TD Bonus n°1

Programmation dynamique

Ce TD est un rappel de programmation dynamique et de la méthode à suivre pour résoudre un problème par programmation dynamique.

Résoudre un problème par « programmation dynamique », c'est résoudre le problème de telle sorte à ce que l'on ne réalise pas de calculs redondants. On stockera donc des résultats intermédiaires dans un tableau n-dimensionnel[1] que l'on notera DP.[2]

(1) Quelle est la taille de la table DP ? À quoi correspond une entrée de la table DP ?
(2) Calcul d'une entrée. Comment peut on calculer une entrée en fonction des autres ?
(3) Cas de base. Quelles entrées ne dépendent pas des autres ?
(4) Ordre de calcul. Dans quel ordre doit on réaliser les calculs de telle sorte que les valeurs nécessaires pour chaque entrée aient déjà été réalisées dans les étapes précédentes ?
(5) Extraire la solution. Comment la solution finale peut être extraite de la table DP remplie ?
(6) Temps de calcul. Quelle est la complexité de votre solution ?
(7) Écrire *explicitement* l'algorithme.

Fig. 1. Méthode de résolution par programmation dynamique

L'objectif des problèmes ci-après est d'appliquer cette méthode.

Exercice I. Collection de pièces.

On joue à un jeu vidéo dont le but est de collecter un maximum de pièces, dans une grille de taille $m \times n$. On représentera le monde par un tableau $\boldsymbol{A}$ de taille $m \times n$ où chaque cellule contient : ou bien un pièce (notée `C`), ou bien un obstacle (noté `#`), ou bien rien (noté `·`).

Le joueur commence à la case $(1, 1)$, et cette case sera toujours sans obstacle. À chaque tour, vous pouvez choisir d'aller vers le bas, ou vers la droite, ou bien de s'arrêter (ce qui termine le jeu). Le mouvement « aller vers la droite » sera représentée par la transformation $(x, y) \to (x, y + 1)$[3] et $(x, y) \to (x + 1, y)$ pour « aller vers le bas ».

En allant dans une cellule `C`, le joueur gagne une pièce. Le but est d'obtenir le nombre maximal de pièces.

[1] Généralisation d'une matrice à n-dimensions, au lieu de 2.

[2] Pour *Dynamic Programming*

[3] Bien que les coordonnées semblent inversés dans cette transformation, ce sont les bons. Il faut considérer la table $\boldsymbol{A}$ comme une matrice, on indexe donc sur les lignes avant les colonnes.

Exemple. On considère le tableau $\boldsymbol{A}$ de taille 5×6 ci-contre (Fig. 2). Le joueur peut collecter 5 pièces en suivant le chemin rouge. C'est le maximum. Un chemin sous-optimal, comme le chemin violet, ne permet d'avoir que 4 pièces.

Remarque. On prendra soin de n'accéder qu'aux cellules de la forme $\boldsymbol{A}[i,j]$ avec $i \in [\![1,m]\!]$ et $j \in [\![1,n]\!]$, afin de ne pas « sortir » de la grille.

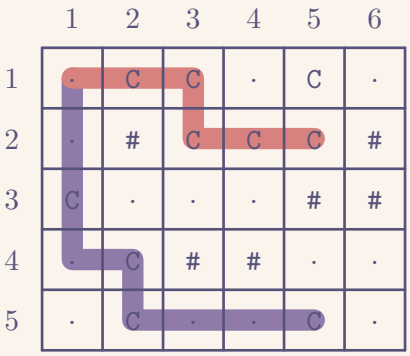

Fig. 2. Petit exemple . . .

Q1. Proposer un algorithme d'une fonction récursive $f(x,y)$ qui prendrait en argument la position (x,y) du joueur et qui retourne le nombre maximale de pièces que le joueur peut collecter (en supposant qu'il n'en n'avait aucune avant). Ceci sera l'implémentation naïve, que l'on améliorera par la suite.

RÉPONSE. On construit l'algorithme ci-dessous.

