1ère NSI — Exercices d'Entraînement

Algorithmique

(et un peu de traitement de données en table / dictionnaires)

Lycée Fustel de Coulanges, Massy

Marc Biver, février 2024, *v0.3*

Version incluant les réponses aux exercices.

Exercice 1: Docstring & Tests

Soit la fonction de calcul de puissance suivante (dont on a étudié l'algorithme en cours) :

```
def Puissance(x, n):
    r = 1
    for i in range(n):
        r = r * x
    return r
```

a. Rédiger la docstring d'une telle fonction ;

b. Déterminer un jeu de tests pour cette fonction – en d'autres termes, compléter le tableau suivant qui liste les couples (x, n) qu'il conviendra d'utiliser pour convenablement tester cette fonction.

x	n	Résultat attendu
2	2	4
.....	.....	.....

```
def Puissance(x, n):
    '''
    Fonction qui prend en entrée deux entiers naturels et renvoie le
    ↪ premier
    élevé à la puissance du second.
    '''
    r = 1
    for i in range(n):
        r = r * x
    return r
```

Il n'y a pour le jeu de tests pas "une" bonne réponse, mais ce qu'on va chercher à faire c'est effectuer au moins deux tests de valeurs "normales" dont un avec une grande valeur (en plus, donc, du (2,2) de l'énoncé), ainsi que des tests avec les valeurs aux limites – en l'occurrence 0 et 1 – associées à des valeurs choisies au hasard. On pourra donc proposer quelque chose comme :

x	***n***	***Résultat attendu***
2	*2*	*4*
27	*7*	*10.460.353.203*
21	*0*	*1 (tout nombre élevé à la puissance 0 vaut 1)*
35	*1*	*35*
1	*12*	*1*
0	*99*	*0*
0	*0*	*1 (tout nombre élevé à la puissance 0 vaut 1 – même 0)*

Exercice 2: Variant & invariant de boucle

Soit la fonction suivante :

```python
def SommeEltLst(liste):
    '''
    Fonction qui prend en entrée une liste de nombres et qui renvoie la
    somme des nombres qui la constituent.
    '''
    res = 0
    for i in range(len(liste)):
        res += liste[i]

    return res
```

a. Donner un variant de boucle pour cette fonction et montrer qu'elle se termine systématiquement ;

> **Rappel:**
>
> On rappelle qu'un variant de boucle est une quantité entière positive qui décroît strictement à mesure qu'on passe dans la boucle, ce qui permet de justifier que la boucle, tôt ou tard, se terminera.

b. Donner un invariant de boucle pour cette fonction et montrer qu'elle est correcte.

> **Rappel:**
>
> On rappelle qu'un invariant de boucle est une quantité ou une propriété qui est vraie avant et après chaque itération de la boucle – et en particulier *avant* que l'on rentre dans la boucle, et *après* sa dernière itération. Il permet de justifier que le résultat voulu sera atteint. On procède pour cette technique en quatre étapes :
>
> (a) On choisit l'invariant :
>
> — Comprendre clairement le but de la boucle – qu'est-elle censée accomplir ? Quel est le résultat attendu ?
>
> — Partir "de la fin", c'est-à-dire du résultat attendu et identifier quelle quantité est "construite" au fur et à mesure des itérations de la boucle pour constituer ce résultat.
>
> — Ceci devrait vous mettre sur la voie de votre invariant – une propriété (somme d'éléments déjà traités, ordre d'éléments dans une liste...) qui ne change pas malgré les itérations de la boucle.
>
> (b) On montre que l'invariant est vérifié avant la boucle (initialisation) ;
>
> (c) On montre que si l'invariant est vérifié *avant* un passage dans la boucle, alors il est préservé *après* le passage dans la boucle ;
>
> (d) On peut conclure sur la valeur finale à la sortie de la boucle.

a. *Variant de boucle – on est dans le cas ultra-classique du parcours d'une liste par indice, le variant est donc la quantité* $len(liste) - i - 1$ *: il est en effet positif au départ (au pire nul si la liste est vide), et il décroît strictement*

à chaque itération de la boucle puisque len(liste) reste constant et que i est incrémenté de 1 à chaque fois. Il va donc atteindre 0, ce qui terminera la boucle.

b. *Décomposons la démarche :*

   (a) *Choix de l'invariant : le but de la boucle est le calcul de la somme des éléments de la liste, et c'est la variable* **`res`** *qui est construite pour atteindre ce résultat : à chaque étape elle contient la somme des éléments d'indice de 0 à* **`i`** *de la liste. On peut donc dire que l'invariant de boucle est : "la variable* **`res`** *contient la somme des éléments d'indice de 0 à* **`i`** *de la liste" ;*

   (b) *Avant la première itération de la boucle,* **`i`** *vaut 0 et* **`res`** *aussi – donc on peut dire que l'invariant est vérifié ;*

   (c) *Si res contient les i premiers éléments de la liste et que l'on passe dans la boucle une fois supplémentaire,* **`i`** *et* **`res`** *deviennent respectivement : $i' = i + 1$ et $res' = res + liste[i'] = res + liste[i + 1]$ – et donc l'invariant est bien vérifié à la fin de l'itération de la boucle.*

   (d) *A la fin de la boucle i vaut $(n - 1)$ donc l'invariant s'exprime "la variable* **`res`** *contient la somme des éléments d'indice de 0 à* **`(n-1)`** *de la liste" – soit tous les éléments de la liste, et donc la fonction est correcte.*

Exercice 3: Variant & invariant de boucle — suite...

Soit la fonction suivante :

```
1  def RechercheDiviseur(nb):
2      '''
3      Fonction qui prend en entrée un entier strictement supérieur à 1 nb
4      et renvoie son plus petit diviseur autre que 1.
5      Si elle n'en trouve pas (que le nombre est donc premier), elle
6      renvoie 0.
7      '''
8      div = 0
9      i = 2
10     while i < nb:
11         # On vérifie si nb est un multiple de i
12         if nb % i == 0:
13             # Dès qu'on en a trouvé un on le renvoie
14             div = i
15             return div
16         i += 1
17     # On atteindra ce point si on n'a pas trouvé de diviseur et on
       ↪  renverra donc la valeur initiale de div, soit 0.
18     return div
```

a. Donner un variant de boucle pour cette fonction et montrer qu'elle se termine systématiquement ;

b. Donner un invariant de boucle pour cette fonction et montrer qu'elle est correcte.

