EE3980 Algorithms

Homework 4. Trading Stock

By 105061212 王家駿

2019/03/30

1. Introduction

In this homework, we find the maximum earning at for one-buy-one-sell stock trading. We have N data, with each datum including the date and the price of the stock at each date, so we can convert the prices into an array with price changes, which indicate the price differences between a day and its previous day. Hence, the problem becomes finding a range of the array with maximum contiguous sum. Because the sum is as same as the price difference of the day between where the array begins and where the array ends, the maximum sum would lead to the maximum earning. In order to find the maximum contiguous sum, we use two methods, the brute-force approach and the divide-and-conquer recursion, and compare the time consumed between the two methods when executing.

2. Implementation

In the program, we first read all inputs and store them in an array with type **STKprice**, which contains the information of the date, the price and the price change. Then we use the two methods, **MaxSubArrayBF** and **MaxSubArray**, which would

be explained later, to implement the network connectivity, and also record the CPU time consumed. Finally, we show the results including the date we should buy and sell, the stock price at the dates, the total earning per share, and the CPU time on the screen.

2.1. Brute-force approach

```
1.  Algorithm MaxSubArrayBF(A, n, low, high)
2.  {
3.      max := 0; low := 1; high := n;        // Initialize
4.      for j := 1 to n do {                  // Try all possible ranges: A[j:k]
5.          for k := j to n do {
6.              sum := 0;
7.              for i := j to k do {          // Summation for A[j:k]
8.                  sum := sum + A[i];
9.              }
10.             if (sum > max) then {         // Record the max value and range
11.                 max := sum; low := j; high := k;
12.             }
13.         }
14.     }
15.     return max;
16. }
```

In the function **MaxSubArrayBF**, we implement the brute-force approach to find the maximum contiguous sum. The brute-force solution is to check every subarray, i.e. to check every possible date for buying and selling, and then compare their value to fine the largest one. So we first let j run through the array to get the date of buying, and let k run through the array to get the date of selling, and sum up the values to get

the price differences. Then we record the value if it is the largest one until now. After the iterations go to end, the value left must be the largest value of summation of contiguous subarray. That is, the maximum earning of the stock.

For the time complexity, the first(outer) loop would execute N times, the second loop N-j+1 times, and the third k-j+1 times. Since j and k are related to the problem size N, the overall time complexity is $O(n^3)$, for average case, worst case, and best case.

For the space complexity, we need extra parameters: i, j, k, max, low, high, and the initial array with size N, so the space complexity is $N + 6$ which is $O(n)$.

Time complexity: $O(n^3)$

Space complexity: $O(n)$

2.2. Divide-and-conquer approach

```
1.  Algorithm MaxSubArray(A, begin, end, low, high)
2.  {
3.      if (begin = end) then {                          // termination condition
4.          low := begin; high := end; return A[begin];
5.      }
6.
7.      mid := (begin+end) / 2;
8.      lsum := MaxSubArray(A, begin, mid, llow, lhigh); // left region
9.      rsum := MaxSubArray(A, mid+1, end, rlow, rhigh); // right region
10.     // cross boundary
11.     xsum := MaxSubArrayXB(A, begin, mid, end, xlow, xhigh);
12.
```

```
    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;
    }
    // cross-boundary is the largest
    low := xlow; high := xhigh; return xsum;
}
```

In the function **MaxSubArray**, we implement the divide-and-conquer approach to find the maximum contiguous sum. When using this method, we divide the whole problem to small pieces, and calculate the maximum contiguous sum of each segment by recursion. In order to deal with the situation where the date to buy and the date to sell are not in the same segment, we use the function **MaxSubArrayXB**.

