Unit - II
Divide and Conquer

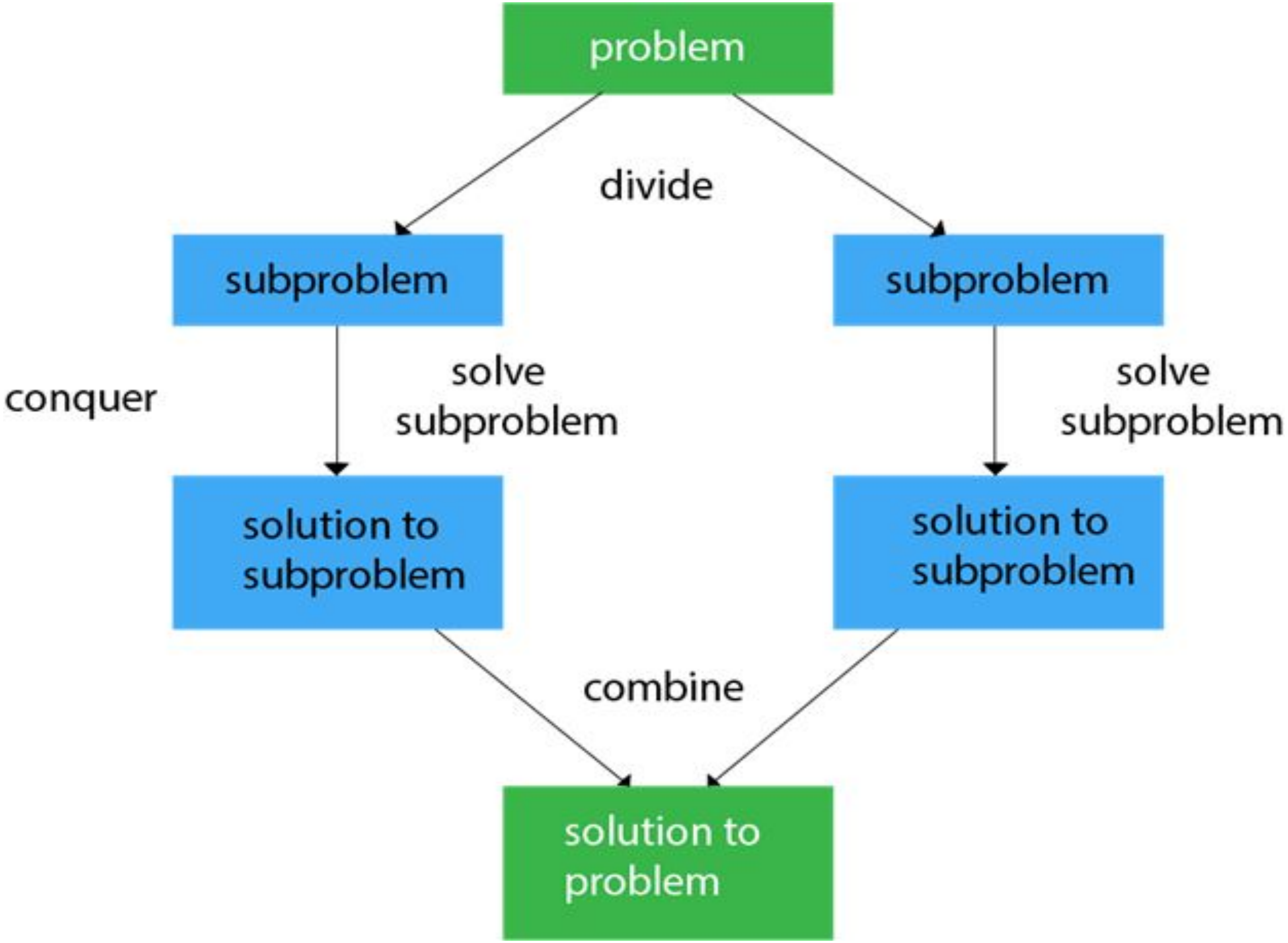
Contents

Divide and Conquer Introduction

- Divide and Conquer is an **algorithmic pattern.**
- In algorithmic methods, the design is to take a dispute on a huge input, **break the input into minor pieces, decide the problem on each of the small pieces, and then merge the piecewise solutions into a global solution.**

Divide and Conquer algorithm consists of following three steps.

1. **Divide the original problem** into a set of subproblems.
2. **Conquer:** Solve every subproblem individually, recursively.
3. **Combine:** Put together the solutions of the subproblems to get the solution to the whole problem.


problem
divide
subproblem
subproblem
conquer
solve subproblem
solve subproblem
solution to subproblem
solution to subproblem
combine
solution to problem

Examples: The specific computer algorithms are based on the Divide & Conquer approach:

- Binary Search
- Merge Sort
- Quick Sort
- Strassen’s Matrix Multiplication

Fundamental of Divide & Conquer Strategy:

There are two fundamental of Divide & Conquer Strategy:

1. **Relational Formula:** It is the formula that we generate from the given technique. After generation of Formula we apply D&C Strategy, i.e. **we break the problem recursively & solve the broken subproblems.**

2. **Stopping Condition:** When we **break** the problem using Divide & Conquer Strategy, then we need to know that for how much time, we need to apply divide & Conquer. So the **condition** where the need to stop our recursion steps of D&C is called as Stopping Condition.

Applications of Divide and Conquer Approach:

1. **Binary Search:** The binary search algorithm is a searching algorithm, which is also called **a half-interval search or logarithmic search.** It works by comparing the target value with the **middle element** existing in a sorted array.
2. **Quicksort:** It is the most efficient sorting algorithm, which is also known as partition-exchange sort. It starts by selecting **a pivot** value from an array followed by **dividing the rest of the array elements into two sub-arrays.**
3. **Merge Sort:** It is a sorting algorithm that sorts an array by making comparisons. It starts by **dividing an array into sub-array and then recursively sorts each of them**. After the sorting is done, it merges them back.
4. **Strassen's Algorithm:** It is an algorithm for matrix multiplication, which is named after Volker Strassen. It has proven to be **much faster** than the traditional algorithm when **works on large matrices.**

Advantages of Divide and Conquer

1. This algorithm is **much faster** than other algorithms.
2. It efficiently uses cache memory without occupying much space because it solves simple **subproblems within the cache memory** instead of accessing the slower main memory.
3. It is more proficient than that of its counterpart Brute Force(Try every possibility)technique.
4. Since these algorithms inhibit parallelism, it does not involve any modification and is handled by systems incorporating **parallel processing**.

Disadvantages of Divide and Conquer

1. Since most of its algorithms are designed by incorporating recursion, so it necessitates **high memory management.**
2. An explicit stack may **overuse the space**.
3. It may even crash the system if the recursion is performed **rigorously greater than the stack present in the CPU.**

1. Binary Search

1. In Binary Search technique, we search an element in a sorted array by **recursively dividing the interval in half.**

2. Firstly, we take the whole array as an interval.

3. If the Pivot Element (the item to be searched) is **less** than the item in the middle of the interval, We discard the second half of the list and recursively repeat the process for the **first half of the list by calculating the new middle and last element.**

4. If the Pivot Element (the item to be searched) is **greater** than the item in the middle of the interval, we discard the first half of the list and work recursively on the **second half by calculating the new beginning and middle element.**

5. Repeatedly, check until the value is found or interval is empty.

6. Binary search is a fast search algorithm with run-**time complexity of O(log n).**

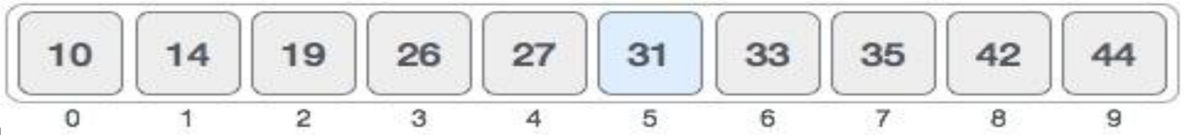
7. Space Complexity: The space complexity of merge sort is **O(1).**

How Binary Search Works?

- For a binary search to work, it is mandatory for the target array to be sorted.

