					j,i								swap pivot
(02	08	16	06	04)	**24**								
pivot ,j	i												swap pivot
02	(08	16	06	04)									
	pivot	i		j									swap i &j
		04		16									
		j	i										
	(06	04)	**08**	(16)									swap pivot
	pivot ,J	i											
	(04)	**06**											swap pivot
	04 pivot , J,i												
				16 pivot , J,i									
(02	**04**	**06**	**08**	**16**	**24**)	38							
							(56	57	58	79	70	45)	

							pivot	i				j	swap i
								45				57	
								j	i				
							(45)	**56**	(58	79	70	57)	swap pivot
							45 pivot, j,1						swap pivot
									(58 pivo	79 i	70	57) j	swap i
										57		79	
										j	i		
									(57)	**58**	(70	79)	swap pivot
									57 pivot, j,1				
											(70	79)	
											pivot, j	i	swap pivot
											70		
												79 pivot, j,1	
							(45	56	57	58	70	79)	
02	**04**	**06**	**08**	**16**	**24**	**38**	**45**	**56**	**57**	**58**	**70**	**79**	

Analysis of QuickSort:

Like merge sort, quick sort is recursive, and hence its analysis requires solving a recurrence formula. We will do the analysis for a quick sort, assuming a random pivot
We will take T (0) = T (1) = 1, as in merge sort.
The running time of quick sort is equal to the running time of the two recursive calls plus the linear time spent in the partition (The pivot selection takes only constant time). This gives the basic quick sortrelation:

$$T (n) = T (i) + T (n - i - 1) + Cn \qquad - \qquad (1)$$

Where, $i = |S1|$ is the number of elements in $S1$.

Worst CaseAnalysis

The pivot is the smallest element, all the time. Then i=0 and if we ignore T(0)=1, which is insignificant, the recurrenceis:

$$T (n) = T (n - 1) + Cn \qquad n>1 \qquad - \qquad (2)$$

Using equation – (1) repeatedly,thus

$$T (n - 1) = T (n - 2) + C (n - 1)$$

$$T(n-2) = T(n-3) + C(n-2)$$
$$- - - - - - - -$$
$$T(2) = T(1) + C(2)$$

Adding up all these equationsyields

$$= O(n^2) \qquad\qquad - \qquad\qquad (3)$$

Best CaseAnalysis

In the best case, the pivot is in the middle. To simply the math, we assume that the two sub-files are each exactly half the size of the original and although this gives a slight over estimate, this is acceptable because we are only interested in a Big – oh answer.

$$T(n) = 2T(n/2) + Cn \qquad\qquad - \qquad\qquad (4)$$

Divide both sides byn and Substitute n/2 for 'n'

Finally,

Which yields, $T(n) = C\, n \log n + n = $ **O(n logn)** -

This is exactly the same analysis as merge sort, hence we get the sameanswer.

Average CaseAnalysis

The number of comparisons for first call on partition: Assume left_to_right moves over k smaller element and thus k comparisons. So when right_to_left crosses left_to_right it has made n-k+1 comparisons. So, first call on partition makes n+1 comparisons. The average case complexity of quicksort is

T(n) = comparisons for first call onquicksort

+

$\{\Sigma\ 1<=nleft,nright<=n\ [T(nleft) + T(nright)]\}n = (n+1) + 2\ [T(0) +T(1) + T(2) + $
----- $+T(n-1)]/n$

$nT(n) = n(n+1) + 2\ [T(0) +T(1) + T(2) + ----- + T(n-2) +T(n-1)]$

$(n-1)T(n-1) = (n-1)n + 2\ [T(0) +T(1) + T(2) + ----- + T(n-2)]\backslash$

Subtracting bothsides:

$nT(n) –(n-1)T(n-1) = [\ n(n+1) – (n-1)n] + 2T(n-1) = 2n + 2T(n-1)\ nT(n)$

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

$T(n) = 2 +(n+1)T(n-1)/n$

The recurrence relation obtained is:

$T(n)/(n+1) = 2/(n+1) +T(n-1)/n$

Using the method ofsubstitution:

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

T(2)/3 = 2/3 + T(1)/2 T(1)/2 = 2/2 +T(0)

Adding bothsides:

T(n)/(n+1) + [T(n-1)/n + T(n-2)/(n-1) + ------------- + T(2)/3 +T(1)/2]

= [T(n-1)/n + T(n-2)/(n-1) + ------------- + T(2)/3 + T(1)/2] + T(0)+ [2/(n+1)

 + 2/n + 2/(n-1) + ---------- +2/4 +2/3]

Cancelling the commonterms:

T(n)/(n+1) = 2[1/2 +1/3+1/4+--------------+1/n+1/(n+1)]

Finally,

We will get,

O(n log n)

UNIT-II

DEARCHING AND TRAVERSAL TECHNIQUES

Disjoint Set Operations

Set:

A set is a collection of distinct elements. The Set can be represented,for examples, asS1={1,2,5,10}.

Disjoint Sets:

The disjoints sets are those do not have any common element.
For example S1={1,7,8,9} and S2={2,5,10}, then we can say that S1 and S2are two disjoint sets.

Disjoint Set Operations:

The disjoint set operations are
1. Union
2. Find

Disjoint setUnion:

If Si and Sj are two disjoint sets, then their union Si U Sj consists of all the elements x such that x is in Si or Sj.

Example:

S1={1,7,8,9}   S2={2,5,10}
S1 US2={1,2,5,7,8,9,10}

Find: Given the element I, find the set containing I.

Example:

S1={1,7,8,9}   S2={2,5,10}   s3={3,4,6}

Then,

Find(4)=S3   Find(5)=S2   Find97)=S1

Set Representation:

The set will be represented as the tree structure where all children will store the address of parent / root node. The root node will store null at the place of parent address. In the given set of elements any element can be selected as the root node, generally we select the first node as the root node.

Example:

S1={1,7,8,9}   S2={2,5,10}   s3={3,4,6}

Then these sets can be represented as
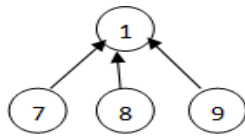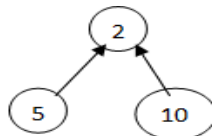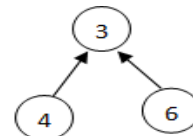
Disjoint Union:

To perform disjoint set union between two sets Si and Sj can take any one root and make it sub-tree of the other. Consider the above example sets S1 and S2 then the union of S1 and S2 can be represented as any one of the following.
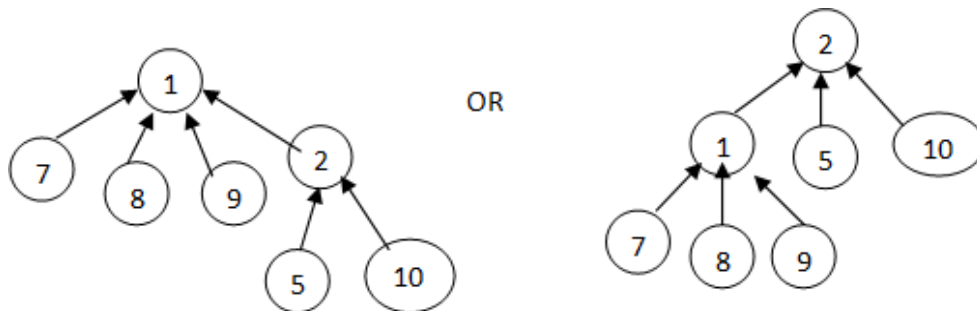
S1 U S2

Find:

   To perform find operation, along with the tree structure we need to maintain the name of each set. So, we require one more data structure to store the set names. The data structure contains two fields. One is the set name and the other one is the pointer to root.
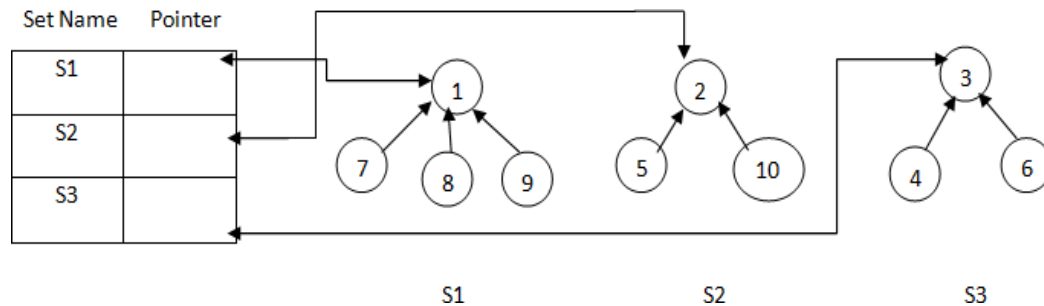
Union and Find Algorithms:

   In presenting Union and Find algorithms, we ignore the set names and identify sets just by the roots of trees representing them. To represent the sets, we use an array of 1 to n elements where n is the maximum value among the elements of all sets. The index values represent the nodes (elements of set) and the entries represent the parent node. For the root value the entry will be'-1'.

Example:

For the following sets the array representation is as shownbelow.
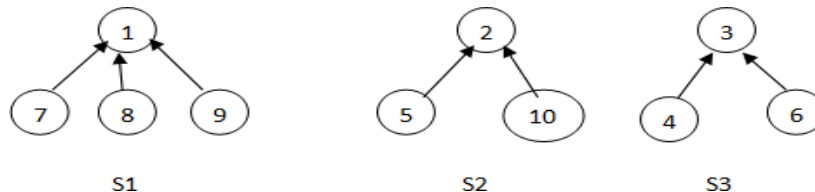
I	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
P	-1	-1	-1	3	2	3	1	1	1	2

Algorithm for Union operation:

   To perform union the **SimpleUnion(i,j)** function takes the inputs as the set roots i and j . And make the parent of i as j i.e, make the second root as the parent of first root.

Algorithm SimpleUnion(i,j)
```
{
        P[i]:=j;
        }
```

**Algorithm for find operation:**

The SimpleFind(i) algorithm takes the element i and finds the root node of i.It starts at i until it reaches a node with parent value-1.

Algorithm SimpleFind(i)
{

   while(P[i]≥0)
   i:=P[i];
   returni;

}

Analysis of SimpleUnion(i,j) and SimpleFind(i):

Although the SimpleUnion(i,j) and SimpleFind(i) algorithms are easy to state, their performance characteristics are not very good. For example, consider the sets
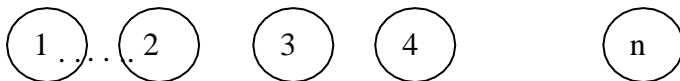
Then if we want to perform following sequence of operations   Union(1,2) ,Union(2,3)……. Union(n-1,n) and sequence of Find(1), Find(2)………Find(n).

The sequence of Union operations results the degenerate tree as below.
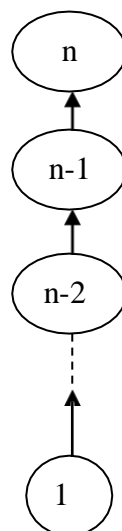
Since, the time taken for a Union is constant, the n-1 sequence of union can be processed in time O(n).And for the sequence of Find operations it will take

We can improve the performance of union and find by avoiding the creation of degenerate tree by applying weighting rule for Union.

Weighting rule forUnion:

If the number of nodes in the tree with root I is less than the number in the tree with the root j, then make 'j' the parent of i; otherwise make 'i' the parent of j.
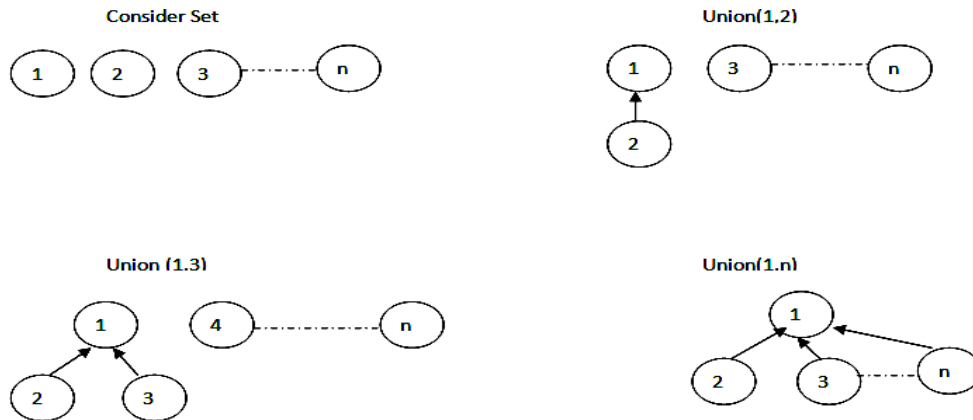
To implement weighting rule we need to know how many nodes are there in every tree. To do this we maintain "count" field in the root of every tree. If 'i' is the root then count[i] equals to number of nodes in tree with rooti.

Since all nodes other than roots have positive numbers in parent (P) field, we can maintain count in P field of the root as negative number.

Algorithm WeightedUnion(i,j)
**//Union sets with roots i and j, i≠j using the weighted rule**

// P[i]=-count[i] andp[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 nodes
            P[j]:=i;
            P[i]:=temp;
        }
```
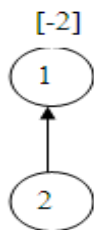
}

Collapsing rule for find:

If j is a node on the path from i to its root and p[i]≠root[i], then set P[j] to root[i].
Consider the tree created by WeightedUnion() on the sequence of1≤i≤8.
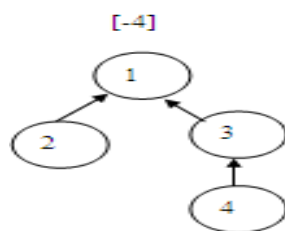Union(1,2), Union(3,4), Union(5,6) and Union(7,8)

[-1]	[-1]	[-1]	[-1]	[-1]	[-1]	[-1]	[-1]
1	2	3	4	5	6	7	8
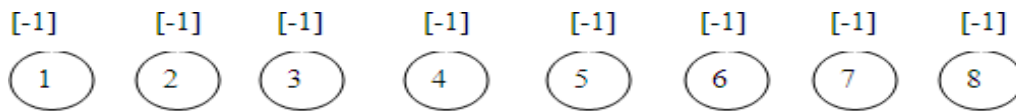
Union(1,2)   Union(3,4)   Union(5,6)   Union(7,8)
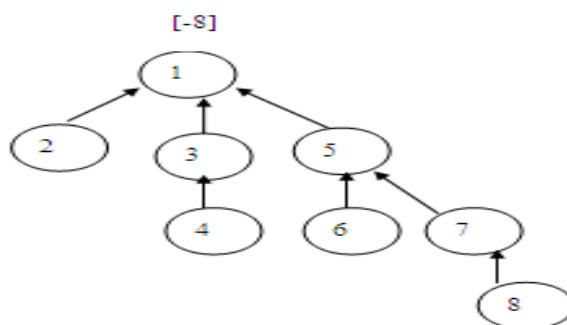
[-2]   [-2]   [-2]   [-2]

Union(1,3)   Union(5,7)

[-4]   [-4]

Union(1,5)

[-8]
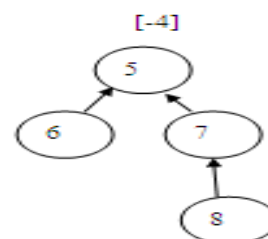
Now process the following eight find operations

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.

When Collapsing find is used the first Find(8) requires going up three links and resetting three links. Each of remaining seven finds require going up only one link field. Then the total cost is now only 13 moves.(3 going up + 3 resets + 7 remaining finds).

Algorithm CoIlapsingFind(i)

// Find the root of the tree containing element i. Use the

// collapsing rule to collapse all nodes from i to the root .
{   r := *i;*
    while *(p[r] >0)* do
     r := *p[r] ; /* Find the root,
        while (i< r) do // Collapse nodes from i to root r,
             r:=*p[i];*
             return r;
}

SEARCHING

Search means finding a path or traversal between a start node and one of a set of goal nodes.
Search is a study of states and their transitions.

Search involves visiting nodes in a graph in a systematic manner, and may or may
not result into a visit to all nodes. When the search necessarily involved the examination
of every vertex in the tree, it is called the traversal.

Techniques for Traversal of a Binary Tree:

A binary tree is a finite (possibly empty) collection of elements. When the binary tree
is not empty, it has a root element and remaining elements (if any) are partitioned into two
binary trees, which are called the left and right subtrees.

There are three common ways to traverse a binary tree: Preorder, Inorder, postorder
In all the three traversal methods, the left sub tree of a node is traversed before the right
sub tree. The difference among the three orders comes from the difference in the time at
which a node is visited.

Inorder Traversal:

In the case of inorder traversal, the root of each subtree is visited after its left subtree
has been traversed but before the traversal of its right subtree begins. The steps for
traversing a binary tree in inorder traversal are:

1. Visit the left subtree, using inorder.
2. Visit the root.
3. Visit the right subtree, using inorder.

The algorithm for preorder traversal is as follows:
  treenode =record

{
        Type data;   //Type is the data type of data.
         Treenode *lchild, *rchild;
}
Algorithm inorder(t)
// t is a binary tree. Each node of t has three fields: lchild, data, and rchild.
{
        If(t $\neq$0)then

        {

                inorder (t$\rightarrow$ lchild);
                visit(t);
                inorder (t $\rightarrow$rchild);

```
            }

    }

    **Preorder Traversal:**
```

In a preorder traversal, each node is visited before its left and right subtrees are traversed. Preorder search is also called backtracking. The steps for traversing a binary tree in preorder traversal are:

1. Visit the root.
2. Visit the left subtree, using preorder.
3. Visit the right subtree, using preorder.

The algorithm for preorder traversal is as follows:
Algorithm Preorder (t)

// t is a binary tree. Each node of t has three fields; lchild, data, and rchild.

