# ALGORITHMES: PREMIERS PAS - CORRECTION

## 1. MESURE D´UNE MOYENNE:

```
1 def moyenne(liste):
2     ''' prend une liste en paramètre et
3     retourne la valeur moyenne des
4     éléments de cette liste '''
5     somme=liste[0]
6     n=len(liste)
7     for i in range(1,n):
8         somme+=liste[i]
9     return somme/n
10
11 notes=[10,13,8,14,11,16]
12 print(moyenne(notes))
```

**Détermination du coût:**
On détermine le nombre d´opérations élémentaires. On remarque que pour cet algorithme, il n´y a ni meilleurs ni pire des cas puisque toutes les opérations sont effectuées.

→ Ligne 5: 1 affectation
→ Ligne 6: 1 affectation
→ Ligne 7: n-1 affectations (pour i)
→ Ligne 8: n-1 affectations (de la variable `somme`)
→ Ligne 9: 1 calcul effectué (`somme/n`) et 1 valeur retournée

Bilan: **`C(n)=1+1+n-1+n-1+1+1=2n+2`**

Le coût de l´algorithme est proportionnel à la taille n de la liste, sa complexité est donc *linéaire*, c´est à dire en `O(n)`.

L´appel de la fonction pour la liste `notes` retourne la valeur 12.0

## 2. EXPERIMENTONS UN TEMPS D´EXECUTION

a)

```
def rdmTab(n):
    ''' prend la taille de la liste à créer
    en paramètre et retourne un liste de valeurs
    aléatoires comprises entre 0 et 1000 '''
    rdmList=[]
    for i in range(n):
        rdmList.append(randint(0,1000))
    print(rdmList)
    return(rdmList)
```

*On n´oublie pas d´importer le module `random`.*

Exemple: `RechercheMin(rdmTab(10))` pour chercher la valeur minimum contenu dans un tableau de 10 valeurs aléatoires.

b) Calcul du temps d´exécution:

```
def afficheTemps(n):
    ''' retourne le temps d´exécution de la recherche du
    minimum dans un tableau aléatoire de n valeurs '''
    debut=time()
    RechercheMin(rdmTab(n))
    fin=time()
    return fin-debut
```

*On n´oublie pas d´importer le module `time`.*

c) Dans la console, on exécute plusieurs fois l´appel `afficheTemps(1000).` On constate que le temps mesuré varie légèrement pour chaque appel. En effet, les listes créées à chacun de ces appels offrent des configurations plus ou moins optimales (voir cours sur la recherche du minimum).

*d)* Modification de la fonction de mesure du temps, testez-la dans la console:

```
def afficheTemps(n):
    ''' retourne une liste contenant le temps d´exécution
    de la recherche du minimum
    dans 100 tableaux aléatoires de n valeurs '''
    liste=[]
    for i in range(100):
        debut=time()
        RechercheMin(rdmTab(n))
        fin=time()
        liste.append(fin-debut)
    return liste
```

On utilise la fonction **`moyenne(liste)`** de l´exercice précédent. Si on fait plusieurs fois cet appel dans la console: `moyenne(afficheTemps(1000))`, on remarque que le temps mesuré est beaucoup plus stable.

Doublez maintenant la taille de la liste: `moyenne(afficheTemps(2000))`, le temps mesuré est-il bien deux fois plus grand?

## 3. CALCUL D´UNE FACTORIELLE

```
1 def factorielle(n):
2     p=1
3     i=n
4     while i>1:
5         p=p*i
6         i=i-1
7     return p
```

a) Observons attentivement la méthode calcul:

$$n! = \underbrace{1\times 2\times 3\times \ldots\times (n-1)}_{(n-1)!}\times n$$

La suite de multiplications soulignée correspond à la factorielle de (n-1), on en déduit que $n! = (n-1)!\times n$

b) Observez bien la boucle `while`, elle réalise n tours de boucles (de i=n à i=1 compris) → ligne 2: 1 affectation → ligne 3: 1 affectation → ligne 5: n affectations → ligne 6: n affectations → ligne 7: 1 valeur retournée.

