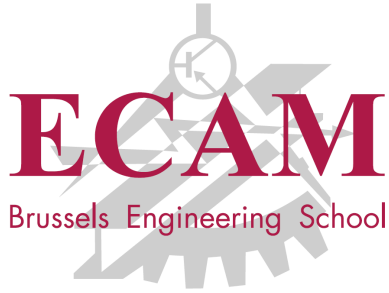


PI2T Développement informatique

Séance 4

Algorithmique I : Récursion et arbre

Sébastien Combéfis, Quentin Lurkin

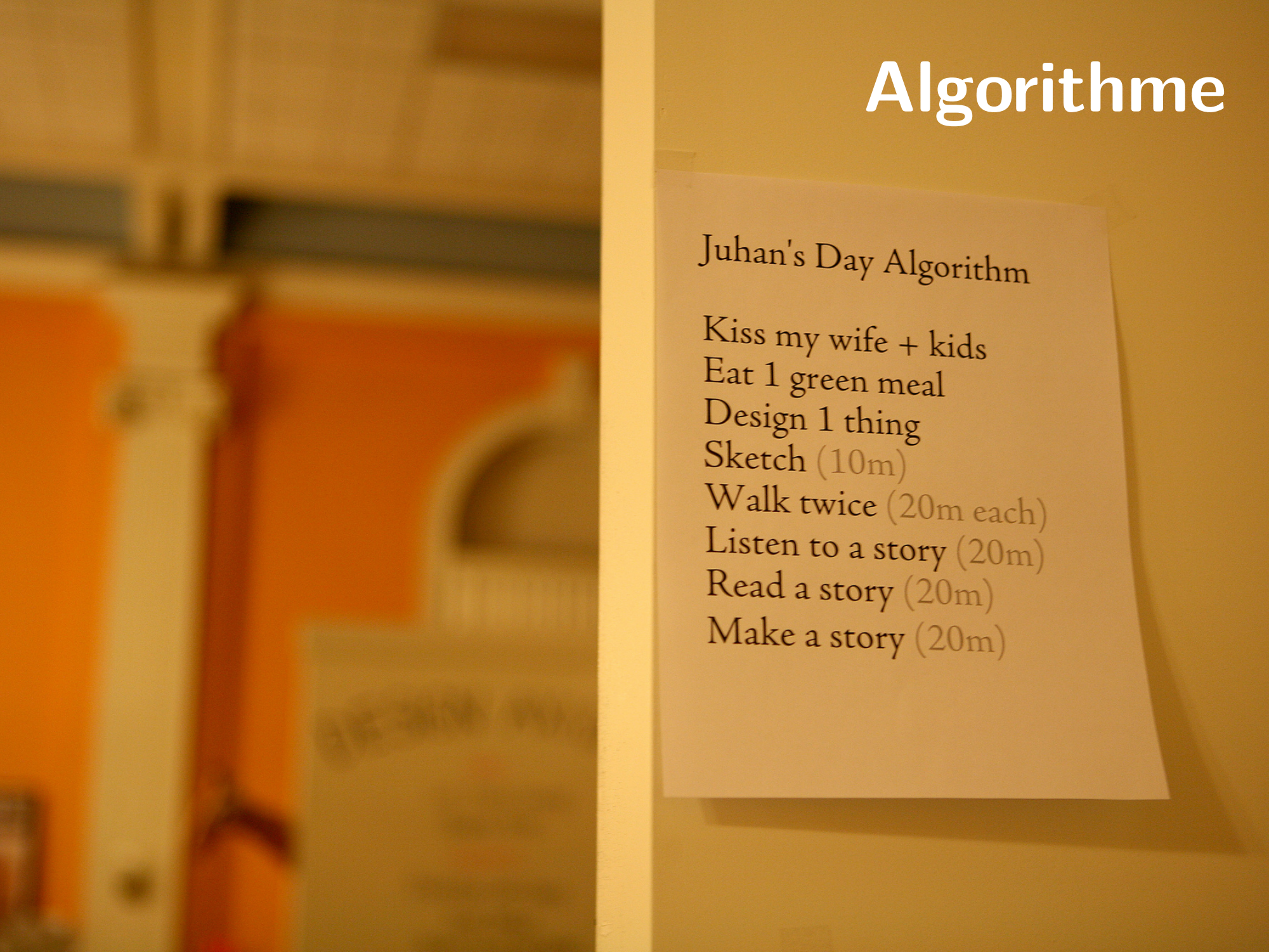
3 mars 2016

Ce(tte) œuvre est mise à disposition selon les termes de la Licence Creative Commons Attribution – Pas d'Utilisation Commerciale – Pas de Modification 4.0 International.

