EE3980 Algorithm
HW6. Stock Short Selling Revisited
106061225 楊宇羲

1. Problem Description:

In homework 4, we have discussed how the maximum sub array is found. Here are some introduction about how we can maximum the stock selling profit: Our strategy is to sell stocks at high price and buy it at low price. Thus, we just have to find a large price followed by a minimum value to create max profit. 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. 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. We previously solved the brute force solution by finding max sum between a variating range of the array, which is $C_2^n = \frac{n(n-1)}{2}$ combinations. However, is this homework we will revise the brute-force version to calculation-free version, which means that we don't have to calculate the sum. We will merely just pick one of the C_2^n combinations such that the head price minus the tail price brings the biggest price difference (with a minus sign). The most important part of this assignment is to come up with an idea to solve the max sub array problem with process time less than the divide & conquer method. To accomplish this requirement, we have to find a method with time complexity that is smaller than $O(n\lg n)$. After doing some research and some brainstorming, I decided to apply a dynamic programming approach (Kadane's algorithm) to solve the problem.

2. Approach:

First we will read in all the needed data, including repetition time *R*, data size *N*, and store the dates, price in the structure STKprice with everyday's change calculated. When all data are set, we can then start the timer and execute the MaxSubArrayBT(), MaxSubArrayBTI() in order with only one execution since it will take forever to finish 5000 executions. In this homework, we are actually going to use the *change* in the structure to carry out the maximum sub array problem instead of the improved version of brute force approach, since we only have to find the minimum price value followed by a large enough value. After the first two algorithms' time are recorded and printed, we can then execute the remaining two algorithms for 5000 times, which are divide & conquer and the dynamic program methods, then print the outcome as well.

Here are some specialized declaration of structure to complete this assignment:

```
Typedef struct STKprice {
        int year, month, day;
        double price, change;
}STKprice;

Typedef struct RSLT {
        int sell, buy;
        double min;
}RSLT;
```

The *STKprice* is same as the homework4, and the *RSLT* is for storing all the informations that are going to be returned, including sell date, buy date and the actual price difference (stated as *min* since we are looking for the max value with a negative sign). We also have to note that in the function *PrintData()* we have to shift the sell date one day earlier since the *sell* value we returned is the change of day(*sell)* - day(*sell* - 1), so the starting date is actually the day before *sell*. The min value also have to be displayed as the absolute value so we have to add a minus sign when displaying it.

```
Algorithm main(void)
{
        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
        result := SubArrayBTI();          // implement improved 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
        t := GetTime();                   // start counting time
        for i := 1 to Repeat do{          // repeat R times
                result := SubArrayDP();   // implement dynamic programming 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 improved brute force approach is:

```
// Input: global array data[n]
// Output: result.min, result.sell, result.buy
// Solve the max sub array problem by checking possible combinations' difference
Algorithm MaxSubArrayBTI (void)
{
        result.min := 0;                                    // result initializations
        result.sell := 1;
        result.buy := n;
        for i := 1 to n do {                                // pick head index
                for j := i to n do {                        // pick tail index
                        if (data[j] - data[i] < result.min) {   // compare difference
                                result.min = data[j] - data[i]; // update new minimum
                                result.sell = i + 1;        // update new sell date
                                result.buy = j;             // update new buy date
                        }
                }
        }
        return result;
}
```

In comparison with the original version of brute force method, the improved version abandoned the sun variable, which means that there are only two loops left, which will significantly increase the performance as $O(n^3)$ is revised to $O(n^2)$. In this algorithm, we merely just check the C_2^n combinations and see which combination can create the most negative value when the front element minus the back element, which brings the time complexity of $O(n^2)$ and space complexity of $O(n)$ since there are only target array with size n and variables for loop and storing result.

On the other hand, the new introduced method's pseudo code is provided as below:

```
// Input: global array data[n]
// Output: result.min, result.sell, result.buy
// Solve the max sub array problem by Kadane's algorithm
Algorithm MaxSubArrayDP (void)
{
        sell = 1;                                           // temporary sell date
        localmin = 0;                                       // temporary minimum
        result.min := 1000000;                              // set absolute min to ∞
        result.sell := 1;                                   // initializations
```

