# Advanced Algorithms

## Master Données et Systèmes Connectés

## Master Machine Learning and Data Mining

## Master Cyber-Physical Social Systems

Amaury Habrard

amaury.habrard@univ-st-etienne.fr

Laboratoire Hubert Curien, UMR CNRS 5516
Université Jean Monnet Saint-Étienne
amaury.habrard@univ-st-etienne.fr

1st Semester - September to December

# Advanced Algorithms

What we will see:

- *non trivial* algorithms for solving some (easy) problems,
- algorithms (accompanied with heuristics) for solving *difficult problems*.
- How to characterize that an algorithm is "smarter" ("better") than another one?
  $\Rightarrow$ the complexity of the algorithm that can be measured in different ways (worst case, average case, best case, expected behavior,...)

But we will not see how to characterize the difficulty of a problem but you will study this in the complexity/calculability class.
$\Rightarrow$ this is generally done with respect to different classes of complexity

# Materials

- Slides and materials available online.
- Practical aspects: project.
- References:
  - Jon Kleinberg and Eva Tardos. Algorithm Design. *Pearson International Edition, Addison Wesley*.
    (Electronic version available on Claroline (ENT))
  - Thomas Cormen, Charles Leiserson, Ronald Rivest and Clifford Stein. Introduction to algorithms. *Dunod 2nd edition*.
    (Available on Claroline, both in french and english)
- For the exam: you will be allowed to use a copy of the slides and your personal notes on the exercices, no books, no electronic device.

# Introduction and "recaps"

NP-completeness, Polynomial aspects, Optimization problems

# Measuring the complexity by Asymptotic Order of Growth

Let $T(n)$ be a **non negative** function measuring the (worst case) running time of a given algorithm on an input of size $n$.

## Classical measure: Asymptotic Upper Bounds

- Given a function $f(n)$, we say that $T(n)$ is $O(f(n))$ (*ie* $T(n)$ *has order of* $f(n)$ *in terms of time complexity*) if there exist **2 constants** $c > 0$ and $n_0 \geq 0$ s.t. for all $n \geq n_0$ we have
$$T(n) \leq c \times f(n).$$
In words: *for sufficiently large* $n$, $T(n)$ *is bounded above by* $f(n)$ *up to a constant factor*.

Note that $O(\cdot)$ expresses an **asymptotic** upper bound and not the exact growth rate of the function $T(n)$ encoding the complexity of the algorithm. The constant $c$ must be fixed **independently** from $n$.

$\Rightarrow$ Try to draw some plots to illustrate the definition

$\Rightarrow$ What is the intuition behind the two constants?

# Measuring the complexity by Asymptotic Order of Growth (below)

## Classical measure: Asymptotic Lower Bounds

- Given another function $f(n)$, we say that $T(n)$ is $\Omega(f(n))$ (*i.e.* $f(n)$ *is order of* $T(n)$ *in terms of time complexity*), also written $T(n) = \Omega(f(n))$, if there exist 2 constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have
$$c \times f(n) \leq T(n).$$
In words: *for sufficiently large* $n$, $T(n)$ *is asymptotically lower bounded by (a constant factor of)* $f(n)$.

Note that $\Omega(\cdot)$ expresses an **asymptotic** lower bound that does not follow necessarily the exact growth rate of the function $T(n)$.

# Measuring the complexity by Asymptotic Order of Growth (average)

## Classical measure: Asymptotic Tight Bounds

- Given another function $f(n)$, if $T(n)$ is both $O(f(n))$ and $\Omega(f(n))$ then $T(n)$ is $\theta(f(n))$, also written $T(n) = \theta(f(n))$.
  In words: *for sufficiently large n, $T(n)$ grows exactly like $f(n)$ to within a constant factor.*

Note that $\theta(\cdot)$ expresses an **asymptotic** tight bound on the order of growth of $T(n)$.

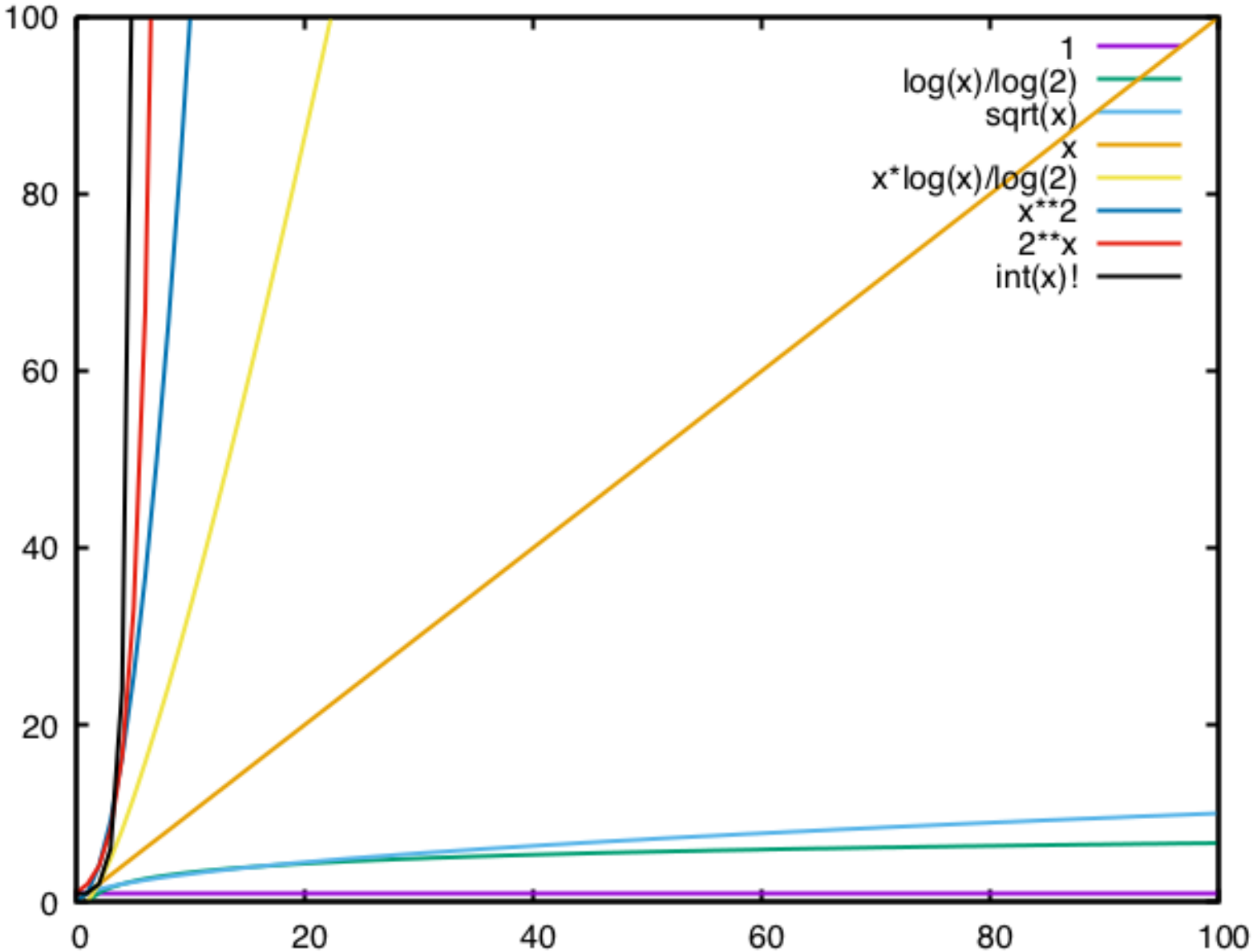
For the three measures, $f(n)$ is assumed to be a positive function, which is natural for modeling time complexity.

# Survey of classic running time

- Constant time $O(1)$: atomic operations
- Linear time $O(n)$: search in an unsorted array
- $O(n \log n)$: sorting, some divide and conquer approaches, sometimes called quasi linear
- Quadratic time $O(n^2)$:
- Polynomial time $O(n^k)$ with $k \geq 1$ a constant
- Exponential time: $O(2^n)$ and more generally $O(k^n)$ with $k > 1$ a constant (or $O(k^{\text{poly(n)}})$
- Factorial time $O(n!)$
- Doubly exponentiel $O(2^{2^n})$, or more...

Note: for any $x > 0$, $\log_b(x) = \log_a(x)/\log_a(b)$

# Illustration



# From a practical side (1/2)

| *Complexity* | $\log_{10}(n)$ | $n$ | $n^2$ | $n^3$ | $2^n$ | $n!$ |
|---|---|---|---|---|---|---|
| *duration* $= 1s$ | *astronomic* : $10^{10^6}$ | $10^6$ | $10^3$ | $10^2$ | 19 | 10 |
| $1min$ | *astronomic* | $6.10^7$ | $8.10^3$ | $4.10^2$ | 25 | 11 |
| $1h$ | *astronomic* | $4.10^8$ | $2.10^4$ | 1500 | 31 | 13 |
| $1day$ | *astronomic* | $9.10^{10}$ | $10^5$ | 4400 | 36 | 16 |

Table 1: Approximative amount of a data an algorithm can process with respect to a duration in line and a complexity in column. For example, in 1 hour a program with an exact complexity of $n^3$ can process an input of size 1500. Astronomic refers to a size greater than the estimated number of atoms in universe ($10^{80}$)

(From Bosc, Guyomard and Miclet, Conception d'algorithmes, Eyrolles).

# From a practical side (2/2)

| *Complexity* | $\log_2(n)$ | $n$ | $n\log(n)$ | $n^2$ | $2^n$ |
|---|---|---|---|---|---|
| *size* $= 10$ | $3\mu s$ | $10\mu s$ | $30\mu s$ | $100\mu s$ | $100\mu s$ |
| 100 | $7\mu s$ | $100\mu s$ | $700\mu s$ | $1/100s$ | $10^{14}$centuries |
| 1000 | $10\mu s$ | $1000\mu s$ | $1/100s$ | $1s$ | *astronomic* |
| 10000 | $13\mu s$ | $1/100s$ | $1/7s$ | $1.7min$ | *astronomic* |
| 100000 | $17\mu s$ | $1/10s$ | $2s$ | $2.8h$ | *astronomic* |

Table 2: Assuming that an elementary operation can be done in $1\mu$s, this table gives the time an algorithm needs to process an amount of data of size mentioned in line given an exact complexity in column. For example, a program in $n^2$ can process an input of size 1000 in one second. An astronomic time is larger than a billion billion years.

(From Bosc, Guyomard and Miclet, Conception d'algorithmes, Eyrolles).

# Polynomial time complexity: A class of interest

From a theoretical standpoint, it is generally admitted that problems that can be solved in polynomial time algorithm are tractable. However as we saw, there are some exceptions:

- If the constant $k$ is too big *e.g.* $O(n^{100})$ but even $O(n^4)$
- If the input data are too large *e.g.* $n \sim 10^9$ (big data)

Polynomial time aspect is interesting in various aspects:

- In many computational models a problem that can be solved in polynomial time in a model can also be solved in polynomial time in another one. For example, the class of problems solved in polynomial time by a Turing machine can also be solved in poly time by a parallel computer.
- Closure properties:
  - If an algorithm makes calls to many polynomial algorithms a finite number of times, it remains polynomial.
  - Using the output from a polynomial time algo as input of another polynomial time algo remains polynomial.

