CSD2183: Data Structures

Dynamic Programming

Bingjie Xu

Contact: bingjie.xu@singaporetech.edu.sg

16 March 2026

Information

- Homework project 2 is due on 4 April 2026 at 11.59 PM.
- Homework project 1 results have been released on xSITe.

Overview

	Basics (Week 1 - 6)	**Advanced (Week 8 - 13)**
1	Foundations	Graph Foundations and Traversal
2	Running Times	Disjoint Sets and Minimum Spanning Trees
3	Sorting	Shortest Paths (Single-Source)
4	List and Hash Tables	**Dynamic Programming**
5	Trees	Greedy Algorithms
6	Consultation	Consultation

Cormen, Thomas H., et al. Introduction to algorithms 4th edition. MIT press, 2022.

Recap

This course has the following topics:

Basics and DS

- Running Times
- Data Structures (array, stack, queue, linked-list, hash table, tree, graph, DS for disjoint sets)

Algorithms

- Sorting
- Graphs (traversal, MST, SSSP)

Techniques

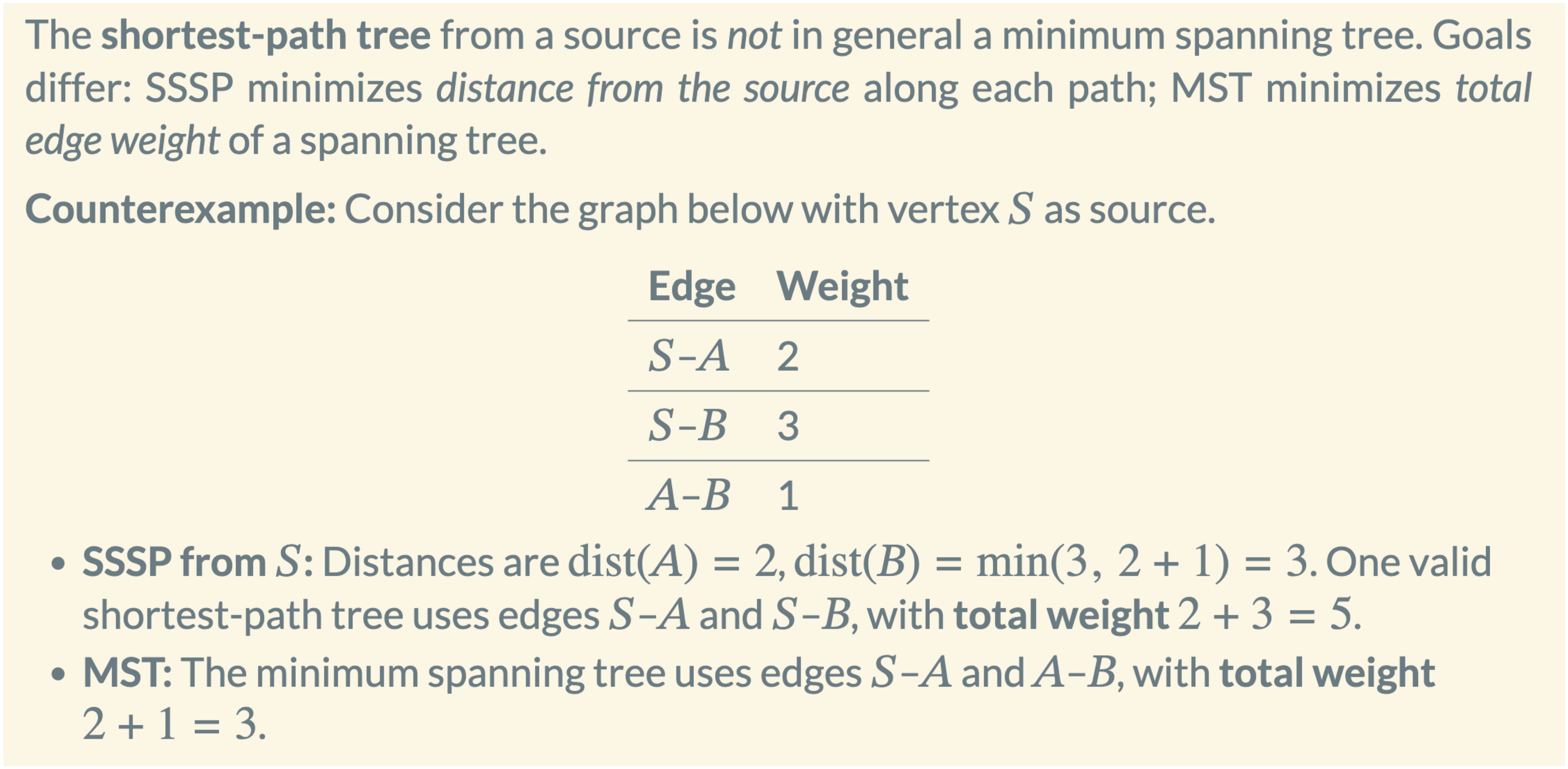
- Recursion
- Divide-and-Conquer
- Dynamic Programming
- Greedy Methods

Recap: Graph Algorithms

Graph Algorithm	Description	Graph Construction	Representatives
Graph Traversal	Reachability from a source vertex to another vertex	Both directed or undirected, **unweighted**	BFS, DFS
Minimum Spanning Trees	Connect all vertices with the min cost $\min \sum_{(u,v) \in T} w(u,v)$	Connected & undirected graph, weighted	Prim, Kruskal
Shortest Paths (Single-Source)	Connect a source vertex to another vertex with the min cost $\min d(s,v)$	Directed (undirected treated as bidirectional), weighted	Dijkstra, Bellman-Ford, A*, BMSSP etc.

Recap: Graph Algorithms

Does SSSP guarantee MST? No!



The **shortest-path tree** from a source is *not* in general a minimum spanning tree. Goals differ: SSSP minimizes *distance from the source* along each path; MST minimizes *total edge weight* of a spanning tree.

Counterexample: Consider the graph below with vertex S as source.

Edge	**Weight**
S–A	2
S–B	3
A–B	1

- **SSSP from S:** Distances are $\text{dist}(A) = 2, \text{dist}(B) = \min(3,\ 2 + 1) = 3$. One valid shortest-path tree uses edges S–A and S–B, with **total weight** $2 + 3 = 5$.
- **MST:** The minimum spanning tree uses edges S–A and A–B, with **total weight** $2 + 1 = 3$.

Learning Objectives

- Compare memoization (top-down) vs. tabulation (bottom-up) approaches in Dynamic Programming.
- Design **overlapping subproblems and optimal substructure**, the two key properties of DP.
- Analyze the time and space complexity of DP solutions.
- Learn how to identify DP problems and apply DP to real-world problems.

