Informatique Tronc Commun
TD n°6 Révisions

I range

R1. Qu'affiche la suite d'instructions suivantes :

```
for i in range(4):
    print(i)
```

Solution:

```
0
1
2
3
```

R2. Quels sont les rangs extrêmes dans la liste `L = [4,1,6,7,1,3]` ?
De façon générale quels sont les rangs extrêmes d'une liste `L` ?

Solution: Les rangs de `L = [4,1,6,7,1,3]` sont compris entre 0 et 5.
De façon générale, les rangs extrêmes de `L` sont 0 et `len(L)-1`

R3. Pour parcourir entièrement une liste, quelles valeurs doivent prendre `i` (pour `L[i]`) ?

Solution: `i` doit prendre toutes les valeurs entre 0 et `len(L)-1`

R4. Qu'affiche la suite d'instructions suivantes :

```
L = [4,1,6,7,1,3]
for i in range(len(L)):
    print(i)
```

Solution:

```
0
1
2
3
4
5
```

R5. Qu'affiche la suite d'instructions suivantes :

```
L = [4,1,3]
for i in range(len(L)):
    print(L[i+1])
```

Quel est le problème ?

Solution:

```
1
3
Error OutOfRange
```

Pour la dernière valeur de `i = len(L)-1`, alors `i+1=len(L)` et `L[i+1]` n'existe pas.

R6. Qu'affiche la suite d'instructions suivantes :

```
L = [4,1,3]
for i in range(len(L)):
    print(L[i-1])
```

Quel est le problème ?

Solution:

```
3
4
1
```

Pour `i=0`, `i-1=-1`, et `L[i-1]=L[-1]` renvoie le dernier élément de la liste

R7. Compléter le range pour que toute la liste ait été parcourue. Que fait cette suite d'instructions ?

```
L = [4,1,3]
for i in range(..... ):
    if L[i]<L[i+1] :
        L[i],L[i+1] = L[i+1],L[i]
```

Solution:

```
L = [4,1,3]
for i in range(0,len(L)-1):
    if L[i]<L[i+1] :
        L[i],L[i+1] = L[i+1],L[i]
```

R8. Compléter le range pour que toute la liste ait été parcourue. Que fait cette suite d'instructions ?

```
L = [4,1,3]
for i in range(..... ):
    if L[i]<L[i-1] :
        L[i],L[i-1] = L[i-1],L[i]
```

Solution:

```
L = [4,1,3]
for i in range(1,len(L)):
    if L[i]<L[i-1] :
        L[i],L[i-1] = L[i-1],L[i]
```

II Tris

On souhaite trier une liste, par ordre croissant. On donne ici quelques exemples de tris classiques. Aucun n'est rigoureusement au programme, tout en l'étant tous... Les algorithmes ne sont pas à connaître mais il faudrait pouvoir les écrire une fois l'algorithme rappelé.

Exercice n°1 Tri fusion

Le tri fusion est un **tri récursif** basé sur le principe de « diviser pour régner » : un problème initial sur n données est divisé en deux sous-problèmes portant sur $\frac{n}{2}$ données si possible. Ici, il s'agit de passer du tri d'une liste de n éléments aux tris de deux listes contenant moitié moins d'éléments. On trie alors la première moitié de la liste, la seconde moitié de la liste, et on fusionne ces deux listes triées en une seule liste triée en utilisant une fonction annexe.

R1. Écrire une fonction `fusion(L1:list,L2:list)->list` qui prend en entrée deux listes triées dans le même ordre et renvoie une liste triée résultant de la fusion des deux listes.

Solution:

```
def fusion(L1,L2):
    """Fonction récursive qui fusionne deux listes triées en une seule liste triée"""
    if L1==[]:
        return L2
    elif L2==[]:
        return L1
    elif L1[0]<L2[0]:
        return [ L1[0] ] + fusion (L1[1:] , L2 )
    else:
        return [ L2[0] ] + fusion (L2[1:] , L1 )
```

R2. Écrire la fonction récursive `tri_fusion(L:list)->list` qui renvoie la liste triée en utilisant l'algorithme du tri fusion.

Solution:

```
def tri_fusion(L):
    """tri fusion d'un tableau en récursif"""
    if len(L)<=1: #condition d'arrêt
        return L
    #fusion des deux demi-listes triées  (appels récursifs)
    return fusion ( tri_fusion( L[:len(L)//2] ) , tri_fusion( L[len(L)//2:] ) )
```

Exercice n°2 Tri rapide

Le tri rapide est un tri récursif dans lequel on divise un problème initial en deux sous-problèmes. On choisit un élément, que l'on notera `p`, appelé **pivot** (qui peut être, le premier élément de la liste, le dernier, ...), de l'enlever de la liste et de partitionner la liste en deux sous-tableaux : une liste contenant les éléments strictement inférieurs à `p` et une liste contenant les éléments strictement supérieurs à `p`. On trie récursivement chacune des deux listes et on rassemble tout.

R3. Écrire la fonction `tri_rapide(L:list)->list` qui renvoie la liste `L` triée.

Solution:

```
def tri_rapide(L):
    if len(L)<=1 :
        return M
    p=L[0] #choix du pivot, le premier élément de L
    L1,L2,L3=[],[],[]#création des trois listes vides L1, L2, L3, qui contiendront respectivement les éléments strictement inférieurs au pivot, égaux au pivot, strictement supérieurs
    for x in L : #parcours des éléments de la liste
        if x<p:
            L1.append(x)
        elif x==p
            L2.append(x)
        else:
            L3.append(x)
    return tri_rapide(L1) + L2 + tri_rapide(L3)  # appels récursifs et concaténation
```

Exercice n°3 Tri bulles

On considère une liste `L` à n éléments. Le tri à bulles consiste à faire remonter les éléments les plus grands en permutant successivement les éléments du tableau : on parcourt le tableau, et à chaque fois que l'élément de gauche est strictement supérieur à l'élément de droite, on les permute. À la fin de ce parcours, le plus grand élément du tableau est en dernière position. On recommence le parcours du tableau pour trier les $n-1$ éléments, puis les $n-2$ éléments, etc. Le nom « tri à bulles » vient du fait que les éléments les plus grands remontent plus vite, comme les bulles dans l'eau.

