Unit 11 Divide and Conquer

Structure:

11.1 Introduction

Given a function to compute n inputs, the divide-and-conquer strategy suggests splitting the inputs into K distinct subsets, $1< K \leq n$ yielding K subproblems. These subproblems must be solved and then a method must be found to combine subsolutions into a solution as the whole. If the subproblems are still relatively large, then the divide-and-conquer strategy can be possibly re-applied.

Often, the subproblems resulting from a divide-and-conquer design are of the same type as the original problem. For those cases, the reapplication of the divide-and-conquer principle is naturally expressed by a recursive algorithm. Now, smaller and smaller subproblems of the same kind are generated until eventually subproblems that are small enough to be solved without splitting is produced.

Objectives:

After studying this unit, you should be able to:

- apply divide and conquer strategy to practical problems
- analyze the binary search algorithm and compute the time for successful and unsuccessful search
- analyze Max and Min algorithm using divide-and-conquer method
- analyze the time for the Merge Sort Algorithm
- use the working of Quick Sort Algorithm.

11.2 Divide and Conquer Strategy

Given a function to compute on n inputs, the divide-and-conquer strategy suggests splitting the inputs into *K* distinct subsets, $1<K<n$, yielding *K* subproblems. These subproblems must be solved and then a method must be found to combine subsolutions into a solution as a whole. If the subproblems are still relatively large, then the divide-and-conquer strategy can be possibly reapplied.

We can write a control abstraction that mirrors the way an algorithm based on divide-and-conquer will look. By a control abstractions we mean a procedure whose flow of control is clear but whose primary operations are specified by other procedures whose precise meanings are left undefined.

Algorithm 11.1

```
 1.procedure DANDC (p,q)
 2.global integer n, A(1:n)
 3.integer p,q; //1≤ p ≤ q ≤ n//
 4.local integer m;
 5.if small (p,q)
 6.then return (G(p,q))
 7.else m ← divide (p,q) //p ≤ m < q
 8.return (combine(DANDC(p,m), DANDC(m + 1,q)))
 9.endif
10.end DANDC
```

In Algorithm 11.1, procedure is called by DANDC *(1,n)*

- Small *(p,q)* is a Boolean function and will return true or false. True if input size is small enough that the answer can be computed without splitting. If so, function *G* is invoked.
- Divide *(p,q),* divides into *(p, m)* and *(m+1, q)*
- Combine function combines the results of *p,m* and *m+1, q*. Combine is a function that determines the solution to *p* using the solutions to the *K* sub problems.

If the size of *p* is *n* and the sizes of the *K* subproblems are n_1, n_2…….., n_k respectively, then the computing time of DANDC is described by the recurrence relation.

$$T(n) = \begin{cases} g(n) & \text{if } n \text{ is small} \\ T(n_1) + T(n_2) + ... + T(n_k) + f(n) & \text{otherwise} \end{cases}$$

$$\text{i.e. } T(n) = \begin{cases} g(n) & \text{if } n \text{ small} \\ 2T(n/2) + f(n) & \text{otherwise} \end{cases}$$

Where, T(n) is the time for DANDC on any input of size n & g (n) is the time to compute the answer directly for small inputs. The function f(n) is the time for dividing P and combining the solutions to subproblems.

The complexity of many divide-and-conquer algorithms is given by recurrences of the form.

$$T(n) = \begin{cases} T(1) & n = 1 \\ aT(n/b) + f(n) & n > 1 \end{cases}$$

Where, a and b are known constants. We assume that *T(1)* is known and *n* is a power of *b* (i.e. $n = b^k$)

Self Assessment Question

1. The complexity of many divide-and-conquer algorithms is given by recurrences of the form ——————.

11.3 Binary Search

Let a_i, $1 \le i \le n$ be a list of elements that are sorted in non-decreasing order. Consider the problem of determining whether a given element *x* is present in the list. If *x* is present we are to determine a value *j* such that $a_j = x$. If *x* is not present in the list, then *j* is to be set to zero.

Let $P = (n, a_i, \ldots.., a_1, x)$ denote an arbitrary instance of this search problem, where,

n is the number of element in the list, a_i ……….. a_1 is the list of elements and

x is the element searched for.

