# Programmation Dynamique

La programmation dynamique est une technique efficace pour résoudre certains problèmes en les décomposant en problèmes plus simples. La différence majeure avec la technique « diviser pour régner » est que les sous-problèmes se chevauchent, et ne pas prendre en compte cette propriété dupliquerait inutilement leur résolution.

## 1 Présentation

La *programmation dynamique* est donc une technique applicable aux problèmes qui satisfont les propriétés suivantes :

- le problème consiste à trouver une solution *optimale* selon un certain critère (minimalité ou maximalité d'une grandeur) ;
- le problème se décompose en sous-problèmes ;
- la solution optimale se calcule facilement à partir des solutions optimales des sous-problèmes en question ;
- les sous-problèmes se recoupent tout ou partie.

Nous allons considérer quelques exemples de problèmes pour illustrer ce domaine. Il y a bien d'autres problèmes intéressants qui illustrent ce principe, la liste suivante n'est pas exhaustive.

**distance d'édition** (distance de Levenshtein) : étant données deux chaînes de caractères $a$ et $b$, trouver le nombre minimal d'opérations d'éditions qui transforme $a$ en $b$. Cette distance est très pratique pour la correction orthographique et la recherche de mots résistante aux erreurs de frappe. Les opérations d'édition « classiques » sont :

1. insérer un caractère à la position $i$,
2. enlever le caractère à la position $i$,
3. remplacer le caractère à la position $i$ par un autre.

**plus longue sous-séquence commune** : étant données deux séquences d'objets (chaînes de caractères, listes d'entiers, etc.) $a$ et $b$, trouver une séquence $c$ de longueur maximale telle que $c$ soit une sous-séquence de $a$ et $b$ (i.e. obtenue à partir de $a$, respectivement $b$, uniquement en enlevant des éléments). Par exemple pour $a$ = "glace lol" et $b$ = "glass hello" on aura $c$ = "glaelo" (plusieurs solutions de même longueur sont possibles).

**weighted interval scheduling** : choisir un ensemble de tâches (« jobs ») pour maximiser la somme de leur poids. Par exemple, un ministre a une liste de de rendez-vous, réunions et visites à effectuer [1], mais ne peut en traiter qu'une à la fois [2]. Chaque tâche possède trois attributs :

1. une heure de début (un entier),
2. une heure de fin (un entier),
3. un *poids* qui symbolise son importance (un entier aussi en général).

Le but du ministre, qui veut être efficace, est évidemment de maximiser le poids total des tâches qu'il choisit d'effectuer (en choisissant parmi les tâches incompatibles).

**matrix chain multiplication** : on veut calculer un produit de matrices $A_1 \times A_2 \times \cdots \times A_n$. Chaque produit coûte cher, en fonction de la taille des matrices. Le produit est associatif. Par exemple si les dimensions sont $A : 10\times30, B : 30\times5, C : 5\times60$, alors $A\times(B\times C)$ prendra $10\cdot30\cdot60+30\cdot5\cdot60$ opérations élémentaires (produit de 2 nombres, sans compter les sommes), soit 27000 opérations ; $(A\times B)\times C$ ne prendra que $10\cdot30\cdot5+10\cdot5\cdot60 = 4500$ opérations. Il vaut donc mieux parenthéser à gauche dans ce cas.

# 2 Mémoïsation

## 2.1 Par l'exemple : Distance de Levenshtein

Ces algorithmes admettent des solutions récursives. Examinons par exemple le cas de la distance d'édition. Nous avons deux chaînes $a$ et $b$ (de type string), et nous voulons calculer leur distance d'édition. Pour cela nous définissons la fonction $D(i, j)$ qui calcule la distance entre $a_{1\ldots i}$ et $b_{1\ldots j}$ (les préfixes de $a$ et $b$ de longueurs respectives $i$ et $j$), puis on calcule $D(|a|, |b|)$ avec $|a|$ la longueur de $a$ (autrement dit $a = a_{1,\ldots,|a|}$). Cette fonction est récursivement définie par

