Algorithms & Data Structures

Lecture 09
Dynamic Programming

Giovanni Bacci
giovbacci@cs.aau.dk

Outline

- Algorithmic Principle: Dynamic Programming
- The Rod Cutting Problem
- The Matrix-chain Multiplication Problem
- General elements of Dynamic Programming

Intended Learning Goals

KNOWLEDGE

- Mathematical reasoning on concepts such as recursion, induction, concrete and abstract computational complexity
- Data structures, algorithm principles e.g., search trees, hash tables, dynamic programming, divide-and-conquer
- Graphs and graph algorithms e.g., graph exploration, shortest path, strongly connected components.

SKILLS

- Determine abstract complexity for specific algorithms
- Perform complexity and correctness analysis for simple algorithms
- Select and apply appropriate algorithms for standard tasks

COMPETENCES

- Ability to face a non-standard programming assignment
- Develop algorithms and data structures for solving specific tasks
- Analyse developed algorithms

Dynamic Programming

- Dynamic programming, like divide-and-conquer, solves problems by **combining the solutions of subproblems**
- The word “programming” refers to a **tabular-method**
- It applies **when the subproblems overlap** (i.e., when subproblems share subproblems)
- Typically applied to **optimisation problems:**
  - Problems that admit many possible solutions
  - Each solution has a value (or cost)
  - One want find **a optimal solution** with max value (or min cost)

Quiz

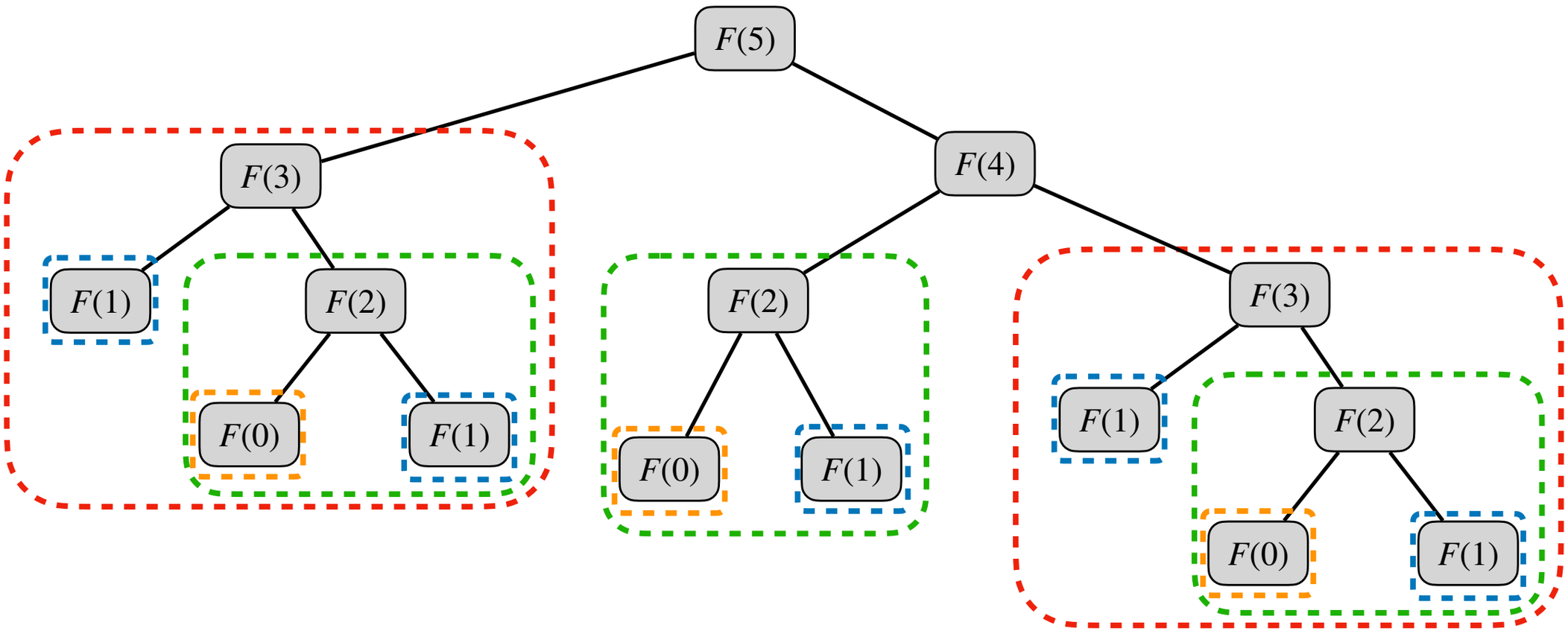
- Did we already see one example of a recursive definition with overlapping subproblems?

- How efficient was its naive implementation of the recursion?

- How did we improve its running-time?

Answer: the Fibonacci Recurrence

$$F(n) = \begin{cases} 1 & \text{if } n \in \{0, 1\}, \\ F(n-1) + F(n-2) & \text{if } n > 1 \end{cases}$$



The Fibonacci Recurrence

$$F(n) = \begin{cases} 1 & \text{if } n \in \{0, 1\}, \\ F(n-1) + F(n-2) & \text{if } n > 1 \end{cases}$$

```
FIB-REC(n)
1  if n < 2
2        return 1
3  else
4        return FIB-REC(n − 1) + FIB-REC(n − 2)
```

Repeating the work for all overlapping sub-problems leads to an exponential running time $O(2^n)$

```
FIB-ITER(n)
1  Initialise the array F[1 . . n + 1]
2  F[1] = 1
3  F[2] = 1
4  for i = 3 to n + 1
5        F[i] = F[i − 2] + F[i − 1]
6  return F[n + 1]
```

Stores the solutions of the sub-problems in the table $F[1..n+1]$ to reuse them when needed.
It cuts the running time to $\Theta(n)$

Dynamic Programming

When developing a dynamic-programming algorithm we follow a sequence of four steps:

1. Characterise the structure of an optimal solution
2. Recursively define the value of an optimal solution
3. Compute the value of an optimal solution, typically in a bottom-up fashion
4. Construct an optimal solution from computed information

Rod Cutting

Rod-cutting problem

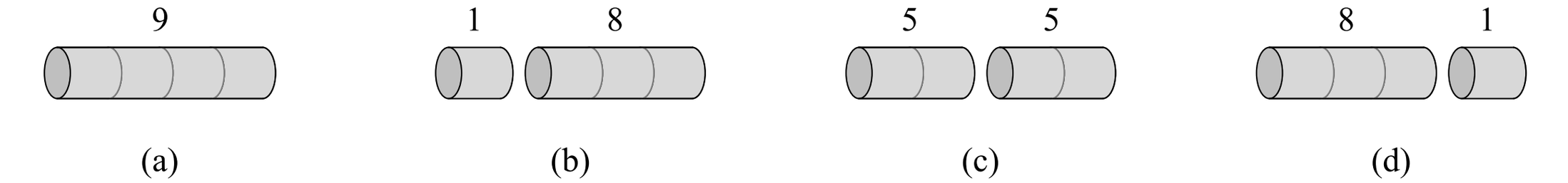
- Serling enterprises buys long steel rods and cuts them into shorter rods, which it then sells.
- Assume that cutting the rod is free.
- For $i = 1,2,\ldots,n$ we denote by p_i the price that the company charges for an i inch long rod.

