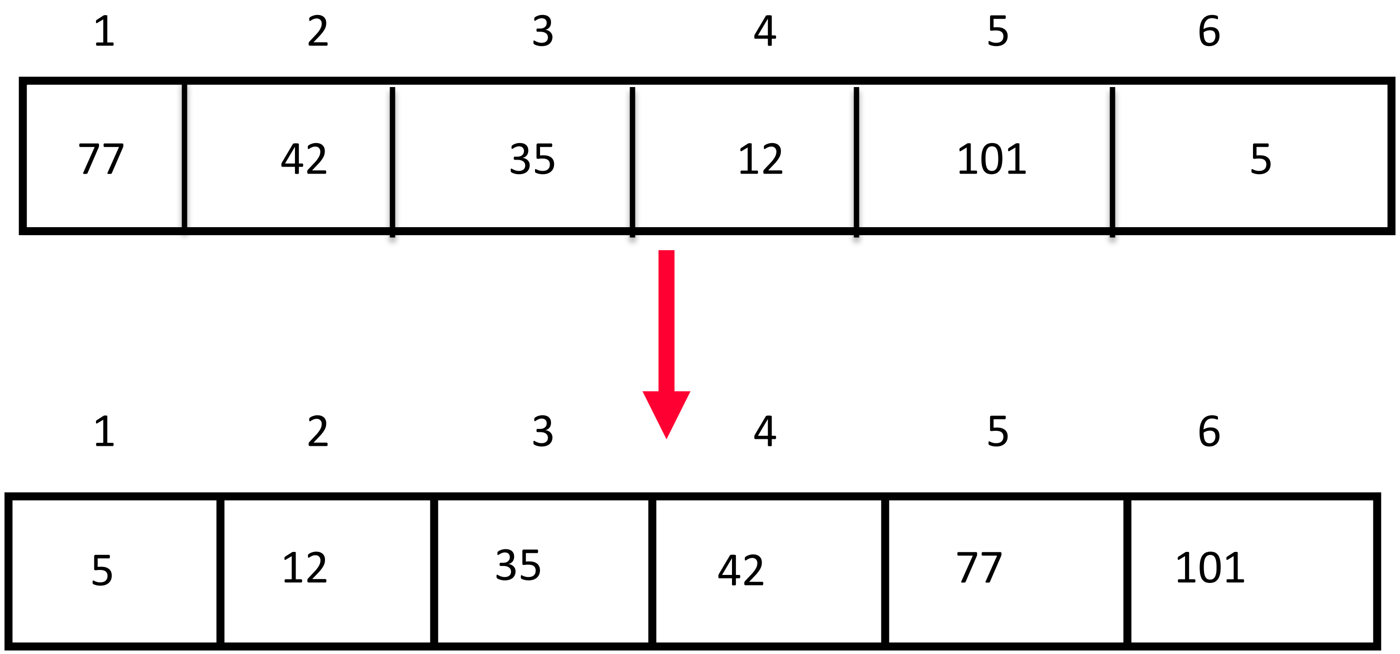
Sorting Unit-6

- Sorting is a rearrangement of data in some logical order, it may be in ascending or descending order.
- Sorting takes an unordered collection and makes it an ordered one.
- The importance of sorting lies in the fact that data searching can be optimized to a very high level, if data is stored in a sorted manner.



Examples of Sorting:

- Words in a dictionary are sorted so that searching of any word becomes easy.
- Files in a directory are often listed in sorted order.
- The index of a book is sorted.
- Many banks provide statements that list cheque in increasing order (by cheque number).
- records in database.
- telephone number in telephone directory.

Why Sorting?

- Imagine finding the phone number of your friend in your mobile phone, but the phone book is not sorted.
- Imagine finding the meaning of particular words in a dictionary but it is not sorted.

Different Sorting Algorithms

- There are many different techniques available for sorting, differentiated by their efficiency (time complexity and space requirement).
- Following are some types of sorting techniques:

Comparison sorting algorithm

1. Selection Sort
2. Bubble Sort
3. Insertion Sort
4. Shell Sort

Divide and Conquer sorting algorithm

1. Merge Sort
2. Heap Sort

1. Selection Sort Algorithm

- Selection sort is conceptually the most simplest sorting algorithm.

- This algorithm will first find the **smallest** element in the array and swap it with the element in the **first** position, then it will find the **second smallest** element and swap it with the element in the **second** position, and it will keep on doing this until the entire array is sorted.

- It is called selection sort because it repeatedly **selects** the next-smallest element and swaps it into the right place.

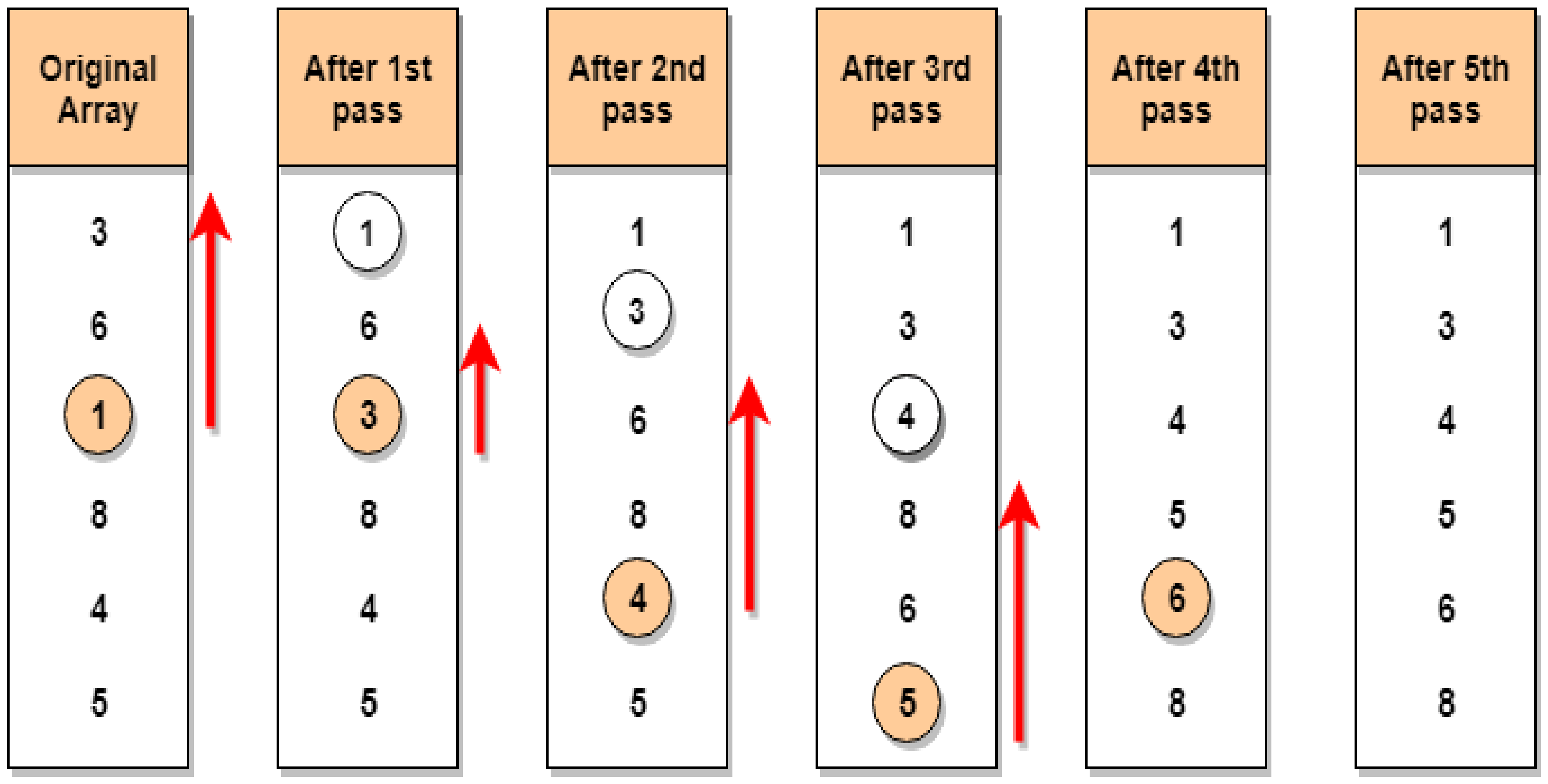
How does Selection Sort work?

1. Start with the 1st element, scan the entire list to find its smallest element and exchange it with the 1st element.
2. Start with the 2nd element, scan the remaining list to find the smallest among the last (N-1) elements and exchange it with the 2nd element.
3. Continue this process….

Example: **89 45 68 90 29 34 17**

```
17 | 45   68   90   29   34   89
17   29 | 68   90   45   34   89
17   29   34 | 90   45   68   89
17   29   34   45 | 90   68   89
17   29   34   45   68 | 90   89
17   29   34   45   68   89 | 90
17   29   34   45   68   89   90
```

- Pictorial representation of how selection sort will sort the given array. Let's consider an array with values {3, 6, 1, 8, 4, 5}



Q. Sort the following using selection sort.

89 45 68 90 29 34 17

Ans: Given number:

89 45 68 90 29 34 17

Pass 1: 17 | 45 68 90 29 34 89

Pass 2: 17 29 |68 90 45 34 89

Pass 3: 17 29 34 |90 45 68 89

Pass 4: 17 29 34 45 |90 68 89

Pass 5: 17 29 34 45 68 |90 89

Pass 6: 17 29 34 45 68 89 |90

Pass 7: 17 29 34 45 68 89 90

// C program Selection Sort [Method 1] sel1.cpp

```
#include<stdio.h>
#include<conio.h>
 // function to swap elements at the given index values
void swap(int arr[], int firstIndex, int secondIndex)
{
int temp;
temp = arr[firstIndex];
arr[firstIndex] = arr[secondIndex];
arr[secondIndex] = temp;
}

// function to look for smallest element in the given subarray
int indexOfMinimum(int arr[], int startIndex, int n)
{
int minValue = arr[startIndex];
int minIndex = startIndex;

for(int i = minIndex + 1; i < n; i++)
{
if(arr[i] < minValue)
{
minIndex = i;
minValue = arr[i];
}
}
return minIndex;
}

void selectionSort(int arr[], int n)
{
for(int i = 0; i < n; i++)
{
1.        int index = indexOfMinimum(arr, i, n);
swap(arr, i, index);
}
}

void printArray(int arr[], int size)
{
int i;
for(i = 0; i < size; i++)
{
printf("\n%d\n ", arr[i]);
}
printf("\n");
}

int main()
{
int arr[20],n,i;
//int arr[]={10,30,2,1,4};
clrscr();
printf("Enter number of elements:");
scanf("%d", &n);
printf("Enter %d Numbers:\n", n);
for (i = 0; i < n; i++)
scanf("%d", &arr[i]);
//int n = sizeof(arr)/sizeof(arr[0]);
selectionSort(arr, n);
printf("Sorted array: \n");
printArray(arr, n);
getch();
return 0;
}
```

//a c program to sort the element using selection sort method2.

```
#include<stdio.h>
#include<conio.h>
int main()
{
int a[100], n, i, j, position, swap;
clrscr();
printf("Enter number of elements:");
scanf("%d", &n);
printf("Enter %d Numbers:", n);
for (i = 0; i < n; i++)
scanf("%d", &a[i]);
for(i = 0; i < n - 1; i++)
{
position=i;
for(j = i + 1; j < n; j++)
{
if(a[position] > a[j])
position=j;
}
if(position != i)
{
swap=a[i];
a[i]=a[position];
a[position]=swap;
}
}
printf("Sorted Array:\n");
for(i = 0; i < n; i++)
printf("%d\n", a[i]);
getch();
return 0;
}
```

Complexity of Selection Sort

The first pass requires (n-1) comparisions to find the location of smallest element. The second pass requires (n-2) comparisons and so on. The last pass requires only 1 comparison. So total number of comparisons are

$F(n) = (n-1)+(n-2)+(n-3)+ \ldots\ldots\ldots + 3+2+1$

$\quad =(n-1)(n-2)/2$

$F(n) = O(n^2)$

For a given input size of n, following will be the time and space complexity for selection sort algorithm:

- Worst Case Time Complexity [Big-O]: **$O(n^2)$**
- Best Case Time Complexity [Big-omega]: **$O(n^2)$**
- Average Time Complexity [Big-theta]: **$O(n^2)$**
- Space Complexity: **$O(1)$**

Q. Sort the following number using selection sort:

20 35 40 100 3 10 15

Q. Sort the following number using selection sort:

5, 1, 6, 2, 4, 3

2. Bubble Sort Algorithm

- **Bubble Sort** is a simple algorithm which is used to sort a given set of n elements provided in form of an array with n number of elements.
- Bubble Sort compares all the element one by one and sort them based on their values.
- If the given array has to be sorted in ascending order, then bubble sort will start by comparing the first element of the array with the second element, if the first element is greater than the second element, it will **swap** both the elements, and then move on to compare the second and the third element, and so on.
- If we have total n elements, then we need to repeat this process for n-1 times.

- It is known as **bubble sort**, because with every complete iteration the largest element in the given array, bubbles up towards the last place or the highest index, just like a water bubble rises up to the water surface.

- Sorting takes place by stepping through all the elements one-by-one and comparing it with the adjacent element and swapping them if required.

