Introduction to Searching Algorithms

Not even a single day pass, when we do not have to search for something in our day to day life, car keys, books, pen, mobile charger and what not. Same is the life of a computer, there is so much data stored in it, that whenever a user asks for some data, computer has to search it's memory to look for the data and make it available to the user. And the computer has it's own techniques to search through it's memory fast, which you can learn more about in our Operating System tutorial series.

What if you have to write a program to search a given number in an array? How will you do it?

Well, to search an element in a given array, there are two popular algorithms available:

1. Linear Search
2. Binary Search

Linear Search

Linear search is a very basic and simple search algorithm. In Linear search, we search an element or value in a given array by traversing the array from the starting, till the desired element or value is found.

It compares the element to be searched with all the elements present in the array and when the element is **matched** successfully, it returns the index of the element in the array, else it return `-1`.

Linear Search is applied on unsorted or unordered lists, when there are fewer elements in a list.

Features of Linear Search Algorithm

1. It is used for unsorted and unordered small list of elements.
2. It has a time complexity of **O(n)**, which means the time is linearly dependent on the number of elements, which is not bad, but not that good too.
3. It has a very simple implementation.

We will implement the Linear Search algorithm in the next tutorial.

Binary Search

Binary Search is used with sorted array or list. In binary search, we follow the following steps:

1. We start by comparing the element to be searched with the element in the middle of the list/array.
2. If we get a match, we return the index of the middle element.
3. If we do not get a match, we check whether the element to be searched is less or greater than in value than the middle element.
4. If the element/number to be searched is greater in value than the middle number, then we pick the elements on the right side of the middle element(as the list/array is sorted, hence

on the right, we will have all the numbers greater than the middle number), and start again from the step 1.

5. If the element/number to be searched is lesser in value than the middle number, then we pick the elements on the left side of the middle element, and start again from the step 1.

Binary Search is useful when there are large number of elements in an array and they are sorted.

So a necessary condition for Binary search to work is that the list/array should be sorted.

Features of Binary Search

1. It is great to search through large sorted arrays.
2. It has a time complexity of **O(log n)** which is a very good time complexity. We will discuss this in details in the Binary Search tutorial.
3. It has a simple implementation.

Linear Search Algorithm

Linear search is a very basic and simple search algorithm. In Linear search, we search an element or value in a given array by traversing the array from the starting, till the desired element or value is found.

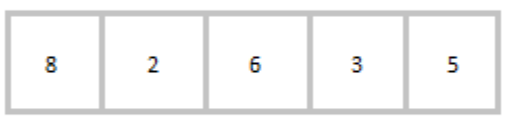
As we learned in the previous tutorial that the time complexity of Linear search algorithm is **O(n)**, we will analyse the same and see why it is **O(n)** after implementing it.

Implementing Linear Search

Following are the steps of implementation that we will be following:

1. Traverse the array using a `for` loop.
2. In every iteration, compare the `target` value with the current value of the array.
    - If the values match, return the current index of the array.
    - If the values do not match, move on to the next array element.
3. If no match is found, return `-1`.

To search the number **5** in the array given below, linear search will go step by step in a sequential order starting from the first element in the given array.




```
/*
    below we have implemented a simple function
    for linear search in C


    - values[] => array with all the values
    - target => value to be found
    - n => total number of elements in the array
*/
int linearSearch(int values[], int target, int n)
{
    for(int i = 0; i < n; i++)
    {
```

```
        if (values[i] == target)
        {
            return i;
        }
    }
    return -1;
}
```

Copy

Some Examples with Inputs

```
Input: values[] = {5, 34, 65, 12, 77, 35}
       target = 77

Output: 4

Input: values[] = {101, 392, 1, 54, 32, 22, 90, 93}
       target = 200

Output: -1 (not found)
```

Final Thoughts

We know you like Linear search because it is so damn simple to implement, but it is not used practically because binary search is a lot faster than linear search. So let's head to the next tutorial where we will learn more about binary search.

Binary Search Algorithm

Binary Search is applied on the sorted array or list of large size. It's time complexity of **O(log n)** makes it very fast as compared to other sorting algorithms. The only limitation is that the array or list of elements must be sorted for the binary search algorithm to work on it.

Implementing Binary Search Algorithm

Following are the steps of implementation that we will be following:

1. Start with the middle element:
    - If the **target** value is equal to the middle element of the array, then return the index of the middle element.
    - If not, then compare the middle element with the target value,
        - If the target value is greater than the number in the middle index, then pick the elements to the right of the middle index, and start with Step 1.
        - If the target value is less than the number in the middle index, then pick the elements to the left of the middle index, and start with Step 1.
2. When a match is found, return the index of the element matched.
3. If no match is found, then return `-1`

```
/*
    function for carrying out binary search on given array
    - values[] => given sorted array
    - len => length of the array
    - target => value to be searched
*/
int binarySearch(int values[], int len, int target)
{
    int max = (len - 1);
    int min = 0;

    int guess;  // this will hold the index of middle elements
    int step = 0;  // to find out in how many steps we completed the search

    while(max >= min)
```

```
    {
        guess = (max + min) / 2;
        // we made the first guess, incrementing step by 1
        step++;

        if(values[guess] ==  target)
        {
            printf("Number of steps required for search: %d \n", step);
            return guess;
        }
        else if(values[guess] >  target)
        {
            // target would be in the left half
            max = (guess - 1);
        }
        else
        {
            // target would be in the right half
            min = (guess + 1);
        }
    }

    // We reach here when element is not
    // present in array
    return -1;
}

int main(void)
{
    int values[] = {13, 21, 54, 81, 90};
```

```
    int n = sizeof(values) / sizeof(values[0]);

    int target = 81;

    int result = binarySearch(values, n, target);

    if(result == -1)

    {

        printf("Element is not present in the given array.");

    }

    else

    {

        printf("Element is present at index: %d", result);

    }

    return 0;

}
```

Copy

We hope the above code is clear, if you have any confusion, post your question in our Q & A Forum.

Now let's try to understand, why is the time complexity of binary search **O(log n)** and how can we calculate the number of steps required to search an element from a given array using binary search without doing any calculations. It's super easy! Are you ready?

Time Complexity of Binary Search $O(\log n)$

When we say the time complexity is $\log n$, we actually mean $\log_2 n$, although the **base** of the log doesn't matter in asymptotic notations, but still to understand this better, we generally consider a base of 2.

Let's first understand what $\log_2(n)$ means.

```
Expression: log₂(n)

- - - - - - - - - - - - - - -

For n = 2:

log₂(2¹) = 1

Output = 1

- - - - - - - - - - - - - - -

For n = 4

log₂(2²) = 2

Output = 2

- - - - - - - - - - - - - - -

For n = 8
```

```
log₂(2³) = 3

Output = 3

- - - - - - - - - - - - - - -

For n = 256

log₂(2⁸) = 8

Output = 8

- - - - - - - - - - - - - - -

For n = 2048

log₂(2¹¹) = 11

Output = 11
```

Now that we know how $\log_2(n)$ works with different values of n, it will be easier for us to relate it with the time complexity of the binary search algorithm and also to understand how we can find out the number of steps required to search any number using binary search for any value of n.

Counting the Number of Steps

As we have already seen, that with every incorrect guess, binary search cuts down the list of elements into half. So if we start with 32 elements, after first unsuccessful guess, we will be left with 16 elements.

So consider an array with 8 elements, after the first unsuccessful, binary search will cut down the list to half, leaving behind 4 elements, then 2 elements after the second unsuccessful guess, and finally only 1 element will be left, which will either be the target or not, checking that will involve one more step. So all in all binary search needed at most 4 guesses to search the target in an array with 8 elements.

If the size of the list would have been 16, then after the first unsuccessful guess, we would have been left with 8 elements. And after that, as we know, we need atmost 4 guesses, add 1 guess to cut down the list from 16 to 8, that brings us to 5 guesses.

So we can say, as the number of elements are getting doubled, the number of guesses required to find the target increments by 1.

Seeing the pattern, right?

Generalizing this, we can say, for an array with n elements,

the number of times we can repeatedly halve, starting at n, until we get the value 1, plus one.

And guess what, in mathematics, the function $\log_2 n$ means exactly same. We have already seen how the **log** function works above, did you notice something there?

For $n = 8$, the output of $\log_2 n$ comes out to be 3, which means the array can be halved 3 times maximum, hence the number of steps(at most) to find the target value will be (3 + 1) = 4.

Question for you: What will be the maximum number of guesses required by Binary Search, to search a number in a list of **2,097,152** elements?

Now that we have learned the Binary Search Algorithms, you can also learn other types of Searching Algorithms and their applications:

- Linear Search
- Jump Search

Jump Search Algorithm

Jump Search Algorithm is a relatively new algorithm for searching an element in a sorted array.

The fundamental idea behind this searching technique is to search fewer number of elements compared to linear search algorithm (which scans every element in the array to check if it matches with the element being searched or not). This can be done by skipping some fixed number of array elements or **jumping ahead by fixed number of steps** in every iteration.

Lets consider a sorted array $A[]$ of size n, with indexing ranging between **0** and **n-1**, and element **x** that needs to be searched in the array $A[]$. For implementing this algorithm, a block of size **m** is also required, that can be skipped or jumped in every iteration. Thus, the algorithm works as follows:

- **Iteration 1**: if ($x==A[0]$), then success, else, if ($x > A[0]$), then jump to the next block.
- **Iteration 2**: if ($x==A[m]$), then success, else, if ($x > A[m]$), then jump to the next block.
- **Iteration 3**: if ($x==A[2m]$), then success, else, if ($x > A[2m]$), then jump to the next block.
- At any point in time, if ($x < A[km]$), then a **linear search** is performed from index $A[(k-1)m]$ to $A[km]$

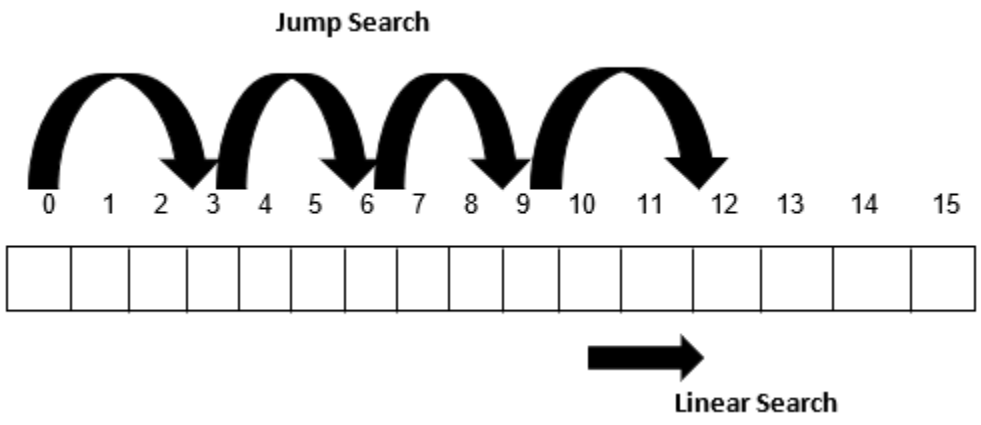


Figure 1: Jump Search technique

Optimal Size of m (Block size to be skipped)

The worst-case scenario requires:

- n/m jumps, and
- $(m-1)$ comparisons (in case of linear search if $x < A[km]$)

Hence, the total number of comparisons will be $(n/m+(m-1))$. This expression has to be minimum, so that we get the smallest value of m (block size).

