UNIT-II

DIVIDE AND CONQUER: General method, applications-analysis of binary search, quick sort, merge sort, AND OR Graphs.

GREEDY METHOD: General method, Applications-job sequencing with deadlines, 0/1 knapsack problem, minimum cost spanning trees, Single source shortest path problem.

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, $1<k\leq n$, 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 reapplication of the divide-and-conquer principle is naturally expressed by a recursive algorithm. Now smaller and smaller sub problems of the same kind are generated until eventually sub problems that are small enough to be solved without splitting are produced.

Recurrence Relation:

$$T(n) = \begin{cases} g(n) & \text{n small} \\ T(n_1) + T(n_2) + \ldots\ldots\ldots + T(n_k) + f(n) & \text{otherwise} \end{cases}$$

Where T(n) is the time for DAndC on any input of size n and g(n) is the time to compute the answer directly for small inputs. The function f(n) is the time for dividing P and combining the solutions to sub problems.

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

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

Algorithm:

```
Algorithm DAndC(P)
{
        if Small(P) then return S(P);
        else
        {
            divide P into smaller instances P1, P2,……,Pk, k≥1;
            Apply DAndC to each of these sub problems;
            return Combine(DAndC(P1),DAndC(P2),….,DAndC(Pk));
        }
}
```

APPLICATIONS

BINARY SEARCH

Binary search performs well when compared to the linear search. However, the constraint of binary search to work is the items in the list must be in a sorted order(either in ascending or descending order). This is prerequisite condition for binary search. Note that this is not at all required in the case of a sequential search. However, the binary search cannot be applied always because of the sorted order limitation. If the list to be search for a specific item is not in the sorted order, binary search will not work. The complexity of binary search is O(log n).

The binary search uses an approach called as divide and conquer. The logic of divide and conquer is to divide the problem into different subtasks and solve it with smaller set of the data.

Suppose in a given list of elements L in ascending order, we wish to search for the position of an element equal to S. Let minindex is the first element position of given list and maxindex is the last element position in the list. The binary search begins at the mid positiom computed by integer division ((minindex+maxindex)/2), and compare it with the value X. If they are equal, the search is successful and the mid position is returned. Otherwise, if X is lower than the item at midposition, the search continues in the first half of the list, and if X greater than the item at the midposition , the search continues in the second half of the list. The process repeats until the item is found, or there are no elements in the sublist. The terminating condition is found when the minindex exceeds the maxindex.

The best, worst and average cases fort Binary Search are:

BestCase: item in the middle. Binary Search examines only one (middle) position.

WorstCase: item in the lost possible division. The maximal number of items in an array of length N can be divided is $\log_2 N$.

AverageCase: somewhere in between ; $1/2\text{Log}_2 N$

Example:

Consider a list A={7, 12, 23, 45, 67, 89, 90}. Observe that the elements in the list are in sorted order. Here the size of the list is 7. The minimum index of the list, say minindex, is 1, and the maximum index in the list, say maxindex, is 7. If we want to search an element N=67 in the list A using the binary search algorithm , the steps to be followed are given as follows:

1. Compare the element with middle element of the list. That is, middle element index, midposition=(minindex+maxindex)/2=(1+7)/2=4. So compare search element with element 4 in the list.

   67 = A[4] (67 >45)

2. The element 4 in the list=45 is less than the search element. As the elements are in the sorted order and the search element is greater than the element 4, the search element must be present only after 45 in the list (second half). Now search for the element in the list by considering only the next elements after 45 in the list (excluding the middle element i.e 45).

   Consider {67, 89, 90}

3. Now the minindex is 4+1=5 and maxindex is 7.

4. Now the item to be compared is middle element=(5+7)/2=12/2=6.

5. Compare the search element N=67 with the middle element=89.

   67 = A[6] (67 < 89)

6. As the search element is less than the middle element 89, the search element might be present only before the element 89 in the list. That is between minindex element and the midposition element.

7. Now search for the element between minindex and midposition element by considering the middle element index as the max index.

8. Now the maxindex=5 and the minindex=5.

9. As the minindex and maxindex are same there is only one element to consider to be compared with the search element.

10. As the element remaining to be considered for comparison with the search element is same i.e. 67. The search is successful.

The binary search return 5, which is the element(=67) position in the list.

* Let a_i, $1 \leq i \leq n$ be a list of elements which are stored in non decreasing order.

* Consider the problem of determining whether a given element x is present in the list.

* In case x is present, we are to determine a value j such that $a_j = x$.

* If x is not in the list then j is to be set to zero.

* Divide-and-conquer suggests breaking up any instance $I = (n, a_1, \ldots..a_n, x)$ of this search problem into sub problems.

* One possibility is to pick an index k and obtain three instances:

$I_1 = (k-1, a_1, \ldots\ldots, a_{k-1}, x)$, $I_2 = (1, a_k, x)$ and $I_3 = (n-k, a_{k+i}, \ldots..,a_n, x)$

* The search problem for two of these three instances is easily solved by comparing x with a_k.

* If $x=a_k$ then j=k and I_1 and I_3 need not be solved.

* If $x<a_k$ then for I_2 and I_3, j=0 and only I_1 remains to be solved.

* If $x>a_k$ then for I_1 and I_2, j=0 and only I_3 remains to besolved.

* After a comparison with a_k, the instance remaining to be solved can be solved by using divide and conquer scheme.

* If k is always chosen such that a_k is the middle element i.e. k=(n+1)/2 then the resulting search algorithm is known as binary search.

Algorithm:

procedure BINSRCH(A, n, x, j)

```
// given an array A(1:n) of elements in non decreasing order
// n≥0, determine if x is present, and if so, set j such that x=A(j)
// else j=0
integer low, high, mid, j, n;
low←1; high←n;
while low≤high do
mid ← (low+high)/2;
case
     : x<A(mid)   : high ← mid-1;
     : x>A(mid)   : low ← mid+1;
     : else : j ← mid; return
end case
repeat
j ← 0
end BINSRCH
```

QUICK SORT

As the name suggests, Quick sort is a fast sorting algorithm invented by C.A.R.Hoare. It has $O(N\log_2 N)$ for best and average case performance; and $O(N^2)$ for worst-case performance.