Implementing Bubble Sort Algorithm

Following are the steps involved in bubble sort(for sorting a given array in ascending order):

- Starting with the first element(index = 0), compare the current element with the next element of the array.
- If the current element is greater than the next element of the array, swap them.
- If the current element is less than the next element, move to the next element. **Repeat Step 1**

Q. Sort the numbers using bubble sort:

89 45 68 90 29 34 17

Pass 1

89	**45**	**68**	**90**	**29**	**34**	**17**
45	**89**	**68**	90	29	34	17
45	68	**89**	**90**	29	34	17
45	68	89	**90**	**29**	34	17
45	68	89	29	**90**	**34**	17
45	68	89	29	34	**90**	**17**
45	**68**	**89**	**29**	**34**	**17**	**90**

Pass 2

45	68	89	29	34	17	90
45	68	89	29	34	17	90
45	68	89	29	34	17	90
45	68	29	89	34	17	90
45	68	29	34	89	17	90
45	68	29	34	17	89	90
45	68	29	34	17	89	90

Pass 3

45	68	29	34	17	89	90
45	68	29	34	17	89	90
45	29	68	34	17	89	90
45	29	34	68	17	89	90
45	29	34	17	68	89	90
45	29	34	17	68	89	90
45	29	34	17	68	89	90

Pass 4

45	29	34	17	68	89	90
29	45	34	17	68	89	90
29	34	45	17	68	89	90
29	34	17	45	68	89	90
29	34	17	45	68	89	90
29	34	17	45	68	89	90
29	34	17	45	68	89	90

Pass 5

29	34	17	45	68	89	90
29	34	17	45	68	89	90
29	17	34	45	68	89	90
29	17	34	45	68	89	90
29	17	34	45	68	89	90
29	17	34	45	68	89	90
29	17	34	45	68	89	90

Pass 6

29	17	34	45	68	89	90
17	29	34	45	68	89	90
17	29	34	45	68	89	90
17	29	34	45	68	89	90
17	29	34	45	68	89	90
17	29	34	45	68	89	90
17	29	34	45	68	89	90

Hence the sorted data using bubble sort is:17 29 34 45 68 89 90

Q. Sort the following number using bubble sort:

20 35 40 100 3 10 15

Q. Sort the following number using bubble sort:

5, 1, 6, 2, 4, 3

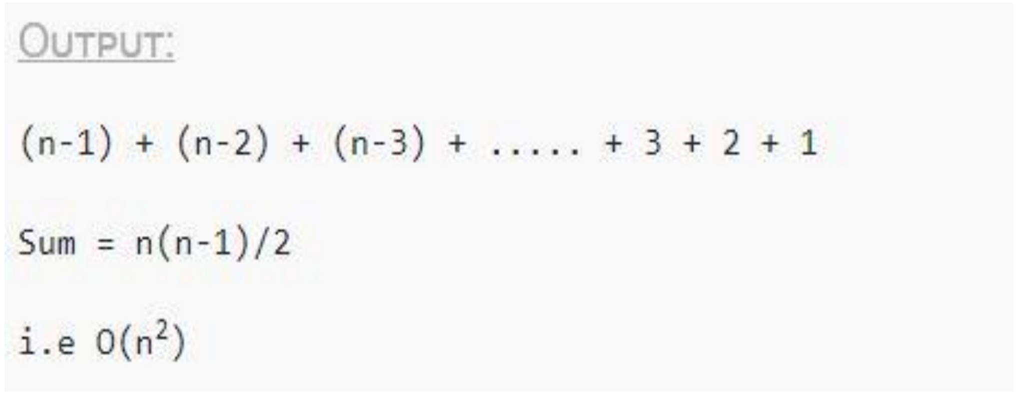
// program for bubble sort(bubble2.cpp)

```
#include <stdio.h>
#include <conio.h>

void bubbleSort(int arr[], int n)
{
int i, j, temp, flag=0;
for(i = 0; i < n; i++)
{
for(j = 0; j < n-i-1; j++)
{
// introducing a flag to monitor swapping
if( arr[j] > arr[j+1])
{
// swap the elements
temp = arr[j];
arr[j] = arr[j+1];
arr[j+1] = temp;
// if swapping happens update flag to 1
flag = 1;
}
}
// if value of flag is zero after all the iterations of inner loop
// then break out
if(flag==0)
{
break;
}
}
// print the sorted array
printf("Sorted Data in Array are: ");
for(i = 0; i < n; i++)
{
printf("%d  ", arr[i]);
}
}

int main()
{
int arr[100], i, n, step, temp;
clrscr();
// ask user for number of elements to be sorted
printf("Enter the number of elements to be sorted: ");
scanf("%d", &n);
// input elements if the array
for(i = 0; i < n; i++)
{
printf("Enter element no. %d: ", i+1);
scanf("%d", &arr[i]);
}
// call the function bubbleSort
bubbleSort(arr, n);
getch();
return 0;
}
```

```
// a simple C program for bubble sort
//bubble.cpp
#include<stdio.h>
#include<conio.h>

void bubble_sort(int a[], int n)
{
int i, j, t;
for(i = 0; i < n; i++)
{
for(j = 0; j < n-i-1; j++)
{
if( arr[j] > arr[j+1])
{
// swap the elements
t = arr[j];
arr[j] = arr[j+1];
arr[j+1] = temp;
}
}
}

// print the sorted array
printf("Sorted Array:\n");
for(i = 0; i < n; i++)
{
printf("%d\n", a[i]);
}
}

int main()
{
int arr[100], i, n;
clrscr();
printf("Enter the number of elements to be
sorted: ");
scanf("%d", &n);
Printf("Enter %d numbers",n);
for(i = 0; i < n; i++)
{
scanf("%d", &a[i]);
}

bubble_sort(a, n);
getch();
return 0;
}
```

Complexity Analysis of Bubble Sort

- In Bubble Sort, n-1 comparisons will be done in the 1st pass, n-2 in 2nd pass, n-3 in 3rd pass and so on. So the total number of comparisons will be,



OUTPUT:

```
(n-1) + (n-2) + (n-3) + ..... + 3 + 2 + 1

Sum = n(n-1)/2

i.e O(n²)
```

- Hence the **time complexity** of Bubble Sort is **O(n^2)**.
- The **space complexity** for Bubble Sort is **O(1)**, because only a single additional memory space is required i.e. for temp variable.
- Also, the **best case time complexity** will be **O(n)**, it is when the list is already sorted.

Following are the Time and Space complexity for the Bubble Sort algorithm.

- Worst Case Time Complexity [Big-O]: **$O(n^2)$**
- Best Case Time Complexity [Big-omega]: **$O(n)$**
- Average Time Complexity [Big-theta]: **$O(n^2)$**
- Space Complexity: **$O(1)$**

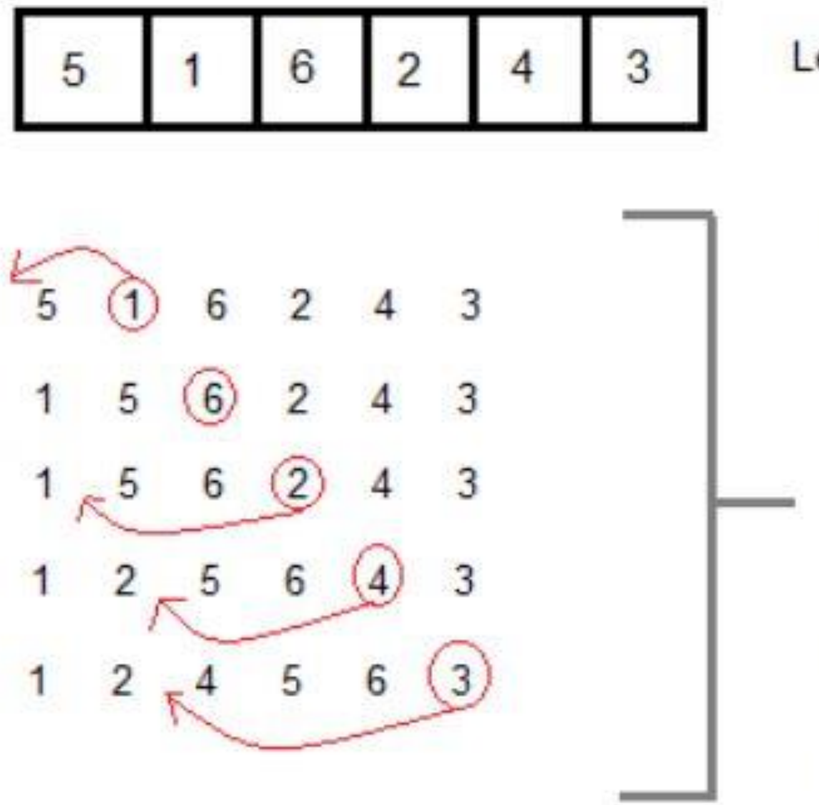
3. Insertion Sort Algorithm

- Consider you have 10 cards out of a deck of cards in your hand. And they are sorted, or arranged in the ascending order of their numbers.
- If I give you another card, and ask you to **insert** the card in just the right position, so that the cards in your hand are still sorted. What will you do?
- Well, you will have to go through each card from the starting or the back and find the right position for the new card, comparing it's value with each card. Once you find the right position, you will **insert** the card there.
- Similarly, if more new cards are provided to you, you can easily repeat the same process and insert the new cards and keep the cards sorted too.
- This is exactly how **insertion sort** works. It is efficient for smaller data sets, but very inefficient for larger lists.

How Insertion Sort Works?

- We start by making the second element of the given array, i.e. element at index 1, the key. The key element here is the new card that we need to add to our existing set of cards.
- We compare the key element with the element(s) before it, in this case, element at index 0:
  - If the key element is less than the first element, we insert the key element before the first element.
  - If the key element is greater than the first element, then we insert it after the first element.
- Then, we make the third element of the array as key and will compare it with elements to it's left and insert it at the right position.
- And we go on repeating this, until the array is sorted.

- Let’s understand it with an example. Just carefully read the steps in the image below and you will be able to visualize the concept of this algorithm.



Lets take this Array.

As we can see here, in insertion sort, we pick up a key, and compares it with elemnts ahead of it, and puts the key in the right place

5 has nothing before it.

1 is compared to 5 and is inserted before 5.

6 is greater than 5 and 1.

2 is smaller than 6 and 5, but greater than 1, so its is inserted after 1.

And this goes on...

(Always we start with the second element as key.)

Sort the following number using insertion sort:
8, 5, 9, 2, 6, 3

```
Step-1: 8 | 5 9 2 6 3
Step-1: 5 8 | 9 2 6 3
Step-1: 5 8 9 | 2 6 3
Step-1: 2 5 8 9 | 6 3
Step-1: 2 5 6 8 9 | 3
Step-1: 2 3 5 6 8 9 |
```

//Sort elements of an array in ascending order using insertion sort algorithm

```
#include<stdio.h>
#include<conio.h>
void main()
{
int data[100],n,temp,i,j;
clrscr();
printf("Enter number of terms(should be less than 100): ");
scanf("%d",&n);
printf("Enter %d elements:\n",n);
for(i=0;i<n;i++)
{
scanf("%d",&data[i]);
}
for(i=1;i<n;i++)
{
temp = data[i];
j=i-1;
while(temp<data[j] && j>=0)
/*To sort elements in descending order, change temp<data[j]
to temp>data[j] in above line.*/
{
data[j+1] = data[j];
--j;
}
data[j+1]=temp;
}
printf("the data in ascending order are as follows:\n");
for(i=0; i<n; i++)
printf("%d\n",data[i]);
getch();
}
```

Q. Sort the following number using insertion sort:

20 35 40 100 3 10 15

Q. Sort the following number using insertion sort:

23 78 45 8 32 56

Shell Sort

4. Merge Sort Algorithm

- Merge Sort follows the rule of **Divide and Conquer** to sort a given set of numbers/elements.

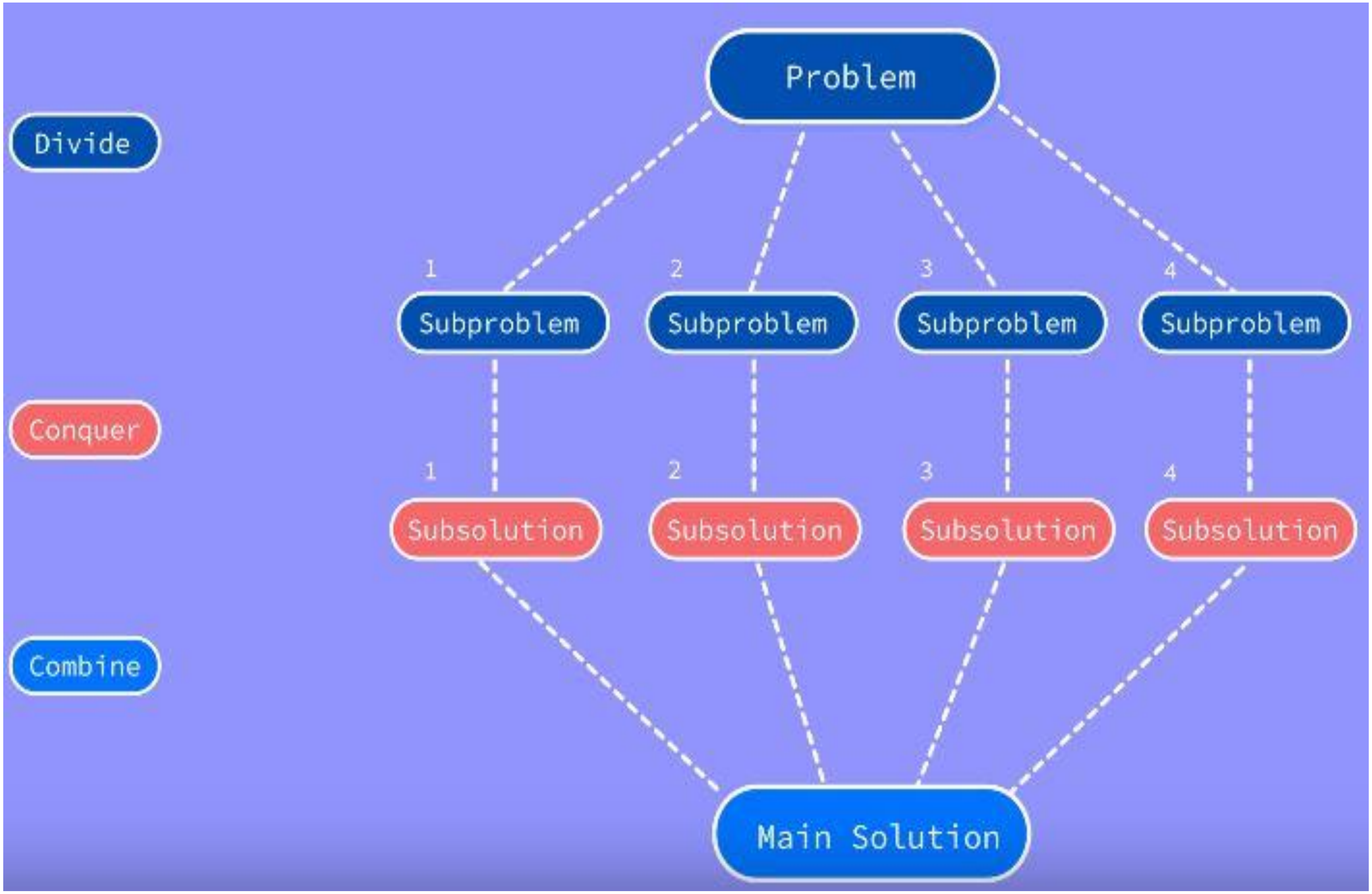
- The concept of Divide and Conquer involves three steps:

a. **Divide:** break the problem into multiple sub problems.

b. **Conquer:** solve the sub-problems.

c. **Combine**: arrange the solutions of the sub-problems to find the solution of the actual problem.

