# Algorithmes classiques

## FICHE SYNTHESE

### PROGRAMME DE PREMIERE:

| Contenus | Capacités attendues | Commentaires |
|---|---|---|
| Parcours séquentiel d'un tableau | Ecrire un algorithme de recherche d'une occurrence sur des valeurs de type quelconque.<br>Ecrire un algorithme de recherche d'un extrémum, de calcul de moyenne | On montre que le coût est linéaire |
| Tris par insertion, par sélection | Ecrire un algorithme de tri.<br>Décrire un invariant de boucle qui prouve la correction des tris par insertion, par sélection | La terminaison de ces algorithmes est à justifier.<br>On montre que leur coût est quadra-tique dans le pire des cas |
| Algorithme des k plus proches voi-sins | Ecrire un algorithme qui prédit la classe d'un élément en fonction de la classe majoritaire de ses k plus proches voisins | Il s'agit d'un exemple d'algorithme d'apprentissage |
| Recherche dichotomique dans un tableau trié | Montrer la terminaison de la re-cherche dichotomique à l'aide d'un variant de boucle | Des assertions peuvent être utilisées. La preuve de la correction peut être présentée par le professeur |
| Algorithmes gloutons | Résoudre un problème grâce à un algorithme glouton. | Exemples: problèmes du sac à dos ou du rendu de monnaie.<br>Les algorithmes gloutons constituent une méthode algorithmique parmi d'autres qui seront vues en terminale |

Le concept de méthode algorithmique est introduit; de nouveaux exemples seront vus en terminale. Quelques algorithmes classiques sont étudiés. L'étude de leurs coûts respectifs prend tout son sens dans le cas de données nombreuses, qui peuvent être préférentiellement des données ouvertes. Il est nécessaire de montrer l'intérêt de prouver la correction d'un algorithme pour lequel on dispose d'une spécification précise, notamment en mobilisant la notion d'invariant sur des exemples simples. La nécessité de prouver la terminaison d'un programme est mise en évidence dès qu'on utilise une boucle non bornée (ou, en terminale, des fonctions récursives) grâce à la mobilisation de la notion de variant sur des exemples simples.

## PREREQUIS

Seule une connaissance générale des algorithmes classiques est supposée.

## PARTIE 0 - ALGORITHMES CLASSIQUES – REVISION

- Version itérative : algorithme basé sur des itérations (for, while)
- Version récursive: algorithme basé sur la récursivité mathématiques (« du haut vers le bas »)
- Programmation dynamique: algorithme traitant les problèmes du « bas vers le haut »

Exemple 1 : **Calcul de factorielle n**

```
def factiteratif(n):
 p=1
 for i in range(2,n+1):
     p=p*i
 return p
```

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

Exemple 2 : **Calcul du n-ième nombre de Fibonacci**

```
def Fibonacciiteratif(n):
 p0=1
 p1=1
 for i in range(2,n+1):
     p2=p0+p1
     p0=p1
     p1=p2
 return p2

def Fibonaccirecursif(n):
    if n==0 or n==1:
        return 1
    else:
        return Fibonaccirecursif(n-1)+Fibonaccirecursif(n-2)
```

**Exercice 1:** Comparer les temps de calculs des deux algorithmes précédents. Commentez… On pourra faire un **import time**, puis on utilisera **a=time.time()** ou tout autre moyen de votre convenance.

Exemple 3 : **Calcul du coefficient binomial C(n,p)**

```
def binomial(n,p):
    if n>p and p>0:
        return binomial(n-1,p-1)+binomial(n-1,p)
    else:
        return 1
```

**Exercice 2:** Comparer les temps de calculs de cet algorithme avec ceux de la version itérative… On utilisera la même bibliothèque que précédemment.

Exemple 4 : **Suite de Syracuse**
La suite de Syracuse d'un entier x est la suite x[n] définie par: x[1]=x et x[n+1]=x[n]/2 si x[n]est pair, et x[n+1]=3x[n]+1 sinon. La suite s'arrête dès que l'on trouve la valeur 1. Ecrire en Python une fonction itérative donnant la suite de Syracuse commençant par l'entier x. Ecrire la version récursive. Aujourd'hui, personne n'a été capable de démontrer que la suite de Syracuse se termine pour toute valeur de x.

**Exercice 3: Autres exemples à connaître :**
**- Recherche d'un extrémum :** Écrire un programme Python permettant de rechercher les extrema des éléments situés dans un tableau.

**- Algorithme d'Euclide :** Cet algorithme permet de calculer le PGCD de deux nombres entiers positifs a et b, avec a>=b. Le principe est le suivant : si b divise a alors le PGCD(a,b)=b, sinon PGCD(a,b)=PGCD(b,r) où r est le reste de la division euclidienne de a par b. Écrire un programme en Python permettant de calculer le PGCD de deux nombres entiers positifs de façon itérative (puis récursive).

