The Day Trader Problem

COSC 320 - Algorithm Analysis
Professor: Dr. Yong Gao
November, 2018

Parsa Rajabi
Rohan Chauhan
Rachelle Gelden
Jasper Looman

Table Of Contents

1. Problem Formulation

1.1 The Problem

This problem is asking for the largest sum of consecutive profits and losses possible on the input set of days.

```
Instance: A set of n days of profits/losses {$d_1$, $d_2$, … , $d_n$}
Question: Largest sum of consecutive days possible?
```

We can restate this problem from a computer science perspective as finding the largest sum of entries possible in any sub-array of an array containing all of the trader's net profit/loss per day. Now by calculating this maximum sub-array sum, we can know the largest possible sum of consecutive days of trading in the original problem, which is the strength of the day trader.

```
Instance: An array A of n profits/losses {$d_1$, $d_2$, … , $d_n$}
Question: Largest possible sum in any sub-array?
```

Note that we are not considering an empty set (strength of zero) to be a possible solution to this problem.

2. Algorithm Design and Analysis

To approach this problem, we considered several different approaches, starting from a brute force approach, and refining our approach by making observations on how we can improve the runtime of the algorithm.

2.1 A Brute Force Approach

To begin, we considered the Brute Force approach to solving this problem. This approach is the most intuitive for someone who does not have prior experience with computation and algorithms.

2.1.1 The Idea

The Brute Force approach will sum the entries of every possible sub-array of an array A, and return the maximum sum it finds along the way. This largest sum will be the maximum of consecutive trading days in the set of trading days corresponding to the array A, and thus the strength of a day trader for that set of trading days.

2.1.2 Pseudo-code

```
strengthBruteForce(A):
    // A = input array of days of profits/losses
    max ← A[0];
```

```
n ← A.length;
for i = 0 to n-1 do
 |   for j = i to n-1 do
 |     |   strength ← A[i];
 |     |   for k = i+1 to j do
 |     |    |   strength ← strength + A[k];
 |     |   end
 |   if (strength > max) then
 |     |   max = strength;
 |   end
end
return max;
```

2.1.3 Runtime and Space Complexity

The runtime of this algorithm will be O(n^3) as each for-loop takes O(n) time. The space complexity of this algorithm will be O(1) since only two variables are being written to the memory.

Since this approach is very exhaustive and time-inefficient, albeit correct, we look at what improvements are possible to make.

With clever observation, we can see that it is possible to improve the algorithm by noticing that the sum of any sub-array from index i to index j, let's denote this as `sum(i,j)`, is equal to the sum of the sub-array from index i to index j-1 plus the value of day j. This gives us the following formula:

Equation 1: `sum(i,j) = sum(i,j-1) + A[j]`

This allows us to build a bottoms up approach to find the solution, using solutions we calculate in the previous iteration to efficiently calculate the sum of a sub-array by storing it.

2.2 A Dynamic Programming Approach

To improve upon the Brute Force approach, we can use this recursive formula to skip recalculating some of the sums of sub problems to save time. This is more of a Dynamic Programming approach.

2.2.1 The Idea

Based on the formula from above we can calculate a matrix that contains the sum of profits/losses for all possible consecutive sub-arrays in a given a set of profits over a time period. Each entry in the matrix P_{ij} will be equal to `sum(i,j)` and the greatest element in the matrix is the largest sum of any sub-array of A, and thus the day-trader's strength.

For $i,j \in \mathbb{N}_0$ where $i = j$ $\quad (\mathbb{N}_0 := \mathbb{N} \cup \{0\})$

Let $\{d_0, d_1, \dots, d_j\}$ = a set of profits for consecutive trading days

Let $P_{i,j}$ = the net profit earned over days i to j inclusive.

The matrix will look like this:

$$\begin{array}{c|ccccc}
 & \boldsymbol{d_0} & \boldsymbol{d_1} & \boldsymbol{d_2} & \dots & \boldsymbol{d_n} \\
\hline
\boldsymbol{d_0} & P_{00} & P_{01} & P_{02} & \dots & P_{n0} \\
\boldsymbol{d_1} & P_{10} & P_{11} & P_{12} & \dots & P_{n1} \\
\boldsymbol{d_2} & P_{20} & P_{21} & P_{22} & \dots & P_{n2} \\
\vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\
\boldsymbol{d_n} & P_{0n} & P_{1n} & P_{2n} & \dots & P_{nn}
\end{array}$$

To compute any given $P_{i,j}$, we must use the following recursive formula:

$$P_{i,j} = P_{i,\, j-1} + P_{j,j}$$

Once the matrix is computed, the largest value in the matrix is the day trader's maximum performance and strength.

