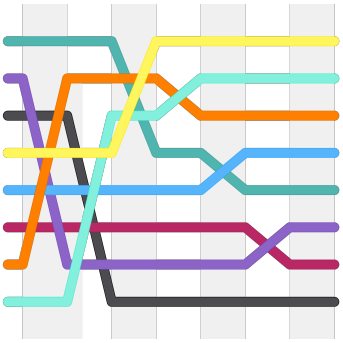
Data Structures

Assignment-4

Sorting Algorithms

Due Date: 11pm on Wednesday, 13th April, 2016

- *This assignment can be submitted till 11pm on Saturday, April 16th, 2016 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.

Introduction

This assignment consists of six major tasks, each involving the implementation of a different sorting algorithm. We have provided some starter code, 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 and compare the sorting times of each of the five sorting algorithms. You will be required to sort these numbers in arrays, as well as lists for some of the algorithms.

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 for testing as well as the main that you will use to run your sorts. 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.

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 1, 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.

For each task, you will record the time required to sort the input using the time recording routines provided in the starter code. To handle various sources of randomness in such an experimental study, you are required to run the same sorting routine multiple times using different sets of input generated by the input generator.

Task 1 (Insertion Sort – array based):

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

Task 2 (Mergesort – using linked lists):

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

Task 3 (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

   Compare the running times of the above strategies individually on different input sizes.

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 4 (Heapsort – using heaps):

For this task, implement the array based Heapsort algorithm using heaps and determine the time it takes for sorting different inputs.

Task 5 (Bucketsort – using hashtables):

Implement the Bucketsort algorithm using Hashtables. For this algorithm, the maximum key (i.e., hashtable size) will be passed as a function parameter.

Task 6 (Smart Search):

You are given an unsorted vector that contains N numbers. You are also given a number k. You have to find all pairs of numbers in the vector, which add up to k. For example, given [3, 4, 1, 2, 5] and k=7, you would output:

```
2, 5
3, 4
4, 3
5, 2
```

The trick here, however, is that you need to do it in O(N log N) time. Zero marks would be given for doing this in $O(N^2)$ time. Also print the time taken to do this. Implement this task in "smartSearch.cpp".

Task 7 (Mixed Sort):

Based on your knowledge of sorting algorithms (Insertion Sort, Mergesort, Heapsort, and Quicksort), design an interface which is capable of deciding which of these sorting algorithms to use based on the input size and the initial condition of the array (i.e., whether it is sorted, random, reverse sorted, or almost sorted). The requirements are as follows:

- Design an interface that asks the user for the input size
  - *Input sizes could be 10, 10000, or 100000 only*
- The interface should also ask for the initial condition of the array
  - *Initial conditions could be sorted, random, reverse sorted, or almost sorted*
- Based on these input parameters, your code must be able to decide which sorting algorithm to use.
  - *You must be able to justify your choice of algorithm in the vivas.*
- It should display which sorting algorithm was used, the sorted array and then display the time it took to execute.
- For all combinations of input sizes and the initial conditions of the input (i.e., 12 combinations), save the total execution time as well as the sorting algorithm that was used for each combination. Note that one combination would correspond to (input size, initial condition of the

array) e.g., (10000, almost sorted).

- Now for all the 12 combinations, run Insertion Sort, Mergesort, Heapsort, and Quicksort individually and plot the execution times along with the execution time of Mixed Sort.
    - You need to submit the plot along with your code.

BEST OF LUCK