Algorithmique Avancée
Partie 2 - Programmation Dynamique

Nicolas Bousquet

August 22, 2022

Le but de cette partie du cours est d'introduire et d'illustrer le principe de la programmation dynamique. Nous allons voir une batterie d'exemples assez simples de programmation dynamique sur différents types de problèmes (rendu de monnaies, graphes, mots). En général plus la structure est complexe plus l'algorithme de programmation dynamique est complexe. On va donc, dans le cas des graphes se concentrer sur des graphes simples pour éviter des analyse inutilement compliquées.

1 Suite récurrentes

Pour illustrer la programmation dynamique, le premier exemple le plus simple est sans doute celui des suites récurrentes. La suite de Fibonacci est la suite définie par $u_0 = 0, u_1 = 1$ et les termes suivants sont définis parfois

$$u_{n+1} = u_n + u_{n-1}.$$

Comment calculer le n-eme terme de cette suite avec un programme? Il suffit de calculer u_n et de calculer u_{n-1} et de faire leur somme. Donc il suffit de calculer récursivement les deux valeurs et de les sommer pour obtenir la valeur finale (Algorithme 1).

Procedure 1 FibNaif

Input: Un entier n.
Output: La valeur u_n
  if $n = 0$ **then**
    Renvoyer 0
  else if $n = 1$ **then**
    Renvoyer 1
  else
    $a = FibNaif(n-1)$
    $b = FibNaif(n-2)$
    Renvoyer $a + b$.
  end if

Le problème de cet algorithme, on le voit assez facilement quand on fait "l'arbre de branchement" de notre algorithme est que le nombre d'opérations qu'il va effectuer est en fait exponentiel. Ainsi pour calculer les valeurs 0 et 1, il faudra 0 appels récursifs. Pour calculer la valeur 2, il faudra 2 appels récursifs. Pour calculer la valeur 3, il faudra 4 appels récursifs (2 initiaux puis 2 pour 2 et 0 pour 1). Pour calculer la valeur 4, il faudra $2 + 4 + 2 = 8$ appels récursifs.

n	0	1	2	3	4	5	6	7	8	9
Nb. app. rec.	0	0	2	4	8	14	24	40	66	108

Cette complexité n'est donc pas acceptable pour calculer le n-ème terme de la suite. L'explosion combinatoire est trop forte et on va rapidement être "bloqué". On double presque le nombre d'appels récursifs à chaque étape.

On va donc utiliser une autre approche. On remarque que dans l'arbre de branchement de l'algorithme récursif naïf, on calcule en fait très souvent la même valeur dans différentes branches. Ainsi pour l'appel de $n = 5$, on va calculer la valeur pour $n = 3$ à la fois d'appel de 5 et dans l'appel de 4. On va mettre à profit ce fait pour en fait ne calculer qu'une seule fois chaque valeur. On pourrait par exemple dire que chaque fois qu'on a déjà calculé une valeur, on la stocke pour ne pas avoir à la recalculer. En faisant le bilan de tout ça, on se rend compte qu'il suffit de lancer l'algorithme suivant:

Procedure 2 Fibonacci

Input: Un entier n.
Output: La valeur de u_n.
  Créer un tableau Tde longueur $n + 1$.
  $T[0] = 0$
  $T[1] = 1$
  for $i \leq n$ **do**
    $T[i] = T[i-1] + T[i-2]$
  end for
  Renvoyer $T[n]$

Si on regarde dans le détail ce que l'on a fait, on a "simplement" calculer toutes les valeurs intermédiaires pour trouver la valeur finale. Autrement dit, pour calculer ce que l'on désirait, on a en fait calculer bien plus... On a calculé toutes les valeurs de u_i pour $i \leq n$ et on a utilisé la formule de récurrence pour trouver la valeur de u_n.

1.1 Suites à plusieurs variables

La même méthode fonctionne encore quand on a des suites à plusieurs variables. $\binom{n}{k}$ désigner le nombre de parties de taille k dans un ensemble de taille n. On va avoir besoin des petites formules mathématiques suivantes:

Exercice 1.
1. Montrer que $\binom{n}{0} = 1$ et $\binom{n}{n} = 1$ pour tout n.
2. Montrer que $\binom{n}{k} = \frac{n!}{k!(n-k)!}$.
3. Montrer que $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n}{k-1}$.

