EE3980 Algorithms

HW3 Trading Stock

104061212 馮立俞

2017/3/24

Introduction

In this assignment, we are asked to read a sequence of dates and the stock prices of Google on that day. Then, to find out how to create greatest profit from buying the stock, we'll use both brute-force and divide-and-conquer approach to compute the maximum sum. Finally, we'll show how the two algorithms differ in terms of complexity.

Approach

Suppose a sequence of stock prices is known. We can compute how much price changes with regards to previous day. Then, Finding the best time to buy or sell stock is simply calculating the maximum contiguous sum in the price change sequence. It can be done through brute-force or divide-and-conquer approach.

Brute-force Approach

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

In brute-force approach, since there's $\frac{n(n-1)}{2}$ possibilities for (buy date, sell date) pair. We'll need to try out every pair of them. In the lecture slides, it is said that the third loop would cause another $O(n)$ time complexity, making the overall complexity $O(n^3)$. However, the third loop can be replaced by subtracting the price(high) with price(low) operation, through which an overall $O(n^2)$ complexity can be reached. In this assignment, I chose the $O(n^3)$ version to stay aligned with the slides.

By the way, the space complexity is $O(n)$ since there's no other array declared.

Divide and Conquer Approach

```
Algorithm MaxSubArray(A, begin, end, low, high) // Find low and  high to maximize
                                        ΣA   [i], begin ≦ low ≦ i ≦ high ≦ end.
    {
        if (begin = end) { // termination condition.
            low: = begin;
            high: = end;
            return A[begin];

        }
       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);
// cross boundary

        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

    }
```

```
Algorithm MaxSubArrayXB(A, begin, mid, end, low, high) 2 // Find low  and high to
                              maximize ΣA[i], begin ≦ low ≦ mid ≦ high ≦ end.
    {
        lsum: = 0;
        low: = mid;
        sum: = 0;
        for i: = mid to begin step− 1 do { // find low to maximize
                                            //ΣA[low : mid ]
                sum: = sum + A[i];
                if (sum > lsum) then {
                   lsum = sum;
                    low: = i;
                }
            }
        rsum: = 0;
        high: = mid + 1;
        sum: = 0;
        for i: = mid + 1 to end do { // find end to maximize
                      ΣA[mid + 1 : high ]
                sum: = sum + A[i];
                if (sum > rsum) then {
                    rsum = sum;
                    high: = i;
                }
            }
        return lsum + rsum;
    }
```

In the above `MaxSubArray` function we can see ordinary divide and conquer i.e. termination condition → split → merge structure.

When dealing with the cross boundary situation, it is kept in mind that if a maximum contiguous sum contains `mid`, then the sequence before and after `mid` are also maximum contiguous sums.

Since we divide the task into two parts ($O(\log(n))$), dealing with cross boundary situation for each part($O(n)$). It can be estimated that the time complexity is $O(n * \log(n))$, more robust proof is already shown in course slides.

Because recursion is used, and we need to at least store `begin, mid, end` variables for each part we split. There would be about $2n - 1$ parts, so the space complexity is $O(n)$.

Need to count the memory space needed due to recursion.

Results and analysis

Let's first tabulate the CPU time w.r.t. the algorithms.

Table 1. task size v.s CPU time (in seconds)

task size	9	30	72	155	325	658	1331	2672	3414
Brute-force	2.86E-06	3.81E-05	0.000509	0.0049	0.0336	0.23	1.82	12.9	27.1378
D & C	1.03E-06	4.07E-06	1.06E-05	2.29E-05	3.51E-05	9.08E-05	1.81E-04	3.85E-04	5.11E-04

We can obviously see divide and conquer approach outperform brute-force approach by great margin, especially as we encounter large task size. The difference is more evident if we plot above table, as Figure 1.