```
    result.buy := n;
    for i := 1 to n do {                          // find min(localmin + data[i], data[i])
        if (data[i] < localmin + data[i]) {
            localmin = data[i];                   // update localmin
            sell = i;                             // update temporary sell date
        }else {
            localmin = data[i] + localmin;        // update localmin
        }
        if (localmin < result.min) {              // find min(localmin, result.min)
            result.min = localmin;                // update result.min
            result.buy = i;                       // update buy
            result.sell = sell;                   // update sell
        }
    }
    return result;
}
```

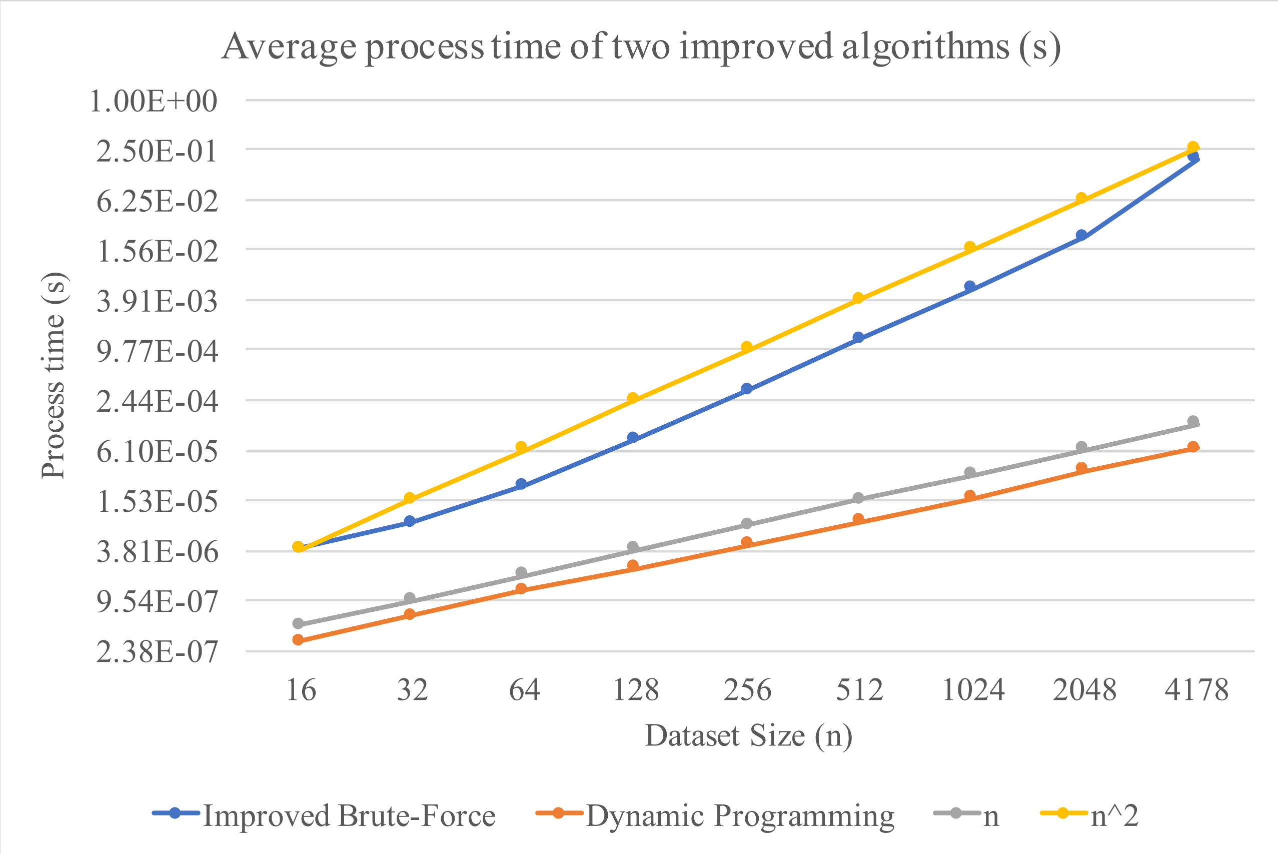
It is kind of unbelievable when I first found this $O(n)$ algorithm since we have been focused on the n^3, n^2 methods so think that the $n \cdot lg(n)$ method will be the furthest optimization. However, we can find it interesting and effective when realized how this algorithm work. First of all, we need to have a concept that "if we know the max sub array in the range 1 to (i - 1), then we can know the max sub array in the range 1 to i". To prove the correctness of the algorithm, we define *Maxsum(i)* as the max sub array that contains index *i*, which is Max{sum(1, i), sum(2, i), sum(3, i), ···, sum(i, i)}, as we found *Maxsum(i)*, we can also find *Maxsum(i* + 1*)* since it will definitely contain element *i* + 1, so it has only two possibilities which is element (*i* + 1) or *Maxsum(i)* + element (*i*+1). Hence we can split into two situations to discuss :

a) *Maxsum(i* + 1*)* = element (*i* + 1): sell date has to temporarily set to *i* + 1, then we compare *Maxsum(i* + 1*)* to the current absolute max value, if the former is larger then we update the sell date and the buy date.

b) *Maxsum(i* + 1*)* = element (*i* + 1) + *Maxsum(i* + 1*)*: we compare it to the current absolute max value directly and update the newest sell date and buy date.

After updating the current maximum value we can continue to check if *Maxsum(i + 2)* is larger or not. Eventually we got the time complexity of $O(n)$ since we just run through the array for one time, and the space complexity of $O(n)$ also because there are only target array with size *n* and variables for loop, local sell position and local minimum value.