The rod-cutting problem: given a rod of length n inches and a table of prices p_i for $i = 1,2,\ldots,n$, determine the maximum revenue r_n obtainable by cutting the rod and selling the pieces.

Rod-cutting problem

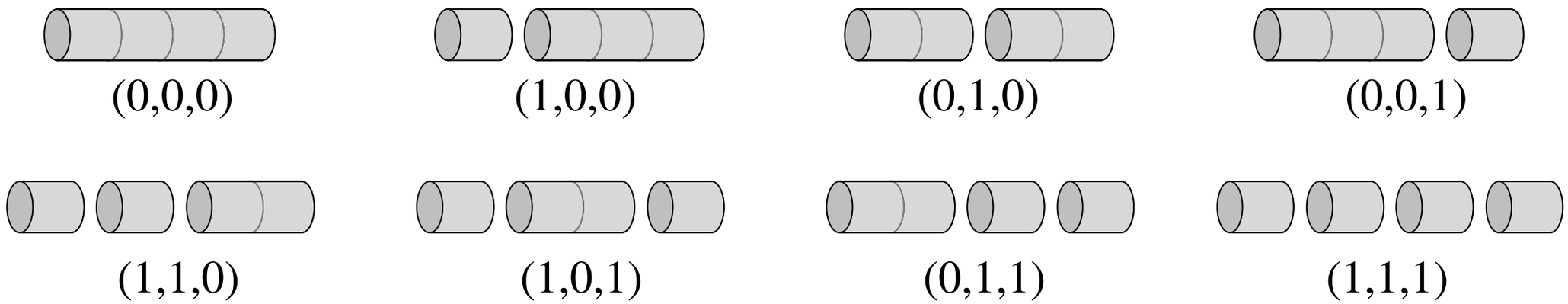
The rod-cutting problem: given a rod of length n inches and a table of prices p_i for $i = 1,2,\ldots,n$, determine the maximum revenue r_n obtainable by cutting the rod an selling the pieces.

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



Naive approach: brute-force

- In general we can can cut a rod in of length n in 2^{n-1} different ways



- A naive approach would require one to check an exponential amount of solutions!

Structure of an optimal solution

- We denote a composition of pieces using additive notation. e.g., 7 = 2 + 2 + 3 indicates that a rod of length 7 is cut into 3 pieces — 2 of length 2 and one of length 3

- In a rod of length n cut into k pieces ($1 \leq k \leq n$) of length respectively $i_1, \ldots, i_k$ is indicated by $n = i_1 + i_2 + \cdots + i_k$

- An the associated revenue is $r = p_{i_1} + p_{i_2} + \cdots + p_{i_k}$
- We can frame the optimal value r_n of a rod of length n in terms of the optimal decompositions each cut

$$r_n = \max(p_n, \max_{i=1..n} r_i + r_{n-i})$$

Structure of an optimal solution

- We denote a composition of pieces using additive notation. e.g., 7 = 2 + 2 + 3 indicates that a rod of length 7 is cut into 3 pieces — 2 of length 2 and one of length 3

- In a rod of length n cut into k pieces ($1 \leq k \leq n$) of length respectively $i_1, \ldots, i_k$ is indicated by $n = i_1 + i_2 + \cdots + i_k$

- An the associated revenue is $r = p_{i_1}$
- We can frame the r_n o terms of the optimal decompositions each cut

No cuts.
Sell the rod as it is

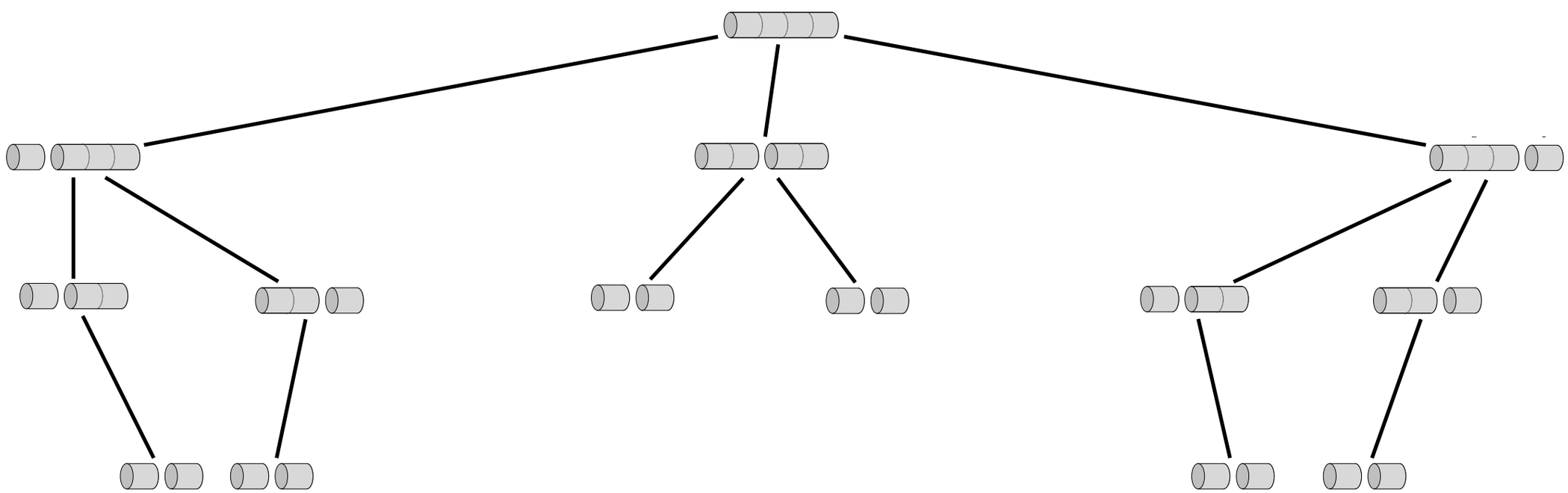
Cut the rod at i inches, and recursively optimally decompose the two parts

$$r_n = \max(p_n, \max_{i=1..n} r_i + r_{n-i})$$

Structure of an optimal solution

- To solve the original problem, we solve smaller problems of the same type, but smaller.
- The overall optimal solution incorporates optimal solutions of the subproblems which can be solved independently: **optimal substructure**

