EE3980 Algorithms

hw05 Trading Stock

106061146 陳兆廷

Introduction:

In this homework, I will be analyzing, implementing, and observing 2 algorithms. 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 ~~price~~ 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 my ubuntu with linux kernel.

Lastly, the observation of them will be focusing on the time complexity of the results from implementation. Moreover, I will compare 2 algorithms with each other and make some rankings. Finally, I will check the experimented results with my analysis.

Analysis:

1. **What is a Max Subarray?**

   The max subarray of an array is the subarray that has the maximum value. The definition of “value” can be altered in different cases. In this case, the value of a subarray is the difference between the first item and the last item of the subarray, since it represents the act that a person is buying and selling stocks.

2. **Brute-force approach**

   The brute-force approach means that the approach of solving a problem is rather direct and straight-forward. Mostly it involves finding every combination and testing all possible solutions for this problem. It is more time-consuming but is rather easy way to think of when solving a problem.

3. **Divide and Conquer**

   “Divide and conquer” is a technique of solving a problem. Given an input set P, this technique breaks the input into k distinct subsets, which forms k subproblems, and solve them individually. Finally, it combines all sub-solutions into a solution for the original problem.

4. MaxSubArrayBF (brute-force)

a. Abstract:

MaxSubArrayBF 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 changes between each share of stocks. And there is a minor change in the algorithm when adapting *MaxSubArrayBF* to stock problem. I must ignore the first change of the stock when counting since it does not count as the gained or lost profit.

b. Algorithm:

```
// 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 MaxSubArrayBF(A, n, low, high)
{
    max := 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 := 0 ;
            for i := j + 1 to k do { // Summation for A[j + 1 : k ]
                sum := sum + A[i ] ;
            }
            if (sum > max) then { // Record the maximum value and range.
                max := sum ;
                low := j ;
                high := k ;
            }
        }
```

```
21.     }
22.     return max ;
23. }
```

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
`1. Algorithm MaxSubArrayBF(A, n, low, high)`	0	0	0
`2. {`	0	0	0
`3.     max := 0 ;`	1	1	1
`4.     low := 1 ;`	1	1	1
`5.     high := n ;`	1	1	1
`6.     for j := 1 to n do {`	N	1	N
`7.         for k := j to n do {`	N	N	N^2
`8.             sum := 0 ;`	1	N^2	N^2
`9.             for i := j + 1 to k do {`	N/2	N^2	1/2N^3
`10.                sum := sum + A[i ] ;`	1	1/2N^3	1/2N^3
`11.            }`	0	1/2N^3	0
`12.            if (sum > max) then {`	1	N^2	N^2
`13.                max := sum ;`	1	N^2	N^2
`14.                low := j ;`	1	N^2	N^2
`15.                high := k ;`	1	N^2	N^2

`}`	0	N	0
`}`	0	0	0
`}`	0	0	0
`return max ;`	1	1	1
`}`	0	0	0
p.s. N/2 sine it goes through 1 ~ N in N iterations	N^3 + 6N^2 + 4		

The time complexity of *MaxSubArrayBF* is O(N^3), 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 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 7 integers and N pairs of elements in array of stocks. **The space complexity would be O(N).**

5. MaxSubArray (Divide and Conquer)

a. Abstract:

MaxSubArray divides the searching process into 3 parts, first half, second half and combination that crosses the boundary, which is the share in the middle. It involves *MaxSubArrayXB* for the third part.

The A[i] in the below algorithm indicates the changes between each share of stocks. And there is a minor change in the algorithm when adapting *MaxSubArray* to stock problem. I must ignore the first change of the stock when counting since it does not count as the gained or lost profit.