**- Décomposition binaire d'un entier :** Écrire un programme Python permettant d'afficher la décomposition binaire d'un nombre entier donné par l'utilisateur.

**- Puissance (exponentiation) rapide :** Écrire un programme Python itératif (puis récursif) permettant de calculer la puissance n d'un nombre x de façon plus rapide que la méthode naïve.

**- Recherche d'un élément dans un tableau trié :** Écrire un programme Python itératif (puis récursif) permettant de rechercher si un élément se situe dans un tableau trié. Modifier le programme pour qu'il renvoie l'indice de la case du tableau où se situe l'élément recherché.

**- Devinons un nombre :** Écrire un programme Python qui permet à un utilisateur de rechercher un entier entre 1 et 100 généré aléatoirement par la machine (**randint(1,100)**). À chaque proposition de l'utilisateur le programme devra afficher si le nombre de l'ordinateur est plus grand, plus petit ou égal au nombre proposé. Le joueur a seulement 7 propositions possibles.

**- Nombres amis :** Un entier naturel est un nombre parfait s'il est égal à la somme de ses diviseurs propres. 6 est parfait car 6=1+2+3. Deux entiers sont amis si chacun d'eux est égal à la somme des diviseurs de l'autre. Écrire un programme Python permettant d'afficher tous les nombres parfaits, et tous les couples de nombres amis.

**Programmation dynamique:** La programmation dynamique consiste à résoudre un problème en le décomposant en sous-problèmes, puis à résoudre les sous-problèmes, des plus petits aux plus grands en stockant les résultats intermédiaires. La programmation dynamique est souvent utilisée lorsqu'une solution récursive se révèle inefficace. Rappelons que la solution récursive utilise des appels récursifs sur des valeurs décroissantes (du haut vers le bas). À contrario, la programmation dynamique va utiliser la récursivité dans le sens inverse (du bas vers le haut) pour éviter de calculer plusieurs fois la même chose. Par exemple, pour le calcul des nombres de Fibonacci, un programme dynamique pourrait être :

```
def Fibodynamique(n):
    t=[0]*(n+1)
    t[0]=1
    t[1]=1
    for i in range(2,n+1):
        t[i]=t[i-1]+t[i-2]
    return t[n]
```

**Exercice 4:** Écrire un programme Python (version dynamique) pour calculer le coefficient binomial C(n,p). Comparez les temps de calculs avec la versions récursive.

**Les questions à se poser :**

- **Est-ce que l'algorithme donne un résultat (terminaison)?**
- **Est-ce que l'algorithme donne le résultat attendu (correction)?**
- **Est-ce que l'algorithme donne le résultat en un temps acceptable (complexité)?**

**Définition:** On appelle **convergent** une quantité qui prend ses valeurs dans un ensemble minoré d'entiers et qui diminue strictement à chaque passage dans une boucle.
**Théorème:** Si une boucle possède un convergent, alors elle se termine

**Définition:** On appelle **invariant de boucle** une propriété qui, si elle est vraie avant l'entrée dans une boucle, reste vraie après chaque passage dans la boucle, et donc est vraie aussi à la sortie de cette boucle.
**Théorème:** La mise en évidence d'un invariant de boucle adapté permet de prouver la correction d'un algorithme

**Exemple de preuve de terminaison et de correction pour 'factorielle':**
**Terminaison :** Dans la version itérative, le convergent est (n+1-i). En effet, pour i allant de 2 à n, la valeur n+1-i décroit de n-1 à 1, ce qui garantit la finitude de la boucle.
Dans la version récursive, le convergent est la valeur x telle que : x est égal à n lors du premier appel, et à chaque nouvel appel récursif x est décrémenté d'une unité. Lorsque x atteint 0 la fonction renvoie 1, donc la récursivité s'arrête.
**Correction :** Mathématiquement, n! = n*(n-1)! Avec 0! =1. Cette définition récursive donne directement la correction de l'algorithme récursif.
Pour la version itérative, un invariant de boucle pourrait être : "p contient factorielle de i".

**Exercices 5: (preuve de terminaison et de correction):** Pour chaque algorithme (version itérative et récursive) de la Partie 0, montrer la terminaison de l'algorithme et la correction en utilisant des convergents et des invariants de boucle.

