Introduction à l'algorithmique probabiliste

Philippe Duchon

Version 4 (25 avril 2006)

Ces notes sont le support du cours “Algorithmique probabiliste” (module IF206) proposé en option aux élèves de deuxième année, en filière Informatique de l'ENSEIRB.

La source principale pour le contenu, est le livre de R. Motwani et P. Raghavan [7]. On trouvera, en fin de ce document, une courte bibliographie (qui ne se veut nullement exhaustive).

Malgré tous les efforts, le sang et la sueur que m'ont coûté la rédaction de ce document, il s'agit, et il s'agira probablement éternellement, d'un travail en cours. Je serai parfaitement reconnaissant au lecteur attentif qui me signalera une des nombreuses corrections à y apporter en vue d'atteindre ce Graal des enseignants : le polycopié sans erreurs.

Table des matières

1 Introduction **1**
1.1 Tri avec **QuickSort** et ses variantes . . . 1
1.1.1 L'algorithme **QuickSort** . . . 1
1.1.2 Une première analyse : complexité face à un pivot idéal . . . 1
1.1.3 Seconde analyse : complexité dans le cas le pire . . . 2
1.1.4 L'algorithme **RandQuickSort** . . . 3
1.2 Le problème de la coupe minimale . . . 5
1.2.1 L'algorithme **RandMinCut** . . . 5
1.2.2 Analyse de l'algorithme . . . 6
1.3 Algorithmes probabilistes . . . 8
1.3.1 Algorithmes Las Vegas . . . 8
1.3.2 Algorithmes Monte Carlo . . . 8
1.3.3 La question de la terminaison . . . 9
1.3.4 Temps d'exécution . . . 10
1.3.5 Quelques classes de complexité probabilistes . . . 10
1.4 Principes et outils . . . 11
1.5 N'est-ce pas de l'analyse probabiliste ? . . . 11
1.6 Exercices . . . 12

2 Modèles d'algorithmes probabilistes **15**
2.1 Définition par la source . . . 15
2.2 Comparaison entre les modèles de sources . . . 16
2.2.1 Source de Bernoulli . . . 17
2.2.2 Source de bits aléatoires . . . 17
2.2.3 Source réelle . . . 20
2.3 Modèle de la bande aléatoire . . . 20
2.4 Alors, quelle définition ? . . . 21
2.5 Réalisations de sources aléatoires . . . 21
2.5.1 Accumulateurs d'entropie . . . 21
2.5.2 Générateurs pseudo-aléatoires . . . 22
2.5.3 Validation de générateurs pseudo-aléatoires . . . 23
2.6 Exercices . . . 24

3 Sélection probabiliste **27**
3.1 Le problème de la sélection . . . 27
3.2 Bornes théoriques de complexité . . . 28

3.3 Algorithme déterministe 28
3.4 Un premier algorithme probabiliste 29
3.5 L'algorithme **RandLazySelect** 29
3.5.1 Probabilité de *rater* $S_{(k)}$ 31
3.5.2 Probabilité que P soit trop grand 31
3.5.3 Commentaires finaux 32

4 Algorithmes issus de la théorie des nombres 35
4.1 Rappels sur l'arithmétique modulaire 35
4.1.1 Opérations arithmétiques modulaires 35
4.1.2 Résidus quadratiques et racines carrées 36
4.1.3 Répartition des nombres premiers 37
4.2 Calcul de racines carrées modulaires 37
4.3 Le test de primalité de Miller-Rabin 40

5 Structures de données probabilistes 45
5.1 Opérations génériques 45
5.2 Principe d'une structure de données probabiliste 45
5.3 Listes à sauts aléatoires 46
5.3.1 Listes à sauts 46
5.3.2 Modèle probabiliste 47
5.4 Treaps 50
5.4.1 Structure de treap 50
5.4.2 Modèle probabiliste de treaps 51
5.4.3 Hauteur des arbres binaires de recherche aléatoires 52
5.5 Tables de hachage 54
5.5.1 Tableaux 55
5.5.2 Tables et fonctions de hachage 55
5.5.3 Familles 2-universelles de fonctions de hachage 55
5.5.4 Construction d'une "petite" famille 2-universelle 56
5.5.5 Extension aux familles k-universelles 58

A Rappels de probabilités 61
A.1 Définitions et notations 61
A.1.1 Notions de base 61
A.1.2 Indépendance 65
A.1.3 Probabilités conditionnelles 66
A.1.4 Série génératrice de probabilités 67
A.1.5 Espaces produits 68
A.2 Quelques exemples de lois de probabilité 69
A.2.1 Loi de Bernoulli 69
A.2.2 Loi géométrique 69
A.2.3 Loi binômiale 70
A.2.4 Loi de Poisson 71
A.2.5 Lois uniformes 71
A.2.6 Loi exponentielle 71
A.3 Inégalités utiles 72

A.3.1 Inégalité de Markov . . . 72
A.3.2 Inégalité de Tchebycheff . . . 72
A.3.3 Inégalité de Chernoff . . . 73

Bibliographie **75**

Liste des Algorithmes **77**

Index **79**

Chapitre 1

Introduction

L'algorithmique probabiliste est l'étude d'algorithmes dont certaines actions sont explicitement aléatoires. Plutôt que de donner immédiatement des définitions et des modèles, nous commençons par deux exemples volontairement très différents : la version "randomisée" de **QuickSort**, et un algorithme de calcul de coupe minimale d'un graphe.

1.1 Tri avec QuickSort et ses variantes

Considérons le problème du *tri* d'un tableau T de valeurs.

Typiquement, la mesure de complexité adoptée pour un algorithme de tri est le nombre de *comparaisons*[1] entre entrées du tableau. C'est celle que nous adopterons ici.

1.1.1 L'algorithme QuickSort

L'algorithme **QuickSort**[2], dû à Hoare, donne une solution récursive au problème du tri de tableau : choisir un élément x du tableau (le *pivot*) ; puis, en comparant chaque autre élément au pivot, former les tableaux $T_<$ et $T_>$, respectivement formés des éléments inférieurs et supérieurs à x. Trier, au moyen de **QuickSort**, les tableaux $T_<$ et $T_>$, et renvoyer le tableau T' formé des éléments triés de $T_<$, puis x, puis des éléments triés de $T_>$.

1.1.2 Une première analyse : complexité face à un pivot idéal

Supposons, dans un premier temps, que le choix du pivot puisse être effectué de telle sorte que, à chaque appel récursif, les tableaux $T_>$ et $T_<$ soient de même taille, à au plus un élément près pour des raisons évidentes de parité (en d'autre termes, chaque pivot est un élément médian du tableau courant). Si le tableau compte n entrées, les deux sous-tableaux ont donc pour tailles respectives $\lfloor (n-1)/2 \rfloor$ et $\lceil (n-1)/2 \rceil$. Avec ces suppositions, le coût C_n du tri d'un tableau de taille n, vérifie alors la récurrence suivante :

[1] Pour se convaincre de la pertinence de ce modèle, on pourra imaginer que le tableau contient des références à des structures plus conséquentes ; la comparaison de deux structures sera alors généralement au moins aussi coûteuse que, par exemple, l'échange de deux références dans le tableau

[2] Cet algorithme, légèrement modifié, est celui qui est utilisé par l'utilitaire `sort` sous Unix.

Entrée : un ensemble S de n clés
Sortie : les éléments de S dans l'ordre croissant

1. Si $n \leq 1$, retourner S.
2. Choisir un élément x de S.
3. En comparant chaque autre élément de S à x, former l'ensemble S_1 des éléments de S inférieurs à x, et l'ensemble S_2 des éléments de S supérieurs à x.
4. Trier S_1 et S_2 en utilisant **QuickSort** ; retourner les éléments de S_1 triés, puis x, puis les éléments de S_2 triés.

ALGO. **1.1:** QuickSort

$$C_0 = 0 \quad (1.1)$$
$$C_1 = 0 \quad (1.2)$$
$$C_{n+1} = n + C_{\lfloor \frac{n}{2} \rfloor} + C_{\lceil \frac{n}{2} \rceil} \quad (1.3)$$

Il est facile de voir que la suite solution de cette récurrence est croissante ; nous nous concentrons donc sur la sous-suite $C'_k = C_{2^k-1}$, qui a l'avantage de nous éviter d'avoir à manipuler des parties entières : la récurrence 1.3 devient

$$C'_k = 2^k - 2 + 2C'_{k-1} \quad (1.4)$$

dont il est facile de vérifier[3] que la solution (pour les conditions initiales $C'_0 = C'_1 = 0$) est $C'_k = (k-2)2^k + 2$.

On a donc $C_{2^k-1} = (k-2)2^k + 2$; si l'on ajoute à cela la croissance de la suite (C_n), on en déduit à la fois une borne inférieure et une borne supérieure :

$$C_n = \Theta(n \log n). \quad (1.5)$$

De plus, les constantes "cachées" par l'utilisation de la notation Θ sont données par le calcul précédent : si n est compris entre $2^k - 1$ et $2^{k+1} - 1$ (donc, $k = \log_2(n+1)$ à 1 près), on a

$$(k-2)2^k + 2 \leq C_{2^k-1} \leq C_n \leq C_{2^{k+1}-1} = (k-1)2^{k+1} + 2,$$

ce qui fournit un encadrement de

$$n \log_2(n) + o(n \log n) \leq C_n \leq 2n \log_2(n) + o(n \log n).$$

1.1.3 Seconde analyse : complexité dans le cas le pire

L'analyse précédente utilisait une hypothèse particulièrement irréaliste : le pivot était toujours supposé être un élément *médian* du tableau à trier. Or, le problème de trouver un élément médian ne peut être résolu en temps constant dans un ensemble non trié. Il existe pour

[3] Pour *trouver* cette expression, poser $d_k = C'_k.2^{-k}$ et résoudre directement la récurrence que l'on obtient pour d_k.

cela des algorithmes déterministes en temps linéaire (voir par exemple l'algorithme décrit au Chapitre 3, page 28), et sélectionner le pivot en temps linéaire résulterait bien en un algorithme de tri de complexité asymptotique $\Theta(n \log n)$, mais quoi qu'il en soit, les constantes en seraient *a priori* dégradées.

Il est à noter toutefois que l'hypothèse faite (le pivot est toujours un élément médian) n'est pas nécessaire pour obtenir un temps d'exécution en $\Theta(n \log n)$; il suffirait pour cela de savoir qu'un tableau de taille k sera toujours coupé en deux tableaux de tailles comprises entre $k/4$ et $3k/4$ (voir Exercice 1.3), ou même entre λk et $(1-\lambda)k$ pour une constante[4] $0 < \lambda < 1/2$.

Toutefois, la complexité *dans le cas le pire* de **QuickSort** est quadratique. En effet, supposons, pour fixer les idées, que le pivot soit toujours le premier élément de S (dont on suppose qu'il est présenté sous une forme ordonnée). Si le tableau initial est, par exemple, déjà dans l'ordre croissant ou décroissant, à chaque étape de l'algorithme l'un des tableaux S_1 ou S_2 sera vide, et il est facile de voir que dans ce cas, chaque élément de S sera finalement comparé à chaque autre, pour un nombre total de comparaisons de $n(n-1)/2 = \Theta(n^2)$. (Le fait que, pour cette stratégie relativement naturelle de choix du pivot, les tableaux déjà triés soient les pires, est particulièrement gênant : il n'est pas rare, en pratique, d'avoir à trier des éléments qui sont déja "presque" triés.)

1.1.4 L'algorithme RandQuickSort

Une "solution" au dilemme présenté ci-dessus, et qui permet en quelque sorte de "racheter" l'algorithme de Hoare, est l'Algorithme 1.2.

Entrée : un ensemble S de n valeurs comparables
Sortie : les éléments de S, triés en ordre croissant

1. Si $n \leq 1$, retourner S.
2. Choisir un élément x de S, **au hasard** (chaque élément de S ayant la même probabilité d'être sélectionné).
3. En comparant chaque autre élément de S à x, former l'ensemble S_1 des éléments de S inférieurs à x, et l'ensemble S_2 des éléments de S supérieurs à x.
4. Trier S_1 et S_2 en utilisant **RandQuickSort** ; retourner les éléments de S_1 triés, puis x, puis les éléments de S_2 triés.

ALGO. 1.2: L'algorithme **RandQuickSort**

Contrairement à la situation classique d'un algorithme déterministe, la mesure de complexité (nombre de comparaisons) de l'exécution de notre algorithme sur une instance donnée (un tableau), n'est plus fixe ; il s'agit d'une *variable aléatoire*, qui ici dépend non seulement du tableau S, mais aussi des choix aléatoires faits, lesquels sont supposés non nécessairement identiques d'une exécution à l'autre.

L'analyse dans le cas le pire, qui donne une complexité de $\binom{n}{2}$ au plus, reste valable (l'algorithme évite toujours de comparer deux éléments plus d'une fois ; et il est possible, bien que peu probable, que **RandQuickSort** sélectionne à chaque étape un pivot minimal

[4]Les constantes multiplicatives c_1 et c_2 apparaissant dans les bornes de la forme $c_1 n \log n \leq C_n \leq c_2 n \log n$, implicites dans la notation Θ, dépendent alors de λ.

ou maximal, ce qui le conduira effectivement à comparer chaque paire) ; toutefois, et c'est un point critique, l'analyse qui suit est valable *quel que soit le tableau S initial.*

Soient $s_1 \leq s_2 \leq \cdots \leq s_n$ les n éléments de S. Pour $1 \leq i < j \leq n$, nous noterons X_{ij} la variable aléatoire qui vaut 1 si s_i et s_j sont comparés au cours de l'exécution de **RandQuickSort**, et 0 sinon ; une telle variable aléatoire est appelée *indicatrice de l'évènement* (s_i *est comparé à* s_j). Le nombre de comparaisons effectuées par l'algorithme est alors

$$X = \sum_{1 \leq i < j \leq n} X_{ij}.$$

Chaque X_{ij} ne pouvant prendre comme valeurs que 0 ou 1, son espérance est simplement la probabilité p_{ij} que s_i et s_j soient comparés.

Bien que les différentes variables aléatoires X_{ij} ne soient pas *a priori* indépendantes, la *linéarité de l'espérance* s'applique toujours, et permet d'écrire

$$\mathbb{E}(X) = \sum_{1 \leq i < j \leq n} \mathbb{E}\left(X_{ij}\right) = \sum_{1 \leq i < j \leq n} p_{ij}. \tag{1.6}$$

Il nous reste donc seulement à évaluer p_{ij}, et à calculer la somme de ces probabilités.

Pour évaluer p_{ij}, remarquons que s_i et s_j seront comparés entre eux, *si et seulement si* l'un d'entre eux est pris comme pivot avant qu'aucun des éléments $s_{i+1}, \ldots, s_{j-1}$ le soit (en effet, dès qu'un tel élément est pris comme pivot, s_i et s_j se retrouvent dans deux tableaux différents et ne seront donc plus jamais comparés). Or, lors du déroulement de l'algorithme, chaque fois qu'un sous-tableau contenant s_i et s_j est trié, il contient également chacun des éléments intermédiaires, et aucun parmi les $j - i + 1$ éléments $s_i, \ldots, s_j$ n'a plus de chances d'être sélectionné comme pivot qu'un autre. Par conséquent, la probabilité que s_i ou s_j le soit avant chacun des $j - i - 1$ autres est *exactement*

$$p_{ij} = \frac{2}{j - i + 1}. \tag{1.7}$$

Une fois cette probabilité déterminée, le reste n'est "que" calcul. En combinant 1.6 et 1.7, nous obtenons

$$\begin{aligned}
\mathbb{E}(X) &= \sum_{1 \leq i < j \leq n} \frac{2}{j - i + 1} \\
&= 2 \sum_{i=1}^{n-1} \sum_{k=2}^{n-i+1} \frac{1}{k} \\
&\leq 2 \sum_{i=1}^{n} \sum_{k=1}^{n} \frac{1}{k} \\
&= 2n \sum_{k=1}^{n} \frac{1}{k}.
\end{aligned}$$

On reconnaît dans cette dernière expression le *n-ème nombre harmonique* $H_n = \sum_{k=1}^{n} 1/k = \ln(n) + O(1)$. Nous avons donc obtenu le

Théorème 1.1 *Pour tout tableau S, de taille n, l'espérance du nombre de comparaisons effectuées par* ***RandQuickSort*** *sur l'instance S est au plus de $2nH_n$.*

Un calcul exact donnerait $\mathbb{E}(X) = 2(n+1)H_n - 4n = 2n\ln(n) + O(n)$, ce qui montre que la majoration effectuée ici n'est pas trop généreuse.

Il existe de nombreux résultats plus précis sur la *loi de distribution* de la complexité de **RandQuickSort** et de ses variantes. On peut voir que cette loi ne dépend aucunement du tableau initial S, mais uniquement de sa taille.

Un calcul similaire à celui que nous venons de faire pour l'espérance, mais quelque peu plus technique, et concernant la *variance* du nombre de comparaisons, aboutirait au résultat que

$$\mathbf{Var}(X_n) = O(n^2).$$

On en déduit que l'écart-type est en $O(n)$, ce qui est asymptotiquement négligeable devant l'espérance. Selon un schéma de raisonnement classique (mettant en œuvre l'inégalité de Tchebycheff), il en résulte que la suite (de variables aléatoires) $X_n/(2n\ln(n))$ converge *en probabilités* vers 1. En particulier,

Théorème 1.2 *Pour tout $\epsilon > 0$, on a*

$$\mathbb{P}(X_n > (2+\epsilon)n\ln n \text{ ou } X_n < (2-\epsilon)n\ln n) = O\left(\ln(n)^{-2}\right).$$

En d'autres termes, *il est très peu probable que, sur de grands tableaux,* ***RandQuickSort*** *effectue sensiblement plus (ou sensiblement moins) de $2n\ln n$ comparaisons.* Ce résultat est particulièrement intéressant si l'on tient compte du fait que $2\ln(n) \approx 1.39\log_2(n)$: presque toujours, **RandQuickSort** n'effectuera que 39% de comparaisons en plus de la borne inférieure fournie par le nombre de permutations possibles des n clés.

1.2 Le problème de la coupe minimale

Considérons un graphe $G = (V, E)$, connexe, non orienté, à arêtes multiples mais sans boucles (il peut exister entre deux sommets quelconques un nombre arbitraire, nul ou non, d'arêtes ; mais aucune arête ne peut avoir ses deux extrêmités confondues en un seul sommet). Une *coupe* de G est l'ensemble $\mathcal{C}$ des arêtes de G qui ont une extrêmité dans chacun de deux ensembles de sommets X et Y, ces deux ensembles formant une partition de V. Une *coupe minimale* est une coupe de taille (nombre d'arêtes) minimale.

Le problème de la coupe minimale consiste, étant donné G, à exhiber une coupe minimale de G. Celui-ci peut être résolu par un algorithme déterministe classique, à base de calculs de flots ; nous présentons toutefois un algorithme probabiliste extrêmement simple, même s'il n'est pas sans défauts et demande un peu de travail pour être compétitif ; l'intérêt réside essentiellement dans le type de résultat obtenu, et dans la méthode d'analyse.

1.2.1 L'algorithme RandMinCut

Etant donné un graphe G et une de ses arêtes e, la *contraction* de e est l'opération consistant à fusionner les deux sommets extrêmités de e et à retirer l'arête e ; dans notre cas, nous considèrerons que l'on retire également toutes les boucles formées lors de la fusion (c'est-à-dire toutes les autres arêtes de G qui pourraient joindre les mêmes sommets que e). Le graphe ainsi obtenu sera noté $G_{/e}$.

Remarquons qu'une autre façon de voir $G_{/e}$ est comme un graphe dont les sommets forment une partition de l'ensemble des sommets de départ, la contraction d'une arête correspondant alors à remplacer deux "sommets" par leur union.

Notre algorithme de calcul d'une coupe de G est présenté Algorithme 1.3.

Entrée : un graphe G, à arêtes multiples, sans boucles.
Sortie : une coupe de G.

1. Tant que G a plus de deux sommets, répéter :
   (a) Choisir une arête de G au hasard (chaque arête ayant la même probabilité d'être choisie) ;
   (b) Remplacer G par $G_{/e}$.
2. Retourner toutes les arêtes de G.

ALGO. 1.3: RandMinCut

1.2.2 Analyse de l'algorithme

Il est facile de voir que l'ensemble retourné par cet algorithme est toujours une coupe du graphe de départ. Toutefois, il n'est pas assuré que cette coupe soit une coupe minimale ; c'est même, en toute généralité, peu probable, au moins pour certains graphes.

Nous allons néanmoins démontrer la proposition suivante :

Proposition 1.3 *Pour tout graphe à n sommets, la probabilité que **RandMinCut** retourne une coupe minimale de G est supérieure ou égale à $2/n^2$.*

Preuve: Considérons une coupe minimale $\mathcal{C}$ de G, de cardinalité k ; nous allons montrer que $\mathcal{C}$ a probabilité au moins $2/n^2$ d'être retournée par **RandMinCut**.

Une première remarque est que G a au moins $kn/2$ arêtes : en effet, si une coupe de cardinalité k est minimale, cela implique que chaque coupe de la forme $(\{u\}, E - \{u\})$ est de taille au moins k, ce qui se traduit par le fait que chaque sommet est de degré au moins k.

Une seconde remarque, peut-être moins évidente, est que $\mathcal{C}$ sera retournée par **RandMinCut**, si et seulement si aucune des arêtes de $\mathcal{C}$ n'est sélectionnée pour être contractée. L'une des implications sous-jacentes est triviale (si $\mathcal{C}$ est retournée, c'est qu'aucune de ses arêtes n'a été contractée) ; pour nous convaincre de la réciproque, faisons l'hypothèse qu'aucune arête de $\mathcal{C}$ n'a été contractée.

Remarquons d'abord que les seules arêtes qui ne sont pas retournées sans toutefois avoir été contractées, sont celles qui sont transformées en boucles par la contraction d'autres arêtes. Or, si une des arêtes de $\mathcal{C}$ subit un tel sort, cela implique que ses deux extrêmités ont été fusionnées, ce qui n'est possible que si une autre arête de $\mathcal{C}$ a été contractée. Par conséquent, si aucune arête de $\mathcal{C}$ n'est contractée, toutes les arêtes de $\mathcal{C}$ sont finalement retournées. Il suffit maintenant de vérifier qu'aucune arête n'appartenant pas à $\mathcal{C}$ ne peut être retournée, ce qui est vrai car l'algorithme ne se termine que lorsqu'il ne reste que deux sommets : ces deux sommets correspondent donc exactement aux deux parties séparées par $\mathcal{C}$, et toute arête qui n'appartient pas à $\mathcal{C}$ joint deux sommets de la même partie, et aura donc été éliminée, soit par contraction, soit comme boucle.

Concentrons-nous maintenant sur la *probabilité* qu'aucune arête de $\mathcal{C}$ ne soit contractée au cours des $n-2$ étapes. Soit $\mathcal{A}_i$ l'évènement "l'arête contractée à l'étape i n'est pas une arête de $\mathcal{C}$" ; nous devons minorer la probabilité de l'évènement $\mathcal{A} = \cap_{i=1}^{n-2} \mathcal{A}_i$.

Puisque G a au moins $kn/2$ arêtes, et que seules k sont dans $\mathcal{C}$, la probabilité de $\mathcal{A}_1$ est au moins de $1 - 2/n$. En supposant que $\mathcal{A}_1$ se produit, il restera au moins $k(n-1)/2$ arêtes (le graphe obtenu n'a plus que $n-1$ sommets, et sa taille de coupe minimale est toujours de k), et le même raisonnement donne

$$\mathbb{P}(\mathcal{A}_2|\mathcal{A}_1) \geq 1 - \frac{2}{n-1}.$$

De manière générale, la probabilité que $\mathcal{A}_i$ se produise *sachant que* $\mathcal{A}_1, \ldots, \mathcal{A}_{i-1}$ *se produisent* est la probabilité qu'un choix aléatoire d'une arête parmi au moins $k(n-i+1)/2$ (graphe à $n-i+1$ sommets, de taille de coupe minimale k) en évite k :

$$\mathbb{P}\left(\mathcal{A}_i | \cap_{j=1}^{i-1} \mathcal{A}_j\right) \geq 1 - \frac{2}{n-i+1}.$$

En appliquant de manière itérée la définition d'une probabilité conditionnelle, il vient

$$\begin{aligned}
\mathbb{P}(\mathcal{A}) &= \mathbb{P}\left(\cap_{i=1}^{n-2} \mathcal{A}_i\right) \\
&= \mathbb{P}(\mathcal{A}_1) \prod_{i=2}^{n-2} \mathbb{P}\left(\mathcal{A}_i | \cap_{j=1}^{i-1} \mathcal{A}_j\right) \\
&\geq \prod_{i=1}^{n-2} \left(1 - \frac{2}{n-i+1}\right) \\
&= \prod_{i=1}^{n-2} \frac{n-i-1}{n-i+1} \\
&= \frac{2}{n(n-1)}
\end{aligned}$$

ce qui prouve la proposition. □

La proposition 1.3 semble à première vue catastrophique pour l'utilisation de notre algorithme en ce qui concerne la recherche d'une coupe minimale : un algorithme qui n'aurait qu'une probabilité aussi faible de succès n'est pas acceptable. Il est toutefois possible, de par son caractère *non déterministe*, de l'exécuter *plusieurs fois* sur le même graphe, et de ne conserver que la plus petite coupe obtenue, ce qui permet de rendre aussi faible que l'on souhaite la probabilité de ne pas obtenir une coupe minimale (nous utilisons ici le fait que, bien que le résultat d'une exécution de **RandMinCut** ne soit pas forcément une coupe minimale, c'est en revanche toujours une coupe ; et que l'identification du meilleur candidat parmi les résultats de plusieurs exécutions indépendantes est donc triviale).

Proposition 1.4 *Pour tout graphe à n sommets et tout réel $0 < \epsilon < 1$, la probabilité que $n^2 \ln(1/\epsilon)/2$ répétitions indépendantes de **RandMinCut** donnent une coupe minimale, est supérieure ou égale à $1 - \epsilon$.*

Preuve: D'après la Proposition 1.3, la probabilité qu'une exécution de l'algorithme ne donne pas une coupe minimale est au plus de $1-2/n^2$. Par conséquent, la probabilité que k répétitions indépendantes échouent toutes à trouver une coupe minimale, est au plus de $\left(1 - 2/n^2\right)^k$.

Pour tout réel $x > 1$, on a $(1 - 1/x)^x < 1/e$; en appliquant cette inégalité à $x = 2/n^2$, on obtient $\left(1 - 2/n^2\right)^{n^2/2} < 1/e$, et donc

$$\left(1 - \frac{2}{n^2}\right)^{n^2 \ln(1/\epsilon)/2} < e^{-\ln(1/\epsilon)} = \epsilon$$

□

1.3 Algorithmes probabilistes

Les algorithmes **RandQuickSort** et **RandMinCut** sont deux exemples, volontairement très différents, de ce que nous appellerons algorithmes probabilistes ; leurs propriétés sont d'ailleurs différentes et, à première vue, incompatibles :

- l'algorithme **RandQuickSort** donne de manière certaine un résultat exact, au bout d'un temps qui est lui-même une variable aléatoire ;
- l'algorithme **RandMinCut** donne un résultat qui a une probabilité non nulle d'être incorrect (en un temps qui n'est pas, *stricto sensu*, déterministe, mais cette caractéristique n'est pas essentielle ici).

En revanche, une caractéristique commune de nos *analyses* de ces deux algorithmes est la suivante : les bornes supérieures que nous avons calculées (sur l'espérance de la complexité, dans le cas de **RandQuickSort** ; sur la probabilité de donner une réponse incorrecte, dans le cas de **RandMinCut**) sont valables *quelle que soit* l'instance considérée du problème (le tableau à trier, ou le graphe G dont on recherche une coupe minimale). En ce sens, nos analyses, si elles sont essentiellement *probabilistes*, sont bien des analyses *dans le cas le pire* ; contrairement à ce qui se fait classiquement lors d'analyses *en moyenne*, nous n'avons fait aucune hypothèse sur la "probabilité" que l'instance considérée soit telle ou telle.

