Module 2

Divide and Conquer

Divide and Conquer

Divide and conquer strategy is as follows: divide the problem instance into two or more smaller instances of the same problem, solve the smaller instances recursively, and assemble the solutions to form a solution of the original instance. The recursion stops when an instance is reached which is too small to divide.

The divide-and-conquer design strategy involves the following steps:

1. **Divide** an instance of a problem into one or more smaller instances.

2. **Conquer (solve**) each of the smaller instances. Unless a smaller instance is sufficiently small, use recursion to do this.

3. If necessary, **combine** the solutions to the smaller instances to obtain the solution to the original instance.

Divide and Conquer : Control Abstraction

A control abstraction is a procedure whose flow of control is clear but whose primary operations are specified by other procedures whose precise meanings are left undefined. The control abstraction for divide and conquer technique is **DANDC(P)**, where P is the problem to be solved.

```
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));
    }
```

}

SMALL (P) is a Boolean valued function which determines whether the input size is small enough so that the answer can be computed without splitting. If this is so function 'S' is invoked otherwise, the problem 'p' into smaller sub problems. These sub problems p1, p2, . . . , pk are solved by recursive application of DANDC.

If the sizes of the two sub problems are approximately equal then the computing time of DANDC is: ⌈ g (n) T (n) = ⌊ 2 T(n/2)+ f (n) n small otherwise Where, T (n) is the time for DANDC on 'n' inputs g (n) is the time to complete the answer directly for small inputs and f (n) is the time for Divide and Combine.

(examples : check recursive solutions in pdf given by Anoop sir)

Divide and Conquer : Finding Maximum and Minimum

Straightforward maximum and minimum

```
1   Algorithm StraightMaxMin(a, n, max, min)
2   // Set max to the maximum and min to the minimum of a[1 : n].
3   {
4       max := min := a[1];
5       for i := 2 to n do
6       {
7           if (a[i] > max) then max := a[i];
8           if (a[i] < min) then min := a[i];
9       }
10  }
```

StraightMaxMin requires2(n - 1) element comparisons in the best, average, and worst case. An immediate improvement is possible by realizing that the comparisona ***[i] < min*** is necessary only when ***a[i] > max*** is false. Hence we can replace the contents of the for loop by

```
if (a[i] > max) then max := a[i];
else if (a[i] < min) then min := a[i];
```

Now the best case occurs when the elements are in increasing order. The number of element comparisons is n – 1. The worst case occurs when the elements are in decreasing order. In this case the number of element

comparisons is 2(n- 1). The average number of element comparisons is less than 2(n - 1).

A divide-and-conquer algorithm for this problem would proceed as follows: Let $P = (n, a[i], \ldots, a[j])$ denote an arbitrary instance of the problem. Here n is the number of elements in the list $a[i], \ldots, a[j]$ and we are interested in finding the maximum and minimum of this list. Let Small(P)/ be true when $n \leq 2$. In this case, the maximum and minimum are $a[i]$ if $n = 1$. If $n = 2$, the problem can be solved by making one comparison.

If the list has more than two elements, P has to be divided into smaller instances. For example, we might divide P into the two instances $P_1 = (\lfloor n/2 \rfloor, a[1], \ldots, a[\lfloor n/2 \rfloor])$ and $P_2 = (n - \lfloor n/2 \rfloor, a[\lfloor n/2 \rfloor + 1], \ldots, a[n])$. After having divided P into two smaller subproblems, we can solve them by recursively invoking the same divide-and-conquer algorithm. How can we combine the solutions for P_1 and P_2 to obtain a solution for P? If MAX(P) and MIN(P) are the maximum and minimum of the elements in P, then MAX(P) is the larger of MAX(P_1) and MAX(P_2). Also, MIN(P) is the smaller of MIN(P_1) and MIN(P_2).

