Lycée Faidherbe, 2019-2020

Cours d'informatique

MP, PC et PSI

Version du 8 septembre 2019

Table des matières

I Cours de seconde année **5**

I Analyse des algorithmes **7**
1 Rappels 7
2 Complexité 8
3 Terminaison 12
4 Correction 14
5 Exercices 17
6 Solutions 19

II Tris simples **21**
1 Position du problème 21
2 Tri par insertion 24
3 Tri par sélection 27
4 Exercices 28
5 Solutions 30

III Récursivité **35**
1 Exemples 35
2 Analyse des algorithmes récursifs 39
3 Avantages et inconvénients 40
4 Exercices 42
5 Solutions 45

IV Tris rapides **53**
1 Retour sur les tris classiques 53
2 Tri-fusion 55
3 Tri rapide 61
4 Exercices 65
5 Solutions 69

V Piles **77**
1 Introduction 77
2 Type de données abstrait 79
3 Piles concrètes 80
4 Applications 84
5 Exercices 87

6 Solutions 93

II Révisions de première année **101**

R-I Modules **103**

1 numpy 103
2 matplotlib 105
3 scipy 107
4 polynomial 109
5 random 110

R-II Méthode d'Euler **111**

1 Équation différentielle 111
2 Méthodes de résolution 113

Première partie

Cours de seconde année

Chapitre I

Analyse des algorithmes

Tester un programme peut montrer que des bogues sont présents, mais jamais montrer leur absence.
E.W. Dijkstra

Résumé

Dans ce chapitre nous allons rappeler la notion de complexité et compléter l'analyse des algorithmes en introduisant l'étude de leur **terminaison** *et de leur* **preuve***. On présente deux outils qui peuvent être utilisés :* **variant** *et* **invariant***.*

1 Rappels

L'analyse d'un programme se fait en trois étapes.

Le programme fournit-il un résultat ? On emploie aussi le verbe **terminer** sous une forme intransitive et on parle de terminaison. Ce qui est en jeu est que l'on doit sortir des boucles.

Le programme fournit-il le bon résultat ? Il existe des outils théoriques (la logique de Hoare) qui permettent une formalisation de l'analyse de ce problème. Nous allons nous contenter de démonstrations de preuves plus élémentaires

Le programme fournit-il le bon résultat dans un temps raisonnable ? On a vu que le but n'est pas de prévoir le temps exact que va demander la résolution d'un problème mais d'avoir une idée de l'accroissement du temps quand la taille des données d'entrée augmente.

Nous avons rencontré jusqu'à présent des programmes suffisamment simples pour que ces questions puissent sembler avoir une réponse évidente mais ce n'est pas toujours le cas.

Nous allons commencer par décrire la dernière étape, c'est la plus importante dans le cadre du programme.

2 Complexité

Le programme fournit-il le bon résultat dans un temps raisonnable ?
Rappelons les étapes de ce calcul.

1. Le plus souvent on choisit une instruction élémentaire signifiante Ce pourra être
   - le nombre de d'affectations,
   - le nombre d'opérations arithmétiques, sommes, produits, quotients, ...
   - le nombre de comparaisons,
   - etc
2. On choisit une mesure de la taille des données d'entrée. Ce peut être
   - la valeur d'une variable entière,
   - le nombre d'éléments des listes, tableaux, fichiers,
   - la longueur d'un mot,
   - le degré d'un polynôme,
   - le nombre de lignes (ou de colonnes ou leur produit) dans le cas d'une matrice,
   - le nombre de bit dans sa représentation en base 2 d'une variable entière,
   - etc
3. On calcule (ou on majore) le nombre maximal d'opérations élémentaires signifiante en fonction de la taille de la donnée d'entrée. On parle de complexité maximale.
   On cherche donc une fonction $C(n)$ telle que, pour toutes les données d'entrée de taille n le nombre d'instruction effectuées est majoré par $C(n)$.
4. On donne un ordre de grandeur de cette fonction sous la forme $C(n) = \mathcal{O}\big(f(n)\big)$ où f est une fonction simple.
   On rappelle que $C(n) = \mathcal{O}\big(f(n)\big)$ signifie qu'il existe un réel K et un entier n_0 tels que $C(n) \leqslant Kf(n)$ pour tout entier n supérieur à n_0.
   - On parle de complexité linéaire quand la complexité est un $\mathcal{O}(n)$,
   - de complexité quadratique quand elle est un $\mathcal{O}(n^2)$,
   - de complexité polynomiale quand elle est un $\mathcal{O}(n^p)$,
   - de complexité logarithmique (ou quasi-constante) quand elle est un $\mathcal{O}\big(\log_2(n)\big)$,
   - de complexité quasi-linéaire quand elle est un $\mathcal{O}\big(n\log_2(n)\big)$.
   - de complexité exponentielle quand elle est un $\mathcal{O}(a^n)$.

2.1 Exemples

Somme d'une liste

```
def somme(liste):
    """Entrée : une liste de nombres
       Sortie : la somme des termes de la liste"""
    s = 0
    n = len(liste)
    for i in range(n):
        s = s + liste[i]
    return s
```

La boucle est effectuée n fois et on y fait une affectation : la complexité en nombre d'affectations est donc $n + 2 = \mathcal{O}(n)$.

Recherche dans une liste

On reprend l'algorithme classique

```
def appartient(a,liste):
    """Entrée : un élément a et une liste
       Sortie : vrai ou faux selon que a est
                ou n'est pas dans la liste"""
    present = false # On n'a pas encore trouvé
    n = len(liste)
    i  = 0 # On doit gérer les indices
    while not present and i < n:
        if liste[i] == a:
            present = True
        i = i + 1
    return present
```

La boucle est effectuée au plus n fois on y fait une comparaison : la complexité est donc linéaire en fonction de la taille n de la liste si on compte le nombre de comparaisons .

Logarithme entier

Il est souvent utile de chercher l'entier p tel que $2^p \leqslant n < 2^{p+1}$:
p est le logarithme entier de n.

```
def logEntier(n):
    """Entrée : un entier strictement positif
       Sortie : le plus grand entier p tel que 2^p <= n"""
    p = 0
    puiss2 = 1
    while n >= puiss2:
        p = p + 1
        puiss2 = 2*puiss2
    return (p - 1)
```

La boucle est parcourue $p+1$ fois et on a $2^p \leqslant n$ donc $p \leqslant \log_2(n)$:
la complexité est un $\mathcal{O}\big(\ln(n)\big)$.

Zéro d'une fonction

```
def zero(f, a, b, e =1e-6):
    """Entrées : une fonction continue de [a;b] dans R,
                 telle que f(a).f(b) <= 0
                 et une marge d'erreur e
       Sortie  : un réel c tel que f s'annule entre c-e et c+e
         """
    u = a
    v = b
    while abs(v-u) > e:
        w = (u + v)/2
        if f(u)*f(w) <= 0:
            v = w
        else:
            u = w
    return u
```

Si n est le nombre d'itérations la complexité est un $\mathcal{O}(n)$.
On a $v-u=(b-a).2^{-n}$ et on s'arrête lorsqu'on a $u-v \leqslant \varepsilon$ pour la première fois

ainsi on a $(b-a).2^{-(n-1)} > \varepsilon$ d'où $2^n < \frac{2(b-a)}{\varepsilon}$.
Si on écrit la précision sous la forme 10^{-p}, on a $2^n < 2.10^p(b-a)$ donc $n < \frac{ln(10)}{\ln(2)}p + frac\ln\big(2(b-a)\ln(2)$:
la complexité est un $\mathcal{O}(p)$.

2.2 Une étude de cas

On suppose donnée une liste de nombres qui peuvent être positifs ou négatifs.
Le but est trouver la somme de termes consécutifs maximale.
Par exemple si la liste est `var = [-2, 2, -1, 3, -4, 1]` la somme maximale est $4 = 2-1+3$ obtenue en sommant les termes d'indices 1 à 3. En notation Python c'est la somme de la liste extraite `var[1:4]`.
On prendra la convention que la somme des termes d'une liste vide est nulle : comme une des somme possibles est alors 0 , le maximum sera positif (ou nul).
On évaluera la complexité des algorithmes en comptant le nombre d'additions.

Algorithme naïf

La première idée est de suivre l'énoncé.
On écrit une fonction (classique) de somme des termes d'une liste puis on calcule toutes les sommes partielles de la liste pour en déterminer le maximum.

```
1  def somme(liste,debut,fin):
2      """Entrée : une liste de nombres et 2 indices
3         Sortie : la somme des termes entre les deux indices,
4                  les bornes sont comprises."""
5      somme = 0
6      for i in range(debut,fin+1):
7          somme = somme + liste[i]
8      return somme
9
10 def sommeMax(liste):
11     """Entrée : une liste de nombres
12        Sortie : la somme maximale de termes consécutifs."""
13     n = len(liste)
14     maxi = 0
15     for i in range(n):
16         for j in range(i,n):
17             calcul = somme(liste,i,j)
18             if calcul > maxi:
19                 maxi = calcul
20     return maxi
```

- À la ligne 5 on doit délimiter par `fin + 1` pour s'arrêter à `fin`.
- Comme le cas d'une liste vide a été pris en compte avec `maxi = 0` on ne fera les calculs que pour $0 \leqslant i \leqslant j \leqslant n-1$ ce qui explique les lignes 14 et 15.

Complexité : la fonction `somme` effectue `fin + 1 - debut` additions.
Le nombre d'additions effectuées pour une liste de taille n dans `sommeMax` est donc, en posant $k = j+1-i$ puis $p = n-i$,

$$C_1(n) = \sum_{i=0}^{n-1}\sum_{j=i}^{n-1}(j+1-i) = \sum_{i=0}^{n-1}\sum_{k=1}^{n-i}k = \sum_{i=0}^{n-1}\frac{(n-i)(n-i+1)}{2}$$
$$= \sum_{p=1}^{n}\frac{p(p+1)}{2} = \frac{1}{2}\sum_{p=1}^{n}p^2 + \frac{1}{2}\sum_{p=1}^{n}p = \frac{n^3+3n^2+2n}{12} = \mathcal{O}(n^3)$$

Ne pas répéter

L'algorithme précédent est peu efficace. On peut remarquer que lorsque l'on calcule la somme des termes d'indices 0 à 3 on recommence à calculer la somme des termes d'indices 0 à 2.
Une des règles de bases de l'informatique est de ne jamais répéter deux fois les mêmes instructions : il y a sans doute moyen d'améliorer la complexité.
Il suffit de calculer les sommes dont le premier indice est i en ajoutant simplement le dernier terme à chaque étape et en comparant avec le maximum provisoire.

```
def sommeMax(liste):
    """Entrée : une liste de nombres
       Sortie : la somme maximale de termes consécutifs."""
    n = len(liste)
    maxi = 0
    for i in range(n):
        calcul = 0
        for j in range(i,n):
            calcul = calcul + liste[j]
            if calcul > maxi:
                maxi = calcul
    return maxi
```

Il faut penser à initialiser la somme partielle pour chaque i.
Complexité : le programme principal a la même structure mais on ne fait pas appel à une fonction externe : dans la boucle d'indice j on ne fait qu'une addition.

$$C_2(n) = \sum_{i=0}^{n-1}\sum_{j=i}^{n-1} 1 = \sum_{i=0}^{n-1}(n-1-i+1) = \sum_{p=1}^{n} p = \frac{n^2+n}{2} = \mathcal{O}(n^2)$$

Encore mieux

Il semble difficile de faire mieux que la complexité quadratique ci-dessus. En effet il y a $\frac{n(n+1)}{2}$ sommes partielles dont il faut déterminer le maximum. Pour pouvoir diminuer la complexité il faudra ne pas calculer un grand nombre de sommes. Mais cela est en fait possible.

On s'intéresse à la tranche maximale de la liste des i premier termes pour chaque $i \in \{0, 1, \ldots, n\}$, n étant la longueur de la liste ; on la note MaxiProv(i) (pour maximum provisoire). Nous allons calculer cette tranche maximale de proche en proche.

$$\texttt{MaxiProv(i)} = \max\left\{\sum_{k=p}^{q-1}\texttt{liste[k]}\ ;\ 0 \leqslant p \leqslant q < i\right\}$$

On doit pouvoir choisir $p = q$ pour obtenir une somme sans indice donc nulle

- `MaxiProv(0)` vaut 0 : c'est la seule somme possible avec 0 termes.
- Comme toute tranche des $i-1$ premiers termes est aussi une tranchedes i premiers termes on a la croissance des maximums provisoires : `MaxiProv(i) <= MaxiProv(i+1)`.
- Quand on a `MaxiProv(i) < MaxiProv(i+1)` c'est qu'on a trouvé une tranche plus grande en ajoutant le terme d'indice `i` : cette tranche doit donc contenir `i`. C'est une tranche terminale, somme des j derniers termes ($j \leqslant i+1$).
- On est conduit à calculer, pour chaque i, la tranche terminale maximale

$$\texttt{MaxiTerm(i)} = \max\left\{\sum_{k=p}^{i-1}\texttt{liste[k]}\ ;\ 0 \leqslant p \leqslant i\right\}$$

- On a donc `MaxiProv(i+1) = max(MaxiProv(i), MaxiTerm(i+1))`

- Il reste à calculer `MaxiTerm(i+1)`, on va l'exprmier en fonction de `MaxiTerm(i)`.
  On a envie de calculer `MaxiTerm(i+1) = MaxiTerm(i) + liste[i]`. Cependant il y a un cas particulier. En effet `MaxiTerm(i)` est supérieur à toute somme finissante en $i-1$ donc `MaxiTerm(i) + liste[i]` est supérieur à toute somme finissante en i **qui contient** `liste[i]`.
  Il reste à considérer la somme des 0 derniers termes qui est aussi une somme finissante. On aboutit à la formule `MaxiTerm(i+1) = max(MaxiTerm(i) + liste[i], 0)`.

On peut donc écrire l'algorithme

```
def sommeMax(liste):
    """Entrée : une liste de nombres
       Sortie : la somme maximale de termes consécutifs."""
    n = len(liste)
    maxiProv = 0 # Initialisation pour i = 0
    maxiTerm = 0     # Initialisation pour i = 0
    for i in range(n): # On passe de i à i+1
        maxiTerm = max(maxiTerm + liste[i], 0)
        maxiProv = max(maxiProv, maxiTerm)
    return maxiProv
```

La complexité est alors immédiate : on effectue un nombre fini d'instructions dans la boucle (1 additions, 2 comparaisons, 2 affectations) donc la complexité est un $\mathcal{O}(n)$.

3 Terminaison

Nous étudions dans cette partie la première question à se poser lorsqu'on analyse complètement un algorithme : **Le programme fournit-il un résultat ?**
Une réponse simple serait "*on le lance et on voit bien s'il s'arrête*"
Cependant il se peut que le programme boucle indéfiniment pour certaines valeurs, pas toutes. Il arrive aussi que le temps de calcul soit plus long que le temps d'attente supportable.
La question revient à se demander si le nombre d'instruction est borné : parfois l'étude de la terminaison se fait en parallèle de l'étude de la complexité.
Cependant on pourra aussi prouver la terminaison directement et ainsi utiliser l'existence d'un résultat dans la suite.
Il faut donc essayer de prouver la terminaison par une étude a-priori.
On peut remarquer deux limites à cette interrogation :

3.1 Délimitation du problème

Un programme python est composé d'instructions élémentaires qui sont des appels à d'autres fonctions et d'instructions conditionnelles simples (`if`) ou de répétitions (`for` et `while`).
Les boucles `for` en Python ne font qu'un nombre fini et fixé à l'avance d'opérations.
Ainsi la répétition sans fin peut se produire dans les cas suivants.

- Lors d'un appel à une fonction il se peut que celle-ci ne termine pas. C'est un problème rencontré lorsqu'une fonction s'appelle elle-même : c'est la récursivité que nous étudierons plus tard.
- Dans une boucle `while` il se peut que la condition qui déclenche l'exécution de la boucle soit toujours vérifiée : un exemple caricatural est le cas d'une instruction `while True`.

Pour prouver qu'un programme comportant une boucle `while` termine il faut donc prouver que la condition testée dans la ligne `while` *condition* finit par être contredite après un nombre fini d'itérations.

3.2 Exemples

Recherche dans une liste

```
def appartient(a,liste):
    """Entrée : un élément a et une liste
       Sortie : vrai ou faux selon que a est
                ou n'est pas dans la liste"""
    present = false # On n'a pas encore trouvé
    n = len(liste)
    i  = 0 # On doit gérer les indices
    while not present and i < n:
        if liste[i] == a:
            present = True
        i = i + 1
    return present
```

Comme on incrémente `i` à chaque passage de la boucle la condition `not present and i < n` devient fausse après n étapes au plus : l'algorithme termine

Cette terminaison est plus directement visible avec l'écriture utilisant une boucle `for` que permet python.

```
def appartient(a,liste):
    """Entrée : un élément a et une liste
       Sortie : vrai ou faux selon que a est
                ou n'est pas dans la liste"""
    for x in liste:
        if x == a:
            return True
    return False
```

Logarithme entier

```
def logEntier(n):
    """Entrée : un entier strictement positif
       Sortie : le plus grand entier p tel que 2^p <= n"""
    p = 0
    puiss2 = 1
    while n >= puiss2:
        p = p + 1
        puiss2 = 2*puiss2
    return (p - 1)
```

La variable `puiss2` vaut 2^p et p augmente de 1 à chaque passage dans la boucle donc `puiss2` tend vers l'infini et finit par dépasser n après un nombre fini de passage : la boucle s'arrête.

On peut aussi considérer l'entier calculé à partir des variables : `n - puiss2`. Il décroît de 1 au moins à chaque passage dans la boucle et il est positif tant que les instructions de la boucle peuvent être exécutées. Comme une suite strictement decroissante d'entiers ne peut prendre qu'un nombre fini de valeurs, la boucle ne peut être parcourue qu'un nombre fini de fois.
C'est un variant de boucle :

Définition : Variant de boucle
Un variant de boucle est une valeur entière dépendant des variables du programme
- *qui est positive quand la condition de la boucle est réalisée*
- *et qui décroît strictement à chaque passage dans la boucle.*

S'il existe un variant de boucle alors l'algorithme termine.
Les boucles `for i in range(n)` ont le variant de boucle naturel `n - i` : on retrouve ainsi le fait qu'elles terminent toujours.

Suite de Syracuse

On peut remarquer que la preuve de terminaison peut être impossible.
La suite de Syracuse de terme initial a est la suite définie par

$$u_0 = a \text{ et } u_{n+1} = \begin{cases} u_n/2 & \text{si } u_n \text{ est pair} \\ 3u_n + 1 & \text{si } u_n \text{ est pair} \end{cases} \text{ pour tout } n \in \mathbb{N}$$

On remarque que, quelle que soit la valeur entière positive de u_0, la suite finit par devenir périodique. Dès que u_n vaut 1, la suite se poursuit sous la forme $1, 4, 2, 1, 4, 2, 1, 4, 2, \cdots$.
Cependant on ne sait pas (pour l'instant) prouver ce résultat.
On ne sait donc pas prouver que l'algorithme suivant termine.

```
def syracuse(a):
    """Entrée : un entier strictement positif
       Sortie : le premier entier n en lequel la suite de
          Syracuse
                de terme initial a prend la valeur 1"""
    u = a
    n = 0
    while u != 1:
        if u%2 == 0:
            u = u//2
        else:
            u = 3*u + 1
        n = n + 1
    return n
```

4 Correction

Le programme fournit-il le bon résultat ?
Une réponse courante est de tester le programme sur des entrées et espérer que, s'il a donné le bon résultat 1000 fois de suite, ce sera encore le cas la 1001-ième fois. On sait cependant que ce n'est pas le cas, même si on remplace 1000 par 1 000 000.

Voici un programme simple

```
def maxi(liste):
    """Entrée : une liste non vide de nombres
       Sortie : l'élément maximal de la liste"""
    grand = 0
    for x in liste:
        if x > grand:
            grand = x
    return grand
```

Il est probable que nous allons le tester avec des listes du genre `[25, 54, 37, 22]` et considérer qu'il est correct. Cependant, lors de son utilisation, il risque de poser un problème avec `maxi([-2, -3])` car il renverra 0 au lieu de -2.
On voit qu'il est assez simple de montrer qu'un algorithme n'est pas correct, il "suffit" de trouver un contre-exemple.
Un simple changement d'initialisation par `grand = liste[0]` suffit à résoudre ce problème mais on ne peut être certain qu'il n'y aura pas d'autres problèmes. Il faudra une preuve formelle.

Le cas des instructions sans boucle est élémentaire : la preuve est celle des instructions utilisées. Il existe un outil formel qui permet de construire une preuve.

4.1 Invariants de boucles

Dans la cas de boucles `for` la preuve est souvent donnée par la construction de l'algorithme qui est en fait une récurrence.

```
def somme(liste):
    """Entrée : une liste de nombres
       Sortie : la somme des termes de la liste"""
    s = 0
    n = len(liste)
    for i in range(n):
        s = s + liste[i]
    return s
```

On veut prouver que la variable `s` contienne à la fin la somme des termes. On remarque que la calcul `s = s + liste[i]` correspond à la récurrence $\sum_{k=0}^{i} x_k = x_i + \sum_{k=0}^{i-1} x_k$. Si on note $P(i)$ la propriété `s` a pour valeur $\sum_{k=0}^{i-1} x_k$ on a les propriétés

- $P(0)$ est vraie avant le premier passage de la boucle car une somme vide vaut 0,
- si $P(i)$ est vraie avec $i < n-1$ alors $P(i+1)$ est vérifiée après l'exécution des instructions de la boucle pour l'indice i,
- $P(n)$ prouve le résultat attendu.

On arrive donc à la notion d'invariant de boucle

Définition : Invariant de boucle indicée

Un invariant d'une boucle `for i in range(a,b):` *est une propriété $P(i)$ dépendant des variables et de l'indice telle que*

- *$P(a)$ est vraie avant le premier passage de la boucle,*
- *si $P(i)$ est vraie avec $i < b$ alors $P(i+1)$ est vérifiée après l'exécution des instructions de la boucle pour l'indice i,*
- *$P(b)$ prouve le résultat attendu.*

Dans le cas des boucles plus générales, les boucles `while` *condition*, il n'y a pas d'indice qui permet de suivre les itérations des boucles. On généralise la notion d'invariant.

Définition : Invariant de boucle

Un invariant d'une boucle `while` *condition est une propriété P dépendant des variables telle que*

- *P est vraie avant le premier passage de la boucle,*
- *si P est vraie et si la condition est vérifiée alors P est vraie après l'exécution des instructions de la boucle,*
- *P et la négation de la condition prouvent le résultat attendu.*

4.2 Exemples

Recherche dans une liste

```
def appartient(a,liste):
    """Entrée : un élément a et une liste
       Sortie : vrai ou faux selon que a est
                ou n'est pas dans la liste"""
    present = false # On n'a pas encore trouvé
    n = len(liste)
    i  = 0 # On doit gérer les indices
    while not present and i < n:
        if liste[i] == a:
            present = True
        i = i + 1
    return present
```

Un invariant de boucle possible est $P(i)$: `present` est vrai si et seulement si un des termes de la liste d'indice compris entre 0 et $i-1$ vaut `a`.

Logarithme entier

```
def logEntier(n):
    """Entrée : un entier strictement positif
       Sortie : le plus grand entier p tel que 2^p <= n"""
    p = 0
    puiss2 = 1
    while n >= puiss2:
        p = p + 1
        puiss2 = 2*puiss2
    return (p - 1)
```

Un premier invariant, facilement démontrable, est `puiss2 = 2**p`.
Cependant in ne suffit pas à prouver le résultat souhaité car la négation de la condition donne seulement $n < 2^p$ et on voudrait avoir $2^{p-1} \leqslant n$.
On peut simplement ajouter cette condition dans l'invariant :
$\mathcal{P}$: `puiss2 = 2**p` et `2**(p-1) <= n`

- Lors de l'initialisation on a `p = 0`, `puiss2 = 1 = 2**p` et `2**(p-1) = 1/2 <= 1 <= n`.
- Si $\mathcal{P}$ est vraie avec `n >= puiss2`, on note `p '` et `puiss2'` les valeurs des variables après les instructions de la boucle. On a `puiss2' = 2*puiss2 = 2*2**p = 2**(p+1) = 2**p'` et `2**(p'-1) = 2**p <= n` car la condition de boucle est vérifiée.
  Si $\mathcal{P}$ est vraie après le passage dans la boucle.
- La négation de la condition implique `n < puiss2` donc, d'après $\mathcal{P}$,
  `2**(p-1) <= n v 2**p` : $p-1$ est bien la valeur attendue.

5 Exercices

Exercice 1 — pif paf pouf
Évaluer le nombre d'exécution des opérations `pif`, `paf`, `pouf` lors du déroulement des trois fonctions suivantes

```
def un(n):
    for i in range(n):
        pif
    for j in range(n):
        paf
    for k in range(n):
        pouf
```

```
def deux(n):
    for i in range(n):
        pif
        for j in range(n):
            paf
    for k in range(n):
        pouf
```

```
def trois(n):
    for i in range(n):
        pif
        for j in range(n):
            paf
            for k in range(n):
                pouf
```

Exercice 2 — Valeurs d'une suite
On définit une suite (u_n) par $u_0 = 1$ et $u_{n+1} = \frac{1}{2}\left(u_n + \frac{2}{u_n}\right)$.

Étant donné un paramètre n on souhaite renvoyer une liste contenant les valeurs de u_k pour k variant de 0 à n. Voici les solutions proposées par trois étudiants.

```
def u(n):
    """Entrée : un entier n
       Sortie : la valeur du n-ième terme de la suite"""
    res = 1.0
    for i in range(n):
        res = (res + 2/res)/2
    return res

def suiteA(n):
    """Entrée : un entier n
       Sortie : la suite des termes d'indice 0 à n  de la
          suite"""
    res = []
    for i in range(n+1):
        res.append(u(i))
    return res
```

```
def suiteB(n):
    res = [1.0]
    for i in range(n):
        u = res[i-1]
        v = (u + 2/u)/2
        res.append(v)
    return res
```

```python
def suiteC(n):
    res = [1.0]*(n+1)
    for i in range(n):
        u = res[i]
        res[i+1] = (u + 2/u)/2
    return res
```

L'un des algorithme est faux, corrigez-le et calculer la complexité.
Calculer la complexité des 2 autres.

Exercice 3 — Exponentiation rapide

1. Écrire un algorithme simple calculant a^n. Étudier sa complexité.
2. Si on écrit n en base 2, $n = \sum_{k=0}^{p} \varepsilon_k 2^k$ avec $\varepsilon_k \in \{0,1\}$, on voit qu'on peut écrire

$$x^n = x^{\sum_{k=0}^{p} \varepsilon_k 2^k} = \prod_{k=0}^{p} \left(x^{2^k}\right)^{\varepsilon_k}$$

En calculant x^{2^k} sous la forme $x^{2^{k+1}} = x^{2^k}.x^{2^k}$ on voit qu'on pourra calculer la puissance avec $2p+1$ produits.
Voici un algorithme qu'on peut déduire de ces remarques.

```python
def expRapide(x,n):
    """Entrée : un réel x et un entier positif n
      Sortie : x à la puissance n"""
    reste = n
    produit = 1
    puissance = x
    while reste > 0:
        if reste%2 == 1:
            produit = produit*puissance
        reste = reste//2
        puissance = puissance*puissance
    return produit
```

- Prouver que l'algorithme termine.
- Prouver que `produit*(puissance**reste)` est un invariant de boucle. En déduire une preuve de l'algorithme.
- Quelle est sa complexité ?

Exercice 4 — Méthode de Hörner

Un polynôme $P(X) = \sum_{k=0}^{d} a_k X^k$ est représenté par la liste de longueur $d+1$ de ses coefficients $[a_0, a_1, \dots, a_d]$.

1. Écrire un algorithme `eval(P,a)` qui calcule $P(a)$ où P est représenté par une liste. On donnera un algorithme qui effectuera au plus $2n$ multiplications et aucun calcul de puissance (`a**k`) où n est la longueur de la liste.
2. La méthode de Hörner consiste à écrire $P(x)$ sous la forme

$$P(x) = a_0 + x\left(a_1 + x\left(a_2 + \cdots + x\big(a_{d-1} + x(a_d)\big)\right)\right)$$

Écrire un algorithme `eval(P,a)` qui calcule $P(a)$ en utilisant cette méthode.
Combien d'addition et de multiplication effectue-t-il ?

6 Solutions

Solution de l'exercice 1 -
Dans `un` : n `pif`, n `paf` et n `pouf`.
Dans `deux` : n `pif`, n^2 `paf` et n `pouf`.
Dans `trois` : n `pif`, n^2 `paf` et n^3 `pouf`.

Solution de l'exercice 2 -
`suiteA` calcule u_k pour chaque k avec `u`, ce qui demande $3k$ opérations. Au total on effectue $\sum_{k=0}^{n} 3k = \frac{3}{2}n(n+1)$ opérations : la complexité est un $\mathcal{O}(n^2)$.

`suiteB` pourrait donner un erreur dès le premier appel car il cherche `res[0-1]` mais, en Python, cela fournit le dernier terme de la liste, ici `res[0]`.
Au premier passage de la boucle on obtient bien $[u_0, u_1]$.
Mais ensuite, pour $i = 1$, on appelle `res[1-1]` donc on calcule de nouveau u_1.
Le résultat sera donc $[u_0, u_1, u_1, u_2, \ldots, u_{n-1}]$.
Pour corriger l'algorithme il suffit d'écrire `u = res[i]` à la ligne 6.
L'algorithme effectue alors $3n$ opérations : la complexité est un $\mathcal{O}(n)$.

`suiteC` ne calcule qu'une fois chaque u_k : la complexité est un $\mathcal{O}(n)$.

Solution de l'exercice 3 -
Ici aussi un travail d'écriture rendra l'algorithme plus efficace.

1. C'est une boucle classique

```
def exp(x,n):
    """Entrée : un réel x et un entier positif n
      Sortie : x à la puissance n"""
    produit = 1
    for i in range(n):
        produit = produit*x
    return produit
```

   On effectue un produit à chaque passage dans la boucle : la complexité est un $\mathcal{O}(n)$.

2. On notera r_k, p_k et x_k les valeurs des variables `reste`, `produit` et `puissance` lors des passages dans la boucle `while`.
   Initialement on a $r_0 = n$, $p_0 = 1$ et $x_0 = x$.
   - Pour $r_k > 0$ on a $r_k//2 \leqslant \frac{r_k}{2} < r_k$ donc la suite entière (r_k) est strictement décroissante : elle atteint la valeur 0 donc l'algorithme termine. On note m le nombre de passage dans la boucle.
   - Si r_k est pair on a $p_{k+1} = p_k$, $r_{k+1} = \frac{r_k}{2}$ et $x_{k+1} = x_k^2$ donc $p_{k+1}x_{k+1}^{r_{k+1}} = p_k x_k^{2r_k/2} = p_k x_k^{r_k}$.
     Si r_k est impair on a $p_{k+1} = p_k x_k$, $r_{k+1} = \frac{r_k - 1}{2}$ et $x_{k+1} = x_k^2$ donc $p_{k+1}x_{k+1}^{r_{k+1}} = p_k x_k x_k^{2(r_k-1)/2} = p_k x_k^{r_k}$.
     Dans les deux $p_k x_k^{r_k}$ est conservé, c'est un invariant de boucle.
     On a $p_0 x_0^{r_0} = x^n$ et, en sortie de boucle, $r_m = 0$ donc $p_m x_m^{r_m} = p_m$. la valeur retournée est p_m qui vaut bien x^n en raison de l'invariance de boucle.
   - Il y a 3 affectations initiales.
     Dans chaque passage de la boucle on fait 1 calcul de reste, 1 comparaison 1 ou 2 produits, une division et 1 ou 2 affectations. Le nombre d'opérations élémentaires est majoré par 6.
     La complexité est majorée par $6m + 3$.

     On suppose qu'on a $n > 0$ (sinon il n'y a pas de boucle).
     Il existe un entier p tel que $2^p \leqslant n = r_0 < 2^{p+1}$.

