Programmation dynamique

I. Les principes de la programmation dynamique

La programmation dynamique est une méthode pour résoudre des problèmes d'optimisation. Plusieurs méthodes ont déjà été vues :

- *la force brute* : on essaye toutes les solutions possibles.
- *le principe « diviser pour régner »* : il consiste à diviser le problème en deux sous-problèmes distincts, que l'on traite séparément et dont on réunit ensuite les solutions. Il conduit le plus souvent à des programmes récursifs. Exemples : la recherche dichotomique, le tri fusion, le tri rapide, la multiplication de matrices (algorithme de Strassen), la multiplication de polynômes (algorithme de Karatsuba), la transformée de Fourier rapide, etc..
- *l'heuristique gloutonne* : cette méthode consiste à résoudre un problème étape par étape en cherchant à chaque étape la meilleure solution, et en ne changeant pas ce choix ensuite. Le problème est que cette optimisation locale ne conduit pas forcément à la fin à une optimisation globale. Exemples : rendu de monnaie, problème du sac à dos, problème du voyageur de commerce, etc...

La *programmation dynamique* est une méthode permettant d'obtenir une solution optimale à un problème à partir de solutions optimales à des sous-problèmes qui ne sont pas indépendants (c'est-à-dire qui apparaissent plusieurs fois) ; là où la récursivité résoudrait le même problème plusieurs fois, ici, les solutions des sous-problèmes seront stockées pour être réutilisées quand cela sera utile.

Il y a pour cela deux façons de faire :

- *approche descendante* (ou top-down) : on utilise un algorithme récursif, dans lequel la résolution des problèmes plus petits est faite au travers des appels récursifs successifs. On stocke alors au fur et à mesure les résultats déjà calculés (c'est la *mémoïsation*) et on teste lors de chaque appel récursif si le résultata déjà été calculé. Ce stockage des résultats peut être fait à l'aide d'un dictionnaire.
- *approche ascendante* (ou bottom-up) : on comence par résoudre les problèmes les plus petits puis on itère les calculs, avec là aussi stockage des résultats intermédiaires.

Pour illustrer cela, avant de voir des exemples plus élaborés, reprenons l'exemple de la suite de Fibonacci, définie par :

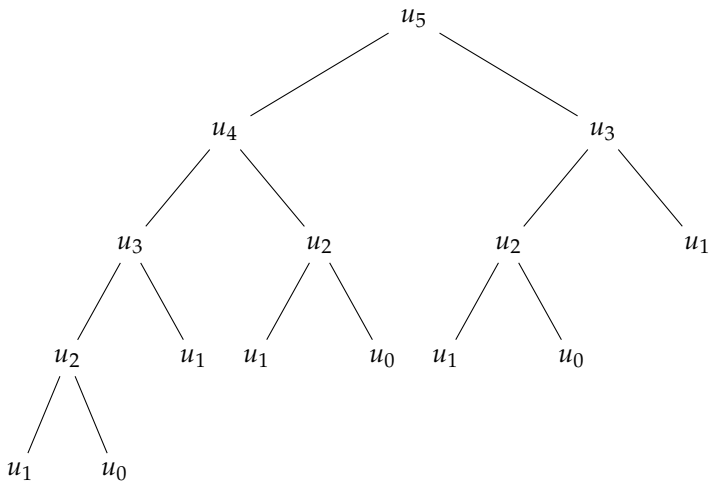
$$u_0 = u_1 = 1 \quad \text{et} \quad \forall n \in \mathbb{N},\ u_{n+2} = u_{n+1} + u_n\,.$$

- *1ère méthode : force brute*

  On peut considérer le programme récursif suivant :

```
def fibo1(n):
    if n < 2:
        return 1
    return fibo1(n-1) + fibo1(n-2)
```

  Voici un arbre qui représente l'exécution de ce programme pour $n = 5$.



