

Analysis of Algorithms

Dynamic Programming

Dynamic Programming

General Strategy:

- ★ Solve recursively the problem – **top-down** – based on solutions for **sub-problems**.
- ★ Find a **bottom-up** order to compute all the **sub-problems**.

Time complexity:

- ★ **If** there are **polynomial** number of sub-problems.
- ★ **If** each sub-problem can be computed in **polynomial** time.
- ★ **Then** the solution can be found in **polynomial** time.

Remark: Greedy also applies a **top-down** strategy but usually on one sub-problem so the order of computation is clear.

Fibonacci Numbers

The sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

Recursive Definition:

$$F_k = \begin{cases} 0 & \text{for } k = 0 \\ 1 & \text{for } k = 1 \\ F_{k-1} + F_{k-2} & \text{for } k \geq 2 \end{cases}$$

Fibonacci Numbers — Some Facts

- ★ $F_k = (\phi^k - \hat{\phi}^k) / \sqrt{5}$.
 - $\phi = (1 + \sqrt{5}) / 2 \approx 1.618$ (the **golden ratio** number).
 - $\hat{\phi} = (1 - \sqrt{5}) / 2$.
 - ϕ and $\hat{\phi}$ are the roots of $x^2 = x + 1$.
- ★ $|\hat{\phi}| < 1$ implies that $F_k \approx \phi^k / \sqrt{5}$.
- ★ $k \approx \log_{\phi} F_k = \Theta(\log F_k)$.

Top-Down Recursive Computation

Fibo(k)

if $k = 0$ then return(0)

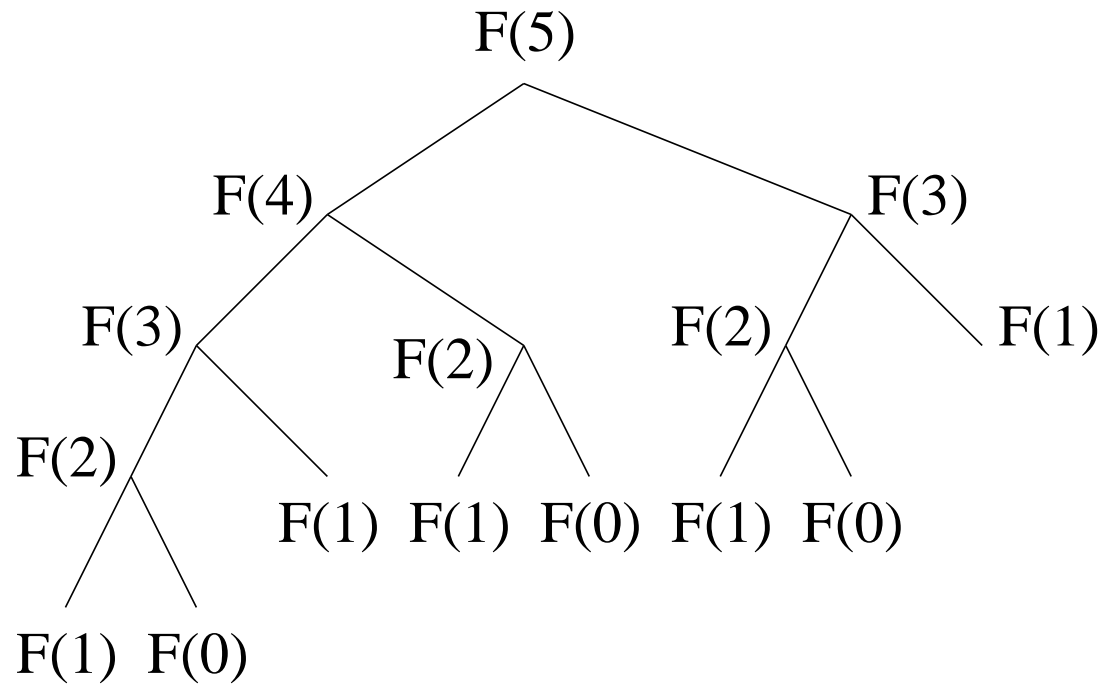
if $k = 1$ then return(1)

return(Fibo($k - 1$)+Fibo($k - 2$))

Memory complexity: $\Theta(k)$.

★ Because k is the maximum **depth** of the recursion.

The Recursion Tree



Time Complexity I

Notation: $T(k)$

- ★ Number of calls to **Fibo** excluding **Fibo(0)** and **Fibo(1)**.
- ★ Number of **internal** nodes in the recursion tree.

Initial values: $T(1) = T(0) = 0$.

Recursive formula: $T(k) = 1 + T(k - 1) + T(k - 2)$.

The sequence: 0, 0, 1, 2, 4, 7, 12, 20, ...

Time Complexity I

Guess: $T(k) = F_{k+1} - 1$ for $k \geq 0$.

Why? The recursive formula for $T(k)$ is very **similar** to the formula for F_k .

Proof by induction:

$$T(0) = F_1 - 1 = 0$$

$$T(1) = F_2 - 1 = 0$$

$$\begin{aligned} T(k) &= 1 + T(k-1) + T(k-2) \\ &= 1 + (F_k - 1) + (F_{k-1} - 1) \\ &= F_{k+1} - 1. \end{aligned}$$

Time Complexity II

Notation: $T(k)$

- ★ Number of calls to **Fibo** including **Fibo(0)** and **Fibo(1)**.
- ★ Number of **all** nodes in the recursion tree.

Initial values: $T(1) = T(0) = 1$.

Recursive formula: $T(k) = 1 + T(k - 1) + T(k - 2)$.

The sequence: 1, 1, 3, 5, 9, 15, 25, 41, ...

Time Complexity II

Solution: $T(k) = 2F_{k+1} - 1$ for $k \geq 0$.

Why? The number of leaves in a binary tree is one more than the number of internal nodes.

