Chapitre n° 11 : présentation de l'algorithmique

Compétences attendues

- Définition des notions de correction, de terminaison et de coût d'un algorithme.
- Appliquer ces notions à quelques algorithmes classiques simples.
- Savoir écrire des algorithmes utilisant le parcours séquentiel d'un tableau.
- Montrer la terminaison d'un algorithme de recherche dichotomique dans un tableau trié à l'aide d'un variant de boucle.

1 Les outils

1.1 Compteurs et accumulateurs

Définition : *un compteur, dont le rôle est de compter, est une variable initialisée à zéro qui est incrémentée d'une unité à chaque passage dans une boucle, éventuellement suite à un test.*

Exemples : compteur et boucle conditionnelle.
Sans test :

```
def taille(n):
    cpt = 0
    while n > 0:
        cpt += 1
        n = n // 2
    return cpt
```

Dans cet exemple, on compte le nombre de divisions euclidiennes successives de n par 2, jusqu'à arriver à un quotient nul. On obtient donc le nombre de chiffres dans l'écriture binaire de n.

Avec test :

```
def nombre_de_1(n):
    cpt = 0
    while n > 0:
        if n % 2 == 1:
            cpt += 1
        n = n // 2
    return cpt
```

Le programme est identique au précédent, mais on incrémente le compteur uniquement quand un reste vaut 1. On compte donc le nombre de 1 dans l'écriture binaire de n.

Exemple : compteur et boucle inconditionnelle.
Le nombre de passage dans la boucle étant connu à l'avance, il est inutile d'utiliser un compteur sans test. En revanche, il peut être utile d'utiliser un compteur avec test.

```
def diviseur(n):
    cpt = 0
    for d in range(1, n +1):
        if n % d == 0:
            cpt += 1
    return cpt
```

Le compteur est incrémenté quand n est divisible par d, donc la fonction renvoie le nombre de diviseur de n.

Définition : *un accumulateur est semblable à un compteur, mais il est en général incrémenté d'une valeur différente de 1. Il peut également être décrémenté, par exemple dans le calcul d'une somme de termes.*

Exemple : *liste* est une liste de nombres.
Sans test :

```
def somme(liste):
    acc = 0
    for x in liste:
        acc += x
    return acc
```

La fonction renvoie la somme des nombres contenus dans la liste.

Avec test :

```
def somme(liste):
    acc = 0
    for x in liste:
        if x % 2 == 0:
            acc += x
    return acc
```

La fonction renvoie la somme des nombres pairs contenus dans la liste.

1.2 Permutation de valeurs

Les permutations de valeurs sont utilisées très couramment. Par exemple, ces échanges sont présents dans les algorithmes de tris que nous verrons dans le

prochain chapitre.

Rappel, exercice : que valent les variables à l'issue du code suivant ? Comment éviter ce problème ?

```
var1 = 17
var2 = 23
var1 = var2
var2 = var1
```

Les deux variables ont la même valeur, 23. pour éviter ce problème, il faut utiliser une variable temporaire pour stocker une valeur.

```
var1 = 17
var2 = 23
temp = var1
var1 = var2
var2 = temp
```

Dans ce cas, var1 *vaut 23 et* var2 *vaut 17, on a bien permuté les deux valeurs.*

Remarque : grâce au tuples, il est possible d'alléger la procédure avec Python.

```
var1, var2 = var2, var1
```

1.3 Tests et boucles

*Les tests (*if . . . elif . . . else*) et les boucles (*while *et* for*), y compris imbriquées, sont très utilisées dans les algorithmes.*

2 Validité d'un algorithme itératif

2.1 Présentation

Lorsqu'on écrit un algorithme, il est impératif de vérifier que cet algorithme va produire un résultat en un temps fini et que ce résultat sera correct dans le sens où il sera conforme à une spécification précise. Nous dirons alors que l'algorithme est valide.

2.2 Correction

Un algorithme itératif est construit avec des boucles. Pour prouver qu'il est correct, nous disposons de la notion d'invariant de boucle.

Définition : *un invariant d'une boucle est une propriété qui est vérifiée avant l'entrée dans une boucle, à chaque passage dans cette boucle et à la sortie de*

cette boucle.

Pour démontrer qu'une propriété est un invariant de boucle, on utilise un raisonnement semblable au raisonnement par récurrence utilisé en mathématiques.