On differentiating this expression with respect to m and equating it with **0**, we get:

```
n/-m2+1 = 0



n/m2 = 1



m = √n
```

Copy

Hence, the **optimal jump size** is **√n**, where **n** is the size of the array to be searched or the total number of elements to be searched.

Algorithm of Jump Search

Below we have the algorithm for implementing Jump search:

Input will be:

- Sorted array `A` of size `n`
- Element to be searched, say `item`

Output will be:

- A valid location of `item` in the array `A`

Steps for Jump Search Algorithms:

Step 1: Set `i=0` and `m = √n`.

Step 2: Compare `A[i]` with `item`. If `A[i] != item` and `A[i] < item`, then jump to the next block. Also, do the following:

1. Set **i = m**
2. Increment `m` by `√n`

Step 3: Repeat the step 2 till **m < n-1**

Step 4: If `A[i] > item`, then move to the beginning of the current block and perform a linear search.

1. Set **x = i**
2. Compare `A[x]` with `item`. If `A[x]== item`, then print `x` as the valid location else set `x++`
3. Repeat Step 4.1 and 4.2 till **x < m**

Step 5: Exit

Pictorial Representation of Jump Search with an Example

Let us trace the above algorithm using an example:

Consider the following inputs:

- `A[]` = {0, 1, 1, 2, 3, 5, 8, 13, 21, 55, 77, 89, 101, 201, 256, 780}
- `item` = 77

Step 1: **m = √n = 4** (Block Size)

Step 2: Compare `A[0]` with `item`. Since **A[0] != item** and **A[0]<item**, skip to the next block

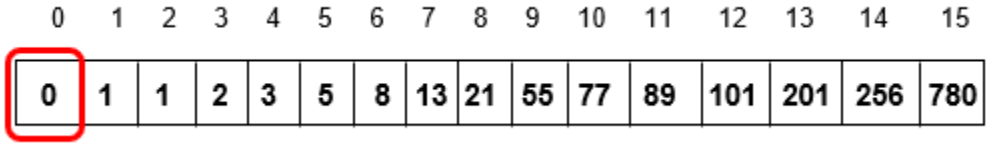


Figure 2: Comparing A[0] and item

Step 3: Compare A[3] with item. Since A[3] != itemand A[3]<item, skip to the next block

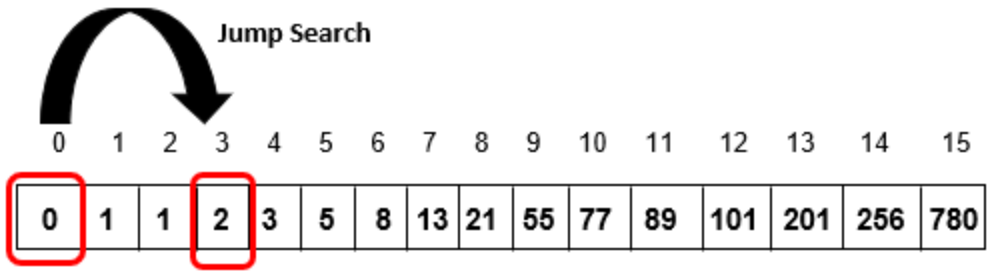


Figure 3: Comparing A[3] and item

Step 4: Compare A[6] with item. Since A[6] != itemand A[6]<item, skip to the next block

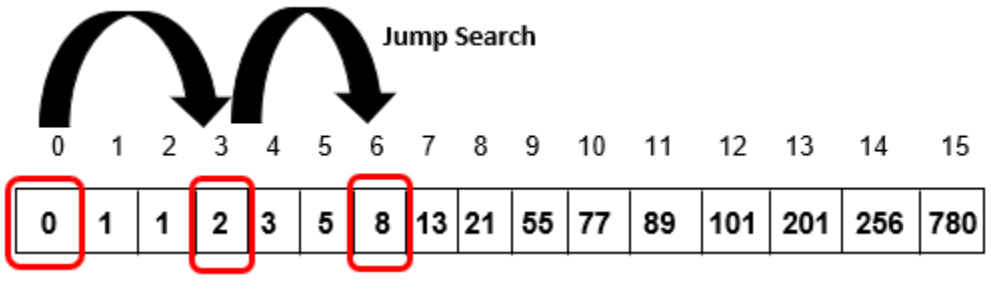


Figure 4: Comparing A[6] and item

Step 5: Compare A[9] with item. Since A[9] != itemand A[9]<item, skip to the next block

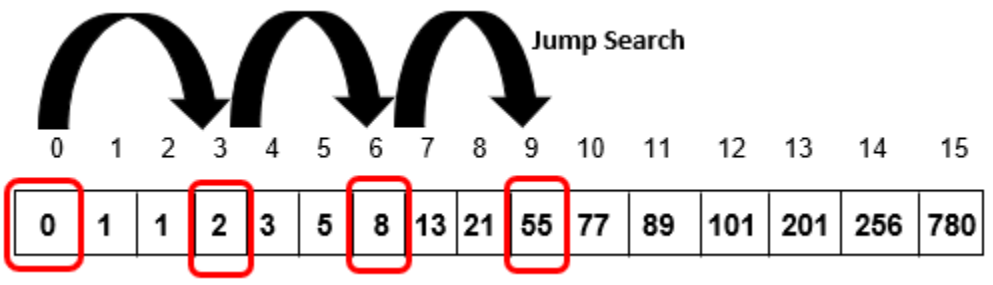


Figure 5: Comparing A[9] and item

Step 6: Compare A[12] with item. Since A[12] != item and A[12] >item, skip to A[9] (beginning of the current block) and perform a linear search.

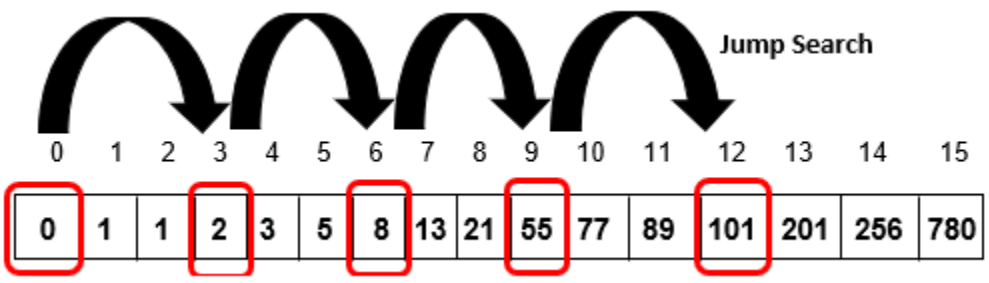


Figure 6: Comparing A[12] and item

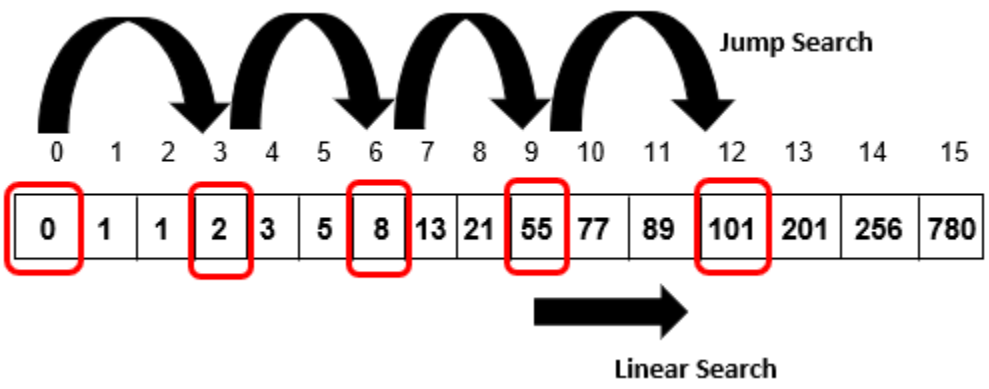


Figure 7: Comparing A[9] and item (Linear Search)

- Compare A[9] with item. Since A[9] != item, scan the next element
- Compare A[10] with item. Since A[10] == item, index 10 is printed as the valid location and the algorithm will terminate

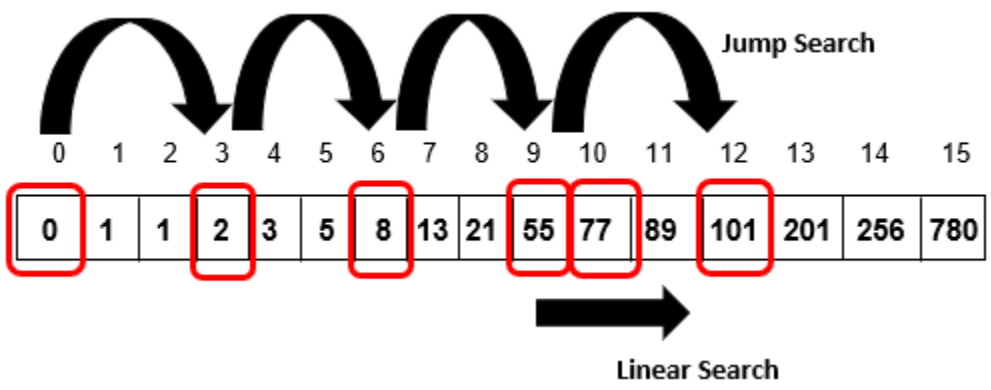


Figure 8: Comparing A[10] and item (Linear Search)

Implementation of Jump Search Algorithm

Following is the program in which we have implemented the Jump search algorithm in C++ language:

```
#include<iostream>

#include<cmath>



using namespace std;

int jump_Search(int a[], int n, int item) {

   int i = 0;

int m = sqrt(n); //initializing block size= √(n)


while(a[m] <= item && m < n) {

   // the control will continue to jump the blocks

   i = m;  // shift the block

   m += sqrt(n);

   if(m > n - 1)  // if m exceeds the array size

      return -1;

}


for(int x = i; x<m; x++) { //linear search in current block

   if(a[x] == item)

      return x; //position of element being searched

   }

   return -1;

}


int main() {

   int n, item, loc;

   cout << "\n Enter number of items: ";

   cin >> n;
```

```
   int arr[n]; //creating an array of size n

   cout << "\n Enter items: ";


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

      cin >> arr[i];

   }


   cout << "\n Enter search key to be found in the array: ";

   cin >> item;

   loc = jump_Search(arr, n, item);

   if(loc>=0)

      cout << "\n Item found at location: " << loc;

   else

      cout << "\n Item is not found in the list.";

}
```

Copy

The input array is the same as that used in the example:

```
Enter number of items: 16

Enter items: 0 1 1 2 3 5 8 13 21 55 77 89 101 201 256 780

Enter search key to be found in the array: 77

Item found at location: 10
```

Note: The algorithm can be implemented in any programming language as per the requirement.