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

    rsum := 0; high := mid+ 1; sum := 0;    // Initialize for higher half
    for i := mid + 1 to end do {            // find high to maximize
        sum := sum + A[i];                  // Continue to add.
        if (sum > rsum) then {              // record if larger
            rsum := sum; high := i;
        }
    }

```

```
19.     return lsum + rsum;                    // Overall sum.
20. }
```

In the function **MaxSubArrayXB**, we calculate the maximum contiguous sum where the begin is at the left segment and the end is at the right segment. This is same as finding the maximum contiguous sum of the left segment plus the maximum contiguous sum of the right segment. Since the subarray must be contiguous, we must find from mid to begin at the left segment, and find from mid+1 to end at the right segment to ensure the subarray contiguous. Thus, we find the subarray both for the two parts, and record the value when it is the largest one until now. At the end of the iterations, we can get the maximum contiguous sum for both two parts and sum up them, so then we could get the cross-boundary maximum contiguous sum.

Now we go back to the **MaxSubArray** function. At first, we define the terminal condition to stop the recursion when begin is equal to end, i.e. the segment only has one element. Then, at each recursion step, we divide the array in two pieces from the middle, and call the recursive function at both left and right parts to find the sum of each segment. We also find the cross-boundary sum. Finally, we compare the results of the maximum contiguous sum of the three part: left segment, right segment and cross-boundary, and choose the largest one to be the maximum contiguous sum of the whole array.

For the time complexity, when the problem size is n for the **MaxSubArray**, and the time consumed is T(n), and we let Txb(n) be the time consumed of **MaxSubArrayXB** with problem size n.

For the **MaxSubArrayXB** function, the iterations go through the left part and the right part, which is total n steps. Thus, we could find out that $\mathrm{Txb}(n) = n$.

For the **MaxSubArray** function, their would be three comparison, two **MaxSubArray** with problem size n/2, and a with problem **MaxSubArrayXB** size n. Since the comparisons are constant time, we could just ignore them. Assume problem size n is 2^k. So, the time consumed with n is:

$$\begin{aligned}
\mathrm{T}(n) &= \mathrm{T}\left(\frac{n}{2}\right) * 2 + \mathrm{Txb}(n) = \mathrm{T}\left(\frac{n}{2}\right) * 2 + n \\
&= \left(\mathrm{T}\left(\frac{n}{4}\right) * 2 + \frac{n}{2}\right) * 2n \\
&= \mathrm{T}\left(\frac{n}{4}\right) * 4 + 2n \\
&= \cdots \\
&= \mathrm{T}\left(\frac{n}{2^k}\right) * 2^k + k * n \\
&= n + n * \log(n)
\end{aligned}$$

Thus, the time complexity of the divide-and-conquer method is $O(n * \log n)$.

