LECTURE NOTES ON

DESIGN AND ANALYSIS OF ALGORITHMS

B. TECH.

III YR I SEM

PREPARED BY

PRATIBHA SWAMY

SRIDEVI WOMENS ENGINEERING COLLEGE,

GANDIPET, HYDERABAD,

TELANGANA 500075

SYLLABUS

UNIT – I: Introduction-Algorithm definition, Algorithm Specification, Performance Analysis-Space complexity, Time complexity, Randomized Algorithms. Divide and conquer- General method, applications – Binary search, Merge sort, Quick sort, Strassen's Matrix Multiplication.

UNIT – II: Disjoint set operations, union and find algorithms, AND/OR graphs, Connected Components and Spanning trees, Bi-connected components Backtracking-General method, applications The 8-queen problem, sum of subsets problem, graph coloring, Hamiltonian cycles.

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

UNIT – IV: Dynamic Programming- General Method, applications- Chained matrix multiplication, All pairs shortest path problem, Optimal binary search trees, 0/1 knapsack problem, Reliability design, Traveling sales person problem.

UNIT – V: Branch and Bound- General Method, applications-0/1 Knapsack problem, LC Branch and Bound solution, FIFO Branch and Bound solution, Traveling sales person problem. NP-Hard and NP-Complete problems- Basic concepts, Non-deterministic algorithms, NP – Hard and NP- Complete classes, Cook's theorem.

UNIT – I

INTRODUCTION TO ALGORITHMS

Analysis of Algorithm:

Introduction – Analyzing control structures- average case analysis- Solving recurrences.

ALGORITHM

Informal Definition:

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 i/p into the o/p.

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.

Issues or study of Algorithm:

- How to device or design an algorithm creating and algorithm.
- How to express an algorithm definiteness.
- How to analysis an algorithm time and space complexity.
- How to validate an algorithm fitness.
- Testing the algorithm checking for error.

Algorithm Specification:

Algorithm can be described in three ways.

1. Natural language like English:

When this way is choused care should be taken, we should ensure that each & every statement is definite.

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 Conventions:

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>
}
```

repeat -until:

```
repeat
<statement-1>
                .
                .
                .
        <statement-n>
until<condition>
```

8. A conditional statement has the following forms.

```
If <condition> then <statement>
If <condition> then <statement-1>
Else <statement-1>
```

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 lists)

    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.

Next we present 2 examples to illustrate the process of translation problem into an
algorithm.

Selection Sort:

- Suppose we Must devise an algorithm that sorts a collection of n>=1 elements of arbitrary type.

- A Simple solution given by the following.

- (From those elements that are currently unsorted ,find the smallest & place it next in the sorted list.)

Algorithm:

```
For i:= 1 to n do
{
Examine a[I] to a[n] and suppose the smallest element is at a[j];
Interchange a[I] and a[j];
}
```

Finding the smallest element (sat a[j]) and interchanging it with a[i]

- We can solve the latter problem using the code,

```
        t    := a[i];
a[i]:=a[j];
                a[j]:=t;
```

- The first subtask can be solved by assuming the minimum is a[I];checking a[I] with a[I+1],a[I+2]…….,and whenever a smaller element is found, regarding it as the new minimum. a[n] is compared with the current minimum.

- Putting all these observations together, we get the algorithm Selection sort.

Theorem:

Algorithm selection sort(a,n) correctly sorts a set of n>=1 elements .The result remains is a a[1:n] such that a[1] <= a[2] ….<=a[n].

Selection Sort:

Selection Sort begins by finding the least element in the list. This element is moved to the front. Then the least element among the remaining element is found out and put into second position. This procedure is repeated till the entire list has been studied.

Example:

LIST L = 3,5,4,1,2

1 is selected , 1,5,4,3,2 2 is selected,
1,2,4,3,5
3 is selected, 1,2,3,4,5
4 is selected, 1,2,3,4,5

Proof:

- We first note that any I, say I=q, following the execution of lines 6 to 9,it is the case that a[q] Þ a[r],q<r<=n.
- Also observe that when ‘i’ becomes greater than q, a[1:q] is unchanged. Hence, following the last execution of these lines (i.e. I=n).We have a[1] <= a[2] <=……a[n].
- We observe this point that the upper limit of the for loop in the line 4 can be changed to n-1 without damaging the correctness of the algorithm.

Algorithm:

```
1. Algorithm selection sort (a,n)
2. // Sort the array a[1:n] into non-decreasing order.
3.{
4.      for I:=1 to n do
5.      {
6.              j:=I;
7.              for k:=i+1 to n do
8.                      if (a[k]<a[j])
```

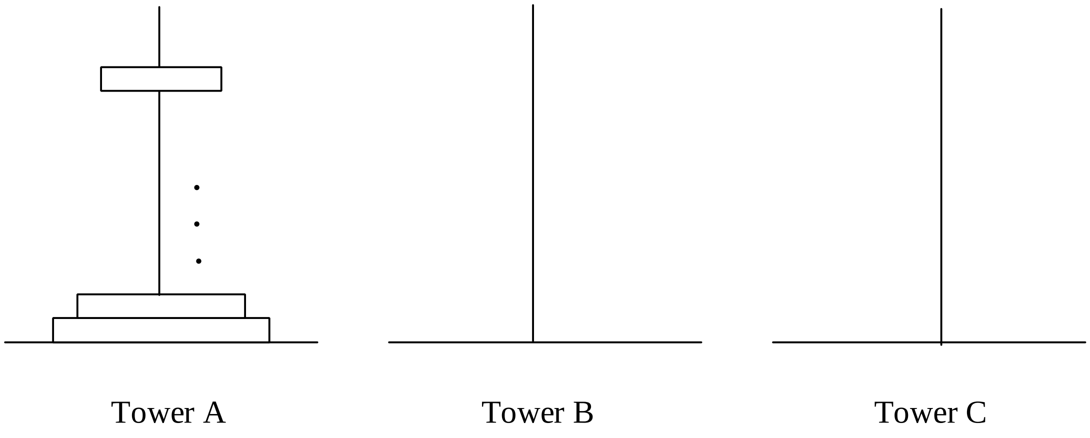
```
            t:=a[I];
            a[I]:=a[j];
            a[j]:=t;
    }
    }
```

Recursive Algorithms:

- A Recursive function is a function that is defined in terms of itself.
- Similarly, an algorithm is said to be recursive if the same algorithm is invoked in the body.
- An algorithm that calls itself is Direct Recursive.
- Algorithm ‘A’ is said to be Indirect Recursive if it calls another algorithm which in turns calls ‘A’.
- The Recursive mechanism, are externally powerful, but even more importantly, many times they can express an otherwise complex process very clearly. Or these reasons we introduce recursion here.
- The following 2 examples show how to develop a recursive algorithms.

In the first, we consider the Towers of Hanoi problem, and in the second, we generate all possible permutations of a list of characters.

1. **Towers of Hanoi:**



- It is Fashioned after the ancient tower of Brahma ritual.

- According to legend, at the time the world was created, there was a diamond tower (labeled A) with 64 golden disks.
- The disks were of decreasing size and were stacked on the tower in decreasing order of size bottom to top.
- Besides these tower there were two other diamond towers(labeled B & C)
- Since the time of creation, Brehman priests have been attempting to move the disks from tower A to tower B using tower C, for intermediate storage.
- As the disks are very heavy, they can be moved only one at a time.
- In addition, at no time can a disk be on top of a smaller disk.
- According to legend, the world will come to an end when the priest have completed this task.
- A very elegant solution results from the use of recursion.
- Assume that the number of disks is ‘n’.
- To get the largest disk to the bottom of tower B, we move the remaining ‘n-1’ disks to tower C and then move the largest to tower B.
- Now we are left with the tasks of moving the disks from tower C to B.
- To do this, we have tower A and B available.
- The fact, that towers B has a disk on it can be ignored as the disks larger than the disks being moved from tower C and so any disk scan be placed on top of it.

Algorithm:

```
Algorithm TowersofHanoi(n,x,y,z)
//Move the top ‘n’ disks from tower x to tower y.
{

        .
        .
        .

if(n>=1) then
        {
        TowersofHanoi(n-1,x,z,y);
        Write(“move top disk from tower “ X ,”to top of tower “ ,Y);
        Towersofhanoi(n-1,z,y,x);
        }
        }
```

2. Permutation Generator:

- Given a set of n>=1elements, the problem is to print all possible permutations of this
  set.
- For example, if the set is {a,b,c} ,then the set of permutation is,

  { (a,b,c),(a,c,b),(b,a,c),(b,c,a),(c,a,b),(c,b,a)}
- It is easy to see that given 'n' elements there are n! different permutations.
- A simple algorithm can be obtained by looking at the case of 4 statement(a,b,c,d)
- The Answer can be constructed by writing

1. a followed by all the permutations of (b,c,d)
2. b followed by all the permutations of(a,c,d)
3. c followed by all the permutations of (a,b,d)
4. d followed by all the permutations of (a,b,c)

Performance Analysis:

1. **Space Complexity:**

The space complexity of an algorithm is the amount of money it needs to run to compilation.

2. **Time Complexity:**

The time complexity of an algorithm is the amount of computer time it needs to run to compilation.

Space Complexity:

Space Complexity Example:

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

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

1.A fixed part that is independent of the characteristics (eg:number,size)of the inputs and outputs.

The part typically includes the instruction space (ie. Space for the code), space for simple variable and fixed-size component variables (also called aggregate) space for constants, and so on.

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

- The space requirement s(p) of any algorithm p may therefore be written as,

S(P) = c+ Sp(Instance characteristics)

Where ‘c’ is a constant.

Example 2:

```
        Algorithm sum(a,n)
        {
            s=0.0;
  for I=1 to n do    s=
s+a[I];
            return s;
        }
```

- The problem instances for this algorithm are characterized by n,the number of elements to be summed. The space needed d by ‘n’ is one word, since it is of type integer.
- The space needed by ‘a’a is the space needed by variables of tyepe array of floating point numbers.
- This is atleast ‘n’ words, since ‘a’ must be large enough to hold the ‘n’ elements to be summed.
- So,we obtain Ssum(n)>=(n+s)

[n for a[],one each for n,I a& s]

Time Complexity:

The time T(p) taken by a program P is the sum of the compile time and the run time(execution time)

The compile time does not depend on the instance characteristics. Also we may assume that a compiled program will be run several times without recompilation .This rum time is denoted by tp(instance characteristics).

The number of steps any problem statemn t is assigned depends on the kind of statement.

For example, comments 0 steps.
Assignment statements 1 steps.
[Which does not involve any calls to other algorithms]

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

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

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

Algorithm:

```
Algorithm sum(a,n)
{  s=  0.0;  count
        =
        count+1;
        for I=1 to
        n do
        {
         count    =count+1;
        s=s+a[I];
        count=count+1;
        }              count=count+1;
        count=count+1; return s;
        }
```

If the count is zero to start with, then it will be 2n+3 on termination. So each invocation of sum execute a total of 2n+3 steps.

2. The second method to determine the step count of an algorithm is to build a table in which we list the total number of steps contributes by each statement.

First determine the number of steps per execution (s/e) of the statement and the total number of times (ie., frequency) each statement is executed.

By combining these two quantities, the total contribution of all statements, the step count for the entire algorithm is obtained.

Statement	*S/e*	*Frequency*	*Total*
1. Algorithm Sum(a,n)	0	-	0
2.{	0	-	0
3. S=0.0;	1	1	1
4. for I=1 to n do	1	n+1	n+1
5. s=s+a[I];	1	n	n 1
6. return s;	1	1	0
7. }	0	-	
Total			2n+3

AVERAGE –CASE ANALYSIS

- Most of the time, average-case analysis are performed under the more or less realistic assumption that all instances of any given size are equally likely.
- For sorting problems, it is simple to assume also that all the elements to be sorted are distinct.
- Suppose we have ‘n’ distinct elements to sort by insertion and all n! permutation of these elements are equally likely.
- To determine the time taken on a average by the algorithm ,we could add the times required to sort each of the possible permutations ,and then divide by n! the answer thus obtained.
- An alternative approach, easier in this case is to analyze directly the time required by the algorithm, reasoning probabilistically as we proceed.
- For any I,2≤ I≤n, consider the sub array, T[1….i].

- The partial rank of T[I] is defined as the position it would occupy if the sub array were sorted.
- For Example, the partial rank of T[4] in [3,6,2,5,1,7,4] in 3 because T[1….4] once sorted is [2,3,5,6].
- Clearly the partial rank of T[I] does not depend on the order of the element in
- Sub array T[1…I-1].

Analysis

Best case:

This analysis constrains on the input, other than size. Resulting in the fasters possible run time

Worst case:

This analysis constrains on the input, other than size. Resulting in the fasters possible run time

Average case:

This type of analysis results in average running time over every type of input.

Complexity:

Complexity refers to the rate at which the storage time grows as a function of the problem size

Asymptotic analysis:

Expressing the complexity in term of its relationship to know function.
This type analysis is called asymptotic analysis.

Asymptotic notation:

Big 'oh': the function $f(n)=O(g(n))$ iff there exist positive constants c and no such that $f(n) \leq c*g(n)$ for all n, $n \geq no$.

Omega: the function $f(n)=\Omega(g(n))$ iff there exist positive constants c and no such that $f(n) \geq c*g(n)$ for all n, $n \geq no$.

Theta: the function f(n)=ɵ(g(n)) iff there exist positive constants c1,c2 and no such that $c1\ g(n) \leq f(n) \leq c2\ g(n)$ for all n, $n \geq no$.

Recursion:

Recursion may have the following definitions:
-The nested repetition of identical algorithm is recursion.
-It is a technique of defining an object/process by itself.
-Recursion is a process by which a function calls itself repeatedly until some specified condition has been satisfied.

When to use recursion:

Recursion can be used for repetitive computations in which each action is stated in terms of previous result. There are two conditions that must be satisfied by any recursive procedure.

1. Each time a function calls itself it should get nearer to the solution.
2. There must be a decision criterion for stopping the process.

In making the decision about whether to write an algorithm in recursive or non-recursive form, it is always advisable to consider a tree structure for the problem. If the structure is simple then use non-recursive form. If the tree appears quite bushy, with little duplication of tasks, then recursion is suitable.

The general procedure for any recursive algorithm is as follows,

1. Save the parameters, local variables and return addresses.
2. If the termination criterion is reached perform final computation and goto step 3 otherwise perform final computations and goto step 1

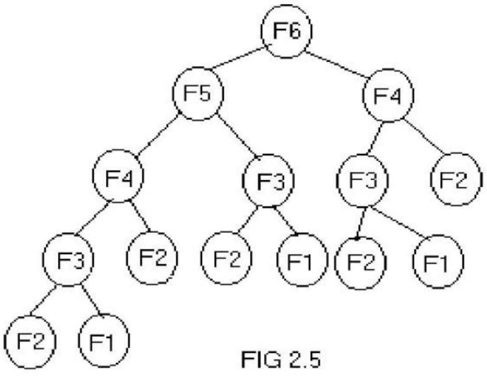


FIG 2.5

3. Restore the most recently saved parameters, local variable and return address and goto the latest return address.

Iteration v/s Recursion:

Demerits of recursive algorithms:

1. Many programming languages do not support recursion; hence, recursive mathematical function is implemented using iterative methods.
2. Even though mathematical functions can be easily implemented using recursion it is always at the cost of execution time and memory space. For example, the recursion tree for generating 6 numbers in a Fibonacci series generation is given in fig 2.5. A Fibonacci series is of the form 0,1,1,2,3,5,8,13,…etc, where the third number is the sum of preceding two numbers and so on. It can be noticed from the fig 2.5 that, f(n-2) is computed twice, f(n-3) is computed thrice, f(n-4) is computed 5 times.
3. A recursive procedure can be called from within or outside itself and to ensure its proper functioning it has to save in some order the return addresses so that, a return to the proper location will result when the return to a calling statement is made.
4. The recursive programs needs considerably more storage and will take more time.

Demerits of iterative methods :

- Mathematical functions such as factorial and Fibonacci series generation can be easily implemented using recursion than iteration.
- In iterative techniques looping of statement is very much necessary.

Recursion is a top down approach to problem solving. It divides the problem into pieces or selects out one key step, postponing the rest.

Iteration is more of a bottom up approach. It begins with what is known and from this constructs the solution step by step. The iterative function obviously uses time that is O(n) where as recursive function has an exponential time complexity.

It is always true that recursion can be replaced by iteration and stacks. It is also true that stack can be replaced by a recursive program with no stack.

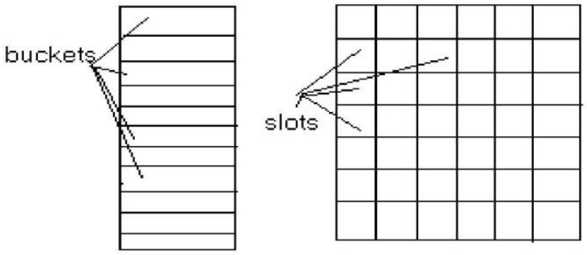


Fig 2.6

SOLVING RECURRENCES :-(Happen again (or) repeatedly)

- The indispensable last step when analyzing an algorithm is often to solve a recurrence equation.
- With a little experience and intention, most recurrence can be solved by intelligent guesswork.
- However, there exists a powerful technique that can be used to solve certain classes of recurrence almost automatically.
- This is a main topic of this section the technique of the characteristic equation.

1. Intelligent guess work:

This approach generally proceeds in 4 stages.

1. Calculate the first few values of the recurrence
2. Look for regularity.
3. Guess a suitable general form.
4. And finally prove by mathematical induction(perhaps constructive induction).

Then this form is correct.

Consider the following recurrence,

$$T(n) = \begin{cases} 0 & \text{if } n=0 \\ 3T(n \div 2)+n & \text{otherwise} \end{cases}$$

- First step is to replace n ÷ 2 by n/2

- It is tempting to restrict ‘n’ to being ever since in that case n÷2 = n/2, but recursively dividing an even no. by 2, may produce an odd no. larger than 1.
- Therefore, it is a better idea to restrict ‘n’ to being an exact power of 2.
- First, we tabulate the value of the recurrence on the first few powers of 2.

n	1	2	4	8	16	32
T(n)	1	5	19	65	211	665

* For instance, T(16) = 3 * T(8) +16
= 3 *
65 +16
= 211.

* Instead of writing T(2) = 5, it is more useful to write T(2) = 3 * 1 +2.

Then,

T(A)
= 3 *
T(2)
+4

= 3 *
(3 *
1
+2) +4

n	**T(n)**
1	1
2	3 * 1 +2
2^2	$3^2 * 1 + 3 * 2 + 2^2$
2^3	$3^3 * 1 + 3^2 * 2 + 3 * 2^2 + 2^3$
2^4	$3^4 * 1 + 3^3 * 2 + 3^2 * 2^2 + 3 * 2^3 + 2^4$
5	5 4 3 2 2 3 4 5

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

* We continue in this way, writing ‘n’ as an explicit power of 2.

2 3 * 1 + 3 * 2 + 3 * 2 + 3 * 2 + 3 * 2 + 2

- The pattern is now obvious.

$T(2^k) = 3^k 2^0 + 3^{k-1}2^1 + 3^{k-2}2^2+\ldots+3^1 2^{k-1} + 3^0 2^k.$

$= \sum\ 3^{k-i}\, 2^i$

$= 3^k \sum\ (2/3)^i$

$= 3^k * [(1 – (2/3)^{k + 1}) / (1 – (2/3)]$
$= 3^{k+1} – 2^{k+1}$

Proposition: (Geometric Series)

Let S_n be the sum of the first n terms of the geometric series a, ar, ar^2….Then $S_n = a(1-r^n)/(1-r)$, except in the special case when r = 1; when $S_n = a_{n.}$

$= 3^k * [(1 – (2/3)^{k+1}) / (1 – (2/3))]$

$= 3^k * [((3^{k+1} – 2^{k+1})/ 3^{k+1}) / ((3 – 2) / 3)]$

$3^{k+1} – 2^{k+1}$ 3
$= 3^k *$ -----------------
* ---- 3
k+1 1

$3^{k+1} – 2^{k+}1$
$= 3^k *$ ---------------
-- 3^{k+1-1}

$= 3^{k+1} – 2^{k+1}$

* It is easy to check this formula against our earlier tabulation.

EG : 2

$\{$ 0

n=0 t_n = 5

n=1 $3t_{n-1}$ +

$4t_{n-2}$, otherwise

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

Characteristics Polynomial, $x^2 – 3x – 4 = 0$

$(x – 4)(x + 1) = 0$

Roots $r_1 = 4$, $r_2 = -1$

General Solution, $f_n = C_1r_1^n + C_2\, r_2^n$

(A) n=0 □ $C_1 + C_2 = 0$ (1)

n=1 □ $C_1r_1 + C_2r_2 = 5$ (2)

Eqn 1 □ $C_1 = -C_2$

sub C_1 value in Eqn (2)

$-C_2r_1 + C_2r_2 = 5$

$C_2(r_2 – r_1) = 5$

5

C_2 = -------

r_2 – r_1

5

=

-1 + 4

= 5 / (-5) = -1

$C_2 = -1$, $C_1 = 1$

Sub C_1, C_2, r_1 & r_2 value in equation (A)

$f_n = 1.\ 4^n$ + (-

1) . $(-1)^n$ **$f_n = 4^n$**

$+ 1^n$

2. Homogenous Recurrences :

* We begin our study of the technique of the characteristic equation with the resolution of homogenous linear recurrences with constant co-efficient, i.e the recurrences of the form, $a_0t_n + a_1t_{n-1} + \ldots.. + a_kt_{n-k} = 0$

where the t_i are the values we are looking for.

* The values of t_i on 'K' values of i (Usually $0 \le i \le k-1$ (or) $0 \le i \le k$) are needed to determine the sequence.

* The initial condition will be considered later.

* The equation typically has infinitely many solution.

* The recurrence is, linear because it does not contain terms of the form t_{n-i}, t_{n-j}, t^2_{n-i}, and soon homogeneous because the linear combination of the t_{n-i} is equal to zero. With constant co-efficient because the a_i are constants

* Consider for instance our non familiar recurrence for the Fibonacci sequence, $f_n = f_{n-1} + f_{n-2}$

* This recurrence easily fits the mould of equation after obvious rewriting.

* $f_n – f_{n-1} – f_{n-2} = 0$

* Therefore, the fibonacci sequence corresponds to a homogenous linear recurrence with constant co-efficient with $k=2,a_0=1\&a_1=a_2 = -1$.

* In other words, if f_n & g_n satisfy equation.

So $\sum_{i=0}^{k} a_i f_{n-i} = 0$ & similarly for g_n & f_n

We set $t_n = C f_n + d g_n$ for arbitrary constants C & d, then t_n is also a solution to equation.

* This is true because,

$$a_0t_n + a_1t_{n-1} + \ldots + a_kt_{n-k}$$
$$= a_0(C f_n +d g_n) + a_1(C f_{n-1} + d g_{n-1}) + \ldots + a_k(C f_{n-k} + d g_{n-k})$$

$= C(a_0 f_n + a_1 f_{n-1} + \ldots +$
$a_k f_{n-k}) +$ $d(a_0 g_n + a_1 g_{n-}$
$_1 + \ldots + a_k g_{n-k})$ $= C * 0$
$+ d * 0$ $= 0.$

1) (Fibonacci) Consider the recurrence.

n if
n=0 or n=1
$f_n =$ $f_{n-1} + f_{n-2}$
otherwise

We rewrite the recurrence as,
$f_n - f_{n-1} - f_{n-2} = 0.$

The characteristic polynomial is,
$x^2 - x - 1 = 0.$

The roots are,

$$x = \frac{-(-1) \pm \sqrt{((-1)2 + 4)}}{2}$$

$$= \frac{1 \pm \sqrt{(1 + 4)}}{2}$$

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

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

The general solution is,

$$f_n = C_1 r_1^n + C_2 r_2^n$$

when n=0, $f_0 = C_1 + C_2 = 0$
when n=1, $f_1 = C_1r_1 + C_2r_2 = 1$

$C_1 + C_2 = 0$ (1)
$C_1r_1 + C_2r_2 = 1$ (2)

From equation (1)

$C_1 = -C_2$

Substitute C_1 in equation(2)
$-C_2r_1 + C_2r_2 = 1$
$C_2[r_2 - r_1] = 1$

Substitute r_1 and r_2 values

$1 \quad - \sqrt{5} \quad 1 - \sqrt{5}$
C_2 --------- - --------- = 1
$2 \quad 2$
$1 - \sqrt{5} - 1 - \sqrt{5}$
C_2 --------------------- = 1
2

$-C_2 * 2\sqrt{5}$
-------------- = 1
2

$-\sqrt{5}C_2 = 1$

$C_1 = 1/\sqrt{5}$ $C_2 = -1/\sqrt{5}$

Thus,
$1 \quad 1 + \sqrt{5} \quad ^{n} \quad -1 \quad 1$
$-\sqrt{5} \quad ^{n} \quad f_n =$ ---- --------- +
---- -------- $\sqrt{5} \quad 2$
$\sqrt{5} \quad 2$
$1 \quad 1 + \sqrt{5} \quad ^{n} \quad 1 - \sqrt{5} \quad ^{n}$
= ---- --------- - ---------
$\sqrt{5} \quad 2 \quad 2$

3. Inhomogeneous recurrence :

* The solution of a linear recurrence with constant co-efficient becomes more difficult when the recurrence is not homogeneous, that is when the linear combination is not equal to zero.

* Consider the following recurrence

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

* The left hand side is the same as before,(homogeneous) but on the right-hand side we have $b^n p(n)$, where, b is a constant p(n) is a polynomial in 'n' of degree 'd'.

Example(1) :

Consider the recurrence,

$t_n - 2t_{n-1} = 3^n$ (A)

In this case, b=3, p(n) = 1, degree = 0.

The characteristic polynomial is,

$(x - 2)(x - 3) = 0$

The roots are, $r_1 = 2$, $r_2 = 3$

The general solution,

$t_n = C_1 r_1^n + C_2 r_2^n$

$t_n = C_1 2^n + C_2 3^n$ (1)

when n=0, $C_1 + C_2 = t0$ (2)

when n=1, $2C_1 + 3C_2 = t1$ (3)

sub n=1 in eqn (A)

$t_1 - 2t_0 = 3$

$t_1 = 3 + 2t_0$

substitute t_1 in eqn(3),

(2) * 2 □ $2C_1 + 2C_2 = 2t_0$

$2C_1 + 3C_2 = (3 + 2t_0)$

$-C_2 \quad = \quad -3$

$C_2 = 3$

Sub $C_2 = 3$ in eqn (2)

$C_1 + C_2 = t_0$

$C_1 + 3 = t_0$

$C_1 = t_0 – 3$

Therefore $t_n = (t_0\text{-}3)2^n + 3.\ 3^n$

$= \text{Max}[O[(t_0 – 3)\ 2^n], O[3.3^n]]$

$= \text{Max}[O(2^n), O(3^n)]$ constants

$= O[3^n]$

Example :(2)

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

This is Inhomogeneous

In this case, b=3, p(n) = n+5, degree = 1

So, the characteristic polynomial is,

$(x\text{-}2)(x\text{-}3)^2 = 0$

The roots are,

$r_1 = 2, r_2 = 3, r_3 = 3$

The general equation, t_n

$= C_1r_1^n + C_2r_2^n + C_3nr_3^n$

(1)

when n=0, $t_0 = C_1 + C_2$ (2)

when n=1, $t_1 = 2C_1 + 3C_2 + 3C_3$ (3)

substituting n=1 in eqn(A),

$t_1 –$

$2t_0 = 6\ .\ 3$

$t_1 – 2t_0 = 18$
$t_1 = 18 + 2t_0$

substituting t_1 value in eqn(3)
$2C_1 + 3C_2 + 3C_3 = 18 + 2t_0$
(4) $C_1 + C_2 + = t_0$
(2)

Sub. n=2 in eqn(1)
$4C_1 + 9C_2 + 18C_3 = t_2$ (5)

sub n=2 in eqn (A)
$t_2 – 2t_1 = 7. 9$
$t_2 = 63 + 2t_1$
$= 63 + 2[18 +$
$2t_0]$ $t_2 = 63 +$
$36 +4t_0$ $t_2 =$
$99 +4t_0$

sub. t_2 value in eqn(3), $4C_1$
$+9C_2 + 18C_3 = 99 + 4t_0$
(5)

solve eqn (2),(4) & (5)

n=0, $C_1 + C_2 = t_0$
(2) n=1, $2C_1 + 3C_2 + 3C_3 = 18 +$
$2t_0$ (4) n=2, $4C_1 + 9C_2 + 18C_3$
$= 99 + 4t_0$ (5)

(4) * 6 □ $12C_1 + 18C_2 +18C_3 = 108 + 2t_0$ (4)
(5) □ $4C_1 + 9C_2 + 18C_3 = 99 + 4t0$ (5)

$8C_1 + 9C_2 = 9 + 8t_0$ (6)

(2) * 8 □ $8C_1 + 8C_2 = 8t_0$ (2)
(6) □ $8C_1 + 9C_2 = 9 +8t_0$ (6)

$-C_2 = -9$
$C_2 = 9$

Sub, $C_2 = 9$ in eqn(2)

$C_1 + C_2 = t_0$

$C_1 + 9 = t_0$

$C_1 = t_0-9$

Sub C_1 & C_2 in eqn (4)

$2C_1 + 3C_2 + 3C_3 = 18 + 2t_0$

$2(t_0-9) + 3(9) + 3C_3 = 18 + 2t_0$

$2t_0 - 18 + 27 + 3C_3 = 18 + 2t_0$

$2t_0 + 9 + 3C_3 = 18 + 2t_0$

$3C_3 = 18 - 9 + 2t_0 - 2t_0$

$3C_3 = 9$

$C_3 = 9/3$

$C_3 = 3$

Sub. C_1, C_2, C_3, r_1, r_2, r_3 values in eqn (1) $t_n = C_1 2^n + C_2 3^n + C_3.n.3^n$

$= (t_0 - 9)2^n + 9.3^n + 3.n.3^n$

$= \text{Max}[O[(t_0-9),2^n], O[9.3^n], O[3.n.3^n]]$

$= \text{Max}[O(2^n), O(3^n), O(n3^n)]$ tn = $O[n3^n]$

Example: (3)

Consider the recurrence,

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

$t_n - 4t_{n-1} = -2_n$ (A)

In this case , c=2, p(n) = -1, degree =0

$(x-4)(x-2) = 0$

The roots are, $r_1 = 4$, $r_2 = 2$

The general solution ,

$t_n = C_1r_1^n + C_2r_2^n$

$t_n = C_14^n + C_22^n$ (1)

when n=0, in (1) □ $C_1 + C_2 = 1$ (2)

when n=1, in (1) □ $4C_1 + 2C_2 = t1$ (3)

sub n=1 in (A),

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

$t_1 - 4t_0 = -2$ $t_1 =$

$4t_0 - 2$ [since $t_0 = 1$]

$t_1 = 2$

sub t1 value in eqn (3)

$4C_1 + 2C_2 = 4t_0 - 2$ (3)

(2) * 4 □ $4C_1 + 4C_2 = 4$

$-2C_2 = 4t_0 - 6$

$= 4(1) - 6$

$= -2$

$C_2 = 1$

$-2C_2 = 4t_0 - 6$

$2C_2 = 6 - 4t_0$

$C_2 = 3 - 2t_0$

$3 - 2(1) = 1$

$C_2 = 1$

Sub. C_2 value in eqn(2),

$C_1 + C_2 = 1$

$C_1 + (3\text{-}2t_0) = 1$

$C_1 + 3 - 2t_0 = 1$

$C_1 = 1 - 3 + 2t_0$

$C_1 = 2t_0 - 2$

$= 2(1) -$

$2 = 0$ $C_1 = 0$

Sub C_1 & C_2 value in eqn (1) $t_n = C_1 4^n + C_2 2^n$

$= \text{Max}[O(2t_0 – 2).4^n, O(3 – 2t_0).2^n]$

$= \text{Max}[O(2^n)]$

$\mathbf{t_n = O(2^n)}$

Example : (4)

if n=0 $\left\{ \begin{array}{l} 0 \\ t_n = \\ 2t_{n-1} + n + 2^n \text{ otherwise} \end{array} \right.$

$t_n – 2t_{n-1} = n + 2^n$ (A)

There are two polynomials.

For n; b=1, p(n), degree = 1

For 2n; b=2, p(n) = 1, degree =0

The characteristic polynomial is,

$(x-2)(x-1)^2(x-2) = 0$

The roots are, $r_1 = 2$, $r_2 = 2$, $r_3 = 1$, $r_4 = 1$. So, the general solution, $t_n = C_1 r_1^n + C_2 n r_2^n + C_3 r_3^n + C_4 n r_4^n$

sub r_1, r_2, r_3 in the above eqn

$t_n = 2^n C_1 + 2^n C_2 n + C_3 . 1^n + C_4 .n.1^n$ (1)

sub. n=0 □ $C_1 + C_3 = 0$ (2) sub. n=1 □ $2C_1 + 2C_2 + C_3 + C_4 = t_1$ (3)

sub. n=1 in eqn (A)

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

$t_1 – 2t_0 = 1 + 2$ t_1

$-\ 2t_0 = 3$ t_1
$= 3$ [since $t_0 = 0$]

sub. n=2 in eqn (1)

$2^2\ C_1 + 2.\ 2^2.C_2 + C_3 + 2.C_4 = t_2$
$4C_1 + 8C_2 + C_3 + 2C_4 = t_2$

sub n=2 in eqn (A)

$t_2 - 2t_1 = 2 + 2^2$

$t_2 - 2t_1 =$
$2 + 4$ t_2
$-\ 2t_1 = 6$
$t_2 = 6 + 2t_1$
$t_2 = 6 + 2.3$
$t_2 = 6 + 6$
$t_2 = 12$

○ $4C_1 + 8C_2 + C_3 + 2C_4 = 12$ (4)

sub n=3 in eqn (!)

$2^3\ C_1 + 3.2^3.C_2 + C_3 + 3C_4 = t_3$
$3C_1 + 24C_2 + C_3 + 3C_4 = t_3$

sub n=3 in eqn (A)

$t_3 - 2t_2 =$
$3 + 2^3$ t_3
$-\ 2t_2 = 3 + 8$
$t_3 - 2(12) =$
11 $t_3 - 2_4$
$=$ 11
$t_3 = 11 + 24$
$t_3 = 35$

○ $8C_1 + 24C_2 + C_3 + 3C_4 = 35$ (5)

n=0, solve; $C_1 + C_3 = 0$ (2)
n=1,(2), (3), (4)&(5) $2C_1 + 2C_2 + C_3 + C_4 = 3$ (3)
n=2, $4C_1 + 8C_2 + C_3 + 2C_4 = 12$ (4)
n=3, $8C_1 + 24C_2 + C_3 + 3C_4 = 35$ (5)

$-4C_1 – 16C_2 – C_4 = -23$ (6)

solve: (2) & (3)

(2) □ $C_1 + C_3 = 0$

(3) □ $2C_1 + C_3 + 2C_2 + C_4 = 3$

$-C_1 – 2C_2 – C_4 = -3$ (7)

solve(6) & (7)

(6) □ $-4C_1 – 16C_2 – C_4 =$
-23 (7) □ $-C_1 –$ $2C_2 –$
$C_4 = -3$

$-3C_1 – 14C_2 = 20$ (8)

4. Change of variables:

* It is sometimes possible to solve more complicated recurrences by making a change of variable.

* In the following example, we write T(n) for the term of a general recurrences, and t_i for the term of a new recurrence obtained from the first by a change of variable.

Example: (1)

Consider the recurrence,

$$T(n) = \begin{cases} 1 & \text{, if } n=1 \\ 3T(n/2) + n & \text{, if ‘n’ is a power of 2, } n>1 \end{cases}$$

□ Reconsider the recurrence we solved by intelligent guesswork in the previous section, but only for the case when ‘n’ is a power of 2

$$T(n) = \begin{cases} 1 \\ 3T(n/2) + n \end{cases}$$

* We replace ‘n’ by 2^i.

* This is achieved by introducing new recurrence t_i, define by $t_i = T(2^i)$ * This transformation is useful because n/2 becomes $(2^i)/2 = 2^{i-1}$

* In other words, our original recurrence in which T(n) is defined as a function of T(n/2) given way to one in which t_i is defined as a function of t_{i-1}, precisely the type of recurrence we have learned to solve.

$t_i = T(2^i) = 3T(2^{i-}$

${}^{1}) + 2^i$ $t_i = 3t_{i-1} +$

2^i

$t_i - 3t_{i-1} = 2^i$ (A)

In this case,

b = 2, p(n) = 1, degree = 0

So, the characteristic equation,

$(x - 3)(x - 2) = 0$

The roots are, r1 = 3, r2 = 2.

The general equation,

$t_n = C_1 r_1^i + C_2 r_2^i$

sub. r_1 & r_2: $t_n = 3^n C_1 +$

$C_2 2^n$ $t_n =$

$C_1 3^i + C_2 2^i$

We use the fact that, $T(2^i) = t_i$ & thus T(n) = tlogn when n= 2^i to obtain,

$T(n) = C_1.\ 3^{\log_2 n} + C_2.\ 2^{\log_2 n}$

$T(n) = C_1 .\ n^{\log_2 3} + C_2.n$ [i = logn]

When ‘n’ is a power of 2, which is sufficient to conclude that,

T(n) = O($n^{\log 3}$) ‘n’ is a power of 2

Example: (2)

Consider the recurrence,

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

Where ‘n’ is a power of 2, n ≥ 2.

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

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

□ $t_i - 4t_{i-1} = 4^i$

In this eqn,

$b = 4, P(n) = 1$, degree = 0

The characteristic polynomial,

$(x - 4)(x - 4) = 0$

The roots are, $r_1 = 4, r_2 = 4$.

So, the general equation, $t_i = C_1 4^i$
$+ C_2 4^i.i$ [since i = logn]
$= C_1\, 4^{\log n} + C_2.\, 4^{\log n} .\log n$ [since 2^i
$= n$]
$= C_1 .\, n_{\log 4} + C_2.\, n_{\log 4}.n_{\log 1}$

$T(n) = O(n^{\log 4})$ 'n' is the power of 2.

EXAMPLE : 3

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

When 'n' is a power of 2, $n \geq 2$

$t_i = T(2^i) = 2T(2^i/2) + 2^i .i$ [since $2^i = n$; $i = \log n$] t_i
$- 2t_{i-1} = i.\, 2^i$

In this case,

$b = 2, P(n) = i$, degree = 1

$(x - 2)(x - 2)^2 = 0$

The roots are, $r_1 = 2, r_2 = 2, r_3 = 2$

The general solution is,
$t_n = C_1 2^i + C_2.\, 2^i .\, i + C_3.\, i^2.$

2^i $= nC_1 + nC_2 +$
$nC_3(\log n^2{}_2 n)$

$\mathbf{t_n = O(n.\log^2{}_2 n)}$

Example: 4

$$T(n) = \begin{cases} 2 & ,n=1 \\ 5T(n/4) + Cn^2 & , n>1 \end{cases}$$

$t_i = T(4^i) = 5T(4^i/4)$
$+ C(4^i)^2$ $=$
$5T\ \ 4\ \ ^{i-1} + C.\ \ 16^i$
$= 5t_{\ i-1} + C.16^i$
$t_i – 5t_{\ i-1} = C.\ 16^i$

In this case, $b = 16$,
$P(n) = 1$, degree $= 0$

The characteristic eqn,
$(x – 5)(x – 16) = 0$

The roots are, $r_1 =5$, $r_2 = 16$

The general solution,
$t_i = C_1.5^i + C_2.16^i$
$= C_1.5^i + C_2.(4^2)^i$
$t_n = O(n^2)$

EXAMPLE: 5

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

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

$=$

$T(2^i/2) + C.$

$2^i = T(2$

${}^{i-1}) + C. 2^i$

$t_i = t_{i-1} + C.$

$2^i \quad t_i - t_{i-1} =$

$C. 2^i$

In this case, b=2, P(n) =1, degree =0

So, the characteristic polynomial,

$(x - 1)(x - 2) = 0$

The roots are, $r_1 =$
1, $r_2 = 2$ $t_i = C_1. 1^i$
$+ c_2. 2^i \quad = C_1. 1^{\log}$
${}_2{}^n + C_2.n$

$= C_1 . n^{\log_2 1} + C_2.n$

$t_n = O(n)$

EXAMPLE: 6

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

$t_i = T(2^i) =$
$3T(2^i/2) + 2^i$
$= 3T(2^{i-1}) + 2^i$

$t_i = 3t_{i-1} + 2^i$

So, b = 2, P(n) =1, degree = 0

$(x - 3)(x - 2) = 0$

The roots are, $r_1 = 3$,
$r_2 = 2$ $t_i = C_1.\ 3^i +$
$C_2.\ 2^i$

$= C_1.\ 3_{log2n} + C_2.\ 2_{log2n}$

$= C_1.\ n^{log_2 3} + C_2.\ n^{log_2 2} = 1$

$= C_1.\ n^{log_2 3} + C_2.n$

$t_n = O(n_{log23})$

EXAMPLE: 7

$T(n) = 2T(n/2) + n.\ logn$

$t_i = T(2^i) =$
$2T(2^i/2) + 2^i\ .\ i$
$= 2T(2^{i-1}) + i.\ 2^i$
$= 2t_{i-1} + i.2^i$ $t_i -$
$2t_{i-1} = i.\ 2i$

□ b=2, P(n) = I, degree = 1
The roots is $(x - 2)(x - 2)2 = 0$
$x = 2,2,2$

General solution,
$t_n = C_1.r_1^i + C_2.i.r_2^i + C_3.$
$i2.\ r_3^i$ $= C_1.2^i + C_2.\ i.2^i$
$+ C_3.\ i^2.2^i$

$= C_1.\ n + C_2.n.^{log_2 n} + C_3.i^2.n$

$= C_1.n + C_2.n.^{log_2 n} + C_3(2^{\ log_2 n}).n$

$t_n = O(n.\ 2^{\ log_2 n})$

5. Range Transformation:

* When we make a change of variable, we transform the domain of the recurrence. * Instead it may be useful to transform the range to obtain a recurrence in a form that we know how to solve
* Both transformation can be sometimes be used together.

EXAMPLE: 1

Consider the following recurrence , which defines T(n). where ‘n’ is the power Of 2

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

The first step is a change of variable, Let t_i denote $T(2^i)$

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

$= 2^i t^2_{\,i-1}$

* This recurrence is not clear, furthermore the co-efficient 2^i is not a constant.

* To transform the range, we create another recurrence by using ui to denote lg t_i $u_i = \lg t_i = i + 2\lg t_{\,i-1}$ $= i + 2u_{\,i-1}$

 $u_i - 2u_{\,i-1} = i$

$(x-2)(x-1)^2 = 0$

The roots are, x = 2, 1,

1. G.S, $u_i = C_1.2^i + C_2.1^i + C_3.i.1^i$

Sub. This solution into the recurrence,

For u_i yields,

$i = u_i - 2u_{\,i-1}$

$= C_1 2^i + C_2 + C_3.i - 2(C_1.\ 2^{\,i-1} + C_2 + C_3\ (i-1))$

$= (2C_3 - C_2) - C_3 i.$

$C_3 = -1$ & $C_2 = 2C_3 = -2$ $u_i = C_1 2^i - i - 2$

This gives us the G.S for t_i & T(n) $t_i = 2^{u_i} = 2^{C1 2i - i-2}$

$T(n) = t_{\lg n} = 2^{C1n - \log n - 2}$

$= 2^{C_1 n} / 4n$

We use he initial condition T(1) = 1/3
To determine C1: $T(1) = 2^{C_1} / 4 = 1/3$
Implies that C1 = lg(4/3) = 2 – log 3

The final solution is

$$\mathbf{T(n) = 2^{2n} / 4n\ 3^n}$$

1. Newton Raphson method: $x_2 = x_1 - f(x_1)/f^1(x_1)$

SEARCHING

Let us assume that we have a sequential file and we wish to retrieve an element matching with key 'k', then, we have to search the entire file from the beginning till the end to check whether the element matching k is present in the file or not.
There are a number of complex searching algorithms to serve the purpose of searching. The linear search and binary search methods are relatively straight forward methods of searching.

Sequential search:

In this method, we start to search from the beginning of the list and examine each element till the end of the list. If the desired element is found we stop the search and return the index of that element. If the item is not found and the list is exhausted the search returns a zero value.

In the worst case the item is not found or the search item is the last (nth) element. For both situations we must examine all n elements of the array so the order of magnitude or complexity of the sequential search is n. i.e., O(n). The execution time for this algorithm is proportional to n that is the algorithm executes in linear time.
.

Binary search:

Binary search method is also relatively simple method. For this method it is necessary to have the vector in an alphabetical or numerically increasing order. A search for a particular item with X resembles the search for a word in the dictionary. The approximate mid entry is located and its key value is examined. If the mid value is greater than X, then the list is chopped off at the (mid-1)th location. Now the list gets reduced to half the original list. The middle entry of the left-reduced list is examined in a similar manner. This procedure is repeated until the item is found or the list has no more elements. On the other hand, if the mid value is lesser than X, then the list is chopped off at (mid+1)th location. The middle entry of the right-reduced list is examined and the procedure is continued until desired key is found or the search interval is exhausted.

The algorithm for binary search is as follows,

Algorithm : binary search
Input : A, vector of n elements
K, search element
Output : low –index of k
Method : low=1,high=n

```
While(low<=high-1)
{
mid=(low+high)/2
if(k<a[mid])
high=mid
else
low=mid
if end
}
if(k=A[low])
{
write("search  successful")
write(k is at location low)
exit();  }  else  write
(search
unsuccessful);
if      end;
algorithm
ends.
```

SORTING

One of the major applications in computer science is the sorting of information in a table. Sorting algorithms arrange items in a set according to a predefined ordering relation. The most common types of data are string information and numerical information. The ordering relation for numeric data simply involves arranging items in sequence from smallest to largest and from largest to smallest, which is called ascending and descending order respectively.

The items in a set arranged in non-decreasing order are {7,11,13,16,16,19,23}. The items in a set arranged in descending order is of the form {23,19,16,16,13,11,7}

Similarly for string information, {a, abacus, above, be, become, beyond}is in ascending order and { beyond, become, be, above, abacus, a}is in descending order.

There are numerous methods available for sorting information. But, not even one of them is best for all applications. Performance of the methods depends on parameters like, size of the data set, degree of relative order already present in the data etc.

Selection sort :

The idea in selection sort is to find the smallest value and place it in an order, then find the next smallest and place in the right order. This process is continued till the entire table is sorted.

Consider the unsorted array, a[1] a[2] a[8]

20	35	18	8	14	41	3	39

The resulting array should be

a[1] a[2] a[8]

3	8	14	18	20	35	39	41

One way to sort the unsorted array would be to perform the following steps:

- Find the smallest element in the unsorted array
- Place the smallest element in position of a[1]

  i.e., the smallest element in the unsorted array is 3 so exchange the values of a[1] and a[7]. The array now becomes,

a[1] a[2] a[8]

3	35	18	8	14	41	20	39

Now find the smallest from a[2] to a[8] , i.e., 8 so exchange the values of a[2] and a[4] which results with the array shown below,

a[1] a[2] a[8]

3	8	18	35	14	41	20	39

Repeat this process until the entire array is sorted. The changes undergone by the array is shown in fig 2.2.The number of moves with this technique is always of the order O(n).

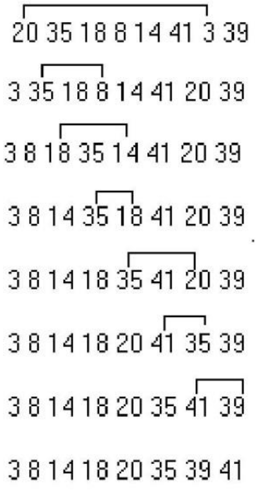


FIG 2.2

Bubble sort:

Bubble Sort is an elementary sorting algorithm. It works by repeatedly exchanging adjacent elements, if necessary. When no exchanges are required, the file is sorted.

BUBBLESORT (A)

```
for i ← 1 to length [A] do
    for j ← length [A] downto i+1 do
        If A[A] < A[j-1] then
    Exchange A[j] ↔ A[j-1]
```

Here the number of comparison made

$1 + 2 + 3 + . . . + (n-1) = n(n-1)/2 = O(n^2)$

Another well-known sorting method is bubble sort. It differs from the selection sort in that instead of finding the smallest record and then performing an interchange two records are interchanged immediately upon discovering that they are of out of order.

When this approach is used there are at most n-1 passes required. During the first pass k1and k2 are compared, and if they are out of order, then records R1 AND R2 are interchanged; this process is repeated for records R2 and R3, R3 and R4 and so on .this method will cause with small keys to bubble up. After the first pass the record with the largest key will be in the nth position. On each successive pass, the records with the next largest key will be placed in the position n-1, n-2 …2 respectively, thereby resulting in a sorted table.

After each pass through the table, a check can be made to determine whether any interchanges were made during that pass. If no interchanges occurred then the table must be sorted and no further passes are required.

Insertion sort :

Insertion sort is a straight forward method that is useful for small collection of data. The idea here is to obtain the complete solution by inserting an element from the unordered part into the partially ordered solution extending it by one element. Selecting an element from the unordered list could be simple if the first element of that list is selected.

a[1] a[2] a[8]

20	35	18	8	14	41	3	39

Initially the whole array is unordered. So select the minimum and put it in place of a[1] to act as sentinel. Now the array is of the form,

a[1] a[2] a[8]

3	35	18	8	14	41	20	39

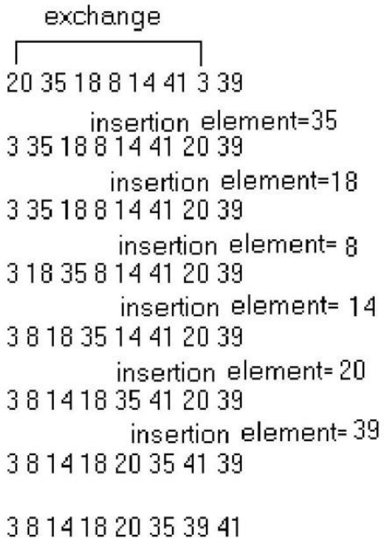
Now we have one element in the sorted list and the remaining elements are in the unordered set. Select the next element to be inserted. If the selected element is less than the preceding element move the preceding element by one position and insert the smaller element.

In the above array the next element to be inserted is x=35, but the preceding element is 3 which is less than x. Hence, take the next element for insertion i.e., 18. 18 is less than 35, so move 35 one position ahead and place 18 at that place. The resulting array will be,

a[1] a[2] a[8]

3	18	35	8	14	41	20	39

Now the element to be inserted is 8. 8 is less than 35 and 8 is also less than 18 so move 35 and 18 one position right and place 8 at a[2]. This process is carried till the sorted array is obtained.