Objectifs

- Découverte de l'algorithmique
  - Définition et principe
  - Programmation récursive
- Le type abstrait de données arbre
  - Type abstrait de données
  - Arbre, propriétés et implémentation récursive
  - Backtracking

Algorithme


Juhan's Day Algorithm
Kiss my wife + kids
Eat 1 green meal
Design 1 thing
Sketch (10m)
Walk twice (20m each)
Listen to a story (20m)
Read a story (20m)
Make a story (20m)

Algorithme

- Un algorithme est un ensemble d'opérations à effectuer

  Décrit un processus qu'il est possible d'exécuter

- Nom provenant de al-Khwārizmī

  Mathématicien, astronome, géographe et savant Perse

- Plusieurs applications
  - Calcul
  - Traitement de données
  - Raisonnement automatique

Description d'algorithme

- Méthode effective pour calculer une fonction

  Décrite dans un langage formel bien défini

- Plusieurs façons de décrire un algorithme
  - Langue naturelle
  - Pseudo-code
  - Langage de programmation
  - Formalisme mathématique

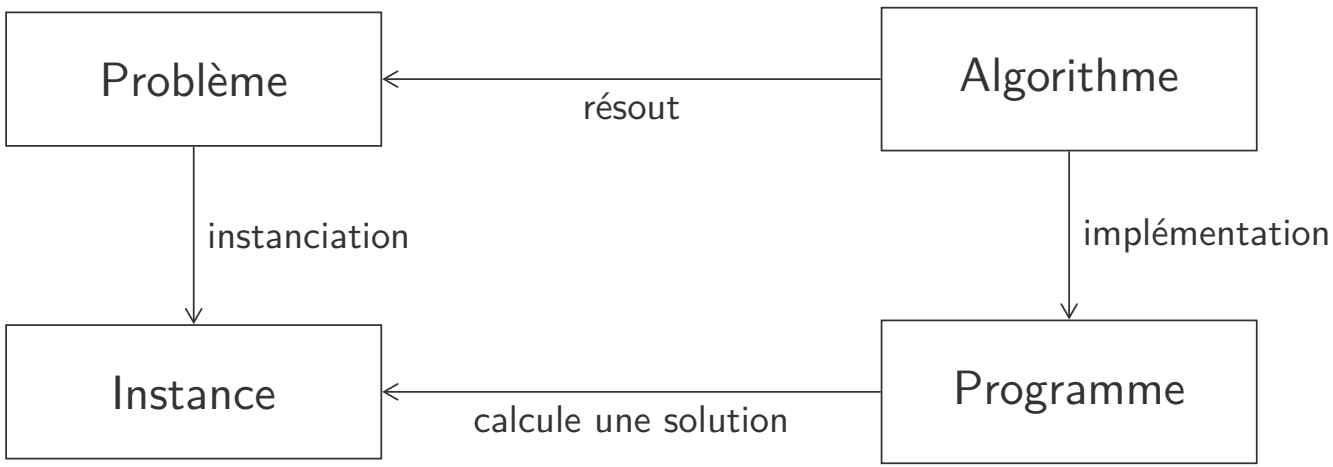
Problème

- Un algorithme a pour but de résoudre un problème

  Qui est exprimé sous la forme d'une fonction à calculer

- Implémentation dans un langage de programmation

  Permet de résoudre concrètement des instances du problème



Spécification d'un problème (1)

- Quatre éléments à identifier pour spécifier un problème
  - **Entrées** : *données à fournir qui sont nécessaires au problème*
  - **Sorties** : *résultat produit qui est solution du problème*
  - **Effet de bord** : *modification de l'environnement et des paramètres*
  - **Situations exceptionnelles** *où une exception est levée*

```
1  def factorial(n):
2      """Calcule la factorielle d'un nombre naturel.
3
4      Input: n, un nombre naturel.
5      Output: n!, la factorielle de n.
6      Raise: ArithmeticError lorsque n < 0
7      """
8      pass
```

Spécification d'un problème (2)

- Quatre éléments à identifier pour spécifier un problème
  - **Précondition** : *conditions qui doivent être satisfaites sur l'environnement et les paramètres avant de pouvoir faire l'appel*
  - **Sorties** : *conditions qui sont satisfaites sur l'environnement, la valeur renvoyée et les paramètres après l'appel, si les préconditions étaient satisfaites*
  - **Situations exceptionnelles** *où une exception est levée*

```
1  def factorial(n):
2      """Calcule la factorielle d'un nombre naturel.
3
4      Pre: n > 0.
5      Post: La valeur renvoyée contient la factorielle de n.
6      Raise: ArithmeticError lorsque n < 0
7      """
8      pass
```

Implémentation

- Plusieurs implémentations possibles pour un algorithme

  Langages et choix d'implémentation différents

- Plusieurs algorithmes possibles pour un problème

  Approches et structures de données différentes

Exemple : Recherche dans une liste

- Étant donné une liste, rechercher si un élément en fait partie

  Attention, la liste peut être vide

- Spécification du problème

```
1  def contains(data, elem):
2      """Recherche un élément donné dans une liste de données.
3
4      Pre: -
5      Post: La valeur renvoyée est
6            True si elem apparait au moins une fois dans la liste
7            et False sinon.
8      """
9      pass
```

Algorithme en langue naturelle

1. Si la liste est vide, on renvoie directement `False` et l'algorithme se termine
2. Pour chaque élément de la liste
    1. Si l'élément parcouru est égal à celui recherché, on renvoie directement `True` et l'algorithme se termine
3. L'élément recherché n'est pas dans la liste, on renvoie `False`

Algorithme en pseudo-code

- Le pseudo-code est une façon de décrire un algorithme

  De manière indépendante de tout langage de programmation

- Pas de réelle convention pour le pseudo-code

  Les opérations sont de haut niveau

Algorithm 1: Recherche si un élément se trouve dans une liste

```
if n = 0 then
    return false
foreach e ∈ L do
    if e = elem then
        return true
return false
```

Optimisation de l'algorithme

- Possibilité d'optimisation d'un algorithme
  - Diminution du nombre d'opérations à effectuer
  - Simplification de l'algorithme
- Le cas $n = 0$ ne doit pas être traité séparément

