Algorithmique

I NOTION D'ALGORITHME

1 Définition

Définition : Algorithme

Suite d'instructions permettant de résoudre un problème et pouvant être exécuté par une machine.

Exemple

Ingrédients (pour 2 personnes) : - 400 à 600 g de pâtes (suivant l'appétit). - thon en boîte émietté - 1 oignon en lamelles - 1 petite boîte de concentré de tomates - 1 verre de crème fraîche légère liquide - gruyère râpé - sel et poivre

Préparation de la recette :

Faire cuire les pâtes selon votre goût et le type de pâtes.

Dans une casserole, faire revenir les oignons avec un peu d'huile, ajouter le thon, le concentré de tomates, la crème, le sel et le poivre.

Égoutter les pâtes, et les mélanger avec la préparation.

Ajouter le gruyère râpé.

2 Pseudo-Langage vs implémentation Python

Un algorithme doit pouvoir se traduire dans n'importe quel langage de programmation.

On commence par l'écrire en pseudo-langage : mots en français, affectation : `a←10`, etc.

On peut ensuite le convertir dans n'importe quel langage, par exemple en Python.

3 Justifications d'algorithmes

Tout algorithme doit être justifié pour vérifier qu'il fait bien ce qu'on attend de lui et qu'il se termine.

Cela est surtout délicat lorsqu'il y a des boucles et nécessite la notion d'**invariant de boucle** définie ci-après .

II TESTS ET BOUCLES

1 Tests

On peut effectuer des tests du type :

« Si ... Alors ... FinSi »
« Si ... Alors ... Sinon ... FinSi »
« Si ... Alors ... Sinon Si ... Alors ... FinSi ».

2 Boucles non conditionnelles

Les boucles non conditionnelles sont les boucles pour lesquelles on connaît à l'avance le nombre d'itérations. Elle sont décrites en pseudo-langage par

« Pour ... Faire .... FinPour ».

3 Boucles conditionnelles

Les boucles conditionnelles sont les boucles pour lesquelles on ne connaît pas à l'avance le nombre d'itérations. Elle sont décrites en pseudo-langage par

« Tant que ... Faire .... FinTantQue ».

4 Notion d'invariant de boucle

Pour justifier le résultat d'une opération dans une boucle, on doit exhiber un **invariant de boucle** : c'est une propriété portant sur les variables intervenant dans la boucle et qui doit être vérifiée à chaque étape de la boucle.

En général on peut formuler un invariant pour l'entrée de la boucle et un invariant pour la sortie de la boucle. Évidemment, ce qui est valable à une sortie l'est encore à l'entrée suivante.

L'invariant se prouve par récurrence et permet de justifier que l'algorithme renvoie bien ce qui est demandé.

Exemples

E1 – Calcul de la somme des entiers entre 1 et 150

```
somme ← 0
Pour i allant de 1 à 150 Faire
    somme ← somme + i
FinPour
Retourner somme
```

Invariant en entrée :

Invariant en sortie :

Démonstration par récurrence.

L'algorithme est valide :

E 2 – Calcul de liste des chiffres en base 10

```
chiffres10(n)
    Si n≠ 0 Alors
        L ← liste vide
        x ← n
        Tant que                    Faire
            Ajouter                 à L
            x ←
        FinTq
    Sinon
        L ← [0]
    FinSi
    Retourner L
```

Invariant :

Démonstration par récurrence.

L'algorithme est valide :

5 Terminaison des boucles

Dans le cas des boucles conditionnelles, il est impératif de justifier la terminaison de celles-ci. L'argument le plus courant est un argument de suite d'entiers strictement croissante majorée ou de suite d'entiers strictement décroissante minorée.

Exemple : Terminaison de la boucle de l'exercice précédent

6 Exercices

Exercices

Ex 1 – Écrire, en pseudo-langage puis en Python, une fonction `deg2rad(angle)` qui demande un angle `angle` en degrés en entrée et renvoie l'angle en radians correspondant.

Ex 2 – Même question avec `rad2deg(angle)`.

Ex 3 – Écrire une fonction `pythagore(a,b,c)` permettant de savoir si le triplet `(a,b,c)` est pythagoricien.

Ex 4 – Écrire une fonction `valeur\_absolue(x)` qui porte bien son nom.

