Algorithmique

Objectif : La méthode algorithmique est introduite dans ce chapitre. Des algorithmes classiques comme les algorithmes de tri sont étudiés et mis en œuvre sur des exemples simples mais aussi avec des données plus nombreuses ou plus complexes ce qui justifie les préoccupations d'optimisation des traitements (durée et quantité de mémoire mobilisée).
L'étude du coût temporel ou mémoriel est lié à deux autres problèmes : apporter la preuve qu'un algorithme fait bien ce qu'il est sensé faire (correction de l'algorithme) et qu'il va aboutir à un résultat en un temps fini (terminaison de l'algorithme). La notion d'invariant de boucle va ainsi permettre de prouver la correction d'un algorithme tandis que la nécessité de prouver la terminaison d'un programme sera mise en évidence sur les boucles non bornées, grâce à la mobilisation de la notion de variant.
Plan du cours : Ce chapitre est divisé en deux parties.
La 1re partie est plus théorique et générale. Nous commençons par rappeler les outils de base dont on dispose et qui ont largement été étudiés et utilisés dans le chapitre de programmation : les boucles, les variables, les tests, les types de données. Nous définissons ensuite les concepts qui permettent d'étudier les algorithmes, d'en prouver la validité (correction, terminaison) et d'évaluer le coût (notion de complexité).
La 2e partie est plus pratique et technique. Il s'agit ici d'étudier certains algorithmes, d'en caractériser le fonctionnement à l'aide des outils préalablement définis :

- Parcours séquentiel et recherche dichotomique dans un tableau trié
- Tris par insertion ou par sélection
- Algorithmes gloutons
- Algorithme des k plus proches voisins

Aperçu historique :

La notion d'algorithme est très ancienne. Le terme lui-même vient du nom d'un savant de Bagdad, Abu Abdallah Muhammad Ibn Musa Al-Khwârizmi (780-850). Khwârizmi est le nom de sa province d'origine (dans l'actuel Ouzbekistan) ; toujours est-il que ses travaux ont permis de diffuser les chiffres indo-arabes jusqu'en Europe et qu'ils ont apporté de nouvelles lumières sur un domaine des mathématiques qui porte aujourd'hui un nom dérivé de son ouvrage Kitâb al-mukhtasar fî hisâb al-jabr wa-l-muqâbala (abrégé du calcul par la restauration et la comparaison), autrement dit l'algèbre.
De très anciens algorithme nous sont parvenus comme celui d'Euclide (vers 300 avant J.-C.) pour la détermination du PGCD de deux entiers. Encore plus éloigné de nous, l'algorithme exposé par Héron d'Alexandrie (1er siècle après J.-C.) a sans doute été utilisé par les savants de Babylone pour déterminer l'excellente valeur approchée de $\sqrt{2}$ *(*$1,4142129$ *au lieu de* $1,4142135$*) donnée par la tablette YBC7289 (1900 et 1600 avant J.-C.).*
Les algorithmes dont nous parlons s'appliquent aux mathématiques mais il y en a dans de nombreux autres domaines : les recettes de cuisine en particulier sont des algorithmes. Pour nous déplacer, par exemple pour aller d'un point à un autre dans Paris, nous utilisons des algorithmes : en fonctions des données (temps dont je dispose, météo, heure de la journée, etc.) j'effectue des choix (moyen de transport le plus adapté) qui amène un programme (me rendre à pied à telle station de métro, prendre telle métro jusqu'à telle station, prendre telle sortie, etc.).

Les algorithmes aujourd'hui sont partout : à chaque fois que nous interrogeons un moteur de recherche sur internet, que nous payons un achat avec une carte de crédit, que nous consultons une carte sur notre téléphone portable, nous utilisons plusieurs algorithmes. Ils nous facilitent souvent la vie et nous inquiètent parfois (quel bachelier est entièrement rassuré de savoir que son avenir dépend en partie de l'algorithme Parcoursup ?)[1]
Le théoricien majeur qui a écrit un ouvrage monumental sur la question (The Art of Computer Programming, 1968) est Donald Knuth (1938-). Voici ce qu'il donne comme définition d'un algorithme : un algorithme a des entrées (zéro ou plusieurs) qui lui sont données avant qu'il ne commence et qui sont prises dans un ensemble d'objets spécifié ; il a aussi des sorties (zéro ou plusieurs) ; chaque étape étant définie précisément, les actions à mener doivent être spécifiées rigoureusement et sans ambigüité pour chaque cas ; un algorithme se termine toujours après un nombre fini d'étapes ; toutes les opérations que l'algorithme doit accomplir doivent être suffisamment basiques pour pouvoir être en principe réalisées dans une durée finie par un homme utilisant un papier et un crayon.

1. La méthode algorithmique

a. Outils de programmation

Il faut distinguer l'algorithme du programme : l'algorithme est une description des actions à réaliser qui ne suit pas la syntaxe formelle d'un langage de programmation. On doit pouvoir rédiger un algorithme dans un pseudo-langage qui n'est pas la langue parlée ou écrite, souvent floue ou trop subtile avec son riche vocabulaire, mais qui s'en approche. Une des principales raisons à cela est qu'il existe de très nombreux langages de programmation, et qu'ils n'ont pas tous, et loin de là (d'où leurs différences) les mêmes règles syntaxiques.
Comme dans ce cours nous utilisons presque exclusivement le langage Python, nous écrirons les algorithmes en pseudo-français et leur traduction en programme avec Python.

Boucles

Presque tous les algorithmes utilisent des boucles, et parfois des boucles imbriquées.

Certaines boucles sont non conditionnelles, on les dit bornées.
Elles sont utilisées quand on connait à l'avance le nombre de tours de boucle qu'il faut faire.

Exemple 1 – Pour calculer la somme des n premiers carrés d'entiers, je dois utiliser une boucle bornée.

```
Saisir n (le nombre d'entiers)
Initialiser la somme à 0
Pour i allant de 1 à n
Ajouter i² à la somme
Fin de la boucle "pour"
Afficher la somme
```

```
n=int(input("Combien d'entiers? "))
somme=0
for i in range(1,n+1):   # la boucle
    somme=somme+i**2     #
print("Somme des",n,"premiers carrés=",somme)
```

```
Combien d'entiers? 100
Somme des 100 premiers carrés= 338350
```

La répétition d'une boucle conditionnelle est assujettie à un test.
Le principe est qu'on répète les instructions de la boucle tant qu'une condition est vraie.
D'après ce principe, il est très possible d'écrire une boucle qui ne s'achève jamais. Une telle boucle, qualifiée d'infinie, peut être difficile à détecter. Il s'agit souvent, pour le programmeur débutant, d'une erreur de programmation. On teste par exemple la valeur d'une variable, mais on ne modifie pas cette variable dans le corps de la boucle : si on a pu entrer dans la boucle une 1re fois, on y restera toujours.
La détermination du PGCD de deux nombres par l'algorithme d'Euclide utilise typiquement une boucle conditionnelle (voir l'exemple 5 et aussi l'exemple 4 du chapitre 2).

1. Lire sur ce sujet : Le temps des algorithmes, de Serge Abiteboul et Gilles Dowek, Ed. Le Pommier, 2017.

Exemple 2 – Un nombre n est-il premier (divisible seulement par lui-même et par 1) ?
Pour le savoir, on va diviser n par tous les entiers supérieurs à 1 et inférieurs ou égaux à $\sqrt{n}$ (au-delà c'est inutile) et après chaque division, on examine le reste : si il est nul, le nombre n'est pas premier et on sort de la boucle.

```
Saisir n (l'entier à tester)
Initialiser le reste r à n
Initialiser le diviseur d à 1
Initialiser dmax à l'entier supérieur ou à égal à √n
Tant que r>0 et que d<dmax
Ajouter 1 à d
Affecter le reste de la division de n par d à r
Fin de la boucle "tant que"
Afficher "premier" si d=dmax et r=0
```

```
from math import ceil,sqrt
n=int(input("Quel entier? "))
d,r=1,n
dmax=ceil(sqrt(n))
while r>0 and d<=dmax: #
    d+=1               # la boucle
    r=n%d              #
if r>0 and d>dmax:
    print(n,"est premier")
```

```
Quel entier? 101
101 est premier
```

Remarques :

- Dans une boucle "pour", il y a une variable qui compte les tours de boucle (la variable `i` de mon 1er programme). Qu'on s'en serve ou pas dans le corps de la boucle, cette variable existe toujours. En Python (et dans de nombreux autres langages) on n'a pas besoin de l'incrémenter : par défaut, cette variable augmente de 1 à chaque tour.
  Dans une boucle "tant que", par contre, il faut initialiser un tel compteur si on en a besoin, et l'incrémenter comme il convient dans le corps de la boucle.
- Avec une boucle "tant que", la condition testée doit être vraie au départ pour entrer dans la boucle. J'ai ainsi voulu tester le reste des divisions successives mais, au départ, n'ayant pas fait de division, il m'a fallu initialiser cette variable à une valeur arbitraire (j'ai choisi n mais j'aurai pu mettre n'importe quel nombre non nul).
- Les boucles "pour" peuvent toujours être transformées en boucles "tant que" : il suffit de gérer un compteur et de tester s'il ne dépasse pas la valeur prévue. Cependant, à chaque fois qu'il est possible d'écrire une boucle "pour", il est préférable de le faire. C'est un principe de bonne programmation : il faut toujours privilégier les solutions les plus simples, ne serait-ce que pour bien comprendre le programme.
- Il est possible d'interrompre une boucle depuis l'intérieur de celle-ci : en Python (et de nombreux autres langages), cela se fait grâce à l'instruction `break` disposée dans une instruction conditionnelle. Dans le cas de boucles imbriquées, cette instruction ne fait sortir que de la boucle contenant ce `break`. Une autre possibilité algorithmique offerte par Python (et de nombreux autres langages) est l'instruction `continue` disposée dans une instruction conditionnelle : ce qui suit cette instruction est ignoré et le programme passe au tour de boucle suivant. Des exemples d'utilisation de `break` et `continue` sont donnés dans le chapitre 2.

Instructions conditionnelles

Cet outil permet d'effectuer une instruction ou un bloc d'instructions à condition qu'un test soit vrai. Il existe une option facultative à cette instruction : une instruction ou un bloc d'instructions à condition que le test soit faux.
On notera ces deux possibilités :

- Si <test> alors <instructions> fin du "si"
- Si <test> alors <instructions1> sinon <instructions2> fin du "si"

Dans les deux cas, l'instruction `fin du "si"` permet de savoir où se termine le blocs d'instructions. En Python, il suffit de revenir à l'indentation initiale (après les deux-points, l'indentation est augmentée d'une tabulation, voir le chapitre 2 ou l'exemple qui suit).
Bien sûr, on peut imbriquer des instructions conditionnelles ou les disposer à la suite les unes des autres. La logique algorithmique peut être compliquée à suivre et s'il s'avère qu'elle contient des erreurs, celles-ci peuvent être particulièrement difficiles à détecter. Il est donc conseillé d'envisager tous les cas au moment de la conception et de faire le plus simple possible.