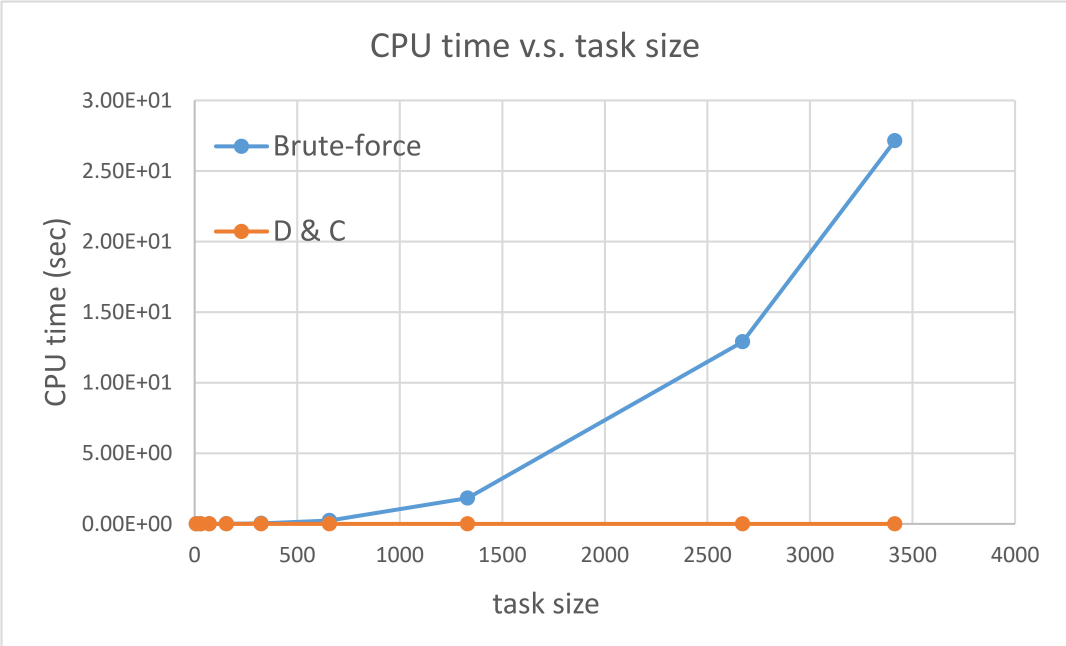

Figure 1. D & C v.s. BF CPU time

Then, to prove that the above time complexity analysis correct, we can plot

Figure 1. in log-log scale, as Figure 2.

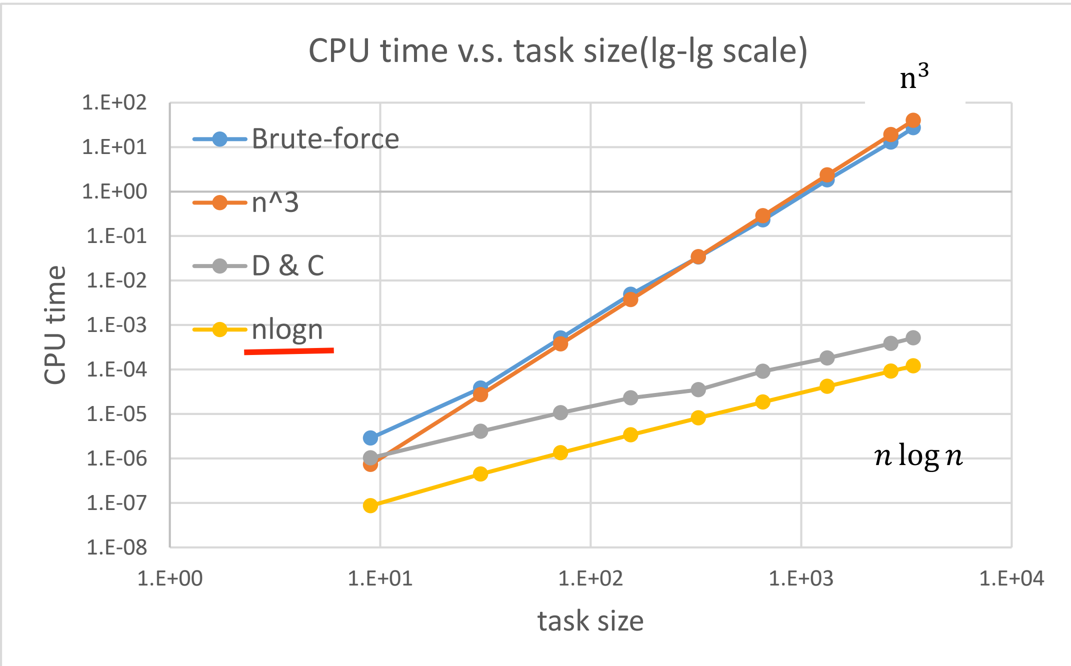

Figure 2. D & C v.s. BF CPU time (log-log scale)

The two curves are quite similar to our expectations.

Observations and Conclusion

We've seen huge performance difference above. However, even if we adopt the $O(n^2)$ version of BF approach, it's still no match with D & C. For example, below is the execution result of largest task in this assignment, with improved BF version

```
./a.out < s9.dat
N = 3414
Brute-force approach:
    CPU time 0.036952 s
…
Divide and Conquer approach:
    CPU time 0.000557042 s
…
```

It's still a great improvement for BF approach compared with initial execution time (27.1378 sec) nonetheless.