Nous appellerons *algorithme probabiliste*, un algorithme dans lequel il peut être fait usage d'instructions de la forme "tirer un entier au hasard, uniformément entre 1 et n" - le modèle exact pour ces instructions sera précisé ultérieurement.

1.3.1 Algorithmes Las Vegas

Un algorithme probabiliste dont le résultat est toujours exact, est appelé *algorithme Las Vegas* ; la principale grandeur de nature probabiliste attachée à un tel algorithme, est typiquement son *temps d'exécution*. Un exemple est l'algorithme **RandQuickSort** présenté Section 1.1.4.

Généralement, on ne demande pas qu'un algorithme Las Vegas ait une complexité en temps qui soit bornée *a priori* ; on se contentera le plus souvent d'une garantie sur l'*espérance* de cette complexité en temps. On dira donc que l'on a un algorithme Las Vegas en temps moyen polynomial s'il existe une fonction polynôme f telle que, pour toute entrée x de taille n, le temps moyen de l'algorithme sur l'entrée x est inférieur ou égal à $f(n)$.

La plupart des algorithmes que nous étudierons seront de type Las Vegas.

1.3.2 Algorithmes Monte Carlo

Un algorithme probabiliste qui peut donner un résultat erroné, est appelé *algorithme Monte Carlo* ; il est alors essentiel de *borner* la probabilité d'erreur. On dira donc d'un algorithme qu'il

est un *algorithme Monte Carlo d'erreur* λ, si, pour toute entrée, la probabilité qu'il retourne un résultat erroné est au plus de λ. On considère normalement, sauf mention explicite du contraire (voir Exercice 1.6) qu'un algorithme Monte Carlo doit avoir une probabilité d'erreur strictement inférieure à 1.

Dans tous les cas, il est important de noter que la majoration de la probabilité d'erreur, doit être valable pour **toutes les instances** du problème considéré.

Dans le cas des problèmes de décision, où la réponse correcte est `OUI` ou `NON`, on distinguera par ailleurs deux sous-classes d'algorithmes Monte Carlo :

- un algorithme est un *algorithme Monte Carlo à erreur unilatérale* (en Anglais : *one-sided error Monte Carlo algorithm*) si, pour toute instance pour laquelle la réponse est `NON` (instance négative), l'algorithme répond `NON` avec probabilité 1 (en d'autres termes, il ne peut se tromper que dans un seul sens : en répondant `NON` à une instance positive - pour laquelle la bonne réponse est `OUI`) ;
- un algorithme est un *algorithme Monte Carlo à erreur bilatérale* (ou *two-sided error Monte Carlo algorithm*) si, pour au moins une instance positive et au moins une instance négative, la probabilité qu'il réponde `OUI` est strictement comprise entre 0 et 1.

Il est fréquent de demander, pour un algorithme Monte Carlo, que la terminaison ait lieu en temps borné de manière déterministe – c'est-à-dire qu'il existe une fonction f telle que, pour toute entrée de taille n, l'algorithme se termine forcément en temps au plus $f(n)$. Si f est majorée par une fonction polynôme, on parlera alors d'algorithme Monte Carlo en temps polynomial.

L'algorithme **RandMinCut**, à condition d'utiliser l'astuce de répétition $\Theta(n^2)$ fois, est un exemple d'algorithme Monte Carlo ; par ailleurs, si on le transforme de manière naturelle en algorithme de décision (en ajoutant à l'entrée un entier k et en demandant de décider si toute coupe du graphe a un poids strictement supérieur à k), il devient un algorithme à erreur unilatérale. La probabilité d'erreur peut être rendue aussi petite que l'on veut en changeant le nombre de répétitions.

1.3.3 La question de la terminaison

Pour un algorithme déterministe, on exige normalement que le calcul se termine pour toute instance. Dans le cas d'un algorithme probabiliste, cette terminaison est *a priori* un évènement qui dépend à la fois de l'instance et des choix aléatoires faits au cours de l'exécution.

On pourrait envisager différents jeux de conditions sur la terminaison d'un algorithme probabiliste. Par ordre croissant d'exigence :

Probabilité positive Pour toute instance, la probabilité que l'algorithme se termine est strictement positive.

Probabilité λ $(0 < \lambda < 1)$ Pour toute instance, la probabilité que l'algorithme se termine est au moins égale à λ.

Presque sûre Pour toute instance, la probabilité que l'algorithme se termine est 1.

Sûre Pour toute instance, l'algorithme se termine quelque soient les tirages aléatoires.

Dans tout ce qui suit, sauf mention explicite du contraire, nos algorithmes probabilistes sont supposés se terminer de façon presque sûre.

1.3.4 Temps d'exécution

Lorsque l'on étudie un algorithme de type Monte Carlo (qui, donc, a "le droit" de donner un résultat erroné), on considère qu'il doit y avoir, pour chaque *taille d'instance* n, une borne **absolue** $f(n)$ sur le temps d'exécution : *aucune exécution de l'algorithme sur une instance de taille n, quels que soient les résultats des tirages aléatoires effectués, ne peut prendre un temps plus grand que $f(n)$.* La raison en est assez intuitive : si, au bout du temps autorisé, la solution n'a toujours pas été trouvée, il est toujours possible de se "rabattre" sur une solution par defaut qui peut être incorrecte ; du moment que cette limite de temps n'a qu'une faible probabilité d'être atteinte, cette modification ne remettra pas en cause le statut "algorithme Monte Carlo". Cela s'applique même dans le cas où on cherche à obtenir un algorithme Monte Carlo à erreur unilatérale pour un problème de décision : lorsque la limite de temps est atteinte, l'algorithme peut toujours répondre `NON` (en d'autres termes, l'algorithme ne répond pas exactement à la question *est-il vrai que...*, mais à la question *êtes-vous sûr que...*).

Ainsi, un algorithme Monte Carlo *à temps polynomial* sera un algorithme Monte Carlo pour lequel il existe une fonction polynôme f, de telle sorte que, pour toute instance x, l'algorithme répond en un temps qui ne peut pas excéder $f(|x|)$.

Dans le cas des algorithmes Las Vegas, la situation est un peu plus complexe, dans la mesure où la réponse, lorsqu'elle est donnée, doit impérativement être correcte. Demander une borne absolue sur le temps d'exécution est parfois trop contraignant, et rend impossibles certaines réductions. On se rabat donc sur une notion de *temps moyen* : on dira qu'un algorithme Las Vegas a un *temps moyen au plus* $f(n)$, pour une fonction f donnée, si, *pour toute instance x de taille n, l'espérance du temps d'exécution de l'algorithme sur l'instance x est au plus de $f(n)$*. Encore une fois, il s'agit donc bien d'un cas le pire (on considère toutes les instances de même taille, et la majoration doit être valable pour toutes à la fois, avec la même limite $f(n)$), mais il se peut que, par suite des résultats des tirages aléatoires effectués, certaines exécutions dépassent la limite en question – de telles exécutions sont simplement peu probables (typiquement, il faut garder à l'esprit les inégalités comme l'inégalité de Markov : la probabilité que le temps d'exécution soit supérieur à $2f(n)$ ou à $3f(n)$, ne peut pas dépasser $1/2$ ou $1/3$, respectivement).

Ainsi, un algorithme Las Vegas *à temps moyen polynomial* est un algorithme pour lequel il existe un fonction polynôme f, telle que, pour toute instance x, l'algorithme calcule une réponse correcte en un temps (*a priori* aléatoire) dont l'**espérance** est d'au plus $f(|x|)$.

1.3.5 Quelques classes de complexité probabilistes

Nous ne traiterons pas ici de la théorie des classes de complexité probabiliste. Mentionnons tout de même quelques exemples, censés capturer une partie de la différence (supposée) entre $\mathcal{P}$ et $\mathcal{NP}$.

Un langage L appartient à la classe $\mathcal{RP}$ (*randomized polynomial*) s'il existe un algorithme de type Monte Carlo, à temps polynomial, à erreur unilatérale, de probabilité d'erreur au plus $1/2$, pour décider de l'appartenance d'un mot x à L (en d'autres termes, l'algorithme doit se terminer en temps au plus $f(|x|)$, rejeter tout mot ne faisant pas partie de L, et accepter tout mot de L avec probabilité au moins $1/2$). La probabilité $1/2$ n'a rien de magique : n'importe quelle valeur (fixe !) strictement comprise entre 0 et 1 pourrait être utilisée, et la classe $\mathcal{RP}$ définie serait la même.

Un langage L appartient à la classe co-$\mathcal{RP}$, si son complémentaire (pour un alphabet

donné) appartient à la classe $\mathcal{RP}$. En d'autres termes, L est un langage co-$\mathcal{RP}$ si et seulement s'il existe un algorithme probabiliste en temps polynomial, qui accepte tout mot de L de manière certaine et rejette tout mot n'appartenant pas à L avec probabilité au moins $1/2$.

Un langage L appartient à la classe $\mathcal{ZPP}$ (*zero-probability polynomial*) s'il existe un algorithme Las Vegas, à temps moyen polynomial, qui décide de l'appartenance à L.

Il est clair[5] que l'on a $\mathcal{P} \subset \mathcal{RP} \subset \mathcal{NP}$, ainsi que $\mathcal{P} \subset \mathcal{ZPP} \subset \mathcal{NP}$. Il est un peu moins évident (mais c'est un bon exercice que d'en chercher une preuve) que l'on a $\mathcal{ZPP} = \mathcal{RP} \cap \text{co-}\mathcal{RP}$.

Enfin, notons qu'il existe également une définition de la classe $\mathcal{NP}$ elle-même au moyen d'algorithmes probabilistes, et parfaitement équivalente aux définitions classiques.

1.4 Principes et outils

Les deux problèmes principaux qui se posent lorsque l'on désire utiliser un algorithme probabiliste, sont grosso modo les suivants.

- La conception : il arrive fréquemment que les algorithmes probabilistes proposés aient un fonctionnement extrêmement simple – beaucoup plus que les algorithmes déterministes équivalents. De nombreux obstacles, qui ont tendance à augmenter la complexité dans le cas le pire d'algorithmes déterministes, peuvent être contournés en effectuant des choix aléatoires. Une connaissance de certains résultats plus ou moins élémentaires de théorie des probabilités, permet souvent de faire ici des choix judicieux.
- L'analyse : une fois choisi un algorithme, les grandeurs probabilistes qui lui sont associées (complexité en temps et/ou en espace, probabilité de succès) doivent être étudiées. Cette partie, indispensable, peut être assez complexe ; toutefois, même des outils proabilistes élémentaires permettent souvent de mener assez loin cette analyse.

Pour ce qui est de la conception d'algorithmes probabilistes, l'exemple de **RandQuickSort** est typique : l'aléatoire n'est, en quelque sorte, utilisé que pour assurer que, *sur toute instance*, la complexité moyenne est égale à la complexité moyenne (sous hypothèse uniforme) d'un algorithme déterministe correspondant. Cette technique porte le nom de *randomisation*[6]. Une technique en quelque sorte inverse, appelée *dérandomisation*, permet parfois de construire un algorithme déterministe à partir d'un algorithme qu'il est plus facile de décrire comme un algorithme probabiliste.

1.5 N'est-ce pas de l'analyse probabiliste ?

Il est facile de confondre l'analyse d'un algorithme probabiliste et l'analyse probabiliste d'un algorithme déterministe – et, dans certains cas, les outils et arguments mathématiques mis en œuvre peuvent être exactement les mêmes.

Lors de l'analyse probabiliste (souvent, il s'agit d'analyse en moyenne) d'un algorithme déterministe, on est amené à faire *a priori* une hypothèse sur les *instances* soumises à l'algorithme étudié, sous la forme d'un choix de *distribution de probabilités* sur l'ensemble des instances – souvent, une hypothèse d'*uniformité* parmi les instances d'une certaine taille.

[5] En tout cas, il devrait être clair...

[6] Il ne s'agit pas vraiment d'un anglicisme ; le mot anglais *random* a la même racine que le désuet, mais français *à randon*, signifiant au hasard, et que l'on retrouve dans *randonnée*. Merci à mon collègue Jean-Guy Penaud qui a attiré mon attention sur ce point.

Dans le cas, par exemple, de **QuickSort**, cela revient à supposer que les tableaux à trier sont "parfaitement mélangés" au sens où la permutation à leur appliquer pour les trier, est une permutation aléatoire uniforme. Toute affirmation de la forme "la complexité *moyenne* de **QuickSort** est $\Theta(n \log n)$" repose nécessairement sur une telle hypothèse (pas forcément l'hypothèse uniforme ; l'affirmation est assez robuste, en ce sens qu'elle reste vraie si la loi de l'ordre des données, sans être complètement uniforme, est seulement "pas trop déséquilibrée").

En revanche, l'analyse (en moyenne ou en distribution) d'un algorithme probabiliste n'a typiquement pas besoin d'hypothèses sur la nature des instances qui sont soumises à l'algorithme – cette analyse peut parfaitement être valable pour *toute* instance, même choisie par un adversaire dans le seul but de faire "trébucher" l'algorithme. Les affirmations concernant la complexité moyenne d'un algorithme comme **RandQuickSort** ne reposent que sur l'hypothèse que l'on est effectivement capable de choisir un pivot au hasard, indépendemment d'un choix à l'autre.

1.6 Exercices

Exercice 1.1 (Borne inférieure pour le tri) *Soit A un algorithme de tri n'utilisant, comme outil de tri, que des comparaisons entre deux éléments, et dont on suppose qu'il trie tout tableau de taille n en au plus D_n comparaisons. Montrer que l'on a nécessairement $D_n = \Omega(n \log n)$.*

Indication : *on pourra utiliser la formule de Stirling :*

$$\ln(n!) = n \ln n - n + \frac{1}{2} \ln n + O(1) \tag{1.8}$$

Exercice 1.2 (QuickSort idéal) *Dans cet exercice, C_n représente la complexité hypothétique de l'algorithme **QuickSort** dans le cas où le pivot est toujours un élément médian.*

Prouver la relation $C_{2^k-1} = (k-2)2^k + 2$. On pourra poser $d_k = C_{2^k-1} - 2$, et chercher une relation de récurrence pour (d_k).

En utilisant la croissance de la suite (C_n), en déduire deux constantes explicites A et B telles que l'on ait, pour tout $n > 0$,

$$An \log n \leq C_n \leq Bn \log n$$

Exercice 1.3 (QuickSort presque idéal) *Dans le cadre de l'algorithme **QuickSort**, on suppose que le choix du pivot est fait de telle sorte que chaque tableau de taille k soit toujours coupé en deux tableaux de taille au moins $k/4$.*

Trouver deux constantes A et B telles que le nombre de comparaisons utilisées pour trier un tableau de taille n soit toujours compris entre $An \log n$ et $Bn \log n$.

Exercice 1.4 *On considère l'algorithme **RandQuickSort**, dans lequel on retire l'hypothèse que le choix du pivot est toujours fait de manière uniforme ; on remplace cette hypothèse par la suivante :* dans un tableau de taille k, la probabilité pour chaque élément d'être choisi comme pivot est toujours comprise entre $1/2k$ et $2/k$. *On conserve en revanche l'hypothèse d'indépendance entre les différents choix de pivots.*

En reprenant l'analyse du cours, obtenir une majoration de la complexité moyenne de cet algorithme "affaibli".

Exercice 1.5 *Soit* Π *un problème pour lequel on dispose d'un algorithme Monte Carlo* $\mathcal{A}$ *qui, pour toute instance de taille* n*, s'exécute en temps au plus* $T_{\mathcal{A}}(n)$ *et donne une réponse correcte avec probabilité au moins* $p_{\mathcal{A}}(n)$ *; ainsi que d'un algorithme* $\mathcal{V}$ *qui, pour toute instance de taille* n *et toute réponse possible, décide en temps au plus* $T_{\mathcal{V}}(n)$ *si la réponse est correcte.*

1. *Décrire un algorithme Las Vegas pour le problème* Π.
2. *Donner une borne supérieure sur l'espérance du temps d'exécution de votre algorithme Las Vegas.*

Exercice 1.6 *Soit* Π *un problème de décision,* $0 < \alpha < 1$*, et* $\mathcal{A}$ *un algorithme Monte Carlo à erreur unilatérale pour* Π*, dont on suppose qu'il est à probabilité d'erreur au plus* α.

Soit $0 < \epsilon < \alpha$ *; combien de répétitions indépendantes de* $\mathcal{A}$*, au plus, sont nécessaires pour assurer que la probabilité d'avoir une réponse correcte soit au moins égale à* $1 - \epsilon$ *?*

Exercice 1.7 *Soit* Π *un problème de décision. La notion d'algorithme Monte Carlo à erreur bilatérale et de probabilité d'erreur* $1/2$ *a-t-elle un intérêt ? Qu'en est-il si la probabilité d'erreur n'est que de* 0.49 *? Dans le cas d'un algorithme à erreur unilatérale, comparer à l'Exercice 1.6.*

Exercice 1.8 *On considère une variante du problème de la coupe minimale : chaque arête du graphe* G *se voit attribuer un* poids, *et on cherche une coupe dont la* somme des poids des arêtes constituantes *soit minimale. Chercher une variante appropriée de l'algorithme* ***RandMinCut****, dans les deux cas suivants :*

1. *les poids sont des entiers positifs, compris entre 1 et une valeur maximale* M *;*
2. *les poids sont des réels strictement positifs.*

Exercice 1.9 *Utiliser l'inégalité de Markov pour majorer, en fonction de* n *et d'une constante* $c > 0$*, la probabilité que* ***RandQuickSort*** *ait besoin de plus de* cn^2 *comparaisons pour trier* n *éléments.*

(Il convient de noter que la majoration obtenue est très loin d'être optimale.)

Chapitre 2

Modèles d'algorithmes probabilistes

Dans ce chapitre, nous décrivons les modèles dans lesquels nous nous plaçons pour définir nos algorithmes, et explorons les relations entre diverses variantes de ces modèles.

2.1 Définition par la source

Intuitivement, un *algorithme probabiliste* est un algorithme qui a la possibilité d'effectuer des choix aléatoires. Reste alors à définir proprement ce que l'on entend par *choix aléatoires*.

Dans les exemples du chapitre 1, nous avons supposé qu'il était possible de "choisir au hasard" un élément d'un ensemble fini. Dans notre cas, cela revenait à obtenir un entier compris entre 1 et une certaine valeur n, *uniformément* (chaque entier ayant une même probabilité $1/n$ d'être obtenu). Dans d'autres cas, nous pourrons nous contenter d'une primitive plus simple, correspondant à *tirer à pile ou face avec une pièce équilibrée* (obtenir un symbole parmi $\{P, F\}$, chaque symbole ayant probabilité $1/2$ d'apparaître), ou, au contraire, nous pourrons supposer qu'il nous est possible de tirer un *réel* au hasard, selon la loi *uniforme* sur $[0, 1]$.

Dans tous les cas, une caractéristique essentielle est l'*indépendance* des différents tirages aléatoires réalisés. Cette indépendance est implicitement présente dans l'idée de tirage à pile ou face : si une même pièce est lancée un certain nombre de fois, connaître les résultats d'un nombre quelconque de lancers ne permet de faire aucun pronostic quant au résultat d'un autre lancer[1] – ou, plus précisément, le pronostic ne sera pas plus éclairé qu'il ne l'aurait été auparavant.

Définition 2.1 *Une* source aléatoire de loi μ *est une procédure qui, à chacun de ses appels, retourne une variable aléatoire de loi μ,* indépendante *de toutes les autres variables aléatoires déjà renvoyées. On distinguera différents cas particuliers :*

Source de Bernoulli *Une source de Bernoulli de paramètre p ($0 < p < 1$) est une source dont la loi est la loi de Bernoulli de paramètre p (qui attribue la probabilité p à 1, et $1 - p$ à 0).*

Source de bits aléatoires *Une source de Bernoulli de paramètre $1/2$ est également appelée source de bits aléatoires.*

[1]Noter que la pièce n'est pas d'emblée supposée *équilibrée* – si cette information n'est pas donnée, on peut évidemment envisager que l'observation d'un certain nombre de tirages préalables permettra d'émettre des doutes sur, ou au contraire de confirmer, cet équilibre. Mais il s'agit alors de statistiques (qu'apprend-on sur le système représenté par la pièce en l'observant) plutôt que de probabilités (que peut-on prévoir sur le comportement de ce système).

Source réelle *Une source dont la loi est la loi uniforme sur l'intervalle réel* $[0, 1]$*, sera appelée source réelle.*

Définition 2.2 *On appellera* algorithme probabiliste*, un algorithme faisant usage d'une source aléatoire.*

Les mesures classiques de complexité (en temps, en espace) s'appliquent également aux algorithmes probabilistes – avec la différence notable que, pour chaque entrée, ce sont des quantités *aléatoires*, dont on s'attachera à *encadrer* (majorer, minorer) des caractéristiques comme espérance et variance. Une nouvelle mesure s'ajoute toutefois : celle du *nombre de tirages aléatoires* utilisés, qui est également une variable aléatoire.

Comme on le verra (voir par exemple l'Exercice 2.2), les différents modèles de Bernoulli sont plus ou moins équivalents entre eux. La puissance du modèle de la source réelle n'est généralement ni nécessaire, ni réaliste ; elle permet toutefois des simplifications dans la description de certains algorithmes.

Définition 2.3 *On appellera* source de Bernoulli généralisée *une fonction qui, appelée avec un paramètre réel* p*, renvoie une variable aléatoire de Bernoulli de paramètre* p*, indépendante des variables aléatoires renvoyées lors d'autres appels.*

La source de Bernoulli généralisée constitue typiquement un bon compromis entre la simplicité de la source de bits, et la surpuissance inutile de la source réelle. Il n'est pas très difficile de construire une source de Bernoulli généralisée à partir d'une source de bits aléatoires, ce qui permet, par la suite, de supposer que l'on dispose d'une telle source.

Le modèle de source le plus proche des générateurs pseudo-aléatoires typiquement implantés sur la plupart des systèmes informatiques est celui de la source de bits aléatoire ; c'est également celui qui est le plus simple à analyser.

Dans la pratique, on supposera généralement qu'il est possible, en temps constant, de choisir un entier compris entre 0 et $n-1$, uniformément[2] ; cette hypothèse est légèrement irréaliste si n est grand, au moins dans un modèle basé sur une source de bits ($O(\log n)$ serait plus réaliste ; voir Exercice 2.4).

2.2 Comparaison entre les modèles de sources

L'exercice 2.2 propose une comparaison détaillée des différents modèles de sources aléatoires. En d'autres termes, on se place dans l'un des modèles de source, et on cherche à *simuler* un type différent de source.

Les questions essentielles pour le changement de modèle sont les suivantes. On suppose que l'on dispose d'une source aléatoire de loi μ, et qu'on cherche à "fabriquer" une source de loi ν.

1. quelle méthode employer ?
2. quelle sont les performances (en termes de nombre d'échantillons de la loi μ dont a besoin l'algorithme choisi pour produire un, ou n, échantillons de loi ν) ?
3. peut-on dire quelque chose sur les performances *optimales* ?

[2] Il s'agit ni plus ni moins ici que d'une source dont la loi est la loi uniforme sur l'intervalle (d'entiers) $[[0, n-1]]$.

En première approche, la première question se ramène à la suivante : comment un algorithme disposant d'une source de loi μ, peut-il engendrer une variable aléatoire distribuée suivant la loi ν ; il s'agit donc d'un problème de *génération de variable aléatoire*, sujet traité en détails dans le livre de Devroye [8]. Nous nous bornerons à examiner quelques exemples.

2.2.1 Fabrication générique d'une source de Bernoulli

La source la plus simple à construire est une source de Bernoulli de paramètre p, au moins pour certaines valeurs de p. La construction qui suit est fournie à titre d'exemple.

On suppose que l'on dispose d'une source de loi μ, et que l'on connait un ensemble de valeurs A pour la loi μ, tel que $0 < \mu(A) < 1$; la lecture depuis cette source est représentée par la fonction `TirageMu()`.

Entrée :
Sortie : une variable de Bernoulli, de paramètre $p = \mu(A)$
Utilise : `TirageMu()` : source de loi μ ; test d'appartenance à A.

1. $x \leftarrow$ `TirageMu()`
2. Si $x \in A$, retourner 1
3. Retourner 0

Algo. 2.1: Construction d'une source de Bernoulli

Il est immédiat que l'algorithme 2.1 fournit une construction pour une source de Bernoulli de paramètre $p = \mu(A)$; et, de manière certaine, il ne nécessite qu'un seul tirage de la loi μ pour chaque tirage de Bernoulli.

Il se peut toutefois qu'il soit difficile de trouver un ensemble A ayant exactement la bonne mesure $\mu(A)$, ou même qu'il n'existe pas de tel ensemble (penser au cas où la loi μ a un support fini et ne donne que des probabilités rationnelles aux singletons, et où l'on cherche à simuler une Bernoulli de paramètre $1/\pi$ ou, plus généralement, de paramètre irrationnel). Dans ce cas, plusieurs tirages de loi μ sont nécessaires, et le nombre de tirages sera lui-même une variable aléatoire *a priori* non bornée.

Dans le cas particulier où la loi μ est la loi uniforme sur $[0, 1]$ (source réelle), on peut bien évidemment prendre $A = [0, p]$.

2.2.2 D'une source de Bernoulli à une source de bits aléatoires

La construction précédente fournit une source de Bernoulli ; pour peu que l'on sache trouver un ensemble A tel que $\mu(A) = 1/2$, on peut alors l'appliquer pour fabriquer une source de bits aléatoires[3].

Toutefois, si l'on ne dispose que d'une source de Bernoulli de paramètre $p \neq 1/2$ (par exemple, l'équivalent d'une pièce de monnaie non équilibrée), la construction de l'algorithme 2.2 permet de construire une source de bits aléatoires, qui est la source "classique".

[3] Il convient de noter que l'on a également besoin de tester efficacement si une valeur appartient ou non à A.

Entrée :
Sortie : Un bit aléatoire
Utilise : `TirageBernoulli()` : source de Bernoulli de paramètre p

1. $x \leftarrow 0$, $y \leftarrow 1$
2. Répéter tant que $x \neq y$:
   (a) $x \leftarrow$ `TirageBernoulli()`
   (b) $y \leftarrow$ `TirageBernoulli()`
3. Retourner x.

Algo. 2.2: Construction d'une source de bits aléatoires

Correction et analyse de l'algorithme

Pour justifier de la correction de l'algorithme, nous devons prouver deux choses :

1. chaque invocation de l'algorithme se termine avec probabilité 1, et renvoie 0 ou 1 de manière équiprobable ;
2. deux invocations de l'algorithme renvoient des résultats *indépendants*.

Commençons par la terminaison : lors de chaque exécution de la boucle, l'algorithme se termine si les deux symboles x et y sont différents, c'est-à-dire si l'on a tiré 01 ou 10.

Soit $\mathcal{E}_k$ l'évènement "les tirages $2k-1$ et $2k$ donnent respectivement 1 et 0", et $\mathcal{F}_k$, "les tirages $2k-1$ et $2k$ donnent respectivement 0 et 1. En notant X_k le résultat du k-ème tirage, on a donc $\mathcal{E}_k = \{X_{2k-1}X_{2k} = 10\}$ et $\mathcal{F}_k = \{X_{2k-1}X_{2k} = 01\}$.

