Module-2

Divide & Conquer

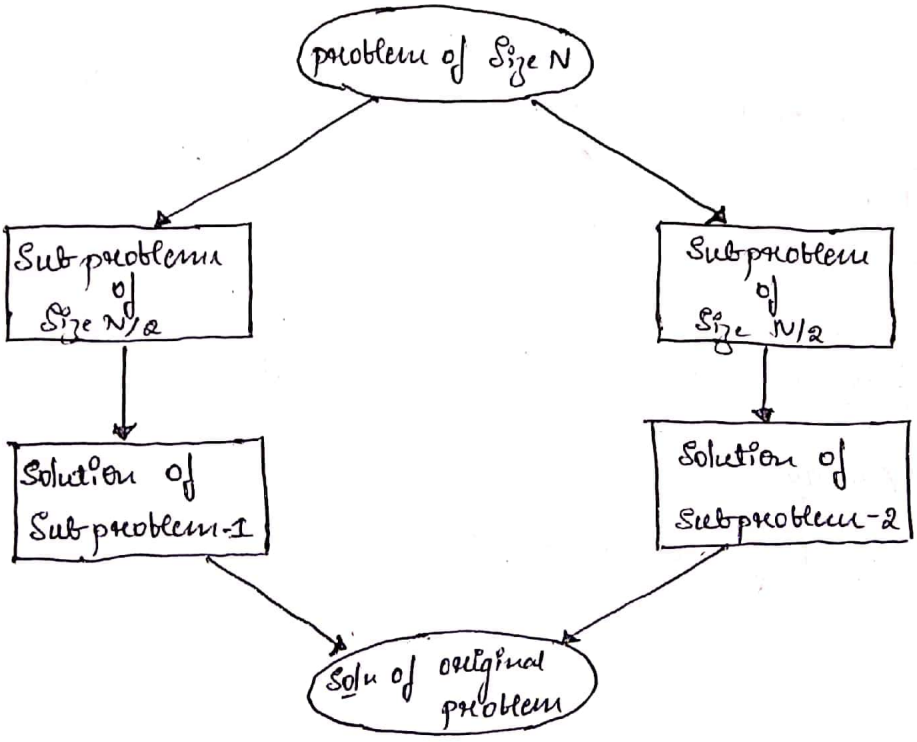
We know that some of the problems can be solved straight-away using Brute-force Approach. But, in many cases, Brute-force fails. So we'll make use of divide-&-conquer.

* General plan of Divide-&-Conquer

→ An Instance of a given problem is divided into several smaller instances of same type of problem & of Equal Size.

→ These smaller problems are solved, usually by recursive Method.

→ The Solutions of all these subproblems are combined to get the solution of original problem.



→ Divide-&-Conquer is one of the best design techniques, It is not necessarily more Efficient than Brute-force in many of the Situations.
If the problem suits the Criteria of Divide-&-Conquer, then

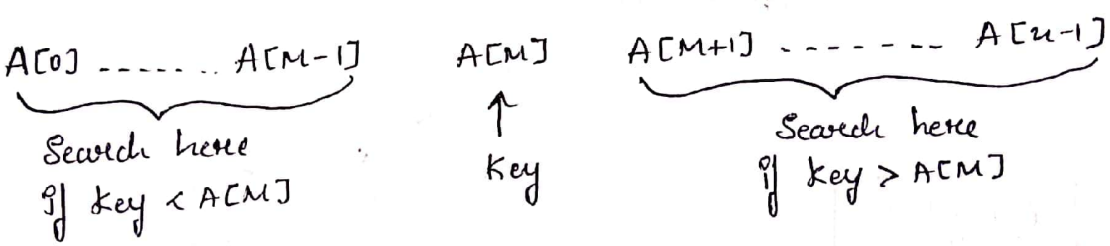
It will yield an Efficient Algorithm.

→ Divide-&-Conquer is Ideally Suited for parallel computation problems.

* Binary Search :-

Is a Very Efficient Searching Algorithm on Sorted list. Here, the key Element is Searched with the Middle Element of the list

* If they are Equal, the position of Middle Element is returned & Algorithm Stops.
* If the key is greater than the Middle Element, then the Searching is Applied on Second half of the Array.
* Otherwise on the first half of the Array.



