10802EE 398000 Algorithms

Homework 6. Trading Stock, II

- **Introduction**

We have already solved the **single-buy-single-sell stock trading problem** in Homework 5. The definition of the **single-buy-single-sell stock trading problem** is that we are only permitted to complete at most one transaction (i.e., buy one and sell one share of the stock). We have to find the buying day and the price, the selling day and the price, and the maximum earning. Please note that we can only sell a stock after we buy a stock. That is, the selling day must be after the buying day.

The input files, s1.dat – s9.dat, are the history of closing price of the Google stock. The first line of each file contains the number of data entries in that file which is followed by the date and the closing price of that day.

In Homework 5, we used two kinds of **maximum subarray algorithms**, **Brute-Force Approach** called **MaxSubArrayBF and Divide and Conquer Approach** called **MaxSubArray** to solve this problem. The definition of **maximum subarray algorithm** is that given the input array of size n, A[n], we have to output *range, low and high*, such that the equation(1) holds:

$$\sum_{i=low}^{high} A[i] = \max_{1 \le j \le k \le n} \sum_{i=j}^{k} A[i] \qquad (1)$$

We can transform each input file into daily price change information by subtracting the closing price yesterday from the closing price today. Then we can use the maximum subarray algorithm to solve this problem. The elements A[i] of array A[n] in this case is the difference between the closing

price on i-th day and the closing price on (i - 1)-th day. Please note that A[0] = 0. Thus, the buying day for the stock is actually *low – 1*.

However, **Brute-Force Approach** in Homework 5 is inefficiency, which possesses a time complexity of $O(n^3)$. Thus, this homework is intended to fix the problem. There are two parts in this homework:

1. Make **Brute-Force Approach,** Algorithm **MaxSubArrayBF,** achieve $O(n^2)$ complexity.

2. Design an algorithm in solving the single-buy-single-sell stock trading problem which can achieve a lower time complexity, lower than O(n lg n), which Algorithm **MaxSubArray**, **Divide and Conquer** version of **MaxSubArrayBF**, possesses.

I have already analyzed the space and time complexities of **MaxSubArrayBF** and **MaxSubArray** in homework 5. Thus, I will focus on introducing two new algorithms: **MaxSubArrayBF2** and **MaxMin**. Algorithm **MaxSubArrayBF2** is the advanced version of Algorithm **MaxSubArrayBF** which can achieve $O(n^2)$ complexity. Algorihtm **MaxMin** is the new algorithm that can achieve a time complexity lower than O(n lg n).

I will explain the correctness of **MaxSubArrayBF2** and **MaxMin**, analyze their space and time complexities, implement both of them, measure their CPU time. Finally, I will compare the CPU time among **MaxSubArrayBF**, **MaxSubArray, MaxSubArrayBF2,** and **MaxMin** to see whether we have achieved the goal. That is, whether **MaxSubArrayBF2** achieves $O(n^2)$ complexity and **MaxMin** achieve a time complexity lower than O(n lg n).

● Approach

The pseudocode of **MaxSubArrayBF** is shown below. It is **Brute-Force Approach** of the **maximum subarray algorithm** with O(n) space complexity and $O(n^3)$ time complexity.

```
// Find low and high to maximize∑A[i], low ≤ i ≤ high.
// Input: A[1 : n], int n, A[i] is the difference of closing price between i-th day and (i - 1)-th day.
// Output: 1 ≤ low,high ≤ n and max, low – 1 is the buying day, high is the selling day.
1 Algorithm MaxSubArrayBF(A,n,low,high)
2 {
3       max := 0; // Initialize
4       low := 1;
5       high := n;
6       for j := 1 to n do { // Try all possible ranges: A[j : k].
7             for k := j to n do {
8                   sum := 0;
9                   for i := j to k do { // Summation for A[j : k]
10                        sum := sum + A[i];
11                  }
12                  if (sum > max) then { // Record the maximum value and range.
13                        max := sum;
14                        low := j;
15                        high := k;
16                  }
17            }
18      }
19      return max;
20 }
```

Algorithm 1 The pseudocode of MaxSubArrayBF.

