CSE-2050 – Data Structures and Object-Oriented Design

Fall 2022

Instructor	Office	Contact Details	Office Hours
Hasan Baig	305C	Email: hasan.baig@uconn.edu	See details on Discord
Joseph Steenhuisen (TA)	Virtual *(Contact for in-person meeting)*	Email: joseph.steenhuisen@uconn.edu	See details on Discord

Assignment – 2

Release Date: October 20, 2022

Part-1 due by: **November 03, 2022 (11:59 pm)**
Part-2 due by: **November 10, 2022 (11:59 pm)**

Total points: 350

Points obtained:

Scaled marks (out of 12.5):

Students' Name(s): Andrew Chacon	**Students' ID(s):**

Purpose:
The purpose of this assignment is to help you strengthen your understandings about the topics related to Recursion, searching and sorting algorithms. In addition, programming tasks help you practice implementing these concepts.

Instructions:
1. This assignment should be submitted in a group of up to two students.
2. All theoretical questions should be answered in the space provided and the scanned copies be uploaded on GradeScope under **Assignment-2 – Part-1 – Theory&Concepts** category. You can attach extra sheet if needed.
3. Source files for Programming exercises should be directly uploaded on GradeScope under **Assignment-2 – Part-2 – Task-x** category, where x is the number of task (see below).

Grading Criteria:
1. Your assignments will be checked by instructor/TA followed by a Viva (if needed).
2. During the viva, you will be judged whether you understood the question yourself or not. If you are unable to answer correctly to the question you have attempted right, you may lose your points. No excuses/justifications will be entertained.
3. Zero will be given if the assignment is found to be plagiarized.
4. Untidy work will result in reduction of your points.

Late submission penalty:
- 1-day late submission – 20% deduction of the maximum allowable marks.
- 2-days late submission – 40% deduction of the maximum allowable marks.
- No submission will be accepted after two days of the original deadline.

CLO Assessment:
This assignment assesses students for some portion of the following course learning outcomes.

Course Learning Outcomes		CLO Assessed
CLO 1	Write programs in python using imports, functions, and object-oriented programming.	✓
CLO 2	Compare data structures and algorithms based on time and space complexity and choose the correct ones for a given problem.	✓
CLO 3	Implement abstract data types (stacks, queues, dequeues, mappings, priority queues) using various data structures (lists, linked lists, doubly linked lists, heaps, trees, graphs) and algorithms	✓
CLO 4	Use recursive algorithms to solve problems.	✓

Assignment-2 – Part-1 – Theory & Concepts	[15 + 25 + 10 points]

1) What is the difference between bubble-sort, selection-sort, insertion-sort, merge-sort, quick-sort and quick-select algorithms? Briefly explain their methodologies and indicate their time complexities. Also, mention which algorithm(s) is(are) in-place algorithms. **[2.5 x 6 = 15]**

Bubble Sort
Iterates over every pair in collection, swaps out of order pairs, after x iterations, the last x items are in their final stage place, sorted. It is an inplace algorithm
Best case: O(n), Worst case: O(n^2)

Selection Sort
Iterates over every unsorted item in collection, selects next smallest or largest element, after x iterations, the last x items are in their final place, sorted
This is an inplace algorithm
Best case: O(n^2), Worst case: O(n^2)

Insertion Sort
iterates over progressively growing sorted section of the list, bubbles the next unsorted ithem into place. After x iterations the first x items are sorted but may not be in their final place. This is an inplace algorithm
Best case: O(n), Worst case: O(n^2)

Merge Sort
We keep dividing our data into smaller and smaller subset problems, sort all of the subsets recursively. Then we move up the tree and combine the list until we have merge all the sub arrays into one final solution. This is an inplace algorithm
Best case: O(nlogn), Worst case: O(nlogn)

Quick Sort
Very similar to how we divide and conquer in merge sort however we select an element to become the piviot point, creating 1 sub array. Then creating 2 more sub arrays, one that is values less than piviot and one that is values greater than piviot. We then break those arrays into sub arrays, selecting a new piviot point each time. We then move up the tree merging these sub arrays into one final solution. This is an inplace algorithm.
Best case: O(nlogn), Worst case: O(n^2)