* Algorithm Binary Search (A[0...n-1], key)

I/P :- Sorted Array A[0...n-1]

O/P :- position of key, otherwise -1

```
l ← 0
n ← n-1
while l ≤ n do
    m ← ⌊(l+n)/2⌋
    if key = A[M]
        return M
```

Else if key < A[M]

$r \leftarrow m-1$;

Else

$l \leftarrow m+1$;

return -1.

* Analysis :-

1. the parameter is input size n
2. Basic operation is comparison of key with Array Element.
3. no of times the Basic operation is Executed, "C(n/2)" is the time required for Searching Either of Subarray & "1" is for comparing the key with Middle Element.

$$C(n) = \begin{cases} C(n/2)+1, & n>1 \\ 1C, & n=1 \end{cases}$$

$\therefore C(n) = C(n/2) + 1$

$= C(n/4) + 1 + 1$ $\quad \because C(n/2) = C(n/4)+1$

$= C(n/2^2) + 2$

$\vdots$

$= C(n/2^k) + k$ $\quad \because$ Assume $n = 2^k$

$= C(2^k/2^k) + k$

$= C(1) + k$

$= 1 + k$ ---------- (1)

→ As we considered $n = 2^k$, Apply $\log_2$ on both side

$\log_2 n = \log_2 2^k$

$\log_2 n = k \log_2 2$

$\therefore k = \log n$ ---------- (2)

Substitute (2) in (1)

$$= 1 + \log_2^n$$

$$= \log_2^n$$

$$\therefore C(n) = \Theta(\log_2^n)$$

* Merge sort Algorithm is one of the most efficient sorting algorithm, it works on the principle of Divide & Conquer.

→ Merge sort repeatedly breaks down a list into several sublists until each sublist consists of a single element. & Merging those sublists in a manner that results into a sorted list.

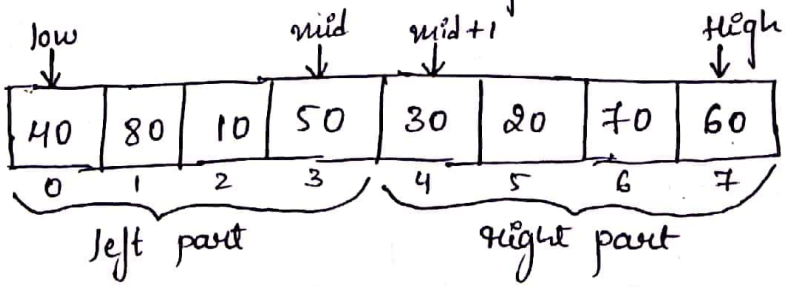
→ Consider the following Example program for 40, 80, 10, 50, 30, 20, 70, 60.

* Step 1:- Firstly check for elements are present in Array

if (low > high) return

* Step 2:- Divide the given Array of elements into Equal parts

$$mid = (low + high)/2$$



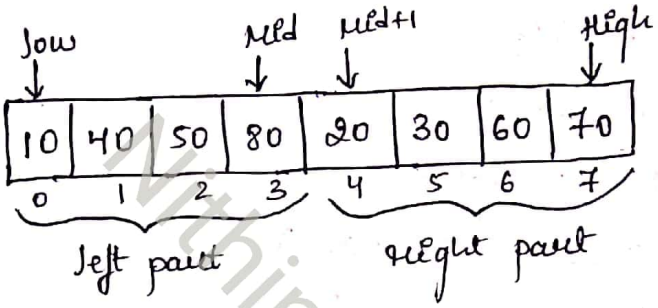
* Step 3:- Sort the left part of the Array A.

MergeSort(a, low, mid)

* Step 4 :- Sort the right part of the Array A

Mergesort (a, mid+1, high)

