UNIT – I

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.
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.

The study of Algorithms includes many important and active areas of research.

There are four distinct areas of study one can identify

1**. How to device algorithms**-

Creating an algorithm is an art which many never fully automated. A major goal is to study various design techniques that have proven to be useful. By mastering

these design strategies, it will become easier for you to device new and useful algorithms. some of techniques may already be familiar, and some have been found to be useful. Dynamic programming is one technique. Some of the techniques are especially useful in fields other than computer science such as operations research and electrical engineering.

2. How to validate algorithms:

Once an algorithm is devised, it is necessary to show that it computes the correct answer for all possible legal inputs. We refer to this process as **algorithm validation**. The algorithm need not as yet be expressed as a program. The purpose of validation is to assure us that this algorithm will work correctly independently. Once the validity of the method has been shown, a program can be written and a second phase begins. This phase is referred to as program proving or sometimes as **program verification**.

A proof of correctness requires that the solution be stated in two forms. One form is usually as a program which is annotated by a set of assertions about the input and output variables of the program. These assertions are often expressed in the predicate calculus. The second form is called a specification, and this may also be expressed in the predicate calculus. A complete proof of program correctness requires that each statement of a programming language be precisely defined and all basic operations be proved correct.

3. How to analyze algorithms:

As an algorithm is executed, it uses the computer's central processing unit (CPU) to perform operations and its memory to hold the program and data. Analysis of algorithms or performance analysis refers to the task of determining how much computing time and storage algorithms replace.we analyze the algorithm based on time and space complexity.The amount of time neede to run the

algorithm is called time complexity.The amount of memory neede to run the algorithm is called space complexity

4. How to test a program:

Testing a program consists of two phases

1. Debugging
2. Profiling

Debugging: It is the process of executing programs on sample data sets to determine whether faulty results occur and, if so to correct them. However, as E. Dijkstra has pointed out, “debugging can only point to the presence of errors, but not to the absence".

Profiling*:* Profiling or performance measurement is the process of executing a correct program on data sets and measuring the time and space it takes to compute the results.

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:**

   This method describe algorithms as program, which resembles language like Pascal & algol.

Pseudo-Code Conventions for expressing algorithms:

1. Comments begin with // and continue until the end of line.

2. Blocks are indicated with matching braces {and}.

3. An identifier begins with a letter. The data types of variables are not explicitly declared.

4. Compound data types can be formed with records. Here is an example,

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

Here link is a pointer to the record type node. Individual data items of a record can be accessed with → and period.

5. Assignment of values to variables is done using the assignment statement.

   <Variable>:= <expression>;

6. There are two Boolean values TRUE and FALSE.

   → Logical Operators AND, OR, NOT
   →Relational Operators <, <=,>,>=, =, !=

7. The following looping statements are employed.

   For, while and repeat-until

While Loop:

```
While < condition > do
{
        <statement-1>
              .
              .
```

```
                    .

            <statement-n>
        }
```

For Loop:

```
  For variable: = value-1 to value-2 step step do

{
  <statement-1>
        .
        .
        .
<statement-n>
}
```

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)

Examples:

→ algorithm for find max of two 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;
}
.
```

➔ Algorithm for Selection Sort:

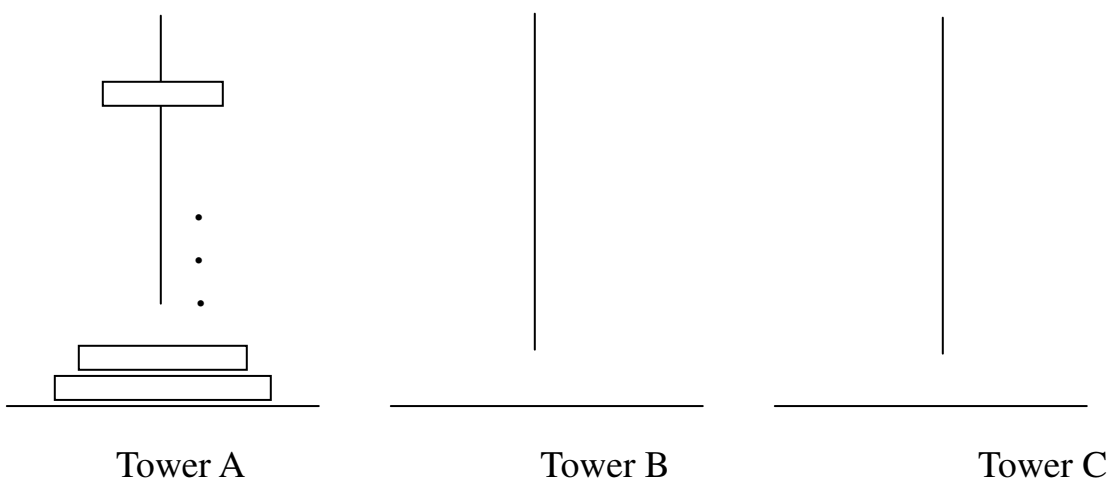
```
Algorithm selection sort (a,n)
      // Sort the array a[1:n] into non-decreasing order.
{
        for i:=1 to n do
        {
                j:=i;
                for k:=i+1 to n do
                        if (a[k]<a[j]) then j:=k;
                        t:=a[i];
                        a[i]:=a[j];
                        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:**



Towers of Hanoi is a problem in which there will be some disks which of decreasing sizes and were stacked on the tower in decreasing order of size bottom to top. Besides this there are two other towers (B and C) in which one tower will be act as destination tower and other act as intermediate tower. In this problem we have to move the disks from source tower to the destination tower. The conditions included during this problem are:

1) Only one disk should be moved at a time.
2) No larger disks should be kept on the smaller disks.

Consider an example to explain more about towers of Hanoi:

Consider there are three towers A, B, C and there will be three disks present in tower A. Consider C as destination tower and B as intermediate tower. The steps involved during moving the disks from A to B are

Step 1: Move the smaller disk which is present at the top of the tower A to C.
Step 2: Then move the next smallest disk present at the top of the tower A to B.
Step 3: Now move the smallest disk present at tower C to tower B
Step 4: Now move the largest disk present at tower A to tower C
Step 5: Move the disk smallest disk present at the top of the tower B to tower A.
Step 6: Move the disk present at tower B to tower C.
Step 7: Move the smallest disk present at tower A to tower C
In this way disks are moved from source tower to destination tower.

ALGORITHM FOR TOWERS OF HANOI:

```
Algorithm Towersofhanoi (n, X ,Y, Z)
        {
            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);
               }
      }
```

TIME COMPLEXITY OF TOWERS OF HANOI:

The recursive relation is:

```
t(n)=1;              if n=0
    =2t(n-1)+2   if n>=1
```

Solve the above recurrence relation then the time complexity of towers of Hanoi is O(2^n)

Performance Analysis:

1. **Space Complexity:**
   The space complexity of an algorithm is the amount of memory 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:

→ 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.

1. 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 1:

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

In this algorithm sp=0;let assume each variable occupies one word.
Then the space occupied by above algorithm is >=3.

S(P)>=3

Example 2:

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

In the above algoritm n,s and occupies one word each and array 'a' occupies n number of words so S(P)>=n+3

Example 3:

ALGORITHM FOR SUM OF NUMBERS USING RECURSION:

```
Algorithm RSum (a, n)
  {
      if(n<=0) then
          return 0.0;
      else
          return RSum(a,n-1)+a[n];
  }
```

The space complexity for above algorithm is:

In the above recursion algorithm the space need for the values of n, return address and pointer to array. The above recursive algorithm depth is (n+1). To each recursive call we require space for values of n, return address and pointer to array. So the total space occupied by the above algorithm is S(P) >= 3(n+1)

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.

->We can determine the number of steps needed by a program to solve a particular problem instance in Two ways.

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.

Example1:

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.

Example 2:

```
Algorithm RSum(a,n)
{

   count:=count+1;// For the if conditional

   if(n<=0)then

{

   count:=count+1; //For the return

   return 0.0;

}

else

{

   count:=count+1; //For the addition,function invocation and return

  return RSum(a,n-1)+a[n];

}

}
```

Example3:

ALGORITHM FOR MATRIX ADDITION

```
Algorithm Add(a,b,c,m,n)
{
 for i:=1 to m do
{
   count:=count+1; //For 'for i'
   for j:=1 to n do
  {
     count:=count+1; //For 'for j'
     c[i,j]=a[i,j]+b[i,j];
     count:=count+1; //For the assignment
}
count:=count+1; //For loop initialization and last time of 'for j'
```

}

count:=count+1; //For loop initialization and last time of 'for i'

If the count is zero to start with, then it will be 2mn+2m+1 on termination. So each invocation of sum execute a total of 2mn+2m+1 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.

Example 1:

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
6. return s;	1	1	1
7. }	0	-	0
Total			2n+3

step table for algorithm sum

Example 2:

		frequency		total steps	
Statements	s/e	n=0	n>0	n=0	n>0
1 algorithm Rsum(a,n)	0	_	_	0	

			0
2 {			
3 if(n<=0) then	1	1 1	1 1
4 return 0.0;	1	1 0	1 0
5 else return			
6 Rsum(a,n-1)+a[n];	1+x	0 1	0 1+x
7 }	0	_ _	0 0
Total			2 2+x

step table for algorithm recursive sum

Example 3:

Statements	s/e	frequency	total steps
1 Algorithm Add(a,b,c,m,n)	0	_	0
2 {	0	_	0
3 for i:=1 to m do	1	m+1	m+1
4 for j:=1 to n do	1	m(n+1)	mn+m
5 c[I,j]:=a[I,j]+b[I,j];	1	mn	mn
6 }	0	_	0
Total			2mn+2m+1

step table for matrix addition

Example 4:

Algorithm to find nth fibnocci number

```
Algorithm Fibonacci(n)
//Compute the nth Fibonacci number
{
   if(n<=1) then
     write (n);
  else
  {
     fnm2:=0;
     fnm1:=1;
    for i:=2 to n do
    {
       fn:=fnm1+fnm2;
```

```
    fnm:=fnm1;
   fnm1:=fn;
 }
 write(fn);
}
```

Asymptotic Notations:

The best algorithm can be measured by the efficiency of that algorithm.The efficiency of an algorithm is measured by computing time complexity.The asymptotic notations are used to find the time complexity of an algorithm.

Asymptotic notations gives fastest possible,slowest possible time and average time of the algorithm.

The basic asymptotic notations are Big-oh(O),Omega(Ω) and theta(Θ).

1:BIG-OH(O) NOTATION:

(i)It is denoted by 'O'.

(ii)It is used to find the upper bound time of an algorithm , that means the maximum time taken by the algorithm.

Definition : Let f(n),g(n) are two non-negative functions. If there exists two positive constants c ,n0 . such that c>0 and for all n>=n0 if f(n)<=c*g(n) then we say that f(n)=O(g(n))

THE GRAPH FOR BIG-OH (O) NOTATION:

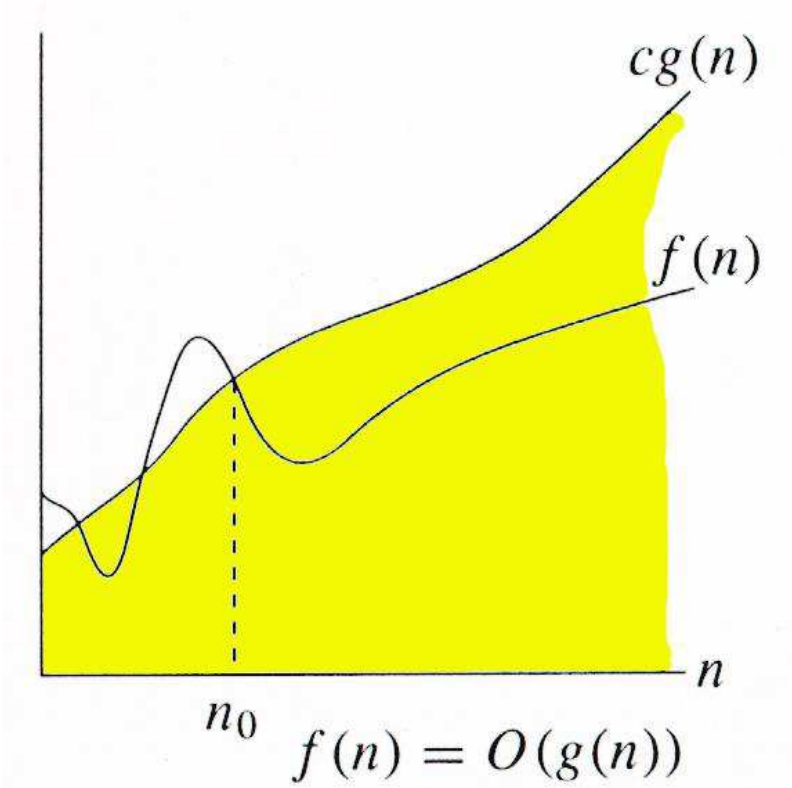


Figure 1

example : consider f(n)=2n+3 and g(n)=n^2
Sol : f(n)<=c*g(n)
let us assuming as c=1,
then f(n)<=g(n)
if n=1,
2n+3<=n^2 = 2(1)+3<=1^2 =>5<=1(false)
If n=2,
2n+3<=n^2=2(2)+3<=2^2= 7<=4(false)
if n=3,
2n+3<=n^2= 2(3)+3<=3^2=9<=9 (true)
if n=4,
2n+3<=n^2=>2(4)+3<=4^2=11<=6 (true)
if n=5,
2n+3<=n^2=2(5)+3<=5^2=13<=25 (true)
If n=6,2n+3<=n^2=2(6)+3<=6^2=15<=36 (true)
.:n>=3, f(n)=O(n^2) i.e, f(n)=O(g(n))

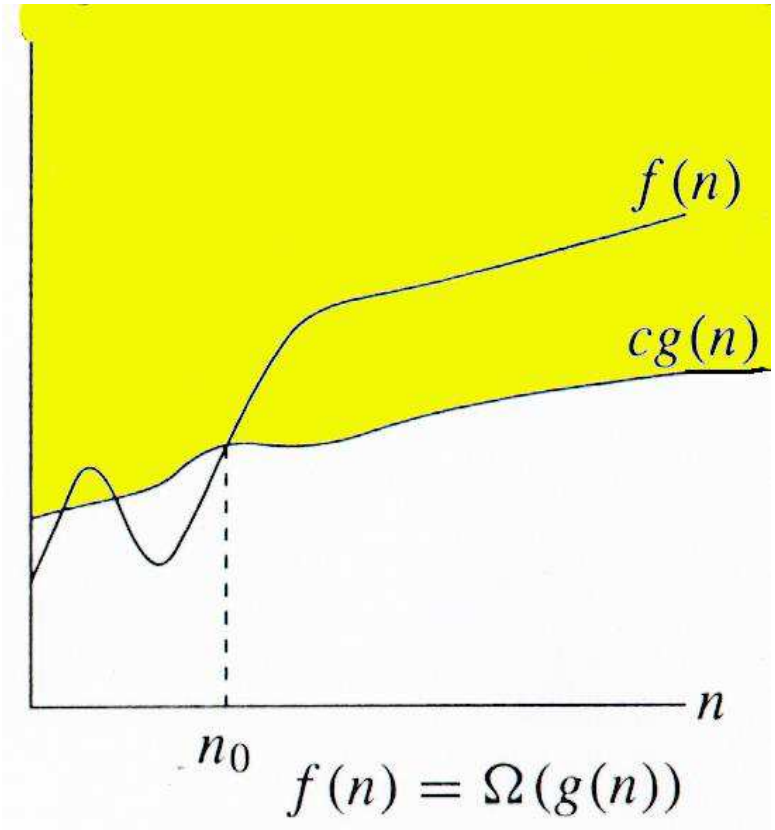
2:OMEGA (Ω) NOTATION:

(i)It is denoted by ' Ω'.

(ii)It is used to find the lower bound time of an algorithm, that means the minimum time taken by an algorithm.

Definition : Let f(n),g(n) are two non-negative functions. If there exists two positive constants c,n0.such that c>0 and for all n>=n0.if f(n)>=c*g(n) then we say that f(n)=Ω(g(n))

THE GRAPH FOR OMEGA NOTATION:



Example : consider f(n)=2n+5, g(n)=2n

Sol : Let us assume as c=1

If n=1:2n+5>=2n => 2(1)+5>=2(1) => 7>=2 (true)

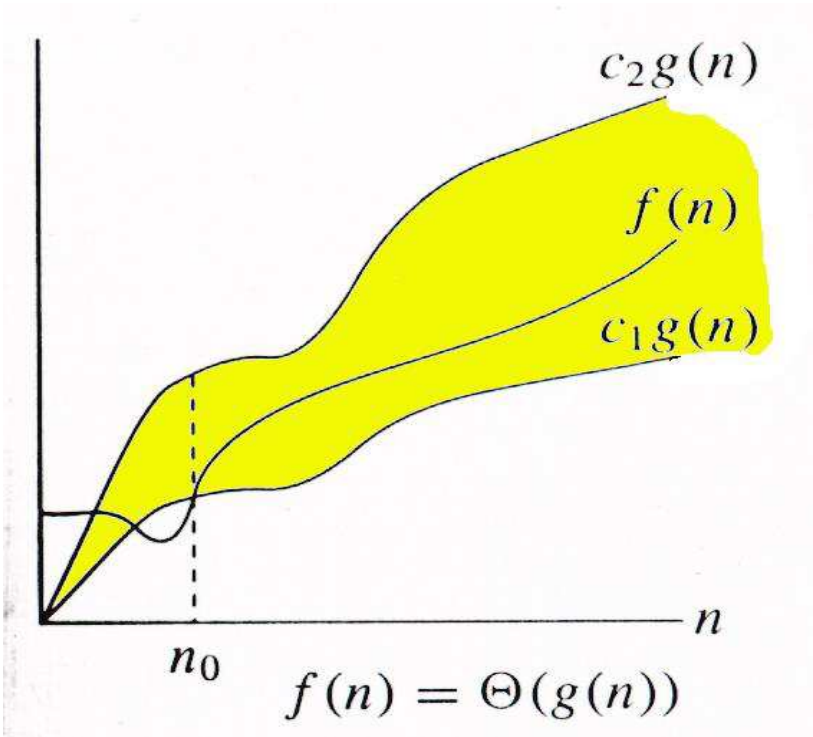
if n=2:2n+3>=2n=> 2(2)+5>=2(2)=> 9>=4 (true)
if n=3:2n+3>=2n=> 2(3)+5>=2(3)=> 11>=6 (true)
for all .:n>=1, f(n)=Ω(n) i.e , f(n)=Ω(g(n))

3:THETA (Θ) NOTATION:

(i)It is denoted by the symbol called as (Θ).

(ii)It is used to find the time in-between lower bound time and upper bound time of an algorithm.

Definition : Let f(n),g(n) are two non-negative functions. If there exists positive constants c1,c2,n0.such that c1>0,c2>0 and for all n>=n0.if c1*g(n)<=f(n)<=c2*g(n) then we say that f(n)=Θ(g(n))



Example : consider f(n)=2n+5, g(n)=n
Sol :c1*g(n)<=f(n)<=c2*g(n)
let us assuming as c1=3 then c1*g(n)=3n
if n=1,
3n<=2n+5=>3(1)<=2(1)+5=>3<=7 (true)
If n=2,
3n<=2n+5=>3(2)<=2(2)+5=>6<=9 (true)

If n=3,
3n<=2n+5=>3(3)<=2(3)+5=>9<=11 (true)

c2=4 c2*g(n)=4n
if n=1,
2n+5<=4n=>2(1)+5<=4(1)=>7<=4
If n=2,
2n+5<=4n=>2(2)+5<=4(2)=>9<=8
If n=3,
2n+5<=4n=>2(3)+5<=4(3)=>11<=12 (true)
If n=4,
2n+5<=4n=>2(4)+5<=4(4)=>13<=16 (true)
for all .:n>=3 f(n)=Θ(n) f(n)= Θ (g(n))

4:LITTLE-OH (O) NOTATION:

Definition : Let f(n),g(n) are two non-negative functions
if $\lim_{n\to\infty} [f(n) / g(n)] = 0$ then we say that f(n)=o(g(n))

example : consider f(n)=2n+3, g(n)=n^2
sol : let us
lim f(n)/g(n) = 0
n->∞
lim (2n+3) / (n^2)
n->∞
=lim n(2+(3/n)) / (n^2)
n->∞
=lim (2+(3/n)) /n
n->∞
=2/∞
=0
.:f(n)=o(n^2).

5:LITTLE OMEGA NOTATION:

Definition: Let f(n) and g(n) are two non-negative functions.
if lim g(n)/f(n) = 0 then we say that f(n)=ω(g(n))
n->∞
example : consider f(n)=n^2, g(n)=2n+5
sol : let us

$$\lim_{(n \to \infty)} g(n)/f(n) = 0$$
$$= \lim_{(n \to \infty)} (2n+5) / (n^2)$$
$$= \lim_{(n \to \infty)} n(2+(5/n)) / (n^2)$$
$$= \lim_{(n \to \infty)} (2+(5/n)) / n = 2/\infty = 0$$

.:f(n)= ω(n).

Amortized analysis:

Amortized analysis means finding average running time per operation over a worst case sequence of operations.

Suppose a sequence I1,I2,D1,I3,I4,I5,I6,D2,I7 of insert and delete operations is performed on a set.

Assume that the actual cost of each of the seven inserts is one and for delete operations D1 and D2 have an actual cost of 8 and 10 so the total cost of sequence of operations is 25.

In amortized scheme we charge some of the actual cost of an operation to other operations. This reduce the charge cost of some operations and increases the cost of other operations. The amortized cost of an operation is the total cost charge to it.

The only requirement is that the some of the amortized complexities of all operations in any sequence of operations be greater than or equal to their some of actual complexities i.e.,

$$\sum_{1 \le i \le n} amortized(i) \ge \sum_{1 \le i \le n} actual(i) \longrightarrow \quad (1)$$

Where amortized(i) and actual(i) denote the amortized and actual complexities of the ith operations in a sequence on n operations.

