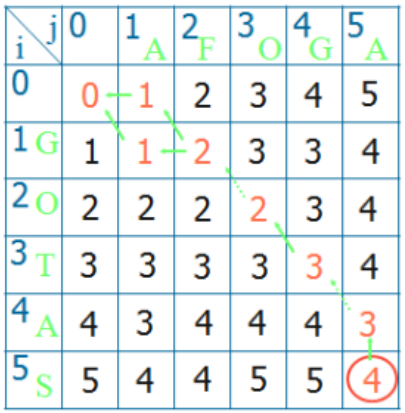
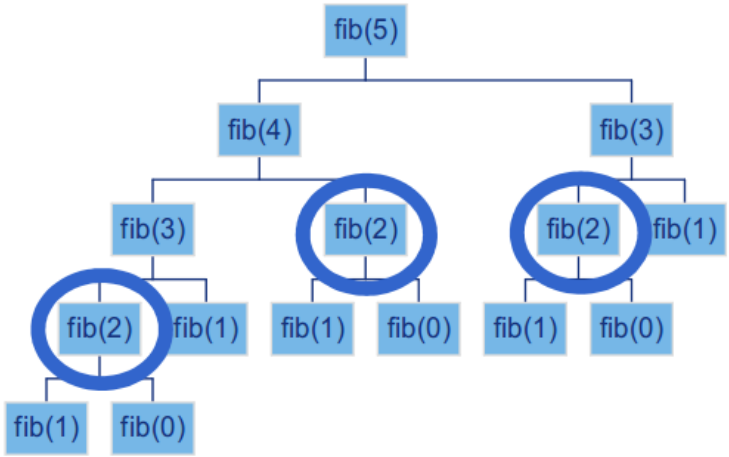
Dynamic Programming

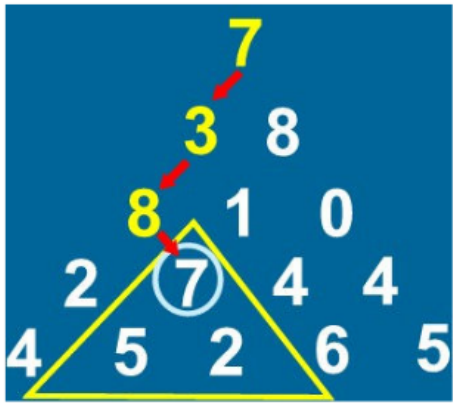
Pedro Ribeiro

DCC/FCUP

2026/2027







Fibonacci Numbers

Probably the most famous **number sequence**, defined by Leonardo Fibonacci



0,1,1,2,3,5,8,13,21,34,...

Fibonacci Numbers

$F(0) = 0$
$F(1) = 1$
$F(n) = F(n-1) + F(n-2)$

Fibonacci Numbers

- How to implement?
- Implementing directly from the definition:

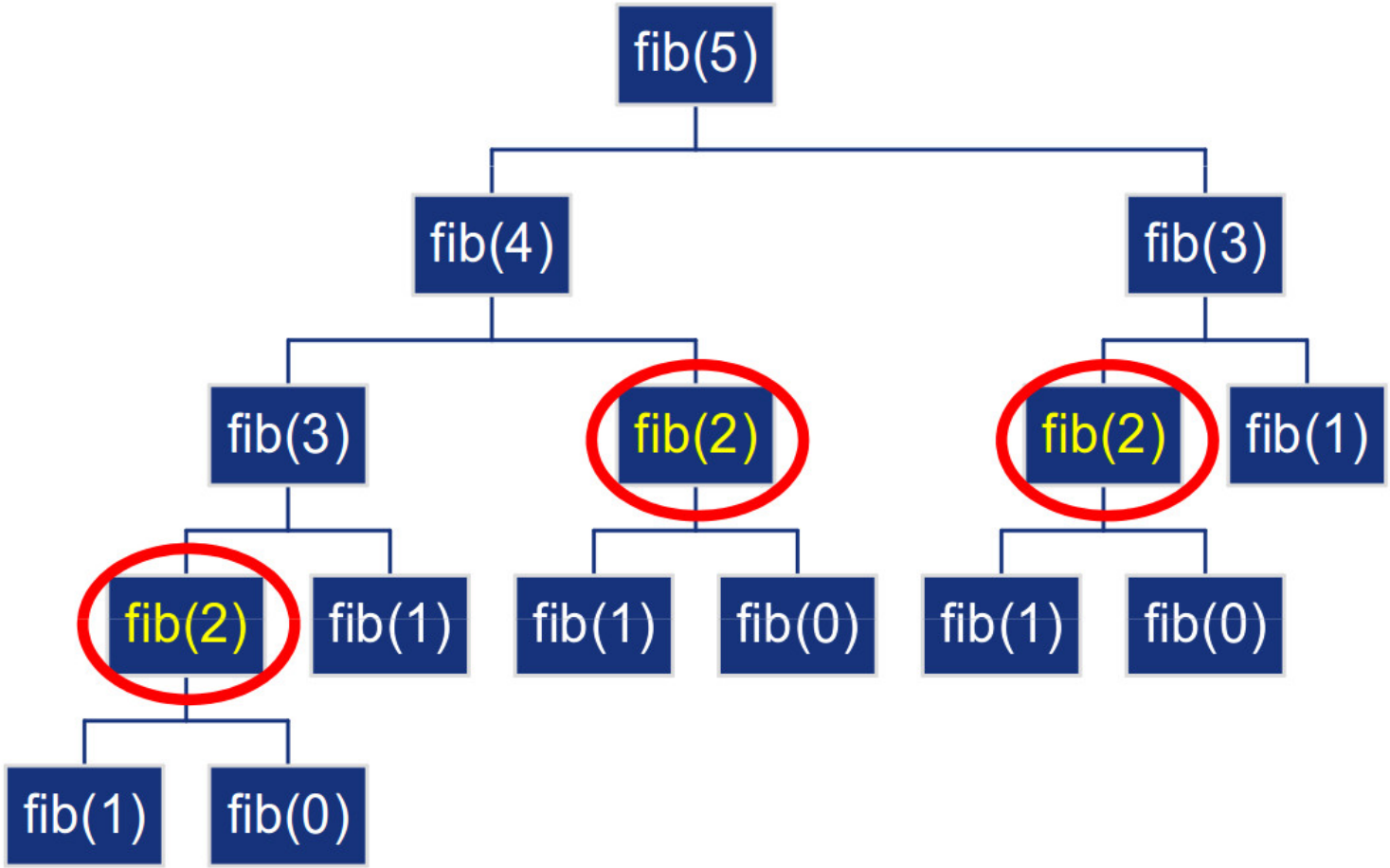
Fibonacci (from the definition)

fib(n):
  If $n = 0$ or $n = 1$ **then**
    return n
  Else
    return $\text{fib}(n-1) + \text{fib}(n-2)$

- Negative points of this implementation?

Fibonacci Numbers

- Computing **fib(**5**)**:



For instance, fib(2) is called 3 times!

Fibonacci Numbers

- How to **improve**? Go from the **start** to the **end**
  - Start from zero and keep in memory the last two numbers of the sequence

Fibonacci (more efficient iterative version)

fib(n):
  If $n = 0$ or $n = 1$ **then**
    return n
  Else
    $f_1 \leftarrow 1$
    $f_2 \leftarrow 0$
    For $i \leftarrow 2$ **to** n **do**
      $f \leftarrow f_1 + f_2$
      $f_2 \leftarrow f_1$
      $f_1 \leftarrow f$
    return f

Fibonacci Numbers

- **Another way**: keep the recursion, but **remember** what we have already computed
  - Keep a table with the values we already know

Fibonacci (with memoization)

Initialization: $memo[i] \leftarrow -1$ for all i // -1 = not computed

fib(n):
  If $n = 0$ or $n = 1$ **then return** n
  If $memo[n] \neq -1$ **then return** $memo[n]$
  $memo[n] \leftarrow \text{fib}(n-1) + \text{fib}(n-2)$
  return $memo[n]$

- Each $fib(k)$ is computed **once**, and every later call is just a lookup: $\mathcal{O}(n)$ instead of exponential
- This technique is called **memoization**

Fibonacci Numbers

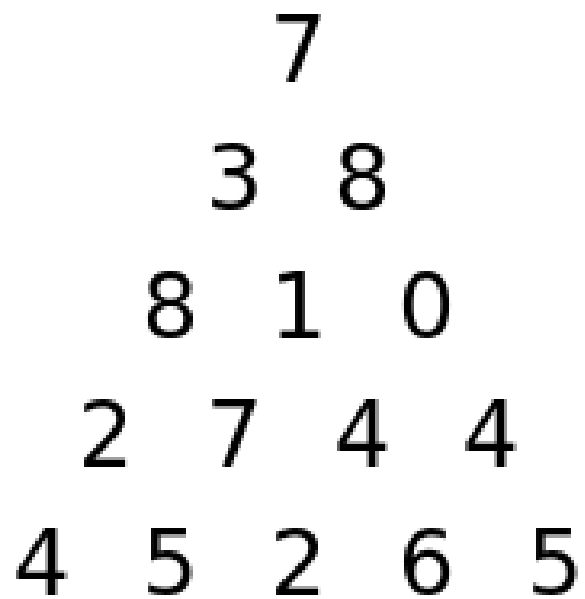
- **Concepts** to recall:
  - Dividing a problem in **subproblems of the same type**
  - Calculating the same subproblem **just once**

- **Can these ideas be used in more "complicated" problems**?

