CM 3 : Tableaux, Complexité

Info1.Algo1

2022-2023 Semestre Impair

Plan

1. Tableaux et matrices
   - Types et abstraction
   - Tableaux
   - Exercice
   - Matrice

2. Complexité
   - Notion de complexité
   - Application : Recherche d'une valeur dans un tableau

Rappel sur les types

À toute variable est associée un **type** qui permet au traducteur du langage (compilateur ou interpréteur) de lui affecter la **place nécessaire en mémoire**.

Définition

Un **type** définit :

- Un **espace de valeurs**.
- Un **ensemble d'opérations autorisées**.

Remarque : En Python, il n'y a pas de déclaration explicite, elle découle de l'affectation d'une valeur et le type de la valeur détermine le type de la variable.

Rappel sur les types

Exemple: le type `int` Python

- Espace de valeurs : $\mathbb{N}$.
- Opérations autorisées : `+`, `-` , `%` , `-=` , `+=` , etc.

Exemple: le type `bool` Python

- Espace de valeurs : $\{$`False`, `True`$\}$.
- Opérations autorisées : `not`, `and` , `or` , `==`, etc.

Abstraction de données

Définition

De manière plus générale on appelle **abstraction de données** la définition d'un type de données avec la description des **opérations qu'il est possible de lui appliquer**, indépendamment de la manière dont cela sera programmé.

La **description de chaque opération** précise:

- Ses entrées.
- Les préconditions qui doivent être vérifiées pour que l'opération s'exécute correctement.
- Ses sorties.
- Les post conditions vraies après exécution de l'opération.

Structure de données

Définition

Une **structure de données** est décrite en présentant :

- L'**abstraction de données** qui la définit.
- Son **implémentation**, c'est-à-dire sa mise en oeuvre matérielle.

Structure de données

Exemple

Structure de données **Matrice** :

- L'utilisateur de ce type de structure n'est concerné que par les opérations qu'il peut effectuer avec, ici « comment accéder à l'élément (i, j) de la matrice ». Il doit donc connaître l'**abstraction**
- Le programmeur de la structure de données doit choisir entre les différentes **implémentations possibles** de cette abstraction : tableau bidimensionnel, tableau de lignes ou liste de ses éléments non nuls (matrice creuse)...

L'abstraction de données **Tableau**

Principes généraux

Un **tableau** permet de mémoriser une séquence d'éléments :

- Le **nombre d'éléments** du tableau est **fixe**.
- Tous les éléments du tableau sont de **même type**.
- Chaque élément est repéré par sa position dans le tableau, son indice.

Remarque : d'autres façons de représenter des séquences de données (listes chaînées, piles, files. . .) seront vues ultérieurement.

L'abstraction de données **Tableau**

Opération possibles

- La **création** d'un tableau précise :
  - Le **nombre d'éléments** qu'il contient.
  - Le **type** des éléments du tableau.
- L'**accès à un élément** en **lecture et écriture** précise l'**indice** dans le tableau.

Syntaxe usuelle

`tableau[i]` : élément d'indice `i` dans `tableau`

Remarque : La validité de l'indice `i` dépend de la dimension `DIM` du tableau et du langage :

- Python, C, Java, ... : `0<=i<DIM`
- Matlab, ... : `1<=i<=DIM`

Implémentation usuelle

Représentation en mémoire

Dans une grande partie des langages de programmation, les tableaux sont représentés en mémoire sous la forme de **cellules contiguës** :

Adresse mémoire	Donnée	
0xFA70	01001000	← *Début du tableau (indice 0)*
0xFA71	11100111	*(indice 1)*
0xFA72	10000111	*(indice 2)*
0xFA73	00101111	...

- Adresse mémoire de l'élément `tableau[i]` :
  adresse de `tableau[0]` + i × taille d'une cellule.
- Le **temps d'accès** à l'élément d'indice `i` (lecture ou modification) est donc constant (indépendant de la valeur de `i` et de la dimension du tableau.