# Some classic properties

## $f$, $g$, $h$ functions taking non negative values

- If $f = O(g)$ and $g = O(h)$ then $f = O(h)$
- If $f = \Omega(g)$ and $g = \Omega(h)$ then $f = \Omega(h)$
- If $f = \theta(g)$ and $g = \theta(h)$, then $f = \theta(h)$
- If $f = O(h)$ and $g = O(h)$ then $f + g = O(h)$
- If $g = O(f)$ then $f + g = \theta(f)$
- If $f$ is polynomial of degree $d$ in which the coefficient $a_d$ associated to the term of degree $d$ is positive, then $f = O(n^d)$
- For every $b > 1$ and every $x > 0$, we have $\log_b(n) = O(n^x)$
- For every $r > 1$ and every $d > 0$, we have $n^d = O(r^n)$

# Abstract Problem, instance, solution

An abstract problem $Q$ is defined as a binary relation between a set of instances of the problem and a set of solutions for this problem.

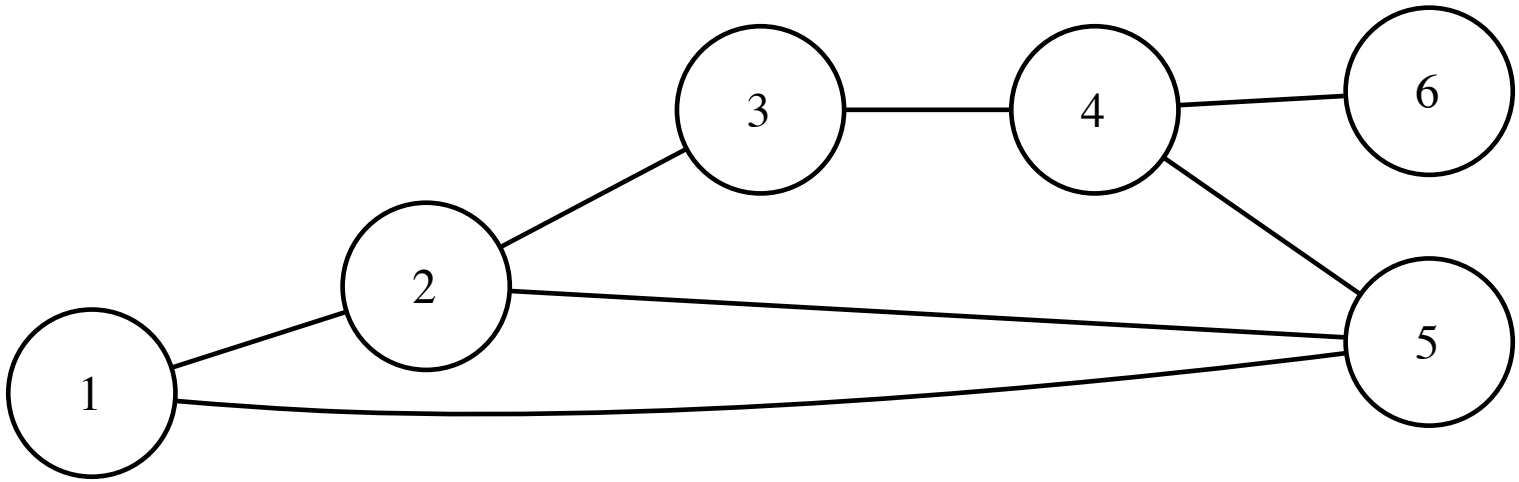
## Example: Shortest Path Problem in a graph

Consider the problem where the objective is to find the shortest path between two nodes of an undirected and unweighted graph $G = (S, A)$

- Instance: a triple $(G, u, v)$ where $G$ is a graph and $u$ and $v$ are two nodes of the graph.
- Solution: a sequence a nodes defining a shortest path between $u$ and $v$ in $G$.

We associate to each instance (composed of a graph and two nodes $u$ and $v$ of the graph) a shortest path between $u$ and $v$.

# Example




The shortest path problem applied on this graph and the nodes 1 and 6 admits only one solutions:

- 1, 5, 4, 6

# Different versions of a problem

## Decision problem

An abstract decision problem is a mapping from a set of instances to the set $\{0, 1\}$ of solutions.

## Example: the *path* decision problem

Given a graph $G = (V, E)$, two nodes $u$ and $v$, an integer $k \geq 0$, Is there a path between $u$ and $v$ of length lower or equal to $k$?

If $i = (G, u, v, k)$ is an instance of this problem, then $path(i) = 1$ if there exists a path between $u$ and $v$ of length lower or equal to $k$ in $G$.

# Concrete Problems and class of complexity $P$

## Concrete problem

A concrete pb corresponds to the encoding of an abstract problem (use of data structures, use of binary strings, use of a formal language framework)

In this class we will not really pay attention to the encoding problems, even though it can be an important aspect, we will assume that every input/output is encoded as a string.

## The class $P$

A concrete problem $Q$ can be solved in polynomial time if there exists an algorithm solving it in $O(n^k)$ for any input of size $n$ with $k \geq 0$ a constant. The set of problems that can be solved in polynomial time defines the class of complexity $P$.

# Polynomial time certification - the class NP

## Polynomial certifier

An abstract problem admits a polynomial certifier if for any instance $\mathtt{s}$ of the problem and any proposed solution $\mathtt{t}$ (a certificate), there exists a **polynomial** algorithm that can check if $\mathtt{t}$ is a solution for $\mathtt{s}$.
Note: $|\mathtt{t}|$ must be $\leq poly(|\mathtt{s}|)$, the polynomial constraint is only needed for positive solutions.

## Non-deterministic Polynomial time (NP)

The class NP is defined as the set of problems for which there exists a polynomial certifier.

## Example with the *path* problem

Given a sequence a nodes, it is easy to check that it forms a path of length lower or equal than $k$.

# NP problem - other example

## Problem: Hamiltonian Cycle (HC)

Given an undirected graph $G = (V, S)$, is there a simple cycle containing each vertex of $V$ exactly once?

- To solve this problem, we could imagine to consider all the possible permutations of the nodes of $G$ and check if each permutation defined an Hamiltonian path: $\Omega(n!)$, clearly not polynomial.
- If we only look for a certifier, given a sequence of nodes, we need to verify if it is an Hamiltonian path: we just to see if there exists an edge between two consecutive nodes and check that all the nodes appear exactly once $\rightarrow$ polynomial (linear).
- Thus $HC \in NP$

# NP-Completeness

## Reductions and transitivity

A problem $Q$ can be reduced to a problem $Q'$ (denoted by $Q \leq_P Q'$) if there exists a function, computable in polynomial time, able to transform any instance of $Q$ to an instance of $Q'$. $Q$ is polynomial-time reducible by $Q'$ or in other words any instance of $Q$ can be solved by $Q'$ threw a polynomial transformation.

The notation $P_1 \leq P_2$ means that $P_2$ is as hard as $P_1$, with respect to a polynomial factor.
Another interpretation: (i) if $P_2$ can be solved in polynomial time, then $P_1$ can be solved in polynomial time, (ii) if $P_1$ cannot be solved in poly time, then $P_2$ cannot be solved in poly time.

# NP-complete problems

## NP-Complete (NP-C) problems

A decision problem $Q$ is NP-complete if:

1. $Q \in NP$
2. For every problem $Q' \in NP$, $Q' \leq_P Q$

The set of NP-complete decision problems defines the class of complexity NPC. If a problem verifies the property 2 (and not necessarily property 1), it is said NP-hard, any optimization problem having NP-complete decision problem version is NP-hard.

$\Rightarrow$ NP-complete problems have an "interesting property": they are all equivalent with respect to a polynomial factor. As a consequence, if one of the problems can be solved in polynomial time then all the problems in NP can be solved in polynomial time !

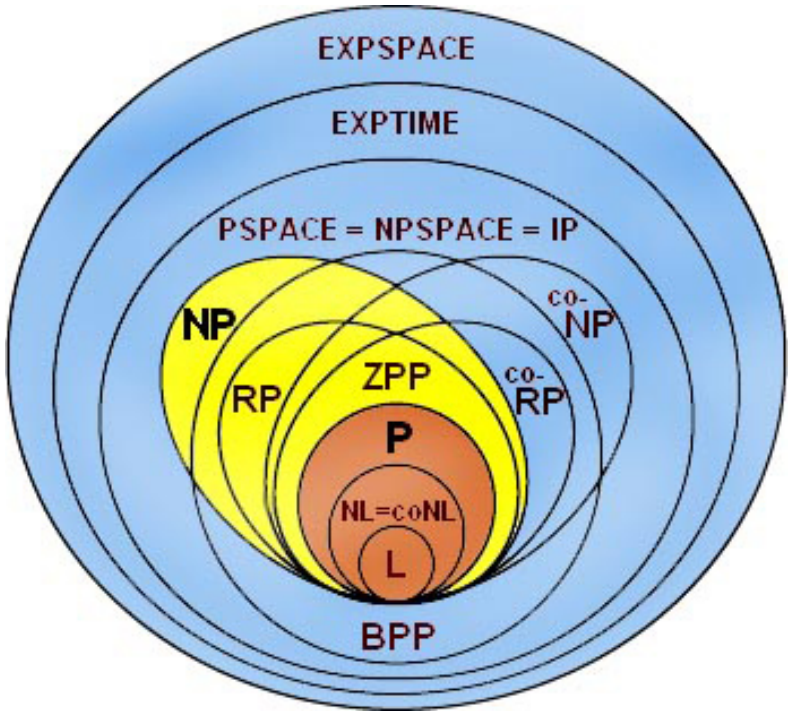
# NP-complete problems

## Important remarks

- we know that $P \subseteq NP$: easy since any problem $P$ can be solved in polynomial time, you have directly a certifier,
- The question ?$NP = P$? is still an open problem today ...
- Note that If one can prove that one NP-complete problem can be solved in polynomial time, then all NP-complete problems can be solved in polynomial time.
- The term *NP-complete* is used for decision problems, otherwise we use the term *NP-hard* for the associated general problem: they correspond to the problems that are at least as hard as the hardest problems in *NP*.

**Note :** in this class we will not directly work on the notion of NP-complete problems - you will study this in the complexity/calculability class.
But we will see some strategies to solve in practice these problems.

# Note: there exist many classes of complexity




Problems solvable in EXP-TIME, EXP-SPACE, P-SPACE, LOG-SPACE, randomized algorithms. The figure above is not complete.

Illustration taken from

`https://jeremykun.com/2012/02/29/other-complexity-classes/`

# Optimization problems

In this class, we will generally consider optimization problems. They have potentially different solutions, each of these having a specific value and we look for the solution with the best possible value (minimum or maximum): the optimal value. If there are more than one optimal values, we just take one.