Introduction to Dynamic Programming (DP)

Dynamic Programming in 1950s

Copy from Bellman’s autobiography

I spent the Fall quarter (of 1950) at RAND. My first task was to find a name for multistage decision processes. An interesting question is, Where did the name, dynamic programming, come from? The 1950s were not good years for mathematical research. We had a very interesting gentleman in Washington named Wilson. He was Secretary of Defense, and he actually had a pathological fear and hatred of the word research. I’m not using the term lightly; I’m using it precisely. His face would suffuse, he would turn red, and he would get violent if people used the term research in his presence. You can imagine how he felt, then, about the term mathematical. The RAND Corporation was employed by the Air Force, and the Air Force had Wilson as its boss, essentially. Hence, I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics inside the RAND Corporation. What title, what name, could I choose? In the first place I was interested in planning, in decision making, in thinking. But planning, is not a good word for various reasons. I decided therefore to use the word “programming”. I wanted to get across the idea that **this was dynamic, this was multistage, this was time-varying** I thought, lets kill two birds with one stone. Lets take a word that has an absolutely precise meaning, namely dynamic, in the classical physical sense. It also has a very interesting property as an adjective, and that is it's impossible to use the word dynamic in a pejorative sense. Try thinking of some combination that will possibly give it a pejorative meaning. It's impossible. Thus, I thought dynamic programming was a good name. It was something not even a Congressman could object to. So I used it as an umbrella for my activities.

In short, his boss hates math.

He needs a cool term, which sounded less math.

Programming refers to a military schedule.

Dynamic, multistage, and time-varying.

Richard Bellman

Dynamic Decision Problem

Given a directed graph G with a starting vertex v_0.

- each vertex represents some state
- each edge e has some reward r_e (can be negative)

We define the reward of a path $p = v_0 v_1 v_2 \cdots$ be

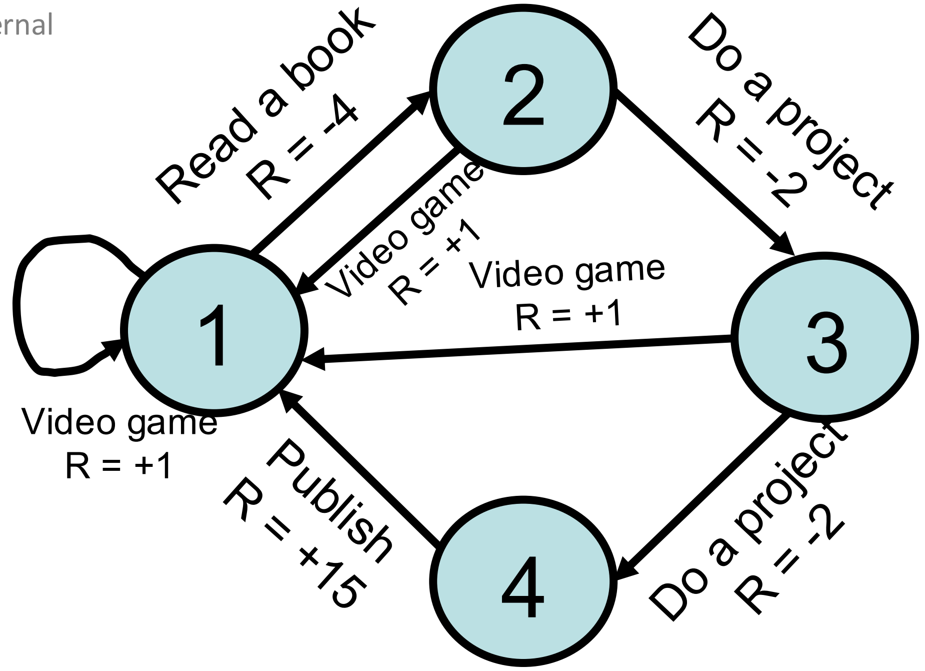
$$R(p) = r_{v_0 v_1} + \gamma \cdot r_{v_1 v_2} + \gamma^2 \cdot r_{v_2 v_3} + \cdots$$

for some given discount factor $0 < \gamma < 1$.

Goal: Find a path p starts at v_0 with maximum reward $R(p)$.

(This is a simplified version of Markov Decision process.)

Example



Start vertex is 1.

Question: What is the best path?

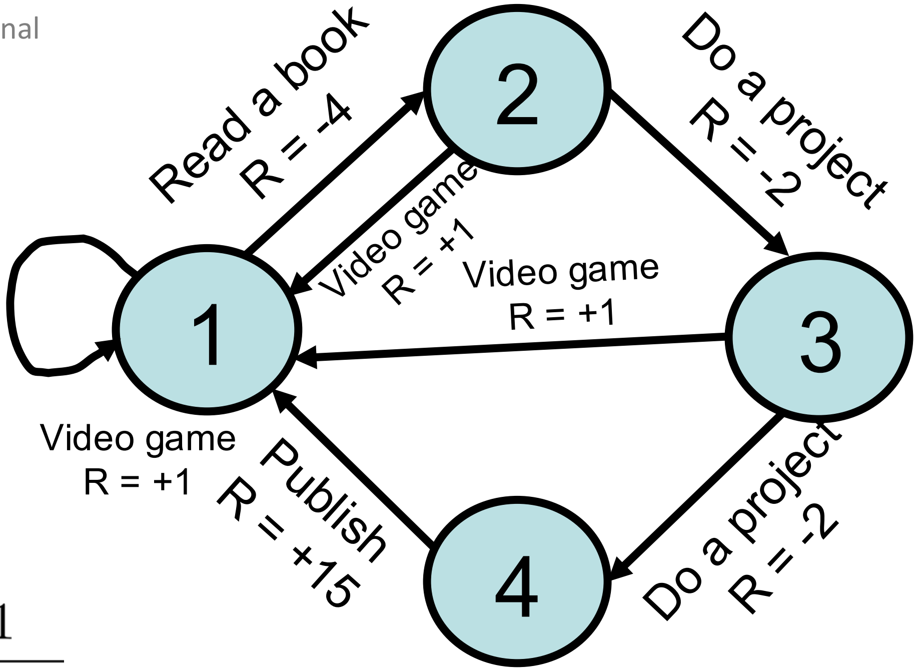
There are two natural choices:

1. Simply play video game every day (1,1,1,1,1,…)
   Take the immediate, simple reward of the video game
2. Keep publishing (1,2,3,4,1,2,3,4,…)
   Publish or Perish ☹

So, which is the best path to maximize reward?

It depends on γ. (Or, how many days you have left?)

Example



Start vertex is 1.

Path 1: 1,1,1, ⋯.

The reward is $1 + \gamma^1 + \gamma^2 + \cdots = \frac{1}{1-\gamma}$

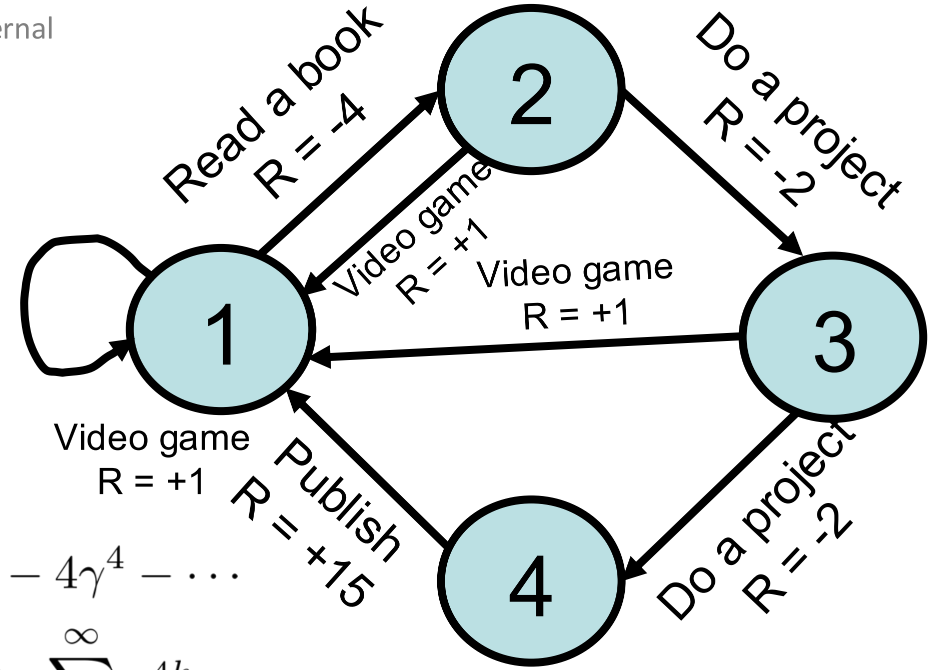
We can view γ is the probability of the process continuing.

