Module 2 Workbook

This week, we're going to develop your coding interview strategy. The exercises in this workbook are designed to help you practice each step of the process and reinforce the skills.

It's important to realize that some of this may feel uncomfortable at first. You're learning a brand new skill that is likely different from how you've approached interviewing in the past.

Therefore, it's really important that you just follow along and do the best you can. If you're getting frustrated, just take a few minutes away and come back to the exercises later. As you get more practice, they will get easier and you'll be glad that you put in the effort.

Throughout this workbook, we will be referring back to the steps of your problem solving strategy. For reference, here are the steps:

1.	Understand the problem	[3-5 minutes]
2.	Find a brute force solution	[5 minutes]
3.	Optimize your solution	[15 minutes]
4.	Code your solution	[15 minutes]
5.	Test your solution	[5 minutes]

Table of Contents

Day 1

If you haven't already, make sure that you've watched the videos for this week. They will give you all of the context you need to get the most out of these exercises.

Today we will focus on Step 1 of our problem solving strategy: Understanding the problem.

For today's exercises, do the following for each problem:

1. Write down your assumptions about the problem. What, if anything, are you assuming about the input or output? Were you given any specific information (ie. the input is sorted)?
2. Work through the given examples. Verify how these inputs actually lead to the desired output.
3. Write at least 2 sample inputs of your own and test them. Try to write tests that will hit on potential edge cases.
4. Write the function signature. Be sure to define the type of the input and the output. If you're using a dynamically typed language, you should still be clear on the expected input and output type.

NOTE: Make sure you write down your answers somewhere. We are going to refer back to these problems and complete the rest of the problem later this week.

Problems:

1. Maximum Sum Subarray (Leetcode)
2. Number of Islands (Leetcode)
3. Climbing Stairs (Leetcode)
4. Merge Intervals (Leetcode)
5. Maximum Binary Tree Depth (Leetcode)

Day 2

We're going to continue from what we were working on yesterday.

NOTE: It may feel like we're jumping around a lot but there are 2 specific reasons for this. 1) It makes sure that you're really learning each individual skill. 2) It is important that you learn to clearly articulate yourself. By having to remember from day to day what you've been working on, you improve your ability to be clear.

Part 1:

Part 1 we're going to do exactly what we did yesterday. Do the following for each problem:

1. Write down your assumptions about the problem. What, if anything, are you assuming about the input or output? Were you given any specific information (ie. the input is sorted)?
2. Work through the given examples. Verify how these inputs actually lead to the desired output.
3. Write at least 2 sample inputs of your own and test them. Try to write tests that will hit on potential edge cases.
4. Write the function signature. Be sure to define the type of the input and the output. If you're using a dynamically typed language, you should still be clear on the expected input and output type.

Problems:

1. Linked List Cycles (Leetcode)
2. Buy and Sell Stock (Leetcode)

Day 2, Part 2 continues on the next page.

Part 2:

Once you've done the first part, we're going to start coming up with brute force solutions to some of the problems that you did yesterday.

For each of the problems below, come up with a basic brute force solution. There's no need to code it up fully, but write enough pseudocode that you will be able to refer back to it and remember what you were doing.

As a reminder, here are some of the things we talked about in class when finding a brute force solution:

- Just find the dumbest/most obvious solution
- Ignore everything you might already know about the problem
- Some strategies:
    - Can you enumerate all the possibilities?
    - How would you solve the problem by hand?
    - Can you solve a similar/simpler problem?
    - Can you solve the problem in multiple stages?

Find brute force solutions for the following problems:

1. Maximum Sum Subarray (Leetcode)
2. Number of Islands (Leetcode)

Day 3

Part 1:

For each of the problems below, come up with a basic brute force solution. There's no need to code it up fully, but write enough pseudocode that you will be able to refer back to it and remember what you were doing.

As a reminder, here are some of the things we talked about in class when finding a brute force solution:

- Just find the dumbest/most obvious solution
- Ignore everything you might already know about the problem
- Some strategies:
    - Can you enumerate all the possibilities?
    - How would you solve the problem by hand?
    - Can you solve a similar/simpler problem?
    - Can you solve the problem in multiple stages?

Find brute force solutions for the following problems:

1. Climbing Stairs (Leetcode)
2. Merge Intervals (Leetcode)

Part 2:
Today we're going to start working on Step 3, optimizing our solutions. This part is going to start to get more time consuming so we'll start with just one problem.