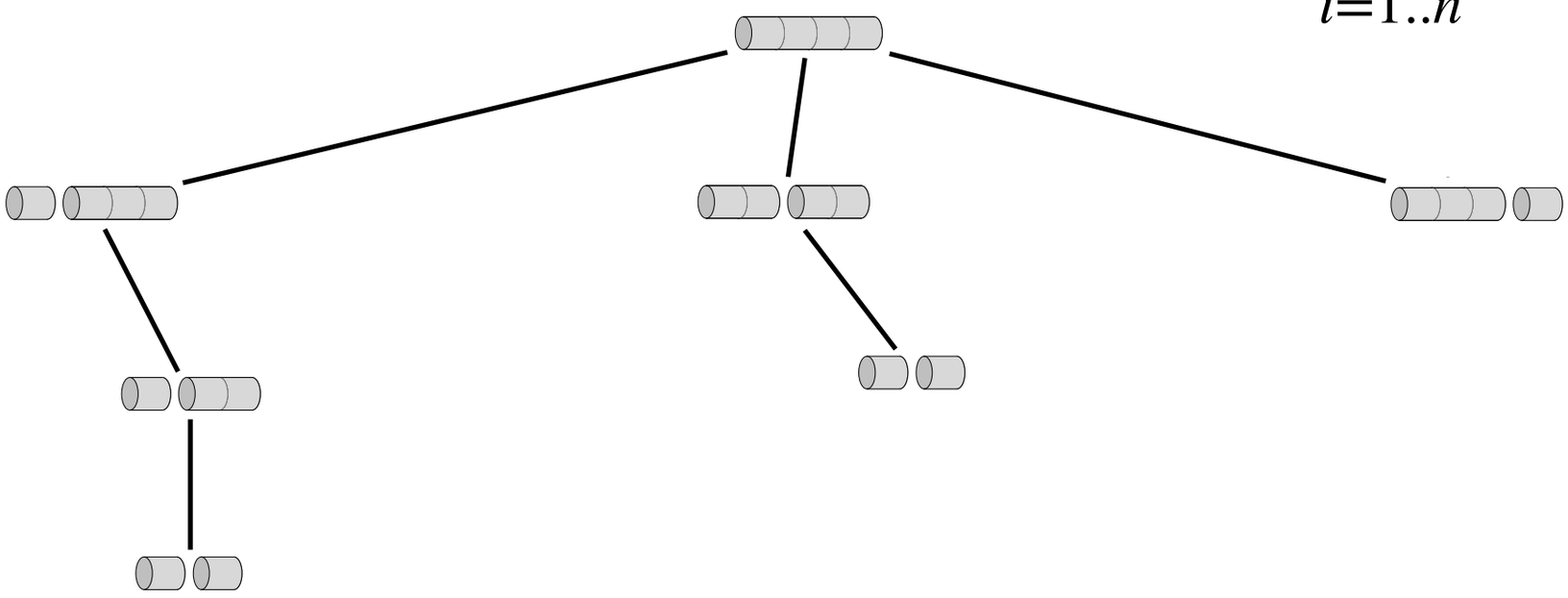
$$r_n = \max(p_n, \max_{i=1..n} r_i + r_{n-i})$$



Structure of an optimal solution

- We can **simplify the decomposition** as follows
  - Cut the road at i inches
  - Do not further cut the left part
  - Recursively cut the second part
- This results in **avoiding some overlapping** in related subproblems
- Some **overlap still remains**

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



D&C implementation

The following pseudocode implements the equation below in a straightforward, top-down, recursive manner

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

```
CUT-ROD(p, n)
1   if n == 0
2       return 0
3   q = -∞
4   for i = 1 to n
5       q = max(q, p[i] + CUT-ROD(p, n - i))
6   return q
```

Cut-Rod: run-time

We analyse the total number of calls made by CUT-ROD when called with its second parameter equal to n

```
CUT-ROD(p, n)
1  if n == 0
2       return 0
3  q = -∞
4  for i = 1 to n
5       q = max(q, p[i] + CUT-ROD(p, n - i))
6  return q
```

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

Root

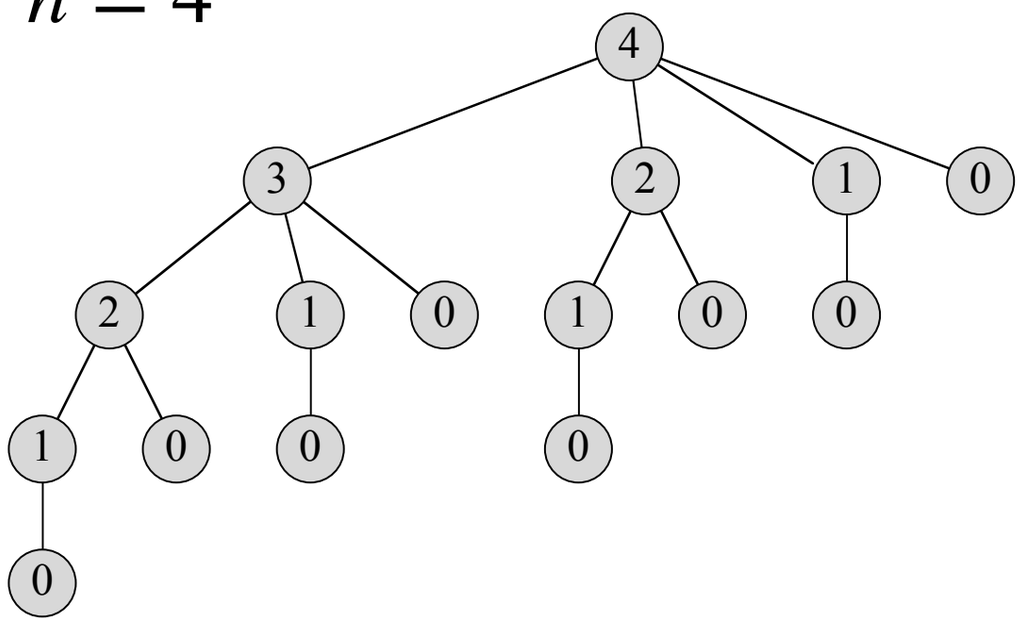
# of calls in the recursive calls

- One can prove (using substitution method) that $T(n) = 2^n$.
- Hence, also the running-time of CUT-ROD is exponential in n

Cut-Rod: run-time

We analyse the total number of calls made by CUT-ROD when called with its second parameter equal to n

