Week 5 — Advanced DP, Greedy Algorithms & Amortized Analysis

Félix Chavelli felix.chavelli@inria.fr

March 18, 2026 · Semester 2

PSL

Today's Agenda

Recap & Motivation

Advanced Dynamic Programming (CLRS Ch. 14)

Greedy Algorithms (CLRS Ch. 15)

Amortized Analysis (CLRS Ch. 16)

Exercises (TD)

Summary & What's Next

Part 1

Recap & Motivation

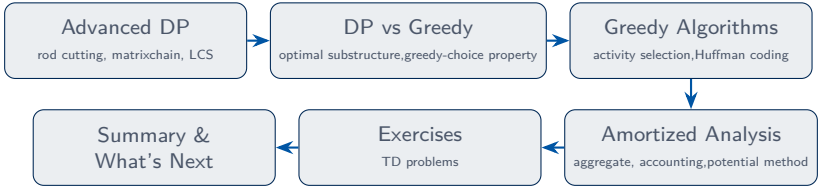
What We Know from S1

- **Memoization:** cache results of recursive calls to avoid recomputation
- **Tabulation (DP):** fill a table iteratively, bottom-up
- **Examples seen in S1:**
  - Making change (number of ways / minimum coins)
  - 0–1 Knapsack
  - Levenshtein edit distance
- **Greedy:** briefly seen with fractional knapsack
- **Key conditions for DP:** optimal substructure + overlapping subproblems

Key Idea

Today we go deeper: new DP problems (rod cutting, matrix chain, LCS), a formal framework for greedy algorithms, and a new tool — **amortized analysis**.

Today's Roadmap



Part 2

Advanced Dynamic Programming (CLRS Ch. 14)

The Four Steps of Dynamic Programming

DP Recipe (CLRS 14)

1. **Characterize** the structure of an optimal solution
2. **Recursively define** the value of an optimal solution
3. **Compute** the value bottom-up (or top-down with memoization)
4. **Construct** the optimal solution from computed information

- Steps 1–3 find the value; step 4 finds the solution itself
- Step 4 often requires keeping an auxiliary table recording **which choice** was made

Rod Cutting — Problem Statement (CLRS 14.1)

- A rod of length n inches; price table p_i for length i
- **Goal:** cut the rod to maximize revenue
- Each cut is free; we may sell the rod uncut
- Number of ways to cut: 2^{n-1} (each of $n-1$ positions: cut or not)

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

$n = 4$

Best: $2 + 2 \to p_2 + p_2 = 10$

Compare: $p_4 = 9$, $p_1 + p_3 = 9$, $p_1 + p_1 + p_2 = 7$, ...

Rod Cutting — Recurrence & Naïve Recursion

Recurrence: Cut a first piece of length i, then optimally cut the remainder:

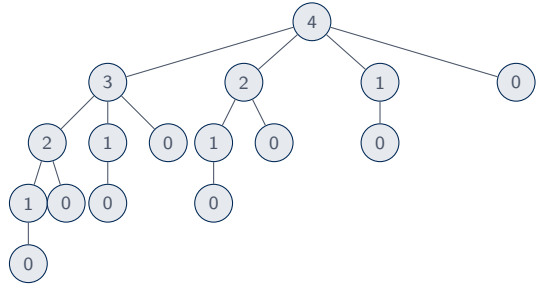
$$r_n = \max_{1 \leqslant i \leqslant n} \{p_i + r_{n-i}\}, \qquad r_0 = 0$$

Naïve recursive:

Input: Prices $p[1..n]$, length n

1: **if** $n = 0$ **then**
2: **return** 0
3: **end if**
4: $q \leftarrow -\infty$
5: **for** $i = 1$ **to** n **do**
6: $q \leftarrow \max\{q,\ p[i] + \textsc{Cut-Rod}(p, n-i)\}$
7: **end for**
8: **return** q

Recursion tree for $n = 4$:



$T(n) = 2^n$ calls!

