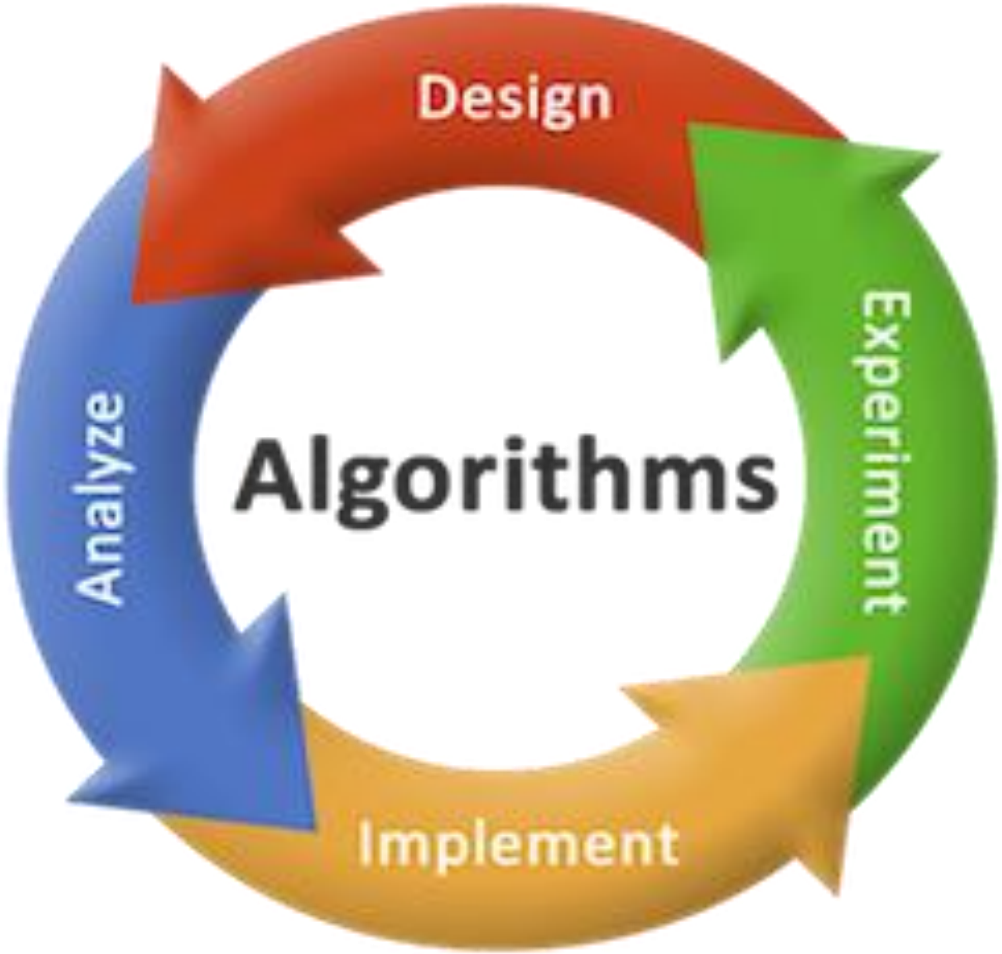
Design and analysis of algorithms

Prerequisites

- Should have basic knowledge of programming and Discrete mathematics.
- Should know the Data Structures very well.
- Should have basic understanding of Formal Language and Automata Theory

- Design and Analysis of Algorithms mainly includes



Course outcomes

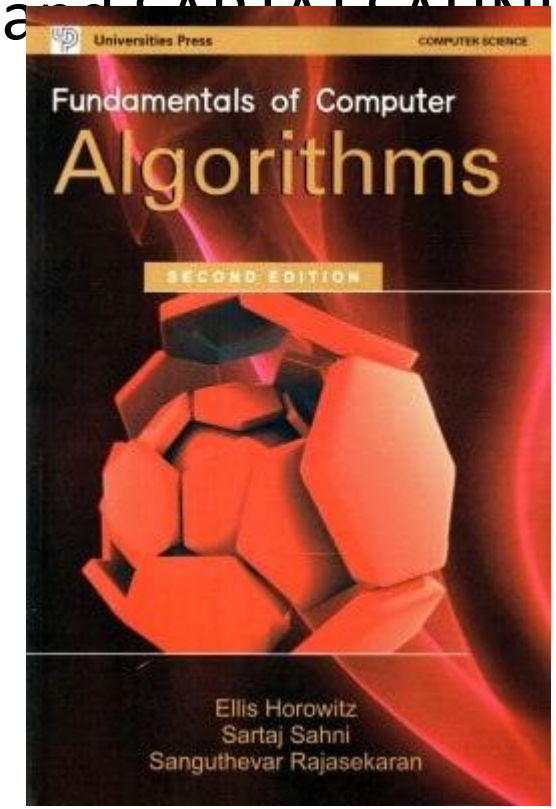
After successful completion of the course, the student will be able to:

- ❖Describe asymptotic notation and basic concepts of algorithms
- ❖Apply Divide and Conquer paradigm to solve various problems
- ❖Use Greedy technique to solve various problems
- ❖Apply Dynamic Programming technique to various problems
- ❖Employ Backtracking technique to various problems
- ❖Apply Branch and Bound technique to various problems

Text Book

Title: Fundamentals of Computer Algorithms.

Authors: ELLIS HOROWITZ a



Definition

Algorithm is a finite set of instructions that if followed, accomplishes a particular task.

All algorithms must satisfy the following criteria.

- Input: Zero or more quantities are externally supplied.
- Output: At least one quantity is produced.
- Definiteness: Each instruction is clear and unambiguous.
- Finiteness: If we trace out the instructions of an algorithm then for all cases the algorithm terminate after a finite number of steps.
- Effectiveness: Algorithm must not only definite and also feasible.

The study of algorithms mainly includes

- How to devise algorithms
- How to validate algorithms
- How to analyze algorithms
- How to test a program

BASIC TECHNIQUES FOR DESIGN OF EFFICIENT ALGORITHMS

There are basically 5 fundamental techniques which are used to design an algorithm efficiently:

- ❖Divide-and-Conquer
- ❖Greedy method
- ❖Dynamic Programming
- ❖Backtracking
- ❖Branch-and-Bound

Design strategy	**Problems that follows**
Divide & Conquer	▪ Binary search ▪ Multiplication of two n-bits numbers ▪ Quick Sort ▪ Heap Sort ▪ Merge Sort
Greedy Method	▪ Knapsack (fractional) Problem ▪ Minimum cost Spanning tree ✓ Kruskal's algorithm ✓ Prim's algorithm ▪ Single source shortest path problem ✓ Dijkstra's algorithm
Dynamic Programming	▪ All pair shortest path-Floyed algorithm ▪ Chain matrix multiplication ▪ Longest common subsequence (LCS) ▪ 0/1 Knapsack Problem ▪ Traveling salesmen problem (TSP)
Backtracking	▪ N-queen's problem ▪ Sum-of subset
Branch & Bound	▪ Assignment problem ▪ Traveling salesmen problem (TSP)

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.

Pseudo-Code Conventions for expressing algorithms:

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.

Pseudo-Code Conventions for expressing algorithms:

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 <, <=,>,>=, =, !=

Pseudo-Code Conventions for expressing algorithms:

7. The following looping statements are employed.
   For, while and repeat-until

While Loop:

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

Pseudo-Code Conventions for expressing algorithms:

For Loop:

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

Pseudo-Code Conventions for expressing algorithms:

repeat-until:

repeat

  <statement-1>

  .

  .

  .

  <statement-n>

until<condition>

Pseudo-Code Conventions for expressing algorithms:

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>

}

Pseudo-Code Conventions for expressing algorithms:

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)

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’.

Direct Recursion

```
int num()
{
      . . . . .
      . . . . .
      int num();
}
```

INDIRECT RECURSION

```
int num()
{
      . . . . .
      . . . . .
      int sum();
}
int sum()
{
      . . . . .
      . . . . .
      int num();
}
```

Factorial of Number

```
Algorithm Factorial(int n)
        {
            if n = = 1 then
                return 1
            else
                return factorial (n-1)*n
        }
```

Performance Analysis

Space Complexity:

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

Time Complexity:

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

❖Performance evaluation can be divided as:

- Priori Analysis
- Posteriori Analysis

Priori Analysis : It means we do analysis (space and time) of an algorithm prior to running it on a specific system.(It is independent of language of compiler and types of hardware.)

Posteriori Analysis : it means we do analysis of algorithm only after running it on a system. It directly depends on system and changes from system to system.(It is dependent on language of compiler and type of hardware.)

Space Complexity

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

- A fixed part - This part typically includes the instruction space (i.e., Space for the code), space for simple variable, space for constants and so on.

Space Complexity

- A variable part that consists of the space needed by component variables whose size is dependent on the particular problem instance being solved, and the recursion stack space.

Space Complexity

- 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.

- When analyzing the space complexity of an algorithm, we concentrate solely on estimating Sp (instance characteristics).

- For any given problem, we need first to determine which instance characteristics to use to measure the space requirements and this is very problem specific.

Example 1

```
Algorithm abc(a,b,c)
{
   return a+b+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 algorithm n, s occupies one word each and array ‘a’ occupies n number of words so S(P)>=n+3

Time Complexity

- Time Complexity of an algorithm represents the amount of time required by the algorithm to run to completion.
- 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(i.e. no. of inputs, no. of outputs, magnitude of inputs, magnitude of outputs etc.) . Also we may assume that a compiled program will be run several times without recompilation .

Time Complexity

- We are concerned with just the runtime of a program, This runtime is denoted by t_p (instance characteristics).
- We can determine the number of steps needed by a program to solve a particular problem instance in one of two ways.
  1. Count method
  2. s/e method (steps per execution)

Count Method

- In this method we introduce a variable, count into the program. This is a global variable with initial value 0.
- Statements to increment count by the appropriate amount are introduced into the program. This is done so that each time a statement in the original program is executed, count is incremented by the step count of that statement.