$n = 4$



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

Root

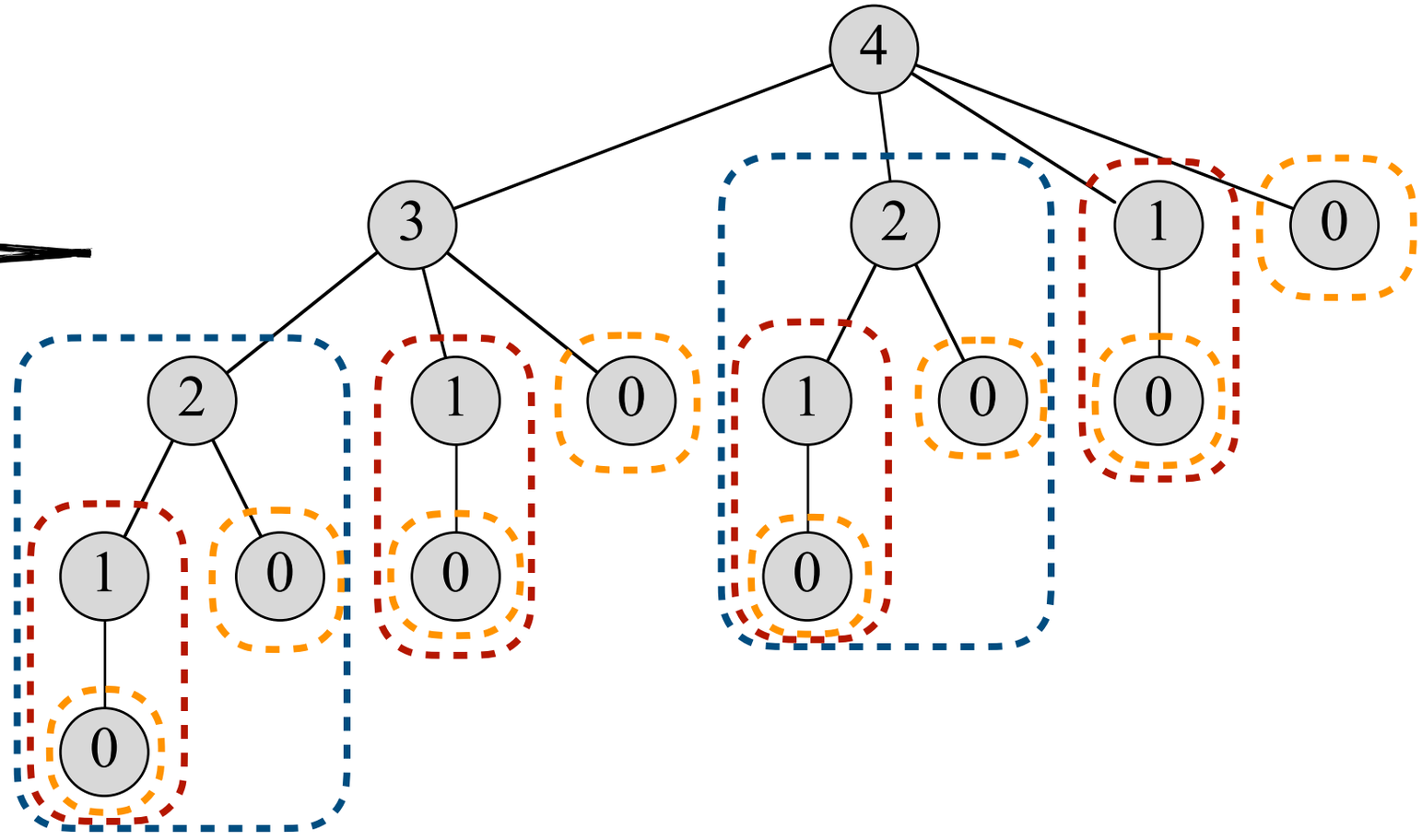
# of calls in the recursive calls

- One can prove (using substitution method) that $T(n) = 2^n$.
- Hence, also the running-time of CUT-ROD is exponential in n

Can we do better?

Cut-Rod calls itself recursively over and over again with the same parameter values, **solving the same subproblems repeatedly**.

There the number of recursive calls grows exponentially, but the number of subproblems is linear in n



Dynamic Programming

Basic Idea

- We arrange for each subproblem to be solved only once, storing its solution
- If we need to refer to this subproblem's solution again later, we can just look it up, rather than recompute it

- Dynamic programming uses **additional memory to save computation time**
- Two equivalent approaches to implement the idea:
  - Top-down with memoization
  - Bottom-up method

Dynamic Programming

Top-down with memoization

- Write the procedure recursively in a natural manner
- But modified to save the result of each subproblem (usually in an array or an hash table)
- The procedure now first checks to see whether it has previously solved the subproblem
  - If so it returns the saved value
  - Otherwise it computes the value in the usual manner
- We say that the recursive procedure has memoized (it remembers) what results it has compute previously

Memoized Rod-Cutting

```
MEMOIZED-CUT-ROD(p, n)
1  let r[0..n] be a new array
2  for i = 0 to n
3      r[i] = −∞
4  return MEMOIZED-CUT-ROD-AUX(p, n, r)
```

The array stores the solutions of the subproblems.
$-\infty$ stands for "unknown result"

Call auxiliary procedure

```
MEMOIZED-CUT-ROD-AUX(p, n, r)
1  if r[n] ≥ 0
2      return r[n]
3  if n == 0
4      q = 0
5  else q = −∞
6      for i = 1 to n
7          q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n − i, r))
8  r[n] = q
9  return q
```

Recover stored optimal value

Same as ROD-CUT

Memoized Cut-Rod: run-time

```
MEMOIZED-CUT-ROD(p, n)
1  let r[0..n] be a new array
2  for i = 0 to n
3      r[i] = -∞
4  return MEMOIZED-CUT-ROD-AUX(p, n, r)
```

$\Theta(n)$

$T(n)$

```
MEMOIZED-CUT-ROD-AUX(p, n, r)
1  if r[n] ≥ 0
2      return r[n]
3  if n == 0
4      q = 0
5  else q = -∞
6      for i = 1 to n
7          q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n - i, r))
8  r[n] = q
9  return q
```

Iterates n times

Each call is computed just once

Memoized Cut-Rod: run-time

- A recursive call to solve a previously solved subproblem returns immediately
- MEMOIZED-CUT-ROD solves each subproblem just once
- To solve a subproblem of size n the for-loop in lines 6-7 iterates n times

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

- total number of iterations of this for loop, over all recursive calls of MEMOIZED-CUT-ROD, forms an arithmetic series

