Université Paris Sud

Algorithmique

et Structures de Données

Karim Belabas

Cours/TD, 20h. Dernière révision : Janvier 2005.

Table des matières

1. Algorithmique

1.1. **Algorithmes.** Algorithmique = conception et analyse d'algorithmes.

Définition 1.1. Étant donné un problème, un *Algorithme* décrit une suite finie d'opérations qui le résout. À partir de données (entrées) concrètes d'un certain type, suivre la méthode produit un résultat (sortie) en un nombre fini d'opérations. L'énoncé du problème spécifie la relation entrée/sortie souhaitée.

Par exemple, l'algorithme d'Euclide résoud le problème du calcul du pgcd de 2 entiers, QuickSort résout le problème du tri.

Pour ces différentes notions, nous nous en tiendrons à une approche naïve, à des descriptions en langage naturel, non formalisé. Il faudrait définir plus rigoureusement les mots suivants :

Machine : modèle abstrait, indépendant de la machine concrète ou du langage de programmation utilisé. En pratique, on reprend les caractéristiques essentielles des ordinateurs connus à ce jour, à savoir la donnée d'une *mémoire* (suite de cases où l'on peut lire/écrire) et d'une liste d'*opérations* permises. Il y a plusieurs modèles (RAM[1], machine de Turing[2]...). RAM et machines de Turing sont équivalents du point de vue de la calculabilité (quels problèmes sont résolubles par un programme exécuté sur telle machine ?), de la complexité grossière (quels problèmes sont résolubles en temps polynomial ?), pas du point de vue des coûts détaillés. Voir [1, Chapitre 1] pour une description concise de ces modèles.

Problème : description adaptée, en particulier finie, des objets utilisés. La représentation des *données* et du *résultat* est incluse dans cette spécification. Un *problème* est une description générale (pgcd de 2 entiers), une *instance* un cas concret particulier, entièrement spécifié ($\mathrm{pgcd}(2,3)$).

Il existe des problèmes pour lequel aucun algorithme ne peut exister, par exemple le problème de l'arrêt, ou la solubilité d'un système d'équations polynomiales sur $\mathbb{Q}$. Dans ce cours, on étudiera uniquement des problèmes pour lesquels il existe des algorithmes, et même des algorithmes efficaces. Plus précisément, on associe une *taille* T à chaque instance x d'un problème, et un *coût* $\mathcal{C}$ à un algorithme appliqué à cette instance. Ce n'est en rien canonique ; à nous de choisir des définitions réalistes, adaptées à la représentation des données et à l'analyse de l'algorithme. Pour différents algorithmes résolvant le problème, notre but est d'évaluer $\mathcal{C}(T)$, ou plus précisément $\mathcal{C}(x)$ pour les x de taille T. L'algorithme

[1] `Random Access Machine`, permet un accès en temps constant à toute case mémoire.

[2] Ruban mémoire sur lequel se déplace une tête de lecture/écriture, les opérations permises sont le déplacement de la tête d'une case, la lecture ou l'écriture d'une valeur. Restreindre l'ensemble des valeurs (par exemple à $\{0,1\}$) ou admettre plusieurs rubans/têtes ne change pas fondamentalement les propriétés du modèle.

le plus adapté est celui qui minimise cette fonction de coût pour les familles d'instances qui nous intéressent.

1.2. **Taille d'une instance d'un problème.** On choisit une taille liée à la place occupée en mémoire par la description du problème ou, dans certains cas, par sa solution. C'est un nombre positif, tel qu'il n'existe qu'un nombre *fini* d'instances de taille $\leqslant T$ fixé, pour tout T. Autant que possible, le choix est adapté au problème, à son analyse (au moins asymptotique), et indépendant des détails de la représentation, du langage, et de l'habileté du programmeur. Exemples :

- pour un nombre entier $n \in \mathbb{N}^*$: il est naturel de choisir son nombre de chiffres, mais dans quelle base (2, 10, 2^{32}...) ? $T(n) := \log(n)$ est un bon compromis, puisque le nombre de chiffre en base B est un $O(\log(n))$. Problème technique : $T(0)$ n'est pas définie, et on apprécie de manipuler des tailles entières ; au final, on définit souvent $T(n) := \lceil \log_2(n+1) \rceil$ (coût *logarithmique*, c'est le nombre de chiffres de l'écriture en base 2 de n).

  Cette taille est adaptée au calcul multiprécision, où le nombre de chiffres significatifs est non borné a priori. Si on ne traite que des entiers ou des flottants machine, en tout cas de taille fixée, on prendra $T = 1$ (coût *uniforme*). C'est ce que nous ferons.
- pour un problème de tri : le nombre d'objets à trier. Remarquons que si on trie N entiers distincts, la taille réelle en mémoire dans le modèle logarithmique est au minimum de l'ordre de $\sum_{i \leqslant N} \log(i+1) \sim N \log(N)$.
- pour du traitement d'image : le nombre de pixels, en supposant que le nombre de couleurs différentes est fixé.
- si le problème est d'afficher les 2^n parties d'un ensemble à n éléments, poser $T(n) := 2^n$ est plus adapté que $\log(n)$ comme dans le premier point.

Si nécessaire, on peut conserver plusieurs paramètres indépendants qui décrivent mieux le problème qu'une taille globale.

1.3. **Coût d'un algorithme.**

1.3.1. *Coût moyen / maximal.* Pour une fonction de coût $\mathcal{C}$ arbitraire définie sur l'ensemble des instances du problème, on s'intéresse à deux types de coûts :

Définition 1.2.

- dans le cas le pire (`worst-case cost`) :
$$\mathcal{C}^{\max}(N) := \max_{x, T(x)=N} \mathcal{C}(x),$$
où x parcourt les instances de taille N. Des instances de même taille peuvent avoir des coûts très différents.
- moyen (`expected cost`) :
$$\mathcal{C}^{\text{moy}}(N) := \sum_{x, T(x)=N} \mathcal{C}(x) \Big/ \sum_{x, T(x)=N} 1.$$

Plus généralement, on peut se donner une distribution de probabilité Pr sur $\{x, T(x) = N\}$ et évaluer

$$\mathcal{C}^{\text{moy}}(N, \Pr) := \sum_{x, T(x)=N} \Pr(t = x)\, \mathcal{C}(x)$$

pour mieux modéliser un problème. Le cas précédent correspond à la distribution uniforme.

Remarquer que ces différents coûts ne dépendent que de la taille des données. On s'intéresse essentiellement au temps d'exécution et à l'espace mémoire nécessaire.

1.3.2. *Coût en temps.* C'est le nombre d'opérations. En l'absence de formalisation, on est a priori conduit à compter séparément les opérations de chaque type (accès mémoire, opération arithmétique, branchement). Il est difficile d'évaluer les coûts relatifs des différentes opérations réellement effectuées par l'ordinateur après traduction de notre programme en langage machine ; on supposera que chaque instruction de base du langage s'exécute en temps $O(1)$.

En pratique, on choisit de privilégier un très petit nombre d'opérations décrétées « fondamentales » et de ne compter qu'elles. Ce choix est bon si le nombre total d'opérations est proportionnel au nombre d'opérations fondamentales.

Mais il devient alors difficile de comparer plus finement des programmes utilisant des opérations de types différents. Est-il rentable de remplacer une opération arithmétique par un accès mémoire ? Par cent ? Pour des algorithmes dont les coûts ont le même ordre de grandeur asymptotique, on en sera parfois réduit à comparer des implantations.

1.3.3. *Coût en espace.* C'est le nombre maximal de cases mémoires occupées simultanément au cours de l'exécution du programme. Ce coût est de moins en moins primordial, sauf contraintes techniques particulières (carte à puce), vu les évolutions techniques depuis le temps des cartes perforées.

Ce coût est au moins linéaire en la taille des données et du résultat. Pour certains problèmes spécifiques, faisant intervenir l'algèbre linéaire par exemple, il peut devenir crucial puisque la représentation naturelle d'une matrice carrée de dimension N a un coût N^2. Il est tout de même important de tenir compte de ce coût : à la limite, après un précalcul de toutes les instances de taille $\leqslant N$ ces instances se calculent dorénavant en temps $O(1)$: un accès dans une table, si celle-ci est bien organisée.

1.3.4. *Remarques.* Il peut y avoir de grandes disparités entre ces différents coûts, rendant des compromis désirables : par exemple augmenter le coût en mémoire pour diminuer le temps d'exécution. Parfois, le comportement est satisfaisant dans un cas générique, mais aberrant sur des cas dégénérés peu fréquents. Suivant les applications, il peut être important d'éviter à tout prix le cas le pire, en payant au besoin une (modeste) prime d'assurance ralentissant l'exécution sur le cas générique, ou de réduire au maximum le coût en mémoire.