```
    {

        If( t ≠0)then

        {

                visit(t);

                Preorder (t→lchild);
                Preorder (t→rchild);

        }

    }
```

   Postorder Traversal:

   In a Postorder traversal, each root is visited after its left and right subtrees have been traversed. The steps for traversing a binary tree in postorder traversal are:

1. Visit the left subtree, using postorder.
2. Visit the right subtree, using postorder
3. Visit the root

The algorithm for preorder traversal is as follows:
Algorithm Postorder (t)
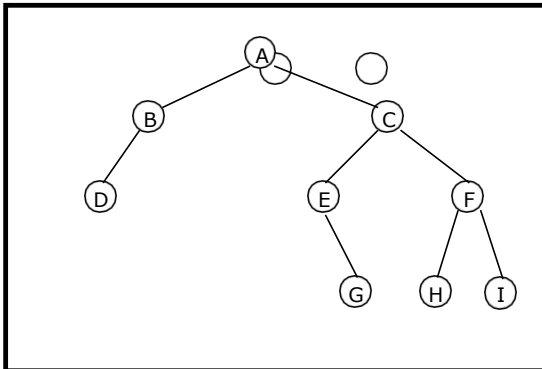
// t is a binary tree. Each node of t has three fields : lchild, data, and rchild.

```
    {

        If( t ≠0)then

        {

                Postorder(t→  child);
                Postorder(t→rchild);
                visit(t);

        }   }
```

Examples for binary tree traversal/search technique:

Example1:

Traverse the following binary tree in pre, post and in-order.

Preordof the vertices: A, B,
D, C, E, G, F, H, I.

Post order of the vertices: D,
B, G, E, H, I, F, C, A.

Inorder of the vertices: D,
B, A, E, G, C, H, F, I

Binary Tree   Pre,Post and In-order Traversing

Non Recursive Binary Tree Traversal Algorithms:
At first glance, it appears we would always want to use the flat traversal functions since the use less stack space. But the flat versions are not necessarily better. For instance, some overhead is associated with the use of an explicit stack, which may negate the savings we gain from storing only node pointers. Use of the implicit function call stack may actually be faster due to special machine instructions that can be used.

Inorder Traversal:
Initially push zero onto stack and then set root as vertex. Then repeat the following steps until the stack is empty:

1. Proceed down the left most path rooted at vertex, pushing each vertex onto the stack and stop when there is no left son of vertex.
2. Pop and process the nodes on stack if zero is popped then exit. If a vertex with right son exists, then set right son of vertex as current vertex and return to step one.

The algorithm for inorder Non Recursive traversal is asfollows:
Algorithm **inorder**()

{

      stack[1] = 0
      vertex =root

top: while(vertex ≠0)

      {

```
                push the vertex into the
                stack                     vertex
                =leftson(vertex)

        }

        pop the element from the stack and make it as vertex

        while(vertex ≠0)

        {

                print the vertex node
                if(rightson(vertex)
                ≠0)

                {

                        vertex                  =
                        rightson(vertex)    goto
                        top

                }

                pop the element from the stack and made it as vertex

        }

}
```

Preorder Traversal:

  Initially push zero onto stack and then set root as vertex. Then repeat the following steps until the stack is empty:

1. Proceed down the left most path by pushing the right son of vertex onto stack, if any and process each vertex. The traversing ends after a vertex with no left child exists.
2. Pop the vertex from stack, if vertex ≠ 0 then return to step one otherwise exit.

The algorithm for preorder Non Recursive traversal is as follows:

  Algorithm **preorder**()

```
{

        stack[1]:  =   0
        vertex  :=   root.
        while(vertex ≠0)

        {

                print   vertex   node
                if(rightson(vertex)
                ≠0)
```

push the right son of vertex into the

stack. if(leftson(vertex) ≠0)

vertex :=leftson(vertex)

else

pop the element from the stack and made it as vertex

}
}

Postorder Traversal:

Initially push zero onto stack and then set root as vertex. Then repeat the following steps until the stack is empty:

1. Proceed down the left most path rooted at vertex. At each vertex of path push vertex on to stack and if vertex has a right son push –(right son of vertex) onto stack.

2. Pop and process the positive nodes (left nodes). If zero is popped then exit. If a negative node is popped, then ignore the sign and return to step one.

The algorithm for postorder Non Recursive traversal is as follows:
   Algorithm **postorder**()

```
{
        stack[1] := 0
        vertex:=root

top: while(vertex ≠0)

        {
                push vertex onto stack
                if(rightson(vertex) ≠0)

                        push -(vertex) onto stack
                vertex :=leftson(vertex)

        }
        pop from stack and make it as
        vertex while(vertex >0)

        {
                print the vertex node

                pop from stack and make it as vertex

        }
        if(vertex <0)
        {
                vertex :=-(vertex)
                goto top
        }
}
```

Example1:
Traverse the following binary tree in pre, post and inorder using non-recursive traversing algorithm.
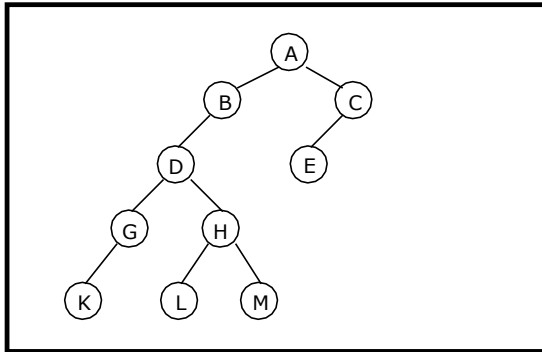
- Preorder traversal yields: A, B, D, G , K, H, L, M , C , E

- Postorder t raversal yields: K, G , L, M , H, D, B, E, C , A

- Inorder traversal yields: K, G , D, L, H, M , B, A, E, C

Inorder Traversal:

Initially push zero onto stack and then set root as vertex. Then repeat the following steps until the stack is empty:

1. Proceed down the left most path rooted at vertex, pushing each vertex onto the stack and stop when there is no left son of vertex.
2. Pop and process the nodes on stack if zero is popped then exit. If a vertex with right son exists, then set right son of vertex as current vertex and return to step one.

Current vertex	Stack	Processed nodes	Remarks
A	0		PUSH0
	0 A B D GK		PUSH the left most path ofA
K	0 A B DG	K	POPK
G	0 A BD	KG	POP G since K has no right son
D	0 AB	K GD	POP D since G has no right son
H	0 AB	K GD	Make the right son of D as vertex
H	0 A B HL	K GD	PUSH the leftmost path of H
L	0 A BH	K G DL	POPL
H	0 AB	K G D LH	POP H since L has no right son
M	0 AB	K G D LH	Make the right son of H as vertex
	0 A BM	K G D LH	PUSH the left most path of M
M	0 AB	K G D L HM	POPM
B	0A	K G D L H MB	POP B since M has no right son
A	0	K G D L H M BA	Make the right son of A as vertex
C	0 CE	K G D L H M BA	PUSH the left most path of C
E	0C	K G D L H M B AE	POPE
C	0	K G D L H M B A EC	Stop since stack is empty

Postorder Traversal:
Initially push zero onto stack and then set root as vertex. Then repeat the following steps until the stack is empty:

1. Proceed down the left most path rooted at vertex. At each vertex of path push vertex on to stack and if vertex has a right son push -(right son of vertex) onto stack.
2. Pop and process the positive nodes (left nodes). If zero is popped then exit. If a negative node is popped, then ignore the sign and return to step one.

Current	Stack	Processed nodes	Remark
A	0		PUSH0
	0 A -C B D -H GK		PUSH the left most path of A with a -ve for right sons
	0 A -C B D-H	KG	POP all +ve nodes K and G
H	0 A -C BD	KG	Pop H
	0 A -C B D H -ML	KG	PUSH the left most path of H with a -ve for right sons
	0 A -C B D H-M	K GL	POP all +ve nodes L
M	0 A -C B DH	K GL	PopM
	0 A -C B D HM	K GL	PUSH the left most path of M with a -ve for rightsons
	0 A-C	K G L M H DB	POP all +ve nodes M, H, D
C	0A	K G L M H DB	PopC
	0 A CE	K G L M H DB	PUSH the left most path of C with a -ve for rightsons
	0	K G L M H D B E	POP all +ve nodes E, C andA
	0		Stop since stack isempty

Preorder Traversal:
Initially push zero onto stack and then set root as vertex. Then repeat the following steps until the stack is empty:

1. Proceed down the left most path by pushing the right son of vertex onto stack, if any and process each vertex. The traversing ends after a vertex with no left child exists.
2. Pop the vertex from stack, if vertex $\neq 0$ then return to step one otherwise exit.

Current vertex	Stack	Processednodes	Remarks
A	0		PUSH0
	0 CH	A B D GK	PUSH the right son of each vertex onto stack and process each vertex in the left most path
H	0C	A B D GK	POPH

	0 CM	A B D G K HL	PUSH the right son of each vertex onto stack and process each vertex in the left most path
M	0C	A B D G K HL	POPM
	0C	A B D G K H LM	PUSH the right son of each vertex onto stack and process each vertex in the left most path; M has no leftpath
C	0	A B D G K H LM	PopC
	0	A B D G K H L M CE	PUSH the right son of each vertex onto stack and process each vertex in the left most path; C has no right son
	0	A B D G K H L M CE	Stop since stack is empty

Subgraphs and SpanningTrees:

Subgraphs: A graph G' = (V$^{'}$, E$^{'}$) is a subgraph of graph G = (V, E) iff V' ⊓V and E'⊓ E.

The undirected graph G is connected, if for every pair of vertices u, v there exists a path from u to v. If a graph is not connected, the vertices of the graph can be divided into **connected components**. Two vertices are in the same connected component iff they are connected by a path.

Tree is a connected acyclic graph. A **spanning tree** of a graph G = (V, E) is a tree that contains all vertices of V and is a subgraph of G. A single graph can have multiple spanning trees.

Lemma 1: *Let T be a spanning tree of a graph G. Then*

1. *Any two vertices in T are connected by a unique simple path.*
2. *If any edge is removed from T, then T becomes disconnected.*
3. *If we add any edge into T, then the new graph will contain a cycle.*
4. *Number of edges in T isn-1.*

Minimum Spanning Trees(MST):

A spanning tree for a connected graph is a tree whose vertex set is the same as the vertex set of the given graph, and whose edge set is a subset of the edge set of the given graph. i.e., any connected graph will have a spanning tree.

Weight of a spanning tree w (T) is the sum of weights of all edges in T. The Minimum spanning tree (MST) is a spanning tree with the smallest possible weight.
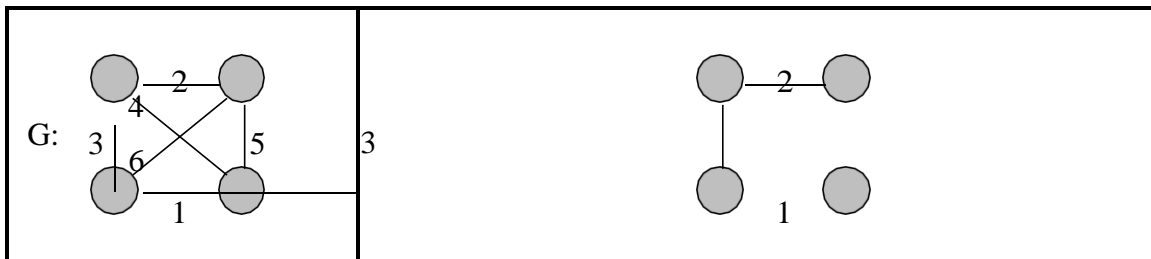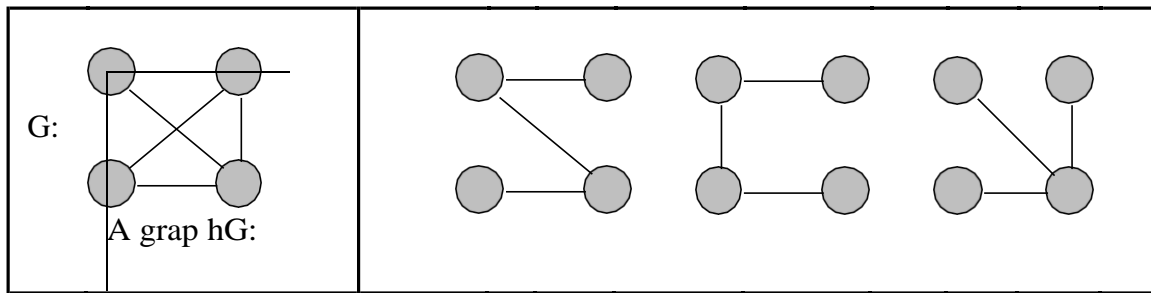
A weighted graphG: Theminimalspanningtreefromweightedgraph G:

Examples:

To explain the Minimum Spanning Tree, let's consider a few real-world examples:

1. One practical application of a MST would be in the design of a network. For instance, a group of individuals, who are separated by varying distances, wish to be connected together in a telephone network. Although MST cannot do anything about the distance from one connection to another, it can be used to determine the least cost paths with no cycles in this network, thereby connecting everyone at a minimum cost.

2. Another useful application of MST would be finding airline routes. The vertices of the graph would represent cities, and the edges would represent routes between the cities. Obviously, the further one has to travel, the more it will cost, so MST can be applied to optimize airline routes by finding the least costly paths with no cycles.

To explain how to find a Minimum Spanning Tree, we will look at two algorithms: the Kruskal algorithm and the Prim algorithm. Both algorithms differ in their methodology, but both eventually end up with the MST. Kruskal's algorithm uses edges, and Prim's algorithm uses vertex connections in determining the MST.

Kruskal's Algorithm

This is a greedy algorithm. A greedy algorithm chooses some local optimum(i.e. picking an edge with the least weight in a MST).

Kruskal's algorithm works as follows: Take a graph with 'n' vertices, keep on adding the shortest (least cost) edge, while avoiding the creation of cycles, until (n - 1) edges have been added. Sometimes two or more edges may have the same cost. The order in which the edges are chosen, in this case, does not matter. Different MSTs may result, but they will all have the same total cost, which will always be the minimum cost.

Algorithm:

The algorithm for finding the MST, using the Kruskal's method is as follows:

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;

                           // Each vertex is in a different set.

       i := 0; mincost :=0.0;
       while ((i < n -1) and (heap not empty))do
       {
              Delete a minimum cost edge (u, v) from the heap and re-
              heapify using Adjust;
              j := Find (u); k := Find(v); if
              (j □k)then
              {
                   i := i +1;
                   t [i, 1] := u; t [i, 2] := v; mincost
                   :=mincost + cost [u,v]; Union
                   (j,k);
              }
       }
       if (i □n-1) then write ("no spanning tree"); else
       return mincost;
}

Running time:

- The number of finds is at most 2e, and the number of unions at most n-1. Including the initialization time for the trees, this part of the algorithm has a complexity that is just slightly more than O (n +e).
- We can add at most n-1 edges to tree T. So, the total time for operations on T is O(n).

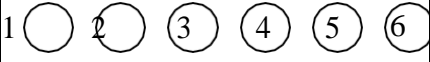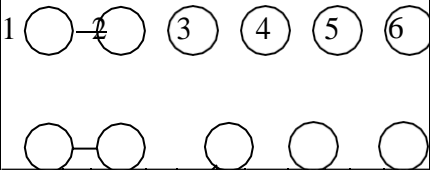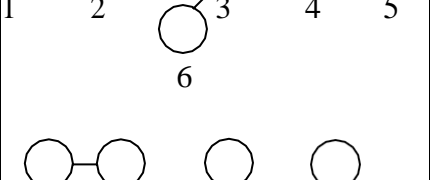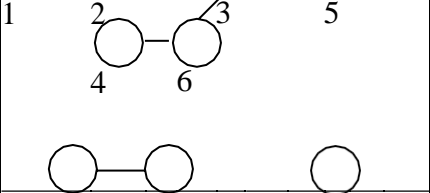Summing up the various components of the computing times, we get O (n + e log e) as asymptotic complexity
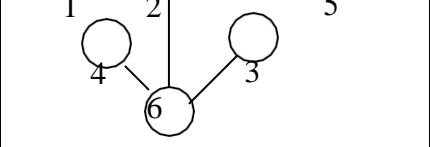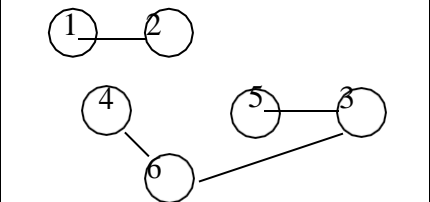
Example1:
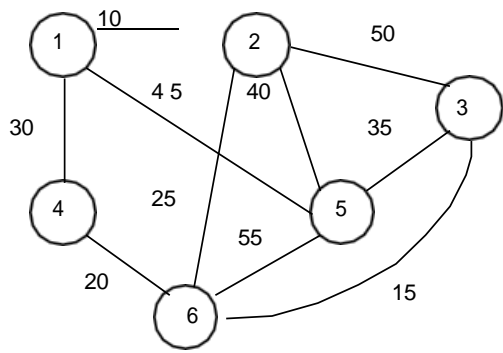
Arrange all the edges in the increasing order of their costs:

Cost	10	15	20	25	30	35	40	45	50	55
Edge	(1,2)	(3,6)	(4,6)	(2,6)	(1,4)	(3,5)	(2,5)	(1,5)	(2,3)	(5,6)

The edge set T together with the vertices of G define a graph that has up to n connected components. Let us represent each component by a set of vertices in it. These vertex sets are disjoint. To determine whether the edge (u, v) creates a cycle, we need to check whether u and v are in the same vertex set. If so, then a cycle is created. If not then no cycle is created. Hence two **Find**s on the vertex sets suffice. When an edge is included in T, two components are combined into one and a **union** is to be performed on the two sets.

Edge	Cost	Spanning Forest		
(1, 2)	10			
(3, 6)	15			
(4, 6)	20			
(2, 6)	25			
(1, 4)	30	Reject		
(3, 5)	35			

MINIMUM-COST SPANNING TREES: PRIM'SALGORITHM

A given graph can have many spanning trees. From these many spanning trees, we have to select a cheapest one. This tree is called as minimal cost spanning tree.

Minimal cost spanning tree is a connected undirected graph G in which each edge is labeled with a number (edge labels may signify lengths, weights other than costs). Minimal cost spanning tree is a spanning tree for which the sum of the edge labels is as small as possible

The slight modification of the spanning tree algorithm yields a very simple algorithm for finding an MST. In the spanning tree algorithm, any vertex not in the tree but connected to it by an edge can be added. To find a Minimal cost spanning tree, we must be selective - we must always add a new vertex for which the cost of the new edge is as small as possible.

This simple modified algorithm of spanning tree is called prim's algorithm for finding an Minimal cost spanning tree.

Prim's algorithm is an example of a greedy algorithm.

Algorit

hm

Algorit

hm

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 number or if no edge (i, j)exists.
// A minimum spanning tree is computed and stored as a set of
// edges in the array t [1:n-1, 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 of minimum cost in E;
        mincost := cost [k,l];
        t [1, 1] := k; t [1, 2] :=l;
        for   i :=1 to n do                            //Initialize near if
                (cost [i, l] < cost [i, k]) then near [i] :=l;
                else near [i] :=  k;
        near [k] :=near [l] :=0;
```

```
        for  i:=2 to n - 1do                              // Find n - 2 additional edges fort.
        {
                Let j be an index such that near [j] □0and
                cost [j, near [j]] is minimum;
                t [i, 1] := j; t [i, 2] := near [j]; mincost :=
                mincost + 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
                                [k] :=j;
        }
        return mincost;
}
```

Running time:

We do the same set of operations with dist as in Dijkstra's algorithm (initialize structure, m times decrease value, n - 1 times select minimum). Therefore, we get $O(n^2)$ time when we implement dist with array, $O(n + E \log n)$ when we implement it with a heap.