On a alors $2^{p-1} \leqslant r_1 = r_0//2 < 2^p$ puis, par récurrence, $2^{p-k} \leqslant r_k < 2^{p+1-k}$ tant que $r_{k-1} > 0$.
Pour $k = p$ on obtient $1 \leqslant r_p < 2$ donc $r_p = 1$ puis $r_{p+1} = 0$.
On effectue donc $p+1$ passages dans la boucle donc la complexité est majorée par $6p+9$.
On a, en prenant les logarithmes, $p \leqslant \frac{\ln(n)}{\ln(2)}$ donc la complexité est majorée par $A\ln(n) + B = \mathcal{O}\big(\ln(n)\big)$.

Solution de l'exercice 4 -
Comme on calcule toutes les puissances de a il vaut mieux éviter de faire appel à une fonction d'exponentiation même rapide.

1. On va calculer x^k et ajouter $a_k x^k$ dans la même boucle.

```
def eval(P,a):
    """Entrée : un polynôme P sous la forme
                de la liste de ses coefficients
                en commençant par le coefficient constant.
       Sortie : P(a)"""
    n=len(P)
    somme = 0
    puissance = 1
    for i in range(n): # n = deg(P)+1
        somme = somme + P[i]*puissance
        puissance = puissance*a
    return somme
```

   On fait 2 multiplications dans chaque passage de la boucle donc $2n$ multiplications.
   Un invariant de boucle peut être somme $= \sum_{k=0}^{i-1} a_k a^k$ au début du passage dans la boucle pour i.

2. Il faut faire attention ici car on ajoute les termes à partir du dernier.

```
def eval(P,a):
    """Entrée : un polynôme P sous la forme
                de la liste de ses coefficients
                en commençant par le coefficient constant.
       Sortie : P(a)"""
    n=len(P)
    somme = 0
    for i in range(n): # n = deg(P)+1
        somme = somme*x + P[n-1-i]
        puissance = puissance*a
    return somme
```

   On fait 1 multiplication dans chaque passage de la boucle donc n multiplications.
   La complexité asymptotique n'a pas été améliorée, on a toujours un $\mathcal{O}(n)$, mais on fait 2 fois moins de multiplications et autant d'additions.
   Un invariant de boucle peut être $P(a) = \sum_{k=0}^{n-i-2} a_k a^k + \texttt{somme} a^{n-1-i}$ au début du passage dans la boucle pour i.

Chapitre II

TRIS SIMPLES

Résumé *Dans ce chapitre nous allons donner deux algorithmes de tris de listes qui consistent à avancer pas-à-pas dans l'objectif souhaité. On en donnera l'analyse complète.*

1 Position du problème

On a souvent besoin d'utiliser un ensemble trié d'éléments
- pour déterminer le rang d'un élément,
- pour calculer la médiane,
- pour utiliser la recherche rapide d'un élément,
- pour sélectionner selon un critère,
- pour maintenir une liste de priorités,
- pour déterminer les résultats aberrants,
- ...

En pratique on se donne une collection d'éléments, en python ce sera une liste et on veut obtenir une liste triée.
On commence par préciser les termes ci-dessus.

1.1 Relation d'ordre

On doit toujours comparer deux éléments :
l'ensemble des éléments est muni d'une relation d'ordre total.
- Ce peut être des nombres ou des chaînes de caractères, dans ce cas la relation d'ordre est définie dans Python par les relations `<`, `<=`, `>`, `>=`.

- Dans des cas plus généraux on devra définir une fonction à résultat booléen : par exemple `plusGrand(a,b)` qui devra renvoyer `True` si `a` est strictement supérieur à `b` pour la relation de comparaison et `False` si `a` est inférieur ou égal à `b`.
  On est dans ce cas, par exemple, dans le cas de listes ou de tuples dont une des composantes est un nombre, la clé. Les clés servent à la comparaison.

La complexité sera calculée en comptant le nombre de comparaisons.

Exemple

Dans le tableau ci-dessous on veut trier en fonction de la note du DS4.

nom	DS1	DS2	DS3	DS4
André	10	12	15	9
Bernard	8	16	7	14
Céline	13	12	12	10
Dominique	8	6	9	8
Éric	9	8	17	10
François	14	13	12	15

Les lignes sont représentées par des liste : `["André", 10, 12, 15, 9]`.
La relation d'ordre est définie par la fonction

```
def plusGrand(a,b):
    """Entrée : 2 éléments à comparer
       Sortie : True ssi a > b"""
    return a[4] > b[4]
```

On obtient

nom	DS1	DS2	DS3	DS4
Dominique	8	6	9	8
André	10	12	15	9
Céline	13	12	12	10
Éric	9	8	17	10
Bernard	8	16	7	14
François	14	13	12	15

Dans la suite, pour simplifier, nous illustrerons les tris avec des listes de nombres en employant les comparateurs internes de Python.

1.2 Résultat attendu

On veut une liste triée, en général par ordre croissant : l'élément d'indice i devra être plus inférieur ou égal à l'élément d'indice $i+1$.

Tri externe ou en place

Le résultat souhaité par un tri, "*on veut une liste triée*", est flou.

1. On peut souhaiter construire une nouvelle liste, triée, on parle alors de tri **externe**.
   L'avantage est qu'alors la liste initiale n'est pas détruite.
2. On peut souhaiter modifier la liste initiale, on trie **en place**.
   L'avantage est qu'on ne crée par une seconde liste ce qui peut être indispensable lorsque la liste à trier est très grande.

La plupart des tris que nous étudierons seront des tris en place.
Les fonctions de tris seront alors des fonctions sans instruction `return`.

Le docstring des tris sera donc, le plus souvent,

```
def tri(liste):
    """Entrée : une liste d'éléments comparables
       Sortie : les éléments sont permutés pour obtenir
                une liste triée par ordre croissant"""
    ...
```

Dans le cas des tris en place, un outil souvent utilisé est une fonction de permutation qui échange deux éléments dans une liste. Une écriture de base est

```
def echange(liste,i,j):
    """Entree : une liste et deux indices i, j
       Requis : 0 <= i,j < len(liste)
       Sortie : les termes i et j sont échangés"""
    temp = liste[i] # on sauvegarde liste[i]
    liste[i] = liste[j]
    liste[j] = temp
```

Cas d'égalité

Le résultat d'un tri est défini sans ambiguïté lorsque qu'il n'existe pas deux éléments distincts de la liste qui sont égaux pour la relation d'ordre
En cas d'égalité, comme dans l'exemple ci-dessus pour les lignes Céline et Éric, il n'y a pas unicité de l'ordre final car on pourrait intervertir les deux lignes.
On peut imposer que, comme dans l'exemple, les éléments indiscernables pour la relation d'ordre se retrouvent dans le même ordre à la fin.

Définition : Tri stable
Un tri est stable si des éléments égaux pour la comparaison sont dans le même ordre dans le résultat final qu'à l'origine.

2 Tri par insertion

2.1 Principe du tri

Le tri par sélection est celui que les joueurs de cartes peuvent employer : pour trier un tas de carte, on prend une carte à la fois et on l'insère dans les cartes déjà triées.
Dans le cadre du tri en place d'une liste, on trie les éléments les uns après les autres en laissant en place ceux qu'on n'a pas encore considérés
L'algorithme peut s'énoncer sous la forme

- Pour i allant de 0 à $n-1$ (n est la longueur de la liste)
- insérer le i-ième terme parmi les $i-1$ premiers en conservant le caractère trié.

On peut remarquer qu'on a, avant d'écrire le programme, un invariant de boucle :
pour chaque entier i on a

- les i premiers éléments de la liste (d'indices 0 à $i-1$) sont les i premiers éléments de la liste initiale placés dans l'ordre
- les derniers éléments (d'indices i à $n-1$) sont les éléments d'origine de la liste, à leur place.

On va représenter le principe le l'algorithme en montrant les étapes du tri à partir de la liste `[6, 9, 3, 8, 7, 2]`. Les listes ci-dessous sont représentées verticalement avec l'indice 0 en haut.

6	6	6	3	3	3	2
9	9	9	6	6	6	3
3	3	3	9	8	7	6
8	8	8	8	9	8	7
7	7	7	7	7	9	8
2	2	2	2	2	2	9

On peut donc écrire le programme suivant

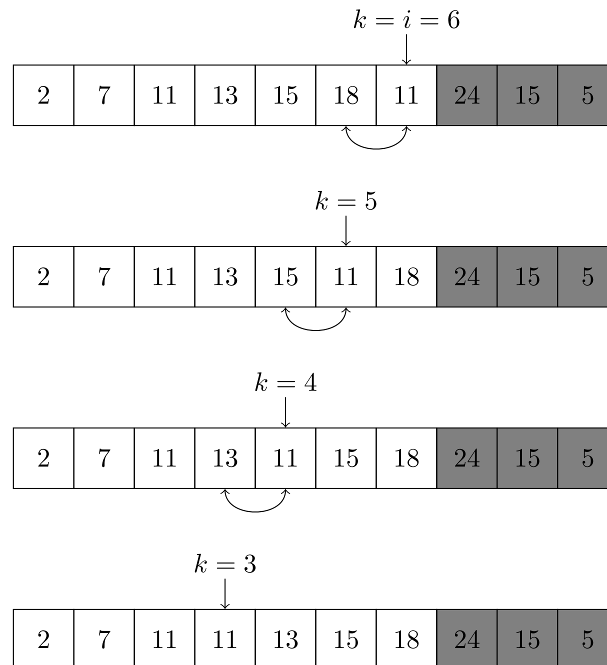
```
def triInsertion(liste):
    """Entrée : une liste d'éléments comparables
       Sortie : les éléments sont permutés pour obtenir
                une liste triée par ordre croissant"""
    n = len(liste)
    for i in range(n):
        placer(liste,i)
```

2.2 Insertion

Pour réaliser `placer(liste,i)` on peut itérer le procédé suivant en commençant par $k = i$.

1. Si $k = 0$ on a fini.
2. Si on a $k > 0$ et si `liste[k-1] > liste[k]` on échange les éléments des positions k et $k-1$ puis on diminue k de 1 (on décrémente k).
3. Si on a $k > 0$ et si `liste[k-1] <= liste[k]` on a fini.

On povait choisir de faire l'échange aussi en cas d'égalité mais alors le tri n'aurait pas été stable.
Exemple : exécution de `inserer(liste, 6)`
avec `liste = [2, 7, 11, 13, 15, 18, 11, 24, 15, 5]`.



En appliquant ces idées on peut écrire le programme

```
def placer(liste,i):
    """Entrées : une liste d'éléments compares par plusGrand
                 une entier i < len(liste)
       Requis  : la liste est triée de 0 a i-1
       Sortie  : la liste est triée entre 0 et i"""
    k = i
    while k > 0 and plusGrand(liste[k-1],liste[k]):
        echange(liste, k, k-1)
        k = k - 1
```

On a utilisé une particularité courante des langages dans l'évaluation des expressions booléennes : si la première condition est fausse alors la seconde (celle qui suit le `and`) n'est pas évaluée. Pour $k = 0$ on n'aura donc pas d'erreur alors que `liste[0-1]` n'existe pas [1]

1. Sauf en python où `liste[-1]` a un sens, c'est le dernier élément.

2.3 Analyse de l'insertion

1. La fonction `inserer` termine car l'entier `k` décroît strictement à chaque passage de la boucle donc finit par être nul si la condition `plusGrand(liste[k-1],liste[k])` est vraie à chaque passage.
2. On note `liste0` la valeur de la liste initiale, envoyée en paramètre de la fonction `inserer`.
   On rappelle que `liste0[0:i]` est supposée triée.
   On note `cle = liste0[i]`
   Un invariant de boucle possible, $P(k)$, est formé des propriétés

   $$\begin{cases} P_1(k) : \texttt{liste[0:k] = liste0[0:k]} \\ P_2(k) : \texttt{liste[k] = liste0[i]} \\ P_3(k) : \texttt{liste[k+1:i+1] = liste0[k:i]} \\ P_4(k) : \texttt{liste[k:i+1]} \text{ est croissante.} \end{cases}$$

   On a, dans tous les cas, les termes d'indices au-delà de i inchangés.
   On note aussi que $P_1(k)$, $P_2(k)$ et $P_3(k)$ impliquent que `liste[0:i+1]` contient les mêmes éléments que `liste0[0:i+1]`.
   - Pour $k = i$, `liste` est égale à `liste0` donc 1) et 2) sont évidentes. 3) n'a pas d'objet (les listes extraites sont vides) et 4) est vraie trivialement car une liste réduite à un élément est toujours croissante : $P(k)$ est vérifiée.
   - On suppose que $P(k)$ est vérifiée avec $k > 0$ et `plusGrand(liste[k-1],liste[k])`.
     On note `liste'` la valeur de la liste après la boucle.
     On effectue un échange donc `liste'[k-1] = liste[k]` et `liste'[k] = liste[k-1]` et les autres termes sont égaux.
     (a) `liste'[0:k-1] = liste[0:k-1] = liste0[0:k-1]` d'après $P_1(k)$: $P_1(k-1)$ est vérifiée.
     (b) `liste'[k-1] = liste[k] = liste0[i]` d'après $P_2(k)$: $P_2(k-1)$ est vérifiée.
     (c) `liste'[k:i+1] = liste'[k] + liste'[k+1:i+1] = liste[k-1]+liste[k+1:i+1]`
     or `liste[k-1] = liste0[k-1]` d'après $P_1(k)$
     et `liste[k+1:i+1] = liste0[k:i]` d'après $P_1(k)$
     donc `liste'[k:i+1] = liste0[k-1] + liste0[k:i] = liste0[k-1:i]` :
     $P_3(k-1)$ est vérifiée.
     (d) `liste'[k-1:i+1] = liste'[k-1] + liste'[k] + liste'[k+1:i+1]` d'où `liste'[k-1:i+1] = list`
     puis. `liste'[k-1:i+1] = liste[k] + liste0[k-1] + liste0[k:i]` puis.
     Or `liste[k-1]` est plus grand que `liste[k-1] = liste0[k-1]`
     et `liste0[0:i]` est croissante donc `liste0[k-1] + liste0[k:i]` est croissante.
     On en déduit que `liste'[k-1:i+1]` est croissante : $P_4(k-1)$ est vérifiée.
   - Si on a $k = 0$, $P_4(0)$ montre qu'on a `liste[0:i+1]` croissante et $P_2(0)$ et $P_3(0)$.
     Si on a $k > 0$ et `plusGrand(liste[k-1],liste[k])` non vérifié alors le dernier élément de `liste[0:k]` est plus peit que le prmier élément de `liste[k:i+1]` et ces deux listes extraites sont croissantes d'après $P_4(k)$, $P_1(k)$ et la croissance de `liste0[0;i]` donc `liste[0:i+1]` est croissante.

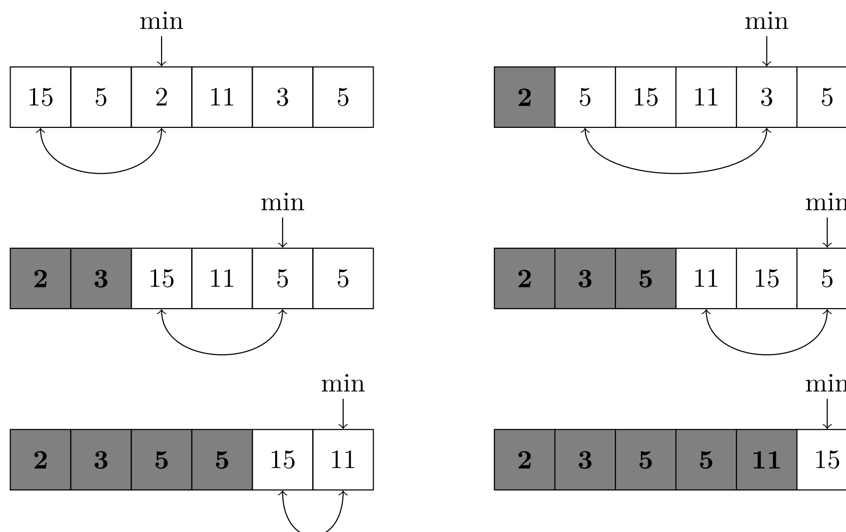
   On a donc démontré qu'à la fin de l'exécution de `placer(i, liste)` avec une liste dont les i premiers éléments sont triés on aboutit à une liste dont les $i+1$ premiers éléments sont triés et qui contient les mêmes éléments.
3. On éffectue une comparaison pour chaque k de i à k_0 avec $1 \leqslant k_0 \leqslant i$ donc le nombre de comparaisons est compris entre 1 et i.

2.4 Analyse du tri

1. La fonction `triInsertion` fait appel $n-1$ fois à la fonction `inserer` qui termine donc le tri termine.
2. Un invariant de boucle possible est que, pour chaque i, les i premiers termes forment une liste croissante et que la liste contient les éléments du départ.
   (a) C'est immédiat à l'initialisation : la liste est inchangée.
   (b) La conservation de cette propriété lorsqu'on exécute les instructions pour i découle de la preuve de `placer`.
   (c) Pour $i = n$ le résultat indique que la liste est triée et contient les éléments initiaux.
   Le tri est prouvé.
3. Le nombre de comparaisons est majoré par $\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$, c'est un $\mathcal{O}n^2$.
   D'après l'étude de la fonction `inserer` la complexité maximale est atteinte lorsque l'élément d'indice i est inférieur (strictement) à tous ceux qui le précèdent, pour chaque i : c'est le cas pour une liste strictement décroissante au départ.

3 Tri par sélection

Dans le tri par insertion on créait la liste triée en prenant un par un, comme ils venaient, les élément non encore triés. À chaque étape on doit alors modifier la partie triée.
On peut aussi construire directement la liste triée en mettant à leur place les éléments par ordre croissant. On sélectionne alors les élements : on parle de tri par sélection.



À chaque étape on a déterminé le minimum parmi les termes restants et on l'a placé.
On a donc besoin du fonction qui recherche l'indice du minimum à partir d'un rang.

```
def indMinDepuis(liste,i):
    """Entrées : une liste d'éléments compares par plusGrand
                 une entier i < len(liste)
       Sortie  : le premier indice en lequel la liste extraite
                 entre i et la fin atteint son minimum"""
```

Le tri par sélection est étudié sous forme d'exercices.

4 Exercices

4.1 Tri par sélection

Exercice 1 — Stabilité
Pourquoi choisit-on le premier indice dans la fonction `indMinDepuis` ?

Exercice 2 — Sélection
Écrire une fonction `indMinDepuis`

Exercice 3 — Tri par sélection
Écrire une fonction `triSelection` en utilisant `indMinDepuis`.

Exercice 4 — Terminaison
Prouver que la fonction de tri par sélection termine.

Exercice 5 — Complexité
Quelle est la complexité de `triSelection` en nombre de comparaisons
pour les listes de longueur n ?

Exercice 6 — Preuve de la sélection
Donner la preuve de `indMinDepuis`.

Exercice 7 — Preuve du tri par sélection
Donner la preuve de `triSelection`.
On voit que le tri par sélection est semblable au tri par insertion par sa complexité quadratique.
En moyenne le tri par insertion est cependant 2 fois plus rapide.

Exercice 8 — Nombre maximal de comparaisons
Montrer que le nombre de comparaisons **distinctes** à effectuer pour trier une liste de taille n est au plus $\frac{1}{2}n(n-1)$.

4.2 Tri par insertion

Exercice 9 — Simplification de l'insertion
Dans la fonction `placer` du tri par insertion on fait 3 affectations lors de chaque échange.
Un des éléments qui intervient dans l'échange est toujours le même.
Proposer une fonction qui demande moins d'affectations.

Exercice 10 — Complexité minimale du tri par insertion
Quel est le nombre minimal de comparaisons que fait le tri par insertion ?
Dans quel cas ce minimum est-il atteint ?

Exercice 11 — Complexité moyenne du tri par insertion
On commence par évaluer la complexité moyenne en nombre de comparaisons de `inserer(liste, i)` avec $i \geqslant 1$: $\overline{C}_{ins}(i)$. On fait les hypothèses :

1. la liste est triée entre les indices 0 et $i-1$,
2. (*) les positions de `liste[i]` après l'insertion sont équiprobables.

En comptant le nombre de comparaisons pour chaque position prouver que le nombre moyen de comparaisons est $\overline{C}_{ins}(i) = \dfrac{i}{2} + 1 - \dfrac{1}{i+1}$.

Pour calculer la complexité moyenne du tri par insertion en nombre de comparaisons on considère toutes les permutations d'une liste de n éléments **distincts** et on admet qu'alors l'hypothèse (*) est vérifiée pour chaque i. Prouver que le nombre moyen de comparaisons pour le tri par insertion des listes de taille n est $\overline{C}(n) = \dfrac{n^2+3n}{4} - \sum_{k=1}^{n} \dfrac{1}{k}$.

4.3 Autres tris

Exercice 12 — Tri à bulles, CCP 2015
On considère la fonction

```
from copy import copy
def trier(p):
    t = copy(p)
    for i in range(len(t)):          # boucle externe
        for j in range(len(t)-i-1): # boucle interne
            if (t[j+1]<t[j]):
                t[j], t[j+1] = t[j+1], t[j]
    return t
```

1. Montrer que la fonction termine.
2. Combien de comparaison sont effectuées dans le cas d'une liste de taille n ?
3. Prouver que la fonction renvoie une liste avec les mêmes éléments que la liste initiale en les plaçant dans l'ordre croissant. On pourra, en plus de la conservation de l'ensemble des valeurs, utiliser les invariants de boucles suivants.
   $P_1(i)$ pour la boucle externe :
   $\forall k \in \{n-i-1, \ldots, n-2\}$, `t[k] <= t[k+1]` et `t[k] <= t[n-i]`.
   $P_2(i,j)$ pour la boucle interne : $P_1(i)$ et $\forall k \in \{0, 1, \ldots, j\}$, `t[k] <= t[j]`.

Exercice 13 — Un tri très rapide ?
On suppose que l'on veut trier une liste d'éléments dont les clés sont des entiers **distincts** et appartenant à un ensemble $\{0, 1, 2, \ldots, N-1\}$.
Proposer une méthode de tri sans comparaison qui utilise une liste de taille N.
La clé de tri est fournie par une fonction `cle`.
Déterminer sa complexité : pourquoi ce tri n'est-il pas utilisé ?

Exercice 14 — Cas particulier
L'idée de l'exercice précédent peut cependant être utilisée dans un cas particulier.
On suppose que l'on veut trier une liste d'éléments dont les clés sont des entiers appartenant à un ensemble $\{0, 1, 2, \ldots, N-1\}$. L'entier N est "petit" par rapport à la taille de la liste mais plusieurs élément peuvent avoir la même clé. C'est ce qui se produit, par exemple, quand on veut trier une liste d'étudiants en fonction de la note d'un devoir.
Proposer une méthode de tri sans comparaison qui utilise une liste de listes, de taille N.

Exercice 15 — Tri d'une liste de 4 éléments
Pour trier une liste de 4 éléments le tri par insertion peut demander jusqu'à
$\frac{4(4-1)}{2} = 6$ comparaisons (le tri par sélection en demande toujours 6).
Donner une fonction qui trie en place une liste de longueur 4 en effectuant au plus 5 comparaisons (et au plus 5 appels de `echange`).

5 Solutions

Solution de l'exercice 1 -
On choisit la première apparition du minimum pour que le tri soit stable.

Solution de l'exercice 2 -

```
def indMinDepuis(liste,i):
    """Entrées : une liste d'éléments comparés par plusGrand
                 une entier i < len(liste)
       Sortie  : le premier indice en lequel la liste extraite
                 entre i et la fin atteint son minimum"""
    n = len(liste)
    indMin = i
    for j in range(i+1,n):
        if plusGrand(liste(indMin),liste(j)):
            indMin = j
    return indMin
```

Solution de l'exercice 3 -

```
def triSelection(liste):
    """Entrée : une liste d'éléments comparables par plusGrand
       Sortie : les éléments sont permutés pour obtenir
                une liste triée par ordre croissant"""
    n = len(liste)
    for i in range(n):
        k = indMinDepuis(liste,i)
        echange(liste,i,k)
```

On pourrait arrêter la boucle à $i = n-2$ car le dernier élément est à sa place quand on a placé les $n-1$ éléments les plus petits au début de la liste.

Solution de l'exercice 4 - La fonction `indMinDepuis` utilise une boucle `for` donc termine.
La fonction `triSelection` utilise `indMinDepuis` dans une boucle `for` donc termine.

Solution de l'exercice 5 - Dans `indMinDepuis` on effectue une comparaison pour chaque indice de $i+1$ à $n-1$: le nombre de comparaisons est $n-i-1$.
Dans `triSelection(liste` on appelle `indMinDepuis(i, liste)` pour chaque i de 0 à $n-1$: il y a donc $\displaystyle\sum_{i=0}^{n-1} n-i-1 = \sum_{p=0}^{n-1} p = \frac{n(n-1)}{2}$ comparaisons.
La complexité est la même pour toute liste, c'est la même que la complexité maximale de `triInsertion`.

Solution de l'exercice 6 - Un invariant de boucle est
$P(j)$: `indMin` est le premier indice en lequel la liste atteint son minimum entre i et j.

Solution de l'exercice 7 - Un invariant de boucle est $P(k)$, formé de 3 propriétés :

$$\begin{cases} P_1(k) : \text{la liste est une permutation de la liste initiale,} \\ P_2(k) : \text{la liste est triée entre les indices 0 et } i-1 \\ P_3(k) : \texttt{liste[i-1]} \text{ est plus petit (ou égal) que } \texttt{liste[j]} \text{ pour tout } j \geqslant i. \end{cases}$$

Solution de l'exercice 8 - Le nombre de paires d'éléments dans un ensemble de n éléments est $\binom{n}{2} = \frac{1}{2}n(n-1)$: c'est le nombre maximal de comparaisons distinctes que l'on peut faire.

Solution de l'exercice 9 - Il suffit de copier les grands éléments à leur droite et, à la fin, de placer l'élément à insérer à sa place. Il ne faut pas oublier de mettre cet élément dans une variable.

```
def inserer(liste,i):
    """Entrées : une liste d'éléments comparables par
       plusGrand
                 une entier i < len(liste)
       Requis  : la liste est triée de 0 a i-1
       Sortie  : la liste est triée entre 0 et i"""
    cle = liste[i]
    k = i
    while k > 0 and plusGrand(liste[k-1],cle):
        liste[k] = liste[k-1]
        k = k - 1
    liste[k] = cle
```

On fait $p+1$ affectations au lieu de $3p$ si la boucle est effectuée p fois.

Solution de l'exercice 10 - Le nombre de comparaisons est majoré par $\sum_{i=1}^{n-1} 1 = n-1$.

Cette complexité minimale est atteinte lorsque l'élément d'indice i est supérieur à tous ceux qui le précèdent, c'est-à-dire lorsque la liste est croissante au départ.

Solution de l'exercice 11 - On note k_0 la position où aboutit `liste[i]`.
Pour arriver à k on doit échanger `liste[k]` et `liste[k-1]` pour $k \in \{i, i-1, \dots, k_0+1\}$: à chaque fois on a fait une comparaison. Si k_0 est non nul on doit faire une dernière comparaison pour finir la boucle. On fait $i+1-k_0$ comparaisons pour $k > 0$ et i comparaisons pour $k_0 = 0$. Chaque position à une probabilité de $\frac{1}{i+1}$ donc

$$\overline{C}_{ins}(i) = \frac{1}{i+1}\left(i + \sum_{k_0=1}^{i} i+1-k_0\right) = \frac{1}{i+1}\left(i + \sum_{p=1}^{i} p\right) = \frac{i+\frac{i(i+1)}{2}}{i+1} = \frac{i}{2} + \frac{i}{i+1}$$

Il suffit alors de sommer les moyennes

$$\overline{C}(n) = \sum_{i=1}^{n-1} \frac{i}{2} + 1 - \frac{1}{i+1} = \frac{1}{2}\sum_{i=1}^{n-1} i + n - 1 - \sum_{i=1}^{n-1} \frac{1}{i+1} = \frac{1}{2}\frac{n(n-1)}{2} + n - 1 - \sum_{p=2}^{n} \frac{1}{p}$$
$$= \frac{n(n-1)}{4} + n - \sum_{p=1}^{n} \frac{1}{p} = \frac{n^2}{4} + \frac{3n}{4} - \sum_{p=1}^{n} \frac{1}{p}$$

La complexité moyenne est donc de l'ordre de la moitié de la complexité maximale.

Solution de l'exercice 12 -

1. Il n'y a que des itérations par des boucles `for` donc la fonction termine.
2. Il y a 1 comparaison pour chaque passage donc le nombre est
   $$\sum_{i=0}^{n-1}\sum_{j=0}^{n-2-i} 1 = \sum_{i=0}^{n-1} n-1-i = \sum_{k=0}^{n-1} k = \frac{n(n-1)}{2}$$
3. $P_1(0)$ n'a pas d'objet car on a $n-1 > n-2$ et `t[n]` n'existe pas, elle est donc valide.
   Si $P_1(i)$ est valide pour $i < n-1$ on rentre dans la boucle interne avec $j = 0$.
   - $P_2(i, 0)$ demande simplement `t[0] <= t[0]`.
   - Si $P_2(i, j)$ est vérifiée avec $j < n-i-2$ et `t[j+1] < t[j]` alors on échange les termes d'indices j et j+1. On note `t'` la nouvelle liste obtenue. On a `t'[j] = t[j+1] < t[j] = t'[j+1]` et `t'[k] = t[k] <= t[j] = t'[j+1]` pour $k < j$. Comme $P_1(i)$ reste vraie on a $P_2(i, j+1)$ est valide.
   - Si `t[j+1] >= t[j]` alors la liste est inchangée. On a `t[j] <= t[j+1]` et `t[k] <= t[j] <= t[j+1]` pour $k < j$ donc $P_2(i, j+1)$ est valide.

- $P_2(i, n-i-1)$ signifie qu'on a, à la position $n-i-1$, un majorant des éléments précédents qui est majoré par `t[n-i]` : cela implique, avec $P_1(i)$ la propriété $P_1(i+1)$.

$P_1(n-1)$ prouve alors que la liste est triée (il ne se passe rien pour $i = n-1$).

Solution de l'exercice 13 - On crée une liste de taille N avec la valeur -1 (par exemple), on place l'élément i à la place i puis on lit les éléments distincts de -1 dans l'ordre. C'est tri qui renvoie une nouvelle liste.

```
def tri(liste, N):
    """Entrées : une liste d'entiers distincts
                 compris entre 0 et N-1
       Sortie  : la liste triée des éléments"""
    casier= [-1]*N
    n = len(liste)
    for i in range(n):
        k = cle(liste[i])
        casier[k] = liste[i]
    sortie = []
    for i in range(N):
        if casier[i] != -1:
            sortie.append(casier[i])
    return sortie
```

On ne fait aucune comparaison, on fait $n+N$ lectures et $2n$ écritures.
Si N n'est pas trop grand (majoré par $10n$ par exemple) on obtient un tri très rapide car linéaire.
En général N est trop grand pour qu'il soit raisonnable de créer une liste de taille N, c'est ce qui rend le tri peu utilisable.
Un cas particulier et caricatural est celui où on veut trier une permutation des n premier entiers : il suffit de renvoyer `list(range(n))` sans jamais lire la liste passée en paramètre.

Solution de l'exercice 14 - Une liste de liste vides ne peut pas être créée par `[]*N`.

```
def tri(liste, N):
    """Entrées : une liste d'entiers distincts
                 compris entre 0 et N-1
       Sortie  : la liste triée des éléments"""
    casier= [[] for i in range(N)]
    for item in liste:
        k = cle(item)
        casier[k].append(item)
    sortie = []
    for paquet in casier:
        for item in paquet:
            sortie.append(item)
    return sortie
```

On fait $2n$ utilisations de `append` et $N+2n$ lectures.