Quick Select
This algorithm is similar to quick sort however when were using recursion we use it to find a specific nth smallest element in an unsorted array. This is an inplace algorithm
Best case: O(n), Worst case: O(n^2)

2) The students of 2050 - Data Structures & Object-Oriented Design course were given an assignment. They started to argue with the course instructor that they have several other assignments due during the same period, so they need an extension. The instructor agrees to extend the deadline ONLY if not a single student is willing to submit on the original deadline. Fortunately, or unfortunately, one of the students agrees to submit the assignment on the given deadline. So, the course instructor informed students about this situation, over the email, and refused their request of deadline extension.

Now, the students start furiously looking for that specific "smart guy" because of whom their "chilled" life is going to be ruined. They want to convince him/her not to submit the assignment on time. All what they know is his/her university ID. They want to determine his/her name and contact details.

Students, somehow, managed to hold on the data record of 100 students who are registered in 2050 course. This data contains student IDs, their first names, and the statuses (whether the students are registered or dropped the course). The glimpse of that data is shown in Figure 1 below.

['2260', 'Joe', 'Registered']	['641', 'Phillip', 'Registered']	['3458', 'Rizwan', 'Registered']	
0	1	2	
['3511', 'Nick', 'Registered']	…..	['3974', 'Aby', 'Registered']	['3588', 'Ryan', 'Registered']
3	…..	98	99

Figure 1. The entire data of students registered in 2050 course. It is the list which contains nested lists having 3 elements in each – ID, Name, Status. In this figure the index of each student data is also shown directly below it.

The students were informed that the ID of that "smart guy" is **3490**. They plan to develop an algorithm to find out who this guy is. They know that they have learnt two search algorithms – linear and binary – which runs on the unsorted and sorted data respectively. They have also studied 4 different sorting algorithms each of which have different pros and cons. The have to adopt an approach which gives them the required information in a minimal time. They are also out of additional memory, so they have to use such an approach which avoids using auxiliary memory.

Answer the following questions:

(a) Is the data given in Figure 1 sorted? **[1]**

This data is not sorted at all since element 0 with a value of 2260 is greater than element 1 which as a value of 641. Element 2 has a value of 3458 is there is no order in the ID's

(b) Considering that the data is large, what should be the next step – searching or sorting? Explain briefly. **[3]**

Our next step should be to use some kind of linear sort in order to sort our data, since the way our data is currently formated, there is no order

(c) If your answer to part (b) is sorting, which sorting algorithm would you implement and why? **[6]**

The type of sorting algorithm i would use is merge sort, my reason for choosing it is that it works for data of large sets, which is currently what we have as our data for this problem

(d) If your answer to part (b) is searching, which search algorithm will you use and why? **[2]**

(e) Suppose, after running part (c) and part (d) (to some extent), the search is narrowed down to the highlighted portion of data as shown below in Figure 2:

		….	[‘3471’, ‘Emily’, ‘Registered’]	[‘3473’, ‘Jordan’, ‘Dropped’]	[‘3475’, ‘Omar’, ‘Registered’]
0	1	….	50	51	52

[‘3479’, ‘Chen’, ‘Registered’]	[‘3481’, ‘Cole’, ‘Registered’]	[‘3482’, ‘Ariel’, ‘Registered’]
53	54	55

[‘3487’, ‘John’, ‘Registered’]	[‘3490’, ‘Tedi’, ‘Dropped’]	[‘3495’, ‘Jakob’, ‘Registered’]
56	57	58

[‘3503’, ‘Amari’, ‘Registered’]	[‘3505’, ‘Jan’, ‘Registered’]	[‘3508’, ‘Pedro’, ‘Registered’]
59	60	61

…..	99

Figure 2. The search range of a data narrowed down to 12 elements. In this figure the index of each student data is also shown directly below it.

(i) In the highlighted data shown above, which iteration (starting from 1) of a loop will give the required element? **[12]**

The fourth iteration will give us our required element, 57

(ii) What will be the low, high and middle index values when the required element is found? **[1]**

The low, high, and middle index values would be found at [57, 57, 57]

After getting the required information, you realized that the “smart guy”, who agreed to submit the assignment on the original due date, has already dropped out this course.