> **Indice:**
>
> On rappelle qu'un invariant de boucle n'est pas nécessairement une formule mathématique – il peut aussi être une propriété que l'on pourrait exprimer par une phrase du genre (où bien entendu il faut remplir les trous) : "Quel que soit l'entier k tel que 2 ≤ k ≤ ___, k n'est pas___".

c. (question bonus 1) Cette boucle "`while`" semble assez artificielle – on dirait presque qu'elle n'est là que pour vous forcer à travailler sur la notion de variant de boucle... Quelle est la boucle "`for`" équivalente qu'on utiliserait plus logiquement dans un tel cas ?

d. (question bonus 2) Cette fonction n'est pas franchement optimale... Comment remplacer la condition "`while i < nb:`" pour lui faire faire nettement moins de tests tout en gardant le bon résultat ?

a. *Variant de boucle – situation assez simple : la quantité $nb - i$ débute à la valeur $nb - 2$ qui est nécessairement positive puisqu'il est spécifié que nb est strictement supérieur à 1 ; cette quantité décroît strictement à chaque itération de la boucle puisque nb reste constant et que i est incrémenté de 1 à chaque fois. Il va donc atteindre 0, ce qui terminera la boucle.*

b. *Décomposons la démarche :*

   (a) *Choix de l'invariant : le but de la boucle est de trouver le diviseur le plus petit diviseur possible de nb, donc à chaque étape de la boucle, aucun nombre plus petit que celui que l'on est en train de considérer n'est diviseur de nb. On peut donc exprimer l'invariant : "quel que soit l'entier k tel que 2 ≤ k ≤ (i-1), k n'est pas un diviseur de nb" ;*

   (b) *Avant la première itération de la boucle, `i` vaut 2, donc il n'existe aucun entier k tel que 2 ≤ k ≤ (i-1) – donc on peut dire que l'invariant est vérifié ;*

   (c) *Si l'invariant est vérifié pour une valeur de i, alors pour la valeur suivante $i' = i + 1$, deux cas sont possibles :*

   - *Soit i est un diviseur de nb et la fonction le renvoie et donc se termine (donc la question de l'invariant ne se pose plus) ;*
   - *Soit i n'est pas un diviseur de nb – et la boucle se poursuit et l'invariant est vérifié : "quel que soit l'entier k tel que 2 ≤ k ≤ (i'-1), k n'est pas un diviseur de nb" (puisque $i' - 1 = i$).*

   (d) *A la fin de la boucle, dans l'hypothèse où on n'est pas sorti de la fonction, i vaut nb, donc l'invariant s'exprime "quel que soit l'entier k tel que 2 ≤ k ≤ (nb-1), k n'est pas un diviseur de nb" – et donc nb est par définition un nombre premier.*

c. *On fait varier i de 2 à nb-1 donc la boucle correspondante s'écrirait "`for i in range(2, nb)`".*

d. *Il est évident que le plus grand diviseur d'un nombre (autre que lui-même) est sa moitié s'il est pair, ou son tiers s'il est impair mais un multiple de trois, etc. En tout état de cause, aucun nombre ne pourra avoir de diviseur strictement supérieur à sa moitié (ou la partie entière de sa moitié dans le cas d'un nombre impair). En écrivant la condition "`while i <= (nb`*

// 2)" on économiserait presque la moitié des opérations effectuées par la boucle tout en maintenant le résultat correct...

Exercice 4: Variant & invariant de boucle — encore un !

Soit la fonction suivante :

```
def RechercheMax(Lst):
    '''
    Fonction qui prend en entrée une liste de nombres positifs
    et renvoie la valeur du plus grand d'entre eux.
    '''
    res = 0
    for i in range(len(lst)):
        if lst[i] > res:
            res = lst[i]

    return res
```

Donner un invariant de boucle pour cette fonction et montrer qu'elle est correcte.

1. *Choix de l'invariant : le but de la boucle est de trouver le maximum de la liste, donc avant chaque étape de la boucle, la variable* `res` *contient le maximum des éléments d'indice de 0 à* `i-1` *de la liste ;*
2. *Avant la première itération de la boucle,* `i` *vaut 0, donc la liste d'éléments dont l'indice est compris entre 0 et (i-1) est vide – donc on peut dire que l'invariant est vérifié ;*
3. *Si l'invariant est vérifié pour une valeur de i, alors pour la valeur suivante $i' = i + 1$, deux cas sont possibles :*
   - *Soit lst[i] $\leq$ res et donc, par définition, l'invariant reste vérifié (le maximum n'a pas changé) ;*
   - *Soit lst[i] > res, res prend alors la valeur lst[i] qui est le nouveau maximum des éléments d'indice compris entre 0 et (i'-1) (donc i), et donc l'invariant est vérifié aussi.*
4. *A la fin de la dernière itération de la boucle, i vaut len(lst), donc res contient le maximum des éléments d'indice de 0 à* `len(lst)-1` *de la liste – donc de toute la liste, et donc la boucle est correcte.*

Exercice 5: Complexité – inversion de liste

Soit la fonction suivante :

```
def InverseListe(lst):
    '''
    Fonction qui prend en entrée une liste quelconque et qui renvoie la
    ↪  liste inversée.
    '''
    # On initialise la liste résultat - liste vide
    res = []
    # On la remplit en partant de la fin de la liste en entrée
    for i in range(len(lst)):
        res.append(lst[len(lst) - i - 1])

    return res
```

a. Montrer, en utilisant un invariant de boucle, qu'elle fait bien ce que sa

docstring spécifie.

b. Compter son nombre d'opérations élémentaires, et en déduire sa complexité (préciser sa dénomination et sa notation en "$\mathcal{O}$").

a. L'invariant de boucle est la propriété, pour une valeur de i et après la réalisation de la boucle : "res est une liste contenant les i+1 derniers éléments de lst, dans l'ordre inverse". Initialisation : i=0, res contient uniquement lst[len(lst) - 1], soit le dernier élément de lst, donc l'invariant est vérifié. Si c'est vérifié pour i, lorsqu'on passe à i' = i + 1, on ajoute à la fin de res l'élément de lst d'indice (len(lst) - i' - 1) qui est le (i' + 1)^{ème} en partant de la fin, et on a donc rallongé res de 1 (sa longueur est donc maintenant i'+1) – l'invariant est donc bien vérifié à i'. La boucle se termine lorsque i = après len(lst) itérations, quand i est égal à len(lst) - 1, et donc à ce moment l'invariant s'exprime "res est une liste contenant les (len(lst)) derniers éléments de lst, dans l'ordre inverse" – donc tous les éléments, et donc la boucle est correcte.

b. Si l'on compte strictement les opérations on a :

- *Deux opérations en dehors de la boucle (`res = []` et `return res`) ;*
- *Une opération à l'intérieur de chaque itération de la boucle (le append sur la lsite res) ;*
- *Deux opérations à chaque itération de la boucle – l'incrémentation de i et sa comparaison avec len(lst).*

Si on note n la longueur de lst, la boucle est réalisée n fois et donc le nombre d'opérations est : $3 \times n + 2$ – et donc on en déduit que la complexité est en $\mathcal{O}(n)$, c'est-à-dire linéaire.

