CS1010X: Programming Methodology I
Chapter 13: Memoization and Dynamic Programming

Lim Dillion
dillionlim@u.nus.edu

2025/26 Special Term 2
National University of Singapore

Chapter 13: Memoization and Dynamic Programming

Learning Objectives

By the end of this chapter, students should be able to:

- **Explain** why naive recursion can be inefficient due to repeated evaluation of the same subproblems.
- **Define memoization** as a top-down optimization technique that caches results of expensive function calls.
- **Implement** memoization using the **wrapper (decorator) pattern** in Python.
- **Identify** the two key properties required for dynamic programming: **optimal substructure** and **overlapping subproblems**.
- **Contrast** top-down memoization (lazy evaluation) with bottom-up dynamic programming (eager evaluation).
- **Design** DP solutions by specifying a state, recurrence, base cases, and evaluation order.
- **Apply** dynamic programming to classical problems (Fibonacci, $\binom{n}{k}$, coin change, rod cutting, 0/1 knapsack), including **solution reconstruction**.
- **Evaluate** when greedy strategies are appropriate and why they can fail when compared to DP.
- **Implement** exception handling using `try`, `except`, `else` and `finally`.

13.1 Motivation: Why Naive Recursion Breaks Down

Recursion is an elegant way to solve problems that decompose into smaller, self-similar subproblems. However, naive recursion can be catastrophically inefficient when the same subproblems are recomputed many times.

13.1.1 Case Study: The Fibonacci Sequence

The Fibonacci numbers are defined by:

$$F(0) = 0, \quad F(1) = 1, \quad F(n) = F(n-1) + F(n-2) \;\; (n \geq 2).$$

```python
def fib(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    return fib(n-1) + fib(n-2)
```

The call tree grows rapidly. More importantly, *it contains repeated subtrees*. For example, `fib(3)` appears multiple times when computing `fib(5)`.

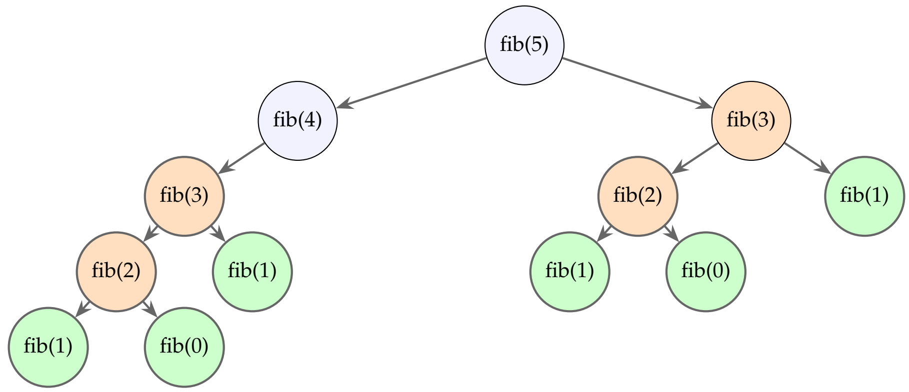


Figure 13.1: Call tree for `fib(5)`. Base cases are in green; repeated subproblems are highlighted in orange (**Note:** Not all nodes are drawn out due to space constraints).

The time complexity of naive Fibonacci recursion is exponential (often expressed as $O(\varphi^n)$ where $\varphi \approx 1.618$). In practice, `fib(100)` is infeasible with this approach.

13.2 Dynamic Programming

Dynamic programming (DP) is a systematic technique for accelerating recursive solutions. Dynamic programming is applicable when a problem satisfies both:

- **Optimal substructure:** An optimal solution can be constructed from optimal solutions to subproblems.
- **Overlapping subproblems:** The recursion revisits the same subproblems many times.

13.2.1 Optimal Substructure

A problem has **optimal substructure** if solving it optimally requires solving its subproblems optimally. Examples:

- Fibonacci: $F(n)$ depends on $F(n-1)$ and $F(n-2)$, which must be correct.
- Coin change: the best way to make amount A uses the best way to make $A - c$ after choosing coin c.

13.2.2 Overlapping Subproblems

A problem has **overlapping subproblems** when a naive recursive solution recomputes the same states repeatedly (like `fib(3)` above). DP avoids this by computing each subproblem *once* and storing its result.

Memoization vs Dynamic Programming

Generally, we use the two terms above interchangeably. In this course, however, memoization **exclusively** refers to the top-down optimization, while DP refers to the bottom-up version.

13.3 Memoization (Top-Down Optimization)

Memoization accelerates recursion by caching previously computed results. When a memoized function is called:

1. Look up the argument tuple in a cache table. Dictionaries are well-suited for this because they provide $O(1)$ insert and lookup.
2. If present, return the cached answer immediately.
3. Otherwise, compute the answer, store it, then return it.