For each vertex u in the graph we dequeue it and check all its neighbors in $O(1 + \deg(u))$ time.
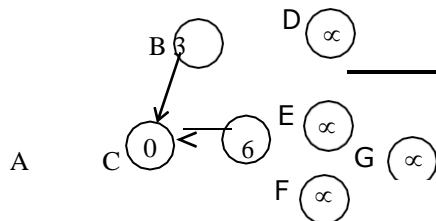
EXAMPLE1:

Use Prim's Algorithm to find a minimal spanning tree for the graph shown below starting with the vertex A.
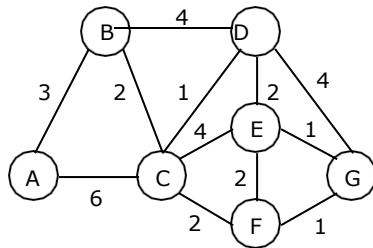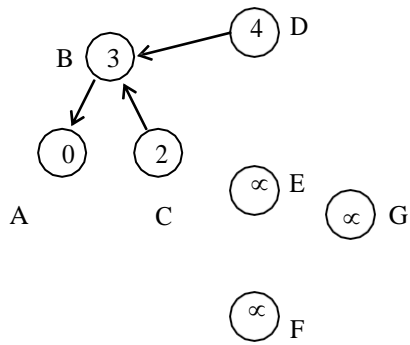

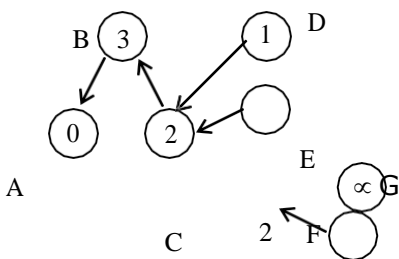The stepwise progress of the prim's algorithm is as follows:

Step1:

Vertex	A	B	C	D	E	F	G
Status	0	1	1	1	1	1	1
Dist.	0	3	6	∝	∝	∝	∝
Next	*	A	A	A	A	A	A

Step2:

Vertex	A	B	C	D	E	F	G
Status	0	0	1	1	1	1	1
Dist.	0	3	2	4	∝	∝	∝
Next	*	A	B	B	A	A	A

Step3:

Vertex	A	B	C	D	E	F	G
Status	0	0	0	1	1	1	1
Dist.	0	3	2	1	4	2	∝
Next	*	A	B	C	C	C	A

Step4:
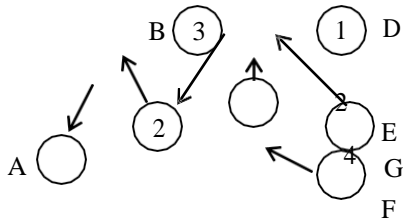
Vertex	A	B	C	D	E	F	G
Status	0	0	0	0	1	1	1
Dist.	0	3	2	1	2	2	4
Next	*	A	B	C	D	C	D

Step5:
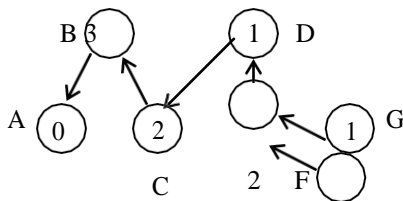
Vertex	A	B	C	D	E	F	G
Status	0	0	0	0	1	0	1
Dist.	0	3	2	1	2	2	1
Next	*	A	B	C	D	C	E

Step6:
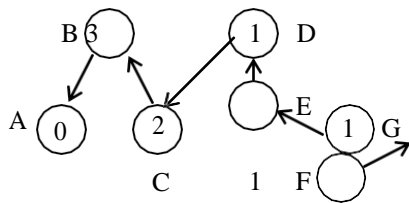
Vertex	A	B	C	D	E	F	G
Status	0	0	0	0	0	1	0
Dist.	0	3	2	1	2	1	1
Next	*	A	B	C	D	G	E

Step7:
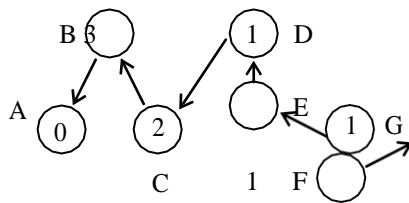
Vertex	A	B	C	D	E	F	G
Status	0	0	0	0	0	0	0
Dist.	0	3	2	1	2	1	1
Next	*	A	B	C	D	G	E

GRAPH ALGORITHMS
Basic Definitions:

- **Graph G** is a pair (V, E), where V is a finite set (set of vertices) and E is a finite set of pairs from V (set of edges). We will often denote n := |V|, m :=|E|.
- Graph G can be **directed**, if E consists of ordered pairs, or undirected, if E consists of unordered pairs. If (u, v) □E, then vertices u, and v are adjacent.
- We can assign weight function to the edges: $w_G(e)$ is a weight of edge e □E. The graph which has such function assigned is called **weighted graph**.
- **Degree** of a vertex v is the number of vertices u for which (u, v) □E (denote deg(v)). The number of **incoming edges** to a vertex v is called **in–degree** of the vertex (denote indeg(v)). The number of **outgoing edges** from a vertex is called **out-degree** (denote outdeg(v)).

Representation of Graphs:

Consider graph G = (V, E), where V= {v1,v2,….,vn}.

Adjacency matrix represents the graph as an n x n matrix $A = (a_{i,j})$, where

The matrix is symmetric in case of undirected graph, while it may be asymmetric if the graph is directed.

We may consider various modifications. For example for weighted graphs, we may have

Where default is some sensible value based on the meaning of the weight function (for example, if weight function represents length, then default can be⬚, meaning value larger than any other value).

Adjacency List: An array Adj [1 . . . . n] of pointers where for 1 ≤v ≤n, Adj [v] points to a linked list containing the vertices which are adjacent to v (i.e. the vertices that can be reached from v by a single edge). If the edges have weights then these weights may also be stored in the linked list elements.
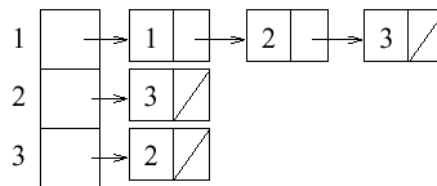
Adjacency matrix   Adjacency list

Paths and Cycles:

A path is a sequence of vertices (v1, v2, ... , vk), where for all i, (vi, vi+1) ⬚E. **A path is simple** if all vertices in the path are distinct.

A (simple) cycle is a sequence of vertices (v1, v2, ... , vk, vk+1 = v1), where for all i, (vi, vi+1) ⬚E and all vertices in the cycle are distinct except pair v1,vk+1.

Techniques forgraphs:
Given a graph G = (V, E) and a vertex V in V (G) traversing can be done in two ways.
1. Depth first search
2. Breadth first search

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.

This is also called connected graph or spanning tree.

Spanning trees obtained using BFS then it called breadth first spanning trees

```
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 do
If (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 1st 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.

```
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);
{
```
54

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

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

Bi-connected Components:

A graph G is biconnected, iff (if and only if) it contains no articulation point (joint or junction).

A vertex v in a connected graph G is an articulation point, if and only if (iff) the deletion of vertex v together with all edges incident to v disconnects the graph into two or more none empty components.

The presence of articulation points in a connected graph can be an undesirable(un wanted) feature in many cases.
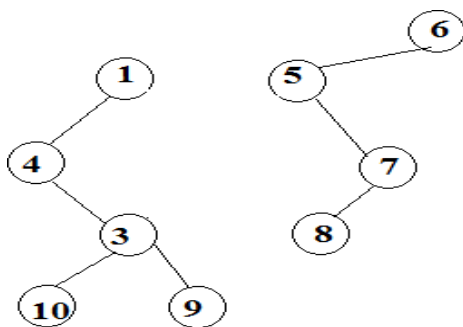
For example

if G1→Communication network with
    Vertex → communication stations.
    Edges→ Communication lines.

Then the failure of a communication station I that is an articulation point, then we loss the communication in between other stations. F
Form graph G1

After deleting vertex (2)

There is an efficient algorithm to test whether a connected graph is biconnected. If the case of graphs that are not biconnected, this algorithm will identify all the articulation points.

Once it has been determined that a connected graph G is not biconnected, it may be desirable (suitable) to determine a set of edges whose inclusion makes the graph biconnected.

GREEDY METHOD AND DYNAMIC PROGRAMMING

GENERALMETHOD

Greedy is the most straight forward design technique. Most of the problems have n inputs and require us to obtain a subset that satisfies some constraints. Any subset that satisfies these constraints is called a feasible solution. We need to find a feasible solution that either maximizes or minimizes the objective function. A feasible solution that does this is called an optimal solution.

The greedy method is a simple strategy of progressively building up a solution, one element at a time, by choosing the best possible element at each stage. At each stage, a decision is made regarding whether or not a particular input is in an optimal solution. This is done by considering the inputs in an order determined by some selection procedure. If the inclusion of the next input, into the partially constructed optimal solution will result in an infeasible solution then this input is not added to the partial solution. The selection procedure itself is based on some optimization measure. Several optimization measures are plausible for a given problem. Most of them, however, will result in algorithms that generate sub-optimal solutions. This version of greedy technique is called *subset paradigm*. Some problems like Knapsack, Job sequencing with deadlines and minimum cost spanning trees are based on *subset paradigm.*

For the problems that make decisions by considering the inputs in some order, each decision is made using an optimization criterion that can be computed using decisions already made. This version of greedy method is *ordering paradigm*. Some problems like optimal storage on tapes, optimal merge patterns and single source shortest path are based on *ordering paradigm.*

CONTROLABSTRACTION

Algorithm Greedy (a,n)
```
// a(1 : n) contains the 'n' inputs
{
        solution:=∅ ;    // initialize the solution to be empty
        for i:=1 to ndo
        {
                x := select(a);
                if feasible (solution, x)then
                        solution := Union (Solution,x);
        }
        return solution;
}
```

Procedure Greedy describes the essential way that a greedy based algorithm will look, once a particular problem is chosen and the functions select, feasible and union are properly implemented.

The function select selects an input from 'a', removes it and assigns its value to 'x'. Feasible is a Boolean valued function, which determines if 'x' can be included into the solution vector. The function Union combines 'x' with solution and updates the objective function.

KNAPSACK PROBLEM

  Let us apply the greedy method to solve the knapsack problem. We are given 'n' objects and a knapsack. The object 'i' has a weight w_i and the knapsack has a capacity 'm'. If a fraction x_i, $0 < x_i < 1$ of object i is placed into the knapsack then a profit of $p_i x_i$ is earned. The objective is to fill the knapsack that maximizes the total profit earned.

Since the knapsack capacity is 'm', we require the total weight of all chosen objects to be at most 'm'. The problem is stated as:

$$\text{Maximize} \sum_{i=1}^{n} v_i x_i$$

subject to

$$\sum_{i=1}^{n} w_i x_i \leqslant W, \qquad x_i \in \{0, 1\}$$

The profits and weights are positive numbers.

Algorithm

If the objects are already been sorted into non-increasing order of p[i] / w[i] then the algorithm given below obtains solutions corresponding to this strategy.

Algorithm GreedyKnapsack (m,n)

// P[1 : n] and w[1 : n] contain the profits and weights respectively of

// Objects ordered so that p[i] / w[i]> p[i + 1] / w[i + 1].

// m is the knapsack size and x[1: n] is the solution vector.

```
{
        for i := 1 to n do
         x[i] :=0.0 ;     //initialize the solution vector
         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];
}
```

Running time:

The objects are to be sorted into non-decreasing order of p_i / w_i ratio. But if we disregard the time to initially sort the objects, the algorithm requires only O(n)time.

Example:

Consider the following instance of the knapsack problem: n = 3, m = 20, (p1, p2, p3) = (25, 24, 15) and (w1, w2, w3) = (18, 15,10).

1. First, we try to fill the knapsack by selecting the objects in some order:

x1	x2	x3	$\sum w_i x_i$	$\sum p_i x_i$
1/2	1/3	1/4	18 x 1/2 + 15 x 1/3 + 10 x1/4 =16.5	25 x 1/2 + 24 x 1/3 + 15 x 1/4= 24.25

2. Select the object with the maximum profit first (p = 25). So, x1 = 1 and profit earned is 25. Now, only 2 units of space is left, select the object with next largest profit (p = 24). So, x2 =2/15

x1	x2	x3	$\sum w_i x_i$	$\sum p_i x_i$
1	2/15	0	18 x 1 + 15 x 2/15 =20	25 x 1 + 24 x 2/15 =28.2

3. Considering the objects in the order of non-decreasing weights w_i.

x1	x2	x3	$\sum w_i x_i$	$\sum p_i x_i$
0	2/3	1	15 x 2/3 + 10 x 1 =20	24 x 2/3 + 15 x 1 =31

4. Considered the objects in the order of the ratio p_i / w_i.

p1/w1	**p2/w2**	**p3/w3**
25/18	24/15	15/10
1.4	1.6	1.5

Sort the objects in order of the non-increasing order of the ratio p_i / x_i. Select the object with the maximum p_i / x_i ratio, so, $x_2 = 1$ and profit earned is 24. Now, only 5 units of space is left, select the object with next largest p_i / x_i ratio, so $x_3 = ½$ and the profit earned is 7.5.

x1	x2	x3	$\sum w_i x_i$	$\sum p_i x_i$
0	1	1/2	15 x 1 + 10 x 1/2 =20	24 x 1 + 15 x 1/2 =31.5

This solution is the optimal solution.

JOB SEQUENCING WITHDEADLINES

Given a set of 'n' jobs. Associated with each Job i, deadline $d_i \geq 0$ and profit $P_i \geq 0$. For any job 'i' the profit pi is earned iff the job is completed by its deadline. Only one machine is available for processing jobs. An optimal solution is the feasible solution with maximum profit.

Sort the jobs in 'j' ordered by their deadlines. The array d [1 : n] is used to store the deadlines of the order of their p-values. The set of jobs j [1 : k] such that j [r], $1 \leq r \leq k$ are the jobs in 'j' and d (j [1]) $\leq$ d (j[2]) $\leq$ . . . $\leq$ d (j[k]). To test whether J U {i} is feasible, we have just to insert i into J preserving the deadline ordering and then verify that d [J[r]] $\leq$ r, $1 \leq r \leq k+1$.

Example:
Let n=4,(P1,P2,P3,P4,)=(100,10,15,27)and(d1 d2 d3 d4)=(2,1,2,1).The feasible solutions and their values are:

Sl.No	Feasible Solution	Procuring sequence	Value	Remarks
1	1,2	2,1	110	
2	1,3	1,3 or3,1	115	

60

3	1,4	4,1	127	**OPTIMA**
4	2,3	2,3	25	
5	3,4	4,3	42	
6	1	1	100	
7	2	2	10	
8	3	3	15	
9	4	4	27	

Algorithm:

The algorithm constructs an optimal set J of jobs that can be processed by their deadlines.

Algorithm GreedyJob (d, J,n)

// J is a set of jobs that can be completed by their deadlines.

```
{
        J :={1};
        for i := 2 to ndo
        {
                if (all jobs in J U {i} can be completed by their deadlines) then J
                := J U{i};
        }
}
```
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$\rightarrow$ dead line, j$\rightarrow$subset of jobs ,n$\rightarrow$ total number of jobs
// $d[i] \geq 1$ $1 \leq i \leq n$ are the dead lines,
// the jobs are ordered such that $p[1] \geq p[2] \geq$---$\geq p[n]$
//j[i] is the ith job in the optimal solution $1 \leq i \leq k$, k$\rightarrow$ 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;
}
```

The Single Source Shortest-Path Problem: DIJKSTRA'SALGORITHMS

In the previously studied graphs, the edge labels are called as costs, but here we think them as lengths. In a labeled graph, the length of the path is defined to be the sum of the lengths of its edges.

In the single source, all destinations, shortest path problem, we must find a shortest path from a given source vertex to each of the vertices (called destinations) in the graph to which there is a path.

Dijkstra's algorithm is similar to prim's algorithm for finding minimal spanning trees. Dijkstra's algorithm takes a labeled graph and a pair of vertices P and Q, and finds the shortest path between then (or one of the shortest paths) if there is more than one. The principle of optimality is the basis for Dijkstra'salgorithms.Dijkstra's algorithm does not work for negative edges at all.
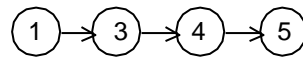
The figure lists the shortest paths from vertex 1 for a five vertex weighted digraph.
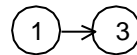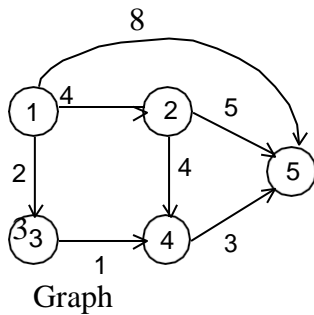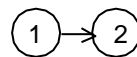
Graph

Shortest Paths

Algorithm:

Algorithm Shortest-Paths (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 the digraph G with n vertices.

// dist [v] is set to zero. G is represented by its

62

// cost adjacency matrix cost [1:n,1:n].

