COURS

ALGORITHMIQUE

&

PROGRAMMATION

LANGAGE PYTHON

ISEL 2

Prérequis

Bases de l'algorithmique, et applications en langage Python (re-voir polycopié ISEL 1).

Bibliographie

« Algorithmique, Techniques fondamentales de programmation, exemples en Python », F. Ebel, S. Rohaut, Editions ENI, 2018, 510 p.

« The art of computer programming », D. Knuth,, Vol. 1-4, Addison-Wesley, 2015.

« Apprendre à programmer avec Python 3 », G.Swinnen, Eyrolles, 2017, 435 p.

« Apprenez à programmer en Python », V. Le Goff, Eyrolles, 2016, 420 p.

« Apprendre à programmer en Python », F. Laroche, Ellipses, 2015, 334 p.

« Algorithmes et structures de données », N. Wirth, Eyrolles, 1989, 320 p.

1. Introduction

Ce cours fait suite au cours d'algorithmique et programmation d'ISEL 1.
Le plan général pour l'ensemble de l'année est le suivant :

- Rappels d'algorithmique élémentaire et de syntaxe Python
- Approfondissement des structures de données
- Approfondissement algorithmique (propriétés de suites mathématiques, récursivité, tris, résolution de systèmes linéaires,…)
- bibliothèques de sous-programmes
- fichiers de données

2. Quelques rappels

Pour revoir les notions ci-dessous plus en détail, se reporter au polycopié de première année. Ce qui suit dans cette section est essentiellement un résumé des principales notions déjà vues. Il vous appartient de les revoir.

2.1. Algorithmique et principes généraux

Rappelons que l'analyse descendante constitue sans doute la méthode la plus efficace et la plus sûre pour résoudre un problème informatique.

Cette démarche a été synthétisée sous la forme suivante par J. Arsac (un des pionniers de l'enseignement de l'informatique en France) :

```
Définir le problème à résoudre:
      expliciter les données
        préciser: leur nature
        leur domaine de variation
        leurs propriétés
      expliciter les résultats
        préciser: leur structure
        leur relations avec les données
fin définir;

Décomposer le problème en sous-problèmes;

Pour chaque sous-problèmes identifié faire
 si solution évidente alors écrire le morceau de
programme
 sinon appliquer la méthode au sous-problème
 fin si
fin pour.
```

Pour tout algorithme, toujours bien détailler les trois parties suivantes :

* La liste des données du programme,
* Le résultat attendu,
* La suite d'instructions pour parvenir au résultat.

Ce qui se traduira sous la forme :

Nom du programme ou sous-programme - liste des données en entrée : noms et types - liste des données attendues en sortie : noms et types du/des résultats -Bloc d'instructions : détail des étapes de l'algorithme

2.2. Instructions décisionnelles

Les instructions de décision permettent de contrôler le flux d'exécution d'un programme selon la valeur d'une condition logique (test).

Il existe plusieurs types d'instructions décisionnelles, bien qu'un seul soit possible en langage Python. Algorithmiquement, cela s'écrit sous l'une des formes suivantes :

Si (condition)
Alors bloc d'instructions exécuté si la condition est vraie
FinSi

ou

Si (condition)
Alors bloc d'instructions 1 – si la condition est vraie
Sinon bloc d'instructions 2 – si la condition est fausse
FinSi

Exemple :

Programme Comparaison

Données d'entrée :
a, b : réels
Résultat de sortie :
Comparaison de a et b
Début
Ecrire('donner les valeurs de a et b ')
Lire (a)
Lire(b)

```
        si (a<=b)
                alors
                        écrire('a est inférieur ou égal à b')
                sinon
        finsi
fin
```

Rappelons également qu'en Python, l'indentation permet de déterminer le début et la fin d'un bloc d'instructions. La syntaxe des instructions de décision est :

if (condition) **:**
    Bloc d'instructions 1
Suite des instructions du programme

Ou

if (condition) **:**
    Bloc d'instructions 1
else :
    Bloc d'instructions 2
Suite des instructions du programme

2.3. Instructions de répétition

2.3.1 Boucle Pour ….Faire…FinPour

Cette structure de boucle est utilisée lorsque le nombre de « tours » de boucle est fixé par un compteur, prédéterminé, et ne dépend pas des résultats éventuels obtenus au fil des répétitions.

Algorithmiquement :
Pour compteur de *a* à *b* (pas par pas, de 1 en général) **faire**
    Bloc d'instructions
FinPour

Avec pour syntaxe Python :

for c **in** séquence_de_valeurs :
    Bloc d'instructions
suite du programme…….