Lesson learnt: Struggle to work on your assignment instead of struggling to extend the deadline ☺.

3) Time complexities:

a. What will be the complexity of quick sort algorithm for the following data? The pseudocode of the algorithm has been provided for help. Explain your answer. **[5]**

Array:

2	2	2	2	2	2

```
quickSort(Array, L-Idx, H-Idx)
    if L-Idx < H-Idx

        pivot = partition(Array, L-Idx, H-Idx)
        quickSort(Array, L-Idx, pivot-1)
        quickSort(Array, pivot+1, H-Idx)

end
```

```
partition(Array, low, high)
    pivotindex = (low+high)/2
    swap(pivotindex, high)

    i = low

    for j = low to high
        if Array[j] <= Array[high]
            swap(i, j)
            i++
    return i-1

end
```

Since all of the values in the data are the same, the condition for swapping will swap all of the values since it is checking for <= values

This would be the worst case scenario for algorithm, resulting in a time complexity of O(n^2)

b. For the given data, what will be the time complexity to run the code shown as pseudocode below: **[5]**

The Data set given is :

Data =

a	b	c	d	e	f	g

```
for i = 1 to Data.length
    j = i
    while j>0 AND Data[j-1] > Data[j]
        swap(Data, j, j-1)
        j = j -1
end
```

What will be the time complexity of running the above code on the given Data?

Since only our for loop would be running to loop through the list, the time complexity for the code is O(n)

Assignment-2 – Part-2 – Programming Exercises	[3 x 100 points]

Task-1

Use recursion to determine if a knight starting in a given square on a chessboard can capture all pawns on that board using only moves that capture a pawn (moves to spaces without a pawn, or spaces with pawns that have already been captured, are not allowed).

Consider an 8x8 chessboard with a knight and some pawns:

```
#  0 1 2 3 4 5 6 7
#0 - - - - - - - - Row 0: (0, 0), (0, 1), (0, 2), ... (0, 7)
#1 - - - p - - - -
#2 - p - - - p - -
#3 - - - k - - - -
#4 - - p - - - - -
#5 - - - - - p - -
#6 - - - p - - - -
#7 - - - - - - - -Row 7: (7, 0), (7, 1), (7, 2), ... (7, 7)
k_idx = (3, 3) # Initial knight index
p_idxs = {(1, 3), (2, 1), (2, 5), (4, 2), (5, 5), (6, 3)} # Set of pawn indices
```

This board is solvable. Our knight can capture all 6 pawns with 6 consecutive moves:

```
(2, 5), (1, 3), (2, 1), (4, 2), (6, 3), (5, 5)
```

However, the following board is not solveable:

```
# 01234567
#0 - - - - - - - - Row 0: (0, 0), (0, 1), (0, 2), ... (0, 7)
#1 - - - - - - - -
#2 - p - - - p - -
#3 - - - k - - - -
#4 - - p - - - - -
#5 - - - - - p - -
#6 - - - p - - - -
#7 - - - - - - - -Row 7: (7, 0), (7, 1), (7, 2), ... (7, 7)
k_idx = (3, 3) # Initial knight index
p_idxs = {(2, 1), (2, 5), (4, 2), (5, 5), (6, 3)} # Set of pawn indices
```

There is no way for our knight to capture all 5 pawns in just 5 moves.

a) Write Unittests

Use TDD on this assignment- write a test, run it, then implement functionality. You will be graded on your tests. Starter code for **TestHw2a.py** includes some guidance on tests. Do not use the examples above as boards in your tests.

There are no (visible) autograder tests on this assignment. We have a few hidden tests to help speed up manual grading, but they will not directly impact your grade.

b) Implement valid_moves(k_idx)

valid_moves(k_idx) returns a set of tuples representing all valid moves for a knight at **k_idx.**

c) Implement solveable(p_idxs, k_idx):

- **p_idxs** - a set of tuples representing pawn locations.
- **k_idx** - a tuple representing the starting position of a knight.
- Return a boolean denoting whether the passed in **p_idxs** and **k_idx** represent a solveable board.