{

     for i :=1 to n do

     {

          S [i]:=false;   //Initialize   S.

          dist [i] :=cost [v,i];

     }

     S[v] := true; dist[v] :=0.0;   // Put v in S.

     for num := 2 to n – 1do

     {

          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 is S.

          for (each w adjacent to u with S [w] = false)do

               if (dist [w] > (dist [u] + cost [u, w])then   //Update   distances

                    dist [w] := dist [u] + cost [u,w];

     }

}

Runningtime:

Depends on implementation of data structures fordist.

- Build a structure with nelements   A
- at most m = ⊓E ⊓times decrease the value of anitem   mB
- 'n' times select the smallestvalue   nC
- For <u>array</u> A = O (n); B = O (1); C = O (n) which gives O (n^2)total.
- For <u>heap</u> A = O (n); B = O (log n); C = O (log n) which gives O (n + m logn) total.

     Example1:

Use Dijkstras algorithm to find the shortest path from A to each of the other six vertices in

the graph:
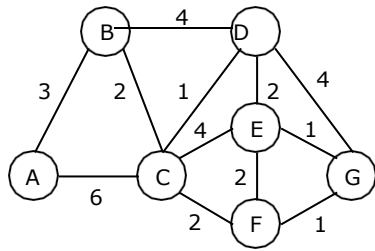
The problem is solved by considering the following information:

- Status[v] will be either '0', meaning that the shortest path from v to v0 has definitely been found; or '1', meaning that it hasn't.

- Dist[v] will be a number, representing the length of the shortest path from v to v0 found so far.

- Next[v] will be the first vertex on the way to v0 along the shortest path found so far from v to v0
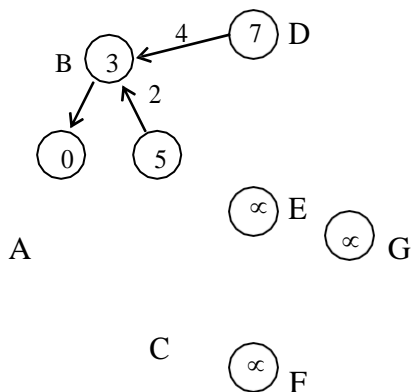
The progress of Dijkstra's algorithm on the graph shown above is as follows:
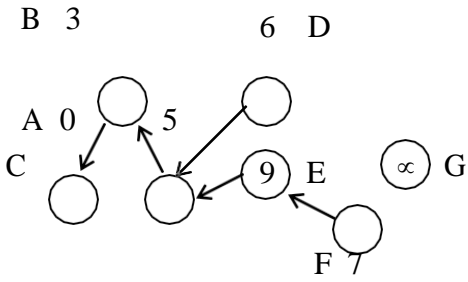
Step1:
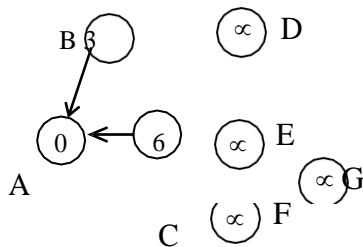
Vertex	A	B	C	D	E	F	G
Status	0	1	1	1	1	1	1
Dist.	0	3	6	∝	∝	∝	∝
Next	*	A	A	A	A	A	A

Step2:

Vertex	A	B	C	D	E	F	G
Status	0	0	1	1	1	1	1
Dist.	0	3	5	7	∝	∝	∝
Next	*	A	B	B	A	A	A

64

Step3:

Vertex	A	B	C	D	E	F	G
Status	0	0	0	1	1	1	1
Dist.	0	3	5	6	9	7	∝
Next	*	A	B	C	C	C	A

Step4:
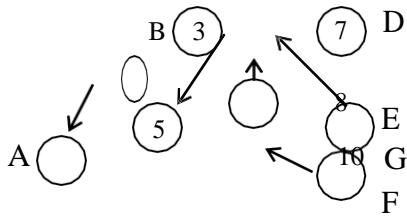
Vertex	A	B	C	D	E	F	G
Status	0	0	0	0	1	1	1
Dist.	0	3	5	6	8	7	10
Next	*	A	B	C	D	C	D

Step5:
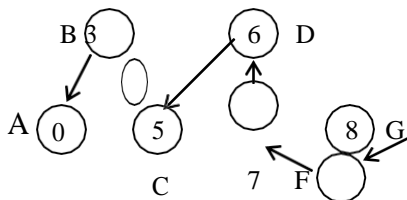
Vertex	A	B	C	D	E	F	G
Status	0	0	0	0	1	0	1
Dist.	0	3	5	6	8	7	8
Next	*	A	B	C	D	C	F

65

Step6:
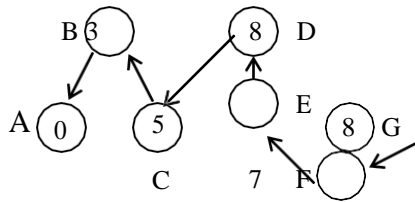
Vertex	A	B	C	D	E	F	G
Status	0	0	0	0	0	0	1
Dist.	0	3	5	6	8	7	8
Next	*	A	B	C	D	C	F

Step7:
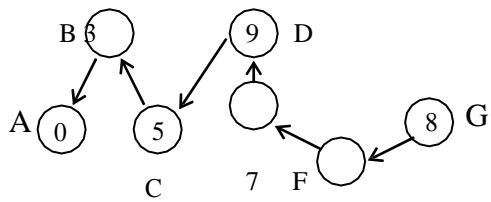
Vertex	A	B	C	D	E	F	G
Status	0	0	0	0	0	0	0
Dist.	0	3	5	6	8	7	8
Next	*	A	B	C	D	C	F

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.

Dynamic programming differs from the greedy method since the greedy method produces only one feasible solution, which may or may not be optimal, while dynamic programming produces all possible sub-problems at most once, one of which guaranteed to be optimal. Optimal solutions to sub-problems are retained in a table, thereby avoiding the work of recomputing the answer every time a sub-problem is encountered

The divide and conquer principle solve a large problem, by breaking it up into smaller problems which can be solved independently. In dynamic programming this principle is carried to an extreme: when we don't know exactly which smaller problems to solve, we simply solve them all, then store the answers away in a table to be used later in solving larger problems. Care is to be taken to avoid recomputing previously computed values, otherwise the recursive program will have prohibitive complexity. In some cases, the solution can be improved and in other cases, the dynamic programming technique is the best approach.

Two difficulties may arise in any application of dynamic programming:

1. It may not always be possible to combine the solutions of smaller problems to form the solution of a larger one.
2. The number of small problems to solve may be un-acceptably large.

There is no characterized precisely which problems can be effectively solved with dynamic programming; there are many hard problems for which it does not seen to be applicable, as well as many easy problems for which it is less efficient than standard algorithms.

5.1   MULTI STAGEGRAPHS

A multistage graph $G = (V, E)$ is a directed graph in which the vertices are partitioned into $k \geq 2$ disjoint sets V_i, $1 \leq i \leq k$. In addition, if $<u, v>$ is an edge in E, then $u \, V_i$ and $v \, V_{i+1}$ for some i, $1 \leq i < k$.

Let the vertex 's' is the source, and 't' the sink. Let c (i, j) be the cost of edge $<i, j>$. The cost of a path from 's' to 't' is the sum of the costs of the edges on the path. The multistage graph problem is to find a minimum cost path from 's' to 't'. Each set V_i defines a stage in the graph. Because of the constraints on E, every path from 's' to 't' starts in stage 1, goes to stage 2, then to stage 3, then to stage 4, and so on, and eventually terminates in stage k.

A dynamic programming formulation for a k-stage graph problem is obtained by first

noticingthateverystoppathistheresultofasequenceofk–2decisions.Thei^{th}

decision involves determining which vertex in v_{i+1}, $1 \leq i \leq k - 2$, is to be on the path. Let c (i, j) be the cost of the path from source to destination. Then using the forward approach, we obtain:

$$\text{cost (i, j)} = \min_{\text{l in } V_{i+1}} \{c (j, l) + \text{cost} (i + 1, l)\}$$

$$<j, l> \text{ in E}$$

ALGORITHM:
Algorithm Fgraph(G, k, n,p)

// The input is a k-stage graph $G = (V, E)$ with n vertices
// indexed in order or stages. E is a set of edges and c [i,j]
// is the cost of (i, j). p [1 : k] is a minimum cost path.
{
    cost [n] :=0.0;
    for j:= n - 1 to 1 step – 1do
    {   // compute cost[j]
        let r be a vertex such that (j, r) is an edge
        of G and c [j, r] + cost [r] is minimum;
        cost [j] := c [j, r] + cost[r];
        d [j] :=r:
    }
    p [1] := 1; p [k] :=n;   // Find a minimum cost path.
    for j := 2 to k - 1 do

p [j] := d [p [j -1]];
}
The multistage graph problem can also be solved using the backward approach. Let bp(i, j) be a minimum cost path from vertex s to j vertex in V_i. Let Bcost(i, j) be the cost of bp(i, j). From the backward approach we obtain:

Bcost (i, j) = min { Bcost (i –1, l) + c (l, j)}
1 in V_{i-1}

<l, j> in E

Algorithm Bgraph(G, k, n,p)

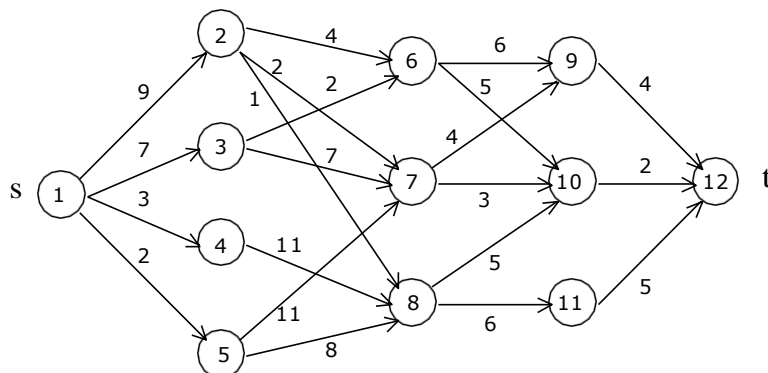// Same function asFgraph
{
      Bcost [1] :=0.0;
      for j := 2 to ndo
      {   // Compute Bcost[j].
            Let r be such that (r, j) is an edge of
            G and Bcost [r] + c [r, j] is
            minimum; Bcost [j] := Bcost [r] + c
            [r,j];
            D [j] :=r;
      }   //find a minimum costpath
      p [1] := 1; p [k] :=n;
      for j:= k - 1 to 2 do p [j] := d [p [j +1]];
}

EXAMPLE1:

Find the minimum cost path from s to t in the multistage graph of five stages shown below. Do this first using forward approach and then using backward approach.

FORWARDAPPROACH:

We use the following equation to find the minimum cost path from s to t:

cost (i, j) = min $\{c (j, l) + cost (i + 1,l)\}$

l inV_{i+1}

$<j, l>$inE

cost $(1, 1)$ = min $\{c (1, 2) + cost (2, 2), c (1, 3) + cost (2, 3), c (1, 4) + cost (2,4),$
$c (1, 5) + cost (2,5)\}$
= min $\{9 + cost (2, 2), 7 + cost (2, 3), 3 + cost (2, 4), 2 + cost (2,5)\}$

──────────────

Now first starting with,

cost $(2, 2)$ = min$\{c (2, 6) + cost (3, 6), c (2, 7) + cost (3, 7), c (2, 8) + cost (3,8)\}$
= min $\{4 + cost (3, 6), 2 + cost (3, 7), 1 + cost (3,8)\}$

──────────────

cost$(3,6)$ = min $\{c (6, 9) + cost (4, 9), c (6, 10) + cost (4,10)\}$
= min $\{6 + cost (4, 9), 5 + cost (4,10)\}$

cost$(4,9)$ = min $\{c (9, 12) + cost (5, 12)\}$ = min $\{4 + 0)$ =4

cost $(4, 10)$ = min $\{c (10, 12) + cost (5, 12)\}$ =2

Therefore, cost $(3, 6)$ = min $\{6 + 4, 5 + 2\}$ =7

──────────────

cost$(3,7)$ = min $\{c (7, 9) + cost (4, 9) , c (7, 10) + cost (4,10)\}$
= min $\{4 + cost (4, 9), 3 + cost (4,10)\}$

cost$(4,9)$ = min $\{c (9, 12) + cost (5, 12)\}$ = min $\{4 + 0\}$ =4

Cost $(4, 10)$ = min $\{c (10, 2) + cost (5, 12)\}$ = min $\{2 + 0\}$ =2

Therefore, cost $(3, 7)$ = min $\{4 + 4, 3 + 2\}$ = min $\{8, 5\}$ =5

──────────────

cost$(3,8)$ = min $\{c (8, 10) + cost (4, 10), c (8, 11) + cost (4,11)\}$
= min $\{5 + cost (4, 10), 6 + cost (4 +11)\}$

cost (4, 11) = min {c (11, 12) + cost (5, 12)} =5

Therefore, cost (3, 8) = min {5 + 2, 6 + 5} = min {7, 11} =7

Therefore, cost (2, 2) = min {4 + 7, 2 + 5, 1 + 7} = min {11, 7, 8} =7

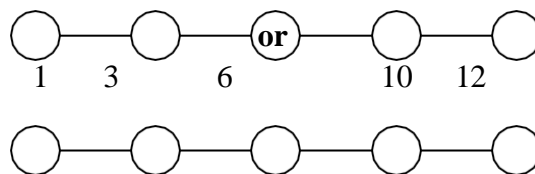Therefore, cost (2, 3) = min {c (3, 6) + cost (3, 6), c (3, 7) + cost (3,7)}
        = min {2 + cost (3, 6), 7 + cost (3,7)}
        = min {2 + 7, 7 + 5} = min {9, 12} =9

cost (2, 4) = min {c (4, 8) + cost (3, 8)} = min {11 + 7} =18
cost (2, 5) = min {c (5, 7) + cost (3, 7), c (5, 8) + cost (3,8)}
        = min {11 + 5, 8 + 7} = min {16, 15} =15

Therefore, cost (1, 1) = min {9 + 7, 7 + 9, 3 + 18, 2 +15}
                = min {16, 16, 21, 17} =16

The minimum cost path is16.

The path is   1   2   7   10   12

                1   3   6   10   12

BACKWARDAPPROACH:

We use the following equation to find the minimum cost path from t tos: Bcost (i,

J) = min {Bcost (i – 1, l) + c (l,J)}
            l in v_{i-1}

            <l,
            j>inE

Bcost (5, 12) = min {Bcost (4, 9) + c (9, 12), Bcost (4, 10) + c (10,12),
        Bcost (4, 11) + c (11,12)}
            = min {Bcost (4, 9) + 4, Bcost (4, 10) + 2, Bcost (4, 11) +5}

Bcost (4, 9) = min {Bcost (3, 6) + c (6, 9), Bcost (3, 7) + c (7,9)}
          = min {Bcost (3, 6) + 6, Bcost (3, 7) +4}
Bcost (3, 6) = min {Bcost (2, 2) + c (2, 6), Bcost (2, 3) + c (3,6)}
          = min {Bcost (2, 2) + 4, Bcost (2, 3) +2}

Bcost (2, 2) = min {Bcost (1, 1) + c (1, 2)} = min {0 + 9} =9

Bcost (2, 3) = min {Bcost (1, 1) + c (1, 3)} = min {0 + 7} =7

Bcost (3, 6) = min {9 + 4, 7 + 2} = min {13, 9} =9

Bcost (3, 7) = min {Bcost (2, 2) + c (2, 7), Bcost (2, 3) + c (3,7),
          Bcost (2, 5) + c (5,7)}

Bcost (2, 5) = min {Bcost (1, 1) + c (1, 5)} =2

Bcost (3, 7) = min {9 + 2, 7 + 7, 2 + 11} = min {11, 14, 13} =11

Bcost (4, 9) = min {9 + 6, 11 + 4} = min {15, 15} =15

Bcost (4, 10) = min {Bcost (3, 6) + c (6, 10), Bcost (3, 7) + c (7,10),
          Bcost (3, 8) + c (8,10)}

Bcost (3, 8) = min {Bcost (2, 2) + c (2, 8), Bcost (2, 4) + c (4,8),
          Bcost (2, 5) + c (5,8)}
Bcost (2, 4) = min {Bcost (1, 1) + c (1, 4)} =3

Bcost (3, 8) = min {9 + 1, 3 + 11, 2 + 8} = min {10, 14, 10} =10

Bcost (4, 10) = min {9 + 5, 11 + 3, 10 + 5} = min {14, 14, 15) =14

Bcost (4, 11) = min {Bcost (3, 8) + c (8, 11)} = min {Bcost (3, 8) +6}
          = min {10 + 6} =16

Bcost (5, 12) = min {15 + 4, 14 + 2, 16 + 5} = min {19, 16, 21} =16.

All pairs shortestpaths

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 ≠ 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:

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

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 ≤i ≤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 x n. Therefore, the algorithm has a complexity of $O(n^3)$.

