Parcours séquentiel : cours

qkzk

Parcours séquentiel d'un tableau

Attendus : Écrire un algorithme de recherche d'une occurrence sur des valeurs de type quelconque. Écrire un algorithme de recherche d'un extremum, de calcul d'une moyenne.

Commentaire : On montre que le coût est linéaire.

Algorithmes sur les tableaux

Lorsqu'on dispose de plusieurs données similaires (notes à un devoirs, dépenses journalières etc.) il est commode de les ranger dans un tableau.

On est alors amené à effectuer de nombreuses opérations sur ces tableaux.

```
notes = [12, 10, 8, 6, 20]
```

Est-ce qu'un élève a eu 20 ?

La réponse est oui, 20 est le dernier élément du tableau. Pour une machine, répondre à cette question n'est pas immédiat.

En pratique, la seule manière qu'elle ait de répondre est de parcourir le tableau entier et de comparer un par un les éléments avec `20`.

C'est ce qu'on appelle un **parcours séquentiel**.

Outils natifs de Python

Python propose l'instruction `in` qui teste l'appartenance :

```
>>> 20 in tableau
True
>>> 13 in tableau
False
```

Mais comment fait-il ?

Relire la partie précédente si vous n'avez pas deviné, lisez le code source code de Python si vous doutez.

Autre outil natif pour la moyenne

De même, pour calculer la moyenne à ce devoir, on pourrait utiliser :

```
>>> moyenne = sum(tableau) / len(tableau)
>>> moyenne
11.2
```

Mais comment ?

L'autre danger des *outils natifs*, c'est que de telles instructions **n'existent pas dans tous les langages !**

Objectifs

- Nous allons étudier de simples algorithmes qui utilisent le caractère séquentiel d'un tableau.
- Systématiquement, nos algorithmes vont utiliser une seule boucle qui parcourt les éléments du tableau.
- À chaque fois, il faudra adapter ce que nous faisons dans la boucle pour résoudre le problème.

Les tableaux en informatique

Un tableau est une série d'objets, généralement situés côte à côté dans la mémoire.

On suppose pouvoir parcourir le tableau, élément par élément.

Intérêt des tableaux

En enregistrant des objets côte à côte dans la mémoire, on peut accéder très rapidement à l'un d'entre eux par son indice.

Considérons `[5, 7, 3, 8, 12]`.

Ces entiers sont petits, tous entre 0 et 255, on peut utiliser 1 octet (8 bits) par entier.

On les enregistre côte à côte en mémoire à partir de l'adresse mémoire `1234`.

Donc, `5` est enregistré à l'adresse `1234`.

Si chaque case utilise 1 octet, l'adresse suivante, `1235` correspond à la case contenant 7.

Et l'adresse de l'élément d'indice 3 (qui est la valeur 8) est `1234 + 3 = 1237`.

Pour la machine, ces calculs sont très rapides.

Les tableaux en Python

En Python, pour illustrer les tableaux, on utilise les objets `list`.

```
>>> T = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> type(T)
<class 'list'>
```

Constructions de tableaux

Il existe quatre méthodes simples pour construire les tableaux en Python

Construction directe

- En les définissant directement

  ```
  equipe = ["Diego", "Franz", "Michel", "Johann",
  ...       "Lionel", "Christiano"]
  ```

Ajouts successifs

- En ajoutant, élément par élément avec `append`,

  ```
  longueurs = []
  for joueur in equipe:
    longueurs.append(len(joueur))
  ```

  On obtient ici :

  ```
  [5, 5, 6, 6, 6, 10]
  ```

Tableaux par compréhension

- par compréhension,

  ```
  cubes = [x ** 3 for x in [1, 2, 3, 4]]
  ```

`range`

- cas des nombres espacés régulièrement : `range`

  La fonction `range` rentourne un itérable composés de nombres séparés régulièrement.

