# Sorting Algorithms

Q. Why Sorting algorithms are important?

Efficient Sorting Algorithms are important for **optimizing the efficiency of other algorithms** (such as search and merge algorithms) that require input data to be in sorted lists. Sorting is also often useful for **canonicalizing data and for producing human-readable output.** Sorting has direct applications in database algorithms, divide and conquer methods, data structure algorithms, and many more.

Note: Canonicalization is the process of converting data that involves more than one representation into a standard approved format.

Q. Classify Sorting Algorithms

Sorting algorithms can be categorized based on the following parameters:

1. **Based on the Number of Swaps or Inversion**. This is the number of times the algorithm swaps elements to sort the input. Selection Sort requires the minimum number of swaps.
2. **Based on the Number of Comparisons**. This is the number of times the algorithm compares elements to sort the input. Using Big-O notation, the sorting algorithm examples listed above require at least $O(n \log n)$ comparisons in the best case and $O(n^2)$ comparisons in the worst case for most of the outputs.
3. **Based on Recursion or Non-Recursion**. Some sorting algorithms, such as Quick Sort, use recursive techniques to sort the input. Other sorting algorithms, such as Selection Sort or Insertion Sort, use non-recursive techniques. Finally, some sorting algorithms, such as Merge Sort, make use of both recursive as well as non-recursive techniques to sort the input.
4. **Based on Stability**. Sorting algorithms are said to be stable if the algorithm maintains the relative order of elements with equal keys. In other words, two equivalent elements remain in the same order in the sorted output as they were in the input.
    - Insertion sort, Merge Sort, and Bubble Sort are stable
    - Heap Sort and Quick Sort are not stable
5. **Based on Extra Space Requirement.** Sorting algorithms are said to be *inplace* if they require a constant $O(1)$ extra space for sorting.
    - Insertion sort and Quick-sort are inplace sort as we move the elements about the pivot and do not actually use a separate array which is NOT the case in merge sort where the size of the input must be allocated beforehand to store the output during the sort.
    - Merge Sort is an example of outplace sort as it requires extra memory space for its operations.

Q. Explain what is an ideal sorting algorithm?

The **Ideal Sorting Algorithm** would have the following properties:

- **Stable**: Equal keys aren't reordered.
- **Operates in place:** requiring O(1) extra space.
- Worst-case O(n log n) key comparisons.
- Worst-case O(n) swaps.
- **Adaptive**: Speeds up to O(n) when data is nearly sorted or when there are few unique keys.

There is **no** algorithm that has all of these properties, and so the choice of sorting algorithm depends on the application.

Q. What's the difference between External vs Internal sorting?

- In **internal sorting** all the data to sort is **stored in memory** at all times while sorting is in progress.
- In **external sorting** data is **stored outside memory** (like on disk) and only loaded into memory in small chunks. External sorting is usually applied in cases when data can't fit into memory entirely.

So in internal sorting, you can do something like shell sort - just access whatever array of elements you want at whatever moment you want. You can't do that in external sorting - the array is not entirely in memory, so you can't just randomly access any element in memory, and accessing it randomly on disk is usually extremely slow. The external sorting algorithm has to deal with loading and unloading chunks of data in an optimal manner.

**Divide and conquer** algorithms like **merge sort** are commonly used for external sorting because they break up the problem of sorting everything into a series of smaller sorts in chunks at a time. It doesn't require random access to the dataset and can be made to operate in chunks that fit in memory. In some cases, the in-memory chunks may be sorted using an in-memory (internal) sorting algorithm.

Q. Which sort algorithm works best on mostly sorted data?

- Only a few items: **Insertion Sort**
- Items are mostly sorted already: **Insertion Sort**
- Concerned about worst-case scenarios: **Heap Sort**
- Interested in a good average-case result: **Quick Sort**
- Items are drawn from a dense universe: **Bucket Sort**
- Desire to write as little code as possible: **Insertion Sort**

## Insertion Sort

1. Insertion sort is a **stable**, **in-place sorting algorithm** that builds the final sorted array one item at a time.
2. It is not the very best in terms of performance but is more efficient traditionally than most other simple $O(n^2)$ algorithms such as selection sort or bubble sort.
3. Insertion sort is also used in **Hybrid sort**, which combines different sorting algorithms to improve performance.
4. It is also a well-known online algorithm since it can **sort a list as it receives it**. In all other algorithms, we need all elements to be provided to the sorting algorithm before applying it. But an insertion sort allows us to start with a partial set of elements, and sort it (called a partially sorted set). If we want, we can insert more elements (these are the new set of elements that were not in memory when the sorting started) and sorts them.

