zachary_hightower_433_project_3_report

Pseudo-code

Brute force

ALGORITHM `bruteForceMaxProfit(prices):`

```
initialize buyIndex, sellIndex, maxProfit
for i from 0 to length of prices - 1:
for j from i + 1 to length of prices:
calculate tempProfit as prices[j] - prices[i]
if tempProfit > maxProfit:
maxProfit -> tempProfit
buyIndex -> i
sellIndex -> j
return Result(buyIndex, sellIndex, maxProfit)
```

Divide and Conquer

ALGORITHM `maxProfitDivideAndConquer(prices, low, high):`

```
if high <= low:
return Result(-1, -1, 0)
mid = (low + high) / 2
leftResult = maxProfitDivideAndConquer(prices, low, mid)
rightResult = maxProfitDivideAndConquer(prices, mid + 1, high)
initialize minPrice, maxPrice, buyIndex, sellIndex
for i from low to high:
if i <= mid and prices[i] < minPrice:
minPrice -> prices[i]
buyIndex -> i
if i > mid and prices[i] > maxPrice:
maxPrice -> prices[i]
sellIndex -> i
crossMaxProfit = maxPrice - minPrice
if leftResult.profit == -infty and rightResult.profit == -infty and crossMaxProfit < 0:
return Result(buyIndex, sellIndex, crossMaxProfit)
if leftResult.profit >= rightResult.profit and leftResult.profit >= crossMaxProfit:
```

```
return leftResult
if rightResult.profit >= leftResult.profit and rightResult.profit >=
crossMaxProfit:
return rightResult
return Result(buyIndex, sellIndex, crossMaxProfit)
```

Program Rundown

Imports

We import a few simple Input and Output packages so that we can use them for the file and data reading. The only other package is for our IO exceptions. Standard stuff.

getAlgorithmName

This method's entire purpose is to compartmentalize the grabbing of the name of the algorithm we want to use. It also outputs, in the case of there being an issue, the names of the actually implemented algorithms in the program.

Keeping it as a separate method helps us to have a cleaner looking overall program that can be more easily read.

Result

This class and its associated constructor helps us to have a simple way to output the actual results of what we've found. It contains the three things we're interested in for the output. The spot we bought. The spot we sold. The profit per share of stock.

It also helps us later down the line as some of the variables we would otherwise have to initialize throughout the rest of the program are held here and can be easily updated and referenced as we go along.

binReader

This method just handles taking in the information we want to parse from the provided bins. It uses the imports we got from the IO packages and creates the prices array that we work with throughout the rest of the program.

bruteForceMaxProfit

This is the brute force implementation of finding the maximum profit. It uses two nested for loops and runs on Exponential n^2 time. The general logic of it is that it goes through and finds every possible combination of the numbers in our prices array. The way one might calculate a

2d matrix. It keeps track of the largest difference between them and when it finishes, it outputs that for us using the result object we made earlier.

maxProfitDivideAndConquer

Here's the bulk of the actual program. We start by checking the array with the if statement to see if there are actually any values contained within it. If not we output the -1,-1,0 result to show that there's an issue with the provided array.

Then we find the mid point and set it.

We call the method recursively for the two divisions to the left and to the right of the midpoint and use our result objects to keep track of them.

After doing this, we get the variables for the rest of the program's logic instantiated.

We use a for loop and two if statements to populate the variables we've just instantiated with the correct values. It will first select whatever it comes across as the minPrice, buyIndex, maxPrice, sellIndex. Then it will update them as the for loops iterates through the space between our low and high variables.

We find the crossing point of our maximum profit by finding the difference between our max and min prices.

Then we proceed to the logic for handling of the three possible cases, and the case when we have no actual profit at all.

The first handles the case in which we have no profit, we just set the result as we would if we'd finished going through the logic. The next statement checking to see if our max profit is found in the left side. The next checks the right side. The last case handles the possibility that we need to create the object and set it as the neither the left nor the right result. Then, we return the result that we've found.

main

This is the collection of all our method calls and some error handling for the command line arguments. We check to see if the CL argument has the correct length. If so, we proceed to grab the filename and the algorithm selected.

After that, we read through the selected bin, and then we find the result using the selected method. If there's an issue with the selected algorithm, we output an error.

Then, we present the result obtained in the previous method calls, and if we've gotten to this point and have not triggered an error, but need to, we have the logic to do so here, for both the file and the algorithm number.

If there's an issue with adding the explanatory Bought: Sold: and Profit: portions, along with leaving the Profit as a non-rounded value, I'd ask for a chance to resubmit with that changed.

Time Complexity Analysis

Brute Force

The brute force method uses a nested loop to iterate through every possible pair of buying and selling combinations.

Where n is the number of elements in our prices

The outer loop will go from i = 0 to n -2
The inner loop will go from j = i+1 to n-1

We can see from this that the time complexity will be around $\Theta(n^2)$

The recurrence equation of this portion will look like this

$$T(n) = T(n-2) \times n - 1$$

$$T(n) = \Theta(n^2)$$

So we can see that after we drop the constants from the equation we have a brute force time complexity of n^2

Divide and Conquer

The divide and conquer method uses a system of recursive calls and the combination of the left and right halves to find the maximum profit. It also handles the calculation of the cross section of the halves so that we don't miss anything.

- Division, calculation of the midpoint, takes constant time
- Conquering, which is where we call our function recursively a great deal of times in order to iterate through all our possibilities, is more complex. It is called twice on subarrays of roughly $\frac{n}{2}$ size. So we can say that the time complexity here is $2 \times (\frac{n}{2})$
- Combination and calculation, we find the maximum profit possible within the divided portions of our array. This takes linear time, as we must iterate over the full breadth of the array. The time complexity here is n

So, knowing all this, we can then write the recurrence relation as the following

$$T(n) = 2T(\frac{n}{2}) + n$$

This is of the form which we can use the Master Theorem on. Making this a far simpler solve than it would be otherwise.

$$a = 2, b = 2, d = 1$$

$$T(n)\ is\ \Theta(n^d\ log\ n)\ if\ a = b^d$$

$$2 = 2^1$$

So our time complexity is

$$T(n) \in \Theta(n\ log\ n)$$