Exercice 1 (Les bûcherons) : L'entreprise WoodStock achète des troncs d'arbres (supposés cylindriques) et les découpe en bouts de bois plus courts (avec une granularité de 1m). On peut effectuer la coupe où l'on souhaite pourvu que la longueur découpée soit entière et ensuite on vend le bois coupé. On cherche à trouver la découpe qui va optimiser le prix de vente. Supposons que l'on sache, pour $i = 1, \dots, n$, le prix p_i que l'on peut tirer de la vente d'une coupe de longueur i m.
Par exemple :

longueur (en m)	1	2	3	4	5	6
prix p_i	1	5	8	9	10	17

Soit un tronc de longueur n mètres. On ne s'autorisera qu'à faire des découpes de taille un nombre entier de mètres. On note r_n le revenu maximal que l'on peut tirer à partir d'un tronc de longueur n.

1. Combien y a-t-il de découpes possibles ? (pour simplifier, on considèrera par exemple que $(1, 1, 2)$ et $(1, 2, 1)$ sont deux découpes différentes d'un tronc de taille 4)
2. Établir une relation de récurrence satisfaite par r_n.
3. Écrire une fonction `calculeR p` qui prend en entrée un tableau de longueur n représentant les prix et qui calcule r_n en utilisant une approche de programmation dynamique. Votre fonction devra également renvoyer la découpe optimale.
4. Quelle est la complexité temporelle ? Spatiale ?

Exercice 2 (Le rendu de monnaie) : On fera l'hypothèse que l'ensemble des pièces dans le système monétaire comporte au moins une pièce de valeur 1. Ainsi quel que soit la valeur de la monnaie qu'on veuille rendre, il existe au moins une façon de la rendre.

Pour modéliser le système monétaire européen classique, on prendra comme unité le centime et on considèrera les pièces $p = [|1; 2; 5; 10; 20; 50|]$.

Approche gloutonne. L'approche gloutonne repose sur l'idée suivante : pour rendre la somme on rend la pièce de la plus grande valeur tant que c'est possible puis on passe à la deuxième etc. jusqu'à avoir rendu toute la monnaie. On supposera que les pièces du système monétaire sont représentées par le tableau trié de leurs valeurs.

1. Proposer une implémentation itérative ou récursive de cette approche.

Programmation dynamique. L'approche gloutonne est optimale pour la plupart des systèmes monétaires du monde. Mais elle ne permet pas en toute généralité de trouver le nombre minimal de pièces nécessaires pour rendre la monnaie. Par exemple, l'approche gloutonne pour rendre 6 avec les pièces $[1, 3, 4]$ conduit à rendre la monnaie avec 3 pièces, alors qu'il existe manifestement une solution optimale avec deux pièces uniquement.

On pose $s(i, k)$ le nombre de pièces minimal nécessaire pour obtenir la somme k avec les i premières pièces du système monétaire, on va adopter une approche de programmation dynamique pour traiter le problème.

2. On suppose que les pièces sont représentées par un tableau de valeurs p. Justifier que l'équation de programmation dynamique satisfaite par s est :

$$s(i,k) = \min \begin{cases} 1 + s(i, k - p.(i)) \text{ si (k} - \text{p.(i))} \geq 0 \\ s(i-1, k) \text{ si i} \geq 1 \end{cases}$$

3. Proposer une implémentation de cette méthode. Votre fonction prendra en entrée un tableau correspondant aux valeurs faciales des pièces et un entier correspondant au montant à rendre et renverra le nombre minimal de pièce nécessaires pour rendre la monnaie.
4. Améliorer votre programme pour renvoyer la liste des pièces impliquées.

Exercice 3 (Distance de Levenshtein) : Soit u une chaîne de caractères, on appelle opération élémentaire la suppression, l'insertion, ou la modification d'une lettre dans la chaîne. On appelle distance d'édition (ou distance de Levenshtein) entre deux mots u et v le nombre minimal $d(u, v)$ d'opérations élémentaires nécessaires pour obtenir v à partir de u.

1. (si vous êtes en avance) Vérifier que d est une distance au sens de votre cours de maths.
2. Soit u, v deux mots de longueurs l_u et l_v. Pour $i \leq l_u$ et $j \leq l_v$, on note $d_{i,j}$ la distance d'édition entre les préfixes de u et de v de longueurs respectives i et j. Déterminer une relation de récurrence entre $d_{i,j}$, $d_{i,j-1}$, $d_{i-1,j-1}$ et $d_{i-1,j}$.
3. Décrire un algorithme permettant de calcul de la distance de Levenshtein fondé sur le principe de la programmation dynamique. En proposer une implémentation.

Exercice 4 (Problème du sac à dos) : On dispose d'un sac à dos ayant une capacité maximale c_m, dans lequel on peut mettre des objets. Chaque objet a un poids et une valeur. On numérote les objets de 0 à $n-1$, et on note w_i et v_i le poids (*weight*) et la valeur de l'objet i.

Le but est de maximiser la valeur des objets dans le sac à dos, sans excéder la limite de poids. On veut donc trouver le meilleur ensemble $X \subseteq [\![0, n-1]\!]$ tel que :

- $\sum_{i \in X} w_i \leq c_m$;
- $\sum_{i \in X} v_i$ est maximal.

1. Définir un type enregistrement `item` ayant deux attributs `v` et `w`.
2. Définir une fonction `somme` de type `item list -> (int * int)` qui prend une liste d'objets en entrée et renvoie la somme des valeurs et la somme des poids des objets de la liste.

Algorithme glouton. Une stratégie gloutonne consiste à trier les objets par valeur décroissante, et à mettre les plus précieux en premier tant qu'ils rentrent dans le sac à dos. [1]

3. Écrire une fonction `sac_a_dos_glouton` de type `item array -> int -> (int * int * item list)` qui prend un ensemble d'objets sous forme d'un tableau et une capacité maximale, et renvoie une liste d'objets à mettre dans le sac à dos en suivant la stratégie gloutonne (ainsi que la valeur et le poids de cette liste).

Programmation dynamique On note $KP(i, c)$ une valeur maximale du sous-problème où qui utilise uniquement les i premiers objets et un sac à dos de capacité c. Le but est alors de calculer $KP(n, c_m)$.

4. Montrer qu'on a :
   - $\forall c \leq c_m,\ KP(0, c) = 0$
   - si $c < w_i$, alors $KP(i, c) = KP(i-1, c)$
   - si $c \geq w_i$, alors $KP(i, c) = \max\{KP(i-1, c), KP(i-1, c-w_i) + v_i\}$.
5. Écrire une fonction `calcul_kp` prenant un ensemble d'objets sous la forme d'un `item array` et une capacité c_m, et renvoie une matrice contenant les valeurs de $KP(i, c)$.
6. Écrire une fonction `sac_a_dos_dyna` de type `item array -> int -> (int *int * item list)` qui prend un ensemble d'objets sous forme d'un tableau et une capacité maximale, et renvoie une liste d'objets optimale à mettre dans le sac à dos (ainsi que la valeur et le poids de cette liste).

1. Il y en a d'autres. Par exemple, trier les objets par rapport aux valeurs ($\frac{v_i}{w_i}$), ce qui fonctionne pas mal en pratique.