En Python, la contraction `elif` de `else if` permet de ne pas ajouter d'indentation supplémentaire. Cette construction a été étudiée au chapitre 2 (exemple 2, page 7).

Exemple 3 – Combien d'années révolues y a t-il entre deux évènements ?
Les évènements étant définis chacun par une date (jour, mois année), en appelant `d` la différence entre les années : si la date anniversaire de l'évènement initial est passée la réponse est `d`, sinon c'est `d-1`. Mais pour identifier que la date anniversaire de l'évènement initial est passée, il faut tester le mois (si le mois initial est passé, la date est passée) et si celui-ci est identique, le jour.
En plus de ces tests, on peut tester si les dates entrées sont valides : le mois doit être compris entre 1 et 12, le jours entre 1 et le maximum (à déterminer) du mois. On peut aussi vouloir vérifier que la date initiale (date 1) précède la date finale (date 2). Pour simplifier, je vais juste tester ce dernier point (on supposera les dates valides).

```
Saisir la date initiale (j1,m1,a1)
Saisir la date finale (j1,m1,a1)
Si date initiale après date finale
alors afficher "erreur"
Sinon Si m1>m2 ou m1=m2 et j1>j2
alors afficher a2-a1-1
Sinon afficher a2-a1
Fin du dernier "si"
Fin du premier "si"
```

```
j1,m1,a1=input("Date initiale (jour,mois,an)? ").split(",")
j2,m2,a2=input("Date finale   (jour,mois,an)? ").split(",")
j1,m1,a1=int(j1),int(m1),int(a1)
j2,m2,a2=int(j2),int(m2),int(a2)
if a1>a2 or a1==a2 and (m1>m2 or m1==m2 and j1>j2):
    print("date initiale postérieure à date finale")
elif m1>m2 or m1==m2 and j1>j2:
    print("le nombre d'années révolues est",a2-a1-1)
else:
    print("le nombre d'années révolues est",a2-a1)
```

```
    Date initiale (jour,mois,an)? 14,7,1789
    Date finale   (jour,mois,an)? 13,10,2019
    le nombre d'années révolues est 230
```

Pour simplifier l'entrée des données en Python, j'ai utilisé une construction un peu particulière qui découpe la saisie en un tuple, le séparateur étant la virgule (avec la méthode `.split(",")`). Ce tuple est associé aux trois variables `j,m,a` mais comme celles-ci sont encore considérées par Python comme des chaînes de caractères, je les convertis en entiers (lignes 3 et 4) pour pouvoir calculer avec.

Dans certains langages, il existe une structure conditionnelle supplémentaire, le `switch` (aiguillage), qui permet d'effectuer un branchement à partir de la valeur d'une variable. On l'utilise lorsque les cas à gérer sont nombreux. La syntaxe de ce `switch` en C, Java ou php est donnée à gauche, tandis qu'à droite, j'ai traduit ce fonctionnement en Python où le `switch` n'existe pas, contrairement au `elif`.

```
switch (variable) {
case "valeur1": action1; break;
case "valeur2": action2; break;
case "valeur3": action3; break;
case "valeur4": action4; break;
default: actionpardefaut;
}
```

```
if variable==valeur1:
    action1
elif variable==valeur2:
    action2
elif variable==valeur3:
    action3
elif variable==valeur4:
    action4
else: actionpardefaut
```

Utilisation des variables

Les variables qui entrent ou sortent d'un algorithme ne peuvent généralement pas suffire.
On a déjà mentionné les compteurs de boucle qu'ils soient impératifs (boucles non conditionnelles) ou facultatifs (boucles conditionnelles). Certaines variables jouent un rôle de compteur mais comptent autre chose que des tours de boucle : on les utilise pour dénombrer l'effectif d'une certaine population. Les compteurs sont des variables qu'on incrémente ou décrémente de 1 unité à chaque fois.
D'autres variables comptabilisent une quantité sans effectuer de dénombrement. On les appelle des accumulateurs car on y fait la somme de ces quantités.

Exemple 4 – Je veux générer n familles en tirant au hasard le sexe des enfants successifs et en supposant que le couple arrête d'avoir des enfants dès qu'il obtient un garçon. La question est de savoir combien d'enfants en moyenne aura ce type de famille et dans quelle proportion y aura-t-il plus de filles que de garçon dans les familles. Je suppose qu'à chaque naissance, la chance d'avoir un garçon est de $\frac{1}{2}$.

```
Entrer la valeur de n
Initialiser total_enfants à 0
Initialiser plusdefilles à 0
Pour i allant de 1 à n
Initialiser nb_filles à 0
Tant que naissance d'une fille
Ajouter 1 à nb_filles
Fin du "tant que"
Si nb_filles>1
alors incrémenter plusdefilles
fin du "si"
Ajouter nb_filles+1 à total_enfants
Fin du "pour"
Affichage de total_enfants/n
```

```
from random import randint
n=100000                    #nombre de familles
total_enfants=0
plusdefilles=0
for i in range(n):
    nb_filles=0
    while randint(0,1)==0:     #naissance d'une fille
        nb_filles+=1
    if nb_filles>1 : plusdefilles+=1
    total_enfants+=nb_filles+1 #toujours un garçon
print("moyenne=",total_enfants/n)
print("plus de filles :",round(plusdefilles/n*100),"%")
```

```
moyenne= 2.00216
plus de filles : 25 %
```

Dans cet exemple, il y a différentes sortes de compteurs et un accumulateur :

- `i` est un compteur pour la boucle « Pour »
- `nb_filles` est un compteur pour la boucle « Tant que »
- `plusdefilles` est un compteur spécialisé de la boucle « Pour »
- `total_enfants` est un accumulateur

Dans un algorithme, il est souvent nécessaire d'échanger les valeurs des variables entre elles.
En l'absence d'une structure de programmation spécifique, l'outil algorithmique qui permet cela est l'utilisation d'une variable tampon : pour échanger `a` et `b`, on met d'abord `a` dans `c` (la variable tampon), puis `b` dans `a` (cela écrase l'ancienne valeur de `a`) et enfin `c` dans `b` (on ne peut y mettre le contenu de `a` qui a été remplacé).
Python offre la possibilité de se passer de la variable tampon `c`, en utilisant un tuple : on met tout simplement le tuple `a,b` dans le tuple `b,a`.

Exemple 5 – L'algorithme d'Euclide nous donne un bel exemple de ce besoin d'échanger les valeurs. Dans sa version « divisions » successives (l'algorithme existe aussi dans une version « soustractions »), l'ancien diviseur devient le dividende et l'ancien reste devient le diviseur.

```
Entrer la valeur de a
Entrer la valeur de b
Tant que le reste de a/b n'est pas nul
Affecter b à c(tampon)
Affecter le reste de a/b à b
Affecter c à a
Fin du "tant que"
Afficher b
```

```
a=int(input("a? "))
b=int(input("b? "))
while a%b!=0 :
    a,b=b,a%b
print('PGCD(a,b) =',b)
```

```
a? 420
b? 56
PGCD(a,b) = 28
```

Á chaque fois que l'on aura une suite définie par une relation de récurrence, que ce soit une récurrence simple du type $u_{n+1} = f(u_n)$ ou une récurrence portant sur plusieurs termes (par exemple $u_{n+2} = au_{n+1} + bu_n$), il faudra échanger les contenus de plusieurs variables.

Exemple 6 – La suite de Fibonacci, définie par $F_{n+2} = F_{n+1} + F_n$ avec $F_1 = F_0 = 1$ est un exemple typique, où les contenus de trois variables s'échangent d'une façon très semblable à l'algorithme d'Euclide.

```
Initialiser a et b à 1
Entrer la valeur de n
Pour i allant de 1 à n
Affecter a+b à c(tampon)
Affecter b à a
Affecter c à b
Fin du "pour"
Afficher b
```

```
a,b=1,1
n=int(input("n? "))
for i in range(1,n):
    a,b=b,a+b
print('FIBO(n) =',b)
```

```
n? 2
FIBO(n) = 2

n? 10
FIBO(n) = 89

n? 100
FIBO(n) = 573147844013817084101
```

b. Etude des algorithmes

Un algorithme doit être valide :

- il doit produire ce qu'il est sensé produire (correction)
- il doit se terminer en un temps fini (terminaison)

Correction

Un invariant de boucle est une propriété qui est vérifiée avant l'entrée dans une boucle, à chaque passage dans la boucle et à la sortie de boucle. Bien choisi, un invariant de boucle permet de montrer qu'un algorithme est correct.
Pour montrer qu'une propriété est un invariant de boucle, on utilise un raisonnement semblable au raisonnement par récurrence :

- Initialisation : on montre que la propriété est vraie avant d'entrer dans la boucle.
- Hérédité : on montre que si elle est vraie en entrant dans la boucle alors elle est encore vraie à la fin de ce tour de boucle

On déduit de ces deux éléments que la propriété est vraie à la sortie de la boucle. Si cette propriété en sortie de boucle équivaut à réaliser ce que l'algorithme est sensé réaliser, on a bien montré ainsi la correction de l'algorithme.

Exemple 7 – Montrons que le quotient et le reste de la division euclidienne de `a` par `b` peuvent être obtenus grâce à l'algorithme ci-dessous.

```
Entrer les valeurs de a et b
Initialiser q à 0
Initialiser r à a
Tant que r>=b
Affecter r-b à r
Incrémenter q
Fin du "tant que"
Afficher q et r
```

```
a=int(input("a? "))
b=int(input("b? "))
q,r=0,a
while r>=b:
    r,q=r-b,q+1
print('Q=',q,'R=',r)
```

```
a? 15         a? 365
b? 3          b? 7
Q= 5 R= 0     Q= 52 R= 1
```

Montrons qu'à l'étape `q` (quand on entre pour la `q`^{e} fois dans la boucle), `r` contient `a-b×q`.

- Á l'entrée initiale dans la boucle `q=0`, `r=a=a-b×0`. La propriété est donc initialement vérifiée.
- Supposons qu'à la `q`^{e} entrée dans la boucle, `r=a-b×q`.
  Dans le corps de la boucle, `r` est diminué de `b`. Sa nouvelle valeur est donc
  `r'=r-b=a-b×q-b=a-(q+1)×b`. Or `q'=q+1` est la nouvelle valeur de `q`.
  On a donc `r'=a-q'×b`. La propriété est donc vérifiée à la sortie du `q`^{e} tour de boucle.

