4 DECEMBER 2019 / #ALGORITHMS

Sorting Algorithms Explained with Examples in Python, Java, and C++

What is a Sorting Algorithm?

Sorting algorithms are a set of instructions that take an array or list as an input and arrange the items into a particular order.

(called lexicographical) order, and can be in ascending (A-Z, 0-9) or descending (Z-A, 9-0) order.

Why Sorting Algorithms are Important

Since sorting can often reduce the complexity of a problem, it is an important algorithm in Computer Science. These algorithms have direct applications in searching algorithms, database algorithms, divide and conquer methods, data structure algorithms, and many more.

Trade-Offs of Algorithms

When using different algorithms some questions have to be asked. How big is the collection being sorted? How much memory is at disposal to be used? Does the collection need to grow?

The answers to these questions may determine what algorithm is going to work best for the situation. Some algorithms like merge sort may need a lot of space to run, while insertion sort is not always the fastest but it doesn't require many resources to run.

You should determine what the requirements of the system are and its limitations before deciding what algorithm to use.

Some Common Sorting Algorithms

Some of the most common sorting algorithms are:

- Bubble Sort
- Insertion Sort
- Merge Sort
- Quick Sort
- Heap Sort
- Counting Sort
- Radix Sort
- Bucket Sort

But before we get into each of these, let's learn a bit more about what makes classifies a sorting algorithm.

Classification of a Sorting Algorithm

Sorting algorithms can be categorized based on the following parameters:

1. Based on 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 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(nlogn)` 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` or `Insertion Sort` , use non-recursive techniques. Finally, some sorting algorithm, 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.
5. `Insertion sort` , `Merge Sort` , and `Bubble Sort` are stable
6. `Heap Sort` and `Quick Sort` are not stable
7. Based on Extra Space Requirement Sorting algorithms are said to be `in place` if they require a constant `O(1)` extra space for sorting.
8. `Insertion sort` and `Quick-sort` are `in place` 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.
9. `Merge Sort` is an example of `out place` sort as it require extra memory space for its operations.

Best possible time complexity for any comparison based sorting

Any comparison based sorting algorithm must make at least nLog2n comparisons to sort the input array, and Heapsort and merge sort are asymptotically optimal comparison sorts. This can be easily proved by drawing a decision tree diagram.

Bucket Sort

Bucket Sort is a comparison sort algorithm that operates on elements by dividing them into different buckets and then sorting these buckets individually. Each bucket is sorted individually using a separate sorting algorithm or by applying the bucket sort algorithm recursively.

Bucket Sort is mainly useful when the input is uniformly distributed over a range.

Assume you have the following problem in front of you

You have been given a large array of floating point integers lying uniformly between the lower and upper bound. This array now needs to be sorted.

A simple way to solve this problem would be to use another sorting algorithm such as Merge sort, Heap Sort or Quick Sort. However, these algorithms guarantee a best case time complexity of O(NlogN). However, using bucket sort, the above task can be completed in O(N)time.

Let's have a closer look at it.

Consider that you need to create an array of lists, that is of buckets. Elements now need to be inserted into these buckets on the basis of their properties. Each of these buckets can then be sorted individually using Insertion Sort.

Pseudo Code for Bucket Sort:

```
{

    for(each floating integer 'x' in n)

    {
        insert x into bucket[n*x];
    }

    for(each bucket)

    {
        sort(bucket);
    }

}
```

Counting Sort

Counting Sort is a sorting technique based on keys between a specific range. It works by counting the number of objects having distinct key values (kind of hashing). Then doing some arithmetic to calculate the position of each object in the output sequence.

Example:

```
For simplicity, consider the data in the range 0 to 9.
Input data: 1, 4, 1, 2, 7, 5, 2
  1) Take a count array to store the count of each unique object
  Index:     0  1  2  3  4  5  6  7  8  9
  Count:     0  2  2  0  1  1  0  1  0  0

  2) Modify the count array such that each element at each index
  stores the sum of previous counts.
  Index:     0  1  2  3  4  5  6  7  8  9
  Count:     0  2  4  4  5  6  6  7  7  7

The modified count array indicates the position of each object in
```

```
3) Output each object from the input sequence followed by
decreasing its count by 1.
Process the input data: 1, 4, 1, 2, 7, 5, 2. Position of 1 is 2
Put data 1 at index 2 in output. Decrease count by 1 to place
next data 1 at an index 1 smaller than this index.
```

Properties

- Space complexity: O(K)
- Best case performance: O(n+K)
- Average case performance: O(n+K)
- Worst case performance: O(n+K)
- Stable: Yes (K is the number of distinct elements in the array)

Implementation in JavaScript

```
let numbers = [1, 4, 1, 2, 7, 5, 2];
let count = [];
let i, z = 0;
let max = Math.max(...numbers);
// initialize counter
for (i = 0; i <= max; i++) {
    count[i] = 0;
}
for (i=0; i < numbers.length; i++) {
    count[numbers[i]]++;
}
for (i = 0; i <= max; i++) {
    while (count[i]-- > 0) {
        numbers[z++] = i;
    }
}
```

```
    console.log(numbers[i]);
}
```

C++ Implementation

```
#include <iostream>

void countSort(int upperBound, int lowerBound, std::vector<int> nur
{
  int range = upperBound - lowerBound;                  //create a
  std::vector<int> counts (range);                      //initializ
  std::fill(counts.begin(), counts.end(), 0);           //fill vect

  for (int i = 0; i < numbersToSort.size(); i++)
  {
      int index = numbersToSort[i] - lowerBound; //For example, if
  }

  std::cout << counts << std::endl;
}
```

Swift Implementation

```
func countingSort(_ array: [Int]) {
  // Create an array to store the count of each element
  let maxElement = array.max() ?? 0
  var countArray = [Int](repeating: 0, count: Int(maxElement + 1))

  for element in array {
    countArray[element] += 1
  }
  var z = 0
  var sortedArray = [Int](repeating: 0, count: array.count)

  for index in 1 ..< countArray.count {
```

```
        sortedArray[z] = index
        z += 1
        countArray[index] -= 1
      }
    }

    print(sortedArray)
  }
```

Insertion Sort

Insertion sort is a simple sorting algorithm for a small number of elements.