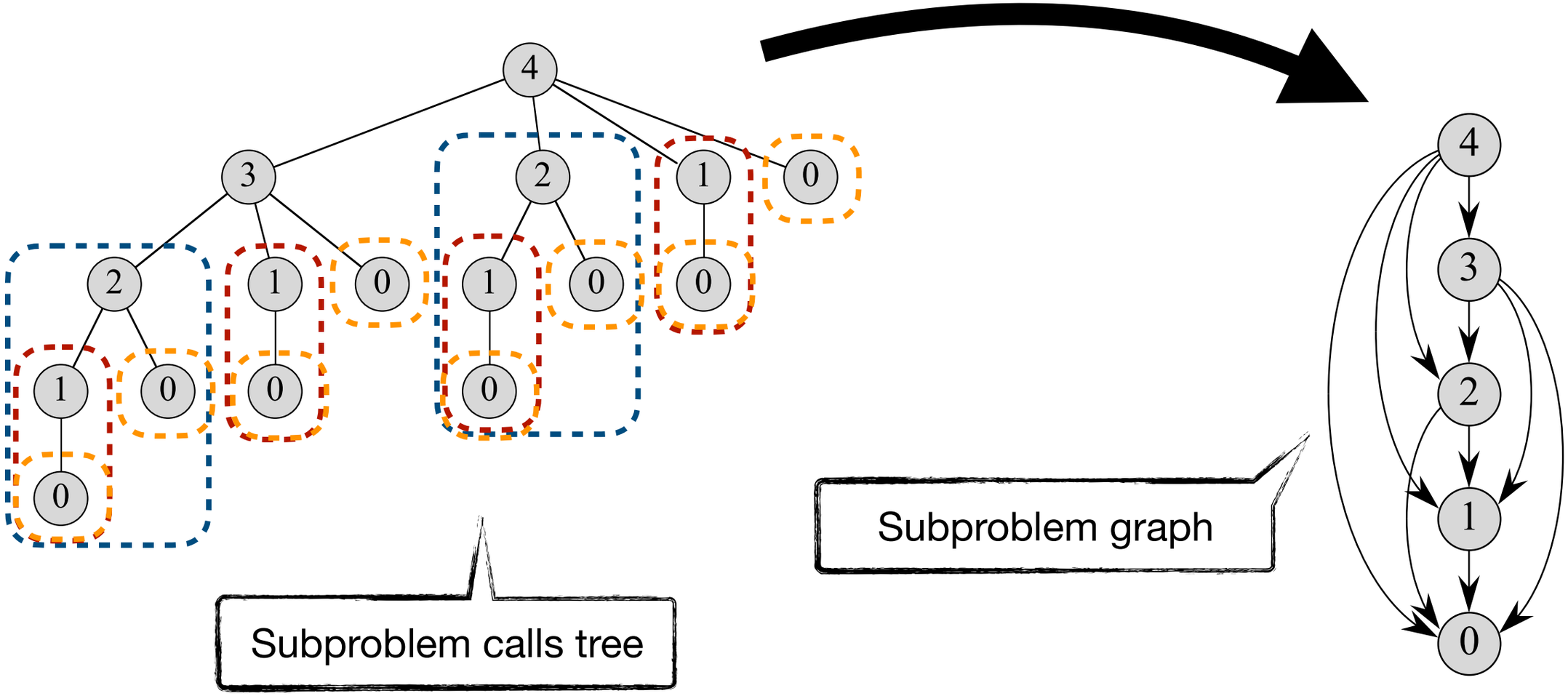
Dynamic Programming

Bottom-up method

- Depends on some natural notion of the “size” of the subproblem: solving any particular subproblem depends only on solving “smaller” subproblems
- We sort the subproblems by size and solve them in size order, smallest first.
- When solving a particular subproblem, we have already solved all of the smaller subproblems its solution depends on (and we have saved their solutions)

Subproblem Graph

- We need to understand the set of subproblems involved and how subproblems depend on one another
- The **subproblem graph** for the problem embodies exactly this information



Bottom-Up Cut-Rod

```
BOTTOM-UP-CUT-ROD(p, n)
1   let r[0..n] be a new array
2   r[0] = 0
3   for j = 1 to n
4       q = -∞
5       for i = 1 to j
6           q = max(q, p[i] + r[j - i])
7       r[j] = q
8   return r[n]
```

Base case

Solve all subproblems.
The order matters!

Retrieve the results from the array r

Store the new result in the array r

Return the solution of the biggest problem

Bottom-Up Cut-Rod: run-time

Takes time $\Theta(n^2)$ due to its doubly nested loop structure

```
BOTTOM-UP-CUT-ROD(p, n)
1  let r[0..n] be a new array
2  r[0] = 0
3  for j = 1 to n
4      q = -∞
5      for i = 1 to j
6          q = max(q, p[i] + r[j - i])
7      r[j] = q
8  return r[n]
```

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

Constructing the solution

- So far we focused on calculating the optimal revenue
- What about the optimal list of cuts?
- We use an array $s[0..n]$ to store the optimal choice
  - $r_j = \max_{i=1..j}(p_i + r_{j-i})$ — r_j is the revenue of a subproblem of size j
  - $s[j] = \underset{i=1..j}{argmax}(p_i + r_{j-i})$ — $s[j]$ is the length of the first cut in a subproblem of size j

```
PRINT-CUT-ROD-SOLUTION(p, n)
1  (r, s) = EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
2  while n > 0
3      print s[n]
4      n = n - s[n]
```

Size of the second part

List of cuts:
- Print the first cut
- Iteratively print the list of cuts for the second part

Constructing the solution

```
EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
 1  let r[0..n] and s[0..n] be new arrays
 2  r[0] = 0
 3  for j = 1 to n
 4      q = -∞
 5      for i = 1 to j
 6          if q < p[i] + r[j - i]
 7              q = p[i] + r[j - i]
 8              s[j] = i
 9      r[j] = q
10  return r and s
```

We use a new array to store in $s[j]$ the length of the first cut in a subproblem of size j

Update the revenue and the size of the first cut of the subproblem of size j

```
PRINT-CUT-ROD-SOLUTION(p, n)
1  (r, s) = EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
2  while n > 0
3      print s[n]
4      n = n - s[n]
```

Size of the second part

List of cuts:

- Print the first cut
- Iteratively print the list of cuts for the second part

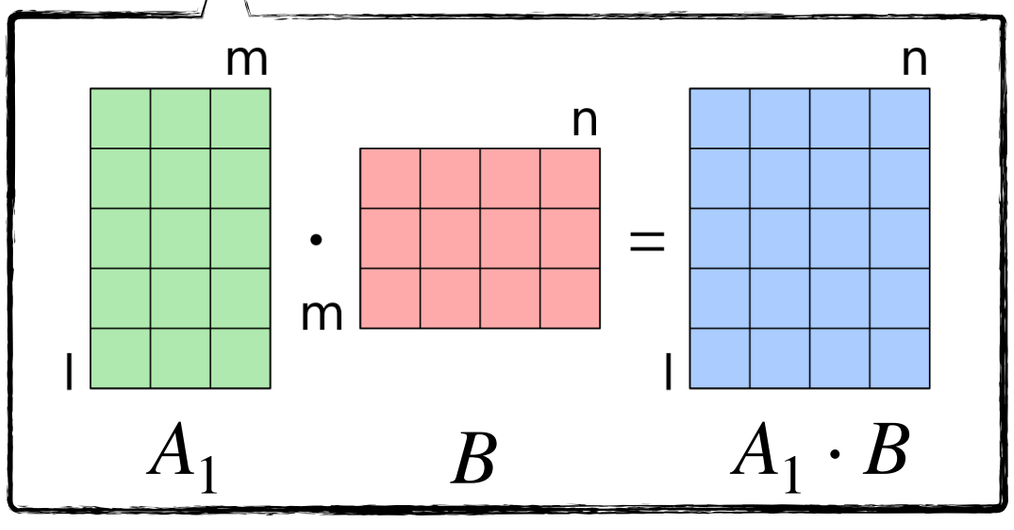
Matrix-chain multiplication

Problem description

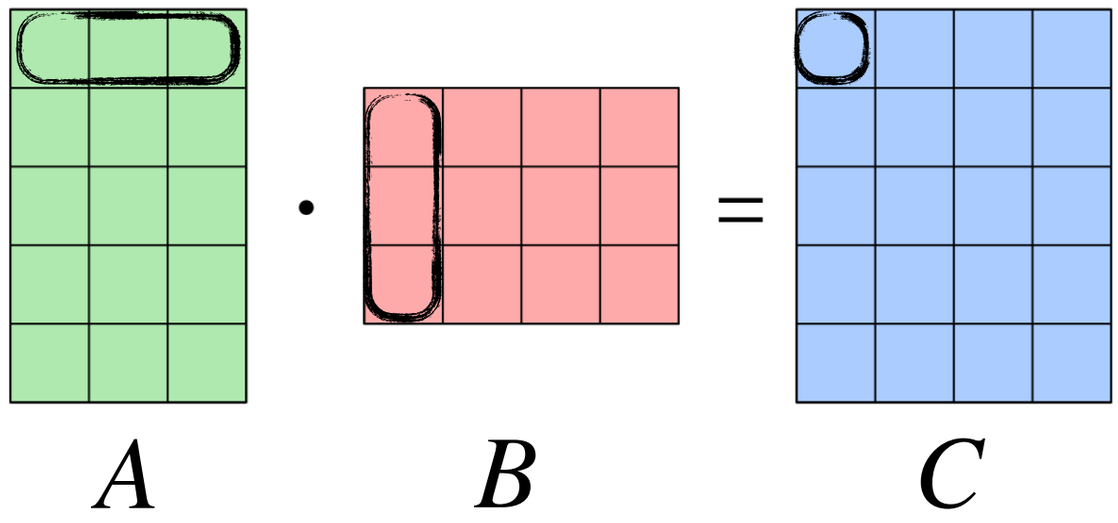
- **Input:** a sequence $A_1, A_2, \ldots, A_n$ of n matrices
- **Output:** compute the product $A_1 A_2 \cdots A_n$

