CSE 100: Algorithm Design and Analysis
Chapter 15: Dynamic Programming

Sungjin Im

University of California, Merced

Last Update: 03-10-2023

Outline

- What is DP?
- How does it work?
- How do we analyze the running time?

Examples: Fibonacci numbers, Rod cutting, Matrix chain multiplication, Longest common subsequence.

Sample Questions

- Derive a recursion for a given problem.
- Does the naive implementation of the recursion have an exponential running time?
- Solve the following problem using DP. Bottom-up vs. Top-down.
- Translate a given recursion into a DP.
- Analyze the running time of a given DP.
- Find an optimal solution by a traceback method.

Examples: Fibonacci numbers, Rod cutting, Matrix chain multiplication, Longest common subsequence.

Dynamic Programming

- Not a specific algorithm, but a technique.
- Not actual computer programming
- Used for an optimization (minimization or maximization) problem whose recursion makes lots of overlapping recursive calls.
  - Optimization problem: Find *a* solution with *the* optimal value.

Warm-up: Fibonacci number

The Fibonacci Sequence is 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, $\cdots$. Formally, $f(0) = 0, f(1) = 1$ and $f(i) = f(i-1) + f(i-2)$ for all $i \geq 2$, where $f(i)$ denotes the ith Fibonacci number. Would like to compute nth Fib number.

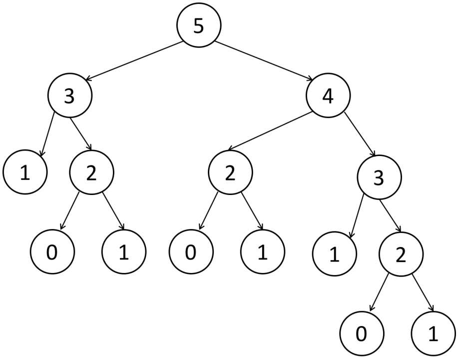
A naive implementation:

```
int F(n)
  if n == 0 return 0
  if n == 1 return 1
  return F(n-2) + F(n-1)
```

Warm-up: Fibonacci number

The Fibonacci Sequence is 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, $\cdots$.
Formally, $f(0) = 0, f(1) = 1$ and $f(i) = f(i-1) + f(i-2)$ for all $i \geq 2$, where $f(i)$ denotes the ith Fibonacci number. Would like to compute nth Fib number.

Recursion tree of the naive implementation:



Warm-up: Fibonacci number

DP via bottom-up: Pseudocode

The Fibonacci Sequence is 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, $\cdots$. Formally, $f(0) = 0, f(1) = 1$ and $f(i) = f(i-1) + f(i-2)$ for all $i \geq 2$, where $f(i)$ denotes the ith Fibonacci number. Would like to compute nth Fib number.

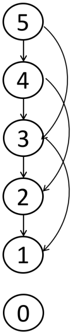
```
bottom up:
int F(n)
  Array A[0 ... n]
  A[0] = 0, A[1] = 1
  for i = 2; i <= n ; i++
     A[i] = A[i-1] + A[i-2]
  return A[i]
```

Warm-up: Fibonacci number

DP via bottom-up

The Fibonacci Sequence is 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, $\cdots$. Formally, $f(0) = 0, f(1) = 1$ and $f(i) = f(i-1) + f(i-2)$ for all $i \geq 2$, where $f(i)$ denotes the ith Fibonacci number. Would like to compute nth Fib number.

Dependency graph:



Warm-up: Fibonacci number

DP via top-down with memoization

The Fibonacci Sequence is 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, $\cdots$. Formally, $f(0) = 0, f(1) = 1$ and $f(i) = f(i-1) + f(i-2)$ for all $i \geq 2$, where $f(i)$ denotes the ith Fibonacci number. Would like to compute nth Fib number.

