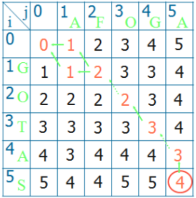
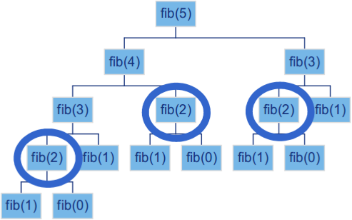
Dynamic Programming

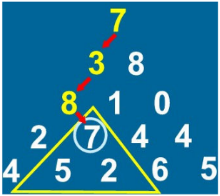
Pedro Ribeiro

DCC/FCUP

2020/2021







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:

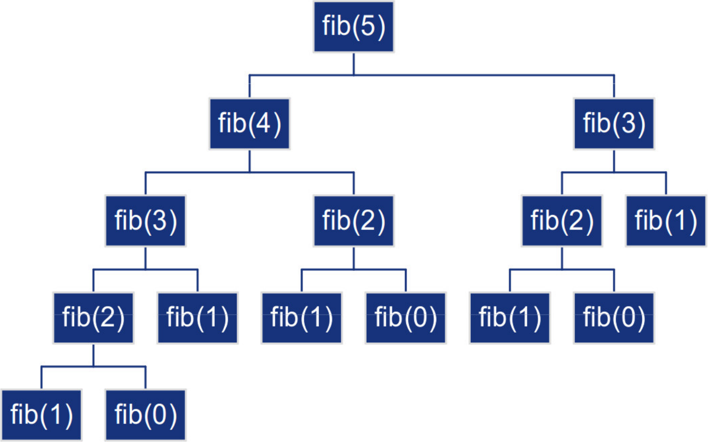
Fibonacci (from the definition)

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

- Negative points of this implementation?

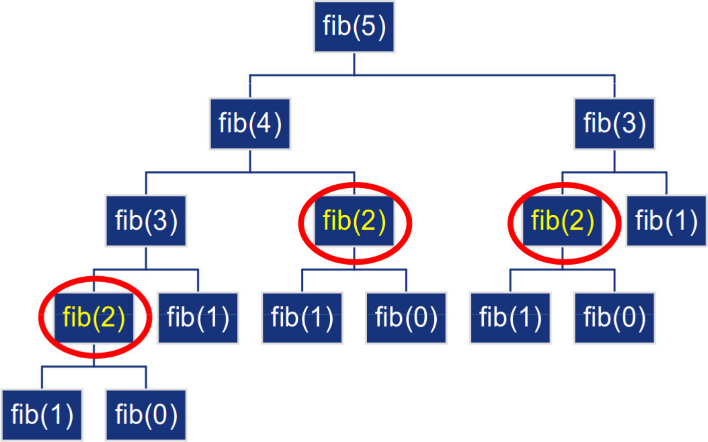
Fibonacci Numbers

- Computing **fib(5)**:



Fibonacci Numbers

- Computing **fib(5)**:



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

Fibonacci Numbers

- How to **improve**?
  - 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

- **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**?

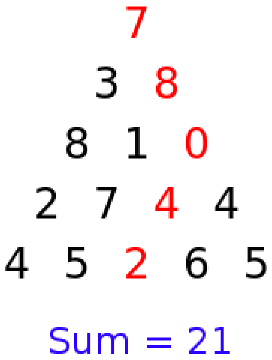
Number Pyramid

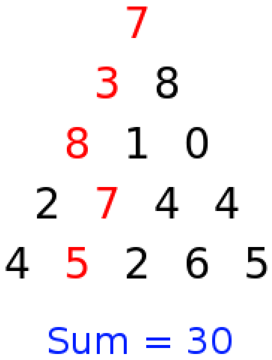
- "Classic" problem from the 1994 **International Olympiad in Informatics**
- Compute the **path**, starting on the top of the pyramid and ending on the base, with the **biggest sum**. In each step we can go diagonally down and left or down and right.

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

Number Pyramid

- Two possible paths:





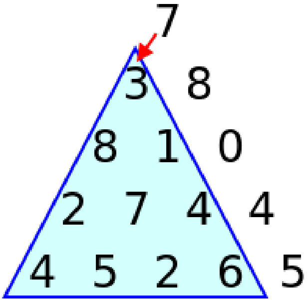
- **Constraints:** all the numbers are integers between 0 and 99 and the number of lines in the pyramid is at most 100.

