.2. Divide and Conquer

Divide and conquer (D&C) is an important algorithm design paradigm. It works by recursively breaking down a problem into two or more sub-problems of the same (or related) type, until these become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem. A divide and conquer algorithm is closely tied to a type of recurrence relation between functions of the data in question; data is "divided" into smaller portions and the result calculated thence.

Steps

- Splits the input with size n into k distinct sub problems $1 < k \leq n$
- Solve sub problems
- Combine sub problem solutions to get solution of the whole problem. Often sub problems will be of same type as main problem. Hence solutions can be expressed as recursive algorithms

Control Abstraction

Control abstraction is a procedure whose flow of control is clear but primary operations are specified by other functions whose precise meaning is undefined

```
DAndC (P)
{
    if Small(P) return S(P);
    else
    {
            divide P into smaller instances P1, P2,…,Pk k1;
            apply DandC to each of these sub problems;
            retun Combine(DandC(P1), DandC(P2),…,DandC(Pk));
    }
}
```

The recurrence relation for the algorithm is given by

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

g(n) – time to computer answer directly from small inputs.
f(n) – time for dividing P and combining solution to sub problems

2.1 Binary Search

Problem: Given a list of n sorted elements a_i, $1 \leq i \leq n$ sorted in ascending order. Check whether an element x is present in the list or not. If present, find the position of x in the list else return zero

Program 2.1

```
int BinSearch (int a[], int n, int x)
{
    int low=1, high=n, mid;
    while (low <= high)
```

```
    {
            mid = (low + high)/2;
            if (x < a[mid])
                    high = mid – 1;
            else if (x > a[mid])
                    low = mid + 1;
            else
                    return (mid);
    }
    return (0);
}
```

An instance of the problem can be represented as
$P = (n, a_i, \ldots, a_l, x)$
where n : no of elements
$a_i, \ldots, a_l$: elements in the list
x : number to be searched

Comparing the program with the Divide and Conquer control abstraction,
S(P) : Index of the element, if the element is present else it will return zero.
g(1) : $\Theta(1)$
If P has more than one element, divide the problem into sub problems as given below

- Pick an index 1 in the range [i, l]
- Compare x with a_q
  - $x = a_q$, solved
  - $x < a_q$, search x in sub list $a_i, a_{i+1}, \ldots, a_{q-1}$. Then $P = (q-i, a_i, \ldots, a_{q-1}, x)$
  - $x > a_q$, search x in sub list $a_{q+1}, a_{q+2}, \ldots, a_l$. Then $P = (l-q, a_{q+1}, \ldots, a_l, x)$

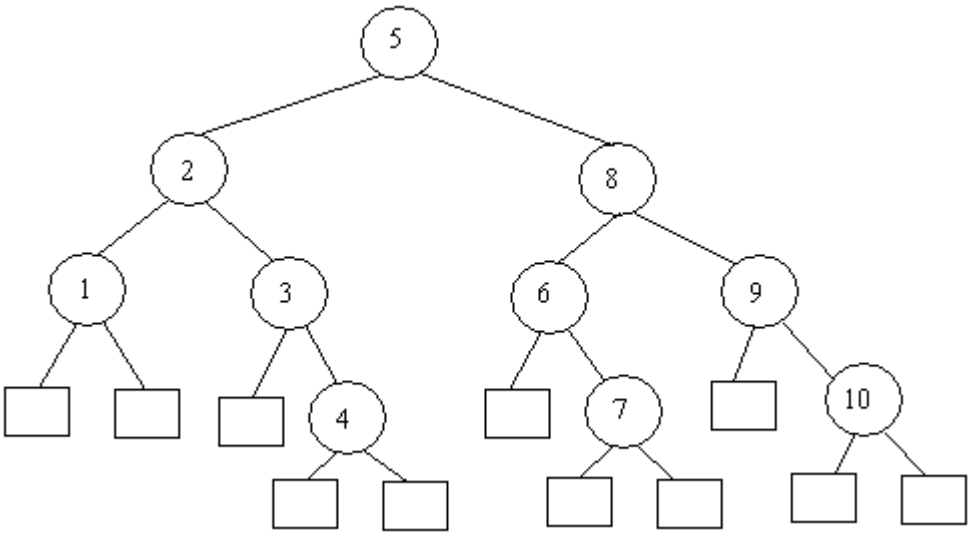
Time to divide P into one new sub problem is $\Theta(1)$. q is selected such that
$q = \lfloor (n+1)/2 \rfloor$. Answer to new sub problem is answer to P. Hence function Combine is not needed here.