4. Results:

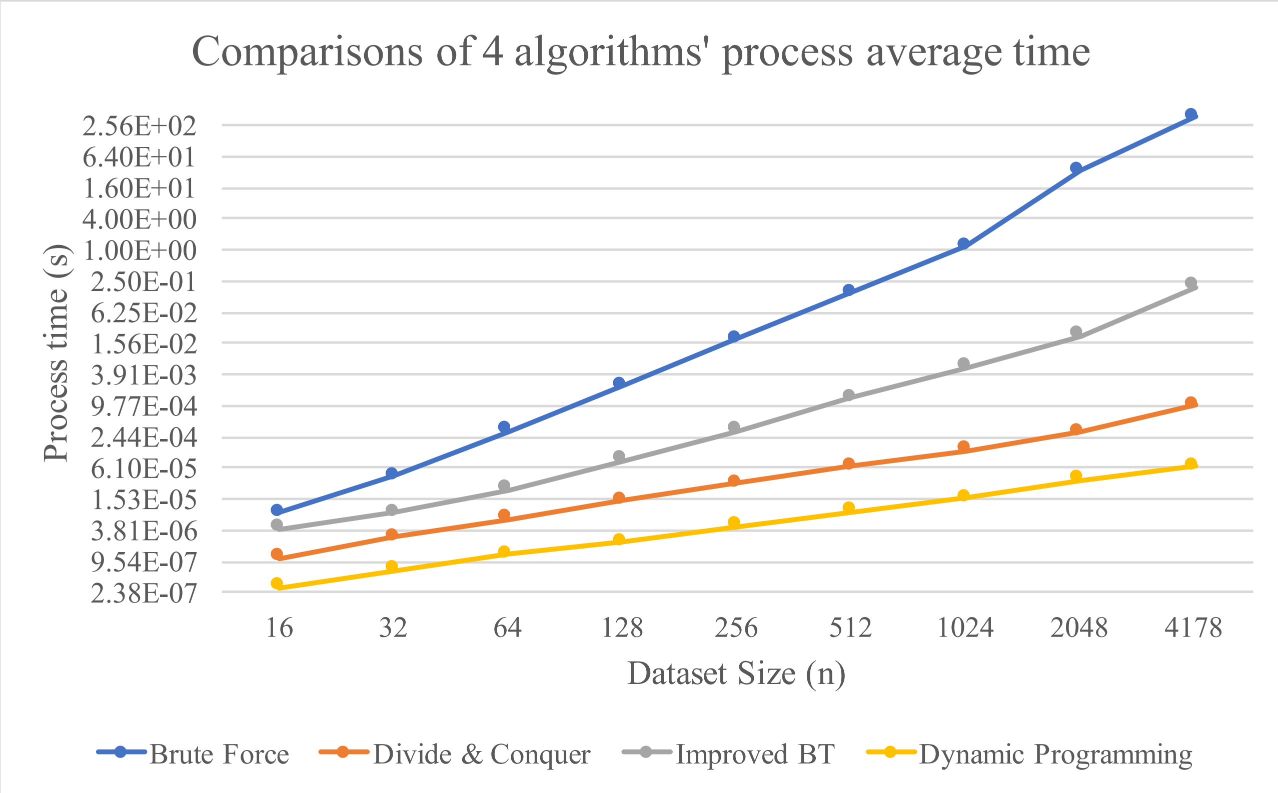


dataset size(n)	best sell date	sell price (USD)	best buy date	buy price (USD)	maximum earning per share (USD)	improved brute-force (s)	dynamic programming (s)
16	2004/8/23	54.8694	2004/9/3	50.1598	4.7096	4.05E-06	3.04E-07
32	2004/8/23	54.8694	2004/9/3	50.1598	4.7096	8.11E-06	6.30E-07
64	2004/11/1	98.3185	2004/11/10	84.1899	14.1286	2.29E-05	1.28E-06
128	2004/11/1	98.3185	2004/11/22	82.8056	15.5129	8.39E-05	2.29E-06
256	2005/7/21	157.4561	2005/8/22	137.4292	20.0269	3.20E-04	4.40E-06
512	2006/1/10	236.5452	2006/3/13	169.0518	67.4934	1.33E-03	8.46E-06
1024	2007/11/6	372.0435	2008/3/10	207.4504	164.5931	5.30E-03	1.59E-05
2048	2007/11/6	372.0435	2007/11/24	129.1186	242.9249	2.22E-02	3.50E-05
4178	2020/2/19	1524.87	2020/3/23	1054.13	470.74	1.96E-01	6.45E-05

This is the final result of this homework's revised algorithms. We can see that the result is closely matched to our analysis and the correctness of the algorithm got no problem as well.

5. Observation and conclusion:

The below figure and table is the overall comparisons of the four algorithms, and we can observe that the improved version in this homework bring significant impact on performance.



dataset size(n)	brute force process time (s)	divide & conquer average time (s)	improved brute-force (s)	dynamic programming (s)
16	8.10E-06	1.12E-06	4.05E-06	3.04E-07
32	4.31E-05	2.83E-06	8.11E-06	6.30E-07
64	3.15E-04	6.29E-06	2.29E-05	1.28E-06
128	2.34E-03	1.44E-05	8.39E-05	2.29E-06
256	1.86E-02	3.10E-05	3.20E-04	4.40E-06
512	1.47E-01	6.63E-05	1.33E-03	8.46E-06
1024	1.15E+00	1.30E-04	5.30E-03	1.59E-05
2048	3.29E+01	2.90E-04	2.22E-02	3.50E-05
4178	3.64E+02	1.00E-03	1.96E-01	6.45E-05

The last table shows each algorithms space complexity and time complexity, and it shows that the divide & conquer and the dynamic programming methods ace on performance.

	space complexity	time complexity
Brute force	$O(n)$	$\Theta(n^3)$
Improved brute force	$O(n)$	$\Theta(n^2)$
Divide & conquer	$O(n)$	$O(n \cdot lg(n))$
Dynamic programming	$O(n)$	$O(n)$

