EE 3980 Algorithms

Homework 5. Trading Stock

105061110 周柏宇

2020/4/12

1. Introduction

In this homework, we implemented two maximum subarray algorithms with brute-force approach and divide and conquer approach to solve the trading stock problem, which aims to find the optimal buy and sell date that maximizes the earning. To study their performance, we executed both algorithms with a different number of stock information. In the end, we recorded the result and observe the growth of CPU time to verify our derivation of their time complexity.

2. Analysis & Implementation

2.1 Maximum Subarray Problem and Stock Trading

A maximum subarray problem is a problem that aims to maximize the sum of a subarray of an array A with size N, i.e.

$$\max_{\text{low,high}} \sum_{i=\text{low}}^{\text{high}} A[i]$$

$$\text{s.t. } 1 \le \text{low} \le \text{high} \le \text{N}$$

In this homework, our objective is to solve the one-buy-one-sell stock

trading problem. We try to calculate the day to buy and sell that maximize our earning, which can be transformed into a maximum subarray problem by adding "change of price" to our stock information. Therefore, the stock trading problem is equivalent to finding the maximum subarray of the "change of price" array. Notice that if we obtain the optimal range to be [2:3], it means that $(P2-P1)+(P3-P2)=P3-P1$ is the maximal earning, but the actual buy date is at index 1 and the sell day is at index 3.

Index	1	2	3	4
Price	P1	P2	P3	P4
Change of price	0	P2 – P1	P3 – P2	P4 – P3

2.2 Maximum Subarray – Brute-Force Approach

```
1. // Find low and high to maximize ΣA[i], low <= i <= high.
2. // Input: A[1 : n], int n
3. // Output: 1 <= low <= high <= n and max
4. Algorithm MaxSubArrayBF(A, n, low, high)
5. {
6.     max := 0; // initialize
7.     low := 1;
8.     high := n;
9.     for j := 1 to n do { // try all possible A[j : k]
10.          for k := j to n do {
11.              sum := 0;
12.              // summation of A[j : k]
13.              for i := j to k do {
14.                  sum := sum + A[i];
15.              }
16.              // record the maximum value and range
```

```
17.                 if (sum > max) then {
18.                     max := sum;
19.                     low := j;
20.                     high := k;
21.                 }
22.             }
23.         }
24.         return max;
25.     }
```

Solving maximum subarray problem using brute-force approach is simply enumerating all the possible subarrays of array A, and calculating the sum of each subarray to see if it is larger. If it is larger, then we record the range and the maximum value.

As we can see, the brute-force approach has three layers of loop, if we count the number of times line 14 is executed, then it will be

$$\sum_{j=1}^{n}\sum_{k=j}^{n}\sum_{i=j}^{k} 1 \approx \sum_{j=1}^{n}\sum_{k=j}^{n} k-j \approx \sum_{j=1}^{n}[(n-j)(-j)+\frac{(j+n)(n-j)}{2}] = \mathcal{O}(n^3)$$

As for the space complexity, we need n space to store the array, thus it is $\mathcal{O}(n)$.

Best-case, average and worst-case time complexity: $\mathcal{O}(n^3)$

Space complexity: $\mathcal{O}(n)$

2.3 Maximum Subarray – Divide-and-Conquer Approach

```
1. // Find low and high to maximize ΣA[i],
```

```
// begin <= low <= i <= high <= end.
// Input: A, int begin <= end
// Output: begin <= low <= high <= end and max
Algorithm MaxSubArray(A, begin, end, low, high)
{
    if (begin = end) then { // termination condition
        low := begin; high := end;
        return A[begin];
    }
    mid := ⌊(begin + end) / 2⌋;
    // left region
    lsum := MaxSubArray(A, begin, mid, llow, lhigh);
    // right region
    rsum := MaxSubArray(A, mid + 1, end, rlow, rhigh);
    // cross boundary
    xsum :=
        MaxSubArrayXB(A, begin, mid, end, xlow, xhigh);
    // lsum is the largest
    if (lsum >= rsum and lsum >= xsum) then {
        low := llow; high := lhigh;
        return lsum;
    }
    // rsum is the largest
    else if (rsum >= lsum and rsum >= xsum) then {
        low := rlow; high := rhigh;
        return rsum;
    }
    low := xlow; high := xhigh;
    return xsum; // cross-boundary is the largest
}
```

```
// Find low and high to maximize ΣA[i],
// begin <= low <= mid <= high <= end.
// Input: A, int begin <= mid <= end
// Output: low <= mid <= high and max
Algorithm MaxSubArrayXB(A, begin, mid, end, low, high)
```

```
6. {
7.     lsum := 0; // initialize for lower half
8.     low := mid;
9.     sum := 0;
10.      // find low to maximize ΣA[low : mid]
11.      for i := mid to begin step -1 do {
12.          sum := sum + A[i]; // continue to add
13.          if (sum > lsum) then { //record if larger
14.              lsum := sum;
15.              low := i;
16.          }
17.      }
18.      rsum := 0; // initialize for upper half
19.      high := mid + 1;
20.      sum := 0;
21.      // find high to maximize ΣA[mid + 1 : high]
22.      for i := mid + 1 to end do {
23.          sum := sum + A[i]; // continue to add
24.          if (sum > rsum) then { // record if larger
25.              rsum := sum;
26.              high := i
27.          }
28.      }
29.      return lsum + rsum; // overall sum
30.  }
```

Solving maximum subarray using divide and conquer approach involves dividing the subarray into three kinds and combining the result of each kind. In *MaxSubArray*, we divide the subarray into three kinds by considering where the low/high index locate. If both low/high index locates in the lower half region, then we call *MaxSubArray* with the range at [begin:mid]. If both low/high index locates in the upper half region, then we call *MaxSubArray* with the range at [mid+1:end].

