**Experiment no-2 Date:10/2/19**

**AIM:** Experiment based on divide and Conquer approach

**Merge sort**

Algorithm:-

**MergeSort(arr[], l, r)**

If r > l

**1.** Find the middle point to divide the array into two halves:

middle m = (l+r)/2

**2.** Call mergeSort for first half:

Call mergeSort(arr, l, m)

**3.** Call mergeSort for second half:

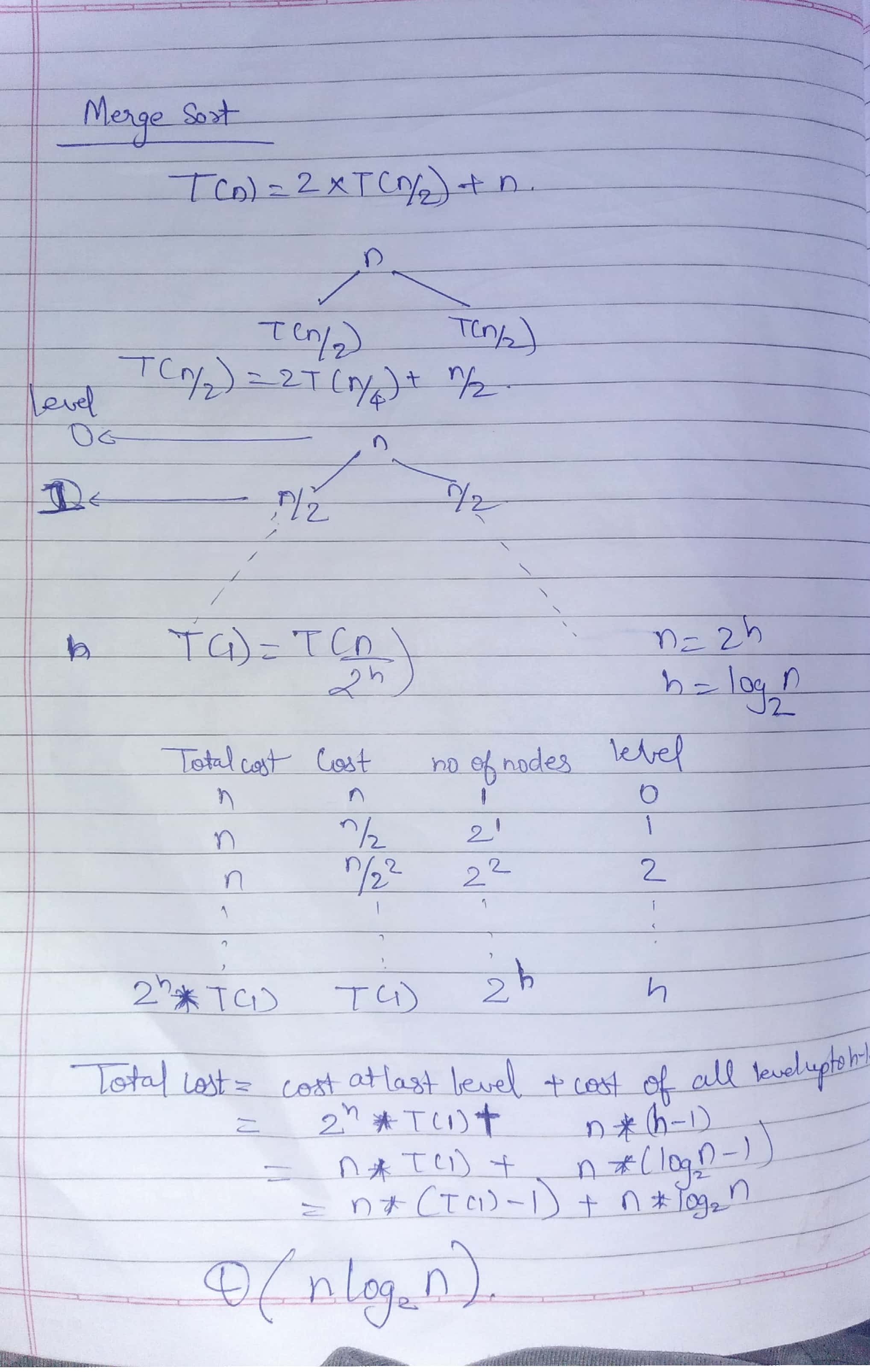
Call mergeSort(arr, m+1, r)

**4.** Merge the two halves sorted in step 2 and 3:

Call merge(arr, l, m, r)



**Time Complexity:** Sorting arrays on different machines. Merge Sort is a recursive algorithm and time complexity can be expressed as following recurrence relation.  
T(n) = 2T(n/2) + n  
The above recurrence can be solved either using Recurrence Tree method or Master method. It falls in case II of Master Method and solution of the recurrence is **Theta(nlogn)** .  
Time complexity of Merge Sort is **Theta(nlogn)**  in all 3 cases (worst, average and best) as merge sort always divides the array in two halves and take linear time to merge two halves.