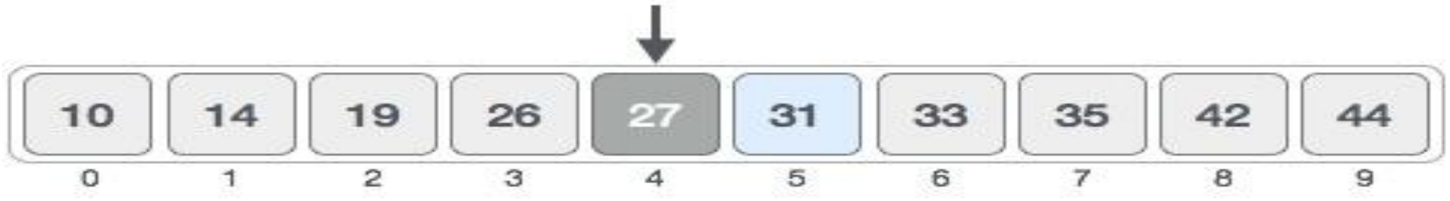
Step 1 :

- The following is our sorted array and let us assume that we need to search the location of value **31** using binary search.



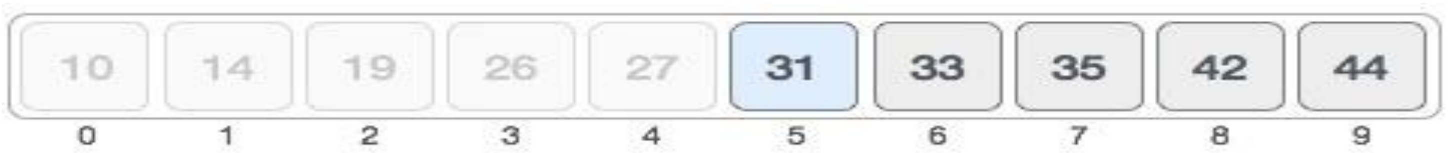
- **mid = (beg + end)/2**
- 0 + (9) / 2 = 4
- (integer value of 4.5). So, 4 is the mid of the array.

Step 2 :



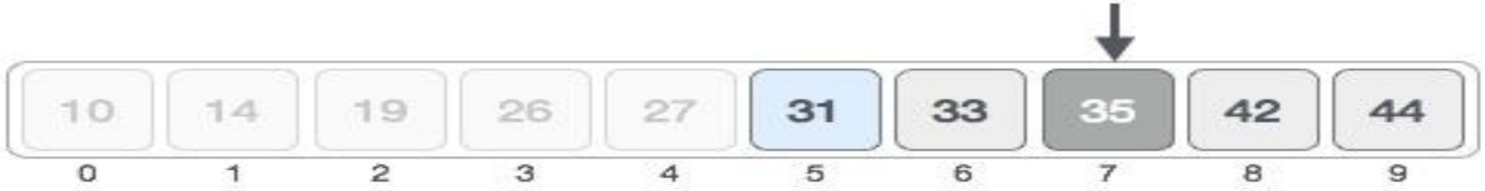
- Now we compare the value stored at location 4, with the value being searched, i.e. **31.**
- We find that the value at location 4 is **27**, which is not a match.
- As the value is greater than 27 and we have a sorted array, so we also know that the target value must be in the **upper(in Left Most)** portion of the array.

Step 3 :

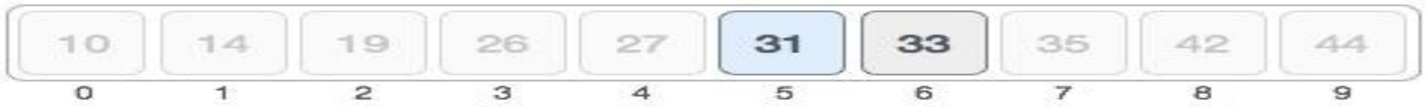


- We change our low to mid + 1 and find the new mid value again.
- beg = mid + 1
- mid = (beg + end)/2
- mid=(5+9)/2
- Our new mid is **7** now. We compare the value stored at location 7 with our target value 31.

Step 4 :

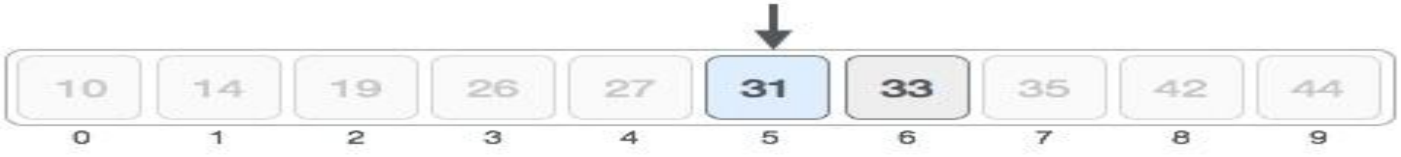


The value stored at location 7 is not a match, rather it is more than what we are looking for. So, the value must be in the lower part from this location.

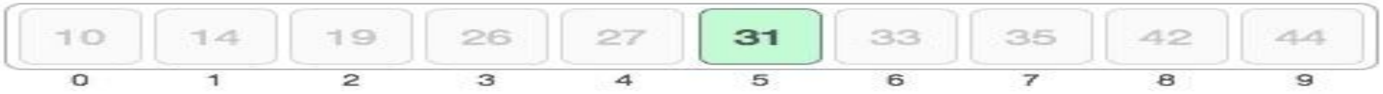


Hence, we calculate the mid again. This time it is **5**.

Step 5 :



We compare the value stored at location 5 with our target value. We find that it is a match.



- We conclude that the target value 31 is stored at location 5.
- Binary search halves the searchable items and thus reduces the count of comparisons to be made to very less numbers.

Algorithm:

```
**Binary-Search**(numbers[], x, l, r)
if l = r then
   return l
else
   m := ⌊(l + r) / 2⌋
   if x ≤ numbers[m]  then
      return Binary-Search(numbers[], x, l, m)
   else
      return Binary-Search(numbers[], x, m+1, r)
```

Analysis:

Input: an array A of size n, already sorted in the ascending or descending order.

Output: analyze to search an element item in the sorted array of size n.

Logic: Let T (n) = number of comparisons of an item with n elements in a sorted array.

Set BEG = 1 $\text{int}\left(\frac{beg + end}{2}\right)$ n

Find mid =

Compare the search item with the mid item.

Case 1: item = A[mid], then LOC = mid, but it the best case and **T (n) = 1**

Case 2: item ≠A [mid], then we will split the array into two equal parts of size Binary Search.

And again find the midpoint of the half-sorted array and compare with search element.

Repeat the same process until a search element is found.

so,Using this **recurrence relation T(n)=logn.**

Binary Search Recurrence by Master Method.