Example:

In Insertion sort, you compare the `key` element with the previous elements. If the previous elements are greater than the `key` element, then you move the previous element to the next position.

Start from index 1 to size of the input array.

[8 3 5 1 4 2]

Step 1 :

```
key = 3 //starting from 1st index.

Here `key` will be compared with the previous elements.

In this case, `key` is compared with 8. since 8 > 3, move the
to the next position and insert `key` to the previous positio
```

```
-                    -
```

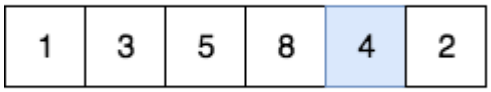
Step 2 :


```
key = 5 //2nd index

8 > 5 //move 8 to 2nd index and insert 5 to the 1st index.

Result: [ 3 5 8 1 4 2 ]
```

Step 3 :

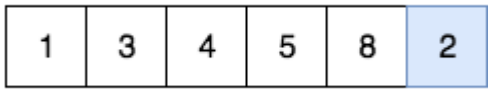

```
key = 1 //3rd index

8 > 1     => [ 3 5 1 8 4 2 ]

5 > 1     => [ 3 1 5 8 4 2 ]

3 > 1     => [ 1 3 5 8 4 2 ]
```

Step 4 :



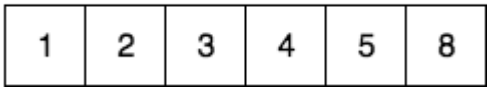

```
key = 4 //4th index

8 > 4   => [ 1 3 5 4 8 2 ]

5 > 4   => [ 1 3 4 5 8 2 ]

3 > 4   ≠>  stop

Result: [ 1 3 4 5 8 2 ]
```

Step 5 :




```
key = 2 //5th index

8 > 2   => [ 1 3 4 5 2 8 ]

5 > 2   => [ 1 3 4 2 5 8 ]
```

```
3 > 2   => [ 1 2 3 4 5 8 ]

1 > 2   ≠> stop

Result: [1 2 3 4 5 8]
```



The algorithm shown below is a slightly optimized version to avoid swapping the `key` element in every iteration. Here, the `key` element will be swapped at the end of the iteration (step).

```
InsertionSort(arr[])
  for j = 1 to arr.length
     key = arr[j]
     i = j - 1
     while i > 0 and arr[i] > key
        arr[i+1] = arr[i]
        i = i - 1
     arr[i+1] = key
```

Here is a detailed implementation in JavaScript:

```
function insertion_sort(A) {
    var len = array_length(A);
    var i = 1;
    while (i < len) {
        var x = A[i];
```

```
            A[j + 1] = A[j];
            j = j - 1;
        }
        A[j+1] = x;
        i = i + 1;
    }
}
```

A quick implementation in Swift is shown below:

```
var array = [8, 3, 5, 1, 4, 2]

func insertionSort(array:inout Array<Int>) -> Array<Int>{
    for j in 0..<array.count {
        let key = array[j]
        var i = j-1

        while (i > 0 && array[i] > key){
            array[i+1] = array[i]
            i = i-1
        }
        array[i+1] = key
    }
    return array
}
```

The Java example is shown below:

```
public int[] insertionSort(int[] arr)
      for (j = 1; j < arr.length; j++) {
         int key = arr[j]
         int i = j - 1
         while (i > 0 and arr[i] > key) {
            arr[i+1] = arr[i]
            i -= 1
```

```
        }
        return arr;
```

And in c....

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

Properties:

- Space Complexity: O(1)
- Time Complexity: O(n), O(n* n), O(n* n) for Best, Average, Worst cases respectively.
- Best Case: array is already sorted
- Average Case: array is randomly sorted
- Worst Case: array is reversely sorted.

- Stable: Yes

Heapsort

Heapsort is an efficient sorting algorithm based on the use of max/min heaps. A heap is a tree-based data structure that satisfies the heap property – that is for a max heap, the key of any node is less than or equal to the key of its parent (if it has a parent).

This property can be leveraged to access the maximum element in the heap in O(logn) time using the maxHeapify method. We perform this operation n times, each time moving the maximum element in the heap to the top of the heap and extracting it from the heap and into a sorted array. Thus, after n iterations we will have a sorted version of the input array.

The algorithm is not an in-place algorithm and would require a heap data structure to be constructed first. The algorithm is also unstable, which means when comparing objects with same key, the original ordering would not be preserved.

This algorithm runs in O(nlogn) time and O(1) additional space [O(n) including the space required to store the input data] since all operations are performed entirely in-place.

The best, worst and average case time complexity of Heapsort is O(nlogn). Although heapsort has a better worse-case complexity than quicksort, a well-implemented quicksort runs faster in practice. This is a comparison-based algorithm so it can be used for non-numerical data sets insofar as some relation (heap property) can be defined over the elements.

An implementation in Java is as shown below:

```
import java.util.Arrays;
public class Heapsort {

	public static void main(String[] args) {
		//test array
		Integer[] arr = {1, 4, 3, 2, 64, 3, 2, 4, 5, 5, 2
		String[] strarr = {"hope you find this helpful!",
		arr = heapsort(arr);
		strarr = heapsort(strarr);
		System.out.println(Arrays.toString(arr));
		System.out.println(Arrays.toString(strarr));
	}

	//O(nlogn) TIME, O(1) SPACE, NOT STABLE
	public static <E extends Comparable<E>> E[] heapsort(E[]
		int heaplength = arr.length;
		for(int i = arr.length/2; i>0;i--){
			arr = maxheapify(arr, i, heaplength);
		}

		for(int i=arr.length-1;i>=0;i--){
			E max = arr[0];
			arr[0] = arr[i];
			arr[i] = max;
			heaplength--;
			arr = maxheapify(arr, 1, heaplength);
		}

		return arr;
	}

	//Creates maxheap from array
	public static <E extends Comparable<E>> E[] maxheapify(E
		Integer left = node*2;
		Integer right = node*2+1;
		Integer largest = node;

		if(left.compareTo(heaplength) <=0 && arr[left-1]
			largest = left;
		}
```

```
                }
                if(largest != node){
                        E temp = arr[node-1];
                        arr[node-1] = arr[largest-1];
                        arr[largest-1] = temp;
                        maxheapify(arr, largest, heaplength);
                }
                return arr;
        }
}
```

