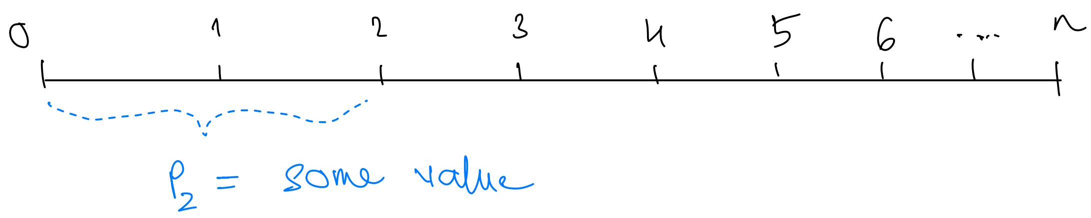
DYNAMIC PROGRAMMING

Rod Cutting

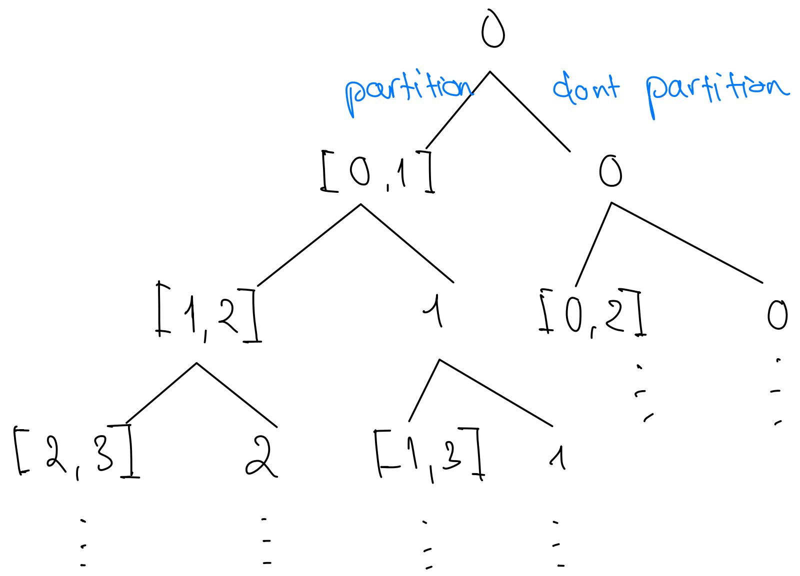
Given a rod length n, and price p_i be the value of rod length i (Constraint $1 \leq i \leq n$)



Goal: Find optimal way to cut a given rod of length n, that maximise the pieces' prices.

Brute Force Approach:

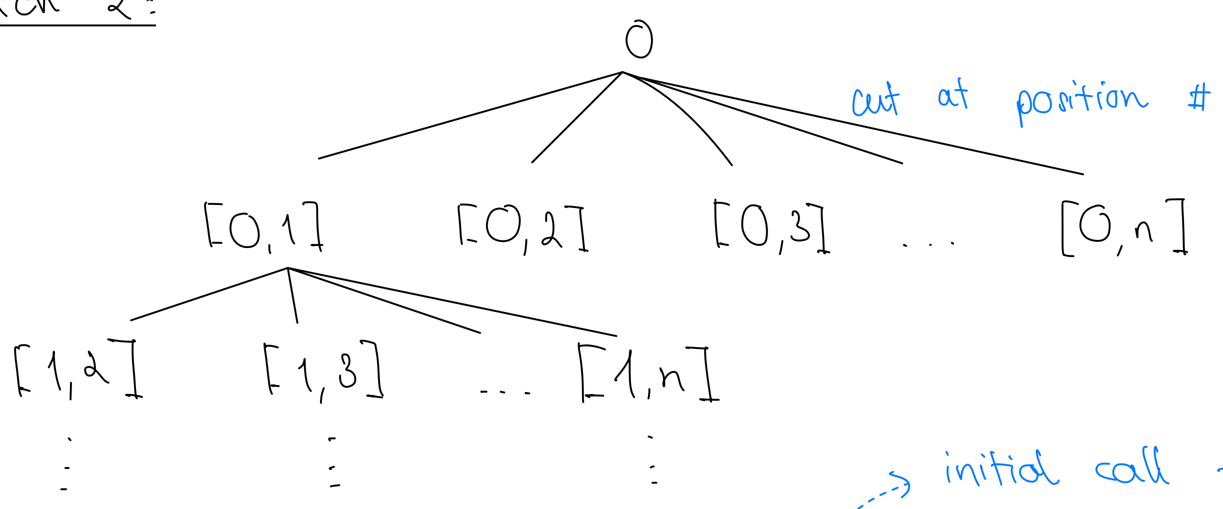
Approach 1:



