



Universität  
Zürich <sup>UZH</sup>

Institut für Informatik

# Informatics II

## Tutorial Session 12

Wednesday, 18<sup>th</sup> of May 2022

Discussion of Exercise 11,  
Dynamic Programming

14.00 – 15.45

BIN 0.B.06



## Agenda

- Exam Preview
- Review of Exercise 10
  - Dynamic Programming



**Universität  
Zürich** <sup>UZH</sup>

**Institut für Informatik**

# Outlook on Final Exam

- General Information
- How to Hand in the Exam
- Exam Strategy
- Remarks on Auxiliary Materials



## Final Exam: General Information

- *Venue:* online at <https://fs3.epis.uzh.ch>
- *Modus:* download PDF with tasks and upload PDF with solutions
- *Date and time:* **Wednesday, June 1<sup>st</sup>, 14.00 h**
- *Available time:* **110 minutes**
- *Number of questions, maximum achievable points:* **tba**
- *Supervision:* **no supervision** during the examination
- *Topics:* **tba**
- *Auxiliary materials:* open-Book, **all learning materials can be used**
- *Support during exam:* via **MS Teams**
- *Testrun:* **Wednesday, May 25<sup>th</sup>, 07.00 to 19.00 h**



## How to Hand In the Exam

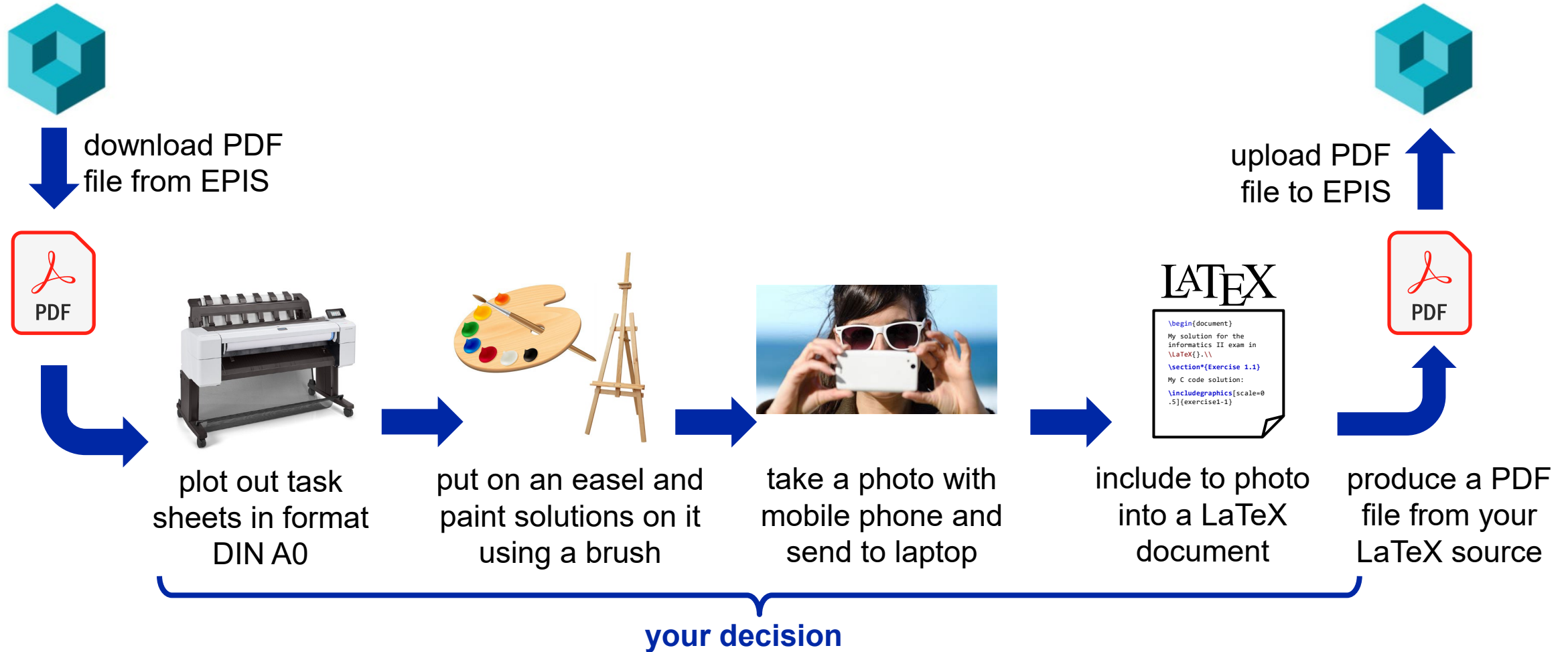
Any way to solve and hand in your solution is fine as long as the solution is

- clearly discernible / readable,
- clearly attributable to a task from the exam
- in a PDF file
- handed in within the time frame of the exam via EPIS

How you do this, is up to you (and your responsibility).

Also, note that once you have clicked on the «final submission» / «endgültige Abgabe» button, you cannot go back and change your submission.

## Modes for Solving the Exam



## Modes for Solving the Exam

Some of the possible modes to answer the exam include:

- 1) print exam sheet – write by hand on exam sheet (below question) – scan exam sheet
- 2) write by hand on a white paper (indicate task number from exam) – scan white paper
- 3) annotate the PDF exam sheet digitally on a tablet computer – save and upload
- 4) annotate the PDF exam sheet digitally with the keyboard – save and upload
- 5) write your solution with the keyboard into a text editor / MS Word file / Latex file / ... – save as PDF
- 6) ...

Whatever method you decide to use: **Absolutely, positively try it out at least once before the exam!**

And: Based on your trials, **reserve a sufficient amount of time for upload!** Murphy's law applies.

(If your method involves scanning: a resolution of 300 dpi should be sufficient and results in much smaller files which in turn results in less time / risk for uploading.)

## Exam Strategy



- Be ready, e.g. have auxiliary materials ready, stable internet connection (with backup if possible, e.g. personal hotspot), suitable working space, ...
- Start with tasks which you feel comfortable to solve. Do not get lost in a coding task (or any other task, switch to other tasks if you do). Do *not* strive for perfection in the exam (e.g. trying to cover each and every edge case).
- Make sure, you always at least **write something**. Note down observations and thoughts for example.
- If you struggle with formal notation, **use natural language** additionally (or alternatively).
- Maybe have some templates / code skeletons ready of frequently used algorithms.
- Never give up.





## Remarks on Auxiliary Materials and the Open Book Setting

Auxiliary materials can be a **nice help**. Notes save you from learning stupid things by heart. They may protect from «blackout situations» where you suddenly can't remember things that you actually know.

*Although:*

- Auxiliary materials are **useless** and can't save you, **if you don't understand** the content and **did not practice** the application of the content.
- Preparing auxiliary materials may be a good preparation and repetition. I suggest, though, to **spend most of the time with solving example tasks** (e.g. old midterms and assignments) which is far more important and rewarding in my opinion.



