Divide and conquer: General method, Defective chess board, Binary search, Finding the maximum and minimum, Merge Sort, Quick sort, performance measurement, Randomized Sorting algorithms.

DIVIDE AND CONQUER

General Method:

- Given a function to compute on *'n'* inputs, the divide and conquer strategy
    - Splits the input into k subsets, $1<k\leq n$, it yields k subproblems.
    - These subproblems must be solved.
    - A method must be found to combine subsolutions into a solution of the whole.
- The Divide and Conquer strategy is reapplied, if the subproblems are large.
- Often these subproblems are of the same type of the original problem. For this reason the divide-and-conquer principle is expressed by a *recursive algorithm*.
- Splitting the problem into subproblems is continued until the subproblems become small enough to be solved without splitting.

Control Abstraction:

- 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.
- Control abstraction that mirrors the way an algorithm is based on divide-and-conquer is shown below.

```
Algorithm 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 problems;
        return combine(DAndC(P1), DAndC(P2) . . . DAndC(Pk));
   }
}
```

- Initially algorithm is invoked as *DAndC(P)*, where *P* is the problem to be solved.
- *Small(P)* is a boolean-valued function, it determines whether the input size is small enough that the answer can be computed without splitting.
- *S(P)* is invoked if *Small(P)* returns *true*.
- If *Small(P)* returns false, then the problem *P* is divided into subproblems $P_1, P_2, . . P_k$. These subproblems are solved by recursive application of *DAndC*.
- Combine function combines the solutions of the k subproblems to determine the solution to problem *P*.
- If the size of *P* is n, and the sizes of k subproblmes are $n_1, n_2, . . n_k$, then computing time of the *DAndC* is described by the recurrence relation.

$$T(n) = \begin{cases} g(n) & n \text{ small} \\ T(n_1) + T(n_2) + ...+ T(n_k) + f(n) & otherwise \end{cases} \qquad ----(1)$$

Where

T(n) – time for DAndC on any input of size n.
g(n) – time to compute answer directly for small inputs.
f(n) – time for dividing P into subproblems and combing solutions of subproblems.

DAndC strategy produces subproblems of type original problem, Therefore it is convenient to write these algorithms using recursion.

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

$$T(n) = \begin{cases} T(1) & n = 1 \\ a.T\left(\frac{n}{b}\right) + f(n) & n > 1 \end{cases} \qquad \text{----(2)}$$

Where *a* and *b* are known as constants.

To solve this recurrence relation, we assume that *T(1)* is known, *n* is a power of *b* (i.e., $n = b^k, \quad \log_b n = k$)
Substitution method of solving Recurrence Relation repeatedly makes substitution for each occurrence of the function in right hand side until all such occurrences disappear.

Example: consider the case in which *a=2* and *b=2, T(1)=2* and *f(n)=n.*

$$T(n) = \begin{cases} T(1) & n = 1 \\ a.T\left(\frac{n}{b}\right) + f(n) & n > 1 \end{cases}$$

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

$$= 2\left[2.T\left(\frac{n}{4}\right) + \frac{n}{2}\right] + n$$

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

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

$$= 8.T\left(\frac{n}{8}\right) + 3.n$$

$$\vdots$$

In general $\quad = 2^i.T\left(\frac{n}{2^i}\right) + i.n \qquad$ for any $\log_2 n \geq i \geq 1$

In particular $\; T(n) = 2^{\log_2 n}.T\left(\frac{n}{2^{\log_2 n}}\right) + n.\log_2 n$