*After Sorting the left part & right part, the contents of Array



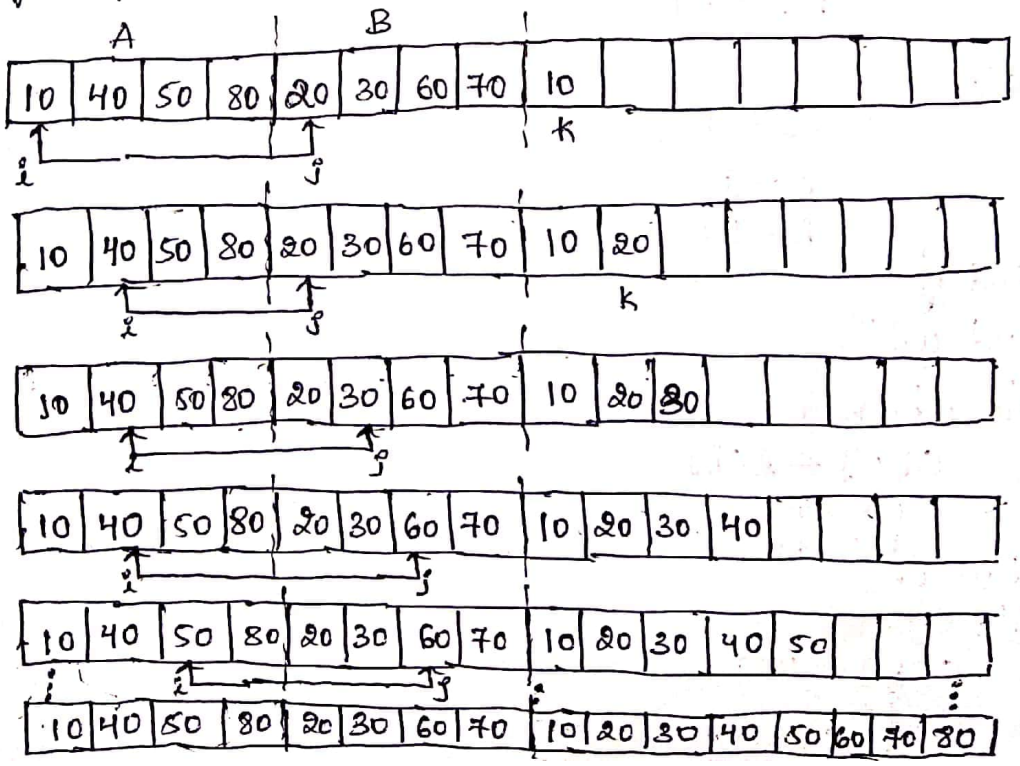
* Step 5 :- Merge the left part of Array & right part of Array

Simple Merge (a, low, mid, high)

Two Indexes are initialized to point the first Elements of two Arrays. Now their positional values are compared & Smaller Element is copied into a third resulting Array.

→ This process is continued till Either of two Arrays is completed.

Then the remaining Elements of Non-computed Array are directly copied into resulting Array.



* Algorithm Mergesort (a, low, high)

I/p :- Unsorted Array A[0----n-1]

O/p :- Sorted Array A[0----n-1]

```
if (low > high) return
mid ← (low+high)/2
Mergesort (a, low, mid)
Mergesort (a, mid+1, high)
Simple Merge (a, low, mid, high)
```

* Algorithm Simple Merge (A, B, C, m, n)

I/p :- A is a Sorted Array with m Elements A[0----m]
B is a Sorted Array with n Elements B[0----n]

O/p :- C is a Sorted Array obtained After Merging A & B

```
i ← j ← k ← 0
while (i < m and j < n)
    if (A[i] < B[j]) then
            C[k] ← A[i]
              i ← i+1
              k ← k+1
    Else
            C[k] ← B[j]
              j ← j+1
              k ← k+1
    End if
    End while
while (i < m)
        C[k] ← A[i]
          i ← i+1
          k ← k+1
End while
while (j < n)
        C[k] ← B[j]
          j ← j+1
          k ← k+1
```

End while

* Analysis :-

* The Input parameter is n
* The Basic operation is "Comparison"
* The Recurrence Relation can be given as

$$C(n) = \begin{cases} 0 & \text{if } n = 1 \\ C(n/2) + C(n/2) + n & \text{if } n > 1 \end{cases}$$

Here, two terms $C(n/2)$ indicate the time required for Sorting two halves of given array & $C(n)$ denotes the time required for Merging two arrays.

$$C(n) = C(n/2) + C(n/2) + n$$

$$= 2C(n/2) + n \qquad \text{i.e } C(n/2) = 2C(n/2^2) + n/2$$

$$= 2[2C(n/2^2) + n/2] + n$$

$$= 2^2 C(n/2^2) + 2n$$

$$\vdots$$

$$= 2^k C(n/2^k) + kn$$

Assume that $2^k = n$ ---------- (1)

$$\therefore C(n) = nC(n/n) + kn$$

$$= nC(1) + kn$$

$$= n(0) + kn$$

$$= kn \text{ ---------- (2)}$$

Apply $\log_2$ on both side of Equn (1)

$$\log_2 2^k = \log_2 n$$

$$k \log_2^2 = \log_2^n$$

$$\boxed{k = \log_2^n}$$

Substitute 'k' in Equation (2)

$$C(n) = n \log_2^n$$

∴ Time complexity of Merge sort is $\theta(n \log_2^n)$ //

* Advantages of Mergesort

→ Mergesort Algorithm is Stable Algorithm
→ It can be Applied to files of Any Size
→ It can be used for Internal Sorting using arrays in Main Memory.
→ Time complexity is n.logn which is very Efficient.

