

```

"""
TP2 - 2026-2027
Rendu de monnaie (glouton, recursivite, programmation dynamique)
"""

#1)
#200 + 200 + 100 + 20 + 10 + 2

#2)

#Version while

def plus_grande_piece(lv,s):
    k=len(lv)-1
    while lv[k]>s: #on cherche dans la liste lv en décroissant
        k=k-1
    return lv[k]

#Version for avec return

def plus_grande_piece(lv,s):
    for k in range(len(lv)-1,-1,-1): #par cours des indices décroissants de len(lv)-1 à 0
        if lv[k]<=s: #on cherche dans la liste lv en décroissant
            return lv[k]

#3
#On cherche la plus grande piece tant que la somme n'est pas rendue
def monnaie_glouton(lv,s):
    nb_pieces=0 #nombre de pieces à renvoyer
    liste_pieces=[] #liste des pieces à renvoyer
    a_rendre=s #somme restant à rendre
    while a_rendre !=0:
        piece = plus_grande_piece(lv,a_rendre)
        a_rendre = a_rendre-piece
        nb_pieces+=1
        liste_pieces.append(piece)
    return nb_pieces,liste_pieces

#4)
#a_rendre suite strictement decroissante d'entiers naturels, donc l'algo termine

#5)
#Avec lv = [1,3,4] et s=4, l'algo renvoie les pièces (4,1,1).
#Ce qui n'est pas optimal : on peut renvoyer (3,3)

#6) Avec les données de la question 5,  $n_6 = 1 + \min(n_5, n_3, n_2)$ 

#7)

def monnaie_recursive(lv , s):
    if s==0:
        return 0
    else:
        #on calcule le minimum, avec l'algo classique de recherche du minimum
        mini=s #le nombre de pièce est nécessairement plus petit que la somme
        for p in lv:
            if p <=s:
                ns_p = monnaie_recursive(lv , s-p) #appel recursif pour le
                #calcul de n_{s-p}
                if ns_p <mini: #si record battu
                    mini = ns_p

```

```
return 1+mini
```

#8) Par récurrence forte sur s on prouve que la fonction renvoie bien quelque chose
#Propriété : P(s) : "fonctmonnaie_recursive(lv , s) renvoie quelque chose"
#Pour s=0 c'est OK, elle renvoie 0
#Soit s fixé, supposons que P(k) est vraie pour tout k<=s
#Au rang s+1, on effectue des appels récursifs sur s-p qui est bien inférieur à s
#donc P(s-p) est vraie soit monnaie_recursive(lv , s-p) renvoie qqch.
#Conclusions : P(s) vraie pour tout s.

#9)

```
def monnaie_recursive_bis(lv , s):  
    print(s)  
    if s==0:  
        return 0  
    else:  
        #on calcule le minimum  
        mini=s #le nombre de pièce est nécessairement plus petit que la somme  
        for p in lv:  
            if p <=s:  
                ns_p = monnaie_recursive_bis(lv , s-p)  
                if ns_p <mini:  
                    mini = ns_p  
        return 1+mini
```

#Des appels sont effectués plusieurs fois

#10)

```
def monnaie_memoisation(lv,s):  
  
    D = {} #dictionnaire contenant les couples (s,ns)  
  
    #Fonction récursive interne  
    def monnaie_rec(som):  
        if som in D: #Si n_som a déjà été calculée  
            return D[som]  
        else: #Si n_som n'a pas déjà été calculée  
            if som==0: #Cas de base  
                D[som]=0  
                return 0  
            else:  
                #on calcule le minimum  
                mini=som #le nombre de pièce est nécessairement plus petit que la somme  
                for p in lv:  
                    if p <=som:  
                        ns_p = monnaie_memoisation(som-p)  
                        if ns_p <mini:  
                            mini = ns_p  
                D[som]=1+mini  
                return D[som]  
    return D[s]
```

#11) Version itérative, les valeurs n_s sont stockés dans un tableau T à la case T[s]

```
def monnaie_dyn(lv, s):  
    T = [0 for k in range(s+1)] #on initialise le tableau à 0  
    for k in range(1, s+1):  
        #calcul d'un minimum  
        mini = k #le nombre de pièces ne peut excéder la somme k
```

```

    for p in lv:
        if p <= k:
            n = T[k-p]
            if n < mini: #record battu
                mini = n
            T[k] = mini+1
    return T[s]

#Complexité :
#une affectation avant la première boucle for : coût constant
#Une boucle for (for k...): s tours de boucle
#une affectation
#une boucle for (for p...) : N tours de boucles où N est la longueur de Lv
#Au pire : 7 opérations, au mieux 2 opérations

#Complexité : version longue
#Total au pire : C = 1 + somme pour k=1...s (1 + somme pour i=1...N de 7)
#
# = 1 + s + 7Ns = O(Ns)
#Total au pire : calcul similaire, C(S) = 1+s+2Ns = O(Ns)
#
#Version courte :
#Au pire/au mieux :
#O(1) + 1 + somme pour k=1...s (O(1) + somme pour i=1...N de O(1)) = O(Ns)

#Une deuxième version où la liste lv est parcourue par indice plutôt que par valeur

def monnaie_dyn(lv, s):
    T = [0 for k in range(s+1)] #on initialise un tableau à 0
    for k in range(1, s+1):
        #calcul d'un minimum
        mini = k #le nombre de pièces ne peut excéder la somme k
        for j in range(len(lv)):
            if lv[j] <= k:
                n = T[k-lv[j]]
                if n < mini: #reconrd battu
                    mini = n
            T[k] = mini+1
    return T[s]

#Exercice - Hachage par division

"""Hachage par division"""
#1.
"""même chose que present dans 1."""
def presCouple(l,c):
    for cl,val in l:
        if cl==c:
            return True
    return False

#2.
"""si c est présent, c'est forcément à l'indice c%m. """
def present(dic,c):
    m=len(dic)
    k=c%m
    return presCouple(dic[k],c)

#3.
def valCouple(l,c):
    for cl,val in l:
        if cl==c:
            return val
def valeur(dic,c):

```

```

    m=len(dic)
    k=c%m
    return valCouple(dic[k],c)
#4.
"""on additionne les longueurs des différentes listes"""
def nbreElts(dic):
    s=0
    for l in dic:
        s=s+len(l)
    return s
#5.
"""on ajoute le couple (c,v) à la bonne liste"""
def ajout(dic,c,v):
    m=len(dic)
    dic[k].append((c,v))

```