Lycée Faidherbe, 2019-2020

Cours d'informatique

MP, PC et PSI

Version du 21 février 2021

Table des matières

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**.

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 indiquer quelques pistes.

Le programme fournit-il le bon résultat dans un temps raisonnable ? On va choisir des indicateurs de la mesure du temps qui permettent d'évaluer le temps pris pour le traitement des données en fonction de leur taille.

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.

1 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, on nommera **tranche** une suite de termes consécutifs, cela correspond à une extraction `liste[i:j]` de la liste originale.
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 : ainsi une des somme possibles est 0, correspondant à la tranche vide, le maximum sera positif (ou nul). Lors de la recherche du maximum, on pourra donc initialiser le maximum à 0 et chercher à améliorer ce maximum en considérant toutes les tranches non vides.

1.1 Première fonction

La première idée est de suivre les indications ci-dessus.
On écrit une fonction 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_tranche(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 trancje_max1(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_tranche(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.

1.2 Mesurer le temps

Pour mesurer la complexité de l'algorithme, une idée naturelle est de mesurer le temps pris par l'exécution de l'algorithme.

- La mesure du temps peut être faite par la fonction `time()` du module `time`.

```
from time import time
```

`time()` renvoie un nombre flottant qui donne, en secondes, le temps écoulé depuis une origine arbitraire *epoch* : pour les machines sous `Linux` c'est le 1er janvier 1970 à 00 :00 :00.
On pourra afficher le temps d'exécution en secondes.

```
init = time()
# calculs
fin = time()
print(fin - début)
```

- On a besoin de listes de taille fixée sur lesquelles faire tourner la fonction.
  On peut créer des listes simples, `[1]*n`, pour lesquelles on connait la somme maximale de tranche (n) ou on peut créer des listes aléatoires à l'aide de fonctions du module `random`.

```
import random as rd
```

Par exemple la fonction `randint(a, b)` renvoie un entier choisi au hasard avec une probabilité uniforme dans $\{a, a+1, \ldots, b-1, b\}$. On définit alors une fonction qui crée une liste de taille donnée formée d'entiers aléatoires entre deux bornes.

```
def liste_alea(taille, mini, maxi):
    l = [0]*taille
    for i in range(taille):
        l[i] = rd.randint(mini, maxi)
    return l
```

- On obtient les valeurs suivantes (arrondies) pour des tailles qui doublent à chaque étape.

n	10	20	40	80	160	320	640	1280
temps (secondes)	3.10^{-5}	10^{-4}	8.10^{-4}	5.10^{-3}	4.10^{-2}	$0,25$	2	18

On remarque que le temps n'est pas proportionnel à la taille.

2 Comment prédire l'augmentation du temps ?

Prévoir le temps exact n'est pas possible, cela va dépendre de beaucoup de critères indépendant de l'algorithme écrit : la vitesse du processeur, les autres activités de l'ordinateur, le langage utilisé, ...
On va se contenter d'estimer l'accroissement du temps quand la taille des données d'entrée augmente.

Que se passe-t-il si on double la taille des données ?

Dans l'exemple ci-dessus, les rapports entre les temps calculs pour $2n$ et pour n donnent,

4,1 5,7 6,2 7,5 7,5 8,3 7,9

2.1 Que mesure-t-on ?

On peut compter le temps utilisé en nombre de cycles du processeur.
Cependant il est souvent difficile de comprendre ce qui se passe à un niveau aussi bas d'instructions.
Voici, par exemple le code correspondant à la fonction `somme_tranche`

```
      0 LOAD_CONST               1 (0)
      2 STORE_FAST               3 (s)

      4 LOAD_GLOBAL              0 (range)
      6 LOAD_FAST                1 (i_min)
      8 LOAD_FAST                2 (i_max)
     10 LOAD_CONST               2 (1)
     12 BINARY_ADD
     14 CALL_FUNCTION            2
     16 GET_ITER
>>   18 FOR_ITER                16 (to 36)
     20 STORE_FAST               4 (i)

     22 LOAD_FAST                3 (s)
     24 LOAD_FAST                0 (liste)
     26 LOAD_FAST                4 (i)
     28 BINARY_SUBSCR
     30 BINARY_ADD
     32 STORE_FAST               3 (s)
     34 JUMP_ABSOLUTE           18

>>   36 LOAD_FAST                3 (s)
     38 RETURN_VALUE
```

On pourrait compter le nombre d'instruction sans chercher à les comprendre mais il y a des sauts (marqués par ») dont il est nécessaire de savoir combien de fois ils adviennent.

On compte plutôt le nombre d'opérations "élémentaires" dans le langage choisi. Ce sont
1. les affectations,
2. les opérations arithmétiques, sommes, produits, quotients, ...
3. les comparaisons,
4. etc

Le plus souvent cela sera souvent inutilement compliqué ; on choisit une instruction élémentaire signifiante et on compte le nombre d'exécution de cette instruction. Dans notre exemple nous allons compter le nombre d'additions. On choisit avec soin le type d'instruction afin que le temps de calcul soit, grossièrement, proportionnel au nombre de ces instructions.

2.2 Quel résultat est souhaité ?

Ce nombre d'instructions sera évalué en fonction d'une mesure de la taille des données d'entrée. Ce peut être
1. la valeur d'une variable entière,
2. le nombre d'éléments des listes, tableaux, fichiers,
3. la longueur d'un mot,
4. le degré d'un polynôme,
5. le nombre de lignes (ou de colonnes ou leur produit) dans le cas d'une matrice,
6. le nombre de bit dans sa représentation en base 2 d'une variable entière,
7. etc

On aboutit à une fonction de complexité dont la variable est cette mesure [1].

Cependant la complexité peut varier selon les différentes entrées possible d'une même taille. Nous choisirons souvent de calculer la complexité maximale [2].
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 $C(n)$.

Même cette fonction $C(n)$ est en général inutilement précise : on rappelle que ce qui nous intéresse est l'évolution avec la taille n. L'indication que nous allons garder est la recherche d'un $\mathcal{O}\big(g(n)\big)$ où g est une fonction "simple"
$C(n) = \mathcal{O}\big(g(n)\big)$ signifie qu'il existe une constante K telle que $C(n) \leqslant Kg(n)$ pour tout n (ou pour tout $n \geqslant n_0$).
1. On parle de complexité linéaire quand la complexité est un $\mathcal{O}(n)$,
2. On parle de complexité quadratique quand elle est un $\mathcal{O}(n^2)$,
3. On parle de complexité polynomiale quand elle est un $\mathcal{O}(n^p)$,
4. On parle de complexité quasi-constante quand elle est un $\mathcal{O}\big(\log_2(n)\big)$,
5. On parle de complexité quasi-linéaire quand elle est un $\mathcal{O}\big(n\log_2(n)\big)$.
6. On parle de complexité exponentielle quand elle est un $\mathcal{O}(a^n)$,

2.3 Complexité de `tranche_max1`

La fonction `somme_tranche` effectue `fin + 1 - debut` additions.
Le nombre d'additions effectuées pour une liste de taille n dans `tranche_max1` est donc, en posant $k = j + 1 - i$ puis $p = n - i$,

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

1. Parfois l'entrée sera caractérisée par plusieurs mesures (lorsqu'il y a plusieurs listes en paramètres, dans le cas d'une matrice, ...).
2. Mais on peut aussi déterminer la complexité en moyenne.

