CM 3 : Tableaux, Complexité

Info1.Algo1

2025-2026 Automne

Plan

Spécifier un type

Objectif de cette partie

Spécifier deux types : **tableau** et **matrice**

Rappel : À toute variable est associée un **type** qui permet à l'interpréteur du langage de la **représenter** en mémoire et de la **modifier** selon les opérations réalisées.

Définition

Spécifier un type consiste donc à :

- Définir un **espace de valeurs**.
- Définir un **ensemble d'opérations autorisées**.
  Chaque opération doit alors avoir sa propre spécification.

Spécifier un type

Exemple: le type `int` Python

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

Exemple: le type `bool` Python

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

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.

Spécifier un type

Une opération peut être vue comme une fonction, où :

- les **paramètres** sont les **opérandes**.
- la **valeur de retour** est le **résultat**.

Exemple : `a+b` peut être vu comme `plus(a,b)`

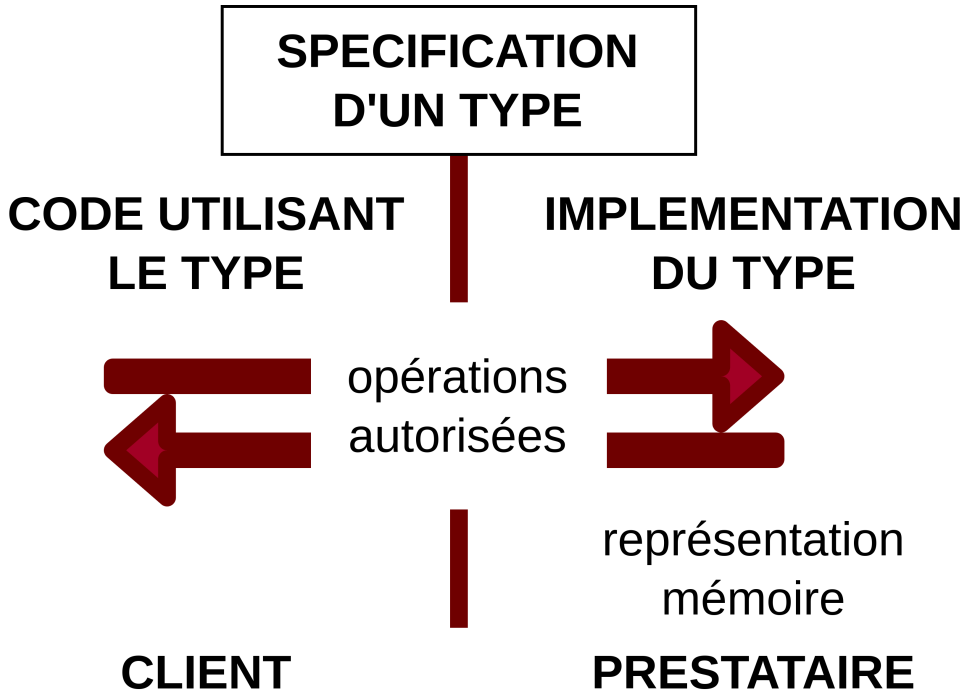
Spécification des opérations

Comme pour une fonction, la **spécification de chaque opération** précise :

- Le type de ses opérandes et de son résultat.
- Sa pré-condition et sa post-condition.

Spécification difficile : ne sera pas vue dans cette UE.
On se limitera à une spécification informelle.

Spécifier un type



Spécifier un type

De façon générale :

Vocabulaire

- On appelle **type abstrait** la définition d'un type de données avec la spécification des **opérations autorisées**, indépendamment de son implémentation.
- Une **structure de données** définit aussi son **implémentation**, c'est-à-dire sa mise en oeuvre matérielle.

Motivation

Objectif général

On souhaite dans ce chapitre spécifier un **type abstrait** permettant de manipuler des **séquences** d'éléments.

Question

Pourquoi ne pas utiliser tel quel le type `list` natif de Python ?