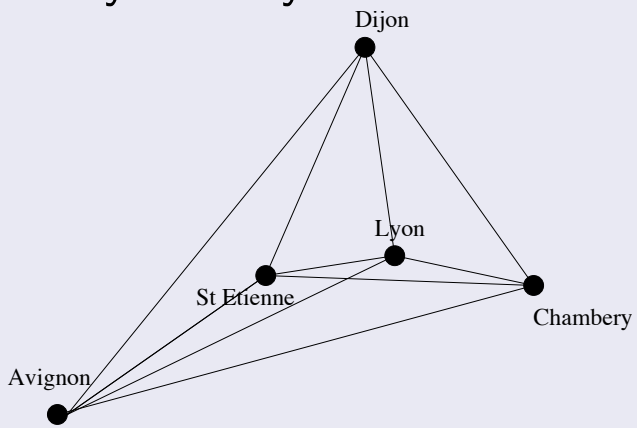
## Shortest path

Given a triple $(G, u, v)$ where $G = (S, A)$ is a graph and $u$ and $v$ are two nodes of the graph, find the minimum path between $u$ and $v$ in $G$.

# Optimization problems

## TSP: Traveling Salesman Problem

Given a list of cities and the distances between each pair of them, find the shortest possible route that visits each city exactly once and returns to the origin city?

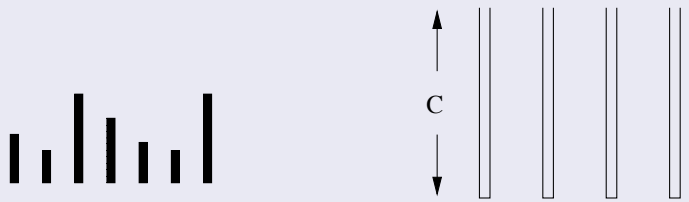



This can be modeled as finding an Hamiltonian cycle with the minimal weight in an undirected weighted graph.

# Optimization problems

## Bin packing

Given a finite set of bins of capacity $C$, and a set of objects $\{o_1, \ldots, o_n\}$ each of them with a capacity $v_i$, is it possible to pack all of them using less than $k$ bins?




a set of 7 objects  a set of 4 bins of capacity $C$

## Knapsack problem

**Input:** the capacity $C$ of the knapsack, $n$ different types of objects, and for each object its weight $w(i)$ and its value $v(i)$

**Output:** Find a collection of objects such that the sum of the individual weights is lower or equal than $C$ and that maximizes the sum of the individual values.

# Needing heuristics

Optimization problems are generally NP-hard. Using naive algorithms to solve them is generally intractable (exponential time complexity). The idea is to use heuristics to find approximated reasonable solutions in reasonable time. This can be done by using various algorithms:

- Dynamic programming
- Greedy algorithm
- Stochastic/randomized approaches
- Branch and bound algorithms
- Linear programming
- Genetic algorithms
- ...

In this class, we propose to study some of these methods. We will sometimes see that for some problems that may appear difficult, there exits a polynomial solution using a non-naive analysis of the problem (dynamic programming and greedy approaches).

# Next !

- We will see two strategies for solving problems that may appear (NP-)Hard in polynomial time : Dynamic Programming and Greedy Approaches.
- Then, we will some algorithm and strategies for solving or at least approximate the solution of NP-Hard problems. **Note that Greedy strategies can also be used to find approximated solutions.**

# Dynamic Programming (DP)

# A step back on the divide and conquer approach

## Principle

- **Divide** a problem into independent subproblems
- **Conquer** these subproblems by solving them recursively (or directly when their input size is very small)
- **Combine** the solutions of the subproblems to build the solution of the initial problem

## Example Merge-sort

```
MERGE-SORT(A, p, r)
1  if p < r
2      q = ⌊(p + r)/2⌋
3      MERGE-SORT(A, p, q)
4      MERGE-SORT(A, q + 1, r)
5      MERGE(A, p, q, r)
```

- $A$ is an array of size $n$
- $p, q, r$ indexes between 1 and n
- MERGE is a linear procedure (see next slide)

Example: Apply it and draw the recursive calls on $[5, 2, 6, 1, 5, 2, 3, 1]$

# The Merge-sort algorithm

MERGE procedure: MERGE$(A, p, q, r)$

```
1  n_1 = q − p + 1
2  n_2 = r − q
3  let L[1..n_1 + 1] and R[1..n_2 + 1] be new arrays
4  for i = 1 to n_1
5      L[i] = A[p + i − 1]
6  for j = 1 to n_2
7      R[j] = A[q + j]
8  L[n_1 + 1] = ∞
9  R[n_2 + 1] = ∞
10 i = 1
11 j = 1
12 for k = p to r
13     if L[i] ≤ R[j]
14         A[k] = L[i]
15         i = i + 1
16     else A[k] = R[j]
17         j = j + 1
```

Complexity of the algo: $T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 \end{cases}$

$\Rightarrow \Theta(n \log n)$

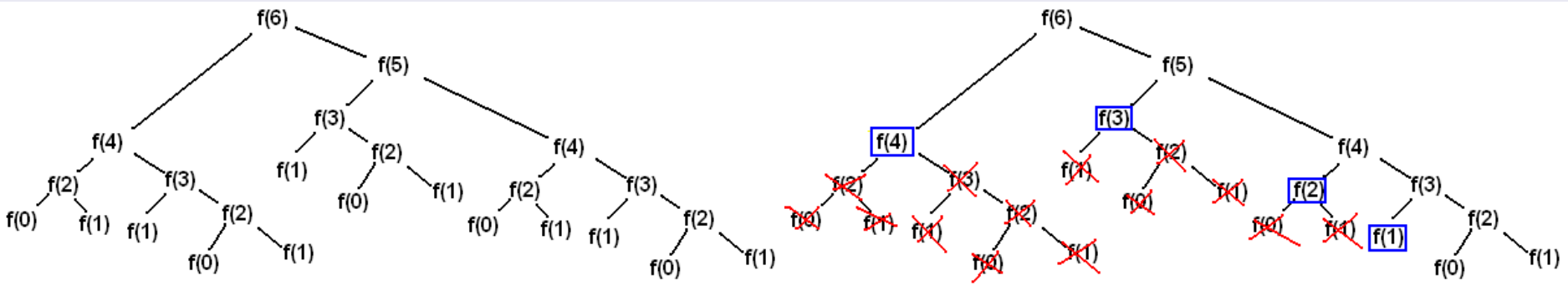
# Another example: the Fibonacci series

## Computing a Fibonacci number

$$Fib(n) = \begin{cases} \text{if } n = 0 \text{ or } n = 1 \text{ return}(1) \\ \text{otherwise return}(Fib(n-1) + Fib(n-2)) \end{cases}$$

Complexity: $u_n = u_{n-1} + u_{n-2} + 2 \Rightarrow O(1.62^n)$ Exponential!

## Idea: store the results of the calls (memorization): $\Rightarrow O(n)$




$$Fib(n) = \begin{cases} \text{if } n = 0 \text{ or } n = 1\text{: return}(1) \\ \text{if } Fib[n] \neq -1\text{: return}(Fib[n])) \\ \text{otherwise } Fib[n] = Fib[n-1] + Fib[n-2]); \text{return}(Fib[n]) \end{cases}$$

# Dynamic Programming: The Principles

We consider optimization problems a priori difficult to solve. The idea is to solve them by dividing the problem into subproblems and storing the results of each subproblems to make sure that the result is available in O(1) in case of a new call on the same subproblem.

## The 4 steps of a dynamic programming approach

1. Characterize the structure of an optimal solution
2. Recursively define the value associated to an optimal solution according to subproblems
3. Compute the value of an optimal solution by solving the subproblems in a bottom-up fashion (start from the smallest problems and build up solutions to larger and larger problems)
4. Build an optimal solution from computed information

# DP Example 1: matrix chain multiplication

- Find an optimal way to multiply a sequence of $n$ matrices $< A_1, \ldots, A_n >$
- Many solutions are possible: e.g. for $< A_1, A_2, A_3, A_4 >$: $(A_1(A_2(A_3A_4)))$, $(A_1((A_2A_3)A_4))$, $((A_1A_2)(A_3A_4))$, ...
- Recap on matrix multiplication: $A$ has dimension $p \times q$ and $B$ dimension $q \times r$, then the matrix $C : A \cdot B$ has dimension $p \times r$ and the simple algorithm has a complexity in $\Theta(pqr)$
- The multiplication order has some importance: consider 3 matrices $< A_1, A_2, A_3 >$ of dimensions $10 \times 100$, $100 \times 5$ and $5 \times 50$
  - $((A_1A_2)A_3)$ costs $10 * 100 * 5 + 10 * 5 * 50 = 7500$ multiplications
  - $(A_1(A_2A_3))$ costs $100 * 5 * 50 + 10 * 100 * 50 = 75000$ multiplications
- Considering all the possible parenthesizations is intractable $\Omega(4^n/n^{3/2})$ (Catalan number) $\Rightarrow$ use DP

# Matrix chain multiplication

## Formalization of the problem

**Input:** a sequence $< A_1, \ldots, A_n >$ of matrices to multiply, for any $i$ $A_i$ has dimension $p_{i-1} \times p_i$

**Output:** parenthesization of the matrix product minimizing the number of multiplications

# Step1: Characterizing an optimal substructure

Notations: $A_{i\ldots j}$ is the matrix solution of the product $A_i \times A_{i+1} \times \ldots \times A_j$ and $P(i \ldots j)$ be a parenthesization defined on this matrix sequence.

## Consider an optimal parenthesization $P_{opt}(1 \ldots n)$ of the sequence

- It divides the product into 2 subproducts at an index $k$:
  $A_1 \times \ldots \times A_n = (A_1 \times \ldots \times A_k) \times (A_{k+1} \times \ldots \times A_n)$
  cost=cost of $A_{1\ldots k}$+cost of $A_{k+1\ldots n}$+cost of $A_{1\ldots k} \times A_{k+1\ldots n}$
- Then $P(1 \ldots k)$ the parenthesization defined by $P_{opt}(1 \ldots n)$ over $< A_1 \ldots, A_k >$ must be optimal for this sequence
  **Proof:** by contradiction, assume there exists a strictly better $P'(1 \ldots k)$ for $< A_1, \ldots, A_k >$, then by replacing $P(1 \ldots k)$ by $P'(1 \ldots k)$ in $P_{opt}(1 \ldots n)$ we obtain a better solution, which contradicts that $P_{opt}(1 \ldots n)$ is optimal
  $\Rightarrow P(1 \ldots k)$ is optimal
- we can use the same argument for proving that $P(k+1 \ldots n)$ is optimal
- $\Rightarrow$ the optimal solution can be decomposed into optimal solutions of subproblems

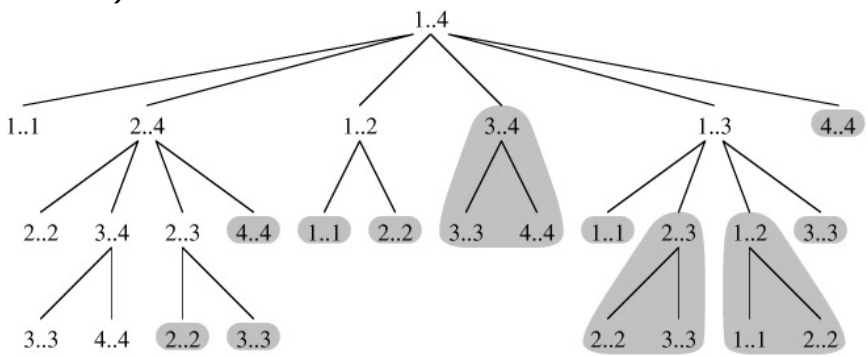
# Step2: computing recursively the optimal solution

