Notes de cours d'algorithmique

Gilles Zémor

Février 2021

1 Algorithmes gloutons

1.1 Principe général d'une démonstration qu'un algorithme glouton est optimal

On prend une solution gloutonne $g_1, g_2, \ldots, g_n$, dans l'ordre du déroulement de l'algorithme, que l'on compare à une solution optimale. On suppose que $g_1, g_2, \ldots, g_m$ font partie de la solution optimale O et l'on montre qu'il existe alors une solution optimale O' qui contient $g_1, g_2, \ldots, g_m, g_{m+1}$.

Exemple 1. On veut exprimer un entier positif S comme somme du plus petit nombre possible de «pièces» appartenant à l'ensemble $\{20, 10, 5, 2, 1\}$. La solution gloutonne consiste à choisir la pièce P la plus grande inférieure ou égale à S puis à recommencer avec $S \leftarrow S - P$.

Considérons le premier moment où l'algorithme glouton diverge avec une solution optimale. À ce moment il s'agit de reconstituer S. Supposons $S > 20$. La solution gloutonne nous dit de choisir une pièce de 20. Puisque nous divergeons de la solution optimale, c'est qu'il n'y a pas de pièce de 20 dans la solution optimale O. On remarque qu'il ne peut y avoir au plus qu'une pièce de 10 dans la solution optimale, car s'il y en a deux, on aurait une solution meilleure en remplaçant deux pièces de 10 par une pièce de 20. De même il y a au plus une pièce de 5. Il peut y avoir deux pièces de 2, mais pas 3, car $3 \times 2 = 5 + 1$. Enfin, il y a au plus une pièce de 1. Mais l'ensemble de toutes les pièces autorisées ne somme qu'à $20 < S$. Donc, au moment où la solution gloutonne divergerait de la solution optimale on ne peut qu'avoir $S \leqslant 20$. Si $S = 20$ la solution est triviale et coïncide avec la gloutonne. Enfin, pour $S < 10$ on peut raisonner de manière similaire à ci-dessus et conclure que la solution gloutonne est toujours l'*unique* solution optimale.

Exemple 2 : recherche d'un stable maximum dans un graphe d'intervalles. On nous donne un ensemble $\mathcal{I}$ d'intervalles de la droite rélle, et on cherche un ensemble maximum d'intervalles de $\mathcal{I}$ deux à deux disjoints. La solution gloutonne consiste à choisir, parmi les intervalles disjoints des intervalles déjà choisis, l'intervalle $[a, b]$ minimisant la valeur de b. Soit $I_1, I_2, \ldots, I_m, I_{m+1}, \ldots$ la solution gloutonne, et supposons que $I_1, I_2, \ldots, I_m, J_{m+1}, J_{m+2}, \ldots$ est une solution optimale, où $I_{m+1} = [a_{m+1}, b_{m+1}]$ ne fait pas partie de cette solution. L'intervalle doit intersecter au moins un intervalle de la solution optimale O, sinon on pourrait le rajouter à O et améliorer la solution. Comme I_{m+1} n'intersecte pas $I_1, \ldots I_m$, il doit intersecter un certain $J_i, i > m$. Or I_{m+1} est choisi de telle sorte que son extrêmité droite b_{m+1} est inférieure à l'extrêmité droite de tous les $J_j, j > m$. Il ne peut donc pas y avoir deux intervalles distincts J_i et J_j qui intersectent I_{m+1} sinon ces deux intervalles s'intersecteraient en b_{m+1}. On peut donc fabriquer une nouvelle solution optimale, en enlevant de O l'unique intervalle J_i qui intersecte I_{m+1}, et en le remplaçant par l'intervalle I_{m+1}. C'est une procédure d'échange. De proche en proche on en déduit qu'il existe une solution optimale qui contient la solution gloutonne et donc qui lui est égale.