R4. Écrire une fonction itérative `tri_bulles(L:list)->list` qui trie une liste selon l'algorithme du tri à bulles.

Solution:

```
def tri_bulles(L):
    for i in range(len(L)-1): # i sert de compteur, une fois len(L)-1 éléments bien triés, ils sont tous triés
        for j in range(0,len(L)-1-i):
            if L[j]>L[j+1]:
                L[j] , L[j+1]   = L[j+1] , L[j]
    return L
```

R5. Écrire une fonction récursive `tri_bulles_rec(L:list)->list` qui trie une liste selon l'algorithme du tri à bulles.

Solution:

```
def tri_bulles_rec(L):
    if len(L)<=1: # condition d'arrêt
        return L
    else:
        for j in range(0,len(L)-1):
            if L[j]>L[j+1] : # on fait remonter l'élément le plus grand
                L[j] , L[j+1] = L[j+1] , L[j]
```

```
8         return tri_bulles_rec(L[:len(L)-1]) + [ L[:len(L)-1] ] # appel
    récursif, on applique la fonction sur la liste L sans le dernier
    élément, et on concatène avec le dernier élément
```

III Programmation dynamique

Exercice n°1 Le problème des stations-services

On considère le problème suivant : on s'apprête à partir en voyage en voiture et prévoit un itinéraire jusqu'à destination. Le voyage étant long, il faudra faire le plein plusieurs fois. Heureusement, on connait la position de stations-service sur le trajet et peut anticiper les arrêts.

Remarques préliminaires

Rappels de Python :

- Si `L` est une liste, alors `L[i]` désigne le i-ème élément de cette liste, où l'entier i est supérieur ou égal à 0 et strictement plus petit que la longueur `len(L)` de la liste.
- La commande `L[i]=x` affecte la valeur de l'expression `x` au i-ème élément de la liste `L`.
- L'expression `[]` construit une liste vide. L'expression `n*[x]` construit une liste de longueur `n` contenant `n` occurrences de `x`.
- La commande `L.append(x)` modifie la liste `L` en lui rajoutant un nouvel élément final contenant `x`.
- La commande `L.pop()` modifie la liste `L` en supprimant son dernier élément et en renvoyant sa valeur.

Important : Seules les opérations sur les listes apparaissant dans le paragraphe précédent sont autorisées dans les réponses. Si une fonction Python standard est nécessaire, elle devra être réécrite.

Notations liées au problème

Le problème des stations-service est formalisé de la manière suivante : n stations sont situés consécutivement sur une même route. On numérote par $0, 1, \ldots, n-1$ les stations et, pour $i \in [\![0, n-1]\!]$, on note d_i la distance (en km) de la station 0 à la station i.

On suppose les inégalités suivantes : $0 = d_0 < d_1 < \ldots < d_{n-1}$. Dans l'ensemble du problème, on travaillera avec une liste `dist` de taille n contenant les valeurs des d_i. Par exemple, la liste suivante représente une répartition de 6 stations :

```
dist = [0, 10, 20, 30, 70, 100]
```

On appelle **itinéraire** une suite d'indices strictement croissants commençant par 0 et terminant par $n-1$. Un itinéraire représente un trajet de la station 0 à la station $n-1$ et indique à quelle(s) station(s)-service intermédiaires un arrêt sera fait pour se ravitailler en carburant au cours du voyage.

On appelle **taille** d'un itinéraire I le nombre de stations de I qui ne sont ni 0 , ni $n-1$. Par exemple, `I=[0,1,4,5]` est un itinéraire de taille 2.

I Minimiser le nombre d'arrêts

Dans cette partie, on considère une version simplifiée du problème : la voiture possède un réservoir à capacité infinie, mais chaque station a une quantité limitée de carburant. Cette quantité sera donnée par une valeur s_i représentant la quantité de carburant disponible à la station i, en onces, avec l'hypothèse (pour simplifier) qu'une once de carburant permet de parcourir exactement 1 km . On dispose à cet effet d'une liste stock de taille n telle que `stock[i]` est égal à s_i.

On dit qu'un **itinéraire** est **valide** si, en récupérant tout le carburant disponible à chaque station de l'itinéraire, la voiture n'est jamais à cours de carburant. (*Le réservoir est initialement vidé et rempli à la station* $n°0$ *avec* s_0 *onces de carburant pour commencer le trajet.*)

On cherche à déterminer la taille minimale d'un itinéraire valide. Par exemple, si **dist** est la liste donnée précédemment et stock est donnée par :

```
stock = [20, 60, 30, 10, 40, 0]
```

alors `I=[0,1,4,5]` est un itinéraire valide de taille minimale.

Le trajet s'effectue de la manière suivante : la voiture dispose initialement de 20 onces de carburant. On conduit avec 20 onces jusqu'à la station 1, on remplit 60 onces (soit $20+60-10=70$ onces dans le réservoir à ce moment), on conduit jusqu'à la station 4, on remplit 40 onces (soit $70+40-60=50$ onces maintenant) et on conduit sans souci jusqu'au point d'arrivée, la station 5.

1. Proposer un schéma gradué sur un axe horizontal rectiligne faisant apparaitre les stations, les stocks, ainsi que l'itinéraire `I=[0,1,4,5]`.
2. On garde les mêmes variables `dist` et `stock` que précédemment.
   L'itinéraire `I=[0,2,3,4,5]` est-il valide ? Justifier.
   Proposer un itinéraire valide, non minimal, mais ne passant pas par toutes les stations.
3. Écrire une fonction `itineraire(I,n)` qui prend en arguments une liste d'entiers I et un entier n et renvoie un booléen `True` ou `False` selon que `I` est un itinéraire pour un trajet à `n` stations ou non (c'est-à-dire contenant des entiers triés compris entre 0 et $n-1$ commençant par 0 et finissant par $n-1$).
4. Écrire une fonction `valide(I,dist,stock)` qui prend en arguments une liste `I`, une liste de distances et une liste de stock de carburant et renvoie un booléen `True` ou `False` selon que `I` soit un itinéraire valide ou non. On effectuera par sécurité un test d'assertion pour vérifier que `dist` et `stock` sont de même taille.
5. En déduire une fonction `trajet_possible(dist,stock)` qui prend en arguments détermine s'il existe au moins un itinéraire valide ou non pour le problème des stations-service.