To define the potential function p(i) as:

p(i)=amortized(i)-actual(i)+p(i-1) $\longrightarrow$ (2)

If we sum equation (2) for 1≤i≤n we get

$\sum_{1\le i\le n} p(i) = \sum_{1\le i\le n}(amortized(i) - actual(i) + p(i-1))$

$$\sum_{1\le i\le n} p(i) - \sum_{1\le i\le n} p(i-1) = \sum_{1\le i\le n}(amortized(i) - actual(i))$$

P (n)-p (0) = $\sum_{1\le i\le n}(amortized(i) - actual(i))$

From equation (1) we say that

P (n)-p (0) ≥0 $\longrightarrow$ (3)

Under assumption p(0)=0,p(i) is the amount by which the first 'i' operations have been over charged (i.e., they have been charged more than the actual cost).

The methods to find amortized cost for operations are:

1. Aggregate method.
2. Accounting method.
3. Potential method.

1. Aggregate method:

The amortized cost of each operation is set equal to Upper Bound On Sum Of Actual Costs(n)/n.

2. Accounting method:

In this method we assign amortized cost to the operations (possibly by guessing what assignment will work),compute the p(i) using equation(2) and show that p(n)-p(0)>=0.

3.Potential method:

Here we start with potential function that satisfies equation(3) and compute amortized complexities using equation(2).

Example:
Let assume we pay $50 for each month other than March, June, September, and December $100 for every June, September. calculate cost by using aggregate, accounting and potential method .

Month	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**
Actual cost:	**50**	**50**	**100**	**50**	**50**	**100**	**50**	**50**	**100**	**50**	**50**	**200**	**50**	**50**	**100**	**50**
Amortized cost:	**75**	**75**	**75**	**75**	**75**	**75**	**75**	**75**	**75**	**75**	**75**	**75**	**75**	**75**	**75**	**75**
P( ):	**25**	**50**	**25**	**50**	**75**	**50**	**75**	**100**	**75**	**100**	**125**	**0**	**25**	**50**	**25**	**50**

Aggregate Method:

$=200 \times \lfloor n/12 \rfloor + 100(\lfloor n/3 \rfloor - \lfloor n/12 \rfloor) + 50(n - \lfloor n/3 \rfloor)$
$=100 \times \lfloor n/12 \rfloor + 50 \lfloor n/3 \rfloor + 50n$
$\leq 100 \times (n/12) + 50 \times (n/3) + 50 \times n$
$=50\, n\, ((1/6) + (1/3) + (1))$
$=50\, n\, ((1+2+6)/6)$
$=50\, n\, (9/6)$
$=75n.$

In the above problem the actual cost for 'n' months does not exceed 200n from the aggregate method the amortized cost for 'n' months does not exceed $75. The amortized cost for each month is set to $75.
Let assume p(0)=0 the potential for each and every month.

Accounting method:
From the above table we see that using any cost less than $75 will result in $p(n)-p(0) \leq 0$.
The amortized cost must be ≥ 75.
If the amortized cost ≤ 75 then only the condition $p(n)-p(0) <= 0$.
Potential method:
To the given problem we start with the potential function as:
P (n) =0 n mod 12=0
P (n) =25 n mod 12=1 or 3

P (n) =50 n mod 12=4, 6, 2
P (n) =75 n mod 12=5, 7, 9
P (n) =100 n mod 12=8, 10
P (n) =125 n mod 12=4

From the above potential function the amortized cost for operation is evaluated for amortized(i)=p(i)-p(i-1)+actual(i).

Probabilistic analysis:

In probabilistic analysis we analyze the algorithm for finding efficiency of the algorithm.The efficiency of algorithm is also depend upon distribution of inputs.In this we analyze algorithm by the concept of probability.

For example the company wants to recruiting k persons from the n persons.To do this the company assigns ranking to all n persons depend upon their performance.The rankings of n persons from r_1 to r_n.To n persons we get n! permutations out of n! permutations the company selects any one combination that is from r_1 to r_k

UNIT-II

Divide and Conquer

General Method

Divide and conquer is a design strategy which is well known to breaking down efficiency barriers. When the method applies, it often leads to a large improvement in time complexity. For example, from O (n^2) to O (n log n) to sort the elements.

Divide and conquer strategy is as follows: divide the problem instance into two or more smaller instances of the same problem, solve the smaller instances recursively, and assemble the solutions to form a solution of the original instance. The recursion stops when an instance is reached which is too small to divide. When dividing the instance, one can either use whatever division comes most easily to hand or invest time in making the division carefully so that the assembly is simplified.

Divide and conquer algorithm consists of two parts:

Divide : Divide the problem into a number of sub problems. The sub problems are solved recursively.

Conquer : The solution to the original problem is then formed from the solutions to the sub problems (patching together the answers).

Traditionally, routines in which the text contains at least two recursive calls are called divide and conquer algorithms, while routines whose text contains only one recursive call are not. Divide–and–conquer is a very powerful use of recursion.

Control Abstraction of Divide and Conquer

A control abstraction is a procedure whose flow of control is clear but whose primary operations are specified by other procedures whose precise meanings are left undefined. The control abstraction for divide and conquer technique is DANDC(P), where P is the problem to be solved.

```
DANDC (P)
{
        if SMALL (P) then return S (p);
        else
        {
                divide p into smaller instances p1, p2, .... Pk, k ≥ 1;
                apply DANDC to each of these sub problems;
                return (COMBINE (DANDC (p1) , DANDC (p2),...., DANDC (pk));
        }
}
```

SMALL (P) is a Boolean valued function which determines whether the input size is small enough so that the answer can be computed without splitting. If this is so function 'S' is invoked otherwise, the problem 'p' into smaller sub problems. These sub problems p_1, p_2, . . . , p_k are solved by recursive application of DANDC.

If the sizes of the two sub problems are approximately equal then the computing time of DANDC is:

$$T(n) = \begin{cases} g(n) & n \text{ small} \\ 2T(n/2) + f(n) & \text{otherwise} \end{cases}$$

Where, T (n) is the time for DANDC on 'n' inputs
g (n) is the time to complete the answer directly for small inputs and
f (n) is the time for Divide and Combine

Binary Search

If we have 'n' records which have been ordered by keys so that $x_1 < x_2 < \ldots < x_n$. When we are given a element 'x', binary search is used to find the corresponding element from the list. In case 'x' is present, we have to determine a value 'j' such that a[j] = x (successful search). If 'x' is not in the list then j is to set to zero (un successful search).

In Binary search we jump into the middle of the file, where we find key a[mid], and compare 'x' with a[mid]. If x = a[mid] then the desired record has been found. If x < a[mid] then 'x' must be in that portion of the file that precedes a[mid], if there at all. Similarly, if a[mid] > x, then further search is only necessary in that past of the file which follows a[mid]. If we use recursive procedure of finding the middle key a[mid] of the un-searched portion of a file, then every un-successful comparison of 'x' with a[mid] will eliminate roughly half the un-searched portion from consideration.

Since the array size is roughly halved often each comparison between 'x' and a[mid], and since an array of length 'n' can be halved only about $\log_2 n$ times before reaching a trivial length, the worst case complexity of Binary search is about $\log_2 n$

Algorithm Algorithm

```
BINSRCH (a, n, x)
 // array a(1 : n) of elements in increasing order, n ≥ 0,
 // determine whether 'x' is present, and if so, set j such that x = a(j)
 // else return j

 {
        low :=1 ; high :=n ;
        while (low ≤ high) do
        {
                mid :=|(low + high)/2|
                if (x < a [mid]) then high:=mid – 1;
                else if (x > a [mid]) then low:= mid + 1
                        else return mid;
        }
        return 0;
 }
```

low and *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 by at least one. Thus we have two sequences of integers approaching each other and eventually *low* will become greater than *high* causing termination in a finite number of steps if 'x' is not present.

Example for Binary Search

Let us illustrate binary search on the following 9 elements:

Index	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
Elements	-15	-6	0	7	9	23	54	82	101

The number of comparisons required for searching different elements is as follows:

1. Searching for x = 101

low	high	mid
1	9	5
6	9	7
8	9	8
9	9	9
		found

Number of comparisons = 4

2. Searching for x = 82

low	high	mid
1	9	5
6	9	7
8	9	8
		found

Number of comparisons = 3

3. Searching for x = 42

low	high	mid
1	9	5
6	9	7
6	6	6
7	6	not found

Number of comparisons = 4

4. Searching for x = -14

low	high	mid
1	9	5
1	4	2
1	1	1
2	1	not found

Number of comparisons = 3

Continuing in this manner the number of element comparisons needed to find each of nine elements is:

Index	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
Elements	-15	-6	0	7	9	23	54	82	101
Comparisons	*3*	*2*	*3*	*4*	*1*	*3*	*2*	*3*	*4*

No element requires more than 4 comparisons to be found. Summing the comparisons needed to find all nine items and dividing by 9, yielding 25/9 or approximately 2.77 comparisons per successful search on the average.

There are ten possible ways that an un-successful search may terminate depending upon the value of x.

If x < a[1], a[1] < x < a[2], a[2] < x < a[3], a[5] < x < a[6], a[6] < x < a[7] or a[7] < x < a[8] the algorithm requires 3 element comparisons to determine that 'x' is not present. For all of the remaining possibilities BINSRCH requires 4 element comparisons. Thus the average number of element comparisons for an unsuccessful search is:

(3 + 3 + 3 + 4 + 4 + 3 + 3 + 3 + 4 + 4) / 10 = 34/10 = 3.4

The time complexity for a successful search is O(log n) and for an unsuccessful search is Θ(log n).

Successful searches			**un-successful searches**
Θ(1),	Θ(log n),	Θ(log n)	Θ(log n)
Best	average	worst	best, average and worst

Analysis for worst case

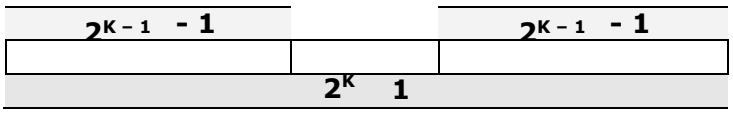
Let T (n) be the time complexity of Binary search

The algorithm sets mid to [n+1 / 2]

Therefore,

$$
\begin{aligned}
T(0) &= 0 \\
T(n) &= 1 && \text{if } x = a[mid] \\
&= 1 + T([(n+1)/2] - 1) && \text{if } x < a[mid] \\
&= 1 + T(n - [(n+1)/2]) && \text{if } x > a[mid]
\end{aligned}
$$

Let us restrict 'n' to values of the form $n = 2^K - 1$, where 'k' is a non-negative integer. The array always breaks symmetrically into two equal pieces plus middle element.



Algebraically this is $\left\lceil \frac{n+1}{2} \right\rceil = \left\lceil \frac{2^K - 1 + 1}{2} \right\rceil = 2^{K-1}$ for K > 1

Giving,

$$
\begin{aligned}
T(0) &= 0 \\
T(2^k - 1) &= 1 && \text{if } x = a[mid] \\
&= 1 + T(2^{K-1} - 1) && \text{if } x < a[mid] \\
&= 1 + T(2^{k-1} - 1) && \text{if } x > a[mid]
\end{aligned}
$$

In the worst case the test x = a[mid] always fails, so

$w(0) = 0$

$w(2^k - 1) = 1 + w(2^{k-1} - 1)$

This is now solved by repeated substitution:

$$
\begin{aligned}
w(2^k - 1) &= 1 + w(2^{k-1} - 1) \\
&= 1 + [1 + w(2^{k-2} - 1)] \\
&= 1 + [1 + [1 + w(2^{k-3} - 1)]] \\
&= \ldots\ldots\ldots \\
&= \ldots\ldots\ldots \\
&= i + w(2^{k-i} - 1)
\end{aligned}
$$

For $i \leq k$, letting $i = k$ gives $w(2^k - 1) = K + w(0) = k$

But as $2^K - 1 = n$, so $K = \log_2(n + 1)$, so

$$w(n) = \log_2(n + 1) = O(\log n)$$

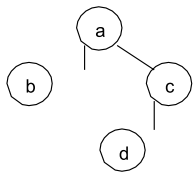
for $n = 2^K - 1$, concludes this analysis of binary search.

Although it might seem that the restriction of values of 'n' of the form $2^K - 1$ weakens the result. In practice this does not matter very much, w(n) is a monotonic increasing function of 'n', and hence the formula given is a good approximation even when 'n' is not of the form $2^K - 1$.

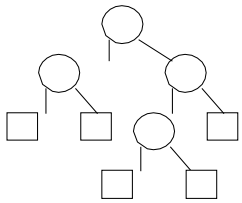
External and Internal path length:

The lines connecting nodes to their non-empty sub trees are called edges. A non-empty binary tree with n nodes has n–1 edges. The size of the tree is the number of nodes it contains.

When drawing binary trees, it is often convenient to represent the empty sub trees explicitly, so that they can be seen. For example:

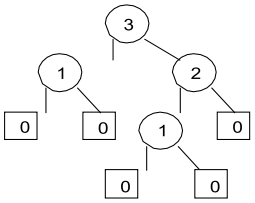


The tree given above in which the empty sub trees appear as square nodes is as follows:

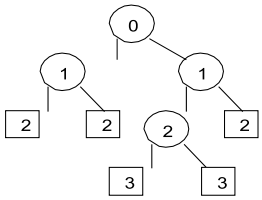


The square nodes are called as external nodes E(T). The square node version is sometimes called an extended binary tree. The round nodes are called internal nodes I(T). A binary tree with n internal nodes has n+1 external nodes.

The height h(x) of node 'x' is the number of edges on the longest path leading down from 'x' in the extended tree. For example, the following tree has heights written inside its nodes:



The depth d(x) of node 'x' is the number of edges on path from the root to 'x'. It is the number of internal nodes on this path, excluding 'x' itself. For example, the following tree has depths written inside its nodes:



The internal path length I(T) is the sum of the depths of the internal nodes of 'T':

$$I(T) = \sum_{x \in I(T)} d(x)$$

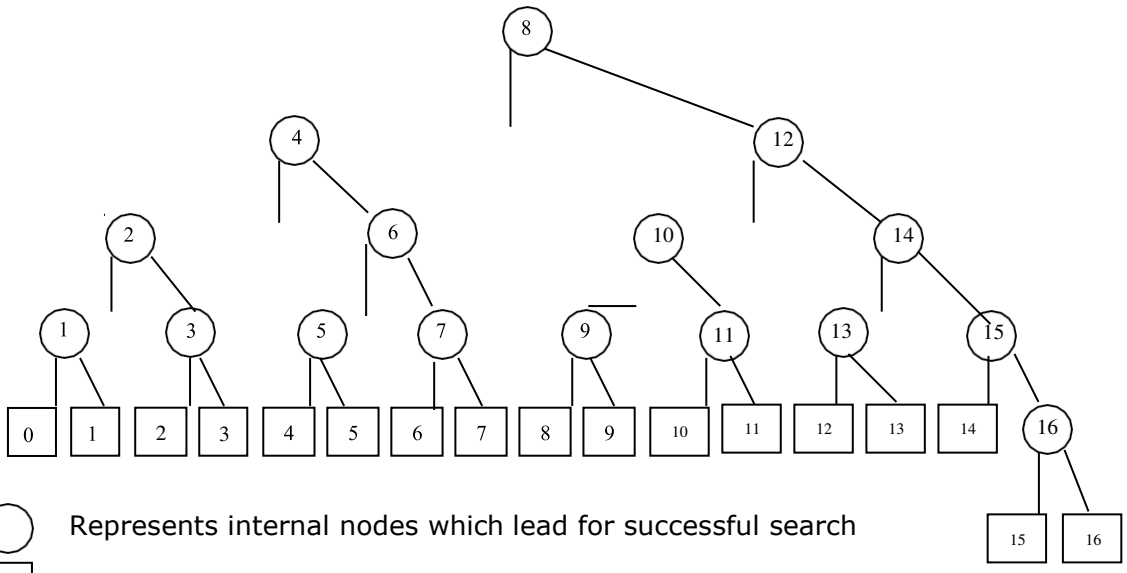
The external path length E(T) is the sum of the depths of the external nodes:

$$E(T) = \sum_{x \in E(T)} d(x)$$

For example, the tree above has I(T) = 4 and E(T) = 12.

A binary tree T with 'n' internal nodes, will have I(T) + 2n = E(T) external nodes.

A binary tree corresponding to binary search when n = 16 is



Represents internal nodes which lead for successful search

External square nodes, which lead for unsuccessful search.

Let C_N be the average number of comparisons in a successful search.

C'_N be the average number of comparison in an un successful search.

Then we have,

$$C_N = 1 + \frac{\text{internal pathlengthoftree}}{N}$$

$$C'_N = \frac{\text{External path length of tree}}{N+1}$$

$$C_N = \left(1 + \frac{1}{N}\right) C'_N - 1$$

External path length is always 2N more than the internal path length.

Merge Sort

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

This strategy is so simple, and so efficient but the problem here is that there seems to be no easy way to merge two adjacent sorted arrays together in place (The result must be build up in a separate array).

The fundamental operation in this algorithm is merging two sorted lists. Because the lists are sorted, this can be done in one pass through the input, if the output is put in a third list.

The basic merging algorithm takes two input arrays 'a' and 'b', an output array 'c', and three counters, *a ptr, b ptr* and *c ptr,* which are initially set to the beginning of their respective arrays. The smaller of *a[a ptr]* and *b[b ptr]* is copied to the next entry in 'c', and the appropriate counters are advanced. When either input list is exhausted, the remainder of the other list is copied to'c'.

To illustrate how merge process works. For example, let us consider the array 'a' containing 1, 13, 24, 26 and 'b' containing 2, 15, 27, 38. First a comparison is done between 1 and 2. 1 is copied to 'c'. Increment *a ptr* and *c ptr.*

1	*2*	*3*	*4*
1	13	24	26
h ptr			

5	*6*	*7*	*8*
2	15	27	28
j ptr			

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*
1							
i ptr							

and then 2 and 13 are compared. 2 is added to 'c'. Increment *b ptr* and *c ptr.*

1	*2*	*3*	*4*
1	13	24	26
	h ptr		

5	*6*	*7*	*8*
2	15	27	28
j ptr			

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*
1	2						
	i ptr						

then 13 and 15 are compared. 13 is added to ‘c’. Increment *a ptr* and *c ptr*.

1	*2*	*3*	*4*
1	13	24	26
	h ptr		

5	*6*	*7*	*8*
2	15	27	28
	j ptr		

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*
1	2	13					
		i ptr					

24 and 15 are compared. 15 is added to ‘c’. Increment *b ptr* and *c ptr*.

1	*2*	*3*	*4*
1	13	24	26
		h ptr	

5	*6*	*7*	*8*
2	15	27	28
	j ptr		

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*
1	2	13	15				
			i ptr				

24 and 27 are compared. 24 is added to ‘c’. Increment *a ptr* and *cptr*.

1	*2*	*3*	*4*
1	13	24	26
		h ptr	

5	*6*	*7*	*8*
2	15	27	28
		j ptr	

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*
1	2	13	15	24			
				i ptr			

26 and 27 are compared. 26 is added to ‘c’. Increment *a ptr* and *cptr.*

1	*2*	*3*	*4*
1	13	24	26
			h ptr

5	*6*	*7*	*8*
2	15	27	28
		j ptr	

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*
1	2	13	15	24	26		
					i ptr		

As one of the lists is exhausted. The remainder of the b array is then copied to ‘c’.