Pour calculer u_5, il faut d'abord calculer u_4 et u_3 . Or pour calculer u_4, il faut calculer u_3 etc... Puisque les appels récursifs sont indépendants, la valeur de u_2 par exemple va être calculée 3 fois. Pour des valeurs de n supérieures, on s'aperçoit qu'un grand nombre de valeurs des u_i sont calculées plusieurs fois.

Plus précisément, notons $A(n)$ le nombre d'additions effectuées par la version récursive. $A(n)$ vérifie :

$$A(0) = A(1) = 0 \quad \text{et} \quad \forall n \geqslant 2,\ A(n) = A(n-1) + A(n-2) + 1\,.$$

La suite a_n de terme général $a_n = A(n) + 1$ vérifie donc les relations :

$$a_0 = a_1 = 1 \quad \text{et} \quad \forall n \geqslant 2,\ a_n = a_{n-1} + a_{n-2}\,,$$

ce qui permet de trouver facilement (récurrence linéaire d'ordre 2) :

$$\forall n \in \mathbb{N},\ A(n) = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^{n+1} - \left(\frac{1-\sqrt{5}}{2}\right)^{n+1}\right) + 1\,.$$

On voit donc que $A(n) \underset{n\to+\infty}{\sim} \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^{n+1}$ (avec $\frac{1+\sqrt{5}}{2} \approx 1,618$) : la complexité temporelle est exponentielle.

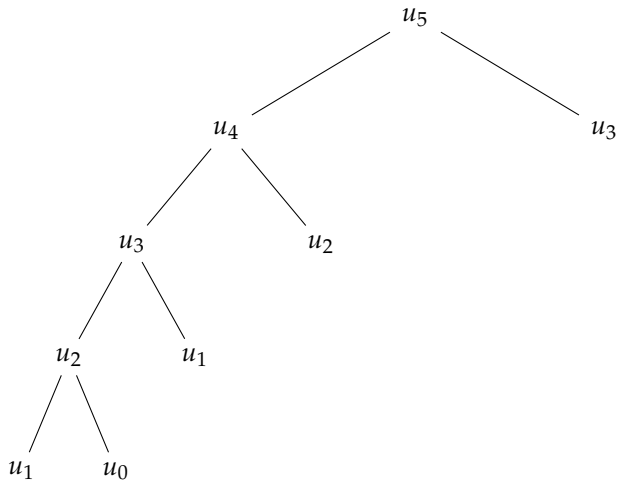
- *2ème méthode : programmation dynamique, approche descendante*

  On écrit encore un programme récursif, mais avec stockage des calculs déjà faits dans un dictionnaire (mémoïsation). Cela donne le programme ci-dessous :

```
dico = {0:1 , 1:1}
def fibo2(n):
    if not n in dico:
        dico[n] = fibo2(n-1) + fibo2(n-2)
    return dico[n]
```

  (on notera que ce code suppose l'emploi d'une variable globale, ce qui est en général à éviter...).

  L'arbre d'exécution est ici beaucoup plus simple :



  À chaque fois, seul le premier appel à `fibo2` conduit à un appel récursif, le second fera appel à une valeur déjà calculée et stockée dans le dictionnaire.

  La complexité temporelle de cet algorithme est donc en $O(n)$, au détriment de la complexité spatiale (comme cela est souvent le cas).

- *3ème méthode : programmation dynamique, approche ascendante*

  On calcule ici les termes de la suite au fur et à mesure à partir des premiers termes, exactement comme on le fait à la main :

```python
def fibo3(n):
    suite = [1, 1]
    while len(suite) <= n:
        suite.append( suite[-1] + suite[-2] )
    return suite[-1]
```

La complexité temporelle est en $O(n)$, ainsi que la complexité spatiale. Les performances sont donc théoriquement semblables à celle de `fibo2`, mais elles sont légèrement meilleures dans la pratique car il n'y a pas à gérer ici des appels récursifs successifs.

