Trading Stock

110191019 YU-TING CHUNG

Introduction

I implemented four different algorithms to solve the one-buy-one-sell stock trading problem. The aim is to find the optimal buy and sell dates that maximize earnings. The problem is transformed into a maximum subarray problem, where stock price differences are treated as elements in an array [A]. First, I implemented two maximum subarray algorithms: the brute-force approach and the divide-and-conquer approach. Then, I optimized the brute-force approach by replacing the summation operation with a simple subtraction, which can reduce unnecessary computations. Finally, I searched for an even more efficient algorithm which is called Kadane's algorithm to solve the problem in optimal time complexity.

To analyze the performance of these approaches, I tested them using historical Google stock (GOOGL) closing price data from nine files, s1.dat to s9.dat. Each file contains stock price records, where the first line specifies the number of entries, and the following are daily closing prices. I measured the CPU time of each algorithm on different dataset sizes and compared the results with theoretical complexity analysis to evaluate efficiency.

The Maximum Subarray problem

The Maximum Subarray problem is to find a contiguous subarray within a given array of integers (both positive and negative) that yields the highest possible sum. This means identifying a sequence of consecutive elements where their total is maximized.

Approach

- Brute-force

1. Algorithm:

```
// Find low and high to maximize ∑A[i], low ≤ i ≤ high
// Input:A[1:n], int n
// Output: 1 ≤ low, high ≤ n and max.
Algorithm MaxSubArrayBF(A,n,low,high)
{
    max:=0; // Initialize
    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{ // Summation for A[j:k]
                sum:=sum + A[i];
            }
            if sum > max then{ // Record the maximum value and range.
            max:=sum;
            low:=j;
            high:=k;
            }
        }
    }
    return max;
}
```

The brute-force algorithm finds the maximum subarray by checking all possible subarrays and calculating their sums. It initializes max to 0 and low, high to track the best subarray. It iterates over all start indices j and end indices k, and computes subarray sums. If a new maximum sum is found, it updates max, low, and high. Finally, it returns the maximum sum and its corresponding indices.

2. Proof of correctness:

Max is initialized to 0, and low and high are set to cover the entire range of possible values. Before each iteration of j, all subarrays A [1: j−1] have been checked, and max holds the maximum sum found so far.

Before each iteration of k, all subarrays A [j: k−1] have been checked, which can ensure completeness. Before each iteration of i, sum accumulates the values from A [j: i−1], so it can be sure of an accurate computation of A[j:k].

Since every subarray is considered and max is updated only if sum is greater, the algorithm ensures the maximum sum is recorded by the end of the loops. The algorithm terminates after considering all subarrays. Since it tracks the highest sum found and the corresponding indices, it correctly

outputs the maximum subarray sum along with its boundaries.

3. Time and space complexities:

Time complexity:

There are three for-loops in the brute-force and each of them can iterate n times in the worst case, where n is the size of the array. Thus, the time complexity of this algorithm is $O(n^3)$.

Space complexity:

8 integers (i, j, k, sum, max, low, high, n), and the space that we need to store the array is n, so the space complexity is O(n).

- Improved brute force

1. Algorithm:

```
1   Algorithm MaxSubArrayBFIm(A, n, low, high)
2   {
3       max := 0; // Initialize
4       low := 1;
5       high := n;
6
7       for i := 1 to n do { // Try all possible starting points A[i]
8           sum := 0; // Reset sum for each new starting point
9           for j := i to n do { // Expand the range A[i:j]
10              sum := sum + A[j];
11
12              if sum > max then { // Record the maximum value and range
13                  max := sum;
14                  low := i;
15                  high := j;
16              }
17          }
18      }
19      return max;
20  }
```

This algorithm aims to find the subarray with the maximum sum in a given array, such as sequences of stock price changes, and returns the indices of the subarray along with the maximum sum. For retaining the original stock prices instead of converting them to price changes, the summation in the innermost loop (lines 9–11) in original brute-force method is replaced with a simple subtraction operation.

The algorithm begins by initializing variables max, sum, low, and high. The max variable starts at 0. The outer loop iterates through all potential starting points i, while the inner loop examines possible ending points j. For each subarray data[i:j], the algorithm computes the sum of its elements and compares it to the current max. If the sum exceeds the current maximum, max is updated, and the indices low and high are adjusted accordingly.