Rod Cutting — Top-Down with Memoization

Input: Prices $p[1..n]$, length n
1: Create array $r[0..n]$, all $= -\infty$
2: **return** MEMO-CUT(p, n, r)

Input: p, n, r
1: **if** $r[n] \geqslant 0$ **then**
2: **return** $r[n]$ // cached
3: **end if**
4: **if** $n = 0$ **then**
5: $q \leftarrow 0$
6: **else**
7: $q \leftarrow -\infty$
8: **for** $i = 1$ **to** n **do**
9: $q \leftarrow \max\{q, p[i] +$ MEMO-CUT$(p, n-i, r)\}$
10: **end for**
11: **end if**
12: $r[n] \leftarrow q$
13: **return** q

Key Idea

Each subproblem solved **at most once**. $n + 1$ subproblems, each takes $\mathcal{O}(n) \Rightarrow \Theta(n^2)$ total.

Result for the example:

n	1	2	3	4	5
r_n	1	5	8	10	13
cut	1	2	3	2+2	2+3

Rod Cutting — Bottom-Up DP

Input: Prices $p[1..n]$, length n

1: Create arrays $r[0..n]$, $s[0..n]$
2: $r[0] \leftarrow 0$
3: **for** $j = 1$ **to** n **do**
4: $\quad q \leftarrow -\infty$
5: $\quad$ **for** $i = 1$ **to** j **do**
6: $\quad\quad$ **if** $q < p[i] + r[j-i]$ **then**
7: $\quad\quad\quad q \leftarrow p[i] + r[j-i]$
8: $\quad\quad\quad s[j] \leftarrow i$ // record cut
9: $\quad\quad$ **end if**
10: $\quad$ **end for**
11: $\quad r[j] \leftarrow q$
12: **end for**
13: **return** r, s

› **Time:** $\Theta(n^2)$ **Space:** $\Theta(n)$

Reconstructing the solution:

Input: n, array s

1: **while** $n > 0$ **do**
2: $\quad$ **print** $s[n]$
3: $\quad n \leftarrow n - s[n]$
4: **end while**

Trace for $n = 7$

$r_7 = 18$, $s[7] = 1$
Print 1; $n \leftarrow 6$
$s[6] = 6$; print 6; $n \leftarrow 0$ ✓
Cut: $1 + 6 \rightarrow 1 + 17 = 18$

Rod Cutting — Python Implementation

Python

```python
def bottom_up_cut_rod(p, n):
    """Return (max_revenue, first_cut_sizes)."""
    r = [0] * (n + 1)        # r[j] = optimal revenue for length j
    s = [0] * (n + 1)        # s[j] = first piece length in optimal cut
    for j in range(1, n + 1):
        q = -float('inf')
        for i in range(1, j + 1):
            if q < p[i] + r[j - i]:
                q = p[i] + r[j - i]
                s[j] = i
        r[j] = q
    return r[n], s

def print_cuts(s, n):
    while n > 0:
        print(s[n], end=" ")
        n -= s[n]
```

Matrix Chain Multiplication (CLRS 14.2)

- Multiply matrices $A_1 A_2 \cdots A_n$
- Matrix A_i has dimensions $p_{i-1} \times p_i$
- Matrix multiplication is **associative**: parenthesization matters for cost!
- Cost of multiplying $p \times q$ by $q \times r$: pqr scalar multiplications

A_1 : 10×100, A_2 : 100×5, A_3 : 5×50

$(A_1A_2)A_3$:
$10\cdot100\cdot5 + 10\cdot5\cdot50 = 7500$

$A_1(A_2A_3)$:
$100\cdot5\cdot50 + 10\cdot100\cdot50 = 75000$

Key Idea

The number of parenthesizations is the Catalan number $C_{n-1} = \Omega(4^n/n^{3/2})$. Brute force is exponential!

Matrix Chain — DP Formulation

Let $m[i, j] =$ min cost to compute $A_i \cdots A_j$.

$$m[i,j] = \begin{cases} 0 & \text{if } i = j, \\ \min\limits_{i \leqslant k < j} \{ m[i,k] + m[k{+}1, j] + p_{i-1}\, p_k\, p_j \} & \text{if } i < j. \end{cases}$$

- **Subproblems:** $\binom{n}{2} + n = \Theta(n^2)$ entries
- Each entry considers $\mathcal{O}(n)$ split points k
- **Total time:** $\mathcal{O}(n^3)$ **Space:** $\Theta(n^2)$
- Array $s[i, j]$ records the best split point k for reconstruction

Matrix Chain — Worked Example

Dimensions: A_1: 30×35, A_2: 35×15, A_3: 15×5, A_4: 5×10, A_5: 10×20, A_6: 20×25.
So $p = \langle 30, 35, 15, 5, 10, 20, 25\rangle$.