La séquence de valeurs peut être représentée par un intervalle, une liste, une chaîne de caractères, un tuple,…

Dans le cas d'un intervalle, nous aurions par exemple :

```
for i in range(a,b,p) :
        print(i)
suite du programme…….
```

Ici, la variable c prendra les valeurs de **a inclus** à **b exclu** par **pas de p** et le programme effectuera le bloc d'instructions à chaque fois. Attention, si la borne a n'est pas indiquée, le compteur commence à 0 :

```
for i in range(b) :      # i variera de 0 à b-1 par pas de 1
        print(i)
suite du programme…….
```

Dans le cas d'une liste :

```
liste=['Python','C','Java','PHP']
for c in liste :
        print(c)
suite du programme…….
```

2.3.2 Boucle Tant que….Faire…FinTantQue

Cette structure de boucle effectue des répétitions sans savoir à l'avance combien seront nécessaires.

Algorithmiquement :

Tant que (condition logique – booléenne) = vrai **faire**
    Bloc d'instructions
FinTantQue

La condition logique est testée avant d'entrer dans la boucle.
On peut rapidement percevoir que si aucune variable présente dans la condition de test ne change de valeur au fil des « tours » de boucle, la condition logique restera vraie si elle l'était au départ, ce qui entrainera une « boucle infinie ». Il est donc essentiel de s'assurer que la condition logique deviendra fausse au bout d'un certain nombre d'itérations. C'est pourquoi cette condition s'appelle **condition d'exécution** (et sa négation **condition d'arrêt)**.

Il faut donc :

- Bien définir l'expression logique,
- **Initialiser en amont la/les variable(s) présente(s) dans l'expression** logique,
- M**odifier la/les variable(s) présente(s) dans l'expression** logique de sorte que la condition devienne fausse pour éviter une boucle infinie.

La syntaxe Python d'un tel type de boucle est :

while condition **:**
    bloc d'instructions à répéter
suite du programme

Rappel important : Toujours vérifier que l'on effectue bien le **nombre de passages** souhaités dans la boucle, ET que **le premier et le dernier passage** sont bien ceux prévus également. Il est donc utile de faire 'tourner à la main' l'algorithme pour le premier passage, puis de supposer que l'on a atteint l'avant dernier tour afin de tester la suite d'instructions pour le dernier passage.

2.4. Sous-programmes

Un sous-programme est un groupe d'instructions rassemblées sous un même nom afin d'exécuter des tâches très spécifiques chaque fois que ce sous-programme est appelé (utilisé) dans le déroulement du programme. Nous distinguons les fonctions et les procédures.

Avec l'intégration de fonctions et procédures, la structure générale d'un algorithme sera de la forme :

```
Programme Exemple_d_algorithme_avec_fonctions_et_procédures
Données : rep, entier
Sortie : resultat, entier

Procedure premiereProcedure(donnée : n, entier)
Variable locale : i, entier
Début
        i ← 1
        tant que i < n + 1 faire
                écrire(i)
                i ←i + 1
        écrire(« fin de la procédure d'affichage)
Fin

Fonction premiereFonction(donnée : n, entier)
Variables locales :     c, entier
                        s, entier, variable de sortie
Début
        s ← 0
        pour c de 1 à n par pas de 1 faire
                s ←s + c
        retourner s
Fin

############################################################
```

```
Programme principal
Début
        Ecrire (« Entrez une valeur entière » )
        Lire(rep)
        premiereProcedure(rep)
        Ecrire()                                        ####saut de ligne
        resultat←premiereFonction(rep)
        Ecrire(« resultat de la fonction= »,resultat)
        Ecrire(« fin du programme » )
Fin
```

La transcription avec la syntaxe Python sera donc de la forme :

```
# -*- coding: utf-8 -*-
"""
auteur :.....
"""

#importation des fonctions externes :
from math import *    #si nécessaire

#définition des sous-programmes locaux

def procedure(n):    #affiche les valeurs de 1 à n
        i=1
        while i< n+1 :
            print(i)
        i = i + 1
        print("fin de la procédure d'affichage")

def fonction(n):    #renvoie la somme des valeurs de 1 à n
s=0
   for c in range(1,n+1,1):
      s=s+c
   return s

####################################
#corps principal du programme, de préférence définir les sous-programmes en amont

print("entrez une valeur entière")
rep=int(input())
procedure(rep)
print()
resultat=fonction(rep)
print('résultat de la fonction =', resultat)
print('fin du programme')
```

Rappels :

1) En Python, il n'y a pas de procédure au sens habituel, ce sont des fonctions qui retournent **None** si 'return' n'est pas utilisé ou utilisé sans valeur.