2. Proof of correctness:

The algorithm starts by initializing max, low, and high to make sure that every valid subarray is considered, and they are prepared to update as the loops progress. For each iteration of i, the algorithm checks all subarrays from data [0] up to data[i-1] and keeps track of the highest sum so far in the variable max. Then, for every index i, it evaluates all subarrays from index i to j and calculates their sums to ensure completeness.

When a subarray's sum exceeds the current max, the algorithm updates both max and the indices (low and high). By the end of all iterations, it guarantees that max holds the largest sum and low and high pinpoint the boundaries of the subarray that achieves this sum.

3. Time and space complexities:

Time complexity:

There are two for-loops in the improved brute-force method and each of them can iterate n times in the worst case, where n is the size of the array. Thus, the time complexity of this algorithm is $O(n^2)$.

Space complexity:

7 integers (i, j, sum, max, low, high, n), and the space that we need to store the array is n, so the space complexity is O(n).

- Kadane's algorithm

1. Algorithm:

```
Algorithm Kadane(A, n, low, high)
{
    max := 0; // Initialize maximum sum
    sum := 0; // Current sum
    low := 1; // Start index of max subarray
    high := 1; // End index of max subarray
    tempLow := 1; // Temporary start index

    for i := 1 to n do {
        if sum + A[i] > A[i] then {
            sum := sum + A[i];
        } else {
            sum := A[i];
            tempLow := i; // Update start index
        }

        if sum > max then {
            max := sum;
            low := tempLow; // Update low
            high := i; // Update high
        }
    }
    return max;
}
```

The algorithm solves the problem by iterating over the array and dynamically maintaining the best subarray sum found up to the current index. The key idea is to either extend the current subarray (if the sum remains positive) or start a new subarray at the current element (if adding the current element results in a smaller sum). The low and high indices represent the start and end of the subarray that yields the maximum sum.

The algorithm starts with max set to 0. The low and high indices are initialized to 0, and tempLow is also initialized to 0. It represents the potential start of a subarray. As we iterate through each element, if we add

the current element to the existing subarray sum (in the code is called sum) yields a greater value, we extend the subarray. If not, we start a new subarray at the current index i, and tempLow is updated to i. When the current sum exceeds the max, we update max, as well as the indices low and high that track the start and end of the best subarray found so far.

2. Proof of correctness:

When the first element is processed, sum is initialized to the first element's value, and max is initialized to 0. Since there's only one element, max is updated to that value if it's greater than 0.

Assume that the algorithm works correctly for the first k elements, i.e., it finds the maximum subarray sum and the correct indices low and high for the first k elements. When the next element (data[k+1]) is processed, the algorithm correctly either: extends the subarray if the sum is greater than the element itself or starts a new subarray if the element is greater than the sum of the previous subarray.

At each step, the max and the corresponding low and high are updated correctly, and the algorithm continues to maintain the correct values.

Thus, the algorithm is correct for all N elements.

3. Time and space complexities:

   Time complexity:

   There is one for-loop in the Kadane's algorithm, and it can iterate n times in the worst case, where n is the size of the array. Thus, the time complexity of this algorithm is O(n).

   Space complexity:

   7 integers (i, sum, max, low, tempLow, high, n), and the space that we need to store the array is n, so the space complexity is O(n).

- Divide and Conquer

1. Algorithm:

```
1   // Find low and high to maximize ∑A[i], begin ≤ low ≤ i ≤ high ≤ end.
2   // Input: A, int begin ≤ end
3   // Output: begin ≤ low, high ≤ end and max.
4   Algorithm MaxSubArray(A,begin,end,low,high)
5   {
6       if begin = end then{ //termination condition.
7           low:=begin; high:=end;
8           return A[begin];
9       }
10      mid:=⌊(begin + end)/2⌋;
11      lsum:=MaxSubArray(A,begin,mid,llow,lhigh); //left region
12      rsum:=MaxSubArray(A,mid+1,end,rlow,rhigh); //right region
13      xsum:=MaxSubArrayXB(A,begin,mid,end,xlow,xhigh); // cross boundary
14      if lsum >= rsum and lsum>=xsum then{ // lsum is the largest
15          low:=llow;high:=lhigh;
16      return lsum;
17      }
18      else if rsum >= lsum and rsum >= xsum then{ // rsum is the largest
19          low:=rlow;high:=rhigh;
20          return rsum;
21      }
22      low:=xlow; high:=xhigh;
23      return xsum; // cross-boundary is the largest
24  }
```

The algorithm uses the divide-and-conquer approach to solve the maximum subarray problem. The goal is to find the subarray within a given range [begin, end] that maximizes the sum of its elements, and to determine the indices (low, high) that represent this subarray.