* Disadvantages of Mergesort

→ The Algorithm uses Extra Space proportional to N.
→ It uses more Memory on Stack because of recursion.

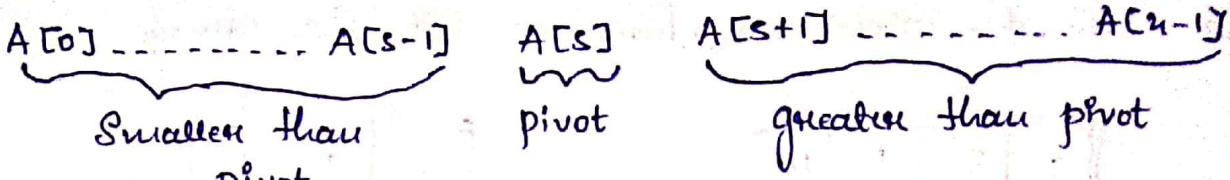
* Quicksort :-

is another important Sorting Algorithm that is based on Divide-&-Conquer Approach. unlike Mergesort, which divides its input elements According to their position in Array. Quicksort divides them According to their values.

→ Quicksort rearranges Elements of given Array A[0...n-1] to achieve its partition

A situation where all the Elements before some partition 's' are Smaller than or Equal to A[s] & all the elements after partition s are greater than or Equal to A[s]

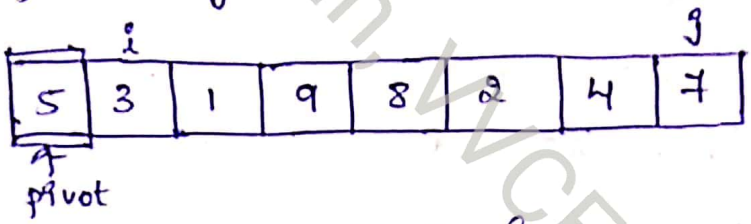
→ After first partition, the Array of n Elements May look like



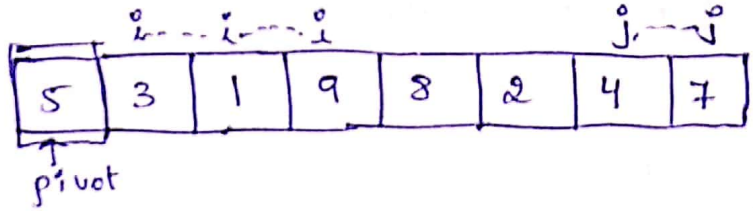
→ The procedure of partitioning the given Array is as shown

* Usually, the first Element of the given Array is treated as "pivot"

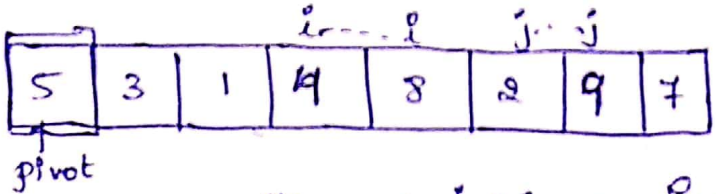
The partition of Second Element will be first Index Variable 'i' & the position of last Element will be Index Variable 'j'



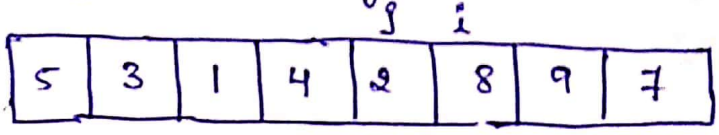
* Now, the Index Variable 'i' is incremented by one till the value Stored at position 'i' is greater than pivot.
Similarly 'j' is decremented by one till the value Stored at 'j' is Smaller than pivot.



