Count Sort	**Ω(N + k)**	**Θ(N + k)**	**O(N + k)**	**O(k)**
Bucket Sort	**Ω(N + k)**	**Θ(N + k)**	**O(N2)**	**O(N)**

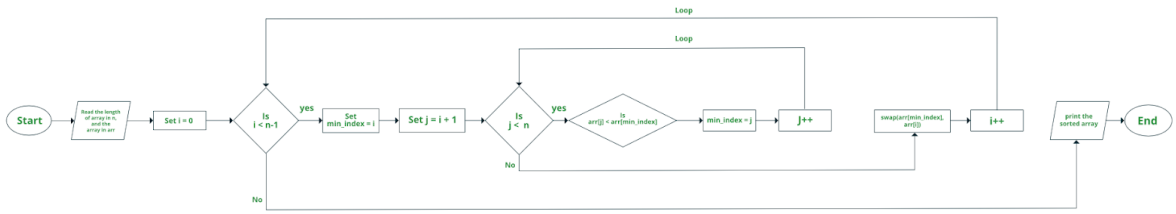
Selection Sort:

The **selection sort algorithm** sorts an array by repeatedly finding the minimum element (considering ascending order) from the unsorted part and putting it at the beginning.
The algorithm maintains two subarrays in a given array.

- The subarray which already sorted.
- The remaining subarray was unsorted.

In every iteration of the selection sort, the minimum element (considering ascending order) from the unsorted subarray is picked and moved to the sorted subarray.

Flowchart of the Selection Sort:



Flowchart for Selection Sort

How selection sort works?

Lets consider the following array as an example: ***arr[] = {64, 25, 12, 22, 11}***

First pass:

- *For the first position in the sorted array, the whole array is traversed from index 0 to 4 sequentially. The first position where* ***64*** *is stored presently, after traversing whole array it is clear that* ***11*** *is the lowest value.*

64 *25 12 22 11*

- *Thus, replace 64 with 11. After one iteration* ***11****, which happens to be the least value in the array, tends to appear in the first position of the sorted list.*

11 *25 12 22 64*

Second Pass:

- *For the second position, where 25 is present, again traverse the rest of the array in a sequential manner.*

*11 **25** 12 22 64*

- *After traversing, we found that **12** is the second lowest value in the array and it should appear at the second place in the array, thus swap these values.*

*11 **12** 25 22 64*

Third Pass:

- *Now, for third place, where **25** is present again traverse the rest of the array and find the third least value present in the array.*

*11 12 **25** 22 64*

- *While traversing, **22** came out to be the third least value and it should appear at the third place in the array, thus swap **22** with element present at third position.*

*11 12 **22** 25 64*

Fourth pass:

- *Similarly, for fourth position traverse the rest of the array and find the fourth least element in the array*
- *As **25** is the 4th lowest value hence, it will place at the fourth position.*

*11 12 22 **25** 64*

Fifth Pass:

- *At last the largest value present in the array automatically get placed at the last position in the array*
- *The resulted array is the sorted array.*

*11 12 22 **25** 64*

Follow the below steps to solve the problem:

- Initialize minimum value(**min_idx**) to location 0.
- Traverse the array to find the minimum element in the array.
- While traversing if any element smaller than **min_idx** is found then swap both the values.
- Then, increment **min_idx** to point to the next element.
- Repeat until the array is sorted.

