Project 3

Anket Vilasrao Patil

1 Problem Analysis

We You are given a set of n types of rectangular 3-D boxes, where the i^th box has height h(i), width w(i) and depth d(i) (all real numbers). You want to create a stack of boxes which is as tall as possible, but you can only stack a box on top of another box if the dimensions of the 2-D base of the lower box are each strictly larger than those of the 2-D base of the higher box. Of course, you can rotate a box so that any side functions as its base. It is also allowable to use multiple instances of the same type of box.

The main idea to solve this problem to generate all box rotations, sorts them by descending base area, and use dynamic programming to maximize stack height by checking viable stacking options for each box as the topmost box.

2 Pseudocode

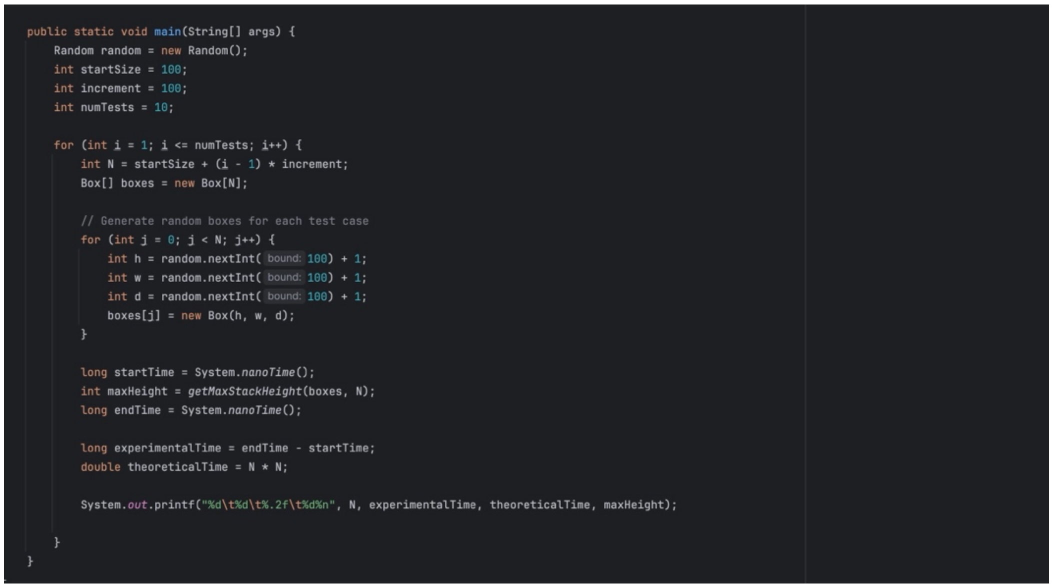
```
function getMaxStackHeight (boxes, n):
  Initialize allRotations array of size 3 * n

  for each box in boxes:
    Add original orientation, rotated with width as height, and rotated with depth as height to allRotations

  Sort allRotations by base area in descending order

  Initialize maxHeight array with each element as the height of the corresponding box
  Initialize maxOverallHeight as 0

  for i from 1 to size of allRotations:
    for j from 0 to i-1:
      if allRotations[i] can be stacked on allRotations[j]:
        maxHeight[i] = max(maxHeight[i], maxHeight[j] + allRotations[i]. height)

    Update maxOverallHeight to max(maxOverallHeight, maxHeight[i])

  return maxOverallHeight
```

2 Theoretical Analysis

The primary computation involves a double loop iterating through the possible rotations, resulting in a time complexity of $O((3N)^2) = O(N^2)$ and sorting the boxes, which takes O(3n * log(3n)) = O(nlogn) where n is the number of boxes. $O(N^2)$ dominates O(nlogn). Hence the time complexity of our algorithm will be $O(N^2)$.

3 Experimental Analysis (GitHub)

3.1 Program Listing



```java
public static void main(String[] args) {
    Random random = new Random();
    int startSize = 100;
    int increment = 100;
    int numTests = 10;

    for (int i = 1; i <= numTests; i++) {
        int N = startSize + (i - 1) * increment;
        Box[] boxes = new Box[N];

        // Generate random boxes for each test case
        for (int j = 0; j < N; j++) {
            int h = random.nextInt( bound: 100) + 1;
            int w = random.nextInt( bound: 100) + 1;
            int d = random.nextInt( bound: 100) + 1;
            boxes[j] = new Box(h, w, d);
        }

        long startTime = System.nanoTime();
        int maxHeight = getMaxStackHeight(boxes, N);
        long endTime = System.nanoTime();

        long experimentalTime = endTime - startTime;
        double theoreticalTime = N * N;

        System.out.printf("%d\t%d\t%.2f\t%d%n", N, experimentalTime, theoreticalTime, maxHeight);

    }
}
```

Figure 1: Main Program Code Snippet

We are starting with n=100 and incrementing n by 100 each iteration up to 10 times.

3.2 Data Normalization Notes

Theoretical values are scaled by a factor of 34.92300857. This scaling factor is derived by calculating the ratio of sum of experimental time to the sum of theoretical time.

3.3 Output Numerical Data

N	Experimental (ns)	Theoretical (units)	Scaled Theoretical (ns)
100	5612417	10000	349230.0857
200	5046958	40000	1396920.343
300	4219666	90000	3143070.771
400	6062625	160000	5587681.371
500	8482500	250000	8730752.143

Table 1: Experimental Data

3.4 Graph

The Graph compares experiment and theoretical times across 10 test cases. The horizontal axis ranges from 100 to 1000, while vertical axis represents time taken in nanoseconds.

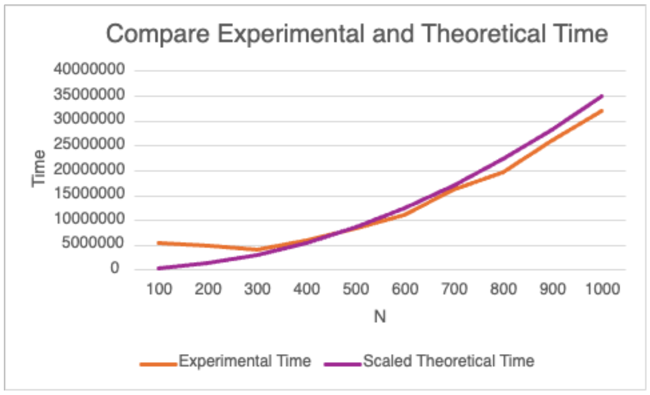


Figure 2 Comparison between Experimental and Scaled Theoretical Times

3.5 Graph Observations

It is evident from the Fig 2; the Graph shows a positive correlation between experimental and theoretical times, and both growing at a quadratic rate of O(N^2). While experimental times generally follows theoretical trend, there are deviations for smaller values of N due to interruptions from background process.

4 Conclusions

This project demonstrates an effective dynamic programming approach to maximize stack height by generating all box rotations and sorting by base area. Experimental results align closely with the O(N^2) theoretical complexity, confirming predictable quadratic growth, especially for larger N. Minor deviations at smaller N are likely due to background overhead. Overall, the algorithm is efficient, scales well, and meets the project objective of maximizing stack height under the given constraints.