For the space complexity, we need extra parameter like mid, sum, lsum, rsum… And since the recursion would execute at most log(n) times, we must have extra spaces for these parameters at each recursion step. Adding the initial array with space n, the overall space complexity is $O(\log n + n) = O(\log n)$.

This is incorrect!

Time complexity: $O(n * \log n)$

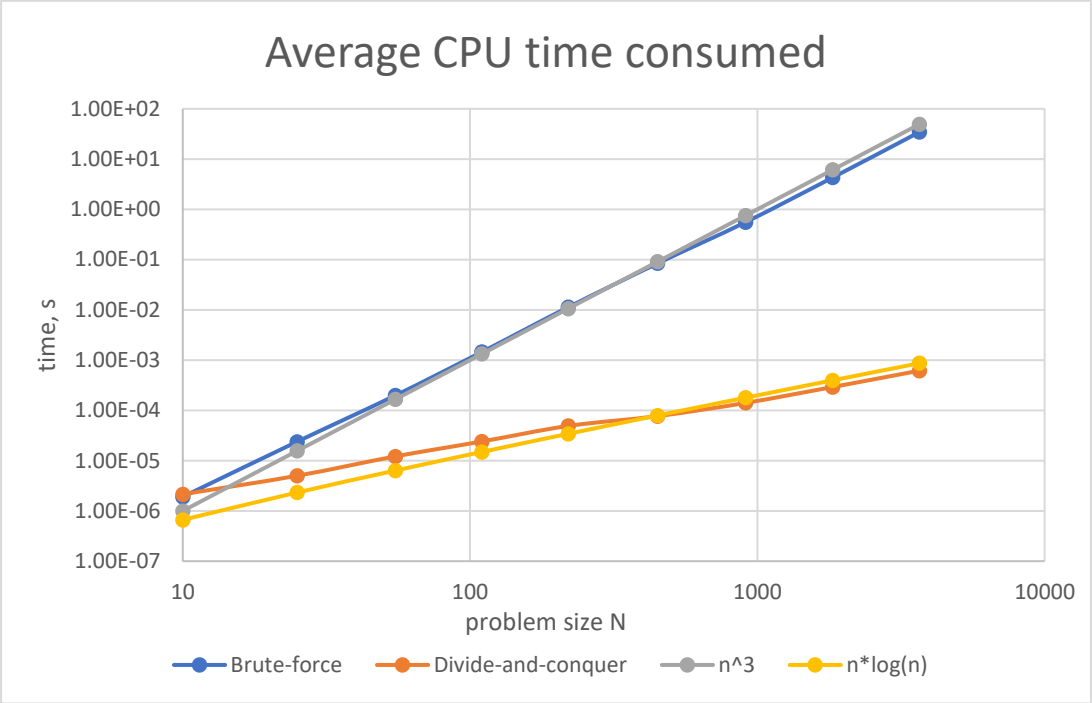
Space complexity: $O(\log n)$

3. Executing results

We run the testing data from s1.dat to s9.dat with different input data size by brute-force approach one time and divide-and-conquer 1000 times, and record the average CPU time used.

Data size	Brute-force approach	Divide-and-conquer	Earning per share
10	1.907 μs	2.146 μs	9.065
25	23.84 μs	5.007 μs	20.81
55	198.8 μs	12.16 μs	96.02
110	1.456ms	24.08 μs	103.9
220	11.35ms	49.11 μs	204.1
450	84.34ms	77.01 μs	371.6
910	555.1ms	140.9 μs	641.8
1830	4.295s	292.1 μs	641.8

3671	34.56s	617.0 μ s	1185



4. Result analysis and conclusion

From the graph, we could observe that the brute-force approach has a trend of n^3, and the divide-and-conquer approach has a trend of $\mathrm{n} * \log n$, which are same as our estimation.

Note that for the problem size 10, the brute-force approach is faster than the divide-and-conquer. It might be the effect that the divide-and-conquer approach use recursive functions, which might consume a lot of time while the function calls. Thus, the iteration methods might be faster than the recursion method for the small input data size. However, when the problem size grows up, the time consumed of the brute-

force grows too fast so that the function calls take less time with compared to the brute-force approach.

Compare the space complexities of the two methods. The brute-force approach takes $O(n)$ and $O(\log n)$ for the divide-and-conquer method, which means that the later would need more memories for operating than the former. Thus, it is a trade-off between time and space efficiency at the two methods. Since the nowadays techniques have great improvement at the computer memories, we would often emphasize the time efficiency than the space, so we usually take the divide-and-conquer approach to reduce the time used.

To use the algorithm in real stock trading, we must to know the stock price on every day, and we could calculate the time when we should buy and sell efficiently by the divide-and-conquer approach. However, due to the fact that all the prices have to be known at first, we couldn't predict the stock price in the future, what we can only do is to calculate the time for maximum earning for the past. Perhaps there's someone or someway that could correctly predict the stock market by the past data. Then we could use the algorithm to analyze the past data efficiently to make money.

Score: 87

o. See return.

[Analysis] space complexity of the divide-and-conquer approach is not correct!

[Coding] Can use space characters more effectively to make your codes more legible.

hw04.c