General formula: $\min \{A^{k-1} (i, k) + A^{k-1} (k, j)\}, c (i,j)$
$\qquad 1 \le k \le n$

Solve the problem for different values of k = 1, 2 and3
Step 1: Solving the equation for, k =1;

$A^1 (1, 1) = \min \{(A^0 (1, 1) + A^0 (1, 1)), c (1, 1)\} = \min \{0 + 0, 0\} = 0$

$A^1 (1, 2) = \min \{(A^0 (1, 1) + A^0 (1, 2)), c (1, 2)\} = \min \{(0 + 4), 4\} = 4$

$A^1 (1, 3) = \min \{(A^0 (1, 1) + A^0 (1, 3)), c (1, 3)\} = \min \{(0 + 11), 11\} = 11$

$A^1 (2, 1) = \min \{(A^0 (2, 1) + A^0 (1, 1)), c (2, 1)\} = \min \{(6 + 0), 6\} = 6$

$A^1 (2, 2) = \min \{(A^0 (2, 1) + A^0 (1, 2)), c (2, 2)\} = \min \{(6 + 4), 0)\} = 0$

$A^1 (2, 3) = \min \{(A^0 (2, 1) + A^0 (1, 3)), c (2, 3)\} = \min \{(6 + 11), 2\} = 2$

$A^1 (3, 1) = \min \{(A^0 (3, 1) + A^0 (1, 1)), c (3, 1)\} = \min \{(3 + 0), 3\} = 3$

$A^1 (3, 2) = \min \{(A^0 (3, 1) + A^0 (1, 2)), c (3, 2)\} = \min \{(3 + 4), 0\} = 7$

$A^1 (3, 3) = \min \{(A^0 (3, 1) + A^0 (1, 3)), c (3, 3)\} = \min \{(3 + 11), 0\} = 0$

Step 2: Solving the equation for, K =2;

$A^2 (1, 1) = \min \{(A^1 (1, 2) + A^1 (2, 1), c (1, 1)\} = \min \{(4 + 6), 0\} = 0$

$A^2 (1, 2) = \min \{(A^1 (1, 2) + A^1 (2, 2), c (1, 2)\} = \min \{(4 + 0), 4\} = 4$

$A^2 (1, 3) = \min \{(A^1 (1, 2) + A^1 (2, 3), c (1, 3)\} = \min \{(4 + 2), 11\} = 6$

$A^2 (2, 1) = \min \{(A (2, 2) + A (2, 1), c (2, 1)\} = \min \{(0 + 6), 6\} = 6$

$A^2 (2, 2) = \min \{(A (2, 2) + A (2, 2), c (2, 2)\} = \min \{(0 + 0), 0\} = 0$

$A^2 (2, 3) = \min \{(A (2, 2) + A (2, 3), c (2, 3)\} = \min \{(0 + 2), 2\} = 2$

$A^2 (3, 1) = \min \{(A (3, 2) + A (2, 1), c (3, 1)\} = \min \{(7 + 6), 3\} = 3$

A^2 (3, 2) = min {(A (3, 2) + A (2, 2), c (3, 2)} = min {(7 + 0), 7} =7

A^2 (3, 3) = min {(A (3, 2) + A (2, 3), c (3, 3)} = min {(7 + 2), 0} =0

$$A^{(2)} = \begin{matrix} 0 & 4 \\ 6 & 0 \\ 3 & 7 \end{matrix}$$

Step 3: Solving the equation for, k =3;

$A^3 (1, 1) = \min \{A^2 (1, 3) + A^2 (3, 1), c (1, 1)\} = \min \{(6 + 3), 0\} = 0$

$A^3 (1, 2) = \min \{A^2 (1, 3) + A^2 (3, 2), c (1, 2)\} = \min \{(6 + 7), 4\} = 4$

$A^3 (1, 3) = \min \{A^2 (1, 3) + A^2 (3, 3), c (1, 3)\} = \min \{(6 + 0), 6\} = 6$

$A^3 (2, 1) = \min \{A^2 (2, 3) + A^2 (3, 1), c (2, 1)\} = \min \{(2 + 3), 6\} = 5$

$A^3 (2, 2) = \min \{A^2 (2, 3) + A^2 (3, 2), c (2, 2)\} = \min \{(2 + 7), 0\} = 0$

$A^3 (2, 3) = \min \{A^2 (2, 3) + A^2 (3, 3), c (2, 3)\} = \min \{(2 + 0), 2\} = 2$

$A^3 (3, 1) = \min \{A^2 (3, 3) + A^2 (3, 1), c (3, 1)\} = \min \{(0 + 3), 3\} = 3$

$A^3 (3, 2) = \min \{A^2 (3, 3) + A^2 (3, 2), c (3, 2)\} = \min \{(0 + 7), 7\} = 7$

$A^3 (3, 3) = \min \{A^2 (3, 3) + A^2 (3, 3), c (3, 3)\} = \min \{(0 + 0), 0\} = 0$

$$A^{(3)} = \begin{matrix} 0 & 4 & 6 \\ 5 & 0 & 2 \\ 3 & 7 & 0 \end{matrix}$$

TRAVELLING SALESPERSONPROBLEM

Let G = (V, E) be a directed graph with edge costs C_{ij}. The variable c_{ij} is defined such that $c_{ij} > 0$ for all I and j and $c_{ij} = \square$ if < i, j> $\square$ E. Let |V| = n and assume n > 1. A tour of G is a directed simple cycle that includes every vertex in V. The cost of a tour is the sum of the cost of the edges on the tour. The traveling sales person problem is to find a tour of minimum cost. The tour is to be a simple path that starts and ends at vertex1.

Let g (i, S) be the length of shortest path starting at vertex i, going through all vertices in S, and terminating at vertex 1. The function g (1, V − {1}) is the length of an optimal salesperson tour. From the principal of optimality it followsthat:

```
C(S, i) = min { C(S-{i}, j) + dis(j, i)} where j belongs to S, j != i and j != 1.
```

The Equation can be solved for g (1, V − 1}) if we know g (k, V − {1, k}) for all

116

choices of k.

Complexity Analysis:

Foreachvalueof|S|therearen–1choicesfori.ThenumberofdistinctsetsSof

size k not including 1 and i is $k \binom{}{} \cdot$
Hence, the total number of g (i, S)'s to be computed before computing g (1, V – {1})

To calculate this sum, we use the binominaltheorem:

This is Φ (n 2^{n-2}), so there are exponential number of calculate. Calculating one g (i, S) require finding the minimum of at most n quantities. Therefore, the entire algorithm is Φ (n^2 2^{n-2}). This is better than enumerating all n! different tours to find the best one. So, we have traded on exponential growth for a much smaller exponential growth. The most serious drawback of this dynamic programming solution is the space needed, which is O (n 2^n). This is too large even for modest values of n.

Example1:

For the following graph find minimum cost tour for the traveling sales person problem:
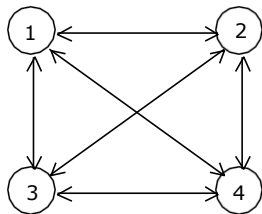
The cost adjacency matrix =

0	10	15	20
5	0	9	10
6	13	0	12
8	8	9	0

Let us start the tour from vertex1:

$$g (1, V – \{1\}) = \min \{c_{1k} + g (k, V – \{1,K\})\} \qquad - \qquad (1)$$
$2 \leq k \leq n$

117

More generally writing:

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

Clearly, $g(i, 0) = c_{i1}$, $1 \le i \le n$.

$$g(2, 0) = C_{21} = 5$$

$$g(3, 0) = C_{31} = 6$$

$$g(4, 0) = C_{41} = 8$$

Using equation – (2) we obtain:

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

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

$g(3, \{4\}) = \min \{c_{34} + g(4, 0)\} = 12 + 8 = 20$

$g(4, \{3\}) = \min \{c_{43} + g(3, 0)\} = 9 + 6 = 15$

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

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

$g(2, \{4\}) = \min \{c_{24} + g(4, 0)\} = 10 + 8 = 18$

$g(4, \{2\}) = \min \{c_{42} + g(2, 0)\} = 8 + 5 = 13$

Therefore, $g(3, \{2, 4\}) = \min \{13 + 18, 12 + 13\} = \min \{41, 25\} = 25$

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

$g(2, \{3\}) = \min \{c_{23} + g(3, 0)\} = 9 + 6 = 15$

$g(3, \{2\}) = \min \{c_{32} + g(2, 0)\} = 13 + 5 = 18$

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 SEARCHTREE

Let us assume that the given set of identifiers is $\{a_1, \ldots, a_n\}$ with $a_1 < a_2 < \ldots <$ 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 $a_i < x < a_{i+1}$, $0 \le i \le n$ (assume $a_0 = -$ and
☐ $a_{n+1} = ⊞$). 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:

$$\text{Expected Cost of tree} = \sum_{i=1}^{n} \text{cost}(k_i)p_i$$
$$= \sum_{i=1}^{n} (\text{depth}(k_i) + 1)p_i$$
$$= \sum_{i=1}^{n} \text{depth}(k_i)p_i + \sum_{i=1}^{n} p_i$$
$$= \left(\sum_{i=1}^{n} \text{depth}(k_i)p_i\right) + 1$$

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:

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 $O(nm - m^2)$.

Example 1: The possible binary search trees for the identifier set $(a_1, a_2, a_3) = $ (do, if, stop) are as follows. Given the equal probabilities $p(i) = Q(i) = 1/7$ for all i, we have:
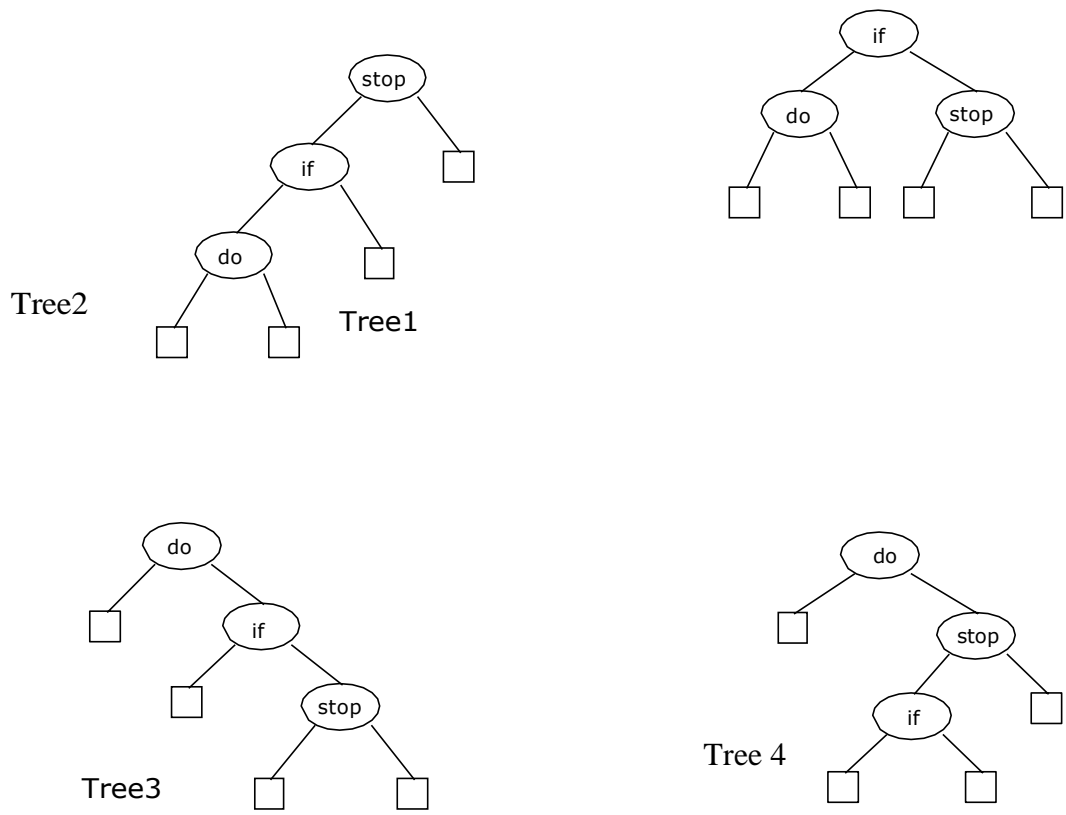
Huffman coding tree solved by a greedy algorithm has a limitation of having the data only at the leaves and it must not preserve the property that all nodes to the left of the root have keys, which are less etc. Construction of an optimal binary search tree is harder, because the data is not constrained to appear only at the leaves, and also because the tree must satisfy the binary search tree property and it must preserve the property that all nodes to the left of the root have keys, which are less.

A dynamic programming solution to the problem of obtaining an optimal binary search tree can be viewed by constructing a tree as a result of sequence of decisions by holding the principle of optimality. A possible approach to this is to make a decision as which of the a_i's be arraigned to the root node at 'T'. If we choose 'a_k' then is clear that the internal nodes for $a_1, a_2, \ldots a_{k-1}$ as well as the external nodes for the classes $E_0, E_1, \ldots \ldots E_{k-1}$ will lie in the left sub tree, L, of the root. The remaining nodes will be in the right subtree, R. The structure of an optimal binary search tree is:

The C (i, J) can be computedas:

$$C (i, J) = \min_{i < k \leq J} \{C (i, k-1) + C (k, J) + P (K) + w (i, K-1) + w (K,J)\}$$

$$= \min_{i < k \leq J} \{C (i, K-1) + C (K, J)\} + w (i,J) \qquad \text{--} \qquad (1)$$

Where $W (i, J) = P (J) + Q (J) + w (i,J-1)$ --   (2)

Initially C (i, i) = 0 and w (i, i) = Q (i) for $0 \leq i \leq n$.

Equation (1) may be solved for C (0, n) by first computing all C (i, J) such that J - i = 1 Next, we can compute all C (i, J) such that J - i = 2, Then all C (i, J) with J - i = 3 and soon.

C (i, J) is the cost of the optimal binary search tree 'Tij' during computation we record the root R (i, J) of each tree 'Tij'. Then an optimal binary search tree may be constructed from these R (i, J). R (i, J) is the value of 'K' that minimizes equation(1).

We solve the problem by knowing W (i, i+1), C (i, i+1) and R (i, i+1), $0 \leq i < 4$; Knowing W (i, i+2), C (i, i+2) and R (i, i+2), $0 \leq i < 3$ and repeating until W (0, n), C (0, n) and R (0, n) areobtained.

The results are tabulated to recover the actualtree.

Example1:

Let n = 4, and (a1, a2, a3, a4) = (do, if, need, 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				

Thiscomputationiscarriedoutrow-wisefromrow0torow4.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 ≤ J. Start with i = 0; so j = 1; as i < k ≤ 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 (value of 'K' that is minimum in the above equation). Next

with i = 1; so j = 2; as i < k ≤ 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 ≤ 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 ≤ 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
R (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 ≤ J. Start with i = 0; so j = 2; as i < k ≤ J, so the possible values for k = 1 and2.

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

122

R (0, 2) =1
Next, with i = 1; so j = 3; as i < k ≤ j, so the possible value for k = 2 and3.

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
R (1, 3) =2

Next, with i = 2; so j = 4; as i < k ≤ 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
R (2, 4) =3

Third, Computing all C (i, j) such that J - i = 3; j = i + 3 and as 0 ≤i < 2; i = 0,1;
 i < k ≤ J. Start with i = 0; so j = 3; as i < k ≤ j, so the possible values for k = 1, 2 and3.

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
R (0, 3) =2

Start with i = 1; so j = 4; as i < k ≤ j, so the possible values for k = 2, 3 and4. W (1, 4)

= P (4) + Q (4) + W (1, 3) = 1 + 1 + 9 =11
C (1, 4) = W (1, 4) + min {[C (1, 1) + C (2, 4)], [C (1, 2) + C (3,4)],
          [C (1, 3) + C (4,4)]}
     = 11 + min {(0 + 8), (7 + 3), (12 + 0)} = 11 + 8 =19
R (1, 4) =2

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

Start with i = 0; so j = 4; as i < k ≤ j, so the possible values for k = 1, 2, 3 and4.

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
R (0, 4) =2

From the table we see that C (0, 4) = 32 is the minimum cost of a binary search tree for
(a1, a2, a3, a4). The root of the tree 'T04' is'a2'.

123

Hence the left sub tree is 'T01' and right sub tree is T24. The root of 'T01' is 'a1' and the root of 'T24' isa3.

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 isa4.

Example2:

Consider four elements a1, a2, a3 and a4 with Q0 = 1/8, Q1 = 3/16, Q2 = Q3 = Q4 = 1/16 and p1 = 1/4, p2 = 1/8, p3 = p4 =1/16. Construct an optimal binary search tree. 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 ≤ j, so the possible value for k =1

W (0, 1) = P (1) + Q (1) + W (0, 0) = 4 + 3 + 2 =9
C (0, 1) = W (0, 1) + min {C (0, 0) + C (1, 1)} = 9 + [(0 + 0)] =9
R (0, 1) = 1 (value of 'K' that is minimum in the above equation). Next

with i = 1; so j = 2; as i < k ≤ j, so the possible value for k =2

W (1, 2) = P (2) + Q (2) + W (1, 1) = 2 + 1 + 3 =6
C (1, 2) = W (1, 2) + min {C (1, 1) + C (2, 2)} = 6 + [(0 + 0)] =6
R (1, 2) =2

Next with i = 2; so j = 3; as i < k ≤ 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 ≤ 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
R (3, 4) =4

Second, Computing all C (i, j) such that j - i = 2; j = i + 2 and as 0 ≤i < 3; i = 0, 1, 2; i < k ≤J

Start with i = 0; so j = 2; as i < k ≤ j, so the possible values for k = 1 and 2. W (0,

2) = P (2) + Q (2) + W (0, 1) = 2 + 1 + 9 =12
C (0, 2) = W (0, 2) + min {(C (0, 0) + C (1, 2)), (C (0, 1) + C (2,2))}
          = 12 + min {(0 + 6, 9 + 0)} = 12 + 6 =18
R (0, 2) =1
Next, with i = 1; so j = 3; as i < k ≤ j, so the possible value for k = 2 and3.

W (1, 3) = P (3) + Q (3) + W (1, 2) = 1 + 1+ 6 =8
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), (6 + 0)} = 8 + 3 =11
R (1, 3) =2

Next, with i = 2; so j = 4; as i < k ≤ 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
R (2, 4) =3

Third, Computing all C (i, j) such that J - i = 3; j = i + 3 and as 0 ≤i < 2; i = 0,1;
 i < k ≤ J. Start with i = 0; so j = 3; as i < k ≤ j, so the possible values for k = 1, 2 and3.