Implementation in C++

```
#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 (l < n && arr[l] > arr[largest])
        largest = l;
    if (r < n && arr[r] > arr[largest])
        largest = r;
    if (largest != i)
    {
        swap(arr[i], arr[largest]);


        heapify(arr, n, largest);
    }
}


void heapSort(int arr[], int n)
{

    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);
```

```
    for (int i=n-1; i>=0; i--)
    {

        swap(arr[0], arr[i]);


        heapify(arr, i, 0);
    }
}
void printArray(int arr[], int n)
{
    for (int i=0; i<n; ++i)
        cout << arr[i] << " ";
    cout << "\n";
}
int main()
{
    int arr[] = {12, 11, 13, 5, 6, 7};
    int n = sizeof(arr)/sizeof(arr[0]);

    heapSort(arr, n);

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

Radix Sort

Prerequisite: Counting Sort

QuickSort, MergeSort, and HeapSort are comparison-based sorting algorithms. CountSort is not. It has the complexity of O(n+k), where k is the maximum element of the input array. So, if k is O(n), CountSort becomes linear sorting, which is better than comparison based sorting algorithms that have O(nlogn) time complexity.

The idea is to extend the CountSort algorithm to get a better time

The Algorithm:

For each digit i where i varies from the least significant digit to the most significant digit of a number, sort input array using countsort algorithm according to ith digit. We used count sort because it is a stable sort.

Example: Assume the input array is:

10, 21, 17, 34, 44, 11, 654, 123

Based on the algorithm, we will sort the input array according to the one's digit (least significant digit).

0: 10
1: 21 11
2:
3: 123
4: 34 44 654
5:
6:
7: 17
8:
9:

So, the array becomes 10, 21, 11, 123, 24, 44, 654, 17.

Now, we'll sort according to the ten's digit:

0:
1: 10 11 17

3: 34
4: 44
5: 654
6:
7:
8:
9:

Now, the array becomes : 10, 11, 17, 21, 123, 34, 44, 654.

Finally, we sort according to the hundred's digit (most significant digit):

0: 010 011 017 021 034 044
1: 123
2:
3:
4:
5:
6: 654
7:
8:
9:

The array becomes : 10, 11, 17, 21, 34, 44, 123, 654 which is sorted. This is how our algorithm works.

An implementation in C:

```
void countsort(int arr[],int n,int place){
```

```
        int output[n];

        for(i=0;i<n;i++)
                freq[(arr[i]/place)%range]++;

        for(i=1;i<range;i++)
                freq[i]+=freq[i-1];

        for(i=n-1;i>=0;i--){
                output[freq[(arr[i]/place)%range]-1]=arr[i];
                freq[(arr[i]/place)%range]--;
        }

        for(i=0;i<n;i++)
                arr[i]=output[i];
}

void radixsort(ll arr[],int n,int maxx){            //maxx is the

        int mul=1;
        while(maxx){
                countsort(arr,n,mul);
                mul*=10;
                maxx/=10;
        }
}
```

Selection Sort

Selection Sort is one of the simplest sorting algorithms. This algorithm gets its name from the way it iterates through the array: it selects the current smallest element, and swaps it into place.

Here's how it works:

1. Find the smallest element in the array and swap it with the first element.

second element in the array.

3. Find the third smallest element and swap wit with the third element in the array.
4. Repeat the process of finding the next smallest element and swapping it into the correct position until the entire array is sorted.

But, how would you write the code for finding the index of the second smallest value in an array?

An easy way is to notice that the smallest value has already been swapped into index 0, so the problem reduces to finding the smallest element in the array starting at index 1.

Selection sort always takes the same number of key comparisons — N(N − 1)/2.

Implementation in C/C++

The following C++ program contains an iterative as well as a recursive implementation of the Selection Sort algorithm. Both implementations are invoked in the `main()` function.

```
#include <iostream>
#include <string>
using namespace std;

template<typename T, size_t n>
void print_array(T const(&arr)[n]) {
    for (size_t i = 0; i < n; i++)
        std::cout << arr[i] << ' ';
    cout << "\n";
```

```
int minIndex(int a[], int i, int j) {
    if (i == j)
        return i;
    int k = minIndex(a, i + 1, j);
    return (a[i] < a[k]) ? i : k;
}

void recurSelectionSort(int a[], int n, int index = 0) {
    if (index == n)
        return;
    int k = minIndex(a, index, n - 1);
    if (k != index)
        swap(a[k], a[index]);
    recurSelectionSort(a, n, index + 1);
}

void iterSelectionSort(int a[], int n) {
    for (int i = 0; i < n; i++)
    {
        int min_index = i;
        int min_element = a[i];
        for (int j = i + 1; j < n; j++)
        {
            if (a[j] < min_element)
            {
                min_element = a[j];
                min_index = j;
            }
        }
        swap(a[i], a[min_index]);
    }
}

int main() {
    int recurArr[6] = { 100,35, 500, 9, 67, 20 };
    int iterArr[5] = { 25, 0, 500, 56, 98 };

    cout << "Recursive Selection Sort"  << "\n";
    print_array(recurArr); // 100 35 500 9 67 20
    recurSelectionSort(recurArr, 6);
    print_array(recurArr); // 9 20 35 67 100 500

    cout << "Iterative Selection Sort" << "\n";
    print_array(iterArr); // 25 0 500 56 98
```

```
}
```

Implementation in JavaScript

```
function selection_sort(A) {
    var len = A.length;
    for (var i = 0; i < len - 1; i = i + 1) {
        var j_min = i;
        for (var j = i + 1; j < len; j = j + 1) {
            if (A[j] < A[j_min]) {
                j_min = j;
            } else {}
        }
        if (j_min !== i) {
            swap(A, i, j_min);
        } else {}
    }
}

function swap(A, x, y) {
    var temp = A[x];
    A[x] = A[y];
    A[y] = temp;
}
```

Implementation in Python