Exemple 3 : recherche d'un arbre couvrant de poids minimum. Algorithme de Kruskal. On nous donne un graphe pondéré (chaque arête a un poids). On souhaite mettre en évidence un arbre couvrant de poids minimal. L'algorithme de Kruskal est un algorithme glouton qui, à chaque étape choisit l'arête (ou une arête) de poids minimum parmi les arêtes qui ne créent pas de cycle avec le sous-ensemble d'arêtes déjà choisies. Montrons que cette procédure produit toujours un arbre couvrant de poids minimal. Soient la suite d'arêtes $g_1, g_2, \ldots, g_{n-1}$ choisies de manière gloutonne, et soit $g_1, g_2, \ldots, g_m, e_{m+1}, \ldots, e_{n-1}$ un arbre couvrant M de poids minimal qui ne comprend pas l'arête g_{m+1} mais contient les arêtes $g_1, \ldots g_m$. Ajoutons l'arête g_{m+1} à l'arbre M. Cela créé un cycle C. Remarquons que ce cycle doit contenir une arête $e_i, i > m$, car $g_1, g_2, \ldots, g_m, g_{m+1}$ ne contient pas de cycle. Si l'on enlève l'arête e_i de ce cycle, on recréé un arbre de nouveau. En d'autres termes on créé un nouvel arbre à partir de M en *échangeant* les arêtes e_i et g_{m+1}. Forcément le poids de l'arête g_{m+1} est inférieur ou égal au poids de l'arête e_i par définition de la procédure gloutonne (sinon l'algorithme glouton aurait préféré l'arête e_i à l'arête g_{m+1}). On a donc fabriqué un nouvel arbre couvrant de poids minimum qui coïncide plus avec l'arbre glouton, et de proche en proche on obtient que l'arbre glouton est un arbre couvrant de poids minimal.

1.2 Matroïdes

On appelle *matroïde* tout ensemble fini X muni d'un ensemble de parties $\mathcal{I}$ vérifiant les propriétés suivantes :

- (hérédité) si $A \in \mathcal{I}$ et $B \subset A$, alors $B \in \mathcal{I}$.
- (échange) si $A, B \in I$ et $|B| > |A|$, alors il existe $b \in B \subset A$ tel que $A \cup \{b\} \in \mathcal{I}$.

Les éléments de $\mathcal{I}$ sont appelés les *indépendants* du matroïde. On appelle *base* d'un matroïde tout indépendant maximal (qui n'est pas inclus dans un indépendant de cardinalité strictement supérieure). La propriété d'échange implique clairement que toutes les bases ont le même nombre d'éléments, c'est la *dimension* du matroïde. Enfin on appelle *circuit* du matroïde, toute partie C de X non-indépendante mimale, i.e. telle que $C \setminus \{c\} \in \mathcal{I}$ pour tout $c \in C$.

La notion de matroïde est une généralisation abstraite à la fois d'un graphe et d'un espace vectoriel.

Exemples de matroïdes.

Matroïdes de cardinalité. X est un ensemble fini à n éléments et $\mathcal{I}$ est l'ensemble des parties de X contenant au plus k éléments, pour $k < n$ fixé.

Matroïdes graphiques. X est l'ensemble des arêtes d'un graphe fini G, et $\mathcal{I}$ est constitué des ensembles d'arêtes qui ne contiennent pas de cycle.

Matroïdes vectoriels (ou linéaires). X est un ensemble fini de vecteurs d'un espace vectoriel V, et $\mathcal{I}$ est constitué des ensembles de vecteurs de X linéairement indépendants. Autrement dit, X est représentable par l'ensemble des colonnes d'une certaine matrice, et les indépendants sont les ensembles de colonnes linéairement indépendantes. Ce cas particulier est derrière la terminologie de «matroïde» et de partie «indépendante». La notion de circuit est quand à elle empruntée au contexte des graphes.

On peut montrer (exercice !) qu'un matroïde graphique est aussi un matroïde vectoriel. Il en est de même des matroïdes de cardinalité. Tous les matroïdes ne sont pas représentables comme des matroïdes vectoriels.

Lemme 1. *Soit $A \in \mathcal{I}$ un indépendant. Soit $x \in X$ tel que $\{x\} \in \mathcal{I}$ et tel que $A \cup \{x\}$ ne soit plus un indépendant. Soit $C \subset A \cup \{x\}$ un circuit et soit $y \neq x$ un élément du circuit. Alors $A \setminus \{y\} \cup \{x\}$ est un indépendant.*

