Dynamic Programming

Dr. Navjot Singh

Design and Analysis of Algorithms

Dynamic Programming

- Dynamic Programming is an algorithm design technique for optimization problems: often minimizing or maximizing.
- Invented by American mathematician Richard Bellman in the 1950s to solve optimization problems.
- “Programming” here means “planning”.
- Like divide and conquer, DP solves problems by combining solutions to subproblems.
- Unlike divide and conquer, subproblems are not independent.
  - Subproblems may share subsubproblems,
  - However, solution to one subproblem may not affect the solutions to other subproblems of the same problem.

Dynamic Programming

- DP reduces computation by
  - Solving subproblems in a bottom-up fashion.
  - Storing solution to a subproblem the first time it is solved.
  - Looking up the solution when subproblem is encountered again.
- Key Elements:
  - Optimal Substructure
  - Overlapping Subproblems

Optimal Substructure

- Show that a solution to a problem consists of making a choice, which leaves one or more subproblems to solve.
- Suppose that you are given this last choice that leads to an optimal solution.
- Given this choice, determine which subproblems arise and how to characterize the resulting space of subproblems.
- Show that the solutions to the subproblems used within the optimal solution must themselves be optimal. Usually use cut-and-paste.
- Need to ensure that a wide enough range of choices and subproblems are considered.

Optimal Substructure

- Optimal substructure varies across problem domains:
  - *How many subproblems* are used in an optimal solution.
  - *How many choices* in determining which subproblem(s) to use.
- Informally, running time depends on (# of subproblems overall) $\times$ (# of choices).
- How many subproblems and choices do the examples considered contain?
- Dynamic programming uses optimal substructure **bottom-up**.
  - *First* find optimal solutions to subproblems.
  - *Then* choose which to use in optimal solution to the problem.

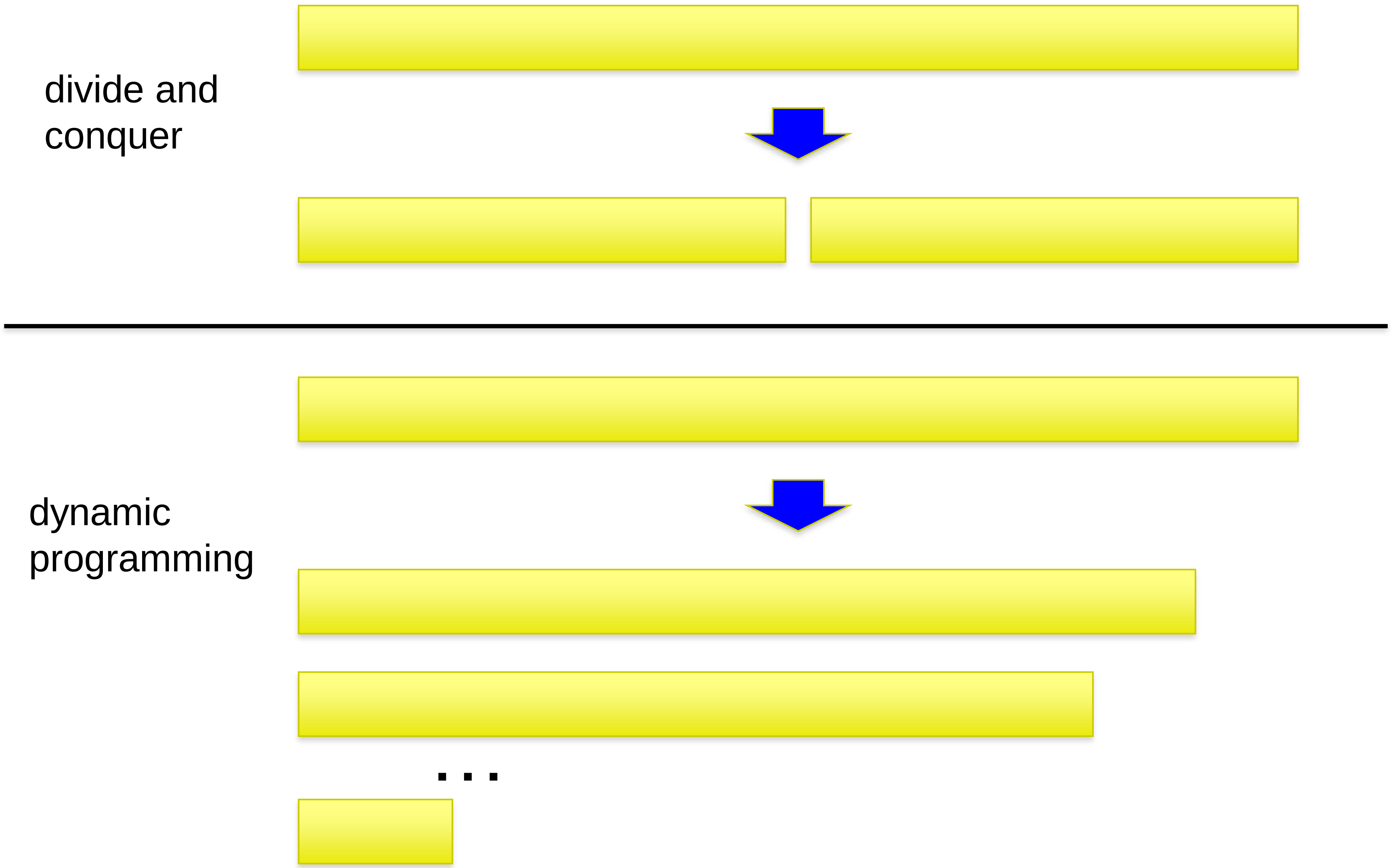
Optimal Substucture

- Does optimal substructure apply to all optimization problems? No.
- Applies to determining the shortest path but NOT the longest simple path of an unweighted directed graph.
- Why?
  - Shortest path has independent subproblems.
    - Solution to one subproblem does not affect solution to another subproblem of the same problem.
  - Subproblems are not independent in longest simple path.
    - Solution to one subproblem affects the solutions to other subproblems.

Overlapping Subproblems

- The space of subproblems must be "small".
- The total number of distinct subproblems is a polynomial in the input size.
  - A recursive algorithm is exponential because it solves the same problems repeatedly.
  - If divide-and-conquer is applicable, then each problem solved will be brand new.

Overlapping sub-problems



Steps in Dynamic Programming

1. Characterize structure of an optimal solution.
2. Define value of optimal solution recursively.
3. Compute optimal solution values either top-down with caching or bottom-up in a table.
4. Construct an optimal solution from computed values.

Fibonacci numbers

1, 1, 2, 3, 5, 8, 13, 21, 34, …

What is the recurrence for the n^{th} Fibonacci number?

F(n) = F(n-1) + F(n-2)

The solution for n is defined with respect to the solution to smaller problems (n-1 and n-2)

Fibonacci: a first attempt

```
FIBONACCI(n)
if n = 1 or n = 2
        return 1
else
        return FIBONACCI(n − 1) + FIBONACCI(n − 2)
```

Is it correct?

```
FIBONACCI(n)
if n = 1 or n = 2
        return 1
else
        return FIBONACCI(n − 1) + FIBONACCI(n − 2)
```

$$F(n) = F(n-1) + F(n-2)$$

Running time

```
FIBONACCI(n)
if n = 1 or n = 2
        return 1
else
        return FIBONACCI(n − 1) + FIBONACCI(n − 2)
```