Implémentation usuelle

Les opérations d'insertion ou de suppression d'éléments dans cette implémentation impliqueraient alors de déplacer d'autres éléments :

Adresse mémoire	Donnée	
0xFA70	01001000	← *Début du tableau*
~~0xFA71~~	~~11100111~~	← *Élément à supprimer*
0xFA72	10000111	↑ *Éléments à déplacer*
0xFA73	00101111	↑ ...

Attention

La **suppression et l'insertion** ne sont donc pas autorisées pour le type abstrait **Tableau**.

Utilisation en Python natif

Utilisation en Python natif

Utilisation du type `list` natif de Python, en s'imposant les règles suivantes :

- La **taille** et le **type** des éléments sont fixés à la première affectation.
- Toutes les opérations modifiant la taille de l'objet de type `list` (append, insert, `del`...) sont interdites.
- On ne mélange pas les types à l'intérieur de l'objet de type `list`.
- Les seules opérations autorisées sont la lecture et l'écriture à un indice donné.

Exemple :

```
# Creation d’un tableau d’entiers de taille 20 :
tableau = [0]*20
# Ecriture dans le tableau :
tableau[0] = int(input())
# Lecture et ecriture
tableau[1] = 2*tableau[0]
```

Utilisation en C

Utilisation en C

Un programme respectant l'abstraction tableau se traduira naturellement en langage C. Ainsi l'exemple précédent donnera :

```c
int main () {
  // Tableau d'entiers de taille 20 :
  int tableau[20] = {0}; // Valable uniquement pour 0
  // Ecriture dans le tableau :
  scanf("%d",&tableau[0]);
  // Lecture et ecriture
  tableau[1] = 2*tableau[0];
  return(0);
}
```

Exercice

On considère la fonction suivante :

```
def deplacer(tableau,i_source,i_destination):
  valeur = tableau.pop(i_source)
  tableau.insert(i_destination,valeur)
```

1. Décrire l'opération réalisée par cette fonction.
2. Écrire sous forme d'assertion une pré-condition pertitente.
3. Pour quelles raisons cette fonction ne respecte pas l'abstraction tableau ?
4. Corriger le code de la fonction afin qu'elle respecte l'abstraction tableau.
5. Quelle contrainte peut-on imposer sur ce code afin de garantir que les éléments du tableaux sont conservés? Corriger éventuellement le code en conséquence.

L'abstraction de données **Matrice**

Principes généraux

Une matrice permet de mémoriser des éléments selon deux dimensions :

- Le **nombre de lignes** et de **colonnes** est `fixe`.
- Tous les éléments de la matrice sont de **même type**.
- Chaque élément est repéré par sa position dans la matrice, constituée par le couple (numéro de ligne, numéro de colonne).

L'abstraction de données **Matrice**

Opération possibles

- La **création** d'une matrice précise :
    - Son **nombre de lignes et de colonnes**.
    - Le **type** des éléments de la matrice.
- L'**accès à un élément** en **lecture et écriture** précise les **indices de ligne et de colonne**.

L'abstraction de données **Matrice**

Syntaxe habituelle :

matrice[i][j] est l'élément de la matrice situé à la ligne `i` et à la colonne `j`.

Remarque 1 : La validité des indices `i` et `j` dépend du langage :

- Généralement : `0<=i<NB_LIGNES` et `0<=j<NB_COLONNES`
- Autre convention : `1<=i<=NB_LIGNES` et `1<=j<=NB_COLONNES`

Remarque 2 :

- `matrice[i]` est un tableau de dimension `NB_COLONNES`.
- Certains langages autorisent aussi l'écriture `matrice[i,j]`.

Utilisation en Python natif

Utilisation en Python natif

Utilisation du type `list` natif de Python, sous forme d'une **liste de listes**. Comme pour l'abstraction tableau, les règles suivantes doivent être respectées :

- Le **nombre de lignes, de colonnes** et le **type** des éléments sont fixés à la première affectation.
- Toutes les opérations de modification de taille (`append`, `insert`, `del`...) sont interdites.
- Aucune modification de type n'est autorisée.
- Les seules opérations autorisées sont la lecture et l'écriture à une position donnée.