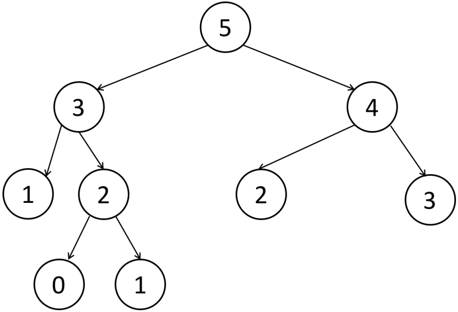
```
top down:
int F(n)
  Array A[0 ... n]
  A[0] = 0, A[1] = 1
  A[2] = A[3]  = ... = A[n] = - infinity
  return Aux-F(A, n)

int Aux-F(n)
  if A[n] >=  0 return A[n]
  A[n] = Aux-F(A, n-1) + Aux-F(A, n-2)
  return A[n]
```

Warm-up: Fibonacci number

DP via top-down with memoization

The Fibonacci Sequence is 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, $\cdots$. Formally, $f(0) = 0, f(1) = 1$ and $f(i) = f(i-1) + f(i-2)$ for all $i \geq 2$, where $f(i)$ denotes the ith Fibonacci number. Would like to compute nth Fib number.



Rod Cutting

Cut a given steel rod into pieces of integer lengths in order to maximize the revenue. Cutting is free.
Input:

- n: the length of the given rod
- p_i: the price of a rod/piece of length i, $1 \leq p_i \leq n$. ($p_0 = 0$).

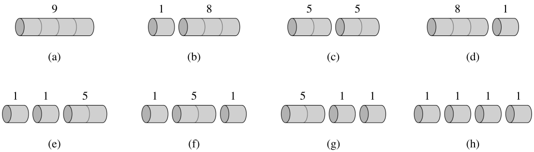
length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

Rod Cutting

Cut a given steel rod into pieces of integer lengths in order to maximize the revenue. Cutting is free.
Input:

- n: the length of the given rod
- p_i: the price of a rod/piece of length i, $1 \leq p_i \leq n$. ($p_0 = 0$).

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

1. Interpretation as an Optimization Problem

- This is an optimization problem.
- A feasible solution is a cut of the given rod into pieces of integer lengths.
- Each solution yields a certain profit.
- The *objective* is the profit, which is to be *maximized*.
- In general, a solution maximizing (or minimizing) the objective is called an *optimum solution*. The maximum (or minimum) objective is called the *optimum*. There can be multiple optimum solutions, but the optimum is unique.

Let r_n denote the optimum for rod of length n, i.e., the max profit we can get out of a rod of length n.

Let r_n denote the optimum for rod of length n, i.e. the max profit we can get out of a rod of length n.

$$
\begin{aligned}
r_1 &= 1 && \text{from solution } 1 = 1 \quad \text{(no cuts)} , \\
r_2 &= 5 && \text{from solution } 2 = 2 \quad \text{(no cuts)} , \\
r_3 &= 8 && \text{from solution } 3 = 3 \quad \text{(no cuts)} , \\
r_4 &= 10 && \text{from solution } 4 = 2 + 2 , \\
r_5 &= 13 && \text{from solution } 5 = 2 + 3 , \\
r_6 &= 17 && \text{from solution } 6 = 6 \quad \text{(no cuts)} , \\
r_7 &= 18 && \text{from solution } 7 = 1 + 6 \ \text{ or } \ 7 = 2 + 2 + 3 , \\
r_8 &= 22 && \text{from solution } 8 = 2 + 6 , \\
r_9 &= 25 && \text{from solution } 9 = 3 + 6 , \\
r_{10} &= 30 && \text{from solution } 10 = 10 \quad \text{(no cuts)} .
\end{aligned}
$$

Rod Cutting

2. Finding a Recursion to Compute the Optimum

Let r_n denote the optimum for rod of length n, i.e. the max profit we can get out of a rod of length n.
Let's first try to find a recursion to compute r_n. Then, we will try to find a solution that achieves r_n.

$$r_j = \max_{1 \le i \le j} (p_i + r_{j-i}) \text{ if } j \ge 1$$

$$r_0 = 0$$

Interpretation: Here, i is the length of the first piece.

