Unit 2

Introduction

As the name (Divide and rule) suggests, in this strategy the big problem is broken down into smaller sub problems and solution to these sub problems is obtained. The applications such as binary search, merge sort and quick sort use divide and conquer concept.

Divide and conquer

1. Divide into smaller sub problems

2. These sub problems are solved independently

3. Combining all the solutions of sub problems into a solution of the whole.

If the sub problems are large enough then divide and conquer is reapplied. The generated sub problems are usually of same type as the original problem. Hence recursive algorithms are used on divide and conquer strategy. A control abstraction for divide and conquer is as given below- using control abstraction a flow of control of a procedure is given.

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)= { g(n) n small
T(n1)+T(n2)+……+T(nk)+f(n); otherwise.

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.

Control algorithm for Divide and Conquer Method:

```
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) = { T(1) n=1
AT(n/b)+f(n) n>1

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

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

Example:

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

We have,

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

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

Divide-and-Conquer Algorithms

The ***divide and conquer*** strategy solves a problem by:
1. Breaking into ***sub problems*** that are themselves smaller instances of the same type of problem.
2. Recursively solving these sub-problems.
3. Appropriately combining their answers.

Two types of sorting algorithms which are based on this divide and conquer algorithm:
1. **Quick sort:** Quick sort also uses few comparisons (somewhat more than the other two). Like heap sort it can sort "in place" by moving data in an array.
2. **Merge sort:** Merge sort is good for data that's too big to have in memory at once, because its pattern of storage access is very regular. It also uses even fewer comparisons than heap sort, and is especially suited for data stored as linked lists.

Searching techniques

Linear Search/Sequential Search-sequential search on unsorted data or on sorted data

In this Searching Technique, search element is compared Sequentially with each element in an array and process of comparison is stopped when element is matched with array element. If not, element is not found.

Algorithm

Algorithm LinearSearch(a,n,x)

```
1. {
2. i:=0
3. while(i<n)
4. {
5. if(a[i]= =x) then
6. key element is found and exit;
7. i:=i+1;
8. }
9. element not found
10. }
```

Analysis:

If there are n items in the list, then it is obvious that in worst case (i.e. when there is no target element in the list) N comparisons are required. Hence the worst case performance of this algorithm is roughly proportional to N as O(n).

The best case, in which the first comparison returns match, it requires a single comparison and hence it is O(1).

The average time depends on the probability that the key will be found in the list. Thus the average case roughly requires N/2 comparisons to search the element. That means, the average time, as in worst case, is proportional to N and hence it is O(N).

Binary search/Divide and conquer scheme-Search on sorted data

- Here elements must be in Ascending/Descending order.

Iterative Binary Search Algorithm

Algorithm BinarySearch(a,n,x)

```
1. {
2. low:=1,high:=n;
3. while(low<=high) do
4. {
5.        mid=⌊(low + high)/2⌋;
6.       if(x<a[mid]) then high:=mid-1;
7.       else if(x>a[mid]) then low:=mid+1;
8.       else return mid;
9. }
10. return 0;
11. }
```

Recursive Binary Search Algorithm

Algorithm BinarySearch(a,l,h,x)

```
1. {//given an array a[l:h] in increasing order
```

```
2. if(l= =h) then
3. {
4.        if(x= =a[h]) then return h;
5.        else return 0;
6.  }
7. else
 8. {
9.  mid=⌊(low + high)/2⌋;
10. if(x= =a[mid]) then return mid;
11. else if(x<a[mid]) then return BinarySearch(a,l,mid-1,x);
12. else return BinarySearch(a,mid+1,h,x);
13. }
14. }
```

Analysis:

The sequential search situation will be in worst case if the element is at the end of the list.
For eliminating this problem, we have one efficient search technique called binary search. The condition for binary search is that all the data should be in sorted array. We compare the element with middle element of the array. If it is less than the middle element then search it in the left portion of the array and if it is greater than the middle element then search will be in the right portion of the array. Now we will take that portion only for search and compare with middle element of that portion. This process will be in iteration until we find the element or middle element has no left or right portion to search.

Each step of the algorithm divides the list into two parts, and the search continues in one of them and the other is discarded.

$T(N)=T(N/2)+1$ if $N>1$
$T(1)=1$ if $N=1$

$T(N) = T(N/2)+1$
$= T(N/4)+2$
$= T(N/8)+3$
:
:
$= T(N/2^k)+k$
$= T(1)+\log_2 N$
$O(\log_2 N)$

The search requires at most K steps of comparison where $2^K >= N$ which results $k=log_2 N$. Thus the running time(both average and worst cases) of a binary search is proportional to $log_2 N$ i.e $O(log_2 N)$.

Finding the Maximum and Minimum

```
Algorithm StraightMaxMin(a,n,max,min)
1.  {
2.       max:=min:=a[1];
3.      for i:=2 to n do
4.      {
5.          if(a[i]>max) then max:=a[i];
6.          if(a[i]<max) then min:=a[i];
7.      }
8.  }
```

StrainghtMaxMin algorithm takes 2(n-1) comparisons in the best, average and worst cases.

Recursive algorithm for finding maximum and minimum

```
Algorithm MaxMin(i,j,max,min)
1.  {   // given a[i:j]
2.      if(i = =j) then max:=min:=a[i]; //if one element
3.      else if(i = = j-1) then  //if two elements
4.      {
5.          if(a[i]<a[j]) then
6.          {
7.              max:=a.[j]; min:=a[i];
8.          }
9.          else
10.         {
11.             max:=a[i];min:=a[j];
12.         }
13.     else  //if more than two elements exist split the array
14.     {
15.         mid:= ⌊(i + j)/2⌋;
16.         MaxMin(i,mid,max,min);
17.         MaxMin(mid+1,j,max1,min1);
18.         if(max<max1) then max:=max1;
19.         if(min>min1) then min:=min1;
20.     }
21. }
```

Time Complexity Analysis of MaxMin Algorithm

$$T(N) = \begin{cases} T\left(\left\lceil \frac{N}{2} \right\rceil\right) + T\left(\left\lceil \frac{N}{2} \right\rceil\right) + 2 & N > 2 \\ 1 & N = 2 \\ 0 & N = 1 \end{cases}$$

T(N) =2T(N/2)+2

$=2(2T(N/4)+2)+2$
$=4T(N/4)+4+2$
:
:
$=2^{k-1}T(N/2^{k-1})+2^{k-1}+\ldots..+4+2$
$=2^{k-1}T(2)+\sum_{1\le i\le k-1} 2^i$
$=N/2+2^k - 2$
$=N/2+N-2 \qquad =3N/2-2$

Compared with 2N-2, MaxMin algorithm is taking 3N/2-2 which reduces 25% of comparisons. But there exists overhead of stack space for recursion.

Merge Sort

If there are two sorted lists of array then process of combining these sorted lists into sorted order is called merging.

Take one element of each array, compare them and then take the smaller one in third array. Repeat this process until the elements of any array are finished. Then take the remaining elements of unfinished array in third array.

Merge sort

We take the pair of consecutive array elements, merge them in sorted array and then take adjacent pair of array elements and so on until all the elements of array are in single list.

//Merge sort

```
Algorithm MergeSort(low,high)
{
        if(low<high) then
            {
                mid=⌊(low + high)/2⌋;
                MergeSort(low,mid);
                MergeSort(mid+1,high);
                Merge(low,mid,high);
            }
}
Algorithm Merge(low, mid, high)
{
            i=low;
            j=mid+1;
            k=low;
            while(i<=mid && j<=high) do
              {
                    if(a[i]<a[j]) then
                        b[k++]=a[i++];
                    else
                        b[k++]=a[j++];
              }
        while(i<=mid) do
```