- Recurrence relation: $T(n) = 2T(n-1) + O(1)$

  $\Rightarrow O(2^n)$

Approach 2:



- Recurrence relation: $T(n) = 1 + \sum_{k=1}^{n-1} T(k)$ → initial call from the root

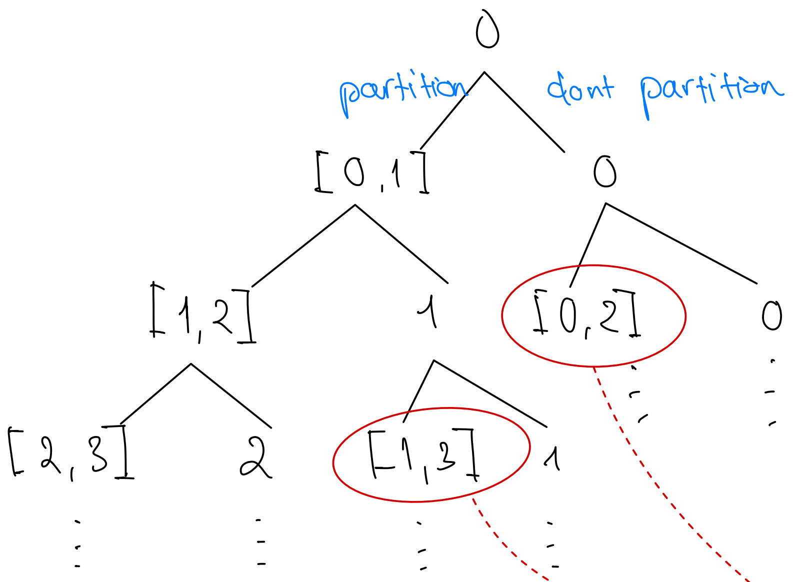
  $\Rightarrow O(2^n)$ < Exercise 14.1-1 >

Approach 3: Using number theory, the number of different ways to cut up the rod corresponds to the "partitions" of n, denoted $p(n)$:

- $p(n) = \Omega(2^{\sqrt{n}})$
- $\ln p(n) \sim \pi\sqrt{2/3} \cdot \sqrt{n}$ < linear >

Dynamic Programming Approach

Avoid repeated work



repeated calculation, solving the same problem of $n = 2$

<u>Solution:</u> Store repeating values in a (hash) table

table =

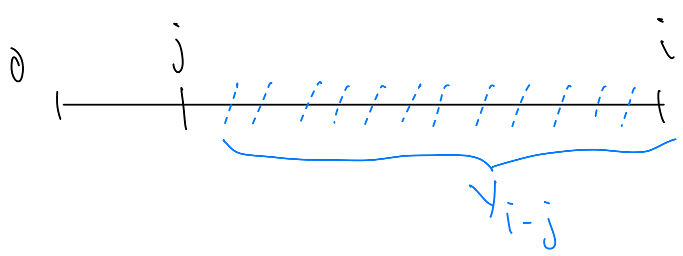
length	optimal_value
1	v_1
2	v_2
⋮	⋮
n	v_n

