Pseudo-code

Guicheteau

Objectifs

- Comprendre ce qu'est un algorithme.
- Savoir écrire un algorithme en pseudo-code
- Comprendre les notions d'arrêt et de validité d'un algorithme

Introduction

Un **algorithme** est tout simplement une suite d'étapes à suivre, comme une recette de cuisine, pour résoudre un problème ou accomplir une tâche. L'**algorithmique** est la discipline fondamentale de l'informatique qui s'intéresse à la conception, à l'étude et à l'analyse de méthodes permettant de résoudre des problèmes de manière systématique.

Les humains ont inventé des algorithmes depuis très longtemps. Il y a plus de 2000 ans, les mathématiciens de l'Antiquité cherchaient déjà des méthodes pour faire des calculs, comme trouver le plus grand nombre commun à deux valeurs. Le mot « algorithme » vient d'ailleurs du nom d'un savant du IXe siècle, **Al-Khwarizmi**, qui a expliqué comment résoudre des équations du second degré.

Remarque

Il ne faut pas confondre un **algorithme** avec un **programme**.

- Un algorithme, c'est l'idée : la méthode générale pour arriver au résultat.
- Un programme, c'est la traduction de cet algorithme dans un langage que l'ordinateur comprend.

Définition

Un **problème algorithmique** est défini par son entrée et sa sortie.

Exemple

Par exemple, le problème du tri peut se définir comme suit :

Problème	Tri
Entrée	Un ensemble de $n \geq 1$ données $a_1, a_2, \dots, a_n$
Sortie	Une permutation des données telle que $a'_1 \leq a'_2 \leq \dots \leq a'_n$

Certaines spécifications peuvent être trop vagues ou trop générales, auquel cas il devient difficile voire impossible d'écrire un algorithme de résolution.

Exemple de problème mal défini

Problème	Meilleur chemin
Entrée	Une carte routière, deux points A et B sur la carte
Sortie	Le meilleur chemin entre A et B

Ici la notion de "meilleur chemin" n'a pas de sens. S'agit-il du chemin le plus court, le plus rapide, le moins cher ? Il est important de bien définir les objets et leurs relations.

Définition

Une **instance** d'un problème algorithmique est la donnée d'une entrée spécifique à ce problème.

Exemple

Pour le problème du tri :

- $\{235, 564, 12, 325, 95, 631, 223\}$
- `Nicolas, Jean-Pierre, Pascal, Thierry, Valérie, Joseph`

sont deux instances de ce problème. Remarquons que dans chacun des cas il est important de bien définir la relation d'ordre entre les éléments. Dans le premier cas il s'agit de l'ordre naturel sur les nombres alors que dans le deuxième cas il s'agit de l'ordre lexicographique.

Principe fondamental

Un algorithme de résolution d'un problème doit fonctionner pour **toute instance** du problème.

Le pseudo-code

On utilise classiquement trois types de langage pour décrire un algorithme :

— le langage naturel
— le pseudo-code
— les langages de programmation

Chaque façon représente un certain compromis entre facilité d'expression et précision.

i Exemple : Test de primalité d'un entier

Problème	Savoir si un nombre est premier
Entrée	Un entier $n \geq 1$
Sortie	VRAI si l'entier est premier, FAUX sinon

En langage naturel : *On teste si le nombre est divisible ou non par tous les entiers inférieurs strictement à lui-même et supérieurs strictement à 1. Si un de ces nombres le divise, il n'est pas premier, sinon, il est premier.*
En Python :

```
def EstPremier(n):
    for d in range(2, n):
        if n%d==0:
            return False
    return True
```

En pseudo-code :

```
ALGORITHME EstPremier(n): booléen
DONNEES:
  n: entier
VARIABLES:
  d: entier
DEBUT
  d ← 2
  TQ d < n FAIRE
    SI n mod d = 0 ALORS
      RENVOYER FAUX
    FSI
    d ← d+1
  FTQ
  RENVOYER VRAI
FIN
```

! Règle d'or

Quelle que soit la méthode choisie, c'est la clarté de la description qui doit primer. Tout algorithme repose sur une **idée clé**, la description de l'algorithme doit permettre de la faire apparaître clairement.