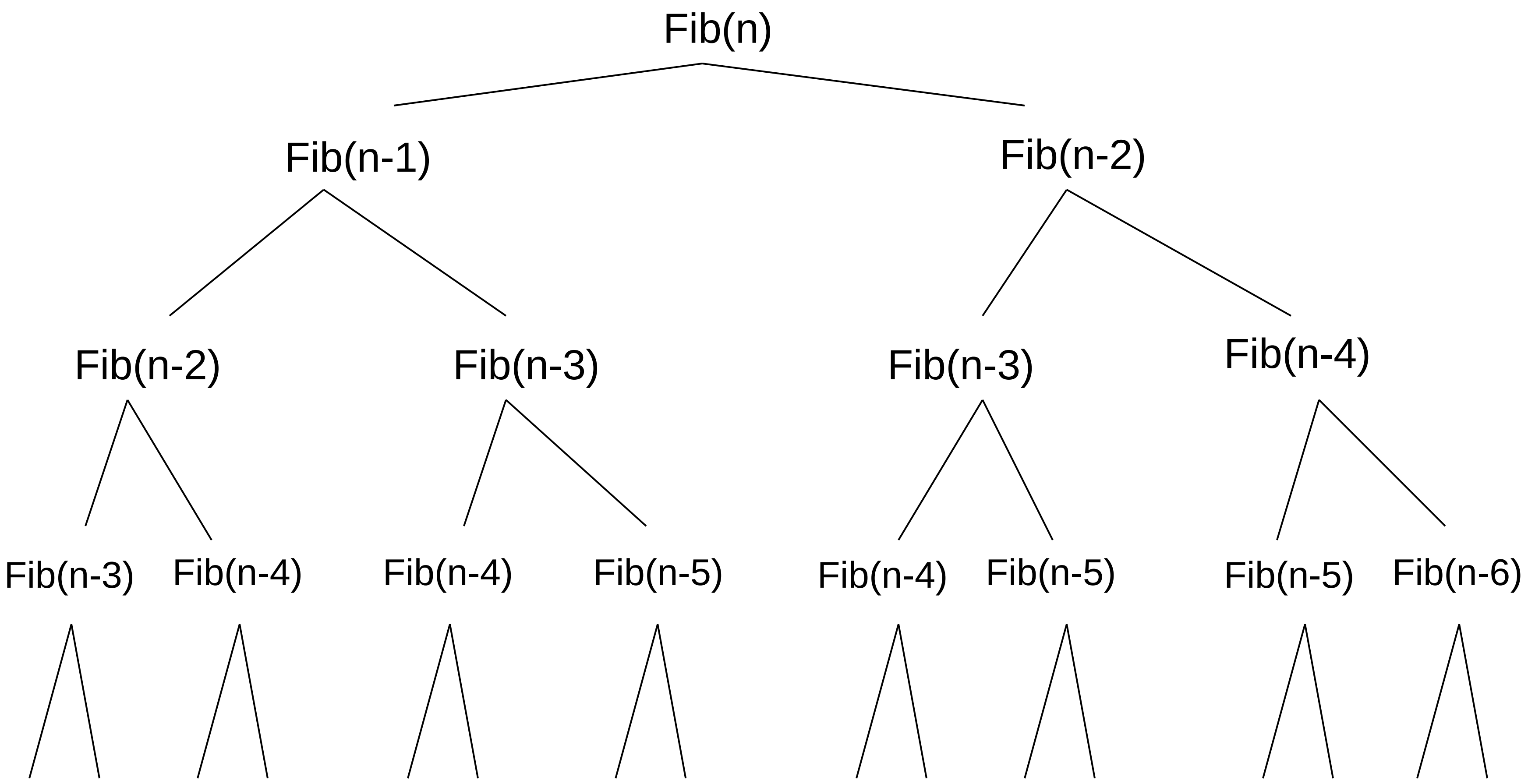
Each call creates two recursive calls

Each call reduces the size of the problem by 1 or 2

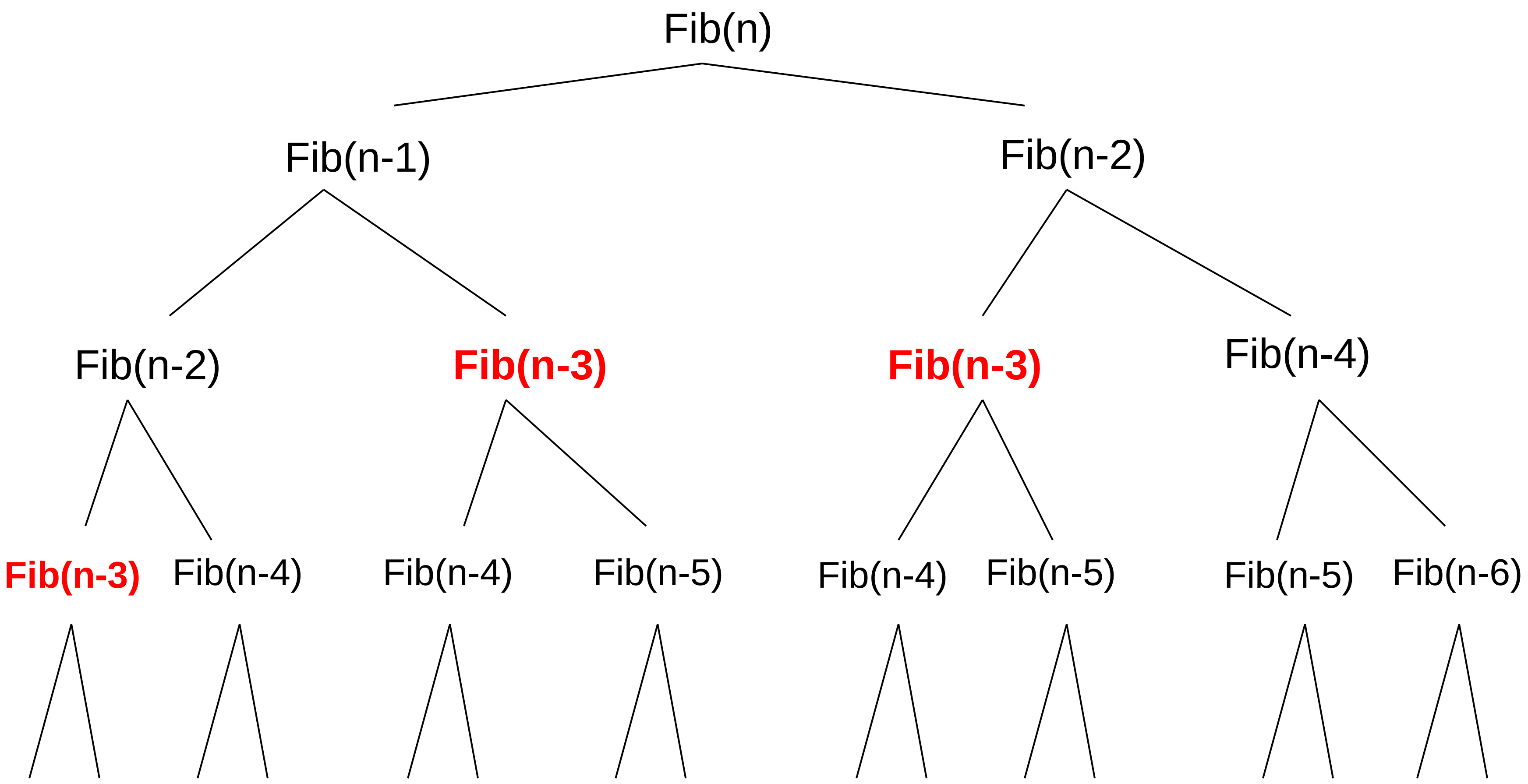
Creates a full binary of depth n

$O(2^n)$

Can we do better?



A lot of repeated work!



Creating a dynamic programming solution

Step 1: Identify a solution to the problem with respect to **smaller** subproblems (pretend like you have a solver, but it only works on smaller problems):

- F(n) = F(n-1) + F(n-2)

Step 2: **bottom up** - start with solutions to the smallest problems and build solutions to the larger problems

use an array to store solutions to subproblems

$\textsc{Fibonacci-DP}(n)$

```
1  fib[1] ← 1
2  fib[2] ← 1
3  for i ← 3 to n
4          fib[i] ← fib[i − 1] + fib[i − 2]
5  return fib[n]
```

Is it correct?

```
FIBONACCI-DP(n)
fib[1] ← 1
fib[2] ← 1
for i ← 3 to n
        fib[i] ← fib[i − 1] + fib[i − 2]
return fib[n]
```

$$F(n) = F(n-1) + F(n-2)$$

Running time?

FIBONACCI-DP(n)

$fib[1] \leftarrow 1$
$fib[2] \leftarrow 1$
for $i \leftarrow 3$ **to** n
    $fib[i] \leftarrow fib[i-1] + fib[i-2]$
return $fib[n]$

$\Theta(n)$

Longest common subsequence (LCS)

For a sequence $X = x_1, x_2, \ldots, x_m$, a subsequence is a subset of the sequence defined by a set of increasing indices $(i_1, i_2, \ldots, i_k)$ where
$1 \le i_1 < i_2 < \ldots < i_k \le m$

X = A B A C D A B A B

ABA?

Longest common subsequence (LCS)

For a sequence $X = x_1, x_2, \ldots, x_m$, a subsequence is a subset of the sequence defined by a set of increasing indices $(i_1, i_2, \ldots, i_k)$ where
$1 \leq i_1 < i_2 < \ldots < i_k \leq m$

X = A B A C D A B A B

ABA

Longest common subsequence (LCS)

For a sequence $X = x_1, x_2, \ldots, x_m$, a subsequence is a subset of the sequence defined by a set of increasing indices $(i_1, i_2, \ldots, i_k)$ where
$1 \leq i_1 < i_2 < \ldots < i_k \leq m$

X = A B A C D A B A B

ACA?

Longest common subsequence (LCS)

For a sequence $X = x_1, x_2, \ldots, x_m$, a subsequence is a subset of the sequence defined by a set of increasing indices $(i_1, i_2, \ldots, i_k)$ where
$1 \le i_1 < i_2 < \ldots < i_k \le m$

X = A B A C D A B A B

ACA

Longest common subsequence (LCS)

For a sequence $X = x_1, x_2, \ldots, x_m$, a subsequence is a subset of the sequence defined by a set of increasing indices $(i_1, i_2, \ldots, i_k)$ where
$1 \leq i_1 < i_2 < \ldots < i_k \leq m$

X = A B A C D A B A B

DCA?

Longest common subsequence (LCS)

For a sequence $X = x_1, x_2, \ldots, x_m$, a subsequence is a subset of the sequence defined by a set of increasing indices $(i_1, i_2, \ldots, i_k)$ where
$1 \le i_1 < i_2 < \ldots < i_k \le m$

X = A B A C D A B A B

~~DCA~~

Longest common subsequence (LCS)

For a sequence $X = x_1, x_2, \ldots, x_m$, a subsequence is a subset of the sequence defined by a set of increasing indices $(i_1, i_2, \ldots, i_k)$ where
$1 \le i_1 < i_2 < \ldots < i_k \le m$

X = A B A C D A B A B

AADAA?

Longest common subsequence (LCS)

For a sequence $X = x_1, x_2, \ldots, x_m$, a subsequence is a subset of the sequence defined by a set of increasing indices $(i_1, i_2, \ldots, i_k)$ where
$1 \leq i_1 < i_2 < \ldots < i_k \leq m$

X = A B A C D A B A B

AADAA

LCS problem

Given two sequences $X = x_1, x_2, \ldots, x_m$ and $Y = y_1, y_2, \ldots, y_n$, a **common subsequence** is a subsequence that occurs in both X and Y

What is the longest common subsequence?

X = A B C B D A B

Y = B D C A B A

LCS problem

Given two sequences $X = x_1, x_2, \ldots, x_m$ and $Y = y_1, y_2, \ldots, y_n$, a **common subsequence** is a subsequence that occurs in both X and Y

What is the longest common subsequence?

X = A B C B D A B

Y = B D C A B A

Naïve Algorithm

- For every subsequence of X, check whether it's a subsequence of Y .
- Time: $\Theta(n2^m)$.
  - 2^m subsequences of X to check.
  - Each subsequence takes $\Theta(n)$ time to check: scan Y for first letter, for second, and so on.

Step 1: Define the problem with respect to subproblems

X = A B C B D A B

Y = B D C A B A

Assume you have a solver for smaller problems

Step 1: Define the problem with respect to subproblems

X = A B C B D A ?

Y = B D C A B ?

Is the last character part of the LCS?

Step 1: Define the problem with respect to subproblems

X = A B C B D A ?

Y = B D C A B ?

Two cases: either the characters are the same or they're different

Step 1: Define the problem with respect to subproblems

$X = \boxed{A\ B\ C\ B\ D\ A}\ A$

LCS

$Y = \boxed{B\ D\ C\ A\ B}\ A$

The characters are part of the LCS

What is the recursive relationship?

If they're the same

Step 1: Define the problem with respect to subproblems

X = $\boxed{\text{A B C B D A}}$ B

LCS

Y = $\boxed{\text{B D C A B A}}$

If they're different

Step 1: Define the problem with respect to subproblems

X = $\boxed{\text{A B C B D A B}}$

LCS

Y = $\boxed{\text{B D C A B}}$ A

If they're different

Step 1: Define the problem with respect to subproblems

X = $\boxed{\text{A B C B D A}}$ B

Y = $\boxed{\text{B D C A B A}}$

?

X = $\boxed{\text{A B C B D A B}}$

Y = $\boxed{\text{B D C A B}}$ A

If they're different

Optimal Substructure

Theorem

Let $Z = \langle z_1, \ldots, z_k \rangle$ be any LCS of X and Y.

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.
2. If $x_m \neq y_n$, then either $z_k \neq x_m$ and Z is an LCS of X_{m-1} and Y.
3. or $z_k \neq y_n$ and Z is an LCS of X and Y_{n-1}.

Notation:

prefix $X_i = \langle x_1, \ldots, x_i \rangle$ is the first i letters of X.

Optimal Substructure

Theorem

Let $Z = \langle z_1, \ldots, z_k \rangle$ be any LCS of X and Y.

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and $Z_{k\text{-}1}$ is an LCS of $X_{m\text{-}1}$ and $Y_{n\text{-}1}$.
2. If $x_m \neq y_n$, then either $z_k \neq x_m$ and Z is an LCS of $X_{m\text{-}1}$ and Y.
3. or $z_k \neq y_n$ and Z is an LCS of X and $Y_{n\text{-}1}$.

Proof: (case 1: $x_m = y_n$)

Any sequence Z' that does not end in $x_m = y_n$ can be made longer by adding $x_m = y_n$ to the end. Therefore,

(1) longest common subsequence (LCS) Z must end in $x_m = y_n$.

(2) $Z_{k\text{-}1}$ is a common subsequence of $X_{m\text{-}1}$ and $Y_{n\text{-}1}$, and

(3) there is no longer CS of $X_{m\text{-}1}$ and $Y_{n\text{-}1}$, or Z would not be an LCS.

Optimal Substructure

Theorem

Let $Z = \langle z_1, \ldots, z_k \rangle$ be any LCS of X and Y.

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.
2. If $x_m \neq y_n$, then either $z_k \neq x_m$ and Z is an LCS of X_{m-1} and Y.
3. or $z_k \neq y_n$ and Z is an LCS of X and Y_{n-1}.

Proof: (case 2: $x_m \neq y_n$, and $z_k \neq x_m$)

Since Z does not end in x_m,

(1) Z is a common subsequence of X_{m-1} and Y, and

(2) there is no longer CS of X_{m-1} and Y, or Z would not be an LCS.

Recursive Solution

- Define $c[i, j]$ = length of LCS of X_i and Y_j .
- We want $c[m,n]$.

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

This gives a recursive algorithm and solves the problem.

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

	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i							
1	A							
2	B							
3	C							
4	B							
5	D							
6	A							
7	B							

Lets fill in the entries

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

	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0						
2	B	0						
3	C	0						
4	B	0						
5	D	0						
6	A	0						
7	B	0						

Need to initialize values within 1 smaller in either dimension.

