EE3980 Algorithms

Homework 5. Trading Stock II

By 105061212 王家駿

2019/04/05

1. Introduction

In this homework, we improve our previous homework, trading stocks with finding the maximum contiguous subarray by both brute-force and divide-and-conquer method. As for the brute-force approach, we reduce the time complexity from $O(n^3)$ to $O(n^2)$, which might deviate from using the maximum subarray approach; and for the divide-and-conquer with time complexity of $O(n * \log n)$, we adopt a method called Kadane's algorithm to make it reach just $O(n)$. In the end, we compare the time consumed of both methods with what we got in the last homework, and analyze the cause of the differences.

2. Implementation

In the program, we use the main structure of the previous homework, and only revise two parts of the code. First, we revise the **MaxSubArrayBF** function to make it more efficient. Second, we replace the divide-and conquer method by Kadane's algorithm, which would be explained later, in the **MaxSubArray** function.

2.1. Brute-force approach

In our previous homework, we implement the brute-force approach with the time complexity of $O(n^3)$, since it has to runs through the array with the start and end points of the subarray, which contributes to $O(n^2)$, and then sum up all the items in the subarray, which contributes to $O(n)$.

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

We have to implement this way since we treat it as the maximum contiguous subarray sum problem. However, we had known the actual price of stocks at each moment, and the contiguous subarray sum stands for the price difference between the start point and the end point. So, we could get the sum just by calculating the difference of the prices between the buy date and sell date, instead of summing up the price differences through the array. Thus, we replace line 6 ~ 9 by only one assign statement.

```
1.  Algorithm MaxSubArrayBF_revised(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 = A[k] - A[j];          // price diffence between k and j
7.              if (sum > max) then {       // Record the maximum value and range
8.                  max := sum; low := j; high := k;
9.              }
10.         }
11.     }
12.     return max;
13. }
```

For the time complexity, the outer and inner loop still contribute $O(n^2)$, where the iterations may go through the array. And at line 6 ~ 9, there are at most only one comparison and four assignment, which contribute a time complexity of constant time. Thus, the overall time complexity of the brute-force approach becomes $O(n^2)$.

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

Time complexity: $O(n^2)$

Space complexity: $O(n)$

2.2. Kadane's algorithm

In our previous homework, we implemented the divide-and-conquer method to solve the maximum contiguous sum problem, with the time complexity of $O(n * \log n)$. Yet, we try to use the Kadane's algorithm, contributed by an American professor of computer science Joseph B. Kadane, in the **MaxSubArray** function to reduce the time complexity of the problem.

```
Algorithm Kadane(A, n, start, end)
{
    start := 2; end := 1;           // initialize two ends
    start_tmp := 2;                 // temporary start index
    max := 0; now := 0;             // initialize values
    for i := 2 to n do {            // go through the array
        now := now + A[i];          // subarray value until i
        if (now < 0) then {         // if value < 0, reset
            now := 0;
            start_tmp := i + 1;
        }
        if (now > max) then {       // record if value is largest
            max := now;
            start := start_tmp;
            end := i;
        }
    }
    return max;
}
```

The feature of Kadane's algorithm is that the iteration only goes through the array one time, and it doesn't need any recursion. In the algorithm, we use two variables *max* and *now* to record the value of the maximum contiguous sum so far and the contiguous sum with A[i], respectively.

During each iteration step, we first add the current datum A[i] to the variable *now*, which then stands for the contiguous subarray sum including A[i]. And if *now* is less than zero, it means that the contiguous sum is negative. That is, the price is lower at this point than where the sum take start. Thus, we reset the sum by assign *now* to zero, and take the current index to be the new beginning of the contiguous sum.

Then, we check whether the contiguous sum *now* is larger than the contiguous sum so far. If yes, we refresh the value of *max* to record the maximum sum since we want to get the maximum value until now. At the end of each iteration step, we can get the maximum contiguous sum from 1 to the iteration index i, which is either the maximum contiguous sum we recorded before or the contiguous sum including A[i]. Therefore, when the iteration goes to end, we could find the maximum contiguous sum from 1 to n.