Preuve : Remarquons que le circuit C doit exister : si $A \cup \{x\}$ n'est pas un circuit, c'est qu'il existe un $a \in A$ tel que $A \setminus \{a\} \cup \{x\}$ n'est pas un indépendant. On supprime alors a de $A \cup \{x\}$, pour obtenir un nouvel ensemble non indépendant et

on recommence jusqu'à obtenir un circuit C. Ce circuit contient d'autres éléments que x puisqu'on a supposé $\{x\}$ indépendant. Soit donc $y \in C \setminus \{x\}$. Notons que $|C \setminus \{y\} \cup \{x\}| = |C|$. Si $|A| > |C|$, alors la propriété d'échange nous dit qu'on peut ajouter un élément a de A à $C \setminus \{y\} \cup \{x\}$ et obtenir encore un indépendant. L'élément a qu'on ajoute ne peut pas être égal à y car un indépendant ne peut pas contenir $C \notin \mathcal{I}$. Le même argument d'échange nous permet de continuer cette procédure jusqu'à obtenir que $A \setminus \{y\} \cup \{x\}$ doit être un indépendant. ∎

Algorithme glouton : recherche d'une base de poids minimal dans un matroïde. Il nous est donné un matroïde $(X, \mathcal{I})$ *pondéré*, c'est-à-dire que chaque élément $x \in X$ est muni d'un poids $p(x)$. On cherche une base B de poids minimale, i.e. minimisant $\sum_{b \in B} p(b)$. L'algorithme glouton consiste à construire itérativement la base B, en partant de $B = \emptyset$ et en l'augmentant d'un élément à chaque étape. Chaque étape consiste à rajouter à B un élément b de poids minimal parmi ceux tels que $B \cup \{b\}$ est un indépendant. On remarque que dans le cas où le matroïde est un matroïde graphique, le problème est celui de la recherche d'un arbre couvrant de poids minimal et l'algorithme est exactement l'algorithme de Kruskal.

Preuve d'optimalité de l'algorithme glouton. Soit $g_1, g_2, \ldots, g_m, g_{m+1}, \ldots, g_k$ une base G produite par l'algorithme glouton et soit $g_1, \ldots, g_m, b_{m+1}, \ldots, b_k$ une base B de poids minimal telle que g_m ne figure pas parmi les éléments de B. Considérons $B \cup \{g_{m+1}\}$: comme cet ensemble ne peut pas être indépendant, il doit contenir un circuit C (comme nous l'avons déjà remarqué, il suffit de supprimer des éléments de $B \cup \{b_{m+1}\}$ jusqu'à obtenir un non-indépendant minimal). Ce circuit doit contenir g_{m+1} (sinon on aurait $C \subset B$ ce qui n'est pas possible puisque B est indépendant), et aussi au moins un élément b_i de B qui n'est pas dans $\{g_1, \ldots, g_m\}$ (sinon on aurait $C \subset G$). Le lemme 1 nous dit que $B' = B \setminus \{b\} \cup \{g_{m+1}\}$ est encore une base. Comme b_i est différent de $g_1, \ldots, g_m$, on a forcément que $p(g_{m+1}) \leqslant p(b_i)$, sinon l'algorithme glouton aurait préféré b_i à g_{m+1}, et B' doit être une base de poids minimal. De proche en proche on obtient que la base gloutonne est une base de poids minimal. ∎

Matroïde transversal. Soit (X, Y, E) un graphe biparti, dont chaque arête $e \in E$ relie un sommet de X à un sommet de Y. Un *couplage* du graphe est un sous-graphe de degré 1, c'est un dire un ensemble $C \subset E$ d'arêtes reliant $A \subset X$ à $A' \subset Y$ de telle sorte que chaque $a \in A$ et chaque $a' \in A'$ est incident à une unique arête de C.

On appelle *transversal partiel* de X une partie $A \subset X$ pouvant être couplée à une partie $A' \subset Y$. Soit $\mathcal{I}$ l'ensemble des transversaux partiels de X.

Proposition 2. *L'ensemble* $\mathcal{I}$ *munit* X *d'une structure de matroïde, appelé* matroïde transversal.

Preuve : La propriété d'hérédité est trivialement vérifiée, il s'agit donc de prouver la propriété d'échange. Soit $A, B \subset X$ des transversaux partiels, avec $|B| > |A|$. Il s'agit de montrer qu'il existe $b \in B \setminus A$ tel que $A \cup \{b\}$ puisse être couplé avec une partie de Y.