In conclusion, we've known that how algorithm complexity can largely determine execution time when faced with large-scale tasks. And to reduce complexity, divide and conquer is one good way to go.

```
/************************************
  EE3980 Algorithms HW03 Trading Stock
  Li-Yu Feng 104061212
  Date:2018/3/24
*************************************/

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

typedef struct sSTKprice {      //store date, price & price change
    int year,month,day;
    double price,change;
} STKprice;

typedef struct retval{  //store low, high index and the maximum sum between
    int low, high;      //for convenient return
    double sum;
} RETval;

double GetTime(void);
void MaxSubArrayBF(STKprice *A, int n);                 //Brute-force method
RETval MaxSubArray(STKprice *A, int begin, int end);    //D & C method
RETval MaxSubArrayXB(STKprice *A, int begin, int mid, int end);
                                                    //cross boundary
void PrintResult(STKprice *A, RETval result, double time);  //as its name

double GetTime(void)
{
    struct timeval tv;
    gettimeofday(&tv,NULL);
    return tv.tv_sec+1e-6*tv.tv_usec;
}

void MaxSubArrayBF(STKprice *A, int n){
    double max = 0, sum;
    double t;               //time
    int low = 0, high = n-1;
    int i,j,k;

    t = GetTime();
    for (j = 0; j < n; j++){        //try all n(n-1)/2 possible situations
        for (k = j; k < n; k++){
            //sum = A[k].price - A[j].price;    // n^2 complexity version
            sum = 0;
            for(i = j; i <= k; i++){
                sum += A[i].change;
            }
            if(sum > max){
                max = sum;
```

```
                low = j;
                high = k;
            }
        }
    }
    t = GetTime() - t;

    //print result
    printf("Brute-force approach:\n");
    printf("  CPU time %g s\n", t);
    if(low != 0)printf("  Buy: %d/%d/%d at $%g\n",A[low-1].year,
                    A[low-1].month,A[low-1].day,A[low-1].price );
    else printf("  Buy: %d/%d/%d at $%g\n",A[0].year,
                    A[0].month,A[0].day,A[0].price );
    printf("  Sell: %d/%d/%d at $%g\n",A[high].year,A[high].month,
                                        A[high].day,A[high].price );
    printf("  Earning: $%g per share.\n",max);
}


RETval MaxSubArray(STKprice *A, int begin, int end){

    int mid;
    RETval Ans,lret,rret,xret;      //store return values from divided aprts
    double lsum,rsum,xsum;

    if(begin == end){
        Ans.low = begin;
        Ans.high = end;
        Ans.sum = A[begin].change;
        return Ans;
    }
    mid = (begin + end) / 2;
    lret = MaxSubArray(A,begin,mid);
    rret = MaxSubArray(A,mid+1,end);
    xret = MaxSubArrayXB(A,begin,mid,end);

    lsum = lret.sum;
    rsum = rret.sum;
    xsum = xret.sum;

    if(lsum >= rsum && lsum >= xsum){ //left side returns maximum
        Ans.low = lret.low;
        Ans.high = lret.high;
        Ans.sum = lsum;
    }
    else if (rsum >= xsum){         //right side
        Ans.low = rret.low;
        Ans.high = rret.high;
        Ans.sum = rsum;
```

```
    }
    else{                           //cross boundary
        Ans.low = xret.low;
        Ans.high = xret.high;
        Ans.sum = xsum;
    }
    return Ans;
}

RETval MaxSubArrayXB(STKprice *A, int begin, int mid, int end){
    double lsum = 0,rsum = 0;
    int low = mid;
    int high = mid + 1;
    double sum = 0;
    int i;
    RETval Ans;

    for(i = mid; i >= begin; i--){      //find left side max sum
        sum = sum + A[i].change;
        if(sum > lsum){
            lsum = sum;
            low = i;
        }
    }
    sum = 0;
    for(i = mid + 1; i <= end; i++){    //find at right side
        sum = sum + A[i].change;
        if(sum > rsum){
            rsum = sum;
            high = i;
        }
    }

    Ans.low = low;
    Ans.high = high;
    Ans.sum = lsum + rsum;
    return Ans;
}

void PrintResult(STKprice *A, RETval result, double time){
    int low = result.low;
    int high = result.high;
    double sum = result.sum;

    printf("Divide and Conquer approach:\n");
    printf("  CPU time %g s\n", time);
    if(low != 0)printf("  Buy: %d/%d/%d at $%g\n",A[low-1].year,    //to avoid

                    A[low-1].month,A[low-1].day,A[low-1].price );   //segfault
    else printf("  Buy: %d/%d/%d at $%g\n",A[0].year,
```

```
                    A[0].month,A[0].day,A[0].price );
    printf("  Sell: %d/%d/%d at $%g\n",A[high].year,A[high].month,
                                        A[high].day,A[high].price );
    printf("  Earning: $%g per share.\n",sum);

}



int main()
{
    int Ndays;
    int i;
    double t;
    STKprice *Prices;
    RETval result;

    scanf("%d",&Ndays);
    Prices = malloc(Ndays * sizeof(STKprice));

    for ( i = 0; i < Ndays; i++){
        scanf(" %d %d %d %lf", &Prices[i].year, &Prices[i].month,
                                 &Prices[i].day, &Prices[i].price );
    }

    Prices[0].change = 0;       //calculate the price changes
    for ( i = 1; i < Ndays; i++)
        Prices[i].change = Prices[i].price - Prices[i-1].price;


    printf("N = %d\n",Ndays );
    MaxSubArrayBF(Prices,Ndays);

    t = GetTime();
    for ( i = 0; i < 1000; i++)
    {
        result = MaxSubArray(Prices,0,Ndays-1);
    }
    t = GetTime() - t;
    PrintResult(Prices,result,t/1000);


    return 0;
}
```

Score: 92

[Space complexity] of the divide-and-conquer approach needs to include memory space needed due to recursion.

[Observation] can discuss the earning for different lengths of time.