- *Initialisation : on commence par vérifier que la propriété est vraie avant l'entrée dans la boucle.*
- *Hérédité : on prouve ensuite que si la propriété est vraie avant un passage dans la boucle, alors elle est vraie après ce passage.*
- *Conclusion : on peut alors conclure que la propriété est vraie à la sortie de la boucle.*

Exemple : voici un algorithme de calcul avec une boucle conditionnelle et deux variables a et b ayant pour valeur un entier naturel.

```
1 m = 0
2 p = 0
3 while m < a:
4     m += 1
5     p += b
```

Nous allons montrer que $p = m \times b$ est un invariant de la boucle *while*.

- Initialisation : avant le premier passage dans la boucle, $m = 0$ et $p = 0$, donc l'égalité $p = m \times b$ est vraie.
- Hérédité : supposons que $p = m \times b$ avant un passage dans la boucle.
  Après le passage, les nouvelles valeurs des variables valent $m' = m + 1$ et $p' = p + 1$.
  Donc $p' = m \times b + b = (m + 1) \times b = m' \times b$.
  Par conséquent, la propriété est vraie après ce passage dans la boucle.
- Conclusion : nous pouvons donc conclure qu'à la sortie de la boucle, $p = m \times b$. Or, comme la dernière valeur de la variable m est $m = a$, nous avons finalement obtenu le produit $p = a \times b$.

2.3 Terminaison

Un algorithme ne doit toujours comporter qu'un nombre fini d'étapes.

Dans le cas où l'algorithme ne comporte pas de boucle, ou uniquement des boucles inconditionnelles (boucles *for*, dont le nombre de passages dans la boucle est déterminé), alors le nombre d'étapes est déterminé et donc fini.

Afin de prouver la terminaison d'un algorithme itératif comportant au moins une boucle inconditionnelle (boucles *while*), nous utiliserons la notion de *variant*.

Définition : *un variant est une expression, la plus simple étant une variable, telle que la suite formée par les valeurs de cette expression au cours des itérations converge en un nombre fini d'étapes vers une valeur satisfaisant la condition d'arrêt.*

Exemple : considérons le code suivant où la valeur de la variable a est un nombre quelconque.

```
x = 0
while x ** 2 < a:
    x += 1
```

Si la valeur de a est négative ou nulle, il n'y a aucun passage dans la boucle. Sinon, le variant choisi est la suite des valeurs de la variable x i.e. $0, 1, 2, \ldots, n$. Dans ce cas, n est la première valeur supérieure ou égale à la racine carrée de a. Le nombre de passage dans la boucle est donc fini.

Exercice : revenons sur le code de la sous-partie précédente. Nous avions prouvé la correction de l'algorithme, à savoir qu'en sortie de boucle la valeur p valait le produit $a \times b$. Montrer maintenant la terminaison de cet algorithme.

Correction : nous choisissons comme variant la variable m. Cette variable prend comme valeurs successives $0, 1, 2, \ldots, a$. Il y a donc exactement a passages dans la boucle, ce qui prouve la terminaison.

3 Coût d'un algorithme

3.1 Présentation

Les exemples de programmes que nous avons vu depuis le début de l'année s'exécutaient quasiment instantanément. Mais lorsque un programme doit traiter de très grandes quantités de données, le temps d'exécution peut être très long.

Considérons l'exemple d'un programme qui doit gérer une liste de 10^7 éléments puis une liste de 10^8 éléments. Est-ce que la durée d'exécution sera proportionnelle au nombre d'éléments, et simplement multipliée par 10 ?

De façon générale, lorsqu'on écrit un algorithme, il faut réfléchir au rapport entre le temps d'exécution et la taille de la liste.

Les réponses sont variées, et dépendent de l'algorithme et de la liste. Un algorithme peut être plus rapide qu'un autre avec une liste donnée et plus lent avec une autre liste. Parfois, un programme peut même être plus lent avec une liste plus courte.

Pour traiter un même problème, nous pouvons avoir différents algorithmes. Mais un même algorithme peut avoir des temps d'exécutions différents selon

le langage de programmation utilisé et la machine sur laquelle il est exécuté. L'étude n'est pas simple à réaliser, et *pour comparer deux algorithmes, nous allons ici nous concentrer sur le nombre d'opérations à effectuer, en essayant d'évaluer un ordre de grandeur de ce nombre en fonction de la taille des données. Nous parlerons de coût d'un algorithme ou de sa complexité.*

Ce coût pouvant être très différent pour une même taille de données, nous nous placerons dans le pire des cas : celui où le coût est le plus important.