Number Pyramid

- How to solve the problem?
- 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)

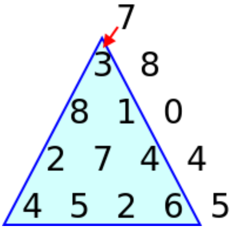
Number Pyramid

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



- In each case, we need to have in 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

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

Number Pyramid

Number Pyramid (from the recursive definition)

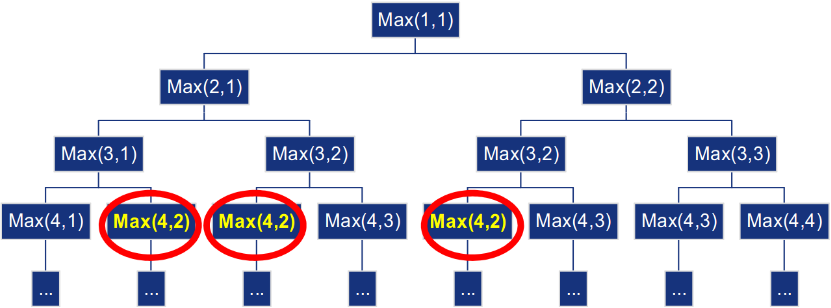
Max(i, j):
  If $i = n$ **then**
    return $P[i][j]$
  Else
    return $P[i][j]$ + **maximum** **(**Max$(i+1, j)$, Max$(i+1, j+1)$**)**

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

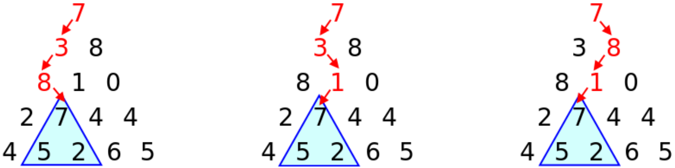
	1	**2**	**3**	**4**	**5**
1	7				
2	3	8			
3	8	1	0		
4	2	7	4	4	
5	4	5	2	6	5

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?

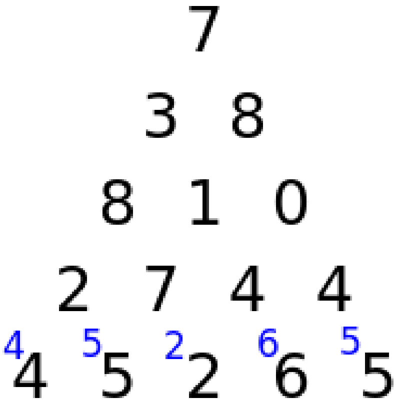
Number Pyramid

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

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

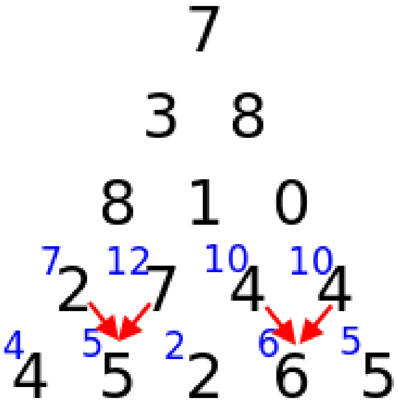
Number Pyramid

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



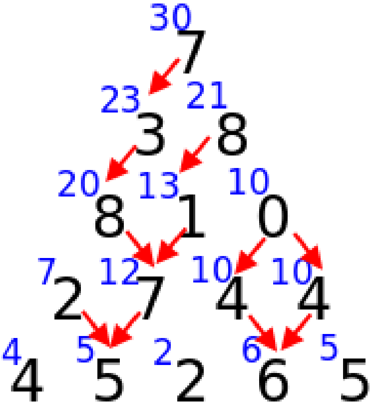
Number Pyramid

- 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]:

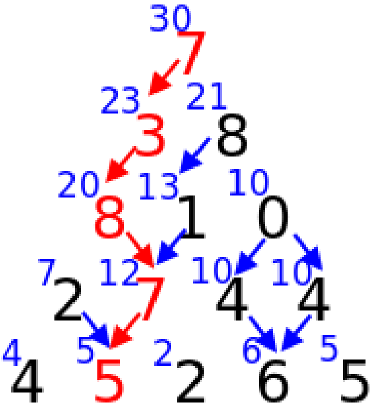
Number Pyramid (polynomial solution)

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

- With this the solution is in... P[1][1]
- Now the time needed to solve the problem only grows polynomially ($\mathcal{O}(\mathrm{n}^2)$) 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!**



To solve the number pyramid number 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**

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 number 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**.

