

Algorithmique et structures de données

Chapitre 11 – La programmation dynamique

Samar MOUCHAWRAB, PhD Eng

2A Cycle Préparatoire – Semestre 1
2017/2018

Introduction

La programmation dynamique est une méthode de conception d'algorithme qui peut être utilisée lorsque la solution à un problème peut être considérée comme le résultat d'une séquence de décisions.

Cette méthode consiste à résoudre un problème en le décomposant en sous-problèmes, puis à résoudre les sous-problèmes, des plus petits aux plus grands en stockant les résultats intermédiaires.

Le concept a été introduit au début des années 1950 par Richard Bellman pour résoudre des problèmes d'optimisation. L'efficacité de cette méthode repose sur le principe d'optimalité énoncé Monsieur Bellman : « *toute politique optimale est composée de sous-politiques optimales* ».

Introduction

La programmation dynamique est une technique algorithmique qui est généralement basée sur une **formule récurrente** et **un (ou plusieurs) états de départ**. Une sous-solution du problème est construite à partir des solutions des sous-problèmes.

Pour mieux comprendre le concept, nous présenterons des exemples de problèmes avec les solutions proposées.

Problème de retour de monnaie

Etant donné une liste de N pièces, leurs valeurs (V_1, V_2, \dots, V_N) , et la somme totale S . Trouver le nombre minimal de pièces dont la somme est S (on peut utiliser autant de pièces de monnaie que l'on veut), ou signaler qu'il n'est pas possible de sélectionner des pièces de manière à ce qu'elles correspondent à S .

Prenons un exemple:

Soient des pièces avec les valeurs 1, 3 et 5 et la somme S qui est égale à 11. Combien de pièces de chaque valeur faut-il choisir pour avoir la somme S avec le nombre minimal de pièces ?

La solution est:

Il faut choisir deux pièces de 5 et une pièce de 1

Mais comment trouver cette solution en utilisant un algorithme ?

Solution avec Dynamic Programming (DP)

Commençons maintenant la construction d'une solution DP:

Tout d'abord, nous devons trouver un état pour lequel une solution optimale est trouvée et à l'aide de laquelle nous pouvons trouver la solution optimale pour l'état suivant.

Que signifie un "état"?

C'est une façon de décrire une situation, une sous-solution au problème. Par exemple, un état serait la solution pour une somme i , où $i \leq S$. Un état plus petit que l'état i serait la solution pour toute somme j , où $j < i$.

Pour trouver un état i , nous devons d'abord trouver tous les états j plus petits que i ($j < i$). Après avoir trouvé le nombre minimum de pièces qui se résument à i , nous pouvons facilement trouver l'état suivant - la solution pour $i + 1$.

Solution avec Dynamic Programming (DP)

Comment pouvons-nous trouver cet état i (sous-solution pour la somme S) ?

C'est simple. Si on veut trouver la solution pour la somme i , alors pour chaque pièce j tel que $V_j \leq i$, identifier le nombre minimum de pièces trouvées pour la somme de $i - V_j$ (nous l'avons déjà trouvé précédemment car on procède du bas en haut).

Que ce nombre soit m .

Si $m + 1$ est inférieur au nombre minimum de pièces déjà trouvées pour la somme courante i , nous écrivons le nouveau résultat pour celle-ci.

Solution avec Dynamic Programming (DP)

Tout d'abord nous notons que **pour l'état 0 (somme 0)** nous avons trouvé **une solution** avec un nombre minimum de **0 pièces**.

Nous passons ensuite à la somme 1. Tout d'abord, nous notons que nous n'avons pas encore trouvé de solution pour cet état (**on initialise la solution de cet état à infini**).

On peut constater que **seule la pièce 1 est inférieure ou égale à la somme courante**.

En l'analysant, on voit que pour la somme $1 - V_1 = 0$ nous avons une solution avec 0 pièces. Parce que nous ajoutons une pièce à cette solution, nous aurons une solution avec 1 pièce pour la somme 1. C'est la seule solution trouvée pour cette somme. Nous l'écrivons (sauvegardons). Puis on passe à l'état suivant - somme 2.

Solution avec Dynamic Programming (DP)

On constate encore que la seule pièce inférieure ou égale à cette somme est la première pièce, ayant la valeur 1. La solution optimale trouvée pour la somme $(2-1) = 1$ est la pièce 1.

Cette pièce de monnaie 1 plus la première pièce fera la somme de 2, et ainsi pour obtenir la somme de 2 il nous faut 2 pièces de monnaie de valeur 1. C'est la meilleure et la seule solution pour la somme 2.

Maintenant, nous procédons à la somme 3. Nous avons maintenant 2 pièces qui doivent être analysées - la première et la seconde, ayant des valeurs de 1 et 3. Pour la première, il existe une solution pour la somme 2 ($3-1$) et nous pouvons donc en construire une solution pour la somme 3 en y ajoutant la première pièce. Parce que la meilleure solution pour la somme 2 que nous avons trouvée a 2 pièces, la nouvelle solution pour la somme 3 aura 3 pièces de 1 chacune.