For this problem, consider how we can optimize our brute force solution. You should complete the following steps:

1. Identify the time and space complexity
2. Identify the Best Conceivable Runtime and Space
3. Brainstorm and experiment with different ways to optimize the solution (I recommend you refer back to the slides)
4. Write up a plain-English description of your solution. This should be detailed enough that you can read through it once and immediately know how to code up the solution. We will be doing this in a few days.

Complete the above for Maximum Sum Subarray (Leetcode).

Day 4

Part 1:

For each of the problems below, come up with a basic brute force solution. There's no need to code it up fully, but write enough pseudocode that you will be able to refer back to it and remember what you were doing.

As a reminder, here are some of the things we talked about in class when finding a brute force solution:

- Just find the dumbest/most obvious solution
- Ignore everything you might already know about the problem
- Some strategies:
    - Can you enumerate all the possibilities?
    - How would you solve the problem by hand?
    - Can you solve a similar/simpler problem?
    - Can you solve the problem in multiple stages?

Find brute force solutions for the following problems:

1. Maximum Binary Tree Depth (Leetcode)
2. Linked List Cycles (Leetcode)

Day 4 Part 2 continues on the next page.

Part 2:

Today we're going to start working on Step 3, optimizing our solutions. This part is going to start to get more time consuming so we'll start with just one problem.

For this problem, consider how we can optimize our brute force solution. You should complete the following steps:

1. Identify the time and space complexity
2. Identify the Best Conceivable Runtime and Space
3. Brainstorm and experiment with different ways to optimize the solution (I recommend you refer back to the slides)
4. Write up a plain-English description of your solution. This should be detailed enough that you can read through it once and immediately know how to code up the solution. We will be doing this in a few days.

Complete the above for the following problems:

1. Number of Islands (Leetcode)
2. Climbing Stairs (Leetcode)

Day 5

Part 1:

For each of the problems below, come up with a basic brute force solution. There's no need to code it up fully, but write enough pseudocode that you will be able to refer back to it and remember what you were doing.

As a reminder, here are some of the things we talked about in class when finding a brute force solution:

- Just find the dumbest/most obvious solution
- Ignore everything you might already know about the problem
- Some strategies:
  - Can you enumerate all the possibilities?
  - How would you solve the problem by hand?
  - Can you solve a similar/simpler problem?
  - Can you solve the problem in multiple stages?

Find brute force solutions for the following problems:

1. Buy and Sell Stock (Leetcode)

Day 5 Part 2 continues on the next page.

Part 2:

Today we're going to start working on Step 3, optimizing our solutions. This part is going to start to get more time consuming so we'll start with just one problem.

For this problem, consider how we can optimize our brute force solution. You should complete the following steps:

1. Identify the time and space complexity
2. Identify the Best Conceivable Runtime and Space
3. Brainstorm and experiment with different ways to optimize the solution (I recommend you refer back to the slides)
4. Write up a plain-English description of your solution. This should be detailed enough that you can read through it once and immediately know how to code up the solution. We will be doing this in a few days.

Complete the above for the following problems:

1. Maximum Binary Tree Depth (Leetcode)
2. Linked List Cycles (Leetcode)

Day 6

Part 1:

Today we're going to start working on Step 3, optimizing our solutions. This part is going to start to get more time consuming so we'll start with just one problem.

For this problem, consider how we can optimize our brute force solution. You should complete the following steps:

1. Identify the time and space complexity
2. Identify the Best Conceivable Runtime and Space
3. Brainstorm and experiment with different ways to optimize the solution (I recommend you refer back to the slides)
4. Write up a plain-English description of your solution. This should be detailed enough that you can read through it once and immediately know how to code up the solution. We will be doing this in a few days.

Complete the above for the following problems:

1. Merge Intervals (Leetcode)
2. Buy and Sell Stock (Leetcode)

Day 6 Part 2 continues on the next page.

Part 2:

Yay time for some actual coding. Be very aware of how long it takes for you to code up each one of these problems. If you did the previous step correctly it shouldn't take you more than 15 minutes. If not, then you'll want to focus on better understanding the problems in the future.

For these problems, do the following:

- Code up the entire solution by hand. DO NOT look anything up. If you forget the function definition, just put a placeholder or best guess.
- Once you think your code is correct, test it by hand.
- If it works, then and only then, copy it verbatim into an IDE or Leetcode and try to run your code.
- Make a list of all the errors in your code to refer back to later.

Problems:

1. Maximum Sum Subarray (Leetcode)
2. Number of Islands (Leetcode)

Day 7

Rest day!