Proof by induction:

$$T(0) = 2F_1 - 1 = 1$$

$$T(1) = 2F_2 - 1 = 1$$

$$\begin{aligned} T(k) &= 1 + T(k-1) + T(k-2) \\ &= 1 + (2F_k - 1) + (2F_{k-1} - 1) \\ &= 2F_{k+1} - 1. \end{aligned}$$

Bottom-Up Dynamic Programming

Fibo(k)

$$F(0) = 0$$

$$F(1) = 1$$

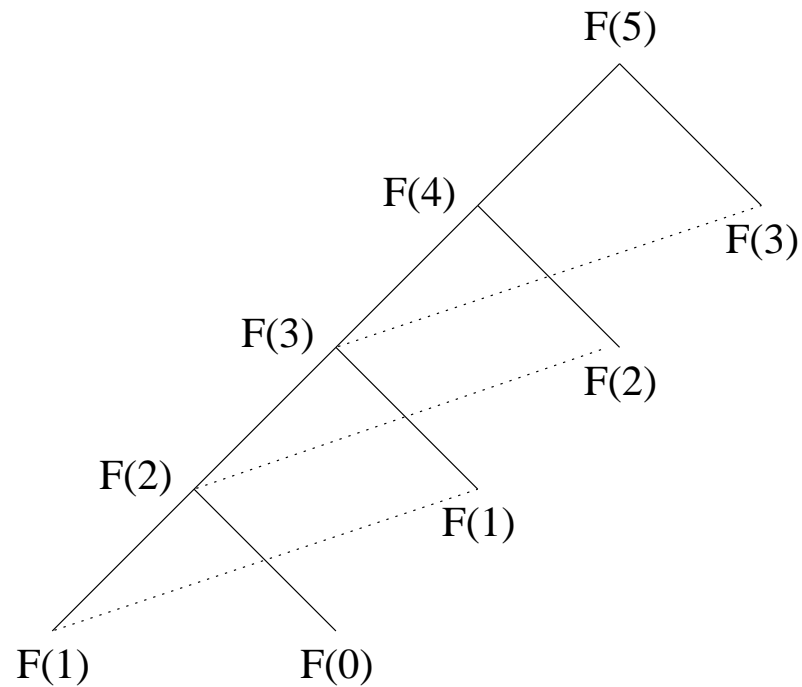
for $i = 2$ to k do

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

return($F(k)$)

Time and Memory Complexity: $\Theta(k) = \Theta(\log F_k)$.

Bottom-Up Dynamic Programming



Saving Memory

Fibo(k)

$x = 0$

$y = 1$

for $i = 2$ **to** k **do**

$z = x + y$

$x = y$

$y = z$

return(z)

Time Complexity: $\Theta(k) = \Theta(\log F_k)$.

Memory Complexity: $\Theta(1)$.

Top-Down Dynamic Programming

$\text{Fibo}(k)$

$\text{Fibo}(0) = 0$

$\text{Fibo}(1) = 1$

for $i = 2$ to k do

$\text{Fibo}(i) = \infty$

$F(k)$ (* the recursion *)

$F(k)$

if $\text{Fibo}(k) \neq \infty$

then return($\text{Fibo}(k)$)

else return($F(k - 1) + F(k - 2)$)

Top-Down Dynamic Programming

Time and Memory Complexity: $\Theta(k) = \Theta(\log F_k)$.

Advantage: It is easier to **imitate** the recursive solution.

Disadvantage: The **same** complexity, but **less efficient** with both time and memory.

Binomial Coefficients

- ★ $C(n, k) = \binom{n}{k}$:
- ★ The number of size k subsets from the set $\{1, 2, \dots, n\}$.
- ★ The coefficient of x^k in the expansion of $(1 + x)^n$.

Recursive Definition:

$$C(n, k) = \binom{n}{k} = \begin{cases} 1 & \text{for } k = 0 \\ 1 & \text{for } k = n \\ \binom{n-1}{k-1} + \binom{n-1}{k} & \text{for } 0 < k < n \\ 0 & \text{otherwise} \end{cases}$$

The Pascal Triangle

				1								
				1		1						
			1		2		1					
		1		3		3		1				
	1		4		6		4		1			
	1		5		10		10		5		1	
1		6		15		20		15		6		1

Binomial Coefficients – Some Facts

$$\star \binom{n}{k} = \frac{n!}{k!(n-k)!} = \frac{n(n-1)\cdots(n-k+1)}{k(k-1)\cdots 1}.$$

$$\star \binom{n}{k} = \binom{n}{n-k}.$$

$$\star \left(\frac{n}{k}\right)^k \leq \frac{n}{k} \cdot \frac{n-1}{k-1} \cdots \frac{n-k+1}{1} \leq (n-k+1)^k$$

