CS 202 - Data Structures

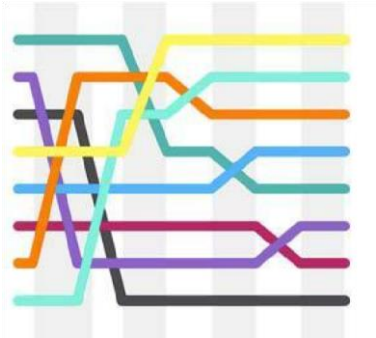
Assignment-4

Sorting Algorithms

Due Date: ***18 April 2019, 11 pm***

This assignment can be submitted till 11pm on Sunday, 21st April, 2018 with a 10% penalty per day.

*Your code must run on the **Mars** server!*



In this assignment, you will implement and evaluate five sorting algorithms.

The course policy about plagiarism is as follows:

1. Students must not share actual program code with other students.
2. Students must be prepared to explain any program code they submit.
3. Students must indicate with their submission any assistance received.
4. All submissions are subject to plagiarism detection.
5. Students cannot copy code from the Internet.

Students are strongly advised that any act of plagiarism will be reported to the Disciplinary Committee.

Note:

This is a long assignment and may take more time than you expect, so start early. There would be no extensions.

This assignment consists of six major tasks, the first one involves implementing the heap data structure. Tasks 2-5 involve the implementation of different sorting algorithms while the last one pertains to the application of one of the sorting algorithms. For tasks 2 to 5, we have provided some code in generator.cpp, which generates a sequence of numbers between 1 and n. Based on user input, the sequence will be random, sorted, reverse sorted or almost sorted. For each task, you have to sort this input. You will be required to sort these numbers in arrays, as well as lists for some of the algorithms. Make sure that your algorithm works for negative values as well.

The header file sorts.h has declarations for all the sort functions. You must implement all of these functions in sorts.cpp. You may also define additional functions if it aids you in any way. But you must implement the functions already declared in sorts.h. The generator.cpp file contains functions that generate random input cases so that you can observe your implementation. You are also given a standard implementation of the Linked List data structure in the list.h and list.cpp files. You must write all your code in sorts.cpp. No other file should be altered. After writing your code in sorts.cpp, just compile generator.cpp and run. For final testing you are provided individual test cases for each sorting algorithm.

Every function that you must implement should accept an integer vector as input. The implementations must internally duplicate the vector into an array or linked list (as mandated by each task). Once the sorting is complete, the numbers must be put back into a vector in sorted order and that vector should be returned e.g., for Task 2, in InsertionSort(), you should take all elements present in the input

vector, put them into an array and then apply the insertion sort algorithm. After sorting, put all the elements from the array into a vector (you can overwrite into the old vector or store in a new vector) and then return that.

Task 1: Heap Data Structure

In this task, you are expected to implement the heap data structure. You are provided the heap.h, heap.cpp and heap_test.cpp files. You are only required to make changes to the "heap.cpp" file and you are expected to implement all the methods in that file. You can test your implementation using the heap_test.cpp file. You will be using this implementation of heap data structure in task 5.

Task 2: Insertion Sort (array based)

For this task, implement the in-place Insertion Sort algorithm using an array.

Task 3: Mergesort (using linked lists):

In this task, you need to implement the Merge sort algorithm using a linked list

Task 4: Quicksort (both array and linked lists)

Implement both the in-place array-based as well as the linked list method of Quicksort.

I. For the array-based implementation, use the following strategies for selecting a pivot:
    a. First element of the array as the pivot
    b. Then use the median-of-three pivot

c. Finally, use the last element of the array as pivot Use the strategy with lowest running time in your test.

II. For the linked list based implementation, use a random pivot.

Essentially this task includes two sub-tasks:

(i) Array based implementation – using the above mentioned techniques

(ii) Linked List implementation – using a random pivot.

Task 5: Heapsort (using heaps)

For this task, implement the array based Heapsort algorithm using the heap implementation in task 1.

Task 6: Queen of all sorts

Your classmate in CS202 claims that he has invented a sorting algorithm **Queen of all sorts** that is much faster compared to the algorithms taught in class. He claims that his algorithm can sort a list of integers in O(list size). He also claims that his algorithm if modified a little can maintains all the original duplicate values. Steps for his algorithm are given below.

He creates a very large array **arr** of some size **k**;

He iterates over the list to be sorted and put each list element **p** at index **p** in array **arr.**

If **p** is greater than the size of **arr** he creates another array **arr'** of size **p+1,** Copy all the elements in new array and put the element **p** at index **p** in new array**.**

After the complete iteration on the list he iterates over **arr'** from left to right and put each element that appears in **arr'** in a list queen_of_all_sorts.

He claims that this list is sorted and takes O(n) time .

Your tasks:
Is his claim that his algorithm runs in O(n) true? Explain. If not what is the original time complexity in terms of Big-O?

If it is not true how will you modify the algorithm so that it actually produces a sorted list.

Explain how will you modify it for handling duplicates? If his claim is not true then explain why?

Your classmate presented his sorting algorithm to Dr.Ihsan. He points out that choice of the array **arr**(in the above part) size is very inefficient**. Note: All other problems in above part are also present.**
You are required to implement a fully modified version of his algorithm containing these Properties.

- Your Algorithm produces a sorted list
- Sorted list contains duplicates as given in input list.
- It has much better implementation for choice of array size for **arr** and **arr'.**

The following test cases require flags:
g++ test_insertion_sort.cpp -pthread –std=c++11 g++ test_merge_sort.cpp -pthread –std=c++11 g++ test_quickarray_sort.cpp -pthread –std=c++11 g++ test_quicklist_sort.cpp -pthread –std=c++11 g++ test_heap_sort.cpp -pthread –std=c++11 test_queen_all_of_sorts.cpp -pthread –std=c++11 test2_queen_of_all_sorts.cpp -pthread –std=c++11

BEST OF LUCK