For the time complexity, the loop at line 6 goes through the whole array with size n, which contribute the time complexity of $O(n)$. In the loop, there are only comparisons, addition, and assignment operations, which would take constant time.

Thus, the overall time complexity is $O(n)$, and that means we can solve the maximum contiguous sum problem by using Kadane's algorithm with linear time.

As for the space complexity, we need extra parameters: i, start, start_tmp, end, max, now and the initial array with size N. So, the space complexity is N + 6 which is $O(n)$.

Time complexity: $O(n)$

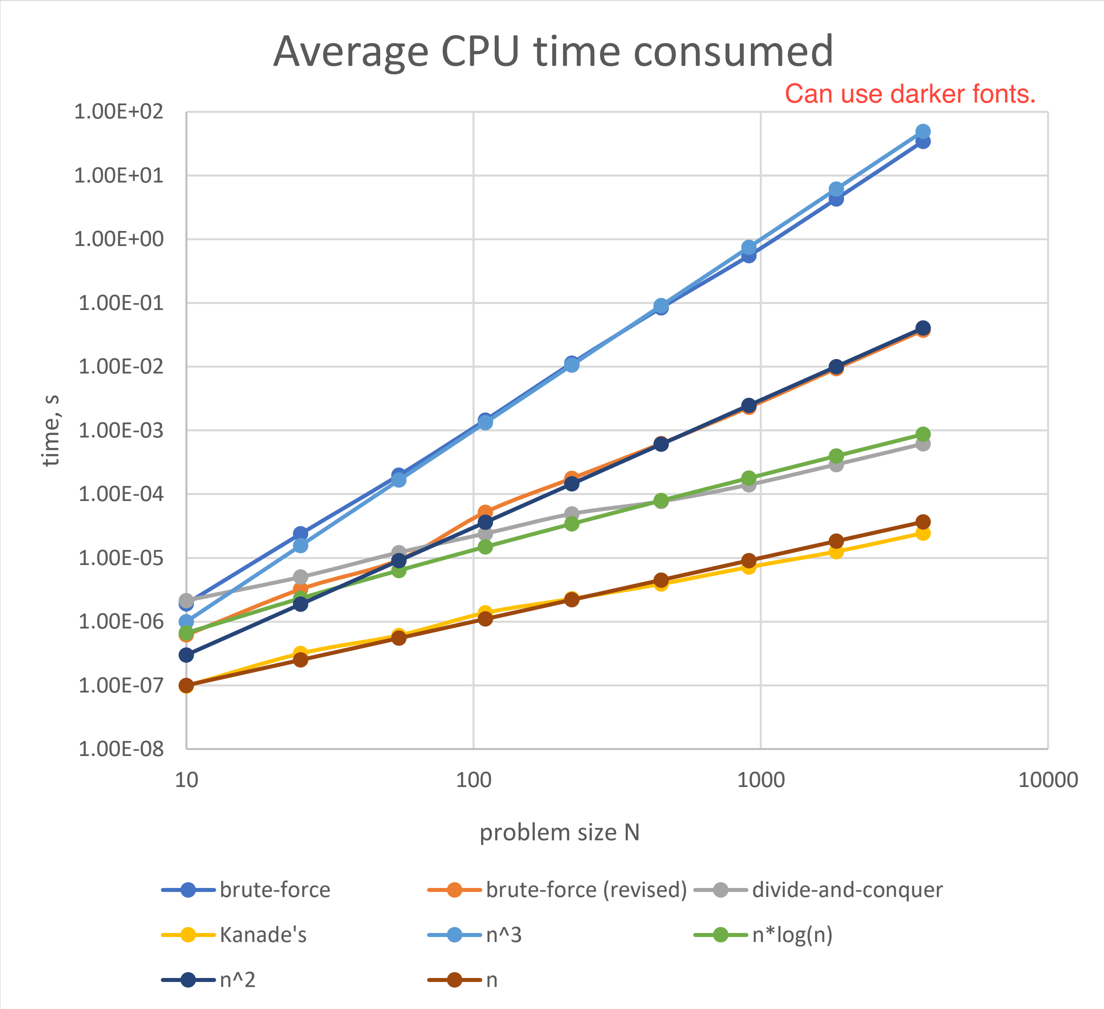
Space complexity: $O(n)$

3. Executing results

We run the testing data from s1.dat to s9.dat with different input data size by revised brute-force approach and Kadane's algorithm for 500 times, record the average CPU time used, and compare to the results we got in the previous homework.