Recall that the meaning of element A[i] of array A[n] in **Algorithm 1** is the difference between the closing price on i-th day and the closing price on (i - 1)-th day. Thus, what line 9 to line 11 has done is to sum up the difference of the closing price of (j - 1)-th day and j-th day, (j + 1)-th day and j-th day, …, to (k – 1)-th day and k-th day. This can be simplified as equation(2):

$$sum = \sum_{i=j}^{k} A[i] = (p[j] - p[j-1]) + (p[j+1] - p[j]) + \cdots + (p[k] - p[k-1])$$

$$= p[k] - p[j-1] \qquad (2)$$

$$where\ \ p[i]\ \ is\ \ the\ \ closing\ \ price\ \ of\ \ the\ \ i-th\ \ day$$

From equation(2), we can replace line 9 to line 11 with p[k] – p[j]. Why do I use p[j] instead of p[j – 1]? Recall that for Algorithm **MaxSubArrayBF**, the buying day for the stock is actually *low – 1*. If I use p[j] instead of p[j – 1], the buying day for the stock can become *low* which is more intuitively and may reduce errors. Thus, I replace line 9 to line 11 in **MaxSubArrayBF** with sum = A[k] – A[j] and get the new algorithm, **MaxSubArrayBF2**, which is shown in the next page.

Please be careful that in Algorithm **MaxSubArrayBF2**, the meaning of elements A[i] of array A[n] is the closing price of the i-th day which is different from **MaxSubArrayBF.** Besides, the buying day for the stock is *low* instead of *low – 1*.

We can use the more intuitive way to explain **MaxSubArrayBF2** instead of pure mathematics. The earning is the difference of the closing price between the selling day and the buying day. Thus, we calculate every possible difference in line 6 to line 15 of **MaxSubArrayBF2** and choose the maximum one. The outer loop index j is the buying day. The inner loop index k is the selling day. Since the selling day must be after the buying day, the inner loop starts from j.

The space complexity of **MaxSubArrayBF2** is O(n) because it is characterized by n, the number of elements in the list A.

```
// Find low and high to maximize A[high] – A[low], low ≤ i ≤ high.
// Input: A[1 : n], int n, A[i] is the closing price of the i-th day.
// Output: 1 ≤ low,high ≤ n and max. low is the buying day, high is the selling day.
1 Algorithm MaxSubArrayBF2(A,n,low,high)
2 {
3       max := 0; // Initialize
4       low := 1;
5       high := n;
6       for j := 1 to n do { // Try all possible ranges: A[j : k].
7             for k := j to n do {
8                   sum := A[k] – A[j];
9                   if (sum > max) then { // Record the maximum value and range.
10                        max := sum;
11                        low := j;
12                        high := k;
13                  }
14            }
15      }
16      return max;
17 }
```

Algorithm 2 The pseudocode of MaxSubArrayBF2.

The time complexity is $O(n^2)$ because there is a double for loop.

In previous analysis, we have already transformed the maximum subarray problem into the maximum difference problem. The next question is that can we find the maximum difference using only one for loop? The first idea comes to my mind is finding the maximum and the minimum price, then the maximum earning is the difference of them. However, this is wrong because the selling day must be after the buying day.

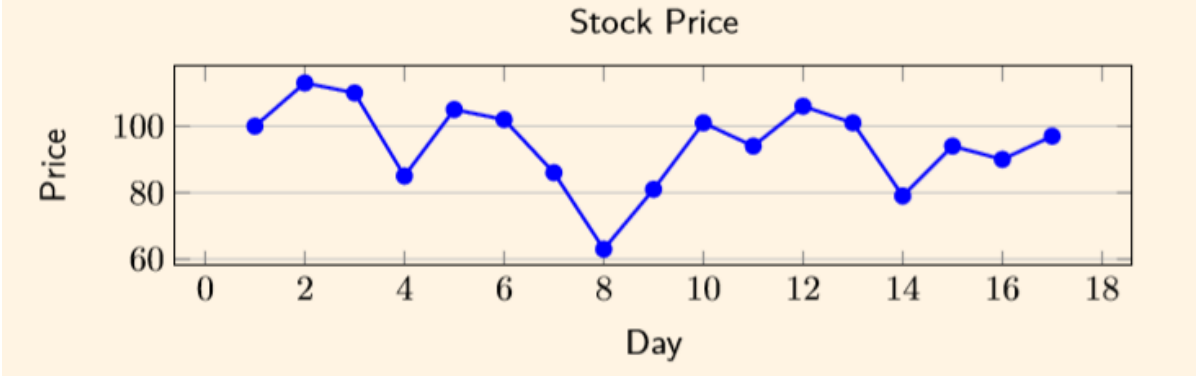

Figure 1 The line graph of the stock price. (From P.23, lec31 handout)

