Complexité

1 Complexité

Notation mathématique

Type de ressource

Complexité dans le pire des cas

Complexité en moyenne

Complexité temporelle et temps de calcul

2 Calcul de complexité temporelle

Algorithme itératif

Exercice 1 : Doublons dans un tableau

On désire obtenir un algorithme qui détermine si un tableau présente des doublons en son sein.

1. Rédiger un algorithme naïf qui résout le problème. Quelle est sa complexité ?
2. Rédiger maintenant un second algorithme en supposant cette fois le tableau trié. Quelle est sa complexité ? Sachant qu'il existe des algorithmes de tri de complexité $\Theta(n \log n)$, a-t-on intérêt à trier le tableau pour résoudre ce problème ?

Exercice 2 : Équilibre d'un tableau

On se donne un tableau t de n entiers relatifs et on cherche la valeur minimale de

$$\Delta_k \coloneqq (t_0 + \cdots + t_{k-1}) - (t_k + \cdots + t_{n-1}) = \sum_{i=0}^{k-1} t_i - \sum_{i=k}^{n-1} t_i$$

lorsqu'on fait varier k dans $[\![0, n]\!]$.

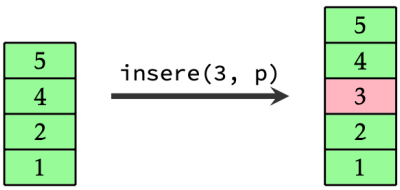
1. Rédiger une fonction `delta(t: list[int], k: int) -> int` qui calcule la quantité Δ_k et en déduire une fonction `equilibre(t: list[int]) -> int` qui résout le problème posé. Évaluer la complexité temporelle de cette dernière fonction.
2. Écrire une fonction `equilibre_lineaire(t: list[int]) -> int` qui résout ce problème en temps linéaire.

Exercice 3 : Tri d'une pile

On rappelle qu'une *pile* en Python est une liste `p` pour laquelle seules les opérations élémentaires suivantes sont autorisées :

— Créer une pile vide : `p = []`
— Tester si la pile est vide : `len(p) == 0`
— Ajouter l'élément x au sommet de la pile : `p.append(x)`
— Obtenir l'élement au sommet d'une pile non vide : `p[-1]`
— Enlever l'élément au sommet d'une pile non vide : `p.pop()`

1. À l'aide d'une pile auxiliaire, rédiger une fonction `insere(x: int, p: list[int]) -> NoneType` qui prend pour argument un entier x et une pile p formée d'entiers triés par ordre croissant et qui insère x au sein de p en préservant l'ordre relatif des éléments.



2. En déduire une fonction `tri(p: list[int]) -> NoneType` qui prend pour argument une pile d'entiers et qui renvoie une nouvelle pile contenant les mêmes éléments triés par ordre croissant.
3. Quelle est la complexité temporelle de cette fonction ?

Exercice 4 : Cherche un entier comme somme de deux entiers

1. Écrire une fonction `cherche_somme(t: list[int], s: int) -> tuple[int, int]` ayant la spécification suivante :
   — Si la fonction renvoie `(i, j)`, alors on a $0 \leqslant i < j < |t|$ et $t_i + t_j = s$.
   — Si la fonction renvoie `None`, alors il n'existe pas de couple (i, j) vérifiant $0 \leqslant i < j < |t|$ et $t_i + t_j = s$.
   Quelle est sa complexité ?

```
In [1]: cherche_somme([15, 1, 3, 5, 6, 7, 10, 1, 8], 11)
Out[1]: (6, 7)

In [2]: cherche_somme([15, 1, 3, 5, 6, 7, 10, 1, 8], 19)
Out[2]: None
```

2. On suppose maintenant que le tableau `t` est trié par ordre croissant. Écrire une fonction

   `cherche_somme_croissant(t: list[int], s: int) -> tuple[int, int]`

   ayant la même spécification que `cherche_somme` mais de complexité linéaire en la taille du tableau.

```
In [4]: cherche_somme_croissant([1, 1, 3, 5, 6, 7, 7, 10, 12, 15], 13)
Out[4]: (0, 8)

In [5]: cherche_somme_croissant([1, 1, 3, 5, 6, 7, 7, 10, 12, 15], 17)
Out[5]: (3, 8)
```

3. Déterminer un algorithme permettant d'implémenter `cherche_somme` avec une complexité quasi-linéaire.

Exercice 5 : Minimum local

On suppose donné un tableau t de longueur n contenant au moins 3 éléments et possédant la propriété suivante : $t_1 \leqslant t_0$ et $t_{n-2} \leqslant t_{n-1}$. Pour tout $k \in [\![1, n-2]\!]$, on dit que le tableau possède un minimum local en k lorsque $t_k \leqslant t_{k-1}$ et $t_k \leqslant t_{k+1}$.

