EE 3980 Algorithms

Homework 6. Trading Stock II

105061110 周柏宇

2020/4/19

1. Introduction

In this homework, our objective is the same as homework 5 (the stock trading problem) but solving it with different algorithms. We modified the maximum subarray algorithm with brute-force approach to achieve $O(n^2)$ time complexity and implemented the Kadane's algorithm to solve the maximum subarray problem with $O(n)$ time complexity. To study their performance, we executed all four algorithms (from both homework 5 and 6) with a different number of stock information. In the end, we recorded the result and observe the growth of CPU time to verify our derivation of their time complexity.

2. Analysis & Implementation

2.1 Maximum Subarray Problem and Stock Trading

A maximum subarray problem is a problem that aims to maximize the sum of a subarray of an array A with size N, i.e.

$$\max_{low,high} \sum_{i=low}^{high} \mathrm{A}[i]$$

$$s.t. 1 \le low \le high \le N$$

In this homework, our objective is to solve the one-buy-one-sell stock trading problem. We try to calculate the day to buy and sell that maximize our earning, which can be transformed into a maximum subarray problem by adding “change of price” to our stock information. Therefore, the stock trading problem is equivalent to finding the maximum subarray of the “change of price” array. Notice that if we obtain the optimal range to be [2:3], it means that $(P2 - P1) + (P3 - P2) = P3 - P1$ is the maximal earning, but the actual buy date is at index 1 and the sell day is at index 3.

Index	1	2	3	4
Price	P1	P2	P3	P4
Array A (Change of price)	0	P2 – P1	P3 – P2	P4 – P3

2.2 Maximum Subarray – Modified Brute-Force

```
// Find low and high to maximize ΣA[i], low <= i <= high.
// Input: A[1 : n], int n
// Output: 1 <= low <= high <= n and max
Algorithm MaxSubArrayMBF(A, n, low, high)
{
    max := 0; // initialize
    low := 1;
    high := n;
   // try all possible A[j] and A[k]
   for j := 1 to n do {
       for k := j to n do {
```

```
12.            // difference of A[k] and A[j]
13.            sum := A[k] - A[j];
14.            // record the maximum value and range
15.            if (sum > max) then {
16.                max := sum;
17.                low := j;
18.                high := k;
19.            }
20.        }
21.    }
22.    return max;
23. }
```

In the previous homework, we solve the stock trading problem by transforming it to a maximum subarray problem. If we solve the maximum subarray problem with brute-force approach, we need to enumerate all the possible subarrays and sum up all the elements in the subarray. But what we really are interested in is the difference of the head and tail of the subarray. Thus, we modified the third layer of loop with line 13.

As we can see, the modified brute-force approach has reduced to only two layers of loop, if we count the number of times line 13 is executed, then it will be

$$\sum_{j=1}^{n}\sum_{k=j}^{n} 1 \approx \sum_{j=1}^{n}(n-j) = n^2 - \frac{(1+n)n}{2} = O(n^2)$$

As for the space complexity, we need *n* space to store the array and the function takes a constant amount of space to store the local variables.

Therefore, the overall space complexity is $\mathcal{O}(n) + \mathcal{O}(1) = \mathcal{O}(n)$.

Best-case, average and worst-case time complexity: $\mathcal{O}(n^2)$

Space complexity: $\mathcal{O}(n)$

2.3 Maximum Subarray – Kadane's Algorithm

The question states as follows: *Is there a lower complexity, lower than $\Theta(n \cdot lg\ n)$, algorithm in solving the single-buy-single-sell stock trading problem?* The answer is yes. Using the Kadane's algorithm, the maximum subarray problem (or the stock trading problem) can be solved with $\mathcal{O}(n)$ time complexity deterministically.

```
// Find low and high to maximize ΣA[i], low <= i <= high.
// Input: A[1 : n], int n
// Output: 1 <= low <= high <= n and max
Algorithm MaxSubArrayKadane(A, n, low, high)
{
    low := 1; // initialize
    high := n;
    checkpoint := 1;
    max := 0;
    sum := 0;
    for i := 1 to n do {
        sum := sum + A[i]; // add the value to sum
        // record the maximum and range
        if (sum > max) then {
            max := sum;
            low := checkpoint;
            high := i;
        }
        if (sum < 0) then {
            sum := 0; // reset
            // start adding from next index
            checkpoint := i + 1;
        }
    }
    return max;
}
```

The Kadane's algorithm records the maximum of subarray in A[1:i-1], denoted $\mathrm{M}_{\text{i-1}}$, and see if including A[i] can make the maximum of subarray in A[1:i] larger. Therefore, the update rule for finding the maximum of the subarray in A[1:i] will be $\mathrm{M}_{\mathrm{i}} = max(\mathrm{M}_{\text{i-1}}, \mathrm{M}_{\text{i-1}} + \mathrm{A[i]})$. Moreover, if we sum up the elements of an interval and the result turns out to be negative, then we should reset the interval where the summation is performed and start adding from the next element.