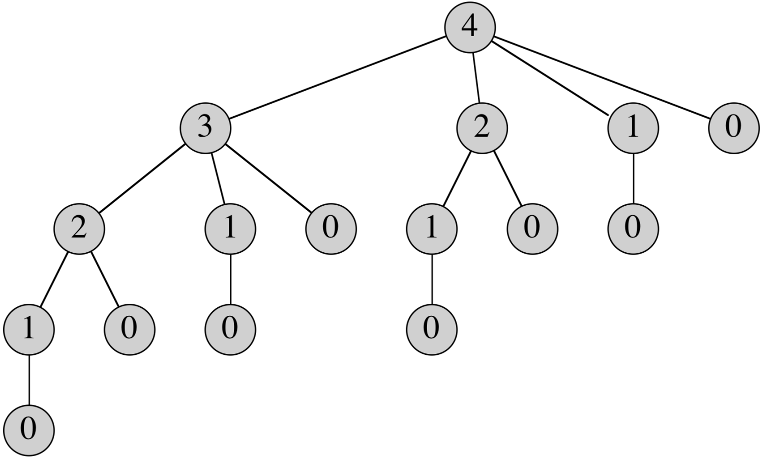
Rod Cutting

2. Finding a Recursion to Compute the Optimum: Pseudocode

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

Rod Cutting

3. Identifying Redundancies in the Recursion Tree and Deciding What to Store



Rod Cutting

4. USE DP for Speed-up: Bottom-up DP description in words

(Let $p[i] = p_i$ denote the price of a rod of length i)
Set up a DP table with entries $r[i]$ where $0 \le i \le n$
$r[0] = 0$
Compute $r[1], r[2], ..., r[n]$ in this order using the recursion,
$r[j] = \max_{1 \le i \le j}(p[i] + r[j-i])$
Return $r[n]$ as the optimum

length i	0	1	2	3	4	5	6
p_i	0	1	5	8	9	10	17
r_i							

Rod Cutting

4. USE DP for Speed-up: Bottom-up DP pseudocode

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

Rod Cutting

5. RT Analysis

(Let $p[i] = p_i$ denote the price of a rod of length i)
Set up a DP table with entries $r[i]$ where $0 \leq i \leq n$
$r[0] = 0$
Compute $r[1], r[2], ..., r[n]$ in this order using the recursion,
$r[j] = \max_{1 \leq i \leq j}(p[i] + r[j-i])$
Return $r[n]$ as the optimum.

Number of entires in the DP table to fill out:

Rod Cutting

5. RT Analysis

(Let $p[i] = p_i$ denote the price of a rod of length i)
Set up a DP table with entries $r[i]$ where $0 \leq i \leq n$
$r[0] = 0$
Compute $r[1], r[2], ..., r[n]$ in this order using the recursion,
$r[j] = \max_{1 \leq i \leq j}(p[i] + r[j-i])$
Return $r[n]$ as the optimum.

Number of entires in the DP table to fill out: n.
Time needed to fill out each DP table entry:

Rod Cutting

5. RT Analysis

(Let $p[i] = p_i$ denote the price of a rod of length i)
Set up a DP table with entries $r[i]$ where $0 \leq i \leq n$
$r[0] = 0$
Compute $r[1], r[2], ..., r[n]$ in this order using the recursion,
$r[j] = \max_{1 \leq i \leq j}(p[i] + r[j-i])$
Return $r[n]$ as the optimum.

Number of entires in the DP table to fill out: n.
Time needed to fill out each DP table entry: $O(n)$.

Rod Cutting

5. RT Analysis

(Let $p[i] = p_i$ denote the price of a rod of length i)
Set up a DP table with entries $r[i]$ where $0 \leq i \leq n$
$r[0] = 0$
Compute $r[1], r[2], ..., r[n]$ in this order using the recursion,
$r[j] = \max_{1 \leq i \leq j}(p[i] + r[j - i])$
Return $r[n]$ as the optimum.

Number of entires in the DP table to fill out: n.
Time needed to fill out each DP table entry: $O(n)$.
Thus, we can compute the optimum, r_n in $n * O(n) = O(n^2)$ time.

Key idea: To store the length of the first piece in an optimum solution/cut.

$$r_j = \max_{1 \le i \le j}(p_i + r_{j-i}) \text{ if } j \ge 1$$

$$r_0 = 0$$

Let s_j be i such that $p_i + r_{j-i} = r_j$. In other words, s_j is the length of the first piece in a cut giving profit r_j.

