Sorting algorithms

I. Description and running time

II. Comparison between algorithms

I. Description and running time

Algorithms used:

1) BubbleSort
2) CountingSort
3) MergeSort
4) QuickSort
5) ->RadixSort: in base 256
   ->RadixSort: in base 16
6) STL NativeSort

1) BubbleSort

-it's a comparison based sorting

Idea: repeatedly comparing pairs of adjacent elements and swapping their position if they exist in the wrong order.

-Average case time complexity: O(n^2)

-Worst case occurs when the array is reverse sorted

-Best case time complexity: when the array is already sorted

-Auxiliary space O(1)

-Stable

2) CountingSort

-**not** a comparison based sorting

Idea: 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.

-Time complexity: O(n + k), where n is the number of elements in input array and k is the range of input.

-Auxiliary space O(n + k)

-Stable

3) MergeSort

-Comparison based algorithm

Idea: divides the input array into two halves, calls itself for the two halves, and then merges the two sorted halves

-Time complexity: O(nlogn)

-Auxiliary space: O(n)

-Stable

4) **QuickSort**

-Comparison based algorithm

Idea: create two empty arrays to hold elements less than the pivot value and elements greater than the pivot value, and then recursively sort the sub arrays.

-Time complexity: O(nlogn)

-Stable

5) **RadixSort in base 256**

-**not** a comparison based algorithm

Idea: creating and distributing elements into 256 buckets according to their radix

-Time complexity: O((n+256) * log256(k)

-Auxiliary Space: O(256)

-Stable

6) **RadixSort in base 16**

-**not** a comparison based algorithm

Idea: creating and distributing elements into 16 buckets according to their radix

-Time complexity: O((n+16) * log16(k)

-Auxiliary Space: O(16)

-Stable

7) **STL NativeSort**

-Comparison based algorithm

Idea: internally uses IntroSort - implemented using hybrid of QuickSort, HeapSort and InsertionSort.By default, it uses QuickSort but if QuickSort is doing unfair partitioning and taking more than N*logN time, it switches to HeapSort and when the array size becomes really small, it switches to InsertionSort

-Time complexity: O(nlogn)

-Memory: O(logn)

-not stable

II. Comparison between algorithms

1)BubbleSort vs CountingSort

BubbleSort: ->Average complexity: O(n^2)

->Worst complexity:O(n^2)

->Best complexity:O(n^2)

CountingSort: Best, average and worst case time complexity: n+k where k is the size of count array.

Comparison: BubbleSort can have less elements than CountSort, but with larger values.

2) RadixSort vs CountingSort

RadixSort: Time complexity: O((n+256) * log256(k)

Comparison: Radix Sort can have larger values than counting sort. Also, Radix Sort because it uses an auxiliary space equal with the number of buckets (16 or 256 in our case), in contrast to the counting sort that uses an additional array of 10^5 elements, therefore is better if we have limited space. However, with less than 10^4 elements, countingsort will test better.

3) MergeSort vs QuickSort

MergeSort: Worst case complexity: worst case and average case has same complexities O(n log n)

QuickSort: Worst case complexity: O(n2) as there is need of lot of comparisons in the worst condition.

Comparison: Merge sort is more efficient and works faster than quick sort in case of larger array size or datasets, whereas quick sort is more efficient and works faster than merge sort in case of smaller array size or datasets.