```
            b[k++]=a[i++];
        while(j<=high) do
            b[k++]=a[j++];
        for i:=low to high do
            a[i]=b[i];
}
```

Analysis:

Let us take an array of size n is used for merge sort. Because here we take elements in pair and merge with another pair after sorting. So, merge sort requires maximum $\log_2 n$ passes. Hence we can say merge sort requires $n*\log_2 n$ comparisons which is $O(n\log_2 n)$. The main disadvantage of merge sort is space requirement. It requires extra space of O(n).

$$T(N) = \begin{cases} 1 & N = 1, a\ is\ a\ constant \\ 2T\left(\frac{N}{2}\right) + cN & N > 1, c\ is\ a\ constant \end{cases}$$

T(N) = 2T(N/2)+cN=2(2T(n/4) +cn/2) +cn

=4T(N/4)+2cN=4(2T(n/8)+cn/4)+2cn

=8T(N/8)+3cN

:

:

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

$=N+c*N\log_2 N$

$=O(N\log_2 N)$

Quick sort/Partition exchange sort

The basic version of quick sort algorithm was invented by C. A. R. Hoare in 1960 and formally introduced quick sort in 1962. It is used on the principle of divide-and-conquer. Quick sort is an algorithm of choice in many situations because it is not difficult to implement, it is a good "general purpose" sort and it consumes relatively fewer resources during execution.

The idea behind this sorting is that sorting is much easier in two short lists rather than one long list. Divide and conquer means divide the big problem into two small problems and then those two small problems into two small ones and so on. As example, we have a list of 100 names and we want to list them alphabetically then we will make two lists for names. A-L and M-Z from original list. Then we will divide list A-L into A-F and G-L and so on until the list could be easily sorted. Similar policy we will adopt for the list M-Z.

Algorithm QuickSort(p,q)

```
1. {
2. if (p<q) then
3. {
4.         j=partition(a,p,q+1);
5.         QuickSort(p, j-1);
6.         QuickSort(j+1, q);
```

```
7.  }
8.  }
```

Quick sort works by partitioning a given array a[*p* . . *q*] into two non-empty sub array a[*p* . . *j-1*] and a[*j*+1 . . *q*] such that every key in *a*[*p* . . *j-1*] is less than or equal to every key in *A*[*j*+1 . . *q*]. Then the two subarrays are sorted by recursive calls to Quick sort. The exact position of the partition depends on the given array and index j is computed as a part of the partitioning procedure.

Note that to sort entire array, the initial call Quick Sort (*a*, 1, length[*a*])

As a first step, Quick Sort chooses as pivot one of the items in the array to be sorted. Then array is then partitioned on either side of the pivot. Elements that are less than or equal to pivot will move toward the left and elements that are greater than or equal to pivot will move toward the right.

Partitioning the Array

Partitioning procedure rearranges the subarrays in-place.

Algorithm Partition(a, m, n)