  `range` accepte de 1 à 3 arguments.

  ```
  range(debut, fin, pas)
  ```

Les nombres entre `debut` (inclu) et `fin` (exclu) séparés de `pas` à chaque fois

`range(3, 19, 4)`

Les nombres entre `3` (inclu) et `19` (exclu) séparés de `4` à chaque fois

On va obtenir un itérable correspondant à `[3, 7, 11, 15]`

`19` n'y figure pas car la borne de droite est toujours exclue.

`range` avec ou sans bornes

- S'il n'y a que deux nombres, ce sont les bornes de début et de fin. Le pas est 1.
- S'il n'y a qu'un nombre, c'est la borne de fin. La borne de début est 0.

`range(6)` retourne l'itérable correspondant à la liste des entiers `[0, 1, 2, 3, 4, 5]`

Parcourir un tableau

Cela demande forcement une boucle.

Rappelons qu'il existe deux types de boucles : `for` et `while` ou bornée et non bonrnée.

1. for

1. **boucles avec `for`** qui parcourt directement un objet *itérable* (`list`, `tuple`, `dict`, etc.).

   ```
   for joueur in tableau:
       faire_quelque_chose_avec(joueur)
   ```

2. **boucles bornée qui parcourt les indices**

   ```
   tab = ["a", "b", "c", "d"]
   for i in range(len(tab)):
     lettre = tab[i]
     print(f"A la position {i} on a la lettre {lettre}")
   ```

   ```
   A la position 0 on a la lettre a
   A la position 1 on a la lettre b
   A la position 2 on a la lettre c
   A la position 3 on a la lettre d
   ```

2. while

1. **boucles avec while** qui utilise une condition d'arrêt. En comptant les éléments

   ```
   compteur = 0
   while compteur < 5:
     print(compteur) # ou faire autre chose...
     compteur = compteur + 1
   ```

   Une boucle avec `while` qui s'arrête doit comporter tous ces éléments.

2. **boucle infinie** : c'est une boucle qui ne s'arrête pas :

- serveur qui attend des messages

  ```
  while True:
    message = serveur_ecoute_message()
    reponse = serveur_traite_message(message)
    envoyer_reponse(reponse)
  ```

- dans un jeu vidéo :

  ```
  while True:
    ecouter_les_actions_joueur()
    calculer_nouvel_etat_jeu()
    dessiner_les_graphismes()
  ```

Algorithmes sur les tableaux

Recherche d'un élément dans un tableau

Contexte : on dispose d'un tableau,

par exemple `T=[0, 1, 2, ..., 10]`.

On veut savoir si un nombre x figure dans le tableau.

L'algorithme produit un booléen on ne s'intéresse donc pas aux indices: `for x in T` convient.

Algorithme :

```
contient (tableau T, objet x) -> booléen:
  Pour chaque élément e de T,
    Si e = x, alors on retourne Vrai
  Si la boucle se termine, on retourne Faux.
```

ATTENTION remarquez bien la position de la condition... on ne renvoie `Faux` que si la boucle se termine.

Exemple

```
T = [2, 5, -4, 12]
```

A-t-on 9 dans le tableau ?

élément	2	5	-4	12
`élément == 9 ?`	Faux	Faux	Faux	Faux

Le parcours de la boucle se termine et l'algorithme retourne Faux.

A-t-on -4 dans le tableau ?

élément	2	5	-4	12
`élément == 9 ?`	Faux	Faux	Vrai	

L'algorithme retourne Vrai. La dernière case du tableau n'est jamais visitée !

Autre exemple : indice d'un élément d'un tableau

On dispose d'un tableau et d'une clé. On veut connaître l'indice de la clé dans le tableau. On répond -1 si la clé n'est pas présente.