$1/(1-\gamma)$ is the expected life span.

Hence, the reward per day (reward/expected life span) is simply 1.

$$\boxed{R(p) = r_{v_0 v_1} + \gamma \cdot r_{v_1 v_2} + \gamma^2 \cdot r_{v_2 v_3} + \cdots}$$

Example



Start vertex is 1.

Path 2: $1,2,3,4,1,2,3,4,\cdots$

The reward is $-4-2\gamma^1-2\gamma^2+15\gamma^3-4\gamma^4-\cdots$

$$=(-4-2\gamma^1-2\gamma^2+15\gamma^3)\cdot\sum_{k=0}^{\infty}\gamma^{4k}$$

$$=\frac{-4-2\gamma^1-2\gamma^2+15\gamma^3}{1-\gamma^4}$$

The reward/day is $\dfrac{-4-2\gamma^1-2\gamma^2+15\gamma^3}{1-\gamma^4}\Big/\dfrac{1}{1-\gamma}=\dfrac{15\gamma^3-2\gamma^2-2\gamma-4}{\gamma^3+\gamma^2+\gamma+1}$

When $\gamma\approx 0$ (representing a very short expected lifespan), the reward/day is around -4. (worse than Path 1)

When $\gamma\approx 1$ (representing a long expected lifespan), the reward/day is around $7/4$. (better than Path 1)

Dynamic Decision Problem

Given a directed graph G with a starting vertex v_0.

- each vertex represents some state
- each edge e has some reward r_e (can be negative)

We define the reward of a path $p = v_0 v_1 v_2 \cdots$ be

$$R(p) = r_{v_0 v_1} + \gamma \cdot r_{v_1 v_2} + \gamma^2 \cdot r_{v_2 v_3} + \cdots$$

for some given discount factor $0 < \gamma < 1$.

Goal: Find a path p starts at v_0 with maximum reward $R(p)$.

Bellman shows how to solve it using “Dynamic Programming”.

Hints: Instead of finding the path, find the reward first!

Bellman Equation

Fix any vertex v_0.
Let $p = v_0 v_1 v_2 \cdots$ be a path with maximum reward.
Let $q = v_1 v_2 v_3 \cdots$

Note that
$$\begin{aligned} R(p) &= r_{v_0 v_1} + \gamma \cdot r_{v_1 v_2} + \gamma^2 \cdot r_{v_2 v_3} + \gamma^3 \cdot r_{v_3 v_4} + \cdots \\ &= r_{v_0 v_1} + \gamma \left(r_{v_1 v_2} + \gamma \cdot r_{v_2 v_3} + \gamma^2 \cdot r_{v_3 v_4} + \cdots \right) \\ &= r_{v_0 v_1} + \gamma \cdot R(q) \end{aligned}$$
Hence, if p maximizes reward at v_0, q maximizes reward at v_1.

Let $R(v)$ be the maximum reward for path starting at v.
We have $\boxed{R(v) = \max_u [r_{vu} + \gamma \cdot R(u)]}$ (Bellman equation).

Compute smaller subproblems to achieve larger ones.

DP Key Properties

- Give a solution of a problem using smaller **overlapping subproblems** where the parameters of all sub-problems are determined in-advance.
- Useful when the same subproblems happen over and over again.

DP Key Properties

Optimal substructure: First characterize the structure of an optimal solution.

- We must prove that we can create an optimal solution to a problem using optimal solutions to subproblems.
- Can't use DP if the optimal solution to a problem might not require subproblem solutions to be optimal. This often happens when the subproblems are not **independent** of each other.

To get the optimal substructure, imagine that the DP gods tells you what was the last choice made in an optimal solution.

Real-World Applications of DP

Dynamic programming appears throughout industry and research whenever you need to make **optimal decisions over stages** while reusing solutions to smaller subproblems.

- Bioinformatics: Sequence alignment (e.g. Needleman–Wunsch, Smith–Waterman) uses DP to compare DNA or protein sequences; the same idea underlies edit distance and longest common subsequence.
- Finance & operations: Portfolio optimization, scheduling, and resource allocation often reduce to knapsack-like or stage-based optimization solved with DP.
- Routing & planning: Finding shortest or constrained paths in networks, and many planning problems, are solved with DP or DP-inspired methods.

Classical DP Problems

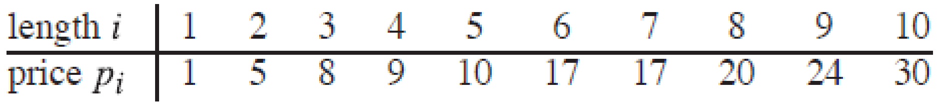
- **Rod cutting**
- Matrix-chain multiplication

Rod Cutting Problem

How to cut steel rods into pieces to maximize the revenue you can get? Each cut is free. A piece of length i (integer) is worth p_i dollars.

Input: A length n and table of prices p_i , for $i = 1,2, \dots n$.

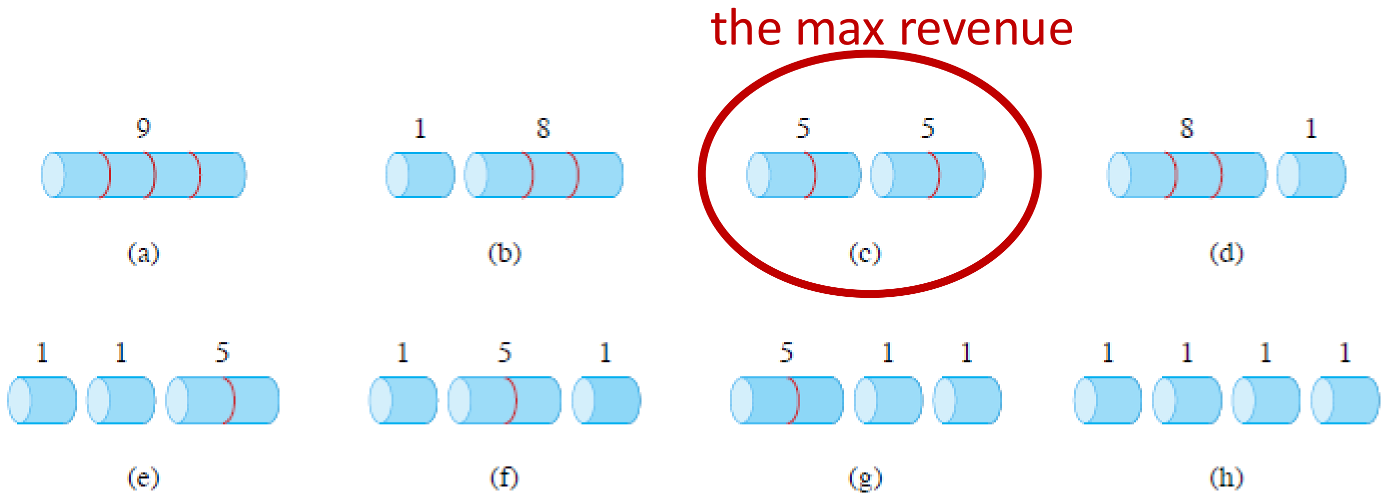
Output: The maximum revenue obtainable for rods whose lengths sum to n, computed as the sum of the prices for the individual rods.