Rod Cutting

6. Finding an Optimum Solution

Key idea: To store the length of the first piece in an optimum solution/cut.

```
EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
let r[0..n] and s[0..n] be new arrays
r[0] = 0
for j = 1 to n
    q = -∞
    for i = 1 to j
        if q < p[i] + r[j - i]
            q = p[i] + r[j - i]
            s[j] = i
    r[j] = q
return r and s
```

Rod Cutting

6. Finding an Optimum Solution

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

Rod Cutting

Another DP solution: Top-down with memoization

Solve recursively, but store each result in a table.
To find the solution to a subproblem, first look in the table.
If the answer is there, use it.
Otherwise, compute the solution to the subproblem and store it in the table for future use.

Rod Cutting

Another DP solution: Top-down with memoization

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

```
MEMOIZED-CUT-ROD-AUX(p, n, r)
if r[n] ≥ 0
    return r[n]
if n == 0
    q = 0
else q = -∞
    for i = 1 to n
        q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n - i, r))
r[n] = q
return q
```

Matrix chain multiplication

Input: a sequence of n matrices, $A_1, A_2, ..., A_n$ where A_i is $p_{i-1} \times p_i$.
Output: fully parenthesize the product $A_1 A_2 \cdots A_n$ such that the number of multiplications is minimized.

Definition: A product of matrices is fully parenthesized if it is a single matrix or the product of two fully parenthesized matrix produces that is surrounded by parentheses.

Matrix chain multiplication

Input: a sequence of n matrices, $A_1, A_2, ..., A_n$ where A_i is $p_{i-1} \times p_i$.
Output: fully parenthesize the product $A_1 A_2 \cdots A_n$ such that the number of multiplications is minimized.

Definition: A product of matrices is fully parenthesized if it is a single matrix or the product of two fully parenthesized matrix produces that is surrounded by parentheses.

Questions: Is the following produce fully parenthesized?

- A: Yes
- AB: No
- (AB): Yes
- $(AB)C$: No

Matrix multiplication recap

- To be able to multiply two matrices, A and B, it must be the case that $\#$ of A's columns $= \#$ of B's rows.
- If A is $p \times q$ and B is $q \times r$, we assume that AB requires exactly pqr multiplications (using the standard matrix multiplication method).
- AB is $p \times r$.
- Matrix multiplication is associative, i.e. $(AB)C = A(BC)$.
- 1 to 1 mapping between full parenthesization and tree representation.

Matrix chain multiplication

Example

Input: a sequence of n matrices, $A_1, A_2, ..., A_n$ where A_i is $p_{i-1} \times p_i$.
Output: fully parenthesize the product $A_1 A_2 \cdots A_n$ such that the number of multiplications is minimized.

Example: $A_1(10), A_2(100 \times 5), A_3(5 \times 50)$.
Two possible ways of full parenthesizations:

- $(A_1 A_2) A_3$: $10 \cdot 100 \cdot 5 + 10 \cdot 5 \cdot 50 = 5000 + 2500 = 7500$.
- $A_1(A_2 A_3)$:
  $100 \cdot 5 \cdot 50 + 10 \cdot 100 \cdot 50 = 25000 + 50000 = 75000$.

Matrix chain multiplication

Side note: Catalan number

The number of full parenthesizations for the produce of n matrices is $P(n)$, where
$P(n) = \sum_{k=1}^{n-1} P(k)P(n-k)$ when $n \geq 2$ and $P(1) = 1$.

$P(n)$ has a special name, nth Catalan number, which appears in various forms in many problems. $P(n) = \Omega(2^n)$.

As usual, we will first try to compute the optimum, i.e. the minimum number of multiplications needed to compute the chain of products. So, the objective is the number of multiplications performed, which is to be minimized.

Then, we will find a full parenthesization achieving the optimum.

Matrix chain multiplication

2. Finding a Recursion to Compute the Optimum

Let's consider the tree representation, which is more intuitive.

1. The highest level split induces two subproblems.
2. Optimal solution comes from optimal solutions to the subproblems.
3. (We will need search for the best split).

Matrix chain multiplication

2. Finding a Recursion to Compute the Optimum

Let's consider the tree representation, which is more intuitive.

1. The highest level split induces two subproblems.
2. Optimal solution comes from optimal solutions to the subproblems.
3. (We will need search for the best split).

A natural subproblem is on $A_{i\dots j} := A_i A_{i+1} \cdots A_j$.

Matrix chain multiplication

2. Finding a Recursion to Compute the Optimum

Let's consider the tree representation, which is more intuitive.

