Mater Theorem

Master's Theorem is a popular method for solving the recurrence relations.

Master's theorem solves recurrence relations of the form-

$$T(n) = a\,T(n/b) + \theta\,(n_k \log_p n).$$

Here, a >= 1, b > 1, k >= 0 and p is a real number.

Master Theorem Cases-

To solve recurrence relations using Master's theorem, we compare a with b_k.

Then, we follow the following cases-

Case-01:

If $a > b^k$, then $T(n) = \theta$

$(n^{\log_{ba}})$

Case-02:

If $a = b_k$ and

- If $p < -1$, then $T(n) = \theta\,(n^{\log_{ba}})$
- If $p = -1$, then $T(n) = \theta\,(n^{\log_{ba}}.\log^2 n)$
- If $p > -1$, then $T(n) = \theta\,(n^{\log_{ba}}.\log^{p+1} n)$

Case-03:

If $a < b_k$ and

- If $p < 0$, then $T(n) = O\,(n_k)$
- If $p >= 0$, then $T(n) = \theta\,(n_k \log_p n)$

PRACTICE PROBLEMS BASED ON MASTER THEOREM-

Problem-01:

Solve the following recurrence relation using Master's theorem- T(n)

= 3T(n/2) + n_2

Solution-

We compare the given recurrence relation with T(n) = aT(n/b) + θ ($n_k \log_p n$). Then,

we have-

a = 3

b = 2

k = 2

p = 0

Now, a = 3 and $b_k = 2_2 = 4$.

Clearly, a < b_k.

So, we follow case-03.

Since p = 0, so we have-

T(n) = θ ($n_k \log_p n$)

T(n) = θ ($n_2 \log_0 n$) Thus,

$\boxed{T(n) = \theta\ (n_2)}$

Problem-02:

Solve the following recurrence relation using Master's theorem- T(n)

= 2T(n/2) + nlogn

Solution-

We compare the given recurrence relation with T(n) = aT(n/b) + θ ($n_k \log_p n$). Then,

we have-

a = 2

b = 2

k = 1

p =1

Now, $a = 2$ and $b_k = 2_1 = 2$. Clearly,

$a = b_k$.

So, we follow case-02.

Since $p = 1$, so we have-

$T(n) = \theta (n_{\log_b a}.\log_{p+1} n)$ T(n)

$= \theta (n_{\log_2 2}.\log_{1+1} n)$

Thus,

$$\boxed{T(n) = \theta (n\log_2 n)}$$

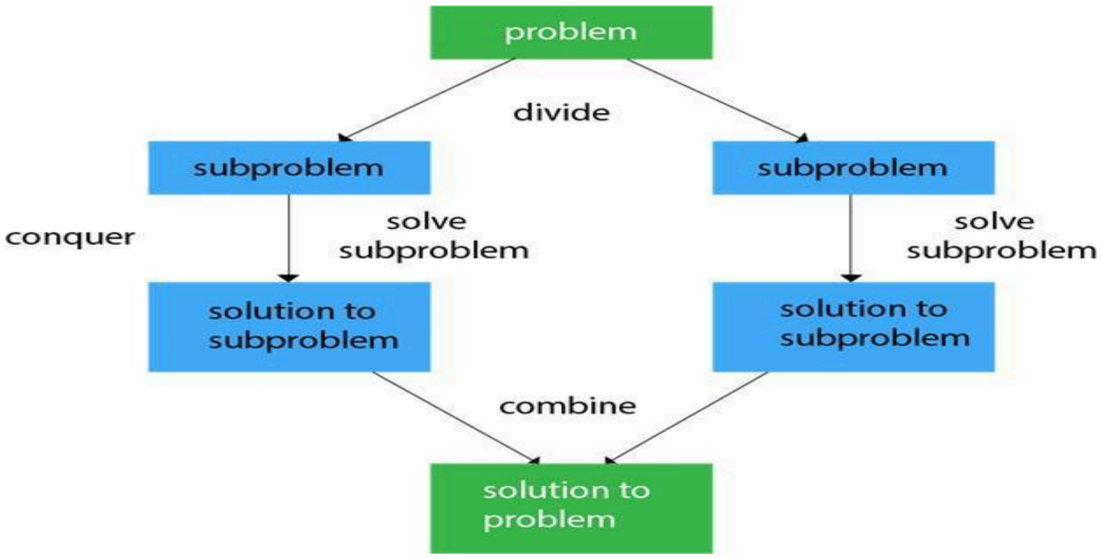
Divide-and-conquer method: Divide-and-conquer is probably the best known general algorithm design technique. The principle behind the Divide-and-conquer algorithm design technique is that it is easier to solve several smaller instance of a problem than the larger one.

The "divide-and-conquer" technique involves solving a particular problem by dividing it into one or more cub-problems of smaller size, recursively solving each sub-problem and then "merging" the solution of sub-problems to produce a solution to the original problem.

