EE3980 Algorithms

Homework 6. Stock Short Selling Revisited

106061151 劉安得

1. Introduction

- Maximum Subarray Problem

In this homework, we are going to use the concept of Maximum Subarray to solve the Stock Short Selling problem. Maximum Subarray Problem is given an array A[1:n], find the range such that

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

- Minimum Subarray Problem

In Maximum Subarray Problem, we find the maximum contiguous sum. However, we can modify the problem to find the minimum contiguous sum, and the problem turn out to be Minimum Subarray Problem

- Stock Short Selling

Stock Short Selling is one-sell-one-buy trading, to be more explicit, borrow some stock to sell at a high price first and buy the same amount of stocks back at a lower price later. The price difference is then the earning he/she makes.

The stock price data can be transformed into daily price change information. Then the problem is to find the range of the subarray with the minimum contiguous sum. That is, this problem can be seen as Minimum Subarray Problem.

In hw04, we solve this problem by two different approach, Brute-Force Approach and Divide and Conquer. The time complexity of brute-force approach is $O(n^3)$ and divide and conquer is $O(n*lg(n))$.

- Modify Brute-Force Approach

However, Brute-Force Approach do summation for A[j : k] for every j and k, which cause inefficiency. Therefore, we can modify Brute-Force Approach to achieve $O(n^2)$ complexity. If we do summation $S[j] = \Sigma A[1:k]$ at first, then whenever we need $\Sigma A[j : k]$, we only need to calculate $S[k] – S[j]$. At first, we transform the stock price data to use the maximum subarray approach. However, we sum the price change again, which is the original stock price. So, we can use the stock price directly in this algorithm.

- Dynamic Programming

Moreover, we can use Dynamic Programming algorithm to get lower time complexity than the divide-and-conquer approach. Dynamic Programming (DP) is an algorithmic technique for solving an optimization problem by breaking it down into simpler subproblems and utilizing the fact that the optimal solution to the overall problem depends upon the optimal solution to its subproblems.

In this homework, we use the history of Google stock (GOOGL) closing price as input data, record the CPU times and correlate the performance to analyses.

2. Approach

A. Brute-Force Approach

```
// Find low and high to minimize A[j]-A[i], low <= i <= j <= high.
// Input: A[1 : n], int n
// Output: 1 <= low, high <= n and min.
Algorithm MinSubArrayBF(A, n, low, high)
{
    min := 0; // Initialize
    low := 1;
    high := n;
    for j := 1 to n do { // Try all possible ranges: A[j : k].
        for k := j to n do {
            sum = A[k]-A[j];
            if sum < min then { // Record the minimum value and range.
                min := sum;
                low := j;
                high := k;
            }
        }
    }
    return min;
}
```

I. Correctness

The input array A is the original stock price. the first two loop try all possible ranges and calculate the summation by A[k] – A[j]. If the minimum subarray is A[low : high], and it will be record when j = low and k = high. The comparison makes sure that the low and high variables record the index of minimum subarray.

II. Time Complexity

n is the size of Array A.

Statement	s/e	Freq.	Total steps
`Algorithm MinSubArrayBF(A, n, low, high)`	0	-	0
`{`	0	-	0

`min := 0;`	1	1	1
`low := 1;`	1	1	1
`high := n;`	1	1	1
`for j := 1 to n do {`	1	n+1	n+1
`for k := j to n do {`	1	Σ(n-j+2)	Σ(n-j+2)
`sum = A[k]-A[j];`	1	Σ(n-j+1)	Σ(n-j+1)
`if sum < min then {`	1	Σ(n-j+1)	Σ(n-j+1)
`min := sum;`	1	0~ Σ(n-j+1)	0~ Σ(n-j+1)
`low := j;`	1	0~ Σ(n-j+1)	0~ Σ(n-j+1)
`high := k;`	1	0~ Σ(n-j+1)	0~ Σ(n-j+1)
`}`	0	-	0
`}`	0	-	0
`}`	0	-	0
`return min;`	1	1	1
`}`	0	-	0
Total			

$$T(n) \le n + 5 + \sum_{j=1}^{n} 6\mathrm{n} - 6\mathrm{j} + 7$$

$$= n + 5 + \sum_{j=1}^{n} 6n - 6j + 7$$

$$= n + 5 + 6n^2 - 3n(n+1) + 7n$$

$$\approx 3n^2 = \mathrm{O}(\mathrm{n}^2)$$

III. Space Complexity

n for array *A*, one for each variable: *n*, *low*, *high*, *i*, *j*, *min* and *sum*