Complexity Analysis for Jump Search

Let's see what will be the time and space complexity for the Jump search algorithm:

Time Complexity:

The while loop in the above C++ code executes n/m times because the loop counter increments by m times in every iteration. Since the optimal value of m= √n , thus, n/m=√n resulting in a time complexity of **O(√n)**.

Space Complexity:

The space complexity of this algorithm is **O(1)** since it does not requireany other data structure for its implementation.

Key Points to remember about Jump Search Algorithm

- This algorithm works only for sorted input arrays
- Optimal size of the block to be skipped is √n, thus resulting in the time complexity O(√n^2)
- The time complexity of this algorithm lies in between linear search (O(n)) and binary search (O(log n))
- It is also called block search algorithm

Advantages of Jump Search Algorithm

- It is faster than the linear search technique which has a time complexity of O(n) for searching an element

Disadvantages of Jump Search Algorithm

- It is slower than binary search algorithm which searches an element in O(log n)
- It requires the input array to be sorted

Introduction to Sorting

Sorting is nothing but arranging the data in ascending or descending order. The term **sorting** came into picture, as humans realised the importance of searching quickly.

There are so many things in our real life that we need to search for, like a particular record in database, roll numbers in merit list, a particular telephone number in telephone directory, a particular page in a book etc. All this would have been a mess if the data was kept unordered and unsorted, but fortunately the concept of **sorting** came into existence, making it easier for everyone to arrange data in an order, hence making it easier to search.

Sorting arranges data in a sequence which makes searching easier.

Sorting Efficiency

If you ask me, how will I arrange a deck of shuffled cards in order, I would say, I will start by checking every card, and making the deck as I move on.

It can take me hours to arrange the deck in order, but that's how I will do it.

Well, thank god, computers don't work like this.

Since the beginning of the programming age, computer scientists have been working on solving the problem of sorting by coming up with various different algorithms to sort data.

The two main criterias to judge which algorithm is better than the other have been:

1. Time taken to sort the given data.
2. Memory Space required to do so.

Different Sorting Algorithms

There are many different techniques available for sorting, differentiated by their efficiency and space requirements. Following are some sorting techniques which we will be covering in next few tutorials.

1. Bubble Sort
2. Insertion Sort
3. Selection Sort
4. Quick Sort
5. Merge Sort
6. Heap Sort

Although it's easier to understand these sorting techniques, but still we suggest you to first learn about Space complexity, Time complexity and the searching algorithms, to warm up your brain for sorting algorithms.

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.

NOTE: If you are not familiar with Sorting in data structure, you should first learn what is sorting to know about the basics of sorting.

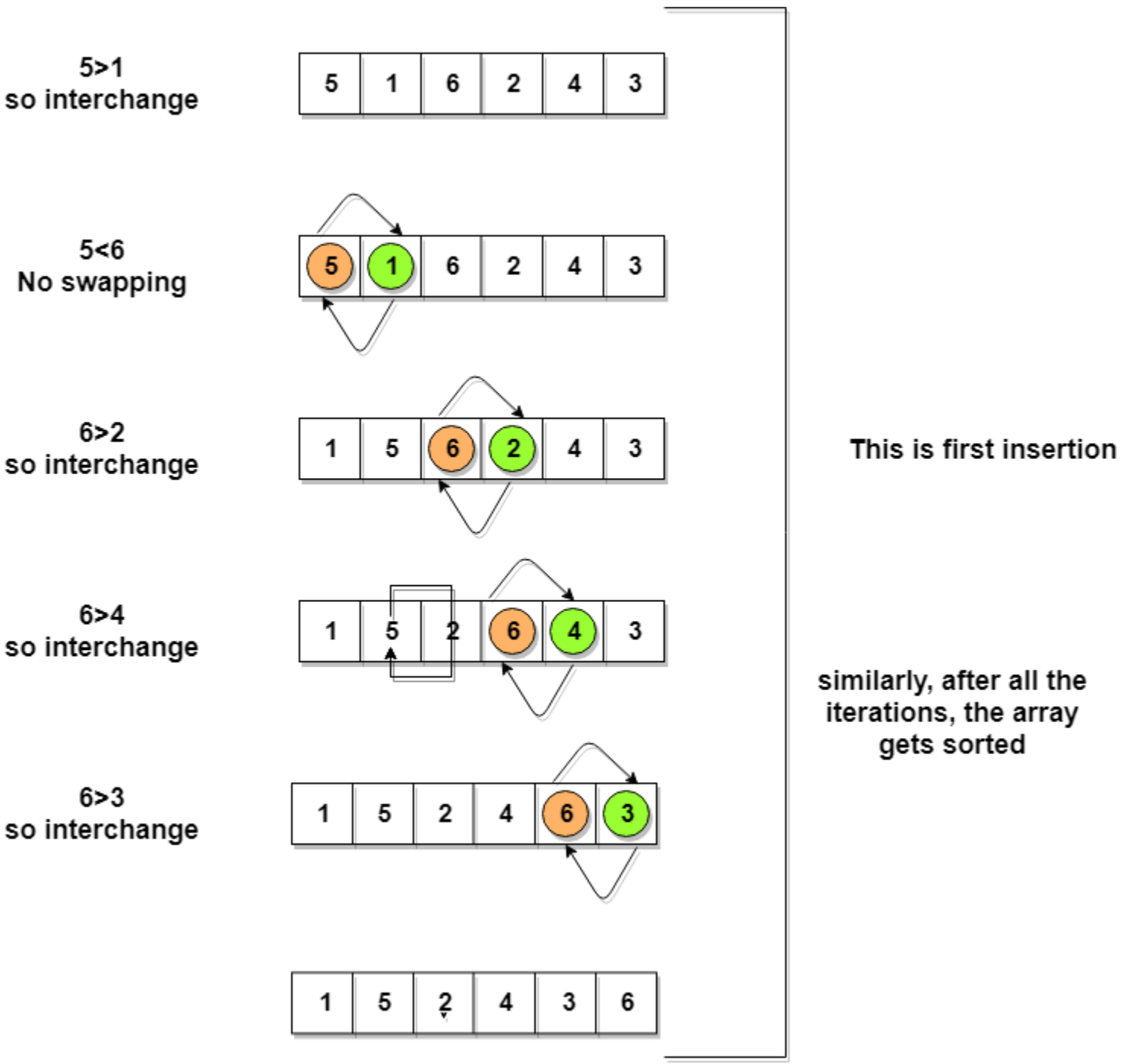
Implementing Bubble Sort Algorithm

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

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

Let's consider an array with values {5, 1, 6, 2, 4, 3}

Below, we have a pictorial representation of how bubble sort will sort the given array.



So as we can see in the representation above, after the first iteration, 6 is placed at the last index, which is the correct position for it.

Similarly after the second iteration, 5 will be at the second last index, and so on.

Time to write the code for bubble sort:

```c
// below we have a simple C program for bubble sort

#include <stdio.h>



void bubbleSort(int arr[], int n)

{

    int i, j, temp;

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

    {

        for(j = 0; j < n-i-1; j++)

    {

        if( arr[j] > arr[j+1])

        {

            // swap the elements

            temp = arr[j];

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

            arr[j+1] = temp;

        }

    }

}



// print the sorted array

    printf("Sorted Array: ");

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

    {

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

    }

}



int main()

{

    int arr[100], i, n, step, temp;

    // 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);

    return 0;
}
```

Copy

Although the above logic will sort an unsorted array, still the above algorithm is not efficient because as per the above logic, the outer for loop will keep on executing for **6** iterations even if the array gets sorted after the second iteration.

So, we can clearly optimize our algorithm.