- On souhaite pouvoir **mettre en oeuvre les algorithmes** étudiés dans cette UE **indépendamment du langage de programmation utilisé** *(assembleur, C, Java, JavaScript, ...)*.
- Certaines fonctionnalités des listes Python ne sont pas disponibles de façon immédiate dans les autres langages.

Rappel : l'objectif de l'UE **Info1.Algo1** est d'**acquérir des notions d'algorithmique**, pas de faire un cours avancé de Python. Connaître **le langage Python n'est pas un but mais un moyen.**

Motivation

Précision de l'objectif

On cherche un **ensemble minimal d'opérations** vérifiant les conditions suivantes :

- Il permet d'effectuer des **traitement sur des séquences** d'éléments.
- Il est disponible sur une **majorité de langages** de programmation.

Le type abstrait **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.

Le type abstrait **Tableau**

Opérations autorisées

- **Créer** un tableau en précisant :
  - Le **nombre d'éléments** qu'il contient.
  - Le **type** des éléments du tableau.
- **Déterminer la longueur** d'un tableau.
- **Lire** un élément à un indice valide.
- **Écrire** un élément à un indice valide.

Le type abstrait **Tableau**

Spécification de l'interface

- **Créer** un tableau : on utilise une fonction de création précisant la longueur et la valeur par défaut des éléments du tableau.
  `tableau = creer_tableau(longueur,defaut)`
- **Déterminer la longueur** d'un tableau : `len(tableau)`
- **Lire** un élément à un indice `i` valide : `valeur = tableau[i]`
- **Écrire** un élément à un indice `i` valide : `tableau[i] = valeur`

Les **indices** `i` **valides** sont ceux qui vérifient :
`0<=i<len(tableau)`

Implémentations

On choisit dans ce cours d'utiliser **de façon restreinte** le **type** `list` **natif** de Python.

- La fonction de création donnée dans certains exercices est donc :

```
1 def creer_tableau(longueur,defaut):
2     return [defaut]*longueur
```

- Si celle-ci n'est pas donnée, on s'autorise à écrire directement :

```
1 tableau = [defaut]*longueur
```

Comme la **longueur** et le **type** des éléments sont fixés à la création :

- Toutes les méthodes modifiant la longueur de l'objet de type `list` (`append`, `insert`, `pop`...) sont donc interdites.
- On ne mélange pas les types à l'intérieur de l'objet de type `list`.

Implémentations

Exemple :

```
1 def creer_tableau(longueur,defaut):
2     return [defaut]*longueur
3
4 # Creation d'un tableau d'entiers de longueur 20 :
5 tableau = creer_tableau(20,0)
6 # Ecriture dans le tableau :
7 tableau[0] = int(input())
8 # Lecture et ecriture
9 tableau[1] = 2*tableau[0]
```

Implémentations

Implémentation en C

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

```c
int main () {
    // Tableau d'entiers de longueur 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);
}
```

Implémentations

Remarque : pourquoi choisir cet ensemble d'opérations ?
Cet ensemble d'opérations permet d'implémenter une séquence d'éléments en utilisant des **cellules contiguës** de mémoire :

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

- Si les éléments sont de **même type**, les cellules peuvent avoir la **même taille**.
- La création d'un tableau revient à demander **une seule fois l'allocation** d'une zone de taille pré-définie de la mémoire.
- L'accès en écriture et en lecture revient à **calculer l'addresse en mémoire** de l'élément :
  adresse de `tableau[0]` + i × longueur d'une cellule.

Exercice

On considère la fonction suivante :

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

1) Décrire l'opération réalisée par cette fonction.

2) Écrire sous forme d'assertion une pré-condition pertinente.

3) Pour quelles raisons cette fonction ne respecte pas le type abstrait tableau ?

4) Corriger le code de la fonction afin qu'elle respecte le type abstrait 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.