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 + 11), (9 + 3), (18 + 0)} = 14 + 11 =25
R (0, 3) =1

Start with i = 1; so j = 4; as i < k ≤ j, so the possible values for k = 2, 3 and4. W (1, 4)

= P (4) + Q (4) + W (1, 3) = 1 + 1 + 8 =10
C (1, 4) = W (1, 4) + min {[C (1, 1) + C (2, 4)], [C (1, 2) + C (3,4)],
                          [C (1, 3) + C (4,4)]}
          = 10 + min {(0 + 8), (6 + 3), (11 + 0)} = 10 + 8 =18
R (1, 4) =2

Fourth, Computing all C (i, j) such that J - i = 4; j = i + 4 and as 0 ≤i < 1; i =0;

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

$W(0, 4) = P(4) + Q(4) + W(0, 3) = 1 + 1 + 14 = 16$
$C(0, 4) = W(0, 4) + \min \{[C(0, 0) + C(1, 4)], [C(0, 1) + C(2,4)],$
$\qquad\qquad\qquad\qquad [C(0, 2) + C(3, 4)], [C(0, 3) + C(4,4)]\}$
$\qquad = 16 + \min [0 + 18, 9 + 8, 18 + 3, 25 + 0] = 16 + 17 = 33$
$R(0, 4) = 2$

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

Column Row	0	1	2	3	4
0	2, 0,0	1, 0,0	1, 0,0	1, 0,0,	1, 0,0
1	9, 9,1	6, 6,2	3, 3,3	3, 3,4	
2	12, 18,1	8, 11,2	5, 8,3		
3	14, 25,2	11, 18,2			
4	16, 33,2				

From the table we see that $C(0, 4) = 33$ is the minimum cost of a binary search tree for $(a1, a2, a3, a4)$

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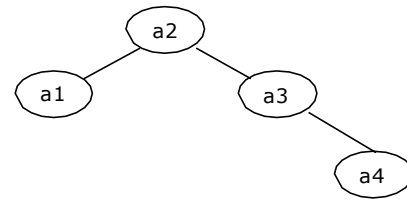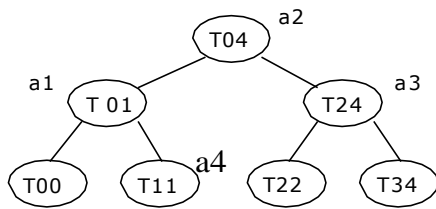
The left and right sub trees for 'T01' are 'T00' and 'T11' respectively. The root of T01 is 'a1'

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, ... , x_n. A decision on variable x_i involves determiningwhichofthevalues0or1istobeassignedtoit.Letusassume that
decisions on the x_i are made in the order x_n, x_{n-1}, ...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}, . . . , x_1 must be optimal with respect to the problem state resulting from the decision on x_n. Otherwise, x_n,. . . . , x_1 will not be optimal. Hence, the principal of optimalityholds.

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

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

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

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

When the w_i's are integer, we need to compute $f_i (y)$ for integer y, $0 \leq y \leq m$. Sincef_i
$(y) = - \square$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 $\Theta (m\ n)$ time to compute f_n. When the w_i's are real numbers, $f_i (y)$ is needed for real numbers y such that $0 \leq y \leq m$. So, f_i cannot be explicitly computed for all y in this range. Even when the w_i's are integer, the explicit $\Theta (m\ n)$ computation of f_n may not be the most efficient computation. So, we explore **an alternative method for bothcases.**

The $f_i (y)$ is an ascending step function; i.e., there are a finite number of y's, $0 = y_1 < y_2 < \ldots < y_k$, such that $f_i (y_1) < f_i (y_2) < \ldots < f_i (y_k)$; $f_i (y) = - ,\overline{y} < y_1$; $f_i (y) = f (y_k)$, $y \geq y_k$; and $f_i (y) = f_i (y_j)$, $y_j \leq y \leq y_{j+1}$. 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

128

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 firstcomputing:

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

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 $W_j \geq W_k$, 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).

Example1:

Consider the knapsack instance n = 3, (w_1, w_2, w_3) = (2, 3, 4), (P_1, P_2, P_3) = (1,2, 5) and M =6.

Solution:

Initially, f_0 (x) = 0, for all x and f_i (x) = - $\square$ if x < 0. F_n

(M) = max {f_{n-1} (M), f_{n-1} (M - w_n) +p_n}

F_3 (6) = max (f_2 (6), f_2 (6 – 4) + 5} = max {f_2 (6), f_2 (2) +5}

F_2 (6) = max (f_1 (6), f_1 (6 – 3) + 2} = max {f_1 (6), f_1 (3) +2}

F_1 (6) = max (f_0 (6), f_0 (6 – 2) + 1} = max {0, 0 + 1} =1

F_1 (3) = max (f_0 (3), f_0 (3 – 2) + 1} = max {0, 0 + 1} =1

Therefore, F_2 (6) = max (1, 1 + 2} =3

F_2 (2) = max (f_1 (2), f_1 (2 – 3) + 2} = max {f_1 (2), - 0+ 2}

F_1 (2) = max (f_0 (2), f_0 (2 – 2) + 1} = max {0, 0 + 1} =1

F_2 (2) = max {1, - 0+ 2} =1

Finally, f3 (6) = max {3, 1 + 5} =6

OtherSolution:

For the given data wehave:

$S^0 = \{(0,0)\};$ $S^0_1 = \{(1,2)\}$

$S^1 = (S^0 \cup S^0_1) = \{(0, 0), (1,2)\}$

X - 2 = 0 => x =2.   y – 3 = 0 => y =3
X - 2 = 1 => x =3.   y – 3 = 2 => y =5

$S^1_1 = \{(2, 3), (3,5)\}$

$S^2 = (S^1 \cup S^1_1) = \{(0, 0), (1, 2), (2, 3), (3,5)\}$

X – 5 = 0 => x =5.   y – 4 = 0 => y =4
X – 5 = 1 => x =6.   y – 4 = 2 => y =6
X – 5 = 2 => x =7.   y – 4 = 3 => y =7
X – 5 = 3 => x =8.   y – 4 = 5 => y =9

$S^2_1 = \{(5, 4), (6, 6), (7, 7), (8,9)\}$

$S^3 = (S^2 \cup S^2_1) = \{(0, 0), (1, 2), (2, 3), (3, 5), (5, 4), (6, 6), (7, 7), (8,9)\}$

By applying Dominancerule,

$S^3 = (S^2 \cup S^2_1) = \{(0, 0), (1, 2), (2, 3), (5, 4), (6,6)\}$

From (6, 6) we can infer that the maximum Profit $\sqcap p_i\ x_i = 6$ and weight $\sqcap x_i\ w_i = 6$

ReliabilityDesign

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, i $\leq$ i $\leq$ 10, then r_i = .904. Hence, it is desirable to duplicate devices. Multiply copies of the same device type are connected inparallel.

Ifstage I contains miscopies ofdeviceD_i.Thentheprobabilitythatallm_i havea malfunction is $(1 - r_i)^{m_i}$. Hence the reliability of stage i becomes $1 – (1 - r)^{m_i}_i$

The reliability of stage 'i' is given by a function $_i(m_i)$.

Our problem is to use device duplication. This maximization is to be carried out under a cost constraint. Let c_i be the cost of each unit of device i and let c be the maximum allowable cost of the system beingdesigned.

Clearly, $f_0(x) = 1$ for all x, $0 \leq x \leq C$ and $f(x) = -\infty$ for all $x < 0$.

Let S^i consist of tuples of the form (f, x), where $f = f_i(x)$.

There is at most one tuple for each different 'x', that result from a sequence of decisions on $m_1, m_2, \ldots m_n$. The dominance rule (f_1, x_1) dominate (f_2, x_2) if $f_1 \geq f_2$ and $x_1 \leq x_2$. Hence, dominated tuples can be discarded fromS^1.

DominanceRule:

If S^i contains two pairs (f_1, x_1) and (f_2, x_2) with the property that $f_1 \geq f_2$ and $x_1 \leq x_2$, then (f_1, x_1) dominates (f_2, x_2), hence by dominance rule (f_2, x_2) can be discarded. Discarding or pruning rules such as the one above is known as dominance rule. Dominating tuples will be present in S^i and Dominated tuples has to be discarded fromS^i.

Case 1: if $f_1 \leq f_2$ and $x_1 > x_2$ then discard (f_1, x_1)

Case 2: if $f_1 \geq f_2$ and $x_1 < x_2$ the discard (f_2, x_2)

Case 3: otherwise simply write (f_1, x_1)

$S2 = \{(0.72, 45), (0.864, 60), (0.8928, 75)\}$

S_3^3 $\square\square(0.63, 105), \square 1.756, 120\square, \square 0.7812, 135\square\square$

If cost exceeds 105, remove thattuples

S^3 = {(0.36, 65), (0.437, 80), (0.54, 85), (0.648, 100)}

The best design has a reliability of 0.648 and a cost of 100. Tracing back forthe solution through S^1 's we can determine that $m_3 = 2$, $m_2 = 2$ and $m_1 = 1$.

OtherSolution:

According to the principle ofoptimality:

$f_n(C) = \max_{1 \le m_n \le u_n} \{ o_n (m_n) . f_{n-1} (C - C_n m_n)$ with $f_0 (x) = 1$ and $0 \le x \le C$;

Since,wecanassumeachc$_i$ >0,eachmimustbeintherange1≤m$_i$ ≤ u$_i$.

BACKTRACKING AND BRANCH AND BOUND

GeneralMethod:

Backtracking is used to solve problem in which a sequence of objects is chosen from a specified set so that the sequence satisfies some criterion. The desired solution is expressed as an n-tuple $(x_1, \ldots, x_n)$ where each $x_i \in S$, S being a finite set.

The solution is based on finding one or more vectors that maximize, minimize, or satisfy a criterion function $P(x_1, \ldots, x_n)$. Form a solution and check at every step if this has any chance of success. If the solution at any point seems not promising, ignore it. All solutions requires a set of constraints divided into two categories: explicit and implicit constraints.

Explicit constraints are rules that restrict each x_i to take on values only from a given set Explicit constraints depend on the particular instance I of problem being solved. All tuples that satisfy the explicit constraints define a possible solution space for I.

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's must relate to eachother.

For 8-queensproblem:

Explicit constraints using 8-tuple formation, for this problem are S= {1, 2, 3, 4, 5, 6, 7,8}. The implicit constraints for this problem are that no two queens can be the same (i.e., all queens must be on different columns) and no two queens can be on the same diagonal.

Backtracking is a modified depth first search of a tree. Backtracking algorithms determine problem solutions by systematically searching the solution space for the given problem instance. This search is facilitated by using a tree organization for the solution space.

Backtracking is the procedure where by, after determining that a node can lead to nothing but dead end, we go back (backtrack) to the nodes parent and proceed with the search on the next child.

A backtracking algorithm need not actually create a tree. Rather, it only needs to keep track of the values in the current branch being investigated. This is the way we implement backtracking algorithm. We say that the state space tree exists implicitly in the algorithm because it is not actually constructed.

Terminology:

Problem state is each node in the depth first search tree.

solution states are the problem states 'S' for which the path from the root node to 'S' defines a tuple in the solution space.

Answer states are those solution states for which the path from root node to s defines a tuple that is a member of the set of solutions.

State space is the set of paths from root node to other nodes. *State space* tree is the tree organization of the solution space. The state space trees are called static trees. This terminology follows from the observation that the tree organizations are independent of the problem instance being solved. For some problems it is advantageous to use different tree organizations for different problem instance. In this case the tree organization is

determined dynamically as the solution space is being searched. Tree organizations that are problem instance dependent are called dynamic trees.

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. In other words, an E-node is a node currently being expanded.

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.

Branch and Bound refers to all state space search methods in which all children of an E-node are generated before any other live node can become the E-node.

Depth first node generation with bounding functions is called *backtracking*. State generation methods in which the E-node remains the E-node until it is dead, lead to branch and bound methods.

ueensProblem:

Let us consider, N = 8. Then 8-Queens Problem is to place eight queens on an 8 x 8 chessboard so that no two "attack", that is, no two of them are on the same row, column, ordiagonal.All solutions to the 8-queens problem can be represented as 8-tuples $(x_1, \ldots, x_8)$, where x_i is the column of the i^{th}row where the i^{th}queen isplaced.

The explicit constraints using this formulation are $S_i = \{1, 2, 3, 4, 5, 6, 7, 8\}$, $1 \leq i \leq 8$

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

The implicit constraints for this problem are that no two x_i's can be the same (i.e., all queens must be on different columns) and no two queens can be on the same diagonal.

This realization reduces the size of the solution space from 8^8 tuples to 8!Tuples.

The promising function must check whether two queens are in the same column or diagonal:

Suppose two queens are placed at positions (i, j) and (k, l)Then:

- Column Conflicts: Two queens conflict if their x_i values are identical.
- Diagonal conflict: Two queens i and j are on the same diagonal

i – j = k –l.

      This implies, j – l = i –k

- Diagonal conflict:

$$i + j = k + l.$$

      This implies, j – l = k –i

Therefore, two queens lie on the same diagonal if and only if:

      |j – l|= |i – k|

Where, j be the column of object in row i for the i^{th}queen and l be the column of object in row 'k' for the k^{th}queen.

To check the diagonal clashes, let us take the following tile configuration:

In this example, we have:

i	1	2	3	4	5	6	7	8
x_i	2	5	1	8	4	7	3	6

Let us consider for the case whether the queens on 3rd row and 8th row are conflicting or not. In this case $(i, j) = (3, 1)$ and $(k, l) = (8, 6)$

Therefore:

$|j - l| = |i - k|$ is $|1 - 6| = |3 - 8|$ which is $5 = 5$

In the above example we have, $|j - l| = |i - k|$, so the two queens are attacking. This is not a solution.

Example:

Suppose we start with the feasible sequence 7, 5, 3,1.

Step1:

Add to the sequence the next number in the sequence 1, 2, . . . , 8 not yet used.

Step2:

If this new sequence is feasible and has length 8 then STOP with a solution. If the new sequence is feasible and has length less then 8, repeat Step1.

Step3:

If the sequence is not feasible, then *backtrack* through the sequence until we find the *most recent* place at which we can exchange a value. Go back to Step 1.

On a chessboard, the **solution**

4 – QueensProblem:

Let us see how backtracking works on the 4-queens problem. We start with the root node as the only live node. This becomes the E-node. We generate one child. Let us assume that the children are generated in ascending order. Let us assume that the children are generated in ascending order. Thus node number 2 of figure is generated and the path is now (1). This corresponds to placing queen 1 on column 1. Node 2 becomes the E-node. Node 3 is generated and immediately killed. The next node generated is node 8 and the path becomes (1, 3). Node 8 becomes the E-node. However, it gets killed as all its children represent board configurations that cannot lead to an answer node. We backtrack to node 2 and generate another child, node 13. The path is now (1, 4). The board configurations as backtracking proceeds is as follows:

(a) (b) (c) (d)

(e) (f) (g) (h)
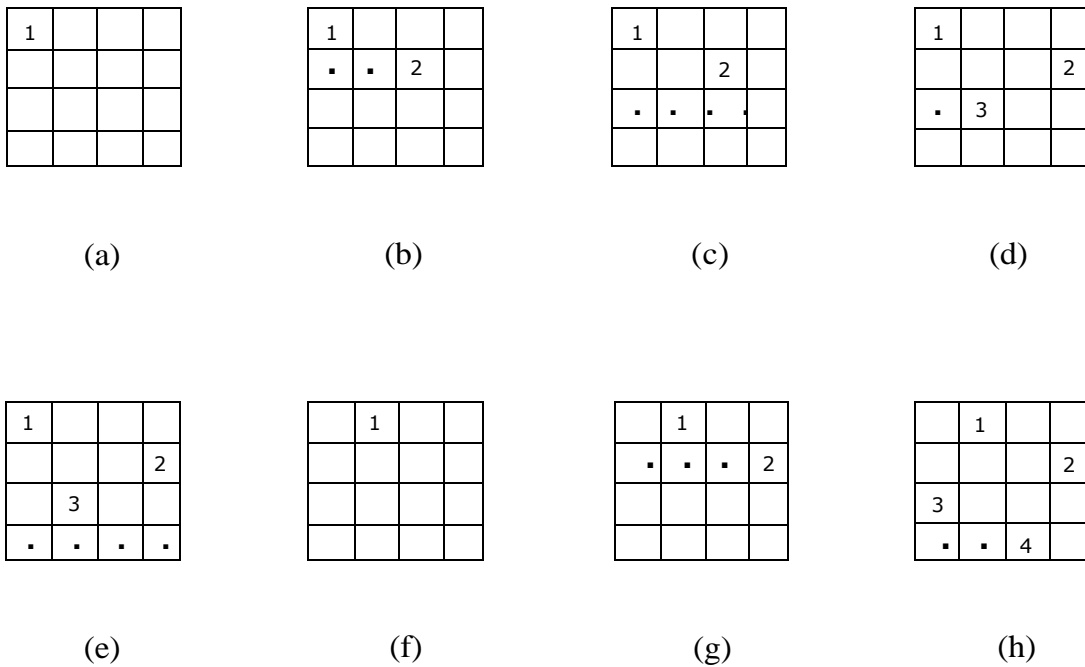
The above figure shows graphically the steps that the backtracking algorithm goes through as it tries to find a solution. The dots indicate placements of a queen, which were tried and rejected because another queen was attacking.

In Figure (b) the second queen is placed on columns 1 and 2 and finally settles on column 3. In figure (c) the algorithm tries all four columns and is unable to place the next queen on a square. Backtracking now takes place. In figure (d) the second queen is moved to the next possible column, column 4 and the third queen is placed on column 2. The boards in Figure (e), (f), (g), and (h) show the remaining steps that the algorithm goes through until a solution is found.
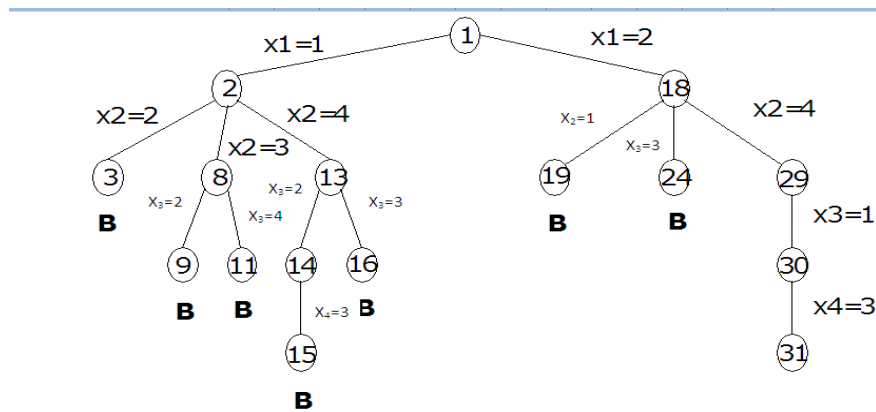
Portion of the tree generated during backtracking

= 19, 173, 961nodes

= 19, 173, 961nodes

Sum of Subsets:

Given positive numbers wi, $1 \le i \le n$, and m, this problem requires finding all subsets of wi whose sums are 'm'. All solutions are k-tuples, $1 \le k \le n$. Explicit constraints: $x_i \in \{j \mid j \text{ is an integer and } 1 \le j \le n\}$. Implicit constraints: No two x_i can be the same. The sum of the corresponding wi's be m. $x_i < x_{i+1}$, $1 \le i < k$ (total order in indices) to avoid generating multiple instances of the same subset (for example, (1, 2, 4) and (1, 4, 2) represent the same subset).