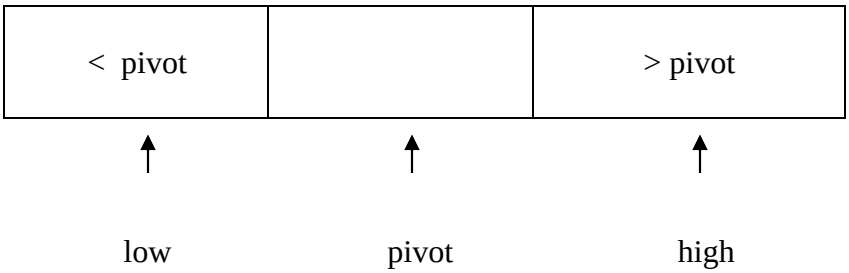
Quick sort approaches sorting in a radically different way from the other sorts. Rather than successively sorting the same data set, it attempts to partition the data set into two sections, then sort each section separately- adopting divide and conquer strategy.

In quick sort, we divide the array of items to be sorted into two partitions and then call the quick sort procedure recursively to sort the two partitions. To partition the data elements, a **pivot** element is to be selected that all the items in the lower part are less than the pivot and all those in the upper part greater than it.

Consider an array A of N elements to be sorted. Select a pivot element among the N elements. The selection of pivot element is some what arbitrary; however, the first element is a convenient one.

The array is divided into two parts so that the pivot element is placed into its proper position satisfying the following properties:

1. All elements to the left of pivot are less than the pivot element.
2. All elements to the right of pivot are greater than or equal to the pivot element.

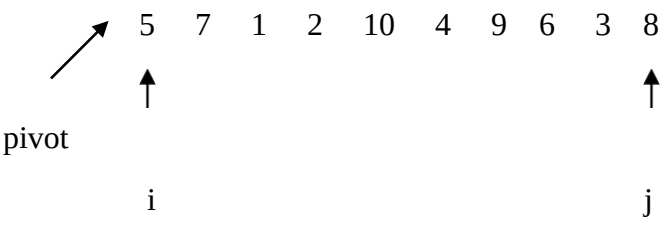


At this stage, the array is divided into two sub arrays. The same process is then repeated recursively on the two parts of the array on either side of the pivot in the same way. Subsequent sub arrays that might result during the process are also sorted like wise until all the elements in the array are in sorted position.

Illustrate the Quick Sort method by applying it to the following array of elements.

5 7 1 2 10 4 9 6 3 8

Take the first element as the pivot element, i.e. pivot=A[1]=5. Now, consider two indices I and j to represent the lower and upper bounds of the elements to be sorted.



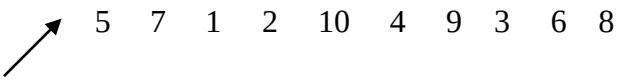
In the partitioning phase, move the elements less than or equal to pivot toward the left and those greater than or equal to toward the right.

The following are the steps for partitioning the array and keeping the pivot at correct place:

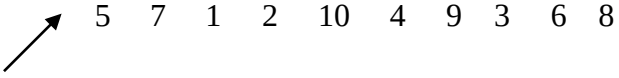
1. Select first element of array A as pivot.
2. Initialize i and j to first and last elements of the array respectively.
3. Increment i, until A[i]>pivot; stop
4. Decrement j, until A[j]<pivot; stop
5. If i<j, Exchange A[i] and A[j]
6. Repeat steps 3, 4, and 5 until i>j, i.e. when I and j cross each other.
7. Exchange the pivot element with the element pointed to by j which is correct place for pivot.

Note that in step 5, the swapping keeps the larger element to the right and smaller element to the left, relative to the pivot.

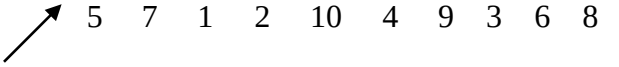
The sequence of steps indicating the movement of I and j for keeping the pivot is shown below.



pivot i j



pivot i j



pivot i j

5 7 1 2 10 4 9 3 6 8

pivot i j

5 7 1 2 10 4 9 3 6 8

pivot i j

Since i<=j swap A[i] and A[j] and repeat the process.

5 3 1 2 10 4 9 7 6 8

pivot i → j

5 3 1 2 10 4 9 7 6 8

pivot i → j

5 3 1 2 10 4 9 7 6 8

pivot i → j

5 3 1 2 10 4 9 7 6 8

pivot i j

5 3 1 2 10 4 9 7 6 8

pivot i ← j

5 3 1 2 10 4 9 7 6 8

pivot i j

5 3 1 2 10 4 9 7 6 8

pivot i j

Since i<=j, swap A[i] and A[j] and repeat the process.

5 3 1 2 4 10 9 7 6 8

pivot i j

5 3 1 2 4 10 9 7 6 8

pivot ij

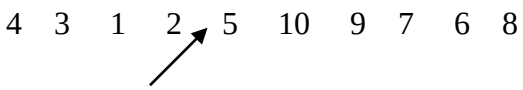
5 3 1 2 4 10 9 7 6 8

pivot ← ji

5 3 1 2 4 10 9 7 6 8

pivot j i

Since i>j, now swap the pivot with A[j].



pivot j i

At this stage the pivot 5 is in its proper place, i.e. elements left to it are less than or equal to 5 and elements to its right are greater than or equal to 5. Now the problem is break down into two smaller problems: sorting the array between start and j-1 and sorting the array between j+1 and end. The quick sort process can be applied to each of these sub arrays until all the elements are in sorted order.

[5 7 1 2 10 4 9 3 6 8]

[4 3 1 2] 5 [10 9 7 6 8]

[2 3 1] 4 5 [10 9 7 6 8]

[1] 2 [3] 4 5 [10 9 7 6 8]

1 2 [3] 4 5 [10 9 7 6 8]

1 2 3 4 5 [10 9 7 6 8]

1 2 3 4 5 [8 9 7 6] 10

1 2 3 4 5 [7 6] 8 [9] 10

1 2 3 4 5 6 [7] 8 [9] 10

1 2 3 4 5 6 7 8 [9] 10

1 2 3 4 5 6 7 8 9 10

The time analysis for Quick Sort is explained below:

The requirement to sort N elements using quick sort procedure involves 3 components (a) time taking for partitioning the array, which is roughly proportional to N, (b) time required for sorting lower sub array, and (c) time required for sorting upper sub array. Assume that there are K elements in the lower sub array.