# Dynamic Programming

- Introduction: What is Dynamic Programming
- Motivation: Fibonacci Numbers
- Memoization, Top-Down / Bottom-Up Approach
- Optimal Substructure
- Dynamic Programming and Decision Trees
- Example: 0/1 Knapsack Problem
- Example: Rod Cutting Problem
- Example: Levenshtein Distance



Richard Ernest Bellman (1920 – 1984), applied mathematician, who introduced dynamic programming in 1953



Charles Erwin Wilson (1890 – 1961), Secretary of Defense of the USA



RAND Corporation is an US think tank established in 1948 to offer research and analysis to the United States Armed Forces. It is financed by the U.S. government and other sources

Further reading: <https://www.jstor.org/stable/3088448?seq=1>

## What is Dynamic Programming?

Dynamic programming is a general, powerful [algorithms design technique](#). It can be applied in all sorts of different situations. Originally it was developed to solve problems involving multi-stage decision processes (and this heritage can still be seen in many applications).

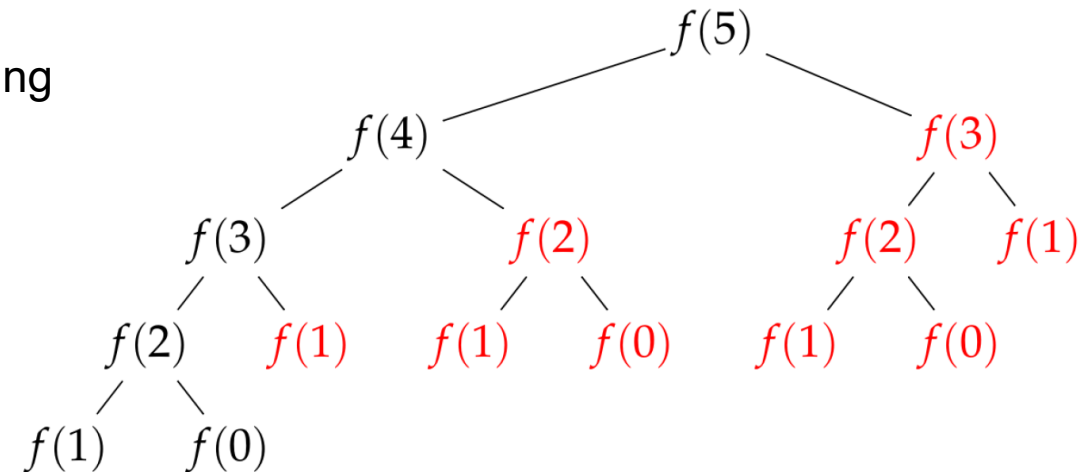
### Typical characteristics:

- The [sub-tasks overlap](#) (therefore, a naïve algorithm solves the same sub-tasks over and over again).
  - This is a difference to divide-and-conquer algorithms where sub-tasks are independent of each other and do not overlap.
- [Reuse solutions](#) already calculated (to avoid calculating the same thing multiple times).
  - This may improve efficiency if the total number of *distinct* sub-tasks is of polynomial order while the naïve algorithm will require an exponentially growing number of steps.
- Search for a somehow [optimal solution](#) by combining solutions to sub-tasks.

## Dynamic Programming: Motivation, Memoization

The Fibonacci numbers can be calculated recursively using the following algorithm:

```
int fib_rec(int n) {  
    if (n < 2) { return n; }  
    return fib_rec(n-1) + fib_rec(n-2);  
}
```



As we can see from the recursion tree, this approach calculates the same values multiple times which is very inefficient (overlapping sub-problems which are solved multiple times). Also, the number of calls grows exponentially with  $n$ . Examining the recurrence relation for the above recursive solution yields the same result:  $T(n) = T(n - 1) + T(n - 2) + \Theta(1)$  which is in  $O(2^n)$  which is *very bad*.

*Idea:* Store intermediate results and reuse them instead of calculating the same thing over and over again. This storing of previously calculated values is called **memoization**.


## Dynamic Programming: Motivation, Memoization

To implement the idea of storing the intermediate results and reusing them, we use an array as a lookup table for this. This solution is in  $O(n)$  which is a lot better than  $O(2^n)$ .

```
#define EMPTY -1 /* dummy value to indicate empty cells */
#define MAX 1000
int lookup[MAX]; /* has to be initialized before usage (not shown here) */

int fib_top_down(int n) {
    if (lookup[n] == EMPTY) {
        if (n <= 1) {
            lookup[n] = n;
        }
        else {
            lookup[n] = fib_top_down(n-1) + fib_top_down(n-2);
        }
    }
    return lookup[n];
}
```

$n$	1	2	3	4	5	6	7	8
$F_n$	1	1	2	3	5	8	13	21



## Dynamic Programming: Motivation, Memoization

The table can also be filled in iteratively, bottom-up:

```
int fib_bottom_up(int n) {  
    int lookup[n+1];  
    lookup[0] = 0;  
    lookup[1] = 1;  
    for (int i = 2; i <= n; i++) {  
        lookup[i] = lookup[i-1] + lookup[i-2];  
    }  
    return lookup[n];  
}
```

(The bottom-up approach can be optimized in this case: Since we do need only the last two elements of the lookup table, we only store these two and not all of them.)

## Dynamic Programming: Two Strategies: Top-Down / Bottom-Up

There are two basic strategies applied for dynamical programming:

- Memoized recursive approach / top-down approach:
  - Find a recursive approach to solve a problem. Keep the recursive solution but rewrite it such that a result for a sub-task is stored in a lookup table the first time it is solved.
  - Before applying the recursive solution to a sub-task, check whether it is already present in the lookup table.
- Bottom-up approach (iterative approach):
  - Replace the recursive solution with an iterative solution.
  - Start with solving the smallest sub-tasks and successively fill in a table by constructing the solution for the current sub-task from previous subtasks (i.e. the lookup table is more of a construction table than a lookup table in this approach).



## Dynamic Programming: Two Strategies: Top-Down / Bottom-Up: Which one to use?

Both strategies can be applied to solve a given problem (unless one approach is explicitly requested) and both approaches will give exactly the same results.

- The **memoized recursive approach / top-down approach** can have advantages when some of the subproblems are not required to be solved.
- The **bottom-up approach (iterative approach)** is usually better when all subproblems must be solved. It is a bit more efficient in practice then because it doesn't require as many function calls (overhead for recursion) and has the potential to save memory.