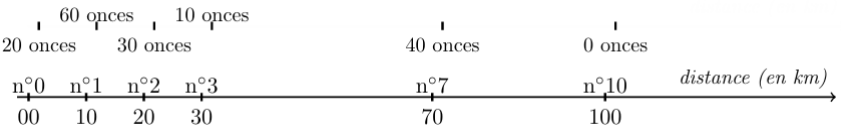
Pour $i \in [\![0, n-1]\!]$ et $k \in \mathbb{N}$, on pose $D(k,i)$ la distance maximale qu'on peut parcourir en s'arrêtant à au plus k stations parmi les stations d'indices compris entre 1 et i.

6. Pour $k \in \mathbb{N}$, que vaut $D(k,0)$? Pour $i \in [\![0, n-1]\!]$, que vaut $D(0,i)$?
7. Montrer que pour $i \in [\![0, n-2]\!]$ et $k \in \mathbb{N}$, on a :

$$D(k+1,i+1) = \begin{cases} D(k+1,i) & \text{si } D(k,i) < d_{i+1} \\ \max\left(D(k+1,i), D(k,i)+s_{i+1}\right) & \text{sinon} \end{cases}$$

8. En déduire un algorithme de programmation dynamique qui répond au problème des stations-service et l'implémenter sous forme d'une fonction `min_taille(dist,stock)`. (On autorise la fonction `max` ici.)
   Cette fonction renverra la taille minimale d'un itinéraire valide ou `-1` s'il n'existe pas d'itinéraire valide.
9. Déterminer la complexité temporelle de la fonction précédente en fonction de n, le nombre de stations.
10. [**Bonus**] Modifier l'algorithme précédent afin que la fonction renvoie étagement un itinéraire minimal `I` possible (n'importe lequel s'il en existe au moins un, et la liste vide sinon).

1. **Solution:**



2. **Solution:** On garde les mêmes variables dist et stock que précédemment. L'itinéraire I $= [0, 2, 3, 4, 5]$ n'est pas valide car arrivé à la station n°3, à 30 km du départ, on a rempli le réservoir avec 20+30+10 = 60 donc impossible d'arriver à la station n°4 qui est à 70 km du départ.
   Pour construire un itinéraire valide, non minimal, mais ne passant pas par toutes les stations, il suffit d'ajouter au moins une station à l'itinéraire minimal (ce n'est pas obligatoire mais cela fonctionne) sans les mettre toutes. Un exemple est donc $I = [0, 1, 3, 4, 5]$ ou $I = [0, 1, 2, 4, 5]$.

3. **Solution:**

```
def itineraire(I,n):
    for i in range(len(I) - 1): # vérifie le tri
        if I[i] >= I[i+1]:
            return False
    if I[0] == 0 and I[len(I)-1] == n-1: # vérifie que ça va de 0 à n-1
        return True
    else:
        return False
```

4. **Solution:**

```
def valide(I, dist, stock):
    assert len(dist)==len(stock)
    n = len(dist)
    if itineraire(I, n)==False:
        return False
    reservoir = 0
    for i in range(len(I)-1):
        if reservoir < 0:
            return False
        reservoir += stock[I[i]] # on ajoute le stock
        reservoir -= dist[I[i+1]] - dist[I[i]] # on enlève la distance
    parcourue
    return True
```

5. **Solution:**

```
def trajet_possible(dist,stock):
    I = [k for k in range(len(dist))] # itinéraire avec des arrêts à
    toutes les stations
    return valide(I,dist,stock) # s'il est possible, renvoie True, s'il
    n'est pas possible, aucun trajet possible
```

6. **Solution:** Pour $k \in \mathbb{N}, D(k, 0) = s_0$ et pour $i \in [\![0, n-1]\!], D(0, i) = s_0$ également. En effet dans les deux cas on ne peut s'arrêter à aucune autre station que celle d'indice 0 . Il suffit donc d'évaluer la distance qui peut être parcourue avec le stock initial.

7. **Solution:** Pour $i \in [\![0, n-2]\!]$ et $k \in \mathbb{N}$:
   - Si $D(k,i) < d_{i+1}$, alors on ne peut pas atteindre la station $i+1$ en s'arrêtant au plus k fois aux stations précédentes. Ainsi, les $k+1$ arrêts ne peuvent se faire qu'à des stations d'indices $\leqslant i$. On en déduit $D(k+1, i+1) = D(k+1, i)$.
   - Sinon, on peut soit faire $k+1$ arrêts aux stations d'indices $\leqslant i$, soit faire k arrêts aux stations d'indices $\leqslant i$ et un arrêt à la station $i+1$.
   - On compare la distance qu'il est possible de parcourir dans ces deux cas pour garder la plus grande. On a bien $D(k+1, i+1) = \max\left(D(k+1, i), D(k, i) + s_{i+1}\right)$.

8. **Solution:**

```
def min_taille(dist, stock):
    n = len(dist)
    D = [[0]*n for i in range(n)]
    for k in range(n):
        D[k][0] = stock[0]
        D[0][k] = stock[0]
    for i in range(n-1):
        for k in range(i):
            if D[k][i] < dist[i]:
                D[k+1][i+1] = D[k+1][i]
            else:
                D[k+1][i+1] = max(D[k+1][i],D[k][i]+stock[i])
    for k in range(n):
        if D[k][n-2] >= dist[n-1]:
            return k
    return -1
```

9. **Solution:** La complexité est quadratique en n, car il y a une boucle sur k de i itérations imbriquées dans une boucle sur i de n itérations, donc $\sum_{i=0}^{n-1} i = O(n^2)$.

10. **Solution:**

Exercice n°2 Saut dans une liste *(CAPES 2023)*

Partie I - Programmes divers

Q1. Écrire une fonction `indice_min(li)` qui prend en argument une liste d'entiers `li` et renvoie l'indice d'un de ses minimums.

Q2. Que renverra `indice_min([1,0,2,0])` avec votre programme ?

Q3. Écrire une fonction `lettre_majoritaire(ch)` qui prend en argument une chaîne de caractères non vide et renvoie le caractère qui apparait le plus fréquemment. Ainsi, `lettre_majoritaire('abcdedde')` devrait renvoyer `'d'`. L'utilisation efficace d'un dictionnaire sera valorisée et on pourra utiliser l'opérateur `in`.

