prog_dynamique_luminy.py

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-

from __future__ import print_function

########################
## Toblérone cutting ##
########################

exemple_prix = [1, 5, 8, 9, 10, 17, 17, 20, 24, 26]

## Solution brute force

def valeur(decoupage, tab_prix):
    taille = 1
    val = 0
    for x in decoupage:
        if x:
            val += tab_prix[taille - 1]
            taille = 1
        else:
            taille += 1
    return val + tab_prix[taille - 1]

def enumere(n):
    """Renvoie la liste des listes de booléens distinctes de longueur n."""
    if n == 0:
        return [[]]
    listes = enumere(n - 1)
    return [t + [True] for t in listes] + [t + [False] for t in listes]

def opt_brute_force(tab_prix):
    """Renvoie un couple (decoupage, valeur) optimal pour tab_prix."""
    n = len(tab_prix)
    val_max, decoupe_max = tab_prix[-1], [False] * (n - 1)
    for decoupe in enumere(n - 1):
        if valeur(decoupe, tab_prix) > val_max:
            val_max, decoupe_max = valeur(decoupe, tab_prix), decoupe
    return decoupe_max, val_max

##>>> opt_brute_force(exemple_prix)
##([False, False, False, False, False, True, False, True, False], 27)

# ou bien via la décomposition en binaire

def binaire(n, nb_chiffres):
    """n en binaire sur nb_chiffres booléens, moins significatif d'abord."""
    res = []
    while n != 0:
        res.append(n % 2 == 1)
        n = n // 2
    return res + [False] * (nb_chiffres - len(res))

def opt_brute_force_bis(tab_prix):
    n = len(tab_prix)
    decoupe_max = [False] * (n - 1)
    for k in range(1, 2**(n - 1) + 1):
        decoupe = binaire(k, n - 1)
        if valeur(decoupe, tab_prix) > valeur(decoupe_max, tab_prix):
            decoupe_max = decoupe
    return decoupe_max, valeur(decoupe_max, tab_prix)

# Version dynamique itérative

def val_opt_dyn(tab_prix):
    tab_aux = [0]
    for k in range(len(tab_prix)):
        y = max(tab_prix[i] + tab_aux[- 1 - i] for i in range(k + 1))
        tab_aux.append(y)
    return tab_aux[-1]


# Version récursive

## Naïve

def val_opt_rec(tab_prix, n):
    if n == 0:
        return 0
    return max(tab_prix[k - 1] + val_opt_rec(tab_prix, n - k)
               for k in range(1, n + 1))

## Dynamique récursive : mémoïsation

def val_opt_rec_mem(tab_prix):
        def val_aux(n):
                if cache[n] != -1:
                        return cache[n]
                res = max(tab_prix[k - 1] + val_aux(n - k)
                          for k in range(1, n + 1))
                cache[n] = res
                return res
        n = len(tab_prix)
        cache = [0] + [-1] * n
        return val_aux(n)

# Reconstruction d'une solution optimale

def decoupe_opt(tab_prix):
    tab_aux = [tab_prix[0]]
    for k in range(1, len(tab_prix)):
        v_opt = tab_prix[k]
        petit = k + 1
        for i in range(1, k):
            if tab_aux[i] + tab_aux[- i - 1] > v_opt:
                v_opt, petit = 0
        y = max(tab_aux[i] + tab_aux[- i - 1] for i in range(k))
        tab_aux.append(max(x, y))
    return tab_aux[-1]

# Sous-tableau équilibré

# 2 valeurs possibles : un tableau suffit, un dictionnaire est plus commode.

def plus_grand_equilibre(t):
    deltas = {0 : [0, 0]}
    d = 0
    for i, x in enumerate(t):
        if x:
            d += 1
        else:
            d -= 1
        if d in deltas:
            deltas[d][1] = i + 1
        else:
            deltas[d] = [i + 1, i + 1]
    deb, fin = 0, 0
    for i, j in deltas.values():
        if j - i > fin - deb:
            deb, fin = i, j
    return deb, fin, deltas