L'accroissement du temps de calcul lorsque l'on multiplie la taille de la liste par 2 est donc un facteur 8, ce que l'on a constaté pour les grandes valeurs de n, pour les plus petites valeurs de n les termes de degré moindre ne sont pas négligeables.

3 Aller plus vite

L'algorithme précédent est peu efficace. Il ne faut pas pour autant le rejeter comme inutile.
Lors de la résolution d'un problème, on applique le principe :

D'abord écrire un programme qui fonctionne,
Ensuite écrire un programme qui fonctionne rapidement.

Pour améliorer le temps de calcul, 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 tranche_max2(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.

Temps de calcul : voici les temps mesurés

n	10	20	40	80	160	320	640	1280
temps (secondes)	8.10^{-06}	2.10^{-05}	7.10^{-05}	2.10^{-04}	9.10^{-04}	4.10^{-03}	2.10^{-02}	6.10^{-02}
rapport		$2,2$	$3,5$	$3,2$;	$4,0$	$4,6$	$4,3$	$3,8$

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.

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

On aboutit donc à une complexité quadratique qui explique la multiplication par 4 des temps de calcul.

4 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.

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

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 tranche des 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 $\mathcal{O}(n)$.

5 Autres analyses

Nous allons illustrer les autres composantes de l'analyse par la décomposition en base 2.
C'est l'exercice 10 du TP de révisions.

```
def base10To2(n):
    nb = n
    liste = []
    while nb > 0:
        liste.append(nb%2)
        nb = nb//2
    return liste
```

5.1 Terminaison

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.
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. Il est donc indispensable que quelque chose soit modifié dans cette boucle.

Exemple : `base10To2`.
La condition de la boucle `while` est `nb > 0` : il faut donc prouver que `nb` finit par prendre la valeur 0.
On suppose qu'on a $2^p \leqslant n < 2^{p+1}$.
Une possibilité de démonstration consiste à remarquer que si on a $2^k \leqslant nb < 2^{k+1}$ avec $k \geqslant 1$ alors $2^{k-1} \leqslant nb//2 < 2^k$. Ainsi l'encadrement de nb par des puissance de 2 est strictement décroissant et, au bout de p itérations de la boucles on a $2^0 \leqslant nb < 2^1$ donc $nb = 1$. Une itération supplémentaire donne $nb = 1//2 = 0$ et on sort de la boucle.
On a prouvé que l'algorithme termine.
On a même prouvé qu'on itérait $p+1$ fois les calculs avec p tel que $2^p \leqslant n < 2^{p+1}$ donc $p \leqslant \log_2(n)$ la complexité est donc majorée par $K.(p+1) \leqslant K.\big(\log_2(n)+1\big) = \mathcal{O}\big(\log(n)\big)$.

5.2 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 était-ce le seul problème ? Ici, c'est le cas mais on peut souhaiter une preuve définitive.
Le cas des instructions sans boucle est élémentaire : la preuve est celle des instructions utilisées.

5.3 Boucles for

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 $\sum_{k=0}^{i} x_k = x_i + \sum_{k=0}^{i-1} x_k$.

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.

P est un invariant de boucle.

5.4 Boucles while

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 en cherchant une propriété qui dépend de variables et qui vérifie.

- elle est vraie avant le premier passage de la boucle,
- si elle est vraie et si la condition du `while` est vérifiée alors elle reste vraie après l'exécution des instructions de la boucle,
- la négation de la condition combinée avec la propriété prouve le résultat attendu.

Exemple : `base10To2`. Pour déterminer une propriété invariante on peut commencer par un exemple. On calcule `base10To2(43)` et regarde les valeurs des variables ; comme une liste est associée à un entier, on détermine aussi l'entier associé à la liste.

nombre de passages	`nb`	`liste`	entier	longueur
0	43	`[]`	0	0
1	21	`[1]`	1	1
2	10	`[1, 1]`	3	2
3	5	`[1, 1, 0]`	3	3
4	2	`[1, 1, 0, 1]`	11	4
5	1	`[1, 1, 0, 1, 0]`	11	5
6	0	`[1, 1, 0, 1, 0, 1]`	43	6

Après quelques [3] tâtonnements on peut s'apercevoir qu'on a $nb.2^p + r = n$ où p est la longueur de la liste et r est l'entier associé à cette liste.
On considère donc la propriété générale P(`nb`, `liste`) : $nb.2^p + r = n$ avec les notations ci-dessus.

- Lors de l'initialisation, on a $nb = n$ et la liste est vide donc $p = 0$ et $r = 0$:
  on a bien $nb.2^p + r = n.2^0 + 0 = n$.
- On suppose que la propriété est vérifiée avec $nb \neq 0$ et on effectue un passage.
  On a $nb = 2.nb' + b$ avec $nb' = nb//2$ et $b = nb\%2 \in \{0, 1\}$.
  On adjoint b à la liste donc la longueur est maintenant $p' = p + 1$ et la valeur associée est $r' = r + b.2^p$.
  On a alors $n = nb.2^p + r = (2.nb' + b).2^p + r = nb'.2.2^p + (r + 2^p.b) = nb'.2^{p+1} + r'$:
  la propriété est toujours vérifiée.
- Si la condition du `while` n'est plus vérifiée alors $nb = 0$ et la propriété implique qu'on a $n = 0.2^p + r$, c'est-à-dire n est bien représenté par la liste.

3. Beaucoup !

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

L'ordre provient de comparaisons entre éléments : leur ensemble 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
```

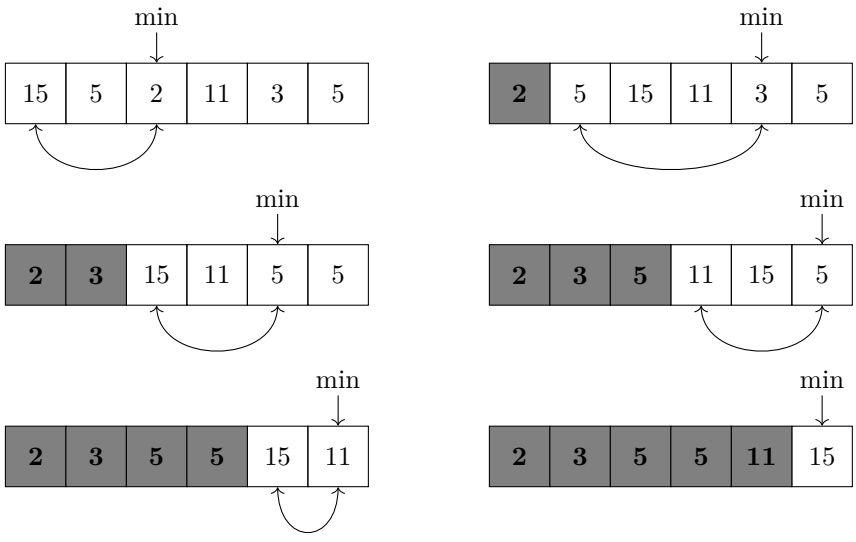
Cas d'égalité

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 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 d'une fonction qui recherche l'indice du minimum à partir d'un rang.
C'est un algorithme classique.

```
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"""
    n = len(liste)
    ind_min = i
    mini = liste[i]
    for j in range(i+1,n):
        if plusGrand(mini, liste[j]):
            mini = liste[j]
            ind_min = j
    return ind_min
```

On choisit la première apparition du minimum pour que le tri soit stable.

On peut alors écrire la fonction de tri.