```
1  /* EE3980 HW04 Trading Stock
2   * 105061212, Chia-Chun Wang
3   * 2019/03/30
4   */
5
6  #include <stdio.h>
7  #include <stdlib.h>
8  #include <sys/time.h>
9
10 typedef struct sSTKprice                        // stock data structure
11 {
12     int year, month, day;
13     double price, change;
       Can use comments to explain the purpose of each item.
14 } STKprice;
15
16 typedef struct sResult                          // max contiguous sum structure
17 {
18     int buy;
19     int sell;
20     double earning;
21 } Result;
22
23 void readInput(void);                           // read data input
24 double GetTime(void);                           // get current CPU time
25 Result MaxSubArrayBF(void);                     // brute-force approach
26 Result MaxSubArray(int begin, int end);         // divide-and-conquer
27 // find maxsubArray with cross boundary
28 Result MaxSubArrayXB(int begin, int mid, int end);
29 // print out results
30 void printResult(double t_BF, double t_DandC, Result r_BF, Result r_DandC);
31
32 int N;                                          // number of data input
33 int Nrepeat = 1000;                             // number of repetitions
34 STKprice* data;                                 // Array to store input data
35
36 int main()
   int main(void)
37 {
38     int i;                                      // loop index
39     double t0, t1, t2;                          // CPU time
40     double t_BF, t_DandC;                       // average CPU time
41     Result r_BF, r_DandC;                       // max contigous sum results
                                                          contigous
42
43     readInput();                                // read data input
44
45     t0 = GetTime();                             // get current CPU time
```

```
46
47      r_BF = MaxSubArrayBF();                     // find result by brute-force
48
49      t1 = GetTime();                             // get current CPU time
50
51      for (i = 1; i <= Nrepeat; i++){             // repeat Nrepeat times
        for (i = 1; i <= Nrepeat; i++) {
52          // find result by devide-and-conquer
53          r_DandC = MaxSubArray(1, N);
54      }
55
56      t2 = GetTime();                             // get current CPU time
57
58      // calculate average CPU time
59      t_BF = t1 - t0;
60      t_DandC = (t2 - t1) / Nrepeat;
61
62      printResult(t_BF, t_DandC, r_BF, r_DandC);  // print out results
63
64      free(data);                                 // free dynamic memories
65
66      return 0;
67  }
68
69  void readInput(void)                            // read data input
70  {
71      int i;                                      // loop index
72
73      scanf("%d", &N);                            // number of data
74
75      // allocate dynamic memories for data input
76      data = (STKprice*)malloc(sizeof(STKprice) * (N+1));
77
78      // read the first data
79      scanf("%d", &data[1].year);
80      scanf("%d", &data[1].month);
81      scanf("%d", &data[1].day);
82      scanf("%lf", &data[1].price);
83      data[1].change = 0;                         // change of the first data = 0
84
85      for (i = 2; i <= N; i++){                   // read the rest data
        for (i = 2; i <= N; i++) {
86          scanf("%d", &data[i].year);
87          scanf("%d", &data[i].month);
88          scanf("%d", &data[i].day);
89          scanf("%lf", &data[i].price);
90          // calcute the price changes
91          data[i].change = data[i].price - data[i-1].price;
92      }
93  }
```