Á la sortie de boucle, on a `r<b` ce qui est la condition à respecter pour le reste de la division euclidienne par `b`. La valeur de cette variable vérifiant `r=a-q×b` $\iff$ `a=q×b+r` avec `r<b`, cela montre que `q` et `r` sont bien le quotient et le reste cherchés.

Terminaison

Une boucle non conditionnelle se termine toujours et on connait à l'avance le nombre de tours.
Pour montrer qu'une boucle conditionnelle se termine, il faut choisir une expression – un variant – tout ou partie de la condition d'arrêt de la boucle, qui atteint en un nombre fini d'étapes une valeur qui déclenchera la sortie de boucle.
Le plus évident des variants est une variable qui est augmentée à chaque tour de boucle d'une valeur strictement supérieure à zéro, la condition d'arrêt étant que cette variable dépasse une valeur fixée.
De même pour une variable qui est diminuée à chaque tour de boucle d'une valeur strictement supérieure à zéro, la condition d'arrêt étant que cette variable devienne inférieure à une valeur fixée.

La boucle de l'exemple 2 (primalité d'un entier) se termine d'une façon évidente, le variant étant la variable `d` qui est incrémentée de 1 à chaque tour. Ce variant dépassera forcément la valeur `dmax` qui est fixée dès le départ.

La boucle de l'exemple 5 (algorithme d'Euclide) se termine également : à chaque étape de la boucle, `a` et `b` décroissent car, si `a>b`, `a%b` (inférieur à `b`) remplace `b` et `b` (inférieur à `a`) remplace `a`. Si `a<b`, ils

sont échangés en une seule étape du processus ce qui nous ramène au cas précédent. Le variant `a%b` est un entier qui décroît donc strictement à partir de l'étape 2, tout en restant positif. Par conséquent, il finit par atteindre 0 ce qui provoque la sortie de la boucle et la fin de l'algorithme.

Citons, sur ce sujet, un théorème important de l'informatique : il n'existe pas de programme permettant de dire si un algorithme termine toujours ou non. Le problème de l'arrêt (dire si un programme termine ou pas) est indécidable, au sens algorithmique du terme.

Exemple 8 – On aimerait savoir à partir de combien d'étapes, une quantité augmentée à chaque étape d'une valeur égale à un pourcentage `t` de cette quantité, est doublée.
Pour cela on établit l'algorithme suivant qui calcule le coefficient multiplicateur global, pour `n` étapes, et qui teste si ce coefficient dépasse 2.

```
Entrer les valeurs du taux t
Initialiser n à 0
Tant que (1+t/100)^n<2
Incrémenter n de 1
Fin du "tant que"
Afficher n
```

```
t=float(input("Quel taux? "))
n=0
while (1+t/100)**n<2:
    n+=1
print('n=',n)
```

```
Quel taux? 0.2      Quel taux? 0.0001
n= 347              n= 693148
```

Lorsque `t>0` alors on a `1+t/100>1` et le variant `(1+t/100)^n` est croissant quand `n` croit. Mais cela ne suffit pas, une quantité peut croitre sans jamais atteindre certaines valeurs (on dit que la suite est croissante et bornée[2]). Ici, on peut montrer que la suite en question a pour limite $+\infty$ (voir dans le cours de mathématiques : suites géométriques de raison $q > 1$) et donc le variant dépassera la valeur fixée, ici 2, pour une certaine valeur de `n`[3].
Lorsque `t<0` alors on a `1+t/100<1` et le variant `(1+t/100>1)^n` est décroissant. Dans ce cas, il est certain qu'il n'atteindra jamais la valeur fixée. La boucle ne termine pas.

Dans certains cas, la boucle peut, en théorie, ne pas se terminer, mais dans la pratique se termine toujours. Dans l'exemple 4, la boucle « tant que » peut ne pas se terminer (on tire toujours 0, jamais 1 : le couple a une infinité de filles). Cet évènement a une probabilité nulle ($\lim_{n\to+\infty} \frac{1}{2^n} = 0$) et donc, dans la pratique, la boucle se termine rapidement.

Dans d'autres cas, plus problématique, la boucle se termine toujours et pourtant personne n'a encore réussi à trouver un variant qui convienne. La suite de Syracuse vue dans l'exercice 2.11 (chapitre programmation) a cette propriété provisoire : en partant d'un nombre entier quelconque, on détermine le successeur qui est la moitié du nombre si celui-ci est pair, le triple augmenté de 1 sinon. On recommence ensuite avec le successeur, et on poursuit de même jusqu'à obtenir 1. Un jour sans doute, quelqu'un expliquera pourquoi cette suite termine toujours sur 1. Pour l'instant on doit se contenter de le vérifier sur des exemples.

Complexité

Le coût temporel d'un algorithme est une propriété importante car, lorsque la taille des données augmentent, les temps d'exécution peuvent s'allonger démesurément. Pour évaluer le coût temporel d'un algorithme, on estime le nombre d'opérations élémentaires (additions, multiplications, affectations, etc.) qu'il doit effectuer dans le pire des cas, selon les données. On se place dans le pire des cas pour majorer cette estimation : s'il y a une instruction conditionnelle, on choisit la situation où il y a le plus d'opérations à effectuer.
L'objectif est de montrer que le coût temporel – on parle aussi de complexité temporelle– est, soit proportionnel aux données (complexité linéaire), soit proportionnel au carré des données (complexité quadratique), etc.

2. Un bon exemple de suite croissante et bornée : $1 + \frac{1}{4} + \frac{1}{9} + \frac{1}{16} + \frac{1}{25} + \dots$, la somme du carré des inverses de tous les entiers est évidemment croissante mais elle n'atteint jamais 2. Euler montra que sa limite vaut $\frac{\pi^2}{6} \approx 1,644934$

3. Avec la fonction logarithme (étudiée en terminale), on a même le moyen de déterminer par le calcul la valeur de n pour laquelle $\left(1 + \frac{t}{100}\right)^n > 2$: il s'agit de $\frac{\ln 2}{\ln\left(1+\frac{t}{100}\right)}$.

Quand on parle de complexité linéaire, le nombre d'opérations à effectuer peut se mettre sous la forme $\alpha n + \beta$ où n est la taille des données, α et β deux nombres réels positifs. Il est donc majoré par une expression proportionnelle aux données de la forme $\alpha' n$.
Pour une complexité quadratique, le nombre d'opérations à effectuer peut se mettre sous la forme $\alpha n^2 + \beta n + \gamma$, majorée par une expression proportionnelle au carré des données de la forme $\alpha' n^2$.

Exemple 9 – Calcul de la somme des factorielles des entiers inférieurs ou égaux à un entier n.
La factorielle de k est le produit $k! = k \times (k-1) \times (k-2) \times (k-3) \ldots 2 \times 1$.
On doit donc effectuer le calcul $1 + 2 \times 1 + 3 \times 2 \times 1 + 4 \times 3 \times 2 \times 1 + \cdots + n!$
L'algorithme naïf suivant réalise cet objectif :

```
Entrer la valeur de n
Initialiser somme à 0
Pour j allant de 1 à n
Initialiser factorielle à 1
Pour i allant de 1 à j
Multiplier factorielle par i
Fin du "pour" intérieur
Ajouter factorielle à somme
Fin du "pour" extérieur
Afficher somme
```

```
n=int(input("Quel entier? "))
somme=0
for j in range(1,n+1):
    factorielle=1
    for i in range(1,j+1):
        factorielle*=i
    somme+=factorielle
print('somme fact=',somme)
```

```
Quel entier? 10
somme fact= 4037913
```

On voit que cet algorithme est composé de deux boucles bornées imbriquées :

- Dans la boucle intérieure, il y a 2 opérations élémentaires (une affectation et un produit) et celles-ci sont exécutées `j` fois. Cette boucle coûte `2j`.
- Dans le corps de la boucle extérieure, on compte 3 opérations élémentaires (2 affectations et une somme) en plus du coût de la boucle intérieure. Un passage dans la boucle extérieure coûte donc `3+2j` opérations élémentaires (je ne compte pas ici les opérations nécessaires au mécanisme de la boucle).

Le coût total de l'algorithme est donc $\sum_{j=1}^{n} 3 + 2j = 3n + 2\sum_{j=1}^{n} j = 3n + 2\frac{n(1+n)}{2} = n^2 + 4n$.
Cet algorithme a donc une complexité quadratique. La durée d'exécution

- est multipliée par 4 quand la taille des données double (pour $n = 100$ elle devrait valoir environ $2^2 = 4$ fois plus que pour $n = 50$)
- est multipliée par 100 quand la taille des données décuple (pour $n = 1000$ elle devrait valoir environ $10^2 = 100$ fois plus que pour $n = 100$)

Python permet de mesurer les temps d'exécution d'un programme :
La fonction `clock` du module `time` donne un nombre de millisecondes.
Il suffit de faire la différence entre ces nombres à la fin et au début du programme.
Voici les valeurs obtenues :

```
from time import clock
n=int(input("Quel entier? "))
somme=0
temps1=clock()
for j in range(1,n+1):
    factorielle=1
    for i in range(1,j+1):
        factorielle*=i
    somme+=factorielle
temps2=clock()
print('durée=',temps2-temps1)
```

```
Quel entier? 10
durée= 2.68e-05

Quel entier? 50
durée= 0.0005923

Quel entier? 100
durée= 0.0017423

Quel entier? 1000
durée= 0.20663389
```

On remarquera que notre estimation ne tient pas compte du coût temporel réel d'une opération élémentaire. Pourtant ici, les nombres manipulés deviennent immenses (pour `n=1000` la somme comporte 2568 chiffres). Si le programme finit par s'étendre bien davantage que notre estimation grossière ne le prévoit, c'est certainement à cause de l'allongement des temps de calcul pour ces grands nombres. L'estimation rudimentaire que nous avons faite compte toutes les opérations élémentaires de la même façon alors que chacune correspond à un algorithme interne qui possède sa propre complexité temporelle.

Un problème donné peut être résolu par des algorithmes ayant des complexités différentes.