```
Algorithm sum(a,n)
{
   s= 0.0;
   count: = count+1; // count is global; it is initially zero
   for i:=1 to n do
   {
          count =count+1; // For for
          s:=s+a[i];
          count:=count+1; // For assignment
   }
   count:=count+1; // For last time of for
   count:=count+1; // For return
   return s;
} //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];
    }
}
```

❖When analyzing a recursive program for its step count, we often obtain a recursive formula for the step count as

$$t_{\text{RSum}}(n) = \begin{cases} 2 & \text{if } n = 0 \\ 2 + t_{\text{RSum}}(n-1) & \text{if } n > 0 \end{cases}$$

$$
\begin{aligned}
t_{\text{RSum}}(n) &= 2 + t_{\text{RSum}}(n-1) \\
&= 2 + 2 + t_{\text{RSum}}(n-2) \\
&= 2(2) + t_{\text{RSum}}(n-2) \\
&\vdots \\
&= n(2) + t_{\text{RSum}}(0) \\
&= 2n + 2, \qquad\qquad n \geq 0
\end{aligned}
$$

s/e method

- 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 steps
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			$2n+3$

Example 2

Statement	s/e	frequency $n=0$	frequency $n>0$	total steps $n=0$	total steps $n>0$
1 **Algorithm** RSum(a,n)	0	–	–	0	0
2 {					
3 **if** $(n \leq 0)$ **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	$2+x$

$x = t_{\text{RSum}}(n-1)$

Example 3

Statement	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			$2mn+2m+1$

Asymptotic Notations

- The asymptotic notations are used to find the time complexity of an algorithm.

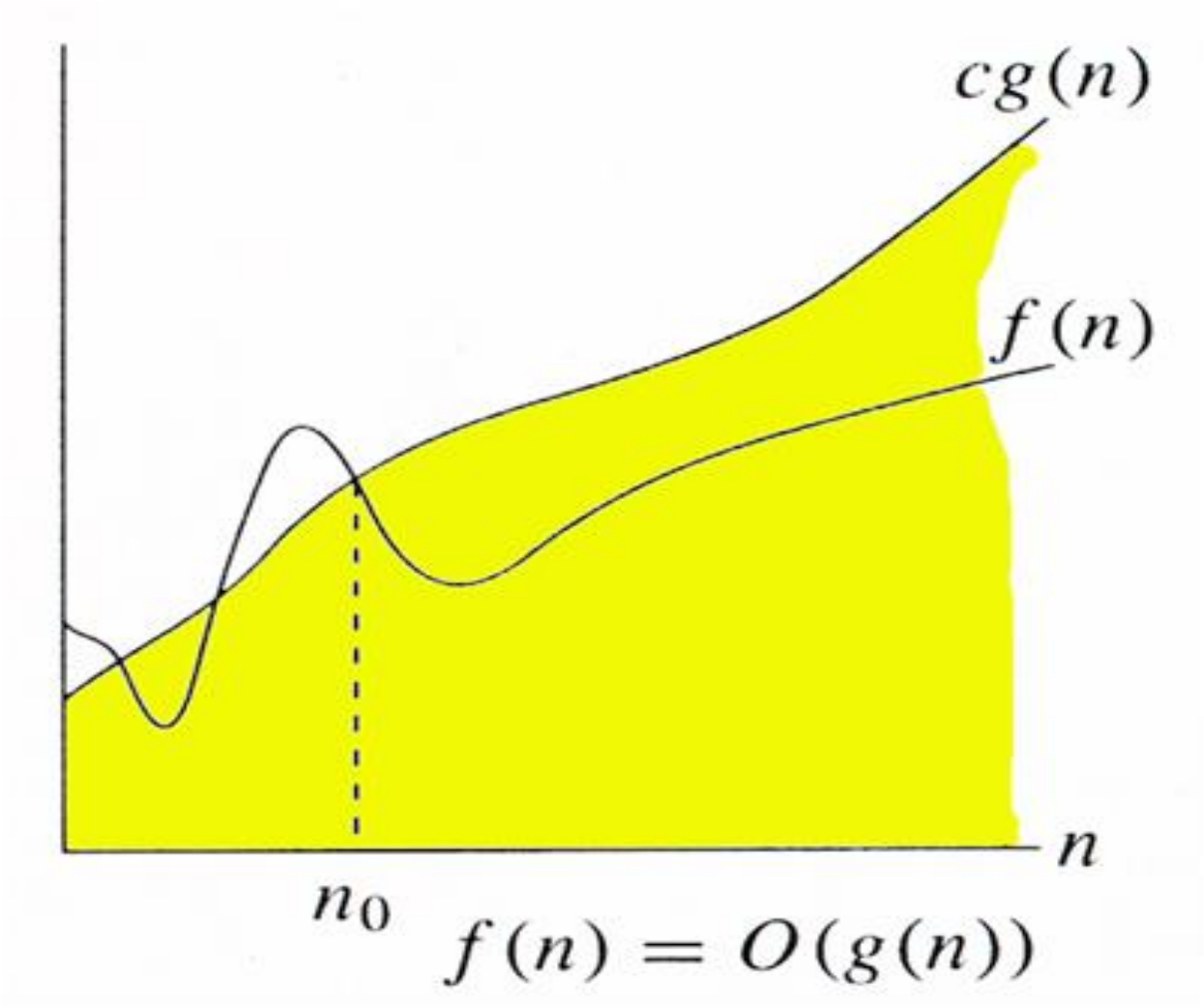
- Asymptotic notations gives fastest possible, slowest possible and average time of the algorithm.
- The basic asymptotic notations are
  - Big-oh(O),
  - Omega(Ω) and
  - theta(Θ).

BIG-OH(O)NOTATION

- Big-O, commonly written as **O**, It provides us with an ***asymptotic upper bound*** for the growth rate of the runtime of an algorithm.

- The function f(n)=O(g(n)) iff there exist positive constants c and n_0 such that $f(n) \leq c*g(n)$ for all n, $n \geq n_0$

BIG-OH(O)NOTATION



Examples

1. The function $3n+2=O(n)$ as $3n+2 \leq 4n$ for all $n \geq 2$.

2. The function $10n^2+4n+2=O(n^2)$ as $10n^2+4n+2 \leq 11n^2$ for all $n \geq 5$.

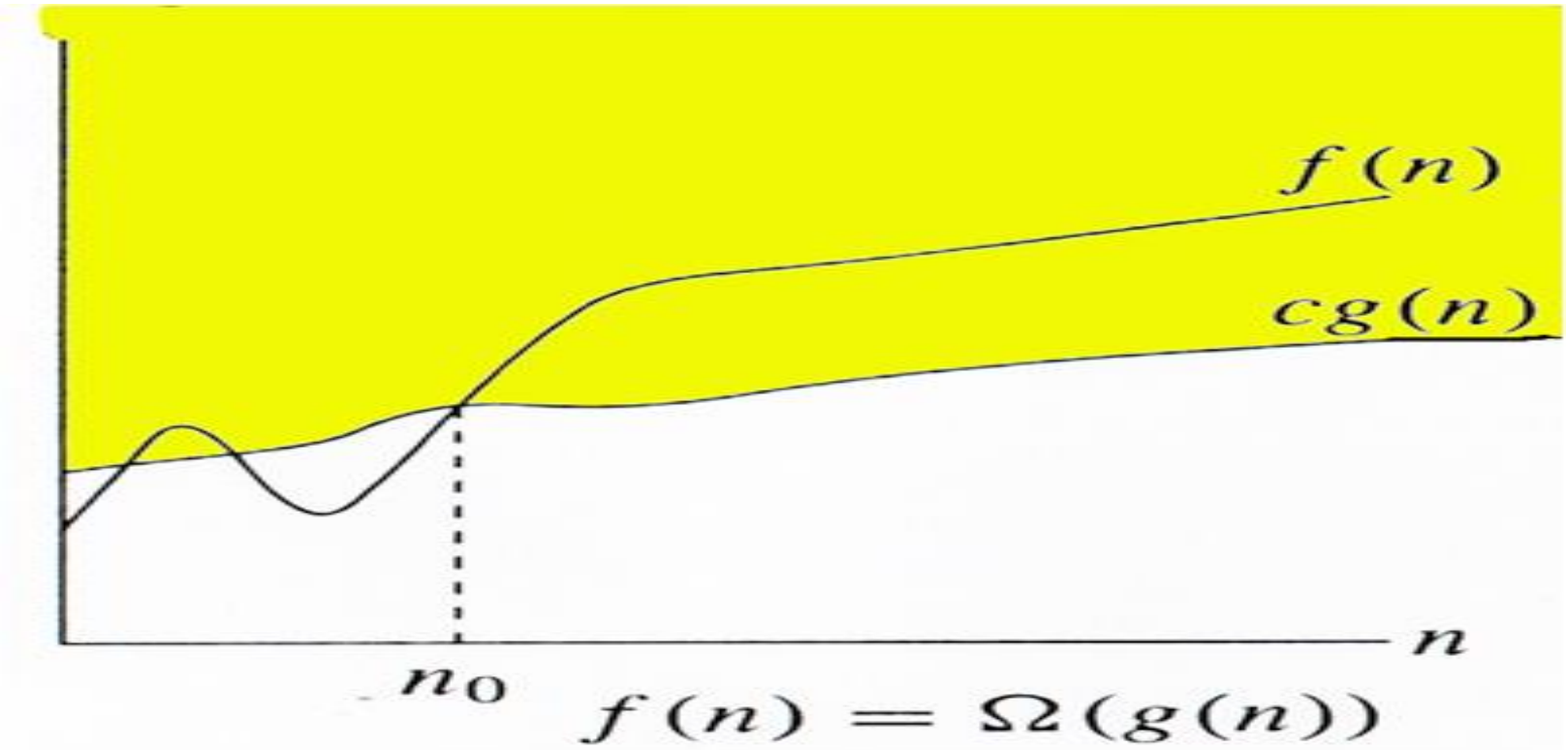
3. The function $100n^3+100n^2-6=O(n^3)$ as $1000n^3+100n^2-6 \leq 101n^2$ for all $n \geq 100$.

OMEGA(Ω)NOTATION

❖Omega, commonly written as **Ω**, It provides us with an ***asymptotic lower bound*** for the growth rate of the runtime of an algorithm.

❖The function f(n)= **Ω**(g(n)) iff there exist positive constants c and n_0 such that f(n) ≥ c*g(n) for all n, n≥ n_0

OMEGA(Ω)NOTATION



Example

consider f(n)=3n+5, g(n)=n

Sol : Let us assume as c=2

$$f(n) >= C*g(n)$$

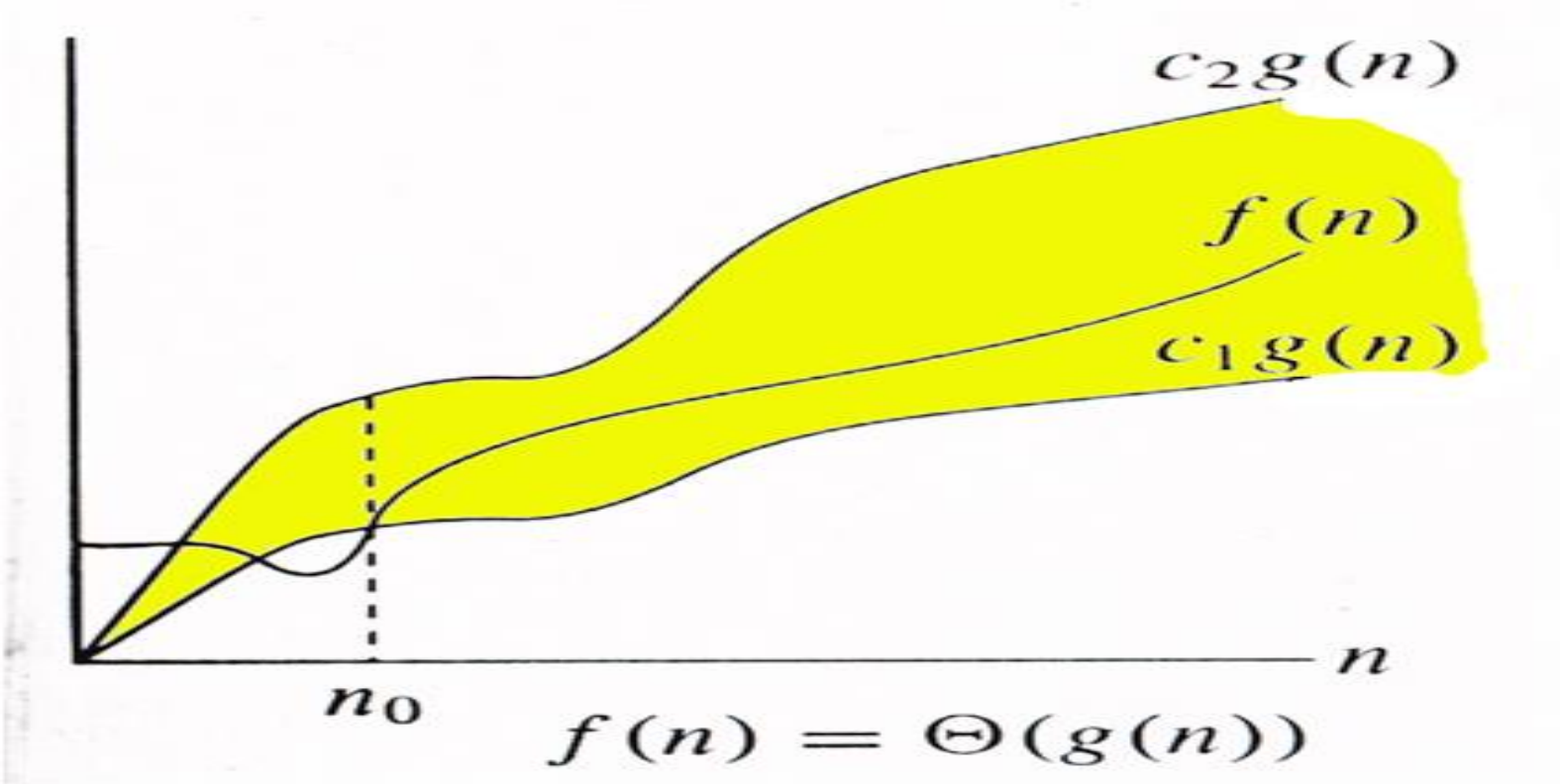
$$3n+5 >= 2n$$

for all n>=1, f(n)=Ω(n) i.e , f(n)=Ω(g(n))

THETA(Θ)NOTATION

- Theta, commonly written as **Θ**, is an Asymptotic Notation to denote the ***asymptotically tight bound*** on the growth rate of runtime of an algorithm.
- The function f(n)= **Θ**(g(n)) iff there exist positive constants c_1, c_2 and n_0 such that $c_1 * g(n) \leq f(n) \leq c_2 * g(n)$ for all n, $n \geq n_0$

THETA(Θ)NOTATION



Example

consider f(n)=3n+5, g(n)=n

Sol :c1*g(n)<=f(n)<=c2*g(n)

let us assuming as c1=2 and c2=4 then

$$2n <= 3n+5 <= 4n$$

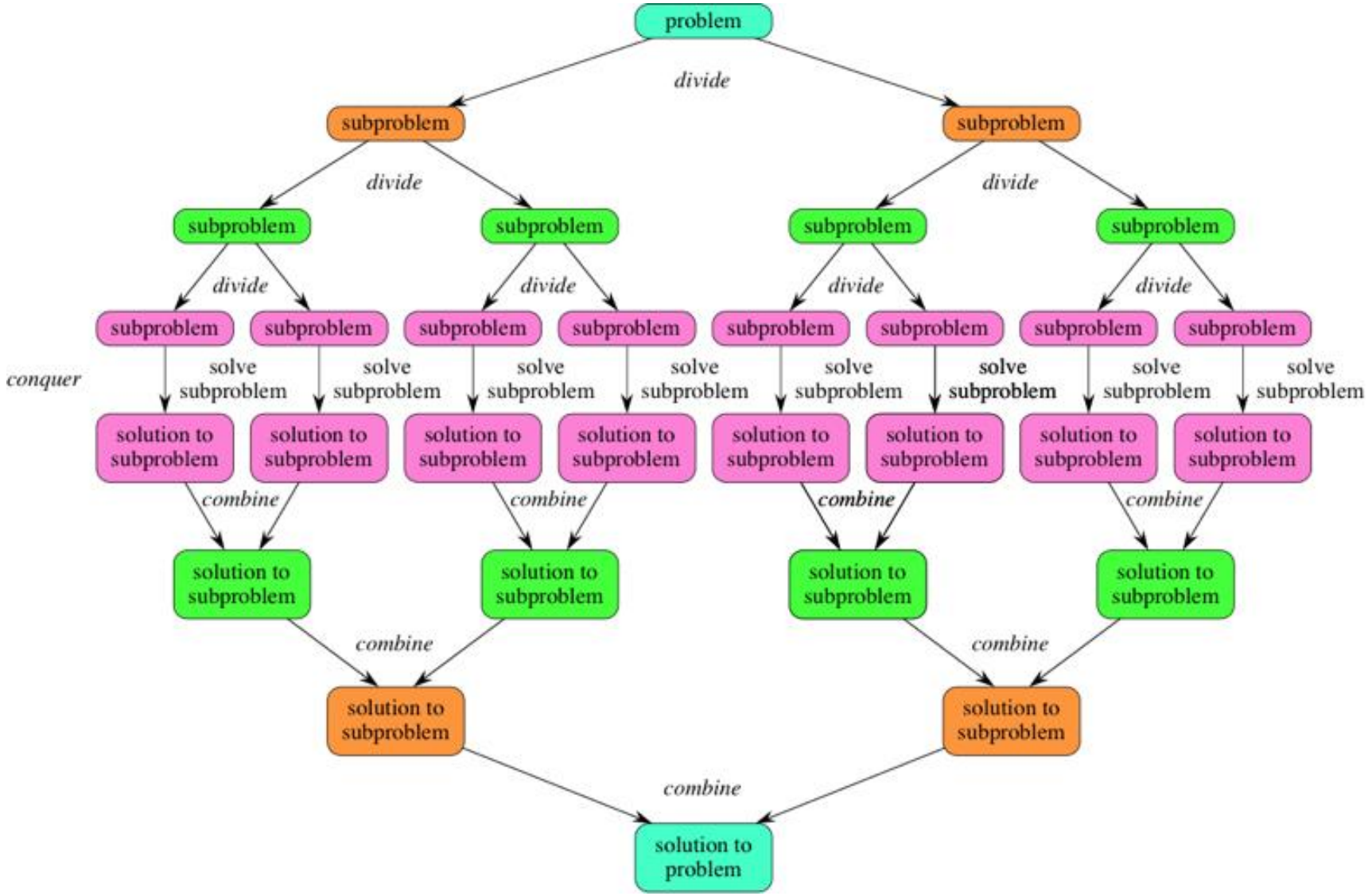
for all :n>=3 $f(n)=\Theta(n)$ $f(n)=\Theta(g(n))$

Performance Measurement

- Space Complexity
- Time Complexity

DIVIDE AND CONQUER

- **Divide and Conquer** - A divide-and-conquer algorithm works by recursively dividing a problem into two or more sub-problems until these become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem.


problem
divide
subproblem
subproblem
divide
divide
subproblem
subproblem
subproblem
subproblem
divide
divide
divide
divide
subproblem
subproblem
subproblem
subproblem
subproblem
subproblem
subproblem
subproblem
conquer
solve subproblem
solve subproblem
solve subproblem
solve subproblem
solve subproblem
solve subproblem
solve subproblem
solve subproblem
solve subproblem
solve subproblem
solution to subproblem
solution to subproblem
solution to subproblem
solution to subproblem
solution to subproblem
solution to subproblem
solution to subproblem
solution to subproblem
combine
combine
combine
combine
solution to subproblem
solution to subproblem
solution to subproblem
solution to subproblem
combine
combine
solution to subproblem
solution to subproblem
combine
solution to problem

Pros and cons of Divide and Conquer Approach

- Divide and conquer approach supports parallelism as sub-problems are independent, can be solved simultaneously.
- In this approach, most of the algorithms follow recursions, very high memory is required for recursion stack

Application of Divide and Conquer Approach

- Finding the maximum and minimum of a sequence of numbers
- Merge sort
- Binary search
- Quick Sort
- Matrix Multiplication (Strassen's algorithm)