Q4. (a) Écrire une fonction itérative `fibonacci(n)` qui prend en argument un entier `n` supérieur ou égal à 2 et renvoie la valeur du `n`-ième terme de la suite de Fibonacci $(F_n)_{n\in\mathbb{N}}$ définie par $F_0 = 0$ et $F_1 = 1$ et $\forall n \geqslant 2, F_n = F_{n-1} + F_{n-2}$ (chaque terme est la somme des deux précédents).

(b) On donne ci-contre une fonction récursive répondant à la question **Q4.**a mais de façon récursive.

Expliquer pourquoi elle est moins pertinente que sa version itérative.

```
1 def fib(n:int)->int:
2     if n<=1:
3         return n
4     return fib(n-1)+fib(n-2)
```

Proposer alors une modification de cette fonction pour rendre celle-ci plus efficace en introduisant une mémoïsation. On définira la nouvelle fonction récursive `fib_memo(n:int,memo:{int:int})` avec la variable `memo`, dictionnaire initialisé en dehors de la fonction.

Partie II - Saut de valeur maximale

A. Introduction

Dans une liste de flottants `li`, on appelle saut un couple (i, j) avec $0 \leqslant$ `i` $\leqslant$ `j` $<$ `len(li)`, et la valeur d'un saut est la valeur `li[j]-li[i]`. On va ici programmer plusieurs manières de trouver un saut de valeur maximale dans une liste. Par exemple, dans la liste `[2.0,0.2,3.0,5.3,2.0]`, un tel saut est `(1,3)` (car `0.2` et `5.3` sont aux indices `1` et `3` respectivement).

Q5. Écrire une fonction `valeur(li,saut)` qui prend en argument une liste et un saut et renvoie la valeur du saut.

Q6. Donner un exemple de liste avec exactement deux sauts de valeur maximale et préciser ces sauts.

Q7. À l'aide d'un contre-exemple, montrer qu'on ne peut pas se contenter de chercher le minimum et le maximum d'une liste pour trouver un saut de valeur maximale.

Q8. Écrire une fonction `saut_max_naif(li)` qui renvoie un saut de valeur maximale en testant tous les couples `(i,j)` tels que $0 \leqslant$ `i` $\leqslant$ `j` $<$ `len(li)`.

B. Programmation dynamique

On décrit ici un algorithme utilisant le paradigme de la programmation dynamique pour résoudre ce problème : pour chaque `k` entre 1 et `len(li)`, on va calculer m_k l'indice du minimum de `li[0:k]`, et le couple (i_k, j_k) un saut de valeur maximale dans `li[0:k]`.

Ainsi, on aura $m_1 = i_1 = j_1 = 0$ car `li[0:1]` ne comporte qu'un seul élément.

Q9. Pour `k` $<$ `len(li)`, expliquer comment calculer efficacement m_{k+1} à partir de m_k et des valeurs dans `li`.

Q10. Justifier que la relation $(i_{k+1}, j_{k+1}) = \begin{cases} (i_k, j_k) \text{ si } \texttt{li}[k] - \texttt{li}[m_k] < \texttt{li}[j_k] - \texttt{li}[i_k] \\ (m_k, k) \text{ sinon} \end{cases}$ est correcte.

Q11. Écrire une fonction `saut_max_dynamique(li)` qui renvoie un saut de valeur maximale en utilisant la relation de la question **Q10.**.

Q12. Déterminer la complexité de votre programme dans le pire cas, puis comparer cette complexité avec celle du programme donné en question **Q8.**.

1. **Solution:**

```
def indice_min(li):
    imin=0
    for i in range(len(li)):
        if li[i]<li[imin]:
            imin=i
    return imin
```

2. **Solution:** La fonction renverra 1.
Rq : si vous avez mis `li[i]<=li[imin]`, le minimum est mis à jour quand on retombe sur 0, et renverra donc 3.

3. **Solution:**

```
def lettre_majoritaires(ch):
    assert len(ch)!=0
    # constitution du dictionnaire des occurrences
    d_occ={}
    for lettre in ch:
        if lettre in d_occ:
            d_occ[lettre]+=1
        else:
            d_occ[lettre]=1
    # recherche de la clé de valeur maximale
    vmax=0 # il y en a au moins 1
    for lettre in d_occ:
        if d_occ[lettre]>vmax:
            cmax=lettre  # clé du maximum
    return cmax
```

4. **Solution:**

a)

```
def fibonacci(n):
    f,g=0,1 # initialisation de F_{n-2} et F_{n-1}
    for i in range(2,n+1): # pour calculer jusqu'à n inclus, à partir de F_2
        f,g=g,f+g
    return g
```

b) La fonction itérative est de complexité linéaire.
La fonction récursive est de complexité exponentielle, car $C(n) = C(n-1)+C(n-2)$. Chaque valeur de F_n est recalculée même si elle l'a déjà été une fois précédente, car aucune valeur n'est stockée.

c)

```
memo={0:0,1:1}
def fib_memo(n,memo):
    if n in memo:
        return memo[n]
    else:
        return fib_memo(n-1,memo) + fib_memo(n-2,memo)
```

5. **Solution:**

```
def valeur(li,saut):
    return li[saut[1]]-li[saut[0]]
```

6. **Solution:** Un exemple de liste avec exactement deux sauts de valeur maximale est $li = [2, 0, 2, 2]$ de sauts $(1, 2)$ et $(1, 3)$.

7. **Solution:** Un contre-exemple possible est $li = [7, 5, 4, 2, 0]$: le maximum vaut 7 et le minimum vaut 0 mais le saut maximal est $(1, 2)$ de valeur -1 (logique vu que la liste est ordonnée par valeurs décroissantes.) ce qui montre qu'on ne peut pas se contenter de chercher le minimum et le maximum d'une liste pour trouver un saut de valeur maximale.

