TP — Programmation dynamique

Ce TP est une collection de six exercices **indépendants**.

Consignes : on n'utilise que les fonctions et méthodes de l'annexe du programme (en particulier ni `min`, ni `max`, ni `sum`, ni `sorted`), les fonctions ne modifient pas les listes qu'elles reçoivent en argument, et les complexités sont données avec la notation O.

Exécuter d'abord la première cellule de code (le module `dessins.py` doit se trouver dans le même dossier que ce notebook). Elle définit `INFINI = float("inf")` et la fonction `minimum3(a, b, c)`, qui remplace `min` pour trois valeurs.

Sommaire

— 1. Plus grand carré de 1 dans une matrice (Q1 à Q6)
— 2. Distance de Levenshtein (Q7 à Q14)
— 3. Produit d'une chaîne de matrices (Q15 à Q21)
— 4. Sac à dos (Q22 à Q28)
— 5. Bonus : la plus longue descente (Q29 à Q31)
— 6. Bonus : le jeu des pièces (Q32 et Q33)

1. Plus grand carré de 1 dans une matrice

Une matrice de 0 et de 1 à n lignes et p colonnes est représentée par une liste `m` de n listes de p entiers. La case (x, y) est celle de la ligne x et de la colonne y, la case $(0, 0)$ étant en haut à gauche. Le **carré de côté k et de coin supérieur gauche** (x, y) est formé des cases $(x + a, y + b)$ pour $0 \leq a < k$ et $0 \leq b < k$.

On cherche la **taille** (le côté) du plus grand carré ne contenant que des 1. Pour la matrice `M` ci-dessous, c'est 3.

matrice 6 x 8, plus grand carré de 1 : à trouver

0	1	1	0	1	1	1	0
1	1	1	0	1	1	1	1
1	1	1	1	1	1	1	0
0	1	1	1	0	1	1	1
1	1	1	1	0	0	1	1
0	1	1	0	1	1	1	1

À la main : recherche exhaustive

Question 1. Écrire une fonction `est_carre_de_uns(m, x, y, k)` qui prend en arguments une matrice `m`, les coordonnées `(x, y)` d'une case et un entier `k` ≥ 1, tels que le carré de côté `k` et de coin supérieur gauche `(x, y)` soit inclus dans la matrice, et renvoie `True` si ce carré ne contient que des 1, `False` sinon. Quelle est sa complexité ?

La fonction suivante essaie tous les carrés, par côté croissant.

```
def plus_grand_carre_naif(m):
    n, p = len(m), len(m[0])
    k = 0                          # il existe un carré de 1 de côté k
    trouve = True
    while trouve and k < n and k < p:
        trouve = False             # cherche un carré de 1 de côté k + 1
        for x in range(n - k):
            for y in range(p - k):
                if est_carre_de_uns(m, x, y, k + 1):
                    trouve = True
        if trouve:
            k = k + 1
    return k

print(plus_grand_carre_naif([[0, 1, 1], [1, 1, 1], [1, 1, 1]]))
```

Question 2. Quelle est la complexité de `plus_grand_carre_naif` pour une matrice de taille $n \times n$?

Proposer des améliorations possibles de la fonction `plus_grand_carre_naif`.

Sous-problèmes et récurrence

Pour chaque case (x, y), on note $C[x][y]$ la taille du plus grand carré de 1 dont (x, y) est le **coin inférieur droit** (et $C[x][y] = 0$ si $m[x][y] = 0$). La réponse est la plus grande valeur du tableau C : il y a un sous-problème par case.

Question 3. Compléter à la main le tableau `C` de la matrice `P` ci-dessous (le petit chiffre en haut à gauche de chaque case est le coefficient de `P`, les cases bleues sont celles qui contiennent un 1).

14 case(s) de C à compléter

0	1	1	1	1
1	1	2	1	1
1	1	1	0	0
1	1	1	1	0
0	1	1	1	0

Question 4. On suppose que $m[x][y] = 1$. Que vaut $C[x][y]$ si $x = 0$ ou $y = 0$? Sinon, en s'aidant du tableau précédent, exprimer $C[x][y]$ en fonction de $C[x-1][y]$, $C[x][y-1]$ et $C[x-1][y-1]$. *Bonus* : démontrer cette relation par double inégalité, en s'appuyant sur un dessin.

