UNIT - II

DIVIDE AND CONQUER:

General method:

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

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

- If the sub problems are still relatively large, then the divide-and-conquer strategy can possibly be reapplied.

- Often the sub problems resulting from a divide-and-conquer design are of the same type as the original problem.

- For those cases the re application of the divide-and-conquer principle is naturally expressed by a recursive algorithm.

- D And C(Algorithm) is initially invoked as D and C(P), where 'p' is the problem to be solved.

- Small(P) is a Boolean-valued function that determines whether the i/p size is small enough that the answer can be computed without splitting.

- If this so, the function 'S' is invoked.

- Otherwise, the problem P is divided into smaller sub problems.

- These sub problems P1, P2 …Pk are solved by recursive application of D And C.

- Combine is a function that determines the solution to p using the solutions to the 'k' sub problems.

- If the size of 'p' is n and the sizes of the 'k' sub problems are n1, n2 ….nk, respectively, then the computing time of D And C is described by the recurrence relation.

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

Where T(n) → is the time for D And C on any I/p of size 'n'.
g(n) → is the time of compute the answer directly for small I/ps.
f(n) → is the time for dividing P & combining the solution to sub problems.

```
1. Algorithm D And C(P)
2. {
3. if small(P) then return S(P);
4. else
5. {
6. divide P into smaller instances
        P1, P2… Pk, k>=1;
7.  Apply D And C to each of these sub problems;
8. return combine (D And C(P1), D And C(P2),…….,D And C(Pk));
9.   }
10. }
```

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

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

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

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

Example:

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

$T(n) = 2T(n/2)+n$
$= 2[2T(n/2/2)+n/2]+n$
$= [4T(n/4)+n]+n$
$= 4T(n/4)+2n$
$= 4[2T(n/4/2)+n/4]+2n$
$= 4[2T(n/8)+n/4]+2n$
$= 8T(n/8)+n+2n$
$= 8T(n/8)+3n$
*
*
*

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

→ T(n) =2^log n T(n/2^log n) + n log n

→Corresponding to the choice of i=log n

→ Thus, T(n) = 2^log n T(n/2^log n) + n log n

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

BINARY SEARCH

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

- Algorithm, describes this binary search method, where Binsrch has 4I/ps a[], I , l & x.
- It is initially invoked as Binsrch (a,1,n,x)
- A non-recursive version of Binsrch is given below.
- This Binsearch has 3 i/ps a,n, & x.
- The while loop continues processing as long as there are more elements left to check.
- At the conclusion of the procedure 0 is returned if x is not present, or 'j' is returned, such that a[j]=x.
- We observe that low & high are integer Variables such that each time through the loop either x is found or low is increased by at least one or high is decreased at least one.

- Thus we have 2 sequences of integers approaching each other and eventually low becomes > than high & causes termination in a finite no. of steps if 'x' is not present.

Example:

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

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

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

→ We try the following values for x: 151, -14 and 9.
for 2 successful searches &
1 unsuccessful search.

- Table. Shows the traces of Bin search on these 3 steps.

X=151	low	high	mid
	1	14	7
	8	14	11
	12	14	13
	14	14	14
			Found

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

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

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

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

- Initially low =1, high= n,n>=0, and a[1]<=a[2]<=……..<=a[n].
- If n=0, the while loop is not entered and is returned.

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

Maximum and Minimum:

- Let us consider another simple problem that can be solved by the divide-and-conquer technique.

- The problem is to find the maximum and minimum items in a set of 'n' elements.

- In analyzing the time complexity of this algorithm, we once again concentrate on the no. of element comparisons.

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

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

```
1.  Algorithm straight MaxMin(a,n,max,min)
2.   // set max to the maximum & min to the minimum of a[1:n]
3.  {
4.          max:=min:=a[1];
5.          for I:=2 to n do
6.          {
7.              if(a[I]>max) then max:=a[I];
8.              if(a[I]<min) then min:=a[I];
9.           }
10.  }
```

Algorithm: Straight forward Maximum & Minimum

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

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

- Hence we can replace the contents of the for loop by,
  If(a[I]>max) then max:=a[I];
  Else if (a[I]<min) then min:=a[I];

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

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

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

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

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

  → Let P=(n, a[I] ,……,a[j]) denote an arbitrary instance of the problem.
  → Here 'n' is the no. of elements in the list (a[I],….,a[j]) and we are interested in finding the maximum and minimum of the list.

