Dhiraj Bodake
1814216

Tutorial No-6

* Divide and Conquer.

General method :-

```
Algorithm DAnd (P)
{
    if small(P) the return S(P);
    else
    { divide P into smaller intances P1, P2, P3 ... Pk
      k >= 1;
      Apply DAnd to each other subproblems
      return combine (DAndC(P1), DAnd (P2) ....
          DAnd C (Pk)';
    }
}
```

Binary Search :-

```
Algorithm Bisrch (a, i, l, x)
// Given an array a[i:l] of elements i
// non decreasing order; 1 <= i < l determine
// whether x is present and if so return
// j such that x = a[j], else return 0

{ if (l=i) then
    { if (x = a[i]) then return i;
      else return 0;
    }
  else
    { mid := |(i+l)/2|;
      if (x = a[mid]) then return mid;
      else if ( x < a[mid] ) then
        return Bisrch (a, i, mid-1, x);
      else return Bisrch (a, mid+1, l, x);
    }
```

GoodLuck Page No.
Date

finding maximum & minimum :-

```
Algorithm MaxMin (i, j, max, min)
// a[1:n] is a global array. parameters i and
// j are integers, 1<=i ≤ j ≤ n. The
// effect is to set max and min to the
// largest and smallest values in a[i:j] resp.

{
    if (i=j) then max := min := a[i]; //small(P)
   else if (i=j-1) then //another case of small(P)
      {
         if (a[i] < a[j]) then
         {
            max := a[j]; min := a[i];
         }
         else
         {
            max := a[i]; min := a[j];
         }
      }
      else
      { //if p is not small, divide p into
         subproblems. //Find where to split
      //the set.
         mid := [(i+j)/2];
      //solve the subproblems.
         MaxMin(i, mid, max, min);
         MaxMin(mid+1, j, max1, min1);
      //combine the solutions.
         if (max < max1) then max := max1;
         if (min > min1) then min := min1;
      }
}
```

GoodLuck Page No.
Date

merge sort:-

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

Algorithm Merge (low, mid, high)
// a[low : high] is a global array containing
// two sorted subsets in a[low : mid] and in
// a[mid+1 : high]. The goal is to merge
// these two sets into a single set residing
// in a[low : high]. b[] is an auxillary
// global array.

{   h := low ; i := low ; j := mid+1;
    while ((h ≤ mid) and (j ≤ high)) do
    {   if (a[h] ≤ a[j]) then
        {   b[i] := a[h] ; h := h+1;
        }

```
else
{   b[i] := a[j];  j := j+1;
}
i := i+1;
}
if (h > mid) then
    for k := j to high do
    {  b[i] := a[k];  i := i+1;
    }
else
    for k := h to mid do
    {  b[i] := a[k];  i := i+1;
    }
for k := low to high do  a[k] := b[k];
}
```

Quick Sort :-

```
Algorithm QuickSort(p,q)
// sorts the elements a[p],......,a[q] which
// reside in the global array a[1:n]
// into ascending order : a[n+1] is
// considered to be defined and must
// be >= all the elements in a[1:n]
{
    if (p < q) then // if there are more than one
    {                                     element
      // divide p into two subproblems
        j := partition (a, p, q+1);
            // j is the position of the
            // partitioning element.
```

```
    // solve the subproblems
        QuickSort (p, j-1);
        QuickSort (j+1, q);
    // There is no need for combining solutions.
    }
}

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

        repeat
            j:= j-1;
        until ( a[j] ≤v);
        if (i<j) then Interchange (a, i, j);
    } until (i>,j)
    a[m] := a[j] ; a[j]:=v ; return j;
}

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

selection sort:-

```
Algorithm Select1 (a, n, k)
// selects the k^th smallest element in
// a[1:n] and places it in the k^th
// position of a[]. The remaining element
// are ~~it~~ rearranged such that a[m] ≤ a[k]
// for 1 ≤ m < k , and a[m] ≥ a[k]
// for k < m ≤ n.
{
    low := 1 ; up := n+1 ;
    a[n+1] := ∞;   // a[n+1] is set to infinity
    repeat
    {
      // Each time the loop is entered,
       // 1 ≤ low ≤ k ≤ up ≤ n+1.
      j := Partition (a, low, up);
           // j is such that a[j] is the
           // j^th smallest value in a[].
      if (k = j) then return;
      else if (k < j) then up := j;
         // j is the new upper limit.
       else low := j+1; // j+1 is the new
              // lower limit.
    } until (false);
}
```

GoodLuck Page No.
Date

* Greedy method

General method :-

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

container loading :-