We use divide-and-conquer strategy to solve this problem. Let small *(p, q)* be true for *n = 1.*

In this case, *G(p, q)* will take the value *i* if $x = a_i$, otherwise it will take the value *0.*

Then, $g(1) = \theta\ (1)$, if P has more than one element, it can be divided (or reduced) into a new subproblem as follows:

Pick an index q in the range $[i, l]$ and compare x with a_q. There are 3 possibilities.

1) $x = a_q$: In this case, the problem P is immediately stored.
2) $x < a_q$: In this case, x has to be searched in the sublist $(a_i, a_{i+1}, \ldots, a_{q-1})$

   $\therefore$ P reduces to $(q - i, a_1, a_{q-1}, x)$
3) $x > a_q$: In this case, x has to be searched in the sublist $(a_{q+1}, \ldots\ldots , a_l)$

   $\therefore$ P reduces to $(l - q, a_{q+1}, \ldots\ldots, a_l, x)$

Any given problem, P gets divided (reduced) into one new subproblem. This division takes only $\theta(1)$ time. After a comparison with a_q, the instance remaining to be solved (if any) can be solved by using divide-and-conquer rule.

If q is always chosen such that a_q is the middle element (i.e. $q = \lfloor (n+1)/ 2 \rfloor$), then, the resulting search algorithm is known as binary search.

Algorithm 11.2 shows how binary search works for n elements where $n \geq 0$.

Algorithm 11.2

```
1.  procedure BINSRCH (x, j)
2.  integer low, mid, high, x,j;
3.  low ← 1, high ← n;
4.  while low ≤ high do
5.  mid ← ⌊(low + high)/2⌋
6.  case
7.  :x<A(mid):high ← mid — 1
8.  :x>A(mid): low ← mid + 1
9.  :else:j ← mid; return
10. end case
11. repeat
12. j ← o
13. endBINSRCH
```

Suppose there are *n* elements, then, for successful search $\rightarrow$ there is *n* comparisons and for unsuccessful search $\rightarrow$ there is $n + 1$ comparisons.

To determine the average behavior, we need to look more closely at the binary decision tree and equate its size to the number of element comparison in the algorithm.

- The distance of a node from the root is one less than its level.
- The internal path length *I* is the sum of the distances of all internal nodes from the root.
- The external path length *E* is the sum of the distances of all external nodes from the root.

Let *s(n)* = average number of comparisons in a successful search.
u(n) = average number of comparisons in an unsuccessful search.

For successful search
Total distance of all node to the root = *I*

Average distance of a node from root = $\frac{I}{n}$

$\therefore$ Average number of element comparison for successful search = $\frac{I}{n}+1$

$$\boxed{S(n)=1+\frac{I}{n}} \text{ ———————— } (1)$$

For unsuccessful search
Total distance from node = *E*
Average distance of any node from root

$$\boxed{u(n)=\frac{E}{(n+1)}} \text{ ———————— } (2)$$

$E \,\alpha (n + 1)$

In other words, $E \,\alpha n(n + 1)$

Substituting in (2) $\boxed{u(n)\,\alpha\, log(n)}$

Self Assessment Question

2. Average number of element comparison for successful search ————.

11.4 Max. and Min.

The problem is to find the maximum and minimum items in a set of *n* elements.

Algorithm 11.3

```
1.  procedure SMAXMIN(max, min)
2.  integer i, max, min;
3.  max ← min ← $A(1)$;
4.  for $I$ ←$2$ to $n$ do
5.  if $A(i)>max$ then max ← A(i)
6.  else if $A(i)<min$ then $min ← A(i)$
7.  endif
8.  endif
9.  repeat
10. end SMAXMIN
```

Best Case: If we have *n* element, *(n – 1)* element comparison.

Worst Case: If *n* elements are in descending order, *2* comparisons will take place.

$\therefore$We have *2(n – 1)* element comparison.

Average Case: $\frac{3}{2}(n-1)$ element comparison

Algorithm 11.4 MaxMin is a recursive algorithm that finds the maximum and the minimum of set of elements *{a(i),a(i+1)….,a(j)}.* The situation of set sizes one *(i = j)* and two *(i = j – 1)* are handled separately. For sets containing more than two elements, the midpoint is determined (just as in binary search) and two new subprograms are generated. When the maxima and minima of these subproblems are determined, the two maxima are compared and two minima are compared to achieve the solution for the entire set.