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

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

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

Algorithm: Recursively Finding the Maximum & Minimum

```
1. Algorithm MaxMin (I,j,max,min)
2. //a[1:n] is a global array, parameters I & j
3. //are integers, 1<=I<=j<=n.The effect is to
4. //set max & min to the largest & smallest value
5. //in a[I:j], respectively.
6. {
7. if(I=j) then max:= min:= a[I];
8. else if (I=j-1) then // Another case of small(p)
9. {
10. if (a[I]<a[j]) then
11. {
12. max:=a[j];
13. min:=a[I];
14. }
```

```
15. else
16. {
17. max:=a[I];
18. min:=a[j];
19. }
20. }
21. else
22. {
23. // if P is not small, divide P into subproblems.
24. // find where to split the set mid:=[(I+j)/2];
25. //solve the subproblems
26. MaxMin(I,mid,max.min);
27. MaxMin(mid+1,j,max1,min1);
28. //combine the solution
29. if (max<max1) then max=max1;
30. if(min>min1) then min = min1;
31. }
32. }
```

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

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

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

No. of element Comparison:

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

$$T(n)=\begin{cases} T([n/2]+T[n/2]+2 & n>2 \\ 1 & n=2 \\ 0 & n=1 \end{cases}$$

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

T(n) = 2T(n/2) +2
= 2(2T(n/2)+2)+2
= 4T(n/2)+4+2
*
*
= 2^k-1T(2)+
= 2^k-1+2^k-2
= 2^k/2+2^k-2
= n/2+n-2
= (n+2n)/2)-2

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

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

MERGE SORT

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

Algorithm For Merge Sort:

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

```
14. //combine the solutions .
15. merge(low,mid,high);
16. }
17. }
```

Algorithm: Merging 2 sorted subarrays using auxiliary storage.

```
1.  Algorithm merge(low,mid,high)
2.  //a[low:high] is a global array containing
3.  //two sorted subsets in a[low:mid]
4.  //and in a[mid+1:high].The goal is to merge these 2 sets into
5.  //a single set residing in a[low:high].b[] is an auxiliary global array.
6.  {
7.  h=low; I=low; j=mid+1;
8.  while ((h<=mid) and (j<=high)) do
9.  {
10. if (a[h]<=a[j]) then
11. {
12.     b[I]=a[h];
13.     h = h+1;
14. }
15. else
16. {
17.     b[I]= a[j];
18.      j=j+1;
19. }
20. I=I+1;
21. }
22. if (h>mid) then
23.    for k=j to high do
24.     {
25.          b[I]=a[k];
26.          I=I+1;
27.      }
28.  else
29.      for k=h to mid do
30.      {
31.           b[I]=a[k];
32.           I=I+1;
33.       }
34.      for k=low to high do a[k] = b[k];
35. }
```

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

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

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

→ Where vertical bars indicate the boundaries of sub arrays.

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

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

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

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

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

→ Elements a[6] &a[7] are merged.

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

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

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

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

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

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

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

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

T(n)=O(n log n)

QUICK SORT

- The divide-and-conquer approach can be used to arrive at an efficient sorting method different from merge sort.
- In merge sort, the file a[1:n] was divided at its midpoint into sub arrays which were independently sorted & later merged.
- In Quick sort, the division into 2 sub arrays is made so that the sorted sub arrays do not need to be merged later.
- This is accomplished by rearranging the elements in a[1:n] such that a[I]<=a[j] for all I between 1 & n and all j between (m+1) & n for some m, 1<=m<=n.
- Thus the elements in a[1:m] & a[m+1:n] can be independently sorted.
- No merge is needed. This rearranging is referred to as partitioning.
- Function partition of Algorithm accomplishes an in-place partitioning of the elements of a[m:p-1]
- It is assumed that a[p]>=a[m] and that a[m] is the partitioning element. If m=1 & p-1=n, then a[n+1] must be defined and must be greater than or equal to all elements in a[1:n]