Le type abstrait **Matrice**

Principes généraux

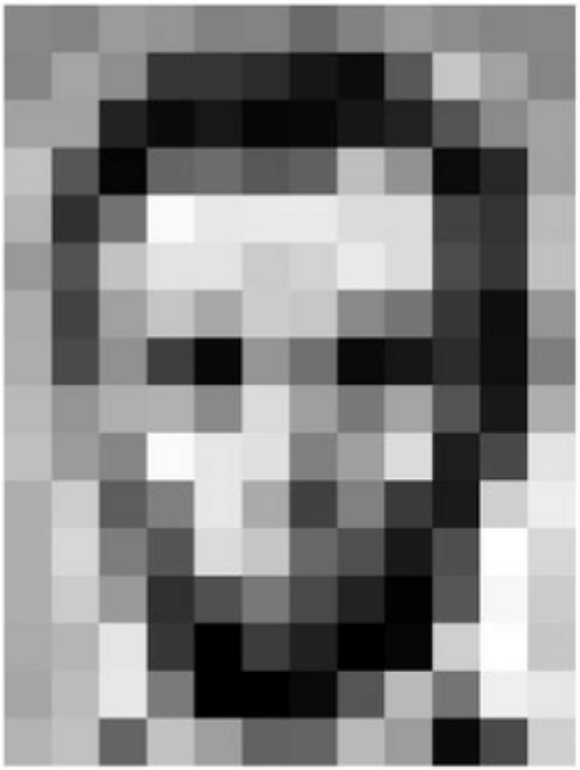
Une **matrice** permet de mémoriser des éléments selon deux dimensions :

- Le **nombre de lignes** et le **nombre de colonnes** sont **fixes** et **non nuls**.
- Tous les éléments de la matrice sont de **même type**.
- Chaque élément est repéré par sa position dans la matrice. Cette position est caractérisée par un **couple d'indices** (indice de ligne, indice de colonne).

Le type abstrait Matrice

Motivation : représenter des données en 2D

- cartes, images, écran, ...



157	153	174	168	150	152	129	151	172	161	155	156
155	182	163	74	75	62	33	17	110	210	180	154
180	180	50	14	34	6	10	33	48	106	159	181
206	109	5	124	131	111	120	204	166	15	56	180
194	68	137	251	237	239	239	228	227	87	71	201
172	105	207	233	233	214	220	239	228	98	74	206
188	88	179	209	185	215	211	158	139	75	20	169
189	97	165	84	10	168	134	11	31	62	22	148
199	168	191	193	158	227	178	143	182	106	36	190
205	174	155	252	236	231	149	178	228	43	95	234
190	216	116	149	236	187	86	150	79	38	218	241
190	224	147	108	227	210	127	102	36	101	255	224
190	214	173	66	103	143	96	50	2	109	249	215
187	196	235	75	1	81	47	0	6	217	255	211
183	202	237	145	0	0	12	108	200	138	243	236
195	206	123	207	177	121	123	200	175	13	96	218

157	153	174	168	150	152	129	151	172	161	155	156
155	182	163	74	75	62	33	17	110	210	180	154
180	180	50	14	34	6	10	33	48	106	159	181
206	109	5	124	131	111	120	204	166	15	56	180
194	68	137	251	237	239	239	228	227	87	71	201
172	105	207	233	233	214	220	239	228	98	74	206
188	88	179	209	185	215	211	158	139	75	20	169
189	97	165	84	10	168	134	11	31	62	22	148
199	168	191	193	158	227	178	143	182	106	36	190
205	174	155	252	236	231	149	178	228	43	95	234
190	216	116	149	236	187	86	150	79	38	218	241
190	224	147	108	227	210	127	102	36	101	255	224
190	214	173	66	103	143	96	50	2	109	249	215
187	196	235	75	1	81	47	0	6	217	255	211
183	202	237	145	0	0	12	108	200	138	243	236
195	206	123	207	177	121	123	200	175	13	96	218

Le type abstrait Matrice

Motivation : notion mathématique

Recouvre la **notion mathématique de matrice** qui permet entre autre de représenter des transformations du plan ou de l'espace...

Exemple : rotations en 3D

```
rot = [
    [ 0, 1, 0],
    [-1, 0, 0],
    [ 0, 0, 1]
]
```

$$R = \begin{pmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Le type abstrait **Matrice**

Opérations autorisées

- **Créer** une matrice en précisant :
  - Son **nombre de lignes et de colonnes**.
  - Le **type** des éléments de la matrice.
- **Déterminer les dimensions** d'une matrice.
- **Lire** un élément à des indices valides.
- **Écrire** un élément à des indices valides.