$m[i,j]$	$j{=}1$	$j{=}2$	$j{=}3$	$j{=}4$	$j{=}5$	$j{=}6$
$i{=}1$	0	15750	7875	9375	11875	15125
$i{=}2$		0	2625	4375	7125	10500
$i{=}3$			0	750	2500	5375
$i{=}4$				0	1000	3500
$i{=}5$					0	5000
$i{=}6$						0

Optimal: $m[1,6] = 15125$; Parenthesization: $((A_1(A_2A_3))((A_4A_5)A_6))$

Matrix Chain — Python

Python

```python
def matrix_chain_order(p):
    """p = list of dimensions, len(p) = n+1 for n matrices."""
    n = len(p) - 1
    m = [[0]*n for _ in range(n)]  # m[i][j] = min cost
    s = [[0]*n for _ in range(n)]  # s[i][j] = best split
    for l in range(2, n + 1):        # chain length
        for i in range(n - l + 1):
            j = i + l - 1
            m[i][j] = float('inf')
            for k in range(i, j):
                q = m[i][k] + m[k+1][j] + p[i]*p[k+1]*p[j+1]
                if q < m[i][j]:
                    m[i][j] = q
                    s[i][j] = k
    return m[0][n-1], s
```

Longest Common Subsequence (CLRS 14.4)

LCS

Given sequences $X = \langle x_1, \ldots, x_m \rangle$ and $Y = \langle y_1, \ldots, y_n \rangle$, find the longest sequence Z that is a **subsequence** of both (not necessarily contiguous).

Example

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

LCS: $\langle B, C, B, A \rangle$ (length 4)

Applications:

- Diff tools (`diff`, `git diff`)
- DNA sequence alignment
- Version control, plagiarism detection

LCS — Recurrence

Let $c[i, j]$ = length of LCS of $X[1..i]$ and $Y[1..j]$.

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

- $\Theta(mn)$ subproblems
- Each in $\Theta(1)$
- **Total:** $\Theta(mn)$ time and space

Key Idea

Notice the similarity with **Levenshtein distance** (S1)! LCS uses $\max$ instead of $\min$, and does not count substitutions.

LCS — Worked Example

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

	$\emptyset$	**B**	**D**	**C**	**A**	**B**	**A**
$\emptyset$	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

$c[7, 6] = 4$

LCS $= \langle B, C, B, A \rangle$, length 4. Reconstruction follows backpointers from $c[m, n]$.

DP — When Does It Apply? (Summary)

Two Hallmarks of DP (CLRS 14.3)

1. **Optimal substructure:** an optimal solution contains optimal solutions to subproblems.
2. **Overlapping subproblems:** a recursive algorithm revisits the same subproblems many times.

Pitfall

The **longest simple path** problem has no optimal substructure (subpaths may reuse vertices). DP does *not* apply!

Contrast with **shortest paths**, which *do* have optimal substructure (no cycles in shortest paths of positive-weight graphs).

Part 3

Greedy Algorithms (CLRS Ch. 15)

From DP to Greedy

Dynamic Programming:

- Explore all choices for each subproblem
- Pick the best after seeing results
- Guarantees optimality (when applicable)
- Often $\mathcal{O}(n^2)$ or $\mathcal{O}(n^3)$

Greedy:

- Make the locally best choice immediately
- Never reconsider
- Faster — often $\mathcal{O}(n \log n)$ or $\mathcal{O}(n)$
- But only works when a "greedy-choice property" holds

Key Idea

Two conditions for greedy:

1. **Optimal substructure** (same as DP)
2. **Greedy-choice property:** a locally optimal choice leads to a globally optimal solution

Activity Selection (CLRS 15.1)

Problem

Given n activities with start/finish times (s_i, f_i), select the **maximum number** of mutually compatible activities (no two overlap).