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

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

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

```
1. Algorithm Partition(a,m,p)
2. //within a[m],a[m+1],…..,a[p-1] the elements
3. // are rearranged in such a manner that if
4. //initially t=a[m],then after completion
5. //a[q]=t for some q between m and
6. //p-1,a[k]<=t for m<=k<q, and
7. //a[k]>=t for  q<k<p. q is returned
8. //Set a[p]=infinite.
9. {
10. v=a[m];I=m;j=p;
11. repeat
12. {
13. repeat
14.     I=I+1;
15.  until(a[I]>=v);
16. repeat
17.      j=j-1;
18. until(a[j]<=v);
19. if (I<j) then interchange(a,i.j);
20. }until(I>=j);
21. a[m]=a[j]; a[j]=v;
22. retun j;
23. }
```

```
1. Algorithm Interchange(a,I,j)
2. //Exchange a[I] with a[j]
3. {
4. p=a[I];
5. a[I]=a[j];
6. a[j]=p;
7. }
```

Algorithm*:* Sorting by Partitioning

```
1. Algorithm Quicksort(p,q)
2. //Sort the elements a[p],….a[q] which resides
3. //is the global array a[1:n] into ascending
```

```
4.  //order; a[n+1] is considered to be defined
5.  // and must be >= all the elements in a[1:n]
6.  {
7.  if(p<q) then // If there are more than one element
8.  {
9.  // divide p into 2 subproblems
10. j=partition(a,p,q+1);
11. //'j' is the position of the partitioning element.
12. //solve the subproblems.
13. quicksort(p,j-1);
14. quicksort(j+1,q);
15. //There is no need for combining solution.
16. }
17. }
```

Record Program: Quick Sort

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

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

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

Output:
Enter the no. of elements 5
Enter the array elements
3
8
1
5
2
The sorted elements are,
1
2
3
5
8

STRASSON'S MATRIX MULTIPLICAION

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

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

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

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

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

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

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

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

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

For EX:

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

The Divide and conquer method

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

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

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

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

Example

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

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

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

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

T(n)= | b n<=2 a &b are
      | 8T(n/2)+cn^2 n>2 constant

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

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

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

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

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

GREEDY METHOD

- Greedy method is the most straightforward designed technique.
- As the name suggest they are short sighted in their approach taking decision on the basis of the information immediately at the hand without worrying about the effect these decision may have in the future.

DEFINITION:

- A problem with N inputs will have some constraints .any subsets that satisfy these constraints are called a feasible solution.
- A feasible solution that either maximize can minimize a given objectives function is called an optimal solution.

Control algorithm for Greedy Method:

```
1.Algorithm Greedy (a,n)
2.//a[1:n] contain the 'n' inputs
3. {
4.solution =0;//Initialise the solution.
5.For i=1 to n do
6.{
7.x=select(a);
8.if(feasible(solution,x))then
9.solution=union(solution,x);
10.}
11.return solution;
12.}
```

* The function select an input from a[] and removes it. The select input value is assigned to X.

- Feasible is a Boolean value function that determines whether X can be included into the solution vector.
- The function Union combines X with The solution and updates the objective function.
- The function Greedy describes the essential way that a greedy algorithm will once a particular problem is chosen ands the function subset, feasible & union are properly implemented.

Example

- Suppose we have in a country the following coins are available :

Dollars(100 cents)
Quarters(25 cents)
Dimes(10 cents)
Nickel(5 Cents)
Pennies(1 cent)

- Our aim is paying a given amount to a customer using the smallest possible number of coins.
- For example if we must pay 276 cents possible solution then,

→ 1 doll+7 q+ 1 pen➔9 coins
→ 2 doll +3Q +1 pen➔6 coins
→ 2 doll+7dim+1 nic +1 pen➔11 coins.

KNAPSACK PROBLEM

- we are given n objects and knapsack or bag with capacity M object I has a weight Wi where I varies from 1 to N.

- The problem is we have to fill the bag with the help of N objects and the resulting profit has to be maximum.

