Lab Session 5

Dynamic Programming, Greedy & Amortized Analysis

The Art of Computer Programming 2 — Week 5 — March 20, 2026

Objectives

By the end of this lab, you should be able to:

- Fill DP tables by hand (rod cutting, LCS) and trace back optimal solutions
- Implement top-down (memoized) and bottom-up DP algorithms in Python
- Reconstruct an optimal solution from a DP table
- Implement greedy algorithms (activity selection, Huffman coding)
- Measure and verify amortized costs empirically (dynamic array)
- Compare naïve recursive, memoized, and bottom-up DP performance

1 Warm-up: DP Tables by Hand (15 min)

Pen and paper — no computer needed.

1.1 Rod cutting (CLRS 14.1)

Recall

Rod cutting: given a rod of length n and a price table p_i for each length i, find cuts to maximise revenue. Recurrence:

$$r_n = \max_{1 \leqslant i \leqslant n} \{p_i + r_{n-i}\}, \qquad r_0 = 0$$

Consider the price table:

length i	1	2	3	4	5	6	7	8
price p_i	1	5	8	9	10	17	17	20

(a) Fill the table $r_0, r_1, \ldots, r_8$ by computing each entry bottom-up.

(b) For each r_j, record the **first cut** s_j (the value of i achieving the max).

(c) Using s, trace back the optimal cuts for $n = 8$.

(d) How many subproblems are solved? What is the time complexity?

1.2 Longest Common Subsequence (CLRS 14.4)

Recall

LCS: given two sequences $X = \langle x_1, \ldots, x_m \rangle$ and $Y = \langle y_1, \ldots, y_n \rangle$, find the longest subsequence common to both.

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ c[i{-}1, j{-}1] + 1 & \text{if } x_i = y_j, \\ \max(c[i{-}1, j],\ c[i, j{-}1]) & \text{otherwise.} \end{cases}$$

Let $X = \langle A, B, C, B, D, A, B \rangle$ and $Y = \langle B, D, C, A, B, A \rangle$.

(a) Fill the complete $c[i, j]$ table (an 8×7 grid, including the row/column of 0's).

(b) What is the length of the LCS?

(c) Trace back from $c[7, 6]$ to find an LCS. At each cell, indicate which arrow you follow ($\nwarrow$, $\uparrow$, or $\leftarrow$).

(d) Is the LCS unique? Can you find another one?

Hint

At each cell, $\nwarrow$ if $x_i = y_j$ (include this character), $\uparrow$ if $c[i-1, j] \geqslant c[i, j-1]$, $\leftarrow$ otherwise.

2 Implement Rod Cutting (25 min)

2.1 Naïve recursive solution

```python
def cut_rod_naive(p, n):
    """Naive recursive rod cutting. Returns max revenue."""
    if n == 0:
        return 0
    q = -1
    for i in range(1, n + 1):
        q = max(q, p[i] + cut_rod_naive(p, n - i))
    return q
```

2.2 Top-down with memoization

```python
def cut_rod_memo(p, n):
    """Top-down memoized rod cutting. Returns max revenue."""
    memo = [-1] * (n + 1)

    def helper(j):
        if memo[j] >= 0:
            return memo[j]
        if j == 0:
            q = 0
        else:
            q = -1
            for i in range(1, j + 1):
                q = max(q, p[i] + helper(j - i))
        memo[j] = q
        return q

    return helper(n)
```

2.3 Bottom-up with solution reconstruction

Implement the bottom-up version that also records the optimal first cut $s[j]$ for each length j, then reconstructs the full solution.

```python
def cut_rod_bottom_up(p, n):
    """Bottom-up rod cutting. Returns (max revenue, list of cuts)."""
```

```
    r = [0] * (n + 1)   # r[j] = optimal revenue for rod of length j
    s = [0] * (n + 1)   # s[j] = optimal first cut for length j

    for j in range(1, n + 1):
        q = -1
        for i in range(1, j + 1):
            if p[i] + r[j - i] > q:
                q = p[i] + r[j - i]
                s[j] = i
        r[j] = q

    # Reconstruct the cuts
    cuts = []
    # YOUR CODE HERE: use s[] to find all cuts

    return r[n], cuts
```

Hint

To reconstruct: start with n, append $s[n]$, then $n \leftarrow n - s[n]$. Repeat until $n = 0$.