5. In the real world, data to be sorted is usually not static, but rather dynamic. If even one additional element is inserted during the sorting process, other algorithms don't respond easily. But only Insertion sort algorithm is not interrupted and can respond well with the additional element.

## How insertion sort works?

The idea is to divide the array into two subsets – sorted subset and unsorted subset. Initially, a sorted subset consists of only one first element at index 0. Then for each iteration, insertion sort removes the next element from the unsorted subset, finds the location it belongs within the sorted subset and inserts it there. It repeats until no input elements remain. The following example explains it all:

i = 1 [ **3** 8 5 4 1 9 -2 ]
i = 2 [ **3 8** 5 4 1 9 -2 ]
i = 3 [ **3 5 8** 4 1 9 -2 ]
i = 4 [ **3 4 5 8** 1 9 -2 ]
i = 5 [ **1 3 4 5 8** 9 -2 ]
i = 6 [ **1 3 4 5 8 9 -2** ]
[ **-2 1 3 4 5 8 9** ]

## Insertion Sort Performance

The worst-case time complexity of insertion sort is $O(n^2)$, where n is the size of the input. The worst case happens when the array is reverse sorted.
The best-case time complexity of insertion sort is O(n). The best case happens when the array is already sorted.
The auxiliary space used by the iterative version is O(1) and O(n) by the recursive version for the call stack.

## Iterative implementation of the insertion sort algorithm

```cpp
void insertionSort(vector<int> &arr)
{
    int n = arr.size();

    // start from the second element (the element at index 0 is already sorted)
    for (int i = 1; i < n; i++)
    {
        int value = arr[i];
        int j = i;

        // find index `j` within the sorted subset `arr[0…i-1]` where element `arr[i]`
           belongs
        while ((j > 0) && (arr[j - 1] > value))
        {
            arr[j] = arr[j - 1];
            j--;
        }

        // note that the subarray `arr[j…i-1]` is shifted to the right by one position,
         i.e., `arr[j+1…i]`

        arr[j] = value;
    }
}
```

Recursive implementation of the insertion sort algorithm

```cpp
void insertionSort(vector<int> &arr,int i,int n)
{

    int value = arr[i];
    int j = i;

    // find index `j` within the sorted subset `arr[0…i-1]` where element `arr[i]` belongs
    while ((j > 0) && (arr[j - 1] > value))
    {
        arr[j] = arr[j - 1];
        j--;
    }

    arr[j] = value;

    // note that the subarray `arr[j…i-1]` is shifted to the right by one position, i.e.,
      `arr[j+1…i]`

    if (i + 1 ≤ n)
    {
        insertionSort(arr, i + 1, n);
    }
}

// start from the second element (the element at index 0 is already sorted). Calling the
recursive insertionSort function
    insertionSort(arr, 1, n - 1);
```

# Bubble Sort

1. Bubble sort is a **stable**, **in-place sorting algorithm** named for smaller or larger elements "bubble" to the **top of the list**.
2. Although the algorithm is simple, it is too slow and impractical for most problems even compared to **insertion sort**, and is not recommended for large input.
3. The only significant advantage that bubble sort has over most other implementations, even **Quicksort**, but not insertion sort, is the ability to detect if the list is already sorted. When the list is already sorted (best-case), bubble sort runs in linear time.