Exercice 6: Complexité – note pondérée

Soit la fonction suivante :

```
1  def EltPondere(lst_notes, indice):
2      '''
3      Fonction qui prend en entrée une liste de notes et un indice dans
       ↪  cette liste et qui renvoie la valeur pondérée de cette note (donc
       ↪  sa valeur divisée par le nombre de notes).
4      '''
5      res = lst_notes[indice] / len(lst_notes)
6      return res
```

Déterminer sa complexité.

Aucun piège ici – certes, on utilise len(lst_notes) dans la fonction, mais pas dans le cadre d'une boucle. On n'exécute donc ici que deux opérations : le calcul de res et le return. On est donc dans une complexité constante, soit $\mathcal{O}(1)$.

Exercice 7: Complexité – somme de produits ou produit de sommes ?

Soit les deux fonctions suivantes :

```python
def ProdSomme(n1, n2):
    '''
    Fonction qui prend en entrée deux entiers n1 et n2 et renvoie
    le produit des sommes de tous les i et j pour i < n1 et j < n2.
    '''
    res1 = 0
    res2 = 0
    for i in range(n1):
        res1 += i
    for j in range(n2):
        res2 += j

    res = res1 * res2

    return res
```

```python
def SommeProd(n1, n2):
    '''
    Fonction qui prend en entrée deux entiers n1 et n2 et renvoie
    la somme de tous les produits i.j pour i < n1 et j < n2.
    '''
    res = 0
    for i in range(n1):
        for j in range(n2):
            res += i * j
            #On rappelle que "a += b" est équivalent à "a = a + b"

    return res
```

a. Démontrer (simplement) que si j'applique ces deux fonctions aux mêmes paramètres n1 et n2 leur retour sera identique – qu'en d'autres termes elles font le même calcul.

b. Quelles sont les nombres d'opérations élémentaires qu'effectuent chacune de ces deux fonctions (exprimés en fonction de n1 et n2) ? Laquelle des deux, du coup, vous semble la meilleure pour atteindre le résultat ?

a. *On voit de manière évidente que si l'on développe le produit de ProdSomme on tombera sur la somme de SommeProd :*

ProdSomme renvoie : $[1+2+(...)+(n_1-1)] \times [1+2+(...)+(n_2-1)]$
SommeProd renvoie : $1\times 1+1\times 2+(...)+(n_1-1)\times(n_2-1)$

b.
- *ProdSomme effectue successivement deux boucles de longueurs respectives n1 et n2 – donc le nombre d'opérations élémentaires qu'elle va effectuer est de l'ordre de* $n1+n2$ [a].
- *SommeProd, lui, effectue deux boucles imbriquées : la boucle intérieure (`for j in range(n2)`) va effectuer n2 itérations tandis que la boucle extérieure (`for i in range(n1)`) va en effectuer n1 – donc le nombre d'opérations élémentaires qu'elle va effectuer est de l'ordre de* $n1\times n2$ [a].

Il est donc évident que, pour des valeurs importantes de n1 et n2, ProdSomme sera beaucoup moins complexe (et donc beaucoup plus performante)

que SommeProd – et ce pour le même résultat. Si on choisit par exemple $n1 = 1.000.000$ (un million) et $n2 = 2.000.000$ alors ProdSomme effectuera environ $n1 + n2 = 3.000.000$ d'opérations, tandis que SommeProd en effectuera environ $n1 \times n2 = 2.000.000.000.000$ (deux mille milliards) – soit plus de 650.000 fois plus !

a. Notez qu'ici on parle bien d'ordre de grandeur et qu'on néglige les opérations unitaires (comme le `res = 0` ou `return res`) qui ne changeront rien à la performance de la fonction quand on commencera à utiliser de grandes valeurs de n1 et n2.

Exercice 8: Tri à bulles - bidouillons-le un peu...

On rappelle le code Python du tri à bulles dans sa version la plus basique :

```
1  def TriBulles(liste):
2      '''
3      Fonction qui effectue le tri par permutations de la liste passée en
       ↪  entrée.
4      '''
5      n = len(liste)
6      for i in range(n):
7          for j in range(n-1):
8              if liste[j] > liste[j+1]:
9                  # Cette syntaxe permet d'affecter deux variables
                   ↪  simultanément et donc de ne pas passer par une
                   ↪  variable intermédiaire
10                 liste[j], liste[j+1] = liste[j+1], liste[j]
11     return liste
```

a. A quoi sert l'indice "i" dans cette fonction ?

b. Que se passe-t-il si on remplace la ligne 7 par `for j in range(n-1, -1, -1)` ? Cette syntaxe – qui n'est rien de plus qu'une utilisation de la syntaxe `range` habituelle mais dans le contexte d'une suite décroissante) permet de réaliser une boucle allant de n-2 (le premier paramètre) à 0 (arrêt juste avant d'atteindre le deuxième paramètre) par pas de -1 (le troisième paramètre). L'indice va donc décrire la suite (n-2), (n-3), ..., 1, 0 au lieu de 0, 1, ..., (n-3), (n-2).

c. Que se passe-t-il si on remplace la ligne 8 par `if liste[j] < liste[j+1]:` ?

d. La boucle interne ne fait-elle pas des opérations inutiles ? Comment la modifier pour la rendre plus efficace ?

a. *Uniquement à compter le nombre de fois que l'on fait tourner la boucle interne ! La variable i, comme on peut le constater, n'est utilisée nulle part hormis dans le `for` de la boucle.*

b. *Eh bien... Rien du tout sur le résultat : on continue à échanger les éléments si celui de droite (d'indice supérieur) est inférieur à celui de gauche, donc on met à terme la liste dans le même ordre que précédemment. En revanche, on s'y prend dans l'autre sens, ce qui modifie la démarche : au lieu de faire "remonter" le plus grand élément du début à la fin de la liste, on va faire ici "descendre" le plus petit de la fin au début. Concrètement, si on passe en entrée la liste qu'on a utilisée en cours (`[2, 5, 3, 1]`) on passera par les étapes suivantes :*