b. Algorithm:

```
// Find low and high to maximize ΣA[i ], begin  low  i  high  end.
// Input: A, int begin  end
// Output: begin  low, high  end and max.
Algorithm MaxSubArray(A, begin, end, low, high)
{
    if (begin = end) then { // termination condition.
        low := begin ; high := end ;
        return 0;
    }
    mid := ⌊(begin + end)/2⌋ ;
    lsum := MaxSubArray(A, begin, mid, llow, lhigh) ; // left region
    rsum := MaxSubArray(A, mid + 1, end, rlow, rhigh) ; // right region
    xsum := MaxSubArrayXB(A, begin, mid, end, xlow, xhigh) ; // X region
    if (lsum >= rsum and lsum >= xsum) then { // lsum is the largest
        low := llow ; high := lhigh ;
        return lsum ;
    }
    else if (rsum >= lsum and rsum >= xsum) then { // rsum is the largest
        low := rlow ; high := rhigh ;
        return rsum ;
    }
    low := xlow ; high := xhigh ;
    return xsum ; // cross-boundary is the largest
}
```

```
// Find low and high to maximize ΣA[i], begin <= low <= mid <= high <= end.
// Input: A, int begin <= mid <= end
// Output: low <= mid <= high and max.
```

```
Algorithm MaxSubArrayXB(A, begin, mid, end, low, high)
{
    lsum := 0 ; // Initialize for lower half.
    low := mid ;
    sum := 0 ;
    for i := mid to begin + 1 step −1 do { //find low to maximize ΣA[low : mid]
        sum := sum + A[i ] ; // continue to add
        if (sum > lsum) then { // record if larger.
            lsum := sum ;
            low := i ;
        }
    }
    rsum := 0 ; // Initialize for higher half.
    high := mid + 1 ;
    sum := 0 ;
    for i := mid + 1 to end do { // find end to maximize ΣA[mid + 1 : high ]
        sum := sum + A[i ] ; // Continue to add.
        if (sum > rsum) then { // Record if larger.
            rsum := sum ;
            high := i ;
        }
    }
    return lsum + rsum ; // Overall sum.
}
```

c. Proof of correctness:

In this algorithm, it calculates the profits for every possible combination of buying and selling this stock by calculating the best solution for the first half, second half and *MaxSubArrayXB*. In *MaxSubArrayXB*, it returns a best solution that contains *mid* through testing all solutions. Using induction, we can conclude that in every level of *MaxSubArray*, it finds the best solution among the 3 parts. *MaxSubArray* terminates when *begin* == *end*.

d. Time complexity:

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

	s/e	freq	total
`Algorithm MaxSubArrayXB(A, begin, mid, en`	0	0	0
`d, low, high)`	0	0	0
`{`	0	0	0
`lsum := 0 ;`	1	1	1
`low := mid ;`	1	1	1
`sum := 0 ;`	1	1	1
`for i := mid to begin step -1 do {`	c	1	c
`sum := sum + A[i ] ;`	1	c	c
`if (sum > lsum) then {`	1	c	c
`lsum := sum ;`	1	c	c
`low := i ;`	1	c	c
`}`	0	c	0
`}`	0	c	0
`rsum := 0 ;`	1	1	1
`high := mid + 1 ;`	1	1	1
`sum := 0 ;`	1	1	1
`for i := mid + 1 to end do {`	c	1	c
`sum := sum + A[i ] ;`	1	c	c
`if (sum > rsum) then {`	1	c	c
`rsum := sum ;`	1	c	c
`high := i ;`	1	c	c
`}`	0	c	0
`}`	0	c	0
`return lsum + rsum ;`	1	1	1
`}`	0	0	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 $T_{XB}(n)$ is time complexity of *MaxSubArray*XB, we can imply this:

$$T(n) = 2T\left(\frac{n}{2}\right) + T_{XB}(n)$$

Then, assuming $n = 2^k$, and $T_{XB}(n) = n$ is known:

$$T(2^k) = 2T(2^{k-1}) + n$$

$$T(2^k) = 2(2T(2^{k-2}) + n/2) + n$$

$$T(2^k) = 2^k\left(T(2^{k-k})\right) + kn$$

Then change k back to $\lg n$:

$$T(n) = n + n\lg n$$

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

<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 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.

e. Space Complexity:

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

6. 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.

