Introduction aux algorithmes probabilistes

Bruno Grenet

Septembre 2018

Ce chapitre présente les algorithmes probabilistes. Dans la première partie, on commence par deux exemples d'algorithmes probabilistes : l'un pour trouver le $k^{\text{ème}}$ plus petit élément dans un tableau, et l'autre pour trouver une coupe minimale dans un graphe. Dans la deuxième, on analyse l'algorithme de tri rapide QUICKSORT et on montre que'on peut aller plus loin que simplement borner l'espérance du temps de calcul. La troisième partie fait un récapitulatif des notions introduites par les trois exemples : on présente les deux grandes familles d'algorithmes probabilistes (Monte Carlo et Las Vegas), ainsi que les enjeux de l'analyse d'algorithmes probabilistes.

1 Deux premiers algorithmes probabilistes

Dans cette partie, on étudie nos deux premiers algorithmes probabilistes (QUICKSELECT et RANDMINCUT), afin de comprendre comment on peut utiliser l'aléatoire dans des algorithmes.

1.1 Problème de la sélection

Le problème est de la sélection est le suivant : étant donné un tableau T d'entiers non triés (deux à deux distincts pour simplifier) et un entier k, on veut trouver le $k^{\text{ème}}$ plus petit élément de T, que l'on note $T_{(k)}$. Par exemple, $T_{(1)}$ est le plus petit élément de T. Deux algorithmes simples peuvent être donnés : soit on compte, pour chaque élément du tableau, le nombre d'éléments qui lui sont inférieur et on renvoie celui qui a $(k-1)$ éléments plus petits que lui ; soit on trie le tableau puis on renvoie le $k^{\text{ème}}$ élément du tableau. La première solution a une complexité $O(n^2)$ (où n est la taille de T) tandis que la deuxième a une complexité $O(n \log n)$ en utilisant un tri efficace (par ex. tri fusion ou tri rapide dont on reparlera plus tard). Ici, et dans la suite, on exprime la complexité en termes de nombre de comparaisons effectuées.

Nous allons présenter un algorithme *probabiliste* qui résout ce problème en temps $O(n)$. On commence par décrire l'algorithme avant de discuter de ce que signifie *probabiliste* dans ce contexte.

QUICKSELECT(T, k) :
Entrée : un tableau T de n entiers non triés, un entier k.
Sortie : le $k^{ème}$ plus petit élément de T.
1. $i \leftarrow$ entier aléatoire entre 0 et $n-1$
2. $B, H \leftarrow$ tableaux vides
3. Pour $j = 0$ à $n-1$, $j \neq i$:
4. Si $T[j] < T[i]$:
5. Ajouter $T[j]$ à B

6. Sinon :
7. Ajouter $T[j]$ à H
8. Si $|B| = k-1$: renvoyer $T[i]$
9. Si $|B| > k-1$: renvoyer QUICKSELECT(B,k)
10. Sinon : renvoyer QUICKSELECT$(H, k-|B|-1)$

Lemme 1.1. *L'algorithme* QUICKSELECT *est correct : il renvoie le $k^{\text{ème}}$ plus petit élément de T.*

Démonstration. Les lignes 3 à 7 partitionnent T en deux sous-tableaux B et H tels que B contiennent tous les éléments strictement inférieurs à $T[i]$ et H les éléments strictement supérieurs à $T[i]$. Si B contient exactement $k-1$ éléments, $T[i]$ est $T_{(k)}$ (le $k^{\text{ème}}$ plus petit élément de T) et l'algorithme est correct. Sinon, si B contient plus de $k-1$ éléments, $T_{(k)}$ appartient forcément à B et est son $k^{\text{ème}}$ plus petit élément (autrement dit $T_{(k)} = B_{(k)}$). Sinon, il appartient à H. Comme B contient $|B|$ éléments inférieurs à ceux de H et que $T[i]$ est également inférieur à ceux de H, tout élément de H a au moins $|B|+1$ éléments plus petits que lui dans T. Donc le $\ell^{\text{ème}}$ élément de H est le $(\ell+|B|+1)^{\text{ème}}$ élément de T. Donc $T_{(k)} = H_{(k-|B|-1)}$.