Soit C_A et C_B les couplages issus de A et B respectivement. Considérons C_A comme un ensemble d'arêtes rouges et C_B comme un ensembles d'arêtes bleues. Réunissons les arêtes bleues et rouges pour former un graphe G_{AB}. Notons que nous distinguons toutes les arêtes bleues et rouges, c'est-à-dire que si une arête bleue et une arête rouge sont issues de la même arête d'origine, elles deviennent deux arêtes distinctes qui forment un cycle de longueur 2 dans G_{AB}. Nous remarquons que le graphe G_{AB} est un graphe de degré maximum 2, et est donc constitué d'une réunion disjointe de cycles et de chemins. Dans chaque cycle et chaque chemin, les arêtes bleues et rouges alternent. Comme $|B| > |A|$, il y a strictement plus d'arêtes bleues que d'arêtes rouges, et comme un cycle contient autant d'arêtes bleues que d'arêtes rouges, il existe un chemin contenant strictement plus d'arêtes bleues que d'arêtes rouges. Comme les arêtes bleues et rouges alternent sur ce chemin, il comporte exactement une arête bleue de plus et ses deux extrémités sont incidentes à des arêtes bleues. On constate que si dans le couplage C_A on remplace les arêtes rouges du chemin par les arêtes bleues on obtient un nouveau couplage constitué d'une arête supplémentaire et qui couple A augmenté d'un élément de B à une partie de Y. La situation est illustrée figure 1. ∎

1.3 Algorithme de Huffman

L'algorithme de Huffman est un algorithme de *compression*. Le contexte est le suivant. On suppose donnée une *source*, qui produit une suite $X_1, X_2, \ldots, X_N$ de symboles aléatoires chacun choisi parmi les symboles d'un même alphabet fini $\mathcal{X} = \{x_1, \ldots, x_m\}$ et avec une même loi de probabilité $p = (p_1, \ldots, p_m)$, $p_i = P(X_j = x_i)$ pour tout j.

Un *encodage* de la source consiste à associer à chaque symbole $x_i \in \mathcal{X}$ un mot d'un *code* C constitué d'un ensemble de m mots. Une suite de symboles de la source est ensuite transformée en la chaîne de bits obtenue par concaténation des encodages des symboles successifs.

Exemple. Soit $\mathcal{X} = \{1, 2, 3, 4\}$ et soit X à valeurs dans $\mathcal{X}$ de loi $p_1 = 1/2, p_2 =$

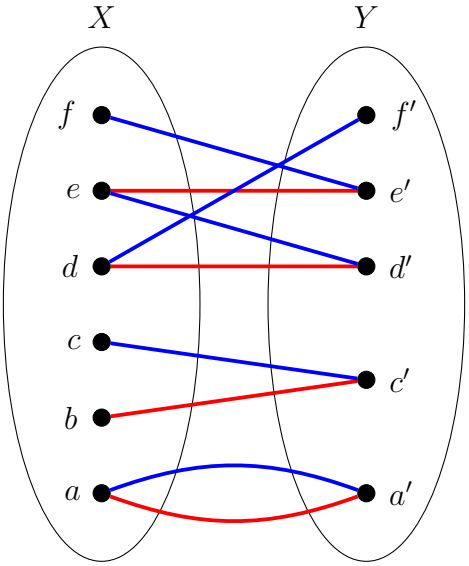

FIGURE 1 – Exemple d'augmentation de couplage : $A = \{a, b, d, e\}$ et $B = \{a, c, d, e, f\}$. Le chemin alterné f, e', e, d', d, f' est celui qui permet d'augmenter le transversal partiel A en $A \cup \{f\}$ par échange des arêtes rouges et bleues.

$1/4, p_3 = 1/8, p_4 = 1/8$ où $p_i = P(X = i)$. Soit le codage c défini par

$$\begin{aligned} c(1) &= 0 \\ c(2) &= 10 \\ c(3) &= 110 \\ c(4) &= 111. \end{aligned}$$

Le mot 011000101110 est l'encodage de la suite de symboles 1311241. Le code C est un code dit *préfixe*, ce qui veut dire qu'aucun mot de C n'est le préfixe d'un autre. Ceci permet de reconstituer sans ambiguïté la suite de symboles d'origine à partir de son encodage par l'algorithme glouton consistant à lire la suite de bits depuis la gauche et à lui associer un mot de code dès qu'on peut. Sur l'exemple, comme le premier symbole 0 est un mot du code, on le prend, ensuite ni 1 ni 11 ne sont dans le code, on continue donc à lire et on convertit 110 en «3» car $110 \in C$, et ainsi de suite.