Program:-

#include<stdio.h>

void merge(int arr[],int l,int m,int r,int \*comp)

{

int i, j, k;

int n1 = m - l + 1;

int n2 = r - m;

int L[n1], R[n2];

for (i = 0; i < n1; i++)

L[i] = arr[l + i];

for (j = 0; j < n2; j++)

R[j] = arr[m + 1+ j];

i = 0;

j = 0;

k = l;

while (i < n1 && j < n2)

{

if (L[i] <= R[j])

{

arr[k++] = L[i++];

(\*comp)++;

}

else

{

arr[k++] = R[j++];

(\*comp)++;

}

(\*comp)++;

}

while (i < n1)

{

arr[k++] = L[i++];(\*comp)++;;

}

while (j < n2)

{

arr[k++] = R[j++];(\*comp)++;;

}

}

void mergeSort(int a1[],int l,int r,int \*comp)

{

if(l<r)

{

int m=(l+r)/2;

mergeSort(a1,l,m,comp);

mergeSort(a1,m+1,r,comp);

merge(a1,l,m,r,comp);

}

}

int main()

{

int a1[6]={55,75,25,45,15,65};

int comp=0,i=0;

printf("Unsorted array is\n");

for(i=0;i<6;i++)

printf("%d\t",a1[i]);

mergeSort(a1,0,5,&comp);

printf("\n%d Comparisons made \nSorted array is\n",comp);

for(i=0;i<6;i++)

printf("%d\t",a1[i]);

return 0;

