Topic Introduction

Today's PrepLetter is all about **Dynamic Programming for Optimal Subsequences**. If you've spent time on LeetCode or in interviews, you've probably seen problems that ask: "Find the longest sequence that fits these rules." But what's really going on under the hood?

Let's break it down:

What is a Subsequence?
A subsequence is a sequence that can be derived from another sequence by deleting some or no elements, without changing the order of the remaining elements. For example, in `[3, 1, 5, 2, 6]`, `[3, 5, 6]` is a subsequence.

Dynamic Programming (DP) for Subsequences:
DP is a powerful way to break problems into overlapping subproblems, store their solutions, and build up to the answer. For subsequences, DP often helps us find the "best" (longest, largest, most optimal) subsequence that meets certain criteria.

When and Why is This Useful?
Interviewers love these problems because they test your ability to recognize patterns, break down problems, and optimize brute-force approaches. They also pop up in real-world tasks like scheduling, bioinformatics, and data analysis.

Simple Example (Not from our main problems):
Given `[1, 2, 3, 2, 4, 6]`, what is the length of the longest consecutive increasing subsequence?

- `[1, 2, 3]` is one, `[2, 4, 6]` is another.
- The answer is `3`.

Usually, you solve this by:

- Keeping track of the length of increasing sequences.
- Updating your answer as you go.

The Core Pattern:
For subsequence problems, you'll often:

- Sort or preprocess the input (sometimes).
- Use DP to store the answer for subproblems (like, "What is the optimal answer ending at index i?").
- Build up your answer as you iterate through the sequence.

Today, we'll tackle three classic problems:

- Longest Increasing Subsequence (LIS)
- Russian Doll Envelopes
- Largest Divisible Subset

Why these three?
All three problems require you to find the largest or longest subsequence that follows a set of ordering rules:

- LIS uses value order.
- Russian Doll Envelopes uses 2D sorting with containment rules.
- Largest Divisible Subset uses divisibility constraints.

Despite their differences, they all share the DP backbone, and can sometimes be further optimized with clever tricks like binary

search.

Problem 1: Longest Increasing Subsequence

Problem Statement (Rephrased):

LeetCode 300: Longest Increasing Subsequence

Given an integer array `nums`, find the length of the longest strictly increasing subsequence.

Example:

Input: `[10, 9, 2, 5, 3, 7, 101, 18]`

Output: `4`

Explanation: The longest increasing subsequence is `[2, 3, 7, 101]`.

How to Think About It:

Try writing out all possible increasing subsequences. It gets complicated fast!

- Try pen and paper: For `[0, 1, 0, 3, 2, 3]`, what are the increasing subsequences?
- The brute-force approach would be to try all subsequences — but this is exponential in time.

Another Test Case to Try:

Input: `[7, 7, 7, 7, 7]`

What should the output be?

Brute-force Approach

- Try every possible subsequence and check if it's increasing.
- Time Complexity: O(2^n) — very slow for large arrays.

Optimal DP Approach

Core Pattern:

We want to know, for each index, what is the length of the longest increasing subsequence that ends at that index.

Step-by-Step Logic:

- Create a DP array where `dp[i]` is the length of the LIS ending at index `i`.
- For every index `i`, look at all previous indices `j < i`:
  - If `nums[j] < nums[i]`, then `nums[i]` can extend the LIS ending at `j`.
  - So, set `dp[i] = max(dp[i], dp[j] + 1)`.
- Initialize all `dp` values to 1 (every element is an LIS of length 1 by itself).
- The answer is the maximum value in `dp`.

Note: There is a more optimal O(n log n) algorithm using binary search, but let's focus on DP for learning!

Python Solution

```
def lengthOfLIS(nums):
    if not nums:
        return 0

    n = len(nums)
    # dp[i] stores the length of the LIS ending at index i
    dp = [1] * n

    for i in range(n):
        for j in range(i):
            if nums[j] < nums[i]:
                dp[i] = max(dp[i], dp[j] + 1)

    return max(dp)
```

Time Complexity: O(n^2)

Space Complexity: O(n)

Explanation

- `dp[i]` starts at 1 because every number is an LIS of length 1 by itself.
- For each `i`, we check all `j < i`:
  - If `nums[j] < nums[i]`, then we might be able to extend the LIS.
  - We update `dp[i]` to be the longer of what it is already, or `dp[j] + 1`.
- Finally, we return the maximum value in `dp`, since the LIS could end anywhere.

Trace Example

Let's trace `[10, 9, 2, 5, 3, 7, 101, 18]`:

- Start with `dp = [1, 1, 1, 1, 1, 1, 1, 1]`
- At `i=3` (`nums[3]=5`):
  - Check `nums[2]=2` (2 < 5): so `dp[3]=dp[2]+1=2`
  - So `dp = [1, 1, 1, 2, ...]`
- Continue this for all elements.
- Final `dp = [1, 1, 1, 2, 2, 3, 4, 4]`
- Maximum is `4`.

Try Manually:

Input: `[0, 8, 4, 12, 2]`

What does `dp` look like at each step?

Take a moment to solve this on your own before jumping into the code.

> Did you know?
> The O(n log n) approach uses patience sorting and binary search. If you're feeling ambitious, give that a try after mastering DP!