## Optimal cost for $A_{i\ldots j}$ $1 \leq i \leq j < n$ stored in $m[i,j]$

- $$m[i,j] = \begin{cases} 0 & \text{if } i = j \\ \min_{i \leq k < j}(m[i,k] + m[k+1,j] + p_{i-1}p_k p_j) & \text{if } i < j \end{cases}$$

  $m[1,n]$ is thus the optimal parenthesization cost for the whole sequence
- We also store the optimal $k$ in $s[i,j]$ to be able to build the solution later (ie $m[i,j] = m[i,k] + m[k+1,j] + p_{i-1}p_k p_j$)

What are the number of subproblems: one for all $i,j$, $1 \leq i \leq j < n$ i.e. $\binom{n}{2} + n = \frac{n(n-1)}{2} + n : \Theta(n^2)$. Note the superposed subproblems in the recursive calls (good indication for DP)



# Step3: algorithm (bottom-up)

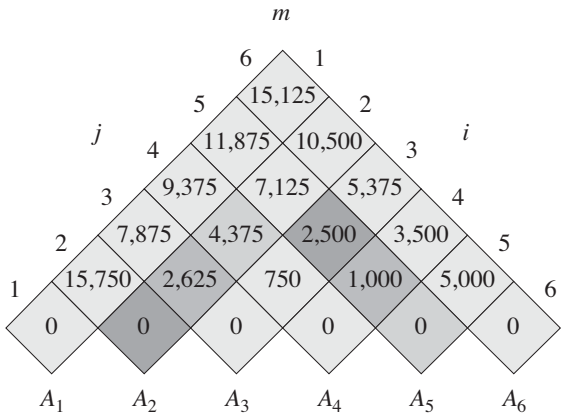
Principle: work on matrix subsequences with increasing length

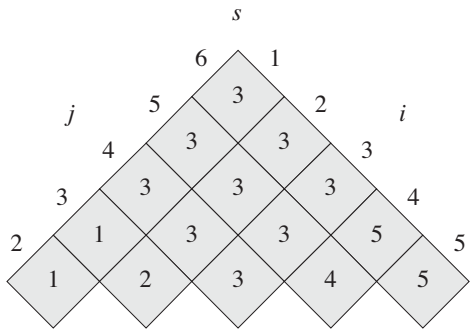
## Algo - complexity $\Theta(n^3)$ (and $\Theta(n^2)$ for storage)

MATRIX-CHAIN-ORDER$(p)$

```
1   n = p.length − 1
2   let m[1..n, 1..n] and s[1..n − 1, 2..n] be new tables
3   for i = 1 to n
4       m[i, i] = 0
5   for l = 2 to n                    // l is the chain length
6       for i = 1 to n − l + 1
7           j = i + l − 1
8           m[i, j] = ∞
9           for k = i to j − 1
10              q = m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
11              if q < m[i, j]
12                  m[i, j] = q
13                  s[i, j] = k
14  return m and s
```

# Example with 6 matrices $< A_1, \ldots, A_6 >$




| matrix | $A_1$ | $A_2$ | $A_3$ | $A_4$ | $A_5$ | $A_6$ |
|---|---|---|---|---|---|---|
| dimension | $30 \times 35$ | $35 \times 15$ | $15 \times 5$ | $5 \times 10$ | $10 \times 20$ | $20 \times 25$ |

In the this slide, tables are rotated such that the diagonal appears horizontally. The $m$ table uses the main diagonal and upper triangle and the $s$ table only the upper triangle. The optimal solution of the problem is $m[1,6] = 15,125$. Among the darker entries in the previous slide, the pairs with the same shading are taken together in line 10 when computing:

$$
\begin{aligned}
m[2,5] &= \min \begin{cases} m[2,2] + m[3,5] + p_1 p_2 p_5 &= 0 + 2500 + 35 \cdot 15 \cdot 20 &= 13{,}000\,, \\ m[2,3] + m[4,5] + p_1 p_3 p_5 &= 2625 + 1000 + 35 \cdot 5 \cdot 20 &= 7125\,, \\ m[2,4] + m[5,5] + p_1 p_4 p_5 &= 4375 + 0 + 35 \cdot 10 \cdot 20 &= 11{,}375 \end{cases} \\
&= 7125\,.
\end{aligned}
$$

# Step4: Constructing an optimal solution

```
PRINT-OPTIMAL-PARENS(s, i, j)
1  if i == j
2       print "A"_i
3  else print "("
4       PRINT-OPTIMAL-PARENS(s, i, s[i, j])
5       PRINT-OPTIMAL-PARENS(s, s[i, j] + 1, j)
6       print ")"
```

This algorithm applied on the previous problem outputs:

$$((A_1(A_2A_3))((A_4A_5)A_6))$$

# Another Example: Longest Common Subsequence (LCS)

## Definitions (sequence, subsequence)

- sequence of letters $X =< x_1, x_2, \ldots, x_n >$
- prefix $X_i =< x_1, \ldots, x_i >$ of size $i$ of $X$ ($X_0$ empty sequence)
- subsequence: original sequence without some of the letters
- common subsequence between two sequences $X$ and $Y$: subsequence both of $X$ and $Y$
- LCS: the longest common subsequence between $X$ and $Y$.
- Example
  - $Z =< A, C, G >$ is not a subsequence of $X < A, G, G, T, C, A >$, but $X' =< A, G >$ is a subsequence of X
  - if $Y =< T, C, A, G, A, T, A >$ then $Z' =< A, G, T, A >$ is a LCS of $X$ and $Y$.
- Applications: DNA sequence alignment, text comparisons, dictionary search

# The problem

## Longest Common Subsequence (LCS)

Input: $X =< x_1, \ldots, x_m >$ and $Y =< y_1, \ldots, y_n >$ 2 sequences

Output: $Z$ a LCS of $X$ and $Y$

# Step1: characterize an optimal solution

**Theorem: Let $Z =< z_1, \ldots, z_k >$ an LCS of $X$ and $Y$**

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and $Z_{k-1}$ is an LCS of $X_{m-1}$ and $Y_{n-1}$.
2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies $Z$ is an LCS of $X_{m-1}$ and $Y_n$.
3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies $Z$ is an LCS of $X_m$ and $Y_{n-1}$.

**Proof:** (1) If $z_k \neq x_m$, then we could append $x_m = y_n$ to $Z$ to obtain a common subsequence of $X$ and $Y$ of length $k+1$, contradicting the supposition that $Z$ is a longest common subsequence of $X$ and $Y$. Thus, we must have $z_k = x_m = y_n$. Now, the prefix $Z_{k-1}$ is a length-$(k-1)$ common subsequence of $X_{m-1}$ and $Y_{n-1}$. We wish to show if it is an LCS. Suppose for the purpose of contradiction that there exists a common subsequence $W$ of $X_{m-1}$ and $Y_{n-1}$ with length greater than $k-1$. Then, appending $x_m = y_n$ to $W$ produces a common subsequence of $X$ and $Y$ whose length is greater than $k$, which is a contradiction.

# Step1: characterize an optimal solution

**Theorem: Let $Z =< z_1, \ldots, z_k >$ an LCS of $X$ and $Y$**

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and $Z_{k-1}$ is an LCS of $X_{m-1}$ and $Y_{n-1}$.
2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies $Z$ is an LCS of $X_{m-1}$ and $Y_n$.
3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies $Z$ is an LCS of $X_m$ and $Y_{n-1}$.

**Proof (ctd):**
(2) If $z_k \neq x_m$, then $Z$ is a common subsequence of $X_{m-1}$ and $Y$. If there was a common subsequence $W$ of $X_{m-1}$ and $Y_n$ with length greater than $k$, then $W$ would also be a common subsequence of $X_m$ and $Y_n$, contradicting the assumption that $Z$ is an LCS of $X$ and $Y$.
(3) The proof is symmetric to (2). □

- $\Rightarrow$ An LCS of two sequences contains within it an LCS of prefixes of the two sequences
- $\Rightarrow$ LCS has an optimal-substructure property
- $\Rightarrow$ We need then to define a recursive solution with overlapping subproblems.

# Step 2: A recursive solution

- We need to take into account the 3 parts of the previous theorem and take into account limit cases (empty strings).
- Idea: Let $c[i,j]$ be the length of an LCS of the prefix sequences $X_i$ and $Y_j$.
  We can deduce the following recursive formula:

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ c[i-1,j-1]+1 & \text{if } i,j > 0 \text{ and } x_i = y_j, \\ max(c[i,j-1], c[i-1,j]) & \text{if } i,j > 0 \text{ and } x_i \neq y_j. \end{cases}$$

$\Rightarrow$ The considered subproblems are based on properties of the original problem (some of them are ruled out by these conditions).

# Step 3: algorithm

$c[i,j]$ table storing the LCS between prefixes, we use an auxiliary table to $b[i,j]$ to build an optimal solution, the arrows indicate which preceding subproblem is used to compute the current solution.

LCS-LENGTH$(X, Y)$

```
1  m = X.length
2  n = Y.length
3  let b[1..m, 1..n] and c[0..m, 0..n] be new tables
4  for i = 1 to m
5      c[i, 0] = 0
6  for j = 0 to n
7      c[0, j] = 0
8  for i = 1 to m
9      for j = 1 to n
10         if x_i == y_j
11             c[i, j] = c[i − 1, j − 1] + 1
12             b[i, j] = "↖"
13         elseif c[i − 1, j] ≥ c[i, j − 1]
14             c[i, j] = c[i − 1, j]
15             b[i, j] = "↑"
16         else c[i, j] = c[i, j − 1]
17             b[i, j] = "←"
18 return c and b
```

# Example

Compute the LCS between $X =< A, B, C, B, D, A, B >$ and $Y =< B, D, C, A, B, A >$.

## Tables $b$ and $c$ produced

| $i$ | | $j$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|---|---|
| | | $y_j$ | | $B$ | $D$ | $C$ | $A$ | $B$ | $A$ |
| 0 | $x_i$ | | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | $A$ | | 0 | ↑ 0 | ↑ 0 | ↑ 0 | ↖ 1 | ← 1 | ↖ 1 |
| 2 | $B$ | | 0 | ↖ 1 | ← 1 | ← 1 | ↑ 1 | ↖ 2 | ← 2 |
| 3 | $C$ | | 0 | ↑ 1 | ↑ 1 | ↖ 2 | ← 2 | ↑ 2 | ↑ 2 |
| 4 | $B$ | | 0 | ↖ 1 | ↑ 1 | ↑ 2 | ↑ 2 | ↖ 3 | ← 3 |
| 5 | $D$ | | 0 | ↑ 1 | ↖ 2 | ↑ 2 | ↑ 2 | ↑ 3 | ↑ 3 |
| 6 | $A$ | | 0 | ↑ 1 | ↑ 2 | ↑ 2 | ↖ 3 | ↑ 3 | ↖ 4 |
| 7 | $B$ | | 0 | ↖ 1 | ↑ 2 | ↑ 2 | ↑ 3 | ↖ 4 | ↑ 4 |

One possible LCS is $< B, C, B, A >$ of length 4, the shaded entries illustrate how the sequence is obtained. Is there another solution?