Algorithm 11.4

```
    Algorithm MaxMin (i,j, max, min)
    // a[1:n] is a global array Parameter i and j are integers,
    // $1 \le i \le j < n$. 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 subproblems
        // Find where to split the set
        mid:= $\lfloor (i+j)/2 \rfloor$;
        //Solve the subproblems
        Maxmin (i, mid, max, min);
        MaxMin (mid+i, j, max1, min1);
        // Combine the solutions
        if (max<max 1) then max:=max1;
        if(min>min1) then min:=min1;
      }
  }
```

Algorithm 11.4 Recursively finds the maximum and minimum

The procedure is initially invoked by the statement MaxMin*(1,n,x,y)*

Suppose, we simulate MaxMin on the following nine elements:

A:	*[1]*	*[2]*	*[3]*	*[4]*	*[5]*	*[6]*	*[7]*	*[8]*	*[9]*
	22	*13*	*–5*	*–8*	*15*	*60*	*17*	*31*	*47*

A good way of keeping track of recursive calls is to build a tree by adding a node each time a new call is made. For this algorithm, each node has four items of information: i,j, max and min. On the array *a [ ]* above, the tree of Fig. 11.1 is produced.

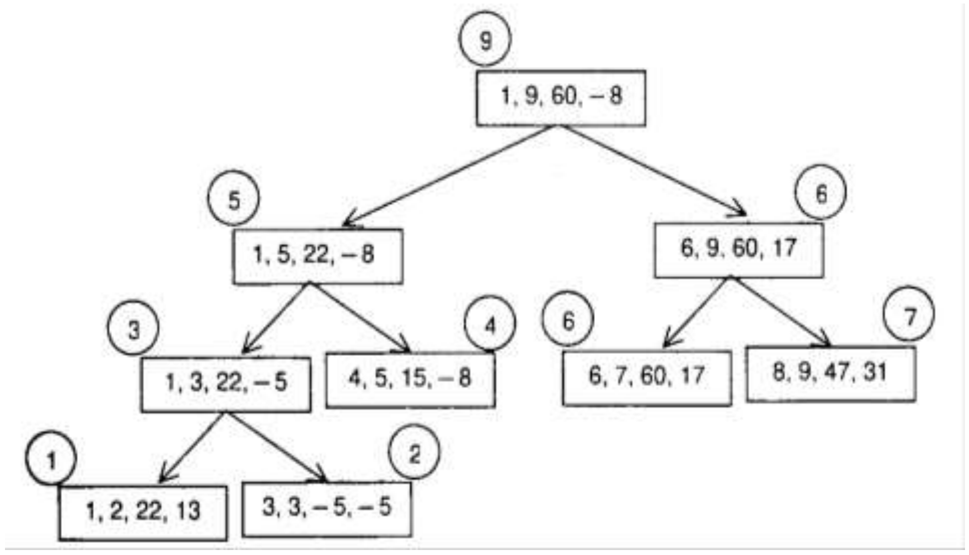

Fig. 11.1: Trees of recursive calls of MaxMin

Examining Fig 11.1, we see that the root node contain

s 1 and 9 as the values of i and j corresponding to the initial call to MaxMin. This execution produces two new calls to MaxMin, where i and j have the values 1, 5 and 6, 9 respectively and thus split the set into two subsets of approximately the same size. From the tree we can immediately see that the maximum depth of recursion is four (including the first call). The circled numbers in the upper left corner of each node represent the orders in which max and min are assigned values.

If (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

$$\begin{aligned} T(n) &= 2\cdot T\left(\frac{n}{2}\right)+2 \\ &= 2\left(2T\left(\frac{n}{4}\right)+2\right)+2 \\ &= 4T\left(\frac{n}{4}\right)+4+2 \\ &\quad \cdots \\ &\quad \cdots\cdot \\ &= 2^{k-1}T(2)+\sum_{1\le i\le k-1}2^i \\ &= 2^{k-1}+2^k-2=\frac{3n}{2}-2 \end{aligned}$$

Note that $\frac{3n}{2}-2$ is the best, average and worst-case number of comparisons when n is a power of two.

