EE3980 Algorithm

HW4. Stock Short Selling

106061225 楊宇羲

1. Problem Description:

When it comes to buying and selling stocks, everyone would wish to make money by selling at a high price while buying it at a low price. Being provided with a chart with everyday's price on, we have to be able to figure out the max value array to maximize the profit we get. So what is maximum sub array? Given an array with size n, we have to choose two integer *i, j* in the range of n such that the sum from A[i] to A[j] gets the biggest value. However, to solve the max sub-array problem directly involves tedious loop calculation that requires three layers of for loop since setting the start point and end point already brings us $C_2^n = \frac{n(n-1)}{2}$ combinations, where n is the dataset size. After adding the implementation of calculating sum ,the total process step can be unbelievably cumbersome, the detailed steps will be discussed in analysis. As the dataset size are growing bigger, the process time will grow to an extremely large size as well (in accurate, at the time complexity of $\Theta(n^3)$). Hence there is a necessity to do some improvement toward the method of calculating subarray problem like this, thus the divide & conquer approach are introduced here with a significant decrease in processing time.

2. Approach:

First of all, instead of merely returning the max value, I choose to create a type of structure that contains the max value ,start point and end point, so that it will be easier to do the recursive part and there will be no need to initialize global variable besides the data array and its size. For doing analysis, we will use the GetTime to record the brute-force method for one time (since performing hundreds time can take more than an hours to run), and set repetition time to 1000 to run the divide & conquer approach, then there are 9 datasets with each dataset the size growing exponentially, but note that we are going to compare with data's change so that we make sure to get max profit, which is *data[i] - data[i-1]*, so after we obtain the result we have to shift the selling day one day earlier. We are then required to carry out the two methods to find the correct max sub array to create the maximized profit and record the process time simultaneously. After the two methods are all completed, we can compare the run time of the former one and the average time of the latter one ,then combine it to a line chart not only to compare the speed of them but also observe the

growing tendency of the two methods while the dataset size are growing exponentially in the power of 2. The main function's pseudo code and some notices are provided below:

a) We are going to find the value that has the largest absolute value with negative sign since we will sell first then buy it, so we are going to seek for the minimum value to create max profit.

b) We are comparing each data's change value, so as we found the start point and end point we have to minus 1 to start point so that it provides the correct data.

```
Algorithm main()
{
        ReadData();                          // read the date and the price of the corresponding date
        t := GetTime();                      // start counting time
        result := SubArrayBT();              // implement brute force approach
        t := GetTime() - t;                  // stop counting time
        Write (t, result);                   // print outcome
        t := GetTime();                      // start counting time
        for i := 1 to Repeat do{             // repeat R times
                result := SubArray();        // implement divide & conquer method
        }
        t := GetTime() - t;                  // stop counting time
        Write (t/Repeat, result);            // print outcome
}
```

3. Analysis:

As mentioned in the problem description, the pseudo code of brute force approach is:

```
Algorithm MaxSubArrayBT (data, n)
{
        result.max := 0;
        result.sell := 1;
        result.buy := n;
        for i := 1 to n do {
                for j := i to n do {
                        sum := 0;
                        for k := i to j do {
                                sum := sum + data[k];
                        }
                        if sum < result.max then {
                                result.max := sum;
                                result.sell := i;
                                result.buy := j;
                        }
                }
```

```
        }
        return result;
}
```

Where result structure are initialized as:

Struct *result*{
        int *sell, buy;*
        double *max;*
};

The outer two loop determines the position of start index and end index. With all the combination possibilities, we calculate sum of each of them, then return the start point and end point that creates the max value. This is the most direct thought to carry out the result, however, the total execute steps are going to be $\frac{n(n-1)}{2}$ times the average steps to calculate sum, which is n time. The difference between best case and worst case is not obvious since the algorithm **always** runs through all the combinations and calculate the sum, so we can obtain the total time complexity to be $\Theta(n^3)$. So as we are seeking s1.dat's result it may seem to be not that slow, but when we are carrying out s9.dat you can just leave your computer and take a rest due to long time of waiting. The total space complexity will be O(n), which is the size of array and variables for loop, sum, sell & buy.