Le type abstrait **Matrice**

Spécification de l'interface

- **Créer** une matrice : on utilise une fonction de création précisant les dimensions et la valeur par défaut :

```
matrice = creer_matrice(nb_lignes,nb_colonnes,defaut)
```

- **Déterminer les dimensions** d'une matrice :

```
nb_lignes = len(matrice)
nb_colonnes = len(matrice[0])
```

- **Lire** un élément à un indice de ligne `i` et un indice de colonne `j` valides : `valeur = matrice[i][j]`
- **Écrire** un élément à des indices `i` et `j` valides :

```
matrice[i][j] = valeur
```

Les **indices `i` et `j` valides** sont ceux qui vérifient :

```
0<=i<len(matrice)and 0<=j<len(matrice[0])
```

Implémentations

On choisit dans ce cours d'utiliser le **type `list` natif** de Python sous forme d'une **liste non vide de listes non vides**.

- La fonction de création donnée dans certains exercices est donc :

```
1 def creer_matrice(nb_lignes,nb_colonnes,defaut):
2     matrice = [None]*nb_lignes
3     for i in range(nb_lignes):
4         matrice[i] = [defaut]*nb_colonnes
5     return matrice
```

- Si celle-ci n'est pas donnée, on s'autorise à écrire directement :

```
1 matrice = [None]*nb_lignes
2 for i in range(nb_lignes):
3     matrice[i] = [defaut]*nb_colonnes
```

Implémentations

Comme pour le type abstrait tableau, le **nombre de lignes, de colonnes** et le **type** des éléments sont fixés à la la création :

- Toutes les méthodes modifiant la longueur des objets de type `list` (`append`, `insert`, `pop`...) sont donc interdites.
- On ne mélange pas les types des éléments.

Implémentations

Exemple :

```
1  def creer_matrice(nb_lignes,nb_colonnes,defaut):
2      matrice = [None]*nb_lignes
3      for i in range(nb_lignes):
4          matrice[i] = [defaut]*nb_colonnes
5      return matrice
6
7  # Creation matrice d'entiers de 19 lignes et 13 colonnes :
8  matrice = creer_matrice(19,13,0)
9  # Ecriture dans la matrice :
10 matrice[3][4] = int(input())
11 # Lecture et ecriture
12 matrice[2][7] = 2*matrice[3][4]
```

Implémentations

Implémentation en C

Un programme respectant le type abstrait 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

Notion de complexité

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

Notion de complexité

Exemple 1

La fonction `somme_entiers_1` définie ci-contre accepte en paramètre un entier `n` positif et retourne la somme des entiers de `1` à `n` :

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

1) Exprimer les ressources utilisées par cette la fonction `somme_entiers_1` en fonction de la taille du paramètre.

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`.

Observations expérimentales

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.

Observations expérimentales

Exemple 1 (suite)

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

```
1  import time
2
3  def somme_entiers_1(n):
4      ...
5
6  for k in range(5,11):
7      n = 10**k
8      tic = time.time()
9      somme_entiers_1(n)
10     tac = time.time()
11     print('n=10^',k,' : ',tac-tic,' sec',sep='')
```

Observations expérimentales

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 : 0.0467035770416259 8 sec
n=10^7 : 0.4283683300018310 5 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 `somme_entiers_1(n)` lorsque le paramètre `n` est multiplié par `10` ? **Interpréter.**

5) Estimer (en années) le temps nécessaire à l'exécution de l'appel `somme_entiers_1(10**50)`.