Solution de l'exercice 15 -

```
def tri4(liste):
    """Entrée : une liste de 4 éléments
       Sortie : la liste est triée"""
    if plusGrand(liste[0],liste[2]):
        echange(0,2,liste)
    if plusGrand(liste[1],liste[3]):
        echange(1,3,liste)
    if plusGrand(liste[0],liste[1]):
        echange(0,1,liste)
    if plusGrand(liste[2],liste[3]):
        echange(2,3,liste)
    if plusGrand(liste[1],liste[2]):
        echange(1,2,liste)
```

Chapitre III

RÉCURSIVITÉ

GNU : GNU's Not UNIX

Résumé

Dans ce chapitre nous allons définir un nouveau moyen de faire des calculs répétitifs : on appellera la fonction dans sa définition. C'est un outil très puissant qui permettra d'écrire simplement des algorithmes difficiles. La contrepartie est qu'il faudra gérer avec soin les conditions de terminaison.
Une fonction est **récursive** *si, dans sa définition, elle fait référence à elle-même. Les fonctions non récursives seront dites* **itératives**.
Une notion importante sera celle de **condition d'arrêt**.

1 Exemples

1.1 Factorielle

La factorielle de n, $n!$, est définie par $n! = \prod_{k=1}^{n} k$ avec la convention $0! = 1$. On peut traduire cette définition par une définition par récurrence : $n! = \begin{cases} 1 & \text{si } n = 0 \\ n.(n-1)! & \text{sinon} \end{cases}$.
Cela donne immédiatement l'algorithme

```
def factoriel(n):
  """Entrée : un entier positif n
     Sortie : la factorielle de n"""
    if n==0 :
        return 1
    else :
        return n*factoriel(n-1)
```

Que fait la machine lors de l'appel `factoriel(3)` ?

- Elle stocke la formule `3*factoriel(2)`.
- L'appel `factoriel(2)` remplace `factoriel(2)` par `2*factoriel(1)`.
- L'appel `factoriel(1)` remplace `factoriel(1)` par `1*factoriel(0)`.
- L'appel `factoriel(0)` retourne 1 ce qui permet de calculer $1 * 1$ pour `factoriel(1)`
- Les calcul stockés donnent alors $2 * 1$ pour `factoriel(2)` puis $3 * 2$ pour `factoriel(3)`.

Sur cet exemple, on voit que la machine doit stocker en mémoire un ensemble de formules. Nous verrons dans la suite la structure de **`pile`** qui représente ce stockage.
L'appel à `factoriel(0)` signe la fin de la construction de cette pile. Ensuite elle est parcourue en sens inverse pour permettre les calculs numériques. On comprend l'étymologie du mot récursivité qui vient de *recurrere* en latin qui signifie *revenir en arrière*.

S'il n'existait pas de condition d'arrêt l'empilement continuerait sans fin, il y aurait saturation de la mémoire : on parle de *stack overflow* en anglais. Dans la pratique Python bloque la pile à une longueur de 1000.

1.2 Tours de Hanoï

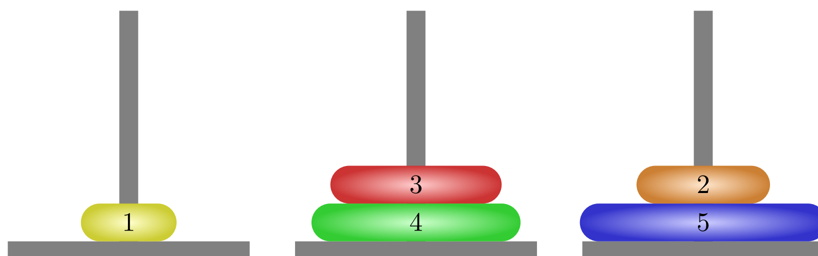
Le problème des tours de Hanoï est un jeu de réflexion imaginé par le mathématicien français Édouard Lucas consistant à déplacer des disques de diamètres différents d'une tour de "départ" à une tour d'"arrivée" en passant par une tour "intermédiaire" tout en respectant les règles suivantes :

- on ne peut déplacer plus d'un disque à la fois,
- on ne peut placer un disque que sur un autre disque plus grand que lui ou sur un emplacement vide.

Au départ les disques sont placés en respectant la seconde règle sur la tout de départ.
On part de :

On veut arriver à

Une position intermédiaire possible serait



Dans cette dernière position il y a 3 déplacements possibles :

- déplacer le disque 1 au-dessus du disque 3,
- déplacer le disque 1 au-dessus du disque 2,
- déplacer le disque 2 au-dessus du disque 3.

La résolution est en fait simple à énoncer en renonçant à écrire la stratégie de manière précise : pour déplacer n disques de la tour A la tour B il suffit

- de ne rien faire si $n = 0$
- de déplacer les $n-1$ disques supérieurs de la tour A vers la tour C,
  puis de déplacer le disque restant vers la tour B
  puis enfin de déplacer les $n-1$ disques supérieurs de la tour C vers la tour B.

On voit apparaître naturellement un algorithme récursif.

```
def hanoi(n,ch1,ch2,ch3):
    """Entrée : un entier et 3 noms de tours
       Sortie : les mouvements à faire pour déplacer
                les disques depuis la tour nommée ch1
                vers la tour nommée ch2"""
    if n != 0:
        hanoi(n-1,ch1,ch3,ch2)
        print('Déplacer le disque supérieur de {} vers {}'.
            format(ch1,ch2))
        hanoi(n-1,ch3,ch2,ch1)
```

On obtient alors un mode d'emploi.

```
Déplacer le disque supérieur de A vers C
Déplacer le disque supérieur de A vers B
Déplacer le disque supérieur de C vers B
Déplacer le disque supérieur de A vers C
Déplacer le disque supérieur de B vers A
Déplacer le disque supérieur de B vers C
Déplacer le disque supérieur de A vers C
Déplacer le disque supérieur de A vers B
Déplacer le disque supérieur de C vers B
Déplacer le disque supérieur de C vers A
Déplacer le disque supérieur de B vers A
Déplacer le disque supérieur de C vers B
Déplacer le disque supérieur de A vers C
Déplacer le disque supérieur de A vers B
Déplacer le disque supérieur de C vers B
```

On voit ici le caractère un peu magique de la récursivité : on dit très simplement les choses et le programme produit un résultat compliqué. Contrairement à l'exemple suivant, la récursivité ici est tout-à-fait naturelle ; il est difficile d'écrire un programme non récursif.

Complexité
On va estimer la complexité en comptant le nombre, noté $C(n)$, d'instructions `print` que le programme effectue, on note $C(n)$. Ici encore on va avoir une récurrence.
Si $n = 0$ on a $C(0) = 0$.
Quand on appelle la fonction pour n disques on effectue

- $C(n-1)$ instructions `print` dans `hanoi(n-1,ch1,ch3,ch2)`
- 1 instruction `print` dans `print('Déplacer le disque supérieur de',ch1,'vers',ch2)`
- $C(n-1)$ instructions `print` dans `hanoi(n-1,ch3,ch2,ch1)`.
- On a donc $C(n) = 2C(n-1) + 1$ pour $n \geqslant 1$.

$\big(C(n)\big)$ vérifie donc une relation arithmético-géométrique.
Le point fixe est -1 donc $u_n = C(n) - (-1)$ vérifie $u_n = 2u_{n-1}$ avec $u_0 = C(0) + 1 = 1$ d'où $u_n = 2^n$ puis $C(n) = 2^n - 1$. Nos 5 petites lignes de code ont créé un monstre de complexité !

1.3 Transformation d'un programme

Considérons le programme classique de calcul de la somme des termes d'une liste

```
def somme(liste):
    """Entrée : une liste de nombres
       Sortie : la somme des termes de la liste"""
    resultat = 0
    n = len(liste)
    for i in range(n):
        resultat = resultat + liste[i]
    return resultat
```

On a défini un invariant de boucle : à chaque valeur de i au début de la boucle `resultat` vaut la somme de i premiers termes. En fait cet invariant définit la somme par récurrence avec, comme toutes les récurrences, 2 assertions.
Comment initialiser ? Ici une somme de 0 termes vaut 0.
Comment avancer ? $\displaystyle\sum_{k=0}^{i} u_k = u_i + \sum_{k=0}^{i-1} u_k$.

La récursivité consiste à traduire ces propriétés en un algorithme :

- une somme vide vaut 0,
- la somme de n termes d'une liste est l'addition du dernier terme à la somme des $n-1$ premiers.

```
def somme(liste):
    """Entrée : une liste de nombres
       Sortie : la somme des termes de la liste"""
    n = len(liste)
    if n == 0:
        return 0
    else:
        return  liste[n-1] + somme(liste[:n-1])
```

Remarque : dans le programme ci-dessus on fait bien n additions pour calculer une somme : on n'a pas compliqué le calcul. Cependant à chaque appel on invoque `liste[:n-1]` qui effectue une copie de la liste, cela demande $n-1$ opérations de copie de valeurs. Quand on fait le bilan on arrive à $\frac{n(n-1)}{2}$ copies. Ce temps de recopie finit par surpasser le temps de calcul des sommes et le programme sera lent pour des grandes valeurs de n.
Cette difficulté doit inciter à la prudence : il est inutile d'écrire une fonction récursive si on sait écrire une fonction simplement sans récursivité.

2 Analyse des algorithmes récursifs

2.1 Emploi de récurrences

L'outil principal qui est utilisé dans l'analyse de fonctions récursive est la récurrence.

Terminaison L'expérience montre qu'il est très facile d'écrire un algorithme récursif qui ne termine pas. En effet il peut se produire que la fonction fasse indéfiniment appel à elle même. Un cas caricatural est la fonction suivante.

```
def f(x):
  return f(x+1)
```

Pour qu'un algorithme récursif termine il est donc indispensable qu'il existe une **condition d'arrêt**, c'est-à-dire une condition qui, si elle est réalisée, fait exécuter des instructions qui ne font pas appel (récursivement) à la fonction. Il est recommandé de commencer par ce cas dans la rédaction de l'algorithme.
Il faut de plus que l'on soit certain que toute suite d'appels à la fonction finit par aboutir à un cas où cette condition est vérifiée.
On pourra souvent mettre en évidence un paramètre (ou d'une expression des paramètres) qui ne prend que des valeurs entières positives. La preuve de la terminaison se fera alors par récurrence sur ce paramètre.

Correction Les algorithmes récursifs sont souvent bien adaptés pour être prouvés par récurrence. Parfois même ils sont la traduction d'une définition récursive qui en fournit alors directement la preuve.

Complexité temporelle La complexité vérifiera souvent une relation de récurrence, parfois sous la forme d'une inégalité. Ici encore on aura souvent besoin de faire une démonstration par récurrence.

2.2 Exemples

Somme des termes d'une liste On montre par récurrence sur n la propriété $\mathcal{P}(n)$:
pour tout liste de taille n la fonction renvoie la somme des termes de la liste en effectuant n additions.

Tours de Hanoï On montre par récurrence sur n la propriété $\mathcal{P}(n)$:
pour tout liste de taille n la fonction affiche les instructions qui permettent de déplacer n disques en respectant les règles.
La complexité a été calculée, par récurrence.

Factorielle On montre par récurrence sur n la propriété $\mathcal{P}(n)$:
pour tout n la fonction renvoie $n!$ en effectuant n multiplications.

Coefficients binomiaux Pour $0 \leqslant p \leqslant n$, $\binom{n}{p} = \frac{n!}{p!(n-p)!}$.

Un résultat mathématique classique est que $\binom{n}{p} = \frac{n}{p}.\binom{n}{p}n-1p-1$ pour $p \geqslant 1$.

En ajoutant la condition d'arrêt pour $p = 0$, $\binom{n}{0} = 1$, on obtient l'algorithme

```
def binomial(n,p):
  """Entrée : deux entiers positifs
     Requis : 0 <= p <= n
     Sortie : p parmi n"""
    if p == 0:
        return 1
    else :
        return n*binomial(n-1,p-1)//p
```

On peut remarquer que l'on ne fait que des calculs sur les entiers.

- Le programme renvoie un résultat pour $p = 0$ et pour tout n.

S'il renvoie un résultat pour p et pour tout n tel que $n \geqslant p$ alors `binomial(n, p+1)` avec $n \geqslant p+1$ fait appel à `binomial(n - 1, p)` qui fournit un résultat $(n-1 \geqslant p)$ et effectue 2 opérations pour renvoyer un résultat
Par récurrence sur p on voit que le programme renvoie un résultat pour tout p et pour tout $n \geqslant p$.

- Le programme renvoie 1 pour $p = 0$ et on a bien $\binom{n}{0} = 1$
  Si `binom(n p)` renvoie $\binom{n}{p}$ pour tout $n \geqslant p$ alors `binomial(n, p+1)` avec $n \geqslant p+1$ renvoie $\frac{n}{p+1}.\binom{n-1}{p} = \binom{n}{p+1}$.
  Par récurrence sur p on voit que le programme renvoie $\binom{n}{p}$ pour tous $n \geqslant p$.
- On a vu que `binomial(n, p)` effectuait 2 opérations de plus que `binomial(n-1, p-1)` donc $2p$ opérations de plus que `binomial(n-p, 0)` : la complexité est $2p$.

3 Avantages et inconvénients

Les exemples ont permis de mettre en évidence des avantages de la programmation récursive par rapport à la programmation itérative.

- Le code est facile à écrire et à lire en général.
- La définition récurrente mathématique trouve immédiatement sa traduction informatique.
- Pour certaines structures de données, celles qui sont elles-mêmes définies récursivement, la récursivité permet de trouver des algorithmes simples et clairs.
- Les démonstrations de terminaison, correction et de complexité sont plus simples à écrire.

Un inconvénient a été entrevu : lors des différents appels à la fonction récursive le langage de programmation doit stocker les valeurs des calculs à faire. Il emploie pour cela une pile mais celle-ci a une taille limitée déterminée à l'avance. Lorsque cette mémoire est saturée le langage renvoie le message d'erreur "**stack overflow**".

Il peut apparaître un autre problème lorsqu'on traduit une définition par récurrence : il se peut que l'on multiplie les calculs sans s'en rendre compte.

Exemple : suite de Fibonacci

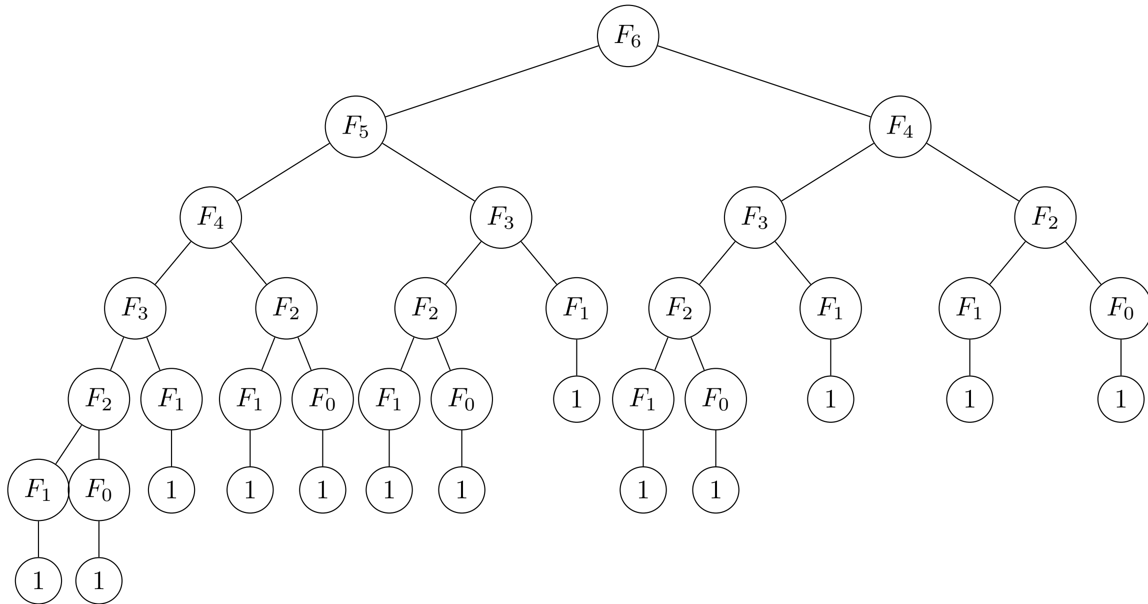
La suite de Fibonacci est définie par $\begin{cases} F_0 = 0 \\ F_1 = 1 \\ F_n = F_{n-1} + F_{n-2} & \text{pour } n \geqslant 2 \end{cases}$
La traduction récursive immédiate est

```
def fibo(n) :
  """Entrée : un entier positif n.
     Sortie : le n-ième nombre de Fibonacci"""
    if n <= 1 :
        return(1)
    else :
        return fibo(n-1)+fibo(n-2)
```

La terminaison et la correction de l'algorithme ne posent pas de problème.
Comptons le nombre d'additions, noté $C(n)$, pour calculer F_n.
On a facilement $C(0) = C(1) = 0$ et $C(n) = C(n-1) + 1 + C(n-2)$.
Si on retranche[1] -1 à $C(n)$ en posant $u_n = C(n) - (-1)$ on obtient $u_0 = 1$, $u_1 = 1$ et $u_n = C(n-1) + 1 + C(n-2) + 1 = u_{n-1} + u_{n-2}$ donc (u_n) vérifie la même récurrence que F_n avec les valeurs initiales décalées d'où $C(n) = u_n - 1 = F_{n+1} - 1$. On montre classiquement que $F_n = \dfrac{\alpha^{n+1} - \beta^{n+1}}{\sqrt{5}}$ avec $\alpha = \dfrac{1+\sqrt{5}}{2}$ et $\beta = \dfrac{1-\sqrt{5}}{2}$ d'où $C(n) = \mathcal{O}(\alpha^n)$.
Cette complexité exponentielle rend impossible le calcul pour n dépassant quelques dizaines.

1. L'équation $u_{n+2} = u_{n+1} + u_n$ admet (-1) comme suite constante solution.

Par exemple F_{40}, que l'on calcule à la main avec 39 additions, engendre $F_{41} - 1 \sim 4.10^8$ additions dans le programme ci-dessus ; le calcul prend 80 secondes sur un ordinateur (en 2016).
Que se passe-t-il ? Voici les calculs faits pour `fibo(6)`



On remarque qu'on a fait 5 fois le calcul de $F_2 = 1 + 1$ par exemple.
On peut aussi regarder le résultat du programme suivant qui détaille les différents calculs

```
def fiboDecompose(n):
  """Entrée : un entier positif n.
     Sortie : les calculs intermédiaires du calcul
              du n-ième nombre de Fibonacci"""
    print("fibo a ete appele avec n = {}".format(n))
    if n <= 1:
        print('Un cas de base est  atteint')
        return 1
    else:
        res = fiboDecompose(n-1)+fiboDecompose(n-2)
        print('Calcul de F({}) + F({}) = {}'.format(n-1,n-2,
           res))
```

Pour parvenir à une complexité raisonnable le plus simple ici est de revenir à un programme itératif.

```
def fibo(n):
  """Entrée : un entier positif n.
     Sortie : le n-ième nombre de Fibonacci"""
    terme=1   # On initialise le premier terme
    suivant=1 # Et le suivant
    for k in range(n):
        precedent = terme
        terme = suivant
        suivant = terme + precedent
    return terme
```

On pouvait utiliser le raccourci Python dans la boucle :

```
    for k in range(n):
        terme, suivant = suivant, terme + suivant
```

4 Exercices

Exercice 1 — Somme de liste
Donner un algorithme récursif de calcul de la somme des termes d'une liste qui soit de complexité linéaire.

Exercice 2 — Suites
Écrire en Python une fonction récursive qui calcule le terme d'indice n de la suite (u_n) dans les cas suivants. On précisera le cas de base et la récurrence.

1. La suite arithmétique de premier terme 1 et de raison 7.
2. La suite géométrique de premier terme 1 et de raison $\sqrt{2}$
3. $u_0 = 2$ et, pour $n \geqslant 1$, $u_n = 4u_{n-1} - 1$.
4. $u_1 = 4$ et, pour $n \geqslant 2$, $u_n = \sqrt{n + u_{n-1}}$.
5. $u_0 = 1$ et, pour $n \geqslant 1$, $u_n = \dfrac{1}{2}\left(u_{n-1} + \dfrac{2}{u_{n-1}}\right)$.

Exercice 3 — Logarithme entier
Le logarithme entier (ou discret) d'un entier $n \geqslant 1$ est l'entier p tel que $2^p \leqslant n < 2^{p+1}$.
Écrire une fonction récursive `logEntier(n)` qui le calcule.

Exercice 4 — Coefficients binomiaux
À l'aide de la relation $\binom{n}{p} = \binom{n-1}{p} + \binom{n-1}{p-1}$, écrire en Python une fonction récursive `binomAdd(n,p)` qui renvoie la valeur de $\binom{n}{p}$.
On donnera la complexité temporelle en fonction de n.
Si cette complexité n'est pas raisonnable, proposer une amélioration.

Exercice 5 — Exponentiation

1. Que calcule `mystere(a,n)` pour a réel a et n entier naturel ?

```
def mystere(a,n):
    """Entrée : un réel a et un entier positif n
       Sortie : ???"""
    if n==0:
        return 1
    else:
        return a*mystere(a,n-1)
```

2. Pour $k \in \mathbb{N}^*$ on remarque qu'on a $a^{2k} = (a^k)^2$ et $a^{2k+1} = (a^k)^2.a$.
   Utiliser ce résultat pour écrire une fonction récursive `puissance(a,n)` qui retourne a^n (a réel et n entier naturel). Combien de multiplications sont effectuées ?

4.1 Décomposition de Fibonacci

Exercice 6 — Suite
Écrire une fonction `suiteFibo(n)` *qui renvoie la liste des nombres de Fibonacci de* F_0 *à* F_n*.*

Une décomposition de Fibonacci d'un entier $n \geqslant 1$ est une suite $(F_{k_2}, F_{k_2}, \dots, F_{k_r})$ de nombres de Fibonacci telle que

$$\begin{cases} k_1 \geqslant 2 \\ k_{i+1} \geqslant k_i + 2 \text{ pour } 1 \leqslant i < r \\ n = F_{k_1} + F_{k_2} + \cdots + F_{k_r} \end{cases}$$

Par exemple $300 = 1 + 3 + 8 + 55 + 233 = F_2 + F_4 + F_6 + F_{10} + F_{13}$

On peut montrer que si $(F_{k_2}, F_{k_2}, \dots, F_{k_r})$ est une décomposition de Fibonacci de n alors $F_{k_r} \leqslant n < F_{k_r+1}$. On en déduit [2] que n admet au plus une décomposition de Fibonacci.

2. Par récurrence sur n.

Exercice 7 — Nombre de Fibonacci précédent
Écrire une fonction `precFibo(n)` *qui renvoie l'unique entier* $p \geqslant 2$ *tel que* $F_p \leqslant n < F_{p+1}$.

On peut montrer aussi que si p est la valeur de `precFibo(n)` alors une suite de Fibonacci de n peut être construite à partir de celle de $n - F_p$ en lui ajoutant F_p.
Ainsi une décomposition de Fibonacci existe toujours.

Exercice 8 — Calcul de la décomposition
Écrire une fonction `decFibo(n)` *qui renvoie la décomposition d'un entier* n *sous la forme d'une liste strictement croissante.*

4.2 Recherche par dichotomie

On suppose qu'on a une liste triée (d'entiers par exemple).
On veut savoir si un élément appartient à la liste.
Pour cela on teste au milieu de la liste :

- si on a trouvé, c'est fini,
- si la valeur au milieu est inférieure à la valeur recherchée alors on doit chercher dans la partie supérieure,
- sinon on doit chercher dans la partie inférieure.

L'algorithme est naturellement récursif si on considère une fonction qui recherche un élément dans une liste entre deux positions.

Exercice 9 — Algorithme récursif
Écrire une fonction récursive `appRec(x, liste, debut, fin)` *qui teste récursivement si un élément est présent dans une liste croissante entre les positions* `debut` *et* `fin` *(extrémités comprises).*
En déduire une `appartient(x, liste)` *qui teste l'appartenance d'un élément* x *à une liste triée.*

Exercice 10 — Terminaison
Prouver que la fonction `appRec` *termine.*
On pourra noter k *la valeur de* `fin - debut` *et procéder par récurrence généralisée sur* k.

Exercice 11 — Preuve
Prouver que la fonction `appRec` *donne le résultat annoncé par son docstring.*

Exercice 12 — Complexité
Déterminer le nombre maximal de comparaisons effectués par `appRec(liste, x, debut, fin)` *en fonction de* k*, la valeur de* `fin - debut`. *En déduire que la complexité de* `appartient(x,liste)` *en nombre de comparaisons est un* $\mathcal{O}\big(\log_2(n)\big)$ *où* n *est la longueur de la liste.*

4.3 Écriture récursive des tris

Les tris étudiés dans le chapitre précédent peuvent être écrits de manière récursive.
Un guide pour écrire les algorithmes est de partir des invariants de boucle.

1. Pour le tri par insertion un invariant de boucle est que, pour chaque i, la liste des i premiers termes est le tri des premiers termes de la liste initiale, le reste étant inchangé.
   On peut donc écrire une fonction récursive qui trie les i premiers termes.
   (a) il n'y a rien à faire si $i = 0$,
   (b) pour $i \geqslant 1$, on appele la fonction avec $i - 1$ puis on insère l'élément d'indice i

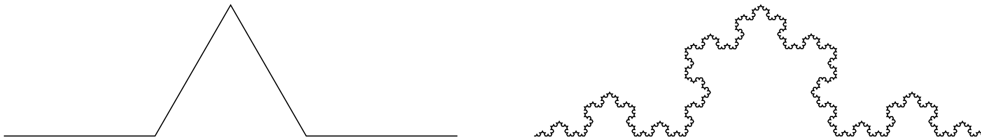
Exercice 13 — Tri par insertion récursif
Écrire une fonction récursive `triPartiel(liste, i)` *qui trie les* i *premiers éléments d'une liste en appliquant l'algorithme ci-dessus.*
En déduire une fonction de tri.
On utilisera une fonction `inserer`.

2. Pour le tri par sélection, un invariant de boucle est que, pour chaque i, les i premiers termes sont les i plus petits termes de la liste.

Exercice 14 — Tri par sélection récursif
Écrire une fonction récursive `triPremiers(liste, i)` *qui place les i plus petits éléments d'une liste triés en tête de liste.*
En déduire une fonction de tri.
On utilisera une fonction `indMinDepuis`*.*

4.4 Fractales

La fractale liée à un motif composé de segments adjacents est la courbe obtenue en remplaçant chaque segment par le motif réduit et en recommençant de manière récursive. La fractale mathématique est l'objet limite, nous allons en tracer une approximation en itérant n fois le procédé.
Par exemple si le motif de base de gauche est transformé en la fractale de droite, appelée parfois "flocon de Von Koch".



Le module `turtle` est un ensemble d'outils permettant de dessiner à l'aide d'instructions simples. Ses principales fonctions sont

- `reset()` : on efface tout et on recommence,
- `goto(x,y)` : aller à l'endroit de coordonnés x et y,
- `forward(distance)` : avancer d'une distance donnée,
- `backward(distance)` : reculer, :
- `up()` et `down()` : relever et abaisser le crayon,
- `color(couleur)` : Changer de couleur, ('red', 'blue' ...)
- `left(angle)` et typeright(angle) : tourner d'un angle donné (en degrés)
- `width(épaisseur)` : Choisir l'épaisseur du tracé
- `fill(1)` : Remplir un contour fermé à l'aide de la couleur sélectionnée,
  on termine la construction par `fill(0)`,
- `write(texte)` : écrire un texte.

Dans l'exemple ci-dessus on remplace un segment de longueur c, `forward(c)` ; par le programme

```
forward(c/3.0)
left(60)
forward(c/3.0)
right(120)
forward(c/3.0)
left(60)
forward(c/3.0)
```

Exercice 15 — Tracé
Écrire en Python une fonction récursive `motif(c,n)` *qui trace l'approximation de niveau n de la courbe définie ci-dessus.*
Essayer d'autres motifs.

5 Solutions

Solution de l'exercice 1 -

1. Ici le problème est de ne pas copier la liste ; il faut donc considérer la liste comme globale. Dans ce cas elle ne peut pas être paramètre de la fonction récursive. On va donc définir une fonction auxiliaire récursive dans laquelle on ajoute l'indice de fin de calcul comme paramètre.

```
def sommeJusquA(liste, k):
    """Entrées : une liste de nombres, un entier
       Requis  : 0 <= k <= len(liste)
       Sortie : la somme des k premiers termes de la liste"""
    if k == 0:
        return 0
    else:
        return  liste[k-1] + sommeJusquA(liste, k-1)

def somme(liste):
    """Entrée : une liste de nombres
       Sortie : la somme des termes de la liste"""
    n = len(liste)
    return sommeJusquA(liste, n)
```

2. On pouvait aussi inclure cette fonction auxiliaire **dans** la fonction principale, on n'a plus besoin de donner la liste comme paramètre car elle est une variable globale.

```
def somme(liste):
    """Entrée : une liste de nombres
       Sortie : la somme des termes de la liste"""
    n = len(liste)
    def sommeJusquA(k):
        if k == 0:
            return 0
        else:
            return  liste[k-1] + sommeJusquA(k-1)
    return sommeJusquA(n)
```

3. Il existe une construction (spécifique à python) qui permet de se passer d'une fonction auxiliaire. On introduit une variable optionnelle dont la valeur par défaut (ici -1) déclenche l'appel avec la valeur adéquate.

```
def somme(liste, long = -1):
    """Entrée : une liste de nombres
       Sortie : la somme des termes de la liste"""
    if longueur == -1:
        return somme(liste, len(liste))
    elif longueur == 0:
        return  0
    else:
        return  liste[-1] + somme(liste, longueur - 1)
```

4. On pouvait aussi séparer la liste rapidement avec la méthode pop. Cependant cela modifie la liste initiale : il faut donc travailler sur une copie.

```
def somme(liste, longueur = -1):
    """Entrée : une liste de nombres
       Sortie : la somme des termes de la liste"""
    liste1 = copie(liste) # à définir, par ex. copy.deepcopy
    def sommeRec(l):
        if l == []:
            return 0
        else:
            x = l.pop()
            return x + sommeRec(l)
    return sommeRec(liste1)
```

Un exemple de copie.

```
def copie(liste):
    return [x for x in liste]
```

Solution de l'exercice 2 -

1.
```
def u(n):
    if n == 0:
        return 1
    else:
        return 7 + u (n-1)
```

2.
```
def u(n):
    if n == 0:
        return 1
    else:
        return 2**0.5*u (n-1)
```

3.
```
def u(n):
    if n == 0:
        return 2
    else:
        return 4*u (n-1)-1
```

4.
```
def u(n):
    if n <= 1:
        return 4
    else:
        return (n+u(n-1))**0.5
```

5.
```
def u(n):
    if n <= 0:
        return 4
    else:
        v = u(n-1) # Ne pas demander 2 fois le même calcul
        return (v + 2/v)/2
```

Solution de l'exercice 3 -
Si on a $2^p \leqslant n < 2^{p+1}$ avec $p \geqslant 1$ alors on a $2^{p-1} \leqslant n//2 < 2^p$.
On peut donc déduire la valeur du logarithme discret de n à partir de celui de $n//2$.
Le cas d'arrêt est 1.

```
def logEntier(n)
    if n==1:
        return 0
    else:
        return 1 + logEntier(n//2)
```

Solution de l'exercice 4 -
Si on applique la définition on peut écrire

```
def binomAdd(n,p):
    if p < 0 or n < p:
        return 0
    elif p == 0 or p == n:
        return 1
    else:
        return binomAdd(n-1,p-1) + binomAdd(n-1,p)
```

Le programme termine car la valeur de n diminue de 1 à chaque appel.
Si on note $C(n,p)$ le nombre d'additions pour calculer `binomAdd(n,p)` on a
$C(n,n) = C(n,0) = 0$ et $C(n,p) = C(n-1,p-1) + C(n-1,p) + 1$.
On remarque que si on pose $C'(n,p) = C(n,p) + 1$ alors $C'(n,n) = C'(n,0) = 1$
$C'(n,p) = C'(n-1,p-1) + C'(n-1,p)$: on a donc $C'(n,p) = \binom{n}{p}$.
Pour p fixé on a ainsi $C(n,p) = \binom{n}{p} - 1 = \mathcal{O}(n^p)$ ce qui est peu efficace.
Par exemple `binomAdd(30,10)` demande 20 secondes de calcul.