**Complexité d'un algorithme:** Déterminer la complexité d'un algorithme, c'est évaluer les ressources nécessaires à son exécution en temps et en mémoire. On se focalisera ici sur la complexité en temps. Il s'agit d'évaluer le nombre f(n) d'opérations élémentaires (opérations arithmétiques, comparaisons de données, transferts de données, …) en fonction de la taille n de la donnée en entrée. En pratique, on donnera le comportement de cette quantité lorsque n tend vers l'infini (Log n, n, n log n, $n^2$, …, $k^n$ , …). La notation la plus utilisée est la notation du "grand O".

$$f(n) = O(\alpha_n) \iff \exists B > 0 \mid f(n) \leqslant B\alpha_n.$$

| | $\log n$ | $n$ | $n \log n$ | $n^2$ | $n^3$ | $2^n$ |
|---|---|---|---|---|---|---|
| $10^2$ | 7 ns | 100 ns | 0,7 µs | 10 µs | 1 ms | $4 \cdot 10^{13}$ années |
| $10^3$ | 10 ns | 1 µs | 10 µs | 1 ms | 1 s | $10^{292}$ années |
| $10^4$ | 13 ns | 10 µs | 133 µs | 100 ms | 17 s | |
| $10^5$ | 17 ns | 100 µs | 2 ms | 10 s | 11,6 jours | |
| $10^6$ | 20 ns | 1 ms | 20 ms | 17 mn | 32 années | |

**Exercice 6:** Donner un algorithme pour chaque type de complexité.

## PARTIE 1 - ALGORITHMES DE TRIS CLASSIQUES

**1) Tri par sélection:** Ce tri est parfois appelé naïf. Le principe consiste à rechercher le minimum de la liste et de le placer en début de liste, puis recommencer sur la sous-liste des autres éléments. Par exemple, prenons la liste L=[6,9,1,5,3,3,10,4,8,2]. A l'étape 1, on recherche la valeur minimum de la liste (ici c'est 1 en position 3), puis on échange cette valeur avec la première valeur de la liste (ici, c'est 6). La liste L devient alors L=[1,9,6,5,3,3,10,4,8,2]. On recommence alors ce processus pour la sous-liste [9,6,5,3,3,10,4,8,2]. Le minimum est 2, et on l'échange avec 9. La liste L devient alors [1,2,6,5,3,3,10,4,8,9]. On recommence ainsi jusqu'à ce que la sous-liste à trier soit de taille 1.

**Version itérative:**

```
def triselection(L):
      taille=len(L)
      for i in range(taille-1):
         indicemin=i
         for j in range(i+1,taille):
             if L[indicemin]>L[j]:
                 indicemin=j
         L[indicemin],L[i]=L[i],L[indicemin]
      return L
```

**Version récursive:**

```
def triselectionrecursif(L,i):
    if i<len(L):
         indicemin=i
         for j in range(i+1,len(L)):
             if L[indicemin]>L[j]:
                 indicemin=j
         L[indicemin],L[i]=L[i],L[indicemin]
         triselectionrecursif(L,i+1)
    return L
```

**2) Tri par insertion :** Il s'agit du tri utilisé par le joueur de carte. Le principe consiste à insérer successivement chaque élément L[i] à sa bonne place dans la partie de la liste déjà triée L[0:i]. Par exemple, prenons la liste L=[9,6,1,5,3,3,10,4,8,2]. À l'étape 1, on insère la valeur 6 à sa place dans la liste triée [9]. La liste L devient alors L=[6,9,1,5,3,3,10,4,8,2]. On recommence alors ce processus en insérant la valeur 1 dans la sous-liste [6,9]. On obtient alors L=[1,6,9,5,3,3,10,4,8,2]. On recommence ainsi jusqu'au traitement du dernier élément de la liste.

**Version itérative:**

```
def triinsertion(L):
       taille=len(L)
       for i in range(1,taille):
           val=L[i]
           j=i-1
           while j>=0 and L[j]>val:
               L[j+1]=L[j]
               j=j-1
           L[j+1]=val
       return L
```

**Version récursive:**

```
def insere(L,j):
    if j>0 and L[j]<L[j-1]:
            L[j-1],L[j]=L[j],L[j-1]
            insere(L,j-1)

def triinsertionrecursif(L,j):
    if j<len(L):
        insere(L,j)
        triinsertionrecursif(L,j+1)
    return L
```

**3) Tri à bulles** : Le principe consiste à faire des parcours successifs de la liste en échangeant les valeurs contigües mal classées. Le premier parcours de la liste place le maximum de la liste en dernière position. Par exemple, prenons la liste L=[9,6,1,5,3,3,10,4,8,2], les échanges effectués lors du premier passage sont: [9,6,1,5,3,3,10,4,8,2]  [6,9,1,5,3,3,10,4,8,2]  [6,1,9,5,3,3,10,4,8,2]  [6,1,5,9,3,3,10,4,8,2]  [6,1,5,3,9,3,10,4,8,2]  [6,1,5,3,3,9,10,4,8,2]  [6,1,5,3,3,9,4,10,8,2] ]  [6,1,5,3,3,9,4,8,10,2]  [6,1,5,3,3,9,4,8,2,10]. On recommence ensuite le processus sur la sous-liste obtenue en ne considérant pas ce maximum.