```
void containerLoading (container * c,
    int capacity , int numberOfcontainers, int * x)
{
  // Greedy Algo. for container loading
     // set x[i] = 1 iff container i, i >= 1
    // is loaded.
     // sort into increasing order of weight.
  heapSort (c, numberOfContainers);
  int n = numberOfContainers;
  // initialize x
  for (int i=1 ; i<n ; i++)
       x[i] = 0;
  // select containers in order of weight
  for ( int i=1 ; i<=n && c[i].weight <= capacity; i++)
  { // enough capacity for
     // container c[i].id
```

x [c[i].id] = 1;
capacity -= c[i].weight
}

}

Knapsack :-

Algorithm knapsack (Array w, Array v, int m)

{

1. for j <- 1 to size(v)
2. calculate cost[i] <- v[i] / w[i]
3. sort - Descending (cost)
4. i ← 1
5. while (i <= size(v))
6. if w[i] <= M
7. M ← M - W[i]
8. ~~tot~~ total ← total + V[i];
9. if W[i] > M
10. i ← i+1;

Tree vertex splitting :-

Algorithm TVS (i, δ)

// Determine and output a minimum
// cardinality split set. The tree is
// realized using the sequential representation
// Root is a tree [1]. N is the largest
// number such that tree[N] has tree node

GoodLuck Page No.
Date

Algorithm TVS (T, δ)
// Determine and output the nodes to be split
// w() is the weighting function for edge.
```
{
    if (T ≠ 0) then
    {
        d[T] := 0;
        for each child v of T do
        {   TVS(v, δ)
            d[T] := max {d[T], d[v] + w(T, v)};
        }

        if ((T is not the root) and
                (d[T] + w(parent(T), T) > δ)) then
        {
            write (T); d[T] := 0;
        }
    }
}
```

Job sequencing with deadlines :-

Algorithm JS (d, j, n)
```
// d[i] >= 1, 1 <= i <= n are the deadlines,
// n >= 1, the jobs are ordered such that
// p[1] >= p[2] >= ..... >= p[n]. 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.
{
    d[0] := J[0] := 0;    // Initialize
```

GoodLuck | Page No.
Date

```
J[1] := 1; // include job 1
k := 1;
for {i := 2 to n do
{
  // consider jobs in non increasing order
  // of p[i]. find position for i and
  // check feasibility of 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
      {
        // insert i into J[].
        for q:=k to (r+1) step -1 do
            J[q+1] := J[q];
          J[r+1] := i;
          k := k+1;
      }
}
return k;
}
```

minimum cost spanning trees:-

Algorithm prim (E, cost, n, t)
// E is the set of edges in G. cost[1:n, 1:n]
// is the cost adjacency matrix of an n
// vertex graph such that cost[i, j] is
// either a positive real number or ∞
// if no edge (i, j) exists. A minimum

```
// spanning tree is computed and stored as
// a set of edges in the array
// t[1: n-1, 1:2]. (t[i,1], t[i,2]) is
// an edge in the minimum-cost spanning
// tree. The final cost is returned.
{
    Let (k,l) be a edge of minimum
     cost in E;
    min cost := cost [k,l];
    t[1,l] := 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
    {
    // find n-2 additional edges for t
      Let j be a an index such that near
      [j] ≠ 0 and cost [j, near[j]] is
      minimum;
      t[i,1] := j;   t[i,2] := near[j];
     min cost := mincost + cost [j, near [j]];
      near[j] := 0;
     for k := 1 to n do // update near.
        if ((near [k] ≠ 0) and (cost [k, near[k]]
              > cost [k, j]))
           then near [k] := j;
    }
    return mincost;
}
```

GoodLuck Page No.
Date

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

GoodLuck Page No.
Date

single source shortest path:-

```
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 diagraph G with n
// vertices. dist [v] is set of zero. G
// is represented by its cost adjacency
// matrix cost [1:n, 1:n]
{
  for i:=1 to n do
  { // initialize S
      S[i] := false; dist [i] := cost [v, i];
  }
  S[v] = true; dist [v] := 0.0; //put v in S.
  for num := 2 to n-1 do
  {
    // Determine n-1 paths from v.
    choose u from among those vertices
    not in S such that dist[u] is minimum;
    S[u] := true; // put u in S.
    for (each w adjacent to u with S[w]
          = false) do
        // update distances.
      if (dist [w] > dist [u] + cost [u, w]))
      then
          dist [w] := dist [u] + cost [u, w];
  }
}
```