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

	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	?					
2	B	0						
3	C	0						
4	B	0						
5	D	0						
6	A	0						
7	B	0						

LCS(A, B)

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

	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	0					
2	B	0						
3	C	0						
4	B	0						
5	D	0						
6	A	0						
7	B	0						

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

	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	0	0	0	?		
2	B	0						
3	C	0						
4	B	0						
5	D	0						
6	A	0						
7	B	0						

LCS(A, BDCA)

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

i	j	0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	0	0	0	1		
2	B	0						
3	C	0						
4	B	0						
5	D	0						
6	A	0						
7	B	0						

LCS(A, BDCA)

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

	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	0	0	0	1	1	1
2	B	0	1	1	1	1	2	2
3	C	0	1	1	2	2	2	2
4	B	0	1	1	2	2	?	
5	D	0						
6	A	0						
7	B	0						

LCS(ABCB, BDCAB)

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

j		0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	0	0	0	1	1	1
2	B	0	1	1	1	1	2	2
3	C	0	1	1	2	2	2	2
4	B	0	1	1	2	2	3	
5	D	0						
6	A	0						
7	B	0						

LCS(ABCB, BDCAB)

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

	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	0	0	0	1	1	1
2	B	0	1	1	1	1	2	2
3	C	0	1	1	2	2	2	2
4	B	0	1	1	2	2	3	3
5	D	0	1	2	2	2	3	3
6	A	0	1	2	2	3	3	4
7	B	0	1	2	2	3	4	4

Where's the final answer?

The algorithm

```
LCS-LENGTH(X, Y)
m ← length[X]
n ← length[Y]
c[0, 0] ← 0
for i ← 1 to m
        c[i, 0] ← 0
for j ← 1 to n
        c[0, j] ← 0
for i ← 1 to m
        for j ← 1 to n
                if x_i = y_i
                        c[i, j] ← 1 + c[i − 1, j − 1]
                elseif c[i − 1, j] > c[i, j − 1]
                        c[i, j] ← c[i − 1, j]
                else
                        c[i, j] ← c[i, j − 1]
return c[m, n]
```

The algorithm

```
LCS-LENGTH(X, Y)
m ← length[X]
n ← length[Y]
c[0, 0] ← 0
for i ← 1 to m
        c[i, 0] ← 0
for j ← 1 to n
        c[0, j] ← 0
for i ← 1 to m
        for j ← 1 to n
                if x_i = y_i
                        c[i, j] ← 1 + c[i − 1, j − 1]
                elseif c[i − 1, j] > c[i, j − 1]
                        c[i, j] ← c[i − 1, j]
                else
                        c[i, j] ← c[i, j − 1]
return c[m, n]
```

Base case initialization

The algorithm

```
LCS-LENGTH(X, Y)
m ← length[X]
n ← length[Y]
c[0, 0] ← 0
for i ← 1 to m
        c[i, 0] ← 0
for j ← 1 to n
        c[0, j] ← 0
for i ← 1 to m
        for j ← 1 to n
                if x_i = y_i
                        c[i, j] ← 1 + c[i − 1, j − 1]
                elseif c[i − 1, j] > c[i, j − 1]
                        c[i, j] ← c[i − 1, j]
                else
                        c[i, j] ← c[i, j − 1]
return c[m, n]
```

Fill in the matrix

The algorithm

```
LCS-LENGTH(X, Y)
m ← length[X]
n ← length[Y]
c[0, 0] ← 0
for i ← 1 to m
        c[i, 0] ← 0
for j ← 1 to n
        c[0, j] ← 0
for i ← 1 to m
        for j ← 1 to n
                if x_i = y_i
                        c[i, j] ← 1 + c[i − 1, j − 1]
                elseif c[i − 1, j] > c[i, j − 1]
                        c[i, j] ← c[i − 1, j]
                else
                        c[i, j] ← c[i, j − 1]
return c[m, n]
```

The algorithm

```
LCS-LENGTH(X, Y)
m ← length[X]
n ← length[Y]
c[0, 0] ← 0
for i ← 1 to m
        c[i, 0] ← 0
for j ← 1 to n
        c[0, j] ← 0
for i ← 1 to m
        for j ← 1 to n
                if x_i = y_i
                        c[i, j] ← 1 + c[i − 1, j − 1]
                elseif c[i − 1, j] > c[i, j − 1]
                        c[i, j] ← c[i − 1, j]
                else
                        c[i, j] ← c[i, j − 1]
return c[m, n]
```

The algorithm

```
LCS-LENGTH(X, Y)
m ← length[X]
n ← length[Y]
c[0, 0] ← 0
for i ← 1 to m
        c[i, 0] ← 0
for j ← 1 to n
        c[0, j] ← 0
for i ← 1 to m
        for j ← 1 to n
                if x_i = y_i
                        c[i, j] ← 1 + c[i − 1, j − 1]
                elseif c[i − 1, j] > c[i, j − 1]
                        c[i, j] ← c[i − 1, j]
                else
                        c[i, j] ← c[i, j − 1]
return c[m, n]
```

Running time?

LCS-LENGTH(X, Y)

```
m ← length[X]
n ← length[Y]
c[0, 0] ← 0
for i ← 1 to m
        c[i, 0] ← 0
for j ← 1 to n
        c[0, j] ← 0
for i ← 1 to m
        for j ← 1 to n
                if x_i = y_i
                        c[i, j] ← 1 + c[i − 1, j − 1]
                elseif c[i − 1, j] > c[i, j − 1]
                        c[i, j] ← c[i − 1, j]
                else
                        c[i, j] ← c[i, j − 1]
return c[m, n]
```

$\Theta(nm)$

Keeping track of the solution

LCS algorithm only calculated the length of the LCS between X and Y

What if we wanted to know the actual sequence?

Keep track of this as well…

```
for i ← 1 to m
    for j ← 1 to n
        if x_i = y_i
            c[i, j] ← 1 + c[i − 1, j − 1]
        elseif c[i − 1, j] > c[i, j − 1]
            c[i, j] ← c[i − 1, j]
        else
            c[i, j] ← c[i, j − 1]
return c[m, n]
```

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

i	j	0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	0	0	0	1	1	1
2	B	0	1	1	1	1	2	2
3	C	0	1	1	2	2	2	2
4	B	0	1	1	2	2	3	3
5	D	0	1	2	2	2	3	3
6	A	0	1	2	2	3	3	4
7	B	0	1	2	2	3	4	4

We can follow the arrows to generate the solution

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

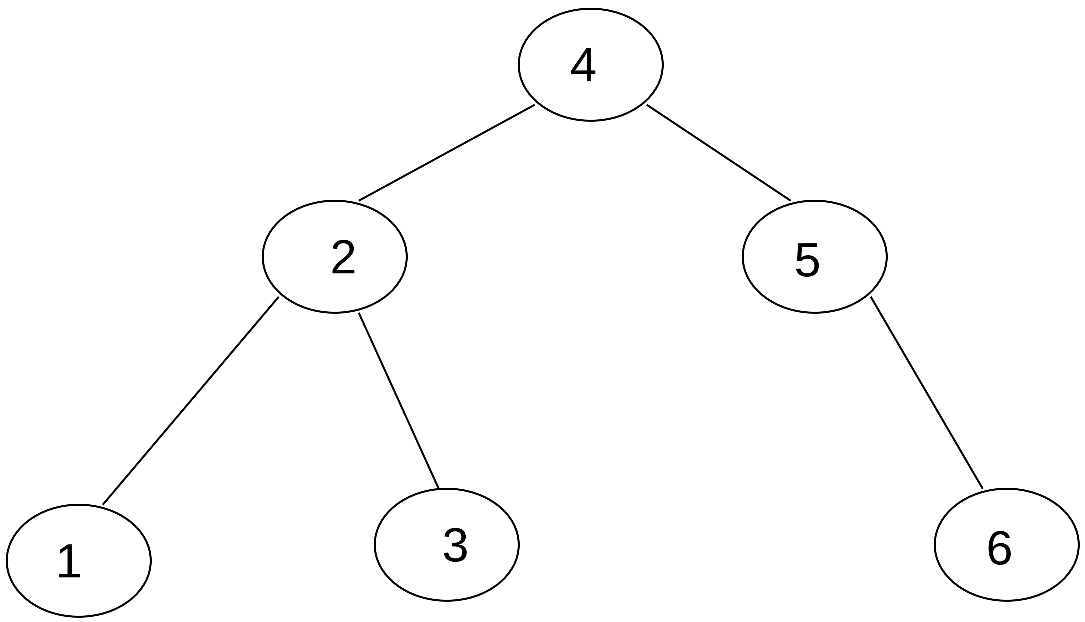
i	j	0	1	2	3	4	5	6
	y_j		B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	0	0	0	1	1	1
2	B	0	1	1	1	1	2	2
3	C	0	1	1	2	2	2	2
4	B	0	1	1	2	2	3	3
5	D	0	1	2	2	2	3	3
6	A	0	1	2	2	3	3	4
7	B	0	1	2	2	3	4	4

We can follow the arrows to generate the solution

BCBA

Counting binary search trees

How many unique binary search trees can be created using the numbers 1 through n?

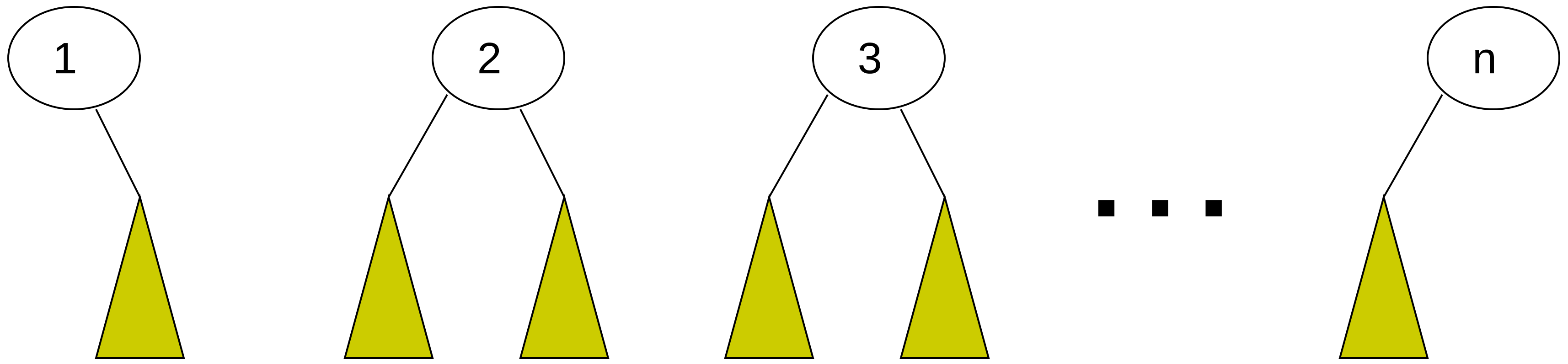


Step 1:
What is the subproblem?

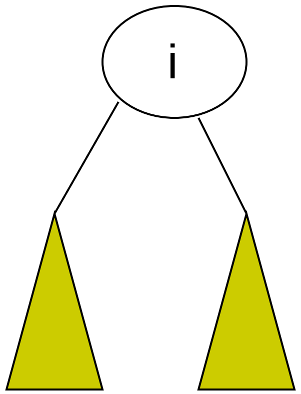
Assume we have some black box solver (call it T) that can give us the answer to smaller subproblems

How can we use the answer from this to answer our question?

How many options for the root are there?

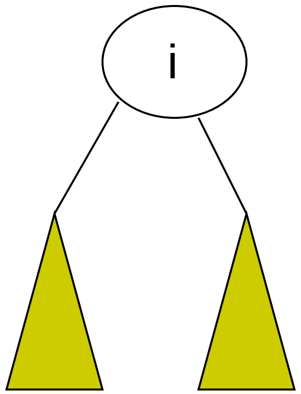


Subproblems



How many trees have i as the root?