Data size	Brute-force approach	Brute-force approach (revised)	Divide-and-conquer	Kadane's algorithm	Earning per share
10	1.907 μs	616.1ns	2.146 μs	97.75ns	9.065
25	23.84 μs	3.222 μs	5.007 μs	315.7ns	20.81
55	198.8 μs	9.306 μs	12.16 μs	608.0ns	96.02
110	1.456ms	52.25 μs	24.08 μs	1.372 μs	103.9
220	11.35ms	177.0 μs	49.11 μs	2.280 μs	204.1
450	84.34ms	622.69 μs	77.01 μs	3.902 μs	371.6

910	555.1ms	2.315ms	140.9 μ s	7.176 μ s	641.8
1830	4.295s	9.386ms	292.1 μ s	12.58 μ s	641.8
3671	34.56s	37.41ms	617.0 μ s	24.46 μ s	1185



4. Result analysis and conclusion

From the graph, we could observe that the advised brute-force approach has a trend of n^2, and the Kadane's algorithm has a trend of n, which are same as our estimation.

Compared to the previous homework, the time complexity of the brute-force approach successfully reduces from $O(n^3)$ to $O(n^2)$, and we had found an algorithm whose time complexity is lower than $O(n * \log n)$. There are large scales of improvements on both the methods.

The Kadane's algorithm is the fastest on all input data. It might be the fact that it runs through the array for only one time, and it doesn't need any recursive function calls. Thus, the algorithm could be used on a wide range of input data size, making it faster than all other methods we mentioned above. Furthermore, if we just want to know the contiguous sum instead of the indexes, we could take off the start, start_tmp, end variables in the algorithm, which could be much faster.

The least time complexity of solving maximum contiguous sum we got so far is $O(n)$, and it might not be less. Since we must know the content of each index in the array with size n to solve the problem, we have to run through the array to get the values, which contributes the time complexity with $O(n)$. Thus, $O(n)$ must be the least time complexity of the maximum contiguous sum problem.

Score: 97

o. See return.

[Writing] minor corrections.

hw05.c

```c
/* EE3980 HW05 Trading Stock II
 * 105061212, Chia-Chun Wang
 * 2019/04/05
 */

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

typedef struct sSTKprice                        // stock data structure
{
    int year, month, day;
    double price, change;
    can add comments to explain the purpose of each item.
} STKprice;

typedef struct sResult                          // max contiguous sum structure
{
    int buy;                                    // date to buy
    int sell;                                   // date to sell
    double earning;                             // price difference
} Result;

void readInput(void);                           // read data input
double GetTime(void);                           // get current CPU time
Result MaxSubArrayBF(void);                     // brute-force approach
Result MaxSubArray(void);                       // Kadane's method
// print out results
void printResult(double t_BF, double t_DandC, Result r_BF, Result r_DandC);

int N;                                          // number of data input
int Nrepeat = 500;                              // number of repetitions
STKprice* data;                                 // Array to store input data

int main(void)
{
    int i;                                      // loop index
    double t0, t1, t2;                          // CPU time
    double t_BF, t_Kadane;                      // average CPU time
    Result r_BF, r_Kadane;                      // max contiguous sum results

    readInput();                                // read data input

    t0 = GetTime();                             // get current CPU time
    for (i = 1; i <= Nrepeat; i++) {            // repeat Nrepeat times
        r_BF = MaxSubArrayBF();                 // find result by brute-force
    }