The algorithm works recursively by dividing the array into two halves and solving the problem in each half. Additionally, it calculates the maximum subarray that crosses the boundary between the two halves to ensure that the overall maximum subarray is found.

2. Proof of correctness:

When the subarray is reduced to a single element, the algorithm identifies that element as the maximum subarray, which makes sense and confirms the correctness of this step.

Assume the algorithm correctly computes the maximum subarray sums for [begin, mid] and [mid+1, end]. In the recursive process, it splits the array further and combines results from the left, right, and crossing subarrays. Because each recursive call reliably solves its respective subproblem, and the algorithm picks the largest sum from these, the overall solution is guaranteed to be accurate.

3. Time and space complexities:

Time complexity:

The number of comparisons for divide-and-conquer algorithm, MaxSubArray, is dominated by $T(n)=2\cdot T(n/2)+T_{XB}(n)$, where T_{XB} is the number of comparisons of the algorithm MaxSubArrayXB. And $T_{XB}(n)=n$

Thus, assuming $n=2^k$,

$T(n)=2\cdot T(n/2)+n$

$=2(2\cdot T(n/2^2)+n/2)+n$

$=2^2\cdot T(n/2^2)+2n$

$=\cdots$

$=2^k\cdot T(n/2^k)+k\cdot n$

$=n+n\cdot \lg n$

The computational complexity of the divide-and-conquer MaxSubArray is $\Theta(n\cdot \lg n)$.

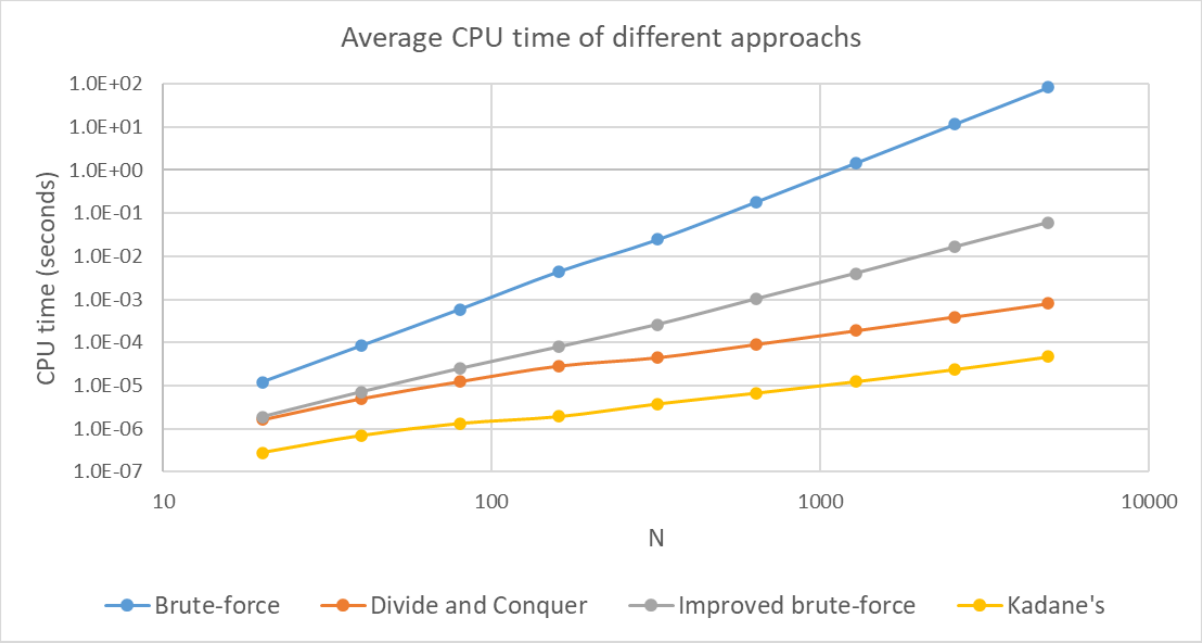
Space complexity:

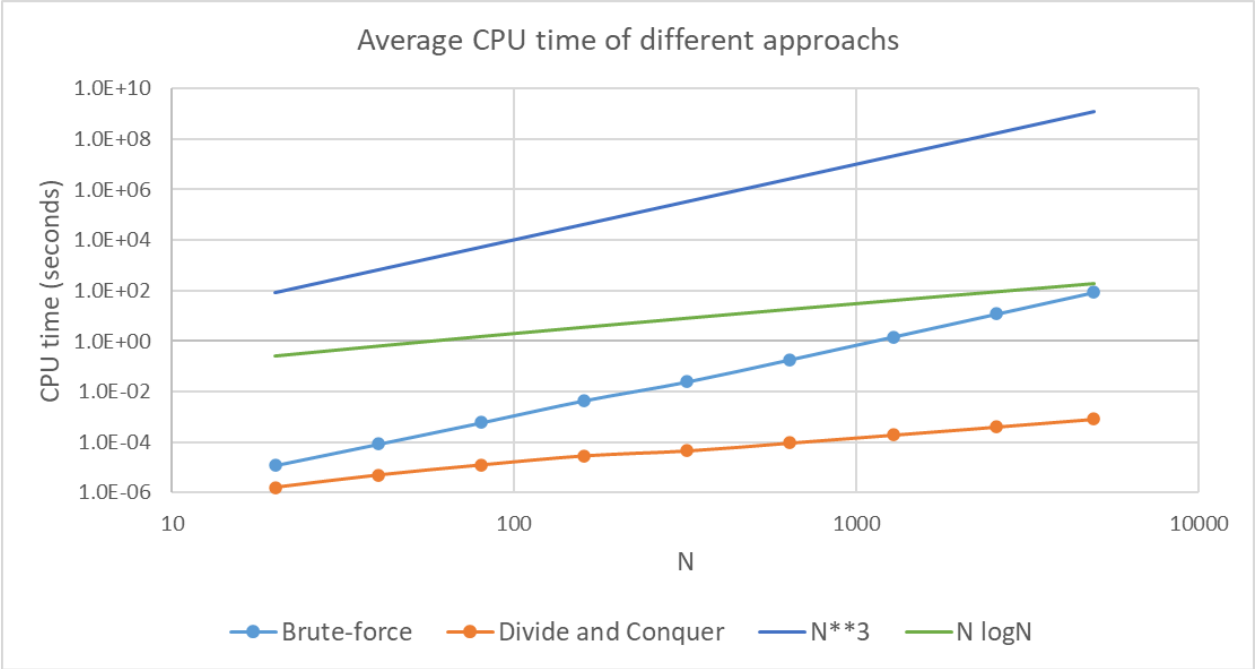
15 integers (rsum, lsum, xsum, low, rlow, llow, xlow, mid, high, rhigh, lhigh, xhigh, begin, end, n), and the space that we need to store the array is n, so the space complexity is O(n).