3.2 Exemples

Considérons à nouveau l'algorithme suivant :

```
1 m = 0
2 p = 0
3 while m < a:
4     m += 1
5     p += b
```

Les passages dans la boule ont lieu pour les valeurs de m égales à $0, 1, 2, \ldots, a-1$ si la valeur de la variable a est un entier naturel strictement positif. Nous avons dans ce cas exactement a passages dans la boucle.

A chaque passage, il y a deux additions et deux affectations, donc un total de quatre opérations. Au final, le programme réalise $4a$ opérations : nous dirons que le coût est proportionnel à a ou qu'il est linéaire.

Remarque : avec une boucle *for*, le nombre de passages dans la boucle est bien déterminé, le calcul du nombre d'opération est simple. Il peut-être plus compliqué avec une boucle *while*, puisque le nombre de passages dans une boucle varie selon les cas pour une même taille de données. Il faut alors identifier le pire des cas, c'est à dire celui où le nombre de passages est maximal.

3.3 Complexité linéaire

Définition : *soit n la taille d'une donnée. Si le nombre d'opérations à effectuer peut s'écrire $\alpha n + \beta$, avec α et β réels, $\alpha > 0$, alors l'algorithme a un coût linéaire ou une complexité linéaire.*

3.4 Complexité quadratique

Définition : *soit n la taille d'une donnée. Si le nombre d'opérations à effectuer peut s'écrire $\alpha n^2 + \beta n + \gamma$, avec α, β et γ réels, $\alpha > 0$, alors l'algorithme a un coût quadratique ou une complexité quadratique.*

Dans le cas de deux boucles *for* imbriquées, il existe trois cas typiques. Dans un cas, le coût est linéaire, dans les deux autres cas il est quadratique.

Premier cas : n est la taille de la donnée, k est un nombre fixé.
Les pointillés représentent un nombre fixe d'opérations.

```
for i in range(n):
    ... # q opérations
    for j in range(k):
        ... # r opérations
```

Il y a n passages dans la boucle externe.
A chaque passage, nous avons un nombre fixe q d'opérations, puis k passages dans la boucle interne.
Dans la boucle interne, il y a un nombre fixe r d'opérations.
Donc, pour chaque valeur de i il y a $q + k \times r = \alpha$ opérations.
Le nombre total d'opérations est αn et le coût est linéaire.

Second cas : n est la taille de la donnée.
Les pointillés représentent un nombre fixe d'opérations.

```
for i in range(n):
    ... # q opérations
    for j in range(n):
        ... # r opérations
```

Il y a n passages dans la boucle externe.
A chaque passage, nous avons un nombre fixe q d'opérations, puis n passages dans la boucle interne.
Dans la boucle interne, il y a un nombre fixe r d'opérations.
Donc, pour chaque valeur de i il y a $q + n \times r$ opérations.
Le nombre total d'opérations est $n \times (q + n \times r) = r \times n^2 + q \times n$
Le coût est quadratique.

Troisième cas : n est la taille de la donnée.
Les pointillés représentent un nombre fixe d'opérations.

```
for i in range(n):
    ... # q opérations
    for j in range(i):
        ... # r opérations
```

Il y a n passages dans la boucle externe.
A chaque passage, nous avons un nombre fixe q d'opérations, puis i passages dans la boucle interne.
Dans la boucle interne, il y a un nombre fixe r d'opérations.
Donc, pour chaque valeur de i il y a $q + i \times r$ opérations.
Les valeurs de i sont successivement $0, 1, 2, \ldots, n-1$.
Le nombre total d'opérations est $q+(q+1\times r)+(q+2\times r)+\cdots+(q+(n-1)\times r)$.
i.e. $n \times q + r \times (1 + 2 + \cdots + n - 1)$.
Or, la somme des $n-1$ est connue en mathématiques, elle vaut $\frac{n\times(n-1)}{2}$.

Donc le nombre total d'opération vaut :
$n \times q + r \times \frac{n \times (n-1)}{2} = \frac{r}{2} \times n^2 + (q - \frac{r}{2}) \times n$.
Le résultat est de la forme $\alpha n^2 + \beta n$ donc le coût est quadratique.