```

```
    t1 = GetTime();                             // get current CPU time

    for (i = 1; i <= Nrepeat; i++) {            // repeat Nrepeat times
        r_Kadane = MaxSubArray();               // find result by Kadane
    }

    t2 = GetTime();                             // get current CPU time

    // calculate average CPU time
    t_BF = (t1 - t0) / Nrepeat;
    t_Kadane = (t2 - t1) / Nrepeat;

    printResult(t_BF, t_Kadane, r_BF, r_Kadane);// print out results

    free(data);                                 // free dynamic memories

    return 0;
}

void readInput(void)                            // read data input
{
    int i;                                      // loop index

    scanf("%d", &N);                            // number of data

    // allocate dynamic memories for data input
    data = (STKprice*)malloc(sizeof(STKprice) * (N+1));

    // read the first data
    scanf("%d", &data[1].year);
    scanf("%d", &data[1].month);
    scanf("%d", &data[1].day);
    scanf("%lf", &data[1].price);
    data[1].change = 0;                         // change of the first data = 0

    for (i = 2; i <= N; i++) {                  // read the rest data
        scanf("%d", &data[i].year);
        scanf("%d", &data[i].month);
        scanf("%d", &data[i].day);
        scanf("%lf", &data[i].price);
        // calcute the price changes
        data[i].change = data[i].price - data[i-1].price;
    }
}

double GetTime(void)                            // get local time in seconds
{
    struct timeval tv;                          // time interval structure

    gettimeofday(&tv, NULL);                    // write local time into tv
```

```

    return tv.tv_sec + tv.tv_usec * 0.000001;   // return time with microsecond
}

Result MaxSubArrayBF(void)                      // brute-force approach
{
    int j, k;                                   // loop index
    double sum;                                 // temporary sum
    Result r;                                   // result

    r.earning = 0;                              // initialize r

    for (j = 1; j <= N; j++) {                  // try begin from 1 to N
        for (k = j; k <= N; k++) {              // try end from begin to N
            sum = data[k].price - data[j].price;// sum is the price difference
            if (sum > r.earning) {              // record max value and range
                r.earning = sum;
                r.buy = j;
                r.sell = k;
            }
        }
    }

    return r;
}

Result MaxSubArray(void)                        // Kadane's method
{
    int i;                                      // loop index
    int start = 2, end = 1;                     // two ends of max subarray
    int start_tmp = 2;                          // temporary start point
    double max = 0;                             // max value of subarray so far
    double now = 0;                             // max value of subarray now
    Result r;                                   // result returned

    for (i = 2; i <= N; i++) {                  // go through the array
        now = now + data[i].change;             // subarray value until i
        if (now < 0) {                          // if value < 0, reset
            now = 0;
            start_tmp = i + 1;
        }
        if (now > max) {                        // record if value is largest
            max = now;
            start = start_tmp;
            end = i;
        }
    }

    // return the result
    r.buy = start - 1;
```

```c
    r.sell = end;
    r.earning = max;

    return r;
}

// print out the results
void printResult(double t_BF, double t_Kadane, Result r_BF, Result r_Kadane)
{
    // the buy/sell date data
    STKprice BF_buy, BF_sell, Kadane_buy, Kadane_sell;

    // find data by the results we got
    BF_buy = data[r_BF.buy];
    BF_sell = data[r_BF.sell];
    Kadane_buy = data[r_Kadane.buy];
    Kadane_sell = data[r_Kadane.sell];

    // print out all the results
    printf("N = %d\n", N);
    printf("Brute-force approach: time %e s\n", t_BF);
    printf("Buy: %d/%d/%d at %lf\n", BF_buy.year, BF_buy.month,
            BF_buy.day, BF_buy.price);
    printf("Sell: %d/%d/%d at %lf\n", BF_sell.year, BF_sell.month,
            BF_sell.day, BF_sell.price);
    printf("Earning: %lf per share.\n", r_BF.earning);
    printf("Divide and Conquer: time %e s\n", t_Kadane);
    printf("Buy: %d/%d/%d at %lf\n", Kadane_buy.year, Kadane_buy.month,
            Kadane_buy.day, Kadane_buy.price);
    printf("Sell: %d/%d/%d at %lf\n", Kadane_sell.year, Kadane_sell.month,
            Kadane_sell.day, Kadane_sell.price);
    printf("Earning: %lf per share.\n", r_Kadane.earning);
}
```