```
1.  {
2.     pivot = a[m]; i=m; j=n;
3.     while (i < j) do
4.    {
5.       while ( a[i] < pivot ) do i++;
6.       while ( a[j] > pivot ) do j--;
7.        if  ( i < j ) then interchange a[i] and a[j];
8.     }
9.   a[m]=a[j];
10.  a[j]=pivot;
11. return j;
12. }
```

Partition selects the first key, a[*p*] as a pivot key about which the array will partitioned: Keys≤a[p] will be moved towards the left. Keys ≥ a[q] will be moved towards the right. The running time of the partition procedure is O(*n*) where *n* = *q* - *p* +1 which is the number of keys in the array.

Another argument that running time of partition on a subarray of size O(*n*) is as follows: Pointer *i* and pointer *j* start at each end and move towards each other, conveying somewhere in the middle. The total number of times that *i* can be incremented and *j* can be decremented is therefore O(*n*). Associated with each increment or decrement there are O(1) comparisons and swaps. Hence, the total time is O(*n*).

Performance of Quick Sort

The running time of quick sort depends on whether partition is balanced or unbalanced, which in turn depends on which elements of an array to be sorted are used for partitioning.

A very good partition splits an array up into two equal sized arrays. A bad partition, on other hand, splits an array up into two arrays of very different sizes. The worst partition puts only one element in one array and all other elements in the other array. If the partitioning is balanced, the

Quick sort runs asymptotically as fast as merge sort. On the other hand, if partitioning is unbalanced, the Quick sort runs asymptotically as slow as insertion sort.

Analysis:

Time requirement of quick sort depends on the position of pivot in the list, how pivot is dividing list into sub lists. It may be equal division of list or maybe it will not divide also.

Average Case: In average case we assume that list is equally divided means list1 is equally divided in to two sub lists, these two sub lists into four sub lists and so on.

If there are n elements in the list, where n is approximately 2^k. Hence we can say **$k=\log_2 n$**.

In the list of n elements, pivot is placed in the middle with n comparisons and the left and right subtrees have approximately n/2 elements each. Hence these require again n/2 comparisons each to place their pivots in the middle of the lists. These two lists are again divided into 4 sublists and so on.

Total number of comparisons=n+2*n/2+4*n/4+8*n/8+…….n*n/n

=n+n+n+……+n(k times)

=O(n*k)

$=O(n\log_2 n)$

The number of comparison at any level will be maximum n. So we can say run time of quick sort will be of O(nlogn)

Best Case

The best thing that could happen in Quick sort would be that each partitioning stage divides the array exactly in half. In other words, the best to be a median of the keys in a[p . . q] every time procedure 'Partition' is called. The procedure 'Partition' always split the array to be sorted into two equal sized arrays.

If the procedure 'Partition' produces two regions of size $n/2$. the recurrence relation is then

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

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

$= O(n \lg n)$

Worst case Partitioning

The worst-case occurs if given array A[1 . . n] is already sorted. The Partition(a,m, p) call always return p so successive calls to partition will split arrays of length n, n-1, n-2, . . . , 2 and running time proportional to $n + (n-1) + (n-2) + \ldots + 2 = [(n+2)(n-1)]/2 = \Theta(n^2)$. The worst-case also occurs if a[1 . . n] starts out in reverse order.

Suppose list of elements are already in sorted order. When we find the pivot then it will be first element. So here it produces only 1 sub list which is on right side of first element second element. Similarly other sub lists will be created only at right side. The number of comparison for first element is n, second element requires n-1 comparisons and so on. So the total number of comparisons will be

=n+(n-1)+(n-2)+……+3+2+1

=n(n-1)/2

Which is $O(n^2)$

Worst case $=O(n^2)$

Average case=Best Case=$O(n\log_2 n)$

Example:

Elements in red indicate swaps.
Elements in blue indicate comparisons.
Special commentary is in green.

3 1 4 5 9 2 6 8 7
Calling quickSort on elements 1 to 9
Definitions: a "small" element is one whose value is less than or equal to the value of the pivot. Likewise, a "large" element is one whose value is larger than that of the pivot. At the beginning, the entire array is passed into the quicksort function and is essentially treated as one large partition.
At this time, two indices are initialized: the left-to-right search index, *i*, and the right-to-left search index, j.
The value of *i* is the index of the first element in the partition, in this case 1, and the value of j is 9, the index of the last element in the partition. The relevance of these variables will be made apparent in the code below.
3 1 4 5 9 2 6 8 7
The first element in the partition, 3, is chosen as the pivot element, around which two subpartitions will be created. The end goal is to have all the small elements at the front of the partition, in no particular order, followed by the pivot, followed by the large elements.
To do this, quicksort will scan rightwards for the first large element. Once this is found, it will look for the first small element from the right. These two will then be swapped. Since *i* is currently set to one, the pivot is actually compared to itself in the search of the first large element.
3 1 4 5 9 2 6 8 7
The search for the first large element continues rightwards. The value of *i* gets incremented as the search moves to the right.
3 1 4 5 9 2 6 8 7
Since 4 is greater than the pivot, the rightwards search stops here. Thus the value of *i* remains 3.
3 1 4 5 9 2 6 8 7
Now, starting from the right end of the array, quicksort searches for the first small element. And so *j* is decremented with each step leftwards through the partition.
3 1 4 5 9 2 6 8 7
3 1 4 5 9 2 6 8 7
3 1 4 5 9 2 6 8 7
Since 2 is not greater than the pivot, the leftwards search can stop.
3 1 2 5 9 4 6 8 7
Now elements 4 and 2 (at positions 3 and 6, respectively) are swapped.
3 1 2 5 9 4 6 8 7
Next, the rightwards search resumes where it left off: at position 3, which is stored in the index *i*.
3 1 2 5 9 4 6 8 7
Immediately a large element is found, and the rightwards search stops with *i* being equal to 4.
3 1 2 5 9 4 6 8 7
Next the leftwards search, too, resumes where it left off: j was 6 so the element at position 6 is compared to the pivot before *j* is decremented again in search of a small element.
3 1 2 5 9 4 6 8 7
This continues without any matches for some time...

3 1 2 5 9 4 6 8 7
3 1 2 5 9 4 6 8 7
The small element is finally found, but no swap is performed since at this stage, *i* is equal to *j*. This means that all the small elements are on one side of the partition and all the large elements are on the other.
2 1 3 5 9 4 6 8 7
Only one thing remains to be done: the pivot is swapped with the element currently at *i*. This is acceptable within the algorithm because it only matters that the small element be to the left of the pivot, but their respective order doesn't matter. Now, elements 1 to (*i*) form the left partition (containing all small elements) and elements *j* + 1 onward form the right partition (containing all large elements).
Calling quickSort on elements 1 to 2
The right partition is passed into the quicksort function.
2 1 3 5 9 4 6 8 7
2 is chosen as the pivot. It is also compared to itself in the search for a small element within the partition.
2 1 3 5 9 4 6 8 7
The first, and in this case only, small element is found.
2 1 3 5 9 4 6 8 7
Since the partition has only two elements, the leftwards search begins at the second element and finds 1.
1 2 3 5 9 4 6 8 7
The only swap to be made is actually the final step where the pivot is inserted between the two partitions. In this case, the left partition has only one element and the right partition has zero elements.
Calling quickSort on elements 1 to 1
Now that the left partition of the partition above is quicksorted: there is nothing else to be done
Calling quickSort on elements 3 to 2
The right partition of the partition above is quicksorted. In this case the starting index is greater than the ending index due to the way these are generated: the right partition starts one past the pivot of its parent partition and goes until the last element of the parent partition. So if the parent partition is empty, the indices generated will be out of bounds, and thus no quicksorting will take place.
Calling quickSort on elements 4 to 9
The right partition of the entire array is now being quicksorted 5 is chosen as the pivot.
1 2 3 5 9 4 6 8 7
1 2 3 5 9 4 6 8 7
The rightwards scan for a large element is initiated.
9 is immediately found.
1 2 3 5 9 4 6 8 7
Thus, the leftwards search for a small element begins...
1 2 3 5 9 4 6 8 7
1 2 3 5 9 4 6 8 7
1 2 3 5 9 4 6 8 7
At last, 4 is found. Note *j* = 6.
1 2 3 5 4 9 6 8 7

Thus the first large and small elements to be found are swapped.
1 2 3 5 4 9 6 8 7
The rightwards search for a large element begins anew.
1 2 3 5 4 9 6 8 7
Now that it has been found, the rightward search can stop.
1 2 3 5 4 9 6 8 7
Since j was stopped at 6, this is the index from which the leftward search resumes.
1 2 3 5 4 9 6 8 7
1 2 3 4 5 9 6 8 7
The last step for this partition is moving the pivot into the right spot. Thus the left partition consists only of the element at 4 and the right partition is spans positions 6 to 9 inclusive.
Calling quickSort on elements 4 to 4
The left partition is quicksorted (although nothing is done).
Calling quickSort on elements 6 to 9
The right partition is now passed into the quicksort function.
1 2 3 4 5 9 6 8 7
9 is chosen as the pivot.
1 2 3 4 5 9 6 8 7
The rightward search for a large element begins.
1 2 3 4 5 9 6 8 7
1 2 3 4 5 9 6 8 7
No large element is found. The search stops at the end of the partition.
1 2 3 4 5 9 6 8 7
The leftwards search for a small element begins, but does not continue since the search indices *i* and j have crossed.
1 2 3 4 5 7 6 8 9
The pivot is swapped with the element at the position *j*: this is the last step in splitting this partition into left and right subpartitions.
Calling quickSort on elements 6 to 8
The left partition is passed into the quicksort function.
1 2 3 4 5 7 6 8 9
6 is chosen as the pivot.
1 2 3 4 5 7 6 8 9
The rightwards search for a large element begins from the left end of the partition.
1 2 3 4 5 7 6 8 9
The rightwards search stops as 8 is found.
1 2 3 4 5 7 6 8 9
The leftwards search for a small element begins from the right end of the partition.
1 2 3 4 5 7 6 8 9
Now that 6 is found, the leftwards search stops. As the search indices have already crossed, no swap is performed.
1 2 3 4 5 6 7 8 9
So the pivot is swapped with the element at position *j*, the last element compared to the pivot in the leftwards search.
Calling quickSort on elements 6 to 6
The left subpartition is quicksorted. Nothing is done since it is too small.

Calling quickSort on elements 8 to 8
Likewise with the right subpartition.
Calling quickSort on elements 10 to 9
Due to the "sort the partition starting one to the right of the pivot" construction of the algorithm, an empty partition is passed into the quicksort function. Nothing is done for this base case.
1 2 3 4 5 6 7 8 9
Finally, the entire array has been sorted.

Randomized Quick Sort

In the randomized version of Quick sort we impose a distribution on input. This does not improve the worst-case running time independent of the input ordering.

RANDOMIZED_QUICKSORT (*A, p, r*)

Quicksort can be modified so that it performs well on every input. The solution is the use of randomizer. This is a Las Vegas algorithm since it will always output the correct answer. Every call to the randomizer Random takes a certain amount of time. If there are only a very few elements to sort, the time taken by the randomizer may be comparable to the rest of the computation. For this reason, we invoke the randomizer only if (q-p) >5.

Algorithm Rquicksort(p, q)

```
{
  if ( p < q ) then
  {
          if((q-p)>5) then
          x=random(q-p+1)+p;
                  interchange a[x] and a[p]
          int j = partition(a,p,q+1);
          Rquicksort(p,j-1);
          Rquicksort(j+1, q);
  }
}
```

Like other randomized algorithms, RANDOMIZED_QUICKSORT has the property that no particular input elicits its worst-case behavior; the behavior of algorithm only depends on the random-number generator. Even intentionally, we cannot produce a bad input for RANDOMIZED_QUICKSORT unless we can predict generator will produce next.

Advantages:
One of the fastest algorithms on average.
Does not need additional memory (the sorting takes place in the array - this is called **in-place** processing).
Compare with merge sort: merge sort needs additional memory for merging.
Disadvantages: The worst-case complexity is $O(N^2)$

Applications:

Commercial applications use Quicksort - generally it runs fast, no additional memory, this compensates for the rare occasions when it runs with $O(N^2)$

Never use in applications which require **guaranteed response time**:

Life-critical (medical monitoring, life support in aircraft and space craft)

Mission-critical (monitoring and control in industrial and research plants handling dangerous materials, control for aircraft, defence, etc) **unless you assume the worst-case response time**.

Quick sort Comparison with mergesort:

mergesort guarantees O(NlogN) time, however it requires additional memory with size N.

quicksort does not require additional memory, however the speed is not guaranteed usually mergesort is not used for main memory sorting, only for external memory sorting. So far, our best sorting algorithm has **O(*n*log *n*)** performance: can we do any better? *In general*, the answer is **no.**

Partition-based general selection algorithm

A general selection algorithm that is efficient in practice, but has poor worst-case performance, was conceived by the inventor of quicksort, C.A.R. Hoare, and is known as **Hoare's selection algorithm** or **quickselect**.

In quicksort, there is a subprocedure called partition that can, in linear time, group a list (ranging from indices `left` to `right`) into two parts, those less than a certain element, and those greater than or equal to the element.

In quicksort, we recursively sort both branches, leading to best-case O(*n* log *n*) time. However, when doing selection, we already know which partition our desired element lies in, since the pivot is in its final sorted position, with all those preceding it in sorted order and all those following it in sorted order. Thus a single recursive call locates the desired element in the correct partition:

//Selection problem