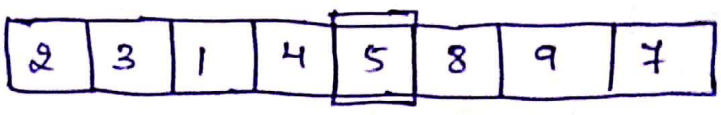
* Now, the two Elements A[i] & A[j] are Interchanged (Swap).



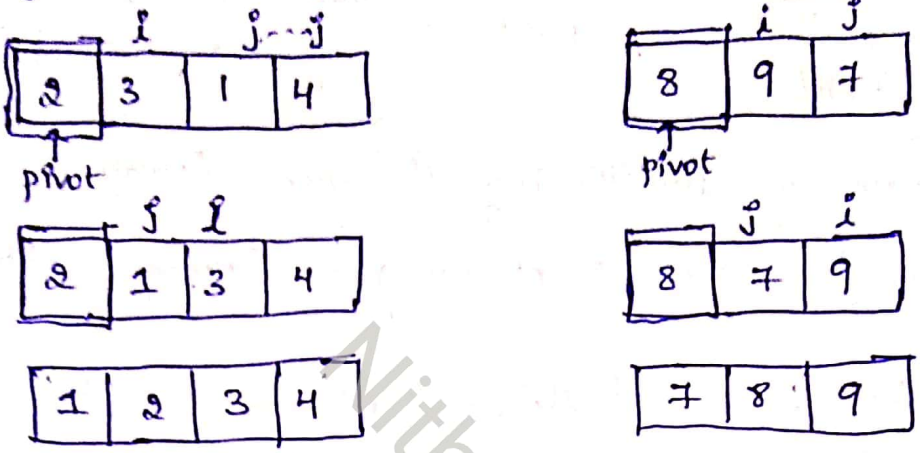
* Again the Current position of i & j are incremented & decremented respectively & Exchange are made if required



Finally, when i & j crossover, Swap A[j] & pivot



→ Apply the above procedure on both Sub Arrays



* Algorithm Quicksort (A[l---r])

I/P :- A array A[l---r]

O/P :- Sorted Array A[l---r]

if $l < r$

$s \leftarrow$ partition (A[l---r])

Quicksort (A[l----s-1])

Quicksort (A[s+1----r])

* Algorithm partition (A[l---r])

I/P :- A array A[l---r]

O/P :- A partition of A[l---r]

pivot $\leftarrow$ A[l]

$i \leftarrow l$

$j \leftarrow r+1$

Repeat $i \leftarrow i+1$ until $A[i] \geq$ pivot

Repeat $j \leftarrow j-1$ until $A[j] \leq$ pivot

Swap (A[i], A[j])

until ($i \geq j$)

Swap (A[pivot], A[j])

Return j

* Analysis :-

* The Input parameter is 'n'
* The Basic operation is "Comparison"
* The Recurrence relation can be given as

$$C(n) = \begin{cases} 0 & \text{if } n = 1 \\ C(n/2) + C(n/2) + n & \text{if } n > 1 \end{cases}$$

Here, two terms $C(n/2)$ indicates the time required for Sorting two halves of given Array & $C(n)$ denotes the time required to partition n Elements

$$C(n) = C(n/2) + C(n/2) + n$$

[Apply Same Procedure given in Mergesort; because Analysis of Mergesort & Quicksort are Same]