Règles du pseudo-code

Il s'agit d'un langage universel en ce sens qu'il ne dépend pas d'une architecture machine ou d'un langage de programmation. Il reprend la structure d'un programme classique sans référence à un langage particulier et permet de décrire précisément un algorithme en faisant abstraction de certaines difficultés techniques.

⚠ Règles pour ce cours

Le Pseudo-code sera composé de trois blocs :

1. **Données** : la ou les entrées à traiter par l'algorithme
2. **Variables** : l'ensemble des différentes variables que manipule l'algorithme
3. **Corps de l'algorithme** : la séquence des instructions à exécuter, encadrée par les mots clés **DEBUT** et **FIN**

Mots clés : DEBUT, FIN, SI, ALORS, FSI, SINON, ET, OU, TQ, FTQ, FAIRE, RENVOYER
L'affectation d'une variable se fait avec l'opérateur ←
Opérations standard sur les entiers : +, -, *, /, %
Manipulation des tableaux de tailles fixes :

1. **les indices commencent à 1 !!!**
2. l'accès aux éléments se fait par l'opérateur [ ]

Des procédures complémentaires sont en général précisées en fonction du contexte (comme par exemple la procédure AFFICHER).

ℹ Exemple : Calcul de la factorielle d'un nombre

Problème	Calculer la factorielle d'un nombre
Entrée	Un entier $n \geq 1$
Sortie	Le résultat de $n!$

```
ALGORITHME Factorielle(n):
DONNEES:
  n : entier
VARIABLES:
  f : entier
  i : entier
DEBUT
  i ← 1
  f ← 1
  TQ i   n FAIRE
    f ← f * i
    i ← i + 1
  FTQ
  RENVOYER f
FIN
```

Condition d'arrêt

Lorsqu'on utilise des boucles ou de la récursivité, il faut prouver que notre algorithme se termine bien.

Pour cela, il faut montrer que la condition d'arrêt de la boucle est bien réalisée.

On appelle cela une **preuve d'arrêt** dont le but est de vérifier que quelle que soit l'instance du problème qui est traitée, l'algorithme finit toujours par s'arrêter.

i Exemple de preuve d'arrêt

Prouver que l'algorithme suivant se termine :

```
ALGORITHME Somme(n): entier
DONNEES:
  n: entier
VARIABLES:
  i: entier
  s: entier
DEBUT
  i ← 1
  s ← 0
  TQ i   n FAIRE
    s ← s + i
    i ← i + 1
  FTQ
  RENVOYER s
FIN
```

Preuve d'arrêt : Il faut montrer que chaque boucle de l'algorithme voit sa condition de continuation invalidée au bout d'un certain nombre d'étapes.
Ici il suffit de montrer que la variable i finit toujours par prendre une valeur supérieure à $n+1$. Or i est initialisée à 1 et est incrémentée de 1 à chaque tour de boucle ($i \leftarrow i+1$). La variable i prend donc les valeurs $1, 2, 3, ...$, la condition de continuation sera donc invalidée lorsque $i = n+1$. Formellement, nous ferons appel aux suites pour montrer cela.

! Remarque

On ne peut pas toujours prouver qu'une boucle s'arrête, il existe des cas indécidables comme par exemple l'algorithme de Syracuse.

Validité d'un algorithme

Une deuxième chose essentielle en algorithmique est que notre algorithme doit répondre au problème quelque soit l'instance choisie. Pour montrer cela, nous ferons une **preuve de justesse**.

Les preuves de justesse sont en général assez difficiles à rédiger rigoureusement, même pour des algorithmes assez simples, c'est pourquoi dans ce cours nous nous limiterons aux preuves d'arrêt dans la plupart des cas.

La preuve par récurrence est un des outils les plus utiles pour démontrer la validité d'un algorithme. Elle permet de montrer qu'une propriété $\mathcal{P}(n)$ est vraie pour tout n plus grand qu'une certaine constante (en général 0).

La propriété $\mathcal{P}(n)$ choisie correspond à un invariant de boucle qui permettra, une fois la boucle finie, de répondre au problème donné.