Pour finir la démonstration, il suffit de montrer le résultat par récurrence sur la taille n de T. Si $n = 1$, on se rend aisément compte que B et H sont vides après la boucle. Comme $k = 1$ nécessairement dans ce cas, l'algorithme renvoie bien le seul élément du tableau à la ligne 8. Maintenant si $n > 1$, l'appel récursif de la ligne 9 ou 10 (selon les cas) est correct par hypothèse de récurrence. La discussion précédente montre que l'algorithme renvoie bien $T_{(k)}$.

□

Comme on le voit, les probabilités n'interviennent pas dans cette preuve de correction : cet algorithme probabiliste renvoie *toujours* le bon résultat. Nous étudions maintenant sa complexité. On remarque que si on n'a pas de chance à la ligne 1, l'entier i choisi est tel que $T[i] = T_{(1)}$ par exemple. Supposons qu'on cherche le $k^{\text{ème}}$ élément, $k > 1$. Alors l'algorithme effectuera un appel récursif sur le tableau H, qui sera de taille $n-1$. Si à chaque appel récursif on a la même malchance, la taille du tableau sur lequel l'appel récursif est effectué diminuera de 1 à chaque étape. Puisque chaque étape coûte $n-1$ comparaisons (puisqu'une boucle parcourt tout le tableau), la complexité dans le pire des cas de cet algorithme est $(n-1)+(n-2)+\cdots+1 = O(n^2)$ comparaisons, c'est-à-dire aussi mauvais que l'algorithme naïf initial. Cependant, il faudrait beaucoup de malchance pour avoir un comportement aussi défavorable ! Le lemme suivant quantifie cette remarque, montrant qu'*en général, tout se passe bien*.

Lemme 1.2. *Soit T un tableau de taille n et k un entier entre 1 et n. Alors l'*espérance *du nombre de comparaisons effectuées par* QUICKSELECT(T,k) *est bornée par $4n$.*

Remarque. L'aspect probabiliste (l'espérance) concerne les choix aléatoires effectués au cours de l'algorithme, et en aucun cas le choix de T ou de k. Le résultat précédent est bien valable *quelque soit T et k*.

Démonstration. On veut montrer que *pour tout tableau T de taille n et toute valeur de k entre 1 et n*, l'espérance du nombre de comparaisons effectuées par QUICKSELECT(T,k) est bornée par $4n$. On procède pour cela par récurrence sur n. Le cas $n = 1$ est trivial : il n'y a pas de passage dans la boucle, donc pas de comparaison effectuée. Ainsi, l'espérance du nombre de comparaison est 0 et le cas de base est vérifié. Il s'agit maintenant de montrer que si le résultat est vrai pour tout $m < n$, il reste vrai pour n.

Fixons donc un tableau T de taille n et un entier k entre 1 et n. On définit la variable aléatoire X_n qui compte le nombre de comparaisons effectuées au cours de l'algorithme [1]. On cherche à borner $\mathbb{E}[X_n]$.

1. La variable X_n dépend de T et de k, l'indice n rappelle que c'est la dépendance en la taille du tableau qui nous intéresse.

On utilise la formule de l'espérance totale, en conditionnant X_n à la taille du tableau B après la boucle :

$$\mathbb{E}[X_n] = \sum_{m=0}^{n-1} \mathbb{P}[|B| = m]\,\mathbb{E}[X_n||B| = m].$$

Pour tout $m < n$, $\mathbb{P}[|B| = m] = 1/n$ car chaque élément du tableau peut être choisi avec cette probabilité : si on choisit le plus petit, $|B| = 0$, etc.

On cherche maintenant à borner $\mathbb{E}[X_n||B| = m]$ pour les différentes valeurs de m. On remarque que cette espérance conditionnelle dépend de la valeur de m par rapport à k : si $m = k-1$, le $k^{\text{ème}}$ élément a été trouvé et $\mathbb{E}[X_n||B| = k-1] = n-1$ [2]. Si $m < k-1$, l'appel récursif est effectué sur H, qui est de taille $n-m-1$. Par hypothèse de récurrence, l'espérance du nombre de comparaisons effectuées lors de cet appel récursif est bornée par $4(n-m-1)$. Ainsi, $\mathbb{E}[X_n||B| = m] \leq (n-1) + 4(n-m-1)$ dans ce cas. Si $m > k-1$, l'appel récursif est effectué sur B et par hypothèse de récurrence $\mathbb{E}[X_n||B| = m] \leq (n-1)+4m$ dans ce dernier cas.