- In **Merge Sort**, the given unsorted array with n elements, is divided into n sub-arrays, each having **one** element, because a single element is always sorted in itself. Then, it repeatedly merges these sub-arrays, to produce new sorted sub-arrays, and in the end, one complete sorted array is produced.


Problem
Divide
1
Subproblem
2
Subproblem
3
Subproblem
4
Subproblem
Conquer
1
Subsolution
2
Subsolution
3
Subsolution
4
Subsolution
Combine
Main Solution

How Merge Sort Works?

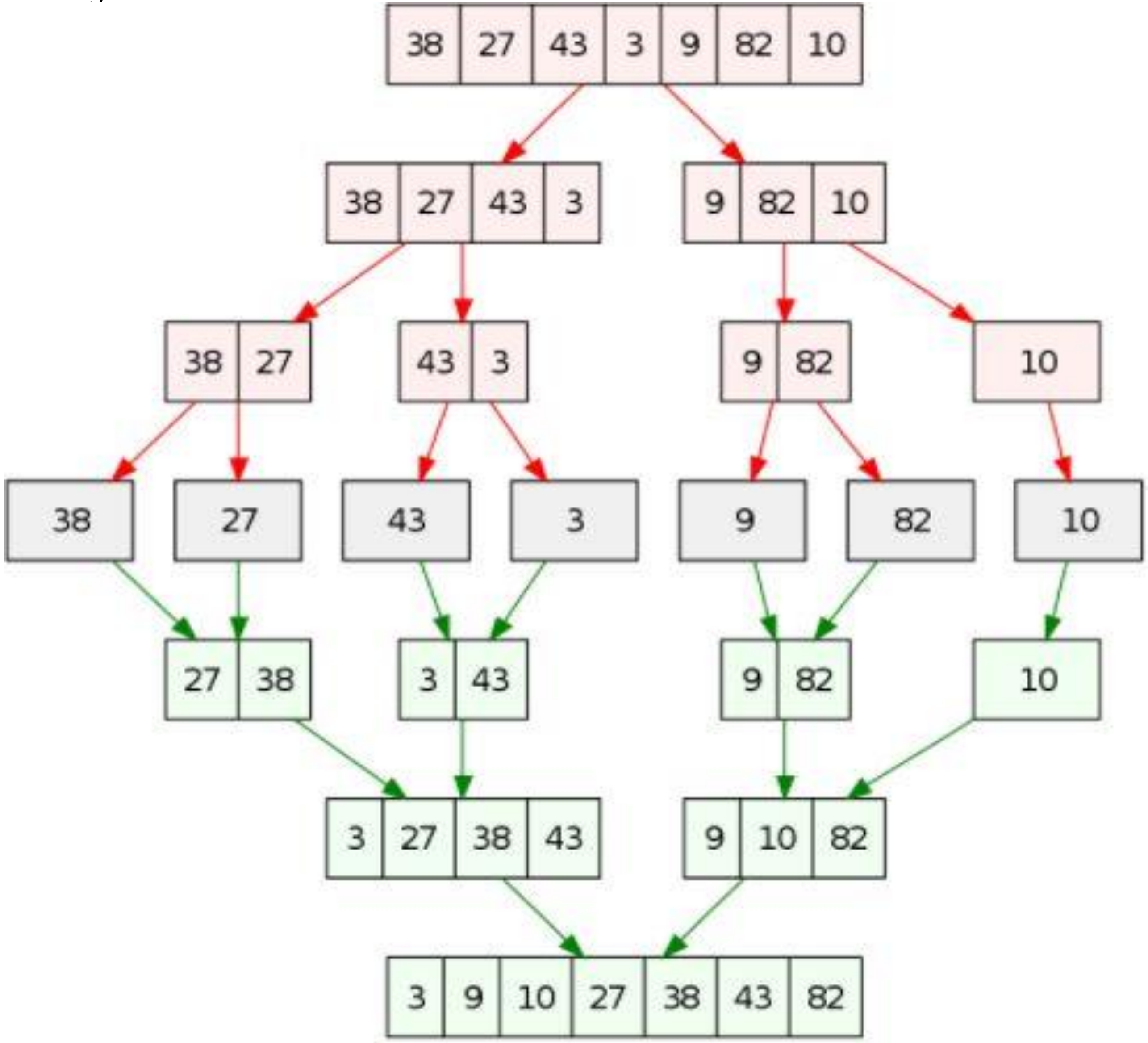
- In merge sort, we break the given array midway, for example if the original array had 6 elements, then merge sort will break it down into two sub-arrays with 3 elements each.
- But breaking the original array into 2 smaller sub-arrays is not helping us in sorting the array.
- So we will break these sub-arrays into even smaller sub-arrays, until we have multiple sub-arrays with **single element** in them. Now, the idea here is that an array with a single element is already sorted, so once we break the original array into sub-arrays which has only a single element, we have successfully broken down our problem.
- And then we have to merge all these sorted sub-arrays, step by step to form one single sorted array.

In merge sort we follow the following steps:

- We take a variable p and store the starting index of our array in this. And we take another variable r and store the last index of array in it.
- Then we find the middle of the array using the formula (p + r)/2 and mark the middle index as q, and break the array into two sub-arrays, from p to q and from q + 1 to r index.
- Then we divide these 2 sub-arrays again, just like we divided our main array and this continues.
- Once we have divided the main array into sub-arrays with single elements, then we start merging the sub-arrays.

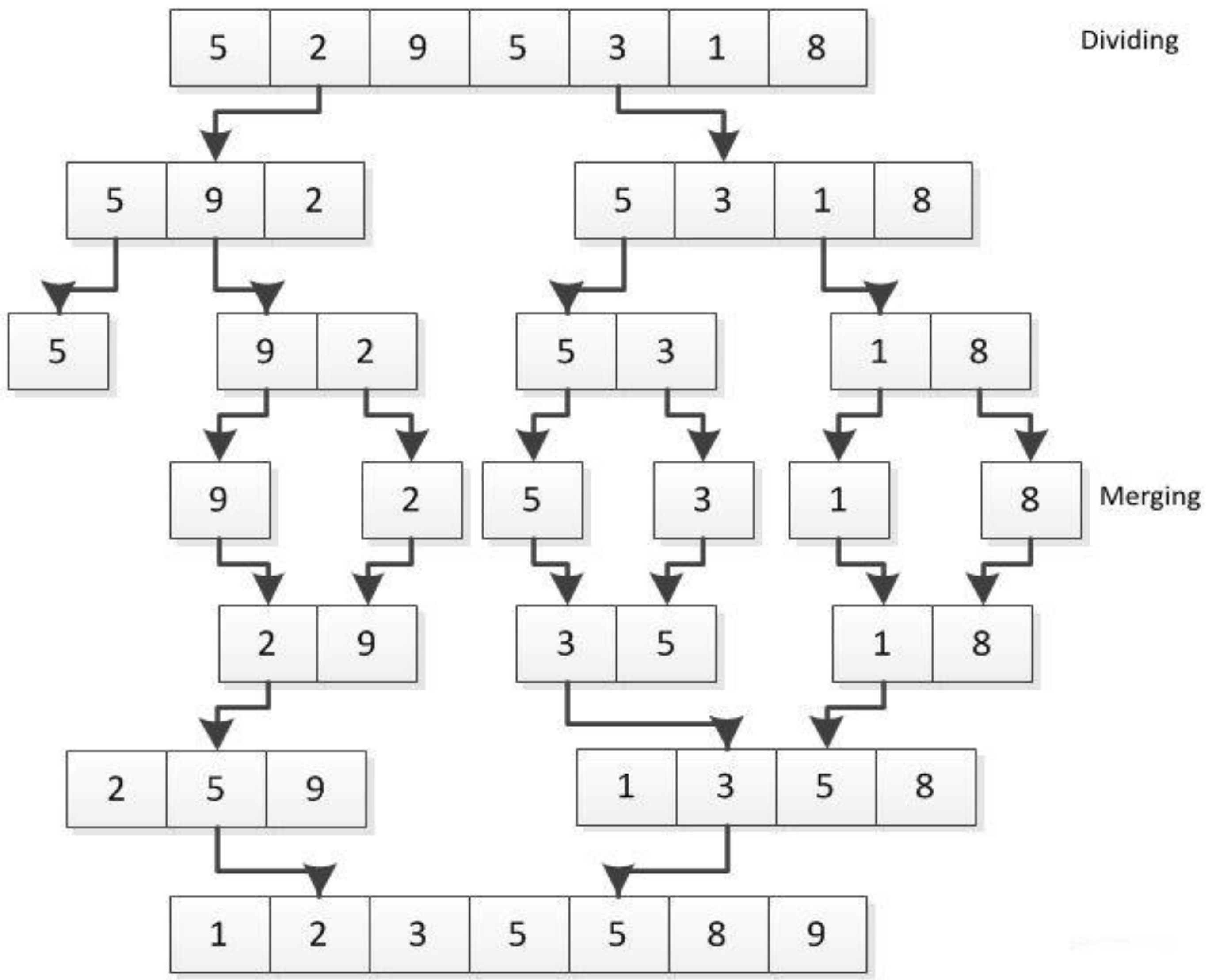
Example1:

Let's consider an array with values 38, 27, 43, 3, 9, 82, 10. Below, we have a pictorial representation of how merge sort will sort the given array.



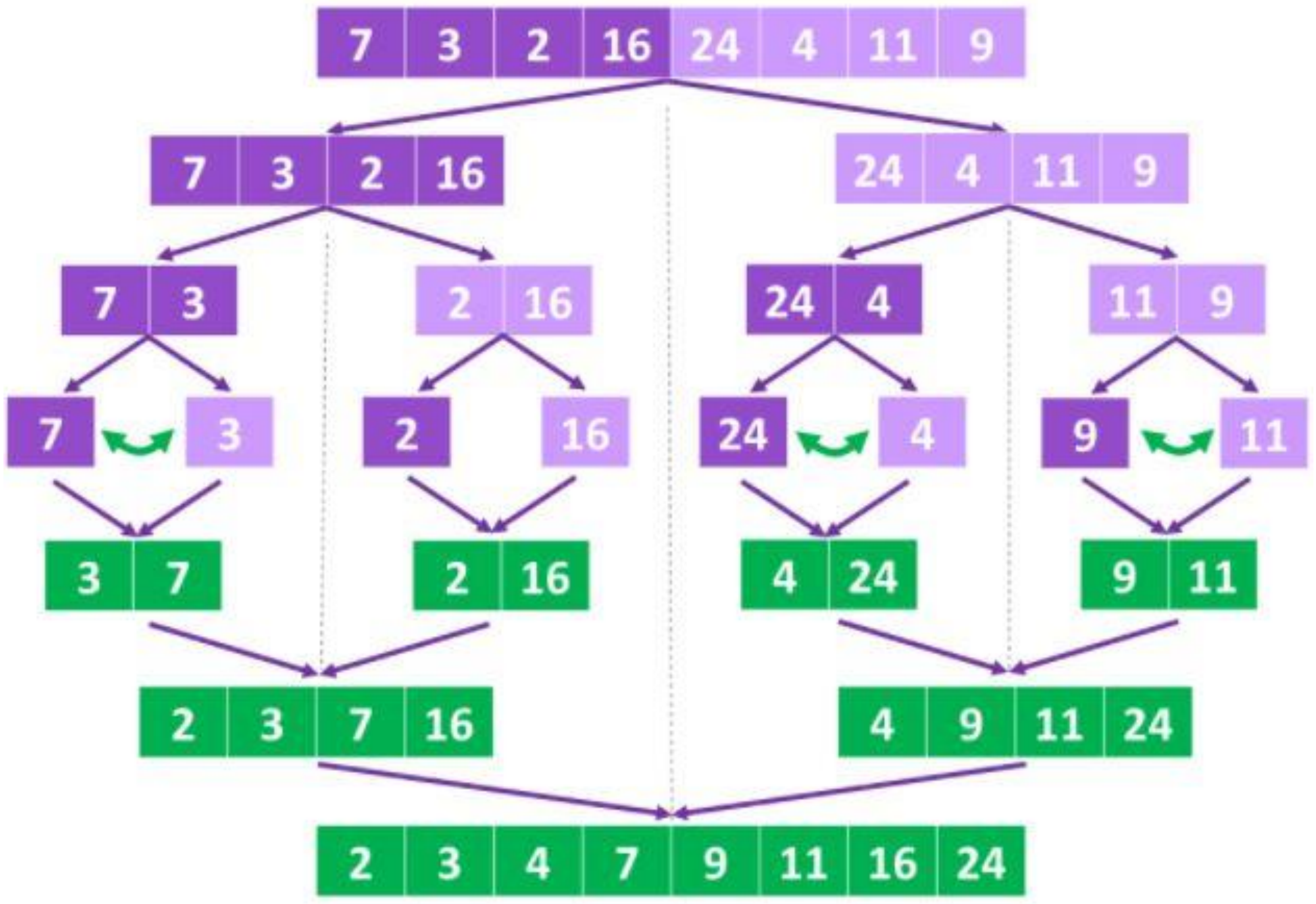
Example2:

Let's consider an array with values 5, 2, 9, 5, 3, 1, 8. Below, we have a pictorial representation of how merge sort will sort the given array.