4 Parcours séquentiel d'un tableau

Dans cette partie, nous allons voir quelques exemples d'algorithmes qui parcourent les éléments d'un tableau (ou d'une liste).

4.1 Calcul d'une moyenne

Le calcul d'une moyenne se fait à partir de l'algorithme du calcul d'une somme, utilisant un accumulateur.

Voici l'algorithme permettant de calculer la moyenne des éléments de la liste, non vide, appelée *liste*.

```
1  def moyenne(liste):
2      n = len(liste)
3      s = 0
4      for u in liste:
5          s += u
6      return s / n
```

Exercice :

1. Le coût de cet algorithme est-il linéaire, quadratique, ou autre ?
   Il y a 2 affectations puis n additions puis 1 division, donc $n+3$ opérations : le coût est linéaire.

2. Pourquoi est-ce qu'il faut que la liste soit non vide ?
   La fonction renvoie la valeur s/n. Il faut donc que n soit non nul.

4.2 Recherche d'une occurrence

Il s'agit de rechercher de manière séquentielle la présence d'une valeur dans un tableau (ou une liste ou un tuple ou une chaîne de caractères). Cette méthode est également appelée méthode par balayage ou recherche linéaire (*linear search* en anglais).

On cherche une valeur précise, qu'on va comparer successivement à toutes les valeurs du tableau. L'algorithme s'arrête dès que l'élément est trouvé ou si la fin du tableau est atteinte.

```
1  def recherche(x, tableau):
2      n = len(tableau)
3      i = 0
4      while i < n and x != tableau[i]:
5          i += 1
6      if i < n:
```

```
7           return i
```

Cette fonction renvoie l'indice de l'élément recherché, ou ne renvoie rien si l'élément n'est pas dans le tableau.

Pour évaluer le coût de l'algorithme, nous ajoutons aux deux affectations initiales le nombre de comparaisons qui sont effectuées. Au pire cas, il faudra parcourir tout le tableau (par exemple si l'élément n'est pas dans le tableau), et donc effectuer n comparaisons (le tableau comportant n éléments).

Le coût est $n+2$, donc le coût est linéaire en n.

Remarque : en Python, il est possible d'utiliser une boucle for avec une sortie de boucle éventuellement prématurée.

```
1  def recherche(x, tableau):
2      for i in range(len(t)):
3      if tableau[i] == x:
4          return i
```

4.3 Recherche d'un extremum

Le but est de trouver un extremum (maximum ou minimum) dans un tableau.

L'algorithme consiste à supposer que l'extremum provisoire est le premier élément. Puis à parcourir la liste en effectuant une comparaison à chaque fois et éventuellement en changeant la valeur de l'extremum provisoire.

Exemple : fonction recherchant le maximum d'une liste.

```
1  def maximum(liste):
2      maxi = liste[0]
3      for x in liste:
4          if x > maxi:
5              maxi = x
6      return maxi
```

Exercice : écrire le code de la fonction *minimum* qui renvoie le plus petit élément de la liste.

Réponse :

```
1  def minimum(liste):
2      mini = liste[0]
3      for x in liste:
4          if x > mini:
5              mini = x
6      return mini
```

Pour un tableau de n éléments, l'algorithme comprend une affectation, puis il effectue n fois une comparaison, et de 0 à $n-1$ affectations.
Le coût au pire cas est donc $2 \cdot n$: *cet algorithme a un coût linéaire.*

De façon générale, un algorithme basé sur le parcours séquentiel d'une liste, avec un nombre borné d'opérations pour chaque élément, a un coût linéaire.

5 Recherche dichotomique

5.1 Présentation de l'algorithme

La recherche par dichotomie s'effectue dans un tableau trié.

Rappel : Python possède la fonction *sorted* qui prend en argument une liste et renvoie la liste triée, sans modifier la liste d'origine. Une alternative est la méthode *sort*, qui trie la liste sur laquelle elle est appliquée.

```
liste = [4, 1, 3, 2]
liste2 = sorted(liste) # liste2 est triée, mais liste n'est pas modifiée
liste.sorted() # liste est modifiée, elle est triée
```

*Le principe de la dichotomie (*binary search *en anglais) est, à chaque étape, de couper le tableau en deux et d'effectuer un test pour savoir dans quelle partie se trouve l'élément cherché.* Ce principe est également appelé *diviser pour régner* (*divide-and-conquer* en anglais).