<u>Formally:</u>

- Let v_i be the optimal value of a rod of length i
- Suppose we are on the way up the recursive tree, and we are trying to determine v_i with the result of $v_0, \ldots, v_{i-1}$ already determined, we can determine v_i by:

$$v_i = \underset{1 \leq j \leq i}{\arg\max} \left(p_j + v_{i-j} \right)$$

where j is the length of one piece we decided to cut on the way down the decision tree, so:
- p_j is price of that piece
- v_{i-j} is previous computed optimal value



Pseudo Code (Top-down approach)

```
Top-down (p, n):
    let v[0:n] be array
    for i = 0 → n:                                   init memo array
        v[i] = -∞
    return Dfs(p, n, v)

Dfs(p, n, v):
    if n == 0:                                       base case
        return 0
    if v[n] ≥ -∞ :                                   retrieve from memo
        return v[n]
    res = -∞
    for j = 1 → n :
        res = max{ res, p[j] + Dfs(p, n-j, v) }      recursive
    v[n] = res                                       memoization
    return v[n]
```

Pseudo Code (Bottom-up approach)

```
Bottom-Up (p, n):
    let v[0:n] be new array
    v[0] = 0
    for i = 1 → n:
        res = -∞
        for j = 1 → i :
            res = max{ res, p[j] + v[i-j] }
        v[j] = res
    return v[n]
```

⇒ From the pseudo code, we can see that for each subproblem size $i : 1 \to n$, the code runs $j : 1 \to i$, created a nested loop structure. Hence the complexity is $O(n^2)$

Matrix-chain Multiplication: (Burst Balloons Leetcode # 312)

- Given a sequence of matrices $(A_1, A_2, A_3, \dots, A_n)$.
  Get the product of these matrices with the lowest cost possible
- The problem can also be intepreted as: Find the best way to parenthesize the sequence $(A_1, A_2, A_3, \dots, A_n)$ so that the product cost is the lowest. This is because:

$$(A_1 . A_2) A_3 = A_1 (A_2 . A_3)$$

- For example:
  Consider 3 matrices $A_1 \in \mathbb{R}^{10 \times 100}$, $A_2 \in \mathbb{R}^{100 \times 5}$, $A_3 \in \mathbb{R}^{5 \times 50}$
  The cost 2 ways of calculating the product is
  - $(A_1 . A_2) . A_3$ costs $10.100.5 + 10.5.50 = 7500$ scalar multiplications
  - $A_1 (A_2 . A_3)$ costs $100.5.50 + 10.100.50 = 75\,000$ scalar multiplications (10 times more)

Formally defined problem:

Let
- $A_{i:j}$ be the resulting matrix of multiplying sequence $(A_i, A_{i+1}, \dots A_j)$
- $m[i,j]$ be minimum cost of multiplying the sequence $(A_i, A_{i+1}, A_{i+2}, \dots A_j)$, where $1 \leq i \leq j \leq n$
- k be the optimal way to split $(A_i, A_{i+1}, \dots A_j)$ into $(A_i, A_{i+1}, \dots A_k)$ and $(A_k, A_{k+1}, \dots A_j)$.

  In other words, to optimally calculate A_{ij}, we need to also optimally calculate A_{ik} and A_{kj}, then combine them to get A_{ij}

→ Optimal Structure: To optimally parenthesize $(A_i, A_{i+1}, \dots A_j)$. You need to optimally parentheize $(A_i, A_{i+1}, \dots, A_k)$ and $(A_{k+1}, A_{k+2}, \dots, A_j)$ and recursively for the subsequence

⇒ Recursive equation:

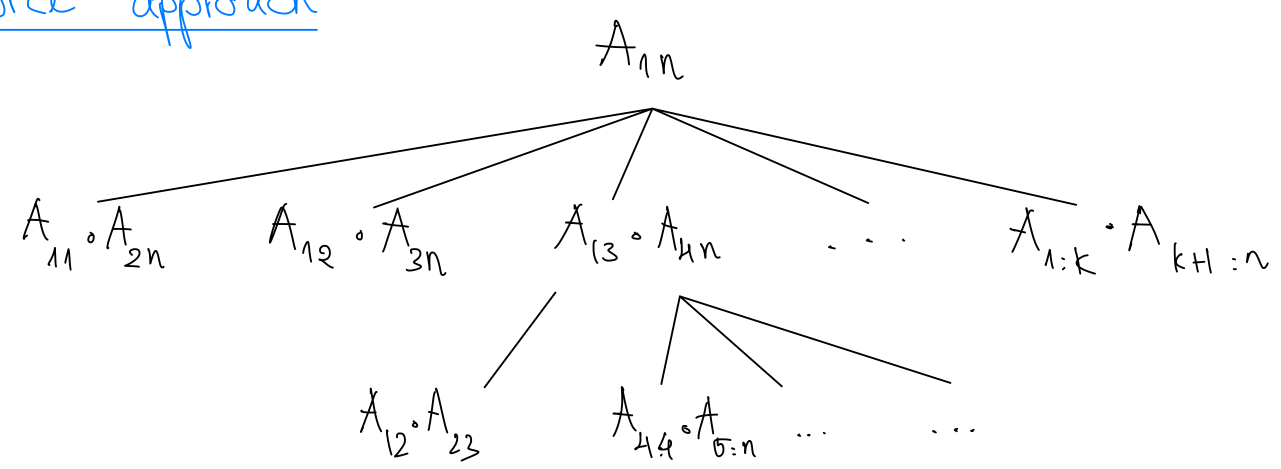
Let p be sequence of dimensions. Example: $p = (\overbrace{10, \underbrace{100,}_{A_1} \overbrace{5,}^{A_2}}^{} \underbrace{50}_{A_3})$

previously computed values

cost of $A_{ik} \cdot A_{kj}$

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

Brute force approach



$\Rightarrow \quad O(2^n)$

Dynamic Programming Approach

Bottom-Up pseudo code

```
Bottom-up (p, n):

    let m[1:n , 1:n] and s[1:n-1 , 2:n] be 2 tables to
                                    store min costs, best k

    for i = 1 → n:
        m[i,j] = 0            # base case: zero cost if theres only 1 matrix

    for l = 2 → n:
        for i = 1 → n-l+1:       slide window size l from
            j = i+l-1            1 to n

            m[i,j] = ∞
            for k = i → j-1:
                cost = m[i,k] + m[k+1,j] + P_{i-1} P_k P_j
                if cost < m[i,j]:
                    m[i,j] = cost      # save cost
                    s[i,j] = k         # save index
    return m and s
```

Complexity:

Time: $O(n^3)$ since we can clearly see it is 3-level deep nested loops

Space: $O(n^2)$ since tables m and s requires ~ $n \times n$ spaces

Elements of Dynamic Programming

Two keys ingredients that an optimization problem must have in order for dynamic programming to apply are Optimal Substructure and Overlapping Subproblems

Optimal substructure

A problem exists optimal substructure if its optimal solution contains within it optimal solutions to subproblems.

Common Pattern in discovering Optimal Structure

1. Show optimal solution includes subproblems:

   You show that to obtains the optimal solution, certain results must be true.

   You dont concern about how to obtain those results.

2. Show the structure of the subproblems (in relation to the optimal solution)

   Given that these results, you determine the structure of the subproblems (how many branches does the recursive tree split into?)

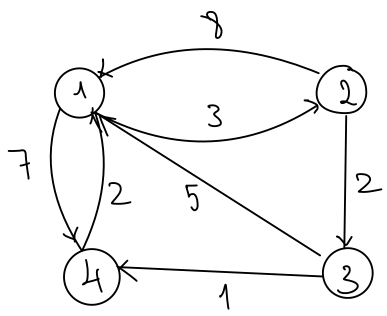
3. Show that optimal solution requires optimal solutions from subproblems (prove by contradiction)

   You show that the solutions to the subproblems must themselves be optimal.