$$C(n) = n \log_2^n$$

∴ Time Complexity of Quicksort is $\Theta(n \log_2^n)$ //

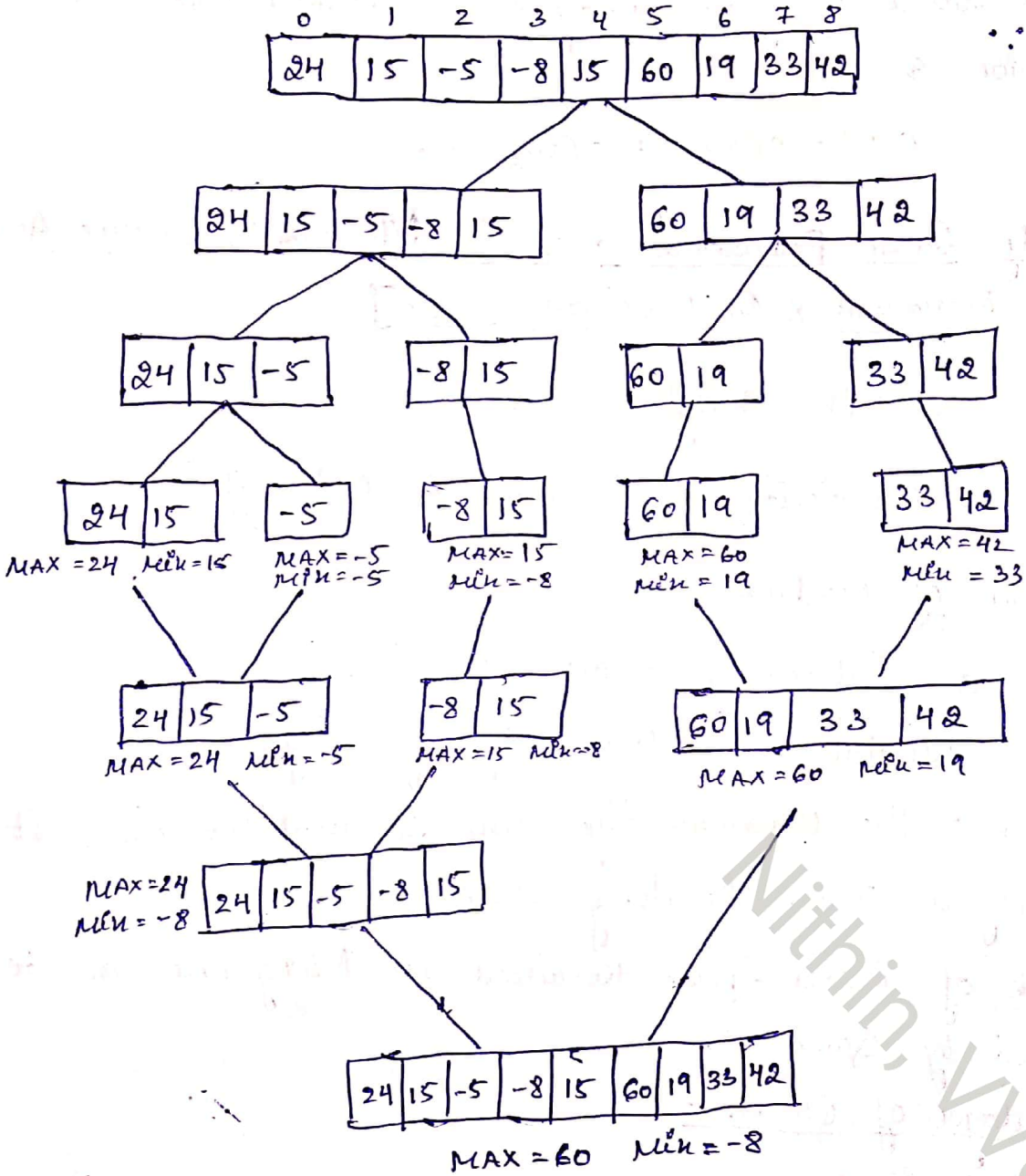
* Advantages of Quicksort
→ It has an Extremely Short Inner loop.
→ Quicksort requires time complexity of $n \log n$
→ In-place : The Quicksort Algorithm is in-place since it uses only a Small Auxillary Stack.
→ Drawback of Extra-Space required in Mergesort can be overcomed by Quicksort.

* Disadvantages of Quicksort
→ It is not Stable
→ It is very difficult for beginners to grasp. so this Algo requires a higher level of knowledge for programming.

* Finding Maximum & Minimum Using Divide & Conquer

In this Approach, the Array is divided into two halves. Then using recursive Approach Maximum & Minimum numbers in Each halves are found.
Later, return the Maximum of two max of Each half & the Minimum of two Minima of Each half.

→ Consider the foll Example problem



∴ Maximum = 60
Minimum = -8 //

* Algorithm MAX-MIN (int a[], int i, int j, max, min)

I/p : List of Nos a[0...n-1]

O/p : Max & Min value in a

```
if (i == j)
        max = min = a[i];                    } for n = 1
Else if (i = j-1)
        if (a[i] > a[j])
                max = a[i], min = a[j]       } for n = 2
        Else
                max = a[j], min = a[i]
        End if
End if
mid = (i + j/2);
max_min (a, i, mid, max1, min1)              } for n > 2
max_min (a, mid+1, j, max2, min2)
if (max1 > max2)
                return max1;
        Else
                return max2;                 } Final comparison b/w results of left & right half.
if (min1 < min2)
                return min1;
        Else
                return min2;
End if
End if
```

* Analysis :-

* The Algorithm is directly dependent on I/p 'n'
* The Basic operation is 'comparison'.
* The Recurrence Equation is given by

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

Here, two terms $T(n/2)$ indicate the time required for Max & Min Calculation in two halves of given Array & '2' denote the time required for calculating Max among the max1 & Max2, Min among the min1 & min2.

