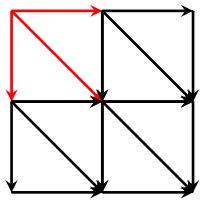
TD - Programmation dynamique

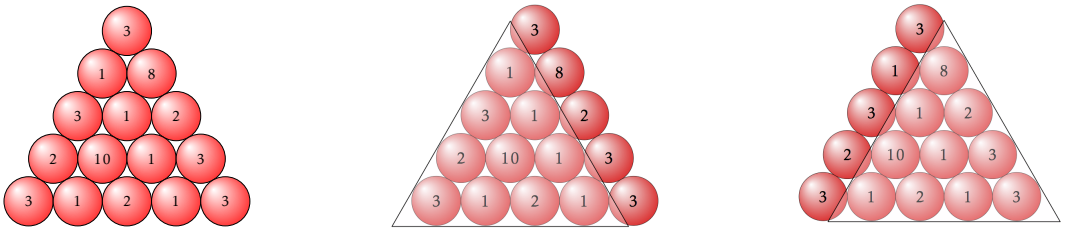
Exercice 1 (Chemins sur un quadrillage).



Le but de l'algorithme est d'être capable de déterminer, pour n'importe quelle grille $n \times m$ de ce type, le nombre de chemins menant du coin supérieur gauche au coin inférieur droit.

1. Une grille comporte 2×2 cases. Partant du coin supérieur gauche, déterminer le nombre de chemins menant au coin inférieur droit si les seules directions de déplacements autorisées sont celles indiquées par les flèches.
2. Une grille comporte $(m+1) \in \mathbb{N}^*$ lignes et $(n+1) \in \mathbb{N}^*$ colonnes. On note $c_{i,j}$ le nombre de chemins issus d'un nœud $(i,j) \in [\![0,m]\!] \times [\![0,n]\!]$. Déterminer $c_{0,n}$ et $c_{m,0}$ puis, pour $i \geqslant 1$ et $j \geqslant 1$, établir une relation de récurrence donnant $c_{i,j}$.
3. Écrire une fonction `npath_rec : int -> int -> int` qui calcule le nombre de chemins permettant d'aller du coin supérieur gauche au coin inférieur droit d'une grille $m \times n$ de manière récursive. Discuter sa complexité temporelle et spatiale.
4. Écrire une fonction `npath_top_down_memo : int -> int -> int` qui calcule ce même nombre de chemins par une approche *top-down*.
5. Écrire une fonction `npath_bottom_up_memo : int -> int -> int` qui calcule ce même nombre de chemins par une approche *bottom-up*.
6. Peut-on améliorer la complexité spatiale de cette dernière solution ? Écrire une fonction `npath_bottom_up_memo_better : int -> int -> int` tentant une amélioration dans ce sens.

Exercice 2 (Pyramides).
L'objet de cet exercice est de déterminer la somme maximale obtenue en parcourant, de haut en bas, une pyramide de nombres entiers positifs de hauteur n. Seuls les deux éléments situés immédiatement en dessous d'un nombre sont pris en compte pour le calcul de la somme. Sur la figure, la pyramide de hauteur 5 a pour somme maximale 24 (valeur optimale), la somme étant obtenue en passant successivement par les nombres 3, 8, 1, 10, 2 (solution optimale).



Une pyramide est représentée par un tableau de n sous-tableaux. Par exemple, la pyramide ci-dessus est représentée par le tableau bidimensionnel suivant :

$$p = \begin{array}{ccccc} 3 & & & & \\ 1 & 8 & & & \\ 3 & 1 & 2 & & \\ 2 & 10 & 1 & 3 & \\ 3 & 1 & 2 & 1 & 3 \end{array}$$

1. Implémenter un algorithme glouton `pyramide_greedy` pour résoudre ce problème. Qu'en pensez-vous ? Proposer un exemple pour montrer les limites d'un tel algorithme.
2. Une pyramide de hauteur n peut être décomposée en deux sous-pyramides de hauteurs $(n-1)$ (voir deux figures les plus à droite). On désigne par $s_{i,j}$ la somme maximale pour une sous-pyramide de sommet l'élément $p_{i,j}$. Établir une relation de récurrence entre les sommes $s_{i,j}$.
3. En utilisant cette relation de récurrence, écrire une fonction `pyramide_rec : int array array -> -> int` qui calcule la somme maximale. Quelle est sa complexité temporelle ? Quelle sa complexité spatiale ?
4. Écrire une fonction `pyramide_rec_memo` utilisant une approche *top-down* avec mémoïsation. Quelle est maintenant la complexité temporelle ? La complexité spatiale ?
5. Écrire une fonction `pyramide_bottom_up` utilisant une approche *bottom-up*. Quelle est maintenant la complexité temporelle ? La complexité spatiale ?
6. On peut encore améliorer la complexité spatiale dans le tas de cette implémentation, en analysant finement l'ordre de calcul des sous-problèmes et en ne conservant que les résultats des sous-problèmes encore utiles pour la suite. Écrire une fonction `pyramide_bottom_up_better` prenant en compte cette possibilité d'amélioration. Quelle est maintenant la complexité temporelle ? La complexité spatiale ?
7. Comment reconstituer le chemin des entiers ayant permis d'obtenir cette somme maximale ?