```
def seletion_sort(arr):
         if not arr:
         return arr
    for i in range(len(arr)):
         min_i = i
         for j in range(i + 1, len(arr)):
              if arr[j] < arr[min_i]:
                  min_i = j
         arr[i], arr[min_i] = arr[min_i], arr[i]
```

Implementation in Java

```
public void selectionsort(int array[])
{
    int n = array.length;            //method to find length of ar
    for (int i = 0; i < n-1; i++)
    {
        int index = i;
        int min = array[i];          // taking the min element as 
        for (int j = i+1; j < n; j++)
        {
            if (array[j] < array[index])
            {
                index = j;
                min = array[j];
            }
        }
        int t = array[index];         //Interchange the places of 
        array[index] = array[i];
        array[i] = t;
    }
}
```

Implementation in MATLAB

```
function [sorted] = selectionSort(unsorted)
    len = length(unsorted);
    for i = 1:1:len
        minInd = i;
        for j = i+1:1:len
           if unsorted(j) < unsorted(minInd)
               minInd = j;
           end
        end
        unsorted([i minInd]) = unsorted([minInd i]);
    end
```

Properties

- Space Complexity: **O(n)**
- Time Complexity: **O(n2)**
- Sorting in Place: **Yes**
- Stable: **No**

Bubble Sort

Just like the way bubbles rise from the bottom of a glass, **bubble sort** is a simple algorithm that sorts a list, allowing either lower or higher values to bubble up to the top. The algorithm traverses a list and compares adjacent values, swapping them if they are not in the correct order.

With a worst-case complexity of O(n^2), bubble sort is very slow compared to other sorting algorithms like quicksort. The upside is that it is one of the easiest sorting algorithms to understand and code from scratch.

From technical perspective, bubble sort is reasonable for sorting small-sized arrays or specially when executing sort algorithms on computers with remarkably limited memory resources.

Example:

first two elements in the array, 4 and 2. It swaps them because 2 < 4: `[2, 4, 6, 3, 9]`

- It compares the next two values, 4 and 6. As 4 < 6, these are already in order, and the algorithm moves on: `[2, 4, 6, 3, 9]`
- The next two values are also swapped because 3 < 6: `[2, 4, 3, 6, 9]`
- The last two values, 6 and 9, are already in order, so the algorithm does not swap them.

Second pass through the list:

- 2 < 4, so there is no need to swap positions: `[2, 4, 3, 6, 9]`
- The algorithm swaps the next two values because 3 < 4: `[2, 3, 4, 6, 9]`
- No swap as 4 < 6: `[2, 3, 4, 6, 9]`
- Again, 6 < 9, so no swap occurs: `[2, 3, 4, 6, 9]`

The list is already sorted, but the bubble sort algorithm doesn't realize this. Rather, it needs to complete an entire pass through the list without swapping any values to know the list is sorted.

Third pass through the list:

- `[2, 4, 3, 6, 9] => [2, 4, 3, 6, 9]`
- `[2, 4, 3, 6, 9] => [2, 4, 3, 6, 9]`
- `[2, 4, 3, 6, 9] => [2, 4, 3, 6, 9]`
- `[2, 4, 3, 6, 9] => [2, 4, 3, 6, 9]`

Still, it's simple to wrap your head around and implement yourself.

Properties

- Space complexity: O(1)
- Best case performance: O(n)
- Average case performance: O(n*n)
- Worst case performance: O(n*n)
- Stable: Yes

Video Explanation

Bubble sort algorithm

Example in JavaScript

```
let arr = [1, 4, 7, 45, 7,43, 44, 25, 6, 4, 6, 9],
    sorted = false;

while(!sorted) {
  sorted = true;
  for(var i=0; i < arr.length; i++) {
    if(arr[i] < arr[i-1]) {
      let temp = arr[i];
      arr[i] = arr[i-1];
      arr[i-1] = temp;
      sorted = false;
    }
  }
}
```

Example in Java.

```
public class BubbleSort {
    static void sort(int[] arr) {
        int n = arr.length;
        int temp = 0;
         for(int i=0; i < n; i++){
                 for(int x=1; x < (n-i); x++){
                          if(arr[x-1] > arr[x]){
                                 temp = arr[x-1];
                                 arr[x-1] = arr[x];
                                 arr[x] = temp;
                         }

                 }
         }

    }
    public static void main(String[] args) {

                for(int i=0; i < 15; i++){
                        int arr[i] = (int)(Math.random() * 100 +
                }

                System.out.println("array before sorting\n");
                for(int i=0; i < arr.length; i++){
                        System.out.print(arr[i] + " ");
                }
                bubbleSort(arr);
                System.out.println("\n array after sorting\n");
                for(int i=0; i < arr.length; i++){
                        System.out.print(arr[i] + " ");
                }

        }
}
```

Example in C++

```
// Recursive Implementation
void bubblesort(int arr[], int n)
```

```
                return;
        bool swap_flag = false;
        for(int i=0;i<n-1;i++)  //After this pass the largest ele
        {
                if(arr[i]>arr[i+1])
                {
                        int temp=arr[i];
                        arr[i]=arr[i+1];
                        arr[i+1]=temp;
                        swap_flag = true;
                }
        }
        // IF no two elements were swapped in the loop, then retu
        if(swap_flag == false)
                return;
        bubblesort(arr,n-1);    //Recursion for remaining array
}
```

Example in Swift

```
func bubbleSort(_ inputArray: [Int]) -> [Int] {
    guard inputArray.count > 1 else { return inputArray } // make s
    var numbers = inputArray // function arguments are constant by
    for i in 0..<(numbers.count - 1) {
        for j in 0..<(numbers.count - i - 1) {
            if numbers[j] > numbers[j + 1] {
                numbers.swapAt(j, j + 1)
            }
        }
    }
    return numbers // return the sorted array
}
```

Example in Python