Pour le problème de la somme des factorielles, on peut exhiber un algorithme qui a une complexité linéaire par rapport aux données. Il suffit en effet de réaliser simultanément le produit par `j` de la factorielle et l'ajout de cette factorielle à la somme. Dans ce cas, il n'y a que 4 opérations élémentaires dans la boucle et le coût total de l'algorithme est $\sum_{j=1}^{n} 4 = 4n$, ce qui est bien proportionnel aux données.

```
n=int(input("Quel entier? "))
somme=0
factorielle=1
temps1=clock()
for j in range(1,n+1):
    factorielle*=j
    somme+=factorielle
temps2=clock()
print('durée=',temps2-temps1)
print('somme fact=',somme)
```

```
Quel entier? 10
durée= 1.06e-05

Quel entier? 50
durée= 2.75e-05

Quel entier? 100
durée= 4.84e-05

Quel entier? 1000
durée= 0.0012047
```

La nécessité d'optimiser un code ne se fait pas sentir sur les petites valeurs mais on comprend l'intérêt d'évaluer le coût temporel quand on travaille sur de grandes données. Si on montre qu'un algorithme a une complexité exponentielle (nombre d'opérations élémentaires proportionnel à 2^n par exemple), il sera très difficile, voir impossible d'atteindre certaines valeurs.

2. Problèmes algorithmiques

a. Parcours d'un tableau

Tableaux simples (listes)

La façon la plus simple de chercher un élément dans une liste est de la parcourir du début à la fin, en comparant chaque élément à la valeur cherchée.
Le coût temporel d'un tel algorithme est évidemment proportionnel aux données : dans le pire des cas (si l'élément n'est pas dans la liste), il faut en parcourir tous les éléments. On verra plus loin que si on dispose d'un tableau trié, on peut abaisser la complexité d'une telle recherche avec un algorithme basé sur le principe du « diviser pour régner » (recherche dichotomique).

Exemple 10 – Les chaînes de caractères en Python s'utilisent comme des listes : on peut accéder à chaque caractère en donnant son rang. Par exemple, avec la chaîne `nombre="3,14159"`, si on affiche `nombre[3]` on obtient 4.
Supposons que l'on cherche dans une chaîne de caractères donnant un million de décimales de π [4] une sous-chaine particulière, par exemple votre date de naissance au format `jjmmaa`. On peut se contenter de chercher le rang de la première occurrence d'une telle sous-chaine.
Python offrent un outil pour obtenir ce résultat : la méthode `<maListe>.index("<maSousChaine>")`), mais ici nous voulons obtenir ce résultat par un parcours séquentiel, quitte à comparer ensuite les performances de ce parcours et de la méthode native de Python.

```
def lecture():
    decimales=""
    with open("pi.txt","r") as pi:
        for ligne in pi:
            decimales+=ligne[:-1]
    return decimales

decimales=lecture()
recherche("140759")

recherche("170759")
```

```
def recherche(chaine):
    i,long=0,len(decimales)
    while decimales[i:i+6]!=chaine and i<long:
        i+=1
    if i==long : print("La chaine",chaine,"n'est pas presente")
    else : print("La chaine",chaine,"debute à la décimale",i+1)
```

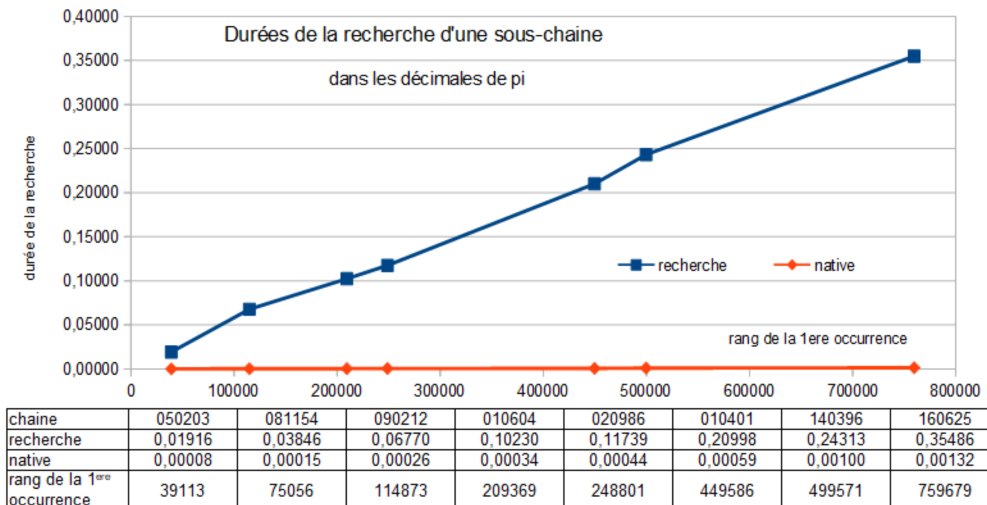
```
La chaine 140759 debute à la décimale 900020

La chaine 170759 n'est pas presente
```

La fonction `lecture()` crée d'abord cette chaîne de un million de caractères à partir d'un fichier texte obtenu sur le site indiqué (ces chiffres sont fournis dans ce fichier par des lignes de 50 caractères suivies du caractère de fin de ligne).

4. Un million de décimales de π sur `http://www.eveandersson.com/pi/digits/` qui procure également un outil pour chercher une sous-chaîne

La fonction `recherche(chaine)` lit la sous-chaîne commençant à l'indice `i` et la compare à celle passée en argument. Si la chaîne est trouvée, le parcours de la liste des décimales s'arrête, sinon il continue jusqu'à la millionième.
Afin de mesurer le coût temporel de cet algorithme et de le comparer à celui de la méthode native, j'insère quelques instructions pour effectuer ces mesures et sélectionne quelques séquences de 6 chiffres ressemblant à des dates présentes dans le fichier :
050203 - 090212 - 081154 - 010604 - 020986 - 010401 - 140396 - 160625



chaine	050203	081154	090212	010604	020986	010401	140396	160625
recherche	0,01916	0,03846	0,06770	0,10230	0,11739	0,20998	0,24313	0,35486
native	0,00008	0,00015	0,00026	0,00034	0,00044	0,00059	0,00100	0,00132
rang de la 1ère occurrence	39113	75056	114873	209369	248801	449586	499571	759679

On ne peut que constater l'allure rectiligne qui signe une belle proportionnalité : la recherche séquentielle a un coût linéaire.
La 2e constatation est encore plus saillante : la méthode native est incomparablement plus rapide. A t-elle tout de même un coût linéaire ? La réponse est oui, mais si il faut environ $0,46$s pour passer en revue un million de décimales avec ma fonction `recherche(chaine)`, il ne faut que $0,0017$s pour la méthode native (270 fois moins).

Tableaux de tableaux

Des tableaux de tableaux sont très souvent utilisées : deux indices permettent alors d'identifier la ligne et la colonne d'une donnée.

- Dans les fichiers CSV les données d'un enregistrement sont matérialisées sur une ligne et les différentes valeurs d'un même enregistrement sont séparées par un caractère particulier, généralement une virgule (d'où le nom puisque le C est celui de *Coma*, virgule).
- Les listes de listes servent à traduire la notion mathématique de matrice de dimension 2 (possèdant lignes et colonnes). On les utilise chaque fois qu'une valeur est caractérisée par deux variables.

Imaginons une liste de coordonnées de points `coord=[[0,1],[5,2],[-1,4]]`.
Pour atteindre l'abscisse du 3e point (-1), il suffit de taper `coord[2][0]` où `[2]` est l'indice de la ligne et `[0]` l'indice de la colonne. En effet, les indices commençant à 0 en Python, la 3e ligne est repérée par l'indice 2 et la 1re colonne est repérée par l'indice 0. Pour comprendre les termes de ligne et colonne, il faut écrire la matrice correspondant à cette liste de listes :

$$\begin{array}{r} \\ \text{ligne } 0 \\ \text{ligne } 1 \\ \text{ligne } 2 \end{array} \begin{array}{c} \begin{array}{cc} \text{colonne } 0 & \text{colonne } 1 \end{array} \\ \begin{pmatrix} 0 & 1 \\ 5 & 2 \\ -1 & 4 \end{pmatrix} \end{array}$$

Cette liste de listes peut se retrouver dans un tableau CSV sous la forme d'un des enregistrements suivant, selon le type de séparateur utilisé :

0,1	0;1	0 1	0/1	0.1
5,2	5;2	5 2	5/2	5.2
-1,4	-1;4	-1 4	-1/4	-1.4

Un algorithme de recherche séquentielle dans un tableau de tableaux examine successivement, selon les besoins, chaque ligne et chaque colonne. Bien sûr, ce principe général est a adapter selon les besoins afin de gagner en efficacité :

- Si on cherche le maximum d'une colonne ou le nombre d'occurrences d'une valeur dans une colonne, il faut lire toutes les lignes et dans chacune, aller directement lire la colonne voulue.
- Si on cherche une valeur particulière, sans savoir dans quelle colonne elle se trouve, il faut examiner successivement les lignes et, pour chacune d'elle, examiner successivement les colonnes jusqu'à la valeur cherchée. Une fois celle-ci trouvée la recherche s'achève (pas besoin d'aller jusqu'au bout).

Exemple 11 – Dans l'exercice 2.34 (chapitre programmation) nous avons enregistré les résultats de la recherche du nombre de points à coordonnées entières appartenant à un même cercle centré sur l'origine et de rayon $n \geqslant 1$ dans le fichier `"cercle.txt"`.
Les cinq premières lignes de ce fichier sont : `1-4-0`, `2-4-0`, `3-4-0`, `4-4-0` et `5-12-0 3 4`.
Le format de ces enregistrements est le suivant :

- le rayon n (un entier) suivi d'un tiret (les rayons sont tous présents à partir de 1)
- le nombre de points à coordonnées entières appartenant au cercle centré sur l'origine de rayon n (un multiple de 4) suivi d'un tiret
- les abscisses de ces points dans le 1er quadrant séparés par un espace s'il y en a plusieurs

L'enregistrement `5-12-0 3 4` par exemple, indique qu'il y a 12 points à coordonnées entières sur le cercle centré sur l'origine et de rayon 5 ; ces points se trouvent, dans le 1er quadrant, aux abscisses 0, 3 et 4.
Cherchons, dans ce fichier, quel est le plus petit rayon d'un cercle contenant 100 points à coordonnées entières. Pour cela, j'écris le programme suivant :

```
def recherche(n):
    with open("cercle.txt","r") as cercles:
        for ligne in cercles:
            rayon,nombre,abscisses=ligne[:-1].split("-")
            if int(nombre)==n:
                return rayon,abscisses.split(" ")
    return []
print(recherche(108))
print(recherche(100))
```

```
('1105', ['0', '47', '105', '169', '264', '272', '425', '468', '520',
'561', '576', '663', '700', '744', '817', '855', '884', '943', '952',
'975', '1001', '1020', '1071', '1073', '1092', '1100', '1104', ''])

('4225', ['0', '468', '580', '615', '1040', '1183', '1625', '2016', '
2047', '2145', '2535', '2652', '2975', '3000', '3289', '3380', '3640'
, '3696', '3713', '3900', '4056', '4095', '4180', '4185', '4199', ''])
```

On est ici dans la recherche d'une valeur particulière, sachant dans quelle colonne lire la valeur cherchée. Le programme lit donc les lignes une à une, transforme chaque ligne en un tableau de trois valeurs : `rayon`, `nombre`, `abscisses`. Si la valeur de la colonne `nombre` correspond à la valeur cherchée, le programme renvoie le `rayon` et un tableau généré à partir du champs `abscisses` (utilisant le séparateur spécifique : un espace). Le résultat est donc 4225 : il faut aller jusqu'à un rayon de 4225 unités pour trouver exactement 100 points à coordonnées entières sur un cercle (en étant centré sur un point à coordonnées entières, ici l'origine).