Below is the implementation of the above approach:

- C++
- C
- Python3
- Java
- C#
- PHP
- Javascript

```
// C++ program for implementation of
// selection sort
#include <bits/stdc++.h>
using namespace std;

//Swap function
void swap(int *xp, int *yp)
{
    int temp = *xp;
    *xp = *yp;
    *yp = temp;
}

void selectionSort(int arr[], int n)
{
    int i, j, min_idx;

    // One by one move boundary of
    // unsorted subarray
    for (i = 0; i < n-1; i++)
```

```
    {

        // Find the minimum element in
        // unsorted array
        min_idx = i;
        for (j = i+1; j < n; j++)
        if (arr[j] < arr[min_idx])
            min_idx = j;

        // Swap the found minimum element
        // with the first element
        if(min_idx!=i)
            swap(&arr[min_idx], &arr[i]);
    }
}

//Function to print an array
void printArray(int arr[], int size)
{
    int i;
```

```
    for (i=0; i < size; i++)
        cout << arr[i] << " ";
    cout << endl;
}

// Driver program to test above functions
int main()
{
    int arr[] = {64, 25, 12, 22, 11};
    int n = sizeof(arr)/sizeof(arr[0]);
    selectionSort(arr, n);
    cout << "Sorted array: \n";
    printArray(arr, n);
    return 0;
}
// This is code is contributed by rathbhupendra
```

Output

Sorted array:

11 12 22 25 64

Complexity Analysis of Selection Sort:

Time Complexity: The time complexity of Selection Sort is O(N2) as there are two nested loops:

- One loop to select an element of Array one by one = O(N)
- Another loop to compare that element with every other Array element = O(N)

Therefore overall complexity = O(N) * O(N) = O(N*N) = O(N2)
Auxiliary Space: O(1) as the only extra memory used is for temporary variables while swapping two values in Array. The selection sort never makes more than O(N) swaps and can be useful when memory write is a costly operation.

Bubble Sort:

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order. This algorithm is not suitable for large data sets as its average and worst-case time complexity is quite high.
How does Bubble Sort Work?

Input: *arr[] = {5, 1, 4, 2, 8}*

First Pass:

- *Bubble sort starts with very first two elements, comparing them to check which one is greater.*
    - *(**5 1** 4 2 8) –> (**1 5** 4 2 8), Here, algorithm compares the first two elements, and swaps since 5 > 1.*
    - *(1 **5 4** 2 8) –> (1 **4 5** 2 8), Swap since 5 > 4*
    - *(1 4 **5 2** 8) –> (1 4 **2 5** 8), Swap since 5 > 2*
    - *(1 4 2 **5 8**) –> (1 4 2 **5 8**), Now, since these elements are already in order (8 > 5), algorithm does not swap them.*

Second Pass:

- *Now, during second iteration it should look like this:*
    - *(**1 4** 2 5 8) –> (**1 4** 2 5 8)*
    - *(1 **4 2** 5 8) –> (1 **2 4** 5 8), Swap since 4 > 2*
    - *(1 2 **4 5** 8) –> (1 2 **4 5** 8)*
    - *(1 2 4 **5 8**) –> (1 2 4 **5 8**)*

Third Pass:

- *Now, the array is already sorted, but our algorithm does not know if it is completed.*
- *The algorithm needs one **whole** pass without **any** swap to know it is sorted.*
    - *(**1 2** 4 5 8) –> (**1 2** 4 5 8)*
    - *(1 **2 4** 5 8) –> (1 **2 4** 5 8)*
    - *(1 2 **4 5** 8) –> (1 2 **4 5** 8)*
    - *(1 2 4 **5 8**) –> (1 2 4 **5 8**)*

Illustration:

i = 0	j	0	1	2	3	4	5	6	7
	0	5	3	1	9	8	2	4	7
	1	3	5	1	9	8	2	4	7
	2	3	1	5	9	8	2	4	7
	3	3	1	5	9	8	2	4	7
	4	3	1	5	8	9	2	4	7
	5	3	1	5	8	2	9	4	7
	6	3	1	5	8	2	4	9	7
i=1	0	3	1	5	8	2	4	7	9
	1	1	3	5	8	2	4	7	
	2	1	3	5	8	2	4	7	
	3	1	3	5	8	2	4	7	
	4	1	3	5	2	8	4	7	
	5	1	3	5	2	4	8	7	
i=2	0	1	3	5	2	4	7	8	
	1	1	3	5	2	4	7		
	2	1	3	5	2	4	7		
	3	1	3	2	5	4	7		
	4	1	3	2	4	5	7		
i=3	0	1	3	2	4	5	7		
	1	1	3	2	4	5			
	2	1	2	3	4	5			
	3	1	2	3	4	5			
i=4	0	1	2	3	4	5			
	1	1	2	3	4				
	2	1	2	3	4				
i=5	0	1	2	3	4				
	1	1	2	3					
i=6	0	1	2	3					
		1	2						