1. The highest level split induces two subproblems.
2. Optimal solution comes from optimal solutions to the subproblems.
3. (We will need search for the best split).

A natural subproblem is on $A_{i\ldots j} := A_i A_{i+1} \cdots A_j$.
So, define $m[i, j]$ as the min number of multiplications needed to compute $A_{i\ldots j}$. Our goal is now to compute $m[1, n]$ fast.

Matrix chain multiplication

2. Finding a Recursion to Compute the Optimum

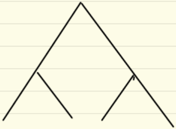
$$m[i,j] = \begin{cases} \min_{i \leq k \leq j-1} m[i,k] + m[k+1,j] + p_{i-1}p_k p_j & \text{if } i < j \\ 0 & \text{if } i = j \end{cases}$$

A_1 A_2 A_3 A_4

2 3 2 3 2

$(A_1 \mid (A_2 \ (A_3 \ A_4)))$

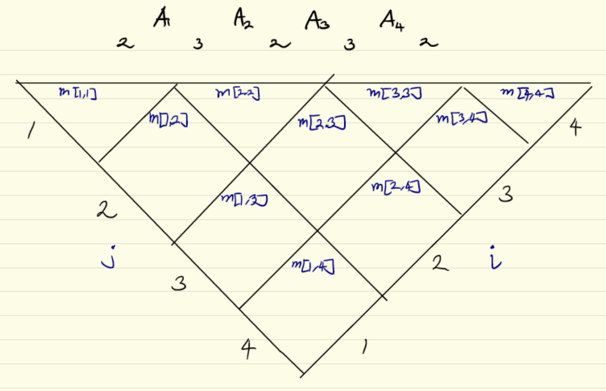
$(A_1 \mid ((A_2 \ A_3) \ A_4))$



$((A_1 \ A_2) \mid (A_3 \ A_4))$

$(((A_1 \ (A_2 \ A_3)) \mid A_4)$

$(((A_1 \ A_2) \ A_3) \mid A_4)$



A1
A2
A3
A4
2
3
2
3
2
m[1,1]
m[2,2]
m[3,3]
m[4,4]
m[1,2]
m[2,3]
m[3,4]
m[1,3]
m[2,4]
m[1,4]
1
2
3
4
j
4
3
2
1
i

Matrix chain multiplication

3. Identifying Redundancies in the Recursion Tree and Deciding What to Store

This is left as an exercise.

Matrix chain multiplication

4. USE DP for Speed-up: Bottom-up DP description in words; and 5. RT Analysis

Set up a DP table with entries $m[i, j]$ where $1 \le i \le j \le n$.
Compute $m[i, j]$ using the recursion in the following order:
$m[1, 1], m[2, 2], ..., m[n, n]$.
$m[1, 2], m[2, 3], ..., m[n-1, n]$.
$m[1, 3], m[2, 4], ..., m[n-2, n]$.
...
$m[1, n]$.
Return $m[1, n]$ as the optimum.

Matrix chain multiplication

4. USE DP for Speed-up: Bottom-up DP description in words; and 5. RT Analysis

Set up a DP table with entries $m[i,j]$ where $1 \leq i \leq j \leq n$.
Compute $m[i,j]$ using the recursion in the following order:
$m[1,1], m[2,2], ..., m[n,n]$.
$m[1,2], m[2,3], ..., m[n-1,n]$.
$m[1,3], m[2,4], ..., m[n-2,n]$.
...
$m[1,n]$.
Return $m[1,n]$ as the optimum.

Running time:

Matrix chain multiplication

4. USE DP for Speed-up: Bottom-up DP description in words; and 5. RT Analysis

Set up a DP table with entries $m[i, j]$ where $1 \leq i \leq j \leq n$.
Compute $m[i, j]$ using the recursion in the following order:
$m[1, 1], m[2, 2], ..., m[n, n]$.
$m[1, 2], m[2, 3], ..., m[n-1, n]$.
$m[1, 3], m[2, 4], ..., m[n-2, n]$.
...
$m[1, n]$.
Return $m[1, n]$ as the optimum.

