

---

# *Algorithm Design Strategies IV*

---

Joaquim Madeira

Version 0.5 – October 2024

# Overview

- Dynamic Programming – Recap + Questions ?
- Example – Computing Delannoy Numbers
- Homework – Computing Bernstein Polynomials
- Example – The Coin Row Problem
- The 0-1 Knapsack Problem

# DYNAMIC PROGRAMMING – RECAP

# Dynamic Programming – Recap

- General algorithm design technique
- Apply to
  - Computing recurrences – Last week – Recap
  - Solving optimization problems – Today !!
- Problem solution expressed **recursively** (top-down)
- **BUT**, proceed **bottom-up** and **store results** for later use

# Dynamic Programming – Recap

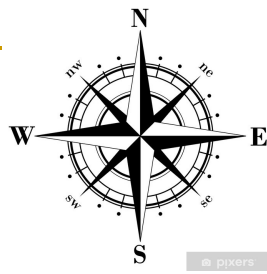
- Proceed **bottom-up** and **store results** for later use
- Big advantage, if **sub-problems overlap !!**
- NOW, there is **no need to repeatedly solve** the same sub-problems **!!**
- **Iterative algorithms** with “acceptable” **complexity** order

# Examples from last week

- Fibonacci Numbers :  $F(i)$
- Linear Robot :  $R(i)$
- Binomial Coefficients :  $C(n,p)$
  
- Any questions / problems ?
  
- Let's do another example !

# COMPUTING DELANNOY NUMBERS

# Delannoy Numbers – $D(i,j)$



- Rectangular grid of size  $(m,n)$
- Start at SW corner :  $(0,0)$
- Steps allowed in **N**, **E** or **NE** directions
- **$D(i,j)$**  = number of different paths from  $(0,0)$  to  $(i,j)$ 
  - Recursive definition ?
  - Trivial cases ?