Simulation

Consider 10 elements 10, 20, 30, 40, 50, 60, 70, 80, 90, 100
Number of comparison needed to search element is as per the table given below

Position	1	2	3	4	5	6	7	8	9	10
Element	10	20	30	40	50	60	70	80	90	100
No. of comparisons required	3	2	3	4	1	3	4	2	3	4

The search tree for the above list of elements will be as given below.



If the element is present, then search end in a circular node (inner node), if the element is not present, then it end up in a square node (leaf node)
Now we will consider the worse case, average case and best case complexities of the algorithm for a successful and an unsuccessful search.

Worse Case
Find out k, such that $2^{k-1} <= n <= 2^k$
Then for a successful search, it will end up in either of the k inner nodes. Hence the complexity is O(k), which is equal to $O(\log_2 n)$.
For an unsuccessful search, it need either k-1 or k comparisons. Hence complexity is Θ (k), which is equal to $\Theta(\log_2 n)$.

Average Case
Let I and E represent the sum of distance of all internal nodes from root and sum of distance of all external nodes from the root respectively. Then
$E = I + 2n$

Let $A_s(n)$ and $A_u(n)$ represents average case complexity of a successful and unsuccessful search respectively. Then
$A_s(n) = 1 + I/n$
$A_u(n) = E / (n+1)$

$A_s(n) = 1 + I/n$
$= 1 + (E-2n)/n$
$= 1 + (A_u(n)(n+1) – 2n)/n$
$= (n + (A_u(n)(n+1) – 2n))/n$
$= (n(A_u(n) -1) + A_u(n))/n$
$= A_u(n) – 1 + A_u(n)/n$
$A_s(n) = A_u(n)(1+1/n) - 1$

$A_s(n)$ and $A_u(n)$ are directly related and are proportional to $\log_2 n$.
Hence the average case complexity for a successful and unsuccessful search is $O(\log_2 n)$

Best Case

Best case for a successful search is when there is only one comparison, that is at middle position if there is more than one element. Hence complexity is $\Theta(1)$

Best case for an unsuccessful search is $O(\log_2 n)$

	Best Case	**Average Case**	**Worse Case**
Successful	$\Theta(1)$	$O(\log_2 n)$	$O(\log_2 n)$
Unsuccessful	$O(\log_2 n)$	$O(\log_2 n)$	$O(\log_2 n)$

2.2 Maximum and Minimum

Problem: Given a list of n elements a_i, $1 \leq i \leq n$, find the maximum and the minimum element from the list

Given below is a straight forward method to calculate the maximum and minimum from the list of n elements.

Program 2.2

```
Void StraightMinMax (int a[], int n, int *max, int *min)
{
    int i;
    *max = a[0];
    *min = a[0];
    for (i=1; i<n; i++)
    {
            if (a[i] > *max) *max = a[i];
            if (a[i] < *min)  *min = a[i];
    }
}
```

Let us calculate the time complexity of the algorithm. Here, only element comparisons are considered, because the frequency counts of other operations are same as that of element comparison. 2(n-1) comparisons are required for worse, average and best case. But if we modify the code as

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

Then Best case complexity is (n-1), it happens when elements are sorted in increasing order.

Worse case complexity is 2(n-1), when elements are sorted in decreasing order

Average case complexity is (2(n-1)+ (n-1))/2 = 3(n-1)/2

Now let us solve the problem using Divide and Conquer method. An instance of the problem can be represented as

P = (n, a[i],….,a[j])

where n : no of elements

a[i],….,a[j]: elements in the list

When n=1, max = min = a[1]. When n=2, the problem can be solved in one comparison. Hence, Small(P) is true when n<=2. If there are more elements, then we have to divide the problem into sub problems. So divide the problem into two instances,

$P_1 = (\lfloor n/2 \rfloor, a[1], \ldots, a[\lfloor n/2 \rfloor])$ and $P_2 = (n - \lfloor n/2 \rfloor, a[\lfloor n/2 \rfloor + 1], \ldots, a[n])$

To solve the problem, invoke Divide and Conquer algorithm recursively. Combine solution for P_1 and P_2. Set Max(P) as larger among Max(P_1) and Max(P_2) and Min(P) as smaller among Min(P_1) and Min(P_2)

