Chapter 14. Dynamic Programming

Joon Soo Yoo

May 15, 2025

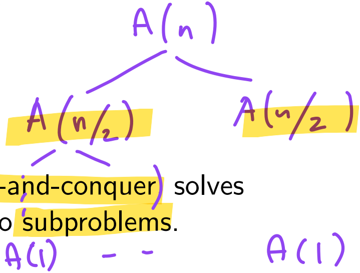
Assignment

- Read §14.1
- Problems
  - §14.1 - 2, 5

Chapter 14: Dynamic Programming

- Chapter 14.1: Rod Cutting
- Chapter 14.2: Matrix-Chain Multiplication
- Chapter 14.3: Elements of Dynamic Programming
- Chapter 14.4: Longest Common Subsequence
- Chapter 14.5: Optimal Binary Search Trees

Dynamic Programming: Overview



- Dynamic programming, like divide-and-conquer solves problems by combining solutions to subproblems.
- **Note:** “Programming” here refers to a **tabular method**, not writing code.
- Divide-and-conquer divides a problem into *disjoint* subproblems.
- Dynamic programming applies when subproblems **overlap**.

When to Use Dynamic Programming

- In divide-and-conquer, subproblems are independent.
- In dynamic programming, subproblems **share sub-subproblems**.
- Recomputing shared subproblems leads to inefficiency.
- Dynamic programming solves each sub-subproblem **once**, storing results in a table.

Dynamic Programming: Problem Type

- Typically applies to **optimization problems**.
- Many possible solutions exist; each has a **value**.
- Goal: Find a solution with the **optimal** (min or max) value.
- We refer to this as an *optimal solution*, not necessarily *the* optimal solution.

Four Steps to Solve with Dynamic Programming

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution (usually bottom-up).
4. Construct an optimal solution from computed information.

Note: If only the value is needed, step 4 can be skipped.

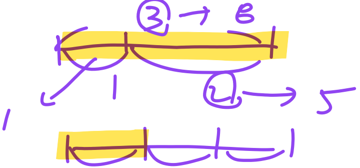
Four Steps in Dynamic Programming: Rod Cutting

1. **Optimal Substructure:** If we make a first cut of length i, then the best revenue is: $\underline{p[i] + r[n-i]}$
2. **Recursive Definition:** $\left(\underline{r[n]} = \max_{1 \leq i \leq n}(p[i] + \underline{r[n-i]})\right)$
3. **Bottom-Up Computation:** Fill array $r[0 \ldots n]$ iteratively using the recurrence.
4. **Solution Construction:** Track the first cut $s[n]$, then reconstruct by reducing $n \to n - s[n]$.

Chapter 14: Dynamic Programming

- **Chapter 14.1: Rod Cutting**
- Chapter 14.2: Matrix-Chain Multiplication
- Chapter 14.3: Elements of Dynamic Programming
- Chapter 14.4: Longest Common Subsequence
- Chapter 14.5: Optimal Binary Search Trees

14.1 Rod Cutting: Problem Setup



- **Given**: Price table $p[i]$ for rods of length i, and a rod of length n.
- **Goal**: Determine the maximum revenue r_n by cutting (or not cutting) the rod.

r_n

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

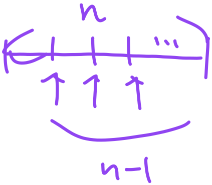
Example: For $n = 4$, cutting into two 2-inch pieces gives:
$r_4 = p[2] + p[2] = 5 + 5 = 10$ → optimal.

$(3) \rightarrow 8$

$(2,1) \leftrightarrow (1,2) \rightarrow 6$

$(1,1,1) \rightarrow 3$

How Many Ways to Cut?

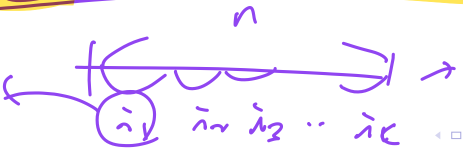


2^{n-1}

- For rod of length n, there are $n-1$ potential cut positions.
- Each position: cut or don't cut → 2^{n-1} total ways to cut.
- Example notation: $7 = 2 + 2 + 3$ means cut into pieces of lengths 2, 2, and 3.
- We want a decomposition $n = i_1 + i_2 + \cdots + i_k$ that **maximizes:** $r_n = p[i_1] + p[i_2] + \cdots + p[i_k]$

$r_n = p[i_1] + \cdot\cdot + p[i_k]$

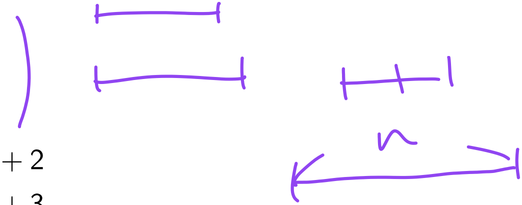
$p[i_1]$
$p[i_2]$



→ $i_1 + \cdot\cdot + i_k = n$

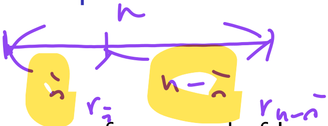
Rod Cutting: Optimal Revenue by Inspection

- $r_1 = 1$ from 1
- $r_2 = 5$ from 2
- $r_3 = 8$ from 3
- $r_4 = 10$ from $2 + 2$
- $r_5 = 13$ from $2 + 3$
- $r_6 = 17$ from 6
- $r_7 = 18$ from $1 + 6$ or $2 + 2 + 3$
- $r_8 = 22$ from $2 + 6$
- $r_9 = 25$ from $3 + 6$
- $r_{10} = 30$ from 10



Observation: Sometimes no cut yields the optimal value.