```
exchange
20 35 18 8 14 41 3 39
        insertion element=35
3 35 18 8 14 41 20 39
          insertion element=18
3 35 18 8 14 41 20 39
           insertion element= 8
3 18 35 8 14 41 20 39
           insertion element= 14
3 8 18 35 14 41 20 39
           insertion element= 20
3 8 14 18 35 41 20 39
            insertion element= 39
3 8 14 18 20 35 41 39

3 8 14 18 20 35 39 41
```

FIG 2.3

The changes undergone are shown in fig 2.3.

One of the disadvantages of the insertion sort method is the amount of movement of data. In the worst case, the number of moves is of the order O(n^2). For lengthy records it is quite time consuming.

Heaps and Heap sort:

A heap is a complete binary tree with the property that the value at each node is atleast as large as the value at its children.

The definition of a max heap implies that one of the largest elements is at the root of the heap. If the elements are distinct then the root contains the largest item. A max heap can be implemented using an array an[ ].

To insert an element into the heap, one adds it "at the bottom" of the heap and then compares it with its parent, grandparent, great grandparent and so on, until it is less than or equal to one of these values. Algorithm insert describes this process in detail.

```
Algorithm Insert(a,n)
{
// Insert a[n] into the heap which is stored in a[1:n-1]
I=n;  item=a[n];  while( (I>n)
and (a[ I!/2 ] < item)) do
{
a[I] = a[I/2];
I=I/2
;     }
a[I]=
item;
retur
n
(true
);
}
```

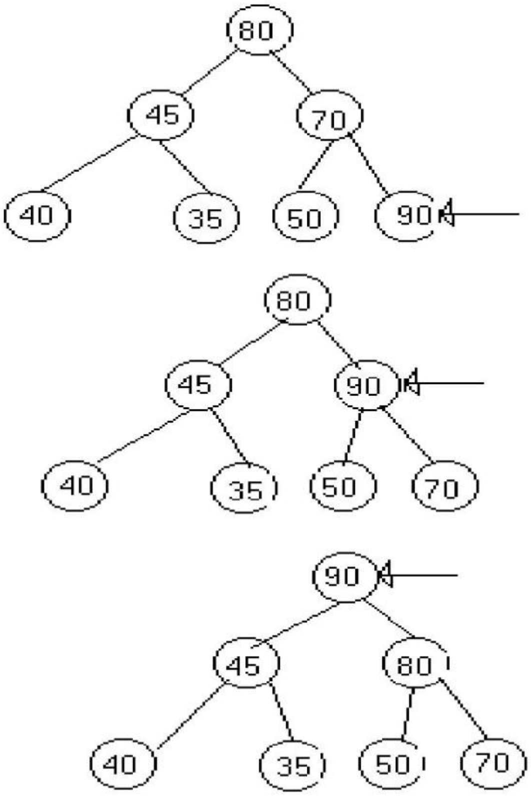


Fig 7.5

The figure shows one example of how insert would insert a new value into an existing heap of five elements. It is clear from the algorithm and the figure that the time for insert can vary. In the best case the new elements are correctly positioned initially and no new values need to be rearranged. In the worst case the number of executions of the while loop is proportional to the number of levels in the heap. Thus if there are n elements in the heap, inserting new elements takes O(log n) time in the worst case.

To delete the maximum key from the max heap, we use an algorithm called Adjust. Adjust takes as input the array a[ ] and integer I and n. It regards a[1..n] as a complete binary tree. If the subtrees rooted at 2I and 2I+1 are max heaps, then adjust will rearrange elements of a[ ] such that the tree rooted at I is also a max heap. The maximum elements from the max heap a[1..n] can be deleted by deleting the root of the corresponding complete binary tree. The last element of the array, i.e. a[n], is copied to the root, and finally we call Adjust(a,1,n-1).

```
Algorithm Adjust(a,I,n)
{
j=
2I
;
ite
m
=a
[I]
;
while (j<=n) do
{
if  ((j<=n)  and  (a[j]<  a[j+1]))
then  j=j+1;
                                                  //compare left and right child and let j be the
                                                  right //child
if ( item >= a[I]) then
break;   // a position
for  item  is  found
a[i/2]=a[j];  j=2I;  }
a[j/2]=item;
}
```

```
Algorithm Delmac(a,n,x)
// Delete the maximum from the heap a[1..n] and store it in x
{
if (n=0) then
{
write('heap is empty");
return (false);
}           x=a[1];           a[1]=a[n];
Adjust(a,1,n-1);
Return(true);
}
```

Note that the worst case run time of adjust is also proportional to the height of the tree. Therefore if there are n elements in the heap, deleting the maximum can be done in O(log n) time.

To sort n elements, it suffices to make n insertions followed by n deletions from a heap since insertion and deletion take O(log n) time each in the worst case this sorting algorithm has a complexity of O(n log n).

```
Algorithm sort(a,n)
{
for   i=1   to   n   do
Insert(a,i);
for i= n to 1 step –1 do
{
Delmax(a,i,x);
a[i]=x;
}
}
```

CHAPTER- 2

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.

- D And C(Algorithm) is initially invoked as D and C(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 …Pk are solved by recursive application of D And C.

- 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 D And C is described by the recurrence relation.

$$T(n)= \begin{cases} g(n) & \text{n small} \\ T(n1)+T(n2)+\ldots\ldots\ldots\ldots+T(nk)+f(n); & \text{otherwise.} \end{cases}$$

Where T(n) is the time for D And C 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 D And C(P)
{
if small(P) then return S(P);
else
{
divide P into smaller instances
        P1, P2… Pk, k>=1;
Apply D And C to each of these sub problems;
return combine (D And C(P1), D And C(P2),…….,D And C(Pk));
}
}
```

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

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

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 $\log n >=I>=1$.

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

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

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

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

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

$= 2n + n \log n$

BINARY SEARCH

```
1.  Algorithm Bin search(a,n,x)
2.  // Given an array a[1:n] of elements in non-decreasing 3. //order,
    n>=0,determine whether 'x' is present and
4.      // if so, return 'j' such that x=a[j]; else return 0.
5.      {
6.      low:=1; high:=n;
7.      while (low<=high) do
8.      {
9.      mid:=[(low+high)/2];
10.     if (x<a[mid]) then high; 11.        else if(x>a[mid]) then
        low=mid+1;
12.   else return mid;
13.   }
14.   return 0;
15.    }
```

- Algorithm, describes this binary search method, where Binsrch has 4I/ps a[], I , l & 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/ps a,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 Bin search on these 3 steps.

X=151	low	high	mid
	1	14	7
	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.

Maximum and Minimum:

- Let us consider another simple problem that can be solved by the divide-andconquer technique.
- The problem is to find the maximum and minimum items in a set of 'n' elements.
- In analyzing the time complexity of this algorithm, we once again concentrate on the no. of element comparisons.

- More importantly, when the elements in a[1:n] are polynomials, vectors, very large numbers, or strings of character, the cost of an element comparison is much higher than the cost of the other operations.

- Hence, the time is determined mainly by the total cost of the element comparison.

```
Algorithm straight MaxMin(a,n,max,min)
// set max to the maximum & min to the minimum of a[1:n]
{
    max:=min:=a[1];
    for I:=2 to n do
    {
        if(a[I]>max) then max:=a[I];
        if(a[I]<min) then min:=a[I];
    }
}
```

Algorithm: Straight forward Maximum & Minimum

- Straight MaxMin requires 2(n-1) element comparison in the best, average & worst cases.

- An immediate improvement is possible by realizing that the comparison a[I]<min is necessary only when a[I]>max is false.

  Hence we can replace the contents of the for loop by,

```
If(a[I]>max) then max:=a[I];
Else if (a[I]<min) then min:=a[I];
```

- Now the best case occurs when the elements are in increasing order. The no. of element comparison is (n-1).

- The worst case occurs when the elements are in decreasing order. The no. of elements comparison is 2(n-1)

- The average no. of element comparison is < than 2(n-1)

- On the average a[I] is > than max half the time, and so, the avg. no. of comparison is 3n/2-1.

- A divide- and conquer algorithm for this problem would proceed as follows:

Let P=(n, a[I] ,……,a[j]) denote an arbitrary instance of the problem.

Here ‘n’ is the no. of elements in the list (a[I],….,a[j]) and we are interested in finding the maximum and minimum of the list.

- If the list has more than 2 elements, P has to be divided into smaller instances.

- For example , we might divide ‘P’ into the 2 instances,
  P1=([n/2],a[1],……..a[n/2]) & P2= (n-[n/2],a[[n/2]+1],…..,a[n])

- After having divided ‘P’ into 2 smaller sub problems, we can solve them by recursively invoking the same divide-and-conquer algorithm.

Algorithm: Recursively Finding the Maximum & Minimum

```
Algorithm MaxMin (I,j,max,min)
//a[1:n] is a global array, parameters I & j
//are integers, 1<=I<=j<=n.The effect is to
//set max & min to the largest & smallest value
//in a[I:j], respectively.
{
    if(I=j) then max:= min:= a[I];
    else if (I=j-1) then // Another case of small(p)
    {
        if (a[I]<a[j]) then
        {
            max:=a[j];
            min:=a[I];
        } else
        {
            max:=a[I];
            min:=a[j];
        }
    }
    else
    {
```

```
// if P is not small, divide P into subproblems.
// find where to split the set mid:=[(I+j)/2];
//solve the subproblems
MaxMin(I,mid,max.min);
MaxMin(mid+1,j,max1,min1);
//combine the solution
if (max<max1) then max=max1;
if(min>min1) then min = min1;
}
}
```

- The procedure is initially invoked by the statement,
  MaxMin(1,n,x,y)
- Suppose we simulate MaxMin on the following 9 elements

A:	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
	22	13	-5	-8	15	60	17	31	47

- A good way of keeping track of recursive calls is to build a tree by adding a node each time a new call is made.
- For this Algorithm, each node has 4 items of information: I, j, max & imin.
- Examining fig: we see that the root node contains 1 & 9 as the values of I &j corresponding to the initial call to MaxMin.
- This execution produces 2 new calls to MaxMin, where I & j have the values 1, 5 & 6, 9 respectively & thus split the set into 2 subsets of approximately the same size.
- From the tree, we can immediately see the maximum depth of recursion is 4. (including the 1st call)
- The include no.s in the upper left corner of each node represent the order in which max & min are assigned values.

No. of element Comparison:

- If T(n) represents this no., then the resulting recurrence relations is

T(n)={ T([n/2]+T[n/2]+2 n>2

n=2

0 n=1

When ‘n’ is a power of 2, n=2^k for some +ve integer ‘k’, then

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

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

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

*

*

$= 2^k-1T(2)+$

$= 2^k-1+2^k-2$

$= 2^k/2+2^k-2$

$= n/2+n-2$

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

$T(n)=(3n/2)-2$

*Note that (3n/3)-3 is the best-average, and worst-case no. of comparisons when ‘n’ is a power of 2.

MERGE SORT

- As another example divide-and-conquer, we investigate a sorting algorithm that has the nice property that is the worst case its complexity is O(n log n)
- This algorithm is called merge sort
- We assume throughout that the elements are to be sorted in non-decreasing order.
- Given a sequence of ‘n’ elements a[1],…,a[n] the general idea is to imagine then split into 2 sets a[1],…..,a[n/2] and a[[n/2]+1],….a[n].
- Each set is individually sorted, and the resulting sorted sequences are merged to produce a single sorted sequence of ‘n’ elements.
- Thus, we have another ideal example of the divide-and-conquer strategy in which the splitting is into 2 equal-sized sets & the combining operation is the merging of 2 sorted sets into one.

Algorithm For Merge Sort:

1. Algorithm MergeSort(low,high)
2. //a[low:high] is a global array to be sorted
3. //Small(P) is true if there is only one element 4. //to sort. In this case the list is already sorted.
5. {
6. if (low<high) then //if there are more than one element
7. {
8. //Divide P into subproblems
9. //find where to split the set

```
mid = [(low+high)/2]; //solve the subproblems.
mergesort (low,mid);
mergesort(mid+1,high);
//combine the solutions .
merge(low,mid,high);
}
}
```

Algorithm: Merging 2 sorted subarrays using auxiliary storage.

```
Algorithm merge(low,mid,high)
//a[low:high] is a global array containing
//two sorted subsets in a[low:mid]
//and in a[mid+1:high].The goal is to merge these 2 sets into
//a single set residing in a[low:high].b[] is an auxiliary global array.
{
    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 high do
        {
            b[I]=a[k];
            I=I+1;
        }
    else
        for k=h to mid do
    {
    b[I]=a[k];
    I=I+1;
    }
    for k=low to high do a[k] = b[k];
    }
```

- Consider the array of 10 elements a[1:10] =(310, 285, 179, 652, 351, 423, 861, 254, 450, 520)

- Algorithm Mergesort begins by splitting a[] into 2 sub arrays each of size five (a[1:5] and a[6:10]).
- The elements in a[1:5] are then split into 2 sub arrays of size 3 (a[1:3]) and 2(a[4:5])
- Then the items in a a[1:3] are split into sub arrays of size 2 a[1:2] & one(a[3:3])
- The 2 values in a[1:2} are split to find time into one-element sub arrays, and now the merging begins.

(310| 285| 179| 652, 351| 423, 861, 254, 450, 520)

Where vertical bars indicate the boundaries of sub arrays.

Elements a[I] and a[2] are merged to yield,
(285, 310|179|652, 351| 423, 861, 254, 450, 520)

Then a[3] is merged with a[1:2] and
(179, 285, 310| 652, 351| 423, 861, 254, 450, 520)

Next, elements a[4] & a[5] are merged.
(179, 285, 310| 351, 652 | 423, 861, 254, 450, 520)

And then a[1:3] & a[4:5]
(179, 285, 310, 351, 652| 423, 861, 254, 450, 520)

Repeated recursive calls are invoked producing the following sub arrays.
(179, 285, 310, 351, 652| 423| 861| 254| 450, 520)

Elements a[6] &a[7] are merged.

Then a[8] is merged with a[6:7]
(179, 285, 310, 351, 652| 254,423, 861| 450, 520)

Next a[9] &a[10] are merged, and then a[6:8] & a[9:10]
(179, 285, 310, 351, 652| 254, 423, 450, 520, 861)

At this point there are 2 sorted sub arrays & the final merge produces the fully sorted result.
(179, 254, 285, 310, 351, 423, 450, 520, 652, 861)

- If the time for the merging operations is proportional to 'n', then the computing time for merge sort is described by the recurrence relation.

T(n) = { a n=1,'a' a constant
2T(n/2)+cn n>1,'c' a constant.

When 'n' is a power of 2, n= 2^k, we can solve this equation by successive substitution.

T(n) =2(2T(n/4) +cn/2) +cn
= 4T(n/4)+2cn
= 4(2T(n/8)+cn/4)+2cn
*
*
= 2^k T(1)+kCn.
= an + cn log n.

It is easy to see that if s^k<n<=2^k+1, then T(n)<=T(2^k+1). Therefore,

T(n)=O(n log n)

QUICK SORT

- The divide-and-conquer approach can be used to arrive at an efficient sorting method different from merge sort.

- In merge sort, the file a[1:n] was divided at its midpoint into sub arrays which were independently sorted & later merged.

- In Quick sort, the division into 2 sub arrays is made so that the sorted sub arrays do not need to be merged later.

- This is accomplished by rearranging the elements in a[1:n] such that a[I]<=a[j] for all I between 1 & n and all j between (m+1) & n for some m, 1<=m<=n.

- Thus the elements in a[1:m] & a[m+1:n] can be independently sorted.

- No merge is needed. This rearranging is referred to as partitioning.

- Function partition of Algorithm accomplishes an in-place partitioning of the elements of a[m:p-1]

- It is assumed that a[p]>=a[m] and that a[m] is the partitioning element. If m=1 & p-1=n, then a[n+1] must be defined and must be greater than or equal to all elements in a[1:n]

- The assumption that a[m] is the partition element is merely for convenience, other choices for the partitioning element than the first item in the set are better in practice.

- The function interchange (a,I,j) exchanges a[I] with a[j].

Algorithm: Partition the array a[m:p-1] about a[m]

```
Algorithm Partition(a,m,p)
//within a[m],a[m+1],…..,a[p-1] the elements
// are rearranged in such a manner that if
//initially t=a[m],then after completion
//a[q]=t for some q between m and
//p-1,a[k]<=t for m<=k<q, and  7. //a[k]>=t for  q<k<p. q is returned
    //Set a[p]=infinite.
    {
    v=a[m];I=m;j=p;
    repeat
    {
    repeat
    I=I+1;
    until(a[I]>=v);
    repeat
    j=j-1;
    until(a[j]<=v);
    if (I<j) then interchange(a,i.j);
    }until(I>=j);
    a[m]=a[j]; a[j]=v;
    retun j;
    }
```

```
Algorithm Interchange(a,I,j)
//Exchange a[I] with a[j]
{
p=a[I];
a[I]=a[j];
a[j]=p;
}
```

Algorithm*:* Sorting by Partitioning

```
Algorithm Quicksort(p,q)
//Sort the elements a[p],….a[q] which resides
//is the global array a[1:n] into ascending
//order; a[n+1] is considered to be defined
// and must be >= all the elements in a[1:n]
{
if(p<q) then // If there are more than one element
{
// divide p into 2 subproblems
j=partition(a,p,q+1);
//’j’ is the position of the partitioning element.
//solve the subproblems.
quicksort(p,j-1);
quicksort(j+1,q);
//There is no need for combining solution.
}
}
```

Record Program: Quick Sort

```
#include  <stdio.h>
#include
<conio.h>          int
a[20]; main() {
    int n,I;
    clrscr();
   printf("QUICK SORT");      printf("\n
Enter    the    no.    of    elements    ");
scanf("%d",&n);         printf("\nEnter the
array elements");           for(I=0;I<n;I++)
scanf("%d",&a[I]);        quicksort(0,n-1);
printf("\nThe    array    elements    are");
for(I=0;I<n;I++)        printf("\n%d",a[I]);
getch(); }
quicksort(int p, int q)
{
   int          j;
if(p,q)

{          j=partition(p,q+
1);           quicksort(p,j-
1);
       quicksort(j+1,q);
    }
}

Partition(int m, int p)
{
    int  v,I,j;         v=a[m];
i=m;          j=p;          do
{                            do
i=i+1;
while(a[i]<v);              if
(i<j)
interchange(I,j);            }
while   (I<j);   a[m]=a[j];
a[j]=v; return j; }

Interchange(int I, int j)
{
    int       p;
p=        a[I];
a[I]=a[j];
a[j]=p;
}
```

```
Output:
Enter the no. of elements 5
Enter the array elements
3
8
1

5
2
The sorted elements are,
1
2
3
5
8
```

STRASSON'S MATRIX MULTIPLICAION

- Let A and B be the 2 n*n Matrix. The product matrix C=AB is calculated by using the formula,

  C (i ,j)=   A(i,k) B(k,j) for all 'i' and and j between 1 and n.

- The time complexity for the matrix Multiplication is O(n^3).

- Divide and conquer method suggest another way to compute the product of n*n matrix.

- We assume that N is a power of 2 .In the case N is not a power of 2 ,then enough rows and columns of zero can be added to both A and B .SO that the resulting dimension are the powers of two.

- If n=2 then the following formula as a computed using a matrix multiplication operation for the elements of A & B.

- If n>2,Then the elements are partitioned into sub matrix n/2*n/2..since ‘n’ is a power of 2 these product can be recursively computed using the same formula .This Algorithm will continue applying itself to smaller sub matrix until ‘N” become suitable small(n=2) so that the product is computed directly .
- The formula are

$$\begin{pmatrix} A11 & A12 \\ A21 & A21 \end{pmatrix} * \begin{pmatrix} B11 & B12 \\ B21 & B22 \end{pmatrix} = \begin{pmatrix} C11 & C12 \\ C21 & C22 \end{pmatrix}$$

C11 = A11 B11 + A12 B21
C12 = A11 B12 + A12 B22
C21 = A21 B11 + A22 B21
C22 = A21 B12 + A22 B22

For EX:

$$4 * 4 = \begin{pmatrix} 2 & 2 & 2 & 2 \\ 2 & 2 & 2 & 2 \\ 2 & 2 & 2 & 2 \\ 2 & 2 & 2 & 2 \end{pmatrix} * \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

The Divide and conquer method

$$\begin{pmatrix} \begin{vmatrix} 2 & 2 \\ 2 & 2 \end{vmatrix} & \begin{vmatrix} 2 & 2 \\ 2 & 2 \end{vmatrix} \\ \begin{vmatrix} 2 & 2 \\ 2 & 2 \end{vmatrix} & \begin{vmatrix} 2 & 2 \\ 2 & 2 \end{vmatrix} \end{pmatrix} * \begin{pmatrix} \begin{vmatrix} 1 & 1 \\ 1 & 1 \end{vmatrix} & \begin{vmatrix} 1 & 1 \\ 1 & 1 \end{vmatrix} \\ \begin{vmatrix} 1 & 1 \\ 1 & 1 \end{vmatrix} & \begin{vmatrix} 1 & 1 \\ 1 & 1 \end{vmatrix} \end{pmatrix} = \begin{pmatrix} \begin{vmatrix} 4 & 4 \\ 4 & 4 \end{vmatrix} & \begin{vmatrix} 4 & 4 \\ 4 & 4 \end{vmatrix} \\ \begin{vmatrix} 4 & 4 \\ 4 & 4 \end{vmatrix} & \begin{vmatrix} 4 & 4 \\ 4 & 4 \end{vmatrix} \end{pmatrix}$$

- To compute AB using the equation we need to perform 8 multiplication of n/2*n/2 matrix and from 4 addition of n/2*n/2 matrix.
- Ci,j are computed using the formula in equation 4
- As can be sum P, Q, R, S, T, U, and V can be computed using 7 Matrix multiplication and 10 addition or subtraction.
- The Cij are required addition 8 addition or subtraction.

$$T(n)=\begin{cases} b & n<=2 \text{ a \&b are} \\ 7T(n/2)+an^2 & n>2 \text{ constant} \end{cases}$$

Finally we get T(n) =O(n ^log27)

Example

$$\begin{vmatrix} 4 & 4 \\ 4 & 4 \end{vmatrix} * \begin{vmatrix} 4 & 4 \\ & 4 \end{vmatrix} \; 4$$

P=(4*4)+(4+4)=64
Q=(4+4)4=32
R=4(4-4)=0
S=4(4-4)=0
T=(4+4)4=32
U=(4-4)(4+4)=0
V=(4-4)(4+4)=0
C11=(64+0-32+0)=32
C12=0+32=32
C21=32+0=32
C22=64+0-32+0=32

So the answer c(i,j) is $\begin{vmatrix} 32 & 32 \\ 32 & 32 \end{vmatrix}$

since n/2n/2 &matrix can be can be added in Cn for some constant C, The overall computing time T(n) of the resulting divide and conquer algorithm is given by the sequence.

$$T(n)=\begin{cases} b & n<=2 \text{ a \&b are} \\ 8T(n/2)+cn^2 & n>2 \text{ constant} \end{cases}$$

That is T(n)=O(n^3)

* Matrix multiplication are more expensive then the matrix addition O(n^3).We can attempt to reformulate the equation for Cij so as to have fewer multiplication and possibly more addition .

- Stressen has discovered a way to compute the Cij of equation (2) using only 7 multiplication and 18 addition or subtraction.
- Strassen's formula are

P= (A11+A12)(B11+B22)
Q= (A12+A22)B11
R= A11(B12-B22)
S= A22(B21-B11)
T= (A11+A12)B22
U= (A21-A11)(B11+B12)
V= (A12-A22)(B21+B22)

C11=P+S-T+V
C!2=R+t
C21=Q+T
C22=P+R-Q+V

UNIT-III

GREEDY METHOD:

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

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

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

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

Greedy algorithms neither postpone nor revise the decisions (ie., no back tracking).
Example: Kruskal's minimal spanning tree. Select an edge from a sorted list, check, decide, and never visit it again.