Calcul de bas en haut

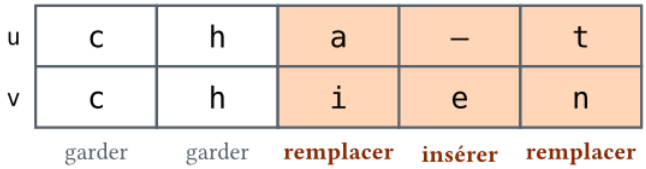
Question 5. Dans quel ordre peut-on remplir le tableau `C` ? Écrire une fonction `tableau_carres(m)` qui prend en argument une matrice `m` et renvoie le tableau `C` (une nouvelle liste de listes, de mêmes dimensions que `m`), puis une fonction `plus_grand_carre(m)` qui prend en argument une matrice `m` et renvoie la taille de son plus grand carré de 1. Quelle est leur complexité ?

Reconstruction

Question 6. Écrire une fonction `un_plus_grand_carre(m)` qui prend en argument une matrice `m` et renvoie un triplet `(x, y, k)`, où `k` est la taille du plus grand carré de 1 de `m` et `(x, y)` le coin **supérieur gauche** d'un tel carré (et `(0, 0, 0)` si `m` ne contient aucun 1).

2. Distance de Levenshtein

On transforme un mot u en un mot v par des **opérations** qui portent chacune sur une lettre : insérer une lettre, supprimer une lettre, ou remplacer une lettre par une autre (substitution). La **distance de Levenshtein** (ou distance d'édition) entre u et v est le nombre minimal d'opérations nécessaires. Par exemple, la distance entre `chat` et `chien` vaut 3.



On note n et m les longueurs de u et de v.

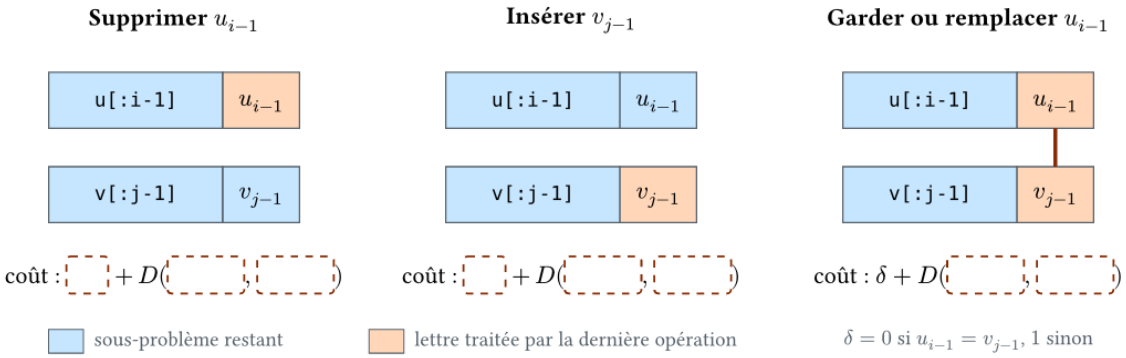
À la main : un algorithme glouton

Question 7. Un algorithme glouton compare les deux mots position par position : il remplace la lettre de u quand les deux lettres diffèrent, puis il supprime ou insère les lettres en trop à la fin. Combien d'opérations utilise-t-il pour transformer `chat` en `achat` ? Est-ce optimal ?

Sous-problèmes et récurrence

Pour $0 \leq i \leq n$ et $0 \leq j \leq m$, on note $D(i, j)$ la distance entre `u[:i]` (les i premières lettres de u) et `v[:j]` (les j premières lettres de v). La réponse est $D(n, m)$.

Question 8. Que valent $D(i, 0)$ et $D(0, j)$? Pour $i, j \geq 1$, une transformation de `u[:i]` en `v[:j]` se termine de l'une des trois façons du schéma ci-dessous. Compléter le schéma, puis en déduire la relation de récurrence vérifiée par D, en la justifiant brièvement.



Question 9. Pour `u = "copie"` (en lignes) et `v = "ecole"` (en colonnes), compléter à la main le tableau des $D(i, j)$. Dans quel ordre l'avez-vous rempli ?

20 case(s) à compléter

i\j	0	1 e	2 c	3 o	4 l	5 e
0	0	1	2	3	4	5
1 c	1	1	1	2	3	4
2 o	2					
3 p	3					
4 i	4					
5 e	5					

Mémoïsation

Question 10. Écrire une fonction `levenshtein_memo(u, v)` qui prend en arguments deux mots `u` et `v` et renvoie leur distance, calculée par une fonction récursive mémoïsée (avec un dictionnaire). Quelle profondeur la récursion peut-elle atteindre ?