Recursive Formulation: Two Subproblems

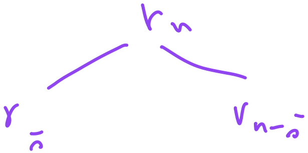


- We express r_n — the max revenue from a rod of length n — using:

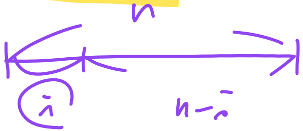
$$r_n = \max\{p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots \}$$

$$r_n = \max\{p_n,\ r_1 + r_{n-1},\ r_2 + r_{n-2},\ \ldots,\ r_{n-1} + r_1\}$$

- **Option 1:** No cut → revenue = p_n
- **Option 2:** First cut at position $i \in [1, n-1]$ → split into sizes i and $n - i$, and recursively solve both
- This uses **optimal substructure**: solve two smaller rod-cutting subproblems.



Recursive Formulation: Single Subproblem View



$r_n = p_i + r_{n-i}$

- A simplified recurrence:

$$r_n = \max_{1 \le i \le n} \{p_i + r_{n-i}\}$$

$\longleftrightarrow$

$r_n = \max_{1 \le i \le n} \{r_i + r_{n-i}$

- View each decomposition as:
  - Cutting off a piece of size i
  - Recursively solving the remainder of size $n - i$
- If $i = n$, this means no cut at all: $r_n = p_n + r_0 = p_n + 0$
- Advantage: only one recursive call per term

Recursive Rod Cutting: Top-Down Approach

DP

① define

② recursive

③ algorithmic

Top-down   bottom-up

- Implements the recurrence:

$$r_n = \max_{1 \le i \le n} (p[i] + r[n-i])$$

- Straightforward, recursive algorithm:

✓

```
CUT-ROD(p, n)  → T(n)
    if $n == 0$
        return 0
    $q = -\infty$
    for $i = 1$ to $n$
        $q = \max\{q, p[i] + \text{CUT-ROD}(p, n-i)\}$
    return $q$
```

$$\max \left\{ \begin{array}{l} p(1) + CR(n-1) \\ p(2) + CR(n-2) \\ \vdots \\ p[n] + CR(0) \end{array} \right\}$$

$T(0) = 1$

$$T(n) = 1 + T(n-1) + T(n-2) + \dots + T(0)$$
$$= 1 + \sum^{n-1} T(n)$$

Why This Recursive Version is Inefficient

- This version works — it computes the correct value of r_n
- But it has **exponential time complexity:**
  - Each call spawns multiple recursive calls
  - Many subproblems are solved repeatedly
- For example, `CUT-ROD(p, 3)` calls:

$$CUT\text{-}ROD(p,2),\ CUT\text{-}ROD(p,1),\ CUT\text{-}ROD(p,0)$$

- Example: For $n = 40$, execution can take minutes or even hours.
- **Time roughly doubles for each increase in** n
- This motivates the need for a better solution: **memoization or tabulation**.

The Recursion Tree Grows Exponentially

- Define $T(n)$ as the number of calls made to `CUT-ROD(p, n)`.
- The recurrence:

$$T(n) = 1 + \sum_{j=0}^{n-1} T(j) \quad \text{with base case } T(0) = 1$$

- Solves to:

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

- So the naive recursive version takes **exponential time**.

Why is $T(n) = 2^n$? (Step-by-Step Proof)

Let $T(n)$ be the number of calls made by `CUT-ROD(p, n)`.

- Recursive definition:

$$T(n) = 1 + \sum_{j=0}^{n-1} T(j)$$

- Base case:

$$T(0) = 1$$

- Build up step-by-step:

$$\begin{aligned}
T(1) &= 1 + T(0) = 1 + 1 = 2 \qquad 2^1 \\
T(2) &= 1 + \underset{1}{T(0)} + \underset{2}{T(1)} = 1 + 1 + 2 = 4 \qquad 2^2 \\
T(3) &= 1 + T(0) + T(1) + T(2) = 1 + 1 + 2 + 4 = 8 \qquad 2^3 \\
T(4) &= 1 + T(0) + T(1) + T(2) + T(3) = 1 + 1 + 2 + 4 + 8 = 16 \qquad 2^4
\end{aligned}$$

$T(4) = 2^4$

- Pattern: $T(n) = 2^n$

Conclusion: The recursion tree grows exponentially — `CUT-ROD` has exponential time complexity.