(a) `[2, 5, 3, 1]`
(b) `[1, 2, 5, 3]`
(c) `[1, 2, 3, 5]`

c. *Là, en revanche, on inverse le sens de tri – on fait une permutation si l'élément de droite (d'indice supérieur) est* ***supérieur*** *à celui de gauche : donc on obtiendra à la fin la liste triée par ordre décroissant au lieu de croissant.*

d. *A la fin de la première itération de la boucle externe (donc quand la boucle interne a fini son travail une première fois), on a ramené le plus grand élément de la liste à la fin de celle-ci ; à la fin de la deuxième itération, le deuxième plus grand est en avant-dernière position ; (...) ; à la fin de la $i^{ième}$ itération, le $i^{ième}$ plus grand est en $i^{ième}$ position en partant de la fin, et tous ceux qui le suivent sont positionnés en ordre croissant (si vous ne me croyez pas, démontrez-le en utilisant un invariant de boucle – c'est exactement à cela qu'ils servent !!). Donc quand on passe à la $(i+1)^{ième}$ itération de la boucle, on a déjà les i éléments "de droite" (de plus grands indices) de la liste qui sont déjà triés et qu'il ne sert plus à rien, de trier de nouveau – on peut donc modifier la boucle interne pour qu'elle ne regarde plus ces éléments, et donc soit plus optimale, en la faisant aller non plus de 0 à n-2 mais de 0 à n-i-2. Le résultat sera le suivant :*

```
def TriBullesOptim(liste):
    '''
    Fonction qui effectue le tri par permutations de la liste
    ↪ passée en entrée.
    '''
    n = len(liste)
    for i in range(n):
        # On ne regarde dans la boucle interne que les éléments qui
        ↪ ne sont pas déjà triés
        for j in range(n-i-1):
            if liste[j] > liste[j+1]:
                liste[j], liste[j+1] = liste[j+1], liste[j]
    return liste
```

Exercice 9: Devinette de nombre

Le code utilisé pour l'implémentation de la fonction de recherche dichotomique est le suivant :

```python
def RechDich(T, x):
    '''
    Fonction qui effectue une recherche dichotomique de x dans T.
    '''
    debut = 0
    fin = len(T) - 1
    while fin >= debut:
        milieu = (fin + debut) // 2
        if T[milieu] == x:
            return milieu
        elif x > T[milieu]:
            debut = milieu + 1
        else:
            fin = milieu - 1
    return -1
```

Appuyez-vous sur cette approche et développez le jeu évoqué en cours – vous choisissez un nombre entre 0 et 100 et l'ordinateur doit le deviner en vous posant des questions.

Exemple de séquence :

```
Pensez à un nombre entre 0 et 100...
Si il est plus grand que 50, tapez 1; s'il est plus petit, tapez 2;
s'il est égal à 50 tapez 3.
2
Si il est plus grand que 25, tapez 1; s'il est plus petit, tapez 2;
s'il est égal à 25 tapez 3.
3
J'ai trouvé! Votre nombre est 25.
```

Indices :

- Vous allez évidemment devoir utiliser la fonction "input()" pour recueillir les réponses de l'utilisateur ;
- Vous allez construire de nombreuses fois le même message, avec des valeurs différentes. Voici un exemple qui pourra vous aider :

```
>>> nb = 50
>>> msg = "Voici un message avec le nombre " + str(nb) + " dedans."
>>> print(msg)
Voici un message avec le nombre 50 dedans.
```

Version améliorée : l'ordinateur doit être capable de détecter si l'utilisateur triche (change de nombre en cours de route)...

Le code demandé ici est relativement simple :

```python
def DevineNb():
    '''
    Fonction qui s'appuie sur la recherche dichotomique pour deviner un nombre entre 0 et 100 auquel pense l'utilisateur.
    '''
    debut = 0
    fin = 100
    # Compteur de coups avant d'avoir deviné
    cps = 0
    print("Vous devez penser à nombre compris entre 0 et 100 et je dois le deviner. C'est parti...")
    while fin >= debut:
        cps += 1
        milieu = (fin + debut) // 2
        print("Je devine que le nombre est", milieu, ".")
        print("Si j'ai raison, entrez 1; si votre nombre est supérieur à", milieu, "entrez 2; s'il est inférieur, entrez 3.")
        rep = int(input("Votre réponse?  "))
        if rep == 1:
            print("J'ai trouvé en", cps, "coup(s)!")
            # Un return force à sortir de la fonction
            return
        # Application stricte de l'algorithme de recherche dichotomique
        elif rep == 2:
            debut = milieu + 1
        else:
            fin = milieu - 1
```

Pour ce qui est de la version améliorée, il est impossible de détecter que l'utilisateur a menti tant qu'on n'a pas fini l'algorithme : en effet, à tout moment, tant que `debut <= fin`*, on n'a pas couvert la totalité des nombres possibles et donc le nombre deviné par l'utilisateur peut en être un qu'on n'a pas encore couvert. En revanche, si l'on termine l'algorithme sans avoir trouvé (lorsque* `debut > fin`*), on est forcément dans un cas où l'utilisateur a triché (puisque, on l'a démontré en cours, l'algorithme est correct). On peut donc, après la boucle, ajouter un message pour dire à l'utilisateur qu'on l'a capté !*

Exercice 10: Recherche dichotomique

Donnez deux exemples d'exécution de recherche dichotomique où le nombre d'exécutions de la boucle est exactement 4 — un sans solution (retour de -1), une avec une solution. Dans les deux cas il vous est demandé de donner la liste considérée et le nombre recherché.

Le cas le plus simple est celui où l'on ne trouve pas la valeur cherchée : en effet, comme on l'a vu en cours, c'est le cas le pire du point de vue de la complexité, et dans ce cas elle est en $\mathcal{O}(log_2(n))$. Donc pour avoir 4 opérations, il faut une liste de longueur $2^4 = 16$. Donc, par exemple, la recherche de "20" dans la liste `[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16]` *se fera en 4 itérations.*

Si l'on y réfléchit bien, le cas où l'on va trouver l'élément (si on ne demande pas plus de conditions comme c'est le cas ici) n'est pas beaucoup plus simple puisque l'algorithme de la recherche dichotomique est construit de telle sorte

que les derniers éléments à être considérés sont les extrémités du tableau. On peut donc dire que la recherche de "1" (tout comme celle de "16") dans la liste `[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16]` *se fera en 4 itérations.*

Exercice 11: Recherche dichotomique modifiée

a. Modifier l'algorithme de recherche dichotomique afin qu'il indique le nombre valeurs examinées dans un tableau donné pour localiser un élément dans un tableau trié ou indiquer qu'il n'est pas présent dans le tableau

b. Quels tests peut-on imaginer pour cet algorithme ?

a. *La modification de l'algorithme est simple – il suffit d'ajouter un compteur et de l'afficher (très similaire à ce que l'on a fait avec la variable* `cps` *dans l'exercice 9 ci-dessus). Notez :*