Example

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

Optimal: $\{a_1, a_4, a_8, a_{11}\}$ (4 activities)

Activity Selection — Greedy Strategy

Greedy Choice (CLRS Th. 15.1)

Always select the activity with the **earliest finish time** that is compatible with previously selected activities.

Proof idea:

1. Sort activities by finish time: $f_1 \leqslant f_2 \leqslant \cdots \leqslant f_n$
2. The first activity a_1 (earliest finish) is in some optimal solution:
   if not, replace the first activity in any optimal solution by a_1 — it finishes no later, so compatibility is preserved
3. After choosing a_1, the remaining problem is the same type $\rightarrow$ recurse

Input: Activities sorted by f_i

1: $A \leftarrow \{a_1\}$; $k \leftarrow 1$
2: **for** $i = 2$ **to** n **do**
3: **if** $s_i \geqslant f_k$ **then**
4: $A \leftarrow A \cup \{a_i\}$; $k \leftarrow i$
5: **end if**
6: **end for**
7: **return** A

Time: $\mathcal{O}(n \log n)$ (sort) + $\mathcal{O}(n)$ (scan) = $\mathcal{O}(n \log n)$

Huffman Codes (CLRS 15.3)

Problem

Given a set of characters with known frequencies, design a **prefix-free binary code** that minimizes the total encoding length.

Example

char	a	b	c	d	e	f
freq	45	13	12	16	9	5
fixed	000	001	010	011	100	101
Huffman	0	101	100	111	1101	1100

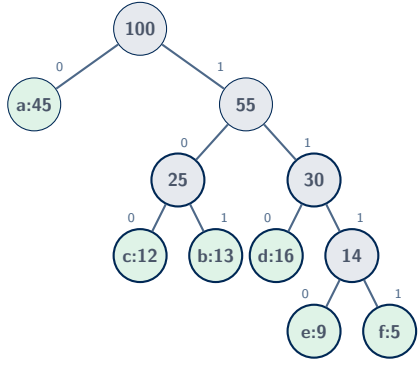
Fixed: 300 bits; Huffman: 224 bits

Greedy strategy:

1. Build a **min-priority queue** Q of characters by frequency
2. Repeat $n-1$ times:
   - Extract two lowest-frequency nodes x, y
   - Create new internal node z with $z.freq = x.freq + y.freq$
   - Insert z into Q
3. Result: a full binary tree

Time: $\mathcal{O}(n \log n)$ using a binary heap.

Huffman — Tree Construction Example



Left edge = 0, right edge = 1. Code for ‘a’ = 0, ‘b’ = 101, etc.

Optimality (CLRS Th. 15.5)

Huffman’s algorithm produces an optimal prefix-free code.

DP vs Greedy — Decision Guide

	Dynamic Programming	Greedy
Choices	Explore all options	Make one locally best choice
Structure	Bottom-up / memoized	Top-down, never reconsider
Proof	Optimal substructure	+ Greedy-choice property
Speed	Typically $\mathcal{O}(n^2)$–$\mathcal{O}(n^3)$	Typically $\mathcal{O}(n \log n)$
0-1 Knapsack	DP (greedy fails)	—
Fractional Knapsack	—	Greedy (sort by ratio)
Activity Selection	DP works but overkill	Greedy (earliest finish)
Matrix Chain	DP required	No greedy-choice
Huffman Coding	—	Greedy

Part 4

Amortized Analysis (CLRS Ch. 16)

What is Amortized Analysis?

Amortized Analysis

Bound the **average cost per operation** in a *worst-case* sequence of n operations. No probability involved — purely worst-case!

- A single operation may be expensive (say $\mathcal{O}(n)$)
- But if expensive ops are rare, the **amortized cost per op** can be $\mathcal{O}(1)$
- **Key difference from average-case:** no probability distribution assumed

Three methods:

1. **Aggregate analysis** — compute total cost, divide by n
2. **Accounting method** — assign "charges" to operations; save credit for future
3. **Potential method** — define a potential function on the data structure

Running Example: Dynamic Array (`std::vector`)

Operation: `push_back` into a dynamic array.