Pour retrouver une complexité raisonnable il faut éviter de calculer plusieurs fois les mêmes résultats ; la fonction ci-dessus tombe dans ce piège.
On peut calculer la suite des $\binom{m}{q}$ pour q variant de 0 à m pour chaque m de 0 à n.

```
def suiteBinom(n,p):
    if n == 0:
        return [1]
    else:
        prec = suiteBinom(n-1)
        liste = [1]
        for i in range(1,n):
            liste.append(prec[k-1]+prec[k])
        liste.append(1)
        return liste
```

```
def binomAdd1(n,p):
    return suiteBinom(n,p)[p]
```

La complexité est majorée par np. `binomAdd1(30,10)` est immédiat (2.10^{-4} s, `binomAdd1(300,100)` demande 12 ms.

Solution de l'exercice 5 -

1. La fonction calcule a^n.
2. On utilise 2 fois a^k, cependant il ne faut le calculer qu'une fois : on place sa valeur dans une variable.

```
def puissance(a,n):
    """Entrée : un réel a et un entier positif n
```

```
        Sortie : a à la puissance n"""
    if n==0:
        return 1
    else:
        b = puissance(a, n//2)
        if n%2 == 0:
            return b*b
        else:
            return b*b*a
```

Si on note $C(n)$ le nombre de multiplication on a $C(n) = C(n//2)+1$ ou $C(n) = C(n//2)+2$ selon que n est pair ou impair avec $C(0) = 0$.
Si on a $2^p \leqslant n < 2^{p+1}$ avec $p \geqslant 1$ alors on a $2^{p-1} \leqslant n//2 < 2^p$.
On note $C'(p)$ un majorant de $C(n)$ pour $2^p \leqslant n < 2^{p+1}$: on voit qu'on peut choisir $C'(p) = 2 + C'(p-1)$. Comme on peut aussi choisir $C'(0) = C(1) = 2$ on en déduit, par récurrence, que $C'(p) = 2(p+1)$ convient.
Pour $2^p \leqslant n < 2^{p+1}$ on a $p \leqslant \log_2(n)\big)$ donc $C(n) \leqslant C'(p) = 2(p+1) \leqslant 2\big(\log_2(n)+1\big)$
d'où $C(n)$ est un $\mathcal{O}\big(\log_2(n)\big)$.

Solution de l'exercice 6 -

```
def suiteFibo(n):
    if n == 0:
        return [0]
    elif n == 1:
        return [0, 1]
    else:
        resultat = suiteFibo(n-1)
        # la liste est de taille n
        f = resultat[n-1] + resultat[n-2]
        resultat.append(f)
    return resultat
```

Solution de l'exercice 7 -

```
def precFibo(n):
    p = 0
    f = 0
    f_suivant = 1
    while f_suivant <= n:
        p = p + 1
        f_vieux = f
        f = f_suivant
        f_suivant = f_vieux + f
    return p
```

Solution de l'exercice 8 -

```
def decFibo(n):
    if n == 1:
        return [1]
    else:
        p = precFibo(n)
        l = suiteFibo(p)
        f = l[-1]
        dec = decFibo(n-f)
        dec.append(f)
        return dec
```

Cet algorithme n'est pas optimal car il calcule la liste des nombres de Fibonacci à chaque fois. Dans le calcul de la décomposition de $F_2 + F_4 + \cdots + F_{2r} = F_{2r+1} - 1$ on fera donc $3 + 5 + \cdots + 2r - 1 = (2r)^2 - 1$ sommes, la complexité est quadratique en p.
Pour faire mieux on peut écrire

```
def decFibo(n):
    p = precFibo(n)
    l = suiteFibo(p)
    k = len(l)
    indice = k - 1
    dec_inverse = []
    while indice > 1:
        f = l[indice]
        if f <= n:
            dec_inverse.append(f)
            n = n - f
        indice = indice - 1
    dec = dec_inverse[ : : -1] # on retourne la liste
    return dec
```

Comme les liste sont de taille p, complexité est linéaire en p.

Solution de l'exercice 9 -

```
def appRec(x, liste, debut, fin):
    """Entrées : un nombre x, une liste de nombres,
                 deux indices
       Requis  : la liste est triée
                 0 <= debut, fin < len(liste)
       Sortie  : True ou False selon que x
                 aparaît ou non dans la liste
                 entre debut et fin"""
    if debut > fin: # On ne peut plus chercher
        return False
    else:
        m = (debut + fin)//2
        if liste[m] == x:
            return True
        elif liste[m] < x:
            return appRec(x, liste, m+1, fin)
        else:
            return appRec(x, liste, debut, m-1)
```

```
def appartient(x, liste):
    """Entrées : un nombre x, une liste triée de nombres
       Sortie  : l'existence de x (True/False) dans la liste
          """
    n = len(liste)
    return appRec(x,liste,0,n-1)  # on cherche partout
```

Solution de l'exercice 10 - Pour $k < 0$ l'algorithme fournit le résultat `False`.
On suppose que l'algorithme fournit un résultat pour tout $k < k_0$.
Si `fin - debut` vaut k_0 on calcule m : on a `debut <= m <= fin`.

- Si on a `liste[m] == x`, l'algorithme fournit le résultat `True`.
- Si on a `liste[m] < x`, l'algorithme appelle `appRec(x, liste, m+1, fin)` qui fournit un résultat car on a `fin - (m+1) <= fin - debut - 1 < fin - debut = k0`.
- De même si on a `liste[m] > x`, l'algorithme fournit un résultat.

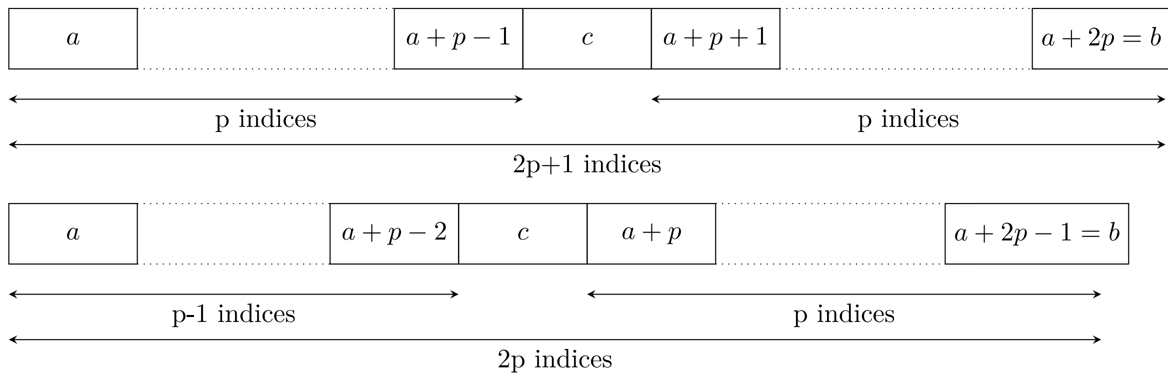
Dans tous les cas l'algorithme renvoie un résultat.
On a ainsi prouvé que l'algorithme termine pour toutes valeurs d'entrée.

Solution de l'exercice 11 - Avec les mêmes notations, si on a $k < 0$, la réponse `False` est correcte car x ne peut pas apparaître dans une liste vide.
On suppose que l'algorithme renvoie la réponse correcte pour tout k avec $k < k_0$.
Si `fin - debut` vaut k_0 on calcule m : on a `debut <= m <= fin`.

- Si on a `liste[m] == x`, la réponse `True` est correcte car x apparaît bien dans la liste.
- Si on a `liste[m] < x`, alors x ne peut pas apparaître avant m donc x est dans la liste entre `debut` et `fin` si et seulement si il apparaît entre $m + 1$ et `fin`. Ainsi le résultat de `app(x, liste, debut, fin)` doit être celui de `app(x, liste, m + 1, fin)` qui est correct d'après l'hypothèse de récurrence.
- De même si on a `liste[m] > x`, la fonction renvoie le bon résultat.

Dans tous les cas la fonction renvoie le bon résultat.

Solution de l'exercice 12 -
On notera que le nombre d'indices parmi lesquels chercher, si on n'a pas trouvé, diminue au moins de moitié : on note a et b les valeurs de `debut` et de `fin`.
Si $b = a + 2p$, le milieu laisse p indices à gauche et p indices à droite, si $b = a + 2p - 1$, le milieu laisse $p - 1$ indices à gauche et p indices à droite.



On note $\gamma(l, a, b)$ le nombre de comparaisons nécessaires à l'exécution de `appartient(x,l,a,b)`.
On a ainsi 3 possibilités, en notant `c = (a+b)//2`,

$$\gamma(l, a, b) = \begin{cases} 1 \\ 2 + \gamma(l, a, c - 1) \\ 2 + \gamma(l, c + 1, n) \end{cases}$$

On note $C(k)$ la borne supérieure des complexités pour traiter k indices.
D'après l'indication du texte, $C(2p + 1) = 2 + C(p)$ et $C(2p) = 2 + C(p)$; il y a égalité car, dans les deux cas, on peut ne pas trouver l'élément et aboutir à une liste de taille p.

On a $C(0) = 0$ d'où $C(1) = 2$ puis, par récurrence, $C(2^k) = 2(k+1)$.

Prouvons alors que, si $2^k \leqslant n < 2^{k+1}$, on a $C(n) = 2(k+1)$.

- Comme on a $C(1) = 2$ et $C(2) = 4$, le résultat est vrai pour $k = 0$ et $k = 1$.
- Si la propriété est vraie pour un entier k, on considère n tel que $2^{k+1} \leqslant n \leqslant 2^{k+2} - 1$.
  — Si n est pair alors, comme $2^{k+2} - 1$ est impair, ces entiers sont distincts donc $2^{k+1} \leqslant n = 2p \leqslant 2^{k+2} - 2$. On a alors $2^k \leqslant p \leqslant 2^{k+1} - 1$ et l'hypothèse de récurrence donne $C(p) = 2(k+1)$ puis $C(n) = C(p) + 2 = 2(k+2)$.
  — Si n est impair alors, comme 2^{k+1} est pair, on a $2^{k+1} + 1 \leqslant n = 2p + 1 \leqslant 2^{k+2} - 1$ d'où $2^{k+1} \leqslant 2p \leqslant 2^{k+2} - 2$ puis $2^k \leqslant p \leqslant 2^{k+1}$. L'hypothèse de récurrence donne $C(p) = 2(k+1)$ puis $C(n) = C(p) + 2 = 2(k+2)$.

  La propriété est alors vraie pour $k+1$.
- Ainsi, par récurrence, la propriété est vraie pour tout k.

Si on $2^k \leqslant n$ alors $k \leqslant \log_2(n)$ donc la complexité, en nombre de comparaisons, est majorée par $2\log_2(n) + 2$, c'est un $\mathcal{O}\big(\log_2(n)\big)$.

Solution de l'exercice 13 -

```
def triPartiel(liste, i):
    """Entrées : une liste et un indice
       Requis  : 0 <= i < len(liste)
       Sortie  : les i premiers éléments sont triés, en place
                 les autres sont inchangés"""
    if i > 0:
        triPartiel(liste, i-1)
        inserer(liste, i-1)
```

```
def tri(liste):
    """Entrée : une liste
       Sortie : la liste est triée"""
    n = len(liste)
    triPartiel(liste, n)
```

Solution de l'exercice 14 -

```
def triPremiers(liste, i):
    """Entrées : une liste et un indice
       Requis  : 0 <= i < len(liste)
       Sortie  : les i plus petits éléments sont triés"""
    if i > 0:
        triPremiers(liste, i-1)
        k = indMinDepuis(liste, i-1)
        echange(liste, k, i-1)
```

```
def tri(liste):
    """Entrée : une liste
       Sortie : la liste est triée"""
    n = len(liste)
    triPremiers(liste, n)
```

On pourrait n'appeler que `triPremiers(liste, n-1)`.

Solution de l'exercice 15 -

```
def motif(c,n):
    if n==0 :
        forward(c)
```

```
    else :
        motif(c/3, n-1)
        left(60)
        motif(c/3, n-1)
        right(120)
        motif(c/3, n-1)
        left(60)
        motif(c/3, n-1)
```

Chapitre IV

TRIS RAPIDES

1 Retour sur les tris classiques

La dernière étape du tri par insertion d'une liste de taille n consiste à

1. trier les $n-1$ premier éléments,
2. insérer le dernier élément.

On peut voir le tri par sélection d'une liste de taille n sous la forme

1. on choisit le plus petit élément et on le place en tête
2. on trie les $n-1$ derniers éléments.

Ces écritures sont, de manière essentielle, récursives.

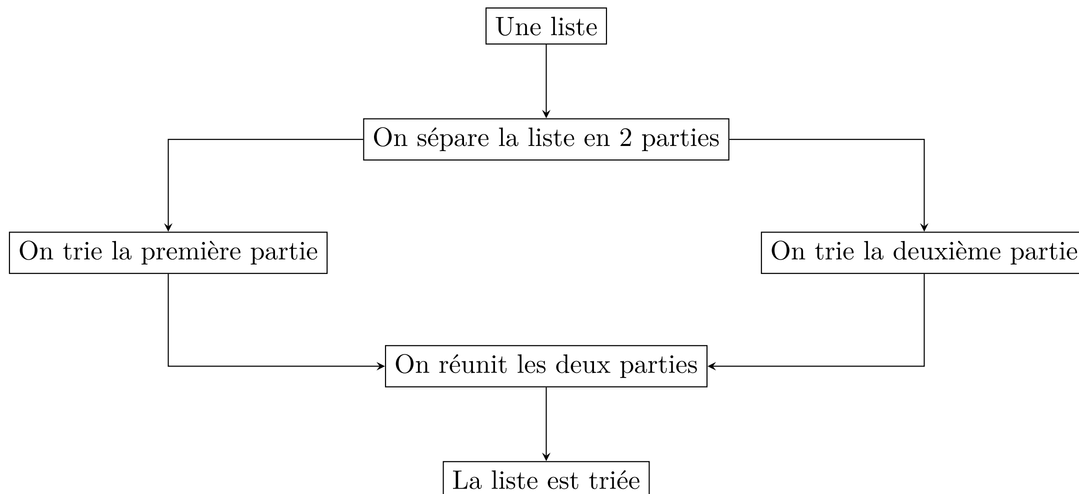
On a laissé dans le flou la signification de "*trier les $n-1$ premiers éléments*" et "*on trie les $n-1$ derniers éléments*", les détails sont demandés dans les exercices du chapitre précédent.

On peut rassembler les deux algorithmes sous la forme

1. On sépare un élément du reste :
    - le dernier élément dans le cas du tri par insertion,
    - le plus petit élément dans le cas du tri par sélection.
2. On trie les éléments restants de manière récursive.
3. On ajoute l'élément séparé :
    - on l'insère dans le cas du tri par insertion,
    - c'est automatique dans le cas du tri par sélection.

On va généraliser ce schéma en n'imposant plus de couper la liste de taille n en un élément et une liste de taille $n-1$ mais en séparant en deux parties de tailles quelconques.

1. On sépare la liste en deux sous-listes `l1` et `l2`.
2. On trie `l1` et `l2` de manière récursive
3. On assemble `l1` et `l2` pour obtenir une liste triée.



Dans le cas du tri par insertion la séparation est facile, on isole un terme mais l'assemblage est difficile car on doit insérer ce terme à sa place.
Par contre, dans le cas du tri par sélection, la séparation est difficile, on doit chercher la plus petit élément mais l'assemblage est facile car les élément sont à leur place.
Il serait idéal de trouver un tri pour lequel séparation et assemblage sont simples mais cela semble impossible. Nous allons proposer deux tris : pour l'un la séparation est immédiate mais l'assemblage est difficile, c'est le tri-fusion, pour l'autre la séparation sera la partie coûteuse, c'est le tri rapide.

	Séparation facile, assemblage difficile	Séparation difficile, assemblage facile
Un singleton et le reste	Tri par insertion	Tri par sélection
Deux parties tailles quelconques	Tri fusion	Tri rapide

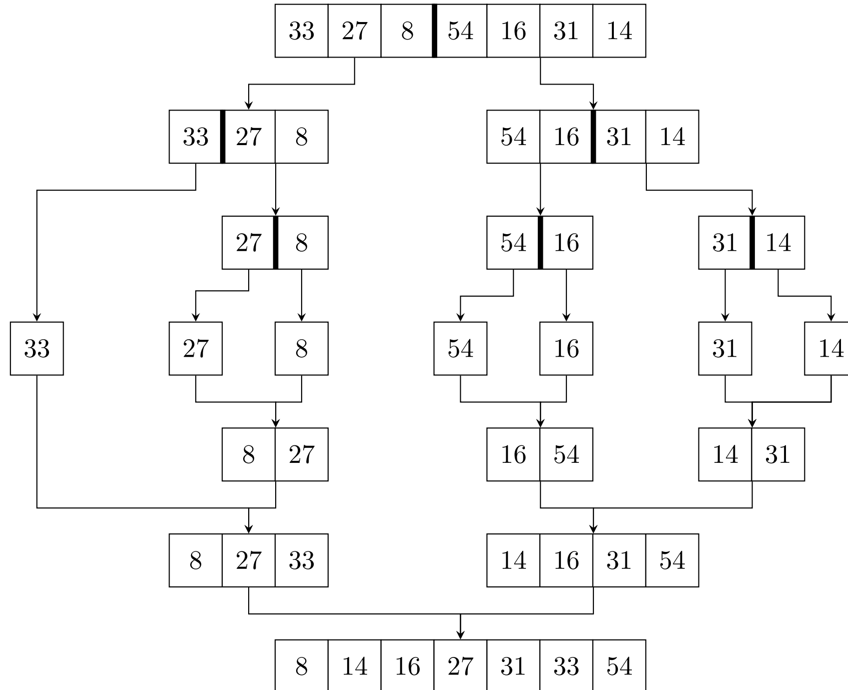
On supposera que les éléments des listes à trier sont comparés à l'aide d'une fonction, nommée `plusGrand`, qui renvoie `True` si le premier argument est supérieur (ou égal) au second.
Dans le cas d'éléments simples, la fonction `plusGrand` est simple aussi

```
def plusGrand(x1, x2):
    """Entree : deux nombres
       Sortie : True ou False selon que x1 > x2 ou non"""
    return x1 > x2
```

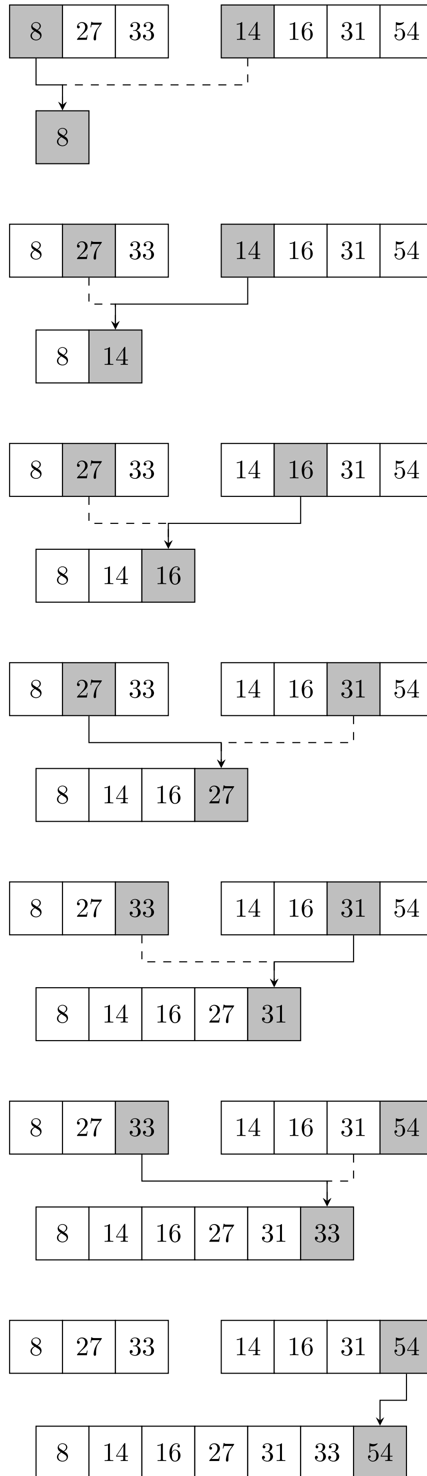
2 Tri-fusion

2.1 Principe

Le tri-fusion est une application du principe **diviser pour régner** : on sépare les données en deux parts presque égales, on traite chaque partie, on rassemble.



Nous verrons que la partie difficile, l'assemblage, ne se fait pas en échangeant directement les éléments dans la liste. On va devoir effectuer beaucoup de copies d'éléments de la liste.
On montre les détails de la dernière étape.


8
27
33
14
16
31
54
8
8
27
33
14
16
31
54
8
14
8
27
33
14
16
31
54
8
14
16
8
27
33
14
16
31
54
8
14
16
27
8
27
33
14
16
31
54
8
14
16
27
31
8
27
33
14
16
31
54
8
14
16
27
31
33
8
27
33
14
16
31
54
8
14
16
27
31
33
54

2.2 Tri externe

Dans un premier temps on choisit d'écrire un algorithme de tri externe : la liste initial est inchangée et on renvoie une nouvelle liste qui contient tous les éléments initiaux mais triés.
La division va être facile à écrire : on extrait deux listes en coupant à la moitié. L'algorithme est récursif : on doit déterminer et traiter le cas d'arrêt.
Lors de la séparation la taille des listes diminue sauf pour une liste de longueur 1 ; dans ce cas on renvoie une copie de la liste. On peut y inclure le cas de la longueur 0, ce qui permettra de pouvoir traiter le tri d'une liste vide.

Programme IV.1 – Tri fusion externe

```
def triFusion(liste):
    """Entree : une liste
       Sortie : une liste triée avec les mêmes éléments"""
    n = len(liste)
    if n <= 1:
        return deepcopy(liste)
    else:
        l1 = triFusion(liste[0:n//2])
        l2 = triFusion(liste[n//2:n])
        return fusion(l1,l2)
```

Il reste à écrire la fusion de deux liste ordonnées. L'idée est de placer, dans l'ordre, les éléments des deux listes en choisissant le plus petit des éléments non encore placés.

Programme IV.2 – Fusion de deux listes triées en une nouvelle liste

```
1  def fusion(l1,l2):
2      """Entree : deux listes
3         Requis : les listes sont triées
4         Sortie : une liste triée contenant tous
5                  les éléments des deux listes initiales"""
6      n1 = len(l1)
7      n2 = len(l2)
8      resultat = []
9      pos1 = 0
10     pos2 = 0
11     for i in range(n1+n2):
12         if pos1 == n1:
13             resultat.append(l2[pos2])
14             pos2 = pos2 + 1
15         elif pos2 == n2:
16             resultat.append(l1[pos1])
17             pos1 = pos1 + 1
18         elif plusGrand(l2[pos2],l1[pos1]):
19             resultat.append(l1[pos1])
20             pos1 = pos1 + 1
21         else:
22             resultat.append(l2[pos2])
23             pos2 = pos2 + 1
24     return resultat
```

- Ligne 8 : on initialise la liste à renvoyer.
- Ligne 9-10 : dans les listes à fusionner on doit savoir où sont les éléments à comparer. Ces éléments sont indiquéspar deux indices de position `pos1` et `pos2`. Ils indiquent le premier élément non encore placé dans la liste finale. Comme les deux listes sont triées le plus petit élément restant est à l'une de ces deux positions.
- Ligne 11 : on répète autant de fois qu'il y d'éléments. On pourrait structurer le remplissage à l'aide d'une boucle `while` mais on connaît le nombre d'éléments à placer donc une boucle `for` est plus simple.
- Lignes 12-14 : quand une des deux listes a été complètement utilisée il n'y a pas besoin de comparer, on place le premier élément non vu de la seconde liste. On traite ici le cas où la première sous-liste est entièrement copiée.
- Lignes 15-17 : de même si la deuxième sous-liste est épuisée.
- Lignes 18-23 : on compare les premiers termes disponibles de chaque sous-liste et on copie le plus petit.
- Lignes 14, 17, 20, 23 : à chaque pas on incrémente l'indice de la sous-liste utilisée

2.3 Analyse

Terminaison

`fusion` termine car elle ne fait appel qu'à une boucle `for`.
La terminaison de la fonction récursive `triFusion` se fait par récurrence.

- Elle termine directement si la liste a moins d'un élément.
- Si elle termine pour les listes de moins de $n-1$ éléments avec $n \geqslant 2$, l'appel de la fonction pour une liste de taille n fait appel à la fonction pour des listes de taille $n//2$ et $n - n//2$.
  Pour n pair, $n = 2p$ on a $n//2 = n - n//2 = p < 2p = n$ car p est non nul pour $n \geqslant 2$.
  Pour n impair, $n = 2p+1$ on a $n//2 = p < n - n//2 = p+1 < 2p+1 = n$.
  Dans tous les cas les appels récursifs terminent d'après l'hypothèse de récurrence.
  Comme `fusion` termine on en déduit que `triFusion` termine pour les liste de taille n.
- La fonction `triFusion` termine pour toutes les listes.

Preuve

On commence par la preuve de la fusion
On suppose que les listes `l1` et `l2` sont triées.
On peut prouver (voir l'exercice 3) que les propriétés suivantes définissent un invariant de boucle dans la fonction fusion

$$\mathcal{P}(i) \begin{cases} (1) & \texttt{pos1 <= n1} \\ (2) & \texttt{pos2 <= n2} \\ (3) & \texttt{pos1 + pos2 = i} \\ (4) & \texttt{resultat + l1[pos1:n1]} \text{ est triée} \\ (5) & \texttt{resultat + l2[pos2:n2]} \text{ est triée} \\ (6) & \texttt{resultat + l1[pos1:n1] + l2[pos2:n2]} \text{ est une permutation de } \texttt{l1 + l2} \end{cases}$$

On peut alors prouver le tri.
Une liste de taille 0 ou 1 est recopiée et est déjà triée, l'algorithme est correct dans ce cas.
On suppose que la fonction renvoie une liste triée avec les mêmes élément pour toute liste de taille majorée par $n-1$. On considère une liste de taille n.
Les listes extraites sont de taille strictement inférieure à n donc les listes `l1` et `l2` sont triées et contiennent les éléments des deux listes extraites. La preuve de la fusion montre alors que la liste renvoyée est triée et contient tous les éléments de la liste initiale.

Complexité

La complexité est le nombre de comparaisons d'éléments de la liste.
La complexité de la fusion de deux listes de taille respectives n_1 et n_2 est au plus $n_1 + n_2$ car on fait au plus une comparaison pour chaque i dans la boucle (en fait au plus $n_1 + n_2 - 1$).
Lors du tri fusion on sépare en deux listes de tailles respectives $n//2$ et $n - n//2$ que l'on trie puis on fusionne les listes triées.
La complexité du tri, $C(n)$, vérifie donc

$$C(n) \leqslant C(n//2) + C(n - n//2) + n$$

Théorème : Complexité du tri-fusion

La complexité du tri fusion d'une liste de taille n en nombre de comparaisons d'éléments vérifie $C(n) = \mathcal{O}\big(n\log_2(n)\big)$.

Démonstration On va montrer par récurrence sur p que $C(n) \leqslant p.2^p$ pour $n \leqslant 2^p$.
Pour $p = 0$ cela découle de $C(0) = C(1) = 0$.
Si la propriété est vraie pour p on suppose qu'on a $n \leqslant 2^{p+1}$.
Alors $n//2 \leqslant 2^p$ et $n - n//2 \leqslant 2^p$ d'où $C(n//2) \leqslant p.2^p$ et $C(n - n//2) \leqslant p.2^p$.
On en déduit $C(n) = C(n//2) + C(n - n//2) + f(n//2, n - n//2) \leqslant p2^p + p2^p + 2^{p+1}$
Ainsi $C(n) \leqslant (p+1)2^{p+1}$: la propriété est vraie pour $p + 1$.
Si p est choisi tel que $2^{p-1} < n \leqslant 2^p$ on a $2^p \leqslant 2n$ et $p \leqslant 1 + \log_2(n)$ d'où
$C(n) \leqslant \big(1 + \log_2(n)\big)2n = \mathcal{O}\big(n\log_2(n)\big)$.

2.4 Tri en place

On peut modifier le schéma proposé en introduction en triant une portion de liste plutôt que des listes extraites. Le tri se fera alors en place et demandera moins de listes créées. Cependant on n'échappe pas à la recopie double de tous les éléments à chaque étape. Voir l'exercice 6 pour une amélioration.
On choisit ici de définir la fonction auxilliaire à l'intérieur de la fonction principale, cela permet de ne pas donner la liste comme paramètre car elle est définie dans la fonction.

Programme IV.3 – Tri fusion en place

```
def triFusion(liste):
    """Entree : une liste
       Sortie : une liste triée avec les mêmes éléments"""
    n = len(liste)
    def triFusionEntre(i, j):
        """Entrées : deux indices
           Sortie : la liste est triée entre i et j (compris)
               """
        if i < j: # Au moins deux points
            p = (i+j)//2
            triFusionEntre(i, p)
            triFusionEntre(p+1, j)
            fusionInterne(liste, i, j, p)
    return triFusionEntre(0, n-1)
```

Programme IV.4 – Fusion de deux parties triées de liste

```
def fusionInterne(liste, i, j, p):
    """Entree : une liste et 3 indices
       Requis : i <= p < j, la liste est triée entre i et p
                            et entre p+1 et j
       Sortie : la liste est triée entre i et j"""
    n1 = p - i +1
    n2 = j - p
    l1 = liste[i: p+1]
    l2 = liste[p+1: j+1]
    pos1 = 0
    pos2 = 0
    for k in range(i, j+1):
        if pos1 == n1:
            liste[k] = l2[pos2]
            pos2 = pos2 + 1
        elif pos2 == n2:
            liste[k] = l1[pos1]
            pos1 = pos1 + 1
        elif plusGrand(l2[pos2],l1[pos1]):
            liste[k] = l1[pos1]
            pos1 = pos1 + 1
        else:
            liste[k] = l2[pos2]
            pos2 = pos2 + 1
```

3 Tri rapide

Le tri rapide inverse la difficulté : plutôt que fusionner deux sous-listes triées qui viennent d'un découpage arbitraire il découpe la liste selon un **pivot** qui sert de borne pour séparer les éléments en les comparant au pivot.
La séparation en deux listes demande donc un travail, par contre l'assemblage des listes triées est immédiat car les éléments de la première sous-liste sont inférieurs aux éléments de la seconde.

3.1 Écriture du tri

Dans le tri fusion on a renvoyé une nouvelle liste (triée) car la fusion en place est ardue.
Pour le tri rapide il est possible de réaliser le tri en place car le découpage peut se faire ainsi.
On doit choisir l'élément qui sert de pivot : nous allons utiliser le dernier élément de la liste à trier mais on pourra utiliser un élément choisi au hasard. Pour ce faire il suffirait d'échanger l'élément choisi et le dernier avant d'appliquer les fonctions ci-dessous.
Comme les sous-listes sont incluses dans la liste de taille n on devra écrire des fonctions intermédiaires qui travaillent sur une portion de la liste. Il faudra inclure deux paramètres qui correspondent au début et à la fin du segment, on choisira les paramètre a et b tels que les indices utilisés sont ceux allant de a à b, bornes comprises.

On commence par le découpage en deux sous-segments.
On veut transformer un segment de la forme

33	27	8	54	16	31	14	29

en un segment, qui pourrait être

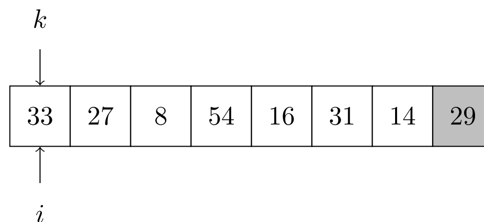
27	8	16	14	29	31	54	33

On peut remarquer que le pivot est à sa place. En effet les termes placés avant lui sont inférieurs et les termes placés ensuite sont supérieurs.