1	*2*	*3*	*4*
1	13	24	26

h ptr

5	*6*	*7*	*8*
2	15	27	28
		j ptr	

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*
1	2	13	15	24	26	27	28

i ptr

Algorithm

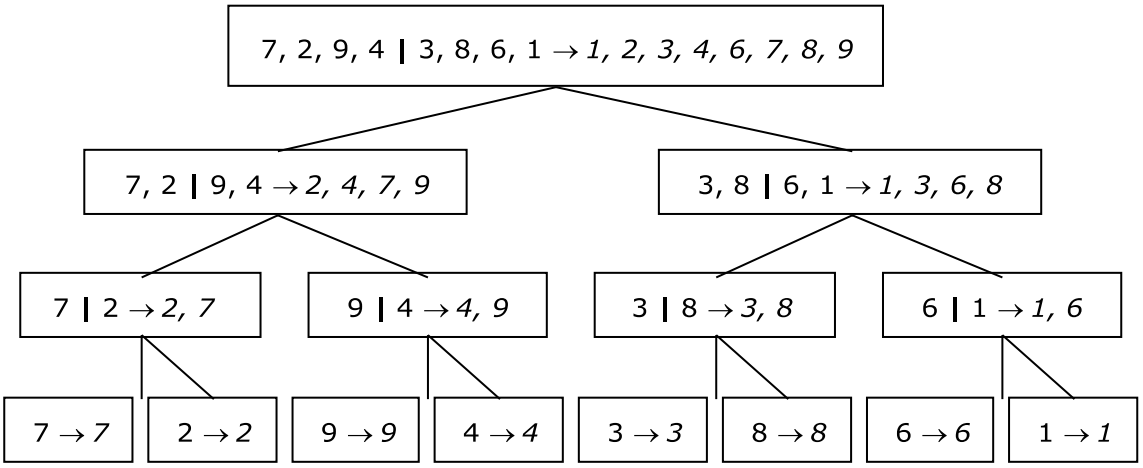
Algorithm MERGESORT (low, high)

```
// a (low : high) is a global array to be sorted.
{
        if (low < high)
        {
                mid := ⌊(low  + high)/2⌋          //finds where to split the set
                MERGESORT(low,  mid)              //sort one subset
                MERGESORT(mid+1, high) //sort the other subset
                MERGE(low,  mid,  high)           // combine the results
        }
}
```

```
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 objective is to merge these sorted sets into single sorted
// set residing in a (low : high). An auxiliary array B is used.
{
        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 + l;
                }
        for k := low to high do
                a[k] := b[k];
}
```

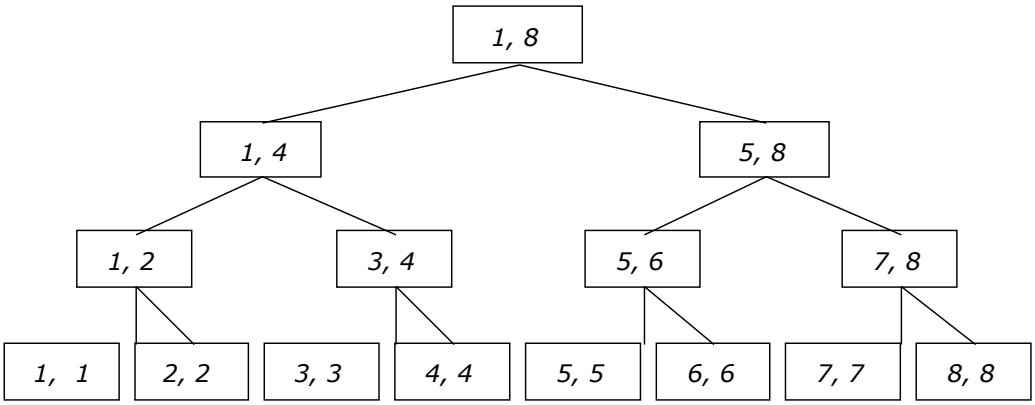
Example

For example let us select the following 8 entries 7, 2, 9, 4, 3, 8, 6, 1 to illustrate merge sort algorithm:



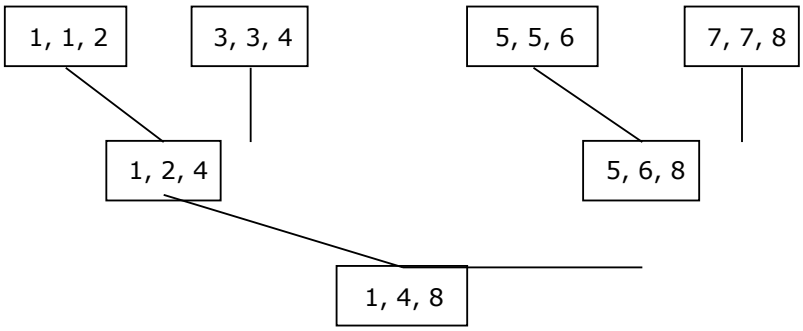
Tree Calls of MERGESORT(1, 8)

The following figure represents the sequence of recursive calls that are produced by MERGESORT when it is applied to 8 elements. The values in each node are the values of the parameters low and high.



Tree Calls of MERGE()

The tree representation of the calls to procedure MERGE by MERGESORT is as follows:



Analysis of Merge Sort

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

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

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

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

We have, $T(n) = 2T(n/2) + n$

Since we can substitute n/2 into this main equation

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

We have,

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

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

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

So we have,

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

Continuing in this manner, we obtain:

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

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

$$T\,(n) = 2^{\log_2 n}\,T\left(\frac{2^k}{2^k}\right) + \log_2 n\,.\,n$$
$$= n\,T(1) + n \log n$$
$$= n \log n + n$$

Representing this in O notation:

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

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

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

Strassen's Matrix Multiplication:

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

The usual way to multiply two n x n matrices A and B, yielding result matrix 'C' as follows :

```
for i := 1 to n do
        for j :=1 to n do
                c[i, j] := 0;
                for K: = 1 to n do
                        c[i, j] := c[i, j] + a[i, k] * b[k, j];
```

This algorithm requires n^3 scalar multiplication's (i.e. multiplication of single numbers) and n^3 scalar additions. So we naturally cannot improve upon.

We apply divide and conquer to this problem. For example let us considers three multiplication like this:

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

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

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

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

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

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

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

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

Which leads to $T(n) = O(n^3)$, where n is the power of 2.

Strassens insight was to find an alternative method for calculating the C_{ij}, requiring seven (n/2) x (n/2) matrix multiplications and eighteen (n/2) x (n/2) matrix additions and subtractions:

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

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

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

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

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

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

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

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

$C_{12} = R + T$

$C_{21} = Q + S$

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

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

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

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

$$T(2^k) = 7\,T(2^{k-1})$$
$$= 7^2\,T(2^{k-2})$$
$$= \text{- - - - - -}$$
$$= \text{- - - - - -}$$
$$= 7^i\,T(2^{k-i})$$

Put i = k

$$= 7^k\,T(1)$$
$$= 7^k$$

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

$$= n^{\log_2 7}$$
$$= O(n^{\log_2 7}) = O(^2n^{.81})$$

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

Quick Sort

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

Hoare his devised a very efficient way of implementing this idea in the early 1960's that improves the $O(n^2)$ behavior of SIS algorithm with an expected performance that is O(n log n).

In essence, the quick sort algorithm partitions the original array by rearranging it into two groups. The first group contains those elements less than some arbitrary chosen value taken from the set, and the second group contains those elements greater than or equal to the chosen value.

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

The function partition() makes use of two pointers 'i' and 'j' which are moved toward each other in the following fashion:

- Repeatedly increase the pointer 'i' until a[i] >= pivot.
- Repeatedly decrease the pointer 'j' until a[j] <= pivot.

- If j > i, interchange a[j] with a[i]
- Repeat the steps 1, 2 and 3 till the 'i' pointer crosses the 'j' pointer. If 'i' pointer crosses 'j' pointer, the position for pivot is found and place pivot element in 'j' pointer position.

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

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

Algorithm Algorithm

```
QUICKSORT(low, high)
/* sorts the elements a(low), . . . . . , a(high) which reside in the global array    A(1 :
n) into ascending order a (n + 1) is considered to be defined and must be greater
than all elements in a(1 : n); A(n + 1) = + ∝ */
{
        if low < high then
        {
                j := PARTITION(a, low, high+1);
                                        // J is the position of the partitioning element
                QUICKSORT(low, j – 1);
                QUICKSORT(j + 1 , high);
        }
}

Algorithm PARTITION(a, m, p)
{
        V ← a(m); i ← m; j ← p;                    // A (m) is the partition element
        do
        {
                loop  i  := i   + 1  until  a(i) ≥ v            // i moves left to right
                loop  j  := j  – 1  until  a(j)  ≤ v            // p moves right to left
                if (i < j) then INTERCHANGE(a, i, j)
         } while (i ≥ j);
        a[m] :=a[j]; a[j] :=V; // the partition element belongs at position P
        return j;
}
```

```
Algorithm INTERCHANGE(a, i, j)
{
        P:=a[i];
        a[i] := a[j];
        a[j] := p;
}
```

Example

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

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

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*Remarks*
38	08	16	06	79	57	24	56	02	58	04	70	45	
pivot				i						j			swap i & j
				04						79			
					i			j					swap i & j
					02			57					
						j	i						
(24	08	16	06	04	02)	**38**	(56	57	58	79	70	45)	swap pivot & j
pivot					j, i								swap pivot & j
(02	08	16	06	04)	**24**								
pivot, j	i												swap pivot & j
02	(08	16	06	04)									
	pivot	i		j									swap i & j
		04		16									
			j	i									
	(06	04)	**08**	(16)									swap pivot & j
	pivot, j	i											
	(04)	**06**											swap pivot & j
	04 pivot, j, i												
				16 pivot, j, i									
(02	**04**	**06**	**08**	**16**	**24)**	38							
							(56	57	58	79	70	45)	

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

Analysis of Quick Sort:

Like merge sort, quick sort is recursive, and hence its analysis requires solving a recurrence formula. We will do the analysis for a quick sort, assuming a random pivot (and no cut off for small files).

We will take $T(0) = T(1) = 1$, as in merge sort.

The running time of quick sort is equal to the running time of the two recursive calls plus the linear time spent in the partition (The pivot selection takes only constant time). This gives the basic quick sort relation:

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

Where, $i = |S_1|$ is the number of elements in S_1.

Worst Case Analysis

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

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

Using equation – (1) repeatedly, thus

T (n – 1) = T (n – 2) + C (n – 1)

T (n – 2) = T (n – 3) + C (n – 2)

- - - - - - - -

T (2) = T (1) + C (2)

Adding up all these equations yields

$$T(n) = T(1) + \sum_{i=2}^{n} i$$

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

Best Case Analysis

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

T (n) = 2 T (n/2) + C n - (4)

Divide both sides by n

$$\frac{T(n)}{n} = \frac{T(n/2)}{n/2} + C \qquad - \qquad (5)$$

Substitute n/2 for 'n' in equation (5)

$$\frac{T(n/2)}{n/2} = \frac{T(n/4)}{n/4} + C \qquad - \qquad (6)$$

Substitute n/4 for 'n' in equation (6)

$$\frac{T(n/4)}{n/4} = \frac{T(n/8)}{n/8} + C \qquad - \qquad (7)$$

- - - - - - - -

- - - - - - - -

Continuing in this manner, we obtain:

$$\frac{T(2)}{2} = \frac{T(1)}{1} + C \qquad - \qquad (8)$$

We add all the equations from 4 to 8 and note that there are log n of them:

$$\frac{T(n)}{n} = \frac{T(1)}{1} + C \log n \qquad - \qquad (9)$$

Which yields, T (n) = C n log n + n = **O(n log n)** - (10)

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

Average Case Analysis

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

T(n) = comparisons for first call on quicksort
+
{Σ 1<=nleft,nright<=n [T(nleft) + T(nright)]}n = (n+1) + 2 [T(0) +T(1) + T(2) + ----- + T(n-1)]/n

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

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

Subtracting both sides:

nT(n) –(n-1)T(n-1) = [n(n+1) – (n-1)n] + 2T(n-1) = 2n + 2T(n-1)
nT(n) = 2n + (n-1)T(n-1) + 2T(n-1) = 2n + (n+1)T(n-1)
T(n) = 2 + (n+1)T(n-1)/n
The recurrence relation obtained is:
T(n)/(n+1) = 2/(n+1) + T(n-1)/n

Using the method of subsititution:

T(n)/(n+1) = 2/(n+1) + T(n-1)/n
T(n-1)/n = 2/n + T(n-2)/(n-1)
T(n-2)/(n-1) = 2/(n-1) + T(n-3)/(n-2)
T(n-3)/(n-2) = 2/(n-2) + T(n-4)/(n-3)
. .
. .
T(3)/4 = 2/4 + T(2)/3
T(2)/3 = 2/3 + T(1)/2 T(1)/2 = 2/2 + T(0)
Adding both sides:
T(n)/(n+1) + [T(n-1)/n + T(n-2)/(n-1) + ---------- + T(2)/3 + T(1)/2]
= [T(n-1)/n + T(n-2)/(n-1) + ---------- + T(2)/3 + T(1)/2] + T(0) +
[2/(n+1) + 2/n + 2/(n-1) + ---------- +2/4 + 2/3]
Cancelling the common terms:
T(n)/(n+1) = 2[1/2 +1/3 +1/4+----------+1/n+1/(n+1)]

T(n) = (n+1)2[$\sum_{2 \le k \le n+1} 1/k$

=2(n+1) [–]
=2(n+1)[log (n+1) – log 2]
=2n log (n+1) + log (n+1)-2n log 2 –log 2
T(n)= O(n log n)

3.8. Straight insertion sort:

Straight insertion sort is used to create a sorted list (initially list is empty) and at each iteration the top number on the sorted list is removed and put into its proper

place in the sorted list. This is done by moving along the sorted list, from the smallest to the largest number, until the correct place for the new number is located i.e. until all sorted numbers with smaller values comes before it and all those with larger values comes after it. For example, let us consider the following 8 elements for sorting:

Index	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*
Elements	27	412	71	81	59	14	273	87

Solution:

Iteration 0:	unsorted	412	71	81	59	14	273	87	
	Sorted	27							
Iteration 1:	unsorted	412	71	81	59	14	273	87	
	Sorted	27	412						
Iteration 2:	unsorted	71	81	59	14	273	87		
	Sorted	27	71	412					
Iteration 3:	unsorted	81	39	14	273	87			
	Sorted	27	71	81	412				
Iteration 4:	unsorted	59	14	273	87				
	Sorted	274	59	71	81	412			
Iteration 5:	unsorted	14	273	87					
	Sorted	14	27	59	71	81	412		
Iteration 6:	unsorted	273	87						
	Sorted	14	27	59	71	81	273	412	
Iteration 7:	unsorted	87							
	Sorted	14	27	59	71	81	87	273	412

UNIT 3

Greedy Method

GENERAL METHOD

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

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

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

CONTROL ABSTRACTION

Algorithm Greedy (a, n)

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

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

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

KNAPSACK PROBLEM

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

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

$$\text{maximize} \quad \sum_{i=1}^{n} p_i x_i$$

$$\text{subject to} \quad \sum_{i=1}^{n} a_i x_i \le M \qquad \text{where, } 0 \le x_i \le 1 \text{ and } 1 \le i \le n$$

The profits and weights are positive numbers.

Algorithm

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

Algorithm GreedyKnapsack (m, n)

```
// P[1 : n] and w[1 : n] contain the profits and weights respectively of
// Objects ordered so that p[i] / w[i] > p[i + 1] / w[i + 1].
// m is the knapsack size and x[1: n] is the solution vector.
{
        for i := 1 to n do x[i]  := 0.0                  // initialize x
        U := m;
        for i := 1 to n do
        {
                if  (w(i) > U) then break;
                x [i] := 1.0; U := U – w[i];
        }
        if (i ≤ n) then x[i] := U / w[i];
}
```

Running time:

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

Example:

Consider the following instance of the knapsack problem: n = 3, m = 20, (p_1, p_2, p_3) = (25, 24, 15) and (w_1, w_2, w_3) = (18, 15, 10).

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

x_1	x_2	x_3	$w_i\ x_i$	$p_i\ x_i$
1/2	1/3	1/4	18 x 1/2 + 15 x 1/3 + 10 x 1/4 = 16.5	25 x 1/2 + 24 x 1/3 + 15 x 1/4 = 24.25

2. Select the object with the maximum profit first (p = 25). So, x_1 = 1 and profit earned is 25. Now, only 2 units of space is left, select the object with next largest profit (p = 24). So, x_2 = 2/15

x_1	x_2	x_3	$w_i\ x_i$	$p_i\ x_i$
1	2/15	0	18 x 1 + 15 x 2/15 = 20	25 x 1 + 24 x 2/15 = 28.2

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

x_1	x_2	x_3	$w_i\ x_i$	$p_i\ x_i$
0	2/3	1	15 x 2/3 + 10 x 1 = 20	24 x 2/3 + 15 x 1 = 31

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

p_1/w_1	**p_2/w_2**	**p_3/w_3**
25/18	24/15	15/10
1.4	1.6	1.5

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

x_1	x_2	x_3	$w_i\ x_i$	$p_i\ x_i$
0	1	1/2	15 x 1 + 10 x 1/2 = 20	24 x 1 + 15 x 1/2 = 31.5

This solution is the optimal solution.

JOB SEQUENCING WITH DEADLINES

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

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

Example:

Let n = 4, (P_1, P_2, P_3, P_4,) = (100, 10, 15, 27) and (d_1 d_2 d_3 d_4) = (2, 1, 2, 1). The feasible solutions and their values are:

S. No	Feasible Solution	Procuring sequence	Value	Remarks
1	1,2	2,1	110	
2	1,3	1,3 or 3,1	115	
3	1,4	4,1	127	**OPTIMAL**
4	2,3	2,3	25	
5	3,4	4,3	42	
6	1	1	100	
7	2	2	10	
8	3	3	15	
9	4	4	27	

Algorithm:

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

Algorithm GreedyJob (d, J, n)

```
// J is a set of jobs that can be completed by their deadlines.
{
        J := {1};
        for i := 2 to n do
        {
                if (all jobs in J U {i} can be completed by their dead lines)
                then J := J U {i};
        }
}
```

OPTIMAL MERGE PATERNS

Given 'n' sorted files, there are many ways to pair wise merge them into a single sorted file. As, different pairings require different amounts of computing time, we want to determine an optimal (i.e., one requiring the fewest comparisons) way to pair wise merge 'n' sorted files together. This type of merging is called as 2-way merge patterns. To merge an n-record file and an m-record file requires possibly n + m record moves, the obvious choice choice is, at each step merge the two smallest files together. The two-way merge patterns can be represented by binary merge trees.

Algorithm to Generate Two-way Merge Tree:

```
struct treenode
{
        treenode * lchild;
        treenode * rchild;
};
```

Algorithm TREE (n)

```
// list is a global of n single node binary trees
{
        for i := 1  to n – 1 do
        {
                pt   new treenode
                (pt   lchild)   least (list);          //   merge  two   trees  with   smallest
        lengths
                (pt  rchild)  least  (list);
                (pt  weight)  ((pt  lchild)  weight) + ((pt  rchild)  weight);
                 insert (list, pt);
        }
        return  least  (list);                          // The tree left  in list is the      merge
tree
}
```

Example 1:

Suppose we are having three sorted files X_1, X_2 and X_3 of length 30, 20, and 10 records each. Merging of the files can be carried out as follows:

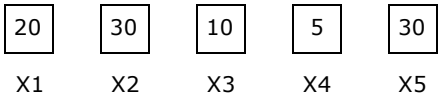
S.No	First Merging	Record moves in first merging	Second merging	Record moves in second merging	Total no. of records moves
1.	X_1 & X_2 = T1	50	T_1 & X_3	60	50 + 60 = 110
2.	X_2 & X_3 = T1	30	T_1 & X_1	60	30 + 60 = 90

The Second case is optimal.

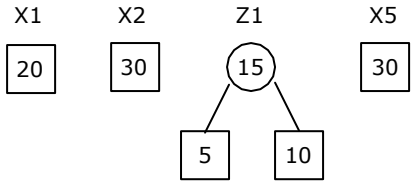
Example 2:

Given five files (X1, X2, X3, X4, X5) with sizes (20, 30, 10, 5, 30). Apply greedy rule to find optimal way of pair wise merging to give an optimal solution using binary merge tree representation.

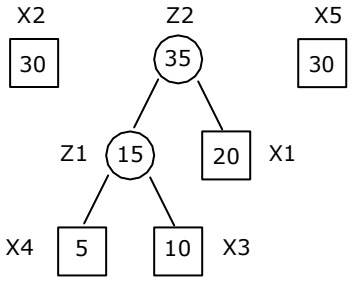
Solution:



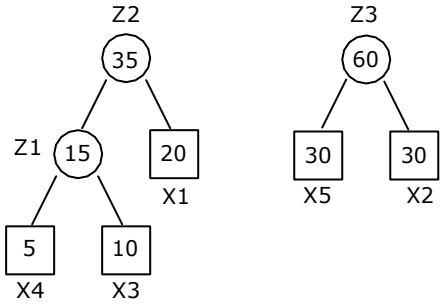
Merge X_4 and X_3 to get 15 record moves. Call this Z_1.



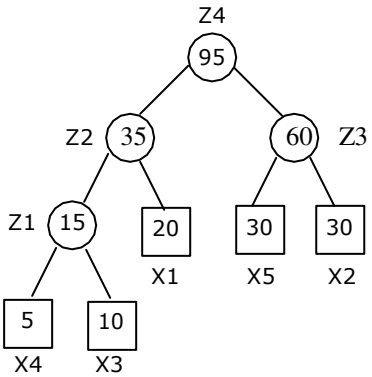
Merge Z_1 and X_1 to get 35 record moves. Call this Z_2.



Merge X_2 and X_5 to get 60 record moves. Call this Z_3.



Merge Z_2 and Z_3 to get 90 record moves. This is the answer. Call this Z_4.



Therefore the total number of record moves is 15 + 35 + 60 + 95 = 205. This is an optimal merge pattern for the given problem.

Huffman Codes

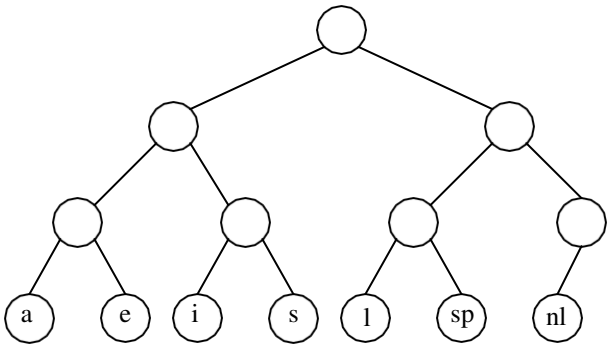
Another application of Greedy Algorithm is file compression.

Suppose that we have a file only with characters a, e, i, s, t, spaces and new lines, the frequency of appearance of a's is 10, e's fifteen, twelve i's, three s's, four t's, thirteen banks and one newline.

Using a standard coding scheme, for 58 characters using 3 bits for each character, the file requires 174 bits to represent. This is shown in table below.

Character	Code	Frequency	Total bits
A	000	10	30
E	001	15	45
I	010	12	36
S	011	3	9
T	100	4	12
Space	101	13	39
New line	110	1	3

Representing by a binary tree, the binary code for the alphabets are as follows:

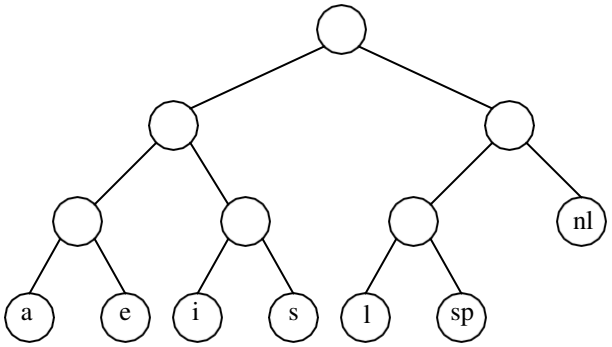


The representation of each character can be found by starting at the root and recording the path. Use a 0 to indicate the left branch and a 1 to indicate the right branch.