Un code préfixe peut être utilement représenté par un arbre binaire dont les mots du code sont les feuilles. Le code de l'exemple ci-dessus est ainsi représenté par l'arbre de la figure 2.

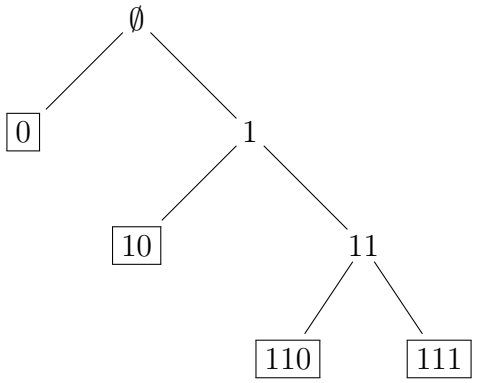

FIGURE 2 – L'arbre associé au code préfixe $\{0, 10, 110, 111\}$

Longueur moyenne d'un code. Appelons ℓ_i la longueur du mot binaire qui encode le symbole x_i. La longueur moyenne associée au code préfixe et à la loi de probabilité p est

$$\overline{\ell} = p_1\ell_1 + \cdots + p_m\ell_m$$

et représente le nombre de bits moyen représentant chaque symbole encodé.

Un code préfixe (ou un arbre) est dit *optimal* pour une loi p s'il minimise la longueur moyenne $\overline{\ell}$. L'algorithme de Huffman permet de construire un arbre optimal.

Algorithme de Huffman. A chaque étape on dispose d'un ensemble de sous-arbres disjoints. La valeur de chaque sous-arbre est la somme des valeurs de ses feuilles (une somme de p_i). Au début les sous-arbres sont juste m sommets uniques associées aux valeurs $p_1, \ldots, p_m$. L'algorithme de Huffman est un algorithme glouton qui à chaque étape réunit par un sommet père commun deux sous-arbres de valeurs minimales. Au bout de $m-1$ étapes on a donc fabriqué un arbre.

Exemple. Soit l'ensemble $\mathcal{X} = \{x_1, x_2, \ldots, x_6\}$ et la loi $p_1 = 0.4, p_2 = 0.04, p_3 = 0.14, p_4 = 0.18, p_5 = 0.18, p_6 = 0.06$. L'arbre obtenu par l'algorithme de Huffman est représenté sur la figure 3. La première étape consiste à joindre les sommets terminaux (feuilles) x_2 et x_6 associés aux probabilités p_2 et p_6 les plus faibles et à créer ainsi un sommet intermédiaire i de l'arbre associé à la probabilité $p_i = p_2 + p_6 = 0.1$. Puis on recommence la procédure sur l'ensemble $\mathcal{X}' = \{x_1, x_3, x_4, x_5, i\}$ pour la loi $p_1 = 0.4, p_3 = 0.14, p_4 = 0.18, p_5 = 0.18, p_i = 0.1$. Les probabilités les plus faibles sont p_3 et p_i, on joint donc x_3 et i en un sommet père ii de probabilité $p_{ii} = 0.24$. La procédure se termine par l'arbre de la figure 3.

Pour démontrer l'optimalité de l'algorithme de Huffman nous utiliserons le lemme suivant.

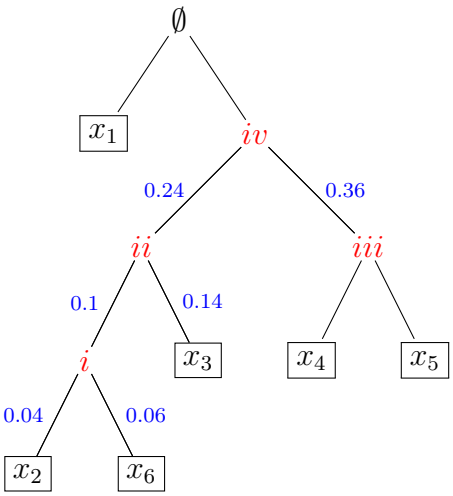

FIGURE 3 – Arbre obtenu par application de l'algorithme de Huffman

Lemme 3. *Soit l'ensemble* $\mathcal{X} = \{x_1, \ldots, x_m\}$ *et la loi* $p = (p_1, \ldots, p_m)$ *où l'on a ordonné les* x_i *de telle sorte que la suite des* p_i *décroisse. Soit* A *un arbre optimal pour la loi* p*.*