```
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.

2.1 Analyse

Terminaison

Les fonctions n'utilisent que des boucles `for`, on est donc assurer qu'elles terminent.

Preuve

On a construit la liste triée pas-à-pas.

```
def triSelection(liste):
    n = len(liste)
    for i in range(n):
        # Les i premiers éléments de la liste triée
        # sont à leur place
        k = indMinDepuis(liste, i)
        echange(liste, i, k)
        # Les (i+1) premiers éléments de la liste triée
        # sont à leur place
```

À la fin de la boucle, i a pris la valeur $n - 1$ et la propriété implique que les $n - 1 + 1$ premiers éléments de la liste triée sont à leur place : la liste est triée.
On doit donc s'assurer qu'on place le $i + 1$-ième élément trié à la position i. Comme les i premiers éléments, les plus petits, sont déjà placés, on doit choisir le minimum de ceux qui restent.

On peut prouver que `indMinDepuis(liste, i)` détermine bien le minimum en prouvant que `mini` contient, au début de la boucle pour j, le minimum des valeurs de la liste entre i et $j-1$, c'est notrre invariant.

Si on note a_k la valeur de `liste[k]` et $m_{i,k} = \min\{a_i, a_{i+1}, \dots, a_k\}$, l'algorithme calcule m_j en fonction de m_{j-1} en utilisant la propriété
$m_j = m_{j-1}$ si $a_j \geqslant m_{j-1}$ et $m_j = a_j$ sinon.

Complexité

On effectue une comparaison pour chaque j de $i+1$ à $n-1$ donc le nombre de comparaisons est $n-1-i$
On en déduit que le nombre de comparaisons de `triSelection` pour une liste de taille n est donc $C(n) = \sum_{i=0}^{n-1}(n-i-1) = \sum_{p=0}^{n-1} p = \frac{n(n-1)}{2}$, la complexité est un $\mathcal{O}(n^2)$.

3 Tri par insertion

3.1 Principe du tri

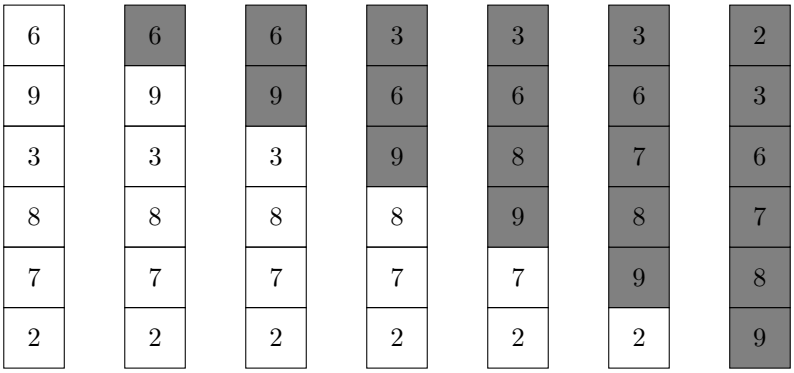
Le tri par sélection est celui que les joueurs de cartes peuvent employer : pour trier un tas de carte, on prend une carte à la fois et on l'insère dans les cartes déjà triées.
Dans le cadre du tri en place d'une liste, on trie les éléments les uns après les autres en laissant en place ceux qu'on n'a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



On peut donc écrire le programme suivant

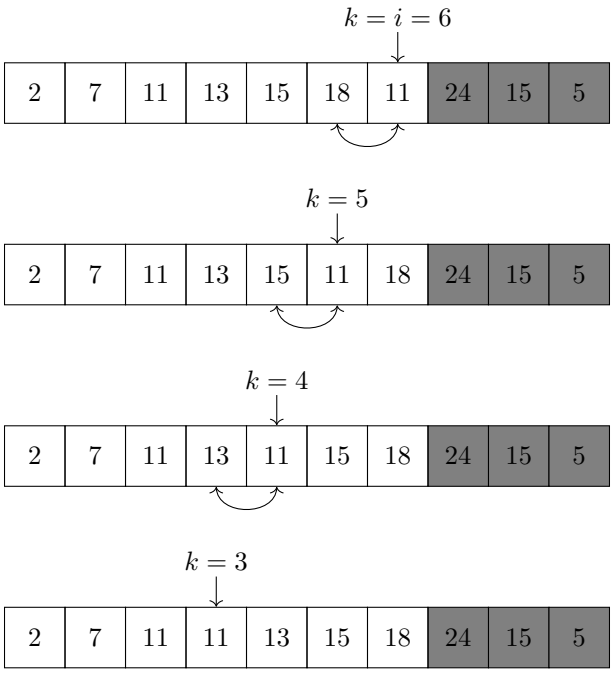
```
def triInsertion(liste):
    """Entrée : une liste d'éléments comparables
       Sortie : les éléments sont permutés pour obtenir
                une liste triée par ordre croissant"""
    n = len(liste)
    for i in range(n):
        inserer(liste, i)
```

3.2 Insertion

Pour réaliser `inserer(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.

Le fait de ne pas échanger en cas d'égalité donnera un tri 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 inserer(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:
        if plusGrand(liste[k-1], liste[k]):
            echange(liste, k, k-1)
            k = k - 1
        else:
            break
```

3.3 Analyse

Terminaison

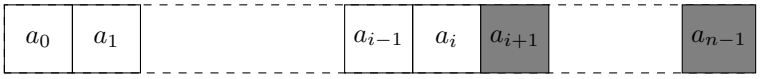
La fonction `inserer` termine car l'entier `k` décroît strictement à chaque passage de la boucle donc finit par être nul si n'est pas sorti de la boucle auparavant.
La fonction `triInsertion` fait appel $n - 1$ fois à la fonction `inserer` qui termine donc le tri termine.

Preuve

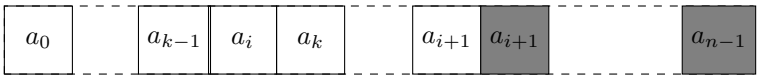
On veut que la liste soit triée : pour cela on trie des portions de plus en plus grandes.
On introduit une propriété vérifiée lors du passage de la boucle.

```
def triInsertion(liste):
    n = len(liste)
    for i in range(n):
        # Les i premiers éléments de la liste sont triés,
        # les autres sont inchangés
        inserer(liste, i)
        # Les i+1 premiers éléments de la liste sont triés,
        # les autres sont inchangés
   # Les n éléments de la liste sont triés
```

À la fin de la boucle, i a pris la valeur $n-1$ et la propriété implique que les $n-1+1$ premiers éléments sont triés : la liste est triée.
On doit donc prouver que `inserer(liste, i)` transforme la propriété de i à $i+1$.
On note a_j la valeur de `liste[j]`.



On peut remarquer que les transformations successives donnent des résultats



On peut exprimer ces positions par les propriétés

```
liste[0] <= liste[1]   <= ... <= liste[k-1]
liste[k] <= liste[k+1] <= ... <= liste[n-1]
liste[k-1] <= liste[k]
```

On prouve alors que les instructions