13.3.1 The Wrapper / Decorator Pattern

Memoization is most cleanly implemented by **wrapping** an existing function with caching logic (a decorator-style approach). The wrapped function behaves like the original, but with caching.

Below is a generic memoizer that:

- Uses a per-function cache dictionary (no global cross-function collisions).
- Accepts arbitrary positional arguments via `*args`.

```
memoize_table = {} # Each dictionary is for a different function
def memoize(f):
    """Return a memoized wrapper of f using an internal cache dict."""
    name = f.__name__ # In-built system call to get the name of the function
    if name not in memoize_table:
        memoize_table[name] = {} # If the name is not already in the dict, we will create a dictionary within the function
    table = memoize_table[name]
    def helper(*args):
        if args in table:
            return table[args]
        result = f(*args)
        table[args] = result
        return result
    return helper
```

Calling the Wrong Function

A frequent mistake is calling the wrong function because one has forgotten to update the internal recursive calls.

```
# PROBLEM: The recursive step inside 'fib' calls 'fib',
# NOT 'memo_fib'. The cache is bypassed!
memo_fib = memoize(fib)
```

If the recursive step calls `fib` instead of `memo_fib`, the internal steps are never checked against the cache, defeating the purpose. We will see how to fix this with the decorator pattern later.

13.3.2 Case Study: Fibonacci

We can implement the memoized Fibonacci sequence in two distinct ways. Both achieve $O(n)$ time complexity but manage the lookup table differently.

Method 1: Overwrite the function name (simple)

This approach uses the generic `memoize` function defined previously. It relies on the shared `memoize_table`. To fix the recursion gap mentioned in the warning above, we must overwrite the variable `fib` with the wrapped version. This ensures that when the function calls `fib(n-1)` internally, it calls the *memoized* version, not the original one.

```python
def fib(n):
    if n < 2:
        return n
    return fib(n-1) + fib(n-2)

fib = memoize(fib)  # overwrite name so memoized version is used
print(fib(50))      # fast
```

If we were to just write `memo_fib = memoize(fib)`, as per the warning above, the recursive step in `fib` will still call `fib` instead of `memo_fib`. By overwriting the name, the memoized version is used instead. This gives us the desired time complexity.

However, having to write `fib = memoize(fib)` is tedious and leads to many inaccuracies. Python (and many languages) resolve this problem by making use of the **decorator pattern**.

Decorator Syntax in Python

Python decorators allow you to modify or extend the behavior of functions and methods without changing their actual code.

Therefore, in Python, we use the `@` symbol as **syntactic sugar** for wrapping a function. The following code snippets are equivalent:

Using the `@` syntax

```python
def decorator(func):
    def wrapper():
        print("Before func.")
        func()
        print("After func.")
    return wrapper

@decorator
def say_whee():
    print("Whee!")

say_whee()
```

Using manual assignment

```python
def decorator(func):
    def wrapper():
        print("Before func.")
        func()
        print("After func.")
    return wrapper

def say_whee():
    print("Whee!")

# Manually wrapping the function
say_whee = decorator(say_whee)

say_whee()
```

As explored in Lecture 5, functions in Python are first-class citizens, so we can pass them around as if they were objects. Therefore, the decorator is able to take in the current function and return a new, higher-order function that extends the behaviour of functions, without having to change the underlying function code.

Common Built-in Decorators

Python provides several built-in decorators that are used in class definitions. These decorators modify behavior of methods and attributes in a class, making it easier to manage and use them effectively. The most frequently used built-in decorators are `@staticmethod` and `@classmethod`.

`@staticmethod` is used to define a method that does not operate on an instance of class (i.e. it does not use `self`). Static methods are called on the class itself, not on an instance of class. (We will explore more implications of static methods in the chapter on OOP in Java).

For example, if we had a class to do math operations, it makes no sense to have to instantiate a class to use an add function. We therefore define the method as static, so that it is associated with the class.

```python
class MathOperations:
    @staticmethod
    def add(x, y):
        return x + y

# Using the static method
res = MathOperations.add(5, 3)
print(res)
```

Then, `add` is a static method defined with `@staticmethod` decorator. It can be called directly on class MathOperations without creating an instance.

`@classmethod` is used to define a method that operates on class itself (i.e. it uses `cls`). Class methods can access and modify class state that applies across all instances of class.