**Version `lourde'**

```
def tribullelourd(L):
        taille=len(L)
        for i in range(taille-1,0,-1):
            for j in range(i):
                if L[j]>L[j+1]:
                    L[j],L[j+1]=L[j+1],L[j]
        return L
```

**Version `optimisée'**

```
def tribulleoptimise(L):
        taille=len(L)
        trie=False
        indice=1
        while not trie:
            trie=True
            for i in range(taille-indice):
                if L[i]>L[i+1]:
                    L[i],L[i+1]=L[i+1],L[i]
                    trie=False
            indice=indice+1
        return L
```

**<u>Exercice 7 :</u>** Écrire un programme Python permettant de réaliser un tri à bulle shaker. Les passages sont effectués alternativement de gauche à droite puis de droite à gauche.

```
def tribulleshaker(L):
        taille=len(L)
        sens=1
        deb=0
        fin=taille-1
        trie=False
        i=deb
        while not trie:
            trie=True
            while (sens==1 and i<fin) or (sens==0 and i>=deb):
                if L[i]>L[i+1]:
                    L[i],L[i+1]=L[i+1],L[i]
                    trie=False
                if sens==1:
                    i=i+1
                else:
                    i=i-1
            if sens==1:
                fin=fin-1
                i=fin-1
            else:
                deb=deb+1
                i=deb
            sens=1-sens
        return L
```

**4) Tri fusion :** Il s'agit d'un exemple d'algorithme de type `diviser pour régner'. Le principe est de trier deux moitiés de la liste puis de fusionner les deux sous-listes triées. Il s'agit d'un tri récursif. Par exemple, si L=[4,1,8,3,2,9], alors le tri des deux demi listes donne [1,4,8] et [2,3,9], et la fusion de ces listes termine le tri.

```
def fusion(K,L):
    c=[]
    tailleK=len(K)
    tailleL=len(L)
    i=0
    j=0
    while i<tailleK and j<tailleL:
        if K[i]<L[j]:
            c.append(K[i])
            i=i+1
        else:
            c.append(L[j])
            j=j+1
    if i==tailleK:
        for k in range(j,tailleL):
                c.append(L[k])
    else:
        for k in range(i,tailleK):
                c.append(K[k])
    return c
```

```
def trifusion(L):
    taille=len(L)
    if taille<=1:
        return L
    else:
        m=taille//2
        return fusion(trifusion(L[0:m]),trifusion(L[m:taille]))
```

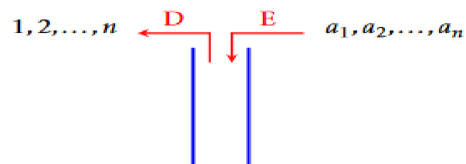
**5) Tri Quicksort :** Il s'agit d'un exemple d'algorithme de type `diviser pour régner'. Le principe est de choisir un élément, appelé pivot, et de séparer la liste en deux sous-listes comprenant respectivement les éléments inférieurs et les éléments supérieurs au pivot. On utilise ensuite la récursivité sur les deux sous-listes. Le pivot sera le premier élément de la liste. Par exemple, si L=[6,5,1,9,2,8], le pivot est 6 et les deux sous-listes sont [5,1,2] et [9,8].

```
def partition(L,deb,fin):
    pivot = L[deb]
    i = deb+1
    j = fin
    partage = False
    while not partage:
        while i <= j and L[i] <= pivot:
            i = i + 1
        while L[j] >= pivot and j >= i:
            j = j -1
        if j < i:
            partage = True
        else:
            L[i],L[j]=L[j],L[i]
    L[deb],L[j]=L[j],L[deb]
    return j

def quicksort(L,deb,fin):
    if deb<fin:
        indicepivot = partition(L,deb,fin)
        quicksort(L,deb,indicepivot-1)
        quicksort(L,indicepivot+1,fin)
    return L
```

**6) Recherche dichotomique :** Il s'agit d'un exemple d'algorithme de type `diviser pour régner'. Le principe de recherche d'un élément dans un tableau trié consiste à comparer l'élément avec la valeur située au milieu du tableau et d'en déduire la partie du tableau où il apparaît.

```
def dicho(elmt, L):
    if len(L)==1 :
        return 0
    m = len(L)//2
    if L[m] == elmt :
        return m
    elif L[m] > elmt :
        return dicho(elmt, L[:m])
    else :
        return m + dicho(elmt, L[m:])
```

**7) Tri par pile :** On dit qu'une liste L (représentant une permutation) est triable par pile lorsqu'il est possible, à l'aide d'une pile initialement vide, d'ordonner ses éléments en utilisant les opérations suivantes :
- empiler l'élément courant de la liste d'entrée (E)
- dépiler le sommet de la pile dans la liste de sortie (D).