Submission

At a minimum, submit the following files:

- TestHw2a.py
- hw2a.py

Submit on **GradeScope under "Assignment2 – Part2 – Task-1"** category.

Grading:

100% manually graded.

Task-2

Many of the problems used to teach recursion are not efficiently solved with recursion - summing numbers, finding Fibonacci's, and calculating factorials are all problems that have intuitive but inefficient recursive solutions.

Here, we look at a problem type where recursive solutions tend to be genuinely useful: exploring branching paths until we hit a dead end or find a solution. We'll see this a lot in graphs later this semester, and, in general, this is a useful first step for modelling most games.

The basic problem abstraction is this:

- We have multiple options of what to do at each step
- We want to see if *any* series of steps exists that connect a starting point to a valid solution

Stepping Game

We will model a circular board game. The board consists of a series of tiles with numbers on them.

- The number on a tile represents the number of tiles you can move from there, forwards (clockwise) or backwards (counter-clockwise).
- It's okay to "circle around" - moves that go before the first tile or after the last are valid.
- The goal is to reach the final tile (the tile 1 counter-clockwise from the start). This is the only valid solution - **finding a tile with "0" is not necessarily a valid solution, since non-final tiles may contain 0.**

  Not all boards will be solvable. See below for a solvable (Figure 2) and an unsolvable (Figure 3) example.

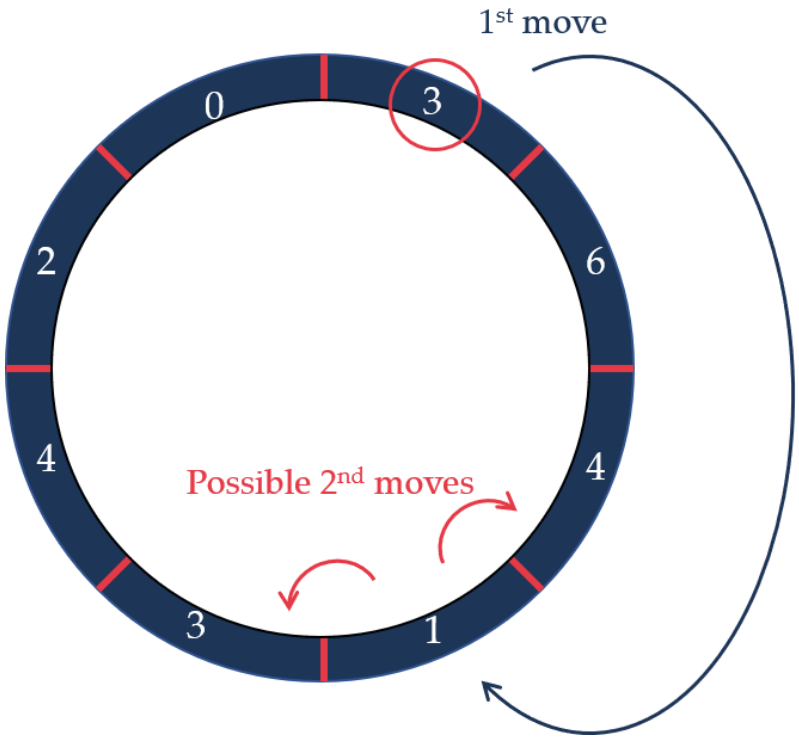

Figure 1: Solvable game: [3, 6, 4, 1, 3, 4, 2, 0].