## Dynamic Programming: Memoization / Generalized Top-Down Approach / Template Algorithm

The top-down approach for memoization can be applied to *any* recursive algorithm. The template algorithm and description below shows in recipe style how this can be done:

- In the beginning, create an empty lookup table (outside memoized recursive function; called «lookup» in the adjacent example pseudo code).
- Within the recursive function, first check whether the value for the current call is already stored in the lookup table. If this is the case: just return the value from the lookup table.
- If the value is not stored in the lookup table yet, then run the algorithm recursively and store the result in the lookup table finally.

### Algorithm: memoized\_recursion(n)

```
1  if n contained in lookup then
2    | return get_from_lookup(n)
3  else
4    | if base case then
5      |   apply base case
6    | else
7      |   result = apply_recursive_step(..)
8      |   store_in_lookup(result)
9    | return result
```

## Principle of Optimality, Optimal Substructure

Obviously, the lookup table used in dynamic programming is only a meaningful help if there are actually repeating calculations that can be saved (otherwise it's just a waste of memory and time). But there is a second condition that needs to be considered to decide whether a lookup table is a suitable solution approach: The **construction of a solution from sub-problems must create an optimal solution**. This is not always the case.

This is where the **optimal substructure** principle (also known as principle of optimality) comes into play. If the problem in question does not fulfill this property, dynamic programming is not an effective solution approach. Therefore, it should always be ensured (ideally with a proof) that this property holds.

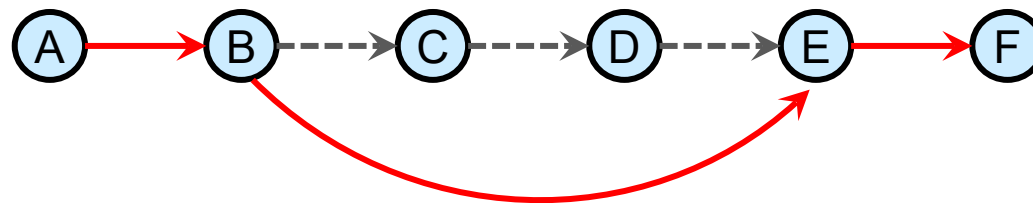
Furthermore, this property can help to find out what the different cases of the recursive formulation should be (or vice-versa).

## Optimal Substructure: Example: Shortest Path

The shortest path problem is a canonical example of the principle of optimality / of an optimal substructure:

If the shortest path between two nodes (A and F, for example) is found, then any part between two nodes belonging to this shortest path (e.g. the path between B and E, but also the path between E and F) is itself the shortest path between the respective nodes: «Subpaths of shortest paths are shortest paths.»

Justification: If some subpath was not the shortest path, one could substitute the shorter subpath and create a shorter total path.



A more formal way to put this is the following:

$$\begin{array}{lclclcl}
 \text{optimal solution of} & & & & & & \\
 \text{complete problem} & = & \text{solution of} & + & \text{part of optimal} & \Rightarrow & \text{solution of partial} \\
 \text{path A - F} & & \text{partial problem} & & \text{solution} & & \text{problem is optimal} \\
 & & \text{path A - E} & & \text{path E - F} & & \text{path A - E}
 \end{array}$$



## Dynamic Programming: Intermediate Summary

### Basic idea of dynamic programming:

- Split the task into smaller sub-tasks (which are overlapping and have optimal substructure).
- Solve the sub-tasks, saving the intermediate results (= memoization).

### Remark:

- The solution to the final task might use only some intermediate results (unless it's possible to tell which will or will not be used, all sub-tasks are solved).



## Exercise 11, Task 1.2

Consider for coin change of amount 14 with coins 10, 7, 2, 1. Give a solution with least coins by dynamic programming.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

## Exercise 11, Task 1.1: Multiple Choice Question Coin Change Problem

The coin change problem can be solved by...

- A. Greedy algorithm, can obtain the global optimal solution
- B. Divide and conquer recursion, can obtain the global optimal solution
- C. Dynamic programming, can obtain the global optimal solution
- D. Greedy algorithm, can not obtain the global optimal solution

→ **A, C**

Option D is *false* because a greedy algorithm *can* obtain an optimal solution in certain cases. It would be correct to state that it *will not always* obtain the optimal solution. For the same reason, option A is correct since a greedy algorithm *can* obtain the optimal solution but will not always achieve this. Option C is correct: Dynamic programming (as implemented in the lecture) can and always will obtain the optimal solution (if there is any) because it has optimal substructure.



## Excursion: Greedy Algorithms

A greedy approach will successively always take the biggest possible partial solution for the remaining problem. This strategy will not always yield an optimal solution. If we know for some reason that the greedy approach will give an optimal result or if we do not necessarily need an optimal solution (heuristics), we can still apply a greedy approach.

For example, a greedy approach to the minimal coin change problem, i.e. always picking the biggest denomination that «fits» the remaining amount, will not necessarily result in an optimal solution (or any solution at all). Consider denominations  $d = [8, 6, 1]$  and amount  $x = 12$ , this approach will yield 5 coins ( $8 + 1 + 1 + 1 + 1$ ) but optimal would be 2 ( $6 + 6$ ).





## Dynamic Programming Example: Levenshtein Distance

The Levenshtein distance (also called: edit distance) of two strings is the number of operations needed to convert one string into the other.

Allowed operations are:

- insertion of a single character
- deletion of a single character
- substitution of a single character with another single character

We will assume here, that all of the operations mentioned above have the same weight / cost of 1.



## Dynamic Programming Example: Levenshtein Distance

*Example:*

What is the Levenshtein distance between the strings  $s_1 = \text{"baum"}$  and  $s_2 = \text{"bus"}$ ?

Transformation of  $s_1$  into  $s_2$  can be done by:

- substitution of 'm' into 's' in  $s_1$
- deletion of character 'a' in  $s_1$

$$\text{LD}(\text{"baum"}, \text{"bus"}) = \text{LD}(s_1, s_2) = 2$$

## Dynamic Programming Example: Levenshtein Distance

Consider two strings  $a = a_1a_2\dots a_n$  and  $b = b_1b_2\dots b_m$ . The following recursion formula  $\text{lev\_dist}(i, j)$  calculates the Levenshtein distance between two substrings  $a_1a_2\dots a_i$  and  $b_1b_2\dots b_j$ :

$$\text{lev\_dist}(i, j) = \begin{cases} b & \text{if } a \text{ is the empty string} \\ a & \text{if } b \text{ is the empty string} \\ \text{lev\_dist}(i-1, j-1) & \text{if } a_i == b_i \\ \min(\text{lev\_dist}(i-1, j-1), \text{lev\_dist}(i-1, j), \text{lev\_dist}(i, j-1)) + 1 & \text{otherwise} \end{cases}$$

do nothing: both current letters are identical  
↙

↑ substitution (go to upper left in table)  
↑ deletion (go up in table)  
↑ insertion (go to left in table)