# Step 4: building an optimal solution

We use the matrix $b$, the initial call is PRINT-LCS($b, X, X.length, Y.length$).

```
PRINT-LCS(b, X, i, j)
1  if i == 0 or j == 0
2      return
3  if b[i, j] == "↖"
4      PRINT-LCS(b, X, i − 1, j − 1)
5      print x_i
6  elseif b[i, j] == "↑"
7      PRINT-LCS(b, X, i − 1, j)
8  else PRINT-LCS(b, X, i, j − 1)
```

## Complexity analysis

- The algo LCS-LENGTH is $\Theta(mn)$ (each entry of the $c$ tab is computed in $O(1)$)
- The PRINT-LCS procedure is $O(m + n)$

# Dynamic programming - Summary and Take home message

To solve optimization problems by dynamic prog., 2 properties are necessary:

- Optimal substructure property: an optimal solution can be decomposed into optimal solutions of sub-problems. This property can be shown by using proofs by contradiction.
  Sometimes, there exist many ways to arrange the solution of a problem according to subproblems, the best solution is the one leading to the lowest complexity. There exists no general principle for doing it, "just" try to have a natural expression of the subproblem ;-).
- Overlapping subproblems property: the subproblems must overlap, *i.e.* be called many times to solve the original problem. If this is not the case, a divide and conquer approach is more relevant.
- In the examples we saw, we used a bottom-up approach which implies in fact that we are able to define "the bottom" - *i.e.* where to begin the algorithm. Sometimes it is not possible or not easy (chess game for example), in this case we use a top-down procedure by starting from the big original problem and call the subproblems: we check if a subproblem has already been solved, if it's not the case you solve it recursively otherwise you take the available solution from a (relevant) table.
- We can also solve problems that are not optimization pbs (*e.g.* Fibonacci).

# A remark on the optimality principle

## DP relies on the Optimality principle from Richard Bellman

*Every sub-policy of an optimal policy is also optimal*.
From applying dynamic programming we take the hypothesis that: *The optimal solution of a problem can be obtained by the optimal solution from the optimal solutions of the associated sub-problems.*

Even if the proofs made in these lectures may appear "trivial", this is not always possible/true.
Let us consider two different problems of path search in an unweighed oriented graph:

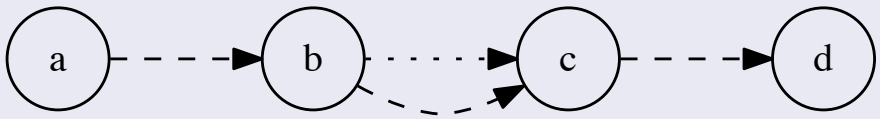
- Shortest path problem $\rightarrow$ the optimality principle holds
- Longest path problem $\rightarrow$ the optimality principle **do not hold**

# Shortest path in a oriented graph

- Consider an unweighted oriented graph $G$ and the problem of computing the shortest path between 2 nodes.
- **Claim**: A path is of minimal length if all its subpaths are of minimal length

## Proof by contradiction

- Consider a shortest path between 2 nodes a and d (dashed line) that goes threw two other nodes b and c.
  Let $[a - d]$ the length of the path between a and d on this path.

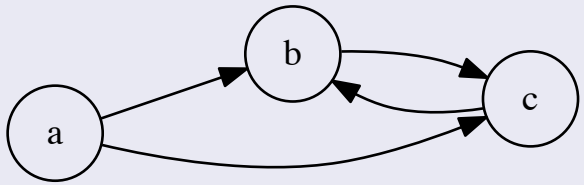



- The length of $[a - d] = [a - b] + [b - c] + [c - d]$.
- Suppose there exists one subpath that is not optimal, *e.g.* $[b - c]$
- Then there exists another subpath between b and c of smaller size (dotted line), by using this path between b and c, we can obtain a path shorter path than $[a - d]$ which leads to a contradiction with respect to the optimality of $[a - d]$.

$\Rightarrow$ Classic proof

# Longest path in an oriented graph (without loops)

- Consider an unweighted oriented graph $G$ with loops and the problem of computing the longest path between 2 nodes.
- Consider the graph below and two nodes a and c




- The longest path (without loop) between a and c is of length 2 ($a \rightarrow b \rightarrow c$), the longest path between a and b is also of length 2 ($a \rightarrow c \rightarrow b$), and the longest one between b and c is of length 1.
- The longest path between a and c can then not be obtained by the composition of longest subpaths since the longest path between a and b is of size 2 !
- The optimality principle is not valid in this case.

Note: this does not mean that applying dynamic programming is totally impossible, because one could imagine another strategy for modeling the problem. However, this may be a good indicator that applying dynamic programming is at least difficult (or even impossible) and then we may consider branch and bound approaches.

# Greedy algorithms

# Principle

Optimization problems are generally solved through a sequence of steps with a set of choices at each step.
Dynamic programming approaches consider all the possible solutions but deal with the different choices in a neat way to define a not too complex algorithm (e.g. polynomial).

A **greedy algorithm** always makes the choice that looks best at the moment, this is called a *local optimal choice*. Sometimes, this strategy will lead to a global optimal solution but this is not always the case in general. One advantage of such algorithms is to provide an efficient procedure.

# Example 1: Activity selection problem (interval scheduling)

## Problem definition

**Input:** A set $S = \{a_1, \ldots, a_n\}$ of activities. Each activity has a start time $s_i$ and a finish time $f_i$ $(0 \leq s_i < f_i < \infty)$.

**Output:** The largest subset $S' \subseteq S$ of compatible activities. Two activities $a_i$ and $a_j$ are compatible if their time interval do not overlap, ie: $f_j \leq s_i$ or $f_i \leq s_j$.

Applications: memory allocation, supply chain management, scheduling a room for multiple concurrent events, ...

To work on this problem, we will consider a set of 11 activities, defined in the table below and **sorted in monotonically increasing order of finish time**

| $i$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $s_i$ | 1 | 3 | 0 | 5 | 3 | 5 | 6 | 8 | 8 | 2 | 12 |
| $f_i$ | 4 | 5 | 6 | 7 | 9 | 9 | 10 | 11 | 12 | 14 | 16 |

# Solving the problem

- We need to define a greedy choice: *select the activity compatible with the previously selected activities and having the earliest finish time*.
- This choice leaves the resource available for as many other activities as possible.

```
GREEDY-ACTIVITY-SELECTOR(s, f)
1  n = s.length
2  A = {a_1}
3  k = 1
4  for m = 2 to n
5      if s[m] ≥ f[k]
6          A = A ∪ {a_m}
7          k = m
8  return A
```

Complexity: $O(n \log n)$ (we need to sort), and $\Theta(n)$ if the activities are already sorted

Result on the previous 11 activity set: $A = \{1, 4, 8, 11\}$.

Question: Does this algorithm always output an optimal solution?

# Proving optimality for Greedy Algorithms

Two possible solutions

## "Greedy stays ahead" arguments

In a context where the optimal structure property of the considered problem has been justified, we prove that the 1st choice made by the greedy algo is optimal.

## Exchange arguments

Show that you can iteratively transform an optimal solution into the solution produced by the greedy algorithm without modifying the cost of the optimal solution.

1st method is used in slides, second in some exercises.

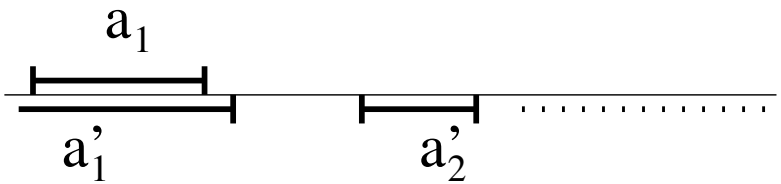
# Optimality

## Theorem

The solution found by the algorithm is optimal.

**Proof:** (i) we prove that $a_1$ belongs to an optimal solution. Let $A = \{a'_1, \ldots, a'_k\}$ be an optimal solution with activities ordered such that $a'_1 < \ldots < a'_n$. If $a_1 = a'_1$ we are done. Otherwise we have:

- $f_{a_1} < f_{a'_1}$ by definition of $a_1$
- $f_{a'_1} \leq s_{a'_j}$ $\forall j$, because the activities of the solution are compatible




Thus $a_1$ is compatible with $a'_2, \ldots, a'_k$ and $B = A \cup \{a_1\} - \{a'_1\}$ is a solution with the same size as $A$, thus $B$ is optimal and contains $a_1$

# Optimality

**Theorem**

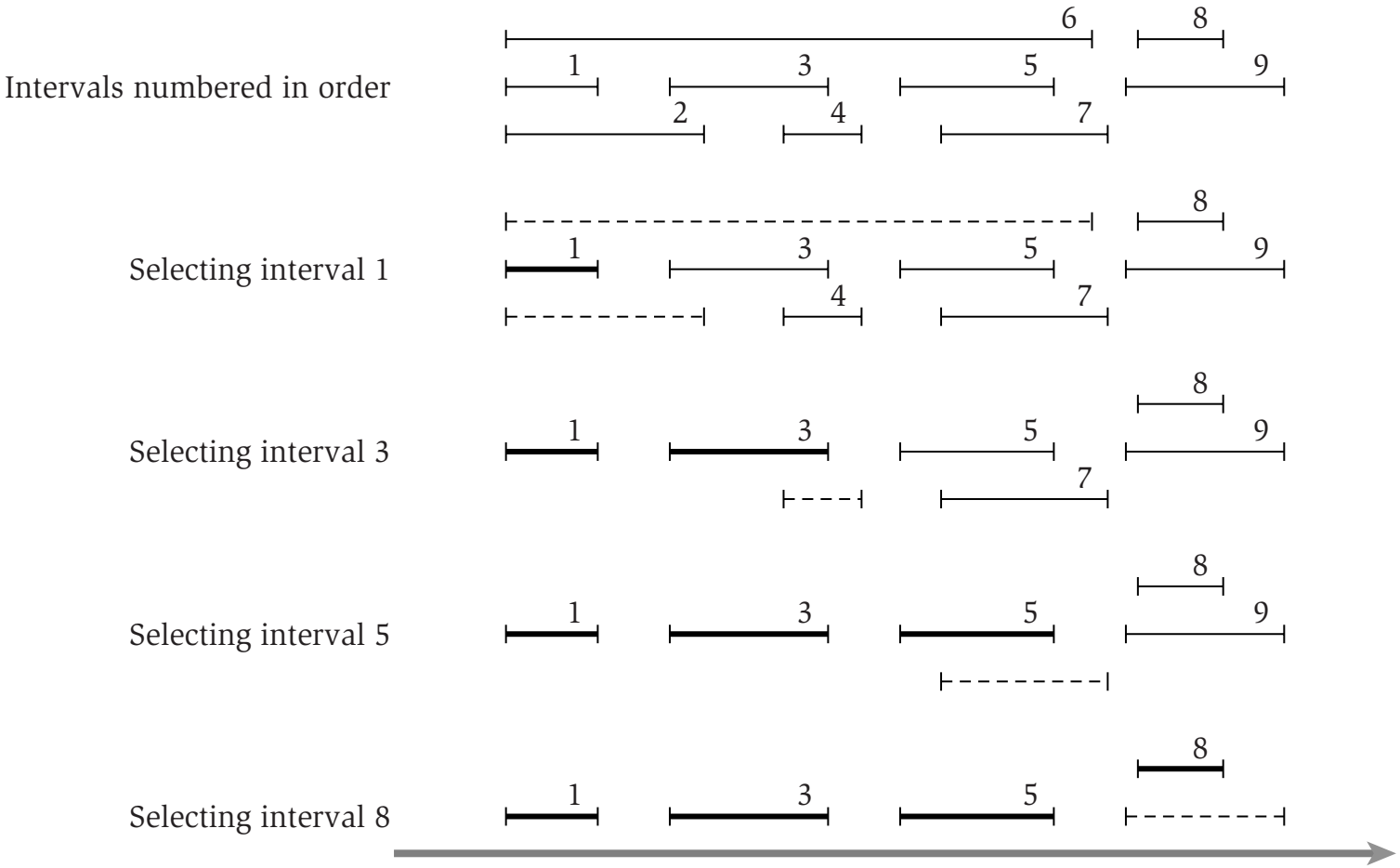
The solution found by the algorithm is optimal