2) Pour récupérer plusieurs résultats en sortie de sous-programme avec une fonction :

```
def modifier(a, b):   #noms 'locaux'
        a = 'nouvelle valeur'
        b = b + 2
        #assignés à de nouveaux éléments
        return a, b     #nouvelles valeurs

x = 'ancienne valeur'
y= 10
x, y = modifier(x, y)
print('x =',x,'et y = ',y)
```

affichera :

x = nouvelle valeur et y = 12

Ou encore :

```
def modifier(a, b):   #noms 'locaux'
        a = 'nouvelle valeur'
        b = b + 2
        #assigne de nouvelles valeurs
        return (a, b)    #nouvelles valeurs

x = 'ancienne valeur'
y= 10
t = modifier(x, y)
print('x =',t[0],'et y = ',t[1])
```

affichera :

x = nouvelle valeur et y = 12

2.5. Récursivité

Déjà abordée en ISEL 1, la notion de récursivité se rencontre dans de nombreux domaines. Reprenons l'exemple des poupées russes. Nous pouvons définir une poupée russe de la manière suivante :

« Une poupée russe est une poupée en bois, creuse, et contenant une poupée russe. »
Cette définition sous-entend que toute poupée russe contient une infinité de poupées russes…Ceci ne correspond donc pas tout à fait à la réalité. Une meilleure définition serait alors :

« Une poupée russe est une poupée en bois, creuse ou pleine ; et si elle est creuse, elle contient une poupée russe. »

Cette seconde définition permet de mettre en évidence les deux points essentiels sur lesquels il est nécessaire de porter son attention pour transcrire la notion de récursivité en algorithmique :

1) L'énoncé du problème doit se référer à au moins un problème de même forme (similaire) **et** plus petit
2) Il faut au moins une clause qui mette fin à la récursivité : un test d'arrêt.

Définition : Une procédure ou fonction P est **récursive** si son exécution peut provoquer un ou plusieurs appels (dits récursifs) à P (elle-même). Il s'agit de récursivité « simple » (ou encore « directe »). On dit que P s'appelle elle-même.

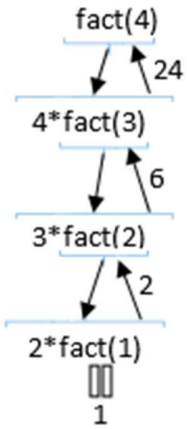
Remarque : La récursivité est dite « croisée » lorsque l'appel d'une procédure ou fonction P1 entraîne celui d'une autre P2, cette autre (P2) appelant à son tour P1.

L'exemple suivant illustre l'appel d'une fonction factorielle récursive :

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

#programme principal
v=int(input('valeur de n = ?'))
print(fact(v))
```

Lors de l'exécution de l'algorithme, nous pouvons représenter la suite d'appels de la fonction *fact(n)* pour n = 4 sous la forme d'un arbre d'appels comme suit :



2.6. Les tuples

Un tuple est une séquence contenant des éléments de types éventuellement différents. Ils diffèrent des listes dans la mesure où il est possible d'ajouter des éléments mais pas d'en modifier (ni supprimer) directement.

La syntaxe de déclaration en Python est de la forme :

mon_tuple = (élément_1, élément_2,……,élément_n)

ou simple_tuple = ('un seul élément',) # noter que dans ce cas, il y a une virgule après l'unique élément.

Exemple :

```
t=('un', 2, 1+2)
print(t[1])      #affichera l'élément de rang 1 (donc le deuxième) du tuple : ici la valeur « 2 ».
```

Les opérations applicables aux tuples sont similaires à celles des listes, à l'exception des modifications qui sont impossibles.

3. Les exceptions

Le langage Python permet de gérer certains types d'erreurs sans interrompre le programme.
La syntaxe générale est la suivante :

```
try
    # bloc d'instruction à tenter de réaliser
except :
    # bloc exécuté en cas d'erreur, i.e. d'impossibilité d'effectuer le bloc précédent (du "try").
    pass    # pour ne rien faire en cas d'erreur
else :
    # bloc d'instructions réalisée quand il n'y a pas eu d'erreur (peu utilisé)
finally :
    # actions effectuées dans tous les cas
```

Si l'on veut préciser (facultatif) l'action à réaliser selon la nature du type de l'erreur rencontrée :

```
try
    # bloc d'instruction à tenter de réaliser
except nom_du_type_erreur :        # ou except (nom_type_erreur1, nom_type_erreur2,...):
```

avec nom_du_type_erreur =

- NameError : non reconnaissance d'une variable (non définie,...)
- TypeError : problème de type dans une instruction (types incompatibles par exemple dans une opération
- ValueError : incompatibilité de type lors de la lecture (saisie) d'une valeur
- ZeroDivisionError : division par zéro

Les sections suivantes précisent quelques exemples selon les cas souhaités.

3.1. Si l'on ne peut pas recommencer après s'être trompé lors de la saisie