Exemple :

```
def dichotomie(x, liste):
    gauche = 0
    droite = len(liste)
    while gauche < droite - 1:
        k = (gauche + droite) // 2
        if x < liste[k]:
            droite = k
        else:
            gauche = k
    if x == liste[gauche]:
        return gauche
    else:
        return False
```

La fonction envoie l'indice de l'élément x recherché.

5.2 Terminaison de l'algorithme

Preuve de la terminaison de l'algorithme avec le variant $droite - gauche$.

Soit n tel que la taille du tableau est inférieure à 2^n.
Donc, initialement, le variant $droite - gauche \leq 2^n$

A chaque passage dans la boucle, le variant $droite - gauche$ est divisé par 2.
Donc après k passages, il est divisé par 2^k, donc $droite - gauche \leq \frac{2^n}{2^k}$.
La suite des valeurs du variant $droite - gauche$ est strictement décroissante, et après n passages, on a donc $droite - gauche \leq \frac{2^n}{2^n}$.
Donc $droite - gauche \leq 1$, donc la boucle *while* s'arrête.

Par exemple, pour un tableau de 100 éléments, il faut 7 étapes ; pour un tableau de 1000 éléments il faut 10 étapes.

Le coût de la recherche dichotomique est de l'ordre du nombre de chiffres de l'écriture binaire de n, donc *nettement inférieur à un coût linéaire.*

5.3 Correction de l'algorithme

Remarque : la preuve de la terminaison de l'algorithme par dichotomie est à connaître et à savoir refaire. En revanche, la preuve de sa correction est présentée ici, mais n'est pas exigible.

Initialisation : avant le premier passage dans la boucle, *gauche* a pour valeur 0 et *droite* a pour valeur $len(liste)$.
Si $x < liste[0]$ ou si $x > liste[droite - 1]$ alors x n'est pas dans la liste, et la recherche n'aboutit pas (la fonction renvoie *False*).

Sinon, $liste[gauche] \leq x \leq liste[droite - 1]$.
Nous considérons alors que nous ajoutons un élément en fin de liste, strictement supérieur à $liste[droite - 1]$, par exemple égal à $liste[droite - 1] + 1$.
Nous choisissons comme invariant de boucle la propriété :
$liste[gauche] \leq x < liste[droite]$.
Cette propriété est vraie avant l'entrée dans la boucle d'après ce qui précède.

Hérédité : nous supposons la propriété vraie avant un passage dans la boucle.
$liste[gauche] \leq x < liste[droite]$.
D'après le choix de k, $liste[gauche] \leq liste[k] < liste[droite]$, car la liste est triée.

Si $x < liste[k]$, on obtient $liste[gauche] \leq x < liste[k]$. Dans ce cas la nouvelle valeur de *droite* est k.
L'invariant $liste[gauche] \leq liste[k] < liste[droite]$ est vrai après le passage dans la boucle.

Sinon $liste[k] \leq x$, on obtient $liste[k] \leq x < liste[droite]$. Dans ce cas la nouvelle valeur de *gauche* est k.
L'invariant $liste[gauche] \leq liste[k] < liste[droite]$ est vrai après le passage dans la boucle.

Conclusion : l'invariant est donc vrai après le dernier passage dans la boucle. Il nous reste alors à examiner l'état des variables.

Lors du dernier passage dans la boucle, $gauche < droite - 1$
i.e. $gauche \leq droite - 2$ i.e. $droite - gauche \geq 2$ i.e. $gauche < k < droite$.

Si $droite$ prend la valeur k ou si $gauche$ prend la valeur k, alors
$droite - gauche > 0$.
Mais s'il n'y pas plus de passage dans la boucle, alors $gauche \geq droite - 1$,
i.e. $droite - gauche \leq 1$.
La seule possibilité pour que $0 < droite - gauche \leq 1$ est $droite - gauche = 1$.
Or $liste[gauche] \leq x < liste[droite]$, donc $liste[gauche] \leq x < liste[gauche + 1]$

En conclusion, nous avons deux possibilités : soit $x = liste[gauche]$, soit x n'est pas dans la liste.