```
Algorithm DANDC (P)
{
// if problem P is small, find solution to p and return
   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));
   }
}
```

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

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

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

FINDING THE MAXIMUM AND MINIMUM using DIVIDE AND CONQUER Strategy

- Let P = (n, a [i],……,a [j]) denote an arbitrary instance of the problem.
- Here ‘n’ is the no. of elements in the list (a [i],….,a[j]) and we are interested in finding the maximum and minimum of the list.
- If the list has more than 2 elements, P has to be divided into smaller instances.
- For example, we might divide ‘P’ into the 2 instances, P1=([n/2],a[1],……..a[n/2]) & P2= (n-[n/2], a[[n/2]+1],….., a[n]) After having divided ‘P’ into 2 smaller sub problems, we can solve them by recursively invoking the same divide-and-conquer algorithm.

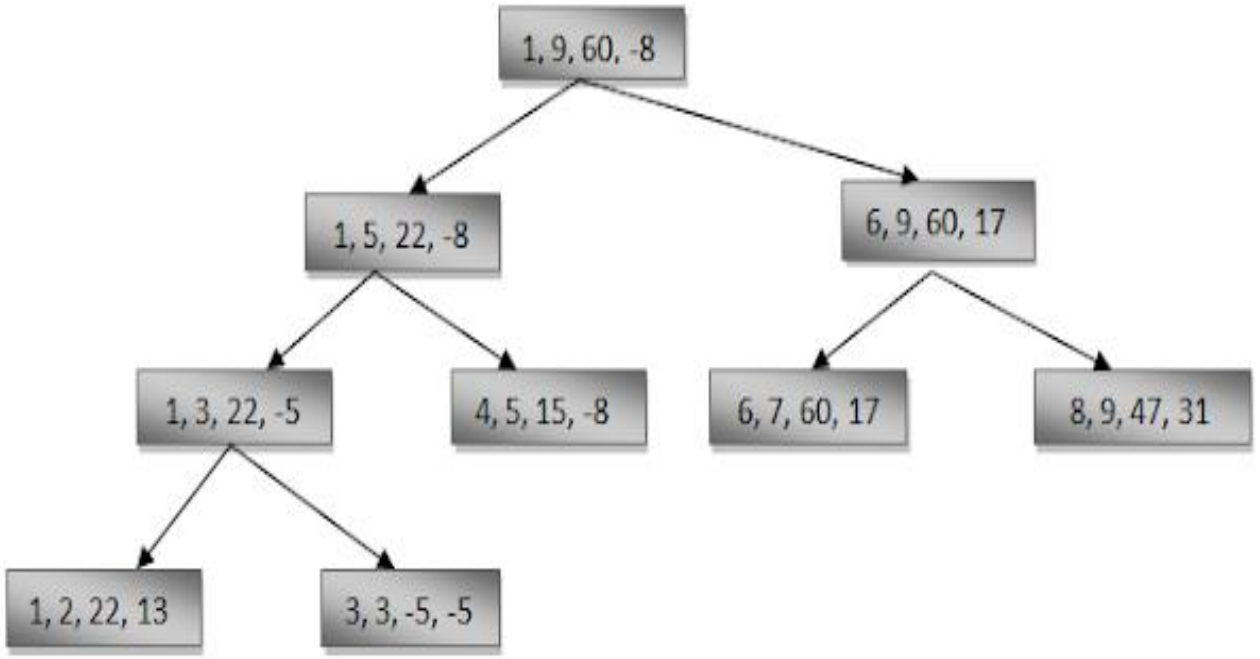
```
MaxMin (i, j, max, min)
 // a [1: n] is a global array, parameters i& j are integers, 1<= i<=j <=n. The effect is
to4.
// Set max & min to the largest & smallest value 5 in a [i: j], respectively.
{
If (i=j) then Max = Min = a[i];
Else if (i=j-1) then
{
if (a[i] < a[j]) then
        {
                Max = a[j];
                Min = a[i];
        }
        Else
        {
                Max = a[i];
                Min = a[j];
        }
}Else
{
Mid = (i + j) / 2;
MaxMin (I, Mid, Max, Min);
MaxMin (Mid +1, j, Max1, Min1);
If (Max < Max1) then Max = Max1;
If (Min > Min1) then Min = Min1;
}}
```

The procedure is initially invoked by the statement, MaxMin (1, n, x, y)

- The procedure is initially invoked by the statement MaxMin(1,n,x,y). for this algorithm each node has four items of information: i, j, max, min. Suppose we simulate MaxMin on the following nine elements:

  a: [1] [2] [3] [4] [5] [6] [7] [8] [9]

  22 13 -5 -8 15 60 17 31 47



- As shown in figure, in this Algorithm, each node has 4 items of information: i, j, max & min.
- In figure, root node contains 1 & 9 as the values of i& j corresponding to the initial call to MaxMin.
- This execution produces 2 new calls to MaxMin, where i& j have the values 1, 5 & 6, 9 respectively & thus split the set into 2 subsets of approximately the same size.
- Maximum depth of recursion is 4.

Binary search

- Binary Search algorithm is used for finding an item from a sorted list,
- Binary Search works by repeatedly dividing input list into two halves, where in next step we can eliminate one half which cannot contain the required key to find. The process repeats until we are left with only one element
- If elements are not sorted, sort the array then apply Binary Search