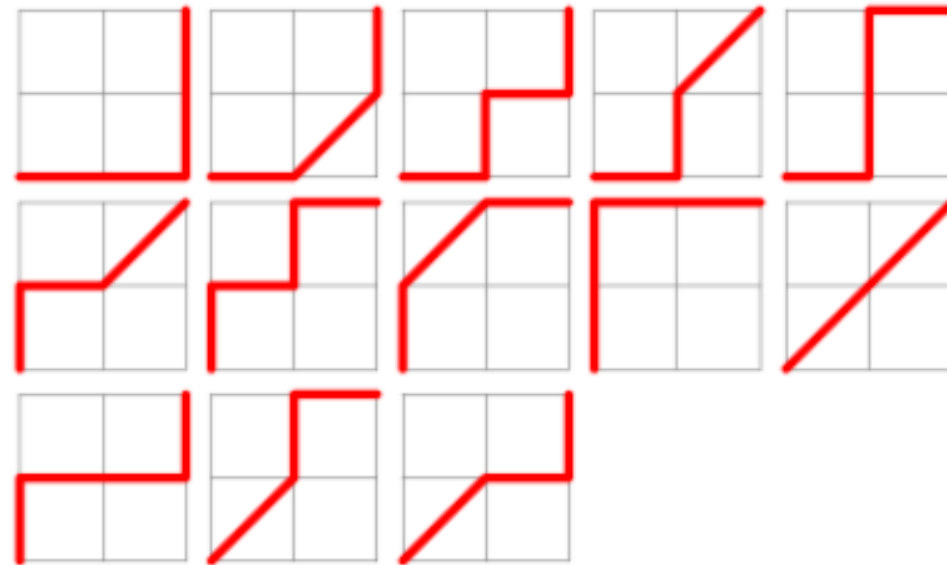


# $D(n,n)$ – Central Delannoy Numbers

■  $D(1,1)$



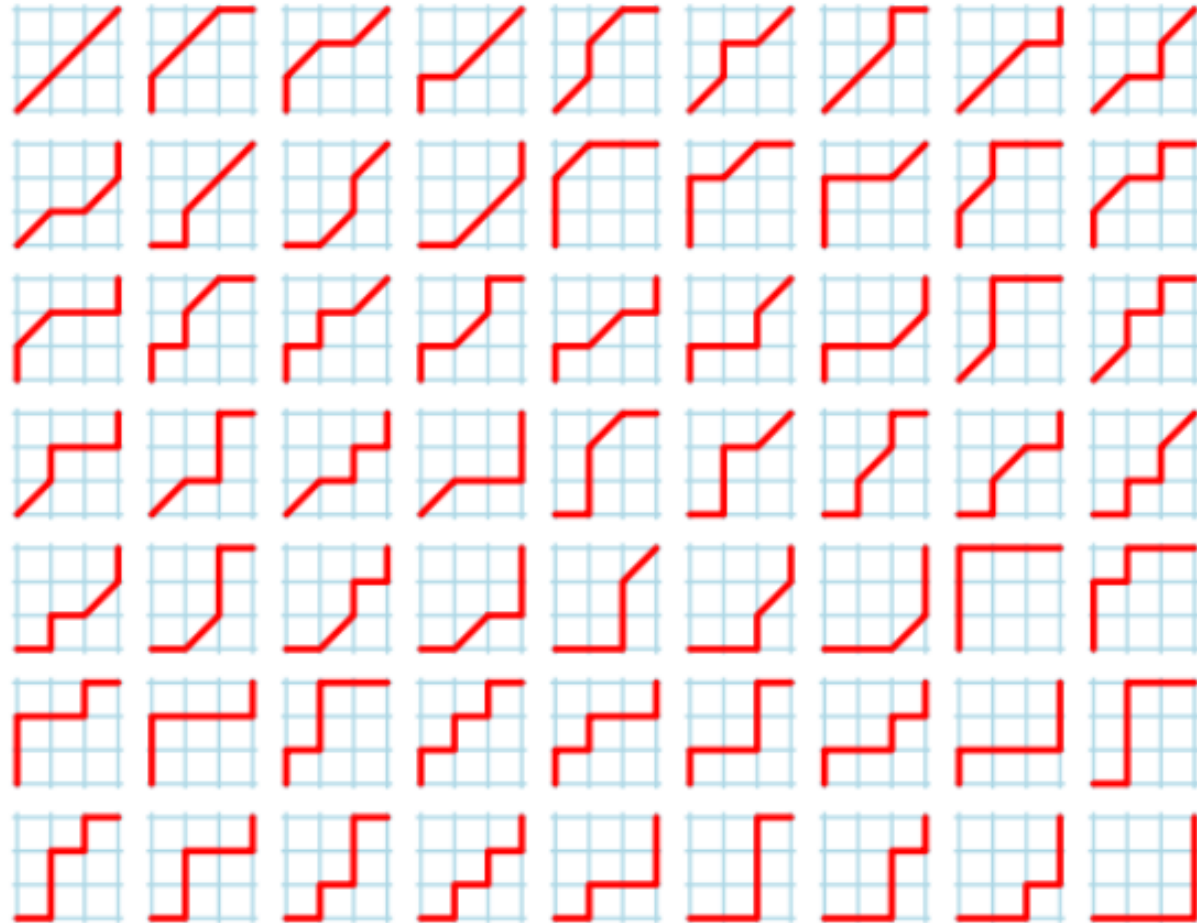
■  $D(2,2)$



[Mathworld]

# $D(n,n)$ – Central Delannoy Numbers

■  $D(3,3)$



[Wikipedia]

# Delannoy Numbers

$$D(m,n) = 1, \text{ if } m = 0 \text{ or } n = 0$$

$$\begin{aligned} D(0,0) &= 1 \\ D(0,j) &= 1 \\ D(i,0) &= 1 \end{aligned}$$

$$D(m,n) = D(m-1, n) + D(m-1, n-1) + D(m, n-1)$$

- $D(1,1) = ?$
- $D(2,2) = ?$
- $D(2,3) = ?$
- $D(3,2) = ?$
- Arrange the calculations in a **geometrical representation**

# TASKS

# Tasks – $V1 + V2 + V3 + V4$

- **Implement** four functions for computing  $D(i,j)$ 
  - $V1$  : using **recursion**
  - $V2$  : using a 2D **array**
  - $V3$  : using **two** 1D **arrays**
  - $V4$  : using **memoization**
- **Count** the number of **additions** carried out
- **Tables** ?
- How fast does  $D(n,n)$  grow ?

# Delannoy Numbers

Delannoy's Matrix - Recursive Function

1	1	1	1	1	1	1	1	1	1	1
1	3	5	7	9	11	13	15	17	19	21
1	5	13	25	41	61	85	113	145	181	221
1	7	25	63	129	231	377	575	833	1159	1561
1	9	41	129	321	681	1289	2241	3649	5641	8361
1	11	61	231	681	1683	3653	7183	13073	22363	36365
1	13	85	377	1289	3653	8989	19825	40081	75517	134245
1	15	113	575	2241	7183	19825	48639	108545	224143	433905
1	17	145	833	3649	13073	40081	108545	265729	598417	1256465
1	19	181	1159	5641	22363	75517	224143	598417	1462563	3317445
1	21	221	1561	8361	36365	134245	433905	1256465	3317445	8097453