```
try:
        b=int(input())
exceptValueError:
        print("ce n'est pas un ENTIER, b ne sera pas affecté\n")
```

3.2. Si l'on ne peut se tromper qu'une seule fois

```
try:
        b=int(input())
except ValueError:
        print("ce n'est pas un ENTIER réessayez:\n")
try:
        b=int(input())
except ValueError:
        print("ce n'est toujours pas un ENTIER, terminé\n")
```

3.3. Si l'utilisateur peut se tromper plusieurs fois

La boucle se terminera avec « break » puisque la boucle est « infinie » :

```
while True:
        try:
                b=int(input())
                break           #interrompt la boucle
        exceptValueError:
                print("saisir un ENTIER \n")
```

3.4. Pour effectuer d'autres actions en même temps

```
print("saisir un entier inférieur à 180 et supérieur à 0 :\n")
x=0             # exception gérant le type + l'intervalle de valeurs pour b
while x==0:
        try:
                b=int(input())
                while b>180 or b<0:
                        print("saisir un entier entre 0 et180\n")
                        b=int(input())
                x=1
        except ValueError:
                print("saisir un ENTIER \n")
```

3.5. Pour tester plusieurs types d'erreurs

```
try:
    b=int(input())
except ValueError:
    print("saisir un second ENTIER \n")
except NameError:
    print("variable non reconnue \n")
…….
```

4. Quelques couleurs dans l'affichage

Il est possible d'améliorer l'affichage avec quelques effets dans la console Python, sans avoir besoin de faire appel à une bibliothèque particulière.
À chaque couleur de texte, chaque couleur d'arrière-plan et chaque mise en forme est associé un code numérique :
Codes couleurs ∈ [30 , 39] : couleur du texte,
Codes couleurs ∈ [40 , 49] : couleur d'arrière-plan.

Codes	Valeurs :
30,40	Noir
31,41	Rouge
32,42	Vert
33,43	Jaune
34,44	Bleu
35,45	Rose
36,46	Cyan
37,47	gris

Formatages supplémentaire :

Codes	Effets :
3	Italique
4	Souligné
7	Surligner

La syntaxe est la suivante :
Ajouter :
\033[*xx* **m** *xx* étant un code couleur
avant le contenu à colorer.
Pour revenir aux couleurs par défaut, *xx* prendra la valeur 0, soit le code : \033[0m
Par exemple :
\033[31m texte en rouge **\033[0m** suite de texte en noir

Ce qui donne l'instruction complète suivante pour afficher un texte en couleur (plusieurs écritures possibles):

print('**\033[31m**' + ' Texte rouge ' + '\033[0m')

print('**\033[31m** Texte rouge \033[0m')

print("**\033[31m**" + " Texte rouge " + "\033[0m")

print("**\033[31m** Texte rouge \033[0m")

print('**\033[32m**' + ' Texte vert ' + '\033[0m')

Pour cumuler couleur de texte et couleur **d'arrière plan** :

\033[34m \033[**45m** texte en bleu et arrière-plan rose \033[0m

Pour cumuler couleur de texte, couleur d'arrière plan, et mise en forme :

\033[34m texte bleu \033[3m texte bleu et italique \033[4m texte en bleu, italique et souligné…\033[0m

Pour appliquer une couleur à des valeurs de variables : appliquer « format() » à la chaîne de caractères :

print("\033[32m{:^3}\033[0m".format(i))

Le cumul direct de codes par écriture compactée est possible :

print("\033[4;31;40m" + " Texte rouge, fond gris, souligné" + "\033[34m")

print("\033[4;31;40m Texte rouge, fond gris, souligné \033[34m")

5. Les dictionnaires

Il s'agit d'un type « composite » : les dictionnaires sont modifiables comme les listes, mais non ordonnés au sens des listes. Pour autant, les éléments sont accessibles via un index appelé « clé » (d'un type numérique, alphabétique, ou composite).
Contenu : n'importe quel type d'éléments.

La syntaxe de déclaration en Python est de la forme :

mon_dico = {clé_1 :valeur_1,……,clé_n :valeur_n}

Quelques opérations :

```
#dictionnaire associant à chaque clé sa traduction en anglais
dic={}
dic['clé']='key' #ajout d'élément
dic['programme']='program'
dic['donnée']='data'
dic['fichier']='file'
dic['répertoire']='directory'
dic['disque']='disk'
print("il y a", len(dic),"entrées dans ce dictionnaire") #len() : nb éléments
print(dic)
print(dic['fichier'])
del dic['programme'] #del : suppression d'une entrée
print(dic)

if "disque" in dic:
    print("disque se traduit par",dic['disque'])
```

Noter que si *dic* est de type dictionnaire, l'appel de la fonction clear() par *dic.clear()* vide totalement la variable *dic*.

Lors d'un parcours, les éléments d'un dictionnaire sont extraits de façon imprévisible (structure non ordonnée).

