Exercice 1 Implementation generale

1.1

Si k désigne le numéro d'un objet, L_objets[k][1] représente la valeur de l'objet k

1.2

```python
def valeur_choix(L_e:list[int],L_objets:list[tuple[int]])->int:
    """
    Calcule la valeur totale des objets choisis
    :param L_e: liste d'index d'objets choisis
    :param L_objets: liste d'objets
    :return: valeur totale
    """
    return sum(L_objets[k][1] for k in L_e)
```

1.3

```python
def poids_choix(L_e:list[int],L_objets:list[tuple[int]])->int:
    """
    Calcule le poids total des objets choisis
    :param L_e: liste d'index d'objets choisis
    :param L_objets: liste d'objets
    :return: poids total
    """
    return sum(L_objets[k][0] for k in L_e)
```

Exercice 2 Algorithme naïf

2.1

Il y a deux valeurs possibles pour chaque objet donc le nb de combinaison est de 2^n

2.2

$$2^{1000} \approx 10^{300}$$

2.3

Il y a un nombre beaucoup trop important d'étape, pour que le programme s'execute dans un temps raisonnable.

Exercice 3 Algorithme glouton

3.1

```
def ratio(o:tuple[int])->float:
    """
    Calcule le ratio valeur/poids d'un objet
    :param o: objet
    :return: ratio
    """
    return o[1]/o[0] if o[0] != 0 else 0
```

3.2

```
def tri(L_objets:list[tuple[int]])->list:
    """
    Trie la liste d'objets par ordre décroissant de ratio valeur/poids
    :param L_objets: liste d'objets
    :return: liste d'objets triée
    """
    def tri_insertion_decroissant(L: list[tuple[int]]) -> None:
        """
        Trie la liste L par ordre décroissant de ratio valeur/poids en utilisant le
tri par insertion.
        :param L: liste d'objets
        """
        for i in range(1, len(L)):
            key = L[i]
            j = i - 1
            while j >= 0 and ratio(L[j]) < ratio(key):
                L[j + 1] = L[j]
                j -= 1
            L[j + 1] = key
    L_triee = L_objets.copy()
    tri_insertion_decroissant(L_triee)
    return L_triee
```

3.3

```python
def  SaD_glouton(L_objets:list[tuple[int]],P:int)->tuple[list[int], int, int]:
    """
    Algorithme glouton pour le problème du sac à dos
    :param L_objets: liste d'objets
    :param P: poids maximum
    :return: liste d'index d'objets choisis, valeur totale, poids total
    """
    L_triee = tri(L_objets)
    L_e = []
    while True:
        if len(L_triee) == 0:
            break
        o = L_triee[0]
        if poids_choix(L_e + [o[2]],L_objets) <= P:
            L_e.append(o[2])
            L_triee.remove(o)
        else:
            break
    return L_e, valeur_choix(L_e,L_objets), poids_choix(L_e,L_objets)
```

3.4

La complexité de la fonction de tri est en n*log(n) alors que l'algo glouton est en n. Donc c'est la méthode de tri qui importe.

3.5

Oui car 1000*log(1000) = 7000 ce qui est gigantesquement moins que l'algo naïf.

3.6

([0, 1, 2], 29, 13) est le resultat de l'algorithme glouton pour L_objets2 = [(4,10,0),(5,11,1),(4,8,2),(15,30,3)] et P = 15. Prendre uniquement le dernier objet serait plus rentable.

Exercice 4 Algorithme optimisé : programmation dynamique

4.1

nl = len(L_objets) et nc = P + 1 la valeur maximale de l'objet est de 2^64 = 10^20

4.2 et 4.3

```python
def SaD_pDyn(L_objets:list[tuple[int]],P:int)->tuple[list[int],int ,int]:
    """
    Algorithme dynamique pour le problème du sac à dos
    :param L_objets: liste d'objets
    :param P: poids maximum
    :return: liste d'index d'objets choisis, valeur totale, poids total
    """
    #initialisation
    import numpy as np
    nl = len(L_objets)
    nc = P + 1
    V=np.zeros ((nl, nc) ,np.uint64)
    for i in range(len(V)):
        o=L_objets[i]
        for j in range(len(V[i])):
            if i==0: #cas de base
                if j>=o[0]:
                    V[i,j]=o[1]
            else:
                if o[0] > j or V[i-1,j] > V[i-1,j - o[0]] + o[1]:
                    V[i,j]=V[i-1,j]
                else:
                    V[i,j]=V[i-1,j - o[0]] + o[1]
    #reconstruction de la solution:
    L_e =[]
    j=P
    for i in range(len(V) -1,-1,-1):
        if i>0:
            if V[i,j] != V[i-1,j]:#on a pris l'objet i
                L_e.append(i)
                j -= L_objets[i][0]
        else:
            if  V[i,j] != 0: #on a pris l'objet 0
                L_e.append(i)
    return L_e ,valeur_choix(L_e ,L_objets),poids_choix(L_e ,L_objets)
```