Observations expérimentales

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
```

Motivation

Rappels

- Un **problème** est caractérisé par sa **spécification** (pré-condition et post-condition).
- Une **implémentation** est une fonction qui respecte cette spécification.
- Un **algorithme** est l'ensemble des opérations permettant de respecter cette spécification, indépendamment du langage de programmation utilisé.

Exemple 1 (suite)

L'entrée du **problème** est un entier positif `n`, et sa sortie est la somme `1+2+...+n`.

Les **algorithmes** mis en œuvre dans les fonctions `somme_entiers_1` et `somme_entiers_2` permettent de l'implémenter.

Motivation

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.

Application : Racine carrée entière

Problème posé

On considère un entier positif `n` et l'on souhaite déterminer la **partie entière de sa racine carrée**.

Spécification

- **Type du paramètre :** `n` est un entier.
- **Type de la valeur de retour :** `a` est un entier.
- **Pré-condition :** `n>=0`
- **Post-condition :** `a>=0 and a**2<=n<(a+1)**2`

Remarque : même si la condition `a>=0` ne semble pas absolument nécessaire, elle sera utile lors du chapitre sur l'invariant pour s'assurer que le carré `a**2` définit `a` de manière univoque

Application : Racine carrée entière

Algorithme 1 : par balayage

On cherche la racine carrée en partant de l'entier `0` et en incrémentant jusqu'à obtenir la solution.