# COMPUTING BERNSTEIN POLYNOMIALS

# Computing Bernstein Polynomials

$$B_{0,0}(t) = 1$$

Para a esquerda multiplicas 1-t  
Para a direita multiplicas por t  
E somas

$$B_{n,0}(t) = (1 - t) B_{n-1,0}(t) ; t \text{ in } [0,1]$$

$$B_{n,n}(t) = t B_{n-1,n-1}(t) ; t \text{ in } [0,1]$$

$$B_{n,j}(t) = (1 - t) B_{n-1,j}(t) + t B_{n-1,j-1}(t) ; j = 1, 2, \dots, n - 1 ; t \text{ in } [0,1]$$

- There are **(n + 1) polynomials of degree n**
- How to obtain the **expression** of such a polynomial ?
- Arrange the calculations in a **triangular representation** !
  - Have you seen a similar triangle before ?



# Computing Bernstein Polynomials

- How to compute the value of a polynomial for a given  $t^*$  ?
- **V1** : Compute  $B_{n,j}(t^*)$  recursively
- $B_{3,2}(1/2) = ?$
- Number of **recursive calls** ?
- Are there **overlapping sub-problems** ?

# Computing Bernstein Polynomials

- **V2** : Compute  $B_{n,j}(t^*)$  using a 2D array
- $B_{3,2}(1/2) = ?$
- How to ?
- Have you seen a similar procedure before ?
- Can we use **less** memory space ?

# TASKS

# Tasks – $V1 + V2 + V3 + V4$

- **Implement** functions for computing  $B_{n,j}(t)$ 
  - **V1** : using recursion
  - **V2** : using a 2D **array**
  - **V3** : using a 1D **array**
  - **V4** : using **memoization**
- **Count** the number of **multiplications** carried out
- **Tables** ?
- **Complexity order** ?

# Bernstein Polynomials for $t = 0.5$

Polynomials' Triangle - Recursive Function -  $t = 0.5$

1.000								
0.500	0.500							
0.250	0.500	0.250						
0.125	0.375	0.375	0.125					
0.062	0.250	0.375	0.250	0.062				
0.031	0.156	0.312	0.312	0.156	0.031			
0.016	0.094	0.234	0.312	0.234	0.094	0.016		
0.008	0.055	0.164	0.273	0.273	0.164	0.055	0.008	
0.004	0.031	0.109	0.219	0.273	0.219	0.109	0.031	0.004

# Multiplications count – Recursive

Number of Multiplications - Recursive Function

0									
1	1								
2	4	2							
3	8	8	3						
4	13	18	13	4					
5	19	33	33	19	5				
6	26	54	68	54	26	6			
7	34	82	124	124	82	34	7		
8	43	118	208	250	208	118	43	8	

# Multiplications count – Dynamic Prog.

Number of Multiplications - Dynamic Programming - V. 2

0									
2	2								
6	6	6							
12	12	12	12						
20	20	20	20	20					
30	30	30	30	30	30				
42	42	42	42	42	42	42			
56	56	56	56	56	56	56	56		
72	72	72	72	72	72	72	72	72	72

# Multiplications count – Memoization

Number of Multiplications - Memoized Function

0									
1	1								
1	2	1							
1	2	2	1						
1	2	2	2	1					
1	2	2	2	2	1				
1	2	2	2	2	2	1			
1	2	2	2	2	2	2	1		
1	2	2	2	2	2	2	2	1	
1	2	2	2	2	2	2	2	2	1



---

# THE COIN ROW PROBLEM

# The Coin Row Problem

- Row of  $n$  coins
- Integer values  $c_1, c_2, \dots, c_n$ 
  - Not necessarily distinct
- **Goal:** Pick up the **maximum** amount of money
- **Restriction:** No two adjacent coins can be picked up

AQUELE QUE ELE FEZ NA AULA EM CIMA DO  
COMPUTADOR DO OUTRO

# The Coin Row Problem

- Can we solve it by Exhaustive Search ?
- Or using heuristics ?
- Or generating random candidate solutions ?
- How ?
- Efficiency ?

# Exhaustive Search

## ■ How to proceed?

- Generate the  $2^n$  subsets of a set of  $n$  coins
- Filter out the coin subsets with adjacent coins
- For each such subset, compute its total value
- And keep the most valuable feasible coin subset

# A Simple Heuristic

Choose a coin  
Remove Adjacents  
Repeat until there are no coins left

## ■ How to proceed?

- ❑ Build a solution **step-be-step**
- ❑ While it is possible :
- ❑ Pick the **largest valued coin** that is **not adjacent** to a previously chosen coin
- ❑ Do we always get the **optimal solution** ?
- ❑ How do we solve “**ties**”, i.e., more than one choseable **coin** of the **same value** ?
  - Smallest index (stupid)
  - Choose the one with smaller adjacent value