Naive solution

- We can resolve the problem by recursively computing $B = A_2 \cdots A_n$ and then multiply $A_1 \cdot B$



Matrix Multiplication




```
MATRIX-MULTIPLY(A, B)
1  if A.columns ≠ B.rows
2       error "incompatible dimensions"
3  else let C be a new A.rows × B.columns matrix
4       for i = 1 to A.rows
5            for j = 1 to B.columns
6                 c_ij = 0
7                 for k = 1 to A.columns
8                      c_ij = c_ij + a_ik · b_kj
9       return C
```

- If A is a $p \times q$ matrix and B is a $q \times r$ matrix, then C will result in a $p \times r$ matrix
- The time to compute C is dominated by the number of scalar multiplications in line 8, which is pqr

Smart parenthesization

- Consider the problem of computing $A_1 A_2 A_3$ where
  - A_1 is a 10×100 matrix
  - A_2 is a 100×5 matrix
  - A_3 is a 5×50 matrix
- **Key observation:** matrix multiplication is associative

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

- Same result, but **computation time may differ a lot**

Smart parenthesization

- Consider the problem of computing $A_1A_2A_3$ where
  - A_1 is a 10×100 matrix
  - A_2 is a 100×5 matrix
  - A_3 is a 5×50 matrix

- **Key observation:** matrix multiplication is associative

$$A_1(A_2A_3) = (A_1A_2)A_3$$

- **Compute** A_2A_3
  - **takes** $100 \cdot 5 \cdot 50 = 25000$ **scalar multiplications**
  - **And** A_2A_3 **is a** 100×50 **matrix**
- **Multiply** A_1 **with** A_2A_3
  - **Takes** $10 \cdot 100 \cdot 50 = 50000$ **scalar multiplication**
- **In total:** $25000 + 50000 = 75000$ **scalar multiplications**

Smart parenthesization

- Consider the problem of computing $A_1A_2A_3$ where
  - A_1 is a 10×100 matrix
  - A_2 is a 100×5 matrix
  - A_3 is a 5×50 matrix

- **Key observation:** matrix multiplication is associative

$$A_1(A_2A_3) = (A_1A_2)A_3$$

- **Compute A_1A_2**
  - **takes $10 \cdot 100 \cdot 5 = 5000$ scalar multiplications**
  - **And A_1A_2 is a 10×5 matrix**
- **Multiply A_1A_2 with A_3**
  - **Takes $10 \cdot 5 \cdot 50 = 2500$ scalar multiplications**
- **In total: $5000 + 2500 = 7500$ scalar multiplications**

10x faster!!

Find optimal parenthesization

The matrix-chain multiplication problem: given a chain $\langle A_1, A_2, \ldots, A_n \rangle$ of n matrices, where for $i = 1 \ldots n$, matrix A_i has dimension $p_{i-1} \times p_i$, fully parenthesize the product $A_1 A_2 \cdots A_n$ in a way that minimises the number of scalar multiplications.

- Note that this problem does not ask to actually multiply the matrices.

- Our goal is only to **determine an order for multiplying matrices that has the lowest cost**

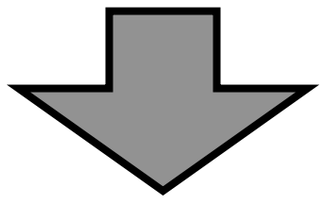
- Typically, the time invested in determining this optimal order is worth the time saved later on when performing the multiplication

Brute-force search

- **Find all possible parenthesizations**
- Chose the one with minimal cost

- How many parenthesizations do we need to check?

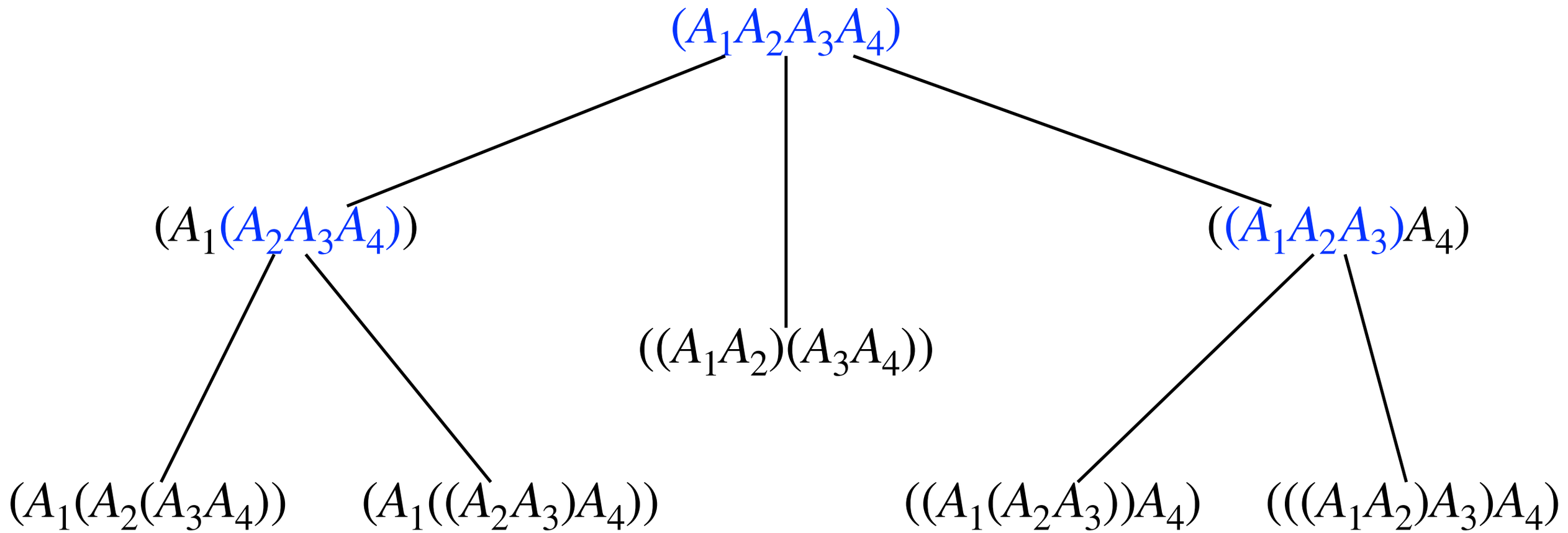
$$\langle A_1, A_2, A_3, A_4 \rangle$$



$(A_1(A_2(A_3A_4)))$ $((A_1A_2)(A_3A_4))$ $(((A_1A_2)A_3)A_4)$

$(A_1((A_2A_3)A_4))$ $((A_1(A_2A_3))A_4)$

Counting the number of parenthesizations



Counting the number of parenthesizations

Denote the number of alternative parethesizations of a sequence of n matrices by $P(n)$

$$P(n) = \begin{cases} 1 & \text{if } n = 1, \\ \sum_{k=1}^{n-1} P(k)\,P(n-k) & \text{if } n \geq 2. \end{cases}$$