$$T(n) = T\left(\frac{n}{2}\right) + 1$$

$a = 1, \quad b = 2$

$\log_2^1 = 0 \; ; \; k = 0 \; , \; P = 0 > -1$

By Case 2,

$\Theta(1 \log n) \; \ldots \; \boxed{\log^{0+1} n}$

$\boxed{\Theta(\log n)}$

BINARY SEARCH

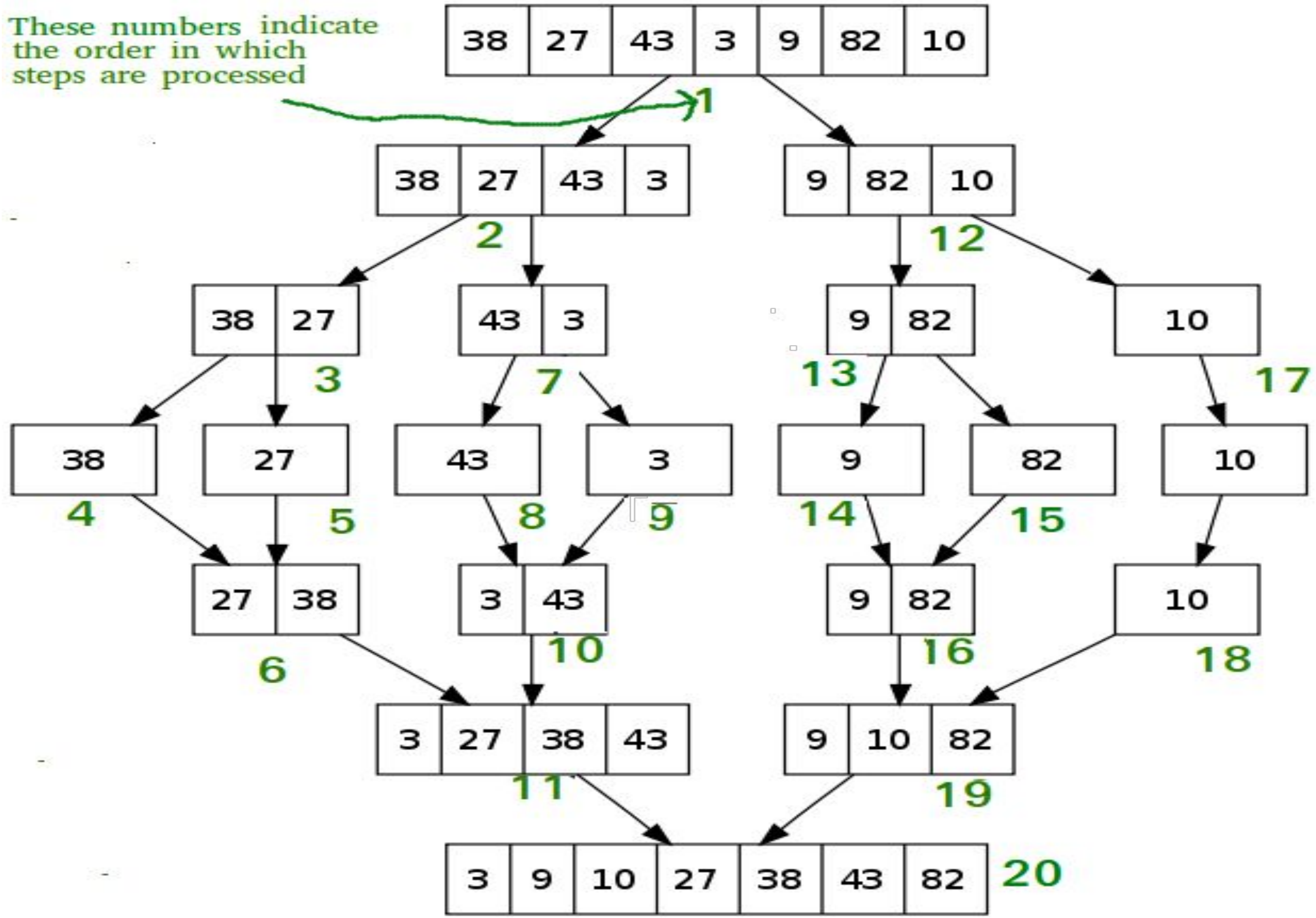
/*** Program to Search an Array using Binary Search *****/**

```
#include <stdio.h>
void binary_search();
int a[50], n, item, loc, beg, mid, end, i;
main()
{
printf("\nEnter size of an array: ");
scanf("%d", &n);//5
printf("\nEnter elements of an array in sorted form:\n");
for(i=0; i<n; i++)//5
scanf("%d", &a[i]);//3 6 78 90 112
printf("\nEnter ITEM to be searched: ");//90
scanf("%d", &item);
binary_search();
getch();
}
```

```
void binary_search()
{
beg = 0; end = n-1;
mid = (beg+end)/2;//0+4
while ((beg<=end) && (a[mid]!=item))//78!=90
{
if (item < a[mid])//90>78
end = mid-1;
else
beg = mid+1;//3
mid = (beg+end)/2; //3+
}
if (a[mid] == item)
printf("\n\nITEM found at location %d", mid+1);
else
printf("\n\nITEM doesn't exist");
}
```

2. Merge Sort

1. Merge Sort is a Divide and Conquer algorithm.
2. It divides the input array into two halves, calls itself for the two halves, and then merges the two sorted halves.
3. The merge() function is used for merging two halves.
4. The merge(arr, l, m, r) is a key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted subArrays into one.


These numbers indicate the order in which steps are processed
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
38 27 43 3 9 82 10
38 27 43 3
9 82 10
38 27
43 3
9 82
10
38
27
43
3
9
82
10
27 38
3 43
9 82
10
3 27 38 43
9 10 82
3 9 10 27 38 43 82

Algorithm

```
MergeSort(arr[], l,  r)
If r > l
     1. Find the middle point to divide the array into
two halves:
             middle m = l+ (r-l)/2
     2. Call mergeSort for first half:
             Call mergeSort(arr, l, m)
     3. Call mergeSort for second half:
             Call mergeSort(arr, m+1, r)
     4. Merge the two halves sorted in step 2 and 3:
             Call merge(arr, l, m, r)
```

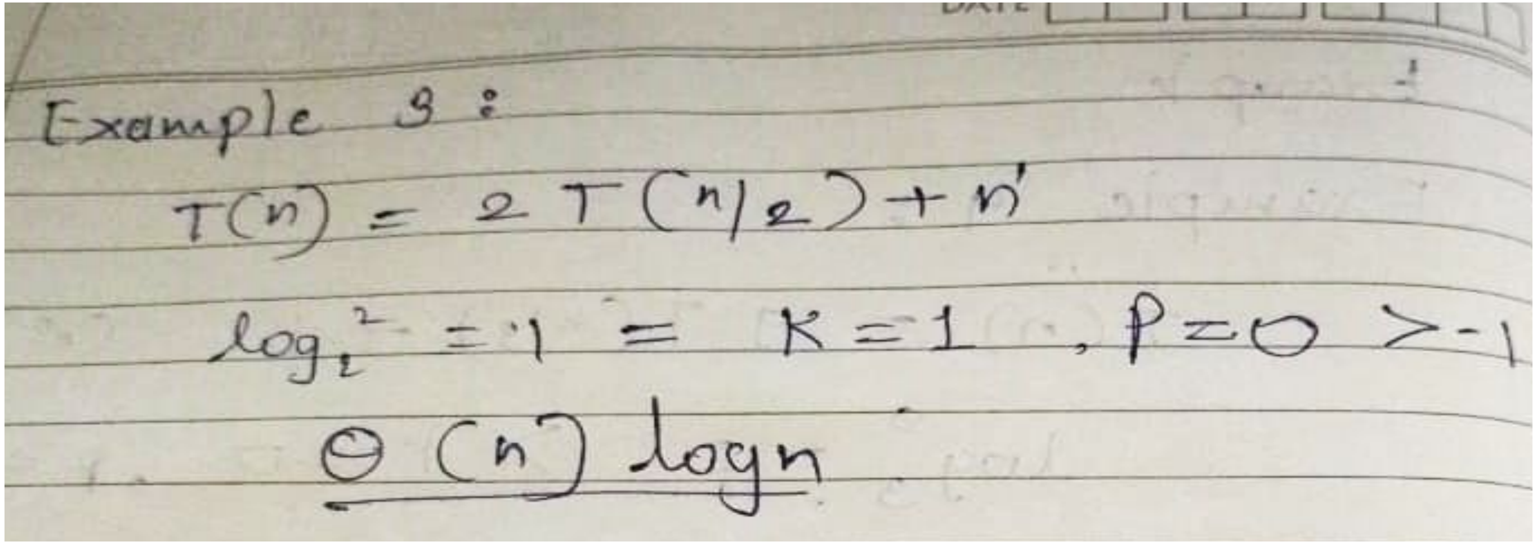
Time Complexity: Merge Sort

- Sorting arrays on different machines.
- Overall time complexity of Merge sort is **O(nLogn)**.
- Merge Sort is a recursive algorithm and time complexity can be expressed as following recurrence relation.

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

- **Stopping Condition T (1) =0** because at last, there will be only 1 element left that need to be copied, and there will be no comparison.
- **Space Complexity:** The space complexity of merge sort is **O(n).**

Recurrence Relation by Master Method



Example 3 :

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

$\log_2^2 = 1 = K = 1, P = 0 > -1$

$\underline{\Theta(n) \log n}$

Merge Sort Applications

The concept of merge sort is applicable in the following areas:

1. **Inversion count problem**
2. **External sorting**
3. **E-commerce applications**

3. Quick Sort

- It is an algorithm of Divide & Conquer type.
- **Divide:** Rearrange the elements and split arrays into two sub-arrays and an element in between search that each element in left subarray is less than or equal to the average element and each element in the right sub- array is larger than the middle element.
- **Conquer:** Recursively, sort two sub arrays.
- **Combine:** Combine the already sorted array.

3. Quick Sort..

It is also called partition-exchange sort. This algorithm divides the list into three main parts:

1. **Elements less than the Pivot element**
2. **Pivot element(Central element)**
3. **Elements greater than the pivot element**

Pivot element can be any element from the array, it can be the **first** element, the **last** element or any random element. In this tutorial, we will take the rightmost element or the last element as pivot.

3. Quick Sort..

For example: In the array **{52, 37, 63, 14, 17, 8, 6, 25}**, we take **25 as pivot**. So after the first pass, the list will be changed like this.

{6 8 17 14 25 63 37 52}

Hence **after the first pass**, pivot will be set at its position, with all the elements **smaller to it on its left** and all the elements **larger than to its right**. Now **6 8 17 14 and 63 37 52** are considered as two separate subarrays, and same recursive logic will be applied on them, and we will keep doing this until the complete array is sorted.

