1 Warm-up: DP Tables by Hand

1.1 Rod cutting (CLRS 14.1)

Solution

(a) Bottom-up computation of $r_0, \ldots, r_8$:

j	0	1	2	3	4	5	6	7	8
r_j	0	1	5	8	10	13	17	18	22
s_j (first cut)	–	1	2	3	2	2	6	1	2

Detailed computation:

- $r_1 = p_1 = 1$, $s_1 = 1$.
- $r_2 = \max(p_1 + r_1,\ p_2 + r_0) = \max(2, 5) = 5$, $s_2 = 2$.
- $r_3 = \max(p_1 + r_2,\ p_2 + r_1,\ p_3) = \max(6, 6, 8) = 8$, $s_3 = 3$.
- $r_4 = \max(p_1{+}r_3,\ p_2{+}r_2,\ p_3{+}r_1,\ p_4) = \max(9, 10, 9, 9) = 10$, $s_4 = 2$.
- $r_5 = \max(p_1{+}r_4,\ p_2{+}r_3,\ p_3{+}r_2,\ p_4{+}r_1,\ p_5) = \max(11, 13, 13, 10, 10) = 13$, $s_5 = 2$.
- $r_6 = \max(\ldots) = 17$, $s_6 = 6$ (sell uncut for price 17).
- $r_7 = \max(\ldots) = 18$, $s_7 = 1$ (cut 1+6: $1 + 17 = 18$).
- $r_8 = \max(\ldots) = 22$, $s_8 = 2$ (cut 2+6: $5 + 17 = 22$).

(b) First-cut table s: given above.

(c) Traceback for $n = 8$: $s_8 = 2 \to$ cut piece of length 2, remaining $= 6$. $s_6 = 6 \to$ sell uncut. Optimal cuts: $\mathbf{2} + \mathbf{6}$, revenue $= 5 + 17 = 22$.

(d) Number of subproblems: $n + 1 = 9$ (for r_0 through r_8). Each takes $\mathcal{O}(j)$ time to compute, giving total $\sum_{j=0}^{n} j = \Theta(n^2)$.

Common Mistakes

- Students forget to consider selling the rod uncut (i.e., $i = j$ in the loop).
- Confusing r_j (optimal revenue) with p_j (price for uncut rod of length j).
- Wrong traceback — students sometimes record a different s_j than the one achieving the max.

1.2 Longest Common Subsequence (CLRS 14.4)

Solution

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

(a) + (b) Complete $c[i, j]$ table:

$c[i,j]$		B	D	C	A	B	A
	0	0	0	0	0	0	0
A	0	0	0	0	**1**	1	1
B	0	**1**	1	1	1	**2**	2
C	0	1	1	**2**	2	2	2
B	0	1	1	2	2	**3**	3
D	0	1	**2**	2	2	3	3
A	0	1	2	2	**3**	3	**4**
B	0	1	2	2	3	**4**	4

LCS length $= c[7,6] = 4$.

(c) Traceback from $c[7,6]$:

- $c[7,6] = 4$: $X_7 = B \neq Y_6 = A$, $c[6,6] = 4 \geqslant c[7,5] = 4 \rightarrow$ go $\uparrow$ to $(6,6)$.
- $c[6,6] = 4$: $X_6 = A = Y_6 = A \rightarrow \nwarrow$, include **A**, go to $(5,5)$.
- $c[5,5] = 3$: $X_5 = D \neq Y_5 = B$, $c[4,5] = 3 \geqslant c[5,4] = 2 \rightarrow$ go $\uparrow$ to $(4,5)$.
- $c[4,5] = 3$: $X_4 = B = Y_5 = B \rightarrow \nwarrow$, include **B**, go to $(3,4)$.
- $c[3,4] = 2$: $X_3 = C \neq Y_4 = A$, $c[2,4] = 1 < c[3,3] = 2 \rightarrow$ go $\leftarrow$ to $(3,3)$.
- $c[3,3] = 2$: $X_3 = C = Y_3 = C \rightarrow \nwarrow$, include **C**, go to $(2,2)$.
- $c[2,2] = 1$: $X_2 = B \neq Y_2 = D$, $c[1,2] = 0 < c[2,1] = 1 \rightarrow$ go $\leftarrow$ to $(2,1)$.
- $c[2,1] = 1$: $X_2 = B = Y_1 = B \rightarrow \nwarrow$, include **B**, go to $(1,0)$.
- $c[1,0] = 0$: done.