```
echange(liste, k, k-1)
            k = k - 1
```

conservent les propriétés, elles forment un invariant, lorsque l'on a `liste[k] < liste [k-1]`.
Par contre, si on a `liste[k] >= liste [k-1]`, la portion de 0 à i est triée donc on a le résultat souhaité.
De même, pour $k=0$ la portion est triée aussi.
Ainsi la fonction `inserer` produit bien le résultat souhaité.

Complexité

On effectue 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 pour `inserer(liste, i)`.
On en déduit que le nombre de comparaisons de `triInsertion` pour une liste de taille n, $C(n)$ vérifie $\displaystyle\sum_{i=0}^{n-1} 1 = n \leqslant C(n) \leqslant \sum_{i=0}^{n-1} i = \frac{n(n-1)}{2}$, la complexité est un $\mathcal{O}\big(n^2\big)$.
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.
De même la complexité est minimale pour une liste strictement croissante au départ.
Le tri par insertion est préférable au tri par sélection, il peut être très rapide dans le cas de liste qui sont peu désordonnées. On l'emploie parfois dans des tris plus rapides lors des dernières étapes pour lesquelles la liste est triée par blocs.
Si on compte aussi les échanges, on voit que le tri par sélection ne fait que n échanges alors que le tri par insertion en effectue en 0 et $\frac{n(n-1)}{2}$.

3.4 Un exemple de complexité moyenne

On rappelle qu'on compte le nombre de comparaisons pour déterminer la complexité.
Comme la complexité du tri par insertion est variable, on peut souhaiter calculer sa **complexité moyenne** pour les listes de taille n. On ne peut pas calculer la moyenne sur toutes les listes de taille n car il y en a une infinité. On considère plus simplement comme toutes les permutations d'une liste de n éléments **distincts**, il y en a $n!$.
Pour une liste $L = [a_1, a_2, \dots, a_n]$ d'éléments distincts, on note L_σ la liste obtenue en permutant les termes avec la permutation σ : $L_\sigma = [a_{\sigma(1)}, \dots, a_{\sigma(n)}]$.
La complexité moyenne recherchée est donc $\overline{C}(n) = \dfrac{1}{n!}\displaystyle\sum_{\sigma\in S_n} C(L_\sigma)$ où $C(L_\sigma)$ est le nombre de comparaisons effectuées lors du tri par insertion de L_σ.
On remarque que cette moyenne ne dépend pas de la liste L avec n termes distincts.
On a vu que $C(L_\sigma) = \displaystyle\sum_{i=0}^{n-1} C_{ins}(L_\sigma, i)$ où $C_{ins}(L_\sigma, i)$ est le nombre de comparaisons effectuées lors de l'insertion du terme d'indice i parmi les i premieres.
Si le terme d'indice i est placé à la position j, on a comparé le terme avec $L_\sigma[i-1]$, $L_\sigma[i-2]$ jusqu'à $L_\sigma[j-1]$ pour $j \geqslant 0$, soit $i-j+1$ comparaisons. Si le terme d'indice i est placé à la position 0 on ne fait que i comparaisons. Si on note $C_{ins}(L_\sigma, i, j)$ le nombre de comparaisons effectuées pour insérer le terme d'indice i en j, on a donc $C_{ins}(L_\sigma, i, j) = i - j + 1$ pour $1 \leqslant j \leqslant j$ et $C_{ins}(L_\sigma, i, 0) = i$
On admet que les emplacements possible pour le terme d'indice i lors de son insertion sont équiprobables. Il y a $i+1$ valeurs possibles, de 0 à i, chacune obtenue par $\frac{n!}{i+1}$ permutations. Ainsi

$$\begin{aligned}\frac{1}{n!}\sum_{\sigma\in S_n} C_{ins}(L_\sigma, i) &= \frac{1}{n!}\sum_{j=0}^{i}\frac{n!}{i+1}C_{ins}(L_\sigma, i, j) = \frac{1}{i+1}\left(i + \sum_{j=1}^{i}(i-j+1)\right)\\ &= \frac{1}{i+1}\left(i++\sum_{k=1}^{i}k\right) = \frac{1}{i+1}\left(i + \frac{i(i+1)}{2}\right)\\ &= \frac{i}{2} + \frac{i}{i+1} = \frac{i}{2} + 1 - \frac{1}{i+1}\end{aligned}$$

On conclut donc

$$\begin{aligned}\overline{C}(n) = \frac{1}{n!}\sum_{\sigma\in S_n} C(L_\sigma) &= \frac{1}{n!}\sum_{\sigma\in S_n}\left(\sum_{i=0}^{n-1} C_{ins}(L_\sigma, i)\right) = \sum_{i=0}^{n-1}\left(\frac{1}{n!}\sum_{\sigma\in S_n} C_{ins}(L_\sigma, i)\right)\\ &= \sum_{i=0}^{n-1}\left(\frac{i}{2} + 1 - \frac{1}{i+1}\right) = \frac{n(n-1)}{4} + n - \sum_{p=1}^{n}\frac{1}{p} = \frac{n^2}{4} + \frac{3n}{4} - \sum_{p=1}^{n}\frac{1}{p}\end{aligned}$$

La complexité moyenne reste quadratique, il y a simplement une division par 2 du terme en n^2.

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.

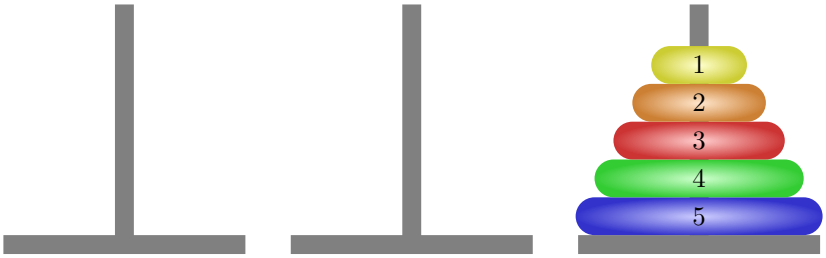
1.2 Tours de Hanoï

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 :

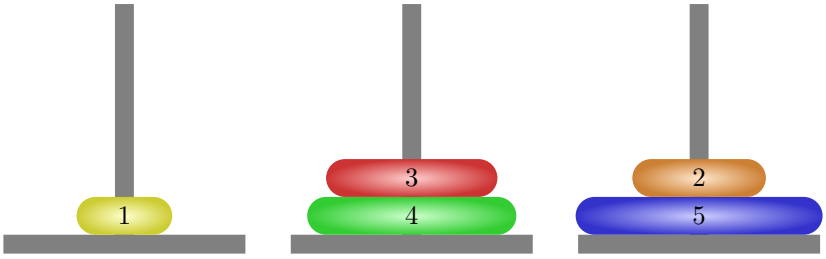
- on ne peut déplacer plus d'un disque à la fois,
- on ne peut placer un disque que sur un autre disque plus grand que lui ou sur un emplacement vide.

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 ? $\sum_{k=0}^{i} u_k = u_i + \sum_{k=0}^{i-1} u_k$.

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.

2.2 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.

2.3 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.

2.4 Complexité

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.

3 Exemples

3.1 Factorielle

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

3.2 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.

3.3 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.

3.4 Coefficients binomiaux

Pour $0 \leqslant p \leqslant n$, $\binom{n}{p} = \dfrac{n!}{p!(n-p)!}$.
Un résultat mathématique classique est que $\binom{n}{p} = \dfrac{n}{p}.\binom{n}{p} n-1 p-1$ 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 $2p$.

3.5 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.

```
def chercher_entre(x, liste, debut, fin):
    """Entrées : un nombre x, une liste de nombres, 2 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 chercher_entre(x, liste, m+1, fin)
        else:
            return chercher_entre(x, liste, debut, m-1)
```

On peut alors en déduire une fonction de recherche dans une liste triée.