Program 2.2

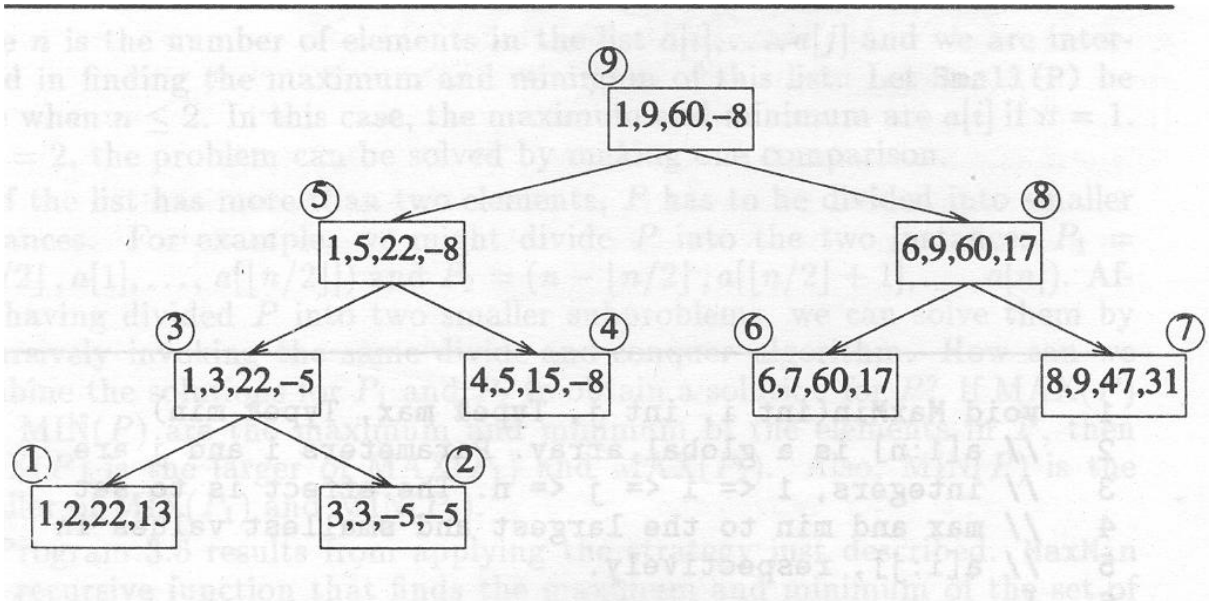
```
Void MaxMin (int i, int j, int *max, int *min)
{
    if (i==j)
    {
            *max = a[i];
            *min = a[i];
    }
    else if (i == j-1)
    {
            if (a[i] < a[j])
            {
                    *max = a[j];
                    *min = a[i];

            }
            else
            {
                    *max = a[i];
                    *min = a[j];

            }
    }
    else
    {
            mid = (i+j)/2;
            MaxMin (i, mid, *max, *min);
            MaxMin (mid+1, j, *max1, *min1);
            if (*max < *max1)      *max = *max1;
            if (*min > *min1)      *min = *min1;
    }
}
```

Example 2.1

Consider 9 elements 22, 13, -5, -8, 15, 60, 17, 31 and 47. The recursive call can be represented using a tree as given below



This can be represented by recurrence relation as

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

If n is a power of two, $n = 2^k$

$T(n) = 2T(n/2) + 2$
$= 2\,(2T(n/4) + 2) + 2$
$= 4T(n/4) + 4 + 2$
$= \ldots$
$= 2^{k-1} T(2) + \sum_{i=1}^{k-1} 2^i$
$= 2^{k-1} + 2^k - 2$
$= 2^{k-1} + n - 2$
$= n/2 + n - 2$

$T(n) = 3n/2 - 2$ (Best, average, Worst)

Comparing with the Straight forward method, Divide and Conquer is 50% more faster. But additional stack space is required for recursion.

