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 first algorithm and use divide and conquer for the second algorithm. 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.

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

1. Algorithm:
2. // Find low and high to maximize A[low] – A[high], low  high.
3. // Input: A[1 : n ], int n
4. // Output: 1 >= low, high <= n and max.
5. Algorithm MaxSubArrayBF2(A, n, low, high)
6. {
7. max := 0 ; // Initialize
8. low := 1 ;
9. high := n ;
10. **for** j := 1 to n **do** { // Try all possible ranges: A[j : k ].
11. **for** k := j to n **do** {
12. sum := A[k] – A[j] ;
13. **if** (sum > max) then { // Record the maximum value and range.
14. max := sum ;
15. low := j ;
16. high := k ;
17. }
18. }
19. }
20. **return** max ;
21. }  ­
22. 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.

1. Time complexity:

|  |  |  |  |
| --- | --- | --- | --- |
|  | s/e | freq | total |
| 1. Algorithm MaxSubArrayBF2(A, n, low, high) 2. { 3. max := 0 ; 4. low := 1 ; 5. high := n ; 6. **for** j := 1 to n **do** { 7. **for** k := j to n **do** { 8. sum := A[k] – A[j] ; 9. **if** (sum > max) then { 10. max := sum ; 11. low := j ; 12. high := k ; 13. } 14. } 15. } 16. **return** max ; 17. }  ­ | 0  0  1  1  1  N  N/2  1  1  1  1  1  0  0  0  1  0 | 0  0  1  1  1  1  N  N^2  N^2  N^2  N^2  N^2  N  0  0  1  0 | 0  0  1  1  1  N  1/2N^2  N^2  N^2  N^2  N^2  N^2  0  0  0  1  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.

1. Space Complexity:

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

1. **MaxSubArrayN (Search Extreme Values)**
2. 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.

1. Algorithm:
2. // Find low and high to maximize A[low] – A[high], low  high.
3. // Input: A[1 : n ], int n
4. // Output: 1 >= low, high <= n and max.
5. Algorithm MaxSubArrayN(A, n, low, high)
6. {
7. minprice := 0 ; // Initialize
8. sum := 0 ;
9. **for** i := 1 to n **do** { // Try all possible ranges: A[j : k ].
10. **if**  A[i] > minprice **do** {
11. minprice = A[i];
12. }
13. }
14. **for** i := 1 to n **do** { // Try all possible ranges: A[j : k ].
15. **if**  A[i] < minprice **do** {
16. minprice = A[i];
17. low2 = i;
18. }  **else** **if** (A[i] - minprice) > sum **do** {
19. sum = A[i] - minprice;
20. low1 = low2;
21. high = i;
22. }
23. }
24. **return** sum ;
25. }
26. 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.

1. Time complexity:

First, I use table-method to calculate *MaxSubArrayXB’s* time complexity.

|  |  |  |  |
| --- | --- | --- | --- |
|  | s/e | freq | total |
| 1. Algorithm MaxSubArrayXB(A, begin, mid, end, low, high) 2. { 3. lsum := 0 ; 4. low := mid ; 5. sum := 0 ; 6. **for** i := mid to begin step −1 **do** { 7. sum := sum + A[i ] ; 8. **if** (sum > lsum) then { 9. lsum := sum ; 10. low := i ; 11. } 12. } 13. rsum := 0 ; 14. high := mid + 1 ; 15. sum := 0 ; 16. **for** i := mid + 1 to end **do** { 17. sum := sum + A[i ] ; 18. **if** (sum > rsum) then { 19. rsum := sum ; 20. high := i ; 21. } 22. } 23. **return** lsum + rsum ; 24. } | 0  0  0  1  1  1  c  1  1  1  1  0  0  1  1  1  c  1  1  1  1  0  0  1  0 | 0  0  0  1  1  1  1  c  c  c  c  c  c  1  1  1  1  c  c  c  c  c  c  1  0 | 0  0  0  1  1  1  c  c  c  c  c  0  0  1  1  1  c  c  c  c  c  0  0  1  0 |
|  | 10c + 7 | | |

C is a constant between 0 to N/2. Therefore, **the time complexity of *MaxSubArrayXF* is O(N)**, where N is the number of stock shares.

Next, I use divide-and-conquer to calculate *MaxSubArray*’s time complexity. Let *MaxSubArray*’s time complexity be T(n), where T is a function of N. And, using divide-and-conquer, where is time complexity of *MaxSubArray*XB, we can imply this:

Then, assuming , and is known:

Then change back to :

Therefore, **the time complexity of *MaxSubArray* is O(n lg n).**

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 finds all best solutions in the divided 3 parts 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.

1. Space Complexity:

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

1. **Comparison:**

|  |  |  |
| --- | --- | --- |
|  | *MaxSubArrayBF* | *MaxSubArray* |
| Time Complexity | O(N^3) | O(N lg N) |
| Space Complexity | O(N) | O(N) |

**Speed (fast>slow): *MaxSubArray >>> MaxSubArrayBF*.**

**Implementation:**

1. **Speed Test:**

Speed Test is to find the actual speed and time complexities of the 2 algorithms, *MaxSubArrayBF* *a*nd *MaxSubArray*. 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 2 algorithms are different. Due to the fact that *MaxSubArrayBF* runs much slower than *MaxSubArray*, I can only run *MaxSubArrayBF* once and record the CPU runtime. However, I will run *MaxSubArray* 1000 times and record the average runtime for it.

Workflow :

1. t\_MaxSubArrayBF = GetTime();              // initialize time counter
2. MaxSubArrayBF();
3. t\_MaxSubArrayBF = GetTime() - t\_MaxSubArrayBF;  // calculate CPU time
4. t\_MaxSubArray = GetTime();              // initialize time counter
5. **for** i := 0 to 1000 **do** {
6. MaxSubArray();
7. }
8. t\_MaxSubArray = (GetTime() - t\_MaxSubArray) / 1000; // calculate CPU time

Results:

|  |  |  |
| --- | --- | --- |
| N | *MaxSubArrayBF* | *MaxSubArray* |
| 16 | 4.05312E-06 | 7.46965E-07 |
| 32 | 2.38419E-05 | 1.66988E-06 |
| 64 | 1.76907E-04 | 3.62515E-06 |
| 128 | 1.69802E-03 | 9.54986E-06 |
| 256 | 7.87807E-03 | 1.84182E-05 |
| 512 | 6.46009E-02 | 3.65239E-05 |
| 1024 | 3.97145E-01 | 7.20381E-05 |
| 2048 | 3.29296E+00 | 1.55273E-04 |
| 3890 | 2.56944E+01 | 3.91071E-04 |

**Observation:**

1. Speed, Time complexity:

**Actual Speed (> means faster): *MaxSubArray >>> MaxSubArrayBF.***

The result matches my analysis precisely. The time complexity of *MaxSubArrayBF* is **O(N^3)** and this of *MaxSubArray* is **O(N lg N)**. The reason of this is *MaxSubArray* uses Divide-and-Conquer instead of brute-force approach. And therefore we can conclude that **Divide-and-Conquer did improve the time performance when solving a problem.**

Overall, the implemented results meet my analysis.

**Conclusions:**

1. Time and space complexities of the 3 algorithms:

|  |  |  |
| --- | --- | --- |
|  | *MaxSubArrayBF* | *MaxSubArray* |
| Time Complexity | O(N^3) | O(N lg N) |
| Space Complexity | O(N) | O(N) |

1. Actual runtime comparison:

**Actual Speed (> means faster): *MaxSubArray >>> MaxSubArrayBF.***

1. Divide-and-Conquer **did** improve the time perfomance when solving a problem.