```
def bubbleSort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n-i-1):
                if arr[j] > arr[j+1] :
                        arr[j], arr[j+1] = arr[j+1], arr[j]
    print(arr)
```

Example in PHP

```
function bubble_sort($arr) {
    $size = count($arr)-1;
    for ($i=0; $i<$size; $i++) {
        for ($j=0; $j<$size-$i; $j++) {
            $k = $j+1;
            if ($arr[$k] < $arr[$j]) {
                // Swap elements at indices: $j, $k
                list($arr[$j], $arr[$k]) = array($arr[$k], $arr[$j
            }
        }
    }
    return $arr;// return the sorted array
}

$arr = array(1,3,2,8,5,7,4,0);
print("Before sorting");
print_r($arr);

$arr = bubble_sort($arr);
print("After sorting by using bubble sort");
print_r($arr);
```

Example in C

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

```
int main(void) {
  int arr[] = {10, 2, 3, 1, 4, 5, 8, 9, 7, 6};
  BubbleSort(arr, 10);

  for (int i = 0; i < 10; i++) {
    printf("%d", arr[i]);
  }
  return 0;
}
int BubbleSort(int array[], n)
{
for (int i = 0 ; i < n - 1; i++)
  {
    for (int j = 0 ; j < n - i - 1; j++)     //n is length of array
    {
      if (array[j] > array[j+1])      // For decreasing order use
      {
        int swap   = array[j];
        array[j]   = array[j+1];
        array[j+1] = swap;
      }
    }
  }
}
```

Quick Sort

Quick sort is an efficient divide and conquer sorting algorithm. Average case time complexity of Quick Sort is O(nlog(n)) with worst case time complexity being O(n^2) depending on the selection of the pivot element, which divides the current array into two sub arrays.

For instance, the time complexity of Quick Sort is approximately `O(nlog(n))` when the selection of pivot divides original array into two nearly equal sized sub arrays.

On the other hand, if the algorithm, which selects a pivot element of the input arrays, consistently outputs 2 sub arrays with a large difference in terms of array sizes, quick sort algorithm can achieve the worst case time complexity of O(n^2).

The steps involved in Quick Sort are:

- Choose an element to serve as a pivot, in this case, the last element of the array is the pivot.
- Partitioning: Sort the array in such a manner that all elements less than the pivot are to the left, and all elements greater than the pivot are to the right.
- Call Quicksort recursively, taking into account the previous pivot to properly subdivide the left and right arrays. (A more detailed explanation can be found in the comments below)

Example Implementations in Various Languages

Implementation in JavaScript:

```
const arr = [6, 2, 5, 3, 8, 7, 1, 4];

const quickSort = (arr, start, end) => {

  if(start < end) {

    // You can learn about how the pivot value is derived in the co
    let pivot = partition(arr, start, end);

    // Make sure to read the below comments to understand why pivot
    // These are the recursive calls to quickSort
    quickSort(arr, start, pivot - 1);
```

```
  }

const partition = (arr, start, end) => {
  let pivot = end;
  // Set i to start - 1 so that it can access the first index in th
  // Succeeding comments will expound upon the above comment
  let i = start - 1,
      j = start;

  // Increment j up to the index preceding the pivot
  while (j < pivot) {

    // If the value is greater than the pivot increment j
    if (arr[j] > arr[pivot]) {
      j++;
    }

    // When the value at arr[j] is less than the pivot:
    // increment i (arr[i] will be a value greater than arr[pivot])
    else {
      i++;
      swap(arr, j, i);
      j++;
    }

  }

  //The value at arr[i + 1] will be greater than the value of arr[p
  swap(arr, i + 1, pivot);

  //You return i + 1, as the values to the left of it are less than
  // As such, when the recursive quicksorts are called, the new sub
  return i + 1;
}

const swap = (arr, firstIndex, secondIndex) => {
  let temp = arr[firstIndex];
  arr[firstIndex] = arr[secondIndex];
  arr[secondIndex] = temp;
}

quickSort(arr, 0, arr.length - 1);
console.log(arr);
```

Implementation in C

```
#include<stdio.h>
void swap(int* a, int* b)
{
    int t = *a;
    *a = *b;
    *b = t;
}
int partition (int arr[], int low, int high)
{
    int pivot = arr[high];
    int i = (low - 1);

    for (int j = low; j <= high- 1; j++)
    {
        if (arr[j] <= pivot)
        {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}
void quickSort(int arr[], int low, int high)
{
    if (low < high)
    {
        int pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}


void printArray(int arr[], int size)
{
    int i;
    for (i=0; i < size; i++)
```

```
}


int main()
{
    int arr[] = {10, 7, 8, 9, 1, 5};
    int n = sizeof(arr)/sizeof(arr[0]);
    quickSort(arr, 0, n-1);
    printf("Sorted array: n");
    printArray(arr, n);
    return 0;
}
```

Implementation in Python3

```
import random

z=[random.randint(0,100) for i in range(0,20)]

def quicksort(z):
    if(len(z)>1):
        piv=int(len(z)/2)
        val=z[piv]
        lft=[i for i in z if i<val]
        mid=[i for i in z if i==val]
        rgt=[i for i in z if i>val]

        res=quicksort(lft)+mid+quicksort(rgt)
        return res
    else:
        return z

ans1=quicksort(z)
print(ans1)
```

Implementation in MATLAB

```
a = [9,4,7,3,8,5,1,6,2];

sorted = quicksort(a,1,length(a));

function [unsorted] =  quicksort(unsorted, low, high)
    if low < high
        [pInd, unsorted] = partition(unsorted, low, high);
        unsorted = quicksort(unsorted, low, pInd-1);
        unsorted = quicksort(unsorted, pInd+1, high);
    end

end

function [pInd, unsorted] = partition(unsorted, low, high)
    i = low-1;
    for j = low:1:high-1
        if unsorted(j) <= unsorted(high)
            i = i+1;
            unsorted([i,j]) = unsorted([j,i]);

        end
    end
    unsorted([i+1,high]) = unsorted([high,i+1]);
    pInd = i+1;