L'évènement $\mathcal{T}_k$, "l'algorithme se termine en exactement $2k$ tirages", est égal à

$$\mathcal{T}_k = (\mathcal{E}_k \cup \mathcal{F}_k) \cap \left(\bigcap_{1 \leq i < k} \overline{\mathcal{E}_i \cup \mathcal{F}_i} \right).$$

Les différents évènements $\mathcal{E}_i \cup \mathcal{F}_i$ sont indépendants entre eux (ils dépendent uniquement de tirages distincts, lesquels sont supposés indépendants), et chacun a pour probabilité $q = 2p(1-p)$. Par conséquent, l'évènement $\mathcal{T}_k$ a pour probabilité $(1-q)^{k-1}q$.

L'évènement $\mathcal{T}$ "l'algorithme se termine" est la *réunion* des évènements $\mathcal{T}_k$, qui sont deux à deux incompatibles ; par conséquent, on a

$$\begin{aligned} \Pr(\mathcal{T}) &= \sum_{k=1}^{\infty} \Pr(\mathcal{T}_k) \\ &= q \sum_{k=0}^{\infty} (1-q)^k \\ &= q \frac{1}{1-(1-q)} = 1. \end{aligned}$$

Nous avons donc prouvé que l'algorithme se termine avec probabilité 1.

Pour vérifier qu'il retourne effectivement 0 ou 1 de manière uniforme, il suffit de remarquer que l'évènement $\mathcal{U}_k$ "l'algorithme retourne 1 après $2k$ tirages" est exactement

$$\mathcal{U}_k = \mathcal{E}_k \cap \left(\bigcap_{1 \le i < k} \overline{\mathcal{E}_i \cup \mathcal{F}_i} \right),$$

et, de même, pour $\mathcal{Z}_k$ "l'algorithme retourne 0 après $2k$ tirages",

$$\mathcal{Z}_k = \mathcal{F}_k \cap \left(\bigcap_{1 \le i < k} \overline{\mathcal{E}_i \cup \mathcal{F}_i} \right).$$

Il est immédiat de vérifier que $\mathcal{U}_k$ et $\mathcal{Z}_k$ ont la même probabilité $\frac{1}{2}q(1-q)^{k-1}$, ce qui implique que les évènements $\mathcal{U}$ "l'algorithme retourne 1" et $\mathcal{Z}$ "l'algorithme retourne 0" ont des probabilités qui sont définies par la même séries, et sont donc égales (et donc, égales à 1/2).

Il resterait à démontrer l'*indépendance* des résultats renvoyés par un nombre quelconque d'invocations de l'algorithme. La preuve formelle de cette affirmation pourtant bien intuitive est relativement complexe, et n'est pas incluse ici[4].

Analyse de l'algorithme

Passons maintenant à l'analyse des performances de notre algorithme. Le calcul de la probabilité de $\mathcal{T}_k$ montre que la probabilité que l'algorithme 2.2 utilise $2k$ bits est $q(1-q)^{k-1}$, en quoi l'on reconnaît la probabilité qu'une variable *géométrique* de paramètre q prenne pour valeur k. Par conséquent, le *nombre de tirages de Bernoulli* utilisés par l'algorithme, X, est le double d'une variable géométrique de paramètre $q = 2p(1-p)$.

En particulier :
- $\mathbb{E}(X) = 2/q$;
- $\mathbf{Var}(X) = 4(1-q)/q^2$;
- $\mathbb{P}(X > 2k) = (1-q)^k$.

À titre d'exemple numérique, pour $p = 1/3$, on obtient $q = 0.44\ldots$ et il faut en moyenne 4.5 tirages de Bernoulli pour un bit aléatoire ; de plus, la probabilité d'avoir besoin de plus de k tirages, décroît exponentiellement vite avec k, ce qui est satisfaisant. Toutefois, lorsque p devient très proche de 1 (ou de 0), q se rapproche de 0 et les constantes se dégradent.

Remarque sur la construction

Il est intéressant de noter que l'analyse qui précède ne dépend pas de la valeur de p ; en fait, elle ne suppose même pas que l'on connaisse la valeur de p, mais seulement de la double hypothèse que p ne change pas de tirage en tirage, et que les différents tirages sont indépendants.

Il est donc possible, à partir d'une pièce de monnaie *non équilibrée* et dont les propriétés statistiques sont *inconnues*, de *simuler* une pièce parfaitement équilibrée. Cette simulation a toutefois un coût : le nombre moyen de lancers par bit produit est $\frac{1}{p(1-p)}$, qui est toujours supérieur à 4.

La construction inverse, d'une source de Bernoulli de paramètre p fixé (et connu) à partir d'une source de bits aléatoires, est suggérée dans l'exercice 2.2.

[4] Elle figurait dans une précédente version, mais ne servait guère qu'à embrouiller les choses.

2.2.3 Source réelle

La construction d'une source réelle (c'est-à-dire d'une suite de variables aléatoires indépendantes, équidistribuées, suivant la loi uniforme sur $[0, 1]$) se heurte au problème suivant : la connaissance complète d'une seule telle variable aléatoire, est équivalente (voir l'exercice 2.3) à celle d'une *infinité* de bits aléatoires.

Dans la pratique, le modèle de la source réelle est en fait bien trop puissant, et irréaliste (voir Exercice 2.3). Par l'expression "engendrer une variable aléatoire de loi μ", lorsque μ est une loi diffuse, on entend, de manière plus réaliste, l'une des deux actions possibles (par ordre croissant de difficulté) :

- engendrer une variable aléatoire "de type flottant", suivant une distribution "proche" de la distribution μ – de telle manière que, si la précision des calculs tend vers 0, la distribution effective "tend vers" la distribution μ ;
- engendrer une variable aléatoire qui est distribuée comme l'arrondi, à la précision souhaitée, de la loi μ ; plus précisément, si l'on s'intéresse à une génération à 2^{-k} près, on devrait avoir, pour tout nombre $m.2^{-k}$ avec $m \in \mathbb{Z}$, $\mathbb{P}(X = m.2^{-k}) = \mu([m.2^{-k}, (m+1).2^{-k}])$.
- fournir un préfixe de longueur finie d'une suite de bits aléatoires, si possible raffinable (prolongeable à volonté et à la demande), et qui soit le développement binaire d'une variable aléatoire de loi μ.

Dans la pratique, on se limite aux deux premières conditions.

2.3 Modèle de la bande aléatoire

La définition donnée plus haut d'un algorithme probabiliste au moyen d'une "source" aléatoire présente l'avantage d'être proche des implantation réelles, dans lesquelles une source pseudo-aléatoire est utilisée en remplacement de la "vraie" source aléatoire ; dans les systèmes informatiques réels, on trouve des fonctions de bibliothèque ou même des appels système qui fournissent une source pseudo-aléatoire de qualité variable. Cette définition n'est toutefois pas la seule possible.

Une autre définition possible présente le double avantage de permettre le rapprochement avec le modèle théorique classique de la machine de Turing, et de fournir naturellement une construction pour l'espace probabilisé sous-jacent ; nous l'appellerons le modèle de la bande aléatoire.

Définition 2.4 *On appellera* algorithme probabiliste*, un algorithme déterministe au sens classique, dont l'entrée est divisée en deux :*

- *une suite finie de symboles, dite "instance", et qui est seule sous le contrôle de l'utilisateur ;*
- *une seconde suite, infinie, de symboles, dite "bande aléatoire".*

L'algorithme peut, sous la forme d'une action élémentaire, "lire" le premier symbole encore non lu de la bande aléatoire.

L'ensemble des suites de symboles possibles pour la source aléatoire, est muni de la loi produit $\mu^{\times\mathbb{N}}$ *(en d'autres termes, chaque symbole de la source aléatoire est aléatoire de loi μ, et indépendant des autres).*

Nous ne donnerons pas de preuve formelle de l'équivalence de cette définition avec la précédente ; le passage d'un algorithme à source de loi μ au modèle de bande aléatoire muni

de la loi $\mu^{\times\mathbb{N}}$, et vice versa, est assez naturel. L'*espace produit* (voir Appendice A) correspond exactement à demander un accès à une *suite de tirages équidistribués indépendants* dont la loi est μ.

Lorsque la bande aléatoire est fixée, l'exécution de l'algorithme devient *déterministe* ; la précision de la loi de probabilités sur la bande aléatoire (qui découle du choix de la loi μ) induit donc une *loi de probabilités sur les exécutions de l'algorithme*, ce qui transforme les grandeurs usuelles de complexité en variables aléatoires.

Une troisième définition, que nous n'explorerons pas ici, propose de considérer des algorithmes dont *chaque instruction élémentaire*, au lieu d'avoir un résultat certain, a deux résultats possibles (qui peuvent être identiques), supposés équiprobables[5] ; une fois de plus, on suppose alors que les résultats de plusieurs instructions élémentaires sont indépendants. En particulier, une difficulté d'un tel modèle est la définition, comme mesure de complexité, de la grandeur "nombre de tirages aléatoires".

2.4 Alors, quelle définition ?

Il n'est pas important de choisir un modèle plutôt qu'un autre pour définir ce qu'est un algorithme probabiliste ; les différents modèles présentés sont essentiellement équivalents, au sens où tout ce qui peut être réalisé sous un modèle peut l'être sous l'autre, et qui plus est, avec des complexités sensiblement équivalentes[6].

Dans tous les cas, le principe est le même : un algorithme probabiliste est un algorithme dont l'*exécution* est, tout bien considéré, une *variable aléatoire* exprimée en fonction de tirages indépendants les uns des autres, et dont la loi est connue.

Le reste des définitions est là pour assurer que l'on reste dans un cadre *algorithmique* : ce n'est pas parce que l'on met une loi de probabilités sur un "ensemble d'exécutions" que l'on a un algorithme probabiliste. En particulier, la loi de probabilités en question doit vérifier une condition du type : l'exécution du programme, *jusqu'au $k+1$-ème appel de la source aléatoire*, dépend de manière *déterministe* des résultats des appels jusqu'au k-ème.

2.5 Réalisations de sources aléatoires

Les modèles de sources que nous avons présentés dans ce chapitre ont un inconvénient majeur : il est difficile de les implanter dans des machines dont on s'est efforcé d'assurer que le comportement est parfaitement déterministe. La discussion qui suit est essentiellement empruntée à Knuth [5], qui consacre près de la moitié d'un volume de son *Art of Computer Programming* à la génération de nombres pseudo-aléatoires.

2.5.1 Accumulateurs d'entropie

Sous le terme d'accumulateurs d'entropie, on regroupe toutes les méthodes "physiques" ou algorithmiques de production de valeurs non déterministes. Dès les balbutiements de l'informatique, au début des années 1950, certains ordinateurs disposaient d'une instruction permettant de remplir un registre avec des bits issus d'un "générateur de bruit blanc".

[5] Ce modèle correspond assez précisément à une source de bits aléatoires, ou à une bande aléatoire dont la loi μ de base est uniforme sur la paire $\{0, 1\}$.

[6] Oui, cette affirmation est floue et imprécise.

Le système Linux[7] propose un *device* `/dev/random`, dans lequel on peut lire des valeurs calculées par le système, *a priori* lors de basculements de tâches. Chaque octet n'est lisible qu'une seule fois, ce qui est censé donner une séquence d'octets aléatoires (et rend le dispositif assez similaire à ce que nous avons appelé source aléatoire, ou à une bande qui ne serait lisible qu'une fois). Un tel système ne permet qu'un débit assez lent ; il est toutefois utilisé par les applications cryptographiques comme `ssh` pour la génération de clés.

Une autre possibilité est d'utiliser le caractère intrinsèquement probabiliste de la mécanique quantique. Dans certaines conditions, la mécanique quantique prévoit que le résultat d'une mesure, même en tenant compte de tous les résultats de mesures préalables, a une probabilité $1/2$ de donner chacun des deux résultats possibles. Il est donc, en théorie, possible de baser une source de bits aléatoires parfaite sur ce principe. Dans la pratique, la réalisation de tels dispositifs est au bas mot difficile et coûteuse, surtout s'il s'agit d'obtenir une source *rapide*.

2.5.2 Générateurs pseudo-aléatoires

Outre la difficulté de les réaliser correctement, les sources réellement aléatoires ont, dans un contexte informatique, un inconvénient quelque peu paradoxal : s'ils sont réellement non déterministes, il est impossible de reproduire leur comportement, ce qui rend problématique le test et la mise au point de programmes. Dans ce contexte, la reproductibilité est un atout essentiel ; on se tourne donc vers des dispositifs qui ne fournissent que des valeurs *déterministes*, mais qui *semblent* aléatoires. On parle alors de *générateurs pseudo-aléatoires.*

Générateurs itératifs

Un grand nombre de générateurs pseudo-aléatoires fonctionnent suivant un schéma *itératif* : le générateur maintient une valeur de registre X (qui est, typiquement, un mot, voire un double mot, du processeur : $X \in \{0,1\}^l$, avec $l = 32$ par exemple). À chaque appel du générateur, X est remplacé par $f(X)$, et une certaine valeur $g(X)$ est renvoyée. En d'autres termes, un *générateur pseudo-aléatoire itératif* est défini par un triplet (X_0, f, g), où

- $X_0 \in \{0,1\}^l$ est le *mot d'initialisation* ;
- f est une fonction (à bien choisir !) $\{0,1\}^l \to \{0,1\}^l$;
- $g : \{0,1\}^l \to V$ est une fonction qui transforme un mot en une "valeur" ; par exemple, si l'on prétend obtenir une source de bits aléatoires, on prendra $V = \{0,1\}$. Souvent, g sera l'identité, ou consistera à ne prendre qu'une partie des bits de X.

En clair, la suite de valeurs de V produite est $(Y_k)_{k\geq 0}$, où

$$\begin{aligned} Y_n &= g(X_n) \\ X_n &= f(X_{n-1}) \end{aligned}$$

Il est clair que, quelles que soient les fonctions f et g, la suite (Y_n) est ultimement périodique, avec une période au plus égale au cardinal de l'ensemble image de f. Par conséquent, la suite produite ne peut imiter une suite aléatoire que tant qu'on ne l'observe pas sur une trop longue durée, et obtenir une suite de longue période est un objectif prioritaire. Une longue période ne saurait en revanche être suffisante : la fonction f définie par $f(X) = (X+1) \mod 2^l$ (un mot binaire de longueur est ici interprété comme un entier modulo 2^l), avec l'identité pour g, fournit clairement un générateur de période maximale, mais il est difficile de prétendre que la suite $5423, 5424, 5425\ldots$ est une très bonne approximation d'une suite aléatoire !

[7] Et peut-être d'autres versions d'Unix ?

Générateurs congruentiels linéaires

Une classe importante de générateurs pseudo-aléatoires itératifs est fournis par les *générateurs congruentiels linéaires*. Un tel générateur est défini par quatre entiers : un *module* $m > 0$, un *multiplicateur* a $(1 \leq a < m)$, un *incrément* c $(0 \leq c < m)$, et le vecteur d'initialisation $X_0 < m$. Le générateur est essentiellement défini par sa fonction d'itération

$$f(X) = (aX + c) \mod m$$

Sous Linux, la page de manuel `drand48` indique que le générateur pseudo-aléatoire fourni par les fonctions de la famille `?rand48` est effectivement un générateur congruentiel linéaire, qui utilise par défaut les valeurs $m = 2^{48}$, $a = 3740067437$, et $c = 11$[8].

Générateurs additifs à trous

Une autre classe fort simple à implanter de générateurs pseudo-aléatoires est fournie par des récurrences de type *Fibonacci à trous* comme

$$Y_n = (Y_{n-24} + Y_{n-55}) \mod m$$

(de tels générateurs sont itératifs, à condition de considérer un vecteur X contenant 55 valeurs consécutives de la suite Y ; ils sont implantables de manière élémentaire en utilisant un tableau circulaire de 55 mots)

De tels générateurs se comportent remarquablement bien (compte tenu de leur simplicité) vis-à-vis des tests statistiques (voir ci-dessous) ; on en trouvera une discussion détaillée dans Knuth [5], pages 10 à 40.

2.5.3 Validation de générateurs pseudo-aléatoires

Il n'est pas question de donner ici une description explicite des tests qui permettent, sinon de valider un candidat à être un "bon" générateur pseudo-aléatoires, du moins d'en exclure des quantités de mauvais ; on pourra pour cela se reporter à l'ouvrage de Knuth [5], ou à un ouvrage traitant de cryptologie comme [9] – la cryptologie est à la fois friande de générateurs pseudo-aléatoires, et exigeante sur leur qualité. Citons toutefois quelques-uns des tests proposés par Knuth.

La première catégorie de tests est *statistique* : il s'agit alors de vérifier que des parties, de longueur grande mais finie, de la suite engendrée, ont des propriétés statistiques qui correspondent à celles d'une suite de valeurs aléatoires indépendantes. Pour la plupart, ces tests se ramènent ultimement à un test du χ^2 sur des statistiques censées être indépendantes. Quelques exemples :

Test de fréquence On regroupe les valeurs pouvant être prises par la suite X en un petit nombre k de catégories dont on calcule les probabilités théoriques, et on applique le test du χ^2 (à $k-1$ degrés de liberté) aux fréquences empiriques de ces catégories.

Test de séries Il s'agit de vérifier que des valeurs consécutives de la suite semblent effectivement indépendantes. Essentiellement, on découpe la suite en tronçons de longueur 2 (X_{2j}, X_{2j+1}) ou 3 $(X_{3j}, X_{3j+1}, X_{3j+2})$, et on applique le test de fréquence à la suite des tronçons.

[8] La même page de manuel précise aussi que ce générateur doit être considéré comme obsolète, et que les fonctions `rand` ou `random` devraient lui être préférées.

Test des intervalles Pour deux valeurs α et β quelconques, on mesure les intervalles (nombres de termes de la suite (X_n)) entre deux passages dans l'intervalle $[\alpha, \beta]$. Ces intervalles, pour une suite aléatoire, devraient avoir une distribution géométrique ; on "tronque" la géométrique à une certaine valeur finie t, et on applique le test du χ^2 à $t-1$ degrés de liberté.

Test du collectionneur de coupons Un peu comme pour le test de fréquence, on découpe les valeurs de la suite (X_n) en un petit nombre k de catégories, et cette fois on mesure les longueurs des segments qui visitent toutes les catégories (ainsi, pour $k = 3$, la suite

$$1101221022120202001$$

a comme segments (11012), (210), (22120) et (202001) de longueurs respectives 5, 3, 5, 6). La distribution théorique, quand k est petit, de ces longueurs est connue, et on peut une fois de plus appliquer un χ^2 à une version tronquée.

Test des permutations Pour k fixé, des tronçons de longueur k de la suite (X_k), de la forme $(X_{kj}, X_{kj+1}, \dots, X_{kj+k-1})$ devraient présenter des valeurs qui sont dans un ordre aléatoire parmi les $k!$ permutations possibles. Le test des permutations consiste à vérifier que ces $k!$ permutations sont effectivement équiprobables.

Enfin, il est parfois possible d'effectuer des tests *théoriques* sur certains générateurs pseudo-aléatoires – tests qui peuvent par exemple prouver que, sur l'ensemble de sa période, le générateur prend autant de valeurs paires qu'impaires, ou que la moitié des valeurs paires sont suivies de valeurs impaires. De tels tests théoriques ont leur place, mais, tels quels, sont souvent incapables de détecter des anomalies locales – qui peuvent par exemple survenir sur des tronçons longs, mais très courts devant la période.

2.6 Exercices

Exercice 2.1 *Soit μ une loi de probabilités sur $\mathbb{R}$, dont la fonction de répartition est F (c'est-à-dire, si une variable aléatoire X suit la loi μ, on a $\mathbb{P}(X \leq x) = F(x)$ pour tout $x \in \mathbb{R}$). On suppose que l'on dispose d'une fonction* ***Frec****, qui calcule la fonction réciproque de F ; montrer qu'une source réelle permet alors de construire une source de loi μ.*
Application : *Indiquer comment engendrer une variable aléatoire suivant la loi exponentielle de paramètre $\lambda > 0$ (dont la densité est $\lambda e^{-t/\lambda}$). Montrer au passage que si X_λ et X_μ sont deux variables aléatoires de paramètres respectifs λ et μ, alors X_λ/λ et X_μ/μ ont la même loi.*

Note : la simulation classique de variables aléatoires par la méthode sugérée dans cet exercice, bien qu'efficace en apparence, fait généralement appel, pour des lois diffuses, à des calculs en virgule flottante qui sont source d'imprécision – en supposant que l'on dispose d'un moyen efficace de calculer la réciproque de la fonction de répartition. Il existe, pour de nombreuses lois de probabilité, des méthodes plus sophistiquées et plus précises.

Exercice 2.2 (Équivalence entre différents modèles) *Le but du présent exercice est de montrer en quel sens les différents modèles de sources que nous avons décrits sont plus ou moins équivalents.*

1. *Comment fabriquer une source de Bernoulli de paramètre p (quelconque) à partir d'une source réelle ?*

2. *Décrire un algorithme permettant, à partir d'une source de Bernoulli de paramètre* p*, de fabriquer une source de bits aléatoire. Étudier sa complexité moyenne : quel est, en moyenne, le nombre d'appels de la source de Bernoulli qui sont nécessaires pour un appel de la source de bits aléatoires ?*

   Indication : *on pourra remarquer que, indépendemment de* p*, les probabilités de trouver, en deux positions successives,* 01 *ou* 10 *sont les mêmes.*

3. *Inversement, décrire un algorithme permettant, à partir d'une source de bits aléatoires, de fabriquer une source de Bernoulli de paramètre* $1/4$*, et étudier sa complexité moyenne, et sa complexité dans le cas le pire.*

4. *Même question dans le cas où l'objectif est d'obtenir une source de Bernoulli de paramètre* $1/3$*; montrer en particulier qu'il ne peut exister de solution pour laquelle la complexité dans le cas le pire soit finie. Généraliser l'étude à la construction, à partir d'une source de bits aléatoires, d'une source de Bernoulli de paramètre* p *quelconque, pour peu que l'on connaisse le développement binaire de* p *(ou que l'on dispose d'un moyen de le calculer).*

Exercice 2.3 *L'objectif de cet exercice est de se convaincre de ce que le modèle de la source réelle, si l'on suppose que l'on dispose d'une précision absolue pour les calculs sur des nombres réels, est irréaliste.*

1. *Soit* $(X_n)_{n\geq 1}$ *une suite infinie de bits aléatoires indépendants. On définit* $U_n = \sum_{k\leq n} X_k 2^{-k}$*, et* $U = \lim U_n$*. Déterminer les lois de probabilités de* U_n *et* U*.*

2. U *étant la variable aléatoire définie précédemment, montrer que l'on peut, avec probabilité 1, retrouver chacune des variables aléatoires* X_k *(expliquer comment). En déduire qu'une seule variable aléatoire de distribution uniforme sur* $[0,1]$*, permet de construire toute une source de bits aléatoires* $(B_n)_{n\in\mathbb{N}}$*.*

3. *Montrer comment une variable aléatoire* U*, de distribution uniforme sur* $[0,1]$*, permet de construire deux variables aléatoires* U_1 *et* U_2*, uniformes sur* $[0,1]$*, indépendantes l'une de l'autre.*

4. *En déduire qu'il est équivalent, en faisant l'hypothèse d'une précision absolue sur les nombres réels, de disposer d'*une *variable aléatoire uniforme sur* $[0,1]$*, ou d'une* source *de telles variables.*

5. *Expliquer en quoi l'analyse précédente s'effondre si l'on suppose que* U *n'est connue qu'avec une précision de* k *bits.*

Note : le modèle de la source réelle n'est pas seulement irréaliste ; il est également trop puissant pour la plupart des applications usuelles. En effet, chaque fois que l'on utilise une variable uniforme U pour prendre une décision, il est parfaitement possible de tirer le *début* du développement binaire de U (exercice précédent) jusqu'à être capable de prendre la décision ; cela peut être fait en n'utilisant qu'une source de bits.

Une confusion supplémentaire est introduite par le fait que les générateurs pseudo-aléatoires implantés sur les systèmes réels prétendent généralement fournir une variable réelle uniforme U ; dans la pratique, ils ne fournissent qu'une mantisse d'un nombre fixé de bits, ce qui est amplement suffisant pour la plupart des utilisations – mais pas équivalent.

Exercice 2.4 *L'objectif est ici de sélectionner un entier compris entre* 0 *et* $k-1$*; une telle primitive est implicitement supposée disponible dans les algorithmes du chapitre 1.*

1. *Montrer comment, en partant d'une source de Bernoulli généralisée, il est possible de choisir un entier aléatoire entre* 0 *et* $n-1$*, uniformément. Étudier la complexité, en moyenne et dans le cas le pire, de votre algorithme.*
2. *Même question dans le cas où l'on ne dispose que d'une source de bits aléatoires. On distinguera deux cas, suivant que* n *est ou non une puissance de 2, et on montrera en particulier qu'aucun algorithme de complexité bornée (dans le cas le pire) ne peut exister si* n *n'est pas une puissance de 2.*

Exercice 2.5 ***(Adapté de Knuth, d'après Walker (1974))*** *Soit* $\pi = (p_0, \dots, p_{k-1})$ *une distribution de probabilités quelconque sur les entiers de* 0 *à* $k-1$ *(c'est-à-dire que* $p_1 + \cdots + p_{k-1} = 1$*). On souhaite tirer un entier aléatoires selon la distribution* π*.*

1. *Dans un premier temps, on se propose de le faire en utilisant le schéma suivant :*
   - *Obtenir une variable aléatoire uniforme* U *sur* $[0, 1]$ *;*
   - *Déterminer* I *tel que*
   $$\sum_{0 \le i < I} p_i \le U < \sum_{0 \le i \le I} p_i;$$
   - *Sélectionner* I*.*

   Montrer que ce schéma est correct. Évaluer sa complexité.
2. *Montrer qu'il est possible (et donner un algorithme ; on pourra rechercher un algorithme de type glouton) de mettre en place deux tableaux* P *et* Y*, respectivement de* k *réels et de* k *entier, de telle sorte que le schéma suivant soit correct :*
   - *Obtenir un entier aléatoire* K*, uniforme sur* $[0, k-1]$*, et un réel aléatoire* V*, uniforme sur* $[0, 1]$*, indépendante de* K *;*
   - *Si* $V \le P_K$*, retourner* K *; sinon, retourner* Y_K*.*
3. *Montrer que l'on peut obtenir le couple* (K, V) *à partir de* U *(uniforme sur* $[0, 1]$*) par* $K = \lfloor kU \rfloor$*,* $V = kU - K$*.*

Chapitre 3

Sélection probabiliste

Dans ce chapitre, nous nous intéressons au problème de la *sélection* d'un élément d'un tableau non trié, identifié par son rang. L'étude d'algorithmes probabilistes efficaces pour ce problème sera l'occasion de mettre en pratique l'*inégalité de Tchebycheff*. De plus, l'un des algorithmes probabilistes présentés ici a la particularité d'avoir une complexité *moyenne* équivalente (asymptotiquement) à une minoration connue pour les algorithmes déterministes – en d'autres termes, notre algorithme **RandLazySelect** (Algorithme 3.1) est, en moyenne, au moins aussi rapide que tout algorithme déterministe résolvant de manière générique le problème de sélection.

3.1 Le problème de la sélection

Le problème de la *sélection* est essentiellement le suivant : étant donnés un ensemble de n objets, chacun étant caractérisé par une clé appartenant à un univers totalement ordonné, et un entier $k \leq n$, obtenir l'objet x_k dont la clé est la k-ème dans l'ordre croissant (c_k si les n clés, ordonnées en ordre croissant, sont $c_1 < c_2 < \cdots < c_n$).

Dans l'ensemble du chapitre, nous supposerons que l'ensemble est représenté sous la forme d'un tableau, permettant ainsi d'accéder à une clé en temps constant.

Il est évident que, si l'ensemble de clés est trié, la complexité du problème de sélection est essentiellement celle de l'accès à un élément (temps $O(1)$, donc, avec un tableau) ; et, en tout état de cause, il est toujours possible de trier un tableau en temps $O(n \log n)$ (de manière déterministe, par exemple avec un tri par tas ; ou, de manière probabiliste, en moyenne dans le cas le pire, avec **RandQuickSort**, qui a typiquement une meilleure constante multiplicative que le tri par tas). Mais peut-on faire mieux ?