We've done a lot this week so I didn't schedule anything for you on this last day. Use this time to take a little break or catch up on anything you may have gotten behind on :)

Workbook Solutions

Day 1

1. Maximum Sum Subarray
    a. Assumptions:
        i. The array has to be contiguous
        ii. Can contain both positive and negative values
        iii. Minimum array length is `0`
        iv. Maximum array length is `n`
        v. The result will never be `< 0`
        (because if all the values are negative we find the empty array)
        vi. Just need to find the sum itself
    b. Function Definition: `int maxSumSubarray(int[] arr)`

2. Number of Islands
    a. Assumptions:
        i. Minimum number of islands is `0`
        ii. Maximum number is n^2
        (probably less because then they would be touching making one contiguous island)
        iii. Islands can be any size
        iv. 2 islands can't be touching each other
    b. Function Definition: `int numIslands(int[][] arr)`

3. Climbing Stairs
    a. Assumptions:
        i. The minimum staircase size must be `0`, in which case the result is `1`
        ii. The order of steps matters here
        iii. Just find the number of paths
    b. Function Definition: `int stairs(int n)`

4. Merge Intervals
    a. Assumptions:
        i. We could have multiple overlapping intervals
        ii. The max interval range is the maximum range of the individual intervals
        iii. Merging intervals is associative, so it doesn't matter what order we merge them in
        iv. The input is in no particular order
        v. We will end up with at most `n` intervals, if `n` is the number of intervals we started with
    b. Function Definition:
    `List<List<Integer>> mergeIntervals(int[] intervals)`

5. Maximum Binary Tree Depth
    a. Assumptions:
        i. The max depth is the number of nodes
        ii. The min depth is `log(number of nodes)`
        iii. Input is the root node of the tree
        iv. Can probably do some sort of DFS or BFS
        v. Just need to find the length of the longest path
    b. Function Definition: `int maxDepth(TreeNode root)`

Day 2

Part 1:

1. Linked List Cycles
    a. Assumptions:
        i. There may or may not be a cycle at all
        ii. Multiple nodes can have the same value so we should compare objects and not values
        iii. Single-directional linked list
    b. Function Definition: `boolean hasCycle(ListNode head)`

2. Buy and Sell Stock
    a. Assumptions:
        i. Only one transaction per day
        ii. A buy transaction has to be followed by a sell transaction - can we group them?
        iii. Only one transaction -> want to find the biggest difference
    b. Function Definition: `int maxProfit(int[] prices)`

Part 2:

1. Maximum Sum Subarray
    a. Brute force solution: The easiest approach here is just to compare all of the different subarrays and find the one that has the maximum sum. Remember that often our approach can just involve enumerating all the possibilities, particularly when a problem asks for the best solution.
2. Number of Islands
    a. Brute force solution: We can do a variant of either BFS or DFS here. It doesn't really matter which one we choose. We can just start at each different position on the map and expand out as much as we can from there to find a single island. Then we repeat this for all of the unvisited areas of the map.

Day 3

Part 1:

1. Climbing Stairs
    a. Brute force solution: We want to find all the different ways to climb the staircase, so let's just enumerate all of the different ways to find the staircase. The easiest way to do this is going to be recursively.
2. Merge Intervals
    a. Brute force solution: The easiest brute force solution here is to just find the maximum range of the intervals and for each time slice check whether or not it is in any of the intervals. Basically, we can just build a boolean array the size of the maximum range and then iterate through each interval and mark the range in our array.

Part 2:

1. Maximum Sum Subarray
    a. Time Complexity
        i. There are n^2 different subarrays
        ii. It takes `O(n)` time to compute the sum
        iii. n^2 `* O(n) => O(`n^3`)`
    b. Space Complexity
        i. We don't require any extra space
        ii. `O(1)`
    c. Best Conceivable Runtime
        i. At minimum we have to visit all of the values in our array
        ii. Our array is length `n`
        iii. `O(n)`
    d. Best Conceivable Space
        i. We are already `O(1)`
    e. Optimization approaches
        i. Can we do this without looking at all the subarrays?
        ii. Is there a way we can avoid visiting the same value multiple times?
        iii. Can we avoid having to recompute the sum every time?
        iv. Could we use a running sum?
        v. Could we use a sliding window?

f. Plain-English solution
    i. Let's just compute a running sum from the beginning to the end of our array. We will track the value at each index
    ii. Then all we need to do is one of the following:
        1. As long as the running sum is positive, keep adding to the running sum
        2. If the running sum is negative, then reset the sum to `0`
    iii. Finally, iterate over all of our sums and return the largest value
    iv. *Note:* We could also track the max value as we go, although that doesn't affect our runtime
    v. *Explanation:* This is easiest illustrated with the 3 possible cases:

        1. As we compute the running sum it keeps increasing
        `arr: [1, 2, 3, 4, 5]`
        `sum: [1, 3, 6, 10, 15]`
        `subarraySum: 15`
        Since all the numbers are positive and increase the sum, we want to include all of them in our result

        2. As we compute the running sum, it goes up and down
        `arr: [5, -4, 3, -2, 5]`
        `sum: [5, 1, 4, 2, 7]`
        `subarraySum: 7`
        As long as the running sum is positive for the first part of the array, we want to add it to the rest, because it will increase the sum of the subarray. Look at any other subarray and you can see why the result would be less than `7`.