## Dynamic programming example: Levenshtein distance

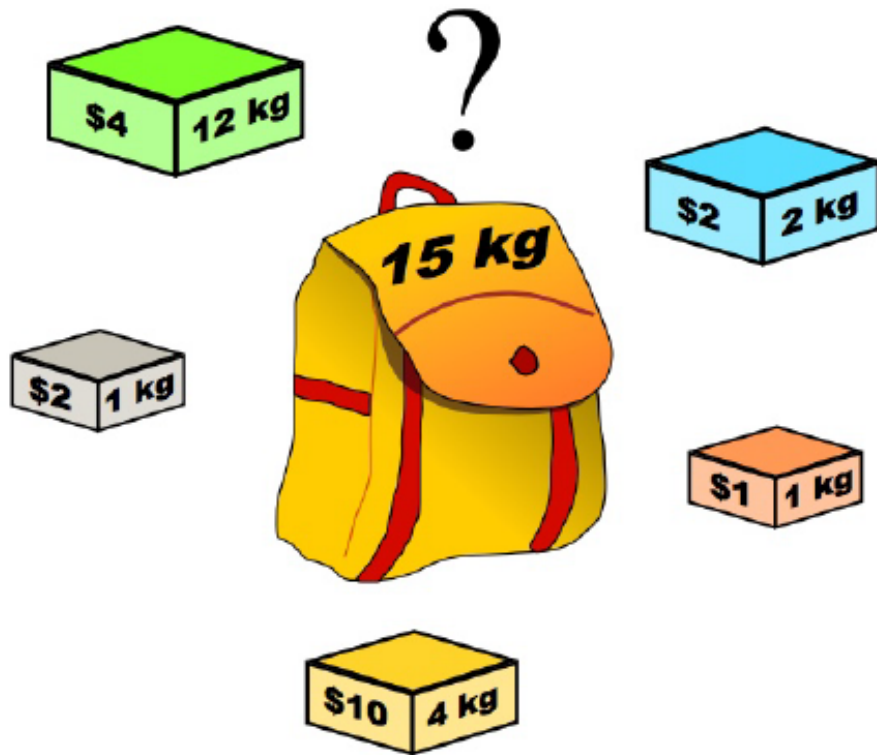
The Levenshtein distance can be calculated using dynamic programming:

	∅	P	L	A	N	E	T
∅	0	1	2	3	4	5	6
P	1	0	1	2	3	4	5
A	2	1	1	1	2	3	4
P	3	2	2	2	2	3	4
E	4	3	3	3	3	2	3
R	5	4	4	4	4	3	3

$= \min(3 + 1, 3 + 1, 2 + 1),$   
 $2 + 1$  because  
 'T' ≠ 'R'

	∅	P	L	A	N	E	T
∅	<b>0</b>	1	2	3	4	5	6
P	1	<b>0</b>	<b>1</b>	2	3	4	5
A	2	1	1	<b>1</b>	2	3	4
P	3	2	2	2	<b>2</b>	3	4
E	4	3	3	3	3	<b>2</b>	3
R	5	4	4	4	4	3	<b>3</b>

## Dynamic Programming Example: The 0/1 Knapsack Problem



## Dynamic Programming Example: The 0/1 Knapsack Problem

As a first step, we find a recursive solution for the problem. It looks as follows:

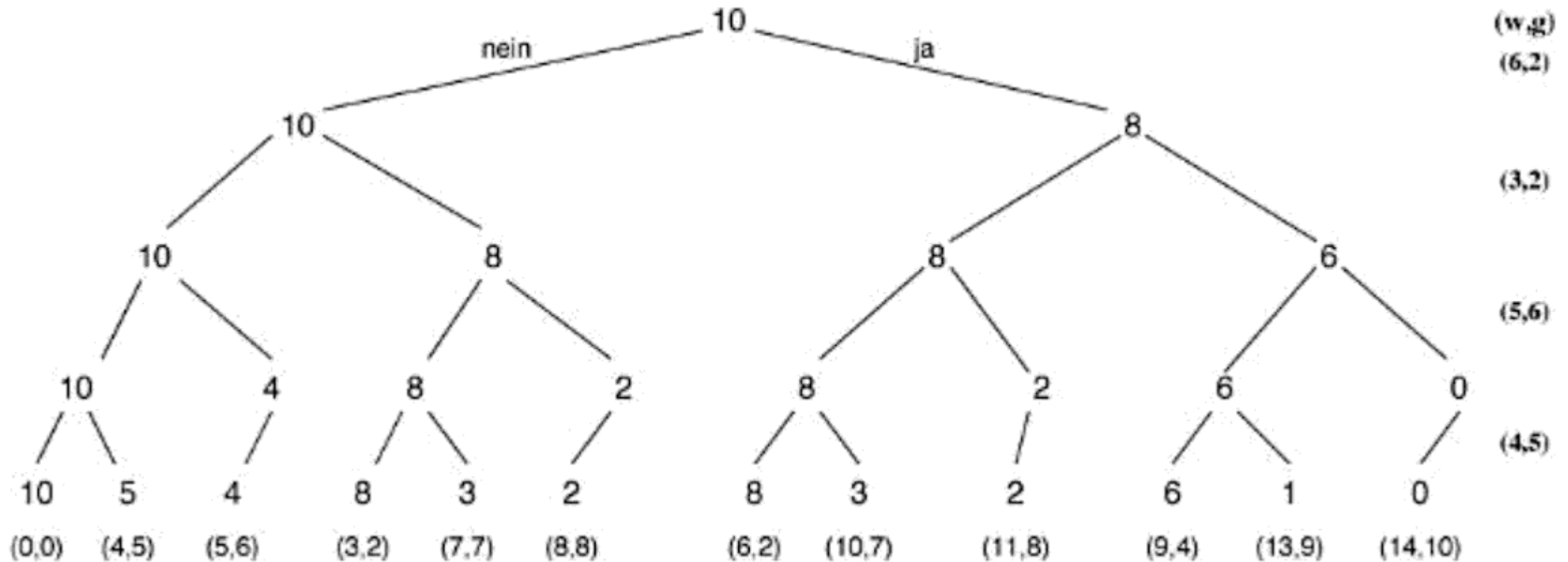
$\text{profit}(\text{considered until item, remaining capacity}) = \text{profit}(x, c)$

$$\text{profit}(x, c) = \begin{cases} 0 & \text{Base case: no items left or no capacity left} \\ \text{profit}(j - 1, c) & \text{Recursive case 1: remaining capacity is insufficient} \\ \max(\text{profit}(x - 1, a), \text{profit}(x - 1, c - s[x]) + p[x]) & \text{Recursive case 2: either take or do not take item} \end{cases}$$

do *not* take item currently under consideration;  
items selectable reduced, capacity unchanged

do take item currently under consideration;  
items available reduced, capacity reduced,  
profit increased