Solution avec Dynamic Programming (DP)

Maintenant, prenons la deuxième pièce avec une valeur égale à 3. La somme pour laquelle cette pièce doit être ajoutée pour faire 3, est 0 ($3 - 3$).

Nous savons que la somme 0 est composée de 0 pièces. Ainsi, nous pouvons faire une somme de 3 avec une seule pièce qui est une pièce de 3. Cette solution est meilleure que la solution trouvée précédemment pour la somme 3, qui était composée de 3 pièces.

Alors, nous mettons à jour la solution pour l'état 3 et nous la marquons comme ayant seulement 1 pièce.

On continue de la même façon avec la somme 4, et nous obtenons une solution de 2 pièces de monnaie - $1 + 3$. Et ainsi de suite ...

Solution avec Dynamic Programming (DP)

Le tableau suivant résume les sous-solutions des états jusqu'à trouver la solution pour la somme $S = 11$

| Somme (état) | Min. Nb. de pièces | Valeur de pièce ajoutée à une somme plus petite (celle entre parenthèses) |
|--------------|--------------------|---|
| 0 | 0 | - |
| 1 | 1 | 1 (0) |
| 2 | 2 | 1 (1) |
| 3 | 1 | 3 (0) |
| 4 | 2 | 1 (3) |
| 5 | 1 | 5 (0) |
| 6 | 2 | 3 (3) |
| 7 | 3 | 1 (6) |
| 8 | 2 | 3 (5) |
| 9 | 3 | 1 (8) |
| 10 | 2 | 5 (5) |
| 11 | 3 | 1 (10) |

Solution avec Dynamic Programming (DP)

```
#include <stdio.h>

void main() {
    int S, Min[1000], N, V[N];

    puts("Entrer la somme: ");
    scanf("%d", &S);
    puts("Entrer le nombre de pieces: ");
    scanf("%d", &N);
    puts("saisir les valeurs des pieces");
    for (int j=0; j< N; j++) scanf("%d", &V[j]);

    //Initialisation à l'infini, on suppose infini = 10000
    for (int i = 1; i <= S; i++) Min[i] = 10000;
    Min[0] = 0;
    for (int i=1; i<= S; i++)
        for (int j=0; j<N; j++)
            if (V[j] <= i && (Min[i-V[j]] + 1) < Min[i])
                Min[i] =Min[i-V[j]] + 1;

    printf("\nNombre de pieces minimum pour avoir la somme de %d
        est : %d", S, Min[S]);
}
```

Problème de la plus longue sous-séquence commune (LCS – Longest Common sequence)

Le problème de la plus longue sous-séquence commune consiste à déterminer la (ou les) plus longue chaîne de caractères qui est sous-chaîne de deux chaînes de caractères.

Exemple, la plus longue sous-séquence commune à « ABABC » et « ABCBA » est la chaîne « ABC » de longueur 3. Les autres sous-chaînes communes telles que « AB », « BC » et « BA » sont plus courtes.

Une sous-séquence est une séquence qui apparaît dans le même ordre relatif mais pas nécessairement contigu. Exemple: pour les séquences « abcdefg » et « abxdfg », les sous-séquences suivantes sont communes : « a », « b », « d », « f », « g », « ab », « df », « dfg », « abd », « abdfg ». La plus longue sous-séquence est « abdfg »

Solution avec Dynamic Programming (DP)

Pour comprendre la solution, prenons un exemple de deux séquences L1 de longueur m et L2 de longueur n : L1 = « AGGTAB » et L2 = « GXTXAYB » et essayons de trouver la sous-séquence la plus longue.

Deux cas à prendre en considération:

1 – Les derniers caractères sont égaux : dans ce cas là incrémenter LCS de 1 et vérifier les sous-chaînes L1[m – 1] et L2[n – 1]
Dans l'exemple ci-dessus les deux derniers caractères sont les mêmes donc $\text{LCS}(L1, L2) = \text{LCS}(\text{AGGTA}, \text{GXTXAY}) + 1$

2 – Les derniers caractères ne sont pas égaux: dans ce cas, trouver le maximum de $\text{LCS}(L1[m - 1] \text{ avec } L2[n])$ et $\text{LCS}(L1[m] \text{ avec } L2[n - 1])$
Continuons le calcul ci-dessus $\text{LCS}(\text{AGGTA}, \text{GXTXAY}) = \max (\text{LCS}(\text{AGGT}, \text{GXTXAY}), \text{LCS}(\text{AGGTA}, \text{GXTXA}))$

Solution avec Dynamic Programming (DP)

Construisons une matrice LCS qui contiendra la longueur de la sous-séquence la plus longue de chaque couple de sous-séquences de L1 et L2. A noter qu'on utilise ϕ pour représenter une chaîne vide. On initialise la première ligne et la première colonne à 0 car pas de sous-séquences communes avec la chaîne vide.