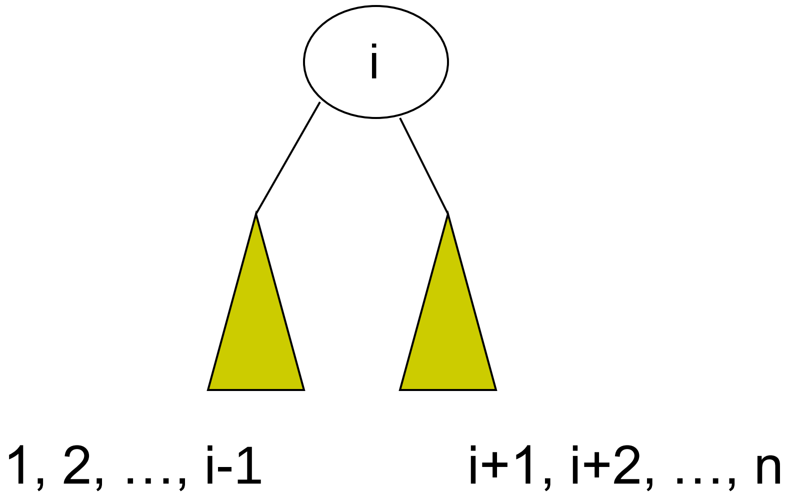
Subproblems



1, 2, …, i-1 i+1, i+2, …, n

?

Subproblems

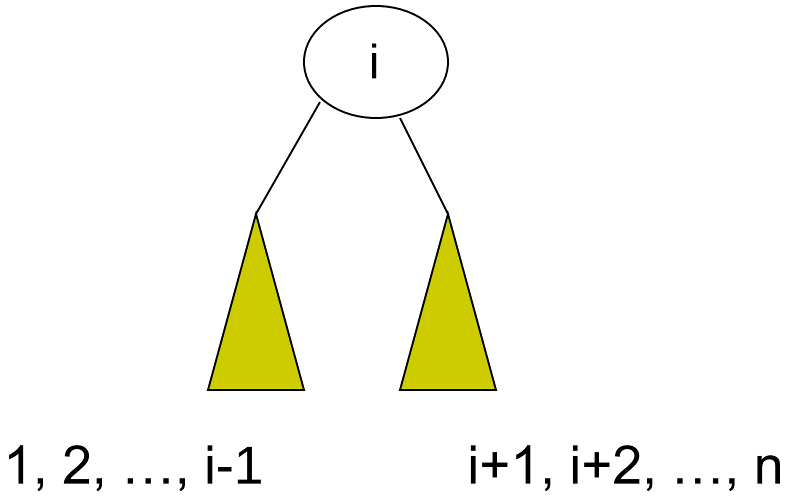


T(i-1)

?

subproblem of
size i-1

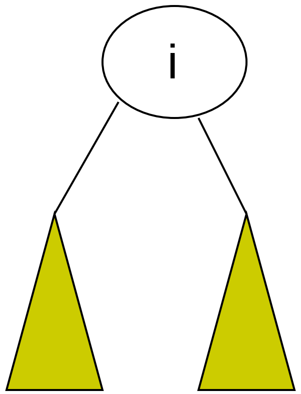
Subproblems



T(i-1)

Number of trees for i+1, i+2, …, i+n is the same as the number of trees from 1, 2, …, n-i

Subproblems

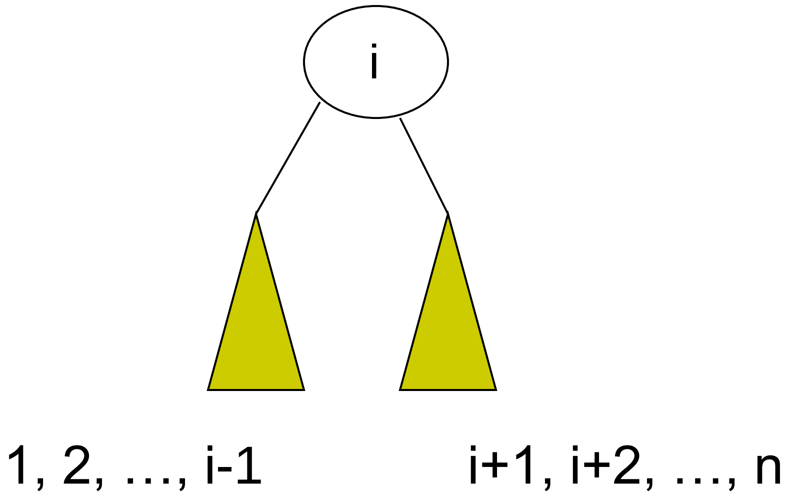


1, 2, …, i-1　　　　i+1, i+2, …, n

T(i-1)　　　　T(n-i)

Given solutions for T(i-1) and T(n-i) how many trees are there with i as the root?

Subproblems



$T(i-1)$ $T(n-i)$

$$T(i) = T(i-1) * T(n-i)$$

Step 1: define the answer with respect to subproblems

T(i) = T(i-1) * T(n-i)

$$T(n)=\sum_{i=1}^{n} T(i-1)*T(n-i)$$

```
BST-COUNT(n)
if n = 0
        return 1
else
        sum = 0
        for i ← 1 to n
                sum ← sum + BST-COUNT(i − 1) * BST-COUNT(n − i)
return sum
```

Is there a problem?

```
BST-COUNT(n)
if n = 0
        return 1
else
        sum = 0
        for i ← 1 to n
                sum ← sum + BST-COUNT(i − 1) * BST-COUNT(n − i)
return sum
```

As with Fibonacci, we're repeating a lot of work

Step 2: Generate a solution from the bottom-up

```
BST-COUNT(n)
if n = 0
        return 1
else
        sum = 0
        for i ← 1 to n
                sum ← sum + BST-COUNT(i − 1) ∗ BST-COUNT(n − i)
return sum
```

```
BST-COUNT-DP(n)
c[0] = 1
c[1] = 1
for k ← 2 to n
        c[k] ← 0
        for i ← 1 to k
                c[k] ← c[k] + c[i − 1] ∗ c[k − i]
return c[n]
```

BST-COUNT-DP(n)

```
c[0] = 1
c[1] = 1
for k ← 2 to n
        c[k] ← 0
        for i ← 1 to k
                c[k] ← c[k] + c[i − 1] * c[k − i]
return c[n]
```

0 1 2 3 4 5 … n

BST-COUNT-DP(n)

```
c[0] = 1
c[1] = 1
for k ← 2 to n
        c[k] ← 0
        for i ← 1 to k
                c[k] ← c[k] + c[i − 1] * c[k − i]
return c[n]
```

1	1						
0	1	2	3	4	5	…	n

BST-COUNT-DP(n)

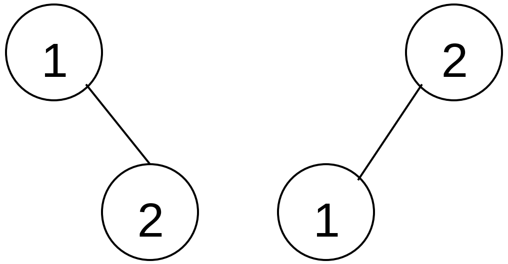
```
c[0] = 1
c[1] = 1
for k ← 2 to n
        c[k] ← 0
        for i ← 1 to k
                c[k] ← c[k] + c[i − 1] * c[k − i]
return c[n]
```

c[0]*c[1] + c[1]*c[0]

1	1						
0	1	2	3	4	5	…	n

BST-COUNT-DP(n)

```
c[0] = 1
c[1] = 1
for k ← 2 to n
        c[k] ← 0
        for i ← 1 to k
                c[k] ← c[k] + c[i − 1] * c[k − i]
return c[n]
```



c[0]*c[1] + c[1]*c[0]

1 1

0 1 2 3 4 5 … n

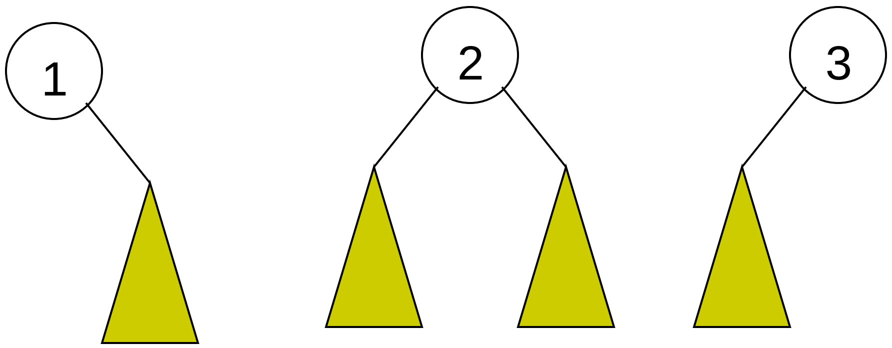
BST-COUNT-DP(n)

```
c[0] = 1
c[1] = 1
for k ← 2 to n
        c[k] ← 0
        for i ← 1 to k
                c[k] ← c[k] + c[i − 1] * c[k − i]
return c[n]
```

1	1	2					
0	1	2	3	4	5	…	n

BST-COUNT-DP(n)

```
c[0] = 1
c[1] = 1
for k ← 2 to n
        c[k] ← 0
        for i ← 1 to k
                c[k] ← c[k] + c[i − 1] * c[k − i]
return c[n]
```



c[0]*c[2] + c[1]*c[1] + c[2]*c[0]

1   1   2

0   1   2   3   4   5   …   n

BST-COUNT-DP(n)

```
c[0] = 1
c[1] = 1
for k ← 2 to n
        c[k] ← 0
        for i ← 1 to k
                c[k] ← c[k] + c[i − 1] * c[k − i]
return c[n]
```

1	1	2	5				
0	1	2	3	4	5	…	n

```
BST-COUNT-DP(n)
c[0] = 1
c[1] = 1
for k ← 2 to n
        c[k] ← 0
        for i ← 1 to k
                c[k] ← c[k] + c[i − 1] * c[k − i]
return c[n]
```

1	1	2	5	…			
0	1	2	3	4	5	…	n

Running time?

```
BST-COUNT-DP(n)
c[0] = 1
c[1] = 1
for k ← 2 to n
        c[k] ← 0
        for i ← 1 to k
                c[k] ← c[k] + c[i − 1] * c[k − i]
return c[n]
```

$\Theta(n^2)$

Optimal Binary Search Trees

- Problem
  - Given sequence $K = k_1 < k_2 < \cdots < k_n$ of n sorted keys, with a search probability p_i for each key k_i.
  - Want to build a binary search tree (BST) with minimum expected search cost.
  - Actual cost = # of items examined.
  - For key k_i, cost = $\text{depth}_T(k_i)+1$, where $\text{depth}_T(k_i)$ = depth of k_i in BST T .

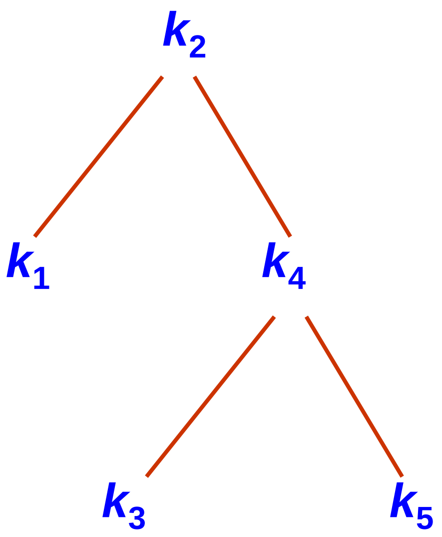
Expected Search Cost

$$
\begin{aligned}
&E[\text{search cost in } T] \\
&\quad = \sum_{i=1}^{n} (\text{depth}_T(k_i) + 1) \cdot p_i \\
&\quad = \sum_{i=1}^{n} \text{depth}_T(k_i) \cdot p_i + \sum_{i=1}^{n} p_i \\
&\quad = 1 + \sum_{i=1}^{n} \text{depth}_T(k_i) \cdot p_i
\end{aligned}
$$

Sum of probabilities is 1.