On va considérer que $\binom{n}{k}$ est une suite récurrente linéaire à deux variables $u(n, k)$. On peut donc calculer $\binom{n}{k}$ à partir de $u(n-1, k-1)$ et $u(n, k-1)$ d'après l'exercice 1. Comme dans le cas de la formule récurrente de Fibonacci, on peut donc en déduire un algorithme efficace pour calculer cette valeur par récurrence:

Procedure 3 Binomial

Input: Deux entiers n, k avec $k \leq n$ et $n \geq 1$.
Output: La valeur $\binom{n}{k}$.
  Créer un tableau T de taille $n \times k$.
  for $1 \leq i \leq n$ **do**
    $T[i][0] = 1$
  end for
  for $1 \leq i \leq n$ **do**
    for $1 \leq j \leq i$ **do**
      $T[i][0] = T[i-1, j-1] + T[i, j-1]$
    end for
  end for

2 Programmation dynamique

Le principe de la programmation est exactement le même que celui des suites récurrentes linéaires. On désire calculer la n-me valeur en fonction des valeurs précédentes. La différence avec les suites récurrentes est juste qu'il va falloir trouver la formule de récurrence et qu'elle n'est pas donnée immédiatement ! Et dans les cas les plus compliqués, il faut parfois aussi trouver la bonne façon de "faire" cette récurrence.

Take home message: Formule de récurrence = Programmation dynamique.

On a vu dans les parties précédentes que la programmation dynamique pouvait se faire à l'aide d'un tableau (pour les suites récurrentes) ou à l'aide d'un tableau multi-dimensionnel (dans le cas des produits binomiaux). On verra que parfois, on aura des structures plus compliquées hérités de graphes sous-jacents par exemple.

2.1 Rendu de monnaie

On se donne k valeurs de pièces $c_1, \ldots, c_k$ (par exemple $1, 2, 5, 10, 20, 50$ en France). On se demande comment rendre la monnaie de la meilleure façon possible. Imaginons par exemple qu'on veuille rendre 35 euros. L'algorithme glouton consiste à rendre d'abord la plus grande pièce (20) puis dix, puis 5 et on a ensuite fini de rendre la monnaie.

Dans la suite on supposera que les pièces sont classées par ordre croissant (c_1 étant la plus petite valeur et c_k étant la plus grande). On supposera aussi que $c_1 = 1$ pour garantir qu'il est toujours possible de rendre la monnaie.

On va considérer l'algorithme glouton 4.

Procedure 4 Algorithme glouton de rendu de monnaie

Input: Des valeurs de pièces $c_1, \ldots, c_k$. Un entier n.
Output: Un façon de rendre la monnaie pour la valeur n.

```
v = n
T = tableau de taille k initialisé à 0
while v ≠ 0 do
    Soit c_i la plus grande valeur plus petite que v
    T[i] = T[i] + 1 (Rendre une pièce de type c_i en plus)
    d = d − c_i
end while
Renvoyer T (Rendre T[i] pièces de type c_i pour chaque i)
```

On peut naturellement se poser la question suivante: l'algorithme est-il optimal?

Exercice 2. Montrer que l'algorithme glouton n'est pas forcement optimal.

Il n'est pas possible à première vue de déterminer si un système est optimal, mais il existe en fait un algorithme polynomial pour le déterminer...

Exercice 3.
1. Quelle est la complexité de l'algorithme glouton ci dessus?
2. Pouvez vous réduire la dépendance en k?

Comme le rendu n'est pas forcement optimal, on peut se demander quelle est la complexité d'un algorithme optimal. Encore une fois, on peut faire un algorithme récursif qui choisit quelle pièce est rendue en premier et ensuite trouve, pour chaque choix la meilleure solution quand on a rendu cette pièce. Mais on voit que l'on va se retrouver, encore une fois, avec une véritable explosion combinatoire du nombre de cas comme pour la suite de Fibonacci.

L'approche va encore une fois consister à faire un algorithme de programmation dynamique. Toute la question est de savoir comment le conceptualiser. Pour faire ça, on utilise toujours le même canevas:

- Qu'est ce qui est recalculé?

- Peut-on exprimer ce qui est recalculé "facilement" en fonction des valeurs précédentes.

Ici, si on veut rendre n, on voit que l'on a la formule:

$$OptRendu(n) = 1 + \min_{i/c_i \leq n} (n - c_i).$$

Cette formule est correcte mais elle a un désavantage: elle oblige à calculer le minimum de k termes ce qui est un peu compliqué à écrire algorithmiquement. On décide alors de regarder les valeurs $T(i,j)$ qui est la meilleure façon de rendre i avec les j premières pièces. On a alors la relation suivante qui est bien plus simple:

$$T(i,j) = \min\Big(T(i - c_j, j), T(i, j-1)\Big)$$

avec des valeurs des infinis si la première coordonnée est négative (autrement dit, on ne fait pas le min mais prend seulement le second terme si $i - c_j$ est négatif, autrement dit que la j-ème pièce vaut plus que i.

On peut alors calculer à l'aide d'un algorithme de programmation dynamique les valeurs $T(i,j)$ pour tout i, j à l'aide de la formule ci-dessus et du fait que:

- pour tout n, $T(n,1) = n$,
- pour tout j, $T(0,j) = 0$.