Comme pour le problème du tri, nous mesurerons les complexités sous la forme de *nombres de comparaisons de clés deux à deux*, et nous ferons l'hypothèse que les comparaisons sont les seules opérations possibles sur l'univers des clés. De plus, nous supposerons que les clés sont deux à deux distinctes, ce qui assure que le k-ème élément est uniquement déterminé.

Fixons d'ores et déjà quelques notations. Afin d'éviter de les surcharger, nous ne ferons pas de différence entre chaque objet et sa clé. Le *rang* d'un objet x dans un ensemble S sera noté $r_S(x)$; inversement, l'objet de rang k dans l'ensemble S sera noté $S_{(k)}$ (à ne pas confondre avec la notation $S[k]$ que nous utiliserons pour désigner l'objet d'indice k dans un tableau). Le problème de la sélection est donc d'identifier $S_{(k)}$.

3.2 Bornes théoriques de complexité

Exercice 3.1 *Montrer qu'aucun algorithme, sur aucune instance, ne peut prétendre résoudre exactement le problème de la sélection en utilisant moins de $n-1$ comparaisons.*

On trouvera chez Knuth [6], une preuve courte du fait que, pour $k=2$, une minoration plus précise serait $n-2+\lceil \log_2 n \rceil$.

Par ailleurs, d'après Motwani et Raghavan [7], il a été prouvé que, pour le cas particulier du calcul d'un *median* ($k=\lfloor n/2 \rfloor$), une minoration asymptotique pour les algorithmes déterministes de sélection est $2n$ (c'est-à-dire qu'aucun algorithme déterministe pour le calcul du médian ne peut travailler en temps $(2-\epsilon)n+o(n)$ dans le cas le pire, si $\epsilon>0$). De plus, le meilleur algorithme déterministe connu n'atteint pas cette limite de $2n$, puisqu'il travaille en $3n+o(n)$. L'algorithme que nous décrivons ci-dessous est loin de ce meilleur algorithme connu ; il s'agit seulement de donner un exemple d'algorithme déterministe de complexité linéaire.

3.3 Algorithme déterministe

Nous décrivons maintenant un algorithme déterministe, raisonnablement simple à décrire et à analyser, pour le problème de la sélection, et qui fonctionne en temps $O(n)$ dans le cas le pire.

Le principe est le suivant : on suppose que le nombre n de clés est de la forme $7(2q+1)$ (ce qui peut être assuré en ajoutant, au plus, 13 clés). Ces clés sont alors séparées en $2q+1$ ensembles de 7, et chaque ensemble est trié au moyen d'un algorithme optimal.

Exercice 3.2 (Tri de 7 éléments) *La minoration absolue sur le nombre de comparaisons nécessaires pour trier 7 éléments est $\lceil \log_2(7!) \rceil = 13$. Trouver un algorithme qui trie effectivement 7 éléments en au plus 13 comparaisons.*

Indication : *On commencera par comparer 3 paires indépendantes, puis on triera les 3 éléments maximaux des paires. On se posera également la question du nombre minimal de comparaisons nécessaires pour insérer un nouvel élément dans une liste déjà triée.*

Revenons à notre algorithme déterministe pour la sélection. Chacun des $2q+1$ sous-ensembles est triable en 13 comparaisons, ce qui permet de déterminer les $2q+1$ médians des sous-ensembles. On détermine alors le "médian des $2q+1$ médians" x (récursivement, en utilisant le même algorithme), puis son rang $r_S(x)$ dans l'ensemble entier (ce qui, une fois x connu, nécessite $4q$ comparaisons supplémentaires : dans $2q$ listes triées de 3 éléments, on compare d'abord x au médian, puis à un autre élément). Ce faisant, on peut partitionner explicitement $S-\{x\}$ en $S_{<x}$ et $S_{>x}$. Le médian-des-médians étant plus grand qu'exactement q des médians, il est plus grand qu'au moins $4q+3$ (et qu'au plus $10q+3$) des éléments de S ; en d'autres termes, $S_{<x}$ et $S_{>x}$ ont chacun entre $4q+3$ et $10q+3$ éléments.

Une fois connu le rang de x (et $S-\{x\}$ séparé en deux ensembles dont les éléments sont séparés par x), il ne reste plus qu'à appliquer le même algorithme, récursivement, à l'une des deux parties, qui est maintenant de taille au plus $10q+3$.

Il resterait à préciser comment trouver le médian pour de *petites* valeurs de n (afin de terminer la récursion) ; n'importe quel algorithme, même un qui trie complètement le tableau, est acceptable pour notre propos, qui est seulement d'exhiber un algorithme déterministe en temps linéaire. Alors, l'analyse précédente montre que, si $F(n)$ est le nombre maximal de

comparaisons nécessaires à trouver l'élément $S_{(k)}$ dans un tableau de taille inférieure ou égale à n (pour la pire valeur de k), on a

$$F(n) \leq 30q + 13 + F(2q+1) + F(10q+3)$$

dès lors que $14q - 6 \leq n \leq 14q + 7$. Cette récurrence, à son tour, permet de montrer que l'on peut avoir $F(n) \leq 15n - 163$ pour $n > 32$.

3.4 Un premier algorithme probabiliste

Une solution naturelle, pour résoudre le problème de la sélection, est de réutiliser l'idée de **QuickSort** et de **RandQuickSort** : utiliser un pivot (que l'on pourra choisir soit de manière déterministe, soit de manière aléatoire), comparer chaque autre élément à ce pivot pour séparer les éléments plus grands que le pivot de ceux qui sont plus petits, et, en comparant le rang cherché k au nombre d'éléments plus petits que le pivot, continuer la recherche récursivement sur l'une ou l'autre des parties.

Exercice 3.3 *On considère la version* déterministe *de l'algorithme : à chaque étape, le pivot est, par exemple, le premier élément du tableau.*

Déterminer, en fonction de k et n, la complexité dans le cas le pire.

Exercice 3.4 *On considère maintenant la version* randomisée *de l'algorithme, définie par le fait que, lorsque l'on a n éléments parmi lesquels chercher celui de rang k, le pivot est choisi aléatoirement, uniformément parmi les n possibilités.*

Soit, pour $i < j$, p_{ij} la probabilité que les éléments $S_{(i)}$ et $S_{(j)}$ soient comparés au cours de la recherche, et X_{ij} la variable aléatoire qui compte le nombre de comparaisons de $S_{(i)}$ et $S_{(j)}$.

- *Expliciter la relation entre p_{ij} et X_{ij}.*
- *Soit X la variable aléatoire qui compte le nombre de comparaisons effectuées par l'algorithme. Quelle est la relation entre X et les différentes variables aléatoires X_{ij} ?*
- *En distinguant les cas $k < i < j$, $i < k < j$ et $i < j < k$, déterminer dans chaque cas $p_{i,j}$.*
- *En déduire une expression exacte, en fonction de n et k, pour la complexité moyenne $\mathbb{E}(X)$ de l'algorithme. Estimer le plus précisément possible cette complexité pour la pire valeur possible de k (à n fixé).*

3.5 L'algorithme RandLazySelect

L'algorithme précédent est raisonnablement efficace – il est essentiellement aussi bon que les meilleurs algorithmes déterministes connus, en moyenne du moins. Nous allons toutefois en étudier un autre, appelé **RandLazySelect**, qui présente la particularité de ne nécessiter, *avec forte probabilité* et en espérance, que $2n+o(n)$ comparaisons – ce qui le rend asymptotiquement aussi efficace que la minoration théorique pour les algorithmes déterministes, et, en espérance toujours, strictement meilleur que les meilleurs algorithmes déterministes connus.

Contrairement aux algorithmes précédents, l'approche de cet algorithme n'est *pas* de type “diviser pour régner” ; on ne cherche pas à utiliser un pivot et à procéder récursivement. L'idée de **RandLazySelect** est de trouver rapidement, parmi les éléments de S, deux éléments a et b tels que, *avec très forte probabilité*, chacune des deux propriétés suivantes soit vraie :

- $S_{(k)}$ se trouve entre a et b (ou, en d'autres termes, $r_S(a) \leq k \leq r_S(b)$;
- l'ensemble P des élements de S qui se trouvent entre a et b (au nombre de $r_S(b) - r_S(a) + 1$) est "petit".

Si ces deux conditions sont remplies, nous pouvons alors, en comparant a et b à chaque élément de S, calculer à la fois P, $r_S(a)$, et $r_S(b)$, puis trier complètement P pour trouver $S_{(k)} = P_{(k-r_S(a)+1)}$. Notre définition de "petit" revient donc essentiellement à demander que P puisse être trié complètement sans que le temps total de l'algorithme n'en soit sensiblement affecté.

Comment trouver ces a et b miraculeux ? L'idée est la suivante : si l'on tire au hasard un nombre grand (mais petit par rapport à n) d'éléments de S, les rangs dans S de ces éléments seront probablement répartis à peu près uniformément entre 1 et n. En gros, si l'on sélectionne une proportion $1/n^{1/4}$ des éléments de S pour former un ensemble R, les éléments de R devraient avoir des rangs dans S qui sont de l'ordre de $n^{1/4}$ fois leurs rangs dans R. L'essentiel de la démonstration des performances de **RandLazySelect** consiste précisément à donner un sens précis à cette idée générale.

Entrée : un ensemble S de n valeurs comparables, un entier $k \leq n$
Sortie : $S_{(k)}$

1. $x \leftarrow kn^{-1/4}$
2. $\ell \leftarrow \max(\lfloor x - \sqrt{n} \rfloor, 1)$, $h \leftarrow \min(\lceil x + \sqrt{n} \rceil, n)$
3. Tirer indépendamment $m = \lceil n^{3/4} \rceil$ éléments aléatoires de S (avec remise) pour former un multi-ensemble T.
4. Trier T en temps $O(n^{3/4} \log n)$
5. $a \leftarrow T_{(\ell)}$, $b \leftarrow T_{(h)}$
6. En comparant a et b à chaque élément de S, déterminer $r_S(a)$, $r_S(b)$, $\{s \in S : s < a\}$, $\{s \in S : a < s < b\}$ et $\{s \in S : b < s\}$.
7. – Si $k < n^{1/4}$, $P \leftarrow \{s \in S : s \leq b\}$
   – Sinon si $k > n - n^{1/4}$, $P \leftarrow \{s \in S : s \geq a\}$
   – Sinon $P \leftarrow \{s \in S : a \leq s \leq b\}$
8. Si $|P| > 4n^{3/4} + 2$, ou si $S_{(k)} \notin P$, retourner en (3).
9. Trier P en temps $O(n^{3/4} \log n)$, et retourner $S_{(k)} = P_{(k-r_S(a)+1)}$.

Algo. 3.1: RandLazySelect

Il est clair que, si les étapes (1) à (8) de **RandLazySelect** ne sont exécutées qu'une seule fois, le nombre total de comparaisons est

$$2n + O(n^{3/4} \log n)$$

(seule l'étape (6) effectue un nombre linéaire de comparaisons).

Pour montrer que **RandLazySelect** trie en temps $2n + o(n)$ avec probabilité $1 - o(1)$, il nous faut donc majorer la probabilité des deux événements $\{S_{(k)} \notin P\}$ et $\{|P| > 4n^{3/4} + 2\}$ – essentiellement, nous allons montrer que ces deux probabilités tendent vers 0.

Le principe des preuves est assez simple : puisque l'ensemble T comprend une fraction $1/n^{1/4}$ des éléments de S, le rang de a dans S devrait être de l'ordre de $k - n^{3/4}$, et celui de

b, de l'ordre de $k + n^{3/4}$. Donc, $S_{(k)}$ devrait, avec bonne probabilité, se trouver entre a et b, et l'ensemble P devrait contenir de l'ordre de $2n^{3/4}$ éléments. Par conséquent, il devrait être possible de prouver que les deux évènements considérés sont peu probables, pour peu que les variables aléatoires $r_S(a)$ et $r_S(b)$ s'écartent peu de leur espérance – notre outil final dans cette analyse sera l'inégalité de Tchebycheff.

Une note toutefois, avant d'entamer les calculs menant à l'estimation de complexité de cet algorithme. Les constantes utilisées dans la description de **RandLazySelect** peuvent apparaître comme "magiques" ; il n'en est rien. En fait, une description plus intuitive commencerait par "tirer un nombre $F_1(n)$, que nous déterminerons plus tard, d'éléments de S", et préciserait au fur et à mesure les conditions à imposer aux différentes fonctions de n qui paramétrisent la description de l'algorithme (par exemple, il est essentiel que trier $F_1(n)$ éléments se fasse en temps $o(n)$, ce qui implique que $F_1(n) = o(n)$) ; une telle démarche semble plus naturelle, mais implique des calculs d'optimisation (que ce soit pour optimiser la complexité, ou l'estimation que l'on en obtient) qui peuvent eux aussi être assez lourds.

3.5.1 Probabilité de *rater* $S_{(k)}$

Commençons par majorer la probabilité de l'évènement $\mathcal{A} = \{S_{(k)} \notin P\}$. Nous nous plaçons dans le cas $n^{3/4} \leq k \leq n - n^{3/4}$, ce qui assure que l'on a $\ell = \lfloor x - \sqrt{n} \rfloor$ et $h = \lceil x + \sqrt{n} \rceil$; les cas extrêmes, où k est plus proche de 0 ou de n, sont plus simples.

Soit B le nombre d'éléments de T qui sont inférieurs ou égaux à $S_{(k)}$: B est une variable *binomiale* de paramètres m (nombre d'éléments de T) et k/n (probabilité pour un élément aléatoire d'être inférieur ou égal à $S_{(k)}$). Une variable binômiale de paramètres N et p a pour espérance et pour variance, respectivement, Np et $Np(1-p)$. Ici, nous avons donc pour espérance km/n, soit x[1].

$\mathcal{A}$ correspond au cas où $S_{(k)}$ est, soit plus petit que $T_{(\ell)}$ (ce qui signifie que B est inférieure à $\mathbb{E}(B) - \sqrt{n}$), soit plus grand que $T_{(h)}$ (c'est-à-dire que B est supérieure à $\mathbb{E}(B) + \sqrt{n}$). Définissons $\lambda = \sqrt{n}/\sigma_B$: on a donc (c'est exactement l'inégalité de Tchebycheff !) $\mathbb{P}(\mathcal{A}) = \mathbb{P}(|B - \mathbb{E}(B)| \geq \lambda\sigma_B) \leq 1/\lambda^2 = \sigma_B^2/n$. Or, nous avons $\sigma_B^2 \leq \mathbb{E}(B) = km/n$, donc

$$\mathbb{P}(\mathcal{A}) \leq \frac{km}{n^2} \leq n^{-1/4}.$$

(L'important est ici que cette probabilité tende vers 0 quand n tend vers $+\infty$)

3.5.2 Probabilité que P soit trop grand

Le second évènement qui pourrait faire échouer la première passe de **RandLazySelect** est $\mathcal{C} = \{|P| > 4n^{3/4} + 2\}$; nous allons également montrer que sa probabilité est faible. Une fois de plus, nous nous plaçons dans le cas le plus difficile, qui correspond à $n^{3/4} \leq k \leq n - n^{3/4}$, soit $\sqrt{n} \leq x \leq n^{3/4} - \sqrt{n}$.

Le nombre d'éléments de P est $r_S(b) - r_S(a) + 1$; par conséquent, pour que $\mathcal{C}$ se produise, il faut que l'on ait $r_S(b) - r_S(a) \geq 4n^{3/4} + 2$.

Donc, pour que $\mathcal{C}$ se produise, il faut au moins que, soit $r_S(b)$ soit supérieur à $k + 2n^{3/4} + 1$, soit $r_S(a)$ soit inférieur à $k - 2n^{3/4} - 1$. En d'autres termes, on a $\mathcal{C} \subset \mathcal{E} \cup \mathcal{F}$, avec

$$\begin{aligned} \mathcal{E} &= \{r_S(b) > k + 2n^{3/4} + 1\} \\ \mathcal{F} &= \{r_S(a) < k - 2n^{3/4} - 1\} \end{aligned}$$

[1] À $1/n$ près...

Pour majorer convenablement la probabilité de $\mathcal{C}$, il nous suffit donc de majorer les probabilités de $\mathcal{E}$ et $\mathcal{F}$.

L'évènement $\mathcal{E}$ se produit si, parmi les m choix indépendants d'éléments de S qui sont choisis pour T, le h-ème plus petit est strictement plus grand que $k + 2n^{3/4} + 1$ éléments de S ; ou, dit autrement, si moins de h des éléments choisis pour T sont parmi les $k + 2n^{3/4} + 1$ plus petits de S.

Soit, une fois de plus, B^+ la variable aléatoire égale au nombre des éléments de T qui sont choisis parmi les $k + 2n^{3/4} + 1$ plus petits de S. Chacun des éléments de T étant choisi uniformément, indépendemment des autres, B^+ est donc une variable aléatoire *binomiale* de paramètres m et $p^+ = (k + 2n^{3/4} + 1)/n$.

De même, soit B^- la variable aléatoire égale au nombre des éléments de T qui sont parmi les $k - 2n^{3/4} - 1$ plus petits de S : B^- est une variable binomiale de paramètres m et $p^- = (k - 2n^{3/4} - 1)/n$.

La probabilité de $\mathcal{E}$ est donc la probabilité que B^+ soit inférieure à h ; de même, la probabilité de $\mathcal{F}$ est la probabilité que B^- soit supérieure à ℓ.

Rappelons qu'une variable binomiale de paramètres m et p, a pour espérance mp, et pour variance $mp(1-p)$, que l'on peut majorer par mp. Dans le cas de B^+, on obtient $\mathbb{E}(B^+) = x + 2\sqrt{n} + O(n^{-1/4})$, soit $h + \sqrt{n}$. Or, l'écart-type de B^+ (la racine carrée de la variance) est au plus de l'ordre de $n^{3/8}$. Par conséquent, un écart supplémentaire de $\sqrt{n}$ représente au moins $\Omega(n^{1/8})$ fois cet écart-type, et l'inégalité de Tchebycheff nous donne

$$\mathbb{P}(\mathcal{E}) = O(1/n^{1/4})$$

De manière similaire, on obtient la même majoration pour $\mathbb{P}(\mathcal{F})$.

La preuve est maintenant complète (au moins pour $n^{3/4} \leq k \leq n - n^{3/4}$) : avec probabilité $1 - O(n^{-1/4})$, l'algorithme **RandLazySelect** se termine en $2n + O(n^{3/4} \log n)$ comparaisons ; de plus, le nombre moyen de tentatives nécessaires pour obtenir un ensemble T convenable est une variable géométrique de paramètre $1 - O(n^{-1/4})$, et donc d'espérance $1 + O(n^{-1/4})$, et par conséquent le nombre moyen de comparaisons de l'algorithme est également $2n + o(n)$.

Exercice 3.5 *Compléter la preuve, en démontrant des inégalités similaires dans le cas* $k < n^{3/4}$.

3.5.3 Commentaires finaux

Nous avons démontré pour l'algorithme **RandLazySelect** une propriété particulière : sa complexité *moyenne* est atteinte *avec grande probabilité*. Essentiellement, cela signifie que cette complexité se comporte "presque" comme si elle était déterministe, et fonction uniquement de n (il convient toutefois de noter que la vitesse de convergence que nous avons démontrée est relativement lente : un facteur $1/n^{1/4}$ n'est pas écrasant de rapidité). Le fait que cette complexité moyenne et presque sûre[2] soit asymptotiquement égale à une minoration pour la complexité de tout algorithme déterministe, prouve que cet algorithme est réellement performant.

Au niveau technique, l'outil principal permettant l'analyse est l'*inégalité de Tchebycheff* – utilisée ici, non pas (comme on pourrait s'y attendre) sur la complexité, mais sur une variable aléatoire annexe, pour montrer qu'une certaine probabilité tend vers 0.

[2] Le terme "presque sûr" n'est pas ici pris dans son sens mathématique de "avec probabilité 1", mais dans le sens plus informel de "avec probabilité tendant vers 1".

D'un point de vue plus général, notons que l'algorithme **RandLazySelect** est essentiellement un algorithme de type Monte Carlo (avec la particularité de signaler les cas d'échec) répété jusqu'à obtenir un succès ; il se trouve simplement, et c'est là le cœur de l'analyse, que sa probabilité de succès dès la première tentative tend vers 1 lorsque la taille du problème tend vers l'infini.

Chapitre 4

Algorithmes issus de la théorie des nombres

La théorie des nombres, et à travers elle la cryptologie, est friande d'algorithmes probabilistes, qui dans de nombreux cas lui fournissent des algorithmes performants – parfois beaucoup plus que les meilleurs algorithmes déterministes connus.

Le *test de primalité de Miller-Rabin*, présenté en fin de chapitre, constitue par ailleurs un exemple éloquent d'algorithme Monte Carlo basé sur le paradigme de l'*abondance de témoins*. Si un problème de la classe $\mathcal{NP}$ possède de plus la propriété que chaque instance positive admet un *grand nombre* de témoins, de telle sorte que *la probabilité qu'un mot, pris au hasard dans une certaine classe facilement identifiable – par exemple l'ensemble des mots binaire d'une certaine longueur – soit un certificat, est non négligeable*, alors il est facile de construire un algorithme $\mathcal{RP}$ pour ce problème, dans lequel des mots aléatoires "candidats" à être des certificats sont choisis et testés.

4.1 Rappels sur l'arithmétique modulaire

Dans tout ce chapitre, p désigne un nombre premier différent de 2.

4.1.1 Opérations arithmétiques modulaires

Les entiers modulo un nombre premier p sont représentables au moyen de mots binaires dont la longueur est $\ell = \lfloor \log_2(p) \rfloor$. C'est naturellement ce paramètre ℓ qui, pour les algorithmes mettant en œuvre des opérations modulaires, servira de "taille" ; un algorithme en temps polynomial opèrera donc en temps $P(\log n)$, pour un certain polynôme P, sur l'entier n.

La réduction d'un entier modulo p se ramène au calcul du reste d'une division entière : elle est réalisable en temps au plus quadratique sur des entiers de tailles quelconques sous forme binaire. De même, l'addition et la soustraction modulo p sont réalisables en temps linéaire, et la multiplication en temps quadratique est élémentaire (il existe des algorithmes plus rapides).

L'inverse d'un entier non nul a modulo p n'est pas beaucoup plus difficile : l'*algorithme d'Euclide étendu* permet de calculer des *coefficients de Bezout* pour a et p, soit deux entiers α et β tels que l'on ait $a\alpha + p\beta = 1$; en réduisant modulo p, on voit que α est bien l'inverse de a modulo p. Or, l'algorithme d'Euclide étendu opère en un nombre linéaire de divisions entières : sa complexité est donc, sans sophistication particulière, au plus cubique.

La dernière opération fondamentale est l'exponentiation : étant donnés deux entiers x et y et un entier premier p, calculer $x^y \pmod p$. Le petit théorème de Fermat permet de remplacer y par n'importe quel entier qui lui est congru modulo $p-1$ (en d'autres termes, on peut supposer $0 \leq y \leq p-2$), et le calcul se fait en essentiellement $O(\log y)$ opérations qui sont des élévations au carré ou des produits modulo p ; la complexité totale est donc en $O(\ell^3)$.

(Toutes ces remarques restent vraies si p n'est pas un nombre premier ; la seule précaution à prendre est que, si p n'est pas premier, il ne suffit pas pour a de ne pas être nul modulo p pour avoir un inverse : encore faut-il qu'il soit premier avec p. De plus, pour l'exponentiation modulo n, les exposants sont en règle générale à prendre modulo $\phi(n)$.)

4.1.2 Résidus quadratiques et racines carrées

Un *résidu quadratique modulo* p est un entier x, compris entre 1 et $p-1$, pour lequel il existe un entier y tel que l'on ait $y^2 = x \pmod p$. L'entier y est alors appelé *racine carrée de* x *modulo* p.

Il convient de noter que 0 n'est *pas* considéré comme un résidu quadratique, alors qu'il est son propre carré, et donc sa propre (unique) racine carrée.

Si y est une racine carrée modulo p de x, $-y$ en est une autre (y et $-y$ sont distincts modulo p puisque p est impair, sauf bien sûr si $x = 0 \pmod p$)). Par ailleurs, il est facile de vérifier qu'un entier ne peut pas avoir plus de deux racines carrées modulo p (sinon, le polynôme X^2-x aurait plus de deux racines dans le corps $\mathbb{Z}/p\mathbb{Z}$; or, la propriété selon laquelle un polynôme de degré k ne peut jamais avoir plus de k racines est vraie dans un corps commutatif quelconque). Par conséquent, les $p-1$ entiers compris entre 1 et $p-1$ sont les racines carrées d'exactement $\frac{p-1}{2}$ résidus quadratiques différents : *exactement la moitié des entiers non nuls modulo* p *sont des résidus quadratiques.*

Par ailleurs, les résidus quadratiques modulo un entier premier ont une caractérisation fort pratique :

Proposition 4.1 *Soit* p *un entier premier impair, et* $x \in [[1, p-1]]$.

Si x *est un résidu quadratique modulo* p, *alors* $x^{(p-1)/2} = 1 \mod p$. *Si* x *n'est pas un résidu quadratique modulo* p, *alors* $x^{(p-1)/2} = -1 \pmod p$.

Preuve: Si x est résidu, alors $x = y^2 \mod p$. On a donc, d'après le petit théorème de Fermat, $x^{(p-1)/2} = y^{p-1} = 1 \mod p$.

Par ailleurs, que x soit ou non résidu, on a $(x^{(p-1)/2})^2 = x^{p-1} = 1 \mod p$, donc $x^{(p-1)/2}$ a pour carré (modulo p) 1 ; il s'agit donc de l'une des deux racines carrées de 1, qui sont 1 et -1. Il nous suffit donc, pour compléter la preuve de la proposition, de justifier que, si x n'est pas résidu, alors $x^{(p-1)/2}$ n'est pas égal à 1.

Or, dans $\mathbb{Z}_p$ comme dans n'importe quel autre corps, une équation polynômiale ne saurait avoir plus de solution que son degré. Nous connaissons déjà $(p-1)/2$ racines à l'équation $x^{(p-1)/2} - 1 = 0$ (les résidus quadratiques), donc aucun non-résidu ne peut en être solution. □

En particulier, il est relativement facile de *tester* explicitement si un entier est ou non un résidu quadratique modulo un nombre premier donné. De plus, si 1 est toujours un résidu quadratique, -1 n'en est un que si $p = 1 \pmod 4$ (c'est-à-dire, selon la parité de $(p-1)/2$).

Par ailleurs, cette proposition a une conséquence immédiate intéressante :

Proposition 4.2 *Soit p un nombre premier impair, et soient x et y deux entiers non nuls modulo p.*

Alors xy est un résidu quadratique modulo p, si et seulement si, soit x et y sont tous deux des résidus quadratiques, soit ni x ni y ne sont des résidus quadratiques.

4.1.3 Répartition des nombres premiers

S'il est généralement bien connu (et facile à démontrer) qu'il existe une infinité de nombres premiers, la question de leur *répartition* est moins simple à régler. En première approximation, la question qui se pose est de savoir *combien*, en fonction de n, il existe de nombres premiers inférieurs à n. En ce domaine, le théorème fondamental est appelé *théorème des nombres premiers* :

Théorème 4.3 (Théorème des nombres premiers) *Soit $\pi(x)$, le nombre d'entiers premiers inférieurs ou égaux à x.*

$$\lim_{x\to+\infty} \frac{\pi(x)}{x/\ln x} = 1$$

En termes de proportions : la proportion des nombres premiers parmi les entiers inférieurs à x, se comporte comme $1/\ln x$ lorsque x tend vers $+\infty$.

