Module 3 Workbook

Last week we learned the general approach we will be taking to solve coding interview questions. Hopefully you made some good progress!

This framework is incredibly valuable for approaching these interview questions. Having a step by step process makes all the difference.

However, we do want to make sure that not only are we able to come up with working solutions to our problems but that we are able to optimize them as well.

As you saw in this weeks videos, we went through a whole bunch of strategies that you can use to optimize your code. Throughout the exercises this week, you will have a chance to practice applying these strategies.

Table of Contents

Day 1

As we get into the exercises this week, there are still a few problems from last week that we didn't completely finish. I want to make sure that you don't get shortchanged on those either so they are mixed in here.

Today we're going to continue with what we were working on last week.

Part 1:

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

Part 2:

If you haven't already, make sure to watch the videos for this week.

Day 2

Today we're going to practice applying some optimization strategies. We will apply it to the problems that we've already done and then for the rest of this week, we will practice applying the entire interview process to some new problems.

For each of these problems, I want you to do the following:

1. Refer back to your previous brute force solution as a baseline.
2. Work through the BUD optimization. Go through each (Bottlenecks, Unnecessary work, and Duplicated work) one at a time and write out as many examples of each as you can.
3. Next try to apply your existing knowledge. Try to list as many possible approaches or patterns you can find that might be useful.

1. Maximum Sum Subarray (Leetcode)
2. Climbing Stairs (Leetcode)
3. Merge Intervals (Leetcode)
4. Buy and Sell Stock (Leetcode)

Day 3

Okay just a few more problems that we need to finish up from last week and then we're done!

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 Binary Tree Depth (Leetcode)
2. Linked List Cycles (Leetcode)
3. Buy and Sell Stock (Leetcode)

Day 4

Today, we are going to work on optimizing code using the BUD framework. For each of the following problems, I've given you a basic brute force solution. I want you to do the following:

1. Compute the Best Conceivable Runtime (BCR) for the problem
    a. Identify any bottlenecks
    b. Identify any unnecessary work
    c. Identify any duplicated work
2. Draw on you preexisting knowledge plus what you've just come up with to try to find the most optimal solution you can to the problem

It is very important that you take this one step at a time. I'm trusting you here to follow the steps one at a time in order. You want to make sure you're practicing each one. Not all problems will have all parts of the BUD framework but at least take a minute to look.

Since I want you to focus specifically on the optimization of these problems, I've also included descriptions of the brute force solutions on the next page. If you're struggling with the brute force solutions, this may be a good chance for some extra practice, but otherwise, you can save some time by using the solution given.

1. Longest Consecutive Sequence (Leetcode)
2. Longest Palindromic Substring (Leetcode)
3. Merge K Sorted Lists (Leetcode)

Brute Force Solutions:

1. **Longest Consecutive Sequence**
   The simplest approach that we can take here is simply to sort the input array. Once the input is sorted, it is trivial for us to iterate over it and determine if any given sequence is consecutive.

2. **Longest Palindromic Substring**
   It is easy for us to find every substring and it is also easy for us to determine if any arbitrary string is a palindrome. Therefore we can just do each of these two steps and determine which substring that is a palindrome is the longest.

3. **Merge K Sorted Lists**
   The easiest brute force approach is just to copy everything into some sort of data structure that can either be sorted (an array) or maintains a sorted order (a BST or heap).

Day 5

Today is the same deal as yesterday. This is one of those things where getting plenty of practice and seeing multiple examples is going to be the best way to see results.

Today, we are going to work on optimizing code using the BUD framework. For each of the following problems, I've given you a basic brute force solution. I want you to do the following:

1. Compute the Best Conceivable Runtime (BCR) for the problem
    a. Identify any bottlenecks
    b. Identify any unnecessary work
    c. Identify any duplicated work
2. Draw on you preexisting knowledge plus what you've just come up with to try to find the most optimal solution you can to the problem

It is very important that you take this one step at a time. I'm trusting you here to follow the steps one at a time in order. You want to make sure you're practicing each one. Not all problems will have all parts of the BUD framework but at least take a minute to look.