Procédure $f(x,y)$
- **Si** $x \notin [\![1,m]\!]$ ou $y \notin [\![1,n]\!]$ **alors**
  - **Renvoyer** 0
- **Fin si**
- **Si** $\boldsymbol{A}[x,y] =$ « # » **alors**
  - **Renvoyer** 0
- **Sinon si** $\boldsymbol{A}[x,y] =$ « · » **alors**
  - **Renvoyer** $\max(f(x+1,y), f(x,y+1))$
- **Sinon** ($\boldsymbol{A}[x,y] =$ « C »)
  - **Renvoyer** $1 + \max(f(x+1,y), f(x,y+1))$
- **Fin si**

Par exemple, dans la grille de la Fig. 2, on a $f(1,1) = 5$, $f(2,1) = 4$, $f(5,5) = 0$.

Q2. Démontrer que l'algorithme de Q1 termine. Démontrer que l'algorithme de Q1 est correct.

RÉPONSE. Considérons $V(x,y) = (m-x) + (n-y) + 1$, montrons que c'est un variant qui décroit strictement.

- Premièrement, $V(x,y) \in \mathbb{N}$. En effet, les valeurs de x et y restent dans la zone de jeu. Et, même si l'on sort, on sera à $x = m+1$ et $y \in [\![1,n]\!]$ ou $x \in [\![1,m]\!]$ et $y = n+1$. Ce qui permet de montrer que $V(x,y) \geq 0$.
- Montrons que $V(x,y)$ décroit strictement. L'appel de $f(x,y)$ engendre au plus deux appels à la fonction f avec $f(x+1,y)$ et $f(x,y+1)$. On vérifie aisément que $V(x+1,y) = V(x,y+1) < V(x,y)$.

Le variant décroit strictement dans un ensemble ordonné bien fondé $(\mathbb{N}, \leq)$. On en déduit qu'il est ultimement stationnaire, *i.e.* le programme termine.

On considère l'invariant $(\mathscr{I})$: l'appel de $f(x,y)$ renvoie le nombre maximal de pièces récupérables. Montrons que cet invariant en est un.

- On a $f(x,y) = 0$ pour $x \notin [\![1,m]\!]$ ou $y \notin [\![1,n]\!]$ ou $\boldsymbol{A}[x,y] =$ « # ». Ceci est en accord avec $(\mathscr{I})$. En effet, par hypothèse du jeu, ces situations ne sont pas possibles.
- Dans l'autre cas, par hypothèse d'induction, les appels de $f(x+1,y)$ et $f(x,y+1)$ sont corrects. Et, par les hypothèses du jeu, on ne peut *que* aller à droite ou en bas. On en déduit que l'invariant $(\mathscr{I})$ est encore vérifié.

La propriété $(\mathscr{I})$ est bien un invariant. Ainsi, l'algorithme est correct.

L'algorithme de Q1 réalise des calculs redondants. On décide donc de sauvegarder ces résultats. On appelle ça la *mémoïsation*.

Q3. Réécrire l'algorithme de Q1 en appliquant de la mémoïsation. Démontrer que l'appel $f(1,1)$ a une complexité pire cas en $\mathrm{O}(m \cdot n)$.

RÉPONSE. On supposera avoir, dans la mémoire globale, une matrice $\boldsymbol{M} \in \mathscr{M}_{m,n}(\mathbb{R})$. On initialisera cette matrice avec la valeur -1 (représentant « rien n'est stocké »). On modifie donc l'algorithme pour obtenir celui ci-dessous.

Procédure $f(x, y)$
  Si $x \notin [\![1, m]\!]$ ou $y \notin [\![1, n]\!]$ **alors**
    Renvoyer 0
  Fin si

  Si $\boldsymbol{M}[x, y] = -1$ **alors**
    Si $\boldsymbol{A}[x, y] =$ « `#` » **alors**
      $\boldsymbol{M}[x, y] \leftarrow 0$
    Sinon si $\boldsymbol{A}[x, y] =$ « `·` » **alors**
      $\boldsymbol{M}[x, y] \leftarrow \max(f(x+1, y), f(x, y+1))$
    Sinon ($\boldsymbol{A}[x, y] =$ « `C` »)
      $\boldsymbol{M}[x, y] \leftarrow 1 + \max(f(x+1, y), f(x, y+1))$
    Fin si
  Fin si