```
$ gcc hw06.c
$ a.out < s7.dat
N = 1024
Brute-force approach: CPU time 0.467717 s
  Sell: 2007/11/6 at 372.0435
  Buy: 2008/3/10 at 207.4504
  Earning: 164.5931 per share.
Improved Brute-force approach: CPU time 0.001091 s
  Sell: 2007/11/6 at 372.0435
  Buy: 2008/3/10 at 207.4504
  Earning: 164.5931 per share.
Divide and Conquer: CPU time 4.76854e-05 s
  Sell: 2007/11/6 at 372.0435
  Buy: 2008/3/10 at 207.4504
  Earning: 164.5931 per share.
Dynamic Programming: CPU time 3.78361e-06 s
  Sell: 2007/11/6 at 372.0435
  Buy: 2008/3/10 at 207.4504
  Earning: 164.5931 per share.
```

Good, solution is correct.
'MaxSubArrayDP' algorithm can be more efficient if it works on the stock price, instead of the price change.

---

score: 76.0

- Overall report writing
  - English writing needs more practice.
- Report format
  - Need double line spacing
- Introduction
  - Introduction can be strengthened
- Approach
  - 'MaxSubArrayDP' algorithm can be more efficient if it works on the stock price, instead of the price change.
- Program format can be improved

– Please follow the coding guidelines.

hw06.c