```
// The partition based select function
Algorithm Select(a,n,k)
{
        low:=1;high:=n;
        while(low<high) do
        {
                j:=partition(a,low,high);
                if(k= =j) then return j;
                else if(k<j) then high=j;
                else low:=j+1;
        }
}
```

Strassen Matrix multiplication

Suppose we want to multiply two matrices A and B each of size N i.e.

C=A x B then

$$\begin{matrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{matrix} = \begin{matrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{matrix} \text{ X } \begin{matrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{matrix}$$

Then multiplication gives

$C_{11}= A_{11} xB_{11}+ A_{12} xB_{21}$

$C_{12}= A_{11} xB_{12}+ A_{12} xB_{22}$

$C_{21}= A_{21} xB_{11}+ A_{22} xB_{21}$

$C_{22}= A_{21} xB_{12}+ A_{22} xB_{22}$

Thus to accomplish 2 x 2 matrix multiplication there are total 8 multiplications and 4 additions.

To accomplish this multiplication we can write the following algorithm for the same.

Algorithm Mat_Mul(A,B,C,n)

```
{
        for i:=1 to n do
                for j:=1 to n do
                C[i,j]:=0;
                for k:=1 to n do
                        C[i,j]:=C[i,j]+A[i,j] x b[i,j];
}
```

The time complexity of above algorithm turns to be O(nxnxn)=O(n^3)

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

Strassen showed that 2 x 2 matrix multiplications can be accomplished in 7 multiplications and 18 additions or subtractions.

The divide and conquer approach can be used for implementing **Strassen's matrix multiplication.**

--Divide: Divide matrices into sub-matrices:A0,A1,A2 etc.

--Conquer: use a group of matrix multiply equations.

--Combine: recursively multiply sub-matrices and get the final result of multiplication after performing required additions or subtractions.

$S1= (A_{11}+ A_{22})(B_{11}+ B_{22})$

$S2=(A_{11}+ A_{22}) \times B_{11}$

$S3= A_{11} \times (B_{12} - B_{22})$

$S4= A_{22} \times (B_{21} – B_{11})$

$S5=(A_{11}+ A_{12}) \times B_{22}$

$S6= (A_{21}- A_{11})(B_{11}+ B_{12})$

$S7= (A_{12}- A_{22})(B_{21}+ B_{22})$

$C_{11}=S1+S4-S5+S7$

$C_{12}=S3+S5$

$C_{21}=S2+S4$

$C_{22}=S1+S3-S2+S6$

Now we will compare the actual our traditional matrix multiplication procedure with Strassens procedure. In Strassens multiplication

$C_{11}=S1+S4-S5+S7$

$=(A_{11}+ A_{22})(B_{11}+ B_{22}) + A_{22} \times (B_{21} – B_{11}) -(A_{11}+ A_{12}) \times B_{22+}(A_{12}- A_{22})(B_{21}+ B_{22})$

$= A_{11} xB_{11}+ A_{12} xB_{21}$

Matrix multiplication by using recursion to divide the problem in terms of sub problems and solving the terminating condition of recursion by using Strassens's matrix multiplication.

Steps involved in this approach are:

1. Divide the matrix in 4 different parts by using divide and conquer technique.
2. Multiply each part by using recursion
3. Solve the terminating condition of the recursion by using Strassens matrix multiplication

Algorithm Matmul(A[n][n],B[n][n],C[n][n])

//This algorithm implements matrix multiplication by dividing matrix into sub matrix A0,A1,A2……..B0,B1,B2……And multiplying each of them recursively.

//Input: Two matrices A and B of dimension n

//Output: matrix C of dimension n

If(n= =2)

Multiply the two input matrix by using Strassens's Matrix

Else

Divide the matrix in to sub matrix of dimension n/2 and solve recursively.

- Divide matrices into submatrices:A0,A1,A2 etc
- Use blocked matrix multiply equations
- Recursively multiply sub matrices

A=$\begin{matrix} A & A \\ A & A \end{matrix}$,B=$\begin{matrix} B & B \\ B & B \end{matrix}$,C=$\begin{matrix} C & C \\ C & C \end{matrix}$,D=$\begin{matrix} D & D \\ D & D \end{matrix}$, a=$\begin{matrix} a & a \\ a & a \end{matrix}$, b=$\begin{matrix} b & b \\ b & b \end{matrix}$, c=$\begin{matrix} c & c \\ c & c \end{matrix}$,d=$\begin{matrix} d & d \\ d & d \end{matrix}$

$\begin{matrix} A & B \\ C & D \end{matrix}$ x $\begin{matrix} a & b \\ c & d \end{matrix}$ =

$$\begin{bmatrix} \begin{bmatrix} Aa+Bc \end{bmatrix} & \begin{bmatrix} Ab+Bd \end{bmatrix} \\ \begin{bmatrix} Ca+Dc \end{bmatrix} & \begin{bmatrix} Cb+Dd \end{bmatrix} \end{bmatrix}$$

Each letter represents an n/2 by n/2 matrix

We can use the breakdown to form a divide and conquer algorithm

Applying above algorithm when the size is not in the power of 2

Steps required is as follows

1. Make the size to nearest power of 2
2. Make the value of all extra rows to be zero
3. Make the value of all extra columns to be zero
4. Then apply above algorithm
   Suppose the dimension of given matrix is 3 then
   Nearest of 3 is 4 which is power of 2

Analysis of Algorithm

T(1)=1 Assume $n=2^k$ =>$k=\log_2 n$

T(n)=7 T(n/2)

$T(n)=7^k T(n/2^k)$

$T(n)=7^{\log n}$

$T(n)= 7^{\log n}=(2^{2.81})^{\log_2 n} =n^{2.81}$

Thus divide and conquer is an algorithmic strategy having with the principle idea of dividing the problem into sub problems. Then solution to these problems is obtained in order to get the final solution for the given problem.

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 $\begin{vmatrix} 32 & 32 \\ 32 & 32 \end{vmatrix}$

GREEDY METHOD

- Greedy method is the most straightforward designed technique.
- As the name suggests 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 maximizes or minimizes a given objective 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;//Initialize 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 and 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 W*i* 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 $x_i p_i$ subject to $X_i W_i <= M$

  Where X_i 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 solutions from which we have to find the optimal solution.
- But in this algorithm, we 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 weight 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.Algorithm GreedyKnapsack (m,n)
2//p[1:n] and the w[1:n] contain the profit
3.// & weights of the n objects ordered.
4.//such that p[i]/w[i] >=p[i+1]/w[i+1]
5.//m is the Knapsack size and x[1:n] is the solution vertex.
6.{
7.for i=1 to n do x[i]=0.0;
8.U=m;   p=0;
9.for i=1 to n do
10.{
11.if (w[i]>U) then break;
13.x[i]=1.0;U=U-w[i];p=p+p[i];
14.}
15.if(i<=n) then x[i]=U/w[i]; p=p+x[i]*p[i];
16.}
```

Example:

Capacity=20
N=3, M=20
W_i=18,15,10
P_i=25,24,15
P_i/W_i=25/18=1.36,24/15=1.6,15/10=1.5
Descending Order ➔ P_i/W_i➔1.6 1.5 1.36

P_i = 24 15 25
W_i = 15 10 18
X_i = 1 5/10 0

P_iX_i=1*24+0.5*15➔31.5
The optimal solution is ➔31.5

X_1	X_2	X_3	W_iX_i	P_iX_i
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 . This solution is optimal for the given problem instance.