        3. As we compute the running sum it goes negative
        `arr: [1, -2, 3, 4, -5]`
        `sum: [1, -1, 3, 7, 2]`
        `subarraySum: 7`
        Notice here that after index `1`, we reset the running sum to `0` (hence the running sum at index `2` is `3`, not `-1 + 3`). That's because we will never be able to maximize our subarray if we are adding a negative value to the beginning.

Day 4

Part 1:

1. Maximum Binary Tree Depth
    a. Brute force solution: This is another DFS problem. If we just enumerate all the paths in the tree, we can select whichever is the longest one.
2. Linked List Cycles
    a. Brute force solution: The easiest solution here is to track all the nodes we've seen so far. We can add them all to a set. Then we'll just check each time we visit a node whether or not we've seen it before. The question is whether we can do this without extra space.

Part 2:

1. Number of Islands
    a. Time Complexity
        i. Our search will visit every cell in our grid at least once and no more than 4 times (each cell has 4 neighbors)
        ii. If we have an `nxm` grid, we have `n*m` cells
        iii. `O(4*n*m) => O(n*m)`
    b. Space Complexity
        i. Let's assume we do DFS recursively
        ii. The maximum depth of our recursion is `n*m`
        iii. We have to store a stack frame for each recursive call
        iv. `O(n*m)`
    c. Best Conceivable Runtime
        i. We're going to have to visit all the cells
        ii. `O(n*m)`
    d. Best Conceivable Space
        i. Maybe we could improve here, but we are going to need to at minimum track where we have visited already
        ii. If we can modify the input, we can use the pre-allocated space to track, in which case we may be able to do `O(1)`
        iii. Otherwise we're stuck with `O(n*m)`
    e. Optimization approaches
        i. Based on the Best Conceivable Runtime, we can't do this any faster
        ii. Can we optimize for space? Let's assume we can't modify the input
        iii. There's not really anything we can do

   f. Plain-English solution
      i. Iterate over the entire grid
      ii. If a cell is marked as land, use DFS to find all the connected land. Then mark all those cells as visited
      iii. Continue this process, incrementing each time you find a new piece of land

2. Climbing Stairs
   a. Time Complexity
      i. There are 2^n different ways to combine all of the stairs
      ii. We have to find all of these and then filter the ones that are valid
      iii. Since we're creating an array for each combination, that takes an additional `O(n)` time to allocate/set the values
      iv. $O(2^n * n) => O(2^{n+1})$
   b. Space Complexity
      i. We are saving all of the possible combinations in memory before sorting them
      ii. We have 2^n combinations each of length `O(n)`
      iii. $O(2^{n+1})$
   c. Best Conceivable Runtime
      i. At minimum we need to iterate over all of the steps
      ii. `O(n)`
   d. Best Conceivable Space
      i. There is no inherent need for us to store any particular amount of information
   e. Optimization approaches
      i. Immediately because this is recursive and doing a lot of repetitive computations, I think dynamic programming
      ii. Can we avoid recomputing lots of subproblems?
      iii. Can we avoid storing all these combinations to memory?
      iv. Can we avoid computing all combinations and only compute valid combinations?
      v. Can we avoid computing the combinations at all?
   f. Plain-English solution
      i. We can use DP for this
      ii. We will allocate an array of length `n`, where `arr[i]` represents the number of combinations for the first `i` steps
      iii. As we iterate over the array, each `arr[i]` is equal to the sum of the 3 previous. Ie. `arr[i] = arr[i-1] + arr[i-2] + arr[i-3]`
      iv. See the FAST Method for more info on dynamic programming

Day 5

Part 1:

1. Buy and Sell Stock
    a. Brute force solution: We're only able to buy and sell once, so let's just look at every possible buy-sell combination.