Exercice 4.1 *Soit $\pi'(x)$, le nombre d'entiers premiers compris entre x et $2x$. Montrer que l'on a, lorsque x tend vers $+\infty$,*

$$\pi(x) \sim \pi'(x)$$

Généraliser, en donnant un équivalent du nombre d'entiers premiers compris entre x et λx, pour $\lambda > 1$ (fixe) quelconque et x tendant vers l'infini.

Cela se traduit grosso modo par une remarque quelque peu surprenante : *un entier de N chiffres binaires aléatoires a une probabilité d'environ $1.44/N$ d'être premier* (ici, 1.44 représente $1/\ln 2$). Cela peut sembler faible si N est de l'ordre de 1000, mais a tout de même pour conséquence qu'il est relativement facile de fabriquer de grands nombres premiers : parmi un millier de nombres de 1000 chiffres (binaires) chacun, il n'est pas surprenant qu'un au moins soit premier.

Exercice 4.2 *Un numéro de téléphone portable est typiquement composé de 10 chiffres décimaux, les deux premiers étant 06. Évaluer la probabilité qu'une personne se voie attribuer un numéro de téléphone qui soit un nombre premier. Évaluer la probabilité que, dans un groupe de 25 personnes dont chacune possède son propre téléphone portable, aucune d'entre elles n'ait un numéro premier.*

4.2 Calcul de racines carrées modulaires

Le problème du calcul des racines carrées modulaires est le suivant : étant donnés un nombre premier p, différent de 2, et un résidu quadratique modulo p, x, trouver l'une des deux racines carrées de x modulo p. Comme les deux racines sont forcément opposées l'une de l'autre, trouver l'une d'entre elles ou trouver les deux est équivalent.

Il se trouve que, suivant la valeur de p modulo 4, les solutions déterministes connues varient du tout au tout. Si $p = 3 \pmod 4$, alors le petit théorème de Fermat fournit une solution immédiate. Soit en effet x un résidu quadratique modulo $p = 4k + 3$, et soient y et $p - y$ ses deux racines carrées modulo p. Si l'on applique le petit théorème de Fermat à y, il vient $y^{4k+3} = y \pmod p$, soit

$$y^{4k+4} = y^2 = x \pmod p$$

Or, par hypothèse, $y^2 = x \pmod p$, et on a donc $x^{2k+2} = x \pmod p$. Or, $2k+2$ est pair : nous avons donc un nombre que nous pouvons calculer explicitement et dont le carré vaut x, à savoir, x^{k+1}.

Ce beau calcul s'effondre malheureusement dans le cas où $p = 1 \pmod 4$. En effet, on peut alors écrire $x^{2k+1} = x \pmod 4$, mais, l'exposant étant maintenant impair, le problème de la racine carrée n'en est pas simplifié.

L'algorithme dont nous allons maintenant voir le principe est, à la base, un algorithme Monte Carlo qui, avec probabilité au moins 1/2, trouve une racine carrée modulo p du résidu quadratique fourni en entrée ; et ce, en temps polynomial. Il est alors élémentaire de le transformer en un algorithme Las Vegas à temps moyen polynomial.

Calcul formel modulaire avec $\sqrt{x}$

Soit x un résidu quadratique modulo p, et soit $\sqrt{x}$ l'une, fixée, de ses racines carrées modulaires ; nous n'avons pas forcément besoin de préciser laquelle, mais nous pouvons, par exemple, spécifier que $\sqrt{x}$ désigne *la* racine carrée modulo p de x qui se trouve entre 1 et $(p-1)/2$ (les deux racines carrées sont opposées l'une de l'autre, donc de la forme y et $p-y$, donc exactement une se trouve entre 1 et $(p-1)/2$).

Si a, b, c et d sont des entiers modulo p quelconques, il est facile, même sans connaître la valeur de $\sqrt{x}$, de calculer e et f tels que l'on ait

$$(a + b\sqrt{x})(c + d\sqrt{x}) = e + f\sqrt{x}(\mod p)$$

(en développant, il vient tout simplement $e = ac + xbd \pmod p$ et $f = ad + bc \pmod p$). En adaptant le principe de l'exponentiation modulaire par calculs de carrés et de produits, on obtient clairement, pour tout a, en temps polynomial, les constantes c et d telles que

$$(a + \sqrt{x})^{\frac{p-1}{2}} = c + d\sqrt{x} \pmod p \tag{4.1}$$

Exercice 4.3 *Soient p un nombre premier impair, x un résidu quadratique modulo p, et $a \in [[1, p-1]]$ un entier non nul modulo p.*

Montrer que si le calcul formel de $(a + \sqrt{x})^{(p-1)/2}$ décrit ci-dessus donne $c + d\sqrt{x}$, alors le même calcul pour $(a - \sqrt{x})^{(p-1)/2}$ donne $c - d\sqrt{x}$.

L'algorithme

L'algorithme proposé est très simple à décrire ; c'est, comme souvent, la preuve de ses performances qui est un peu complexe.

Entrée : un entier premier p et un résidu quadratique x modulo p
Sortie : une racine carrée de x modulo p, ou `ECHEC`

1. Si $p = 3 \pmod 4$, retourner $x^{(p+1)/4} \pmod p$
2. Choisir un entier aléatoire $a \in [[1, p-1]]$, uniformément.
3. Si $a^2 = x \pmod p$, retourner a.
4. Calculer formellement c et d tels que $(a + \sqrt{x})^{(p-1)/2} = c + d\sqrt{x} \pmod p$.
5. Si $c \neq 0 \pmod p$, retourner `ECHEC`.
6. Retourner l'inverse de d modulo p.

Algo. 4.1: Calcul de racine carrée modulo p

Analyse de l'algorithme

Il est clair, d'après les remarques faites auparavant, que l'algorithme 4.1 a une complexité en temps raisonnable ($O(\ell^3)$ si les produits modulaires sont effectués en $O(\ell^2)$). Il reste à justifier, d'une part, que les valeurs retournées (autres que `ECHEC`) sont effectivement des racines carrées, et, d'autre part, que la valeur `ECHEC` n'est pas retournée avec trop grande probabilité.

Si l'entier a choisi n'est pas une racine carrée de x, alors $(a + \sqrt{x})^{(p-1)/2}$ vaut 1 ou -1 modulo p. Par conséquent, si $c = 0$, l'inverse de d est soit $\sqrt{x}$, soit $-\sqrt{x}$ – dans les deux cas, c'est effectivement une des deux racines carrées de x modulo p, et le résultat est correct.

La question qui reste en suspens est donc de déterminer quelle est la probabilité que l'algorithme retourne effectivement une valeur différente de `ECHEC`. En d'autres termes, il nous faut évaluer le nombre de valeurs de a pour lesquelles on trouve $c = 0$.

Lemme 4.4 *On trouve $c = 0$ si et seulement si $a^2 - x$ n'est pas un résidu quadratique modulo p.*

Preuve: Puisque x est un résidu quadratique, on a $a^2 - x = (a + \sqrt{x})(a - \sqrt{x})$.

Il est clair (par récurrence sur l'exposant ; voir l'exercice 4.3) que, si $(a + \sqrt{x})^{(p-1)/2} = c + d\sqrt{x}$, alors $(a - \sqrt{x})^{(p-1)/2} = c - d\sqrt{x}$.

En tant que puissances $(p-1)/2$-èmes d'entiers non nuls, $c + d\sqrt{x}$ et $c - d\sqrt{x}$ valent chacun 1 ou -1, et leur produit vaut 1 si $a^2 - x$ est un résidu quadratique, et -1 sinon. Par conséquent, $c + d\sqrt{x} = c - d\sqrt{x}$ si $a^2 - x$ est un résidu quadratique, et $c + d\sqrt{x} = -c + d\sqrt{x}$ sinon.

Dans le premier cas ($a^2 - x$ résidu quadratique), la différence (qui est nulle) vaut $2d\sqrt{x}$, ce qui implique que l'on a $d = 0$ (et, par conséquent, c vaut 1 ou -1). Dans le second cas ($a^2 - x$ non résidu quadratique), la somme (qui est nulle) vaut $2c$, donc $c = 0$, et $d\sqrt{x}$ vaut 1 ou -1, ce qui implique que $1/d$ a bien x pour carré.

Au total, on obtient : si $a^2 - x$ est un résidu quadratique, alors $d = 0$ et $c^2 = 1$; sinon, $c = 0$ est $(d\sqrt{x})^2 = 1$. □

Il nous reste donc à prouver que, lorsque a est choisi au hasard, la probabilité que $a^2 - x$ ne soit pas un résidu quadratique est assez élevée. Pour cela, nous démontrons d'abord un autre lemme.

Lemme 4.5 *Soit f la fonction définie sur $[[1, p-1]] - \{\sqrt{x}, p - \sqrt{x}\}$ par*

$$(a + \sqrt{x})f(a) = (a - \sqrt{x}) \pmod p$$

La fonction f établit une bijection entre $[[1, p-1]] - \{\sqrt{x}, p - \sqrt{x}\}$ et $[[2, p-2]]$.

Preuve:

Posons, de manière équivalente, $f(a) = (a - \sqrt{x}).(a + \sqrt{x})^{-1} \pmod{p}$. L'hypothèse $a \neq -\sqrt{x}$ implique que $f(a)$ est bien défini, et $a \neq \sqrt{x}$, que l'on a bien $f(a) \neq 0$.

De plus, on ne peut avoir $f(a) = 1$ (cela nécessiterait $\sqrt{x} = 0$), ni $f(a) = -1$ (qui impliquerait $a = 0$).

Enfin, f est injective sur $[[1, p-1]] - \{\sqrt{x}, p - \sqrt{x}\}$: en effet, $f(a) = f(b)$ implique (par différence) $(b - a)f(a) = (b - a) \pmod{p}$, ce qui, avec $f(a) \neq 1$, implique $a = b$.

Au total, la fonction f est injective sur $[[1, p-1]] - \{\sqrt{x}, p - \sqrt{x}\}$, et prend ses valeurs dans $[[2, p-2]]$. Les deux ensembles ayant le même cardinal $p - 3$, f est bijective. □

Revenons à l'algorithme (4.1). Les valeurs de a qui le font échouer sont exactement celles pour lesquelles $a^2 - x$ est un résidu quadratique, c'est-à-dire celles pour lesquelles $a + \sqrt{x}$ et $a - \sqrt{x}$ sont, soit tous deux des résidus, soit tous deux des non-résidus. Par définition de f, ce sont donc les valeurs de a telles que $f(a)$ soit un résidu quadratique.

Il y a donc autant de "mauvaises" valeurs de a, que de résidus quadratiques entre 2 et $p-2$. Or, il y a exactement $(p-1)/2$ résidus quadratiques au total, parmi lesquels on trouve 1 (qui est toujours un résidu quadratique) et $p - 1 = -1 \pmod{p}$ (dont la puissance $(p-1)/2$-ème est 1, puisque $(p-1)/2$ est pair). Par conséquent, il n'y a que $(p-5)/2$ résidus quadratiques modulo p entre 2 et $p - 2$.

Nous avons donc démontré le théorème suivant :

Théorème 4.6 *Lorsque $p = 1 \pmod{4}$, la probabilité que l'algorithme 4.1 échoue à trouver une racine carrée modulo p est*

$$\frac{p-5}{2(p-1)} < \frac{1}{2}.$$

En d'autres termes, notre algorithme est un *algorithme Monte Carlo d'erreur* 1/2 *et à temps polynomial* pour le calcul de racines carrées modulaires.

Exercice 4.4 *Donner un algorithme Las Vegas à temps moyen polynomial pour le même problème.*

Il convient de noter ici que l'on ne connaît, à l'heure actuelle, *aucun* algorithme déterministe pour le calcul des racine carrées modulo unnombre premier quelconque, et dont il soit *prouvé* qu'il s'exécute toujours en temps polynomial. En particulier, on ne sait pas prouver (même si cela semble extrêmement vraisemblable, du moins aux spécialistes) que prendre de manière déterministe, dans notre algorithme précédent, l'entier a égal successivement à $1, 2, 3, \ldots$, donnerait un algorithme à temps polynomial dans le cas le pire.

4.3 Le test de primalité de Miller-Rabin

Notre second exemple d'utilisation de l'algorithmique probabiliste en théorie des nombres, est celui des tests de primalité. Jusqu'à l'annonce, au cours de l'été 2002, d'un algorithme (déterministe) en temps polynomial permettant, sans aucune hypothèse supplémentaire[1], de

[1] Il existait déjà des algorithmes déterministes pour décider de la primalité, mais, pour affirmer que leur complexité est polynomiale, on a apparemment besoin de l'*hypothèse de Riemann généralisée*, une conjecture qui résiste depuis plus d'un siècle.

décider si un entier est ou non premier, on ne savait pas si la primalité était décidable en temps polynomial, alors que le *test de Miller-Rabin*, que nous allons présenter, et qui est connu depuis 1976, est un algorithme de type Monte Carlo beaucoup plus simple. Celui-ci reste d'ailleurs, en pratique, tout à fait compétitif par rapport aux algorithmes déterministes connus.

Le test de Miller-Rabin, comme d'autres qui l'ont précédé et qui sont moins efficaces, est, au sens strict, un *test de non-primalité de type Monte Carlo à erreur unilatérale* : essentiellement, il cherche une *preuve de non primalité* qui lui permettrait de conclure que l'entier présenté n'est pas premier, et, s'il n'en trouve pas, le déclare premier. Pour prouver que le test est effectivement un algorithme Monte Carlo, il faut donc prouver que les fameuses "preuves de non primalité" sont fréquentes.

Le point de départ du test de Miller-Rabin est une généralisation du petit théorème de Fermat, et du fait que les racines carrées de 1 modulo p sont toujours 1 et -1. Soit p un entier premier impair, et $a \in [[1, p-1]]$ un entier non nul modulo p. Puisque $a^{p-1} = 1 \pmod p$, et que $p-1$ est pair, il s'ensuit que $a^{p-1} \pmod p$ vaut soit 1, soit -1. S'il vaut 1 et que $(p-1)/2$ est pair, alors, à son tour, $a^{(p-1)/4} \pmod p$ vaut soit 1, soit -1. On peut continuer ainsi à diviser l'exposant par 2, soit jusqu'à ce qu'il devienne impair, soit jusqu'à trouver une valeur de -1. Nous avons ainsi la proposition suivante :

Proposition 4.7 *Soit p un nombre premier impair, et soit r le plus grand facteur impair de $p-1$ (c'est-à-dire que l'on a $p-1 = 2^s r$ pour un certain entier s). Soit également $a \in [[1, p-1]]$ un entier non nul modulo p.*

Alors soit $a^r = 1 \pmod p$ (ce qui implique évidemment $a^{2^j r} = 1 \pmod p$ pour $0 \leq j \leq s-1$), soit $a^{2^j r} = -1 \pmod p$ pour un entier j, $0 \leq j \leq s-1$.

Inversement, si un entier a est tel que $a^r \neq 1 \pmod p$ et $a^{2^j r} \neq -1 \pmod p$ pour $0 \leq j \leq s-1$, c'est forcément que p n'est pas premier. C'est l'idée qui se trouve derrière la définition d'un *témoin fort de non-primalité*[2] :

Définition 4.8 *Soient n un entier impair, et $a \in [[1, n-1]]$. Soient s et r, avec r impair, tels que $n-1 = 2^s r$.*

On dit que a est un témoin fort de non-primalité *pour n, si $a^r \neq 1 \pmod n$ et, pour $0 \leq j \leq s-1$, $a^{2^j r} \neq -1 \pmod n$.*

La proposition 4.7 affirme simplement que, si n est premier, il n'a aucun témoin fort de non-primalité. Le cœur du test de Miller-Rabin est la proposition suivante, que nous ne démontrerons pas :

Proposition 4.9 *Tout entier n, impair et non premier, a au moins $3(n-1)/4$ témoins fort de non-primalité.*

En d'autres termes : si n est un entier impair composé, choisir un entier $a \in [[1, n-1]]$, aléatoire et uniforme, uniforme, a une probabilité d'au moins $3/4$ de donner un témoin fort de non-primalité.

De la discussion qui précède (et du fait que les calculs sont clairement exécutables en temps polynomial), il ressort le théorème suivant :

[2]On parle de témoin *fort* car il existe une notion, plus faible, de "témoin de non-primalité" ; cette notion est la base d'un test, dit *test de Fermat*, qui, contrairement à Miller-Rabin, ne constitue pas un *bon* test de non-primalité.

Entrée : Un entier impair n
Sortie : `Premier` ou `Composé`

1. $r \leftarrow n$, $s \leftarrow 0$
2. Tant que r est pair, faire $r \leftarrow r/2$, $s \leftarrow s+1$
3. Choisir un entier a, aléatoire uniforme dans $[[1, n-1]]$.
4. $x \leftarrow a^r \pmod n$
5. Si $x = 1$ ou $x = n-1$, retourner `Premier`
6. $j \leftarrow 1$
7. Tant que $j < s$ et $y \neq n-1$, faire
   (a) $y \leftarrow y^2 \pmod n$, $j \leftarrow j+1$
   (b) Si $y = 1$, retourner `Composé`
8. Retourner `Premier`

ALGO. 4.2: Test de Miller-Rabin

Théorème 4.10 *L'algorithme 4.2 est un test de non-primalité de type Monte Carlo, à erreur unilatérale et à temps polynomial, et de probabilité d'erreur* $1/4$ *; à savoir,*
- *si n est premier, alors* $\Pr(MR(n) = \texttt{Premier}) = 1$ *;*
- *si n n'est pas premier, alors* $\Pr(MR(n) = \texttt{Composé}) \geq 3/4$.

L'erreur étant unilatérale, il est clair qu'il est possible de faire passer la probabilité d'erreur (c'est-à-dire la probabilité de déclarer premier un entier composé) à $(1/4)^k$ en répétant k fois l'algorithme : il suffit d'un seul résultat `Composé` pour être certain que n est effectivement composé.

Exercice 4.5 *Évaluer la complexité du test de Miller-Rabin. On considèrera que la complexité d'un produit modulo n est $O(\log^2 n)$.*

Évaluer, en fonction de ϵ et de $\log n$, la complexité d'un test basé sur Miller-Rabin qui garantisse une probabilité d'erreur inférieure à ϵ.

Exercice 4.6 *Étant donné un entier ℓ, on se propose de construire un "générateur aléatoire de nombres premiers" de taille (longueur binaire) ℓ, selon le modèle suivant : on engendre des chaînes aléatoires binaires de longueur ℓ, auxquelles on fait passer le test de Miller-Rabin un nombre k de fois, jusqu'à en obtenir une qui passe avec succès le test.*

Montrer que tout nombre premier compris entre $2^{\ell-1}$ et $2^\ell - 1$ a la même probabilité d'être retourné par cet algorithme. Évaluer (majorer) également la probabilité qu'un tel algorithme retourne un nombre non premier. On utilisera le théorème des nombres premiers, et on s'autorisera, pour simplifier les calculs, à faire comme s'il donnait exactement le nombre de nombres premiers compris dans l'intervalle considéré (c'est-à-dire que la proportion de nombres premiers parmi les x entiers de 1 à x est exactement de $1/ln(x)$). À l'aide de cette analyse, préciser le choix de k si l'on dispose d'un paramètre supplémentaire ϵ, et que l'on désire garantir que la probabilité de retourner un nombre qui n'est pas premier soit au plus de ϵ.

Note : dans la pratique, il est assez rare pour un entier composé n de n'avoir que de l'ordre de $3n/4$ témoins de non-primalité ; le plus souvent, donc, la probabilité de détecter une non-

primalité est très supérieure à ce qui est garanti. En particulier, le plus petit nombre composé dont aucun des 8 plus petits nombres premiers[3] (2, 3, 5, 7, 11, 13, 17 et 19) ne soit un témoin fort de non-primalité, est $N_0 = 341550071728321$ (15 chiffres décimaux). Cette rareté des “entiers fortement pseudo-premiers” est telle que certaines implantations (fautives, au sens strict !) du test de Miller-Rabin n'utilisent pas des entiers aléatoires, mais la suite des nombres premiers.

Exercice 4.7 *Expliquer pourquoi de telles variantes déterministes du test de Miller-Rabin ne fournissent pas un test de non-primalité de type Monte Carlo.*

[3] Il existe une assez bonne raison de tester surtout des nombres premiers comme candidats à être des témoins : si ni a_1, ni a_2 ne sont des témoins forts de non-primalité pour n, il est assez vraisemblable que $a_1 a_2$ n'en sera pas un non plus.

Chapitre 5

Structures de données probabilistes

Dans ce chapitre, nous nous intéressons à la conception de structures de données probabilistes, typiquement plus efficaces que leurs homologues déterministes.

5.1 Opérations génériques

Les structures de données que nous envisagerons sont censées représenter des ensembles d'objets, chaque objet x se voyant associer une *clé* $k(x)$ dont nous supposerons qu'elle appartient à un ensemble totalement ordonné. Les objets eux-mêmes n'intervenant que comme données auxiliaires attachées aux clés, nous identifierons chacun à sa clé. Une structure de données doit supporter un certain nombre d'opérations parmi les suivantes :

- CREER(S) : crée un nouvel ensemble S.
- INSERER(k, S) : ajoute la clé k à l'ensemble S.
- RETIRER(k, S) : retire l'objet de clé k de l'ensemble S.
- TROUVER(k, S) : retourne l'objet de clé k de l'ensemble S, s'il existe.
- UNION(S_1, S_2) : remplace deux ensembles S_1 et S_2, par $S = S_1 \cup S_2$.
- SEPARER(k, S) : remplace S par deux ensembles S_1 et S_2, avec $S_1 = \{x \in S : k(x) < k\}$ et $S_2 = \{x \in S : k(x) \geq k\}$.
- PREMIER(k, S) : retourne l'objet de plus petite clé de S.
- SUIVANT(k, S) : retourne l'objet dont la clé est la plus petite des clés strictement supérieures à k.

Classiquement, on choisit pour ce genre de problème une variante de l'idée d'*arbre binaire de recherche*. Les arbres binaires de recherche permettent de réaliser les opérations en temps proportionnel à leur hauteur, et les différentes variantes d'arbres équilibrés permettent d'atteindre des temps en $\Theta(\log n)$ pour les opérations élémentaires TROUVER, INSERER et RETIRER.

5.2 Principe d'une structure de données probabiliste

Dans ce chapitre, nous décrivons des exemples de *structures de données probabilistes*. Le "problème à résoudre" est donc de maintenir, face à une séquence *a priori* quelconque de requêtes, une structure permettant d'effectuer la prochaine opération rapidement.

Dans une structure de données probabiliste, la représentation en mémoire de la structure n'est pas fonction déterministe de la séquence d'opérations qui ont servi à sa création – on

y incorpore une part d'aléatoire, de manière à "contrer" les tentatives de dégradation des performances d'un hypothétique adversaire chargé de choisir ladite séquence.

5.3 Listes à sauts aléatoires

5.3.1 Listes à sauts

Une *liste à sauts*[1] est une variante de la liste chaînée triée. Nous nous intéressons essentiellement aux opérations TROUVER, INSERER, PREMIER, et SUIVANT.

Considérons un ensemble S de clés appartenant à un ensemble totalement ordonné. On forme, à partir de cet ensemble, une liste L, constituée des éléments de S, auxquels on adjoint deux éléments fictifs $-\infty$ et $+\infty$.

Une *liste à sauts* d'ensemble de base S est la donnée d'une suite de listes $L_1, \ldots, L_r$ vérifiant les conditions suivantes :
- $L_1 = L$;
- $L_r = (-\infty, +\infty)$;
- pour tout $i < r$, $L_{i+1} \subset L_i$.

Il faut concevoir la liste à sauts comme la superposition de $r + 1$ listes, de plus en plus clairsemées à mesure que l'on considère des listes de plus haut niveau.

Pour chaque élément $x \in S$, on note $\ell(x)$ le "niveau" de x, défini par

$$\ell(x) = \max\{i : x \in L_i\}.$$

La représentation en mémoire d'une liste à sauts est indiquée Figure 5.1. Chaque cellule de chacune des listes pointe à la fois vers la cellule suivante de la même liste, et sur la cellule correspondant au même élément de la liste précédente ; l'accès global à la structure complète se fait par le début de la dernière liste (L_r).

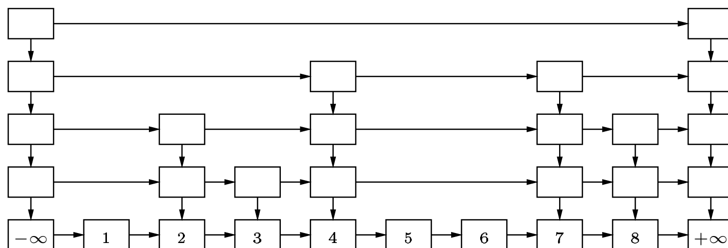

FIG. 5.1 – Un exemple de liste à sauts

Plutôt que de considérer chaque cellule de la liste L_i comme représentant une clé donnée, il peut être préférable de la voir comme représentant un *intervalle* de clés, à savoir, si $L_i = (x_1^{(i)}, \ldots, x_k^{(i)})$, la j-ème cellule de L_i représente l'intervalle $[x_j^{(i)}, x_{j+1}^{(i)}[$. Il est alors naturel de considérer la liste complète comme un *arbre* (non binaire) d'intervalles : la racine correspond à l'intervalle $[-\infty, +\infty[$, et un sommet (intervalle) I, de hauteur i, est un fils d'un sommet (intervalle) J de profondeur $i + 1$, lorsque $I \subset J$. Dans cette représentation, on note $f(I)$ le nombre de fils de l'intervalle I ; répétons qu'a priori, $f(I)$ n'est pas borné : rien n'empêche un

[1]En Anglais : *skip list*.

intervalle de hauteur $i+1$ d'être, à hauteur i, découpé en un grand nombre d'intervalles. La figure 5.2 montre la liste de la figure 5.1, mise sous forme d'arbre.

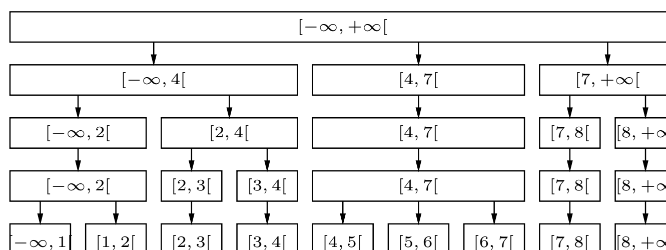

FIG. 5.2 – Représentation sous forme d'arbre d'intervalles

Exercice 5.1 *Soient k une clé et S un ensemble représenté sous forme de liste à sauts. Soit, pour $i \leq r$, $I_i(k)$ l'intervalle de hauteur i qui contient k.*

Montrer que l'opération TROUVER(k, S) *peut être réalisée en temps*

$$O\left(\sum_{i=1}^{r}(1 + f(I_i(k)))\right).$$

Qu'en est-il des opérations INSERER(k, S) *et* RETIRER(k, S) *?*

5.3.2 Modèle probabiliste

Pour préciser un modèle probabiliste de listes à sauts, il nous faut dire comment, pour un ensemble S donné, nous allons choisir les éléments qui composent chacun des niveaux de la liste.

Le modèle que nous étudierons est extrêmement simple : pour chaque clé k, son niveau $\ell(k)$, choisi *lors de l'insertion* et qui ne sera *jamais modifié* tant que la clé se trouvera dans la liste, est déterminé par une variable aléatoire *géométrique* de paramètre $1/2$. Une autre façon de voir les choses, parfaitement équivalente, est de dire que, le niveau L_i étant déterminé, chaque élément de L_i (autre que les sentinelles $-\infty$ et $+\infty$) a une probabilité $1/2$ d'apparaître encore au niveau $i+1$, indépendemment des autres. La hauteur totale r est, par définition, la première à laquelle aucun des éléments de S n'apparaît plus.