Calcul de bas en haut

Question 11. Écrire une fonction `tableau_levenshtein(u, v)` qui prend en arguments deux mots `u` et `v` et renvoie le tableau `D` des $D(i, j)$ (une liste de listes), puis une fonction `distance_levenshtein(u, v)` qui renvoie la distance entre `u` et `v` (un entier). Quelles sont les dimensions du tableau ? Quelle est la complexité en temps et en mémoire ?

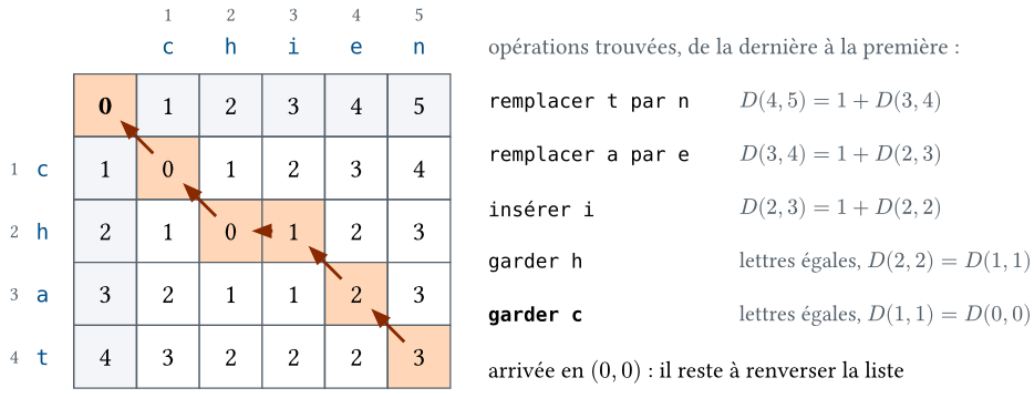
Reconstruction

On décrit une transformation par la liste de ses opérations, lue de gauche à droite : chaque opération est une chaîne de caractères de la forme `"garder c"`, `"remplacer a par i"`, `"insérer e"` ou `"supprimer y"`. Garder une lettre ne coûte rien. La figure du début de l'exercice correspond à la liste

```
["garder c", "garder h", "remplacer a par i", "insérer e", "remplacer t par n"].
```

La fonction `appliquer(u, ops)` du module `dessins` applique une telle liste au mot `u` et renvoie le mot obtenu.

Pour reconstruire une liste optimale, on remonte dans le tableau depuis la case (n, m) en cherchant, à chaque case, quel terme du minimum donne sa valeur : c'est ce que montre l'animation ci-dessous pour `chat` et `chien` (sur papier, seule sa dernière image est reproduite). Elle trouve une autre liste optimale que celle de la figure du début.



Question 12. Écrire une fonction `operations_levenshtein(u, v)` qui prend en arguments deux mots `u` et `v` et renvoie une liste optimale d'opérations transformant `u` en `v`, obtenue en remontant dans le tableau `D` depuis la case (n, m). Quelle est la complexité de la remontée ?

Bonus : mémoire, variante

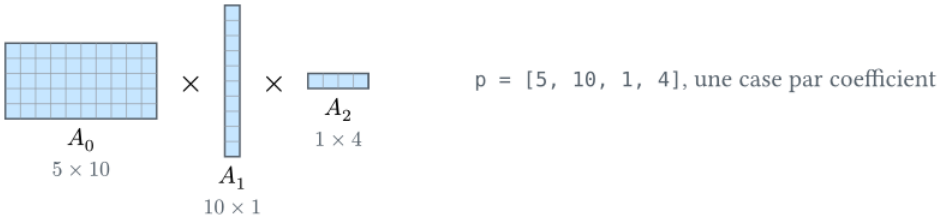
Question 13 *(bonus)*. Écrire une fonction `levenshtein_2lignes(u, v)` qui prend en arguments deux mots `u` et `v` et renvoie leur distance en n'utilisant qu'une mémoire $O(m)$. Que perd-on ?

Question 14 *(bonus)*. On n'autorise plus que l'insertion et la suppression (remplacer une lettre demande alors deux opérations). Que devient la relation de récurrence ? Écrire une fonction `distance_insertions_suppressions(u, v)` qui prend en arguments deux mots `u` et `v` et renvoie cette distance (5 pour `chat` et `chien`).

