EE3980 Algorithms

hw06 Trading Stock, II

106061146 陳兆廷

Introduction:

In this homework, I will be analyzing, implementing, and observing 2 algorithms, and comparing them with the 2 algorithms in previous homework. The goal of the algorithms is to find the best buying point and selling point for a set of stock price. The input of them will be history of Google stock closing price, and the output of them will be the best buying point, the best selling point and the profit per share.

During the analysis process, I will be using table-counting method to calculate the time complexities of the 2 algorithms. Furthermore, I will try to find the best-case, worst-case, and average-case conditions for the 2 algorithms, respectively. Before implementing on C code, I will try to predict the result based on my analysis. Finally, I will calculate their space complexity for the total spaces used by the algorithm.

The implementation of the 2 algorithms on C code mainly focus on the average time, best-case and worst-case conditions. 9 testing data are given by Professor Chang, and the working environment is on my ubuntu with linux kernel 5.3.0, and gcc version 7.5.0.

Lastly, the observation of them will be focusing on the time complexity of the results from implementation. Moreover, I will compare 4 algorithms with each other and make some rankings. Finally, I will check the experimented results with my analysis.

Analysis:

1. **Peak and Valley in an array**

   Peak and valley are the maximum and minimum values in an array. There are a lot of problems that needs to find peaks and valleys in arrays, for this task, stock trading problem, it can be used to find the best buying and selling point, since the way of acquiring maximum profit is to buy at the lowest price and sell at the highest price.

2. **MaxSubArrayBF2 (Brute-force with O(N^2) time complexity)**

   a. Abstract:

      MaxSubArrayBF2 goes through all combinations of buying and selling stocks and finds the 1 way of doing so that has the maximum profit.

      The A[i] in the below algorithm indicates the price of each share of stocks. By subtracting the prices of 2 shares, the algorithm gets the profits and find the maximum value among them.

   b. Algorithm:

```
1.  // Find low and high to maximize A[low] – A[high], low  high.
2.  // Input: A[1 : n ], int n
3.  // Output: 1 >= low, high <= n and max.
4.  Algorithm MaxSubArrayBF2(A, n, low, high)
5.  {
6.      max := 0 ; // Initialize
7.      low := 1 ;
8.      high := n ;
9.      for j := 1 to n do { // Try all possible ranges: A[j : k ].
10.         for k := j to n do {
11.             sum := A[k] – A[j] ;
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. }
```

c. Proof of correctness:

In this algorithm, it calculates the profits for every possible combination of buying and selling this stock in N*N iterations, where N is the number of stock prices. In line 15 ~ 19, it constantly replaces the stored maximum combinations. Using induction, we can conclude that in every iteration, it either stores the best way so far to buy and sell a stock, or do not store anything. The algorithm terminates when all combination is tested and return the best way.

d. Time complexity:

	s/e	freq	total
`Algorithm MaxSubArrayBF2(A, n, low, high)`	0	0	0
	0	0	0
`{`	1	1	1
`    max := 0 ;`	1	1	1
`    low := 1 ;`	1	1	1
`    high := n ;`	N	1	N
`    for j := 1 to n do {`	N/2	N	1/2N^2
`        for k := j to n do {`	1	N^2	N^2
`            sum := A[k] - A[j] ;`	1	N^2	N^2
`            if (sum > max) then {`	1	N^2	N^2
`                max := sum ;`	1	N^2	N^2
`                low := j ;`	1	N^2	N^2
`                high := k ;`	0	N	0
`            }`	0	0	0
`        }`	0	0	0
`    }`	1	1	1
`    return max ;`	0	0	0
`}`			
p.s. N/2 sine it goes through 1 ~ N in N iterations	5.5N^2 + N + 4		

The time complexity of *MaxSubArrayBF2* is O(N^2), where N is the number of stock shares.

Best case, Worst case and Average case:

The difference between best case and worst case for this is not obvious. The reason for this is that either way, it goes through all iterations anyway. The only difference is the times of updating the maximum value, which we can neglect since it costs only few steps. And for the above reasons, the average case is quite the same, too.

e. Space Complexity:

The algorithm uses 6 integers and N pairs of elements in array of stocks. **The space complexity would be O(N).**

3. **MaxSubArrayN (Search Extreme Values)**

a. Abstract:

MaxSubArrayN finds the peak and valley in the array of stock shares. By subtracting the maximum and minimum value of the shares, it gets the maximum profit, the best buying and selling point for this set of stocks. Be aware that the valley should be prior to the peak.