La structure représentant un ensemble S, obtenue à partir d'une suite quelconque d'insertions et de suppressions, ne dépend donc que des choix aléatoires faits lors de la dernière insertion de chaque clé se trouvant encore dans l'ensemble S : les insertions et suppressions d'autres clés, ainsi que l'ordre des insertions, ne jouent aucun rôle. Il s'agit là d'une caractéristique extrêmement utile pour l'*analyse* des performances de la structure de données considérée : on peut en effet se contenter d'examiner la structure elle-même à un moment donné (c'est-à-dire un objet probabiliste *statique*), et non la *façon* dont on l'a obtenu (c'est-à-dire le processus *dynamique* qui a abouti à l'objet tel qu'il est actuellement).

Exercice 5.2 *Montrer que l'espérance de l'espace mémoire requis pour stocker une liste à sauts aléatoire de n objets, est $\Theta(n)$.*

Proposition 5.1 *Soit R_n le nombre de niveaux d'une liste à sauts aléatoire de n objets. Alors $\mathbb{E}(R_n) = O(\log n)$, et il existe une constante K, indépendante de n, telle que*

$$\mathbb{P}(R_n > K \log n) = o(1)$$

(ce que l'on résume généralement par : $R = O(\log n)$ avec forte probabilité).

Preuve: Commençons par la seconde affirmation. La hauteur totale de la liste est $R = 1 + \max_{x \in S} \ell(x)$, et les différents $\ell(x)$ sont des variables géométriques de paramètre $1/2$, indépendantes.

On a, pour tout $x \in S$, $\mathbb{P}(\ell(x) > m) = (1/2)^m$, ce qui entraine $\mathbb{P}(R \geq m) \leq n.(1/2)^m$. En posant $m = \alpha \log_2 n$, il vient

$$\mathbb{P}(R > \alpha \log_2 n) \leq \frac{1}{n^{\alpha-1}},$$

ce qui montre que toute constante $K > 1$ convient.

Passons maintenant à la majoration de l'*espérance* de R, qui découle du calcul précédent. Remarquons que, pour une variable aléatoire entière positive N, on a $\mathbb{E}(N) = \sum_{m \geq 0} \mathbb{P}(N > m)$. Pour $m \leq \log_2(n)$, nous majorons $\mathbb{P}(R > m)$ par 1 ; et, pour $m > \log_2(n)$, nous utilisons l'inégalité $\mathbb{P}(R > m) \leq 1/2^{m-\log_2(n)}$.

Au total, nous avons (en supposant par exemple $n > 2$)

$$\begin{aligned}
\mathbb{E}(R) &= \sum_{k \geq 0} \mathbb{P}(R > k) \\
&\leq \log_2(n) + \sum_{m > \log_2(n)} \frac{1}{2^{m-\log_2(n)}} \\
&\leq \log_2(n) + \sum_{m \geq 0} \frac{1}{2^m} \\
&\leq \log_2(n) + 2
\end{aligned}$$

□

Nous allons maintenant étudier la complexité moyenne de l'opération TROUVER dans une liste à sauts aléatoire. Comme vu précédemment, la complexité de TROUVER(k, S) est déterminée par

$$\sum_{i=1}^{R} (1 + f(I_i(k))) = R + \sum_{i=1}^{R} f(I_i(k))$$

Proposition 5.2 *Soit I un intervalle d'une liste à sauts aléatoire. Alors on a*

$$\mathbb{E}(f(I)) \leq 2,$$

sauf si $I = [-\infty, +\infty[$, auquel cas on a

$$\mathbb{E}(f(I)) \leq 3.$$

Preuve: Soit I l'intervalle considéré, et $i+1$ son niveau. Soit y_0 la borne inférieure de I, et soient $y_0, y_1, \ldots, y_k = +\infty$ les éléments de L_i (le niveau *en-dessous* de I) au moins égaux à y_0.

Il est facile de voir que le nombre de fils $f(I)$ de l'intervalle I, est égal au plus petit indice j tel que $y_j \in L_{i+1}$. Or, chacun de ces éléments étant de niveau au moins i par construction, il a probabilité $1/2$ d'être de niveau strictement supérieur à i et donc d'appartenir à L_{i+1} – mis à part $y_k = +\infty$, qui est dans L_{i+1} par construction. Par conséquent, $f(I)$ est distribué comme une variable géométrique de paramètre $1/2$, *conditionnée* à être au plus égale à k ; l'espérance d'une telle variable aléatoire est, quelle que soit la valeur de k, majorée par 2 (l'espérance d'une variable géométrique de paramètre $1/2$ non conditionnée).

Cette analyse doit être très légèrement modifiée dans le cas où $I = [-\infty, +\infty[$ (c'est-à-dire si I est l'unique intervalle de niveau R). En effet, puisque R est le premier niveau à ne contenir aucun élément de S, on aura par construction $f(I) \geq 2$. Dans ce cas, $f(I)$ est toujours dominé stochastiquement[2] par une variable géométrique de paramètre $1/2$, conditionnée à être au moins égale à 2 ; l'espérance d'une telle variable aléatoire est de 3. □

Proposition 5.3 *Soit S une liste à sauts aléatoire de taille n, et k une clé.*
Alors le coût moyen de l'opération TROUVER(k, S) *est* $O(\log n)$.

Preuve: Nous savons déjà que la hauteur de l'arbre est, en moyenne et avec forte probabilité, $O(\log n)$, et que chaque sommet du chemin de recherche de k a un nombre espéré de fils $O(1)$.

Le résultat prévu ne découle pas seulement du résultat sur la hauteur moyenne ; nous avons réellement besoin du résultat de forte probabilité (la complexité s'exprime comme somme d'un nombre aléatoire de variables aléatoires dont nous n'avons qu'une majoration des espérances ; on ne peut pas déduire de telles hypothèses que l'espérance de la somme soit, par exemple, majorée par le produit des espérances).

Soit Y_i le nombre d'intervalles visités à hauteur i de l'arbre ; on sait que l'on a $E(Y_i) \leq 2$, et $Y_i \leq n$. Soit également $Y = \sum_i Y_i$ le nombre total d'intervalles visités au cours de la recherche.

Nous avons

$$\begin{aligned}
\mathbb{E}(Y) &= \sum_{i=0}^{+\infty} \mathbb{E}(Y_i) \\
&= \sum_{i=0}^{+\infty} \mathbb{E}(Y_i | Y_i \geq 1)\mathbb{P}(Y_i \geq 1) \\
&= \sum_{i=0}^{+\infty} \mathbb{E}(Y_i | Y_i \geq 1)\mathbb{P}(R \geq i) \\
&\leq \sum_{i=0}^{+\infty} 2\mathbb{P}(R \geq i) \\
&\leq 2\log_2(n) + 2 \sum_{i > \log_2(n)} \frac{n}{2^i} \\
&\leq 2\log_2(n) + 4
\end{aligned}$$

[2] On dit qu'une variable aléatoire X en domine stochastiquement une autre Y, si l'on a $\mathbb{P}(X \geq x) \leq \mathbb{P}(Y \geq x)$ pour tout x. Il s'agit d'une relation entre les *lois* de X et de Y, et qui est équivalente à l'affirmation suivante : il existe, dans un certain espace de probabilités, deux variables X' et Y', de mêmes lois respectives que X et Y, et telles que l'on ait partout $X \geq Y$. Voilà, vous pouvez oublier cette notion quelque peu exotique...

(Pour affirmer que l'on a $\mathbb{E}(Y_i|Y_i \geq 1) \leq 2$, il suffit de remarquer que Y_i est au plus égal au nombre de fils du dernier intervalle examiné à hauteur $i+1$; c'est donc, dans tous les cas, une variable aléatoire stochastiquement dominée par une variable géométrique de paramètre 1/2.) □

Exercice 5.3 *Obtenir une majoration plus fine de la complexité moyenne de* TROUVER(x, S)*, en calculant, pour deux indices i et j, la probabilité que l'intervalle $[x_i, x_j[$ soit visité à hauteur ℓ de l'arbre. Cette probabilité peut s'exprimer exactement en fonction de i, j, ℓ et de l'indice k défini par*

$$k = \max\{i : x_i \leq x\}$$

qui représente la position où s'insèrerait un élément de clé x.

Il serait possible de pousser un peu plus l'analyse pour obtenir un peu plus : *en moyenne et avec forte probabilité, l'opération* TROUVER *sur une liste à sauts aléatoire de taille n a un coût $O(\log n)$.*

Le même type d'analyse peut être appliqué aux opérations INSERER et RETIRER, avec le même type de résultat : ces deux opérations, en moyenne et avec forte probabilité, ont un coût $O(\log n)$.

5.4 Treaps

Il est difficile de donner une traduction française satisfaisante du mot-valise *treap*, formé à partir de *tree* et de *heap* ; un treap est la conjonction d'un arbre binaire de recherche et d'un tas, et fournit également une solution efficace au problème de structuration de données.

5.4.1 Structure de treap

Rappelons rapidement quelques définitions :

arbre binaire de recherche un arbre binaire dont chaque sommet interne u est étiqueté par une clé $k(u)$ (les clés, distinctes, sont issues d'un ensemble totalement ordonné), est un *arbre binaire de recherche* si, pour chaque sommet interne v,
- si u est un sommet du sous-arbre gauche issu de v, $k(u) < k(v)$;
- su w est un sommet du sous-arbre droit issu de v, $k(v) < k(w)$.

Le parcours symétrique des sommets internes, fournit les clés dans l'ordre croissant.

tas binaire Un arbre binaire, dont chaque sommet interne u est étiqueté par une clé $k(u)$, est un *tas binaire* si, pour chaque sommet u autre que la racine, de père v, on a $k(v) < k(u)$[3].

Un *treap* est un arbre binaire dont chaque sommet interne maintient deux informations *a priori* distinctes : une *clé* $k(u)$, et une *priorité* $p(u)$; les clés forment un arbre binaire de recherche, et les priorités, un tas binaire.

Proposition 5.4 *Soit $S \subset K \times P$ un ensemble fini de couples (k_i, p_i), où les ensembles K et P (ensembles des clés et des priorités, respectivement) sont totalement ordonnés, et tels que tous les k_i et tous les p_i soient distincts. Il existe un unique treap dont les sommets sont étiquetés par les éléments de S.*

[3]On ne conserve pas, de la définition classique du tas en tant que structure de donnée, la condition d'équilibre qui veut, en particulier, que toutes les feuilles de l'arbre soit de même profondeur à 1 près.

Preuve: Démontrons par récurrence sur $n = |S|$ la propriété. Pour $n = 0$, la propriété est triviale : il existe un unique arbre binaire complet sans sommet interne.

Supposons donc la propriété (d'existence et d'unicité) vraie pour tout $m \leq n$, et soit S un ensemble de taille $n + 1$. Soit (k, p) l'élément de S de priorité minimale, et soient $S_< = \{(k', p') : k' < k\}$ et $S_> = \{(k', p') : k' > k\}$; ces deux ensembles sont de taille au plus n, et admettent donc chacun un unique treap.

Il est immédiat de vérifier que l'arbre binaire dont la racine est étiquetée (k, p), dont le sous-arbre gauche est le treap de $S_<$ et le sous-arbre droit est le treap de $S_>$, vérifie les propriétés d'arbre binaire de recherche (pour les clés) et de tas binaire (pour les priorités) ; c'est donc bien un treap pour S. De plus, la propriété de tas binaire implique que la racine *doit* avoir la plus petite des priorités : par conséquent, dans tout treap pour S, la racine est étiquetée (k, p). La propriété d'arbre binaire de recherche implique alors que les clés du sous-arbre gauche ne peuvent être que celles qui sont inférieures à k, et celles du sous-arbre droit, celles qui sont supérieures à k ; par conséquent, le sous-arbre gauche doit être un treap pour $S_<$, et le sous-arbre droit, un treap pour $S_>$, ce qui implique que le treap pour S est également unique. □

L'opération classique TROUVER (portant sur les clés) est réalisée sur un treap de la même manière que sur un arbre binaire de recherche.

L'opération INSERER(k, p, S) est implémentée en commençant par effectuer un TROUVER(k, S), puis en ajoutant le nouveau sommet interne à la place de la feuille où la recherche se termine par un échec. Il est bien entendu possible que la condition de tas soit alors violée ; cette condition sera réparée en effectuant une rotation entre le nouveau sommet et son père, et en recommençant en remontant vers la racine tant que le sommet remonté a une priorité inférieure à celle de son père.

Enfin, RETIRER(k, S) se fait en effectuant une rotation entre le sommet de clé k et celui de ses fils qui a la plus petite priorité ; cette opération est répétée jusqu'à ce que le sommet de clé k ait deux feuilles comme fils, auquel cas on peut simplement éliminer le sommet et le remplacer par une feuille.

Clairement, chacune des opérations de recherche et de mise à jour s'implémente en temps $O(h)$, où h désigne la hauteur totale de l'arbre binaire.

5.4.2 Modèle probabiliste de treaps

Pour obtenir un modèle probabiliste de treaps, il nous faut préciser comment, à partir d'un ensemble de clés, choisir des couples de clés et de priorités.

Une solution simple et satisfaisante est la suivante : la priorité associée à un objet quelconque est une variable aléatoire uniforme sur $[0, 1]$, indépendante des autres (et de la clé associée !). Cette priorité est fixée lors de l'insertion, et n'est pas modifiée tant que l'objet reste dans l'arbre.

À vrai dire, il n'est pas essentiel que les priorités suivent une loi uniforme ; la seule chose qui nous intéresse ici est que n priorités soient toutes distinctes avec probabilité 1, et qu'elles soient ordonnées suivant une permutation aléatoire uniforme de n éléments. En ce sens, n'importe quelle loi diffuse conviendrait ; la loi uniforme est seulement la plus simple !

Grâce à la propriété d'unicité, le treap obtenu après une suite d'insertions et de destructions ne dépend que des tirages aléatoires des priorités des objets qui *restent* dans l'arbre, et non pas, ni de l'ordre de ces insertions, ni des éventuels objets qui ont été retirés pour ne

pas être réinsérés. En particulier, le treap obtenu est *identique* à celui que l'on aurait obtenu en insérant les clés dans l'ordre croissant des priorités qui leur ont été assignées. Dans cette situation, aucune rotation n'a lieu : l'insertion se fait exactement comme dans un arbre binaire de recherche. Cette propriété est essentiellement la même que celle que nous avons déjà mentionnées pour les skip lists, et présente les mêmes avantages.

En fait, parce que les priorités sont choisies indépendemment des clés, le treap obtenu, si l'on ne retient que les clés, n'est rien d'autre qu'un arbre binaire aléatoire (arbre binaire obtenu en insérant les clés dans un ordre aléatoire, toutes les permutations étant équiprobables). En particulier, toute grandeur probabiliste calculée sur les arbres binaires aléatoires, peut être transposée à notre modèle de treaps.

5.4.3 Hauteur des arbres binaires de recherche aléatoires

Dans le cas qui nous intéresse, le coût d'une insertion est majoré par la hauteur de l'arbre, qui est donc la grandeur que nous devons analyser. Essentiellement, nous allons démontrer le théorème suivant :

Théorème 5.5 *La hauteur h_n d'un arbre binaire de recherche aléatoire de n sommets, vérifie*

$$\mathbb{E}(h_n) = O(\log n)$$

Il existe plusieurs méthodes différentes pour prouver ce théorème ; celle qui est présentée ici est essentiellement celle proposée dans [2].

Le lemme 5.6 nous fournit une description explicite, en fonction de l'ordre d'insertion des clés, des ancêtres d'une clé dans un arbre binaire de recherche.

Lemme 5.6 *Soit σ une permutation des n entiers $1, \dots, n$, et soit T l'arbre binaire de recherche obtenu en insérant les entiers dans l'ordre $\sigma_1, \dots, \sigma_n$.*

Alors, pour $i < j$, σ_i est un ancêtre de σ_j si et seulement si

$$\sigma_i = \min \{k_\ell : 1 \leq \ell \leq i \text{ et } \sigma_\ell > \sigma_j\}$$

ou

$$\sigma_i = \max \{k_\ell : 1 \leq \ell \leq i \text{ et } \sigma_\ell < \sigma_j\}$$

(en d'autres termes, la i-ème clé insérée devient un ancêtre de la j-ème si et seulement si aucune clé comprise entre elles n'a été insérée avant la i-ème)

Preuve: Soit T_i l'arbre obtenu en n'insérant que les clés $\sigma_1, \dots, \sigma_i$. Si σ_i est un ancêtre de σ_j, alors, lors de son insertion dans T_{j-1}, la clé σ_j suit le chemin de la racine à σ_i. Ce chemin est exactement celui qu'il aurait suivi dans T_i ; par conséquent, si σ_j était inséré dans T_i, il se retrouverait comme fils de σ_i. Or, dans un arbre binaire de recherche, une feuille est soit la plus grande clé qui soit inférieure à son père (si c'est une feuille gauche), soit la plus petite qui soit supérieure à son père (si c'est une feuille droite).

Supposons maintenant que σ_i soit la plus petite clé, parmi les i premières, qui soit supérieure à σ_j (l'autre cas, où σ_i est la plus grande qui soit inférieure à σ_j, se traite de la même manière). Alors, lors de l'insertion de σ_j, il est indiscernable de σ_i tant qu'on ne le compare qu'à des clés insérées avant σ_i - ce qui est le cas de toutes les clés qui se trouvent sur le chemin de la racine à σ_i. Par conséquent, σ_j suivra le même chemin que σ_i lors de son insertion, et sera donc inséré comme descendant de σ_i. □

Pour une clé $s = \sigma_j$, partitionnons l'ensemble de ses ancêtres en "ancêtres gauches" (ceux qui sont supérieurs à s, dont s appartient au sous-arbre gauche) et "ancêtres droits" (inférieurs à s) :

$$\begin{aligned} L_s &= \{\sigma_i : i < j, \sigma_i \leq \sigma_j, \sigma_i = \max\{\sigma_\ell : \ell \leq i, \sigma_\ell \leq \sigma_j\}\} \\ R_s &= \{\sigma_i : i < j, \sigma_i \geq \sigma_j, \sigma_i = \min\{\sigma_\ell : \ell \leq i, \sigma_\ell \geq \sigma_j\}\} \end{aligned}$$

Pour $r < s$, $r \in R_s$ si et seulement si la clé r est inséré avant chacune des clés $r+1, \ldots, s$; l'ordre d'insertion étant aléatoire, cet événement se produit avec probabilité $\frac{1}{s-r+1}$. De même, pour $r > s$, $r \in L_s$ si et seulement si r est inséré avant chacune des clés $s, s+1, \ldots, r-1$, soit avec probabilité $\frac{1}{r-s+1}$.

En conséquence, on a

$$\begin{aligned} \mathbb{E}(|R_s|) &= H_s - 1 \\ \mathbb{E}(|L_s|) &= H_{n-s+1} - 1 \end{aligned}$$

On en déduit que le nombre $A_s = |R_s| + |L_s|$ d'ancêtres de s, vérifie

$$\mathbb{E}(A_s) = H_s + H_{n-s+1} - 2 \leq 2\ln(n) + O(1).$$

Toutefois, ce simple calcul d'espérance ne permet pas de conclure quant à l'espérance de la hauteur d'un arbre binaire de recherche : en effet, la hauteur de l'arbre est le *maximum* des A_s, lesquels ne sont pas du tout indépendants. Pour affirmer une inégalité de la forme $\mathbb{P}(h \leq k) < \epsilon$, il nous faudrait une inégalité du type $\mathbb{P}(A_s \leq k) < \frac{\epsilon}{n}$; or, l'inégalité de Markov (la seule applicable si l'on n'a *que* l'espérance) ne nous donnerait une telle inégalité que pour des valeurs de k bien supérieures à $\ln n$.

Pour obtenir une inégalité plus précise, nous devons nous intéresser à la *loi de probabilité* de $|R_s|$ (l'analyse est essentiellement la même pour L_s). Clairement, R_s ne dépend que de l'ordre d'insertion des clés $1, \ldots, s$ (les clés supérieures n'interviennent pas). La variable aléatoire $1 + |R_s|$ (le $+1$ provient de s, qui n'est pas pris en compte dans R_s) est le nombre de *records* (éléments plus grands que tous ceux qui les ont précédés) dans la suite d'insertion des éléments 1 à s. Or, la probabilité pour la k-ème insertion de constituer un record est exactement $1/k$ (ce qui va nous redonner une espérance de H_s pour $1 + |R_s|$; toutefois, ces événements sont maintenant indépendants (le fait que le k-ème élément de la suite soit un record dépend uniquement de la comparaison aux éléments précédents, indépendemment de leur ordre d'insertion). Par conséquent, on a :

Proposition 5.7 *Le nombre $|R_s|$ d'ancêtres droits de s, est distribué comme la somme de $s-1$ variables de Bernoulli indépendantes, de probabilités $1/i$ pour i variant de 2 à s.*

De même, le nombre $|L_s|$ d'ancêtres droit de s, est distribué comme la somme de $n-s$ variables de Bernoulli indépendantes, de probabilités $1/i$ pour i variant de 2 à $n-s+1$.

Maintenant que nous disposons de la loi, peut-être un simple calcul de variance suffira-t-il à conclure ? L'exercice suivant permet de montrer qu'il n'en est rien.

Exercice 5.4 *On reprend les notations précédentes : R_s est le nombre d'ancêtres droits de la clé s dans un arbre binaire de recherche aléatoire de taille n, dont les clés sont $1, \ldots, n$.*

1. *Exprimer la* variance *de $|R_s|$ sous la forme d'une somme dépendant de s.*

2. *Montrer que l'on a* $\mathbf{Var}(|R_s|) = H_s - O(1)$ *(le* $O(1)$ *est ici uniforme : il doit exister une majoration indépendante aussi bien de* s *que de* n*).*
3. *Appliquer l'inégalité de Tchebycheff à la variable aléatoire* $|R_s|$*, pour obtenir une valeur* k *telle que* $\mathbb{P}(|R_s| > k) < \frac{1}{n}$*. On pourra supposer* $n/4 \leq s \leq 3n/4$*.*

Puisque un simple calcul de variance ne nous permet pas de conclure, il convient de se rabattre sur une inégalité portant, de manière générale, sur les sommes de variables aléatoires indépendantes : l'*inégalité de Chernoff* (voir l'annexe A, inégalité A.9).

Il est clair, d'après la discussion précédente, que, pour des constantes C_1 et C_2 appropriées, et n assez grand, on a

$$\ln(n) - C_1 \leq \mathbb{E}(A_s) \leq 2\ln(n) + C_2.$$

En conséquence, on a, d'après l'inégalité de Chernoff,

$$\mathbb{P}\left(A_s > (1+\delta)(2\ln(n) + C_2)\right) \leq \left(\frac{e^\delta}{(1+\delta)^{1+\delta}}\right)^{\ln(n)-C_1}$$

Si l'on prend δ assez grand pour que l'on ait

$$\frac{e^\delta}{(1+\delta)^{1+\delta}} < \frac{1}{2}$$

(par exemple $\delta = 1.4$), on obtient

$$\mathbb{P}\left(A_s > (1+\delta)(2\ln(n) + C_2)\right) < \frac{C'}{n^2}.$$

Par conséquent, la probabilité que l'une au moins des n variables aléatoires A_s soit plus grand que $(1+\delta)(2\ln(n) + C_2)$ (c'est-à dire que la hauteur h soit plus grand que cette même valeur), est majorée par C'/n. Or, h est, de manière certaine, majorée par n. On a donc, en posant $f(n) = (1+\delta)(2\ln(n) + C_2)$,

$$\begin{aligned} \mathbb{E}(h) &\leq f(n)\mathbb{P}(h \leq f(n)) + n\mathbb{P}(h > f(n)) \\ &\leq f(n) + C' \end{aligned}$$

On a donc bien $\mathbb{E}(h) = O(\log n)$, ce qui prouve le théorème et permet d'affirmer que, en moyenne, chaque mise à jour ou requête sur un treap aura un coût $O(\log n)$.

5.5 Tables de hachage

Une *table de hachage* (*hashtable* en Anglais) est une structure de données permettant d'effectuer les opérations TROUVER, INSERER et RETIRER en temps essentiellement constant ; en contrepartie, l'espace requis est plus important, et il est coûteux de parcourir l'ensemble des objets contenus dans la table.

L'ensemble des clés sera noté K, avec $m = |K|$. On pourra par exemple supposer que l'on a $K = \{0, 1, \ldots, m-1\}$.

5.5.1 Tableaux

Une idée élémentaire pour effectuer recherches et mises à jour en temps constant, consiste à stocker une table indexée par les clés (en d'autres termes, un tableau de taille m) ; pour chaque clé x, la cellule d'indice x contiendra soit NIL (s'il n'y a pas d'élément de clé k dans l'ensemble S), soit u si u est l'élément (supposé unique) de clé x dans S.

L'inconvénient majeur de cette approche est évident : il se situe au niveau de l'espace mémoire requis. Dans les cas typiques, m (la taille de l'espace des clés) est de beaucoup supérieur à la taille des ensembles que l'on désire représenter, et, en tout état de cause, il est impraticable de stocker un tableau de taille m (penser à $m = 2^{32}$, ce qui correspond au cas où les clés sont des entiers sur une machine 32 bits).

5.5.2 Tables et fonctions de hachage

Une *table de hachage* est constituée des éléments suivants :
- un ensemble K de clés, de taille m ;
- un entier $n < m$; N désignera l'ensemble $\{0, 1, \ldots, n-1\}$;
- un tableau T, de taille n, indexé par les éléments de N ;
- une *fonction de hachage* h, qui à chaque clé $k \in K$ associe un élément de N.

L'idée de base est fort simple : l'objet de clé x, s'il existe, est stocké dans la cellule $T[h(x)]$.

Définition 5.8 *Pour une fonction de hachage h, on dira qu'il se produit une* collision *entre deux clés x et y si $h(x) = h(y)$; on dira qu'une telle collision a lieu en $h(x)$.*

Définition 5.9 *Une fonction de hachage est dite* parfaite *pour un ensemble $S \subset K$ si elle ne provoque aucune collision entre les éléments de S.*

Il est clair que, pour tout ensemble S de taille au plus n, il existe une fonction de hachage parfaite (pas forcément facile à calculer) pour S. En revanche, il ne peut pas exister de fonction de hachage qui soit parfaite pour *chaque* sous-ensemble de taille $k \leq n$ d'un ensemble de taille supérieure à n (tout simplement parce qu'il y aura forcément au moins deux éléments qui auront la même image, et donc, la fonction de hachage ne sera pas parfaite pour un ensemble contenant deux tels éléments).

La solution adoptée est classique : choisir une fonction de hachage aléatoirement parmi un certain ensemble de fonctions, de telle sorte qu'elle soit, de manière très probable, "presque" parfaite – c'est-à-dire qu'elle provoque peu de collisions.

5.5.3 Familles 2-universelles de fonctions de hachage

Définition 5.10 *Un ensemble H de fonctions de hachage est une* famille 2-universelle *si, quels que soient les éléments x et y de K, on a, si h est une fonction prise aléatoirement dans H (selon la distribution uniforme),*

$$\mathbb{P}(h(x) = h(y)) \leq \frac{1}{n}.$$

(Dans l'inégalité ci-dessus, la probabilité porte sur le choix de la fonction h dans H ; cette inégalité doit être vérifiée simultanément pour *tous* les couples d'éléments de K.)

La valeur de $1/n$ comme limite de probabilité de collision, correspond à ce que l'on obtient comme probabilité de collision si les images de toutes les clés sont choisies uniformément dans

N, indépendemment les unes des autres ; de ce point de vue, les fonctions de hachage d'une famille 2-universelle se comportent un peu comme des fonctions aléatoires.

Par ailleurs, cette valeur de $1/n$ est difficile à battre, et est asymptotiquement optimale ; on peut montrer que, pour toute famille H, il existe deux clés x et y telles que la probabilité de collision entre x et y soit au moins de $1/n - 1/m$.