Exercise1:

A Thief enters a house for robbing it. He can carry a maximum weight of 60kg into his bag. There are 5 items in the house with the following weights and values. What item should thief take when he can even take the fraction of any item with him?

Item	Weight	Value
1	5	30
2	10	40
3	15	45
4	22	77
5	25	70

Exercise2:

Find the optimal solution for the for knapsack problem by using greedy approach consider

n=5, w=60kg

(w1,w2,w3,w4,w4,w5)=(5,10,15,22,25)

(p1,p2,p3,p4,p5)=(30,40,45,77,90)

OPTIMAL STORAGE ON TAPES

Optimal storage problem:

Input: We are given 'n' programs that are to be stored on computer tape of length L and the length of program i is Li

Such that $1 \leq i \leq n$ and $\Sigma\ 1 \leq k \leq j\ Li \leq 1$

Output: A permutation from all n! For the n programs so that when they are stored on tape in the order the Mean Retrieval Time (MRT) is minimized.

Example1:

Let n = 3, (l1, l2, l3) = (8, 12, 2). As n = 3, there are 3! =6 possible ordering.

All these orderings and their respective d value are given below:

Ordering	**d (i)**	**Value**
1, 2, 3	8 + (8+12) + (8+12+2)	50
1, 3, 2	8 + 8 + 2 + 8 + 2 + 12	40
2, 1, 3	12 + 12 + 8 +12 + 8 + 2	54
2, 3, 1	12 + 12 +2 +12 + 2 + 8	48
3, 1, 2	2 + 2 + 8 + 2 + 8+ 12	34
3, 2, 1	2 + 2 +12 + 2 + 12 + 8	38

The optimal ordering is 3, 1, 2.

The greedy method is now applied to solve this problem. It requires that the programs are stored in non-decreasing order which can be done in O (nlogn) time.

Greedy solution:

i. Make tape empty

ii. For i:= 1 to n do;

iii. Grab the next shortest path

iv. Put it on next tape.

The algorithm takes the best shortest term choice without checking to see whether it is a big long term decision.

Algorithm:

```
Algorithm Store(n,m)

//n programs on m tapes

{

        j:=0;
```

```
for i:=1 to n do
        {
        write("append program",i, to permutation for tape", j);
        j:=(j+1)mod m;
        }
}
```

Exercise: Find an optimal placement for 13 programs on 3 tapes T0, T1 & T2 where the program are of lengths 12, 5, 8, 32, 7, 5, 18, 26, 4, 3, 11, 10 and 6.

OPTIMAL MERGE PATTERN

Merge a set of sorted files of different length into a single sorted file. We need to find an optimal solution, where the resultant file will be generated in minimum time.

There exist many ways to merge records into a single sorted file. This merge can be performed pair wise. Hence, this type of merging is called as **2-way merge patterns**.

As, different pairings require different amounts of time, in this strategy we want to determine an optimal way of merging many files together. At each step, two shortest sequences are merged.

To merge a **p-record file** and a **q-record file** requires possibly **p + q** record moves, the obvious choice being, merge the two smallest files together at each step.

Two-way merge patterns can be represented by binary merge trees. Let us consider a set of **n** sorted files **{f_1, f_2, f_3, …, f_n}**. Initially, each element of this is considered as a single node binary tree. To find this optimal solution, the following algorithm is used.

Treenode=

Record {treenode *llink,*rlink;

Integer weight; };

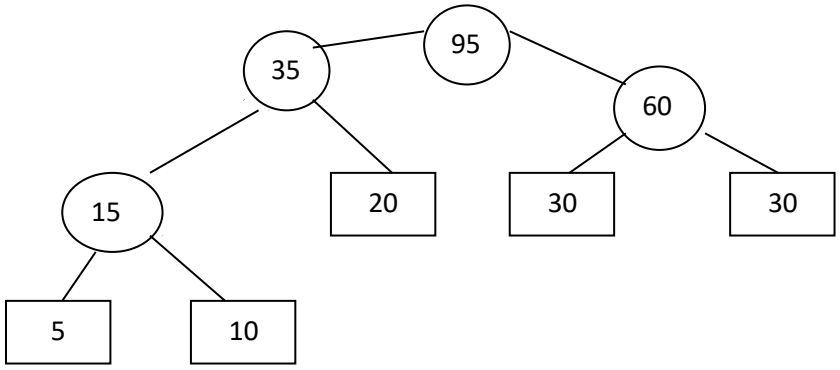
```
Algorithm: TREE (n)
{
for i := 1 to n - 1 do
   {
```

```
        pt->llink := Least (list)
        pt->rlink:= Least (list)
        pt->weight := ((pt->llink)->weight) + ((pt->rlink)->weight)
        Insert (list, pt);
    }
return Least(list);
}
```

At the end of this algorithm, the weight of the root node represents the optimal cost.

Example1: Find an optimal binary merge pattern for five files whose lengths are 20,30,10,5 and 30.

Solution:



Exercise1: Find an optimal binary merge pattern for ten files whose lengths are 28,32,12,5,84,53,91,35,3 and 11.

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 jobs, 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 an 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: High-Level description of job sequencing algorithm

```
Algorithm GreedyJob(d,J,n)
{
J:={1};
        for i:=2 to n do
        {
                if(all jobs in J U{i} can be completed by their deadlines) then J:=J U{i};
        }
}
```

ALGORITHM: Greedy Algorithm for sequencing unit time jobs with deadlines and profits

```
Algorithm JS(d,J,n)
//The jobs are ordered such that p[1]>p[2]…>p[n]
//J[i]  is  the i^th 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 J;
}
```

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.

Exercise 1:

Job	**T1**	**T2**	**T3**	**T4**	**T5**	**T6**	**T7**	**T8**	**T9**
Deadline	7	2	5	3	4	5	2	7	3
Profit	15	20	30	18	18	10	23	16	25

Exercise 2: Solve the job sequencing problem for the given data n=5, profits (1,5,20,15,10) and deadlines (1,2,4,1.3) using greedy strategy

Exercise 3: What is the solution generated by the function Job sequencing when N=7, (P_1, P_2, …..P_7) = (3, 5, 20, 18, 1, 6, 30) and (d_1, d_2, … d_7) = (1, 3, 4, 3, 2, 1, 2)

Exercise 4: Solve the job sequencing problem for the given data n = 5, profits (1,5,20,15,10) and deadlines (1,2,4,1.3) using greedy strategy.