b. Algorithm:

```
// Find low and high to maximize A[low] – A[high], low  high.
// Input: A[1 : n ], int n
// Output: 1 >= low, high <= n and max.
Algorithm MaxSubArrayN(A, n, low, high)
{
    minprice := 0 ; // Initialize
    sum := 0 ;
    for i := 1 to n do { // Try all possible ranges: A[j : k ].
        if  A[i] > minprice do {
            minprice = A[i];
        }
    }
    for i := 1 to n do { // Try all possible ranges: A[j : k ].
        if  A[i] < minprice do {
            minprice = A[i];
            low2 = i;
        }  else if (A[i] - minprice) > sum do {
            sum = A[i] - minprice;
            low1 = low2;
            high = i;
        }
    }
    return sum ;
}
```

c. Proof of correctness:

In this algorithm, it first finds the maximum value among the array A in order to set a boundary for the next step, find minimum. It takes N iterations to find it. Using simple induction, we can prove that we can find the maximum value in the array. In each iteration, we store the i-i-th value if it is larger than previous maximum value.

Next, it searches for the valley and peak in an array. In each iteration in N iterations, it updates the valley, which is the best buying point, and updates the peak, which is the best selling point. Using induction, as aforementioned paragraph, we can prove that it is correct. Be aware that the buying point is updated when it finds selling point later than it.

d. Time complexity:

	s/e	freq	total
`Algorithm MaxSubArrayN(A, n, low, high)`	0	0	0
`{`	0	0	0
`  minprice := 0 ;`	1	1	1
`  sum := 0 ;`	1	1	1
`  for i := 1 to n do {`	N	1	N
`    if  A[i] > minprice do {`	1	N	N
`      minprice = A[i];`	1	N	N
`    }`	0	N	0

`}`	0	1	0
`for i := 1 to n do {`	N	1	N
`  if A[i] < minprice do {`	1	N	N
`    minprice = A[i];`	1	N	N
`    low2 = i;`	1	N	N
`  } else if (A[i] - minprice) > sum do{`	1	N	N
`    sum = A[i] - minprice;`	1	N	N
`    low1 = low2;`	1	N	N
`    high = i;`	1	N	N
`  }`	0	0	0
`}`	0	0	0
`return sum ;`	1	1	1
`}`	0	0	0
	11N + 4		

The time complexity of *MaxSubArrayBF2* is O(N), where N is the number of stock shares.

<u>Best case, Worst case and Average case:</u>

The difference between best case and worst case for this is not obvious. The reason for this is that either way, it finds peak and valley of the array. The only difference is the times of updating the maximum value, which we can neglect since it costs only few steps. And for the above reasons, the average case is quite the same, too.

e. Space Complexity:

The algorithm uses 7 integers and N pairs of elements in array of stocks. **The space complexity would be O(N).**

4. Time & Space Complexity Comparison:

	MaxSubArrayBF	*MaxSubArray*	*MaxSubArrayBF2*	*MaxSubArrayN*
Time	O(N^3)	O(N lg N)	O(N^2)	O(N)
Space	O(N)	O(N)	O(N)	O(N)

Speed (fast>slow):

MaxSubArrayN > MaxSubArray> MaxSubArrayBF2 > MaxSubArrayBF.

Implementation:

1. **Speed Test:**

Speed Test is to find the actual speed and time complexities of the 4 algorithms, *MaxSubArrayBF, MaxSubArray, MaxSubArray* and *MaxSubArrayN*. We use 9 test inputs given by Professor and get the CPU runtimes before and after the algorithms perform their tasks. The implementation is done on my laptop. However, the time recording methods for the 4 algorithms are different. Since *MaxSubArrayBF* runs much slower than the rest of them, I can only run *MaxSubArrayBF* once and record the CPU runtime. However, I will run the others 1000 times and record the average runtime for it.

Workflow for *MaxSubArrayBF* :