Quick Sort Pivot Partitioning Algorithm

Step 1 – Choose the highest index value has pivot

Step 2 – Take two variables to point left and right of the list excluding pivot

Step 3 – left points to the low index

Step 4 – right points to the high

Step 5 – while value at left is less than pivot move right

Step 6 – while value at right is greater than pivot move left

Step 7 – if both step 5 and step 6 does not match swap left and right

Step 8 – if left ≥ right, the point where they met is new pivot

Quick Sort Algorithm

Using pivot algorithm recursively, we end up with smaller possible partitions.

Each partition is then processed for quick sort. We define recursive algorithm for quicksort as follows −

Step 1 − Make the right-most index value pivot

Step 2 − partition the array using pivot value

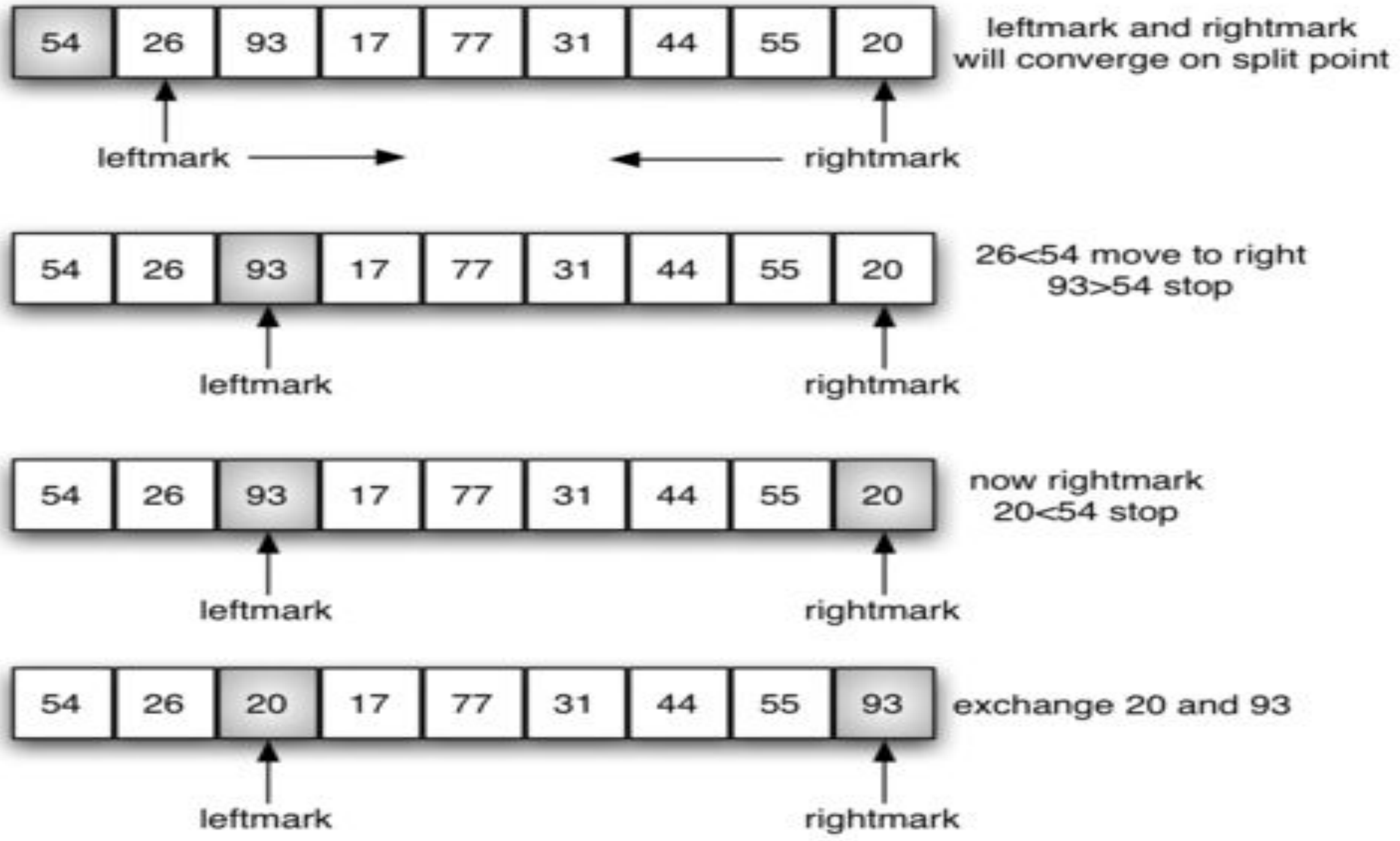
Step 3 − quicksort left partition recursively

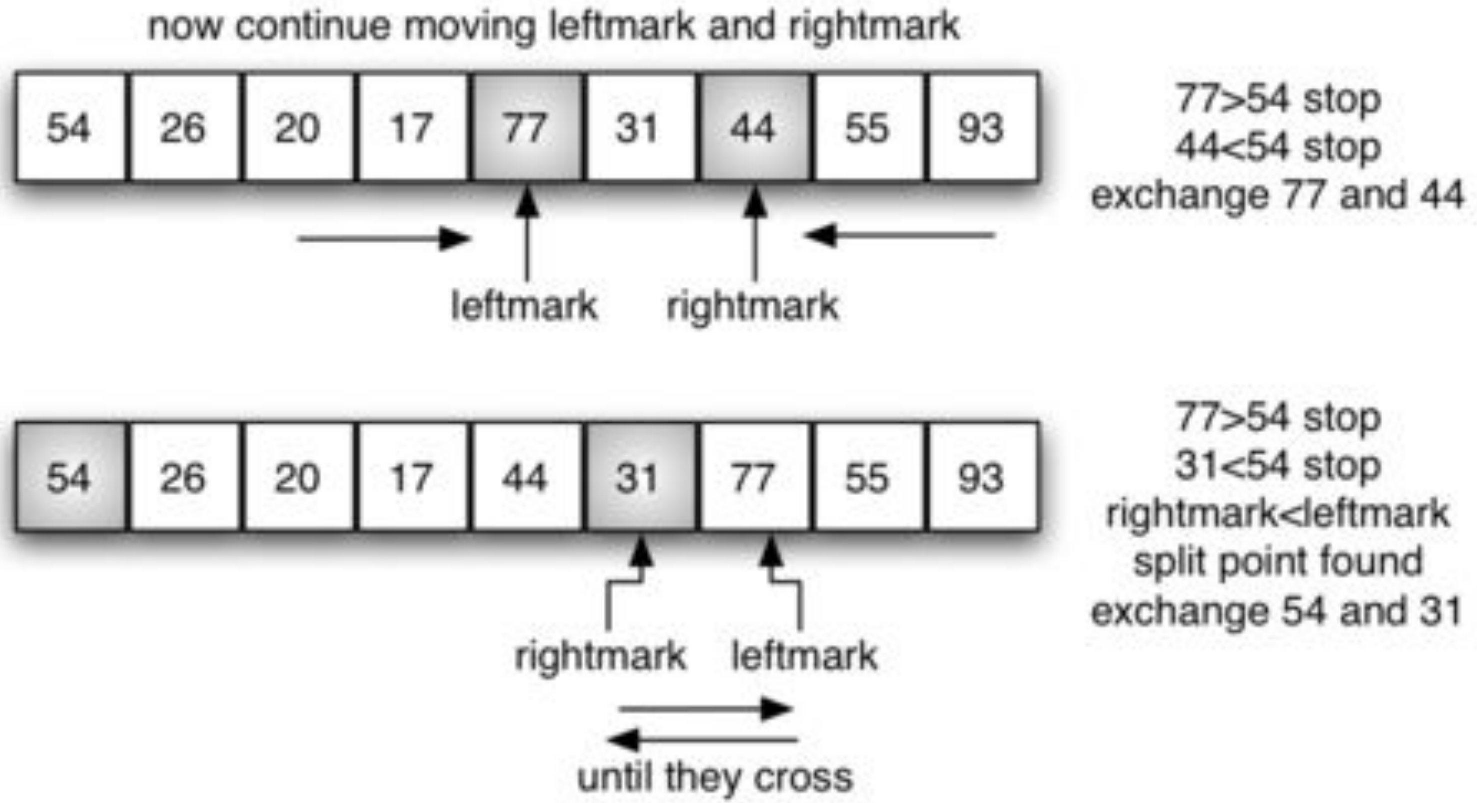
Step 4 − quicksort right partition recursively