end
```

The space complexity of quick sort is `O(n)`. This is an improvement over other divide and conquer sorting algorithms, which take `O(nlog(n))` space.

Quick sort achieves this by changing the order of elements within the given array. Compare this with the merge sort algorithm which creates 2 arrays, each length `n/2`, in each function call.

However there does exist the problem of this sorting algorithm being of time `O(n*n)` if the pivot is always kept at the middle. This can be

Complexity

Best, average, worst, memory: n log(n)n log(n)n 2log(n). It's not a stable algorithm, and quicksort is usually done in-place with O(log(n)) stack space.

The space complexity of quick sort is O(n). This is an improvement over other divide and conquer sorting algorithms, which take O(n log(n)) space.

Timsort

Timsort is a fast sorting algorithm working at stable O(N log(N)) complexity.

Timsort is a blend of Insertion Sort and Mergesort. This algorithm is implemented in Java's Arrays.sort() as well as Python's sorted() and sort(). The smaller parts are sorted using Insertion Sort and are later merged together using Mergesort.

A quick implementation in Python:

```
def binary_search(the_array, item, start, end):
    if start == end:
        if the_array[start] > item:
            return start
        else:
            return start + 1
    if start > end:
        return start

    mid = round((start + end)/ 2)
```

```
            return binary_search(the_array, item, mid + 1, end)

        elif the_array[mid] > item:
            return binary_search(the_array, item, start, mid - 1)

        else:
            return mid

    """
    Insertion sort that timsort uses if the array size is small or if
    the size of the "run" is small
    """
    def insertion_sort(the_array):
        l = len(the_array)
        for index in range(1, l):
            value = the_array[index]
            pos = binary_search(the_array, value, 0, index - 1)
            the_array = the_array[:pos] + [value] + the_array[pos:inde
        return the_array

    def merge(left, right):
        """Takes two sorted lists and returns a single sorted list by c
        elements one at a time.
        [1, 2, 3, 4, 5, 6]
        """
        if not left:
            return right
        if not right:
            return left
        if left[0] < right[0]:
            return [left[0]] + merge(left[1:], right)
        return [right[0]] + merge(left, right[1:])

    def timsort(the_array):
        runs, sorted_runs = [], []
        length = len(the_array)
        new_run = [the_array[0]]

        # for every i in the range of 1 to length of array
        for i in range(1, length):
            # if i is at the end of the list
            if i == length - 1:
                new_run.append(the_array[i])
                runs.append(new_run)
```

```
            if the_array[i] < the_array[i-1]:
                # if new_run is set to None (NULL)
                if not new_run:
                    runs.append([the_array[i]])
                    new_run.append(the_array[i])
                else:
                    runs.append(new_run)
                    new_run = []
            # else if its equal to or more than
            else:
                new_run.append(the_array[i])

        # for every item in runs, append it using insertion sort
        for item in runs:
            sorted_runs.append(insertion_sort(item))

        # for every run in sorted_runs, merge them
        sorted_array = []
        for run in sorted_runs:
            sorted_array = merge(sorted_array, run)

        print(sorted_array)