- If space available: insert in $\mathcal{O}(1)$
- If full: double the capacity, copy all elements, then insert
- Single resize: $\Theta(n)$ — seems bad!

Sequence of n push_back ops:
Resizes at sizes $1, 2, 4, 8, \ldots, 2^{\lfloor \lg n \rfloor}$.

op #	cap	cost
1	$1 \to 1$	1
2	$1 \to 2$	$1 + 1$
3	$2 \to 4$	$1 + 2$
4	4	1
5	$4 \to 8$	$1 + 4$
6	8	1
7	8	1
8	8	1
9	$8 \to 16$	$1 + 8$

Aggregate Analysis

Dynamic Array: Amortized $\mathcal{O}(1)$ per push_back

Total cost of n push_back operations:

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

Amortized cost: $T(n)/n \leqslant 3 = \mathcal{O}(1)$ per operation.

Key Idea

Aggregate analysis is the simplest method: compute the total cost of n operations, then divide by n.
Works well when all operations have the "same type."

Accounting Method

- Assign an **amortized cost** $\hat{c}_i$ to each operation i
- Requirement: $\sum_{i=1}^{n} \hat{c}_i \geqslant \sum_{i=1}^{n} c_i$ for **all** n
- If $\hat{c}_i > c_i$: store excess as **credit** on specific elements
- If $\hat{c}_i < c_i$: pay with stored credit

Dynamic array

Charge $\hat{c} = 3$ for each push_back:

- \$1 pays for the insertion itself
- \$1 saved for the element's future copy during resize
- \$1 saved for copying one already-present element

When resize happens: each of the $n/2$ new elements since last resize has \$2 credit $\rightarrow$ pays for copying all n elements.

Credit is *never negative* $\Rightarrow$ amortized cost $= 3 = \mathcal{O}(1)$.

Potential Method

Potential Function

Define $\Phi : D_i \to \mathbb{R}$ mapping the data structure state after operation i to a real number ($\Phi(D_0) = 0$, $\Phi(D_i) \geqslant 0$).

Amortized cost: $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$

Then: $\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0) \geqslant \sum_{i=1}^{n} c_i$

Dynamic array

Let $\Phi = 2 \cdot \mathit{num} - \mathit{cap}$ where $\mathit{num} = \#$ elements, $\mathit{cap} =$ capacity.

- No resize: $c_i = 1$, $\Delta\Phi = 2$, so $\hat{c}_i = 3$
- Resize ($\mathit{num} = \mathit{cap} = m$): $c_i = m + 1$, new cap $= 2m$
  $\Phi_{\text{after}} = 2(m+1) - 2m = 2$, $\Phi_{\text{before}} = 2m - m = m$
  $\hat{c}_i = (m + 1) + (2 - m) = 3$

Every operation: $\hat{c}_i = 3 = \mathcal{O}(1)$ ✓

Another Example: Multi-Pop Stack (CLRS 16.1)

Operations on a stack:

- $\textsc{Push}(S, x)$: cost 1
- $\textsc{Pop}(S)$: cost 1
- $\textsc{Multi-Pop}(S, k)$: pop $\min(k, |S|)$ elements, cost $= \min(k, |S|)$

A single $\textsc{Multi-Pop}$ can cost $\mathcal{O}(n)$.
But in n operations...

Amortized $\mathcal{O}(1)$

Aggregate: Each element is pushed & popped *at most once*. Total pops $\leqslant n$ $\Rightarrow$ total cost $\leqslant 2n$.

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

Accounting: charge \$2 per Push (\$1 push + \$1 credit). Pop/Multi-Pop use the credit.

Incrementing a Binary Counter (CLRS 16.1)

An k-bit binary counter, repeatedly incremented by 1.

Single INCREMENT: flip the lowest 0-bit to 1 and all lower bits to 0. Worst case: $\Theta(k)$ bit flips.

Aggregate analysis:

- Bit 0 flips every increment: n times
- Bit 1 flips every 2 increments: $\lfloor n/2 \rfloor$ times
- Bit j flips $\lfloor n/2^j \rfloor$ times

$$\text{Total} = \sum_{j=0}^{k-1} \left\lfloor \frac{n}{2^j} \right\rfloor < 2n$$

Amortized: $< 2 = \mathcal{O}(1)$ per increment.