Bilan: **`C(n)=1+1+n+n+1=2n+3`**

Le coût de l´algorithme est proportionnel à n, sa complexité est donc *linéaire*, c´est à dire en `O(n)`.

c) On propose pour cet algorithme la propriété invariante suivante: " **A la fin de l´exécution de la k$^e$ boucle, nous avons la relation:** $p_k \times i_k! = n!$ "

INITIALISATION: La propriété est-elle vraie avant d´entrer dans la boucle (donc k=0)? A cet instant k=0 et $p_0$=1 et $i_0$=n (lignes 2 et 3). On peut écrire: $p_0 \times i_0! = 1 \times n!$.

**La propriété est bien vérifiée pour k=0.**

CONSERVATION: La propriété $p_k \times i_k! = n!$ est considérée comme vraie au k$^{ème}$ tour de boucle: $p_k \times i_k! = n!$. Est-elle toujours vraie au tour boucle suivant (donc pour k+1)**?**
Cherchons à calculer $p_{k+1} \times i_{k+1}!$:
Au (k+1)$^{ème}$ tour de boucle, les calculs des lignes 5 et 6 donnent:
→ $p_{k+1} = p_k \times i_k$ et $i_{k+1} = i_k - 1$ donc: $p_{k+1} \times i_{k+1}! = p_k \times i_k \times (i_k - 1)!$

Or la question a) nous a appris que: $n! = (n-1)! \times n$

On peut donc affirmer que $i_k \times (i_k - 1)! = i_k!$. Remplaçons dans l´expression précédente:

$p_{k+1} \times i_{k+1}! = p_k \times i_k!$ or, par hypothèse, $p_k \times i_k! = n!$ donc: $p_{k+1} \times i_{k+1}! = n!$

**La propriété est bien vérifiée au rang k+1.**

TERMINAISON: Lorsque la boucle se termine elle a effectué n tours et $i_n$=1. La propriété invariante à cet instant s´écrit: $p_n \times i_n! = n!$. Comme $i_n$=1, $p_n = n!$

**On en conclut que l´algorithme retourne bien la factorielle de l´entier n.**

## 4. LES HARICOTS DE GRIES

a)

| | Au début | Si on tire deux blancs | Si on tire deux noirs | Si on tire un blanc et un noir |
|---|---|---|---|---|
| Nombre haricots noirs | **N** | N ← **N+1** | N ← **N-1** | N ← **N-1** |
| Nombre haricots blancs | B | B ← **B-2** | B ← **B** | B ← **B** |
| Nombre total dans la boite | N+B | **N+B-1** | **N+B-1** | **N+B-1** |

b) Montrez que le jeu se termine toujours.

On observant la dernière ligne du tableau, on remarque quel que soit le tirage il y a **décroissance stricte de un haricot** à chaque tour de boucle. Il y aura forcément un moment où il n´en restera plus qu´un. Cette décroissance est appelée *variant de boucle.*

c) Cherchez la *propriété qui reste invariante* tout au long du jeu.

L´étude de la deuxième ligne du tableau nous permet de proposer cette propriété: **la parité du nombre de haricot** est une propriété invariante tout au long du jeu.

→ Si on a au départ un nombre pair de haricots blancs, il y aura toujours un nombre pair dans la boite.
→ Si on a au départ un nombre impair de haricots blancs, il y aura toujours un nombre impair dans la boite.

d) A l'aide de cette propriété, pouvez-vous dire la *couleur du dernier haricot* dans la boite?

La propriété nous permet de conclure immédiatement que si le nombre de haricots blancs est impair au départ, **le dernier sera blanc** et le nombre de haricots blancs est pair au départ, **le dernier sera noir**.

Les deux situations sont équiprobables.

Pour information, voici une simulation possible du jeu en Python:

```
def recherche(liste,couleur):
    for i in range(len(liste)):
        if liste[i]==couleur:
            return i
    return None

def haricots():
    n= []
    for i in range(randint(5,10)):
        n.append('n')
    b= []
    for i in range(randint(5,10)):
        n.append('b')
    boite=n+b
    shuffle(boite)
    print(boite)
    while len(boite)>1:
        tirage=sample(boite, 2)
        if tirage[0]==tirage[1]:
            if tirage[0]=='b':
                boite.append('n')
                del boite[recherche(boite,'b')]
                del boite[recherche(boite,'b')]
            elif tirage[0]=='n':
                del boite[recherche(boite,'n')]
        else: del boite[recherche(boite,'n')]
    return boite
```