    timsort([2, 3, 1, 5, 6, 7])
```

Complexity:

Tim sort has a stable Complexity of O(N log(N)) and compares really well with Quicksort. A comparison of complexities can be found on this chart.

Merge Sort

Merge Sort is a Divide and Conquer algorithm. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves. The major portion of the algorithm is given two sorted arrays, and we have to merge them into a single sorted array. The

into three steps-

- Divide the array into two halves.
- Sort the left half and the right half using the same recurring algorithm.
- Merge the sorted halves.

There is something known as the Two Finger Algorithm that helps us merge two sorted arrays together. Using this subroutine and calling the merge sort function on the array halves recursively will give us the final sorted array we are looking for.

Since this is a recursion based algorithm, we have a recurrence relation for it. A recurrence relation is simply a way of representing a problem in terms of its subproblems.

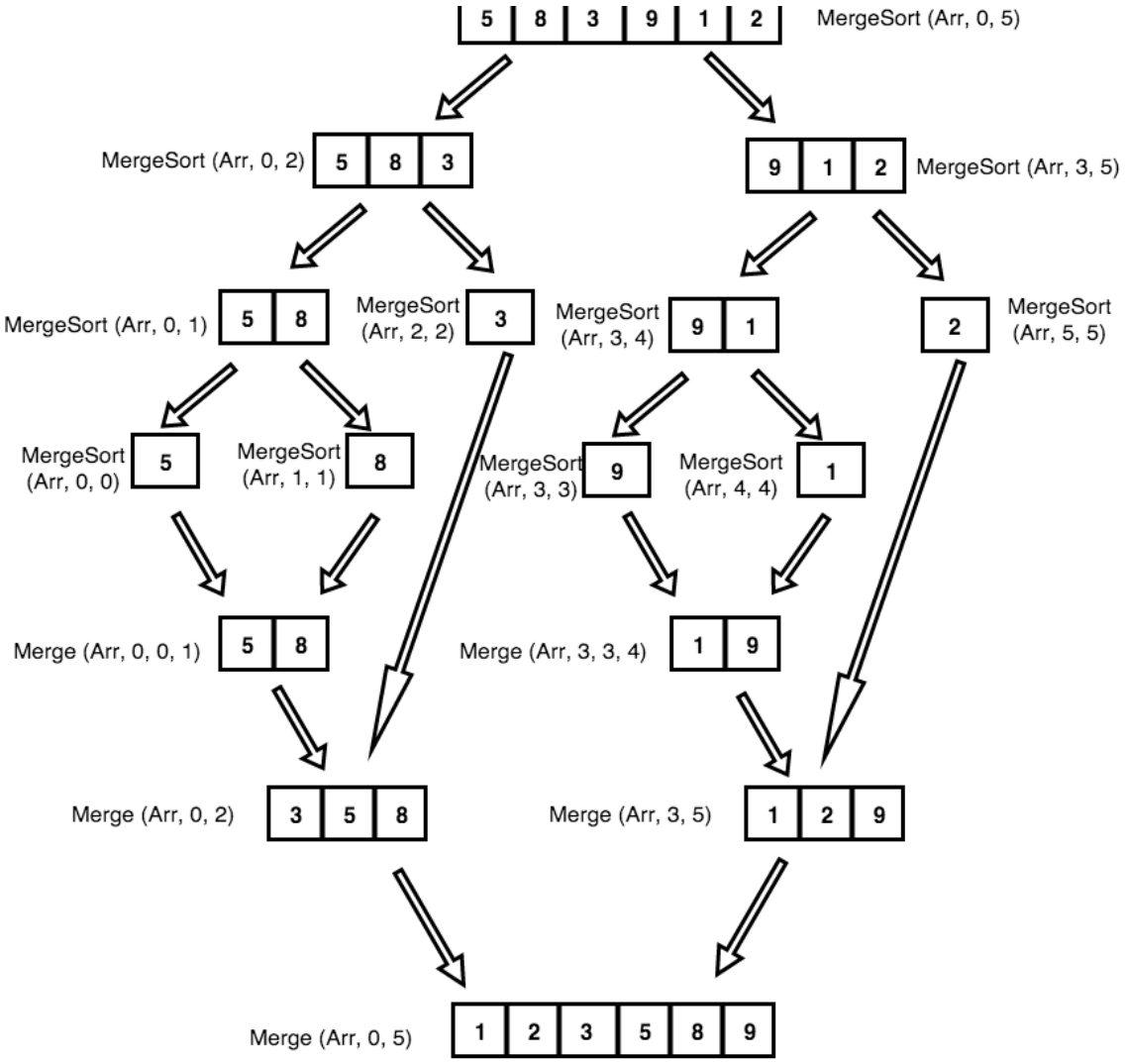
```
T(n) = 2 * T(n / 2) + O(n)
```

Putting it in plain English, we break down the subproblem into two parts at every step and we have some linear amount of work that we have to do for merging the two sorted halves together at each step.

Complexity

The biggest advantage of using Merge sort is that the time complexity is only n*log(n) to sort an entire Array. It is a lot better than n^2 running time of bubble sort or insertion sort.

Before we write code, let us understand how merge sort works with the help of a diagram.



- Initially we have an array of 6 unsorted integers Arr(5, 8, 3, 9, 1, 2)
- We split the array into two halves Arr1 = (5, 8, 3) and Arr2 = (9, 1, 2).

(3) and Arr5 = (9, 1) and Arr6 = (2)

- Again, we divide them into two halves: Arr7 = (5), Arr8 = (8), Arr9 = (9), Arr10 = (1) and Arr6 = (2)
- We will now compare the elements in these sub arrays in order to merge them.

Properties:

- Space Complexity: O(n)
- Time Complexity: O(n*log(n)). The time complexity for the Merge Sort might not be obvious from the first glance. The log(n) factor that comes in is because of the recurrence relation we have mentioned before.
- Sorting In Place: No in a typical implementation
- Stable: Yes
- Parallelizable :yes (Several parallel variants are discussed in the third edition of Cormen, Leiserson, Rivest, and Stein's Introduction to Algorithms.)

C++ Implementation

```
void merge(int array[], int left, int mid, int right)
{
    int i, j, k;

    // Size of left sublist
int size_left = mid - left + 1;

// Size of right sublist
int size_right =  right - mid;

/* create temp arrays */
```

```
/* Copy data to temp arrays L[] and R[] */
for(i = 0; i < size_left; i++)
{
    Left[i] = array[left+i];
}

for(j = 0; j < size_right; j++)
{
    Right[j] = array[mid+1+j];
}

// Merge the temp arrays back into arr[left..right]
i = 0; // Initial index of left subarray
j = 0; // Initial index of right subarray
k = left; // Initial index of merged subarray

while (i < size_left && j < size_right)
{
    if (Left[i] <= Right[j])
    {
        array[k] = Left[i];
        i++;
    }
    else
    {
        array[k] = Right[j];
        j++;
    }
    k++;
}

// Copy the remaining elements of Left[]
while (i < size_left)
{
    array[k] = Left[i];
    i++;
    k++;
}

// Copy the rest elements of R[]
while (j < size_right)
{
    array[k] = Right[j];
    j++;
```

```
}

void mergeSort(int array[], int left, int right)
{
    if(left < right)
    {
        int mid = (left+right)/2;

        // Sort first and second halves
    mergeSort(array, left, mid);
    mergeSort(array, mid+1, right);

    // Finally merge them
    merge(array, left, mid, right);
}
}
```

JavaScript Implementation

```
function mergeSort (arr) {
  if (arr.length < 2) return arr;
  var mid = Math.floor(arr.length /2);
  var subLeft = mergeSort(arr.slice(0,mid));
  var subRight = mergeSort(arr.slice(mid));
  return merge(subLeft, subRight);
}
```

First we check the length of the array. If it is 1 then we simply return the array. This would be our base case. Else, we will find out the middle value and divide the array into two halves. We will now sort both of the halves with recursive calls to MergeSort function.

```
function merge (a,b) {
    var result = [];
    while (a.length >0 && b.length >0)
```

```
}
```

When we merge the two halfs, we store the result in an auxilliary array. We will compare the starting element of left array to the starting element of right array. Whichever is lesser will be pushed into the results array and we will remove it from there respective arrays using [shift() operator. If we still end up with values in either of left or right array, we would simply concatenate it in the end of the result. Here is the sorted result:

```
var test = [5,6,7,3,1,3,15];
console.log(mergeSort(test));

>> [1, 3, 3, 5, 6, 7, 15]
```

A Merge Sort YouTube Tutorial

Here's a good YouTube video that walks through the topic in detail.

Implementaion in JS