8. **Solution:**

```
def saut_max_naif(li):
    M=valeur(li,(0,1))
    saut=(0,1)
    n=len(li)
    for i in range(n):
        for j in range(i+1,n):
            if valeur(li,(i,j)) >M:
                    M=valeur(li,(i,j))
                    saut=(i,j)
    return saut
```

9. **Solution:** Tout dépend de si le dernier élément change la valeur du minimum m_k ou pas donc de la valeur de 1i[k] par rapport à m_k. Pour k < len(li), on a donc :

$$m_{k+1} = \begin{cases} m_k & \text{si li[k]} < m[k] \\ k & \text{sinon} \end{cases}$$

10. **Solution:**
Premier cas : La valeur du saut ne change pas en ajoutant le k-ième élément.
C'est à dire que $\text{li}[k] - li\,[m_k] < li\,[j_k] - li\,[i_k]$ et donc $1i[k] - li\,[m_k]$ est la valeur maximale que peut prendre le saut.
Second cas : La valeur du saut change et est nécessairement (m_k, k) alors.
La relation $(i_{k+1}, j_{k+1}) = \begin{cases} (i_k, j_k) & \text{si li } [k] - \text{li}\,[m_k] < \text{li}\,[j_k] - \text{li}\,[i_k] \\ (m_k, k) & \text{sinon} \end{cases}$ est donc correcte.

11. **Solution:**

```
def saut_max_dynamique(li):
    n=len(li)
    saut,m=(0,0),0
```

```
    for k in range(1,n):
        if not li[k]-li[m]<valeur(saut):
            saut=(m,k)
        if li[k]<li[m]:
            m=k
    return saut
```

12. **Solution:** La complexité du programme précédent dans le pire cas est linéaire en $\mathcal{O}$ (len(li)). (présence d'une simple boucle for contenant un nombre d'instructions en $\mathcal{O}(1)$.)
Celle du programme donné en question Q8. est quadratique en $\mathcal{O}\left(\text{len}(li)^2\right)$. (présence d'une double boucle for triangulaire contenant un nombre d'instructions en $\mathcal{O}(1)$.)

Exercice n°3 Justification d'un paragraphe

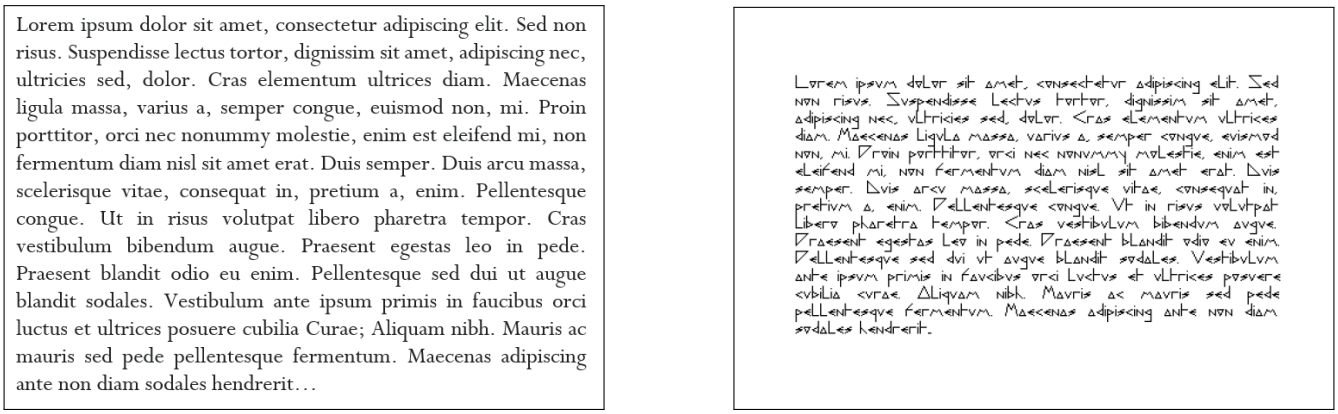


FIGURE 1 – Illustration de la justification de paragraphe pour différentes polices

Le paragraphe à justifier est constitué d'une liste de mots (chaînes de caractères). La difficulté est de placer des espaces entre les mots et de découper la liste en sous-listes pour que les lignes soient équilibrées (pas de ligne avec beaucoup d'espaces à la fin ou entre les mots par exemple). Pour simplifier le problème, on considère que le paragraphe est constitué d'une liste `lmots` d'entiers correspondant aux longueurs de chaque mot du paragraphe dans l'ordre d'apparition des mots (`lmots[i]` correspond au nombre de caractères du mot d'indice `i`). On note `L` le nombre de caractères et espaces que peut contenir une ligne au maximum. Il faut au minimum un espace entre deux mots d'une même ligne, on suppose que cet espace minimal correspond à un caractère.

On propose dans un premier temps l'algorithme glouton suivant :

```
1  def glouton(Lmots:[int],L:int)->[[int]]:
2      lignes=[]
3      nligne=[]
4      l=0
5      for c in Lmots :
6          if (c + l) > L:
7              lignes.append(nligne)
8              nligne=[c]
9              l=c+1
10         else:
11             l=l+c+1
12             nligne.append(c)
13     lignes.append(nligne)
14     return lignes
```

❑ **Q21** Expliquer en une ou deux phrases le principe de l'algorithme et pourquoi il est dit glouton.

Cet algorithme fournit une solution mais qui n'est pas nécessairement optimale. Si on teste cet algorithme sur le paragraphe suivant extrait du lorem ipsum : `ut enim ad minima veniam` pour une longueur de ligne maximale `L=10`, le résultat obtenu est le découpage noté a). Si on utilise une méthode de programmation dynamique, on obtient le découpage noté b).

a) Découpage obtenu par l'algorithme glouton

```
ut enim ad
minima
veniam
```

b) Découpage obtenu par programmation dynamique

```
ut enim
ad minima
veniam
```

Pour évaluer la pertinence du placement d'espaces et de retours à la ligne, on définit une fonction coût à minimiser pour que la répartition soit la plus harmonieuse possible.

Cette fonction coût correspond au nombre d'espaces disponibles sur une ligne élevé au carré, si on commence la ligne du mot i au mot j inclus sur cette même ligne :

$$cout(i,j) = (L - (j-i) - \sum_{k=i}^{j} lmots[k])^2$$

Cette fonction prend la valeur ∞ lorsque la somme des longueurs des mots de i à j est supérieure à L (ce qui signifie qu'on ne peut pas placer les mots i à j sur une même ligne).