Optimized Bubble Sort Algorithm

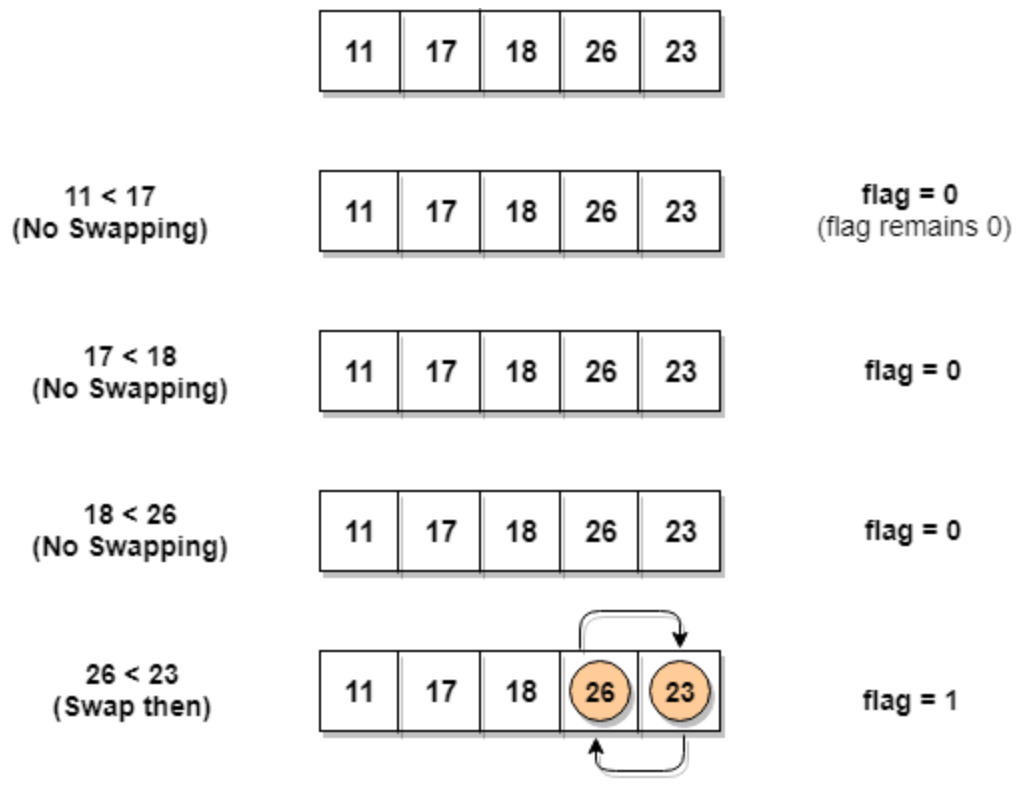
To optimize our bubble sort algorithm, we can introduce a flag to monitor whether elements are getting swapped inside the inner for loop.

Hence, in the inner for loop, we check whether swapping of elements is taking place or not, everytime.

If for a particular iteration, no swapping took place, it means the array has been sorted and we can jump out of the for loop, instead of executing all the iterations.

Let's consider an array with values {11, 17, 18, 26, 23}

Below, we have a pictorial representation of how the optimized bubble sort will sort the given array.



As we can see, in the first iteration, swapping took place, hence we updated our flag value to 1, as a result, the execution enters the for loop again. But in the second iteration, no swapping will occur, hence the value of flag will remain 0, and execution will break out of loop.

```c
// below we have a simple C program for bubble sort

#include <stdio.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 Array: ");

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

```
    {
        printf("%d  ", arr[i]);
    }
}


int main()
{
    int arr[100], i, n, step, temp;
    // 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);


    return 0;
}
```

Copy

```
Enter the number of elements to be sorted: 5
Enter element no. 1: 2
Enter element no. 2: 3
Enter element no. 3: 34
Enter element no. 4: 22
Enter element no. 5: 11
Sorted Array: 2  3  11  22  34
```

In the above code, in the function bubbleSort, if for a single complete cycle of j iteration(inner for loop), no swapping takes place, then flag will remain 0 and then we will break out of the for loops, because the array has already been sorted.

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,

```
(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(n²)**.

The main advantage of Bubble Sort is the simplicity of the algorithm.

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(n²)**
- Best Case Time Complexity [Big-omega]: **O(n)**
- Average Time Complexity [Big-theta]: **O(n²)**
- Space Complexity: **O(1)**

Now that we have learned Bubble sort algorithm, you can check out these sorting algorithms and their applications as well:

- Insertion Sort
- Selection Sort
- Quick Sort
- merge Sort
- Heap Sort
- Counting Sort

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?

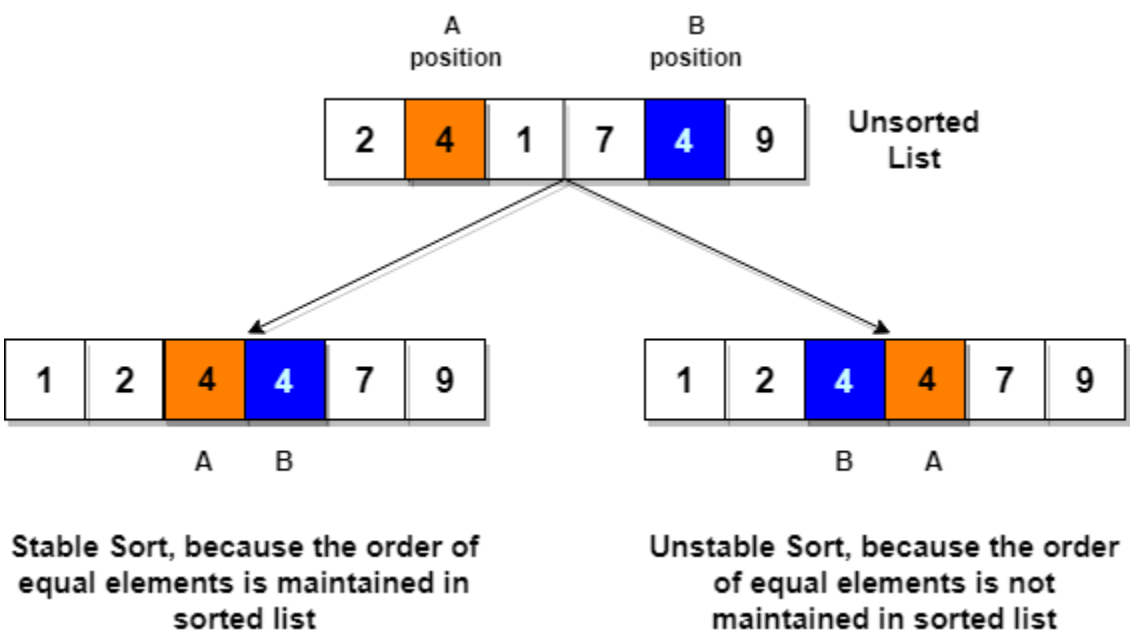
Well, you will have to go through each card from the starting or the back and find the right position for the new card, comparing it's value with each card. Once you find the right position, you will **insert** the card there.

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 starts from the index `1`(not `0`), and each index starting from index `1` is like a new card, that you have to place at the right position in the sorted subarray on the left.

Following are some of the important **characteristics of Insertion Sort**:

1. It is efficient for smaller data sets, but very inefficient for larger lists.
2. Insertion Sort is adaptive, that means it reduces its total number of steps if a partially sorted array is provided as input, making it efficient.
3. It is better than Selection Sort and Bubble Sort algorithms.
4. Its space complexity is less. Like bubble Sort, insertion sort also requires a single additional memory space.
5. It is a **stable** sorting technique, as it does not change the relative order of elements which are equal.



How Insertion Sort Works?

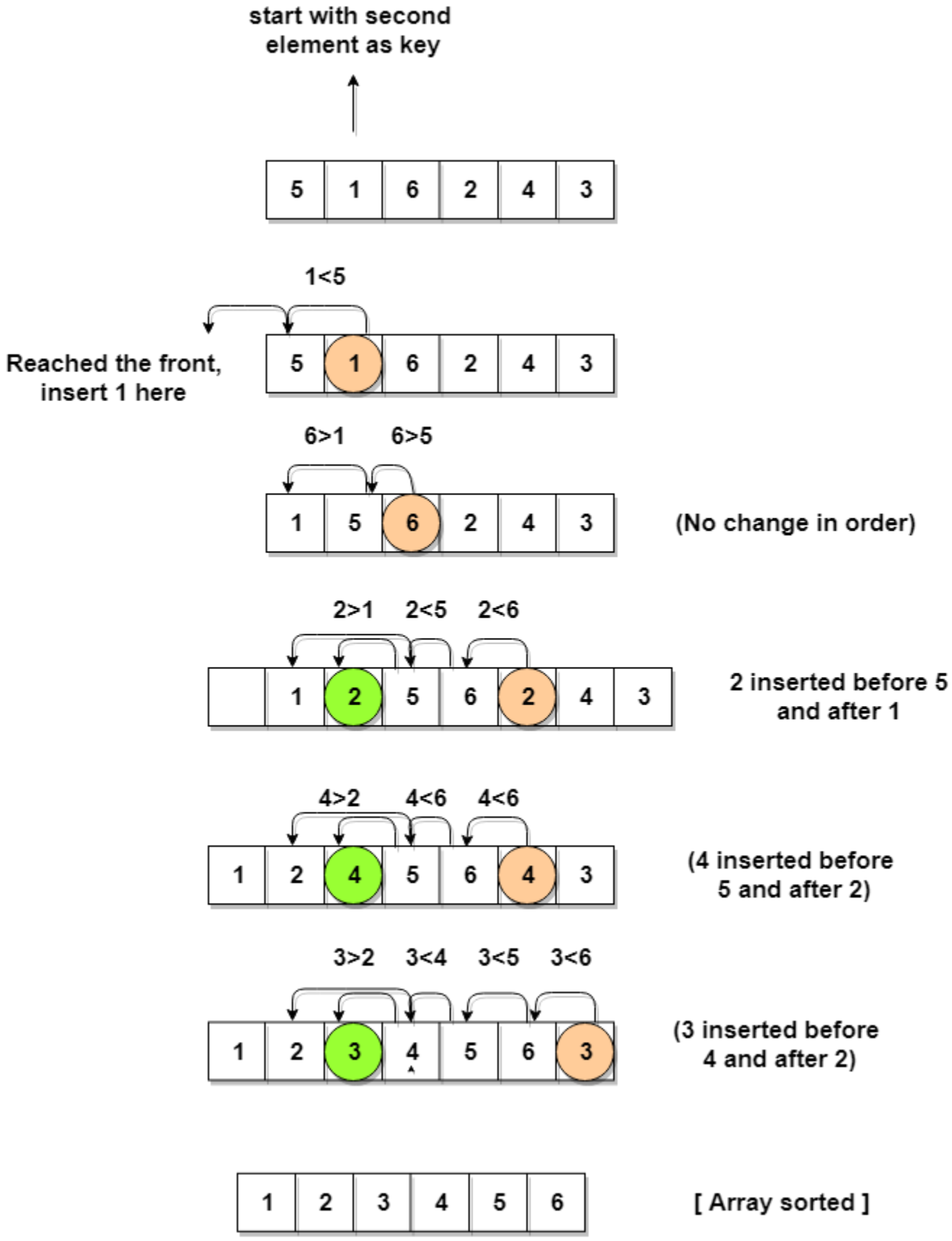
Following are the steps involved in insertion sort:

1. 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 sorted set of cards(remember the example with cards above).
2. 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.
3. 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.
4. And we go on repeating this, until the array is sorted.

Let's consider an array with values `{5, 1, 6, 2, 4, 3}`

Below, we have a pictorial representation of how bubble sort will sort the given array.