From Observing **Figure 1**, we can try to find the maximum difference forwardly to make the selling day be after the buying day. First, we set the price of day1 to be the local minimum value. Then we go to the price of the next day and compare it with the local minimum value. If it is smaller than the local minimum value, we set it as the new local minimum value, otherwise we calculate the difference between it and the local minimum value. If the difference is larger than the maximum difference until now, we update the maximum difference (also the value of *low* and *high*), otherwise we do nothing. Then, we go to the price of the next day, and so on.

The pseudocode of this algorithm **(Algorithm 3)** is shown in the next page. There are two important condition: low ≤ i ≤ high, and A[high] – A[low] is maximized. Both conditions are satisfied because we keep updating the local minimum value, so the difference between the price of following days(if the price of following days is larger than the local minimum value) and it can be maximized. Then, we pick the largest one. Also, we do this forwardly. The high-th day must be after low-th day.

```
// Find low and high to maximize A[high] – A[low], low ≤ i ≤ high.
// Input: A[1 : n], int n, A[i] is the closing price of the i-th day.
// Output: 1 ≤ low,high ≤ n and maxGain. low is the buying day, high is the selling day.
1 Algorithm MaxMin(A,n,low,high)
2 {
3       maxGain = 0; // Initialize
4       min := A[0]; // The minimum value
5       low := 1;
6       lowTemp := 1; // The index of the local minimum
7       high := 1;
8       for i := 1 to n do {
9             if (min > A[i]) then { // Record the minimum value and its index.
10                  min := A[i];
11                  lowTemp := i;
12            }
13            else {
14                  if (A[i] – min > maxGain) then {
15                        maxGain = A[i] – min; // update the max gain
16                        low = lowTemp; // update the return index low and high
17                        high = i;
18                  }
19            }
20      }
21      return maxGain;
22 }
```

Algorithm 3 The pseudocode of MaxMin.

The space complexity of **MaxMin** is O(n) because it is characterized by n, the number of elements in the list A.

The time complexity of **MaxMin** is O(n) because there is only one for loop. It only scans each element in A array once. The size of A array is n.

Same as **MaxSubArrayBF2**, the meaning of elements A[i] of array A[n] is the closing price of the i-th day. Besides, the buying day for the stock is *low*.

	Space complexity	Time complexity
MaxSubArrayBF	O(n)	$O(n^3)$
MaxSubArray	O(n)	O(n lg n)
MaxSubArrayBF2	O(n)	$O(n^2)$
MaxMin	O(n)	O(n)

Table 1 The time and space complexities.

- **Results**

Data file	N	CPU Time (seconds)				Earning (per share)
		MaxSubArrayBF	MaxSubArray	MaxSubArrayBF2	MaxMin	
s1	16	1.00136E-05	1.39689E-06	1.35493E-06	1.45912E-07	9.065
s2	32	4.88758E-05	3.51000E-06	5.49197E-06	3.36885E-07	35.05
s3	64	3.22104E-04	9.42492E-06	1.91581E-05	9.53913E-07	96.02
s4	128	2.36988E-03	2.24380E-05	6.44238E-05	1.40309E-06	110.85
s5	256	1.82462E-02	4.65081E-05	2.09724E-04	2.26212E-06	213.93
s6	512	1.25126E-01	7.22802E-05	7.75693E-04	4.24194E-06	371.62
s7	1024	7.86945E-01	1.48792E-04	3.08091E-03	7.73597E-06	641.78
s8	2048	6.03797E+00	3.12796E-04	1.24379E-02	1.40531E-05	662.49
s9	3890	4.11115E+01	6.24696E-04	4.51731E-02	2.63069E-05	1384.68