$$D(i,j) = \begin{cases} j & \text{si } i = 0 \\ i & \text{si } j = 0 \\ \min \begin{pmatrix} D(i-1,j)+1 \\ D(i,j-1)+1 \\ D(i-1,j-1)+(1-\delta_{a_i,b_j}) \end{pmatrix} & \text{sinon} \end{cases}$$

avec $\delta_{x,y}$ le symbole de Kronecker, qui vaut 1 si $x = y$ et 0 sinon. Les cas $D(i-1, j)+1$ et $D(i, j-1)+1$ correspondent respectivement à une déletion d'un caractère dans $a$ et à l'ajout d'un caractère dans $a$. Le troisième cas correspond à éditer le dernier caractère (ou à le laisser tel quel s'il correspond déjà). Plus précisément, si $a_i = b_j$, nul besoin d'éditer, il suffit de calculer la distance de $a_{1\ldots i-1}$ à $b_{1\ldots j-1}$.

1. d'où le terme « avoir un emploi du temps de ministre »...
2. du moins si on suppose que le ministre ne prend pas de drogues.

On voit déjà que comme dans le cas de la version naïve du calcul de la suite de Fibonacci, des appels récursifs vont être dupliqués. Par exemple sur $a = \text{abc}$ et $b = \text{abb}$, on commence par évaluer $D(3,3)$ qui va appeler $D(2,2)$, $D(2,3)$ et $D(3,2)$. Ces deux derniers vont eux-même évaluer $D(2,2)$, d'où une duplication du travail qui ne va qu'empirer quand $i$ et $j$ se rapprochent de 1.
La *mémoïsation* permet de ne pas dupliquer ce travail, en mémorisant le résultat de $D(i,j)$ dans (typiquement) une table de hachage, avec la clé $(i,j)$, et en utilisant ce résultat précalculé quand $D(i,j)$ est demandé à nouveau. Cette table est parfois appelée « cache ». L'implémentation correspondante en Caml est la suivante (attention les chaînes en Caml sont indexées à partir de 0), en utilisant le module hashtbl standard pour les tables de hachage :

```
let distance a b =
    let m = string_length a
    and n = string_length b in
    let cache = hashtbl__new 128 in
    (* fonction auxiliaire *)
    let rec aux i j =
        if i = 0 then j
        else if j = 0 then i
        else
            try hashtbl__find cache (i,j)
            with Not_found ->
                let d1 = aux (i-1) j + 1
                and d2 = aux i (j-1) + 1
                and d3 = aux (i-1)(j-1)
                    + (if a.[i-1] = b.[j-1] then 0 else 1)
                in
                let d = min d1 (min d2 d3) in
                hashtbl__add cache (i,j) d;
                d
    in
    aux m n ;;
```

**Exercice :** Écrire la version non mémoïsée de distance et comparer les performances des deux versions sur des chaînes de longueurs variées. On pourra utiliser les fonctions de mesure du temps du DM2 (tp5) ainsi que des fonctions permettant de créer des chaînes d'une longueur donnée.
**Exercice :** Extraire la fonction aux (en lui ajoutant cache, a et b comme arguments) et utiliser #trace "aux";; pour suivre l'exécution de la fonction sur deux chaînes courtes.
**Exercice :** Évaluer la complexité de distance en fonction des longueurs de $a$ et $b$ (encore une fois on comptera le nombre d'appels récursifs).
**Exercice :** Généraliser cette notion de distance en décidant que la distance est désormais de type float, avec le coût de l'opération de remplacement du caractère $c_1$ par $c_2$ valant $\frac{|c_2 - c_1|}{26}$ (remplacer 'a' par 'z' est donc plus cher que remplacer 'a' par 'b'). On pourra utiliser les fonctions abs_float et int_of_char : char -> int.

## 2.2 Autres exemples

Examinons d'autres sous-problèmes présentés au début. On va chercher une solution récursive qui s'adapte bien à la mémoïsation.