As you can see in the diagram above, after picking a `key`, we start iterating over the elements to the left of the `key`.

We continue to move towards left if the elements are greater than the `key` element and stop when we find the element which is less than the `key` element.

And, insert the `key` element after the element which is less than the `key` element.

Implementing Insertion Sort Algorithm

Below we have a simple implementation of Insertion sort in C++ language.

```
#include <stdlib.h>
#include <iostream>

using namespace std;

//member functions declaration
void insertionSort(int arr[], int length);
void printArray(int array[], int size);

// main function
int main()
{
     int array[5] = {5, 1, 6, 2, 4, 3};
     // calling insertion sort function to sort the array
     insertionSort(array, 6);
     return 0;
}

void insertionSort(int arr[], int length)
{
     int i, j, key;
     for (i = 1; i < length; i++)
     {
          j = i;
          while (j > 0 && arr[j - 1] > arr[j])
          {
               key = arr[j];
```

```
                arr[j] = arr[j - 1];

                arr[j - 1] = key;

                j--;

            }

        }

        cout << "Sorted Array: ";

        // print the sorted array

        printArray(arr, length);

}


// function to print the given array

void printArray(int array[], int size)

{

        int j;

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

        {

                cout <<" "<< array[j];

        }

        cout << endl;

}
```

Copy

```
Sorted Array:  1 2 3 4 5 6
```

Now let's try to understand the above simple insertion sort algorithm.

We took an array with **6** integers. We took a variable `key`, in which we put each element of the array, during each pass, starting from the **second** element, that is `a[1]`.

Then using the `while` loop, we iterate, until `j` becomes equal to **zero** or we find an element which is greater than `key`, and then we **insert** the `key` at that position.

We keep on doing this, until `j` becomes equal to **zero**, or we encounter an element which is smaller than the `key`, and then we stop. The current `key` is now at the right position.

We then make the next element as `key` and then repeat the same process.

In the above array, first we pick **1** as `key`, we compare it with **5**(element before 1), **1** is smaller than **5**, we insert **1** before **5**. Then we pick **6** as `key`, and compare it with **5** and **1**, no shifting in

position this time. Then **2** becomes the `key` and is compared with **6** and **5**, and then **2** is inserted after **1**. And this goes on until the complete array gets sorted.

Complexity Analysis of Insertion Sort

As we mentioned above that insertion sort is an efficient sorting algorithm, as it does not run on preset conditions using `for` loops, but instead it uses one `while` loop, which avoids extra steps once the array gets sorted.

Even though insertion sort is efficient, still, if we provide an already sorted array to the insertion sort algorithm, it will still execute the outer `for` loop, thereby requiring `n` steps to sort an already sorted array of `n` elements, which makes its **best case time complexity** a linear function of `n`.

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)**

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.

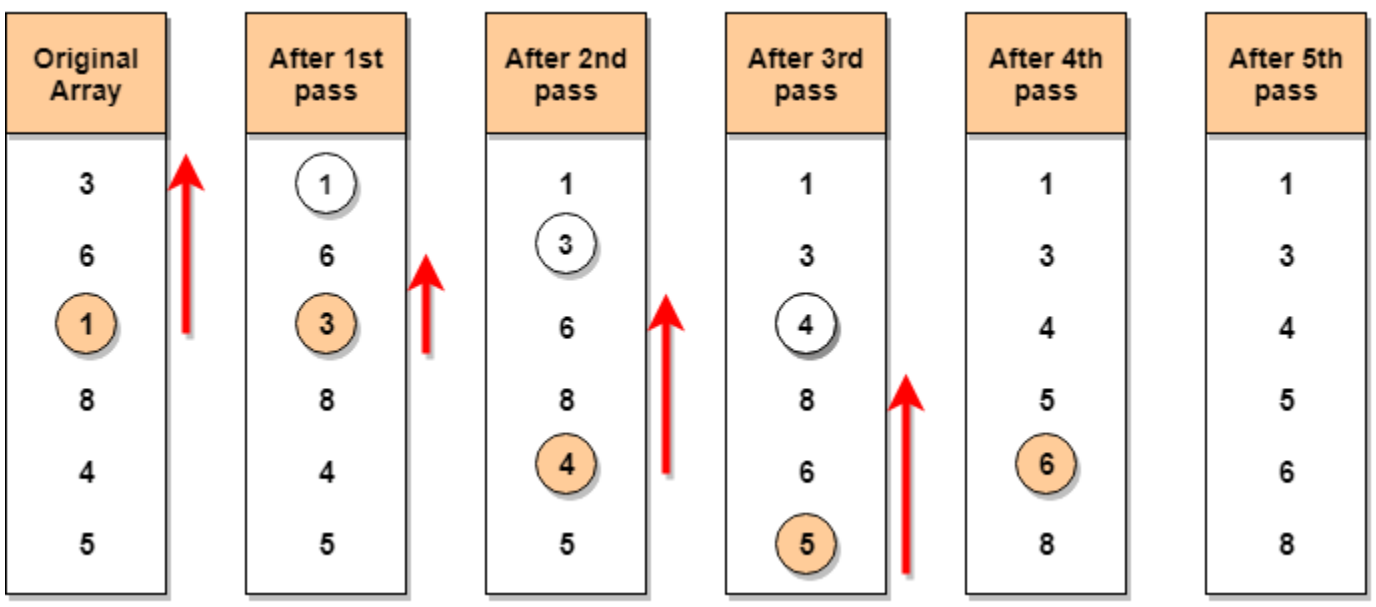
How Selection Sort Works?

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

1. Starting from the first element, we search the smallest element in the array, and replace it with the element in the first position.
2. We then move on to the second position, and look for smallest element present in the subarray, starting from index `1`, till the last index.
3. We replace the element at the **second** position in the original array, or we can say at the first position in the subarray, with the second smallest element.
4. This is repeated, until the array is completely sorted.

Let's consider an array with values `{3, 6, 1, 8, 4, 5}`

Below, we have a pictorial representation of how selection sort will sort the given array.



In the **first** pass, the smallest element will be `1`, so it will be placed at the first position.

Then leaving the first element, **next smallest** element will be searched, from the remaining elements. We will get `3` as the smallest, so it will be then placed at the second position.

Then leaving `1` and `3`(because they are at the correct position), we will search for the next smallest element from the rest of the elements and put it at third position and keep doing this until array is sorted.

Finding Smallest Element in a subarray

In selection sort, in the first step, we look for the smallest element in the array and replace it with the element at the first position. This seems doable, isn't it?

Consider that you have an array with following values $\{3, 6, 1, 8, 4, 5\}$. Now as per selection sort, we will start from the first element and look for the smallest number in the array, which is 1 and we will find it at the **index** 2. Once the smallest number is found, it is swapped with the element at the first position.

Well, in the next iteration, we will have to look for the second smallest number in the array. How can we find the second smallest number? This one is tricky?

If you look closely, we already have the smallest number/element at the first position, which is the right position for it and we do not have to move it anywhere now. So we can say, that the first element is sorted, but the elements to the right, starting from index 1 are not.

So, we will now look for the smallest element in the subarray, starting from index 1, to the last index.

Confused? Give it time to sink in.

After we have found the second smallest element and replaced it with element on index 1(which is the second position in the array), we will have the first two positions of the array sorted.

Then we will work on the subarray, starting from index 2 now, and again looking for the smallest element in this subarray.

Implementing Selection Sort Algorithm

In the C program below, we have tried to divide the program into small functions, so that it's easier fo you to understand which part is doing what.

There are many different ways to implement selection sort algorithm, here is the one that we like:

```
// C program implementing Selection Sort

# include <stdio.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++)
    {
        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("%d ", arr[i]);
    }
    printf("\n");
}


int main()
{
    int arr[] = {46, 52, 21, 22, 11};
    int n = sizeof(arr)/sizeof(arr[0]);
    selectionSort(arr, n);
    printf("Sorted array: \n");
    printArray(arr, n);
    return 0;
}
```

Copy

Note: Selection sort is an **unstable sort** i.e it might change the occurrence of two similar elements in the list while sorting. But it can also work as a stable sort when it is implemented using linked list.

Complexity Analysis of Selection Sort

Selection Sort requires two nested `for` loops to complete itself, one `for` loop is in the function `selectionSort`, and inside the first loop we are making a call to another function `indexOfMinimum`, which has the second(inner) `for` loop.

Hence 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)**

Quick Sort Algorithm

Quick Sort is one of the different Sorting Technique which is based on the concept of **Divide and Conquer**, just like merge sort. But in quick sort all the heavy lifting(major work) is done while **dividing** the array into subarrays, while in case of merge sort, all the real work happens during **merging** the subarrays. In case of quick sort, the combine step does absolutely nothing.

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**.

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 sunarrays, and same recursive logic will be applied on them, and we will keep doing this until the complete array is sorted.