# TASKS

# Tasks – V1 + V2

- **Implement** functions for solving the coin row problem
  - **V1** : using **exhaustive search**
  - **V2** : using the **simple heuristic**
- **Computational cost ?**
- **Generate random problem instances**
- **Compare the obtained solutions**

# THE COIN ROW PROBLEM - DYNAMIC PROGRAMMING



# The Coin Row Problem

- How to derive a **recurrence** ?
- **$F(n) = ?$** 
  - **Maximum amount** that can be picked up from the row of  **$n$  coins**
- **$n^{\text{th}}$  coin was picked up / not picked up ?**
- **Trivial cases ?**

# The Coin Row Problem

- $F(0) = 0$

- $F(1) = C_1$

- $$F(n) = \max \left\{ \begin{array}{l} c_n + F(n - 2), \\ F(n - 1) \end{array} \right. \quad \text{(escolher última coin)}, \quad \text{for } n > 1$$

n ( n coins )  
n - index of last coin

- Example: 5, 1, 2, 10, 6, 2

- $F(6) = ? \max(c_6 + F(4), F(5))$

## Example 2 for Coin-Row Problem

Solve for the coin-row problem for the instance {7, 3, 9, 10, 8, 6}

Index	0	1	2	3	4	5	6
C		7	3	9	10	8	6
F	0	7	7	16	17	24	24
History		C1	-	C3	C4	C5	-
		-	F[1]	F[1]	F[2]	F[3]	F[5]

F[0]	0			
F[1]	C1 = 7			
F[2]	Max( C2 + F[0], F[1] ) = Max( 3 + 0, 7 ) = 7			
F[3]	Max( C3 + F[1], F[2] ) = Max( 9 + 7, 7 ) = 16			
F[4]	Max( C4 + F[2], F[3] ) = Max( 10 + 7, 16 ) = 17			
F[5]	Max( C5 + F[3], F[4] ) = Max( 8 + 16, 17 ) = 24			
F[6]	Max( C6 + F[4], F[5] ) = Max( 6 + 17, 24 ) = 24			

**Dynamic Programming**

Coins: C1, C3, C5; Value = 24

**Greedy**

C4, C1, C6 (in the dec. order of their values and non-overlap with neighboring coins); Value = 23

Index	0	1	2	3	4	5	6
C		7	3	9	10	8	6
F	0	7	7	16	17	24	24
History		C1	-	C3	C4	C5	-
		-	F[1]	F[1]	F[2]	F[3]	F[5]

# The Coin Row Problem

- The DP algorithm solves the problem for the **first  $i$  coins** in the row,  $1 \leq i \leq n$ 
  - We get the **optimal solution** for every **sub-problem**
- How to **find the coins** of an optimal solution ?
  - **Trace back** the computations
  - **OR** use an **additional array** to record which term was **larger** at every step