```
Algorithm BinSrch(a, i, l, x)
// Given an array a[i : l] of elements in nondecreasing
// order, 1 ≤ i ≤ l, determine whether x is present, and
// if so, return j such that x = a[j]; else return 0.
{
    if (l = i) then  // If Small(P)
    {
        if (x = a[i]) then return i;
        else return 0;
    }
    else
    { // Reduce P into a smaller subproblem.
        mid := ⌊(i + l)/2⌋;
        if (x = a[mid]) then return mid;
        else if (x < a[mid]) then
                return BinSrch(a, i, mid − 1, x);
            else return BinSrch(a, mid + 1, l, x);
    }
}
```

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

$$-15, -6, 0, 7, 9, 23, 54, 82, 101, 112, 125, 131, 142, 151$$

$x = 131$	*low*	*high*	*mid*
	1	14	7
	8	14	11
	12	14	13
	14	14	14
			found

$x = -14$	*low*	*high*	*mid*
	1	14	7
	1	6	3
	1	2	1
	2	2	2
	2	1	not found

$x = 9$	*low*	*high*	*mid*
	1	14	7
	1	6	3
	4	6	5
			found

- **Calculating Time complexity:**
  - Let say the iteration in Binary Search terminates after **k** iterations.
  - At each iteration, the array is divided by half. So let's say the length of array at any iteration is **n**
  - At **Iteration 1**,Length of array = **n**
  - At **Iteration 2**,Length of array = $n/2$
  - At **Iteration 3**,Length of array = $(n/2)/2 = n/2^2$
  - Therefore, after **Iteration k**, Length of array = $n/2^k$
  - Also, we know that after k divisions, the **length of array becomes 1**
  - Therefore Length of array = $n/2^k = 1$ => $n = 2^k$
  - Applying log function on both sides:=> $\log_2(n) = \log_2(2^k)$ => $\log_2(n) = k \log_2(2)$
  - As $(\log_a(a) = 1)$ Therefore,=> $k = \log_2(n)$
- **Hence, the time complexity of Binary Search is**

$$\log_2(n)$$

Time complexity of binary search

successful searches			unsuccessful searches
$\Theta(1)$,	$\Theta(\log n)$,	$\Theta(\log n)$	$\Theta(\log n)$
best,	average,	worst	best, average, worst

Merge sort

- A sorting algorithm which has the nice property that in the worst case its complexity is *O(n log2 n). This algorithm is called merge sort.*
- *We shall assume throughout* that the elements are to be sorted in non decreasing order.
- *Thus we* have another ideal **example of the divide-and-conquer** strategy where the splitting is into two equal size sets and the combining operation is the merging of two sorted sets into one.
- Procedure MERGESORT describes this process very succinctly using recursion and a sub procedure MERGE which merges together two sorted Sets.

- Given a sequence of *n. elements (also called keys) A(l), ... , A(n) the general idea* is to imagine them split into two sets .
- *Each set is individually sorted and the resulting sequences* are merged to produce a single sorted sequence of *n elements.*

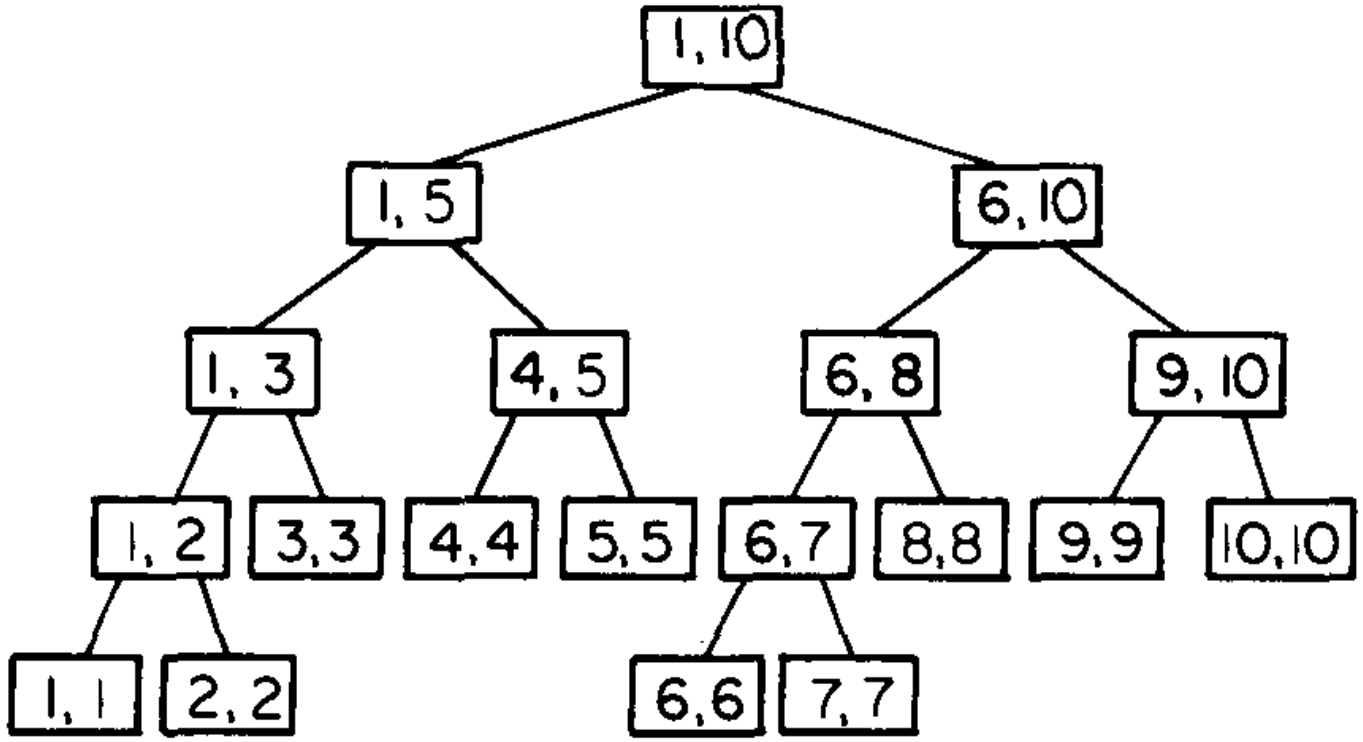
Merge sort algorithm