2.4 Test your implementations

```
prices = {1: 1, 2: 5, 3: 8, 4: 9, 5: 10,
          6: 17, 7: 17, 8: 20, 9: 24, 10: 30}

# Check that all three versions agree
for n in range(1, 11):
    r_naive = cut_rod_naive(prices, n)
    r_memo = cut_rod_memo(prices, n)
    r_bu, cuts = cut_rod_bottom_up(prices, n)
    assert r_naive == r_memo == r_bu, (
        f"Mismatch at n={n}: {r_naive}, {r_memo}, {r_bu}")
    # Verify the cuts sum to n and revenue matches
    assert sum(cuts) == n
    assert sum(prices[c] for c in cuts) == r_bu
    print(f"n={n:>2}: revenue={r_bu:>2}, cuts={cuts}")

print("All rod cutting tests passed!")
```

3 Longest Common Subsequence (15 min)

3.1 LCS length — bottom-up

```
def lcs_length(X, Y):
    """Compute the LCS table. Returns (c, b) where
    c[i][j] = length of LCS of X[:i] and Y[:j],
    b[i][j] = direction for traceback."""
    m, n = len(X), len(Y)
    c = [[0] * (n + 1) for _ in range(m + 1)]
    b = [[""] * (n + 1) for _ in range(m + 1)]

    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if X[i - 1] == Y[j - 1]:
```

```
                c[i][j] = c[i - 1][j - 1] + 1
                b[i][j] = "diag"   # match
            elif c[i - 1][j] >= c[i][j - 1]:
                c[i][j] = c[i - 1][j]
                b[i][j] = "up"
            else:
                c[i][j] = c[i][j - 1]
                b[i][j] = "left"
    return c, b
```

3.2 LCS reconstruction

```
def lcs_reconstruct(b, X, i, j):
    """Reconstruct one LCS from the b table."""
    # YOUR CODE HERE
    # Follow b[i][j]: if "diag", include X[i-1] and recurse (i-1,j-1)
    #                 if "up", recurse (i-1,j)
    #                 if "left", recurse (i,j-1)
    # Base case: i == 0 or j == 0 -> return ""
    pass
```

3.3 Test your implementation

```
X = list("ABCBDAB")
Y = list("BDCABA")

c, b = lcs_length(X, Y)

# Print the DP table
print("    ", "  ".join(["-"] + Y))
for i in range(len(X) + 1):
    row_label = "-" if i == 0 else X[i - 1]
    print(f" {row_label} ", " ".join(f"{c[i][j]:2d}" for j in range(len(Y) + 1)))

lcs = lcs_reconstruct(b, X, len(X), len(Y))
print(f"\nLCS length: {c[len(X)][len(Y)]}")
print(f"LCS: {lcs}")

# Expected: length 4, one LCS is "BCBA"
assert c[len(X)][len(Y)] == 4
assert len(lcs) == 4

# Verify it is indeed a subsequence of both X and Y
def is_subsequence(s, t):
    it = iter(t)
    return all(c in it for c in s)

assert is_subsequence(lcs, X)
assert is_subsequence(lcs, Y)
print("All LCS tests passed!")
```