$$\star \binom{n}{k} = \Theta(n^k) \text{ for a fix } k.$$

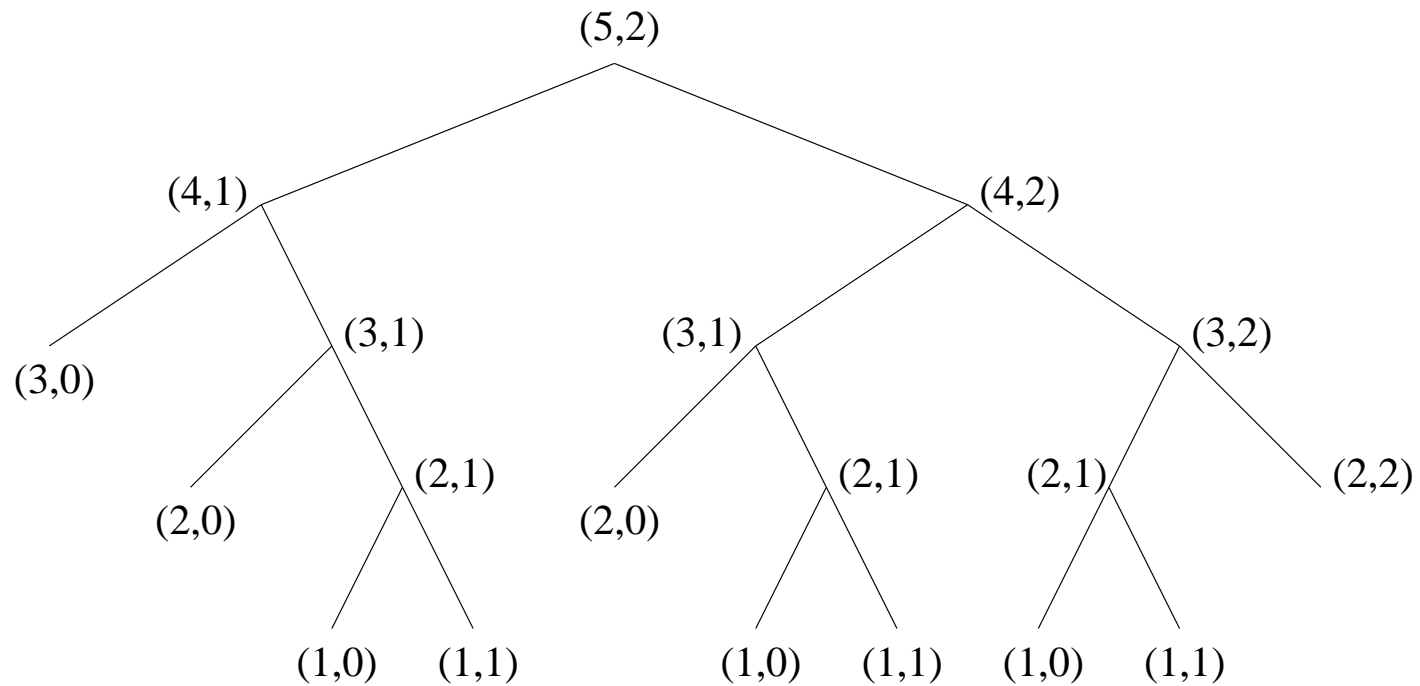
Top-Down Recursive Computation

```
 $C(n, k)$     (* integers  $0 \leq k \leq n$  *)  
  if  $k = 0$  then return(1)  
  if  $k = n$  then return(1)  
  return( $C(n - 1, k - 1) + C(n - 1, k)$ )
```

Memory complexity: $\Theta(n)$.

- ★ Because there exists a path of length $n - 1$ in the recursion tree (RR...RLL...L).

The Recursion Tree



Time Complexity I

Notation: $T(n, k)$

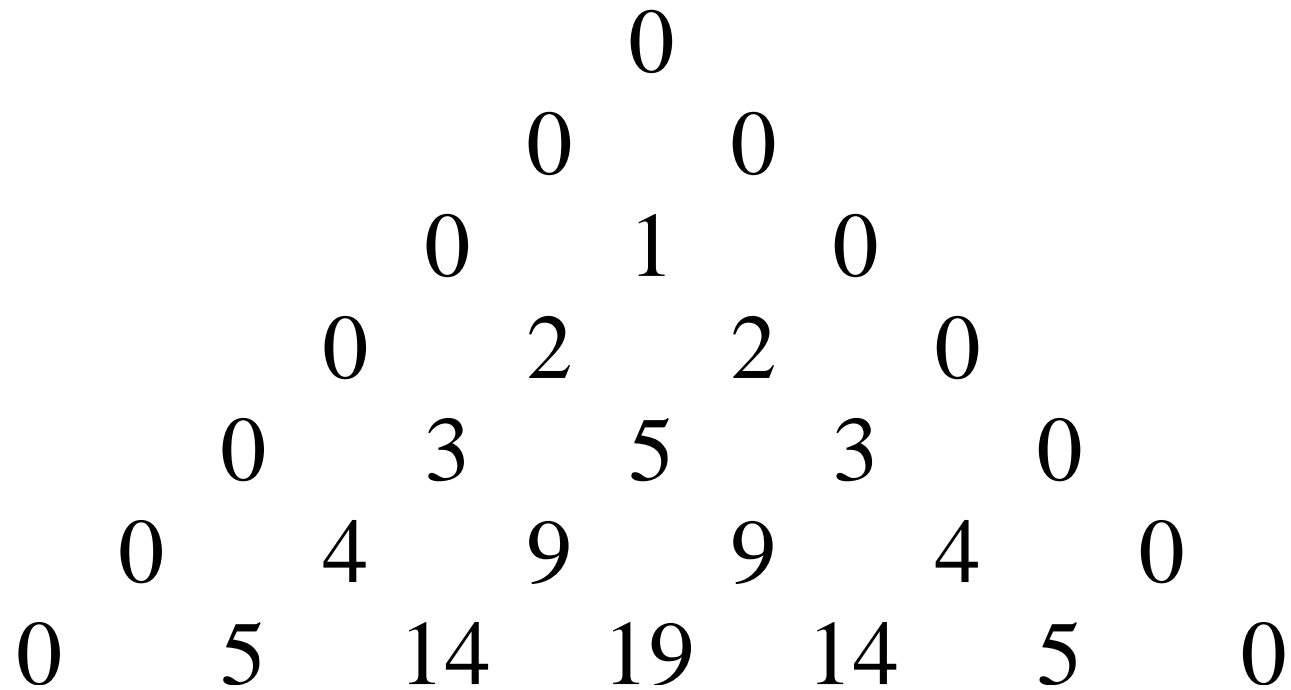
- ★ Number of calls to $C(i, j)$ **excluding** $C(i, i)$ and $C(i, 0)$.
- ★ Number of **internal** nodes in the recursion tree.

Initial values: $T(i, i) = T(i, 0) = 0$ for $0 \leq i \leq n$.

Recursive formula:

$$T(n, k) = 1 + T(n - 1, k - 1) + T(n - 1, k).$$

Time Complexity I - the Triangle



Time Complexity I

Guess: $T(n, k) = \binom{n}{k} - 1$.