Example

- Consider 5 keys with these search probabilities:
$p_1 = 0.25$, $p_2 = 0.2$, $p_3 = 0.05$, $p_4 = 0.2$, $p_5 = 0.3$.

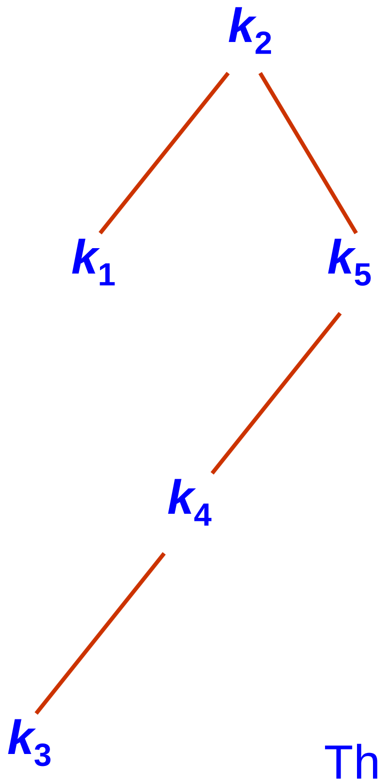


i	$\text{depth}_T(k_i)$	$\text{depth}_T(k_i) \cdot p_i$
1	1	0.25
2	0	0
3	2	0.1
4	1	0.2
5	2	0.6
		1.15

Therefore, E[search cost] = 2.15.

Example

- $p_1 = 0.25$, $p_2 = 0.2$, $p_3 = 0.05$, $p_4 = 0.2$, $p_5 = 0.3$.



i	$\text{depth}_T(k_i)$	$\text{depth}_T(k_i) \cdot p_i$
1	1	0.25
2	0	0
3	3	0.15
4	2	0.4
5	1	0.3
		1.10

Therefore, E[search cost] = 2.10.

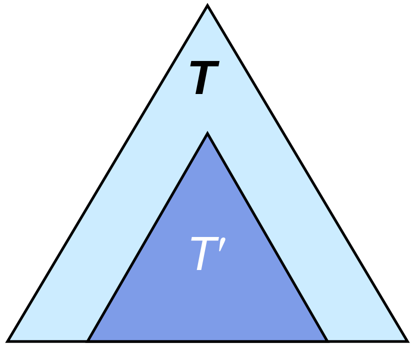
This tree turns out to be optimal for this set of keys.

Example

- **Observations:**
  - Optimal BST may not have smallest height.
  - Optimal BST may not have highest-probability key at root.
- Build by exhaustive checking?
  - Construct each n-node BST.
  - For each,
    assign keys and compute expected search cost.
  - But there are $\Omega(4^n/n^{3/2})$ different BSTs with n nodes.

Optimal Substructure

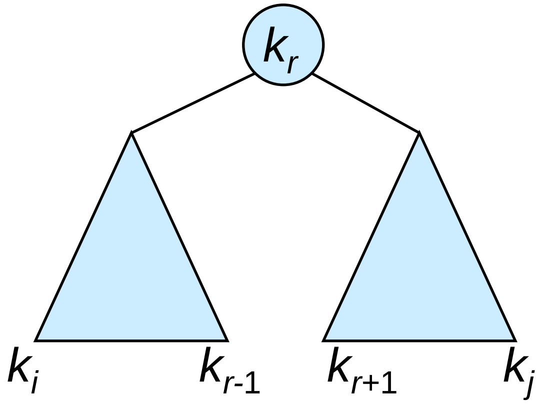
- Any subtree of a BST contains keys in a contiguous range k_i, ..., k_j for some $1 \le i \le j \le n$.



- If T is an optimal BST and
  T contains subtree T' with keys k_i, ... ,k_j,
  then T' must be an optimal BST for keys k_i, ..., k_j.

Optimal Substructure

- One of the keys in k_i, …,k_j, say k_r, where $i \leq r \leq j$, must be the root of an optimal subtree for these keys.
- Left subtree of k_r contains $k_i,\ldots,k_{r-1}$.
- Right subtree of k_r contains $k_{r+1}, \ldots, k_j$.



- To find an optimal BST:
  - Examine all candidate roots k_r , for $i \leq r \leq j$
  - Determine all optimal BSTs containing $k_i,\ldots,k_{r-1}$ and containing $k_{r+1},\ldots,k_j$

Recursive Solution

- Find optimal BST for $k_i,\ldots,k_j$, where $i \geq 1$, $j \leq n$, $j \geq i-1$. When $j = i-1$, the tree is empty.
- Define $e[i, j\,]$ = expected search cost of optimal BST for $k_i,\ldots,k_j$.

- If $j = i-1$, then $e[i, j\,] = 0$.
- If $j \geq i$,
  - Select a root k_r, for some $i \leq r \leq j$.
  - Recursively make an optimal BSTs
    - for $k_i,..,k_{r-1}$ as the left subtree, and
    - for $k_{r+1},..,k_j$ as the right subtree.

Recursive Solution

- When the OPT subtree becomes a subtree of a node:
  - Depth of every node in OPT subtree goes up by 1.
  - Expected search cost increases by

$$\boxed{w(i,j) = \sum_{l=i}^{j} p_l}$$

- If k_r is the root of an optimal BST for $k_i,..,k_j$:
  - $e[i, j\,] = p_r + (e[i, r\text{–}1] + w(i, r\text{–}1)) + (e[r\text{+}1, j] + w(r\text{+}1, j))$

    $= e[i, r\text{–}1] + e[r\text{+}1, j] + w(i, j).$ (because $w(i, j)\text{=}w(i,r\text{–}1) + p_r + w(r + 1, j)$)
- But, we don't know k_r. Hence,

$$\boxed{e[i,j] = \begin{cases} 0 & \text{if } j = i-1 \\ \min\limits_{i \le r \le j} \{e[i, r-1] + e[r+1, j] + w(i,j)\} & \text{if } i \le j \end{cases}}$$

Computing an Optimal Solution

For each subproblem (i,j), store:

- expected search cost in a table $e[1..n+1, 0..n]$
  - Will use only entries $e[i, j]$, where $j \geq i-1$.
- $root[i, j]$ = root of subtree with keys $k_i,..,k_j$, for $1 \leq i \leq j \leq n$.
- $w[1..n+1, 0..n]$ = sum of probabilities
  - $w[i, i-1] = 0$ for $1 \leq i \leq n$.
  - $w[i, j] = w[i, j-1] + p_j$ for $1 \leq i \leq j \leq n$.

Pseudo-code

```
OPTIMAL-BST(p, q, n)
    for i ← 1 to n + 1
        do e[i, i– 1] ← 0
           w[i, i– 1] ← 0
    for l ← 1 to n
        do for i ← 1 to n–l + 1
            do j ←i + l–1
               e[i, j ]←∞
               w[i, j ] ← w[i, j–1] + p_j
               for r ←i to j
                   do t ← e[i, r–1] + e[r + 1, j ] + w[i, j ]
                      if t < e[i, j ]
                         then e[i, j ] ← t
                              root[i, j ] ←r
    return e and root
```

Consider all trees with *l* keys

Fix the first key

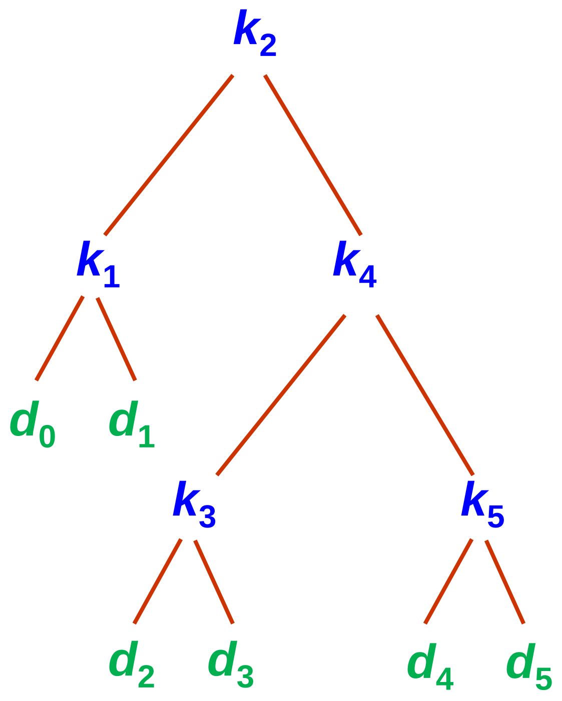
Fix the last key

Determine the root of the optimal (sub)tree

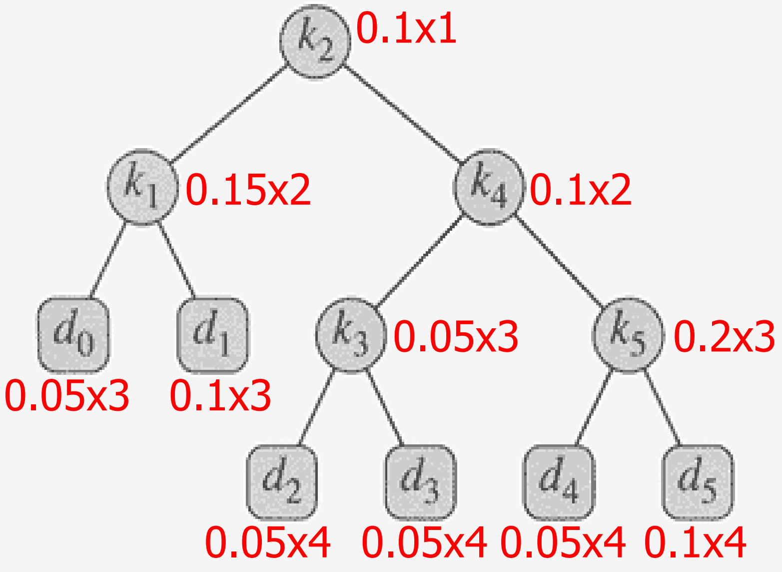
Time: $O(n^3)$

Pseudo-code with dummy keys

```
OPTIMAL-BST(p, q, n)
for i ← 1 to n + 1
    do e[i, i– 1] ← 0                    e[i, i– 1] ← q_{i-1}
       w[i, i– 1] ← 0                    w[i, i– 1] ← q_{i-1}
for l ← 1 to n
    do for i ← 1 to n–l + 1
        do j ←i + l–1
           e[i, j ]←∞
           w[i, j ] ← w[i, j–1] + p_j  + q_j
           for r ←i to j
               do t ← e[i, r–1] + e[r + 1, j ] + w[i, j ]
                  if t < e[i, j ]
                     then e[i, j ] ← t
                          root[i, j ] ←r
return e and root
```