Running time:

1. # of DP entries:

Matrix chain multiplication

4. USE DP for Speed-up: Bottom-up DP description in words; and 5. RT Analysis

Set up a DP table with entries $m[i,j]$ where $1 \le i \le j \le n$.
Compute $m[i,j]$ using the recursion in the following order:
$m[1,1], m[2,2], ..., m[n,n]$.
$m[1,2], m[2,3], ..., m[n-1,n]$.
$m[1,3], m[2,4], ..., m[n-2,n]$.
...
$m[1,n]$.
Return $m[1,n]$ as the optimum.

Running time:

1. # of DP entries: $O(n^2)$

Matrix chain multiplication

4. USE DP for Speed-up: Bottom-up DP description in words; and 5. RT Analysis

Set up a DP table with entries $m[i,j]$ where $1 \leq i \leq j \leq n$.
Compute $m[i,j]$ using the recursion in the following order:
$m[1,1], m[2,2], ..., m[n,n]$.
$m[1,2], m[2,3], ..., m[n-1,n]$.
$m[1,3], m[2,4], ..., m[n-2,n]$.
...
$m[1,n]$.
Return $m[1,n]$ as the optimum.

Running time:

1. # of DP entries: $O(n^2)$
2. RT for computing each entry:

Matrix chain multiplication

4. USE DP for Speed-up: Bottom-up DP description in words; and 5. RT Analysis

Set up a DP table with entries $m[i, j]$ where $1 \leq i \leq j \leq n$.
Compute $m[i, j]$ using the recursion in the following order:
$m[1, 1], m[2, 2], ..., m[n, n]$.
$m[1, 2], m[2, 3], ..., m[n-1, n]$.
$m[1, 3], m[2, 4], ..., m[n-2, n]$.
...
$m[1, n]$.
Return $m[1, n]$ as the optimum.

Running time:

1. # of DP entries: $O(n^2)$
2. RT for computing each entry: $O(n)$:

Matrix chain multiplication

4. USE DP for Speed-up: Bottom-up DP description in words; and 5. RT Analysis

Set up a DP table with entries $m[i,j]$ where $1 \leq i \leq j \leq n$.
Compute $m[i,j]$ using the recursion in the following order:
$m[1,1], m[2,2], ..., m[n,n]$.
$m[1,2], m[2,3], ..., m[n-1,n]$.
$m[1,3], m[2,4], ..., m[n-2,n]$.
...
$m[1,n]$.
Return $m[1,n]$ as the optimum.

Running time:

1. # of DP entries: $O(n^2)$
2. RT for computing each entry: $O(n)$:
3. RT for computing the optimum:

Matrix chain multiplication

4. USE DP for Speed-up: Bottom-up DP description in words; and 5. RT Analysis

Set up a DP table with entries $m[i, j]$ where $1 \leq i \leq j \leq n$.
Compute $m[i, j]$ using the recursion in the following order:
$m[1, 1], m[2, 2], ..., m[n, n]$.
$m[1, 2], m[2, 3], ..., m[n-1, n]$.
$m[1, 3], m[2, 4], ..., m[n-2, n]$.
...
$m[1, n]$.
Return $m[1, n]$ as the optimum.

Running time:

1. # of DP entries: $O(n^2)$
2. RT for computing each entry: $O(n)$:
3. RT for computing the optimum: $O(n^3)$

Complete Description and RT Analysis of Bottom-up DP Algorithm for Computing the Optimum

To get full points for a DP algorithm (bottom-up) description, you must state

1. DP table entries
2. Recursion
3. In which order you compute the entries
4. What is the optimum?

Complete Description and RT Analysis of Bottom-up DP Algorithm for Computing the Optimum

To get full points for a DP algorithm (bottom-up) description, you must state

1. DP table entries
2. Recursion
3. In which order you compute the entries
4. What is the optimum?

To get full points for analysis of RT, you must state

1. # of DP entries
2. RT for computing each entry
3. RT for computing the optimum

Matrix chain multiplication

6. Finding an Optimum Solution

```
MATRIX-CHAIN-ORDER(p)
n = p.length - 1
let m[1..n, 1..n] and s[1..n-1, 2..n] be new tables
for i = 1 to n
    m[i, i] = 0
for l = 2 to n                    // l is the chain length
    for i = 1 to n - l + 1
        j = i + l - 1
        m[i, j] = ∞
        for k = i to j - 1
            q = m[i, k] + m[k+1, j] + p_{i-1} p_k p_j
            if q < m[i, j]
                m[i, j] = q
                s[i, j] = k
return m and s
```

Matrix chain multiplication

6. Finding an Optimum Solution

For each subproblem, remember where we should make the first split: Define $s[i, j]$: $s[i, j] = k$ implies that there is an optimal solution for $A_{i \ldots j}$ that is constructed by $A_{i \ldots k} \times A_{k+1 \ldots j}$.