Why? The recursive formula for $T(n, k)$ is very **similar** to the formula for $\binom{n}{k}$.

Proof by induction:

$$T(i, i) = \binom{i}{i} - 1 = 0$$

$$T(i, 0) = \binom{i}{0} - 1 = 0$$

$$\begin{aligned} T(n, k) &= 1 + T(n-1, k-1) + T(n-1, k) \\ &= 1 + \left(\binom{n-1}{k-1} - 1 \right) + \left(\binom{n-1}{k} - 1 \right) \\ &= \binom{n}{k} - 1. \end{aligned}$$

Time Complexity II

Notation: $T(n, k)$

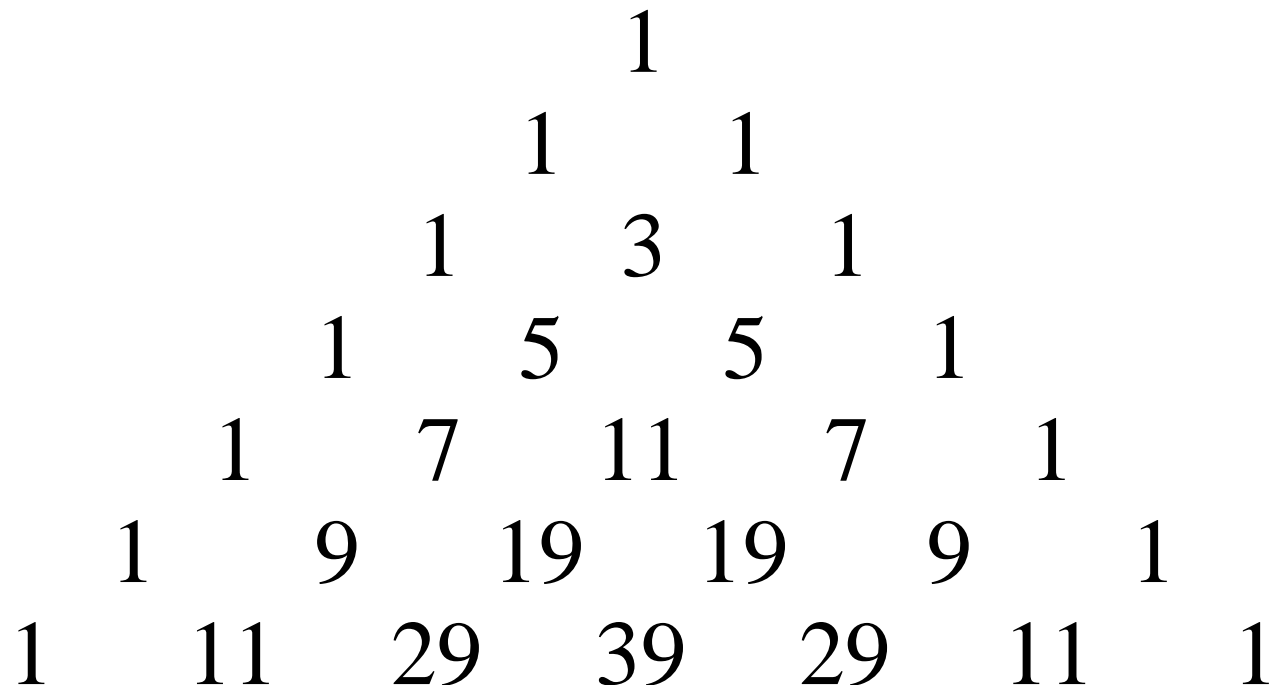
- ★ Number of calls to $C(i, j)$ **including** $C(i, i)$ and $C(i, 0)$.
- ★ Number of **all** nodes in the recursion tree.

Initial values: $T(i, i) = T(i, 0) = 1$ for $0 \leq i \leq n$.

Recursive formula:

$$T(n, k) = 1 + T(n - 1, k - 1) + T(n - 1, k).$$

Time Complexity II – the Triangle



Time Complexity II

Solution: $T(n, k) = 2\binom{n}{k} - 1$.

Why? The number of leaves in a binary tree is one more than the number of internal nodes.

Proof by induction:

$$T(i, i) = 2\binom{i}{i} - 1 = 1$$

$$T(i, 0) = 2\binom{i}{0} - 1 = 1$$

$$\begin{aligned} T(n, k) &= 1 + T(n-1, k-1) + T(n-1, k) \\ &= 1 + \left(2\binom{n-1}{k-1} - 1\right) + \left(2\binom{n-1}{k} - 1\right) \\ &= 2\binom{n}{k} - 1. \end{aligned}$$

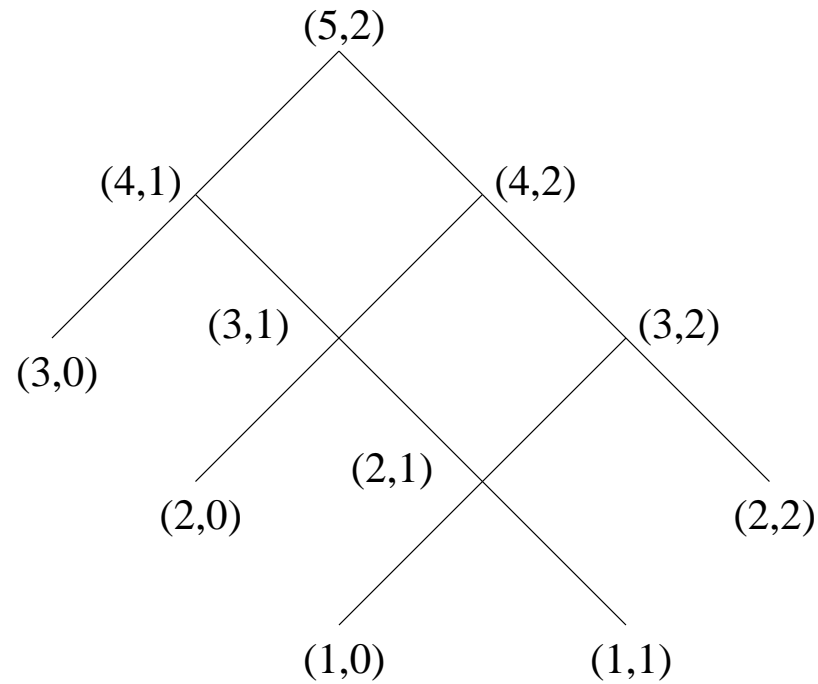
Bottom-Up Dynamic Programming

```
 $C(n, k)$  (* integers  $0 \leq k \leq n$  *)  
for  $i = 0$  to  $n$  do  
     $c(i, i) = 1$   
     $c(i, 0) = 1$   
for  $i = 2$  to  $n$  do  
    for  $j = 1$  to  $\min\{i - 1, k\}$  do  
         $c(i, j) = c(i - 1, j - 1) + c(i - 1, j)$   
return( $c(n, k)$ )
```

