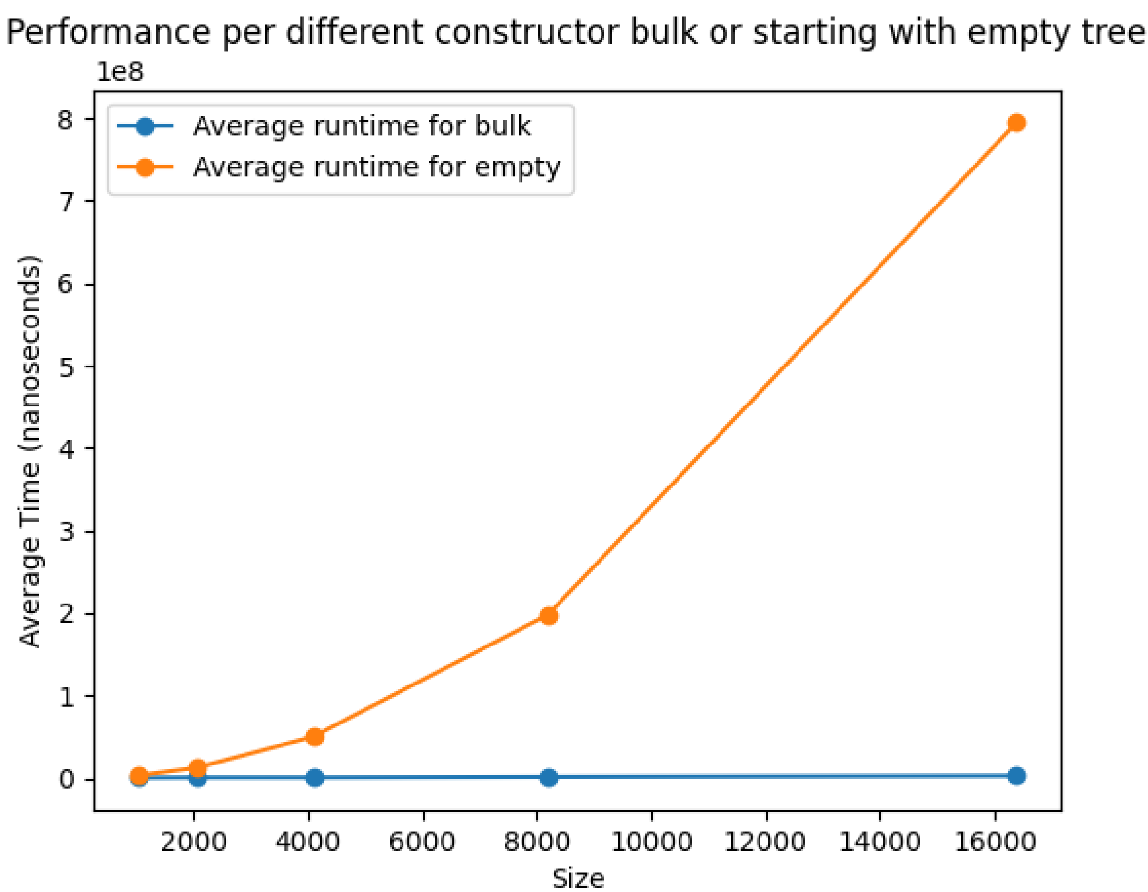
Construction Experiment

Perform an experiment to compare the time to construct a tree with N segments using a worst case input using the "bulk construction" constructor compared with inserting the segments one at a time into an initially empty tree. As an example, consider a bunch of vertical segments. What is the "worst case" order to insert them? Plot the runtimes of the 2 methods for building the tree. Do your experiments match the Big O growth rates you expected?



The worst case order for setting segments into a BSP tree involves adding segments in sequence that leads to an unbalanced tree where the segments are inserted in a sorted manner and the tree grows uniformly which leads to a heavy skewed tree. By using the worst case for both different constructors where the bulk constructor suggests a time complexity between linear and logarithmic; closer to O (n log n).

