CS293 Lab 4
A Game of Trees

General Instructions

- **There is no in-lab and out-lab component of this assignment.**
- **Your final solution is due by Mon, Aug 29, 2pm on Moodle. Please refer to submission instructions to know what EXACTLY to submit.**
- **You must however submit your partial work by 5.15 pm on Fri, Aug 26 on Moodle.**
- **Those who don't submit their partial work by the above deadline will not have their final submissions evaluated.**
- **Please go through the announcement made on Moodle in this regard.**

Introduction

Remember you implemented BSTs in the previous lab, and immediately after, studied AVL trees, which have the additional feature of balance. Is there any algorithm which also tries to divide an array into 2^1 (ideally equal) parts? Why, it's quicksort. There's an interesting albeit unique connection between quicksort and binary search trees that we hope to explore in this lab. With the context clear, let's finally come to the problem statement.

Problem Statement

The goal is to implement Quicksort for an array of objects, and see an interesting connection between (a) the sequence of pivots used in Quicksort and (b) the binary search tree (BST) obtained by inserting the pivots in sequence.

As before, we will use journey records as our basic objects. Each such record has two non-negative integer fields: a journey code and a price. You may think of the journey code as uniquely identifying the source station, destination station, train number, start time and end time. Therefore, we will not store source station, destination station etc. explicitly for now. A Journey has simply been abstracted as a (code, price) pair.

Quicksort

Given an array of journey records, our first task is to sort it with respect to journey codes using Quicksort. We have provided a skeleton code to help you get started with this part. You should flesh out this skeleton code to arrive at your implementation.

[1] can be generalised to k

Recall from CS 213 lectures that an invocation of quicksort finds a pivot element, partitions the given array into two parts based on which array entries are "less than" or "greater than equal to" this pivot, and then recursively invokes quicksort on each of these two parts.

We have provided blackbox (hidden) implementations of a few alternative functions for choosing a pivot index from a given array of journey records, and the skeleton code uses one of these functions based on an input parameter. **Please do not change this**.

We have also provided an implementation of the comparison operator "<" for journey records that indicates whether the journey code of one journey record is less than that of another journey record. Every time such a comparison is made, it counts as one unit of cost. The given implementation of "<" automatically counts the total cost of comparisons. **Please do not change this**.

Your implementation of Quicksort must sort the given array of journey records and also print the total cost of comparisons. A function to print an array of journey records has been provided. Please use this to test whether your implementation of Quicksort is really generating an array of journey records sorted by journey code.

Note the variation in total cost of comparisons as different functions are used to choose the pivot, even though the array being sorted remains the same. Why do you think this variation happens?

Making the Tree

In this part, we will augment the Quicksort implementation of part 1 with an extra functionality. Every time a pivot index is generated, you must create a copy of the journey record at that index in the array, and insert this copy of the record in a binary search tree (BST). You must start with an empty BST, and keep inserting the pivot elements into this tree until the last time a pivot index is generated.

You are strongly encouraged to reuse code from lab assignment 3 for constructing and inserting journey records in your BST.

After the entire BST is constructed, you can print the BST using the printBST function that was provided in lab assignment 3.

Once the entire BST is constructed, you MUST also print the difference between the longest path from the root to a leaf and the shortest path from the root to a leaf in the resulting BST. This is a measure of the "imbalance" of path lengths in the BST.

A simple way to keep track of the "imbalance" is to add extra fields, say shortestPathLength and longestPathLength, to each tree node in the BST. These fields can be used to store the shortest and longest path lengths from this node to a leaf in the sub-tree rooted at this node.

Please keep in mind that whenever a node is inserted in the BST, we may need to update shortestPathLength and longestPathLength of several nodes in the BST.

Do you see any relation between the total cost of comparisons in Quicksort and the imbalance in the resulting BST?

Generalising to k-parts (Optional)

This part is COMPLETELY OPTIONAL: Try only if you are seeking bonus marks.
The usual Quicksort algorithm uses a pivot to partition a given array into two parts. Implement a version of Quicksort that uses k pivots (where k is a new input parameter to Quicksort) to partition a given array into k+1 parts as follows. Suppose the k pivots in "increasing order" (as used for sorting) are $p_1, p_2, \ldots p_k$. Then the k+1 partitions are those records that are "less than" p1, those that are "greater than" p_1 but "less than" p_2, those that are "greater than" p_2 but "less than" p_3, and so on. Use the same idea of counting comparison costs as done in Part 1.

For a pivot choosing strategy that randomly chooses k elements as pivots, can you experiment and determine what values of k give the smallest counting comparison costs for a given large array (say, count of elements in array 10^6.

Note that the template code provided is for the compulsory parts. You will need to ideate on your own to achieve this part. (Though no extra files are required)

Testing the Code

- You have taken input from a file in the previous lab. Similar such methods will be used this time for feeding input into your code
- Use the printing functions (or make your own) to print intermediate results as described in detail above
- For printing the tree, you can use the printing method used in lab 3 or optionally use gdb along with its python API for a beautiful rendering of a tree.
- Some Running commands that may be useful

```
g++ -c makeTree.cpp, g++ choose.o makeTree.o
```

Submission Instructions

Make the necessary changes in the files.
Keep all the files in a folder named $< ROLL\_NUMBER >\_L4$ and compress it to a tar file named $< ROLL\_NUMBER >\_L4$.tar.gz using the command

```
tar -zcvf <ROLL_NUMBER>_L4.tar.gz <ROLL_NUMBER>_L4
```

Submit the tar file on Moodle. The directory structure should be -
$< ROLL\_NUMBER >\_L4$
— - - - - (all files which were present initially)
If your Roll number has alphabets, they should be in "small" letters.