Reading in reverse order: LCS $= \langle B, C, B, A \rangle$ (length 4).

(d) The LCS is not unique. Another traceback gives $\langle B, D, A, B \rangle$. The key point where choices differ is when $c[i-1,j] = c[i,j-1]$ (a tie), allowing us to go either $\uparrow$ or $\leftarrow$.

Grading Notes

Full marks require: (1) correct and complete DP table, (2) correct LCS length, (3) a valid traceback with clear indication of which direction is followed at each step.

2 Implement Rod Cutting

2.1 Bottom-up with solution reconstruction

Solution

```python
def cut_rod_bottom_up(p, n):
    """Bottom-up rod cutting. Returns (max revenue, list of cuts)."""
    r = [0] * (n + 1)
    s = [0] * (n + 1)

    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 = []
    remaining = n
    while remaining > 0:
        cuts.append(s[remaining])
        remaining -= s[remaining]

    return r[n], cuts
```

Key insight: the array s records the optimal *first* cut. To get all cuts, we repeatedly apply: append $s[\mathit{remaining}]$ and subtract it from the remaining length.

Common Mistakes

- **Off-by-one in indexing:** the price list is 1-indexed ($p[1]$ through $p[n]$). Students using 0-indexed lists need to be careful.
- **Forgetting to track $s[j]$:** the reconstruction array must be updated inside the inner loop, at the same time as the max is updated.
- **Reconstruction loop:** students sometimes read s incorrectly, e.g., using $s[n]$ to get all cuts at once instead of iteratively consuming the remaining length.

2.2 Tests

Solution

All assertions pass. Output:

```
n= 1: revenue= 1, cuts=[1]
n= 2: revenue= 5, cuts=[2]
n= 3: revenue= 8, cuts=[3]
n= 4: revenue=10, cuts=[2, 2]
n= 5: revenue=13, cuts=[2, 3]
n= 6: revenue=17, cuts=[6]
n= 7: revenue=18, cuts=[1, 6]
n= 8: revenue=22, cuts=[2, 6]
n= 9: revenue=25, cuts=[3, 6]
n=10: revenue=30, cuts=[10]
All rod cutting tests passed!
```

Explanation

Interesting patterns: $n = 6$ and $n = 10$ should be sold uncut. The price table was designed so that $p_6 = 17$ is a "sweet spot" — many optimal solutions include a piece of length 6.

3 Longest Common Subsequence

3.1 LCS reconstruction

Solution

```python
def lcs_reconstruct(b, X, i, j):
    """Reconstruct one LCS from the b table."""
    if i == 0 or j == 0:
        return ""
    if b[i][j] == "diag":
        return lcs_reconstruct(b, X, i - 1, j - 1) + X[i - 1]
    elif b[i][j] == "up":
        return lcs_reconstruct(b, X, i - 1, j)
    else:
        return lcs_reconstruct(b, X, i, j - 1)
```

Alternative iterative version:

```python
def lcs_reconstruct_iter(b, X, m, n):
    """Iterative LCS reconstruction."""
    result = []
    i, j = m, n
    while i > 0 and j > 0:
        if b[i][j] == "diag":
            result.append(X[i - 1])
            i -= 1
            j -= 1
        elif b[i][j] == "up":
            i -= 1
        else:
            j -= 1
    return "".join(reversed(result))
```

3.2 Tests

Solution

Output:

```
      -  B  D  C  A  B  A
  -   0  0  0  0  0  0  0
  A   0  0  0  0  1  1  1
  B   0  1  1  1  1  2  2
  C   0  1  1  2  2  2  2
  B   0  1  1  2  2  3  3
  D   0  1  2  2  2  3  3
  A   0  1  2  2  3  3  4
  B   0  1  2  2  3  4  4

LCS length: 4
LCS: BCBA
All LCS tests passed!
```

The LCS is $\langle B, C, B, A \rangle$ of length 4. Other valid LCS include $\langle B, D, A, B \rangle$.

Common Mistakes

- **Wrong base case:** returning `""` when $i = 0$ or $j = 0$ (correct) vs. when $c[i][j] = 0$ (incorrect — they are not the same!).
- **Appending before recursing:** the character should be appended *after* the recursive call returns (or use reversed order).
- **Stack overflow:** for very long strings, the recursive version may hit Python's recursion limit. The iterative version is preferred.