Therefore,

$T(N) = T(K) + T(N-K-1) + cN$ where c is proportional constant.

In the worst case i.e when the array is already almost sorted, the array is always partitioned into two subarrays in which one of them is always empty. Thus for the worst case analysis,

$T(N) = T(N-1) + cN$ for $N > 1$

We can get the equations for next recursive calls by replacing N with N-1, N-2,.....2.

$T(N-1) = T(N-2) + c(N-1)$

$T(N-2) = T(N-3) + c(N-2)$

……..

……..

$T(2) = T(1) + c(2)$

$T(1) = T(0) + c(1) = c(1)$

Substituting these values in the first equation, we have

$$T(N) = c\sum_{i=1}^{N} = c.\ (N+1)N/2 = O(N^2)$$

The best case can be viewed when the partition takes place exactly at half of the original array.

Thus,

$$T(N) = 2\ T(N/2) + cN$$

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

Proceeding similarly as in worst case for recursive calls, we have

$$T(2)/2 = T(1)/1 + c$$

Assume the total number of elements N is a power of 2, such that $N = 2^M$ i.e. $M=\log_2 N$, there are M steps to add since the array is divided into exactly half. Therefore,

$$T(N)/N = T(1)/1 + c \log_2 N$$

$$T(N) = N + c\ N \log_2 N = O(N \log_2 N)$$

The analysis for average case performance is some what complicated, but it can be also analyzed as $O(N \log_2 N)$.

QuickSort is a good general purpose sorting approach that can be used for sorting most arrays. Its efficient is also relatively good. Because it uses element exchanges, it also requires relatively small amounts of memory.

In Mergesort, the file A(1:n) was divided at its midpoint into subfiles which were independently sorted and later merged.

In Quicksort, the division into two subfiles is made such that the sort subfiles do not need to be later merged.

This is accomplished by rearranging the elements in A(1:n) such that $A(i) \leq A(j)$ for all i between 1 and m and all j between m+1 and n for some m, $1 \leq m \leq n$. thus , the elements in A(1:m) and A(m+1:n) may be independently sorted. No merge is needed.

The rearrangement of the elements is accomplished by picking some element of A, say t=A(s) and then reordering the other elements so that all elements appearing before t in A(1:n) are less than or equal to t and all elements appearing after t are greater than or equal to t and all elements appearing after t are greater than or equal to t. This rearranging is referred to as portioning.

Procedure PARTITION accomplishes an in-place partitioning of the elements of A(M:p-1) . It is assumed that $A(R) \geq A(m)$ and that A(m0 is the portioning elements

if m=1 and p-1=n then A(n+1) must be defined and must be greater than or equal to those elements in A(1:n).

The procedure INTERCHANGE (x,y) performs the assignments:

temp←x; x←y;y←temp;

Algorithm: Partition the set A(m:p-1) about A(m)

```
procedure PARTITION(m, p)
//Within A(m), (m+1),…,A(p-1) the elements are
//rearranged in such a way that if initially t=A(m),
//then after completion A(q) = t, for some queue between m amd p-1
//A(k)≤t for m≤k<q and A(k)≥t for q<k<p
//The final value of p is q

integer m, p, i; global A(m:p)
v←A(m);
i←m;
```

```
loop
        loop i←i+1 until A(i)≥v repeat
        loop p←p-1 until A(p)≤v repeat
if i<p
then call INTERCHANGE(A(i), A(p))
else
exit
endif
repeat
A(m) ←A(p); A(p) ←v
end PARTITION
```

Algorithm: Sorting by partitioning

```
procedure QUICKSORT(p, q)
//sorts the elements A(p),……,A(q) which resides
//in the global array A(1:n) into ascending order
//A(n+1) is considered to be defined
//and must be≥ all elements in A(p:q); A(n+1) = +∞
integer p,q; global n, A(1:n)
if p<q
then j ← q+1
call PARTITION(p,j);
call QUICKSORT(p, j-1);
call QUICKSORT(j+1, q);
endif
end QUICKSORT
```

MERGE SORT

The merge sort algorithm closely follows the divide – and – conquer paradigm. The divide and conquer paradigm breaks the problem into several sub problems that are similar to the original problem but are smaller in size, solve the sub problems recursively and then combine these solutions to create a solution to the original problem.

The divide and conquer paradigm involves three steps at each level of the recursion.

(i) Divide the problem into number of sub problems.
(ii) Conquer the sub problems by solving them recursively. If the sub problems sizes are small enough, however, just solves the sub problems in a straight forward manner.
(iii) Combine the solutions to the sub problems into the solution for the original problem.

The merge sort algorithm closely follows the divide – and – conquer paradigm. Intuitively it operates as follows:

(i) **Divide:** Divide the n-element sequence to be sorted into two subsequences of n/2 elements each.
(ii) **Conquer:** Sort the two sub sequences recursively using merge sort.
(iii) **Combine:** Merge the two sorted sub sequences to produce the sorted order.

The key operation of the merge sort algorithm is the merging of two sorted sequences in the "combine" step. To perform the merging, we use auxiliary procedure MERGE(A,p,q,r). Where A is an array and p, q, r are indices numbering elements of the array such that p <= q < r. The procedure assumes that the sub array A[p..q] and A[q+1 .. r] are inserted order. It merges them to form a single sorted sub array that replaces the current sub array A[p..r].