```
if "disque" in dic:
    print("disque se traduit par",dic['disque'])
print("\n entrées du dictionnaire:")
for i in dic:
    print(i)

print("\n entrées du dictionnaire:")
for i in dic:
    print(i,"=", dic[i])
```

donnera :

```
disque se traduit par disk

 entrées du dictionnaire:
clé
donnée
fichier
répertoire
disque

 entrées du dictionnaire:
clé = key
donnée = data
fichier = file
répertoire = directory
disque = disk
```

```
print("\n \n Contenu du dictionnaire :")
for cle, valeur in dic.items():
    print(cle,"=", valeur)

print("\n\n")
print(dic.items())
```

affichera :

```
 Contenu du dictionnaire :
clé = key
donnée = data
fichier = file
répertoire = directory
disque = disk


dict_items([('clé', 'key'), ('donnée', 'data'),
('fichier', 'file'), ('répertoire', 'directory'),
('disque', 'disk')])
```

La méthode **items()** permet d'extraire les tuples présents dans la structure.
La méthode **keys()** renvoie la séquence des clés du dictionnaire, la méthode **values()** renvoie la liste des valeurs.

```
print("\n \n Clés et valeurs :")
for cle in dic.keys():
    print(cle,"=",dic[cle])

print("\n\n ensemble des valeurs :")
for val in dic.values():
    print(val)
```

affichera :

```
 Clés et valeurs :
clé = key
donnée = data
fichier = file
répertoire = directory
disque = disk


 ensemble des valeurs :
key
data
file
directory
disk
```

Remarque : Les clés peuvent être de tous types : chaînes de caractères, mais aussi des nombres, ou des tuples de coordonnées par exemple si l'on souhaite mémoriser le contenu des cases d'une grille...

6. Algorithmes de tris

Dans ce qui suit on considèrera que les tris à effectuer se feront sur des tableaux d'entiers (ou sur des clés - éléments de types composés, des tuples par exemple).

6.1. Tris élémentaires

6.1.1 Tri par bulles

L'idée principale de ce tri consiste à imaginer que les éléments à trier sont rangés dans un tableau vertical. Les éléments les plus petits sont "moins lourds" et partent comme des bulles vers la surface. On effectue des passages successifs sur le tableau de bas en haut. A chaque étape, si deux éléments sont en ordre décroissant, ils sont inversés. Conséquence : à la fin du premier passage, l'élément le plus petit « remonte » jusqu'à la première position du tableau. Au deuxième passage, l'élément étant le « deuxième plus petit » remonte à la deuxième position du tableau, et ainsi de suite.

Le $i^{ème}$ passage ne s'intéresse donc qu'aux positions situées entre le bas du tableau et la $i^{ème}$ position.

Exemple : Trier la liste 8,5,1,3,7,4.

Pour la première étape :

8	8	8	8	8	1
5	5	5	5	1	8
1	1	1	1	5	5
3	3	3	3	3	3
7	4	4	4	4	4
4	7	7	7	7	7

Le schéma se lit de gauche à droite. La colonne la plus à gauche représente la situation de départ, et la plus à droite, la situation d'arrivée après la première étape. Dans une colonne, les deux nombres soulignés indiquent sur quelles clés se fait le test et l'échange s'il y a lieu.

Pour la deuxième étape, le processus est le même mais on ne s'occupe plus de la valeur en première position. Le tableau suivant indique les différents états du tableau après les différents passages effectués par le tri par bulles. Les éléments en gras représentent ceux sur lesquels on n'effectue plus de tests.

8	**1**	**1**	**1**	**1**	**1**
5	8	**3**	**3**	**3**	**3**
1	5	8	**4**	**4**	**4**
3	3	5	8	**5**	**5**
7	4	4	5	8	**7**
4	7	7	7	7	8
au départ	après i=1	après i=2	après i=3	après i=4	après i=5

Pour n éléments, l'algorithme est de la forme :

```
Données :
        n : entier, t : tableau de n entiers
Sortie :
        t : trié en ordre croissant
variables intermédiaires : i, j : entiers