- *Les lignes 4, 8, et 10 de l'algorithme ci-dessous où l'on implémente le compteur ;*
- *La ligne 7 où l'on stocke T[Milieu] dans une variable, ce qui nous permet de l'utiliser plusieurs fois en ne la lisant qu'une seule fois.*

Entrée: *T, x*
Sortie: *position de l'élément x dans T, -1 si non trouvé*

```
1:  fonction RechDichotomique(T, x)
2:      Debut ← 1
3:      Fin ← longueur(T)
4:      ValLues ← 0
5:      tant que Debut ≤ Fin faire
6:          Milieu ← (Debut + Fin)//2
7:          ValCur ← T[Milieu]
8:          ValLues ← ValLues + 1
9:          si x = ValCur alors          ▷ On a trouvé l'élément recherché
10:             Afficher "On a lu : " ValLues " valeurs dans le tableau"
11:             retourner Milieu
12:         sinon si x > ValCur alors
13:             Debut ← Milieu + 1       ▷ L'élément n'est pas "à gauche"
14:         sinon
15:             Fin ← Milieu − 1         ▷ L'élément n'est pas "à droite"
16:         fin si
17:     fin tant que
18:     retourner −1                     ▷ On ne l'a pas trouvé
19: fin fonction
```

b. *Pour les tests à effectuer il importe d'essayer de couvrir le plus de cas possibles. On peut par exemple imaginer un jeu de tests contenant :*

- *Une recherche de 2 dans* `L = [1, 2, 3, 4, 5, 6]` *(valeur trouvée, qui n'est pas à une extrémité) ;*
- *Une recherche de 5 dans* `L` *(valeur non trouvée) ;*
- *Une recherche de 1 puis 4 dans* `L` *(valeur trouvée, qui est à une extrémité) ;*
- *Recherche de 1 dans* `LVide = []` *(liste vide).*

Exercice 12: Jeu des 7 différences (enfin 2 en l'occurrence...)

Soit le code suivant :

```
1  def RechDich(T, x):
2      '''
3      Fonction qui effectue une recherche dichotomique de x dans T.
4      '''
5      debut = 0
6      fin = len(T) - 1
7      while fin >= debut:
8          milieu = (fin + debut) // 2
9          if T[milieu] == x:
10             return milieu
11         elif x > T[milieu]:
12             debut = milieu
13         else:
14             fin = milieu
15     return -1
```

Voyez-vous les deux différences entre ce code et celui qu'on a développé en cours ? Quel problème est-ce que cela pose ? Que se passerait-il si je lançais la commande `RechDich([1, 2, 3, 4, 5], 5)` ?

Les différences se situent aux lignes 12 et 14 du code où l'on a `debut = milieu` *et* `fin = milieu` *au lieu de* `debut = milieu + 1` *et* `fin = milieu - 1` *respectivement. Ceci pourrait entrainer une boucle infinie dans certains cas comme on va le voir sur l'exemple proposé dans l'énoncé (comme on travaille ici sur du code Python et non du pseudo-code j'utilise les indices de 0 à n) :*

Debut	*Fin*	*Milieu*	*liste[Milieu]*
0	*4*	*2*	*3*
2	*4*	*3*	*4*
3	*4*	*3*	*4*
3	*4*	*3*	*4*

etc.....

Vous remarquerez que c'est le seul cas où le problème se posera — une recherche de n'importe quelle autre valeur fonctionnera sans problème.

Exercice 13: Une fonction tordue

On considère le code de fonction suivant :

```
1  def NbrDup(t):
2      '''
3      Fonction qui fait quelque chose d'assez tordu.....
4      '''
5      n = len(t)
6      n_dup = [0] * n
7      for i in range(n):
8          for j in range(i + 1, n):
9              if t[i] == t[j]:
10                 n_dup[i] += 1
11
12     return n_dup
```

a. A la main, déterminez le résultat de l'appel à cette fonction pour les cas

suivants :

(a) `t = [1, 2, 3]`

(b) `t = [1, 1, 1]`

(c) `t = [1, 1, 3]`

(d) `t = [1, 2, 2]`

b. Comment, du coup, formuleriez-vous la `docstring` de cette fonction pour expliquer ce qu'elle fait ?

c. En termes de complexité (donc du nombre d'opérations effectuées par la fonction), lequel des 4 cas suivants est le pire ?

d. On se place dans un cas "au pire" : combien de fois, en fonction de n la taille du tableau `t`, l'instruction `n_dup[i] += 1` est-elle effectuée lorsque :

(a) i vaut 0 ?

(b) i vaut 1 ?

(c) i vaut $n-1$?

e. En déduire, en justifiant bien, la complexité en fonction de n de la fonction `NbrDup`.

a. Reprenons l'approche que nous avons déjà utilisée plus haut, à savoir un tableau qui nous permet de suivre les valeurs successives de différentes variables :

(a) Pour `t = [1, 2, 3]` :

Ligne Code	*n_dup*	*i*	*j*	*t[i]*	*t[j]*
6	*[0,0,0]*	*-*	*-*	*-*	*-*
9	*[0,0,0]*	*0*	*1*	*1*	*2*
9	*[0,0,0]*	*0*	*2*	*1*	*3*
9	*[0,0,0]*	*1*	*2*	*2*	*3*

Et le traitement renvoie donc `[0,0,0]`

(b) Pour `t = [1, 1, 1]` :

Ligne Code	*n_dup*	*i*	*j*	*t[i]*	*t[j]*
6	*[0,0,0]*	*-*	*-*	*-*	*-*
9	*[0,0,0]*	*0*	*1*	*1*	*1*
10	*[1,0,0]*	*0*	*1*	*1*	*1*
9	*[1,0,0]*	*0*	*2*	*1*	*1*
10	*[2,0,0]*	*0*	*1*	*1*	*1*
9	*[2,0,0]*	*1*	*2*	*1*	*1*
10	*[2,1,0]*	*0*	*1*	*1*	*1*

Et le traitement renvoie donc `[2,1,0]`

(c) Pour `t = [1, 1, 3]` :

Ligne Code	*n_dup*	*i*	*j*	*t[i]*	*t[j]*
6	*[0,0,0]*	-	-	-	-
9	*[0,0,0]*	*0*	*1*	*1*	*1*
10	*[1,0,0]*	*0*	*1*	*1*	*1*
9	*[1,0,0]*	*0*	*2*	*1*	*3*
9	*[1,0,0]*	*1*	*2*	*1*	*3*