4 Greedy Algorithms 💻 (15 min)

4.1 Activity selection (CLRS 15.1)

Recall

Activity selection: Given n activities with start/finish times (s_i, f_i), select the maximum number of mutually compatible activities.
Greedy strategy: always pick the activity with the **earliest finish time** that is compatible with the last selected activity.

```python
def activity_selection(activities):
    """Greedy activity selection.
    activities: list of (start, finish) tuples.
    Returns list of selected activities (sorted by finish time).
    """
    # YOUR CODE HERE
    # 1. Sort by finish time
    # 2. Select the first activity
    # 3. For each remaining activity, select it if its start >= last finish
    pass
```

```python
# Test
activities = [(1, 4), (3, 5), (0, 6), (5, 7), (3, 9),
              (5, 9), (6, 10), (8, 11), (8, 12), (2, 14), (12, 16)]

selected = activity_selection(activities)
print(f"Selected {len(selected)} activities: {selected}")

# Verify: no two selected activities overlap
for i in range(len(selected) - 1):
    assert selected[i][1] <= selected[i + 1][0], (
        f"Overlap: {selected[i]} and {selected[i+1]}")

assert len(selected) == 4
print("Activity selection test passed!")
```

4.2 Huffman coding (CLRS 15.3)

Recall

Huffman coding: build an optimal prefix-free binary code by repeatedly merging the two lowest-frequency nodes. Uses a min-priority queue.

```python
import heapq

class HuffmanNode:
    def __init__(self, char=None, freq=0, left=None, right=None):
        self.char = char
        self.freq = freq
        self.left = left
        self.right = right

    def __lt__(self, other):
        return self.freq < other.freq
```

```python
def huffman_build(freq_table):
    """Build a Huffman tree from a frequency table (dict).
    Returns the root of the Huffman tree."""
    # YOUR CODE HERE
    # 1. Create a leaf node for each character
    # 2. Insert all nodes into a min-heap
    # 3. While heap has more than 1 node:
    #    - Extract the two smallest
    #    - Create a new internal node with these two as children
    #    - Insert the new node back
    # 4. Return the remaining node (root)
    pass


def huffman_codes(root, prefix="", codes=None):
    """Extract codes by traversing the Huffman tree.
    Returns a dict mapping each character to its binary string."""
    if codes is None:
        codes = {}
    # YOUR CODE HERE
    # If leaf (root.char is not None): codes[root.char] = prefix
    # Else: recurse left with prefix+"0", right with prefix+"1"
    return codes
```

```python
# Test
frequencies = {"A": 15, "B": 7, "C": 6, "D": 6, "E": 5}
root = huffman_build(frequencies)
codes = huffman_codes(root)

print("Character | Freq | Code")
print("-" * 30)
for char in sorted(frequencies):
    print(f"    {char}     | {frequencies[char]:>4} | {codes[char]}")

# Verify prefix-free property
for c1 in codes:
    for c2 in codes:
        if c1 != c2:
            assert not codes[c1].startswith(codes[c2]), (
                f"Not prefix-free: {c1}={codes[c1]}, {c2}={codes[c2]}")

# Compute average bits per character
total = sum(frequencies.values())
avg_bits = sum(frequencies[c] * len(codes[c])
               for c in frequencies) / total
print(f"\nAverage bits/char: {avg_bits:.2f}")
print("Huffman test passed!")
```

5 Amortized Analysis: Dynamic Array (10 min)

Recall

A **dynamic array** doubles its capacity when full. A single resize costs $\Theta(n)$, but the **amortized cost per `push_back`** is $\mathcal{O}(1)$.

- **Aggregate:** total cost of n push_backs $\leqslant 3n \Rightarrow$ amortized $= 3$.
- **Accounting:** charge \$3 per push_back (\$1 insert + \$2 credit for future copy).

- **Potential:** $\Phi = 2 \cdot num - cap$; amortized cost $= 3$ always.

5.1 Implement a dynamic array

```python
class DynamicArray:
    """A dynamic array that doubles capacity on overflow."""

    def __init__(self):
        self.capacity = 1
        self.size = 0
        self.data = [None] * self.capacity
        self.total_ops = 0      # total element writes
        self.num_resizes = 0    # number of resize events

    def push_back(self, value):
        """Append a value. Resize if necessary."""
        if self.size == self.capacity:
            self._resize(2 * self.capacity)
        self.data[self.size] = value
        self.size += 1
        self.total_ops += 1     # count the insertion itself

    def _resize(self, new_capacity):
        """Resize internal storage."""
        new_data = [None] * new_capacity
        for i in range(self.size):
            new_data[i] = self.data[i]
            self.total_ops += 1  # count each copy
        self.data = new_data
        self.capacity = new_capacity
        self.num_resizes += 1

    def __len__(self):
        return self.size

    def __getitem__(self, i):
        if 0 <= i < self.size:
            return self.data[i]
        raise IndexError(f"Index {i} out of range")
```