Ex 5 –
1. Écrire une fonction `limitation(lieu)` renvoyant la limite de vitesse pour `lieu` parmi 'agglomération', 'route', 'voie rapide', 'autoroute'. Prévoir le cas où `lieu` n'est pas donné correctement.
2. Écrire une fonction `flash\_radar(vitesse,lieu)` renvoyant un booléen.

Ex 6 – Écrire des fonctions `non(booleen)`, `ou(bool1,bool2)`, `et(bool1,bool2)` sans utiliser `not`, `and` et `or`.

Ex 7 – Donner les algorithmes, préciser des invariants de boucles et justifier la validité.

1. `factorielle(n)`.
2. `puissance(x,n)` calcule x^n.
3. `somme(L)` calcule la somme des éléments de la liste `L`.
4. `maximum(L)` renvoie la valeur maximale parmi les éléments de la liste `L`.
5. `occurence(x,L)` compte le nombre d'occurrences de `x` dans la liste `L`.
6. `miroir(chaine)` renvoyant le mot miroir de `chaine`.
7. `fibonacci(n)` renvoyant le n^e terme de la suite de Fibonacci.

Ex 8 – Donner les algorithmes, préciser des invariants et justifier la terminaison de boucles et la validité des algorithmes.

1. `pppuissance(n)` calcule la plus petite puissance de 2 supérieure à n.
2. `palindrome(chaine)` permet de savoir si `chaine` est un palindrome.
3. `apparition(x,L)` permet de savoir si `x` apparaît dans `L` et renvoie sa position si c'est le cas.
4. On part d'un entier u_0 et on considère la suite définie par $u_{n+1} = \frac{u_n}{2}$ si u_n est pair et $u_{n+1} = 3u_n + 1$ sinon.
   Cette suite et appelée suite de Syracuse. On constate qu'au bout d'un nombre fini d'étape, on finit toujours par trouver 1 (puis 4 puis 2 puis 1 etc.), mais on n'a jamais réussi à le démontrer.
   Écrire une fonction `syracuse(u0)` renvoyant le rang de la première apparition de 1 dans la suite.
5. `est\_fact(n)` permet de savoir si `n` est une factorielle.

III NOTION DE COMPLEXITÉ

1 Définition

Définition : Complexité

Mesure de l'efficacité d'un algorithme, si possible indépendamment d'un langage de programmation et de la machine.

- **Complexité spatiale** : place occupée en mémoire
- **Complexité temporelle** : nombre d'opérations élémentaires

Qu'est-ce qu'une opération élémentaire ?

C'est une notion relative qui dépend du contexte. Suivant l'algorithme que l'on considère on peut décider de compter certaines opérations plutôt que d'autres (considérées comme négligeables).

Les opérations les plus basiques prennent, suivant les machines et les langages, entre 1 ns (10^{-9} s) et 1 μs (10^{-6} s) à être exécutées :

- addition, soustraction, multiplication, division, modulo sur des entiers ou des flottants,
- affectation simple,
- accès aux éléments d'un tableau
- modification des éléments d'un tableau
- taille d'un tableau
- La méthode `append` sur une liste Python peut être considérée comme une opération basique.

2 Notion de tableau

Définition : Tableau

Un tableau informatique est une zone contiguë en mémoire découpée en cases de même taille auxquelles on accède à coût constant.

Ce qui y ressemble le plus en Python ? Les listes. Mais hétérogènes (plusieurs types possibles), extensibles à moindre coût...

3 Augmentation de la taille des données

L'idée est de mesurer l'effet de l'augmentation des données d'un algorithme sur le temps d'exécution.

Exemple

On peut trier un tableau de données de taille n en n^2 opérations élémentaires si on le trie « comme un jeu de carte », et en $n \ln n$ si on s'y prend mieux. Quelle est la différence ?

Imaginons que l'on trie des pages web (Google...) : de l'ordre de 10 000 000, ou des fiches clients d'Amazon (150 000 000 par semaine !).

Si l'on **double** ces chiffres :

- Avec le premier tri, on passe de $(10^7)^2 = 10^{14}$ à 4×10^{14} (ajout de 300 000 milliards !) On multiplie par 4

les opérations à effectuer (et donc le temps d'exécution).
- Avec le second, on passe de $\simeq 1,5 \times 10^9$ à $\simeq 3 \times 10^9$ (ajout de 1,5 milliards), ce qui ne fait ici que doubler le nombre d'opérations...