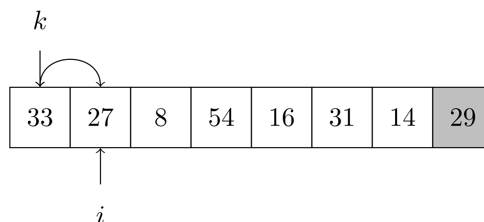
On parcourt la liste entre a et $b-1$ puisque le pivot est en b.
On maintient 2 variables d'indice

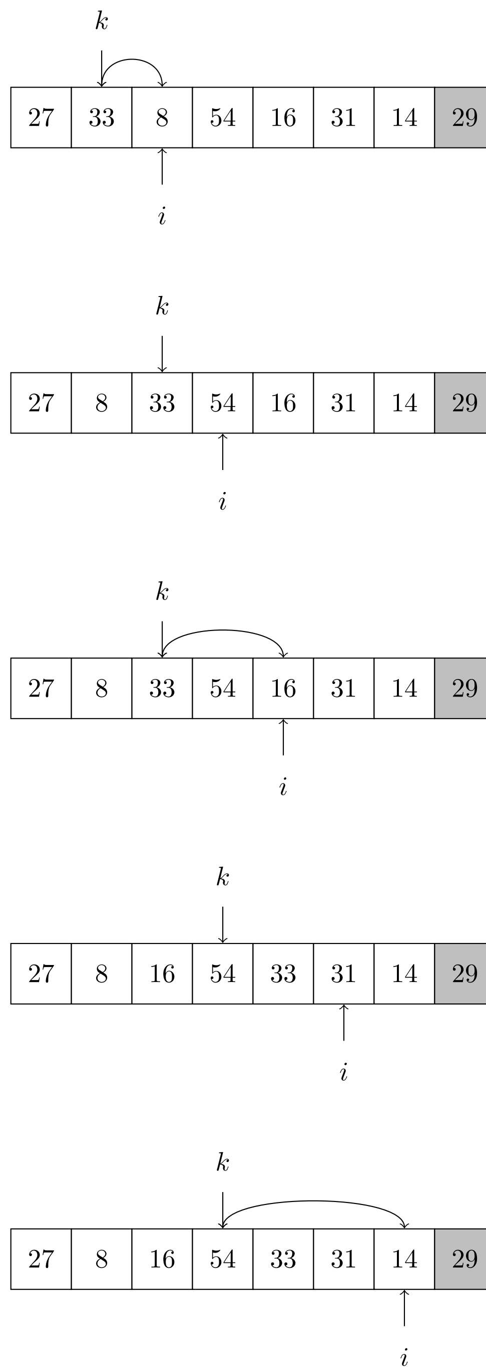
- la première, i, est l'indice de la boucle `for` qui est la position de l'élément étudié,
- la seconde, k, est la première position après les termes plus petits que le pivot.

On compare chaque terme au pivot.
S'il est plus grand on le laisse en place.

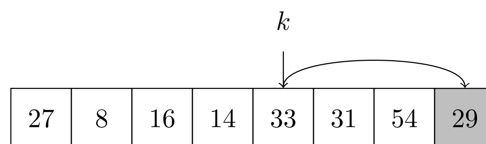


S'il est plus petit, on le permute pour le mettre à la position k puis on incrémente k.





À la fin on permute le pivot à la position k.



On renverra la position finale du pivot car elle servira à définir les deux segments suivants.

Programme IV.5 – Découpage pour le tri rapide

```
def decoupe(liste,a,b):
    """Entree : une liste et deux entiers
       Requis : 0 <= a <= b < len(liste)
       Sortie : un entier p (a <= p <= b) avec
       le dernier élément placé en p qui sépare
       les élément plus petits et plus grands que lui """
    k = a
    pivot = liste[b]
    for i in range(a, b):
        if plusGrand(pivot, liste[i]):
            echange(liste, i, k)
            k = k + 1
    echange(liste,k,b)
    return k
```

Pour écrire le tri on utilise une fonction auxiliaire, récursive qui trie entre 2 bornes .

Programme IV.6 – Tri rapide

```
def triRapideBornes(liste, a, b):
    """Entree : une liste et deux entiers
       Requis : 0 <= a <= b < len(liste)
       Sortie : la liste est triee entre a et b"""
    if b > a:
        p = decoupe(liste, a, b)
        triRapideBornes(liste, a, p-1)
        triRapideBornes(liste, p+1, b)

def triRapide(liste):
    """Entree : une liste
       Sortie : la liste est triee"""
    n = len(liste)
    triRapideBornes(liste, 0, n-1)
```

3.2 Analyse

Terminaison

`decoupe` termine car elle ne fait appel qu'à une boucle `for`.
On prouve que la fonction récursive `triRapideBornes` termine par récurrence sur $b - a$.

- Pour $b - a \leqslant 0$ la fonction ne fait aucune instruction : elle termine.
- On suppose que `triRapideBornes(liste, i, j)` termine pour toute paire d'indices i et j tels que $j - i \leqslant m$. Soient a et b tels que $b - a = m + 1$.
  La fonction `decoupe` termine et renvoie un entier p tel que $a \leqslant p \leqslant b$. On a alors $p - 1 - a \leqslant b - a - 1 \leqslant m$ donc l'appel de `triRapideBornes(liste, a, p-1)` termine.
  De même $b-(p+1) \leqslant b-a-1$ donc l'appel de `triRapideBornes(liste, p+1, b)` termine.
  Ainsi l'appel de `triRapideBornes(liste, a, b)` termine.
- On a prouvé le résultat par récurrence.

On en déduit immédiatement que `triRapide` termine.

Preuve

La fonction de séparation doit produire, à chaque étape, quatre parties :
1. les éléments plus petits que le pivot, on a choisit une inégalité large,
2. suivis des éléments strictement supérieurs au pivot,
3. les éléments non traités suivent
4. et le pivot est à la dernière position durant la boucle.

On peut alors imaginer un invariant de boucle.

$$\mathcal{P}(i)\begin{cases}(1) & \texttt{liste[a:k]} \text{ est majorée par le pivot}\\ (2) & \texttt{liste[k:i]} \text{ est strictement minorée par le pivot}\end{cases}$$

La démonstration est proposée dans l'exercice 8.
La preuve du tri s'en déduit : voir l'exercice 9.

Complexité

`decoupe(l,a,b)` fait une comparaison pour chaque i entre a et $b-1$, sa complexité, en nombre de comparaisons, est donc $b-a$.
On note $C(l,a,b)$ le nombre de comparaisons lors de l'appel de `triRapideBornes(l,a,b)`.
On a donc donc $C(l,a,b) = b-a+C(l,a,p-1)+C(l,p+1,b)$ si la fonction `decoupe` a renvoyé p.

Exemple 1 Dans le cas où `decoupe(l,a,b)` renvoie b à chaque appel, ce qui est le cas lorsque la liste initiale est déjà triée, la relation ci-dessus devient
$C(l,a,b) = b-a+C(l,a,b-1)+C(l,b+1,b) = b-a+C(l,a,b-1)$
car la complexité pour une liste vide est nulle.
On en déduit qu'alors $C(l,a,b) = b-a+(b-a-1)+\cdots+1$.
En particulier $C(l,0,n-1) = \frac{n(n-1)}{2}$: la complexité du tri d'une liste de taille n est un $\mathcal{O}\big(n^2\big)$, le tri n'est pas efficace dans ce cas.

Exemple 2
On suppose que $n = 2^p - 1$ et qu'à chaque étape `decoupe(l,a,b)` renvoie $(a+b)//2$.
On montre alors qu'à chaque étape on a des extractions de taille $2^k - 1$.
Si on note $C'(k) = C(l,a,a+2^k-2)$ on aboutit à $C'(k) = 2^k - 2 + 2C'(k-1)$.
On pose alors $C'(k) = 2^k u_k$ et on obtient $u_k = 1 - 2^{1-k} + u_{k-1}$ avec $u_1 = 0$ donc
$u_k = k-1-2^{-1}-2^{-2}-\cdots-2^{-(k-1)} = k-2+2^{-(k-1)}$ d'où $C'(k) = k2^k - 2^{k+1} + 2$.
Ainsi $C(l,0,n) = \log_2(n+1)(n+1) - 2n$ pour $n = 2^p - 1$; la complexité est un $\mathcal{O}\big(n\log_2(n)\big)$, le tri est efficace dans ce cas.

En moyenne, le tri est efficace.

Théorème : Complexité moyenne du tri rapide
La complexité moyenne du tri rapide d'une liste de taille n en nombre de comparaisons d'éléments vérifie $\mathcal{O}\big(n\log_2(n)\big)$.

Le résultat doit être connu mais pas sa démonstration.
celle-ci est cependant proposée dans l'exercice 13.

4 Exercices

Exercice 1 — Nombre de comparaisons
On suppose que l'on trie à l'aide de comparaisons une liste de taille n.
Montrer que le nombre de comparaisons à effectuer est au plus $\frac{1}{2}n(n-1)$.
Montrer que si on a $2^p < n! \leqslant 2^{p+1}$ alors on doit avoir effectuer au moins $p+1$ comparaisons pour pouvoir trier toutes les listes de taille n.

Rappel : la fonction `floor`, ou partie entière, est définie par la propriété que $\lfloor x \rfloor$ est le plus grand entier inférieur ou égal à x.
On définit de même la fonction `ceil` : $\lceil x \rceil$ est le plus petit entier supérieur ou égal à x.

$$n = \lfloor x \rfloor \iff \begin{cases} n \in \mathbb{Z} \\ n \leqslant x < n+1 \end{cases} \qquad n = \lceil x \rceil \iff \begin{cases} n \in \mathbb{Z} \\ n-1 < x \leqslant n \end{cases}$$

Exercice 2 — Fonctions `floor` et `ceil`

1. Prouver que $\lceil x \rceil = 1 + \lfloor x \rfloor$ si x appartient à $\mathbb{R} \setminus \mathbb{Z}$.
   Que se passe-t-il pour $x \in \mathbb{Z}$?
2. Prouver que, pour tout $n \in \mathbb{Z}$, $\left\lfloor \frac{n}{2} \right\rfloor = n//2$ et $\left\lfloor \frac{n}{2} \right\rfloor + \left\lceil \frac{n}{2} \right\rceil = n$.
   $\left\lfloor \frac{n}{2} \right\rfloor$ et $\left\lceil \frac{n}{2} \right\rceil$ sont donc les cardinaux des sous-listes dans le tri-fusion.
3. Prouver que, pour tout $n \in \mathbb{Z}$, $\left\lfloor \frac{n+1}{2} \right\rfloor = \left\lceil \frac{n}{2} \right\rceil$ et $\left\lceil \frac{n+1}{2} \right\rceil = \left\lfloor \frac{n}{2} \right\rfloor + 1$

Exercice 3 — Invariant de boucle pour la fusion
Prouver que l'ensemble des propriétés suivantes est un invariant de boucle dans la fonction `fusion` (programme IV.4, page 60)

$$\mathcal{P}(i) \begin{cases} (1) & \texttt{pos1 <= n1} \\ (2) & \texttt{pos2 <= n2} \\ (3) & \texttt{pos1 + pos2 = i} \\ (4) & \texttt{resultat + l1[pos1:n1]} \text{ est triée} \\ (5) & \texttt{resultat + l2[pos2:n2]} \text{ est triée} \\ (6) & \texttt{resultat + l1[pos1:n1] + l2[pos2:n2]} \text{ est une permutation de } \texttt{l1 + l2} \end{cases}$$

Exercice 4 — Complexité maximale du tri-fusion
On a vu que le le nombre maximal de comparaisons nécessaire dans le tri-fusion une liste de n éléments, $C(n)$, vérifie $C_0 = C_1 = 0$ et $C_n = C_{\left\lfloor \frac{n}{2} \right\rfloor} + C_{\left\lceil \frac{n}{2} \right\rceil} + n - 1$.

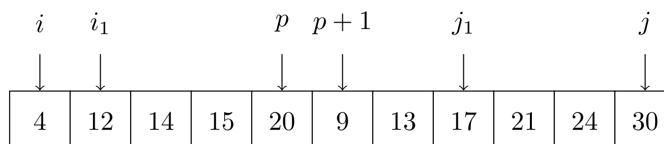
1. On pose $D_n = C_{n+1} - C_n$. Prouver que $D_n = D_{\left\lfloor \frac{n}{2} \right\rfloor} + 1$ avec $D_0 = 0$.
2. En déduire que $D(n) = p+1$ pour n tel que $2^p \leqslant n < 2^{p+1}$. On notera $p = L(n)$.
3. Conclure que $C_n = nL(n) + n + 1 - 2^{L(n)+1}$ pour $n \geqslant 1$.

Exercice 5 — Déplacements partiels dans la fusion en place
Dans le tri fusion en place la fonction de fusion copie tous les éléments dans deux nouvelles listes avant de les replacer dans la liste initiale.
En fait il est possible de garder la portion de liste de droite en mémoire et d'effectuer la fusion. Écrire une fonction `fusion_gauche` qui effectue ainsi la fusion.
En remplissant la portion de liste depuis la droite on peut aussi ne copier que la partie droite : écrire une fonction `fusion_droite` qui correspond.

Exercice 6 — Améliorations de la fusion en place
Dans la pratique il arrive que les listes à trier soient partiellement triées. Les améliorations proposées ici sont destinées à diminuer le nombre de comparaisons effectuées dans le cas de listes peu désordonnées. Ces améliorations sont employées dans le tri `TimSort` de python dû à Tim Peters.
Dans l'exercice précédent on peut remarquer que les éléments les plus grands de la partie de droite (resp. les éléments les plus petits de la partie de gauche) sont laissés en place lors de l'appel à `fusion_gauche` (resp. de `fusion_droite`). On va tenir compte de cette possibilité.

On considère l'appel de `fusionInterne(liste, i, p, j)` avec $i \leqslant p < j$.

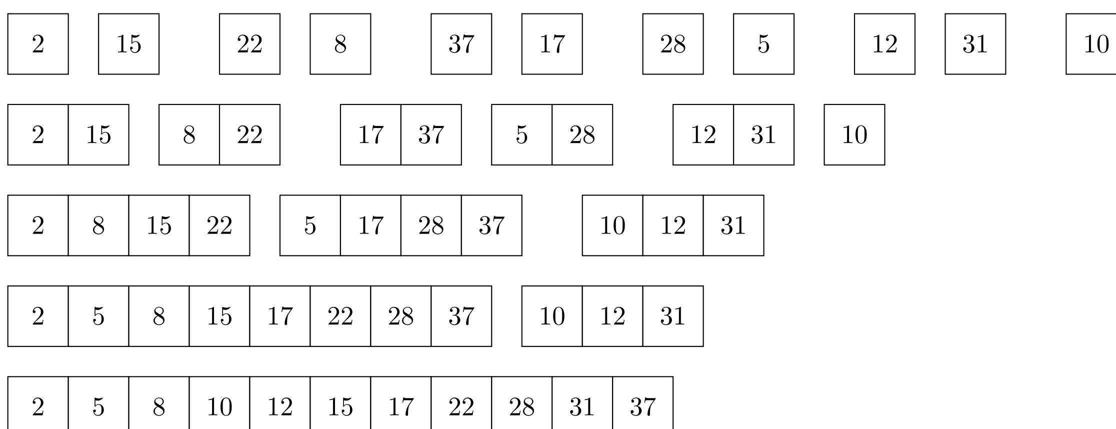
1. Si on a `liste[p] <= liste[p+1]` les deux parties forment déjà une liste triée.
2. Si on a `liste[p] > liste[p+1]` alors on considère le plus grand entier $k \in \{p+1, p+2, \ldots, j\}$, noté j_1, tel que `liste[p] > liste[k]`. Les éléments de la liste entre les indices $j_1 + 1$ et j sont tous plus grands que ceux de la première partie donc sont à leur place.
   De même, si $i_1 = \min\{k\ ;\ i \leqslant k \leqslant p,$ `liste[k] > liste[p+1]`$\}$, les éléments d'indices i à $i_1 - 1$ sont à leur place.
3. Il reste alors à fusionner les parties entre i_1 et p à gauche et entre $p+1$ et j_1 à droite. Pour minimiser les copies on utilisera `fusion_gauche` ou `fusion_droite` selon les tailles des deux parties.



Dans cet exemple on effectue `fusion_droite(liste, i1, p, j1)`.

Après avoir écrit les fonctions de recherche pour i_1 et j_1 en utilisant la méthode de dichotomie, écrire la fonction `fusionInterne` correspondante.

Exercice 7 — Tri fusion itératif
Pour réaliser le tri fusion on peut opérer de manière itérative.
On fusionne les partie de liste de taille 1 deux par deux : on obtient des parties de taille 2 triées.
On fusionne ces parties pour obtenir des parties de taille 4 triées.
On recommence ainsi jusqu'à obtenir une liste triée.
Le travail n'est pas aussi simple car il peut y avoir des résidus de taille plus petite.
Exemple, pour `[2, 15, 22, 8, 37, 17, 28, 5, 12, 31, 23]`, de taille 11.



Écrire la fonction correspondante.

Exercice 8 — Invariant de boucle pour la séparation du tri rapide
Prouver que l'ensemble des propriétés suivantes est un invariant de boucle dans la fonction `decoupe` (programme IV.5, page 63)

$$\mathcal{P}(i) \begin{cases} (1) & \texttt{liste[a:k]} \text{ est majorée par le pivot} \\ (2) & \texttt{liste[k:i]} \text{ est strictement minorée par le pivot} \end{cases}$$

Exercice 9 — Preuve du tri rapide
Prouver que le tri rapide trie la liste passée en paramètre.

Exercice 10 — Complexité maximale du tri rapide
Prouver que la complexité du tri rapide d'une liste de taille n, en nombre de comparaisons, est majorée par $\frac{n(n-1)}{2}$.

Exercice 11 — Exemples de complexités minimale dans le tri rapide
`l3 = [1,3,2,6,5,7,4]` et `l4 = [1,3,2,6,5,7,4,12,9,11,10,14,13,15,8]`.

1. Combien de comparaisons sont effectuées pour trier `l3` et `l4` ?
2. Donner un exemple semblable pour une liste de taille 31.
3. De manière générale comment construire une liste de taille $2^p - 1$ telle que les découpages donneront des listes qui seront toujours de taille $2^k - 1$ comme dans l'étude du cours ?

Exercice 12 — Médiane
L'élément médian d'une liste est un élément p de la liste tel que le nombre d'élément de liste inférieurs à p est égal au nombre d'éléments supérieurs (ou à ce nombre -1).
Par exemple 17 est médian de `[5, 27, 12, 33, 17, 8, 18]`

1. Comment calculer l'élément médian d'une liste de taille n en $\mathcal{O}\big(n\log_2(n)\big)$?
2. En adaptant l'algorithme de tri rapide montrer qu'on peut déterminer l'élément médian en temps linéaire en moyenne. L'idée est de généraliser la recherche au p-ième élément et de le chercher dans la "bonne sous-liste".

Exercice 13 — Complexité moyenne du tri rapide
On s'intéresse à la complexité moyenne du tri-rapide.
On calcule la moyenne sur tous les désordres possibles d'un ensemble de n éléments distincts $l_1 < l_2 < \ldots < l_n$.
On note S_n, de cardinal $n!$, l'ensemble des permutations de $\{1, 2, \ldots, n\}$.
À chaque permutation de n éléments σ, on associe la liste $L_\sigma = [l_{\sigma(1)}, l_{\sigma(2)}, \ldots, l_{\sigma(n)}]$.
La complexité moyenne est donc $T(n) = \dfrac{1}{n!}\displaystyle\sum_{\sigma\in S_n} C(L_\sigma)$.
Si on a $\sigma(n) = k$ alors la coupure sépare la liste en une sous-liste de taille $k-1$, g_σ, le pivot et une sous-liste de taille $n-k$, g_σ, : on admettra[1] que les distributions des g_σ et d_σ sont équiprobables.

1. Quel est le nombre de permutations telles que $\sigma(n) = k$?
2. En utilisant l'équiprobabilité prouver que $T(n)$ vérifie
$$T(n) = \frac{1}{n}\sum_{k=1}^{n}\left(T(k-1) + T(n-k) + n - 1\right)$$
3. Par des changements d'indice en déduire que $T(n) = n - 1 + \dfrac{2}{n}\displaystyle\sum_{i=0}^{n-1} T(i)$.
4. En utilisant aussi la relation pour $n-1$ prouver que $nT(n) = (n+1)T(n-1) + 2(n-1)$.
5. Si on pose $u_n = \frac{1}{n+1}T(n)$ prouver qu'on a $u_n \leqslant u_{n-1} + \frac{2}{n}$.
6. En déduire que $u_n \leqslant 2\displaystyle\sum_{k=2}^{n}\frac{1}{k}$ puis $T(n) \leqslant 2(n+1)\ln(n)$.

1. Exercice de probabilité : démontrez-le

Exercice 14 — Construction du désordre
On peut souhaiter construire des listes non triées, par exemple pour tester les différents tris. Nous allons utiliser la fonction `randint` du module `random` :

```
from random import randint
```

`randint(a,b)` renvoie un entier choisi au hasard dans $\{a, a+1, \cdots, b-1, b\}$ [2].
Écrire une fonction `listeHasard(p,n)` qui renvoie une liste de p nombres, chacun étant choisi au hasard entre 1 et n.

La fonction précédente n'est pas adaptée si on veut des listes sans répétition ; pour n assez grand la probabilité qu'il n'y ait pas deux élément égaux est de l'ordre de $\frac{1}{\sqrt{e}} \sim 0,4$.
Pour s'assurer d'une liste sans doublon de p éléments on va la calculer sous la forme `L[i] = h*p+i` où h est un nombre choisi au hasard entre 1 et p.
On obtiendra une liste de p termes (compris entre p et $p^2 - 1$) tous distincts car les restes modulo p sont distincts.
Écrire cette fonction `listeH(p)`

Exercice 15 — Tri rapide sans récursivité
On peut écrire un algorithme de tri semblable au tri rapide sans utiliser la récursivité.
L'idée est de maintenir, en parallèle de la liste, une liste de booléens qui indique si un élément est bien placé, ce qui signifie ici que tous les éléments placés avant lui sont inférieurs et les éléments placés après sont supérieurs. On s'interdit alors de permuter ces éléments bien placés.
Au fur et à mesure on obtient des éléments en place qui séparent des segments qui restent à trier mais dont les positions finales resteront dans le segment.

Voici une situation intermédiaire :
[**2**, **5**, **7**, 11, 16, 8, 13, 6, **24**, 33, 27]
Les éléments en gras sont à leur place.
Les tranches à trier sont `[11, 16, 8, 13, 6]` et `[33, 27]`
On est à l'indice 3, l'élément n'est pas à sa place.
Il sert de pivot et on le déplacera d'un cran à droite pour chaque élément du segment suivant qui lui est inférieur en le remplaçant par cet élément.
On part de [**2**, **5**, **7**, *11*, 16, 8, 13, 6, **24**, 33, 27].

1. 16 est supérieur : on ne fait rien.
2. 8 est inférieur au pivot on permute les 3 éléments :
   [**2**, **5**, **7**, 8, *11*, 16, 13, 6, **24**, 33, 27].
3. 13 est supérieur au pivot, on ne fait rien
4. 6 est inférieur au pivot on permute les éléments 11, 16 et 6
   [**2**, **5**, **7**, 8, 6, *11*, 13, 16, **24**, 33, 27].
5. On arrive à un élément bien placé : on arrête la boucle et le pivot est bien placé
   [**2**, **5**, **7**, 8, 6, **11**, 13, 16, **24**, 33, 27].

L'élément à l'indice 3 n'est toujours pas marqué comme bien placé on doit recommencer. Cependant le segment diminue à chaque étape donc on finira par le placer.

Écrire une fonction qui applique cet algorithme. Elle pourra commencer par

```
def triKnuth(liste):
    n = len(liste)
    enPlace = [False]*n
    for k in range(n):
        while not enPlace[k]:
            .......
```

2. noter que b est inclus

5 Solutions

Solution de l'exercice 1 - Le nombre de couples d'éléments dans un ensemble de n éléments est $\binom{n}{2} = \frac{1}{2}n(n-1)$: c'est le nombre maximal de comparaisons distinctes que l'on peut faire.
Il y a $n!$ listes possibles avec n éléments distincts. Le tris de chacune de ces listes doivent être distincts donc les comparaisons doivent permettre de les distinguer. k comparaisons permettent de distinguer 2^k cas donc il faut au moins $p+1$ comparaisons pour distinguer tous les désordres possibles.
On remarquera que l'équivalent $\ln(n!) \sim n\ln(n)$ donne $p \sim Kn\ln(n)$: les complexités en $\mathcal{O}\big(n\log_2(n)\big)$. sont optimales.

Solution de l'exercice 2 -

1. Si x n'est pas entier et $n \leqslant x < n+1$ pour $n \in \mathbb{Z}$ alors $n < x < n+1$ d'où $n = \lfloor x \rfloor$ et $n+1 = \lceil x \rceil$. Pour n entier $\lfloor n \rfloor = \lceil n \rceil = n$.
2. Pour n pair, $n = 2p$, $\lfloor \frac{n}{2} \rfloor = \lfloor p \rfloor = p = n//2$ et $\lceil \frac{n}{2} \rceil = \lceil p \rceil = p = n - n//2$.
   Pour n impair, $n = 2p+1$, $\lfloor \frac{n}{2} \rfloor = \lfloor p + \frac{1}{2} \rfloor = p = n//2$
   et $\lceil \frac{n}{2} \rceil = \lceil p + \frac{1}{2} \rceil = p+1 = n - n//2$.
3. Pour n pair, $n = 2p$, $\lfloor \frac{n}{2} \rfloor = \lceil \frac{n}{2} \rceil = p$ et $\lfloor \frac{n+1}{2} \rfloor = p$, $\lceil \frac{n+1}{2} \rceil = p+1$.
   Pour n impair, $n = 2p+1$, $\lfloor \frac{n}{2} \rfloor = p$, $\lceil \frac{n}{2} \rceil = p+1$ et $\lfloor \frac{n+1}{2} \rfloor = \lceil \frac{n+1}{2} \rceil = p+1$

Solution de l'exercice 3 -

- Au départ `resultat` est vide `l1` et `l2` sont triées et `i = pos1 = pos2 = 0` donc les propriétés sont vraies.
- On suppose que ces propriétés sont vraies au départ d'un passage de boucle avec `i < n1 + n2` ; on effectue alors la boucle.
  On notera `pos1'`, `pos2'` et `resultat'` les valeurs des variables après la boucle, on note aussi `i' = i+1`.
  On suppose qu'on a ajouté `l1[pos1]` à la liste `resultat`.
  Cela ne peut pas se produire pour `pos1 == n1`, traité à la ligne 12 : on a `pos1 < n1`.
  On a alors `pos1' = pos1 + 1 < n1 + 1` donc `pos1' <= n1`, ce qui donne (1).
  La conservation `pos2' = pos2` donne (2).
  `pos1' + pos2' = pos1 + 1 + pos2 = i + 1 = i'`, ce qui donne (3).
  `resultat' = resultat + [a]` et `l1[pos1:n1] = [a] + l1[pos1':n1]` donc `resultat' + l1[pos1':n1] = resultat + l1[pos1:n1]`, la propriété (4) reste vraie.
  Comme on a `l2[pos2':n2] = l2[pos2:n2`, la propriété (6) reste vraie aussi.
  `l1[pos1]` minore `l2[pos2]` donc le dernier terme de `resultat'` minore le premier terme de `l2[pos2':n2]`,
  `resultat'` est triée d'après la conservation de (4)
  et `l2[pos2':n2]` est triée car extraite d'une liste triée.
  On en déduit que (5) est vérifié.

  On conclut de même si `a = l2[pos2]`.
  Ainsi, si $\mathcal{P}(i)$ est valide au départ de la boucle pour i, $\mathcal{P}(i+1)$ est valide à la fin de la boucle.
- Pour $i = n1 + n2$, les 3 premières conditions imposent `pos1 = n1` et `pos2 = n2`.
  La condition (4) ou (5) montre que `resultat` est triée et la condition (6) indique que cette liste contient tous les éléments. Ainsi le résultat est bien la fusion triée des deux listes.

On a bien un invariant de boucle qui prouve le résultat attendu.

Solution de l'exercice 4 -

1. $D_n = C_{n+1} - C_n = C_{\lfloor \frac{n+1}{2} \rfloor} + C_{\lceil \frac{n+1}{2} \rceil} + (n+1) - 1 - \left(C_{\lfloor \frac{n}{2} \rfloor} + C_{\lceil \frac{n}{2} \rceil} + n - 1\right)$.
   En utilisant l'exercice ci-dessus on obtient
   $D_n = C_{\lfloor \frac{n}{2} \rfloor + 1} - C_{\lfloor \frac{n}{2} \rfloor} - 1 = D_{\lfloor \frac{n}{2} \rfloor} + 1$. $D_0 = C_1 - C_0 = 0$
2. $C_2 = C_1 + C_1 + 1 = 1$ donc $D_1 = C_2 - C_1 = 1$.
   $2^k \leqslant n < 2^{k+1}$ implique $2^{k-1} \leqslant n/2 < 2^k$ donc $L(\lfloor n \rfloor 2) = L(n) - 1$.

On a ainsi $D_n - L(n) = D_{\lfloor \frac{n}{2} \rfloor} + 1 - \big(L(\lfloor n \rfloor 2) + 1\big) = D_{\lfloor \frac{n}{2} \rfloor} - L(\lfloor n \rfloor 2)$: la suite des différence est constante. En divisant par 2 on aboutit à 1 donc $D_n - L(n) = D_1 - L() = 1$.

3. On a $C_1 = 0 = L(1) = 1.L(1) + 1 + 1 - 2^{0+1}$: la propriété est vraie pour $n = 0$.
On suppose que $C_n = nL(n) + n + 1 - 2^{L(n)+1}$ alors
$C_{n+1} = D(n) + C_n = L(n) + 1 + C_n = (n+1)L(n) + n + 2 - 2^{L(n)+1}$.

Si on a $2^p \leqslant n < 2^{p+1} - 2$ alors $2^p < n < 2^{p+1} - 1$ donc $L(n) = L(n+1) = p$ et on a bien $C_{n+1} = (n+1)L(n+1) + (n+1) + 1 - 2^{L(n+1)+1}$.

Pour $n = 2^{p+1} - 1$ alors $n + 1 = 2^{p+1}$ donc $L(n) = p$ et $L(n+1) = p + 1$.
On a aussi $n + 1 = 2^{L(n+1)}$ d'où

$$\begin{aligned} C_{n+1} &= (n+1)\big(L(n+1) - 1\big) + (n+1) + 1 - 2^{L(n+1)} \\ &= (n+1)L(n+1) + (n+1) + 1 - 2^{L(n+1)} - (n+1) \\ &= (n+1)L(n+1) + (n+1) + 1 - 2.2^{L(n+1)} \\ &= (n+1)L(n+1) + (n+1) + 1 - 2^{L(n+1)+1} \end{aligned}$$

Dans les deux cas la formule est vraie pour $n + 1$.
On a démontré la propriété par récurrence.