Since we also want to know the lower and higher index that delineate the subarray, the variable *checkpoint* records the lower index that the current summation is performed from. If the current sum is greater than current maximum, then we set the lower index *low* to *checkpoint* and the higher index *high* to current index *i* to record the interval that makes the maximum of subarray.

As we can see, Kadane's algorithm requires only one loop to solve the maximum subarray problem. Thus, it is an algorithm with $O(n)$ time complexity, which is indeed better than using divide and conquer whose time complexity is $\Theta(n \cdot lg\, n)$. Also, we need n space to store the array and constant amount of space to store the local variables. The overall space complexity is $O(n) + O(1) = O(n)$

Best-case, average and worst-case time complexity: $O(n)$

Space complexity: $O(n)$

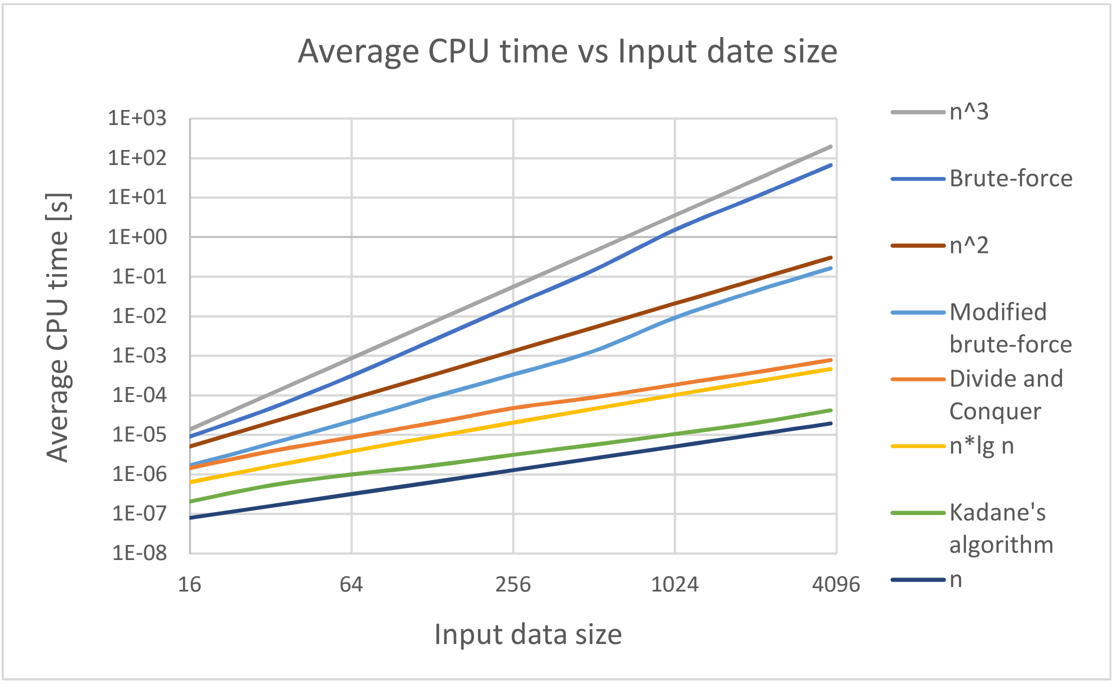
3. Result and Observation

To measure the performance, we execute the maximum subarray algorithm with brute-force approach, divide and conquer approach, modified brute-force approach and Kadane's algorithm to solve the trading stock problem with a different number of stock information. Also, the data points in the table are the average result of 1, 1000, 500 and 500 execution of the algorithms mentioned above, respectively.

N	**Brute-force approach**	**Divide and conquer approach**	**Modified brute-force**	**Kadane's algorithm**
16	8.82149e-06	1.44100e-06	1.63221e-06	2.23637e-07
32	4.69685e-05	3.59893e-06	6.14595e-06	5.44071e-07
64	3.13997e-04	8.94403e-06	2.23002e-05	1.00803e-06
128	2.47407e-03	2.03071e-05	8.55517e-05	1.67418e-06
256	1.97191e-02	4.32661e-05	3.35474e-04	3.04222e-06
512	1.46862e-01	9.37350e-05	1.32970e-03	5.62811e-06

1024	1.17791e+00	1.84481e-04	1.01268e-02	1.05720e-05
2048	9.22634e+00	3.83117e-04	4.23886e-02	1.96519e-05
3890	6.95493e+01	7.83225e-04	1.64025e-01	4.31480e-05

Table 1. Average CPU time [s] vs Input data size



As the graph shows, with a simple modification to the original brute-force approach, the stock trading problem can be solved much more efficiently than using the old one. However, it no longer uses the concept of the maximum subarray problem. Furthermore, before this homework, I thought solving the maximum subarray problem with $\Theta(n \cdot lg\, n)$ is as fast as we possibly could. But it turns out I was wrong, the Kadane's algorithm is not only more efficient, with $O(n)$ time complexity, but also a lot easier to implement, which is a mind-blowing result of this homework.