$\therefore T(n) = 2T(n/2) + 2$

$= 2^2 T(n/2^2) + 2^2 + 2$ $\because T(n/2) = 2T(n/2^2) + 2$

$= 2^3 T(n/2^3) + 2^3 + 2^2 + 2$

$\vdots$

$= 2^{i-1} T\left(n/2^{i-1}\right) + 2^{i-1} + 2^{i-2} \dots\dots 2^3 + 2^2 + 2$

Assume $n = 2^i$

$\therefore T(n) = 2^{i-1} T\left(2^i/2^{i-1}\right) + 2^{i-1} + 2^{i-2} \dots\dots 2^3 + 2^2 + 2$

$= 2^{i-1} T(2) + 2^{i-1} + 2^{i-2} \dots\dots 2^3 + 2^2 + 2$

$= 2^{i-1}(1) + 2^{i-1} + 2^{i-2} \dots\dots 2^3 + 2^2 + 2$ $\because T(2) = 1$

$= \frac{2^i}{2} + 2^{i-1} + 2^{i-2} \dots\dots 2^3 + 2^2 + 2$

$= \frac{n}{2} + 2^{i-1} + 2^{i-2} \dots\dots 2^3 + 2^2 + 2$ $\because 2^i = n$

$= \frac{n}{2} + a\left[\frac{r^n - 1}{r - 1}\right]$

$= \frac{n}{2} + 2\left[\frac{2^{i-1} - 1}{2 - 1}\right]$ $\because a = 2,\ r = 2,\ n = i-1$

$= n/2 + 2^i - 2$

$= n/2 + n - 2$ $\because 2^i = n$

$$T(n) = \frac{3n}{2} - 2$$

* Strassen's Matrix Multiplication

Matrix Multiplication is usually done by brute-force Approach, which will take 8 Multiplications & 4 Additions for 2×2 Matrix.

→ An Algorithm developed by V. Strassen for Matrix Multiplication will reduce the no of Multiplication & reducing the Execution time.

→ The formula is as below

$$\begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} * \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix}$$

$$= \begin{bmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{bmatrix}$$

Here, $m_1 = (a_{00} + a_{11}) * (b_{00} + b_{11})$

$m_2 = (a_{10} + a_{11}) * b_{00}$

$m_3 = a_{00} * (b_{01} - b_{11})$

$m_4 = a_{11} * (b_{10} - b_{00})$

$m_5 = (a_{00} + a_{01}) * b_{11}$

$m_6 = (a_{10} - a_{00}) * (b_{00} + b_{01})$

$m_7 = (a_{01} - a_{11}) * (b_{10} + b_{11})$

Thus, Strassen's algorithm takes only 7 Multiplications & 18 Additions for 2×2 Matrix.

→ let T(n) be the total Multiplications required for two n×n Matrices we have

$$T(n) = \begin{cases} 7\,T(n/2) & n > 1 \\ 1 & n = 1 \end{cases}$$

$\therefore T(n) = 7\,T(n/2)$

$= 7\{7\,T(n/2^2)\}$ $\quad \because T(n/2) = 7\,T(n/2^2)$

$= 7^2\,T(n/2^2)$

$\vdots$

$= 7^k\,T(n/2^k)$

Assume $n = 2^k$

$\therefore T(n) = 7^k\,T(n/n)$

$= 7^k\,T(1)$

$T(n) = 7^k$ ---------- (1)

Apply $\log_2$ on both side of $n = 2^k$

$\log_2 2^k = \log_2 n$

$k \log_2 2 = \log_2 n$

$\boxed{k = \log_2 n}$

Substitute k value in Equation (1)