Quel est le coût temporel d'une telle recherche : dans le pire des cas, ne sachant où se trouve la valeur (elle peut être à la fin du fichier ou pire, ne pas s'y trouver), il faut lire tout le fichier. S'il y a n lignes et p colonnes, le temps d'exécution est proportionnel au produit $n \times p$. S'il s'agissait d'un tableau carré (n lignes sur n colonnes), le temps d'exécution serait proportionnel àn^2 : il s'agirait d'une complexité quadratique.

Je savais qu'on allait trouver un cercle contenant exactement 100 points puisque l'exercice demandait d'arrêter la génération du fichier `"cercle.txt"` dès qu'on trouvait un tel cercle. Pour d'autres valeurs du nombre de points, par contre, on ne peut assurer qu'il s'y trouve. On trouve un cercle contenant exactement 108 points bien plus tôt (un rayon de 1105 unités suffit) mais quel est le 1er multiple de 4 qui ne se trouve pas dans cette liste ? Pour répondre à cette question doit-on lire, potentiellement, tout le fichier pour chacun des multiples de 4 jusqu'à en trouver un pour lequel il n'a pas été trouvé

de cercle ? Il apparait plus judicieux de ne parcourir le fichier qu'une seule fois, en enregistrant, pour chaque ligne, le multiple de 4 qui s'y trouve. Cet enregistrement se faisant dans un dictionnaire où on compte les occurrences de chaque multiple, il suffira ensuite de parcourir ce dictionnaire pour répondre à la question posée.

```
def dictionnaire():
    with open("cercle.txt","r") as cercles:
        dic={}
        maxi=0
        for ligne in cercles:
            rayon,nombre,abscisses=ligne[:-1].split("-")
            if int(nombre) in dic:
                dic[int(nombre)]+=1
            else :
                dic[int(nombre)]=1
                if int(nombre)>maxi :
                    maxi=int(nombre)
    return maxi,dic

maxi,dic=dictionnaire()
for i in range(4,maxi+1,8):
    if i in dic:
        print("nombre :",i," - effectif :",dic[i])
    else :
        print("multiple de 4 non trouvé :",i)
```

```
nombre : 4  - effectif : 1607
nombre : 12  - effectif : 1932
nombre : 20  - effectif : 130
nombre : 28  - effectif : 24
nombre : 36  - effectif : 460
nombre : 44  - effectif : 1
multiple de 4 non trouvé : 52
nombre : 60  - effectif : 54
multiple de 4 non trouvé : 68
multiple de 4 non trouvé : 76
nombre : 84  - effectif : 4
multiple de 4 non trouvé : 92
nombre : 100  - effectif : 1
nombre : 108  - effectif : 12
```

En réalité on ne trouve dans ce fichier que des multiples de 4 qui ne sont pas des multiples de 8 (pour des raisons de symétrie), soit des nombres de la forme $8n + 4$. J'ai affiché les résultats de cette recherche en donnant le nombre d'occurrences des différents nombres de points. La réponse à la question est donc 52 (il se trouve que le 1er ayant ce nombre de points a un rayon de 15625, donc bien supérieur au dernier rayon envisagé dans le présent fichier : 4225).

b. Tri d'un tableau

Trier une liste est une opération algorithmique fondamentale.
Les usages du tri sont multiples, citons en trois :

- Pour accéder à une valeur, si la liste est triée, on peut utiliser la méthode de recherche par dichotomie (voir plus loin) qui a une complexité temporelle (proportionnelle à $\log(n)$) bien inférieure à la recherche séquentielle (proportionnelle à n).
- Pour comparer deux listes (déterminer si elles possèdent les mêmes éléments, pas forcément à la même place), il est beaucoup plus simple de travailler sur des listes triées dont on compare les éléments de même rang.
- Pour déterminer certains paramètres statistiques comme la médiane ou les quartiles, il faut disposer des données triées.

Il y a de nombreuses façons de trier : certains algorithmes de tri sont plus intuitifs mais leur coût est important, d'autres le sont moins mais ils apportent des gains temporels (durée d'exécution d'une complexité inférieure) ou spaciaux (dimension de la mémoire mobilisée inférieure).

Tri par sélection

Le principe de cette méthode de tri est simple : on parcourt toute la liste en cherchant l'élément de valeur minimale ; celui-ci est alors permuté avec l'élément qui était en 1re position ; on recommence ensuite le même processus avec le reste de la liste en augmentant à chaque étape, de une unité la partie triée (placée au début) et en diminuant d'autant la partie non triée (placée à la fin).

Cet algorithme nécessite l'examen de n éléments à la 1re étape, $n-1$ éléments à la 2e étape, etc. jusqu'à la $(n-1)^{\text{e}}$ étape où il ne reste plus qu'un seul élément, ce qui achève l'algorithme. Comme, techniquement, lorsqu'il y a k éléments à examiner, on compare le 1/ier de la liste aux $k-1$ éléments restants, on va effectuer $(n-1)+(n-2)+\cdots+1$ comparaisons au total, soit $\frac{n(n-1)}{2}$. Ce nombre est donc proportionnel au carré de la taille de la liste : la complexité du tri par sélection est quadratique.

Le programme Python ci-dessous réalise le tri par sélection sur une liste.
J'ai ajouté une instruction d'affichage pour visualiser l'état de la liste aux différentes étapes et souligné sur l'affichage en console la partie triée.

```
def tri_selection(t): # tri de la liste en place
    for i in range(len(t)):
        k=i
        for j in range(i+1,len(t)):
            if t[j]<t[k]:
                k=j
        t[i],t[k]=t[k],t[i] # permutation des éléments i et k
        print("état de la liste après",i+1,"étapes:",t)
```

```
t=[12,4,25,0,1,3,8,4,-1]
print("état initial de la liste:",t)
tri_selection(t)
```

```
état initial de la liste: [12, 4, 25, 0, 1, 3, 8, 4, -1]
état de la liste après 1 étapes: [-1, 4, 25, 0, 1, 3, 8, 4, 12]
état de la liste après 2 étapes: [-1, 0, 25, 4, 1, 3, 8, 4, 12]
état de la liste après 3 étapes: [-1, 0, 1, 4, 25, 3, 8, 4, 12]
état de la liste après 4 étapes: [-1, 0, 1, 3, 25, 4, 8, 4, 12]
état de la liste après 5 étapes: [-1, 0, 1, 3, 4, 25, 8, 4, 12]
état de la liste après 6 étapes: [-1, 0, 1, 3, 4, 4, 8, 25, 12]
état de la liste après 7 étapes: [-1, 0, 1, 3, 4, 4, 8, 25, 12]
état de la liste après 8 étapes: [-1, 0, 1, 3, 4, 4, 8, 12, 25]
état de la liste après 9 étapes: [-1, 0, 1, 3, 4, 4, 8, 12, 25]
```

Tri par insertion

Le principe : on compare le 2e élément avec le 1er et s'il lui est inférieur, on permute les deux éléments ; on compare alors le 3e élément avec le 2e : s'il lui est inférieur on permute les deux éléments et on compare ensuite ce 2e élément avec le 1er pour permuter éventuellement les deux éléments ; on recommence ensuite le même processus avec 4e élément, pour le mettre à sa place dans la liste déjà triée des trois premiers ; à chaque étape, la partie triée (placée au début) augmente de une unité.

Cet algorithme nécessite 1 comparaison à la 1re étape, 2 comparaisons (dans le pire des cas) à la 2e étape, etc. jusqu'à la $(n-1)^{e}$ étape où il faut effectuer $n-1$ comparaisons (dans le pire des cas), ce qui achève l'algorithme. On va donc effectuer, dans le pire des cas $(n-1)+(n-2)+\cdots+1=\frac{n(n-1)}{2}$ comparaisons au total, exactement comme le pour le tri par sélection. La complexité du tri par insertion est quadratique.

Le programme Python ci-dessous réalise le type de tri par insertion sur une liste.
Comme précédemment, j'ai ajouté une instruction d'affichage pour visualiser l'état de la liste aux différentes étapes et souligné sur l'affichage en console la partie triée.

```
def tri_insertion(t): # tri de la liste en place
    for i in range(1,len(t)):
        j=i
        while j>0 and t[j]<t[j-1]:
            t[j-1],t[j]=t[j],t[j-1] # permutation des éléments j et j-1
            j-=1
        print("état de la liste après",i,"étapes:",t)
```

```
t=[12,4,25,0,1,3,8,4,-1]
print("état initial de la liste:",t)
tri_insertion(t)
```

```
état initial de la liste: [12, 4, 25, 0, 1, 3, 8, 4, -1]
état de la liste après 1 étapes: [4, 12, 25, 0, 1, 3, 8, 4, -1]
état de la liste après 2 étapes: [4, 12, 25, 0, 1, 3, 8, 4, -1]
état de la liste après 3 étapes: [0, 4, 12, 25, 1, 3, 8, 4, -1]
état de la liste après 4 étapes: [0, 1, 4, 12, 25, 3, 8, 4, -1]
état de la liste après 5 étapes: [0, 1, 3, 4, 12, 25, 8, 4, -1]
état de la liste après 6 étapes: [0, 1, 3, 4, 8, 12, 25, 4, -1]
état de la liste après 7 étapes: [0, 1, 3, 4, 4, 8, 12, 25, -1]
état de la liste après 8 étapes: [-1, 0, 1, 3, 4, 4, 8, 12, 25]
```

Prouvons la validité de l'algorithme de tri par insertion pour une liste $L[0\ldots n]$:
Un invariant de boucle est « à chaque itération i, la sous-liste $L[0\ldots i]$ contient les mêmes éléments que dans la liste de départ, mais triés ».
Initialisation : La propriété est vraie pour $i=0$ car au départ, la sous-liste $L[0]$ ne contenant qu'un élément, elle est bien triée.
Hérédité : On suppose que la propriété est vraie pour la sous-liste $L[0\ldots i-1]$ et on veut montrer qu'elle est alors vraie, à l'étape suivante, pour la sous-liste $L[0\ldots i]$:
L'élément $x=L[i]$ doit être inséré quelque part dans la sous-liste triée $L[0\ldots i-1]$. Pour cela, on déplace successivement les éléments adjacents de cette sous-liste en augmentant leur indice de 1 tant que c'est possible. Lorsque l'élément x trouve sa place (on a rencontré un élément de valeur inférieur), on l'y insère ce qui achève la construction d'une sous-liste $L[0\ldots i]$ triée.