A défaut d'autre indication, on considère les coûts en temps, dans le cas le pire. On peut raffiner les coûts définis ci-dessus : fragmentation mémoire, performance sur machines

parallèles, coût en « hasard » (produire des nombres « aléatoires », par exemple en cryptographie, n'est pas gratuit), possibilité de précalcul (amortissement), voire difficulté de l'implantation, du déboguage, des tests !

L'algorithmique fournit des méthodes pour résoudre un problème donné et des critères pour faire des choix rationnels parmi elles. Bien entendu, on s'efforce de compléter/vérifier ces analyses par une évaluation empirique des implantations effectives (pour des choix de données qui peuvent être réelles, aléatoires ou perverses).

1.4. **Complexité.** L'algorithmique fournit des résultat positifs, du type *l'algorithme A traite une instance de taille $\leqslant N$ en temps $T(N)$*. La complexité évalue la difficulté intrinsèque des problèmes : *tout algorithme capable de traiter toutes les instances de taille $\leqslant N$ a un coût minimal de $T(N)$ dans le cas le pire*. Nous aborderons à peine ce type de questions dans ce cours. On utilise souvent le mot *complexité* à la place de *coût* pour un algorithme, quand le contexte ne prête pas à confusion.

1.5. **Croissance comparée des fonctions.**

Définition 1.3. Soit $f, g : \mathbb{R} \to \mathbb{R}$. On écrit

- $g = O(f)$, s'il existe $C_0 > 0$ et x_0 tels que, pour $x \geqslant x_0$, on ait
$$|g(x)| \leqslant C_0 \, |f(x)| \, .$$
- $g = \theta(f)$, s'il existe $C_1, C_2 > 0$ et x_0 tels que, pour $x \geqslant x_0$, on ait
$$C_1 \, |f(x)| \leqslant |g(x)| \leqslant C_2 \, |f(x)| \, .$$
(Remarquer que $g = \theta(f) \Leftrightarrow f = \theta(g)$.)
- $g = \Omega(f)$, s'il existe $C_0 > 0$ et $x_n \to \infty$ tel que
$$|g(x_n)| \geqslant C_0 \, |f(x_n)| \, .$$

En pratique, on choisit f positive et monotone pour x assez grand ; en particulier $f(x)$ ne s'annule pas. La définition de $g = O(f)$, resp. $g = \theta(f)$ revient à dire que $|g/f|$ est majorée, resp. majorée et minorée non trivialement (0 est un minorant évident), indépendamment de N. Plus délicat, $g = \Omega(f)$ dit simplement que g/f ne tend pas vers 0.

La plupart du temps, on se contente d'une analyse asymptotique des coûts, quand la taille des instances tend vers l'infini, voire d'une majoration en $O(f(N))$ pour une fonction f élémentaire. On gagne en facilité de comparaison (ou d'analyse !) ce que l'on perd en précision, au demeurant souvent illusoire. Pour un problème de taille N, on qualifie souvent les coûts par le vocabulaire suivant

- $O(1)$: constant.
- $O(\log N)$: logarithmique.
- $O(N)$: linéaire (on voit souvent *sous-linéaire* pour $o(N)$).
- $O(N^2)$: quadratique.
- $O(N^k)$, pour une constante k indépendante de N : polynomial.
- $O(c^N)$, pour une constante $c > 1$ indépendante de N : exponentiel.

En théorie de la complexité un algorithme est décrété efficace s'il a un coût polynomial. En pratique, un algorithme en $O(N^2)$ et plus ne sera pas utilisable pour des grandes valeurs de N : doubler N multiplie le coût par 4. Un coût en $O(N(\log N)^k)$ reste raisonnable : doubler N multiplie le coût par une constante légèrement supérieure à 2. On ne peut pas espérer mieux qu'un coût linéaire si on doit traiter N données indépendantes.

Remarque : attention aux mirages de la complexité asymptotique. Un programme asymptotiquement rapide peut-être tout à fait inefficace sur de petites données. On a $1000 \log N > N$, pour $N \leqslant 9118$.

Qu'est-ce qu'un coût acceptable ? Il n'y a pas de réponse absolue ! Il faut être capable de prévoir approximativement les besoins d'un programme en temps et en espace, puis décider en fonction des ressources disponibles. Il peut être justifié d'utiliser un algorithme naïf inefficace[3], c'est même conseillé en première approche. Mais il faut le faire en connaissance de cause, en étant conscient des alternatives et des problèmes possibles... et documenter ses choix.

2. Quelques exemples

2.1. **Tri.** Introduisons d'abord le vocabulaire des problèmes de tri : on dispose de N objets $O_1, \ldots, O_N$, pris dans un certain ensemble E. L'ensemble E est muni d'une relation d'ordre total $<$; pour tout $a, b \in E$ on est dans l'un des trois cas mutuellement exclusifs : $a < b$ ou $a = b$ ou $b < a$. On note $a \leqslant b$ pour « $a < b$ ou $a = b$ ».

Par exemple, les objets sont des entiers, et l'on trie pour la relation d'ordre habituel sur $\mathbb{Z}$; ou bien une chaînes de caractères pour l'ordre lexicographique[4] (celui du dictionnaire). De façon plus flexible, on associe à chaque objet O_i une *clé* C_i, qui est par exemple un entier, et on trie les objets suivant les valeurs de leurs clés. On désire déterminer une permutation p de S_N telle que

$$O_{p(1)} \leqslant O_{p(2)} \leqslant \cdots \leqslant O_{p(N)}$$

La permutation est unique si les objets sont distincts. On peut préférer obtenir directement la liste triée

$$O_{p(1)}, \ldots, O_{p(N)}$$

sans sauvegarder la permutation. Le tri est dit *en place* si son coût en espace est $N + O(1)$ (les données et un nombre borné de variable auxiliaires). En particulier, un tri en place ne peut fournir p, seulement la liste triée.

[3]Par exemple, à cause du coût négligeable par rapport au reste du programme, de la complexité de l'implantation/des tests, ou à cause des données qui seront réellement utilisées. (Quoique, un programme est fréquemment utilisé en dehors du cadre initialement prévu, avec des conséquences potentiellement désastreuses.)

[4]$(a_1, \ldots, a_n) < (b_1, \ldots, b_n)$ si et seulement si il existe un indice j tel que $a_i = b_i$ pour $i < j$, mais $a_j < b_j$; on ordonne arbitrairement l'alphabet.

2.2. **Tri par sélection.** Ce tri trouve l'élément minimal et le place en premier ; puis l'élément minimal parmi ceux qui restent, etc.

Algorithme 2.1 (Tri sélection)
Pour $i := 1, \dots, N$, effectuer

(1) Poser $\min := i$.
(2) *[Trouver le i-ème minimum]* Pour $j := i+1, \dots, N$
  (a) Si $O_j < O_{\min}$, poser $\min := j$.
(3) Poser $p(i) := \min$; échanger O_i et $O_{\min}$.

Dans l'algorithme ci-dessus, on peut supprimer l'instruction $p(i) := \min$ pour un tri en place ; l'échange se fait par référence ou échange de pointeur, *pas* par copie, coûteuse si les objets sont gros.

Analyse : on a

$$\sum_{1 \leqslant i \leqslant N} N - i = N(N-1)/2 = O(N^2)$$

comparaisons et affectations dans la boucle intérieure, et $O(N)$ échanges et affectations dans la boucle extérieure. Coût : $O(N^2)$ en temps, et $O(N)$ en espace. Les coûts moyens sont presque identiques, seule l'affectation $\min := j$ dépend des données. Verdict : trop lent si N est grand.

Exercice 2.2– Donner un algorithme trouvant le min et le max d'un tableau de N objets, qui fasse de l'ordre de $3N/2$ comparaisons dans le cas le pire.

2.3. **Tri par comptage.** On fait l'hypothèse supplémentaire que les clés sont des entiers compris entre 1 et M (la méthode se généralise facilement pour M entiers consécutifs quelconques). On stocke le nombre d'occurrences de chaque clé dans un tableau, puis on redistribue :

Algorithme 2.3 (Tri comptage (`bucket sort`))

(1) *[Initialiser]* Pour $j := 1, \dots, M$, poser $T[j] := 0$.
(2) *[Compter]* Pour $i := 1, \dots, N$, incrémenter $T[C_i]$ de 1.
(3) Pour $j := 2, \dots, M$, effectuer $T[j] := T[j] + T[j-1]$.
(4) *[Distribuer]* Pour $j := N, \dots, 1$, effectuer
  (a) Poser $p(T[C_j]) := j$
  (b) Décrémenter $T[C_j]$ de 1.

Au début de l'étape 3, $T[j]$ est le nombre de clés égales à j. Au début de l'étape 4, c'est le nombre de clés $\leqslant j$. En particulier $T[\max C_j] = \cdots = T[M] = N$.

