Projet 2 : correction

Exercice 1 :

La solution la plus simple de ce problème est d'incrémenter régulièrement la quantité de fuel nécessaire, en fonction du poids du passager. On va d'abord récupérer en entrée la liste des poids des passagers. On itère ensuite sur cette liste, pour chaque personne, on ajoute 60 à la quantité de fuel nécessaire, mais nous devons tester si le poids en question est strictement supérieur à 90 ; si c'est le cas on doit ajouter 20 en plus à la quantité de fuel. On affiche finalement la quantité totale de fuel nécessaire. Cette solution s'effectue en temps linéaire, en fonction du nombre de passagers (soit le nombre de poids passés en entrée) :

Listing 1 – Exercice 1

```
def fuel_compute():
        nb = int(input())
        p = list(map(int, input().split()))
        fuel = 0
        for v in p:
                if v > 90:
                        fuel += 20
                        fuel += 60
        return fuel
print(fuel_compute())
```

Exercice 2 :

Pour résoudre ce problème, on initialise d'abord une chaîne de caractères vide. Le principe est d'ajouter au fur et à mesure les caractères du message initial dans la nouvelle chaîne, si et seulement si le caractère ne représente pas un caractère corrompu. Cependant, nous devons faire attention au caractère '*' qui représente le début d'une zone corrompue du message. On ne veut rien garder du texte initial contenu entre deux '*'. Pour cela nous utiliserons une variable supplémentaire contenant une valeur booléenne (vrai ou faux) qui indique si on se trouve ou non dans une zone corrompue.Cet algorithme s'effectue en temps linéaire, on parcourt simplement la chaîne de caractères une seule fois, en avançant caractère par caractère :

Listing 2 – Exercice 2

```
def remove_interference(s):
        fs = ""
        interference = False
        for c in s:
                if c == '*':
                        interference = not interference
                if c != '*' and c != '.' and not interference:
```

```
                        fs += c
        return fs

length = int(input())
text = input()
print(remove_interference(text))
```

Exercice 3 :

Le principe de résolution de ce problème est d'initialiser une chaîne de caractère avec aucun contenu. Ensuite, il de balayer tous les caractères et si il appartient à la chaîne de caractère '0123456789' alors on ajoute le caractère en question. Si à la suite d'un caractère numérique, on trouve un espace, alors on ajoute une virgule.

On crée alors une liste de caractère (lst)en prenant comme séparateur la virgule.

On a plus qu'à ajouter les termes en les convertissant en entiers.

Listing 3 – Exercice 3

```
def sommeCaractere(chaine:str)-> int:
        N=""
        for i in range(len(chaine)):
                if chaine[i] in '0123456789':
                        N=N+chaine[i]
                if chaine[i]== " " and chaine[i-1] in '0123456789':
                        N=N+","
        lst=N.split(',')

        somme=0
        for i in range(len(lst)):
                if lst[i]!='':
                        somme=somme+int(lst[i])
        return somme

chaine1=" This picture is an oil on canvas painting by Danish artist Anna Petersen between 1845 and 1910 year'"
print(sommeCaractere(chaine1))
```

Exercice 4 :

En résumant l'énoncé d'un point de vue strictement algorithmique, on nous demande de trouver le nombre maximal de points couverts par un segment de longueur M.La solution naïve et évidente consiste simplement à tester à partir de chaque point de début,jusqu'où on peut étendre ce segment et maintenir un compteur au fûr et à mesure. On répète cela pour chaque point et on conserve le maximum global. Cependant, la complexité en temps de cet algorithme est de l'ordre de O(N) ce qui ne passe pas les tests de performance où N et M peuvent aller jusqu'à 106 et 105 respectivement. Souvent, une bonne manière de se rendre compte de pourquoi notre algorithme est trop lent est de réaliser un exemple à la main. En utilisant notre idée

précédente, lorsqu'on étend notre segment le plus loin possible, on va recommencer le segment à zéro pour le prochain début alors qu'on voit qu'on peut au minimum atteindre la même fin de segment. L'idée pour améliorer notre algorithme est donc de ne faire qu'avancer au lieu de revenir inutilement en arrière à chaque étape.Complexité Pour trier les jours, nous pouvons utiliser la fonction de la bibliothèque standard, résultant en un tri en O(N log N). Le reste de l'algorithme ne constitue qu'un parcours des événements ce qui est de l'ordre de O(N). À première vue, on pourrait se dire qu'une boucle imbriquée nécessite un temps quadratique pour être complétée, mais si l'on réfléchit à la manière dont cette boucle fonctionne dans sa totalité (on parle plus précisément d'analyse amortie), on remarque qu'elle ne prend qu'un temps linéaire, car elle va dans le pire des cas parcourir tous les points (on ne revient jamais en arrière, on ne fait qu'avancer !).

Voici le code du document.

Listing 4 – Exercice 4

```
n, m = map(int, input().split())
jours = [int(j) for j in input().split()]
jours.sort()
nb_amis_max = 0
debut = 0
for fin in range(n):
        while debut < fin and jours[fin] - jours[debut] > m:
                debut += 1
        nb_amis_max = max(nb_amis_max, fin - debut + 1)
print(nb_amis_max)
```