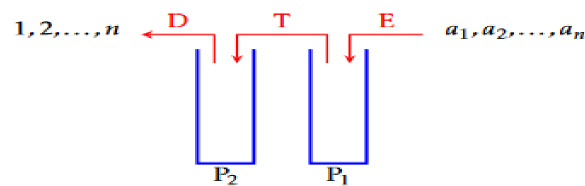

Par exemple, la liste L=[5,4,1,3,2] peut être triée par la succession d'opérations EEEDEEDDDD.
**Exercice 8 :** Parmi les listes suivantes, lesquelles peuvent être triées par une pile ?
[3,4,1,2], [4,3,1,2,6,5], [4,5,3,7,2,1,6]. On pourra effectuer une version procédurale et une version utilisant une class pile.

```python
def estvide(p):
    return len(p)==0

def sommet(p):
    return p[len(p)-1]

def empile(p, elmt):
    print("empile ", elmt)
    p.append(elmt)

def depile(p):
    print("dépile ", p[len(p)-1])
    if not estvide(p):
       p.pop()

def affiche(p, mess):
    print("pile : ", mess,end="   ")
    for i in range(len(p)):
        print(p[i], end="  ")
    print("")

def tripile(a):
    n=len(a)
    p=[]
    sortie=[]
    k=0
    s=0
    for i in range(2*n):
        if not estvide(p) and sommet(p)==s+1:
            s=sommet(p)
            depile(p)
            sortie.append(s)
        elif k<n:
                empile(p, a[k])
                k=k+1
        else:
            return None
    return sortie
```

```python
class pile:
    def __init__(self):
        self.pile=[]
    def sommet(self):
        return self.pile[-1]
    def empile(self,elmt):
        return self.pile.append(elmt)
    def depile(self):
        return self.pile.pop()
    def estvide(self):
        if self.pile == []:
            return True
        else:
            return False
```

```python
def tripile(a):
    n =len(a)
    p = pile()
    sortie=[]
    k = 0
    s = 0
    for _ in range(2*n):
        if not p.estvide() and p.sommet() == s + 1:
            s = p.depile()
            sortie.append(s)
        else:
            if k < n:
             p.empile(a[k])
             k = k + 1
            else:
             return "pas triable"
    return sortie
```

On peut également trier une permutation en utilisant deux piles en séries.




**Exercice 9 :** Montrer que la liste [2,4,3,1] est triable par deux piles au moins de deux manières différentes. De combien de manières la liste [3,2,1] peut-elle être triée ? Plus généralement, montrer que la liste [n,n-1,n-2, … ,2,1] est triable de $2^{\{n-1\}}$ façons différentes. Montrer que la liste [2,4,3,5,7,6,1] n'est pas triable par deux piles. Écrire un programme python permettant de trier (avec deux piles en séries) une permutation.

## PARTIE 2 - AUTRES ALGORITHMES A CONNAITRE

**On appelle « algorithme glouton » tout algorithme qui suit une heuristique permettant de faire *'le meilleur choix'* à chaque étape.** Cette heuristique ne conduit pas toujours à la solution souhaitée, mais doit donner des résultats acceptables dans un bon nombre de cas.
Un algorithme glouton pour le tri à deux piles peut être : effectuer le mouvement légal le plus à gauche à chaque étape : (i) on réalise l'opération D à chaque fois que l'on peut, sinon on réalise l'opération T si elle respecte la contrainte de croissance de la pile P2; sinon on réalise l'opération E. Trouver une liste de longueur 5 pour que cet algorithme échoue, et vérifier qu'elle peut néanmoins être triée par deux piles.

**8) Le rendu de monnaie** : Le problème du rendu de monnaie va nous fournir un second exemple simple pour comprendre un algorithme glouton *: comment décomposer une somme d'argent en utilisant un nombre minimal de pièces à choisir parmi un nombre limité de valeurs possibles ?*
Considérons par exemple les unités monétaires européennes c=[1,2,5,10,20,50,100,200,500] (en euros). On suppose disposer d'un nombre illimité de pièces et billets de chaque type, et on souhaite connaître la décomposition minimale en nombre de pièces et billets d'une certaine somme, par exemple n=493 euros. Une stratégie gloutonne consiste à rendre la pièce ou le billet v de valeur maximale parmi celles de valeurs inférieures ou égales à n (ici c'est v=200), puis réitérer le procédé avec n-v. Sur cet exemple, on obtient 493=200+200+50+20+20+2+1.