MERGE-SORT(A,p,q)

1: If p<r

2: then q ← [(p+r)/2)

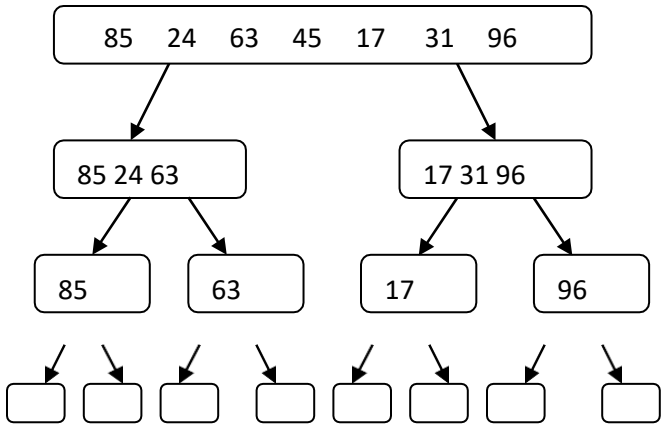
3: MERGE-SORT (A,p,q)

4: MERGE-SORT (A,q+1,r)

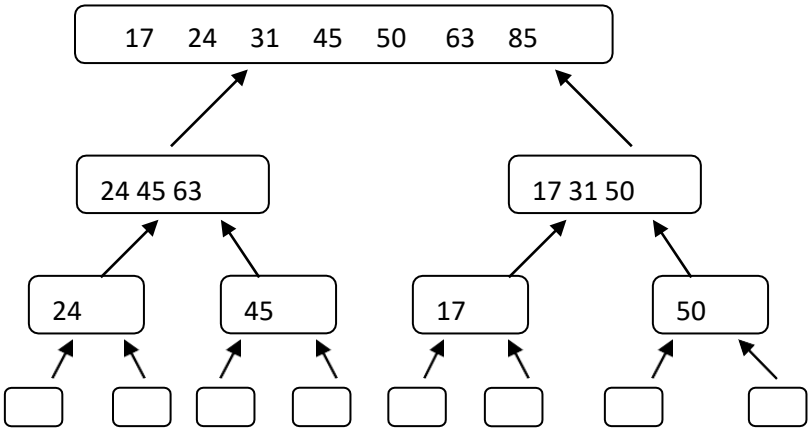
5: MERGE(A,p,q,r)

Example: Merge sort tree T for an execution of the merge-sort algorithm on a sequence with 8 elemen|s :

(a) input sequences processed at each node of T.



(b) Output sequences generated at each node of T.



* a sorting algorithm which has the nice property that in the worst case its complexity is

$O(n \log_2 n)$.

* Assume throughout that the elements are to be sorted in non decreasing order.

* Given a sequence of n elements A(!),…,A(n) the general idea is to imagine them split into two sets A(1)….A(n/2) and A(n/2 + 1),….,A(n).

* Each set is individually sorted and the resulting sequences are merged to produce a single sorted sequence of n elements.

Algorithm: Mergesort

procedure Mergesort(low, high)

//A(low:high) is a global array containing high-low+1≥0

//values which represent the elements to be sorted

integer low, high;

if low < high then

mid ← (low+high)/2;

call MERGESORT(low, mid); //sort one subset

call MERGESORT(mid+1, high) //sort the other subset

call MERGESORT(low, mid, high) //combine the results

endif

end MERGESORT

Algorithm: Merging two sorted sets using auxiliary storage.

procedure MERGE(low, mid, high)

//A(low:high) is a global array containing two sorted sub sets

// in A(low:mid) and in A(mid+1:high)

//The objective is to merge these sorted sets into

// a single sorted set residing in A(low:high). An auxiliary array B is used

integer h, i, j, k, low, mid, high; low≤mid<high

global A(low:high); local B(low:high);

h←low; i←low; j←mid+1;

while h≤mide and j≤high do

```
if A(h) ≤ A(j) then B(i) ←A(h); h←h+1;
                else B(i) ← A(j); j←j+1;
endif
i←i+1;
repeat
if h>mid then for k←j to high do
                B(i) ← A(k); i←i+1;
                repeat
        else for k ← h to mid do
                B(i) ←A(k); i←i+1;
                repeat
endif
for k←low to high do
A(k) ←B(k)
repeat
end MERGE
```

Example:

Consider the array of 10 elements.

A = { 310, 285, 179, 652, 351, 423, 861, 254, 450, 520}

procedure MERGESORT begins by splitting A into two sub files of size five.

The elements in A(1:5) are then split into two sub files of size three and two.

Then the items in A(1:3) are split into sub files of size two and one. The two values in A(1:2) are split a final time into one element sub files and now the merging begins.

Pictorially the file can now be viewed as

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

A(1) and A(2) are merged to yield

(285, 310|179|652, 351|425, 861, 254, 450, 520)

Then A(3) is merged with A(1:2) producing

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

Next, elements A(4) and A(5) are merged

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

followed by the merging of A(1:3) and A(4:5) to give

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

At this point the algorithm has returned to the first invocation of MERGESORT and it is about to process the second recursive call.

(17, 285, 310, 351, 652 | 423|861|254|450, 520)

A(6) and A(7) are merged and then A(8) is merged with A(6:7) giving

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

Next A(9) and A(10) are merged followed by A(6:8) and A(9:10)

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

At this point there are two sorted sub files and the final merge produces the fully sorted result.

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

Analysis:

If the time for the merging operation is proportional to n then the computing time for mergesort is described by the recurrence relation.

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

When n is a power of 2, $n=2^k$, we can solve this equation by successive substitutions, namely

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

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

$$= 4(2T(n/8)+cn/4)+2cn$$

$$= \ldots\ldots\ldots = 2^kT(1)+kcn$$

$$= an + cn \log n$$

If $2^k<n\leq 2^{k+1}$ then $T(n) \leq T(2^{k+1})$.

Therefore $T(n) = O(n\log_2 n)$

GREEDY METHOD

GENERAL METHOD

- Most of the problems have „n" inputs and require us to obtain a subset that satisfies some constraints. Any subset that satisfies these constraints is called a feasible solution.
- A feasible solution that either maximizes or minimizes a given objective function is called an optimal solution.
- The greedy method suggests that one can devise an algorithm which works in stages, considering one input at a time.
- At each stage, a decision is made whether input is in an optimal solution. This is done by considering the inputs in an order determined by some selection procedure.

Algorithm: Greedy Method Control Abstraction

```
procedure GREEDY(A, n)
//A(1:n) contains the n inputs

solution←Ø;
for i←1 to n do
x←SELECT(A)
if FEASIBLE(solution, x)
   then solution←UNION(solution, x)
endif
repeat
return (solution)
end GREEDY
```

The function SELECT selects an input from A and removes it. The selected input"s value is assigned to x. Feasible is a Boolean-valued 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 look, once a particular problem is chosen and the functions Select, Feasible, and Union are properly implemented.