```
const list = [23, 4, 42, 15, 16, 8, 3]

const mergeSort = (list) =>{
  if(list.length <= 1) return list;
  const middle = list.length / 2 ;
  const left = list.slice(0, middle);
  const right = list.slice(middle, list.length);
  return merge(mergeSort(left), mergeSort(right));
}

const merge = (left, right) => {
```

```
    if(left.length && right.length) {
      if(left[0] < right[0]) {
        result.push(left.shift())
      } else {
        result.push(right.shift())
      }
    } else if(left.length) {
        result.push(left.shift())
      } else {
        result.push(right.shift())
      }
    }
  return result;
}

console.log(mergeSort(list)) // [ 3, 4, 8, 15, 16, 23, 42 ]
```

Implementation in C

```
#include<stdlib.h>
#include<stdio.h>
void merge(int arr[], int l, int m, int r)
{
    int i, j, k;
    int n1 = m - l + 1;
    int n2 =  r - m;


    int L[n1], R[n2];

    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1+ j];
    i = 0;
    j = 0;
    k = l;
    while (i < n1 && j < n2)
    {
```

```
            arr[k] = L[i];
            i++;
        }
        else
        {
            arr[k] = R[j];
            j++;
        }
        k++;
    }


    while (i < n1)
    {
        arr[k] = L[i];
        i++;
        k++;
    }

    while (j < n2)
    {
        arr[k] = R[j];
        j++;
        k++;
    }
}

void mergeSort(int arr[], int l, int r)
{
    if (l < r)
    {
        int m = l+(r-l)/2;


        mergeSort(arr, l, m);
        mergeSort(arr, m+1, r);

        merge(arr, l, m, r);
    }
}
void printArray(int A[], int size)
{
    int i;
    for (i=0; i < size; i++)
```

```
}
int main()
{
    int arr[] = {12, 11, 13, 5, 6, 7};
    int arr_size = sizeof(arr)/sizeof(arr[0]);

    printf("Given array is \n");
    printArray(arr, arr_size);

    mergeSort(arr, 0, arr_size - 1);

    printf("\nSorted array is \n");
    printArray(arr, arr_size);
    return 0;
```

Implementation in C++

Let us consider array A = {2,5,7,8,9,12,13} and array B = {3,5,6,9,15} and we want array C to be in ascending order as well.

```
void mergesort(int A[],int size_a,int B[],int size_b,int C[])
{
     int token_a,token_b,token_c;
     for(token_a=0, token_b=0, token_c=0; token_a<size_a && token_b
     {
          if(A[token_a]<=B[token_b])
               C[token_c++]=A[token_a++];
          else
               C[token_c++]=B[token_b++];
      }

      if(token_a<size_a)
      {
          while(token_a<size_a)
               C[token_c++]=A[token_a++];
      }
      else
      {
```

```
        }

    }
```

Implementation in Python

```
def merge(left,right,compare):
        result = []
        i,j = 0,0
        while (i < len(left) and j < len(right)):
                if compare(left[i],right[j]):
                        result.append(left[i])
                        i += 1
                else:
                        result.append(right[j])
                        j += 1
        while (i < len(left)):
                result.append(left[i])
                i += 1
        while (j < len(right)):
                result.append(right[j])
                j += 1
        return result

def merge_sort(arr, compare = lambda x, y: x < y):
     #Used lambda function to sort array in both(increasing and dec
     #By default it sorts array in increasing order
        if len(arr) < 2:
                return arr[:]
        else:
                middle = len(arr) // 2
                left = merge_sort(arr[:middle], compare)
                right = merge_sort(arr[middle:], compare)
                return merge(left, right, compare)

arr = [2,1,4,5,3]
print(merge_sort(arr))
```

Implementation in Java

```
public class mergesort {

        public static int[] mergesort(int[] arr,int lo,int hi) {

                if(lo==hi) {
                        int[] ba=new int[1];
                        ba[0]=arr[lo];
                        return ba;
                }

                int mid=(lo+hi)/2;
                int arr1[]=mergesort(arr,lo,mid);
                int arr2[]=mergesort(arr,mid+1,hi);
                return merge(arr1,arr2);
        }

        public static int[] merge(int[] arr1,int[] arr2) {
                int i=0,j=0,k=0;
                int n=arr1.length;
                int m=arr2.length;
                int[] arr3=new int[m+n];
                while(i<n && j<m) {
                        if(arr1[i]<arr2[j]) {
                                arr3[k]=arr1[i];
                                i++;
                        }
                        else {
                                arr3[k]=arr2[j];
                                j++;
                        }
                        k++;
                }

                while(i<n) {
                        arr3[k]=arr1[i];
                        i++;
                        k++;
                }

                while(j<m) {
                        arr3[k]=arr2[j];
```

```
                }

                return arr3;

        }

        public static void main(String[] args) {
                // TODO Auto-generated method stub
                int arr[]= {2,9,8,3,6,4,10,7};
                int[] so=mergesort(arr,0,arr.length-1);
                for(int i=0;i<arr.length;i++)
                        System.out.print(so[i]+" ");
        }

}
```

Example in Java

```
public class mergesort {
  public static int[] mergesort(int[] arr, int lo, int hi) {
    if (lo == hi) {
      int[] ba = new int[1];
      ba[0] = arr[lo];
      return ba;
    }
    int mid = (lo + hi) / 2;
    int arr1[] = mergesort(arr, lo, mid);
    int arr2[] = mergesort(arr, mid + 1, hi);
    return merge(arr1, arr2);
  }

  public static int[] merge(int[] arr1, int[] arr2) {
    int i = 0, j = 0, k = 0;
    int n = arr1.length;
    int m = arr2.length;
    int[] arr3 = new int[m + n];
    while (i < n && j < m) {
      if (arr1[i] < arr2[j]) {
        arr3[k] = arr1[i];
```

```
      arr3[k] = arr2[j];
      j++;
    }
    k++;
  }
  while (i < n) {
    arr3[k] = arr1[i];
    i++;
    k++;
  }
  while (j < m) {
    arr3[k] = arr2[j];
    j++;
    k++;
  }
  return arr3;
}

public static void main(String[] args) {
  int arr[] = {2, 9, 8, 3, 6, 4, 10, 7};
  int[] so = mergesort(arr, 0, arr.length - 1);
  for (int i = 0; i < arr.length; i++)
    System.out.print(so[i] + " ");
}
}
```