The last kind of subarray is that the low index is at the lower region and the high index is at the upper region, i.e. cross-boundary, then we call the different function *MaxSubArrayXB* with begin, mid, end specified. Once the maximum value of three kinds of subarray obtained, we compare them and record the maximum value and the range from one of the three regions.

In *MaxSubArrayXB,* we find the optimal subarray by starting our low index from mid to begin, and record the maximum of lower half subarray; we start our high index from mid+1 to end to record the maximum of upper half subarray. Finally, the maximum subarray of the cross-boundary kind is obtained with the low index and the high index found by previous instruction and the maximum being the sum of the lower maximum and the upper maximum.

The number of comparisons of the maximum subarray problem with array size n using divide and conquer is

$$\mathrm{T(n)} = 2\mathrm{T}(\mathrm{n}/2) + \mathrm{T_{XB}(n)} + 4 \approx 2\mathrm{T}(\mathrm{n}/2) + \mathrm{T_{XB}(n)}$$

where $\mathrm{T_{XB}(n)} = \mathrm{n}$ is the number of comparisons in *MaxSubArrayXB* with the length of range = n.

Without the loss of generality, we assume $\mathrm{n} = 2^{\mathrm{k}}$, Therefore

$$T(\mathrm{n}) = 2T(\mathrm{n}/2) + \mathrm{n}$$

$$= 2\left(2T\left(\mathrm{n}/2^2\right) + \mathrm{n}/2\right) + \mathrm{n} = 2^2T\left(\mathrm{n}/2^2\right) + 2\mathrm{n}$$

$$= \cdots$$

$$= 2^{k}T\left(n/2^{k}\right) + kn = n + n \cdot \lg n = \mathcal{O}(n \cdot \lg n)$$

We need n space to store the array, thus the space complexity is $\mathcal{O}(n)$.

Best-case, average and worst-case time complexity: $\mathcal{O}(n \cdot \lg n)$

Space complexity: $\mathcal{O}(n)$

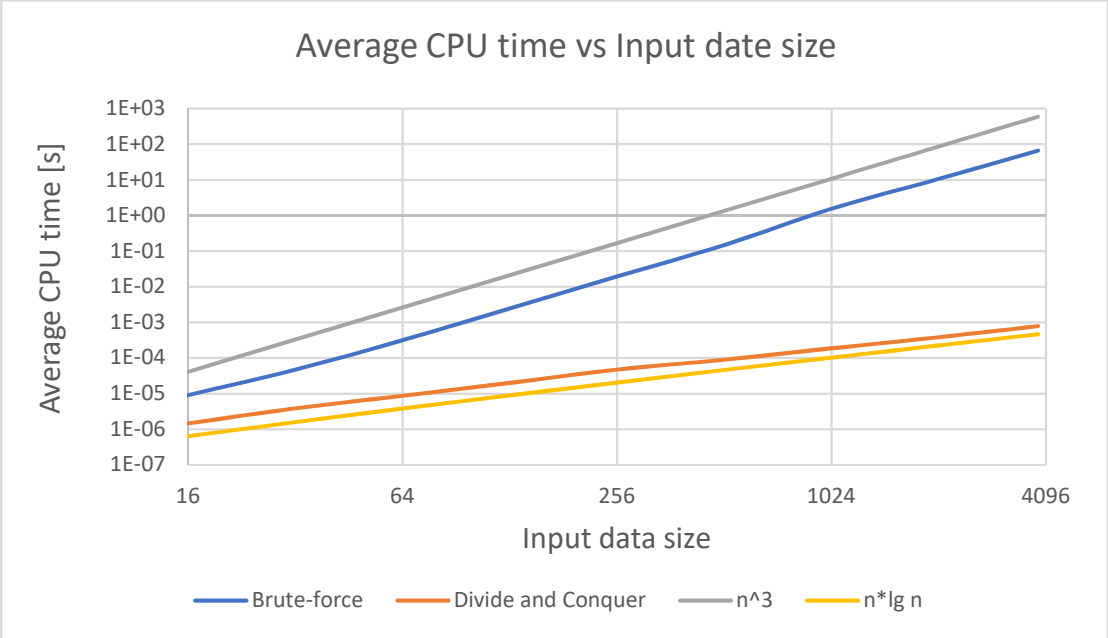
3. Result and Observation

To measure the performance, we execute the maximum subarray algorithm with brute-force approach and divide and conquer approach to solve the trading stock problem with a different number of stock information. Also, each data point in the table is the average result of 1000 executions for the divide and conquer approach. For the brute-force approach, we only execute once for every input data.

N	Brute-force approach	Divide and conquer approach
16	8.82149e-06	1.44100e-06
32	4.69685e-05	3.59893e-06
64	3.13997e-04	8.94403e-06
128	2.47407e-03	2.03071e-05
256	1.97191e-02	4.32661e-05
512	1.46862e-01	9.37350e-05
1024	1.17791e+00	1.84481e-04

2048	9.22634e+00	3.83117e-04
3890	6.95493e+01	7.83225e-04

Table 1. Average CPU time [s] vs Input data size



In the graph above, we can see that solving maximum subarray problem using brute-force approach indeed exhibits $\mathcal{O}(n^3)$ time complexity since the slope is quite similar to that of n^3, while solving the same problem using divide and conquer approach only requires $\mathcal{O}(n \cdot \lg n)$. For our largest input data N=3890, the CPU time of brute-force approach is about 10^5 time longer than that of divide and conquer approach, which is a significant difference.