Programmation dynamique

Résumé

La programmation dynamique est un paradigme algorithmique, au même titre que diviser pour régner ou les algorithmes gloutons.
Comme vu dans le TD, la programmation dynamique est surtout utilisée pour résoudre des problèmes d'optimisation. Le problème doit pouvoir se résoudre à partir de sous-problèmes du même type, la solution dépendant du résultat de ces sous-problèmes. Résoudre un problème en programmation dynamique se fait en trois étapes :

- Trouver une formule de récurrence pour obtenir la valeur optimale (c'est la partie la plus difficile).
- Écrire un algorithme itératif (et non pas récursif) pour obtenir l'optimum. On part des plus petits problèmes et on va vers les plus grands. Cette partie donne la valeur optimale, mais pas la manière dont elle est obtenue.
- Reconstruire la solution optimale a posteriori. Cette dernière étape est parfois ignorée dans le cas de problèmes avec des données très lourdes. On part du plus grand problème et on redescend vers les petits.

Différences entre diviser pour régner et programmation dynamique :

- Diviser pour régner : on part du problème complet, et on le divise en petits morceaux. L'approche est descendante, du haut vers le bas (en anglais : top-down)
- Programmation dynamique : on résout des petits problèmes que l'on combine pour arriver au problème complet. L'approche est ascendante, du bas vers le haut (en anglais bottom-up)
- Diviser pour régner : les « petits problèmes » sont indépendants.
  Exemple : dans le tri fusion avec le tableau [5, 4, 1, 3], trier la partie [5, 4] est indépendant de trier la partie [1, 3]
- Programmation dynamique : les « petits problèmes » sont dépendants. *Exemple* : dans le problème du sac à dos avec un sac de 15 kg, choisir ce que l'on met dans les 7kg restants dépend de ce que l'on a déjà mis dans la partie de 8kg déjà remplie, on ne met pas deux fois le même objet.
- Diviser pour régner : algorithmes récursifs
- Programmation dynamique : algorithmes itératifs.
- L'espace mémoire (la complexité en espace) peut devenir problématique avec la programmation dynamique.
- La programmation dynamique améliore souvent l'approche gloutonne.

Origine du nom :
Cette appellation est due à Bellman (1940) qui effectuait des travaux en optimisation mathématique. Le terme programmation est utilisé dans son sens de planification ou d'ordonnancement de tâches, sans faire référence à une discipline, l'informatique, qui n'existait pas à cette époque. Quant à dynamique, Bellman a déclaré plus tard dans son autobiographie (1984) : « it's impossible to use the word dynamic in a pejorative sense. [. . .] Thus, I thought dynamic programming was a good name. It was something not even a Congressman could object to. ». Selon lui, il avait besoin d'un tel qualificatif pour que l'armée de l'air américaine, qui était l'un de ses principaux financeurs, accepte de continuer de le financer malgré le caractère très théorique et mathématique de ses recherches. Cette histoire est très belle, mais peut-être un peu trop, cf. https://fr.wikipedia.org/wiki/Programmation_dynamique#Histoire.

Exercices

Exercice : le rendu de monnaie.

On rappelle le contexte du problème :
Etant donné un ensemble P (pour pièces) d'entiers, représentant la liste des pièces et billets disponibles, et une somme s, donner :

- Le nombre minimal de pièces/billets de P dont la somme vaut s ;
- Une liste minimal de pièces/billets de P dont la somme vaut s.