## Dynamic Programming Example: The 0/1 Knapsack Problem



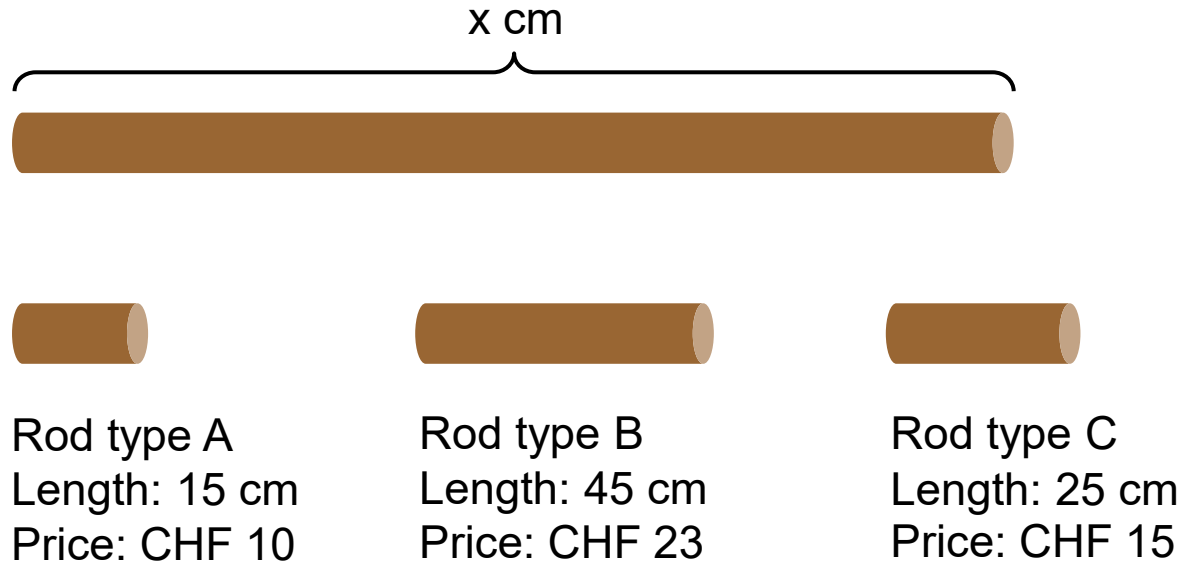
## Dynamic Programming Example: The 0/1 Knapsack Problem

$i$	$w$ $[v] \ c$	0	1	2	3	4	5	6	7
0	[-] -	0	0	0	0	0	0	0	0
1	[4] 3	0	0	0	4 0 4	4 0 4	4 0 4	4 0 4	4 0 4
2	[7] 5	0	0	0	4	4	7 4 7	7 4 7	7 4 7
3	[1] 1	0	1 0 1	1 0 1	4 4 1	5 4 5	7 7 5	8 7 8	8 7 8
4	[5] 4	0	1	1	4	5 5 5	7 7 6	8 8 6	9 8 9



## Dynamic Programming Example: Rod Cutting Problem

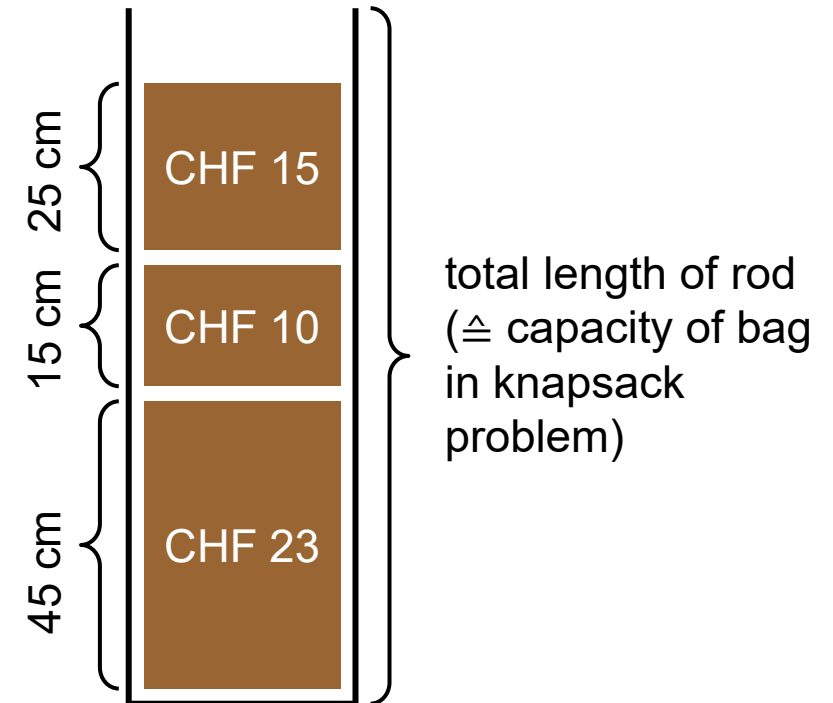
Given a rod with a length of  $x$  centimeters and a set of possible pieces / table of prices, determine the maximum revenue that can be obtained by cutting up the rod and selling the pieces. (Surplus pieces which do not have the length of one of the given types of pieces may not be sold and have no value.)



Type	Length	Price
A	15 cm	CHF 10
B	45 cm	CHF 23
C	25 cm	CHF 15

## Rod Cutting Problem: Knapsack Problem in Disguise

- Cutting a rod with a given length into (types of) pieces of predefined lengths is **equivalent to putting (types of) items of defined weight into a bag with a given weight capacity** (consider the bag as a tube which is filled with the cut pieces).
- Thus, the rod cutting problem is **very similar to the knapsack problem**. The only difference is that we have types of items instead of single items and therefore an «item» can be selected multiple times without being consumed.
- Another very similar problem is the **coin changing problem** discussed in the lecture slides (SL08, p. 43ff.).



## Dynamic Programming: Additional Exercise: Apartment Auction

(Exercise 3 of final exam of FS 2015)

A construction company has completed a new building with  $A$  apartments that are all available in the market. There are  $B$  potential buyers for these apartments. Given a buyer  $b$  and two arrays  $s$ ,  $p$ , the value  $s[b]$  corresponds to the number of apartments buyer  $b$  would like to buy and  $p[b]$  corresponds to the money the buyer is willing to pay for the apartments. The goal of the company is to determine the maximum profit that it can achieve by selling the apartments to the appropriate buyers. Note that not all apartments need to be sold, and a buyer gets either all the apartments he asked for or none.

Example: Assume there are 8 apartments to be sold and 4 potential buyers. The number of apartments that each buyer wants to buy and the price they are willing to pay are available in the arrays:  $s = [4, 1, 2, 3]$  and  $p = [8, 1, 10, 9]$ , respectively. The maximum profit that can be achieved is 20.

This problem is equivalent to the 0/1 knapsack problem, where the number of apartments available takes the role of the capacity of the knapsack, and the buyers and their requested number of apartments and prices they are willing to pay take the role of the items which are put into the knapsack.