# TASKS

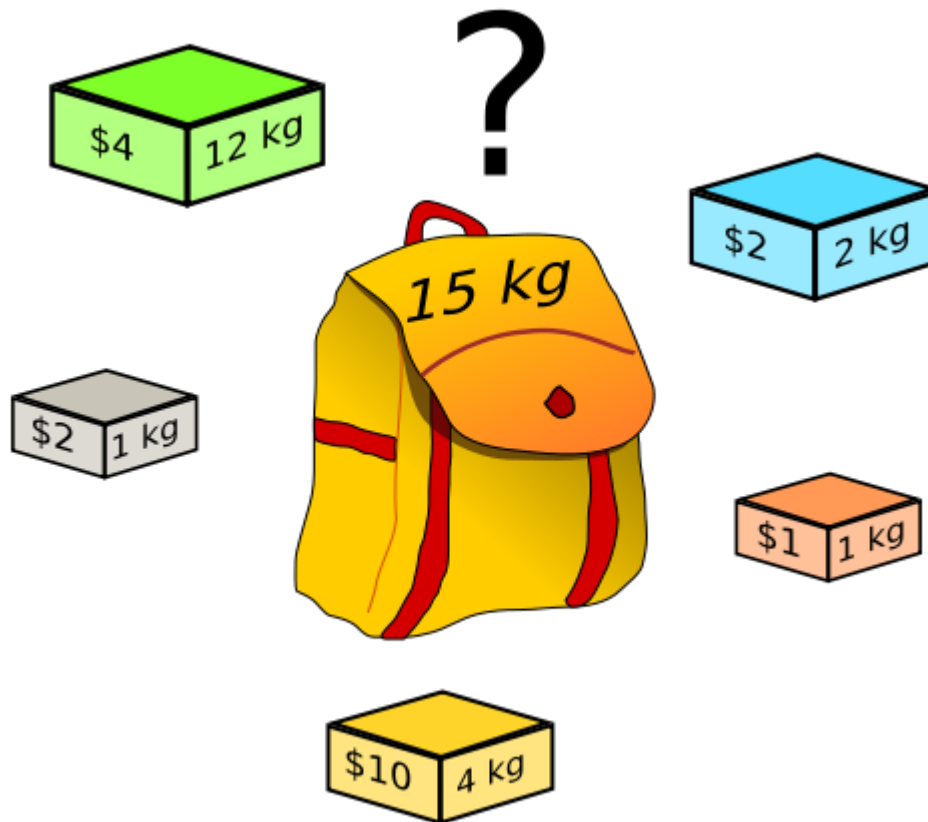
# Tasks – V1 + V2 + V3

- **Implement** functions for computing  $F(n)$ 
  - **V1** : using recursion
  - **V2** : using a 1D **array**
  - **V3** : using an **extra array** to identify the optimal set of coins
- **Count** the number of **comparisons** carried out
- **Tables ?**
- **Complexity order ?**

# THE 0-1 KNAPSACK PROBLEM

# The 0-1 Knapsack Problem

- Find the **most valuable subset** of items, that fit into the knapsack



[Wikipedia]

# The 0-1 Knapsack Problem

- Given  $n$  items
  - Known **weight**  $w_1, w_2, \dots, w_n$
  - Known **value**  $v_1, v_2, \dots, v_n$
- A knapsack of **capacity**  $W$
- Which one is the **most valuable subset** of items that fit into the knapsack ?
  - **More than one** solution ?



# The 0-1 Knapsack Problem

- How to formulate ?

$$\text{max } \sum x_i v_i$$

$$\text{subject to } \sum x_i w_i \leq W$$

$$\text{with } x_i \text{ in } \{0, 1\}$$

# EXHAUSTIVE SEARCH

# Exhaustive Search

- Brute-force approach to **combinatorial problems**
  - I.e., there is a discrete set of feasible solutions
- Strategy
  - Enumerate **all** possible solution **candidates**
  - Check if they **satisfy** the problem's statement (weight menor que upper bound)
  - If needed, **choose one solution** from the set of feasible ones
- How to ensure that we **check all candidates** ?

# Exhaustive Search

## ■ Basic algorithm

```
c ← generate a first candidate solution
while ( c is a candidate ) do
  if ( c is a valid solution )
    then output (c)
  c ← generate the next candidate solution, if any
```

Apart from stopping at the end

## ■ Might also stop after

- ❑ Finding the first valid solution
- ❑ Finding a specified number of valid solutions
- ❑ Testing a given number of candidates
- ❑ Spending a given amount of CPU time

# Exhaustive Search

## ■ Features

- Often simple to implement
- It will **always** find a solution, if there is one (?!?)

## ■ BUT, **cost** proportional to the number of candidate solutions

- **Combinatorial explosion !**
- Practical only for very small problem instances !!

## ■ How to speed up the search ?

# Speeding up Brute-Force Searches

- **Reduce** the search space
  - Use analysis / heuristics to reduce the number of candidate solutions
- **Reorder** the search space
  - Useful whenever searching just for one solution
  - Expected running time depends on the order candidates are tested
  - Test **the most promising** candidates **first** !!

---

# THE 0-1 KNAPSACK PROBLEM - EXHAUSTIVE SEARCH

# The 0-1 Knapsack Problem

## ■ How to proceed?

- ❑ Generate the  $2^n$  subsets of a set of  $n$  items
- ❑ For each such subset, compute its total weight
  - Feasible subset ? (Not larger than weight max)
- ❑ And keep the most valuable feasible subset



# The 0-1 Knapsack Problem

- Knapsack of capacity  $W = 10$
- 4 items
  - Item 1 :  $w = 7$  ;  $v = \$42$
  - Item 2 :  $w = 3$  ;  $v = \$12$
  - Item 3 :  $w = 4$  ;  $v = \$40$
  - Item 4 :  $w = 5$  ;  $v = \$25$
- Optimal solution ?

# The 0-1 Knapsack Problem

## ■ Questions

- ❑ How to generate all subsets ?
- ❑ Does order matter ?

## ■ Efficiency

- ❑  $O(2^n)$  ( $2^n$  subsets)
- ❑ Exhaustive search can only be applied to **small problem instances !!**
- ❑ Alternatives ?