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

Rod Cutting Problem

8 ways to cut a rod of length 4, number indicating the price.



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

Rod Cutting Problem

Thus, can infer the optimal revenue r_n to cut a rod of length n

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

no cut, price of a rod of length n

i	r_i	optimal solution
1	1	1 (no cuts)
2	5	2 (no cuts)
3	8	3 (no cuts)
4	10	$2 + 2$
5	13	$2 + 3$
6	17	6 (no cuts)
7	18	$1 + 6$ or $2 + 2 + 3$
8	22	$2 + 6$

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

Optimal Substructure

Let r_i be the maximum amount of money you can get with a rod of size i. We can view the problem recursively as follows:

- First, cut a piece off the left end of the rod, and sell it.
- Then, find the optimal way to cut the remainder of the rod.

Now we don't know how large a piece we should cut off. So we try all possible cases. First we try cutting a piece of length 1, and combining it with the optimal way to cut a rod of length n-1. Then we try cutting a piece of length 2, and combining it with the optimal way to cut a rod of length n-2. We try all the possible lengths and then pick the best one. We end up with:

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

Implementation v1: Recursive Top-Down

Naive Implementation of $\boxed{r_n = \max_{1 \le i \le n} (p_i + r_{n-i})}$

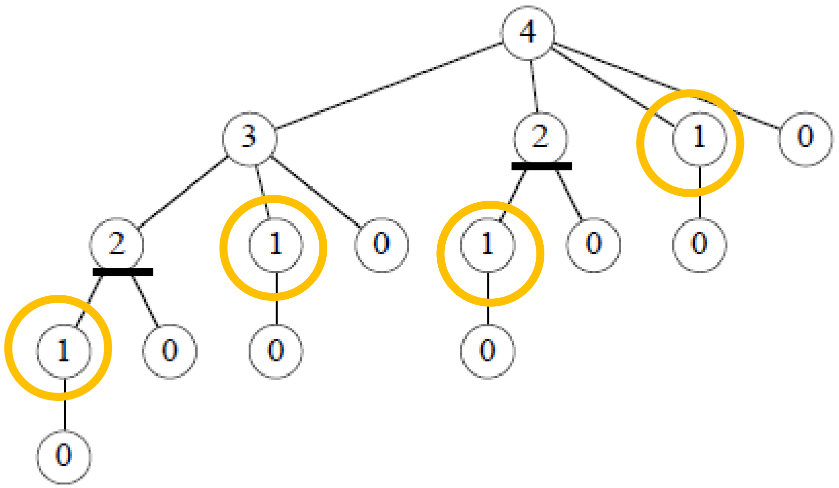
```
CUT-ROD$(p, n)$
if $n == 0$
        return 0
$q = -\infty$
for $i = 1$ to $n$
        $q = \max\{q, p[i] + \text{CUT-ROD}(p, n - i)\}$
return $q$
```

// q is the max revenue

$q = \max\{p_i + \text{CUT-ROD}(p, n - i) : 1 \le i \le n\}$.

Implementation v1: Recursive Top-Down

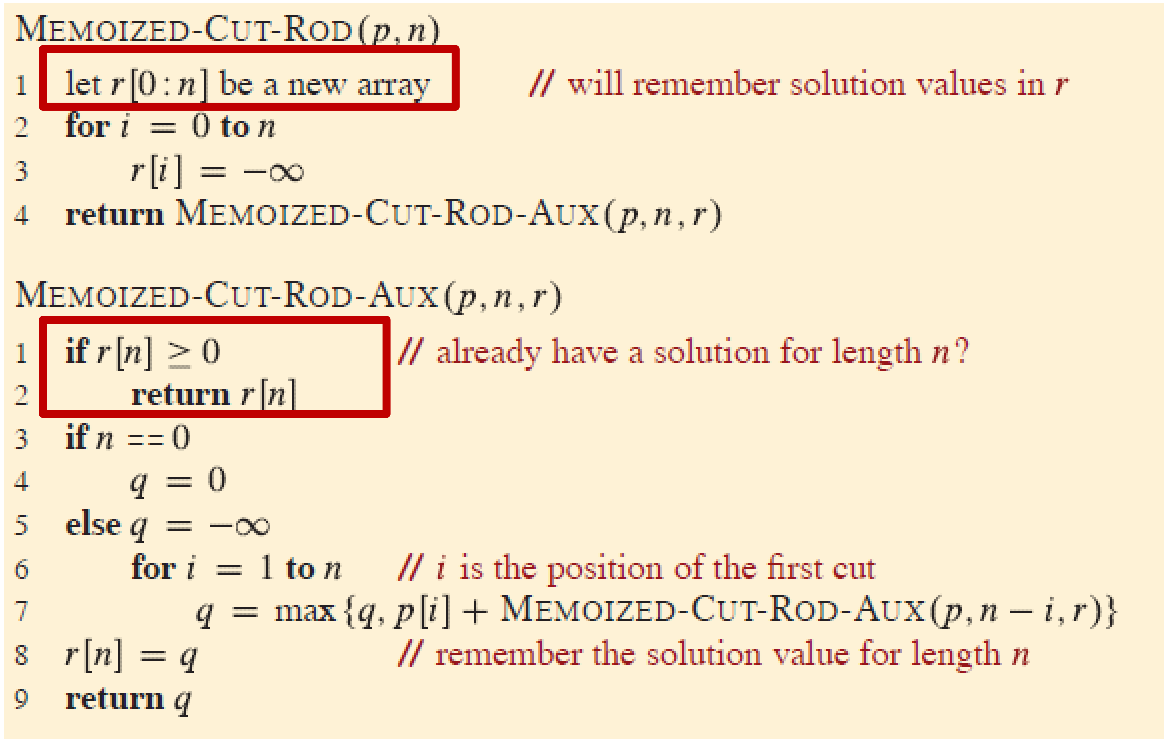
Limitation: Lots of repeated subproblems. Number of calls to CUT-ROD grows exponentially 2^n.



Tree of recursive calls for length n=4

Implementation v2: Top-down (Memoization)

Write the recursion as normal, but store the result of the recursive calls, and if we need the result in a future recursive call, we can use the precomputed value.