}

**OUTPUT:-**

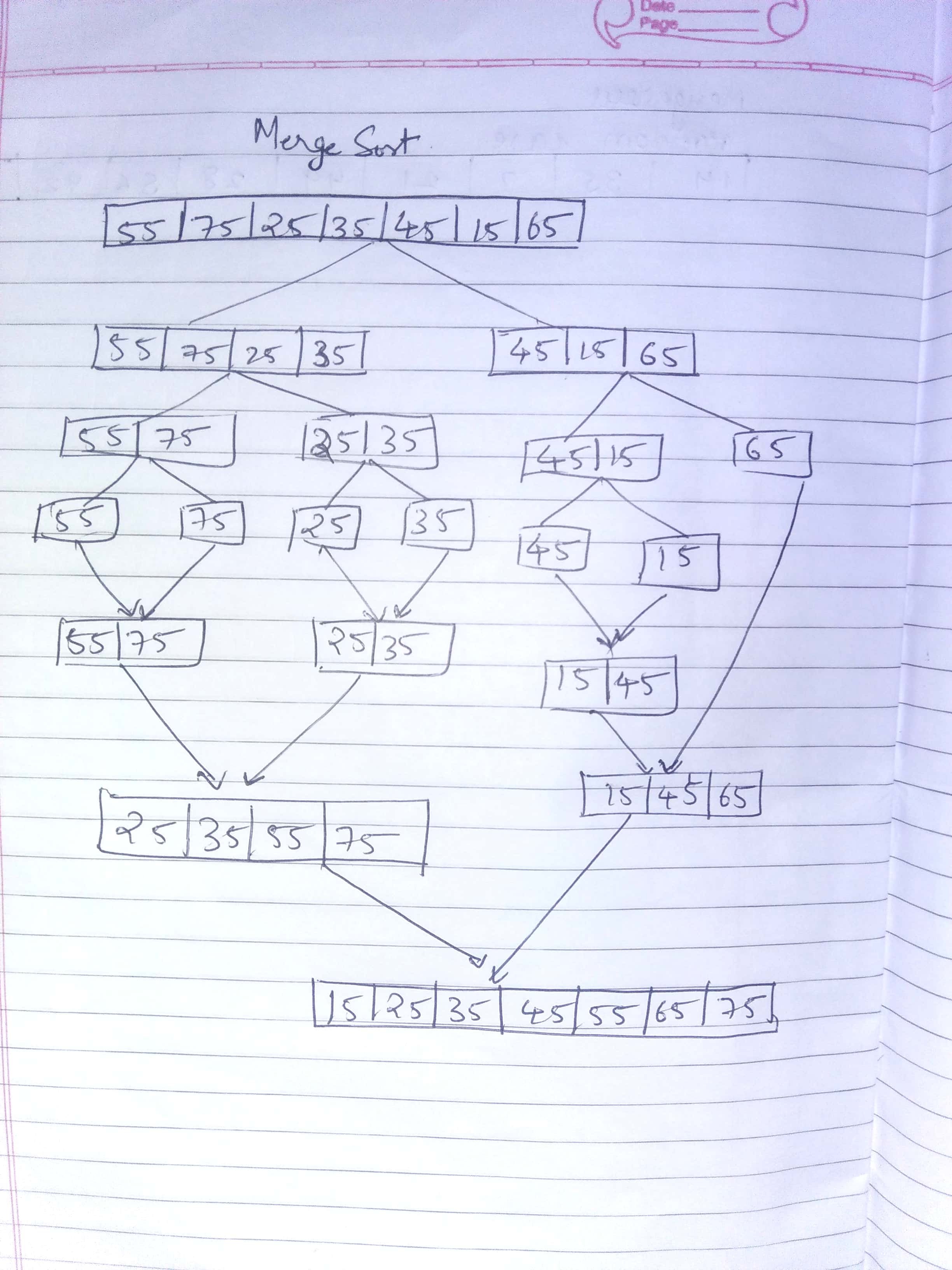
Unsorted array is

55 75 25 45 15 65

26 Comparisons made

Sorted array is

15 25 45 55 65 75



**QUICK SORT:-**

Quick sort is based on the divide-and-conquer approach based on the idea of choosing one element as a pivot element and partitioning the array around it such that: Left side of pivot contains all the elements that are less than the pivot element Right side contains all elements greater than the pivot

It reduces the space complexity and removes the use of the auxiliary array that is used in merge sort. Selecting a random pivot in an array results in an improved time complexity in most of the cases.

**Implementation** :

Select the first element of array as the pivot element First, we will see how the partition of the array takes place around the pivot.   
In the implementation below, the following components have been used: Here,

A[]= array whose elements are to be sorted

start: Leftmost position of the array

end: Rightmost position of the array

i: Boundary between the elements that are less than pivot and those greater than pivot

j: Boundary between the partitioned and unpartitioned part of array

piv: Pivot element

int partition ( int A[],int start ,int end) {

int v, i, j, temp;

v = a[l];

i = l;

j = u + 1;

do {

do

{ i++;}while (a[i] < v && i <= u);

do

{ j--;} while (v < a[j] && j>=l);

if (i < j) {

temp = a[i];

a[i] = a[j];

}

} while (i < j);

a[l] = a[j];

a[j] = v;

return (j);

//return the position of the pivot

}

Now, let us see the recursive function Quick\_sort :