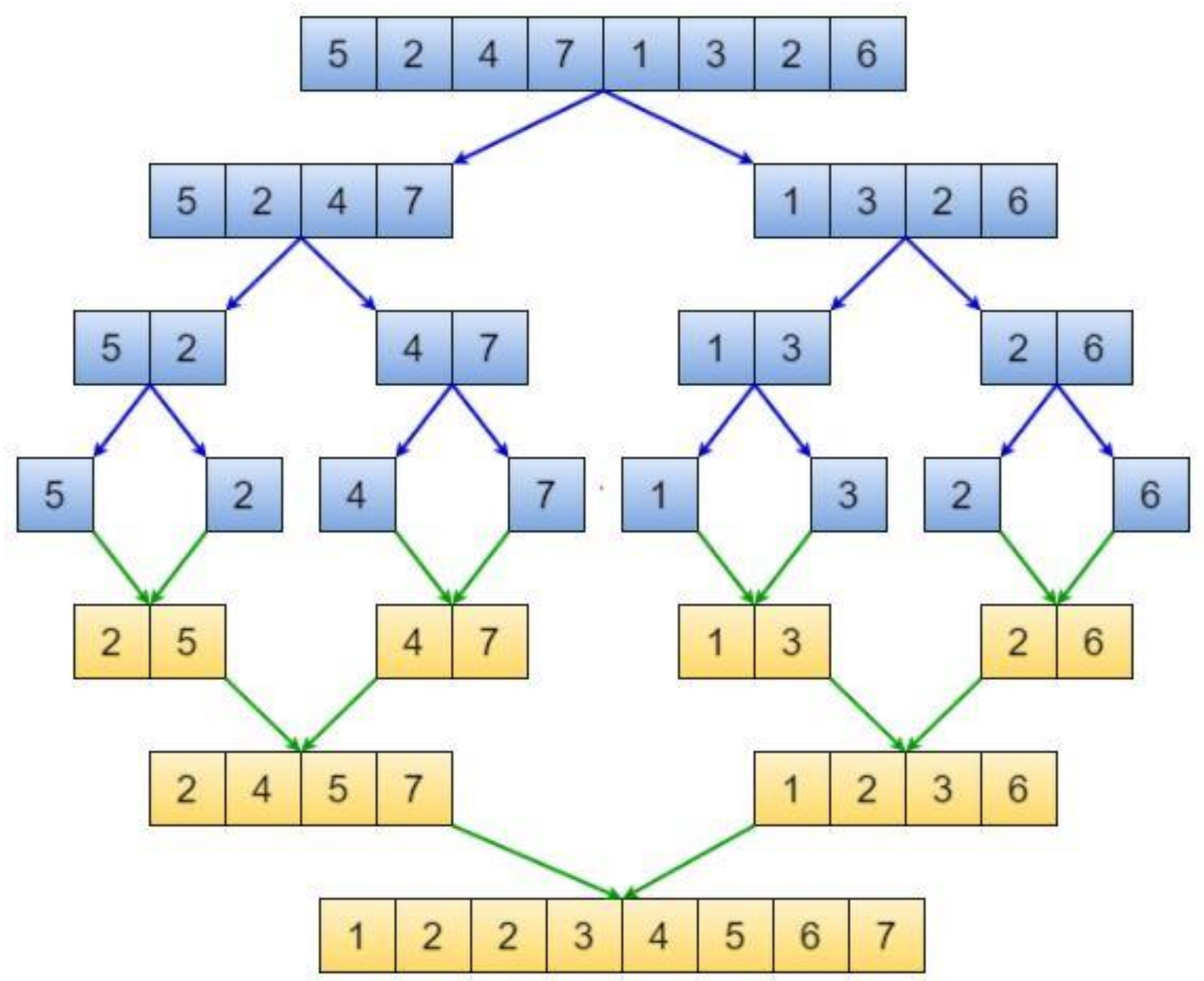
Example3:

Let's consider an array with values 7, 3, 2, 16, 24, 4, 11, 9. Below, we have a pictorial representation of how merge sort will sort the given array.



Example4:

Let's consider an array with values 5, 2, 4, 7, 1, 3, 2, 6. Below, we have a pictorial representation of how merge sort will sort the given array.



//Sort elements of an array in ascending order using merge sort algorithm

```
#include <stdio.h>
#include<conio.h>
#define max 10
int a[11] = { 10, 14, 19, 26, 27, 31, 33, 35, 42, 44, 0 };
int b[10];
void merging(int low, int mid, int high)
{
  int l1, l2, i;
  for(l1 = low, l2 = mid + 1, i = low; l1 <= mid && l2 <=
high; i++)
  {
    if(a[l1] <= a[l2])
                b[i] = a[l1++];
    else
                b[i] = a[l2++];
  }

  while(l1 <= mid)
    b[i++] = a[l1++];
  while(l2 <= high)
    b[i++] = a[l2++];
  for(i = low; i <= high; i++)
    a[i] = b[i];
}
void sort(int low, int high)
{
  int mid;
  if(low < high)
  {
    mid = (low + high) / 2;
    sort(low, mid);
    sort(mid+1, high);
    merging(low, mid, high);
  }
   else
  {
    return;
  }
}
int main()
{
  int i;
  clrscr();
  printf("List before sorting\n");
  for(i = 0; i <= max; i++)
    printf("%d ", a[i]);
  sort(0, max);
    printf("\nList after sorting\n");
  for(i = 0; i <= max; i++)
    printf("%d ", a[i]);
getch();
}
```

5. Quick Sort Algorithm

- Quick Sort Sorting Technique is based on the concept of **Divide and Conquer**, just like merge sort.

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

a. Elements less than the **Pivot** element

b. Pivot element(Central element)

c. 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.

How Quick Sorting Works?

- Following are the steps involved in quick sort algorithm:

1. After selecting an element as **pivot,** we have to divide the array.
2. In quick sort, we call this **partitioning**. It is not simple breaking down of array into 2 sub-arrays, 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 sub-arrays, 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 sub-arrays.

Quick Sort Algorithm

Step 1 – Choose the highest index value as 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

Algorithm:-

(i) take the first element of list as pivot.

(ii) Place pivot at ht proper place in list.

For placing pivot at it's proper place it follows following steps

→ Compare the pivot element one by one from right to left for getting the element which has value less than pivot element.

→ Interchange the element with element.

→ Now, comparison will start from the interchange element position from left to right for getting the element which has higher value than pivot.

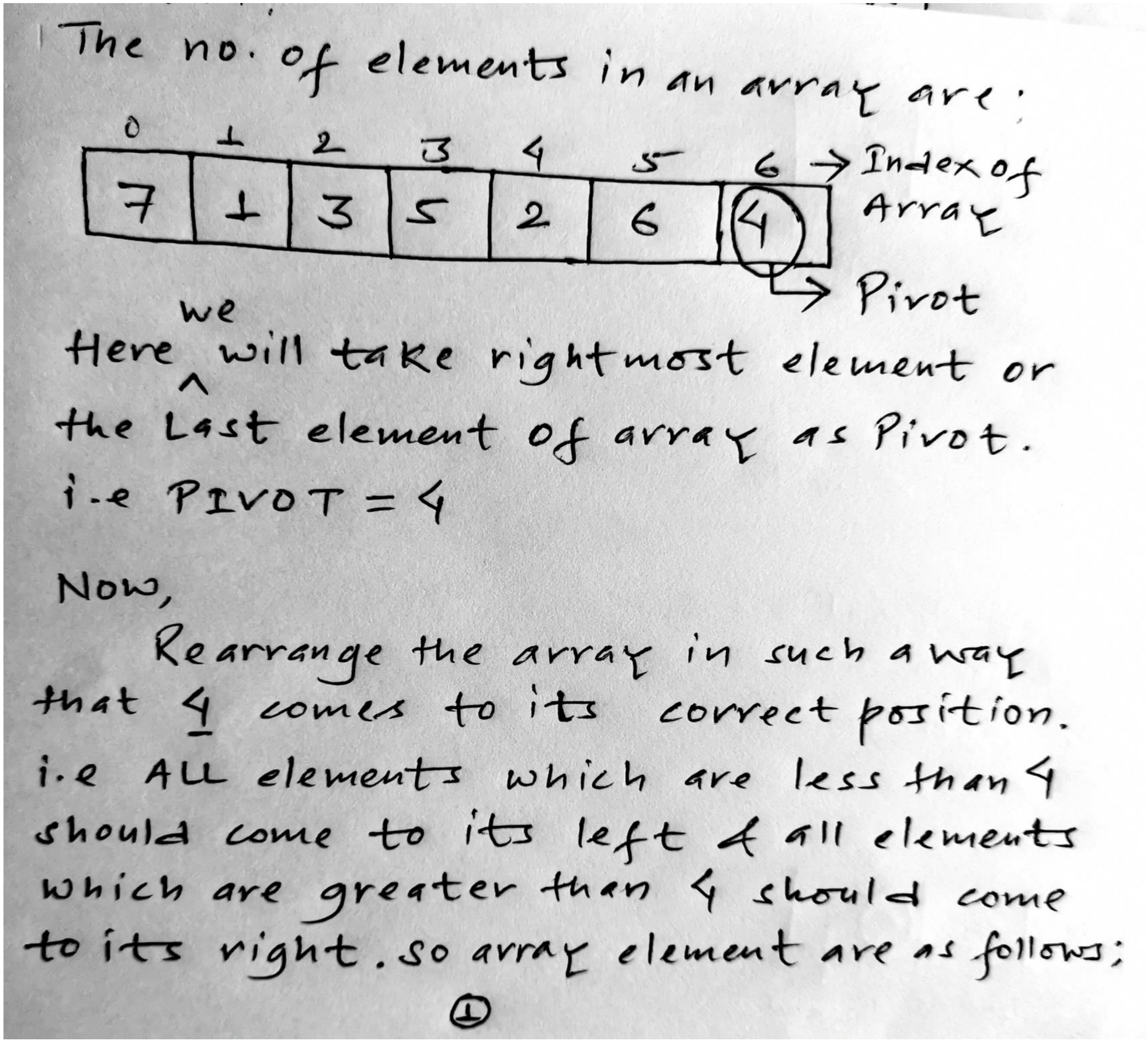
→ Repeat the same process until pivot is at it's proper position.

(iii) Create two subsists left & right side of pivot.

(iv) Repeat the same process until all elements of first are at proper position in list.

Q1. The elements in the array as follows: 7, 1, 3, 5, 2, 6, 4 and sort these elements using quick sort method.