In the greedy method there are three important activities:

1. A selection of solution from the given input domain is performed.
2. The feasibility of the solution is performed. And then all the feasible solutions are obtained.
3. From the set of feasible solutions, the particular solution that minimizes or maximizes the given objective function is obtained. Such a solution is called optimal solution.
4. For a algorithm that uses Greedy method works in stages. At each stage only one input is considered at a time. Based on this input it is decided whether particular input gives the optimal solution or not.

APPLICATIONS

JOB SEQUENCING WITH DEAD LINES

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

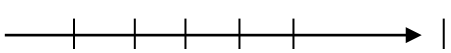
Example 1: Let n=4

n	pi	di
1	70	2
2	12	1
3	18	2
4	35	1

We will follow following rules to obtain the feasible solution.

- Each job takes one unit of time.
- If job starts before or at its deadline, profit is obtained, otherwise no profit.
- Goal is to schedule jobs to maximize the total profit.
- Consider all possible schedules and compute the minimum total time in the system.

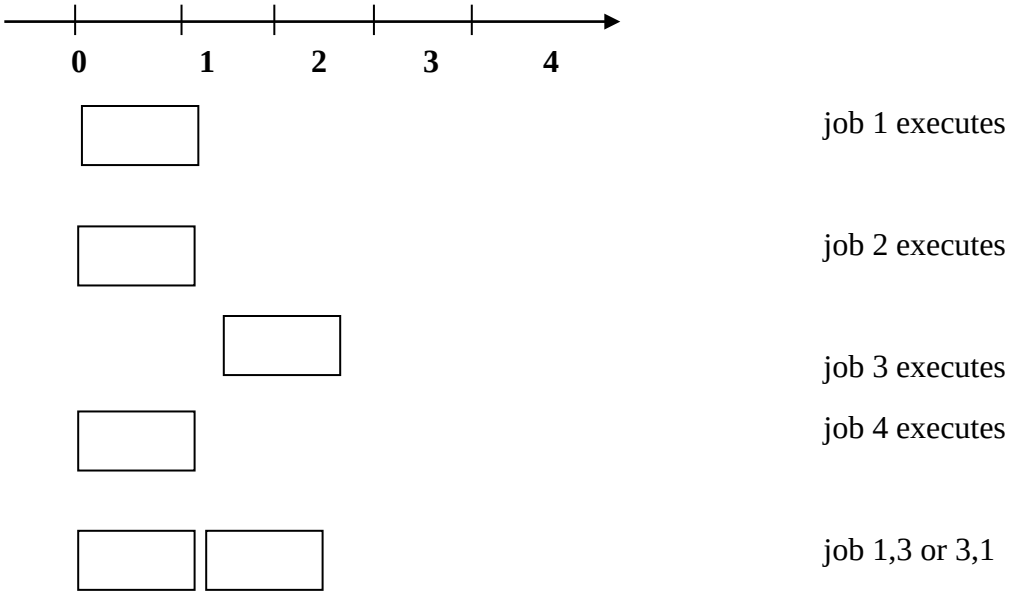
Consider the time line as

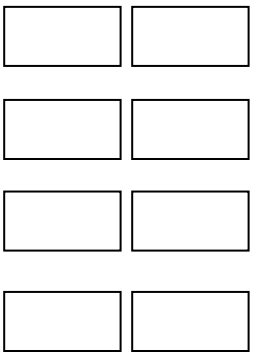


0 1 2 3 4 5

The feasible solutions are obtained by various permutations and combinations of jobs.

n	pi
1	70
2	12
3	18
4	35
1,3	88
2,1	82
2,3	30
3,1	88
4,1	105
4,3	53





job 2,1 executes

job 2,3 executes

job 4,1 executes

job 4,3 executes

Each job takes only one unit of time. Deadline of job means a time on which or before which the job has to be executed. The sequence {2, 4} is not allowed because both have deadline 1. If job 2 is started at 0 it will be completed on 1 but we can not start job 4 on 1 since deadline of job 4 is 1. The feasible sequence is a sequence that allows all jobs in a sequence to be executed within their deadlines and highest profit can be gained.

- Optimal solution is a feasible solution with maximum profit.
- In above example 3,2 is not considered as $d_3 > d_2$ but we have considered the sequence 2, 3 as feasible solution because $d_2 < d_3$
- We have chosen job 1 first then we have chosen job 4. The solution 4, 1 is feasible as the order of execution is 4 then 1. Also $d_4 < d_1$. If we try {1, 3, 4} then it is not a feasible solution, hence reject 3 from the set. Similarly if we add job 2 in the sequence then the sequence becomes {1, 2, 4}. This is also not a feasible solution hence reject it. Finally the feasible sequence is 4, 1. This sequence is optimum solution as well.

Example 2: Let n=4, $(p_1,p_2,p_3,p_4) = (100, 10, 15, 27)$ and $(d_1,d_2,d_3,d_4) = (2, 1, 2, 1)$. The feasible solutions and their values are:

	feasible solution	**processing sequence**	**value**
1.	(1, 2)	2, 1	110
2.	(1, 3)	1, 3 or 3, 1	115
3.	(1, 4)	4, 1	127
4.	(2, 3)	2, 3	25
5.	(3, 4)	4, 3	42
6.	(1)	1	100
7.	(2)	2	10
8.	(3)	3	15

9. (4) 4 27

Solution 3 is optimal. In this solution only jobs 1 and 4 are processed and the value is 127. These jobs must be processed in the order job 4 followed by job 1. Thus the processing of job 4 begins at time zero and that of job 1 is completed at time 2.

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

```
line procedure JS(D, J, n, k)
//D(i) ≥ 1, 1≤i≤n are the deadlines, n≥1. The jobs are ordered such that p1≥ p2,..... ≥pn. J(i) is the
ith job in the optimal solution, 1≤i≤k. Also at termination D(J(i)) ≤ D(J(i+1)), 1≤i<k
integer D(0:n), J(0:n)
D(0) ←J(0) ←0
k←1; J(1) ←1;
for i←2 to n do
r←k;
while D(J( r) ) > D(i) and D(J(r)) ≠ r do
r←r-1
repeat
if D(J( r )) ≤ D(i) and D(i) > r then
for l←k to r+1 by -1 do
J(l+1) ← J(l)
repeat
J(r+1) ← i;
k←k+1;
endif
repeat
end JS
```

The computing time taken by above algorithm is $O(n^2)$

0/1 KNAPSACK PROBLEM

The Knapsack problem can be stated as follows-

- Given n objects and a knapsack or bag.
- Object i has a weight w_i and the knapsack has a capacity m.
- If a fraction x_i, $0 \le x_i \le 1$, of object i is placed into the knapsack, then a profit of $p_i x_i$ is earned.
- The objective is to obtain a filling of the knapsack that maximizes the total profit earned. Since the knapsack capacity is m, the total weight of all chosen objects to be at most m.