Divide-and-conquer algorithms work according to the following general plan.

1. **Divide:** Divide the problem into a number of smaller sub-problems ideally of about the same size.
2. **Conquer:** The smaller sub-problems are solved, typically recursively. If the sub-problem sizes are small enough, just solve the sub-problems in a straight forward manner.
3. **Combine:** If necessary, the solution obtained the smaller problems are connected to get the solution to the original problem.

The following figure shows-



Control abstraction for divide-and-conquer technique:

Control abstraction means a procedure whose flow of control is clear but whose primary operations are satisfied by other procedure whose precise meanings are left undefined.

```
Algorithm DandC(p)
 {
    if small (p) then



    return S(p)
    else
      {
        Divide P into small instances $P_1$, $P_2$, $P_3$........$P_k$, $k \geq 1$;
        Apply DandC to each of these sub-problems;\
        return combine (DandC($P_1$), DandC($P_1$),.... (DandC($P_k$);
      }
 }
```

Algorithm: Control abstraction for divide-and-conquer

DandC(p) is the divide-and-conquer algorithm, where P is the problem to be solved. Small(p) is a Boolean valued function(i.e., either true or false) that determines whether the input size is small enough that the answer can be computed without splitting. If this, is so the function S is invoked. Otherwise the problem P is divided into smaller sub-problems. These sub-problems P_1, P_2, P_3……..P_k, are solved by receive applications of DandC.

Combine is a function that combines the solution of the K sub-problems to get the solution for original problem 'P'.

Example: Specify an application that divide-and-conquer cannot be applied.

Solution: Let us consider the problem of computing the sum of n numbers $a_0, a_1, \ldots . a_{n-1}$. If n>1, we divide the problem into two instances of the same problem. That is to compute the sum of the first [n/2] numbers and to compute the sum of the remaining [n/2] numbers. Once each of these two sum is compute (by applying the same method recursively), we can add their values to get the sum in question-

$$a_0 + a_1 + \ldots + a_{n-1} = (a_0 + a_1 + \ldots + a_{[n/2]-1}) + a_{[n/2]-1} + \ldots\ldots + a_{n-1}).$$

For example, the sum of 1 to 10 numbers is as follows-

$$\begin{aligned}(1+2+3+4+\ldots\ldots+10) &= (1+2+3+4+5)+(6+7+8+9+10)\\ &= [(1+2) + (3+4+5)] + [(6+7) + (8+9+10)]\\ &= \ldots\\ &= \ldots\\ &= (1) + (2) + \ldots\ldots + (10).\end{aligned}$$

This is not an efficient way to compute the sum of n numbers using divide-and-conquer technique. In this type of problem, it is better to use brute-force method.

Applications of Divide-and Conquer: The applications of divide-and-conquer methods are-

1. Binary search.
2. Quick sort
3. Merge sort.

Binary Search:

Binary search is an efficient searching technique that works with only sorted lists. So the list must be sorted before using the binary search method. Binary search is based on divide-and-conquer technique.

The process of binary search is as follows:

The method starts with looking at the middle element of the list. If it matches with the key element, then search is complete. Otherwise, the key element may be in the first half or second half of the list. If the key element is less than the middle element, then the search continues with the first half of the list. If the key element is greater than the middle element, then the search continues with the second half of the list. This process continues until the key element is found or the search fails indicating that the key is not there in the list.

Consider the list of elements: -4, -1, 0, 5, 10, 18, 32, 33, 98, 147, 154, 198, 250, 500.

Trace the binary search algorithm searching for the element -1.

Sol: The given list of elements are:

Low														High
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
-4	-1	0	5	10	18	27	32	33	98	147	154	198	250	500

Searching key '-1': Here the key to search is '-1'

First calculate mid;

Mid = (low + high)/2

= (0 + 14) /2 = 7

Low							Mid							High
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
-4	-1	0	5	10	18	27	32	33	98	147	154	198	250	500

$\longleftarrow$ First Half $\longrightarrow$ $\longleftarrow$ Second Half $\longrightarrow$

Here, the search key -1 is less than the middle element (32) in the list. So the search process continues with the first half of the list.

Low						High								
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
-4	-1	0	5	10	18	27	32	33	98	147	154	198	250	500

Now mid = (0+6)/2
=3.

Low			Mid			High								
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
-4	-1	0	5	10	18	27	32	33	98	147	154	198	250	500

$\longleftarrow$ First Half $\longrightarrow$ $\longleftarrow$ Second Half $\longrightarrow$

The search key '-1' is less than the middle element (5) in the list. So the search process continues with the first half of the list.

Low		High												
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
-4	-1	0	5	10	18	27	32	33	98	147	154	198	250	500

Now mid= (0+2)/2
=1

Low	Mid	High												
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
-4	-1	0	5	10	18	27	32	33	98	147	154	198	250	500

Here, the search key -1 is found at position 1.