Suppose we had a class Employees, and in an internal messaging chat, every message has a prefix associated with it. We want to be able to change the prefix company-wide, so we would need a class method to do this.

```python
class Employee:
    # A class variable to store the prefix for all employees
    message_prefix = "[INTERNAL]"

    def __init__(self, name, salary):
        self.name = name
        self.salary = salary

    @classmethod
    def set_message_prefix(cls, prefix):
        """
        Sets a new message prefix for all instances of the Employee class.
```

```python
        """
        cls.message_prefix = prefix

    def send_message(self, message):
        """
        Simulates sending a message using the current class prefix.
        """
        full_message = f"{self.message_prefix} From {self.name}: {message}"
        print(f"Message sent: {full_message}")

print(f"Initial prefix: {Employee.message_prefix}")

# Create an employee instance and send a message
emp1 = Employee("Alice", 60000)
emp1.send_message("Please review the Q4 reports.")

# Use the class method to change the prefix for all employees
Employee.set_message_prefix("[CONFIDENTIAL]")
print(f"\nUpdated prefix: {Employee.message_prefix}")

# The new prefix is reflected in new and existing instances
emp2 = Employee("Bob", 70000)
emp2.send_message("Meeting moved to 3 PM.")
emp1.send_message("Acknowledged.")
```

Extra: `lru_cache`

While we implement memoization manually to demystify the underlying mechanics, specifically the use of hash maps (dictionaries) to store results, the `functools` package provides a production-ready solution via the `functools.lru_cache` decorator.

Why use `lru_cache` in practice?

- **Memory Management:** Unlike a simple dictionary that grows indefinitely, a Least Recently Used (LRU) cache discards the oldest entries when it reaches a `maxsize`, preventing memory exhaustion.
- **Thread Safety:** The implementation is written in C and is thread-safe, making it suitable for concurrent applications.
- **Performance:** It offers built-in statistics (hits, misses) via the `.cache_info()` method to help you tune your cache size.

In this module, we prioritize the manual approach to help you internalize how state is preserved across recursive calls.

Therefore, we can rewrite our `fib` function as

```python
@memoize
def fib(n):
    if n < 2:
        return n
    return fib(n-1) + fib(n-2)

print(fib(50))       # fast
```

Method 2: Local helper

This approach is self-contained. It encapsulates the cache dictionary *inside* the function scope, avoiding reliance on global variables. This is often cleaner for single-use functions.

```python
def fib_closure(n):
    cache = {}
    def helper(k):
        if k in cache:
            return cache[k]
        if k < 2:
            return k
        cache[k] = helper(k-1) + helper(k-2)
        return cache[k]
    return helper(n)

print(fib_closure(50))
```

13.3.3 Complexity of Memoized Fibonacci

With memoization, each state $0, 1, \ldots, n$ is computed once.

- **Time:** $O(n)$
- **Space:** $O(n)$ for the cache, plus $O(n)$ recursion depth (stack)

Hash Collisions

Memoization assumes average-case $O(1)$ dictionary lookup/insert. In extreme collision scenarios, a hash table can degrade, but Python's `dict` is engineered so average-case $O(1)$ is the correct practical model for algorithmic analysis.

13.3.4 Case Study: N Choose K (Combinations)

We want to compute $\binom{n}{k}$, the number of ways to choose k items from a set of n. The recursive formula (Pascal's Identity) is:

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

Base Cases: $\binom{n}{0} = 1$ and $\binom{n}{n} = 1$.

Using our generic `memoize` wrapper, we can solve this efficiently.

```python
@memoize
def choose(n, k):
    if k == 0 or k == n:
        return 1
    if k > n:
        return 0
    return choose(n-1, k-1) + choose(n-1, k) # Apply memoization
print(choose(50, 25)) # Computes instantly
```

Unlike Dynamic Programming, which fills the entire table (see next section), memoization is **lazy**. It only computes states required by the specific recursion path. Executing `choose(6, 3)` results in a **sparse table**. Notice how many cells remain empty (denoted by `#f`) because the recursion tree never requested them.

	k=0			k=3
n=0	#f	#f	#f	#f
n=1	1	1	#f	#f
	1	2	1	#f
	1	3	3	1
	1	4	6	4
	1	5	10	10
n=6	#f	6	15	20

Sparse Table Concept:
The algorithm calculates values only on a "need-to-know" basis. The empty cells (`#f`) represent computations that were skipped entirely, saving time.

Figure 13.2: The table state after calling `choose(6, 3)`.

Complexity Analysis:

- **Time:** $O(n \cdot k)$. With memoization, we only compute each unique state (n, k) exactly once. The number of unique states needed to solve $\binom{n}{k}$ is roughly the size of the rectangle defined by n and k in Pascal's Triangle.
- **Space:** $O(n \cdot k)$. The memoization table (hash map) stores a result for every unique pair of (n, k) encountered.

13.3.5 Case Study: Coin Change

Given coin denominations `coins` and target amount A, compute the **minimum** number of coins to make A.