Et le traitement renvoie donc `[1,0,0]`

(d) *Pour* `t = [1, 2, 2]` *:*

Ligne Code	*n_dup*	*i*	*j*	*t[i]*	*t[j]*
6	*[0,0,0]*	-	-	-	-
9	*[0,0,0]*	*0*	*1*	*1*	*2*
9	*[0,0,0]*	*0*	*2*	*1*	*2*
9	*[0,0,0]*	*1*	*2*	*2*	*2*
10	*[0,1,0]*	*1*	*2*	*2*	*2*

Et le traitement renvoie donc `[0,1,0]`

b. *On pourrait écrire (mais je vous accorde que c'est bien tordu...) :*

```
''' Fonction qui prend en entrée un tableau t et qui renvoie
un tableau n_dup de même taille que t qui contient, à chaque
indice, le nombre de fois que t[i] est présent dans la suite
du tableau (donc dans les indices [i+1, n-1]).'''
```

c. *La réponse est à la fois logique et facile à constater en regardant les tableaux ci-dessus. La seule instruction qui n'est pas effectuée à chaque fois est* `n_dup[i] += 1` *puisqu'elle dépend de la condition* `if t[i] == t[j]` *; le cas le pire est donc celui où cette condition est vérifiée un maximum de fois — donc dans le cas (b) ci-dessus, celui où toutes les valeurs du tableau sont identiques. Ceci se vérifie aisément en constatant le nombre de lignes des tableaux qu'on a utilisés ci-dessus, qui est maximal dans le cas (b).*

d. *On se place donc ici dans le cas d'un tableau de longueur n contenant n fois la même valeur (voir ci-dessus : cas le pire, celui où la condition* `if t[i] == t[j]` *est vérifiée à chaque itération de la boucle).*

   (a) *Si $i = 0$ alors j ira de 1 à $(n-1)$ et donc l'instruction sera exécutée $(n-1)$ fois.*

   (b) *Si $i = 1$ alors j ira de 2 à $(n-1)$ et donc l'instruction sera exécutée $(n-2)$ fois.*

   (c) *Si $i = (n-1)$ alors j ira de n à $(n-1)$ et donc l'instruction ne sera pas exécutée du tout (exécutée 0 fois).*

e. *On peut généraliser ce que l'on vient de calculer en disant que lorsqu'on a $i = k$, l'instruction* `n_dup[i] += 1` *est exécutée exactement $(n-1-k)$ fois. Donc si l'on regarde le traitement dans son ensemble le nombre d'exécutions de l'instruction sera :*

$$(n-1)+(n-2)+(...)+2+1+0 = \frac{n\times(n-1)}{2}$$
$$= \frac{1}{2}\times n^2 - n\times\frac{1}{2}$$

La complexité de la fonction dépend donc du carré de n, elle est donc quadratique et se note "$\mathcal{O}(n^2)$".

Exercice 14: Un peu de travail sur les dictionnaires

On considère le code de fonction suivant :

```
def Occ(t):

    res = {}
    for x in t:
        if x in res:
            res[x] += 1
        else:
            res[x] = 1

    return res
```

a. ***Encore*** une fonction où on a oublié de mettre la `docstring`!! Que proposez-vous ?

b. Ecrire une fonction `compare_tableaux(t, u)` qui prend deux tableaux `t` et `u` en entrée et détermine si ils contiennent les mêmes éléments, mais sans les trier.

a. *Cette fonction prend un tableau en entrée et renvoie en sortie un dictionnaire avec tous les éléments distincts présents dans le tableau ainsi que le nombre d'occurrences de chacun de ces éléments.*

b. *Il est évident que la fonction précédente ne nous a pas été fournie par hasard – on va s'appuyer dessus pour rédiger la fonction* `compare_tableaux` *:*

```python
def compare_tableaux(t, u):

    # Cas le plus simple pour commencer
    if len(t) != len(u):
        return False

    # On appelle la fonction occ sur les deux tableaux
    dt = Occ(t)
    du = Occ(u)

    # On compare ensuite les deux dictionnaires produits
    for val in dt:
        if val not in du:
            # Une valeur présente dans t qui n'est pas dans u
            return False
        elif dt[val] != du[val]:
            # Une valeur présente dans les deux mais pas le même nombre de fois
            return False

    # Démarche habituelle: on n'est tombé dans aucun des cas False - on retourne donc True
    return True
```

Exercice 15: Ah tiens au fait...

Est-ce que la fonction `Occ` qu'on a vue à l'exercice précédent fonctionnerait si on lui passait en entrée une chaine de caractères au lieu d'une liste ? Que renverrait l'appel `Occ("abracadabra")` ?
Qu'en serait-il de `compare_tableaux` si on lui passait deux chaines de caractères en entrée ?

Dans `Occ` on ne "touche" à la variable passée en argument qu'une seule fois, dans la boucle principale `"for x in t"` – et on a vu en cours qu'on peut tout à fait parcourir une chaîne de caractères de cette manière. Donc l'appel `Occ("abracadabra")` fonctionnerait bien et renverrait `{'a': 5, 'b': 2, 'r': 2, 'c': 1, 'd': 1}`.

Dans `compare_tableaux`, on "touche" à nos variables en entrée deux fois :

- *Via la fonction `len` – dont on a vu en cours qu'elle fonctionne très bien avec les chaines de caractère ;*
- *Via la fonction `Occ` – qu'on vient de regarder.*

Donc en dépit de son nom `compare_tableaux` fonctionnerait très bien avec des chaines de caractères.

Exercice 16: Tris décroissants

a. Ré-écrire l'algorithme du tri par sélection tel qu'on l'a vu en cours pour qu'il produise un tri par ordre décroissant ;

b. Même chose avec l'algorithme de tri par insertion.

a. *Il suffit de substituer la notion de maximum à la notion de minimum pour inverser l'ordre du tri dans le tri par sélection :*

Entrée: *tab, une liste*
Sortie: *tab, triée*

```
1: fonction TriSelection(tab)
2:     n ← longueur(tab)
3:     pour p allant de 1 à (n-1) faire
4:         pmax ← p
5:         pour j allant de (p+1) à n faire
6:             si tab[j] > tab[pmax] alors ▷ On a trouvé un nouveau max
7:                 pmax ← j
8:             fin si
9:         fin pour
10:        Échanger tab[pmax] et tab[p]
11:    fin pour
12:    retourner tab
13: fin fonction
```