Application of Greedy Method:

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

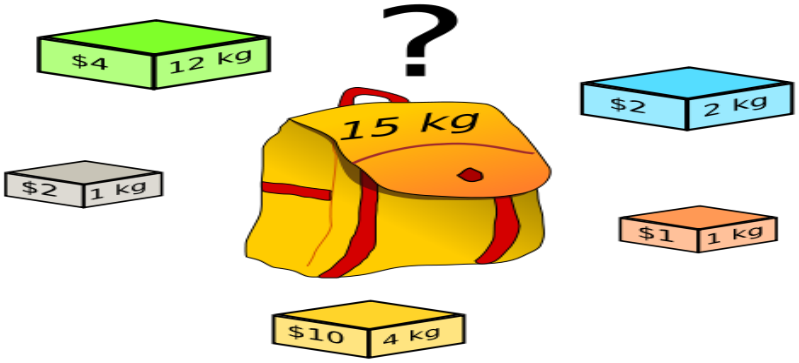
Algorithm for Greedy method

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

Selection → Function, that selects an input from a[] and removes it. The selected input's value is assigned to x.
Feasible → Boolean-valued function that determines whether x can be included into the solution vector.
Union → function that combines x with solution and updates the objective function.

Knapsack problem

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



There are two versions of the problems

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

Solutions to knapsack problems

- **Brute-force approach**:-Solve the problem with a straight farward algorithm
- **Greedy Algorithm**:- Keep taking most valuable items until maximum weight is reached or taking the largest value of eac item by calculating v_i=$value_i$/$Size_i$
- **Dynamic Programming**:- Solve each sub problem once and store their solutions in an array.
- **0/1 knapsack problem:**

- Let there be n items, z_1 to z_n where z_i has a value v_i and weight w_i. The maximum weight that we can carry in the bag is *W*. It is common to assume that all values and weights are nonnegative. To simplify the representation, we also assume that the items are listed in increasing order of weight.
- Maximize $\sum_{i=1}^{n} v_i x_i$ subject to $\sum_{i=1}^{n} w_i x_i \leqslant W, \qquad x_i \in \{0,1\}$
- Maximize the sum of the values of the items in the knapsack so that the sum of the weights must be less than the knapsack's capacity.

Greedy algorithm for knapsack

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

- **Ex**: **-** Consider 3 objects whose profits and weights are defined as
- (P_1, P_2, P_3) = (25, 24, 15)
- $W_1, W_2, W_3)$ = (18, 15, 10)
- n=3→number of objects
- m=20→Bag capacity
-
- Consider a knapsack of capacity 20. Determine the optimum strategy for placing the objects in to the knapsack. The problem can be solved by the greedy approach where in the inputs are arranged according to selection process (greedy strategy) and solve the problem in stages. The various greedy strategies for the problem could be as follows.

(x_1, x_2, x_3)	**$\sum x_i w_i$**	**$\sum x_i p_i$**

(1, 2/15, 0)	18x1+$\frac{2}{15}$x15 = 20	25x1+$\frac{2}{15}$x 24 = 28.2
(0, 2/3, 1)	$\frac{2}{3}$x15+10x1= 20	$\frac{2}{3}$x 24 +15x1 = 31
(0, 1, ½)	1x15+$\frac{1}{2}$x10 = 20	1x24+$\frac{1}{2}$x15 = 31.5
(½, ⅓, ¼)	½ x 18+⅓ x15+ ¼ x10 = 16. 5	½ x 25+⅓ x24+ ¼ x15 = 12.5+8+3.75 = 24.25

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

- **Job Sequence with Deadline:**
- There is set of n-jobs. For any job i, is a integer deadling di≥0 and profit Pi>0, the profit Pi is earned iff the job completed by its deadline.
- To complete a job one had to process the job on a machine for one unit of time. Only one machine is available for processing jobs.
- A feasible solution for this problem is a subset J of jobs such that each job in this subset can be completed by its deadline.
- The value of a feasible solution J is the sum of the profits of the jobs in J, i.e., $\sum_{i \in j} \mathbf{P_i}$
- An optimal solution is a feasible solution with maximum value.
- The problem involves identification of a subset of jobs which can be completed by its deadline. Therefore the problem suites the subset methodology and can be solved by the greedy method.
- **Ex: -** Obtain the optimal sequence for the following jobs.
- j_1 j_2 j_3 j_4
- (P_1, P_2, P_3, P_4) = (100, 10, 15, 27)
-
- (d_1, d_2, d_3, d_4) = (2, 1, 2, 1)
- n =4

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

- In the example solution ‘3’ is the optimal. In this solution only jobs 1&4 are processed and the value is 127. These jobs must be processed in the order j4 followed by j1. the process of job 4 begins at time 0 and ends at time 1. And the processing of job 1 begins at time 1 and ends at time2. Therefore both the jobs are completed within their deadlines. The optimization measure for determining the next job to be selected in to the solution is according to the profit. The next job to include is that which increases ∑pi the most, subject to the constraint that the resulting “j” is the feasible solution. Therefore the greedy strategy is to consider the jobs in decreasing order of profits.
- The greedy algorithm is used to obtain an optimal solution.
- We must formulate an optimization measure to determine how the next job is chosen.

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

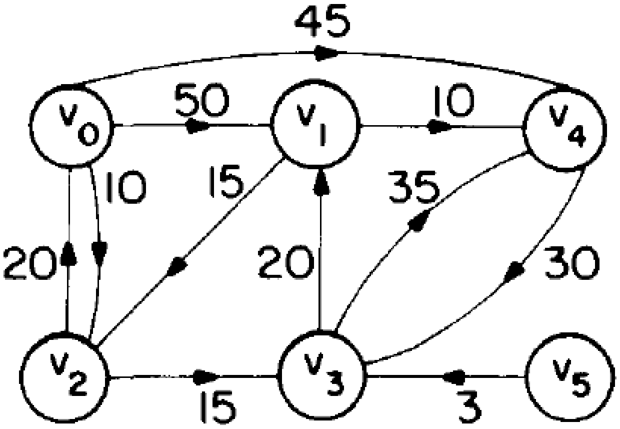
- Note: The size of sub set j must be less than equal to maximum deadline in given list.

<u>Single Source Shortest Paths:</u>

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

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

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



	Path	*Length*
1)	$v_0 v_2$	10
2)	$v_0 v_2 v_3$	25
3)	$v_0 v_2 v_3 v_1$	45
4)	$v_0 v_4$	45

Graph and shortest paths from v_0 to all destinations

- Consider the above directed graph, if node 1 is the source vertex, then shortest path from 1 to 2 is 1,4,5,2. The length is 10+15+20=45.
- To formulate a greedy based algorithm to generate the shortest paths, we must conceive of a multistage solution to the problem and also of an optimization measure.
- This is possible by building the shortest paths one by one.

- As an optimization measure we can use the sum of the lengths of all paths so far generated.

- If we have already constructed ‘i’ shortest paths, then using this optimization measure, the next path to be constructed should be the next shortest minimum length path.

- The greedy way to generate the shortest paths from Vo to the remaining vertices is to generate these paths in non-decreasing order of path length.

- For this 1st, a shortest path of the nearest vertex is generated. Then a shortest path to the 2nd nearest vertex is generated and so on.

Algorithm for finding Shortest Path

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

<u>Minimum Cost Spanning Tree:</u>

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

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

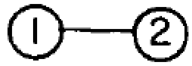
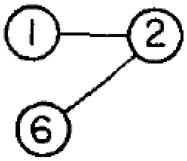
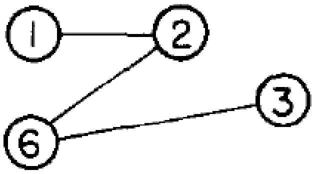
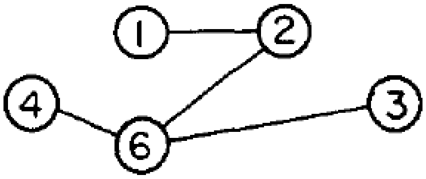
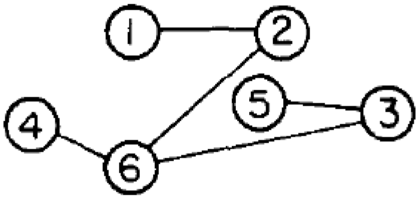
Minimum cost spanning tree: For a given graph 'G' there can be more than one spanning tree. If weights are assigned to the edges of 'G' then the spanning tree which has the minimum cost of edges is called as minimal spanning tree.

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

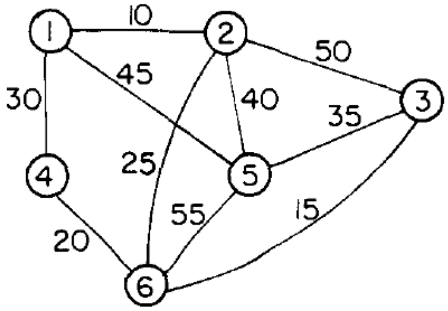
There are two basic algorithms for finding minimum-cost spanning trees, and both are greedy algorithms

→Prim's Algorithm
→Kruskal's Algorithm

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

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

Stages in Prim's Algorithm



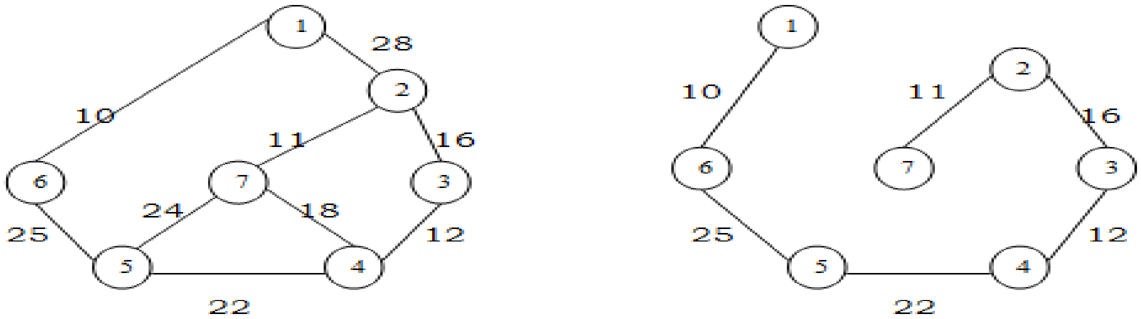
PRIM'S ALGORITHM: -

i) Select an edge with minimum cost and include in to the spanning tree.

ii) Among all the edges which are adjacent with the selected edge, select the one with minimum cost.

iii) Repeat step 2 until 'n' vertices and (n-1) edges are been included. And the sub graph obtained does not contain any cycles.

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



Prim's minimum spanning tree algorithm

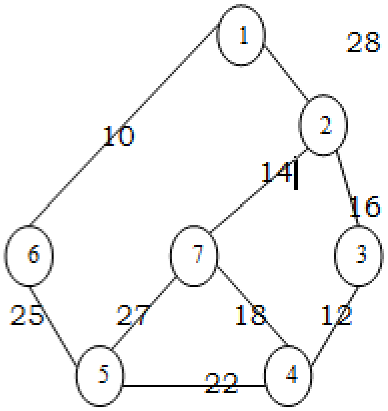
```
Algorithm Prim (E, cost, n,t)
// E is the set of edges in G. Cost (1:n, 1:n) is the
// Cost adjacency matrix of an n vertex graph such that
// Cost (i,j) is either a positive real no. or ∞ if no edge (i,j) exists.
//A minimum spanning tree is computed and
//Stored in the array T(1:n-1, 2).
//(t (i,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

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

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

n = {1,2,3,4,5,6,7)

$$\begin{array}{c|ccccccc} \text{Cost} & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ \hline 1 & \alpha & 28 & \alpha & \alpha & \alpha & 10 & \alpha \\ 2 & 28 & \alpha & 16 & \alpha & \alpha & \alpha & 14 \\ 3 & \alpha & 10 & \alpha & 12 & \alpha & \alpha & \alpha \\ 4 & \alpha & \alpha & 12 & \alpha & 22 & \alpha & 18 \\ 5 & \alpha & \alpha & \alpha & 22 & \alpha & 25 & 24 \\ 6 & 10 & \alpha & \alpha & \alpha & 25 & \alpha & \alpha \\ 7 & \alpha & 14 & \alpha & 18 & 24 & \alpha & \alpha \end{array}$$

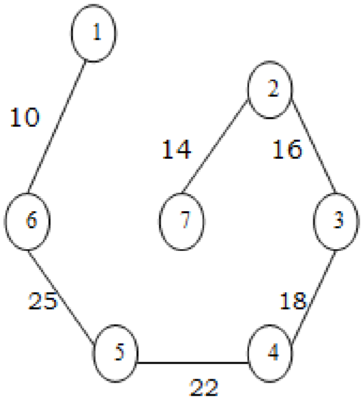


$$\begin{array}{c|cccccc} & 1 & 2 & 3 & 4 & 5 & 6 \\ \hline 1 & 1 & 6 & 5 & 4 & 3 & 2 \\ 2 & 6 & 5 & 4 & 3 & 2 & 7 \end{array}$$

Start Vertex

Ending Vertex

Edges of spanning tree



i) The algorithm will start with a tree that includes only minimum cost edge of G. Then edges are added to this tree one by one.

ii) The next edge (i,j) to be added is such that i is a vertex which is already included in the treed and j is a vertex not yet included in the tree and cost of i,j is minimum among all edges adjacent to ‘i’.

iii) With each vertex ‘j’ next yet included in the tree, we assign a value near ‘j’. The value near ‘j’ represents a vertex in the tree such that cost (j, near (j)) is minimum among all choices for near (j)

iv) We define near (j):= 0 for all the vertices ‘j’ that are already in the tree.

v) The next edge to include is defined by the vertex ‘j’ such that (near (j)) $\neq$ 0 and cost of (j, near (j)) is minimum.

Analysis: -

The time required by the prince algorithm is directly proportional to the no/: of vertices. If a graph ‘G’ has ‘n’ vertices then the time required by prim’s algorithm is **0(n^2)**

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

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

i) All the edges are considered one by one in that order and deleted from the graph and are included in to the spanning tree.

ii) At every stage an edge is included; the sub-graph at a stage need not be a tree. Infect it is a forest.

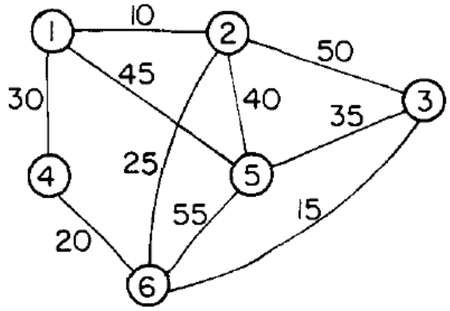
iii) At the end if we include ‘n’ vertices and n-1 edges without forming cycles then we get a single connected component without any cycles i.e. a tree with minimum cost.

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

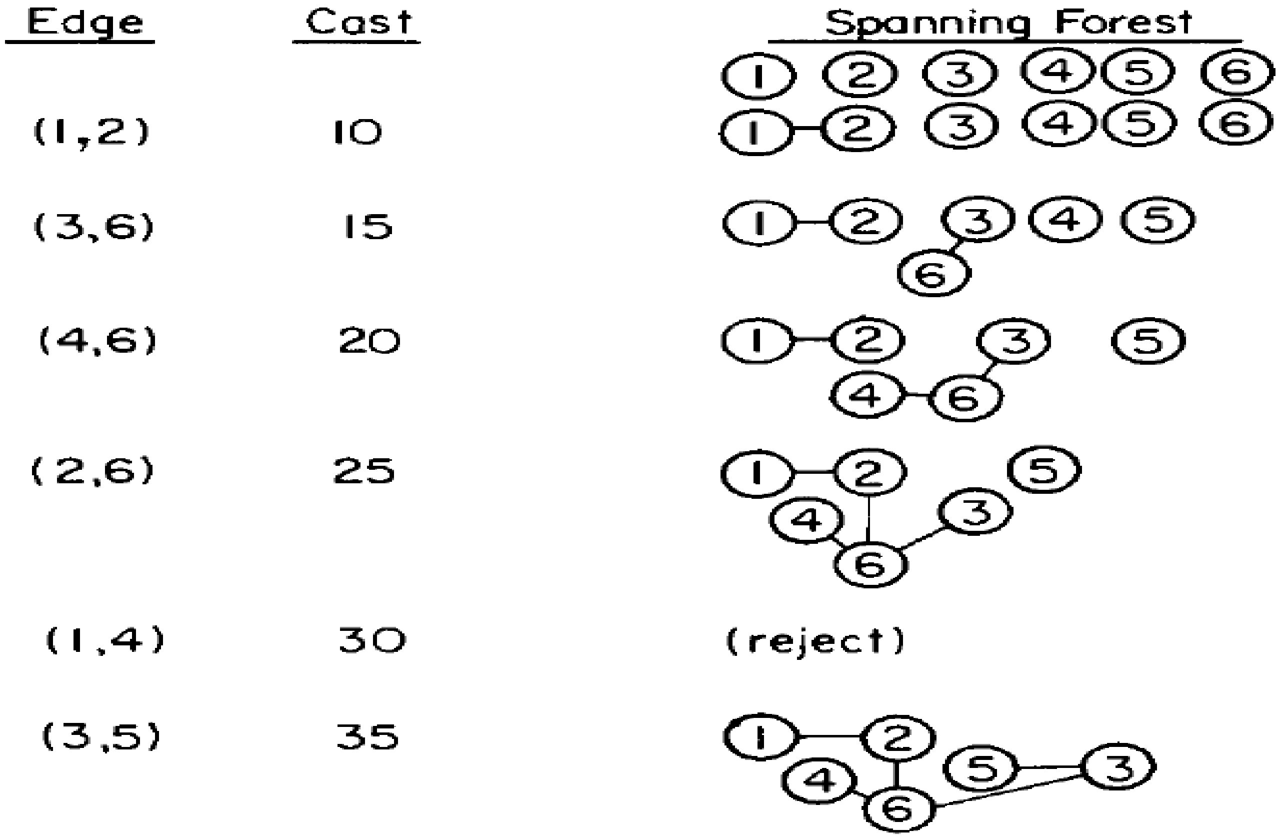
Kruskal minimum spanning tree algorithm

```
Algorithm Kruskal (E, cost, n,t)
//E is the set of edges in G. ‘G’ has ‘n’ vertices
//Cost {u,v} is the cost of edge (u,v) t is the set
//of edges in the minimum cost spanning tree
//The final cost is returned
{ construct a heap out of the edge costs using heapify;
        for i:= 1 to n do parent (i):= -1 // place in different sets
//each vertex is in different set           {1} {1} {3}
```

```
          i: = 0; min cost: = 0.0;
          While (i<n-1) and (heap not empty))do
{
Delete a minimum cost edge (u,v) from the heaps; and reheapify using adjust;
j:= find (u); k:=find (v);
if (j≠k) then
{  i: = 1+1;
   + (i,1)=u; + (i, 2)=v;
    mincost: = mincost+cost(u,v);
    Union (j,k);
    }
}
if (i≠n-1) then write ("No spanning tree");
   else return mincost;
}
```



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



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

Stages in Kruskal's algorithm

Analysis: - If the no/: of edges in the graph is given by /E/ then the time for Kruskals algorithm is given by 0 (|E| log |E|).

UNIT-IV

DYNAMIC PROGRAMMING

Introduction to Dynamic programming; a method for solving optimization problems.
Dynamic programming vs. Divide and Conquer
A few examples of Dynamic programming

- the 0-1 Knapsack Problem
- Chain Matrix Multiplication
- All Pairs Shortest Path
- The Floyd Warshall Algorithm: Improved All Pairs Shortest Path

Recalling Divide-and-Conquer

1. Partition the problem into particular subproblems.
2. Solve the subproblems.
3. Combine the solutions to solve the original one.

5.1 MULTI STAGE GRAPHS

A multistage graph G = (V, E) is a directed graph in which the vertices are partitioned into k ≥ 2 disjoint sets Vi, 1 ≤ i ≤ k. In addition, if <u, v> is an edge in E, then u E Vi and v E Vi+1 for some i, 1 ≤ 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 Vi 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 noticing that every s to t path is the result of a sequence of k – 2 decisions. The ith decision involves determining which vertex in $_{vi+1,}$ $1 \leq i \leq k - 2$, 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:

```
cost (i, j) = min {c (j, l) + cost (i + 1, l)}
                  l c Vi + 1
                 <j, l> c 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.
{  co
      s
      t

      [
      n
      ]

      :
      =

      0
      .
      0
      ;

      for j:= n - 1 to 1 step – 1 do
      {                                         // 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
        ]

        +

        c
        o
        s
        t

        [
        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 Vi. 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)} l e Vi - 1
           <l, j> e E
```

Algorithm Bgraph (G, k, n, p)

```
// Same function as Fgraph {
        Bcost [1] := 0.0; for j := 2 to n do { / / 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 cost path p [1] := 1; p [k] := n;
        for j:= k - 1 to 2 do p [j] := d [p [j + 1]];
}
```

Complexity Analysis:

The complexity analysis of the algorithm is fairly straightforward. Here, if G has ~E~ edges, then the time for the first for loop is CJ (V~ +~E).

EXAMPLE 1:

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.

FORWARD APPROACH:

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 c Vi + 1
            <j, l> c E
```

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,

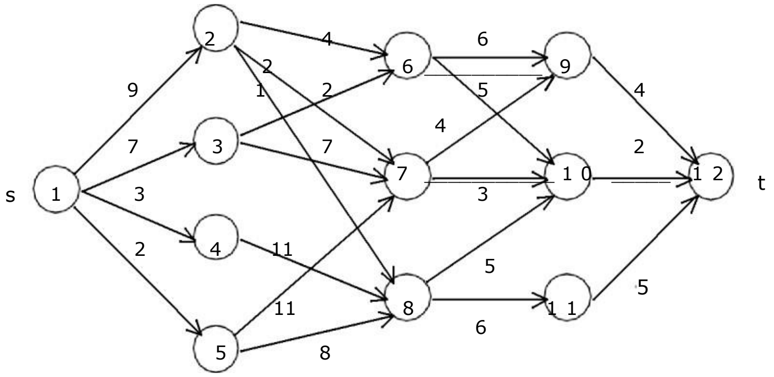
cost (2, 2) = min{c (2, 6) + cost (3, 6), c (2, 7) + cost (3, 7), c (2, 8) + cost (3, 8)} = min {4 + cost (3, 6), 2 + cost (3, 7), 1 + cost (3, 8)}

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

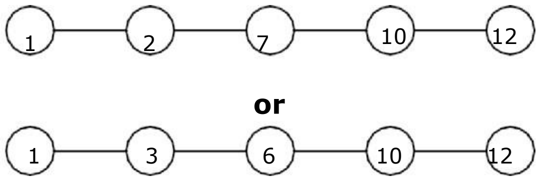
Therefore, cost (3, 6) = min {6 + 4, 5 + 2} = 7



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)

The path is,

The path is



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, 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 is 16.

BACKWARD APPROACH:

We use the following equation to find the minimum cost path from t to s: Bcost (i, J) = min

{Bcost (i – 1, l) + c (l, J)}

l c vi – 1
<l, j> c E

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

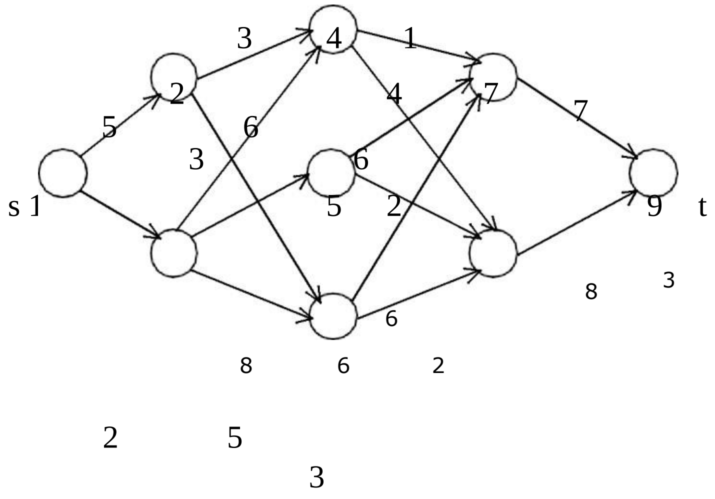
+ 5, 11 + 3, 10 + 5} = min {14, 14, 15) = 14

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. **EXAMPLE 2:**

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.