2.2.2 Pseudo-code

```
strengthDynamicProgramming(A):
    // A = input array of profits
    n ← A.length
    M ← Array[days][days]

    //initialize values on diagonal of matrix
    for i = 0 to n do
     |   M[i][i] = A[i];
    end

    for j = 1 to n do
     |   for i = 0 to j do
     |    |   if (i < j) then
     |    |    |   M[i][j] = M[i][j-1] + M[j][j];
     |    |    |   if (M[i][j] > max) then
     |    |    |    |   max ← M[i][j];
     |    |    |   end
     |    |   end
     |   end
    end
    return max;
```

2.2.3 Runtime and Space Complexity

The runtime of this algorithm is O(n) + O(n)*O(n) = O(n^2) as each for-loop will take at most O(n) time. The space complexity is O(n^2).

While this Dynamic Programming approach is a significant improvement upon the Brute Force approach, we were determined to find an algorithm with an even better running time.

2.2.4

To improve upon this further, we can implement this approach using a one dimensional array. We can see that to calculate the day-trader's strength for a given day *i*, we only ever need the value of the sum of profits from the the beginning of the day trader's set to the previous day and the day trader's profit from day *i*. This means its possible to use a 1 dimensional array, by updating the value of each entry A[i] to be the sum of all entries from i to the current iteration j, Along the way, we still keep track of the maximum found, and the runtime isn't affected, but this way we reduce the space complexity to O(n).

2.3 Improving Further

To improve upon the best running time found to this point, we can make the further observation that for some ending index `j`, since `sum(i,j) = sum(i,j-1) + A[j]` and `A[j]` is the same for every i entry we calculate in the array, the maximum of all the `sum(i,j)` will be the one starting at day i for which `sum(i,j-1)` was maximum. Note that if all `j-1` sums were negative, then just `A[j]` would be the maximum sum for this ending day `j`. Let's denote this maximum sum for the sub-array ending at `j` as `OPT(j)`. `OPT(j)` is the maximum sum possible for any sub-array starting at `i`, `0 ≤ i ≤ j`, and ending at `j`. Therefore, `OPT(j)` is the maximum possible sum for all sub-arrays with an ending index at `j`, and thus the strength of a day trader if we consider the sub-problem for days from `0` to `j` instead of `0` to `n`. This means we can write the following:

```
OPT(j) = max(A[j], OPT(j-1) + A[j])
```

This means if we look at our algorithm that filled out a matrix, we would only be keeping track of one value per ending day column `j`. Then if we do the same space complexity reduction by just updating a single value as we loop through each iteration `j`, then we can write an algorithm using only a single variable. This single variable will track the optimal solution to the sub problem ending at that day, and thus if we keep track of the maximum we find along the way, we have the maximum sum of any sub-array in the whole array A.

2.3.1 The Idea

The primary concept behind this new algorithm is to split the initial problem into smaller sub-problems, specifically by grouping all sub-arrays we have to calculate by their ending index `j`. If we know the maximum possible sum on a sub-array for all sub-arrays that end at that index `j`, then we find the maximum of all of those, then we have the maximum of all possible sub-arrays. We return this maximum as the strength of the day trader.

2.3.2 Pseudo-code

```
strengthGreedy(Array A):
     // A = input array of profits
     OPT ← A[0]; //keeps track of consecutive days
     maxFound ← OPT; //the maximum OPT ever found

     for i = 1 to A.length-1 do
      |    OPT ← max(A[i], OPT + A[i]);
      |    maxFound ← max(streak, maxFound);
     end
     return maxFound;
```

2.3.3 Runtime and Space Complexity

The runtime of this algorithm is O(n) because there is only one for loop that will iterate at most `n` times.

The space complexity is O(1) because we are only storing two variables.

This approach is a significant improvement upon the runtimes from all of our previous algorithms formulated through Brute Force, Divide & Conquer and Dynamic Programming approaches. For this reason, we selected the Greedy Algorithm approach to solve the maximum sub-array problem to compute a day trader's strength.

3. Proof of Correctness

Note that we will not be proving the correctness of the algorithms designed using the Divide & Conquer and Dynamic Programming approaches since we will not be implementing these algorithms due to their computed worst-case running times.