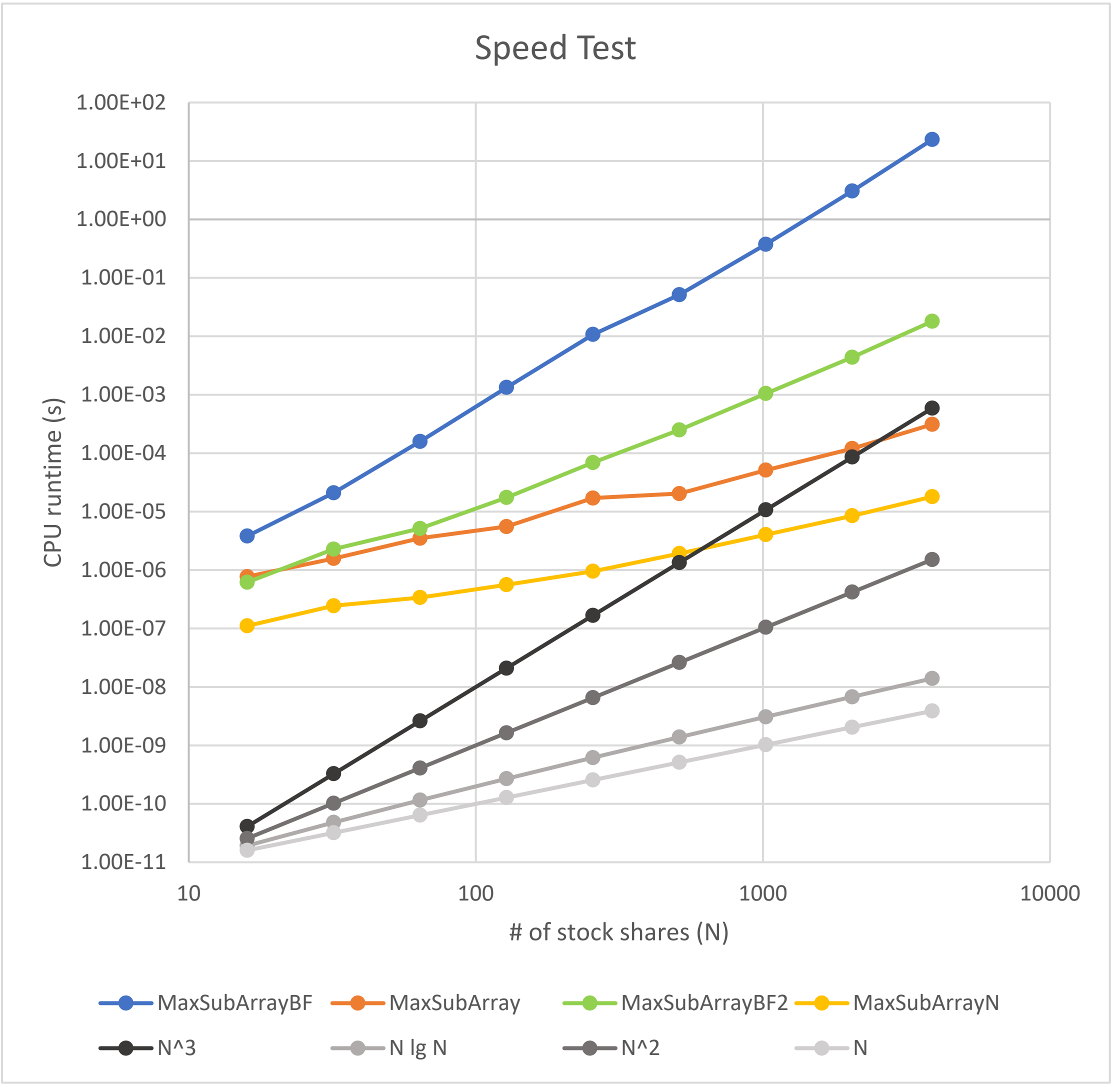
```
t_MaxSubArrayBF = GetTime();              // initialize time counter
MaxSubArrayBF();
t_MaxSubArrayBF = GetTime() - t_MaxSubArrayBF;  // calculate CPU time
```

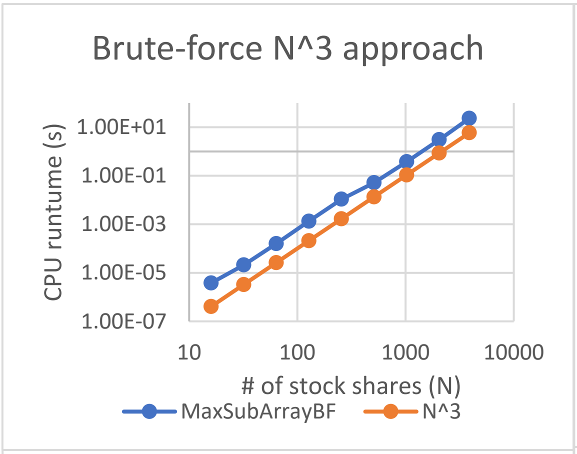
Workflow for the other 3 algorithms :

