Design & Analysis of Algorithm Lab
Code: CS591

Implement Binary Search using Divide and Conquer approach.

Algorithm BinSrch(a,i,l,x)

//Given an array a[i:l] of elements in nondecreasing order, 1≤i≤l, //determine whether x is present, and if so , return that x=a[j]; else return 0.

```
{
        if(l=i) then // If Small(P)
        {
                if(x=a[i]) then returni;
                else return 0;
        }
        else
        {// Reduce P into a smaller subproblem.
                mid:=(i+l)/2;
                if(x=a[mid])then return mid;
                else if(x<a[mid]) then
                                returnBinSrch(a,i,mid-1,x);
                        else returnBinSrch(a,mid+1,l,x);
        }
}
```

Implementing Merge Sort using Divide and Conquer approach

```
Algorithm MergeSort(low,high)
//a[low:high] is a global array to be sorted.
//Small(P) is true if there is ony one element to sort. In this case the list is already sorted.
{
        if(lowhigh) then // If there are ore 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 he soluations.
                        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[low:mid] and in a[mid+1:high]. The goal is to merge these two sets into a single set
//residing in a[low:high]. b[] is an auxiliary global array.
{
        h:=low; i:=low; j:=mid+1;
        while(h<=mid) and j<=high) do
        {
                if(a[h]<=a[j]) then
                {
                        b[i]:=a[h]; h:=h+1;
                }
                else
                {
                        b[i]:=a[j]:j:=j+1;
                }
                i:=i+1;
        }
        if(h>mid) then
                for k:=j to high do
                {
                        b[i]:=a[k]; i:=i+1;
```

```
                }
        else
                for k:=h to mid do
                {
                        b[i]:=a[k]; i:=i+1;
                }
        for k:=low to high do a[k]:=b[k];
}
```

<u>Implementing Quick Sort using Divide and Conquer approach</u>

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 sub problems.
                        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 needed for comining soluation.
        }
}
```

Algorithm Partition(a,m,p)

// Within a[m],a[m+1],….,a[p-1] the elements are rearranged in such a manner that if //initially t=a[m],then after completion a[q]=t for some q between m and p-1, a[k]<=t for //m<=k<q, and a[k]>=t for q<k<p. q is returned . Set a[p]=∞.

```
{
        v:=a[m]; i:=m; j:=p;
        repeat
        {
```

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

Algorthm Interchange(a,i,j)

//Exchange a[i] with a[j].

```
{
    p:=a[i];
    a[i]:=a[j]; a[j]:=p
}
```

Find Maximum and Minimum element from a array of integer using Divide and Conquer approach.

Algorithm MaxMin(a,i,j)

//a[i:j] is a global array. The effect is to set max and min to the largest and smallest values
//in a[i:j], respectively.

```
{
    if(i=j) then
    {
            max=min=a[i];
            return(max,min);
    }
    if(i=j-1) then
    {
```

```
            if(a[i]<a[j])
            max=a[j],min=a[i];
            else
            max=a[i],min=a[j];
            return(max,min);
            }
        else
        {
            mid:=(i+j)/2;
            max1,min1=MaxMin(a,i,mid);
            max2,min2=MaxMin(a,mid+1,j);
            if(max1<max2) then
                max=max2;
            else
                max=max1;
            if(min1<min2) then
                min=min1;
            else
                min=min2;
            return(max,min);
        }
    }
```

Implementing all pair of Shortest path for a graph(Floyed-Warshall Algorithm)

Algorithm All paths(cost,A,n)

//cost[1:n,1:n] is the cost adjacency matrix of a graph with vertices; A[I,j] is the cost of a
//shortest path from vertex I to vertex j. cost[I,i]=0.0, for 1<=i<=n.

```
{
        for i:=1 to n do
                for j:=1 to n do
                        A[I,j]:=cost[i,j]; // Copy cost into A.
        for k:=1 to n do
                for i:=1 to n do
                        for j:=1 to n do
                                A[i,j]:=min(A[i,j],A[i,k]+A[k,j]);
}
```

Implementing Matrix chain multiplication

Algorithm MCM(n,p)

// n=number of matrices

// p= dimension array

// m[1:n,1:n] is a cost matrix where m[1][n] is the minimum cost of matrix chain multiplication.

```
{
        for i:=1 to n do
                m[i][i]:=0;
        x:=2;
        for i:=1 to n-1 do
        {
                j:=x;
                while(j<=n)do
                {
```

```
                    m[i][j]:=∞  ;
                    J++;
            }
            x++;
    }
    x:=2;
    while(x<=n) do
    {
            i:=1;
            j:=x;
            while(i<=n) do
            {
                    if(j<=n) then
                    {
                            for k:=i to j-1 do
                            {
                                    m[i][j]:=min(m[i][j],m[i][k]+m[k+1][j]+p(i-1) p(k) p(j))
                            }
                            j++;
                    }
                    i++;
            }
            x++;
    }
return m[1][n];
}
```

Implementing Single Source shortest path for a graph(Dijkstra algorithm)

Algorithm ShortestPaths(v,cost,dist,n)

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

```
{
        for i:=1 to n do
        { // Initialize S.
                S[i]:=false; dist[i]:=cost[v,i];
        }
        S[v]:=true; dist[v];=0.0; // Put v in S.
        for num:=2 to n do
        {
                // Detrmine n-1 paths from v.
                Choose u from among those vertices not in  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];,
        }
}
```

Implementing Fractional Knapsack problem using Greedy Method

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]. Mis the knapsack size and x[1:n] is the soluation
//vector.

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

```
            if(w[i]>U) then break;
            x:=1.0; U:=U-w[i];
      }
      if(i<=n)then x[i]:=U/w[i];
}
```

Implementing Minimum Cost Spanning tree by Prim's Algorithm using Greedy Method

Algorithm Prim(E,cost,n,t)

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

```
{
      Let (k,l) be an edge of minimum cost in E;
      mincost:=cost[k,l];
      t[1,1]:=k; t[1,2]:=l;
      for i:=1 to n do //Initialize near.
            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 an index such tha near[j]≠0 and
            cost[j,near[j]] is minimum;
            t[i,1]:=j; t[i,2]:=near[j];
            mincost:=mincost+cost[j,near[j]];
            near[]:=0;
            for k:=1 to n do  // Update near[].
                  if((near[k]≠0) and (cost[k,neark]]>cost[k,j]))
                        then near[k]:=j;
      }
      return mincost;
}
```

Implementing Breadth First Search(BFS)

Algorithm BFS(v)

// A breadth first searh of G is carried out beginingat vertex v. For any node i , //visited[i]=1 if i has already been visited. The graph G and array visited[] are global; //visited[] is initialized to zero.

```
{
        u:=v;  //q is a queue of unexplored vertices.
        visited[v]:=1;
        repeat
        {
                for all vertices w adjacent from u do
                {
                        if(visited[w]=0) then
                        {
                                Add w to q;  / / w is unexplored
                                visited[w]:=1;
                        }
                }
                if q is empty then return;  // No unexplored vertex.
                Delete the next element, u , from  q;
                                                // Get first unexplored ertex.
        }until(false);
}
```