1. Dans le système P = Euro, donner les sorties pour s = 13,35 €
2. Rappeler le principe de l'algorithme glouton. Écrire cet algorithme sur papier.
3. L'algorithme glouton est-il optimal pour le système *Pièces* = [1, 4, 6] et s = 8 ?
   Et par curiosité, on peut se poser la même question avec le système impérial anglais ci-dessous, utilisé jusqu'en 1971, avec des survivances jusqu'aux années 1980 :
   La livre était divisée en 20 shillings et un shilling valait 12 pence. Jusque là, c'est bizarre mais ça reste compréhensible. C'est ensuite que ça se complique un peu :
   - Il y avait d'abord les sous-unités du penny : le farthing (1/4 de penny) et le demi-penny.
   - Puis les sous-unités du shilling : le penny évidemment (1/12 shilling), le trois pence (1/4 de shilling), le quatre pence ou groat (1/3 de shilling) et le six pence (1/2 shilling).
   - Enfin, il y avait les sous-unités de la livre : le shilling (1/20 de livre), le florin (1/10 de livre ou 2 shillings), la demi-couronne (1/8 de livre ou 2 shillings et demi), la couronne (1/4 de livre ou 5 shillings), le demi-souverain (1/2 livre) et le souverain or qui valait une livre.
   - La guinée, utilisée de 1663 à 1818, est un cas particulier. Elle valait 21 shillings, soit 1 livre et 1 shilling. Elle était en or, un métal abondamment extrait en Guinée en Afrique de l'Ouest, pays auquel la monnaie doit son nom.
   - Le mark valait 2/3 livres (soit 160 pennies ou 13 shillings et 4 pennies)

   Remarque : La réponse est non. Pour ne pas faire d'anglophobie, précisons que tous les systèmes monétaires d'Europe avant la révolution française, puis Napoléon, étaient aussi complexes. C'est un héritage carolingien : une livre vaut 20 sous et 12 deniers.
   (sources https://omnilogie.fr/ et wikipedia) :

4. Algorithme naïf
   Principe :
   - Si la somme à rendre vaut 0, alors le nombre de pièces nécessaire est 0. L'algorithme renvoie 0.
   - Sinon, pour toutes les pièces p possibles, on soustrait à la somme s la valeur de la pièce p, et on cherche le nombre de pièces à rendre pour la somme $s-p$.
   - On a donc la formule récursive :

$$nb\_pieces(s) = \begin{cases} 0 \text{ si } s = 0 \\ 1 + \min\left(nb\_pieces(s-p) \text{ pour } p \le s\right) \text{ sinon} \end{cases}$$

   L'algorithme récursif correspondant est :

```
Rendu naïf (Pièces , s)
        Si s = 0 renvoyer 0
        (Sinon)                 # automatique donc pas indispensable
        nb_pieces ← s           # au pire on renvoie s pièces de 1 centime
        Pour chaque pièce p de Pièces
                Si p ≤ s :
                        nb_pieces_bis ← Rendu naïf (Pièces , s − p ) + 1
                        nb_pieces ← min(nb_pieces , nb_pieces_bis)
        Renvoyer nb_pieces
```

   Faire tourner à la main cet algorithme avec *Pièces* = [1, 4, 6] et s = 8. Pour cela, on construira l'arbre des appels récursifs.
   Que constatez-vous ?
   Quelle semble être la complexité de l'algorithme ? On pourra imaginer ce qui se passe si on prend $s = 9$, $s = 10$, etc. dans l'exemple.

5. Programmation dynamique
   On va calculer le nombre de pièces nécessaires pour rendre la somme s non pas par une approche descendante, mais par une approche ascendante. On va calculer successivement le nombre de pièces à rendre pour $somme = 0$, $somme = 1$, … jusqu'à $somme = s$.

a. On crée un tableau *nombre_pieces* pour stocker ces résultats. Quelle est la taille du tableau ? Sachant que l'on va chercher à minimiser les éléments de ce tableau (le nombre de pièces à rendre pour chaque somme), à quelle valeur peut-on initialiser les éléments ?
b. On boucle ensuite en remplissant ce tableau dans l'ordre croissant des indices, en réutilisant la formule : $nb\_pieces(somme) = 1 + \min(nb\_pieces(somme - p) \text{ pour } p \leq somme)$

   Faire tourner cette méthode sur l'exemple *Pièces* = {1, 4, 6} et *s* = 8, en modifiant le tableau *nombre_pièces* = [8, 8, 8, 8, 8, 8, 8, 8, 8] au fur et à mesure. On précisera toutes les étapes.
c. Programmer la fonction `renduProgDyn(P, s)` en Python. La tester.
d. Donner la complexité de l'algorithme de rendu de monnaie en programmation dynamique, en fonction de la longueur du tableau *Pièces* et de *s*.
e. Reconstruction de la solution

   On utilise le tableau *nombre_pièces* = [0, 1, 2, 3, 1, 2, 1, 2, 2] obtenu au b. On part de la dernière valeur 2 d'indice 8.