**Proof: (ctd)**

(ii) Let $S_i$ be the subproblem restricted to the activities $a_i$ to $a_n$ and let $A_i$ be an optimal solution for this subproblem. Clearly, if $a$ is chosen by the algorithm at this step, then $A'_i = A_i - \{a\}$ is optimal for $S_{i+1}$:
by contradiction assume $B'$ an optimal solution for $S_{i+1}$ and $|B'| > |A'_i|$, then $B = B' \cup \{a\}$ is optimal for $S_i$ with $|B| > |A_i| \rightarrow$ contradiction.$\square$

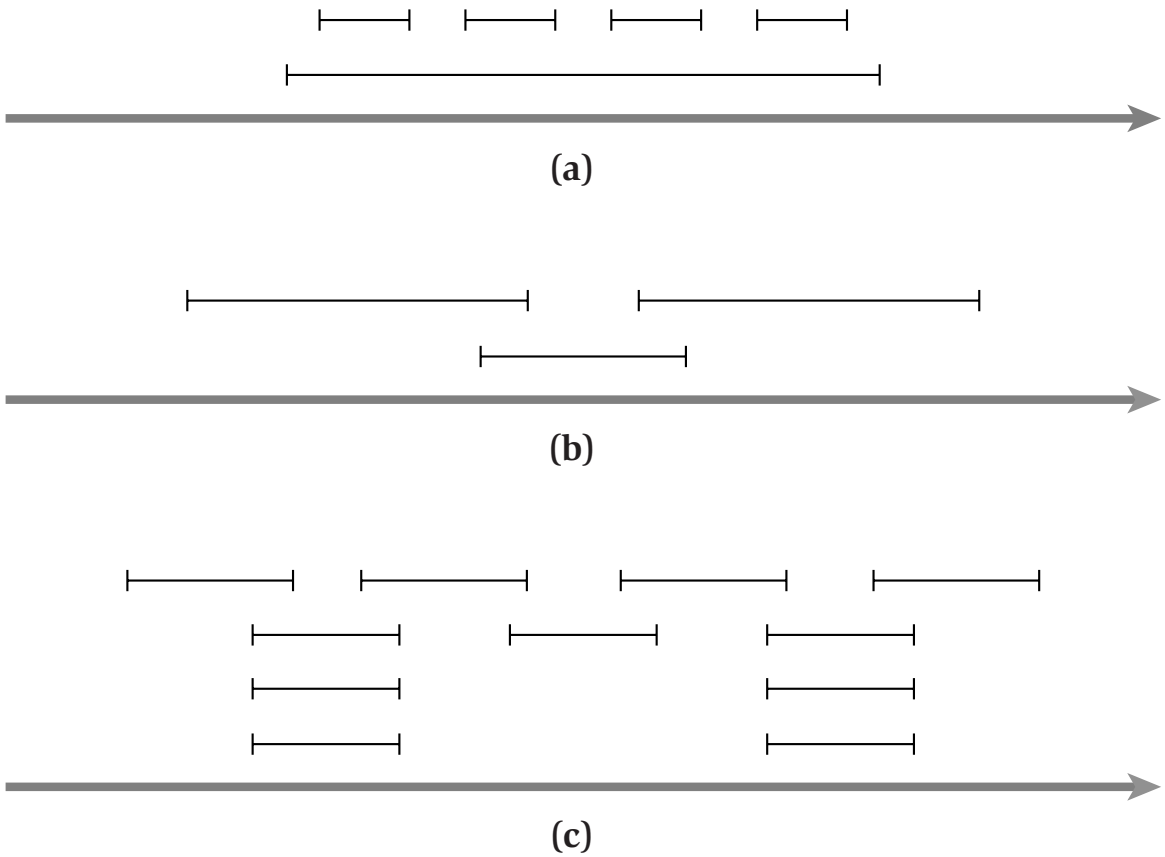
$\Rightarrow a_1$ belongs to an optimal solution and the solutions selected for each subproblem are also in an optimal solution $\rightarrow$ the algo provides an optimal solution.

# Example




Sample run of the activity selection (interval scheduling) Algorithm. At each step the selected intervals are darker lines, and the intervals deleted at the corresponding step are indicated with dashed lines.

# Why do we need to take the finish time?




Some instances of the activity selection problem on which natural greedy algorithms fail to find the optimal solution. In (a), it does not work to select the interval that starts earliest; in (b), it does not work to select the shortest interval; and in (c), it does not work to select the interval with the fewest conflicts.

# What about dynamic programming?

The previous analysis can be seen as a "dynamic programming" one where at each step we have only one subproblem to solve (which is not really relevant for DP). We can make another analysis that really stands into DP scheme but that is unfortunately not so efficient as the greedy approach.

Let $S_{ij}$ be the set of activities starting after $a_i$ and finishing before $a_j$ starts. Let $A_{ij}$ be the maximum set of mutually compatible activities in $S_{ij}$. Assume $a_k \in A_{ij}$ ($a_k$ belongs to an optimal solution) then we have two subproblems $S_{ik}$ and $S_{kj}$, $A_{ik} = A_{ij} \cap S_{ik}$ and $A_{kj} = A_{ij} \cap S_{kj}$. Thus $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$ and we can easily show that $A_{ik}$ and $A_{kj}$ must be optimal solutions for $S_{ik}$ and $S_{kj}$.

Then, the recursive definition of an optimal solution is given by $c[i,j] = c[i,k] + c[k,j] + 1$, which leads to the following procedure:

$$c[i,j] = \begin{cases} 0 & \text{if } S_{ij} = \emptyset\ , \\ \max\limits_{a_k \in S_{ij}} \{c[i,k] + c[k,j] + 1\} & \text{if } S_{ij} \neq \emptyset\ . \end{cases}$$

(note that $c[i,i] = c[i,i+1] = 0$).
But complexity in $\Omega(n^2)$ and $O(n^3)$ !! (other formulations allow one to get $O(n^2)$)

# Another example: Huffman code and data compression

We can represent any encoding of a sequence by a tree: The leaves correspond to the letters to encode and the encoding of a letter is obtained by the sequence of labels of the edges from the root to the leaf. Let $d_T(c)$ be the length of the encoding of a letter $c$, which corresponds to its depth in the encoding tree $T$.

## The problem

**Input:** a sequence of letters from an alphabet $C$, each letter has a frequency $f(c)$

**Output:** a binary encoding for each $c \in C$ of minimal cost, *i.e.* requiring the minimum number of bits to encode the input sequence:

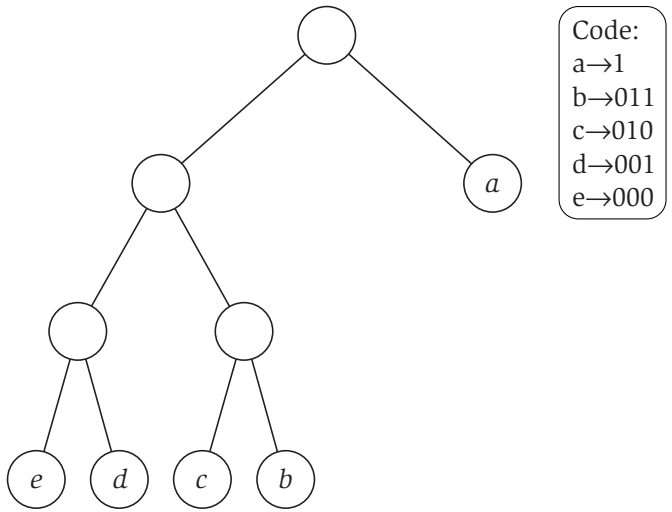
$$B(T) = \sum_{c \in C} f(c) * d_T(c).$$

# Encoding

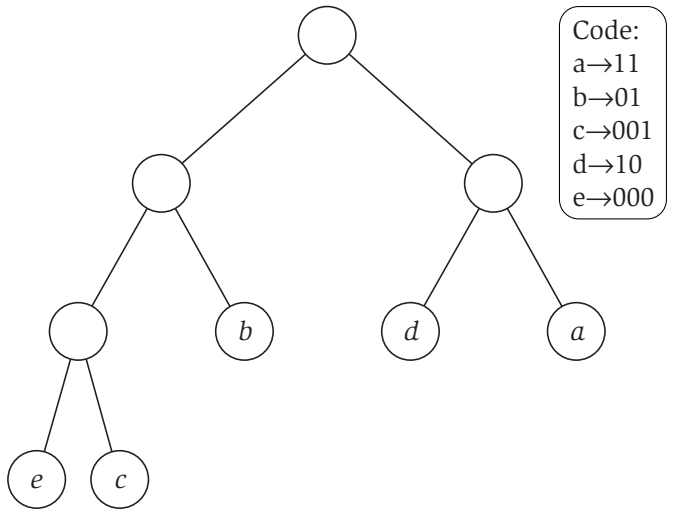
Let $C = \{a, b, c, d\}$, consider the following encoding:
$\gamma(a) = 11, \gamma(b) = 01, \gamma(c) = 001, \gamma(d) = 10, \gamma(e) = 000$.
Then *cecab* is encoded as 0010000011101.

Consider a text with frequencies:
$f(a) = 32, f(b) = 25, f(c) = 20, f(d) = 18, f(e) = 5$, using the prefix code we have need $32 * 2 + 25 * 2 + 20 * 3 + 18 * 2 + 5 * 3 = 225$ bits to encode the text.
Using a fixed-length encoding, we would need at least 3 bits per letter, and the 300 bits to encode the text.
There is a better encoding code for this text:
$\gamma(a) = 11, \gamma(b) = 10, \gamma(c) = 01, \gamma(d) = 001, \gamma(e) = 000 \rightarrow 223$ bits.