Problem 2: Russian Doll Envelopes

Problem Statement (Rephrased):
LeetCode 354: Russian Doll Envelopes
Given a list of envelopes, each with a width and a height, find the maximum number of envelopes you can "nest" (put one inside another). An envelope can fit into another if both its width and height are strictly less.

Example:
Input: `[[5,4],[6,4],[6,7],[2,3]]`
Output: `3`
Explanation: `[2,3] => [5,4] => [6,7]` is a valid nesting.

How is this similar to LIS?
We want the longest chain where each envelope can fit into the next — that's just an LIS in two dimensions!

Brute-force Approach

- Try every possible order and check for nesting.
- This is factorial time: O(n!).

Optimal Approach

Key Insight:

- Sort envelopes by width. But for the same width, sort by height *descending*.
    - Why descending? To prevent false positives when envelopes have the same width.
- Then, find the LIS on the heights.

Step-by-Step:

- Sort envelopes:
    - By width ascending.
    - If widths are equal, by height descending.
- Extract the heights into a separate list.
- Apply the LIS algorithm to the heights.

Pseudocode:

```
Sort envelopes:
    - primary: width ascending
    - secondary: height descending

heights = [height for each envelope in sorted list]

dp = [1] * len(heights)
for i in range(len(heights)):
    for j in range(i):
        if heights[j] < heights[i]:
            dp[i] = max(dp[i], dp[j] + 1)
return max(dp)
```

Example Trace:

Input: `[[5,4],[6,4],[6,7],[2,3]]`

After sorting: `[[2,3], [5,4], [6,7], [6,4]]`

Heights: `[3,4,7,4]`

Apply LIS on `[3,4,7,4]`:

- dp = [1, 2, 3, 2]
- Max is 3.

Try Manually:

Input: `[[1,1],[1,1],[1,1]]`

What's the answer?

Complexity

- Sorting: O(n log n)
- LIS DP: O(n^2)
- Total: O(n^2)

If you use LIS with binary search, you can achieve O(n log n).

Key Similarity:

Both this and LIS reduce to finding the longest increasing subsequence — but here, after a clever sort!

Problem 3: Largest Divisible Subset

Problem Statement (Rephrased):

LeetCode 368: Largest Divisible Subset

Given a set of positive integers, find the largest subset where for every pair, one number divides the other.

Example:

Input: `[1,2,3]`

Output: `[1,2]` or `[1,3]`

Explanation: Both `[1,2]` and `[1,3]` are valid, as 2 is divisible by 1, and 3 is divisible by 1.

What's Different?

Instead of values needing to be increasing, each value must be divisible by previous values.

Brute-force Approach

- Try all subsets, check divisibility for each pair.
- Exponential time: O(2^n).

Optimal Approach

Key DP Pattern:

- Sort the numbers.
- For each number, look at all previous numbers. If current number is divisible by a previous number, you can extend the subset.

Step-by-Step:

- Sort the array.
- For each `i`, check all `j < i`:
    - If `nums[i] % nums[j] == 0`, then `nums[i]` can extend the subset ending at `j`.
- Keep track of:
    - The length of the largest subset ending at each position (`dp` array).
    - The path/previous index for reconstruction.

Pseudocode:

```
Sort nums

dp = [1] * len(nums)
prev = [-1] * len(nums)
max_len = 1
max_idx = 0

for i in range(len(nums)):
    for j in range(i):
        if nums[i] % nums[j] == 0:
            if dp[j] + 1 > dp[i]:
                dp[i] = dp[j] + 1
                prev[i] = j
```

```
    if dp[i] > max_len:
        max_len = dp[i]
        max_idx = i

# Reconstruct subset:
result = []
while max_idx != -1:
    result.append(nums[max_idx])
    max_idx = prev[max_idx]
return reversed(result)
```

Example Trace:

Input: `[1,2,4,8]`

- Sorted: `[1,2,4,8]`
- dp: [1,2,3,4]
- prev: [-1,0,1,2]
- Subset: [8,4,2,1] reversed is [1,2,4,8]

Try Manually:

Input: `[1,3,6,24]`

What is the largest divisible subset?

Complexity

- Sorting: O(n log n)
- DP: O(n^2)
- Space: O(n)

Final Nudge:

Try implementing this yourself! Notice how similar the DP structure is, but the key difference is the divisibility check replacing the "less than" comparison.

Summary and Next Steps

Why These Problems?

All three force you to find the “best” subsequence given an ordering rule. Whether that rule is increasing order, fitting inside, or divisibility, the DP pattern is almost the same!

Key Patterns:

- Sort if it helps simplify comparisons.
- Use a DP array to track optimal subsequences ending at each index.
- For reconstruction, keep track of previous indices.

- Sometimes, further optimizations (like binary search) are possible.

Common Traps:

- Forgetting to sort (especially for Russian Doll Envelopes and Largest Divisible Subset).
- Off-by-one errors in DP loops.
- Not handling equal values or divisibility correctly.

Action List

- Solve all 3 problems on your own, even the one with code provided!
- Try Problem 2 and 3 using the O(n log n) LIS approach (hint: patience sorting + binary search).
- Explore other problems that use DP for subsequences, like "Longest Bitonic Subsequence."
- Compare your solution with the official and top-voted ones for edge-case handling and style.
- Don’t be discouraged if you get stuck — persistence is the main skill!

Keep practicing — each of these patterns will unlock a whole family of interview problems.

Happy coding!