   Pour chaque pièce p le permettant, on calcule l'indice $8-p$. Si $nombre\_pièces[8-p] = nombre\_pièces[8]-1$, alors on a trouvé une valeur possible précédente (il peut y avoir plusieurs possibilités).

   Ici :
   - $nombre\_pièces[8] = 2$
   - $nombre\_pièces[8-1] = nombre\_pièces[7] = 2$ ne convient pas
   - $nombre\_pièces[8-4] = nombre\_pièces[4] = 1$ convient
   - $nombre\_pièces[8-6] = nombre\_pièces[2] = 2$ ne convient pas

   Donc la première pièce rendue a pour valeur 4.

   Appliquer cette méthode pour prouver que la deuxième pièce rendue est aussi 4.

   Programmer la fonction `renduReconstruction(P, s, nombre_pieces)` en Python.

   Quelle est la complexité de cette fonction ?

Exercice : distance de séquences.

Dans cet exercice, on cherche à mesurer la distance entre deux chaînes de caractères. Un des usages importants en est la comparaison de séquences génétiques. Bien sûr, ces méthodes servent aussi aux correcteurs orthographiques, à la reconnaissance de mots lors de scan, etc. La mesure la plus simple (distance de Hamming) est de compter simplement le nombre de caractères différents. Par exemple, `manger` et `mentir` sont à distance 3. On a utilisé cette distance en 1ère dans le TP sur l'algorithme k-nn.

On va ici utiliser une autre distance. A partir de deux mots, on construit un alignement de ces deux mots en insérant des blancs (notés avec le caractère undescore _). Avec ces blancs, on construit deux chaînes de la même longueur, sachant que deux blancs ne sont jamais alignés entre les deux mots. On cherche l'alignement donnant la ressemblance maximale entre les deux chaînes. Dans cet exercice, on calculera le score de la manière suivante : si, au même endroit, deux caractères sont identiques, on ajoute un point ; si les caractères sont différents (y compris un caractère avec un _), on enlève un point.

Remarques :
- Dans le cas général, on utilise un tableau pour donner le score de chaque comparaison de caractères deux à deux (voir par exemple l'exercice 126 p 480 dans votre livre).
- On pourrait tout aussi bien utiliser la distance d'édition (dite aussi distance de Levenshtein ou distance d'Ulam). Cette distance compte le nombre minimal de désaccords entre deux alignements, c'est-à-dire qu'on ne compte que les différences. On cherche alors le minimum plutôt que le maximum.

1. Sans tenir compte des accents (ou autres caractères diacritiques), donner les scores des paires manger-mentir, pécheurs-écriture, niche-chien, algorithme-polyrythmie, génome-énorme. Écrire les alignements correspondants.
2. On va utiliser directement une méthode de programmation dynamique pour réaliser cet alignement de séquences. En effet, la méthode par force brute n'est pas facile à trouver (et est bien moins efficace).

Pour cela, on va construire un tableau `scores` où l'on va construire par méthode ascendante tous les scores possibles, en travaillant sur un exemple, comme génome-énorme.
Le tableau aura autant de lignes que l'un des mots, auquel on peut rajouter un _ en début. De même pour le nombre de colonnes, avec le deuxième mot.

	_	E	N	O	R	M	E
_	0						
G			-2				
E							
N							
O							
M					1		
E							

a. Pour initialiser le tableau, on compare le mot « _ » de longueur 1 avec tous les mots possibles. Sur la première ligne :
   - le score « _ »/« _ » est 0 : ce cas ne peut pas advenir.
   - le score « _ »/« E » est -1.
   - le score « _ »/« EN » est -2 : on rajoute un _ au premier mot, on calcule le score de « __ »/« EN ».
   - etc.