Cette fois il faut l'indice : `for i in range(len(T))`

```
indice_de (tableau T, entier cle) -> entier:
  Pour i allant de 0 à longueur de T - 1 (inclus),
      element = T[i]
      Si element == cle, renvoyer i

  Si la boucle se termine, renvoyer -1
```

```
def indice_de(tab: list, cle: int):
    for i in range(len(tab)):
        element = tab[i]
        if element == cle:
            return i
    return -1
```

Notez la position de `return -1`. Il est au même niveau d'indentation que `for ...`.

En effet, il faut attendre d'avoir parcouru tous les éléments pour savoir si la clé est présente ou non.

Résumé

Un *parcours séquentiel* est un algorithme qui n'a besoin que d'une boucle bornée parcourant une collection pour répondre à une question.

- Un de mes élèves est-il déjà monté sur la tour Eiffel ?

  Je peux répondre par un parcours séquentiel. Il suffit de poser la même question à chaque élève.

- Ces deux tableaux `[3, 5, 1, 7]` et `[1, 5, 3, 7]` contiennent-ils les mêmes éléments ?

  Impossible de répondre par une parcours séquentiel. Il faut, **pour chaque élément du premier**, le comparer à **chaque élément du second**. Deux boucles imbriquées : ce n'est pas un parcours séquentiel.

- Lorsqu'un parcourt séquentiel ne nécessite pas d'indice ("L'un d'entre-vous est-il monté sur la tour Eiffel ?"), on utilise `for elt in tableau:`

- Lorsqu'un parcourt séquentiel nécessite de connaître un indice, on utilise `for i in range(len(tableau))`

  Exemple : ces nombres sont-ils rangés par ordre croissant ?

  Pour chaque nombre, le comparer **au suivant**. Pour accéder au **suivant**, je dois connaître l'indice.

Coût du parcours séquentiel

De manière évidente,

1. en général on parcourt tout le tableau,
2. chaque étape est "identique" aux autres,

Le **coût** d'un algorithme correspond au nombre d'opérations à effectuer.

Il est très difficile de le calculer exactement, beaucoup plus facile de l'estimer.

Dans notre cas, on a, grosso modo, autant d'opérations qu'il y a d'éléments dans le tableau.

Le coût est d'un parcours séquentiel est *proportionnel à la taille du tableau*. On dit qu'il est *linéaire*.

On note : **le parcours séquentiel est un algorithme en** $O(n)$

n désigne la taille du tableau.

Un tableau contient-il une clé ?

```
def recherche_sequentielle(tableau, x):
  for elt in tableau:
    if x == e:
      return True

  # on n'arrive ici que si l'élément n'est pas dans le tableau
  return False
```

`x` figure-t-il dans le tableau `T` ?

- **Sans information supplémentaire**, le parcours séquentiel est la seule manière de répondre.
- **Si on sait que le tableau est trié** alors il existe un algorithme beaucoup plus rapide : **la recherche dichotomique**. On l'étudiera dans un chapitre ultérieur.

Quel est le maximum d'un tableau ?

Contexte : On cherche la valeur extrême d'un tableau de nombres `T`.

Le principe est simple.

- Si on n'a qu'un élément, `maximum` est cet élément.
- Sinon, on parcourt les éléments et à chaque fois qu'une valeur `e` est supérieure à `maximum`, on l'affecte à `maximum`.
- On renvoie `maximum`

Pour le tableau `T = [2, 5, 9, 7]` cela donne :

élément `e`	2	5	9	7
`maximum`	2	5	9	9

Le maximum vaut 9.

Algorithme

```
fonction maximum(tableau T, nombre x) ---> nombre:
  On affecte à max la valeur de l'élément d'indice 0 du tableau.
  Pour chaque élément elt du tableau:
    si elt > max:
      max = elt
  renvoyer max
```

En Python :

```
def maximum(tableau):
  m = tableau[0]
  for elt in tableau:
    if elt > m:
      m = elt
  return m
```

Opérations natives : `min` et `max`

```
>>> tableau = [4, 5, 2, -1, 3]
>>> max(tableau)
5
>>> min(tableau)
-1
```

Calculer la moyenne des éléments d'un tableau

Contexte : Calculer la moyenne d'un tableau de nombres

Le principe repose sur **le cumul d'une valeur**.