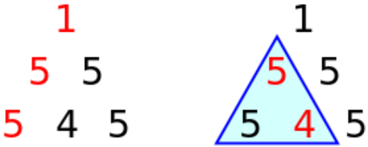
Dynamic Programming - Characteristics

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

Examplo

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
2. **Recursively define** the optimal solution, by using optimal solutions of subproblems
3. **Compute** the solutions of all subproblems: bottom-up or top-down
4. **Reconstruct** the optimal solution, based on the computed values (optional - only if necessary)

Dynamic Programming - Methodology

1) Characterize the optimal solution of the problem

- Really **understand** the problem
- Verify if an algorithm that verifies all solutions (**brute force**) is not enough
- Try to **generalize the problem** (it takes practice to understand how to correctly generalize)
- Try to **divide the problem** in subproblems of the same type
- Verify if the problem obeys the **optimality principle**
- Verify if there are **overlapping subproblems**

Dynamic Programming - Methodology

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

- **Recursively define the optimal solution value**, exactly and with rigour, from the solutions of subproblems of the same type
- Imagine that the **values of optimal solutions are already available** when we need them
- No need to code. You can just **mathematically** define the recursion

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
- Usually this **order is the inverse** to the normal order of the recursive function that solves the problem

```
                 Normal solving order
+---------+   --------------->   +-----+
|Beginning|                      | End |
+---------+   <---------------   +-----+
                 Order when using DP
```

Dynamic Programming - Methodology

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

- There is a technique, known as **"memoization"**, that 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.

Dynamic Programming - Methodology

4) Reconstruct the optimal solution, based on the computed values

- It may (or may not) be needed, given what the problem asks for
- Two alternatives:
  - **Directly** from the subproblems table
  - **New table** that stores the decisions in each step
- If we do not need to know the solution in itself, we can eventually save some space

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 **best(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... bigger or equal
    - 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
- **best(i)** - size of best sequence starting in position i

Recursive Solution for LIS Problem

$\text{best}(n) = 1$
$\text{best}(i) = 1 + \textbf{max}\{\text{best}(j)\text{: } i < j \leq n, num[j] > num[i]\}$
$\quad \text{for } 1 \leq i < n$

Longest Increasing Subsequence (LIS)

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

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

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

Compute():
  $best[n] \leftarrow 1$
  For $i \leftarrow n-1$ **to** 1 **do**
    $best[i] \leftarrow 1$
    For $j \leftarrow i+1$ **to** n **do**
      If $num[j] > num[i]$ **and** $1 + best[j] > best[i]$ **then**
        $best[i] \leftarrow 1 + best[j]$

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

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

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

Longest Increasing Subsequence (LIS)

- Let **index(i)** be the index k of the largest value $num[k]$ such that exists an increasing sequence of length i starting in that position k
- Let **index[]** be a table storing those values:

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

- At each of our n iterations we can just **binary search** on $num[index[i]]$ for the best continuation our current value

Longest Increasing Subsequence (LIS)

- Let **index(i)** be the index k of the largest value $num[k]$ such that exists an increasing sequence of length i starting in that position k
- Let **index[]** be a table storing those values:

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

- At each of our n iterations we can just **binary search** on $num[index[i]]$ for the best continuation our current value

Longest Increasing Subsequence (LIS)

- Let **index(i)** be the index k of the largest value $num[k]$ such that exists an increasing sequence of length i starting in that position k
- Let **index[]** be a table storing those values:

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

- At each of our n iterations we can just **binary search** on $num[index[i]]$ for the best continuation our current value

Longest Increasing Subsequence (LIS)