- *Chaque sommet de* A *qui n'est pas une feuille admet exactement deux sommets successeurs. En particulier il existe deux sommets de profondeur maximale* h *ayant un même sommet père.*
- *Si deux feuilles de profondeur maximale et de même sommet père ne sont pas étiquetées par* x_m *et* x_{m-1}*, associées aux probabilités les plus faibles* p_m *et* p_{m-1}*, alors un échange d'étiquettes des feuilles permet d'avoir un autre arbre optimal dont les deux feuilles considérées sont cette fois étiquetées par* x_m *et* x_{m-1}*.*

Preuve : Si un sommet intermédiaire n'a qu'un successeur, alors on peut le supprimer et le remplacer par son successeur et faire diminuer strictement la longueur moyenne. Ceci montre le premier point. Pour le deuxième point, si les étiquettes des sommets considérés sont x_i et x_j, $i, j \neq m$, alors on peut clairement échanger x_i et x_m sans augmenter la longueur moyenne de l'encodage. De même, si les étiquettes des deux sommets sont x_m et x_i, $i \neq m-1$, on peut échanger x_i avec x_{m-1}. ∎

Optimalité de l'algorithme de Huffman. Soit H un arbre obtenu par l'algorithme de Huffman et soit A un arbre optimal. Nous considérons A et H comme des arbres étiquetés, où chacune des m feuilles est étiquetées par un p_i et chaque autre sommet est étiqueté par la somme des étiquettes de ses successeurs. Nous uti-

lisons la stratégie générale pour démontrer qu'un algorithme glouton est optimal, c'est-à-dire que nous considérons la dernière étape e après laquelle l'algorithme de Huffman produisant H est compatible avec l'arbre A. Ceci veut dire qu'à l'étape e tous les sous-arbres disjoints de l'algorithme de Huffman sont aussi des sous-arbres de A. Notons qu'au tout début de l'algorithme de Huffman (après l'étape 0), lorsque les sous-arbres sont des sommets uniques, ce sont tous des feuilles, donc des sous-arbres de A. Soient $s_1, \ldots, s_{m-e}$ les sommets racines des $m-e$ sous-arbres en question, communs à A et à H. Soit A' le sous-arbre de A issu de sa racine et ayant pour feuilles les sommets $s_1, \ldots s_{m-e}$. Soit p' la loi $p'_1, \ldots, p'_{m-e}$ où p'_j est l'étiquette de s'_j. Rappelons que la quantité $\ell_i = \ell_i(A)$ pour l'arbre A désigne la profondeur de la feuille étiquetée par le symbole x_i. Nous constatons que la longueur moyenne $\overline{\ell}$ pour l'arbre A et la loi p est égale à la longueur moyenne pour l'arbre A' et la loi p' à laquelle il faut ajouter une quantité qui ne dépend que des sous-arbres issus de $s_1, \ldots s_{m-e}$. L'arbre A' est donc optimal pour la loi p'. Maintenant le lemme 3 nous permet d'affirmer qu'il existe une manière d'échanger les étiquettes $s_1, \ldots, s_{m-e}$ associées aux feuilles de A' sans changer l'optimalité de A' et donc d'obtenir un nouvel arbre optimal pour la loi p qui coïncide avec l'arbre de Huffman H jusqu'à l'étape $e+1$. De proche en proche on obtient que l'arbre de Huffman H est optimal.

2 Diviser pour reigner

Les algorithmes de type «diviser pour reigner» (divide and conquer) constitue une famille d'algorithmes dont le principe général est de diviser le problème à traiter, en sous-instances du problème d'origine de taille plus petite, qu'il s'agit ensuite de recombiner. Quand l'algorithme s'appelle ainsi lui même sur une instance plus petite, on dit qu'il est *récursif.*

Exemples.