**plus longue sous-séquence commune** : en fait il s'agit d'une version plus simple de la distance d'édition, dans laquelle l'opération de remplacement est interdite. Seule la suppression et l'ajout sont autorisées. En bref, pour des séquences $a$ et $b$, on définit la fonction $S(i,j)$ qui calcule la plus longue sous-séquence commune à $a_{1\ldots i}$ et $b_{1\ldots j}$. Cette séquence est définie récursivement (en notant $\epsilon$ la séquence vide et $\oplus$ la concaténation de séquences) par :

$$S(i,j) = \begin{cases} \epsilon & \text{si } a = \epsilon \text{ ou } b = \epsilon \\ S(i-1,j-1) \oplus a_i & \text{si } a_i = b_j \\ S(i-1,j) & \text{si } |S(i-1,j)| < |S(i,j-1)| \\ S(i,j-1) & \text{sinon.} \end{cases}$$

**matrix chain multiplication** : on suppose qu'on a une séquence de matrices $A_1, A_2, \ldots, A_n$ ($n \geq 2$) et une fonction taille $t$ à valeur dans $\mathbb{N}^2$ qui associe à chaque matrice ses dimensions. On écrira $t_1$ (resp. $t_2$) le premier (resp. deuxième) composant du tuple $t$. Le but est de trouver un parenthésage qui minimise le coût total du produit $\prod_{i=1}^{n} A_i$. Pour cela on considère le problème du coût $c(i,j)$ du produit $\prod_{k=i}^{j} A_k$ ; le problème initial se réduit au calcul de $c(1,n)$. La fonction $c$ est définie sur $(i,j) \in \{1,\ldots,n\}$ avec $i \leq j$, et obéit à la relation récursive suivante :

$$c(i,j) = \begin{cases} 0 & \text{si } i = j \\ t(A_i)_1 \cdot t(A_i)_2 \cdot t(A_j)_2 & \text{si } i+1 = j \\ \min_{k=i}^{j-1} c(i,k) + c(k+1,j) + t(A_i)_1 \cdot t(A_k)_2 \cdot t(A_j)_2 & \text{sinon} \end{cases}$$

On suppose que le produit est possible, c'est-à-dire $\forall i < n, t(A_i)_2 = t(A_{i+1})_1$. L'étape récursive consiste à choisir un $k$ où séparer la séquence en deux, à calculer le coût des deux sous-produits (gauche et droite), et à ajouter le coût du produit des résultats ; on choisit ensuite $k$ afin de minimiser ce coût. Notons que les dimensions de $\prod_{k=i}^{j} A_k$ sont $t(A_i)_1, t(A_j)_2$.

**Exercice :** Implémenter l'algorithme de la plus longue sous-séquence commune sur des listes en Caml, de type `int list -> int list -> int list`. On comparera les performances d'une version naïve avec celles d'une version mémoïsée. Note : plutôt que d'utiliser `@` pour $\oplus$ il vaut mieux calculer le résultat à l'envers avec `::` et le renverser à la fin avec `rev`.

**Exercice** : Implémenter la fonction de coût du problème *matrix chain multiplication*, de type `(int*int) vect -> int`. Cette fonction prend en entrée le tableau qui à $i$ associe $t(A_i)$ (les dimensions de la $i^{\text{ème}}$ matrice). Tester cette fonction sur l'exemple utilisé plus haut.

**Exercice :** (plus avancé) : à l'aide d'un type auxiliaire, par exemple **type** mult = Atomic | Split **of** mult * int * mult ;;, modifier l'algorithme précédent pour qu'il retourne le coût minimal du produit et le parenthésage (sous forme d'arbre binaire dont les nœuds indiquent l'indice du produit à effectuer, et les enfants le parenthésage des sous-expressions à multiplier).