On the other hand, the coding of divide & conquer method could be more difficult than the former one since it requires recursive solution and cross boundary condition, the pseudo code will look like this:

Struct *result*{
        int *sell, buy;*
        double *max;*
};

```
Algorithm MaxSubArray (data, begin, end)
{
        if begin = end then{                              // terminal condition
            result.sell := begin;
            result.buy := end;
            result.max := data[begin];
            return result;
        }
        mid := (begin + end) / 2;
        lsum := MaxSubarray(data, begin, mid);           // left half sum
        rsum := MaxSubarray(data, mid +1, end);          // right half sum
        xsum := MaxSubarrayXB(data, begin, mid, end);    // sum that contains mid
```

```
        if lsum <=rsum and rsum <= xsum then return lsum;          // left is max
        else if rsum <=lsum and rsum <= xsum then return rsum;   // right is max
        return xsum;                                              // cross boundary is max
}
```
And the cross boundary function will be defined like this:

Algorithm *MaxSubArrayXB* (*data, begin, mid, end*)
```
{
        leftsum := 0;                          // initialize the lower half
        result.sell := mid;                    // initialize the sell date
        sum := 0;                              // initialize sum
        for i := mid to begin step -1 do {     // start from mid
                sum := sum + data[i];
                if  sum > leftsum then {
                        leftsum := sum;        // obtain left half max
                        result.sell := i;      // obtain sell date
                }
        }
        rightsum := 0;                         // initialize upper half
        result.buy := mid + 1;                 // initialize buy date
        sum := 0;                              // initialize sum
        for i := mid + 1 to end do {           // start from mid
                sum := sum + data[i];
                if  sum > rightsum then {
                        rightsum := sum;       // obtain right half max
                        result.buy := i;       // obtain buy date
                }
        }
        result.max := rightsum + leftsum;      // put it together
        return result;                         // return
}
```


The whole algorithm works like this: Every time it calls the recursive function, we will compare the left part sum, right part sum, and the sum that may cross the boundary. As it calls until the size are divided to 1, it returns its own change value, then calls back until the first layer of recursive function has the result. After the terminal condition is triggered, we can then compare the *lsum, rsum, xsum* of each layer, so that the dominating calculation is reduced, replaced by more comparisons. Each layer's function's comparisons contains 2 side function and 1 cross boundary function, which implies that the total comparisons of an entire recursive function can be obtained as follow with some assumed premises:

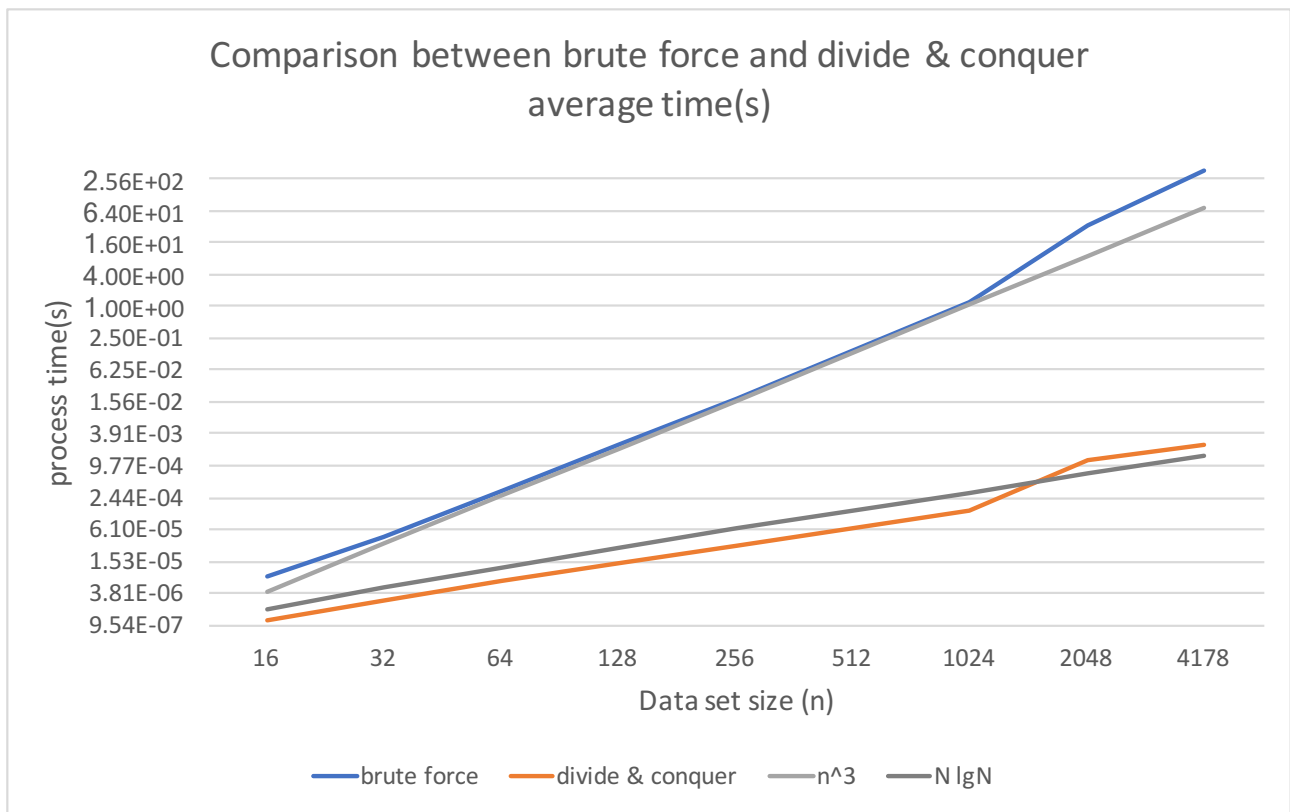(a let the total dataset size n represented as a power of 2: 2^k

