10802EE 398000 Algorithms

Homework 5. Trading Stock

- ## Introduction

The purpose of this report is to find the maximum earning for one-buy-one-sell stock trading . The input files, s1.dat – s9.dat, are the history of closing price of the Google stock. The first line of each file contains the number of data entries in that file which is followed by the date and the closing price of that day. I will find the buying day and the price, the selling day and the price, and the maximum earning made per share using **two kinds of maximum subarray algorithms, Brute-Force Approach and Divide and Conquer Approach**.

I will explain both algorithms, analyze their space and time complexities, implement both of them, measure their CPU time, and compare the difference between them. Finally, I will compare the results CPU time with the theoretical results to see whether they are the same.

- ## Approach

The pseudocodes of **two kinds of maximum subarray algorithms, MaxSubArrayBF and MaxSubArray**, and their analysis are shown below.

MaxSubArrayBF is really intuitive. We use triple for loops to try all possible ranges and sum up the changes in the range. After that, we compare the sum with max to update the maximum value and range. Finally, we can get the output max, low, and high.

```
// Find low and high to maximize∑A[i], low ≤ i ≤ high.
// Input: A[1 : n], int n
// Output: 1 ≤ low,high ≤ n and max.
1 Algorithm MaxSubArrayBF(A,n,low,high)
2 {
3       max := 0; // Initialize
4       low := 1;
5       high := n;
6       for j := 1 to n do { // Try all possible ranges: A[j : k].
7             for k := j to n do {
8                   sum := 0;
9                   for i := j to k do { // Summation for A[j : k]
10                        sum := sum + A[i];
11                  }
12                  if (sum > max) then { // Record the maximum value and range.
13                        max := sum;
14                        low := j;
15                        high := k;
16                  }
17            }
18      }
19      return max;
20 }
```

Algorithm 1 The pseudocode of MaxSubArrayBF.

The space complexity of MaxSubArrayBF is equal to O(n) because it is characterized by n, the number of elements in the list A.

To try out all possible range, the total number of possibilities is $(n - 1) + (n - 2) + \ldots + 1 = n(n - 1) / 2$. Besides, we have to do the summation operation for each case. Thus, the time complexity should be $O(n^3)$.

Please note that the buying day for the stock is actually low – 1 except for low = 0. For low = 0, the buying day is equal to 0.

```
// Find low and high to maximize∑A[i], begin ≤ low ≤ i ≤ high ≤ end.
// Input: A, int begin ≤ end
// Output: begin ≤ low,high ≤ end and max.
1 Algorithm MaxSubArray(A,begin,end,low,high)
2 {
3       if (begin = end) then { // termination condition.
4             low := begin; high := end;
5             return A[begin];
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      xsum := MaxSubArrayXB(A,begin,mid,end,xlow,xhigh); // cross boundary
11      if (lsum >= rsum and lsum >= xsum) then { // lsum is the largest
12            low := llow; high := lhigh;
13            return lsum;
14      }
15      else if (rsum >= lsum and rsum >= xsum) then { // rsum is the largest
16            low := rlow; high := rhigh;
17            return rsum;
18      }
19      low := xlow; high := xhigh;
20      return xsum; // cross-boundary is the largest
21 }
```

Algorithm 2 The pseudocode of MaxSubArray.

In MaxSubArray, We divide the problem half per each recursion call. For each recursion step, we conquer the problem by combining the results of three parts which I illustrate below.

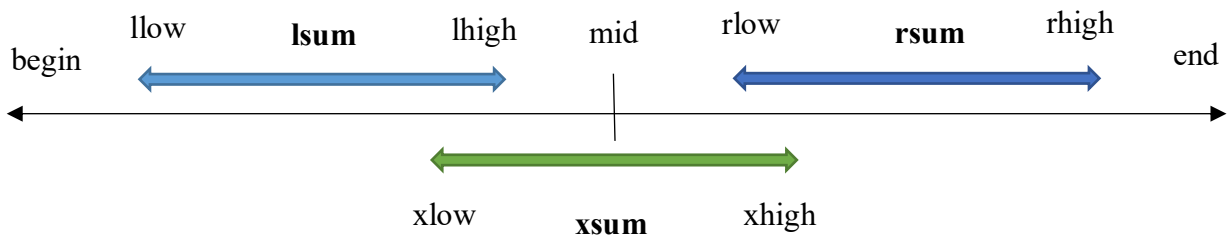

Figure 1 Illustration of MaxSubArray.

