Module 7 Homework - Divide and Conquer

We have seen divide-and-conquer applied to design 4 algorithms:

- binary search
- mergesort
- quicksort
- quickselect

Here, we apply divide-and-conquer to design a new algorithm that solves a new problem - how much could we have made off Bitcoin?

Description

You are provided with list that contains the prices of Bitcoin at market opening for specific period of time. Our goal is to figure out the maximum profit we could have made . There are two constraints to this problem:

- We can only buy and sell once
- The sell date must be after the buy date

A brute source solution here is fairly straight forward - what's the optimal profit if I bought on day 1? What if I bought on day 2? And so on:

```python
# brute force solution
def max_profit_brute(L):
    n = len(L)
    max_sum = 0 # assume we can at least break even - buy and sell on the same day

    # outer loop finds the max profit for each buy day
    for i in range(n):
       # total profit if we bought on day i-1 and sold on day i
        total = L[i]
        if total > max_sum: max_sum = total

        for j in range(i+1, n):
            total += L[j] # total profit if we sell on day j
                          # we assume L[j] is the profit if we bought on day j-1 and sold
                          # on day j; i.e., L is the change in value each day, relative to
                          # the day before
            if total > max_sum: max_sum = total

    return max_sum
```

The brute force algorithm works! This will be helpful for testing our code. However, it's slow: `(n-1)` calculations in the first inner loop, then `(n-2)`, and so on, giving us our familiar "sum of the first `k` integers" running time:

```
s = 1 + 2 + 3 + ... + (n-2) + (n-1) = 1/2*(n-1)^2 + 1/2*(n-1) = O(n^2)
```

Deliverable one - `price_to_profit` plus unittests

Notice in the brute force solution that we increment our total profit by `L[j]` on each sell day - we assume each item in the list represents the change in price from the day before. This makes our algorithm more straightforward, but it won't quite work with the provided CSVs, which give the *price* of Bitcoin every day rather then the *change in price* from the day before.

Write a function `price_to_profit` that takes as input the price on a series of days (a list) and returns a list of the change in value each day.

For instance, see the daily price and corresponding profits below:

```
# Day:     0    1    2    3    4
prices = [100, 105, 97,  200, 150]
# change:  +0   +5   -8  +103  -50
profits = [ 0,   5,  -8,  103, -50]
```

`price_to_profit` should work as below, then:

```
>>> profits = price_to_profit( [100, 105, 97, 200, 150] )
>>> print(profits)
[0, 5, -8, 103, -50]
```

Deliverable two - `max_profit` plus unittests

Write a function `max_profit` that uses divide and conquer to find the optimal profit you could have, given a profit-per-day input as implemented above.

```
>>> profits = price_to_profit( [100, 105, 97, 200, 150] )
>>> x = max_profit(profits)
>>> print(x)
103
```

Using the example given, the max profit is trivial - we simply buy at the lowest point (`97`) and sell at the highest (`200`). However, we cannot simply return `max(prices) - min(prices)` - the lowest point may occur after the highest, and we cannot sell our bitcon before we buy it. We have to find the optimal sell date for each buy date individually, which leads us to the `O(n^2)` brute force algorithm above.

Divide-and-Conquer

We can solve this problem much more efficiently using divide-and-conquer:

```
def max_profit(L, left, right):  # O(nlogn)
    # BASE CASE
    #    Only 1 item? Max profit is easy - it's the profit if we bought the day before
    #    and sold today

    # DIVIDE into three problems and CONQUER:
    # max profit if we...
    #    p1: buy and sell in the left (recursive call)
    #    p2: buy and sell in the right (recursive call)
    #    p3: buy in the left and sell in the right (calls another function)
    # COMBINE subproblems into the solution for this level of recursion
    #    Which is better: p1, p2, or p3?
```

Most of your work will be for the third divide case - buying in the left and selling in the right. There is a `O(n)` solution that is similar to a single inner loop in the brute force solution: the max profit for a fixed sell day, then for a fixed buy day.

```
def max_profit_crossing(L, left, right, median):
    # pa, O(n): Linear scan to find max profit if we sell on the median

    # pb, O(n): Linear scan to find max profit if we buy on the median

    # pc, O(1): Max profit if we buy before and sell after

    # Which is better: pa, pb, or pc?
```

The first two cases will take most of the code. The third is trivial after finding them - For example:

```
>>> x = price_to_profit( [50, 45, 107, 105,  200, 250] )
>>> print(x)
[0, -5, 62, -2, 95, 50]
>>> y = max_profit(x)
>>> print(y)
205
```

Our original call to `max_profit()` will ultimately find the following 3 values:

- `p1` - max profit if we buy and sell in the left half (`[0, -5, 62]`): 62 (found recursively)
- `p2` - max profit if we buy and sell in the right half (`[-2, 95, 50]`): 145 (found recursively)
- `p3` - max profit if we buy in the left, sell in the right, `max_profit_crossing()`:
  - `pa` - max profit if we sell on the median (index 2), but bought on:
    - day 1: 62 (**max pa**)
    - day 0: 62-5 = 57
  - `pb` - max profit if we buy on the median (index 2), but sell on:
    - day 3: -2
    - day 4: -2+95 = 93
    - day 5: 93+50 = 143 (**max pb**)
  - `pc` - max profit if we buy before median, but sell after:
    - 62 + 143 = 205 (**max pc**)
  - The best option is `pc`, so that's what we return
- `p1 = 62`, `p2 = 145`, and `p3 = 205`, so the best option is `p3`.

Note that `max_profit_crossing` is not recursive.

This approach requires `O(n)` checks to find the maximum profit at each level of recursion, and we have `O(logn)` levels of recursion (the number of times we can split a list in half), giving us a running time of

`O(nlogn)`.

Submission

At a minimum, submit the following file with the classes/functions noted:

- `TestMaxProfit.py`
  - `class TestPriceToProfit`
    - include at least one non-pdf unittest
  - `class TestMaxProfitCrossing`
    - Test 1 - `pa` is max profit (sell on median)
    - Test 2 - `pb` is max profit (buy on median)
    - Test 3 - `pc` is max profit (buy before median, sell after)
  - `class TestMaxProfit`
    - Test 1 - buy day is minimum price, sell day is maximum price
    - Test 2 - buy and sell days are *not* minimum and maximum prices
    - Test 3 - a bunch of (at least 100) random lists. Use brute force algorithm to verify results
- `max_profit.py`
  - `price_to_profit()`
  - `max_profit()`
  - `max_profit_crossing()`

Students must submit to Gradescope **individually** by the due date (typically Tuesday at 11:59 pm EST) to receive credit.

Grading

This homework is entirely manually graded. You should include your own unittests. You can use examples from this assignment to get started, but will not recieve any credit for them as unittests - create your own.

- 10 - `price_to_profit`
  - 5 - unittests
  - 5 - functionality
- 90 - `max_profit`
  - 45 - unittests
  - 45 - functionality (including nlogn running time)
    - the O(n^2) brute force algorithm is provided for you. You won't recieve any credit for this or similar algorithms; you must use divide-and-conquer instead.

Our manual grading will consider the following:

- Did you structure your code according to best practices in object-oriented programming?
- Did you thoroughly test your code using the unittest module?
- Did you choose and correctly implement the best data structures and algorithms?
- Is the code well-organized and documented? (use docstrings, comments, whitespace, and reasonable naming conventions)

Feedback

If you have any feedback on this assignment, please leave it here.