Solution: METHOD1:



The no. of elements in an array are:

0	1	2	3	4	5	6	→ Index of Array
7	1	3	5	2	6	4	

→ Pivot

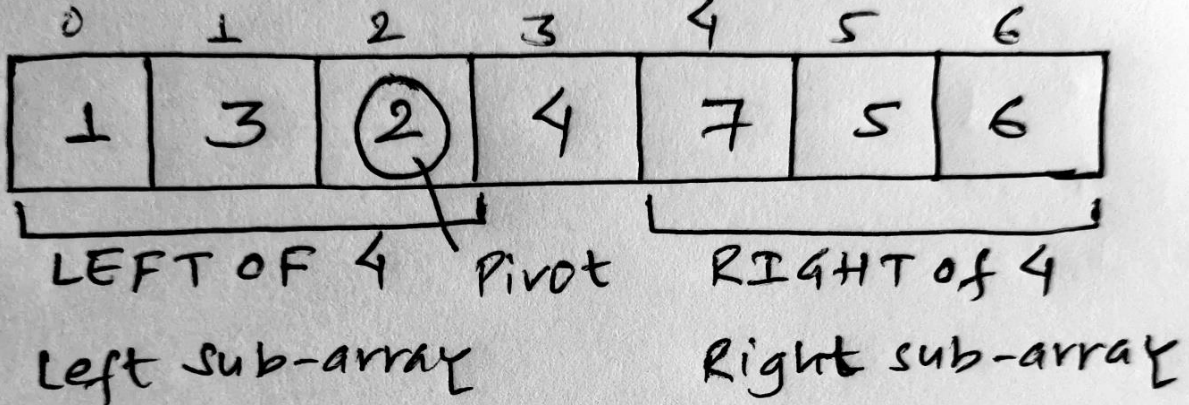
Here we will take rightmost element or the Last element of array as Pivot.

i.e PIVOT = 4

Now,

Rearrange the array in such a way that 4 comes to its correct position. i.e ALL elements which are less than 4 should come to its left & all elements which are greater than 4 should come to its right. so array element are as follows;

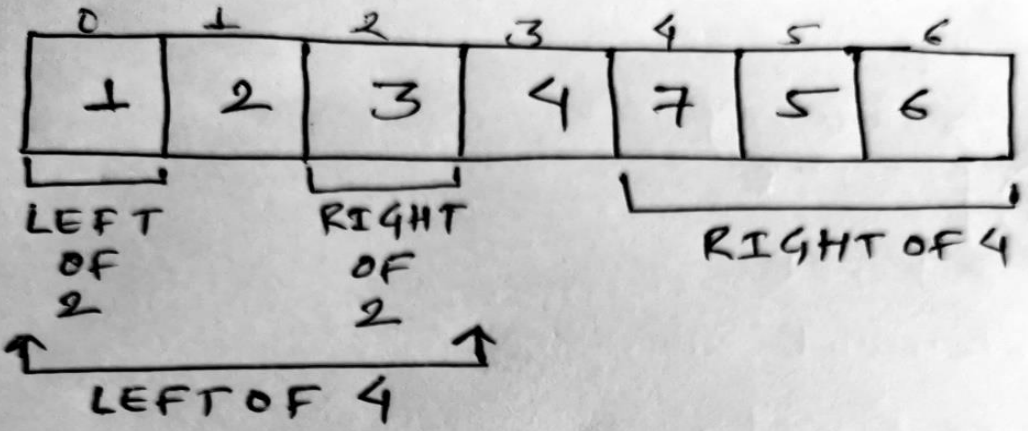
①



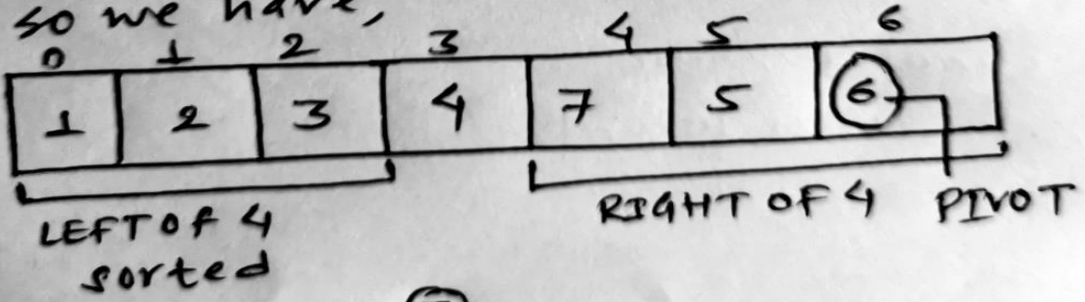
Here we have choosen the pivot and put the pivot in correct position this process is known as Partition.

Now, Again Apply the partition to the left of 4. when Applying partition we will choose 2 as pivot. since 2 is the last element in left of 4.

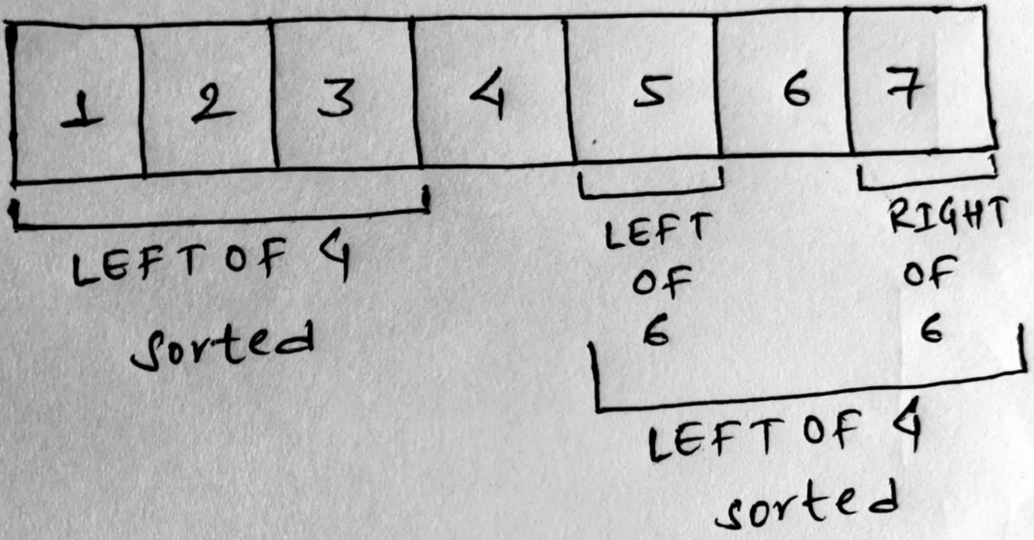
②



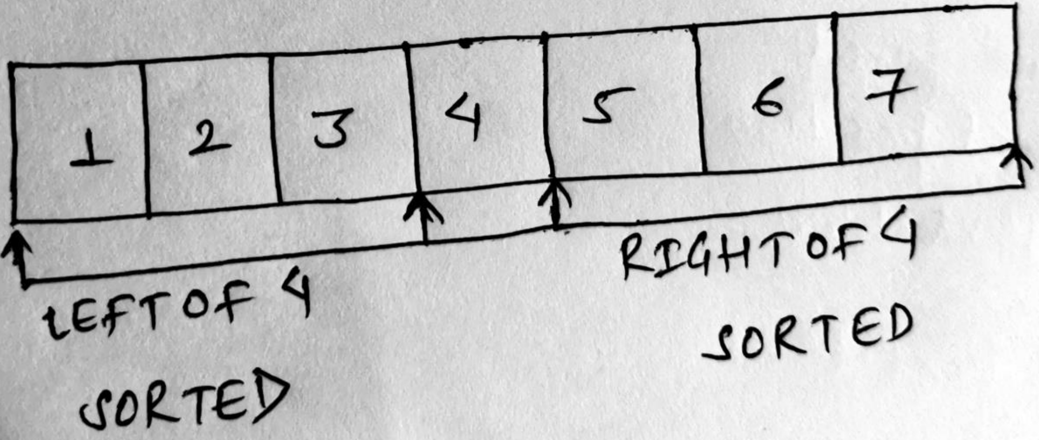
As we know that 2 has come to its correct position.
Now here, single array element 1 & 3 is already in sorted form. Now Again Apply partition algorithm to RIGHT OF 4.
So we have,



③

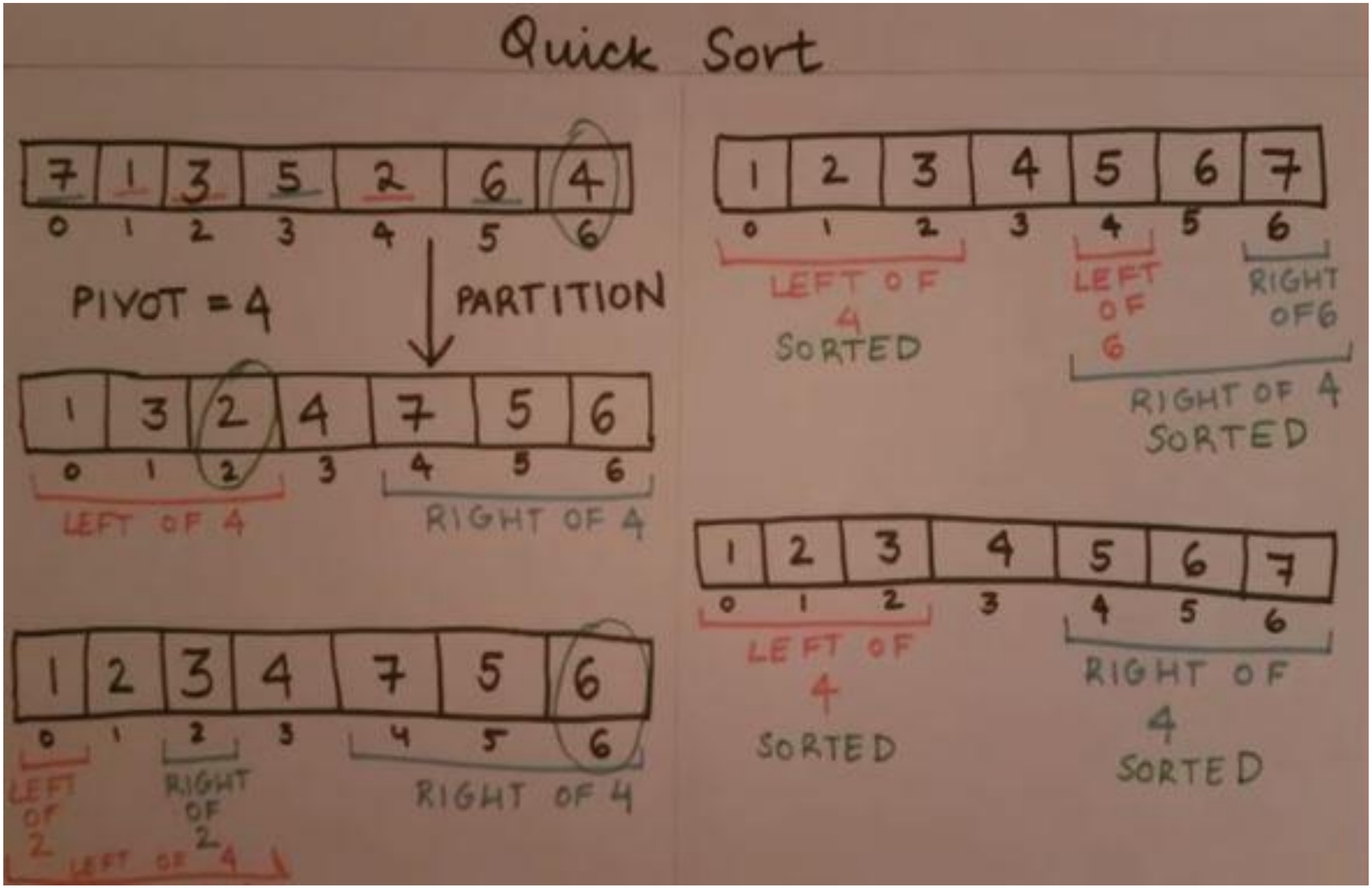


Finally we have :



4

You can do like this if you don't like to add the text.



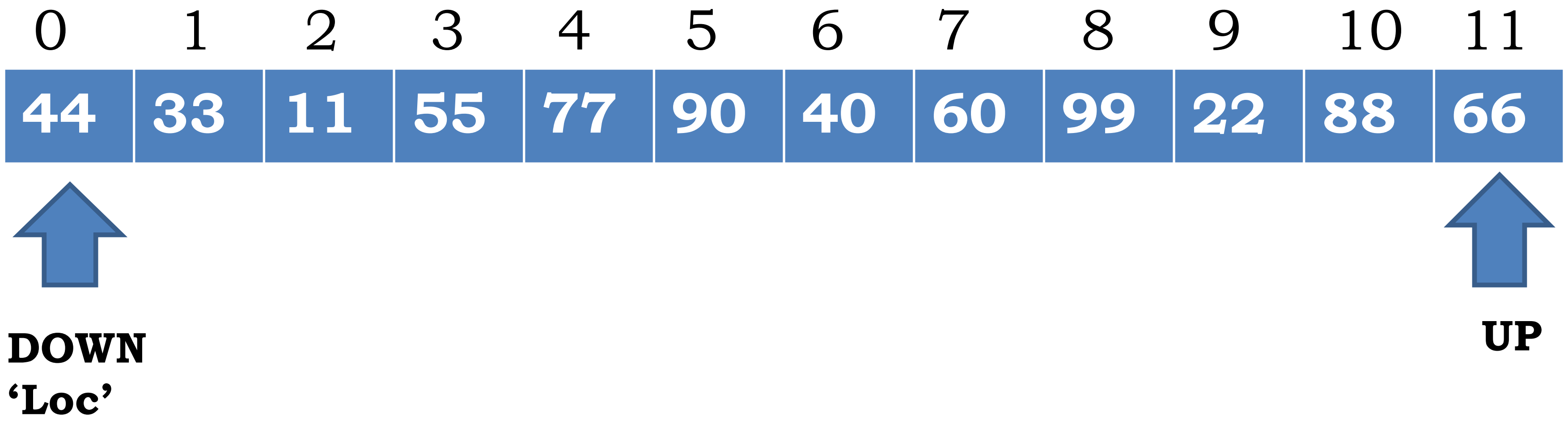
Q2. The elements in the array as follows: 44,33,11,55,77,90,40,60,99,22,88,66 and sort these elements using quick sort method.