Début
        Pour i de 1 à n-1 par pas de 1 Faire
                Pour j de n-1 à i-1 par pas de -1 Faire
                        # échange si éléments en ordre inverse
                        Si t[j] > t[j + 1]
                        Alors
                                Echanger(t[j], t[j + 1])
Fin
```

On peut dans certains cas améliorer l'efficacité de l'algorithme du tri par bulles. Si l'on constate à l'étape i, qu'aucun changement n'a été effectué, il est possible d'arrêter le déroulement de l'algorithme. En effet, il était déjà acquis que toutes les valeurs avant (« au-dessus » de) la position i étaient dans le bon ordre, et nous savons maintenant que tous ceux situés entre la position i et la fin du tableau le sont aussi, puisqu' aucun changement n'a été nécessaire. Le tableau est donc totalement rangé.

6.1.2 Tri par insertion

La seconde méthode de tri que nous considérerons est appelée "tri par insertion", car au $i^{ème}$ passage on "insère" le $i^{ème}$ élément T[i] à la bonne place parmi les positions T[1], T[2], ..., T[i-1], qui contenaient précédemment des éléments rangés dans le bon ordre. D'un point de vue algorithmique, on exécute la séquence :

```
Données :
        n : entier, t : tableau de n entiers
Sortie :
        t : trié en ordre croissant
variables intermédiaires : i, valeur, position : entiers

Début
        Pour i:de 2 à n par pas de 1 Faire
                valeur← T[i]
                position  ← i
                Tant que (position >0) et (T[position - 1]>valeur) Faire
                        T[position] ← T[position - 1] # déplacement des valeurs vers la fin
                        position  ← position – 1
                T[position] ← valeur # insertion en bonne place
Fin
```

Sur l'exemple précédent on aurait les séquences suivantes :

8	**5**	**1**	**1**	**1**	**1**
5	**8**	**5**	**3**	**3**	**3**
1	1	**8**	**5**	**5**	**4**
3	3	3	**8**	**7**	**5**
7	7	7	7	**8**	**7**
4	4	4	4	4	**8**
au départ	après i=2	après i=3	après i=4	après i=5	après i=6

6.1.3 Tri par sélection-échange

L'idée directrice du "tri par sélection-échange" est la suivante : au $i^{ème}$ passage, on sélectionne l'élément ayant la plus petite valeur parmi les positions i, ..., n, et on l'échange avec T[i]. Le schéma de l'algorithme est le suivant :

```
Pour i de 1 à n-1 faire
        sélectionner parmi T[i], ..., T[n], l'élément ayant la plus petite valeur
        et l'échanger avec T[i].
```

Ce qui se traduit par :

```
Données :
        n : entier, t : tableau de n entiers
Sortie :
        t : trié en ordre croissant
variables intermédiaires : i, positionMin : entiers

Début
        Pour i de 1 à n-1 par pas de 1 Faire
                positionMin ← i
                Pour j de i+1 à n par pas de 1  Faire
                        Si T[j] < T[positionMin]
                        Alors
                                positionMin ← j
                Echanger(T[positionMin], T[i])
Fin
```

Ceci donnera, en conservant l'exemple précédent :

8	**1**	1	1	1	1
5	5	3	3	3	3
1	8	8	***4***	4	4
3	3	5	5	5	5
7	7	7	7	7	7
4	4	***4***	8	8	8
au départ	après i=1	après i=2	après i=3	……..	après i=6

6.2. Un tri rapide

L'algorithme de tri étudié ici est "le tri rapide" ou "quicksort". Cet algorithme, dû à Hoare (1962) et amélioré par Sedgewik (1975), consiste à partir d'une liste d'éléments, à choisir un élément de comparaison, ou pivot, et à former deux sous-listes. Dans la première on place tous les éléments inférieurs au pivot, et dans la seconde tous les éléments supérieurs au pivot. Cette opération est répétée **récursivement** avec les sous-listes obtenues, jusqu'à ce que le tableau soit entièrement trié.

La création des deux partitions s'effectue de la manière suivante :

- Le tableau est parcouru de gauche à droite (indice i) jusqu'à trouver un élément e_i plus grand que le pivot (ou égal).
- De même, le tableau est parcouru de droite à gauche (indice j) jusqu'à trouver un élément e_j plus petit que le pivot (ou égal).
- Ces deux éléments e_i et e_j sont alors permutés,
- Le parcours du tableau continue jusqu'à ce que les deux indices se croisent (i > j).

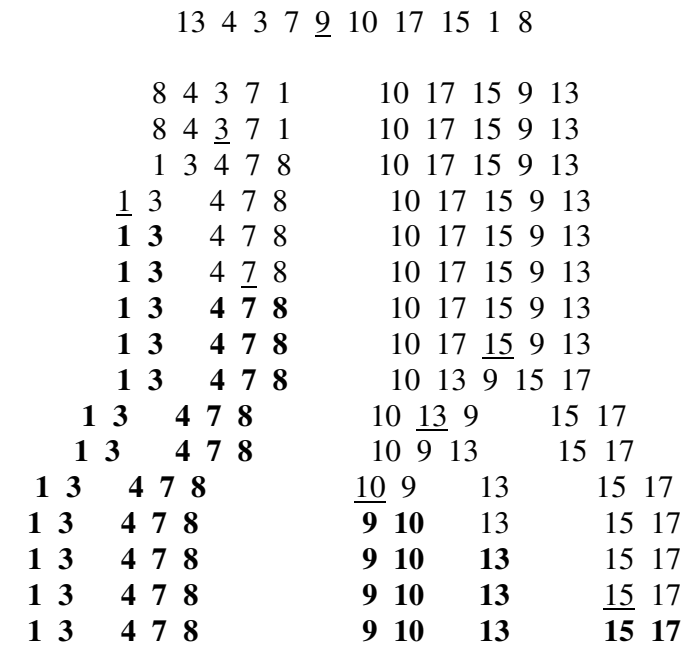
Si l'on suppose alors que l'on appelle la procédure quicksort (*t*, *gauche,droite)*appliquée au tableau *t* (*gauche* et *droite* sont les indices délimitant le sous-tableau en cours de tri : lors du premier appel, *gauche* prend la valeur 1 et *droite* la valeur n), on continue le tri sur les deux sous-tableaux par appels récursifs de la procédure quicksort : si j est strictement supérieur à *gauche* alors on appellera quicksort(*t*, *gauche,* j), et si i est strictement inférieur à droite alors on appellera quicksort(t, i,*droite*).

Le choix du pivot est arbitraire, cependant nous choisirons l'élément médian, ce qui est généralement le cas. La figure suivante illustre le partitionnement d'un tableau de 10 nombres:

13 4 3 7 <u>9</u> 10 17 15 1 8

Les éléments soulignés représentent les pivots, et les éléments en gras les sous-tableaux triés.

Après avoir choisi 9 comme pivot, les éléments sont échangés comme expliqué ci-dessus de sorte que tous les éléments inférieurs à 9 se retrouvent à sa gauche, tous les autres à sa droite. L'on applique récursivement le même principe sur les deux « parties » obtenues à chaque fois en commençant par la partie gauche :



L'algorithme pour un tableau indicé de *gauche* à *droite* sera de la forme :