The following algorithm gives the *iterative binary Search Algorithm*

```
Algorithm BinarySearch(a, n, key)
 {
   // a is an array of size n elements
   // key is the element to be searched
   // if key is found in array a, then return j, such that
   //key = a[i]
   //otherwise return -1.
     Low: = 0;
     High: = n-1;
    While (low ≤ high) do
      {
         Mid: = (low + high)/2;
         If ( key = a[mid]) then
              Return mid;
         Else if (key < a[mid])
             {
               High: = mid +1;
             }
        Else if( key > a[mid])
            {
              Low: = mid +1;
             }
 }
```

The following algorithm gives *Recursive Binary Search*

```
  Algorithms Binsearch ( a, n, key, low, high)
        {
         // a is array of size n
        // Key is the element to be searched
       // if key is found then return j, such that key = a[i].
      //otherwise return -1
If ( low ≤ high) then
  {
    Mid: = (low + high)/2;
         If ( key = a[mid]) then
             Return mid;
         Else if (key < a[mid])
             Binsearch ( a, n, key, low, mid-1);
         Else if ( key > a[mid])
             Binsearch ( a, n, key, mid+1, high);
    }
  Return -1;
}
```

The recursive relation for above recursive relation is

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

To perform binary search time complexity analysis, we apply the master theorem to the equation and get O(log n).

Advantages of Binary Search: The main advantage of binary search is that it is faster than sequential (linear) search. Because it takes fewer comparisons, to determine whether the given key is in the list, then the linear search method.

Disadvantages of Binary Search: The disadvantage of binary search is that can be applied to only a sorted list of elements. The binary search is unsuccessful if the list is unsorted.

Efficiency of Binary Search: To evaluate binary search, count the number of comparisons in the best case, average case, and worst case.

<u>Best Case:</u> The best case occurs if the middle element happens to be the key element. Then only one comparison is needed to find it. Thus the efficiency of binary search is O(1).

Ex: Let the given list is: 1, 5, 10, 11, 12.

Low		Mid		High
1	5	$\boxed{10}$	11	12

Let key = 10.
Since the key is the middle element and is found at our first attempt.

The recursive relation for above recursive relation is

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

The recurrence relation equation is T(n)=a T(n/b)+f(n)

Compare the equation T(n)=T(n/2)+1

To perform binary search time complexity analysis, we apply the master theorem to the equation and get O(log n).

Worst Case: Assume that in worst case, the key element is not there in the list. So the process of divides the list in half continues until there is only one item left to check.

Items left to search	*Comparisons so far*
16	0
8	1
4	2
2	3
1	4

For a list of size 16, there are 4 comparisons to reach a list of size one, given that there is one comparison for each division, and each division splits the list size in half.

In general, if n is the size of the list and c is the number of comparisons, then

$n/(2_k)=1$

we can rewrite it as -

$2_k=n$

by taking log both sides, we get

$k=\log_2 n$

So, in average and worst case, time complexity of binary search algorithm is log(n).

$C = \log_2 n$

$\therefore$ Eficiency in worst case = O(log n)

Average Case: In binary search, the average case efficiency is near to the worst case efficiency. So the average case efficiency will be taken as O(log n).

$\therefore$Efficiency in average case = O (log n).

Binary Search	
Best Case	O(1)
Average Case	O(log n)
Worst Case	O(log n)

Merge Sort-

Merge sort is a famous sorting algorithm.

It uses a divide and conquer paradigm for sorting.

It divides the problem into sub problems and solves them individually.

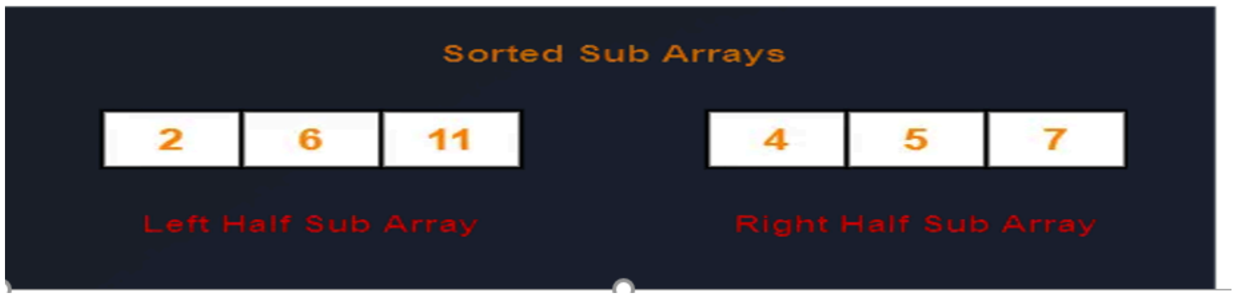
It then combines the results of sub problems to get the solution of the original problem.

Working of Merge sort:-

Before learning how merge sort works, let us learn about the merge procedure of merge sort algorithm.