The problem can be stated as

$$\text{maximize} \sum_{1}^{n} p_i x_i \quad \text{———} \quad (1)$$

$$\text{subject to} \sum_{1}^{n} w_i x_i \le m \quad \text{———} \quad (2)$$

$$\text{and } 0 \le x_i \le 1,\ 1 \le i \le n. \quad \text{———} \quad (3)$$

The profits and weights are positive numbers.

A feasible solution (filling) is any set $(x_1, \ldots x_n)$ satisfying eqn (2) and (3). An optimum solution is a feasible solution for which eqn (1) is maximized.

Example:

n	W_i	p_i
1	18	30
2	15	21
3	10	18

Also m = 20. Obtain the solution for the above given Knapsack problem.

Solution: The feasible solutions are given below

X_1	X_2	X_3
1/2	1/3	1/4
1	2/15	0
0	2/3	1
0	1	1/2

Let us compute $\sum w_i x_i$

1. 1 / 2 * 18 + 1/3*15 +1/4 * 10 = 16.5
2. 1*18 + 2/15 *15 +0*8 = 20
3. 0*8 + 2/3*15 +10 = 20
4. 0*18 + 1*15 +1/2*10 = 20

Let us compute $\sum p_i x_i$

1. 1 / 2 * 30 + 1/3*21 +1/4 * 18 = 26.5
2. 1*30 + 2/15 *21 +0*18 = 32.8
3. 0*30 + 2/3*21 +18 = 32
4. 0*30 + 1*21 +1/2*18 = 30

To summarize this

$\sum w_i x_i$	$\sum p_i x_i$
16.5	26.5
20	32.8
20	32
20	30

The solution 2 gives the maximum profit and hence it turns out to be optimum solution.

Algorithm GreedyKnapsack(m, n)

//p[1:n] and w[1:n] contain the profits and weights respectively of the n objects ordered such that //p[i]/w[i] ≥ p[i+1]/w[i+1]. m is the knapsack size and x[1:n] is the solution vector.

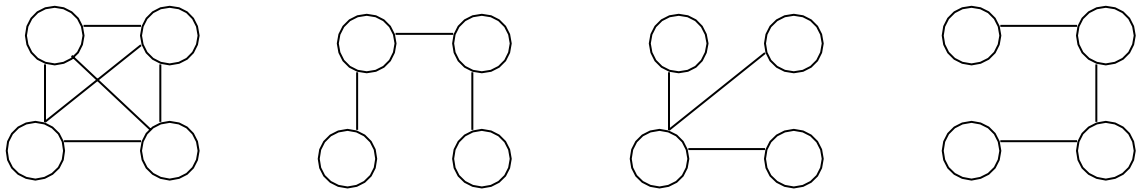
```
{
  for i:=1 to n do x[i] := 0.0;
  U := m;
```

```
 for i := 1 to n do
 {
        if (w[i] > U) then break;
        x[i] := 1.0;
        U := U-w[i];
   }
   if (i ≤ n) then x[i] := U/w[i];
}
```

MINIMU COST SPANNING TREES

Definition: Let G = (V, E) be an undirected connected graph. A sub graph t = (V, E‟) of G is a spanning tree of G iff t is a tree.

Example:

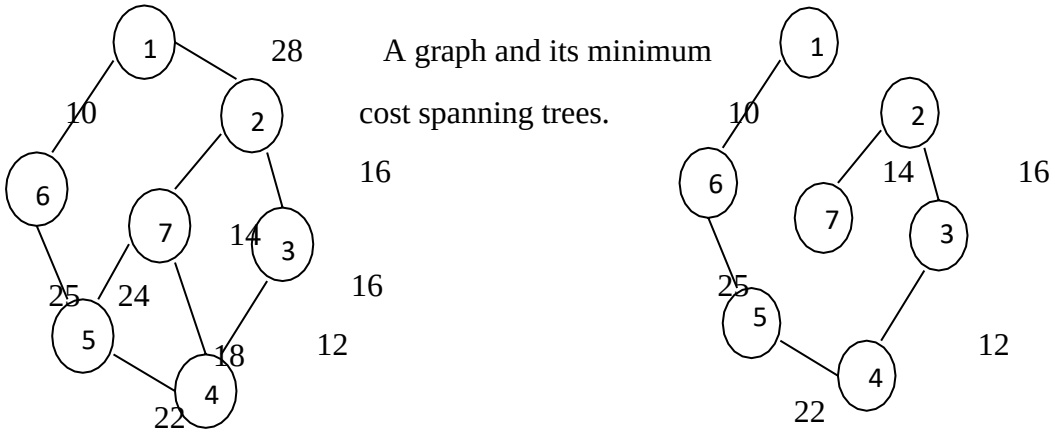


An undirected graph and three of its spanning trees.

Spanning trees have many applications. An application of spanning trees arises from the property that a spanning tree is a minimal sub graph G‟ of G such that V(G‟) = V(G) and G‟ is connected. (A minimal sub graph is one with the fewest number of edges). Any connected graph with n vertices must have at least n-1 edges and all connected graphs with n-1 edges are trees. If the nodes of G represent cities and the edges represent possible communication links connecting two cities, then the minimum number of links needed to connect the n cities is n-1. The spanning trees of G represent all feasible choices.

In practical, the edges have weights assigned to them. These weights may represent the cost of construction, the length of the link, and so on. Given such a weighted graph, one would then wish to select cities to have minimum total cost or minimum total length. In either case the links selected have to form a tree. If this is not so, then the selection of links contain a cycle. Removal

of any one of the links on this cycle results in a link selection of less cost connecting all cities. We are therefore interested in finding a spanning tree of G with minimum cost. The cost of a spanning tree is the sum of the costs of the edges in that tree.



There are two possible ways to interpret this criterion.