5.2 Verify the amortized cost

```python
import matplotlib.pyplot as plt

ns = list(range(1, 10001))
ops_per_n = []
da = DynamicArray()

for n in ns:
    da.push_back(n)
    ops_per_n.append(da.total_ops / da.size)

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 4))

# Left: amortized cost per operation
```

```
ax1.plot(ns, ops_per_n, '-', color='#0055A4', linewidth=0.5)
ax1.axhline(y=3, color='#E74C3C', linestyle='--', label='Theoretical bound (3)')
ax1.set_xlabel('n (number of push_back calls)', fontweight='bold')
ax1.set_ylabel('Average cost per operation', fontweight='bold')
ax1.set_title('Amortized Cost of Dynamic Array')
ax1.legend()
ax1.set_ylim(0, 5)
ax1.spines['top'].set_visible(False)
ax1.spines['right'].set_visible(False)

# Right: total cost vs 3n
total_ops_list = []
da2 = DynamicArray()
for n in ns:
    da2.push_back(n)
    total_ops_list.append(da2.total_ops)

ax2.plot(ns, total_ops_list, '-', color='#0055A4', label='Actual total cost')
ax2.plot(ns, [3 * n for n in ns], '--', color='#E74C3C', label='3n (upper bound)')
ax2.plot(ns, ns, '--', color='#27AE60', label='n (lower bound)')
ax2.set_xlabel('n', fontweight='bold')
ax2.set_ylabel('Total operations', fontweight='bold')
ax2.set_title('Total Cost vs Bounds')
ax2.legend()
ax2.spines['top'].set_visible(False)
ax2.spines['right'].set_visible(False)

plt.tight_layout()
plt.show()

print(f"After {da2.size} push_backs:")
print(f"  Total ops:   {da2.total_ops}")
print(f"  Amortized:   {da2.total_ops / da2.size:.2f}")
print(f"  Resizes:     {da2.num_resizes}")
print(f"  Capacity:    {da2.capacity}")
```

Question: Does the measured amortized cost stay below 3? What happens to the ratio as n grows?

6 Performance Comparison: Naïve vs. Memoized vs. Bottom-Up 💻 (10 min)

6.1 Timing the three approaches

```
import time
import sys
sys.setrecursionlimit(10000)

prices = {i: i for i in range(1, 501)}  # simple prices
sizes = [5, 10, 15, 20, 25, 28]

print(f"{'n':>4} | {'Naive':>12} | {'Memoized':>12} | {'Bottom-up':>12}")
print("-" * 50)

for n in sizes:
    # Naive
    start = time.perf_counter()
```

```
    cut_rod_naive(prices, n)
    t_naive = time.perf_counter() - start

    # Memoized
    start = time.perf_counter()
    cut_rod_memo(prices, n)
    t_memo = time.perf_counter() - start

    # Bottom-up
    start = time.perf_counter()
    cut_rod_bottom_up(prices, n)
    t_bu = time.perf_counter() - start

    print(f"{n:>4} | {t_naive:>12.6f} | {t_memo:>12.6f} | {t_bu:>12.6f}")
```

Important

The naïve version has exponential time complexity $\mathcal{O}(2^n)$. Don't run it for $n > 30$ unless you want to wait a very long time!