Yet another figures are obtained if we include position comparison also, while calculating complexity. For this rewrite Small(P) as

if (i >= j-1) (Small(P)}

$$C(n) = \begin{cases} 2C(n/2) + 3 & n>2 \\ 2 & n=2 \end{cases}$$

$C(n) = 2C(n/2) + 3$
$= 4C(n/4) + 6 + 3$
$= 4C(n/4) + 3(1+2)$
$= \ldots$

$$= 2^{k-1}C(2) + 3\sum_{i=1}^{k-2} 2^i$$

$= 2^{k-1}.2 + 3[2^{k-1}-2]$

$= 2^k + 3.2^{k-1}-3$

$= n + 3.n/2 - 3$

C(n) $= 5n/2 - 3$

While using StraightMinMax the comparison is 3(n-1) which is greater than 5n/2–3. Even though the element comparison is less for MaxMin, it is slower than normal method because of the stack.

.

2.3 Merge Sort

Given a sequence of n elements a[1],...,a[n].

Spilt into two sets $a[1],..,a[\lfloor n/2 \rfloor]$ and $a[\lfloor n/2 \rfloor+1],\ldots, a[n]$

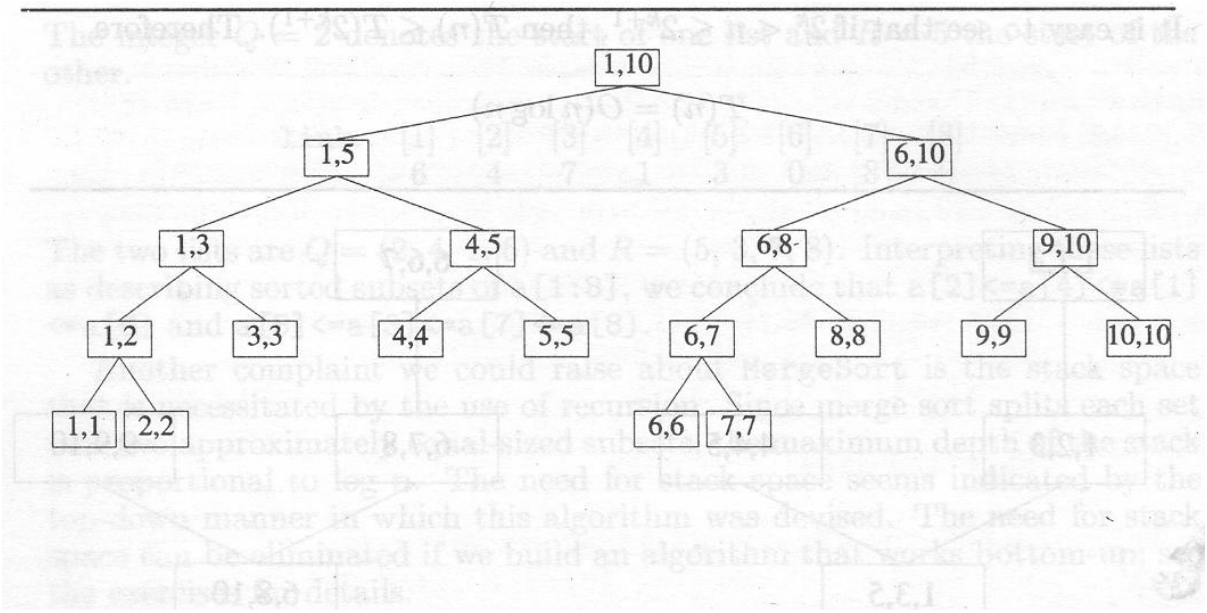
Each set is individually sorted, resultant is merged to form sorted list of n elements

Program 2.3

```
Void MergeSort (int low, int high)
{
    if (low<high)                    //Small(P) is true if there is only one element, if so list is sorted
    {
            mid = (high+low)/2;
            MergeSort(low, mid);      } // Subproblems
            MergeSort(mid+1, high);   }
            Merge(low, mid, high);       // Combine
    }
}
Void Merge (int low, int mid, int high)
{
    int l=low, i = low, j = mid+1, k;
    while((l <= mid) && (j <= high))
    {
            if (a[l] <= a[j])
            {       b[i] = a[l];      h++;)
            else
            {       b[i] = a[j];      j++;}
    }
    if (l<mid)
            for(k=j; k<=high; k++)
            {       b[i] = a[k];      i++;}
    else
            for(k=h; k<=mid; k++)
            {       b[i] = a[k];      i++;}
    for(k=low; k<=high; k++)
            a[k] = b[k];
}
```

Example 2.2

Consider 10 elements 310, 285, 179, 652, 351, 423, 861, 254, 450 and 520. The recursive call can be represented using a tree as given below



The recurrence relation for the algorithm can be written as

$$T(n) = \begin{cases} a & n=1 \\ 2T(n/2)+cn & n>1 \end{cases}$$

When n is power of 2, we can write $n = 2^k$

$T(n) = 2(T(n/4) + cn/2) + cn$
$= 4T(n/4) + 2cn$
$= 4(2T(n/8) + cn/4) + 2cn$
$= \ldots$
$= 2^kT(1) + kcn$
$T(n) = an + cnlogn$

If $2^k<n\leq2^{k+1}$, $T(n)\leq T(2^{k+1})$

Hence **T(n) = O(nlogn)**

2.4 Quick Sort

Given a sequence of n elements a[1],...,a[n].