MINIMUM SPANNING TREE

- Let G'(V,E) be an undirected connected graph with vertices 'V' and edges 'E'.
- A sub-graph t=(V,E) of the G is a Spanning tree of G iff 't' is a tree
- 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 edges 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 tree.
- There are 2 methods 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.
- Edges 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) then
{
i=i+1;
t[i,1]=u;
t[i,2]=v;
mincost=mincost+cost[u,v];
union(j,k);
     }
   }
if (i not equal n-1) then write("No spanning tree")
else return minimum cost;
}
```

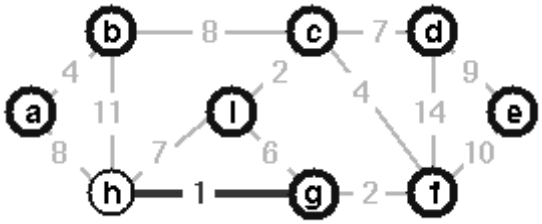
Analysis

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

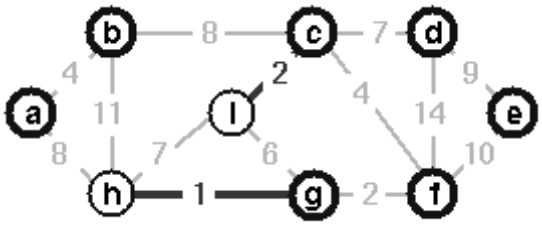
->where E is the edge set of G.

Example: Step by Step operation of Kurskal's algorithm.

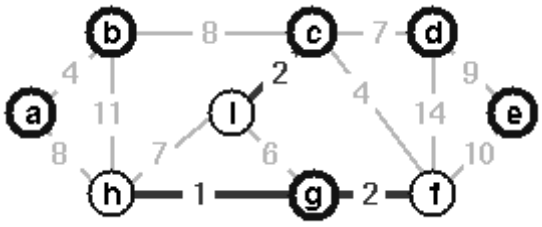
Step 1. In the graph, the Edge(g, h) is shortest. Either vertex g or vertex h could be representative. Lets choose vertex g arbitrarily.



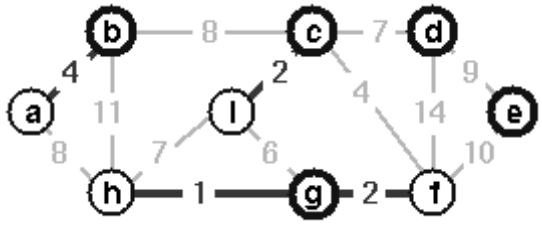
Step 2. The edge (c, i) creates the second tree. Choose vertex c as representative for second tree.



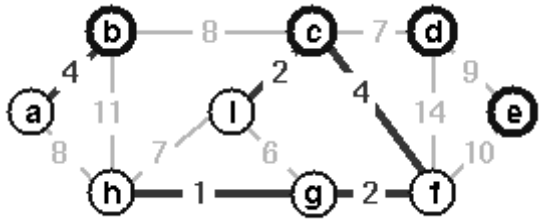
Step 3. Edge (g, g) is the next shortest edge. Add this edge and choose vertex g as representative.



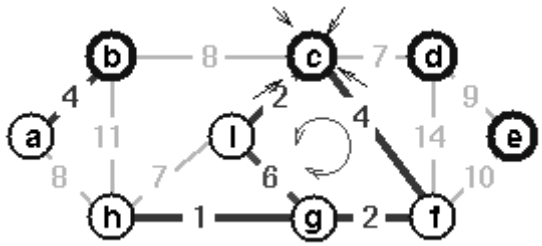
Step 4. Edge (a, b) creates a third tree.



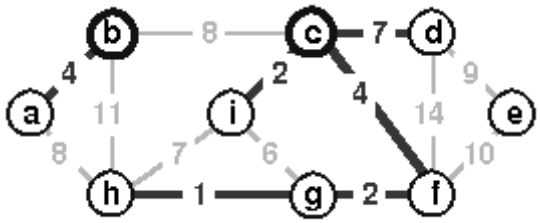
Step 5. Add edge (c, f) and merge two trees. Vertex c is chosen as the representative.



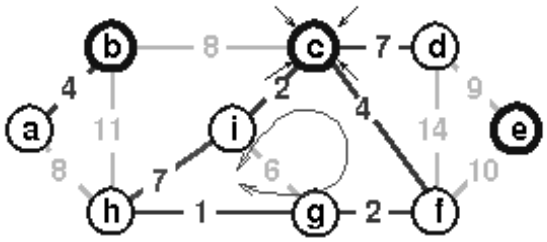
Step 6. Edge (g, i) is the next cheapest, but if we add this edge a cycle would be created. Vertex c is the representative of both.



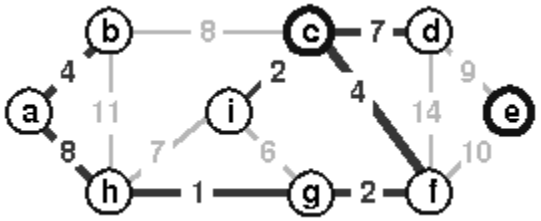
Step 7. Instead, add edge (c, d).



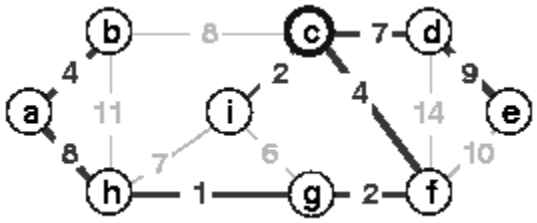
Step 8. If we add edge (h, i), edge(h, i) would make a cycle.



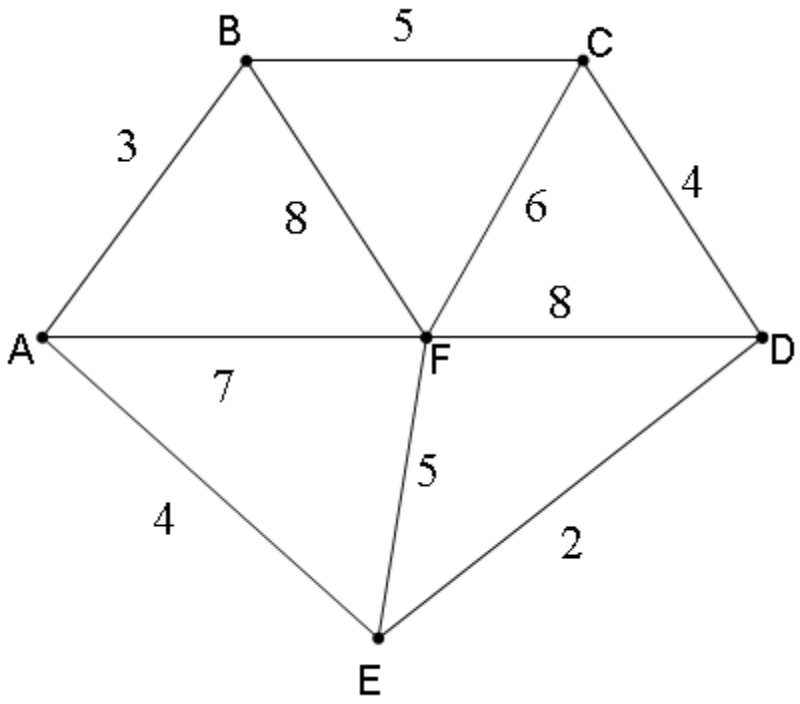
Step 9. Instead of adding edge (h, i) add edge (a, h).



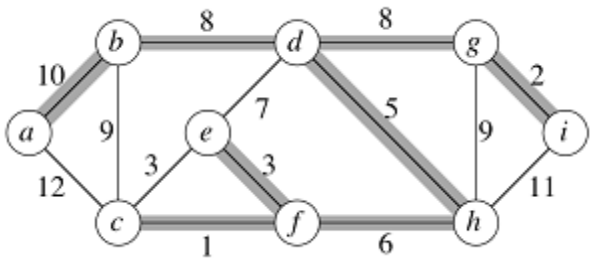
Step 10. Again, if we add edge (b, c), it would create a cycle. Add edge (d, e) instead to complete the spanning tree. In this spanning tree all trees joined and vertex c is a sole representative.



EXERCISE1: Find the optimal path of the following minimal spanning tree by applying Kruskal's Algorithm.

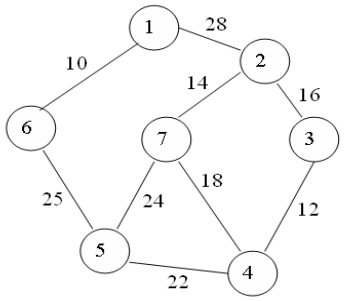


EXERCISE2: Find the optimal path of the following minimal spanning tree by applying Kruskal's Algorithm.



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: High level description of prims

```
Algorithm Prims()
{
T=ϕ;
U={1};
While(U!=V){
Let (u,v) be lowcost edge such that u∈U and v∈V-U;
T=TU{(u,v)};
U=U U{v};
}
```

Algorithm prims(e,cost,n,t)

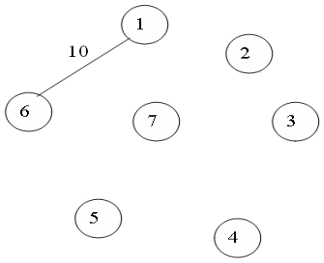
```
{
  Mincost :=cost[k,l];
  t[1,1]:=k; t[1,2]:=l;
  for i:=1 to n do
     if (cost[i,l]<cost[i,k]) then near[i]:=l;
     else near[i]:=k;
 near[k]:=near[l]:=0;
 for i:=2 to n-1 do
  {
     t[i,1]:=j; t[i,2]:=near[j];
    Mincost:=mincost+cost[j,near[j]];
    near[j]:=0;
    for k:=1 to n do
       if ((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.