Follow the below steps to solve the problem:

- Run a nested for loop to traverse the input array using two variables **i** and **j**, such that $0 \leq i < n-1$ and $0 \leq j < n-i-1$
- If **arr[j]** is greater than **arr[j+1]** then swap these adjacent elements, else move on
- Print the sorted array

Below is the implementation of the above approach:

- C

- C++
- Java
- Python3
- C#
- PHP
- Javascript

```
// C program for implementation of Bubble sort
#include <stdio.h>

void swap(int* xp, int* yp)
{
    int temp = *xp;
    *xp = *yp;
    *yp = temp;
}

// A function to implement bubble sort
void bubbleSort(int arr[], int n)
{
    int i, j;
    for (i = 0; i < n - 1; i++)

        // Last i elements are already in place
        for (j = 0; j < n - i - 1; j++)
            if (arr[j] > arr[j + 1])
                swap(&arr[j], &arr[j + 1]);
```

```
}

/* Function to print an array */
void printArray(int arr[], int size)
{
    int i;
    for (i = 0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

// Driver program to test above functions
int main()
{
    int arr[] = { 64, 34, 25, 12, 22, 11, 90 };
    int n = sizeof(arr) / sizeof(arr[0]);
    bubbleSort(arr, n);
    printf("Sorted array: \n");
    printArray(arr, n);
    return 0;
```

```
}
```

Output

Sorted array:

1 2 4 5 8
Time Complexity: O(N2)
Auxiliary Space: O(1)
Optimized Implementation of Bubble Sort:
The above function always runs **O(N2)** time even if the array is sorted. It can be optimized by stopping the algorithm if the inner loop didn’t cause any swap.

Below is the implementation for the above approach:

- C
- C++
- Java
- Python3
- C#
- PHP
- Javascript

```
// Optimized implementation of Bubble sort
#include <stdio.h>
#include <stdbool.h>

void swap(int *xp, int *yp)
{
    int temp = *xp;
    *xp = *yp;
    *yp = temp;
}

// An optimized version of Bubble Sort
void bubbleSort(int arr[], int n)
{
   int i, j;
   bool swapped;
   for (i = 0; i < n-1; i++)
   {
     swapped = false;
     for (j = 0; j < n-i-1; j++)
```

```
    {
      if (arr[j] > arr[j+1])
      {
        swap(&arr[j], &arr[j+1]);
        swapped = true;
      }
    }

    // IF no two elements were swapped by inner loop, then
break
    if (swapped == false)
      break;
  }
}

/* Function to print an array */
void printArray(int arr[], int size)
{
   int i;
   for (i=0; i < size; i++)
      printf("%d ", arr[i]);
```

```
    printf("n");
}

// Driver program to test above functions
int main()
{
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int n = sizeof(arr)/sizeof(arr[0]);
    bubbleSort(arr, n);
    printf("Sorted array: \n");
    printArray(arr, n);
    return 0;
}
```

Output

Sorted array:

1 2 3 4 5 7 8 9

Time Complexity: O(N2)
Auxiliary Space: O(1)
Worst Case Analysis for Bubble Sort:
The **worst-case** condition for bubble sort occurs when elements of the array are arranged in decreasing order.
In the worst case, the total number of iterations or passes required to sort a given array is **(n-1).** where 'n' is a number of elements present in the array.

At pass 1 : *Number of comparisons = (n-1)*

Number of swaps = (n-1)

At pass 2 : *Number of comparisons = (n-2)*

Number of swaps = (n-2)

At pass 3 : *Number of comparisons = (n-3)*

Number of swaps = (n-3)

.

