Divide and Conquer

Unit 3

Binary Search (Recursive)

```
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 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);
    }
}
```

Binary Search (Non-Recursive)

```
Algorithm BinSearch(a, n, x)
// Given an array a[1 : n] of elements in nondecreasing
// 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;
}
```

Binary Search : Exercise

- Devise a “Binary” search algorithm that splits the set not into two sets of (almost) equal sizes but into two sets, one of which is twice the size of the other.
- Compare it with ‘Binary Search’ algorithm.

MaxMin (Non-Recursive)

```
Algorithm StraightMaxMin(a, n, max, min)
// Set max to the maximum and min to the minimum of a[1 : n].
{
    max := min := a[1];
    for i := 2 to n do
    {
        if (a[i] > max) then max := a[i];
        if (a[i] < min) then min := a[i];
    }
}
```

MaxMin(Recursive)

```
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], 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];
            }
        }
```

MaxMin(Recursive)

```
        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;
        }
}
```

MaxMin : Quiz 1

- What if we drop lines 7 to 17?

MaxMin : Quiz 1 answer

- The algorithm is correct.
- But, it will take double time approximately.
  - One more level in the binary tree.

MaxMin : Quiz

- There is an iterative algorithm for finding the maximum and minimum which, though not a divide-and-conquer-based algorithm, is probably more efficient than MaxMin.
- It works by comparing consecutive pairs of elements and then comparing the larger one with the current maximum and the smaller one with the current minimum.
- Write the algorithm completely, and analyze the number of comparisons it requires.

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);
    }
}
```

Merge

```
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;
    }
```

Merge

```
    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];
}
```

Merge Sort : Quiz

- Why is it necessary to have the auxiliary array in MergeSort? Give an example that shows why in-place merging is inefficient.

- Answer :

Assume two sub arrays like (3,4) (1,2)

Now if we do in place merging, according to the algorithm it will work out like

(3,4) (1,2) → (1,4) (3,2) → (1,3) (4,2)

Merge Sort : Quiz

- A sorting method is said to be *stable* if at the end of the method, identical elements occur in the same order as in the original unsorted set. Is merge sort a stable sorting method?
- Answer : Yes

Merge Sort : Quiz

- Write an algorithm to apply merge sort on a singly linked list.

Quick Sort

```
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);
```

Quick Sort

```
            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;
}
```

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.
    }
}
```

Quick Sort : Need for Randomization

- What if the elements are already in sorted order?

Quick Sort : Randomized

```
Algorithm RQuickSort(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 ((q − p) > 5) then
            Interchange(a, Random() mod (q − p + 1) + p, p);
        j := Partition(a, p, q + 1);
            // j is the position of the partitioning element.
        RQuickSort(p, j − 1);
        RQuickSort(j + 1, q);
    }
}
```

Quick Sort : Quiz

- Perform QuickSort on
  - 1, 1, 1, 1, 1, 1, 1
  - 5, 5, 8, 3, 4, 3, 2

Quick Sort : Quiz

- Is Quick Sort stable?

Quick Sort : Quiz

- Discuss the merits and demerits of altering the statement

  if(i<j) to *if(i<=j)*

- Simulate both algorithms on the data set
  - (5, 4, 3, 2, 5, 8, 9)

End.