#	$A[7]$	$A[6]$	$\cdots$	$A[2]$	$A[1]$	$A[0]$	cost
0	0	0		0	0	0	—
1	0	0		0	0	1	1
2	0	0		0	1	0	2
3	0	0		0	1	1	1
4	0	0		1	0	0	3
5	0	0		1	0	1	1
6	0	0		1	1	0	2
7	0	0		1	1	1	1
8	0	1		0	0	0	4

Part 5

Exercises (TD)

Exercise 1: Rod Cutting Variant

Problem: Modify the rod cutting problem so that each cut has a **fixed cost** $c > 0$. The revenue becomes:

$$r_n = \max_{1 \leqslant i \leqslant n} \left\{ p_i + r_{n-i} - \begin{cases} c & \text{if } i < n \\ 0 & \text{if } i = n \end{cases} \right\}$$

With $c = 1$

For $n = 4$, $p = [1, 5, 8, 9]$:
No cut: $r_4 = 9$
Cut $2 + 2$: $5 + 5 - 1 = 9$
Cut $1 + 3$: $1 + 8 - 1 = 8$
Best: $r_4 = 9$ (no cut or $2{+}2$)

Tasks:

1. Write the modified recurrence.
2. Implement a bottom-up DP solution in Python.
3. What is the time complexity?

Exercise 1: Solution

Modified recurrence:

$$r_n = \max\Big\{p_n,\ \max_{1 \leqslant i < n} \big\{p_i + r_{n-i} - c\big\}\Big\}, \qquad r_0 = 0$$

Implementation: Same bottom-up structure, subtract c when $i < j$.

```
1: r[0] ← 0
2: for j = 1 to n do
3:     q ← p[j]                                   // no cut
4:     for i = 1 to j − 1 do
5:         q ← max{q, p[i] + r[j − i] − c}
6:     end for
7:     r[j] ← q
8: end for
```

Complexity: Still $\Theta(n^2)$ time, $\Theta(n)$ space.

Exercise 2: LCS — Trace & Reconstruct

Problem: Given $X = \langle C, A, T \rangle$ and $Y = \langle A, C, T, G \rangle$:

1. Fill in the DP table $c[i, j]$.
2. Trace back to find an LCS.
3. Give the time and space complexity.

Exercise 2: LCS — Trace & Reconstruct

Problem: Given $X = \langle C, A, T \rangle$ and $Y = \langle A, C, T, G \rangle$:

1. Fill in the DP table $c[i, j]$.
2. Trace back to find an LCS.
3. Give the time and space complexity.

Solution:

$c[i,j]$	$\emptyset$	A	C	T	G
$\emptyset$	0	0	0	0	0
C	0	0	1	1	1
A	0	1	1	1	1
T	0	1	1	2	2

LCS $= \langle C, T \rangle$ or $\langle A, T \rangle$ (length 2).

Time: $\Theta(mn)$ where $m = |X|, n = |Y|$. **Space:** $\Theta(mn)$ (or $\Theta(\min(m, n))$ if only length needed).

Exercise 3: Activity Selection

Problem: Given the activities:

i	1	2	3	4	5	6
s_i	1	2	4	1	5	8
f_i	3	5	7	8	9	10

1. Apply the greedy algorithm (earliest finish time). Which activities are selected?
2. Would selecting by shortest duration give the same result?
3. Would selecting by earliest start time give the same result?

Exercise 3: Activity Selection

Problem: Given the activities:

i	1	2	3	4	5	6
s_i	1	2	4	1	5	8
f_i	3	5	7	8	9	10

1. Apply the greedy algorithm (earliest finish time). Which activities are selected?
2. Would selecting by shortest duration give the same result?
3. Would selecting by earliest start time give the same result?