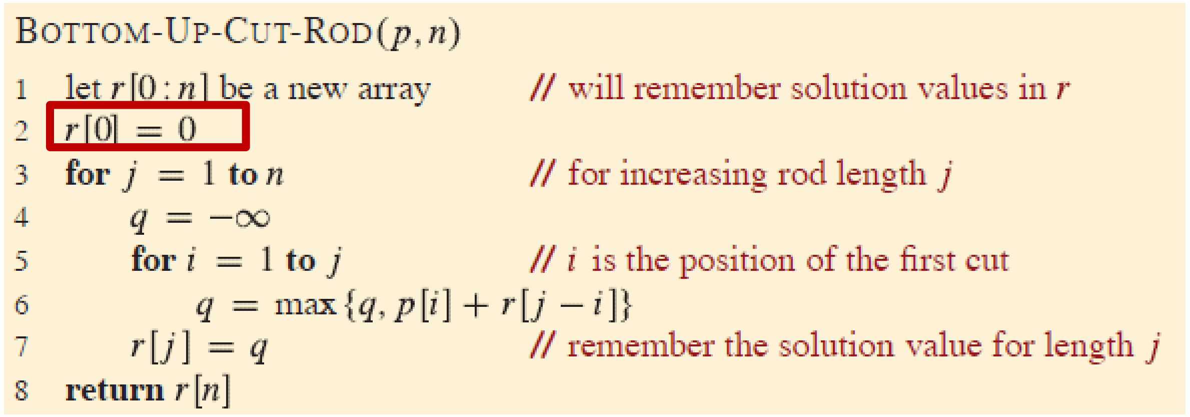
```
MEMOIZED-CUT-ROD$(p, n)$
let $r[0:n]$ be a new array        // will remember solution values in $r$
for $i = 0$ to $n$
    $r[i] = -\infty$
return MEMOIZED-CUT-ROD-AUX$(p, n, r)$

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 $q$
```

Runtime: $\Theta(n^2)$. Each subproblem is solved exactly once, and it solves subproblems for sizes 1,…,n. To solve a subproblem of size n, for loop iterates n times.

In total O(1)+O(2)+…O(n) => $\frac{n(n+1)}{2}$

Implementation v3: Bottom-Up (Tabulation)

Compute the solutions for smaller rods first, knowing that they will later be used to compute the solutions for larger rods. The answer will once again be stored in $r[n]$.



```
BOTTOM-UP-CUT-ROD(p, n)
let r[0 : n] be a new array         // will remember solution values in r
$\boxed{r[0] = 0}$
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 $r[n]$
```

Runtime: $\Theta(n^2)$, double for loop.

Don’t have to keep a recursive call stack.
Many people prefer bottom-up approach in implementing DP.

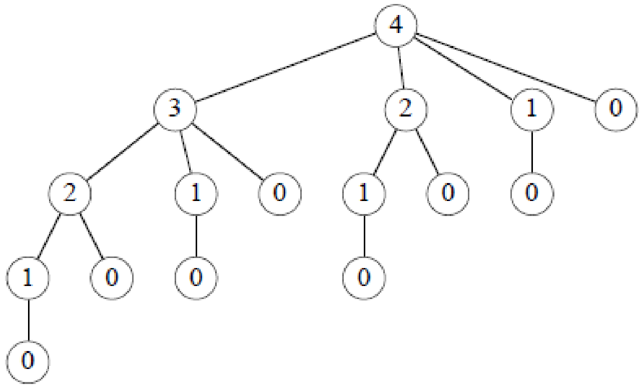
Subproblem Graphs

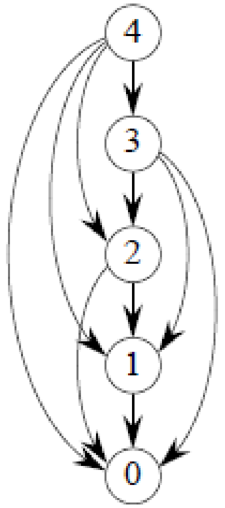
To understand the subproblems in DP procedure.

Directed graph:

- One vertex for each distinct subproblem
- Directed edge (x, y) if computing an optimal solution to subproblem x *directly* requires knowing an optimal solution to subproblem y (sub-step).

Total running time is linear in number of vertices (# of subproblems) and edges (for each subproblem).

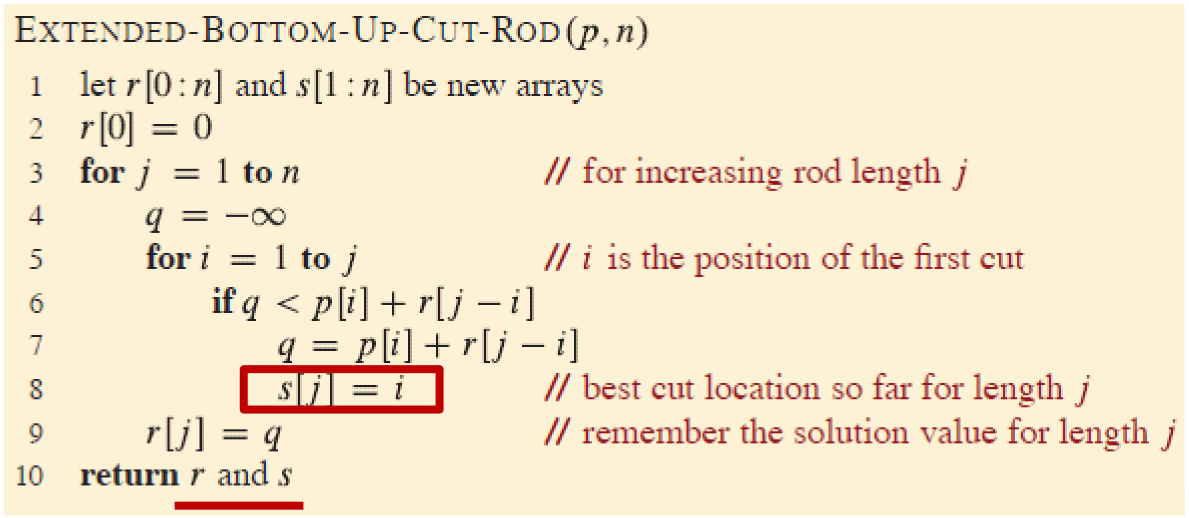




Subproblem graph for rod-cutting with n=4

Reconstructing a Solution

Extend the bottom-up DP approach to record not just optimal revenue, but optimal choices. Save the first cut made in an optimal solution for a problem of size i in $s[i]$.



```
EXTENDED-BOTTOM-UP-CUT-ROD($p, n$)
let $r[0:n]$ and $s[1:n]$ be new arrays
$r[0] = 0$
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
        if $q < p[i] + r[j-i]$
            $q = p[i] + r[j-i]$
            $\boxed{s[j] = i}$           // best cut location so far for length $j$
    $r[j] = q$                        // remember the solution value for length $j$
return $\underline{r \text{ and } s}$
```

i	r_i	optimal solution	s[i]
1	1	1 (no cuts)	1
2	5	2 (no cuts)	2
3	8	3 (no cuts)	3
4	10	$2 + 2$	2
5	13	$2 + 3$	2
6	17	6 (no cuts)	6
7	18	$1 + 6$ or $2 + 2 + 3$	1
8	22	$2 + 6$	2

Reconstructing a Solution

Extend the bottom-up DP approach to record not just optimal revenue, but optimal choices. Save the first cut made in an optimal solution for a problem of size i in $s[i]$.

```
PRINT-CUT-ROD-SOLUTION(p, n)
1  (r, s) = EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
2  while n > 0
3      print s[n]          // cut location for length n
4      n = n - s[n]        // length of the remainder of the rod
```

i	r_i	optimal solution	s[i]
1	1	1 (no cuts)	1
2	5	2 (no cuts)	2
3	8	3 (no cuts)	3
4	10	$2 + 2$	2
5	13	$2 + 3$	2
6	17	6 (no cuts)	6
7	18	$1 + 6$ or $2 + 2 + 3$	1
8	22	$2 + 6$	2

Example: n=8, print solution 2, then 6, finish.