- On cumule un compteur initialisé à 0, il augmente de 1 à chaque étape.
- On cumule la somme des valeurs comme on le ferait à la main.

Pour le tableau `T = [2, 5, 9, 8]` cela donne :

élément `e`	2	5	9	8
`nb_elements`	1	2	3	4
`somme`	2	7	16	24

Le somme vaut 24 et il y a 4 éléments : la moyenne est `24/4 = 6`

Algorithme

```
fonction moyenne(tableau T, nombre x) ---> nombre:
  On affecte à Somme la valeur 0
  On affecte à Effectif la valeur 0
  Pour chaque élément e du tableau:
    Somme = Somme + e
    Effectif = Effectif + 1
  renvoyer Somme / Effectif
```

En Python :

```
def moyenne(tableau):
  somme = 0
  effectif = 0
  for elt in tableau:
    effectif += 1
    somme += elt
  return somme / effectif
```

En python, solution native

Version courte qui n'illustre pas le programme :

```
def moyenne2(tableau):
  return sum(tableau) / len(tableau)
```

Difficile de comprendre ce qui se passe !

Retourner un tableau

```
>>> T = [1, 2, 3]
>>> retourner(tableau)
>>> [3, 2, 1]
```

Comment faire ?

principe

- On crée un tableau vide `R` pour nos éléments retournés
- On parcourt le tableau `T`,
  - pour chaque élément rencontré, on *l'insère au début de R*
- On retourne `R`

python

```
def retourner(tableau):
  tab_retourne = []
  for element in tableau:
    tab_retourne.insert(0, element)
    # variante moins efficace :
    # tab_retourne = [element] + tableau_retourne
  return tab_retourne
```

- faîtes bien attention aux crochets `[element] + tableau_retourne`

  On *ajoute deux tableaux*: Python les met bout à bout !

Opération native en python

```
>>> tableau = [1, 2, 3]
>>> tableau.reverse()
>>> # ne retourne rien mais modifie le tableau initial
>>> tableau
[3, 2, 1]
```

Quel est l'indice du maximum d'un tableau ?

Algorithme

On dipose d'un tableau d'entiers non vide, quel est l'indice de son élément maximal ?

On cherche un indice, on va parcourir les indices.

- On initialise le maximum avec `tableau[0]`. Le tableau étant non vide, c'est toujours possible.

- On initialise la position du maximum à 0.
- Pour `indice` allant de 0 à la longueur - 1,
  - si `tableau[indice] > maximum`, alors `position = indice` et `maximum = tableau[indice]`
- Renvoyer `position`

En Python

```
def indice_du_maximum(tableau: list):
    indice_maxi = 0
    maxi = tableau[0]

    for i in range(len(tableau)):
        if tableau[i] > maxi:
            indice_maxi = i
            maxi = tableau[i]

    return indice_maxi
```

Cette fois on a besoin de `for i in range(len(tableau)):`

Tableaux à deux dimensions

On rencontre souvent des données qui sont présentées sous la forme d'un tableau à deux dimensions :

```
1234
5678
```

Ces tableaux sont appelés des **matrices**. Celle-ci a 2 lignes et 4 colonnes

Affecter une matrice à une variable

Chaque ligne de la matrice est dans un tableau.

```
mat = [[1, 2, 3, 4],
       [5, 6, 7, 8]]
```

Remarquons

- qu'on crée un tableau extérieur `mat = [ ... ]`
- que chaque élément de ce tabeau est **lui même un tableau** : `[1, 2, 3, 4]` etc.

On dit que c'est une structure *imbriquée*.

Cette notation est universelle.

