

Conception de boucles

Fabrice Lembrez - PSI*

Lycée Pierre de Fermat

Pour se remettre en tête des éléments de base.
Chapitre I du poly, sections III et IV.

- 1 Boucles et parcours
 - Les paradigmes de programmation
 - Choisir un mode de parcours
 - Agir au fil de l'eau
- 2 Complexité : les bases
- 3 Petits points de syntaxe pour les instructions de contrôle

Les paradigmes de programmation

Prochaine séance : quelques mots sur les logiques générales d'écriture de programmes

Vous en pratiquez essentiellement deux

- l'écriture de boucles (programmation impérative)
- la programmation récursive

Les paradigmes de programmation

Prochaine séance : quelques mots sur les logiques générales d'écriture de programmes

Vous en pratiquez essentiellement deux

- l'écriture de boucles (programmation impérative)
- la programmation récursive

Choix d'une logique de travail : parcours impératif/logique récursive

En impératif, on pilote un parcours :

- de quel type ?
- piloté par quelles variables ?
- avec quel type d'arrêt ?

En récursif l'idée est de se "ramener à un cas plus simple".

- 1 Boucles et parcours
 - Les paradigmes de programmation
 - Choisir un mode de parcours
 - Agir au fil de l'eau
- 2 Complexité : les bases
- 3 Petits points de syntaxe pour les instructions de contrôle

Choisir un mode de parcours

Types de parcours :

- un indice, parcours prédéterminé : for
- un indice, arrêt en fonction de la situation : while
- deux listes et/ou deux indices

Choisir un mode de parcours

Types de parcours :

- un indice, parcours prédéterminé : for
- un indice, arrêt en fonction de la situation : while
- deux listes et/ou deux indices

parcours parallèles on veut appeler $s1[i]$ et $s2[i]$ pour le même i

parcours imbriqués pour chaque i , parcours de j

parcours concurrents soit i soit j augmente de 1

Choisir un mode de parcours

Types de parcours :

- un indice, parcours prédéterminé : for
- un indice, arrêt en fonction de la situation : while
- deux listes et/ou deux indices
 - parcours parallèles on veut appeler $s1[i]$ et $s2[i]$ pour le même i
 - parcours imbriqués pour chaque i , parcours de j
 - parcours concurrents soit i soit j augmente de 1
- parcours partiel (exemple pour la dichotomie)
- parcours suivant une structure (arbre, graphe...) cf plus tard

Ecrire la boucle

Un conseil important : respecter cet ordre

- ❶ La structure "pure" : structure de contrôle, indices, test d'arrêt
- ❷ "Cœur de boucle" : mettre toutes les variables à jour
- ❸ Initialisation et conclusion adaptées aux choix précédents

En effet il est plus facile de penser l'étape générale d'abord !

- 1 Boucles et parcours
 - Les paradigmes de programmation
 - Choisir un mode de parcours
 - Agir au fil de l'eau
- 2 Complexité : les bases
- 3 Petits points de syntaxe pour les instructions de contrôle

Les boucles : agir au fil de l'eau

Un exemple très basique : soit une suite $(u_n)_{n \in \mathbb{N}}$,

on note $U_n = \sum_{k=0}^n u_k$ la somme partielle,

chercher $M_N = \max_{n \in \llbracket 0, N \rrbracket} U_n$ pour un N donné.

Les boucles : agir au fil de l'eau

Un exemple très basique : soit une suite $(u_n)_{n \in \mathbb{N}}$,

on note $U_n = \sum_{k=0}^n u_k$ la somme partielle,

chercher $M_N = \max_{n \in \llbracket 0, N \rrbracket} U_n$ pour un N donné.

Idée idiote : recommencer le calcul de chaque U_N à zéro

Idée correcte mais maladroite :

- calculer et mémoriser les U_N dans une liste avec seule boucle
- chercher le max de cette liste

Les boucles : agir au fil de l'eau

Un exemple très basique : soit une suite $(u_n)_{n \in \mathbb{N}}$,

on note $U_n = \sum_{k=0}^n u_k$ la somme partielle,

chercher $M_N = \max_{n \in \llbracket 0, N \rrbracket} U_n$ pour un N donné.

Idée idiote : recommencer le calcul de chaque U_N à zéro

Idée correcte mais maladroite :

- calculer et mémoriser les U_N dans une liste avec seule boucle
- chercher le max de cette liste

Démarche à privilégier : tout faire dès que possible

Toujours chercher à faire de préférence un seul passage en lecture en menant toutes les actions demandées. On évitera donc "un passage pour lire et mémoriser" suivi de "un passage pour traiter".

- 1 Boucles et parcours
- 2 Complexité : les bases
 - Terminaison, correction, complexité
 - Complexité temporelle
 - Une idée des temps de calcul concrets
 - Estimer la complexité d'un programme impératif
 - Complexité spatiale
- 3 Petits points de syntaxe pour les instructions de contrôle

Terminaison, correction, complexité

Déf - Terminaison, correction, complexité

On examine l'algorithme pour toutes les données valides possibles (les "instances")

- est-ce qu'il "termine" (en temps fini)
- est-ce qu'il "est correct" (effet conforme à la demande)
 preuve : souvent via un "invariant de boucle"
- quelle est sa complexité (spatiale/temporelle) ?