The merge procedure of merge sort algorithm is used to merge two sorted arrays into a third array in sorted order.

Consider we want to merge the following two sorted sub arrays into a third array in sorted order-



Merge Sort Algorithm-

```
// L : Left Sub Array , R : Right Sub Array , A : Array


merge(L, R, A)

{

   nL = length(L)    // Size of Left Sub Array

   nR = length(R)    // Size of Right Sub Array


   i = j = k = 0


   while(i<nL && j<nR)

   {

    /* When both i and j are valid i.e. when both the sub arrays have elements to insert in A */
    if(L[i] <= R[j])
    {
      A[k] = L[i]
      k = k+1
      i = i+1
    }
    else
    {
      A[k] = R[j]
      k = k+1
      j = j+1
    }
  }

  // Adding Remaining elements from left sub array to array A
  while(i<nL)
  {
    A[k] = L[i]
    i = i+1
    k = k+1
  }

  // Adding Remaining elements from right sub array to array A
  while(j<nR)
  {
    A[k] = R[j]
    j = j+1
```

```
      k = k+1
   }
}
```

Merge Sort Algorithm works in the following steps-

It divides the given unsorted array into two halves- left and right sub arrays.

The sub arrays are divided recursively.

This division continues until the size of each sub array becomes 1.

After each sub array contains only a single element, each sub array is sorted trivially.

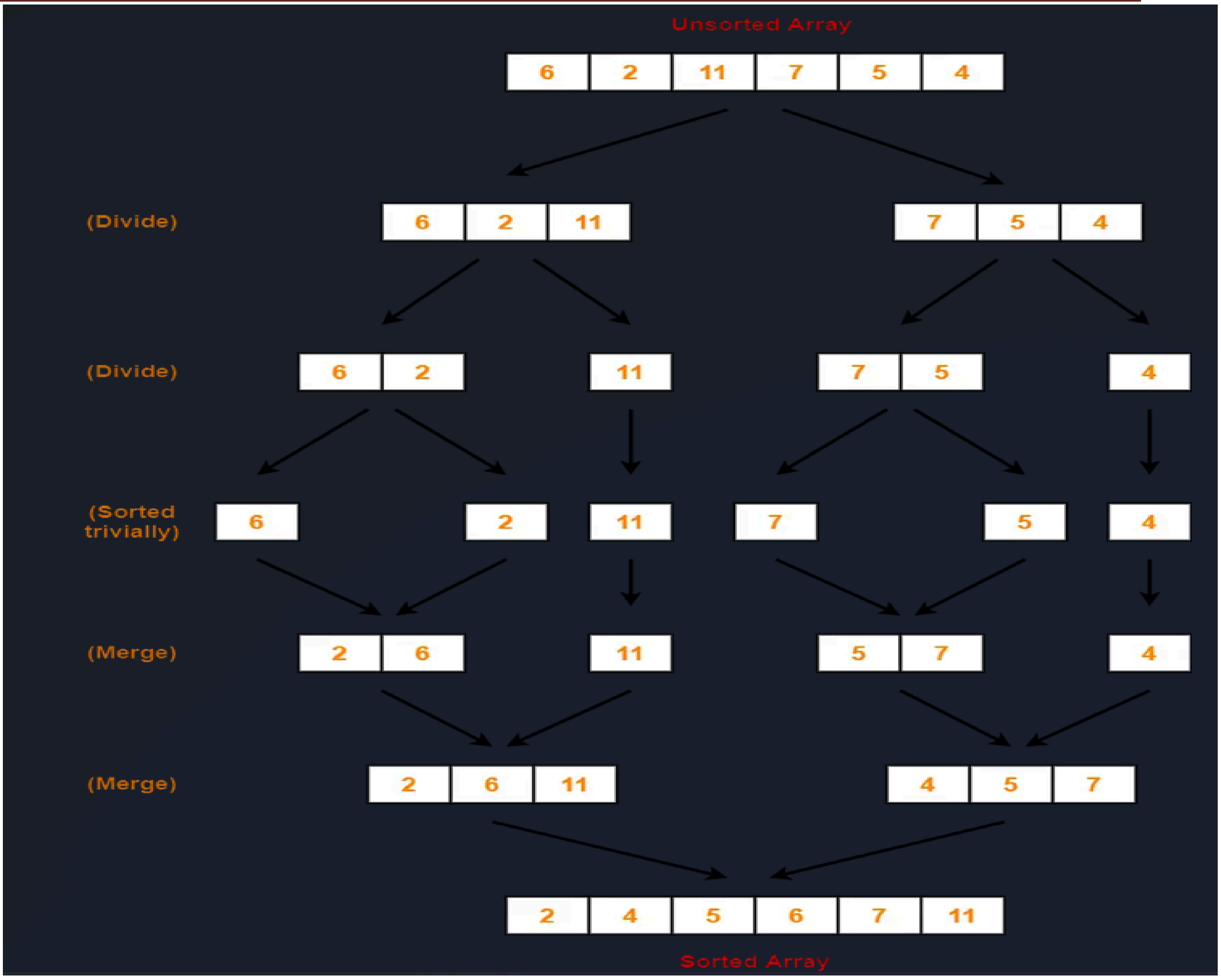
Then, the above discussed merge procedure is called.