**Exercice 10 :** Écrire une fonction Python qui prend en paramètres une somme n et un système de monnaie c supposé trié par ordre croissant et qui retourne la décomposition obtenue suivant la stratégie gloutonne décrite ci-dessus sous la forme d'une liste de valeurs à utiliser. Par exemple :

```
glouton(493,[1,2,5,10,20,50,100,200,500])
[200, 200, 50, 20, 20, 2, 1]
```

Cette stratégie fournit toujours une solution optimale pour ce système de monnaie. Prenons maintenant le système de monnaie britannique d'avant 1971. On a c=[1,3,6,12,24,30]. Trouver un nombre n de pennies sur lequel l'algorithme glouton n'est pas optimal.

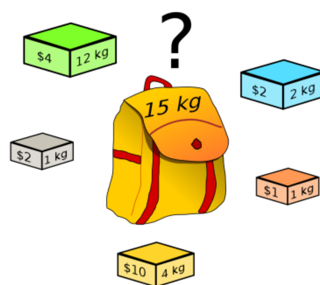
**9) Le problème du sac à dos** - *Étant donné plusieurs objets possédant chacun un poids et une valeur et étant donné un poids maximum pour le sac, quels objets faut-il mettre dans le sac de manière à maximiser la valeur totale sans dépasser le poids maximal autorisé pour le sac ?*

Il existe deux grandes catégories de méthodes de résolution de problèmes d'optimisation combinatoire : les méthodes exactes et les méthodes approchées. Les méthodes exactes permettent d'obtenir la solution optimale à chaque fois, mais le temps de calcul peut être long si le problème est compliqué à résoudre. Les méthodes approchées, encore appelées heuristiques, permettent d'obtenir rapidement une solution approchée, donc pas nécessairement optimale. Supposons que le sac à dos peut contenir au maximum 30 kg, et que nous avons 4 objets O1, O2, O3, O4 de poids respectifs 13, 12, 8, 10, et de valeurs respectives 7, 4, 3 et 3.

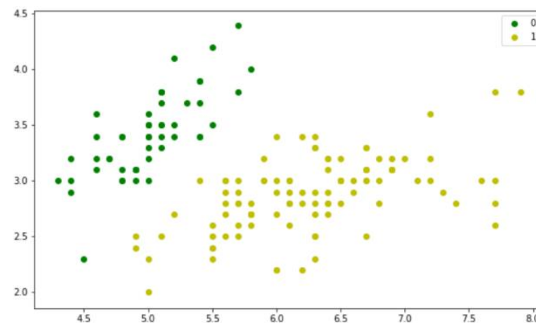
L'algorithme est :

- Calculer v_i/p_i pour les 4 objets (0.54, 0.33, 0.37 et 0.30)
- Trier tous les objets par ordre décroissant de ce rapport
- Sélectionner les objets un à un dans l'ordre du tri et ajouter l'objet sélectionné dans le sac si le poids maximal reste respecté.

**Exercice 11 :** Quels seront les objets contenus dans le sac à dos à la fin de l'algorithme ? Écrire le programme en Python correspondant à cet algorithme.



**10) Algorithmes d'apprentissage : Les k plus proches voisins** – Le principe est `*dis mois qui sont tes amis, je te dirai qui tu es'*. On dispose d'une base de données d'apprentissage constituée de N couples entrée-sortie, et on veut estimer le type d'une nouvelle entrée en recherchant ses k plus proches voisins dans la base.



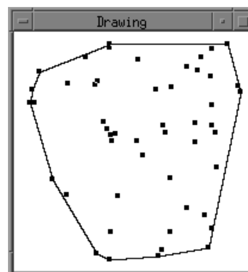

```
from math import sqrt, exp
from random import shuffle

import csv
csvfile = open ("iris1.data", "r")
lines = csv.reader (csvfile)
dataset = list (lines)

for x in range (len (dataset)):
  for y in range (4):
    dataset[x][y] = float (dataset[x][y])

def distance (a, b):
    somme = 0
    for i in range (len (a)):
        somme =somme + (a [i] - b [i]) * (a [i] - b [i])
    return (sqrt (somme))
```

```python
def kplusprochesvoisins (x, k):
  ldist = []
  for i in range (len (dataset)):
    ldist.append (distance (x, dataset [i] [0:4]))
  Kppv = []
  for i in range (k):
    p = float ("inf")
    for j in range (len (dataset)):
      if ldist [j] != 0 and ldist [j] < p and j not in Kppv:
        p = ldist [j]
        indice = j
    Kppv. append (indice)
  return (Kppv)

def predire (l):
  choix = ['Iris-setosa', 'Iris-versicolor', 'Iris-virginica']
  effectifs = [0, 0, 0]
  for x in l :
    for i in range (3):
      if dataset [x] [4] == choix [i]:
        effectifs [i] = effectifs[i] + 1
  maxi = effectifs [0]
  indicemax = 0
  for i in range (1,3):
    if effectifs [i] > maxi:
      maxi = effectifs [i]
      indicemax = i
  return choix[indicemax]
```

```python
print(kplusprochesvoisins ([17,4,39], 3))
print (predire (kplusprochesvoisins ([17,4,39], 3)))
```