Answer: Method 2

Step1:

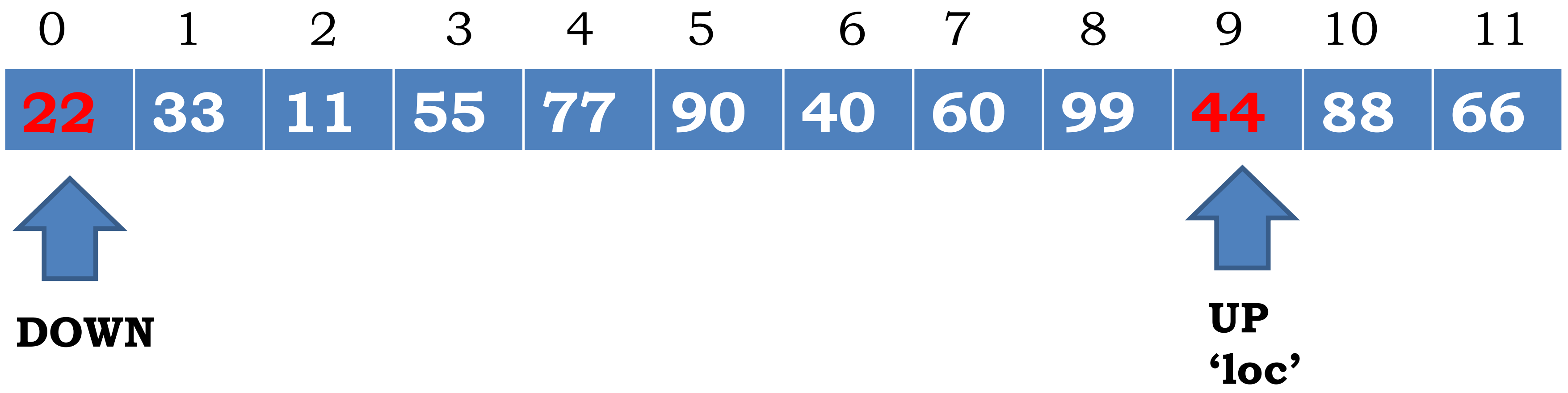
Mark first element as “DOWN” and last element as “UP” and we have ‘loc’ variable or PIVOT which is initially at down as shown in the array below:

0	1	2	3	4	5	6	7	8	9	10	11
44	33	11	55	77	90	40	60	99	22	88	66



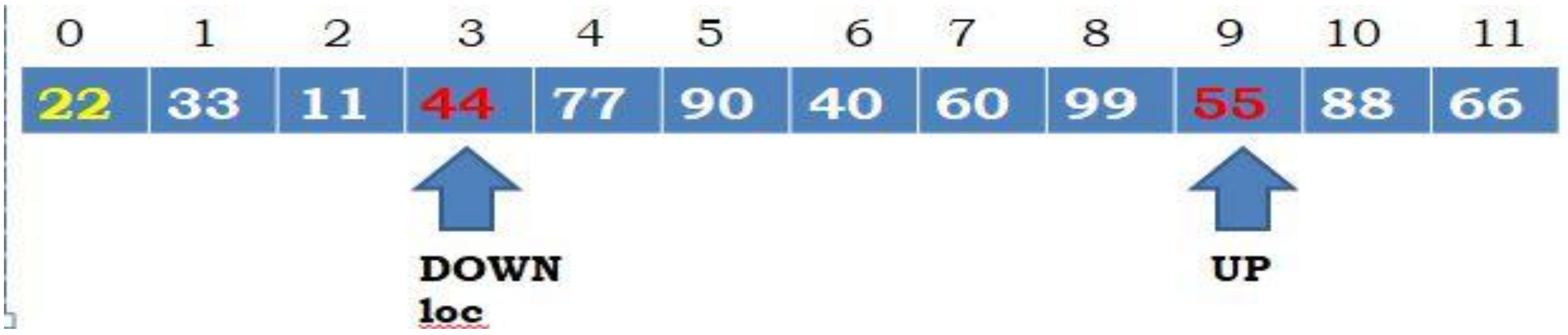
Step2:

Now scan the elements of array from **right to left** till the number less than first is found. After scanning we found 22 is less than 44 so interchange 22 and 44 as shown in the array below:



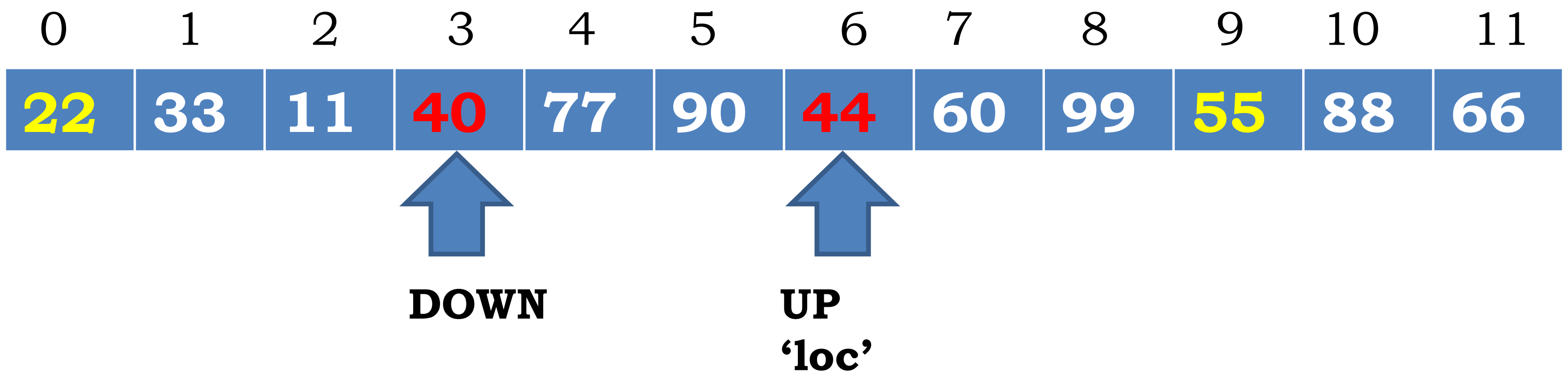
Step3:

Now again scan the elements of array from **left to right** till the number greater than 44 is found. After scanning we found 55 is greater than 44 so interchange 55 and 44 as shown in the array below:



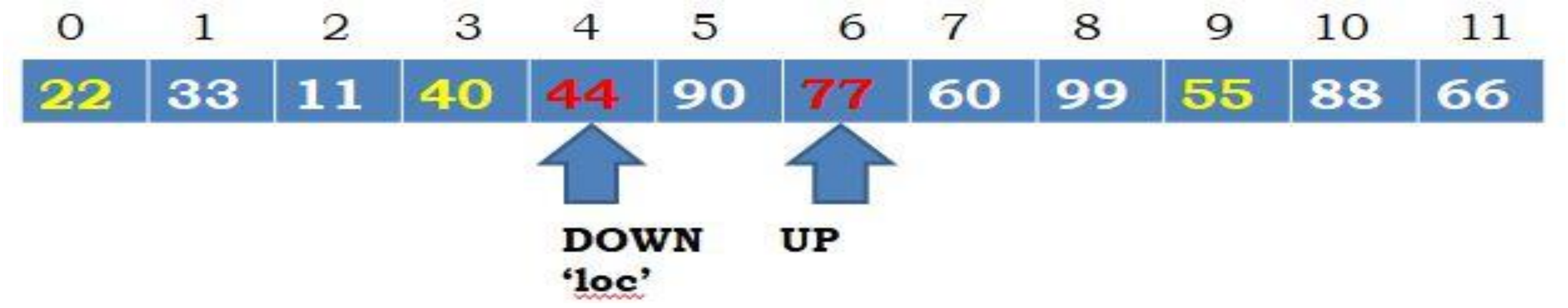
Step4:

Now again scan the elements of array from **right to left** till the number less then 44 is found. After scanning we found 40 is less then 44 so interchange 40 and 44 as shown in the array below:



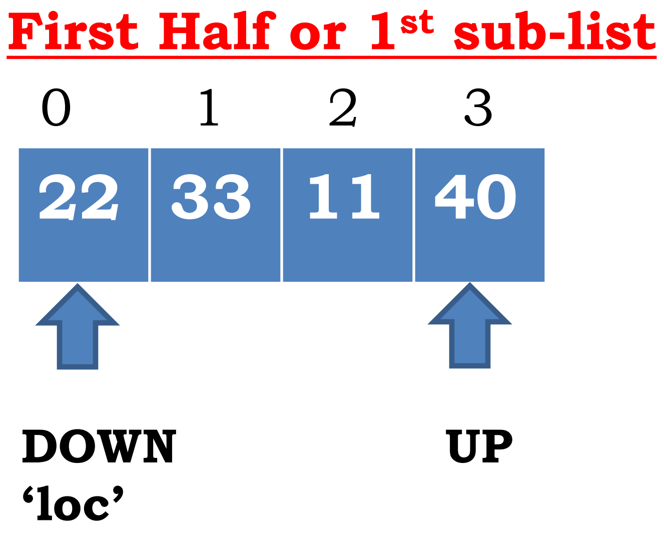
Step5:

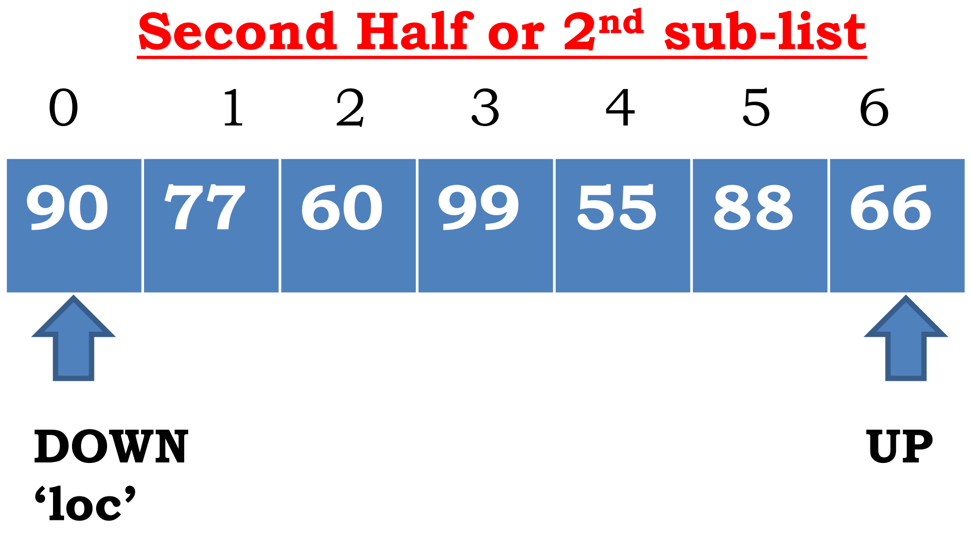
Now again scan the elements of array from **left to right** till the number greater than 44 is found. After scanning we found 77 is greater than 44 so interchange 77 and 44 as shown in the array below:



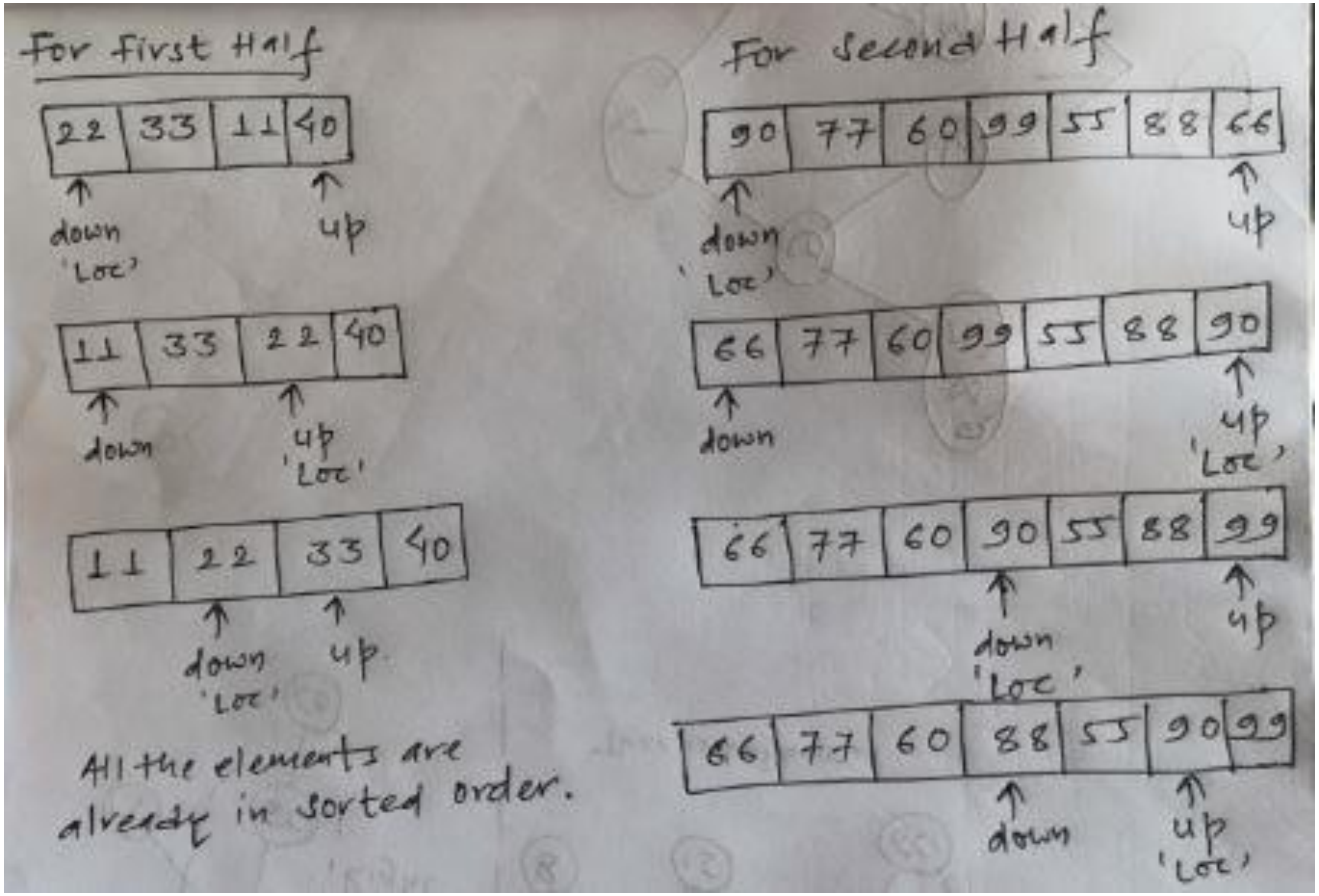
Step6:

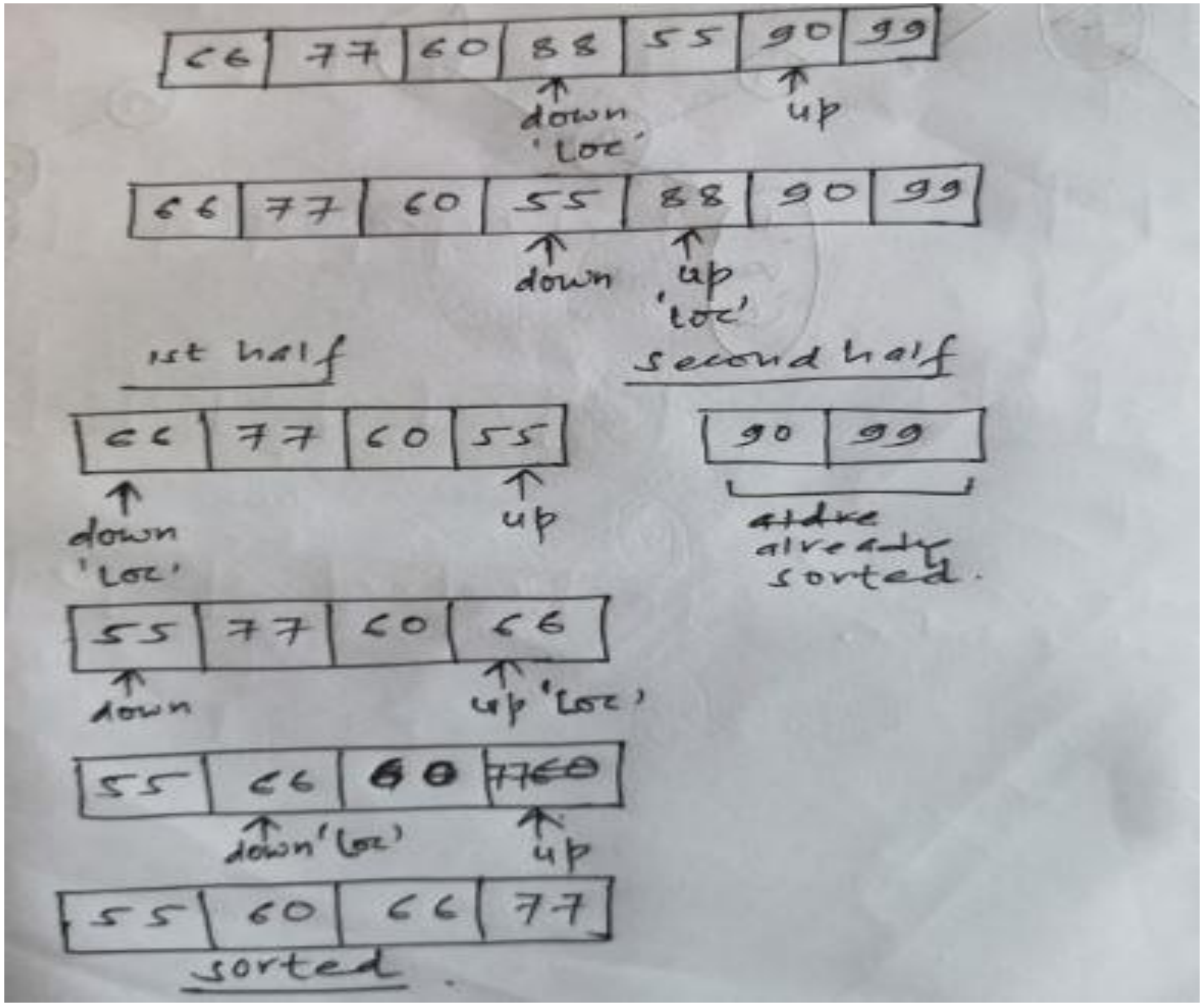
Again scan the elements of array from **right to left** till the number less then 44 is found. After scanning there is no such number which is less than 44, so by the rule of quick sort algorithm divide the array in **2 half or sub-list** so that we get elements less than element at PIVOT or loc variable comes in left sub-list and element greater than elements at PIVOT or loc variable comes in right sub list. Therefore the elements in two sub-list are as follows:





Similarly do the same process for both first sub list and second sub list.


For first Half
22 | 33 | 11 | 40
down 'Loc'
up
11 | 33 | 22 | 40
down
up 'Loc'
11 | 22 | 33 | 40
down 'Loc'
up
All the elements are already in sorted order.
For second Half
90 | 77 | 60 | 99 | 55 | 88 | 66
down 'Loc'
up
66 | 77 | 60 | 99 | 55 | 88 | 90
down
up 'Loc'
66 | 77 | 60 | 90 | 55 | 88 | 99
down 'Loc'
up
66 | 77 | 60 | 88 | 55 | 90 | 99
down
up 'Loc'



- Hence the sorted data in the array as follows:

0	1	2	3	4	5	6	7	8	9	10	11
11	22	33	40	44	55	60	66	77	88	90	99

//sort elements of an array in ascending order using quick sort algorithm

```
#include<stdio.h>
#include<conio.h>
void quicksort(int number[25], int first, int last)
{
int i, j, pivot, temp;
if(first<last)
{
pivot=first;
i=first;
j=last;
while(i<j)
{
while(number[i]<=number[pivot]&&i<last)
i++;
while(number[j]>number[pivot])
j--;
if(i<j)
{
temp=number[i];
number[i]=number[j];
number[j]=temp;
}
}
temp=number[pivot];
number[pivot]=number[j];
number[j]=temp;
quicksort(number,first,j-1);
quicksort(number,j+1,last);

}
}

void main()
{
int i, count, number[25];
clrscr();
printf("How many elements are you going to enter?: ");
scanf("%d",&count);
printf("Enter %d elements:\n", count);
for(i=0;i<count;i++)
scanf("%d",&number[i]);
quicksort(number,0,count-1);
printf("Order of Sorted elements are:\n");
for(i=0;i<count;i++)
printf("%d\n",number[i]);
getch();
}
```

Heap Sort:-

A heap is a binary tree that satisfy the following properties:-

- shape property
- Order property.

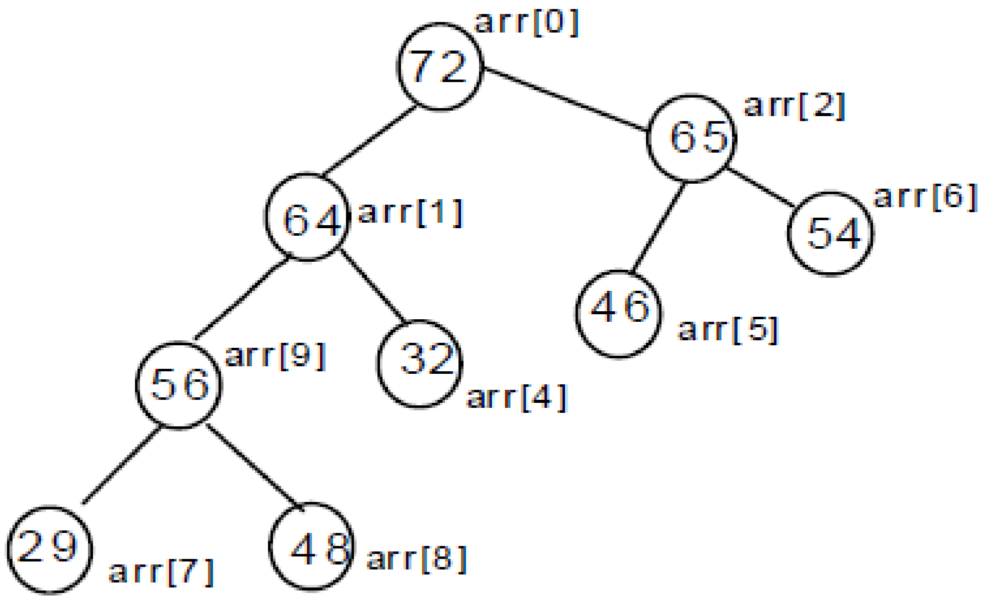
By the shape property we mean that heap must be a complete binary tree where as by order property we mean that for every node in the heap the value store in the heap node is greater than or equal to the value to the value an each of its' children. A heap that satisfied this property is known as max heap.

However, If the order property is such that for every node. In the heap the value stored in that node is less than or equal to the value in each of it's children. That heap is known as minimum heap.

The elements of the heap tree are represented by an array . The root is the largest element of heap tree. As it is maintained in the array. The largest element of heap tree. As it is maintained in the array. The largest value should be the last element of array. For heap sorting root is deleted till there is only one element in the tree.

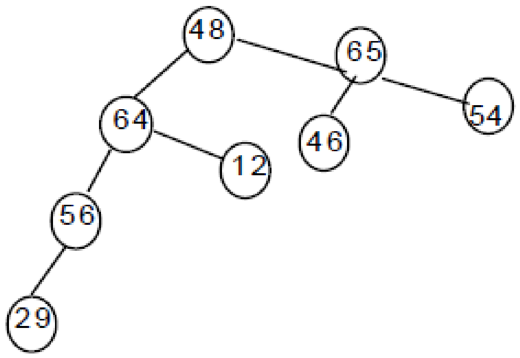
Steps:-

- Replace the root with the last node of heap tree.
- Keep the last node at proper position.
- Repeat step 1 & 2 until there are only one root node in the tree.



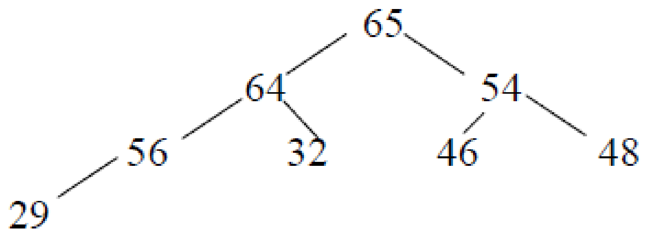
	0	1	2	3	4	5	6	7	8
arr	72	64	65	56	32	46	54	29	48

Step:-1 Root node is delete and this root node is replaced by last node and the previous value of root node is placed in proper place in array.

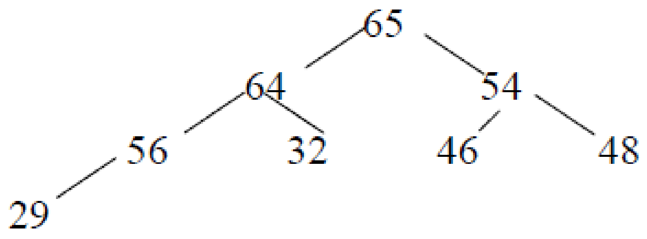


48	64	65	56	32	54	29	72

Here, root node is less than 65. so we interchange the position to make heap tree.

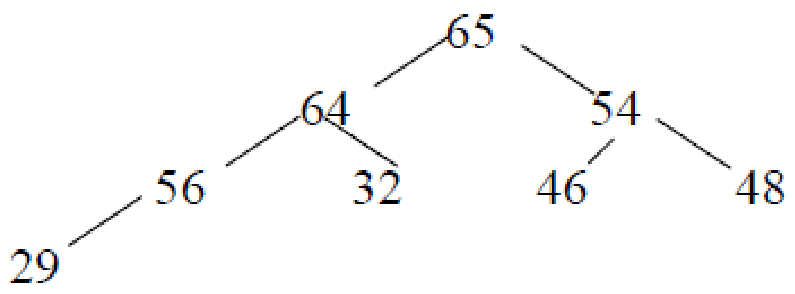


Again , 48< 54



Now, which is in Heap tree form.

Again , 48< 54



Now, which is in Heap tree form.

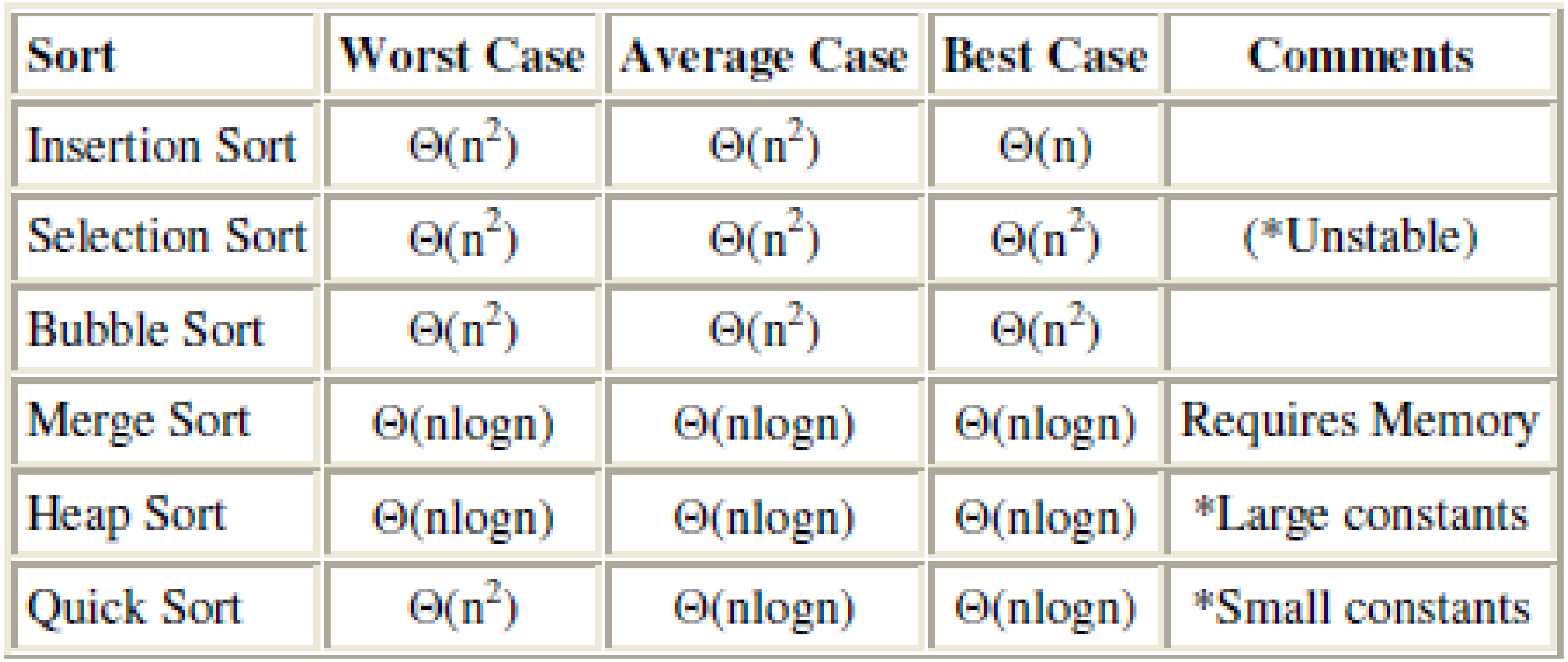
65	64	54	56	32	46	48	29	~~72~~

Again , delete root node & put 29 in root node.