```
def appartient(x, liste):
    n = len(liste)
    return chercher_entre(x, liste, 0, n-1)  # on cherche
       partout
```

Terminaison

Pour prouver que la fonction `chercher_entre` termine on note k la valeur de `fin - debut` et on procède par récurrence généralisée sur k.
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.

Preuve

Avec les mêmes notations, si on a $k < 0$, la réponse `False` est correcte car x ne peut pas apparaître dans une liste vide.

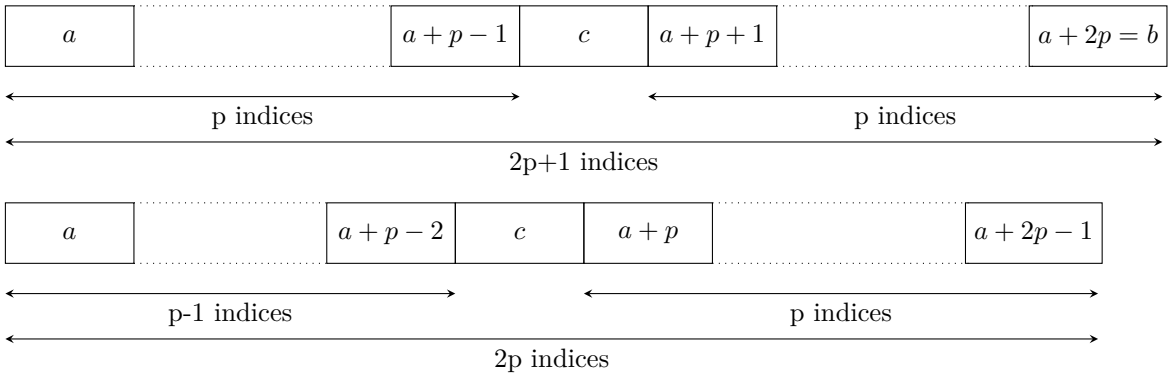
On suppose que l'algorithme renvoie la réponse correcte pour tout k avec $k < k_0$.
Si `fin - debut` vaut k_0 on calcule m : on a `debut <= m <= fin`.

- 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.

Complexité

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

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

Si on note $C(k)$ la borne supérieure des complexités pour traiter k indices, on a donc $C(2p+1) = 2 + C(p)$ 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 $\mathcal{O}\big(\log_2(n)\big)$

4 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 $\begin{cases} F_0 = 0 \\ F_1 = 1 \\ F_n = F_{n-1} + F_{n-2} \quad \text{pour } n \geqslant 2 \end{cases}$

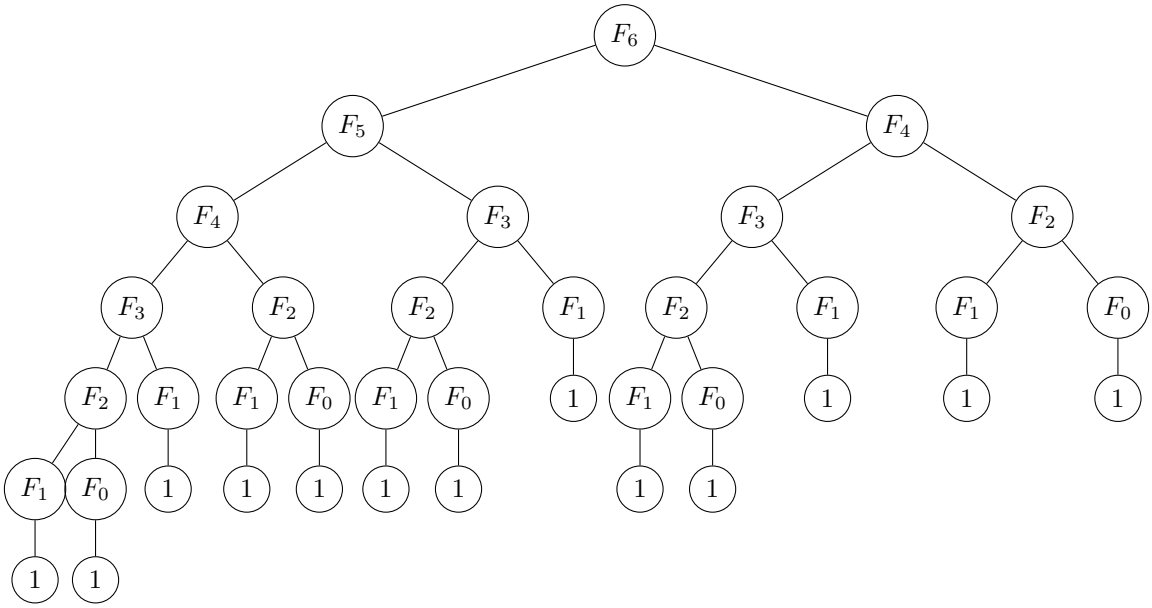
La traduction récursive immédiate est

```
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.
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).

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

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
```

5 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.

5.1 Tri par insertion

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.

1. il n'y a rien à faire si $i = 0$,
2. pour $i \geqslant 1$, on appelle la fonction avec $i - 1$ puis on insère l'élément d'indice i

```
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)
```

5.2 Tri par sélection

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.