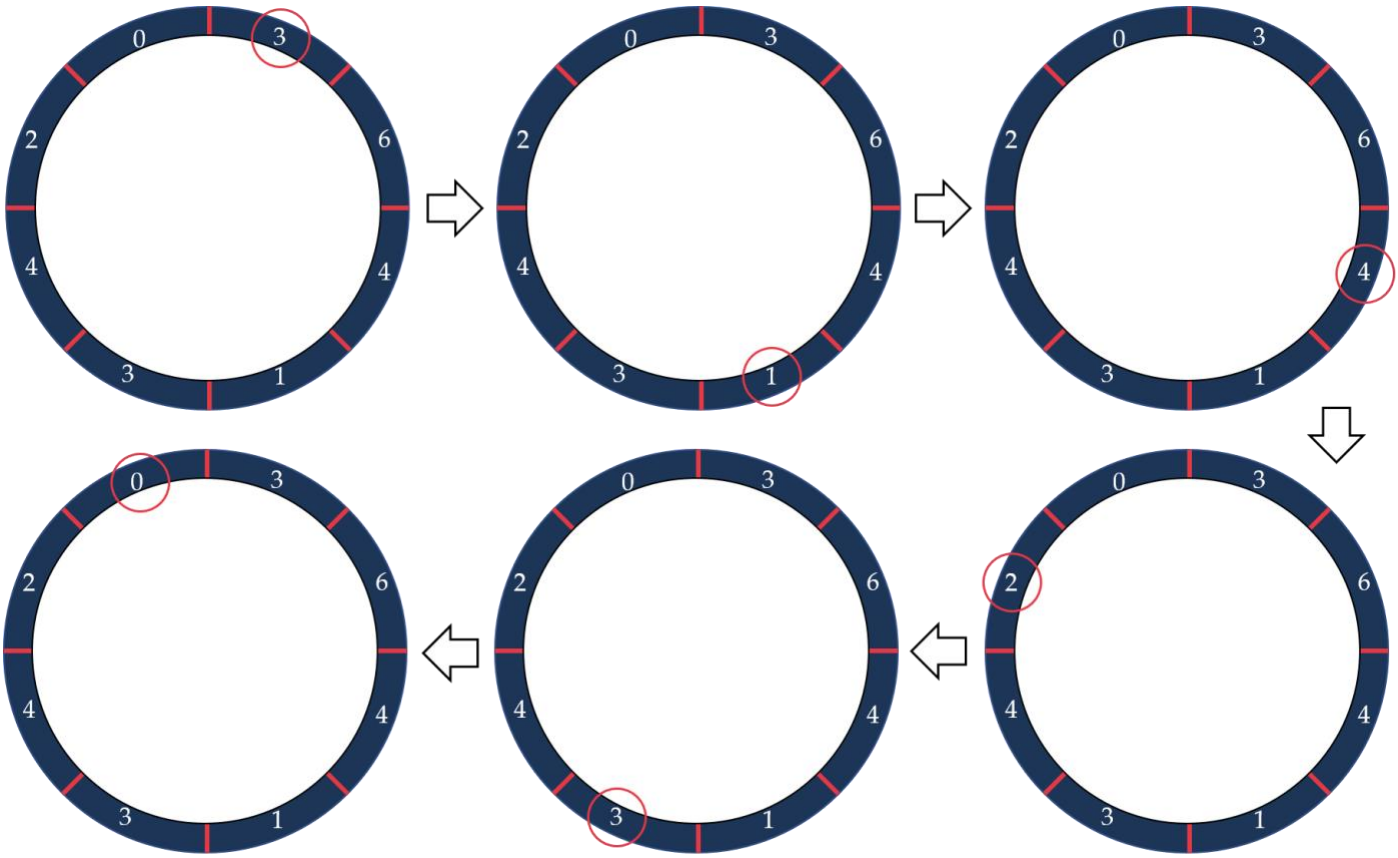

Figure 2: Step by step solutions to the above game.

Deliverables

- **`TestSolvePuzzle.py`**

First, write tests. This helps solidify the rules of the game in your mind and ensures you will know when you get a working algorithm.

- Test at least 4 puzzles:
  - solvable using only clockwise moves
    - make sure counter-clockwise moves do not produce a valid solution here
  - solvable using only counter-clockwise moves
    - make sure clockwise moves do not produce a valid solution here
  - requires a mixture of clockwise and counter-clockwise moves to solve
    - make sure a purely clockwise or purely counter-clockwise step-sequence do not produce a valid solution here
  - unsolvable
- Use the **`Unittest`** package
- Keep the boards for these tests relatively small (<= 5 spaces). This makes debugging easier for you and grading easier for us.
- Do not use any of the boards in this assignment as your custom tests - write your own.