$$T(n) = n.T(1) + n.\log_2 n$$

$$T(n) = n.\log_2 n + 2n$$

Solving recurrence Relation (2) using substitution method, we get

$$T(n) = a^{\log_b n}.T(1) + \log_b n.f(n)$$

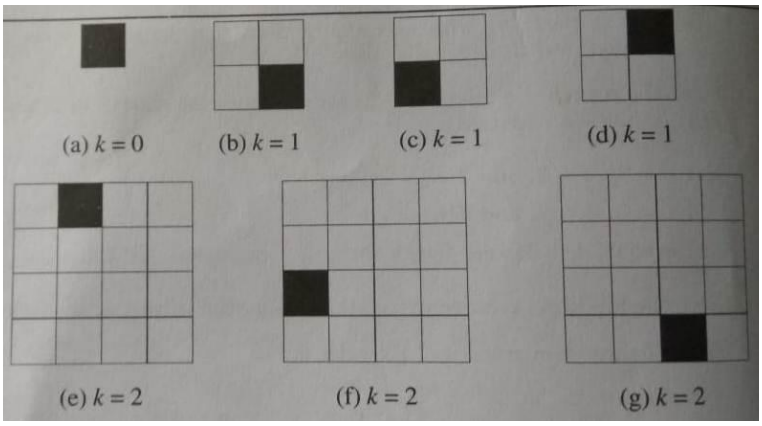
$$= n^{\log_b a}.T(1) + \log_b n.f(n)$$

$$= n^{\log_b a}\left[T(1) + u(n)\right]$$

$$where \; u(n) = \sum_{j=1}^{k} h(b^j) \quad \text{and } h(n) = \frac{f(n)}{n^{\log_b a}}$$

Defective Chessboard Problem :

- A defective chessboard is a $2^k \ X \ 2^k$ board of squares with exactly one defective square.
- Possible defective chessboards for $k \le 2$ are shown below.



- Shaded square is defective.

- When k = 0, the size of the chess board is 1 X 1 and there is only one possible defective chessboard.
- When k = 1, the size of the chess board is 2 X 2, and there can be 4 possible defective chessboards.
- Therefore, for any k, there are exactly 2^{2k} defective chessboards.

Triomino :

- A triomino is an L shaped object that can cover three squares of a chessboard.
- A triomino has four orientations. Following figure shows triominoes with different orientations.

<u>Defective chessboard problem:</u>

- In this problem, we are required to tile a defective chessboard using triominoes.

Constraints:

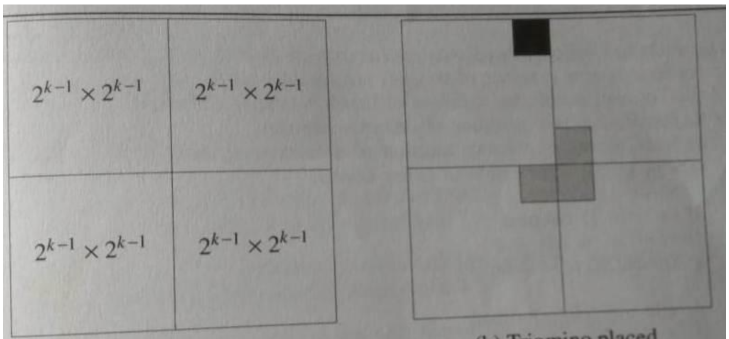
  - Two triominoes may not overlap in this tiling.
  - Triominoes should not cover defective square.
  - Triominoes must cover all other squares.

With the above constaints, number of triominoes required to tile = $\frac{2^{2k} - 1}{3}$

If k=0, number of triominoes = 0
If k=1, number of triominoes = 1
If k=2, number of triominoes = 5

<u>Solution :</u>

- Divide and Conquer leads to an elegant solution to this problem.
- The method suggests reducing the problem of tiling a $2^k \ X \ 2^k$ defective chessboard to tiling a smaller defective chessboard.
- A $2^k \ X \ 2^k$ chessboard can be partitioned into four $2^{k-1} \ X \ 2^{k-1}$ chessboards.



(a) Partitioning (b) Triomino placed

- Only one board has a defective square. To convert the remaining three boards into defective chessboards, we place triomino at the corner formed by these three.
- This partitioning technique is recursively used to tile the entire $2^k\ X\ 2^k$ chessboard.
- This recursion terminates when the chessboard size has been reduced to 1 X 1

Pseudocode for this strategy to solve Defective chessboard problem :

```
Algorithm TileBoard(topRow, topCol, dRow, dCol, size)
//topRow, topCol are top-left corner of the board
//dRow, dCol are row and column numbers of defective square.
//size is length of one side of the chessboard.
{
   if ( size = 1) return;
   tileToUse := tile++;
   quadSize = size / 2;

   //tile top-left quadrant
   if (dRow < topRow + quadSize && dCol < topCol + quadSize) then
        //defect is in this quadrant
        TileBoard(topRow, topCol, dRow, dCol, quadSize);
   else
   {
        //no defect, place a tile in bottom-right corner
        board[topRow + quadSize -1][topCol + quadSize – 1] := tileToUse;
        TileBoard(topRow,topCol, topRow+quadSize – 1, topCol+quadSize-1, quadSize);
   }

   //code for remaining three quadrands is similar
}
```

- Above pseudocode uses two global variables
  - board is a two-dimensional array that represents the chessboard. board[0][0] represents the top-left corner.
  - tile, with initial value 1, gives the index of the next tile to use.
- This algorithm is invoked with the call ***TileBoard(0,0,dRow,dCol,size)***
  Where
  - Size = 2^k
  - dRow and dCol are row and column index of the defective square.