Table 2 Results table

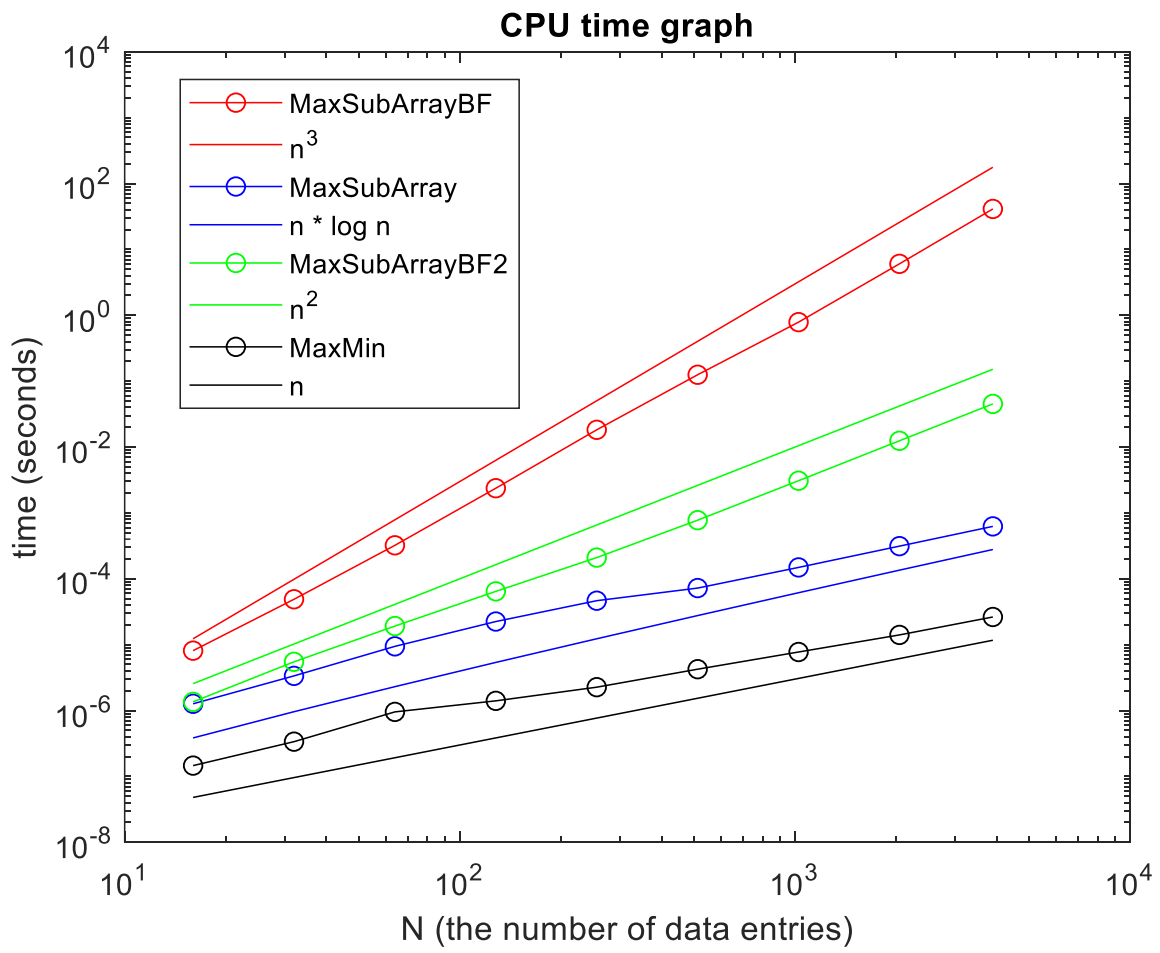

Figure 2 CPU time result.

The results in **Figure 2** is consistent with **Table 1**. We have achieved our goal: the CPU time complexity of MaxSubArrayBF2 is actually $O(n^2)$, and the CPU time complexity of MaxMin is actually O(n) which is lower than O(n lg n).