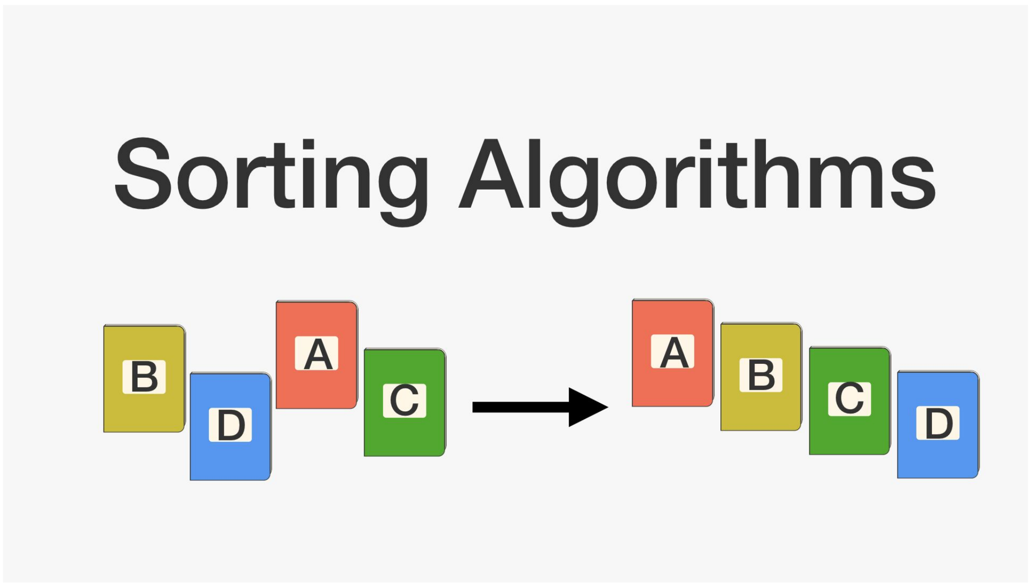
Sorting Algorithms

C++ Implementation

Morteza Rezaei

January 19, 2023



Contents

1 Bubble Sort

Bubble sort is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements and swaps them if they are in the wrong order. The pass through the list is repeated until the list is sorted.

The algorithm has a time complexity of $O(n^2)$ which means that the execution time increases quadratically with the size of the input. This is because for each element in the list, the algorithm needs to compare it with every other element.

It is called Bubble sort because the smaller elements "bubble" to the top of the list as the larger elements "sink" to the bottom. It is not an efficient algorithm for large lists and is generally not used in practice.

First alternative: Using STL swap function

```cpp

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

Second alternative: Using a temporary variable to swap

```cpp

void bubbleSort(vector<int>& nums) {
    int n = nums.size();
    for(int i = 0; i < n; ++i) {
        for(int j = 0; j < n - i - 1; ++j) {
            if(nums[j] > nums[j + 1]) {
                int tmp = nums[j];
                nums[j] = nums[j + 1];
                nums[j + 1] = tmp;
            }
        }
    }
}
```

2 Quick Sort

QuickSort is a sorting algorithm that works by selecting a "pivot" element from the array and partitioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot. The sub-arrays are then recursively sorted.

The pivot is typically chosen as the last element in the array, but it can be chosen randomly or as the median of the array. Here's an example of how it works:

1. Select a pivot element from the array (e.g. the last element)
2. Divide the elements into two sub-arrays:
   a. Elements less than the pivot
   b. Elements greater than the pivot
3. Recursively sort the sub-arrays
4. Combine the sorted sub-arrays and the pivot element to get the final sorted array

```

void quickSort(vector<int>& nums, int left, int right) {
    if (left >= right) {
        return;
    }
    int pivot = nums[right];
    int partitionIndex = left;
    for (int i = left; i < right; ++i) {
        if (nums[i] <= pivot) {
            swap(nums[i], nums[partitionIndex]);
            ++partitionIndex;
        }
    }
    swap(nums[partitionIndex], nums[right]);
    quickSort(nums, left, partitionIndex - 1);
    quickSort(nums, partitionIndex + 1, right);
}
```

The time complexity of QuickSort can be expressed as an average case of O(n * log n) and worst case of O(n^2). The average case happens when the pivot is chosen randomly or as the median of the array and the array is partitioned into roughly two equal-sized sub-arrays. The worst case is when the pivot is always chosen as the smallest or largest element, in which case, one sub-array will always be empty, and the time complexity will degrade to O(n^2).

In practice, QuickSort is usually implemented with a small modification called as "Randomized Quick-Sort" which randomly selects the pivot element and reduces the chance of worst case.

QuickSort is generally considered to be faster than other sorting algorithms like Bubble sort and insertion sort, and it is often used as a sorting algorithm in various libraries and programming languages.

3 Merge Sort

Merge sort is a sorting algorithm that works by dividing the array into two equal-sized sub-arrays, recursively sorting the sub-arrays, and then merging the sorted sub-arrays back together.