Time Complexity:

Let ***t(k)*** denote the time taken by TileBoard to tile a defective chessboard.
When k=0, a constant amount of time is spent. Let the constant be ***d***.
When k>0, recursive calls are made. These calls take ***4.t(k – 1)*** time.
This can be represented by the following recurrence equation.

$$t(k) = \begin{cases} d & K = 0 \\ 4t(k-1) + c & K > 0 \end{cases}$$

By solving this using the substitution method, we obtain

$$t(k) = \theta(4^k) = \theta(number\ of\ tiles\ needed)$$

Binary Search:

- Let a_i be a list of elements that are sorted in nondecreasing order.
- Here i is in the range of 1 to n, $1 \le i \le n$.
- Binary search is a problem of determining whether an element x is present in the list or not.
  - If x is present in the list, then we need to determine value j such that $a_j=x$.
  - If x is not in the list, then j is to be set to zero.

- Let $P = (n, a_{low}, \ldots, a_{high}, x)$ denote an instance of search problem. Here
  - n is number of elements in the list.
  - $a_{low}, \ldots, a_{high}$ list of elements
  - x is the element searched for.

- Divide-and-conquer can be used to solve this problem.
  - Let $Small(P)$ be true if $n=1$.
  - If $x=a_i$ then $Small(P)$ will take the value of i otherwise $Small(P)$ will take the value 0
    ⇨ $g(1) = \Theta(1)$.

- If P has more than one element, it can be divided into subproblems, as follows.
  Pick and index q in the range low to $high$ and compare x with a_q. There are three possibilities.

  1. $\boldsymbol{x = a_q}$: The problem P is immediately solved in this case.
  2. $\boldsymbol{x < a_q}$: x has to be searched in the sublist in this case.
     The sublist is $a_{low}, \ldots, a_{q-1}$.
     Therefore, P reduces to $P = (n, a_{low}, \ldots, a_{q-1}, x)$
  3. $\boldsymbol{x > a_q}$: In this case also x has to be searched in the sublist
     Therefore P reduces to $P = (n, a_{q+1}, \ldots, a_{high}, x)$.

- Dividing the problem P into subproblem takes $\Theta(1)$ time.
- After a comparison with $\boldsymbol{a_q}$, the remaining problem instance can be solved by using divide-and-conquer scheme again.
- If q is choosen that $\boldsymbol{a_q}$ is the middle element, then that search algorithm is called **Binary Serach**.

  i.e., $q = \left\lfloor \frac{(low + high)}{2} \right\rfloor$ 8074816703

- There is no need of combining answers in binary search, because answer of the subproblem is also the answer of the original problem *P*.

Algorithm for recursive binary Search:

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

- It is initially invoked as ***BinSearch(a,1,n,x)***

Algorithm for non-recursive binary search

```
Algorithm BinSearch(a,n,x)
{
    low=1, high=n;
    while (low ≤ high)
    {
        mid:=(low+high)/2;
        if (x=a[mid]) then return mid;
        else if (x<a[mid]) then high:=mid-1;
        else low:=mid+1;
    }
    return 0;
}
```

Non-recursive binary search algorithm has three inputs: *a*, *n*, *x*. The while loop continues processing as long as there are elements left to check. A zero is returned if *x* is not present in the list, or *j* is returned if $a_j=x$.

Example: Consider a list with 14 entries. i.e., n=14.
Place them in a[1:14]

1	2	3	4	5	6	7	8	9	10	11	12	13	14
-15	-6	0	7	9	23	54	82	101	112	125	131	142	151

The variables low, high, mid need to be traced to simulate this algorithm.

x=151	low	high	mid
	1	14	7
	8	14	11
	12	14	13
	14	14	14
			found

x= -14	low	high	mid
	1	14	7
	1	6	3
	1	2	1
	2	2	2
	2	1	Not found

Space Required for Binary Search :
For binary search algorithm, storage is required for – *'n'* elements of the array, variables *low, high, mid, x*. i.e., *n+4* locations.