If the character c_i is at depth d_i and occurs f_i times, the cost of the code is equal to $d_i \ f_i$

With this representation the total number of bits is 3x10 + 3x15 + 3x12 + 3x3 + 3x4 + 3x13 + 3x1 = 174

A better code can be obtained by with the following representation.



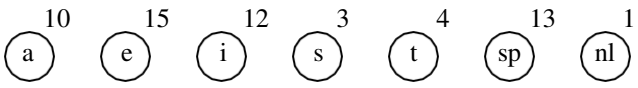
The basic problem is to find the full binary tree of minimal total cost. This can be done by using Huffman coding (1952).

Huffman's Algorithm:

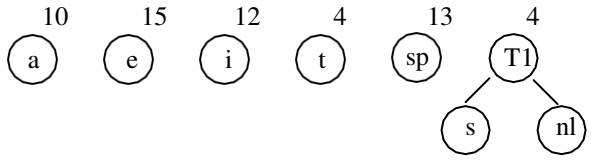
Huffman's algorithm can be described as follows: We maintain a forest of trees. The weights of a tree is equal to the sum of the frequencies of its leaves. If the number of characters is 'c'. c - 1 times, select the two trees T1 and T2, of smallest weight, and form a new tree with sub-trees T1 and T2. Repeating the process we will get an optimal Huffman coding tree.

Example:

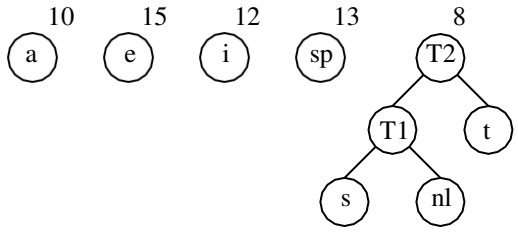
The initial forest with the weight of each tree is as follows:



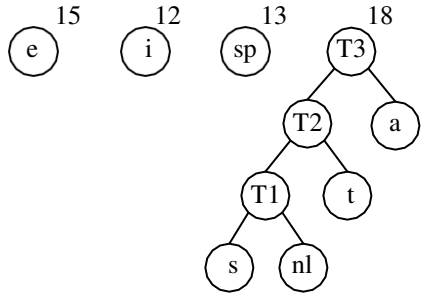
The two trees with the lowest weight are merged together, creating the forest, the Huffman algorithm after the first merge with new root T_1 is as follows: The total weight of the new tree is the sum of the weights of the old trees.



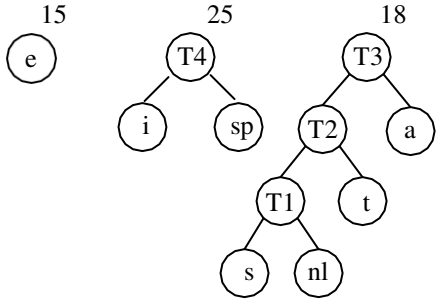
We again select the two trees of smallest weight. This happens to be T_1 and t, which are merged into a new tree with root T_2 and weight 8.



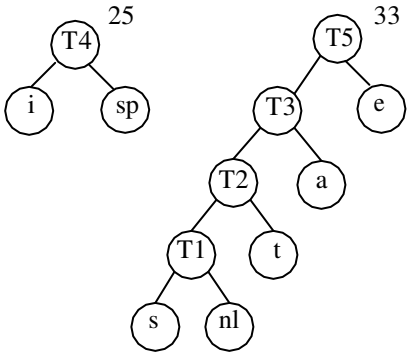
In next step we merge T_2 and a creating T_3, with weight 10+8=18. The result of this operation in



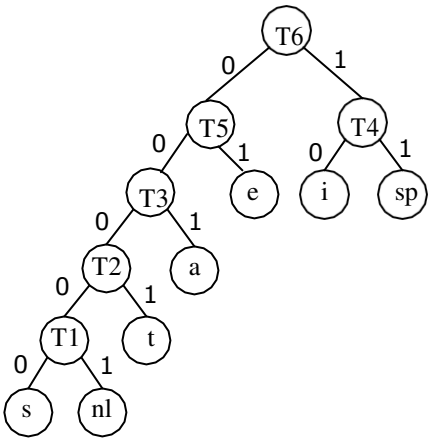
After third merge, the two trees of lowest weight are the single node trees representing i and the blank space. These trees merged into the new tree with root T_4.



The fifth step is to merge the trees with roots e and T_3. The results of this step is



Finally, the optimal tree is obtained by merging the two remaining trees. The optimal trees with root T_6 is:



The full binary tree of minimal total cost, where all characters are obtained in the leaves, uses only 146 bits.

Character	Code	Frequency	Total bits (Code bits X frequency)
A	001	10	30
E	01	15	30
I	10	12	24
S	00000	3	15
T	0001	4	16
Space	11	13	26
New line	00001	1	5
		Total :	146

GRAPH ALGORITHMS

Basic Definitions:

Graph G is a pair (V, E), where V is a finite set (set of vertices) and E is a finite set of pairs from V (set of edges). We will often denote n := |V|, m := |E|.

Graph G can be **directed**, if E consists of ordered pairs, or undirected, if E consists of unordered pairs. If (u, v) E, then vertices u, and v are adjacent.

We can assign weight function to the edges: $w_G(e)$ is a weight of edge e E. The graph which has such function assigned is called **weighted**.

Degree of a vertex v is the number of vertices u for which (u, v) E (denote deg(v)). The number of **incoming edges** to a vertex v is called **in–degree** of the vertex (denote indeg(v)). The number of **outgoing edges** from a vertex is called **out-degree** (denote outdeg(v)).

Representation of Graphs:

Consider graph G = (V, E), where V= $\{v_1, v_2, \ldots, v_n\}$.

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

$$a_{i,j} \quad \begin{matrix} 1, & \text{if } (v_i, v_j) \ E, \\ 0, & \text{otherwise} \end{matrix}$$

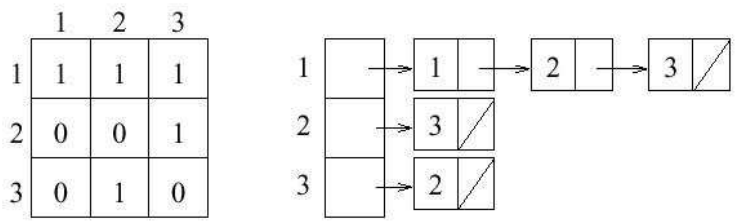
The matrix is symmetric in case of undirected graph, while it may be asymmetric if the graph is directed.

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

$$a_{i,j} \quad \begin{matrix} w(v_i, v_j), & \text{if } (v_i, v_j) \ E, \\ \text{default}, & \text{otherwise}, \end{matrix}$$

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

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



Adjacency matrix Adjacency list

Paths and Cycles:

A path is a sequence of vertices (v_1, v_2, ... , v_k), where for all i, (v_i, v_{i+1}) E. **A path is simple** if all vertices in the path are distinct.

A (simple) cycle is a sequence of vertices (v_1, v_2, ... , v_k, v_{k+1} = v_1), where for all i, (v_i, v_{i+1}) E and all vertices in the cycle are distinct except pair v_1, v_{k+1}.

Subgraphs and Spanning Trees:

Subgraphs: A graph G′ = (V′, E′) is a subgraph of graph G = (V, E) iff V′ V and E′ E.

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

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

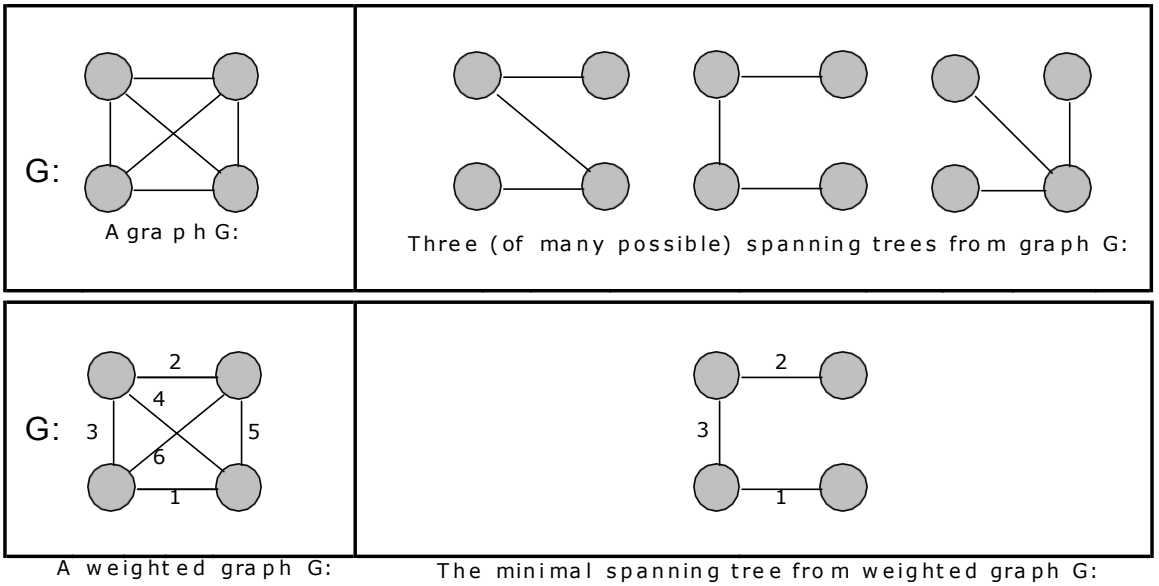
Lemma 1: *Let T be a spanning tree of a graph G. Then*

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

Minimum Spanning Trees (MST):

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

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



Here are some examples:

To explain further upon the Minimum Spanning Tree, and what it applies to, let's consider a couple of real-world examples:

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

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

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

Kruskal's Algorithm

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

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

Algorithm:

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

Algorithm Kruskal (E, cost, n, t)

```
// E is the set of edges in G. G has n vertices. cost [u, v] is the
// cost of edge (u, v). 't' is the set of edges in the minimum-cost spanning tree.
// The final cost is returned.
{
        Construct a heap out of the edge costs using heapify;
        for  i := 1 to n do parent [i] := -1;
                                                // Each vertex is in a different set.
        i := 0; mincost := 0.0;
        while ((i < n -1) and (heap not empty)) do
        {
                Delete a minimum cost edge (u, v) from the heap and
                re-heapify using Adjust;
                j := Find (u); k := Find (v);
                if  (j  k) then
                {
                        i := i + 1;
                        t [i, 1] := u; t [i, 2] := v;
                        mincost :=mincost + cost [u, v];
                        Union (j, k);
                }
        }
        if (i  n-1) then write ("no spanning tree");
        else return mincost;
}
```

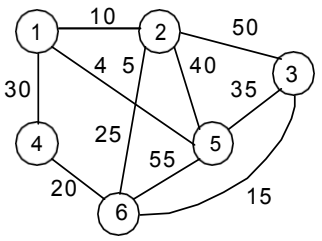
Running time:

The number of finds is at most 2e, and the number of unions at most n-1. Including the initialization time for the trees, this part of the algorithm has a complexity that is just slightly more than O (n + e).

We can add at most n-1 edges to tree T. So, the total time for operations on T is O(n).

Summing up the various components of the computing times, we get O (n + e log e) as asymptotic complexity

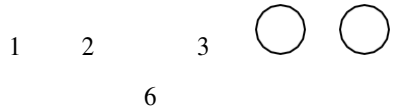
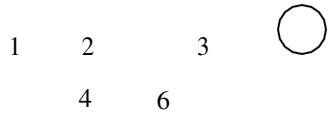
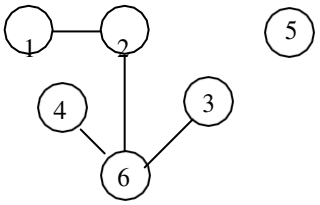
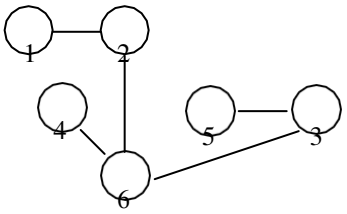
Example 1:



Arrange all the edges in the increasing order of their costs:

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

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

Edge	Cost	Spanning Forest	Edge Sets	Remarks
			{1}, {2}, {3}, {4}, {5}, {6}	
(1, 2)	10		{1, 2}, {3}, {4}, {5}, {6}	The vertices 1 and 2 are in different sets, so the edge is combined
(3, 6)	15		{1, 2}, {3, 6}, {4}, {5}	The vertices 3 and 6 are in different sets, so the edge is combined
(4, 6)	20		{1, 2}, {3, 4, 6}, {5}	The vertices 4 and 6 are in different sets, so the edge is combined
(2, 6)	25		{1, 2, 3, 4, 6}, {5}	The vertices 2 and 6 are in different sets, so the edge is combined
(1, 4)	30	Reject		The vertices 1 and 4 are in the same set, so the edge is rejected
(3, 5)	35		{1, 2, 3, 4, 5, 6}	The vertices 3 and 5 are in the same set, so the edge is combined

MINIMUM-COST SPANNING TREES: PRIM'S ALGORITHM

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

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

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

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

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

Algorithm Algorithm Prim

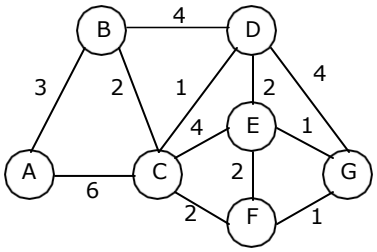
```
(E, cost, n, t)
// E is the set of edges in G. cost [1:n, 1:n] is the cost
// adjacency matrix of an n vertex graph such that cost [i, j] is
// either a positive real number or  if no edge (i, j) exists.
// A minimum spanning tree is computed and stored as a set of
// edges in the array t [1:n-1, 1:2]. (t [i, 1], t [i, 2]) is an edge in
// the minimum-cost spanning tree. The final cost is returned.
{
        Let (k, l) be an edge of minimum cost in E;
        mincost := cost [k, l];
        t [1, 1] := k; t [1, 2] := l;
        for   i :=1 to  n  do                          // Initialize near
                if  (cost [i, l] < cost [i, k]) then near [i] := l;
                else near [i] := k;
        near [k] :=near [l] := 0;
        for  i:=2 to n -   1  do                       // Find n - 2 additional edges for t.
        {
                Let j be an index such that near [j]  0 and
                cost [j, near [j]] is minimum;
                t [i, 1] := j; t [i, 2] := near [j];
                mincost := mincost + cost [j, near [j]];
                near [j] := 0
                for   k:= 1 to  n  do                          // Update near[].
                        if ((near [k]  0) and (cost [k, near [k]] > cost [k, j]))
                                then near [k] := j;
        }
        return mincost;
}
```

Running time:

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

EXAMPLE 1:

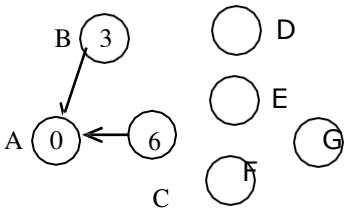
Use Prim's Algorithm to find a minimal spanning tree for the graph shown below starting with the vertex A.



SOLUTION:

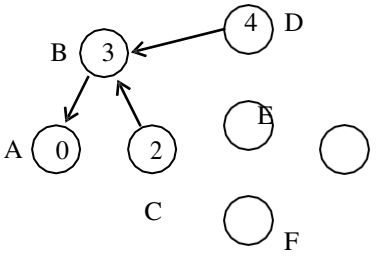
The stepwise progress of the prim's algorithm is as follows:

Step 1:



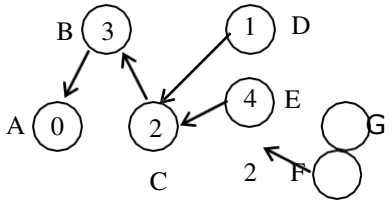
Vertex	A	B	C	D	E	F	G
Status	0	1	1	1	1	1	1
Dist.	0	3	6				
Next	*	A	A	A	A	A	A

Step 2:



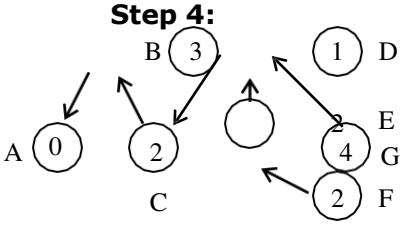
Vertex	A	B	C	D	E	F	G
Status	0	0	1	1	1	1	1
Dist.	0	3	2	4			
Next	*	A	B	B	A	A	A

Step 3:



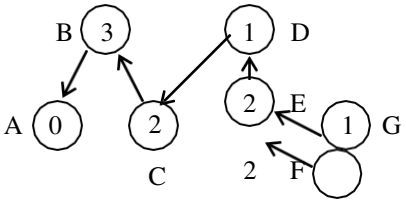
Vertex	A	B	C	D	E	F	G
Status	0	0	0	1	1	1	1
Dist.	0	3	2	1	4	2	
Next	*	A	B	C	C	C	A

Step 4:



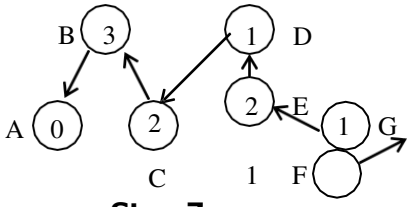
Vertex	A	B	C	D	E	F	G
Status	0	0	0	0	1	1	1
Dist.	0	3	2	1	2	2	4
Next	*	A	B	C	D	C	D

Step 5:



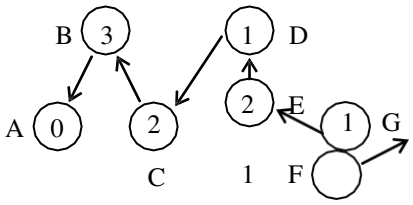
Vertex	A	B	C	D	E	F	G
Status	0	0	0	0	1	0	1
Dist.	0	3	2	1	2	2	1
Next	*	A	B	C	D	C	E

Step 6:



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

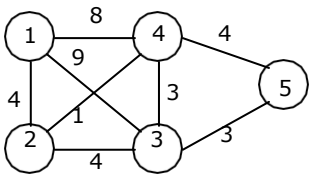
Step 7:



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

EXAMPLE 2:

Considering the following graph, find the minimal spanning tree using prim's algorithm.

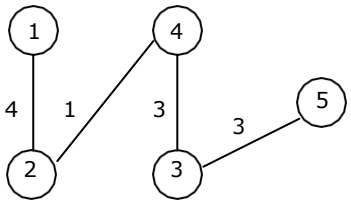


The cost adjacent matrix is

	4	9	8	
4		4	1	
9	4	3		3
8	1	3		4
		3	4	

The minimal spanning tree obtained as:

Vertex 1	Vertex 2
2	4
3	4
5	3
1	2



The cost of Minimal spanning tree = 11.

<u>The steps as per the algorithm are as follows:</u>

Algorithm near (J) = k means, the nearest vertex to J is k.

The algorithm starts by selecting the minimum cost from the graph. The minimum cost edge is (2, 4).

K = 2, l = 4
Min cost = cost (2, 4) = 1

T [1, 1] = 2

T [1, 2] = 4

for i = 1 to 5 Begin i = 1 is cost (1, 4) < cost (1, 2) 8 < 4, No Than near (1) = 2 i = 2 is cost (2, 4) < cost (2, 2) 1 < , Yes So near [2] = 4 i = 3 is cost (3, 4) < cost (3, 2) 1 < 4, Yes So near [3] = 4 i = 4 is cost (4, 4) < cost (4, 2) < 1, no So near [4] = 2 i = 5 is cost (5, 4) < cost (5, 2) 4 < , yes So near [5] = 4 end near [k] = near [l] = 0 near [2] = near[4] = 0	Near matrix 2 | | | | | 1 2 3 4 5 2 | 4 | | | | 1 2 3 4 5 2 | 4 | 4 | | | 1 2 3 4 5 2 | 4 | 4 | 2 | | 1 2 3 4 5 2 | 4 | 4 | 2 | 4 1 2 3 4 5 2 | 0 | 4 | 0 | 4 1 2 3 4 5	Edges added to min spanning tree: T [1, 1] = 2 T [1, 2] = 4
for i = 2 to n-1 (4) do **<u>i = 2</u>** for j = 1 to 5 j = 1 near(1)0 and cost(1, near(1)) 2 0 and cost (1, 2) = 4 j = 2 near (2) = 0 j = 3 is near (3) 0 4 0 and cost (3, 4) = 3		

j = 4
near (4) = 0

J = 5
Is near (5) 0
4 0 and cost (4, 5) = 4

select the min cost from the above obtained costs, which is 3 and corresponding J = 3

min cost = 1 + cost(3, 4)
= 1 + 3 = 4

T (2, 1) = 3
T (2, 2) = 4

Near [j] = 0
i.e. near (3) =0

<u>for (k = 1 to n)</u>

K = 1
is near (1) 0, yes
2 0
and cost (1,2) > cost(1, 3)
4 > 9, No

K = 2
Is near (2) 0, No

K = 3
Is near (3) 0, No

K = 4
Is near (4) 0, No

K = 5
Is near (5) 0
4 0, yes
and is cost (5, 4) > cost (5, 3)
4 > 3, yes
than near (5) = 3

<u>j = 3</u>

<u>for (j = 1 to 5)</u>
J = 1
is near (1) 0
2 0
cost (1, 2) = 4

J = 2
Is near (2) 0, No

2	0	0	0	4
1	2	3	4	5

2	0	0	0	3
1	2	3	4	5

T (2, 1) = 3
T (2, 2) = 4

J = 3
Is near (3) 0, no
Near (3) = 0

J = 4
Is near (4) 0, no
Near (4) = 0

J = 5
Is near (5) 0
Near (5) = 3 3 0, yes
And cost (5, 3) = 3

Choosing the min cost from the above obtaining costs which is 3 and corresponding J = 5

Min cost = 4 + cost (5, 3)
= 4 + 3 = 7

T (3, 1) = 5
T (3, 2) = 3

T (3, 1) = 5
T (3, 2) = 3

Near (J) = 0 near (5) = 0

<u>for (k=1 to 5)</u>

k = 1
is near (1) 0, yes
and cost(1,2) > cost(1,5)
4 > , No

K = 2
Is near (2) 0 no

K = 3
Is near (3) 0 no

K = 4
Is near (4) 0 no

K = 5
Is near (5) 0 no

<u>i = 4</u>

<u>for J = 1 to 5</u>
J = 1
Is near (1) 0
2 0, yes
cost (1, 2) = 4

j = 2
is near (2) 0, No

2	0	0	0	0
1	2	3	4	5

J = 3
Is near (3) 0, No
Near (3) = 0

J = 4
Is near (4) 0, No
Near (4) = 0

J = 5
Is near (5) 0, No
Near (5) = 0

Choosing min cost from the above it is only '4' and corresponding J = 1

Min cost = 7 + cost (1,2)
= 7+4 = 11

T (4, 1) = 1
T (4, 2) = 2

Near (J) = 0 Near (1) = 0

<u>for (k = 1 to 5)</u>

K = 1
Is near (1) 0, No

K = 2
Is near (2) 0, No

K = 3
Is near (3) 0, No

K = 4
Is near (4) 0, No

K = 5
Is near (5) 0, No

End.

