# Dynamic Programming

A top-down approach

Part II

# Common DP Variants

# 0/1 Knapsack

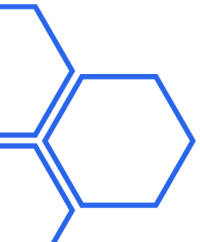
- Knapsack problems generally involve **filling a limited container** with items where we want to count or optimize some quantity associated with the items.

- **In 0/1 knapsack problem** we choose a subset of items such that we **maximize their total value** and their total weight does not exceed the capacity of the container

- Problem Link

# 0/1 Knapsack

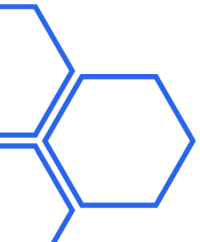
- Start with what constitutes a good substructure. How to break down the problem ?
- Consider the substructure starting from the index `i`.
- What is/are going to be the state/states?



# 0/1 Knapsack

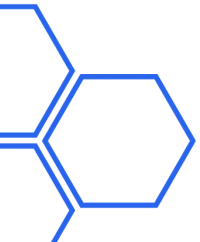
- For the sub-problems starting from the `i`-th index we can categorize the subproblems into two.

    - When we pick the `i`-th item and **(include)**
        - This will increase our value but it decreases our capacity.
        - Therefore, our answer is `val[i]` + the answer for the sub-problem starting from the `i + 1` -th index, but our capacity is decreased.



# 0/1 Knapsack

- For the sub-problems starting from the `i`-th index we can categorize the subproblems into two.
    - When we do not pick an item **(exclude)**
        - This will gain us no value but the capacity stays the same
        - The answer, in this case, is the answer for the sub-problem starting from the `i + 1` -th index with same capacity.



# 0/1 Knapsack

- Thus we can pick the item index `i` and the remaining capacity `W` of our knapsack as the state
    - **(include)** When we pick the i-th item and
        - `include = val[i] + dp(i+1, W - wt[i])`
    - **(exclude)** When we do not pick the i+1-th item
        - `exclude = dp(i+1, W)`
- Since we are aiming for maximum value we will be taking the maximum of the two.

# 0/1 Knapsack

- What about the base cases?
- If we exhaust the list or use all the capacity

```
if i >= len(wt) or W == 0:
    return 0
```

# Longest Increasing Subsequence

Problem Link

# Longest Increasing Subsequence

- Finding the longest subsequence of an array such the elements are increasing

1 3 4 1 2 6 5 7

- A subsequence is a list **that can be derived by deleting zero or more elements of a list**. Is it possible to use 0/1 Knapsack? How?

# Longest Increasing Subsequence

- We can consider each each index and whether we should delete this index or not to form an increasing subsequence.

1 3 4 1 2 6 5 7

- What are the states?
- The pair `(ind, last)` forms the state where `ind` is the current index and `last` is the last element in the increasing subsequence we have built until index `ind`.

# Longest Increasing Subsequence

- We can form a subsequence that starts at index `i` by joining `nums[i]` with all the increasing subsequences that start index `j` if
  - `j` is to the right of `i`
  - `nums[j] > nums[i]`

```
for j > i and nums[j] > nums[i]
    dp(i) = max(dp(j) + 1, dp(i))
```

# Longest Common Subsequence

The Longest Common Subsequence (LCS) problem is **finding the longest subsequence** present in the given two sequences in the same order.

1 3 4 1 2 6 5 7

2 1 8 3 6 7

What is a good substructure?

# Longest Common Subsequence

If we know the answer for `nums1[i+1 ... ]` and `nums2[j+1 ...]`, how can we combine them?

- If `nums1[i] = nums2[j]`, the answer is `1 +` the answer for `nums1[i+1 ...]` and `nums2[j+1...]`. Why?

```
dp(i, j) = 1 + dp(i+1, j+1)
```

- else the answer can not contain both `nums1[i]` and `nums2[j]`

```
dp(i, j) = max(dp(i+1, j), dp(i, j+1))
```

# Longest Common Subsequence

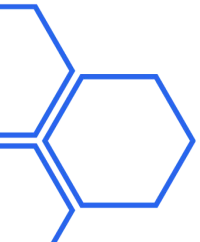
Problem Link

# Common Pitfalls

# 1. Thinking of greedy approach

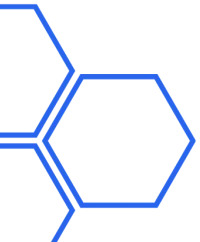
Greedy algorithms aim for **local optimality**, but they can fall short in finding the global optimum, leading to suboptimal or incorrect outcomes.

Greedy stays ahead... sometimes ahead of itself :)



# 2. Incorrect Recurrence Relation

- Formulating an incorrect recurrence relation, can lead to incorrect results or inefficiencies.



# 3. Lack of proper memoization

- Forgetting to apply memoization or implementing it incorrectly can lead to redundant computations.

# 4. Inefficient time complexity

DP is a smart bruteforce at the end of the day. Hence, it can be applied to most problems but it does not mean it is always efficient. Especially, if the subproblems have little to none overlappingness.

**Don't fail to consider alternative data structures or optimizing the recurrence relation.**

# Checkpoint

Link

# Practice Questions

N-th Tribonacci Number

Coin Change

Target Sum

Unique Paths

Minimum Path Sum

Best Time to Buy and Sell Stock with Transaction Fee

Best Time to Buy and Sell Stock with Cooldown

House Robber III

# Resources

Leetcode Explore Card

Dynamic Programming lecture #1 - Fibonacci, iteration vs recursion

Competitive Programmer's Handbook

Dynamic programming for Coding Interviews: A bottom-up approach to problem solving

# Quote of the Day

“Those who cannot remember the past are condemned to repeat it.”

— George Santayana