As described in the *Improving Further* section, this algorithm makes use of the fact that we can split the input array A into sub-arrays. Note that any maximum sub-array sum will have a last index ***i***, and the algorithm makes use of this. Lets denote the maximum sum with last index ***i*** as S_i , Note that the solution then is equal to max(S_0, S_1, … , S_n). Thus if we calculate each S_i for $0 \le i \le n$, if we keep track of the maximum S_i we find along the way as *maxFound*, then *maxFound* will be the correct maximum sub-array sum possible in all of A.

3.1 Proof by induction

Base Case:
B_0 is trivial, since the only possible sum is the element `A[0]`;

Inductive hypothesis:
Assume we have the maximum sum S_k at some iteration $0 \le k \le n$. Then examining S_{k+1}, either S_{k+1} includes S_k or it does not.
Case S_k is included. Then

$$S_{k+1} = S_k + A[k+1];$$

Case S_k is not included. Then

$$S_{k+1} = A[k+1];$$

Therefore, given S_k we know that

$$S_{k+1} = \max(A[k+1],\ S_k + A[k+1]);$$

Thus by induction we know all S_i for $0 \le i \le n$.

In our algorithm's pseudocode, $OPT = S_i$ and *maxFound* keeps track of the biggest S_i found so far as each is calculated. Thus after n iterations, *maxFound* $= \max(S_0, S_1, \ldots, S_n)$, and thus will return the correct maximum possible sub-array sum in all of A.

$\square$

4. Analysis

4.1 Comparison Between Algorithms

During the problem formulation stage, we constructed algorithms that improved upon the initial Brute Force algorithm. Analyzing the difference in the runtimes between the initial Brute Force algorithm and the final algorithm we ended up at, we see that the improvements were very successful. The runtime of the Brute Force algorithm is O(n^3), due to the triple nested for loop, and the runtime of the improved Dynamic Programming algorithm takes O(n) runtime because of the single for loop. O(n) compared to O(n^3) is an improvement of O(n^2) order of magnitude.

Considering the space complexity of these two algorithms, they are equal because both are simply storing a few integer variables. No arrays are being stored, they are only being traversed through.

5. Implementation & Empirical Correctness

5.1 Java Code

Please find our source code attached containing the algorithms and testing class.

5.1.1 StrengthBruteForce Algorithm Java Code

```
public static int strengthBruteForce( int [] A ) {
      int max = A[0];
      int days = A.length;

      for ( int i = 0; i < days; i++ ) {
            for ( int j = i; j < days; j++ ) {
                  int strength = 0;
                  for ( int k = i; k <= j; k++ ) {
                        strength += A[k];
                  }
                  if ( strength > max ) {
                        max = strength;
                  }
            }
      }
      return max;
}
```

5.1.2 StrengthCGLR Algorithm Java code

```
public static int strengthCGLR(int [] A) {
      int streak = A[0];
      int maxFound = A[0];

      for (int i = 1; i < A.length; i++) {
```

```
                streak = Math.max(A[i], streak + A[i]);
                maxFound = Math.max(streak, maxFound);
            }
            return maxFound;
        }
```