## What is Dynamic Programming? – Revisited

- Dynamic Programming = recursion + memoization
- Dynamic Programming = artful application of (lookup) tables

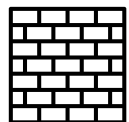
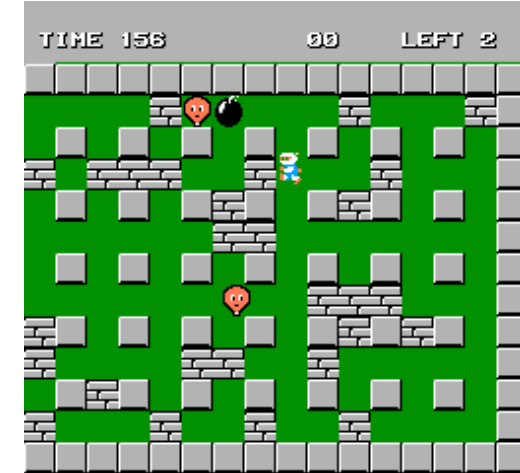
How can I learn dynamic programming?

- Understanding dynamic programming **needs a lot of practice**. Since the remaining time until the exam is quite short, you may also, as substitute, try to **know some of the most common / famous examples** (which might show up in a different outfit), e.g.:
  - Knapsack (= rod cutting  $\approx$  coin change)
  - Longest common subsequence (with many variations), Levenshtein distance
  - Fibonacci, bin packing, ...
- Since the variety of different applications is quite big, it is difficult to give a general recipe, how to find a dynamic programming solution. Although, there are some general principles which are always present.

## Exercise 11, Task 2: Bomberman



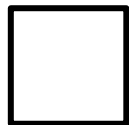
Consider a 2-D matrix  $M$ . Value of each cell of  $M$  is a wall 'W' or a bug 'B' or empty '0'. You can place a bomb at any empty cell with value '0'. The bomb will explode in the following four directions: up, down, left and right. The explosions will be stopped by the walls or the border of the matrix. Consequently, bugs that are covered by the explosions will be eliminated.



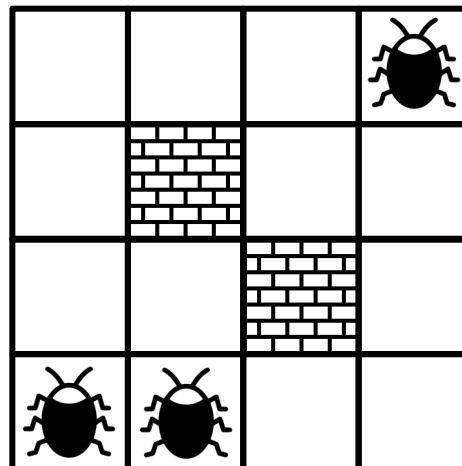
**W** wall



**B** bug



**0** empty



	0	1	2	3
0	0	0	0	B
1	0	W	0	0
2	0	0	W	0
3	B	B	0	0

## Exercise 11, Task 2.1: Bomberman: Example

	0	1	2	3
0	0	0	0	B
1	0	W	0	0
2	0	0	W	0
3	B	B	0	0

	0	1	2	3
0	0	0	0	1
1	0	0	0	0
2	0	0	0	0
3	1	2	2	2

$V_l$  for left direction

	0	1	2	3
0	1	1	1	1
1	0	0	0	0
2	0	0	0	0
3	2	1	0	0

$V_r$  for right direction

	0	1	2	3
0	0	0	0	1
1	0	0	0	1
2	0	0	0	1
3	1	1	0	1

$V_u$  for up direction

	0	1	2	3
0	1	0	0	1
1	1	0	0	0
2	1	1	0	0
3	1	1	0	0

$V_d$  for down direction

## Exercise 11, Task 2.2: Bomberman: Recursive Formulation

$$V_l[i, j] = \begin{cases} 0 & M[i][j] = "W" \\ 1 & M[i][j] = "B", j = 0 \\ V_l[i, j - 1] + 1 & M[i][j] = "B", j > 0 \\ 0 & M[i][j] = "0", j = 0 \\ V_l[i, j - 1] & M[i][j] = "0", j > 0 \end{cases}$$

$$V_u[i, j] = \begin{cases} 0 & M[i][j] = "W" \\ 1 & M[i][j] = "B", i = 0 \\ V_u[i - 1, j] + 1 & M[i][j] = "B", i > 0 \\ 0 & M[i][j] = "0", i = 0 \\ V_u[i - 1, j] & M[i][j] = "0", i > 0 \end{cases}$$

$$V_r[i, j] = \begin{cases} 0 & M[i][j] = "W" \\ 1 & M[i][j] = "B", j = y - 1 \\ V_r[i, j + 1] + 1 & M[i][j] = "B", j < y - 1 \\ 0 & M[i][j] = "0", j = y - 1 \\ V_r[i, j + 1] & M[i][j] = "0", j < y - 1 \end{cases}$$

$$V_d[i, j] = \begin{cases} 0 & M[i][j] = "W" \\ 1 & M[i][j] = "B", i = x - 1 \\ V_d[i + 1, j] + 1 & M[i][j] = "B", i < x - 1 \\ 0 & M[i][j] = "0", i = x - 1 \\ V_d[i + 1, j] & M[i][j] = "0", i < x - 1 \end{cases}$$



## Exercise 11, Task 2.3: Bomberman: Implementation



## Exercise 11, Task 3: Longest Balanced/Valid Parentheses (Non-Contiguous) Subsequence

A parentheses string contains only left and right parentheses. A parentheses string is balanced if and only if left and right parentheses are correctly matched. For example, the strings " $()()$ " and " $()()$ " are balanced while " $()()()$ " is not. Given a parentheses string  $S[0..n-1]$  with  $n$  parentheses, determine the length of longest balanced subsequence of string  $S$ . The subsequence is derived by selecting/not selecting elements, without changing the order of the selected elements. For example, for the parentheses string " $()()()$ ", the length of the longest balanced subsequence is 4. We choose the 1<sup>st</sup>, 2<sup>nd</sup>, 5<sup>th</sup>, and 6<sup>th</sup> characters to form an balanced subsequence " $()()$ ".

Let  $D$  be a two-dimensional matrix with the dimension  $n \times n$  for the parentheses string  $S[0..n-1]$ , and  $D[i,j]$  be the length of longest balanced subsequence of subarray  $S[i..j]$ .