Part 2:

1. Maximum Binary Tree Depth
    a. Time Complexity
        i. We are doing a search in our tree
        ii. We have to consider all possible paths, and therefore we have to visit every node
        iii. `O(n)`
    b. Space Complexity
        i. We are searching recursively and the maximum depth of our recursion is n
        ii. We aren't using any other space
        iii. `O(n)`
    c. Best Conceivable Runtime
        i. We're going to have to visit all the nodes in the worst case so there isn't any improvement to be had here
        ii. `O(n)`
    d. Best Conceivable Space
        i. If we are doing DFS or BFS, we are going to need to use extra space
        ii. `O(n)`
    e. Optimization approaches
        i. In this case, there's not really any optimization we can do
    f. Plain-English solution
        i. We will use DFS to traverse each path in the tree
        ii. As we traverse, we will count the length of the path and keep track of the longest path we have seen
        iii. Finally, we just return the length of the longest path

2. Linked List Cycles
    a. Time Complexity
        i. Our solution requires us to iterate over every node once
        ii. All the other operations we're doing are constant-time operations
        iii. `O(n)`
    b. Space Complexity
        i. We have to store all of the nodes so that we can compare them
        ii. `O(n)`
    c. Best Conceivable Runtime
        i. We're always going to have to visit all the nodes in our list
        ii. `O(n)`
    d. Best Conceivable Space
        i. Seems plausible that we could do this without using extra space
        ii. `O(1)`
    e. Optimization approaches
        i. There's no room to optimize the time complexity, but can we optimize the space?
        ii. If we modify the nodes, we can simply mark them as visited
        iii. If we use 2 pointers and move them at different speeds, they will eventually collide if there's a cycle
    f. Plain-English solution
        i. We initialize 2 pointers, `fast` and `slow`
        ii. We loop, moving fast forward by 2 and slow forward by 1 each turn
        iii. If `fast` reaches the end, there's no loop
        iv. If `fast` and `slow` are equal, then there is a loop
        v. *Note:* This is one of those problems that has a "trick" to it. It is good to know this trick but if you didn't come up with it on your own, it is fine. It is likely that an interviewer would accept our brute force solution
        vi. *Explanation:* Assuming there is a loop, you will eventually end up in one of the following states:

```
1. 1 -> 2 -> 3 -> 4 -> 5
   ^    ^
   fast slow

2. 1 -> 2 -> 3 -> 4 -> 5
   ^         ^
   fast      slow
```

In the first case, it is clear that the pointers will point to the same node in the next step. In the second case, by taking one step forward, you end up with case #1.

Day 6

Part 1:

1. Merge Intervals
    a. Time Complexity
        i. We are computing every possible time slice
        ii. At each time slice, we have to look at all of our intervals
        iii. Take `T` as the total range of times of all the interviews
        iv. Take `n` as the number of intervals
        v. `O(T * n)`
    b. Space Complexity
        i. We are creating a boolean array of length `T`
        ii. `O(T)`
    c. Best Conceivable Runtime
        i. We have to iterate over the input itself at the very least
        ii. `O(n)`
    d. Best Conceivable Space
        i. There is no necessary reason for us to use any space
        ii. We don't count the space allocated for the result of our function
        iii. `O(1)`
    e. Optimization approaches
        i. We don't really need to look at all the time slices. We can just look at the start and end of each interval
        ii. Would it help us to sort the intervals somehow?
        iii. What does it actually mean for 2 intervals to be overlapping? The start of the one interval is between the start and end of the other interval
        iv. Do you have to merge the intervals in any particular order?
    f. Plain-English solution
        i. First, let's sort the intervals by start time
        ii. Now iterate over all of the intervals and merge them together
            1. Take the first interval and see if it overlaps with the second interval
            2. If so, see if that combined interval overlaps with the third interval
            3. When you find that the next interval doesn't overlap, save the previous merged interval to your result array

2. Buy and Sell Stock
    a. Time Complexity
        i. There are n^2 combinations of buy and sell
        ii. It takes `O(1)` time to compute the value of each transaction
        iii. `O(n²)`
    b. Space Complexity
        i. We aren't using any extra space
        ii. `O(1)`
    c. Best Conceivable Runtime
        i. We do have to look at the price at every time
        ii. `O(n)`
    d. Best Conceivable Space
        i. We are already doing this with `O(1)` space
    e. Optimization approaches
        i. Do we need to look at every combination here? No we only need to look at local minima and maxima
        ii. We're only making 1 transaction, so we really just need to find the biggest range between 2 prices
    f. Plain-English solution
        i. Let's iterate through our array
        ii. We will keep track of the smallest value we have seen so far
        iii. For each value that follows, we will compute the difference between that and the smallest value
        iv. We then return the largest difference that we find

Part 2:

Copy your code verbatim into Leetcode and test.