1. Justifier l'existence d'un minimum local dans le tableau t.
2. Écrire une fonction qui détermine un minimum local en cout linéaire.
3. Écrire une fonction récursive qui détermine un minimum local en cout logarithmique.

Exercice 6 : Calcul de complexité

Pour chacune des fonctions suivantes, évaluez la complexité temporelle en fonction de n.

```
def f1(n):
    x = 0
    for i in range(n):
        for j in range(n):
            x = x + 1
    return x
```

```
def f2(n):
    x = 0
    for i in range(n):
        for j in range(i):
            x = x + 1
    return x
```

```
def f3(n):
    x = 0
    for i in range(n):
        j = 0
        while j * j < i:
            x = x + 1
```

```
            j = j + 1
    return x
```

```
def f4(n):
    x = 0
    i = n
    while i > 1:
        x = x + 1
        i = i // 2
    return x
```

```
def f5(n):
    x = 0
    i = n
    while i > 1:
        for j in range(n):
            x = x + 1
        i = i // 2
    return x
```

```
def f6(n):
    x = 0
    i = n
    while i > 1:
        for j in range(i):
            x = x + 1
        i = i // 2
    return x
```

Exercice 7 : Le mur

Vous êtes face à un mur qui s'étend à l'infini dans les deux directions. Il y a une porte dans ce mur, mais vous ne connaissez ni la distance, ni la direction dans laquelle elle se trouve. Par ailleurs, l'obscurité vous empêche de voir la porte à moins d'être juste devant elle. Décrire un algorithme vous permettant de trouver cette porte en un temps linéaire vis-à-vis de la distance qui vous sépare de celle-ci.

Exercice 8 : Problème proposé par le CCC (Comité Contre les Chats)

Le problème est de déterminer à partir de quel étage d'un immeuble sauter du balcon est fatal à un chat. Vous êtes dans un immeuble à n étages (numérotés de 1 à n) et vous disposez de k chats. Il n'y a qu'une opération possible pour tester si la hauteur d'un étage est fatale : faire sauter un chat du balcon. S'il survit, vous pouvez le réutiliser ensuite, sinon vous ne pouvez plus. Vous devez proposer un algorithme pour trouver la hauteur à partir de laquelle un saut est fatal en faisant le minimum de lancers.

1. Si $k \geqslant \lceil \log n \rceil$, proposer un algorithme en $\mathrm{O}(\log n)$ sauts.
2. Si $k < \lfloor \log n \rfloor$, proposer un algorithme en
$$\mathrm{O}\left(k + \frac{n}{2^{k-1}}\right)$$
sauts.
3. Si $k = 2$, proposer un algorithme en $\mathrm{O}(\sqrt{n})$ sauts.

Exercice 9 : Poulidor forever

Expliquer comment trouver le deuxième plus grand élément d'un tableau $[a_0, \ldots, a_{n-1}]$ en effectuant au plus $n + \lfloor \log n \rfloor - 2$ comparaisons. Vous pouvez procéder par analogie avec un tournoi à élimination directe en remarquant que le deuxième joueur le plus fort fait nécessairement partie des adversaires malheureux du vainqueur.

Algorithme récursif

Exercice 10 : Version récursive de la somme des éléments d'un tableau

1. Écrire une fonction `somme_tableau(t: list[int]) -> int` de manière récursive en utilisant le fait que la somme des éléments d'un tableau vide est nulle et que si `t` est un tableau de taille $n \geqslant 1$, la somme de ses éléments est la somme de son premier élément et des éléments restants.

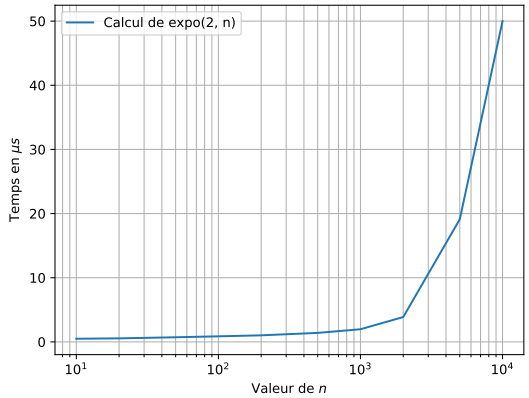
2. Calculer la complexité de cette fonction en utilisant le fait que la création d'une tranche de taille n nécessite n opérations élémentaires.
3. Comparer la performance de cette fonction avec une version itérative de cet algorithme.