Number Pyramid

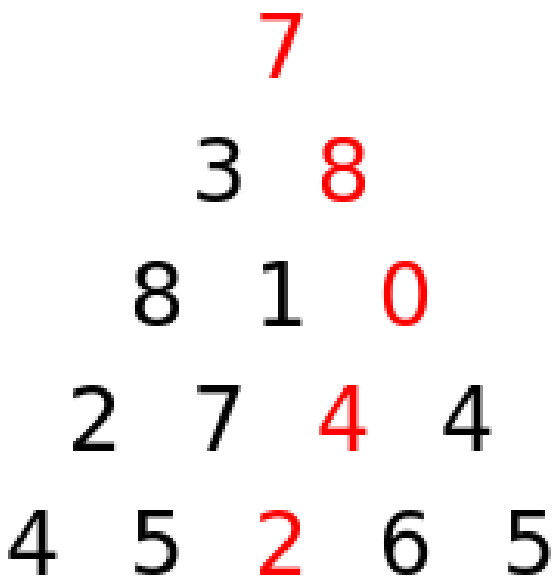
- "Classic" problem from the 1994 **International Olympiad in Informatics**

- Compute the **path**, starting on the top of the pyramid and ending on the base, with the **biggest sum**. In each step we can go diagonally down and left or down and right.



Number Pyramid

- Two possible paths:



Sum = 21



Sum = 30

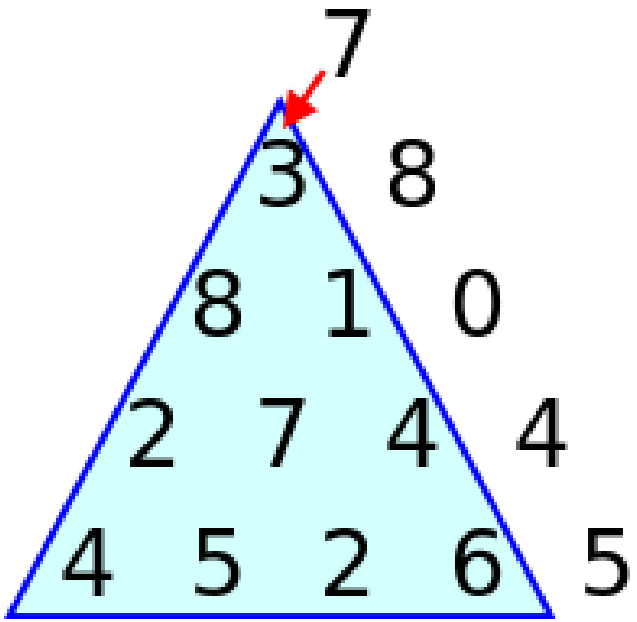
- **Constraints:** all the numbers are integers between 0 and 99 and the number of lines in the pyramid is at most 100.

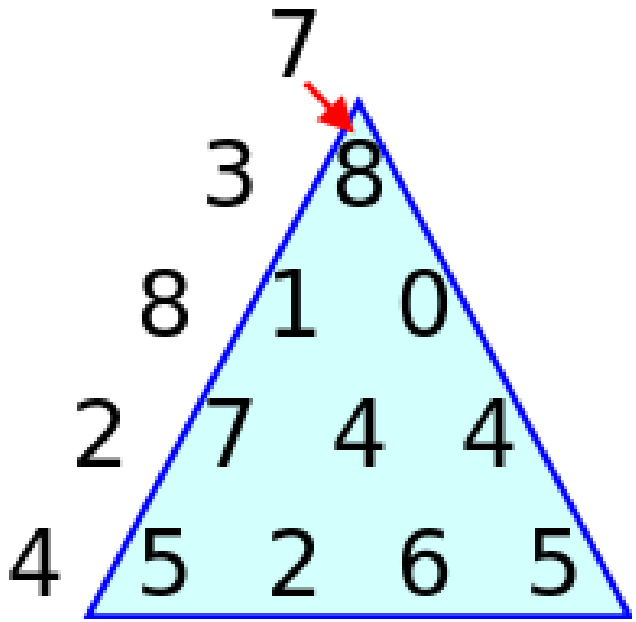
Number Pyramid

- How to solve the problem?
- **Greedy** algorithm does not work... (can you see why?)
- Idea: **Exhaustive search** (*aka* "Brute Force")
  - Evaluate all the paths and choose the best one.
- How much time does this take? How many paths exist?
- Analysing the **temporal complexity**:
  - In each line we can take **one of two decisions**: left or right
  - Let n be the height of the pyramid. A path corresponds to... $n-1$ decisions!
  - There are 2^{n-1} different paths
  - A program to compute all possible paths has therefore complexity $\mathcal{O}(2^n)$: exponential!
  - $2^{99} \sim 6.34 \times 10^{29}$ (633825300114114700748351602688)

Number Pyramid

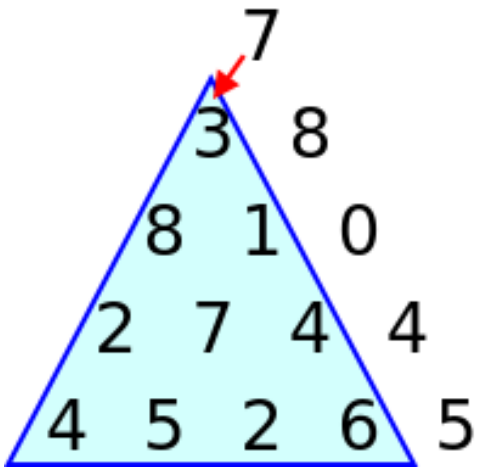
- When we are at the top we have **two possible choices** (left or right):

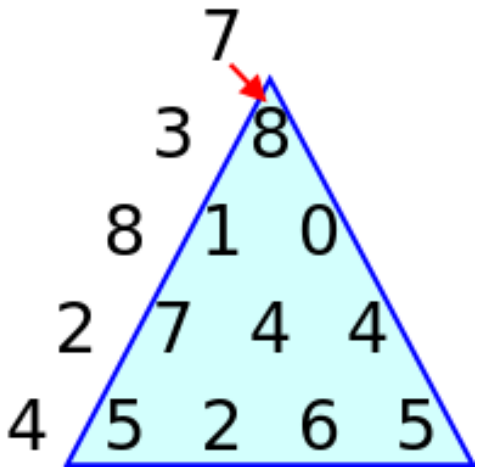




- In each case, we need to take into account **all possible paths of the respective subpyramid**.

Number Pyramid



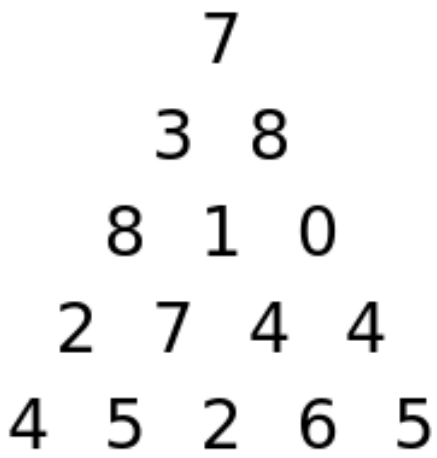


- But what do we really need to know about these subpyramids?
- **The only thing that matters is the best internal path, which is a smaller instance of the same problem!**
- For the example, the solution is 7 plus the maximum between the value of the best paths in each subpyramid

Number Pyramid

- This problem can then be solved **recursively**
  - Let P[i][j] be the j-th number of the i-th line
  - Let Max(i, j) be the best we can do from position (i, j)

	1	**2**	**3**	**4**	**5**
1	7				
2	3	8			
3	8	1	0		
4	2	7	4	4	
5	4	5	2	6	5



Number Pyramid

Number Pyramid (from the recursive definition)