```
Algorithm MergeSort(low, high)
// a[low : high] is a global array to be sorted.
// Small(P) is true if there is only one element
// to sort. In this case the list is already sorted.
{
    if (low < high) then  // If there are more than one element
    {
        // Divide P into subproblems.
            // Find where to split the set.
                mid := ⌊(low + high)/2⌋;
        // Solve the subproblems.
            MergeSort(low, mid);
            MergeSort(mid + 1, high);
        // Combine the solutions.
            Merge(low, mid, high);
    }
}
```

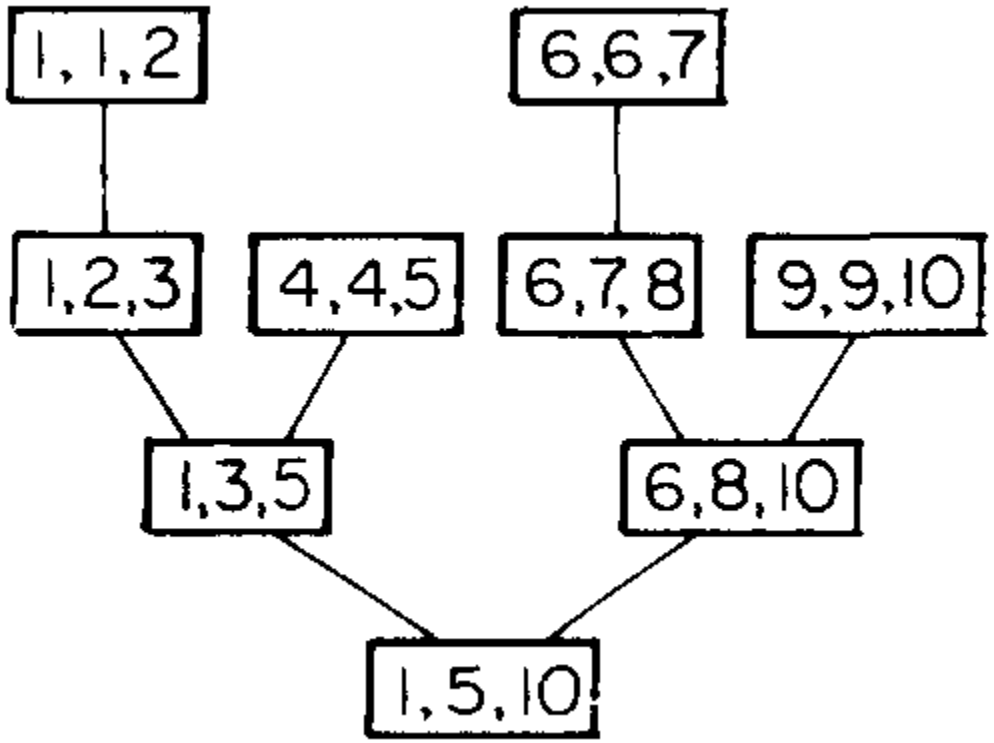
```
Algorithm Merge(low, mid, high)
// a[low : high] is a global array containing two sorted
// subsets in a[low : mid] and in a[mid + 1 : high]. The goal
// is to merge these two sets into a single set residing
// in a[low : high]. b[ ] is an auxiliary global array.
{
    h := low; i := low; j := mid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

Example and tree of calls to merge sort

Consider the array of ten elements *A = (310, 285, 179, 652, 351, 423,*861, 254, 450, 520).



Tree of calls to merge



Time complexity

If the time for the merging operation is proportional to n, then the computing time for merge sort is described by the recurrence relation

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

- When *n is a power of 2, $n = 2^k$, we can solve this equation by successive* substitutions, namely
- *T(n) = 2(2T(n/4) + n/2) + n*

  = 4T(n/4) + n

  = 4(2T(n/8) + *n/4) + 2n*

  …

  = 2^kT(l) + *kn*

  = n *+n log n*
- It is easy to see that if $2^k < n <= 2^{k+1}$ *then*

  T(n) <= T (2^{k+1})*.*
- Therefore $T(n) = O(n \log_2 n)$.

Quick sort

- In merge sort , the file A(l:n) was divided at its midpoint into sub files which were independently sorted and later merged.
- In quick sort, the division into two sub files is made such that the sorted sub files do not need to be later merged.
- This is accomplished by rearranging the elements in A(l :n) such that *A(i)<= A(j) for all i between* 1 and *m and all j between m + 1 and n for some m, 1<=m<=n.*
- *Thus,* the elements in A(l:m) and A(m + l:n) may be independently sorted.

Quick sort algorithm

```
Algorithm QuickSort(p, q)
// Sorts the elements a[p], ..., a[q] which reside in the global
// array a[1 : n] into ascending order; a[n + 1] is considered to
// be defined and must be ≥ all the elements in a[1 : n].
{
    if (p < q) then  // If there are more than one element
    {
        // divide P into two subproblems.
            j := Partition(a, p, q + 1);
                // j is the position of the partitioning element.
        // Solve the subproblems.
            QuickSort(p, j − 1);
            QuickSort(j + 1, q);
        // There is no need for combining solutions.
    }
}
```

Partition algorithm

```
Algorithm Partition(a, m, p)
// Within a[m], a[m+1], ..., a[p-1] the elements are
// rearranged in such a manner that if initially t = a[m],
// then after completion a[q] = t for some q between m
// and p-1, a[k] ≤ t for m ≤ k < q, and a[k] ≥ t
// for q < k < p. q is returned. Set a[p] = ∞.
{
    v := a[m]; i := m; j := p;
    repeat
    {
        repeat
            i := i + 1;
        until (a[i] ≥ v);

        repeat
            j := j - 1;
        until (a[j] ≤ v);

        if (i < j) then Interchange(a, i, j);

    } until (i ≥ j);

    a[m] := a[j]; a[j] := v; return j;
}
```

Interchange algorithm

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

Time complexity of quick sort

- The worst case complexity is $O(n^2)$
- The average case and best case complexity is O(nlogn).

UNIT-II

GREEDY METHOD

GREEDY METHOD

- All of these problems have n inputs and require us to obtain a subset that satisfies some constraints.
- Any subset that satisfies those constraints is called a feasible solution.
- We need to find a feasible solution that either maximizes or minimizes a given objective function.
- A feasible solution that does this is called an optimal solution.
- In Greedy method at each stage, a decision is made regarding whether a particular input is in an optimal solution.

```
Algorithm Greedy(a, n)
// a[1 : n] contains the n inputs.
{
    solution := ∅; // Initialize the solution.
    for i := 1 to n do
    {
        x := Select(a);
        if Feasible(solution, x) then
            solution := Union(solution, x);
    }
    return solution;
}
```

Knapsack Problem

- We are given n objects and a knapsack(bag)
- 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 obtain a filling of 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.

Knapsack Problem

- The problem can be stated as

$$\text{maximize} \sum_{1 \le i \le n} p_i x_i \quad \longrightarrow \quad 1$$

$$\text{subject to} \sum_{1 \le i \le n} w_i x_i \le m \quad \longrightarrow \quad 2$$

$$\text{and } 0 \le x_i \le 1, \quad 1 \le i \le n \quad \longrightarrow \quad 3$$

The profits and weights are positive numbers.