Algorithm 2: Recherche si un élément se trouve dans une liste

```
foreach e ∈ L do
    if e = elem then
        return true
return false
```

Algorithme en langage de programmation

- Implémentation effective dans un langage de programmation

  Utilisation des caractéristiques spécifiques du langage

- Peut être fait par traduction du pseudo-code

  Traduction des constructions de haut niveau dans le langage

```
1  def contains(data, elem):
2      for e in data:
3          if e == elem:
4              return True
5      return False
```

Optimisation du programme

- Possibilité d'optimisation d'un programme
  - Utilisation de constructions spécifiques au langage
  - Exploitation de la librairie standard
- Python possède un opérateur `in` pour tester l'appartenance

```
1 def contains(data, elem):
2     return elem in data
```

Récursion

An Escher Vortex Where All Levels Cross

A strikingly beautiful, and yet at the same time disturbingly grotesque, illustration of the cyclonic "eye" of a Tangled Hierarchy is given to us by Escher in his *Print Gallery* (Fig. 142). What we see is a picture gallery where a young man is standing, looking at a picture of a ship in the harbor of a small town, perhaps a Maltese town, to guess from the architecture, with its little turrets, occasional cupolas, and flat stone roofs, upon one of which sits a boy, relaxing in the heat, while two floors below him a woman—perhaps his mother—gazes out of the window from her apartment which sits directly above a picture gallery where a young man is standing, looking at a picture of a ship in the harbor of a small town, perhaps a Maltese town—What!? We are back on the same level as we began, though all logic dictates that we cannot be. Let us draw a diagram of what we see (Fig. 143).

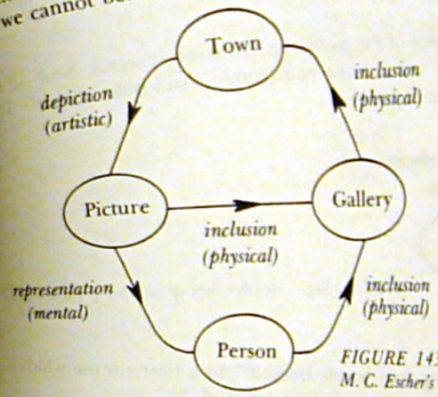


FIGURE 143. Abstract diagram of M. C. Escher's Print Gallery.

What this diagram shows is three kinds of "in-ness". The gallery is *physically in* the town ("inclusion"); the town is *artistically in* the picture ("depiction"); the picture is *mentally in* the person ("representation"). Now while this diagram may seem satisfying, in fact it is arbitrary, for the number of levels shown is quite arbitrary. Look below at another way of representing the top half alone (Fig. 144).

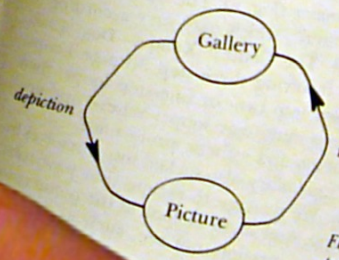


FIGURE 144. A collapsed version of the previous figure.

FIGURE 142. Print Gallery, by M. C. Escher (lithograph, 1956).

program runs. On a low (machine language) level, the program looks like any other program; on a high (chunked) level, qualities such as "will", "intuition", "creativity", and "consciousness" can emerge.

The important idea is that this "vortex" of self is responsible for the tangledness, for the Gödelian-ness, of the mental processes. People have said to me on occasion, "This stuff with self-reference and so on is very amusing and enjoyable, but do you really think there is anything *serious* to it?" I certainly do. I think it will eventually turn out to be at the core of AI, and the focus of all attempts to understand how human minds work. And that is why Gödel is so deeply woven into the fabric of my book.

Récursion

- Processus de récursion découpe en sous-problèmes
  - Dont la structure est la même que le problème original
  - Plus simples à résoudre
- Décompositions successives du problème original

  Jusqu'à avoir un sous-problème qui se résout directement
- Classe des algorithmes récursifs

  Algorithme qui exploite la récursion pour résoudre le problème

Collecte de fonds (1)

- Politicien doit trouver $1000 pour sa campagne

  Collecte de $1 auprès de 1000 supporters

- Algorithme itératif (boucle)

  On demande successivement $1 aux 1000 supporters

```
1 def collect1000():
2     for i in range(1000):
3         # collect $1 from person i
```

Collecte de fonds (2)

- Sous-traiter la recherche d'argent à des intermédiaires

  Trouver 10 personnes qui vont chercher $100

- Algorithme récursif (récursion)

  On délègue à des intermédiaires qui vont eux-mêmes déléguer...

```
1  def collect(n):
2      if n == 1:
3          # donne $1 au supérieur
4      else:
5          for i in range(n): # délégation à n personnes
6              collect(n // 10)
7          # donne l'argent récoltée au supérieur
```