Time required for Binary Search :

- The three possibilities that are needed to be considered are best, average and worst cases.
- To determine time for this algorithm, concentrate on comparisons between *x* and the elements in *a[]*.
- Comparisons between *x* and the elements in *a[]* are referred to as element comparisons.
- To test all successful searches, *x* must take on *'n'* values in *a[]*.
- To test all unsuccessful searches, *x* need to take *(n+1)* comparisons.

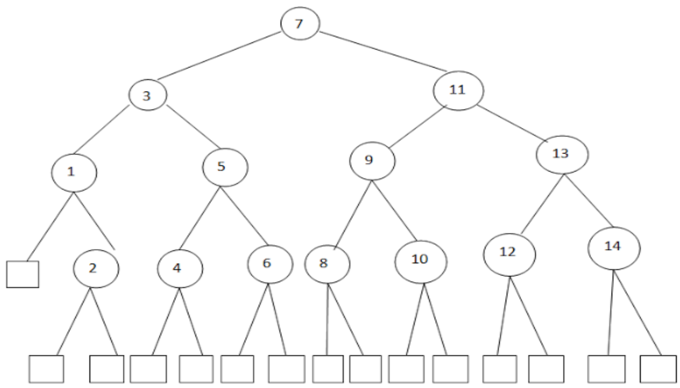
Example: The number of element comparisons needed to find each of the 14 elements is

a:	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Elements:	-15	-6	0	7	9	23	54	82	101	112	125	131	142	151
Comparisons:	3	4	2	4	3	4	1	4	3	4	2	4	3	4

- Average Comparisons for successful search : $\frac{45}{14} \approx 3.21$
- There are (14+1=15) possible ways that an unsuccessful search may terminate.
- If *x<a[1]*, then the algorithm requires 3 element comparisons to determine that x is not present. For remaining cases, the algorithm requires 4 element comparisons.
- Average number of element comparisons for unsuccessful search = $\left(\frac{3+14*4}{15}\right) = \frac{59}{15} \approx 3.93$

But we prefer a formula for *n* elements. A good way to derive a formula is to consider sequence of mid values that are produced by all possible values of *x*. A <u>*Binary Decision Tree*</u> used to describe this. Each tree in this node is the value of mid.

Example : if n=14, a Binary Decision Tree that traces the way in which these values are produced is shown below.



- The first comparison is x with a[7]. If x<a[7], then the next comparison is with a[3]; If x>a[7], then the next comparison is with a[11].
- Each path through the tree represents a sequence of comparisons in the binary search method.
- If x is present then the algorithm will end at one of the circular nodes that lists the index into the array where x was found.

- If x is not present, the algorithm will terminate at one of the square nodes.
- Circular nodes are called *internal nodes*, and square nodes are referred to as *external nodes*.

If n is in the range $[2^{k-1}, 2^k]$ then binary search makes atmost *k* element comparisons for a successful search and either *k-1* or *k* comparisons for an unsuccessful search.
i.e., the time for a successful search is *O(log n)* and for an unsuccessful search is $\Theta(\log n)$

Average, worst case time for binary search:

- From the BDT, it is clear that the distance of a node from the root is one less than its level.
- **Internal Path length (I) :** sum of the distances of all internal nodes from the root.
- **External Path length (E) :** sum of the distances of all external nodes from the root.

- By mathematical induction, we can say that “for any binary tree with n internal nodes, E and I are related by the formula”

$$E=I+2n.$$

- Let $A_s(n)$ be the average number of comparisons in a successful search, and $A_u(n)$ be the average number of comparisons in an unsuccessful search.
- The number of comparisons needed to find an element represented by an internal node is one more than the distance of this node from the root.

$$\text{Hence } A_s(n) = 1 + \frac{I}{n}$$

Since every binary tree with n internal nodes has n+1 external nodes, it follows that

$$A_u(n) = \frac{E}{(n+1)}$$

Using these three formulas,

$$\begin{aligned} A_s(n) &= 1 + \frac{I}{n} \\ &= 1 + \frac{\text{E - 2n}}{\text{n}} \\ &= 1 + \frac{A_u(n).(n+1) - 2n}{n} \\ &= 1 + \frac{A_u(n).n + A_u(n)}{n} - 2 \\ &= \left(1 + \frac{1}{n}\right) A_u(n) - 1 \end{aligned}$$

From this, we see that $A_s(n)$ and $A_u(n)$ are directly related.
From BDT, we conclude that average and worst case comparisons for Binary Search are same within a constant time.

Best-case : For a successful search only one element comparison is needed and for unsuccessful search log n element comparisons are needed in best case.