**11) Algorithme de calcul de l'enveloppe convexe (algorithme de Jarvis) :** On part d'un point arbitraire dont on sait qu'il est sur l'enveloppe. On prendra, par exemple, le point le plus petit pour l'ordre lexicographique selon l'ordonnée *y* puis l'abscisse *x*. Ensuite, on itère la recherche des points successifs qui vont constituer l'enveloppe. Étant donné $P_i$, un point de l'enveloppe d'indice i dans le nuage, le point suivant sur l'enveloppe est le plus petit pour la relation d'ordre $<_i$ définie par :
$P_j <_i P_k \iff angle(P_i P_j , P_i P_k) > 0$

**Exercice 12 :** Écrire un algorithme Python permettant de calculer l'enveloppe convexe d'un nuage de N points générés aléatoirement dans le plan et d'afficher les points et l'enveloppe convexe. Utiliser la bibliothèque matplotlib pour faire une représentation graphique de l'enveloppe convexe et du nuage de points.



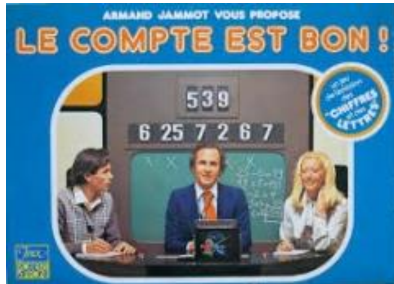

**12) Tour de Hanoï:** Le jeu des tours de Hanoï est un casse-tête composé de trois tours et une pile de disques rangés du plus grand au plus petit comme sur la photo ci-dessous.

Le but est de déplacer la pile de disques sur la tour de droite en ne déplaçant à chaque fois qu'un seul disque et un disque ne peut pas être posé sur un disque plus petit. On appelle les tours A, B et C de gauche à droite et ***n*** est le nombre de disques présents au départ dans la tour A. Pour déplacer tous les disques de la tour A vers la tour C, on peut raisonner comme suit : on déplace n-1 disques de A vers la tour B; on déplace le dernier disque de A vers C, puis on déplace les n-1 disques de B vers C.

Créer une fonction ***hanoi*** qui prend 4 paramètres : ***hanoi(n,debut,inter,fin)*** où ***n*** est le nombre de disques à déplacer, ***debut*** est la tour de départ de nos ***n*** disques, ***inter*** est la tour intermédiaire que l'on peut utiliser pour déplacer et ***fin*** est la tour où doivent se trouver les ***n*** disques au final. Ainsi, au début on va lancer ***hanoi(n,"A","B","C")*** mais quand on va vouloir déplacer les ***n-1*** disques de A vers B, on écrira ***hanoi(n-1,"A","C","B")***.

**Exercice 13 :** Écrire cette fonction recursive ***hanoi(n,debut,inter,fin)*** de manière à afficher (avec print) à chaque étape le déplacement à effectuer sous la forme "A----- > B" pour un déplacement de la tour "A" vers la tour "B" par exemple.

**13) Le compte est bon :** En choisissant 6 nombres (éventuellement égaux) dans l'ensemble {1,2,3,4,5,6,7,8,9,10,25,50,75,100} et en leur appliquant les quatre opérations élémentaires (+,-, *,/), l'objectif est d'atteindre le résultat demandé (ceci est possible dans 94% des cas). Tous les calculs intermédiaires doivent être entiers. Chacun des nombres peuvent être utilisés qu'une seule fois. Si le résultat demandé ne peut pas être atteint, il faut l'approcher au plus prés.




**<u>Exercice 14:</u>** Écrire en python l'algorithme suivant (backtracking)
0) on trie par ordre décroissant les nombres dans un tableau T
1) on choisit une paire de nombres
2) on fait une opération sur cette paire (soit R le nombre obtenu)
3) si on a trouvé le nombre voulu, le programme s'arrête
4) sinon, on remplace les deux nombres par R (le tableau contient alors un nombre de moins)
5) on trie T, et on appelle récursivement la fonction sur T (de taille inférieure) £
6) si cela conduit à un échec, on replace les deux nombres dans le tableau, et on recommence avec deux autres

**14) Le sudoku:** Tout le monde connaît les règles du sudoku!!!! Je ne les rappelle pas ici… Voici un algorithme efficace pour résoudre un sudoku.