We introduce the idea of a state and a transition more formally in the next section, but the brief idea is that **states** are what constitutes a subproblem, and **transitions** are how the subproblems are related. In general, if you know the state and transition of a recursive problem, you should be able to solve the problem. So, for this problem:

- **State:** `coin_change(a)` = minimum coins needed to form amount a.
- **Transition:** for each coin c, try taking it:

$$coin\_change[a] = 1 + \min_{c \in coins} coin\_change(a - c)$$

```python
def coin_change_recursive(coins, amount):
    # 1. Create local cache
    memo = {}

    def helper(curr_amount):
        # Check cache
        if curr_amount in memo: return memo[curr_amount]

        # Base Cases
        if curr_amount == 0: return 0
        if curr_amount < 0: return float('inf') # Impossible path

        # Recursive Step: Try every coin
        min_coins = float('inf')
        for c in coins:
            # 1 (current coin) + result for remainder
            res = 1 + helper(curr_amount - c)
            if res < min_coins:
                min_coins = res

        memo[curr_amount] = min_coins
        return min_coins

    result = helper(amount)
    return result if result != float('inf') else -1
```

- **Time:** $O(A \times C)$. There are $O(A)$ unique subproblems (states from 1 to *amount*). For each state, we iterate through all C coin denominations to find the minimum. Total operations $\approx A \times C$.
- **Space:** $O(A)$.
  - **Memoization Table:** Stores solutions for up to A amounts.
  - **Recursion Stack:** In the worst case (e.g. solving for amount A using only coins of value 1), the recursion depth can reach $O(A)$.

Reconstructing the chosen coins (path reconstruction)

Suppose we now wanted to reconstruct the coins needed. Importantly, we track which coin c resulted in the minimum count to allow for backtracking later. Therefore, to find the coins themselves, we store the `best_coin` chosen for every `curr_amount` and backtrack from the target A down to 0.

```python
def coin_change_with_path(coins, amount):
    memo = {}
    # Stores the first coin used to get the optimal result for amount
    choice_map = {}
    def helper(curr_amount):
        if curr_amount in memo: return memo[curr_amount]
        if curr_amount == 0: return 0
        if curr_amount < 0: return float('inf')
        min_coins = float('inf')
        best_coin = -1
        for c in coins:
            # Recursive step
            res = 1 + helper(curr_amount - c)
            # Update minimum if we found a better path
            if res < min_coins:
                min_coins = res
                best_coin = c
        memo[curr_amount] = min_coins
        choice_map[curr_amount] = best_coin # Store choice for
            backtracking
        return min_coins
    # 1. Run the optimization
    count = helper(amount)
    if count == float('inf'):
        return -1, []
    # 2. Backtrack to find the coins
    coins_used = []
    curr = amount
    while curr > 0:
        c = choice_map[curr]
        coins_used.append(c)
        curr -= c
    return count, coins_used
```

Complexity Analysis:

- **Time:** $O(A \times C)$. The core logic visits every amount state from 1 to A exactly once (due to memoization). For each state, it iterates through all C coin denominations. The backtracking step runs in $O(A)$ time (worst case: solution is all 1), which is subsumed by the $O(A \times C)$ DP process.
- **Space:** $O(A)$.
  - **Storage:** We maintain 2 dictionaries (`memo`, `choice_map`) storing up to A entries.
  - **Stack Depth:** The call stack can grow up to depth A (in the worst case where we keep subtracting 1).

13.4 Bottom-Up Dynamic Programming

While memoization drills down from the top (starting at the target N and recursing down to base cases), **bottom-up dynamic programming** builds the solution from the ground up.

This is often called "eager" because it calculates the answer for *every* possible subproblem, anticipating that they will be needed, rather than waiting to be asked.

Instead of recursion, we use iteration (loops). This offers significant advantages:

- **No Recursion Overhead:** We avoid the memory cost of the call stack.
- **Control:** We explicitly define the order in which states are solved, often allowing for space optimizations.

13.4.1 The 5-Step Framework

To convert a recursive solution to Bottom-Up, follow these steps:

1. **Define State:** What variables identify a subproblem? (e.g. i for index, w for weight).
2. **Recurrence:** How do you derive state i from previous states (e.g. $i-1$)?
3. **Base Cases:** What are the trivial states where the answer is known (e.g. $i=0$)?
4. **Evaluation Order:** In what direction must we loop so that dependencies are ready when needed? (e.g. $0 \to N$).
5. **Complexity:** Analyze the table size (Space) and work per cell (Time).

13.4.2 Case Study: Fibonacci Sequence

State: $dp[i]$ is the i-th Fibonacci number.
Transition: $dp[i] = dp[i-1] + dp[i-2]$.

```python
def dp_fib(n):
    if n < 2: return n

    # 1. Initialize Table
    dp = [0] * (n + 1)

    # 2. Base Cases
    dp[0] = 0
    dp[1] = 1

    # 3. Iteration (Topological Order)
    for i in range(2, n + 1):
        dp[i] = dp[i-1] + dp[i-2]

    return dp[n]
```

Optimization Note: Since $dp[i]$ only depends on the previous two values, we don't need to store the whole array. We can just keep two variables, reducing Space Complexity to $O(1)$.

13.4.3 Case Study: Binomial Coefficient

State: $dp[n][k]$ is the value of $\binom{n}{k}$.
Transition: $dp[i][j] = dp[i-1][j-1] + dp[i-1][j]$ (Pascal's Identity).