- Let **index(i)** be the index k of the largest value $num[k]$ such that exists an increasing sequence of length i starting in that position k
- Let **index[]** be a table storing those values:

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

- At each of our n iterations we can just **binary search** on $num[index[i]]$ for the best continuation our current value

Longest Increasing Subsequence (LIS)

- Let **index(i)** be the index k of the largest value $num[k]$ such that exists an increasing sequence of length i starting in that position k
- Let **index[]** be a table storing those values:

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

- At each of our n iterations we can just **binary search** on $num[index[i]]$ for the best continuation our current value

Longest Increasing Subsequence (LIS)

- Let **index(i)** be the index k of the largest value $num[k]$ such that exists an increasing sequence of length i starting in that position k
- Let **index[]** be a table storing those values:

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

- At each of our n iterations we can just **binary search** on $num[index[i]]$ for the best continuation our current value

Longest Increasing Subsequence (LIS)

- Let **index(i)** be the index k of the largest value $num[k]$ such that exists an increasing sequence of length i starting in that position k
- Let **index[]** be a table storing those values:

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

- At each of our n iterations we can just **binary search** on $num[index[i]]$ for the best continuation our current value

Longest Increasing Subsequence (LIS)

- Let **index(i)** be the index k of the largest value $num[k]$ such that exists an increasing sequence of length i starting in that position k
- Let **index[]** be a table storing those values:

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

- At each of our n iterations we can just **binary search** on $num[index[i]]$ for the best continuation our current value

Longest Increasing Subsequence (LIS)

- Let **index(i)** be the index k of the largest value $num[k]$ such that exists an increasing sequence of length i starting in that position k
- Let **index[]** be a table storing those values:

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

- At each of our n iterations we can just **binary search** on $num[index[i]]$ for the best continuation our current value

Longest Increasing Subsequence (LIS)

- Let **index(i)** be the index k of the largest value $num[k]$ such that exists an increasing sequence of length i starting in that position k
- Let **index[]** be a table storing those values:

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

- At each of our n iterations we can just **binary search** on $num[index[i]]$ for the best continuation our current value

Longest Increasing Subsequence (LIS)

- Let **index(i)** be the index k of the largest value $num[k]$ such that exists an increasing sequence of length i starting in that position k
- Let **index[]** be a table storing those values:

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

- At each of our n iterations we can just **binary search** on $num[index[i]]$ for the best continuation our current value
- Each of the n iterations takes $\log n$ (one binary search), and so our total complexity is $\mathcal{O}(n \log n)$

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 $\{1, 2, 7\}$ with a value of 35
- Choosing **max value** first will also result in $\{1, 2, 7\}$ with a value of 35
- Choosing **min weight** first will result in $\{1, 2, 5\}$ 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 **best(i)** be the best value we can get for capacity i
  - **Base case:** best(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:
    best(i) = $\mathbf{max}\ (v_j + best(i - w_j) : 1 \leq j \leq n, w_j \leq i)$ 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 **best(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 $best(i, j)$ we can now simply use the values of previously computed $best(i - 1, k)$
  - 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 of material i
- **C** - capacity of backpack
- **best(i,j)** - maximum possible value for capacity j using the first i materials

Recursive Solution for 0-1 Knapsack

$$best(0, j) = 0 \text{ for } 0 \leq j \leq C$$

$$\begin{array}{ll} best(i,j) = best(i-1,j) & \text{if } (w_i > j) \\ best(i,j) = \mathbf{max}\ \{best(i-1,j), best(i-1,j-w_i) + v_i\} & \text{if } (w_i \leq j) \\ \quad \text{for } 1 \leq i \leq n, 0 \leq j \leq C & \end{array}$$

The desired result is: **best(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	39	34	35	40

0-1 Knapsack

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

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

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	39	34	35	**40**

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 need the values of another 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 **best[]** 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**
    $best[j] \leftarrow 0$
  For $i \leftarrow 1$ **to** n **do**
    For $j \leftarrow C$ **downto** 0 **do**
      If $weight[i] \leq j$
        $best[j] \leftarrow max(best[j], best[j - weight[i]] + value[i])$

0-1 Knapsack variants

- There are many possible variants for 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
    sum(i, j) = is sum j achievable using the first i items? (T/F)

    $sum(0, 0) = T$
    $sum(0, j) = F$ for $1 \leq j \leq K$
    $sum(i, j) = sum(i-1, j)$ OR $(sum(i-1, j-s_i)$ 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 for knapsack problem.