1. Prim‟s Algorithm
2. Kruskal‟s Algorithm

Prim's Algorithm

A greedy method to obtain a minimum cost spanning tree builds the tree edge by edge. The next edge to include is chosen according to some optimization criterion. The criterion to choose an edge that results in a minimum increase in the sum of the costs of the edges so far included. There are two possible ways to interpret this criterion. In the first, the set of edges so far selected from a tree. Thus, if A is the set of edges selected so far, then A forms a tree. The next edge (u, v) to be included in A is a minimum-cost edge not in A with the property that AU{(u, n)} is also a tree.

Let us obtain a pseudo code algorithm to find a minimum-cost spanning tree using this method. The algorithm will start with a tree that includes only a minimum cost edge of G. Then edges are added to this tree one by one. The next edge (i, j) to be added is such that i is a vertex already included in the tree, j is a vertex not yet included, and the cost of (i, j), cost[i, j] is minimum among all edges (k, l) such that vertex k is in the tree and vertex l is not in the tree. To determine this edge (i, j) efficiently, we associate with each vertex j not yet included in the tree a value near[j]. The value near[j] is a vertex in the tree such that cost[j, near[j]] is minimum among all choices for near[j]. We define near[j] = 0 for all vertices j that are already in the tree. The next

edge to include is defined by the vertex j such that near[j]≠0 (j not already in the tree) and cost[j, near[j]] is minimum.

Algorithm: line procedure PRIM(E, COST, n, T, mincost)

real COST(n, n), mincost;

integer NEAR(n), n, i, j, k, l, T(1:n-1, 2(;

(k,l) ←edge with minimu cost

mincost ← COST(k, l)

(T(1, 1), T(1, 2)) ← (k, l)

for i←1 to n do

if COST(i, l) < COST(i, k) then NEAR(i) ← l

else NEAR(i) ← k endif

repeat

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(1, 1), T(1, 2)) ← (j, 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

endif

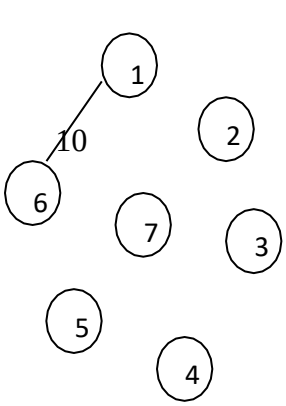
repeat

repeat

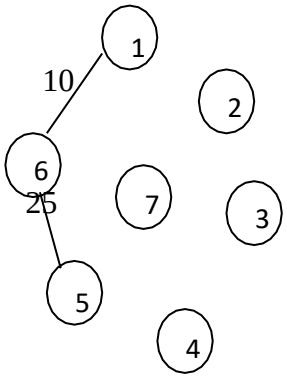
if mincost≥∞ then print (“no spanning tree”) endif

end PRIM

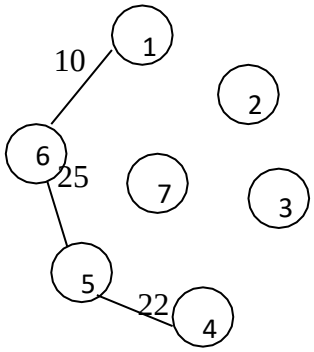
Stages in Prim's algorithm:



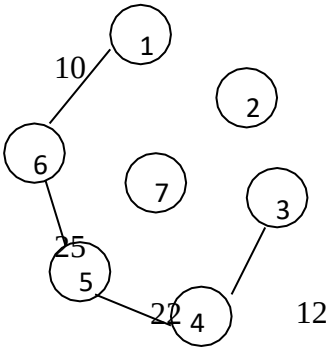
(a)



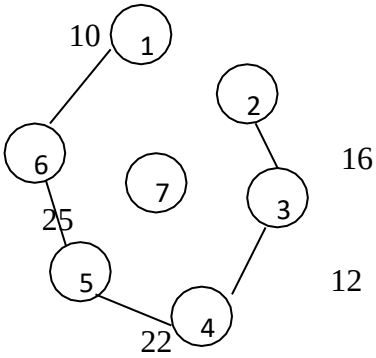
(b)



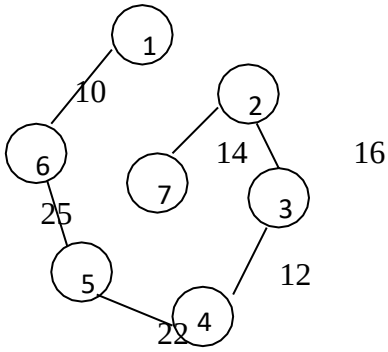
(c)



(d)



(e)



(f)

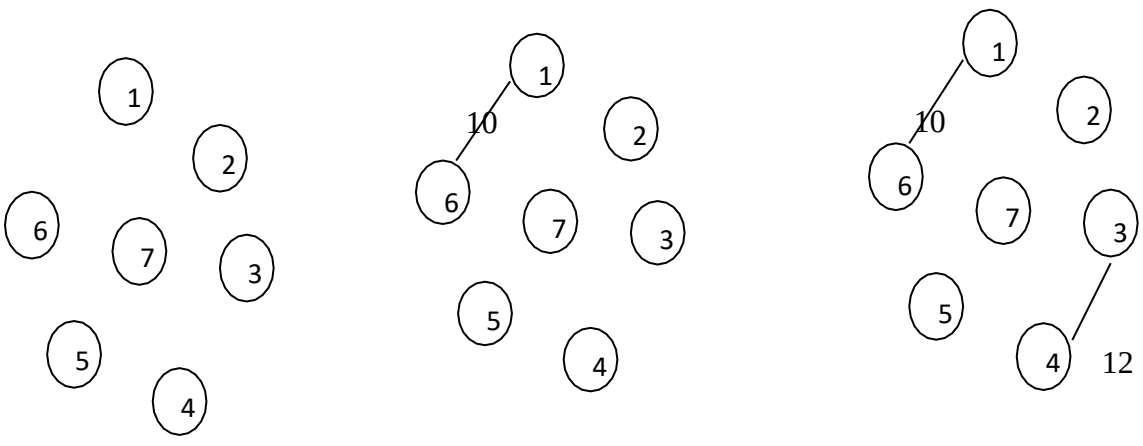
Kruskal's Algorithm

There is a second possible interpretation of optimization criteria mentioned earlier in which the edges of the graph are considered in non decreasing order of cost. This interpretation is that the set t of edges so far selected for the spanning tree be such that it is possible to complete t into a tree. Thus t may not be a tree at all stages in the algorithm. In fact, it will generally only be a forest since the set of edges t can be completed into a tree iff there are no cycles in t.