We iterate row-by-row, essentially building Pascal's Triangle.

```python
def binomial_bottom_up(n, k):
    # dp table of size (n+1) x (k+1)
    dp = [[0 for _ in range(k + 1)] for _ in range(n + 1)]

    for i in range(n + 1):
        # We can't choose more items 'j' than available 'i'
        for j in range(min(i, k) + 1):
            # Base Cases: Choose 0 or Choose All = 1
            if j == 0 or j == i:
                dp[i][j] = 1
            else:
                dp[i][j] = dp[i-1][j-1] + dp[i-1][j]

    return dp[n][k]
```

Complexity:

- **Time:** $O(n \cdot k)$ — We fill a grid of size roughly $n \times k$.
- **Space:** $O(n \cdot k)$ — Size of the 2D grid. (Can be optimized to $O(k)$).

13.4.4 Case Study: Coin Change

State: $dp[a]$ is the minimum coins to make amount a.
Transition: $dp[a] = 1 + \min(dp[a-c])$ for all coins c.

In the recursive version, we started at the target amount and subtracted coins. Here, we start at 0 and build up. This guarantees that when we solve for amount a, the solution for $a - c$ is already computed.

```python
def coin_change_bottom_up(coins, amount):
    # Initialize with a value > amount (as infinity)
    dp = [float("inf")] * (amount + 1)

    # Base Case: 0 coins needed to make amount 0
    dp[0] = 0

    # Loop through every amount from 1 to Target
    for a in range(1, amount + 1):
        for c in coins:
            if a - c >= 0:
                # Can we make this amount with fewer coins
                # by using coin 'c'?
                dp[a] = min(dp[a], dp[a - c] + 1)

    return dp[amount] if dp[amount] != float("inf") else -1
```

Complexity:

- **Time:** $O(A \cdot C)$ — Outer loop runs A times, inner loop runs C times.
- **Space:** $O(A)$ — We only need a 1D array of size $A + 1$. This is highly efficient compared to the recursion stack of the Top-Down approach.

13.5 Greedy vs Dynamic Programming

Greedy algorithms and DP are both strategies for optimization problems, but they rely on different correctness principles.

- **Greedy:** build a solution step-by-step by repeatedly taking the locally best-looking choice.
- **DP:** consider all relevant subproblems and combine their optimal solutions to guarantee a global optimum.

Greedy methods are correct when the problem has the **greedy-choice property**: there exists an optimal solution that begins with a locally optimal choice. Many problems satisfy this (e.g. interval scheduling, minimum spanning tree, Huffman coding), but many do not (e.g. 0/1 knapsack, general coin change, rod cutting below).

Greedy	**Dynamic Programming**
Local decisions: choose best option now.	**Global planning:** solve subproblems and combine optimally.
Correctness requires: greedy-choice property (plus usually optimal substructure).	**Correctness requires:** optimal substructure + overlapping subproblems.
Often very fast ($O(n)$ or $O(n \log n)$).	Often polynomial but heavier (e.g. $O(n^2)$, $O(nC)$, $O(nW)$).
May fail badly if local choice blocks better global structure.	Guaranteed optimum if recurrence/state are correct and full dependency order is respected.

Exchange Argument

To prove a greedy strategy is optimal, we often use the **Exchange Argument**. This is a proof by contradiction (or transformation).

1. **Assumption:** Assume there exists an optimal solution $\mathcal{O}$ that is *different* from the Greedy solution $\mathcal{G}$.
2. **Identify Divergence:** Find the first step where $\mathcal{O}$ makes a choice A, but the Greedy algorithm makes a different choice B.
3. **Exchange:** Show that we can swap A for B in $\mathcal{O}$ without making the total value worse (i.e., the new solution $\mathcal{O}'$ is still optimal).
4. **Conclusion:** By repeating this process, we can transform $\mathcal{O}$ into $\mathcal{G}$ while maintaining optimality. Therefore, the Greedy solution $\mathcal{G}$ must be optimal.

Let us look at some cases where greedy can fail:

Counterexample 1: Coin change

Coins $\{1, 3, 4\}$, amount 6:

- Greedy by largest coin: take 4, then 1, then 1 $\Rightarrow$ 3 coins.
- Optimal: $3 + 3 \Rightarrow 2$ coins.

This fails because the locally best coin (4) produces a poor remainder (2).

Counterexample 2: Rod cutting (density heuristic fails)

Greedy heuristics like "best price per unit length" can fail because the best first cut may leave a remainder that is expensive to complete. DP succeeds by exploring *all* first cuts and combining them with optimal remainders (optimal substructure).

13.5.1 Rod Cutting: DP vs. Greedy

Given a rod of length n and a list of prices P where $P[i]$ is the price of a piece of length i, determine the maximum revenue obtainable.