Solution de l'exercice 5 - On modifie la fusion en place.

```
def fusion_gauche(liste, i, j, p):
    """Entree : une liste et 3 indices
       Requis : i <= p < j, la liste est triée entre i et p
                            et entre p+1 et j
       Sortie : la liste est triée entre i et j"""
    n1 = p - i +1
    l1 = liste[i: p+1]
    pos1 = 0
    pos2 = p+1
    for k in range(i, j+1):
        if pos1 == n1:
            liste[k] = liste[pos2]
            pos2 = pos2 + 1
        elif pos2 == j+1:
            liste[k] = l1[pos1]
            pos1 = pos1 + 1
        elif plusGrand(liste[pos2],l1[pos1]):
            liste[k] = l1[pos1]
            pos1 = pos1 + 1
        else:
            liste[k] = liste[pos2]
            pos2 = pos2 + 1
```

On remplit les positions depuis la droite. On place le plus grand élément restant à placer.

```
def fusion_droite(liste, i, j, p):
    """Entree : une liste et 3 indices
       Requis : i <= p < j, la liste est triée entre i et p
                              et entre p+1 et j
       Sortie : la liste est triée entre i et j"""
    l2 = liste[p+1:j+1]
    pos1 = p
    pos2 = j -p -1
    for k in range(j, i-1, -1):
        if pos1 == i-1:
            liste[k] = l2[pos2]
            pos2 = pos2 - 1
        elif pos2 == -1:
            liste[k] = liste[pos1]
            pos1 = pos1 - 1
        elif plusGrand(liste[pos1],l2[pos2]):
            liste[k] = liste[pos1]
            pos1 = pos1 - 1
        else:
            liste[k] = l2[pos2]
            pos2 = pos2 - 1
```

Solution de l'exercice 6 -

```
def plus_petit_majorant(liste, i, j, x):
    """Entree : une liste et 2 indices et un terme
       Requis : la liste est triée entre i et j, liste[j] > x
       Sortie : le plus petit indice k entre i et j
                tel que liste[k] > x"""
    if i == j:
        return i
    else:
        p = (i+j)//2
        if plusGrand(liste[p], x):
            return plus_petit_majorant(liste, i, p, x)
        else:
            return plus_petit_majorant(liste, p+1, j, x)
```

```
def plus_grand_minorant(liste, i, j, x):
    """Entree : une liste et 2 indices et un terme
       Requis : la liste est triée entre i et j, liste[i] < x
       Sortie : le plus grand indice k entre i et j
                tel que liste[k] < x"""
    if i == j:
        return i
    else:
        p = (i+j)//2
        if plusGrand(x, liste[p+1]):
            return plus_grand_minorant(liste, p+1, j, x)
        else:
            return plus_grand_minorant(liste, i, p, x)
```

```
def fusionInterne(liste, i, j, p):
    """Entree : une liste et 3 indices
       Requis : i <= p < j, la liste est triée entre i et p
                              et entre p+1 et j
       Sortie : la liste est triée entre i et j"""
    if plusGrand(liste[p], liste[p+1]):
        i1 = plus_petit_majorant(liste, i, p, liste[p+1])
        j1 = plus_grand_minorant(liste, p+1, j, liste[p])
        if j1 - p > p - i1:
            fusion_gauche(liste, i1, j1, p)
        else:
            fusion_droite(liste, i1, j1, p)
```

Solution de l'exercice 7 -

```
def triFusion(liste):
    """Entree : une liste
       Sortie : la liste est triée"""
    n = len(liste)
    p = 1
    while p < n:
        m = n//(2*p)
        for k in range(m):
            i = 2*p*k
            fusionInterne(liste, i, i+2*p-1, i+p-1)
        if 2*p*m + p < n:
            fusionInterne(liste, 2*p*m, n-1, 2*p*m+p-1)
        p = 2*p
```

Solution de l'exercice 8 -
Au début du programme on a $k = i = a$ donc $\mathcal{P}(a)$ ne contient aucune condition : il est vérifié.
On suppose que $\mathcal{P}(i)$ est valide au début de la boucle d'indice i avec $i < b - 1$.

1. Si le pivot est plus grand que `liste[i]` on échange les termes d'indices i et k puis on incrémente k, on note k' la nouvelle valeur de k ($k' = k + 1$). `liste[a:k']` est toujours majorée par le pivot car on a ajouté un terme qui vérifie la condition.
   `liste[k':i+1]` comporte les mêmes termes (dans un ordre différent) que l'ancienne liste `liste[k:i]` donc est encore minorée par le pivot. $\mathcal{P}(i + 1)$ est encore valide dans ce cas.
2. Si `liste[i]` majore le pivot alors `liste[a:k]` est inchangée et `liste[k:i+1]` contient un élément de plus que `liste[k:i]` et ce terme est strictement minoré par le pivot aussi.
   $\mathcal{P}(i + 1)$ est valide aussi dans ce cas.

Ainsi $\mathcal{P}(b - 1)$ est vérifié donc tous les éléments d'indices a à $k - 1$ sont majorés par le pivot et tous ceux d'indices k à $b - 1$ sont minorés strictement par le pivot.
L'échange des termes d'indices k et b montre que l'indice renvoyé, k, est compris entre a et b et sépare la liste selon la comparaison au terme `liste[k]`.

Solution de l'exercice 9 -
On prouve que la fonction `triRapideBornes` trie le segment par récurrence sur $b - a$.

- Si $b - a \leqslant 0$ le segment est vide ou a un seul élément et il est inchangé.
  Comme toute liste de 0 ou 1 élément est triée, l'algorithme est correct dans ce cas.
- On suppose que la fonction `triRapideBornes(liste, i, j)` trie le segment en conservant les élément pour tous i et i tels que $j - i \leqslant m$. On considère un segment de a à b avec $b - a = m + 1$.
  La valeur p renvoyé par `decoupe` est comprise entre a et b.
  On a alors $(b - 1) - a \leqslant m$ et $b - (p + 1) \leqslant m$ donc l'hypothèse de récurrence affirme que les segments de a à $p - 1$ et de $p + 1$ à b sont triés après les appels récursifs à la fonction

`triRapideBornes`.
Dans le segment entre a et b on a donc des éléments majorés par `liste[p]` entre a et $p-1$, suivis de `liste[p]` puis suivis par des élément strictement supérieurs : le segment total est trié.
Comme on a modifié la liste uniquement par des échanges, la liste a les mêmes éléments.
La fonction `triRapideBornes` renvoie bien la liste permutée dans l'ordre entre a et b pour toute paire d'indices valides donc, en particulier, pour $a = 0$ et $b = n-1$: `triRapide` est prouvée.

Solution de l'exercice 10 - La complexité du tri d'une liste de taille 1 est 0.
On a établi que la complexité du tri rapide d'une liste entre a et b, $C(a,b)$, vérifie
$C(a,b) = b - a + C(a, p-1) + C(p+1, b)$ où p est la position du pivot.
On remarque que la complexité ne dépend que de la largeur de l'intervalle donc on peut noter $C(a,b) = C'(n)$ avec $n = b - a + 1$.
La relation ci-dessus peut s'écrire $C'(n) = n - 1 + C'(n_1) + C'(n_2)$ avec $n_1 = p - 1 - a + 1 = p - a$ et $n_2 = b - (p+1) + 1 = p - b$ donc $n_1 + n_2 = n - 1$.
On aboutit ainsi à $C'(n) = n - 1 + C'(n_1) + C'(n - 1 - n_1)$ avec $0 \leqslant n_1 \leqslant n - 1$ et $C'(1) = 0$
Si on suppose, par récurrence, que $C'(k) \leqslant \frac{k(k-1)}{2}$ pour $k < n$ on a
$C'(n) \leqslant n - 1 + \frac{n_1(n_1-1)}{2} + \frac{(n-n_1-1)(n-n_1-2)}{2} = \frac{n(n-1)}{2} + n_1(n_1 - n + 1)$.
Or $n_1 \geqslant 0$ et $n - 1 - n_1 \geqslant 0$ d'où $n_1(n_1 - n + 1) \leqslant 0$ puis $C'(n) \leqslant \frac{n(n-1)}{2}$.
Le résultat est prouvé par récurrence.

Solution de l'exercice 11 -

1. 10 et 34
2. `L5 = l4 + [24,17,19,18,22,21,23,20,28,25,27,26,30,29,31,16]`
3. On définit la suites pour $n = 2^{p+1} - 1$ récursivement.
   On calcule la liste pour $2^p - 1$, `l`.
   On veut que le découpage selon le pivot 2^p donne `l` et `l1` avec `l1` obtenue en ajoutant 2^p à chaque élément : dans ce cas les opérations effectuées sur `l` et `l1` seront identiques et on est dans le cas de l'étude.
   Si on place les éléments de `l` puis ceux de `l1` suivis du pivot 2^p le parcours du découpage ne changera rien mais on déplace le pivot à sa position donc il faut envoyer le dernier élément de `l1` en tête.

```
def listeMin(p):
    if p <= 1:
        return [1]
    else:
        l1 = listeMin(p-1)
        pivot = l1[-1]*2
        l2 = [x+pivot for x in l1]
        pivot1 = l2.pop()
        return l1+[pivot1]+l2+[pivot]
```

Solution de l'exercice 12 -

1. On trie la liste, l'élément est le terme d'indice $\left\lfloor \frac{n}{2} \right\rfloor$.
2. Si p est l'indice recherché on sépare la liste. Le pivot se place à l'indice k.
   - Si $k = p$ on a fini.
   - Si $k > p$ on cherche l'élément dans la sous-liste de gauche.
   - Si $k < p$ on cherche l'élément dans la sous-liste de droite.

```
def auxRech(liste,a,b,p):
    """Entrées : une liste et 3 entiers
       Requis : 0 <= a <= p <= b < len(liste)
       Sortie : la valeur du terme d'indice p dans
                la liste triée depuis liste"""
    if b > a:
        k = decoupe(liste,a,b)
        if k == p:
            return liste[k]
        elif k > p:
            return auxRech(liste,a,k-1,p)
        else:
            return auxRech(liste,k+1,b,p)
    else:
        return liste[a]

def recherche(l,p):
    return auxRech(l,0,len(l)-1,p)
```

On fait au plus n comparaisons pour séparer.
Ensuite on recherche dans une liste plus petite.
Si, à chaque étape, on divise ainsi par 2 la taille de la liste le nombre de comparaisons est de l'ordre de $n + n/2 + n/4 + \cdots \sim 2n$.
Bien sur, si la liste est déjà triée on fera de l'ordre de n^2 comparaisons.

Solution de l'exercice 13 -

1. $(n-1)!$
2. On a $C(L_\sigma) = C(g_\sigma) + C(d_\sigma) + n - 1$.
   On note $S_{n,k}$ l'ensemble des permutations telles que $\sigma(n) = k$.
   Les équiprobabilités donnent alors
   $\displaystyle\sum_{\sigma\in S_{n,k}} C(g_\sigma) = (n-1)!T(k-1)$ et $\displaystyle\sum_{\sigma\in S_{n,k}} C(d_\sigma) = (n-1)!T(n-k)$ d'où
   $$\begin{aligned} T(n) &= \frac{1}{n!}\sum_{k=1}^{n}\sum_{\sigma\in S_{n,k}} C(L_\sigma) = \frac{1}{n!}\sum_{k=1}^{n}\sum_{\sigma\in S_{n,k}} (C(g_\sigma) + C(d_\sigma) + n - 1) \\ &= \frac{1}{n!}\sum_{k=1}^{n}((n-1)!T(k-1) + (n-1)!T(n-k) + (n-1)(n-1)!) \\ &= \frac{1}{n}\left(n(n-1) + \sum_{k=1}^{n} T(k-1) + T(n-k)\right) \end{aligned}$$
3. $\displaystyle\sum_{k=1}^{n} T(k-1) = \sum_{i=0}^{n-1} T(i)$ avec $i = k-1$.
   $\displaystyle\sum_{k=1}^{n} T(n-k) = \sum_{i=0}^{n-1} T(i)$ avec $i = n-k$.
4. $\displaystyle nT(n) - (n-1)T(n-1) = 2\sum_{i=0}^{n-1} T(i) + n(n-1) - 2\sum_{i=0}^{n-2} T(i) - (n-1)(n-2)$.
5. En divisant par $n(n+1)$ on obtient $u_n = u_{n-1} + \frac{2(n-1)}{n(n+1)} \leqslant u_{n-1} + \frac{2}{n}$.
6. $T(1) = 0$ donc $u_1 = 0$ puis $\displaystyle u_n \leqslant \sum_{k=2}^{n} \frac{2}{k}$.
   On a $\displaystyle\frac{1}{k} \leqslant \int_{k-1}^{k} \frac{\mathrm{d}t}{t}$ d'où $\displaystyle u_n \leqslant 2\int_{1}^{n} \frac{\mathrm{d}t}{t} = 2\ln(n)$.

Solution de l'exercice 14 -

```
def listeHasard(p,n):
    l = []
    for i in range(p):
        l.append(randint(1,n))
    return l

def listeH(p):
    l = []
    for i in range(p):
        l.append(randint(1,p)*p+i)
    return l
```

Solution de l'exercice 15 -

```
def triKnuth(liste):
    n = len(liste)
    enPlace = [False]*n
    for k in range(n):
        while not enPlace[k]:
            i_pivot = k # i est l'emplacement du pivot
            pivot = liste[k]
            j = k + 1 # j est la position étudiée
            while j < n and not enPlace[j]:
                if liste[j] < pivot:
                    liste[i_pivot] = liste[j]
                    liste[j] = liste[i_pivot + 1]
                    liste[i_pivot + 1] = pivot
                    i_pivot = i_pivot + 1 # pivot déplacé
                j = j + 1
            enPlace[i_pivot] = True
```

Chapitre V

PILES

Résumé

Dans ce chapitre nous allons mettre en évidence dans l'usage de la récursivité une structure de données : les piles. Nous allons ensuite définir et utiliser celle-ci dans d'autres contextes. Une dernière partie définit une autre structure de données : les files.

1 Introduction

1.1 Récursivité

On part de l'algorithme classique du calcul récursif de a^n.

```
def puissance(a,n):
    if n <= 0:
        return 1
    else:
        return a*puissance(a,n-1)
```

On a les évaluations suivantes

```
>>> puissance(1.007, 900)
532.7500489085279
>>> puissance(1.007, 1000)
Traceback (most recent call last):
  File "<console>", line 1, in <module>
  File "<tmp 1>", line 5, in puissance
    return a*puissance(a,n-1)
  File "<tmp 1>", line 5, in puissance
    return a*puissance(a,n-1)
  File "<tmp 1>", line 5, in puissance
    return a*puissance(a,n-1)
  [Previous line repeated 984 more times]
  File "<tmp 1>", line 2, in puissance
    if n <= 0:
RecursionError: maximum recursion depth exceeded in comparison
```

Le message d'erreur est impressionnant par sa longueur.
Il nous parle d'une profondeur maximale de récursivité dépassée.
De quoi s'agit-il ?

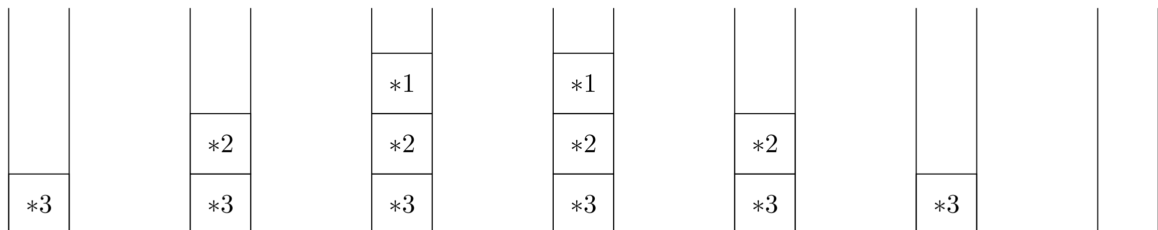
Lors de l'appel de `puissance(1.007, 1000)` la fonction doit calculer `puissance(1.007, 999)`. Pendant qu'elle calcule cette valeur elle doit se souvenir qu'elle doit multiplier le résultat par 1.007, elle met donc cette opération "en attente".
Lorsqu'elle calcule `puissance(1.007, 999)` elle doit aussi considérer le nouveau produit par 1.007 comme inachevé, il faut le mémoriser.
En poursuivant ainsi on voit qu'on doit mettre 1000 fois en attente la multiplication par 1.007.
Dans chaque langage il y a une limite au nombre de calculs que l'on peut mettre en attente. Pour python cette limite est inférieure à 1000.

On détaille cette mise en suspend des calculs avec un autre algorithme récursif.

```
def fact(n):
    if n <= 0:
        return 1
    else:
        return n*fact(n-1)
```

Lors du calcul de `fact(3)` le programme doit calculer `fact(2)` en conservant la multiplication par 3 en mémoire, il écrit donc cette opération en réserve.
De même le calcul de `fact(2)` met le produit par 2 en attente, puis le calcul de `fact(1)` suspend le produit par 1 et on arrive à `fact(0)` qui donne immédiatement 1. On peut alors calculer pas-à-pas le résultat en lisant les calculs mis en attente ; on commence par le dernier calcul mémorisé et on finit lors qu'il n'y a plus rien.

fact(3)	fact(2)	fact(1)	fact(0)	fact(1)	fact(2)	fact(f)
3*fact(2)	2*fact(1)	1*fact(0)	1	1*1	1*2	2*3



1.2 Une structure de données

On voit donc que la récursivité utilise une structure de données qui peut accumuler les données et qui en restitue une à la fois en commençant par la dernière donnée ajoutée. L'acronyme anglais, **LIFO** pour Last In, First Out, exprime bien cette idée que le premier sorti est le dernier entré.

On parlera de **pile** pour ce type de données, ce qui correspond à l'idée d'empiler les données en attente.

Une pile est utilisée dans d'autres contextes.

- Dans un navigateur web, une pile sert à mémoriser les pages Web visitées. L'adresse de chaque nouvelle page visitée est empilée et l'utilisateur dépile l'adresse de la page précédente en cliquant le bouton "Afficher la page précédente."
- L'évaluation des expressions mathématiques en notation post-fixée (ou polonaise inverse) utilise une pile, voir la partie 4.1.
- La vérification du bon parenthésage d'une expression se fait en complexité linéaire avec une pile, voir la partie 4.2.
- La fonction "Annuler la frappe" (Undo) d'un traitement de texte mémorise les modifications apportées au texte dans une pile.

Nous allons donner plusieurs moyens de définir en Python une structure de donnée qui correspond à la pile. Mais, avant de le faire, nous allons définir de manière abstraite ce que l'on veut. Cette méthode permettra de pouvoir passer d'une implémentation à une autre sans se préoccuper des détails internes : toutes les implémentations devront être conformes au cadre imposé. En contrepartie on s'interdira d'utiliser les subtilité de telle ou telle écriture.

2 Type de données abstrait

2.1 Définitions

Le langage python fournit des structures de données comme les listes, tuples et chaînes de caractères, le module `numpy` fournit un type de tableaux. Chaque type de données a ses avantages et ses contraintes propres. Il est judicieux de chercher d'abord la structure de données adaptée à la problématique : choisir une chaîne de caractère, une liste ou un tableau `numpy` peut changer beaucoup d'aspects du programme, comme les complexités spatiale et temporelle du programme.
Nous allons ici renverser la perspective : au lieu d'utiliser des données avec leur caractéristiques imposées par le langage, nous allons définir les propriétés souhaitées et les implémenter dans le langage considéré. Ces propriétés forment le **type de données abstrait**, les implémentations seront un type de données concret.
La spécification d'un type de donnée abstrait est composée :

- de la définition d'un certain nombre d'opérations concernant le type de données,
- des axiomes qui définissent le comportement,
- des préconditions si les opérations sont partiellement définies.

Les fonctions de manipulation des structures de données sont de trois classes :

- les *constructeurs* qui permettent de créer des ensembles et d'ajouter des éléments,
- les *prédicats* qui sont des fonctions dont le résultat est `True` ou `False`,
- les fonctions de *sélection* qui permettent d'extraire certains éléments et informations.

Il existera un choix important à faire lorsque l'on utilisera une structure de données dans laquelle on peut ajouter des éléments.

- Soit on ajoute un élément à un ensemble en conservant l'ensemble, c'est ce que l'on fait par exemple avec la méthode `append` qui modifie une liste sans en créer une nouvelle : on parle de données **mutables**. Dans ce cas la structure est un contenant, ce qui est à l'intérieur peut changer.
- Soit on crée un nouvel ensemble par l'ajout d'un élément : on parle alors de données **persistantes**.
  Dans ce cas la structure est pensée comme l'ensemble des ces éléments, lui ajouter (ou enlever) un élément crée une autre structure.
  Les structures persistantes sont peu utilisées en Python (sauf pour les chaînes de caractères).

2.2 Les piles

Nous allons définir un type mutable pour les piles : pour reprendre l'analogie des piles de dossier, un pile est plutôt un ensemble de dossier qui change de jour en jour plutôt qu'une installation en

art moderne qui représenterait un bureau sans qu'on puisse jamais y modifier quoi que ce soit.

Les spécifications sont assez naturelles.

- Les constructeurs :
  - `create()` qui renvoie une pile ne contenant aucun élément,
  - `push(x, p)` qui ajoute l'élément `x` à la pile `p` ; cette fonction ne renvoie rien.
- Un prédicat :`is_empty(p)` qui renvoie `True` ou `False` selon que la pile `p` contient des élément ou non.
- Les fonctions servent à voir le dernier élément ajouté. Il y a deux cas selon que l'on enlève ou non cet élément.
  - `pop(p)` retire le dernier ajouté à la pile et le renvoie.
  - `top(p)` envoie le dernier élément ajouté à la pile **sans modifier la pile**.
- Ces fonctions ont une précondition : elles ne sont définies que si la pile passée en paramètre est non vide.
- Les axiomes reflètent les propriétés souhaitées.
  - `empty(create())` renvoie `True`
  - `empty(p)` après `push(x, p)` renvoie `False`
  - `pop(p)` après `push(x, p)` renvoie `x` et la suite de ces deux instructions fait retourner `p` à son état initial.

On admet que ces axiomes déterminent de manière unique le comportement des fonctions.

3 Piles concrètes

3.1 Utilisation de listes

Les listes Python donnent, de manière naturelle, une structure de pile.

```
def create():
    return []

def push(x, pile):
    pile.append(x)

def is_empty(pile):
    preturn pile == []

def pop(pile):
    return pile.pop()

def top(pile):
    return pile[-1]
```

3.2 Utilisation de tableaux numpy

Une pile peut aussi être implémentée dans un tableau de taille fixée comme le sont les tableaux `numpy`.
On supposera que le module `numpy` a été importé

```
import numpy as np
```

On remplit les positions libres à partir de 0 ; il faut conserver la position de la dernière case occupée ou de la première case libre. La taille maximale sera fixée à l'avance : on prend le risque d'un dépassement de capacité comme il se produit avec la pile de récursivité.
On choisit ici de considérer la pile comme un couple dont la première composante est le nombre d'éléments dans la pile, c'est l'indice de la première position libre, et la seconde composante est le tableau.

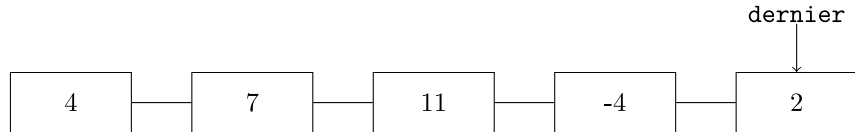
Programme V.1 – Fonctions de piles avec un tableau

```
def create():
    nMax = 1000
    tableau = np.zeros(nMax)
    return [0, tableau]

def is_empty(pile):
    return pile[0] == 0

def push(x, pile):
    n, tableau = pile
    nMax = len(tableau)
    if n < nMax:
        tableau[n] = x
        pile[0] = n + 1
    else:
        print("La pile est pleine")

def pop(p):
    n, tableau = pile
    if n > 0:
        x = tableau[n-1]
        pile[0] = n - 1
        return x
    else:
        print("La pile est vide")

def pop(p):
    n, tableau = pile
    if n > 0:
        return tableau[n-1]
    else:
        print("La pile est vide")
```

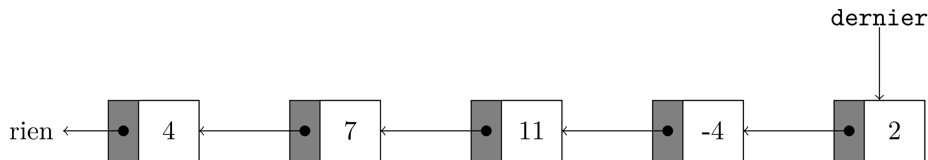
3.3 Utilisation d'objets

Nous allons ici ébaucher l'utilisation de la programmation objet.

On peut voir une pile comme une chaîne de valeurs dont on ne connaît que la dernière



Quand on ajoute un élément, il devient le dernier.
Pour retirer un élément il faut retrouver le précédent ; chaque élément devra donc être associé à son précédent



Dans les langages à objet comme Python on peut définir naturellement une structure de pile.
C'est une **classe** doit contenir une valeur et la pile "en dessous".
On peut remarquer que la définition est récursive.
Lorsque l'élément est le premier on indique `None` pour signifier qu'il n'y a rien en dessous : `None` est donc l'élément terminal.
La classe est définie par méthode de création, appelée `__init__`

```
class Pile:
    def __init__(self, x):
        self.valeur = x
        self.avant = None
```

Le nom `self` pour la variable est unsuel.
On créera un élément par `e = Pile(5)`, par exemple.
Lors de sa création l'élément n'a pas de prédécesseur.
La pile de l'exemple ci-dessus peut être définie par `e5`

```
e1 = Element(4)
e2 = Element(7)
e2.avant = e1
e3 = Element(11)
e3.avant = e2
e4 = Element(-4)
e4.avant = e3
e5 = Element(2)
e5.avant = e4
```

C'est en fait une pile persistante, quand on ajoute un élément on crée une nouvelle pile.
Pour définir une pile mutable il suffit de définir un contenant de pile.
Ce sera aussi une classe, c'est un objet qui réfère au dernier élément.

```
class Stack:
    def __init__(self):
        self.dernier = None
```

Ici une pile créée sera supposée vide.

Pour ajouter une valeur x à une pile

- on crée l'élément associé à x
- son prédécesseur est l'élément associé à la pile
- la pile est maintenant associée à ce nouvel élément.

Programme V.2 – Fonctions de piles avec des objets

```
def create():
    return Stack()

def is_empty(pile):
    return pile.dernier is None

def push(x, pile):
    e = Pile(x)
    penult = pile.dernier
    e.avant = penult
    pile.dernier = e

def pop(pile):
    if is_empty(pile):
        print("La pile est vide")
    else:
        bout = pile.dernier
        pile.dernier = bout.avant
        return bout.valeur

def top(pile):
    if is_empty(pile):
        print("La pile est vide")
    else:
        return pile.dernier.valeur
```

4 Applications

Dans cette partie on donne, sous forme d'exercices, quelques applications des piles.

4.1 Parenthésages

On veut étudier le parenthésage d'expressions arithmétiques ou du code source d'un programme. Il est utile de vérifier que les parenthèses sont bien écrites et aussi de déterminer la parenthèse ouvrante associée à une parenthèse fermante.
Par exemple $1 + 2 * (7 - (4 - 3) * ((2 - 5) + 2 * ((12/4 - 8) + 2 * 3)))$ est bien parenthésée et la parenthèse ouvrante après $(4 - 3)*$ est associée à l'avant dernière parenthèse fermante.
On considère une chaîne de caractère représentant l'expression et on s'intéresse uniquement aux parenthèses classiques "(" et ")".
Pour savoir si une expression est bien parenthésée il faut vérifier qu'il y a autant de parenthèses ouvrantes que de parenthèses fermantes et que chaque parenthèse fermante est associée à une parenthèse ouvrante placée avant elle. Celle-ci est la dernière parenthèse ouvrante du texte non encore fermée. On voit apparaître l'utilité d'une pile.

- On lit les caractères un par un,
- quand on voit une parenthèse ouvrante on empile sa position
- quand on lit une parenthèse fermante on dépile un élément, il sera la position de la parenthèse ouvrante associée.
- Si on doit dépiler alors que la pile est vide ou si la pile est non vide à la fin c'est que l'expression n'est pas bien parenthésée.

Dans la chaîne "1+2*(7-(4-3)*((2-5)+2*((12/4-8)+2*3)))" correspondant à l'expression ci-dessus les opérations de la pile seront

- on empile 4,
- on empile 7
- on dépile 7 qui est associé à 11
- on empile 13
- on empile 14
- on dépile 14 qui est associé à 18
- on empile 22
- on empile 23
- on dépile 23 qui est associé à 30
- on dépile 22 qui est associé à 35
- on dépile 13 qui est associé à 36
- on dépile 4 qui est associé à 37.

On peut donner le résultat sous la forme d'une liste de couples, pour l'exemple ci-dessus on aurait, comme résultat, la liste
`[(7, 11), (14, 18), (23, 30), (22, 35), (13, 36), (4, 37)]`.

Exercice 1 — Liste des parenthèses associées
Écrire une fonction `listePar` *qui reçoit une expression* **supposée bien parenthésée** *et qui retourne la liste des couples d'indices de parenthèses associées.*

Exercice 2 — Test de parenthésage
Écrire une fonction `bienPar` *qui reçoit une expression qui n'est plus supposée bien parenthésée et qui renvoie* `True` *ou* `False` *selon que la liste est bien parenthésée ou non.*

Exercice 3 — Liste des parenthèses associées et parenthèse isolée
Modifier la fonction `listePar` *pour qu'elle renvoie une liste de couples contenant tous les indices de parenthèses même si le texte n'est pas bien parenthésé.*
Une parenthèse fermante qui n'est pas associée à une parenthèse ouvrante sera renvoyée sous la forme sous la forme `(-1 ,i)` *et une parenthèse ouvrante non associée à une parenthèse fermante sera renvoyée sous la forme* `(i, -1)`.
Par exemple `listePar("(1+2))*((5-3)")` renverra `[(0, 4), (-1, 5), (9, 12), (8, -1)]`,

4.2 Notation post-fixée (ou polonaise inversée)

En 1920 le mathématicien polonais Jan Lukasiewicz propose la notation pré-fixé (ou polonaise) qui consiste à considérer les opérations comme des opérateurs à 2 variables, $7+11$ s'écrit $+\ 7\ 11$.
L'avantage est que les parenthèses deviennent inutiles :
$(7-2).\sin(2x+\pi/3)$ devient $*\ -\ 7\ 2\ \sin\ +\ *\ 2\ x\ /\ \pi\ 3$.
La notation polonaise inverse ou notation post-fixée a été proposée par le philosophe et informaticien australien Charles Leonard Hamblin dans le milieu des années 1950.
Elle a été diffusée dans le public comme interface utilisateur avec les calculatrices de bureau de Hewlett-Packard (HP-9100), puis avec la calculatrice scientifique HP-35 en 1972.
Elle consiste simplement à placer l'opérateur **après** les deux opérande, $7+11$ s'écrit $7\ 11\ +$.
L'expression $32-2(12-3(7-2))$ peut s'écrire $32\ 2\ 12\ 3\ 7\ 2\ -\ *\ -\ *\ -$
L'avantage est que, combinée à une pile, cette notation permet d'effectuer les calculs sans faire référence à une quelconque adresse mémoire.

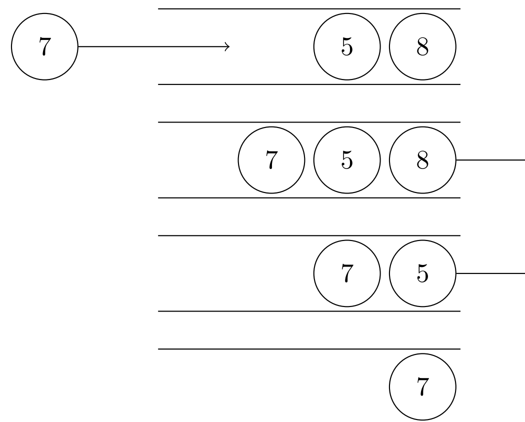
- On lit l'expression terme-à-terme :
- si on lit une valeur numérique, elle est empilée,
- si on lit une opération $\clubsuit$,
  on dépile les deux derniers opérandes a et b
  et on empile le résultat du calcul $a\clubsuit b$.

Exercice 4 — Un exemple
Simuler l'évolution de la pile avec l'expression $32\ 2\ 12\ 3\ 7\ 2\ -\ *\ -\ *\ -$

Exercice 5 — Simulation d'une calculatrice
Écrire une fonction `npi(liste)` *qui prend en entrée une liste contenant des valeurs numériques et des chaînes* `'+'`, `'-'`, `'*'`, `'/'` *et qui calcule la valeur de l'expression en NPI. On considère que les nombres sont des flottants, la division est la division réelle classique.*
Pourquoi est-il difficile de lire directement une chaîne de caractères pour l'évaluer ?

4.3 Files d'attente

Une *file* (*queue* en anglais) est une structure de données basée sur le principe : premier entré, premier sorti, en anglais **FIFO** (First In, First Out), ce qui veut dire que les premiers éléments ajoutés à la file seront les premiers à en être extraits.