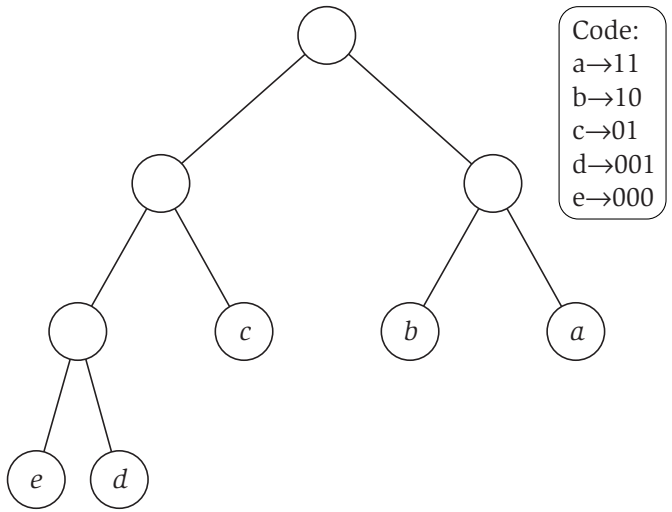
The idea: we will use variable-length codes and to simplify (to avoid ambiguity problems) we consider prefix codes: no codeword is also a prefix of some other codeword (do you see why?). No loss of generality: a prefix code can always achieve the optimal data compression among any character code.


Code:
a→1
b→011
c→010
d→001
e→000
a
e
d
c
b

**(a)**


Code:
a→11
b→01
c→001
d→10
e→000
b
d
a
e
c

**(b)**


Code:
a→11
b→10
c→01
d→001
e→000
c
b
a
e
d

**(c)**

# Algorithm

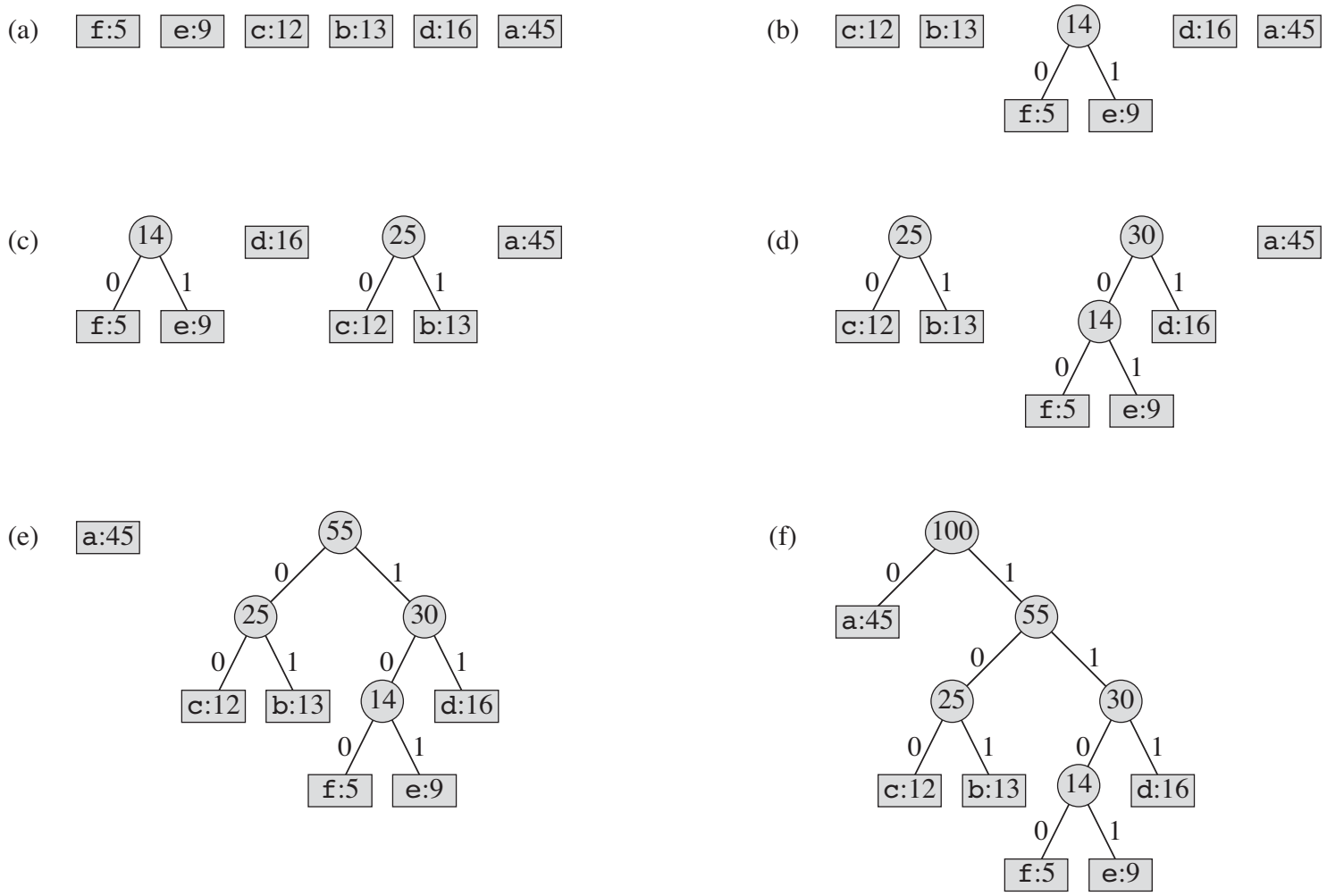
Principle:

- build a leaf for each letter $c$ and associate the value $f(c)$
- Repeat until there remains only one node: merge the two leaves with the lowest frequencies in a node, this node takes the sum of the two leave frequencies as a value.

```
HUFFMAN(C)
1  n = |C|
2  Q = C
3  for i = 1 to n − 1
4       allocate a new node z
5       z.left = x = EXTRACT-MIN(Q)
6       z.right = y = EXTRACT-MIN(Q)
7       z.freq = x.freq + y.freq
8       INSERT(Q, z)
9  return EXTRACT-MIN(Q)      // return the root of the tree
```

We use a min-priority queue (based on the frequencies) to identify the two least frequent objects to merge together.

# Example




Using a good data structure (binary heap) EXTRACT-MIN and INSERT require $O(\log n)$, line 2 requires $O(n) \Rightarrow O(n \log n)$.

# Optimality

## Greedy choice property

Let $x, y \in C$ the two least frequent symbols, there exists an optimal prefix encoding in which the codewords of $x$ and $y$ have the same length and differ only in the last bit. ($x$ and $y$ are two sons of the same node in an optimal tree)

**Proof** Let $T$ be an optimal tree and let $a$ and $b$ two leaves of maximum depth having the same father node. By assumption $f(x) \leq f(a)$ and $f(y) \leq f(b)$, thus by considering the tree $T'$ obtained by exchanging $x$ and $a$, we have by definition of $x$ and $a$:

$$\begin{aligned} B(T) - B(T') &= \sum_{c \in C} f(c) * d_T(c) - \sum_{c \in C} f(c) * d_{T'}(c) \\ &= f(x)d_T(x) + f(a)d_T(a) - f(x) * d_{T'}(x) - f(a)d_{T'}(a) \\ &= f(x)d_T(x) + f(a)d_T(a) - f(x) * d_T(a) - f(a)d_T(x) \\ &= (f(a) - f(x))(d_T(a) - d_T(x)) \geq 0 \end{aligned}$$

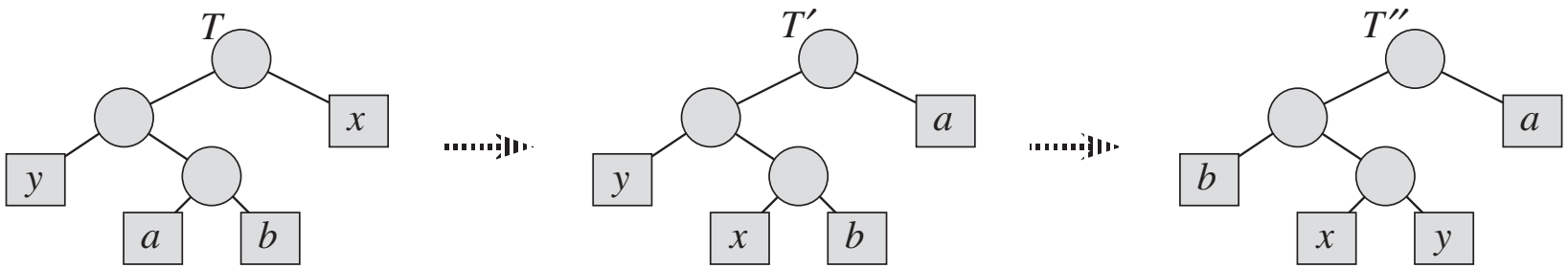
# Optimality (2)

## Greedy choice property

Let $x, y \in C$ the two least frequent symbols, there exists an optimal prefix encoding in which the codewords of $x$ and $y$ have the same length and differ only in the last bit. ($x$ and $y$ are two sons of the same node in an optimal tree)

**Proof (ctd):**

We use the same argument for $T''$ in which $y$ and $b$ have been exchanged: the cost of the tree does not increase $\rightarrow$ there exists an optimal encoding having $x$ and $y$ as maximum depth leaves in the corresponding tree.



# Optimal substructure

The next result shows that the problem of constructing optimal prefix codes has the optimal-substructure property.

Let $T$ be a tree of an optimal encoding for C, $x, y$ two maximum depth leaves of the same father $z$. If $f(z) = f(x) + f(y)$ then $T' = T \backslash \{x, y\} \cup \{z\}$ defines an optimal encoding for $C' = C \backslash \{x, y\} \cup \{z\}$.

**Proof:** We have for any $c \notin \{x, y\} \in C$: $f(c)d_T(c) = f(c)d_{T'}(c)$ and $d_T(x) = d_T(y) = d_{T'}(z) + 1$. Thus,
$f(x)d_T(x) + f(y)d_T(y) = (f(x) + f(y))(d_{T'}(z) + 1) = f(z)d_{T'}(z) + f(x) + f(y)$ and $B(T) = B(T') + f(x) + f(y)$.
If $T'$ is not optimal, then there exists $T''$ in which the leaves are the symbols of $C''$ s.t. $B(T'') < B(T')$. By adding $x, y$ to the leaf $z$ of $T''$, we have a prefix encoding for $C$ with cost $B(T'') + f(x) + f(y) < B(T)$, which contradicts that $T$ is optimal $\rightarrow$ $T'$ is necessarily optimal for $T'$.$\square$

$\Rightarrow$ By combining the two results, we have that the Huffman algorithm produces an optimal prefix code.

# Greedy algorithms - Summary and Take home message

To solve an optimization problem **exactly** by a greedy approach, two properties are necessary:

- Greedy choice property: the local choice made at each step of the algorithm belongs to an optimal solution.
- Optimal substructure property: the optimal solution can be decomposed into optimal solution of a subproblem
- A greedy algorithm must have a "small complexity", local greedy choices must be done in constant time (possibly) or in linear time (in the most general cases).