Method 1: Dynamic Programming (Optimal)

This approach works for **any** pricing scheme. We build a table where we solve for every rod length from 1 up to n.

- **State:** `dp[i]` represents the maximum value for a rod of length i.
- **Transition:** To solve for length i, we consider making a first cut at length j (where $1 \leq j \leq i$). The revenue is the price of that cut $P[j]$ plus the optimal revenue for the remaining length $dp[i-j]$.

$$dp[i] = \max_{1 \leq j \leq i} (P[j] + dp[i-j])$$

```python
def rod_cutting_dp(prices, n):
    # dp[i] will store max profit for rod of length i
    dp = [0] * (n + 1)
    # Solve for every length from 1 to n (Bottom-Up)
    for i in range(1, n + 1):
        max_val = -float("inf")
        # Try every possible first cut 'j'
        for j in range(1, i + 1):
            # Price of cut j + Best way to sell remainder (i-j)
            current_val = prices[j] + dp[i - j]
            if current_val > max_val:
                max_val = current_val
        dp[i] = max_val
    return dp[n]
```

Complexity: Time $O(n^2)$, Space $O(n)$.

Method 2: Greedy Strategy (Suboptimal)

A common heuristic is to calculate the **density** (price per unit length) and always prioritize the cut with the highest density.

```python
def rod_cutting_greedy(prices, n):
    # 1. Calculate density for each cut: (price / length)
    # Store as tuple: (density, length)
    densities = []
    for length in range(1, len(prices)):
        densities.append((prices[length] / length, length))

    # 2. Sort so highest density is first
    densities.sort(reverse=True, key=lambda x: x[0])

    total_value = 0
    remaining_len = n

    # 3. Greedily take as many high-density cuts as possible
    for _, length in densities:
        if remaining_len == 0: break

        # How many of this cut can we fit?
        count = remaining_len // length

        total_value += count * prices[length]
        remaining_len -= count * length

    return total_value
```

Greedy fails in this case. Consider a rod of length **4** with the following prices:

- Length 1: \$1 (Density 1.0)
- Length 2: \$5 (Density 2.5)
- Length 3: \$8 (Density **2.67**) $\leftarrow$ Highest Density!
- Length 4: \$9 (Density 2.25)

1. **Greedy Choice:** Selects Length 3 first (best density).
   - Remaining length: 1. Must choose Length 1.
   - Total: $\$8 + \$1 = \$\mathbf{9}$.
2. **DP Choice:** Sees that $2\times$ Length 2 is better.
   - Total: $\$5 + \$5 = \$\mathbf{10}$.

Key Takeaway

Greedy is fast ($O(n \log n)$ due to sorting), but it fails because rod cutting lacks the **greedy choice property**. A locally optimal choice (best density) can leave a remainder (e.g. length 1) that is extremely inefficient to use, dragging down the global score. DP guarantees the optimum by checking all combinations.

13.6 The 0/1 Knapsack Problem

The $0/1$ Knapsack problem is arguably the most important pattern in Dynamic Programming. It introduces the fundamental decision paradigm found in countless optimization problems: **constraint-based choice**.

Given n items, each with a weight w_i and a value v_i, and a knapsack with limited weight capacity C, we must decide for each item: *do we include it or exclude it?*

- **0/1 Constraint:** Each item is unique. You can take it **once** (1) or **not at all** (0).
- **Goal:** Maximize total value $\sum v_i$ such that total weight $\sum w_i \leq C$.

13.6.1 DP State and Recurrence

We define our state to track two changing variables: the items we are considering and the remaining capacity.

- **State:** `dp[i][w]` = Maximum value using a subset of the first i items with weight limit w.

For every item i (with weight wt and value val), we have two choices:

1. **Exclude (Skip):** We don't take the item. The value is the same as the solution for $i-1$ items with the same capacity w.
2. **Include (Take):** We take the item (if $wt \leq w$). The value is val plus the optimal solution for the *remaining* capacity $(w - wt)$ using previous items.

$$dp[i][w] = \max \begin{cases} dp[i-1][w] & \text{(Skip)} \\ val + dp[i-1][w-wt] & \text{(Take, if } wt \leq w\text{)} \end{cases}$$

```python
def solve_knapsack(weights, values, capacity):
    n = len(weights)
    # dp[i][w] matrix
    dp = [[0 for _ in range(capacity + 1)] for _ in range(n + 1)]
    for i in range(1, n + 1):
        wt = weights[i - 1]
        val = values[i - 1]
        for w in range(capacity + 1):
            # Option 1: Skip this item
            skip_val = dp[i - 1][w]

            # Option 2: Take this item (if it fits)
            if wt <= w:
                take_val = val + dp[i - 1][w - wt]
                dp[i][w] = max(skip_val, take_val)
            else:
                dp[i][w] = skip_val

    return dp[n][capacity]
```

13.6.2 Visualizing the Decision Process

Consider:

- Item 1: (wt=1, val=15)
- Item 2: (wt=3, val=20)
- Capacity: 4

When calculating the cell for **item 2** at **capacity 4**, we compare:

1. **Skip item 2:** Inherit value from item 1 at capacity 4 (15).
2. **Take item 2:** Gain 20 value + look at remainder (capacity $4 - 3 = 1$) from the item 1 row (15). Total value is $20 + 15 = 35$.