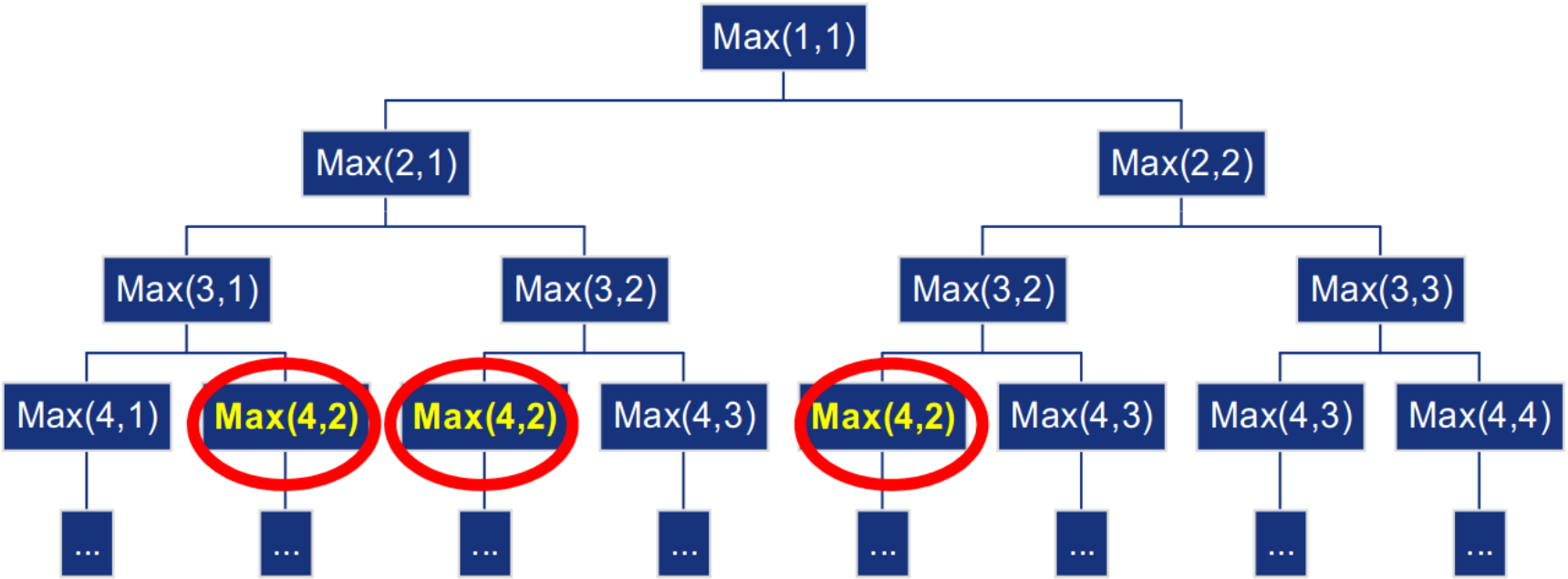
Max(i, j):
  If $i = n$ **then**
    return $P[i][j]$
  Else
    return $P[i][j]$ + **maximum** (Max$(i+1, j)$, Max$(i+1, j+1)$)

- To solve the problem we just need to call... **Max(1,1)**

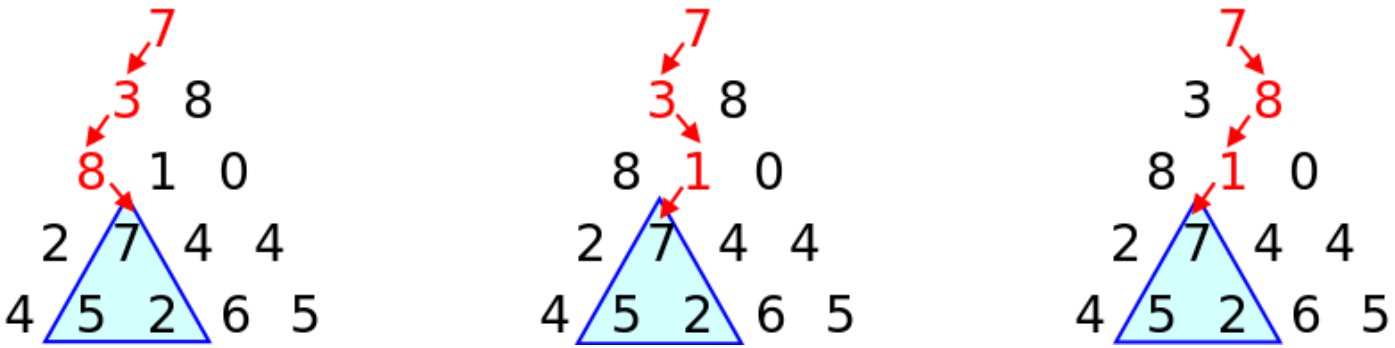
	1	**2**	**3**	**4**	**5**
1	7				
2	3	8			
3	8	1	0		
4	2	7	4	4	
5	4	5	2	6	5

Number Pyramid

- We still have **exponential growth**!



- We are evaluating the same problem several times...

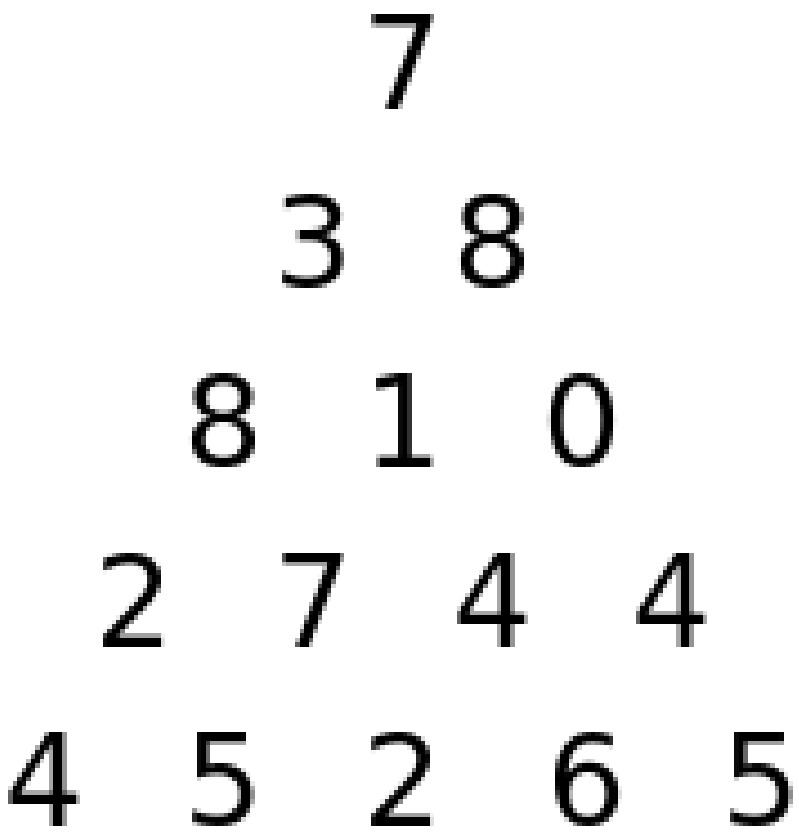


Number Pyramid

- We need to **reuse** what we have already computed
  - Compute only once each subproblem
- Idea: create a **table** with the value we got for each subproblem
  - Matrix M[i][j]
- Is there an **order to fill the table** so that when we need a value we have already computed it?

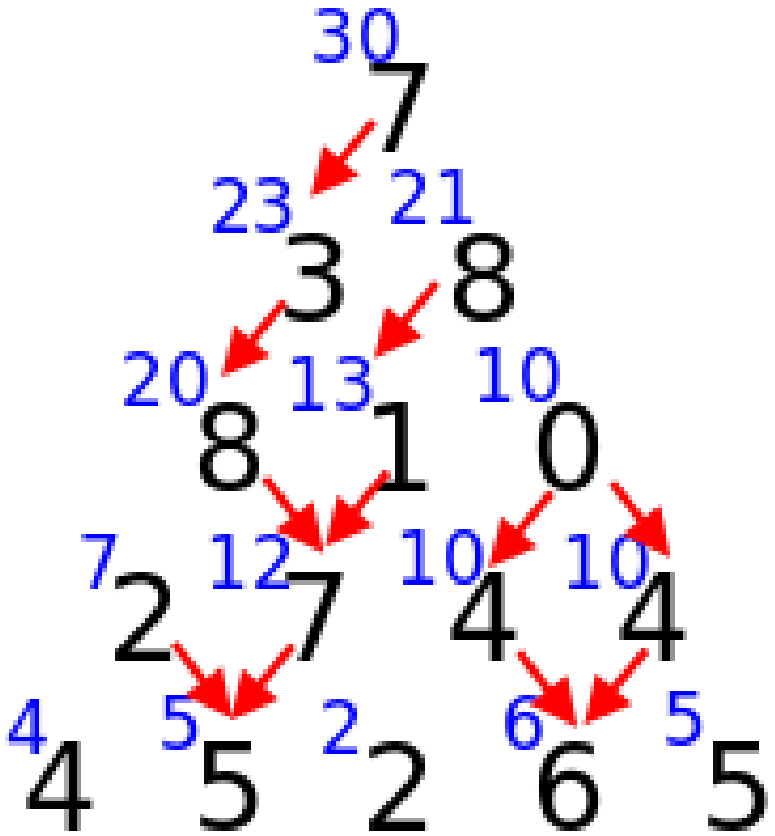
Number Pyramid

- We can start from the end! (pyramid base)