Notre objectif est d'avoir une borne générale *qui ne dépende pas de* k. Pour cela, on suppose être dans le cas *le plus défavorable* à chaque fois, c'est-à-dire qu'il y a un appel récursif sur le plus grand tableau. Mathématiquement, cela signifie qu'on borne l'espérance conditionnelle de X_n par le maximum des trois cas précédents, qui vaut $(n-1) + 4\max(m, n-m-1)$. On obtient

$$\mathbb{E}[X_n] \leq \sum_{m=0}^{n-1} \frac{1}{n} \cdot [(n-1) + 4\max(m, n-m-1)] = (n-1) + \frac{4}{n} \sum_{m=0}^{n-1} \max(m, n-m-1).$$

La fin n'est que du calcul. Dans la somme, $\max(m, n-m-1) = n-m-1$ pour $m \leq (n-1)/2$ et $\max(m, n-m-1) = m$ pour $m > (n-1)/2$. En supposant par simplicité que n est pair, et en regroupant les termes égaux, on obtient

$$\sum_{m=0}^{n-1} \max(m, n-m-1) = 2 \sum_{m=n/2}^{n-1} m = 2\sum_{m=0}^{n-1} m - 2 \sum_{m=0}^{n/2-1} m.$$

En utilisant la formule de la somme des premiers entiers, on trouve que la somme vaut $n(n-1) - n/2(n/2-1)$. Finalement,

$$\mathbb{E}[X_n] \leq (n-1) + \frac{4}{n} n(n-1) - \frac{4}{n}\frac{n}{2}(\frac{n}{2} - 1) = 4n - 3$$

ce qui conclut la récurrence.

□

Exercice. Démontrer le résultat sans supposer n pair.

1.2 Problème de la coupe minimale

Le problème de la coupe minimale consiste à chercher, pour un graphe $G = (V, E)$ donné, une partition des sommets $V = G \sqcup D$ qui minimise la *coupe* $C = \{uv \in E : u \in G, v \in D\}$. Autrement dit, on cherche une partition des sommets telle qu'il existe le moins possible d'arêtes entre les deux sous-ensembles de sommets. On généralise le problème aux *multigraphes* dans lesquels on accepte un nombre arbitraire

2. On a déjà effectué $n-1$ comparaisons dans la boucle, et on n'en effectue plus dans ce cas.

d'arêtes entre deux sommets. Un algorithme naïf consiste à tester toutes les partitions possibles, et a une complexité de l'ordre de $O(2^n)$ où n est le nombre de sommets. Il existe des algorithmes polynomiaux pour ce problème. Nous allons voir un algorithme probabiliste, que nous analyserons ensuite.

RANDMINCUT(G) :
1. Tant que G a au moins 3 sommets :
2. Choisir une arête uv aléatoirement
3. Contracter uv
4. Renvoyer la coupe définie par les deux sommets restants

Avant d'analyser l'algorithme, on donne quelques précisions. La *contraction* d'une arête uv est l'action suivante : on remplace les deux sommets u et v par un unique sommet s_{uv} ; chaque arête entre u ou v et un autre sommet w est remplacée par une arête entre s_{uv} et w ; les arêtes entre u et v sont supprimées. Le graphe résultant peut avoir plusieurs arêtes entre deux sommets (si le graphe original a les arêtes uw et vw par exemple). La *coupe définie par les deux sommets* est obtenue ainsi : après toutes les contractions d'arêtes, chaque sommet va se retrouver contracté dans l'un des deux derniers sommets restants ; en appelant g et d les deux sommets restants, on définit la coupe $V = G \sqcup D$ en mettant dans G tous les sommets qui ont été contractés dans g et D ceux qui ont été contractés dans D.

On remarque qu'à chaque étape, une arête est contractée. La complexité de l'algorithme ne dépend donc pas des choix aléatoire.