Terminaison : l'algorithme s'achève lorsque $i = n$. D'après l'invariant de boucle, la liste $L[0 \dots n]$ est donc bien triée.

Tri par propagation

Le principe : on compare le 2e élément avec le 1er et s'il lui est inférieur, on permute les deux éléments; on compare alors le 3e élément avec le 2e, puis le 4e avec le 3e (avec échange si nécessaire), etc. jusqu'à arriver en fin de liste (au rang n), le dernier élément étant alors bien placé; on recommence alors depuis le début jusqu'à remonter au rang $n-1$, les deux derniers éléments sont alors bien placés; ainsi de suite, on recommence sur des listes de moins en moins longues, jusqu'à l'achèvement du processus.
Cette méthode de tri n'est pas plus efficace que les précédentes : la complexité est quadratique.
Cette méthode est aussi appelée « tri à bulles », sans doute parce qu'elle déplace rapidement les plus grands éléments en fin de tableau, comme des bulles d'air qui remonteraient rapidement à la surface d'un liquide. Voici un programme qui implémente cette méthode et qui en montre le résultat, étape par étape.

```
def tri_propagation(t): # tri de la liste en place (tri à bulles)
    for i in range(len(t)-1,0,-1):
        for j in range(i):
            if t[j+1] < t[j]:
                t[j+1],t[j]=t[j],t[j+1] # permutation des éléments j et j+1
        print("état de la liste après",len(t)-i,"étapes:",t)

t=[12,4,25,0,1,3,8,4,-1]
print("état initial de la liste:",t)
tri_propagation(t)
```

```
état initial de la liste: [12, 4, 25, 0, 1, 3, 8, 4, -1]
état de la liste après 1 étapes: [4, 12, 0, 1, 3, 8, 4, -1, 25]
état de la liste après 2 étapes: [4, 0, 1, 3, 8, 4, -1, 12, 25]
état de la liste après 3 étapes: [0, 1, 3, 4, 4, -1, 8, 12, 25]
état de la liste après 4 étapes: [0, 1, 3, 4, -1, 4, 8, 12, 25]
état de la liste après 5 étapes: [0, 1, 3, -1, 4, 4, 8, 12, 25]
état de la liste après 6 étapes: [0, 1, -1, 3, 4, 4, 8, 12, 25]
état de la liste après 7 étapes: [0, -1, 1, 3, 4, 4, 8, 12, 25]
état de la liste après 8 étapes: [-1, 0, 1, 3, 4, 4, 8, 12, 25]
```

On constate qu'il y a très peu de différence entre cette méthode de tri et le tri par sélection. La propagation se fait certes en sens inverse, mais surtout, les permutations sont plus nombreuses dans la méthode de propagation puisqu'on échange à chaque fois que c'est possible les éléments au lieu de repérer tout d'abord l'élément à échanger (sélection) et ne procéder à l'échange qu'en fin de parcours de la sous-liste. On va voir cependant que les résultats de ces deux méthodes diffèrent sur un point important : l'une est stable (la propagation) alors que l'autre ne l'est pas (la sélection).

Tri d'une table

Les trois méthodes de tri présentées ici ont l'avantage d'être simple à programmer mais leur performances sont assez mauvaises puisque la complexité est quadratique : si la dimension des données est multipliée par 100, le temps d'exécution du tri est multiplié par $100^2 = 10000$ ce qui peut être un facteur limitant.
Ces méthodes effectuent des tris « en place » : une liste en train d'être triée conserve le même emplacement dans la mémoire; seuls les éléments sont permutés. L'intérêt de ces méthodes va donc au-delà de leur simplicité : elles permettent des gains d'espace mémoire et parfois, lorsque les tableaux à trier sont importants, l'espace de la mémoire devient un facteur limitant qui les fera privilégier comparativement à des méthodes plus rapides mais plus gourmandes en espace mémoire (tri rapide, tri fusion).

Pour trier un tableau de tableaux, provenant des enregistrements d'un fichier CSV par exemple, on peut adapter une des méthodes de tri en place vues précédemment. La comparaison des enregistrements (les lignes du tableau) ne peut se faire sur la base d'une comparaison numérique des enregistrements, mais elle peut se faire sur un des éléments de chaque enregistrements.

Notre fichier `"cercle.txt"` contenant les nombres de points à coordonnées entières appartenant à un même cercle centré sur l'origine et de rayon $n \geqslant 1$ a été généré pour des valeurs de n croissantes. Il est donc trié selon ce critère. Mais si nous voulons maintenant l'obtenir trié selon un autre critère,

par exemple le nombre de points à coordonnées entières qu'il contient (le 2e élément d'une ligne), il faut lui appliquer un des trois algorithmes présentés.
Un tri est stable s'il conserve l'ordre relatif des éléments. Ici, notre liste est triée selon la valeur de n et nous allons opérer un nouveau tri qui risque de faire perdre cet ordre initial alors que ce n'est pas souhaité : si plusieurs enregistrements sont ex-æquo pour le 2e tri, nous voudrions qu'ils conservent l'ordre relatif qui était celui du 1er tri. Un tel tri qui conserverait cet ordre est appelé tri stable. Nous allons voir que, sur ce plan, les trois méthodes ne sont pas équivalentes.

```
def lecture(): #création de la liste de listes
    L=[]
    with open("cercle.txt","r") as cercles:
        for ligne in cercles:
            L.append(ligne[:-1].split("-"))
    return L

def tri_selection(t): # tri de la liste L
    for i in range(len(t)):
        k=i
        for j in range(i+1,len(t)):
            if int(t[j][1]) < int(t[k][1]):
                k=j
        t[i],t[k]=t[k],t[i] # permutation des éléments i et k

table=lecture()
tri_selection(table)
for i in range(len(table)):
    print("rayon:",table[i][0]," - nombre:",table[i][1])
```

Fin de l'affichage

Tri par selection :

```
...
rayon: 3575  - nombre: 60
rayon: 2125  - nombre: 84
rayon: 3250  - nombre: 84
rayon: 1625  - nombre: 84
rayon: 3625  - nombre: 84
rayon: 4225  - nombre: 100
rayon: 2465  - nombre: 108
rayon: 3145  - nombre: 108
rayon: 2405  - nombre: 108
rayon: 3445  - nombre: 108
rayon: 1105  - nombre: 108
rayon: 3770  - nombre: 108
rayon: 2665  - nombre: 108
rayon: 3965  - nombre: 108
rayon: 3485  - nombre: 108
rayon: 1885  - nombre: 108
rayon: 2210  - nombre: 108
rayon: 3315  - nombre: 108
```

```
def tri_insertion(t): # tri de la liste L
    for i in range(1,len(t)):
        j=i
        while j>0 and int(t[j][1]) < int(t[j-1][1]):
            t[j-1],t[j]=t[j],t[j-1] # permutation des éléments
            j-=1                                      j et j-1

table=lecture()
tri_insertion(table)
for i in range(len(table)):
    print("rayon:",table[i][0]," - nombre:",table[i][1])
```

Tri par insertion :

```
...
rayon: 4205  - nombre: 60
rayon: 1625  - nombre: 84
rayon: 2125  - nombre: 84
rayon: 3250  - nombre: 84
rayon: 3625  - nombre: 84
rayon: 4225  - nombre: 100
rayon: 1105  - nombre: 108
rayon: 1885  - nombre: 108
rayon: 2210  - nombre: 108
rayon: 2405  - nombre: 108
rayon: 2465  - nombre: 108
rayon: 2665  - nombre: 108
rayon: 3145  - nombre: 108
rayon: 3315  - nombre: 108
rayon: 3445  - nombre: 108
rayon: 3485  - nombre: 108
rayon: 3770  - nombre: 108
rayon: 3965  - nombre: 108
```

L'adaptation du programme de tri ne pose pas de problème : ici, comme les valeurs sur lesquelles portent les comparaisons sont des chaînes de caractères, il faut les convertir en entiers. J'ai donc écrit une fonction de lecture qui transforme les enregistrements du fichiers en une liste de listes et ensuite j'ai appliqué les différents programmes de tri étudiés.
On constate que le tri par sélection n'est pas stable : l'ordre des rayons n'est pas conservé avec cette méthode. Par contre, le tri par insertion et le tri par propagation sont des tris stables : ils conservent l'ordre des rayons. On le voit sur la fin de l'affichage pour le tri par insertion. Je n'ai pas mis dans l'illustration le tri par propagation car les résultats sont exactement les mêmes que pour le tri par insertion.

Tri Python

Il existe deux méthodes de tri en Python :

- Le tri en place avec la méthode `<maListe>.sort()`. Cette méthode ne renvoie rien mais modifie la liste initiale, un peu comme le ferait une des trois méthodes précédentes, sauf qu'elle est plus rapide (meilleure complexité d'un tri : proportionnelle à $n \log n$).
- La fonction `t2=sorted(<maListe>)` renvoie dans la variable `t2` une copie triée de la liste initiale. L'encombrement en mémoire est plus important puisque la liste et sa copie y sont logées, mais la liste initiale n'est pas modifiée.

Ces méthodes peuvent être adaptée au tri d'une table. Pour cela, il faut indiquer une fonction qui renvoie l'élément sur lequel doit s'opérer le tri. Si cette fonction s'appelle <maFonction> alors la syntaxe Python est t2=sorted(<maListe>, key=<maFonction>) ou bien <maListe>.sort(key=<maFonction>) pour un tri en place.

Donnons un exemple en triant la table construite sur le même fichier "cercle.txt" que précédemment. On veut trier selon le 2e élément de chaque ligne. Pour cela on a besoin d'une fonction nombre(L) qui renvoie cet élément pour une ligne L. Ici encore, il faut convertir cet élément en entier puisque dans le fichier il était sous forme de chaîne de caractères.

```
def nombre(L):
    return int(L[1])

table=lecture()
table.sort(key=nombre)
for i in range(len(table)):
    print("rayon:",table[i][0]," - nombre:",table[i][1])
```

```
rayon: 1105  - nombre: 108
rayon: 1885  - nombre: 108
rayon: 2210  - nombre: 108
rayon: 2405  - nombre: 108
rayon: 2465  - nombre: 108
rayon: 2665  - nombre: 108
rayon: 3145  - nombre: 108
rayon: 3315  - nombre: 108
rayon: 3445  - nombre: 108
rayon: 3485  - nombre: 108
rayon: 3770  - nombre: 108
rayon: 3965  - nombre: 108
```

On constate que cette méthode est stable puisque les enregistrements conservent l'ordre primitif.

c. Recherche dichotomique

La recherche dichotomique (*binary search* en anglais) a pour but de trouver la position d'un élément dans un tableau trié.
Le principe : comparer l'élément cherché avec une valeur centrale (l'élément situé au milieu du tableau) ; la recherche est achevée si les valeurs sont égales, mais si l'élément cherché est classé avant l'élément central on recommence dans la 1re moitié du tableau sinon on recommence dans la 2e moitié.