Recherche dans une liste triée. Il nous est donnée une liste $\{a_1 \leqslant a_2 \leqslant \cdots a_n\}$ de n entiers, triée dans l'ordre croissant, ainsi qu'un entier x dont on veut savoir s'il est dans la liste. La recherche par dichotomie consiste à comparer x avec $a_{n/2}$ (en supposant n pair), puis à répéter l'opération avec la première liste $a_1 \ldots a_{n/2-1}$ si $x < a_{n/2}$ ou avec la deuxième liste $a_{n/2+1}, \ldots, a_n$ si $x > a_{n/2}$ (si $x = a_{n/2}$ on a fini). Si on note $T(n)$ le temps de calcul nécessaire à l'execution de la tâche globale, mesuré en le nombre nécessaire de comparaisons de x avec un entier de la liste, il vient de la récursion l'inégalité :

$$T(n) \leqslant T(n/2) + 1. \tag{1}$$

En réappliquant l'inégalité à la sous-liste de taille $n/2$ qui survit, on obtient $T(n) \leqslant T(n/4) + 2$, et de proche en proche $T(n) \leqslant T(1) + n \log_2 n = n \log_2 n$.

Tri. Il nous est donné cette fois-ci une liste $a_1, a_2, \ldots, a_n$ d'entiers dans un ordre quelconque, et il s'agit de les trier. La méthode du *tri fusion* consiste à séparer la liste en 2 parties d'égales longueurs (ou différent de 1 si n est impair), de trier chacune des deux sous-listes, puis de les *fusionner*. Pour fusionner deux listes triées de longueurs ℓ et ℓ', on compare les deux plus petits éléments de chaque liste, et on met le plus petit dans une troisième liste, puis on recommence avec les deux listes initiales dont la somme des longueurs est devenue $\ell + \ell'$, jusqu'à ce que la troisième liste contienne tous les $\ell + \ell'$ éléments de manière ordonnée. Il faut $\ell + \ell' - 1$ comparaisons dans le pire cas pour faire la fusion. On a donc l'inégalité, pour le nombre $T(n)$ total de comparaisons d'entiers nécessaires au tri :

$$T(n) \leqslant 2T(n/2) + n \tag{2}$$

Si on réapplique cette inégalité aux sous-listes, il vient $T(n) \leqslant 4T(n/4) + n + 2(n/2) = 4T(n/4) + 2n$, et de proche en proche :

$$T(n) \leqslant nT(n/n) + (\log_2 n)n = n \log_2 n$$

puisque $T(1) = 0$. On a supposé que n est une puissance de 2 pour simplifier, mais le cas général est à peine différent.

Multiplication d'entiers. Soient deux entiers a et b de n bits au plus ($< 2^n$, donc) à multiplier. Supposons pour simplifier que n soit une puissance de 2. On peut écrire

$$a = a_0 + a_1 2^{n/2}$$
$$b = b_0 + b_1 2^{n/2}$$

où a_0, a_1, b_0, b_1 sont tous des entiers d'au plus $n/2$ bits ($< 2^{n/2}$). On a

$$ab = a_0 b_0 + (a_1 b_0 + a_0 b_1) 2^{n/2} + a_1 b_1 2^n.$$

Ceci donne naissance à un algorithme récursif où il s'agit de faire quatre multiplications d'entiers de $n/2$ bits, pour obtenir $a_0 b_0, a_1 b_0, a_0 b_1, a_1 b_1$, puis de faire des additions d'entiers d'au plus $2n$ bits ainsi que des décalages (multiplications par $2^{n/2}$ et 2^n). Considérons que le coût d'une addition soit de l'ordre de grandeur de $2n$ opérations élémentaires (addition de bits avec retenue), et écrivons donc l'inégalité de récursion, sur le temps de calcul $T(n)$ mesuré en opérations élémentaires :

$$T(n) \leqslant 4T(n/2) + Mn \tag{3}$$

où M est une constante. Toujours en réappliquant l'inégalité, il vient $T(n) \leqslant 4^2 T(n/4) + Mn + 4Mn/2$, et en itérant,

$$T(n) \leqslant 4^k T(n/2^k) + M(n + 4n/2 + \cdots + 4^{k-1} n/2^{k-1}),$$

ce qui, pour $k = \log_2 n$ nous donne :

$$\begin{aligned} T(n) &\leqslant 4^{\log_2 n} T(1) + Mn(1 + 2 + 4 + \cdots + 2^{k-1}) \\ &= n^2 + Mn(2^{\log_2 n} - 1) \\ &\leqslant n^2 + Mn^2. \end{aligned}$$

On retrouve donc la complexité de l'algorithme usuel de multiplication.