Since $35 > 15$, we take the item.

	w=0				w=4
Base Case (0)	0	0	0	0	0
Item 1 (1,15)	0	15	15	15	15
Item 2 (3,20)	0	15	15	20	**35**

Figure 13.3: The DP table deciding to take Item 2.

13.6.3 0/1 vs. Unbounded Knapsack (Rod Cutting)

This pattern directly relates to other famous problems. The key difference lies in whether we can reuse the current item.

0/1 Knapsack	**Rod Cutting (Unbounded)**
Each item can be used **once**.	Items (lengths) can be used **repeatedly**.
When we take item i, we look at the solution for $i - 1$ (previous row) for the remainder.	When we take cut length i, we stay on row i (or the current 1D array) because we might use length i again.
$dp[i][w] = \max(\ldots, val + dp[\mathbf{i-1}][w - wt])$	$dp[w] = \max(\ldots, price + dp[w - len])$

13.6.4 Complexity

- **Time:** $O(n \cdot C)$. We fill a table of size items $\times$ capacity.
- **Space:** $O(n \cdot C)$.

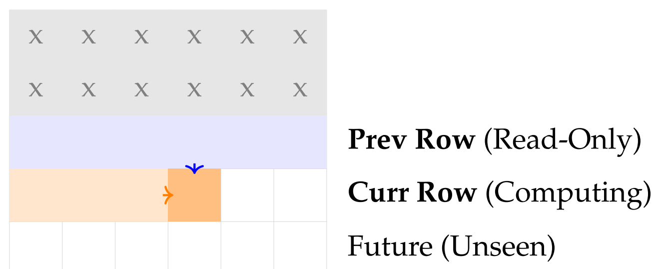
13.7 Out of Syllabus: Space Optimization (Rolling Array)

One of the major downsides of Dynamic Programming is memory usage. A standard 2D DP approach for a problem involving N items and capacity W often requires $O(N \cdot W)$ space. For large inputs, this causes **Memory Limit Exceeded (MLE)** errors.

The *Rolling Array* technique reduces space complexity by observing that we rarely need the *entire* history of the DP table.

In most DP recurrences, the state at index i only depends on a fixed window of previous states (e.g. $i-1$ and $i-2$).

- **Observation:** Once we compute state i, we no longer need state $i-2$.
- **Strategy:** discard old states to free up memory.



Rolling Array: We only keep the blue row in memory to compute the orange row. The gray rows are discarded.

Figure 13.4: Visualizing Space Optimization in 2D DP.

13.7.1 1D Optimization: Fibonacci

Standard ($O(N)$ Space): Stores an array `dp[0...n]`.

```
dp = [0] * (n + 1)
for i in range(2, n + 1):
    dp[i] = dp[i-1] + dp[i-2] # Depends on last 2
```

Optimized ($O(1)$ Space): We only track two variables.

```
prev2, prev1 = 0, 1
for i in range(2, n + 1):
    curr = prev1 + prev2
    prev2 = prev1  # Shift window
    prev1 = curr
return prev1
```

13.7.2 2D Optimization: Rolling Array

Consider the Minimum Path Sum problem on a grid.

Minimum Path Sum (Leetcode 64)

Given a $m \times n$ grid filled with non-negative numbers, find a path from top left to bottom right, which minimizes the sum of all numbers along its path.
Note: You can only move either down or right at any point in time.

$$dp[i][j] = \text{grid}[i][j] + \min(dp[i-1][j], dp[i][j-1])$$

Notice that to calculate row i, we only need data from row i (left neighbor) and row $i-1$ (top neighbor). We never look at row $i-2$.

We can reduce space from $O(M \cdot N)$ to $O(N)$ (one row).

```python
def min_path_sum_optimized(grid):
    rows, cols = len(grid), len(grid[0])
    # Create a 1D array representing the "previous row"
    # Initialize with first row of grid (accumulated)
    dp = [0] * cols
    dp[0] = grid[0][0]
    for j in range(1, cols):
        dp[j] = dp[j-1] + grid[0][j]

    for i in range(1, rows):
        # Update first column of current row
        dp[0] += grid[i][0]

        for j in range(1, cols):
            # dp[j] currently holds value from row i-1 (top)
            # dp[j-1] currently holds value from row i (left)
            dp[j] = grid[i][j] + min(dp[j], dp[j-1])

    return dp[cols-1]
```

We can take the optimization one step further. Notice that once we compute the minimum path to cell (i, j), we strictly **never need the original value** of `grid[i][j]` again.