A better formulation of the problem is where the solution subset is represented by a n-tuple $(x_1, \ldots, x_n)$ such that $x_i \in \{0,1\}$.
The above solutions are then represented by (1, 1, 0, 1) and (0, 0, 1,1). For both

the above formulations, the solution space is 2^n distinct tuples.

For example, n = 4, w = (11, 13, 24, 7) and m = 31, the desired subsets are (11, 13, 7) and (24,7). The following figure shows a possible tree organization for two possible formulations of the solution space for the case n =4.
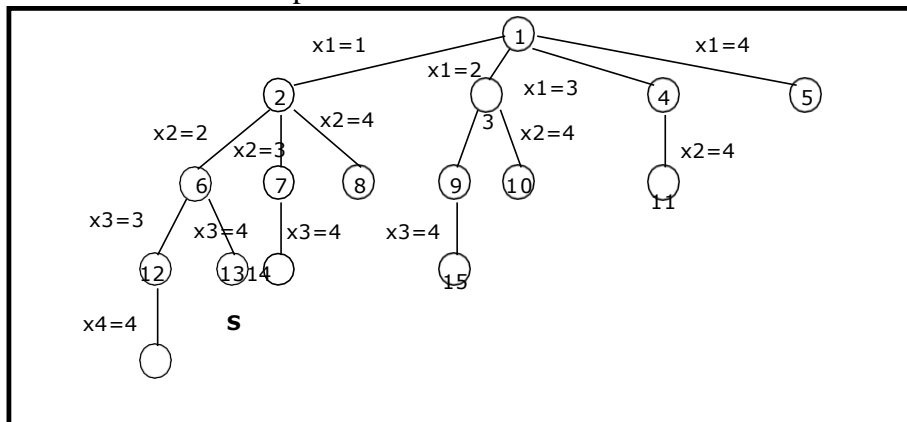
A possible solution space organisation for the

Sum of subsets problem.

The tree corresponds to the variable tuple size formulation. The edges are labeled such that an edge from a level i node to a level i+1 node represents a value for x_i. At each node, the solution space is partitioned into sub - solution spaces. All paths from the root node to any node in the tree define the solution space, since any such path corresponds to a subset satisfying the explicit constraints.
The possible paths are (1), (1, 2), (1, 2, 3), (1, 2, 3, 4), (1, 2, 4), (1, 3, 4), (2), (2, 3), and so on. Thus, the left mot sub-tree defines all subsets containing w1, the next sub-tree defines all subsets containing w2 but not w1, and soon.

Graph Coloring (for planar graphs):

Let G be a graph and m be a given positive integer. We want to discover whether the nodes of G can be colored in such a way that no two adjacent nodes have the same color, yet only m colors are used. This is termed the m-colorabiltiy decision problem. The m-colorability optimization problem asks for the smallest integer m for which the graph G can becolored.

Given any map, if the regions are to be colored in such a way that no two adjacent regions have the same color, only four colors are needed.

For many years it was known that five colors were sufficient to color any map, but no map that required more than four colors had ever been found. After several hundred years, this problem was solved by a group of mathematicians with the help of a computer. They showed that in fact four colors are sufficient for planar graphs.

The function m-coloring will begin by first assigning the graph to its adjacency matrix, setting the array x [] to zero. The colors are represented by the integers 1, 2, . . . m and the solutions are given by the n-tuple $(x1, x2, . . ., xn)$, where x_i is the color of node$i$.

A recursive backtracking algorithm for graph coloring is carried out by invoking the statement mcoloring(1);

Algorithm mcoloring(k)

// This algorithm was formed using the recursive backtracking schema. The graph is
// represented by its Boolean adjacency matrix G [1: n, 1: n]. All assignments of
// 1, 2, ... , m to the vertices of the graph such that adjacent vertices are assigned
// distinct integers are printed. k is the index of the next vertex to color.
{
     repeat

    { // Generate all legal assignments for x[k].

        NextValue(k);   // Assign to x [k] a legal color.

        If (x [k] = 0) then return;   // No new color possible

        If (k = n)then   // at most m colors have been

                // used to color the n vertices.

          write (x [1:n]);

          else mcoloring(k+1);

      } until(false);

    }

Algorithm NextValue(k)

// x [1] , ... x [k-1] have been assigned integer values in the range [1, m] such that

// adjacent vertices have distinct integers. A value for x [k] is determined in the range

//[0,m].x[k]Is assigned the next highest numbered color while maintaining distinctness

// from the adjacent vertices of vertex k. If no such color exists, then x [k] is0.

{

    repeat

    {

x [k]: = (x [k] +1) mod(m+1)   // Next highest color.

If (x [k] = 0) thenreturn;   // All colors have been used for

j := 1 to ndo

{   // check if this color is distinct from adjacent colors

 if ((G [k, j] !=0) and (x [k] = x[j]))

    // If (k, j) is and edge and if adj. vertices have the same color then break;

      }

      if (j = n+1) thenreturn;   // New color found

   }until(false);   // Otherwise try to find another color.

}

Example:

Color the graph given below with minimum number of colors by backtracking using state space tree.

A 4-node graph and all possible 3-colorin gs
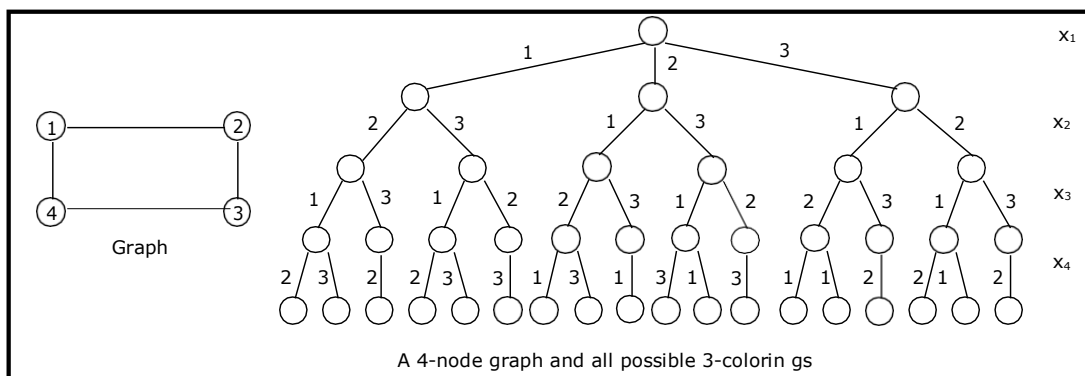
Hamiltonian cycles

Let $G = (V, E)$ be a connected graph with n vertices. A Hamiltonian cycle (suggested by William Hamilton) is a round-trip path along n edges of G that visits every vertex once and returns to its starting position. In other vertices of G are visited in the order $v_1, v_2, \ldots, v_{n+1}$, then the edges (v_i, v_{i+1}) are in E, $1 \leq i \leq n$, and the v_i are distinct expect for v_1 and v_{n+1}, which are equal. The graph G_1 contains the Hamiltonian cycle 1, 2, 8, 7, 6, 5, 4, 3, 1. The graph G_2 contains no Hamiltonian cycle.
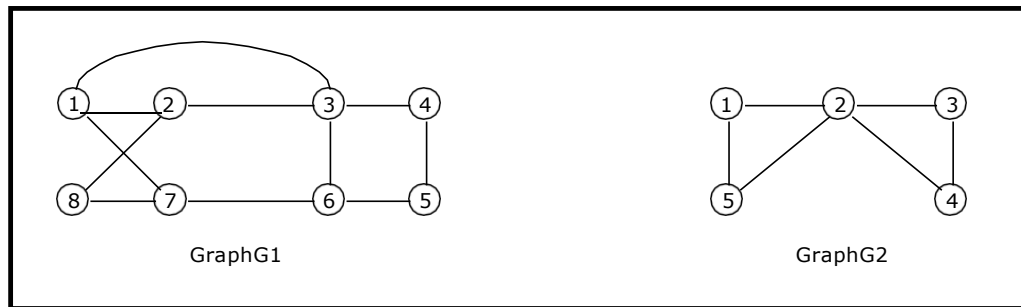
Two graphs to illustrate Hamiltoniancycle

The backtracking solution vector $(x_1, \ldots, x_n)$ is defined so that x_i represents the ith visited vertex of the proposed cycle. If $k = 1$, then x_1 can be any of the n vertices. To avoid printing the same cycle n times, we require that $x_1 = 1$. If $1 < k < n$, then x_k can be any vertex v that is distinct from $x_1, x_2, \ldots, x_{k-1}$ and v is connected by an edge to k_{x-1}. The vertex x_n can only be one remaining vertex and it must be connected to both x_{n-1} and x_1.

Using NextValue algorithm we can particularize the recursive backtracking schema to find all Hamiltonian cycles. This algorithm is started by first initializing the adjacency matrix $G[1: n, 1: n]$, then setting $x[2: n]$ to zero and $x[1]$ to 1, and then executing Hamiltonian(2).

The traveling salesperson problem using dynamic programming asked for a tour that has minimum cost. This tour is a Hamiltonian cycles. For the simple case of a graph all of whose edge costs are identical, Hamiltonian will find a minimum-cost tour if a tour exists.

Algorithm NextValue(k)

// x [1: k-1] is a path of k – 1 distinct vertices . If x[k] = 0, then no vertex has been

// assigned to x [k]. After execution, x[k] is assigned to the next highest numberedvertex

// which does not already appear in x [1 : k – 1] and is connected by an edge to x [k –1].

// Otherwise x [k] = 0. If k = n, then in addition x [k] is connected to x[1].

{

      repeat

```
            {

         x [k] := (x [k] +1) mod(n+1); //Nextvertex.

       If (x [k] = 0) then return;

     If (G [x [k – 1], x [k]] !=0)then

   {                                      // Is there an edge?

        for j := 1 to k – 1 do if (x [j] = x [k]) then break;

                                          // check for distinctness.

        If (j = k) then    // If true, then the vertex is distinct.

        If ((k < n) or ((k = n) and G [x [n], x [1]]  ☐0))

        Then return;

                }

        } until(false);

}


```

Algorithm Hamiltonian(k)

// This algorithm uses the recursive formulation of backtracking to find all theHamiltonian

// cycles of a graph. The graph is stored as an adjacency matrix G [1: n, 1: n]. All cycles begin

// at node1.

```
{

      repeat

      {                                      // Generate values for x[k].

            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);

}
```

BRANCH AND BOUND
Generalmethod:

Branch and Bound is another method to systematically search a solution space. Just like backtracking, we will use bounding functions to avoid generating subtrees that do not contain an answer node. However branch and Bound differs from backtracking in two ways:

1. It has a branching function, which can be a depth first search, breadth first search or based on bounding function.

2. It has a bounding function, which goes far beyond the feasibility test as a mean to prune efficiently the search tree.

Branch and Bound refers to all state space search methods in which all children of the E-node are generated before any other live node becomes the E-node

Branch and Bound is the generalization of both graph search strategies, BFS and D-search.

*   A BFS like state space search is called as FIFO (First in first out) search as the list of live nodes in a first in first out list (or queue).

*   A D search like state space search is called as LIFO (Last in first out) search as the list of live nodes in a last in first out (or stack).

Definition 1: Live node is a node that has been generated but whose children have not yet been generated.

Definition 2: E-node is a live node whose children are currently being explored. In other words, an E-node is a node currently being expanded.

Definition 3: 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.

Definition 4: Branch-an-bound refers to all state space search methods in which all children of an E-node are generated before any other live node can become the E-node.

Least Cost (LC)search:

In both LIFO and FIFO Branch and Bound the selection rule for the next E-node in rigid and blind. The selection rule for the next E-node 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 be obtained by using an "intelligent" ranking function $C(.)$ for live nodes. The next E-node is selected on the basis of this ranking function. The node x is assigned a rank using:
c(x)=f(h(x))+g(x)
where,   c(x) is the cost of x.

  h(x) is the cost of reaching x from the root and f(.) is any non-decreasing function.

   is g(x) is an estimate of the additional effort needed to reach an answer node from x.

 A search strategy that uses a cost function
 c(x)=f(h(x))+g(x) to select next
E-node would always choose for its next E-node a live node with least c(.) is   is is
known as LC–search (Least Cost search)

$$g$$
$$= 0 \text{ and } f(h(x)) = \text{levelof}$$

BFS and D-search are special cases of LC-search.If
node x, then an LC search generates nodes by levels. This is eventually the sameas
a BFS. If f(h(x)) = 0 and $\hat{g}(x) >$ $\hat{g}(y)$wheneveryisachildofx,thenthesearchis
essentially aD-search.

An LC-search coupled with bounding functions is called an LC-branch and bound search

We associate a cost c(x) with each node x in the state space tree. It is not possibleto
easily compute the function c(x). So we compute aestimate $\hat{c}(x)$ofc(x).

Control Abstraction forLC-Search:
Let t be a state space tree and c() a cost function for the nodes in t. If x is a node in t, then
c(x) is the minimum cost of any answer node in the subtree with root x. Thus, c(t) is the
cost of a minimum-cost answer node int.

   $\hat{c}(.)$isusedtoestimatec().Thisheuristicshouldbeeasytocomputeand
Aheuristic
generally has the property that if x is either an answer node or a leaf node, then
c(x)= $\hat{c}(x)$.

LC-search uses $\hat{c}$ to find an answer node. The algorithm usestwofunctionsLeast()and Add() to delete and add a live node from or to the list of live nodes, respectively.

Least() finds a live node with least c(). This node is deleted from the list of live nodes andn returned.

Add(x) adds the new live node x to the list of live nodes. The list of live nodes be implemented as a min-heap.

Algorithm LCSearch outputs the path from the answer node it finds to the root node t. This is easy to do if with each node x that becomes live, we associate a field *parent* which gives the parent of node x. When the answer node g is found, the path from g to t can be determined by following a sequence of *parent* values starting from the current E-node (which is the parent of g) and ending at node t.

Listnode =**record**

{

       Listnode * next, *parent; float cost;

}

Algorithm**LCSearch**(t)

{   //Search t for an answernode

       if *t is an answer node then output *t and
       return; E:=t;   //E-node.

       initialize the list of live nodes to be empty;
       repeat

       {

              for each child x of Edo

              {

                     if x is an answer node then output the path from x to t and
                     return;

                    Add (x);   //x is a new livenode.

                    (x □parent):=E;   // pointer for path to root

              }

              if there are no more live nodes then

```
            {
                     write    ("No    answer
                     node"); return;
            }
            E :=Least();
        } until(false);

}
```

The root node is the first, E-node. During the execution of LC search, this list contains all live nodes except the E-node. Initially this list should be empty. Examine all the children of the E-node, if one of the children is an answer node, then the algorithm outputs the path from x to t and terminates. If the child of E is not an answer node, then it becomes a live node. It is added to the list of live nodes and its parent field set to E. When all the children of E have been generated, E becomes a dead node. This happens only if none of E's children is an answer node. Continue the search further until no live nodes found. Otherwise, Least(), by definition, correctly chooses the next E-node and the search continues from here.

LC search terminates only when either an answer node is found or the entire state space tree has been generated and searched.

Bounding:

A branch and bound method searches a state space tree using any search mechanism in which all the children of the E-node are generated before another node becomes the E-node. We assume that each answer node x has a cost c(x) associated with it and that a minimum-cost answer node is to be found. Three common search strategies are FIFO, LIFO, and LC. The three search methods differ only in the selection rule used to obtain the nextE-node.

good bounding helps to prune efficiently the tree, leading to a faster exploration of the solutionspace.

A costfunction$c(.)$suchthat$c(x) \leq c(x)$ is used to provide lower bounds on

solutionsobtainablefromanynodex.Ifupperisanupperboundonthecostofa

minimum-cost solution, then all live nodes x with $c(x) \geq c(x) >$ upper. The starting value for upper can be obtained by some heuristic or can be set .

As long as the initial value for upper is not less than the cost of a minimum-cost answer node, the above rules to kill live nodes will not result in the killing of a live node that can reach a minimum-cost answer node. Each time a new answer node is found, the value of upper can beupdated.

Branch-and-bound algorithms are used for optimization problems where, we deal directly only with minimization problems. A maximization problem is easily converted to a minimization problem by changing the sign of the objective function.

To formulate the search for an optimal solution for a least-cost answer node in a state space tree, it is necessary to define the cost function c(.), such that c(x) is minimum for all

nodes representing an optimal solution. The easiest way to do this is to use the objective function itself forc(.).

- For nodes representing feasible solutions, c(x) is the value of the objective function for that feasible solution.

- For nodes representing infeasible solutions, c(x) =0.

- For nodes representing partial solutions, c(x) is the cost of the minimum-cost node in the subtree with root x.

Since, c(x) is generally hard to compute, the branch-and-bound algorithm will usean

Estimate (x) suchthat $(x) \leq c(x)$ forallx.

Sum-of-Subsets problem

➢ In this problem, we are given a vector of N values, called weights. The weights are usually given in ascending order of magnitude and are unique.
➢ For example, W= (2, 4, 6, 8, 10) is a weight vector. We are also given a value M, for example 20.
➢ The problem is to find all combinations of the weights that exactly add to M.
➢ In this example, the weights that add to 20 are:
(2, 4, 6, 8); (2, 8, 10); and (4, 6, 10).
➢ Solutions to this problem are often expressed by an N-bit binary solution vector, X, where a 1 in position i indicates that Wiis part of the solution and a 0 indicates it is not.
➢ In this manner the three solutions above could be expressed as: (1,1,1,1,0); (1,0,0,1,1); (0,1,1,0,1)

Sum-of-Subsets problem

➢ We are given 'n' positive numbers called weights and we have to find all combinations of these numbers whose sum is M. this is called sum of subsets problem.
➢ If we consider backtracking procedure using fixed tuple strategy , the elements X(i) of the solution vector is either 1 or 0 depending on if the weight W(i) is included or not.
➢ If the state space tree of the solution, for a node at level I, the left child corresponds to X(i)=1 and right to X(i)=0.