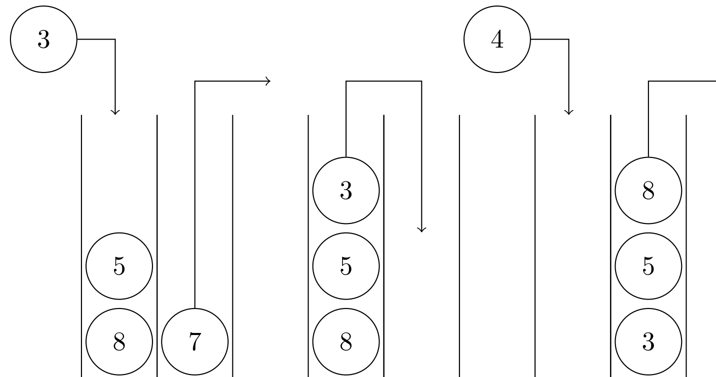
Exemples

- Toutes les situations où l'on est servi en fonction de l'ordre d'arrivée sont des files d'attente : passage à la cantine, à un guichet d'administration, au cinéma ...
- Les serveurs d'impression, qui doivent traiter les requêtes dans l'ordre dans lequel elles arrivent, les insèrent dans une file d'attente.
- Plus généralement on utilise des files pour créer toutes sortes de mémoires tampons (en anglais buffers).

On voit que l'on ajoute les éléments comme dans une pile et qu'on les retire comme dans une pile inversée. On pourrait utiliser une pile en la retournant à chaque fois qu'on a besoin de retirer un élément puis en la reconstituant mais le nombre d'opérations serait trop important.
Il existe une méthode astucieuse qui utilise 2 piles :

- on ajoute dans la première,
- on retire depuis la seconde,
- quand la seconde est vide on retourne la première dans la seconde.

Le retournement peut sembler coûteux mais est amorti : chaque élément ne sera retourné qu'une fois.



Les files sont définies par un type abstrait. En voici les spécifications.

- Les constructeurs : `createQueue()` qui renvoie une file ne contenant aucun élément et `enque(x, file)` qui ajoute l'élément `x` à la file `file` ; cette fonction ne renvoie rien.
- Un prédicat `emptyQueue(file)` qui renvoie `True` ou `False` selon que la file contient des élément ou non.
- La fonction de sélection est `deque(file)` qui retire un élément à la file (celui qui a été ajouté le premier parmi ceux qui restent) et qui le renvoie.

Exercice 6 — Commutativité
Montrer que les deux suites d'instructions suivantes donnent le même résultat pour une même file **non vide** *: les valeurs de* `y` *et l'état de la file* `file` *sont les mêmes à la suite des deux cas.*

```
enque(x,file)
y = deque(file)
```

```
y = deque(file)
enque(x,file)
```

Que se passe-t-il si la file est vide au départ ?

Exercice 7 — Implémentations d'une file à l'aide de deux piles
Donner des codes python pour les fonctions `createQueue`*,* `emptyQueue`*,* `enque` *et* `deque` *en n'employant que les fonctions des piles.*

5 Exercices

5.1 Utilisation des fonctions de piles

Dans les exercices suivants on demande quelques manipulation simples des piles en n'utilisant que les fonctions données par la spécification des piles.

Exercice 8 — Échange des sommets
Écrire une fonction qui intervertit les deux éléments situés au sommet d'une pile de taille au moins égale à 2.

Exercice 9 — Pile inversée
Écrire une fonction qui retourne une pile contenant les éléments de la pile passée en paramètre dans l'ordre inverse. La pile initiale peut être modifiée.

Exercice 10 — Copie d'une pile
Écrire une fonction qui retourne une pile égale à la pile passée en paramètre.
La pile initiale doit rester intacte.

Exercice 11 — Longueur d'une pile
Écrire une fonction `taille` *qui renvoie le nombre d'éléments dans une pile.*
La pile doit rester intacte.

Exercice 12 — n-ième élément d'une pile
Écrire une fonction `element(n, pile)` *qui renvoie le n-ième élément d'une pile et l'enlève de la pile. Les autres éléments de la pile doivent être les mêmes.*

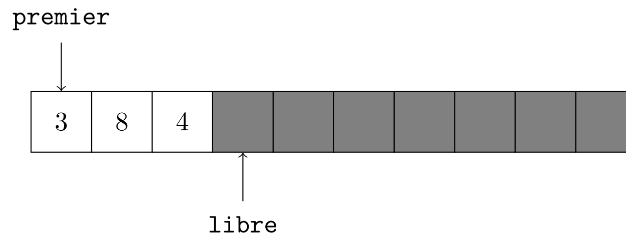
5.2 Implémentations des files par des tableaux

Une idée possible pour implémenter les files est celle des tickets que l'on retire à un distributeur et qui donnent l'ordre de passage aux service souhaité [1].
On maintient un tableau de taille fixée, c'est le nombre de ticket possibles, et deux entiers.
Le premier correspond au numéro du prochain ticket appelé, on notera `"premier"` cet indice
le second est le numéro disponible dans le distributeur, on notera `"libre"` cet indice.
On a donc une liste
Pour donner une représentation lisible de ces 3 éléments nous allons utiliser un **dictionnaire** : ce sera une famille de trois composantes mais celles-ci seront accessibles par leur nom plutôt que par leur indice : par exemple `"données"`, `"premier"` et `"libre"`.

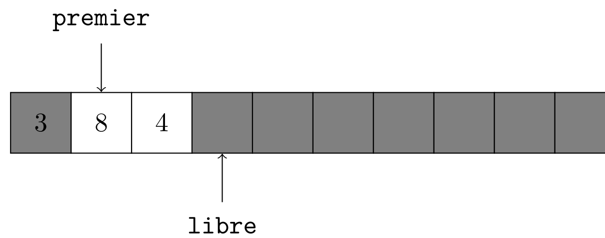
1. Guichet administratif, fromager ou poissonnier dans un supermarché, ...

Exemple On suppose qu'on a 10 emplacements au maximum.

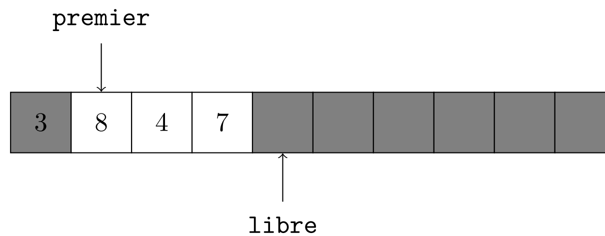
- On a ajouté 3 éléments : 3 puis 8 puis 4.



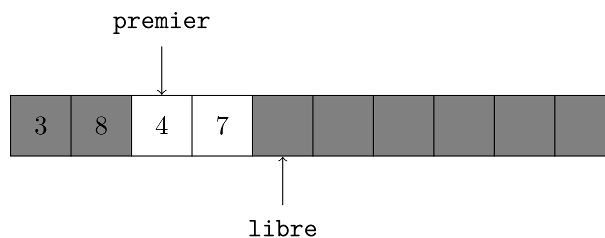
- On retire un élément (cela renvoie 3).



- On ajoute 7.



- On retire un élément (cela renvoie 8).



On peut créer un dictionnaire avec des composantes sous la forme

```
dico = {"données" : [0]*nMax, "premier" : 0, "libre" : 0}
```

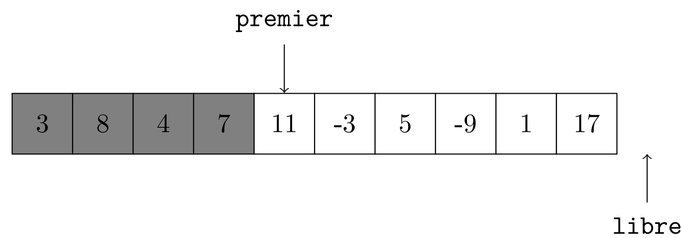
Les composantes seront accessibles respectivement par `dico["données"]`, `dico["premier"]` ou `dico["libre"]`.

Exercice 13 — Implémentations d'une file à l'aide d'une liste
Donner des codes python pour les fonctions `createQueue`*,* `emptyQueue`*,* `enque` *et* `deque` *avec le modèle décrit ci-dessus.*

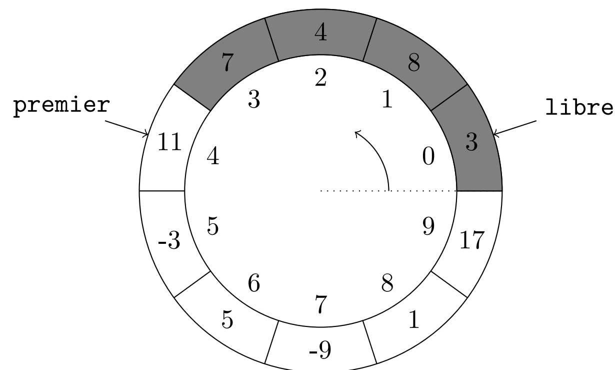
5.3 Implémentations des files par des tableaux "circulaires"

Dans l'implémentation ci-dessus on a le problème, déjà vu avec les piles, d'une taille maximale.
Cependant la situation est plus gênante ici : on peut avoir une file qui est vide est pleine à la fois, cela correspond au cas où toutes les positions ont été attribués et traités : on a alors `dico["premier"]` et `dico["libre"]` égaux à la taille du tableau.
La solution pratique est de recommencer les numéros d'attente au début (on remet un nouveau distributeur de tickets).

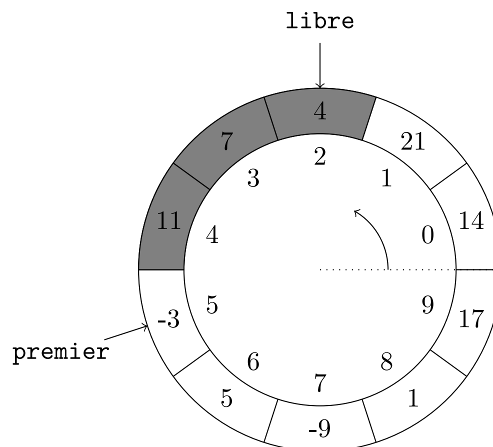
Dans l'exemple ci-dessus on suppose qu"on a ajouté encore les éléments 11, -3, 5, -9, 1, 17 et retiré deux fois un élément (cela renvoie 4 puis 7). On arrive au tableau



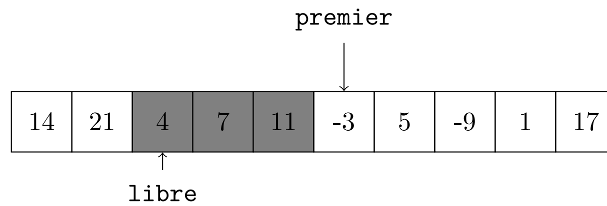
Pour pouvoir ajouter un élément on considère que le tableau est circulaire et que la place libre correspond à la position 0.



On peut alors ajouter des éléments, 14 et 21 par exemple et en retirer un (on obtient 11).



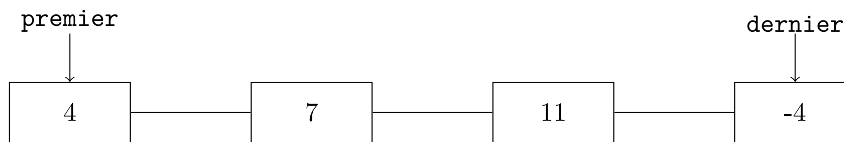
Ce qui correspond au tableau



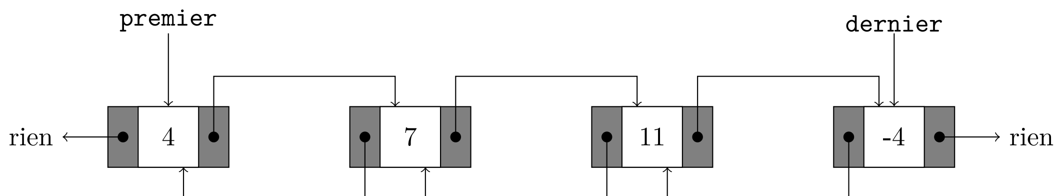
Exercice 14 — Implémentations d'une file à d'une liste circulaire
Donner des codes python pour les fonctions `createQueue`*,* `emptyQueue`*,* `enque` *et* `deque` *avec le modèle décrit ci-dessus.*

5.4 Implémentations des files par des objets python

On peut voir une file comme une chaîne de valeurs dont les deux extrémités sont accessibles.



En plus de la structure vue pour les piles il faut pouvoir enlever le premier élément donc chaque élément de la file doit accéder à l'élément suivant.



On modifie donc les éléments et on définit la classe des files.

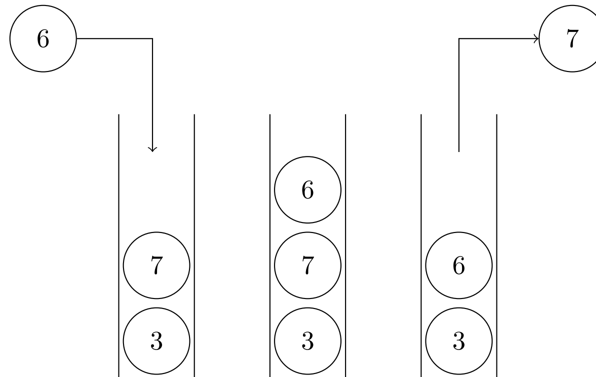
```
class ElementFile:
    def __init__(elem,x):
        elem.valeur = x
        elem.suiv = None
        elem.prec = None

class File:
    def __init__(file):
        file.premier = None
        file.dernier = None
```

Comme on a deux position il faudra gérer spécifiquement les cas où on ajoute un élément à une file vide et où on retire son élément d'une file réduite à un élément.

Exercice 15 — Implémentations d'une file à l'aide d'objets
Donner des codes python pour les fonctions `createQueue`*,* `emptyQueue`*,* `enque` *et* `deque` *avec le modèle décrit ci-dessus.*

5.5 File de priorité

Une **file de priorité** est une collection de données où les éléments sont comparables deux à deux. On souhaite enlever le plus grand (ou le plus petit) lorsque l'on retire un élément. C'est la structure qui permet de gérer les traitements de données en tenant compte de priorités.
Par exemple les tâches qu'un processeur accomplit ont une priorité : le traitement d'un fichier à imprimer est moins important que l'affichage d'un film. On retrouve aussi la notion de file de priorité dans le traitement des urgences dans hôpital.
En informatique certains algorithmes, dits gloutons, doivent choisir à chaque étape un élément qui réalise un optimum, ils utiliseront souvent des files de priorités.



Dans la pratique on utilisera des couples `(x ,k)` où x est un élément et k sa priorité.
Opérations :
- `creerFP()` : retourne une file vide
- `ajouterFP(x, k, fileP)` : ajoute un élément x de priorité k dans la file f.
- `enleverFP(fileP)` : supprime un élément de priorité maximale de la file et le renvoie.
- `videFP(fileP)` : renvoie `True` si la file est vide, `False` sinon.

On va utiliser des tableaux pour implémenter cette structure. On a alors un choix à faire :
- on peut trier les éléments lors de l'adjonction, l'extraction est alors facilitée,
- on peut adjoindre simplement les éléments, il faudra alors chercher le maximum lors de l'extraction.

Chacune de ces deux options demande de parcourir la liste dans une des deux fonctions. Quand la file de priorité a n éléments la complexité est donc de l'ordre de n. Il existe des structures de données plus sophistiquées qui permettent un accès plus rapide, de complexité $\mathcal{O}\big(\ln(n)\big)$.

Exercice 16 — File de priorité avec des listes triées
En s'inspirant de la fonction d'insertion vue lors du tri par insertion implémenter une structure de file de priorité qui utilise des listes triées.

Exercice 17 — File de priorité avec des listes non triées
En utilisant une fonction de recherche du maximum implémenter une structure de file de priorité.

5.6 Ensembles

Un **ensemble** est une collection de données où chaque élément ne peut apparaître qu'une seule fois. L'ordre d'entrée des éléments n'est pas pris en compte. Si on veut enlever un élément, il faut préciser sa valeur. **Opérations :**

- `ensembleVide()` : retourne un ensemble vide
- `videEns(e)` : renvoie vrai si l'ensemble est vide, faux sinon
- `cardinal(e)` : renvoie le nombre d'éléments dans l'ensemble.
- `appartient(x, e)` : renvoie `True` si $x \in e$ et `False` sinon.
- `ajouterEns(x, e)` : ajoute un élément x dans l'ensemble e s'il n'était pas déjà un élément de e.
- `enleverEns(x ,e)` : supprime un élément x passé en entrée d'un ensemble e. Si x n'appartenant pas à e, e est inchangé.
- `ensVersListe(e)` : transforme l'ensemble en liste afin de pouvoir parcourir les éléments.

Exercice 18 — Ensemble avec des listes
Implémenter une structure d'ensemble avec des listes.

Il arrivera parfois que les ensembles considérés soient des sous-ensemble d'un ensemble fixé. On considérera que l'ensemble de base est $\{0, 1, 2, \ldots, n-1\}$ et on pourra représenter un sous-ensemble par un tableau de booléens :
`e[i]` vaut `True` si et seulement si i appartient à e.

Exercice 19 — Ensemble avec des tableaux de booléens
Implémenter une structure d'ensemble comme sous-ensembles de $\{0, 1 \ldots, n-1\}$.
La fonction `ensembleVide` *admettra la taille de l'ensemble initial comme paramètre optionnel.*

6 Solutions

Solution de l'exercice 1 -

```
def listePar(ch):
    n = len(ch)
    l = []
    p = create()
    for i in range(n):
        if ch[i] == "(":
            push(i, p)
        if ch[i] == ")":
            j = pop(p)
            l.append((j,i))
    return l
```

Solution de l'exercice 2 - Il suffit de compter les parenthèses ouvrantes non encore fermées ; on n'a pas besoin d'une pile.

```
def bienPar(ch):
    n = len(ch)
    ouvrantes = 0
    for i in range(n):
        if ch[i] == "(":
            ouvrantes = ouvrantes + 1
        if ch[i] == ")":
            if ouvrantes == 0:
                return False
            else:
                ouvrantes = ouvrantes - 1
    return ouvrantes == 0
```

Solution de l'exercice 3 -

```
def listePar(ch):
    n = len(ch)
    l = []
    p = create()
    for i in range(n):
        if ch[i] == "(":
            push(i, p)
        if ch[i] == ")":
            if empty(p):
                l.append((-1,i))
            else:
                j = pop(p)
                l.append((j,i))
    while not(empty(p)):
        j = pop(p)
        l.append((j,-1))
    return l
```

Solution de l'exercice 4 - Après avoir empilé les nombres on lit les opérateurs.

- La pile contient 32, 2, 12, 3, 7, 2.

- On dépile 2 puis 7, on calcule $7 - 2 = 5$ (attention à l'ordre), on empile 5.
- La pile contient 32, 2, 12, 3, 5.
- On dépile 5 puis 3, on calcule $3.5 = 15$, on empile 15.
- La pile contient 32, 2, 12, 15.
- On dépile 15 puis 12, on calcule $12 - 15 = -3$, on empile -3.
- La pile contient 32, 2, -3.
- On dépile -3 puis 2, on calcule $2.(-3) = -6$, on empile -6.
- La pile contient 32, -6.
- On dépile -6 puis 32, on calcule $32 - (-3) = 38$, on empile 38.
- La pile contient 38 qui est le résultat renvoyé.

Solution de l'exercice 5 -

```
def npi(liste):
    p=creerPile()
    for x in liste:
        if x == '+' :
            b = pop(p)
            a = pop(p)
            push(a + b, p)
        elif x == '-' :
            b = pop(p)
            a = pop(p)
            push(a + b, p)
        elif x == '*' :
            b = pop(p)
            a = pop(p)
            push(a * b, p)
        elif x == '/' :
            b = pop(p)
            a = pop(p)
            push(a . b, p)
        else:
            push(x, p)
    return pop(p)
```

Le problème de la lecture directe d'une chaîne de caractères est le double sens du signe moins : il est à la fois une opération binaire et un changement de signe.

Solution de l'exercice 6 - Si `file` est la file dont les éléments sont, dans l'ordre d'insertion, x_0, x_1, …, x_{n-1} avec $n \geqslant 1$ alors les deux instructions donnent $y = x_0$ et la pile est composée de x_1, …, x_{n-1}, x à la fin.
Si la pile est vide les premières instruction donnent $y = x$ et la pile est vide tandis que les autres renvoient une erreur.

Solution de l'exercice 7 -

```
def createQueue():
    return (create(), create())
```

```
def emptyQueue(file):
    entree, sortie = file
    return empty(entree) and empty(sortie)
```

```
def enque(x, file):
    push(x, f[0])
```

```
def deque(file):
    if emptyQueue(file):
        print('La file est vide')
    else :
        entree, sortie = file
        if empty(sortie):
            while not empty(entree, ):
                push(pop(entree), sortie)
        return pop(sortie)
```

Solution de l'exercice 8 -

```
def inverseSommet(pile):
    """Entrée : une pile
       Sortie : la pile est inchangée si elle contient 0 ou 1
          élément
             les deux premiers éléments sont inversés sinon
                """
    if not is_empty(pile):
        a = pop(pile)
        if is_empty(pile):
            push(a,pile)
        else:
            b = pop(pile)
            push(b, pile)
            push(a, pile)
```

Solution de l'exercice 9 -

```
def inversion(pile):
    elip = create()
    while not is_empty(pile):
        x = pop(pile)
        push(x, elip)
  return elip
```

Solution de l'exercice 10 - On crée une copie inversée de la pile et on en dépile les éléments dans la pile initiale et dans une pile créée.

```
def copie(pile):
    elip = inversion(pile)
    copie = create()
    while not is_empty(elip):
        x = pop(pile)
        push(x, pile)
        push(x, copie)
  return copie
```

Solution de l'exercice 11 - On doit inverser la pile puis on reconstitue la pile initiale en comptant les éléments.

```
def taille(pile):
    n = 0
    elip = inversion(pile)
    while not is_empty(elip):
        x = pop(elip)
        push(x, pile)
        n = n + 1
  return n
```

Solution de l'exercice 12 -

```
def element(n, pile):
    elip = create()
    for i in range(n-1):
        x = pop(pile)
        push(x, elip)
    x_n = pop(pile)
    for i in range(n-1):
        x = pop(elip)
        push(x, pile)
  return x_n
```

Solution de l'exercice 13 - Pour comprendre le cas d'une liste vide on peut enlever tous les éléments et remarquer qu'alors `dico["premier"] = dico["libre"]`. On peut donc initialiser avec la valeur 0 pour `premier` et pour `libre`.

```
def createQueue():
    nMax = 1000
    liste = [0]*nMax
    return {"données" : liste, "premier" : 0, "libre" : 0}
```

```
def emptyQueue(file):
    return file["premier"] == file["libre"]
```

```
def enque(x, file):
    nMax = len(file["données"])
    k = file["libre"]
    if k == nMax:
       print('La file est pleine')
    else:
        f["données"][k] = x
        f["libre"] = k + 1
```

```
def deque(file):
    if emptyQueue(file):
        print('La file est vide')
    else:
        k = file["premier"]
        x = file["données"][k]
        f["premier"] = k + 1
        return x
```

Solution de l'exercice 14 - Il risque d'y avoir ambiguïté entre la notion de pile vide et pleine si la fin est une case avant le début. On "sacrifie" une case et on indique que la liste est vide si `dico["premier"] = dico["libre"]` et que la liste est pleine si `dico["premier"] = dico["libre"] +1`. Les calculs doivent être fait avec le reste de la division par la taille du tableau.

```
def createQueue():
    nMax = 1000
    liste = [0]*nMax
    return {"données" : liste, "premier" : 0, "libre" : 0}
```

```
def emptyQueue(file):
    return file["premier"] == file["libre"]
```

```
def enque(x, file):
    nMax = len(file["données"])
    k = file["libre"]
    p = file["premier"]
    if p== (k + 1)%nMax:
       print('La file est pleine')
    else:
        f["données"][k] = x
        f["libre"] = (k + 1)%nMax
```

```
def deque(file):
    nMax = len(file["données"])
    if emptyQueue(file):
        print('La file est vide')
    else:
        p = file["premier"]
        x = file["données"][p]
        f["premier"] = (p + 1)%nMax
        return x
```

Solution de l'exercice 15 -

```
def createQueue():
    return File()
```

```
def emptyQueue(file):
    return file.premier is None
```

```
def enque(x, file):
    e = ElementFile(x)
    if emptyQueue(file):
        file.premier = x
    else:
        penult = file.dernier
        e.prec = penult
        penult.suiv = e
    file.dernier = e
```

```
def deque(file):
    if emptyQueue(file):
        print("La file est vide")
    else:
        prem = file.premier
        if prem.suiv == None:
            file.dernier = None
        file.premier = prem.suiv
        return prem.valeur
```

Solution de l'exercice 16 -

```
def creerFP() :
    return []
```

```
def videFP(fileP):
    return f == []
```

```
def ajouterFP(x, fileP):
    fileP.append(x)
    i = len(f) -1
    while i > 0 and fileP[i][1] < fileP[i-1][1]:
        fileP[i] = fileP[i-1]
        i = i - 1
    fileP[i] = x
```

```
def enleverFP(fileP):
    if videFP(fileP):
        return "La file est vide"
    else:
        return fileP.pop()
```

Solution de l'exercice 17 -

```
def creerFP() :
    return []
```

```
def videFP(fileP):
    return f == []
```

```
def ajouterFP(x, fileP):
    fileP.append(x)
```

```
def enleverFP(fileP):
    if videFP(fileP):
        return "La file est vide"
    else:
        n = len(fileP)
        max = fileP[0][1]
        iMax = 0
        for i in range(1,n):
            x, k = fileP[i]
            if k > max:
```

```
                max = k
                imax = i
        echange(fileP, iMax, n-1)
        return fileP.pop()
```

Solution de l'exercice 18 -

```
def ensembleVide() :
    return []
```

```
def videEns(e):
    return e == []
```

```
def cardinal(e):
    return len(e)
```

```
def appartient(x, e):
    for y in e:
        if x == y:
            return True
    return False
```

```
def ajouterEns(x, e):
    if not appartient(x,e):
        e.append(x)
```

```
def enleverEns(x, e):
    n = len(e)
    continue  = True
    i = 0
    while i < n and continue:
        if e[i] == x:
            echange(e, i, n-1)
            e.pop()
            continue  = False
```

```
from copy import deepcopy
def ensVersListe(e):
    return deepcopy(e)
```

Solution de l'exercice 19 -

```
def ensembleVide(n=1000) :
    return [False]*n
```

```
def cardinal(e):
    n = len(e)
    card = 0
    for i in range(n):
        if e[i]:
            card = card + 1
    return card
```

```
def estEnsVide(e):
    return cardinal(e) == 0
```

```
def ajouterEns(x, e):
    e[x] = True
```

```
def enleverEns(x, e):
    e[x] = False
```

```
def ensVersListe(e):
    n = len(e)
    return [i for i in range(n) if e[i]]
```

Deuxième partie

Révisions de première année

Chapitre R-I

Modules

Une des richesses principales de Python est son système de modules qui permet d'ajouter de nombreuses fonctionnalités au langage de base. Nous allons présenter dans ce chapitre les modules employés en première année (et d'autres aussi). Une partie du contenu provient des fiches de documentation fournies lors de l'oral de mathématiques du concours Centrale-Supélec

1 numpy

Le module **numpy** sera très utilisé :

```
import numpy as np
```

Il définit un type de données, **array**, qui ressemble aux listes python avec les différences :
- la longueur est fixée, on ne peut pas utiliser la méthode `append`,
- le type des composantes est unique et fixé à l'avance, tous les éléments seront soit des flottants, soit des entiers,
- les calculs sont beaucoup plus efficaces
- les tableaux peuvent être additionnés terme-à-terme comme des vecteurs,
- on peut appliquer une fonction à un tableau, cela applique la fonction à tous les termes,
- en particulier on peut multiplier un tableau par un scalaire, cela multiplie chaque composante,
- dans le cas des matrices on peut simplifier l'accès aux composantes : la notation `m[i][j]` est toujours disponible mais on peut utiliser `m[i,j]`.

```
>>> a = np.array([1.0, 3.5, -1.2])
>>> 2*a*a - 3
array([-1.  , 21.5 , -0.12])
```

Les fonctions utilisables sont celles définies par le module : on n'utilisera plus le module `math`. Pour utiliser une fonction que l'on aurait écrit [1] on la convertit en une fonction `numpy` avec `vectorize`.

```
def signe1(x):
    if x > 0:
        return 1
    elif x == 0:
        return 0
    else:
        return -1
```

```
>>> signe = np.vectorize(signe1)
>>> signe(a)
array([ 1,  1, -1])
```

1. si elle contient des instruction `if`, `for`, `while`, ...

1.1 Création

- la fonction `np.array` convertit les objets python en tableaux numpy,
- la fonction `np.zeros(n)` crée un tableau de taille n rempli de 0,
- la fonction `np.zeros(n, p)` crée une matrice de taille $n \times p$ remplie de 0,
- la fonction `np.ones` est semblable à `np.zeros` mais remplit avec des 1,
- la fonction `np.linspace(a, b, n)` crée un tableau de taille n avec des valeurs de a à b également espacées,
- la fonction `np.arange(a, b, r)` crée le tableau des valeurs qui commencent par a, espacées de r et qui ne dépassent pas b,
- la fonction `np.eye(n)` crée une matrice identité de taille $n \times n$,
- la fonction `np.diag(liste)` crée une matrice diagonale dont les termes diagonaux sont donnés par la liste,
- la fonction `np.copy(a)` crée une copie indépendante de l'original.

1.2 Opérations matricielles

La méthode `shape` (ou la fonction `np.shape`) donne la taille d'une matrice : nombre de lignes, nombre de colonnes. On peut redimensionner une matrice, sans modifier ses termes, à l'aide de la méthode `reshape`.

```
A = np.array([[1, 2, 3], [4, 5, 6]])
>>> A.shape
(2, 3)
>>> A = A.reshape((3, 2))
>>> A
array([[1, 2], [3, 4], [5, 6]])
```

Le produit matriciel s'écrit avec la méthode `dot`, `a.dot(b)`, ou par la fonction `np.dot`, `np.dot(a, b)`.
Le produit vectoriel de deux tableaux de taille 3 est `np.cross`.
La transposée s'obtient avec la fonction `np.transpose` ou, plus simplement, avec la méthode `T` : `A.T`.
La fonction `np.concatenate` permet de créer des matrices par blocs en superposant (`axis=0`) ou en plaçant côte à côte (`axis=1`)) plusieurs matrices.

```
>>> A = np.ones((2,3))
>>> B = np.zeros((2,3))
>>> np.concatenate((A,B), axis=0)
array([[ 1., 1., 1.], [ 1., 1., 1.],
       [ 0., 0., 0.], [ 0., 0., 0.]])
>>> np.concatenate((A,B), axis=1)
array([[ 1., 1., 1., 0., 0., 0.], [ 1., 1., 1., 0., 0., 0.]])
```

`numpy` contient un sous-module, `np.linalg`, qui réalise les opérations de l'algèbre linéaire.

```
import numpy.linalg as alg
```

- `alg.det(A)` calcule le déterminant de A,
- `alg.matrix_rank(A)` calcule le rang de A,
- `np.trace(A)` calcule la trace de A,
- `alg.inv(A)` calcule l'inverse de A,
- `alg.solve(A, B)` calcule la solution du système de Cramer $A.X = B$,
- `alg.poly(A)` calcule le polynôme caractéristique de A,
- `alg.poly(A)` calcule le polynôme caractéristique de A,
- `alg.eigvalsA)` calcule les valeurs propres de A,
- `alg.eig(A)` calcule le couple formé de la liste des valeurs propres et de la liste des vecteurs propres associés,

2 matplotlib

Le module `matplolib`, plus particulièrement son répertoire `pyplot` premettent de nombreux tracés. On l’utilise avec la commande

```
import matplotlib.pyplot as plt
```

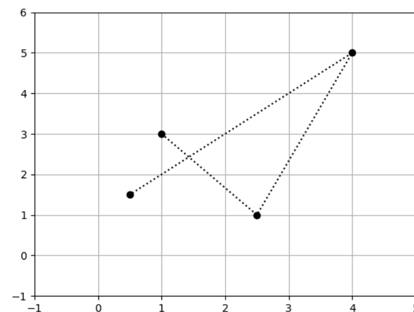
On rappelle qu’il est nécessaire de faire tracer par l’instruction `plt.show()` les éléments que l’on a définis.