---

# TASKS

# Tasks – V1 + V2

- Implement **two functions** for computing the solution to an instance of the Knapsack problem
- **V1** : using a **binary counter** to generate the successive subsets
- **V2** : using Python's **combinatoric generator**  
`combinations( )`

# Tasks – V1 + V2

- How to analyze ?
- Register the number of **subsets generated** and the **execution times** for some test instances
- What happens if we consider
  - 1 more item / 2 more items / ...
- **Extrapolate** the **execution time** for much larger problem instances

- You can use the example instance at

[https://www.rosettacode.org/wiki/Knapsack\\_problem/0-1](https://www.rosettacode.org/wiki/Knapsack_problem/0-1)

- And check the different implementations in Python and other prog. languages

# THE 0-1 KNAPSACK PROBLEM - HEURISTICS

# The 0-1 Knapsack Problem

- An **alternative** to exhaustive search is to use **simple heuristics**
  - **Rule** to construct a **feasible solution step-by-step**
- Sometimes, only an **approximate solution** is found
- How to solve “**ties**” ?
- I.e., whenever there is **more than one equivalent choice**  
stupid - by smaller index  
not stupid - by another heuristic



# The 0-1 Knapsack Problem

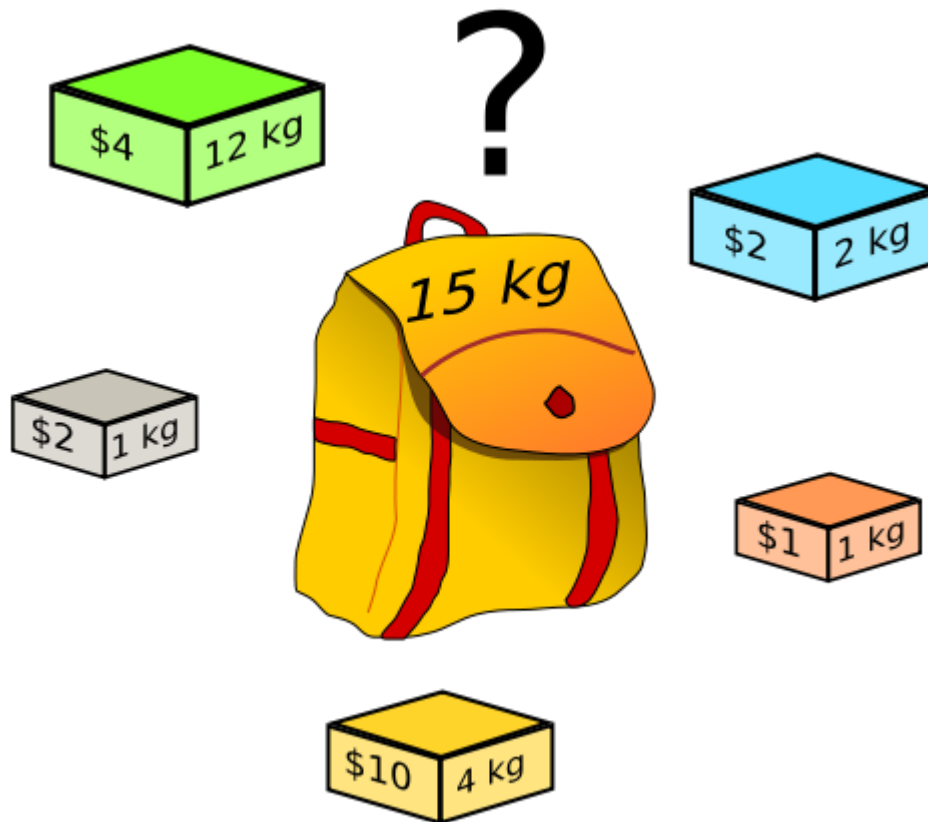
- One very simple idea :
- Successively choose the most valuable item that still fits into the knapsack
- Apply it to the examples
  - Do you get the optimal solution ?
- Solve “ties” by choosing the item with same value but with less weight

# The 0-1 Knapsack Problem

- Another simple idea :
- Successively choose the item with less weight that still fits into the knapsack
- Apply it to the examples
  - Do you get the optimal solution ?
- Solve “ties” by choosing the item with same weight but with largest weight

# The 0-1 Knapsack Problem

- Find the **most valuable subset** of items, that fit into the knapsack



[Wikipedia]

# The 0-1 Knapsack Problem

- Knapsack of capacity  $W = 10$
- 4 items
  - Item 1 :  $w = 7$  ;  $v = \$42$
  - Item 2 :  $w = 3$  ;  $v = \$12$
  - Item 3 :  $w = 4$  ;  $v = \$40$
  - Item 4 :  $w = 5$  ;  $v = \$25$
- Optimal solution ?

# THE 0-1 KNAPSACK PROBLEM - DYNAMIC PROGRAMMING

# Optimization Problems

## ■ Goal

- Minimize or maximize an **objective function**
- Store the **solution's components**

## ■ When can we use **dynamic programming** ?

- Overlapping sub-problems
- **Optimal substructure**

### ■ The principle of optimality

Principle of Optimality.