Since I want you to focus specifically on the optimization of these problems, I've also included descriptions of the brute force solutions on the next page. If you're struggling with the brute force solutions, this may be a good chance for some extra practice, but otherwise, you can save some time by using the solution given.

1. LRU Cache (Leetcode)
2. Generate Parentheses (Leetcode)

Brute Force Solutions:

1. **LRU Cache**
   Implement this with a simple list storing the data. When an item is accessed, move it to the front of the list.

2. **Generate Parentheses**
   We can write a function to validate whether a combination of parentheses is valid (count the number of open parentheses. It should never be negative) and then we can use recursion to generate all possible combinations.

Day 6

Today is the same deal as yesterday. This is one of those things where getting plenty of practice and seeing multiple examples is going to be the best way to see results.

Today, we are going to work on optimizing code using the BUD framework. For each of the following problems, I've given you a basic brute force solution. I want you to do the following:

1. Compute the Best Conceivable Runtime (BCR) for the problem
    a. Identify any bottlenecks
    b. Identify any unnecessary work
    c. Identify any duplicated work
2. Draw on you preexisting knowledge plus what you've just come up with to try to find the most optimal solution you can to the problem

It is very important that you take this one step at a time. I'm trusting you here to follow the steps one at a time in order. You want to make sure you're practicing each one. Not all problems will have all parts of the BUD framework but at least take a minute to look.

Since I want you to focus specifically on the optimization of these problems, I've also included descriptions of the brute force solutions on the next page. If you're struggling with the brute force solutions, this may be a good chance for some extra practice, but otherwise, you can save some time by using the solution given.

1. Median of Sorted Arrays (Leetcode)
2. Word Break (Leetcode)

Brute Force Solutions:

1. **Median of Sorted Arrays**
   SInce we know how to compute the median of a single array, it's no harder to compute the median of these multiple arrays. All we have to do is merge them into a single array, make sure that array is sorted, and then find the median.
   Alternatively, we could count from the ends of both arrays towards the middle.

2. **Word Break**
   There is a set number of ways that we could break up the input string. That means we can just try every possible way of breaking it up and then validate whether or not each of those is comprised of only valid words.

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

Copy your code verbatim into Leetcode and keep track of any bugs.

1. Climbing Stairs
2. Merge Intervals

Day 2

1. Maximum Sum Subarray
    a. Bottlenecks
        i. We have to find all different subarrays
        ii. We have to compute the sum of each subarray
    b. Unnecessary work
        i. Do we need to actually find each separate subarray?
    c. Duplicated work
        i. As we compute the sums of each subarray, we're basically summing up the same set of numbers again and again
    d. Existing knowledge
        i. We could compute some sort of running sum
        ii. We can use the difference between 2 running sums to find the sum of a range
        iii. Either end of our array must be positive for it to be a maximum sum array

2. Climbing Stairs
    a. Bottlenecks
        i. Finding all the combinations is really time consuming
        ii. We have to store all the combinations, which makes this even worse
        iii. We have to validate each combination. It takes $O(n)$ time for each and there are $O(2^n)$ combinations
    b. Unnecessary work
        i. Do we really need to compute every combination?
        ii. Do we need to save every combination to memory or could we validate as we go?
    c. Duplicated work
        i. We are computing the same partial combinations again and again
    d. Existing knowledge
        i. Since we're doing recursion and we're solving the same subproblems over and over this is a prime candidate for DP

3. Merge Intervals
    a. Bottlenecks
        i. We have to iterate over every single time slice, regardless of how many intervals there are
        ii. We have to store every time slice to memory
    b. Unnecessary work
        i. We don't need to look in the middle of each interval. We can just look at the endpoints
    c. Duplicated work
        i. We could potentially eliminate any interval that completely overlaps another
    d. Existing knowledge
        i. We can determine if two intervals overlap by looking at the start and end of the intervals
        ii. Overlaps are transitive. If `A` and `B` overlap and `B` and `C` overlap, then we can merge `A` and `B` first and then merge the result with `C`

4. Buy and Sell Stock
    a. Bottlenecks
        i. We don't need to compare every combination of buying and selling
    b. Unnecessary work
        i. Again, we're making a lot of unnecessary comparisons
    c. Duplicated work
        i. Not much here
    d. Existing knowledge
        i. We can just compare local minima and local maxima
        ii. We can focus on finding the smallest value followed by the largest value
        iii. Pretty easy to take a greedy approach here

Day 3

Copy your code verbatim into Leetcode and keep track of any bugs.