The merge procedure combines these trivially sorted arrays to produce a final sorted array.

Consider the following elements have to be sorted in ascending order-

6, 2, 11, 7, 5, 4

The merge sort algorithm works as-



Time Complexity Analysis-

In merge sort, we divide the array into two (nearly) equal halves and solve them recursively using merge sort only.

So, we have-

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

$$n_L = \text{Left Half}$$

$$n_R = \text{Right Half}$$

$$n_L \approx n_R$$

Finally, we merge these two sub arrays using merge procedure which takes Θ(n) time as explained above.

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

If T(n) is the time required by merge sort for sorting an array of size n, then the recurrence relation for time complexity of merge sort is-

Let's analyze the time complexity of Merge Sort using the Master Theorem:

T(n) = 2T(n/2) + O(n)

Comparing with the Master Theorem form:

T(n) = aT(n/b) + θ(n^k log^p n)

We get:

a = 2

b = 2

k = 1

p = 0

Since a = b^k (2 = 2^1), we apply Case 2 of the Master Theorem.

Case 2: a = b^k

p = 0 ≥ -1, so we use the formula:

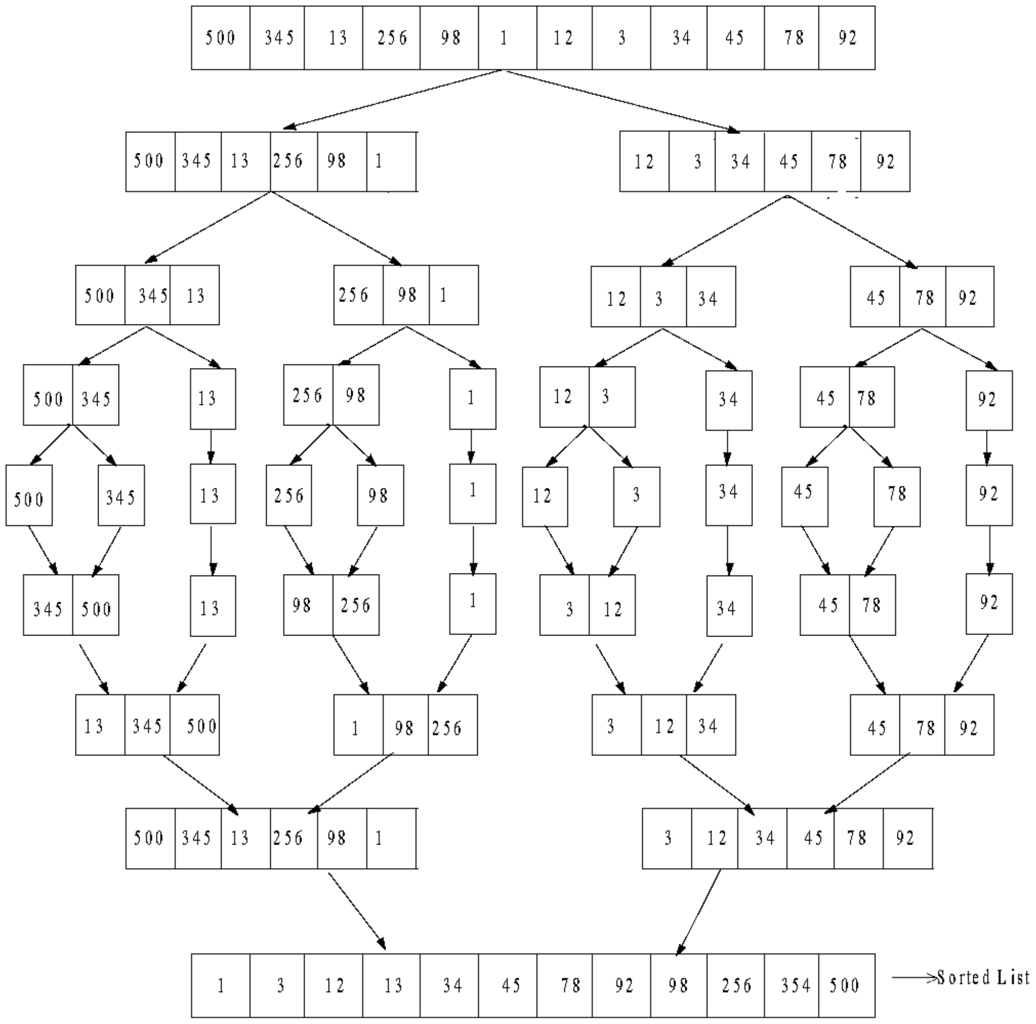
T(n) = θ(n^log_b a log^(p+1) n)

= θ(n^log_2 2 log^1 n)

= θ(n^1 log n)

= θ(n log n)

Ex: Let the list is: - 500, 345, 13, 256, 98, 1, 12, 3, 34, 45, 78, 92.