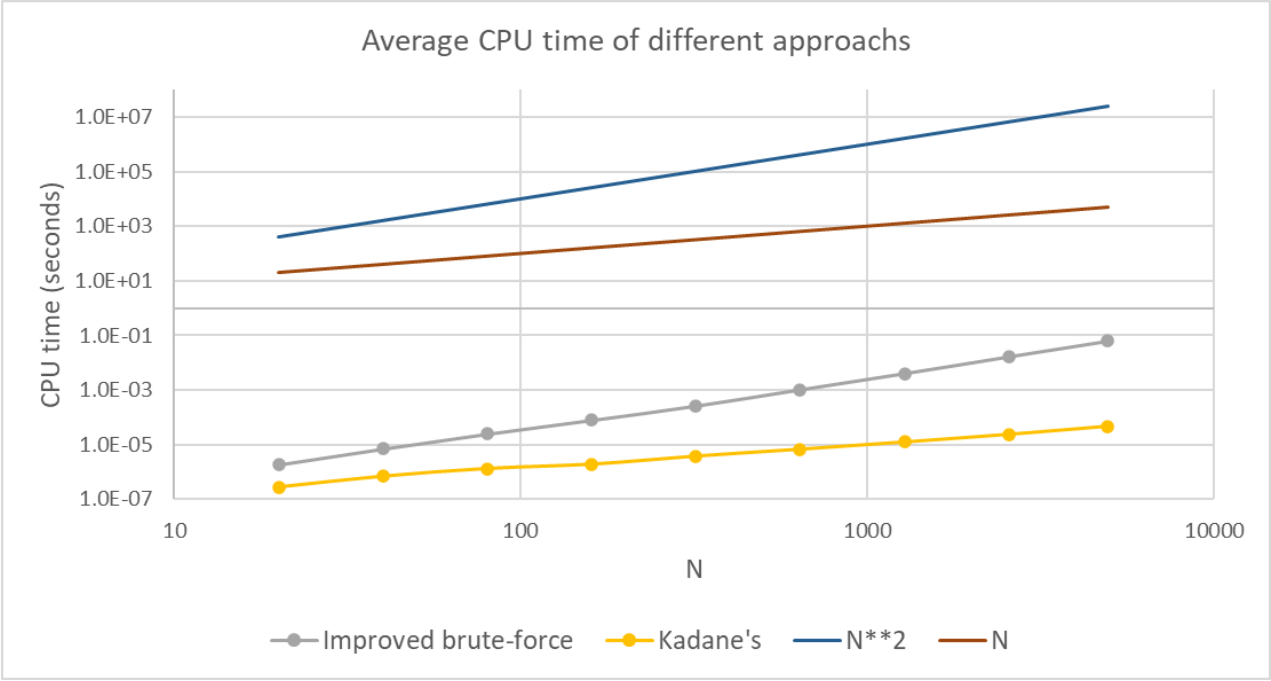
Results and observations

1. Results

Average CPU time of different approachs (seconds)				
N	Brute-force	Divide and Conquer	Improved brute-force	Kadane's
20	1.19209E-05	1.62196E-06	1.85609E-06	2.78950E-07
40	8.32081E-05	5.01680E-06	7.03907E-06	6.96898E-07
80	5.86987E-04	1.25270E-05	2.46820E-05	1.32895E-06
160	4.40192E-03	2.87571E-05	7.86109E-05	1.91116E-06
320	2.43950E-02	4.48871E-05	2.61890E-04	3.76916E-06
640	1.84897E-01	9.15210E-05	1.03642E-03	6.71482E-06
1280	1.46441E+00	1.88966E-04	4.02942E-03	1.23770E-05
2560	1.17335E+01	3.95047E-04	1.65646E-02	2.35920E-05
4941	8.42310E+01	8.02901E-04	6.11846E-02	4.65341E-05







2. Observation

(fast to slow)

Kadane's > Divide and Conquer > Improved brute-force > Brute-force

From the graphs above, we can see that the Brute-force approach is the

slowest because its time complexity is $O(n^3)$, which the slope is quite similar to that of n^3. Thus, its CPU time is far more than other methods.

In addition, although the improved brute-force approach replaces the summation in the innermost loop in original brute-force method with a simple subtraction operation, there are still two for-loops in it. Therefore, the time complexity is still larger than Kadane's and Divide-and-Conquer.

Kadane's algorithm is the fastest because Kadane's algorithm is iterative, so it can avoid the overhead of recursive calls. However, Divide-and-Conquer requires recursion. That's why Kadane's algorithm is faster than Divide-and-Conquer.

Conclusion

1. Time and space complexities comparison:

Algorithm	Time Complexity	Space Complexity
Brute-force	$O(n^3)$	O(n)
Improved brute-force	$O(n^2)$	O(n)
Kadane's	O(n)	O(n)
Divide and conquer	O (n lg n)	O(n)

2. Reducing the number of for-loops significantly improves execution time, as seen in the transition from Brute-force $O(n^3)$ to Improved Brute-force $O(n^2)$.

3. Kadane’s Algorithm is faster than Divide and Conquer because it avoids recursion and processes the array iteratively in a single pass.
4. Kadane’s Algorithm is purely iterative, so it efficiently processes the array with minimal overhead, which makes it the fastest among the four approaches.