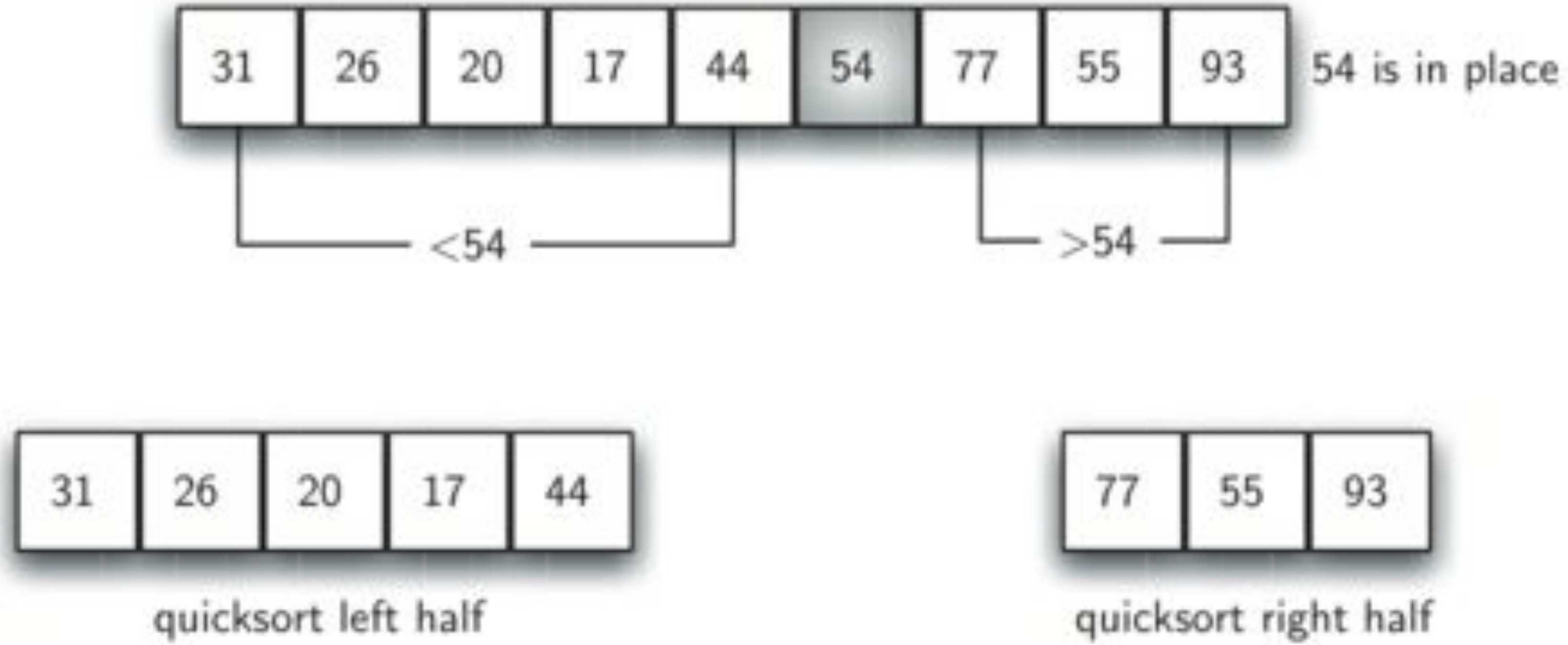
How Quick Sorting Works?

Following are the steps involved in quick sort algorithm:

1. After **selecting an element as pivot,** which is the last/first/random index of the array, we divide the array for the first time.
2. In quick sort, **we call this partitioning**. It is not simple breaking down of array into 2 subarrays, but in case of partitioning, the array elements are so positioned that all the **elements smaller than the pivot will be on the left side of the pivot** and all the elements **greater than the pivot will be on the right side of it.**
3. And the pivot element will be at its **final sorted position.**
4. The elements to the left and right, may not be sorted.
5. Then we pick subarrays, **elements on the left of pivot** and **elements on the right of pivot,** and we perform partitioning on them by **choosing a pivot in the subarrays.**


54
26
93
17
77
31
44
55
20
leftmark and rightmark will converge on split point
leftmark
rightmark
26<54 move to right
93>54 stop
now rightmark
20<54 stop
exchange 20 and 93


now continue moving leftmark and rightmark
54
26
20
17
77
31
44
55
93
leftmark
rightmark
77>54 stop
44<54 stop
exchange 77 and 44
54
26
20
17
44
31
77
55
93
rightmark
leftmark
until they cross
77>54 stop
31<54 stop
rightmark<leftmark
split point found
exchange 54 and 31

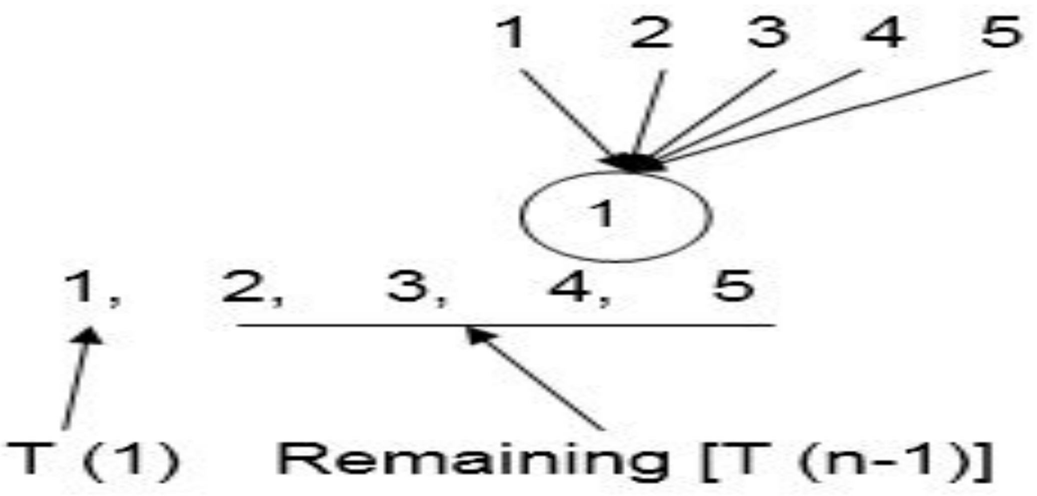

31
26
20
17
44
54
77
55
93
54 is in place
<54
>54
31
26
20
17
44
77
55
93
quicksort left half
quicksort right half

Quicksort Applications

Quicksort is implemented when

1. The programming language is good for recursion
2. Time complexity matters
3. Space complexity matters

- **Time Complexity** is
  - **Best case O(n^2)**
  - **Worst Case O(n*logn)**
- **Space Complexity** is O(nlogn)
- **Recurrence Relation**
  - **Best case 2T(n/2)+n**
  - **Worst Case T (n) =T(1)+T(n-1)+n** Equivalent to **T(n-1)+n**