The merge sort algorithm works as follows-

Step 1: If the length of the list is 0 or 1, then it is already sorted, otherwise,

Step 2: Divide the unsorted list into two sub-lists of about half the size.

Step 3: Again sub-divide the sub-list into two parts. This process continues until each element in the list becomes a single element.

Step 4: Apply merging to each sub-list and continue this process until we get one sorted list.

Efficiency of Merge List: Let 'n' be the size of the given list/ then the running time for merge sort is given by the recurrence relation.

$$T(n) = \begin{cases} a & \text{if } n=1,\ a \text{ is a constant} \\ 2T(n/2) + Cn & \text{if } n>1,\ C \text{ is constant} \end{cases}$$

Assume that 'n' is a power of 2 i.e. $n=2^k$.
This can be rewritten as $k=\log_2 n$.

$$\text{Let } T(n) = 2T(n/2) + Cn \text{ ——— } \textcircled{1}$$

We can solve this equation by using successive substitution.

Replace n by n/2 in equation, ① ,we get

$$T(n/2) = 2T(n/4) + \frac{Cn}{2} \text{ ——— } \textcircled{2}$$

$$\text{Thus, } T(n) = 2\left(2T(n/4) + \frac{Cn}{2}\right) + Cn$$

$$= 4T(n/4) + 2Cn$$

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

...

...

...

$$= 2^k T(1) + KCn \quad \left(\because k = \log_2 n\right)$$

$$= an + Cn\log n$$

$$\therefore T(n) = O(n\log n)$$

As a result, Merge Sort's time complexity is insensitive to the input's initial order, making it:

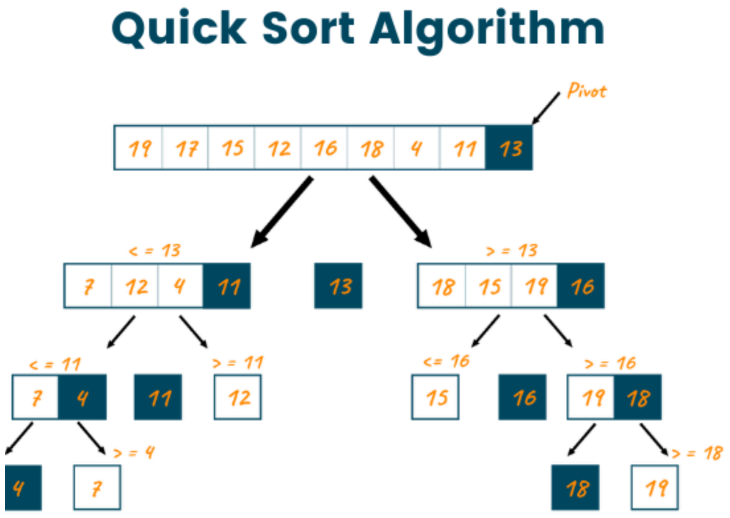
- Best-case: O(n log n) when the input is already sorted or nearly sorted.
- Average-case: O(n log n) for randomly ordered inputs.
- Worst-case: O(n log n) even for reverse-sorted or nearly reverse-sorted inputs.

Quick Sort:

The quick sort is considered to be a fast method to sort the elements. It was developed by CAR Hoare. This method is based on divide-and-conquer technique i.e. the entire list is divided into various partitions and sorting is applied again and again on these partitions. This method is also called as partition exchange sorts. The quick sort can be illustrated by the following

Working in quicksort:

1. Choose a pivot element from the array.
2. Partition the array around the pivot:
   - Elements less than pivot go to the left subarray.
   - Elements greater than pivot go to the right subarray.
3. Recursively apply Quick Sort to the left and right subarrays.



Example: - 54, 26, 93, 17, 77, 31, 44, 55, 20

Step 1: Choose a pivot element

- Array: [54, 26, 93, 17, 77, 31, 44, 55, 20]
- Pivot: 54 (chosen arbitrarily)

Step 2: Partition the array

- Elements less than 54: [26, 17, 31, 20]
- Pivot: 54
- Elements greater than 54: [93, 77, 55, 44]

Step 3: Recursively apply Quick Sort to the left subarray

- Left subarray: [26, 17, 31, 20]
- Pivot: 26 (chosen arbitrarily)
- Partitioning: [17, 20] | 26 | [31]
- Recursively sort: [17, 20] and [31]

Step 4: Recursively apply Quick Sort to the left subarray of the left subarray

- Left subarray: [17, 20]
- Pivot: 17 (chosen arbitrarily)
- Partitioning: [] | 17 | [20]
- Recursively sort: [20] (only one element, so return)