Self Assessment Question

3. A good way of keeping track of ———— is to build a tree by adding a node each time a new call is made.

11.5 Merge Sort

We assume throughout that the elements are to be sorted in non-decreasing order. Given a sequence of *n* elements (also called keys) *a[1], ……., a[n],* the general idea is to imagine them split into two sets *a[1], …….. a[$\lfloor n/2 \rfloor$]* and *a[$\lfloor n/2 \rfloor$+1],………..,a[n]*. Each set is individually sorted and the resulting sorted sequences are merged to produce a single sorted sequence of n elements.

Thus, we have another ideal example of the divide-and-conquer strategy in which the splitting is into two equal-sized sets and the combining operation is the merging of two sorted sets into one.

Mergesort (Algorithm 11.5) describes this process very succinctly using recursion and a function Merge (Algorithm 11.6) which merges two sorted sets. Before executing MergeSort, the *n* elements should be placed in a *[1:n]*. Then MergeSort *(1, n)* causes the keys to be rearranged into the non-decreasing order in a

Algorithm 11.5

```
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:= $\lfloor (low+high)/2 \rfloor$
        //Solve the subproblems
            MergeSort (low,mid);
            MergeSort(mid+1, high);
        //combine the solutions
            Merge(low, mid, high)
        }
}
```

Algorithms 11.6

```
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];
  }
```

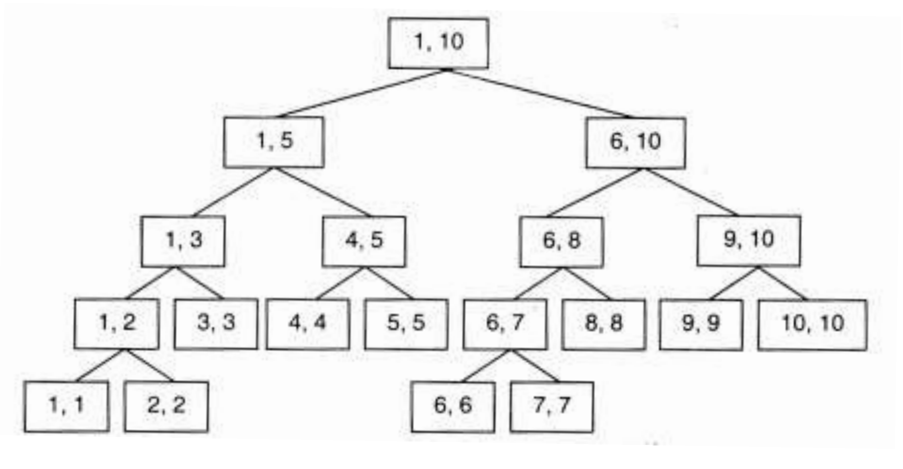

Fig. 11.2: Tree of calls of MergeSort (1, 10)

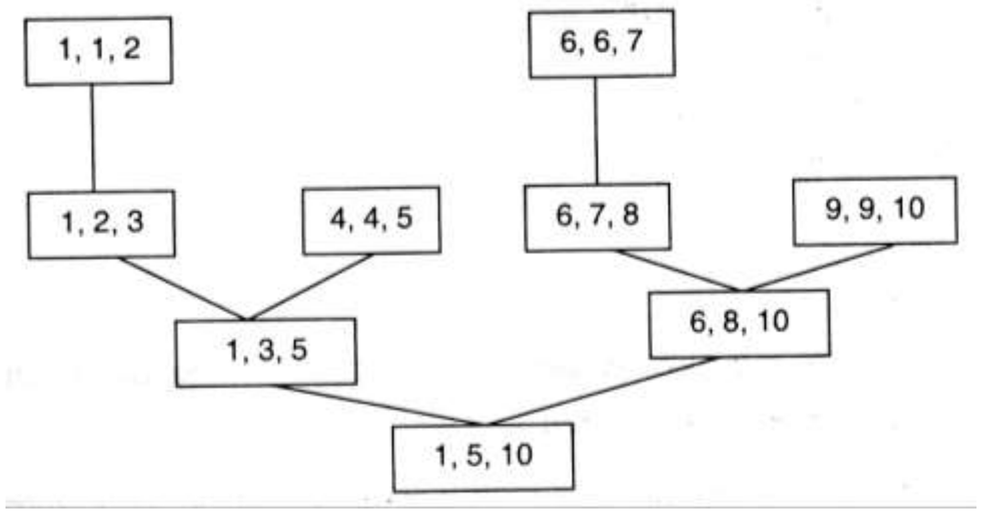

Fig. 11.3: Tree of calls of Merge

Fig. 11.2 is a tree that represents the sequence of recursive calls that are produced by MergeSort when it is applied to ten elements. The pair of values in each node are the values of the parameters low and high. Notice how the Splitting continues until sets containing a single element are produced. Fig. 11.3 is a tree representing the calls to produce Merge by MergeSort.