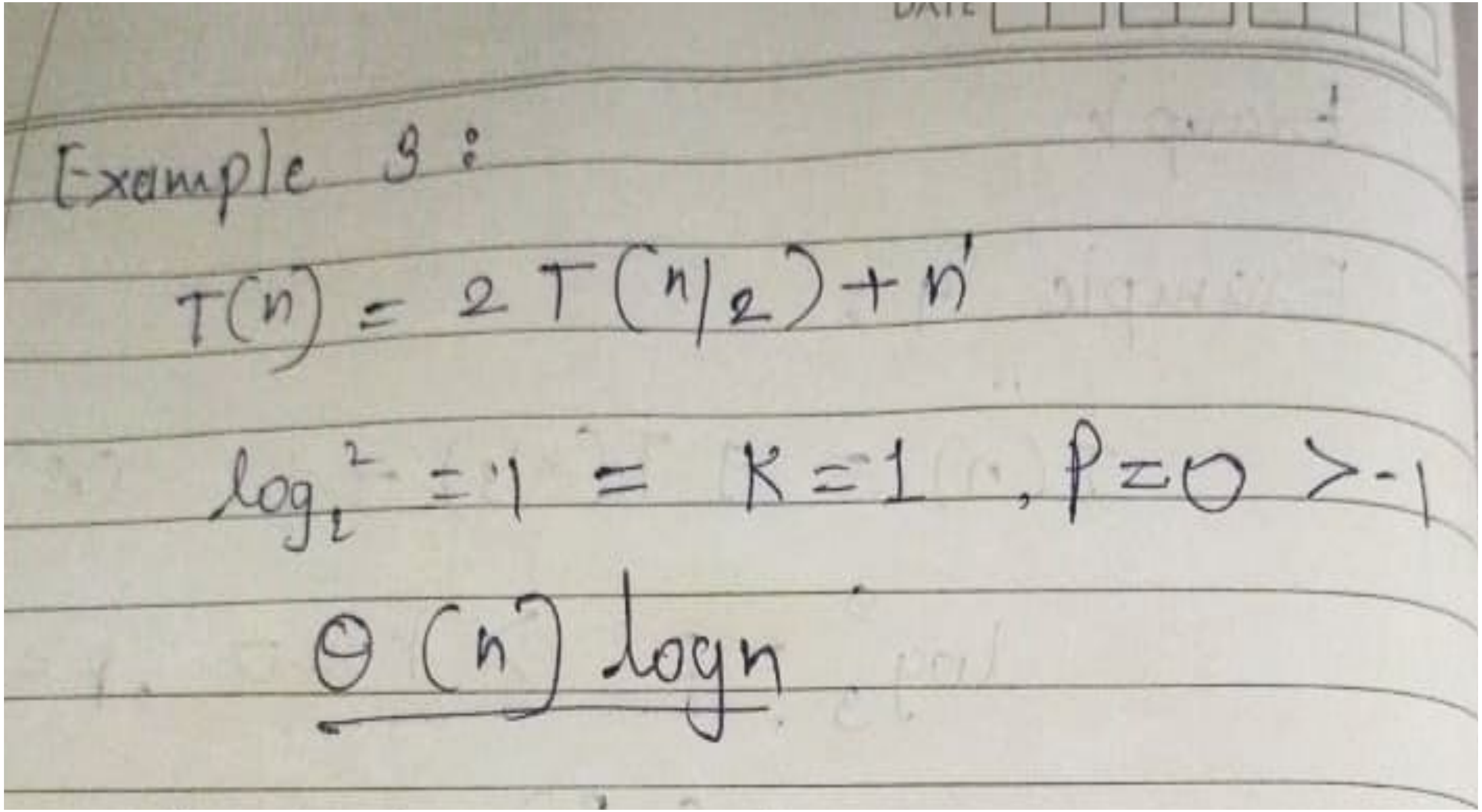


T (1) is time taken by

T (n-1) is time taken

N: the number of com ery element)

Best case 2T(n/2)+n



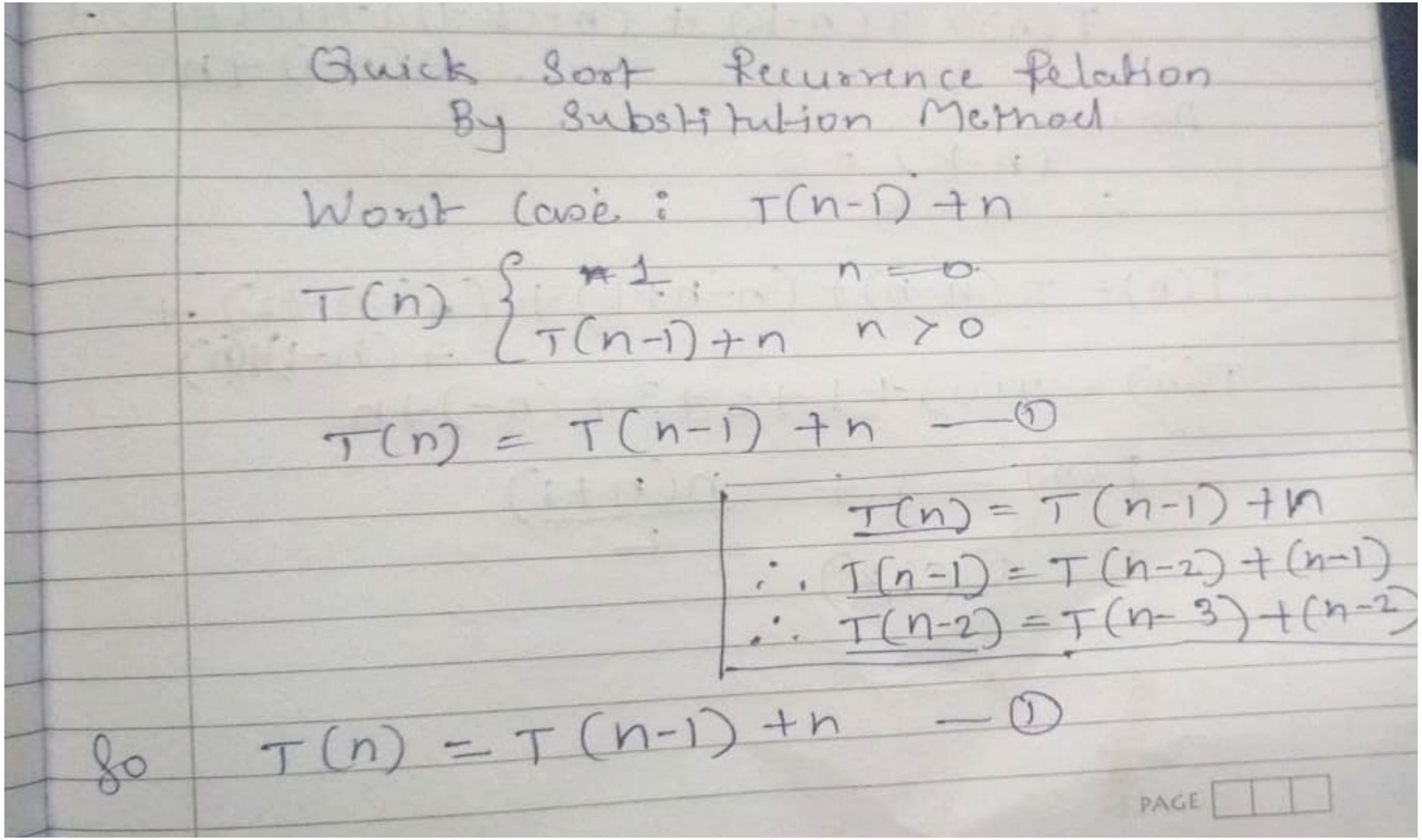
Example 3 :

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

$\log_2^2 = 1 = K = 1 , P = 0 > -1$

$\Theta(n) \log n$

Worst Case T (n) =T(1)+T(n-1)+n



Quick Sort Recurrence Relation
By Substitution Method

Worst Case : $T(n-1) + n$

$$T(n) \begin{cases} 1 & n = 0 \\ T(n-1) + n & n > 0 \end{cases}$$

$T(n) = T(n-1) + n$ — $\textcircled{1}$

$\underline{T(n)} = T(n-1) + n$

$\therefore \underline{T(n-1)} = T(n-2) + (n-1)$

$\therefore \underline{T(n-2)} = T(n-3) + (n-2)$

So $T(n) = T(n-1) + n$ — $\textcircled{1}$

PAGE

After substitution

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

$T(n) = \underline{T(n-2)} + \underline{(n-1) + n}$ — ②

After substitution

$T(n) = [T(n-3) + (n-2)] + \underline{(n-1) + n}$

$T(n) = T(n-3) + (n-2) + (n-1) + n$ — ③

So will get . . .

So will get ...

$$T(n) = T(n-k) + (n-(k-1)) + (n-(k-2)) + \ldots + (n-1) + n$$

Assume

$$(n-k) = 0$$

$$\therefore \quad n = k$$

$$T(n) = T(n-n) + (\cancel{n}-\cancel{n}+1) + (\cancel{n}-\cancel{n}+2) + \ldots + (n-1) + (n)$$

$$T(n) = T(0) + 1 + 2 + 3 + (n-) + n$$

$$T(n) = 1 + \frac{n(n+1)}{2}$$

$$\therefore \quad O(n^2)$$

4. Without Using Strassen's Matrix Multiplication

- Given two square matrices **A and B of size n x n** each, find their **Multiplication Matrix**.
- **Naive Method**

```
void multiply(int A[][N], int B[][N], int C[][N])
{
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; j++)
        {
            C[i][j] = 0;
            for (int k = 0; k < N; k++)
            {
                C[i][j] += A[i][k]*B[k][j];
            }
    }}}
```

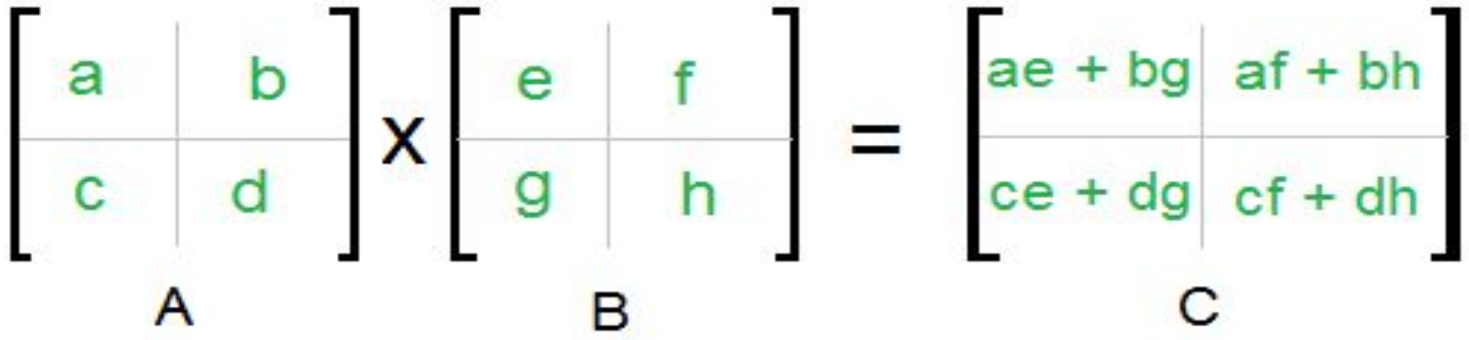
Time Complexity of above method is O(N^3).