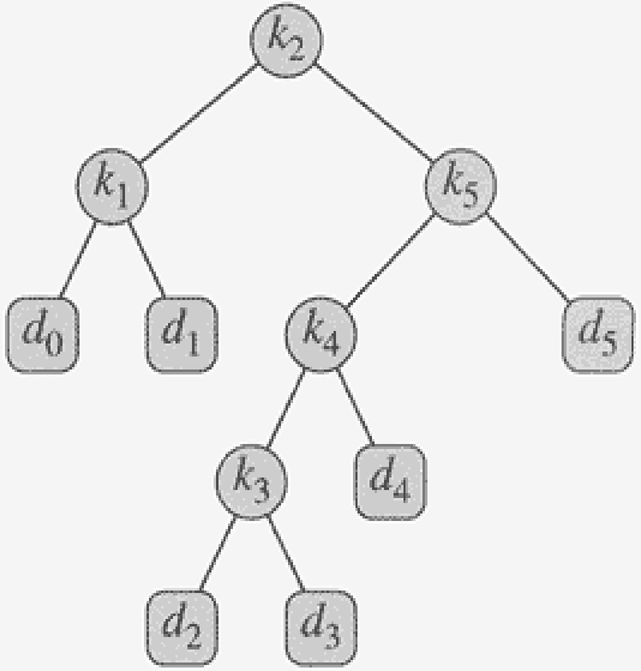


Time: $O(n^3)$



Cost of node in red: total cost 2.80

(a)



(b)

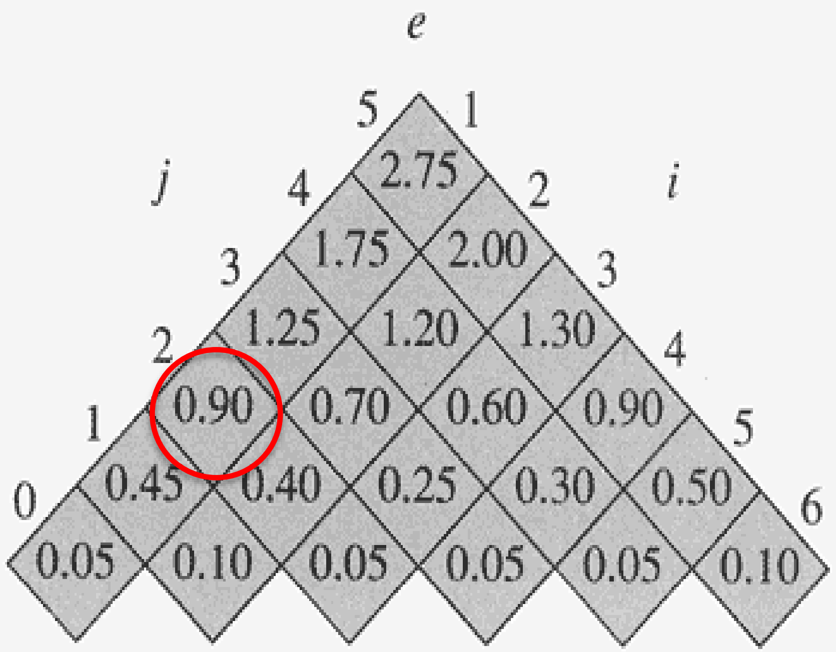
k_i are keys ($k_1 < k_2 < \ldots < k_5$), d_i are dummy values representing "space between" keys k_i and k_{i+1}.

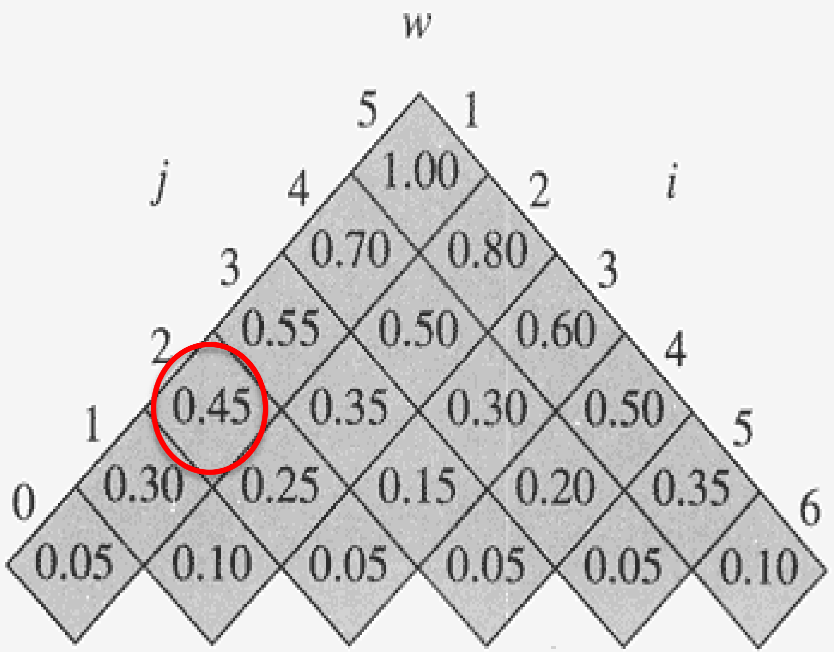
Two binary search trees for a set of $n = 5$ keys with the following probabilities:

i	0	1	2	3	4	5	
p_i		0.15	0.10	0.05	0.10	0.20	←*probability of* k_i
q_i	0.05	0.10	0.05	0.05	0.05	0.10	←*probability of* d_i

Check the expected search cost of the two trees!

(a) A binary search tree with expected search cost 2.80. **(b)** A binary search tree with expected search cost 2.75. This tree is optimal.





$e[i, j] = e[i, r-1] + e[r + 1, j] + w[i, j]$

When r=1

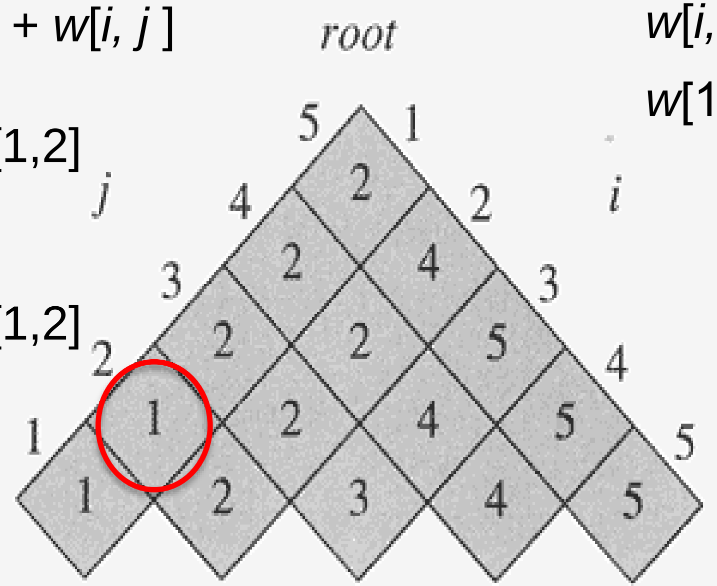
$e[1,2] = e[1,0] + e[2,2] + w[1,2]$

=0.05 + 0.40 + 0.45 = 0.90

When r=2

$e[1,2] = e[1,1] + e[3,2] + w[1,2]$

=0.45 + 0.05 + 0.45 = 0.95



$w[i, j] \leftarrow w[i, j-1] + p_j + q_j$

$w[1, 2] \leftarrow w[1, 1] + p_2 + q_2$

= 0.30 + 0.10 + 0.05 = 0.45

i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10

Matrix-Chain Multiplication

Problem: given a sequence $\langle A_1, A_2, \ldots, A_n \rangle$, compute the product:

$$A_1 \cdot A_2 \cdots A_n$$

- Matrix compatibility:

$C = A \cdot B$

$col_A = row_B$

$row_C = row_A$

$col_C = col_B$

$C = A_1 \cdot A_2 \cdots A_i \cdot A_{i+1} \cdots A_n$

$col_i = row_{i+1}$

$row_C = row_{A1}$

$col_C = col_{An}$

MATRIX-MULTIPLY(A, B)

if columns[A] $\neq$ rows[B]

  then error "incompatible dimensions"

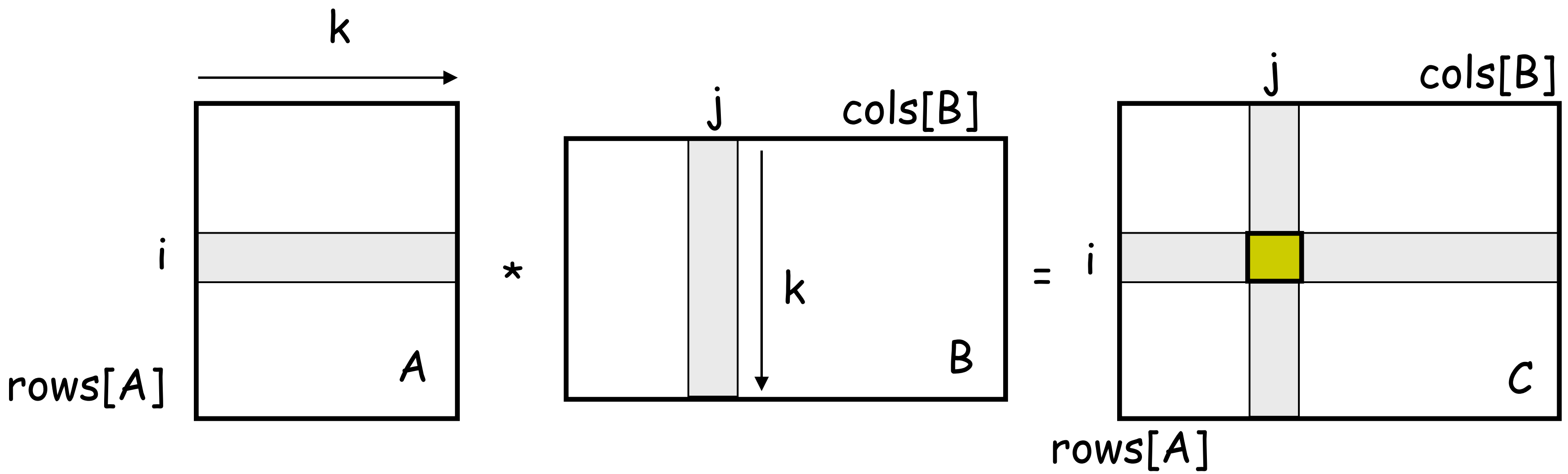
  else for i $\leftarrow$ 1 to rows[A]

    do for j $\leftarrow$ 1 to columns[B]

      do C[i, j] = 0

        for k $\leftarrow$ 1 to columns[A]

          do $C[i, j] \leftarrow C[i, j] + \overbrace{A[i, k]\, B[k, j]}^{\text{rows}[A] \cdot \text{cols}[A] \cdot \text{cols}[B] \text{ multiplications}}$



Matrix-Chain Multiplication

- In what order should we multiply the matrices?

$$A_1 \cdot A_2 \cdots A_n$$

- Parenthesize the product to get the order in which matrices are multiplied
- *E.g.:* $A_1 \cdot A_2 \cdot A_3 = ((A_1 \cdot A_2) \cdot A_3)$

$$= (A_1 \cdot (A_2 \cdot A_3))$$

- Which one of these orderings should we choose?
  - The order in which we multiply the matrices has a significant impact on the cost of evaluating the product

Example

$$A_1 \cdot A_2 \cdot A_3$$

- A_1: 10 x 100
- A_2: 100 x 5
- A_3: 5 x 50

1. $((A_1 \cdot A_2) \cdot A_3)$: $A_1 \cdot A_2 = 10 \times 100 \times 5 = 5{,}000$ (10 x 5)

   $((A_1 \cdot A_2) \cdot A_3) = 10 \times 5 \times 50 = 2{,}500$

   Total: 7,500 scalar multiplications

2. $(A_1 \cdot (A_2 \cdot A_3))$: $A_2 \cdot A_3 = 100 \times 5 \times 50 = 25{,}000$ (100 x 50)

   $(A_1 \cdot (A_2 \cdot A_3)) = 10 \times 100 \times 50 = 50{,}000$

   Total: 75,000 scalar multiplications

one order of magnitude difference!!