Exemple :

```python
# Creation d'une matrice d'entiers de 19 lignes et 13 colonnes :
matrice = [None]*19
for i in range(len(matrice)):
  matrice[i] = [0]*13
# Ecriture dans la matrice :
matrice[3][4] = int(input())
# Lecture et ecriture
matrice[2][7] = 2*matrice[3][4]
```

Utilisation en C

Utilisation en C

Un programme respectant l'abstraction matrice se traduira naturellement en langage C. Ainsi l'exemple précédent donnera :

```c
int main(void) {
  // Matrice d'entiers de 19 lignes et 13 colonnes :
  int matrice[19][13] = {0}; // valable uniquement pour 0
  // Ecriture dans le tableau :
  scanf("%d",&matrice[3][4]);
  // Lecture et ecriture
  matrice[2][7] = 2*matrice[3][4];
  return(0);
}
```

Plan

1. Tableaux et matrices
   - Types et abstraction
   - Tableaux
   - Exercice
   - Matrice
2. Complexité
   - Notion de complexité
   - Application : Recherche d'une valeur dans un tableau

Définition

Définition

On appelle complexité d'un algorithme la quantité de **ressources** nécessaires pour exécuter cet algorithme, exprimée **en fonction de la taille** de son(ses) entrée(s).

- Les **ressources** utilisées peuvent être de deux types :
    - Le **temps** (il s'exprime généralement en nombre d'opérations : additions, multiplications, comparaisons...).
    - La **mémoire**.
- La **taille des entrées** s'exprime selon le problème posé :
    - La longueur d'une liste, d'un tableau
    - La valeur d'une entrée

Exemple 1

La fonction **somme_entiers_1** définie ci-dessous accepte en entrée un entier **n** positif et retourne la somme des entiers de **1** à **n** :

```
def somme_entiers_1(n):
  somme = 0
  for i in range(1,n+1):
    somme = somme+i
  return somme
```

1. Exprimer les ressources utilisées par cette la fonction **somme_entiers_1** en fonction de la taille de l'entrée.
2. Donner une expression simplifiée de $\sum_{i=1}^{n} i = 1 + 2 + ... + n$.
3. Écrire la fonction **somme_entiers_2** retournant la même valeur que **somme_entiers_1** mais telle que sa complexité en temps ne dépende pas de **n**.

Mesure expérimentale

Méthode

- Afin de **mesurer expérimentalement** la complexité en temps d'un algorithme, on peut utiliser la fonction **time.time()** de la librairie python **time** qui retourne le temps (en secondes) écoulé depuis le 1er Janvier 1970 à 00h00m00s (date initiale aussi appelée epoch).
- On mesure alors la **différence entre les temps mesurés avant et après** l'exécution de l'algorithme.

Exemple 1 (suite)

Le programme suivant permet de mesurer le temps d’exécution de la fonction **somme_entiers_1** pour différentes valeurs de **n** :

```
import time
def somme_entiers_1(n):
  ...

for k in range(5,11):
  n = 10**k
  tic = time.time()
  somme_entiers_1(n)
  tac = time.time()
  print(’n=10^’,k,’ : ’,tac-tic,’ sec’,sep=’’)
```

Exemple 1 (suite)

Un exemple d'affichage peut-être alors le suivant (dépend de la machine utilisée):

```
n=10^5 : 0.005068063735961914 sec
n=10^6 : 0.04670357704162598 sec
n=10^7 : 0.42836833000183105 sec
n=10^8 : 4.371985912322998 sec
n=10^9 : 42.5275456905365 sec
n=10^10 : 457.0766546726227 sec
```

4. Comment évolue le temps d'exécution de l'appel **somme_entiers_1(n)** lorsque le nombre **n** passé en paramètre est multiplié par **10** ? **Interpréter.**
5. Estimer (en années) le temps nécessaire à l'exécution de l'appel **somme_entiers_1(10**50)**.