On peut alors écrire l'algorithme de programmation dynamique qui permet de calculer le meilleur rendu de pièces possible.

Remarque: On voit ici à la fois des points communs et des différences avec ce qu'on a fait précédemment. Ce qui est commun est une formule de récurrence qui dépend de valeurs "plus faibles" et des "cas de base" pour initialiser notre récurrence. Il y a une différence ici par rapport au suite récurrente qui est que dans le cas des suite récurrente la fonction des valeurs précédentes était "simple" (addition ou multiplication) alors qu'ici on des valeurs plus compliquées (min...).

Quand on veut utiliser un algorithme de programmation dynamique:

- Il faut trouver les variables dont dépend notre solution (ici k, n).
  $\rightarrow$ Dans nos exemples, elles seront assez explicites (mais dans certains cas elles peuvent ne pas l'être).
- Il faut trouver la formule de récurrence. $\rightarrow$ Elle peut être simple, comme dans les suite récurrentes ou plus compliquée (utiliser des min, max, +, -, puissance...etc...; tant qu'elle est facilement calculable).
- Il faut trouver les valeurs de base pour initialiser notre formule de récurrence.

2.2 Problème du sac à dos

Dans le problème du sac à dos, on se donne une capacité P pouvant être contenu dans un sac à dos et k objects $O_1, \ldots, O_k$ tel que chaque objet O_i a une valeur c_i et un poids p_i. Le but est de trouver le sous ensemble d'objet de poids total inférieur ou égal à C et de valeur maximum.

Exercice 4.

1. Montrer que l'algorithme glouton consistant à choisir gloutonnement les objets de valeur maximum peut renvoyer une solution arbitrairement mauvaise.
2. Montrer que l'algorithme glouton consistant à choisir gloutonnement les objets dont le ration valeur sur poids est maximum n'est pas optimal.
3. Montrer que l'algorithme ci-dessus fournit néanmoins une 2-approximation.
4. Donner un algorithme de programmation dynamique qui résout optimalement le problème du Sac à Dos.

3 Programmation dynamique et mots

3.1 Définitions

Étant donné un alphabet Σ, un mot est une suite (possiblement vide) de caractère de Σ. Dans la suite on ne considérera que des mots de longueur fini. Soit A un mot de longueur n. On dénote par $A_{\leq i}$ pour $i \leq n$ le sous mot restreint aux i premières lettres de A. On dénote par $A[i]$ la i-ème lettre de A.

On se donne deux mots A et B longueur n et m. Les deux mots ont le même alphabet Σ et ils sont représentés par des tableaux de taille n. On dit que A est un *sous-mot de* B s'il existe un position j telle que le mot A et $B[j]....B[j+n-1]$ sont les même mots.

On dit que A et B ont une *sous-séquence commune de longueur* r s'il existe deux suites de r positions $i_1, \ldots, i_r$ et $j_1, \ldots, j_r$ qui sont toutes les deux croissantes telles que le mot induit par les positions $a[i_1], \ldots, a[i_r]$ et $b[j_1], \ldots, b[j_r]$ sont les mêmes.

3.2 Plus longue sous-séquence commune

Le but de cette partie est de montrer qu'on peut calculer la plus longue séquence commune à l'aide d'un algorithme de programmation dynamique. La plus longue séquence commune étant un mot de longueur maximale qui est une sous-séquence commune aux deux mots A et B. Autrement dit le plus long mot M que l'on peut obtenir en supprimant des lettres de A et que l'on peut obtenir en supprimant des lettres de B.

On dénote pas $M[i,j]$ la plus longue sous séquence commune du préfixe de A restreint aux i premiers éléments et B restreint aux j premiers éléments.

On peut d'abord noter que l'on a un cas de base simple, si $i = 0$ ou $j = 0$ alors un des deux mots est vide et on n'a pas de séquence commune, autrement dit, pour tout i, j:

$$M[i,0] = M[0,j] = 0.$$

On peut maintenant se concentrer sur la formule de récurrence. On va montrer que M satisfait la formule de récurrence suivante (pour toute paire $i, j \geq 1$.

$$M[i,j] = \begin{cases} \max(1 + M[i-1, j-1]) & \text{si } a_i = b_j \\ \max(M[i-1,j], M[i,j-1]) & \text{sinon.} \end{cases}$$

Si le dernier caractère des deux mots $A_{\leq i}$ et $B_{\leq j}$ est le même alors il existe une plus longue séquence commune qui contient à la fois a_i et b_j.

Exercice 5. Prouvez le !

Donc la plus longue sous-séquence commune contient la plus longue sous séquence commune de $A_{\leq i-1}$ et $B_{\leq j-1}$ plus $A[i]$ et $B[j]$, d'où notre première ligne.