Exercice 11 : Complexité de l'exponentiation rapide

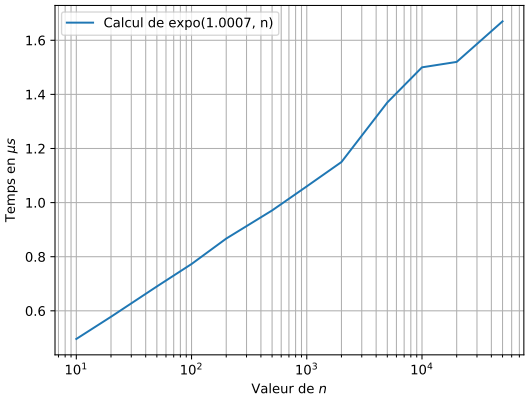
On considère la fonction suivante :

```
def expo(a, n):
    if n == 0:
        return 1
    elif if n % 2 == 0:
        return expo(a * a, n // 2)
    else:
        return a * expo(a, n - 1)
```

1. On note $M(n)$ le nombre de multiplications effectuées lors de l'appel `expo(a, n)`, pour $n \geqslant 0$. Exprimer $M(2n+1)$ et $M(2n)$ en fonction de $M(n)$.
2. En déduire que pour tout $n \geqslant 1$, on a $M(n) \leqslant 1 + 2\log_2 n$.
3. Quelle est la complexité de la fonction `expo` ?
4. Expérimentalement, en mesurant le temps d'exécution de `expo(2, n)` et de `expo(1.0007, n)`, on obtient les courbes suivantes. Comment expliquer ce phénomène ?



Graphique semilog pour l'exponentiation rapide d'un entier.



Graphique semilog pour l'exponentiation rapide d'un flottant.

Exercice 12 : Autour de l'exponentiation rapide

On souhaite écrire une fonction récursive qui calcule a^n.

1. Écrire une telle fonction qui exploite la relation

$$\forall n \in \mathbb{N}, \quad a^n = a^{\lfloor \frac{n}{2} \rfloor} a^{\lceil \frac{n}{2} \rceil}.$$

2. Évaluer le nombre de multiplications à effectuer et comparer cet algorithme avec l'algorithme d'exponentiation rapide.

Exercice 13 : Fibonacci

Dans l'algorithme d'exponentiation rapide permettant de calculer x^n, rien n'impose que x soit un entier. On peut notamment appliquer cet algorithme lorsque x est une matrice. Dans tout l'exercice, lors des questions de complexité, on comptera uniquement les opérations arithmétiques : la somme et le produit de deux entiers.

1. Écrire une fonction Python

   `produit(a: list[list[int]], b: list[list[int]]) -> list[list[int]]`

   qui réalise la multiplication de deux matrices A et B à coefficients entiers, supposées carrées, de même taille $m \times m$. Quelle est la complexité de cette fonction ?
2. En déduire une fonction

   `puissance(m: list[list[int]], n: int) -> list[list[int]]`

   qui calcule M^n pour une matrice M de taille $m \times m$ en utilisant l'algorithme d'exponentiation rapide. Donner la complexité de cette fonction en fonction de m et n.

On définit la suite de Fibonacci par

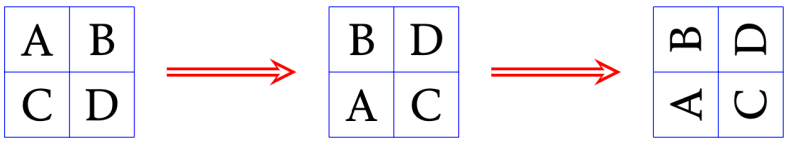
$$F_0 \coloneqq 0, \qquad F_1 \coloneqq 1, \qquad \text{et} \qquad \forall n \in \mathbb{N}, \quad F_{n+2} \coloneqq F_{n+1} + F_n.$$

3. Montrer que
$$\forall n \in \mathbb{N}^*, \quad \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix}$$
4. En déduire une fonction `fibo(n: int) -> int` qui calcule F_n avec une complexité en $\Theta(\log n)$.
5. Soit $b \geqslant 2$. Montrer que si d_n est le nombre de chiffres de F_n en base b, alors
$$d_n = \Theta(n).$$
   En quoi ce résultat semble contradictoire avec le résultat de la question précédente ? Expliquer pourquoi ces deux résultats ne sont finalement pas en contradiction.

Exercice 14 : Rotation

Les processeurs graphiques possèdent en général une fonction de bas niveau appelée *blit* (ou transfert de bloc) qui copie rapidement un bloc rectangulaire d'une image d'un endroit à un autre. L'objectif de cet exercice est de faire tourner une image carrée de $n \times n$ pixels de 90° dans le sens direct en adoptant une stratégie récursive.

— On découpe l'image en 4 blocs de taille $(n/2) \times (n/2)$.
— On déplace chacun de ses blocs à sa position finale à l'aide de 5 blits.
— On fait tourner récursivement chacun de ces blocs.