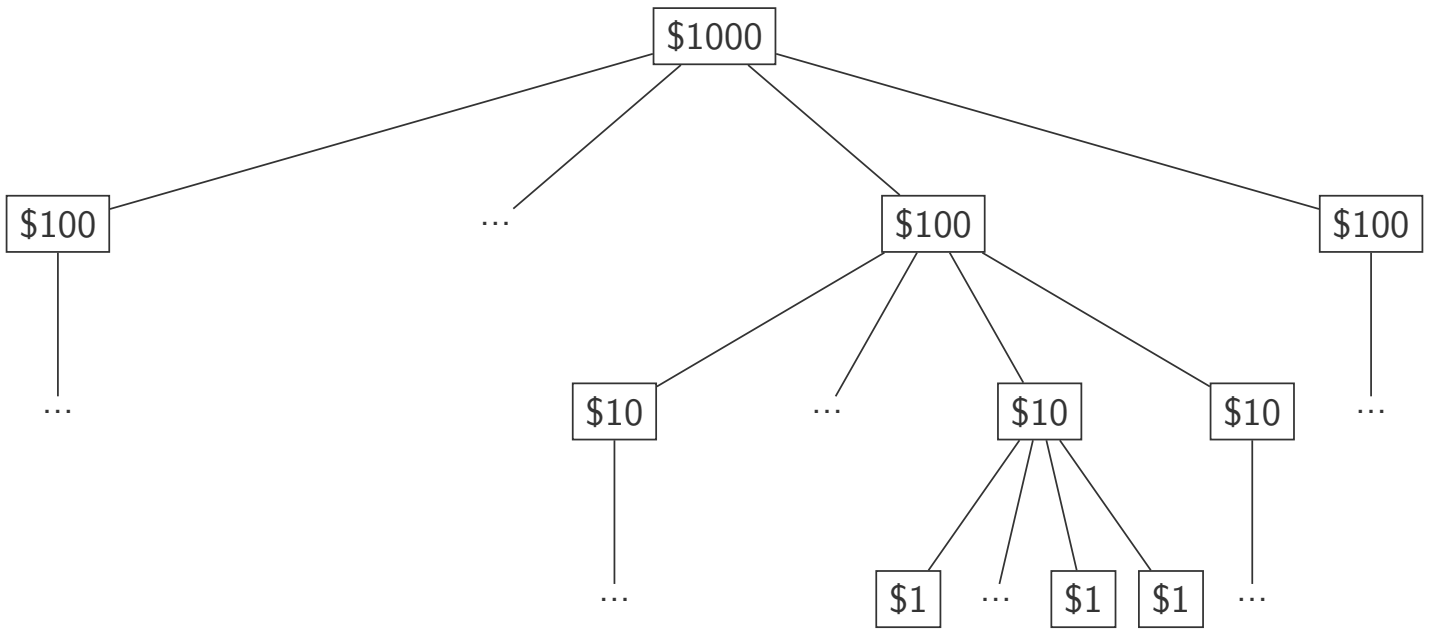
Concepts

- Stratégie de diviser pour régner (*divide-and-conquer*)

  Décomposition du problème original en sous-problèmes

- Cas de base simple et cas récursif à décomposer

  Représentation graphique avec un arbre de solution



Caractérisation

- Suivre l'approche de diviser pour régner

  Plusieurs instances du même problème plus simples à résoudre

- Problème candidat à une solution récursive

  1. On peut **décomposer** le problème original en *instances plus simples* du même problème
  2. Les sous-problèmes doivent finir par *devenir suffisamment simples* que pour être **résolus directement**
  3. On peut **combiner** les solutions des sous-problèmes pour *produire la solution* du problème original

Terminaison

- Un processus récursif peut ne pas se terminer

  Équivalent des boucles infinies pour les itérations

- Le cas de base doit toujours finir par être atteint

- Collecte de fonds avec un délégué pour rechercher les $1000

  Le processus ne se termine pas, cas de base jamais atteint

```
1  def collect(n):
2      if n == 1:
3          # donne $1 au supérieur
4      else:
5          collect(n) # délégation à une personne
6          # donne l'argent récoltée au supérieur
```

Penser récursivement

Achilles *I will be glad to indulge both of you, if you will first oblige me, by telling me the meaning of these strange expressions, "holism" and "reductionism."*

Crab ***Holism*** *is the most natural thing in the world to grasp. It's simply the belief that "the whole is greater than the sum of its parts." No one in his right mind could reject holism.*

Anteater ***Reductionism*** *is the most natural thing in the world to grasp. It's simply the belief that "a whole can be understood completely if you understand its parts, and the nature of their 'sum."' No one in her left brain could reject reductionism.*

"Gödel, Escher, Bach : an Eternal Golden Braid", Douglas R. Hofstadter, 1999.

Fonction récursive

- Résolution du problème original et des sous-problèmes générés

  Les résolutions peuvent se faire avec la même fonction

- Les paramètres permettent d'identifier les sous-problèmes

  Ces paramètres sont utilisés pour gérer les cas de base et récursif

Exemple : Factorielle (1)

- Factorielle d'un nombre naturel

  $n! = n \times (n-1) \times ... \times 1$ et $0!$ par convention

- Deux visions possibles pour calculer cette fonction

  Vision itérative et vision récursive

```
fact(0) =             = 1
fact(1) =           1 = 1
fact(2) =       2 x 1 = 2
fact(3) = 3 x 2 x 1 = 6
```

```
fact(0) = 1
fact(1) = 1 x fact(0)
fact(2) = 2 x fact(1)
fact(3) = 3 x fact(2)
```

Exemple : Factorielle (2)

