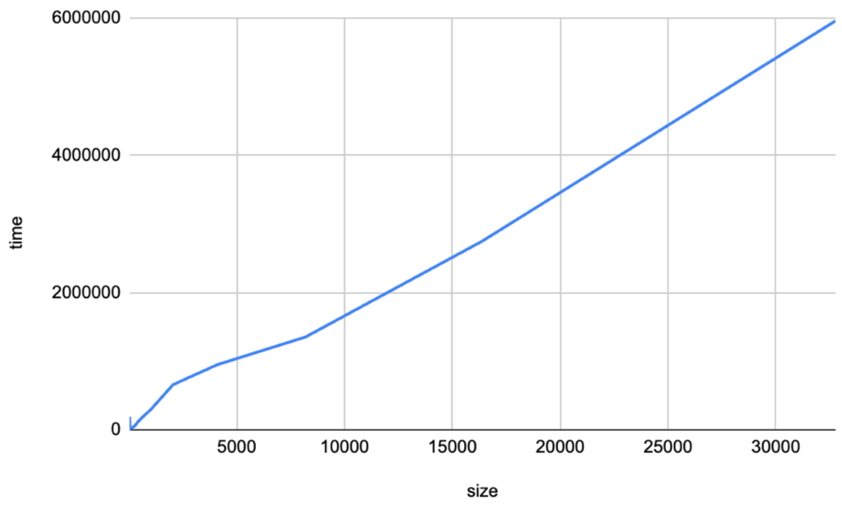
Analysis Document on Assignment09

Tailang (Terry) CAO u1480633

Graphs attached in this document are generated based on the average result of 100 iterations.

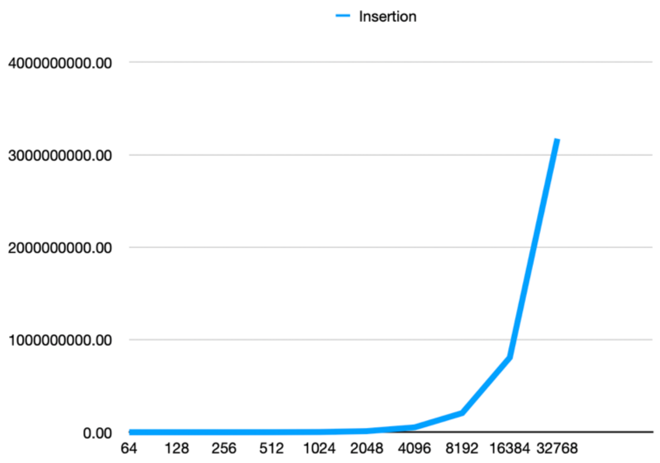
I. Construction

i. Building Tree with **buildTree(ArrayList<Segment> segments)**



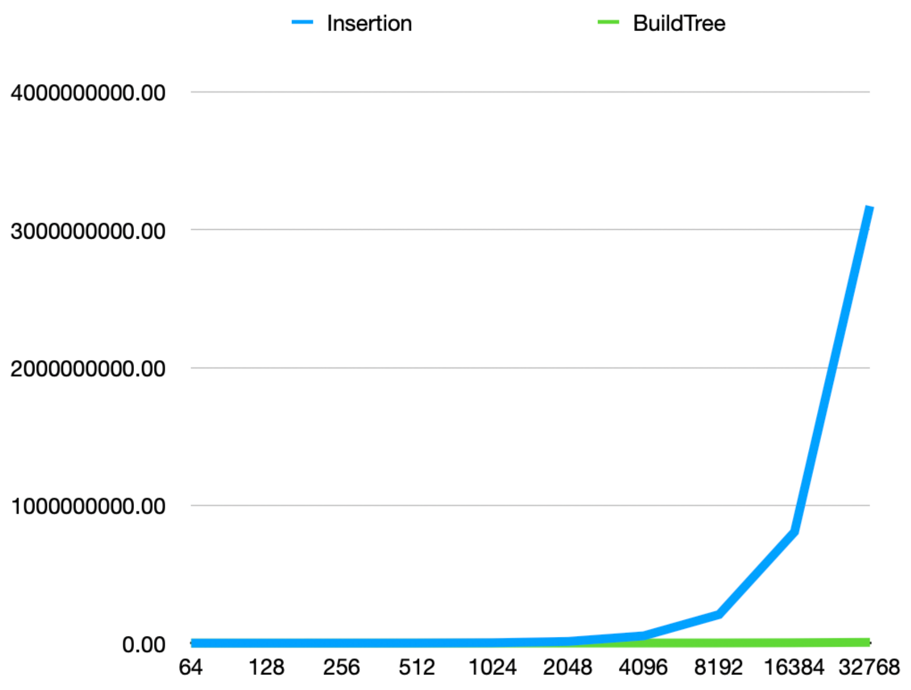
An ArrayList of segments with from 2^5 to 2^15 was used as the input for building the tree. The run time appears to be in a straight line, the maximum runtime is at 10^7.

ii. Building tree with **insert(Segment segment)**



The same ArrayList of segments was used as the input for building the tree. The runtime illustration appears to be a reversed projection, with a much larger runtime magnitude of 10^10.

iii. Runtime Comparison



By putting the two graphs together it is easy to find that the runtime of building the tree via insertion is significantly larger comparing with building the tree via the BuildTree method.
This is believed to be the result of the choose of root in the Build Tree method. In the BuildTree method, the root is chosen by picking a random index of segment at each time of recursion. This largely lowers the possibility of generating a highly unbalanced tree.
In the insertion case, the root is always going to be the first inserted segment. This result in the problem that if the input ArrayList of segments is already (or partially) in order, the generated tree is going to be highly imbalanced.