On supposera dans tout l'exercice que n est une puissance de 2.

1. Exprimer en fonction de n le nombre de fois que la fonction blit est utilisée.
2. Quel est le cout total de cet algorithme lorsque le cout d'un blit d'un bloc $k \times k$ est en $\Theta(k^2)$.
3. Et lorsque ce cout est en $\Theta(k)$.

3 Calcul de complexité temporelle et spatiale

Algorithme itératif

Exercice 15 : Grenouille

Une petite grenouille se trouve face à une rivière. Initialement située sur une des deux rives à la position 0, elle veut se rendre sur la rive opposée à la position $m + 1$. Elle ne peut réaliser que des sauts d'une unité. Heureusement pour elle, des feuilles tombent à la surface de la rivière et peuvent lui permettre de sauter de feuille en feuille.

On se donne un tableau t composé de n entiers représentant les feuilles qui tombent : $t_k \in [\![1, m]\!]$ représente la position où la feuille tombe à l'instant k. L'objectif est de trouver le moment le plus précoce où la grenouille pourra passer d'une rive à l'autre, c'est-à-dire la date à laquelle toutes les positions de 1 à m seront couvertes par une feuille.

1. Rédiger une fonction

   `grenouille(m: int, t: list[int]) -> int`

   qui résout ce problème. Par exemple, pour $m = 5$ et $t = [1, 3, 1, 4, 5, 3, 2, 4]$, cette fonction devra renvoyer 6.
2. Évaluer la complexité temporelle et spatiale de cette fonction.
3. Si ce n'était pas le cas de votre fonction précédente, implémenter une nouvelle version de cette fonction ayant une complexité temporelle en $O(n)$ et une complexité spatiale en $\Theta(m)$.

Algorithme récursif

Exercice 16 : Mémoïsation

On souhaite calculer les termes de la suite (u_n) définie par

$$u_0 \coloneqq 1 \quad \text{et} \quad \forall n \in \mathbb{N}^*, \qquad u_n \coloneqq \frac{u_{n-1}}{1} + \frac{u_{n-2}}{2} + \cdots + \frac{u_0}{n}.$$

Pour cela, on utilise la fonction suivante :

```
def suite(n):
    """suite(n: int) -> float"""
    if n == 0:
        return 1.0
    else:
        s = 0.0
        for k in range(1, n + 1):
            s = s + suite(n - k) / k
        return s
```

On note c_n le nombre de divisions qui sont effectuées lors de l'appel `suite(n)`.

1. Donner une relation de récurrence reliant c_n et les c_k pour $0 \leqslant k < n$.
2. Montrer que

   $$\forall n \in \mathbb{N}, \quad c_n = 2^n - 1.$$

   En déduire la complexité de la fonction `suite`.
3. Écrire une fonction non récursive effectuant le même calcul en un temps plus raisonnable. On pourra utiliser une liste `memo` de longueur $n + 1$, que l'on remplira successivement avec les termes $u_0, u_1, \ldots, u_n$. On déterminera la complexité temporelle et la complexité spatiale de cette nouvelle fonction.

Exercice 17 : Coefficients binomiaux

1. Écrire une fonction récursive `binome(k: int, n: int) -> int` calculant le coefficient $\binom{n}{k}$ en utilisant les relations suivantes :

   $$\forall n \in \mathbb{N}, \qquad \binom{n}{0} = \binom{n}{n} = 1$$
   $$\forall k, n \in \mathbb{N}, \qquad \binom{n+1}{k+1} = \binom{n}{k} + \binom{n}{k+1}.$$

2. Que pensez-vous de la performance de cette fonction ?

Afin d'améliorer les performances de la fonction précédente, on se donne une valeur $N \in \mathbb{N}$ et on crée un tableau bidimensionnel que l'on utilisera comme variable globale.

```
coeff = [[-1 for n in range(N + 1)] for k in range(N + 1)]
```

L'élément `coeff[k][n]` à pour vocation de contenir le coefficient $\binom{n}{k}$. Le fait qu'il contienne initialement -1 est là pour signaler que ce coefficient n'a pas encore été calculé.

3. Écrire une fonction `binome_memoisation(k: int, n: int) -> int` qui, lorsqu'il est appelé avec les entiers k et n tels que $0 \leqslant n \leqslant N$ et $0 \leqslant k \leqslant n$, renvoie $\binom{n}{k}$ s'il est déjà stocké dans le tableau `coeff` à la place `coeff[k][n]` et qui le calcule de manière récursive dans le cas où ce n'est pas déjà le cas, c'est-à-dire si `coeff[k][n]` est égal à -1. On veillera, dans ce dernier cas, à stocker le résultat du calcul dans le tableau `coeff` avant de renvoyer ce résultat.