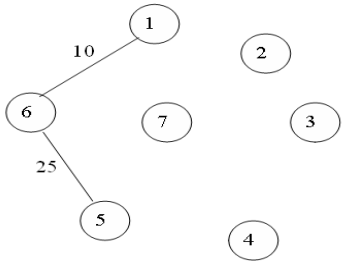
- The working of prims will be explained by following diagram

EXAMPLE1:

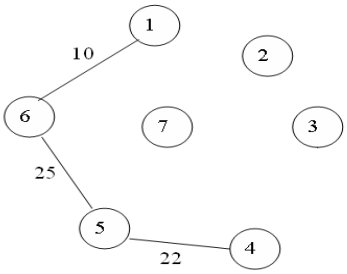
Step 1:



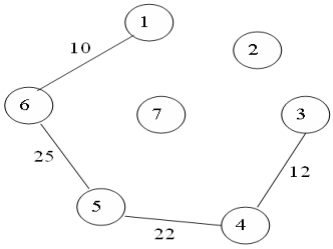
Step 2:



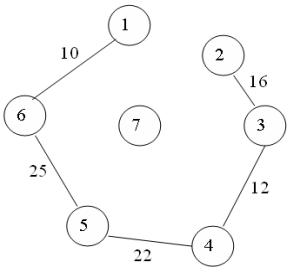
Step 3:



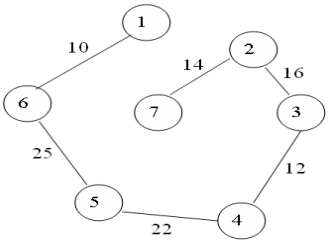
Step 4:



Step 5:

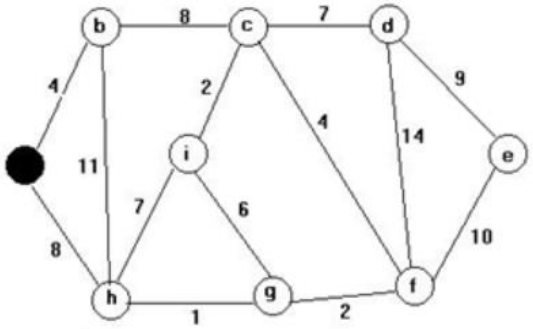


Step 6:

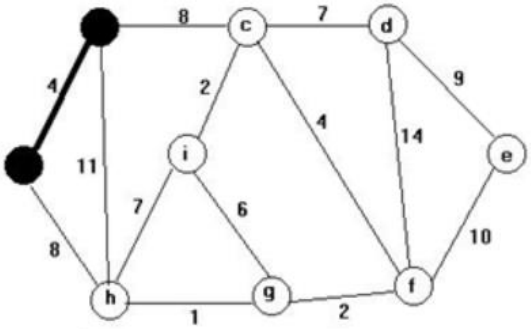


EXAMPLE2: Find the optimal path of the following minimal spanning tree by applying Prim's Algorithm.

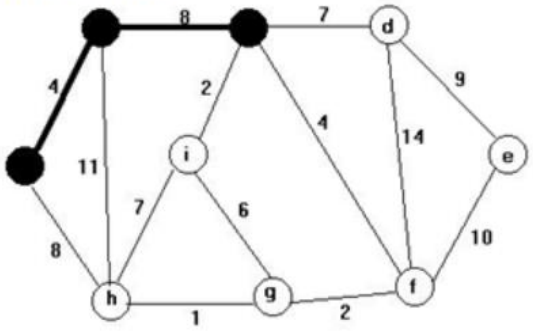
Iteration 0:



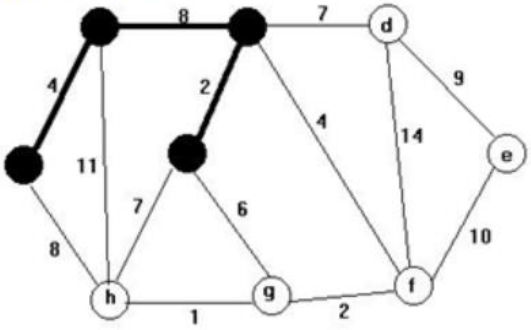
Iteration 1:



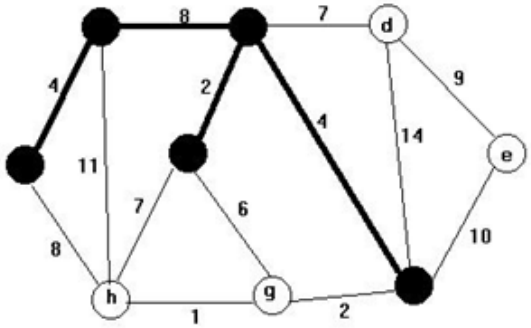
Iteration 2:



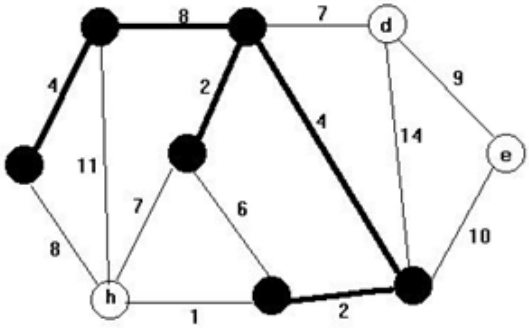
Iteration 3:



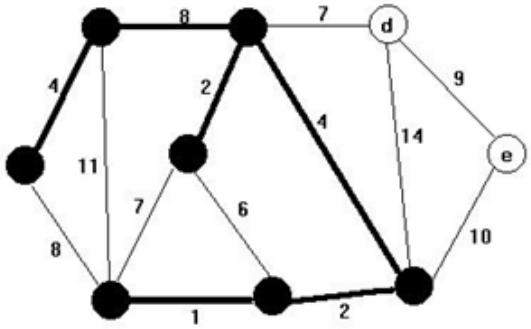
Iteration 4:



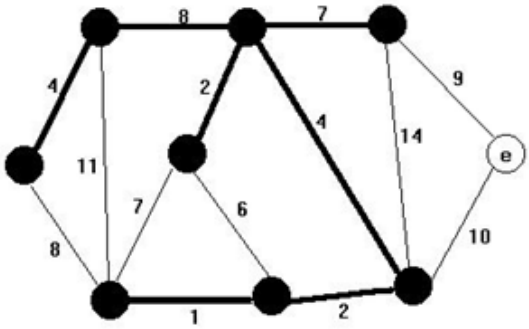
Iteration 5:



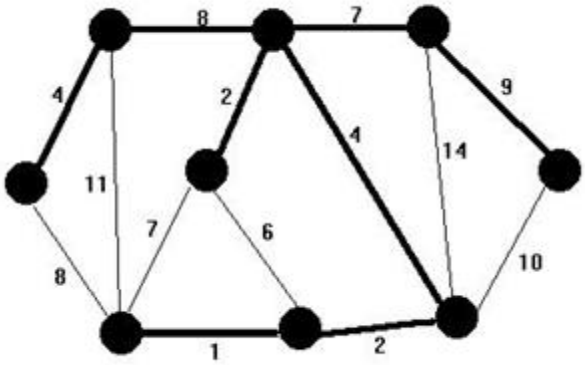
Iteration 6:



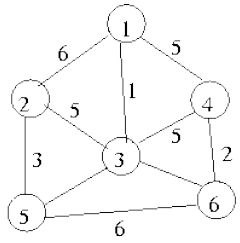
Iteration 7:

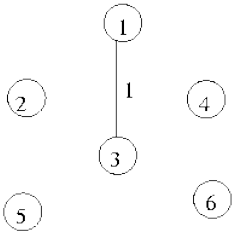


Iteration 8:

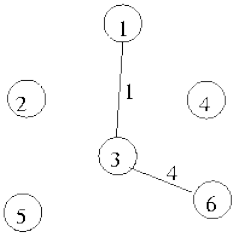


EXAMPLE3: Find the optimal path of the following minimal spanning tree by applying Prim's Algorithm.