```
procédure pivot(Données : t : tableau de (droite–gauche+1) entiers ; gauche : entier ;droite :entier)
Sortie :
        t trié en ordre croissant
Variables locales :
        p, i, j : entiers,
Début
        P ← t[(gauche+droite) div 2] # division entière donnant l'indice milieu du tableau
        i ←gauche
        j ←droite
        Tant que  i <= j Faire
                Tant que t[i] < p Faire
                        i += 1
                Tant que t[j] > p:Faire
                        j -= 1
                Si  i <= j
                Alors
                        Echanger(t[i], t[j])
                        i ← i + 1
                        j ← j - 1
        Si gauche < j
        Alors
                pivot(t, gauche, j)
        Si i < droite
        Alors
                pivot(t, i, droite)
Fin
```

Le tri sera appelé pour l'ensemble des positions, c'est-à-dire de 1 à n.

7. Algorithmes de résolution de systèmes linéaires par des méthodes itératives

On cherche à résoudre un système (I) d'équations linéaires:

$$(I) \quad \begin{cases} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n = y_1, \\ a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n = y_2 \\ \qquad\qquad \ldots \\ a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nn}x_n = y_n \end{cases}$$

que l'on écrit sous la forme: $Ax = y$ (1)
où A est la n x n matrice dont les coefficients sont les a_{ij}, y est le vecteur de composantes $y_1,\ldots,y_n$ et x le vecteur de composantes $x_1,\ldots,x_n$.

On supposera dans cette partie que le système admet une solution unique (pour tout second membre) c'est-à-dire que A est une matrice inversible:$\det(A) \neq 0$.

7.1. Une méthode de calcul impraticable

En théorie puisque la matrice A est inversible, on peut calculer son inverse et calculer la solution de l'équation (1) comme:

$$x=A^{-1}y .$$

En pratique cette méthode conduit aux règles de *Cramer* :

$$x_i = \det(A_i)/\det(A)$$

où A_i est la matrice obtenue à partir de A en remplaçant la i-ème colonne par le vecteur y.
Si le calcul des déterminants est effectué comme par définition en développant le déterminant suivant une ligne ou une colonne, on obtiendra un algorithme qui calcule le vecteur x en un nombre d'opérations qui est de l'ordre de $n^2n!$. Pour n=20, le nombre d'opérations est de l'ordre de 10^{20}, ce qui est impossible à effectuer, même sur un calculateur rapide (en effet si l'on suppose de manière optimiste qu'un calculateur rapide effectue un milliard d'opérations par seconde, il lui faudrait environ 75 siècles).
Il est donc évident qu'il faut trouver d'autres méthodes de calcul.

7.2. Méthodes directes et méthodes itératives

Comme dans la plupart des problèmes de l'analyse numérique, deux grandes classes de méthodes peuvent être envisagées :

a) Les méthodes directes : Il s'agit de trouver des suites de calculs qui donnent la solution de façon exacte (en oubliant les erreurs d'arrondi et de données). Pour notre problème il existe essentiellement une méthode dite de "substitution" qui remonte à *Gauss*