Multiplication de Karatsuba. On peut utiliser une astuce pour faire apparaître une récursion différente. Remarquons que

$$a_1 b_0 + a_0 b_1 = (a_0 + a_1)(b_0 + b_1) - a_0 b_0 - a_1 b_1.$$

On peut donc obtenir le résultat en faisant appel à *trois* multiplications d'entiers d'au plus $n/2+1$ bits, pour obtenir $a_0 b_0$, $a_1 b_1$, et $(a_0 + a_1)(b_0 + b_1)$. Il faut d'abord avoir fait deux additions pour calculer $(a_0 + a_1)$ et $b_0 + b_1$: globalement, il faut plus d'additions qu'avant, mais on a gagné sur le nombre de multiplications et ceci est très intéressant car ces multiplications coûtent nettement plus cher que les additions. On obtient la nouvelle inégalité récursive :

$$T(n) \leqslant 3T(n/2) + O(n). \tag{4}$$

À strictement parler, on a obtenu $T(n) \leqslant 2T(n/2) + T(n/2 + 1) + O(n)$. Mais on peut écrire les deux entiers x et y de $n/2+1$ bits qu'il faut multiplier sous la forme $x = x_0 + 2x_1$ et $y = y_0 + 2y_1$, et comme les multiplications par un nombre de 1 bit ne coûtent rien, on voit que le coût d'une multiplication de deux nombres de $n/2 + 1$ bits est au plus celui d'une multiplication de deux nombres de $n/2$ bits et de quelques additions, ce qui justifie (4).

Il reste à déterminer la complexité asymptotique de la multiplication si on applique cette méthode récursive. Plutôt que traiter chaque équation du type (1),(2),(3),(4) au cas par cas, on comprend qu'il est utile d'avoir un théorème général.

Théorème 4. *Si $T(n)$ désigne le temps de calcul d'un algorithme récursif vérifiant une inégalité du type $T(n) \leqslant aT(\lceil n/b \rceil) + O(n^c)$ pour des constantes $a, c \geqslant 0$, $b > 1$, alors*

$$T(n) \leqslant \begin{cases} O(n^c) & \text{si } c > \log_b a \\ O(n^c \log n) & \text{si } c = \log_b a \\ O(n^{\log_b a}) & \text{si } c < \log_b a. \end{cases}$$

Preuve : Supposons pour simplifier que n soit une puissance de b, $n = b^k$, le cas général n'étant pas fondamentalement différent. D'après l'hypothèse nous avons

$$T(n) \leqslant aT(n/b) + Mn^c \tag{5}$$

pour tout n et pour une certaine constante M. En réappliquant (5) à n/b nous avons :

$$\begin{aligned} T(n) &\leqslant a\left(aT(\frac{n}{b^2}) + M\left(\frac{n}{b}\right)^c\right) + Mn^c \\ &\leqslant a^2 T\left(\frac{n}{b^2}\right) + Mn^c\left(1 + \frac{a}{b^c}\right) \end{aligned}$$

et de proche en proche

$$T(n) \leqslant a^k T\left(\frac{n}{b^k}\right) + Mn^c\left(1 + \frac{a}{b^c} + \cdots + \left(\frac{a}{b^c}\right)^{k-1}\right).$$

Prenons $k = \log_b n$ et convenons que $T(1) = 0$ (sinon on peut appliquer la récursion une fois de plus et convenir que $T(1/b) = 0$). On obtient :

$$T(n) \leqslant Mn^c\left(1 + \frac{a}{b^c} + \cdots + \left(\frac{a}{b^c}\right)^{\log_b n - 1}\right).$$

Si $c > \log_b a$, c'est-à-dire si $a/b^c < 1$, la série converge et nous avons $T(n) \leqslant O(n^c)$. Si $c = \log_b a$, i.e. $a/b^c = 1$, tous les termes de la série valent 1 et nous avons $T(n) \leqslant Mn^c \log_b n$. Enfin si $c < \log_b a$, i.e. $a/b^c > 1$, nous obtenons $T(n) \leqslant M' n^c (a/b^c)^{\log_b n}$ pour une constante M', soit $T(n) \leqslant M' a^{\log_b n} = M' n^{\log_b a}$. ∎