5.2 Testing

5.2.1 Test 1

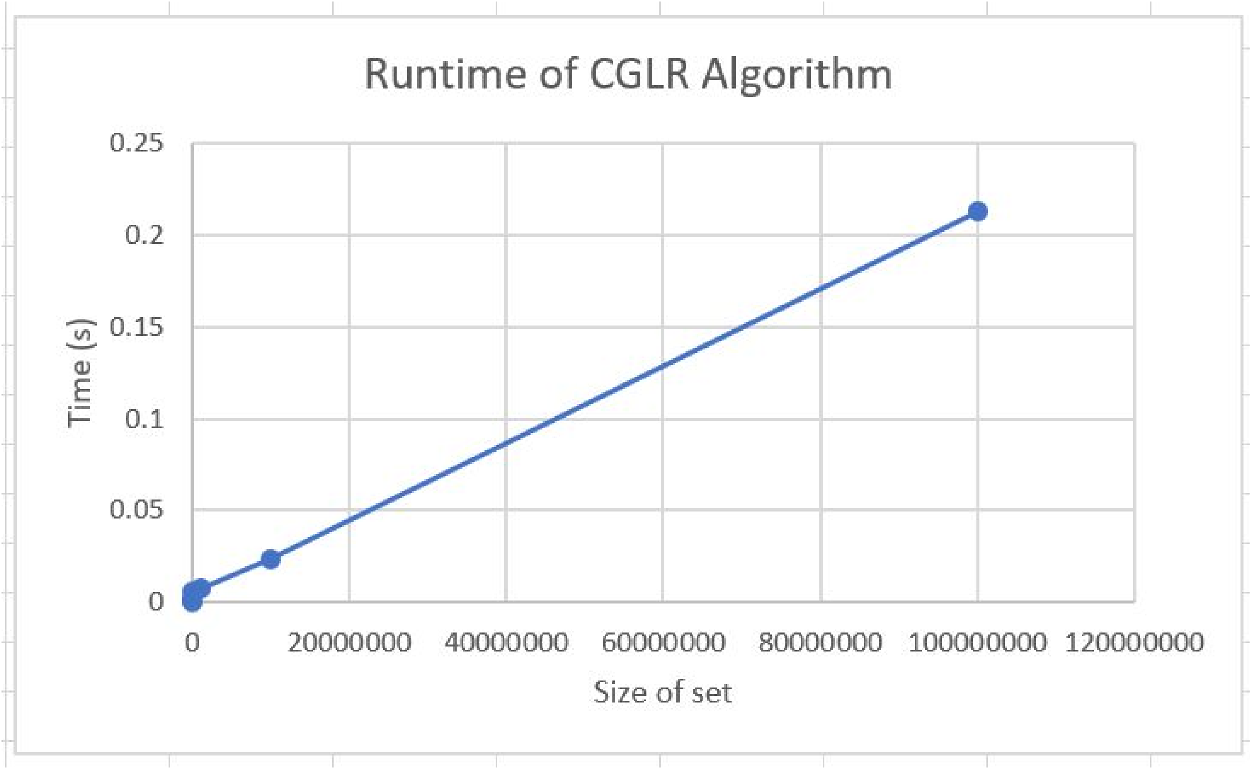
To test the correctness of our algorithms, we ran both algorithms and ensured that they both returned the same maximum performance value for the same input set.

5.2.2 Test 2

We generated a random set of negative integers and ensured that not only both algorithms returned the same value, but also that the strength of the day trader is negative (not zero) as our algorithms are not considering an empty set of strength zero as a possible strength for a day trader.

5.3 Empirical Evaluation

5.3.1 Improved Algorithm



11

Figure 5.1 Displays a constant rate of change for the improved algorithm constructed which is indicative that its runtime is O(n).

5.3.2 Brute Force Algorithm

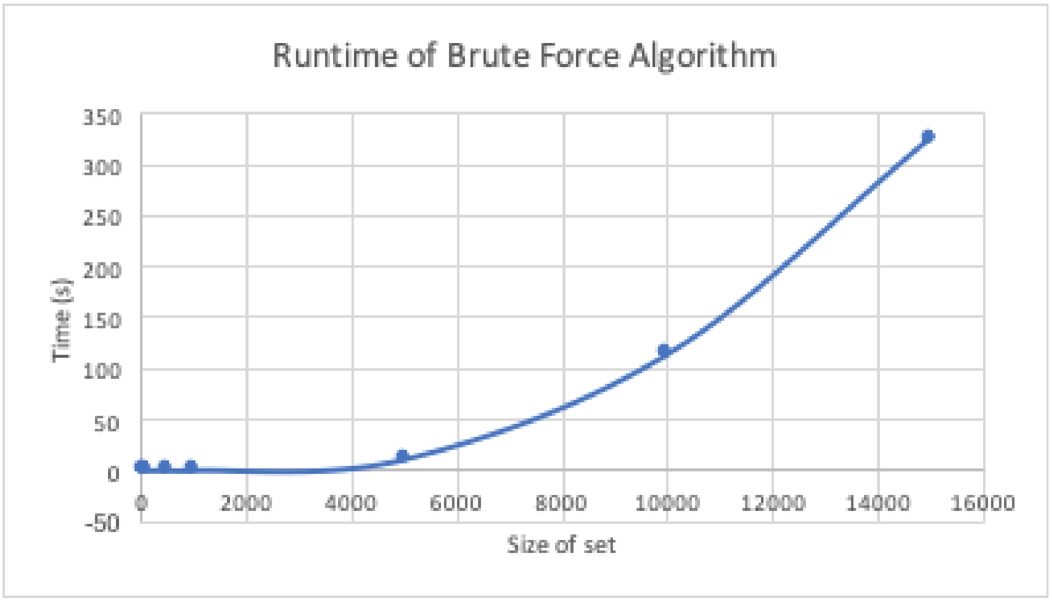


Figure 5.2 Displays cubic increase in rate of change for the Brute Force algorithm constructed which is indicative of our $O(n^3)$ runtime.