S = n+7 = Θ(n)

B. Dynamic Programming

```
// Find low and high to minimize A[j]-A[i], low <= i <= j <= high.
// Input: A[1 : n], int n
// Output: 1 <= low, high <= n and min.
Algorithm MinSubArray(A, n, low, high)
{
    min := 0; // Initialize
```

```
    low := 1;
    high := n;
    local_low := 0; // i which minimize A[j]-A[i] for all i <= j
    for j := 1 to n do { // loop all A[j]
        sum = A[j]-A[local_low];
        if sum < min then { // Record the minimum value and range.
            min := sum;
            low := local_low;
            high := j;
        }
        else if sum > 0 then { // if A[j]-A[local_low] > 0, for j+1,
local_low = j
            local_low = j;
        }
    }
    return min;
}
```

I. Correctness

If we know the minimum contiguous sum when j = i-1. Then for j = i, there could be two possible situations. First, A[i] is included in this subarray. Second, A[i] is excluded in this subarray. If we know the index *local_low* which minimize A[j]-A[local_low] for all j <= i-1, then we can calculate A[i]- A[local_low] and check whether it is the minimum value. And if A[i]- A[local_low] > 0, for i+1, *local_low* is i, because i is the value which minimize A[j]-A[local_low] for all j <= i.

II. Time Complexity

n is the size of Array A.

Statement	s/e	Freq.
Algorithm MinSubArray(A, n, low, high)	0	-
{	0	-
min := 0;	1	1
low := 1;	1	1

`    high := n;`	1	1
`    local_low := 0;`	1	1
`    for j := 1 to n do {`	1	n+1
`        sum = A[j]-A[local_low];`	1	n
`        if sum < min then {`	1	n
`            min := sum;`	1	0~n
`            low := local_low;`	1	0~n
`            high := j;`	1	0~n
`        }`	0	-
`        else if sum > 0 then {`	1	0~n
`            local_low = j;`	1	0~n
`        }`	0	-
`    }`	0	-
`    return min;`	1	1
`}`	0	-
Total		

$$4n + 6 \leq T(\mathrm{n}) \leq 6n + 6$$

$$\mathrm{T(n)} = \Theta(\mathrm{n})$$

III. Space Complexity

n for array *A*, one for each variable: *n*, *low*, *high*, *i*, *local_low*, *min* and *sum*

S = n+7 = Θ(n)

C. Main Function

```
// Driver function to measure MinSubArray functions.
// Input: stock closing price file
// Output: CPU time used, the day and price of selling and buying and
EPS.
Algorithm main(void)
{
    Read n and a list of n data entries and store into A array;
    t0 := GetTime();
    repeat 5000 times {
        MinSubArrayBF(A, n, low, high);
    }
    t1 := GetTime();
```

```
    t := (t1 - t0);
    write(function, t, the day and price of selling and buying, EPS);
    t0 := GetTime();
    repeat 5000 times {
        MinSubArray(A, n, low, high);
    }
    t1 := GetTime();
    t := (t1 - t0)/500;
    write(function, t, the day and price of selling and buying, EPS);
}
```

I. Keys of Implementation

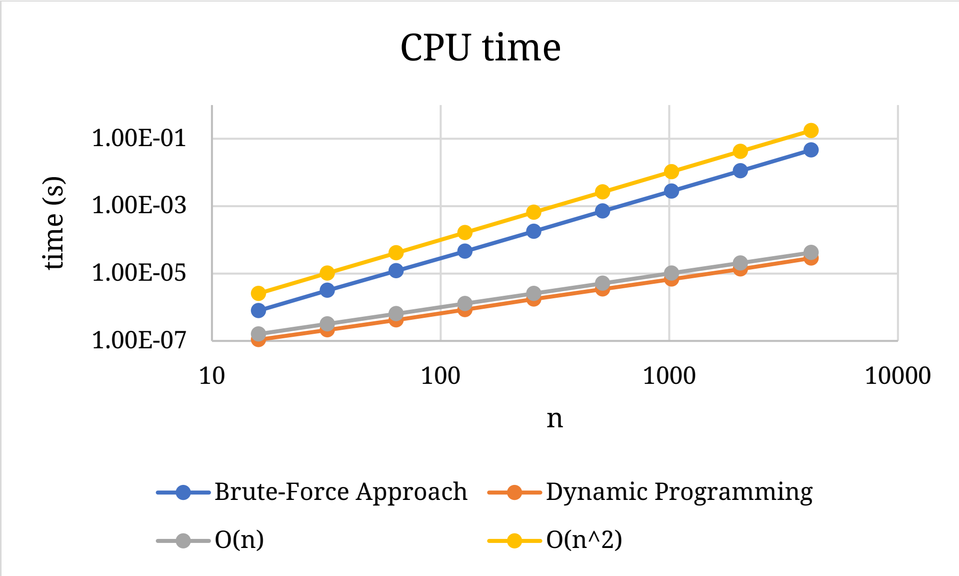
After storing the price data, we can apply minimum subarray algorithms to solve the problem.