SOLUTION:

FORWARD APPROACH:

cost (i, J) = min {c (j, l) + cost (i + 1, l)}
l c Vi + 1
<J, l> EE

cost (1, 1) = min {c (1, 2) + cost (2, 2), c (1, 3) + cost (2, 3)} = min {5 + cost (2, 2), 2 + cost (2, 3)}

cost (2, 2) = min {c (2, 4) + cost (3, 4), c (2, 6) + cost (3, 6)} = min {3+ cost (3, 4), 3 + cost (3, 6)}

cost (3, 4) = min {c (4, 7) + cost (4, 7), c (4, 8) + cost (4, 8)} = min {(1 + cost (4, 7), 4 + cost (4, 8)}

cost (4, 7) = min {c (7, 9) + cost (5, 9)} = min {7 + 0) = 7 cost (4, 8)

= min {c (8, 9) + cost (5, 9)} = 3

Therefore, cost (3, 4) = min {8, 7} = 7

cost (3, 6) = min {c (6, 7) + cost (4, 7), c (6, 8) + cost (4, 8)}
Therefore, cost (2, 2) = min {10, 8} = 8 cost (2, 3) = min {c (3, 4)

+ cost (3, 4), c (3, 5) + cost (3, 5), c (3, 6) + cost (3,6)} cost (3, 5)

= min {c (5, 7) + cost (4, 7), c (5, 8) + cost (4, 8)}= min {6 + 7, 2

+ 3} = 5

104

Therefore, cost (2, 3) = min {13, 10, 13} = 10

cost (1, 1) = min {5 + 8, 2 + 10} = min {13, 12} = 12

BACKWARD APPROACH:

Bcost (i, J) = min {Bcost (i
– 1, l) = c (l,
J)} l E vi – 1
<l ,j>E E

Bcost (5, 9) = min {Bcost (4, 7) + c (7, 9), Bcost (4, 8) + c (8, 9)}
= min {Bcost (4, 7) + 7, Bcost (4, 8) + 3}

Bcost (4, 7) = min {Bcost (3, 4) + c (4, 7), Bcost (3, 5) + c (5, 7),
Bcost (3, 6) + c (6, 7)}
= min {Bcost (3, 4) + 1, Bcost (3, 5) + 6, Bcost (3, 6) + 6}

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

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

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

min {0 + 2} = 2

Therefore, Bcost (3, 4) = min {5 + 3, 2 + 6} = min {8, 8} = 8

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

Bcost (3, 6) = min {Bcost (2, 2) + c (2, 6), Bcost (2, 3) +
c (3, 6)} = min {5 + 5, 2 + 8} = 10

Therefore, Bcost (4, 7) = min {8 + 1, 7 + 6, 10 + 6} = 9

Bcost (4, 8) = min {Bcost (3, 4) + c (4, 8), Bcost (3, 5) + c (5, 8), Bcost
(3, 6) + c (6, 8)}
= min {8 + 4, 7 + 2, 10 + 2} = 9

Therefore, Bcost (5, 9) = min {9 + 7, 9 + 3} = 12 **All**

pairs shortest paths

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

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

The all pairs shortest path problem is to determine a matrix A such that A (i, j) is the length of a shortest path from i to j. The matrix A can be obtained by solving n single-source

105

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:

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

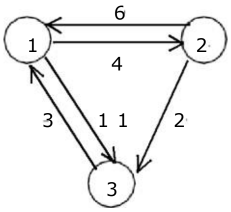
Algorithm All Paths (Cost, A, n)

```
// 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).

Example 1:

Given a weighted digraph G = (V, E) with weight. Determine the length of the shortest path between all pairs of vertices in G. Here we assume that there are no cycles with zero or negative cost.



0 ~r 0 4 112 ~

 Cost adjacency matrix (A) = ~6 0 ~

~L3 ~

General formula: min {A^{k-1} (i, k) + A^{k-1} (k, j)}, c (i, j)}
1<k<n

Solve the problem for different values of k = 1,

2 and 3 **Step 1**: Solving the equation for, k = 1;

A1 (1, 1) = min {(A^o (1, 1) + A^o (1, 1)), c (1, 1)} = min {0 + 0, 0} = 0 A1 (1, 2) = min {(A^o (1, 1) + A^o (1, 2)), c (1, 2)} = min {(0 + 4), 4} = 4

A1 (1, 3) = min {(A^o (1, 1) + A^o (1, 3)), c (1, 3)} = min {(0 + 11), 11} = 11 A1 (2, 1) = min {(A^o (2, 1) + A^o (1, 1)), c (2, 1)} = min {(6 + 0), 6} = 6

A1 (2, 2) = min {(A^o (2, 1) + A^o (1, 2)), c (2, 2)} = min {(6 + 4), 0)} = 0 A1 (2, 3) = min {(A^o (2, 1) + A^o (1, 3)), c (2, 3)} = min {(6 + 11), 2} = 2 A1 (3, 1) = min {(A^o (3, 1) + A^o (1, 1)), c (3, 1)} = min {(3 + 0), 3} = 3 A1 (3, 2) = min
{(A^o (3, 1) + A^o (1, 2)), c (3, 2)} = min {(3 + 4), oc} = 7 A1 (3, 3) = min {(A^o (3, 1) + A^o (1, 3)), c (3, 3)} = min {(3 + 11), 0} = 0

A$_{(1)}$ = ~0 4
~ 11
0
~6 ~
7
~L3 2 ~
0 ~U

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} + A^1 = 0

$_1$ (1, 2) (2, 2), c (1, 2)} = min {(4 + 0), 4} + A

A_2 (1, 2) = min {(A ^{1} (2, = 4

A_2 (1,

3) = min {(A^1 (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,

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

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

$A_{(2)}$ = ~~0 4 62~ 1

~6 0 ~

~L3 0 ~~

7

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

A3 (1, 1) = min {A^2 (1, 3) + A^2 (3, 1), c (1, 1)} = min {(6 + 3), 0} = 0

A3 (1, 2) = min {A^2 (1, 3) + A^2 (3, 2), c (1, 2)} = min {(6 + 7), 4} = 4

A3 (1, 3) = min {A^2 (1, 3) + A^2 (3, 3), c (1, 3)} = min {(6 + 0), 6} = 6 A3

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

A3 (2, 2) = min {A^2 (2, 3) + A^2 (3, 2), c (2, 2)} = min {(2 + 7), 0} = 0

A3 (2, 3) = min {A^2 (2, 3) + A^2 (3, 3), c (2, 3)} = min {(2 + 0), 2} = 2 A3

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

A3 (3, 2) = min {A^2 (3, 3) + A^2 (3, 2), c (3, 2)} = min {(0 + 7), 7} = 7

107

A3 (3, 3) = min {A^2 (3, 3) + A^2 (3, 3), c (3, 3)} = min {(0 + 0), 0} = 0

$A_{(3)}$ = ~ 0 4 6 ~

0 ~

~ ~ 7 2~

5~~ 3 0

~]

TRAVELLING SALESPERSON PROBLEM

Let G = (V, E) be a directed graph with edge costs Cij. The variable cij is defined such that cij > 0 for all I and $_j$ and cij = a if < i, j> o 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 vertex 1.

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 follows that:

g(1, V - {1 }) = 2 ~ *k* ~ *n* ~*c*1*k* ~ *g* ~ *k*, *V* ~ ~ 1, *k* ~~ -- 1

~

min

-- 2

Generalizing equation 1, we obtain (for i o S)

g (i, S) = min{*ci j j* E*S*

The Equation can be solved for g (1, V – 1}) if we know g (k, V – {1, k}) for all choices of k.

Complexity Analysis:

For each value of |S| there + *g* (*i*, *S* - (*j*))) are n – 1 choices for i. The number of distinct sets S of

size k not including 1 and i is I *k* ~ ~ ~ . *n* - 2 ~ ~
~ ~

Hence, the total number of g (i, S)’s to be computed before computing g (1, V – {1}) is:

~ *n* - 2 ~
~ ~*n* ~1~ ~
~ *k* ~

k ~ 0 ~ ~

To calculate this sum, we use the binominal theorem:

[((*n* - 2) ((*n* - 2) ((*n* - 2) ((*n* - 2)1 *n* - 1 (n – 1) 111 11+ ii iI+
ii iI+ - - - - ~ ~~
~~ 0) ~ 1) ~ 2) ~(*n* ~ ^{2})~~

According to the binominal theorem:

[((*n* - 2) ((*n* - 2) ((*n* - 2 ((*n* - 2)1
il 11+ ii iI+ ii ~~~ ~ ~ ~ ~ ~ ~~ ~~~ = 2n - 2
~~ 0 ~ ~ 1 ~ ~ 2 ~ ~(*n* - 2))]

Therefore,

n - 1 ~ *n* _ 2'

~ (*n* _ 1~ ~~ ~*k* = (*n* - 1) 2*n* ~ 2

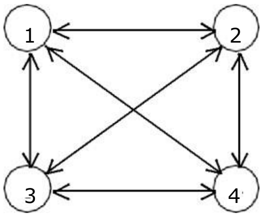
k ~ 0 ~ ~

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.

Example 1:

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



~

r0 10 15 2

10 ~

The cost adjacency matrix = ~

0 9 12~

5 13 0 ~ ~6 8 9 0 1]

~

Let us start the tour from vertex 1:

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

More generally writing:

g (i, s) = min {cij + g (J, s – {J})} - (2)

Clearly, g (i, T) = ci1 , 1 ≤ i ≤ n. So,

g (2, T) = C21

= 5 g (3, T) = C31 = 6

g (4, ~) = C41 = 8

Using equation – (2)

we obtain:

g (1, {2, 3, 4}) = min {c12 + g (2, {3,
4}, c13 + g (3, {2, 4}), c14 + g (4, {2, 3})}

g (2, {3, 4}) = min {c23 + g (3, {4}), c24 + g (4, {3})}
= min {9 + g (3, {4}), 10 + g (4, {3})} g (3,

{4}) = min {c34 + g (4, T)} = 12 + 8 = 20 g

(4, {3}) = min {c43 + g (3, ~)} = 9 + 6 = 15

Therefore, g (2, {3, 4}) = min {9 + 20, 10 +

15} = min {29, 25} = 25 g (3, {2, 4}) = min

{(c32 + g (2, {4}), (c34 + g (4, {2})} g (2, {4})

= min {c24 + g (4, T)} = 10 + 8 = 18

g (4, {2}) = min {c42 + g (2, ~)} = 8 + 5 = 13

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

25} = 25 g (4, {2, 3}) = min {c42 + g (2, {3}), c43 + g (3,

{2})}

g (2, {3}) = min {c23 + g (3, ~} = 9 + 6

= 15 g (3, {2}) = min {c32 + g (2, T}

= 13 + 5 = 18

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

g (1, {2, 3, 4}) = min {c12 + g (2, {3, 4}), c13 + g (3, {2, 4}), c14 + g (4, {2, 3})}
= min {10 + 25, 15 + 25, 20 + 23} = min {35, 40, 43} = 35

The optimal tour for the graph has length = 35

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

OPTIMAL BINARY SEARCH TREE

Let us assume that the given set of identifiers is {a1, . . . , an} with a1 < a2 < . . . . < an. Let p (i) be the probability with which we search for ai. Let q (i) be the probability that the identifier x being searched for is such that ai < x < ai+1, 0 ≤ i ≤ n (assume a0 = - ~ and an+1 = +oc). 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 'ai' is placed at depth 'di', then we want to minimize: *n*

~ *Pi* (1 +

di) . *i* ~1

Let P (i) be the probability with which we shall be searching for 'ai'. 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 'ai' is:

$P\ (i) * level\ (ai\ )$.

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

Q (i) * level ((Ei) - 1)

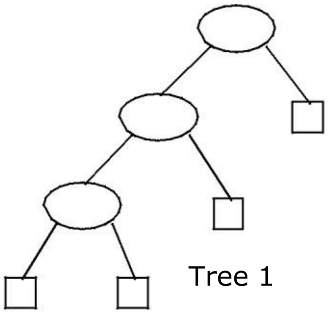
110

The expected cost of binary search tree is: n n

~ $P(i) * level\ (ai) +$ ~ $Q\ (i) * level\ ((Ei\ ) - 1)$

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

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

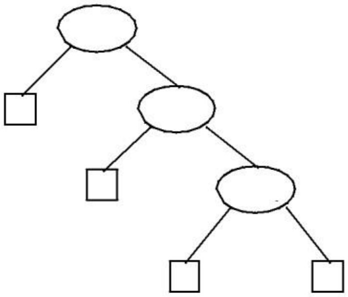


Tree 1

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

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

Example 1: The possible binary search trees for the identifier set



(a1, a2, a3) = (do, if, stop) are as follows. Given the equal probabilities p (i) = Q (i) = 1/7 for all i, we have:

stop

if

do

Tree 2

do

if

stop

Tree 3

(1 x 1 + 1 x 2 + 1 x 3~

Cost (tree # 1) = ~ +

~ 7 7 7)

1 + 2 + 3 1 + 2 + 3 + 3 6 + 9 15

Cost (tree # 3) = ~ ~~ 1 7 x 1 + 17 x 2 + 1 7x 3~ ~ +) (~~ 1 7 x 1 + 17 x 2 +

17 x 3 + 1 7x 3~ ~)

= 1 +7 2 + 3 + 1 + 2 +7 3 + 3 ~ 6 +7(19 x ~1 15 +7 1

Cost (tree # 4) = ~~~ 1 7 x 1 + 17 x 2 ~ 1 7x 3~ ~ ~) ~~ 7

7 x 2 + 17 x 3 + 1 7x 3~ ~)

(1 x 1 +

~ 7

= ~~1 + 7 2 + 3~~ -l- $\frac{1 + 2 + 3 + 3}{7}$ ~$\underline{6\ +}$7$\underline{\ 9}$ ~15

= 7 + 7 ~ 7 ~ 7

(1 *x* 1 + 1 1 *x* 2~ (1 *x* 2 + 1 *x* 2 + 1 *x* 2 + 1 *x* 2~

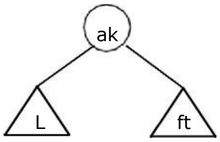
Cost (tree # 2) = ~~ 7 $\overline{7}$ *x* 2 + $\overline{7}$) + ~~ 7 $\overline{7}$ $\overline{7}$ $\overline{7}$)

= $\underline{1 + 2 + 2}$ $\underline{2 + 2 + 2 + 2}$ $\underline{5 + 8}$ $\underline{13}$

7 + 7 ~ 7 ~ 7

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 ai's be arraigned to the root node at 'T'. If we choose 'ak' then is clear that the internal nodes for a1, a2, ... ak-1 as well as the external nodes for the classes Eo, E1, ... Ek-1 will lie in the left sub tree, L, of the root. The remaining nodes will be in the right subtree, ft. The structure of an optimal binary search tree is:



K *K*

Cost (L) = *P*(*i*)* *level* (*a_i*) + *Q*(*i*)* (*level* (*E_i*) - 1)

i ~ 1 *i* ~ 0

n *n*

$$\text{Cost (ft)} = \sum_{i \sim K} P(i)* level\ (a_i) + \sum_{i \sim K} Q(i)* (level\ (E_i) - 1)$$

The C (i, J) can be computed as:

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

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

$$\text{Where } W\ (i, J) = P\ (J) + Q\ (J) + w\ (i, J-1) \qquad \text{--} \qquad (2)$$

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 so on.

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) are obtained.

The results are tabulated to recover the actual tree.

Example 1:

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				

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

Solving for C (0, n):
First, computing all C (i, j) such that j - i = 1; j = i + 1 and as 0 ≤ 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 ft
(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 ft
(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) = 3 + 1 + 8 = 12

C (0, 2) = W (0, 2) + min {(C (0, 0) + C (1, 2)), (C (0, 1) + C (2, 2))} = 12
+ min {(0 + 7, 8 + 0)} = 19
ft (0, 2) = 1
Next, with i = 1; so j = 3; as i < k ≤ j, so the possible value for k = 2 and 3.
W (1, 3) = P (3) + Q (3) + W (1, 2) = 1 + 1+ 7 = 9
C (1, 3) = W (1, 3) + min {[C (1, 1) + C (2, 3)], [C (1, 3) 2) + C (3, 3)]}
= W (1, + min {(0 + 3), (7 + 0)} = 9 + 3 = 12
ft (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
ft (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 and 3.

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

C (0, 3) W (0, 3) + min {[C (0, 0) + C (1, 3)], [C (0, 1) + C (2, 3)],
[C (0, 2) + C (3, = 3)]}
14 + min {(0 + 12), (8 + 3), (19 = 2 + 0)} = 14 + 11 = 25
ft (0, 3)

Start with i = 1; so j = 4; as i < k ≤ j, so the possible values for k = 2, 3 and 4.
W (1, 4) = P (4) + Q (4) + W (1, 3) = 1 + 1 + 9 = 11 = W 2)
C (1, 4) (1, 4) + min {[C (1, 1) + C (2, 4)], [C (1, + C (3, 4)],

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

ft (1, 4)

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 and 4.

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

C (0, 4) = W (0, 4) + min {[C (0, 0) + C (1, 4)], [C (0, 1) + C (2, 4)],
[C (0, 2) + C (3, 4)], [C (0, 3) + C (4, 4)]}
= 16 + min [0 + 19, 8 + 8, 19+3, 25+0] = 16 + 16 = 32 ft (0,
4) = 2

From the table we see that C (0, 4) = 32 is the minimum cost of a binary search tree for (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.

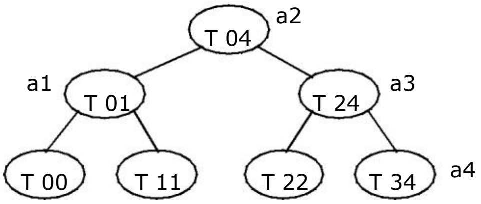
The left and right sub trees for 'T01' are 'T00' and 'T11' respectively. The root of T01 is 'a1'

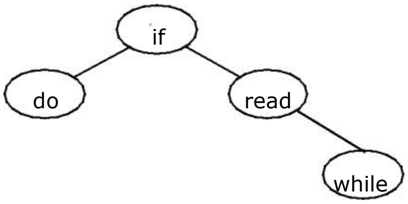
The left and right sub trees for T24 are T22 and T34 respectively.

The root of T24 is 'a3'.

The root of T22 is null

The root of T34 is a4.





Example 2:

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 ≤ 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) = 4 + 3 + 2 = 9

C (0, 1) = W (0, 1) + min {C (0, 0) + C (1, 1)} = 9 + [(0 + 0)] = 9 ft
(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 ft
(1, 2) = 2

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

+ 1 = 3
W (2, 3) = P (3) + Q (3) + W (2, 2) = 1 + 1 3)} = 3 + [(0 + 0)] = 3 C (2,
3) = W (2, 3) + min {C (2, 2) + C (3, ft (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 ft
(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 ft (0, 2) = 1
Next, with i = 1; so j = 3; as i < k ≤ j, so the possible value for k = 2 and 3.
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
ft (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
ft (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 and 3.

W (0, 3) = P (3) + Q (3) + W (0, 2) = 1 + 1 + 12 = 14
C (0, 3) = W (0, 3) + min {[C (0, 0) + C (1, 3)], [C (0, 1) + C (2, 3)], [C (0, 2) + C (3, 3)]}
= 14 + min {(0 + 11), (9 + 3), (18 + 0)} = 14 + 11 = 25 ft (0, 3) = 1

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

W (1, 4) = P (4) + Q (4) + W (1, 3) = 1 + 1 + 8 = 10 = W 2)
C (1, 4) (1, 4) + min {[C (1, 1) + C (2, 4)], [C (1, + C (3, 4)],

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

ft (1, 4)

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 and 4.

W (0, 4) = P (4) + Q (4) + W (0, 3) = 1 + 1 + 14 = 16
C (0, 4) = W (0, 4) + min {[C (0, 0) + C (1, 4)], [C (0, 1) + C (2, 4)], [C (0, 2) + C (3, 4)], [C (0, 3) + C (4, 4)]}

= 16 + min [0 + 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.

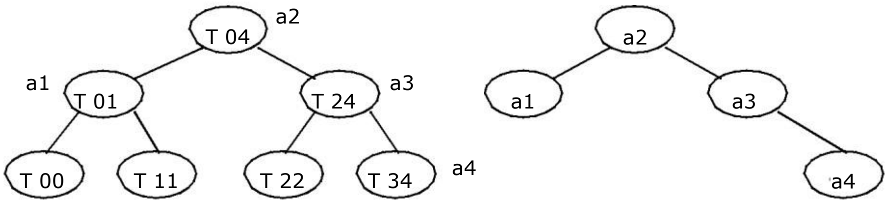
The left and right sub trees for 'T01' are 'T00' and 'T11' respectively. The root of T01 is 'a1'

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 wi and a positive value Vi. 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 x1, x2, ... , xn. A decision on variable xi involves determining which of the values 0 or 1 is to be assigned to it. Let us assume that