0	0	0	0	0
1	2	3	4	5

T (4, 1) = 1
T (4, 2) = 2

4.8.7. The Single Source Shortest-Path Problem: DIJKSTRA'S ALGORITHMS

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

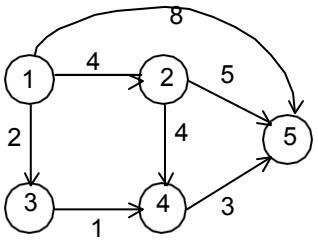
In the single source, all destinations, shortest path problem, we must find a shortest path from a given source vertex to each of the vertices (called destinations) in the graph to which there is a path.

Dijkstra's algorithm is similar to prim's algorithm for finding minimal spanning trees. Dijkstra's algorithm takes a labeled graph and a pair of vertices P and Q, and finds the

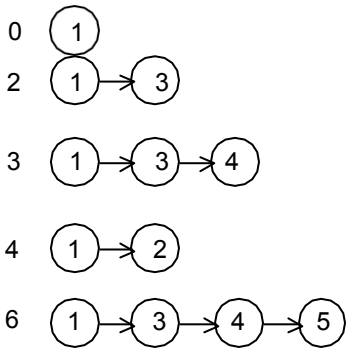
shortest path between then (or one of the shortest paths) if there is more than one. The principle of optimality is the basis for Dijkstra's algorithms.

Dijkstra's algorithm does not work for negative edges at all.

The figure lists the shortest paths from vertex 1 for a five vertex weighted digraph.



Graph



Shortest Paths

Algorithm:

```
Algorithm Shortest-Paths (v, cost, dist, n)
// dist [j], 1 ≤ j ≤ n, is set to the length of the shortest path
// from vertex v to vertex j in the digraph G with n vertices.
// dist [v] is set to zero. G is represented by its
// cost adjacency matrix cost [1:n, 1:n].
{
        for  i :=1 to n do
        {
                S  [i] :=  false;                          // Initialize S.
                dist [i] :=cost [v, i];
        }
        S[v] := true; dist[v]  := 0.0;                     // Put v in S.
        for num := 2 to n – 1 do
        {
                Determine n - 1 paths from v.
                Choose u from among those vertices not in S such that dist[u] is minimum;
                S[u]  :=  true;                                  // Put u is S.
                for (each w adjacent to u with S [w] = false) do
                        if (dist [w] > (dist [u] + cost [u,  w]) then      // Update distances
                                dist [w] := dist [u] + cost [u, w];
        }
}
```

Running time:

Depends on implementation of data structures for dist.

Build a structure with n elements A

at most m = E times decrease the value of an item mB

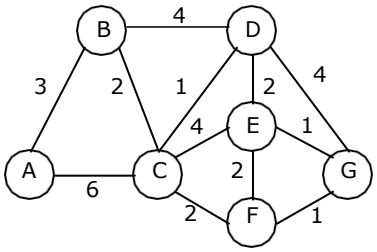
'n' times select the smallest value nC

For <u>array</u> A = O (n); B = O (1); C = O (n) which gives O (n^2) total.

For <u>heap</u> A = O (n); B = O (log n); C = O (log n) which gives O (n + m log n) total.

Example 1:

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



Solution:

The cost adjacency matrix is

0	3	6	-	-	-	-
3	0	2	4	-	-	-
6	2	0	1	4	2	-
4	1	0	2	-	-	4
-	4	2	0	2		1
-	-	2	-	2	0	1
-	-	-	-	4	1	1	0

Here – means infinite

The problem is solved by considering the following information:

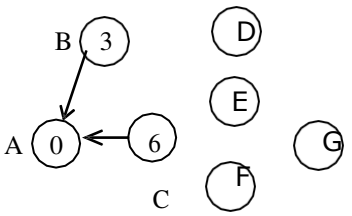
Status[v] will be either '0', meaning that the shortest path from v to v_0 has definitely been found; or '1', meaning that it hasn't.

Dist[v] will be a number, representing the length of the shortest path from v to v_0 found so far.

Next[v] will be the first vertex on the way to v_0 along the shortest path found so far from v to v_0

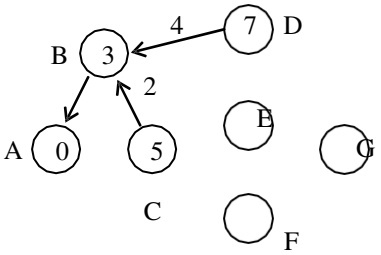
The progress of Dijkstra's algorithm on the graph shown above is as follows:

Step 1:



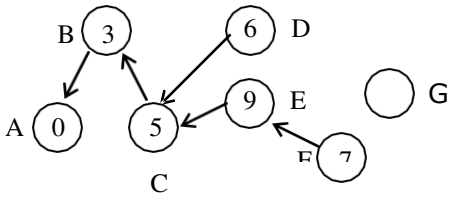
Vertex	A	B	C	D	E	F	G
Status	0	1	1	1	1	1	1
Dist.	0	3	6				
Next	*	A	A	A	A	A	A

Step 2:



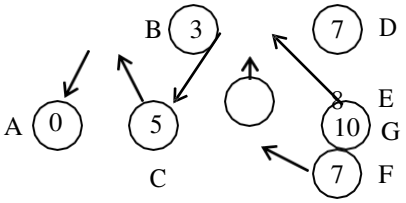
Vertex	A	B	C	D	E	F	G
Status	0	0	1	1	1	1	1
Dist.	0	3	5	7			
Next	*	A	B	B	A	A	A

Step 3:



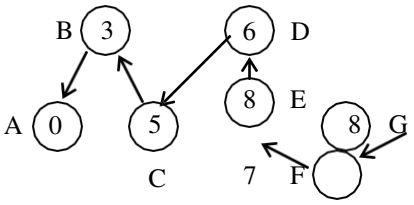
Vertex	A	B	C	D	E	F	G
Status	0	0	0	1	1	1	1
Dist.	0	3	5	6	9	7	
Next	*	A	B	C	C	C	A

Step 4:



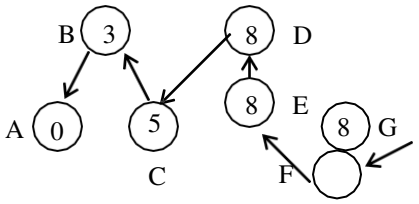
Vertex	A	B	C	D	E	F	G
Status	0	0	0	0	1	1	1
Dist.	0	3	5	6	8	7	10
Next	*	A	B	C	D	C	D

Step 5:



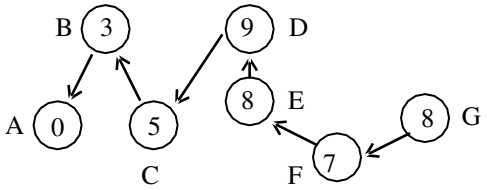
Vertex	A	B	C	D	E	F	G
Status	0	0	0	0	1	0	1
Dist.	0	3	5	6	8	7	8
Next	*	A	B	C	D	C	F

Step 6:



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

Step 7:



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

DYNAMIC PROGRAMMING

- It was invented by a prominent U.S mathematician Richard Bellman, in the 1950s as a general method for optimising multistage decision process.
- Programming standards for planning.
- Dynamic programming is a technique for solving problems with overlapping subproblems.

1. THE GENERAL METHOD

- Dynamic programming is an algorithm design method that can be used when the solution to a problem can be viewed as the result of a sequence of decisions.

Some Examples.

1. Knapsack:-

- The solution to the knapsack problem is the result of sequence of decisions.
- We have to decide the values of $x_i, 1 \leq i \leq n$.
- First we make a decision on x_1, then on x_2, then on x_3, and so on.
- An optimal sequence of decisions maximizes the objective function $\sum p_i x_i$.
- It also satisfies the constraint $\sum w_i x_i \leq m$ and $0 \leq x_i \leq 1$.

2. Shortest Path:

- One way to find a shortest path from vertex i to vertex j in a directed graph G is to decide which vertex should be the second vertex, which the third, which the fourth, and so on, until j is reached.
- An optimal sequence of decisions is one that results in a path of least length.

- For some of the problems that may be viewed in this way, an optimal sequence of decisions can be found by making the decisions one at a time and never making an erroneous decision.
- This is true for all problems solvable by the greedy method.
- For many other problems, it is not possible to make stepwise decisions in such a manner that the sequence of decisions made is optimal.

example. Shortest path:-

- Suppose we wish to find a shortest path from vertex i to vertex j.
- Let A_i be the vertex adjacent from vertex i.
- Which of the vertex in A_i should be the second vertex on the path?
- There is no way to make a decision at this time and guarantee that future decisions leading to an optimal sequence can be made.
- If on the other hand we wish to find a shortest path from vertex i to all other vertices in G, then at each step, a correct decision can be made.
- One way to solve problems for which it is not possible to make a sequence of stepwise decisions leading to an optimal decision sequence is to try all possible decision sequences and then pick out the best.
- In dynamic programming an optimal sequence of decisions is obtained by making explicit appeal to the principle of optimality.

Def. Principle of optimality.

- It states that an optimal sequence of decisions has the property that whatever the initial state and decision are, the remaining decisions must constitute an optimal decision sequence with regard to the state resulting from the first decision.

Difference between the greedy method and dynamic programming.

- In greedy method only one decision sequence ever generated.
- In dynamic programming, many decision sequences may be generated.
- However, sequences containing suboptimal subsequences can't be optimal and so will not be generated.

Example shortest Path:-

- To find the shortest path from vertex i to vertex j. in a directed graph G.
- Assume that $i, i_1, i_2 \ldots i_k, j$ is a shortest path from i to j.
- Starting with the initial vertex i, a decision has been made to go to vertex i_1.
- Following this decision, the problem state is defined by vertex i_1 and we need to find a path from i_1 to j.
- It is clear that the sequence $i_1, i_2 \ldots i_k, j$ must constitute a shortest i_1 to j path.
- If not, let $i_1, r_1, r_2 \ldots r_q, j$ be a shortest i_1 to j path.

then $i, i1, n_1 \cdots n_q, j$ is an i to j path that is shorter than the path $i, i1, i2 \cdots ik, j$.

- Therefore the principle of optimality applies for this problem.

<u>Example</u> <u>0/1 knapsack</u>

- The 0/1 knapsack problem is similar to the knapsack problem except that the x_i's are restricted to have a value of either 0 or 1.
- Using KNAP(l, j, y) to represent the problem

$$\text{maximize } \sum_{l \le i \le j} P_i x_i$$
$$\text{subject to } \sum_{l \le i \le j} w_i x_i \le y$$
$$x_i = 0 \text{ or } 1, \ l \le i \le j.$$

the knapsack problem is KNAP$(1, n, m)$.

- Let $y_1, y_2 \cdots y_n$ be an optimal sequence of 0/1 values for $x_1, x_2 \cdots x_n$ respectively.
- If $y_1 = 0$, then $y_2, y_3 \cdots y_n$ must constitute an optimal sequence for the problem KNAP$(2, n, m)$
- If it does not, then $y_1, y_2 \cdots y_n$ is not an optimal sequence for KNAP$(1, n, m)$.
- If $y_1 = 1$, then $y_2 \cdots y_n$ must be an optimal sequence for the problem KNAP$(2, n, m - w_1)$.
- If it is not, then there is another 0/1 sequence $z_2, z_3 \cdots z_n$ such that $\sum_{2 \le i \le n} w_i z_i \le m - w_1$ and $\sum_{2 \le i \le n} P_i z_i > \sum_{2 \le i \le n} P_i y_i$.
- Hence the sequence $y_1, z_2, z_3 \cdots z_n$ is a sequence with greater value.
- Therefore the principle of optimality applies.

ALL-PAIRS SHORTEST PATHS.

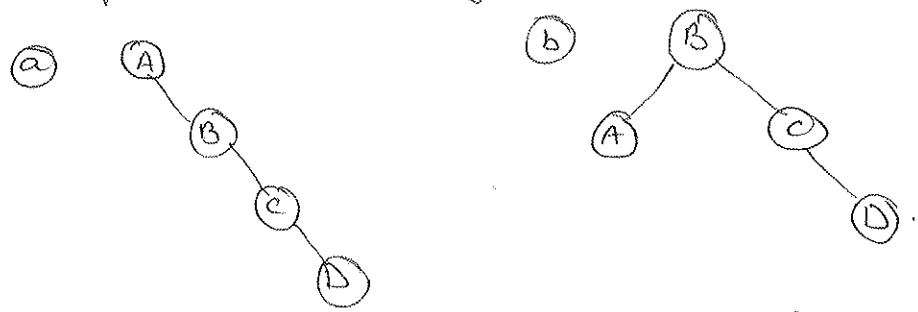
- Let ~~G = (V~~ $G = (V, E)$ be a directed graph with n vertices.
- Let cost be a cost of adjacency matrix for G such that $cost(i, i) = 0, 1 \le i \le n$.

  $cost(i, j) =$ length / cost of edge (i, j) if $(i, j) \in E(G)$.

  $cost(i, j) = \infty$ if $i \ne j$ and $(i, j) \notin E(G)$.
- The all-pairs shortest-path problem is to determine a matrix A such that $A(i, j)$ is the length of the shortest path from i to j.
- If we allow G to contain a cycle of negative length, then the shortest path between any two vertices on this cycle has length $-\infty$.
- Let us examine a shortest i to j path in G, $i \ne j$.
- This path originates at vertex i and goes through some intermediate vertices and terminates at vertex j.
- We can assume that this path contains no cycles for if there is a cycle, then this can be deleted without increasing the path length.
- 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.
- Using $A^k(i, j)$ to represent the length of a shortest path from i to j going through no vertex of index greater than k.

Optimal Binary Search Trees

- A binary search tree is one of the most important data structures in computer science.
- One of its principal applications is to implement a dictionary, a set of elements with the operations of searching, insertion and deletions.
- If probabilities of searching for elements of a set are known then an optimal binary search tree for which the average number of comparisons in a search is the smallest possible.
- Consider four keys A, B, C and D to be searched for with probabilities 0.1, 0.2, 0.4 and 0.3
- Two possible binary search trees are



no. of possible search trees are 14.

$$\frac{{}^{2n}C_n}{n+1}$$

- The average number of comparisons in a successful search (a) $0.1\times1 + 0.2\times2 + 0.4\times3 + 0.3\times4 = 0.1+0.4+1.2+1.2 = 2.9$.

  (b) $0.1\times2 + 0.2\times1 + 0.4\times2 + 0.3\times3$

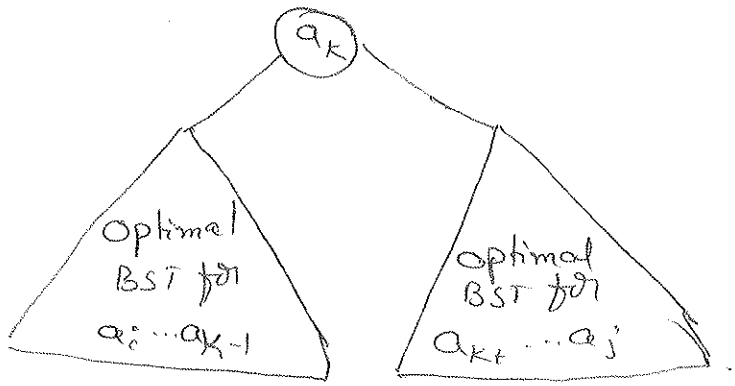
  $= 0.2 + 0.2 + 0.8 + 0.9 = 2.1$

- We would find the optimal tree by generating all binary search trees with these keys.
- The total number of binary search trees with n keys is equal to the nth Caralan Number

$$C(n) = \binom{2n}{n}\frac{1}{n+1} \quad \text{for } n>0$$

$$C(0) = 1.$$

- We will find the values of $c[i,j]$ for all smaller instances of the problem.
- Although we are interested just in $c[1,n]$.
- We will consider all possible ways to choose a root a_k among the keys $a_i \dots a_j$.
- The root key a_k, the left subtree T_i^{k-1} contains keys $a_i \dots a_{k-1}$, and the right subtree T_{k+1}^{j} contains keys $a_{k+1} \dots a_j$



- If we count tree travels starting with 1, the following recurrence relation is obtained.

$$c[i,j] = \min_{i \le l \le j} \Big[P_k + \sum_{s=1}^{k-1} P_s \cdot (\text{level of } a_s \text{ in } T_i^{k-1} + 1) + \sum_{s=k+1}^{j} P_s \cdot (\text{level of } a_s \text{ in } T_{k+1}^{j} + 1) \Big]$$

$$= \min_{i \le l \le j} \Big[P_k + \sum_{s=1}^{k-1} P_s \cdot \text{level of } a_s \text{ in } T_i^{k-1} + \sum_{s=1}^{k-1} P_s + \sum_{s=k+1}^{j} P_s \cdot \text{level of } a_s \text{ in } T_{k+1}^{j} + \sum_{s=k+1}^{j} P_s \Big]$$

$$= \min_{i \le l \le j} \Big[\sum_{s=1}^{k-1} P_s \cdot \text{level of } a_s \text{ in } T_i^{k-1} + \sum_{s=k+1}^{j} P_s \cdot \text{level of } a_s \text{ in } T_{k+1}^{j} + \sum_{s=l}^{j} P_s \Big]$$

$$= \min_{i \le l \le j} \big[c[i,k-1] + c[k+1,j] \big] + \sum_{s=l}^{j} P_s$$

$$\therefore c[i,j] = \min_{i \le l \le j} \big[c[i,k-1] + c[k+1,j] \big] + \sum_{s=i}^{j} P_s \text{ for } 1 \le i \le j \le n.$$

Example

	0	1	2	3
Keys	10	12	16	21
Frequencies.	4	2	6	3.

	0	1	2	3
0	4	$8^{(0)}$	$20^{(2)}$	$26^{(2)}$
1		2	$10^{(2)}$	$16^{(2)}$
2			6	$12^{(2)}$
3				3

Find $c[i,j]$. $c[i,i] = P_i$.

$c[i,j] = cost[i] + cost[j]$
$\min\left[\begin{array}{l}\text{total cost of root } i, \\ \text{total cost of root } j\end{array}\right]$

$l = 1$.
$c[0,0] = 4$
$c[1,1] = 2$
$c[2,2] = 6$
$c[3,3] = 3$.

$l = 2$. $\Rightarrow \{(0,1), (1,2), (2,3)\}$

case 1.

$$c[0,1] = 4 + 2 + \min\left[\begin{array}{ll} 2 & \text{if root is } 0 \\ 4 & \text{if root is } 1 \end{array}\right.$$

$$= 6 + 2 = 8^{(0)}$$

$$c[1,2] = 2 + 6 + \min\left[\begin{array}{ll} 6 & \text{if root is } 1 \\ 2 & \text{if root is } 2 \end{array}\right.$$

$$= 8 + 2 = 10^{(2)}$$

$$c[2,3] = 6 + 3 + \min\left[\begin{array}{ll} 3 & \text{if root is } 2 \\ 6 & \text{if root is } 3 \end{array}\right.$$

$$= 9 + 3 = 12^{(2)}$$

$l = 3 \quad : \{(0,1,2), (1,2,3)\}$

$$c[0,1,2] = 4+2+6+\min\begin{cases} 10 & \text{if root is 0 i.e. cost of } (1,2) \\ 4+6=10 & \text{if root is 1 i.e. cost of } 0 + \text{cost } 2 \\ 8 & \text{if root in 2 i.e. cost of } (0,1) \end{cases}$$

$$= 12+8 = 20^{(2)}$$

$$c[1,2,3] = 2+6+3+\min\begin{cases} 12 & \text{if root is 1 i.e. cost of } (2,3) \\ 2+3 & \text{if root is 2 i.e. cost}[1] + \text{cost}(3) \\ 10 & \text{if root is 3 i.e. cost}(1,2) \end{cases}$$

$$= 11+5 = 16^{(2)}$$

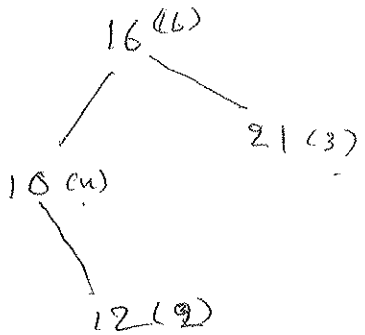
$\therefore$ Minimum $= 16^{(2)}$

$l = 4 \quad \{(0,1,2,3)\}$

$$c[0,1,2,3] = 4+2+6+3+\min\begin{cases} 16 & \text{if root is 0 i.e. } c(1,2,3) \\ 4+12=16 & \text{if root is 1 i.e. } c[0] + c[2,3] \\ 8+3=11 & \text{if root is 2 i.e. } c[0,1] + c[3] \\ 20 & \text{if root is 3 i.e. } c(0,1,2) \end{cases}$$

$$= 15+11 = 26^{(2)}$$

Binary Search tree is



no of comparisons $= (6\times1)+(4\times2)+(3\times2)+2\times3$

$= 6+8+6+6$

$= \underline{26}$

Algorithm. OptimalBST(P[1..n])

// Finds an optimal binary search tree by dynamic programming.