Number Pyramid

- We can start from the end! (pyramid base)



Number Pyramid

- Having in mind the way we fill the table, we can even reuse P[i][j]:

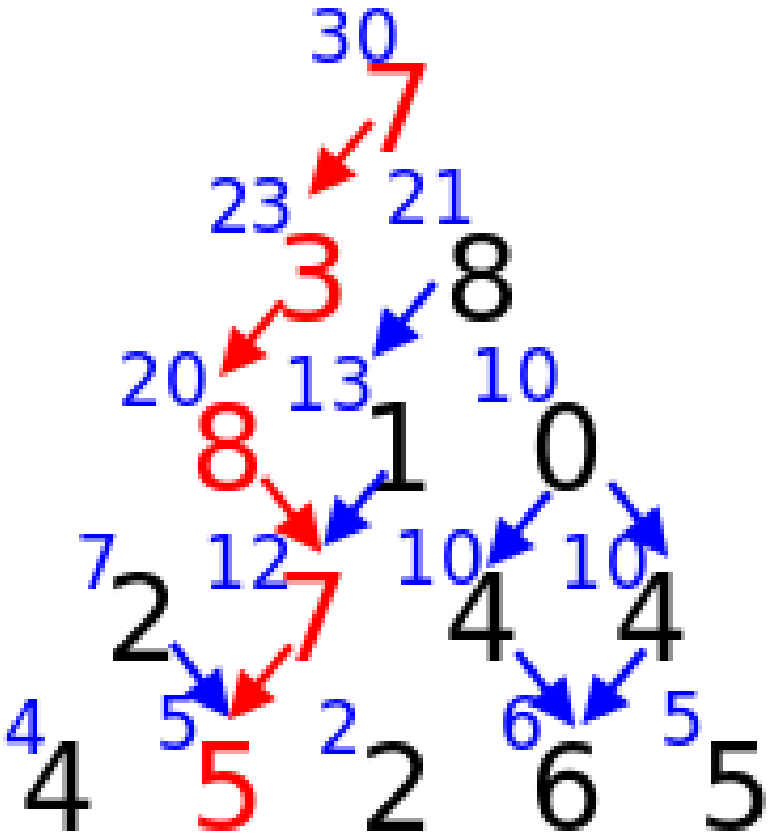
Number Pyramid (polynomial solution)

Compute():
  For $i \leftarrow n-1$ **to** 1 **do**
    For $j \leftarrow 1$ **to** i **do**
      $P[i][j] \leftarrow P[i][j] +$ **maximum** $(P[i+1][j], P[i+1][j+1])$

- With this the solution is in... P[1][1]
- Now the time needed to solve the problem only grows polynomially ($\mathcal{O}(n^2)$) and we have an admissible solution for the problem ($99^2 = 9801$)

Number Pyramid

- What if we need to know what are the numbers in the best path? **We can use the computed table!**



Number Pyramid

To solve the number pyramid we used...

Dynamic Programming (DP)

Dynamic Programming

Dynamic Programming

An **algorithmic technique**, typically used in **optimization problems**, which is based on storing the results of subproblems instead of recomputing them.

- **Algorithmic Technique:** general method for solving problem that have some common characteristics
- **Optimization Problem:** find the "best" solution among all possible solutions, according to a certain criteria (goal function). Normally it means finding a minimum or a maximum.

Classic trade of **space** for **time**

Dynamic Programming

What are then the **characteristics** that a problem must present so that it can be solved using DP?

- Optimal substructure
- Overlapping subproblems

Dynamic Programming - Characteristics

Optimal substructure

When the optimal solution of a problem contains in itself solutions for subproblems of the same type

Example

On the number pyramid problem, the optimal solution contains in itself optimal solutions for subpyramids

- If a problem presents this characteristic, we say that it respects the **optimality principle**.

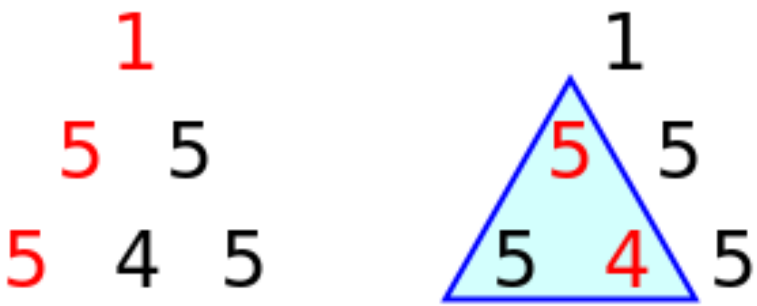
Dynamic Programming - Characteristics

Be careful!

Not all problems present optimal substructure!

Example without optimal substructure

Imagine that in the problem of the number pyramid the goal is to find the path that maximizes the remainder of the integer division between the sum of the values of the path and 10.



The optimal solution ($1 \rightarrow 5 \rightarrow 5$) does not contain the optimal solution for the subpyramid shown ($5 \rightarrow 4$)

Dynamic Programming - Characteristics

Overlapping Subproblems

When the search space is "small", that is, there are not many subproblems to solve because many subproblems are essentially equal.

Example

In the problem of the number pyramid, for a certain problem instance, there are only $n + (n-1) + ... + 1 < n^2$ subproblems because, as we have seen, many subproblems are coincident

Dynamic Programming - Characteristics

Be careful!

This characteristic is also not always present.

- Even with overlapping subproblems there are too many subproblems to solve

  or

- There is no overlap between subproblems

Example

In MergeSort, each recursive call is made to a new subproblem, different from all the others.

Dynamic Programming - Methodology

- If a problem presents these **two characteristics**, we have a good hint that DP is applicable.
- What **steps** should we then follow to solve a problem with DP?

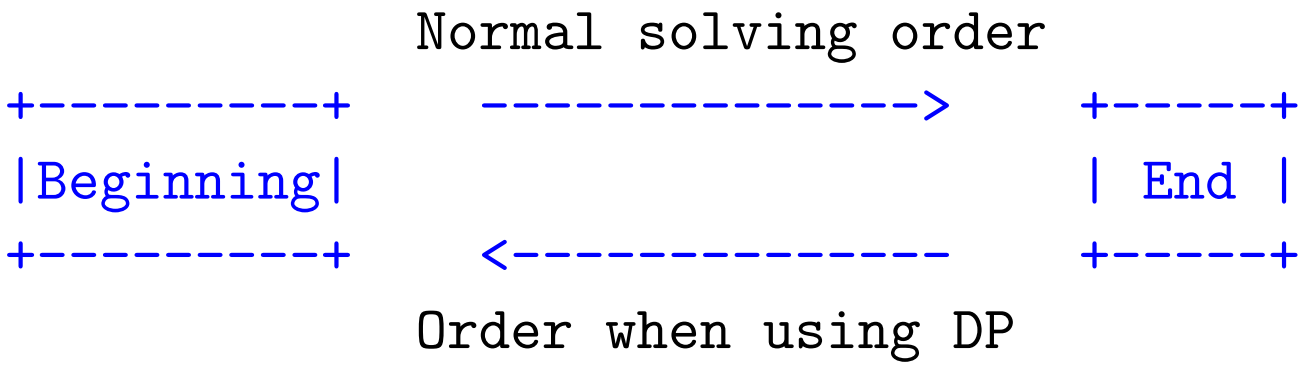
Guide to solve with DP

1. **Characterize** the optimal solution of the problem
   - **Understand** the problem. If **brute force** is not enough, **generalize** it and **divide** it in subproblems
   - Verify the **optimality principle** and the **overlapping subproblems**
2. **Recursively define** the optimal solution, by using optimal solutions of subproblems
   - Exactly, as if the subproblem values were available (no code needed)
3. **Compute** the solutions of all subproblems: bottom-up or top-down
4. **Reconstruct** the optimal solution (optional - only if necessary): **directly** from the table, or with a **new table** of decisions

Dynamic Programming - Methodology

3) Compute the solutions of all subproblems: bottom-up

- **Find the order** in which the subproblems are needed, from the smaller subproblem until we reach the global problem and implement, using a table (*tabulation*)
- Usually this **order is the inverse** to the normal order of the recursive function that solves the problem



3) Compute the solutions of all subproblems: top-down

- Remember **"memoization"** from Fibonacci? It allows us to solve the problem by the normal order.
- Just use the **recursive function** directly obtained from the definition of the solution and keep a table with the results already computed.
- When we need to **access a value** for the first time we need to compute it, and from then on we just need to see the already computed result.