.

.

At pass n-1 : *Number of comparisons = 1*

Number of swaps = 1

Now , calculating total number of comparison required to sort the array

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

= (n-1)(n-1+1)/2 { by using sum of N natural Number formula }*

= ***n (n-1)/2***

For the Worst case:

Total number of swaps = Total number of comparison

Total number of comparison (Worst case) = ***n(n-1)/2***

Total number of swaps (Worst case) = ***n(n-1)/2***

Worst and Average Case Time Complexity: *O(N2). The worst case occurs when an array is reverse sorted.*

Best Case Time Complexity: *O(N). The best case occurs when an array is already sorted.*

Auxiliary Space: *O(1)*

Recursive Implementation Of Bubble Sort:

The idea is to place the largest element in its position and keep doing the same for every other element.

Follow the below steps to solve the problem:

- Place the largest element at its position, this operation makes sure that the first largest element will be placed at the end of the array.
- Recursively call for rest **n – 1** elements with the same operation and place the next greater element at their position.
- The base condition for this recursion call would be, when the number of elements in the array becomes **0** or **1** then, simply return (as they are already sorted).

Below is the implementation of the above approach:

- C++
- Java
- C#

```
//C++ code for recursive bubble sort algorithm
#include <iostream>
using namespace std;
void bubblesort(int arr[], int n)
{
    if (n == 0 || n == 1)
    {
        return;
    }
    for (int i = 0; i < n - 1; i++)
    {
        if (arr[i] > arr[i + 1])
        {
            swap(arr[i], arr[i + 1]);
        }
    }
    bubblesort(arr, n - 1);
}
int main()
{
```

```
    int arr[5] = {2, 5, 1, 6, 9};
    bubblesort(arr, 5);
    for (int i = 0; i < 5; i++)
    {
        cout << arr[i] << " ";
    }
    return 0;
}
//code contributed by pragatikohli
```

Output

1 2 5 6 9

What is the Boundary Case for Bubble sort?

Bubble sort takes minimum time (Order of n) when elements are already sorted. Hence it is best to check if the array is already sorted or not beforehand, to avoid O(N2) time complexity.

Does sorting happen in place in Bubble sort?

Yes, Bubble sort performs swapping of adjacent pairs without the use of any major data structure. Hence Bubble sort algorithm is an in-place algorithm.

Is the Bubble sort algorithm stable?

Yes, the bubble sort algorithm is stable.

Where is the Bubble sort algorithm used?

Due to its simplicity, bubble sort is often used to introduce the concept of a sorting algorithm.

In computer graphics, it is popular for its capability to detect a tiny error (like a swap of just two elements) in almost-sorted arrays and fix it with just linear
complexity (2n).

Example: It is used in a polygon filling algorithm, where bounding lines are sorted by their x coordinate at a specific scan line (a line parallel to the x-axis), and with incrementing y their order changes (two elements are swapped) only at intersections of two lines

Insertion Sort:

Insertion sort is a simple sorting algorithm that works similar to the way you sort playing cards in your hands. The array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed at the correct position in the sorted part.

Characteristics of Insertion Sort:

- This algorithm is one of the simplest algorithm with simple implementation
- Basically, Insertion sort is efficient for small data values
- Insertion sort is adaptive in nature, i.e. it is appropriate for data sets which are already partially sorted.