```
 94
 95 double GetTime(void)                              // get local time in seconds
 96 {
 97     struct timeval tv;                           // time interval structure
 98
 99     gettimeofday(&tv, NULL);                     // write local time into tv
100
101     return tv.tv_sec + tv.tv_usec * 0.000001;    // return time with microsecond
102 }
103
104 Result MaxSubArrayBF(void)                       // brute-force approach
105 {
106     int i, j, k;                                 // loop index
107     double sum;                                  // temporary sum
108     Result r;                                    // result
109
110     r.earning = 0;                               // initialize r
111
112     for (j = 1; j <= N; j++){                    // try begin from 1 to N
        for (j = 1; j <= N; j++) {
113         for (k = j; k <= N; k++){                // try end from begin to N
            for (k = j; k <= N; k++) {
114             sum = 0;
115             for (i = j; i <= k; i++){            // summation from begin to end
116                 sum += data[i].change;
117             }
118             if(sum > r.earning){                 // record max value and range
                if (sum > r.earning) {
119                 r.earning = sum;
120                 r.buy = j;
121                 r.sell = k;
122             }
123         }
124     }
125
126     return r;
127 }
128
129 Result MaxSubArray(int begin, int end)           // devide-and-conquer
130 {
131     int mid;                                     // middle point
132     Result r, lsum, rsum, xsum;                  // result
133
134     if(begin == end){                            // terminal condition
135         r.buy = begin;
136         r.sell = end;
137         r.earning = data[begin].change;
138         return r;
139     }
140
```

```
141     mid = (begin + end) / 2;
142     lsum = MaxSubArray(begin, mid);             // left region
143     rsum = MaxSubArray(mid+1, end);             // right region
144     xsum = MaxSubArrayXB(begin, mid, end);      // cross boundary region
145
146     if(lsum.earning >= rsum.earning && lsum.earning >= xsum.earning){
147         return lsum;                            // lsum is the largest
148     }
149     if(rsum.earning >= lsum.earning && rsum.earning >= xsum.earning){
150         return rsum;                            // rsum is the largest
151     }
152     return xsum;                                // xsum is the largest
153 }
154
155 // find max subarray with cross boundary
156 Result MaxSubArrayXB(int begin, int mid, int end)
157 {
158     int i;                                      // loop index
159     double sum, lsum, rsum;                     // summation
160     Result r;                                   // result
161
162     lsum = 0; sum = 0; r.buy = mid;             // initialize
163     for (i = mid; i >= begin; i--){             // go through the left region
164         sum += data[i].change;                  // continue to add
165         if(sum >= lsum){                        // record if larger
166             lsum = sum;
167             r.buy = i;
168         }
169     }
170
171     rsum = 0; sum = 0; r.sell = mid + 1;        // initialize
172     for (i = mid+1; i <= end; i++){             // go the the right region
173         sum += data[i].change;                  // continue to add
174         if(sum >= rsum){                        // record if larger
175             rsum = sum;
176             r.sell = i;
177         }
178     }
179
180     r.earning = lsum + rsum;                    // overall sum
181     return r;
182 }
183
184 // print out the results
185 void printResult(double t_BF, double t_DandC, Result r_BF, Result r_DandC)
186 {
187     // the buy/sell date data
188     STKprice BF_buy, BF_sell, DandC_buy, DandC_sell;
189
190     // find data by the results we got
```

```
191     BF_buy = data[r_BF.buy];
192     BF_sell = data[r_BF.sell];
193     DandC_buy = data[r_DandC.buy];
194     DandC_sell = data[r_DandC.sell];
195
196     // print out all the results
197     printf("N = %d\n", N);
198     printf("Brute-force approach: time %e s\n", t_BF);
199     printf("Buy: %d/%d/%d at %lf\n", BF_buy.year, BF_buy.month,
200             BF_buy.day, BF_buy.price);
201     printf("Sell: %d/%d/%d at %lf\n", BF_sell.year, BF_sell.month,
202             BF_sell.day, BF_sell.price);
203     printf("Earning: %lf per share.\n", r_BF.earning);
204     printf("Divide and Conquer: time %e s\n", t_DandC);
205     printf("Buy: %d/%d/%d at %lf\n", DandC_buy.year, DandC_buy.month,
206             DandC_buy.day, DandC_buy.price);
207     printf("Sell: %d/%d/%d at %lf\n", DandC_sell.year, DandC_sell.month,
208             DandC_sell.day, DandC_sell.price);
209     printf("Earning: %lf per share.\n", r_DandC.earning);
210 }
```