Appliquons maintenant l'algorithme présenté sur le slide précédent en commençant par les cases en haut à gauche.

Si on compare une lettre en colonne j avec une lettre en ligne i , si elles sont égales alors la case $(i, j) = 1 +$ la valeur de la case $(i - 1, j - 1)$; celle qui la précède en diagonale.

Si les lettres ne sont pas égales, alors la case $(i, j) = \text{maximum}(\text{case}(i, j-1), \text{case}(i-1, j))$. Ce sont les cases avant en ligne et plus haut en colonne.

Solution avec Dynamic Programming (DP)

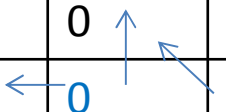
Appliquons cet algorithme sur les chaînes choisies pour l'exemple

| LCS | ϕ | A | G | G | T | A | B |
|--------|--------|---|---|---|---|---|---|
| ϕ | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| G | 0 | | | | | | |
| X | 0 | | | | | | |
| T | 0 | | | | | | |
| X | 0 | | | | | | |
| A | 0 | | | | | | |
| Y | 0 | | | | | | |
| B | 0 | | | | | | |

Solution avec Dynamic Programming (DP)

A est différent de G, donc on choisit le maximum des cases avant et plus haut, donc $\text{maximum}(0,0) = 0$

| LCS | ϕ | A | G | G | T | A | B |
|--------|--------|---|---|---|---|---|---|
| ϕ | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| G | 0 | 0 | 1 | | | | |
| X | 0 | | | | | | |
| T | 0 | | | | | | |
| X | 0 | | | | | | |
| A | 0 | | | | | | |
| Y | 0 | | | | | | |
| B | 0 | | | | | | |



G égal G, donc on ajoute 1 à la valeur en diagonale

Solution avec Dynamic Programming (DP)

Et ainsi de suite jusqu'à remplir le tableau et trouver la longueur de la plus longue sous-séquence commune.

| LCS | ϕ | A | G | G | T | A | B |
|--------|--------|---|---|---|---|---|---|
| ϕ | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| G | 0 | 0 | 1 | 1 | 1 | 1 | 1 |
| X | 0 | 0 | 1 | 1 | 1 | 1 | 1 |
| T | 0 | 0 | 1 | 1 | 2 | 2 | 2 |
| X | 0 | 0 | 1 | 1 | 2 | 2 | 2 |
| A | 0 | 1 | 1 | 1 | 2 | 3 | 3 |
| Y | 0 | 1 | 1 | 1 | 2 | 3 | 3 |
| B | 0 | 1 | 1 | 1 | 2 | 3 | 4 |

Solution avec Dynamic Programming (DP)

Cette matrice nous permet aussi d'identifier la plus longue sous-séquence commune. Il suffit de commencer par le coin droit en bas, si la valeur dans la case n'est pas le maximum des valeurs des cases à côté (avant et en haut) donc cette case correspond à un caractère de la sous-séquence commune. On note le caractère et on avance diagonalement vers le haut.

Dans le cas contraire, on ne choisit pas le caractère, et on avance en suivant le maximum des cases à côté.

Si dans ce cas, les deux cases sont égales, on choisit l'une ou l'autre pour avancer.

En arrivant à 0, on arrête. La plus longue sous-séquence est identifiée.

Solution avec Dynamic Programming (DP)

La plus longue sous-séquence commune est : **GTAB**

| LCS | ϕ | A | G | G | T | A | B |
|----------|--------|----------|----------|----------|----------|----------|----------|
| ϕ | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| G | 0 | 0 | 1 | 1 | 1 | 1 | 1 |
| X | 0 | 0 | 1 | 1 | 1 | 1 | 1 |
| T | 0 | 0 | 1 | 1 | 2 | 2 | 2 |
| X | 0 | 0 | 1 | 1 | 2 | 2 | 2 |
| A | 0 | 1 | 1 | 1 | 2 | 3 | 3 |
| Y | 0 | 1 | 1 | 1 | 2 | 3 | 3 |
| B | 0 | 1 | 1 | 1 | 2 | 3 | 4 |

Solution avec Dynamic Programming (DP)

```
int max (x, y) {
    if (x > y) return x;
    else return y;
}

int lcs (char *X, char *Y, int m, int n) {
    int L[m+1][n+1];
    int i, j;

    for (i=0; i<=m; i++) {
        for (j=0; j<=n; j++) {
            if (i==0 || j==0)
                L[i][j] = 0;
            else if (X[i-1] == Y[j-1])
                L[i][j] = L[i-1][j-1] + 1;
            else
                L[i][j] = max(L[i-1][j], L[i][j-1]);
        }
    }
    return L[m][n];
}
```

Autres problèmes connus résolus avec DP

Parmi les problèmes connus résolus avec la programmation dynamique on peut citer:

- Plus court chemin
- Gestion de stock
- Problème du sac à dos
- Plus longue séquence ascendante