Classical DP Problems

- Rod cutting
- **Matrix-chain multiplication**

Matrix-Chain Multiplication

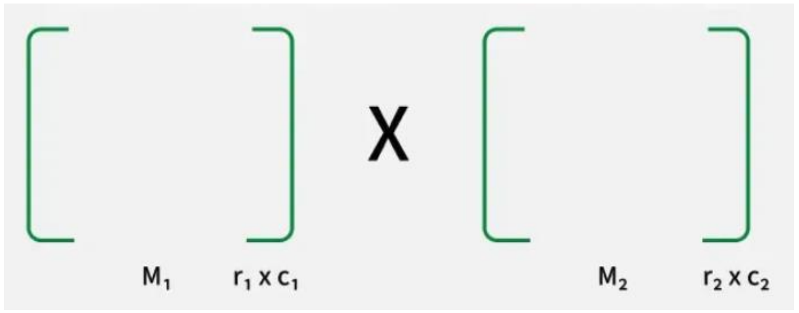
Given: n matrices $M_1, M_2, \cdots, M_n$

Goal: Find the cheapest order to compute $M_1 M_2 \cdots M_n$. Matrix multiplication is associative, so the order of operations changes the cost.

Example: To compute $VWXZY$, we could multiply

$V((WX)(YZ))$ or $\left(V\left(W(XY)\right)\right)Z$

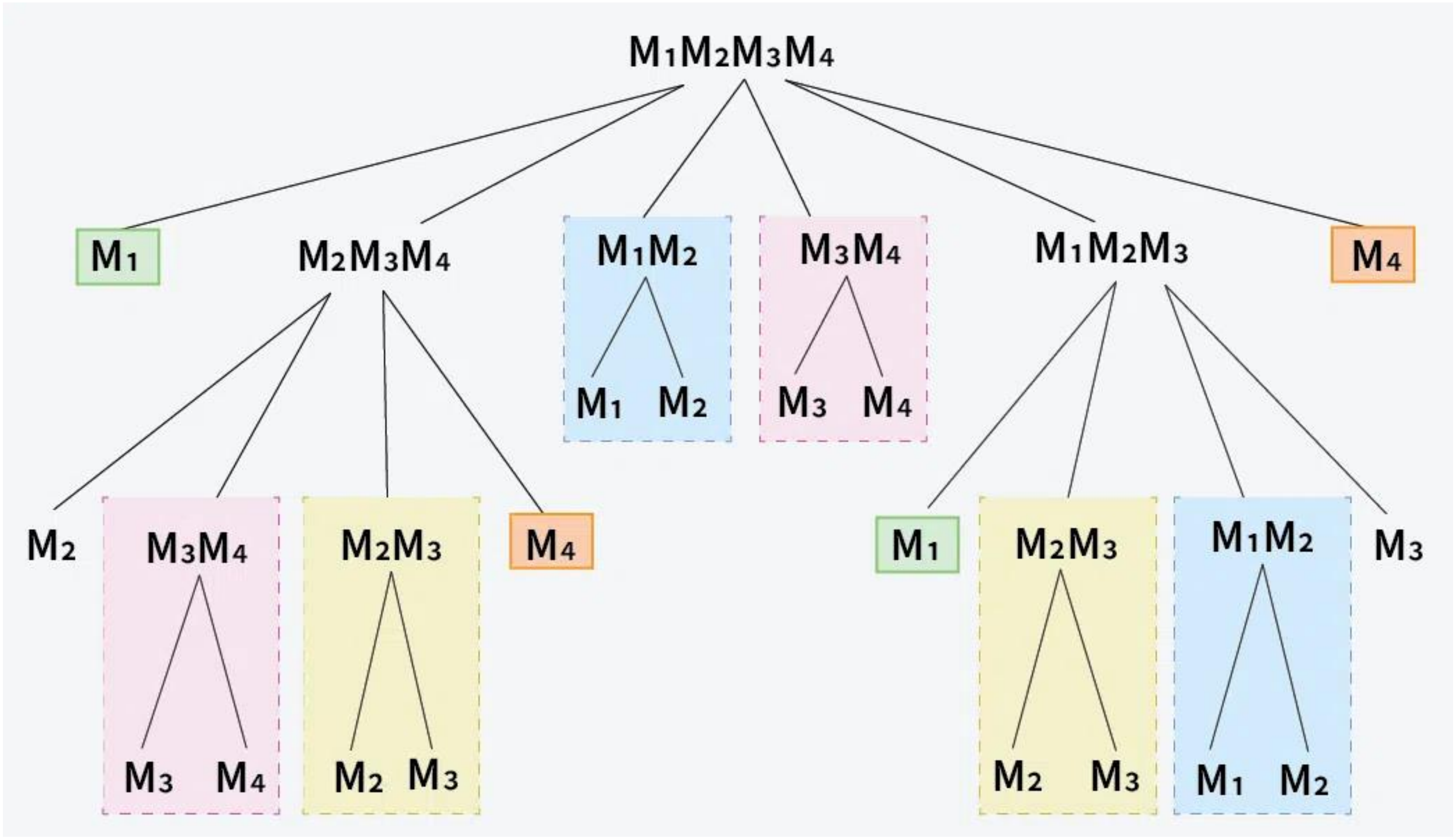
Assumption: M_1 and M_2 can multiply when $c_1 = r_2$, M_1 x M_2 takes $r_1 \times c_2$ x c_1 time.



Result size: $r_1 \times c_2$; each entry requires c_1 operations.