4 Greedy Algorithms

4.1 Activity selection (CLRS 15.1)

Solution

```python
def activity_selection(activities):
    """Greedy activity selection.
    activities: list of (start, finish) tuples.
    Returns list of selected activities (sorted by finish time).
    """
    # Sort by finish time
    sorted_acts = sorted(activities, key=lambda a: a[1])

    selected = [sorted_acts[0]]
    last_finish = sorted_acts[0][1]

    for i in range(1, len(sorted_acts)):
        if sorted_acts[i][0] >= last_finish:
            selected.append(sorted_acts[i])
            last_finish = sorted_acts[i][1]

    return selected
```

Output:

```
Selected 4 activities: [(1, 4), (5, 7), (8, 11), (12, 16)]
Activity selection test passed!
```

The algorithm selects activities with finish times 4, 7, 11, 16 — the maximum number of compatible activities.

Explanation

Correctness: The greedy choice (earliest finish time) is safe because: if activity a_1 has the earliest finish time among all activities, some optimal solution includes a_1. Proof: take any optimal solution S^*. If $a_1 \in S^*$, done. Otherwise, let a_k be the first activity in S^* (earliest finish among selected). Since $f_1 \leqslant f_k$, replacing a_k by a_1 preserves compatibility.
Time complexity: $\mathcal{O}(n \log n)$ for sorting, $\mathcal{O}(n)$ for the greedy scan. Total: $\mathcal{O}(n \log n)$.

4.2 Huffman coding (CLRS 15.3)

Solution

```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


def huffman_build(freq_table):
    """Build a Huffman tree from a frequency table (dict)."""
    heap = [HuffmanNode(char=c, freq=f) for c, f in freq_table.items()]
    heapq.heapify(heap)

    while len(heap) > 1:
        left = heapq.heappop(heap)
        right = heapq.heappop(heap)
        parent = HuffmanNode(freq=left.freq + right.freq,
                             left=left, right=right)
        heapq.heappush(heap, parent)

    return heap[0]


def huffman_codes(root, prefix="", codes=None):
    """Extract codes by traversing the Huffman tree."""
    if codes is None:
        codes = {}
    if root.char is not None:
        codes[root.char] = prefix if prefix else "0"
        return codes
    if root.left:
        huffman_codes(root.left, prefix + "0", codes)
    if root.right:
        huffman_codes(root.right, prefix + "1", codes)
    return codes
```

Step-by-step tree construction for $\{A{:}15, B{:}7, C{:}6, D{:}6, E{:}5\}$:

1. Heap: $[E(5), C(6), D(6), B(7), A(15)]$.
2. Merge $E(5) + C(6) = EC(11)$. Heap: $[D(6), B(7), EC(11), A(15)]$.
3. Merge $D(6) + B(7) = DB(13)$. Heap: $[EC(11), DB(13), A(15)]$.
4. Merge $EC(11) + DB(13) = ECDB(24)$. Heap: $[A(15), ECDB(24)]$.
5. Merge $A(15) + ECDB(24) = root(39)$.

Resulting codes (may vary depending on tie-breaking):

```
Character | Freq | Code
------------------------------
    A     |   15 | 0
    B     |    7 | 111
    C     |    6 | 101
    D     |    6 | 110
    E     |    5 | 100

Average bits/char: 2.23
Huffman test passed!
```

Note: The exact codes depend on how ties are broken in the heap. Different valid Huffman trees may produce different codes, but all have the same **weighted average code length**.

Common Mistakes

- **Forgetting `__lt__`:** Python's `heapq` needs comparison support. Without `__lt__`, `heappush` fails.
- **Single-character edge case:** if only one character exists, the tree is a single leaf. The code should be "0" (not empty string).
- **Not using a min-heap:** using a max-heap or sorted list instead gives wrong codes.

Grading Notes

Accept any valid Huffman tree. Check: (1) prefix-free property holds, (2) average bits/char is optimal, (3) building process correctly uses a min-heap.