void quick\_sort ( int A[ ] ,int start , int end ) {

if( start < end ) {

//stores the position of pivot element

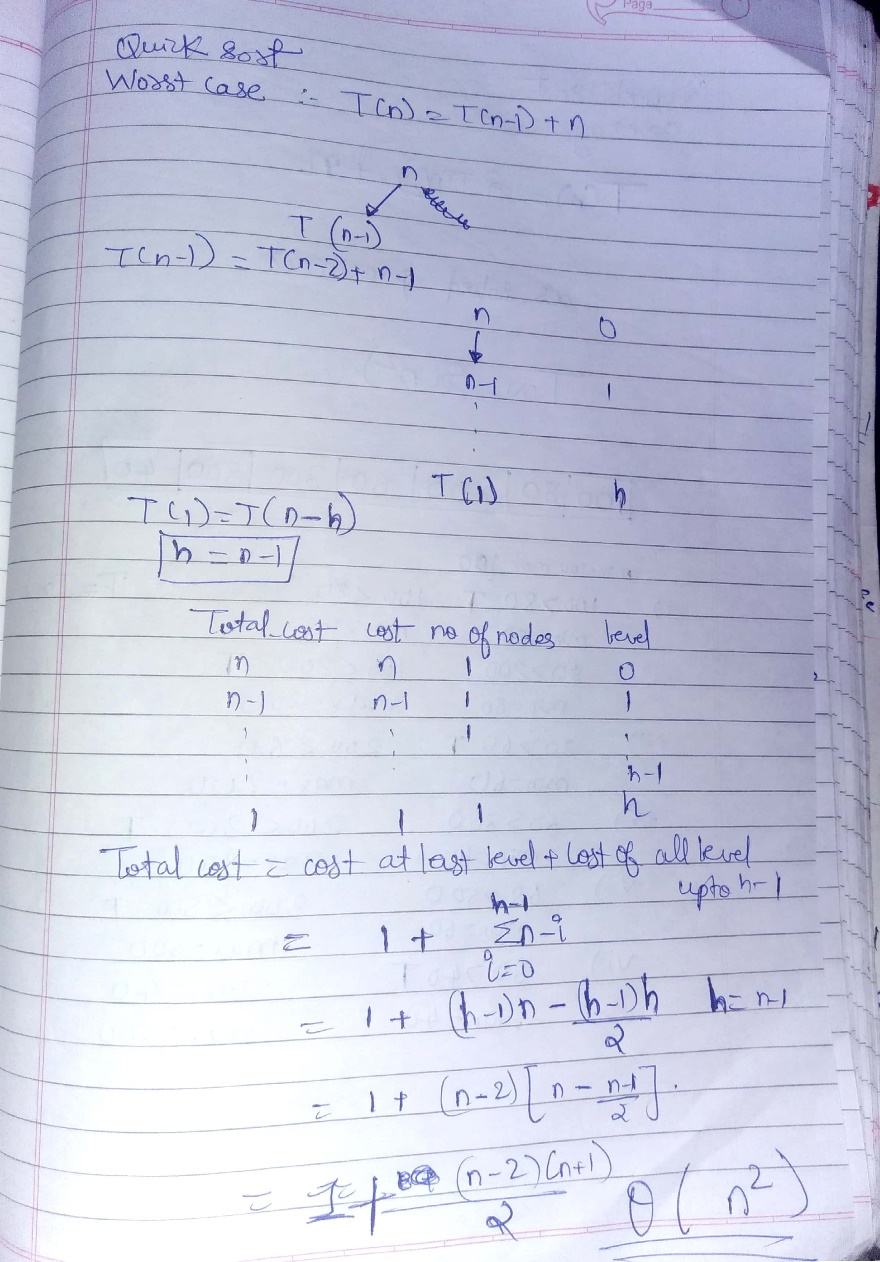
int piv\_pos = partition (A,start , end ) ;

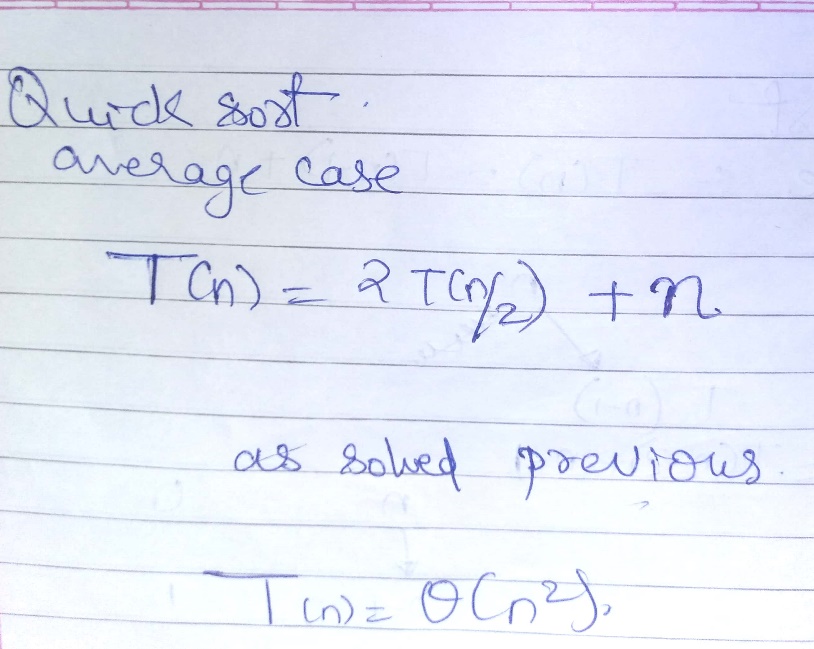
quick\_sort (A,start , piv\_pos -1); //sorts the left side of pivot.

quick\_sort ( A,piv\_pos +1 , end) ; //sorts the right side of pivot.