Line 3 to line 6 in MaxSubArray is the termination condition. When begin == end, there is only one element in the array, so we just return its value. Line 7 to line 9 is the “divide” part where we divide the problem into A[begin : mid] and A[mid + 1 : end]. Line 10 to the end is the “conquer” part where we determine which part has the largest return value. That is, we choose the largest among lsum, rsum, and xsum, and return the value and the range of it.

The MaxSubArrayXB shown in the next page is used to find xsum, xlow, and xhigh. They are the maximum sum and the range of the cross boundary. Line 6 to 12 find the maximum sum of A[low : mid] such that ∑A[xlow : mid] has the maximum value. Line 13 to 22 find the maximum sum of A[mid + 1 : high] such that ∑A[mid + 1 : xhigh] has the maximum value. Finally, we combine the results and return xsum = ∑A[xlow : mid] + ∑A[mid + 1 : xhigh], xlow, and xhigh.

For the space complexity of MaxSubArray, we have to consider the recursion depth which is $\log_2 n$ because we divide the problem half per each recursion call. For each recursion depth, MaxSubArrayXB will travel nearly every element in each partition once. The union of these partitions is list A with n elements (This can be observed by drawing recursion tree). Thus, the space complexity is equal to O(n) for each recursion depth. Therefore, the overall space complexity is equal to O(n * log n).

For time complexity, we can get the recursion function : $T(n) = 2 * T(n / 2) + T_{XB}(n)$, where $T_{XB}(n)$ is the number of comparisons of the algorithm MaxSubArrayXB, and $T_{XB}(n)$ is equal to n. Thus, using Theorem 3.3.8 Master Method, we can get T(n) = O(n * log n).

```
// Find low and high to maximize∑A[i], begin ≤ low ≤ mid ≤ high ≤ end.
// Input: A, int begin ≤ mid ≤ end
// Output: low ≤ mid ≤ high and max.
1 Algorithm MaxSubArrayXB(A,begin,mid,end,low,high)
2 {
3       lsum := 0; // Initialize for lower half.
4       low := mid;
5       sum := 0;
6       for i := mid to begin step −1 do { // find low to maximize∑A[low : mid]
7             sum := sum + A[i]; // continue to add
8             if (sum > lsum) then { // record if larger.
9                   lsum := sum;
10                  low := i;
11            }
12      }
13      rsum := 0; // Initialize for higher half.
14      high := mid + 1;
15      sum := 0;
16      for i := mid + 1 to end do { // find end to maximize∑A[mid + 1 : high]
17            sum := sum + A[i]; // Continue to add.
18            if (sum > rsum) then { // Record if larger.
19                  rsum := sum;
20                  high := i;
21            }
22      }
23      return lsum + rsum; // Overall sum.
24 }
```

Algorithm 3 The pseudocode of MaxSubArrayXB.

The results of the analysis of space and time complexities are shown below.

	Space complexity	Time complexity
MaxSubArrayBF	O(n)	$O(n^3)$
MaxSubArray	O(n * log n)	O(n * log n)

Table 1 The time and space complexities of MaxSubArrayBF and MaxSubArray.