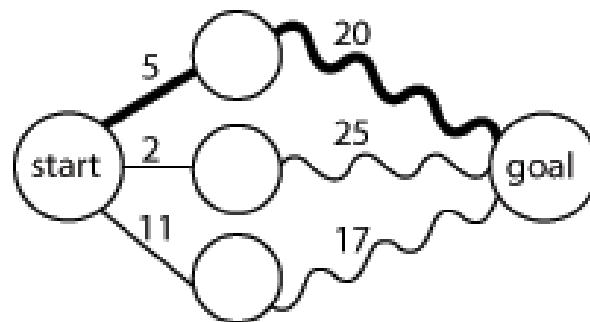
Definition: A problem is said to satisfy the Principle of Optimality if the subsolutions of an optimal solution of the problem are themselves optimal solutions for their subproblems.

# The Principle of Optimality

- Does an optimization problem satisfy the **principle of optimality** ?
- An optimal solution to any of its instances must be made up of **optimal solutions** to its **sub-instances**.

- Example

- ☐ Shortest path



[Wikipedia]

# The 0-1 Knapsack Problem

- Is that the case for the Knapsack Problem ?
- YES !!
- We have solved it by exhaustive search...
- Now, we can solve it using Dynamic Programming !!
- Recurrence ?



# The 0-1 Knapsack Problem

- Given  $n$  items
  - Known **weights**  $w_1, w_2, \dots, w_n$
  - Known **values**  $v_1, v_2, \dots, v_n$
- A knapsack of **capacity**  $W$
- Which one is **the / a** most valuable subset of items that fit into the knapsack?

# The 0-1 Knapsack Problem

- Particular instance  $(i, j)$ 
  - The **first  $i$  items** ( $1 \leq i \leq n$ )
    - Weights  $w_1, w_2, \dots, w_i$
    - Values  $v_1, v_2, \dots, v_i$
  - Knapsack **capacity  $j$**  ( $1 \leq j \leq W$ )
- Value of an optimal solution to instance  $(i, j)$  ?
  - **$V[i, j]$**  = ? Solução ótima para primeiros  $i$  itens e knapsack de capacidade  $j$

# The 0-1 Knapsack Problem

- Goal :  $V[ n, W ] = ?$
- Recurrence ?
- Trivial cases
  - $V[ 0, j ] = 0$  , for all  $j \geq 0$
  - $V[ i, 0 ] = 0$  , for all  $i \geq 0$

# The 0-1 Knapsack Problem

- General cases:
- The **ith item does not fit** into the knapsack
  - $V[i, j] = V[i - 1, j]$  , if  $j - w_i < 0$
- The **ith item fits** into the knapsack
  - $V[i, j] = \max \{ V[i - 1, j] , v_i + V[i - 1, j - w_i] \}$  ,  
if  $j - w_i \geq 0$



The **ith item does not fit** into the knapsack

# The 0-1 Knapsack ]

□  $V[i, j] = V[i-1, j]$ , if  $j - w_i < 0$

The **ith item fits** into the knapsack

□  $V[i, j] = \max \{ V[i-1, j], v_i + V[i-1, j - w_i] \}$ ,  
if  $j - w_i \geq 0$

- To determine  $V[i, j]$ , if  $(j - w_i) \geq 0$  inspect
  - Element in the same column and previous row 
  - Element in column  $(j - w_i)$  and previous row 
- How to proceed ?
  - Fill the table **row by row** or **column by column**
- Implement an iterative function !!

# The 0-1 Knapsack Problem

## ■ Example

□ Capacity  $W = 10$

□ 4 items

- Item 1 :  $w = 7$  ;  $v = \$42$
- Item 2 :  $w = 3$  ;  $v = \$12$
- Item 3 :  $w = 4$  ;  $v = \$40$
- Item 4 :  $w = 5$  ;  $v = \$25$

## ■ Optimal solution

□ Value ?

□ Which items ?

- Trace back the computations !!

Let,  $\text{weight}[] = \{1, 2, 3\}$ ,  $\text{profit}[] = \{10, 15, 40\}$ , Capacity = 6

When  $j\text{thWeight} = 3$ , the maximum possible profit is  $\max(\text{DP}[2][3], 40 + \text{DP}[2][3-3]) = \max(25, 40) = 40$ .

When  $j\text{thWeight} = 4$ , the maximum possible profit is  $\max(\text{DP}[2][4], 40 + \text{DP}[2][4-3]) = \max(25, 50) = 50$ .

When  $j\text{thWeight} = 5$ , the maximum possible profit is  $\max(\text{DP}[2][5], 40 + \text{DP}[2][5-3]) = \max(25, 55) = 55$ .