Lemme 1.3. *L'algorithme* RANDMINCUT *effectue $n-2$ contractions d'arêtes pour un graphe à n sommets. Le temps global d'exécution est $O(n^2)$ si le graphe est représenté par listes d'adjacences.*

Démonstration. La première affirmation est claire puisqu'à chaque contraction, le graphe a un sommet de moins. Le temps pris par une contraction peut être ramené à $O(n)$ lorsqu'on travaille sur des listes d'adjacence, d'où le résultat.

□

Exercice. Montrer qu'une contraction peut être effectuée en temps $O(n)$ lorsque le graphe est représenté par liste d'adjacences.

Maintenant que la complexité est démontrée, il faut voir si l'algorithme renvoie le bon résultat, c'est-à-dire une coupe minimale. En exécutant l'algorithme sur quelques exemples, on se rend facilement compte que ce n'est pas toujours le cas ! On peut cependant montrer qu'*avec un peu de chance, on trouve bien une coupe minimale*.

Lemme 1.4. *L'algorithme* RANDMINCUT *appliqué à un graphe à n sommets renvoie une coupe minimale avec probabilité $\geq 2/n(n-1)$.*

Avant de démontrer ce résultat, faisons une remarque : cette probabilité est très petite ! Autrement dit, l'algorithme ne renvoie que rarement le bon résultat. Nous verrons ensuite comment régler ce problème.

Démonstration. On fixe un graphe G et on s'intéresse à la coupe C renvoyée par RANDMINCUT. On cherche à démontrer que la probabilité que C soit une coupe minimale est au moins $2/n(n-1)$.

Pour cela, fixons *une* coupe minimale (il pourrait en exister plusieurs) $C_{\min}$. On va montrer que $\mathbb{P}[C = C_{\min}] \geq 2/n(n-1)$. Puisque la probabilité que RANDMINCUT renvoie une coupe minimale est

supérieure ou égale à la probabilité qu'il renvoie cette coupe minimale particulière, le résultat sera démontré [3].

L'algorithme renvoie $C_{\min}$ si et seulement si, à chaque passage dans la boucle, l'arête choisie n'appartient pas à $C_{\min}$. On numérote les passages dans la boucle *Tant que* en fonction du nombre de sommets restant dans G : le premier passage est l'étape n, le second l'étape $n-1$, …, le dernier passage l'étape 3. Pour $3 \leq k \leq n$, on note E_k l'évènement « l'arête choisie à l'étape k n'appartient pas à $C_{\min}$ ». On cherche donc à montrer que $\mathbb{P}\left[\bigwedge_k E_k\right] \geq 2/n(n-1)$. Par définition des probabilités conditionnelles,

$$\mathbb{P}\left[\bigwedge_{k=3}^{n} E_k\right] = \mathbb{P}[E_n] \cdot \mathbb{P}[E_{n-1}|E_n] \cdot \mathbb{P}[E_{n-2}|E_n, E_{n-1}] \cdots \mathbb{P}\left[E_3 \middle| E_n, \ldots, E_4\right].$$

On cherche donc à borner chacune des probabilités conditionnelles précédentes. On utilise pour cela un résultat facile de théorie des graphes : *si un graphe G à k sommets possède une coupe minimale de taille m, alors G possède au moins $km/2$ arêtes* [4].

Soit m le nombre d'arêtes de $C_{\min}$. Pour $3 \leq k \leq n$, $\mathbb{P}[E_k|E_n, \ldots, E_{k+1}]$ est la probabilité de ne pas tirer une arête de $C_{\min}$ à l'étape k, *sachant qu'on en a pas tiré précédemment*. Puisqu'à l'étape k, G n'a plus que k sommets mais toujours une coupe minimale de taille m (puisqu'aucune arête de $C_{\min}$ n'a été tirée dans les étapes précédentes), le résultat précédent assure que G a au moins $km/2$ arêtes. La probabilité de tirer une des m arêtes de $C_{\min}$ est donc au plus $m/(km/2) = 2/k$. Autrement dit, $\mathbb{P}[E_k|E_n, \ldots, E_{k+1}] \geq 1-2/k$.

On en déduit que $\mathbb{P}\left[\bigwedge_{k=3}^{n} E_k\right] \geq \prod_{k=3}^{n} n(1-2/k) = 2/n(n-1)$. Ainsi, la probabilité que l'algorithme renvoie une coupe minimale est au moins $2/n(n-1)$.