```
// EE3980 HW06 Stock Short Selling Revisited
// 106061225, 楊宇羲
// 2021/4/13

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

typedef struct STKprice {                   // Stock's price & date
    int year, month, day;
    double price, change;
}STKprice;
} STKprice;

typedef struct RSLT {                       // Contains start, end date and maximum
    int sell, buy;
    double min;
}RSLT;
} RSLT;

void ReadData(void);                        // read everyday's price
void PrintData(RSLT result);
double GetTime(void);                       // Get local time
RSLT MaxSubArrayBF(void);                   // brute-force sub array solution
RSLT MaxSubArrayBFI(void);                  // improved brute-force max sub array
RSLT MaxSubArray(int begin, int end);       // divide & conquer approach
RSLT MaxSubArrayXB(int begin, int mid, int end);    // cross boundary condition
RSLT MaxSubArrayDP(void);

STKprice *data;                             // array to store price
int N;                                      // size of array

int main(void)
{
    int i = 0;                              // for loop
    int R = 5000;                           // Repetition time
    double t;                               // time variable
    RSLT result;                            // contains start & end date and maximum

```

```c
    ReadData();                         // read in data

    t = GetTime();                      // start counting CPU time
    result = MaxSubArrayBF();           // brute-force solution for 1 time
    t = GetTime() - t;                  // end count time
    printf("Brute-force approach: CPU time %g s\n", t); // print outcome
    PrintData(result);

    t = GetTime();                      // start counting CPU time
    result = MaxSubArrayBFI();          // improved brute-force solution
    t = GetTime() - t;                  // end count time
    printf("Improved Brute-force approach: CPU time %g s\n", t); // print
    PrintData(result);

    t = GetTime();                      // start counting time
    for(i = 0; i < R; i++) {            // repeat 5000 times
    for (i = 0; i < R; i++) {            // repeat 5000 times
        result = MaxSubArray(0, N - 1); // divide and counquer approach
    }
    t = (GetTime() - t) / R;            // end counting time
    printf("Divide and Conquer: CPU time %g s\n", t);   // print outcome
    PrintData(result);

    t = GetTime();                      // start counting time
    for(i = 0; i < R; i++) {            // repeat 5000 times
    for (i = 0; i < R; i++) {            // repeat 5000 times
        result = MaxSubArrayDP();       // dynamic programming approach
    }
    t = (GetTime() - t) / R;            // end counting time
    printf("Dynamic Programming: CPU time %g s\n", t);   // print outcome
    PrintData(result);
    return 0;
}
void ReadData(void)                     // read in data
{
    int i = 0;                          // for loop

    scanf("%d", &N);                    // read in size of array
    printf("N = %d\n", N);

    data = (STKprice*)malloc(N * sizeof(STKprice)); // dynamic array allocation
```

```

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

        // derive the change of today and yesterday except the first day
        if (i != 0) data[i].change = data[i].price - data[i - 1].price;
        else data[i].change = 0;
    }
}
```

Need a blank line here.

```
void PrintData(RSLT result)                 // print outcome
{
    printf("  Sell: %d/%d/%d at %.4lf\n", data[result.sell - 1].year,
                                          data[result.sell - 1].month,
                                          data[result.sell - 1].day,
                                          data[result.sell - 1].price);
    printf("  Buy: %d/%d/%d at %.4lf\n", data[result.buy].year,
                                         data[result.buy].month,
                                         data[result.buy].day,
                                         data[result.buy].price);
    printf("  Earning: %.4lf per share.\n", -1 * result.min);
}
```

Need a blank line here.

