# TD Info 4 : Diviser pour régner & Programmation Dynamique

Michael Monerau

7 décembre 2010

## 1 Diviser pour régner

### 1.1 Principe

Le méthode dite "*diviser pour régner*" est bien connue. Elle s'applique aux problèmes qui peuvent se découper en plusieurs sous-problèmes identiques mais plus petits. Le plus souvent, on coupe même le problème en deux parties de même taille. La complexité de l'algorithme $C(n)$ en fonction de $n$ la taille de l'entrée suit donc une récurrence de la forme :

$$C(n) = 2C(n/2) + O(n)$$

où le $O(n)$ est le temps requis pour refusionner les résultats des deux sous-parties. Selon les algorithmes, cette quantité pourrait être plus grande. Ici nous restons dans ce cas-là.

**Question 1.1. Complexité**
Montrer qu'on a alors $C(n) = O(n \ln n)$.

### 1.2 Tri fusion

Une application directe du principe *diviser pour régner* est le tri fusion. C'est un tri optimal (en $O(n \ln n)$) par la formule ci-dessus.

Le tri fusion c'est trois étapes (si le tableau est de longueur 1, c'est fini) :
- Diviser le tableau en 2 parties de même taille (à 1 près...)
- Trier récursivement par fusion les sous-parties
- Fusionner les 2 parties triées en respectant l'ordre

**Question 1.2. Tri fusion**
Parce que ça ne fait jamais de mal, implémenter rapidement le tri fusion sur les listes d'entiers.

Ce tri est-il stable ?

### 1.3 Point les plus proches

Le problème est le suivant : on dispose de $n$ points $(x_i, y_i)_{1 \leq i \leq n}$ du plan. On veut trouver parmi cet ensemble une paire de points qui sont à distance minimale l'un de l'autre.

#### 1.3.1 Naïf

**Question 1.3. Je suis ta paire**
Ecrire un algorithme naïf qui résout le problème. Quelle en est sa complexité ?

#### 1.3.2 Plus vite

Si vous avez trouvé $O(n \ln n)$, bravo, mais votre définition de naïf est étrange. Sinon, améliorons l'algorithme en utilisant *diviser pour régner*.

L'observation de base est que si on partitionne l'ensemble de points $S$ en deux sous-ensembles $S_1$ et $S_2$, alors fatalement une des paires qu'on cherche se trouve soit dans $S_1$, soit dans $S_2$, soit avec une extrémité dans $S_1$ et l'autre dans $S_2$.

On commence à voir comment on procède : il faut trouver un moyen de couper $S$ en deux parties de même taille efficacement, et il faut trouver comment fusionner les résultats provenant de $S_1$ et $S_2$ tout en prenant en compte les segments à cheval.

Pour le premier problème, on a une solution assez simple : on trie les points par abscisse croissante avant de faire quoi que ce soit. Du coup, pour couper $S$ en deux parties de même taille il suffit de couper le tableau de points au milieu.

Pour l'étape de fusion des résultats, posons quelques notations. Les deux sous-exécutions sur $S_1$ et $S_2$ ont donné deux distances minimales $\delta_1$ et $\delta_2$ (avec des paires associées). Posons $\delta = \min(\delta_1, \delta_2)$. D'autre part, on note $x_{med}$ une abscisse qui sépare $S_1$ et $S_2$ (par exemple l'abscisse maximale des points de $S_1$).

S'il existe une paire de points $(p_1, p_2)$ dans $S$ qui est plus proche que $\delta$, alors nécessairement $p_1$ et $p_2$ sont dans le tube de largeur $\delta$ autour de $x_{med}$ (*Contrainte 1*). Et même pire, leurs ordonnées doivent être à moins de $\delta$ l'une de l'autre.

De plus, de part et d'autre de $x_l$, les points sont contraints à être au moins à $\delta$ les uns des autres par hypothèse de récurrence (*Contrainte 2*).

Cela restreint donc la recherche pour les segments à cheval entre $S_1$ et $S_2$.

Comme les points sont triés selon leur abscisse, il est facile de sélectionner la partie du tableau qui nous intéresse, entre $x_l - \delta$ et $x_l + \delta$.

Maintenant supposons que ces points sélectionnés ont été triés selon leur ordonnée. Alors par la *Contrainte 2*, on sait qu'un point donné appartient à une paire de longueur strictement inférieure à $\delta$ si et seulement s'il forme cette paire avec un des septs points qui le suivent dans ce tableau (trié selon $y$, pour rappel).

**Question 1.4. Les septs points**
Montrer pourquoi ce dernier point est vrai.

Pour résumer, on vient de voir comment faire l'opération de combination des deux sous-résultats. Et cette opération est linéaire grâce à l'astuce des sept points. À ceci près qu'on trie des fois, donc on fait intervenir du $O(n \ln n)$ à chaque combination...

Cependant, il n'est pas difficile de voir qu'on peut éliminer cette contrainte : on trie au départ une bonne fois pour toute selon $x$ (on garde l'ordre dans un tableau auxiliaire sans modifier le tableau de points de départ), et on trie selon $y$ une bonne fois pour toutes, idem. Ainsi, on rajoute du $O(n \ln n)$ globalement mais ce n'est pas un problème puisque c'est la complexité de l'algorithme total.

#### 1.3.3 Implémentation

Comme on s'en rend compte à la lecture de l'algorithme, l'implémentation n'est pas évidente. Mais ce qui nous intéresse est surtout d'implémenter un *diviser pour régner*.