```
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)
```

6 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 ?

6.1 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 $(F_{k_2}, F_{k_2}, \ldots, F_{k_r})$ 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 $(F_{k_2}, F_{k_2}, \ldots, F_{k_r})$ 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.

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.*

6.2 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 9 — 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.

2. Par récurrence sur n.

7 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 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.

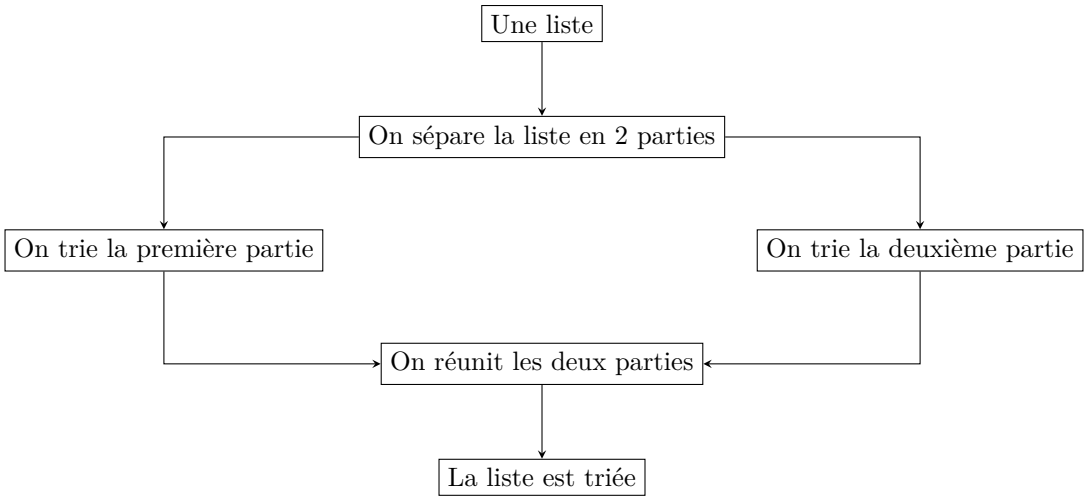
On a laissé dans le flou la signification de "*trier les $n-1$ premiers éléments*" et "*on trie les $n-1$ derniers éléments*", les détails sont demandés dans les exercices du chapitre précédent.

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 pivot

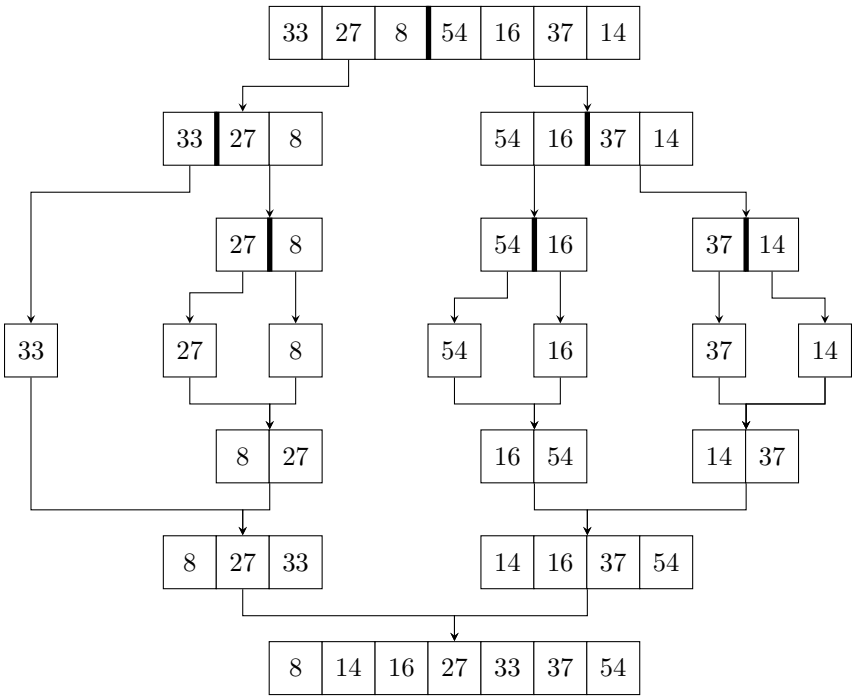
On supposera que les éléments des listes à trier sont comparés à l'aide d'une fonction, nommée `plusGrand`, qui renvoie `True` si le premier argument est supérieur (ou égal) au second.
Dans le cas d'éléments simples, la fonction `plusGrand` est simple aussi

```
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.



On montre les détails de la dernière étape à la page suivante.

2.2 Écriture en python

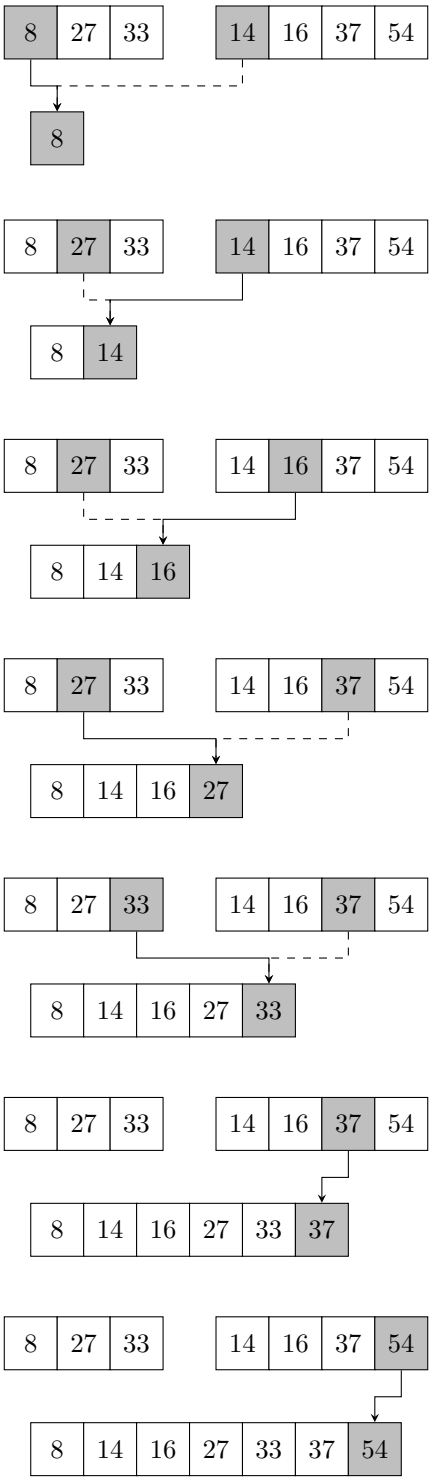
Lors de la séparation de la liste en deux, la taille des listes diminue sauf dans le cas d'une liste de longueur 0 ou 1. C'est un cas terminal,dans ce cas on renvoie une copie de la liste.
On voit que les éléments sont placés sans qu'il semble possible de le faire avec des échanges simples ; on les place dans une nouvelle liste, ce qui impose de définir des listes supplémentaires.

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, illustrée à la page suivante, est de placer, dans l'ordre, les éléments des deux listes en choisissant le plus petit des éléments non encore placés.

Dans l'exemple les 5 premiers éléments sont ajoutés en comparant effectivement les premiers éléments non encore utilisés dans chaque liste. Par contre les deux derniers éléments sont placés sans comparaison car la première liste a complètement été placée.



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     while pos1 < n1 and pos2 < n2:
12         if plusGrand(l2[pos2],l1[pos1]):
13             resultat.append(l1[pos1])
14             pos1 = pos1 + 1
15         else:
16             resultat.append(l2[pos2])
17             pos2 = pos2 + 1
18     if pos1 == n1:
19         return resultat + l2[pos2: n2]
20     else:
21         return resultat + l1[pos1: n1]
```

- 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 indiqués par 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 des éléments restant dans les deux listes.
- Lignes 12-14 Si le plus petit élément restant est dans la première liste on l'ajoute au résultat et on incrémente la position du premier élément à comparer.
- Lignes 15-17 Si le plus petit élément restant est dans la seconde liste on l'ajoute au résultat.
- Ligne8 18-21 En sortie de la boucle une des deux liste a été complètement utilisée et on ajoute ce qui reste de l'autre au résultat.

On a employé une boucle `while`, on peut préférer la sécurité d'une boucle `for`