Atteindre un élément.

```
mat = [[1, 2, 3, 4],
       [5, 6, 7, 8]]
```

Disons qu'on veut atteindre le nombre 3. Il est dans la première ligne, 3ème colonne :

```
>>> mat[0][2]
3
```

On utilise deux séries de `[]` pour les lignes puis pour les colonnes.

Parcourir une matrice : boucles imbriquées

Parcourir une structure imbriquée **demande forcement deux boucles imbriquées.**

```
for ligne in tableau:
  for element in ligne:
    faire_quelque_chose_avec(element)
```

BOUCLES IMBRIQUÉES = L'UNE DANS L'AUTRE.

Si vous devez retenir 1 phrase de cette partie c'est celle là.

coût des boucles imbriquées = multiplier les coûts

- Si j'ai 9 lignes et 7 colonnes... j'ai ... $9 \times 7 = 63$ éléments.

  Mon parcours va visiter chacun de ces 63 éléments.

  Le coût est proportionnel à ce produit.

- Donc : chaque fois qu'on imbrique une boucle, on multiplie le nombre d'étapes.

  Si j'ai 4 boucles **imbriquées** avec respectivement 3, 5, 10 et 4 éléments à visiter,

  je ferai $3 \times 5 \times 10 \times 4 = 600$ opérations.

erreurs fréquentes

- indentation

```
t = [[1, 2, 3],
     [4, 5, 6]]

for ligne in T:

for element in ligne:
  print(element)
```

  cette syntaxe est fausse...

erreurs fréquentes : pire

- même problème mais qui ne plante pas...

```
t = [[1, 2, 3],
     [4, 5, 6]]

for ligne in T:
  pass # on evite de planter !

for element in ligne:
  print(element)
```

  C'est encore pire ! Au moins la première fois on avait une erreur...

  Cette fois que se passe-t-il ?

  1. On visite bien chaque ligne ! Mais on ne fait rien : `pass` !
  2. La deuxième boucle ne travaille que sur la **dernière** ligne

  Que verra-t-on ?

```
4
5
6
```

  Seulement les éléments de la dernière ligne

IL FAUT IMBRIQUER LES BOUCLES. L'UNE DANS L'AUTRE !!!

IL FAUT IMBRIQUER LES BOUCLES. L'UNE DANS L'AUTRE !!!

```
for ligne in matrice:

      for element in ligne:

          faire_qqchose(element)
```

IL FAUT IMBRIQUER LES BOUCLES. L'UNE DANS L'AUTRE !!!

Parcourir une matrice pour afficher ses éléments

On utilise toujours deux boucles **imbriquées** pour parcourir une matrice

```
for ligne in mat:
  for cellule in ligne:
    print(cellule, end=', ')
```

Va nous afficher :

`1, 2, 3, 4, 5, 6, 7, 8,`

Remarque : `print(1, end=', ')`

Normalement à la fin d'un `print` python insère un retour à la ligne. Ici, on le remplace par les caractères `', '` et nos éléments s'affichent en ligne.

Calculer la somme des éléments "à la main"

```
mat = [[3, 9, 1, 2],
       [4, 5, 0, 1]]
```

Calculons la somme de cette matrice à la main :

ligne 0	élément `e`	3	9	1	2
	`effectif`	1	2	3	4
	`somme`	3	12	13	15

On ne réinitialise pas `effectif` et `somme` entre les lignes !

ligne 1	élément `e`	4	5	0	1
	`effectif`	5	6	7	8
	`somme`	19	24	24	25

En Python

```
somme = 0
for ligne in mat:
  for element in ligne:
    somme += element
```

La valeur finale de `somme` est `25`

D'autres représentations des matrices

il est parfois nécessaire de connaître la position d'une cellule dans la matrice.

Nous allons créer une image, enregistrée dans un tableau 2D, présentant un dégradé

Dans l'exemple suivant, non seulement nous parcourons le tableau à l'aide d'indices, mais en plus les éléments de la matrice sont atteints autrement.