- Récursion sur n

  Cas de base $0! = 1$

  Factorielle de 0 *vaut* 1 *par convention*

  Cas récursif $n! = n \cdot (n-1)!$

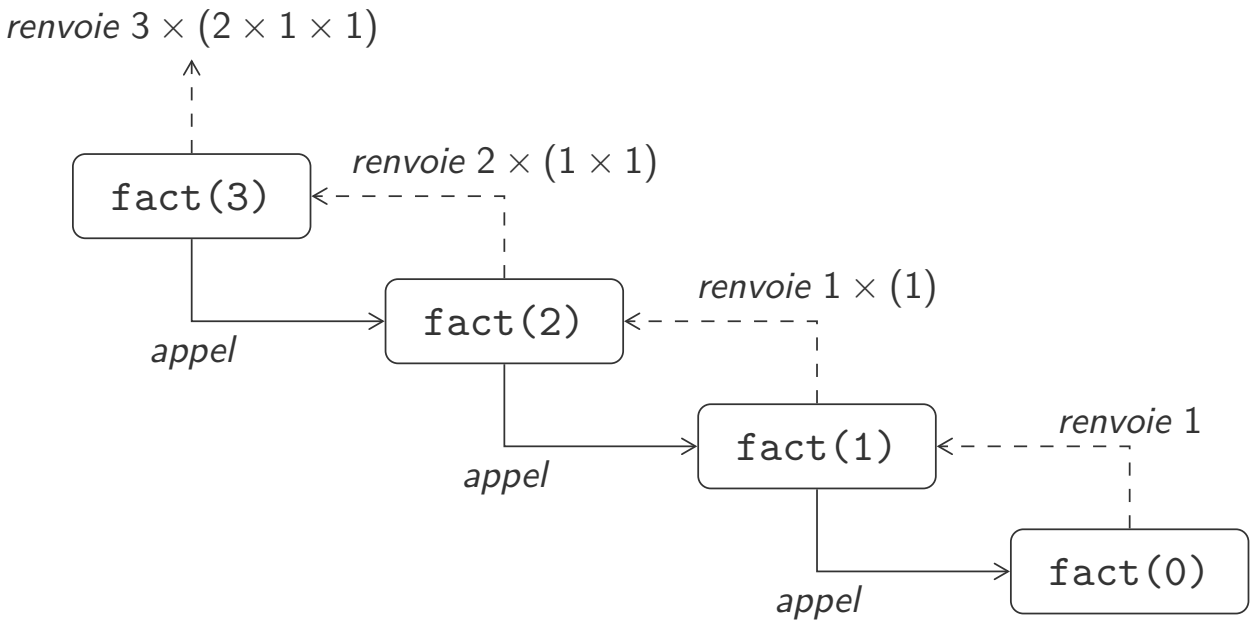
  Factorielle de n se calcule à partir de celle de $n-1$

```
1  def fact(n):
2      if n == 0:
3          return 1
4      return n * fact(n - 1)
```

Exemple : Factorielle (3)

- La fonction `fact` est appelée plusieurs fois

  Une récursion produit également une boucle



Exemple : Fibonacci

- Récursion sur n

  Cas de base $F_1 = 1$ et $F_2 = 1$

  1^{er} et 2^{e} nombres de Fibonacci en cas de base

  Cas récursif $F_n = F_{n-1} + F_{n-2}$

  n^{e} nombre de Fibonacci dépend des deux précédents

```
1  def fibo(n):
2      if n == 1 or n == 2:
3          return 1
4      return fibo(n - 1) + fibo(n - 2)
```

Exemple : Recherche dans une liste

- Récursion sur la taille de la liste

  Cas de base La liste est vide ou contient un élément

  On peut rechercher immédiatement l'élément recherché

  Cas récursif Recherche dans la liste sans le premier élément

  La recherche se fait sur une liste plus courte

```
1  def contains(data, value):
2      if len(data) == 0:
3          return False
4      if data[0] == value:
5          return True
6      return contains(data[1:], value)
```

Exemple : Tri d'une liste par fusion

- Récursion sur la taille de la liste

  Cas de base La liste est vide ou contient un élément

  Une telle liste est déjà triée

  Cas récursif Tri séparé de deux listes (on coupe l'originale)

  Ensuite, fusion des deux listes triées

```
1  def sort(data):
2      def merge(l1, l2):
3          # ...
4      n = len(data)
5      if n <= 1:
6          return data
7      return merge(sort(data[:n//2]), sort(data[n//2:]))
```


Arbre

Type abstrait de données

- Type abstrait de données (TAD) spécifie mathématiquement
  - Un ensemble de données
  - Les opérations qu'on peut effectuer
- Correspond à un cahier des charges

  Implémentation du CDC par une structure de données
- Plusieurs implémentations possibles pour un même TAD

  Se différencient par la complexité calculatoire et spatiale

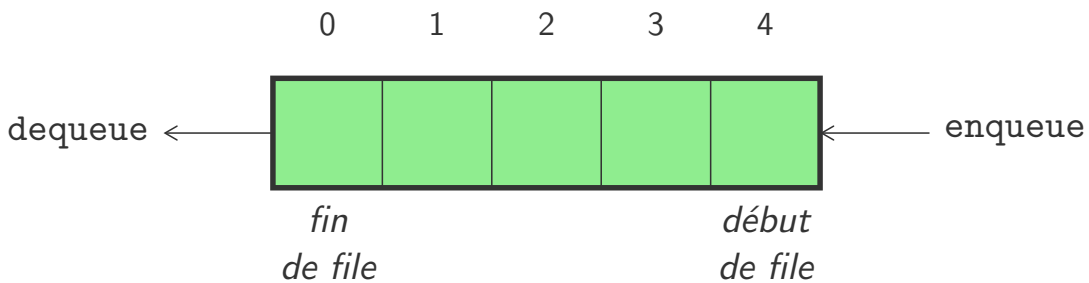
File

- Séquence de type First-in First-out (FIFO)