```cpp

void merge(vector<int>& nums, int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;
    vector<int> L(n1), R(n2);
    for (int i = 0; i < n1; ++i) {
        L[i] = nums[left + i];
    }
    for (int j = 0; j < n2; ++j) {
        R[j] = nums[mid + 1 + j];
    }
    int i = 0, j = 0, k = left;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            nums[k] = L[i];
            ++i;
        } else {
            nums[k] = R[j];
            ++j;
        }
        ++k;
    }

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

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

void mergeSort(vector<int>& nums, int left, int right) {
    if (left >= right) {
        return;
    }
    int mid = left + (right - left) / 2;
    mergeSort(nums, left, mid);
    mergeSort(nums, mid + 1, right);
    merge(nums, left, mid, right);
}
```

Merge sort is a sorting algorithm that uses a divide-and-conquer strategy to efficiently sort an array. The algorithm repeatedly divides the array into smaller sub-arrays, recursively sorts each sub-array, and then merges the sub-arrays back together in a specific order to create a final sorted array. Here's an example of how it works:

1. Divide the array into two equal-sized sub-arrays
2. Recursively sort the sub-arrays
3. Merge the sorted sub-arrays back together to get the final sorted array

Additionally, Merge Sort is a stable sorting algorithm, which means that it preserves the relative order of elements with equal keys. It is also more efficient than QuickSort in certain cases, such as when sorting linked lists, because it doesn't require random access to the elements.

The time complexity of Merge Sort is O(n * log n), which is the same as the time complexity of QuickSort but it is more consistent and has a guaranteed performance, it is not affected by the choices of the pivot element.

4 Selection Sort

Selection sort is a simple sorting algorithm that repeatedly selects the smallest or largest element from an unsorted portion of the array and places it at the beginning or the end of the sorted portion of the array. Here's an example of how it works:

1. Find the smallest or largest element in the array
2. Swap it with the first element
3. Repeat the process for the remaining portion of the array

```cpp

void selectionSort(vector<int>& nums) {
    int n = nums.size();
    for (int i = 0; i < n; ++i) {
        int minIndex = i;
        for (int j = i + 1; j < n; ++j) {
            if (nums[j] < nums[minIndex]) {
                minIndex = j;
            }
        }
        swap(nums[i], nums[minIndex]);
    }
}
```

The time complexity of selection sort is O(n^2) which means that the execution time increases quadratically with the size of the input. This is because for each pass through the array, the algorithm needs to check each element to find the smallest or largest element, and it needs to make n-1 passes through the array to sort n elements. It is generally not used in practice because of its poor performance, especially for large data sets.

5 Insertion Sort

Insertion sort is a simple sorting algorithm that builds the final sorted array one item at a time. It works by iterating through the array, taking each element in turn and inserting it in the correct position in the sorted portion of the array. Here's an example of how it works:

1. Start with an empty sorted portion of the array
2. Take the first element in the unsorted portion of the array
3. Compare it to the elements in the sorted portion of the array
4. Insert it in the correct position in the sorted portion of the array
5. Repeat the process for the remaining elements in the unsorted portion of the array

```

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

The time complexity of insertion sort is O(n^2) which means that the execution time increases quadratically with the size of the input. This is because for each pass through the array, the algorithm needs to check each element in the sorted portion of the array to find the correct position for the current element, and it needs to make n-1 passes through the array to sort n elements. It is generally not used in practice because of its poor performance, especially for large data sets. However, it can be efficient for small data sets or when the data is already partially sorted. It is also efficient for linked list as it doesn't require random access to elements, and it also requires less memory compared to other sorting algorithms.

6 Discussion

To compare and discuss which sorting algorithm is good for what purpose between bubble, quicksort, selection, insertion, and merge sort, we can consider several factors such as time complexity, stability, memory usage, and adaptability to different data structures.

- In terms of time complexity, quicksort and merge sort are the most efficient with O(n log n) on average, while bubble sort, selection sort, and insertion sort have a time complexity of O(n^2) which makes them less efficient for large data sets.
- In terms of stability, merge sort is a stable sorting algorithm, meaning that it preserves the relative order of elements with equal keys, while the other algorithms are not stable.
- In terms of memory usage, quicksort and merge sort have a relatively low memory usage, while bubble sort, selection sort, and insertion sort have a relatively high memory usage.
- In terms of adaptability to different data structures, insertion sort is efficient for linked lists because it does not require random access to elements, while quicksort and merge sort are efficient for arrays.

In general, quicksort and merge sort are considered as a good choice for general purpose sorting, as they have a good average time complexity and low memory usage. However, if the data is already partially sorted or if the data set is relatively small, insertion sort can be a good option as well. On the other hand, bubble sort, selection sort are not commonly used in practice because of their poor performance, they are mostly used as educational examples. It's worth noting that the choice of sorting algorithm depends on the specific requirements of the problem, and in some cases, a combination of sorting algorithms may be required to achieve the desired results.