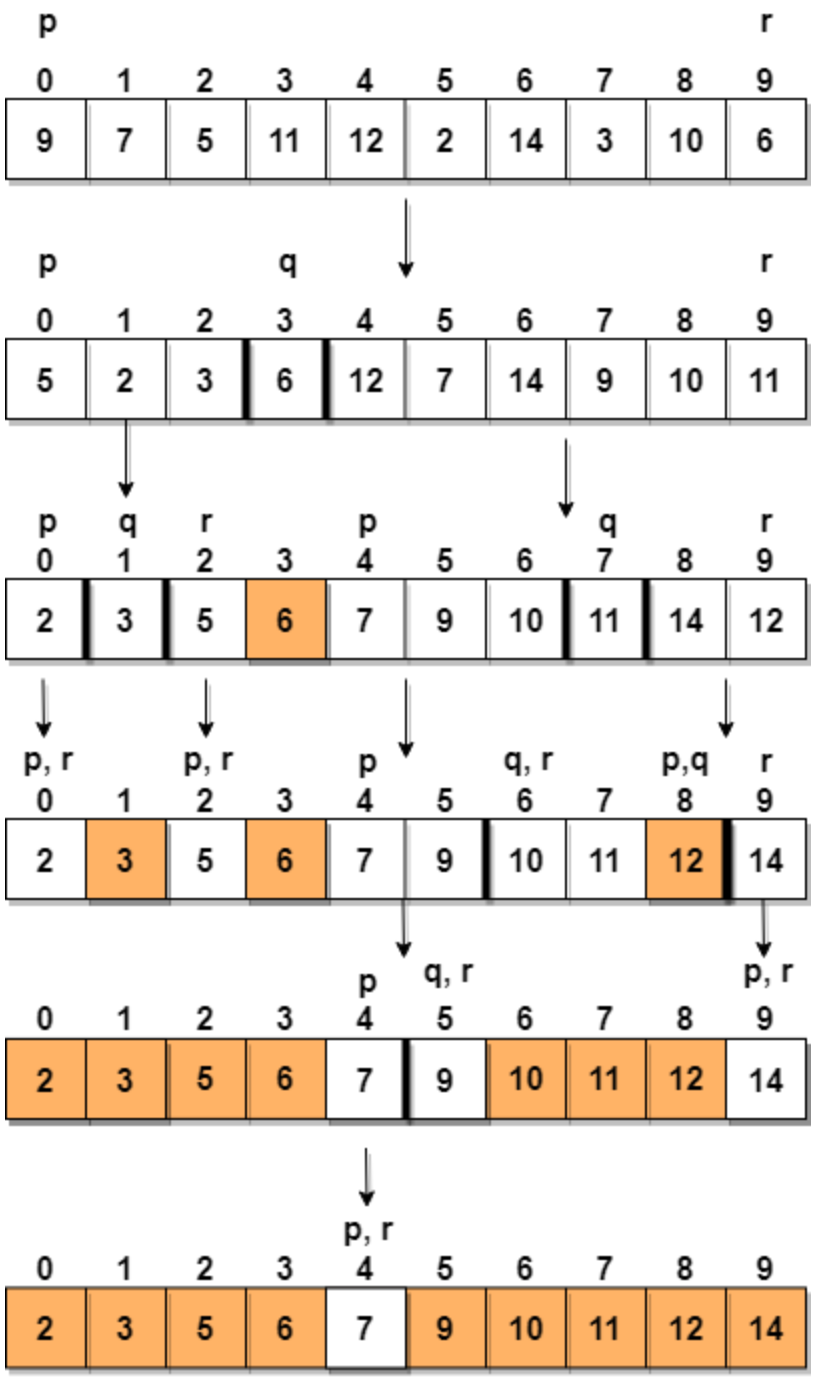
How Quick Sorting Works?

Following are the steps involved in quick sort algorithm:

1. After selecting an element as **pivot**, which is the last index of the array in our case, 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.

Let's consider an array with values {9, 7, 5, 11, 12, 2, 14, 3, 10, 6}

Below, we have a pictorial representation of how quick sort will sort the given array.



In step 1, we select the last element as the **pivot**, which is `6` in this case, and call for `partitioning`, hence re-arranging the array in such a way that `6` will be placed in its final position and to its left will be all the elements less than it and to its right, we will have all the elements greater than it.

Then we pick the subarray on the left and the subarray on the right and select a **pivot** for them, in the above diagram, we chose `3` as pivot for the left subarray and `11` as pivot for the right subarray.

And we again call for `partitioning`.

Implementing Quick Sort Algorithm

Below we have a simple C program implementing the Quick sort algorithm:

```
// simple C program for Quick Sort

#include <stdio.h>

int partition(int a[], int beg, int end);

void quickSort(int a[], int beg, int end);

void main()

{

    int i;

    int arr[10]={90,23,101,45,65,28,67,89,34,29};
```

```
    quickSort(arr, 0, 9);

    printf("\n The sorted array is: \n");

    for(i=0;i<10;i++)

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

}

int partition(int a[], int beg, int end)

{

    int left, right, temp, loc, flag;

    loc = left = beg;

    right = end;

    flag = 0;

    while(flag != 1)

    {

        while((a[loc] <= a[right]) && (loc!=right))

        right--;

        if(loc==right)

        flag =1;

        else if(a[loc]>a[right])

        {

            temp = a[loc];

            a[loc] = a[right];

            a[right] = temp;

            loc = right;

        }

        if(flag!=1)

        {

            while((a[loc] >= a[left]) && (loc!=left))

            left++;

            if(loc==left)

            flag =1;
```

```
            else if(a[loc] < a[left])
            {
                temp = a[loc];
                a[loc] = a[left];
                a[left] = temp;
                loc = left;
            }
        }
    }
    return loc;
}
void quickSort(int a[], int beg, int end)
{
    int loc;
    if(beg<end)
    {
        loc = partition(a, beg, end);
        quickSort(a, beg, loc-1);
        quickSort(a, loc+1, end);
    }
}
```

Copy

```
The sorted array is:
23      28      29      34      45      65      67      89      90      101
```

Complexity Analysis of Quick Sort

For an array, in which **partitioning** leads to unbalanced subarrays, to an extent where on the left side there are no elements, with all the elements greater than the **pivot**, hence on the right side.

And if keep on getting unbalanced subarrays, then the running time is the worst case, which is $O(n^2)$

Where as if **partitioning** leads to almost equal subarrays, then the running time is the best, with time complexity as **O(n*log n)**.

Worst Case Time Complexity [Big-O]: **$O(n^2)$**

Best Case Time Complexity [Big-omega]: **O(n*log n)**

Average Time Complexity [Big-theta]: **O(n*log n)**

Space Complexity: **O(n*log n)**

As we know now, that if subarrays **partitioning** produced after partitioning are unbalanced, quick sort will take more time to finish. If someone knows that you pick the last index as **pivot** all the time, they can intentionally provide you with array which will result in worst-case running time for quick sort.

To avoid this, you can pick random **pivot** element too. It won't make any difference in the algorithm, as all you need to do is, pick a random element from the array, swap it with element at the last index, make it the **pivot** and carry on with quick sort.

- Space required by quick sort is very less, only $O(n*\log n)$ additional space is required.
- Quick sort is not a stable sorting technique, so it might change the occurence of two similar elements in the list while sorting.

Now that we have learned Quick sort algorithm, you can check out these sorting algorithms and their applications as well:

- Insertion Sort
- Selection Sort
- Bubble Sort
- Merge sort
- Heap Sort
- Counting Sort

Merge Sort Algorithm

Merge Sort follows the rule of **Divide and Conquer** to sort a given set of numbers/elements, recursively, hence consuming less time.

Before moving forward with Merge Sort, check these topics out first:

- Selection Sort
- Insertion Sort
- Space Complexity of Algorithms
- Time Complexity of Algorithms

In the last two tutorials, we learned about Selection Sort and Insertion Sort, both of which have a worst-case running time of $O(n^2)$. As the size of input grows, insertion and selection sort can take a long time to run.

Merge sort , on the other hand, runs in $O(n*\log n)$ time in all the cases.

Before jumping on to, how merge sort works and it's implementation, first lets understand what is the rule of **Divide and Conquer**?

Divide and Conquer

If we can break a single big problem into smaller sub-problems, solve the smaller sub-problems and combine their solutions to find the solution for the original big problem, it becomes easier to solve the whole problem.

Let's take an example, **Divide and Rule**.

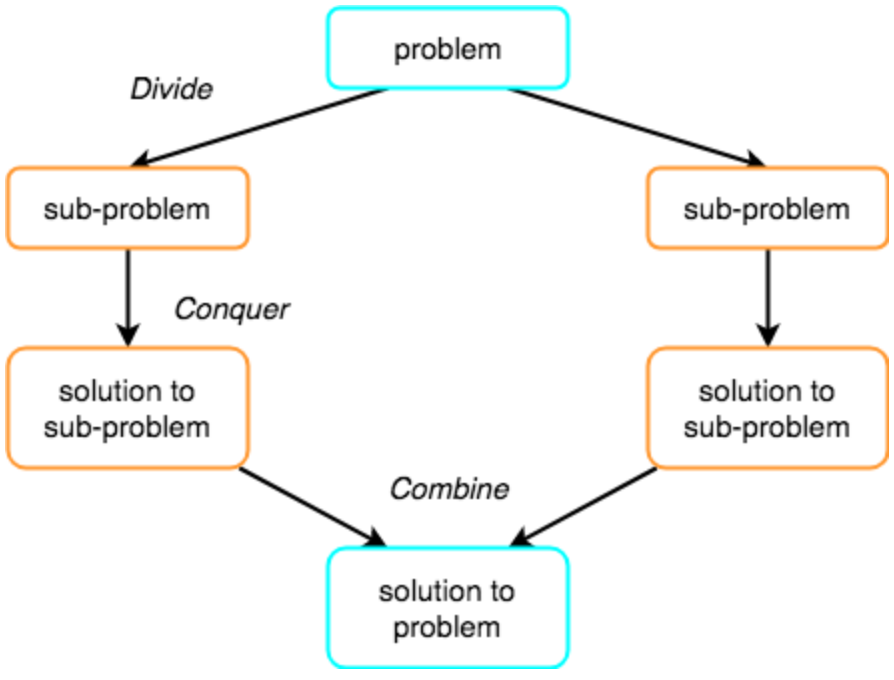
When Britishers came to India, they saw a country with different religions living in harmony, hard working but naive citizens, unity in diversity, and found it difficult to establish their empire. So, they adopted the policy of **Divide and Rule**. Where the population of India was collectively a one big problem for them, they divided the problem into smaller problems, by instigating rivalries between local kings, making them stand against each other, and this worked very well for them.

Well that was history, and a socio-political policy (**Divide and Rule**), but the idea here is, if we can somehow divide a problem into smaller sub-problems, it becomes easier to eventually solve the whole problem.

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

The concept of Divide and Conquer involves three steps:

1. **Divide** the problem into multiple small problems.
2. **Conquer** the subproblems by solving them. The idea is to break down the problem into atomic subproblems, where they are actually solved.
3. **Combine** the solutions of the subproblems to find the solution of the actual problem.



How Merge Sort Works?

As we have already discussed that merge sort utilizes divide-and-conquer rule to break the problem into sub-problems, the problem in this case being, **sorting a given array**.

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 subarrays with 3 elements each.