1. Maximum Binary Tree Depth
2. Linked List Cycles
3. Buy and Sell Stock

Day 4

1. Longest Consecutive Sequence
    a. BCR
        i. `O(n)` - at minimum, we need to iterate over the entire array
    b. Bottlenecks
        i. Sorting is almost always a bottleneck. We are limited by how fast we can sort the array
        ii. Can we do this without sorting somehow?
        iii. We can try to construct a consecutive sequence. For each element in the array, see if the array includes the next consecutive element, then the next, and so on. Try starting from each element in the array and see what leads to the longest consecutive array.
    c. Unnecessary work
        i. If we don't sort, we're repeatedly iterating over the input. We would be better off putting all the items into a set where we can find whether `N` is in the array in `O(1)` time
    d. Duplicated work
        i. If we start from each number in the array, we will iterate over the same sequences multiple times. How do we avoid this? Can we check that an element is the first element in a sequence?
    e. Existing knowledge
        i. We pretty much have everything we need with the above
    f. Optimal solution
        i. First, add all the elements to a set
        ii. Then, for each element in the set, do the following
            1. Check if the element `N-1` is in the set. If it is, skip this because our item N is not the start of a consecutive sequence
            2. If not, check if `N+1` is in the set. Keep incrementing `N` by `1` and checking. Once `N+1` is no longer in the set, you've found the end of the sequence

2. Longest Palindromic Substring
    a. BCR
        i. `O(n)` - We have to iterate over the entire string at minimum
    b. Bottlenecks
        i. Looking at every possible substring is very time consuming
    c. Unnecessary work
        i. Is there a way we can identify whether a substring even has the possibility of being a palindrome? For example, we could eliminate all substrings where the first and last characters are not the same
    d. Duplicated work
        i. Computing whether every substring is a palindrome, we are computing the same substrings again and again as part of larger strings
    e. Existing knowledge
        i. We know what palindromes are and how to validate them
        ii. We know that if we take a palindrome and remove the first and last character, that string is still a palindrome
    f. Optimal solution
        i. Rather than find all substrings, let's just try to construct the largest palindrome we can
        ii. For each index in our string, assume that is the midpoint of our palindrome
        iii. Iteratively expand in both directions as far as you can while remaining a palindrome
        iv. Repeat the above for even-length palindromes wherever you see a character repeated

3. Merge K Sorted Lists
    a. BCR
        i. `O(N*K)` - we have to iterate through all the elements in all the arrays
    b. Bottlenecks
        i. Again, sorting is a bottleneck for us. Hopefully you're seeing a pattern
    c. Unnecessary work
        i. We are resorting the entire thing even though the individual arrays were in sorted order before
    d. Duplicated work
        i. N/A
    e. Existing knowledge
        i. We know that our input is sorted
        ii. From doing mergesorts in the past, we know how to merge 2 sorted arrays efficiently
        iii. We know that the minimum value out of all arrays is the first item in one of our set of arrays
    f. Optimal solution
        i. We know the minimum value must be one of the `K` first elements in the set of arrays, so we can add that to our result array
        ii. Then we know that the second element in our result must be either the second element in that particular array or the first element in any of the other arrays
        iii. It would normally take us `O(K)` time to find the minimum of all these elements, but we can use a priority queue to do this more efficiently
        iv. Since we're only adding/removing 1 element at a time, we add the first `K` elements to the array. Then as we remove the smallest, we add the next element from the array that smallest element came from