Créer un dégradé

```
from PIL import Image
from IPython.display import display # pour Colab
img = Image.new('RGB', (255, 128))
# nouvelle image, 255 de large, 128 de haut
pixels = img.load() # on charge la matrice des pixels

for x in range(255):
  for y in range(128):
    # attention à la notation [x, y] !!!
    pixels[x, y] = (255 - x, 0, 0)

display(img) # afficher dans colab
# img.show() # afficher dans la console
```

dégradé obtenu

Figure 1: Degradé rouge -> noir

Variants et invariants

- Un *invariant de boucle* est un propriété qui est vraie avant et après chaque tour d'une boucle.
- On appelle *variant de boucle "tant que"* toute quantité qui décroit strictement à chaque tour de la boucle. Dans une boucle non bornée (`tant que`), un variant permet de prouver que la boucle se termine.

  Dans une boucle bornée (`for`), il n'est pas nécessaire de donner un variant, la boucle se termine toujours.

```
def indice_du_maximum(tab: list) -> int:
    """
    Renvoie l'indice de l'élément maximal de `tab`
    """
    indice_maxi = 0
    i = 0
    while i < len(tab):
        if tab[i] > tab[indice_maxi]:
            indice_maxi = i
        i = i + 1
    return indice_maxi
```

1. Cette fonction comporte une boucle `while`, il est utile de donner un variant. Le variant proposé est `i`.
   - À chaque tour de la boucle, on fait `i = i + 1`, donc `i` augment de 1.
   - Lorsque `i` atteind `len(tab)`, la boucle s'arrête.

   Donc la fonction termine bien.
2. L'invariant est plus difficile à trouver : `indice_maxi` est l'indice du plus grand élément parmi les `i` premiers.

- C'est vrai avant d'entrer dans la boucle.
- Si c'est vrai au *début* d'un tour... alors deux possibilités :
  - si `tab[i] > tab[indice_maxi]`, alors l'indice courant `i` est celui d'un élément plus grand que `tab[indice_maxi]`. Donc cet indice `i` est celui du plus grand élément parmi les `i` premiers.
  - Sinon... c'est que `tab[indice_maxi] >= tab[i]`, or `tab[indice_maxi]` est le plus grand élément des `i` premiers, c'est toujours le cas après ce tour.

Conclusion : lorsque la boucle se termine à `i = len(tab)`, `indice_maxi` est l'indice du plus grand de tous les éléments.

Exemples

Exponentiation rapide

Pour illustrer la notion de variant, nous allons utiliser un algorithme largement hors programme.

Tout ce qu'on veut, **c'est démontrer que cette fonction va toujours terminer.**

```
def puissance(a: int, n: int) -> int:
    """Renvoie le nombre a exposant n. """

    p = 1

    while n > 0:
        if n % 2 == 0:
            a = a * a
            n = n // 2
        else:
            p = p * a
            n = n - 1
    return p
```

1. Appliquer la fonction pour calculer `puissance(3, i)` pour `i` entre 0 et 4 inclus.
2. Démontrer que cette fonction termine toujours en explicitant un *variant*.

Somme des entiers

On considère l'algorithme suivant :

```
somme( tableau )

    s = 0
    pour chaque element x de tableau, faire
        s = s + x

    renvoyer s
```

Proposer un invariant de boucle

Division euclidienne par soustraction

```
diviser(a, b)
    r = a
    q = 0

    Tant que r >= b, faire
        r = r - b
        q = q + 1

renvoyer q, r
```

1. Faire tourner l'algorithme à la main pour `a = 7, b = 2`
2. Quelle relation vérifient `a, b, q, r` dans nos exemples ?

3. Pourquoi ne peut-on pas toujours s'arrêter après le premier tour ?
4. Quelle propriété supplémentaire doit vérifier `r` ?
5. Proposer un variant et un invariant de boucle.

Exemples en python

variant_invariant.py