- **`solve_puzzle.py`**
  - contains a function **`solve_puzzle():`**
    - Input: a list representing the board. The first item in the list is the starting position, with subsequent items denoting the board in a clockwise fashion. See the figures above for an example of list representations of boards.
    - Output: A Boolean (True or False) denoting if the puzzle is solvable.

Tips:

You will need Memoization to avoid infinite loops. If you use a helper function, avoid using an empty mutable collection (like an empty set) as a default value.

```
def solve_puzzle(L, visited=None): # No helper function
    if visited is None: # initialize
    # the rest of your work here
```

```
def solve_puzzle(L): # Using a helper function
      visited = set()
      return _solve_puzzle(L, visited)


def _solve_puzzle(L, visited):
    # the rest of your work here
```

You can assume the numbers on tiles are non-negative integers (0 is valid).

Submission:

At a minimum, submit the following files:

- **`solve_puzzle.py`**
- **`test_solve_puzzle.py`**

Automatic tests are only worth 10 points to ease frustration if you cannot pass them - you can still get partial credit when we manually grade the assignment.

Conversely, passing the tests does not mean your algorithms are fully functional - be thorough in your own test cases to verify your code works to ensure you receive full credit on the manually graded portion.

Submit on **GradeScope under "Assignment2 – Part2 – Task-2"** category.

Grading:

- 50 - **`solve_puzzle.py`** –
  - 10 - auto graded
  - 40 - manually graded

- 50 - **`TestSolvePuzzle.py`** (manually graded)

Task-3

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

- binary search
- merge sort
- quick sort

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 the price of Bitcoin at market opening for each year from 2015-2020 in csv files. Our goal is to figure out the maximum profit we could have made in each year. 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 is provided, but it won't quite work with the CSVs you have been provided. The CSVs give the price of Bitcoin every day, but the brute source algorithm expects a list of *change in value* for each day, relative to the day before. For instance, if the price over one week was:

```
[100, 105, 97, 200, 150]
```

The change in value each day would be

```
[0, 5, -8, 103, -50]
```

Deliverable one – **`price_to_profit`**

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.

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

Deliverable two – `max_profit`

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

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

Using the example given, the max profit is trivial - we simply buy at the lowest point and sell at the highest. In general, the lowest point may not come until after the highest, so we need a cleverer algorithm than return **`max(L) - min(L)`.**

Brute Force

A brute force algorithm that solves this problem is provided to help you test your function - we find the maximum profit by calculating every possible sell date for every possible buy date.

```
def max_profit_brute(L):
    """Finds maximum profit. Assumes L is a list of profits (i.e. change in price
every day), not raw prices"""
    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 and sold on day j
        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
```

This has a running time of O(n^2):

Divide-and-Conquer

We can solve this problem much more efficiently using divide-and-conquer. If we cut our list in half, the maximum profit comes from a buy-sell pair that is either:

- buy and sell date in the left half
- buy and sell date in the right half
- buy date in the left half, sell date in the right

Some pseudo code to help get started:

```
def max_profit(L, left, right): # O(nlogn)
    # base case? Return today's profit
    # find the max profit in the left-hand sublist
    # find the max profit in the right-hand sublist
    # find the max profit that crosses from left to right (requires a separate
function)

# return the best profit - left, right, or crossing

def max_profit_crossing(L, left, right, median): # O(n)
    # Starting from the median, find the best price moving left
    # starting from just after the median, find the best price moving right
    # return the best profit:
    #     * left of median (inclusive)
    #     * right of median (exclusive)
    #     * buy on the left, sell on the right (does not require a recursive call)
```

Using this approach, we still require O(n) checks to find the maximum profit at each level of recursion, but the number of levels of recursion reduces from O(n) to O(logn).

Submission:

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

- **max_profit.py**
  - **price_to_profit()**
  - **max_profit()**

Submit on **GradeScope under "Assignment2 – Part2 – Task-3"** category.

Grading:

- **10 - price_to_profit (required for some other tests to work)**
- **90 - max_profit**
  - 10 - works on small list
  - 40 - works on 6 years of bitcoin data
  - 40 - works on huge lists (requires O(nlogn) solution)