4 Comparatif

Chaque valeur est multipliée par 10^{-6} s comme majorant du temps d'exécution d'une opération élémentaire.

Croissance	n	10	50	100	500	1000
logarith. →	$\ln n$	2 μs	4 μs	4,6 μs	6 μs	7 μs
	$\sqrt{n}$	3 μs	7 μs	10 μs	20 μs	30 μs
linéaire →	n	10 μs	50 μs	100 μs	0,5 ms	1 ms
	$n\ln n$	20 μs	200 μs	500 μs	3 ms	7 ms
polyn. →	n^2	100 μs	2,5 ms	10 ms	0,25 s	1s
polyn. →	n^3	1 ms	0,1 s	1 s	2 min	16 min
expon. →	2^n	1 ms	36 a	$4 \cdot 10^6$ a	10^{137} a	$3 \cdot 10^{287}$ a
	$n!$	4 s	10^{51} a	$3 \cdot 10^{144}$ a	$3 \cdot 10^{1121}$ a	$3 \cdot 10^{2554}$ a

Ordres de grandeurs :
- **Âge de l'univers** : $13,8 \times 10^9$ années
- **Nombre d'atomes dans l'univers** : 10^{80}.

Échelle de croissances comparées :

$$\ln n << \sqrt{n} << n << n\ln n << n^2 << n^3 << 2^n << n! << n^n$$

n maximum pour un temps d'exécution d'une seconde :

$\ln n$	$\sqrt{n}$	n	$n\ln n$	n^2	n^3	2^n	$n!$
$\simeq 10^{400\,000}$	10^{12}	10^6	87848	10^3	100	20	10

5 Notation O et Θ

Définition : Notations O et Θ

On suppose que $f(n) \geqslant 0$ et $g(n) \geqslant 0$ pour tout entier n.

On dit que f est un « grand Ô » de g et on note $f = O(g)$ lorsqu'il existe un rang n_0 et une constante $c > 0$ tels que

$$\forall\, n \geqslant n_0,\quad f(n) \leqslant c \times g(n)$$

On dit que f est un « theta » de g et on note $f = \Theta(g)$ lorsqu'il existe un rang n_0 et des constantes $c_1, c_2 > 0$ tels que

$$\forall\, n \geqslant n_0,\quad c_1 \times g(n) \leqslant f(n) \leqslant c_2 \times g(n)$$

Il est facile de voir que si $f(n)$ est une somme de fonction, elle est un O de celle qui croît le plus rapidement.

Par exemple, $n^2 + n + 1 = O(n^2)$. On a même plus précisément $n^2 + n + 1 = \Theta(n^2)$.

6 En pratique

Dans la pratique, on peut se contenter d'une O comme ordre de grandeur de la complexité.

Il n'est pas toujours possible d'exprimer la complexité dans tous les cas. On peut alors chercher

- un minorant (complexité dans le meilleur cas)
- un majorant (complexité dans le pire cas)
- la complexité en moyenne : mais c'est plus difficile, cela fait intervenir des probabilités, et ce n'est pas au programme.

7 Cas des listes en Python

Lu sur https://wiki.python.org/moin/TimeComplexity :

Generally, 'n' is the number of elements currently in the container. 'k' is either the value of a parameter or the number of elements in the parameter.

Operation	Average Case	Amortized Worst Case
Copy	$O(n)$	$O(n)$
Append[1]	$O(1)$	$O(1)$
Insert	$O(n)$	$O(n)$
Get Item	$O(1)$	$O(1)$
Set Item	$O(1)$	$O(1)$
Delete Item	$O(n)$	$O(n)$
Iteration	$O(n)$	$O(n)$
Get Slice	$O(k)$	$O(k)$
Del Slice	$O(n)$	$O(n)$
Set Slice	$O(k+n)$	$O(k+n)$
Extend[1]	$O(k)$	$O(k)$
Sort	$O(n \log n)$	$O(n \log n)$
Multiply	$O(nk)$	$O(nk)$
x in s	$O(n)$	
min(s), max(s)	$O(n)$	
Get Length	$O(1)$	$O(1)$

(1) = These operations rely on the "Amortized" part of "Amortized Worst Case". Individual actions may take surprisingly long, depending on the history of the container.

8 Exercices

Calculer les complexités des algorithmes vu dans les exercices sur les boucles `Pour` et `Tant que`.