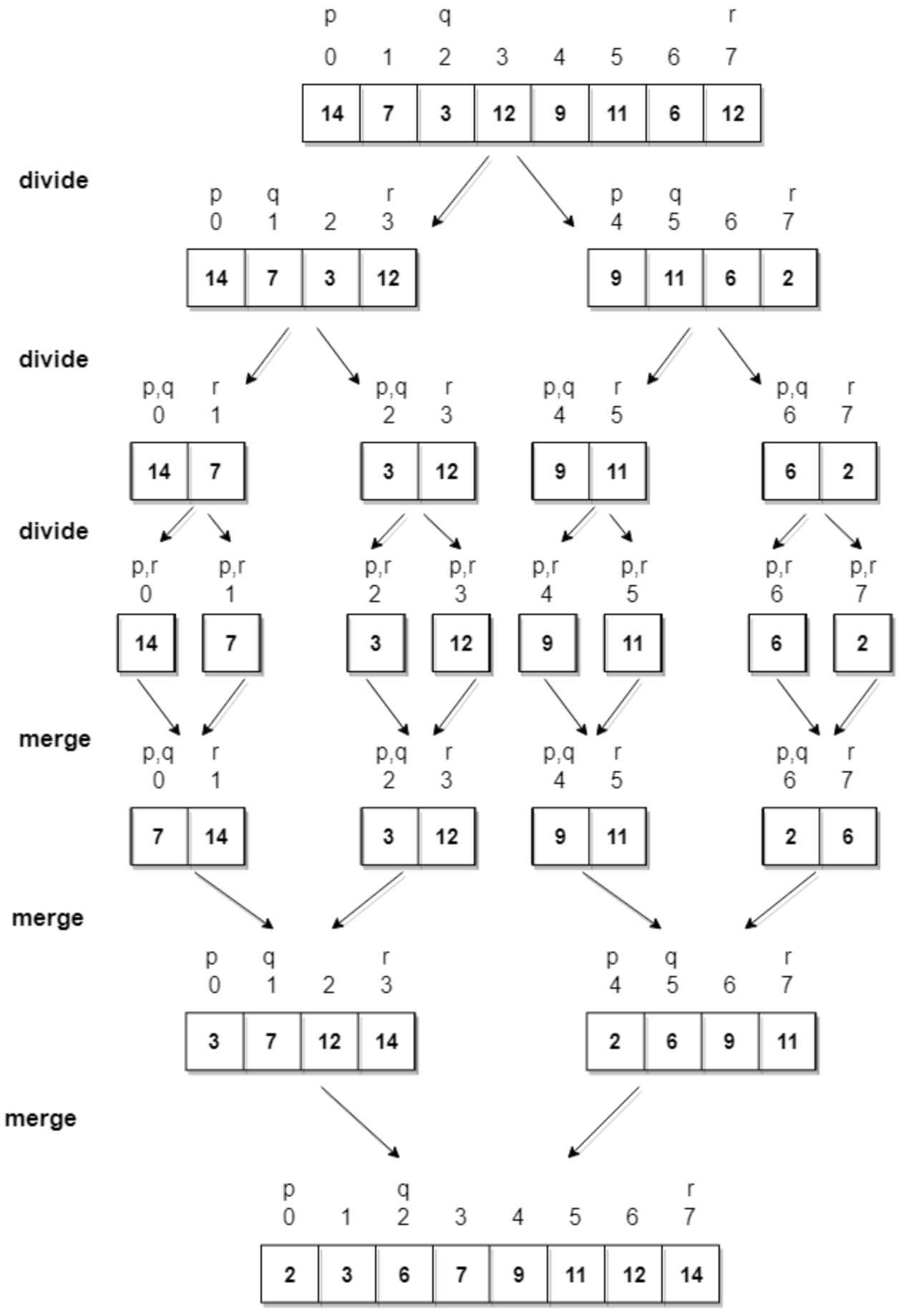
But breaking the orignal array into 2 smaller subarrays is not helping us in sorting the array.

So we will break these subarrays into even smaller subarrays, until we have multiple subarrays 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 subarrays which has only a single element, we have successfully broken down our problem into base problems.

And then we have to merge all these sorted subarrays, step by step to form one single sorted array.

Let's consider an array with values $\{14, 7, 3, 12, 9, 11, 6, 12\}$

Below, we have a pictorial representation of how merge sort will sort the given array.



In merge sort we follow the following steps:

1. 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.
2. 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 subarrays, from p to q and from $q + 1$ to r index.
3. Then we divide these 2 subarrays again, just like we divided our main array and this continues.
4. Once we have divided the main array into subarrays with single elements, then we start merging the subarrays.

Implementing Merge Sort Algorithm

Below we have a C program implementing merge sort algorithm.

```
/*
```

```
    a[] is the array, p is starting index, that is 0,
    and r is the last index of array.
*/

#include <stdio.h>

// lets take a[5] = {32, 45, 67, 2, 7} as the array to be sorted.

// merge sort function
void mergeSort(int a[], int p, int r)
{
    int q;
    if(p < r)
    {
        q = (p + r) / 2;
        mergeSort(a, p, q);
        mergeSort(a, q+1, r);
        merge(a, p, q, r);
    }
}

// function to merge the subarrays
void merge(int a[], int p, int q, int r)
{
    int b[5];   //same size of a[]
    int i, j, k;
    k = 0;
    i = p;
    j = q + 1;
    while(i <= q && j <= r)
```

```
    {
        if(a[i] < a[j])
        {
            b[k++] = a[i++];     // same as b[k]=a[i]; k++; i++;
        }
        else
        {
            b[k++] = a[j++];
        }
    }


    while(i <= q)
    {
        b[k++] = a[i++];
    }


    while(j <= r)
    {
        b[k++] = a[j++];
    }


    for(i=r; i >= p; i--)
    {
        a[i] = b[--k];  // copying back the sorted list to a[]
    }
}


// function to print the array
void printArray(int a[], int size)
{
```

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

int main()
{
    int arr[] = {32, 45, 67, 2, 7};
    int len = sizeof(arr)/sizeof(arr[0]);

    printf("Given array: \n");
    printArray(arr, len);

    // calling merge sort
    mergeSort(arr, 0, len - 1);

    printf("\nSorted array: \n");
    printArray(arr, len);
    return 0;
}
```

Copy

```
Given array:
32 45 67 2 7

Sorted array:
2 7 32 45 67
```

Complexity Analysis of Merge Sort

Merge Sort is quite fast, and has a time complexity of $O(n*\log n)$. It is also a stable sort, which means the "equal" elements are ordered in the same order in the sorted list.

In this section we will understand why the running time for merge sort is $O(n*\log n)$.

As we have already learned in Binary Search that whenever we divide a number into half in every step, it can be represented using a logarithmic function, which is $\log n$ and the number of steps can be represented by $\log n + 1$(at most)

Also, we perform a single step operation to find out the middle of any subarray, i.e. $O(1)$.

And to **merge** the subarrays, made by dividing the original array of n elements, a running time of $O(n)$ will be required.

Hence the total time for mergeSort function will become $n(\log n + 1)$, which gives us a time complexity of $O(n*\log n)$.

Worst Case Time Complexity [Big-O]: **O(n*log n)**

Best Case Time Complexity [Big-omega]: **O(n*log n)**

Average Time Complexity [Big-theta]: **O(n*log n)**

Space Complexity: **O(n)**

- Time complexity of Merge Sort is $O(n*\text{Log } n)$ in all the 3 cases (worst, average and best) as merge sort always **divides** the array in two halves and takes linear time to **merge** two halves.
- It requires **equal amount of additional space** as the unsorted array. Hence its not at all recommended for searching large unsorted arrays.
- It is the best Sorting technique used for sorting **Linked Lists**.

Now that we have learned Insertion sorting algorithm, you can check out these other sorting algorithm and their applications aswell :

- Bubble Sort
- Insertion Sort
- Quick Sort
- Selection Sort
- Heap Sort
- Counting Sort

Heap Sort Algorithm

Heap Sort is one of the best sorting methods being in-place and with no quadratic worst-case running time. Heap sort involves building a **Heap** data structure from the given array and then utilizing the Heap to sort the array.

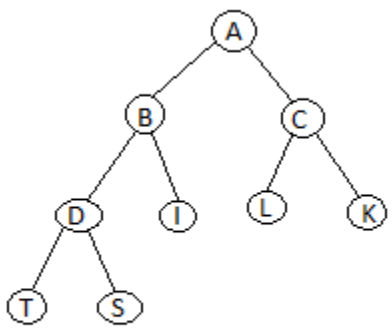
You must be wondering, how converting an array of numbers into a heap data structure will help in sorting the array. To understand this, let's start by understanding what is a Heap.

NOTE: If you are not familiar with Sorting in data structure, you should first learn what is sorting to know about the basics of sorting.

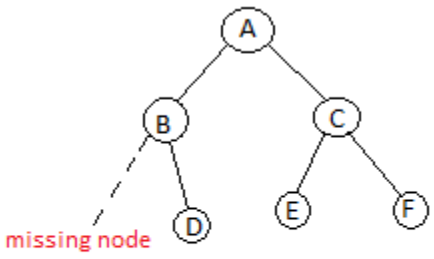
What is a Heap ?

Heap is a special tree-based data structure, that satisfies the following special heap properties:

1. **Shape Property:** Heap data structure is always a Complete Binary Tree, which means all levels of the tree are fully filled.

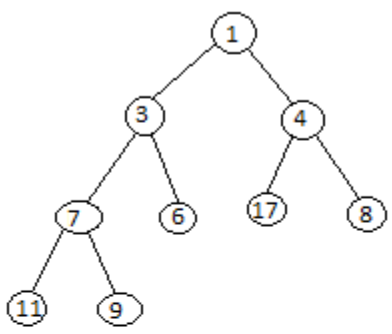


Complete Binary Tree



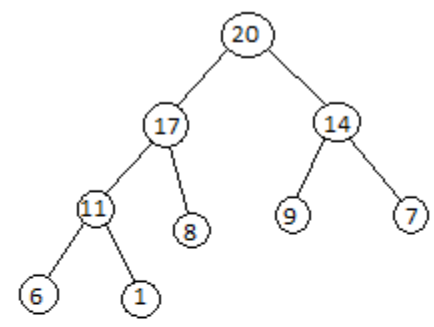
In-Complete Binary Tree

2. **Heap Property:** All nodes are either **greater than or equal to** or **less than or equal to** each of its children. If the parent nodes are greater than their child nodes, heap is called a **Max-Heap**, and if the parent nodes are smaller than their child nodes, heap is called **Min-Heap**.



Min-Heap

In min-heap, first element is the smallest. So when we want to sort a list in ascending order, we create a Min-heap from that list, and picks the first element, as it is the smallest, then we repeat the process with remaining elements.



Max-Heap

In max-heap, the first element is the largest, hence it is used when we need to sort a list in descending order.

How Heap Sort Works?

Heap sort algorithm is divided into two basic parts:

- Creating a Heap of the unsorted list/array.
- Then a sorted array is created by repeatedly removing the largest/smallest element from the heap, and inserting it into the array. The heap is reconstructed after each removal.

Initially on receiving an unsorted list, the first step in heap sort is to create a Heap data structure(Max-Heap or Min-Heap). Once heap is built, the first element of the Heap is either largest or smallest(depending upon Max-Heap or Min-Heap), so we put the first element of the heap in our array. Then we again make heap using the remaining elements, to again pick the first element of the heap and put it into the array. We keep on doing the same repeatedly untill we have the complete sorted list in our array.

In the below algorithm, initially `heapsort()` function is called, which calls `heapify()` to build the heap.

Implementing Heap Sort Algorithm

Below we have a simple C++ program implementing the Heap sort algorithm.