2.1 Lignes brisées

Une ligne brisée liant des points du plan est faite par la fonction `plot` ; elle reçoit les listes des abscisses et des ordonnées, qui doivent avoir la même longueur.

- L’instruction `plt.axis('equal')` permet d’obtenir les mêmes échelles.
- L’instruction `plt.grid())` dessine une grille.
- L’instruction `plt.axis([xmin, xmax, ymin, ymax])` définit la partie du plan représentée.
- La fonction plot admet de nombreuses options de présentation.
  - Le paramètre `color` permet de choisir la couleur : `'g'` ou `'green'` pour le vert, ’r’ pour rouge, ’b’ pour bleu . . .
  - Pour définir le style de la ligne, on utilise `linestyle` :’-’ pour une ligne continue, ’- -’ pour une ligne discontinue, ’ :’ pour une ligne pointillée
  - Si on veut marquer les points des listes, on utilise le paramètre `marker` : ’+’, ’.’, ’o’, ’v’ donnent différents symboles.

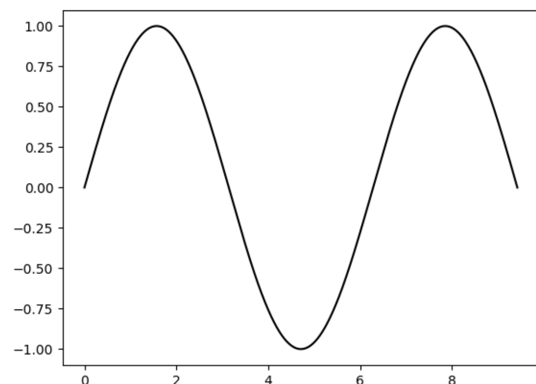
```
X = [1.0, 2.5, 4.0, 0.5]
Y = [3.0, 1.0, 5.0, 1.5]
plt.axis([-1., 5., -1., 6.])
plt.plot(X, Y, linestyle = ':',
               marker = 'o',
               color = 'black')
plt.grid()
plt.show()
```



2.2 Graphe de fonction

Pour tracer le graphe d’une fonction, on définit une liste d’abscisses puis on construit la liste des ordonnées correspondantes.
L’usage de `numpy` est recommandé.

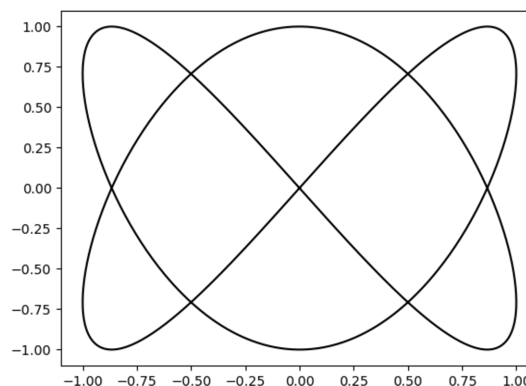
```
X = np.linspace(0, 3*np.pi, 1000)
Y = np.sin(X)
plt.plot(X, Y, color = 'black')
plt.show()
```



2.3 Arc paramétré

Dans le cas d'un arc paramétré, on définit d'abord la liste des valeurs données au paramètre puis on construit la liste des abscisses et des ordonnées correspondantes.

```
T = np.linspace(0, 2*np.pi, 1000)
X = np.sin(2*T)
Y = np.sin(3*T)
plt.plot(X, Y, color = 'black')
plt.show()
```



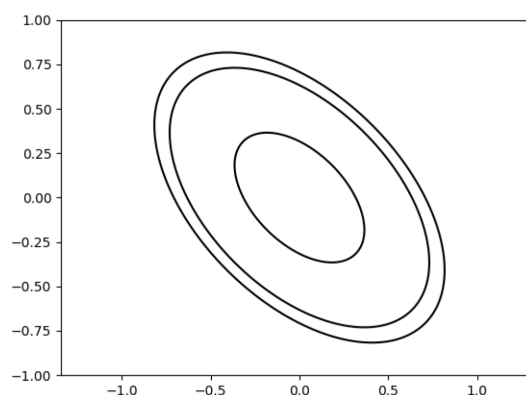
2.4 Lignes de niveau

Pour tracer des courbes d'équation $f(x, y) = k$, on fait une grille en x et en y sur laquelle on calcule les valeurs de f. Pour cela on utilise la fonction `np.meshgrid` de `numpy`.

```
>>> X1 = [-1, 0, 1]
>>> Y1 = [1, 2, 4, 7]
>>> X, Y = np.meshgrid(X1, Y1)
>>> X                              >>> Y
array([[-1,  0,  1],               array([[1, 1, 1],
       [-1,  0,  1],                      [2, 2, 2],
       [-1,  0,  1],                      [4, 4, 4],
       [-1,  0,  1]])                     [7, 7, 7]])
```

On emploie ensuite la fonction `contour` en mettant dans une liste les valeurs de k pour lesquelles on veut tracer la courbe d'équation $f(x, y) = k$.

```
X1 = np.linspace(-1, 1, 300)
Y1 = np.linspace(-1, 1, 300)
X, Y = np.meshgrid(X1, Y1)
Z = X**2 + Y**2 + X*Y
plt.axis('equal')
plt.contour(X, Y, Z,
            [0.1,0.4,0.5],
            colors = 'black')
plt.show()
```



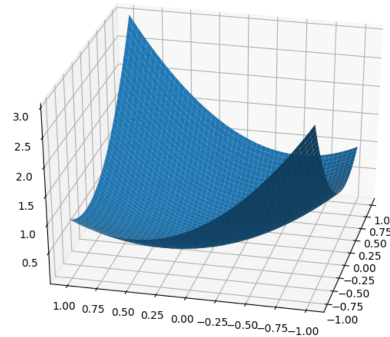
2.5 Surface

`matplotlib` permet de visualiser dans l'espace avec une projection modifiable à la souris. On doit charger la fonction spécifique depuis sa bibliothèque :

```
from mpl_toolkits.mplot3d import Axes3D
```

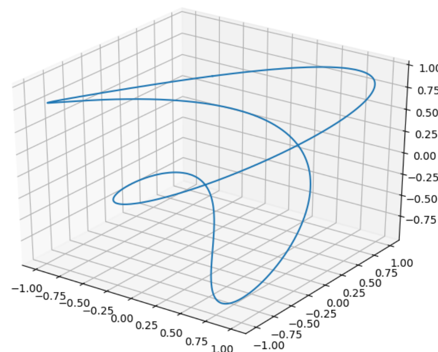
La fonction `Axes3D` crée un environnement à 3 dimension que l'on modifie par la méthode `plot_surface` pour dessiner les points donnés par des tableaux à deux dimensions.

```
X1 = np.linspace(-1, 1, 300)
Y1 = np.linspace(-1, 1, 300)
X, Y = np.meshgrid(X1, Y1)
Z = X**2 + Y**2 + X*Y
ax = Axes3D(plt.figure())
ax.plot_surface(X, Y, Z)
plt.show()
```



Dans le cas de tableaux simples, la méthode `plot` permet de dessiner un arc paramétré dans $\mathbb{R}^3$.

```
T = np.linspace(0, 2*np.pi, 1000)
X = np.sin(2*T)
Y = np.sin(3*T)
Z = np.cos(T)
ax = Axes3D(plt.figure())
ax.plot(X, Y, Z)
plt.show()
```



3 scipy

Le module **scipy** contient les fonctions de calcul numérique.

3.1 Résolution approchée d'équations

Pour résoudre une équation du type $f(x) = 0$ où f est une fonction d'une variable réelle, on peut utiliser la fonction `fsolve` du module `scipy.optimize`. Il faut préciser une valeur initiale x_0 de l'algorithme employé, le résultat peut dépendre de cette condition initiale.

```
from scipy.optimize import fsolve

def f(x):
    return x**2 - 2

>>> fsolve(f, -2.)
array([-1.41421356])
>>> fsolve(f, 2.)
array([ 1.41421356])
```

Dans le cas d'une fonction f de $\mathbb{R}^n$ vers $\mathbb{R}^n$, on utilise la fonction `root`.
Par exemple, pour résoudre $f(x, y) = (x^2 - y^2, x + 2y) = (1, 3)$,

```
from scipy.optimize import root

def f(u):
    x, y = u
    return x**2 - y**2 - 1, x + 2*y -3

>>> sol = root(f, [0,0])
```

```
>>> sol.success
True
>>> sol.x
array([1.30940108, 0.84529946])
```

3.2 Calcul approché d'intégrales

La fonction quad du module scipy.integrate permet de calculer des valeurs approchées d'intégrales. Elle renvoie une valeur approchée de l'intégrale ainsi qu'un majorant de l'erreur commise. Cette fonction peut aussi s'employer avec des bornes d'intégration égales à $+\infty$ ou $-\infty$.

```
from scipy.integrate import quad

def f(x):
    return np.exp(-x)

>>> quad(f, 0, 1)
(0.6321205588285578, 7.017947987503856e-15)
>>> quad(f, 0, np.inf)
(1.0000000000000002, 5.842606742906004e-11)
```

Cette fonction peut être employée pour la définition d'intégrales à paramètres. Ainsi si on veut obtenir des valeurs approchées de $\Gamma(x) = \int_0^{+\infty} e^{-t}t^{x-1}dt$ pour x réel strictement positif on pourra procéder ainsi :

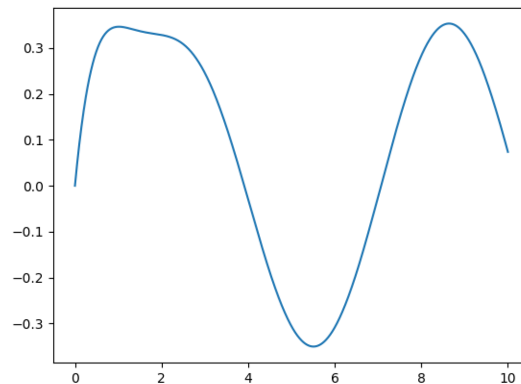
```
from scipy.integrate import quad

def gamma(x):
    def f(t):
        return np.exp(-t)*t**(x-1)
    return quad(f, 0, np.inf)[0]

>>> gamma(0.5)
1.772453850905118
```

3.3 Résolution approchées d'équations différentielles

Pour résoudre une équation différentielle $x' = f(x, t)$, on peut utiliser la fonction `odeint` du module `scipy.integrate`. Cette fonction nécessite comme paramètres, la fonction f, une condition initiale x_0 et une liste de valeurs de t, commençant en t_0. La fonction renvoie des valeurs approchées (aux points contenus dans la liste des valeurs de t) de la solution φ de l'équation différentielle qui vérifie $\varphi(t_0) = x_0$.
La variable x peut être vectorielle, il faut alors utiliser les tableaux numpy qui permettent les opérations vectorielles.
Pour résoudre une équation différentielle scalaire d'ordre 2 de fonction recherchée x, on posera $U = \begin{pmatrix} x \\ x' \end{pmatrix}$.
Ainsi, si on veut représenter la solution de $x''(t) + 2x'(t) + 3x(t) = \sin(t)$ de conditions initiales $x(0) = 0$ et $x'(0) = 1$ on pourra utiliser le code suivant :

```
from scipy.integrate import odeint

def f(u, t):
    x, v = u
    return np.array([v, -2*v - 3*x + np.sin(t)])
```

```
T = np.linspace (0, 10, 1000)
U = odeint(f, np.array([0, 1]), T)
X = U[:, 0] # liste des positions
V = U[:, 1] # liste des vitesses
plt.plot(T, X)
plt.show()
```



4 polynomial

La fonction `Polynomial` du module `numpy.polynomial` permet de travailler avec des polynômes.

```
from numpy.polynomial import Polynomial
```

Pour créer un polynôme, on envoie la liste de ses coefficients par ordre de degré croissant à la fonction.
Par exemple, pour le polynôme $X^3 + 2X - 3$:

```
p = Polynomial([-3, 2, 0, 1])
```

On peut alors utiliser cette variable comme une fonction pour calculer, en un point quelconque, la valeur de la fonction polynôme associée. Cette fonction peut agir également sur un tableau de valeurs, elle calcule alors la valeur de la fonction polynôme en chacun des points indiqués.

```
>>> p(0)
-3.0
>>> p([1, 2, 3])
array([ 0., 9., 30.])
```

4.1 Opérations algébriques

Les polynômes peuvent être ajoutés, multipliés (par une constante ou un autre polynôme).
On peut composer deux polynômes : `p(q)` remplace l'indéterminée X par le polynôme q dans le polynôme p.
L'opérateur `/` permet de diviser un polynôme par un scalaire. Pour diviser deux polynômes il faut utiliser l'opérateur `//` qui renvoie le quotient, l'opérateur `%` calcule le reste.

4.2 Méthodes

Le module définit des fonctions qui s'appliquent aux polynômes sous le forme de méthodes.

- `coef` donne la liste des coefficients, ordonnés par degré croissant.
- `degree()` donne le degré du polynôme.
- `roots()` renvoie la liste des racines, une racine double est donnée deux fois.
- `deriv()` renvoie un nouveau polynôme, dérivé du polynôme initial. Cette méthode prend en argument facultatif un entier positif indiquant le nombre de dérivations à effectuer.
- `integ()` intègre le polynôme, elle prend un paramètre optionnel supplémentaire donnant la constante d'intégration à utiliser, nulle par défaut.

5 `random`

Les fonctions d'échantillonnage et de génération de valeurs pseudo-aléatoires adaptées aux tableaux `numpy` sont regroupées dans la sous-bibliothèque **`numpy.random`**.

```
import numpy.random as rd
```

- L'expression `randint(a, b)` permet de choisir un entier au hasard dans l'intervalle $[\![a, b[\![$.
- La fonction `random()` renvoie un réel compris dans l'intervalle $[0; 1]$. Si X désigne la variable aléatoire correspondant au résultat de la fonction, alors pour tout a et b dans $[0; 1]$ avec $a \leqslant b$, on a $P(a \leqslant X < b) = b - a$.
- La fonction `binomial(n, p)` permet de simuler une variable aléatoire suivant une loi binomiale de paramètres n et p. Elle permet donc également de simuler une variable aléatoire suivant une loi de Bernoulli de paramètres p en prenant simplement $n = 1$.
- La fonction `geometric(k)` permet de simuler une variable aléatoire suivant une loi géométrique de paramètre k.
- La fonction `poisson(k)` permet de simuler une variable aléatoire suivant une loi de Poisson de paramètre k.

Ces fonctions peuvent prendre un troisième paramètre optionnel permettant d'effectuer plusieurs tirages et de renvoyer les résultat sous forme de tableau ou de matrice.

Exemples

```
>>> rd.random((2,4))
array([[ 0.78230688, 0.83803526, 0.62077457, 0.27432819],
       [ 0.66522387, 0.71258365, 0.25813448, 0.28833084]])

>>> rd.poisson(4, 15)
array([5, 2, 3, 4, 6, 0, 5, 3, 1, 5, 1, 5, 9, 4, 6])
```

Chapitre **R-II**

Méthode d'Euler

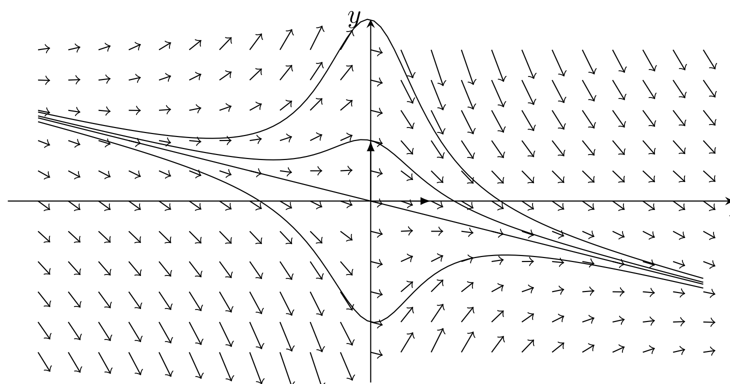
1 Équation différentielle

1.1 Définitions

Une équation différentielle (ordinaire) du premier ordre est une équations de la forme

$$(E) \qquad y' = \varphi(y, t)$$

où ψ est définie sur un ouvert $\mathcal{O}$ de $\mathbb{R}^n \times \mathbb{R}$ vers $\mathbb{R}^n$.
Une solution de (E) est une fonction f de classe $\mathcal{C}^1$ sur un intervalle I de $\mathbb{R}$ vers $\mathbb{R}^n$ telle que $\big(f(t), t\big) \in \mathcal{O}$ et $f'(t) = \psi\big(f(t), t\big)$ pour tout $t \in I$.



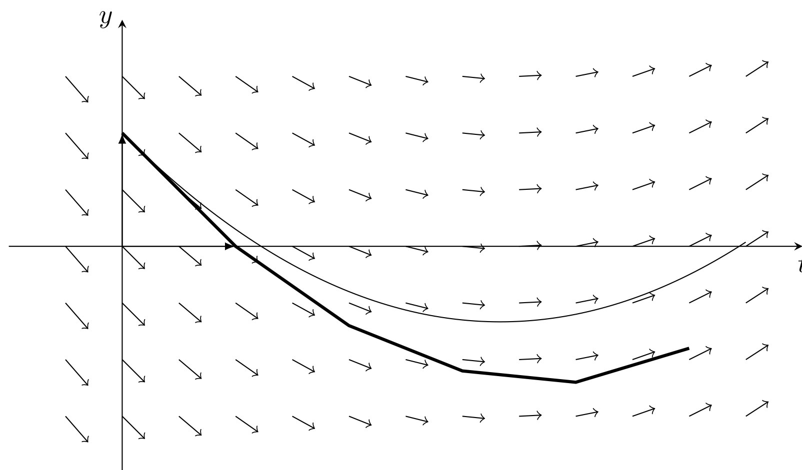
Soit $(t_0, y_0) \in U$. Sous certaines conditions sur φ, le théorème de Cauchy-Lipschitz affirme l'existence et l'unicité d'une solution (I, f) telle que $t_0 \in I$ et $f(t_0) = y_0$.

On peut alors écrire $f(t) = f(t_0) + \displaystyle\int_{t_0}^{t} f'(u)\mathrm{d}u = y_0 + \int_{t}^{t_0} \varphi\big(f(u), u\big)\mathrm{d}u$.
On a remplacé l'équation différentielle par une équation intégrale.

1.2 Recherche d'approximation

Résoudre (on dit aussi intégrer) l'équation différentielle signifie trouver l'unique fonction solution de l'équation pour des conditions initiales données. On cherchera une solution **sur** un segment déterminé, on admettra qu'il existe une solution sur cet intervalle.
Le temps initial doit appartenir au segment : on simplifiera l'étude en supposant que l'intervalle d'étude commence en $t_0 = a$: $[a; b]$. Les algorithmes proposés seront utilisables aussi si on a $b < a$.

Si la condition initiale était donnée au temps $t_0 \in]a;b[$ on diviserait l'étude sur deux intervalles : $[a;t_0]$ et $[t_0;b]$.
Une solution est une fonction, on souhaite la valeur en tout temps t de l'intervalle. Malheureusement il est illusoire d'espérer calculer une "formule" de la solution. On se contente donc de calculer la valeur de la solution en certains points **fixés à l'avance** de l'intervalle.
Pour cela on donnera comme paramètre une liste de temps $[t_0, t_1, \ldots, t_{N-1}]$ avec $t_0 = b$ et $t_{N-1} = b$.
Le plus simple sera d'utiliser la fonction `numpy.linspace(a,b,N)` qui renvoie une liste de N valeurs régulièrement espacés entre a et b.
Si on calcule une valeur approchée de $f(t_k)$ avec y_k on pourra approcher la solution par la fonction affine par morceaux égale à y_k aux points t_k.



Comme on a $f(t_{k+1}) = f(t_k) + \displaystyle\int_{t_k}^{t_{k+1}} f'(u)\mathrm{d}u = f(t_k) + \int_{t_k}^{t_{k+1}} \varphi\big(f(u), u\big)du$ on voit qu'on peut calculer les y_k de proche en proche en déterminant une valeur approchée de $\displaystyle\int_{t_k}^{t_{k+1}} \varphi\big(f(u), u\big)du$.
Pour conserver une cohérence avec les fonctions déjà écrites dans python, toute méthode de résolution aura 3 paramètres (on peut en ajouter d'autres s'ils ont une valeur par défaut), donnés dans l'ordre suivant.

- La fonction `phi` qui définit l'équation différentielle. Elle devra avoir été définie sous la forme

```
def phi(y,t):
    # Calcul de la valeur
    return resultat
```

- La condition initiale, valeur de la solution recherchée en t_0.
- La liste des temps où l'on recherche une valeur approchée de la solution.

La fonction devra renvoyer la liste des valeurs, de même longueur que la liste des temps.

1.3 Équation différentielle d'ordre 2

Une équation différentielle d'ordre 2, de la forme $y'' = \psi(t, y, y')$, se ramène à une équation différentielle vectorielle du premier ordre en posant $u = \begin{pmatrix} y \\ y' \end{pmatrix}$:

$$u' = \begin{pmatrix} y' \\ y'' \end{pmatrix} = \begin{pmatrix} y' \\ \psi(t, y, y')' \end{pmatrix} = \Psi(u, t)$$

2 Méthodes de résolution

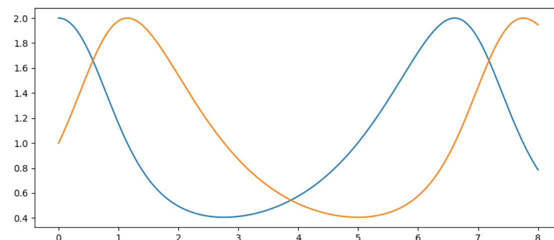
2.1 Équations vectorielles

Dans le cas d'équations vectorielles (la fonction recherchée est à valeurs dans $\mathbb{R}^n$ avec $n >\geqslant 2$), on doit utiliser un type de données qui permette l'addition terme-à-terme et la multiplication par un scalaire : les tableaux `numpy` ont cette caractéristique.

Par exemple le système d'équations $\begin{cases} x' = x - xy \\ y' = -y + xy \end{cases}$ sera défini par la fonction

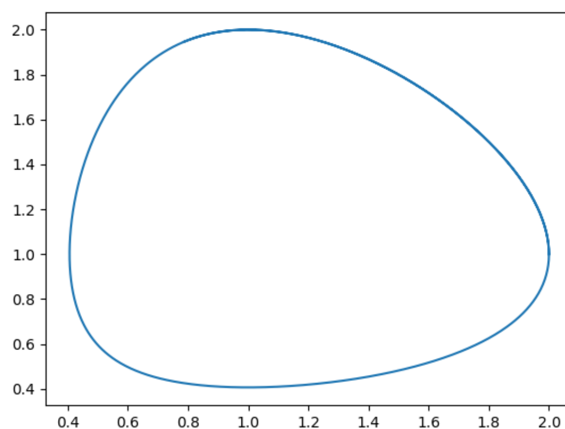
```
def phiSysteme(u,t):
    x, y = u
    dx = x - x*y
    dy = -y + x*xy
    return np.array([dx, dy])
```

Les fonction définies seront appelées comme dans le cas d'une fonction scalaire, la condition initiale devra être aussi un tableau `numpy`.
Le résultat sera une liste de vecteurs, qui peut être tracé directement.



Il sera souvent utile, dans le cas $n = 2$ ou $n = 3$, de tracer l'arc paramétré des composantes.
Si U est la liste des solutions, on calcule les composantes.

```
X = [u[0] for u in U]
Y = [u[1] for u in U]
plt.plot(X, Y)
```



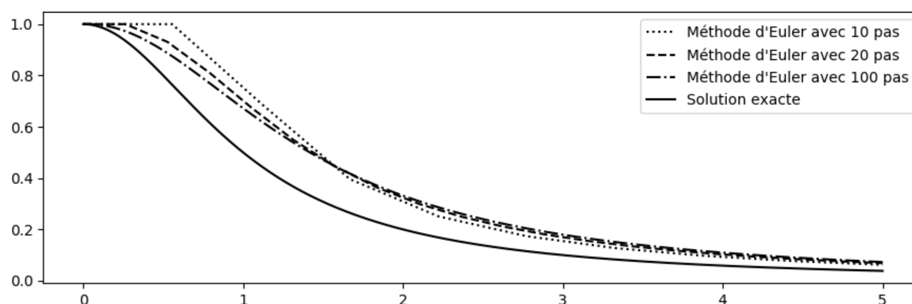
2.2 Méthode d'Euler

La méthode d'Euler consiste à approcher simplement la fonction à intégrer, $\varphi\big(u, f(u)\big)$, par sa valeur initiale $\varphi\big(f(y_k, t_k)\big)$. On ne connaît qu'une valeur approchée de $f(t_k)$ avec y_k donc on a

$$\int_{t_k}^{t_{k+1}} \varphi\big(f(u), u\big)\mathrm{d}u \simeq \int_{t_k}^{t_{k+1}} \varphi(y_k, t_k)\mathrm{d}u = (t_{k+1} - t_k)\varphi(y_k, t_k) = \delta_k$$

L'algorithme suit alors la méthode :

```
def euler(f,y0,T):
    """Entrée : une fonction de ExR vers E
                un élément de E, la condition initiale y0
                une liste des temps
       Sortie : une solution approchée de y' = f(t,y)
                avec les conditions initiales (T[0],y0)
                sous la forme d'une liste des valeurs
                aux temps définis par T"""
    n = len(T)
    Y = [0]*n
    Y[0]= [y0] # ordonnée initiale
    for i in range(n-1): # il reste n-1 valeurs à calculer
        y = Y[i]
        pas = T[i+1] - T[i]
        pente  = f(y,T[i])
        Y[i+1] = y + pas*pente
    return Y
```



2.3 Méthode d'Euler implicite

Si on approche la fonction à intégrer, $\varphi\big(f(u), u\big)$, par sa valeur finale sur l'intervalle $[t_k; t_{k+1}]$ c'est-à-dire $\varphi\big(u, f(u)\big) \simeq \varphi\big(f(t_{k+1}, t_{k+1})\big) \simeq \varphi(y_{k+1}, t_{k+1})$ on obtient

$$y_{k+1} = y_k + \int_{t_k}^{t_{k+1}} \varphi(y_{k+1}, t_{k+1})\mathrm{d}u = y_k + (t_{k+1} - t_k)\varphi(y_{k+1}, t_{k+1})$$

On voit que y_{k+1} apparaît dans le second membre : on ne peut pas le calculer directement.
Pour déterminer t_{k+1} en fonction de t_k il faut donc résoudre, à chaque étape, l'équation $g_k(y) = 0$ avec $g_k(y) = y - y_k - (t_{k+1} - t_k)\varphi(y, t_{k+1})$.
On vu que la fonction `fsolve` du module `scipy.optimize` permet de résoudre cette équation.

2.4 Méthode `odeint`

Le module `scipy` contient, dans la sous-bibliothèque `integrate` une fonction `odeint` qui résout efficacement les équations différentielles. C'est la méthode à utiliser quand la résolution d'une équation est un outil d'un projet plus large.
Ses paramètres sont la fonction qui définit l'équation différentielle, l'ordonnée initiale et un tableau des temps : on doit définir cette liste **avant** d'invoquer la fonction.
Elle renvoie le tableau des valeurs de y.

```
import numpy as np
from scipy.integrate import odeint

Y = odeint(f,y0,T)
```

2.5 Méthode de Heun

La méthode de Heun améliore le schéma d'Euler en approchant l'intégrale par la méthode des trapèzes : $\int_a^b g(u)\mathrm{d}u \simeq (b-a)\frac{g(a)+g(b)}{2}$. On obtient

$$y(t_{k+1}) - y(t_k) \simeq (t_{k+1}-t_k)\frac{\varphi\big(y(t_k),t_k\big)+\varphi\big(y(t_{k+1}),t_{k+1}\big)}{2}$$

Cependant cela devient une équation implicite en $y(t_{k+1})$ qu'on ne souhaite pas résoudre.
On va donc procéder en approchant la valeur de $y(t_{k+1})$ dans le second membre par celle que calcule la méthode d'Euler.

- On calcule $z_k = y_k + \varphi(y_k,t_k)(t_{k+1}-t_k)$ comme valeur approchée de $y(t_{k+1})$
- On calcule la valeur approchée $y_{k+1} = y_k + \frac{\varphi(y_k,t_k)+\varphi(z_k,t_{k+1})}{2}(t_{k+1}-t_k)$.

2.6 Méthode de Runge-Kutta

La méthode de Runge-Kutta s'inspire de la méthode de Simpson :

$$\int_a^b g(u)\mathrm{d}u \simeq (b-a)\frac{g(a)+4g\big(\frac{a+b}{2}\big)+g(b)}{6}$$

On obtient, pour $m_k = \frac{t_k+t_{k+1}}{2}$ et $h = t_{k+1}-t_k$,

$$y(t_{k+1}) \simeq y(t_k) + h\frac{\varphi\big(y(t_k),t_k\big)+4\varphi\big(y(m_k),m_k\big)+\varphi\big(y(t_{k+1}),t_{k+1}\big)}{6}$$

On va, ici encore, approcher provisoirement les valeurs de $y(m_k)$ et $y(t_{k+1})$.
On va en fait utiliser deux approximations de $y(m_k)$.

- $a = y_k + \frac{h}{2}\varphi\big(t_k,y_k\big)$ est une première valeur approchée de $y(m_k)$
- $b = y_k + \frac{h}{2}\varphi\big(t_k+\frac{h}{2},a\big)$ est aussi une valeur approchée de $y(m_k)$
- $c = y_k + h\varphi\big(t_k+\frac{h}{2},b\big)$ est une valeur approchée de $y(t_{k+1})$ obtenue en prenant comme valeur moyenne la valeur au point milieu.
- On pose alors
  $y_{k+1} = y_k + h\frac{\varphi\big(t_k,y_k\big)+2\varphi\big(t_k+\frac{h}{2},a\big)+2\varphi\big(t_k+\frac{h}{2},b\big)+\varphi\big(t_k+h,c\big)}{6}$

2.7 Cas particulier

On voit souvent des équations différentielles d'ordre 2 de la forme $y'' = \psi(y, t)$.
Les conditions initiales forment un couple (y_0, v_0).

1. On peut adapter la méthode d'Euler en faisant intervenir une suite des vitesses :

$$\begin{aligned} y_{i+1} &= y_i + (t_{i+1} - t_i)v_i \\ v_{i+1} &= v_i + (t_{i+1} - t_i)\psi(y_i, t_i) \end{aligned}$$

   Cela revient à résoudre l'équation du premier ordre obtenue par vectorisation.
2. On peut remarquer qu'on n'a pas besoin de la suite des vitesses si le pas est constant : $t_{i+1} - t_i = \Delta t$ pour tout i.

$$\begin{aligned} y_1 &= y_0 + \Delta t v_0 \\ y_{i+2} &= 2y_{i+1} - y_i + (\Delta t)^2\psi(y_i, t_i) \end{aligned}$$

3. Si le pas est constant la **méthode de Verlet** améliore la précision en décalant le calcul de l'accélération. On améliore aussi le calcul de y_1 car, sinon, on perd le bénéfice de la méthode.

$$\begin{aligned} y_1 &= y_0 + \Delta t v_0 + \tfrac{(\Delta t)^2}{2}\psi(y_0, t_0) \\ y_{i+2} &= 2y_{i+1} - y_i + (\Delta t)^2\psi(y_{i+1}, t_{i+1}) \end{aligned}$$

4. Toujours dans le cas d'un pas constant la **méthode saute-mouton** (leapfrog en anglais) donne des résultats précis en entrelaçant dans le temps les calculs de vitesses et de position. On note w_i une approximation de la vitesse au temps $t_i + \frac{1}{2}\Delta t$.

$$\begin{aligned} w_0 &= v_0 + \tfrac{1}{2}\Delta t\psi(t_0, y_0) \\ y_{i+1} &= y_i + \Delta t w_i \\ w_{i+1} &= w_i + \Delta t\psi(y_{i+1}, t_{i+1}) \end{aligned}$$