Floyd-Warshall algorithm

Given digraph $G = (V, E)$. Let $|V| = n$

Find all pairs of shortest path, denote $f(i,j)$



Some possible approachs:

- Brute force:

  for each pair (i,j), compute the shortest path by comparing:

$$f(i,j) = \min \begin{cases} w(i,j) \\ w(i,k_1) + w(k_1,j) \\ w(i,k_1) + w(k_1,k_2) + w(k_2,j) \\ \vdots \\ w(i,k_1) + w(k_1,k_2) + \dots + w(k_{n-2}, j) \end{cases}$$

  $\Rightarrow$ There are $O(n^2)$ pairs (i,j), and the comparing work takes $O(n^n)$

  $\Rightarrow$ Total cost $O(n^{2n}) \sim O(n^n)$

- Greedy method, Dijkstra algorithm

  Frame the problem as, "for each source v_i, find the shortest paths to all other vertices"

  $\Rightarrow$ Costs $O(\underbrace{n}_{\text{no. of sources}} \cdot \underbrace{n^2}_{\text{Dijkstra complexity}}) \sim O(n^3)$

Now we present a dynamic programming approach called Floyd-Warshall.

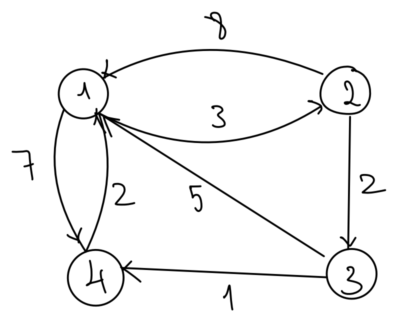
High level explanation:

Let $\begin{cases} k \text{ be a number such that } 0 \leqslant k \leqslant n \\ A^k \text{ denote a matrix with vertex } v_k \text{ such that } x_i \to v_k \to y_j \\ \quad \text{for all pairs } (i,j) \end{cases}$

We will iteratively build $A^0, A^1, A^2, \dots A^n$ s.t the result of matrix A^i is built on the result from A^{i-1}. The base case, A^0 represent all the direct shortest path of any pairs (i,j)

$$A^k[i,j] = \begin{cases} \omega(v_i, v_j) & \text{if } k=0 \\ \min\left\{ A^{k-1}[i,j],\ A^{k-1}[i,k] + A^{k-1}[k,j] \right\} & \text{otherwise} \end{cases}$$

Example: Consider this example



$$A^0 = \begin{array}{cc} & \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{bmatrix} 0 & 3 & \infty & 7 \\ 8 & 0 & 2 & \infty \\ 5 & \infty & 0 & 1 \\ 2 & \infty & \infty & 0 \end{bmatrix} \end{array} \longrightarrow A^1 = \begin{array}{cc} & \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{bmatrix} 0 & 3 & \infty & 7 \\ 8 & 0 & 2 & 15 \\ 5 & 8 & 0 & 1 \\ 2 & 5 & \infty & 0 \end{bmatrix} \end{array}$$

$\dashrightarrow \min\{A^0_{ij},\ A^0_{ik} + A^0_{kj}\}$

$$A^2 = \begin{array}{cc} & \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{bmatrix} 0 & 3 & 5 & 7 \\ 8 & 0 & 2 & 15 \\ 5 & 8 & 0 & 1 \\ 2 & 5 & 7 & 0 \end{bmatrix} \end{array} \longrightarrow A^3 = \begin{array}{cc} & \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{bmatrix} 0 & 3 & 5 & 6 \\ 7 & 0 & 2 & 3 \\ 5 & 8 & 0 & 1 \\ 2 & 5 & 7 & 0 \end{bmatrix} \end{array}$$

$$A^4 = \begin{array}{cc} & \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{bmatrix} 0 & 3 & 5 & 6 \\ 5 & 0 & 2 & 3 \\ 3 & 6 & 0 & 1 \\ 2 & 5 & 7 & 0 \end{bmatrix} \end{array}$$

$\dashrightarrow$ shortest path $f(2,4)$

Time complexity of Floyd-Warshall

We need to build $|V|$ matrices, each matrix costs $O(|V|^2)$

Total costs is $O(|V| \cdot |V|^2)$

$\sim O(n^3)$