## How bubble sort works?

Each pass of bubble sort steps through the list to be sorted compares each pair of adjacent items and swaps them if they are in the wrong order. *At the end of each pass, the next largest element will "Bubble" up to its correct position.* These passes through the list are repeated until no swaps are needed, which indicates that the list is sorted. In the worst-case, we might end up making an n-1 pass, where n is the input size.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | ***3*** | ***5*** | ***8*** | ***4*** | ***1*** | ***9*** | ***-2*** |
| pass 1 | ***3*** | ***5*** | ***4*** | ***1*** | ***8*** | ***-2*** | ***9*** |
| pass 2 | ***3*** | ***4*** | ***1*** | ***5*** | ***-2*** | ***8*** | ***9*** |
| pass 3 | ***3*** | ***1*** | ***4*** | ***-2*** | ***5*** | ***8*** | ***9*** |
| pass 4 | ***1*** | ***3*** | ***-2*** | ***4*** | ***5*** | ***8*** | ***9*** |
| pass 5 | ***1*** | ***-2*** | ***3*** | ***4*** | ***5*** | ***8*** | ***9*** |
| pass 6 | ***-2*** | ***1*** | ***3*** | ***4*** | ***5*** | ***8*** | ***9*** |

## Bubble Sort Performance

The worst-case time complexity of bubble sort is $O(n^2)$, where n is the size of the input. The worst case happens when the array is reverse sorted. The best-case time complexity of bubble sort is O(n). The best case happens when the array is already sorted, and the algorithm is modified to stop running when the inner loop didn't do any swap.
The auxiliary space used by the iterative version is O(1) and O(n) by the recursive version for the call stack.

## Iterative implementation of the bubble sort algorithm

```cpp
void swap(vector<int>& arr, int i, int j)
{
    int temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
}

void bubbleSort(vector<int>& arr, int n)
{
    // flag to detect if the list is already sorted
    bool isSorted = true;

    // n-1 passes
    for (int k = 0; k < n - 1; k++)
    {
        // last k items are already sorted, so the inner loop can avoid looking at the
        // last k items
        for (int i = 0; i < n - 1 - k; i++)
        {
            if (arr[i] > arr[i + 1]) {
                swap(arr, i, i + 1);
                isSorted=false;
            }
        }

        // the algorithm can be terminated if the inner loop didn't do any swap
        if (isSorted)
        {
            break;
        }
    }
}
```

## Recursive implementation of the bubble sort algorithm

```cpp
void swap(vector<int>& arr, int i, int j)
{
    int temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
}

void bubbleSort(vector<int>& arr, int n)
{
    for (int i = 0; i < n - 1; i++)
    {
        if (arr[i] > arr[i + 1])
        {
            swap(arr, i, i + 1);
        }
    }

    if (n - 1 > 1)
    {
        bubbleSort(arr, n - 1);
    }
}

// Calling the recursive bubbleSort function
bubbleSort(arr, n);
```

# Selection Sort

1. Selection sort is an **stable**, **in-place sorting algorithm** known for its simplicity. It has performance advantages over more complicated algorithms in certain situations, particularly where the auxiliary memory is limited.
2. It can be implemented as a stable sort and requires $O(n^2)$ time to sort n items, making it inefficient to use on large lists. Among simple average-case $O(n^2)$ algorithms, selection sort almost always outperforms **bubble sort** and generally performs worse than **insertion sort**.
3. The biggest advantage of using a selection sort is that it does a maximum of n swaps (memory write). The insertion sort, on the other hand, does $O(n^2)$ number of writes. This can be critical if the memory-write operation is significantly more expensive than a memory-read operation, such as flash memory, where every write lessens the memory's lifespan.

## How selection sort works?

The idea is to divide the array into two subsets – sorted subset and unsorted subset. Initially, the sorted subset is empty, and the unsorted subset is the entire input list. The algorithm proceeds by finding the smallest *(or largest, depending on sorting order)* element in the unsorted subset, swapping it with the leftmost unsorted element *(putting it in sorted order)*, and moving the subset boundaries one element to the right. The following example explains it all:

| | |
|---|---|
| | 3 5 8 4 1 9 -2 |
| *i = 0* | **-2** 5 8 4 1 9 3 |
| *i = 1* | **-2 1** 8 4 5 9 3 |
| *i = 2* | **-2 1 3** 4 5 9 8 |
| *i = 3* | **-2 1 3 4** 5 9 8 |
| *i = 4* | **-2 1 3 4 5** 9 8 |
| *i = 5* | **-2 1 3 4 5 8** 9 |

## Selection Sort Performance

Both the worst-case and best-case time complexity of selection sort is $O(n^2)$, where n is the input size, and it doesn't require any extra space.