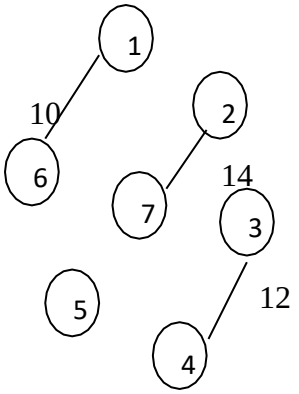
Algorithm: line procedure KRUSKAL(E, COST, n, T, mincost)

real mincost, COST(1:n, 1:n),

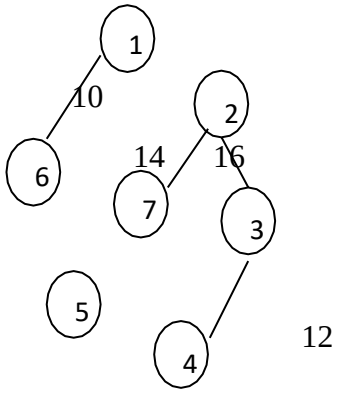
```
INTEGER parent(1:n), T(1:n-1, 2), n
construct a heap out of the edge costs using HEAPIFY
PARENT ← -1
i←mincost←0
while i<n-1 and heap not empty do
delete a minimum cost edge (u, v) from the heap and reheapify using ADJUST
j←FIND(u); k←FIND(v);
if j≠k then i←i+1
T(i, 1) ← u; T(1, 2) ←v
mincost←mincost+COST(u, v)
call UNION(j, k)
endif
repeat
if i≠n-1 then print (“no spanning tree”) endif
return
end KRUSKAL
```



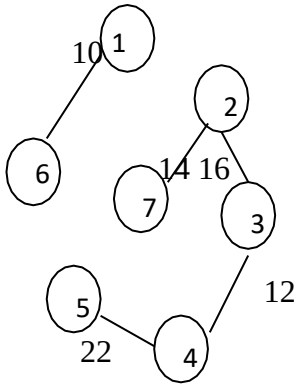
(a) **(b)** **(c)**



(d)



(e)



(f)

Analysis:

- Prim‟s algorithm spends most of its time in finding the smallest edge. So, time of the algorithm basically depends on how do we search this edge. Therefore Prim‟s algorithm runs in $O(n^2)$ time.
- The computing time of Kruskal‟s algorithm is $O(|E| \log |E|)$ where E is the number of edges.

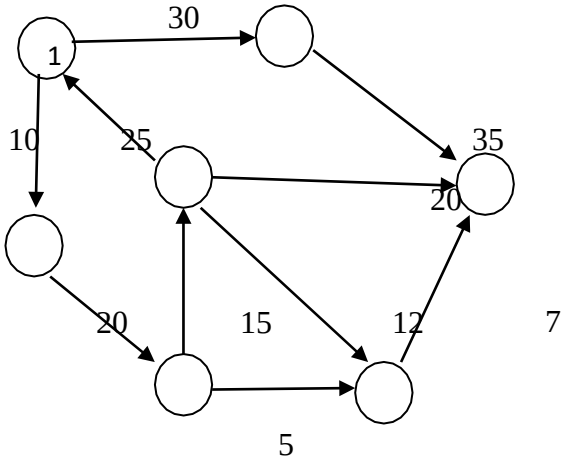
SINGLE SOURCE SHORTEST PATH PROBLEM

Many times, Graph is used to represent the distance between two cities. Everybody is often interested in moving from one city to another as quickly as possible. The single source shortest path is based on this.

In single source shortest path problem the shortest distance from a single vertex called source is obtained.

Let G(V, E) be a graph, then in single source shortest path the shortest paths from vertex v_0 to all remaining vertex is obtained. The vertex v_0 is then called as source and the last vertex is called destination. It is assumed that all the distances are positive.

Example: Consider a graph G as given below:



start from vertex 1. Hence set S[1] = 1.

Now shortest distance from vertex 1 is 10. i.e 1→2 = 10. Hence {1, 2} and min = 10.

From vertex 2 the next vertex chosen is 3.

{1, 2} = 10

{1, 3} = ∞

{1, 5} = ∞

{1, 6} = ∞

{1, 7} = ∞

Now

{1, 2, 3} = 30

{1, 2, 4} = ∞

{1, 2, 7} = ∞

{1, 2, 5} = ∞

{1, 2, 6} = ∞

Hence select 3.

S[3] = 1

{1, 2, 3, 4} = 45

{1, 2, 3, 5} = 35

{1, 2, 3, 6} = ∞

{1, 2, 3, 7} = ∞

Hence select next vertex as 5.

S[5] = 1

Now

{1, 2, 3, 5,6} = ∞

{1, 2, 3, 5,7} = 42

Hence vertex 7 will be selected. In single source shortest path if destination vertex is 7 then we have achieved shortest path 1 – 2 – 3 – 5 – 7 with path length 42. The single source shortest path from each vertex is summarized as below -

1, 2	10
1, 2, 3	30
1, 2, 3, 4	45
1, 2, 3, 5	35
1, 2, 3, 4, 7	65
1, 2, 3, 5, 7	42
1, 6	30
1, 6, 7	65

Algorithm ShortestPaths(v, cost, dist, n)

//dist[j], 1≤j≤n, is set to the length of the shortest path from vertex v to vertex j in a digraph G //with n vertices. dist[v] is set to zero. G is represented by its cost adjacency matrix cost [1:n, 1:n]

```
{
    for i:=1 to n do
    {
        S[i] := false; dist[i] := cost[v, i];
```

```
        }
        S[v] := true; dist[v] := 0.0; // Put v in S
        for num := 2 to n-1 do
        {
                //Determine n-1 paths from v.
                Choose u from among those vertices not in S such that dist[u] is minimum;
                S[u] := true; //Put u in S.
                for (each w adjacent to u with S[w] = false) do
                // Update distances.
                if (dist[w] > dist[u] + cost[u, w])) then
                dist[w] := dist[u] + cost[u, w];
        }
}
```