```
1 def racine_entiere_1(n):
2     assert n>=0, 'Pre-condition'
3     a = 0
4     while (a+1)**2<=n:
5         a += 1
6     assert a>=0 and a**2<=n<(a+1)**2, 'Post-condition'
7     return a
```

Application : Racine carrée entière

Algorithme 2 : par dichotomie

- On établit une **zone de recherche** comprise entre deux entiers.
- On calcule le carré du **milieu** de cette zone. Si ce carré est inférieur à `n`, la recherche n'a besoin d'être poursuivie que dans la moitié droite de la zone de recherche, sinon on poursuit la recherche dans la moitié gauche.
- À chacune des **étapes suivantes** on coupe en deux l'intervalle de recherche et selon le carré 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 méthode de recherche par **dichotomie**.

Application : Racine carrée entière

Notations

- La **zone de recherche** est comprise entre les entiers `a` (inclus) et `b` (exclu).
- À l'**initialisation** de l'algorithme `a` et `b` valent respectivement `0` et `n+1`.
- L'**algorithme se termine** lorsque la longueur de la zone de recherche est `1`.
- Le **milieu** est noté `m` et est obtenu par l'expression `(a+b)//2`,
- C'est le **carré du milieu** `m**2` qui doit être comparé à `n`.

Application : Racine carrée entière

Exemple

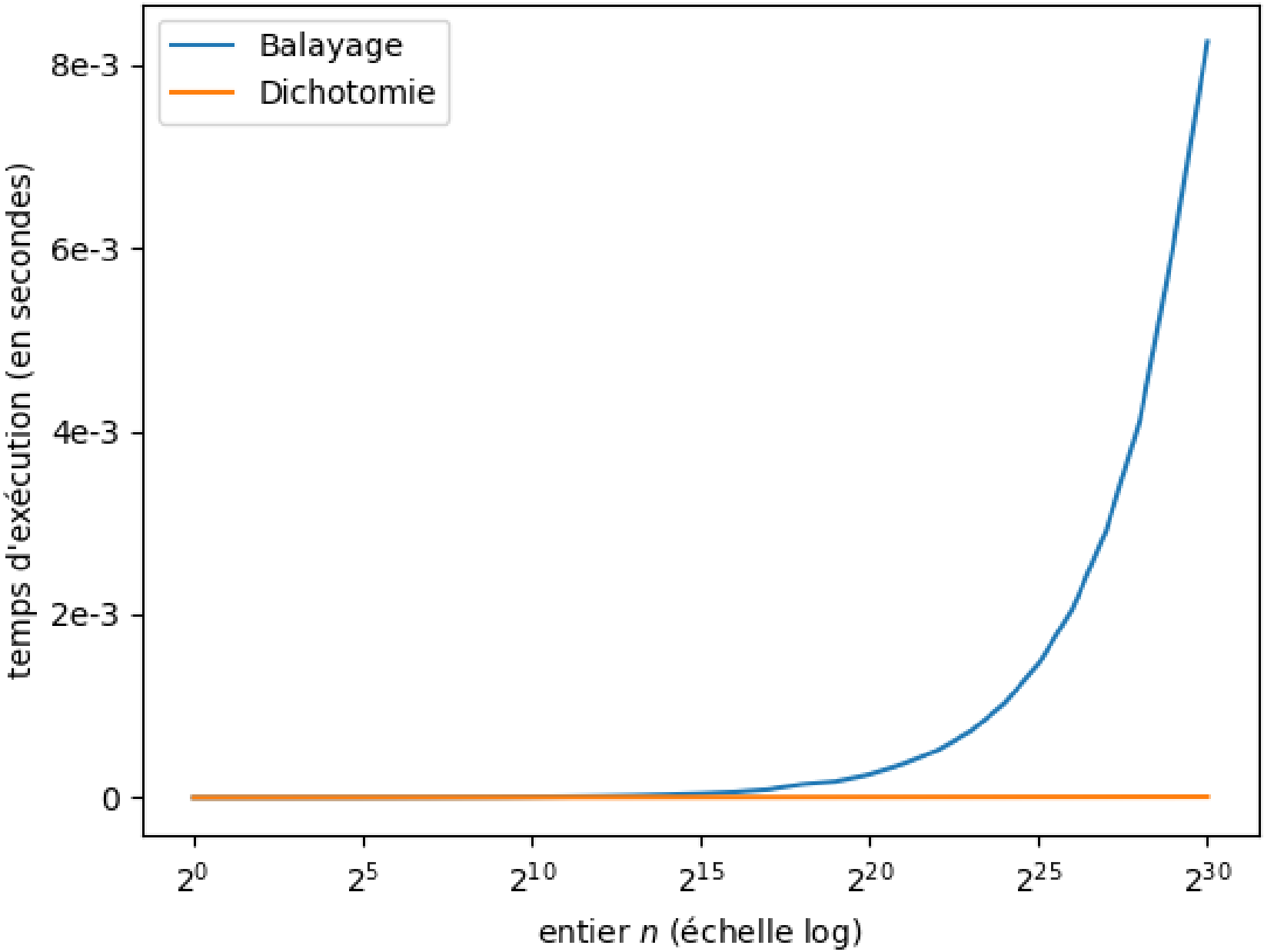
Si `n=27`, on obtient le déroulé suivant :

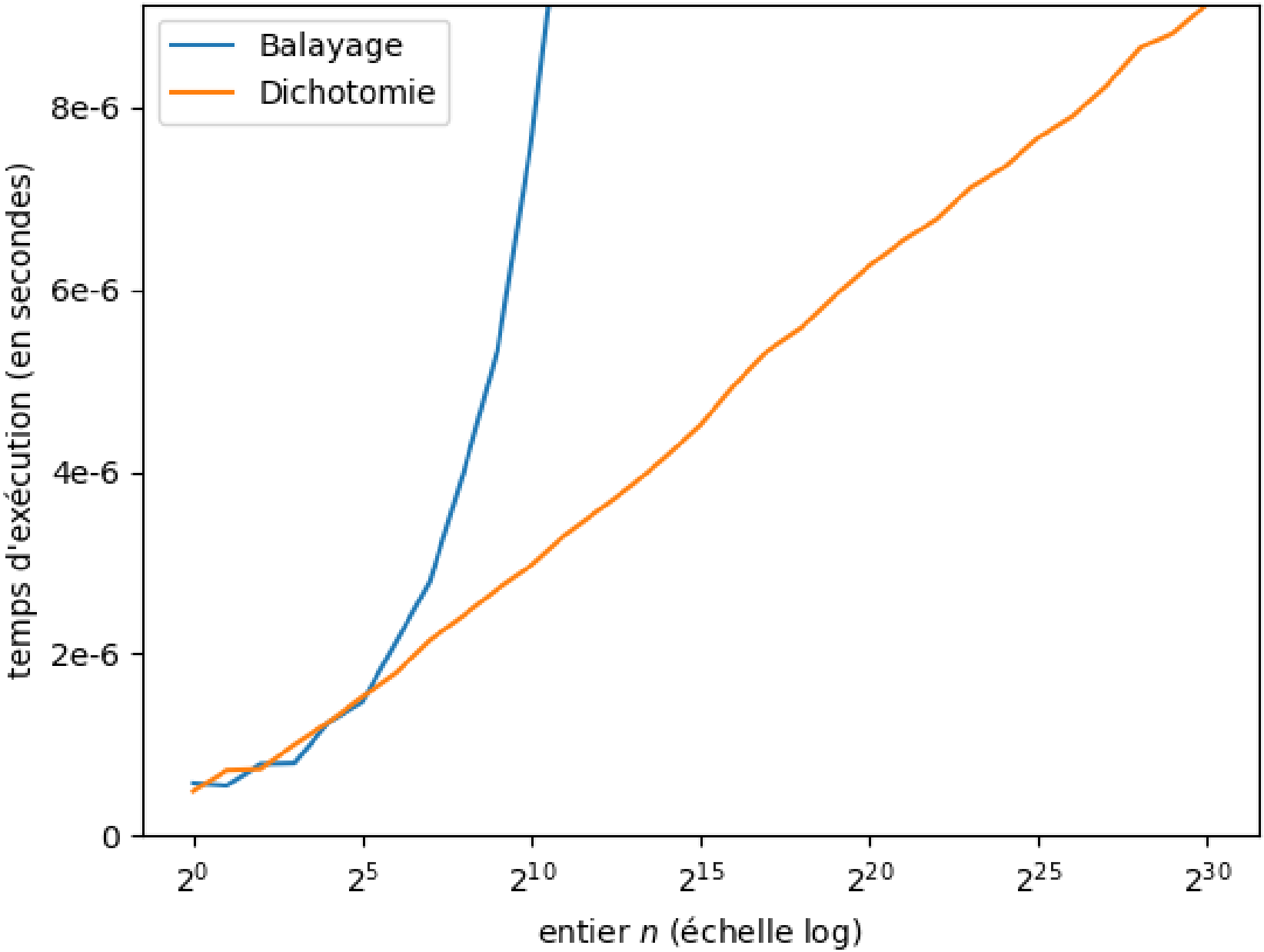
a	b	m	m**2	m**2<=n
0	28	14	196	False
0	14	7	49	False