Matrix chain multiplication

6. Finding an Optimum Solution

Set up a DP table with entries $m[i, j]$ where $1 \leq i \leq j \leq n$.
Compute $m[i, j]$ using the recursion in the following order:
$m[1, 1], m[2, 2], ..., m[n, n]$.
$m[1, 2], m[2, 3], ..., m[n-1, n]$.
...
$m[1, n]$. (When computing $m[i, j]$, we also compute $s[i, j]$; here $s[i, j] = k$ implies that there is an optimal solution for $A_{i...j}$ that is constructed by $A_{i...k} \times A_{k+1...j}$.)

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

Matrix chain multiplication

6. Finding an Optimum Solution

Set up a DP table with entries $m[i,j]$ where $1 \le i \le j \le n$.
Compute $m[i,j]$ using the recursion in the following order:
$m[1,1], m[2,2], ..., m[n,n]$.
$m[1,2], m[2,3], ..., m[n-1,n]$.
...
$m[1,n]$. (When computing $m[i,j]$, we also compute $s[i,j]$; here $s[i,j] = k$ implies that there is an optimal solution for $A_{i...j}$ that is constructed by $A_{i...k} \times A_{k+1...j}$.)

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

Call Print-Optimal-Parens(s, 1, n)

Longest common subsequence

Input: Two sequences, $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$
Goal: Find a longest subsequence common to both.

Def: Z is a subsequence of X iff Z can be obtained by deleting 0 or more elements from X.

Is the following a subsequence of BCBAE?
BB: Yes.
BCE: Yes.
AB: No.

Longest common subsequence

- Each subproblem is computing a LCS of the prefixes of X and Y.
- $X_i := \langle x_1, x_2, ...x_i \rangle$ and $Y_j := \langle y_1, y_2, ...y_j \rangle$
- Define $c[i, j] :=$ length of LCS of X_i and Y_j

Longest common subsequence

$c[i,j] :=$ length of LCS of X_i and Y_i

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

The optimum is $C[m,n]$.

X= ABCBDABE

Y=BDCABAE

C[i,j] DP Table

	i \ j	0	1	2	3	4	5	6	7
			B	D	C	A	B	A	E
	0								
A	1								
B	2								
C	3								
B	4								
D	5								
A	6								
B	7								
E	8								

Longest common subsequence

Compute length of LCS of X and Y:
Compute $c[i,j]$, $0 \leq i \leq m$, $0 \leq j \leq n$ in row-major order using the recursion.
Return $c[m,n]$.

Analysis

of DP tables (subproblems): $\Theta(mn)$.
RT for computing each entry: $\Theta(1)$.
RT: $\Theta(mn)$.

Memory usage: $\Theta(mn)$. If we only need to compute LCS length, we only need $O(m+n)$ memory.

Longest common subsequence

```
LCS-LENGTH(X, Y)
m = X.length
n = Y.length
let b[1..m, 1..n] and c[0..m, 0..n] be new tables
for i = 1 to m
    c[i, 0] = 0
for j = 0 to n
    c[0, j] = 0
for i = 1 to m
    for j = 1 to n
        if x_i == y_j
            c[i, j] = c[i − 1, j − 1] + 1
            b[i, j] = "↖"
        elseif c[i − 1, j] ≥ c[i, j − 1]
            c[i, j] = c[i − 1, j]
            b[i, j] = "↑"
        else c[i, j] = c[i, j − 1]
            b[i, j] = "←"
return c and b
```

Longest common subsequence

```
PRINT-LCS(b, X, i, j)
if i == 0 or j == 0
    return
if b[i, j] == "↖"
    PRINT-LCS(b, X, i − 1, j − 1)
    print x_i
elseif b[i, j] == "↑"
    PRINT-LCS(b, X, i − 1, j)
else PRINT-LCS(b, X, i, j − 1)
```