  Renvoyer $\boldsymbol{M}[x, y]$

L'appel $f(1,1)$ génère des appels récursifs $f(x,y)$ pour tout (x,y) dans la grille. Pour chaque valeur de (x,y), un seul appel de $f(x,y)$ génère des appels récursifs. Ainsi, on peut en déduire que la complexité de l'appel $f(1,1)$ est en $\mathrm{O}(m \cdot n)$, car on ne remplira « pas plus » que la matrice $\boldsymbol{M}$.

La question suivante consiste à réaliser la même tâche, mais sans récursivité.

Q4. Écrire l'algorithme calculant la solution en $\mathrm{O}(m \cdot n)$, mais qui n'utilise pas de récursivité. On procèdera comme expliqué dans la Fig. 1.

RÉPONSE. On utilise la *programmation dynamique*.

(1) La taille de DP est $m \times n$. L'entrée en (i, j) correspond à la valeur de $f(i, j)$.
(2) Chaque entrée de DP est le maximum de :
  (a) s'il y a une pièce sur la case (x, y) ou non (1 si oui, 0 si non), (arrêt)
  (b) s'il y a une pièce sur la case (x, y) ou non, auquel on somme $\mathrm{DP}[x+1, y]$, (vers le bas)
  (c) s'il y a une pièce sur la case (x, y) ou non, auquel on somme $\mathrm{DP}[x, y+1]$. (vers la droite)
(3) Le cas de base est le cas « (arrêt) » ci-dessus : si le chemin s'arrête à (x, y) alors on initialise $\mathrm{DP}[x, y]$ à 1 s'il y a une pièce, et 0 sinon.
(4) Un ordre des solution est de bas en haut, puis de droite à gauche. On peut également faire de droite à gauche puis de bas en haut.
(5) La solution au problème est la valeur de $\mathrm{DP}[1, 1]$.
(6) La complexité est en $\mathrm{O}(m \cdot n)$ car il y a $m \times n$ valeurs à remplir dans la table DP, et que chaque calcul est en $\mathrm{O}(1)$.

(7)
```
Pour y ∈ ⟦n, 1⟧ faire
    Pour x ∈ ⟦m, 1⟧ faire
        DP[x, y] ← 0

        Si x < m et A[x + 1, y] ≠ « # » alors
            DP[x, y] = max(DP[x + 1, y], DP[x, y])
        Fin si

        Si y < n et A[x, y + 1] ≠ « # » alors
            DP[x, y] = max(DP[x, y + 1], DP[x, y])
        Fin si

        Si A[x, y] = « C » alors DP[x, y] ← DP[x, y] + 1
    Fin pour
Fin pour
```

Exercice II. *k*-somme.

On dit qu'un entier $n \in \mathbb{N}$ est une k-somme dès lors que l'on peut l'écrire comme $n = a_1^k + \cdots + a_p^k$, où les nombres $a_1, \ldots, a_p$ sont des entiers naturels **distincts**, pour un certain $p \in \mathbb{N}$.

Par exemple, 36 est une 3-somme car on peut l'écrire comme $36 = 1^3 + 2^3 + 3^3$.

Q1. Par programmation dynamique, écrire un algorithme prenant en entrée deux entiers n et k et renvoyant le booléen ***V*** si et seulement si n est une k-somme.

L'algorithme devrait avoir une complexité asymptotique en $\mathrm{O}\big(n^{1+2/k}\big)$. On procèdera en suivant la méthode de la Fig. 1.

RÉPONSE. Étant donnés n et k, on pose $m = \lfloor \sqrt[k]{n} \rfloor$, le plus petit entier tel que $m^k \leq n$.

(1) La table DP a une taille $n \times m$. Chaque entrée dans la table est un booléen, tel que $\mathrm{DP}[i, j] = \boldsymbol{V}$ *ssi* i est une k-somme (pour un certain k) composée de nombres inférieurs à j (et supérieur ou égal à 1).