(b the total comparison per one cross boundary function will be n time since it only calculates the sum with a given range.

$$2 \cdot T(\frac{n}{2}) + T_{xb}(n)$$

$$= 2 \cdot (2 \cdot T(\frac{n}{4}) + T_{xb}(\frac{n}{2})) + n$$

$$= 2 \cdot (2 \cdot T(\frac{n}{4}) + \frac{n}{2}) + n$$

$$= 4 \cdot (2 \cdot T(\frac{n}{8}) + \frac{n}{4}) + 2n$$

$$= \ldots$$

$$= 2^k \cdot T(1) + k \cdot n$$

$$= n + n \cdot log(n)$$

Since the above derivation is based on the average case, so we can get the final result of the time complexity of divide conquer approach to be $\Theta(n \cdot \lg n)$. The total space complexity will be O(n), which also varies linearly with the increase of size of array.

4. Results:

dataset size(n)	best sell date	sell price (USD)	best buy date	buy price (USD)	maximum earning per share (USD)	brute force process time (s)	divide & conquer average time(s)
16	2004/8/23	54.8694	2004/9/3	50.1598	4.7096	8.10E-06	1.12E-06
32	2004/8/23	54.8694	2004/9/3	50.1598	4.7096	4.31E-05	2.83E-06
64	2004/11/1	98.3185	2004/11/10	84.1899	14.1286	3.15E-04	6.29E-06
128	2004/11/1	98.3185	2004/11/22	82.8056	15.5129	2.34E-03	1.44E-05
256	2005/7/21	157.4561	2005/8/22	137.4292	20.0269	1.86E-02	3.10E-05
512	2006/1/10	236.5452	2006/3/13	169.0518	67.4934	1.47E-01	6.63E-05
1024	2007/11/6	372.0435	2008/3/10	207.4504	164.5931	1.15E+00	1.44E-04
2048	2007/11/6	372.0435	2007/11/24	129.1186	242.9249	3.29E+01	1.20E-03
4178	2020/2/19	1524.8700	2020/3/23	1054.1300	470.7400	3.64E+02	2.40E-03

Comparison between brute force and divide & conquer average time(s)

Y-axis (process time(s)): 2.56E+02, 6.40E+01, 1.60E+01, 4.00E+00, 1.00E+00, 2.50E-01, 6.25E-02, 1.56E-02, 3.91E-03, 9.77E-04, 2.44E-04, 6.10E-05, 1.53E-05, 3.81E-06, 9.54E-07

X-axis (Data set size (n)): 16, 32, 64, 128, 256, 512, 1024, 2048, 4178

Legend: brute force, divide & conquer, n^3, N lgN

5. Observation and conclusion:

As seen in the result, the actual speed and the tendency line are mostly matched with the theorem and my analysis (n^3 and $n \cdot lg(n)$). Hence it is proved that divide & conquer method can improve the performance when solving max sub array problem.

	space complexity	time complexity
brute force	O(n)	$\Theta(n^3)$
divide & conquer	O(n)	$\Theta(n \cdot lg(n))$