In conclusion, the computing times of binary search are

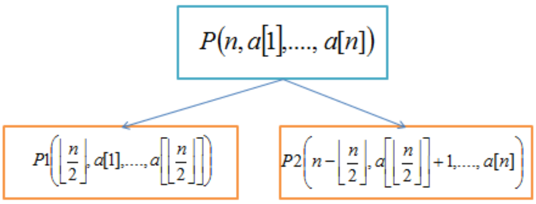
Successful Searches			Unsuccessful Searches
$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$
Best	Average	Worst	Best, average, worst

Finding the Maximum and Minimum:

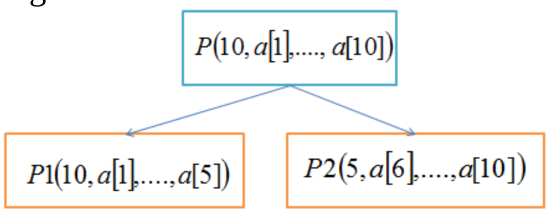
Problem : Find the maximum and minimum items in a set of ‘n’ elements.

Solution by using Divide and Conquer approach:

- Let P = (n, a[i], …, a[j]) denote an instance of the problem.
- Where n is the number of elements in the list and a[i],..a[j] denote the list in which we want to find minimum and maximum.
- Let **Small(P)** be true when n≤2
- If n = 1, the maximum and minimum are a[i]
- If n = 2, the problem can be solved by doing one element comparison.
- Otherwise, P has to be divided into smaller instances.
- Like,



Eg:



- After dividing P into smaller subproblems, we can solve them by recursively invoking the same divide and conquer algorithm.

Combining the solutions :

- Let P is the problem and P1 and P2 are its subproblems, then
  - MAX(P) is larger of MAX(P1) and MAX(P2) and
  - MIN(P) is smaller of MIN(P1) and MIN(P2)

Algorithm : to find maximum and minimum recursively.

```
Algorithm MaxMin(i, j, max, min)
// a[1 : n] is a global array. Parameters i and j are integers, 1 ≤ i ≤ j ≤ n.
//sets the max and min to the smallest and largest values.
{
        if (i = j) then max := min := a[i];
        else if ( i = j – 1) then
        {
                if (a[i] < a[j]) then
                {
                        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) then max := max1;
        if (min > min1) then min := min1;
        }
}
```

The procedure is initially invoked by the statement

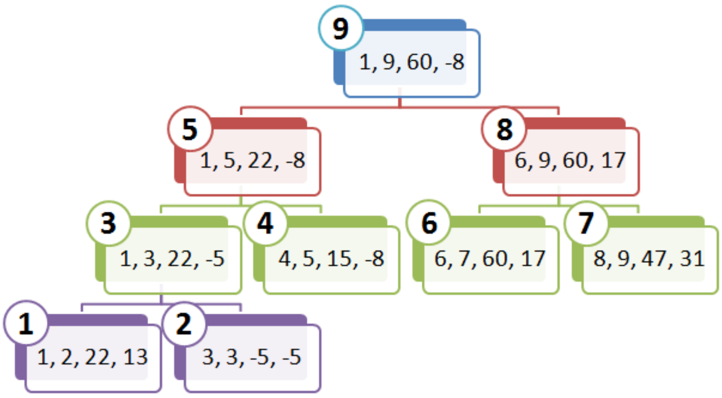
MaxMin(1,n,x,y)

Simulation

n = 9

	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
a	22	13	-5	-8	15	60	17	31	47

Tree of recursive calls



- Root node contains 1 and 9 as values of i, j corresponding to initial call to MaxMin.
- The produced two new calls, where i, j values are 1, 5 and 6, 9 respectively.
- From the tree, maximum depth of recursion is four.
- Circled numbers represent the orders in which Max & Min assigned values.

Analysis:

Computing Time : What is number of element comparsions needed?

$$T(n) = \begin{cases} 0 & n = 1 \\ 1 & n = 2 \\ 2.T\left(\frac{n}{2}\right) + 2 & n > 2 \end{cases}$$

Solve this recurrence equation using substitution method. $T(n) = \frac{3n}{2} - 2$

- It is the best, average, worst case number of comparisons when ‘n’ is power of 2.
- Number of comparsions in a straight method of maximum and minimum is 2n -2 . i.e., this algorithm saves 25% of comparisons.

Storage:

- MaxMin is worse than the straight forward algorithm because it requires stack space for [i,j, max, min, max1, min1]
- For ‘n’ elements, there will be log n + 1 levels of recursion and we need to save seven values for each recursive call.