Analyse : Si on suppose que N est inférieur au plus grand entier machine `MAXINT` (par exemple `MAXINT` $= 2^{32} - 1 \approx 4.29\,10^9$), les opérations sur $T[j]$ se font en temps/espace $O(1)$. Dans ce cas, la complexité en espace est $O(M+N)$ en temps et en espace (idem en moyenne) ; c'est excellent si $M = O(N)$ (coût linéaire).

Exercice 2.4– Montrer que p contient bien une permutation convenable à la fin de l'algorithme.

Exercice 2.5– Comment se modifie l'analyse sans hypothèse sur N ? [*Les fonctions d'allocation mémoire sont incapables d'adresser des tableaux de taille supérieure à* `MAXINT`*, indépendamment de la mémoire de la machine. Pour trier des tableaux de plus grande taille, il faut utiliser le stockage hors mémoire vive (tris* externes*).*]

2.4. **Recherche séquentielle.** Soit $O_1, \ldots, O_N$ un tableau de N objets. On désire savoir si l'objet O est dans la liste, et si oui à quelle position. L'algorithme évident fait une recherche séquentielle :

Algorithme 2.6 (Recherche séquentielle)

(1) Poser $i := 1$.
(2) Tant que $O_i \neq O$ et $i \leqslant N$
  (a) Incrémenter i de 1.
(3) Si $i > N$, échec. Sinon renvoyer i.

Exercice 2.7– Quel sont les coûts associés à cet algorithme en cas d'échec ? En cas de succès ? [*pour le calcul du coût moyen, on supposera les O_i distincts*].

Calculer le coût moyen dans le cas d'une distribution de probabilité p des requêtes (l'objet O_i est requis avec probabilité p_i, avec $\sum p_i = 1$). Recalculer le coût moyen en supposant que cette distribution est connue et que les O_i sont classés de façon optimale. Applications :

(1) $p_i = 1/2^{i+1}$, $i < N$, p_N convenable.
(2) $p_i = (N - i + 1)c$, c convenable.
(3) $p_i = c/i$, c convenable.

2.5. **Recherche par dichotomie.** On reprend la situation du §2.4, en supposant maintenant que le tableau est *trié*. On note $\lfloor x \rfloor$ la partie entière de x, et on appelle *milieu* de $[a, b]$ le point $\lfloor (a + b)/2 \rfloor$. On peut appliquer l'idée familière de la dichotomie pour accélérer la recherche : on cherche $i \in [1, N]$ tel que $O_i = O$, on compare O avec O_M où M est le milieu de $[1, N]$

- Si $O < O_M$, alors, $i \in [1, M - 1]$.
- Si $O > O_M$, alors, $i \in [M + 1, N]$.
- Si $O = O_M$, on a fini !

Il suffit de recommencer, jusqu'à ce que l'on ait trouvé O, ou jusqu'à ce l'intervalle soit de longueur $\leqslant 1$.

Exercice 2.8– Écrire formellement l'algorithme de dichotomie, et prouver qu'il termine. Quels sont les coûts associés en cas d'échec ? En cas de succès ? [*dans ce dernier cas, l'analyse du coût moyen n'est pas immédiate, nous y reviendrons après avoir introduit les arbres binaires*[5]]

[5]On peut simuler la recherche par un chemin dans un arbre binaire parfait, dont les noeuds contiennent les objets ; on numérote les noeuds à partir de la racine et le nombre de comparaisons nécessaires pour découvrir l'objet stocké dans le noeud i (qui n'est pas O_i en général) est la profondeur du noeud. Au final, le nombre moyen de comparaisons est équivalent au coût dans le cas le pire.

2.6. **Récursion.** De nombreux algorithmes procèdent comme la dichotomie : ils découpent le problème en sous-problèmes de taille strictement inférieure, puis résolvent de la même manière les sous-problèmes. Cette manière de procéder est dite *récursive*, et correspond au raisonnement par récurrence des petites classes ; lorsque la taille est très petite[6], le problème devient trivial. On utilise souvent la formule imagée de *diviser pour régner* (`divide and conquer`).

Remarque : Les algorithmes précédents ont été décrits de manière *itérative*, au moyen de boucles (itérer = répéter). Un programme récursif peut toujours s'écrire de manière itérative ; la variante itérative est en général légèrement plus efficace, mais illisible.

Exercice 2.9– Exprimer de façon récursive les algorithmes précédents qui s'y prêtent.

Exercice 2.10– [*Le tri-fusion (*`mergesort`*)*]. On définit le *plafond* de x par $\lceil x \rceil := x - \lfloor x \rfloor$. Pour trier un tableau de taille N, on le coupe en 2 parties de taille $\lfloor N/2 \rfloor$ et $\lceil N/2 \rceil$, on trie chacune des parties récursivement, et on fusionne ces 2 tableaux *triés* avec l'algorithme évident en $O(N)$. Écrire formellement l'algorithme. Montrer que son coût en temps est $O(N \log N)$. Quel est son coût en espace ?

Exercice 2.11– Soit $C : \mathbb{R}^+ \to \mathbb{R}^+$ une fonction vérifiant $C(x) = 0$ pour $x < 1$ et et $C(x) \leqslant aC(x/b) + cx$, pour certains $a > 0$, $b > 1$, $c \in \mathbb{R}$ fixés, et tout $x \in \mathbb{R}^+$. Donner une borne supérieure pour $C(x)$ quand $x \to \infty$ [$O(x^{\log_b a})$ *pour* $a > b$, $O(x \log x)$ *pour* $a = b$, *et* $O(x)$ *pour* $a < b$.]

2.7. **QuickSort.** Le tri rapide (Hoare, 1960) trie en place un tableau S de taille N en choisissant un élément pivot x, et en séparant les entrées en deux sous-tableaux suivant qu'elles sont $< x$ ou $> x$. Le pivot x peut alors être rangé à sa place définitive, et on trie récursivement les deux sous-tableaux de la même façon.

Exercice 2.12– Écrire formellement l'algorithme, en prenant pour pivot le premier élément [*attention aux boucles infinies si le pivot est le min ou le max*]. Le cas le pire est mauvais, en $O(N^2)$; sur quels types de scénario ? Quel est le scénario optimal ? Comment améliorer ? Complexité en espace (cas le pire) ?

Théorème 2.13. *On suppose que le pivot x est choisi uniformément au hasard. Soit $C(N)$ le nombre moyen de comparaisons dans l'algorithme sur un tableau de longueur N ; on a $C(N) \sim 2N \log N$.*

Preuve. Il y a $N + O(1)$ comparaisons dans la procédure avant récursion : $N - 1$ comparaisons entre x et les autres éléments du tableau, et suivant l'implantation exacte on a 0, 1 ou 2 comparaisons supplémentaires avec des sentinelles. Pour les besoins de cette démonstration, on prendra $N + 1$.

[6] Typiquement 0 ou 1 ; en pratique, on peut gagner une constante multiplicative en arrêtant la récursion plus tôt, et en finissant « à la main » avec un algorithme optimal sur les petites instances. Ces seuils dépendent de l'implantation.

Le pivot x est le i-ème plus grand élément de S avec probabilité $1/N$; dans ce cas $|S_1| = i-1$ et $|S_2| = N-i$, soit

$$C(N) = N + 1 + \frac{1}{N}\sum_{i=1}^{N}[C(i-1) + C(N-i)]$$

Soit

$$NC(N) = N(N+1) + 2\sum_{i=0}^{N-1} C(i-1)$$

Par soustraction

$$NC(N) - (N-1)C(N-1) = 2N + 2C(N-1),$$

soit, après simplification et division par $N(N+1)$,

$$\frac{C(N)}{N+1} = \frac{C(N-1)}{N} + \frac{2}{N+1} = \cdots = 2\sum_{i=1}^{N+1}\frac{1}{i}.$$

□

Voici une variante légèrement plus efficace : l'idée est de choisir un échantillon au hasard dans le tableau, et de retenir le médian de l'échantillon comme pivot global.

Algorithme 2.14 (QuickSort(g, d), variante « Médian de 3 »)
S est la liste des entrées à trier en place ; les arguments sont g(auche) et d(roite), indices délimitant la partie du tableau à trier $S[g], \ldots, S[d]$.

(1) [*Cas trivial*] Si $d - g \leqslant 1$, trier la liste en échangeant au besoin les 2 éléments (s'il y en a 2), et quitter la procédure.
(2) [*Choix d'un pivot*] Choisir trois éléments dans S, les trier $s_1 \leqslant s_2 \leqslant s_3$, et poser $x := s_2$. Échanger s_1 avec $S[g]$, $x = s_2$ avec $S[d-1]$, et s_3 avec $S[d]$.
(3) Poser $i := g + 1$ et $j := d - 2$.
(4) [*Écrire $S = S_1 \cup \{x\} \cup S_2$, où $S_1 := \{s \in S, s < x\}$ et $S_2 := \{s \in S, s > x\}$*].
Répéter indéfiniment [*en fait, jusqu'à ce que les pointeurs i et j se croisent*]
  (a) Tant que $S[i] < x$, incrémenter i de 1.
  (b) Tant que $S[j] > x$, décrémenter j de 1.
  (c) Si $i \geqslant j$, aller au Pas (5).
  (d) [*On a $S[j] \leqslant x \leqslant S[i]$*]. Échanger $S[i]$ et $S[j]$.
(5) Échanger $S[d-1]$ $(= x)$ et $S[i]$ $(\geqslant x)$. x est à sa position définitive.
(6) QuickSort(g, $i-1$).
(7) QuickSort($i+1$, d).