# 3 Approche Ascendante

La mémoïsation est certes très pratique et proche de la définition récursive, mais elle souffre de certains défauts. Par exemple, le coût du stockage dans l'algorithme de Levenshtein atteint $m \cdot n$ (sans compter les coûts constants et le gaspillage de place, l'usage d'une table de hachage n'étant pas non plus complètement gratuit). Pour certains algorithmes on peut utiliser des versions « astucieuses » de la définition, qui calculent *en montant*[3].
L'idée est la suivante : en analysant les dépendances de la fonction récursive à calculer, on trouve[4] un ordre de calcul des sous-problèmes qui garantit qu'un sous-problème a déjà été calculé quand on en a besoin, et on « remonte » ainsi jusqu'au problème principal.
Reprenons l'exemple de la distance de Levenshtein (distance d'édition). La définition récursive de $D(i, j)$ dépend essentiellement de $D(i-1, j)$, $D(i, j-1)$ et $D(i-1, j-1)$. Il faut donc calculer ces trois résultats avant de calculer $D(i, j)$ (qui y aura ainsi directement accès). Une possibilité consiste à calculer ligne par ligne, par colonne croissante, dans une matrice $M : m \times n$ (avec $m = |a|$ et $n = |b|$) telle qu'à la fin du calcul $D_{i,j}$ contient $D(i, j)$. Voyons directement l'implémentation (on suppose qu'aucune des chaînes n'est vide) :

```
let distance_montante a b =
    let m = string_length a
    and n = string_length b in
    let D = make_matrix (m+1) (n+1) 0 in
    (* cas limites *)
    for i = 0 to m do
        D.(i).(0) <- i;
    done;
    for j = 0 to n do
        D.(0).(j) <- j;
    done;
    (* cas recursifs *)
    for i = 1 to m do
        for j = 1 to n do
            let d1 = 1 + D.(i-1).(j)
            and d2 = 1 + D.(i).(j-1)
            and d3 = D.(i-1).(j-1)
                + (if a.[i-1] = b.[j-1] then 0 else 1)
            in
            D.(i).(j) <- min d1 (min d2 d3);
```

3. les anglo-saxons, dans la veine poétique qui les caractérise, qualifie cette approche de *bottom-up* (partir du bas et remonter) par opposition à *top-down*.

4. certains professeurs de mathématiques diront « on intuite ». En bref, pas de recette miracle.

```
        done
    done;
    D.(m).(n) ;;
```

**Exercice :** Adapter cette idée à l'algorithme de plus longue sous-séquence commune.
**Exercice :** Modifier cet algorithme pour qu'à chaque étape il n'utilise que deux tableaux : un pour la ligne précédente (initialement la ligne $i = 0$) et un pour la ligne en cours de calcul. On peut ainsi diminuer l'espace mémoire requis de $m \cdot n$ à $2n$.
**Exercice :** (plus avancé) pour de la correction orthographique, on ne s'intéresse au calcul de la distance que si elle est assez petite (par exemple inférieure ou égale à 2). Comment optimiser la fonction distance_montante pour ne pas évaluer certaines cases dont on est sûr(e)s qu'elles impliquent un coût trop grand ?
**Exercice :** Le problème de *weighted interval scheduling* se résout bien de manière montante. On suppose donc une collection de tâches $t_1, \ldots, t_n$ avec pour chaque tâche $t_i$ un début $d_i$, une fin $f_i$ ($d_i < f_i$), et une valeur (« weight ») $v_i \in \mathbb{N}^*$. Deux tâches sont *compatibles* si $[d_i, f_i]$ et $[d_j, f_j]$ ont une intersection vide. On cherche donc un ensemble de tâches compatibles de poids maximal.
Pour cela on suppose (quitte à les trier au préalable) que les tâches sont indexées par temps de fin $f_i$ croissant, et on construit un tableau $p$ tel que $p[j] = \max\{i | f_i \leq d_j\}$ ($p[j]$ est le plus haut $i < j$ tel que $t_i$ est compatible avec $t_j$). Le problème récursif est $c(j)$ qui calcule le poids maximal atteignable avec les tâches $t_1, \ldots, t_j$ (et la solution est $c(n)$). À chaque $j$ on peut choisir ou non d'inclure la tâche $j$ dans l'ensemble, d'où :

$$c(j) = \begin{cases} 0 & \text{si } j = 0 \\ \min(c_{j-1}, v_j + c(p[j])) & \text{sinon} \end{cases}$$

Soit le type OCaml suivant :

```
type tache = {
    poids : int;
    debut : int;
    fin : int;
} ;;
```

Implémenter une fonction tache vect -> int qui calcule le poids maximal pour un tableau de taches triées par .fin croissants. Si besoin, on peut convertir une liste de tâches en tableau trié :

```
let trier l =
    let l = sort__sort (fun t1 t2 -> t1.fin <= t2.fin) l in
    vect_of_list l ;;
```