Merge Sort:

- Merge Sort is a sorting algorithm with the nice property that its worst case complexity is O(n log n).
- Given a sequence of *'n'* elements *a[1],…,a[n]* the general idea is to imagine them split into two sets $a[1],.....,a[\left\lfloor \frac{n}{2} \right\rfloor]$ and $a[\left\lfloor \frac{n}{2} \right\rfloor+1],....a[n]$.
- Each set is individually sorted, and the resulting sorted sequences are merged to produce a single sorted sequence of 'n' elements.

- It is an ideal example of divide-and-conquer strategy, in which
    - Splitting is into two equal sized sets
    - Combining operation is merging of two sorted sets into one.
- *MergeSort* algorithm describes this process using recursion and *Merge* algorithm merges two sorted sets.
- *'n'* elements should be placed in *a[1:n]* before executing *MergeSort*. Then *MergeSort(1,n)* causes the keys to be rearranged into nondecreasing order in *a*.

Algorithm for MergeSort:

```
Algorithm MergeSort(low, high)
//a[low:high] is a global array to be sorted. Small(P) is true if there is only one element
{
    if (low < high ) then   //if there are more than one element
    {
        mid := ⌊(low+high)/2⌋;          //Divide P into sub probelms
        //Solve sub problems
        MergeSort(low, mid);
        MergeSort(mid+1, high);
        Merge(low, mid, high);
    }
}
```

Algorithm for merging two sorted subarrays.

```
Algorithm Merge(low, mid, high)
//a[low:high] is a global array containing two sorted subsets.The goal is to merge these two
//sets into a single set. b[] is an auxiliary 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;
        }
    }
```

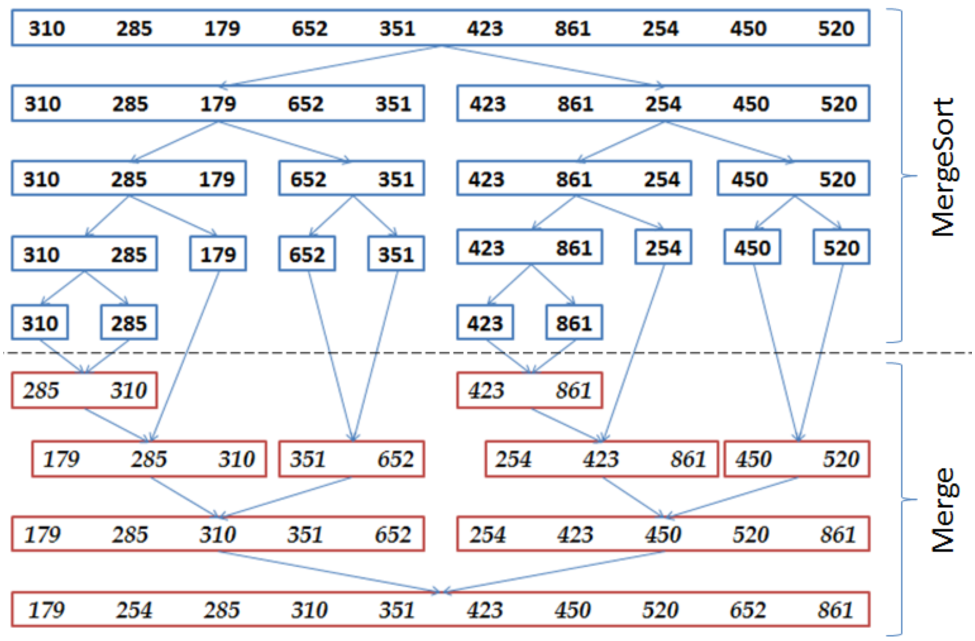
```
        if (h<mid) then
          for k := j to high do
          {
            b[i]:=a[k]; i:=i+1;
          }
        else
          for k:=h to mid do
          {
            b[i]:=a[k]; i:=i+1;
          }
        for k:=low to high do
          a[k]:=b[k];
      }
```

Example :

Consider an array of 10 elements.

a[1:10] = (310, 285, 179, 652, 351, 423, 861, 254, 450, 520)

Algorithm MergeSort begins by splitting *a[]* until they become one-element subarrays. Now merging begins. This division and merging is shown in the below figure.



- Following figure is a tree that represents the sequence of recursive calls that are produced by **MergeSort** when it is applied to *10* elements.
- The pair of values in each node is the values of the parameters *low* and *high*.