What Does Divide and Conquer?

Following is simple **Divide and Conquer method** to multiply two square matrices.

1) Divide matrices **A and B in 4 sub-matrices** of size **N/2 x N/2** as shown in the below diagram.

2) Calculate following values recursively. **ae + bg, af + bh, ce + dg and cf + dh.**



A, B and C are square metrices of size N x N
a, b, c and d are submatrices of A, of size N/2 x N/2
e, f, g and h are submatrices of B, of size N/2 x N/2

- In the above method, we do 8 multiplications for matrices of size N/2 x N/2 and 4 additions. Addition of two matrices takes O(N^2) time. So the time complexity can be written as

  T(N) = 8T(N/2) + O(N^2)

  From Master's Theorem, time complexity of above method is **O(N^3)**which is unfortunately **same as Naive method.**

By Using Master Theorem
Case 1 :

$T(n) = 8T\left(\frac{n}{2}\right) + O(n^2)$

$a = 8, \quad b = 2$

$\log_2^8 = \underline{3} > K = 2$

By Master Theorem (2)

If $\log_b^a > K$ then $\Theta\left(n^{\log_b^a}\right)$

So $T(n) = n^{\log_b^a}$
$= n^{\log_2^8}$

$$\boxed{T(n) = n^3}$$

classmate

Simple Divide and Conquer also leads to O(N^3), can there be a better way?

- In **divide and conquer method,** the main component for **high time complexity is 8 recursive calls.**
- The **idea of Strassen’s method** is to **reduce the number of recursive calls to 7.**
- Strassen’s method is similar to above simple divide and conquer method in the sense that this method also divide matrices to sub-matrices of size N/2 x N/2 but in **Strassen’s method, the four sub-matrices of result are calculated using following formulae.**

$p1 = a(f - h)$ $\qquad$ $p2 = (a + b)h$

$p3 = (c + d)e$ $\qquad$ $p4 = d(g - e)$

$p5 = (a + d)(e + h)$ $\qquad$ $p6 = (b - d)(g + h)$

$p7 = (a - c)(e + f)$

The A x B can be calculated using above seven multiplications.
Following are values of four sub-matrices of result C

$$\underset{A}{\begin{bmatrix} a & b \\ c & d \end{bmatrix}} \times \underset{B}{\begin{bmatrix} e & f \\ g & h \end{bmatrix}} = \underset{C}{\begin{bmatrix} p5 + p4 - p2 + p6 & p1 + p2 \\ p3 + p4 & p1 + p5 - p3 - p7 \end{bmatrix}}$$

A, B and C are square metrices of size N x N
a, b, c and d are submatrices of A, of size N/2 x N/2
e, f, g and h are submatrices of B, of size N/2 x N/2
p1, p2, p3, p4, p5, p6 and p7 are submatrices of size N/2 x N/2

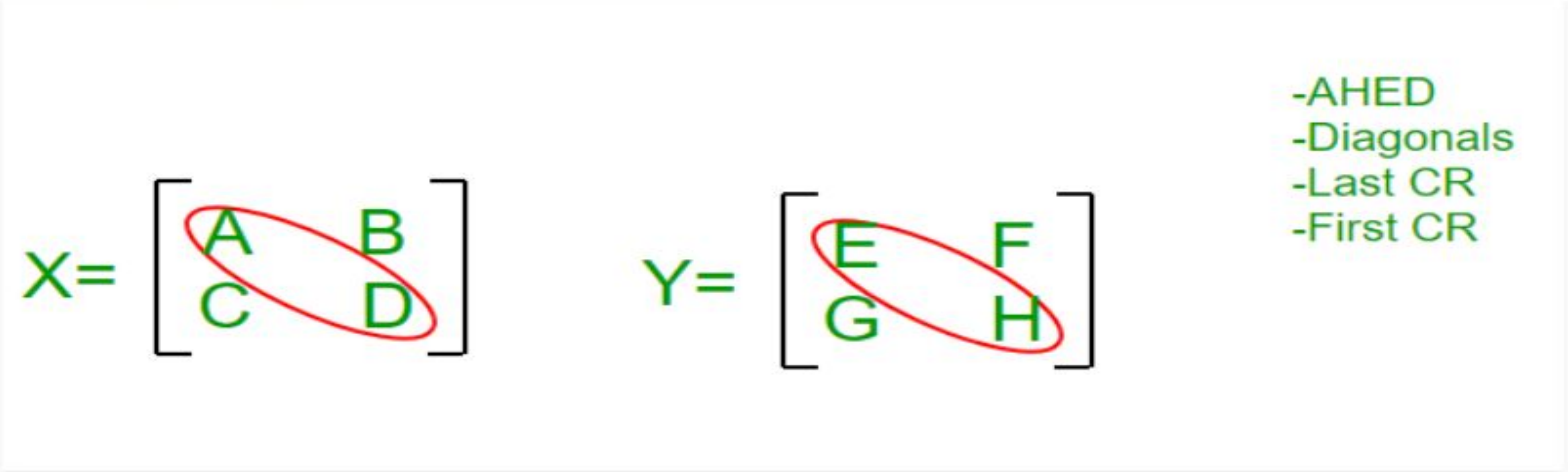
You just need to remember 4 Rules :

- AHED (Learn it as ‘Ahead’)
- Diagonal
- Last CR
- First CR

Also, consider **X as (Row +)** and **Y as (Column -)** matrix

Follow the Steps :

- Write P1 = A; P2 = H; P3 = E; P4 = D
- For P5 we will use Diagonal Rule i.e.
  (Sum the Diagonal Elements Of Matrix X) * (Sum the Diagonal Elements Of Matrix Y), we get
  P5 = (A + D)* (E + H)