For example, the node containing *1, 2* and *3* represent the merging of *[1:2]* with *a[3]*.

If the time for the merging operation is proportional to 'n', then the computing for Mergesort is described by the recurrence relation.

$$T(n) = \begin{cases} a & n = 1, a \text{ is constant} \\ aT(n/2) + cn & n < 1, c \text{ is a constant} \end{cases}$$

When n is a power of 2, $n=2^k$, we can solve this equation by successive substitutions

$$T(n) = 2\left(2T\left(\frac{n}{4}\right) + c\frac{n}{2}\right) + cn$$

$$= 4T\left(\frac{n}{4}\right) + 2cn$$

$$= 4\left(2T\left(\frac{n}{8}\right) + \frac{cn}{4}\right) + 2cn$$

……

…….

$$= 2^k T(1) + kcn$$
$$= an + cn \log n$$

It is easy to see that if $2^k < n \leq 2^{k+1}$, then

$$T(n) \leq T(2^{k+1})$$

Therefore, $\boxed{T(n) = O(n \log n)}$

Self Assessment Question

4. If the time for the merging operation is proportional to 'n', then the computing for Mergesort is described by the recurrence relation ———.

11.6 Quick Sort

In merge sort, the file *a[1:n]* was divided at its midpoint into subarrays which were independently sorted and later merged.

In quick sort, the division into two subarrays is made so that the sorted subarrays do not need to be merged later. This is accomplished by rearranging the elements in a *[1:n]* such that $a[i] \leq a[j]$ for all *i* between *1* and *m* and all *j* between *m+1* and *n* for some $m, 1 \leq m \leq n$. Thus, the elements in a *[1:m]* and *a[m+1:n]* can be independently sorted. No merge is needed.

The rearrangement of the elements is accomplished by picking some element of *a[ ]*, say, *t=a[s]* and then reordering the other elements so that all elements appearing before *t* in *a[1:n]* are less than or equal to *t* and all elements appearing after *t* are greater than or equal to *t*. This rearranging is referred to as partitioning.

Function Partition of Algorithm 11.7 accomplishes an in-place partitioning of the elements of *a[m:p – 1]*. It is assumed that $a[p] \geq a[m]$ and that *a[m]* is the partitioning element. If *m = 1* and *p – 1 = n*, then *a[n + 1]* must be defined and must be greater than or equal to all elements in *a[1:n]*. The assumption that *a[m]* is the partition element is merely for convenience; other choices for the partitioning element than the first item in the set are better in practice. The function, interchange *(a, i, j)* exchanges *a[i]* with *a [j]*.

Algorithm 11.7

```
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]≤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;
}
```

Algorithm 11.8

```
Algorithm QuickSort(p,q)
//Sorts the elements a[p],……,a[q] which reside in
//global array a[1:n] into ascending order; a[n+1] is
//considered to be defined and must be $\geq$ 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.
}
}
```

In analyzing QuickSort, we count only the number of element comparisons $C(n)$. It is easy to see that the frequency count of other operations is of the same order as $C(n)$. We make the following assumptions: the n elements to be sorted are distinct, and the input distribution is such that the partition element $v=a[m]$ in the call to Partition (a,m,p) has an equal probability of being the i^{th} smallest element, $1 \leq i \leq p-m$, in $a[m:p-1]$.

First, let us obtain the worst-case value $Cw(n)$ of $C(n)$. The number of element comparisons in each call of Partition is at most $p - m + 1$. Let r be the total number of elements in all the calls to Partition at any level of recursion. At level one only one call, Partition $(a, 1, n+1)$, is made and $r = n$; at level two at most two calls are made and $r = n - 1$; and so on. At each level of recursion, $O(r)$ element comparisons are made by Partition. At each level, r is atleast one less than the r at the previous level, as the partitioning elements of the previous level are eliminated. Hence, $C_w(n)$ is the sum on r, as r varies from 2 to n, or $O(n^2)$.