Remarque : Sur la même machine le temps d'exécution de l'appel **somme_entiers_2(10**50)** donne :

```
n=10^50 : 1.9073486328125e-06 sec
```

Problèmes et algorithmes

Définitions

- Un **problème** est caractérisé par la **définition de ses entrées** (pré-condition) **et de ses sorties** (post-condition).
- On appelle **algorithme** un ensemble d'opérations permettant de **résoudre un problème**.

Exemple 1 (suite)

Dans l'exemple précédent :

- L'entrée **du problème** est un entier positif **n**, sa sortie est la somme **1+2+...+n**.
- Les **algorithmes** mis en œuvre dans les fonctions **somme_entiers_1** et **somme_entiers_2** permettent d'y répondre.

Motivation de ce chapitre

- Plusieurs algorithmes peuvent résoudre un même problème.
- Ce n'est pas parce qu'un algorithme répond à un problème qu'il est le plus adapté.
- L'étude de la **complexité** est **un des** critères qui permet d'évaluer, pour un même problème, différents algorithmes et de choisir le plus adapté. Leur **capacité d'adaptation au changement** peut par exemple être un critère plus impactant.

Recherche d'une valeur dans un tableau

Problème posé

On considère un tableau d'entiers **tab** et un entier **valeur** qui est un élément ou non de **tab**. On souhaite déterminer l'indice de la **dernière occurrence** de **valeur** dans le tableau **tab** si celle-ci existe ou **-1** si **valeur** n'est pas présente dans **tab**.

Recherche d'une valeur dans un tableau

Recherche exhaustive

Si tab n'est pas trié, on est forcé de consulter tous ses éléments. Tout élément non consulté peut en effet être égal à **valeur**. Cette recherche exhaustive peut se faire grâce à la fonction définie ci-dessous :

```python
def recherche_exhaustive(tab,valeur):
  i = len(tab)-1
  while i>=0 and tab[i]!=valeur:
    i -= 1
  return i
```

Recherche d'une valeur dans un tableau

Recherche par dichotomie

Si tab est trié par ordre croissant, il est possible de tirer partie de cet ordre pour rendre la recherche beaucoup plus rapide :

Modèle de solution

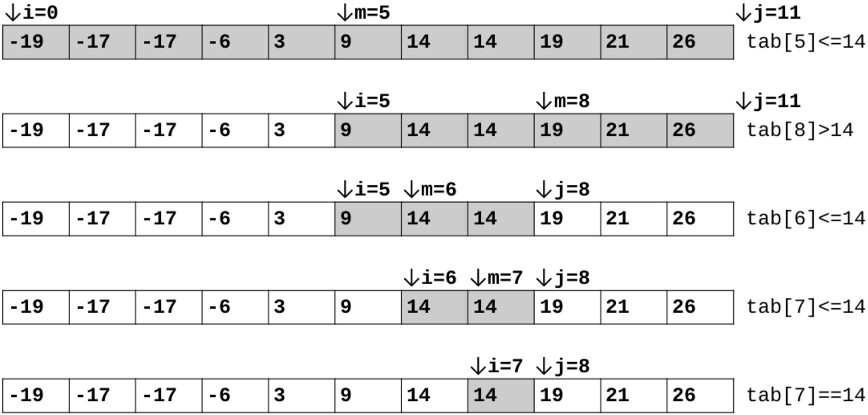
- On consulte l'élément situé au milieu du tableau. Si cet élément est inférieur à la valeur recherchée, la recherche n'a besoin d'être poursuivie que dans la partie droite du tableau, sinon on poursuit la recherche dans la partie gauche.
- À chacune des étapes suivantes on coupe en deux cet intervalle de recherche et selon la valeur de l'élément situé au milieu, on choisit l'intervalle de gauche ou de droite pour poursuivre l'algorithme, jusqu'à ce que cet intervalle soit de longueur minimale.
- Ce principe de division en deux parties donne son nom à la

Notations

- La **zone de recherche** est la sous-liste **tab[i:j]** (où l'indice **i** est inclus et l'indice **j** exclu).
- À l'**initialisation** de l'algorithme **i** et **j** valent respectivement **0** et **len(tab)**.
- L'**algorithme se termine** lorsque la longueur de la zone de recherche est **1**.
- L'**indice du milieu** est noté **m** et est obtenu par l'expression **(i+j)//2**,
- La **valeur du tableau** correspondante est **tab[m]**, qui doit être comparée à **valeur**.