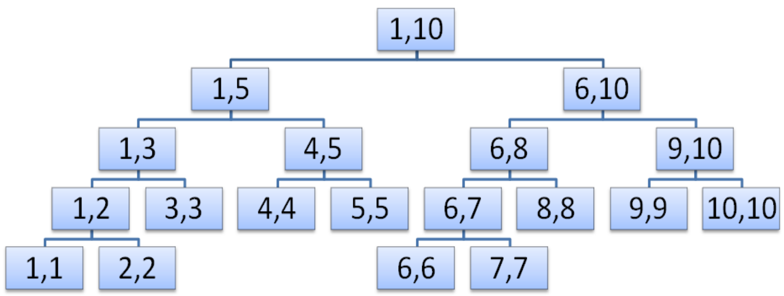


Figure: Tree of calls of MergeSort(1,10)

- Following figure is a tree representing the calls to procedure Merge. For example, the node containing 1, 2, and 3 represents the merging of a[1:2] with a[3].

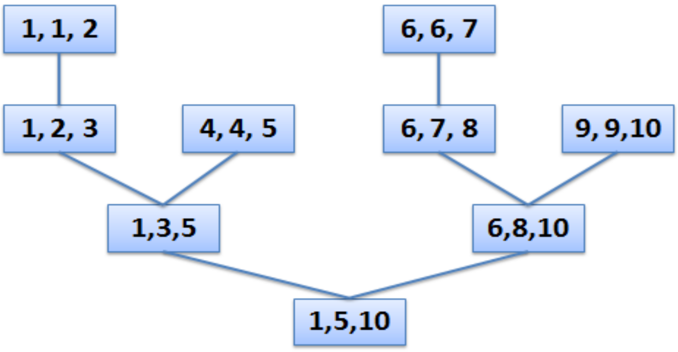


Figure: Tree of calls of Merge

If the time for the merging operation is proportional to *n*,

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

$$T(n) = \begin{cases} a & n = 1 \\ a.T\left(\frac{n}{2}\right) + c.n & n > 1 \end{cases}$$

Where *a* and *c* are constants.

We can solve this equation by successive substitutions.
Assume n is a power of *2*, $n=2^k$, i.e., $\log n = k$.

$$\begin{aligned} T(n) &= 2.T\left(\frac{n}{2}\right) + c.n \\ &= 2\left[2.T\left(\frac{n}{4}\right) + c.\frac{n}{2}\right] + cn. \\ &= 2^2.T\left(\frac{n}{4}\right) + 2cn \\ &= 2^2\left[2.T\left(\frac{n}{8}\right) + c.\frac{n}{4}\right] + 2cn \\ &= 2^3.T\left(\frac{n}{8}\right) + 3cn \\ &\text{after k substitutions} \\ &= 2^k.T(1) + kcn \\ &= an + cn.\log n \end{aligned}$$

It is easy to see that

$$\text{if } 2^k < n \le 2^{k+1}, \text{ then } T(n) \le T(2^{k+1})$$
$$\therefore T(n) = O(n \log n)$$

Quick Sort:

- In Quick sort, the division into two subarrays is made in such a way that, the sorted subarrays do not need to be merged later.
- This is achieved by rearranging the elements in *a[1:n]* such that *a[i]≤a[j]* for all *i* between *1* and *m* and all *j* between *m+1* and *n* for some *m, 1≤m≤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 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*.
    - The rearranging is referred to *partitioning*.

Following algorithm accomplishes partitioning of elements of a[m:p]. It is assumed that a[m] is the partitioning element.

```
Algorithm Partition(a, m, p)
{
    pivot:=a[m], i:=m+1, j:=p;
    while (i<j) do
    {
        while (a[i]≤pivot and i≤j) do
                i:=i+1;
        while (a[j]≥pivot and j≥i) do
                j:=j-1;
        if (i<j) then
                interchange(a, i, j);
    }
    interchange(a, m, j);
    return j;
}
```

The function interchange(a,i,j) exchanges a[i] with a[j].

```
Algorithm interchange(a, i, j)
{
    temp:=a[i];
    a[i]:=a[j];
    a[j]:=temp;
}
```

Example: working of partition.

