# ? Informatique Tronc Commun
# TD n°6 Révisions

## 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.

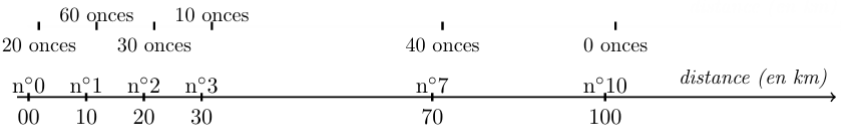
Pour $i \in [\![0, n-1]\!]$ et $k \in \mathbb{N}$, on pose $D(k,i)$ la distance maximale qu'on peut parcourir en s'arrêtant à au plus $k$ stations parmi les stations d'indices compris entre 1 et $i$.

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:**

# II Base de données

On se donne une base de données contenant les tables suivantes :

| stations | |
|---|---|
| id | entier |
| nom | châne |
| latitude | flottant |
| longitude | flottant |
| ouverture | date |

| carburants | |
|---|---|
| station | entier |
| nom | chaîne |
| type | chaîne |
| prix | flottant |

| evolutions | |
|---|---|
| station | entier |
| nom | chaîne |
| evol | flottant |
| date | date |

| utilisations | |
|---|---|
| station | entier |
| immatriculation | chaîne |
| carburant | chaîne |
| quantite | flottant |
| date | date |

| routes | |
|---|---|
| station1 | entier |
| station2 | entier |
| distance | flottant |

La table `stations` décrit les différentes stations services et contient le nom, les coordonnées et la date d'ouverture.

| | | | | |
|---|---|---|---|---|
| 42 | 'CarbuH24 la montagne' | 48.848 | 2.345 | 2008-11-15 |

La table `carburants` décrit les carburants disponibles dans chaque station service, leurs noms, le type de véhicule associé à chaque carburant et le prix du carburant en euros par litre.

| | | | |
|---|---|---|---|
| 42 | 'Gazole' | 'Diesel' | 1.237 |

La table `evolutions` décrit l'évolution des prix de chaque carburant dans chaque station, avec le différentiel et la date de modification.

| | | | |
|---|---|---|---|
| 42 | 'Gazole' | -0.018 | $2010-12-04$ |

La table `utilisations` décrit l'utilisation des pompes de chaque station, avec l'immatriculation des véhicules des utilisateurs, le nom et la quantité de carburant acheté en litres et la date.

| | | | | |
|---|---|---|---|---|
| 42 | '22-MPX-23' | 'Gazole' | 35.7 | $2021-02-15$ |

Enfin, la table `routes` décrit l'existence de routes entre deux stations et contient la distance. On supposera la table symétrique, c'est-à-dire que pour chaque entrée $(s_1, s_2, d)$, il existe une entrée $(s_2, s_1, d)$. Chaque paire de station ne possède pas nécessairement une route les reliant.

| | | |
|---|---|---|
| 42 | 78 | 37.3 |

11. La table `carburants` contient des données redondantes, lesquelles ? Comment modifier la base de données pour éviter ce problème ?
12. Écrire une requête SQL renvoyant la liste des carburants disponibles à la station d'identifiant `42`.
13. Écrire une requête SQL renvoyant les noms des stations possédant du carburant `'Gazole'`, triés par latitude décroissante.
14. Écrire une requête SQL renvoyant le prix du carburant `'Gazole'` à chaque station le 30 septembre 2022.
15. Écrire une requête SQL renvoyant les immatriculations des voitures qui ont déjà fait le plein avec deux types de carburant différents.

11. **Solution:** Le type de véhicule auquel est associé un carburant est toujours le même pour chaque carburant. Au lieu d'une seule table carburants, on aurait pu utiliser deux tables :
    — l'une contenant les informations sur chaque carburant, sans se soucier des stations ;
    — une autre indiquant quel carburant est disponible dans quelle station, avec un identifiant de carbu-

rant pour faire le lien avec la table précédente.

12. **Solution:**

```sql
SELECT nom
FROM carburants
WHERE station=42
```

13. **Solution:**

```sql
SELECT S.nom
FROM stations AS S
JOIN carburants AS
ON id=station
WHERE C.nom="Gazole"
ORDER BY latitude DESC
```

14. **Solution:**

```sql
SELECT C.station, prix + SUM(evol)
FROM carburants AS C
JOIN evolutions AS E
ON C.station = E.station AND C.nom = E.nom
WHERE C.nom = 'Gazole' AND E.date <= '2022-09-30'
GROUP BY C.station
```

15. **Solution:**

```sql
SELECT immatriculation
FROM utilisations AS U
JOIN carburants AS C
ON carburant = C.nom
AND U.station = C.station
GROUP BY immatriculation
HAVING COUNT(DISTINCT type) > 1
```

# 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 $\mathcal{O}(1)$.)