0	7	3	9	True
3	7	5	25	True
5	7	6	36	False
5	6			

Application : Racine carrée entière

1. Écrire la fonction `racine_entiere_2` qui implémente la spécification donnée, à savoir :
   - **Type du paramètre :** `n` est un entier.
   - **Type de la valeur de retour :** `a` est un entier.
   - **Pré-condition :** `n>=0`
   - **Post-condition :** `a>=0 and a**2<=n<(a+1)**2`

   L'algorithme utilisé est une recherche par dichotomie.
2. Déterminer les valeurs successives de `a`, `b` et `m` ainsi que la valeur de retour dans les cas suivants :
   - `n = 0`
   - `n = 1`
   - `n = 17`
3. Exprimer, en fonction de `n`, le nombre d'itérations dans le pire des cas. Comparer avec la fonction `racine_entiere_1`.


Balayage
Dichotomie
8e-3
6e-3
4e-3
2e-3
0
temps d'exécution (en secondes)
2^0
2^5
2^{10}
2^{15}
2^{20}
2^{25}
2^{30}
entier n (échelle log)


Balayage
Dichotomie
8e-6
6e-6
4e-6
2e-6
0
temps d'exécution (en secondes)
2^0
2^5
2^{10}
2^{15}
2^{20}
2^{25}
2^{30}
entier n (échelle log)

Complexités usuelles

Quelques complexités usuelles :

Situation	Complexité
Parcours linéaire d'un tableau de longueur n	n
Boucles imbriquées sur deux indices de 1 à n	n^2
Parcours complet d'une matrice à n_l lignes et n_c colonnes	$n_l \cdot n_c$
Recherche dichotomique dans un tableau trié de taille n	$\log_2 n$
Algorithmes de tri quadratiques (insertion, sélection)	n^2
Algorithmes de tri *diviser pour régner* (fusion, rapide)	$n \cdot \log_2 n$