Exercice 3 (Rendu de monnaie).
Soit un ensemble S_n de n pièces de monnaie de valeurs entières $v_1 < v_2 < \cdots < v_n$, avec $v_1 = 1$. On désigne par s une somme entière à rendre avec le minimum de pièces et par $m_n(s)$ le nombre minimal de pièces pour rendre cette somme avec le système de pièces S_n.

1. Que valent $m_n(0)$, $m_n(1)$ et $m_1(s)$? Établir une relation de récurrence donnant $m_n(s)$.
2. Par une approche *récursive naïve*, écrire une fonction `change_rec: int -> int array -> int` qui reçoit une somme `amount`, un système de monnaie défini par un tableau `monetary_system` et qui renvoie le nombre minimal de pièces utilisées pour rendre la somme `amount`. Discuter son efficacité.
3. Répondre à la même question par une approche *top-down*. On écrira une fonction `change_top_down_memo` ayant le même prototype.
4. Estimer les complexités spatiale et temporelle de cette fonction en fonction du nombre d'unités de valeurs différentes disponibles n et de la somme `amount` ; notée s, à rendre.
5. Répondre à la même question par une approche *bottom-up*. On écrira une fonction `change_bottom_up_memo` ayant toujours le même prototype.
6. Quelles sont les nouvelles complexités spatiale et temporelle ?
7. Peut-on améliorer la complexité spatiale dans le segment du tas de cette fonction ? Si oui, écrire une fonction `change_bottom_up` prenant en compte cette amélioration.
8. Écrire une fonction `get_change` appelant `change_bottom_up` et qui retrouve la liste et le nombre des différentes unités de valeurs rendues.
9. En pratique, quelle algorithme appliquons-nous pour rendre une somme d'argent ? Coder cet algorithme `change_greedy`. Quel résultat renvoie-t-il avec le système $S_4 = \{1, 2, 5, 10\}$ puis avec le système $\{1, 3, 6, 12, 24, 30\}$ pour rendre la somme 49 ? Qu'en pensez-vous ?

Exercice 4 (Sac à dos).
Étant donné un ensemble de $n \in \mathbb{N}^*$ objets, chacun de masse $w_i \in \mathbb{N}^*$ et de valeur $v_i \in \mathbb{N}^*$, avec $i \in [\![1, n]\!]$, le *problème du sac à dos* consiste à choisir certains de ces objets [a] en respectant deux contraintes :

— la masse totale des objets choisis ne doit pas dépasser une valeur [b] $w_{\max}$;

— la valeur totale des objets choisis doit être maximale.

L'objectif est de déterminer un n-uplet $(x_1, \ldots, x_n) \in \{0, 1\}^n$ tel que :

$$\sum_{i=1}^{n} x_i v_i \text{ soit maximale et } \sum_{i=1}^{n} x_i w_i \leqslant w_{\max}$$

1. Quelle serait la complexité d'une méthode exhaustive ?
2. On note $g_i(w)$ le gain maximum généré par le choix, parmi les i premiers objets, de ceux dont la somme des masses ne dépasse pas w. Si $w > 0$ et $i > 0$, établir une relation de récurrence entre $g_i(w)$, $g_{i-1}(w)$, $g_{i-1}(w - w_i)$ et v_i. Que valent $g_0(w)$ et $g_i(0)$?
3. Écrire une fonction récursive `gain_rec : (int * int) array -> int -> int` qui reçoit un tableau de couples d'entiers (v_i, w_i), un entier $w_{\max}$ et qui renvoie la valeur du gain maximum.
4. Écrire une fonction `gain_top_down_memo` qui calcule la valeur maximale qui peut rentrer dans le sac par une *approche top-down* avec mémoïsation.
5. Écrire une fonction `gain_bottom_up_memo` qui réalise la même chose par une *approche bottom-up* avec mémoïsation.
6. Améliorer la fonction précédente en créant une une fonction `gain_top_down_memo_with_selected_obj` qui renvoie en plus la liste des objets qui ont été sélectionnés pour rentrer dans le sac (à vous de choisir sous quelle forme...)

a. Un objet ne peut être choisi qu'une seule fois.
b. Charge maximale que le sac peut emporter.