Solution:

1. Already sorted by f_i. Select a_1 ($f_1 = 3$), then a_3 ($s_3 = 4 \geqslant 3$, $f_3 = 7$), then a_6 ($s_6 = 8 \geqslant 7$). 3 activities selected.
2. Shortest duration: $a_1(2), a_2(3), a_3(3), a_5(4), a_4(7), a_6(2)$. Select $a_1, a_6, a_3 \rightarrow$ only 3. Same count here, but this heuristic is not always optimal.
3. Earliest start: select a_1 or a_4 ($s = 1$). If a_4: only a_4, a_6 (2 activities). Not optimal!

Exercise 4: Huffman Coding

Problem: Build a Huffman tree for characters with frequencies:

A:15, B:7, C:6, D:6, E:5

1. Show each step of the algorithm (which nodes are merged).
2. Give the resulting codes.
3. What is the average bits per character?

Exercise 4: Huffman Coding

Problem: Build a Huffman tree for characters with frequencies:

A:15, B:7, C:6, D:6, E:5

1. Show each step of the algorithm (which nodes are merged).
2. Give the resulting codes.
3. What is the average bits per character?

Solution:

1. Merge D(6)+E(5)=11; merge C(6)+B(7)=13; merge DE(11)+CB(13)=24; merge A(15)+DECB(24)=39.
2. A: 0, D: 100, E: 101, C: 110, B: 111.
3. Weighted average:
$$\frac{15 \cdot 1 + 7 \cdot 3 + 6 \cdot 3 + 6 \cdot 3 + 5 \cdot 3}{39} = \frac{15 + 21 + 18 + 18 + 15}{39} = \frac{87}{39} \approx 2.23 \text{ bits/char.}$$

Exercise 5: Amortized Analysis of a Queue with Two Stacks

Problem: 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 elements from S_1 and push onto S_2; then pop from S_2

Show that DEQUEUE has amortized cost $\mathcal{O}(1)$.

Exercise 5: Amortized Analysis of a Queue with Two Stacks

Problem: 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 elements from S_1 and push onto S_2; then pop from S_2

Show that DEQUEUE has amortized cost $\mathcal{O}(1)$. **Solution (Accounting method):**

- Charge \$3 per ENQUEUE: \$1 for the push onto S_1, \$1 credit for the future pop from S_1, \$1 credit for the future push onto S_2.
- DEQUEUE: each element moved from $S_1 \to S_2$ is paid for by its credit. The final pop from S_2 costs \$1.
- Total charge per Enqueue: 3. Total charge per Dequeue: 1 (self) + 0 (moves paid by credits).
- Credit never negative $\Rightarrow$ amortized cost $= \mathcal{O}(1)$ per operation.

Part 6

Summary & What's Next

Summary — Key Takeaways

1. **DP recipe:** characterize, define recurrence, compute bottom-up, reconstruct
2. **Rod cutting:** $r_n = \max_i\{p_i + r_{n-i}\}$; $\Theta(n^2)$
3. **Matrix chain:** $\mathcal{O}(n^3)$ to find optimal parenthesization
4. **LCS:** $\Theta(mn)$; similar to Levenshtein distance (S1)
5. **Greedy = DP + greedy-choice property:** make the locally best choice
6. **Activity selection:** earliest finish time; $\mathcal{O}(n \log n)$
7. **Huffman codes:** optimal prefix-free encoding; $\mathcal{O}(n \log n)$
8. **Amortized analysis:** aggregate, accounting, potential methods
9. **Key insight:** dynamic arrays have $\mathcal{O}(1)$ amortized push_back

Key Idea

DP, greedy, and amortized analysis are the "power tools" of algorithm design. Choosing the right tool depends on the problem's structure.

What's Next — Week 6

→ Next week: Elementary Graph Algorithms (CLRS Ch. 20)

- Graph representations: adjacency list vs matrix
- Breadth-first search (BFS) & shortest paths in unweighted graphs
- Depth-first search (DFS) & edge classification
- Applications: connected components, cycle detection

To prepare: Think about how you would represent a social network or a road map in code.

Questions?

References

- **Cormen, Leiserson, Rivest, Stein** — *Introduction to Algorithms*, 4th ed., MIT Press, 2022. **Chapters 14–16.**

PSL UNIVERSITÉ PARIS

PSL University · Bachelor of Science in AI · 2025–2026