Step 5: Combine the results of the left subarray

- Sorted left subarray: [17, 20, 26, 31]

Step 6: Recursively apply Quick Sort to the right subarray

- Right subarray: [93, 77, 55, 44]
- Pivot: 93 (chosen arbitrarily)
- Partitioning: [] | 93 | [77, 55, 44]
- Recursively sort: [77, 55, 44]

Step 7: Recursively apply Quick Sort to the right subarray of the right subarray

- Right subarray: [77, 55, 44]
- Pivot: 77 (chosen arbitrarily)
- Partitioning: [] | 77 | [55, 44]
- Recursively sort: [55, 44]

Step 8: Recursively apply Quick Sort to the right subarray of the right subarray of the right subarray

- Right subarray: [55, 44]
- Pivot: 55 (chosen arbitrarily)
- Partitioning: [] | 55 | [44]
- Recursively sort: [44] (only one element, so return)

Step 9: Combine the results of the right subarray

- Sorted right subarray: [44, 55, 77, 93]

Step 10: Combine the results of the entire array

- Sorted array: [17, 20, 26, 31, 44, 54, 55, 77, 93]

Quick Sort is complete!

Algorithm:

Time Complexity:-

Let's analyze the time complexity of Quick Sort using the Master Theorem:

$T(n) = 2T(n/2) + O(n)$

Comparing with the Master Theorem form:

$T(n) = aT(n/b) + \theta(n^k \log^p n)$

We get:

$a = 2$
$b = 2$
$k = 1$
$p = 0$ (since the term is $O(n)$, not $O(n \log^p n)$)

Since a = b^k (2 = 2^1), we apply Case 2 of the Master Theorem.

Case 2: $a = b^k$
$p = 0 \geq -1$, so we use the formula:
$T(n) = \theta(n^{\log_b a} \log^{(p+1)} n)$
$= \theta(n^{\log_2 2} \log^1 n)$
$= \theta(n^1 \log n)$
$= \theta(n \log n)$

So, the best-case time complexity of Quick Sort is indeed $\theta(n \log n)$.

For the average case of Quick Sort, we still have:

$T(n) = 2T(n/2) + O(n)$

Using the Master Theorem:

$a = 2$
$b = 2$
$k = 1$
$p = 0$

Since $a = b^k$ ($2 = 2^1$), we apply Case 2 of the Master Theorem.

Case 2: $a = b^k$
$p = 0 \geq -1$, so we use the formula:
$T(n) = \theta(n^{\log_b a} \log^{(p+1)} n)$
$= \theta(n^{\log_2 2} \log^1 n)$
$= \theta(n^1 \log n)$
$= \theta(n \log n)$

However, in the average case, we need to consider the logarithmic factor hidden in the Big O notation. So, we write it as:

$T(n) = \theta(n \log n \log_2 n)$

Or simply:

$T(n) = \theta(n \log^2 n)$

So, the average-case time complexity of Quick Sort is indeed $\theta(n \log^2 n)$.
For the worst-case scenario of Quick Sort, we have:

$T(n) = 2T(n-1) + O(n)$

Using the Master Theorem:

$a = 2$
$b = 1$ (since the size of the subproblem is $n-1$, not $n/2$)
$k = 1$

p = 0

Since a > b^k (2 > 1^1), we apply Case 1 of the Master Theorem.

Case 1: a > b^k
T(n) = θ(n^log_b a)
= θ(n^log_1 2)
= θ(n^∞)

However, this is not a valid expression. The Master Theorem breaks down in this case because the recursion is too unbalanced.

Instead, we can solve the recurrence relation directly:

T(n) = 2T(n-1) + O(n)
= 2(2T(n-2) + O(n-1)) + O(n)
= 4T(n-2) + O(2n-1)
...
= 2^n T(0) + O(n2^n - 1)

Since T(0) is a constant, we get:

T(n) = O(2^n n)

However, this is still not the tightest bound. We can observe that the recurrence relation has a quadratic-like behavior, leading to:

T(n) = O(n^2)

So, the worst-case time complexity of Quick Sort is indeed O(n^2).

Note that this worst-case scenario occurs when the pivot is always the smallest or largest element, leading to highly unbalanced partitioning.

Strassen's Matrix Multiplication

Consider the product of two matrices A and B

$$\begin{bmatrix} A11 & A12 \\ A21 & A22 \end{bmatrix} \begin{bmatrix} B11 & B12 \\ B21 & B22 \end{bmatrix} = \begin{bmatrix} C11 & C12 \\ C21 & C22 \end{bmatrix}$$

$$A \qquad B \qquad = \qquad C$$

$$C11 = A11\,B11 + A12\,B21$$

$$C12 = A11\,B12 + A12\,B22$$

$$C21 = A21\,B11 + A22\,B21$$

$$C22 = A21\,B12 + A22\,B22$$

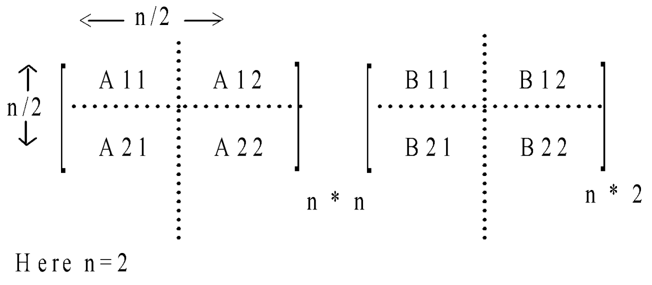
The time complexity of the matrix multiplication is equal to O (n^3).