Time and Memory Complexity: $\Theta(kn)$.

★ Possible with only $\Theta(n)$ memory.

Bottom-Up Dynamic Programming



Top-Down Dynamic Programming

```
 $C(n, k)$     (* integers  $0 \leq k \leq n$  *)  
  for  $i = 0$  to  $n$  do  
     $c(i, i) = 1$   
     $c(i, 0) = 1$   
  for  $i = 2$  to  $n$  do  
    for  $j = 1$  to  $\min\{i - 1, k\}$  do  
       $c(i, j) = \infty$   
  return( $CC(n, k)$ )    (* the recursion *)
```

Top-Down Dynamic Programming

$CC(n, k)$

if $c(n, k) = \infty$

then $c(n, k) = CC(n - 1, k - 1) + CC(n - 1, k)$

else return($c(n, k)$)

Time and Memory Complexity: $\Theta(kn)$.

The 0-1 Knapsack Problem

Input:

- ★ n items I_1, \dots, I_n .
- ★ The **value** of item I_i is v_i and its **weight** is w_i .
- ★ A maximum weight W .

Output:

- ★ A set of items whose total weight is at most W and whose total value is **maximum**.
- ★ The set either **includes** all of item I_i or not.

Assumption: All the weights and W are positive integers.

Dynamic Programming

- ★ **Sub-problems:** $v(i, w)$ for $0 \leq i \leq n$ and $0 \leq w \leq W$.
- ★ $v(i, w)$ is the maximum value for items $\{I_1, \dots, I_i\}$ and maximum weight w .
- ★ The final **solution** is $v(n, W)$.

Dynamic Programming

Initial values:

- ★ $v(i, 0) = 0$ for $0 \leq i \leq n$.
- ★ $v(0, w) = 0$ for $0 \leq w \leq W$.

Recursive formula:

- ★ $v(i, w) = v(i - 1, w)$
 - if $w_i > w$.
- ★ $v(i, w) = v(i - 1, w)$
 - if $v(i - 1, w) \geq v_i + v(i - 1, w - w_i)$.
- ★ $v(i, w) = v_i + v(i - 1, w - w_i)$
 - if $v(i - 1, w) < v_i + v(i - 1, w - w_i)$.

A Bottom-Up Implementation

```
Knapsack( $W, (v_1, w_1), \dots, (v_n, w_n)$ )  
  for  $i = 0$  to  $n$  do  
     $v(i, 0) = 0$   
  for  $w = 0$  to  $W$  do  
     $v(0, w) = 0$   
  for  $i = 1$  to  $n$  do  
    for  $w = 1$  to  $W$  do  
      if  $w \geq w_i$  and  $v_i + v(i - 1, w - w_i) > v(i - 1, w)$   
      then  $v(i, w) = v_i + v(i - 1, w - w_i)$   
      else  $v(i, w) = v(i - 1, w)$   
  return( $v(n, W)$ )
```

A Bottom-Up Implementation

Time and memory complexity: $\Theta(nW)$.

Remark: This is a **weakly** polynomial algorithm since the complexity depends on the **value** of the input (W) and not only on the **size** of the input (n).

Finding the Optimal Set of Items

Definition: $B(i, w)$ is **true** iff item I_i is included in the optimal set for the sub-problem $v(i, w)$.

Implementation:

⋮

if $w \geq w_i$ **and** $v_i + v(i - 1, w - w_i) > v(i - 1, w)$

then $v(i, w) = v_i + v(i - 1, w - w_i)$ **and** $B(i, w) = \text{true}$

else $v(i, w) = v(i - 1, w)$ **and** $B(i, w) = \text{false}$

⋮

Complexity: The same as for computing $v(i, w)$.

Finding the Optimal Set of Items

Output: S The optimal set of items.

Find- $S(B(\cdot, \cdot), W, w_1, \dots, w_n)$

$S = \emptyset$

$w = W$

for $i = n$ **downto** 1 **do**

if $B(i, w)$ **is true then**

$S = S \cup \{I_i\}$

$w = w - w_i$

return(S)

Time complexity: $\Theta(n)$.

A Top-Down Recursive Implementation

$\text{Rec-Knapsack}(i, w)$

if $i = 0$ then return(0)

if $w = 0$ then return(0)

otherwise

if $w < w_i$ then return($\text{Rec-Knapsack}(i - 1, w)$)

else

$x = \text{Rec-Knapsack}(i - 1, w)$

$y = v_i + \text{Rec-Knapsack}(i - 1, w - w_i)$

return($\max \{x, y\}$)

A Top-Down Recursive Implementation

Initial call: $\text{Rec-Knapsack}(n, W)$.

Time complexity: Could be **exponential** in n .

Memory complexity: $\Theta(n)$ because the depth of the recursion is at most n .