}

}





**Program:-**

#include <stdio.h>

void quick\_sort(int[], int, int,int\*,int\*);

int partition(int[], int, int,int\*,int\*);

int partition(int a[], int l, int u,int \*comp,int \*swap) {

int v, i, j, temp;

v = a[l];

i = l;

j = u + 1;

do {

do

{ i++;\*comp+=1;;} while (a[i] < v && i <= u);

do

{ j--;\*comp+=1;;} while (v < a[j] && j>=l);

if (i < j) {

temp = a[i];

a[i] = a[j];

a[j] = temp;

\*swap+=1;;

\*comp+=1;

}

\*comp+=1;;

} while (i < j);

a[l] = a[j];

a[j] = v;

\*swap+=1;

return (j);

}

void quick\_sort(int a[], int l, int u,int \*comp,int \*swap) {

int j;

if (l < u) {

j = partition(a, l, u,comp,swap);

quick\_sort(a, l, j - 1,comp,swap);

quick\_sort(a, j + 1, u,comp,swap);

}

}

int main() {

int a[7]={15,25,35,45,55,65,75};

int i=0,n=7;

printf("\nWorst case\n");

printf("\nArray before sorting:\n");

for (i = 0; i < n; i++)

printf("%d ", a[i]);

int swap=0,comp=0;

quick\_sort(a, 0, n - 1,&comp,&swap);

printf("\nArray after sorting:");

printf("%d comparisions , %d swaps performed\n",comp,swap);

for (i = 0; i < n; i++)

printf("%d ", a[i]);

printf("\n\nAverage case");

int b[7]={55,75,25,35,45,15,65};

printf("\nArray before sorting:\n");

for (i = 0; i < n; i++)

printf("%d ", b[i]);

i=0,swap=0,comp=0;

quick\_sort(b, 0, n - 1,&comp,&swap);

printf("\nArray after sorting:");

printf("%d comparisions , %d swaps performed\n",comp,swap);

for (i = 0; i < n; i++)

printf("%d ", b[i]);

return 0;