29	64	54	56	32	32	46	48	~~65~~	~~72~~

29	64	54	56	32	32	46	48	~~65~~	~~72~~

65	64	54	56	32	46	48	29	~~72~~



Sort	Worst Case	Average Case	Best Case	Comments
Insertion Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	
Selection Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	(*Unstable)
Bubble Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	
Merge Sort	$\Theta(n\log n)$	$\Theta(n\log n)$	$\Theta(n\log n)$	Requires Memory
Heap Sort	$\Theta(n\log n)$	$\Theta(n\log n)$	$\Theta(n\log n)$	*Large constants
Quick Sort	$\Theta(n^2)$	$\Theta(n\log n)$	$\Theta(n\log n)$	*Small constants

Questions:

1. What is sorting? Explain with an example.
2. Explain Selection Sort algorithm with an example.
3. Explain Bubble Sort algorithm with an example.
4. Explain Insertion Sort algorithm with an example.
5. Explain Merge Sort algorithm with an example.
6. Explain Quick Sort algorithm with an example.
7. Sort the following number using selection sort: 20, 35, 40, 100, 3, 10, 15
8. Sort the following number using bubble sort: 5, 1, 6, 2, 4, 3
9. Sort the following number using insertion sort: 23, 78, 45, 8, 32, 56
10. Sort the following number using merge sort: 7, 3, 2, 16, 24, 4, 11, 9
11. The elements in the array as follows: 7, 1, 3, 5, 2, 6, 4 and sort these elements using quick sort method.
12. The elements in the array as follows: 48, 29 ,8 ,59 ,72 ,88, 42, 65 ,95 ,19 ,82, 68 and sort these elements using quick sort method.

Note:

Space Complexity of Algorithms

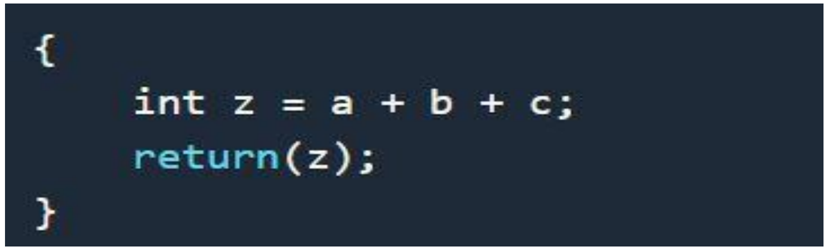
- Whenever a solution to a problem is written, some memory is required to complete. For any algorithm memory may be used for the following:

1. Variables (This include the constant values, temporary values)
2. Program Instruction
3. Execution

- ***Space complexity*** *is the amount of memory used by the algorithm (including the input values to the algorithm) to execute and produce the result.*

Calculating the Space Complexity

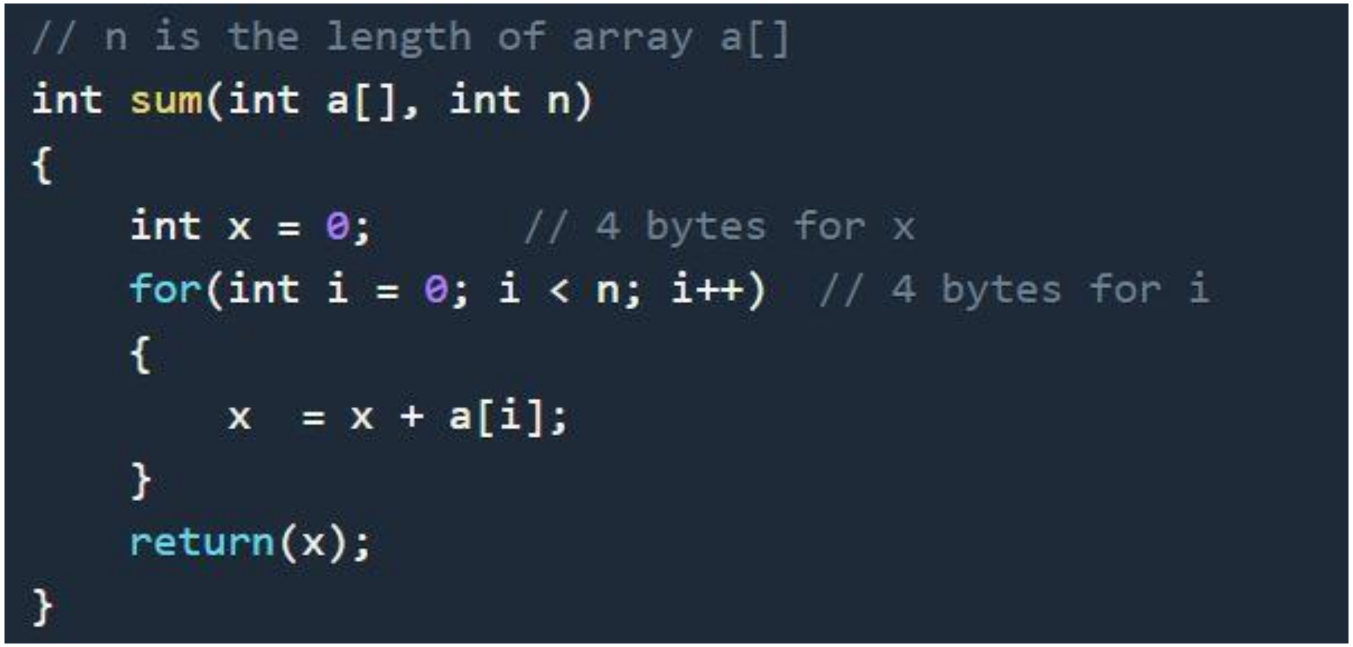
- For calculating the space complexity, we need to know the value of memory used by different type of data type variables, which generally varies for different operating systems, but the method for calculating the space complexity remains the same.
- Now let's learn how to compute space complexity by taking a few examples:



```
{
    int z = a + b + c;
    return(z);
}
```

- In the above expression, variables a, b, c and z are all integer types, hence they will take up 4 bytes each, so total memory requirement will be (4*4 + 4) = 20 bytes, this additional 4 bytes is for **return value**. And because this space requirement is fixed for the above example, hence it is called **Constant Space Complexity**.

Let's have another example, a bit complex one,

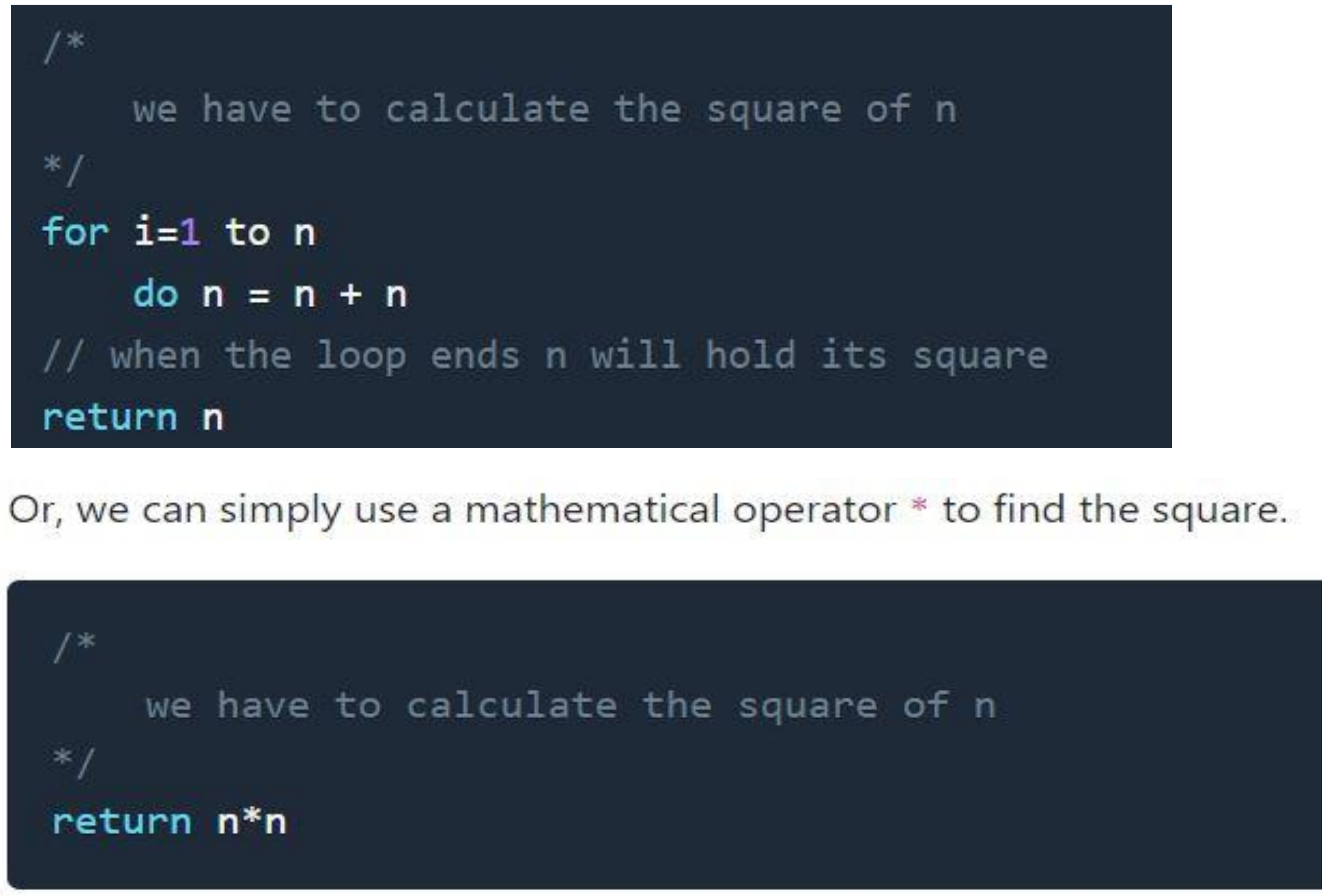


```
// n is the length of array a[]
int sum(int a[], int n)
{
    int x = 0;        // 4 bytes for x
    for(int i = 0; i < n; i++)  // 4 bytes for i
    {
        x  = x + a[i];
    }
    return(x);
}
```

- In the above code, 4*n bytes of space is required for the array a[ ] elements.
- 4 bytes each for x, n, i and the return value.
- Hence the total memory requirement will be (4n + 12), which is increasing linearly with the **increase in the input value n**, hence it is called as **Linear Space Complexity.**

Time Complexity of Algorithms

- For any defined problem, there can be N number of solution. If I have a problem and I discuss about the problem with all of my friends, they will all suggest me different solutions. And I am the one who has to decide which solution is the best based on the circumstances.
- Similarly for any problem which must be solved using a program, there can be infinite number of solutions. Let's take a simple example to understand this. Below we have two different algorithms to find square of a number(for some time, forget that square of any number n is n*n)
- One solution to this problem can be, running a loop for n times, starting with the number n and adding n to it, every time.



```
/*
    we have to calculate the square of n
*/
for i=1 to n
    do n = n + n
// when the loop ends n will hold its square
return n
```

Or, we can simply use a mathematical operator * to find the square.

```
/*
    we have to calculate the square of n
*/
return n*n
```

In the above two simple algorithms, you saw how a single problem can have many solutions. While the first solution required a loop which will execute for n number of times, the second solution used a mathematical operator * to return the result in one line. So which one is the better approach, of course the second one.

What is Time Complexity?

- Time complexity of an algorithm signifies the total time required by the program to run till its completion.
- The time complexity of algorithms is most commonly expressed using the **big O notation**.
- Time Complexity is most commonly estimated by counting the number of elementary steps performed by any algorithm to finish execution.
- Like in the example above, for the first code the loop will run n number of times, so the time complexity will be n at least and as the value of n will increase the time taken will also increase.
- While for the second code, time complexity is constant, because it will never be dependent on the value of n, it will always give the result in 1 step.

Calculating Time Complexity

- The most common metric for calculating time complexity is Big O notation. This removes all constant factors so that the running time can be estimated in relation to **N**, as N approaches infinity. In general you can think of it like this :

```
statement;
```

- Above we have a single statement. Its Time Complexity will be **Constant**. The running time of the statement will not change in relation to N.

```
for(i=0; i < N; i++)
{
    statement;
}
```

The time complexity for the above algorithm will be **Linear**. The running time of the loop is directly proportional to N. When N doubles, so does the running time.

```
for(i=0; i < N; i++)
{
    for(j=0; j < N;j++)
    {
    statement;
    }
}
```

This time, the time complexity for the above code will be **Quadratic**. The running time of the two loops is proportional to the square of N. When N doubles, the running time increases by N * N.