Multiplying n Numbers

Objective: Find $C(n)$, the number of **ways** to compute

$$x_1 \cdot x_2 \cdot \dots \cdot x_n.$$

n	multiplication order
2	$(x_1 \cdot x_2)$
3	$(x_1 \cdot (x_2 \cdot x_3))$
	$((x_1 \cdot x_2) \cdot x_3)$
4	$(x_1 \cdot (x_2 \cdot (x_3 \cdot x_4)))$
	$(x_1 \cdot ((x_2 \cdot x_3) \cdot x_4))$
	$((x_1 \cdot x_2) \cdot (x_3 \cdot x_4))$
	$((x_1 \cdot (x_2 \cdot x_3)) \cdot x_4)$
	$((x_1 \cdot x_2) \cdot x_3) \cdot x_4$

Multiplying n Numbers – Small n

n	$C(n)$
1	1
2	1
3	2
4	5
5	14
6	42
7	132

Multiplying n Numbers – Small n

Recursive equation: Where is the **last** multiplication?

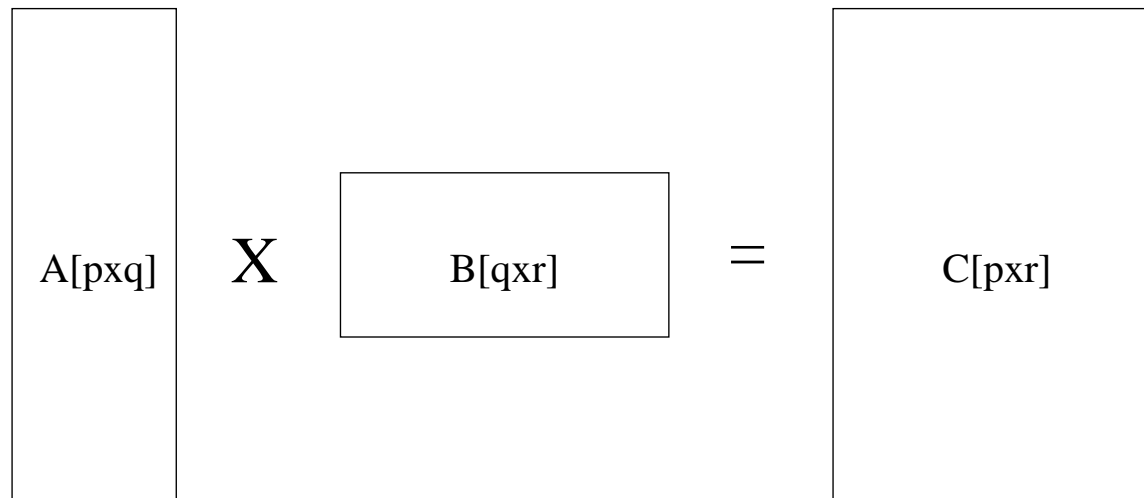
$$C(n) = \sum_{k=1}^{n-1} C(k) \cdot C(n - k)$$

Catalan numbers: $C(n) = \frac{1}{n} \binom{2n-2}{n-1}$.

Asymptotic value: $C(n) \approx \frac{4^n}{n^{3/2}}$.

★ $\frac{C(n)}{C(n-1)} \rightarrow 4$ for $n \rightarrow \infty$.

Multiplying Two Matrices



Objective: Compute $C[p \times r] = A[p \times q] \cdot B[q \times r]$.

Validity condition: $\text{Columns}(A) = \text{Rows}(B)$.

Multiplying Two Matrices

Matrix-Multiply($A[p \times q], B[q \times r]$)

for $i = 1$ to p

for $j = 1$ to r

$C[i, j] = 0$

for $k = 1$ to q

$C[i, j] = C[i, j] + A[i, k] \cdot B[k, j]$

return($C[p \times r]$)

Complexity: Number of multiplications is equal to number of additions: $pqr \Rightarrow$ Total number of operation is $\Theta(pqr)$.

Multiplying n Matrices

Objectives:

- ★ Compute **efficiently**

$$A[p_0 \times p_n] = A_1[p_0 \times p_1] \cdot A_2[p_1 \times p_2] \cdot \dots \cdot A_n[p_{n-1} \times p_n]$$

- ★ Find $m(A)$ - the minimal number of **scalar** multiplications needed to compute A .

Claim: The order **matters!**

Problem: It is **impossible** to check **all** the $\approx \frac{4^n}{n^{3/2}}$ possibilities.

Example

$$\star A[10 \times 50] = A_1[10 \times 100] \cdot A_2[100 \times 5] \cdot A_3[5 \times 50]$$

$$\star A = ((A_1 \cdot A_2) \cdot A_3) \Rightarrow$$

$$\begin{aligned} m(A) &= 5000 + m(A_{12}[10 \times 5] \cdot A_3[5 \times 50]) \\ &= 5000 + 2500 = 7500. \end{aligned}$$

$$\star A = (A_1 \cdot (A_2 \cdot A_3)) \Rightarrow$$

$$\begin{aligned} m(A) &= 25000 + m(A_1[10 \times 100] \cdot A_{23}[100 \times 50]) \\ &= 25000 + 50000 = 75000. \end{aligned}$$

The Recursive Solution

Notation:

★ $A_{i,j}[p_{i-1} \times p_j] = A_i[p_{i-1} \times p_i] \cdots A_j[p_{j-1} \times p_j]$.

★ $m[i, j]$ - minimal number of operations to compute $A_{i,j}$.

Initial values: $A_{i,i} = A_i$ and $m[i, i] = 0$.

Final solution: $A_{1,n}$ and $m[1, n]$.

The Recursive Solution

Key observation:

- ★ If the **last** multiplication is

$A_{i,j}[p_{i-1} \times p_j] = A_{i,k}[p_{i-1} \times p_k] \cdot A_{k+1,j}[p_k \times p_j]$
then $A_{i,k}$ and $A_{k+1,j}$ **should** be computed optimally.

- ★ The cost of this **last** multiplication is

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

The recursive formula:

$$m[i, j] = \min_{i \leq k < j} \{m[i, k] + m[k + 1, j] + p_{i-1}p_kp_j\}.$$

A Bottom-Up Computation

- ★ There are $n + \binom{n}{2} = \Theta(n^2)$ **sub-problems**.
- ★ The computation **order** is by the difference $j - i$.
- ★ $S[i, j]$ - the **last** multiplication in computing $m[i, j]$.