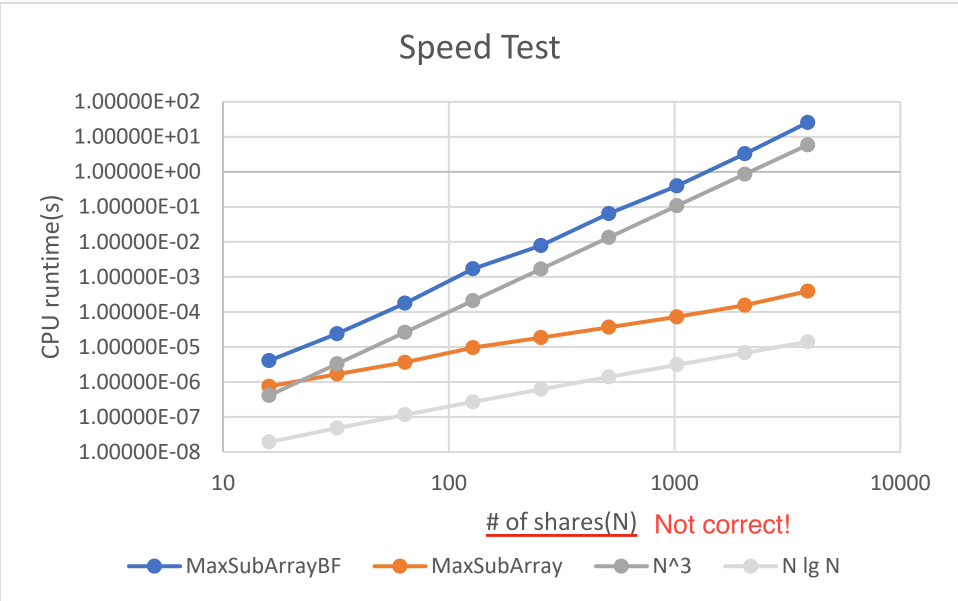
Workflow :

```
t_MaxSubArrayBF = GetTime();                // initialize time counter
MaxSubArrayBF();
t_MaxSubArrayBF = GetTime() - t_MaxSubArrayBF;  // calculate CPU time
t_MaxSubArray = GetTime();                  // initialize time counter
for i := 0 to 1000 do {
   MaxSubArray();
}
t_MaxSubArray = (GetTime() - t_MaxSubArray) / 1000; // calculate CPU time
```

Results:

Time unit?

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)

2. Actual runtime comparison:

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

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

hw05.c

```
// EE3980 HW05 Trading Stock
// 106061146, Jhao-Ting, Chen
// 2020/04/08

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/time.h>

typedef struct sSTKprice {                   // Storing stock shares
    int year, month, day;
    double price, change;
} STKprice;

typedef struct sMaxArray {                   // return found value
    int low, high;
    double change;
}MaxArray;
} MaxArray;

int R = 1000;                                // # of test cycles
int N;                                       // # of stock shares
STKprice *data;                              // Stock list

void readInput(void);                        // read all inputs
void printInput(void);                       // print Input
double GetTime(void);                        // get local time in seconds
MaxArray MaxSubArrayBF(STKprice *A, int N);            // 3 test algorithms
MaxArray MaxSubArray(STKprice *A, int begin, int end);
Need comments
MaxArray MaxSubArrayXB(STKprice *A, int begin, int mid, int end);
Need comments


int main(void)
{
    int i;                                   // loop index
    double t;                                // record CPU time
    MaxArray ans;                            // returned value

    readInput();                             // read input graph
//    printInput();
    printf("N = %d\n", N);

    t = GetTime();                           // initialize timer
    ans = MaxSubArrayBF(data, N);

    printf("Brute-force approach: time %e s\n", (GetTime() - t));   // result
```

```
    printf("  Buy: %d/%d/%d at %g\n", data[ans.low].year, data[ans.low].month,
            data[ans.low].day, data[ans.low].price);
    printf("  Sell: %d/%d/%d at %g\n", data[ans.high].year,
            data[ans.high].month, data[ans.high].day, data[ans.high].price);
    printf("  Earning: %g per share.\n", ans.change);

    t = GetTime();                          // initialize time counter
    for (i = 0; i < R; i++) {               // Connect1 testing
        ans = MaxSubArray(data, 0, N-1);
        ans = MaxSubArray(data, 0, N - 1);
    }

    printf("Divide and Conquer: time %e s\n", (GetTime() - t) / R);
    printf("  Buy: %d/%d/%d at %g\n", data[ans.low].year, data[ans.low].month,
            data[ans.low].day, data[ans.low].price);
    printf("  Sell: %d/%d/%d at %g\n", data[ans.high].year,
            data[ans.high].month, data[ans.high].day, data[ans.high].price);
    printf("  Earning: %g per share.\n", ans.change);


    return 0;
}

void readInput(void)                              // read all inputs
{
    int i;                              // for looping and dynamic store

    scanf("%d\n", &N);                  // read # of Vertices and Edges

    data = (STKprice *)calloc(N, sizeof(STKprice));


    for (i = 0; i < N; i++) {                   // Store shares in data
        scanf("%d %d %d %lf\n", &data[i].year, &data[i].month,
                                &data[i].day, &data[i].price);  // read shares
    }

    data[0].change = 0.0;                       // store each share's change
    for (i = 1; i < N; i++) {
        data[i].change = data[i].price - data[i - 1].price;
    }
}

void printInput(void)                            // print stock list
{
    int i;
    for (i = 0; i < N; i++) {
        printf("%d %d %d %lf %lf\n", data[i].year, data[i].month, data[i].day,
                                    data[i].price, data[i].change);
    }
```

```
}

double GetTime(void)                    // demonstration code from 1.1.3
{
    struct timeval tv;

    gettimeofday(&tv, NULL);
    return tv.tv_sec + 1e-6 * tv.tv_usec;
}

MaxArray MaxSubArrayBF(STKprice *A, int N)
{
    double max = 0;                 // initialize
    double sum;
    int low = 0;
    int high = N - 1;
    int i, j, k;

    MaxArray ans;

    for (j = 0; j < N; j++) {       // Try all possible ranges: A[j : k].
        for (k = j; k < N; k++) {
            sum = 0;
            for (i = j + 1; i <= k; i++) {  // Summation for A[j : k]
                sum = sum + A[i].change;
            }
            if (sum > max) {            // Record the maximum value and range
                max = sum;
                low = j;
                high = k;
            }
        }
    }
    ans.low = low;
    ans.high = high;
    ans.change = max;
    return ans;
}

MaxArray MaxSubArray(STKprice *A, int begin, int end)
{
    int mid;                    // terminate condition
    MaxArray ans;
    if (begin == end) {
        ans.low = begin;
        ans.high = end;
        ans.change = 0;
        return ans;
    }
    mid = (begin + end) / 2;
```

```
    MaxArray lsum, rsum, xsum;
```