4.4

La complexité est de n*P avec n le nombre d'élément et P le poids maximum

Exercice 5 Comparaison entre deux algorithmes

5.1

```
from random import randint, random #import des fonctions de randomisation
N=100 #nombre d'objets
L_objets2 =[] #liste d'objets
somme_p =0 #somme des poids
for i in range(N): #on crée N objets
    r=random()*0.2+5 #on contraint les objets a avoir un ration v/p similaire (entre 5
et 5.2 ici)
    p=randint(100 ,1000) #les poids sont variables
    v=int(r*p) #la valeur est proportionnelle au poids
    L_objets2.append ((p,v,i)) #on ajoute l'objet `a la liste
    somme_p +=p #on calcule la somme des poids
P=somme_p //3 #on se limite `a un tiers du poids total , ce qui enl`eve les cas
triviaux et oblige `a faire des bons choix.
print(SaD_glouton(L_objets2 ,P)) #affichage de la solution gloutonne
print(SaD_pDyn(L_objets2 ,P)) #affichage de la solution dynamique
print(P) #affichage du poids maximum maximum
```

5.2

```
import time
start_time = time.time()
result_glouton = SaD_glouton(L_objets2, P)
end_time = time.time()
print("Temps d'exécution (glouton):", end_time - start_time, "secondes")

start_time = time.time()
result_pDyn = SaD_pDyn(L_objets2, P)
end_time = time.time()
print("Temps d'exécution (dynamique):", end_time - start_time, "secondes")Sortie :
```

```
Temps d exécution (glouton): 0.0041925907135009766 secondes
Temps d exécution (dynamique): 2.573908805847168 secondes
```

l'algorithme glouton prend 613 fois moins de temps que l'algorithme dynamique.