Memoization - Number Pyramid

- Start from the recursive solution we already have, and add a **cache**:

Number Pyramid (memoization)

Initialization: $memo[i][j] \leftarrow -1$ for all i, j // -1 = not computed

Max(i, j):
  If $i = n$ **then return** $P[i][j]$
  If $memo[i][j] \neq -1$ **then return** $memo[i][j]$ // already computed
  $memo[i][j] \leftarrow P[i][j] + \text{maximum}(Max(i+1, j), Max(i+1, j+1))$
  return $memo[i][j]$

- Only two new lines (plus the initialization): the recursion itself does not change
- Each state (i, j) is computed **once**, and every later call is a lookup
- $\mathcal{O}(n^2)$ states $\times$ $\mathcal{O}(1)$ per state $= \mathcal{O}(n^2)$, as in the bottom-up version

Top-down vs Bottom-up

Top-down (memoization)

- \+ Direct translation of the recurrence
- \+ Computes only the states that are needed
- – Recursion overhead, and deep recursion can overflow the stack
- – Harder to save memory

Bottom-up (tabulation)

- \+ No recursion, usually faster
- \+ Can reduce memory (ex: one row of knapsack)
- – We must find the right **order** of the states
- – May compute states that are never used

Be careful when using memoization

- **Reset** the table for each test case (when the input has several)
- The "not computed" marker (ex: -1) must be a value that can never be a real answer

Longest Increasing Subsequence (LIS)

- Given a number sequence:

  7, 6, 10, 3, 4, 1, 8, 9, 5, 2

- Compute the **longest increasing subsequence** (not necessarily contiguous)

  7, 6, 10, 3, 4, 1, 8, 9, 5, 2 (Size 2)
  7, 6, 10, 3, 4, 1, 8, 9, 5, 2 (Size 3)
  7, 6, 10, 3, 4, 1, 8, 9, 5, 2 (Size 4)

Longest Increasing Subsequence (LIS)

1) Characterize the optimal solution of the problem

- Let n be the size of the sequence and num[i] the *i*-th number
- "Brute force", how many options? **Exponential!**
- Generalize and solve with subproblems of the same type:
  - Let **dp(i)** be the size of the best subsequence starting from the *i*-th position
  - **Base case:** the best subsequence from the last position has size... 1!
  - **General case:** for a given i, we can continue to all numbers from $i+1$ to n, as long as they are... strictly bigger
    - ⋆ For those numbers, we only need to know the best starting from them! **(optimality principle)**
    - ⋆ The best, starting from a position, is necessary for computing all the positions of lower index! **(overlapping subproblems)**

Longest Increasing Subsequence (LIS)

2) Recursively define the optimal solution, by using optimal solutions of subproblems

- **n** - sequence size
- **num[i]** - number in position i
- **dp(i)** - size of best sequence starting in position i

Recursive Solution for LIS Problem

$\text{dp}(n) = 1$

$\text{dp}(i) = 1 + \textbf{max}\{\text{dp}(j)\colon\ i < j \leq n, num[j] > num[i]\}$

for $1 \leq i < n$

Longest Increasing Subsequence (LIS)

3) Compute the solutions of all subproblems: bottom-up

- Let **dp[]** the table to save the values of dp()

LIS Problem - $\mathcal{O}(n^2)$

Compute():

$dp[n] \leftarrow 1$
For $i \leftarrow n-1$ **to** 1 **do**
  $dp[i] \leftarrow 1$
  For $j \leftarrow i+1$ **to** n **do**
    If $num[j] > num[i]$ **and** $1 + dp[j] > dp[i]$ **then**
      $dp[i] \leftarrow 1 + dp[j]$

i	1	2	3	4	5	6	7	8	9	10
num[i]	7	6	10	3	4	1	8	9	5	2
dp[i]	**3**	**3**	**1**	**4**	**3**	**3**	**2**	**1**	**1**	**1**

Longest Increasing Subsequence (LIS)

4) Reconstruct the optimal solution

- We will exemplify with an auxiliary table that stores the decisions
- Let **next[i]** be the next position in order to obtain the best solution from position *i* ('X' if it is the last position of the solution).

i	1	2	3	4	5	6	7	8	9	10
num[i]	7	6	10	3	4	1	8	9	5	2
dp[i]	**3**	**3**	**1**	**4**	**3**	**3**	**2**	**1**	**1**	**1**
next[i]	**7**	**7**	**X**	**5**	**7**	**7**	**8**	**X**	**X**	**X**

Longest Increasing Subsequence (LIS)

How can we improve from $\mathcal{O}(n^2)$ to $\mathcal{O}(n \log n)$?

LIS Problem - $\mathcal{O}(n^2)$

Compute():

$dp[n] \leftarrow 1$
For $i \leftarrow n-1$ **to** 1 **do**
  $dp[i] \leftarrow 1$
  For $j \leftarrow i+1$ **to** n **do**
    If $num[j] > num[i]$ **and** $1 + dp[j] > dp[i]$ **then**
      $dp[i] \leftarrow 1 + dp[j]$

- We can change the second loop and transform it into **binary search**

Longest Increasing Subsequence (LIS)

- The inner loop looks for the best **continuation** of position i: a sequence starting at some $j > i$ with $num[j] > num[i]$
- Many continuations have the same length. Which one is easiest to extend backwards?
  - For a given length L, the one that starts with the largest value: it accepts more numbers in front of it, since any smaller number can come before it
- So, for each length L, we only need to remember one thing:
  - Let **index[L]** be the position k with the largest $num[k]$ such that there is an increasing sequence of length L starting at k
- This table has a very useful property, since from any sequence of length $L + 1$ we can drop its first number to get one of length L that starts with a bigger value:

$$num[index[1]] > num[index[2]] > num[index[3]] > \ldots$$

  - The values are **sorted**, so we can use **binary search**!

Longest Increasing Subsequence (LIS)

- When processing position i (from n down to 1):
  1. Binary search the longest length L with $num[index[L]] > num[i]$ (or $L = 0$ if none)
  2. So $dp[i] = L + 1$
  3. Position i is now the best start for length $L + 1$: $index[L + 1] \leftarrow i$

LIS Problem - $\mathcal{O}(n \log n)$

Compute():

$len \leftarrow 0$ // longest sequence found so far
For $i \leftarrow n$ **downto** 1 **do**
  $L \leftarrow$ **BinarySearch**(i)
    // largest $L \in [0, len]$ with $L = 0$ or $num[index[L]] > num[i]$
  $dp[i] \leftarrow L + 1$
  $index[L + 1] \leftarrow i$
  If $L + 1 > len$ **then** $len \leftarrow L + 1$
Return len // the size of the LIS

Longest Increasing Subsequence (LIS)

- Is it safe to **overwrite** $index[L+1]$?
  - Yes! L is the longest length that still fits $num[i]$, so $L+1$ did not fit: $num[index[L+1]] \leq num[i]$
  - Starting at i is therefore never worse for length $L+1$
- Why $num[index[L]] > num[i]$ and not $\geq$?
  - We want a **strictly** increasing sequence, so $num[i]$ must be strictly smaller than the number that follows it
- Each of the n iterations does one binary search over at most n lengths: $\mathcal{O}(n \log n)$ in total
- Let's see it running on our example...

Longest Increasing Subsequence (LIS)

- Position $i = 10$, $num[i] = 2$: no length fits ($L = 0$) $\Rightarrow$ $dp[10] = 1$ and $index[1] \leftarrow 10$

i	1	2	3	4	5	6	7	8	9	**10**
num[i]	7	6	10	3	4	1	8	9	5	2
dp[i]										**1**

L	1	2	3	4
index[L]	**10**			
num[index[L]]	**2**			

Longest Increasing Subsequence (LIS)

- Position $i = 7$, $num[i] = 8$: longest fitting length is $L = 1$ ($num[index[1]] = 9 > 8$) $\Rightarrow$ $dp[7] = 2$ and $index[2] \leftarrow 7$

i	1	2	3	4	5	6	**7**	8	9	10
num[i]	7	6	10	3	4	1	8	9	5	2
dp[i]							**2**	**1**	**1**	**1**

L	1	2	3	4
index[L]	**8**	**7**		
num[index[L]]	**9**	**8**		

Longest Increasing Subsequence (LIS)

- The size of the LIS is the longest length found: $len = 4$
- Each of the n iterations takes $\log n$ (one binary search), and so our total complexity is $\mathcal{O}(n \log n)$

i	1	2	3	4	5	6	7	8	9	10
num[i]	7	6	10	3	4	1	8	9	5	2
dp[i]	**3**	**3**	**1**	**4**	**3**	**3**	**2**	**1**	**1**	**1**

L	1	2	3	4
index[L]	**3**	**7**	**1**	**4**
num[index[L]]	**10**	**8**	**7**	**3**

0-1 Knapsack

0-1 Knapsack Problem

Input: A backpack of capacity C
A set of n materials, each one with weight w_i and value v_i

Output: The allocation of materials to the backpack that maximizes the transported value.