6.2 Plot the running times

```
# Only time memoized vs bottom-up for larger n
sizes_large = list(range(10, 501, 10))
times_memo = []
times_bu = []

for n in sizes_large:
    start = time.perf_counter()
    cut_rod_memo(prices, n)
    times_memo.append(time.perf_counter() - start)

    start = time.perf_counter()
    cut_rod_bottom_up(prices, n)
    times_bu.append(time.perf_counter() - start)

fig, ax = plt.subplots(figsize=(8, 4))
ax.plot(sizes_large, times_memo, '-o', color='#0055A4',
        markersize=3, label='Top-down (memoized)')
ax.plot(sizes_large, times_bu, '-s', color='#27AE60',
        markersize=3, label='Bottom-up')
ax.set_xlabel('n (rod length)', fontweight='bold')
ax.set_ylabel('Time (seconds)', fontweight='bold')
ax.set_title('Memoized vs Bottom-Up Rod Cutting')
ax.legend()
ax.spines['top'].set_visible(False)
ax.spines['right'].set_visible(False)
plt.tight_layout()
plt.show()
```

Questions:

(a) Both are $\Theta(n^2)$ in theory. Do you observe a difference in practice?

(b) Why might the bottom-up version be faster, even though both solve the same n subproblems?

Bonus Exercises

For students who finish early.

Bonus 1 — Matrix Chain Multiplication

Implement the matrix chain multiplication algorithm (CLRS 14.2). Given a sequence of matrix dimensions $\langle p_0, p_1, \ldots, p_n \rangle$ where matrix A_i has dimensions $p_{i-1} \times p_i$:

1. Compute the minimum number of scalar multiplications $m[1, n]$.
2. Reconstruct the optimal parenthesization using the s table.

```python
def matrix_chain_order(p):
    """p: list of dimensions [p0, p1, ..., pn].
    Returns (m, s) where m[i][j] = min multiplications,
    s[i][j] = split point."""
    # YOUR CODE HERE
    pass

def print_optimal_parens(s, i, j):
    """Print the optimal parenthesization."""
    # YOUR CODE HERE
    pass

# Test: CLRS example p = [30, 35, 15, 5, 10, 20, 25]
# Expected: m[1][6] = 15125
```

Bonus 2 — Edit Distance (Levenshtein)

The **edit distance** between two strings is the minimum number of single-character insertions, deletions, and substitutions to transform one into the other.

```python
def edit_distance(s1, s2):
    """Compute the edit distance between s1 and s2."""
    # YOUR CODE HERE
    # Hint: similar structure to LCS, but the recurrence is:
    # d[i][j] = d[i-1][j-1] if s1[i-1] == s2[j-1]
    #         = 1 + min(d[i-1][j],      # delete
    #                   d[i][j-1],      # insert
    #                   d[i-1][j-1])    # substitute
    pass

# Test: edit_distance("kitten", "sitting") == 3
# Test: edit_distance("saturday", "sunday") == 3
```

Bonus 3 — Queue with Two Stacks (Amortized) ⋆

Implement a FIFO queue using two stacks S_1 (input) and S_2 (output).

- ENQUEUE(x): push x onto S_1.
- DEQUEUE: if S_2 is empty, pop all from S_1 and push onto S_2; then pop from S_2.

```python
class QueueTwoStacks:
    def __init__(self):
        self.s1 = []   # input stack
        self.s2 = []   # output stack
        self.total_ops = 0

    def enqueue(self, x):
        # YOUR CODE HERE
        pass

    def dequeue(self):
        # YOUR CODE HERE
        pass

# Verify FIFO behaviour, then measure amortized cost
# of n enqueue + n dequeue operations.
```

Question: Use the **accounting method** to show that each operation costs $\mathcal{O}(1)$ amortized.

Key Formulas

Rod cutting (bottom-up)	$\Theta(n^2)$ time, $\Theta(n)$ space
LCS	$\Theta(mn)$ time, $\Theta(mn)$ space
Matrix chain multiplication	$\mathcal{O}(n^3)$ time, $\Theta(n^2)$ space
Activity selection (greedy)	$\mathcal{O}(n \log n)$ (sorting)
Huffman coding	$\mathcal{O}(n \log n)$ (min-heap)
Dynamic array push_back (amortized)	$\mathcal{O}(1)$ amortized

DP = optimal substructure + overlapping subproblems

Greedy = optimal substructure + greedy-choice property

Next week: Elementary Graph Algorithms (CLRS Ch. 20) — BFS, DFS, and applications.