// Input: An array P[1..n] of search probabilities for a

// sorted list of n keys

// output: Average number of comparisons in successful

// searches in the optimal BST and table R of

// subtrees' roots in the optimal BST.

```
for i = 1 to n do
    c[i, i-1] = 0
    c[i, i] = P[i]
    R[i, i] = i
    c[n+1, n] = 0
    for d = 1 to n-1 do // diagonal count.
        for i = 1 to n-d do
            j = i+d.
            minval = ∞.
            for k = i to j do
                if c[i,k-1] + c[k+1, j] < minval
                    minval = c[i,k-1] + c[k+1, j];
                    kmin = k.
                    R[i,j] = kmin.
                    sum = P[i]
                    for s = i+1 to j do
                        sum = sum + P[s]
                        c[i,j] = minval + sum.
    return c[1,n] R.
```

0/1 Knapsack Problem.

- Given n items of known weights $w_1 \cdots w_n$ and values $v_1 \cdots v_n$ and knapsack capacity w.
- Find the most valuable subset of the items that fit into the knapsack.
- All the weights and the knapsack's capacity are positive integers.
- Let us consider an instance defined by the first i items, $1 \le i \le n$ with weights $w_1 \cdots w_i$ values $v_1 \cdots v_i$ and knapsack capacity j, $1 \le j \le w$.
- Let $v[i,j]$ be the value of an optimal solution to this instance.
  i.e. the value of the most valuable subset of the first i items that fit into the knapsack of capacity j.
- We can divide all the subsets of the first i items that the knapsack of capacity j into two categories.
  1. Those that do not include the ith item.
  2. Those that do.

$$v[i,j] = \begin{cases} \max[v[i-1,j],\ v_i + v[i-1, j-w_i]] & \text{if } j-w_i \ge 0 \\ v[i-1,j] & \text{if } j-w_i < 0 \end{cases}$$

Initial conditions.

$v[0,j] = 0$ for $j \ge 0$

$v[i,0] = 0$ for $i \ge 0$.

Table for solving the knapsack problem.

	0	$j-w_i$	j	w
0	0	0	0	0
$i-1$	0	$v[i-1, j-w_i]$	$v[i-1,j]$	
i	0		$v[i,j]$	
n	0			goal

Example.

item.	1	2	3	4
weight.	2	1	3	2
value.	12	10	20	15

capacity $W = 5$.

	weight	Value	v[i,j]	0	1	2	3	4	5
			0	0	0	0	0	0	0
w_1	2	12	1	0	0	12	12	12	12
w_2	1	10	2	0	10	12	22	22	22
w_3	3	20	3	0	10	12	22	30	32
w_4	2	15	4	0	10	15	25	30	37.

Initially $v[0,j] = 0$, $v[i,0] = 0$

1. $v[1,1]$, $j - w_i = 1 - 2 < 0$

$v[i-1, j] = v[0,1]$

$= 0$

2. $v[1,2]$, $j - w_1 = 2 - 2 = 0$

$\max[v[i-1,j], v_i + v[i-1, j-w_i]]$

$= \max[v[0,2], 12 + v[0,0]]$

$= \max[0, 12 + 0 = 12]$

$= 12$

$v[1,3]$, $j - w_1 = 3 - 2 > 0$

$\max[v[i-1,j], v_i + v[i-1, j-w_i]]$

$= \max[v[0,3], 12 + v[0,1]]$

$= \max[0, 12+0]$

$= 12$

$v[1,4]$, $j - w_1 = 4 - 2 > 0$

$\max[v[i-1,j], v_i + v[i-1, j-w_i]]$

$= \max[v[0,4], 12 + v[0,2]]$

$= \max[0, 12+0]$

$= 12$

$v[1,5]$, $j - w_i = 5 - 2 > 0$

$\max[v[i-1,j], v_i + v[i-1, j-w_i]]$

$= \max[v[0,5], 12 + v[0,3]]$

$= \max[0, 12+0]$

$= 12$

$V[2,1]$ $j-w_2 = 1-1 = 0$

$\max(v[i-1,j], v_2 + v[i-1, j-w_i])$

$= \max[v[1,1], 10 + v[1,0]]$

$= \max[0, 10+0]$

$= 10$

$V[2,3]$ $j-w_2 = 3-1 > 0$

$\max[v[1,3], 10 + v[1,2]]$

$= \max[12, 10 + 12]$ (22)

$= 22$

$V[2,5]$, $j-w_2 = 5-1 > 0$

$\max[v[1,5], 10 + v[1,4]]$

$= \max[12, 10 + 12]$ (22)

$= 22$

$V[3,1]$, $j-w_3 = 1-3 < 0$

$v[i-1,j] = v[2,1]$

$= 10$

$V[3,3]$, $j-w_3 = 3-3 = 0$

$\max[v[2,3], 20 + v[2,0]]$

$= \max[22, 20+0]$

$= 22$

$V[3,5]$, $j-w_3 = 5-3 > 0$

$\max[v[2,5], 20 + v[2,2]]$

$= \max[22, 20 + 12]$ (32)

$= 32$

$V[4,3]$ $j-w_4 = 3-2 > 0$

$\max[v[3,3], 15 + v[3,1]]$

$= \max(22, 15+10)$

$= 25$

$V[4,4]$ $j-w_4 = 4-2 > 0$

$\max[v[3,4], 15 + v[3,2]]$

$\max[30, 15+12]$

$= 30$

$V[2,2]$, $j-w_2 = 2-1 > 0$

$\max[v[1,2], 10 + v[1,1]]$

$= \max[12, 10+0]$

$= 12$

$V[2,4]$. $j-w_2 = 4-1 > 0$

$\max[v[1,4], 10 + v[1,3]]$

$= \max[12, 10 + 12]$ (22)

$= 22$

$V[3,2]$, $j-w_3 = 2-3 < 0$

$v[i-1,j] = v[2,2]$

$= 12$

$V[3,4]$ $j-w_3 = 4-3 > 0$

$\max[v[2,4], 20 + v[2,1]]$

$= \max[22, 20+10]$ (30)

$= \cancel{\max}\ 30$

$V[4,1]$, $j-w_4 = 1-2 < 0$

$v[i-1,j] = v[3,1]$

$= 10$

$V[4,2]$, $j-w_4 = 2-2 = 0$

$\max[v[3,2], 15 + v[3,0]]$

$= \max[12, 15+0]$

$= 15$

$V[4,5]$ $j-w_4 = 5-2 > 0$

$\max[v[3,5], 15 + v[3,3]]$

$= \max[32, 15 + 22]$ (37)

$= 37$

Maximal value is $V[4,5] = 37$.

We can find the composition of an optimal subset by tracking back the computations of this entry in the table.

$V[4,5] \neq V[3,5]$.

$\therefore$ item 4 is selected for optimal subset.

$\therefore$ $5-2 = 3$ remaing weight.

Latter is represented by element $V[3,5]$.

$V[3,3] = V[2,3]$

$\therefore$ 3 is not part of optimal subset.

$V[2,3] \neq V[1,3]$

$\therefore$ item 2 is selected for optimal subset.

$3-1 = 2$ remaing weight.

$V[1,3-1]$ i.e $V[1,2] \neq V[0,2]$

$\therefore$ item 1 is the final part of the optimal subset.

$2-2 = 0$.

$\therefore$ Final set [item 1, item 2, item 4].

item 1 -	2,	12
item 2	1,	10
item 4	2,	15
	5,	37

Example ②

item	1	2	3	4
weight	1	3	4	5
value	1	4	5	7

Total weight = 7

	weight	value	v[i,j]	0	1	2	3	4	5	6	7
w1	1	1		0	0	0	0	0	0	0	0
w2	3	4		0	1	1	1	1	1	1	1
w3	4	5		0	1	1	4	5	5	5	5
w4	5	7		0	1	1	4	5	6	6	9
				0	1	1	4	5	7	8	9

$v[1,1]$ $j-w_1 = 1-1$.

$\max(0,1) + v_1 + v[0,0])$

$0+(1+0)=1$

$v[1,2] = j-w_2 = 2-3 > 0$

$\max(0,2) + v_1 + v[0,1]$

$= 0+1+0 = 1$

$v[1,3] = 3-3 > 0$.

$\max(0,3) + v_1 + v[0,2]$

$= 0+1+0 = 1$

ie $v[1,7] = 1$

$v[3,1]$ $j-w_3 = 1-4 < 0$

$v[2,1] = 1$

$v[3,2] = j-w_3 = 2-4 < 0$

$v[2,2] = 1$

$v[3,3] = j-w_3 = 3-4 < 0$

$v[2,3] = 4$

$v[3,4] = j-w_3 = 4-4 = 0$

$\max(v[2,4], 5+v[2,0])$

$= (5, 5+0) = 5$

$v[3,5] = j-w_3 = 5-4 > 0$

$\max(v[2,5], 5+v[2,2])$

$= (5, 5+1) = 6$

$v[3,6] = (5, 5+1) = 6$

$v[3,7] = (5, 5+4) = 9$

$v[2,1]$ $j-w_2 = 1-3 < 0$.

$v[i-1,j] = v[1,1] = 1$

$v[2,2]$ $j-w_2 = 2-3 < 0$

$v[i-1,j] = v[1,2] = 1$

$v[2,3] = j-w_2 = 3-3 = 0$

$\max(v[1,3], 4+v[1,0]) = \max(1, 4+0) = 4$

$v[2,4] = j-w_2 = 4-3 > 0$

$\max(v[1,4], 4+v[1,1]) = \max(1, 4+1) = 5$

$v[2,5] = j-w_2 = 5-3 > 0$

$\max(v[1,5], 4+v[1,2]) = \max(1, 4+1) = 5$

ie $v[2,7] = 7-3 \geq 0$

$\max(v[1,7], 4+v[1,4]) = \max(1, 4+1) = 5$

$v[4,1] = j-w_4 = 1-5 < 0$ i.e $v[3,1] = 1$

$v[4,2] = 2-5 < 0$ i.e $v[3,2] = 1$

$v[4,3] = 3-5 < 0$ i.e $v[3,3] = 4$.

$v[4,4] = 4-5 < 0$ i.e $v[3,4] = 5$

$v[4,5] = 5-5 = 0$.

$\max(v[3,5], 7+v[3,0]) = (6, 7+0) = 7$

$v[4,6] = 6-5 > 0$

$\max(v[3,6], 7+v[3,1]) = (6, 7+1) = 8$

$v[4,7] = 7-5 > 0$

$\max(v[3,7], 7+v[3,2]) = (9, 7+1) = 9$

$v[4\ 7] = v[3,7]$

$\therefore$ row 4 is not selected

$3[3,7] \neq v[2,7]$.

$\therefore$ item 3 is selected

$7 - 4 = 3$.

Later to represent by element $[2,6]$

$v[2,6] \neq v[1,6]$

$\therefore$ item 2 is selected

$3 - 3 = 0$.

$\therefore$ Final set $\{$item 2, item 3$\}$

item 2 3, 4

item 3 4, 5

7, 9

<u>Algorithm</u>.

```
0/1KNAPSACK(P[ ], w[ ], n, M)
// n number of items, M. capacity
  for j := 0 to M  c[0, j] := 0;
  for i := 0 to n  c[i, 0] := 0

  for i := 1 to n.
     for j = 1 to M
        if (w[i] > j)   // can't pick item.
          c[i,j] := c[i-1, j];
        else
          if (P[i] + c[i-1, j-w(i)]) > c[i-1, j])
            c[i,j] := P[i] + c[i-1, j-w[i]]
          else
            c[i,j] := c[i-1, j];

  return c[n, M];
```

<u>complexity</u> - $\Theta(nM)$

THE TRAVELING SALESPERSON PROBLEM

- Let $G = \langle V, E \rangle$ be a directed graph with edge costs c_{ij}.
- The variable c_{ij} is defined such that

  $c_{ij} > 0$ for all i and j

  $c_{ij} = \infty$ if $(i,j) \notin E$
- 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 salesperson problem is to find a tour of minimum cost.
- A tour to be a simple path that starts and ends at vertex 1.
- Every tour consists of an edge $(1,k)$ for some $k \in V - \{1\}$ and a path from vertex k to vertex 1.
- The path from vertex k to vertex 1 goes through each vertex in $V - \{1,k\}$ exactly once.
- If the tour is optimal, then the path from k to 1 must be a shortest k to 1 path going through all vertices in $V - \{1,k\}$.
- Hence principle ~~opti~~ of optimality holds.
- Let $g(i,S)$ be the length of a 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

$$g(1, V-\{1\}) = \min_{2 \le k \le n} \{ c_{1k} + g(k, V-\{1,k\}) \} \rightarrow (1).$$

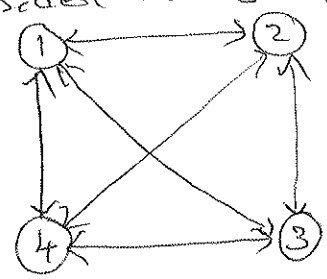
Generalizing eq. (1), we obtain (for $i \notin S$)

$$g(i, S) = \min_{j \in S} \{ c_{ij} + g(j, S-\{j\}) \} \rightarrow (2)$$

- Eq. (2) can be solved for $g(1, V-\{1\})$ if we know $g(k, V-\{1,k\})$ for all choices of k.
- Clearly $g(i, \phi) = c_{i1}$, $1 \le i \le n$.
- Hence we can use eq. (2) to obtain $g(i, S)$ for all S of size 1.
- Then we can obtain $g(i, S)$ for S with $|S| = 2$, and so on.

Example 1

Consider the graph, the edge lengths are given by matrix C.



$$C = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 10 & 15 & 20 \\ 5 & 0 & 9 & 10 \\ 6 & 13 & 0 & 12 \\ 8 & 8 & 9 & 0 \end{bmatrix} \end{array}$$

starting vertex = 1.

$g(i, \phi) = c_{i1}$

$g(2, \phi) = c_{21} = 5$

$g(3, \phi) = c_{31} = 6$

$g(4, \phi) = c_{41} = 8.$

$S = 1$ consider set of 1 element. i.e. $\{2\}, \{3\}, \{4\}$.

set $\{2\}$. $g(2, \{3\}) = c_{23} + g(3, \phi)$

$= 9 + 6 = 15$

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

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

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

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

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

$S = 2$. consider set of 2 elements $\{2,3\}, \{2,4\}, \{3,4\}$

$$g(2,\{3,4\}) = \min\{c_{23} + g(3,\{4\}),\ c_{24} + g(4,\{3\})\}$$
$$= \min\{9+20,\ 10+15\}$$
$$= 25.$$

$$g(3,\{2,4\}) = \min\{c_{32} + g(2,\{4\}),\ c_{34} + g(4,\{2\})\}$$
$$= \min\{13+18,\ 12+13\}$$
$$= 25$$

$$g(4,\{2,3\}) = \min\{c_{42} + g(2,\{3\}),\ c_{43} + g(3,\{2\})\}$$
$$= \min\{8+15,\ 9+18\}$$
$$= 23.$$

$S = 3$ consider set of 3 elements $\{2,3,4\}$.

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

$\therefore$ An optimal tour of the graph has length 35.

. A tour of this length can be constructed if we retain with each $g(i,S)$ the value of j that minimizes the right-hand side of equation (2).

Let $J(i,S)$ be this value.

Then $J(1,\{2,3,4\}) = 2$.

Thus the tour starts from 1 and goes to 2

The remaining tour obtained from $g(2,\{3,4\})$

$\therefore J(2,\{3,4\}) = 4$.

Thus the next edge is $(2,4)$

The remaining tour is for $g(4,\{3\})$

$J(4,\{3\}) = 3$.

$\therefore$ The optimal tour is 1, 2, 4, 3, 1

Algorithm

$c(S, j) = \min c(S - \{j\}, i) + d(i, j)$ where $i \in S$ and $i \neq j$

$c(\{1\}, 1) = 0$

for $s = 2$ to n do

for all subsets $S \in \{1, 2, 3 \cdots n\}$ of size s and containing 1

$c(S, 1) = \infty$.

for all $j \in S$ and $j \neq 1$

$c(S, j) = \min\{c(S - \{j\}, i) + d(i, j)$ for $i \in S$ and $i \neq j\}$

return $\min_j c(\{1, 2, 3 \cdots, n\}, j + d[j, i]$

Time complexity $= \underline{O(2^n \cdot n^2)}$

Example 2

$$C = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 2 & 9 & 10 \\ 2 & 1 & 0 & 6 & 4 \\ 3 & 15 & 7 & 0 & 8 \\ 4 & 6 & 3 & 12 & 0 \end{array}$$

Example 3

$$\begin{bmatrix} 0 & 1 & 15 & 6 \\ 2 & 0 & 7 & 3 \\ 9 & 6 & 0 & 12 \\ 10 & 4 & 8 & 0 \end{bmatrix}$$

starting vertex = 1.

$g(i, \phi) = c_{i1}$

$g(2, \phi) = 1$, $g(3, \phi) = 15$, $g(4, \phi) = 6$.

$s = 1$ consider set of 1 element i.e $\{2\}, \{3\}, \{4\}$.

$g(2, \{3\}) = c_{23} + g(3, \phi) = 6 + 15 = 21$

$g(2, \{4\}) = c_{24} + g(4, \phi) = 4 + 6 = 10$

$g(3, \{2\}) = c_{32} + g(2, \phi) = 7 + 1 = 8$

$g(3, \{4\}) = c_{34} + g(4, \phi) = 8 + 6 = 14$

$g(4, \{2\}) = c_{42} + g(2, \phi) = 3 + 1 = 4$

$g(4, \{3\}) = c_{43} + g(3, \phi) = 12 + 15 = 27$

$s = 2$ consider set of 2 elements $\{2,3\}, \{2,4\}, \{3,4\}$.

$$g(2,\{3,4\}) = \min\{c_{23} + g(3,\{4\}),\ c_{24} + g(4,\{3\})\}$$
$$= \min(6+14,\ 4+27) = 20$$

$$g(3,\{2,4\}) = \min(c_{32} + g(2,\{4\}),\ c_{34} + g(4,\{2\}))$$
$$= \min(7+10,\ 8+4) = 12$$

$$g(4,\{2,3\}) = \min(c_{42} + g(2,\{3\}),\ c_{43} + g(3,\{2\}))$$
$$= \min(3+21,\ 12+8) = 20$$

$s = 3$ consider set of 3 elements $\{2,3,4\}$.

$$g(1,\{2,3,4\}) = \min\{c_{12} + g(2,\{3,4\}),\ c_{13} + g(3,\{2,4\}),\ c_{14} + g(4,\{2,3\})\}$$
$$= \min\{\underset{22}{2+20},\ \underset{21}{9+12},\ \underset{30}{10+20}\}$$
$$= 21$$

$g(1,\{2,3,4\}) \rightarrow J(1,\{2,3,4\}) = 2$

$1 - 2$

$g(2,\{3,4\}) \rightarrow J(2,\{3,4\}) = 3$

$g(3,\{4\}) \rightarrow J(3,\{4\}) = 4$

$1 - 2 - 3 - 4 - 1$

BACKTRACKING

1. THE GENERAL METHOD.

- In the search for fundamental principles of algorithm design, backtracking represents one of the most general techniques.
- The name backtrack was first coined by D.H. Lehmer in the 1950s.
- In many applications of the backtrack method, the desired solution is expressible as an n-tuple $(x_1, x_2 \cdots x_n)$, where the x_i are chosen from some finite set S_i
- Often the problem to be solved calls for finding one vector that maximizes (or minimizes or satisfies) a ~~criteria~~ criterion function $P(x_1, x_2 \cdots x_n)$.
- Sometimes it seeks all vectors that satisfy P.
- Suppose m_i is the size of set S_i. Then there are $m = m_1 m_2 \cdots m_n$ n-tuples that are possible candidates for satisfying the function P.
- The brute force approach would be to form all these n-tuples, evaluate each one with P, and save those which yield the optimum.
- The backtrack algorithm has the ability to yield the same answer with far fewer than m trails.
- Its basic idea is to build up the solution vector one component at a time and to use modified criterion functions $P_i(x_1, x_2 \cdots x_i)$ to test whether the vector being formed has any chance of success.

Advantage: If it is realized that the partial vector $(x_1, x_2 \cdots x_i)$ can in no way to lead to an optimal solution, then $m_{i+1} \cdots m_n$ possible test vectors can be ignored entirely.

- Many of the problems we solve using backtracking require that all the solutions satisfy a complex set of constraints.
- Constraints can be divided into two categories
  1. Explicit 2. Implicit.
- <u>Explicit</u> constraints are rules that restrict each x_i to take on values only from a given set.

  <u>ex</u> ① $x_i \geq 0$ or $x_i = 0$ or 1, $l_i \leq x_i \leq u_i$.

  ② $S_i = \{$all nonnegative real nos$\}$, $S_i = \{0, 1\}$.

  <u>Implicit</u> constraints are rules that determine which of the tuples in the solution space of I satisfies the criterion function.
- Thus implicit constraints describe the way in which the x_i must relate to each other.

<u>Example</u>: <u>8-queens</u>.

- A classic combinatorial problem is to place eight queens on an 8x8 chessboard so that no two attack. i.e. no two of them are on the same row, column or diagonal.
- All solutions to the 8-queens problem can therefore be represented as 8-tuples $(x_1, x_2 \cdots x_8)$, where x_i is the column on which queen i is to be placed ~~on row i~~.
- The <u>explicit</u> constraints using this formulation are

  $S_i = \{1, 2, 3, 4, 5, 6, 7, 8\}$. $1 \leq i \leq 8$.
- The <u>implicit</u> constraints for this problem are that no two x_i's can be the same and no two queens can be on the same diagonal.