Algo .py complet :

```
"""
    L_objets =[(4,2,0) ,(3,3,1) ,(2,5,2)]
    L_objets[k][0] : poids
    L_objets[k][1] : valeur
    L_objets[k][2] : index
"""

def valeur_choix(L_e:list[int],L_objets:list[tuple[int]])->int:
    """
    Calcule la valeur totale des objets choisis
    :param L_e: liste d'index d'objets choisis
    :param L_objets: liste d'objets
    :return: valeur totale
    """
    return sum(L_objets[k][1] for k in L_e)

def poids_choix(L_e:list[int],L_objets:list[tuple[int]])->int:
    """
    Calcule le poids total des objets choisis
    :param L_e: liste d'index d'objets choisis
    :param L_objets: liste d'objets
    :return: poids total
    """
    return sum(L_objets[k][0] for k in L_e)

def ratio(o:tuple[int])->float:
    """
    Calcule le ratio valeur/poids d'un objet
    :param o: objet
    :return: ratio
    """
    return o[1]/o[0] if o[0] != 0 else 0

def tri(L_objets:list[tuple[int]])->list:
    """
    Trie la liste d'objets par ordre décroissant de ratio valeur/poids
    :param L_objets: liste d'objets
    :return: liste d'objets triée
    """
    def tri_insertion_decroissant(L: list[tuple[int]]) -> None:
        """
        Trie la liste L par ordre décroissant de ratio valeur/poids en utilisant le
tri par insertion.
        :param L: liste d'objets
        """
        for i in range(1, len(L)):
            key = L[i]
            j = i - 1
            while j >= 0 and ratio(L[j]) < ratio(key):
                L[j + 1] = L[j]
```

```
                j -= 1
            L[j + 1] = key
    def tri_rapide(L:list[tuple[int]]) -> list:
        """
        Trie la liste d'objets par ordre décroissant de ratio valeur/poids
        en utilisant l'algorithme de tri rapide.
        :param L: liste d'objets
        :return: liste d'objets triée
        """
        if len(L) <= 1:
            return L
        else:
            pivot = L[0]
            pivot_ratio = ratio(pivot)
            # Partitionner les objets en fonction du ratio
            gauche = [obj for obj in L[1:] if ratio(obj) >= pivot_ratio]
            droite = [obj for obj in L[1:] if ratio(obj) < pivot_ratio]
            # Combiner les résultats triés
            return tri(gauche) + [pivot] + tri(droite)
    L_triee = L_objets.copy()
    tri_insertion_decroissant(L_triee) # ou tri_rapide(L_triee)
    return L_triee

def  SaD_glouton(L_objets:list[tuple[int]],P:int)->tuple[list[int], int, int]:
    """
    Algorithme glouton pour le problème du sac à dos
    :param L_objets: liste d'objets
    :param P: poids maximum
    :return: liste d'index d'objets choisis, valeur totale, poids total
    """
    L_triee = tri(L_objets)
    L_e = []
    while True:
        if len(L_triee) == 0:
            break
        o = L_triee[0]
        if poids_choix(L_e + [o[2]],L_objets) <= P:
            L_e.append(o[2])
            L_triee.remove(o)
        else:
            break
    return L_e, valeur_choix(L_e,L_objets), poids_choix(L_e,L_objets)

def SaD_pDyn(L_objets:list[tuple[int]],P:int)->tuple[list[int],int ,int]:
    """
    Algorithme dynamique pour le problème du sac à dos
    :param L_objets: liste d'objets
    :param P: poids maximum
    :return: liste d'index d'objets choisis, valeur totale, poids total
    """
    #initialisation
```

```
    import numpy as np
    nl = len(L_objets)
    nc = P + 1
    V=np.zeros ((nl, nc) ,np.uint64)
    for i in range(len(V)):
        o=L_objets[i]
        for j in range(len(V[i])):
            if i==0: #cas de base
                if j>=o[0]:
                    V[i,j]=o[1]
            else:
                if o[0] > j or V[i-1,j] > V[i-1,j - o[0]] + o[1]:
                    V[i,j]=V[i-1,j]
                else:
                    V[i,j]=V[i-1,j - o[0]] + o[1]
    #reconstruction de la solution:
    L_e =[]
    j=P
    for i in range(len(V) -1,-1,-1):
        if i>0:
            if V[i,j] != V[i-1,j]:#on a pris l'objet i
                L_e.append(i)
                j -= L_objets[i][0]
        else:
            if  V[i,j] != 0: #on a pris l'objet 0
                L_e.append(i)
    return L_e ,valeur_choix(L_e ,L_objets),poids_choix(L_e ,L_objets)

if __name__ == "__main__":
    from random import randint, random #import des fonctions de randomisation
    import time #import de la fonction time pour le chronométrage
    N=100 #nombre d'objets
    L_objets2 =[] #liste d'objets
    somme_p =0 #somme des poids
    for i in range(N): #on crée N objets
        r=random()*0.2+5 #on contraint les objets a avoir un ration v/p similaire
(entre 5 et 5.2 ici)
        p=randint(100 ,1000) #les poids sont variables
        v=int(r*p) #la valeur est proportionnelle au poids
        L_objets2.append ((p,v,i)) #on ajoute l'objet `a la liste
        somme_p +=p #on calcule la somme des poids
    P=somme_p //3 #on se limite `a un tiers du poids total , ce qui enl`eve les cas
triviaux et oblige `a faire des bons choix.
    print(f"poids max : {P}") #affichage du poids maximum
    # Chronométrer la fonction gloutonne
    start_time = time.time()
    result_glouton = SaD_glouton(L_objets2, P)
    end_time = time.time()
    print("Temps d'exécution (glouton):", end_time - start_time, "secondes")
    print(result_glouton)
```

```
    # Chronométrer la fonction dynamique
    start_time = time.time()
    result_pDyn = SaD_pDyn(L_objets2, P)
    end_time = time.time()
    print("Temps d'exécution (dynamique):", end_time - start_time, "secondes")
    print(result_pDyn)
```