Working of Insertion Sort algorithm:

Consider an example: arr[]: {12, 11, 13, 5, 6}

12 11 13 5 6

First Pass:

- *Initially, the first two elements of the array are compared in insertion sort.*

12 ***11*** *13 5 6*

- *Here, 12 is greater than 11 hence they are not in the ascending order and 12 is not at its correct position. Thus, swap 11 and 12.*
- *So, for now 11 is stored in a sorted sub-array.*

11 ***12*** *13 5 6*

Second Pass:

- *Now, move to the next two elements and compare them*

*11 **12** **13** 5 6*

- *Here, 13 is greater than 12, thus both elements seems to be in ascending order, hence, no swapping will occur. 12 also stored in a sorted sub-array along with 11*

Third Pass:

- *Now, two elements are present in the sorted sub-array which are **11** and **12***
- *Moving forward to the next two elements which are 13 and 5*

*11 12 **13** **5** 6*

- *Both 5 and 13 are not present at their correct place so swap them*

*11 12 **5** **13** 6*

- *After swapping, elements 12 and 5 are not sorted, thus swap again*

*11 **5** **12** 13 6*

- *Here, again 11 and 5 are not sorted, hence swap again*

***5** **11** 12 13 6*

- *here, it is at its correct position*

Fourth Pass:

- *Now, the elements which are present in the sorted sub-array are **5, 11** and **12***
- *Moving to the next two elements 13 and 6*