(2) Le calcul d'une entrée de DP se fait comme suit :

$$
\begin{array}{lll}
(\text{♩}) & \mathrm{DP}[0, j] = \boldsymbol{V} & \text{si } j \in [\![0, m]\!] \\
(\text{♪}) & \mathrm{DP}[i, 0] = \boldsymbol{F} & \text{si } i \in [\![1, n]\!] \\
(\text{♫}) & \mathrm{DP}[i, j] = \mathrm{DP}[i - j^k, j - 1] +_{\mathbb{B}} \mathrm{DP}[i, j - 1] & \text{si } i \in [\![j^k, n]\!] \\
(\text{♬}) & \mathrm{DP}[i, j] = \mathrm{DP}[i, j - 1] & \text{sinon}
\end{array}
$$

L'opération $+_{\mathbb{B}}$ correspond au « ou » booléen. L'équation (♩) dit que 0 peut toujours être écris comme une somme (vide) de l'intervalle $[\![1, j]\!]$. L'équation (♪) dit que les valeurs non-nulles ne peuvent pas être obtenues comme somme d'entiers de l'intervalle $[\![1, 0]\!] = \emptyset$. Les équations (♫) et (♬) donne la relation de récurrence : un entier i est une k-somme $a_1^k + \cdots a_p^k$ d'entiers inférieurs à j (on supposera les a_ℓ triés par ordre croissant) *ssi* on a un des cas suivant :

(a) un des a_ℓ est j, c'est ainsi a_p, et donc $a_1^k + \cdots + a_{p-1}^k = i - a_p^k = i - j^k$, où les $a_1, \ldots, a_{p-1}$ sont inférieurs strictement à j ;

(b) aucun des a_ℓ n'est j et donc la k-somme i n'est composée que de nombres inférieurs à j strictement.

(3) Les cas simples sont (♩) et (♪).

(4) Avec les relations de récurrences (♫) et (♬), on ordonne par valeurs de j croissantes. L'ordre sur i n'a pas d'importance.

(5) La solution au problème est $\mathrm{DP}[n, m]$ car tous les a_ℓ dans la k-somme de n (sous reserve d'existence) vérifie $a_\ell^k \leq n$ et donc $a_\ell \leq \lfloor n^{1/k} \rfloor = m$.

(6) La complexité est en $O(n \cdot \sqrt[k]{n})$. En effet, il y a $(n+1)m + 1$ cellules dans la table DP, et que chaque calcul a une complexité en $O(1)$ pour chaque cellule.

Exercice III. Comptage des chaînes de caractères.

Étant donné un mot binaire $s \in \{0,1\}^n$ de taille n, on note $f(n)$ la longueur du plus long sous-mot de 1 consécutifs. Par exemple, $f(0110001101110001) = 3$ car il contient le sous-mot 111 (souligné), mais pas 1111.

Étant donnés n et k, l'objectif est de compter le nombre de mots s de taille n tel que $f(s) = k$.

Q1. En utilisant la méthode présentée en Fig. 1, écrire un algorithme qui, pour n et k donnés avec $k \leq n$, répondant au problème. Cet algorithme devra avoir une complexité *polynomiale*, même $O(n^{11}k^{20})$ est acceptable, mais une solution est possible en $O(nk^2)$.

Exercice IV. Plus long serpent.

On nous donne une grille hexagonale ayant les cases $C_1, \ldots, C_n$. Chaque case contient des entiers naturels, on notera v_i la valeur de la case C_i.

Une liste finie de cases $\left(C_{i_1}, \ldots, C_{i_k}\right)$ est un *serpent* de longueur k si, pour $j \in [\![1, k-1]\!]$, les cases C_{i_j} et $C_{i_{j+1}}$ sont voisines et que leurs valeurs vérifient $v_{i_{j+1}} = v_{i_j} + 1$. La Fig. 3 (page 5) montre un exemple de grille sur laquelle on montre le plus long serpent.