Input: The matrices **dimensions** $\mathcal{P} = \langle p_0, p_1, \dots, p_n \rangle$.

Complexity: $\Theta(n)$ complexity per each sub-problem
 $\Rightarrow \Theta(n^3)$ overall complexity.

A Bottom-Up Computation

Matrix-Chain-Order(\mathcal{P})

for $i = 1$ to n do $m[i, i] = 0$

for $d = 1$ to $n - 1$ do

for $i = 1$ to $n - d$ do

$j = i + d$

$m[i, j] = \infty$

for $k = i$ to $j - 1$ do

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

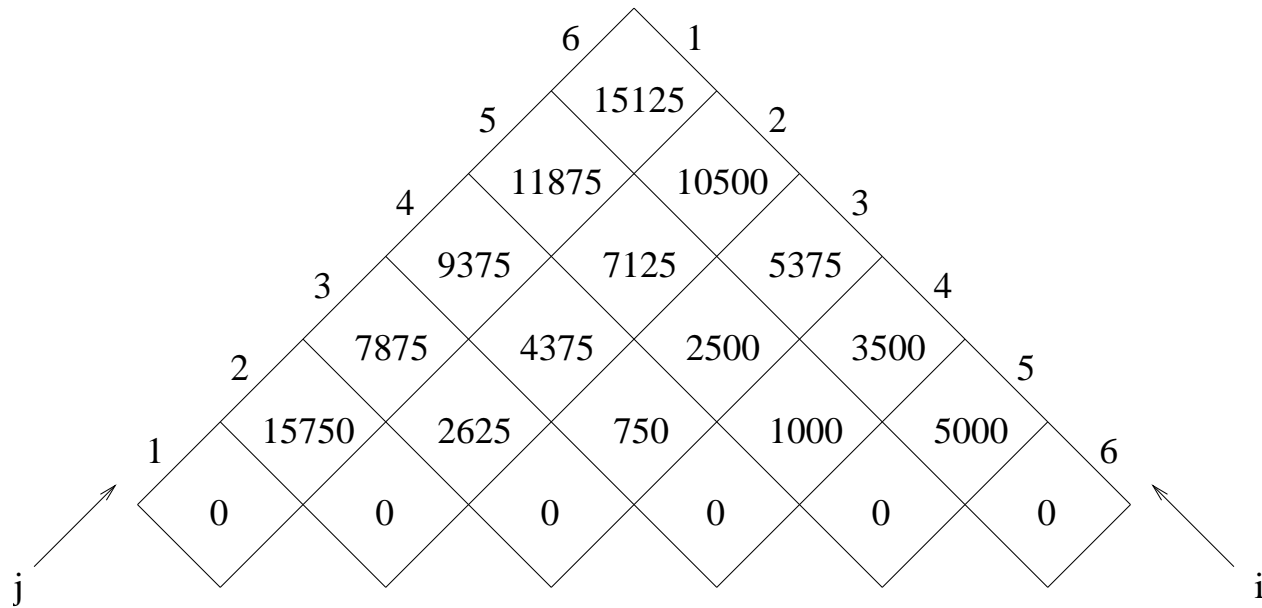
if $q < m[i, j]$ then

$m[i, j] = q$

$S[i, j] = k$

return(m, S)

Example



$$\mathcal{P} = \langle p_0, p_1, p_2, p_3, p_4, p_5, p_6 \rangle = \langle 30, 35, 15, 5, 10, 20, 25 \rangle$$

Example

$$m[1, 3] = \min$$

$$m[1, 1] + m[2, 3] + p_0 p_1 p_3 = 0 + 2625 + 5250 = 7875$$

$$m[1, 2] + m[3, 3] + p_0 p_2 p_3 = 15750 + 0 + 2250 = 18000$$

$$\Rightarrow S[1, 3] = 1$$

$$m[2, 5] = \min$$

$$m[2, 2] + m[3, 5] + p_1 p_2 p_5 = 0 + 2500 + 10500 = 13000$$

$$m[2, 3] + m[4, 5] + p_1 p_3 p_5 = 2625 + 1000 + 3500 = 7125$$

$$m[2, 4] + m[5, 5] + p_1 p_4 p_5 = 4375 + 0 + 7000 = 11375$$

$$\Rightarrow S[2, 5] = 3$$

How to Multiply?

- ★ How to **compute** $A_{1,n}$?
- ★ Compute all the $A_{i,j}$ **needed** to compute $A_{1,n}$.

Matrix-Chain-Multiply(i, j)

if $j > i$ **then**

$k = S[i, j]$

$X = \text{Matrix-Chain-Multiply}(i, k)$

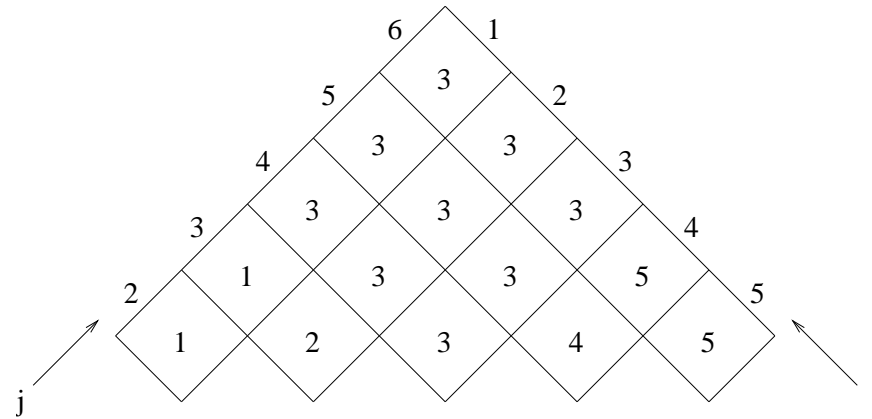
$Y = \text{Matrix-Chain-Multiply}(k + 1, j)$

return(**Matrix-Multiply**(X, Y))

else return(A_i)

Complexity: $m[i, j]$ operations.