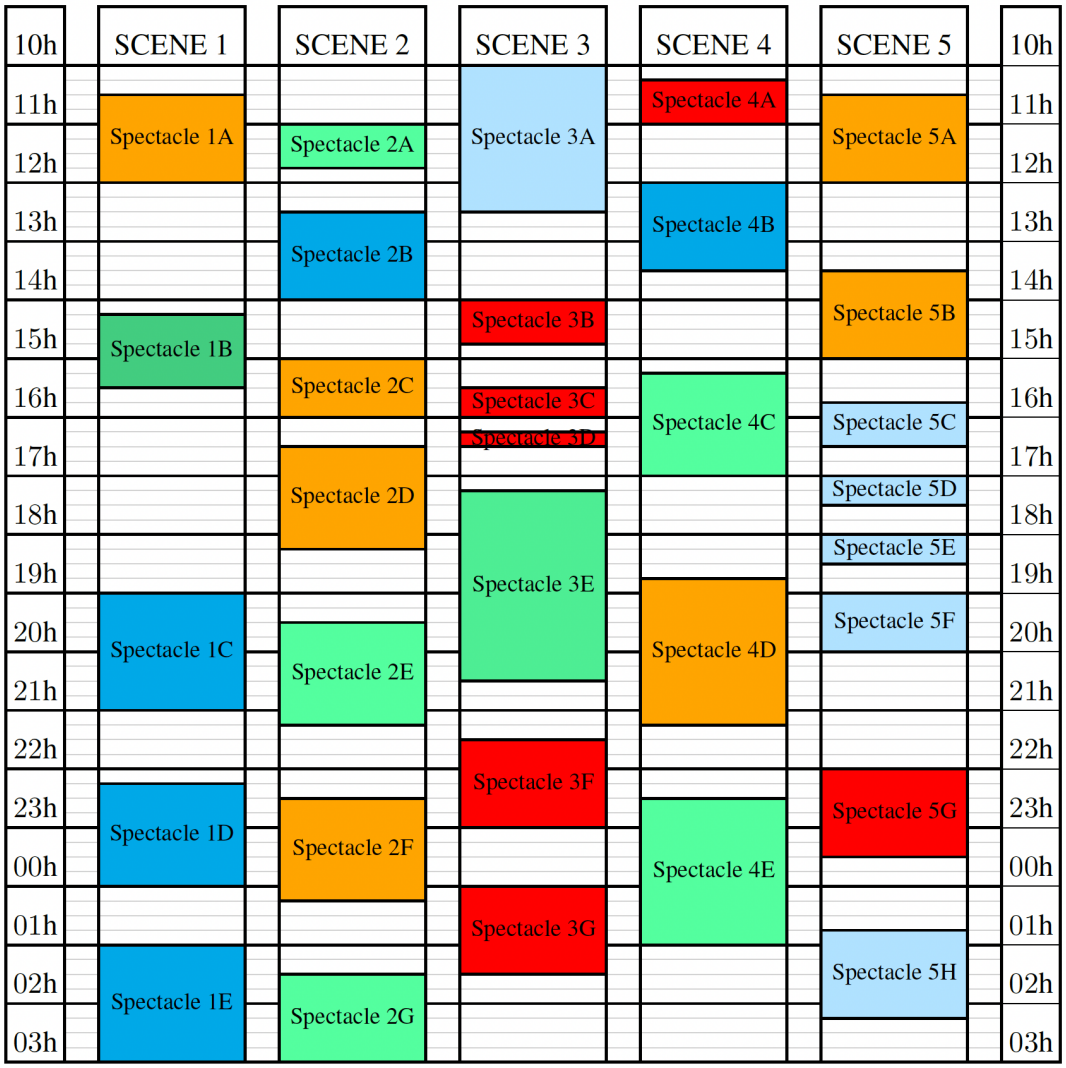
b. *Dans le tri par insertion, c'est encore plus simple : lorsqu'on remonte la partie déjà triée du tableau, on se contente de chercher l'indice où* `temp` *dépasse* `tab[p-1]` *et on insère temp à cet endroit (la seule ligne impactée est la ligne 6 où l'on a remplacé un "$>$" par un "$<$") :*

Entrée: *tab, une liste*
Sortie: *tab, triée*

```
1: fonction TriInsertion(tab)
2:     n ← longueur(tab)
3:     pour i allant de 2 à n faire
4:         temp ← tab[i]
5:         p ← i
6:         tant que p > 1 et tab[p-1] < temp faire
               ▷ On remonte le tableau jusqu'à trouver la place de tab[i]
7:             tab[p] ← tab[p-1]
8:             p ← p-1
9:         fin tant que
10:        tab[p] ← temp                    ▷ On a trouvé la place de tab[i]
11:    fin pour
12:    retourner tab
13: fin fonction
```

Exercice 17: Un festival glouton

Vous venez d'arriver au Festival d'Avignon et découvrez qu'il se déroule sur 5 scènes différentes, avec une programmation extrêmement complexe :



Brillant élève de NSI que vous êtes, vous vous décidez à choisir les spectacles auxquels vous allez assister en appliquant une méthode gloutonne. Les règles à respecter sont :

— Vous voulez voir le nombre maximal possible de spectacles dans la journée ;

— Lorsque vous commencez un spectacle vous voulez le voir en entier (du début à la fin) ;

— On suppose que le temps de trajet entre les scènes est nul.

On va considérer deux algorithmes gloutons séparés : appliquez le déroulé de chacun d'entre eux et concluez sur celui qui fonctionne le mieux.

a. Algorithme A : moins on attend entre deux spectacles plus on en verra, donc à chaque étape on choisit le prochain spectacle qui débute le plus tôt.

b. Algorithme B : plus chaque spectacle que l'on voit finit tôt plus on aura de temps pour voir d'autres spectacles, donc à chaque étape on choisit le prochain spectacle qui finit en premier.

a. *Algorithme A :*

Etape 1 *3A*

Etape 2 *2B*
Etape 3 *3B*
Etape 4 *2C*
Etape 5 *3D*
Etape 6 *2D*
Etape 7 *4D*
Etape 8 *3F*
Etape 9 *3G*
Etape 10 *2G*