The materials **cannot be "broken"** in smaller pieces, that is, for a given material i we can either take it all ($x_i = 1$) or we leave it all ($x_i = 0$)

What we want is therefore to obey the following constraints

- The materials fit in the backpack ($\sum_i x_i w_i \leq C$)
- The value transported is the maximum possible (maximize $\sum_i x_i v_i$)

0-1 Knapsack

- A greedy solution will not work on this integer case

Input Example

Input: 5 objects and $C = 11$

i	1	2	3	4	5
w_i	1	2	5	6	7
v_i	1	6	18	22	28
v_i/w_i	1	3	3.6	3.66	4

- Choosing **max ratio** first will result in $\{1, 2, 5\}$ with a value of 35
- Choosing **max value** first will also result in $\{1, 2, 5\}$ with a value of 35
- Choosing **min weight** first will result in $\{1, 2, 3\}$ with a value of 25
- ...
- None of these is **optimal**: we could get a value of 40 with $\{3, 4\}$

0-1 Knapsack

1) Characterize the optimal solution of the problem

- "Brute force", how many options? **Exponential (2^n)!**

- Let's first consider the **unbounded** case
  (no limit on number of items of each type)
- In this unbounded case we could generalize in the following way:
  - Let **dp(i)** be the best value we can get for capacity i
  - **Base case:** dp(0) = 0 (obviously)
  - **General case:** for a given i, we can simple see all possible items and get the best if we insert that item:
    dp(i) = **max** $(v_j + dp(i - w_j) : 1 \leq j \leq n, w_j \leq i)$ for $1 \leq i \leq C$

- But how can we **limit** the amount of items of each type?

0-1 Knapsack

1) Characterize the optimal solution of the problem

- Let's add more information to our **DP state**
  - Let **dp(i, j)** be the best value we can get for capacity j using only the first i materials
  - For computing the values of a given $dp(i, j)$ we can now simply use the values of previously computed $dp(i-1, k)$
  - Let's put all the pieces into place...

0-1 Knapsack

2) Recursively define the optimal solution

- n - number of materials
- w_i - weight of material i
- v_i - value of material i
- **C** - capacity of backpack
- **dp(i,j)** - maximum possible value for capacity j using the first i materials

Recursive Solution for 0-1 Knapsack

$dp(0,j) = 0$ for $0 \leq j \leq C$

$$dp(i,j) = dp(i-1,j) \qquad \text{if } (w_i > j)$$
$$dp(i,j) = \textbf{max}\ \{dp(i-1,j), dp(i-1,j-w_i) + v_i\} \qquad \text{if } (w_i \leq j)$$
$$\text{for } 1 \leq i \leq n, 0 \leq j \leq C$$

The desired result is: **dp(n,C)**

0-1 Knapsack

Let's see a table of results to better understand the DP formulation:

Input Example

Input: $n = 5$ and $C = 11$

i	1	2	3	4	5
w_i	1	2	5	6	7
v_i	1	6	18	22	28

Capacity

Items	0	1	2	3	4	5	6	7	8	9	10	11
{}	0	0	0	0	0	0	0	0	0	0	0	0
{1}	0	1	1	1	1	1	1	1	1	1	1	1
{1,2}	0	1	6	7	7	7	7	7	7	7	7	7
{1,2,3}	0	1	6	7	7	18	19	24	25	25	25	25
{1,2,3,4}	0	1	6	7	7	18	22	24	28	29	29	40
{1,2,3,4,5}	0	1	6	7	7	18	22	28	29	34	35	40

0-1 Knapsack

3) Compute the solutions of all subproblems: bottom-up

- Let **dp[][]** be the matrix that stores the values of the DP states

0-1 Knapsack Problem - $\mathcal{O}(n \times C)$

Compute():

For $j \leftarrow 0$ **to** C **do**
  $dp[0][j] \leftarrow 0$
For $i \leftarrow 1$ **to** n **do**
  For $j \leftarrow 0$ **to** C **do**
    If $weight[i] > j$
      $dp[i][j] \leftarrow dp[i-1][j]$
    Else
      $dp[i][j] \leftarrow max(dp[i-1][j], dp[i-1][j - weight[i]] + value[i])$

0-1 Knapsack

Can we improve memory usage from the $\mathcal{O}(\text{n} \times \text{C})$ bound?

- Yes: each row of the table only needs the values of the previous row
- We could use $\mathcal{O}(C)$ memory by simply just storing previous row
- In fact, if we carefully consider the order in which we compute the values, we can simply just store one row:
  - Let **dp[]** be the array that stores the current row

0-1 Knapsack Problem - $\mathcal{O}(n \times C)$ time, $\mathcal{O}(C)$ memory

Compute():

For $j \leftarrow 0$ **to** C **do**
  $dp[j] \leftarrow 0$
For $i \leftarrow 1$ **to** n **do**
  For $j \leftarrow C$ **downto** 0 **do**
    If $weight[i] \leq j$
      $dp[j] \leftarrow max(dp[j], dp[j - weight[i]] + value[i])$

0-1 Knapsack

4) Reconstruct the optimal solution

- If needed we could store for each position how we obtained its value:
  - We either used current item or we did not use it

i	1	2	3	4	5
w_i	1	2	5	6	7
v_i	1	6	18	22	28

Capacity

Items	0	1	2	3	4	5	6	7	8	9	10	11
{}	**0**	0	0	0	0	0	0	0	0	0	0	0
{1}	**0**	1	1	1	1	1	1	1	1	1	1	1
{1,2}	**0**	1	6	7	7	7	7	7	7	7	7	7
{1,2,3}	0	1	6	7	7	**18**	19	24	25	25	25	25
{1,2,3,4}	0	1	6	7	7	18	22	24	28	29	29	**40**
{1,2,3,4,5}	0	1	6	7	7	18	22	28	29	34	35	**40**

4) Reconstruct the optimal solution: backtracking

- Start at the bottom right cell, $dp(5, 11) = 40$, and go back up:
- **Item 5**: $dp(4, 11) = 40 = dp(5, 11) \Rightarrow$ item 5 was not taken
- **Item 4**: $dp(3, 11) = 25 \neq 40 \Rightarrow$ item 4 was taken.
  Capacity left: $11 - 6 = 5$
- **Item 3**: $dp(2, 5) = 7 \neq 18 = dp(3, 5) \Rightarrow$ item 3 was taken.
  Capacity left: $5 - 5 = 0$
- **Items 2 and 1**: capacity is 0, so they were not taken
- The solution is $\{3, 4\}$: weight $5 + 6 = 11$ and value $18 + 22 = 40$

0-1 Knapsack

Reconstruction of the 0-1 Knapsack solution

Reconstruct():

$j \leftarrow C$

For $i \leftarrow n$ **downto** 1 **do**

  If $dp[i][j] \neq dp[i-1][j]$ **then** // item i was used

    add item i to the solution

    $j \leftarrow j - weight[i]$

- This needs the **full table** $dp[][]$
- With the $\mathcal{O}(C)$ memory version we lose the previous rows, so we can only recover the **value**, not the items (unless we store the decisions)

Bounded Knapsack

Bounded Knapsack Problem

Same as 0-1 knapsack, but material i now has k_i **copies** available: we can take it $0, 1, \ldots, k_i$ times

- Simplest idea: replace item i by k_i separate items, all equal, and use 0-1 knapsack
- The number of items becomes $\sum_i k_i$, and the time is $\mathcal{O}(C \times \sum_i k_i)$
- Is this fast enough? Imagine $n = 100$ materials, each with $k_i = 1000$ copies, and $C = 10^5$:

$$\sum_i k_i = 10^5 \text{ items} \Rightarrow 10^5 \times 10^5 = 10^{10} \text{ operations}$$

- Can we avoid treating each copy as a "different" element?