decisions on the xi are made in the order xn, xn-1, ...x1. Following a decision on xn, we may be in one of two possible states: the capacity remaining in m – wn and a profit of pn has accrued. It is clear that the remaining decisions xn-1, . . . , x1 must be optimal with respect to the problem state resulting from the decision on xn. Otherwise, xn,. . . . , x1 will not be optimal. Hence, the principal of optimality holds.

$$Fn (m) = \max \{fn-1 (m), fn-1 (m - wn) + pn\} \qquad -- \qquad 1$$

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

$$Fi (y) = \max \{fi-1 (y), fi-1 (y - wi) + pi\} \qquad -- \qquad 2$$

Equation-2 can be solved for fn (m) by beginning with the knowledge fo (y) = 0 for all y and fi (y) = - ~, y < 0. Then f1, f2, . . . fn can be successively computed using equation–2.

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

The fi (y) is an ascending step function; i.e., there are a finite number of y's, 0 = y1 < y2 < . . . . < yk, such that fi (y1) < fi (y2) < . . . . . < fi (yk); fi (y) = - ~ , y < y1; fi (y) = f (yk), y $\geq$ yk; and fi (y) = fi (yj), yj $\leq$ y $\leq$ yj+1. So, we need to compute only fi (yj), 1 $\leq$ j $\leq$ k. We use the ordered set S^i = {(f (yj), yj) | 1 $\leq$ j $\leq$ k} to represent fi (y). Each number of S^i is a pair (P, W), where P = fi (yj) and W = yj. Notice that S^0 = {(0, 0)}. We can compute S^{i+1} from Si by first computing:

$$Si\ 1 = \{(P, W) \mid (P – pi, W – wi)\ e\ S^i\}$$

Now, S^{i+1} can be computed by merging the pairs in S^i and Si 1 together. Note that if Si+1 contains two pairs (Pj, Wj) and (Pk, Wk) with the property that Pj $\leq$ Pk and Wj > Wk, then the pair (Pj, Wj) 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, (Pk, Wk) dominates (Pj, Wj).

Reliability Design

The problem is to design a system that is composed of several devices connected in series. Let ri be the reliability of device Di (that is ri is the probability that device i will function properly) then the reliability of the entire system is fT ri. Even if the individual devices are very reliable (the ri's are very close to one), the reliability of the system may not be very good. For example, if n = 10 and ri = 0.99, i $\leq$ i $\leq$ 10, then fT ri = .904.

Hence, it is desirable to duplicate devices. Multiply copies of the same device type are connected in parallel.

If stage i contains mi copies of device Di. Then the probability that all mi have a malfunction is (1 - ri) mi. Hence the reliability of stage i becomes 1 – (1 - r)^{mi}.

i

The reliability of stage 'i' is given by a function ~i (mi).

Our problem is to use device duplication. This maximization is to be carried out under a cost constraint. Let ci be the cost of each unit of device i and let c be the maximum allowable cost of the system being designed. We wish to solve:

Maximize ~ *qi* (*mi* ~
1 ≤ *i* < *n*
Subject to ~ *Ci mi* < *C*
1 ≤ *i* < *n*

mi ≥ 1 and interger, 1 ≤ i ≤ n

Assume each Ci > 0, each mi must be in the range 1 ≤ mi ≤ ui, where

~ ~ +*Ci* *n* ~
ui ~ ~ ~*C* ~ *C* ~ *Ci* ~ ~ /
IL k ~ ~*J* U
1 ~

The upper bound ui follows from the observation that mj ≥ 1

An optimal solution m1, m2 . . . . . mn is the
result of a sequence of decisions, one
decision for each mi.

~ q\$(*mJ*)

Let fi (x) represent the maximum value of 1 < *j* ≤ *i*

Subject to the constrains:

$$\underset{1 \leq j \leq i}{\tilde{}\; C_J\, m_J \sim x} \quad \text{and } 1 \leq m_j \leq u_J,\ 1 \leq j \leq i$$

The last decision made requires one to choose mn from {1, 2, 3, ... un}

Once a value of mn has been chosen, the remaining decisions must be such as to use the remaining funds C – Cn mn in an optimal way.
The principle of optimality holds on

$$f_n \sim C \sim \sim \max \{ On\ (m_n)\ fn\ \_\ 1\ (C - C_n\ m_n) \}\ 1 < m_n < u_n$$

for any fi (xi), i > 1, this equation generalizes to

$$f_n\,(x) = \max \{ci\ (mi)\ fi - 1\ (x - Ci\ mi)\}\ 1 < mi < ui$$

clearly, f0 (x) = 1 for all x, 0 ≤ x ≤ C and f (x) = -oo for all x < 0.

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

There is atmost one tuple for each different ‘x’, that result from a sequence of decisions on m1, m2, ... mn. The dominance rule (f1, x1) dominate (f2, x2) if f1 ≥ f2 and x1 ≤ x2. Hence, dominated tuples can be discarded from S^i.

Example 1:

Design a three stage system with device types D1, D2 and D3. The costs are $30, $15 and $20 respectively. The Cost of the system is to be no more than $105. The reliability of each device is 0.9, 0.8 and 0.5 respectively.

Solution:

We assume that if if stage I has mi devices of type i in parallel, then *0* i (mi) =1 – (1- ri)^{mi}

Since, we can assume each ci > 0, each mi must be in the range 1 ≤ mi ≤ ui. Where:

$$\sim u_i = \overset{\sim\,\sim}{\sim} \,|C + \overset{n}{C_i} \qquad \overset{\sim}{-\,C} \;\; /$$

~J Ci ~

IL k ~ ~ ~~

1 ~

Using the above equation compute u1, u2 and u3.

105+ 30- (30+15 + 20) 70

*u*1 = 30 = 30 = 2

105+15- (30+15 + 20) 55 = 3

*u*2 = 15 = 15

*u*3 = (30+15 + 20) 60 = 3

105+ 20-

= 20 20

We useS -* *i*:*stage number and J*: *no*. *of devices in stage i* = mi S°

= {f_o (*x*), *x*} *initially* f_o (*x*) = 1 *and x* = 0, *so*, S^o = {1, 0}

Compute S^1, S^2 and S^3 as follows:

S1 = depends on u1 value, as u1 = 2, so

*S*1 = {*S*1, S^1 }

1 2

S2 = depends on u2 value, as u2 = 3, so

*s*2 = {S^2 , S^2 , S^2 }

1 2 3

S3 = depends on u3 value, as u3 = 3, so

*S*3 = {S^3 , S^3 , S^3 }

1 2 3

Now find , 1 *S* (*x*), ~~*x*

*f*1 (*x*) = {*0*1 (1) f_o ~ ~, *0*1 (2) *f* 0 ()} With devices m1 = 1 and m2 = 2 Compute

*Ø*1 (1) and *Ø*1 (2) using the formula: *Øi* (*mi*)) = 1 - (1 - *ri*) *mi*

~~1~ ~ 1~ ~1 ~ *r* ~*m* 1 = 1 – (1 – 0.9)^1 = 0.9

11 1 ~(2) = 1- (1- 0.9) 2 = 0.99

S ~ ~11 *f* 1 ~*x*~, *x* ~ ~ ~ ~ 0.9 , 30

1 = 10.99 , 30 + 30 } = (0.99, 60

Therefore, S^1 = {(0.9, 30), (0.99, 60)}

Next find *S* 12 ~ ~~2*f* (*x*), *x* ~~ *f*2 (*x*) =

{*0*2 (1) * *f*1 ( ), *0*2 (2) * *f*1 ( ), *0*2 (3) * *f*1

( )}

DESIGN AND ANALYSIS OF ALGORITHMS

~2 ~1~ ~ 1 ~ ~1 ~ *rI* ~ *mi*= 1 – (1 – 0.8) = 1 –1 0.2 = 0.8

~2 ~ 2~ ~ 1 ~ ~1 ~ 0.8~ 2 = 0.96

*0*2 (3) = 1 - (1 - 0.8) 3 = 0.992

= {(0.8(0.9),30 + 15), (0.8(0.99),60 + 15)} = {(0.72, 45), (0.792, 75)} =
{(0.96(0.9),30 + 15 +15) , (0.96(0.99),60 + 15 + 15)}
= {(0.864, 60), (0.9504, 90)}

= {(0.992(0.9),30 + 15 +15+15) , (0.992(0.99),60 + 15 + 15+15)}
= {(0.8928, 75), (0.98208,
105)} *S*2 = {S^2 , S^2 , S^2 }

1 2 3

By applying Dominance rule to S^2:

Therefore, S2 = {(0.72, 45), (0.864, 60), (0.8928, 75)} Dominance Rule:

If S^i contains two pairs (f1, x1) and (f2, x2) with the property that f1 ≥ f2 and x1 ≤ x2, then (f1, x1) dominates (f2, x2), hence by dominance rule (f2, x2) 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 from Si.

Case 1: if f1 ≤ f2 and x1 > x2 then discard (f1, x1)

Case 2: if f1 > f2 and x1 < x2 the discard (f2, x2)

Case 3: otherwise simply write (f1, x1)

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

Ø 3 (1) = 1 ~ ~1 _ *rI* ~ *mi* = 1 – (1 – 0.5)^1 = 1 – 0.5 = 0.5

Ø *S* $_{21}$ ~2~ ~ 10.5 ~ ~~ 21 = 0.75
~

3
$_2$ = 0.875
Ø *S* ^{2} ~3~ ~ 1 ~ ~1
~ 0.5~ 3
3
*S*3
$_2$

S 13 = {(0.5 (0.72), 45 + 20), (0.5 (0.864), 60 + 20), (0.5 (0.8928), 75 + 20)}

S 13 = {(0.36, 65), (0.437, 80), (0.4464, 95)}

S_2^3 ={(0.75 (0.72), 45 + 20 + 20), (0.75 (0.864), 60 + 20 +20),
(0.75 (0.8928), 75 + 20 + 20)}

= {(0.54, 85), (0.648, 100), (0.6696, 115)}

S 3 3 = {□ 0.875 (0.72), 45 + 20 + 20 + 20), □0.875 (0.864), 60 + 20 + 20 + 20,

□ 0.875 (0.8928), 75 + 20 + 20 + 20 □ }

S 3

3 = {(0.63, 105), (1.756, 120), (0.7812, 135)}

If cost exceeds 105, remove that tuples

S3 = {(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 for the solution through S^i 's we can determine that m3 = 2, m2 = 2 and m1 = 1.

Other Solution:

According to the principle of optimality: fn(C) = max {~n (mn). fn-1 (C - Cn mn) with fo (x) = 1 and 0 ≤ x ≤ C; 1 ~ *mn* < *un*

Since, we can assume each ci > 0, each mi must be in the range 1 ≤ mi ≤ ui. Where:

~ (*n* ~ ~

S2 ={(0.75 (0.72), 45 + 20 + 20), (0.75 (0.864), 60 + *u*

20 +20),

i

= ~ i*C* + *Ci* _ ~*CJ* r / *Ci* I

~~ *i* ~~ ~~

Using the above equation compute u1, u2 and u3.

105 □30 □ + □

*u*1 = 30 15 20 ~ $\frac{70}{}$30

□ □ □

□ □ - + 20□

=

30

105 15 30 + 55 ~ 3

*u*2 = 1515 20 ~

15

30

105 _15 = 3

*u*3 = 20 20 6020

f3 (105) = max {~3 (m3). f2 (105 - 20m3)} 1 < *m*3 ! *u*3

= max {3(1) f2(105 - 20), 63(2) f2(105 - 20x2), ~3(3) f2(105 -20x3)} = max {0.5 f2(85), 0.75 f2(65), 0.875 f2(45)}

= max {0.5 x 0.8928, 0.75 x 0.864, 0.875 x 0.72} = 0.648.

= max {2 (m2). f1 (85 - 15m2)} 1 ! *m*2 ! *u*2

f2 (85) = = max {2(1).f1(85 - 15), ~2(2).f1(85 - 15x2), ~2(3).f1(85 - 15x3)}

max {0.8 f1(70), 0.96 f1(55), 0.992 f1(40)}

= max {0.8 x 0.99, 0.96 x 0.9, 0.99 x 0.9} = 0.8928

f1 (70) = max {~1(m1). f0(70 - 30m1)}

1 ! *m*1 ! *u*1

= max {~1(1) f0(70 - 30), t1(2) f0 (70 - 30x2)}

= max {~1(1) x 1, t1(2) x 1} = max {0.9, 0.99}

= 0.99 f1 (55) = max {t1(m1). f0(55 - 30m1)}

1 ! *m*1 ! *u*1

= max {~1(1) f0(50 - 30), t1(2) f0(50 - 30x2)}

= max {~1(1) x 1, t1(2) x -oo} = max {0.9, -

oo} = 0.9 f1 (40) = max {~1(m1). f0 (40 -

30m1)}

1 ! *m*1 ! *u*1

= max {~1(1) f0(40 - 30), t1(2) f0(40 - 30x2)}

= max {~1(1) x 1, t1(2) x -oo} = max{0.9, -oo} = 0.9

f2 (65) = max {2(m2). f1(65 -15m2)}
1 ! *m*2 ! *u*2

= max {2(1) f1(65 - 15), 62(2) f1(65 - 15x2), ~2(3) f1(65 - 15x3)}
= max {0.8 f1(50),

0.96 f1(35), 0.992 f1(20)}

= max {0.8 x 0.9, 0.96 x

0.9, -oo} = 0.864 f1 (50) = max

{~1(m1). f0(50 - 30m1)}

1 ! *m*1 ! *u*1

= max {~1(1) f0(50 - 30), t1(2) f0(50 - 30x2)}

= max {~1(1) x 1, t1(2) x -oo} = max{0.9, -oo} = 0.9 f1 (35) = max

~1(m1). f0(35 - 30m1)}

1 ! *m*1 ! *u*1
= max {~1(1).f0(35-30), ~1(2).f0(35-30x2)}

= max {~1(1) x 1, t1(2) x -oo} = max{0.9, -oo} = 0.9

f1 (20) = max {~1(m1). f0(20 - 30m1)}
1 ! *m*1 ! *u*1

= max {~1(1) f0(20 - 30), t1(2) f0(20 - 30x2)}

= max {~1(1) x -, ~1(2) x -oo} = max{-oo, -oo} = -oo

f2 (45) = max {2(m2). f1(45 -15m2)}
1 ! *m*2 ! *u*2

= max {2(1) f1(45 - 15), ~2(2) f1(45 - 15x2), ~2(3) f1(45 - 15x3)}
= max {0.8 f1(30),

0.96 f1(15), 0.992 f1(0)}

= max {0.8 x 0.9, 0.96 x -, 0.99 x -oo} = 0.72
f1 (30) = max {~1(m1). f0(30 - 30m1)} 1 < *m*1 ~ *u*1

= max {~1(1) f0(30 - 30), t1(2) f0(30 - 30x2)}

= max {~1(1) x 1, t1(2) x -oo} = max{0.9, -oo} = 0.9 Similarly, f1 (15) = -,

f1 (0) = -.

The best design has a reliability = 0.648 and Cost = 30 x 1 + 15 x 2 + 20 x 2 = 100.

Tracing back for the solution through S^i ‘s we can determine that:

m3 = 2, m2 = 2 and m1 = 1.

BACKTRACKING

(General method)

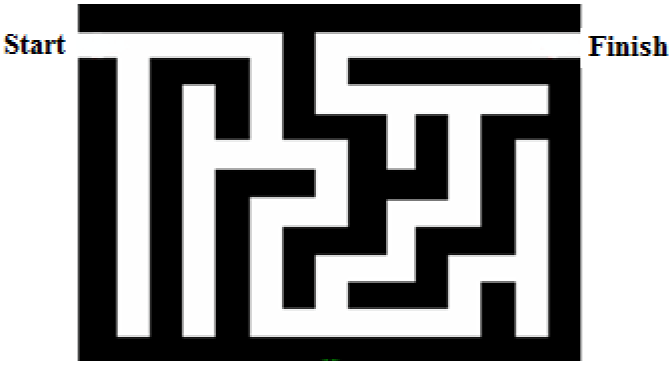
Many problems are difficult to solve algorithmically. Backtracking makes it possible to solve at least some large instances of difficult combinatorial problems.

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

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

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

Example@1 (net example) : Maze (a tour puzzle)



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

- In maze, at each intersection, you have to decide between 3 or fewer choices:
  - ✓ Go straight
  - ✓ Go left
  - ✓ Go right
- You don't have enough information to choose correctly
- Each choice leads to another set of choices.

- One or more sequences of choices may or may not lead to a solution.
- Many types of maze problem can be solved with backtracking.

Example@ 2 (text book):

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

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

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

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

m_i→size of set S_i

$m=m_1m_2m_3$---m_n n tuples that possible candidates for satisfying the function P.

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

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

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

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

- Explicit constraints.
- Implicit constraints.

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

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

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

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

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

Implicit Constraints:

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

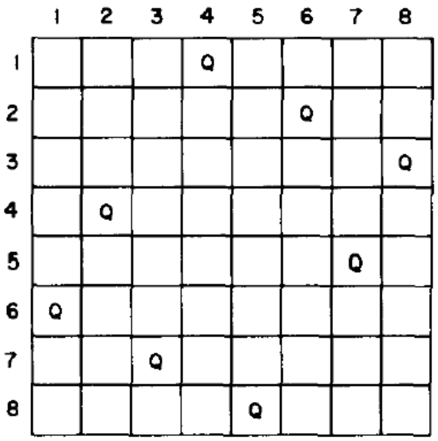
Applications of Backtracking:

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

N-Queens Problem:

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

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



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

One solution to the 8-queens problem

Here queens can also be numbered 1 through 8

Each queen must be on a different row

Assume queen ‘i’ is to be placed on row ‘i’

All solutions to the 8-queens problem can therefore be represented a s s-tuples(x_1, x_2, x_3—x_8)

xi→ the column on which queen ‘i’ is placed

si→{1, 2, 3, 4, 5, 6, 7, 8}, $1 \leq i \leq 8$

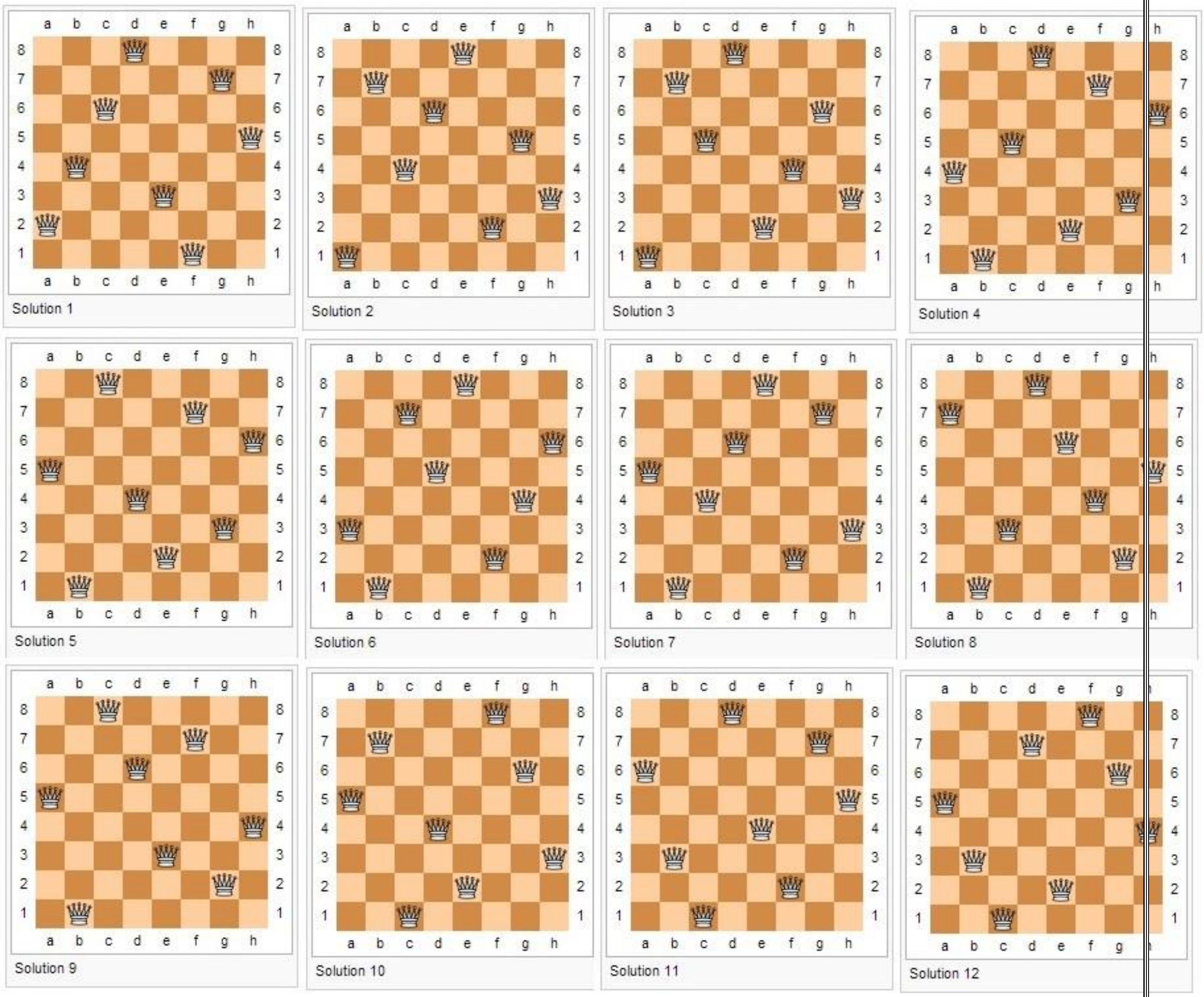
Therefore the solution space consists of 8^8 s-tuples.

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

By these two constraints the size of solution pace reduces from 88 tuples to 8! Tuples.

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

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



Solution 1

Solution 2

Solution 3

Solution 4

Solution 5

Solution 6

Solution 7

Solution 8

Solution 9

Solution 10

Solution 11

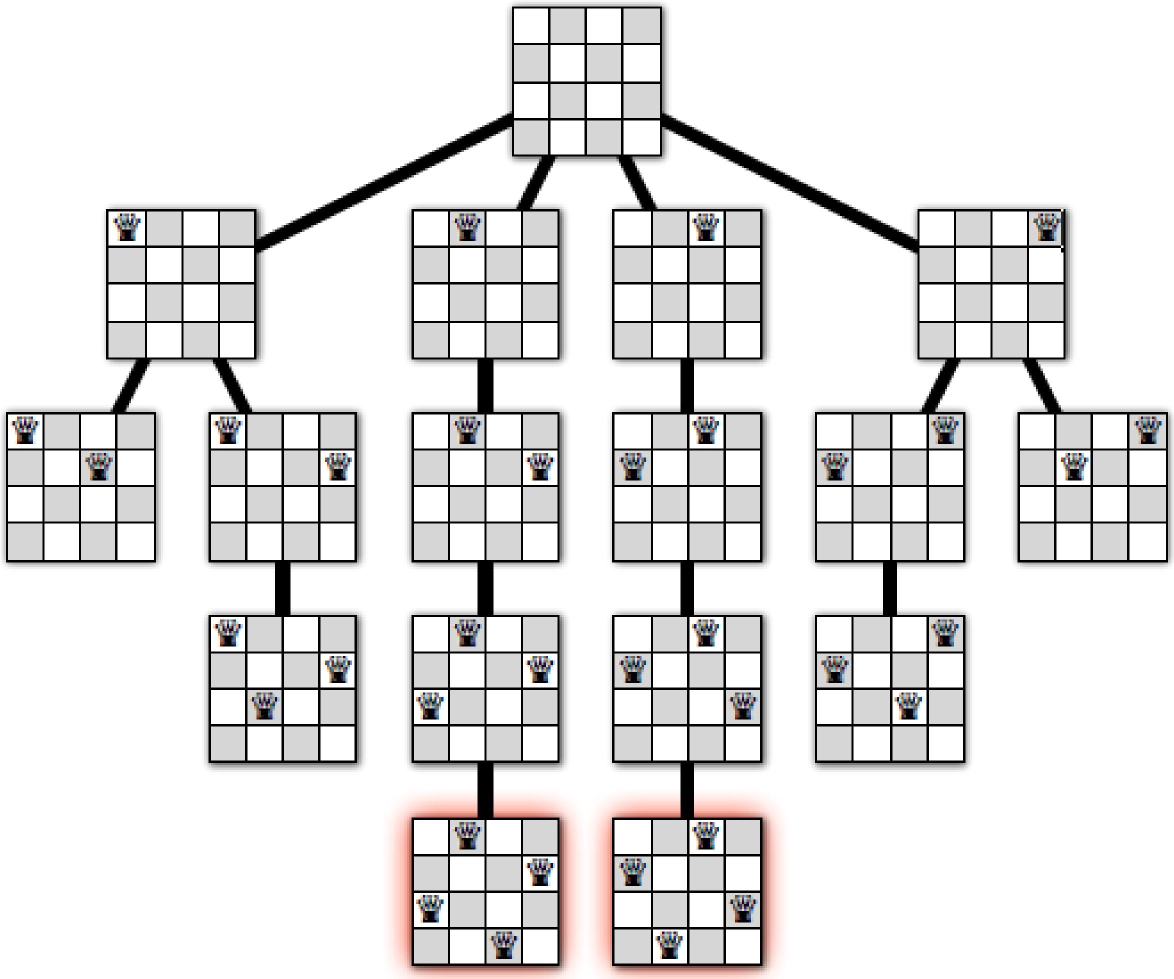
Solution 12

Some solution to the 8-Queens problem

Algorithm for new queen be placed	All solutions to the n·queens problem

Algorithm Place(k,i)
//Return true if a queen can be placed in kth row & ith column
//Other wise return false
{
for j:=1 to k-1 do
	Algorithm NQueens(k, n)
	// its prints all possible placements of n-queens on an n×n chessboard.
	{
	for i:=1 to n do{
	if Place(k,i) then

if(x[j]=i or Abs(x[j]-i)=Abs(j-k)))
then return false
return true
}
	{
	X[k]:=I;
	if(k==n) then write (x[1:n]);
	else NQueens(k+1, n);
	}
	}}



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

Sum of Subsets Problem:

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

Definition: Given n distinct +ve numbers (usually called weights), desire (want) to find all combinations of these numbers whose sums are m. this is called sum of subsets problem.

To formulate this problem by using either fixed sized tuples or variable sized tuples.

Backtracking solution uses the fixed size tuple strategy.

For example:

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

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

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

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

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

W_i→ weight of item i
M→ Capacity of bag (subset)
X_i→ the element of the solution vector is either one or zero.
X_i value depending on whether the weight wi is included or not.
If X_i=1 then wi is chosen.
If X_i=0 then wi is not chosen.

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

The above equation specify that x_1, x_2, x_3, --- x_k cannot lead to an answer node if this condition is not satisfied.

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

The equation cannot lead to solution.

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

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

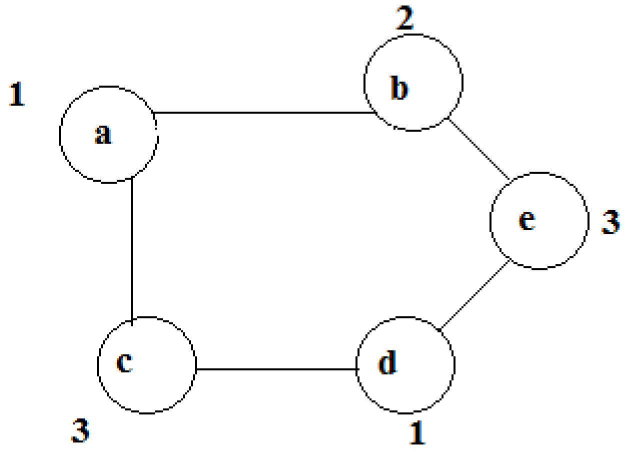
Recursive backtracking algorithm for sum of subsets problem