Recursively finding the maximum and minimum

```
1    Algorithm MaxMin(i, j, max, min)
2    // a[1 : n] is a global array. Parameters i and j are integers,
3    // 1 ≤ i ≤ j ≤ n. The effect is to set max and min to the
4    // largest and smallest values in a[i : j], respectively.
5    {
6        if (i = j) then max := min := a[i]; // Small(P)
7        else if (i = j − 1) then  // Another case of Small(P)
8            {
9                if (a[i] < a[j]) then
10               {
11                   max := a[j]; min := a[i];
12               }
13               else
14               {
15                   max := a[i]; min := a[j];
16               }
17           }
18           else
19           {    // If P is not small, divide P into subproblems.
20               // Find where to split the set.
21                   mid := ⌊(i + j)/2⌋;
22               // Solve the subproblems.
23                   MaxMin(i, mid, max, min);
24                   MaxMin(mid + 1, j, max1, min1);
25               // Combine the solutions.
26                   if (max < max1) then max := max1;
27                   if (min > min1) then min := min1;
28           }
29   }
```

Now what is the number of element comparisons needed for MaxMin? If $T(n)$ represents this number, then the resulting recurrence relation is

$$T(n) = \begin{cases} T(\lceil n/2 \rceil) + T(\lceil n/2 \rceil) + 2 & n > 2 \\ 1 & n = 2 \\ 0 & n = 1 \end{cases}$$

When n is a power of two, $n = 2^k$ for some positive integer k, then

$$\begin{aligned} T(n) &= 2T(n/2) + 2 \\ &= 2(2T(n/4) + 2) + 2 \\ &= 4T(n/4) + 4 + 2 \\ &\vdots \\ &= 2^{k-1}T(2) + \textstyle\sum_{1 \le i \le k-1} 2^i \\ &= 2^{k-1} + 2^k - 2 = 3n/2 - 2 \end{aligned} \qquad (3.3)$$

Note that 3n/2-2 is the best, average, and worst-case number of comparisons when n is a power of two.

Compared with the 2n - 2 comparisons for the straightforward method, this is a saving of 25% in comparison.

But ,in terms of storage, MaxMin is worse than the straightforward algorithm because it requires stack space for i,j,max,min,max1, and min1.Given n elements, there will be $\lfloor \log 2n \rfloor$ +1 levels of recursion and we need to save seven values for each recursive call.

Thus, MaxMin will be slower than StraightMaxMin because of the overhead of stacking i,j,max,and min for the recursion

In divide and conquer technique if the assumption is not true, the technique yields a less-efficient algorithm.Thus the divide-and-conquer strategy is seen to be only a guide to better algorithm design which may not always succeed.

BINARY SEARCH

In divide-and-conquer terminology, Binary Search locates a key x in a sorted (nondecreasing order) array by first comparing x with the middle item of the array.

If x equals the middle item, quit. Otherwise:

1. Divide the array into two subarrays about half as large. If x is smaller than the middle item, choose the left subarray. If x is larger than the middle item, choose the right subarray.

2. Conquer (solve) the subarray by determining whether x is in that subarray. Unless the subarray is sufficiently small, use recursion to do this.

3. Obtain the solution to the array from the solution to the subarray.

Example 2.1

Suppose $x = 18$ and we have the following array:

10 12 13 14 18 20 25 27 30 35 40 45 47.

↑

Middle item

1. Divide the array: Because $x < 25$, we need to search

10 12 13 14 18 20.

2. Conquer the subarray by determining whether x is in the subarray. This is accomplished by recursively dividing the subarray. The solution is:

Yes, x is in the subarray.

3. Obtain the solution to the array from the solution to the subarray:

Yes, x is in the array.

When developing a recursive algorithm for a problem, we need to

• Develop a way to obtain the solution to an instance from the solution to one or more smaller instances.

• Determine the terminal condition(s) that the smaller instance(s) is (are) approaching.

• Determine the solution in the case of the terminal condition(s).

Binary Search (Recursive)

Problem: Determine whether *x* is in the sorted array *S* of size *n*.

Inputs: positive integer *n*, sorted (nondecreasing order) array of keys *S* indexed from 1 to *n*, a key *x*.

Outputs: *location*, the location of *x* in *S* (0 if *x* is not in *S*).

```
index location (index low, index high)
{
  index mid;

  if (low > high)
    return 0;
  else {
    mid = ⌊(low + high)/2⌋;
    if (x == S[mid])
      return mid
    else if (x < S[mid])
      return location(low, mid - 1);
    else
      return location(mid + 1, high);
  }
}
```

Both will have the same time complexity O(log(n)), but they will different in term of space usage.
Recursive May reach to "log(n)" space (because of the stack), in iterative BS it should be "O(1)" space complexity.

(For Strassen's Matrix Multiplication ,Quick Sort, Merge Sort refer Anoop sir's pdf)