Enfin, on peut noter que l'on peut réduire la complexité spatiale à $O(1)$; en effet, il n'est pas nécessaire de stocker tous les termes de la suite, seuls les deux derniers sont importants pour le calcul. Cela donne le programme suivant :

```python
def fibo4(n):
    precedent = 1
    actuel = 1
    for i in range(n-1):
        suivant = precedent + actuel
        precedent, actuel = actuel, suivant
    return actuel
```

EXERCICE 1

Écrire un programme de calcul des coefficients binomiaux $\binom{n}{p}$ en utilisant la formule du triangle de Pascal :

$$\binom{n}{p} = \binom{n-1}{p-1} + \binom{n-1}{p}$$

de trois façons différentes :

1. Force brute (programme récursif)
2. Programmation dynamique approche ascendante, en utilisant un tableau T à deux dimensions, de taille $(n+1)\times(p+1)$, qui sera progressivement rempli en commençant par les coefficients binomiaux les plus petits.
3. Programmation dynamique approche descendante, en utilisant un programme récursif et un dictionnaire pour la mémoïsation.

On supposera (sans le vérifier) que n et p sont bien des entiers naturels, et que $p \leqslant n$.

On calculera à chaque fois la complexité temporelle et spatiale de l'algorithme.

II. Quelques problèmes classiques

II.1. Découpage d'une barre

Il s'agit de calculer le prix maximum que l'on peut tirer de la vente d'une barre de métal de longueur n (en centimètres) lorsqu'on la découpe en morceaux de longueurs inégales (qui seront un nombre entier de centimètres). Le prix de vente p_i d'un morceau de longueur i centimètres est supposé connu ; on supposera la suite (p_i) strictement croissante (ce n'est pas du tout obligatoire, mais cela conduit à des résultats plus cohérents).

Si l'on note u_n le nombre de façons de couper une barre de n centimètres, sachant que l'on peut couper à la distance i centimètres à partir de l'extrémité gauche, ou ne pas couper du tout, on a, pour $n \geqslant 2$:

$$u_n = u_{n-1} + u_{n-2} + \cdots + u_1 + 1$$

ce qui permet d'obtenir par récurrence : $u_n = 2^{n-1}$. Une méthode par force brute va donc vite atteindre ses limites !

EXERCICE 2

1. On note s_n le prix maximum que l'on peut obtenir d'une barre de longueur n, et on pose $s_0 = 0$. Démontrer la relation :
$$s_n = \max_{i\in[\![1;n]\!]}(p_i + s_{n-i}).$$
2. En déduire un programme récursif par force brute qui calcule s_n.
3. *Plus difficile :* Raffiner le programme précédent pour qu'il fournisse aussi la suite des découpes qu'il faut faire pour obtenir s_n.

Le programme ci-dessus est de complexité $O(2^n)$, ce qui est inacceptable. De plus on remarque qu'il y a chevauchement des sous-problèmes, la plupart des valeurs des s_k étant calculées plusieurs fois.

EXERCICE 3

1. Résoudre le problème à l'aide d'une programmation dynamique descendante et mémoïsation.
2. Résoudre le problème à l'aide d'une programmation dynamique ascendante.
3. Montrer que les deux programmes précédents sont de complexité $O(n^2)$.

II.2. Le problème du sac à dos

L'étude de ce problème va permettre de comparer la programmation gloutonne (vue l'an dernier) avec la programmation dynamique :
Le problème est le suivant :

> Étant donnés n objets de valeurs $v_1, \dots, v_n$ et de poids $p_1, \dots, p_n$, comment remplir un sac à dos avec ces objets en maximisant leur valeur, en sachant que le poids maximum des objets transportés est inférieur ou égal à une certaine valeur p_{max} ?

- *Approche gloutonne*

  La méthode gloutonne consiste à chaque étape à choisir un objet dont le rapport $\frac{\text{valeur}}{\text{poids}}$ est maximal, et à l'ajouter au contenu du sac tant que le poids maximum n'est pas dépassé. Cette méthode donne en général un résultat très acceptable bien que pas toujours optimal.

EXERCICE 4

Programmer cette méthode.

Pour la tester, on créera un ensemble de N objets (par exemple $N = 100$), chacun ayant une valeur et un poids choisis au hasard.