Example 2. Sum of subsets.

- Given positive numbers w_i, $1 \le i \le n$, and m, this problem calls for finding all subsets of the w_i whose sums are m.
- All solutions are k-tuples $(x_1, x_2 \cdots x_k)$, $1 \le k \le n$, and different solutions may have different-sized tuples.
- The explicit constraints require $x_i \in \{ j \mid j$ is an integer and $1 \le j \le n \}$.
- The implicit constraints require that no two be the same and that the sum of the corresponding w_i's be m.

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

- Backtracking algorithms determine problem solutions by systematically searching the solution space for the given problem instance.
- This search is facilitated by using a tree organization for the solution space.
- For a given solution space many tree organizations may be possible.
- Let $(x_1, x_2 \cdots x_i)$ be a path from the root to a node in a state space tree.
- The tree organization of the solution space is referred to as the state space tree.
- Let $T(x_1, x_2 \cdots x_i)$ be the set of all possible values for x_{i+1} such that $(x_1, x_2 \cdots x_{i+1})$ is also a path to a problem state.
- $T(x_1, x_2 \cdots x_n) = \phi$.

- We assume the existence of bounding function B_{i+1} such that if $B_{i+1}(x_1, x_2 \cdots x_{i+1})$ is false for a path $(x_1, x_2 \cdots x_{i+1})$ from the root node to a problem state, then the path can't be extended to reach an answer node.
- Thus the candidates for position $i+1$ of the solution vector $(x_1, x_2 \cdots x_n)$ are those values which are generated by T and satisfy B_{i+1}.

<u>Recursive Backtracking Algorithm.</u>

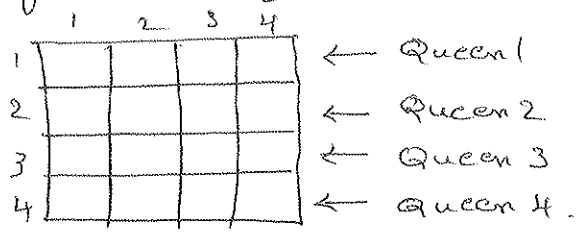
```
Algorithm. Backtrack(k).
// This schema describes the backtracking process
// using recursion. On entering the first k-1
// values x[1], x[2] ... x[k-1] of the solution
//vector x[1:n] have been assigned x[] and n are global.

{ for(each x[k] ∈ T (x[1], .. x[k-1]) do
  {
    if (B_k(x[1], x[2] .. x[k]) ≠ 0 then
    {
      if (x[1], x[2], ... x[k] is a path
            to an answer node) then
        write (x[1:k]);
      if (k < n) then Backtrack(k+1);
    }
  }
}.
```

General Iterative backtracking method.

```
Algorithm IBacktrack(n)
// This schema describes the backtracking process.
// All solutions are generated in x[1:n] and
// printed as soon as they are determined.
{ k := 1;
  while (k ≠ 0) do
  {
    if (there remains an untried
      x[k] ∈ T(x[1], x[2] ... x[k-1]) and
      B_k(x[1], ... x[k]) is true) then
    {
      if (x[1], ... x[k] is a path to an
      answer node) then
        write(x[1:k]);
        k := k+1 // consider the next set.
    }
    else
        k := k-1; // Backtrack to the previous set.
  }
}
```

n-Queens Problem

- The problem is to place n queens on an n-by-n chess board so that no two queens attack each other by being in the same row or in the same column or on the same diagonal.
- For n=1, the problem has a trivial solution, and it is easy to see that there is no solution for n=2 & n=3.
- Let us consider four-queen problem and solve it by backtracking technique
- Since each of the four queens has to be placed in its own row, all we need to do is to assign a column for each queen on the board

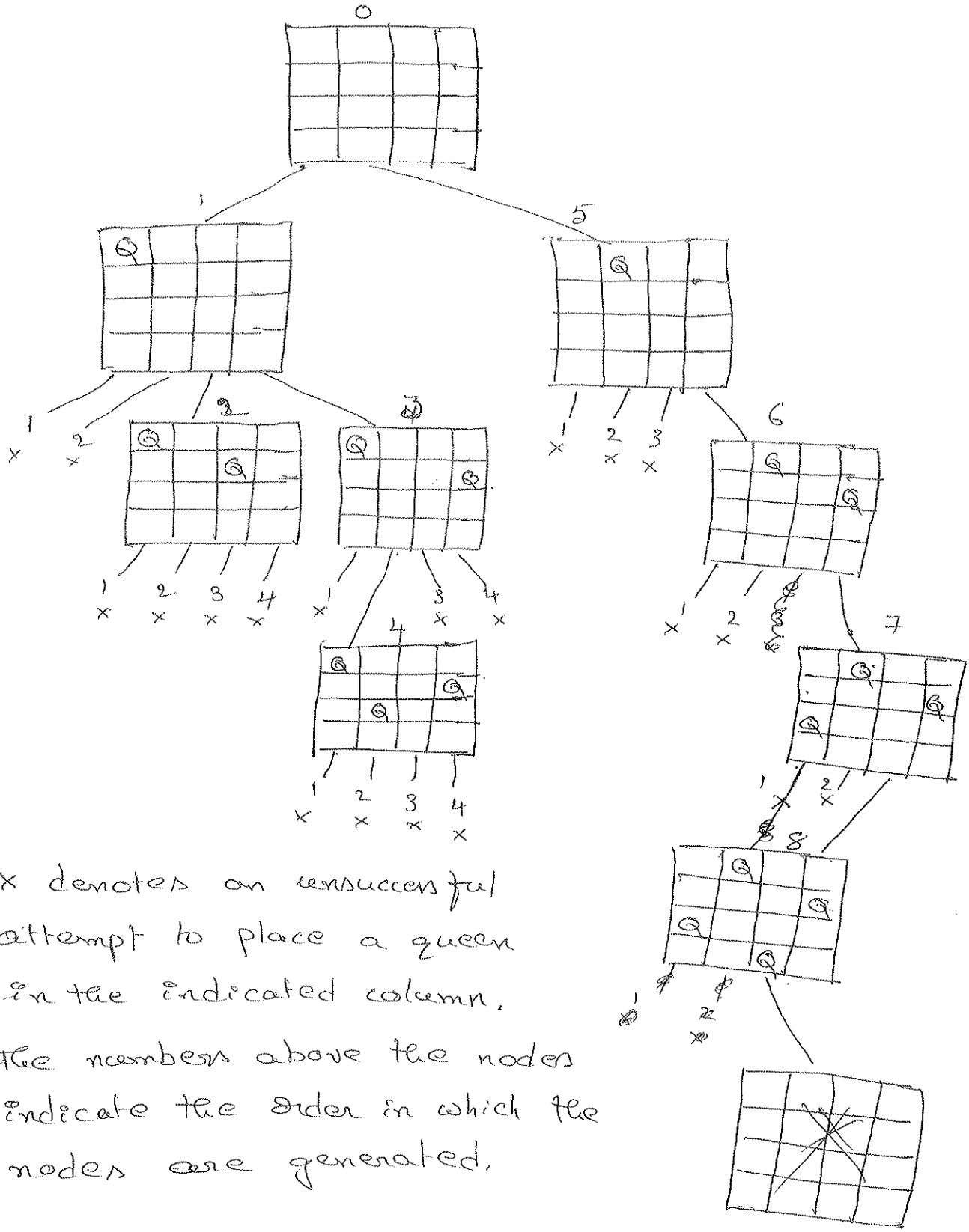


- We start with the empty board and then place Queen 1 in the first possible position of its row, which is in column 1 and row 1.

  (1,1) ← Q1

- Then we place Queen 2, after trying unsuccessfully column 1 and 2, in the first acceptable position for it, which is square (2,3), the square in row 2 and column 3.
- This proves to be a dead end because there is no acceptable position for queen 3.
- So, the algorithm backtracks and puts queen 2 in the next possible position at (2,4).
- Then queen 3 placed at (3,2), which proves to be another dead end.
- The algorithm then backtracks all the way to queen 1 and moves it to (1,2)

· Queen 2 then goes to (2,4), Queen 3 to (3,1) and Queen 4 to (4,3), which is a solution to the problem.



· x denotes an unsuccessful attempt to place a queen in the indicated column.

· The numbers above the nodes indicate the order in which the nodes are generated.

- Let $(x_1, x_2 \cdots x_n)$ represent a solution in which x_i is the column of the ith row where the ith queen is placed.
- The x_i's will all be distinct since no two queens can be placed in the same column.
- If we imagine the chessboard square being numbered as the indices of the two-dimensional array $a[1:n, 1:n]$ then we observe that every element on the same diagonal that runs from the upper left to the lower right has the same row-column value.
- All these squares have a row-column value of 2.
- Also, every element on the same diagonal that goes from the upper right to the lower left has the same row+column value.
- Suppose two queens are placed at positions (i, j) and (k, l).
- Then by the above they are on the same diagonal only if

$$i - j = k - l \quad \text{or} \quad i + j = k + l$$

The first equation implies

$$j - l = i - k$$

The second implies

$$j - l = k - i$$

Therefore two queens lie on the same diagonal if and only if $|j - l| = |i - k|$.

- <u>Place(k, i)</u> returns a boolean value that is true if the kth queen can be placed in column i.
- It tests both whether i is distinct from all previous values $x[1] \cdots x[k-1]$ and wether there is no other

queen on the same diagonal.

Its computing time is $O(k-1)$.

- Using Place, we can refine the general backtracking method NQueens(1, n).

<u>Algorithm</u> <u>Place(k, i)</u>

```
// Returns true if a queen can be placed in kth row
// and ith column. Otherwise it returns false. x[] is
// a global array whose first (k-1) values have been set.
// Abs(r) returns the absolute value of r.
{ for j := 1 to k-1 do
    if ((x[j] = i)  or  // Two in the same column.
        (Abs(x[j] - i) = Abs(j-k))) // or in the same diagonal
    then return false;
  return true;
}.
```

Algorithm NQueens(k, n)

```
// Using backtracking, this procedure prints all
// possible placements of n queens on an nxn
// chessboard so that they are nonattacking.
{ for i := 1 to n do
  {  if Place(k, i) then
     {
        x[k] := i;
        if (k=n) then write (x[1:n]);
        else
           NQueens(k+1, n);
     }
  }
}.
```

Sum of Subsets.

- Given n distinct positive numbers (usually called weights) and we desire to find all combinations of these numbers whose sums are m.
- This is called the sum of subsets problem.
- We could formulate this problem using either fixed- or variable-sized tuples.
- We consider a backtracking solution using the fixed tuple size strategy.
- The element x_i of the solution vector is either <u>one</u> or <u>zero</u> depending on whether the weight w_i is included or not.
- The children of any node are easily generated.
- For a node at level i the left child corresponds to $x_i = 1$ and the right to $x_i = 0$.
- A simple choice for the bounding function is

$$B_K(x_1, \cdots x_K) = \text{true iff}$$

$$\sum_{i=1}^{K} w_i x_i + \sum_{K=K+1}^{n} w_i \geq m$$

- Clearly $x_1 \cdots x_K$ can't lead to an answer node if this condition is not satisfied.
- The bounding function can be strengthened if we assume that w_i's are initially in nondecreasing order.
- $x_1, \cdots x_K$ can't lead to an answer node if

$$\sum_{i=1}^{K} w_i x_i + w_{K+1} > m$$

$\therefore$ The bounding function is

$$B_K(x_1 \cdots x_K) = \text{true iff } \sum_{i=1}^{K} w_i x_i + \sum_{i=K+1}^{n} w_i \geq n.$$

$$\text{and } \sum_{i=1}^{K} w_i x_i + w_{K+1} \leq m.$$

- Since our algorithm will not make use of B_n, we need not be concerned by the appearance of w_{n+1} in this function.
- Algorithm avoids computing $\sum_{i=1}^{k} w_i x_i$ and $\sum_{i=k+1}^{n} w_i$ each time by keeping these values in variables s and r respectively.
- The algorithm assumes $w_1 \le m$ and $\sum_{i=1}^{n} w_i \ge m$.
- The initial call is SumOfSub$(0, 1, \sum_{i=1}^{n} w_i)$.

```
Algorithm. SumOfSub(s, k, r)
// Find all subsets of w[1:n] that sum to m.
// The values of x[j], 1 ≤ j < k, have already been
// determined. s = Σ_{j=1}^{k-1} w[j]*x[j] and r = Σ_{j=k}^{n} w[j].
// The w[j]'s are in nondecreasing order. It is
// assumed that w[1] ≤ m and Σ_{i=1}^{n} w[i] ≥ m.
{
  // Generate left child. Note s + w[k] ≤ m since B_{k-1} is true.
  x[k] := 1;
  if (s + w[k] = m) then write(x[1:k]);
  // Subset found. There is no recursive call here
  // as w[j] > 0, 1 ≤ j ≤ n.
  else if (s + w[k] + w[k+1] ≤ m)
        then SumOfSub(s + w[k], k+1, r - w[k]);
  // Generate right child and evaluate B_k
  if ((s + r - w[k] ≥ m) and (s + w[k+1] ≤ m)) then
  {
     x[k] := 0;
     SumOfSub(s, k+1, r - w[k]);
  }
}.
```

Example

Let $n = 6$, $w[1:6] = \{5, 10, 12, 13, 15, 18\}$ and $m = 30$. Find all possible subsets of w that sum to m. Draw the portion of the state space tree that is generated.

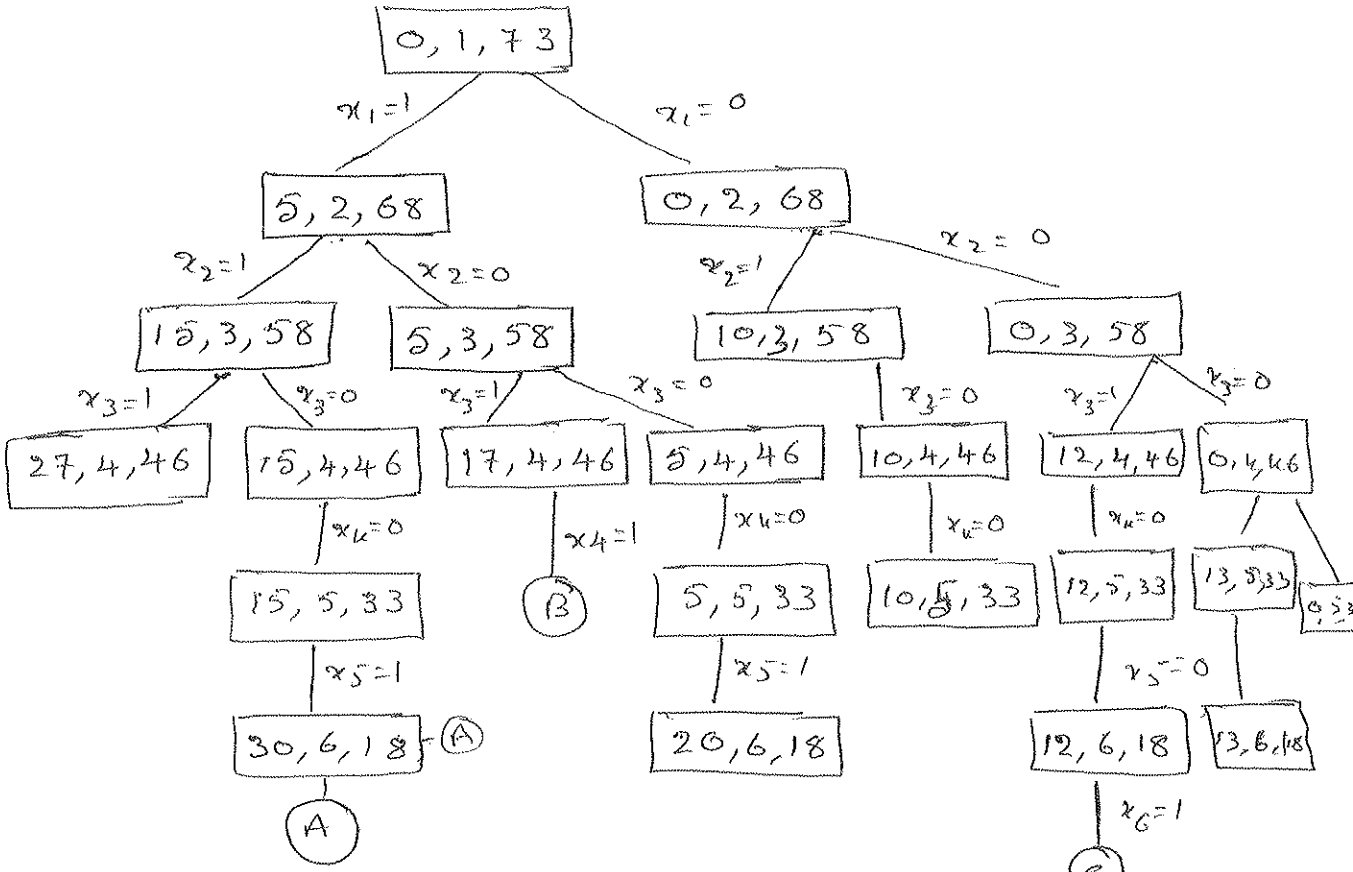
- The rectangular node list the values of s, k and r on each of the calls to SumOfSub.
- Circular nodes represent points at which subsets with sums m are printed out.

$w = \{5, 10, 12, 13, 15, 18\}$.

$A = (1, 1, 0, 0, 1)$, $B = (1, 0, 1, 1)$ $C = (0, 0, 1, 0, 0, 1)$

$\sum_{i=1}^{6} w_i = 73$.

Portion of state space tree.



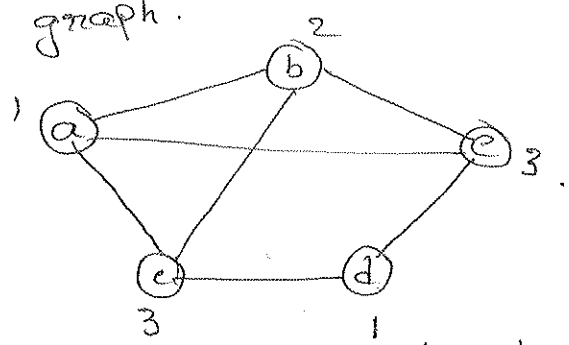
This tree contains only 23 rectangular nodes. The full tree for $n=6$ contains $2^6 - 1 = 63$ nodes from which calls could be made.

Example

Let $w = \{5, 7, 10, 12, 15, 18, 20\}$ and $m = 35$. Find all possible subsets of w that sum to m. Draw the portion of the state space tree that is generated.

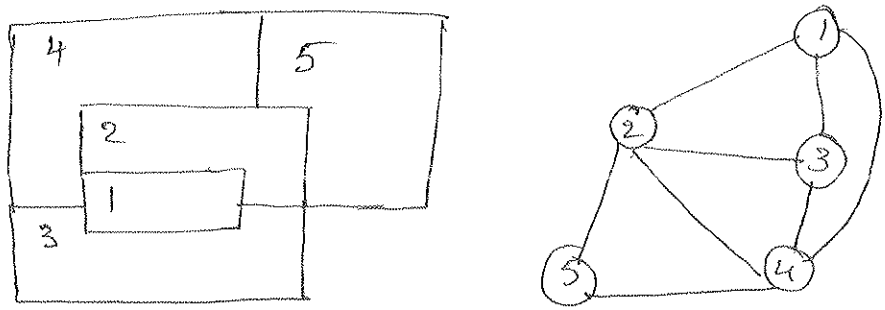
GRAPH COLORING.

- Let G be a graph and m be a given positive integer.
- We want to discover whether the nodes of G can be colored in such a way that no two adjacent nodes have the same color yet only m colors are used.
- This is termed the <u>m-colorability</u> decision problem.
- 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 to as the <u>chromatic number</u> of the graph.



- This graph can be colored with three colors 1, 2, and 3.
- The color of each node is indicated next to it.
- It can also be seen that three colors are needed to color this graph and hence this graph's <u>chromatic</u> number is 3.
- A graph is said to be <u>planar</u> iff it can be drawn in a plane in such a way that no two edges cross each other.
- A famous special case of the m-colorability decision problem is the 4-color problem for planar graphs.
- Given a map, can the regions be colored in such a way that no two adjacent regions have the same color ye only four colors are needed?

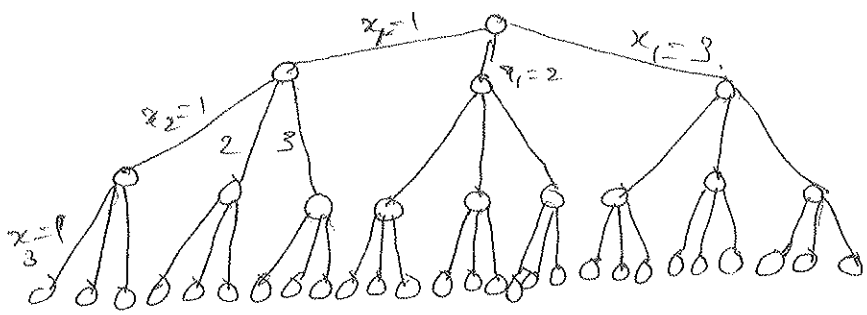
- A map can easily be transformed into graph.
- Each region of the map becomes a node, and if two regions are adjacent then the corresponding nodes are joined by an edge.



Map.

- This map requires four colors.
- We are determining all the different ways in which a given graph can be colored using at most m colors.
- Suppose we represent a graph by its adjacency matrix $G[1:n, 1:n]$, where $G[i,j] = 1$ if if (i,j) is an edge of G and $G[i,j] = 0$ otherwise.
- The colors are represented by the integers $1, 2 \cdots m$ and the solutions are given by the n-tuple $\{x_1, \cdots x_n\}$. where x_i is the color of node i.
- The underlying state space tree used is a tree of degree m and height $n+1$.
- Each node at level i has m children corresponding to the m possible assignments to x_i, $1 \le i \le n$.
- Nodes at level $n+1$ are leaf nodes.