A feasible solution(or filling) is any set(x_i,..., x_n) satisfying equation 2 and 3 above. An optimal solution is a feasible solution for which equation 1 is maximized.

```
Algorithm GreedyKnapsack(m, n)
// p[1 : n] and w[1 : n] contain the profits and weights respectively
// of the n objects ordered such 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];
}
```

Job Sequencing with Deadlines

- Given a set of n jobs. Associated with job i is an integer deadline d_i >0 and a profit p_i >0.
- For any job i the profit p_i is earned iff the job is completed by its deadline.
- To complete a job, one has to process the job on a machine for one unit of time. Only one machine is available for processing jobs.
- A feasible solution for this problem is a subset J of jobs such that each job in this subset can be completed by its deadline.
- The value of a feasible solution J is the sum of the profits of the jobs in J, or $\sum_{i \in J} pi$
- An optimal solution is a feasible solution with maximum value.

Job Sequencing with Deadlines

- The value of a feasible solution J is the sum of the profits of the jobs in J, or $\sum_{i \in J} pi$
- An optimal solution is a feasible solution with maximum value. Here again, since the problem involves the identification of a subset, it fits the subset paradigm.

Job Sequencing with Deadlines

- The greedy algorithm described below always gives an optimal solution to the job sequencing problem-
- Step-01:
- Sort all the given jobs in decreasing order of their profit.
- Step-02:
  - Check the value of maximum deadline.
  - Draw a Gantt chart where maximum time on Gantt chart is the value of maximum deadline.
- Step-03:
  - Pick up the jobs one by one.
  - Put the job on Gantt chart as far as possible from 0 ensuring that the job gets completed before its deadline.
-

Job Sequencing with Deadlines Algorithm

```
Algorithm JobSequencing()
{
    // jobs are arranged according to their profits from highest to lowest
    // dmax – maximum job deadline
    for i = 1 to n do
        set k = min (dmax, deadline(i))
        while k >= 1 do
                if timeslot[k] is empty then
                        timeslot[k] = job[i]
                        break
                end if
                set k = k-1
        end while
    end for
}
```

Jobs	J1	J2	J3	J4	J5	J6
Deadlines	5	3	3	2	4	2
Profits	200	180	190	300	120	100

Jobs	**J4**	**J1**	**J3**	**J2**	**J5**	**J6**
Deadlines	2	5	3	3	4	2
Profits	300	200	190	180	120	100

Job Sequencing with Deadlines

- Step-02:
- Value of maximum deadline = 5.
- So, draw a Gantt chart with maximum time on Gantt chart = 5 units

Gantt Chart

Now,

- We take each job one by one in the order they appear in Step-01.
- We place the job on Gantt chart as far as possible from 0.

Job Sequencing with Deadlines

- Step-03:

➢ We take job J4.

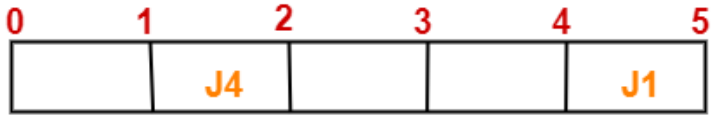
➢Since its deadline is 2, so we place it in the first empty cell before deadline 2

Job Sequencing with Deadlines

- Step-04:

➢ We take job J1.

➢Since its deadline is 5, so we place it in the first empty cell before deadline 5



Job Sequencing with Deadlines

- Step-05:

➢ We take job J3.

➢Since its deadline is 3, so we place it in the first empty cell before deadline 3

Job Sequencing with Deadlines

- Step-06:

➢ We take job J2.

➢Since its deadline is 3, so we place it in the first empty cell before deadline 3.

➢Since the second and third cells are already filled, so we place job J2 in the first cell

Job Sequencing with Deadlines

- Step-07:

➢ We take job J5.

➢Since its deadline is 4, so we place it in the first empty cell before deadline 4.

- Now,

➢The only job left is job J6 whose deadline is 2.

➢All the slots before deadline 2 are already occupied.

➢Thus, job J6 can not be completed.

Job Sequencing with Deadlines

- Write the optimal schedule that gives maximum profit.
- The optimal schedule is-
  - J2 , J4 , J3 , J5 , J1
- This is the required order in which the jobs must be completed in order to obtain the maximum profit.
- Are all the jobs completed in the optimal schedule?

  All the jobs are not completed in optimal schedule.

  This is because job J6 could not be completed within its deadline.
- What is the maximum earned profit?

Maximum earned profit
= Sum of profit of all the jobs in optimal schedule
= Profit of job J2 + Profit of job J4 + Profit of job J3 + Profit of job J5 + Profit of job J1
= 180 + 300 + 190 + 120 + 200
= 990 units

Minimum-cost Spanning Trees

- A **spanning tree** is a subset of an undirected Graph that has all the vertices connected by minimum number of edges.
- For a graph, there may exist more than one spanning tree.
- Properties
  - A spanning tree does not have any cycle.
  - Any vertex can be reached from any other vertex.
- Let G = (V, E) be an undirected connected graph. A subgraph t = (V, E')of G is a spanning tree of G iff t is a tree.
- If there are **n** number of vertices, the spanning tree should have ***n* - 1** number of edges.
- A Minimum Spanning Tree (MST) is a subset of edges that connects all the vertices together with the minimum possible total edge weight. To derive an MST, Prim's algorithm or Kruskal's algorithm can be used.

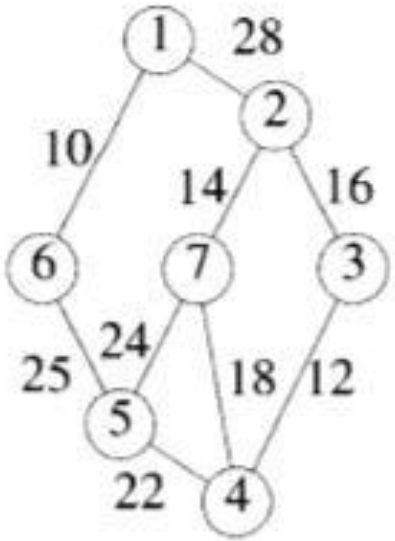
Prim’s algorithm

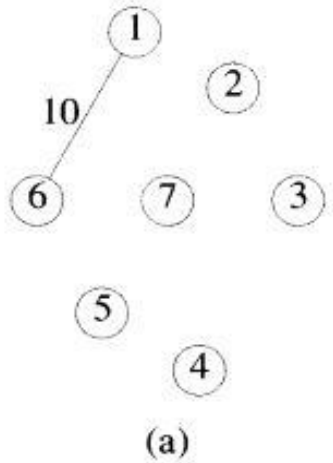
Prim’s algorithm

- A greedy method to construct minimum-cost spanning tree, builds the tree edge by edge. The next edge to include is chosen according to some optimization criterion.
- The simplest such criterion is to choose an edge that results in a minimum increase in the sum of the costs of the edges so far included.
- The next edge(u,v) to be included in A is a minimum-cost edge not in A with the property that A U {(u,v)} is also a tree.

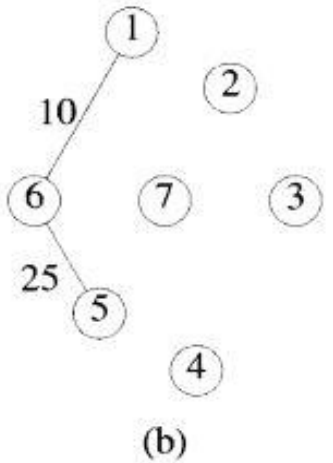
Prim's algorithm

- working principle of Prim's Algorithm

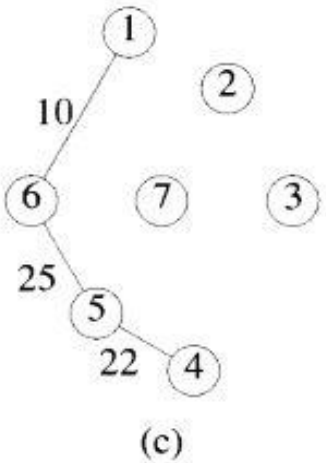




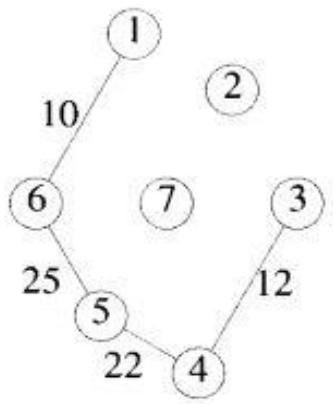
(a)



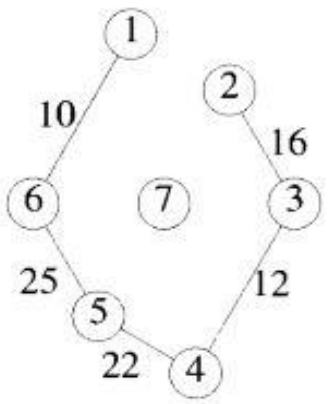
(b)



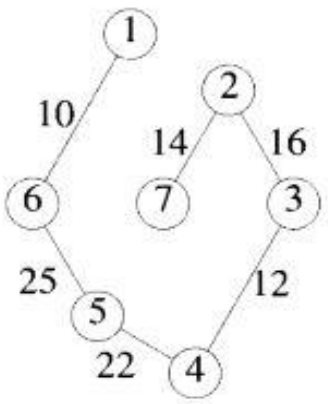
(c)



(d)



(e)



(f)

```
Algorithm AdjacencyMatrix()
{
        for i=1 to n do
                for j=1 to n do
                        Read cost[i][j]
                        if(cost[i][j]==0) then
                                cost[i][j]=999;
                        end if
                end for
        end for
}
```

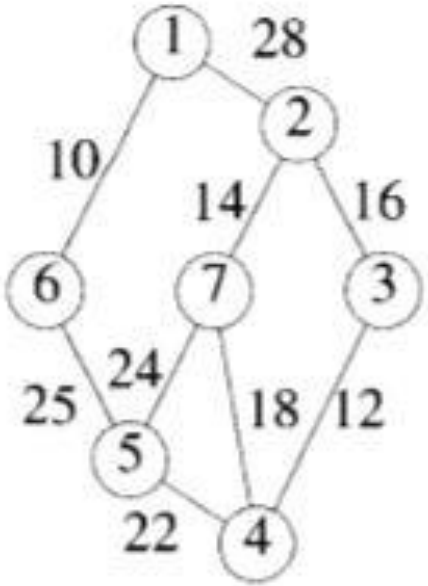
```
// cost matrix is calculated using the above algorithm
Algorithm Prims()
{
        visited[1]=1
        while(ne < n) do
                min = 999
                for i=1 to n
                        if(visited[i]!=0) then
                                for j=1 to n do
                                        if(cost[i][j]< min) then
                                                min=cost[i][j]
                                                u=i
                                                v=j
                                        end if
                                end for
                        end if
                end for
                if(visited[v]==0) then
                        ne++
                        mincost+=min
                        visited[j]=1
                end if
                cost[u][v]=cost[v][u]=999
        end while
}
```

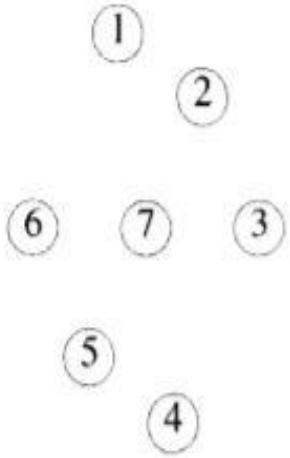
Kruskal's algorithm

- Sort all the edges in non-decreasing order of their weight.
- Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If cycle is not formed, include this edge. Else, discard it.
- Repeat step#2 until there are (V-1) edges in the spanning tree.