Sum of Subsets Algorithm

```
void SumOfSub(float s, int k, float r)
{
// Generate left child.
x[k] = 1;
if (s+w[k] == m)
{       for (int j=1; j<=k; j++)
```
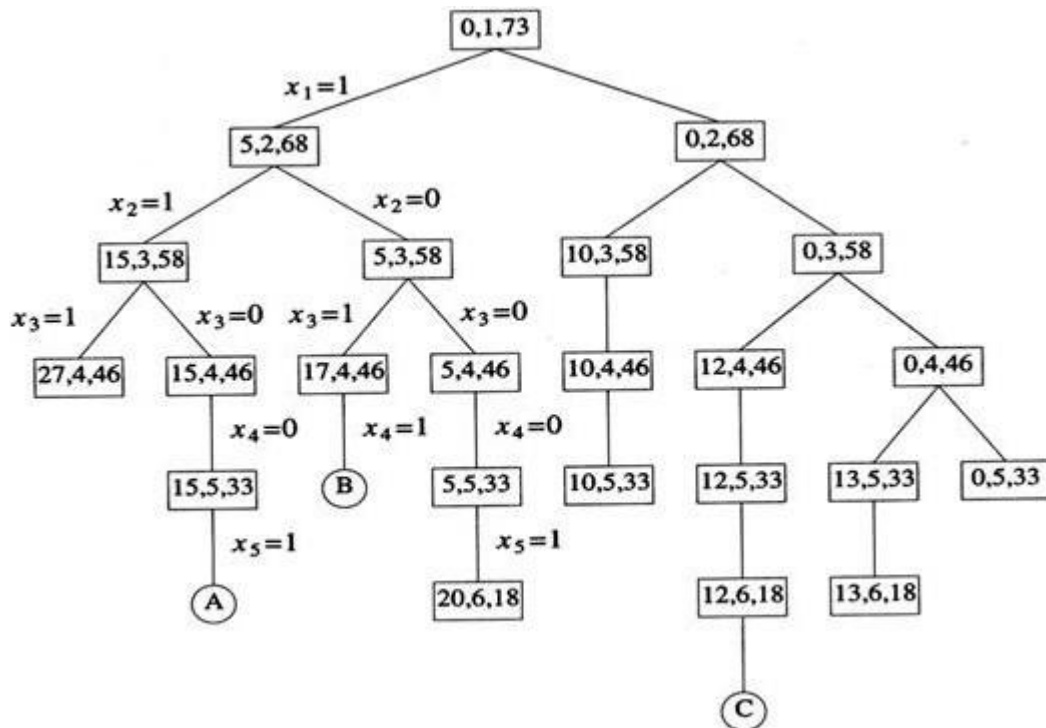
148

```
Print (x[j] )
}
else if (s+w[k]+w[k+1] <= m)
SumOfSub(s+w[k], k+1, r-w[k]);
// Generate right child and evaluate
if ((s+r-w[k] >= m) && (s+w[k+1] <= m)) { x[k] = 0;
SumOfSub(s, k+1, r-w[k]);
}
}
```

Sum of Subsets State Space Tree

Example n=6, w[1:6]={5,10,12,13,15,18},m=30

Branch and Bound Principal

> ➢ The term branch-and-bound refers to all state space search methods in which all children of the £-node are generated before any other live node can become the £-node.
> ➢ We have already seen two graph search strategies, BFS and D-search, in which the exploration of a new node cannot begin until the node currently being explored is fully explored.
> ➢ Both of these generalize to branch-and-bound strategies.

149

- In branch-and-bound terminology, a BFS-like state space search will be called FIFO (First In First Out) search as the list of live nodes is a first-in-first-out list (or queue).
- A D-search 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).

Control Abstraction for Branchand Bound(LC Method)

```
line   procedure LC (T, ĉ)
       //search T for an answer node//
0      if T is an answer node then output T; return; endif
1      E ← T   //E-node//
2      initialize the list of live nodes to be empty
3        loop
4          for each child X of E do
5             if X is an answer node then output the path from X to T
6                                    return
7             endif
8             call ADD(X)   //X is a new live node//
9             PARENT(X) ← E   //pointer for path to root//
10         repeat
11         if there are no more live nodes then print ('no answer node')
12                                    stop
13         endif
14         call LEAST(E)
15       repeat
16     end LC
```

LC Method Control AbstractionExplanation

- The search for an answer node can often be speeded by using an "intelligent" ranking function, c(.), for livenodes.
- The next £-node is selected on the basis of this rankingfunction.
- Let T be a state space tree and c( ) a cost function for the nodes in T.If X is a node in T then c(X) is the minimum cost of any answer node in the subtree with root X. Thus, c(T) is the cost of a minimum cost answer node
- The algorithm uses two subalgorithms LEAST(X) and ADD(X) to respectively delete and add a live node from or to the list of livenodes.

- LEAST{X) finds a live node with least c( ). This node is deleted from the list of live nodes and returned in variableX.

- ADD(X) adds the new live node X to the list of livenodes.

- Procedure LC outputs the path from theanswer

150

The 0/1 knapsack problem

- Positive integer $P_1, P_2, \ldots, P_n$ (profit)

  $\qquad\qquad W_1, W_2, \ldots, W_n$ (weight)

  $\qquad\qquad$ M (capacity)

maximize $\displaystyle\sum_{i=1}^{n} P_i X_i$

subject to $\displaystyle\sum_{i=1}^{n} W_i X_i \le M \quad X_i = 0 \text{ or } 1, i = 1, \ldots, n.$

The problem is modified:

minimize $\displaystyle -\sum_{i=1}^{n} P_i X_i$

The 0/1 knapsack problem
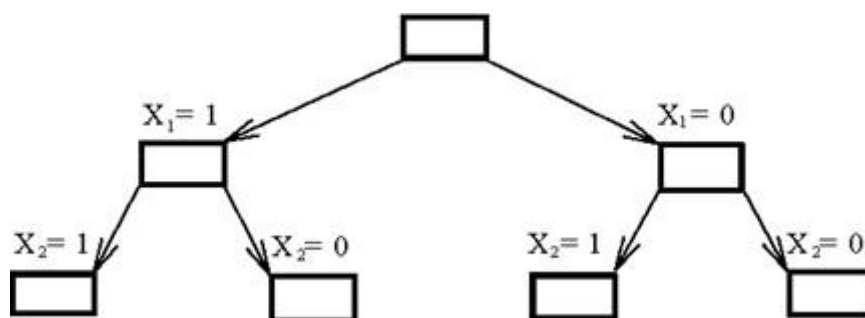
The Branching Mechanism in the Branch-and-Bound Strategy toSolve 0/1KnapsackProblem.

How to find the upper bound?

Ans: by quickly finding a feasible solution in a greedy manner: starting from the smallest available i, scanning towards the largest i's until M is exceeded. The upper bound can be calculated.

How to find the ranking Function

- Ans: by relaxing our restriction from $X_i = 0$ or 1 to $0 \le X_i \le 1$ (knapsack problem)

Let $-\sum\limits_{i=1}^{n} P_i X_i$ be an optimal solution for $0/1$

knapsack problem and $-\sum\limits_{i=1}^{n} P_i X'_i$ be an optimal

solution for fractional knapsack problem. Let

$$Y = -\sum_{i=1}^{n} P_i X_i, \quad Y' = -\sum_{i=1}^{n} P_i X'_i.$$

$$\Rightarrow Y' \le Y$$

How to expand the tree?

➢ By the best-first searchscheme
➢ That is, by expanding the node with the best lower bound. If two nodes have the same lower bounds, expand the node with the lower upper bound.

0/1 Knapsack algorithm using BB

```
procedure UBOUND (p, w, k, M)
    //p, w, k and M have the same meaning as in Algorithm 7.11//
    //W(i) and P(i) are respectively the weight and profit of the ith object//
    global W(1:n), P(1:n); integer i, k, n
    b ← p; c ← w
    for i ← k + 1 to n do
        if c + W(i) ≤ M then c ← c + W(i); b ← b − P(i) endif
    repeat
    return (b)
end UBOUND
```
            Algorithm 8.5   Function $u(\cdot)$ for knapsack problem

0/1 Knapsack Example usingLCBB (Least Cost)

- ➢ Example (LCBB)
- ➢ Consider the knapsackinstance:
- ➢ n = 4;
- ➢ (pi, p2,p3, p4) = (10, 10, 12, 18);

- ➢ (wi. w2, w3, w4) = (2, 4, 6, 9)and
- ➢ M = 15.

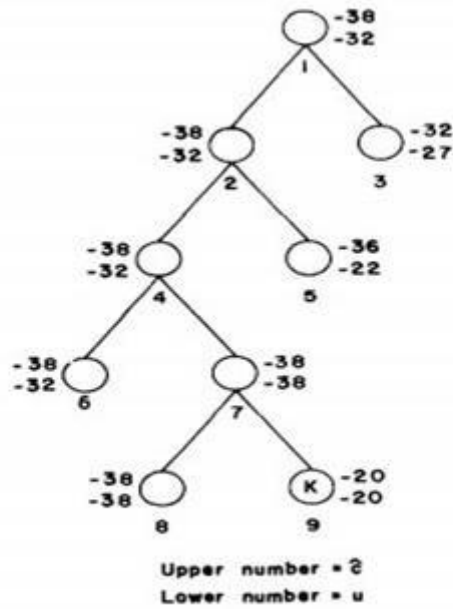**0/1 Knapsack State Space tree ofExample using LCBB**

Figure 8.9   LC Branch-and-bound tree for Example 8.2

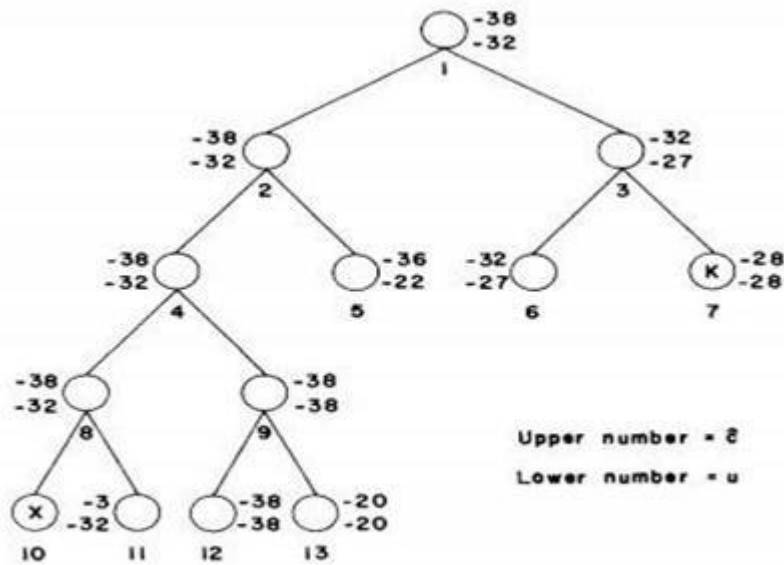**0/1 Knapsack State Space tree ofExample using FIFO BB**

Figure 8.10 FIFO branch-and-bound tree for Example 8.3

The traveling salesperson problem

☐ Given a graph, the TSP Optimization problem is to find a tour, starting from any vertex, visiting every other vertex and returning to the starting vertex, with minimalcost.

The basic idea

☐ There is a way to split the solution space (branch)
There is a way to predict a lower bound for a class of solutions. There is also a way tofind a upper bound of an optimal solution. If the lower bound of a solution exceeds the upper bound, this solution cannot be optimal and thus we should terminate the branching associated with thissolution.

Example- TSP

☐ Example with Cost Matrix(a) and its Reduced Cost Matrix (b)
☐ Reduced matrix means every row and column of matrix should contain at least one Zero and all other entries should be non negative.

$$\begin{bmatrix} \infty & 20 & 30 & 10 & 11 \\ 15 & \infty & 16 & 4 & 2 \\ 3 & 5 & \infty & 2 & 4 \\ 19 & 6 & 18 & \infty & 3 \\ 16 & 4 & 7 & 16 & \infty \end{bmatrix} \qquad \begin{bmatrix} \infty & 10 & 17 & 0 & 1 \\ 12 & \infty & 11 & 2 & 0 \\ 0 & 3 & \infty & 0 & 2 \\ 15 & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & 12 & \infty \end{bmatrix}$$

(a) Cost Matrix   (b) Reduced Cost Matrix

L = 25

Reduced Matrix for node 2,3…10 ofState Space tree using LC Method

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & 2 & 0 \\ 0 & \infty & \infty & 0 & 2 \\ 15 & \infty & 12 & \infty & 0 \\ 11 & \infty & 0 & 12 & \infty \end{bmatrix} \quad \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & \infty & 2 & 0 \\ \infty & 3 & \infty & 0 & 2 \\ 4 & 3 & \infty & \infty & 0 \\ 0 & 0 & \infty & 12 & \infty \end{bmatrix} \quad \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & 0 \\ 0 & 3 & \infty & \infty & 2 \\ \infty & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & \infty & \infty \end{bmatrix}$$

a) path 1,2; node 2   b) path 1,3; node 3   c) path 1,4;node 4

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 10 & \infty & 9 & 0 & \infty \\ 0 & 3 & \infty & 0 & \infty \\ 12 & 0 & 9 & \infty & \infty \\ \infty & 0 & 0 & 12 & \infty \end{bmatrix} \quad \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & \infty & 0 \\ 0 & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & 0 & \infty & \infty \end{bmatrix} \quad \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & \infty & \infty & 0 \\ \infty & 1 & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \\ 0 & 0 & \infty & \infty & \infty \end{bmatrix}$$

d) path 1,5; node 5   e) path 1,4,2; node 6   f) path 1,4,3; node 7

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & 0 & \infty & \infty \\ 0 & 3 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & 0 & \infty & \infty \end{bmatrix} \quad \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty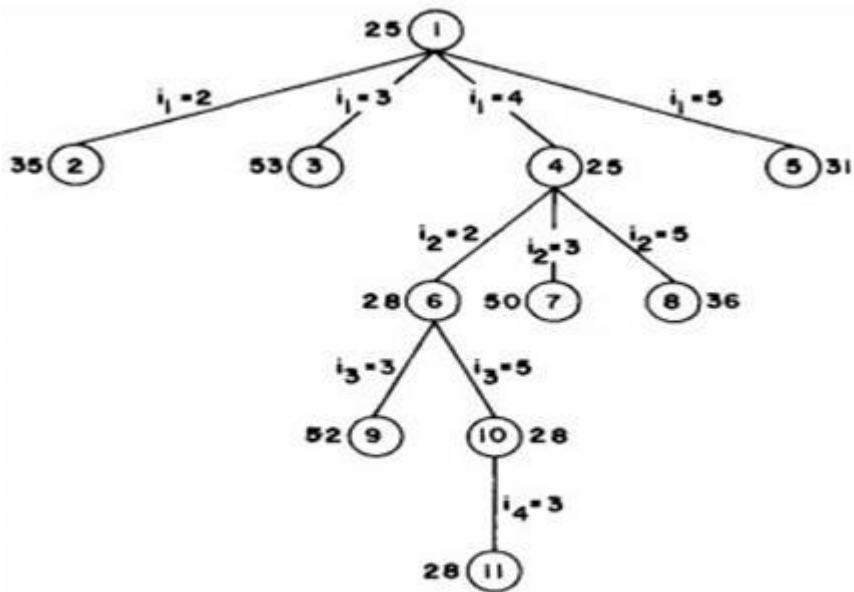 \\ 0 & \infty & \infty & \infty & \infty \end{bmatrix} \quad \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & \infty & \infty \end{bmatrix}$$

g) path 1,4,5; node 8   h) path 1,4,2,3; node 9   i) path 1,4,2,5; node 10

State Space tree of Example using LCMethod

Numbers outside the node are $\hat{c}$ values

State space tree generated by procedure LCBB.

UNIT-V

NP –HARD AND NP-COMPLETE PROBLEMS.

Basic concepts:

NP Nondeterministic Polynomial time.
The problems has best algorithms for their solutions have "Computing times", that cluster into two groups.

Group-1

Problems with solution time bound by a polynomial of a small degree. They are also called "Tractable Algorithms". Most Searching & Sorting algorithms are polynomial time algorithms. **Ex:** Ordered Search **(O (log n)),** Polynomial evaluation **O(n)**
Sorting **O(n.log n)**

Group-II

Problems with solution times not bound by polynomial (simply non polynomial). These are hard or intractable problems. None of the problems in this group has been solved by any polynomial time algorithm. **Ex:** Traveling Sales Person $O(n^2 2^n)$**,** Knapsack $O(2^{n/2})$

No one has been able to develop a polynomial time algorithm for any problem in the group –II. 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 may of the problems with no polynomial time algorithms are computational 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 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 know to be NP-Complete.

Nondeterministic 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 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:

Algorithm Search(x){

//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();

}

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.

A Nondeterministic algorithm terminates unsuccessfully if and only if (iff) there exists no set of choices leading to a successful signal.

Nondeterministic Knapsack algorithm

      Algorithm DKP(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**
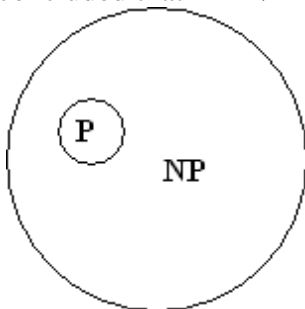
Commonly believed relationship between P & NP

The most famous unsolvable problems in Computer Science is Whether P=NP or P≠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.

159

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 α 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 ie., **Statisfiability α L**

A problem L is NP-Complete if and only if (iff) L is NP-Hard and **L Є NP**
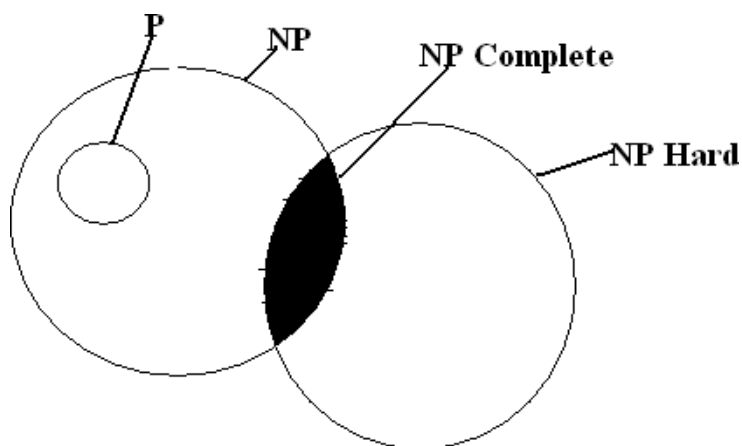
Commonly believed relationship among P, NP, NP-Complete and NP-Hard

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.