□

Exercice. Montrer que $\prod_{k=3}^{n}(1-2/k) = 2/n(n-1)$. *Indication. Écrire $1-2/k$ sous la forme $(k-2)/k$ et trouver des simplifications.*

Si on a un graphe de 100 sommets, la probabilité de succès est donc au moins de $2/990 \simeq 0,02\%$. Autrement dit, il y a très peu de chance que l'algorithme fonctionne. Cependant, on peut améliorer grandement cette probabilité en répétant simplement l'algorithme plusieurs fois, et en gardant la plus petite coupe. Cette technique extrêmement classique en algorithmique probabiliste est appelée l'*amplification*.

Lemme 1.5. *Si on applique N fois l'algorithme* RandMinCut *et qu'on conserve la plus petite des coupes obtenues, cette coupe est minimale avec probabilité au moins $1-(1-2/n(n-1))^N \geq 1-e^{-2N/n(n-1)}$.*

Démonstration. Chaque exécution de l'algorithme est appliquée de manière indépendante. Si on note F_k l'évènement « la $k^{\text{ème}}$ exécution de l'algorithme renvoie la coupe minimale » (pour $1 \leq k \leq N$), les évènements F_k sont donc deux-à-deux indépendants. La probabilité qu'aucune exécution ne renvoie une coupe minimale est

$$\mathbb{P}\left[\bigwedge_{k=1}^{N} \neg F_k\right] = \prod_{k=1}^{N} \mathbb{P}[\neg F_k] \leq \left(1 - \frac{2}{n(n-1)}\right)^N.$$

3. Cette technique est très classique en analyse d'algorithmes probabilistes, et sera réutilisée plusieurs fois. La justification théorique est très simple : s'il existe t coupes minimales $C_{\min}^1$, $C_{\min}^2$, …, $C_{\min}^t$, l'évènement « RandMinCut renvoie une coupe minimale » est l'union des évènements « $C = C_{\min}^s$ » pour $1 \leq s \leq t$; la probabilité de l'union est nécessairement supérieure ou égale à la probabilité de chacun des évènements qui la composent.

4. *Démonstration.* Tous les sommets de G ont au moins m voisins : sinon, s'il existait un sommet s avec $< m$ voisins, la coupe $\{s\} \sqcup G \setminus \{s\}$ serait de taille $< m$, et la coupe minimale ne serait pas de taille m. La somme du nombre de voisins de tous les sommets est le double du nombre d'arêtes (puisque chaque arête est comptée deux fois dans la somme), et la somme vaut au moins km puisque chaque sommet a au moins m sommets. D'où le résultat.

Ainsi, la probabilité d'obtenir une coupe minimale au cours de l'algorithme est au moins $1-(1-2/n(n-1))^N$.

La borne du lemme s'obtient en appliquant l'inégalité $1+x \leq e^x$ avec $x = -2/n(n-1)$: on obtient $1-2/n(n-1) \leq e^{-2n(n-1)}$, d'où $1-(1-2/n(n-1))^N \geq 1-e^{-2N/n(n-1)}$.

□

Une conséquence du lemme est qu'appliquer l'algorithme n^2 fois permet d'avoir une coupe minimale avec probabilité $1-e^{-2n/(n-1)} \geq 1-e^{-4} \geq 98\%$ (car $n \geq 2$). D'après le lemme 1.3, l'algorithme global obtenu est de complexité $O(n^4)$. Si on souhaite une meilleure probabilité de succès, il suffit d'augmenter un peu le nombre de répétitions. Par exemple, avec $2n^2$ répétitions, on obtient une probabilité de succès supérieure à 99,96%.

2 Analyse de l'algorithme de tri rapide

L'algorithme de tri rapide, ou QUICKSORT, est un algorithme de type « diviser-pour-régner » très classique. Il est naturellement probabiliste. L'objectif de cette partie est d'analyser son comportement.