La dimension du tableau étant divisée par 2 à chaque étape, la recherche s'achève très rapidement : Si le tableau possède n éléments, à l'étape 2 il n'y en a plus que $\frac{n}{2}$, à l'étape 3 il n'y en a plus que $\frac{n}{2^2}$ et à l'étape k il n'y en a plus que $\frac{n}{2^k}$. Or si $\frac{n}{2^k} = 1 \iff n = 2^k \iff k = \log_2 n$, la recherche s'achève à coup sûr (elle a pu s'achever avant si, par hasard, un des éléments centraux était l'élément cherché). Il faut donc, au pire, $\log_2 n$ étapes pour trouver un élément avec cette méthode ce qui est bien plus rapide que la méthode de recherche séquentielle qui demande, au pire, de balayer tout le tableau.

Pour se faire une idée de la rapidité, le tableau ci-dessous donne quelques valeurs de $k = \log_2 n$.
Pour un tableau de dix milliards d'éléments, il suffit de 33 itérations pour trouver n'importe quel élément. Et si on double le nombre d'éléments, pour un tableau de vingt milliards d'éléments, il suffit d'une itération supplémentaire, soit 34 itérations.
Lorsqu'on ne dispose pas d'un tableau trié, l'opération de tri est toujours la plus longue puisqu'elle nécessite au mieux un temps proportionnel à $n \log_2 n$ (nos trois algorithmes précédents sont moins performants puisqu'ils sont proportionnels à n^2).

n	10	100	1 000	10 000	100 000	1 000 000	10 000 000	100 000 000	1 000 000 000	10 000 000 000
$k=log_2(n)$	3,32	6,64	9,97	13,29	16,61	19,93	23,25	26,58	29,90	33,22

Certaines structures spécialisées comme les tables de hachage permettent une recherche plus rapide, mais la recherche dichotomique s'applique à davantage de problèmes et, aussi, elle est très simple à comprendre et à programmer.

Exemple 12 (Recherche dans une liste triée) – Un liste triée de mots (chaînes de caractères triées dans l'ordre lexicographique, sensible à la casse et aux accents) étant donnée, le problème est de savoir si un mot donné en entrée appartient à la liste considérée.
Je récupère sur internet [5] une liste de 336 531 mots du français et écris une fonction qui va lire le fichier "motsFrancais.txt" et constituer la liste triée de mots que je nomme dictionnaire.

5. Liste touvée à l'adresse http://www.pallier.org/liste-de-mots-francais.html

La fonction `iterations(m)` réalise l'objectif de trouver le mot `m` passé en argument, mais de façon séquentielle. La fonction `dichotomie(m)` réalise le même objectif mais de façon dichotomique. Je teste mes deux fonctions sur un mot présent et un mot absent et, pour observer l'efficacité de chacune des méthodes, j'ajoute un compteur de boucle.

```
def liste_mots():
    f=open("motsFrancais.txt",'r')
    liste_mots=f.read()
    f.close()
    mots=liste_mots.split("\n")
    mots.pop(0)
    mots.sort()
    return mots

def iterations(m):
    compteur=0
    for mot_dic in dictionnaire :
        if mot_dic == m :
            print("Séquentiellement,",m,"est au rang",compteur,"(",compteur,"iterations)")
            return None
        compteur+=1
    print("Séquentiellement, la recherche n'a rien donné (",compteur,"iterations) pour",m)

def dichotomie(m):
    debut,fin=0,len(dictionnaire)-1
    compteur,trouve=0,False
    while not trouve and debut <= fin :
        compteur+=1
        milieu=(debut+fin)//2
        if dictionnaire[milieu] == m : trouve = True
        elif dictionnaire[milieu]< m : debut = milieu+1
        else : fin = milieu-1
    if trouve : print("Par dichotomie,",m,"est au rang",milieu,"(",compteur,"iterations)")
    else : print("Par dichotomie, la recherche n'a rien donné (",compteur,"iterations) pour",m)

dictionnaire = liste_mots()
mot='anticonstitutionnellement'
dichotomie(mot)
iterations(mot)
mot='anticonstitution'
dichotomie(mot)
iterations(mot)
```

```
Par dichotomie, anticonstitutionnellement est au rang 14261 ( 19 iterations)
Séquentiellement, anticonstitutionnellement est au rang 14261 ( 14261 iterations)
Par dichotomie, la recherche n'a rien donné ( 18 iterations) pour anticonstitution
Séquentiellement, la recherche n'a rien donné ( 336532 iterations) pour anticonstitution
```

Moins de 20 itérations sont nécessaires pour chercher n'importe quel mot avec l'algorithme de dichotomie alors que la recherche séquentielle peut avoir à balayer toute la liste.
Cette recherche n'a été donnée en exemple que pour illustrer le fonctionnement de la méthode dichotomique. Python sait facilement déterminer l'index d'une valeur dans une liste : il suffit d'écrire `if mot in dictionnaire : dictionnaire.index(mot)`. Le test permet de détecter les cas d'absence de la liste, qui déclencheraient, s'il n'y était pas, une erreur à l'exécution.

d. Algorithmes gloutons

Dans un problème d'optimisation chaque solution possède une valeur et on cherche à déterminer, parmi toutes les solutions, celle dont la valeur est optimale. De nombreux algorithmes sont possibles pour résoudre de tels problèmes, mais l'algorithme glouton est le plus simple. On verra qu'il ne trouve pas forcément l'optimum global.
Un algorithme glouton ou gourmand (*greedy algorithm* en anglais) est un algorithme qui suit le principe de faire, étape par étape, un choix optimum local. En d'autres termes, on effectue une succession de choix, chacun d'eux étant celui qui semble être le meilleur sur le moment. Après chaque prise de décision, on résout alors le sous-problème qui en découle. La simplicité de la méthode gloutonne vient du fait qu'on ne remet jamais en question un choix déjà effectué.

Exemple 13 (Rendu de monnaie) – On suppose un ensemble de pièces de différentes valeurs (on peut supposer également la présence de billets, sans que cela change fondamentalement le problème), chacune étant supposée disponible de façon illimitée.
Lors d'un achat, on doit rendre à un client la somme S. Quelles pièces faut-il lui rendre pour que le nombre de pièces rendues soit le minimum possible.
La méthode gloutonne consiste à faire le choix de rendre systèmatiquement la pièce de plus grande valeur possible et de recommencer le même type de choix avec la somme restant à rendre. Suivant l'ensemble des pièces disponible (la monnaie), l'algorithme glouton est optimal ou pas.
Dans le système de pièces européen, les valeurs étant 1, 2, 5, 10, 20, 50, 100 et 200, pour une somme à rendre de 46 centimes, l'algorithme glouton donne le résultat : 20+20+5+1 qui est la solution optimale. On admettra le résultat suivant : avec les pièces de la monnaie européenne, le choix glouton donne toujours la solution optimale. Mais cela n'est pas vrai pour tous les systèmes de monnaie.
Avec un autre système de pièces, avec les valeurs 1, 3, 8, 23, 44 et 72 par exemple, si on a une somme à rendre de 46 centimes, l'algorithme glouton propose une solution à trois pièces 44+1+1 alors que la solution optimale n'en compte que deux de 23. Cette solution optimale ne peut pas être trouvée par le choix glouton qui privilégie toujours aveuglément la plus grande pièce.

Revenons à l'algorithme glouton lui-même et programmons le rendu de monnaie pour une somme à rendre S lorsque les pièces ont des valeurs qui appartiennent à la liste
`monnaie=[1,2,5,10,20,50,100,200]`.

```
def rechercheGlouton(s) :
    k=len(V)
    listePieces=k*[0]
    while s>0:
        while V[k-1]>s:
            k-=1
        s-=V[k-1]
        listePieces[k-1]+=1
    return listePieces

V=[1,2,5,10,20,50,100,200]
S=46
k=len(V)
listePieces=rechercheGlouton(S)
print("Avec la monnaie:",V)
print("Pour rendre",S,"il faut",sum(listePieces),"pièces:")
for rg,val in enumerate(V):
    if listePieces[rg]!=0:
        s='s' if listePieces[rg]>1 else ''
        print(listePieces[rg],"pièce"+s+" de",val)
print("")
V=[1,3,8,23,44,72]
k=len(V)
listePieces=rechercheGlouton(S)
print("Avec la monnaie:",V)
print("Pour rendre",S,"il faut",sum(listePieces),"pièces:")
for rg,val in enumerate(V):
    if listePieces[rg]!=0:
        s='s' if listePieces[rg]>1 else ''
        print(listePieces[rg],"pièce"+s+" de",val)
```

```
Avec la monnaie: [1, 2, 5, 10, 20, 50, 100, 200]
Pour rendre 46 il faut 4 pièces:
1 pièce de 1
1 pièce de 5
2 pièces de 20

Avec la monnaie: [1, 3, 8, 23, 44, 72]
Pour rendre 46 il faut 3 pièces:
2 pièces de 1
1 pièce de 44
```

La fonction `rechercheGlouton(s)` réalise cette méthode : tant que la somme `s` à rendre n'est pas nulle, on rend la pièce de plus grande valeur possible. Chaque fois qu'une pièce peut être rendue, on diminue d'autant la somme à rendre.

Quelle est le coût de cet algorithme ? Dans le pire des cas, il faut rendre le nombre maximum de pièces de chaque type pouvant être rendu sans l'intervention d'une pièce de rang supérieur. Avec les pièces de monnaies européennes, on peut rendre au pire 1 pièce de 1, 2 pièces de 2, 1 pièce de 5, 1 pièce de 10, 2 pièces de 20, 1 pièce de 50 et 1 pièce de 100, ce qui fait 1+2+1+1+2+1+1=9 pièces inférieures à 200 et totalise 210 centimes.
Si la somme est vraiment importante, on rendra ensuite autant de pièces de 200 que nécessaire (on suppose ici l'absence des billets). D'où, si la somme est importante, un nombre d'itérations approximativement proportionnel à $\frac{S}{Vmax}$ où $Vmax$ est la valeur de la pièce de plus grande valeur (les 9 petites pièces rendues dans le pire cas finissent par ne plus compter si $\frac{S}{Vmax}$ devient très grand).

e. Algorithme des k-plus proches voisins

L'algorithme des k-plus proches voisins – désigné en anglais par *k-nearest neighbors*, d'où l'abréviation *k-nn* – est un des plus simples algorithmes d'apprentissage. Ces algorithmes appartiennent à la catégorie du *maching-learning*, une partie de l'intelligence artificielle, et leurs applications sont dors et déjà innombrables et en constante progression. Il s'agit de trouver des structures, des régularités dans les observations afin de prédire des résultats sur de nouvelles observations. L'idée est d'acquérir de l'expérience sur une tâche pour améliorer la performance sur cette tâche : on montre par exemple des images de chats et de chiens à une machine afin qu'elle réussisse progressivement à mieux différencier un chat d'un chien.