3. Produit d'une chaîne de matrices

Le produit d'une matrice de taille $a \times b$ par une matrice de taille $b \times c$ coûte abc multiplications de coefficients. On veut calculer le produit $A_0 A_1 \cdots A_{n-1}$ de $n \geq 1$ matrices, où A_i est de taille $p_i \times p_{i+1}$: les dimensions sont données par la liste `p` $= [p_0, p_1, \ldots, p_n]$, de longueur $n+1$.

Le produit matriciel est associatif : le résultat ne dépend pas du **parenthésage**, mais le nombre de multiplications, si. On cherche un parenthésage de coût minimal. On le décrit par une chaîne de caractères : `"((A0.A1).A2)"` ou `"(A0.(A1.A2))"` pour trois matrices, `"A0"` pour une seule.



À la main : un algorithme glouton

Question 15. Pour `p = [5, 10, 1, 4]`, calculer le coût des deux parenthésages `"((A0.A1).A2)"` et `"(A0.(A1.A2))"`. Lequel obtient l'algorithme glouton qui effectue à chaque étape le produit de deux matrices voisines le moins coûteux ?

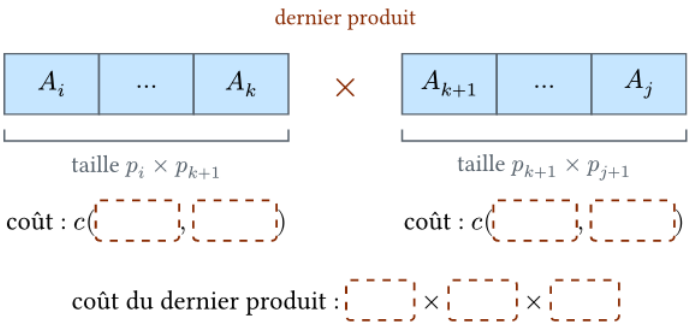
La fonction `cout_parenthesage(p, chaine)` du module `dessins` calcule le coût d'un parenthésage : on peut l'utiliser pour vérifier.

```
p = [5, 10, 1, 4]
print(cout_parenthesage(p, "((A0.A1).A2)"), cout_parenthesage(p, "(A0.(A1.A2))"))
```

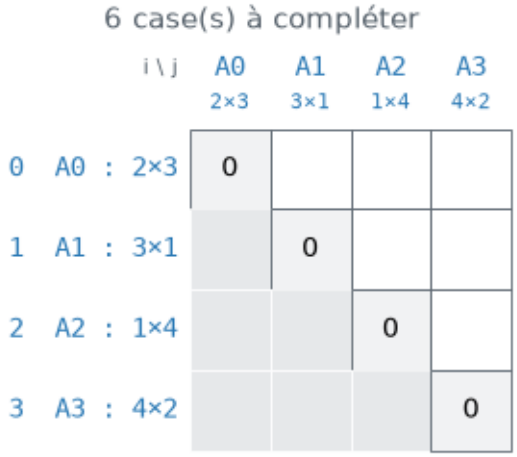
Sous-problèmes et récurrence

Le nombre de parenthésages croît exponentiellement avec n : les essayer tous est exclu. Pour $0 \leq i \leq j \leq n-1$, on note $c(i,j)$ le nombre minimal de multiplications pour calculer le produit $A_i \cdots A_j$. La réponse est $c(0, n-1)$.

Question 16. Que vaut $c(i,i)$? Pour $i < j$, le dernier produit effectué coupe la chaîne après une matrice A_k, avec $i \leq k < j$. Compléter le schéma, puis en déduire la relation de récurrence vérifiée par c. Pourquoi les produits $A_0 \cdots A_j$ (les préfixes) ne suffisent-ils pas comme sous-problèmes ?



Question 17. Compléter à la main le tableau des $c(i, j)$ pour `p = [2, 3, 1, 4, 2]` (seules les cases $i \leq j$ servent).

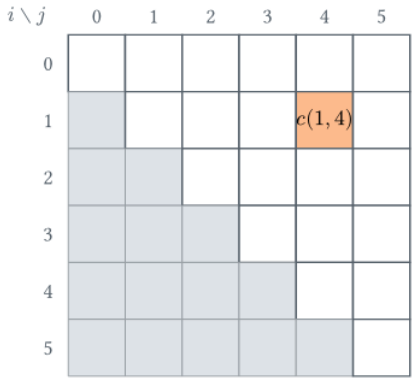


Mémoïsation

Question 18. Écrire une fonction `cout_chaine_memo(p)` qui prend en argument la liste `p` des dimensions et renvoie le coût minimal $c(0, n-1)$, calculé par une fonction récursive mémoïsée (avec un dictionnaire). Quelle est sa complexité en temps et en mémoire ?