The time complexity of the selection sort recursive algorithm remains the same as the iterative version. However, the auxiliary space used by the recursive version is O(n) for the call stack.

## Iterative implementation of the selection sort algorithm

```cpp
void swap(vector<int>& arr, int i, int j)
{
    int temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
}

void selectionSort(vector<int>& arr, int n)
{
    // run n-1 times
    for (int i = 0; i < n - 1; i++)
    {
        // find the minimum element in the unsorted subarray [i…n-1] and swap it with
arr[i]
        int min = i;

        for (int j = i + 1; j < n; j++)
        {
            // if arr[j] is less, then it is the new minimum
            if (arr[j] < arr[min])
            {
                min = j;    // update the index of minimum element
            }
        }

        // swap the minimum element in subarray arr[i…n-1] with arr[i]
        swap(arr, min, i);
    }
}
```

## Recursive implementation of the selection sort algorithm

```cpp
void swap(vector<int>& arr, int i, int j)
{
    int temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
}

void selectionSort(vector<int>& arr, int i, int n)
{
    // find the minimum element in the unsorted subarray [i…n-1]
    // and swap it with arr[i]
    int min = i;
    for (int j = i + 1; j < n; j++)
    {
        // if arr[j] is less, then it is the new minimum
        if (arr[j] < arr[min])
        {
            min = j;    // update the index of minimum element
        }
    }

    // swap the minimum element in subarray arr[i…n-1] with arr[i]
    swap(arr, min, i);

    if (i + 1 < n) {
        selectionSort(arr, i + 1, n);
    }
}

// Calling the recursive selection sort function
    selectionSort(arr, 0, n);
```

# Merge Sort

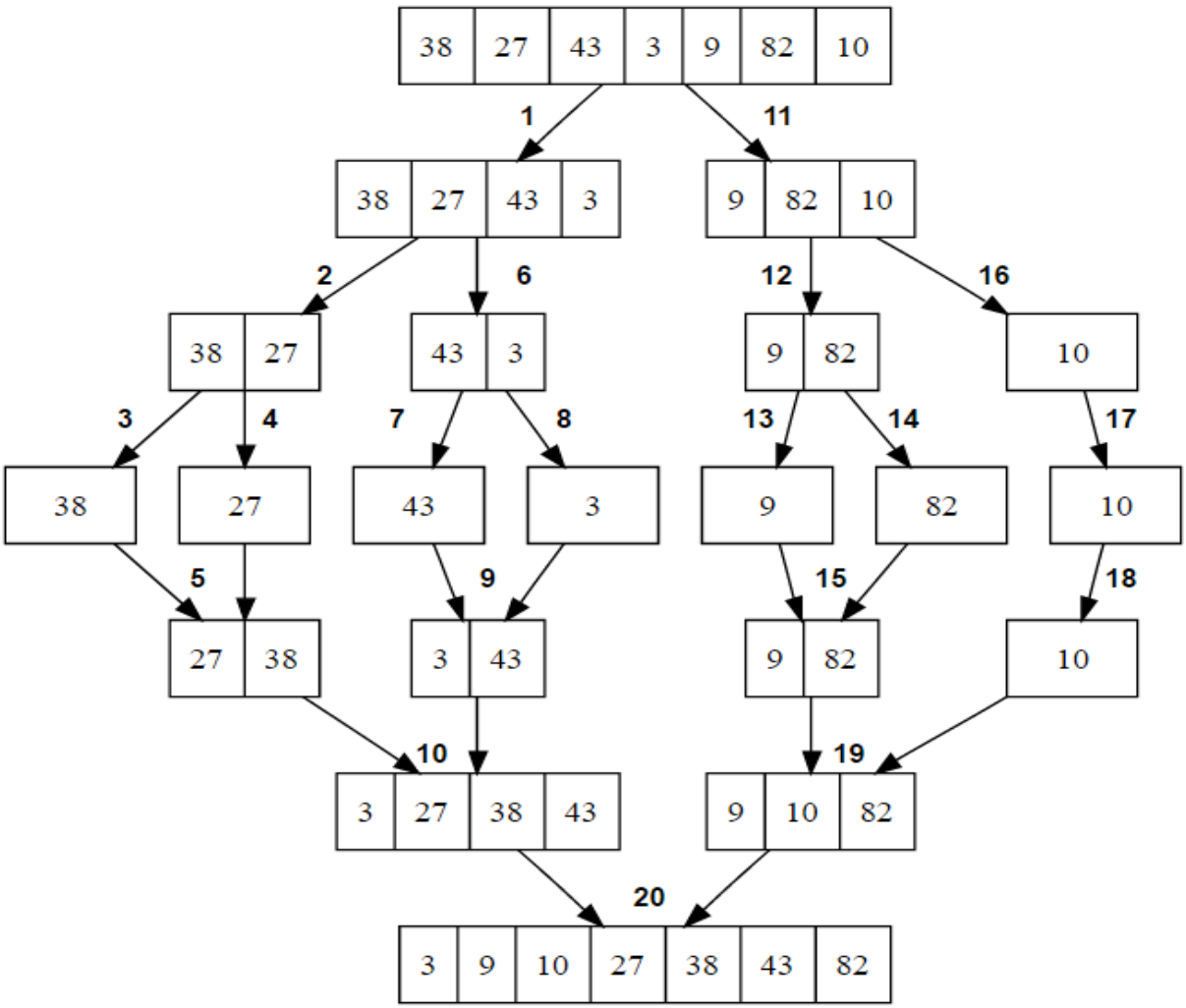
1. Merge sort is a **Divide and Conquer** algorithm. Like all divide-and-conquer algorithms, merge sort divides a large array into two smaller subarrays and then recursively sort the subarrays.
2. Merge sort is an **efficient sorting algorithm** that produces a **stable sort**, which means that if two elements have the same value, they hold the same relative position in the sorted sequence as they did in the input. In other words, the relative order of elements with equal values is preserved in the sorted sequence. Merge sort is a **comparison sort**, which means that it can sort any input for which a *less-than* relation is defined.