A cause du choix de pivot fait en (2), x ne peut être égal au min ou au max du tableau ; ce qui implique que les deux boucles de la boucle intérieure terminent. Voir aussi [3, §5.2.2] pour une implantation optimisée ou [4, Chapitre 9] pour une présentation plus accessible, dérécursifiée. Plus généralement, on peut prendre un échantillon de $2k+1$ valeurs, $k \geqslant 1$, sans grande variation d'efficacité pratique.

Remarque 2.15. Si on arrête la récursion à la taille M pour finir avec un algorithme naïf, on obtient la relation

$$C(N) = N + 1 + \frac{1}{N}\sum_{i=1}^{N}[C(i-1) + C(N-i)], \quad \text{pour } N > M$$

ce qui ne change pas le terme principal du Théorème 2.13 pour $M = O(1)$.

Remarque 2.16. Pour la variante « Médian de 3 », on obtient la récurrence

$$C(N) = N + 1 + \sum_{i=1}^{N} \frac{(N-i)(i-1)}{\binom{N}{3}}[C(i-1) + C(N-i)].$$

La résolution est difficile ; on obtient $C(N) \sim (12/7)N \log N$ et plus généralement

$$C_k(N) \sim \left(\sum_{i=k+2}^{2k+2} \frac{1}{i}\right)^{-1} N \log N$$

pour la variante « Médian de $2k+1$ » [3, Exercice 5.2.2-29].

★ **Exercice 2.17**– Modifier QuickSort en un algorithme QuickSelect qui trouve le k-ième plus grand élément dans une liste de N objets, pour $k \in [1, N]$ fixé (par exemple, le médian) ? Montrer que la complexité en moyenne est linéaire.

★★ **Exercice 2.18**– Comment déterminer le k-ième plus grand élément dans une liste de N objets en temps linéaire ? [*L'idée est de modifier QuickSelect pour garantir qu'une proportion δ des éléments est éliminée à chaque étape, avec $\delta \geqslant \delta_0 > 0$ minoré uniformément. Pour B un petit entier fixé, par exemple 5, séparer les éléments par groupes de B, calculer le médian de chaque groupe en $O_B(1)$ comparaisons. Trouver le médian M de ces médians par application récursive de l'algorithme, utiliser M pour partitionner le tableau initial, puis itérer comme dans QuickSelect.*]

3. Structures de données

Les données des exemples précédents sont organisées d'une façon simple mais précise, comme tableau de cases contigües, de taille fixée à l'avance. On s'intéresse ici à diverses façons utiles d'organiser les données en mémoire, plus flexibles, que nous utiliserons par la suite. Une structure de données est *statique* si sa capacité est fixée au moment de sa création, *dynamique* si elle est variable.

3.1. Deux implantations.

On examine maintenant différentes implantations du concept de *liste*, i.e. de suite ordonnée finie d'éléments (x_i). On a par exemple besoin des primitives suivantes

- Lire/Écrire l'élément x_i
- Supprimer/Insérer un élément en position i. Par rapport à Lire/Écrire, cette primitive décrémente/incrémente le numéro d'ordre des éléments d'indice $j > i$.
- Successeur/Prédécesseur (trouver $x_{i\pm1}$ étant donné x_i).

3.1.1. *Tableaux (unidimensionnels).* Ensemble fini d'éléments (de même type), stockés dans des zones contigües de la mémoire : $T[1], \ldots, T[N]$.

Avantage : Lire/Écrire/Successeur/Prédécesseur se fait en temps $O(1)$.

Inconvénient : Statique. Peut se remplir au fur et à mesure si le nombre maximal d'éléments est connu à l'avance ; mais agrandir le tableau impose de le recopier (coût $O(N)$).

Exercice 3.1– Simuler un tableau à k dimensions dans un tableau à une dimension comme ci-dessus.

Exercice 3.2– On implante un tableau « dynamique » en allouant initialement un tableau de taille 1, et puis en doublant sa taille (et recopiant tous ses éléments) en cas de besoin. Quel est le coût de ce dispositif si le nombre d'éléments finalement stockés est N ? Y aurait-t-il avantage à multiplier la taille du nouveau tableau par une autre constante que 2 ? à prendre une taille initiale différente de 1 ?

3.1.2. *Listes chaînées.* Une *cellule* est un objet comportant 2 champs : un contenu (objet arbitraire, de type prédéfini), et un lien (pointeur) sur une cellule. Une *liste chainée* est un pointeur sur une cellule (pointeur `NULL` pour la liste vide).

Avantage : Dynamique. Successeur en temps $O(1)$. Insérer/Supprimer en position i en temps $O(1)$ à partir d'un pointeur sur l'élément $i-1$.

Inconvénient : Accès séquentiel obligatoire. Lire/Écrire/Prédécesseur du i-ème élément en temps $O(i)$. Coût en espace par rapport à un tableau contenant le même nombre d'éléments.

Exercice 3.3– En C++ (ou en C), quel est le cas le pire (en terme des types stockés) pour le surcoût en espace d'une liste, par rapport à un tableau de N éléments ?

Une liste *doublement chaînée* est un pointeur sur une cellule comportant un contenu et deux liens (précédent/suivant). Permet de faire passer Prédécesseur de $O(i)$ à $O(1)$.

Exercice 3.4– Transformer le tri par sélection du §2.2 en tri par insertion dans une liste chaînée : pour chaque élément O_j, $j = 2, \ldots, N$, on compare O_j avec les O_i, $i < j$ et on insère O_j à l'endroit convenable (on le laisse en place s'il est supérieur à tous les O_i, $i < j$). S'agit-il d'une amélioration ?

On peut implanter l'algorithme dans un tableau. Dans ce cas, la recherche du point d'insertion peut se faire en temps logarithmique par dichotomie. Mais il faut décaler tous les éléments suivant ce point d'un cran vers la droite. Évaluer la complexité de cette nouvelle implantation.

Exercice 3.5– Reprendre l'algorithme de recherche séquentielle du §2.4, en stockant les objets dans une liste chaînée, et en modifiant la liste après chaque recherche couronnée de succès suivant l'une des deux stratégies suivantes

(1) en plaçant l'élément en tête de liste.
(2) en décalant l'élément d'un cran vers la tête de liste.

Quels sont les avantages espérés ? Comment implanter cette recherche adaptative en gardant la structure de tableau ?

3.2. **Piles/Files.** Ce sont des variantes abstraites des listes, à accès restreint. Dans une *pile* (`stack`[7]), lecture/insertion/suppression se font sur un seul élément, en fin de liste. Peut s'implanter par une liste chaînée, ou par un tableau si la taille maximale de la pile est connue.

Dans une *file* (`queue`[8]), l'ajout se fait en fin de liste, la suppression en début de liste ; peut s'implanter par une liste doublement chaînée, ou bien par un tableau qu'on commence à remplir par le milieu si la taille maximale de la file est connue à l'avance.

Exercice 3.6– Déclarer une classe liste/file/pile, implantée par des listes chaînées, et écrire les opérations d'ajout/suppression d'un élément. Trouver des applications des structures files et piles.

Exercice 3.7– Comment implanter une file par un tableau sans perdre de place aux extrémités ? (Attention : avec la solution « naturelle », il y a un problème pour distinguer une pile vide d'une pile pleine.)

Exercice 3.8– Un mot *bien parenthésé* est une suite de caractères (et) telle qu'à chaque parenthèse ouvrante, on puisse faire correspondre une parenthèse fermante plus loin dans le mot. Plus formellement, le mot vide est bien parenthésé, et (x) est bien parenthésé si et seulement si x l'est. On suppose que l'utilisateur entre les parenthèses une par une sur un clavier ; donner un algorithme vérifiant le bon parenthésage d'un mot. Généraliser au cas où l'on a parenthèses et crochets.

Exercice 3.9– Un *palindrome* est un mot qui se lit de la même manière de gauche à droite et de droite à gauche. On suppose que l'utilisateur entre les lettres une par une sur un clavier ; donner un algorithme utilisant 2 piles, qui vérifie si un mot est un palindrome. Quel est son coût (essayer de minimiser le nombre d'opérations de piles) ?

3.3. **Arbres.** Un arbre est un graphe d'un type particulier qui permet l'ajout et la recherche « rapide » d'un élément. Du point de vue théorique, les arbres permettent d'étudier les processus récursifs.