Existe-t-il des familles 2-universelles de fonctions de hachage ? La réponse est oui ; par exemple, l'ensemble de *toutes* les fonctions de K dans N constitue une telle famille. Toutefois, pour implémenter des tables de hachages efficaces, nous avons besoin de familles qui vérifient des conditions supplémentaires : les familles doivent être de petite taille (la table devra stocker suffisamment d'informations pour décrire complètement un élément de la famille), et il doit être possible, à partir de la description “compacte” d'un élément de la famille, de calculer la valeur de la fonction de hachage pour une clé quelconque.

Si l'on dispose d'une famille 2-universelle acceptable H, l'implémentation de tables de hachage devient claire : à la création de la table, un élément $h \in H$ est choisi aléatoirement, et sa description est stockée en annexe de la table. C'est cette fonction de hachage qui est ensuite utiliée pour traiter toutes les requêtes et mises à jour de la table.

Il est nécessaire de prévoir un moyen de traiter les collisions, même si le système a été conçu pour qu'elles soient peu probables. La solution la plus simple consiste à stocker toutes les clés ayant une même valeur de hachage sous la forme d'une liste chaînée ; chaque recherche aura alors comme coût, en plus d'une évaluation de la fonction de hachage, la longueur de la liste stockée dans la cellule correspondante de la table.

Si l'on fait l'hypothèse que l'évaluation de la fonction de hachage se fait en temps $O(1)$, on montre facilement que, pour *toute* séquence de r opérations à partir d'une table vide, telle qu'à aucun moment la table ne contienne plus de s éléments (par exemple, si s est le nombre total d'insertions parmi les opérations), l'espérance du coût total est majorée par $r(1 + \frac{s}{n})$. Pour peu que n soit plus grand que s, cette espérance est inférieure à $2r$. De plus, chaque opération, individuellement, a un coût dont l'espérance est $O(1 + s/n)$.

5.5.4 Construction d'une “petite” famille 2-universelle

Nous allons maintenant présenter un exemple de famille de fonctions de hachage qui vérifie les conditions énoncées précédemment : un ensemble de fonctions de raisonnablement faible cardinalité, facile à décrire et à évaluer, et qui possède la propriété de 2-universalité.

Soit p un nombre premier au moins égal à m (il en existe au moins un entre m et $2m$, et n'importe lequel fera l'affaire). Nous allons travailler dans le corps $\mathbb{Z}_p$ des entiers modulo p.

Nous notons g la fonction de $\mathbb{Z}_p$ dans N définie par $g(x) = x \mod n$.

Pour $a, b \in \mathbb{Z}_p$, soient $f_{a,b}$ la fonction (affine !) de $\mathbb{Z}_p$ dans lui-même, et $h_{a,b}$ la fonction de hachage, définies par

$$\begin{aligned} f_{a,b}(x) &= ax + b \pmod p, \\ h_{a,b}(x) &= g(f_{a,b}(x)). \end{aligned}$$

Théorème 5.11 *La famille $H = \{h_{a,b} : a \in \mathbb{Z}_p^*, b \in \mathbb{Z}_p\}$, est une famille 2-universelle de fonctions de hachage.*

Preuve: Soient x et y deux clés distinctes, et soit $h = h_{a,b} \in H$. Notons $r = f_{a,b}(x)$ et $s = f_{a,b}(y)$. Tout d'abord, remarquons que, dès lors que $a \neq 0$, $f_{a,b}$ est une *bijection* sur $\mathbb{Z}_p$;

par conséquent, $r \neq s$, et il ne peut se produire une collision entre x et y que si r et s sont deux éléments distincts de $\mathbb{Z}_p$ tels que $g(r) = g(s)$.

Renversons les rôles : fixons r et s comme étant deux éléments distincts quelconques de $\mathbb{Z}_p$. Il existe un unique couple (a, b) tel que $f_{a,b}(x) = r$ et $f_{a,b}(y) = s$ (il s'agit d'un système linéaire de 2 équations à 2 inconnues, dans $\mathbb{Z}_p$:

$$\begin{aligned} ax + b &= r \\ ay + b &= s \end{aligned}$$

Le déterminant de ce système est $x - y$, qui est inversible dans $\mathbb{Z}_p$, ce qui implique existence et unicité de la solution.)

Nous savons donc que, pour $x \neq y$ quelconques, le nombre de fonctions de hachage de H qui provoquent une collision est exactement le nombre de couples (r, s), avec $r \neq s$, tels que $g(r) = g(s)$. Remarquons que ce nombre ne dépend pas de x ni de y : la probabilité de collision entre deux clés est la même pour toutes les paires de clés. Nous noterons α ce nombre de couples.

Pour $z \in N$, soit $A_z = \{r \in \mathbb{Z} : g(r) = z\}$. Par construction de g, on a $\lfloor p/n \rfloor \leq |A_z| \leq \lceil p/n \rceil$. Or, le nombre de fonctions de hachage de H qui provoquent en z une collision entre x et y est, d'après le calcul précédent, $|A_z|(|A_z| - 1)$.

Posons $a = \lfloor p/n \rfloor$; soit enfin k le nombre de $z \in N$ pour lesquels on a $|A_z| = a$, et donc $n - k$ le nombre de z pour lesquels $|A_z| = a + 1 = \lceil p/n \rceil$. Les différents A_z forment une partition de $\mathbb{Z}_p$, et par conséquent on a

$$p = k.a + (n - k)(a + 1).$$

Nous avons donc

$$\begin{aligned} \alpha &= ka(a-1) + (n-k)(a+1)a \\ &\leq (ka + (n-k)(a+1))\, a \\ &\leq p \left\lceil \frac{p}{n} - 1 \right\rceil \\ &\leq p \left(\frac{p+n-1}{n} - 1 \right) \\ &\leq \frac{p(p-1)}{n} \end{aligned}$$

Or, le nombre total de fonctions de hachage dans H est exactement $p(p-1)$; par conséquent, la probabilité pour x et y d'entrer en collision (qui est de $\alpha/|H|$) est bien inférieure ou égale à $1/n$. □

Cette famille H répond bien à notre problème : pour peu que l'on choisisse $p = O(m)$, le choix de a et b coûtera au plus $2 \log p = O(\log m)$ bits aléatoires, et le stockage de a et b ne nécessitera pas plus de mémoire. L'évaluation de la fonction de hachage, elle aussi, ne nécessitera que deux opérations arithmétiques modulaires sur des entiers de taille $O(\log m)$ (donc, du même ordre de grandeur que la taille des clés elles-mêmes) et un calcul de réduction modulo n.

5.5.5 Extension aux familles k-universelles

Essentiellement, la propriété qui nous permet d'obtenir une famille 2-universelle est la suivante : si f est une fonction choisie aléatoirement, uniformément, parmi les p^2 fonctions affines (polynômes de degré au plus 1) de $\mathbb{Z}_p$ dans lui-même, alors quels que soient les éléments x et y, distincts, de $\mathbb{Z}_p$, $f(x)$ et $f(y)$ sont deux éléments aléatoires, uniformes, indépendants, de $\mathbb{Z}_p$. En d'autres termes, les p variables aléatoires $(f(x))_{x\in\mathbb{Z}_p}$ sont des variables aléatoires uniformes sur $\mathbb{Z}_p$, **deux à deux** indépendantes.

Il est important de bien comprendre que l'indépendance deux à deux de ces variables aléatoires, n'implique en aucun cas que ces variables soient indépendantes. En particulier, sous les hypothèses ci-dessus, pour tout triplet x, y, z, on peut toujours calculer $f(z)$ en connaissant $f(x)$ et $f(y)$, pour peu que x et y soient différents :

$$f(z) = f(x) + (z - x)(y - x)^{-1}(f(y) - f(x))$$

Exercice 5.5 *Soient E et F deux ensembles finis, de cardinalités respectives n et m. On suppose qu'un ensemble H de fonctions de E dans F est tel que, si $f \in H$ est choisie aléatoirement uniformément, le vecteur de variables aléatoires $(f(x))_{x\in E}$ forme un vecteur de n variables aléatoires indépendantes, distribuées selon la loi uniforme sur F.*

Montrer que H est l'ensemble de toutes les fonctions de E dans F (et donc, de cardinalité m^n).

L'exercice précédent implique que, pour qu'un ensemble de fonctions H de $\mathbb{Z}_p$ dans lui-même fournisse, pour les images de k éléments, k variables aléatoires (uniformes dans $\mathbb{Z}_p$), il faut qu'il soit de cardinalité au moins p^k.

Il se trouve que l'**ensemble des fonctions polynômes de degré au plus** $k-1$ sur $\mathbb{Z}_p$ (ensemble qui a cette cardinalité minimale) a cette propriété, et ce, pour **tout** ensemble de k éléments de $\mathbb{Z}_p$.

Théorème 5.12 *Soit p un entier premier, et k un entier positif quelconque. Alors, quel que soit l'ensemble de k éléments $x_1, \ldots, x_k$ (distincts) de $\mathbb{Z}_p$, et quels que soient $y_1, \ldots, y_k$ éléments (non nécessairement distincts) de $\mathbb{Z}_p$, il existe, parmi les p^k polynômes de degré strictement inférieur à k et à coefficients dans $\mathbb{Z}_p$, un et un seul polynôme f tel que l'on ait $f(x_k) = y_k$.*

Preuve: Si l'on note $a_0, \ldots, a_{k-1}$ les coefficients d'un polynôme dont on cherche à déterminer s'il vérifie les conditions, les équations que doivent vérifier ces coefficients s'écrivent, pour $1 \leq i \leq k$,

$$a_0 + x_i a_1 + x_i^2 a_2 + \cdots + x_i^{k-1} a_{k-1} = y_i$$

On reconnait dans la matrice de ce système linéaire, une matrice de Vandermonde, associée aux valeurs $x_1, \ldots, x_k$. Le déterminant s'écrit donc (au signe près) comme le produit des différences $x_i - x_j$. Les x_i étant tous distincts modulo p, chaque facteur est non nul modulo p. L'entier p étant premier, $\mathbb{Z}_p$ est un corps, et le produit de nombres non nuls modulo p est non nul modulo p. Par conséquent, la matrice du système est de déterminant non nul, donc inversible (pour les sceptiques quant à la validité de tels résultats classiques de l'algèbre linéaire sur $\mathbb{R}$ ou $\mathbb{C}$ dans le cas de corps finis, remarquons par exemple que les formules de Cramer s'appliquent sans problème).

Le système admet donc une solution unique, ce qui prouve le théorème. □

Ce théorème s'interprète de la manière suivante : si un polynôme aléatoire uniforme est choisi parmi ceux à coefficients dans $\mathbb{Z}_p$, de degré inférieur à k (ce qui peut se faire en tirant aléatoirement, indépendamment, chacun de ses k coefficients comme une variable aléatoires uniforme sur $\mathbb{Z}_p$), chaque k-uplet d'éléments distincts de $\mathbb{Z}_p$ a probabilité exactement $1/p^k$ d'avoir pour image, par le polynôme aléatoire choisi, chaque k-uplet de valeurs possibles. En d'autres termes, les p variables aléatoires "valeur du polynôme au point x" (pour $x \in \mathbb{Z}_p$) sont toutes uniformes sur $\mathbb{Z}_p$, **indépendantes k à k**.

Nous avons donc construit un "petit" espace de probabilités (de taille p^k), dans lequel nous disposons de p variables aléatoires k à k indépendantes, de même loi, alors qu'obtenir p variables aléatoires indépendantes demanderait un espace autrement plus grand (de taille p^p).

Un des intérêts d'avoir un grand nombre de variables aléatoires k à k indépendantes, est que tout calcul d'espérance sur une fonction qui se présente comme un *polynôme de degré au plus* k de nos variables aléatoires, donnera automatiquement le même résultat que si les variables étaient réellement indépendantes. Ainsi, si n variables aléatoires sont 2 à 2 indépendantes, la **variance de leur somme** est automatiquement la même que celle de la somme de n variables de mêmes lois respectives qui seraient complètement indépendantes entre elles.

De telles constructions, ou des variantes, sont utiles lorsque l'on désire *dérandomiser* un algorithme probabiliste – un sujet qui sort malheureusement du cadre strict de ce cours.

Annexe A

Rappels de probabilités

Ce chapitre ne se prétend pas un cours complet de probabilités ; il se contente de résumer un certain nombre de définitions et de résultats, élémentaires ou non, de la théorie des probabilités.

A.1 Définitions et notations

A.1.1 Notions de base

Cas des espaces finis ou dénombrables

Rappelons les définitions, fort simples dans le cas d'espaces *finis*[1] ou *dénombrables* :

Espace de probabilités Un espace de probabilités, ou espace probabilisé, est un couple $(\Omega, \mathbb{P})$, où Ω est un ensemble et $\mathbb{P}$ est une fonction, définie sur Ω, à valeurs dans $\mathbb{R}^+$, et telle que

$$\sum_{\omega \in \Omega} \mathbb{P}(\omega) = 1.$$

La fonction $\mathbb{P}$ est appelée fonction (ou mesure) de probabilités, et est étendue additivement à toute partie de Ω.

Événement Un évènement est une partie A de Ω ; sa probabilité est

$$\mathbb{P}(A) = \sum_{\omega \in A} \mathbb{P}(\omega).$$

Variable aléatoire Une variable aléatoire est une fonction[2] définie sur un espace probabilisé.

Ces notions doivent être "interprétées" comme suit : l'espace probabilisé est une description abstraite de tous les résultats possibles d'une expérience non déterministe, chaque élément ω représentant un résultat "élémentaire" possible, qui a une "chance" $\mathbb{P}(\omega)$ de se produire. Un événement n'est alors rien d'autre que la réunion de tous les résultats élémentaires correspondant à une réponse positive à une question portant sur l'expérience, et sa probabilité est, par définition, la somme des probabilités des événements élémentaires qui le composent.

[1] Malheureusement, le cas des espaces finis n'est généralement pas suffisant pour l'étude d'algorithmes probabilistes.

[2] Le terme de variable aléatoire est parfois réservé aux fonctions à valeurs réelles ; nous ne ferons pas de distinction à ce niveau et pourrons désigner ainsi une fonction à valeurs vectorielles, par exemple.

Cas des espaces non dénombrables

Lorsque l'ensemble de base n'est pas dénombrable, les définitions sont compliquées par le besoin de recourir aux notions de *tribu*, de *mesure*, et de *fonction mesurable*. Les complications viennent de ce que, lorsque l'ensemble Ω n'est plus dénombrable, on ne peut plus raisonnablement considérer que *toute* partie de Ω peut se voir affecter une probabilité : il convient de se limiter à une *partie* (une tribu) de l'ensemble des parties de Ω ; ces sous-ensembles (les ensembles *mesurables*) seront les *évènements*, et seules les fonctions *mesurables* auront droit au statut de variables aléatoires.

Le cadre "naturel" de la théorie générale des probabilités est donc celui de l'intégrale de Lebesgue.

Les définitions qui suivent sont incluses essentiellement par souci de rigueur, et pour soulager les scrupules de l'auteur. On trouvera dans n'importe quel ouvrage de probabilités à vocation universitaire, comme par exemple [3], un traitement complet de la théorie élémentaire des probabilités.

Définition A.1 (Tribu) *Soit Ω un ensemble. Une* tribu *(ou σ-algèbre) sur Ω est un ensemble $\mathcal{F}$ de parties de Ω qui vérifie les conditions suivantes :*

- *$\emptyset \in \mathcal{F}$, $\Omega \in \mathcal{F}$.*
- *pour toute partie A de Ω, si $A \in \mathcal{F}$, alors $\overline{A} \in \mathcal{F}$;*
- *si $(A_i)_{i \in I}$ est une famille finie ou dénombrable d'éléments de $\mathcal{F}$, alors $A = \cup_{i \in I} A_i$ est également dans $\mathcal{F}$.*

Le couple $(\Omega, \mathcal{F})$ est appelé espace mesurable*. Un élément de $\mathcal{F}$ est appelé* partie mesurable *de Ω.*

Autrement dit, une tribu est un ensemble de parties qui contient l'ensemble complet, et qui est stable à la fois par prise de complément et par union (ou intersection) dénombrable.

Une autre remarque importante est que l'intersection d'un ensemble quelconque (en particulier, potentiellement non dénombrable) de tribus est encore une tribu. Cela permet de parler de *tribu engendrée* par un ensemble de parties de Ω :

Définition A.2 (Tribu engendrée) *Soit Ω un ensemble non vide, et E un ensemble quelconque de parties de Ω. La* tribu engendrée par *E, notée $\sigma(E)$, est la plus petite tribu (au sens de l'inclusion) qui contienne E ; c'est encore l'intersection de toutes les tribus qui contiennent E.*

L'existence de $\sigma(E)$ est assurée par la remarque précédente, et par la garantie qu'il existe toujours au moins une tribu contenant E, à savoir, la tribu formée de toutes les parties de Ω.

Définition A.3 (Tribu borélienne) *La* tribu borélienne *sur $\mathbb{R}$ est la tribu engendrée sur $\mathbb{R}$ par l'ensemble des intervalles de $\mathbb{R}$.*

La tribu borélienne contient, entre autres, tous les intervalles (ouverts, fermés, ou semi-ouverts), ainsi que leurs unions dénombrables.

La tribu borélienne sert de "tribu par défaut", en ce sens que, chaque fois que l'on parle de variable aléatoire réelle, l'espace mesurable sous-jacent est $\mathbb{R}$ muni de la tribu borélienne.

Définition A.4 (Mesure) *Soit $(\Omega, \mathcal{F})$ un espace mesurable. Une fonction μ, définie sur $\mathcal{F}$ et à valeurs réelles positives ou infinies, est une* mesure positive *si elle vérifie les conditions suivantes :*

- $\mu(\emptyset) = 0$;
- *si* $(A_i)_{i \in I}$ *est une famille finie ou dénombrable de parties mesurables* deux à deux disjointes*, alors*

$$\mu\left(\cup_{i \in I} A_i\right) = \sum_{i \in I} \mu(A_i).$$

Une mesure positive $\mathbb{P}$ *est une* mesure *(ou* loi*)* de probabilité *si on a de plus* $\mathbb{P}(\Omega) = 1$. *Le triplet* $(\Omega, \mathcal{F}, \mathbb{P})$ *est alors appelé* espace de probabilités *ou* espace probabilisé.

Définition A.5 (Fonction mesurable) *Soit* $(E, \mathcal{E})$ *un espace mesurable, et* $(F, \mathcal{T})$ *un espace topologique. Une fonction* $f : E \to F$ *est* mesurable *si* l'image réciproque de toute partie ouverte de F est une partie mesurable de E :

$$\forall B \in \mathcal{T}, \{x \in E, f(x) \in B\} \in \mathcal{E}$$

Définition A.6 (Variable aléatoire) *Une fonction mesurable d'un espace probabilisé vers un espace topologique est appelée* variable aléatoire.

Note : Il arrive que l'on réserve le terme de "variable aléatoire" pour les fonctions mesurables à valeurs *réelles*. Ce n'est pas le point de vue adopté ici.

Définition A.7 (Tribu engendrée par une variable aléatoire) *Soit* Ω *un ensemble non vide quelconque, et* X *une fonction définie sur* Ω *et à valeurs dans un espace* $(F, \mathcal{T})$.

La tribu engendrée par X, *notée* $\sigma(X)$, *est la plus petite tribu sur* Ω *rendant* X *mesurable. C'est aussi la tribu engendrée par les ensembles*

$$X^{-1}(B) = \{\omega \in \Omega : X(\omega) \in B\}$$

où B *parcourt, soit la topologie* $\mathcal{T}$ *de* F, *soit simplement une partie qui l'engendre.*

Dans le cas, fréquent, où Ω est déjà muni d'une tribu $\mathcal{F}$ et où X est déjà définie comme une variable aléatoire, $\sigma(X)$ apparaît naturellement comme une sous-tribu de $\mathcal{F}$. C'est, en quelque sorte, l'ensemble minimal d'évènements permettant de définir la variable aléatoire X.

Égalité en lois

Deux variables aléatoires X et Y, prenant leurs valeurs dans un même espace F, sont *égales en lois*, si elles ont la même loi de probabilité, c'est-à-dire si, pour tout ensemble mesurable B de F, on a l'égalité

$$\mathbb{P}(X \in B) = \mathbb{P}(Y \in B) \tag{A.1}$$

Il est important de noter que X et Y n'ont nullement besoin d'être définies sur le même espace de probabilités ; l'égalité en lois ne "parle" que des lois de probabilités. C'est ce qui permet de parler de "variable aléatoire de loi μ".

L'égalité en lois implique, en particulier, l'égalité de tous les moments définis.

Une condition suffisante (et nécessaire, puisqu'il s'agit d'une sous-condition) pour l'égalité en loi de deux variables, est que (A.1) soit vraie pour tout B appartenant à un ensemble d'évènements qui *engendre* la tribu de F. Une telle propriété est particulièrement intéressante quand F est un espace produit (c'est-à-dire si X et Y sont des *suites* de variables aléatoires) ; elle s'énonce alors ainsi :

Théorème A.8 *Soient $X = (X_n)_{n>0}$ et $Y = (Y_n)_{n>0}$, deux suites de variables aléatoires à valeurs dans un espace F.*

X et Y sont égales en lois, si et seulement si, pour tout $k > 0$ et tout k-uplet $(B_1, \ldots, B_k)$ d'ensembles mesurables dans F, on a l'égalité

$$\mathbb{P}(X_1 \in B_1, \ldots, X_k \in B_k) = \mathbb{P}(Y_1 \in B_1, \ldots, Y_k \in B_k).$$

Espérance et variance

Lorsque X est une variable aléatoire, on note souvent $(X = a)$ l'évènement

$$\{\omega \in \Omega : \; X(\omega) = a\} = X^{-1}(\{a\}).$$

De même, si X est une variable aléatoire réelle (à valeurs dans $\mathbb{R}$), et si I est un intervalle réel quelconque, on note

$$(X \in I) = X^{-1}(I) = \{\omega \in \Omega : \; X(\omega) \in I\}.$$

L'*espérance* d'une variable aléatoire réelle X, notée $\mathbb{E}(X)$, est définie par

$$\begin{aligned} \mathbb{E}(X) &= \sum_a a\mathbb{P}(X = a) \\ &= \sum_{\omega \in \Omega} X(\omega)\mathbb{P}(\omega). \end{aligned}$$

Cette définition n'est valable que dans le cas où X prend ses valeurs dans un ensemble fini ou dénombrable ; dans le cas contraire, la somme est remplacée par une intégrale de Lebesgue.

Une propriété extrêmement utile, et qui ne nécessite aucune hypothèse particulière sur les variables aléatoires[3], est la *linéarité de l'espérance* :

Théorème A.9 *Soient X et Y deux variables aléatoires définies sur le même espace probabilisé, et soient λ et μ deux réels quelconques. Alors la variable aléatoire $Z = \lambda X + \mu Y$ (définie par : pour tout $\omega \in \Omega$, $Z(\omega) = \lambda X(\omega) + \mu Y(\omega)$) vérifie*

$$\mathbb{E}(Z) = \lambda\mathbb{E}(X) + \mu\mathbb{E}(Y). \tag{A.2}$$

La *variance* d'une variable aléatoire est une quantification de sa propension à prendre des valeurs éloignées de son espérance : si cette espérance est $\mu = \mathbb{E}(X)$, on pose

$$\begin{aligned} \mathbf{Var}(X) &= \mathbb{E}\left((X - \mu)^2\right) \\ &= \mathbb{E}(X^2) - \mu^2 \end{aligned}$$

(l'égalité des deux expressions résulte du fait que la variable aléatoire $X - \mu$ est d'espérance nulle).

La variance est souvent notée σ^2, la lettre σ (qui en est la racine carrée positive) représentant alors l'*écart-type* de la variable aléatoire.

Note : Toute variable aléatoire réelle n'a pas forcément une espérance finie (la série, ou l'intégrale, peut diverger), et une variable aléatoire qui a une espérance peut ne pas avoir de variance. Les propriétés énoncées doivent être entendues sous réserve d'existence. Plus précisément, l'existence pour le “membre gauche” entraine l'existence pour le “membre droit”.

[3]Autre que l'existence des espérances, dans le cas d'espaces infinis.

Espérance et variance ne sont que les deux premiers termes d'une suite de *moments* d'une variable aléatoire. On définit, pour tout entier $k > 0$ (et sous réserve d'existence), le *moment d'ordre k* (respectivement, *moment centré d'ordre k* par

$$\begin{aligned} \mu_k(X) &= \mathbb{E}(X^k) \\ \mu_k^c(X) &= \mathbb{E}((X - \mu_1(X))^k) \end{aligned}$$

On reconnaît dans $\mu_1(X)$ l'espérance, et dans $\mu_2^c(X)$, la variance.

A.1.2 Indépendance

Intuitivement, deux évènements ou deux variables aléatoires sont *indépendants* si une information obtenue sur l'un, ne modifie pas la connaissance que l'on a de l'autre. Cela se traduit formellement de la manière suivante.

Définition A.10 *Deux évènements A et B sont dits* indépendants, *si et seulement si*

$$\mathbb{P}(A \cap B) = \mathbb{P}(A)\mathbb{P}(B).$$

On notera qu'un évènement de probabilité 0 ou 1, est indépendant de tout autre évènement.

Définition A.11 *Deux variables aléatoires réelles X et Y sont dites* indépendantes, *si et seulement si les deux affirmations suivantes, qui sont équivalentes, sont vraies :*

1. *pour tous intervalles I et J, les évènements $(X \in I)$ et $(Y \in J)$ sont indépendants ;*
2. *pour toutes valeurs x et y, on a*

$$\mathbb{P}\left(X \leq x \text{ et } Y \leq y\right) = \mathbb{P}(X \leq x)\mathbb{P}(Y \leq y).$$

(Dans le cadre le plus général possible, où les variables aléatoires peuvent être à valeurs dans des espaces quelconques, l'indépendance s'exprime par le fait que les *tribus* engendrées par les deux variables aléatoires sont indépendantes, c'est-à-dire que tout ensemble de l'une est indépendant de tout ensemble de l'autre)

Dans le cas particulier où les variables aléatoires ne peuvent prendre qu'un nombre fini ou dénombrable de valeurs (le cas typique étant celui de variables aléatoires à valeurs *entières*), la définition est également équivalente à la suivante :

Définition A.12 *Deux variables aléatoires X et Y, à valeurs dans un ensemble fini ou dénombrable V, sont* indépendantes *si et seulement si, pour tous x et y dans V, on a*

$$\mathbb{P}(X = x \text{ et } Y = y) = \mathbb{P}(X = x)\mathbb{P}(Y = y).$$

L'indépendance de variables aléatoires est une notion importante, et a des conséquences qui ne sont pas vraies en général :

Proposition A.13 *Soient X et Y deux variables aléatoires indépendantes. On a*

$$\mathbb{E}(XY) = \mathbb{E}(X)\mathbb{E}(Y) \tag{A.3}$$

$$\mathbf{Var}(X + Y) = \mathbf{Var}(X) + \mathbf{Var}(Y) \tag{A.4}$$

Proposition A.14 *Soient X et Y deux variables aléatoires indépendantes. Alors, quelles que soient les fonctions*[4] *f et g (définies sur les ensembles de valeurs de X et Y, respectivement), $f(X)$ et $g(Y)$ sont indépendantes.*

La notion d'indépendance de deux variables aléatoires (ou de deux évènements) s'étend simplement à un nombre quelconque :

Définition A.15 *Un k-uplet $(X_1, \dots, X_k)$ de variables aléatoires est constitué de variables indépendantes, si et seulement si, pour tout vecteur de k ensembles*[5] *de valeurs $(A_1, \dots, A_k)$, on a*

$$\mathbb{P}(X_i \in A_i (1 \leq i \leq k)) = \prod_{1 \leq i \leq k} \mathbb{P}(X_i \in A_i)$$

Définition A.16 *Une famille infinie $(X_i)_{i \in I}$ de variables aléatoires est composée de variables aléatoires indépendantes, si* chaque *sous-famille finie l'est.*

Dissipons tout de suite une erreur fréquemment commise : ce n'est pas parce que 3 variables aléatoires sont deux à deux indépendantes qu'elles sont indépendantes. De même, $k+1$ variables aléatoires peuvent être telles que chaque k-uplet est un k-uplet de variables aléatoires indépendantes (on dit alors que les variables sont k à k indépendantes) sans que l'ensemble le soit. L'exemple le plus simple est celui de k variables de Bernoulli indépendantes, de paramètre $1/2$, la $k+1$-ème étant simplement leur somme modulo 2.