```
Algorithm SumOfSub(s, k, r)
{
//$s = \sum_{j=1}^{k-1} W(j)X(j).$ and $r = \sum_{j=k}^{n} W(j)$
X[k]=1
If(S+w[k]=m) then write(x[1: ]); // subset found.
Else if (S+w[k] + w{k+1] ≤ M)
Then SumOfSub(S+w[k], k+1, r-w[k]);
 if ((S+r - w{k] ≥ M) and (S+w[k+1] ≤M) ) then
{
X[k]=0;
SumOfSub(S, k+1, r-w[k]);
}
}
```

Graph Coloring:

Let G be a undirected graph and ‘m’ be a given +ve integer. The graph coloring problem is assigning colors to the vertices of an undirected graph with the restriction that no two adjacent vertices are assigned the same color yet only ‘m’ colors are used. The optimization version calls for coloring a graph using the minimum number of coloring. The decision version, known as K-coloring asks whether a graph is colourable using at most k-colors. Note that, if ‘d’ is the degree of the given graph then it can be colored with ‘d+1’ colors. The m- colorability optimization problem asks for the smallest integer ‘m’ for which the graph G can be colored. This integer is referred as “**Chromatic number**” of the graph.

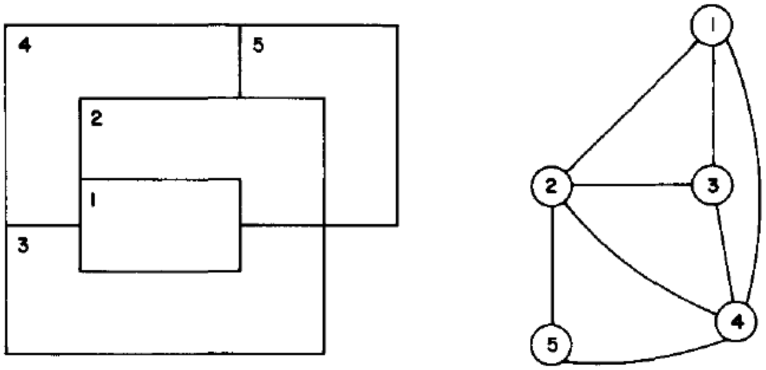
Example



- Above graph can be colored with 3 colors 1, 2, & 3.
- The color of each node is indicated next to it.
- 3-colors are needed to color this graph and hence this graph’ Chromatic Number is 3.

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

  - **Example:**



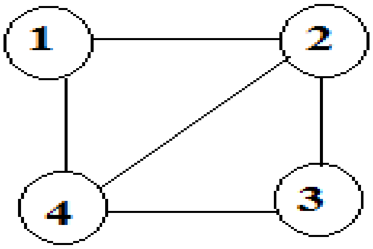
  - A map and its planar graph representation

The above map requires 4 colors.

- Many years, it was known that 5-colors were required to color this map.
- After several hundred years, this problem was solved by a group of mathematicians with the help of a computer. They show that 4-colors are sufficient.

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

Ex:

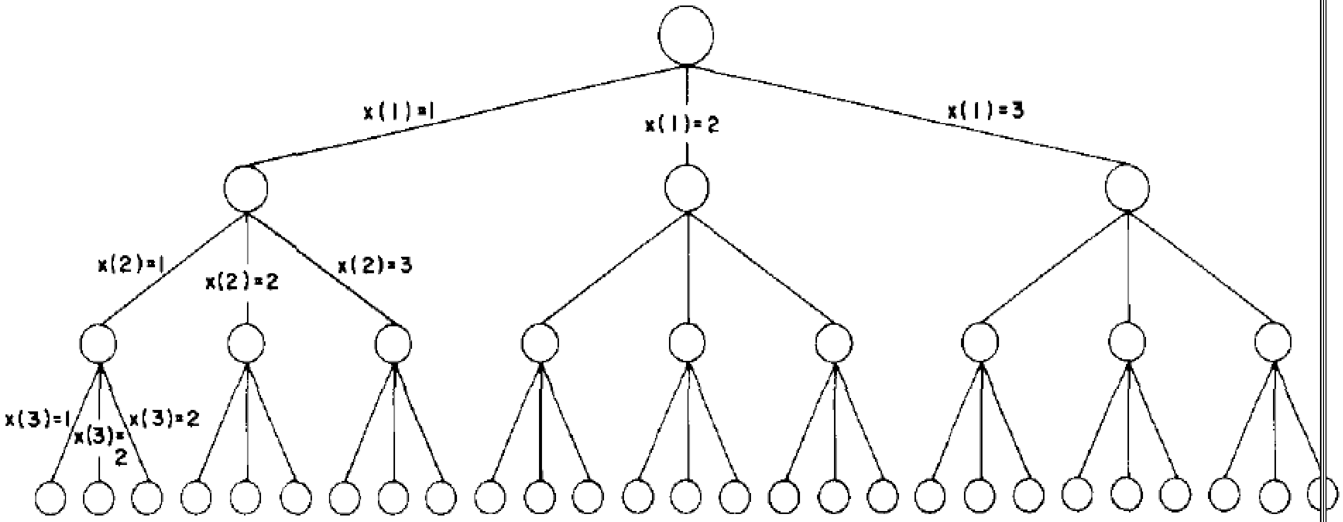


Its adjacency matrix is ⇒

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

Here G[i, j]=1 if (i, j) is an edge of G, and G[i, j]=0 otherwise.
Colors are represented by the integers 1, 2,---m and the solutions are given by the n-tuple (x1, x2,---xn)
xi→ Color of node i.

State Space Tree for
n=3→ nodes
m=3→colors



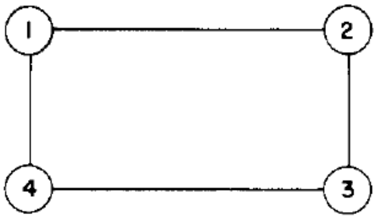
State space tree for MCOLORING when $n = 3$ and $m = 3$

1st node coloured in 3-ways
2nd node coloured in 3-ways
3rd node coloured in 3-ways
So we can colour in the graph in 27 possibilities of colouring.

Finding all m-coloring of a graph	**Getting next color**

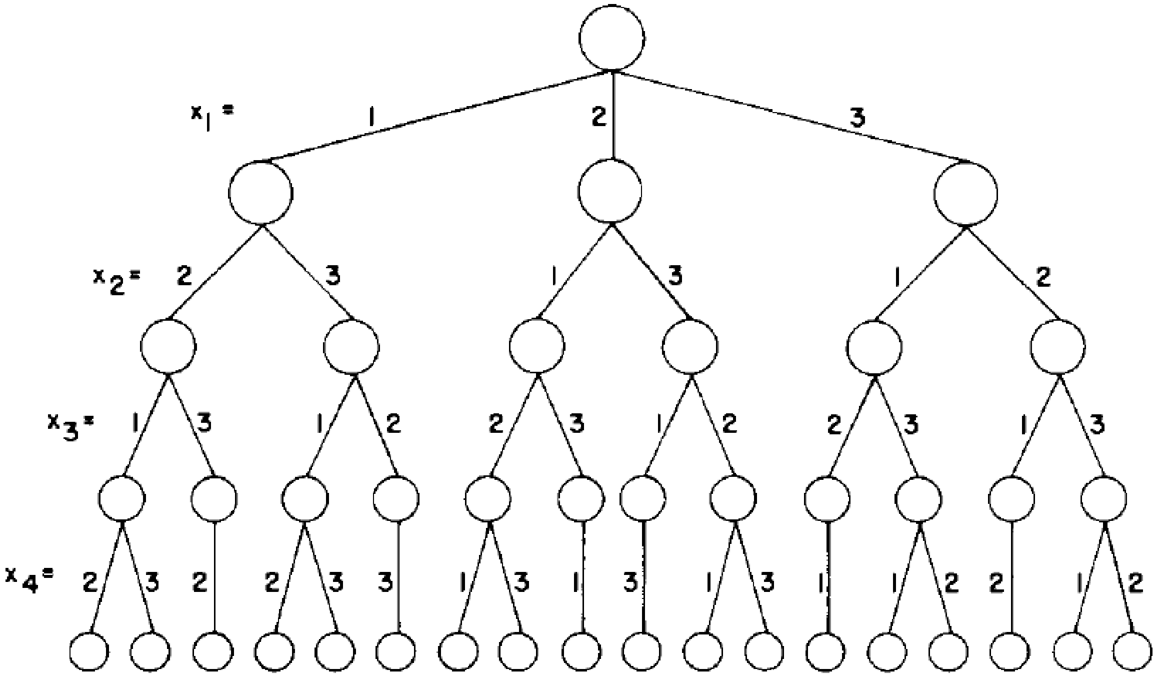
Algorithm mColoring(k){
// g(1:n, 1:n)→ boolean adjacency matrix.
// k→index (node) of the next vertex to color.
repeat{
nextvalue(k); // assign to x[k] a legal color.
if(x[k]=0) then return; // no new color possible
if(k=n) then write(x[1: n];
else mcoloring(k+1);
}
until(false)
}
	Algorithm NextValue(k){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

Previous paper example:



Adjacency matrix is

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

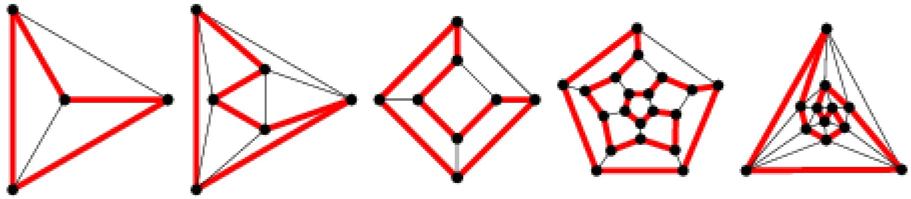


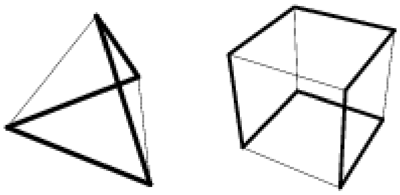
A 4 node graph and all possible 3 colorings

Hamiltonian Cycles:

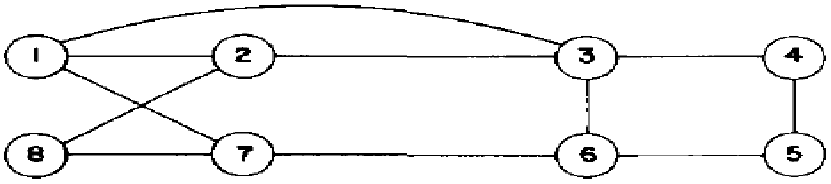
- **Def:** Let G=(V, E) be a connected graph with n vertices. A Hamiltonian cycle is a round trip path along n-edges of G that visits every vertex once & returns to its starting position.
- It is also called the Hamiltonian circuit.
- Hamiltonian circuit is a graph cycle (i.e., closed loop) through a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

Example:



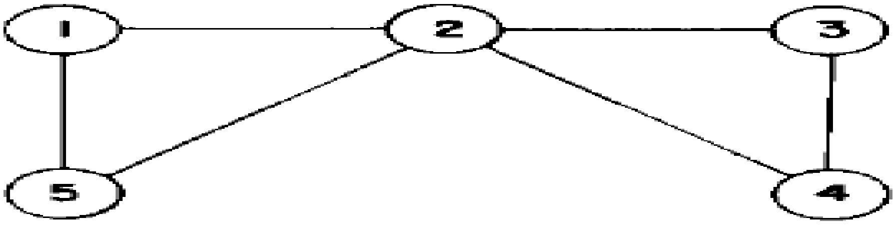


- In graph G, Hamiltonian cycle begins at some vertiex v1 ∈ G and the vertices of G are visited in the order v_1, v_2,---v_{n+1}, then the edges (v_i, v_{i+1}) are in E, $1 \leq i \leq n$.



→g1

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



The above graph contains no Hamiltonian cycles.

- There is no known easy way to determine whether a given graph contains a Hamiltonian cycle.
- By using backtracking method, it can be possible
  - Backtracking algorithm, that finds all the Hamiltonian cycles in a graph.
  - The graph may be directed or undirected. Only distinct cycles are output.
  - From graph g1 backtracking solution vector= {1, 2, 8, 7, 6, 5, 4, 3, 1}
  - The backtracking solution vector (x_1, x_2, --- x_n)
    x_i→ i^{th} visited vertex of proposed cycle.
  - By using backtracking we need to determine how to compute the set of possible vertices for x_k if x_1,x_2,x_3---x_{k-1} have already been chosen.
    If k=1 then x1 can be any of the n-vertices.

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

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

Generating Next Vertex	**Finding all Hamiltonian Cycles**

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

UNIT-V

BRANCH & BOUND

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

➢ The B&B strategy is very similar to backtracking in that a state space tree is used to solve a problem.

➢ The differences are that the B&B method

✓ Does not limit us to any particular way of traversing the tree.

✓ It is used only for optimization problem

✓ It is applicable to a wide variety of discrete combinatorial problem.

➢ B&B is rather general optimization technique that applies where the greedy method & dynamic programming fail.

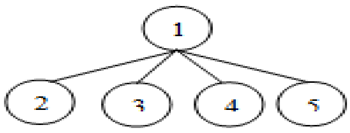
➢ It is much slower, indeed (truly), it often (rapidly) leads to exponential time complexities in the worst case.

➢ The term B&B refers to all state space search methods in which all children of the "E-node" are generated before any other "live node" can become the "E-node"

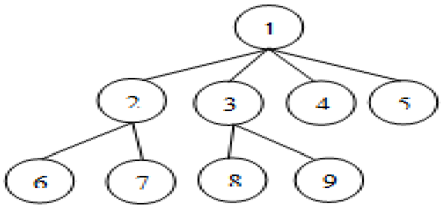
✓ **Live node→** is a node that has been generated but whose children have not yet been generated.

✓ **E-node→**is a live node whose children are currently being explored.

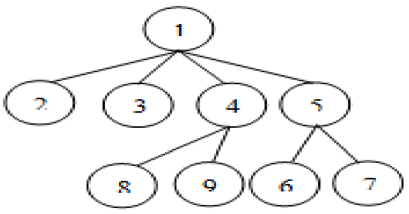
✓ **Dead node→** is a generated node that is not to be expanded or explored any further. All children of a dead node have already been expanded.



Live Node: 2, 3, 4, and 5



FIFO Branch & Bound (BFS)
Children of E-node are inserted in a queue.



LIFO Branch & Bound (D-Search)
Children of E-node are inserted in a stack.

➢ Two graph search strategies, BFS & D-search (DFS) in which the exploration of a new node cannot begin until the node currently being explored is fully explored.

➢ Both BFS & D-search (DFS) generalized to B&B strategies.

✓ **BFS→**like state space search will be called FIFO (First In First Out) search as the list of live nodes is "First-in-first-out" list (or queue).

✓ **D-search (DFS)→** Like state space search will be called LIFO (Last In First Out) search as the list of live nodes is a "last-in-first-out" list (or stack).

➢ In backtracking, bounding function are used to help avoid the generation of sub-trees that do not contain an answer node.

➢ We will use 3-types of search strategies in branch and bound

1) FIFO (First In First Out) search
2) LIFO (Last In First Out) search
3) LC (Least Count) search

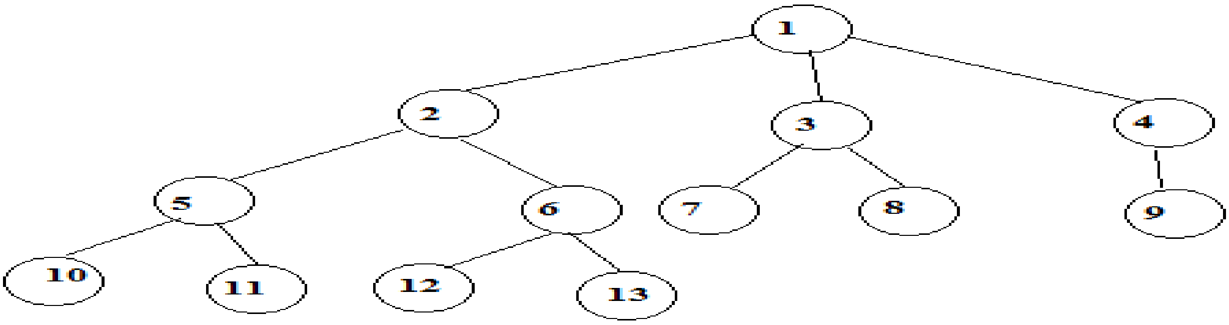
FIFO B&B:

FIFO Branch & Bound is a BFS.

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

Implementation of list of live nodes as a queue

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



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

1

2	3	4

3	4	5	6

4	5	6

5	6

6

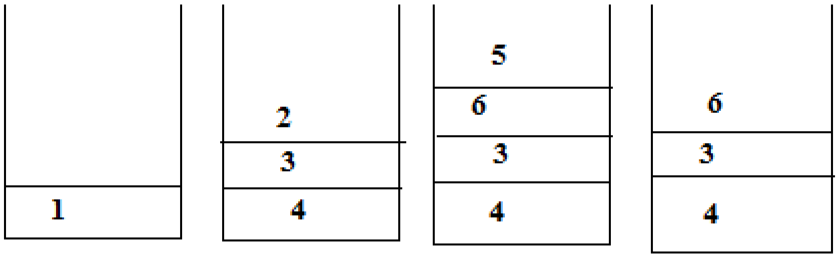
LIFO B&B:

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

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

Implementation of List of live nodes as a stack

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



Least Cost (LC) Search:

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

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

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

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

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

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

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

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

Example:

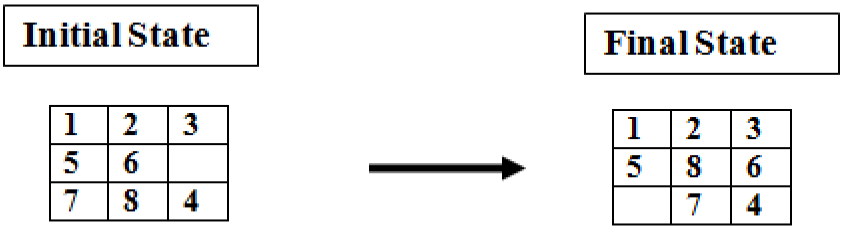
8-puzzle

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

where h(x) = the number of misplaced tiles

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

Assumption: move one tile in any direction cost 1.

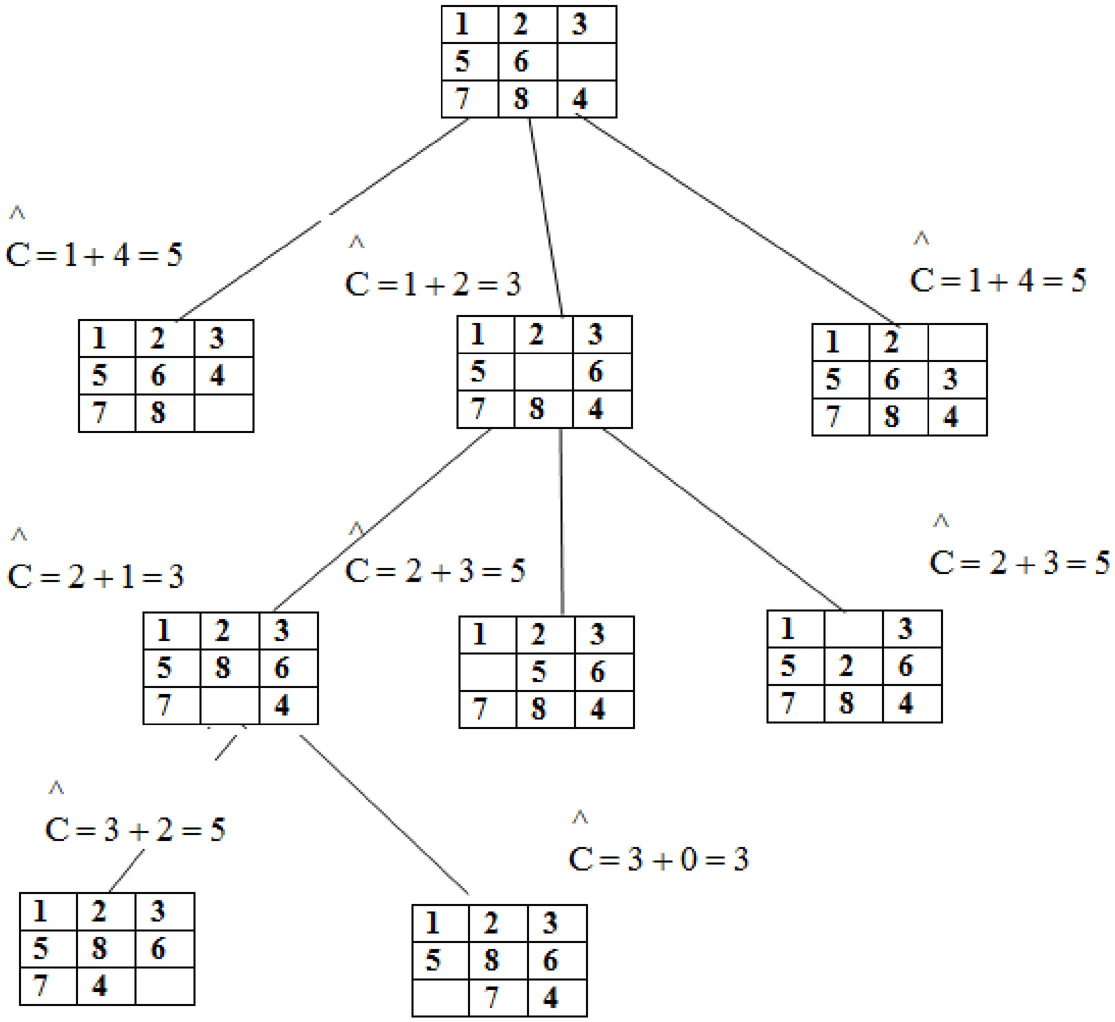


Initial State

1	2	3
5	6	
7	8	4

Final State

1	2	3
5	8	6
	7	4



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

Travelling Salesman Problem:

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

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

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

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

Let C_{ij}→ cost of edge <i, j>

$C_{ij} = \infty$ if <i, j> $\notin$ E

|V|=n→ total number of vertices.

Assume that every tour starts & ends at vertex 1.

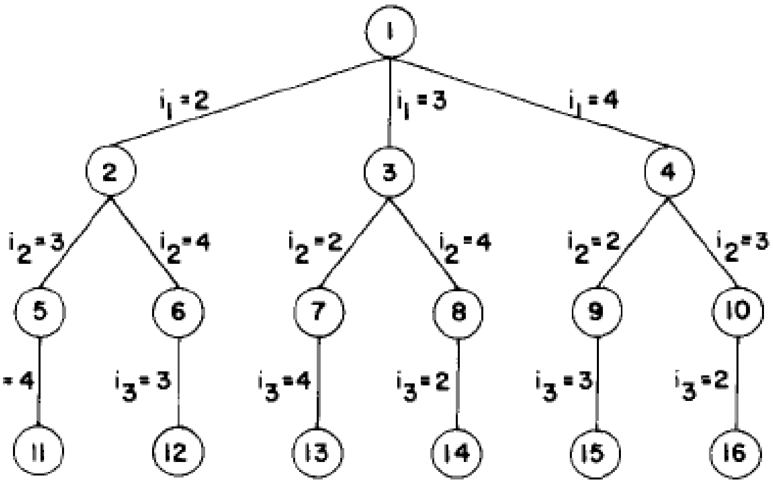
Solution Space S= {1, Π , 1 / Π is a permutation of (2, 3. 4. ----n) } then |S|=(n-1)!

The size of S reduced by restricting S

Sothat $(1, i_1, i2, \text{-----} i_{n-1}, 1\} \in$ S iff $<i_j, i_{j+1}> \in$ E. $O \leq j \leq n-1$, $i_0 - i_n = 1$

S can be organized into “State space tree”.

Consider the following Example



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

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

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

Node 12 represents the tour.

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

Node 14 represents the tour.

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

TSP is solved by using LC Branch & Bound:

To use LCBB to search the travelling salesperson “State space tree” first define a cost function C(.) and other 2 functions Ĉ(.) & u(.)

Such that Ĉ(r) ≤ C(r) ≤ u(r) → ① for all nodes r.

Cost C(.)→ is the solution node with least C(.) corresponds to a shortest tour in G.

C(A)={Length of tour defined by the path from root to A if A is leaf

Cost of a minimum-cost leaf in the sub-tree A, if A is not leaf }

Fr① Ĉ(r) ≤ C(r) then Ĉ(r) → is the length of the path defined at node A.

From previous example the path defined at node 6 is i_0, i_1, i_2=1, 2, 4 & it consists edge of <1,2> & <2,4>

Abetter Ĉ(r) can be obtained by using the reduced cost matrix corresponding to G.

- A row (column) is said to be reduced iff it contains at least one zero & remaining entries are non negative.
- A matrix is reduced iff every row & column is reduced.

$$\begin{bmatrix} \infty & 20 & 30 & 10 & 11 \\ 15 & \infty & 16 & 4 & 2 \\ 3 & 5 & \infty & 2 & 4 \\ 19 & 6 & 18 & \infty & 3 \\ 16 & 4 & 7 & 16 & \infty \end{bmatrix}$$

(a) Cost Matrix

$$\begin{bmatrix} \infty & 10 & 17 & 0 & 1 \\ 12 & \infty & 11 & 2 & 0 \\ 0 & 3 & \infty & 0 & 2 \\ 15 & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & 12 & \infty \end{bmatrix}$$

(b) Reduced Cost Matrix
L = 25

Given the following cost matrix:

$$\begin{bmatrix} inf & 20 & 30 & 10 & 11 \\ 15 & inf & 16 & 4 & 2 \\ 3 & 5 & inf & 2 & 4 \\ 19 & 6 & 18 & inf & 3 \\ 16 & 4 & 7 & 16 & inf \end{bmatrix}$$

- The TSP starts from node 1: **Node 1**
- Reduced Matrix: To get the lower bound of the path starting at node 1

Row # 1: reduce by 10 $\begin{bmatrix} inf & 10 & 20 & 0 \\ 15 & inf & 16 & 4 \\ 3 & 5 & inf & 2 \\ 19 & 6 & 18 & inf \\ 16 & 4 & 7 & 16 \end{bmatrix}$	Row #2: reduce 2 $\begin{bmatrix} inf & 10 & 20 & 0 \\ 13 & inf & 14 & 2 \\ 3 & 5 & inf & 2 \\ 19 & 6 & 18 & inf \\ 16 & 4 & 7 & 16 & i \end{bmatrix}$	Row #3: reduce by 2 $\begin{bmatrix} inf & 10 & 20 & 0 \\ 13 & inf & 14 & 2 \\ 1 & 3 & inf & 0 \\ 19 & 6 & 18 & inf \\ 16 & 4 & 7 & 16 & i \end{bmatrix}$
Row # 4: Reduce by 3: $\begin{bmatrix} inf & 10 & 20 & 0 \\ 13 & inf & 14 & 2 \\ 1 & 3 & inf & 0 \\ 16 & 3 & 15 & inf \\ 16 & 4 & 7 & 16 & i \end{bmatrix}$	Row # 5: Reduce by 4 $\begin{bmatrix} inf & 10 & 20 & 0 \\ 13 & inf & 14 & 2 \\ 1 & 3 & inf & 0 \\ 16 & 3 & 15 & inf \\ 12 & 0 & 3 & 12 & i \end{bmatrix}$	Column 1: Reduce by 1 $\begin{bmatrix} inf & 10 & 20 & 0 \\ 12 & inf & 14 & 2 \\ 0 & 3 & inf & 0 \\ 15 & 3 & 15 & inf \\ 11 & 0 & 3 & 12 & i \end{bmatrix}$

Column 2: It is reduced.	Column 3: Reduce by 3 $\begin{bmatrix} inf & 10 & 17 & 0 & 1 \\ 12 & inf & 11 & 2 & 0 \\ 0 & 3 & inf & 0 & 2 \\ 15 & 3 & 12 & inf & 0 \\ 11 & 0 & 0 & 12 & inf \end{bmatrix}$	Column 4: It is reduced. Column 5: It is reduced.

The reduced cost is: RCL = 25

So the cost of node 1 is: Cost (1) = 25

The reduced matrix is:

Cost (1) = 25

$$\begin{bmatrix} inf & 10 & 17 & 0 & 1 \\ 12 & inf & 11 & 2 & 0 \\ 0 & 3 & inf & 0 & 2 \\ 15 & 3 & 12 & inf & 0 \\ 11 & 0 & 0 & 12 & inf \end{bmatrix}$$

➢ **Choose to go to vertex 2: Node 2**

- Cost of edge <1,2> is: A(1,2) = 10
- Set row #1 = inf since we are choosing edge <1,2>
- Set column # 2 = inf since we are choosing edge <1,2>
- Set A(2,1) = inf
- The resulting cost matrix is:

$$\begin{bmatrix} inf & inf & inf & inf & inf \\ inf & inf & 11 & 2 & 0 \\ 0 & inf & inf & 0 & 2 \\ 15 & inf & 12 & inf & 0 \\ 11 & inf & 0 & 12 & inf \end{bmatrix}$$

- The matrix is reduced:
- RCL = 0
- The cost of node 2 (Considering vertex 2 from vertex 1) is:

Cost(2) = cost(1) + A(1,2) = 25 + 10 = 35

➢ **Choose to go to vertex 3: Node 3**

- Cost of edge <1,3> is: A(1,3) = 17 (In the reduced matrix
- Set row #1 = inf since we are starting from node 1
- Set column # 3 = inf since we are choosing edge <1,3>
- Set A(3,1) = inf
- The resulting cost matrix is:

$$\begin{bmatrix} inf & inf & inf & inf & inf \\ 12 & inf & inf & 2 & 0 \\ inf & 3 & inf & 0 & 2 \\ 15 & 3 & inf & inf & 0 \\ 11 & 0 & inf & 12 & inf \end{bmatrix}$$