La fonction python suivante correspond à l'implémentation de cette fonction coût.

```
def cout(i:int,j:int,lmots:[int],L:int)->int:
    res=sum(lmots[i:j+1])+(j-i)
    if res>L:
        return float("inf")
    else:
        return (L-res)**2
```

❏ **Q22** Évaluer pour les deux découpages a) et b) de l'exemple, ce que renvoie la fonction coût pour chacune des lignes en précisant les indices i et j pour chaque ligne (`lmots=[2,4,2,6,6]`). Conclure sur l'algorithme qui donne la solution la plus harmonieuse en sommant les différents coûts par ligne pour le découpage a) puis pour le découpage b).

La méthode de programmation dynamique consiste dans un premier temps à déterminer une équation de récurrence (équation de Bellman). Le problème peut être reformulé de la manière suivante :

Si on suppose connue la solution pour placer les mots jusqu'à un indice `i`*, le nouveau problème consiste à placer correctement les changements de lignes du mot* `i` *jusqu'à la fin. La question est donc de savoir où placer le changement de ligne à partir du mot d'indice* i*, de manière à minimiser la fonction coût.*

On note $d(i)$ le problème du placement optimal de changement de ligne jusqu'à l'indice `i`. L'équation de Bellman correspondante est alors :

$$d(i) = \min_{i<j\leq n} (d(j) + cout(i, j-1))$$

Un algorithme récursif naïf correspondant à la résolution de ce problème est le suivant.

```
def algo_recursif(i:int,lmots:[int],L:int)->int:
    if i==len(lmots):
        return 0
    else:
        mini=float("inf")
        for j in range(i+1,len(lmots)+1):
            d=algo_recursif(j,lmots,L)+cout(i,j-1,lmots,L)
            if d<mini:
                mini=d
        return mini
```

❏ **Q23** Proposer une modification de la fonction `algo_recursif` pour rendre celle-ci plus efficace en introduisant une mémoïsation.
On définira la nouvelle fonction récursive `progd_memo(i:int,lmots:[int],L:int,memo:{int:int})` avec la variable `memo`, dictionnaire initialisé en dehors de la fonction par `memo={len(m):0}`

On donne finalement une fonction utilisant la méthode de calcul de bas en haut. Cette fonction renvoie le coût optimal au problème de découpage de texte global de manière équivalente à la fonction `progd_memo`.

```
def progd_bashaut(lmots:[int],L:int)->int:
    M=[0]*(len(lmots)+1)
    for i in range(len(lmots)-1,-1,-1):
        mini,indi=float("inf"),-1
        for j in range(i+1,len(lmots)+1):
            d=M[j]+cout(i,j-1,lmots,L)
            if d<mini:
                mini,indi=d,j
        M[i]=mini
    return M[0]
```

❏ **Q24** Analyser, en fonction de n nombres de mots, la complexité temporelle asymptotique associée à l'algorithme récursif naïf (fonction `algo_recursif`) ainsi qu'à l'algorithme de programmation dynamique de bas en haut (fonction `progd_bashaut`) et conclure sur l'intérêt de l'algorithme de programmation dynamique. On ne comptera que les opérations de type additions/soustractions. Il n'est pas attendu de développements mathématiques poussés, les résultats peuvent être donnés sans justification.

On modifie légèrement la fonction `progd_bashaut` en ajoutant un argument `t` en plus des variables précédentes pour extraire les valeurs d'indices de découpe de lignes.

```
def progd_bashaut(lmots:[int],L:int,t:[int])->int:
    M=[0]*(len(lmots)+1)
    for i in range(len(lmots)-1,-1,-1):
        mini,indi=float("inf"),-1
        for j in range(i+1,len(lmots)+1):
            d=M[j]+cout(i,j-1,lmots,L)
            if d<mini:
                mini,indi=d,j
        t[i]=indi
        M[i]=mini
    return M[-1]
```

`t[i]` est la valeur de l'indice dans la liste `lmots` correspondant au placement optimisé sur une même ligne des mots d'indice `i` jusqu'à `t[i]` exclus. Cette liste `t` est modifiée en place dans la fonction. Pour l'exemple étudié précédemment (cas b)), on a ainsi obtenu la liste : `t=[2, 3, 4, 4, 5]`.

On dispose de la liste des mots (chaînes de caractères cette fois-ci) notée `mots`.

La fonction `lignes(mots:[str],t:[int],L:int)->[[str]]` doit renvoyer une liste de listes de mots (chaque sous-liste correspond à une ligne) en fonction de la liste `t` donnée par l'algorithme. La fonction `lignes(["Ut","enim","ad","minima","veniam"],[2,3,4,4,5],10)` renvoie :

`[["Ut","enim"],["ad","minima"],["veniam"]]`.

De même, en prenant,

```
t=[2, 3, 5, 5, 5, 6, 7, 9, 11, 11, 11]
L=15
mots=["Lorem","ipsum","dolor","sit","amet,","consectetur","adipiscing",
"elit.","Sed","non","risus."]
```

la fonction `lignes(mots,t,L)` renvoie :

```
[["Lorem","ipsum"],
["dolor","sit","amet,"],
["consectetur"],
["adipiscing"],
["elit.","Sed"],
["non","risus."]]
```

❏ **Q25** Proposer une implémentation de cette fonction `lignes(mots:[str],t:[int],L:int)`

Il reste à écrire une fonction `formatage(lignesdemots:[[str]],L:int)` qui renvoie une chaîne de caractères correspondant à la justification du paragraphe à partir des listes de mots par ligne `lignesdemots` et de la longueur maximale `L` d'une ligne en termes de caractères et espaces. Les retours à la ligne seront représentés par le symbole `"\n"`. Les espaces devront être répartis équitablement entre les mots pour que la justification se fasse bien entre la marge gauche et la marge droite (en respectant la longueur L maximale imposée). On obtient par exemple pour `L=10` :

```
ut    enim
ad  minima
veniam
```

❏ **Q26** Proposer une implémentation de cette fonction `formatage(lignesdemots:[[str]],L:int)->str` qui renvoie la chaîne de caractères justifiée.

Solution:

R1. ★ Le principe de l'algorithme est de remplir au maximum chaque ligne avec des mots complets. Il s'agit d'un algorithme glouton car à chaque étape on fait le choix optimal pour obtenir à la fin une bonne

solution, que l'on espère optimale.