```
def fusion(l1,l2):
    n1 = len(l1)
    n2 = len(l2)
    resultat = [0]*(n1+n2)
    pos1 = 0
    pos2 = 0
    for i in range(n1+n2):
        if pos1 == n1:
            resultat.append(l2[pos2])
            pos2 = pos2 + 1
        elif pos2 == n2:
            resultat.append(l1[pos1])
            pos1 = pos1 + 1
        elif plusGrand(l2[pos2],l1[pos1]):
            resultat.append(l1[pos1])
            pos1 = pos1 + 1
        else:
            resultat.append(l2[pos2])
            pos2 = pos2 + 1
    return resultat
```

2.3 Analyse

Terminaison

`fusion` termine car `i1+i2` augmente de 1 au moins à chaque étape, on parvient donc à `i1 >= n1` ou `i2 >= n2` après un nombre fini de passages. Dans le cas d'une boucle `for` elle termine naturellement.
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.
Une propriété vérifiée à chaque passage dans la boucle `while` est que la liste `resultat` est triée et tous ses éléments sont majorés par les éléments de `l1[pos1:n1]` et de `l2[pos2:n2]`.
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 $n_1 = n \div 2$ et $n_2 = n - n \div 2$ que l'on trie puis on fusionne les listes triées.
$n \div p$ désigne la division euclidienne, notée `n // p` en Python.

- $n_1 = n_2 = p$ pour n pair, $n = 2p$,
- $n_1 = p$, et $n_2 = p+1$ pour n impair, $n = 2p+1$

La complexité du tri, $C(n)$, vérifie donc $C(n) \leqslant C(n_1) + C(n_2) + n$.

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 $n \leqslant 2^{p+1}$.

- Pour n pair, $n_1 = n_2 = \frac{n}{2} \leqslant 2^p$,
- pour n impair on a $n \leqslant 2^{p+1} - 1$ et $n_1 < n_2 = \frac{n+1}{2} \leqslant 2^p$.

On en déduit, d'après l'hypothèse de récurrence, $C(n_1) \leqslant p2^p$ et $C(n_2) \leqslant p2^p$ d'où
$C(n) \leqslant C(n_1) + C(n_2) + n \leqslant p2^p + p2^p + 2^{p+1} = (p+1)2^{p+1}$: 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

$$C(n) \leqslant \big(1 + \log_2(n)\big)2n = \mathcal{O}\big(n \log_2(n)\big)$$

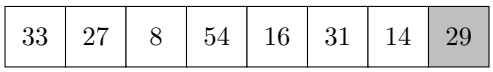
3 Tri pivot (Quicksort)

Le tri pivot 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 selon le pivot : d'un coté les élément plus petits que le pivot, de l'autre les éléments plus grands. 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.

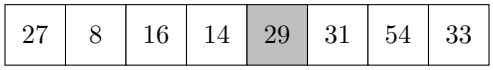
3.1 Écriture du découpage

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 pourrait utiliser le premier élément de la liste ou 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, que l'on nommera **segment**. 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



en un segment, qui pourrait être

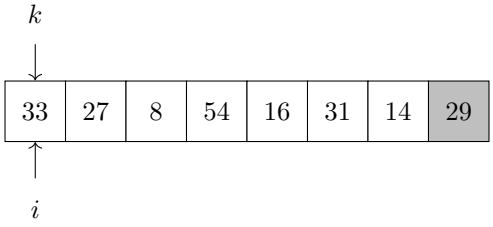


On peut remarquer que le pivot est à sa place. En effet les termes placés avant lui sont inférieurs et les termes placés ensuite sont supérieurs.

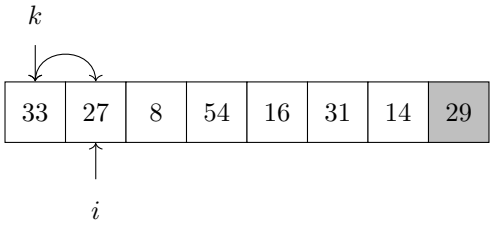
On parcourt la liste entre a et $b-1$ puisque le pivot est en b.
On maintient 2 variables d'indice

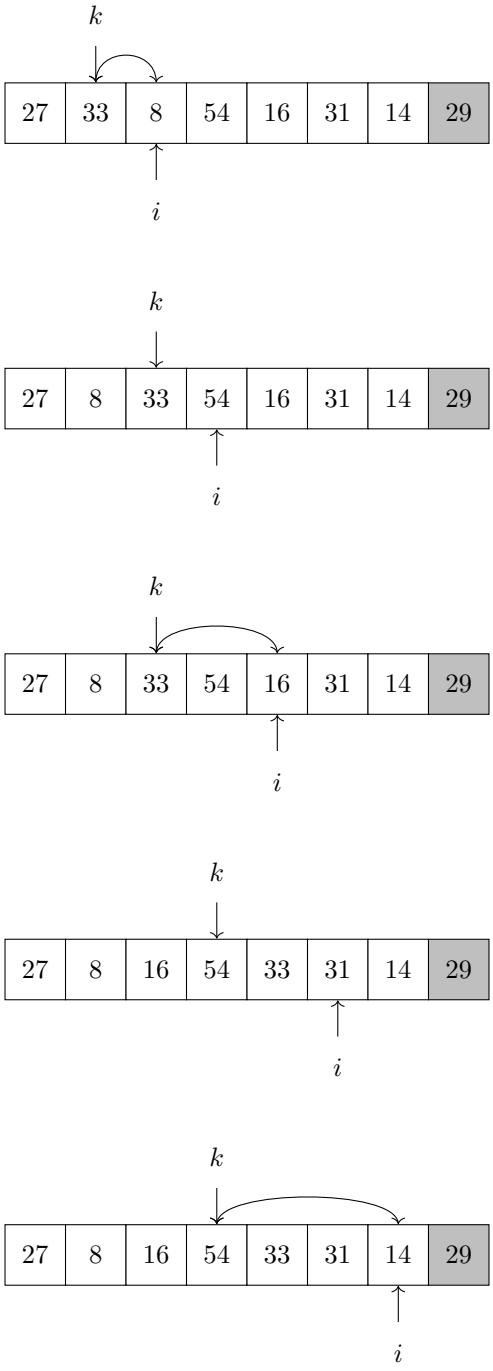
- la première, i, est l'indice de la boucle `for` qui est la position de l'élément étudié,
- la seconde, k, est la première position après les termes plus petits que le pivot.

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

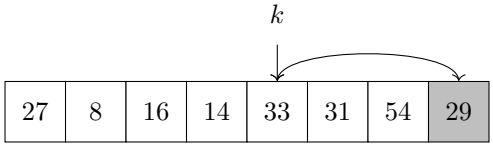


S'il est plus petit, on le permute pour le mettre à la position k puis on incrémente k.





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.

Pour permuter deux éléments dans une liste on utilise la fonction classique.

```
def echanger(liste,a,b):
    """Entree : une liste et deux entiers
       Requis : 0 <= a <= b < len(liste)
       Sortie : les valeurs dans la liste
                aux position a et b sont échangées"""
    temp = liste[a]
    liste[a] = liste[b]
    liste[b] = temp
```

Programme IV.3 – Découpage pour le tri rapide

```
def pivotage(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]):
            echanger(liste, i, k)
            k = k + 1
    echanger(liste, k, b)
    return k
```

3.2 Écriture du tri

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

Programme IV.4 – Tri rapide

```
def tri_entre(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 = pivotage(liste, a, b)
        tri_entre(liste, a, p-1)
        tri_entre(liste, p+1, b)

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

3.3 Analyse

Terminaison

`pivotage` termine car elle ne fait appel qu'à une boucle `for`.
On prouve que la fonction récursive `tri_entre` termine par récurrence sur la longueur de la portion à trier car, lors de chaque appel récursif, celle-ci diminue de 1 au moins.
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. le pivot est à la dernière position durant la boucle.

On peut alors prouver que les propriétés suivantes forment un invariant.