## How merge sort works?

Merge sort is a **Divide and Conquer** algorithm. Like all divide-and-conquer algorithms, merge sort divides a large array into two smaller subarrays and then **recursively sort the subarrays**. Basically, two steps are involved in the whole process:

1. Divide the unsorted array into n subarrays, each of size 1 (an array of size 1 is considered sorted).
2. Repeatedly merge subarrays to produce new sorted subarrays until only 1 subarray is left, which would be our sorted array.

The **following diagram** represents a **top-down** view of the recursive merge sort algorithm used to sort a 7-element integer array:



## Merge Sort Performance

The worst-case time complexity of merge sort is O(nlog(n)), where n is the size of the input.

The recurrence relation is:

$$T(n) = 2T(n/2) + cn = O(n.log(n))$$

The recurrence basically summarizes the merge sort algorithm – Sort two lists of half the original list's size and add the n steps taken to merge the resulting two lists.
The auxiliary space required by the merge sort algorithm is O(n) for the call stack.

# Recursive implementation of the Merge sort algorithm

```cpp
// Merge two sorted subarrays arr[low … mid] and arr[mid+1 … high]
void merge(vector<int>& arr,int low,int mid,int high)
{
  // aux is used to temporary store the vector obtained by merging elements from
 // [low to mid] and [mid+1 to high] in arr
    vector<int> aux; // auxillary array

    int k=low;
    int i=low;
    int j=mid+1;

    // while there are elements in the left and right runs
    while(i≤mid && j≤high)
    {
        if(arr[i]≤arr[j])
        {
            aux.push_back(arr[i]);
            i++;
        }
        else
        {
            aux.push_back(arr[j]);
            j++;
        }
    }

    // copy remaining elements
    while(i≤mid)
    {
        aux.push_back(arr[i]);
        i++;
    }

    // copy remaining elements
    while(j≤high)
    {
        aux.push_back(arr[j]);
        j++;
    }

    // copy back to the original array to reflect sorted order
    for(int i=low;i≤high;i++)
    {
        arr[i]=aux[i-low];
    }


}
void mergeSort(vector<int>& arr,int low,int high)
{
    // Base Case
    if(low≥high) // if run size≤1
    {
        return ;
    }

    // find midpoint
    int mid=low+(high-low)/2;

    // recursively split runs into two halves until run size ≤ 1, then merge them and
return up the call chain
    mergeSort(arr,low,mid);      // split/merge left half
    mergeSort(arr,mid+1,high);  // split/merge right half

    merge(arr,low,mid,high);    // merge the two half runs.
}

void mergeSort(vector<int> & arr, int n)
{
    mergeSort(arr,0,n-1);
}
```