L'algorithme *k-nn* considère en entrée un jeu de données identifiées par une étiquette (on parle aussi de classe ou de *label*) et une donnée, l'objectif étant d'inférer en sortie l'étiquette de la donnée passée en entrée en fonction des étiquettes des plus proches voisins de cette donnée.
Une donnée est décrite par un ensemble de caractéristiques numériques (*features*) des données à décrire, les mêmes pour toutes les données. Les classes sont représentées par un entier. Une donnée étiquetée selon n dimensions est donc représenté par un $(n+1)$-uplet $(x_1, x_2, \cdots, x_n, y)$ où les x_i sont les caractéristiques de la donnée et y son étiquette de classe.
Dans la suite, nous allons réduire l'espace des données à 2 afin de simplifier l'étude et sa représentation graphique. Notre jeu de données sera une liste d'une soixantaine de fruits dont la description comporte x_1 : la masse (en g), x_2 : la taille (en cm) et y : la catégorie (citron, mandarine, orange ou pomme).
La liste des données est présentée ci-dessous :

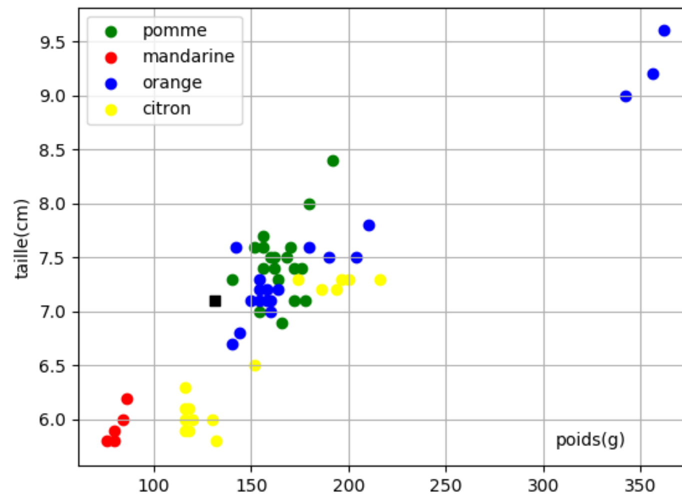
	0			1			2			3			4			5		
1	192	8.4	1	166	6.9	1	160	7.5	1	158	7.1	3	154	7.1	3	130	6.0	4
2	180	8.0	1	172	7.1	1	156	7.4	1	210	7.8	3	180	7.6	3	116	6.0	4
3	176	7.4	1	154	7.0	1	140	7.3	1	164	7.2	3	154	7.2	3	118	5.9	4
4	86	6.2	2	164	7.3	1	170	7.6	1	190	7.5	3	194	7.2	4	120	6.0	4
5	84	6.0	2	152	7.6	1	342	9.0	3	142	7.6	3	200	7.3	4	116	6.1	4
6	80	5.8	2	156	7.7	1	356	9.2	3	150	7.1	3	186	7.2	4	116	6.3	4
7	80	5.9	2	156	7.6	1	362	9.6	3	160	7.1	3	216	7.3	4	116	5.9	4
8	76	5.8	2	168	7.5	1	204	7.5	3	154	7.3	3	196	7.3	4	152	6.5	4
9	178	7.1	1	162	7.5	1	140	6.7	3	158	7.2	3	174	7.3	4	118	6.1	4
10	172	7.4	1	162	7.4	1	160	7.0	3	144	6.8	3	132	5.8	4			

Pour une nouvelle observation, ici un nouveau fruit dont on a mesuré la masse et la taille (x_1 et x_2), on veut prédire sa catégorie y. Voici ce qu'on doit faire :

1. On calcule les distances entre l'observation $X(x_1, x_2)$ et chacune des autres observations du jeu de données
2. On retient les k observations du jeu de données les plus proches (en terme de distance) de X
3. On détermine le mode (la catégorie la plus représentée) de la sélection

La catégorie modale de l'étape 3 est alors la prédiction que l'on attribue à cette nouvelle observation.
Il existe plusieurs fonctions de calcul de distance : distance euclidienne, distance de Manhattan, de Minkowski, de Jaccard, de Hamming, etc. On choisit la fonction de distance en fonction du type des données qu'on manipule. Pour les données quantitatives (poids, salaires, taille, montant du panier électronique, etc.) et du même type, la distance euclidienne est un bon candidat. La distance de Manhattan est une bonne mesure à utiliser quand des caractéristiques numériques ne sont pas du même type (âge et sexe, longueur et poids, etc.).
Ici, nous allons donc utiliser la distance de Manhattan [6], qui est définie entre $X(x_1, x_2)$ et $X'(x_1', x_2')$ par $d = |x_1 - x_1'| + |x_2 - x_2'|$. Pour donner du sens aux distances calculées, il vaut mieux travailler avec des grandeurs comparables : les poids en grammes étant près de 23 fois plus importants que les tailles en centimètres (moyennes des poids : 163g, des tailles :7cm), je vais diviser les poids par 23.
Voici une représentation graphique du jeu de données.

6. La distance de Manhattan est, métaphoriquement, la distance que parcourt un piéton sur les trottoirs de Manhattan où les rues, dessinant un quadrillage, force le piéton à en suivre les directions.



Passons maintenant au traitement : nous voudrions savoir le type probable du fruit caractérisé par les données suivantes $X(x_1 = 131.2, x_2 = 7.1)$) et localisé par le carré noir dans la représentation graphique. Visuellement, il est plus proche d'une pomme mais dans la zone il y a davantage d'oranges. Intuitivement, on comprend qu'avec un $k = 1$, la prévision sera `"pomme"` alors que si $1 < k < 5$ ce sera sans doute `"orange"`. Le choix de la valeur de k est déterminant, généralement on ne prend pas $k = 1$ qui donne trop de poids aux valeurs particulières. On va choisir pour la suite, $k = 3$.

1. Avec la fonction `lecture()` ci-dessous, je lis mon fichier `"fruits.txt"` et constitue la liste `L` des données. Chaque enregistrement contient le poids, la taille et la catégorie du fruit.
2. La fonction `distances(x,L)` détermine ensuite les distances entre la nouvelle donnée `x` et les données de la liste `L`. Ces distances constituent la liste `D` où un enregistrement contient la distance à `x` et la catégorie du fruit.
3. La liste `D` est triée selon les distances croissante (la clé du tri utilise la fonction `distance(x)`).
4. De cette liste, on extrait les $k = 3$ premiers résultats.
5. Pour déterminer la catégorie majoritaire, je constitue un dictionnaire dénombrant les effectifs de chaque catégorie, puis identifie la clé de la catégorie majoritaire en choisissant dans la liste des clés `candidats` celle qui se rencontre en premier dans la liste des `D` des distances triées.

```
def lecture(): #création de la liste des listes
    L=[]
    with open("fruits.txt","r") as fruits:
        for ligne in fruits:
            ligne=ligne[:-1].split(",")
            ligne[0]=float(ligne[0])/23# poids normalisé (/ par rapport des moyennes) en g
            ligne[1]=float(ligne[1])   # taille en cm
            ligne[2]=int(ligne[2])     # catégorie (1:pomme,2:mandarine,3:orange,4:citron)
            L.append(ligne)
    return L

def distances(x,L):
    def distance (x1,x2):# pour dimension = 2
        return round(abs(x1[0]-x2[0])+abs(x1[1]-x2[1]),2)
    D=[]
    for i in range(len(L)):
        D.append([distance(x,L[i]),L[i][2]])# on enregistre distance et catégorie
    return D

def distance(x):
    return x[0]

L=lecture()
x=[131.2,7.1]
x[0]=x[0]/23
D=distances(x,L)
D.sort(key=distance)
D=D[:3];print(D)
categories={1:0,2:0,3:0,4:0}
for i in range(len(D)):
    categories[D[i][1]]+=1
candidats=[c for c,v in categories.items() if v==max(categories.values())]
for i in range(len(D)):
    if D[i][1] in candidats:break
print(categories,"\n",candidats,"\n",D[i][1])
```

```
D           — [[0.58, 1], [0.78, 3], [0.82, 3]]
categories  — {1: 1, 2: 0, 3: 2, 4: 0}
candidats   — [3]
choix final — 3
```

Cet algorithme peut être appliqué à chacun des points de l'espace dans lequel on travaille : en déterminant la catégorie la plus probable selon le jeu de données et selon la valeur du paramètre k, on dresse ainsi une cartographie des catégories. Ci-dessous, j'ai transformé le programme principal précédent en une fonction `categorie(x)` qui détermine, comme on vient de l'expliquer, la catégorie d'un fruit localisé par ses coordonnées `x`. Le programme principal s'occupe de parcourir tout l'espace de la représentation, en plaçant à chaque fois un carré de la couleur de la catégorie la plus probable de fruit. Cette cartographie utilise quelques fonctions du module `matplotlib.pyplot` que j'importe au début.

```
import matplotlib.pyplot as plt
def categorie(x):
    x[0]=x[0]/23
    D=distances(x,L)
    D.sort(key=distance)
    D=D[:3]
    categories={1:0,2:0,3:0,4:0}
    for i in range(len(D)):
        categories[D[i][1]]+=1
    categorie=max([categories[k] for k in categories])
    candidats=[c for c,v in categories.items() if v==max(categories.values())]
    for i in range(len(D)):
        if D[i][1] in candidats:
            return D[i][1]
    return False

L=lecture()
xMax,yMax=365,9.6
xMin,yMin=75,5.8
couleur=['','green','red','blue','yellow']
xPas,yPas=(xMax-xMin)/70,(yMax-yMin)/50
coordonnees=[]
for i in range(70):
    for j in range(50):
        coordonnees.append([xMin+i*xPas,yMin+j*yPas])
for coord in coordonnees:
    plt.scatter(coord[0],coord[1],\
                color=couleur[categorie(coord)],marker='s',s=15)
plt.xlabel("poids(g)")
plt.ylabel("taille(cm)")
plt.grid()
plt.savefig("fruits.png")
plt.show()
```