Définition 3.10. Définition récursive : un arbre est soit vide, soit composé d'un noeud distingué (racine, `root`)[9] directement relié à (pointant sur) une suite[10] finie d'arbres.

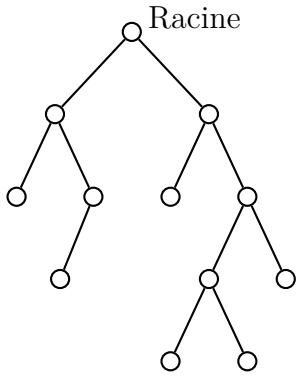
[7] ou `LIFO list` = `Last In First Out` (démodé).

[8] ou `FIFO list` = `First In First Out` (démodé).

[9] On dit plutôt cellule pour les listes, et noeud pour les arbres (ou sommet pour un graphe). C'est la même notion de conteneur : un contenu arbitraire, et une collection de pointeurs vers un conteneur du même type.

[10] ordonnée, donc. Éventuellement vide

Conventionnellement, un arbre est représenté « à l'envers », en descendant de la racine aux feuilles, comme un arbre généalogique :



Définition 3.11.

- *Fils* (`offspring`, `son`) du noeud x : les racines des arbres rattachés à x.
- Si chaque noeud a exactement 2 fils (éventuellement vides), l'arbre est dit *binaire*[11] (`binary tree`) ; on parle alors de fils gauche et fils droit (`left/right son`). En particulier, même s'il n'y a qu'un fils, il est droit ou gauche.
- *Père* (`parent`, `father`) du noeud x : l'unique noeud dont x est le fils (la racine n'a pas de père).
- *Frères* (`siblings`, `brothers`) du noeud x : noeuds qui ont le même père.
- Noeud *interne* (`internal node`) : un noeud qui a au moins un fils.
- *Feuille* (`leaf`) : un noeud qui n'a pas de fils.

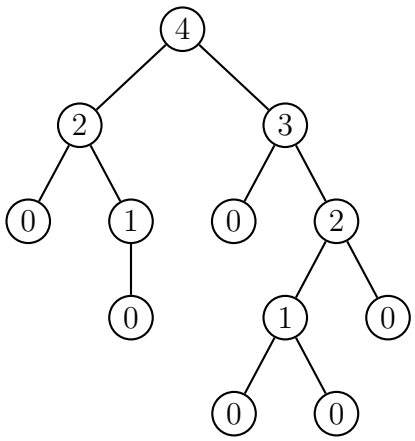
Définition 3.12.

- Profondeur (`depth`) du noeud x : la longueur (en nombre d'arcs) du chemin joignant la racine à x.
- Hauteur (`height`) du noeud x : la longueur du plus long chemin de x à une feuille. Un nœud est une feuille si et seulement si il est de hauteur 0.
- Hauteur d'un arbre : la hauteur de la racine. Par convention, la hauteur de l'arbre vide est -1.

Lemme 3.13. *La hauteur $h(x)$ d'un sommet interne x est égal à $1 + \max h(y)$, où y parcourt les fils de x.*

Voici par exemple un arbre où chaque nœud x est étiqueté par sa hauteur :

[11] Plus généralement l'arbre est k-aire, ou d'arité k, si chaque noeud a k fils.



Lemme 3.14. *La hauteur d'un arbre d'arité k à $N \geqslant 1$ sommets est comprise entre $\lceil \log_k(N+1) \rceil - 1$ et $N-1$.*

Preuve. Exercice. □

Remarquons que le résultat reste vrai pour $N = 0$, avec la convention que l'arbre vide est de hauteur -1.

Exercice 3.15– Déduire du lemme qu'un algorithme triant N objets en ne faisant que des comparaisons a une complexité dans le cas le pire en $\Omega(N \log N)$. [*Construire l'arbre de décision associé à la suite des comparaisons effectuées. Ses feuilles sont les $N!$ permutations possibles.*]

Exercice 3.16– Soit $\alpha \geqslant 1$ fixé. S'inspirer de l'exercice précédent pour montrer qu'aucun algorithme de tri basé sur des comparaisons ne peut résoudre plus de $N!/\alpha^N$ instances distinctes du tri à N éléments en temps linéaire $O(N)$. [*Moralité : aucun de ces algorithmes n'a beaucoup d'excellents cas* .]

3.4. **Exemple : les ABR.** Un Arbre Binaire de Recherche (ABR, `BST` = `Binary Search Tree`) est un arbre binaire dont les nœuds contiennent des clés vérifiant la propriété suivante : la racine d'un sous-arbre est supérieure à tous les éléments de son sous-arbre gauche, et inférieure à tous les éléments de son sous-arbre droit.

Étant donné une clé quelconque on parcourt l'ABR à partir de la racine pour savoir si la clé s'y trouve : on fait une comparaison par nœud, en prenant la branche indiquée par la comparaison. La recherche échoue si on arrive à une feuille sans avoir trouvé la clé. On insère la nouvelle clé dans l'arbre (comme feuille) à la position obtenue à la fin de la recherche et on obtient un nouvel ABR.

Exercice 3.17– Écrire l'algorithme construisant l'ABR associé à un suite de clés en partant de l'arbre vide et en rajoutant successivement les clés non déjà présentes dans l'arbre.

Complexité d'une recherche dans l'arbre en fonction de sa hauteur ? de la construction de l'arbre ? Quelle est le coût moyen de la construction d'un ABR par insertions successives de N clés distinctes ? [*l'analyse est la même que pour QuickSort*].

Théorème 3.18. *Soit $H(N)$ la hauteur moyenne d'un ABR construit par insertions successives de N clés distinctes. On a $H(N) \sim 2 \log N$.*

Preuve. Voir [3, §6.2.1–2]. □

Avantage d'un ABR : Dynamique. Ajout/Suppression et Recherche se font en temps $O(\log N)$ si l'arbre est bien équilibré (de hauteur essentiellement minimale, cf. Lemme 3.14). Alors que la dichotomie a une insertion en $\Omega(N)$.

Inconvénient : Ajout/Suppression/Recherche en $O(N)$ dans le cas le pire. Coût en espace par rapport à un tableau contenant le même nombre d'éléments.

Exercice 3.19– Généraliser la notion d'ABR et l'Exercice 3.17 pour un arbre k-aire de recherche ($k > 2$).

Exercice 3.20– Écrire les algorithmes correspondant aux opérations suivantes sur un ABR :

(1) recherche du minimum.
(2) recherche du maximum.
(3) écriture de la liste des éléments dans l'ordre croissant.
(4) même question pour l'ordre décroissant.
(5) successeur immédiat d'un sommet (pour la relation d'ordre).
(6) prédécesseur immédiat d'un sommet.
(7) liste des éléments de rang compris entre deux indices fixés $a \leqslant b$.

Quelle est leur complexité ? [*Choisir des paramètres appropriés*]

4. Arbres binaires et équilibrage

On a besoin par exemple des primitives suivantes

- Racine.
- Fils gauche/droit d'un sommet (éventuellement VIDE)
- Père d'un sommet (VIDE pour la racine).
- Attacher fils gauche/droit (étant donné un sommet père, et un nouveau sommet fils).
- Supprimer un sommet (en se donnant des règles pour répartir les éléments de ses sous-arbres).

4.1. **Implantation.** L'idée naturelle, récursive, utilise des listes chaînées. La structure `sommet` a trois champs

- un contenu (pointeur sur un objet en général).
- un pointeur sur un fils gauche (éventuellement `VIDE`).
- un pointeur sur un fils droit (éventuellement `VIDE`).

Un arbre est donné par un pointeur sur son sommet racine. Un arbre vide se représente par une sentinelle, par exemple en C/C++ un pointeur `NULL`.

Exercice 4.1– Implanter la primitive Père à partir de cette structure.

De même que pour les listes doublement chaînées, il peut être utile de rajouter un champ père dans la structure *sommet*, pour faire passer l'opération Père de $O(\text{hauteur})$ à $O(1)$.

Exercice 4.2– Réfléchir à une implantation d'un arbre binaire à base de tableaux (Contenu, FilsGauche, FilsDroit) indexés par les numéros des sommets. Avantages et inconvénients ?

Exercice 4.3– Soit $A(N)$ le nombre d'arbre binaires à N sommets ; on pose $A(0) := 1$ par convention (l'arbre vide). Montrer que $A(N) = \sum_{k\geqslant 0} A(k)A(N-1-k)$. Poser $\alpha(x) := \sum_{N\geqslant 0} A(N)x^N$, et en déduire que $x\alpha^2(x) - \alpha(x) + 1 = 0$, puis que

$$A(N) = \frac{1}{N+1}\binom{2N}{N} \sim \frac{4^N}{\sqrt{\pi}N^{3/2}}$$

4.2. **Parcours.** Le but est de visiter une fois et une seule chaque sommet de l'arbre (pour un traitement quelconque). La programmation récursive admet trois variantes :

```
Parcours(sommet s) {
  si s != VIDE, alors {
    // (1)
    Parcours( FilsGauche(s) );
    // (2)
    Parcours( FilsDroit(s) );
    // (3)
  }
}
```

Si les primitives sont en $O(1)$, le parcours se fait en $O(N)$, où N est le nombre de sommets. Si l'instruction de traitement (par exemple, affichage) est placée en

(1) parcours *préfixe* (`preorder`).
(2) parcours *infixe*, ou *symétrique* (`inorder`, `postorder`).
(3) parcours *postfixe* (`endorder`).