Pour simplifier, on supposera que la case C_i est représentée par un indice. De plus, on supposera que l'on peut connaître les voisins de chaque case : on supposera disposer d'une fonction $\mathcal{N}$[4] qui retourne l'ensemble des cases voisines à une case donnée. Un appel à $\mathcal{N}$ est une opération élémentaire.

Q1. En suivant la méthode de la Fig. 1, donner un algorithme en programmation dynamique qui, étant donné un jeu retourne la longueur du plus long serpent. L'algorithme aura une complexité en $O(n \log n)$, où n est le nombre de cases.

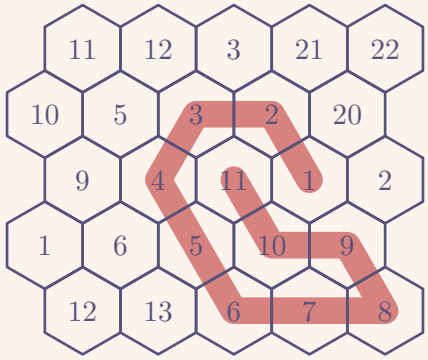

Fig. 3. Exemple du plus long serpent (longueur 11)

RÉPONSE. On applique la méthode.

(1) La table DP est linéaire, de taille n. Chaque case contient la longueur maximum du plus long serpent commençant à cette case.

(2) Le calcul d'une entrée i est donné par :

$$(\♫) \qquad \mathrm{DP}[i] \triangleq 1 + \max_{\substack{C_j \in \mathcal{N}(C_i) \\ v_j = v_i - 1}} \mathrm{DP}[j]$$

Dans cette formule, on supposera que le maximum d'un ensemble vide vaut 0.

(3) Le cas de base correspond au maximum de l'ensemble vide ci-dessus.

(4) On trie les cases par valeur. On traite les cases par ordre croissant de valeurs.

[4] Pour *neighbors*

(5) La solution est $\max_{i\in[\![1,n]\!]} \mathrm{DP}[i]$.
(6) L'ordre des calculs a une complexité en $\mathrm{O}(n\log n)$ avec un bon algorithme de tri. On peut supposer que le calcul (♫) se réalise en $\mathrm{O}(1)$. En effet, le nombre de voisins est de 6 au maximum et, pour n grand, il varie presque pas (sauf pour les cases au bord). Extraire la solution a une complexité en $\mathrm{O}(n)$. On en déduit une complexité en $\mathrm{O}(n\log n)$.
(7) L'écriture de l'algorithme est un exercice laissé au lecteur.

Dans la question Q7, l'algorithme renvoie la *longueur* du serpent le plus long, mais pas le serpent en lui-même.

Q2. Donner un algorithme prenant en entrée le jeu et la table DP de la question Q7 qui renvoie le plus long serpent. S'il y a plus d'un serpent avec une longueur maximale, l'algorithme peut retourner n'importe lequel d'entre eux. Donner la complexité de votre algorithme à l'aide d'un grand Θ en fonction de n.

RÉPONSE. On ne donne pas l'algorithme explicitement, mais on donne l'idée. On trouve le début du serpent en trouvant un indice j_0 vérifiant $\mathrm{DP}[j_0] = \max_{i\in[\![1,n]\!]} \mathrm{DP}[i]$. Pour déterminer la case j suivante à partir de la case i actuelle, on trouve l'indice j (ou un d'entre eux) parmi les voisins de i qui vérifie $\mathrm{DP}[j] = \mathrm{DP}[i] - 1$. On s'arrête une fois que l'on a les n cases de serpent, où $n = \mathrm{DP}[j_0]$.

Le calcul de la case initiale est en $\Theta(n)$. Le calcul de la case suivante a lieu en $\Theta(1)$. Le calcul total du serpent en $\Theta(n)$.

Remarque. On peut également stocker le prédécesseur dans la table DP. Retrouver la longueur du serpent commençant par la case est plus complexe, mais retrouver le serpent de taille maximale est plus simple.

Q3. Question bonus ! Trouver un algorithme en temps linéaire qui trouve le plus long serpent. C'est à dire un algorithme en $\mathrm{O}(n)$ qui, pour un jeu donné, retourne le plus long serpent.