$T(n) = 7^{\log_2 n}$

$= n^{\log_2 7}$

$= n^{2.807}$

$\approx n^3$ //

* Using divide & Conquer Method, Multiply the foll two Matrices with the help of Strassen's Matrix Multiplication

$$\begin{bmatrix} 1 & 2 \\ 5 & 6 \end{bmatrix} \quad \begin{bmatrix} 8 & 7 \\ 1 & 2 \end{bmatrix}$$

⇒ let $A = \begin{bmatrix} 1 & 2 \\ 5 & 6 \end{bmatrix} = \begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix}$: $A_{00} = 1$ $A_{01} = 2$ $A_{10} = 5$ $A_{11} = 6$

$B = \begin{bmatrix} 8 & 7 \\ 1 & 2 \end{bmatrix} = \begin{bmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{bmatrix}$: $B_{00} = 8$ $B_{01} = 7$ $B_{10} = 1$ $B_{11} = 2$

$m_1 = (A_{00} + A_{11}) * (B_{00} + B_{11}) = (1+6)(8+2) = 70$//

$m_2 = (A_{10} + A_{11}) * B_{00} = (5+6)(8) = 88$//

$m_3 = A_{00} * (B_{01} - B_{11}) = 1(7-2) = 5$//

$m_4 = A_{11} * (B_{10} - B_{00}) = 6(1-8) = -42$//

$m_5 = (A_{00} + A_{01}) * B_{11} = (1+2)2 = 6$//

$m_6 = (A_{10} - A_{00}) * (B_{00} + B_{01}) = (5-1)(8+7) = 60$//

$m_7 = (A_{01} - A_{11}) * (B_{10} + B_{11}) = (2-6)(1+2) = -12$//

→ The Final Matrix is given by

$$C = \begin{bmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_4 + m_3 - m_2 + m_6 \end{bmatrix}$$

$$C = \begin{bmatrix} 70 - 42 - 6 - 12 & 5 + 6 \\ 88 - 42 & 70 + 5 - 88 + 60 \end{bmatrix}$$

$$C = \begin{bmatrix} 10 & 11 \\ 46 & 47 \end{bmatrix}$$ //

* Master Method to Solve Recurrences

Master Method is a direct way to get the solution for given recurrence. The Master Method work only for following type of recurrences or for any recurrence that can be transformed to following type

$$\boxed{T(n) = a\,T(n/b) + f(n)}$$ where $a >= 1$ & $b > 1$

→ There are following three cases

* If $a > b^d$ then $T(n) = \Theta(n^{\log_b^a})$
* If $a = b^d$ then $T(n) = \Theta(n^d \log n)$
* If $a < b^d$ then $T(n) = \Theta(n^d)$

Master Method is Mainly derived from Recurrence tree Method.

* $T(n) = T(n/2) + 1$

⇒ $a = 1$ $b = 2$ $1 = n^0 = n^d$ i.e $d = 0$

Compare a & b^d

$1 = 2^0$

$1 = 1$

$\therefore \Theta(n^d \log n)$

$= \Theta(n^0 \log n)$

$T(n) = \Theta(\log n)$ //

* $T(n) = 8\,T(n/2) + \Theta(n^2)$

⇒ $a = 8$ $b = 2$ $d = 0$ Because $n^0 = 1 = n^d$

Compare a & b^d

$8 > 2^0$

$\therefore \Theta(n^{\log_b^a})$

$= \Theta(n^{\log_2^8})$

$= \Theta(n^3)$ //

* $T(n) = T(2n/3) + 1$

$\Rightarrow a = 1 \quad b = 3/2 \quad d = 0$ Because $1 = n^0 = n^d$

Compare a & b^d

$1 = (3/2)^0$

$\therefore \Theta(n^d \log n)$

$= \Theta(n^0 \log n)$

$T(n) = \Theta(\log n)$ //

* $T(n) = 3T(n/4) + n\log n$

$\Rightarrow a = 3 \quad b = 4 \quad d = 1$ Because $n = n^1 = n^d$

Compare a & b^d

$3 < 4^1$

$\therefore T(n) = \Theta(n^d)$

$= \Theta(n^1)$

$T(n) = \Theta(n)$ //

* $T(n) = 9T(n/3) + n$

$\Rightarrow a = 9 \quad b = 3 \quad d = 1$ Because $n = n^1 = n^d$

Compare a & b^d

$9 > 3^1$

$\therefore \Theta(n^{\log_b^a})$

$= \Theta(n^{\log_3^9})$

$T(n) = \Theta(n^3)$ //