5 Amortized Analysis: Dynamic Array

5.1 Verify the amortized cost

Solution

Typical output:

```
After 10000 push_backs:
  Total ops:   20469
  Amortized:   2.05
  Resizes:     14
  Capacity:    16384
```

Analysis:

- The amortized cost per operation converges to ≈ 2 (below the theoretical bound of 3).
- More precisely: total ops $= n + \sum_{j=0}^{\lfloor \lg n \rfloor} 2^j \approx n + 2n = 3n$ in the worst case.
- For $n = 10000$: $\sum 2^j = 1{+}2{+}4{+}\cdots{+}8192 = 16383$. Total $\leqslant 10000{+}16383 = 26383 < 3{\times}10000$.
- Actual total is lower because the last resize to 16384 copies only ≈ 8192 elements, not all 16383.
- The ratio total_ops/n decreases toward 2 as $n \to \infty$ for this implementation (because capacity exactly doubles and the geometric series sums to $< 2n$).

The plot clearly shows:

1. The amortized cost per operation stays well below 3 and converges.
2. The total cost curve is sandwiched between n and $3n$, closer to $2n$.
3. Periodic "steps" in total cost correspond to resize events.

Explanation

Why is the measured ratio ≈ 2 rather than 3?
The bound $T(n) \leqslant 3n$ is a *worst-case upper bound*. The actual cost is:

$$T(n) = n + \sum_{j=0}^{\lfloor \lg n \rfloor} 2^j = n + (2^{\lfloor \lg n \rfloor + 1} - 1) < n + 2n = 3n$$

But $2^{\lfloor \lg n \rfloor + 1} \leqslant 2n$, so $T(n) < 3n$. As n grows, $T(n)/n \to 1 + \frac{2n-1}{n} \to 3$ from below. In practice, for powers of 2, $T(n) = n + (2n - 1) = 3n - 1$, achieving the bound. For other n, the ratio is lower.

6 Performance Comparison: Naïve vs. Memoized vs. Bottom-Up

6.1 Timing the three approaches

Solution

Typical output:

```
   n |        Naive |     Memoized |    Bottom-up
-------------------------------------------------
   5 |     0.000006 |     0.000008 |     0.000005
  10 |     0.000261 |     0.000015 |     0.000008
  15 |     0.008312 |     0.000025 |     0.000012
  20 |     0.279154 |     0.000038 |     0.000017
  25 |     8.934217 |     0.000057 |     0.000022
  28 |    71.856432 |     0.000070 |     0.000026
```

Observations:

- **Naïve:** exponential growth ($\mathcal{O}(2^n)$). Doubles roughly every increase of n by 1. Already ≈ 72 seconds at $n = 28$.
- **Memoized:** polynomial ($\Theta(n^2)$). Sub-millisecond even at $n = 28$.
- **Bottom-up:** polynomial ($\Theta(n^2)$). Slightly faster than memoized (no recursion overhead, no dictionary lookups).
- The **speedup** from memoization: $\approx 10^6\times$ at $n = 28$.

6.2 Memoized vs. bottom-up for larger n

Solution

Both are $\Theta(n^2)$, and the plot confirms quadratic growth for both.
Key observations:

- Bottom-up is consistently 20–50% faster than memoized.
- Both scale identically in asymptotic terms.

(a) Yes, there is a measurable constant-factor difference: bottom-up is faster.
(b) The bottom-up version is faster because:

1. **No recursion overhead:** no function call stack frames.
2. **No memoization lookup:** direct array access vs. checking `memo[j] >= 0`.
3. **Better cache locality:** sequential array access pattern.
4. **No Python function call cost:** Python function calls are expensive ($\sim 1\ \mu$s each).

Grading Notes

Students should clearly observe the exponential blowup of the naïve approach and be able to explain why memoization transforms it into polynomial time. The bottom-up vs. memoized difference is a bonus insight.

Bonus Exercises

Bonus 1 — Matrix Chain Multiplication

Solution

```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."""
    n = len(p) - 1
    m = [[0] * (n + 1) for _ in range(n + 1)]
    s = [[0] * (n + 1) for _ in range(n + 1)]

    for l in range(2, n + 1):          # chain length
        for i in range(1, n - l + 2):  # start
            j = i + l - 1              # end
            m[i][j] = float('inf')
            for k in range(i, j):      # split point
                q = m[i][k] + m[k + 1][j] + p[i - 1] * p[k] * p[j]
                if q < m[i][j]:
                    m[i][j] = q
                    s[i][j] = k
    return m, s


def print_optimal_parens(s, i, j):
    """Print the optimal parenthesization."""
    if i == j:
        print(f"A{i}", end="")
    else:
        print("(", end="")
        print_optimal_parens(s, i, s[i][j])
        print_optimal_parens(s, s[i][j] + 1, j)
        print(")", end="")


# Test: CLRS example
p = [30, 35, 15, 5, 10, 20, 25]
m, s = matrix_chain_order(p)
print(f"Minimum multiplications: {m[1][6]}")
print("Optimal parenthesization: ", end="")
print_optimal_parens(s, 1, 6)
print()
assert m[1][6] == 15125
print("Matrix chain test passed!")
```