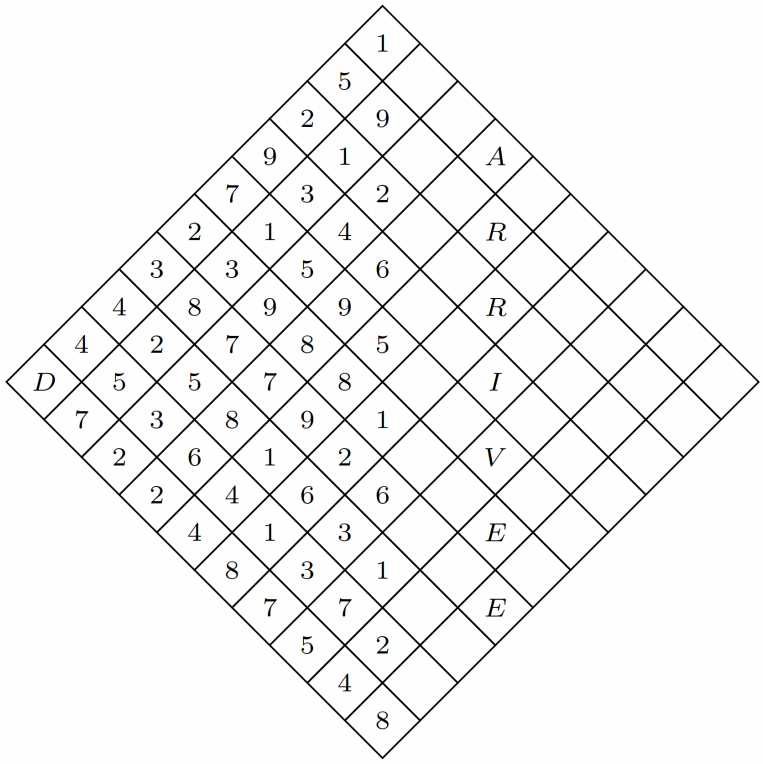
b. Algorithme B :

Etape 1 *4A*
Etape 2 *2A*
Etape 3 *4B*
Etape 4 *3B*
Etape 5 *2C ou 3C*
Etape 6 *3D*
Etape 7 *5D*
Etape 8 *5E*
Etape 9 *5F*
Etape 10 *3F*
Etape 11 *3G*
Etape 12 *2G*

Et on conclut donc que l'approche B est mieux optimisée que l'approche 1A.

Exercice 18: Encore un peu d'algorithmes gloutons...

On considère la figure ci-dessous :



Le jeu sur cette grille consiste à partir n part de la case "**D**" à gauche et de se rendre sur une des cases vides à droite en se déplaçant de case en case. Lorsqu'on est sur une case on peut se déplacer sur une des deux cases voisines situées sur la droite, mais pas sur la case reliée uniquement par un sommet (donc de la case D on peut aller en 4 ou en 7 mais pas en 5).
On note S la somme de toutes les cases traversées. Par exemple on peut effectuer la trajectoire suivante : `D - 7 - 5 - 3 - 5 - 7 - 9 - 8 - 9 - 6` qui donne $S = 59$.
Le but du jeu est d'effectuer la trajectoire qui rend la somme S la plus petite possible.

a. Décrivez une approche gloutonne à la résolution de ce problème. Combien de calculs sont nécessaires pour choisir la trajectoire ?

b. En appliquant cette approche, quelle trajectoire et quelle somme S obtenez-vous ?

c. Sur cette grille, en cherchant bien, la trajectoire optimale donne une somme $S = 23$. Votre algorithme glouton a-t-il trouvé cette trajectoire optimale ? Si la réponse est non, est-ce que la trajectoire trouvée vous semble proche de la trajectoire optimale ?

d. Comment pourrait-on modifier l'algorithme pour l'améliorer ? Combien de calculs sont nécessaires pour choisir la trajectoire avec cette amélioration ?

e. Pour être sûr d'obtenir la solution optimale cependant, il n'y a d'autre choix que d'adopter une approche "force brute" – regarder toutes les trajectoires possibles et choisir celle au score le plus faible. Démontrer qu'on va devoir calculer 512 trajectoires. calculs.

a. *A chaque étape on a le choix entre deux cases : on va systématiquement choisir celle de valeur inférieure et, en cas d'égalité, on va en choisir une*

au hasard. Toutes les trajectoires ont 9 étapes donc on va effectuer 9 comparaisons avant d'obtenir notre trajectoire.

b. `D - 4 - 4 - 2 - 5 - 7 - 7 - 8 - 8 - 5`, *pour* $S = 50$.

c. *Il est évident que notre algorithme n'a pas trouvé la meilleure trajectoire, et d'assez loin.*

d. *On pourrait, à chaque coup, regarder les deux cases suivantes — donc 4 combinaisons à comparer — et choisir la case qui fournit la combinaison la plus faible. Par exemple, en partant de D, les premières étapes seraient :*

   — *Quatre premières combinaisons possibles : 4-4, 4-5, 7-5, 7-2. La combinaison la plus faible est la première (8), donc on choisit 4.*

   — *Quatre combinaisons suivantes : 4-2, 4-3, 5-2, 5-3 — donc on choisit 4.*

   — *Quatre combinaisons suivantes : 3-2, 3-8, 2-8, 2-5 — donc on choisit 3.*

   Et on constate qu'à cette étape on diverge de l'algorithme précédent. Je vous laisse l'appliquer jusqu'au bout — vous verrez qu'on arrive à une solution bien meilleure. Dans cette approche, à chaque étape, on va effectuer quatre calculs – pour arriver à un total de 36 opérations, donc.

e. *A chaque étape on a deux choix possibles – les deux cases sur la droite. Tous les trajets ont une longueur de 9 cases, donc le nombre de trajectoires possibles est bien de* $2^9 = 512$.

Exercice 19: Et un petit exo sur les fichiers CSV pour finir

Soit le code suivant :

```
import csv
fichier = open('Truc.csv', 'r', encoding = 'utf-8')
table = list(csv.DictReader(fichier))
```

Et soit le fichier `Truc.csv` contenant les données suivantes :

```
Eleve,Age,Classe
Loubna,17,1G5
Olivier,16,1G2
Lenny,17,1G2
```

a. Rédigez un programme qui affiche à l'écran tous les prénoms d'élèves du fichier `Truc.csv` les uns à la suite des autres.

b. Rédigez une fonction `AgeMoy(classe)` qui renvoie l'âge moyen des élèves présents dans la classe passée en argument.

a. *Aucune difficulté particulière ici, on parcourt juste la liste de dictionnaires et on affiche la valeur de la clé* `Eleve` *(attention aux majuscules !) à chaque fois :*

```
for elt in table:
    print(elt['Eleve'])
```

b. *Même concept fondamental que pour la question précédente, juste un tout petit peu plus compliqué puisqu'il faut bien penser à :*

   — *Convertir la valeur de la clé* `Age` *en entier (puisqu'elle provient d'un*

fichier et qu'elle est donc au format `string`*) ;*

- *Utiliser deux accumulateurs : l'un,* `NbEleves`*, pour compter les élèves concernés, et l'autre,* `SommeAges`*, pour accumuler tous les âges pour permettre à la fin de calculer la moyenne.*
- *Et bien entendu vérifier que l'élève que l'on considère est bien dans la classe demandée.*

Ce qui nous donne :

```
def AgeMoy(classe):
    SommeAges = 0
    NbAges = 0
    for elt in table:
        if elt['Classe'] == classe:
            NbAges += 1
            SommeAges += int(elt['Age'])
    return(SommeAges / NbAges)
```