Divide the list into two sub arrays and sorted so that the sorted sub arrays need not be merged later. This is done by rearranging elements in the array such that a[i]<=a[j] for $1\leq i\leq m$ and $m+1\leq j\leq n$, $1\leq m\leq n$. Then the elements a[1] to a[m] and a[m+1] to a[n] are independently sorted, Hence no merging is required.

Pick an element, t=a[s], reorder other elements so that all elements appearing before t is less than or equal to t, all elements after t is greater than or equal to t.

Program 2.4

```
Void QuickSort (int p, int q)
{
    if (p<q)        // more than one element, divide into sub problems
    {
            j = partition(a,p,q+1)
            QuickSort(p, j-1) ;  } Combine
            QuickSort(j+1, q) ;  }
    // No need of combining solutions
    }
}
int partition (int a[], int m, int p)
{
    int v=a[m], i=m, j=p ;
    do
    {
            do
                    i++;
            while(a[i]<v);
            do
                    j--;
            while(a[j]>v);
            if (i<j) interchange(a,i,j);
    }
    while(i<j);
    a[m] = a[j];    a[j] = v;        return(j);
}
```

While calculating complexity we are considering element comparison alone. Assume that n elements are distinct and the distribution is such that there is equal probability for any element to get selected for partitioning.

Worse case complexity

According to the previous program the number of elements is atmost p-m+1.
Let the total number of elements in all call of Partition at a level be *r*.
At level one there is only one call of Partition, hence r=n
At level two, there is at most two calls, hence r=n-1.
Hence we can tell, at all levels the complexity is O(R), where the value of r is diminishing at each level.
The worse case complexity $C_w(n)$ is the sum of r at each level, hence $C_w(n) = O(n^2)$

Average case complexity

The partitioning element has equal probability of becoming the i^{th} smallest element, $1 \leq i \leq p-m$ in the array a[m:p-1].
Hence probability for sub arrays being sorted, a[m:j], a[j+1:p-1]is 1/(p-m), $m \leq j < p$.

$$C_A(n) = n+1 + 1/n \sum_{k=1}^{n} [C_A(k-1) + C_A(n-k)]$$

$$nC_A(n) = n(n+1) + \sum_{k=1}^{n} [C_A(k-1) + C_A(n-k)]$$

$$nC_A(n) = n(n+1) + 2 [C_A(0) + C_A(1)+\ldots+ C_A(n-1)] \quad \text{---------- (1)}$$

Replacing n by n-1 in equ(1)

$$(n-1)C_A(n-1) = n(n-1) + 2 [C_A(0) + C_A(1)+\ldots+ C_A(n-2)] \quad \text{---------- (2)}$$

(1) – (2) =>

$$nC_A(n) - (n-1)C_A(n-1) = 2n + 2C_A(n-1)$$

Dividing by n(n+1)

$$nC_A(n)/(n+1) = 2/(n+1) + ((n+1)\ C_A(n-1))/(n(n+1))$$
$$C_A(n)/(n+1) = 2/(n+1) + C_A(n-1)/n$$
$$C_A(n)/(n+1) = C_A(n-1)/n + 2/(n+1)$$

Substituting for $C_A(n-1)$, $C_A(n-2)$, …

$$C_A(n)/(n+1) = C_A(n-2)/(n-1) + 2/n + 2/(n+1)$$
$$= \ldots.$$
$$= C_A(1)/2 + 2/2^{n-1} + 2/(n+1)$$
$$= C_A(1)/2 + 2 \quad 1/k$$
$$\leq \int_{2}^{n+1} 1/x\ dx = \log_e(n+1) - \log_e 2$$

$$C_A(n) = 2(n+1)[\log_e(n+1) - \log_e 2]$$
$$C_A(n) = O(n \log n)$$