  Le premier élément qui a été ajouté sera le premier à sortir

- Opérations possibles

`size`	donne la taille de la file
`isEmpty`	teste si la file est vide
`front`	récupère l'élément en début de file
`enqueue`	ajoute un élément en fin de file
`dequeue`	retire l'élément en début de file



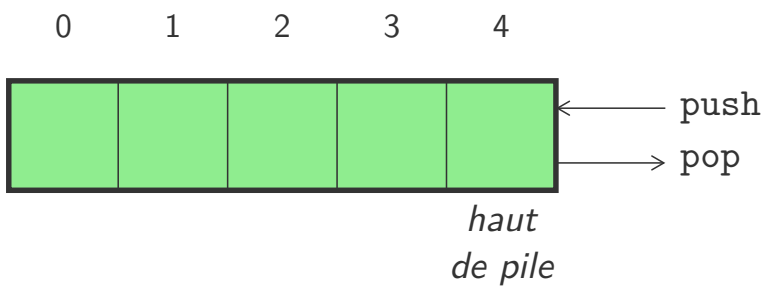
Pile

- Séquence de type Last-in First-out (LIFO)

  Le dernier élément qui a été ajouté sera le premier à sortir

- Opérations possibles

`size`	donne la taille de la pile
`isEmpty`	teste si la pile est vide
`top`	récupère l'élément en haut de la pile
`push`	ajoute un élément en haut de la pile
`pop`	retire l'élément en haut de la pile



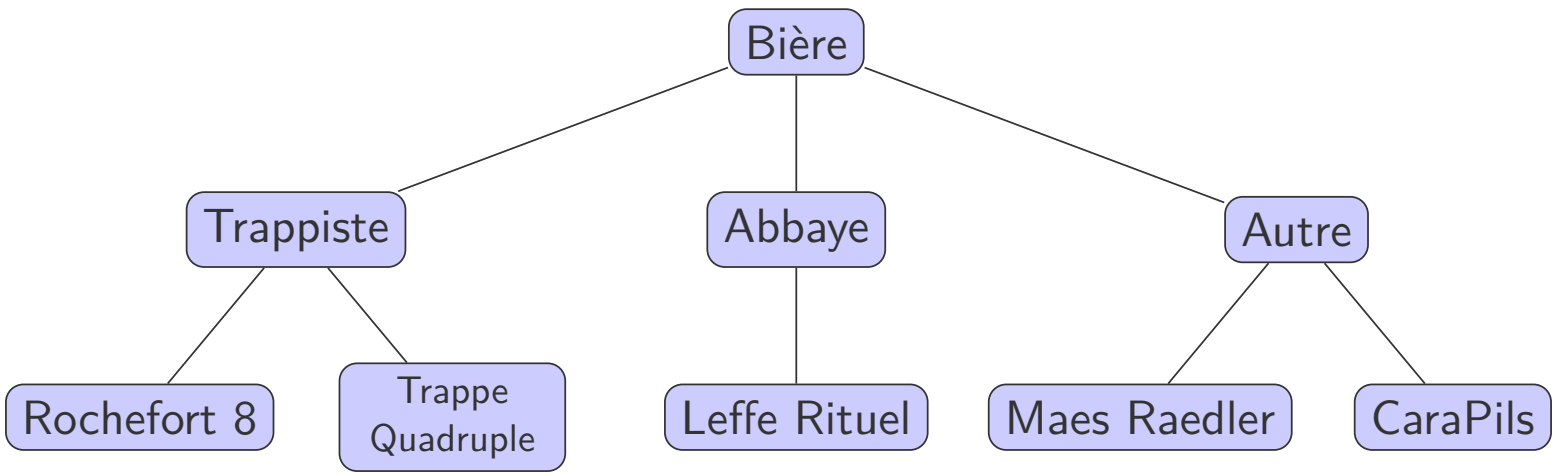
Arbre

- Éléments d'un arbre organisés de manière hiérarchique

  Un arbre est un ensemble de nœuds (qui contiennent les valeurs)

- Chaque nœud possède un parent et zéro ou plusieurs enfants

  Sauf la racine de l'arbre qui n'a pas de parent



Définition récursive

- Deux cas possibles pour définir un arbre
  - Un arbre **vide** (sans enfants)
  - Un nœud avec un élément et une **liste de sous-arbres**
- Opérations possibles

`size`	donne la taille de l'arbre
`value`	récupère la valeur stockée à la racine de l'arbre
`children`	récupère la liste des sous-arbres enfants de la racine
`addChild`	ajoute un sous-arbre comme enfant à la racine

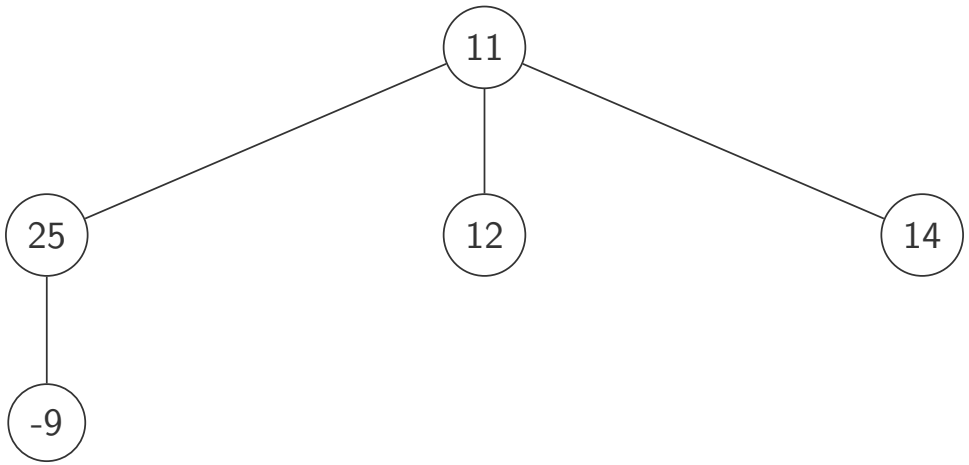
Classe Tree (1)

