Computer Algorithm

Binary Search Algorithm:

Algorithm: Recursive binary search

```
Algorithm BinSrch (a,i,l,x)
//Given an array a[i:l] of elements in 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 small (P)
        {
                if (x = a[i]) then return i;
        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
                return BinSrch(a, i, mid - 1, x);
              else return BinSrch(a, mid + 1, l, x);
        }
}
```

Algorithm: Iterative binary search

```
Algorithm BinSrch (a, n, x)
//Given an array a[1 : n] of elements in non-decreasing order, n>=0, determine //whether x is
present and if so, return j such that x=a[j]; else return 0;
{
        low := 1; high := n;
        while (low<=high) do
         {
            mid := [(low+high)/2];
            if (x < a[mid]) then high := mid -1;
            else if (x > a[mid]) then low := mid + 1;
                    else return mid;
         }
         return 0;
}
```

MinMax Algorithm:

Algorithm: Recursively finding the maximum and minimum

Algorithm Minmax (*i, j, max, min*)
// *a*[1 : *n*] is a global array. Parameters I and j are intergers, 1<= *i* <= *j* <= *n*.
// 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];  // 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 subproblem.
          // Find where to split the set.
            mid := [(i + j) / 2];
         //Solve the subproblem
            MaxMin(i, mid, max, min) ;
            MaxMin( mid+1, j, max1, mini1) ;
         // Combine the solution
            if (max < max1) then max := max1;
            if (min < min1) then min:= min1;

        }
}
```

Merge Sort:
Algorithm: 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: Merging two sorted subarrays using auxiliary storage.

```
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 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];
}
```

Quick Sort:

Algorithm: Sorting by partitioning

```
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 := Partiton(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.
    }
}
```

Prim's Algorithm:
Prim's minimum cost spanning tree algorithm.

```
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
// infinity if no edge (i, j) exists. A minimum spanning tree is computed and
// sorted 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 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 that near[j] =|= 0 and
        cost[j, near[j]] is minimum;
        t[i, l] := j; t[i, 2] := near[j];
        mincost := 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;
}
```

Kruskal's Algorithm:
Kruskal's minimum cost spanning tree algorithm.

```
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 a different set.
    i := 0; mincost := 0.0;
    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[i,2] := v;
            mincost := mincost + cost[u, v];
            Union(j, k);
        }
    }
    if (i =|= n -1) then write ("No spanning tree);
    else return mincost;
}
```