Output:

```
Minimum multiplications: 15125
Optimal parenthesization: ((A1(A2A3))((A4A5)A6))
Matrix chain test passed!
```

Complexity: $\mathcal{O}(n^3)$ time (three nested loops), $\Theta(n^2)$ space.

Bonus 2 — Edit Distance (Levenshtein)

Solution

```python
def edit_distance(s1, s2):
    """Compute the edit distance between s1 and s2."""
    m, n = len(s1), len(s2)
    d = [[0] * (n + 1) for _ in range(m + 1)]

    for i in range(m + 1):
        d[i][0] = i
    for j in range(n + 1):
        d[0][j] = j

    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if s1[i - 1] == s2[j - 1]:
                d[i][j] = d[i - 1][j - 1]
            else:
                d[i][j] = 1 + min(
                    d[i - 1][j],      # delete
                    d[i][j - 1],      # insert
                    d[i - 1][j - 1]   # substitute
                )
    return d[m][n]


assert edit_distance("kitten", "sitting") == 3
assert edit_distance("saturday", "sunday") == 3
assert edit_distance("", "abc") == 3
assert edit_distance("abc", "abc") == 0
print("Edit distance tests passed!")
```

Explanation: "kitten" → "sitting" in 3 steps:

1. kitten → sitten (substitute k → s)
2. sitten → sittin (substitute e → i)
3. sittin → sitting (insert g)

Complexity: $\Theta(mn)$ time and space, same as LCS. In fact, edit distance generalises LCS: edit_distance$(X, Y) = m + n - 2 \cdot \text{LCS}(X, Y)$ when only insertions and deletions are allowed (no substitutions).

Bonus 3 — Queue with Two Stacks (Amortized)

Solution

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

    def enqueue(self, x):
        self.s1.append(x)
        self.total_ops += 1

    def dequeue(self):
        if not self.s2:
            # Transfer all elements from s1 to s2
            while self.s1:
                self.s2.append(self.s1.pop())
                self.total_ops += 2  # one pop + one push
        if not self.s2:
            raise IndexError("Dequeue from empty queue")
        self.total_ops += 1  # the final pop
        return self.s2.pop()


# Test FIFO behaviour
q = QueueTwoStacks()
for i in range(10):
    q.enqueue(i)

for i in range(10):
    assert q.dequeue() == i

# Interleaved operations
q2 = QueueTwoStacks()
q2.enqueue(1)
q2.enqueue(2)
assert q2.dequeue() == 1
q2.enqueue(3)
assert q2.dequeue() == 2
assert q2.dequeue() == 3

# Measure amortized cost
q3 = QueueTwoStacks()
n = 10000
for i in range(n):
    q3.enqueue(i)
for i in range(n):
    q3.dequeue()

print(f"Total ops for {n} enqueue + {n} dequeue: {q3.total_ops}")
print(f"Amortized per operation: {q3.total_ops / (2 * n):.2f}")
print("Queue with two stacks tests passed!")
```

Typical output:

```
Total ops for 10000 enqueue + 10000 dequeue: 40000
Amortized per operation: 2.00
Queue with two stacks tests passed!
```

Accounting method proof:

Charge \$3 per ENQUEUE and \$1 per DEQUEUE:

- ENQUEUE(x): actual cost \$1 (push onto S_1). Remaining \$2 stored as credit on element x:

  - \$1 for the future pop from S_1 during transfer
  - \$1 for the future push onto S_2 during transfer
- DEQUEUE: if S_2 non-empty, just pop (\$1 actual, \$1 charged). If S_2 empty, transfer all k elements from S_1 to S_2: each element's \$2 credit pays for its pop+push. Then pop from S_2: \$1.

Credit is never negative (each element receives \$2 before it needs to be transferred). Therefore:

$$\sum \hat{c}_i = 3 \cdot (\#\text{ enqueues}) + 1 \cdot (\#\text{ dequeues}) \geqslant \sum c_i$$

Amortized cost per operation: $\mathcal{O}(1)$.