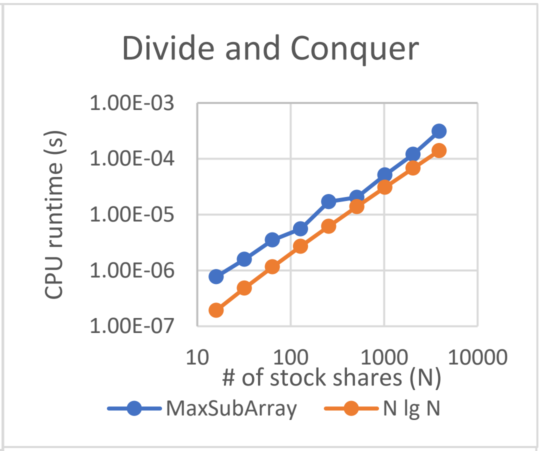
```
t = GetTime();                // initialize time counter
for i := 0 to 1000 do {
    Algorithm();
}
t = (GetTime() - t) / 1000; // calculate CPU time
```

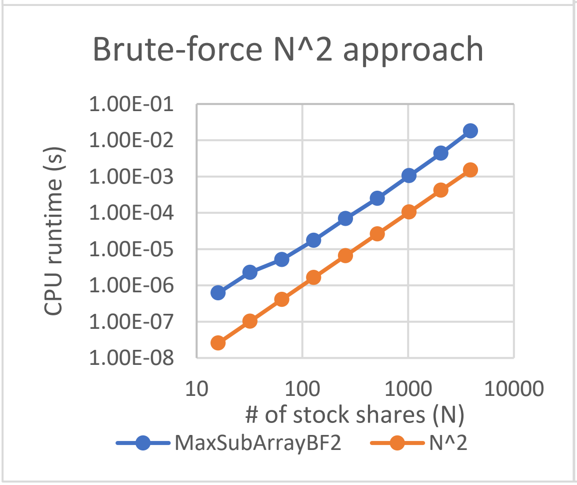
Results:

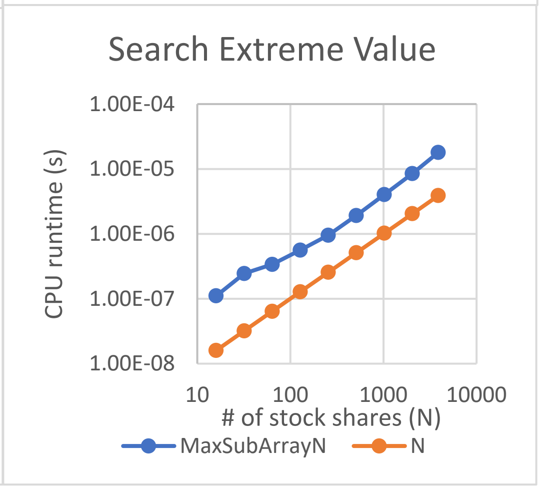
N (s)	*MaxSubArrayBF*	*MaxSubArray*	*MaxSubArrayBF2*	*MaxSubArrayN*
16	3.81E-06	7.72E-07	6.19E-07	1.11E-07
32	2.10E-05	1.58E-06	2.28E-06	2.45E-07
64	1.59E-04	3.51E-06	5.15E-06	3.39E-07
128	1.34E-03	5.55E-06	1.74E-05	5.62E-07
256	1.07E-02	1.71E-05	6.90E-05	9.52E-07
512	5.15E-02	2.03E-05	2.51E-04	1.92E-06
1024	3.76E-01	5.12E-05	1.05E-03	4.04E-06
2048	3.06E+00	1.20E-04	4.38E-03	8.49E-06
3890	2.33E+01	3.11E-04	1.81E-02	1.80E-05











Observation:

1. Speed, Time complexity:

Actual Speed (> means faster):

***MaxSubArrayN > MaxSubArray> MaxSubArrayBF2 > MaxSubArrayBF*.**

The result matches my analysis precisely. The time complexity of *MaxSubArrayBF2* and *MaxSubArrayN* are **O(N^2)** and **O(N)**. Modify and deviate from maximum subarray approach made the algorithm did made the iteration depth for *MaxSubArrayBF2* lesser by N. By seeking peak and valley in an array did made the time complexity goes by O(N) only.

Overall, the implemented results meet my analysis.

Conclusions:

1. Time and space complexities of the 5 algorithms:

	MaxSubArrayBF	*MaxSubArray*	*MaxSubArrayBF2*	*MaxSubArrayN*
Time	O(N^3)	O(N lg N)	O(N^2)	O(N)
Space	O(N)	O(N)	O(N)	O(N)

2. Actual runtime comparison:

   Speed (> means faster):

   ***MaxSubArrayN > MaxSubArray> MaxSubArrayBF2 > MaxSubArrayBF*.**

3. It goes faster when not using the maximum subarray property.