TO multiply the two matrices of order 2 * 2, we require 8 multiplications and 4 additions.

$$\text{So,} \quad T(n) = \begin{cases} b & n \leq 2 \\ 8T(n/2) + cn^2 & n > 2 \end{cases}$$

Strassen's tried to reduce the time complexity from O(n). Using strassen's formula, we can perform the multiplication of two 2 * 2 matrices using only 7 multiplications and 18 additions or subtractions.

Using strassens matrix multiplication, first divide the given matrices into four partitions each of n/2 * n/2, where the given matrices are n*n. Note that n must be a power & 2 i.e. $n=2^k$. In case, when n is not a power of 2, then enough rows and columns of zeros can be added to both matrices A and B. so that the resulting dimensions are a power of 2.



Now find the solutions for each of the n/2 * n/2 matrices using strassens formula as given below

$$\begin{aligned}
P1 &= (A11+ A22)\ (B11+B22)\\
P2 &= (A21+ A22)\, B11\\
P3 &= A11\ (B12- B22)\\
P4 &= A22\ (B21- B11)\\
P5 &= (A11+ A12)\, B22\\
P6 &= (A21+ A11)\ (B11+B12)\\
P7 &= (A12+ A22)\ (B21+B22)
\end{aligned}$$

Now combine the results, we get

$$\begin{bmatrix} C11 & C12 \\ C21 & C22 \end{bmatrix} = \begin{bmatrix} P1 + P4 - P5 + P7 & P3 + P5 \\ P2 + P4 & P1 + P3 - P2 + P6 \end{bmatrix}$$

Example: Multiply the following matrices

$$\begin{bmatrix} 1 & 6 \\ 2 & 3 \end{bmatrix} \quad \begin{bmatrix} 5 & 2 \\ 1 & 4 \end{bmatrix}$$

Sol:

$$\begin{array}{cc} A11 & A12 \end{array} \qquad \begin{array}{c} B11 \end{array}$$

$$\left[\begin{array}{c:c} 1 & 6 \\ \hdashline 2 & 3 \end{array}\right] \quad \left[\begin{array}{c:c} 5 & 2 \\ \hdashline 1 & 4 \end{array}\right]$$

$$\begin{array}{cc} A21 & A22 \end{array} \qquad \begin{array}{cc} B21 & B22 \end{array}$$

$$\begin{aligned}
P1 &= (A11+ A22)\ (B11+B22) &&= (1+3)\ (5+4) &&= 36\\
P2 &= (A21+ A22)\, B11 &&= (2+3)\ 5 &&= 25\\
P3 &= A11\ (B12- B22) &&= 1\ (2-4) &&= -2\\
P4 &= A22\ (B21- B11) &&= 3\ (1-4) &&= -12\\
P5 &= (A11+ A12)\, B22 &&= (1+7)\ 4 &&= 28\\
P6 &= (A21+ A11)\ (B11+B12) &&= (2-1)\ (5+2) &&= 7\\
P7 &= (A12+ A22)\ (B21+B22) &&= (6-3)\ (1+4) &&= 15
\end{aligned}$$

$$\therefore \begin{bmatrix} C11 & C12 \\ C21 & C22 \end{bmatrix} = \begin{bmatrix} P1+P4-P5+P7 & P3+P5 \\ P2+P4 & P1+P3-P2+P6 \end{bmatrix}$$

$$= \begin{bmatrix} 36+(-12)-28+15 & -2+28 \\ 25-12 & 36-2-25+7 \end{bmatrix}$$

$$= \begin{bmatrix} 11 & 26 \\ 13 & 16 \end{bmatrix}$$

Using normal Method:

$$\begin{bmatrix} 1 & 6 \\ 2 & 3 \end{bmatrix} \begin{bmatrix} 5 & 2 \\ 1 & 4 \end{bmatrix} = \begin{bmatrix} 1*5+6*1 & 1*2+6*4 \\ 2*5+3*1 & 2*2+3*4 \end{bmatrix} = \begin{bmatrix} 11 & 26 \\ 13 & 16 \end{bmatrix}$$

Time Complexity of Strassen's Method:

Relation for T(n) is

$$T(n) = \begin{cases} b & n \le 2 \\ 7T(n/2) + cn^2 & n > 2 \end{cases}$$

Where a and c are constants.