P1 = A

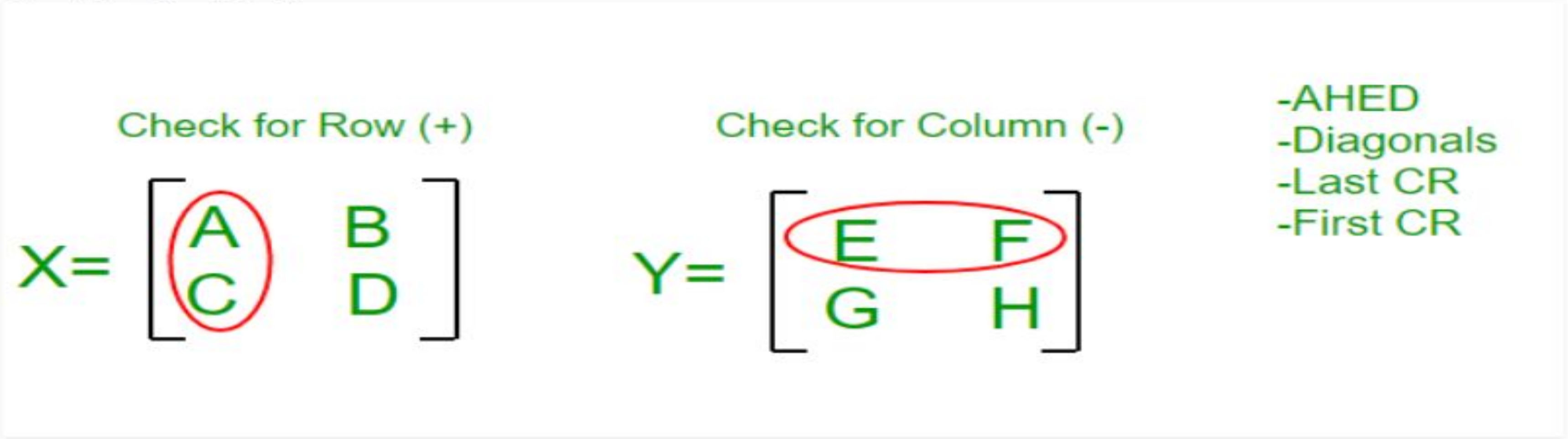
P2= H

P3= E

P4= D

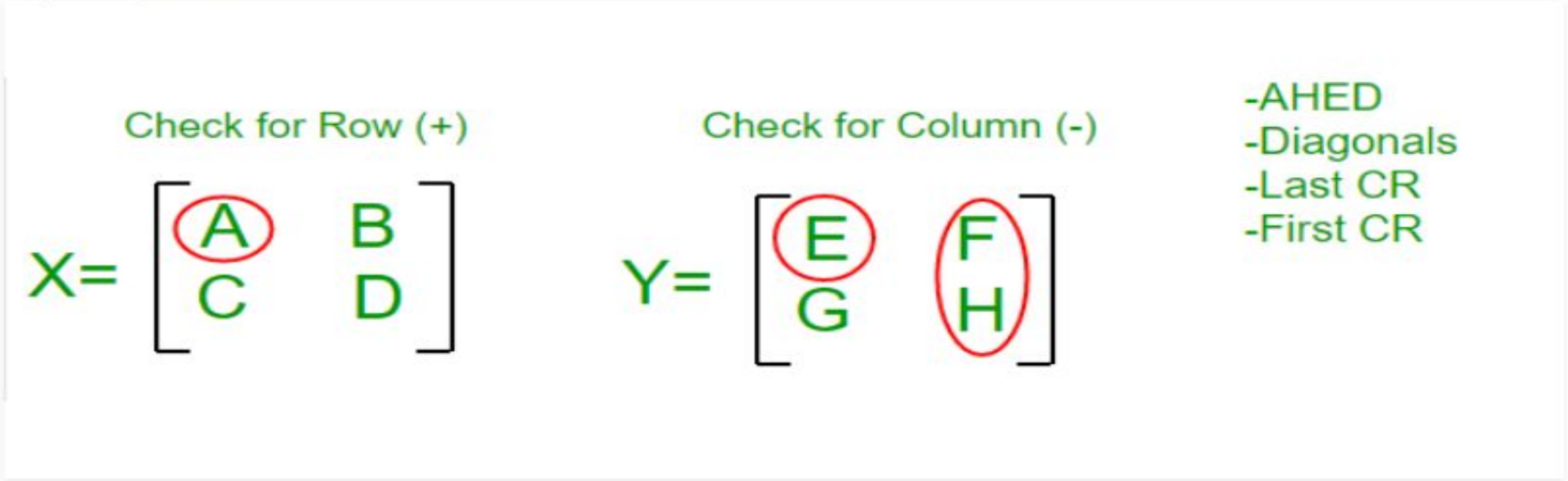
P5= (A + D) * (E + H)

- For P6 we will use Last CR Rule i.e. Last Column of X and Last Row of Y and remember that Row+ and Column- so i.e. (B − D) * (G + H), we get
  P6 = (B − D) * (G + H)
- For P7 we will use First CR Rule i.e. First Column of X and First Row of Y and remember that Row+ and Column- so i.e. (A − C) * (E + F), we get
  P7 = (A − C) * (E + F)



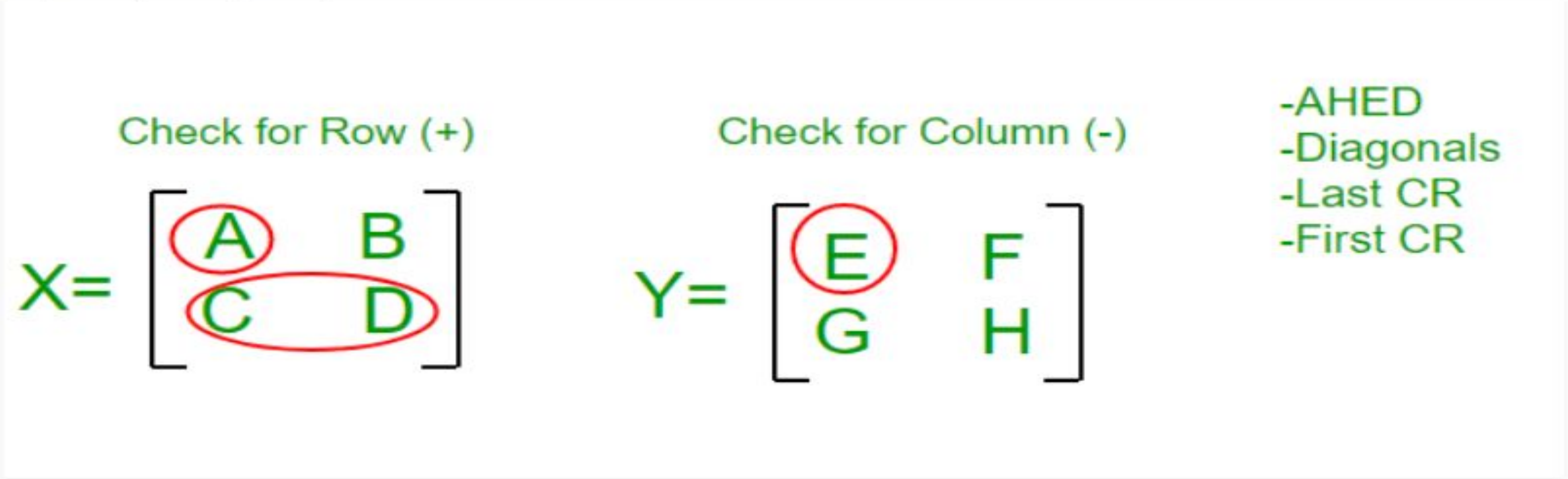
P1 = A
P2= H
P3= E
P4= D
P5= (A + D) * (E + H)
P6= (B − D) * (G + H)
P7= (A − C) * (E + F)

- Come Back to P1 : we have A there and it's adjacent element in Y Matrix is E, since Y is Column Matrix so we select a column in Y such that E won't come, we find F H Column, so multiply A with (F – H)
  So, finally P1 = A * (F – H)



P1 = A * (F – H)
P2= H
P3= E
P4= D
P5= (A + D) * (E + H)
P6= (B – D) * (G + H)
P7= (A – C) * (E + F)

- Come Back to P2 : we have H there and it's adjacent element in X Matrix is D, since X is Row Matrix so we select a Row in X such that D won't come, we find A B Column, so multiply H with (A + B)
  So, finally P2 = (A + B) * H
- Come Back to P3 : we have E there and it's adjacent element in X Matrix is A, since X is Row Matrix so we select a Row in X such that A won't come, we find C D Column, so multiply E with (C + D)
  So, finally P3 = (C + D) * E



P1= A * (F - H)
P2= H * (A + B)
P3= E * (C + D)
P4= D

- Come Back to P4 : we have D there and it's adjacent element in Y Matrix is H, since Y is Column Matrix so we select a column in Y such that H won't come, we find G E Column, so multiply D with (G – E)
  So, finally P4 = D * (G – E)
  We are done with P1 – P7 equations, so now we move to C1 – C4 equations in Final Matrix C :
- Remember Counting : Write P1 + P2 at C2
- Write P3 + P4 at its diagonal Position i.e. at C3
- Write P4 + P5 + P6 at 1st position and subtract P2 i.e. C1 = P4 + P5 + P6 – P2
- Write odd values at last Position with alternating – and + sign i.e. P1 P3 P5 P7 becomes C4 = P1 – P3 + P5 – P7

$$XY = \begin{bmatrix} P6 + P5 + P4 - P2 & P1 + P2 \\ P3 + P4 & P1 + P5 - P3 - P7 \end{bmatrix}$$

Time Complexity of Strassen’s Method

Addition and Subtraction of two matrices takes O(N^2) time. So time complexity can be written as

T(N) = 7T(N/2) + O(N^2)

From Master's Theorem, time complexity of above method is

O(N^Log7) which is approximately O(N^2.8074)i.e. **O(N^2.81)**

$T(n) = 7 + (\frac{n}{2}) + O(n^2)$

By Master Theorem,

$a = 7$, $b = 2$

$\log_2^7 = 2.81 > k = 2$

Apply case ①

if $\log_b^a > k$ then $\Theta(n^{\log_b^a})$

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

$\boxed{T(n) = (n)^{2.81}} < n^3$

THANK YOU