Bounded Knapsack

- **Key idea:** we do not need one item per copy, just enough items so that every number of copies between 0 and k_i can be made
- Split the k_i copies in groups of size $1, 2, 4, 8, \ldots$ and a final group with the remainder
- Example: $k_i = 13 \quad \Rightarrow \quad$ groups of 1, 2, 4, 6
  - With $1, 2, 4$ we can make any number from 0 to 7
  - Adding the group of 6, we can make any number from 0 to 13
- Each group becomes **one 0-1 item**: a group of g copies has weight $g \times w_i$ and value $g \times v_i$
- Only about $\log k_i$ items per material, instead of k_i!
  - The last group is the remainder (and not the next power of two) so we never exceed k_i copies

Bounded Knapsack

Bounded Knapsack - split in groups

Split():
  $items \leftarrow$ empty list
  For $i \leftarrow 1$ **to** n **do**
    $rest \leftarrow k[i]$ $g \leftarrow 1$
    While $rest > 0$ **do**
      $take \leftarrow \text{minimum}(g, rest)$
      add to $items$ the pair $(take \times weight[i], take \times value[i])$
      $rest \leftarrow rest - take$ $g \leftarrow 2 \times g$

- Then just run the 0-1 knapsack ($\mathcal{O}(C)$ memory version) over *items*
- Time: $\mathcal{O}(C \times \sum_i \log k_i)$. In our example, $100 \times 10 \times 10^5 = 10^8$

0-1 Knapsack variants

- There are many possible variants of the knapsack problem.
- **Subset sum**
  - Given a set S of integers and value K, verify if we can obtain a subset of S that sums to K
  - Ex: $S = \{1, 3, 5, 10\}$
    - ⋆ $K = 8$ has answer "yes" because $3 + 5 = 8$
    - ⋆ $K = 7$ has answer "no" because no subset of S has sum 7
  - It's the "same" as knapsack if we disregard values and just consider if a certain weight is achievable:

    $s_i = i$-th element of set S
    $\text{dp}(i, j) =$ is sum j achievable using the first i items? (T/F)

    $dp(0, 0) = T$
    $dp(0, j) = F$ for $1 \leq j \leq K$
    $dp(i, j) = dp(i - 1, j)$ OR $(dp(i - 1, j - s_i)$ AND $s_i \leq j)$
    for $1 \leq i \leq n, 0 \leq j \leq K$

0-1 Knapsack variants

- There are many possible variants of the knapsack problem.
- **Minimum number of coins**
  - Given a set S of coins and a value K, find the minimum number of coins to form quantity K (assume it is possible to form any quantity) There is an unlimited amount of each coin value
  - Ex: $S = \{1, 10, 25\}$
    - ★ $K = 40$ has answer 4 because $10 + 10 + 10 + 10 = 40$
  - Greedy algorithm does not work (the above is a counter-example)
  - It's the "same" as unbounded knapsack if we consider all values to be the same and we now try to minimize total value

    s_i = i-th element of coin set S
    dp(i) = minimum number of coins to form quantity i

    $dp(0) = 0$
    $dp(i) = \textbf{min}\{\ 1 + \text{dp}(i - s_j)\colon\ s_j \leq i,\ 1 \leq j \leq n\ \}$
    for $1 \leq i \leq K$

Edit Distance

- Let's look at another problem, this time with strings:

Edit Distance Problem

Consider two words w_1 and w_2. Our goal is to **transform** w_1 **into** w_2 using only 3 types of transformations:

1. Remove a letter
2. Insert a letter
3. Substitute one letter with another one

What is the **minimum number of transformations** that we have to do to turn one word into the other? This metric is known as **edit distance** (ed).

Example

In order to turn "ghost" into "horse" we need 3 transformations:

$$\text{ghost} \xrightarrow{(1)} \text{_host} \xrightarrow{(2)} \text{horst} \xrightarrow{(3)} \text{horse}$$

Edit Distance

1) Characterize the optimal solution of the problem

- Let **ed(a,b)** be the edit distance between *a* and *b*
- Let "" be the empty word
- Are there any simple cases?
  - Clearly **ed("","")** is zero
  - **ed("",b)**, for any word *b*? It is the size of word *b* (we need to make **insertions**)
  - **ed(a,"")**, for any word *a*? It is the size of word *a* (we need to make **removals**)
- And in the other cases? We must try dividing the problem in subproblems, where we can decide based on the solution of the subproblems.

Edit Distance

- None of the words is empty
- How can we equalize the end of both words?
  - Let l_a be the last letter of a and a' the remaining letters of a
  - Let l_b be the last letter of b and b' the remaining letters of b
- If $l_a = l_b$, then all that is left is to find the edit distance between a' and b' (a smaller instance of the same problem!)
- Otherwise, we have three possible movements:
  - **Substitute** l_a with l_b. We spend 1 transformation and now we need the edit distance between a' and b'.
  - **Remove** l_a. We spend 1 transformation and now we need the edit distance between a' and b.
  - **Insert** l_b at the end of a. We spend 1 transformation and now we need the edit distance between a and b'.

Edit Distance

2) Recursively define the optimal solution

- |a| and |b| - size (length) of words a and b
- **a[i]** and **b[i]** - letter on position i of words a and b
- **dp(i,j)** - edit distance between the first i letters of a and the first j letters of b

Recursive solution for Edit Distance Problem

$\text{dp}(i, 0) = i$, for $0 \leq i \leq |a|$
$\text{dp}(0, j) = j$, for $0 \leq j \leq |b|$

$$\text{dp}(i,j) = \textbf{min}(\text{dp}(i-1, j-1) + \{0 \text{ if } a[i] = b[j],\ 1 \text{ if } a[i] \neq b[j]\},$$
$$\text{dp(i-1, j)} + 1,$$
$$\text{dp(i, j-1)} + 1)$$

for $1 \leq i \leq |a|$ and $1 \leq j \leq |b|$

Edit Distance

3) Compute the solutions of all subproblems: bottom-up

Edit Distance (polynomial solution)

Compute():

For $i \leftarrow 0$ **to** $|a|$ **do** $dp[i][0] \leftarrow i$
For $j \leftarrow 0$ **to** $|b|$ **do** $dp[0][j] \leftarrow j$

For $i \leftarrow 1$ **to** $|a|$ **do**
  For $j \leftarrow 1$ **to** $|b|$ **do**
    If $a[i] = b[j]$ **then** $value \leftarrow 0$
    Else $value \leftarrow 1$
    $dp[i][j] \leftarrow$ **minimum**($dp[i-1][j-1] + value$,
                $dp[i-1][j] + 1$,
                $dp[i][j-1] + 1$)

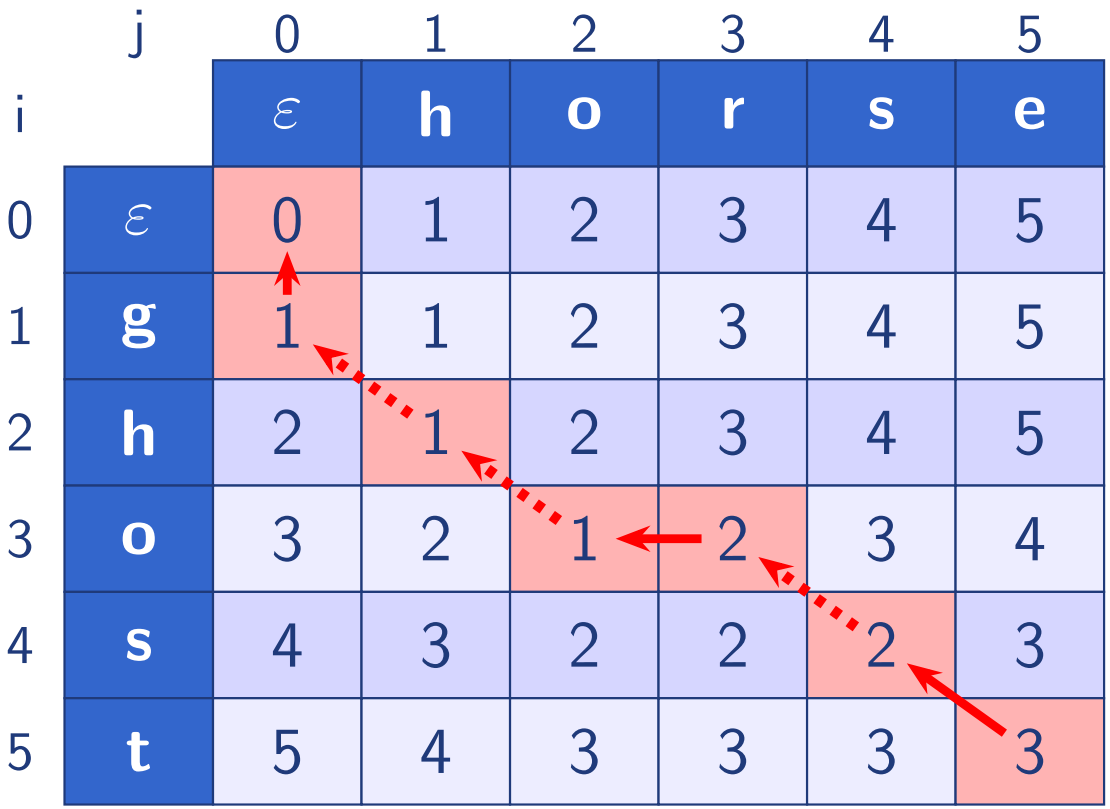
Edit Distance

- Let's see the **table** for the edit distance between "ghost" and "horse":

i \ j		0	1	2	3	4	5
		ε	**h**	**o**	**r**	**s**	**e**
0	ε	0	1	2	3	4	5
1	**g**	1	1	2	3	4	5
2	**h**	2	1	2	3	4	5
3	**o**	3	2	1	2	3	4
4	**s**	4	3	2	2	2	3
5	**t**	5	4	3	3	3	3