If there is only 1 matrix, then there is only one way to parenthesize

For each possible split $k = 1 \ldots n-1$ of the sequence, we pair all possible parethesizations of the two sub-sequences

Counting the number of parenthesizations

Denote the number of alternative parethesizations of a sequence of n matrices by $P(n)$

$\Omega(2^n)$

see CLRS Exercise 15.2-3

$$P(n) = \begin{cases} 1 & \text{if } n = 1 , \\ \sum_{k=1}^{n-1} P(k) P(n-k) & \text{if } n \geq 2 . \end{cases}$$

The brute-force approach is not a practical choice

Applying dynamic programming

- **Step 1:** Characterise the structure of an optimal solution
- **Step 2:** Recursively define the value of an optimal solution
- **Step 3:** Compute the value of an optimal solution
- **Step 4:** Construct an optimal solution from computed information

Step 1

The structure of an optimal solution

- For $i \leq j$, consider the optimal parethesization for the sub-chain $A_i A_{i+1} \cdots A_j$
- If $i = j$, then the there is only one way to solve the problem
- If $i < j$, to parenthesize $A_i A_{i+1} \cdots A_j$ we must split the product between A_k and A_{k+1} for some $i \leq k < j$
  - Suppose that the choice of k is optimal. Then the optimal parethesization for $A_i A_{i+1} \cdots A_j$ **must be obtained from optimal parethesizations for** $A_i \cdots A_k$ **and** $A_{k+1} \cdots A_j$**.**

Towards a contradiction, assume that for $A_i \cdots A_k$ or $A_{k+1} \cdots A_j$ we did not use the optimal parethesization. Then, we could obtain a better parethesization for $A_i A_{i+1} \cdots A_j$, contradicting the initial hypothesis.

Step 2: a recursive solution

- For $i \leq j$, let $m[i, j]$ the min number of scalar multiplications needed to compute $A_i \cdots A_j$.
- Recall that for $i = 1 \ldots n$, matrix A_i has dimension $p_{i-1} \times p_i$.
- Then, $m[i, j]$ can be recursively defined as

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

Range over all possible splits of $A_i \ldots A_j$

Optimal value for the sub-problems $A_i \ldots A_k$ and $A_{k+1} \ldots A_j$

Cost for multiplying $A_i \cdots A_k$ and $A_{k+1} \cdots A_j$

- $A_i \cdots A_k$ is a $p_{i-1} \times p_k$ matrix
- $A_{k+1} \cdots A_j$ is a $p_k \times p_j$ matrix

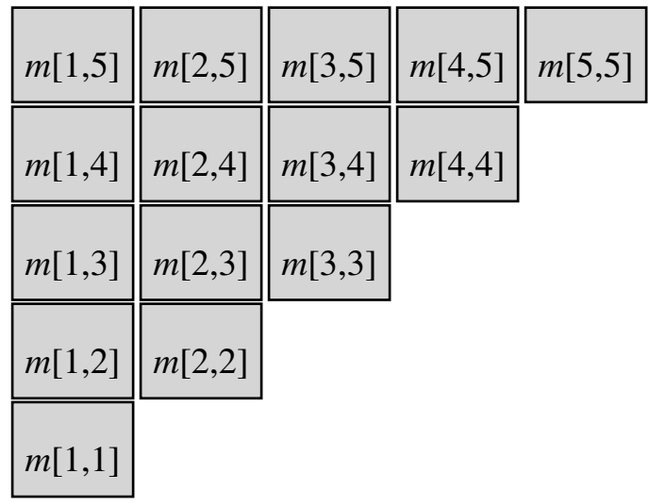
Step 3: computing optimal cost

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

- As for the rod-cut problem, implementing this recursion as it is would take exponential time
- Again, observe that we have relatively few distinct sub-problems: one for each choice of i and j such that $1 \le i \le j \le n$, which is $(n^2 + n)/2 = \Theta(n^2)$

Example

Graphical representation of the sub-problems arising from $m[1,5]$



Step 3: computing optimal cost

- We compute the optimal cost in a bottom-up fashion
- Recall that the matrix A_i has dimensions $p_{i-1} \times p_i$, for $i = 1,2,\ldots,n$.
- Assume the input to be the sequence $\langle p_0, p_1, \ldots, p_n \rangle$ where p.length = $n + 1$

MATRIX-CHAIN-ORDER(p)

```
 1  n = p.length − 1
 2  let m[1..n, 1..n] and s[1..n − 1, 2..n] be new tables
 3  for i = 1 to n
 4      m[i, i] = 0
 5  for l = 2 to n                    // l is the chain length
 6      for i = 1 to n − l + 1
 7          j = i + l − 1
 8          m[i, j] = ∞
 9          for k = i to j − 1
10              q = m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
11              if q < m[i, j]
12                  m[i, j] = q
13                  s[i, j] = k
14  return m and s
```

Step 3: computing optimal cost

- We compute the optimal cost in a bottom-up fashion
- Recall that the matrix A_i has dimensions $p_{i-1} \times p_i$, for $i = 1,2,\ldots,n$.
- Assume the input to be the sequence $\langle p_0, p_1, \ldots, p_n \rangle$ where p.length = $n + 1$

$m[1..n,1..n]$ will store the values $m[i,j]$

```
MATRIX-CHAIN-ORDER(p)
 1  n = p.length − 1
 2  let m[1..n, 1..n] and s[1..n − 1, 2..n] be new tables
 3  for i = 1 to n
 4      m[i, i] = 0
 5  for l = 2 to n                    // l is the chain l
 6      for i = 1 to n − l + 1
 7          j = i + l − 1
 8          m[i, j] = ∞
 9          for k = i to j − 1
10              q = m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
11              if q < m[i, j]
12                  m[i, j] = q
13                  s[i, j] = k
14  return m and s
```

$s[1..n-1,2..n]$ records which index k achieved the optimal value for $m[i,j]$

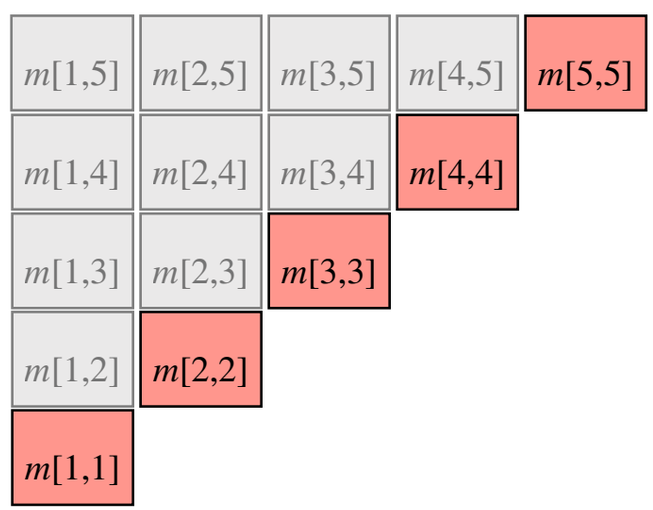
Step 3: computing optimal cost

- We compute the optimal cost in a bottom-up fashion
- Recall that the matrix A_i has dimensions $p_{i-1} \times p_i$, for $i = 1,2,\ldots,n$.
- Assume the input to be the sequence $\langle p_0, p_1, \ldots, p_n \rangle$ where p.length = $n + 1$