Exercice A.1 *Soient $X_1, \dots, X_k$, k bits aléatoires indépendants (variables de Bernoulli de paramètres $1/2$, indépendantes). Montrer que l'on peut construire, dans le même espace, $2^k - 1$ variables de Bernoulli de paramètre $1/2$, deux à deux indépendantes.*

A.1.3 Probabilités conditionnelles

Pour les définitions qui vont suivre, nous nous plaçons résolument dans le cas où l'on ne conditionne que par des évènements (ou variables aléatoires) *de probabilité strictement positive.* Il est possible de définir des conditionnements par rapport à des variables aléatoires quelconques ; cela sort du cadre de ces notes.

Définition A.17 (Probabilité conditionnelle) *Soient A et B deux évènements, avec $\mathbb{P}(B) > 0$.*

La probabilité de A sachant B, *ou* probabilité de A conditionnellement à B, *est définie par*

$$\mathbb{P}(A|B) = \frac{\mathbb{P}(A \cap B)}{\mathbb{P}(B)}.$$

Proposition A.18 *Pour tout évènement B de probabilité strictement positive, la fonction "probabilité sachant B", qui à tout évènement A associe sa probabilité conditionnelle $\mathbb{P}(A|B)$, est une loi de probabilité.*

---

[4] mesurables

[5] mesurables

Définition A.19 *Soit X une variable aléatoire, et B un évènement de probabilité strictement positive.*

La loi de X sachant B *est la loi qui, à tout ensemble I (mesurable) de valeurs possibles pour X, associe la probabilité μ_B définie par*

$$\mu_B(I) = \mathbb{P}(X \in I|B).$$

On définit de même l'espérance conditionnelle de X sachant B (c'est l'espérance d'une variable de loi μ_B), et tous les moments conditionnels.

À quoi correspondent ces notions de conditionnement ? Intuitivement, la probabilité d'un évènement sachant B est "ce que devient" la probabilité de cet évènement si l'on sait que B se réalise.

Théorème A.20 *Un évènement A est indépendant de B, si et seulement si $\mathbb{P}(A|B) = \mathbb{P}(A)$.*

Une variable aléatoire X, de loi μ, est indépendante de B, si et seulement si $\mu_B = \mu$.

Deux variables aléatoires discrètes X et Y sont indépendantes, si et seulement si, pour toute valeur y prise par Y avec probabilité strictement positive, et toute valeur x prise par X,

$$\mathbb{P}(X = x|Y = y) = \mathbb{P}(X = x).$$

A.1.4 Série génératrice de probabilités

Si X est une variable aléatoire ne prenant que des valeurs entières, la *série génératrice de probabilités*[6] de X est la série entière de variable t,

$$S_X(t) = \sum_{k \in \mathbb{N}} \mathbb{P}(X = k) t^k$$

D'après la définition de S_X, on a immédiatement

$$S_X(t) = \mathbb{E}(t^X)$$

La série S_X étant à coefficients positifs, son rayon de convergence est égal à la plus grande valeur *réelle positive* pour laquelle la série converge. Or, la série converge trivialement pour $t = 1$ (avec somme 1) : par conséquent ce rayon de convergence est au moins égal à 1, et S_X est indéfiniment dérivable sur $[0, 1[$.

Deux variables aléatoires qui ont la même série génératrice de probabilités, ont la même loi ; cette propriété est parfois très utile pour démontrer aisément des égalités en loi.

Une autre propriété extrêmement utile de la série génératrice de probabilités, est qu'elle permet souvent de calculer sans efforts les différents moments de la variable aléatoire associée :

$$\begin{aligned}
\mathbb{E}(X) &= S'_X(1) \\
\mathbb{E}(X(X-1)) &= S''_X(1) \\
\mathbb{E}(X \cdots (X-k+1)) &= S^{(k)}_X(1) \\
\mathbf{Var}(X) &= S''_X(1) + S'_X(1) - \left(S'_X(1)\right)^2
\end{aligned}$$

[6]Parfois appelée *fonction génératrice de Laplace*

(toutes les dérivées peuvent être infinies, auquel cas les moments correspondants de X sont également infinis ; dans tous les cas, ces dérivées peuvent être vues comme des limites lorsque t tend vers 1 par valeurs réelles inférieures, ce qui assure leur existence)

Si X et Y sont deux variables aléatoires à valeurs entières, *indépendantes*, alors on a

$$S_{X+Y}(t) = S_X(t)S_Y(t).$$

La réciproque n'est pas vraie : la série génératrice de probabilités de la somme de deux variables aléatoires peut être égale au produit des deux séries génératrices de probabilités, sans que les deux variables aléatoires soient indépendantes (la série génératrice de probabilités ne caractérise que la *loi* d'une variable aléatoire).

A.1.5 Espaces produits

Supposons que l'on dispose de deux espaces probabilisés (qui peuvent être distincts ou non), Ω_1 et Ω_2, et, dans chacun, d'une variable aléatoire X_1 (dans Ω_1) et X_2 (dans Ω_2) ; par exemple, X_1 peut représenter le résultat d'un lancer de dé, et X_2, le nombre de résultats "PILE" dans une série de 10 lancers d'une pièce de monnaie.

Est-il possible de faire "vivre" nos deux variables aléatoires dans le même espace probabilisé, de manière à ce qu'elles soient indépendantes ? La réponse est oui, et la construction correspondante est celle d'un espace produit.

Définition A.21 *Soient* $(\Omega_1, \mathbb{P}_1)$ *et* $(\Omega_2, \mathbb{P}_2)$ *deux espaces probabilisés. Leur* espace produit *est l'espace* $(\Omega, \mathbb{P})$*, où*

$$\begin{aligned} \Omega &= \Omega_1 \times \Omega_2 \\ \mathbb{P}(\omega_1, \omega_2) &= \mathbb{P}_1(\omega_1)\mathbb{P}_2(\omega_2) \end{aligned}$$

Il est facile de voir que, dans Ω, les variables aléatoires X'_1 et X'_2, qui ont même lois que X_1 et X_2 respectivement, sont indépendantes :

$$\begin{aligned} X'_1(\omega_1, \omega_2) &= X_1(\omega_1) \\ X'_2(\omega_1, \omega_2) &= X_2(\omega_2) \end{aligned}$$

Cette construction, éventuellement appliquée de manière répétée, permet, à partir d'un espace probabilisé contenant une variable aléatoire X, de construire un espace probabilisé dans lequel on dispose de n variables aléatoires $X_1, \ldots, X_n$, indépendantes, et toutes de même loi de probabilité que X. En étant un peu plus soigneux, il est également possible de définir un espace contenant une *infinité dénombrable* de variables aléatoires $(X_i)_{i \in \mathbb{N}}$ indépendantes et de même loi que X. C'est dans de tels espaces que "vivent" typiquement les algorithmes probabilistes utilisant des sources de Bernoulli.

Définition A.22 (Espace produit dénombrable) *Soit* $(\Omega, \mathcal{F}, \mathbb{P})$ *un espace probabilisé. On note* $\Omega^{\mathbb{N}}$*, l'espace probabilisé* $\Omega', \mathcal{F}', \mathbb{P}')$*, où*

- Ω' *est l'ensemble des suites de la forme* $(\omega_n)_{n \geq 0}$*, avec* $\omega_n \in \Omega$ *pour tout* n *;*
- $\mathcal{F}'$ *est la tribu engendrée par les ensembles de la forme* $\mathcal{E}_{n,A} = \{\omega \in \Omega' : \omega_n \in A\}$*, où* $A \in \mathcal{F}$ *et* $n \geq 0$ *;*

– *$\mathbb{P}'$ est l'unique mesure de probabilité*[7] *sur $\mathcal{F}'$ qui vérifie, pour toute suite finie $n_1, \ldots, n_k$ et toute suite d'ensembles $\mathcal{F}$-mesurables $A_1, \ldots, A_k$,*

$$\mathbb{P}'\left(\cap_{i=1}^{k} \mathcal{E}_{n_i,A_i}\right) = \prod_{i=1}^{k} \mathbb{P}(A_i)$$

A.2 Quelques exemples de lois de probabilité

Nous présentons ici quelques exemples de lois de probabilité qui interviennent souvent.

En règle générale, pour décrire une loi de probabilités, on se contente de donner les probabilités des différentes valeurs. L'espace de probabilités sous-jacent n'est donc pas clairement défini. Il ne s'agit pas là d'une négligence : l'important, c'est avant tout la loi de probabilités obtenue pour une variable aléatoire, c'est-à-dire l'image par la variable aléatoire de la mesure de l'espace de départ.

Dans la pratique, on est souvent amené à *identifier* la loi d'une certaine variable aléatoire comme étant une loi "connue". Ceci peut se faire soit par les probabilités marginales (on exprime la probabilité que X soit égale à chaque valeur k, et on "reconnaît" la fonction de k), soit par la fonction de répartition, soit par une autre caractérisation (par exemple, la série génératrice de probabilités).

Une autre remarque est qu'il suffit, pour prouver qu'une variable aléatoire X suit une loi de probabilités connue π, de prouver que la probabilité de chaque valeur possible k est *proportionnelle* à $\pi(k)$. L'étape suivante (montrer que cette probabilité est *égale* à $\pi(k)$) en découle immédiatement : le coefficient de proportionnalité ne saurait en effet être différent de $\sum_k \pi(k)$.

A.2.1 Loi de Bernoulli

Une *variable de Bernoulli* de paramètre p ($0 < p < 1$), vaut 1 avec probabilité p et 0 avec probabilité $q = 1 - p$.

Il est facile de voir que le produit de deux variables de Bernoulli indépendantes, de paramètres p_1 et p_2, est une variable de Bernoulli de paramètre $p_1 p_2$.

Si B est une telle variable aléatoire, on a

$$\begin{aligned} \mathbb{E}(B) &= p \\ \mathbf{Var}(B) &= p(1-p) \end{aligned}$$

La série génératrice de probabilités correspondante est $S_B(t) = (1-p) + tp$.

A.2.2 Loi géométrique

Une *variable géométrique* de paramètre p ($0 < p < 1$), vaut k (pour k entier strictement positif) avec probabilité $p_k = p.(1-p)^{k-1}$.

Si l'on a une suite infinie de variables de Bernoulli indépendantes, $(X_n)_{n \in \mathbb{N}}$, toutes de même paramètre p, l'indice de la première de ces Bernoulli qui vaut 1 (soit $G = \min\{n \in \mathbb{N} : X_n = 1\}$) est une variable géométrique de paramètre p.

[7] L'unicité de cette mesure produit est relativement facile à démontrer ; son existence n'est pas aussi immédiate, et résulte d'un théorème dû à Kolmogorov.

Si G est une variable aléatoire géométrique de paramètre p, on a

$$\begin{aligned} \mathbb{E}(G) &= \frac{1}{p} \\ \mathbf{Var}(G) &= \frac{1-p}{p^2} \end{aligned}$$

La série génératrice de probabilités de la géométrique de paramètre p est

$$S_G(t) = \frac{pt}{1-(1-p)t}$$

Une propriété importante de la loi géométrique est la suivante, appelée *propriété de Markov* : si X est une variable géométrique (quelle que soit son paramètre), quels que soient les entiers positifs s, t et u (avec $t < u$), on a

$$\mathbb{P}(X \in [t+s, u+s] | X > s) = \mathbb{P}(X \in [t, u])$$

Autrement dit, une variable géométrique, *conditionnée* à être plus grande que s, suit une loi géométrique (de même paramètre) décalée de s. On dit aussi que les lois géométriques sont *sans mémoire*.
Note : cette propriété de Markov *caractérise* les lois géométriques sur les entiers positifs – le lecteur est invité à montrer que, si une variable aléatoire à valeurs réelles positives vérifie la propriété de Markov ne serait-ce que pour $s = 1$, c'est en fait une variable géométrique dont le paramètre est $1 - \mathbb{P}(X = 0)$.

A.2.3 Loi binômiale

Une *variable binômiale* de paramètres n et p ($n \in \mathbb{N}$, $0 < p < 1$) vaut k (k entier, $0 \leq k \leq n$) avec probabilité[8]

$$p_k = \binom{n}{k} p^k (1-p)^{n-k}$$

Si l'on a une suite de n variables aléatoires de Bernoulli indépendantes, toutes de paramètre p, $(X_i)_{1 \leq i \leq n}$, leur somme $S_n = \sum_{i=1}^{n} X_i$ est une variable binômiale de paramètres n et p.

D'après cette remarque, il est aisé de voir que, si B_1 et B_2 sont des variables binômiales de parametres (n_1, p) et (n_2, p), indépendantes, alors $B_1 + B_2$ est une variable binômiale de paramètres $(n_1 + n_2, p)$.
Note : Cette remarque ne s'applique pas si les probabilités p des deux lois binômiales ne sont pas les mêmes.

Si X est une telle variable aléatoire, on a

$$\begin{aligned} \mathbb{E}(X) &= np \\ \mathbf{Var}(X) &= np(1-p) \end{aligned}$$

La série génératrice de probabilités de la binômiale est

$$S(t) = ((1-p) + pt)^n$$

[8] On rappelle que la notation $\binom{n}{k}$ désigne le coefficient binomial $\frac{n!}{k!(n-k)!}$.

A.2.4 Loi de Poisson

Une *variable de Poisson* de paramètre λ ($\lambda > 0$), vaut k ($k \in \mathbb{N}$) avec probabilité

$$p_k = e^{-\lambda}\frac{\lambda^k}{k!}$$

Si X est une variable de Poisson de paramètre λ, on a

$$\begin{aligned} \mathbb{E}(X) &= \lambda \\ \mathbf{Var}(X) &= \lambda \end{aligned}$$

La série génératrice de probabilités de la loi de Poisson est

$$S_P(t) = e^{\lambda(t-1)}$$

La loi de Poisson est importante car elle apparaît souvent comme "limite" de lois. Par exemple, si μ_n est la loi binômiale de paramètres n et λn (où λ est un réel positif ; μ_n n'est bien entendu définie que si $\lambda n \leq 1$), alors μ_n "tend vers" la loi de Poisson de paramètre λ, lorsque n tend vers $+\infty$ (au sens le plus simple possible : pour tout k, $\mu_n(k) \to e^{-\lambda}\lambda^k/k!$).

A.2.5 Lois uniformes

Par l'expression "loi uniforme", on désigne en fait deux types de lois, suivant que l'ensemble sous-jacent est un ensemble fini ou un intervalle (ou un produit d'intervalles, en dimension supérieure à 1).

Pour un ensemble fini de cardinal n, la loi uniforme est celle qui attribue à chaque élément de l'ensemble la même probabilité, qui est donc égale à $1/n$.

Pour un intervalle $[a, b]$, la loi uniforme est celle qui attribue à chaque sous-intervalle $[c, d]$, une probabilité proportionnelle à sa longueur ; cette probabilité est donc égale à $\frac{d-c}{b-a}$. Il n'existe pas de loi uniforme sur un intervalle non borné.

A.2.6 Loi exponentielle

Une *variable aléatoire exponentielle* de paramètre λ ($\lambda > 0$) a pour densité (définie sur $\mathbb{R}^+$)

$$\rho(x) = \frac{1}{\lambda}e^{-x/\lambda}.$$

Si X est une telle variable aléatoire, on a

$$\begin{aligned} \mathbb{E}(X) &= \lambda \\ \mathbf{Var}(X) &= \lambda^2 \end{aligned}$$

Comme les lois géométriques, les lois exponentielles vérifient la *propriété de Markov* : quels que soient les réels positifs s, t et u (avec $t < u$), on a

$$\mathbb{P}(X \in [t+s, u+s] | X \geq s) = \mathbb{P}(X \in [t, u])$$

Et, comme dans le cas géométrique, cette propriété de Markov caractérise les lois exponentielles parmi les lois de probabilités sur $\mathbb{R}^+$.

La similarité entre lois géométriques et lois exponentielles ne s'arrête pas là. Si G_n est une variable aléatoire géométrique de loi $1/n$, *la loi de G_n/n converge vers la loi exponentielle de paramètre* 1 pour une définition appropriée de la convergence des lois de probabilités qu'il n'est pas question d'expliciter ici ; disons simplement que, si G_ϵ est une variable aléatoire géométrique de paramètre ϵ *petit*, alors ϵG_ϵ "ressemble" à une variable exponentielle de paramètre 1.

A.3 Inégalités utiles

A.3.1 Inégalité de Markov

L'inégalité de Markov permet de *majorer* la probabilité qu'une variable aléatoire *positive ou nulle* soit "très au-dessus" de son espérance :

Proposition A.23 *Soit X une variable aléatoire à valeurs dans $\mathbb{R}^+$, et soit $\mu = \mathbb{E}(X)$ son espérance. Alors on a, pour tout réel strictement positif λ,*

$$\mathbb{P}(X \geq \lambda) \leq \frac{\mu}{\lambda}. \tag{A.5}$$

Preuve: Soit Y la nouvelle variable aléatoire définie par : pour tout $\omega \in \Omega$,

$$Y(\omega) = \begin{cases} 0 & \text{si } X(\omega) < \lambda \\ \lambda & \text{si } X(\omega) \geq \lambda \end{cases}$$

Il est clair que l'on a toujours $Y \leq X$, ce qui entraîne $\mathbb{E}(Y) \leq \mathbb{E}(X)$. Or, $\mathbb{E}(Y) = \lambda \mathbb{P}(Y > 0) = \lambda \mathbb{P}(X \geq \lambda)$. On a donc

$$\begin{aligned} \mathbb{P}(X \geq \lambda) &\leq \frac{\mathbb{E}(Y)}{\lambda} \\ &\leq \frac{\mathbb{E}(X)}{\lambda}. \end{aligned}$$

□

Remarquons qu'une autre façon d'écrire l'inégalité de Markov est la suivante : pour tout réel strictement positif λ,

$$\mathbb{P}(X \geq \lambda\mu) \leq \frac{1}{\lambda}.$$

A.3.2 Inégalité de Чебышев

L'inégalité de Markov est utile, mais elle ne permet pas de majorer la probabilité qu'une variable aléatoire soit *en-dessous* de son espérance. De plus, il arrive fréquemment que l'on soit confronté à des variables aléatoires dont on est capable de calculer espérance et variance, pour lesquelles l'inégalité de Tchebycheff[9] (ou Чебышев[10], ou Chebyshev, dans sa graphie internationale) est souvent plus performante pour majorer la probabilité d'être "loin" de l'espérance.

[9]Les auteurs français, dans un élan patriotard, ont tendance à l'appeler *Inégalité de Bienaymé-Tchebycheff*

[10]Au lecteur qui me soupçonnera d'avoir récemment appris comment inclure du cyrillique dans un document LaTeX 2_ε, je réponds sans hésiter : *coupable*.

Proposition A.24 *Soit X une variable aléatoire, d'espérance μ et de variance σ^2. Alors on a, pour tout réel strictement positif λ,*

$$\mathbb{P}\left(|X-\mu| \geq \lambda\sigma\right) \leq \frac{1}{\lambda^2}. \tag{A.6}$$

Preuve: Considérons la variable aléatoire $Y = (X-\mu)^2$. Par définition, cette variable aléatoire a pour espérance σ^2, et l'inégalité de Markov, appliquée à Y, donne exactement A.6. □

L'inégalité de Tchebycheff peut également se réécrire sous la forme suivante : pour tout $\lambda > 0$,

$$\mathbb{P}\left(|X-\mu| \geq \lambda\right) \leq \frac{\sigma^2}{\lambda^2}.$$

On en déduit immédiatement des majorations similaires sur la probabilité que X soit "loin au-dessus" ou "loin en dessous" de son espérance : pour tout $\lambda > 0$,

$$\mathbb{P}(X \geq \mu+\lambda) \leq \left(\frac{\sigma}{\lambda}\right)^2 \tag{A.7}$$

$$\mathbb{P}(X \leq \mu-\lambda) \leq \left(\frac{\sigma}{\lambda}\right)^2 \tag{A.8}$$

A.3.3 Inégalité de Chernoff

L'inégalité de Chernoff est une inégalité qui ne s'applique telle quelle qu'à une variable aléatoire qui se présente sous la forme d'une somme de variables de Bernoulli *indépendantes* (pas forcément de même paramètre) ; il est possible d'en donner des généralisations à des sommes de variables indépendantes de lois différentes, mais l'indépendance est une condition importante.

Proposition A.25 *Soit $X_1, \dots, X_n$ une suite de n variables aléatoires de Bernoulli* indépendantes*, X_i ayant probabilité p_i de succès avec $0 < p_i < 1$, et soit $X = \sum_{i=1}^n X_i$ leur somme. Soit également $\mu = \sum_{i=1}^n p_i$ l'espérance de X.*

Alors on a, pour tout réel $\delta > 0$,

$$\mathbb{P}\left(X > (1+\delta)\mu\right) \leq \left(\frac{e^\delta}{(1+\delta)^{1+\delta}}\right)^\mu \tag{A.9}$$

(Pour prendre conscience de l'importance de cette inégalité, il convient de noter que $e^\delta < (1+\delta)^{1+\delta}$ pour tout $\delta > 0$, et que cette quantité tend vers 0 lorsque δ tend vers $+\infty$; pour $\delta = 1$, le rapport est approximativement de 0.68. Ainsi, si μ devient grand, la borne supérieure décroit exponentiellement vite avec μ.)

La preuve de l'inégalité de Chernoff, bien qu'assez classique, est un peu calculatoire ; elle est donnée ici par souci de complétude.

Preuve: Soit t un réel positif quelconque (nous préciserons ce choix ultérieurement). Par croissance de la fonction exponentielle, il vient

$$\mathbb{P}\left(X > (1+\delta)\mu\right) = \mathbb{P}\left(e^{tX} > e^{t(1+\delta)\mu}\right).$$

Appliquons l'inégalité de Markov au membre droit (sur la variable aléatoire e^{tX}) ; il vient :

$$\mathbb{P}\left(X > (1+\delta)\mu\right) \leq \frac{\mathbb{E}\left(e^{tX}\right)}{e^{t(1+\delta)\mu}}.$$

Pour évaluer[11] $\mathbb{E}\left(e^{tX}\right)$, nous remarquons que, les variables X_i étant indépendantes, il en est de même des variables e^{tX_i}.

$$\begin{aligned} \mathbb{E}\left(e^{tX}\right) &= \mathbb{E}\left(\prod_{i=1}^{n} e^{tX_i}\right) \\ &= \prod_{i=1}^{n} \mathbb{E}\left(e^{tX_i}\right) \end{aligned}$$

La série génératrice de moments d'une variable de Bernoulli est extrêmement simple :

$$\mathbb{E}\left(e^{tX_i}\right) = p_i e^t + (1 - p_i) = 1 + p_i\left(e^t - 1\right).$$

Utilisons maintenant l'inégalité $1 + u < e^u$, valable pour tout réel $u > 0$, avec $u = p_i\left(e^t - 1\right)$; il vient alors :

$$\mathbb{E}\left(e^{tX_i}\right) < e^{p_i\left(e^t-1\right)},$$

et donc

$$\begin{aligned} \mathbb{P}\left(X > (1+\delta)\mu\right) &< \frac{\prod_{i=1}^{n} e^{p_i\left(e^t-1\right)}}{e^{t(1+\delta)\mu}} \\ &= \frac{e^{\left(e^t-1\right)\sum_{i=1}^{n} p_i}}{e^{t(1+\delta)\mu}} \\ &= \frac{e^{\left(e^t-1\right)\mu}}{e^{t(1+\delta)\mu}} \end{aligned}$$

Cette dernière inégalité est valable pour *tout* réel positif t ; nous pouvons donc choisir comme valeur, celle qui rend le membre droit le plus petit possible. Une étude rapide de cette fonction de t révèle que la fonction est décroissante, puis croissante, avec un minimum atteint en $\ln(1+\delta)$; en remplaçant t par cette valeur, on obtient l'inégalité annoncée.

□

[11] Cette quantité porte le nom de *série génératrice de moments* de la variable aléatoire X ; son existence pour tout $t > 0$ n'est absolument pas garantie si X n'est pas à support borné.

Bibliographie

[1] N. Bouleau, *Probabilités de l'ingénieur* – Hermann, 2002

[2] T. Cormen, C. Leiserson, R. Rivest, *Introduction à l'algorithmique* – Dunod, 1994 (*Introduction to Algorithms*, MIT Press, 1990).

[3] D. Foata, A. Fuchs, *Calcul des probabilités : cours, exercices et problèmes corrigés* – Dunod, 1998 (seconde édition).

[4] D. Foata, A. Fuchs, *Processus stochastiques – Processus de Poisson, chaînes de Markov et martingales : cours et exercices corrigés* – Dunod, 2002.

[5] D.E. Knuth, *The Art of Computer Programming*, volume 2, *Seminumerical Algorithms* – Addison-Wesley, 1998 (troisième édition)

[6] D.E. Knuth, *The Art of Computer Programming*, volume 3, *Sorting and Searching* – Addison-Wesley, 1998 (seconde édition).

[7] R. Motwani, P. Raghavan, *Randomized Algorithms* – Cambridge University Press, 1995.

[8] L. Devroye, *Non-uniform random variate generation* – Springer-Verlag, 1986.

[9] A. Menezes, P. van Oorschot, S. Vanstone, *Handbook of Applied Cryptography* – CRC Press, 1996.

Liste des algorithmes

1.1 QuickSort 2
1.2 RandQuickSort 3
1.3 RandMinCut 6
2.1 Source de Bernoulli 17
2.2 Source de bits aléatoires 18
3.1 RandLazySelect 30
4.1 Calcul de racine carrée modulo p 39
4.2 Test de Miller-Rabin 42

Index

algorithme probabiliste
- défini par bande aléatoire, 20
- défini par la source, 16

Chernoff
- inégalité de, 73

dérandomisation, 11

espace probabilisé
- fini ou dénombrable, 61
- non dénombrable, 63

espérance, 64
évènement, 61

fonction mesurable, 63

indépendance
- d'évènements, 65
- de variables aléatoires, 65

Las Vegas, 8
loi
- binômiale, 31, 70
- de Bernoulli, 69
- de Poisson, 71
- exponentielle, 71
- géométrique, 19, 69
- uniforme, 71

Markov
- inégalité de, 72
- propriété de, 70, 71

mesure, 62
- de probabilité, 63

moment, 65
Monte Carlo, 8
- à erreur bilatérale, 9
- à erreur unilatérale, 9

QuickSort
- déterministe, 1
- randomisé, 3

racine carrée modulaire
- calcul, 38
- définition, 36

randomisation, 11
résidu quadratique, 36

série génératrice
- de moments, 74
- de probabilités, 67

source
- de Bernoulli, 15, 17
- de bits aléatoires, 15, 17
- de loi μ, 15
- implantation, 21
- pseudo-aléatoire, 22
  - congruentielle linéaire, 23
- réelle, 16

sélection, 27
- déterministe, 28
- probabiliste, 29

Tchebycheff
- inégalité de, 31, 72

test de générateurs
- statistiques, 23
- théoriques, 24

test de primalité
- déterministe, 40
- Miller-Rabin, 41

tribu, 62
- borélienne, 62
- engendrée, 62
- engendrée par une variable aléatoire, 63

variable aléatoire, 63
- indicatrice d'évènement, 4

variance, 64