Edit Distance

- If we needed to reconstruct the solution



i \ j		0	1	2	3	4	5
		ε	**h**	**o**	**r**	**s**	**e**
0	ε	0	1	2	3	4	5
1	**g**	1	1	2	3	4	5
2	**h**	2	1	2	3	4	5
3	**o**	3	2	1	2	3	4
4	**s**	4	3	2	2	2	3
5	**t**	5	4	3	3	3	3

ghost $\xrightarrow{(1)}$ _host $\xrightarrow{(2)}$ horst $\xrightarrow{(3)}$ horse

Edit Distance Variants

- There are many possible variants for the edit distance problem. Here is perhaps the most common (and classical one):

- **Longest Common Subsequence Problem (LCS)**
  - Given two strings, find the length of the **longest subsequence** (not necessarily contiguous) common to both strings
  - Ex: LCS("ABAZDC", "BACBAD") = 4 *[corresponding to "ABAD"]*
  - It's the "same" as edit distance if no substitution is allowed (only additions and deletions). Suppose that the strings are a and b:

$$\text{dp}(i, 0) = \text{dp}(0, j) = 0$$

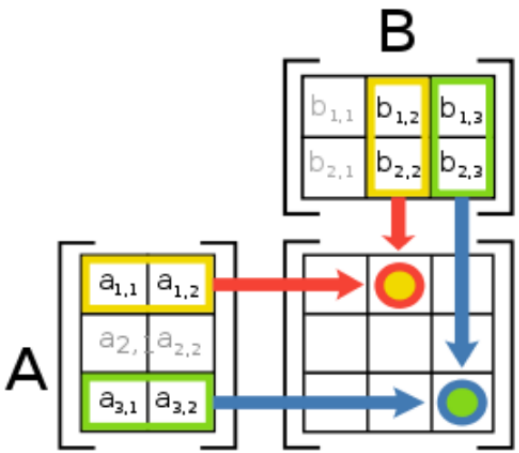
$$\text{dp}(i, j) = \text{dp}(i-1, j-1) + 1 \qquad \text{if } a[i] = b[j]$$
$$\text{dp}(i, j) = \textbf{max}(\text{dp}(i-1, j), \text{dp}(i, j-1)) \qquad \text{if } a[i] \neq b[j]$$
$$\text{for } 1 \leq i \leq |a| \text{ and } 1 \leq j \leq |b|$$

Matrix Chain Multiplication

First some context.

- Let A be an $n \times m$ matrix
- Let B be an $m \times p$ matrix

The product AB is an $n \times p$ matrix whose (i,j)-th entry is $\sum_{k=1}^{m} a_{ik} b_{kj}$



Computing this takes nmp **scalar multiplications**.
Let's use this as the **cost** of one matrix multiplication

Matrix Chain Multiplication

Matrix product is associative: $(AB)C = A(BC)$

We can **choose the order** in which we process the multiplication of several matrices

Suppose we have four matrices:

- $A : 30 \times 1$
- $B : 1 \times 40$
- $C : 40 \times 10$
- $D : 10 \times 25$

$((AB)(CD))$ requires 41200 scalar multiplications

$(A((BC)D))$ requires only 1400 scalar multiplications

What is the best order to **minimize the cost**?

Matrix Chain Multiplication

Matrix Chain Multiplication

Input: n matrices $A_1, \ldots A_2, \ldots, A_n$
where A_i is a $d_{i-1} \times d_i$ matrix

Output: The minimum cost (number of scalar multiplications) required to compute the product $A_1 \bullet A_2 \bullet \ldots \bullet A_n$

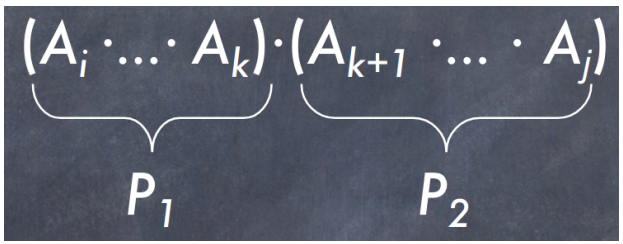
Possibly also the order that gives origin to that minimum

Matrix Chain Multiplication

1) Characterize the optimal solution of the problem
2) Recursively define the optimal solution

- Let **dp(i,j)** be the minimum cost for computing $A_i \bullet \ldots \bullet A_j$
- Our goal is to compute: $dp(1, n)$
- Base case: $dp(i, i) = 0$
- What about the other cases?
  - Consider all the possible ways to split A_i through A_j into two pieces
  - Compare the costs of all these splits and take the best one

$$dp(i, j) = min_k(dp(i, k) + dp(k + 1, j) + d_{i-1} \times d_k \times d_j)$$



P_1 cost is $dp(i, k)$, P_2 cost is $dp(k + 1, j)$
$P_1 \bullet P_2$ cost is $d_{i-1} \times d_k \times d_j$

Matrix Chain Multiplication

3) Compute the solutions of all subproblems

dp:

	1	2	3	4	5
1	0				
2	n/a	0			
3	n/a	n/a	0		
4	n/a	n/a	n/a	0	
5	n/a	n/a	n/a	n/a	0

GOAL!

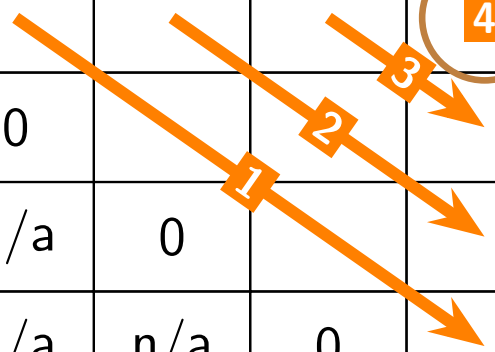
computing the **pink** square requires the **purple** ones: to the left and below.

Matrix Chain Multiplication

3) Compute the solutions of all subproblems

dp:

	1	2	3	4	5
1	0				4
2	n/a	0			
3	n/a	n/a	0		
4	n/a	n/a	n/a	0	
5	n/a	n/a	n/a	n/a	0



A possible order of computation.

Matrix Chain Multiplication

3) Compute the solutions of all subproblems

dp:

	1	2	3	4
1	0	1200	700	1400
2	n/a	0	400	650
3	n/a	n/a	0	10,000
4	n/a	n/a	n/a	0

1: A is 30×1
2: B is 1×40
3: C is 40×10
4: D is 10×25

B×C: 1×40×10

(B×C)×D:
400 + 1×10×25

B×(C×D):
... + 10,000

Matrix Chain Multiplication

3) Compute the solutions of all subproblems: bottom-up

- $dp(i, j)$ needs smaller intervals: compute by increasing size

Matrix Chain Multiplication - $\mathcal{O}(n^3)$

```
Compute():
  For i ← 1 to n do dp[i][i] ← 0
  For size ← 2 to n do     // size of the interval
    For i ← 1 to n − size + 1 do
      j ← i + size − 1
      dp[i][j] ← ∞
      For k ← i to j − 1 do     // try every split
        cost ← dp[i][k] + dp[k + 1][j] + d[i − 1] × d[k] × d[j]
        If cost < dp[i][j] then
          dp[i][j] ← cost
          split[i][j] ← k     // for reconstruction
```

- $\mathcal{O}(n^2)$ states $\times$ $\mathcal{O}(n)$ splits per state $=$ $\mathcal{O}(n^3)$ time, $\mathcal{O}(n^2)$ memory

Matrix Chain Multiplication

4) Reconstruct the optimal solution

- $split[i][j]$ stores the best k for $A_i \bullet \ldots \bullet A_j$
- For our example ($d = 30, 1, 40, 10, 25$):

split	1	2	3	4
1		1	1	**1**
2			2	**3**
3				3

- $split[1][4] = 1$: $A_1 \bullet (A_2 \bullet A_3 \bullet A_4)$
- $split[2][4] = 3$: $A_1 \bullet ((A_2 \bullet A_3) \bullet A_4)$

Print(i, j):
If $i = j$ **then** print A_i
Else print "(", **Print**(i, $split[i][j]$), **Print**($split[i][j] + 1$, j), ")"

Summary

- Every DP problem we have seen follows the same recipe: state, recurrence, order

Problem	**State**	**Transition**	**Time**
Number Pyramid	position (i, j)	best of the 2 below	$\mathcal{O}(n^2)$
LIS	start position i	best next $j > i$	$\mathcal{O}(n^2)$
		(or binary search)	$\mathcal{O}(n \log n)$
0-1 Knapsack	items i, capacity j	take item or not	$\mathcal{O}(nC)$
Edit Distance, LCS	prefixes i and j	match / ins / del / replace	$\mathcal{O}(\|a\|\|b\|)$
Matrix Chain	interval $[i, j]$	try every split k	$\mathcal{O}(n^3)$

- A quick way to estimate the complexity:

$$\text{time} \approx \text{number of states} \times \text{cost of each transition}$$