## Exercise 11, Task 3: Example

- What is the solution for the following:

**( ) ( ( ) ( ) ( ( ) (**

- -> 4
- Not really a good example for dynamic programming.
- Can be solved using a stack in linear time! ACHTUNG: ist das das gleiche Problem?? Oder geht es dort um **\*\*contiguous\*\*** subtrings??
- Hint: parentheses theorem next lab

## Exercise 11, Task 3: Example

**(( ))**

**()()**

*end index j*

	0	1	2	3
0				
1				
2				
3				

*start index i*

	$j - 1$	$j$
$i - 1$		
$i$		

## Exercise 11, Task 3.1: Solution for String ((()))

	0	1	2	3	4	5
0	0	0	0	2	4	6
1	0	0	0	2	4	4
2	0	0	0	2	2	2
3	0	0	0	0	0	0
4	0	0	0	0	0	0
5	0	0	0	0	0	0



## Exercise 11, Task 3.2: Recursive Formulation

$$D[i, j] = \begin{cases} 0 & i \geq j \\ \max\{D[i+1, j-1] + 2, \max_{i \leq k < j} \{D[i, k] + D[k+1, j]\}\} & S[i] = "(" \text{ and } S[j] = ")" \\ \max_{i \leq k < j} \{D[i, k] + D[k+1, j]\} & \text{else} \end{cases}$$



## Exercise 11, Task 3.3: Implementation

## Additional Exercise Dynamic Programming: Learning Profit

Alice is preparing for the final exam in algorithm and data structures. She has  $n$  days of time to learn for the exam. Each day in the morning, Alice decides whether she will work a) a lot, b) a little or c) nothing at all on this day for exam preparation. Although, Alice can only work a lot on any given day if she has taken a break the day before, i.e. if she didn't learn anything the day before.

Depending on how much Alice works for exam preparation, she will have a gain in knowledge. This is different from day to day. If Alice works a lot on day  $k$ , she will have a “learning profit” of  $p_k = a_k$ . If Alice works a little on day  $k$ , she will have a “learning profit” of  $p_k = b_k$ . If Alice does not learn at all on day  $k$ , she will have a profit of  $p_k = 0$  for that day. Note that it is possible that learning a little will result in a higher knowledge gain than learning a lot (i.e. it's possible that  $b_k > a_k$ ). The total knowledge gain of Alice after the  $n$  days of learning will be the sum of the profits in the single days.

Assume that two arrays  $A[1..n]$  and  $B[1..n]$  are given, each containing  $n$  integers. The element  $A[k] = a_k$  of array  $A$  contains the profit which Alice receives when she works a lot on day  $k$ . The element  $B[k] = b_k$  of array  $B$  contains the profit which Alice will receive when she works a little on day  $k$ .

Use a dynamic programming approach to design an algorithm which will decide in linear time, i.e. in  $O(n)$ , how much Alice should learn on the available  $N$  days in order to maximize her gain in knowledge (profit).



## Additional Exercise Dynamic Programming: Learning Profit

*Example:*

- Days available for learning:  $n = 5$
- Daily profits when learning a lot:  $A = [a_1, a_2, a_3, a_4, a_5] = [8, 5, 4, 1, 9]$
- Daily profits when learning a little:  $B = [b_1, b_2, b_3, b_4, b_5] = [2, 2, 3, 2, 6]$
- Maximum achievable profit: 22 (when learning: a lot, a little, a little, nothing, a lot)



## Additional Exercise Dynamic Programming: Learning Profit

The knowledge gain / profit which can be achieved when  $n$  days of learning are available can be stated recursively as follows:

$$\text{profit}(n) = \begin{cases} 0 & \text{if } n = 0 \\ \max(a_1, b_1) & \text{if } n = 1 \\ \max(\text{profit}(n-2) + a_n, \text{profit}(n-1) + b_n) & \text{if } n > 1 \end{cases}$$

if Alice made a break  
the day before, she  
can learn a lot on the  
n-th day.

if Alice did not take a  
break the day before,  
she can learn a little on  
the n-th day.

## FS 2019, Exercise 11 – Task 3: Maximum Wine Profit: Task Description

- Assume you have a collection of  $n$  wines placed next to each other on a shelf. Integers from 1 to  $n$  are attributed to the wine bottles from left to right and the price of the  $i^{\text{th}}$  wine is notated as  $p_i$  (prices of different wines can be different).
- Because the wines get better every year, supposing today is the year 1, on year  $y$  the price of the  $i^{\text{th}}$  wine will be  $y \cdot p_i$ .
- You want to sell all the wines you have, but there are the following two constraints:
  1. You can sell exactly one wine per year, starting this year.
  2. Each year you are allowed to sell only either the leftmost or the rightmost wine on the shelf and you are not allowed to reorder the wines on the shelf (i.e. they must stay in the same order as they are in the beginning).

You need to determine the maximum profit you can get, if you sell the wines in optimal order.

## FS 2019, Exercise 11 – Task 3: Maximum Wine Profit: Task Visualized

Example / test set A:  $n = 4$ ,  $p = [1, 4, 2, 3]$



## Exercise 11 – Task 3: Maximum Wine Profit: Recursive Formulation

Idea: Each year, the decision has to be made, whether the leftmost or the rightmost of the remaining wine bottles on the shelf is sold (remember that only the leftmost and rightmost bottle are accessible according to the constraints in the task description).

$$\text{profit}(p[], n, \text{begin}, \text{end}) = \begin{cases} 0 & \text{Base case: no more wines to sell (i.e. begin > end)} \\ \max( \text{profit}(p[], n, \text{begin} + 1, \text{end}) + \text{year} \cdot p[\text{begin}], & \text{Recursive case: either take the leftmost or rightmost wine on the shelf} \\ \text{profit}(p[], n, \text{begin}, \text{end}-1) + \text{year} \cdot p[\text{end}] ) & \end{cases}$$

When decision to sell the leftmost accessible wine is taken. We get a profit of  $\text{year} \cdot p[\text{begin}]$ . Remaining selling decisions are made without considering leftmost position.

When decision is made to sell the rightmost accessible wine. We get a profit of  $\text{year} \cdot p[\text{end}]$ . Remaining selling decisions are made without considering rightmost position.