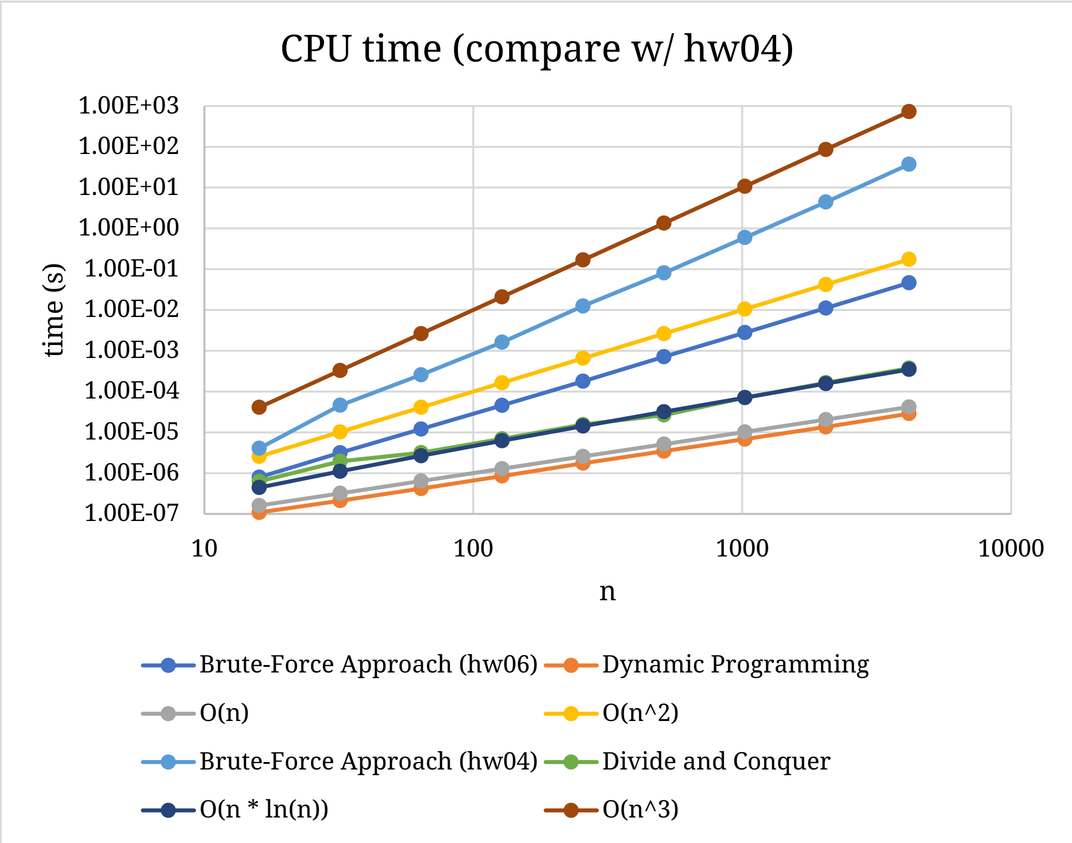
Each algorithm is executed 5000 times to get the average CPU time.

3. Result & Observations

A. CPU Time (Units: s)

N	Brute-Force Approach	Dynamic Programming	EPS
16	7.92599e-07	1.08814e-07	4.7096
32	3.15442e-06	2.11239e-07	4.7096
64	1.2043e-05	4.18186e-07	14.1286
128	4.57988e-05	8.46815e-07	15.5129
256	0.000178397	1.74384e-06	20.0269
512	0.000712084	3.4584e-06	67.4934
1024	0.00277779	6.8192e-06	164.5931
2048	0.0111429	1.35456e-05	242.9249
4178	0.0463519	2.8688e-05	470.7400





B. Conclusions/ Observations

The result of analysis matches the plot, the time complexity of brute-force approach is $O(n^2)$, and the time complexity of dynamic programming is O(n). Including hw04, Dynamic Programming is the fastest in all four algorithms. And it does not use a lot of space like divide and conquer.