On prouve souvent la terminaison en même temps qu'on évalue la complexité temporelle.

1 Boucles et parcours

2 Complexité : les bases

- Terminaison, correction, complexité
- **Complexité temporelle**
- Une idée des temps de calcul concrets
- Estimer la complexité d'un programme impératif
- Complexité spatiale

3 Petits points de syntaxe pour les instructions de contrôle

Complexité temporelle

Déf - Complexité temporelle

Temps d'exécution théorique d'un algorithme, compté en "opérations élémentaires".

Modèle de calcul le plus courant :

- conteneur de longueur connue n (ou $n \times p$),
- contenu "simple" : des nombres entiers ou réels.
- opérations : lire ou affecter une case, ou opération "classique" sur les nombres.

Nombre précis des opérations sans intérêt, on raisonne plutôt en d'ordre de grandeur : notation $O(\cdot)$.

Complexité temporelle : ordres de grandeur

Qualificatifs classiques

$O(1)$ Coût constant

$O(\ln n)$ Coût quasi-constant

$O(n)$ Coût linéaire

$O(n \ln n)$ Coût quasi-linéaire

$O(n^2)$ Coût quadratique

$O(2^n)$ Coût exponentiel

- 1 Boucles et parcours
- 2 **Complexité : les bases**
 - Terminaison, correction, complexité
 - Complexité temporelle
 - **Une idée des temps de calcul concrets**
 - Estimer la complexité d'un programme impératif
 - Complexité spatiale
- 3 Petits points de syntaxe pour les instructions de contrôle

Complexité temporelle : ordres de grandeur

	$\log_2 n$	n	$n \cdot \log_2 n$	n^2	n^3	2^n
10^2	$0.66 \mu s$	$10 \mu s$	$66 \mu s$	1 ms	0.1s	$4 \cdot 10^{15}$ ans
10^3	$1 \mu s$	$100 \mu s$	1 ms	0.1s	100 s	
10^4	$1.3 \mu s$	1 ms	13 ms	10 s	1 jour	
10^5	$1.6 \mu s$	10 ms	0.1 ms	16 mn	3 ans	
10^6	$2 \mu s$	100 ms	2 s	1 jour	3100 ans	
10^7	$2.3 \mu s$	1 s	23 s	115 jours	$3 \cdot 10^6$ ans	

Sous l'hypothèse une opération élémentaire $\approx 10^{-7}$ s.

Justification 10^{-7} secondes : fréquence des processeurs en GHz (donc 10^{-9} s).

NB âge de l'univers $\approx 10^{10}$ années... Les algorithmes de complexité exponentielle deviennent très rapidement totalement irréalisables.

En sens inverse, on comprend le "quasi" pour le log...

1 Boucles et parcours

2 Complexité : les bases

- Terminaison, correction, complexité
- Complexité temporelle
- Une idée des temps de calcul concrets
- **Estimer la complexité d'un programme impératif**
- Complexité spatiale

3 Petits points de syntaxe pour les instructions de contrôle

Estimer la complexité (impératif)

Majoration grossière de la complexité

Analyser la structure

- Majorer le nombre d'exécutions de chaque boucle
- Repérer la structure d'ensemble
 - Boucles successives : les complexités s'ajoutent
 - Boucles imbriquées : les complexités se multiplient
 - Appel de fonction : imbriquer mentalement le bloc de code
- Eviter ou surveiller les opérations "non élémentaires", par exemple :
`l=[i**2 for i in range(n)]`

Exemple : échelonnement par pivot de Gauss sur une matrice $n \times p$ en supposant qu'on n'a pas besoin d'échanges de lignes (pivots qui apparaissent directement sur la diagonale).

Situations plus difficiles

On en verra en exercice

- On peut demander un comptage plus détaillé, par exemple une minoration
- On peut demander une majoration plus fine que celle attendue
- soit on a une réponse formelle en donnant un bon "variant de boucle"

Ex. $i+j$ pour un parcours "concurrent"

- plus souvent il y a une limite "physique" au nombre de tours de boucles

Ex. parcours en largeur d'abord : un sommet est visité une seule fois

- 1 Boucles et parcours
- 2 Complexité : les bases
 - Terminaison, correction, complexité
 - Complexité temporelle
 - Une idée des temps de calcul concrets
 - Estimer la complexité d'un programme impératif
 - Complexité spatiale
- 3 Petits points de syntaxe pour les instructions de contrôle

Complexité spatiale

Déf - Complexité spatiale

Mesure de l'espace mémoire mobilisé *par les variables introduites pour résoudre le problème.*

On note encore $O(1)$, $O(n)$, etc... (comptage du nombre de cases de tableaux). Il s'agit là encore d'ordres de grandeur : introduire 3,5 ou 10 variables de type entier ou réel pour la lisibilité du code n'est pas grave, voire est souhaitable. En revanche créer des tableaux de taille n de façon gratuite, ce n'est pas bien.