- Classe `Tree` possède deux variables d'instance

  Une valeur (racine) et une liste de sous-arbres (enfants)

```
1  import copy
2
3  class Tree:
4      def __init__(self, value, children=[]):
5          self.__value = value
6          self.__children = copy.deepcopy(children)
7
8      @property
9      def value(self):
10         return self.__value
11
12     @property
13     def children(self):
14         return copy.deepcopy(self.__children)
15
16     def addChild(self, tree):
17         self.__children.append(tree)
18
19     # ...
```

Classe Tree (2)




```
1  t1 = Tree(-9)
2  t2 = Tree(25, [t1])
3  t3 = Tree(12)
4  t4 = Tree(14)
5
6  t = Tree(11, [t2, t3, t4])
```

Taille d'un arbre

- Calcul de la taille d'un arbre de manière récursive

  1 + somme des tailles des sous-arbres enfants

  Cas de base Aucun enfant

  Cas récursif Appel récursif pour chaque enfant

```
1     # ...
2
3     @property
4     def size(self):
5         result = 1
6         for child in self.__children:
7             result += child.size
8         return result
9
10    # ...
```

Redéfinition de l'opérateur []

- Définition de l'opérateur `[]` pour les objets de type `Tree`

  Il faut redéfinir la méthode `__getitem__(self, index)`

```
    # ...

    def __getitem__(self, index):
        return self.__children[index]

    # ...

t = Tree(78, [Tree(14), Tree(9)])
for i in range(len(t.children)):
    print(t[i].value)
```

```
14
9
```

Représentation textuelle d'un arbre (1)

- Représentation textuelle de manière récursive

  Valeur concaténée avec les représentations de chaque sous-abre

  Cas de base Aucun enfant

  Cas récursif Appel récursif pour chaque enfant

- Utilisation d'une fonction auxiliaire pour gérer le niveau

```
1      # ...
2
3      def __str__(self):
4          def _str(tree, level):
5              result = '[{}]\n'.format(tree.__value)
6              for child in tree.children:
7                  result += '{}|--{}'.format('    ' * level, _str(
                   child, level + 1))
8              return result
9          return _str(self, 0)
```

Représentation textuelle d'un arbre (2)

```
1  c1 = Tree(25, [Tree(-9)])
2  c2 = Tree(12)
3  c3 = Tree(14)
4
5  t = Tree(11, [c1, c2, c3])
6  print(t)
7
8  t[2].addChild(Tree(8))
9  print(t)
```

```
[11]
|--[25]
   |--[-9]
|--[12]
|--[14]

[11]
|--[25]
   |--[-9]
|--[12]
|--[14]
   |--[8]
```

Backtracking

- Récursion beaucoup utilisée en intelligence artificielle

  Recherche choix optimal dans un ensemble de possibilités

- Faire une tentative de séquences de choix
  - Possibilité de faire marche arrière par rapport à un choix
  - Exploration de nouvelles décisions
- La récursion permet de faire facilement du backtracking

“The Turk”

- Canular d’un automate joueur d’échecs

  Construit et dévoilé en 1770 par Johann Wolfgang von Kempelen

Lookahead

- Explorer un maximum de coups possibles à l'avance

  Sélectionner le coup qui mène à la meilleure situation

- Pas toujours possible d'explorer tous les coups

  Trouver le moins pire étant donné une contrainte temporelle

- Deux notions clés
  - L'**état du jeu** représente la situation de ses joueurs
  - Un **coup** fait la transition entre deux états