R2. ★

— découpage a)

— 1re ligne : $i = 0$ et $j = 2$

$$\begin{aligned} cout(0,2) &= \left(10 - (2-0) - \sum_{k=0}^{2} lmots[k]\right)^2 \\ &= \left(8 - (2+4+2)\right)^2 \\ &= 0 \end{aligned}$$

— 2e ligne : $i = 3$ et $j = 3$

$$\begin{aligned} cout(3,3) &= \left(10 - (3-3) - \sum_{k=3}^{3} lmots[k]\right)^2 \\ &= \left(10 - 6\right)^2 \\ &= 16 \end{aligned}$$

— 3e ligne : $i = 4$ et $j = 4$

$$\begin{aligned} cout(4,4) &= \left(10 - (4-4) - \sum_{k=4}^{4} lmots[k]\right)^2 \\ &= \left(10 - 6\right)^2 \\ &= 16 \end{aligned}$$

Coût total(a)=32

— découpage b)

— 1re ligne : $i = 0$ et $j = 1$

$$\begin{aligned} cout(0,2) &= \left(10 - (2-1) - \sum_{k=0}^{1} lmots[k]\right)^2 \\ &= \left(9 - (2+4)\right)^2 \\ &= 9 \end{aligned}$$

— 2e ligne : $i = 2$ et $j = 3$

$$\begin{aligned} cout(3,3) &= \left(10 - (3-2) - \sum_{k=2}^{3} lmots[k]\right)^2 \\ &= \left(9 - (2+6)\right)^2 \\ &= 1 \end{aligned}$$

— 3e ligne : $i = 4$ et $j = 4$

$$\begin{aligned} cout(4,4) &= \left(10 - (4-4) - \sum_{k=4}^{4} lmots[k]\right)^2 \\ &= \left(10 - 6\right)^2 \\ &= 16 \end{aligned}$$

Coût total(b)=26

Le découpage b) donne bien la solution la plus harmonieuse.

R3. ★

```
def prgd_memo(i,lmots,L,memo):
    if i in memo:
        return memo[i]
    else:
        mini=float("inf")
        for j in range(i+1,len(lmots)+1);:
            d=prgd_memo(j,lmots,L,memo)+cout(i,j-1,lmots,L)
            if d<mini:
                mini=d
        memo[i]=mini # on mémoïse !
        return memo[i]
```

R4. ★ ***Une justification rapide est nécessaire.***

L'algorithme récursif naïf est de complexité exponentielle, en effet à chaque appel récursif il est obligé de recalculer toutes les valeurs de d.

L'algorithme de programmation dynamique de bas en haut présente deux boucles for imbriquées d'au maximum n (n est le nombre de mots, soit la longueur de `lmots`) itérations, avec à l'intérieur l'appel de la fonction `cout` qui effectue la somme des longueurs des mots (au maximum n). La complexité est donc cubique, en $O(n^3)$.

R5.

```
def lignes(mots,t,L):
    i = 0  # i est indice qui va parcourir t de manière non régulière
    Lres= []
    while i<len(t) :
        Lres.append(mots[i:t[i]]) # une phrase commençant à i fini à t[i]-1 inclus
        i = t[i] # on décale l'indice de départ
    return Lres
```

R6.

```
def formatage(lignesdemots,L):
    texteJ = ""
    for lig in lignesdemots :
        lmots= 0 # calcul la longueur de tous les mots de la ligne
        for mot in lig :
            lmots += len(mot)
        nbEspace = L - lmots  # le nombre d'espaces à insérés
        if len(lig) == 1 :
            ligneTexte = lig[0] + " "*nbEspace + "\n"
        else :
            nbEspaceMini = nbEspace//(len(lig)-1) # le nombre minimum d'espace à placer à chaque fois
            espaceSup = nbEspace%(len(lig)-1) # le rab d'espaces
            # on construit un liste contenant le nombre d'espace à insérer entre les mots
            Lespaces = [nbEspaceMini+1]*espaceSup + [nbEspaceMini]*(len(lig)-1-espaceSup)
            # construciton de la ligne
            ligneTexte = ""
```

```
            for i in range(len(lig)-1) :
                ligneTexte += lig[i]+" "*Lespaces[i]
            ligneTexte += lig[len(lig)-1] + "\n"
        # completion du texte
        texteJ += ligneTexte
    return texteJ
```

IV IA

Exercice n°1 TD n°3

Reprendre / faire / finir l'exercice sur Harry Potter.

Exercice n°2 Reconnaissance optique de caractères *(Extrait de CCINP 2023)*

Dans la suite du sujet, on suppose qu'on dispose d'une liste `fichiers_car_ref` contenant les noms des fichiers images d'un grand nombre de caractères ayant des fontes proches de celles du texte scanné. Le nom de chaque fichier est défini de la manière suivante :

`nomFonte + "_" + nomCatégorie+taillePolice + "_" + idSymbole + ".png"`

Les catégories sont définies par la liste : `categories = ["majuscules","minuscules","chiffres","special"`

Les symboles considérés sont définis par la liste :

`symboles = ["ABCDEFGHIJKLMNOPQRSTUVWXYZ","abcdefghijklmnopqrstuvwxyz", "0123456789", ". : , ; ' (!?) éèàçùûûâ"]`. On compte 79 symboles différents.

Toutes les images des caractères de référence sont lues et stockées sous forme de tableaux `array`. On définit un dictionnaire `carac_ref` dont les clés seront les symboles apparaissant dans la liste `symboles` (par exemple "A", "a", ...). À chaque clé sera associée une liste de tableaux `array` représentant des images.

La commande `img=imread(nomFichier)` permet de lire le fichier image `nomFichier` et de stocker le tableau `array` à deux dimensions qui représente l'image dans la variable `img`.

Un caractère à identifier est également stocké sous forme d'un tableau array nommé `carac_test`. On suppose que les dimensions de ce tableau et de tous les tableaux du dictionnaire `carac_ref` sont les mêmes.