Reduce the matrix: Rows are reduced

The columns are reduced except for column # 1:

Reduce column 1 by 11:

$$\begin{bmatrix} inf & inf & inf & inf & inf \\ 1 & inf & inf & 2 & 0 \\ inf & 3 & inf & 0 & 2 \\ 4 & 3 & inf & inf & 0 \\ 0 & 0 & inf & 12 & inf \end{bmatrix}$$

The lower bound is: RCL = 11

The cost of going through node 3 is:

cost(3) = cost(1) + RCL + A(1,3) = 25 + 11 + 17 = 53

- Choose to go to vertex 4: **Node 4**

Remember that the cost matrix is the one that was reduced at the starting vertex 1

Cost of edge <1,4> is: A(1,4) = 0

Set row #1 = inf since we are starting from node 1

Set column # 4 = inf since we are choosing edge <1,4>

Set A(4,1) = inf

The resulting cost matrix is:

$$\begin{bmatrix} inf & inf & inf & inf & inf \\ 12 & inf & 11 & inf & 0 \\ 0 & 3 & inf & inf & 2 \\ inf & 3 & 12 & inf & 0 \\ 11 & 0 & 0 & inf & inf \end{bmatrix}$$

Reduce the matrix: Rows are reduced

Columns are reduced

The lower bound is: RCL = 0

The cost of going through node 4 is:

cost(4) = cost(1) + RCL + A(1,4) = 25 + 0 + 0 = 25

- **Choose to go to vertex 5: Node 5**

- Remember that the cost matrix is the one that was reduced at starting vertex 1
- Cost of edge <1,5> is: A(1,5) = 1
- Set row #1 = inf since we are starting from node 1
- Set column # 5 = inf since we are choosing edge <1,5>
- Set A(5,1) = inf
- The resulting cost matrix is:

$$\begin{bmatrix} inf & inf & inf & inf & inf \\ 12 & inf & 11 & 2 & inf \\ 0 & 3 & inf & 0 & inf \\ 15 & 3 & 12 & inf & inf \\ inf & 0 & 0 & 12 & inf \end{bmatrix}$$

Reduce the matrix:

Reduce rows:

Reduce row #2: Reduce by 2

$$\begin{bmatrix} inf & inf & inf & inf & inf \\ 10 & inf & 9 & 0 & inf \\ 0 & 3 & inf & 0 & inf \\ 15 & 3 & 12 & inf & inf \\ inf & 0 & 0 & 12 & inf \end{bmatrix}$$

Reduce row #4: Reduce by 3

$$\begin{bmatrix} inf & inf & inf & inf & inf \\ 10 & inf & 9 & 0 & inf \\ 0 & 3 & inf & 0 & inf \\ 12 & 0 & 9 & inf & inf \\ inf & 0 & 0 & 12 & inf \end{bmatrix}$$

Columns are reduced

The lower bound is: RCL = 2 + 3 = 5

The cost of going through node 5 is:

cost(5) = cost(1) + RCL + A(1,5) = 25 + 5 + 1 = 31

In summary:

So the live nodes we have so far are:

- ✓ 2: cost(2) = 35, path: 1->2
- ✓ 3: cost(3) = 53, path: 1->3
- ✓ 4: cost(4) = 25, path: 1->4
- ✓ 5: cost(5) = 31, path: 1->5

Explore the node with the lowest cost: Node 4 has a cost of 25

Vertices to be explored from node 4: 2, 3, and 5

Now we are starting from the cost matrix at node 4 is:

Cost (4) = 25

$$\begin{bmatrix} inf & inf & inf & inf & inf \\ 12 & inf & 11 & inf & 0 \\ 0 & 3 & inf & inf & 2 \\ inf & 3 & 12 & inf & 0 \\ 11 & 0 & 0 & inf & inf \end{bmatrix}$$

- ➢ **<u>Choose to go to vertex 2: Node 6 (</u>path is 1->4->2<u>)</u>**

Cost of edge <4,2> is: A(4,2) = 3

Set row #4 = inf since we are considering edge <4,2>

Set column # 2 = inf since we are considering edge <4,2>

Set A(2,1) = inf

The resulting cost matrix is:

$$\begin{bmatrix} inf & inf & inf & inf & inf \\ inf & inf & 11 & inf & 0 \\ 0 & inf & inf & inf & 2 \\ inf & inf & inf & inf & inf \\ 11 & inf & 0 & inf & inf \end{bmatrix}$$

Reduce the matrix: Rows are reduced

Columns are reduced

The lower bound is: RCL = 0

The cost of going through node 2 is:

cost(6) = cost(4) + RCL + A(4,2) = 25 + 0 + 3 = 28

- **Choose to go to vertex 3: Node 7 (path is 1->4->3)**

Cost of edge <4,3> is: A(4,3) = 12

Set row #4 = inf since we are considering edge <4,3>

Set column # 3 = inf since we are considering edge <4,3>

Set A(3,1) = inf

The resulting cost matrix is:

$$\begin{bmatrix} inf & inf & inf & inf & inf \\ 12 & inf & inf & inf & 0 \\ inf & 3 & inf & inf & 2 \\ inf & inf & inf & inf & inf \\ 11 & 0 & inf & inf & inf \end{bmatrix}$$

Reduce the matrix:

Reduce row #3: by 2:

$$\begin{bmatrix} inf & inf & inf & inf & inf \\ 12 & inf & inf & inf & 0 \\ inf & 1 & inf & inf & 0 \\ inf & inf & inf & inf & inf \\ 11 & 0 & inf & inf & inf \end{bmatrix}$$

Reduce column # 1: by 11

$$\begin{bmatrix} inf & inf & inf & inf & inf \\ 1 & inf & inf & inf & 0 \\ inf & 1 & inf & inf & 0 \\ inf & inf & inf & inf & inf \\ 0 & 0 & inf & inf & inf \end{bmatrix}$$

The lower bound is: RCL = 13

So the RCL of node 7 (Considering vertex 3 from vertex 4) is:

Cost(7) = cost(4) + RCL + A(4,3) = 25 + 13 + 12 = 50

- Choose to go to vertex 5: **Node 8** (path is 1->4->5)

Cost of edge <4,5> is: A(4,5) = 0
Set row #4 = inf since we are considering edge <4,5>
Set column # 5 = inf since we are considering edge <4,5>
Set A(5,1) = inf
The resulting cost matrix is:

$$\begin{bmatrix} inf & inf & inf & inf & inf \\ 12 & inf & 11 & inf & inf \\ 0 & 3 & inf & inf & inf \\ inf & inf & inf & inf & inf \\ inf & 0 & 0 & inf & inf \end{bmatrix}$$

Reduce the matrix:
Reduced row 2: by 11

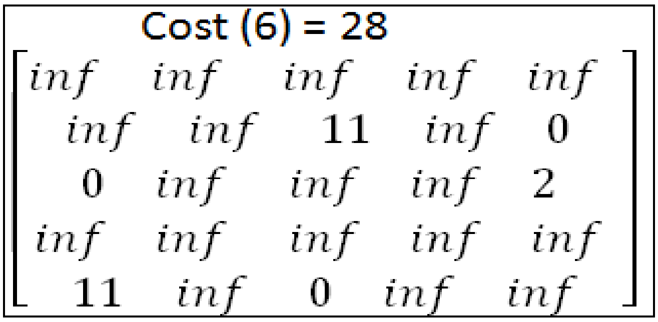
$$\begin{bmatrix} inf & inf & inf & inf & inf \\ 1 & inf & 0 & inf & inf \\ 0 & 3 & inf & inf & inf \\ inf & inf & inf & inf & inf \\ inf & 0 & 0 & inf & inf \end{bmatrix}$$

Columns are reduced
The lower bound is: RCL = 11
So the cost of node 8 (Considering vertex 5 from vertex 4) is:
Cost(8) = cost(4) + RCL + A(4,5) = 25 + 11 + 0 = 36

In summary: So the live nodes we have so far are:

- ✓ 2: cost(2) = 35, path: 1->2
- ✓ 3: cost(3) = 53, path: 1->3
- ✓ 5: cost(5) = 31, path: 1->5
- ✓ 6: cost(6) = 28, path: 1->4->2
- ✓ 7: cost(7) = 50, path: 1->4->3
- ✓ 8: cost(8) = 36, path: 1->4->5
- Explore the node with the lowest cost: Node 6 has a cost of 28
- Vertices to be explored from node 6: 3 and 5

- Now we are starting from the cost matrix at node 6 is:



Cost (6) = 28

$$\begin{bmatrix} inf & inf & inf & inf & inf \\ inf & inf & 11 & inf & 0 \\ 0 & inf & inf & inf & 2 \\ inf & inf & inf & inf & inf \\ 11 & inf & 0 & inf & inf \end{bmatrix}$$

- **<u>Choose to go to vertex 3: Node 9 (</u> path is 1->4->2->3)**

Cost of edge <2,3> is: A(2,3) = 11

Set row #2 = inf since we are considering edge <2,3>

Set column # 3 = inf since we are considering edge <2,3>

Set A(3,1) = inf

The resulting cost matrix is:

$$\begin{bmatrix} inf & inf & inf & inf & inf \\ inf & inf & inf & inf & inf \\ inf & inf & inf & inf & 2 \\ inf & inf & inf & inf & inf \\ 11 & inf & inf & inf & inf \end{bmatrix}$$

Reduce the matrix: Reduce row #3: by 2

$$\begin{bmatrix} inf & inf & inf & inf & inf \\ inf & inf & inf & inf & inf \\ inf & inf & inf & inf & 0 \\ inf & inf & inf & inf & inf \\ 11 & inf & inf & inf & inf \end{bmatrix}$$

Reduce column # 1: by 11

$$\begin{bmatrix} inf & inf & inf & inf & inf \\ inf & inf & inf & inf & inf \\ inf & inf & inf & inf & 0 \\ inf & inf & inf & inf & inf \\ 0 & inf & inf & inf & inf \end{bmatrix}$$

The lower bound is: RCL = 2 +11 = 13

So the cost of node 9 (Considering vertex 3 from vertex 2) is:

Cost(9) = cost(6) + RCL + A(2,3) = 28 + 13 + 11 = 52

- **Choose to go to vertex 5: Node 10 (path is 1->4->2->5)**

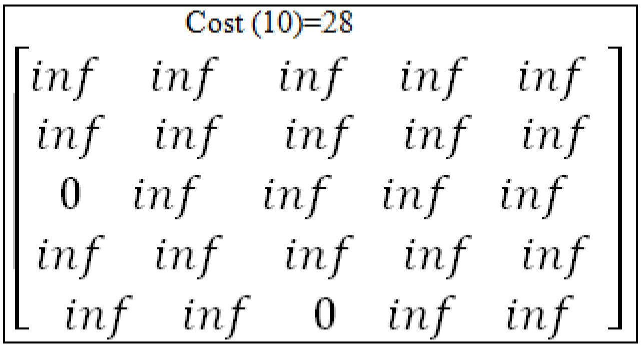
Cost of edge <2,5> is: A(2,5) = 0
Set row #2 = inf since we are considering edge <2,3>
Set column # 3 = inf since we are considering edge <2,3>
Set A(5,1) = inf
The resulting cost matrix is:

$$\begin{bmatrix} inf & inf & inf & inf & inf \\ inf & inf & inf & inf & inf \\ 0 & inf & inf & inf & inf \\ inf & inf & inf & inf & inf \\ inf & inf & 0 & inf & inf \end{bmatrix}$$

Reduce the matrix: Rows reduced
Columns reduced
The lower bound is: RCL = 0
So the cost of node 10 (Considering vertex 5 from vertex 2) is:
Cost(10) = cost(6) + RCL + A(2,3) = 28 + 0 + 0 = 28

In summary: **So the live nodes we have so far are:**

- ✓ 2: cost(2) = 35, path: 1->2
- ✓ 3: cost(3) = 53, path: 1->3
- ✓ 5: cost(5) = 31, path: 1->5
- ✓ 7: cost(7) = 50, path: 1->4->3
- ✓ 8: cost(8) = 36, path: 1->4->5
- ✓ 9: cost(9) = 52, path: 1->4->2->3
- ✓ 10: cost(2) = 28, path: 1->4->2->5
- Explore the node with the lowest cost: Node 10 has a cost of 28
- Vertices to be explored from node 10: 3
- Now we are starting from the cost matrix at node 10 is:



Cost (10)=28

$$\begin{bmatrix} inf & inf & inf & inf & inf \\ inf & inf & inf & inf & inf \\ 0 & inf & inf & inf & inf \\ inf & inf & inf & inf & inf \\ inf & inf & 0 & inf & inf \end{bmatrix}$$

- **Choose to go to vertex 3: Node 11 (path is 1->4->2->5->3)**

Cost of edge <5,3> is: A(5,3) = 0

Set row #5 = inf since we are considering edge <5,3>

Set column # 3 = inf since we are considering edge <5,3>

Set A(3,1) = inf

The resulting cost matrix is:

$$\begin{bmatrix} inf & inf & inf & inf & inf \\ inf & inf & inf & inf & inf \\ inf & inf & inf & inf & inf \\ inf & inf & inf & inf & inf \\ inf & inf & inf & inf & inf \end{bmatrix}$$

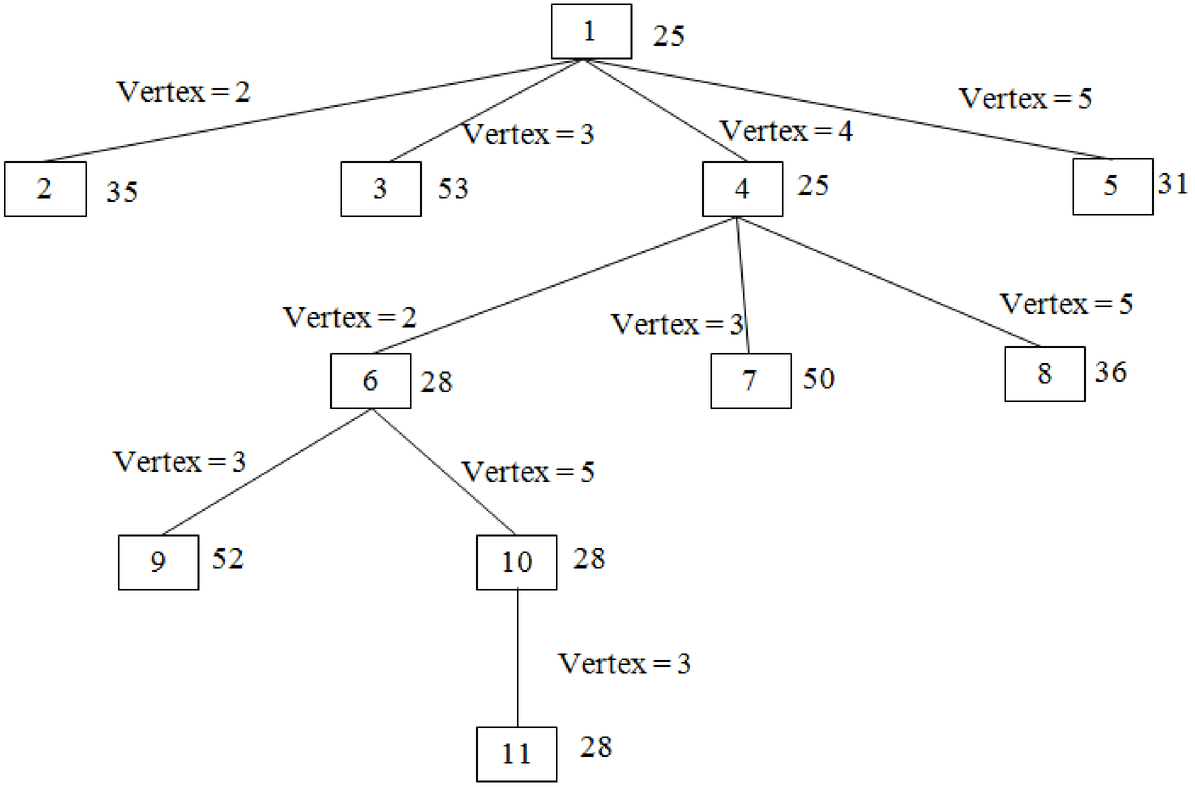
Reduce the matrix: Rows reduced
Columns reduced

The lower bound is: RCL = 0

So the cost of node 11 (Considering vertex 5 from vertex 3) is:

Cost(11) = cost(10) + RCL + A(5,3) = 28 + 0 + 0 = 28

State Space Tree:



O/1 Knapsack Problem

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

O-1 Knapsack Problem can formulate as. Let there be n items, **Z_1 to Z_n** where **Z_i** has value **P_i** & weight **w_i**. The maximum weight that can carry in the bag is m.

All values and weights are non negative.

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

The formula can be stated as

$$\text{maximize} \sum_{1 \le i \le n} p_i x_i$$

$$\text{subject to} \sum_{1 \le i \le n} w_i x_i \le M$$

X_i=0 or 1 1 ≤ i ≤ n

To solve o/1 knapsack problem using B&B:

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

$$\text{Minimize} -\sum_{i=1}^{n} p_i x_i$$

$$\text{subject to} \sum_{i=1}^{n} w_i x_i < m,$$

$$x_i \in \{0,1\},\ 1 \le i \le n$$

- Fixed tuple size solution space:

○ Every leaf node in state space tree represents an answer for which $\sum_{1 \le i \le n} w_i x_i \leqslant m$ is an answer node; other leaf nodes are infeasible

○ For optimal solution, define

$$c(x) = -\sum_{1 \le i \le n} p_i x_i$$

for every answer node x

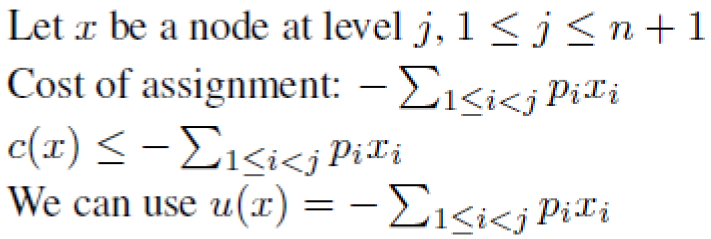
- For infeasible leaf nodes, $c(x) = \infty$
- For non leaf nodes

c(x) = min{c(lchild(x)), c(rchild(x))}

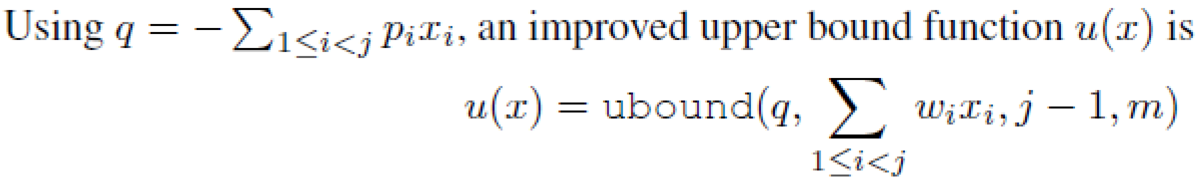
- Define two functions ĉ(x) and u(x) such that for every node x,

ĉ(x) ≤ c(x) ≤ u(x)

- Computing ĉ(·) and u(·)



Let x be a node at level j, $1 \le j \le n+1$
Cost of assignment: $-\sum_{1\le i<j} p_i x_i$
$c(x) \le -\sum_{1\le i<j} p_i x_i$
We can use $u(x) = -\sum_{1\le i<j} p_i x_i$



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

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

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

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	Group 2
➢ Problems with solution time bound by a polynomial of a small degree.	➢ Problems with solution times not bound by polynomial (simply non polynomial)
➢ It also called "Tractable Algorithms"	➢ These are hard or intractable problems
➢ Most Searching & Sorting algorithms are polynomial time algorithms	➢ None of the problems in this group has been solved by any polynomial time algorithm
➢ **Ex:** Ordered Search **(O (log n)),** Polynomial evaluation **O(n)** Sorting **O(n.log n)**	➢ **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 2nd group (i.e., group 2)

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

Theory of NP-Completeness:

Show that 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 NPComplete.

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:

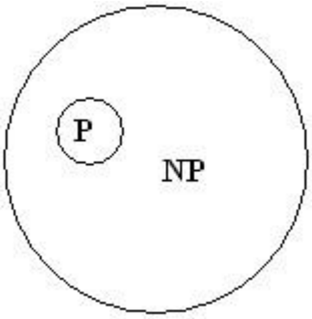
For measuring the complexity of an algorithm, we use the input length as the parameter. For example, An algorithm A is of polynomial complexity p() such that the computing time of A is O(p(n)) for every input of size n.

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 ⊆ 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.

Cook answered this question with

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

□Notation of Reducibility

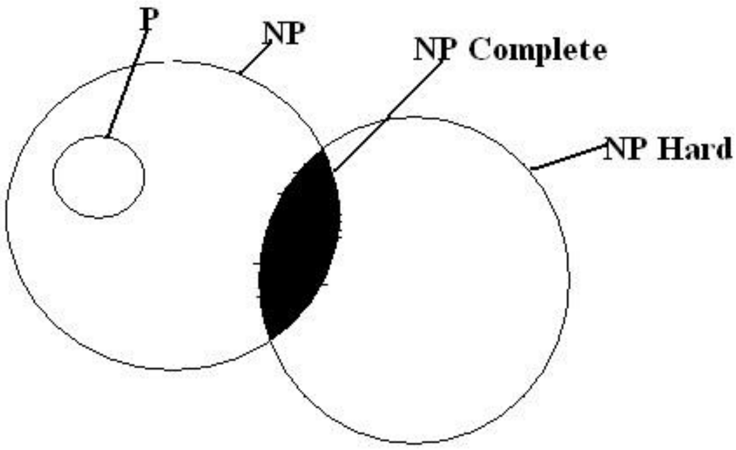
Let L_1 and L_2 be problems, Problem L_1 reduces to L_2 (written $\mathbf{L_1 \alpha L_2}$) iff there is a way to solve L_1 by a deterministic polynomial time algorithm using a deterministic algorithm that solves L_2 in polynomial time

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., $\mathbf{L_1 \alpha L_2}$ and $\mathbf{L_2 \alpha L_3}$ then $\mathbf{L_1 \alpha L_3}$

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.

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

If P=NP then satisfiability is in P

If satisfiability is in P, then P=NP

To do this

- A□ Any polynomial time nondeterministic decision algorithm.

  I□Input of that algorithm

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

- If the length of '**I**' is '**n'** and the time complexity of **A** is **p(n)** for some polynomial **p()** then length of Q is **O(p^3(n) log n)=O(p^4(n))** The time needed to construct Q is also **O(p^3(n) log n).**
- A deterministic algorithm '**Z'** to determine the outcome of '**A**' on any input '**I**'

  Algorithm **Z** computes '**Q**' and then uses a deterministic algorithm for the satisfiability problem to determine whether '**Q**' is satisfiable.
- If **O(q(m))** is the time needed to determine whether a formula of length '**m**' is satisfiable then the complexity of '**Z**' is **O(p^3(n) log n + q(p^3(n)log n)).**
- If satisfiability is '**p**', then '**q(m)**' is a polynomial function of '**m**' and the complexity of '**Z**' becomes '**O(r(n))'** for some polynomial '**r()'.**
- Hence, if satisfiability is in **p**, then for every nondeterministic algorithm **A** in **NP**, we can obtain a deterministic **Z** in **p.**

By this we shows that satisfiability is in **p** then **P=NP**