● Results

data (Input)	N	CPU Time (seconds)		Earning (per share)
		MaxSubArrayBF	MaxSubArray	
s1.dat	16	1.00136E-05	1.39689E-06	9.065
s2.dat	32	4.88758E-05	3.51000E-06	35.05
s3.dat	64	3.22104E-04	9.42492E-06	96.02
s4.dat	128	2.36988E-03	2.24380E-05	110.85
s5.dat	256	1.82462E-02	4.65081E-05	213.93
s6.dat	512	1.25126E-01	7.22802E-05	371.62
s7.dat	1024	7.86945E-01	1.48792E-04	641.78
s8.dat	2048	6.03797E+00	3.12796E-04	662.49
s9.dat	3890	4.11115E+01	6.24696E-04	1384.68

Table 2 Results table

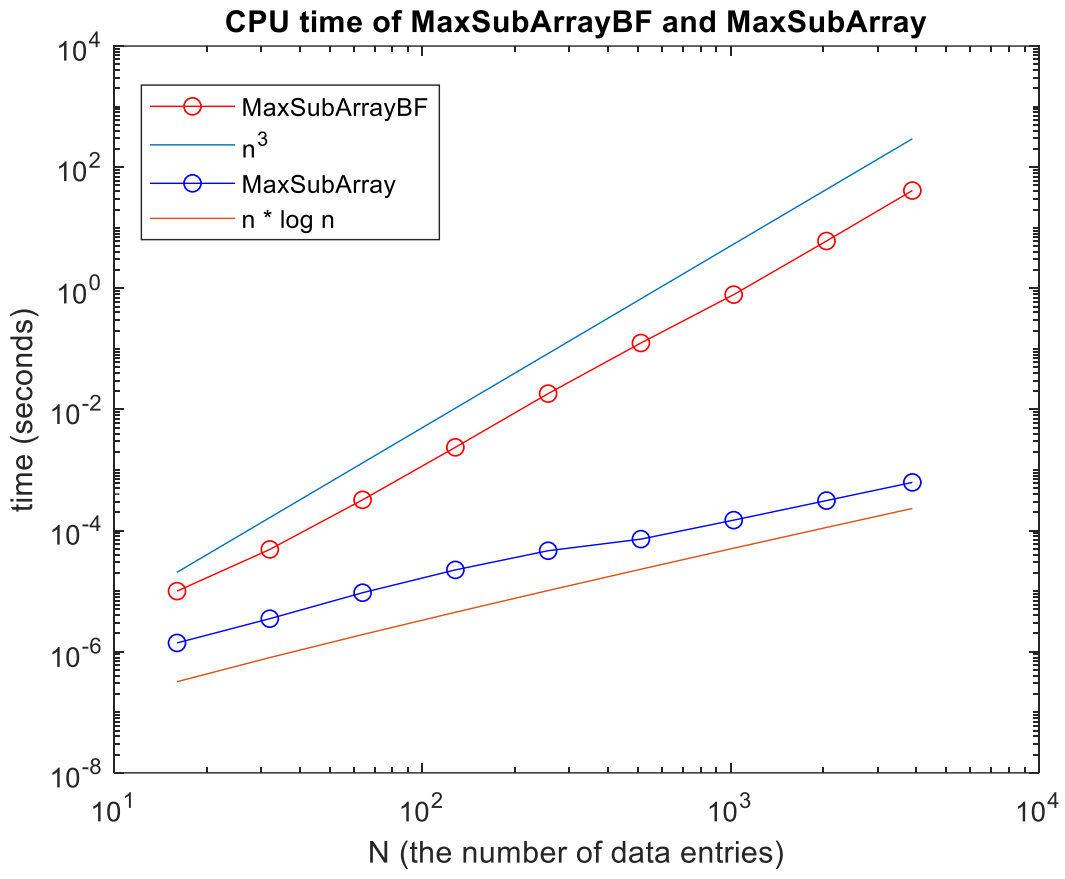

Figure 2 CPU time of MaxSubArrayBF and MaxSubArray.

The results in **Figure 2** consist with **Table 1**. From the results, we can see the power of the divide and conquer method. It can make the problem easier. Also, it is able to achieve the better time complexity. In this case, we improve the time complexity from $O(n^3)$ to $O(n * \log n)$, which is really a great progress.