b) Les méthodes itératives : On calcule une suite de vecteurs (x^n) qui converge vers le vecteur x solution de (1). On devra alors apprécier l'erreur de troncature $x - x^n$ pour une norme vectorielle à choisir. Nous nous intéresserons dans la suite à ce type de méthode.

7.2.1 La méthode itérative de Jacobi

Considérons l'exemple d'un ensemble de 3 équations à 3 inconnues:

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 = y_1, \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 = y_2 \\ \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 = y_3 \end{cases} \qquad (1.1)$$

On résout la première équation par rapport à x_1, la seconde par rapport à x_2, etc..., ce qui donne:

$$\begin{cases} x_1 = (y_1 - a_{12}x_2 + a_{13}x_3)/a_{11}, \\ x_2 = (y_2 - a_{21}x_1 + a_{23}x_3)/a_{22}, \\ \\ x_3 = (y_3 - a_{31}x_1 + a_{32}x_2)/a_{33} \end{cases} \qquad (1.2)$$

Donnons aux inconnues, les valeurs arbitraires x^0_1, x^0_2, x^0_3: que l'on porte dans le second membre de (1.2), on obtient alors 3 nouvelles valeurs :

$$\begin{cases} x^1_1 = (y_1 - a_{12}x^0_2 + a_{13}x^0_3)/a_{11}, \\ x^1_2 = (y_2 - a_{21}x^0_1 + a_{23}x^0_3)/a_{22}, \\ \\ x^1_3 = (y_3 - a_{31}x^0_1 + a_{32}x^0_2)/a_{33} \end{cases} \qquad (1.3)$$

Ce nouvel ensemble porté dans le 2ème membre de (1.2) donne un autre ensemble de valeurs x^2_1, x^2_2, x^2_3, et ainsi de suite.

On arrête le calcul lorsque deux valeurs successives de x_j sont suffisamment voisines et ceci pour tout j. On obtient alors une solution approchée du système.

7.2.2 La méthode itérative de Gauss-Seidel

Partant comme précédemment du système (1.2), on choisit un ensemble de valeurs x^0_1,x^0_2, x^0_3. On reporte alors x^0_2 et x^0_3 dans la première équation de (1.2), ce qui donne :

$x^1_1 = (y_1 - a_{12}x^0_2 - a_{13}x^0_3) / a_{11}$

C'est cette nouvelle valeur de x_1, et non x^0_1, qui est portée dans la 2e équation de (1.2), donnant :

$x^1_2 = (y_2 - a_{21}x^1_1 - a_{23}x^0_3) / a_{22}$

De même dans la 3e équation de (1.2), on porte x^1_1 et x^1_2 au lieu de x^0_1 et x^0_2 :

$x^1_3 = (y_3 - a_{31}x^1_1 - a_{32}x^1_2) / a_{33}$.

Le principe est d'utiliser automatiquement la plus récente valeur calculée lors de la manipulation d'une inconnue. Ceci assure une convergence bien plus rapide que la méthode de Jacobi.

Quelle que soit la méthode, pour l'arrêt du calcul, on peut utiliser l'un des critères ci-dessous :

a) $\forall 1 \le j \le n, \left| x_j^{k+1} - x_j^k \right| \le \varepsilon$ b) $\forall 1 \le j \le n, \left| \frac{x_j^{k+1} - x_j^k}{x_j^{k+1}} \right| \le \varepsilon$

c) $\sum_{j=1}^{n} \left| x_j^{k+1} - x_j^k \right| \le \varepsilon$ d) $\sum_{j=1}^{n} \left| \frac{x_j^{k+1} - x_j^k}{x_j^{k+1}} \right| \le \varepsilon$

e) $\sqrt{\sum_{j=1}^{n} \left(x_j^{k+1} - x_j^k \right)^2} \le \varepsilon$

f) $\sqrt{\sum_{j=1}^{n} \left(\frac{x_j^{k+1} - x_j^k}{x_j^{k+1}} \right)^2} \le \varepsilon$

La convergence du procédé ne dépend pas du choix des valeurs initiales x^0_j, mais seulement des valeurs des coefficients. De plus la convergence est assurée si l'on a, pour tout i :

$|a_{ii}| \ge \sum_{\substack{j=1 \\ i \ne j}}^{n} |a_{ij}|$

En pratique, on peut aussi observer une convergence dans des conditions moins restrictives.

Table des matières