- *Approche dynamique*

  On essaye ici de chercher une relation de récurrence entre les solutions optimales à différentes étapes successives. Pour cela, notons $f(k,p)$ la valeur maximale qu'il est possible d'obtenir avec les k premiers objets, pour un poids total de p.
  Si l'objet d'indice k fait partie de cette solution, alors $f(k,p) = v_k + f(k-1, p-p_k)$ (et nécessairement $p_k \leqslant p$), sinon $f(k,p) = f(k-1,p)$. On peut donc résumer, dans tous les cas :
$$f(k,p) = \begin{cases} \max\left(v_k + f(k-1,p-p_k), f(k-1,p)\right) & \text{si } p_k \leqslant p \\ f(k-1,p) & \text{sinon .} \end{cases}$$

EXERCICE 5

En utilisant cette relation, écrire un programme qui donne la solution optimale au problème :

1. en utilisant l'approche dynamique ascendante : on remplira alors au fur et à mesure un tableau T de taille $(N+1)\times(p_{max}+1)$ (où N est le nombre d'objets), contenant les valeurs des $f(k,p)$ pour $k \in [\![0\,;N]\!]$ et $p \in [\![0\,;p_{max}]\!]$ (en posant $f(0,p) = f(k,0) = 0$. Le résultat cherché sera alors $f(N,p_{max})$.
2. en utilisant l'approche dynamique descendante : on utilisera alors un programme récursif et un dictionnaire pour stocker les valeurs des $f(k,p)$.

Les deux programmes devront renvoyer la valeur maximale trouvée ainsi que la façon d'y parvenir.

II.3. La distance d'édition (ou distance de Levenshtein)

La distance d'édition entre deux chaînes de caractères permet de quantifier la ressemblance entre deux chaînes de caractères : elle désigne le nombre minimal d'opérations élémentaires à faire pour passer d'une chaîne `ch1` à une chaîne `ch2`. Une opération élémentaire est : une *insertion*, une *suppression* ou un *remplacement* d'un des caractères de la chaîne `ch2`.

• *Programmation récursive*

En notant `n1` la longueur de `ch1` et `n2` celle de `ch2`, la distance `d(ch1, ch2)` peut être définie de façon récursive par :

- si `n1 == 0`, alors `d(ch1, ch2) = n2`
- sinon, si `n2 == 0`, alors `d(ch1, ch2) = n1`
- sinon si `ch1[0] == ch2[0]` alors `d(ch1, ch2) = min( d(ch1[1:], ch2[1:]), 1 + d(ch1[1:], ch2), 1+d(ch1, ch2[1:])`
- sinon `d(ch1, ch2) = 1 + min(d(ch1[1:], ch2), d(ch1, ch2[1:]), d(ch1[1:], ch2[1:]))`

Explication :

Les 2 premiers cas sont clairs.
La deuxième égalité correspond aux cas où :

1) on n'a rien fait et transformé `ch1[1:]` en `ch2[1:]`
2) on a supprimé `ch1[0]` puis transformé `ch1[1:]` en `ch2`
3) on a inséré `ch2[0]` au début de `ch1` après avoir transformé `ch1` en `ch2[1:]`

La troisième égalité correspond aux cas où : 1) on a supprimé `ch1[0]` 2) on a inséré `ch2[0]` au début de `ch1` 3) on a remplacé `ch1[0]` par `ch2[0]`

EXERCICE 6

En utilisant cette relation, écrire un programme qui calcule la distance entre les deux chaînes :

1. en utilisant un simple programme récursif.
2. en utilisant l'approche dynamique ascendante : on utilisera alors un programme itératif et un tableau pour stocker les valeurs des $D[i,j]$, en notant $D[i,j]$ la distance entre les chaînes `ch1[0:i]` et `ch2[0:j]` pour $(i,j) \in [\![0\,;n1]\!] \times [\![0\,;n2]\!]$ (D est donc un tableau de taille $(n1+1)\times(n2+1)$). Pour trouver une relation de récurrence entre les $D[i,j]$, on s'inspirera du raisonnement précédent.