Pour ne pas y passer 2h, on va donc faire une étape de combination sans l'astuce des sept points (on fait tous les tests dans la bande autour de $x_l$). De plus, on suppose que le tableau de points en entrée est trié selon abscisses croissantes pour éviter d'avoir à le faire. Les points seront pris comme des éléments d'un type `record` que vous définirez par vous-même (deux champs, $x$ et $y$ de type `int`).

**Question 1.5. Closest pair**
Implémenter cette version simplifiée de l'algorithme, et la tester sur un exemple.

# 2 Programmation dynamique

## 2.1 Principe

La programmation dynamique est une façon d'écrire un algorithme qui ressemble par beaucoup d'aspects à *diviser pour régner*. Le principe de base est toujours de diviser le problème à résoudre en plusieurs sous-problèmes. Mais cette fois-ci, leurs résolutions ne sont pas indépendantes, comme c'était le cas avant. C'est-à-dire que les sous-problèmes partagent des sous-sous-problèmes.

Dans ce cadre, *diviser pour régner* fait plus de travail que nécessaire en résolvant plusieurs fois les mêmes sous-sous-problèmes (penser à l'amélioration qu'apporte `option remember` en Maple).

## 2.2 Plus longue sous-suite commune

Commençons par quelques définitions.

**Définition 1.** *Soit $x = \langle x_1, \ldots, x_k \rangle$ une suite finie. On dit que $z = \langle z_1, \ldots, z_l \rangle$ est une sous-suite de $x$ s'il existe des indices strictements croissants $i_1 < \ldots < i_l$ tels que $z_j = x_{i_j}$.*

**Définition 2.** *On dit que $z$ est une sous-suite commune à deux suites $x$ et $y$ finies si $z$ est une sous-suite à la fois de $x$ et $y$.*

Par exemple, si $x = \langle A, C, E \rangle$ et $y = \langle C, D, E \rangle$, alors $z = \langle C, E \rangle$ est une sous-suite commune à $x$ et $y$.

Le but du problème qu'on se pose est de trouver, étant données deux suites finies $x$ et $y$ la taille d'une plus longue sous-suite commune à $x$ et à $y$.

Posons les notations, $X = \langle x_1, \ldots, x_m \rangle$ et $Y = \langle y_1, \ldots, y_n \rangle$. On suppose également qu'on a une plus longue sous-suite commune $Z = \langle z_1, \ldots, z_k \rangle$. Et on note $U_{[i]}$ pour les $i$ premiers termes de la suite $U$.

On abrège Plus Longue Sous-Suite Commune en PLSSC.

On peut alors établir le :

**Lemme 1.** *Avec les notation ci-dessus :*
- *Si $x_m = y_n$, alors $z_k = x_m = y_n$ et $Z_{[k-1]}$ est une PLSSC de $X_{[m-1]}$ et $Y_{[n-1]}$.*
- *Si $x_m \neq y_n$, alors $z_k \neq x_m$ implique que $Z$ est une PLSSC de $X_{[m-1]}$ et $Y$.*

– *Si $x_m \neq y_n$, alors $z_k \neq y_n$ implique que $Z$ est une PLSSC de $X$ et $Y_{[n-1]}$.*

**Question 2.1**. **Expression récursive**

On note $c_{i,j}$ la taille de la plus longue sous-suite commune à $X_{[i]}$ et $Y_{[j]}$. À partir du lemme précédent, trouver une expression de $c_{i,j}$ en fonction de $c_{i-1,j-1}$ et $c_{\ldots,\ldots}$.

**Question 2.2**. **Implémentation**

Implémenter ce calcul et donner la taille de la plus longue sous-suite commune de deux suites finies que vous choisirez.

## 2.3 Plus court chemin : algorithme de Floyd

On suppose qu'on a un graphe $G = (V, E)$ pondéré (*ie.* on a une fonction de poids des arêtes $w : E \longrightarrow \mathbb{R}^+$).

Le problème est de trouver la longueur du plus court chemin entre chaque paire de sommets du graphe, la longueur d'un chemin étant la somme du poids de ses arêtes.

Plutôt que de calculer le plus court chemin indépendamment, on calcule tout en même temps. C'est adapté à la programmation dynamique car un plus court chemin est héréditaire : une partie d'un plus court chemin est un plus court chemin (entre d'autres sommets). Un calcul de plus court chemin entre plusieurs paires peut donc faire plusieurs fois appel à un plus court chemin particulier, qu'on n'a pas à recalculer si on l'a retenu. C'est l'algorithme de Floyd, il est du même type que l'algorithme de Warhsall du TD2 qui trouvait la clôture transitive d'un graphe.

Ici, on prend la représentation des graphes en matrice d'adjacence : $\forall i, j \in V,\ M(i, j) = w(i, j)$ (et qui est nul si $(i, j) \notin E$).

On définit $dist_k(i, j)$ comme la longueur du plus court chemin de $i$ à $j$ ne contenant que des sommets d'étiquette $\leq k$ (sauf peut-être $i$ ou $j$), pour $k \geq 1$.

**Question 2.3**. $dist_1$ **?**

Que vaut $dist_1$ ?

**Question 2.4**. $dist_2$ **?**

Que vaut $dist_2$ ?

**Question 2.5**. $dist_k$ **?**

Définir $dist_k$ en fonction de $dist_{k-1}$ pour $k \geq 2$.

**Question 2.6**. $dist$ **?**

En quoi la suite de matrices $(dist_k)_{k\geq 0}$ nous donne-t-elle accès à la longueur du plus court chemin entre $(i, j)$ que nous recherchons ?

**Question 2.7**. **Implémentation**

Implémenter l'algorithme de Floyd. Quelle est la complexité ?