- **Minimum number of coins**
  - Given a set S of coins and value K, discover minimum amount of coins to form quantity K (assume it is possible to form any quantity)
    There is a limited amount of any given coin value
  - Ex: $S = \{1, 10, 25\}$
    - ★ $K = 8$ has answer 4 because $10 + 10 + 10 + 10 = 4$
  - 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
  coins(i) = minimum amount of coins to form quantity i

  $coins(0) = 0$
  $coins(i) = \textbf{min}\{\ 1 + \text{coins}(i - s_j)\text{: } s_j \geq i,\ 1 \leq j \leq n\ \}$
  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 **in** 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 turn one word into the other? This metric is known as **edit distance** (ed).

Example

In order to turn "gotas" into "afoga" we need 4 transformations:

$$\text{gotas} \xrightarrow{(1)} \text{gota\_} \xrightarrow{(3)} \textbf{f}\text{ota} \xrightarrow{(3)} \text{fo}\textbf{g}\text{a} \xrightarrow{(2)} \textbf{a}\text{foga}$$

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 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*
- **ed(i,j)** - edit distance between the first *i* letters of *a* and the first *j* letters of *b*

Recursive solution for Edit Distance Problem

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

$$\begin{aligned}\text{ed}(i, j) = \mathbf{min}(&\text{ed}(i-1, j-1) + \{0 \text{ if } a[i] = b[j], 1 \text{ if } a[i] \neq b[j]\},\\ &\text{ed(i-1, j)} + 1,\\ &\text{ed(i, j-1)} + 1)\end{aligned}$$

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

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

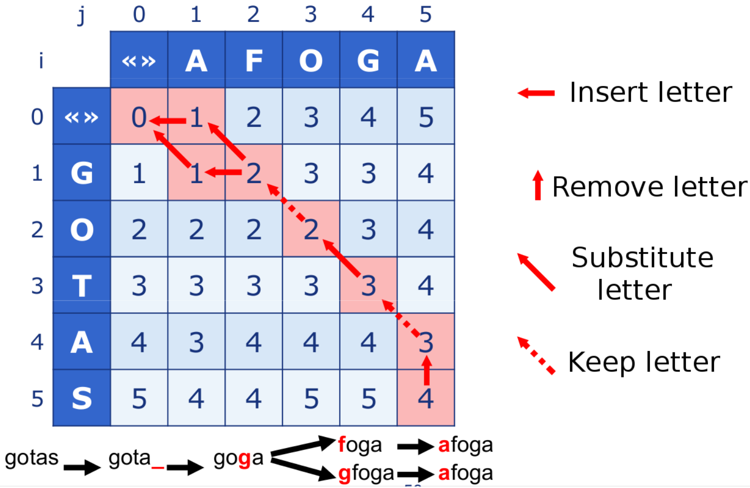
Edit Distance

- Let's see the **table** for the edit distance between "gotas" and "afoga":

j		0	1	2	3	4	5
i		«»	A	F	O	G	A
0	«»	0	1	2	3	4	5
1	G	1	1	2	3	3	4
2	O	2	2	2	2	3	4
3	T	3	3	3	3	3	4
4	A	4	3	4	4	4	3
5	S	5	4	4	5	5	4

Edit Distance

- If we needed to reconstruct the solution



i \ j		0	1	2	3	4	5
		«»	A	F	O	G	A
0	«»	0	1	2	3	4	5
1	G	1	1	2	3	3	4
2	O	2	2	2	2	3	4
3	T	3	3	3	3	3	4
4	A	4	3	4	4	4	3
5	S	5	4	4	5	5	4

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 swapping is allowed (only additions and deletions). Suppose that the strings are a and b:

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

$$\begin{array}{ll} \text{LCS}(i,j) = \text{LCS}(i-1,j-1) + 1 & \text{if } a[i] = b[j] \\ \text{LCS}(i,j) = \mathbf{max}(\text{LCS}(i-1,j), \text{LCS}(i,j-1)) & \text{if } a[i] \neq b[j] \end{array}$$

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