Exercice 4.4– Décrire des fonctions non-récursives correspondant aux trois types de parcours [*Utiliser une pile*]. Décrire une fonction de parcours *hiérarchique* (ou militaire) : on visite les nœuds de gauche à droite, par niveau décroissant, en commençant par la racine [*Transformer la pile en file*].

Exercice 4.5– Soit un arbre binaire dont les N sommets sont $u_1, \ldots, u_N$ en parcours préfixe et $u_{p(1)}, \ldots, u_{p(N)}$ en parcours infixe, pour une permutation $p \in S_N$. Trouver un algorithme permettant de décider si une permutation p correspond à un tel parcours et, le cas échéant, de reconstruire l'arbre à partir de p.

Exercice 4.6– On suppose maintenant que $u_{p(1)}, \ldots, u_{p(N)}$ dénote le parcours *postfixe*. Même question qu'à l'exercice précédent [*on n'a pas unicité de l'arbre, cette fois*]. Montrer qu'une permutation est admissible si et seulement si elle peut être obtenue en empilant et dépilant (dans un ordre convenable) $u_1, \ldots, u_N$, présentés dans cet ordre à l'entrée d'une pile.

Exercice 4.7– Donner un algorithme permettant de supprimer un élément d'un ABR, en gardant la structure d'ABR.

4.3. **Arbres AVL.** La complexité des algorithmes de Recherche, Ajout, Suppression dans un ABR est majorée par la hauteur de l'arbre. On a donc tout intérêt à la minimiser. Si

l'arbre est fixe, on peut reconstruire un arbre quasi-parfait[12] à partir de ses éléments[13]. Bien sûr ceci change la structure combinatoire de l'arbre, vu comme graphe, mais ce qui nous importe ici, est de conserver la propriété ABR pour faire des recherches.

Exercice 4.8– Écrire un algorithme qui transforme un ABR en ABR comportant les même nœuds, de hauteur minimale.

Dans une situation dynamique, où de nouvelles clés sont insérées, on pourrait exécuter ce type d'algorithme à intervalles réguliers (toutes les k insertions), ou lorsqu'on constate que l'arbre se déséquilibre. Nous allons voir qu'on peut assurer sans surcoût notable, qu'à tout moment l'arbre soit bien équilibré.

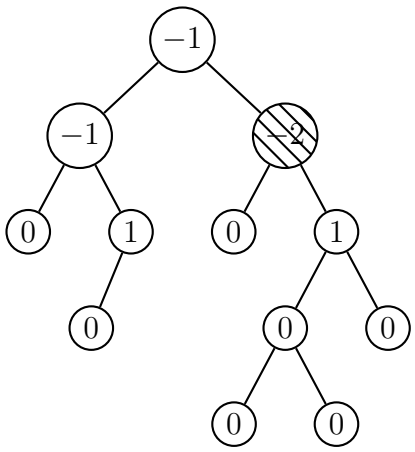
Définition 4.9. Pour un sommet x d'un arbre, on note $G(x)$ son sous-arbre gauche et $D(x)$ son sous-arbre droit, puis $d(x)$ le *déséquilibre* de x, défini comme la différence

$$d(x) := \text{hauteur}(G(x)) - \text{hauteur}(D(x)).$$

(Rappelons que la hauteur de l'arbre vide est -1 par définition).

Définition 4.10. Un arbre AVL (du nom de ses deux inventeurs Adel'son-Vel'skiĭ et Landis, 1962) est un ABR tel que pour tout sommet x, on ait $|d(x)| \leqslant 1$.

Voici par exemple un arbre déséquilibré, où chaque nœud x est étiqueté par $d(x)$:



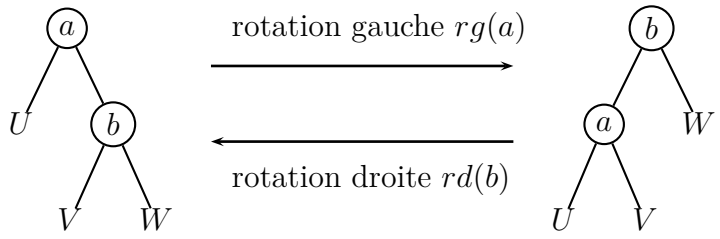
Exercice 4.11– Soit $S(h)$ le nombre minimal de sommets d'un AVL de hauteur h. Montrer que $S(h+1) = S(h) + S(h-1) + 1$. En déduire que la hauteur d'un AVL de N sommets est $O(\log N)$. Donner une borne précise utilisant le nombre d'or $\phi := (1+\sqrt{5})/2$.

Après chaque insertion dans un AVL, on regarde si elle entraîne un déséquilibre, et si oui, on rééquilibre par des *rotations*. Une rotation (gauche ou droite) est définie par le

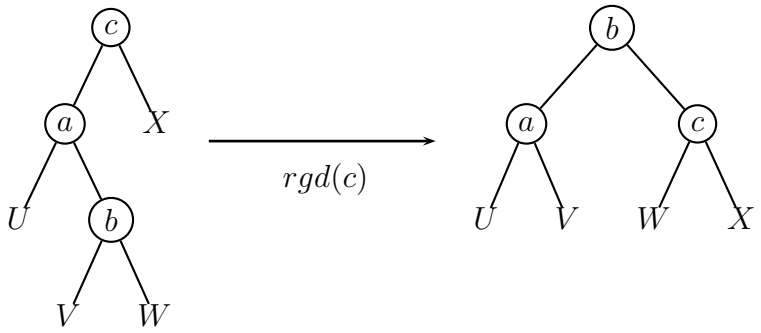
[12] dont toutes les feuilles sont situés sur un ou deux niveaux consécutifs.

[13] Notons que ceci n'est intéressant que si les recherches sont équiprobables, ce qui est rarement le cas en pratique. Voir [3, §6.2.2] pour la construction d'un arbre optimal pour une distribution de probabilité fixée sur les requêtes.

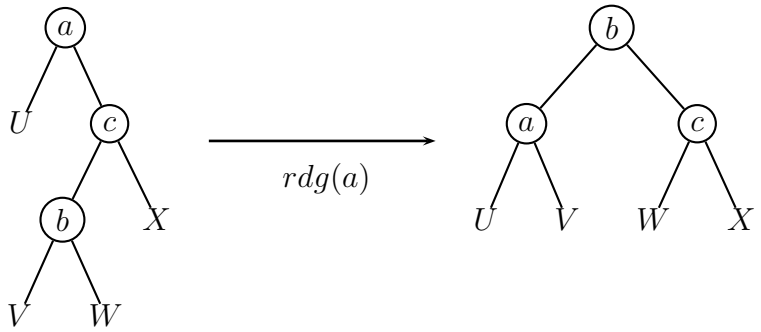
schéma suivant :



où a, b sont des nœuds et U, V, W (et plus tard X) sont des arbres quelconques, éventuellement vides. Une rotation est une opération locale, manifestement en $O(1)$, qui préserve le parcours infixe de l'arbre. Visualisons deux rotations composées utiles :
• $rgd(c) = rg(G(c))$, suivie de $rd(c)$:



• $rdg(a) = rd(D(a))$, suivie de $rg(a)$:



L'algorithme AVL considère un arbre AVL, qui vient d'être déséquilibré par une insertion. Soit x le sommet le plus bas déséquilibré ($d(x) = \pm 2$), alors on se trouve dans l'un des 4 cas suivants

- Si $d(x) = 2$ et $d(G(x)) = -1$, effectuer $rgd(x)$.
- Si $d(x) = 2$ et $d(G(x)) = 1$, effectuer $rd(x)$.
- Si $d(x) = -2$ et $d(D(x)) = -1$, effectuer $rg(x)$.
- Si $d(x) = -2$ et $d(D(x)) = 1$, effectuer $rdg(x)$.

Exercice 4.12– Montrer que les 4 cas ci-dessus couvrent tous les cas possibles, et restaurent un AVL. Écrire un algorithme détaillé de recherche, suivie d'insertion en cas d'échec. Pour implanter raisonnablement l'algorithme AVL, on rajoute au type nœud un champ `déséquilibre`,

qui permet de calculer d trivialement. [*Remonter le long de l'arbre à partir de la feuille insérée. Rajouter un champ* `Père`*, ou utiliser la récursivité (et une pile implicite) pour se souvenir du parcours de recherche*].