Méthode de la preuve par récurrence

Pour montrer qu'une propriété $\mathcal{P}(n)$ est vraie pour tout $n \geq n_0$, il faut montrer 2 points :
- la propriété $\mathcal{P}(n_0)$ est vraie (**initialisation**)
- si la propriété $\mathcal{P}(n)$ est vraie alors la propriété $\mathcal{P}(n+1)$ l'est aussi (**hérédité**)

Exemple : Preuve de justesse pour l'algorithme de la somme des n premiers entiers

Nous allons montrer qu'à chaque fin de passage dans la boucle, $s_i = \sum_{k=1}^{i-1} k$

Soit, pour $i \in \mathbb{N}^*$, $\mathcal{P}(i) : s_i = \sum_{k=1}^{i-1} k$ à la fin de la $i^{\text{ème}}$ itération.

Initialisation : $\mathcal{P}(1)$ est vraie car la variable i a été incrémentée et vaut donc 2, donc $\sum_{k=1}^{2-1} k = 1$ et $s_1 = 0 + 1 = 1$.

Hérédité : Soit $i \geq 1$. Supposons que $\mathcal{P}(i)$ est vraie, montrons qu'à la fin de l'itération suivante, $\mathcal{P}(i+1)$ est vraie.

On a, à la fin de l'itération suivante :

$s_{i+1} = s_i + i$ donc $s_{i+1} = \sum_{k=1}^{i-1} k + i = \sum_{k=1}^{i} k$.

Ainsi $\mathcal{P}(i+1)$ est vraie

Conclusion : A la fin de la boucle, i vaut $n+1$, or $\mathcal{P}(i)$ est vraie pour tout i entier supérieur à 1, donc $\mathcal{P}(n+1)$ et $s = \sum_{k=1}^{n} k$.

L'algorithme est donc correct !

Attention

L'initialisation ne doit pas être oubliée. Une propriété peut être héréditaire sans jamais être vraie pour aucun $n \in \mathbb{N}$.
Exemple : Si 10^n est divisible par 3 alors 10^{n+1} est divisible par 3, mais quel que soit n, 10^n n'est jamais divisible par 3.

Invalidité et tests d'un algorithme

En général, faire une preuve de validité est difficile. Avant de se lancer dans une preuve, on vérifie que l'algorithme n'est pas trivialement faux.

Méthode pour invalider un algorithme

Montrer qu'un algorithme n'est pas valide est en général plus simple : il suffit d'exhiber un **contre-exemple**, c'est-à-dire une instance du problème pour laquelle l'algorithme ne renvoie pas le bon résultat.

Exemple : Ramassage de plots

Problème	Ramassage de plots
Entrée	n plots sur un terrain numérotés de 1 à n
Sortie	Le plus court chemin pour ramasser tous les plots et revenir au point de départ en partant du plot 1

Solution proposée (algorithme glouton) :

```
ALGORITHME RamassePlots()
DEBUT
  Ramasser le plot 1
  TQ (il reste des plots) FAIRE
    ramasser le plot le plus proche
  FTQ
  revenir au début
FIN
```

Test de l'algorithme :
L'instance défavorable montre que l'algorithme glouton ne donne pas toujours la solution optimale !
Par exemple, si les plots sont alignés avec : plot 1 au milieu, plot 2 juste à côté, plot 3 de l'autre côté proche, plot 4 très loin à droite, plot 5 très loin à gauche, l'algorithme glouton va faire des allers-retours inutiles.

❗ Algorithme correct (mais coûteux)

```
ALGORITHME RamassePlotsCorrect()
DEBUT
  Calculer la longueur de tous les parcours possibles
  Choisir le parcours le plus court
FIN
```

Problème : Il y a $(n-1)!$ parcours possibles ! Pour $n = 10$, cela fait déjà 362 880 parcours à tester…

💡 Stratégie générale de test

1. **Cas simples :** tester avec des instances de petite taille
2. **Cas limites :** tester avec des valeurs particulières (0, 1, valeurs maximales…)
3. **Cas défavorables :** chercher des configurations qui pourraient mettre l'algorithme en défaut
4. **Tests aléatoires :** générer des instances au hasard pour détecter des erreurs inattendues