Matrix-Chain Multiplication: Problem Statement

- Given a chain of matrices $\langle A_1, A_2, \ldots, A_n \rangle$, where A_i has dimensions $p_{i-1} \times p_i$, fully parenthesize the product $A_1 \cdot A_2 \cdots A_n$ in a way that minimizes the number of scalar multiplications.

$$A_1 \quad \cdot \quad A_2 \quad \cdots \quad A_i \quad \cdot \quad A_{i+1} \quad \cdots \quad A_n$$

$$p_0 \times p_1 \qquad p_1 \times p_2 \qquad p_{i-1} \times p_i \qquad p_i \times p_{i+1} \qquad p_{n-1} \times p_n$$

What is the number of possible parenthesizations?

- Exhaustively checking all possible parenthesizations is not efficient!
- It can be shown that the number of parenthesizations grows as $\Omega(4^n/n^{3/2})$

1. The Structure of an Optimal Parenthesization

- Notation:

  $$A_{i\ldots j} = A_i\ A_{i+1} \cdots A_j,\ i \le j$$

- Suppose that an optimal parenthesization of $A_{i\ldots j}$ splits the product between A_k and A_{k+1}, where $i \le k < j$

$$\begin{aligned} A_{i\ldots j} &= A_i\ A_{i+1} \cdots A_j \\ &= A_i\ A_{i+1} \cdots A_k\ A_{k+1} \cdots A_j \\ &= A_{i\ldots k}\ A_{k+1\ldots j} \end{aligned}$$

Optimal Substructure

$$A_{i \ldots j} = A_{i \ldots k} \, A_{k+1 \ldots j}$$

- The parenthesization of the "prefix" $A_{i \ldots k}$ must be an optimal parentesization
- If there were a less costly way to parenthesize $A_{i \ldots k}$, we could substitute that one in the parenthesization of $A_{i \ldots j}$ and produce a parenthesization with a lower cost than the optimum $\Rightarrow$ contradiction!
- *An optimal solution to an instance of the matrix-chain multiplication contains within it optimal solutions to subproblems*

2. A Recursive Solution

- Subproblem:

  determine the minimum cost of parenthesizing

  $A_{i\ldots j} = A_i A_{i+1} \cdots A_j$ for $1 \le i \le j \le n$

- Let $m[i, j]$ = the minimum number of multiplications needed to compute $A_{i\ldots j}$
  - full problem ($A_{1..n}$): $m[1, n]$
  - $i = j$: $A_{i\ldots i} = A_i \Rightarrow m[i, i] = 0$, for $i=1,2,\ldots,n$

2. A Recursive Solution

- Consider the subproblem of parenthesizing $A_{i\ldots j} = A_i A_{i+1} \cdots A_j$ for $1 \le i \le j \le n$

$$= A_{i\ldots k} \; A_{k+1\ldots j} \quad \text{for } i \le k < j$$

$m[i, k]$ $\quad$ $p_{i-1}p_kp_j$ $\quad$ $m[k+1,j]$

- Assume that the optimal parenthesization splits the product $A_i A_{i+1} \cdots A_j$ at k $(i \le k < j)$

$$m[i, j] = \underbrace{m[i, k]}_{\text{min \# of multiplications to compute } A_{i\ldots k}} + \underbrace{m[k+1, j]}_{\text{min \# of multiplications to compute } A_{k+1\ldots j}} + \underbrace{p_{i-1}p_kp_j}_{\text{\# of multiplications to compute } A_{i\ldots k}A_{k\ldots j}}$$

2. A Recursive Solution (cont.)

$$m[i, j] = m[i, k] + m[k+1, j] + p_{i-1}p_kp_j$$

- We do not know the value of k
  - There are j – i possible values for k: k = i, i+1, …, j-1
- Minimizing the cost of parenthesizing the product $A_i A_{i+1} \cdots A_j$ becomes:

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

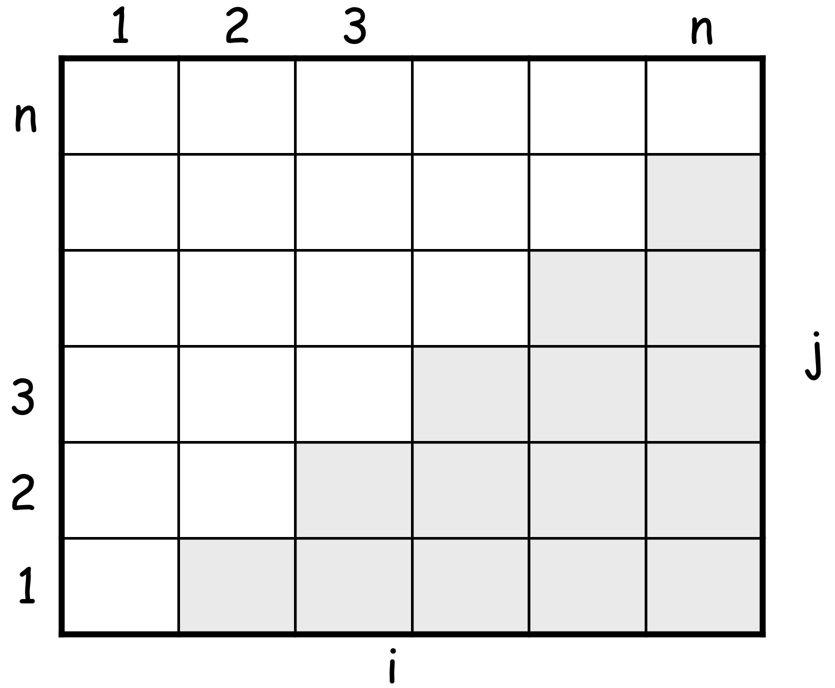
3. Computing the Optimal Costs

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

- Computing the optimal solution recursively takes exponential time!
- How many subproblems?

  $\Rightarrow \Theta(n^2)$

  - Parenthesize $A_{i\ldots j}$ for $1 \le i \le j \le n$
  - One problem for each choice of i and j



3. Computing the Optimal Costs (cont.)

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

- How do we fill in the tables m[1..n, 1..n]?
  - Determine which entries of the table are used in computing $m[i, j]$

$$A_{i\ldots j} = A_{i\ldots k}\, A_{k+1\ldots j}$$

  - Subproblems' size is one less than the original size
  - **Idea:** fill in m such that it corresponds to solving problems of increasing length

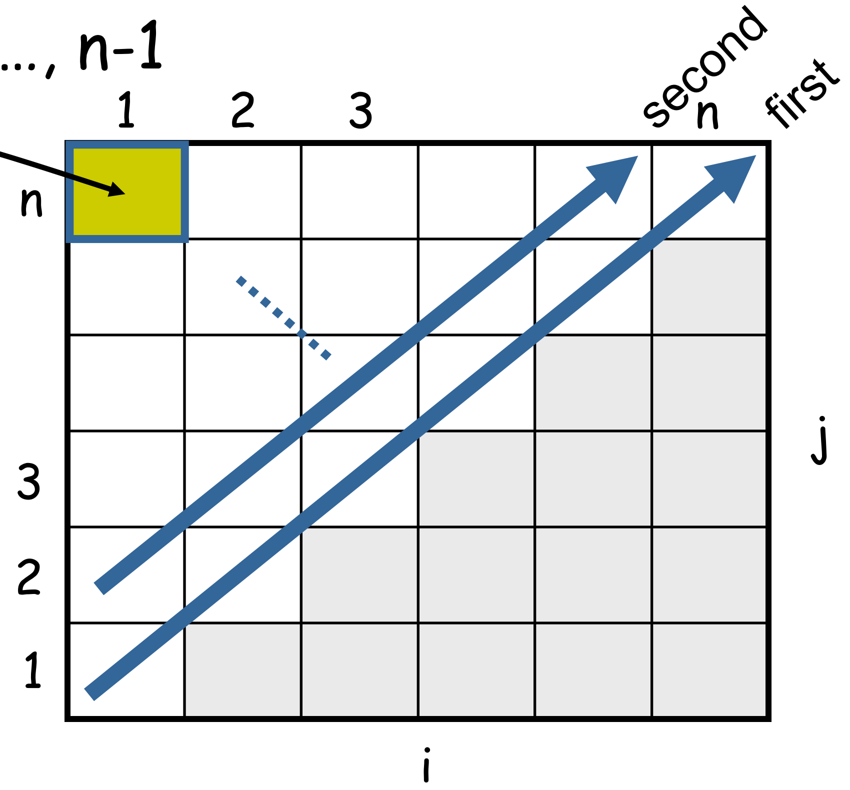
3. Computing the Optimal Costs (cont.)

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

- Length = 1: $i = j$, $i = 1, 2, \ldots, n$
- Length = 2: $j = i + 1$, $i = 1, 2, \ldots, n-1$

$m[1, n]$ gives the optimal solution to the problem

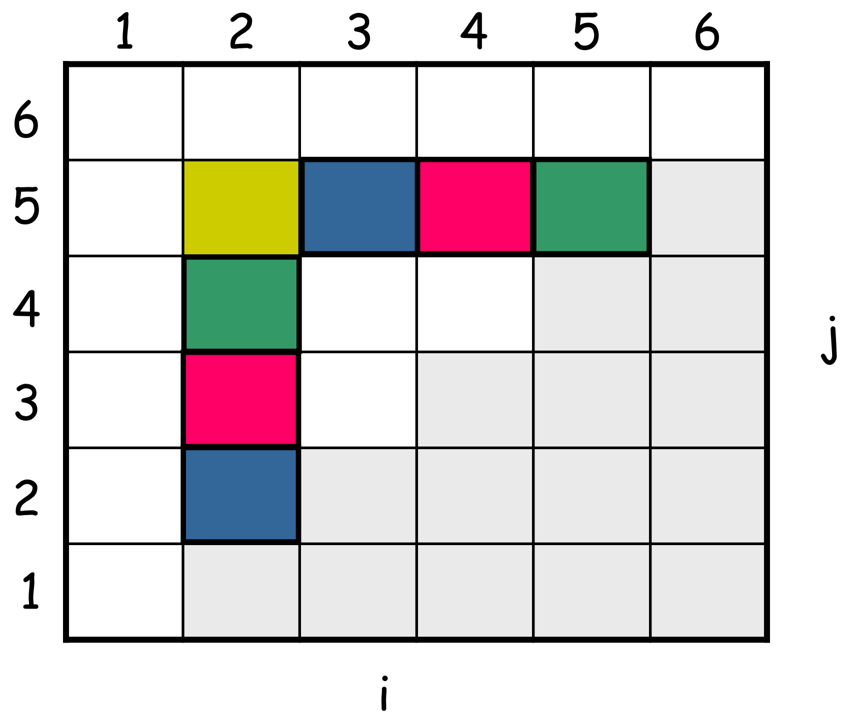
Compute rows from bottom to top and from left to right



Example:

$\min \{m[i,k] + m[k+1,j] + p_{i-1}p_kp_j\}$

$$m[2,5] = \min \begin{cases} m[2,2] + m[3,5] + p_1p_2p_5 & k = 2 \\ m[2,3] + m[4,5] + p_1p_3p_5 & k = 3 \\ m[2,4] + m[5,5] + p_1p_4p_5 & k = 4 \end{cases}$$



- Values $m[i, j]$ depend only on values that have been previously computed

Example:

min {m[i, k] + m[k+1, j] + $p_{i-1}p_kp_j$}

Compute $A_1 \cdot A_2 \cdot A_3$

- A_1: 10 x 100 (p_0 x p_1)
- A_2: 100 x 5 (p_1 x p_2)
- A_3: 5 x 50 (p_2 x p_3)

	1	2	3
3	2 7500	2 25000	0
2	1 5000	0	
1	0		

m[i, i] = 0 for i = 1, 2, 3

$m[1, 2] = m[1, 1] + m[2, 2] + p_0p_1p_2$ (A_1A_2)