- Formally the problem can be stated as

  Maximize xipi subject to XiWi<=M
  Where Xi is the fraction of object and it lies between 0 to 1.

- There are so many ways to solve this problem, which will give many feasible solution for which we have to find the optimal solution.

- But in this algorithm, it will generate only one solution which is going to be feasible as well as optimal.

- First, we find the profit & weight rates of each and every object and sort it according to the descending order of the ratios.

- Select an object with highest p/w ratio and check whether its height is lesser than the capacity of the bag.

- If so place 1 unit of the first object and decrement .the capacity of the bag by the weight of the object you have placed.

- Repeat the above steps until the capacity of the bag becomes less than the weight of the object you have selected .in this case place a fraction of the object and come out of the loop.

- Whenever you selected.

ALGORITHM:

```
1.Algorityhm Greedy knapsack (m,n)
2//P[1:n] and the w[1:n]contain the profit
3.// & weight res'.of the n object ordered.
4.//such that p[i]/w[i] >=p[i+1]/W[i+1]
5.//n is the Knapsack size and x[1:n] is the solution vertex.
6.{
7.for I=1 to n do a[I]=0.0;
8.U=n;
9.For I=1 to n do
10.{
```

11.if (w[i]>u)then break;
13.x[i]=1.0;U=U-w[i]
14.}
15.if(i<=n)then x[i]=U/w[i];
16.}

Example:

Capacity=20
N=3 ,M=20
Wi=18,15,10
Pi=25,24,15

Pi/Wi=25/18=1.36,24/15=1.6,15/10=1.5

Descending Order ➔ Pi/Wi➔1.6 1.5 1.36
Pi = 24 15 25
Wi = 15 10 18
Xi = 1 5/10 0
PiXi=1*24+0.5*15➔31.5

The optimal solution is ➔31.5

X1	***X2***	***X3***	***WiXi***	***PiXi***
½	1/3	¼	16.6	24.25
1	2/5	0	20	18.2
0	2/3	1	20	31
0	1	½	20	31.5

Of these feasible solution Solution 4 yield the Max profit .As we shall soon see this solution is optimal for the given problem instance.

JOB SCHEDULING WITH DEAD LINES

The problem is the number of jobs, their profit and deadlines will be given and we have to find a sequence of job, which will be completed within its deadlines, and it should yield a maximum profit.

Points To remember:

- To complete a job, one has to process the job or a action for one unit of time.
- Only one machine is available for processing jobs.
- A feasible solution for this problem is a subset of j of jobs such that each job in this subject can be completed by this deadline.
- If we select a job at that time ,

→Since one job can be processed in a single m/c. The other job has to be in its waiting state until the job is completed and the machine becomes free.

→So the waiting time and the processing time should be less than or equal to the dead line of the job.

ALGORITHM:

```
Algorithm JS(d,j,n)
//The job are ordered such that p[1]>p[2]…>p[n]
//j[i]  is  the ith job in the optimal solution
// Also at terminal d [ J[ i]<=d[ J {i+1],1≤i<k
 {
  d[0]= J[0]=0;
J[1]=1;
K=1;
For I =1 to n do
{ // consider jobs in non increasing  order of P[I];find the position for I and check feasibility insertion
r=k;
while((d[J[r]]>d[i] )and
  (d[J[r]] ≠  r)do r =r-1;
if (d[J[r]]<d[I])and (d[I]>r))then
{
for q=k to (r+1) step –1 do J [q+1]=j[q]
J[r+1]=i;
K=k+1;
}
}
return k;
}
```

Example :

1. n=5 (P1,P2,…P5)=(20,15,10,5,1)
   (d1,d2….d3)=(2,2,1,3,3)

Feasible solution	***Processing Sequence***	***Value***
(1)	(1)	20
(2)	(2)	15
(3)	(3)	10
(4)	(4)	5
(5)	(5)	1
(1,2)	(2,1)	35

(1,3)	(3,1)	30
(1,4)	(1,4)	25
(1,5)	(1,5)	21
(2,3)	(3,2)	25
(2,4)	(2,4)	20
(2,5)	(2,5)	16
(1,2,3)	(3,2,1)	45
(1,2,4)	(1,2,4)	40

The Solution 13 is optimal

2. n=4 (P1,P2,…P4)=(100,10,15,27)
(d1,d2….d4)=(2,1,2,1)

Feasible solution	***Processing Sequence***	***Value***
(1,2)	(2,1)	110
(1,3)	(1,3)	115
(1,4)	(4,1)	127
(2,3)	(9,3)	25
(2,4)	(4,2)	37
(3,4)	(4,3)	42
(1)	(1)	100
(2)	(2)	10
(3)	(3)	15
(4)	(4)	27

The solution 3 is optimal.