Calcul de bas en haut

Question 19. La figure ci-dessous représente le tableau pour $n = 6$. Quelles cases faut-il lire pour calculer $c(1, 4)$? Peut-on remplir le tableau ligne par ligne, par i croissant ? Proposer un ordre de remplissage qui convient.



Question 20. Écrire une fonction `tableau_chaine(p)` qui prend en argument la liste `p` des dimensions et renvoie le tableau `c` des $c(i, j)$, une liste de n listes de longueur n dont seules les cases $i \leq j$ servent, puis une fonction `cout_chaine(p)` qui renvoie le coût minimal $c(0, n-1)$. Quelle est la complexité en temps et en mémoire ?

Bonus : reconstruction

Question 21 *(bonus)*. Écrire une fonction `parenthesage_optimal(p)` qui prend en argument la liste `p` des dimensions et renvoie un parenthésage optimal. Par exemple, `parenthesage_optimal([5, 10, 1, 4])` renvoie `"((A0.A1).A2)"`. On mémorisera pour chaque case un indice k qui donne le minimum, et on pourra utiliser `str(i)`.

4. Sac à dos

On dispose de n objets, numérotés de 0 à $n-1$: l'objet i a un poids p_i et un prix v_i, entiers strictement positifs. On veut remplir un sac de capacité b en prenant certains objets, chacun au plus une fois, de façon à maximiser le prix total emporté.

En Python, `objets` est la liste des couples `(poids, prix)`, par exemple `[(3, 4), (4, 5), (2, 3)]`.



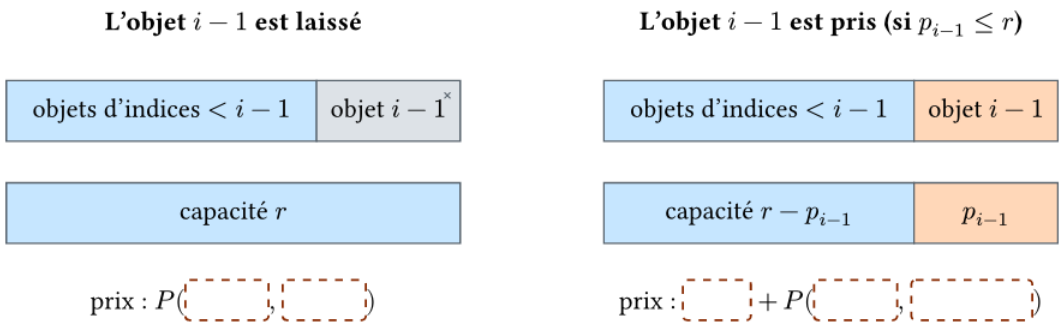
À la main : un algorithme glouton

Question 22. Un algorithme glouton parcourt les objets par rapport prix / poids décroissant et prend chaque objet qui tient encore dans le sac. L'appliquer à `objets = [(3, 4), (4, 5), (2, 3)]` et $b = 7$. Est-ce optimal ?

Sous-problèmes et récurrence

Pour $0 \leq i \leq n$ et $0 \leq r \leq b$, on note $P(i, r)$ le prix maximal réalisable avec les objets d'indices $< i$ et une capacité r. La réponse est $P(n, b)$.

Question 23. Que vaut $P(0, r)$? Pour $i \geq 1$, l'objet $i-1$ est laissé ou pris. Compléter le schéma, puis en déduire la relation de récurrence vérifiée par P. Démontrer que si une solution optimale pour (i, r) prend l'objet $i-1$, alors les autres objets qu'elle prend forment une solution optimale pour $(i-1, r-p_{i-1})$.



Question 24. Compléter à la main le tableau des $P(i, r)$ pour `objets = [(3, 4), (4, 5), (2, 3)]` et $b = 7$ (ligne i : objets d'indices $< i$, colonne r : capacité).

16 case(s) à compléter

i \ r	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1 objet 0 (3, 4)	0	0	0	4	4	4	4	4
2 objet 1 (4, 5)								
3 objet 2 (2, 3)								

Calcul de bas en haut

Question 25. Écrire une fonction `tableau_sac(objets, b)` qui prend en arguments la liste `objets` et la capacité `b`, et renvoie le tableau `P` des $P(i, r)$ (une liste de listes), puis une fonction `sac(objets, b)` qui renvoie le prix maximal $P(n, b)$. Quelle est la complexité en temps et en mémoire ?