Instead of allocating a separate DP array, we simply overwrite the input grid with the cumulative sums as we go.

```python
def min_path_sum_optimized(grid):
    rows = len(grid)
    cols = len(grid[0])
    # 1. Initialize first row (can only come from left)
    for j in range(1, cols):
        grid[0][j] += grid[0][j-1]
    # 2. Initialize first column (can only come from top)
    for i in range(1, rows):
        grid[i][0] += grid[i-1][0]
    # 3. Fill the rest
    for i in range(1, rows):
        for j in range(1, cols):
            # Overwrite current cell with:
            # Original Cost + min(Top Neighbor, Left Neighbor)
            grid[i][j] += min(grid[i-1][j], grid[i][j-1])
    # The bottom-right cell now holds the answer
    return grid[-1][-1]
```

Destructive Algorithms

While this achieves $O(1)$ auxiliary space, it is a **destructive algorithm**. It permanently alters the input data.

- **Safe:** In pure algorithmic problems where the input is not reused.
- **Unsafe:** In software engineering where the caller might expect the 'grid' to remain unchanged for other functions.

Looking Forward: Dynamic Programming Tricks

Note that we have barely scratched the surface about this idea. There are many advanced tricks to be learnt with respect to dynamic programming, such as:

- Dynamic Programming optimizations based on the structure of the recurrence, such as convex hull optimizations, divide and conquer optimizations and Knuth optimizations **(seen in CS2040S/CS3230/CS3233)**.
- Matrix exponentiation to solve recurrences **(seen in MA1522/MA2001)**.
- Speeding up Dynamic Programming solutions using data structures like sliding deques and segment trees **(seen in CS3230/CS3233)**.
- Variants of Dynamic Programming, such as Bitmask DP, Digit DP and Lexicographical DP **(seen in CS3233)**.
- Speeding up Dynamic Programming based on constraints of the input, using a technique known as state shuffling **(seen in CS3233 (2017, knapsack_ex))**.
- Binary search to skip states by making use of monotonicity ("Aliens' Trick") **(seen in CS3233)**.

13.8 Exception Handling

We end off this part with a quick note on exception handling.

Exception handling is a mechanism that allows a program to manage errors gracefully during execution, preventing abrupt crashes. In Python, errors are generally classified into two categories: **Syntax Errors** and **Exceptions**.

13.8.1 Types of Errors

- **Syntax Errors:** These are errors detected before execution when the code violates the grammatical rules of the language (e.g., missing colons or parentheses).
- **Exceptions:** These are errors detected *during* execution, even if the syntax is correct. Common examples include:
  - `ZeroDivisionError`: Raised when attempting to divide by zero.
  - `NameError`: Raised when a local or global name is not found (e.g., using an undefined variable).
  - `TypeError`: Raised when an operation is applied to an object of an inappropriate type (e.g., adding a string to an integer).

13.8.2 The `try-except` Block

The simplest way to handle exceptions is using the `try-except` block. The execution flow is as follows:

1. The `try` clause is executed first.
2. If an exception occurs, the rest of the `try` clause is skipped, and control jumps to the matching `except` clause.
3. If no exception occurs, the `except` clause is skipped entirely.

A single `try` statement may have multiple `except` clauses to handle different types of exceptions specifically. At most one handler will be executed.

```python
def divide_test(x, y):
    try:
        result = x / y
    except ZeroDivisionError:
        print("division by zero!")
    except TypeError:
        print("Invalid types!")
```

If an exception is raised but not caught in the current scope, it propagates up the **call stack** to the calling function. This process continues until a matching handler is found. If the exception reaches the top level of the program without being caught, the interpreter terminates the program and displays a traceback.

13.8.3 The `else` and `finally` Clauses

Python allows optional clauses to handle specific execution paths:

- **`else`:** Executed only if **no exception** occurs in the `try` block.
- **`finally`:** Executed **always**, regardless of whether an exception occurred or not. This is typically used for cleanup actions (e.g., closing files).

```
def divide_test(x, y):
    try:
        result = x / y
    except ZeroDivisionError:
        print("division by zero!")
    else:
        print("result is", result)
    finally:
        print("executing finally clause")
```

13.8.4 Raising Exceptions

The `raise` statement allows the programmer to force a specific exception to occur.

```
raise NameError('HiThere')
'''
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: HiThere
'''
```

13.8.5 User-Defined Exceptions

Programmers can define their own exceptions by creating a new class that inherits from the built-in `Exception` class.

```
class MyError(Exception):
    def __init__(self, value):
        self.value = value
    def __str__(self):
        return repr(self.value)

try:
    raise MyError(2*2)
except MyError as e:
    print('Exception value:', e.value)
```

13.8.6 Why use Exceptions?

Compared to older error-handling methods (like returning special integers such as -1 in C), exceptions are considered superior because they are more natural, easily extensible, and allow for flexible handling via nested structures.