**<u>Exercice 15 :</u>** Écrire en python l'algorithme suivant (backtracking)
1. S'il n'y a pas de case vide, alors c'est gagné.
2. Sinon, choisir la première case vide.
3. Regarder la liste des éléments possibles pour cette case.
4. Si aucun élément ne convient, c'est perdu
5. Sinon, pour chaque élément possible faire
6. Affecter la case avec cet élément.
7. Relancer la fonction récursivement
8. Si on a gagné, le programme se termine
9. sinon changer d'élément (aller en 5)
10. Remise à zéro de la case

| 2 | 3 |  |  |  | 5 |  |  |  |
|---|---|---|---|---|---|---|---|---|
|  |  |  | 3 | 6 |  | 4 |  |  |
|  |  | 5 |  |  |  |  |  |  |
| 6 |  |  |  |  | 8 |  |  |  |
|  |  |  | 1 |  |  |  | 3 |  |
|  | 1 |  | 4 |  | 3 | 9 |  |  |
| 1 |  |  |  | 8 | 2 |  |  |  |
|  | 2 | 4 |  |  |  | 7 | 9 |  |
|  |  |  |  | 9 |  | 1 | 8 |  |

```python
class Sudoku:
    def __init__(self):
        self.field = [ [ 0 for _ in range(9) ] for _ in range(9) ]
    def empty_cells(self):
       return [(r,c) for r, row in enumerate(self.field)
                     for c, val in enumerate(row) if val == 0]

    def available(self, x, y):
        row = [ v for v in self.field[x] ]
        col = [ r[y] for r in self.field ]
        squ = [ self.field[X][Y]

                  for X in range(x//3 * 3, x//3 * 3 + 3)
                  for Y in range(y//3 * 3, y//3 * 3 + 3) ]

        return set(range(1,10)) - set(row + col + squ)

    def print_game(self):
        for i in range(0, 9):
            print(self.field[i])
        print

    def play_game(self):
       e = self.empty_cells()
       if len(e) == 0:
           return self

       x, y = e[0]
       for v in self.available(x, y):
           self.field[x][y] = v

           if self.play_game() != None:
               return self


       self.field[x][y] = 0
       return None

if __name__ == "__main__":
    g = Sudoku()

    g.field = [
      [7,0,0,  0,8,0,  0,0,2],
      [0,5,0,  0,0,7,  0,0,3],
      [0,0,4,  0,0,9,  0,0,6],


      [0,0,2,  0,0,5,  0,3,0],
      [5,0,0,  0,0,0,  0,0,4],
      [0,8,0,  3,0,0,  1,0,0],


      [1,0,0,  6,0,0,  2,0,0],
      [3,0,0,  7,0,0,  0,9,0],
      [4,0,0,  0,9,0,  0,0,5],
    ]

    g.print_game()
    print('***************************')
    g_sol = g.play_game()
    if g_sol != None:
        g_sol.print_game()
    else:
        print('No solution')
```

## PARTIE 3 - ALLER PLUS LOIN

Il est inconcevable de conclure sans parler d'algorithme génétique et de programmation fonctionnelle.

**Algorithme génétique :**
Les algorithmes génétiques utilisent la théorie de Darwin sur l'évolution des espèces. Elle repose sur trois principes : le principe de variation, le principe d'adaptation et le principe d'hérédité. L'idée principale des heuristiques est d'explorer l'espace des solutions en essayant de converger vers la meilleure solution. Cependant, il est important d'éviter une convergence prématurée de l'algorithme vers un extremum, ou optimum, local. Un extremum local est la meilleure solution dans une zone restreinte, en opposition à l'extremum global, qui est la meilleure solution dans l'ensemble.
Voir un exemple sur le site :
**http://igm.univ-mlv.fr/~dr/XPOSE2013/tleroux_genetic_algorithm/fonctionnement.html**

**Programmation fonctionnelle :**
La programmation fonctionnelle s'affranchit de façon radicale des effets secondaires (ou effets de bord) en interdisant toute opération d'affectation. Le paradigme fonctionnel n'utilise pas de machine à états pour décrire un programme, mais un emboîtement de fonctions qui agissent comme des « boîtes noires » que l'on peut imbriquer les unes dans les autres. Chaque boîte possédant plusieurs paramètres en entrée mais une seule sortie, elle ne peut sortir qu'une seule valeur possible pour chaque n-uplet de valeurs présentées en entrée.

## PARTIE 4 - EVALUATION PREMIERE PARTIE BLOC 2

Réaliser un document pédagogique basé sur le jeu des tours de Hanoï.
- Ecrire le programme en Python en version récursive - On pourra éventuellement utiliser une bibliothèque graphique pour illustrer les mouvements.
- Vérifier la terminaison et la correction du programme
- Donner la complexité
- Question bonus: Ecrire en Python une version itérative

Ecrire un support de cours expliquant les différentes notions utilisées.