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

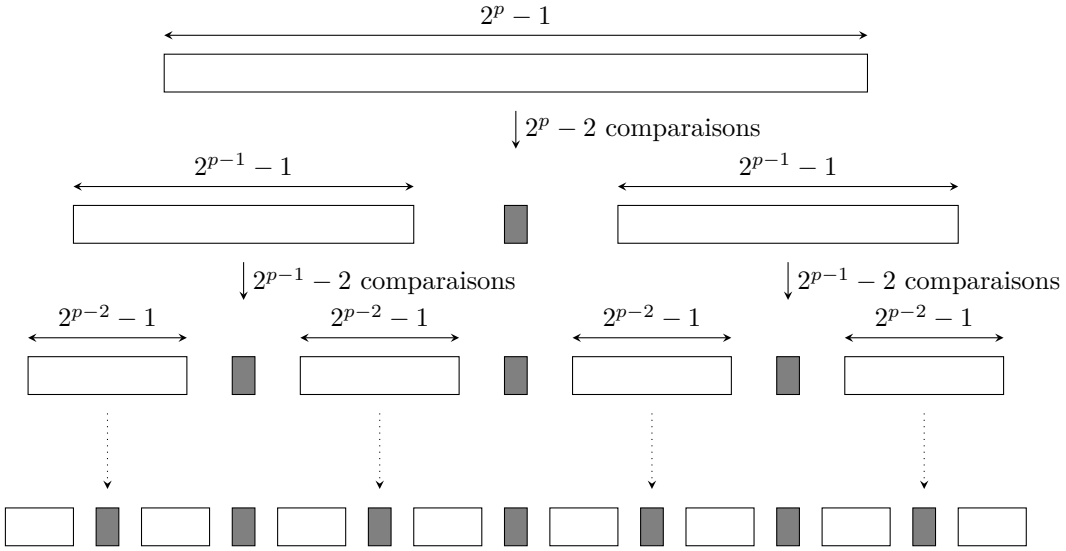
La preuve du tri s'en déduit.

Complexité

`pivotage(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 `tri_entre(l,a,b)`.
On a donc $C(l,a,b) = b-a+C(l,a,p-1)+C(l,p+1,b)$ si la fonction `pivotage` a renvoyé p.

Exemple 1 Dans le cas où `pivotage(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 `pivotage(l,a,b)` renvoie un indice situé au milieu, en particulier, s'il y a un nombre impair d'éléments, `pivotage(l,a,b)` renvoie $\frac{a+b}{2}$.



À chaque étape on double le nombre de segments et, entre deux segments, il y a un élément qui a été un pivot. Si on note m_k le nombre de segments de taille 2^k-1, il sont séparés par m_k-1 pivots donc on a $2^p-1=m_k.(2^k-1)+m_k-1=m_k.2^k-1$: on en déduit $m_k=2^{p-k}$.
Ces m_k segments occasionnent chacun 2^k-2 comparaisons. Le nombre de comparaison est donc

$$\begin{aligned}C(n)&=\sum_{k=2}^{p}m_k.(2^k-2)=\sum_{k=2}^{p}2^{p-k}.(2^k-2)\\&=\sum_{k=2}^{p}\big(2^p-2^{p-k+1}\big)=\sum_{k=2}^{p}2^p-\sum_{k=2}^{p}2^{p-k+1}\\&=(p-1)2^p-\sum_{i=1}^{p-1}2^i=(p-1)2^p-\big(2^p-2\big)=(p-2)2^p-2\end{aligned}$$

On a $2^p=n+1$ donc $p=\log_2(n+1)$ donc $C(n)=\big(\log_2(n+1)-2\big)(n+1)+2$, la complexité est un $\mathcal{O}\big(n\log_2(n)\big)$, le tri est efficace dans ce cas.

Entre ces deux extrêmes on a le cas moyen :

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

Ce résultat doit être connu mais pas sa démonstration.

3.4 Amélioration

On va améliorer la fonction `pivotage` avec l'introduction d'un choix aléatoire. Cela permettra d'obtenir une complexité qui ne sera pas constante mais dont l'espérance est semblable à la complexité moyenne ci-dessus.
On utilise la fonction `randint(a, b)` du module `random` qui choisit aléatoirement un entier compris, avec les bornes comprises, entre a et b.

Programme IV.5 – Découpage aléatoire pour le tri rapide

```
def pivotage(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 """
    p = random.randint(a, b) # on choisit un pivot au hasard
    echanger(liste, p, b)     # on le place à la fin
    pivot = liste[b]          # on continue comme avant
    k = a
    for i in range(a, b):
        if plusGrand(pivot, liste[i]):
            echanger(liste, i, k)
            k = k + 1
    echanger(liste, k, b)
    return k
```

La complexité devient maintenant une variable aléatoire X_C dont on cherche à déterminer l'espérance ; c'est le nombre de comparaisons effectuée lors du tri.
On note $a_1<a_2<\cdots<a_n$ les n éléments, supposés distincts de la liste triée.
Pour calculer l'espérance on utilise la linéarité. Pour cela on définit la variable aléatoire $X_{i,j}$ qui vaut 1 si a_i et a_j sont comparés à un moment dans le tri et 0 si non.
On a alors $X_C=\sum_{1\leqslant i<j\leqslant n}X_{i,j}$ donc $E(X_C)=\sum_{1\leqslant i<j\leqslant n}E(X_{i,j})$

Pour savoir si a_i et a_j sont comparés, on suit leurs positions dans les parties obtenues par les séparations. Si a_i et a_j sont dans un même segment (ce qui est le cas au début) alors le découpage choisit un pivot m. On suppose qu'on a $i < j$ donc $a_i < a_j$.

- Si on a $m < a_i$ ou $a_j < m$ alors a_i et a_j ne sont pas comparés et restent dans un même segment après le découpage.
- Si on a $a_i < m < a_j$ alors a_i et a_j ne sont pas comparés et sont placés dans des segment distincts : ils ne pourront plus être comparés.
- si $a_i = m$ ou $a_j = m$, l'un des deux est le pivot. Les deux éléments sont donc comparés, le pivot sera isolé et l'autre élément appartiendra à un segment, ils ne seront plus comparés.

On voit donc que a_i et a_j sont comparés si et seulement si, lors du choix d'un pivot entre a_i et a_j, ce pivot vaut a_i ou a_j. Il y a donc 2 cas où $X_{i,j}$ vaut 1 parmi les $j - i + 1$ cas possible du choix d'un pivot dans $\{a_i, a_{i+1}, \dots, a_j\}$.

On en déduit qu'on a $E(X_{i,j}) = \dfrac{2}{j-i+1}$.

On peut donc calculer

$$\begin{aligned}
E(X_C) &= \sum_{1\leqslant i<j\leqslant n} E(X_{i,j}) = \sum_{1\leqslant i<j\leqslant n} \frac{2}{j-i+1} \\
&= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \text{ on pose } k = j-i+1 \\
&= \sum_{i=1}^{n-1} \sum_{k=2}^{n-i+1} \frac{2}{k} = \sum_{k=2}^{n} \sum_{i=1}^{n-k+1} \frac{2}{k} \\
&= \sum_{k=2}^{n} \left(\frac{2}{k} \sum_{i=1}^{n-k+1} 1 \right) = \sum_{k=2}^{n} \frac{2}{k}(n-k+1) \\
&= (n+1) \sum_{k=2}^{n} \frac{2}{k} - \sum_{k=2}^{n} \frac{2k}{k} = 2(n+1)\big(H_n - 1\big) - \sum_{k=2}^{n} 2 \\
&= 2(n+1)H_n - 2(n+1) - 2(n-1) \\
&= 2(n+1)H_n - 4n
\end{aligned}$$

On a posé $H_n = \displaystyle\sum_{k=1}^{n} \frac{1}{k}$ et on sait que $H_n \sim \ln(n)$.

On en déduit que $E(X_C) \sim 2n\ln(n) = \mathcal{O}\big(n\ln(n)\big)$.

On peut donc maintenant espérer une complexité log-linéaire pour le tri de toute liste.