}

**OUTPUT:-**

**Worst case**

**Array before sorting:**

**15 25 35 45 55 65 75**

**Array after sorting:39 comparisions , 6 swaps performed**

**15 25 35 45 55 65 75**

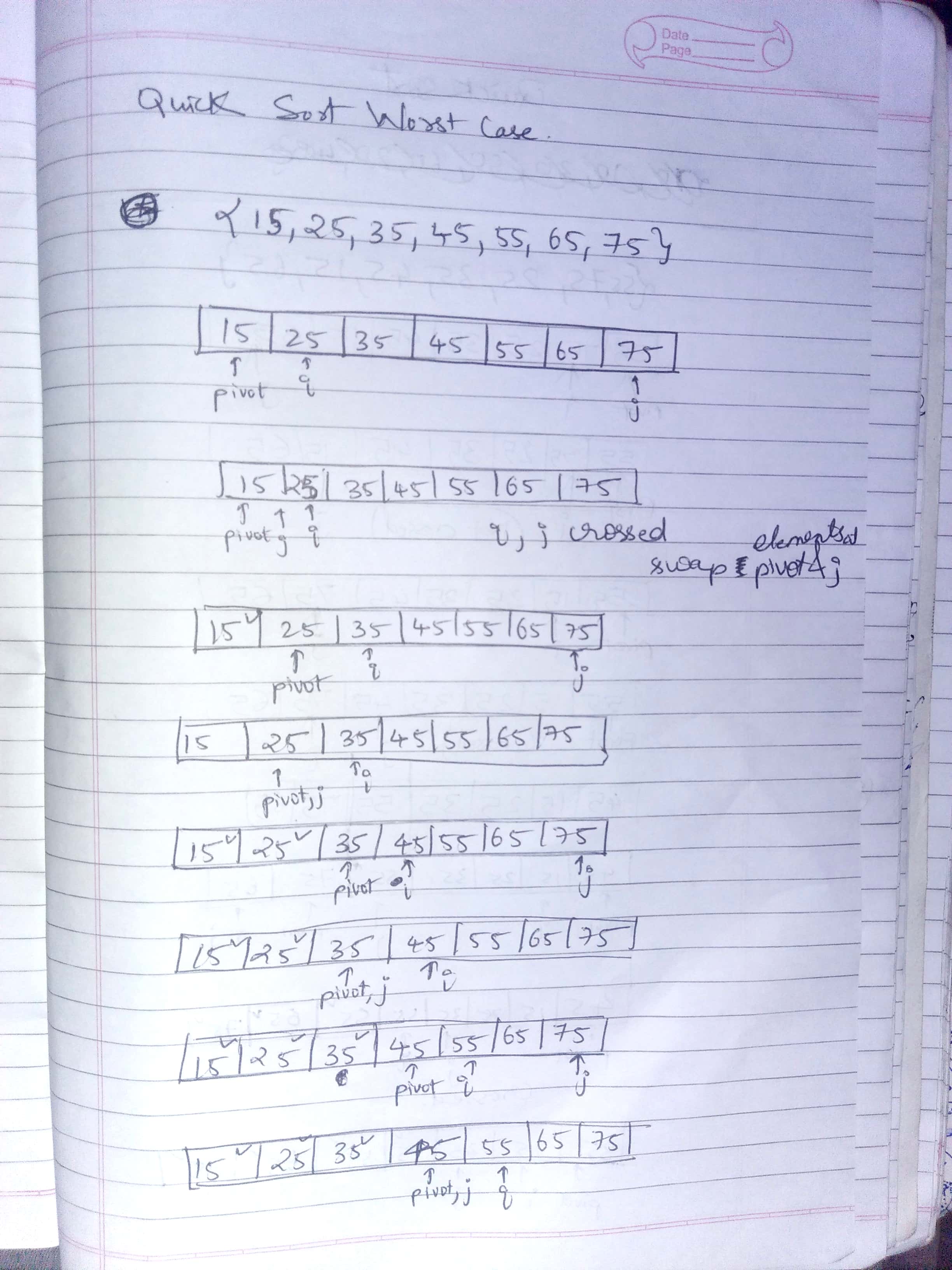
**Average case**

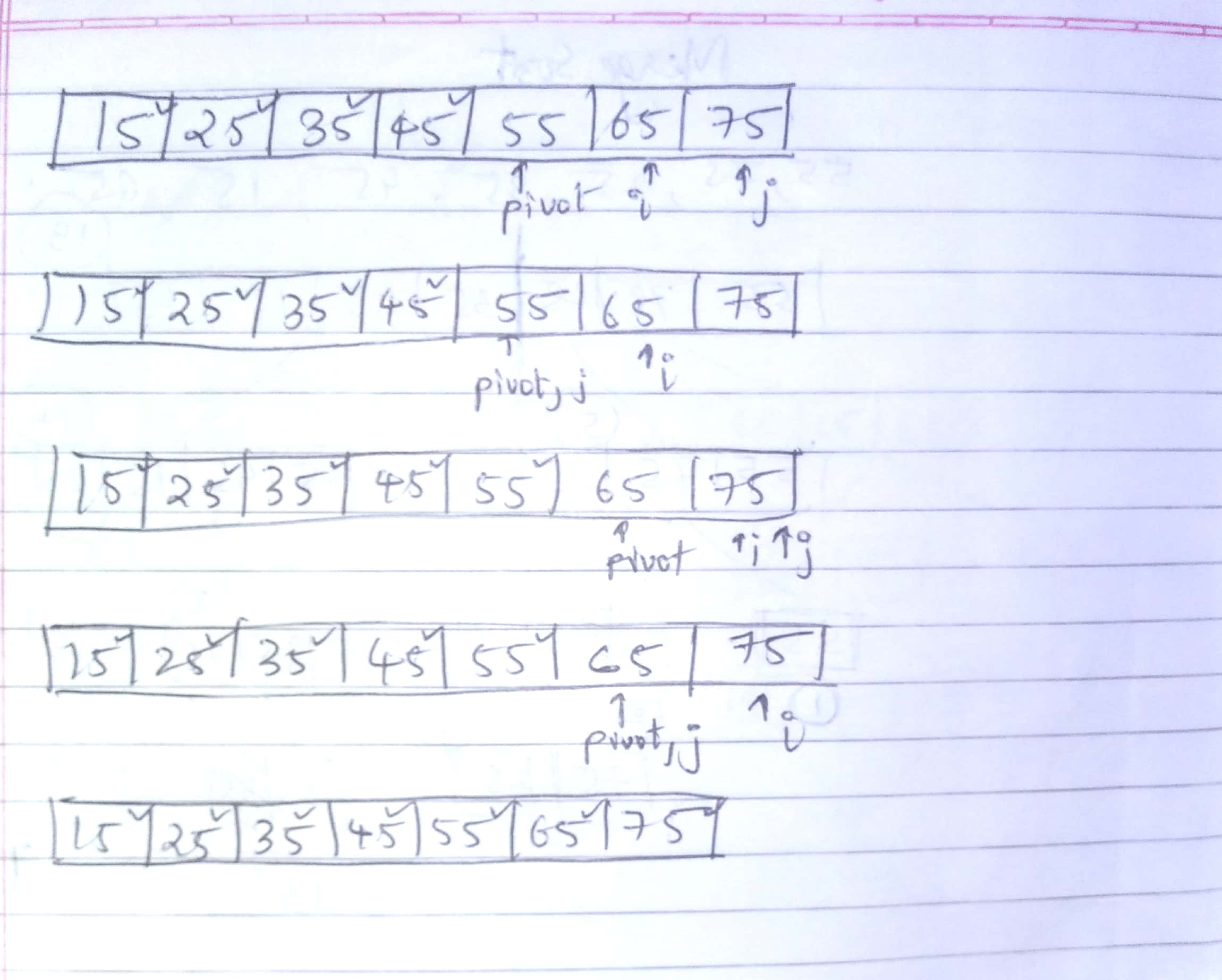
**Array before sorting:**

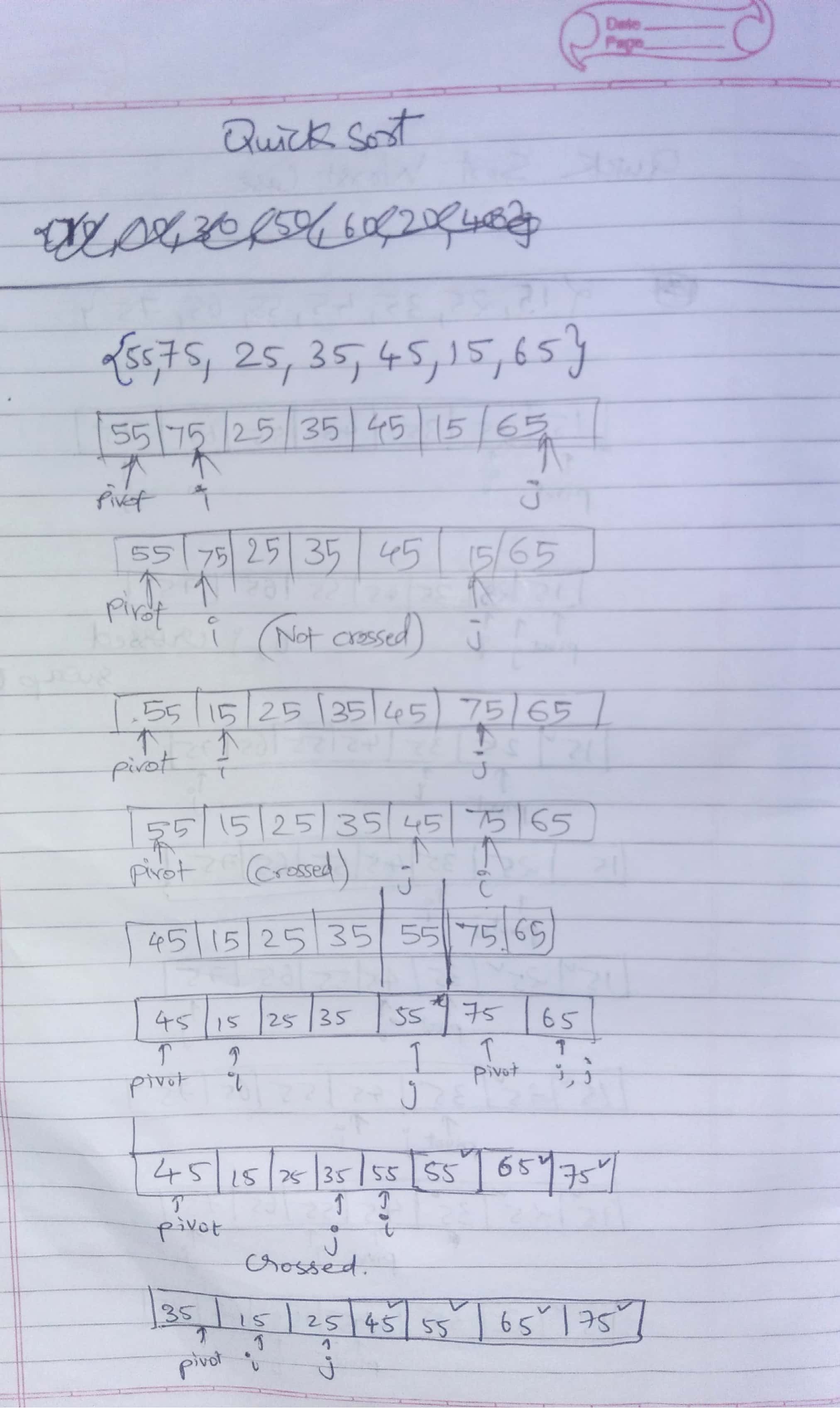
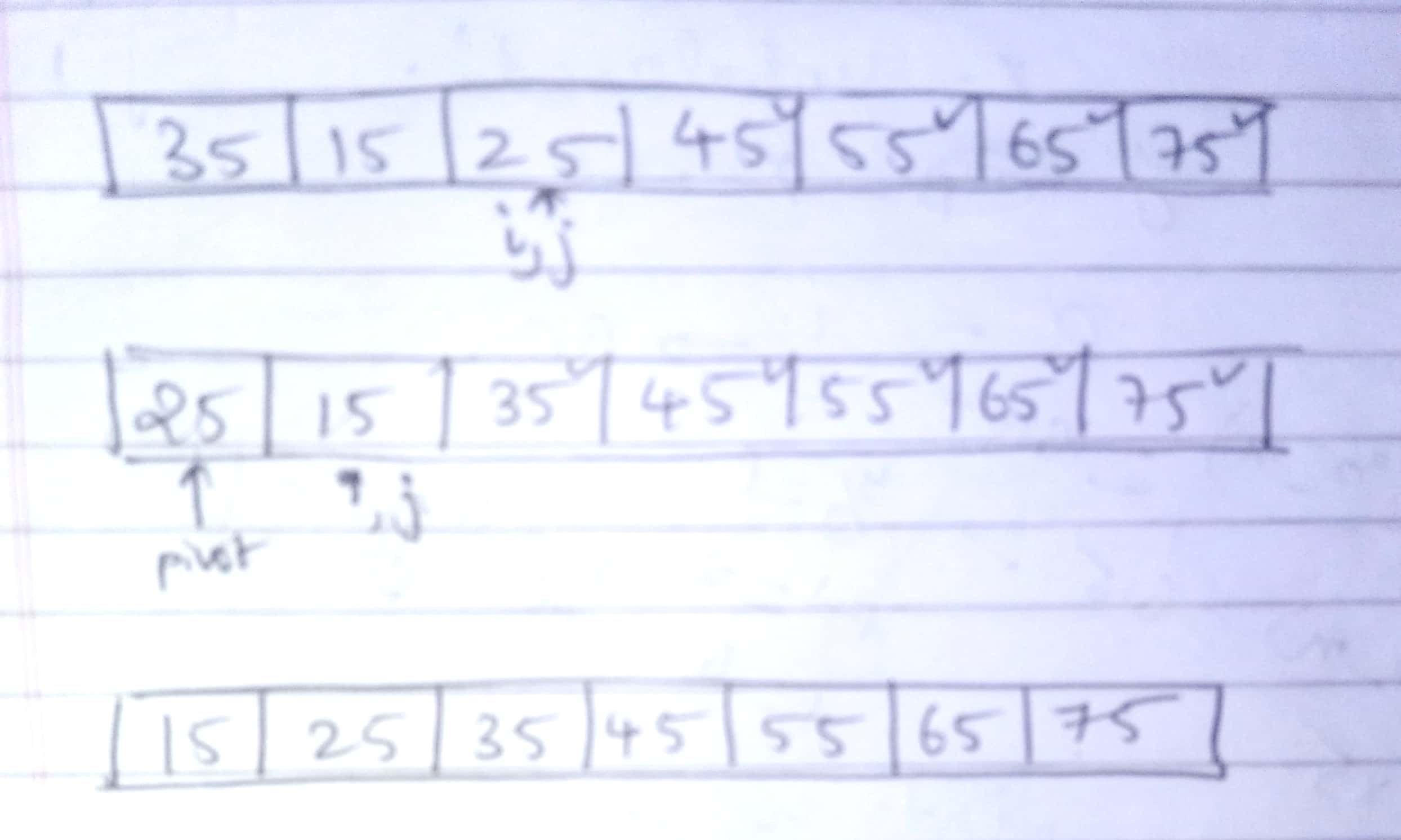
**55 75 25 35 45 15 65**

**Array after sorting:30 comparisions , 6 swaps performed**

**15 25 35 45 55 65 75**





Conclusion:-

1. **Partition of elements in the array** :  
   In the merge sort, the array is parted into just 2 halves (i.e. n/2).  
   whereas  
   In case of quick sort, the array is parted into any ratio. There is no compulsion of dividing the array of elements into equal parts in quick sort.
2. **Worst case complexity** :  
   The worst case complexity of quick sort is O(n2) as there is need of lot of comparisons in the worst condition.  
   whereas  
   In merge sort, worst case and average case has same complexities O(n log n).
3. **Additional storage space requirement** :  
   Merge sort is not in place because it requires additional memory space to store the auxiliary arrays.  
   whereas  
   The quick sort is in place as it doesn’t require any additional storage.
4. **Efficiency** :  
   Merge sort is more efficient and works faster than quick sort in case of larger array size or datasets.  
   whereas  
   Quick sort is more efficient and works faster than merge sort in case of smaller array size or datasets.
5. **Sorting method** :  
   The quick sort is internal sorting method where the data is sorted in main memory.  
   whereas  
   The merge sort is external sorting method in which the data that is to be sorted cannot be accommodated in the memory and needed auxiliary memory for sorting.