Si $A[i] \neq B[j]$ alors la dernière lettre de A ou la dernière lettre de B n'est pas dans la plus longue chaîne commune de A et de B. Donc on peut "supprimer" une des deux lettres et essayer de trouver la plus longue sous séquence commune soit entre $A_{\leq i}$ et $B_{\leq j-1}$ soit entre $A_{\leq i-1}$ et $B_{\leq j}$, d'où la formule.

3.3 Plus long sous-mot commun

Parfois, retenir une valeur par case de notre programme dynamique (i.e. n'avoir qu'une seule suite récurrente) n'est pas suffisant. Ainsi imaginons qu'on ait deux suite u_n et v_n telles que (i) u_0 et v_0 sont donnés et, (ii) $u_n = f(u_{n-1}, v_{n-1})$ et $v_n = f(u_n, v_{n-1})$ alors on peut, à l'aide d'un algorithme de programmation dynamique calculer la valeur de u_n et v_n.

Pour calculer le plus long sous mot commun on a besoin d'utiliser cette méthode. On dénote par $L[i,j]$ le plus long sous-mot commun à $A_{\leq i}$ et $B_{\leq j}$. On dénote par $M[i,j]$ le plus long sous-mot commun à $A_{\leq i}$ et $B_{\leq j}$ qui se termine à la position i pour A et à la position j pour B.

Exercice 6.

1. Donner les valeurs pour $i = 0$ ou $j = 0$ de M et L.
2. Exprimer $M[i,j]$ en fonction de $M[i-1, j-1]$.

3. Exprimer $L[i,j]$ en fonction de $M[i,j]$, $L[i-1,j]$ et $L[i,j-1]$.
4. En déduire un algorithme de programmation dynamique pour le problème du plus long sous-mot commun.

4 Programmation dynamique sur les graphes

4.1 Ensemble indépendant dans un arbre

Il est très facile de calculer un indépendant maximum dans un graphe.

Exercice 7. Soit T un arbre.

1. Montrer qu'il existe un indépendant maximum qui contient toutes les feuilles de T.
2. En déduire un algorithme polynomial pour calculer un indépendant maximum dans un arbre.

Le but de cette section est de faire un algorithme bien plus difficile pour calculer un ensemble indépendant. On va faire pour ça un algorithme de programmation dynamique. Voilà le principe de l'algorithme:

- On enracine l'arbre en un sommet arbitraire.
- On effectue ensuite un algorithme de programmation dynamique qui étant donné le meilleur indépendant sur les sous arbres enracinés sur les enfants d'un noeud u permet de déduire le meilleur indépendant du sous graphe enraciné en u.

Pour le second point, donnons un peu plus de détails.

Soit T un arbre enraciné en u appelé la *racine*. Autrement dit, pour tout sommet v, les arêtes de l'unique chemin de u vers v sont toutes orientés du début du chemin vers la fin. Un noeud v est un *enfant* de u si uv est une arête orienté. S'il existe un chemin orienté de u vers v alors v est un *descendant* de u. Si uv est un arc alors u est le *parent* de v. Le parent de la racine est la racine elle-même.

Exercice 8. Montrer que chaque noeud a un unique parent mais peut avoir un nombre arbitrairement grand d'enfants.

Supposons maintenant que pour chaque descendant $v_1, \dots, v_\ell$ de u on sait:

- La taille maximum d'un indépendant du sous arbre enraciné en v_i qui ne contient pas v_i,
- La taille maximum d'un indépendant du sous arbre enraciné en v_i qui contient v_i.

Maintenant, on peut en déduire les deux valeurs correspondantes pour le sommet u si on connaît toutes ces valeurs. Comme ces valeurs sont facilement calculable sur les feuilles de l'arbre, on en déduit facilement la valeur sur le graphe tout entier.

Le but des deux prochaines sous sections est de généraliser ce résultat sur deux classes de graphes: les graphes d'intervalles et les graphes de largeur arborescente bornée.

4.2 Indépendant maximum dans les graphes d'intervalles

On va maintenant étudier le problème de l'indépendant maximum dans les graphes d'intervalles. Un *graphe d'intervalle* est un graphe tel que on peut associer un intervalle (a,b) à chaque sommet du graphe et il y a une arête entre deux sommets si et seulement si les intervalles correspondant aux sommets s'intersectent.

Exercice 9. Montrer par récurrence que l'algorithme glouton qui consiste à itérativement prendre le sommet qui se termine le premier et supprimer son voisinage dans le graphe renvoie une solution optimale.

On va proposer un autre algorithme qui calcule un indépendant maximum à l'aide de la programmation dynamique.

4.3 Graphes de largeur arborescente bornée (non couvert en 2022)