# Branch and Bound and Approximation methods

# How to deal with NP-hard problems?

- Some problems cannot be solve in polynomial time (NP-hard problems), but sometimes we really need to find a (good) solution (TSP, Knapsack, ...)
- To have an exact solution, we have to use techniques that may be exponential in the worst case
- The naive brute-force-search approach consists in enumerating all the possible solutions and finally output the best one.
- To use such an approach, one must at least take care of some important points:
  - we need a procedure able to enumerate all the solutions, without proposing the same one many times (more than once)
  - we need a procedure able to build quickly a new solution from a previous one
  - we wish that the enumeration would be factorizable, which means that some solutions share a common part that does not have to be recomputed at each step.
- The other way is to look for approximate solutions thanks to some heuristics or particular settings

# Enumeration of the solutions

## To explore the solution space

- we need a method for building a solution in order to explore the solution space by following a tree structure: when a solution is not optimal, we come back (**backtrack**) to the previous choice point and take another branch of the tree $\rightarrow$ we explore a new solution set.
- Or, we could define a graph on the whole set of solutions and look for an Hamiltonian path in this graph!

## Problem: Coloring the nodes of a graph

Input: a graph $G = (V, E)$ and an integer $k$

Question: Is there a coloring of $G$ using at most $k$ colors such that for two vertices linked by an edge do not have the same color: $\forall v \in V$, $1 \leq color(v) \leq k$ and $\nexists (v, v') \in E$ s.t. $color(v) = color(v')$?

A solution is represented by a list of colors, one for each node in the graph, ie a string of size $n$ (assuming $|V| = n$) on the alphabet $\{1, \ldots, k\}$

# An example for the graph coloring problem

A way to enumerate the solutions is given by the following (simple) algo:

**Input**: $G = (V, E)$, $k$: number of colors to use, *color*: (partial) list of the colors of the nodes, $i$ is the index of the first node of $V$ not colored.
**Output**: `true` or `false`

```
if i > |V| then
    return true
else
    foreach c from 1 to k do
        color(i) ← c;
        ok ←true;
        foreach (i, j) ∈ E s.t. j ≤ i do
            if color(i) = color(j) then
                ok ← false; break out from the loop;
        if ok then
            if TryTestColoring(G, k, color, i + 1) is true then
                return true;
    return false;
```

## Using an Hamiltonian path

$V = \{C_i$ string of size $n$ over $\{1 \dots k\}\}$,
$E = \{(C_i, C_j)$ s.t. $C_i$ and $C_j$ give the same color to all the nodes of $V$ except $1\}$
Question: when using an Hamiltonian path how to generate a new solution from another one in $O(1)$ without considering twice the same permutation?

## Using these approaches is not possible for some problems

- *Binpacking:* there is an infinite number of placement for each object (w.r.t. to the decimal precision)
- *TSP:* when $n > 100$ "codable" but hardly tractable

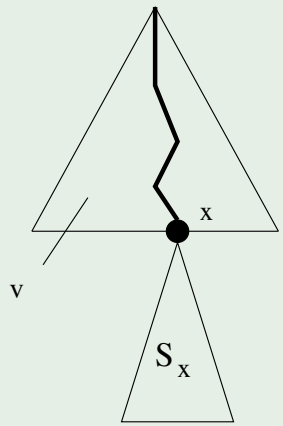
# The Branch and bound approach

- We explore the solution using a tree structure, but we cut some branches of the tree to reduce the search space of the solutions.
- We define $g(sol)$ the function we want to optimize (we assume we are looking for a minimum over $g$). Its definition depends on the problem considered. For example in the TSP problem it corresponds to the length of the cycle defined by the solution.

# The Branch and bound approach

## Principle

- we define a function (algorithm) associating quickly (polynomial time) a value $f(x)$ to each node $x$ in the search tree. It gives the best possible value for any solution belonging to the subtree rooted at $x$. **For the root $x$ we look for a lower bound for the solutions in the subtree rooted by $x$:** $\forall y \in S_x, f(x) < f(y)$.




- assume that at a given step, the best solution found has a value $v$, then in the remaining part of the tree it is not necessary to check subtrees rooted in $x$ when $v < f(x)$.

# The Branch and bound approach

In practice, a greedy heuristic (fast) is used to find a first $v$ close to the optimal in order to cut quickly a lot of branches in the search tree.

The quality of this approach depends on how well $f$ can approximate $g$.

Note: $f$ is a heuristic, because if we would have known how to find $g(x)$ in polynomial time, we could solve the problem in polynomial time!

# Example with TSP

We use a search tree: a path in the tree represents a sequence of nodes to visit, during a top-down search we obtain partial solutions and we add new nodes during the search.

We define a partial solution as: $(k, [i_1, \ldots, i_k], w)$ where $w$ is the cost associated to partial solution.

$S_{partial} = (k, [i_1, \ldots, i_k], w)$;
$best = (n, [i_{1_b}, \ldots, i_{n_b}], w_b)$;
**begin**
  **if** $k = n$ **then**
    $cost \leftarrow w + A[i_k, i_1]$;
    **if** $cost < w_b$ **then**
      $best = (k, [i_1, \ldots, i_k], cost)$;
  **else**
    **foreach** $j \notin [i_1, \ldots, i_k]$ **do**
      $N_s = (k+1, [i_1, \ldots, i_k, j], w + A[i_k, j]$;
      $TSP - naive(N_s, best)$;
**end**

Algorithme 1: $TSP - naive(S_{partial}, best)$

# The Branch and bound solution !

$S_{partial} = (k, [i_1, \ldots, i_k], w)$;
$best = (n, [i_{1_b}, \ldots, i_{n_b}], w_b)$;
**begin**
  **if** $k = n$ **then**
    $cost \leftarrow w + A[i_k, i_1]$;
    **if** $cost < w_b$ **then**
      $best = (k, [i_1, \ldots, i_k], cost)$;
  **else**
    **foreach** $j \notin [i_1, \ldots, i_k]$ **do**
      $cost \leftarrow w + A[i_k, j]$;
      **if** $cost < w_b$ **then**
        $N_s = (k+1, [i_1, \ldots, i_k, j], w + A[i_k, j]$;
        $TSP - branch - naive(N_s, best)$;
**end**

Algorithme 2: $TSP - branch - naive(S_{partial}, best)$

# Another problem: Task assignment

## Problem

We consider the following task assignment problem: we have $n$ agents and $n$ tasks and the problem is to associate a task to each agent in order to minimize the global cost. Indeed, each agent performs differently on the different tasks, and we assume to have a table $c$ indicating for each agent the cost associated to each task. An example of a cost table is provided in Table 1.

| Agent\Task | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 5 | 4 | 13 | 8 |
| 2 | 6 | 1 | 7 | 11 |
| 3 | 8 | 6 | 8 | 7 |
| 4 | 9 | 4 | 6 | 11 |

Table 3: Example of a cost table. The value 6 at position $c[2, 1]$ indicates that for agent number 2 the cost of achieving task 1 is 6.

# Question 1: How to encode a solution?

We can represent a solution as a vector where the first attribute is the number of the task affected to the 1st agent, the second attribute is the task affected to the second agent, and so on.

Given $n$ tasks and $n$ agents, the solutions are the set of permutations of $n$ elements.

Given 4 tasks/agents, a partial solution can be represented as $(2, 3, ?, ?)$ meaning that the first agent receives task 2, the second agent task 3, agents 3 and 4 do not have any task at the moment.

# Q2: Evaluation function?

Now, we would like to define an evaluation function $f$.
We want this function $f$ to be decomposed in two terms: $g^*$ and $h$, where

- $g^*$ corresponds to the real cost of the partial solution found so far
- and $h$ is a lower bound on the expected cost for the remaining tasks that have to selected in the solution.

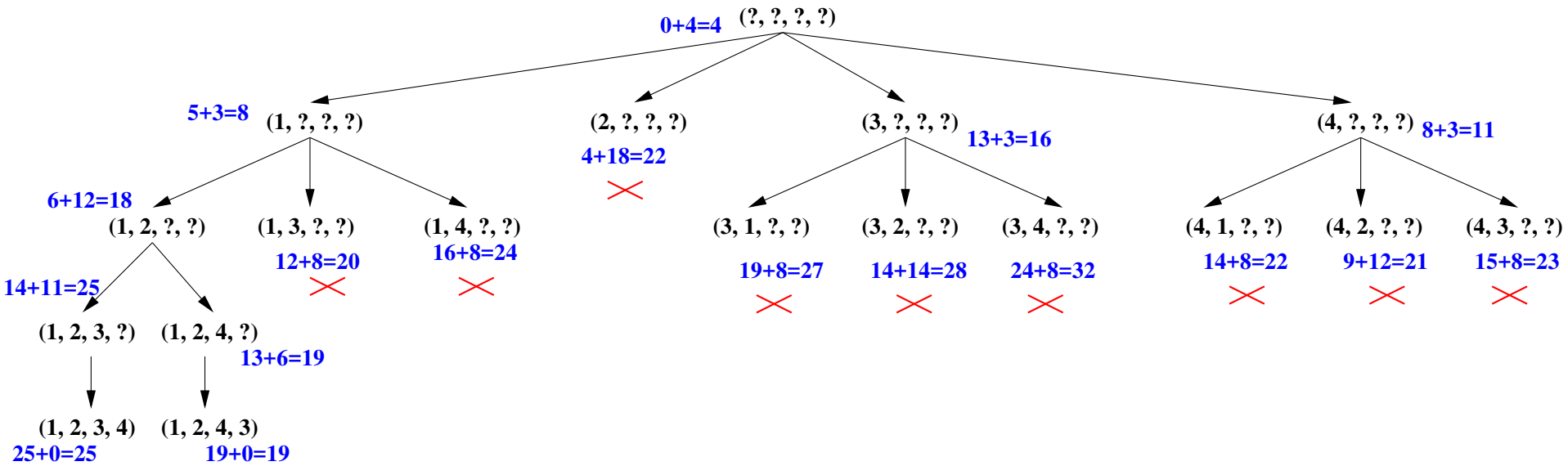
# Q2: cost evaluation

Imagine that we have currently affected a task for $k$ agents out of $n$, for a corresponding solution vector $\mathbf{v}$, we can define the following quantities:

- $g^*(\mathbf{v}) = \sum_{i=1}^{k} c[1, \mathbf{v}[1]]$
  the sum of the costs of the tasks affected to the first $k$ agents
- $$h(\mathbf{v}) = (n-k) \min_{\substack{k<i\leq n \\ 1\leq j\leq n \\ j\neq \mathbf{v}[l] \text{ for } 1\leq l\leq k}} c[i,j]$$

  $(n-k)$ times the minimum possible values among remaining tasks and agents

# Q2: Search Tree




Each node defines a (partial) solution written in black. We associate the quantity $g^* + h = f$ in blue next (or below) to each node (first value before $=$ is $g^*$, second is $h$). The red crosses indicate when the search can be cut.

# Remarks

- Quality of the approximation function is key for efficiency (Task assignment cost function can be improved)
- Another idea for TSP: adding edges instead of nodes $\rightarrow$ see exercise in class
- Another strategy for TSP: find an approximation in an Euclidean space $\rightarrow$ see exercise in class
- Heuristics depend generally on the problem considered
- Other directions: Randomized approaches, genetic algorithms ...

# Branch and Bound: Summary and take home message

- Branch and Bound methods are used to solve complex problems, generally NP-hard problems.
- After a characterization of a search space of the solutions, the strategy consists in exploring this search space in a neat and efficient way by trying to "cut" some branches that cannot lead to optimal solutions.
- Other strategies include the use of heuristics, strong hypothesis on the problem, optimization methods or randomized approaches.
- The same strategy can be used to solve some NP-hard problems that are not necessarily optimization problems.