MATRIX-CHAIN-ORDER(p)

```
1   n = p.length − 1
2   let m[1..n, 1..n] and s[1..n − 1, 2..n] be new tables
3   for i = 1 to n
4       m[i, i] = 0
5   for l = 2 to n                    // l is the chain length
6       for i = 1 to n − l + 1
7           j = i + l − 1
8           m[i, j] = ∞
9           for k = i to j − 1
10              q = m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
11              if q < m[i, j]
12                  m[i, j] = q
13                  s[i, j] = k
14  return m and s
```

Compute the base cases of the recursion $m[i, i] = 0$



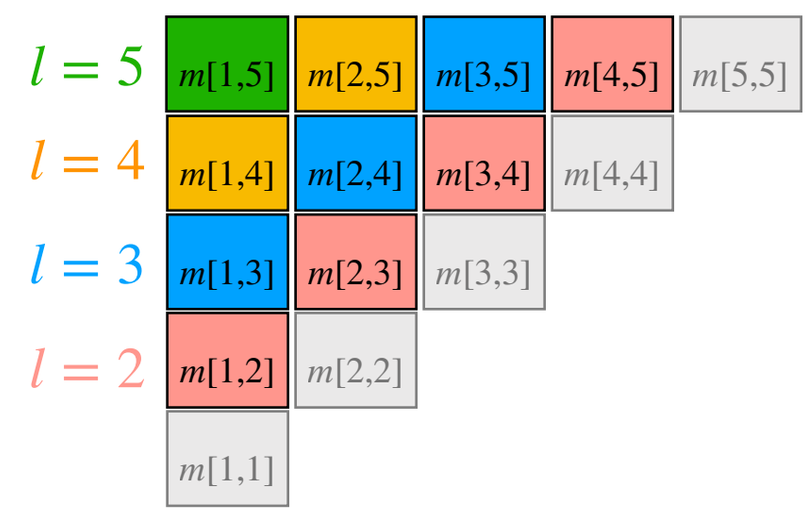
Step 3: computing optimal cost

- We compute the optimal cost in a bottom-up fashion
- Recall that the matrix A_i has dimensions $p_{i-1} \times p_i$, for $i = 1,2,\ldots,n$.
- Assume the input to be the sequence $\langle p_0, p_1, \ldots, p_n \rangle$ where p.length = $n + 1$

MATRIX-CHAIN-ORDER(p)

```
1   n = p.length − 1
2   let m[1..n, 1..n] and s[1..n − 1, 2..n] be new tables
3   for i = 1 to n
4       m[i, i] = 0
5   for l = 2 to n                    // l is the chain length
6       for i = 1 to n − l + 1
7           j = i + l − 1
8           m[i, j] = ∞
9           for k = i to j − 1
10              q = m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
11              if q < m[i, j]
12                  m[i, j] = q
13                  s[i, j] = k
14  return m and s
```

Compute sub-problems gathering them by length



Step 3: computing optimal cost

- We compute the optimal cost in a bottom-up fashion
- Recall that the matrix A_i has dimensions $p_{i-1} \times p_i$, for $i = 1,2,\ldots,n$.
- Assume the input to be the sequence $\langle p_0, p_1, \ldots, p_n \rangle$ where p.length = $n + 1$

```
MATRIX-CHAIN-ORDER(p)
 1  n = p.length − 1
 2  let m[1..n, 1..n] and s[1..n − 1, 2..n] be new tables
 3  for i = 1 to n
 4      m[i, i] = 0
 5  for l = 2 to n                  // l is the chain length
 6      for i = 1 to n − l + 1
 7          j = i + l − 1
 8          m[i, j] = ∞
 9          for k = i to j − 1
10              q = m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
11              if q < m[i, j]
12                  m[i, j] = q
13                  s[i, j] = k
14  return m and s
```

Compute $m[i, j]$ as

$$\min_{i \le k < j} \{m[i,k] + m[k+1, j] + p_{i-1} p_k p_j\}$$

…storing in $s[i, j]$ the optimal choice for k

Runtime Analysis

We use n —length of the matrix-chain— as the input size of the problem

```
MATRIX-CHAIN-ORDER(p)
 1  n = p.length − 1                                          Θ(1)
 2  let m[1..n, 1..n] and s[1..n − 1, 2..n] be new tables
 3  for i = 1 to n                                            Θ(n)
 4      m[i, i] = 0
 5  for l = 2 to n                  // l is the chain length
 6      for i = 1 to n − l + 1
 7          j = i + l − 1
 8          m[i, j] = ∞
 9          for k = i to j − 1
10              q = m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
11              if q < m[i, j]
12                  m[i, j] = q
13                  s[i, j] = k
14  return m and s                                            Θ(1)
```

The loops are nested three deep, and each loop index takes $\Theta(n)$ values. Thus, it takes overall $\Theta(n^3)$ time

Observe that the algorithm requires also $\Theta(n^2)$ space for the tables m and s.

Step 4: Construct the optimal solution

- We can now use the table $s[1..n-1, 2..n]$ to reconstruct the optimal full parenthetization
- The algorithm below takes as input the table s and two integers i and j such that $1 \le i \le j \le n$ and prints out the optimal parethetization for the sub-chain $\langle A_i, \ldots, A_j \rangle$

```
PRINT-OPTIMAL-PARENS(s, i, j)
1  if i == j
2       print "A"_i
3  else print "("
4       PRINT-OPTIMAL-PARENS(s, i, s[i, j])
5       PRINT-OPTIMAL-PARENS(s, s[i, j] + 1, j)
6       print ")"
```

$k = s[i, j]$ is the optimal split index for $A_1 \ldots A_j$

Optimal solution for $A_i \ldots A_k$

Optimal solution for $A_{k+1} \ldots A_j$

Quiz

- Give a recursive algorithm MATRIX-CHAIN-MULTIPLY(A, s, i, j) that performs the optimal matrix-chain multiplication, given the sequence of matrices $A = \langle A_1, \ldots, A_n \rangle$, the table s computed by MATRIX-CHAIN-ORDER, and the indices i and j.
- The initial call would be MATRIX-CHAIN-MULTIPLY$(A, s, 1, n)$

- Assume that all matrices in the chain have all the same dimension $p \times p$. Can you do better?

Elements of Dynamic Programming

- **Optimal Substructure**: the optimal value can be obtained from optimal values of the subproblems that can be computed independently from the main problem
- **Overlapping subproblems**: dynamic programming pays off when the subproblems graph is much smaller than the tree of recursive calls
- **Memoization**: store the optimal value of each subproblem and reuse it when needed again (useful for top-down recursive implementations)
- **Reconstructing an optimal solution** by storing which choice we made in each subproblem

Discover optimal substructure

Common pattern in discovering optimal substructure:

- Show that a solution to the problem consists in making a choice. Making this choice leaves one or more subproblems to be solved
- You can obtain the optimal solution of the problem from the optimal solutions of the subproblems

- Try to keep the space of subproblems as simple as possible
- Inspect the tree of subproblems looking for some symmetry and check if you can prune the tree