COSC 320

Design and Implementation

Team Project
November, 2018
Professor: Dr. Yong Gao
The University of British Columbia Okanagan

Jeena Khaira - 37068137
Varun Kaushal - 64320161
Emmanuel Werimegbe - 29653152
Georges Nasrallah - 41226151

Part One: Problem Formulation

A day trader makes a **profit** every trading day. This can be positive or negative.
-> An array of daily profits over a time period. Ex. { 2, -2, 3 ,4 ,6 }
Each index of the array is the profit he/she made that day. The whole array is a time period (a list of consecutive trading days) of the daily profits.

The **performance** of the day trader is measured by *the sum of his profits over the time period*.
Profit = each index of the array
Time period = consecutive day(s) therefore, any sub array in the array. Could even be just one day (one index).

Ex. { 2, -2, 3 ,4 ,6 } the sum of all the indexes the array = 13 for the time period of 5 days.

The **strength S** of the day trader is the *maximum performance over all possible time periods*.
Ex. the time period with the highest performance number
Therefore, the maximum subarray (the subarray that sums to the largest number) is the strength. This can be the entire array if it is all positive numbers.

Assumptions:

- If the array contains all positive numbers, the strength (maximum performance) of the day trader is the entire array
- If the array contains all negative numbers, the strength (maximum performance) of the the day trader is the day he/she “profited” the most (index with largest number)
- Some performance values can be equivalent

Part Two: Algorithm Design and Implementation

Algorithm A:

```
profit ← []
n = the size of array profit
max ← 0
for i ← 0...n; i++
        for j ← i...n; j++
           performance ← 0
           for k← 0...j; k++
                performance += profit[k]
                If performance > strength
                    strength ← performance
            endfor
        endfor
Endfor
return max
```

Running Time: $O(n^3)$

Algorithm B:

```
performance[] ← 0
for i ← 1 … n; i++
        if (performance[i-1]+profits[i-1]) > performance[i] //the recurrence
                performance[i] ← performance[i-1]+profits[i-1]
        else
                performance[i] ← profits[i-1]
        endif
endfor
strength ← 0
for x ← 1...n; x++
        if performance[x] > strength
                strength ← ar[x]
        endif
endfor
Return strength
```

Running time: O(n log n)

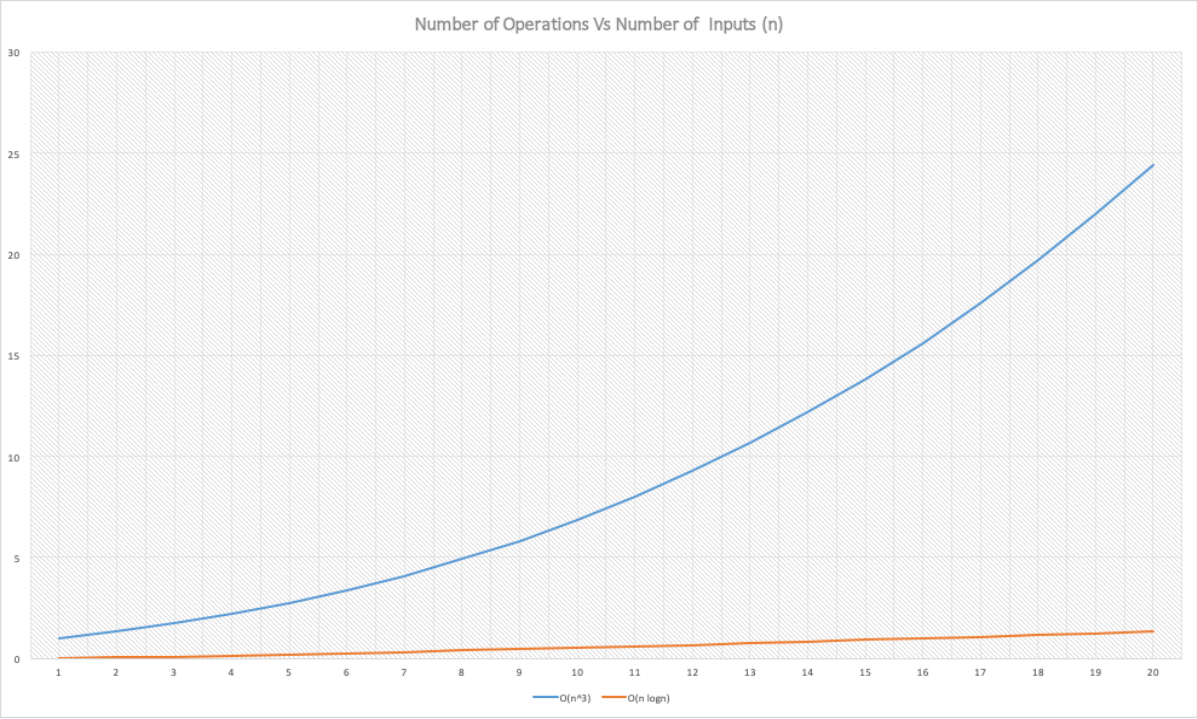
Correctness Proof:

This is a bottom up approach featuring dynamic programming. The recurrence of this following algorithm (after finding the performance array from the profits array) is

$$max = [\ (performance[i-1]+profits[i-1]),\ performance[i]\ ]$$

where i=1.
Base Case: Ar[] has 1 element then the strength is the element
Let Ar[] be an array with indexes from 0…n. This is the array we in which we ant to find the strength from. We then have another array S where S[j] = max(∑A[k], k ≤ j,k≥i). This means in terms of finding the strength or maximum subarray:
Strength[j]=max(S[j-1]+A[j],A[j]). So our recurrence step is comparing performance[i] tp performance [i-1]+profits[i]. This is what allows us to be able to handle a high amount of inputs.

Comparison Using Asymptotic Notation:



Number of Inputs(x-axis)
Number of Operations(y-axis)
This graph compares the running times of our brute force method A which runs in O(n^3) time and our dynamic programming method B which runs in O(nlogn) time.
It can be seen that an increasing number of inputs results in an exponential increase in the number of operations performed for O(n^3) while O(nlogn) increases linearly in number of operations for an increasing number of inputs.

Part Three: Implementation and Empirical Evaluation

```
import java.util.Arrays;

public class DayTraderStrength {
	int algorithmAforStrength(int[] profits) {

		int strength = 0;
		int performance = 0;
		int n = profits.length;
		for (int i = 0; i < n; i++) {
			for (int j = i; j < n; j++) {
				performance = 0;
				for (int k = i; k <= j; k++) {
					performance = performance + profits[k];
					if (strength < performance) {
						strength = performance;
					}
				}
			}
		}
		return strength;
	}

	public int algorithmBforStrength(int[] profits) {
		int[] performance = new int[profits.length + 1];
		performance[0] = 0;
```

```
		for (int i = 1; i < performance.length; i++) {
			int prev = (performance[i - 1] + profits[i - 1]);
			if (prev > profits[i - 1]) {
				performance[i] = prev;
			} else {
				performance[i] = profits[i - 1];
			}
		}

		int strength = 0;
		for (int x = 1; x < performance.length; x++) {
			if (performance[x] > strength)
				strength = performance[x];
		}

		return strength;
	}

// To test the performance of our methods, we used an array generating random
elements.
	public static void main(String[] args) {
		int profits[] = new int[8000];

		for (int i = 0; i < profits.length; i++) {
			profits[i] = (int) (-10 + Math.random() * 20);
		}
		DayTraderStrength i = new DayTraderStrength();
		System.out.println("dynamic  " + i.algorithmBforStrength(profits));
		System.out.println("brute  " + i.algorithmAforStrength(profits));

	}
}
```