Dynamic Programming for Rod Cutting

- Goal: avoid solving the same subproblems repeatedly (as in naive recursion).
- Key idea: **solve each subproblem only once** and **save the result**.
- This transforms the exponential-time recursive solution into a polynomial-time one.
- This is a classic **time–memory trade-off**: *extra space* to store results → *less time* spent recomputing.
- Final runtime: $\theta(n^2)$ instead of $\theta(2^n)$

Two DP Implementations

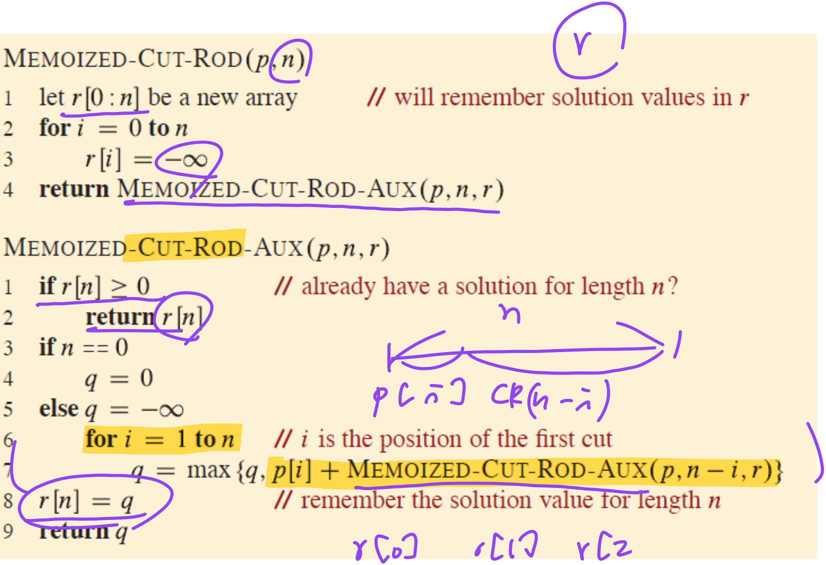
1. **Top-Down with Memoization**
   - Recursive approach that saves results as they are computed.
   - Each call checks: "Have I already solved this subproblem?"
   - If yes → return saved result. If no → compute, save, then return.
   - We call this **memoization**.

2. **Bottom-Up Approach**
   - Solve smaller subproblems first, then build up to the full problem.
   - Use a loop to fill a table from size 0 up to size n.
   - No recursion or function calls; everything is table-driven.

Memoization vs. Bottom-Up

- **Same asymptotic runtime:** Both run in $\Theta(n^2)$
- **Top-Down (Memoized):**
  - Easier to write (recursive)
  - May avoid solving all subproblems if not needed
  - Higher overhead due to recursive calls
- **Bottom-Up:**
  - More efficient in practice
  - Solves all subproblems in order
  - Avoids recursion → better constant factors

Memoized-CUT-ROD (Top-Down)



```
MEMOIZED-CUT-ROD($p, n$)
    let $r[0:n]$ be a new array        // will remember solution values in $r$
    for $i = 0$ to $n$
        $r[i] = -\infty$
    return MEMOIZED-CUT-ROD-AUX($p, n, r$)

MEMOIZED-CUT-ROD-AUX($p, n, r$)
    if $r[n] \geq 0$                   // already have a solution for length $n$?
        return $r[n]$
    if $n == 0$
        $q = 0$
    else $q = -\infty$
        for $i = 1$ to $n$             // $i$ is the position of the first cut
            $q = \max\{q, p[i] + $MEMOIZED-CUT-ROD-AUX$(p, n-i, r)\}$
    $r[n] = q$                         // remember the solution value for length $n$
    return $q$
```

BOTTOM-UP-CUT-ROD (Iterative)

```
BOTTOM-UP-CUT-ROD($p, n$)
let $r[0:n]$ be a new array          // will remember solution values in $r$
$r[0] = 0$
for $j = 1$ to $n$                   // for increasing rod length $j$
    $q = -\infty$
    for $i = 1$ to $j$               // $i$ is the position of the first cut
        $q = \max\{q, p[i] + r[j-i]\}$
    $r[j] = q$                       // remember the solution value for length $j$
return $r[n]$
```

j

$1 + 2 + \cdots + n$

$j = 1, \; i = 1$

r[i]

0	1	2	3	
0	/	///	///	

$q = p[1] + r[0]$

$r[1] = q$

$j = 2, \; i = 1$

$i = 2$

Time Complexity of Dynamic Programming Rod Cutting

Key Idea:

- Both DP versions solve the problem using *n* subproblems: one for each rod length from 1 to *n*.

Bottom-Up Version:

$\Theta(2^n)$

- Outer loop runs *n* times (for $j = 1$ to n)
- Inner loop tries all first cuts $i = 1$ to j
- Total work:

$$\sum_{j=1}^{n} j = \frac{n(n+1)}{2} = \Theta(n^2)$$

$\sum j$

Question?