When  $j\text{thWeight} = 6$ , the maximum possible profit is  $\max(\text{DP}[2][6], 40 + \text{DP}[2][6-3]) = \max(25, 65) = 65$ .

weight→ item i/	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	10	10	10	10	10	10
2	0	10	15	25	25	25	25
3	0	10	15	40	50	55	65

# The 0-1 Knapsack Problem

## ■ Complexity ?

- $O(n W)$

- Pseudo-Polynomial !!

In [computational complexity theory](#), a numeric algorithm runs in **pseudo-polynomial time** if its [running time](#) is a [polynomial](#) in the *numeric value* of the input (the largest integer present in the input)—but not necessarily in the *length* of the input (the number of bits required to represent it), which is the case for [polynomial time](#) algorithms.<sup>[1]</sup>

## ■ It depends on the **magnitude of $W$** !!

- Not just on the number of items

- It will take much time for very large values of  $W$  !!

## ■ What happens, if $W$ increases and we need an **additional bit** to represent its value ?

# The 0-1 Knapsack Problem

- **BUT**, it is a **NP-Complete** problem !!

- Exhaustive search is **exponential**
- Is there a contradiction ?

**NP**  
Class of computational decision problems for which any given *yes*-solution can be verified as a solution in polynomial time by a deterministic Turing machine (or solvable by a *non-deterministic* Turing machine in polynomial time).

**NP-hard**  
Class of problems which are at least as hard as the hardest problems in NP. Problems that are NP-hard do not have to be elements of NP; indeed, they may not even be decidable.

**NP-complete**  
Class of decision problems which contains the hardest problems in NP. Each NP-complete problem has to be in NP.

**NP-easy**  
At most as hard as NP, but not necessarily in NP.

**NP-equivalent**  
Decision problems that are both NP-hard and NP-easy, but not necessarily in NP.

**NP-intermediate**  
If P and NP are different, then there exist decision problems in the region of NP that fall between P and the NP-complete problems. (If P and NP are the same class, then NP-intermediate problems do not exist because in this case every NP-complete problem would fall in P, and by definition, every problem in NP can be reduced to an NP-complete problem.)

- **Number of bits** needed to represent  $W$  ? ( $\log_2 + 1$ )

- $O(\log W)$

- Complexity in terms of that number of bits ?

- $O(2^{\log W})$
- **Exponential !!**



# The 0-1 Knapsack Problem

- Could it be different ?
  - What would that entail ?
- The dynamic programming algorithm serves our purposes !!
  - Except for “exponentially large” values of  $W$

# TASKS

# Tasks – V3 + V4

- Implement **two functions** for computing the solution to an instance of the Knapsack problem
- **V3** : a **recursive function** using the recurrence defined for the DP approach
- **V4** : an **iterative function** implementing the DP algorithm
  - How to identify items belonging to the solution ?

# Tasks – V3 + V4

- How to analyze ?
- Register **execution times** for some test instances
- What happens if we consider
  - 1 more item / 2 more items / ...
  - **twice** the number of items ?
- **Extrapolate** the **execution time** for much larger problem instances

# Solution – Dynamic Programming

## 0-1-Knapsack - Dynamic Programming Solution

Item Values: [None, 42, 12, 40, 25]

Item Weights: [None, 7, 3, 4, 5]

Capacity: W = 0	Optimal value: V = 0	Items = []
Capacity: W = 1	Optimal value: V = 0	Items = []
Capacity: W = 2	Optimal value: V = 0	Items = []
Capacity: W = 3	Optimal value: V = 12	Items = [2]
Capacity: W = 4	Optimal value: V = 40	Items = [3]
Capacity: W = 5	Optimal value: V = 40	Items = [3]
Capacity: W = 6	Optimal value: V = 40	Items = [3]
Capacity: W = 7	Optimal value: V = 52	Items = [2, 3]
Capacity: W = 8	Optimal value: V = 52	Items = [2, 3]
Capacity: W = 9	Optimal value: V = 65	Items = [3, 4]
Capacity: W = 10	Optimal value: V = 65	Items = [3, 4]
Capacity: W = 11	Optimal value: V = 82	Items = [1, 3]
Capacity: W = 12	Optimal value: V = 82	Items = [1, 3]

---

# REFERENCES

# References

- A. Levitin, *Introduction to the Design and Analysis of Algorithms*, 3<sup>rd</sup> Ed., Pearson, 2012
  - Chapter 3 + Chapter 8
- R. Johnsonbaugh and M. Schaefer, *Algorithms*, Pearson Prentice Hall, 2004
  - Chapter 8
- T. H. Cormen et al., *Introduction to Algorithms*, 3<sup>rd</sup> Ed., MIT Press, 2009
  - Chapter 15