TP : programmation dynamique

1 Pyramide de nombres

On dispose d'une pyramide de nombres comme sur le figure 1.

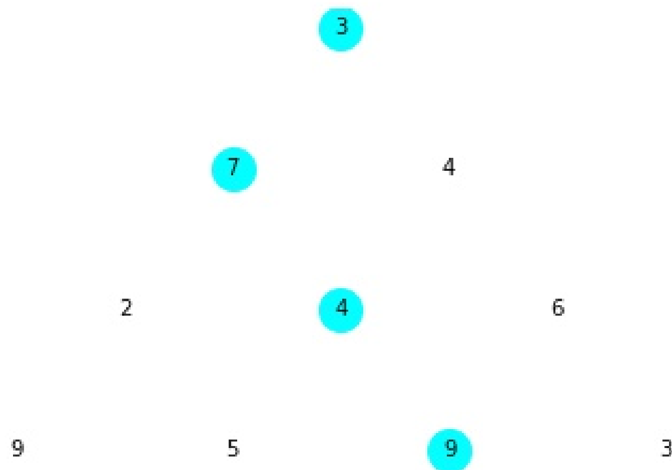

FIGURE 1 – Une pyramide de nombres

On cherche, en partant du sommet de la pyramide, et en se dirigeant vers le bas à chaque étape, à maximiser le total des nombres traversés. Dans l'exemple ci-dessus, le maximum est obtenu pour le chemin en couleur cyan ($3+7+4+9 = 23$).

Pour se déplacer dans la pyramide, il y a une seule règle. Depuis un sommet d'un étage supérieur, on peut seulement se rendre vers deux sommets de l'étage inférieur : le sommet à gauche du sommet courant et celui à droite.

Q.1 Considérons une pyramide à n niveaux (le niveau 0 étant celui du sommet de la pyramide). L'algorithme naïf consiste à explorer tous les chemins possibles.

(a) Combien y a-t-il de chemins possibles depuis la racine ?

(b) Si l'on explore tous ces chemins, combien d'additions sont effectuées ?

Les pyramides d'entiers sont implantées par des tableaux de tableaux. Voici comment représenter la pyramide de la figure 1 :

```
let pyr =
  [|[|3|];[|7;4|];[|2;4;6|];[|9;5;9;3|]|];;
```

Q.2 Écrire une fonction `max_sum_rec : int array array -> int` qui implante la recherche récursive naïve. On ne cherche pas à optimiser à ce stade.

Q.3 On note c_i le nombre d'appels internes pour la profoncdeur i. Déterminer c_0 pour une pyramide à n niveaux.

Q.4 En examinant le calcul de la meilleure somme jusqu'au niveau 2, établir que certains calculs sont faits plusieurs fois.

Q.5 En mémoïsant les résultats intermédiaires dans un dictionnaire, établir une nouvelle version `max_sum_memo : int array array -> int` du calcul de la plus grande somme.

Q.6 Etudier la complexité de cette fonction en temps et en espace.

Dans ce qui précède, nous avons calculé la plus grande somme par une approche *du haut vers le bas* (ou *top-down*). C'est à dire que le père demande (à construire) la valeur des fils pour construire sa propre valeur. Le code ressemble un peu à un parcours en profoncdeur.

À l'inverse, on peut travailler par niveau de profondeur décroissant. Ainsi, on commence les calculs au niveau des feuilles, puis on remplit le niveau au dessus, puis celui d'après etc. Une telle approche est appelée *bottom-up* (du bas vers le haut). La programmation impérative se prête bien à ce calcul.

Q.7 Écrire la fonction `max_sum_matrix : int array array -> int` qui renseigne couche par couche les coefficients d'une matrice des meilleurs sommes au niveau de chaque nœud.

Q.8 Établir la complexité temporelle puis spatiale de cet algorithme.

Plutôt que de gérer une matrice (qui prend beaucoup d eplace en mémoire), on se rend compte qu'un seul tableau unidimensionnel est nécessaire pour construire la solution *bottom-up* si on se débrouille bien.

Q.9 Écrire la fonction `max_sum_array : int array array -> int` qui réalise ce principe.

Q.10 Pour finir, on veut obtenir non seulement la meilleur somme mais aussi la liste des positions qui permettent de la réaliser. Écrire pour ce faire la fonction `max_sum_sol : int array array -> int * (int * int) list`.

```
# max_sum_sol pyr;;
- : int * (int * int) list = (23, [(0, 0); (1, 0); (2, 1); (3, 2)])
```

Q.11 Donner les complexités spatiales et temporelles.