Exemple : pour une suite récurrente $u_{n+1} = f(u_n)$, faire la différence entre

- "renvoyer u_n "
- "renvoyer $(u_0, \dots, u_n)$ "
- "tracer la ligne brisée associée à $(u_0, \dots, u_n)$ "

Et faire aussi la différence avec le cas d'une suite $u_{n+1} = f_n(u_0, \dots, u_n)$.

Complexité spatiale ou temporelle

Priorité à la complexité temporelle

On cherche à minimiser **l'ordre de grandeur** de la complexité temporelle. Ensuite, dans ce cadre, le moins de complexité spatiale c'est le mieux. Rendre le code illisible pour un gain limité n'est pas souhaitable.

Plan

- 1 Boucles et parcours
- 2 Complexité : les bases
- 3 Petits points de syntaxe pour les instructions de contrôle
 - Les booléens dans le "if" et le "while"
 - Instruction composée if/elif/else

Booléens (True/False)

Savoir ce qu'on manipule !

- Une "condition" = un booléen
Exemple : mettre `if estPremier(n)==True` est maladroit
il faut écrire `if estPremier(n)`
- Penser à l'opérateur booléen "not" pour nier une condition
`if not estPremier(n)`
- Utiliser `==` pour une condition et non pas `=` (affectation)

Evaluation paresseuse et dépassement

Avant de lire une case de tableau : test de non dépassement

En général sous la forme d'un "and" avec évaluation paresseuse.

Exemple-type: remise en ordre de cases successives

```
if i+1<len(s) and s[i]<s[i+1]: # respecter l'ordre!  
    s[i+1],s[i]=s[i],s[i+1]
```

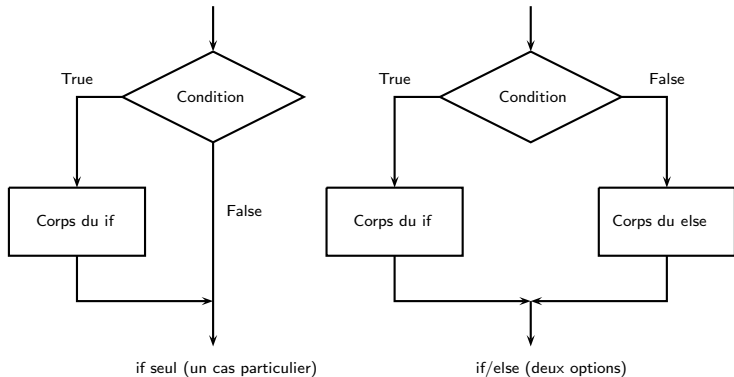
Plan

- 1 Boucles et parcours
- 2 Complexité : les bases
- 3 Petits points de syntaxe pour les instructions de contrôle
 - Les booléens dans le "if" et le "while"
 - Instruction composée if/elif/else

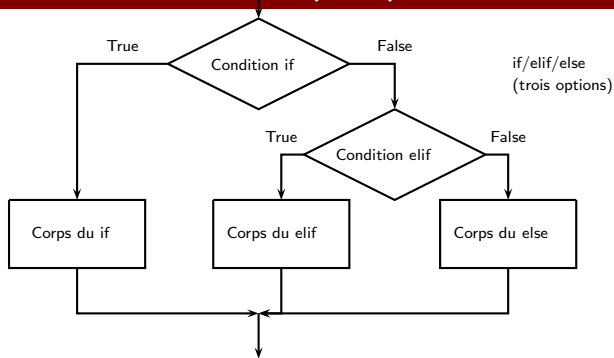
Instruction de base : if, ou if/else

if seul = isoler une situation spécifique, puis reprendre le "flot" des instructions.

if/else = séparer deux cas (True/False), reprendre.



Plusieurs choix exclusifs : le if/elif/else



Cela donne une séparation en "cas" disjoints.
On peut avoir plusieurs elif successifs.

- ▷ S'il n'y a rien à exécuter dans le "cas else" ?
- ▷ S'il n'y a rien à exécuter dans le cas "if" ?

Que choisir ?

Formellement if/elif/else = imbriquer deux if.

- Disjonction de cas : if, elif, else
- Tests successifs indépendants : succession de if

La distinction est importante : si un premier if me pousse à modifier certaines valeurs des variables, le deuxième if s'effectuera avec les nouvelles valeurs.

Que choisir ?

Formellement `if/elif/else` = imbriquer deux `if`.

- Disjonction de cas : `if`, `elif`, `else`
- Tests successifs indépendants : succession de `if`

La distinction est importante : si un premier `if` me pousse à modifier certaines valeurs des variables, le deuxième `if` s'effectuera avec les nouvelles valeurs.

Imbrication ou succession

Dans un algorithme, bien découper les actions par étapes successives. Si pour UNE étape donnée il y a plusieurs cas ne pas faire de `if` successifs, mais `if/elif/else`.