   Compléter la première ligne et la première colonne de ce tableau
b. A quelle comparaison correspond le -2 déjà présent dans le tableau (ligne G, colonne L) ? Idem avec le 1.
c. Pour compléter le tableau, on va se placer « au milieu », en rajoutant une lettre à un ou deux des mots dont le score est déjà calculé
   On veut calculer le score de « GENO »/ «ENOR » à partir de mots plus petits.
   i. Dans quelle case est-ce score ? De quelles cases du tableau peut-on parvenir ? Quelles sont les comparaisons correspondantes ?
   ii. Vérifiez que le score de « GENO »/ «ENO » est 2, celui de « GEN »/« ENOR » est -1, celui de « GEN »/« ENO » est 0.
   iii. On représente la partie du tableau où se passe le calcul

	O	R
N	GEN_/_ENO **0** ↘	GEN_/ENOR **-1** ↓
O	GENO/_ENO **2** →	GENO / ENOR **??**

   Calculer l'évolution du score, arrivant à GENO/ENOR, en partant de chacune des trois cases pouvant être à l'origine de cette comparaison. Quelle est la valeur finalement retenue ? Rajouter les _ manquants éventuellement.
   iv. Reprendre les questions i à iii, pour construire la portion de tableau permettant d'obtenir le score de l'alignement GEN/EN. Quelle est la différence avec le cas précédent ?

	↘	↓
	→	GEN / EN **??**

   v. Finir de compléter le tableau. Que vaut le score d'alignement et où se trouve-t-il ?
d. Programmer la méthode précédente en Python. Donner la complexité de l'algorithme.
   Résumé du d. :
   - La 1ère ligne et 1ère colonne comprennent les valeurs 0, –1, –2, etc.
   - Pour calculer la valeur d'un élément du tableau, on prend le maximum de :
     - Valeur du dessus – 1
     - Valeur de gauche – 1
     - Valeur en diagonale dessus/gauche ± 1 suivant si la lettre à rajouter est identique ou non.
e. Est-il nécessaire de conserver tout le tableau en mémoire pour calculer le score d'alignement ? Justifier. On ne demande pas la programmation.

f. Comment peut-on retrouver l'alignement correspondant au meilleur score ? Expliquer la méthode grâce au tableau, et éventuellement la programmer.
Pour la programmation :
Construire un tableau de « directions » à prendre lors de la reconstruction. On construit ce tableau à partir du tableau des scores comme suit :
- Si tab_scores[i][j] = tab_scores[i - 1][j - 1] ± 1 (suivant si chaine1[i - 1] = ou ≠ chaine2[j - 1])
  Alors le score a été calculé en diagonale descendante, la reconstruction est en diagonale remontante "↖" ;
- Si tab_scores[i][j] = tab_scores[i - 1][j] - 1
  Alors le score a été calculé en descendant, la reconstruction est en remontant " ↑ " ;
- Sinon (tab_scores[i][j] = tab_scores[i][j - 1] - 1)
  Le score a été calculé en allant vers la droite, la reconstruction se fait en allant vers la droite " ←".

Ce tableau permet la construction de l'alignement en partant de la fin des mots à aligner. Il est conseillé d'afficher ce tableau sur un exemple (par exemple pêcheur et écriture), et d'effectuer la reconstruction « à la main » avant d'écrire l'algorithme puis le programme.

Exercice : la ruée vers l'or.