The average value $C_A(n)$ of $C(n)$ is much less than $C_w(n)$.

Even though the worst-case time is $O(n^2)$, the average time is only $O(n \log n)$. Let us now look at the stack space needed by the recursion. In the worst case the maximum depth of recursion may be $n – 1$. This happens, for example, when the partition element on each call to Partition is the smallest value in *a[m:p-1]*. The amount of stack space needed can be reduced to $O(\log n)$ by using an iterative version of Quick Sort in which the smaller of the two subarrays *a[p : j – 1]* and *a[j+1:q]* is always sorted first. Also, the second recursive call can be replaced by some assignment statements and a jump to the beginning of the algorithm.

Algorithm 11.9

```
Algorithm QuickSort 2(p, q)
//Sorts the elements in a [p: q]
{
    //stack is a stack of size 2 log (n)
    repeat
        {
            while (p<q) do
            {
            j:=Partition (a,p,q+1):
            if (j-p)<(q-j) then
            {
                Add (j+1); //Add j+1 to stack
                Add(q); q:=j-1; // Add q to stack
            }
      else
            {
                Add(p; //Add p to stack)
                Add (j-p); p:=j+1; //Add j-1 to stack
            }
    }// Sort the smaller subfile
    if stack is empty then return:
    Delete (q); Delete (p); //Delete q and p from stack
} until (false);
}
```

We can now verify that the maximum stack space needed is *O(log n)*
Let *S(n)* be the maximum stack space needed. Then it follows that

$$S(n) \le \begin{cases} 2 + S\left(\left\lfloor \frac{n-1}{2} \right\rfloor\right) & n > 1 \\ 0 & n \le 1 \end{cases}$$ which is less than *2 log n*

Self Assessment Question

5. The average value $C_A(n)$ of *C(n)* is much less than ——————.

11.7 Summary

- In this unit, we discussed that the complexity of many divide-and-conquer algorithms is given by recurrences of the form
  $$T(n) = \begin{cases} T(1) & n = 1 \\ aT\left(\frac{n}{b}\right) + f(n) & n > 1 \end{cases}$$ (where a and b are known constants).
- For successful binary search, $S(n) = 1 + \frac{I}{n}$ and for unsuccessful binary search, $U(n) = \frac{E}{n+1}$.
- For Max and Min, $3\left(\frac{n}{2}\right) - 2$ is the best, average and worst-case number of comparisons where n is a power of two.
- For Merge sort , we have *T(n) = O(n logn)*
- For quicksort, the worst case time is $O(n^2)$, the average time is only *O(n logn).*

11.8 Terminal Questions

1. Briefly explain the Binary search Method.
2. Write the Algorithm to find the maximum and minimum items in a set of n element.
3. Explain the concept of merge sort.
4. Sketch the Algorithm of Quick sort.
5. Write the Algorithm for sorting by Partitioning.

11.9 Answers

Self Assessment Questions

1. $T(n) = \begin{cases} T(1) & n = 1 \\ aT(n/b) + f(n) & n > 1 \end{cases}$
2. $\frac{l}{n} + 1$
3. Recursive calls
4. $T(n) = \begin{cases} a & n = 1, a \text{ is constant} \\ aT(n/2) + cn & n < 1, c \text{ is a constant} \end{cases}$
5. $C_w(n)$.

Terminal Questions

1. Let a_i, $1 \leq i \leq n$ be a list of elements that are sorted in non-decreasing order. Consider the problem of determining whether a given element *x* is present in the list. If *x* is present, we are to determine a value *j* such that $a_j = x$. If *x* is not present in the list, then *j* is to be set to zero.

   Let $P = (n, a_i, \ldots.., a_1, x)$ denote an arbitrary instance of this search problem, where *n* is the number of element in the list, a_i ……….. a_1 is the list of elements and *x* is the element searched for. (Refer section 11.3)
2. Refer section 11.4
3. Merge sort is based on the divide-and-conquer paradigm. Its worst-case running time has a lower order of growth than insertion sort. Refer section 11.5
4. Quicksort is a sorting algorithm developed by Tony Hoare that, on average, makes $O(n \log n)$ comparisons to sort n items. Refer section 11.6
5. Refer section 11.6