# Quick Sort

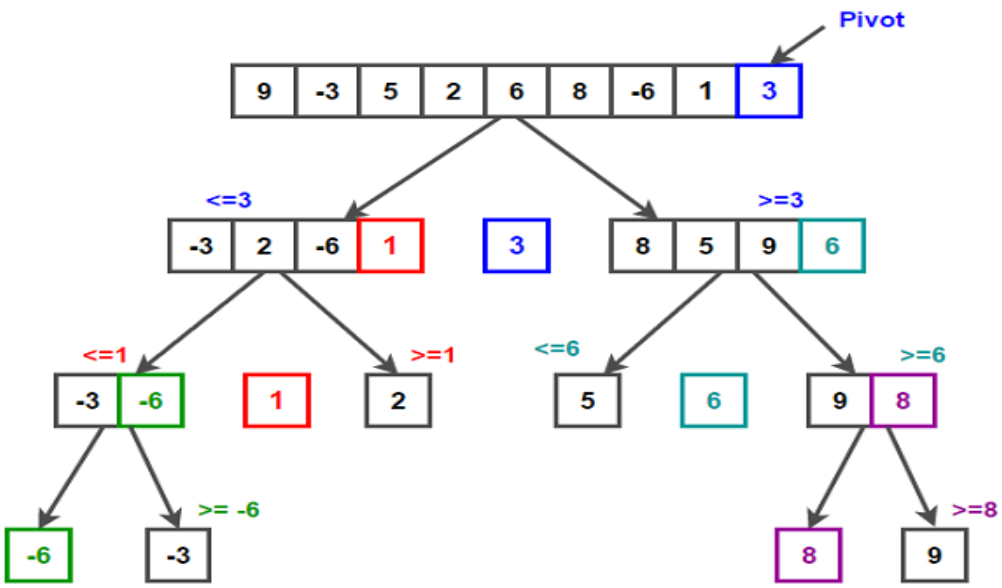
1. Quicksort is an efficient **in-place sorting algorithm**, which usually performs about two to three times faster than **merge sort** and **heapsort** when implemented well.
2. Quicksort is a comparison sort, meaning that it can sort items of any type for which a *less-than* relation is defined. In efficient implementations, it is usually not a stable sort.
3. Quicksort is a **Divide and Conquer** algorithm. Like all divide-and-conquer algorithms, it first divides a large array into two smaller subarrays and then recursively sort the subarrays.
4. Quicksort, on average, makes O(n.log(n)) comparisons to sort n items. In the worst-case, it makes $O(n^2)$ comparisons, though this behavior is very rare.

## How Quick Sort works ?

Quicksort is a **Divide and Conquer** algorithm. Like all divide-and-conquer algorithms, it first divides a large array into two smaller subarrays and then recursively sort the subarrays. Basically, three steps are involved in the whole process:

1. **Pivot selection:** Pick an element, called a pivot, from the array (usually the leftmost or the rightmost element of the partition).
2. **Partitioning:** Reorder the array such that all elements with values less than the pivot come before the pivot. In contrast, all elements with values greater than the pivot come after it. The equal values can go either way. After this partitioning, the pivot is in its final position.
3. **Recur:** Recursively apply the above steps to the subarray of elements with smaller values than the pivot and separately to the subarray of elements with greater values than the pivot.