Un mineur d'or se situe face à une falaise. Grâce à son détecteur ultra sophistiqué, il connaît la densité des filons face à lui. Son but est de collecter le maximum d'or. Il est limité par la manière dont il creuse ses galeries : il ne peut pas retourner en arrière, cela entraînerait un effondrement de la mine. Il va donc de la gauche vers la droite, soit tout droit, soit en diagonale en montant, soit en diagonale en descendant. Il peut partir de n'importe quelle hauteur.
Exemple :

$$\begin{bmatrix} 4 & 2 & 0 & 1 \\ 2 & 3 & 5 & 1 \\ 5 & 0 & 1 & 2 \\ 3 & 6 & 1 & 1 \end{bmatrix}$$

Le mineur a récupéré 14 brouzoufs d'or (le brouzouf étant la monnaie du Brouzoufland)

1. Donner une « récolte » optimale pour la matrice ci-dessus et pour la matrice ci-contre.

$$\begin{bmatrix} 6 & 4 & 0 & 0 & 7 & 5 \\ 4 & 8 & 0 & 7 & 5 & 2 \\ 4 & 4 & 6 & 6 & 8 & 2 \\ 5 & 0 & 1 & 3 & 3 & 6 \\ 6 & 6 & 0 & 1 & 7 & 9 \\ 0 & 2 & 8 & 5 & 8 & 7 \end{bmatrix}$$

Récolte ?

2. On appelle $Or(i,j)$ la quantité maximale d'or que le mineur a pu récupérer en arrivant au coefficient $a_{i,j}$ dans la matrice A qui représente le sol derrière la falaise. Rappel : i est l'indice de ligne, j est l'indice de colonne. La matrice A a pour dimension m (nombre de lignes) et n (nombre de colonnes). Le but de cette question est d'écrire un algorithme en programmation dynamique pour optimiser la quantité d'or récoltée au final, notée *Or_max*.
   a. Que vaut $Or(i,0)$ (en première colonne) ?
   b. Comment trouve-t-on la quantité maximale d'or extractible ?
   c. On suppose que $j \neq 0$. D'où peut provenir le mineur lorsqu'il est en $a_{i,j}$? On donnera la réponse en exprimant les différentes possibilités, qui sont des coefficients de la matrice, en fonction de i et j.
   d. En déduire une formule qui donne $Or(i,j)$ en fonction de plusieurs valeurs de $Or(?,j-1)$. On rappelle que $Or(i,j)$ est la valeur maximale. *Appeler le professeur pour vérification de la formule.*
   e. Écrire un algorithme donnant la quantité maximale d'or que le mineur peut extraire. On essaiera de minimiser la mémoire utilisée.

f. Compléter l'algorithme précédent en reconstruisant le « tracé de la mine », c'est à dire les coordonnées (i, j) des coefficients de la matrice par lesquels le mineur est passé. *On partira de la fin, en utilisant uniquement Or_max et A. On écrira l'algorithme sur papier*
g. Donner la complexité temporelle et spatiale de votre algorithme.
h. Programmer ces algorithmes.

3. Complément pour ceux qui vont vite :
Cette fois-ci, le mineur part du sol (le haut de la matrice) et creuse soit horizontalement, soit verticalement, vers le bas uniquement (vous pouvez aussi essayer sans la contrainte d'aller vers le bas). Il a de plus une contrainte supplémentaire : il ne peut pas creuser dans plus de 3 cases/secteurs/coefficients adjacents (on ne peut pas creuser un carré complet de 4 cases). Résoudre le problème de la récolte maximale. On pourra réfléchir au problème avec les exemples suivants :

$$\begin{bmatrix} 6 & 4 & 0 & 0 & 7 & 5 \\ 4 & 8 & 0 & 6 & 5 & 8 \\ 4 & 4 & 6 & 6 & 8 & 2 \\ 5 & 0 & 1 & 1 & 6 & 6 \\ 6 & 6 & 1 & 4 & 1 & 0 \\ 0 & 2 & 8 & 5 & 8 & 0 \end{bmatrix} \qquad \begin{bmatrix} 2 & 5 & 8 & 1 & 6 & 0 \\ 2 & 7 & 6 & 7 & 8 & 7 \\ 2 & 5 & 8 & 7 & 3 & 5 \\ 7 & 1 & 2 & 4 & 2 & 6 \\ 3 & 5 & 2 & 7 & 0 & 8 \\ 5 & 8 & 5 & 6 & 0 & 1 \end{bmatrix}$$