The state space tree when $n = 3$ and $m = 3$.



Algorithm mColoring(k)

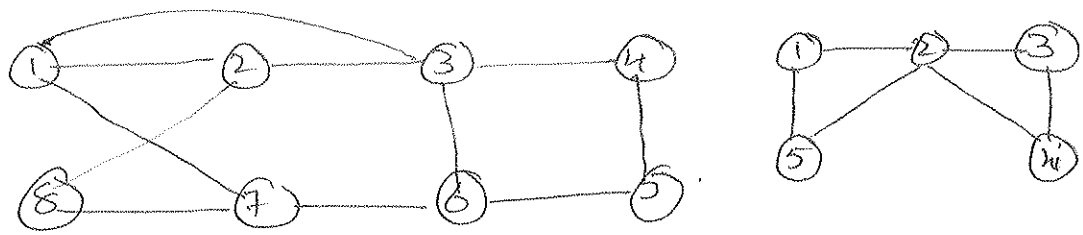
```
// This algorithm was formed using the recursive backtracking
// schema. The graph is represented by its boolean adjacency
// matrix G[1:n, 1:n]. All assignments of 1, 2 .. m to the vertices
// of the graph such that adjacent vertices are assigned
// distinct integers are printed. k is the index of the next vertex
// to color.
{ repeat
  { // Generate all legal assignments for x[k]
    NextValue(k); // Assign to x[k] a legal color.
    if (x[k]=0) then return // No new color possible.
    if (k=n) then  // At most m colors have been used to color
                   the n vertices.
      write(x[1:n]);
    else mCololoring(k+1);
  }until (false);
}
```

Algorithm NextValue(k)

```
// x[1] ... x[k-1] have been assigned integer values in the range
// [1, m] such that adjacent vertices have distinct integers. A
// value for x[k] is determined in the range [0, m]. x[k] is assigned
// the next highest numbered color while maintaining distinctness from
// the adjacent vertices of vertex k. If no such color exists, then x[k] is 0.
{ 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
    { // check if this color is distinct from adjacent colors.
      if ((G[k, j] ≠ 0) and (x[k] = x[j]))
      // If (k, j) is and edge and if adj vertices have the
                 same color.
      then break;
    }
    if (j = n+1) then return; // New color found.
  }until (false); // Otherwise try to find another color.
}
```

HAMILTONIAN CYCLES

- 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 and returns to its starting position.
- In other words, it begins at some vertex $v_1 \in G$ and vertices of G are visited in the order $v_1 v_2 \cdots v_{n+1}$, then the edges (v_i, v_{i+1}) are in E, $1 \le i \le n$, and v_i are distinct except for v_1 and v_{n+1} which are equal.

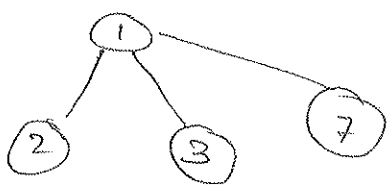


- The Hamiltonian cycle for the graph are

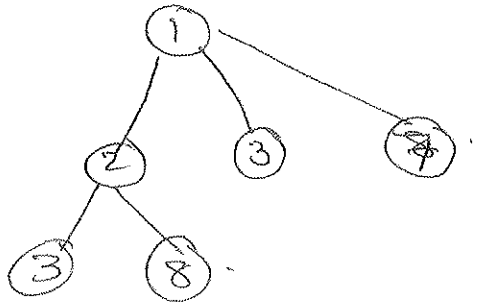
$$1 - 2 - 8 - 7 - 6 - 5 - 4 - 3 - 1$$

$$1 - 3 - 4 - 5 - 6 - 7 - 8 - 2 - 1$$

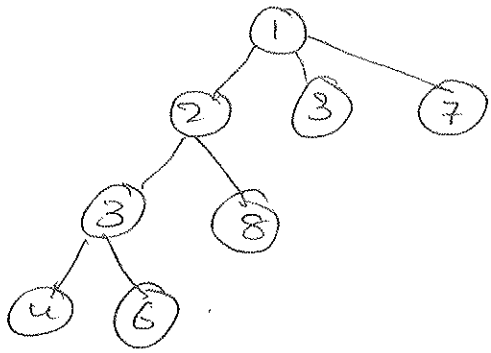
- The backtracking solution vector $(x_1 \cdots x_n)$ is defined so that x_i represents the ith visited vertex of the proposed cycle.
- State space tree.
- The children of node 1 are nodes that are adjacent to 1



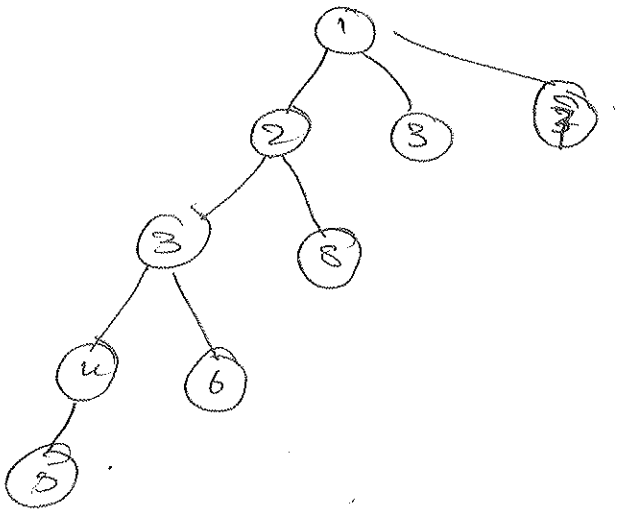
2. The adjacent nodes of 2 are 1, 3, 8. ∵ 1 is already traversed only 3 & 8 are represented.



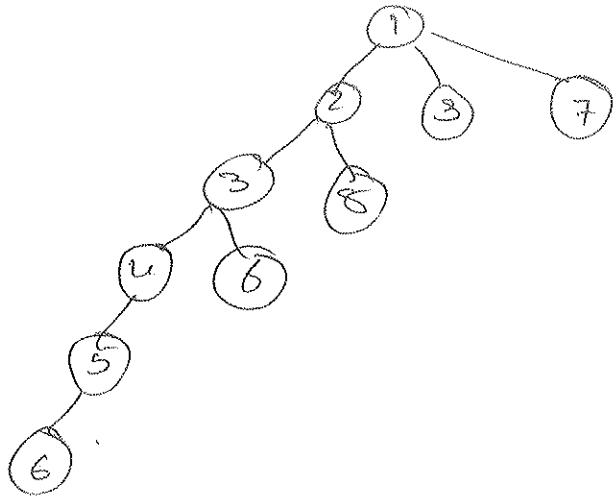
3. The adjacent 3 are 1, 2, 4, 6. Since 1, 2 are traversed. only 4 and 6 are represented.



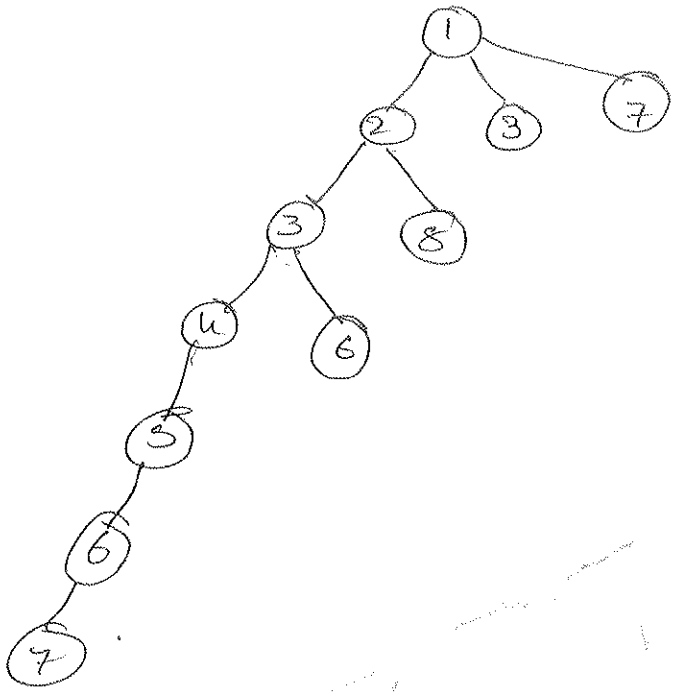
4. The adjacent of 4 are 3, 5. Since 3 are already visited. so 5 only represented.



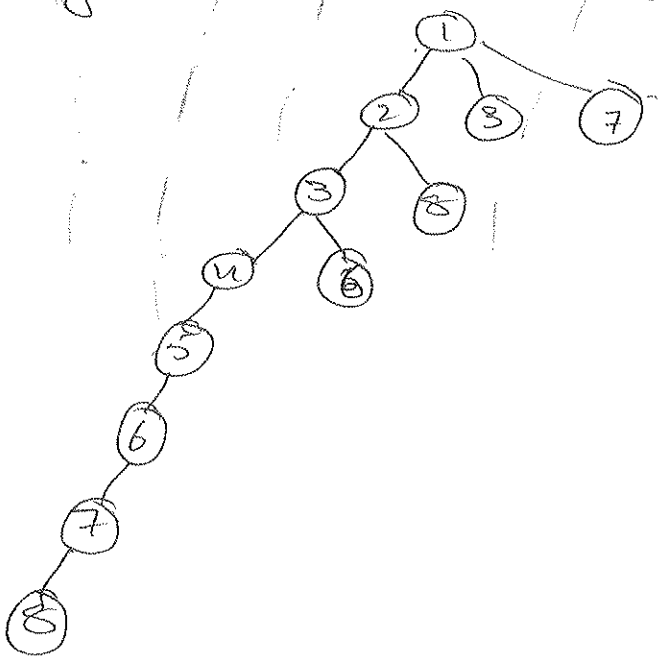
5. The adjacent of are 4, 6. Since 4 is already visited ~~so~~ 6 only represented.



6. The adjacent of 6 are 3, 5, 7. Since 3, 5 are already visited, only 7 is represented.

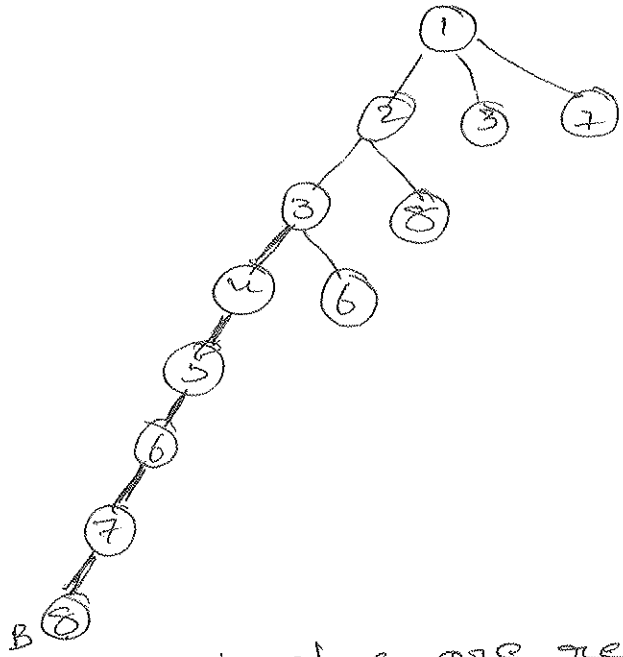


7. The adjacent of 7 are 1, 6, 8. Since 1, 6 are already traversed. only 8 is represented.

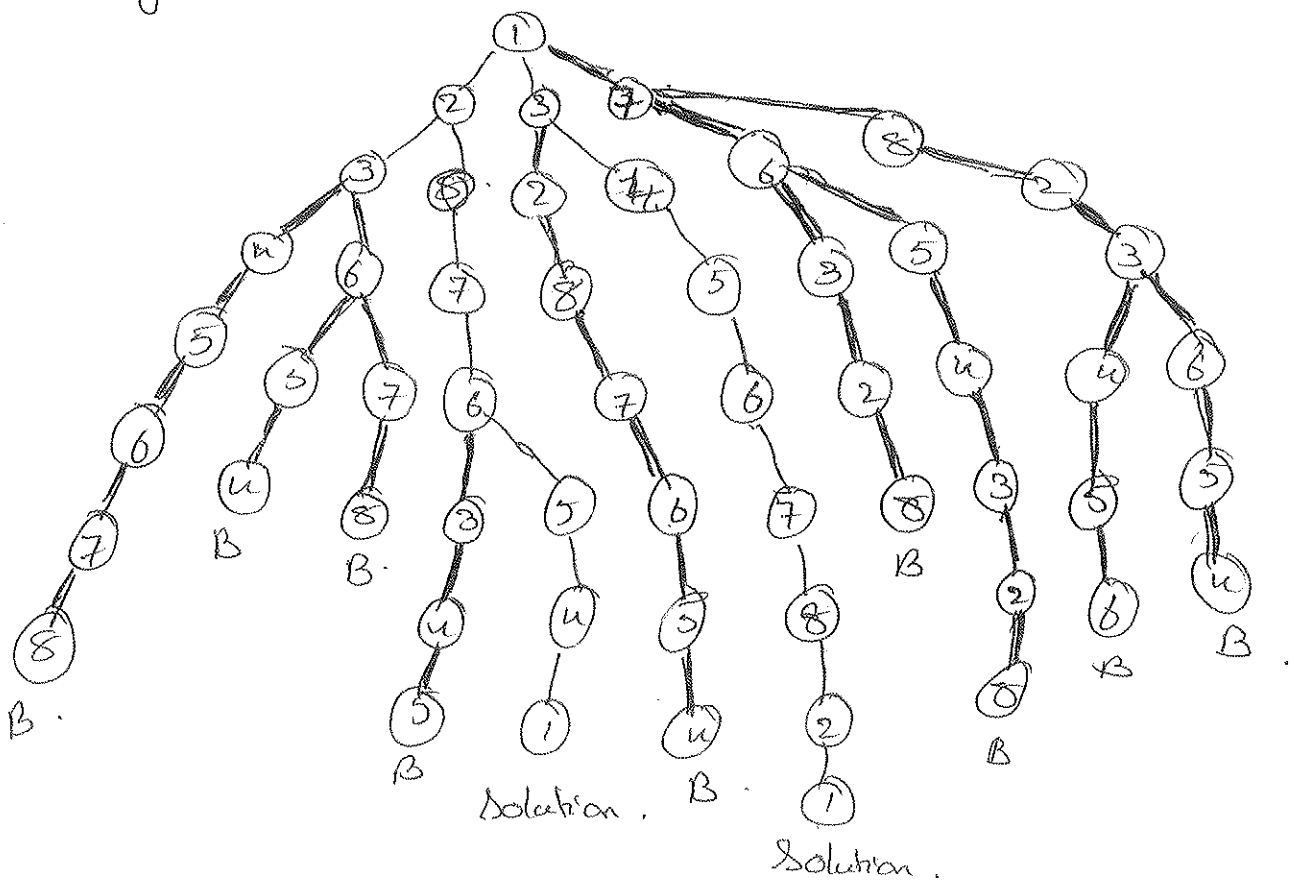


8. The adjacent nodes of 8 are 2, ~~8~~, 7. Since Both are traversed,

9. Backtrack and search for alternate solution.



Backtracked edges are represented with darked edges. Continuing in the same way we get the following tree.



Algorithm Hamiltonian(k)

```
// This algorithm uses the recursive formulation
// of backtracking to find all the Hamiltanian
// cycles of a graph. The graph is stored
// as an adjacency matrix G[1:n, 1:n].
// All cycles begin at node 1.
{
  repeat
   { // Generate values for x[k].
     NextValue(k) // Assign a legal next value to x[k].
     if (x[k] = 0) then return;
     if (k = n) then write(x[1:n]);
     else Hamiltonian(k+1);
   } until (false);
}.
```

Algorithm. NextValue(k).

```
// x[1:k-1] is a path of k-1 distinct vertices. If x[k]=0,
// then no vertex has as yet been assigned to x[k].
// After execution, x[k] is assigned to the next
// heighest numbered vertex which does not already
// appear in x[1:k-1] and is connected by an
// edge to x[k-1]. Otherwise x[k] = 0. If k = n then
// in addition x[k] is connected to x[1].

{ repeat:
  { x[k] := (x[k]+1) mod (n+1); // next vertex.
    if (x[k] = 0) then return;
    if (G[x[k-1], x[k]) ≠ 0 then
    { // Is there any edge?
      for j := 1 to k-1 do
        if (x[j] = x[k]) then break;
                        // check for distinctness.

        if (j = k) then // If true, then the
                           vertex is distict.
          if ((k < n) or ((k = n) and
                G[x[n], x[1]] ≠ 0)
          then return;
      }
  } until (false);
}.
```

UNIT V

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

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. > It also called "Tractable Algorithms" > Most Searching & Sorting algorithms are polynomial time algorithms > **Ex:** Ordered Search **(O (log n)),** Polynomial evaluation **O(n)** Sorting **O(n.log n)**	> Problems with solution times not bound by polynomial (simply non polynomial) > These are hard or intractable problems > None of the problems in this group has been solved by any polynomial time algorithm > **Ex:** Traveling Sales Person **$O(n^2\ 2^n)$** Knapsack **$O(2^{n/2})$**

No one has been able to develop a polynomial time algorithm for any problem in the 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 NP-Complete.

Nondeterministic Algorithms:

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

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

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

To specify nondeterministic algorithms, there are 3 new functions.

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

Failure (), Signals an Unsuccessful completion

Success (), Signals a successful completion.

Example for Non Deterministic algorithms:

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

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.

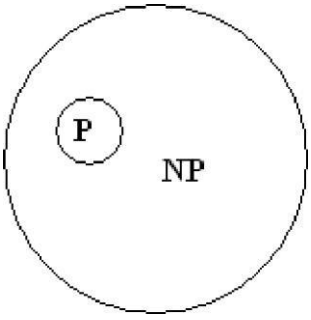
Decision problem/ Decision algorithm: Any problem for which the answer is either zero or one is decision problem. Any algorithm for a decision problem is termed a decision algorithm.

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

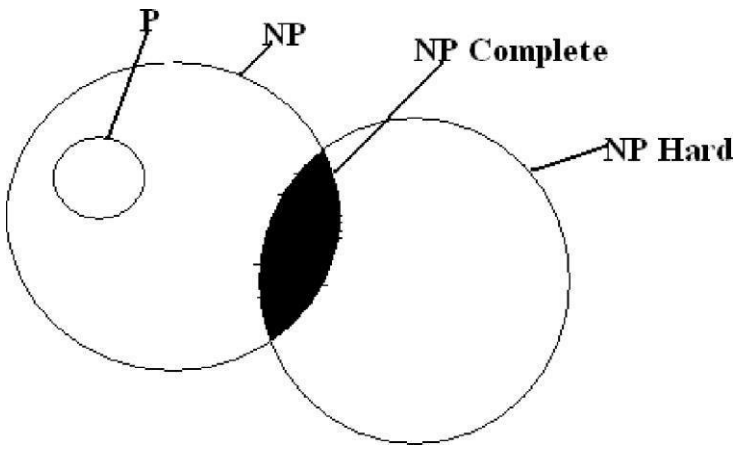
Let L_1 and L_2 be problems, Problem L_1 reduces to L_2 (written $\mathbf{L_1 \alpha L_2}$) iff there is a way to solve L_1 by a deterministic polynomial time algorithm using a deterministic algorithm that solves L_2 in polynomial time

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

Mostnatural 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 $\mathbf{O(p^3(n) \log n)=O(p^4(n))}$

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

> 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**' issatisfiable then the complexity of '**Z**' is $\mathbf{O(p^3(n) \log n + q(p^3(n)\log n))}$**.**

>If satisfiability is '**p**', then '**q(m)**' is a polynomial function of '**m**' and thecomplexity 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 canobtain a deterministic **Z** in **p.**

By this we shows that satisfiability is in **p** then **P=NP**

Trie is an efficient information retrieval data structure. Using Trie, search complexities can be brought to optimal limit (key length). If we store keys in a binary search tree, a well balanced BST will need time proportional to **M * log N**, where M is the maximum string length and N is the number of keys in the tree. Using Trie, we can search the key in O(M) time. However, the penalty is on Trie storage requirements (Please refer to Applications of Trie for more details)

Every node of Trie consists of multiple branches. Each branch represents a possible character of keys. We need to mark the last node of every key as the end of the word node. A Trie node field *isEndOfWord* is used to distinguish the node as the end of the word node. A simple structure to represent nodes of the English alphabet can be as follows,

```
// Trie node
struct TrieNode
{
    struct TrieNode *children[ALPHABET_SIZE];
    // isEndOfWord is true if the node
    // represents end of a word
    bool isEndOfWord;
};
```

Inserting a key into Trie is a simple approach. Every character of the input key is inserted as an individual Trie node. Note that the *children* is an array of pointers (or references) to next level trie nodes. The key character acts as an index to the array *children*. If the input key is new or an extension of the existing key, we need to construct non-existing nodes of the key, and mark the end of the word for the last node. If the input key is a prefix of the existing key in Trie, we simply mark the last node of the key as the end of a word. The key length determines Trie depth.

Searching for a key is similar to an insert operation, however, we only compare the characters and move down. The search can terminate due to the end of a string or lack of key in the trie. In the former case, if the *isEndofWord* field of the last node is true, then the key exists in the trie. In the second case, the search terminates without examining all the characters of the key, since the key is not present in the trie.

The following picture explains the construction of trie using keys given in the example below,

```
        root
      / \   \
      t  a   b
      |  |   |
      h  n   y
      |  | \ |
      e  s  y e
    / |  |
    i r  w
    | |  |
    r e  e
         |
```