```
/*  Below program is written in C++ language  */


#include <iostream>


using namespace std;


void heapify(int arr[], int n, int i)
{
    int largest = i;
    int l = 2*i + 1;
    int r = 2*i + 2;


    // if left child is larger than root
    if (l < n && arr[l] > arr[largest])
        largest = l;


    // if right child is larger than largest so far
    if (r < n && arr[r] > arr[largest])
        largest = r;
```

```
    // if largest is not root
    if (largest != i)
    {
        swap(arr[i], arr[largest]);

        // recursively heapify the affected sub-tree
        heapify(arr, n, largest);
    }
}

void heapSort(int arr[], int n)
{
    // build heap (rearrange array)
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);

    // one by one extract an element from heap
    for (int i=n-1; i>=0; i--)
    {
        // move current root to end
        swap(arr[0], arr[i]);

        // call max heapify on the reduced heap
        heapify(arr, i, 0);
    }
}

/* function to print array of size n */
void printArray(int arr[], int n)
{
```

```
    for (int i = 0; i < n; i++)
    {
        cout << arr[i] << " ";
    }
    cout << "\n";
}



int main()
{
    int arr[] = {121, 10, 130, 57, 36, 17};
    int n = sizeof(arr)/sizeof(arr[0]);


    heapSort(arr, n);


    cout << "Sorted array is \n";
    printArray(arr, n);
}
```

Copy

Complexity Analysis of Heap Sort

Worst Case Time Complexity: **O(n*log n)**

Best Case Time Complexity: **O(n*log n)**

Average Time Complexity: **O(n*log n)**

Space Complexity : **O(1)**

- Heap sort is not a Stable sort, and requires a constant space for sorting a list.
- Heap Sort is very fast and is widely used for sorting.

Now that we have learned Heap sort algorithm, you can check out these sorting algorithms and their applications as well:

- Insertion Sort
- Selection Sort
- Bubble Sort

- Merge sort
- Heap Sort
- Counting Sort

Counting Sort Algorithm

Counting Sort Algorithm is an efficient sorting algorithm that can be used for sorting elements within a specific range. This sorting technique is based on the frequency/count of each element to be sorted and works using the following algorithm-

- **Input**: Unsorted array A[] of n elements
- **Output**: Sorted arrayB[]

Step 1: Consider an input array A having n elements in the range of 0 to k, where n and k are positive integer numbers. These n elements have to be sorted in ascending order using the counting sort technique. Also note that A[] can have distinct or duplicate elements

Step 2: The count/frequency of each distinct element in A is computed and stored in another array, say count, of size k+1. Let u be an element in A such that its frequency is stored at count[u].

Step 3: Update the count array so that element at each index, say i, is equal to -

$$count[i]= \sum count[u] \quad \text{where } 0<= u<= i$$

Step 4: The updated count array gives the index of each element of array A in the sorted sequence. Assume that the sorted sequence is stored in an output array, say B, of size n.

Step 5: Add each element from input array A to B as follows:

a. Set i=0 and $t = A[i]$

b. Add t to B[v] such that $v = (count[t]-1)$.

c. Decrement count[t] by 1

d. Increment i by 1

Repeat steps **(a)** to **(d)** till **i = n-1**

Step 6: Display B since this is the sorted array

Pictorial Representation of Counting Sort with an Example

Let us trace the above algorithm using an example:

Consider the following input array A to be sorted. All the elements are in range 0 to 9

```
A[]= {1, 3, 2, 8, 5, 1, 5, 1, 2, 7}
```

Copy

Step 1: Initialize an auxiliary array, say count and store the frequency of every distinct element. Size of count is 10 (k+1, such that range of elements in A is 0 to k)

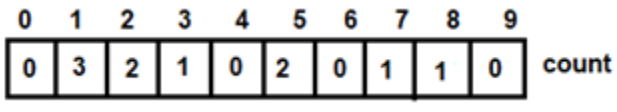


Figure 1: count array

Step 2: Using the formula, updated count array is -

$$count[i]= \sum count[u] \quad \text{where } 0<= u<= i$$

Figure 2: Formula for updating count array

0	1	2	3	4	5	6	7	8	9	
0	3	5	6	6	8	8	9	10	10	count

Figure 3 : Updated count array

Step 3: Add elements of array A to resultant array B using the following steps:

- For, i=0, t=1, count[1]=3, v=2. After adding 1 to B[2], count[1]=2 and i=1

0	1	2	3	4	5	6	7	8	9	
0	0	1	0	0	0	0	0	0	0	B
0	2	5	6	6	8	8	9	10	10	count

Figure 4: For i=0

- For i=1, t=3, count[3]=6, v=5. After adding 3 to B[5], count[3]=5 and i=2

0	1	2	3	4	5	6	7	8	9	
0	0	1	0	0	3	0	0	0	0	B
0	2	5	5	6	8	8	9	10	10	count

Figure 5: For i=1

- For i=2, t=2, count[2]= 5, v=4. After adding 2 to B[4], count[2]=4 and i=3

0	1	2	3	4	5	6	7	8	9	
0	0	1	0	2	3	0	0	0	0	B
0	2	4	5	6	8	8	9	10	10	count

Figure 6: For i=2

- For i=3, t=8, count[8]= 10, v=9. After adding 8 to B[9], count[8]=9 and i=4

0	1	2	3	4	5	6	7	8	9	
0	0	1	0	2	3	0	0	0	8	B
0	2	4	5	6	8	8	9	9	10	count

Figure 7: For i=3

- On similar lines, we have the following:

0	1	2	3	4	5	6	7	8	9	
0	0	1	0	2	3	0	5	0	8	B
0	2	4	5	6	7	8	9	9	10	count

Figure 8: For i=4

0	1	2	3	4	5	6	7	8	9	
0	1	1	0	2	3	0	5	0	8	B
0	1	4	5	6	7	8	9	9	10	count

Figure 9: For i=5

0	1	2	3	4	5	6	7	8	9	
0	1	1	0	2	3	5	5	0	8	B
0	1	4	5	6	6	8	9	9	10	count

Figure 10: For i=6

0	1	2	3	4	5	6	7	8	9	
1	1	1	0	2	3	5	5	0	8	B
0	0	4	5	6	6	8	9	9	10	count

Figure 11: For i=7

0	1	2	3	4	5	6	7	8	9	
1	1	1	2	2	3	5	5	0	8	B
0	0	3	5	6	6	8	9	9	10	count

Figure 12: For i=8

Figure 13: For i=9

Thus, array B has the sorted list of elements.

Program for Counting Sort Algorithm

Below we have a simple program in C++ implementing the counting sort algorithm:

```
#include<iostream>

using namespace std;



int k=0;  // for storing the maximum element of input array



/* Method to sort the array */

void sort_func(int A[],int B[],int n)

{

     int count[k+1],t;

     for(int i=0;i<=k;i++)

     {

          //Initialize array count

          count[i] = 0;

     }

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

     {

         // count the occurrence of elements u of A

         // & increment count[u] where u=A[i]

          t = A[i];

          count[t]++;

     }

     for(int i=1;i<=k;i++)

     {
```

```
        // Updating elements of count array
        count[i] = count[i]+count[i-1];
    }
    for(int i=0;i<n;i++)
    {
        // Add elements of array A to array B
        t = A[i];
        B[count[t]] = t;
        // Decrement the count value by 1
        count[t]=count[t]-1;
        }
}
int main()
{
    int n;
    cout<<"Enter the size of the array :";
    cin>>n;

    // A is the input array and will store elements entered by the
user
    // B is the output array having the sorted elements
    int A[n],B[n];
    cout<<"Enter the array elements: ";
    for(int i=0;i<n;i++)
    {
        cin>>A[i];
        if(A[i]>k)
        {
            // k will have the maximum element of A[]
            k = A[i];
        }
```

```
    }


    sort_func(A,B,n);


    // Printing the elements of array B
    for(int i=1;i<=n;i++)
    {
         cout<<B[i]<<" ";
    }

         cout<<"\n";
         return 0;
}
```

Copy

The input array is the same as that used in the example:

```
Enter the size of the array :10
Enter the array elements: 1 3 2 8 5 1 5 1 2 7
1 1 1 2 2 3 5 5 7 8


...Program finished with exit code 0
Press ENTER to exit console.
```

Figure 14: Output of Program

Note: The algorithm can be mapped to any programming language as per the requirement.

Time Complexity Analysis

For scanning the input array elements, the loop iterates n times, thus taking $O(n)$ running time. The sorted array B[] also gets computed in n iterations, thus requiring O(n) running time. The count array also uses k iterations, thus has a running time of O(k). Thus the total running time for counting sort algorithm is $O(n+k)$.

Key Points:

- The above implementation of Counting Sort can also be extended to sort negative input numbers
- Since counting sort is suitable for sorting numbers that belong to a well-defined, finite and small range, it can be used asa subprogram in other sorting algorithms like radix sort which can be used for sorting numbers having a large range
- Counting Sort algorithm is efficient if the range of input data (k) is not much greater than the number of elements in the input array (n). It will not work if we have 5 elements to sort in the range of 0 to 10,000

- It is an integer-based sorting algorithm unlike others which are usually comparison-based. A comparison-based sorting algorithm sorts numbers only by comparing pairs of numbers. Few examples of comparison based sorting algorithms are **quick sort, merge sort, bubble sort, selection sort, heap sort, insertion sort**, whereas algorithms like radix sort, bucket sort and comparison sort fall into the category of non-comparison based sorting algorithms.

Advantages of Counting Sort:

- It is quite fast
- It is a stable algorithm

Note: For a sorting algorithm to be stable, the order of elements with equal keys (values) in the sorted array should be the same as that of the input array.

Disadvantages of Counting Sort:

- It is not suitable for sorting large data sets
- It is not suitable for sorting string values