$= 0 + 0 + 10 * 100 * 5 = 5{,}000$

$m[2, 3] = m[2, 2] + m[3, 3] + p_1p_2p_3$ (A_2A_3)

$= 0 + 0 + 100 * 5 * 50 = 25{,}000$

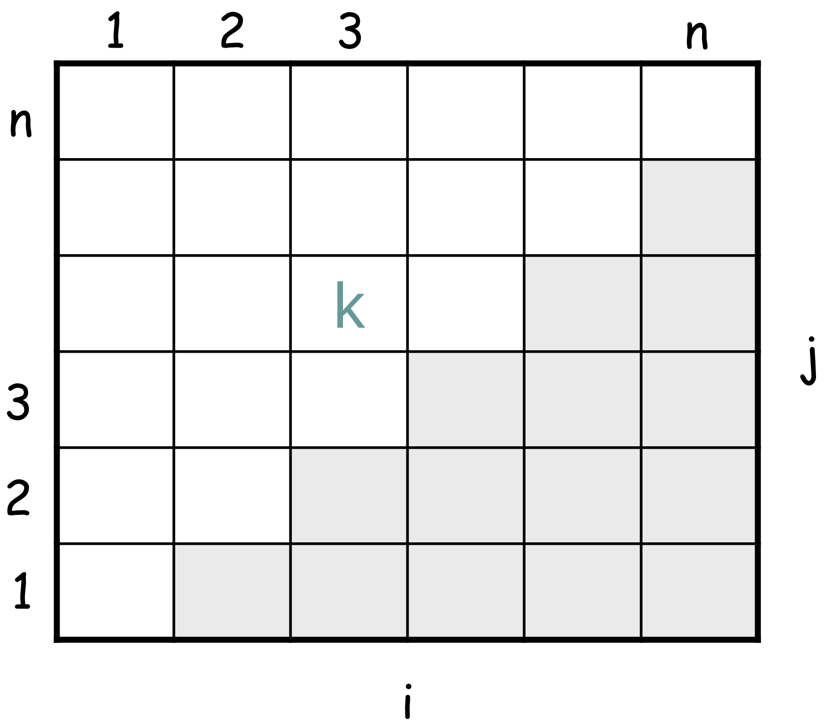
$$m[1, 3] = \min \begin{cases} m[1, 1] + m[2, 3] + p_0p_1p_3 = 75{,}000 & (A_1(A_2A_3)) \\ m[1, 2] + m[3, 3] + p_0p_2p_3 = \color{red}{7{,}500} & ((A_1A_2)A_3) \end{cases}$$

Matrix-Chain-Order(*p*)

```
n ← length[p] - 1
for i ← 1 to n                                  // initialization: O(n) time
    do m[i, i] ← 0
for L ← 2 to n                                  // L = length of sub-chain
    do for i ← 1 to n - L+1
        do j ← i + L - 1                        O(n^3)
            m[i, j] ← ∞
            for k ← i to j - 1
                do q ← m[i, k] + m[k+1, j] + p_{i-1} p_k p_j
                    if q < m[i, j]
                        then m[i, j] ← q
                             s[i, j] ← k
return m and s
```

4. Construct the Optimal Solution

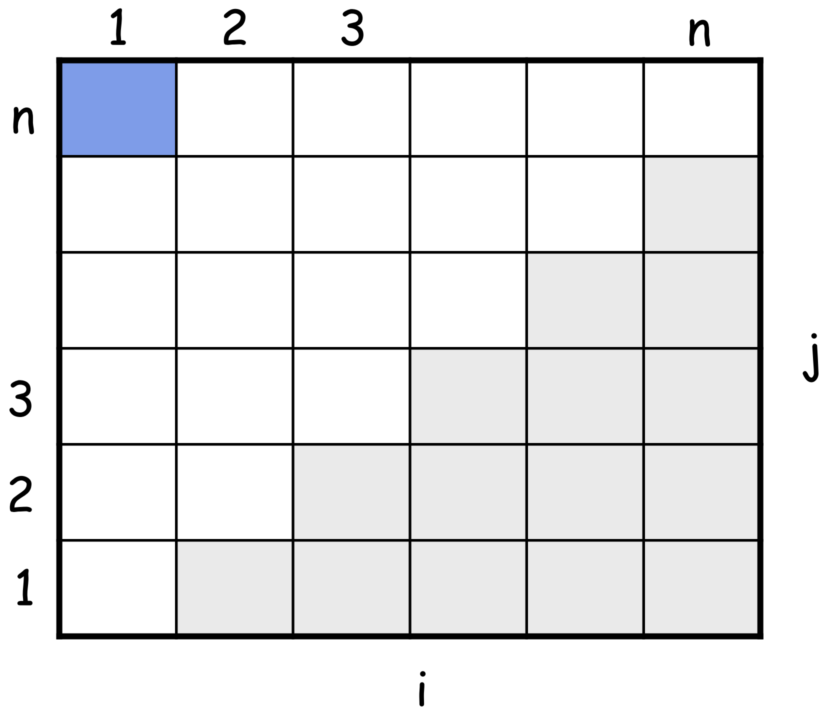
- In a similar matrix s we keep the optimal values of k
- $s[i, j]$ = a value of k such that an optimal parenthesization of $A_{i..j}$ splits the product between A_k and A_{k+1}



4. Construct the Optimal Solution

- $s[1, n]$ is associated with the entire product $A_{1..n}$
  - The final matrix multiplication will be split at k = $s[1, n]$

    $A_{1..n} = A_{1..s[1, n]} \cdot A_{s[1, n]+1..n}$
  - For each subproduct recursively find the corresponding value of k that results in an optimal parenthesization



4. Construct the Optimal Solution

- $s[i, j]$ = value of k such that the optimal parenthesization of $A_i\ A_{i+1} \cdots A_j$ splits the product between A_k and A_{k+1}

	1	2	3	4	5	6
6	3	3	3	5	5	-
5	3	3	3	4	-	
4	3	3	3	-		
3	1	2	-			
2	1	-				
1	-					

j

i

- $s[1, n] = 3 \Rightarrow A_{1..6} = A_{1..3}\ A_{4..6}$
- $s[1, 3] = 1 \Rightarrow A_{1..3} = A_{1..1}\ A_{2..3}$
- $s[4, 6] = 5 \Rightarrow A_{4..6} = A_{4..5}\ A_{6..6}$

4. Construct the Optimal Solution (cont.)

PRINT-OPT-PARENS(s, i, j)

if i = j

   then print "A"$_i$

   else print "("

        PRINT-OPT-PARENS(s, i, s[i, j])

        PRINT-OPT-PARENS(s, s[i, j] + 1, j)

        print ")"

	1	2	3	4	5	6
6	3	3	3	5	5	-
5	3	3	3	4	-	
4	3	3	3	-		
3	1	2	-			
2	1	-				
1	-					

j

i

Eg: $A_1 \cdot\cdot\cdot A_6$ $((A_1(A_2A_3))((A_4A_5)A_6))$

PRINT-OPT-PARENS(*s*, i, j)

if i = j

then print "A"$_i$

else print "("

PRINT-OPT-PARENS(*s*, i, *s*[i, j])

PRINT-OPT-PARENS(*s*, *s*[i, j] + 1, j)

print ")"

s[1..6, 1..6]

j \ i	1	2	3	4	5	6
6	3	3	3	5	5	-
5	3	3	3	4	-	
4	3	3	3	-		
3	1	2	-			
2	1	-				
1	-					

P-O-P(s, 1, 6) s[1, 6] = 3

i = 1, j = 6 "(" P-O-P (s, 1, 3) s[1, 3] = 1

i = 1, j = 3 "(" P-O-P(s, 1, 1) ⇒ "A_1"

P-O-P(s, 2, 3) s[2, 3] = 2

i = 2, j = 3 "(" P-O-P (s, 2, 2) ⇒ "A_2"

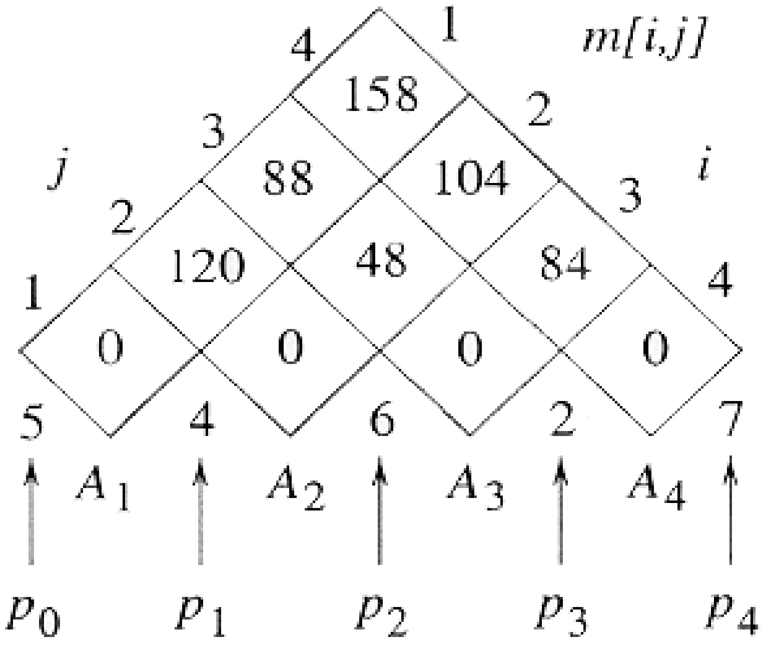
P-O-P (s, 3, 3) ⇒ "A_3"

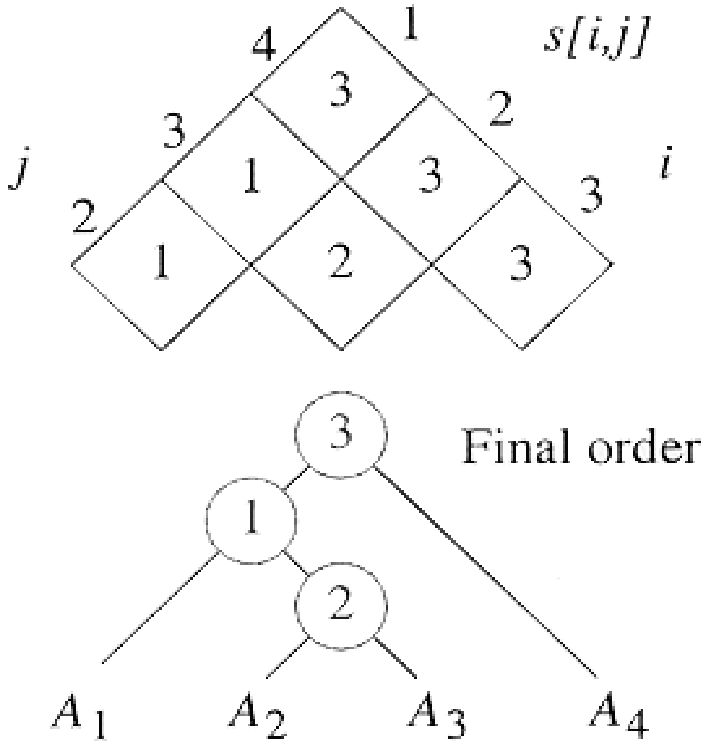
")"

")" …

Example for Practice

- The initial set of dimensions are <5, 4, 6, 2, 7>: we are multiplying A_1 (5x4) times A_2 (4x6) times A_3 (6x2) times A_4 (2x7). Optimal sequence is $(A_1\,(A_2A_3))\,A_4$.





Acknowledgements

- Cormen, T.H., Leiserson, C.E., Rivest, R.L. and Stein, C., Introduction to algorithms. MIT press, 2009
- Dr. David Kauchak, Pomona College
- Prof. David Plaisted, The University of North Carolina at Chapel Hill