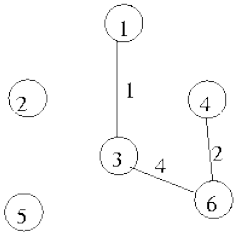




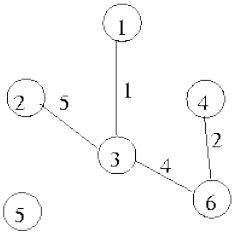
Iteration 1. U = {1}



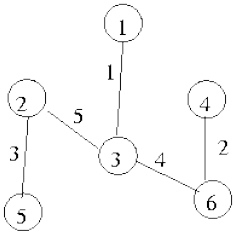
Iteration 2. U = {1,3}



Iteration 3. U = {1,3,6}

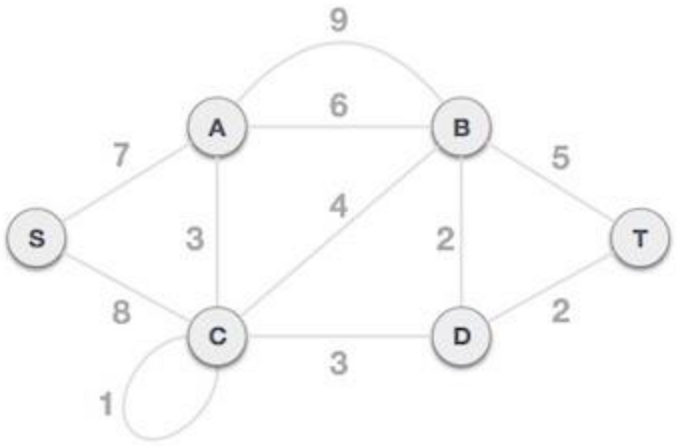


Iteration 4. U = {1,3,6,4}



Iteration 5. U = {1,3,6,4,2}

EXERCISE1: Find the optimal path of the following minimal spanning tree by applying Prim's Algorithm.



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?

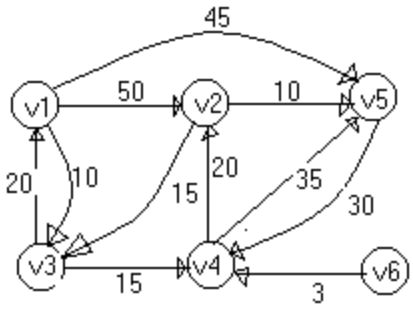


Fig 7.1

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.

Algorithm SingleSource(v,cost,dist,n)
{
  for i:=1 to n **do**
        S[i]:=false; dist[i]:=cost[v,i];
S[v]:=true; dist[i]:=0.0;

```
 for num:=2 to n do
  {
        S[u]:=true;
    for (each w adjacent to u with S[w]==false) do
      if (dist[w] > dist[u] +cost[v,w]) then
        dist[w]:=dist[u]+cost[u,w];
  }
}
```

UNIT-2 Frequently Asked Questions

DIVIDE AND CONQUER

1. What is Divide and Conquer? Give the control abstraction.
2. Write time complexities of
   a) Binary search c) Maximum and Minimum c) Merge sort d) Quick sort
3. Write an algorithm for
   a) Binary search b) Finding Max and Min elements from an array.
   c) Merge sort d) Quick sort
4. Find Matrix Multiplication for given two matrices using Strassen's Matrix.
   A=1 2 B=1 0
   3 4 0 1
5. Write about: Selection Problem of selecting k^{th} smallest element
6. Explain how the merge sort and quick sort algorithm sorts the following numbers 5,8,6,3,7,2

GREEDY METHOD:

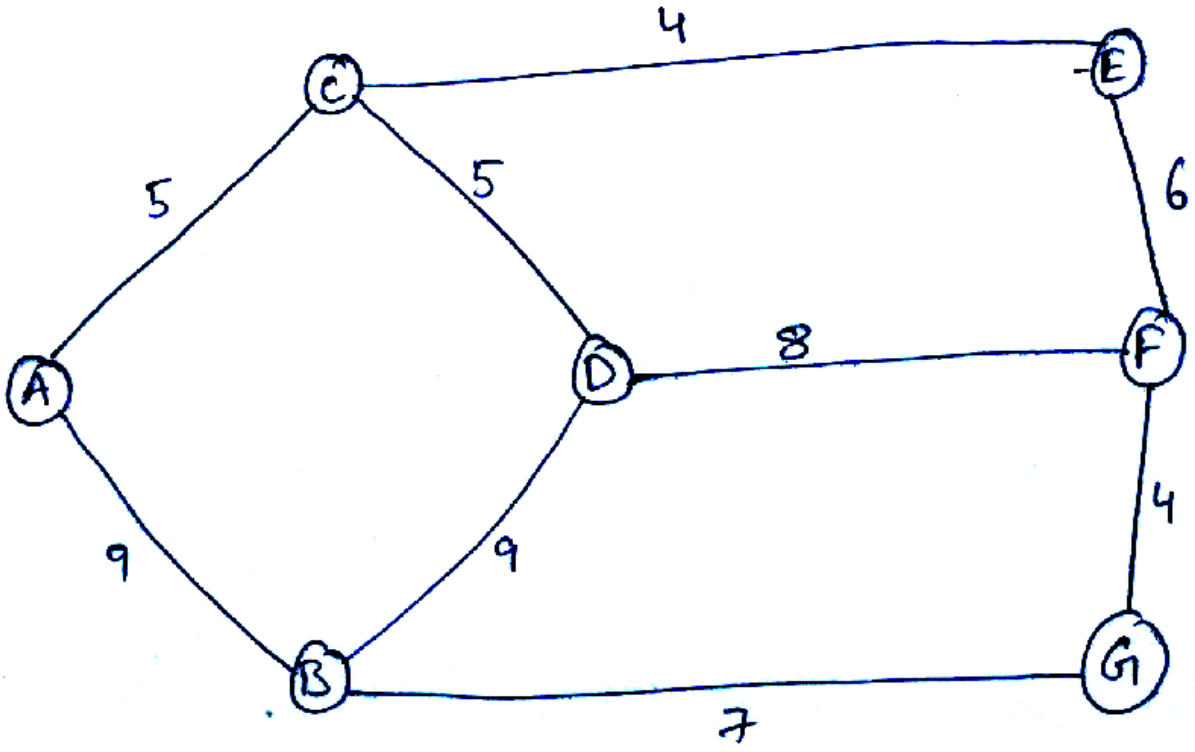
7. Define : a) Feasible solution b) Optimal solution

8. a) What is Knapsack Problem? Explain
   a) Find the Optimal solution to the knapsack instance n=3, M=20 (P1,P2,P3)=(25,24,15) and (W1,W2,W3)=(18,15,10).

9. a) What is Greedy Method? Give the Control Abstraction.
   b) Write a Greedy algorithm for sequencing unit time jobs with deadlines and profits and Give an example.

10. Write about:
    a) Optimal Storage on tapes.
    b) Optimal Merge Pattern.
    c) Single Source Shortest Path

11. Compute Minimum Cost Spanning tree for the graph G Using (a) Prim's algorithm and (b) Kruskal's algorithm.


A
B
C
D
E
F
G
5
9
5
9
4
8
6
4
7