4.4. **Arbres Rouges-Noirs (ARN).**

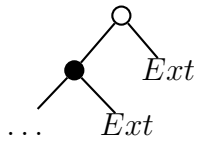
Définition 4.13. Un sous-arbre vide d'un nœud d'un arbre binaire (interne ou feuille), est appelé *nœud externe*.

Attention un nœud externe n'appartient pas à l'arbre (d'où son nom). C'est une définition commode qui permet de dire que chaque véritable nœud d'un arbre binaire a toujours deux sous-nœuds. On a donc des nœuds internes (au moins un fils), les feuilles (pas de fils), et les nœuds externes (fils « complémentaires » des nœuds ayant au plus un fils).

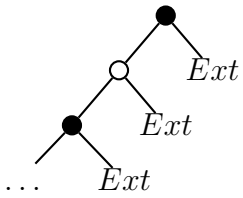
Les arbres Rouges-Noirs (`Red-Black tree`, ou `RB-tree`, inventés par Rudolf Bayer, 1972) sont des ABR dont les nœuds sont colorés en rouge ou noir, suivant les deux règles :

- Si un noeud est rouge, ses descendants immédiats (0, 1 ou 2) sont noirs.
- Chaque chemin de la racine à un nœud externe contient le même nombre de nœuds noirs.

En particulier, ces règles interdisent un arbre rectiligne de trois sommets ou plus. En effet sur les 3 premiers sommets, soit la racine est rouge et le deuxième sommet noir :



donc l'un des chemins partant de la racine ne contient aucun sommet noir, et un autre au moins un ; soit la racine est noire et l'un des deux sommets suivants est noir :



donc l'un des chemins partant de la racine contient un unique sommet noir, et un autre au moins deux.

Lemme 4.14. *La hauteur d'un ARN comportant N sommets est majorée par* $2\log_2(N+1)$.

Preuve. Exercice. □

Exercice 4.15– Écrire un algorithme d'insertion dans un ARN. Complexité ?

Exercice 4.16– Décrypter et commenter l'implantation suivante

```
bool red(link x) { return x? x->red: false; }

void RBinsert(link& h, Item x, bool h_est_D)
```

```
{
  if (!h) { h = new_node(x); return; }
  if (red(h->G) && red(h->D)) {h->red = true; h->G->red = h->D->red = false;}
  if (x < h->item)
  {
    RBinsert(h->G, x, false);
    if (red(h) && red(h->G) && h_est_D) rd(h);
    if (red(h->G) && red(h->G->G)) {rd(h); h->red = false; h->D->red = true;}
  }
  else
  {
    RBinsert(h->D, x, true);
    if (red(h) && red(h->D) && !h_est_D) rg(h);
    if (red(h->D) && red(h->D->D)) {rg(h); h->red = false; h->G->red = true;}
  }
}
```

5. Files de priorité

5.1. **Définition.**

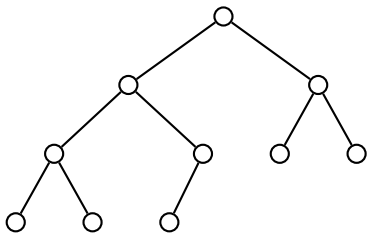
Définition 5.1. Une *file de priorité* (`priority queue`) est une structure de données sur un ensemble ordonné qui supporte les deux opérations : insertion, suppression d'un minimum.

On appelle aussi file de priorité une structure supportant la suppression d'un maximum à la place d'un minimum. Les arbres binaires équilibrés du paragraphe précédent permettent d'implanter des files de priorité dont les deux opérations sont en $O(\log N)$, si la file contient N objets. Nous allons introduire une autre implantation de même complexité asymptotique, mais bien plus simple, avec des constantes cachées plus faibles.

5.2. **Arbres quasi-parfaits.**

Définition 5.2. Un arbre binaire de hauteur n est dit *quasi-parfait* si toutes ses feuilles sont à profondeur n ou $n-1$, et si toutes les feuilles de profondeur n sont regroupées à gauche.

En particulier ce sont des arbres équilibrés, réalisant le minimum de la hauteur pour un nombre donné de sommets. Un sommet au plus aura un unique fils (le père de la feuille la plus à droite) :



La structure particulièrement simple des arbres quasi-parfaits permet de les implanter par un unique tableau T, au lieu de trois pour un arbre binaire quelconque (Contenu, FilsGauche, FilsDroit) : on place la racine en $T[1]$, et pour tout $i \geqslant 1$, les fils gauche et

droit de $T[i]$ sont respectivement en $T[2i]$ et $T[2i+1]$. En particulier, le père de $T[i]$ se trouve en $T[\lfloor i/2 \rfloor]$, et $T[i]$ est une feuille ssi $2i > N$, où N est le nombre de sommets de l'arbre.

Remarque : si N est pair, $T[N/2]$ a un seul fils (gauche) ; d'autre part, $T[0]$ n'est pas utilisé, on peut s'en servir comme sentinelle.

5.3. **Tas.**

Définition 5.3. Un *tas* est un arbre binaire quasi-parfait, dont chaque sommet est inférieur à ses deux fils.

Les inégalités $x \leqslant G(x)$, $x \leqslant D(x)$ sont différentes de celles des ABR, quoique tout aussi strictes, mais la structure de l'arbre est beaucoup plus rigide : le minimum est à la racine, mais une recherche arbitraire est difficile.

Exercice 5.4– Montrer que dans un tas de N sommets, la recherche du maximum est $\Omega(N)$.

Exercice 5.5– Implanter une file de priorité à l'aide d'un tas. [*Suppression : placer la feuille la plus à droite à la place de la racine et l'échanger avec l'un de ses fils tant que nécessaire. Insertion : insérer comme feuille, le plus à droite possible, échanger avec le père tant que nécessaire.*]

5.4. **Une application arithmétique.** On veut déterminer les entiers a, b, c, d dans $[-H, H]$ tels que $a^3+2b^3 = 3c^3+4d^3$. Plus généralement pour des suites (a_i), (b_j), (c_k), (d_l) à H éléments, on veut déterminer (éventuellement, seulement compter) les quadruplets (i, j, k, l) tels que $a_i+b_j = c_k+d_l$. Afin d'éviter de constamment recalculer leurs éléments, on suppose que les 4 suites sont stockées dans des tables, pour un coût $O(H)$ en mémoire.

L'algorithme idiot résout le problème en 4 boucles imbriquées, temps $O(H^4)$, et espace $O(H)$. Un algorithme naïf plus raisonnable contient 3 boucles sur a, b, c et une recherche dans la table des d, préalablement triée, soit $O(H^3 \log H)$ en temps, et toujours $O(H)$ en espace. Plus malin, on exploite la structure du problème et on se rappelle l'idée du trifusion : on tabule séparément $\{a_i + b_j\}$ et $\{c_k + d_l\}$, on trie ces deux listes, et on recherche les collisions entre les tableau triés. Coût : $O(H^2 \log H)$ en temps, mais $O(H^2)$ en espace.

Exercice 5.6– Montrer comment utiliser une file de priorité, par exemple un tas, pour énumérer les $\{a_i + b_j\}$ dans l'ordre *croissant*, toujours en temps $O(H^2 \log H)$ mais en espace $O(H)$ [*On peut supposer que a_i est triée par ordre croissant ; commencer par un tas contenant $\{(a_1 + b_j, 1, j), j \leqslant H\}$*]. En déduire une solution essentiellement optimale au problème de départ.

6. Hachage

6.1. **Principe.** Plutôt que de recourir à des suites de comparaisons et un système de branchement donnant lieu à des structures d'arbres pour implanter un dictionnaire, on peut essayer de généraliser l'idée de l'Algorithme 2.3 (bucketsort) : si les objets sont des petits entiers, on peut maintenir leur liste sous forme d'un tableau dont les indices sont les objets. Pour lever la restriction d'un ensemble des clés réduit aux entiers de 1 à M, on introduit une fonction h, dite de *hachage*, qui associe à chacun des N objets $O_1, \ldots, O_N$

un entier $h(O_i) \in [1, M]$. Optimalement, $h(O_i)$ détermine O_i (h est injective), sinon, il y a *collision* quand $h(O_i) = h(O_j)$ et $i \neq j$.

Exercice 6.1– Quelle est la probabilité qu'une fonction d'un ensemble à N élément dans $[1, M]$, choisie uniformément au hasard, soit injective ? (donne des clés de hachage distinctes). Démontrer le *paradoxe des anniversaires* : dans un groupe de 23 personnes, dont on suppose les jours de naissances équiprobables, il y a plus d'une chance sur deux pour que deux dates anniversaires coïncident. Généraliser.