La méthode d'identification utilisée est celle des K plus proches voisins. Elle consiste à calculer une distance entre l'image du caractère à identifier et toutes les images de référence. En notant (i, j) les coordonnées d'un pixel dans le tableau représentant l'image, p_{ij} le pixel associé à l'image du caractère à identifier et q_{ij} celui d'un caractère de référence, on calcule pour chaque caractère de référence la distance $d = \sqrt{\sum_{i,j} (p_{ij} - q_{ij})^2}$.

Les distances d sont stockées dans un dictionnaire `distances` où, pour chaque clé égale à un symbole de la liste `symboles`, on associe une liste de distances pour chaque image de référence de ce symbole.

R1. Écrire une fonction `distance(im1:array, im2:array)->float` qui calcule la distance entre les deux images `im1` et `im2` supposées de même dimension.

Solution:

```
def distance(im1,im2):
    n1,n2,r=dimensions(im1)
    d = 0
    for i in range(n1):
        for j in range(n2):
            d = d + (im1[i][j]-im2[i][j])**2
    return sqrt(d)
```

R2. Écrire une fonction `calcul_distances(carac_ref:dict, carac_test:array)->dict` qui prend en argument le dictionnaire des tableaux catégorisés et un tableau associé au caractère à tester et qui renvoie le dictionnaire des distances.

Solution:

```
def calcul_distances(carac_ref, carac_test):
    D = { c : [] for c in carac_ref} # dictionnaire des distances
    for c in carac_ref : # pour chaque clé
        for x in carac_ref[c] : # pour chaque image associée au
   caractère c
            d = distance(x,carac_test) # calcul de la distance
            D[c].append(d) # ajout à la liste associée à la clé c
```

```
    return D
def calcul_distances(carac_ref, carac_test):
    D = { } # dictionnaire des distances
    for c in carac_ref : # pour chaque clé
        D[c] = [] # liste vide associée à la clé c
        for x in carac_ref[c] : # pour chaque image associée au
   caractère c
            d = distance(x,carac_test) # calcul de la distance
            D[c].append(d) # ajout à la liste associée à la clé c
    return D
```

La suite consiste à déterminer les K plus petites distances et extraire les clés correspondantes, puis parmi ces clés déterminer la clé majoritaire. Une méthode envisageable est de trier les distances par ordre croissant pour prendre les K premiers éléments. On suppose qu'il y a au total n images de caractères de référence sur l'ensemble des symboles.

Une méthode plus efficace est envisagée pour extraire directement les K plus petits éléments. Elle consiste à construire par tri par insertion la liste de taille K. L'algorithme correspondant est donné ci-dessous.

R3. Écrire, sur votre copie, les trois lignes manquantes dans cet algorithme.

Solution: La fonction initialise la liste à renvoyer ; voisins, par une liste de couples contenant une valeur et un symbole, appelé lettre dans le code, la valeur étant la distance de l'image à l'une des représentations de la lettre. Les distances sont classiquement initialisées à l'infini, ce qui permet de faire rentrer dans la liste au départ n'importe quel élément. On parcourt ensuite les entrées du dictionnaire distances et, pour chacune de ces entrées, la liste des distances qu'elle contient. À chaque instant, la liste voisins contient les K plus proches voisins parmi ceux qui ont été déjà testés, liste complétée tant qu'on n'a pas encore testé K valeurs par des valeurs infinies. La plus grande valeur contenue dans la liste est ainsi celle d'indice $K-1$. Si l'élément lu correspond à une distance inférieure, on l'insère dans la liste sans échanger les éléments comme on le fait classiquement pour trier une liste en place, mais en décalant le dernier élément lu, ce qui permet de remplacer les valeurs devenues trop grandes.

Rq (personnelle) : il s'agit du tri par insertion, qui n'est plus explicitement au programme, en tous cas qui n'apparaît plus comme étant un attendu devant être maitrisé. Par conséquent, cette question posée ainsi me parait difficile...

```
def Kvoisins(distances :dict , K:int)->list:
    voisins = [(float("inf"),"") for k in range(K)]
    for lettre in distances:
        d = distances[lettre]
        for j in range( len(d) ):
            if voisins[K-1][0] > d[j] :
                k = len(voisins)-1
                while k>0 and voisins[k-1][0] > d[j] :
                    voisins[k] = voisins[k-1]
                    k = k - 1
                voisins[k] = [d[j], lettre]
    return voisins
```

R4. Préciser la complexité temporelle asymptotique dans le pire des cas de cet algorithme en fonction de n et de K.

Solution: Dans le pire des cas, on parcourt à chaque fois toute la liste des voisins. Les opérations effectuées étant de complexité constante, la complexité est en $\mathcal{O}(Kn)$.

R5. Écrire une fonction `symbole_majoritaire(voisins:list)->str` qui à partir de la liste voisins renvoyée par la fonction `Kvoisins` renvoie le symbole majoritaire.

Solution:

```
def symbole_majoritaire(voisins: list) -> str:
    Lettres = {}
    for entree in voisins:
        lettre = entree[1]
        if lettre not in Lettres:
            Lettres[lettre] = 1
        else:
            Lettres[lettre] += 1
    record = 0
    for lettre in Lettres:
        if Lettres[lettre] > record:
            record, reference = Lettres[lettre], lettre
    return reference
```

On teste l'algorithme sur les caractères extraits dans la partie précédente (`"Beauté,"`). On obtient les résultats suivants.

Nombre de voisins K	Type d'éléments dans la base de données	Nombre d'éléments dans la base n	Caractères obtenus
1	fonte similaire au texte analysé	79 images correspondant aux 79 symboles	"Bssi !-,"
4	fonte similaire au texte analysé	79 images correspondant aux 79 symboles	"Bssi !-,"
1	40 fontes proches de celle du texte analysé	40*79 images correspondant aux 79 symboles	"Bsauté,"
4	40 fontes proches de celle du texte analysé	40*79 images correspondant aux 79 symboles	"Bsauté,"
1	40 fontes pour 8 polices différentes	320*79 images correspondant aux 79 symboles	"Beauté,"
4	40 fontes pour 8 polices différentes	320*79 images correspondant aux 79 symboles	"Beauté,"

R6. Commenter les résultats obtenus.

Solution: Le nombre d'images de référence semble avoir un impact déterminant sur la qualité de la reconnaissance, ce qui n'est pas le cas du choix du nombre K de voisins.

V Théorie des jeux

Reprendre les exos non terminés de la feuille de TD n°5.