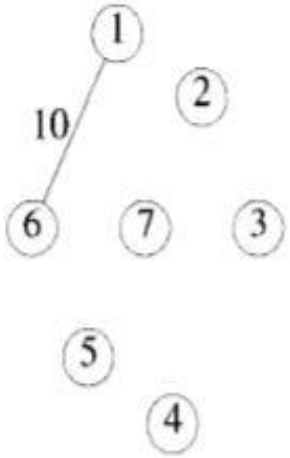
Kruskal's algorithm

- working principle of Kruskal's Algorithm

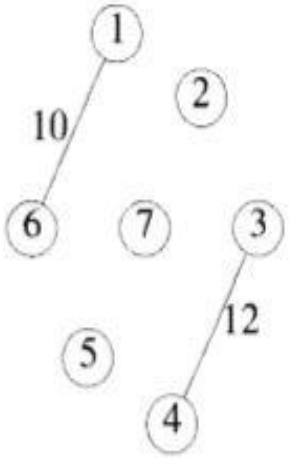




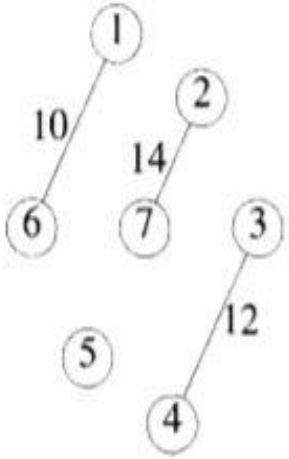
(a)



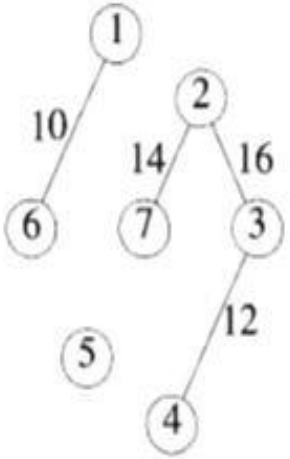
(b)



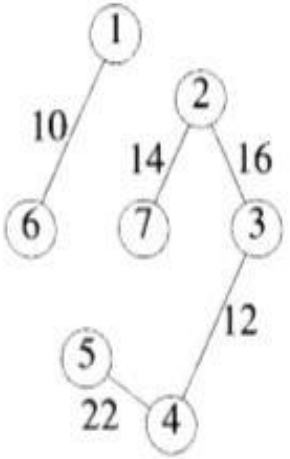
(c)



(d)



(e)



(f)

```
Algorithm Kruskals()
{
 while(ne < n) do
            for i=1 to n
                        for j=1 to n
                                    if(cost[i][j] < min) then
                                                min=cost[i][j];
                                                a=u=i;
                                                b=v=j;
                                    end if
                         end for
            end for
            u=find(u);
            v=find(v);
            if(uni(u,v)) then
                        ne++
                        mincost = mincost+min;
            end if
            cost[a][b]=cost[b][a]=999;
 end while
}
```

```
Algorithm find(int i)
{
        while(parent[i]) do
                i=parent[i];
        return i;
}
```

```
Algorithm uni(int i,int j)
{
        if(i!=j) then
                parent[j]=i;
                return 1;
        return 0;
}
```

Differences between Prim's and Kruskal's

PRIM'S ALGORITHM	KRUSKAL'S ALGORITHM
It starts to build the Minimum Spanning Tree from any vertex in the graph.	It starts to build the Minimum Spanning Tree from the vertex carrying minimum weight in the graph.
It traverses one node more than one time to get the minimum distance.	It traverses one node only once.
Prim's algorithm gives connected component as well as it works only on connected graph.	Kruskal's algorithm can generate forest(disconnected components) at any instant as well as it can work on disconnected components
Prim's algorithm runs faster in dense graphs.	Kruskal's algorithm runs faster in sparse graphs.

Optimal Merge Patterns

- Given n number of sorted files, the task is to find the minimum computations done to reach Optimal Merge Pattern.
- When two or more sorted files are to be merged all together to form a single file, the minimum computations done to reach this file are known as **Optimal Merge Pattern**.
- If more than 2 files need to be merged then it can be done in pairs. For example, if need to merge 4 files A, B, C, D. First Merge A with B to get X1, merge X1 with C to get X2, merge X2 with D to get X3 as the output file.
- If we have two files of sizes m and n, the total computation time will be m+n. Here, we use greedy strategy by merging two smallest size files among all the files present.

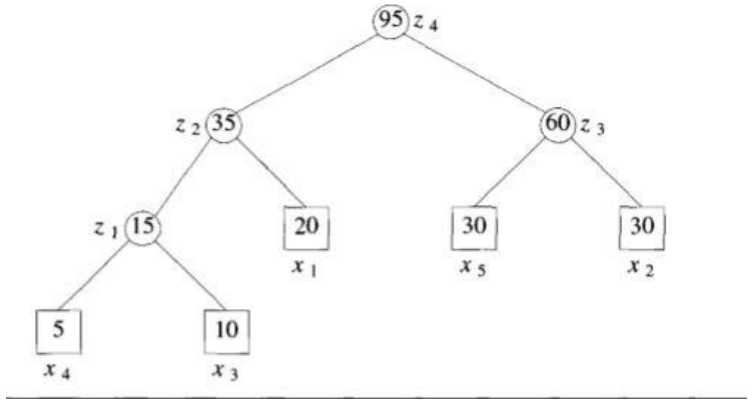
2 Way Merge Pattern

- The two-way merge pattern scan be represented by binary merge trees.

- The leaf nodes are drawn as squares and represent the given files. These nodes are called external node.

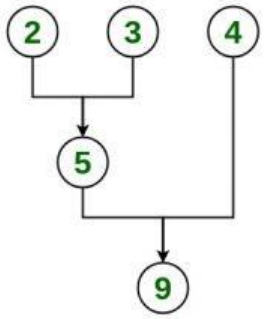
Example

- if we have five files with sizes(20,30,10,5, 30),our greedy rule would generate the following:



Example:

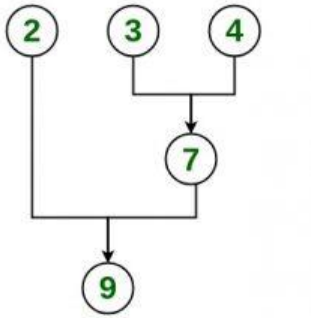
- **Examples:**
  Given 3 files with size 2, 3, 4 units . Find optimal way to combine these files
- ***Input:*** *n = 3, size = {2, 3, 4}*
  Output: *14*
  Explanation: *There are different ways to combine these files:*
  Method 1: *Optimal method*



Cost = 5 + 9 = 14

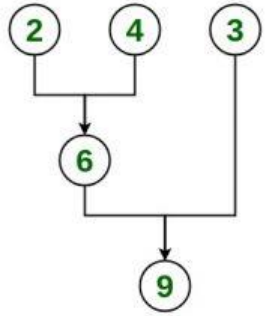
Example:

- ***Method 2:***



Cost = 7 + 9 = 16

Method 3:



Cost = 6 + 9 = 15

```
treenode = record {
    treenode * lchild; treenode * rchild;
    integer weight;
};

Algorithm Tree(n)
// list is a global list of n single node
// binary trees as described above.
{
    for i := 1 to n - 1 do
    {
        pt := new treenode; // Get a new tree node.
        (pt → lchild) := Least(list); // Merge two trees with
        (pt → rchild) := Least(list); // smallest lengths.
        (pt → weight) := ((pt → lchild) → weight)
                +((pt → rchild) → weight);
        Insert(list, pt);
    }
    return Least(list); // Tree left in list is the merge tree.
}
```

Single Source Shortest Paths

- Graphs can be used to represent the highway structure of a state or country with vertices representing cities and edges representing sections of highway.
- The edges can then be assigned weights which may be either the distance between the two cities connected by the edge or the average time to drive along that section of highway.
- The **single-source shortest path** problem, in which we have to find **shortest paths** from a **source** vertex v to all other vertices in the graph.

Single Source Shortest Paths

- A motorist wishing to drive from city A to B would be interested in answers to the following questions:

➢ Is there a path from A to B?

➢ If there is more than one path from A to B, which is the shortest path?