Day 5

1. LRU Cache
    a. BCR
        i. With this problem, our goal complexity is predefined
    b. Bottlenecks
        i. Finding items in our list is by far the most time consuming part here
        ii. Even if we have an easier way to look it up, we still have to find the element itself to move it to the front of the list
    c. Unnecessary work
        i. We might be iterating over the list multiple times, which we can try to avoid although it doesn't affect the time complexity
    d. Duplicated work
        i. Every time we want to find an item in the cache we need to iterate through the entire list
    e. Existing knowledge
        i. We could keep track of whether an element is in the cache using a set, but this doesn't help us update it if we do find it
        ii. If we could directly access the node that represents the value, we could remove it from the list in constant time
    f. Optimal solution
        i. Keep a hashtable that maps values to the node the value is stored in
        ii. When someone gets a value from the cache, look at the hashtable keys to see if the value exists
        iii. If it does, look up the value for that key (the node) and move it to the front of the list

2. Generate Parentheses
    a. BCR
        i. $O(n * 2^n)$ - we have potentially 2^n different combinations and each combination is of length n
    b. Bottlenecks
        i. Computing and storing every possible combination is time consuming
        ii. What if we could generate only the valid combinations?
    c. Unnecessary work
        i. Similar to above, we really don't need to generate all these invalid combinations
    d. Duplicated work
        i. If we do this recursively, maybe we can use DP somehow
    e. Existing knowledge
        i. This is a good opportunity to draw on the core recursive patterns. If you're not in *CIM: Recursion*, you can learn more about these here
    f. Optimal solution
        i. Let's construct valid sets of parentheses
        ii. We know the following:
            1. If the number of open and closed parentheses in the string so far is equal, then the next one must be open
            2. If there are more open than closed, we can include either open or closed as long as we don't exceed the max number of pairs
        iii. With this, we can write a recursive function that simply generates all possibilities given this info
        iv. From there, we could possibly cache partial strings and use DP, but the time cost of copying all the partial strings will eliminate any benefit

Day 6

1. Median of Sorted Arrays
    a. BCR
        i. `O(1)` - if we can find the median of 1 array in `O(1)`, it seems plausible that there could be an `O(1)` solution for this
    b. Bottlenecks
        i. Copying all of our data is limiting us to `O(n)` time
        ii. The only way we'll be able to do better than `O(n)` is some sort of binary search or something where we don't look at all the values
    c. Unnecessary work
        i. For starters, there's no need for us to copy all the values into a separate array, since we can do the same thing in place, but that doesn't improve our time complexity
        ii. Do we need to look at every element in the array?
    d. Duplicated work
        i. N/A
    e. Existing knowledge
        i. We know that if we want to improve this time complexity we need to do some sort of binary search
        ii. We know how to find the median of a single array
    f. Optimal solution
        i. Find the median of each individual array
        ii. If they are equal, then this is the median of both arrays
        iii. If not, then we know that the larger one is larger than the median and the smaller is smaller than the median
        iv. That means we take the smaller half of the array that had the larger median and the larger half of the array with the smaller median
            1. If the smaller value is less than the median, then since the array is sorted, we know that the entire left side of the array is *also* less than the median. So we're removing `N/2` elements that are less than the median and then we are also doing the reverse with `N/2` elements that are larger than the median
        v. Repeat until you find the median

2. Word Break
    a. BCR
        i. `O(n)` - minimum we need to iterate over the entire string
    b. Bottlenecks
        i. We are trying every combination regardless of whether it is valid or not
        ii. We have to validate every single combination
    c. Unnecessary work
        i. If we could generate only valid combinations we would save a lot of time
        ii. We don't really need to generate the combinations at all because we just need to know if a combination exists
        iii. Depending on the number of words in our dictionary we could just try to find all combinations of dictionary words
    d. Duplicated work
        i. We are potentially computing the same substrings again and again
        ii. Maybe we can use DP here
    e. Existing knowledge
        i. We hopefully know how to use the FAST Method for doing DP
    f. Optimal solution
        i. We will solve this using a bottom-up DP approach
        ii. For each index of our array, `arr[i]` represents whether or not `substring(0, i)` can be validly split up
        iii. To compute `arr[i+1]`, we iterate from `j=0` to `i`. For each `arr[j] != 0`, if `substring(j, i+1)` exists in our dictionary, then `arr[i+1]` is true. Otherwise `arr[i+1]` is false.