MINIMUM SPANNING TREE

- Let G(V,E) be an undirected connected graph with vertices ‘v’ and edge ‘E’.
- A sub-graph t=(V,E’) of the G is a Spanning tree of G iff ‘t’ is a tree.3
- The problem is to generate a graph G’= (V,E) where ‘E’ is the subset of E,G’ is a Minimum spanning tree.
- Each and every edge will contain the given non-negative length .connect all the nodes with edge present in set E’ and weight has to be minimum.

NOTE:

- We have to visit all the nodes.
- The subset tree (i.e) any connected graph with ‘N’ vertices must have at least N-1 edges and also it does not form a cycle.

Definition:

- A spanning tree of a graph is an undirected tree consisting of only those edge that are necessary to connect all the vertices in the original graph.
- A Spanning tree has a property that for any pair of vertices there exist only one path between them and the insertion of an edge to a spanning tree form a unique cycle.

Application of the spanning tree:

1. Analysis of electrical circuit.
2. Shortest route problems.

Minimum cost spanning tree:

- The cost of a spanning tree is the sum of cost of the edges in that trees.
- There are 2 method to determine a minimum cost spanning tree are

1. Kruskal's Algorithm
2. Prom's Algorithm.

KRUSKAL'S ALGORITHM:

In kruskal's algorithm the selection function chooses edges in increasing order of length without worrying too much about their connection to previously chosen edges, except that never to form a cycle. The result is a forest of trees that grows until all the trees in a forest (all the components) merge in a single tree.

- In this algorithm, a minimum cost-spanning tree 'T' is built edge by edge.
- Edge are considered for inclusion in 'T' in increasing order of their cost.

  - An edge is included in 'T' if it doesn't form a cycle with edge already in T.
  - To find the minimum cost spanning tree the edge are inserted to tree in increasing order of their cost

Algorithm:

```
Algorithm kruskal(E,cost,n,t)
//E→set of edges in G has 'n' vertices.
//cost[u,v]→cost of edge (u,v).t→set of edge in minimum cost spanning tree
// the first cost is returned.
{
for i=1 to n do parent[I]=-1;
I=0;mincost=0.0;
While((I<n-1)and (heap not empty)) do
{
```

```
j=find(n);
k=find(v);
if(j not equal k) than
{
i=i+1
t[i,1]=u;
t[i,2]=v;
mincost=mincost+cost[u,v];
union(j,k);
    }
  }
if(i notequal n-1) then write("No spanning tree")
else return minimum cost;
}
```

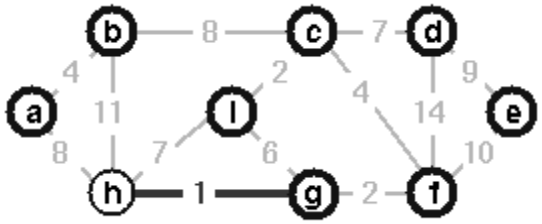
Analysis

- The time complexity of minimum cost spanning tree algorithm in worst case is O(|E|log|E|),

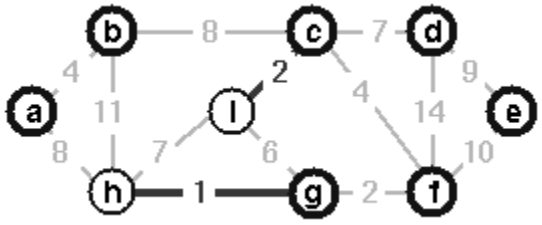
→where E is the edge set of G.

Example: Step by Step operation of Kurskal algorithm.