Consider the following array of 9 elements.
The partition function is initially invoked as **Partition**(*a, 1, 9*).
The element a[1] i.e., 65 is the partitioning element

a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	i	j	
65	70	75	80	85	60	55	50	45	2	9	i<j, swap a[i],a[j]
65	45	75	80	85	60	55	50	70	3	8	i<j, swap a[i],a[j]
65	45	50	80	85	60	55	75	70	4	7	
65	45	50	55	85	60	80	75	70	5	6	
65	45	50	55	60	85	80	75	70	6	5	i>j, swap pivot with a[j]
60	45	50	55	**65**	85	80	75	70			Now elements are partitioned about pivot i.e., 65

Now the elements are partitioned about pivot element and the remaining elements are unsorted.

- Using this method of partitioning, we can directly devise a divide and conquer method for completely sorting *n* elements.
- Two sets S1 and S2 are produced after calling partition. Each set can be sorted independently by reusing the function partition.

Following algorithm describes the complete process.

```
Algorithm QuickSort(low, high)
{
    if (low<high) then            //if there are more than one element.
    {
        j:=partition(a,low,high);       //Partitioning into subproblems
        // Solve the subproblems
        QuickSort(a, low, j-1);
        QuickSort(a, j+1,high);
    }
}
```

Analysis:

- In quicksort, the pivot element we chose divides the array into 2 parts.
  - One of size *k.*
  - Other of size *n-k.*
- Both these parts still need to be sorted.
- This gives us the following relation.

$$T(n) = T(k)+T(n-k)+c.n$$

Where T(n) refers to the time taken by the algorithm to sort n elements.

Worst-case Analysis:

Worst case happens when pivot is the least element in the array.

Then we have $k=1$ and $n-k=n-1$

$$\Rightarrow T(n) = T(1) + T(n-1) + c.n$$

$$= T(1) + [\ T(1) + T(n-2) + c.(n-1)\ ] + c.n$$

$$= T(n-2)+2.T(1)+c.(n-1+n)$$

$$= [\ T(1) + T(n-3) + c.(n-2)\ ] + 2.T(1)+c.(n-1+n)$$

$$= T(n-3)+3.T(1)+c.(n-2+n-1+n)$$

:

Continuing likewise till ith step

$$= T(n-i)+i.T(1)+c.(n-i-1+\ ..\ +n-2+n-1+n$$

$$= T(n-i) + i.T(1) + c.\sum_{j=0}^{i-1}(n-j)$$

This recurrence can go until i=n-1. Substitute i=n-1

$$T(n) = \mathrm{T(1)} + \mathrm{(n-1).T(1)} + \mathrm{c.}\sum_{\mathrm{j=0}}^{\mathrm{n-2}} \mathrm{(n-j)}$$

$$= \mathrm{n.T(1)} + \mathrm{c.}\sum_{\mathrm{j=0}}^{\mathrm{n-2}} \mathrm{(n-j)}$$

$$= \mathrm{O(n^2)}$$

Best-case Analysis:
Best case of Quick Sort occurs when pivot we pick divide the array into two equal parts, in every step.

$$\therefore k = \frac{n}{2}, \quad \mathrm{n-k} = \frac{\mathrm{n}}{2}, \quad \text{for array of size n}$$

$$\text{We have } \mathrm{T(n)} = \mathrm{T(k)} + \mathrm{T(n-k)} + \mathrm{c.n}$$

$$= 2.\mathrm{T}\left(\frac{\mathrm{n}}{2}\right) + c.n$$

Solving this gives ***O(n log n).***

Randomized Quick Sort:

- Algorithm Quick Sort has an average time of *O(n log n)* and worst case of $O(n^2)$ on *'n'* elements.
- It does not make use of any additional memory like Merge Sort.
- Quick Sort can be modified by using randomizer, so that its performance will be improved.
- While sorting the array *a[p:q]*, pick a random element (from *a[p]...a[q]*) as the partition element.
- The randomized algorithm works on any input and runs in an expected *O(n log n)* time, where the expectation is over the space of all possible outcomes of the randomizer.
- The code of randomized quick sort is given below. It is a *Las Vegas algorithm* since it always outputs the correct answer.

```
Algorithm RQuickSort(p, q)
{
    if (p<q) then
    {
        if ((q-p)>5) then
                interchange(a, Random() mod (q-p+1)+p,p);
        j := partition(a, p, q+1);
        RQuickSort(p, j-1);
        RQuickSort(j+1, q);
    }
}
```

- Reason for invoking randomizer only if *(q – p)>5* is
  - Every call to randomizer Random takes a certain amount of time.
  - If there are only a few elements to sort, the time taken by the randomizer may be comparable to the rest of the computation.