Example



$$\mathcal{P} = \langle 30, 35, 15, 5, 10, 20, 25 \rangle$$

$$\begin{aligned} \text{Initially} &\Rightarrow (A_1 \cdot A_2 \cdot A_3 \cdot A_4 \cdot A_5 \cdot A_6) \\ S[1, 6] = 3 &\Rightarrow ((A_1 \cdot A_2 \cdot A_3) \cdot (A_4 \cdot A_5 \cdot A_6)) \\ S[1, 3] = 1 &\Rightarrow ((A_1 \cdot (A_2 \cdot A_3)) \cdot (A_4 \cdot A_5 \cdot A_6)) \\ S[4, 6] = 5 &\Rightarrow ((A_1 \cdot (A_2 \cdot A_3)) \cdot ((A_4 \cdot A_5) \cdot A_6)) \end{aligned}$$

The Recursive Top-Down Computation

Recursive-Matrix-Chain(i, j)

if $i < j$ then

$m = \infty$

for $k = i$ to $j - 1$ do

$q = \text{R-M-C}(i, k) + \text{R-M-C}(k + 1, j) + p_{i-1}p_kp_j$

if $q < m$ then $m = q$

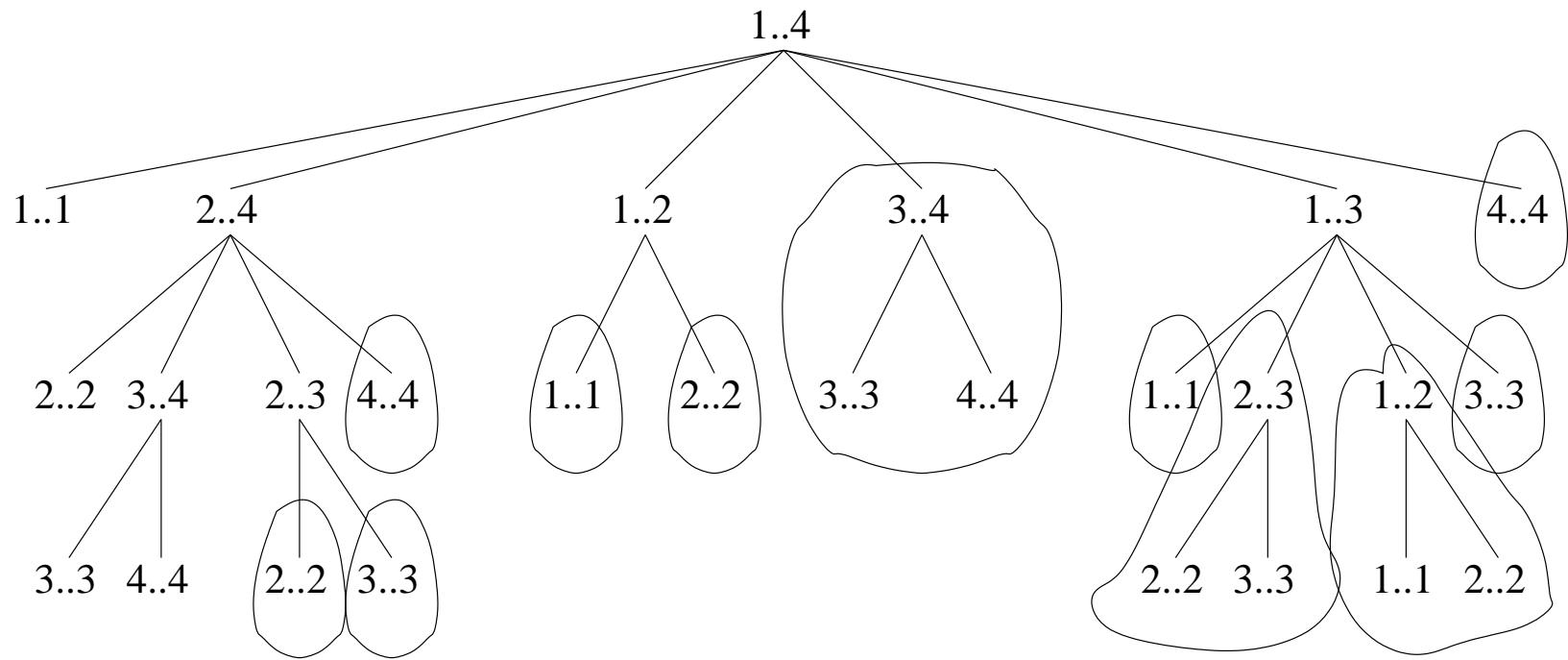
return(m)

else return(0)

Initial call: Recursive-Matrix-Chain($1, n$)

Complexity: Check all possibilities $\Rightarrow \Omega\left(\frac{4^n}{n^{3/2}}\right)$.

Where to Save?



A Top-Down Dynamic Programming

Initial values and first call:

Memoized-Matrix-Chain(\mathcal{P})

for $i = 1$ to n do

$m[i, i] = 0$

 for $j = i + 1$ to n do $m[i, j] = \infty$

return(LookUp-Chain($1, n$))

A Top-Down Dynamic Programming

The recursive procedure:

LookUp-Chain(i, j)

if $m[i, j] = \infty$ then

for $k = i$ to $j - 1$ do

$q = \text{L-C}(i, k) + \text{L-C}(k + 1, j) + p_{i-1}p_kp_j$

if $q < m[i, j]$ then $m[i, j] = q$

return($m[i, j]$)

A Top-Down Dynamic Programming – Complexity

- ★ Number of recursive calls is $\Theta(n^2)$.
- ★ Each call takes $\Theta(n)$ time.
- ★ All together, $\Theta(n^3)$ time.