Subproblems: $C(i, j)$ is the time to compute $M_i M_{i+1} \cdots M_j$

Problem Explaination



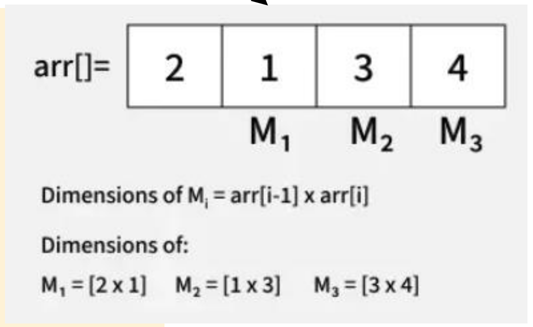
Bottom-up (Tabulation) Solution

Observation: If the last multiplication in optimal solution is

$$(M_i \cdots M_k)(M_{k+1} \cdots M_j)$$

Then, $C(i,j) = \min(\, C(i,j),\ C(i,k) + C(k+1,j) + m_i m_{k+1} m_j)$

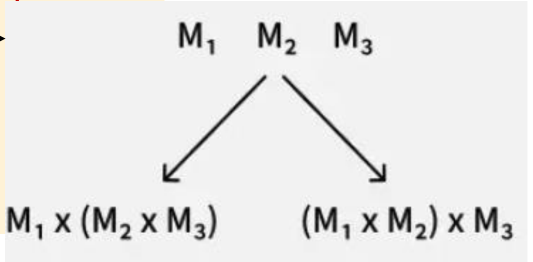
```
MATRIX-CHAIN-ORDER($p, n$)    // p is arr[] for dimension, n matrices
let $m[1:n, 1:n]$ and $s[1:n-1, 2:n]$ be new tables
for $i = 1$ to $n$                      // chain length 1
    $m[i,i] = 0$
for $l = 2$ to $n$                      // $l$ is the chain length
    for $i = 1$ to $n - l + 1$          // chain begins at $A_i$
        $j = i + l - 1$                 // chain ends at $A_j$
        $m[i,j] = \infty$
        for $k = i$ to $j - 1$          // try $A_{i:k} A_{k+1:j}$
            $q = m[i,k] + m[k+1,j] + p_{i-1} p_k p_j$   // suboptimal structure
            if $q < m[i,j]$
                $m[i,j] = q$            // remember this cost
                $s[i,j] = k$            // remember this index
return $m$ and $s$
```



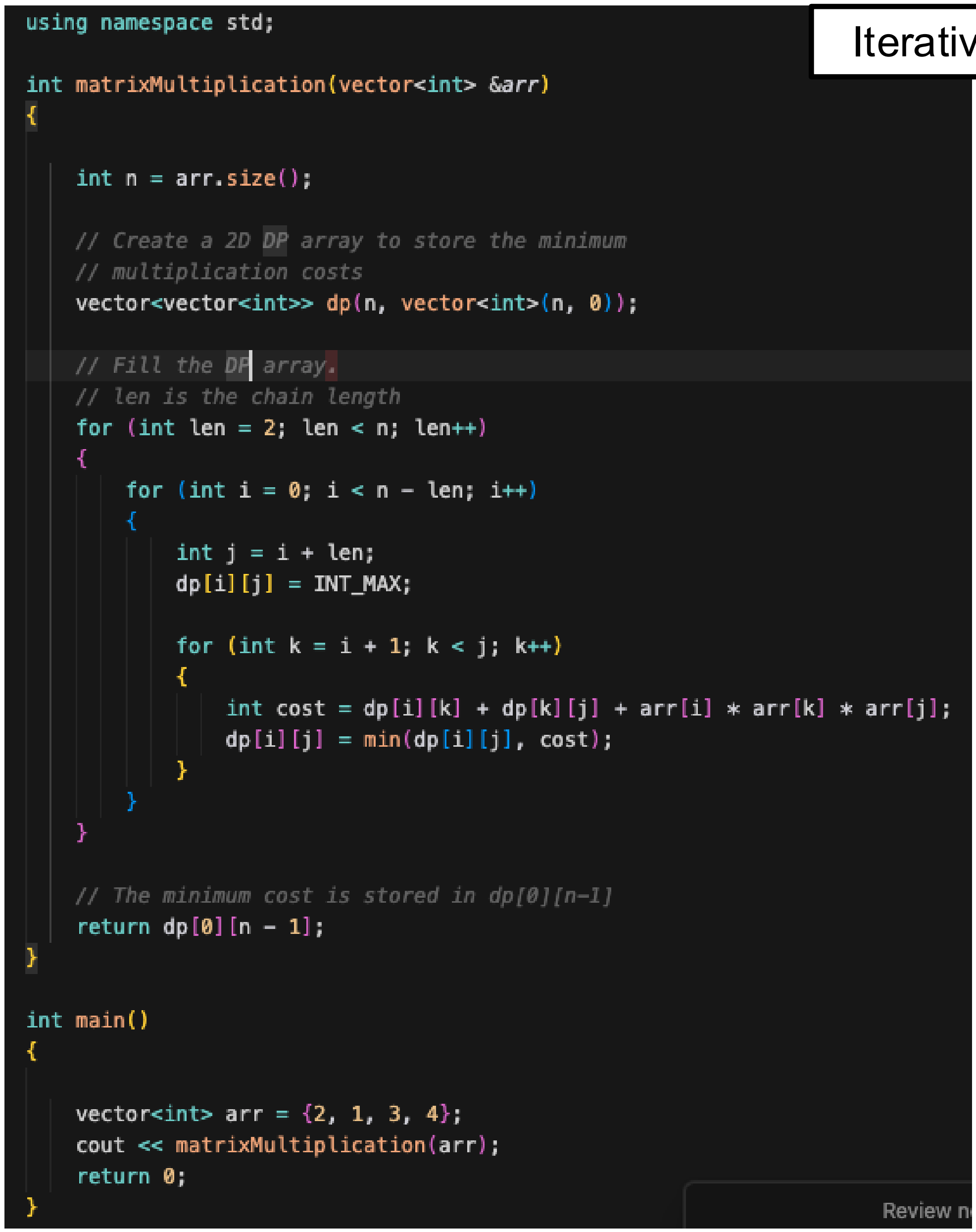
Dimensions of M_i = arr[i-1] x arr[i]

Dimensions of:

M_1 = [2 x 1] M_2 = [1 x 3] M_3 = [3 x 4]