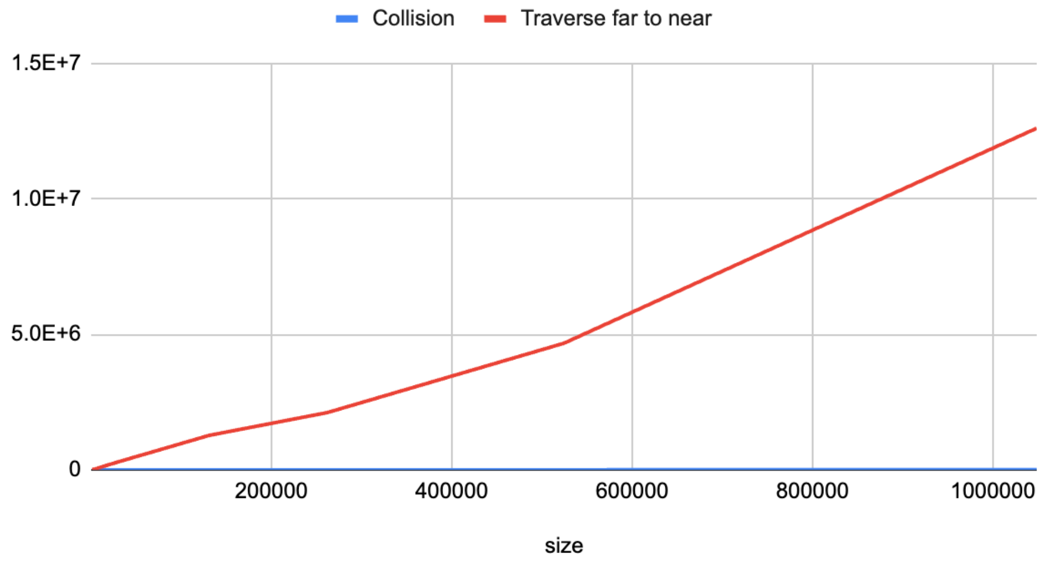
Therefore, the buildTree function outstands the insertion by choosing random nodes as the root in each recursion, effectively lower the appearance of worst case.

iv. Worst Case

The worst case of constructing a tree is if the input segments are already in ascending or descending order which result in a highly unbalanced tree which only has one side. The experiment did match the Big O growth rates I expected, with the insertion being O(N^2) and the buildTree being O(N).

II. Collision Detection

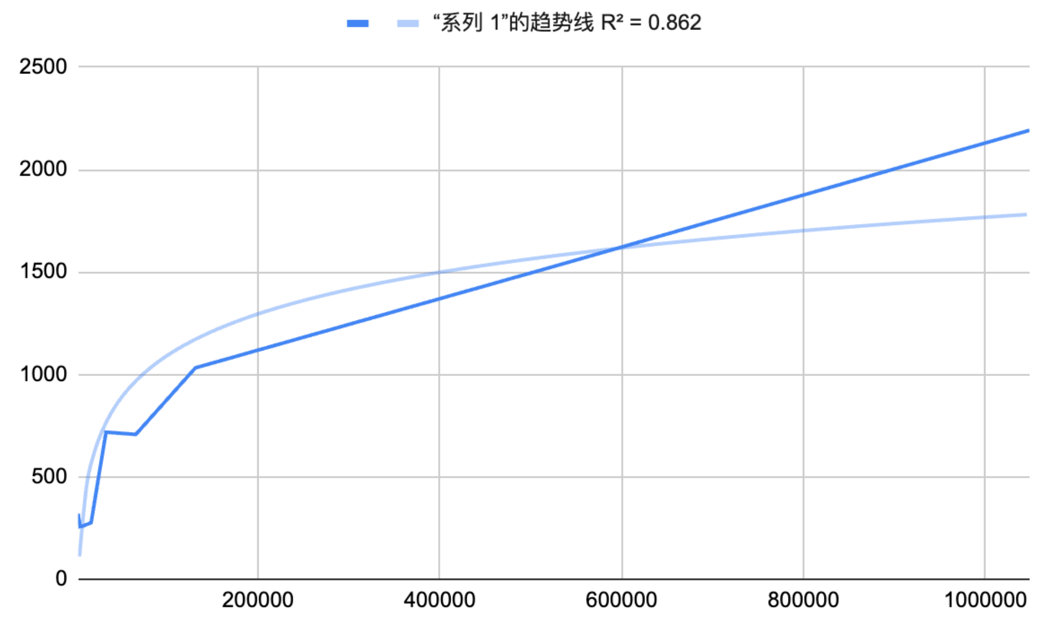
i. Graph Comparison



According to the graph, it is obvious that the collision method I implemented is a lot more effective than the traverse from far to near method which will visit all nodes.

In the collision method I implemented, the tree is split into half each time finding a collision, this can significantly reduce the time complexity of the code to O(logN).

Below is the plot for the collision method I implemented only.



It can be seen that the graph has a R^2 value of 86.2% comparing to a logN graph, which shows that the experiment meets the big O that I expected.

ii. Experiment Design

For the timing experiment of collisions, the data is chose as the average result of 1000 iteration times. The size of the ArrayList of segments is between 2^10 to 2^20. The tree used in the experiment is built through the buildTree method that takes in the ArrayList of segments.

In order to get the runtime of the two collision detection method, I initiated 2 variables to store the corresponding runtime of each method. They each record the average runtime of one collision method over 1000 iterations by taking the average of each starting runtime and ending runtime. I then print out the average runtime of each array list size and make them into the up mentioned plots.