The base case of the recursion is arrays of size 1, which never need to be sorted. The following diagram shows how we choose the leftmost element as pivot at each step of the Quicksort algorithm, partition the array across the pivot, and recur on two subarrays we get after the partition process:



# Recursive implementation of the Quick sort algorithm

```cpp
// Partition using the Lomuto partition scheme
int partition(vector<int>& arr, int start, int end)
{
    // Pick the rightmost element as a pivot from the array
    int pivot = arr[end];

    // elements less than the pivot will be pushed to the left of pIndex
    // elements more than the pivot will be pushed to the right of pIndex
    // equal elements can go either way
    int pIndex = start;

    // each time we find an element less than or equal to the pivot, pIndex
    // is incremented, and that element would be placed before the pivot.
    for (int i = start; i < end; i++)
    {
        if (arr[i] ≤ pivot)
        {
            swap(arr[i], arr[pIndex]);
            pIndex++;
        }
    }

    // swap pIndex with pivot
    swap (arr[pIndex], arr[end]);

    // return pIndex (index of the pivot element)
    return pIndex;
}

// Quicksort Function
void quicksort(vector<int>& arr, int start, int end)
{
    // Base Case
    if (start ≥ end)
    {
        return;
    }

    // rearrange elements across pivot
    int pivot = partition(arr, start, end);

    // recur on subarray containing elements that are less than the pivot
    quicksort(arr, start, pivot - 1);

    // recur on subarray containing elements that are more than the pivot
    quicksort(arr, pivot + 1, end);
}

// Calling quickSort function
    quicksort(arr, 0, n - 1);
```

## Quick Sort Performance

The worst-case time complexity of Quicksort is $O(n^2)$, where $n$ is the size of the input. The worst case happens when the pivot happens to be the smallest or largest element in the list or when all the array elements are equal. This will result in the most unbalanced partition as the pivot divides the array into two subarrays of sizes 0 and $n-1$. If this repeatedly happens in every partition (say, we have sorted array), then each recursive call processes a list of size one less than the previous list, resulting in $O(n^2)$ time.

$T(n) = T(n-1) + cn = O(n^2)$

(Note – partition takes O(n) time that accounts for $cn$)

The best-case time complexity of Quicksort is O(n.log(n)). The best case happens when the pivot divides the array into two nearly equal pieces. Now each recursive call processes a list of half the size.

$T(n) = 2\ T(n/2) + cn = O(nlog(n))$

The auxiliary space required by the Quicksort algorithm is O(n) for the call stack.

# Insertion Sort

6. Insertion sort is a **stable**, **in-place sorting algorithm** that builds the final sorted array one item at a time.
7. It is not the very best in terms of performance but is more efficient traditionally than most other simple $O(n^2)$ algorithms such as selection sort or bubble sort.
8. Insertion sort is also used in **Hybrid sort**, which combines different sorting algorithms to improve performance.
9. It is also a well-known online algorithm since it can **sort a list as it receives it**. In all other algorithms, we need all elements to be provided to the sorting algorithm before applying it. But an insertion sort allows us to start with a partial set of elements, and sort it (called a partially sorted set). If we want, we can insert more elements (these are the new set of elements that were not in memory when the sorting started) and sorts them.

10. In the real world, data to be sorted is usually not static, but rather dynamic. If even one additional element is inserted during the sorting process, other algorithms don't respond easily. But only Insertion sort algorithm is not interrupted and can respond well with the additional element.

## How insertion sort works?

The idea is to divide the array into two subsets – sorted subset and unsorted subset. Initially, a sorted subset consists of only one first element at index 0. Then for each iteration, insertion sort removes the next element from the unsorted subset, finds the location it belongs within the sorted subset and inserts it there. It repeats until no input elements remain. The following example explains it all:

i = 1 [ **3** 8 5 4 1 9 -2 ]
i = 2 [ **3 8** 5 4 1 9 -2 ]
i = 3 [ **3 5 8** 4 1 9 -2 ]
i = 4 [ **3 4 5 8** 1 9 -2 ]
i = 5 [ **1 3 4 5 8** 9 -2 ]
i = 6 [ **1 3 4 5 8 9 -2** ]
[ **-2 1 3 4 5 8 9** ]

## Insertion Sort Performance

The worst-case time complexity of insertion sort is $O(n^2)$, where n is the size of the input. The worst case happens when the array is reverse sorted.
The best-case time complexity of insertion sort is O(n). The best case happens when the array is already sorted.
The auxiliary space used by the iterative version is O(1) and O(n) by the recursive version for the call stack.