Do not mix declarations with statements

```
    lsum = MaxSubArray(A, begin, mid);      // left region
    rsum = MaxSubArray(A, mid + 1, end);    // right region
    xsum = MaxSubArrayXB(A, begin, mid, end);   // cross obundary
    if ((lsum.change >= rsum.change) && (lsum.change >= xsum.change)) {
        ans.low = lsum.low;                     // lsum is the largest
        ans.high = lsum.high;
        ans.change = lsum.change;
        return ans;
    } else if ((rsum.change >= lsum.change) && (rsum.change >= xsum.change)) {
        ans.low = rsum.low;                     // rsum is the largest
        ans.high = rsum.high;
        ans.change = rsum.change;
        return ans;
    }
    ans.low = xsum.low;                         // cross-boundary is the largest
    ans.high = xsum.high;
    ans.change = xsum.change;
    return ans;
}

MaxArray MaxSubArrayXB(STKprice *A, int begin, int mid, int end)
{
    double lsum, sum, rsum;         // initialize
    int i;
    int low, high;
    MaxArray ans;

    low = mid;                      // initialize for lower half
    lsum = 0.0;
    sum = 0.0;

    for (i = mid; i > begin; i--) {     // find low to maximize A[low : mid]
        sum = sum + A[i].change;        // continue to add
        if (sum > lsum) {               // record if larger
            lsum = sum;
            low = i;
        }
    }

    rsum = 0.0;                         // Initiallize for higher half
    high = mid + 1;
    sum = 0.0;

    for (i = mid + 1; i <= end; i++) {  // find end
        sum = sum + A[i].change;        // continue to add
        if (sum > rsum) {               // record if larger
            rsum = sum;
            high = i;
```

```
        }
    }

    ans.low = low - 1;
    ans.high = high;
    ans.change = lsum + rsum;           // overall sum
    return ans;
}
```

[Program Format] can be improved.
[Writing] hw05a.pdf spelling errors: perfomance(1)
[Coding] hw05.c spelling errors: Initiallize(1), obundary(1)
[Space] complexities of two algorithms can be compared.
[Report] errors can be corrected.
[Good] effort in writing the homework!

Score: 90