Notation:

- $n$  = number of wine bottles on the shelf (= number of years to be considered)
- $p[]$  = array of prices of the wines (length  $n$ , indexing starts at 1)
- $\text{begin}$  = position / index of the leftmost bottle accessible;  $\text{end}$  = position / index of the rightmost wine bottle accessible
- $\text{year}$  = current year; can be calculated from the begin and end positions as follows:  $\text{year} = n - (\text{end} - \text{begin} + 1) + 1$  (when indexing starts at 0)

## FS 2019, Exercise 11 – Task 3: Maximum Wine Profit: Purely Recursive Implementation

Purely recursive solution as pseudocode:

**Algo:** WINEPROFITRECURSIVE(*price*, *n*, *begin*, *end*)

---

**if** *begin* > *end* **then**  
    **return** 0;

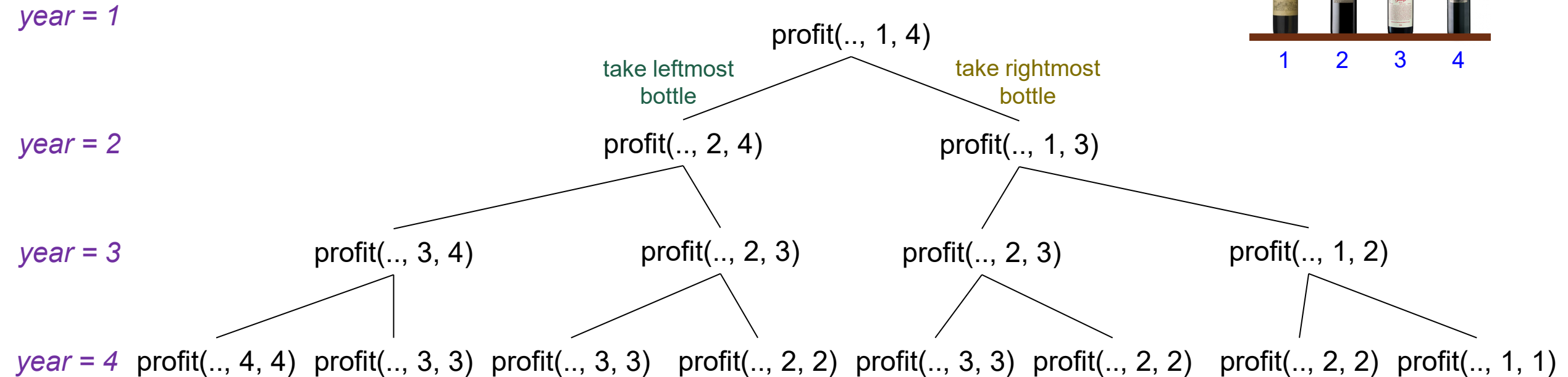
*year* = *n* - (*end* - *begin* + 1) + 1;

**return**  $\max(\text{wineprofitRecursive}(\text{price}, n, \text{begin}+1, \text{end}) + \text{year} * \text{price}[\text{begin}],$   
     $\text{wineprofitRecursive}(\text{price}, n, \text{begin}, \text{end}-1) + \text{year} * \text{price}[\text{end}]);$

## FS 2019, Exercise 11 – Task 3: Maximum Wine Profit: Recursion Tree

Recursive function:  $\text{profit}(p[], n, \text{begin}, \text{end})$

Example / test set A:  
 $n = 4, p = [1, 4, 2, 3]$



## Exercise 11 – Task 3: Maximum Wine Profit: Bottom-Up Solution

		1	4	2	3	<i>price</i>
		1	2	3	4	← <i>end index</i>
1	1	4	19	24	29	
4	2	–	16	22	28	<i>year = 1</i>
2	3	–	–	8	18	<i>year = 2</i>
3	4	–	–	–	12	<i>year = 3</i>
						<i>year = 4</i>

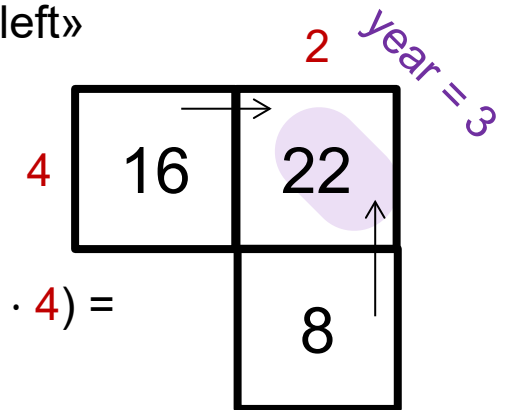
*begin index* ↑

Example / test set A:  
n = 4, p = [1, 4, 2, 3]



Maximum of «left + year · price from up»  
«down + year · price from left»

$$\begin{aligned}
 &\text{e.g. } M[2,3] = \\
 &= \max(M[2,2] + 3 \cdot 2; M[3,3] + 3 \cdot 4) = \\
 &= \max(16 + 6; 8 + 12) = \\
 &= \max(22, 20) = 22
 \end{aligned}$$



## Exercise 11 – Task 3: Maximum Wine Profit: Bottom-Up Solution

**Algo:** WINEPROFITDYNAMIC(price, n)

```
for  $i = 0$  to  $n$  do
```

```
     $m[i][i] = \text{price}[i] * n$ ;
```

```
for  $j = 1$  to  $n$  do
```

```
    for  $i = 0$  to  $n - j$  do
```

```
        begin =  $i$ ;
```

```
        end =  $i + j$ ;
```

```
        year =  $n - (\text{end} - \text{begin})$ ;
```

```
         $m[\text{begin}][\text{end}] = \max(m[\text{begin} + 1][\text{end}] + \text{year} * \text{price}[\text{begin}],$ 
```

```
         $m[\text{begin}][\text{end} - 1] + \text{year} * \text{price}[\text{end}])$ ;
```

```
return  $m[0][n - 1]$ ;
```



## Exercise 11 – Task 3: Maximum Wine Profit: Recursive Memoized (Top-Down) Solution

**Algo:** WINEPROFITMEMOIZED(*price*, *n*, *begin*, *end*)

---

**if** *begin* > *end* **then**

**return** 0;

**if** *m*[*begin*][*end*] > -1 **then**

**return** *m*[*begin*][*end*];

*year* = *n* - (*end* - *begin* + 1) + 1;

*m*[*begin*][*end*] = max(wineprofitMemoized(*price*, *n*, *begin* + 1, *end*) + *year* \*  
    *price*[*begin*], wineprofitMemoized(*price*, *n*, *begin*, *end* - 1) + *year* \* *price*[*end*]);

**return** *m*[*begin*][*end*];



**Universität  
Zürich** <sup>UZH</sup>

**Institut für Informatik**

# Wrap-Up

- Summary
- Feedback
- Outlook
- Questions



## Wrap-Up

- Summary



## Outlook on the Next Lab Session

*Next tutorial:* Wednesday, 25.05.2022, 14.00 h, BIN 0.B.06 (last one!)

*Topics:*

- Review of Exercise 12
- Graphs
- ...
- ... (your wishes)

## Additional Training / Repetition Sessions

I will be here in BIN 0.B.06

- next Friday, 20th of May, from 14.15 h
- next Wednesday, 25th of May, from 16.00 h





**Universität  
Zürich** <sup>UZH</sup>

**Institut für Informatik**

---

**Questions?**



## Outlook on the Next Lab Session

*Next tutorial:* Wednesday, 25.05.2022, 14.00 h, BIN 0.B.06

*Topics:*

- Review of Exercise 12
- Graphs
- ...
- ... (your wishes)



**Universität  
Zürich** <sup>UZH</sup>

Institut für Informatik

---

# Questions?





*Thank you for your attention.*