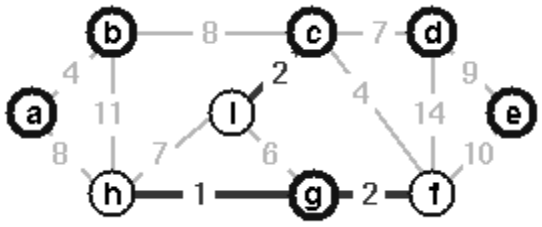
Step 1. In the graph, the Edge(g, h) is shortest. Either vertex g or vertex h could be representative. Lets choose vertex g arbitrarily.



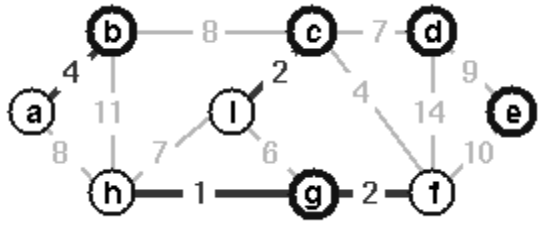
Step 2. The edge (c, i) creates the second tree. Choose vertex c as representative for second tree.



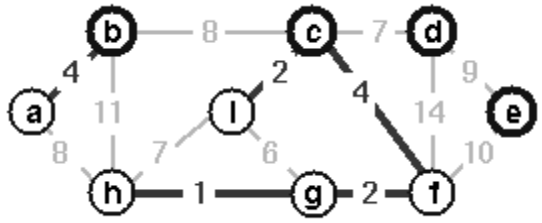
Step 3. Edge (g, g) is the next shortest edge. Add this edge and choose vertex g as representative.



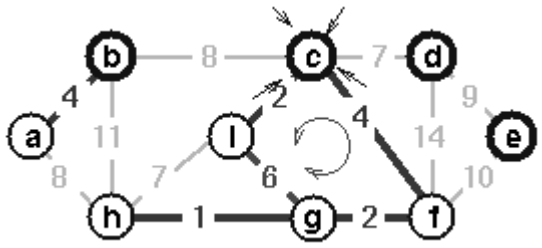
Step 4. Edge (a, b) creates a third tree.



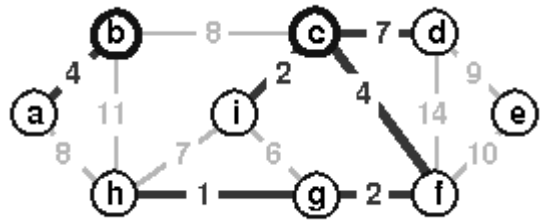
Step 5. Add edge (c, f) and merge two trees. Vertex c is chosen as the representative.



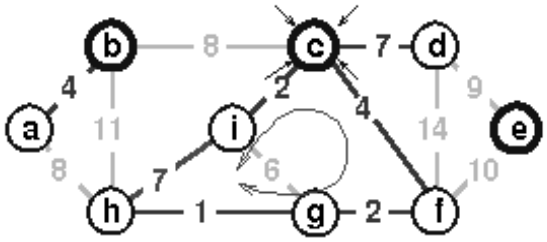
Step 6. Edge (g, i) is the next next cheapest, but if we add this edge a cycle would be created. Vertex c is the representative of both.



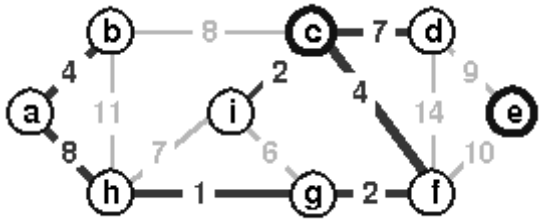
Step 7. Instead, add edge (c, d).



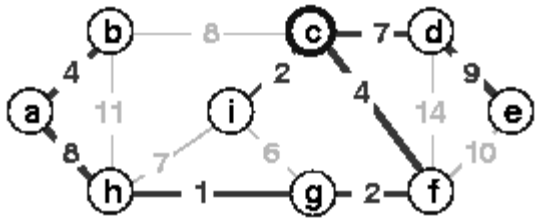
Step 8. If we add edge (h, i), edge(h, i) would make a cycle.