*5 11 12 **13** **6***

- *Clearly, they are not sorted, thus perform swap between both*

*5 11 12 **6** **13***

- *Now, 6 is smaller than 12, hence, swap again*

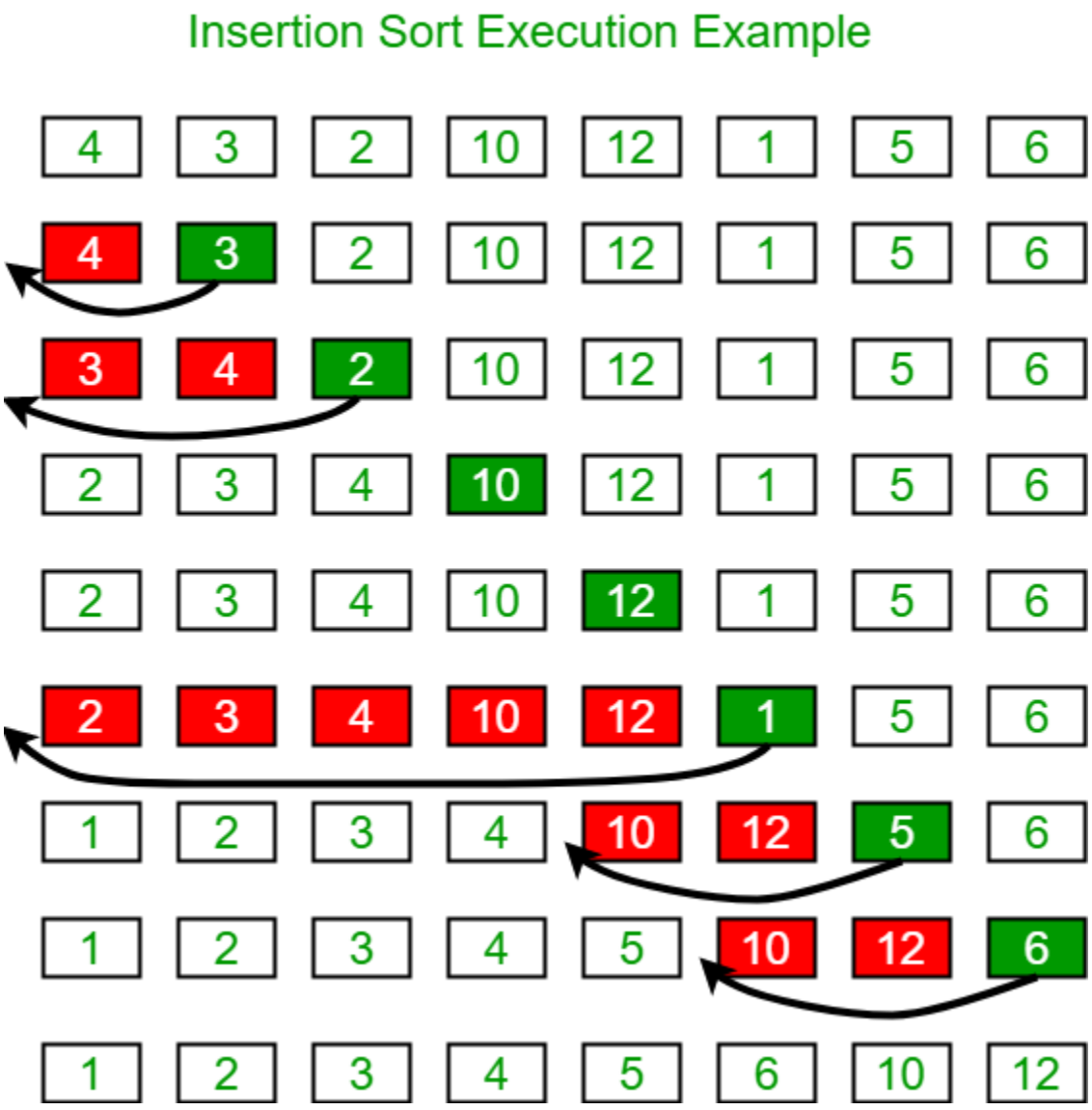
*5 11 **6** **12** 13*

- *Here, also swapping makes 11 and 6 unsorted hence, swap again*

*5 **6** **11** 12 13*

- *Finally, the array is completely sorted.*

Illustrations:



Insertion Sort Algorithm

To sort an array of size N in ascending order:

- Iterate from arr[1] to arr[N] over the array.
- Compare the current element (key) to its predecessor.
- If the key element is smaller than its predecessor, compare it to the elements before. Move the greater elements one position up to make space for the swapped element.

Below is the implementation:

- C++
- C
- Java
- Python
- C#
- PHP
- Javascript

```
// C++ program for insertion sort

#include <bits/stdc++.h>
using namespace std;

// Function to sort an array using
// insertion sort
void insertionSort(int arr[], int n)
{
    int i, key, j;
    for (i = 1; i < n; i++)
    {
        key = arr[i];
        j = i - 1;

        // Move elements of arr[0..i-1],
        // that are greater than key, to one
        // position ahead of their
        // current position
        while (j >= 0 && arr[j] > key)
```

```
        {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}

// A utility function to print an array
// of size n
void printArray(int arr[], int n)
{
    int i;
    for (i = 0; i < n; i++)
        cout << arr[i] << " ";
    cout << endl;
}

// Driver code
int main()
```

```
{
    int arr[] = { 12, 11, 13, 5, 6 };
    int N = sizeof(arr) / sizeof(arr[0]);

    insertionSort(arr, N);
    printArray(arr, N);

    return 0;
}
// This is code is contributed by rathbhupendra
```

Output
5 6 11 12 13
Time Complexity: O(N^2)
Auxiliary Space: O(1)

What are the Boundary Cases of the Insertion Sort algorithm?

Insertion sort takes maximum time to sort if elements are sorted in reverse order. And it takes minimum time (Order of n) when elements are already sorted.

What are the Algorithmic Paradigm of Insertion Sort algorithm?

Insertion Sort algorithm follows incremental approach.

Is Insertion Sort an in-place sorting algorithm?

Yes, insertion sort is an in-place sorting algorithm.

Is Insertion Sort a stable algorithm?

Yes, insertion sort is a stable sorting algorithm.

When is the Insertion Sort algorithm used?

Insertion sort is used when number of elements is small. It can also be useful when input array is almost sorted, only few elements are misplaced in complete big array.

What is Binary Insertion Sort?

We can use binary search to reduce the number of comparisons in normal insertion sort. Binary Insertion Sort uses binary search to find the proper location to insert the selected item at each iteration. In normal insertion, sorting takes O(i) (at ith iteration) in worst case. We can reduce it to O(logi) by using binary search. The algorithm, as a whole, still has a running worst case running time of O(n^2) because of the series of swaps required for each insertion. Refer this for implementation.

How to implement Insertion Sort for Linked List?

Below is simple insertion sort algorithm for linked list.