Reconstruction

Question 26. Écrire une fonction `sac_objets(objets, b)` qui renvoie la liste des indices des objets d'une solution optimale, dans l'ordre croissant, en remontant dans le tableau `P` depuis la case (n, b). Par exemple, `sac_objets([(3, 4), (4, 5), (2, 3)], 7)` renvoie `[0, 1]`. Quelle est la complexité de la remontée ?

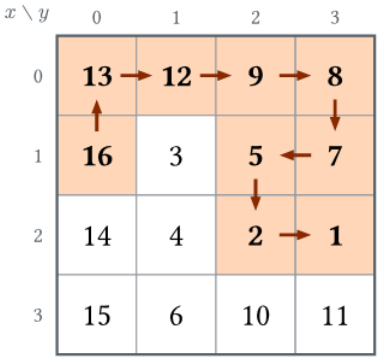
Bonus : mémoïsation, mémoire

Question 27 *(bonus)*. Écrire une fonction `sac_memo(objets, b)` qui renvoie $P(n, b)$, calculé par une fonction récursive mémoïsée (avec un dictionnaire). Quel avantage a-t-elle sur le tableau quand la capacité est grande et les objets peu nombreux ?

Question 28 *(bonus)*. Écrire une fonction `sac_une_liste(objets, b)` qui renvoie $P(n, b)$ en n'utilisant qu'une seule liste de longueur $b + 1$. Dans quel sens faut-il la parcourir ? Que perd-on ?

5. Bonus : la plus longue descente

Une grille `h` de n lignes et p colonnes (une liste de n listes de p entiers) donne l'altitude de chaque case d'un domaine skiable. Une **descente** est une suite de cases dans laquelle chaque case est voisine de la précédente (au-dessus, en dessous, à gauche ou à droite) et d'altitude **strictement plus petite**. Sa longueur est son nombre de cases. On cherche la longueur d'une plus longue descente.



une plus longue descente, de longueur 9 :

$16, 13, 12, 9, 8, 7, 5, 2, 1$

```
H = [[13, 12, 9, 8],
     [16, 3, 5, 7],
     [14, 4, 2, 1],
     [15, 6, 10, 11]]
```

Question 29. Un skieur glouton part de la case la plus haute de la grille `H` et rejoint à chaque étape la plus haute de ses voisines plus basses, pour perdre le moins d'altitude possible. Quelle descente obtient-il ? Est-elle de longueur maximale ?

Question 30. On note $L(x, y)$ la longueur d'une plus longue descente **partant** de la case (x, y). Exprimer $L(x, y)$ en fonction des valeurs de L sur certaines voisines de (x, y). Quelles cases la case $(0, 1)$ de `H` lit-elle ? Et la case $(3, 2)$? Peut-on remplir le tableau L ligne par ligne, de gauche à droite ? En le balayant dans un autre sens fixé à l'avance ? Pourquoi n'y a-t-il pourtant aucune dépendance circulaire ?

Question 31. Écrire une fonction `plus_longue_descente(h)` qui prend en argument une grille `h` et renvoie la longueur d'une plus longue descente, calculée par une fonction récursive mémoïsée (avec un dictionnaire dont les clés sont les couples `(x, y)`). Quelle est sa complexité ? Pourquoi la mémoïsation est-elle ici plus commode qu'un tableau rempli par deux boucles ?

6. Bonus : le jeu des pièces

Des pièces de valeurs $v_0, \dots, v_{n-1}$ sont alignées. Deux joueurs A et B retirent à tour de rôle une pièce à l'**une des deux extrémités** de la rangée, A commençant. Chacun cherche à maximiser la somme de ses pièces.

Question 32. Sur la rangée $8, 15, 3, 7$, que gagne A si chaque joueur prend toujours la plus grande des deux pièces disponibles ? Que gagne-t-il s'il commence par prendre 7 ?

Question 33. Pour $0 \leq i \leq j \leq n-1$, on note $G(i,j)$ le gain du joueur qui commence sur la rangée $v_i, \dots, v_j$ lorsque les deux joueurs jouent au mieux, et $S(i,j) = v_i + \cdots + v_j$. Écrire une fonction `gain_max(v)` qui prend en argument une liste non vide `v` et renvoie le gain $G(0, n-1)$ du joueur A, en $O(n^2)$.