# 3 valeurs possibles : un dictionnaire

def plus_grand_equilibre_bis(t):
    deltas = {(0, 0) : [0, 0]}
```

prog_dynamique_luminy.py

```
    d = e = 0
    for i, x in enumerate(t):
        if x == 0:
          d += 1
        elif x == 1:
          d -= 1
          e += 1
        elif x == 2:
          e -= 1
        else:
          # Il serait bon de lever une exception
          return "problème de valeurs..."
        if (d, e) in deltas:
          deltas[(d, e)][1] = i + 1
        else:
          deltas[(d, e)] = [i + 1, i + 1]
    deb, fin = 0, 0
    for i, j in deltas.values():
        if j - i > fin - deb:
            deb, fin = i, j
    return deb, fin, deltas

# Je vous laisse adapter pour k valeurs...


# Syracuse

def prochain_syr(n):
    if n % 2 == 0:
        return n // 2
    return 3 * n + 1

def max_syr(n, borne):
    def syr_mem(n):
        if n not in cache:
            cache[n] = 1 + syr_mem(prochain_syr(n))
        return cache[n]
    cache = {1 : 0}
    return max(syr_mem(k) for k in range(1, n + 1))

def disque_syr(n):
    visites = set()
    courants = [1]
    for k in range(n):
        nouveaux = []
        for x in courants:
            if 2 * x not in visites:
                nouveaux.append(2 * x)
            if x % 6 == 4 and (x - 1) // 3 not in visites:
                nouveaux.append((x - 1) // 3)
        visites.update(courants)
        courants = nouveaux
    visites.update(courants)
    return len(visites)




##########################################
## Annexe : générer des arbres d'appels ##
##########################################

class arbre:
    def __init__(self, etiquette, fils = ()):
        self.etiquette = etiquette
        self.enfants = list(fils)

    def __str__(self):
        if self.enfants:
            str_enfants = ", ".join(str(fils) for fils in self.enfants)
            return "Noeud(" + str(self.etiquette) + ", " + str_enfants + ")"
        return "Feuille(" + str(self.etiquette) + ")"

    def ajoute_enfants(self, *fils):
        for x in fils:
            self.enfants.append(x)

    def ajoute_iterable(self, it):
        for x in it:
            self.ajoute_enfant(x)

    def est_feuille(self):
        return not self.enfants

    def to_latex(self):
        eti = str(self.etiquette)
        if self.est_feuille():
            return "\\TR{\\fbox{\\verb!" + eti + "!}}"
        noeud = "\\pstree{\\Toval{\\verb!" + eti + "!}}{\n"
        return noeud + " ".join(a.to_latex() for a in self.enfants) + "}"

def arbre_fibo(n):
    if n <= 1:
        return arbre('fib %i' % n)
    ga = arbre_fibo(n - 1)
    dr = arbre_fibo(n - 2)
    return arbre('fib %i' %n, (ga, dr))

def arbre_toblerone(tab_prix, n):
    if n == 0:
        return arbre('v(0)=0')
    return arbre('v(%i)=%i' % (n, val_opt_rec_bis(tab_prix, n)),
                 (arbre_toblerone(tab_prix, k) for k in range(n)))

def arbre_toblerone_sans_valeurs(n):
    if n == 0:
        return arbre('v(0)')
    return arbre('v(%i)' % n,
                 (arbre_toblerone(k) for k in range(n)))

def arbre_toblerone_memo(n, cache):
    if n in cache:
        return arbre('mem %i' % n)
    cache.add(n)
    return arbre('val %i' % n,
                 (arbre_toblerone_memo(k, cache) for k in range(1, n)))

def arbre_syr(n, cache):
    if n == 1:
        return arbre('syr 1')
    if n in cache:
        return arbre('mem %i' %n)
    cache.add(n)
    return arbre('syr %i' % n, (arbre_syr(prochain_syr(n), cache),))

def foret_max_syr(n):
    cache = set()
    return arbre('max\\_syr %i' % n, (arbre_syr(k, cache)
                 for k in range(1, n + 1)))
```