C++ Implementation v1: Bottom-up



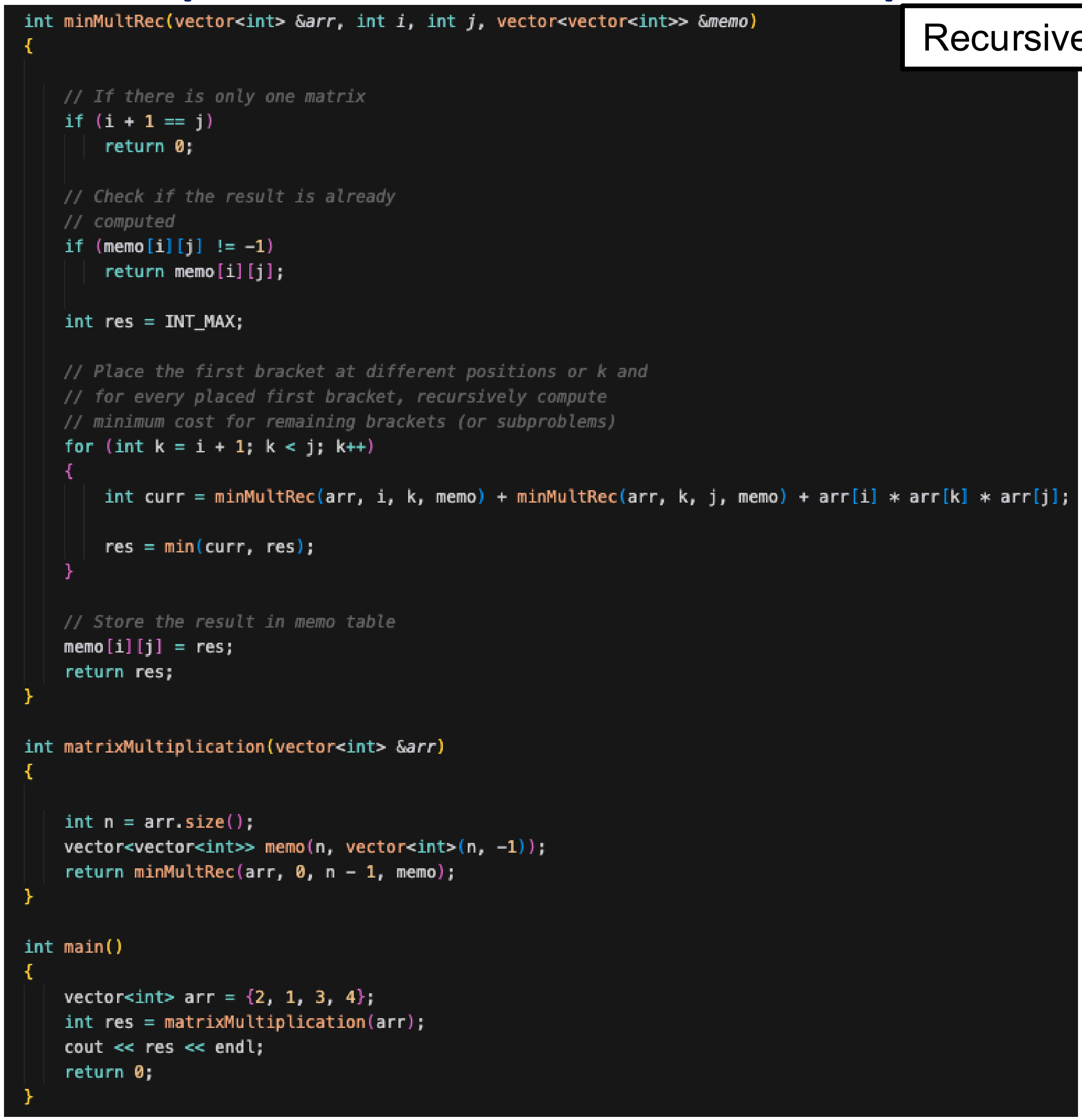
```cpp
using namespace std;

int matrixMultiplication(vector<int> &arr)
{

    int n = arr.size();

    // Create a 2D DP array to store the minimum
    // multiplication costs
    vector<vector<int>> dp(n, vector<int>(n, 0));

    // Fill the DP array.
    // len is the chain length
    for (int len = 2; len < n; len++)
    {
        for (int i = 0; i < n - len; i++)
        {
            int j = i + len;
            dp[i][j] = INT_MAX;

            for (int k = i + 1; k < j; k++)
            {
                int cost = dp[i][k] + dp[k][j] + arr[i] * arr[k] * arr[j];
                dp[i][j] = min(dp[i][j], cost);
            }
        }
    }

    // The minimum cost is stored in dp[0][n-1]
    return dp[0][n - 1];
}

int main()
{

    vector<int> arr = {2, 1, 3, 4};
    cout << matrixMultiplication(arr);
    return 0;
}
```

Review n

Iterative approach

C++ Implementation v2: Top-down

Recursive approach



```
int minMultRec(vector<int> &arr, int i, int j, vector<vector<int>> &memo)
{

    // If there is only one matrix
    if (i + 1 == j)
        return 0;

    // Check if the result is already
    // computed
    if (memo[i][j] != -1)
        return memo[i][j];

    int res = INT_MAX;

    // Place the first bracket at different positions or k and
    // for every placed first bracket, recursively compute
    // minimum cost for remaining brackets (or subproblems)
    for (int k = i + 1; k < j; k++)
    {
        int curr = minMultRec(arr, i, k, memo) + minMultRec(arr, k, j, memo) + arr[i] * arr[k] * arr[j];

        res = min(curr, res);
    }

    // Store the result in memo table
    memo[i][j] = res;
    return res;
}

int matrixMultiplication(vector<int> &arr)
{

    int n = arr.size();
    vector<vector<int>> memo(n, vector<int>(n, -1));
    return minMultRec(arr, 0, n - 1, memo);
}

int main()
{
    vector<int> arr = {2, 1, 3, 4};
    int res = matrixMultiplication(arr);
    cout << res << endl;
    return 0;
}
```

Problem Solving Steps

- General Guideline
- Revisit Bellman-Ford for SSSP

General Guideline

General guideline (not always):

1. The problem will be asking for an optimal value (max or min) of something or the number of ways to do something.
    - What is the minimum cost of doing ...
    - What is the maximum profit of ...
    - How many ways are there to ...
    - What is the longest possible ...
2. At each step, you need to make a "decision", and decisions affect future decisions.
    - A decision could be picking between two elements
    - Decisions affecting future decisions could be something like "if you take an element x, then you can't take an element y in the future"

Mastering the pattern - define subproblems, write a recurrence, then fill a table or use memorization - gives you a reusable toolkit.

Revisit Bellman-Ford Algorithm (DP)

Def: Let $d(v, i)$ be the length of the shortest path with at most i edges from source s to vertex v.

Let us characterize $d(v, i)$.

Case 1: $d(v, i)$ path has less than i edges.

- Then, $d(v, i) = d(v, i - 1)$.

Case 2: $d(v, i)$ path has exactly i edges.

- Let $s, v_1, v_2, \dots, v_{i-1}, v$ be the $d(v, i)$ path with i edges.
- Then, $s, v_1, \dots, v_{i-1}$ must be the shortest s - v_{i-1} path with at most $i - 1$ edges. So,

$$d(v, i) = d(v_{i-1}, i - 1) + w_{v_{i-1}, v}$$

Revisit Bellman-Ford Algorithm (DP)

Def: Let $d(v, i)$ be the length of the shortest path with at most i edges from source s to vertex v.

$$\boldsymbol{d(v,i)} = \begin{cases} 0, & \text{if } v = s \\ \infty, & \text{if } v \neq s, i = 0 \\ \min\limits_{(u,v)\in E} (d(v,i), \boldsymbol{d(u, i-1)} + w_{u,v}) & \end{cases}$$

So, for every v, $d(v, ?)$ is the shortest path from s to v.

But how long do we have to run?

Since G has no negative cycle, it has at most $n - 1$ edges. So, $d(v, n-1)$ is the answer.

Revisit Bellman-Ford Algorithm (DP)

- It breaks down shortest-path problem into subproblems.
- It stores intermediate results to avoid repetitive work.
- It follows Bellman Equation:

$$\boldsymbol{d}(\boldsymbol{v}) = \min_{(u,v)\in E}(d(v), \boldsymbol{d}(\boldsymbol{u}) + w(u, v))$$

where:

- $d(v)$ is the shortest known distance to vertex v,
- $w(u, v)$ is the weight of edge (u, v).

Summary

- When DP applies: A problem has **overlapping subproblems** (the same smaller instances appear many times) and **optimal substructure** (an optimal solution is built from optimal solutions to subproblems).
- Top-down (memoization) uses recursive approach with caching, bottom-up (tabulation) uses iterative approach storing solutions.
- Typical steps: (1) Define subproblems clearly. (2) Write a recurrence (and base cases). (3) Choose memoization or tabulation and implement. (4) Often you can reduce space (e.g. keep only the last row or a few entries) once the recurrence is fixed.
- Running time depends on (# of subproblems overall) x (# of choices).

Additional Materials

- Make full use of the mind maps, flashcards, quizzes, and podcast generated by NotebookLM to help organize and deepen your understanding, e.g., using the mind map as a starting point to guide further discussion with the chat.
- Leetcode