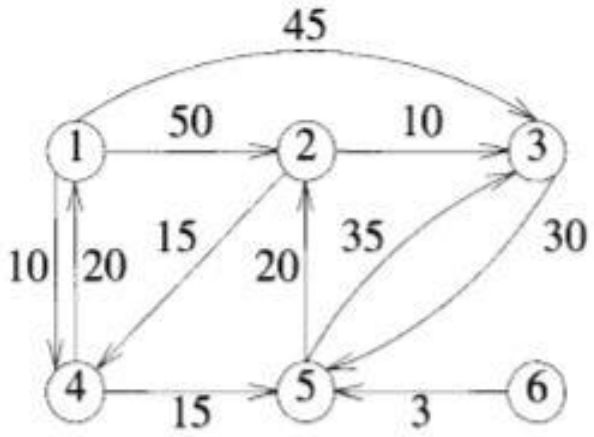
The length of a path is now defined to be the sum of the weights of the edges on that path.

The starting vertex of the path is referred to as the **source**, and the last vertex the **destination**.

Single Source Shortest Paths

- In the problem we consider, we are given a directed graph G = (V,E),a weighting function cost for the edges of G,and a source vertex v_0.
- The problem is to determine the shortest paths from $v_0$to all the remaining vertices of G.
- It is assumed that all the weights are positive.
- The shortest path between v_0 and some other node v is an ordering among a subset of the edges.
- Hence this problem fits the ordering paradigm.

Example



(a) Graph

	Path	*Length*
1)	1, 4	10
2)	1, 4, 5	25
3)	1, 4, 5, 2	45
4)	1, 3	45

(b) Shortest paths from 1

Algorithm(Dijkstra's algorithm)

```
Algorithm ShortestPaths(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 a 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
    { // Initialize S.
        S[i] := false; 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 in S.
        for (each w adjacent to u with S[w] = false) do
            // Update distances.
            if (dist[w] > dist[u] + cost[u, w])) then
                dist[w] := dist[u] + cost[u, w];
    }
}
```

Difference between Bellman-ford and Dijkstra Algorithms

- The only **difference between** the two is that **Bellman-Ford** is also capable of handling negative weights .
- **Dijkstra** Algorithm can only handle positives.

UNIT III

Dynamic Programming

- **Dynamic Programming** is mainly an optimization over plain recursion.
- Wherever we see a recursive solution that has repeated calls for same inputs, we can optimize it using Dynamic Programming.
- The idea is to simply store the results of subproblems, so that we do not have to re-compute them when needed later.
- Dynamic programming is both a **mathematical optimization** method and a **computer programming** method.

- Like divide-and-conquer method, **Dynamic Programming** solves problems by combining the solutions of subproblems. ...
- Moreover, **Dynamic Programming** algorithm solves each sub-problem just once and then saves its answer in a table, thereby avoiding the work of re-computing the answer every time.
- 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.

- All Pairs Shortest Paths
- Single Source Shortest Paths General Weights
- Optimal Binary Search Tree
- String Edition
- 0/1 Knapsack Problem
- Reliability Design

- The **all pair shortest path** algorithm is also known as Floyd-Warshall algorithm is used to find **all pair shortest path** problem from a given weighted graph.
- As a result of this algorithm, it will generate a matrix, which will represent the minimum distance from any node to **all** other nodes in the graph.

- Let G = (V, E) be a directed graph with n vertices.
- The all-pairs shortest-path problem is to determine a matrix A such that A(i,j)is the length of a shortest path from i toj.
- A(i,j)={ 0 if i=j

  the weight of the directed edge ⟨i,j⟩ if i≠j and

  ⟨i,j⟩∈E ∞ if i≠j and ⟨i,j⟩∉E

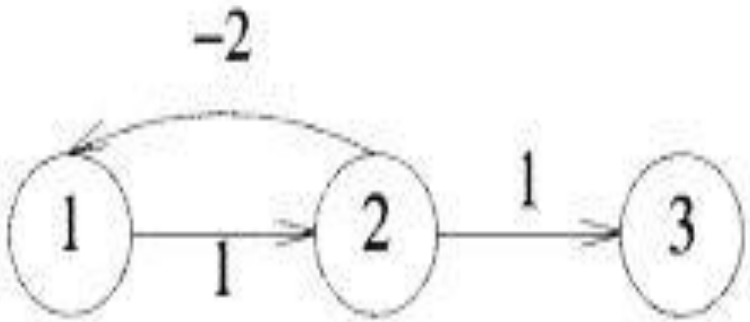
  A(i,j)= min { min {A^k(i,k) +A^k-1(k,j)},cost(i, j)}

  i<k<n

Algorithm for All pairs shortest paths

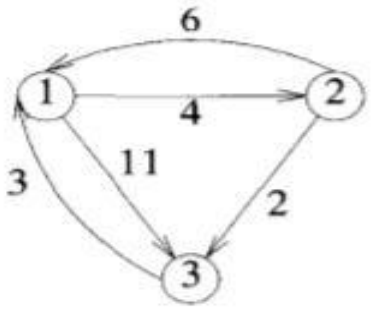
```
Algorithm AllPaths(cost, A, n)
// cost[1 : n, 1 : n] is the cost adjacency matrix of a graph with
// n vertices; A[i, j] is the cost of a shortest path from vertex
// i to vertex j. cost[i, i] = 0.0, for 1 ≤ i ≤ n.
{
     for i := 1 to n do
          for j := 1 to n do
               A[i, j] := cost[i, j]; // Copy cost into A.
     for k := 1 to n do
          for i := 1 to n do
               for j := 1 to n do
                    A[i, j] := min(A[i, j], A[i, k] + A[k, j]);
}
```

Example



$$\begin{bmatrix} 0 & 1 & \infty \\ -2 & 0 & 1 \\ \infty & \infty & 0 \end{bmatrix}$$

Example:



(a) Example digraph

A^0	1	2	3
1	0	4	11
2	6	0	2
3	3	∞	0

(b) A^0

A^1	1	2	3
1	0	4	11
2	6	0	2
3	3	7	0

(c) A^1

A^2	1	2	3
1	0	4	6
2	6	0	2
3	3	7	0

(d) A^2

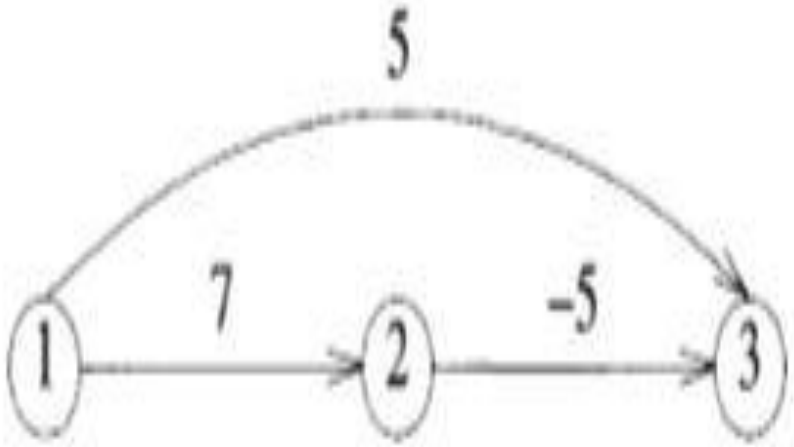
A^3	1	2	3
1	0	4	6
2	5	0	2
3	3	7	0

(e) A^3

Single Source Shortest Paths General Weights

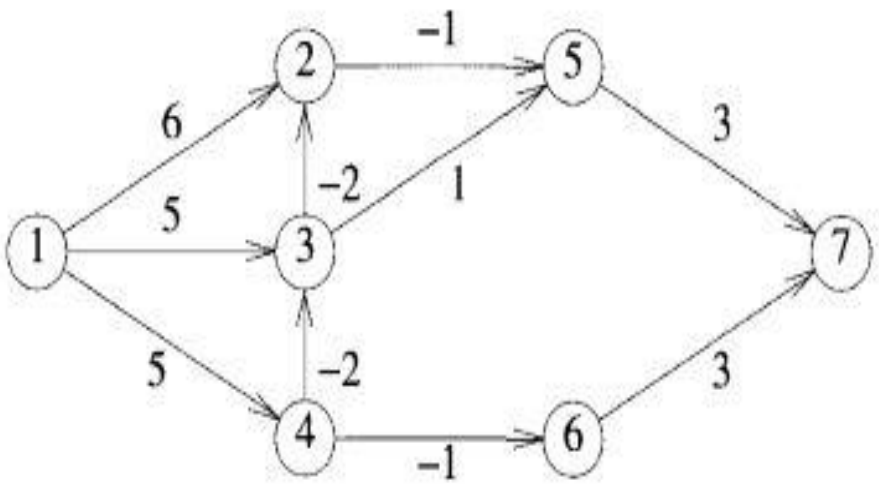
- The idea is to use **Bellman–Ford** algorithm to compute the shortest paths from a single source vertex to all of the other vertices in given weighted digraph.
- **Bellman–Ford** algorithm is slower than Dijkstra's Algorithm but it is capable of handling negative weights edges in the **graph** unlike Dijkstra's.

Single Source Shortest Paths(General Weights)



- When negative edge lengths are permitted ,we require that the graph have no cycles of negative length. This is necessary to ensure that shortest paths consist of a finite number of edge.

Example



(a) A directed graph

	$dist^k[1..7]$						
k	1	2	3	4	5	6	7
1	0	6	5	5	∞	∞	∞
2	0	3	3	5	5	4	∞
3	0	1	3	5	2	4	7
4	0	1	3	5	0	4	5
5	0	1	3	5	0	4	3
6	0	1	3	5	0	4	3

(b) $dist^k$

Algorithm for Single source shortest path using Bellman Ford algorithm

```
Algorithm BellmanFord(v, cost, dist, n)
// Single-source/all-destinations shortest
// paths with negative edge costs
{
     for i := 1 to n do // Initialize dist.
          dist[i] := cost[v, i];
     for k := 2 to n − 1 do
          for each u such that u ≠ v and u has
                    at least one incoming edge do
               for each ⟨i, u⟩ in the graph do
                    if dist[u] > dist[i] + cost[i, u] then
                         dist[u] := dist[i] + cost[i, u];
}
```