Algo .cpp

```cpp
#include <iostream>
#include <vector>
#include <tuple>
#include <algorithm>
#include <cstdlib>
#include <ctime>
#include <chrono>

using namespace std;

typedef tuple<int, int, int> Objet; // poids, valeur, index

int valeur_choix(const vector<int>& L_e, const vector<Objet>& L_objets) {
    int total = 0;
    for (int k : L_e) total += get<1>(L_objets[k]);
    return total;
}

int poids_choix(const vector<int>& L_e, const vector<Objet>& L_objets) {
    int total = 0;
    for (int k : L_e) total += get<0>(L_objets[k]);
    return total;
}

float ratio(const Objet& o) {
    return get<0>(o) != 0 ? (float)get<1>(o) / get<0>(o) : 0;
}

vector<Objet> tri(const vector<Objet>& L_objets) {
    vector<Objet> L_triee = L_objets;
    sort(L_triee.begin(), L_triee.end(), [](const Objet& a, const Objet& b) {
        return ::ratio(a) > ::ratio(b);
    });
    return L_triee;
}

tuple<vector<int>, int, int> SaD_glouton(const vector<Objet>& L_objets, int P) {
    vector<Objet> L_triee = tri(L_objets);
    vector<int> L_e;

    for (const auto& o : L_triee) {
        int idx = get<2>(o);
        if (poids_choix(L_e, L_objets) + get<0>(o) <= P) {
            L_e.push_back(idx);
        }
    }

    return { L_e, valeur_choix(L_e, L_objets), poids_choix(L_e, L_objets) };
}
```

```cpp
tuple<vector<int>, int, int> SaD_pDyn(const vector<Objet>& L_objets, int P) {
    int n = L_objets.size();
    vector<vector<unsigned long long>> V(n, vector<unsigned long long>(P + 1, 0));

    for (int i = 0; i < n; ++i) {
        int poids = get<0>(L_objets[i]);
        int valeur = get<1>(L_objets[i]);
        for (int j = 0; j <= P; ++j) {
            if (i == 0) {
                if (j >= poids) V[i][j] = valeur;
            } else {
                if (poids > j || V[i - 1][j] > V[i - 1][j - poids] + valeur) {
                    V[i][j] = V[i - 1][j];
                } else {
                    V[i][j] = V[i - 1][j - poids] + valeur;
                }
            }
        }
    }

    // Reconstruction de la solution
    vector<int> L_e;
    int j = P;
    for (int i = n - 1; i >= 0; --i) {
        if (i > 0) {
            if (V[i][j] != V[i - 1][j]) {
                L_e.push_back(i);
                j -= get<0>(L_objets[i]);
            }
        } else {
            if (V[i][j] != 0) {
                L_e.push_back(i);
            }
        }
    }

    return { L_e, valeur_choix(L_e, L_objets), poids_choix(L_e, L_objets) };
}

int main() {
    srand((unsigned)time(0));
    int N = 100;
    vector<Objet> L_objets;
    int somme_p = 0;

    for (int i = 0; i < N; ++i) {
        float r = static_cast<float>(rand()) / RAND_MAX * 0.2f + 5.0f;
        int p = rand() % 901 + 100; // 100 à 1000
        int v = static_cast<int>(r * p);
        L_objets.emplace_back(p, v, i);
        somme_p += p;
```

```
    }

    int P = somme_p / 3;
    cout << "Poids max : " << P << endl;

    // Test glouton
    auto start1 = chrono::high_resolution_clock::now();
    auto [res_glouton, val_glouton, poids_glouton] = SaD_glouton(L_objets, P);
    auto end1 = chrono::high_resolution_clock::now();
    chrono::duration<double> elapsed1 = end1 - start1;

    cout << "Temps d'exécution (glouton): " << elapsed1.count() << " secondes" <<
endl;
    cout << "Valeur totale (glouton): " << val_glouton << ", Poids total: " <<
poids_glouton << endl;

    // Test dynamique
    auto start2 = chrono::high_resolution_clock::now();
    auto [res_dyn, val_dyn, poids_dyn] = SaD_pDyn(L_objets, P);
    auto end2 = chrono::high_resolution_clock::now();
    chrono::duration<double> elapsed2 = end2 - start2;

    cout << "Temps d'exécution (dynamique): " << elapsed2.count() << " secondes" <<
endl;
    cout << "Valeur totale (dynamique): " << val_dyn << ", Poids total: " << poids_dyn
<< endl;
    cin.get(); // Pause pour voir les résultats
    return 0;
}
```

Liens

Fichier python

Fichier C++

Fichier .exe

Drive

https://tokamac.com/drive?path=drive/etique_et_numerique