Step 9. Instead of adding edge (h, i) add edge (a, h).

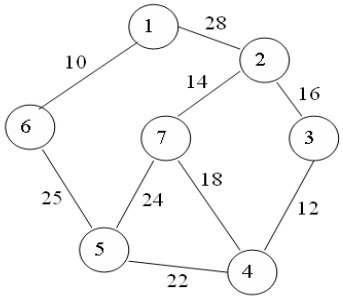


Step 10. Again, if we add edge (b, c), it would create a cycle. Add edge (d, e) instead to complete the spanning tree. In this spanning tree all trees joined and vertex c is a sole representative.



PRIM'S ALGORITHM

Start from an arbitrary vertex (root). At each stage, add a new branch (edge) to the tree already constructed; the algorithm halts when all the vertices in the graph have been reached.

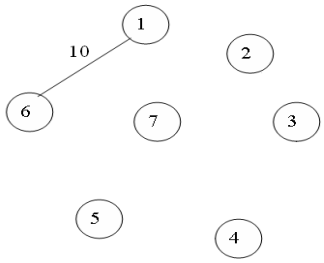


Algorithm prims(e,cost,n,t)

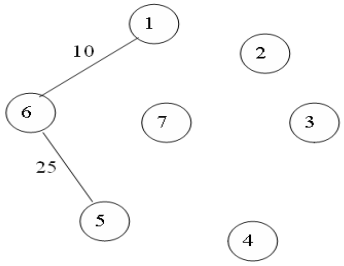
```
{
 Let (k,l) be an edge of minimum cost in E;
 Mincost :=cost[k,l];
 T[1,1]:=k; t[1,2]:=l;
 For I:=1 to n do
    If (cost[i,l]<cost[i,k]) then near[i]:=l;
    Else near[i]:=k;
Near[k]:=near[l]:=0;
For i:=2 to n-1 do
 {
  Let j be an index such that near[j]≠0 and
  Cost[j,near[j]] is minimum;
  T[i,1]:=j; t[i,2]:=near[j];
  Mincost:=mincost+ Cost[j,near[j]];
  Near[j]:=0;
  For k:=0 to n do
     If near((near[k]≠0) and (Cost[k,near[k]]>cost[k,j])) then
    Near[k]:=j;
 }
Return mincost;
}
```

- The prims algorithm will start with a tree that includes only a minimum cost edge of G.
- Then, edges are added to the tree one by one. the next edge (i,j) to be added in such that I is a vertex included in the tree, j is a vertex not yet included, and cost of (i,j), cost[i,j] is minimum among all the edges.
- The working of prims will be explained by following diagram

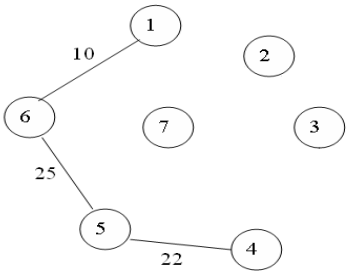
Step 1:



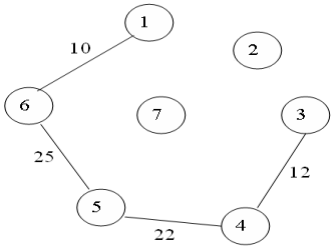
Step 2:



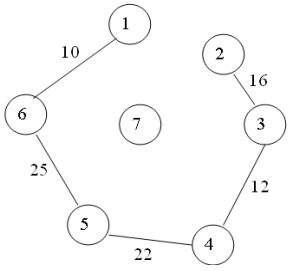
Step 3:



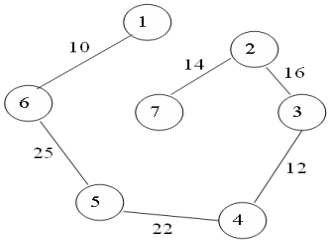
Step 4:



Step 5:



Step 6:



SINGLE SOURCE SHORTEST PATH

Single-source shortest path:

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

1. Is there a path from A to B?

2. If there is more than one path from A to B? Which is the shortest path?

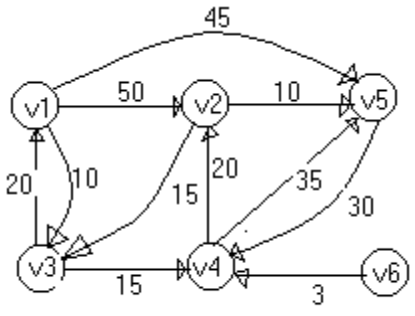


Fig 7.1

The problems defined by these questions are special case of the path problem we study in this section. 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. The graphs are digraphs representing streets. Consider a digraph G=(V,E), with the distance to be traveled as weights on the edges. The problem is to determine the shortest path from v0 to all the remaining vertices of G. It is assumed that all the weights associated with the edges are positive. The shortest path between v0 and some other node v is an ordering among a subset of the edges. Hence this problem fits the ordering paradigm.

Example:
Consider the digraph of fig 7-1. Let the numbers on the edges be the costs of travelling along that route. If a person is interested travel from v1 to v2, then he encounters many paths. Some of them are

1. v1□ v2 = 50 units
2. v1□ v3□ v4□ v2 = 10+15+20=45 units
3. v1□ v5□ v4□ v2 = 45+30+20= 95 units
4. v1□ v3□ v4□ v5□ v4□ v2 = 10+15+35+30+20=110 units

The cheapest path among these is the path along v1□ v3□ v4□ v2. The cost of the path is 10+15+20 = 45 units. Even though there are three edges on this path, it is cheaper than travelling along the path connecting v1 and v2 directly i.e., the path v1□ v2 that costs 50 units. One can also notice that, it is not possible to travel to v6 from any other node.
To formulate a greedy based algorithm to generate the cheapest paths, we must conceive a multistage solution to the problem and also of an optimization measure. One possibility is to build the shortest paths one by one. As an optimization measure we can use the sum of the lengths of all paths so far generated. For this measure to be minimized, each individual path must be of minimum length. If we have already constructed i shortest

paths, then using this optimization measure, the next path to be constructed should be the next shortest minimum length path. The greedy way to generate these paths in non-decreasing order of path length. First, a shortest path to the nearest vertex is generated. Then a shortest path to the second nearest vertex is generated, and so on.
A much simpler method would be to solve it using matrix representation. The steps that should be followed is as follows,

Step 1: find the adjacency matrix for the given graph. The adjacency matrix for fig 7.1 is given below

	V1	V2	V3	V4	V5	V6
V1	-	50	10	Inf	45	Inf
V2	Inf	-	15	Inf	10	Inf
V3	20	Inf	-	15	inf	Inf
V4	Inf	20	Inf	-	35	Inf
V5	Inf	Inf	Inf	30	-	Inf
V6	Inf	Inf	Inf	3	Inf	-

Step 2: consider v1 to be the source and choose the minimum entry in the row v1. In the above table the minimum in row v1 is 10.

Step 3: find out the column in which the minimum is present, for the above example it is column v3. Hence, this is the node that has to be next visited.

Step 4: compute a matrix by eliminating v1 and v3 columns. Initially retain only row v1. The second row is computed by adding 10 to all values of row v3.
The resulting matrix is

	V2	V4	V5	V6
V1□ Vw	50	Inf	45	Inf

V1□ V3□ Vw	10+inf	10+15	10+inf	10+inf
Minimum	50	25	45	inf

Step 5: find the minimum in each column. Now select the minimum from the resulting row. In the above example the minimum is 25. Repeat step 3 followed by step 4 till all vertices are covered or single column is left.

The solution for the fig 7.1 can be continued as follows

	V2	V5	V6
V1□ Vw	50	45	Inf
V1□ V3□ V4□ Vw	25+20	25+35	25+inf
Minimum	45	45	inf

	V5	V6
V1□ Vw	45	Inf
V1□ V3□ V4□ V2□ Vw	45+10	45+inf
Minimum	45	inf

	V6
V1□ Vw	Inf
V1□ V3□ V4□ V2□ V5□ Vw	45+inf
Minimum	inf

Finally the cheapest path from v1 to all other vertices is given by V1□ V3□ V4□ V2□ V5.