Le paradoxe des anniversaires indique qu'espérer une fonction de hachage d'usage général et injective est utopique dès que N est de l'ordre de $\sqrt{M}$. Donc tenter d'empêcher les collisions requiert un coût au moins quadratique en mémoire, sans garantie de succès. Nous avons deux problèmes essentiellement indépendants : trouver une fonction h assurant une répartition uniforme des données, et traiter les inévitables collisions.

L'idée la plus simple consiste à maintenir un tableau de M cases (*table de hachage*, `hashtable`), contenant des listes chaînées. Pour une bonne fonction de hachage, M clés et N objets, le nombre moyen d'élément par case devrait être proche de N/M. On s'attend donc à ce que le calcul de $h(O)$ divise la taille du dictionnaire par un facteur M, ce qui serait idéal pour $N \approx M$ (temps constant, espace linéaire !). Si les objets sont totalement ordonnés, on peut maintenir des listes triées sans coût supplémentaire, pour accélérer les insertions. On peut bien sûr utiliser des arbres équilibrés au lieu des listes, mais il vaut mieux essayer de garantir que les listes seront courtes.

Exercice 6.2– Imaginer une table de hachage dynamique, initialement vide, dans laquelle on insère successivement N objets distincts, où N n'est pas connu à l'avance. La table doit assurer par exemple que $N \approx M$ tout au long de l'algorithme, en augmentant M au besoin. Quel est son coût ?

Le problème de déterminer une « bonne » fonction de hachage dépend des données. Il faut qu'elle soit facile à calculer, idéalement $O(1)$, et assure une répartition plus ou moins régulière des $h(O)$. Une approche générique relativement efficace consiste à introduire une fonction f, codant l'objet sous forme d'un grand entier, et poser $h(O) := f(O) \bmod p$, où p est un nombre premier : la table de hachage comporte p cases, numérotées de 0 à $p - 1$. Par exemple, pour une chaîne de caractères, f pourrait donner le code ASCII de la chaîne. En général, la représentation informatique standard des données, ou d'un échantillon représentatif si leur taille est prohibitive, fournit naturellement la fonction f.

Exercice 6.3– Décrire un mécanisme de *tableau associatif*, dans lequel les indices sont des chaînes de caractères arbitraires. Quelle est la complexité des opérations de base ?

Remarque 6.4. La cryptographie fait une utilisation différente du principe du hachage : dans ce cadre, la fonction doit dépendre de l'intégralité des données, être facile à calculer pour qui connaît une information secrète, et apparaître aléatoire pour qui ne la connaît pas. Une telle fonction de hachage peut par exemple être utilisée pour garantir qu'un document n'a pas été altéré. Penser à une liste de 10^6 comptes en banques où l'on veut pouvoir détecter le déplacement d'une virgule, ou à un programme informatique dont on veut vérifier qu'il n'a pas été modifié/contaminé par un virus.

6.2. **Une variation : le tri digital (`radixsort`).** On se donne une collection de N objets dont les clés de tri sont des entiers, suffisamment grands pour que l'application du tri par comptage (Algorithme 2.3) soit utopique, par exemple des entiers sur 64 bits. Inspiré par les méthodes de hachage, on va trier ces clés en n'en considérant que certains morceaux. Les clés sont écrites dans une base B fixée, $C = \sum_{i \geqslant 0} c_i B^i$, et on suppose que toutes ces clés ont moins de $k+1$ chiffres (donc $i \leqslant k$).

On répartit les clés dans B cases suivant la valeur de c_k, puis récursivement chacune des cases avec la même méthode et des clés qui ont un chiffre de moins. Bien sûr, au début de chaque nouvelle phase, on met simplement à jour l'indice du mot considéré, on ne modifie pas toutes les clés ! Si $B = 2$, c'est particulièrement simple :

Exercice 6.5– Décrire un QuickSort binaire (en place), inspiré de ce qui précède, qui partitionne ses éléments suivant leur écriture en base 2.

En général, comme pour le hachage, on peut stocker les clés dans les cases à l'aide de listes chaînées. Mais l'idée du tri par comptage s'applique aussi directement : pour $c = 0, \ldots, B-1$ on compte le nombre de clés dont le k-ième chiffre vaut c, on découpe un tableau auxiliaire en B intervalles, où le c-ième intervalle est juste assez long pour accueillir les clés de k-ième chiffre c, et on distribue.

Exercice 6.6– Écrire l'algorithme complet correspondant. Expliquer en quoi, en termes de « quantité d'information », cet algorithme peut être sous-linéaire. Quel est le cas le pire ? Pour trier des entiers, la base B est en fait un paramètre libre ; discuter les compromis intervenant dans le choix de B.

Le même algorithme fonctionne pratiquement sans changement pour trier des chaînes de caractères sur un alphabet de taille raisonnable. Comme pour tous les algorithmes récursifs, il est avantageux d'arrêter la récursion lorsque la taille des instances devient petite, et finir le travail par un algorithme naïf ou dédié.

Exercice 6.7– Proposer un algorithme analogue lisant les clés dans l'autre sens, en commençant par les chiffres de poids faibles : $c_0, \ldots, c_k$. Discuter son intérêt.

7. Graphes

7.1. Parcours.

Définition 7.1. Un *graphe* non-orienté (S, A) est la donnée d'un ensemble S (sommets) et d'un sous-ensemble A de l'ensemble des paires d'éléments de S (arêtes).

Pour représenter un graphe, on peut supposer que S est constitué des entiers de 1 à N. La correspondance entre les sommets originels et cet indice est un dictionnaire que l'on peut consulter à l'aide d'une table de hachage ou d'un ABR. Il y a essentiellement deux représentations pour A :

- matrice d'incidence : une matrice $N \times N$ de booléens, $A[i, j]$ valant `vrai` si et seulement s'il existe une arête entre les sommets i et j.
- listes d'incidence : à chaque sommet, on associe la liste des sommets adjacents.

Exercice 7.2– Discuter les mérites et les inconvénients des deux représentations, et les types de graphes auxquels elles sont adaptées.

On peut utiliser un tel graphe pour représenter une relation d'équivalence : les classes d'équivalences sont les composantes connexes du graphe.

Exercice 7.3– Décrire un algorithme en $O(\operatorname{card} S + \operatorname{card} A)$ visitant une et une seule fois tous les sommets du graphe en suivant les arêtes. [*Maintenir un tableau des sommets déja visités indexé par S. Pour chaque sommet, visiter ses voisins s'ils ne l'ont pas déjà été. Attention, le graphe n'est pas nécessairement connexe.*]

Exercice 7.4– Modifier l'algorithme précédent pour déterminer si le graphe possède un cycle (s'il existe deux points reliés par deux chemins différents).

Exercice 7.5– Modifier l'algorithme précédent pour déterminer les composantes connexes du graphe.

7.2. **Union-Find.** *Étant donné* le graphe complet, on peut déterminer ses composantes connexes (Exercice 7.5), puis déterminer facilement s'il existe un chemin connectant deux sommets x et y quelconques. On désire modifier cet algorithme pour une situation dynamique où l'on connaît la liste des sommets, mais où l'on peut arbitrairement rajouter des arêtes : quand une nouvelle arête apparaît, il faut fusionner les composantes connexes (Union). Bien sûr, on veut pouvoir déterminer à tout moment la composante connexe d'un point arbitraire (Find).

Une représentation naturelle du problème est un ensemble d'arbres (une forêt) représentant les composantes connexes du graphe : initialement les points sont isolés ; à chaque nouvelle arête $\{i, j\}$ on détermine les arbres correspondant aux deux extrémités i et j, et on les fusionne s'ils sont distincts. On identifie la composante connexe d'un sommet à la racine de l'arbre qui le contient.

Exercice 7.6– Pour représenter une forêt dont tous les sommets sont connus, on utilise la structure de donnée suivante : un tableau `Pere` indexé par les sommets. Le père d'un sommet non-racine est son père dans l'arbre, et on définit le père d'une racine par une sentinelle, par exemple `NULL`. Pour connaître la composante connexe d'un point (Find), il suffit de suivre les pères consécutifs jusqu'à atteindre une racine.

Expliquer comment réaliser l'opération Union avec cette structure de données. Pour un graphe comportant s sommets et au final a arêtes, donner le cas le pire pour les deux opérations Union et Find. Comment améliorer l'équilibrage des arbres ?

Exercice 7.7– Comparer avec l'algorithme associé à la structure de données naïve qui associe à chaque sommet le numéro de sa composante connexe.

Références

[1] A. V. Aho, J. E. Hopcroft, & J. D. Ullman, *The design and analysis of computer algorithms*, Addison-Wesley, 1975, Second printing.

[2] D. E. Knuth, *The art of computer programming. Vol. 1 : Fundamental algorithms*, Addison-Wesley, 1968.

[3] D. E. Knuth, *The art of computer programming. Vol. 3 : Sorting and searching*, Addison-Wesley, 1973.
[4] R. Sedgewick, *Algorithms*, Addison-Wesley, 1983.