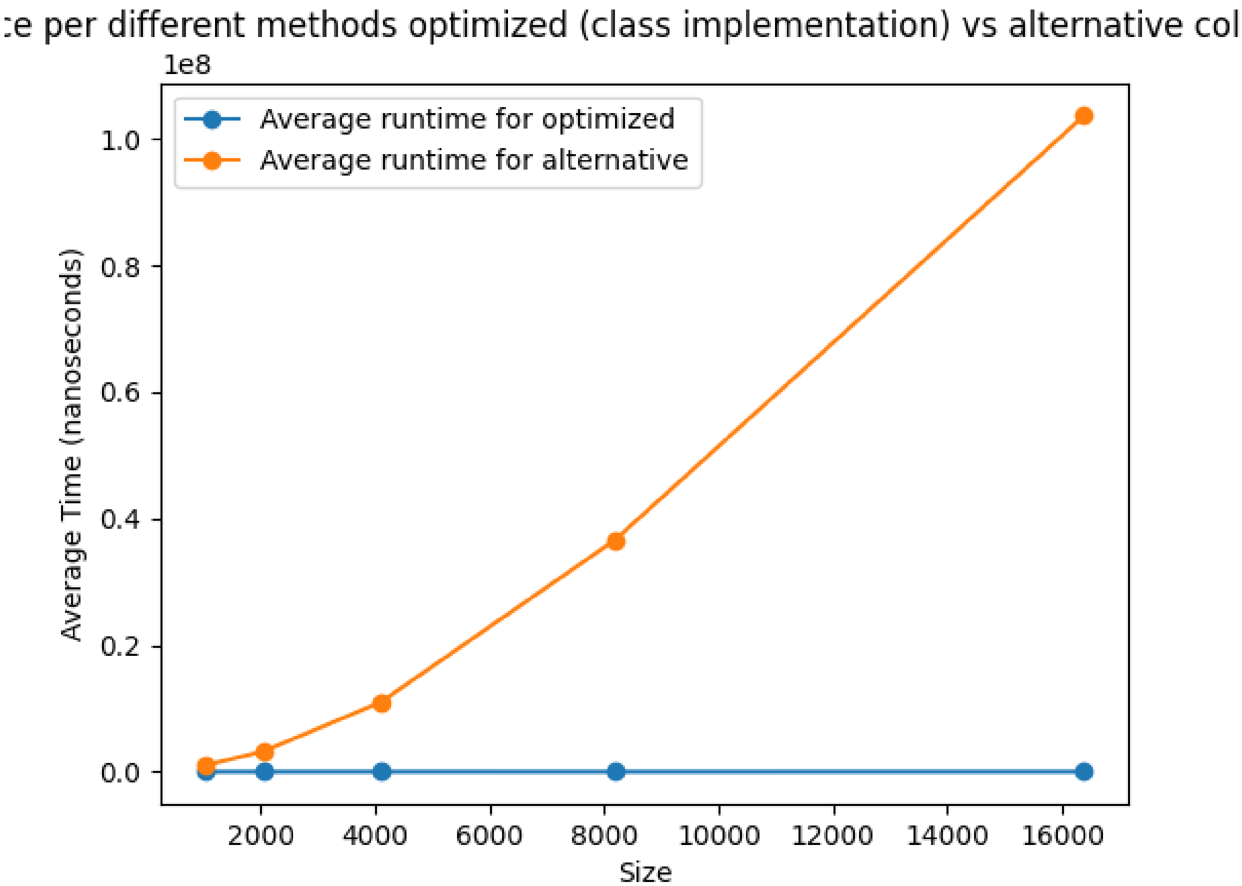
On the other hand, using the empty constructor where one segment is added at time suggests a growth rate between O (n) and O (n^2). In real-world scenarios may not perfectly match theoretical expectations but the expected run time to build a tree from scratch is O(n) and for the bulk constructor it was O (n log n) due to its randomness of segment selection and subsequent splitting logic where each recursive call a segment is randomly selected and the list of remaining segments is split in two sublists based in its relationship to the selected segment.

Collision Experiment

Design and conduct an experiment to determine the effectiveness of the collision detection algorithm you implemented. Compare the optimized `collision` method you implemented with the following approach:

```
query = // pick a segment to test with, you decide how
boolean collisionFound = false;
tree.traverseFarToNear(x, y, //they don't matter
(segment) -> {
	if(segment.intersects(query)){
		collisionFound = true;
	}
}
```

which will visit all nodes. Does our optimized collision detection routine run in the big O you expected? Be sure to describe the details of your experiment



The optimized approach led to consistent low runtimes across different input sizes where its time complexity resembles O (log n) where the complexity grows logarithmically with input size.

The alternative approach using the given collision method is more reminiscent of O (n) and lower than O(n^2) behavior suggesting a linear growth in time complexity with the size of the input. The expected time complexity for the optimized approach was of O (log n) due to the well balanced BSP tree's structure where the search space is consistently reduced in half at each level that leads to a logarithmic growth in the number of levels needed to reach a leaf node. On the other hand; the time complexity of O (n) from the alternative approach was also expected due to the need to traverse many segments whereas traverseFarToNear initiates recursively the entire BSP tree visiting each node from far to near; as the number of segments (n) increases; the algorithm needs to visit each segment exactly once during traversal.