```
double GetTime(void)                        // get local time
{
    struct timeval tv;

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

Need a blank line here.

```
RSLT MaxSubArrayBF(void)                    // brute-force solution
{
    int i, j, k;                            // for loops
    RSLT result;                            // for storing low, high, change
    result.min = 0.0;                       // initializations of result
    result.sell = 0;
    result.buy = N - 1;

```

```
    for (i = 0; i < N; i++) {                   // determine start index
        for (j = i; j < N; j++) {               // determine end index
            double sum = 0;                     // start counting sum
```

Do not mix declarations with statements

```
            for (k = i; k <= j; k++) {          // from start to end
                sum += data[k].change;          // derive sum
            }
            if (sum < result.min) {
                result.min = sum;               // found max, update informations
                result.sell = i;
                result.buy = j;
            }
        }
    }
    return result;
}
```

Need a blank line here.

```
RSLT MaxSubArrayBFI(void)                       // improved brute-force solution
{
    int i, j;                                   // for loops
    RSLT result;                                // for storing low, high, change
    result.min = 0.0;                           // initializations of result
    result.sell = 0;
    result.buy = N - 1;

    for (i = 0; i < N; i++) {                   // determine start index
        for (j = i; j < N; j++) {               // determine end index
            if (data[j].price - data[i].price < result.min) {   // find smaller
                result.min = data[j].price - data[i].price;
                result.sell = i + 1;            // store new sell date
                result.buy = j;                 // store new buy date
            }
        }
    }

    return result;                              // return result
}
```

Need a blank line here.

```
RSLT MaxSubArray(int begin, int end)            // divide and conquer approach
{
    RSLT result;                                // store information
```

```
    int mid;                                // initialization
    if (begin == end) {                     // terminal condition
        result.sell = begin;
        result.buy = end;
        result.min = 0;
        return result;
    }
    mid = (begin + end) / 2;                // determine middle index

    RSLT lsum = MaxSubArray(begin, mid);    // check left sum
```

Do not mix declarations with statements

```
    RSLT rsum = MaxSubArray(mid + 1, end);  // check right sum
    RSLT xsum = MaxSubArrayXB(begin, mid, end); // check cross boundary sum

    // comparisons of the above three sums
    if (lsum.min <= rsum.min && lsum.min <= xsum.min) {
        return lsum;                        // left sum is the smallest
    }
    else if (rsum.min <= lsum.min && rsum.min <= xsum.min) {
        return rsum;                        // right sum is the smallest
    }
    return xsum;                            // cross sum is the smallest
}
```

Need a blank line here.

```
RSLT MaxSubArrayXB(int begin, int mid, int end) // cross boundary condition
{
    RSLT result;                            // store information
    int i;                                  // for loop
    double sum;                             // for checking sum
    double rsum;                            // right hand side's sum
    double lsum;                            // left hand side's sum
    result.sell = 0;                        // initialize sell's date
    result.buy = mid + 1;                   // initialize buy's date

    sum = 0.0;
    lsum = 0.0;
    for (i = mid; i > begin; i--) {
        sum += data[i].change;              // counting sum
        if (sum < lsum) {                   // determine left min
            lsum = sum;                     // update left sum
            result.sell = i;                // update sell's date
```

```
        }
    }

    sum = 0.0;
    rsum = 0.0;                             // right hand side's sum
    for (i = mid + 1; i < end; i++)
    {
        sum += data[i].change;              // counting sum
        if (sum < rsum) {                   // determine right sum
            rsum = sum;                     // update right sum
            result.buy = i;                 // update buy date
        }
    }

    result.min = rsum + lsum;               // update total sum
    return result;
}
Need a blank line here.
RSLT MaxSubArrayDP(void)                    // dynamic programming
{
    RSLT result;                            // initializations
    double localmin = 0.0;
    int i;                                  // for loop & buy date
    int sell;                               // for sell date

    sell = 1;                               // initial values
    result.sell = 1;
    result.buy = N - 1;
    result.min = 10000000;

    for (i = 0; i < N; i++) {
        /* find max(data[i], data[i] + local min) */
        if (data[i].change < localmin + data[i].change) {
            localmin = data[i].change;
            sell = i;                       // sell date recorded
        }else {
        } else {
            localmin = data[i].change + localmin;
        }
        /* find max(local min, absolute min) */
        if (localmin < result.min) {
```

```
            result.min = localmin;          // minimum updated
            result.buy = i;                 // buy date updated
            result.sell = sell;             // sell date updated
        }
    }
    return  result;
}

```