Jeu de Nim

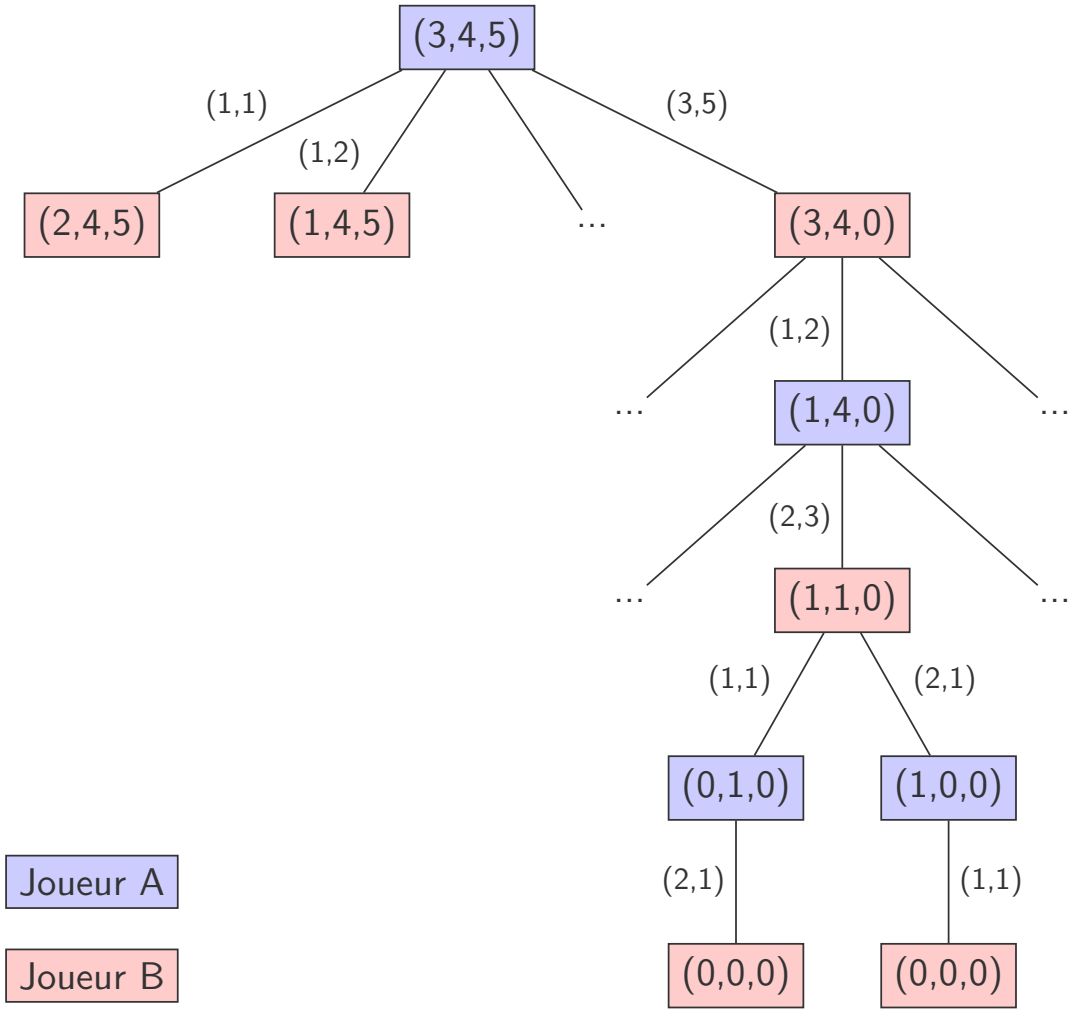
- Le jeu de Nim se joue à deux joueurs
  - Le joueur choisit une rangée et retire autant de pièces qu'il veut
  - Le joueur qui retire la dernière pièce a gagné

Rangée 1

Rangée 2

Rangée 3

Arbre complet du jeu



Trouver le meilleur coup (1)

- Deux fonctions utilitaires
  - Tester si le jeu est **fini** (il ne reste plus de pièces)
  - Générer la liste des **coups possibles**

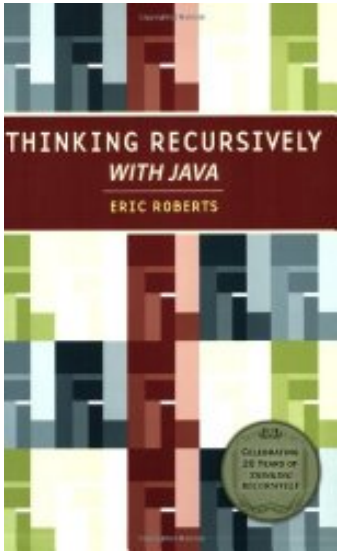
```
1  def isgameover(state):
2      for n in state:
3          if n > 0:
4              return False
5      return True
6
7  def getmoves(state):
8      moves = []
9      for i in range(len(state)):
10         moves += [(i, n) for n in range(1, state[i] + 1)]
11     return moves
```

Trouver le meilleur coup (2)

- Deux fonctions récursives (récursion mutuelle)
  - Tester si un état est **mauvais** (mène à une perte du jeu)
  - Trouver un **bon coup** (qui ne mène pas à un mauvais état)

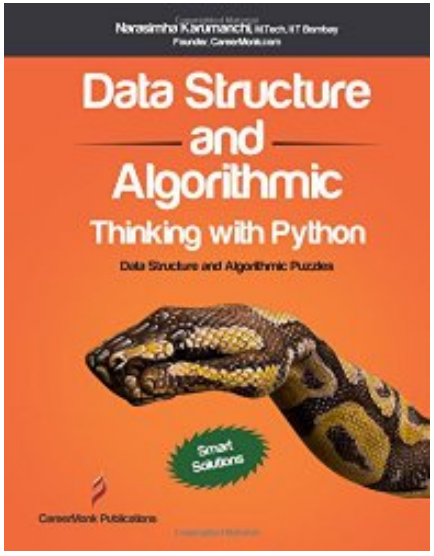
```
1  def isbadposition(state):
2      if isgameover(state):
3          return True
4      return findgoodmove(state) is None
5
6  def findgoodmove(state):
7      for move in getmoves(state):
8          nextstate = tuple(state[i] - move[1] if i == move[0] else
           state[i] for i in range(len(state)))
9          if isbadposition(nextstate):
10             return move
11     return None
```

Livres de référence



ISBN

978-0-471-70146-0



ISBN

978-8-192-10759-2

Crédits

- Photos des livres depuis Amazon
- https://www.flickr.com/photos/juhansonin/8331686714
- https://www.flickr.com/photos/gadl/279433682
- https://www.flickr.com/photos/127497725@N02/16695848708
- http://static-numista.com/catalogue/photos/belgique/g2442.jpg