Exemple

Si **tab=[-19,-17,-17,-6,3,9,14,14,19,21,26]** et **valeur=14**, on obtient le déroulé suivant (la zone de recherche est grisée) :



↓i=0					↓m=5						↓j=11
-19	-17	-17	-6	3	9	14	14	19	21	26	tab[5]<=14

					↓i=5			↓m=8			↓j=11
-19	-17	-17	-6	3	9	14	14	19	21	26	tab[8]>14

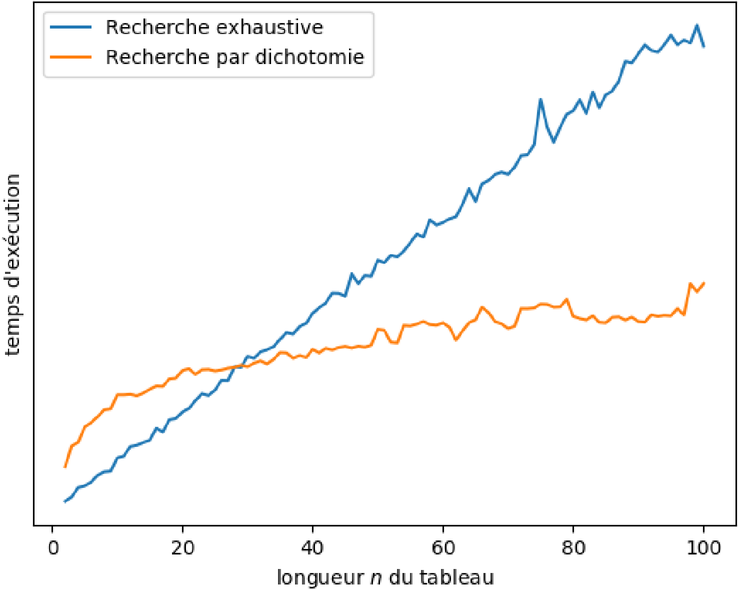
					↓i=5	↓m=6		↓j=8			
-19	-17	-17	-6	3	9	14	14	19	21	26	tab[6]<=14

						↓i=6	↓m=7	↓j=8			
-19	-17	-17	-6	3	9	14	14	19	21	26	tab[7]<=14

							↓i=7	↓j=8			
-19	-17	-17	-6	3	9	14	14	19	21	26	tab[7]==14

1. Écrire la fonction **recherche_par_dichotomie** qui accepte en paramètres un tableau non vide d'entiers **tab** trié par ordre croissant ainsi qu'un entier **valeur** (tel que **tab[0]<=valeur**) et retourne l'indice de la dernière occurrence de **valeur** dans le tableau **tab** si celle-ci existe ou **-1** si valeur n'est pas présente dans **tab** . L'algorithme utilisé est une recherche par dichotomie.
2. Pour le cas où le tableau **tab** est vide ou si **valeur<tab[0]** on souhaite que la fonction retourne **-1**. Traiter ces deux cas en début de fonction de façon que les conditions imposées par le cas général soient respectées dans la suite de l'algorithme.

3. Déterminer les valeurs successives de **i**, **j** et **m** ainsi que l'indice retourné dans les cas suivants :
    - **tab=[] valeur=13**
    - **tab=[5,7,9,10,11] valeur=13**
    - **tab=[6,9,10,14,17,25,26,26] valeur=13**
4. Exprimer, en fonction de la longueur **n** du tableau **tab** le nombre d'itérations dans le pire des cas? Comparer avec la fonction **recherche_exhaustive**.


Recherche exhaustive
Recherche par dichotomie
temps d'exécution
0
20
40
60
80
100
longueur *n* du tableau