2.1 Description et preuve de correction

QUICKSORT(T) :
Entrée : un tableau T de n entiers non trié.
Sortie : le tableau T trié.
1. Si $|T| = 1$, renvoyer T
2. $i \leftarrow$ entier aléatoire entre 0 et $n-1$
3. Partitionner T en deux sous-tableau B et H tel que B contienne les éléments $< T[i]$, et H les éléments $\geq T[i]$
4. Utiliser deux appels récursifs pour trier B et H
5. Renvoyer la concaténation de B, $T[i]$ et H

Remarque. La description ci-dessus est très succincte. On peut implanter cet algorithme *en place*, c'est-à-dire sans utiliser de tableau supplémentaire, mais ce n'est pas ce qui nous intéresse dans le cadre de ce cours.

Lemme 2.1. *L'algorithme* QUICKSORT *est correct, c'est-à-dire qu'il renvoie toujours un tableau trié contenant les mêmes éléments que T.*

Démonstration. On démontre aisément ce résultat par induction : en supposant que B et H ont été correctement triés par les appels récursifs, il est évident que le tableau renvoyé à la dernière ligne est trié.

□

L'algorithme QUICKSORT est donc du même type que QUICKSELECT : le résultat renvoyé est toujours correct. Dans la suite, on s'intéresse à son temps de calcul. Si le pivot choisi (avec l'entier i) est mauvais, les tailles des deux sous-tableaux B et H seront très déséquilibrées. Dans le pire cas, l'un des deux est vide et l'autre contient $n-1$ éléments. La complexité (en nombre de comparaisons) vérifie alors $C(n) = n-1+C(n-1)$, ce qui se résout en $C(n) = O(n^2)$. On va montrer que ce comportement n'est pas celui attendu. Dans la deuxième partie, on montre que l'espérance du temps de calcul est $O(n \log n)$. Dans la troisième, on s'intéresse à la question suivante : le temps de calcul est-il souvent significativement

plus grand que son espérance ? La réponse est non, et de manière assez forte : avec très forte probabilité, le temps de calcul est bien $O(n \log n)$.

2.2 Espérance du temps de calcul

Théorème 2.2. *Soit T un tableau de taille n. Alors l'espérance du nombre de comparaisons effectuées par* QUICKSORT(T) *est $O(n \log n)$.*

On souhaite calculer l'espérance du nombre de comparaisons effectuées au cours de l'algorithme. Pour cela, on calcule pour chaque couple d'éléments de T la probabilité qu'ils soient comparés au cours de l'algorithme. Le nombre de couples qui ont été comparés donne précisément le nombre de comparaisons effectuées.

Pour $1 \leq i \leq n$, soit $T_{(i)}$ le $i^{\text{ème}}$ élément de T par ordre croissant et pour $1 \leq i < j \leq n$, soit X_{ij} la variable aléatoire qui vaut 1 si $T_{(i)}$ est comparé à $T_{(j)}$ au cours de l'algorithme, et 0 sinon. On définit également $X = \sum_{i<j} X_{ij}$. Ainsi, $\mathbb{E}[X]$ est exactement l'espérance du nombre de comparaisons effectuées au cours du calcul.

Par linéarité de l'espérance, $\mathbb{E}[X] = \sum_{i<j} \mathbb{E}\left[X_{ij}\right] = \sum_{i<j} \mathbb{P}\left[X_{ij} = 1\right]$ (puisque $X_{ij} \in \{0,1\}$).

Lemme 2.3. *Pour $1 \leq i < j \leq n$, $\mathbb{P}\left[X_{ij} = 1\right] = 2/(j-i+1)$.*

Démonstration. Pour que $T_{(i)}$ et $T_{(j)}$ soient comparés au cours de l'algorithme, il faut que l'un des deux soit choisi comme pivot alors qu'il appartiennent encore au même sous-tableau. Or si un élément $T_{(k)}$ avec $i < k < j$ a été choisi préalablement, $T_{(i)}$ et $T_{(j)}$ sont dans deux sous-tableaux différents. Autrement dit, $T_{(i)}$ et $T_{(j)}$ sont comparés si et seulement si $T_{(i)}$ ou $T_{(j)}$ est le premier élément parmi $T_{(i)}$, $T_{(i+1)}$, …, $T_{(j)}$ à être choisi comme pivot. Chacun de ces $(j-i+1)$ éléments a la même probabilité d'être choisi en premier, d'où $\mathbb{P}\left[X_{ij} = 1\right] = 2/(j-i+1)$.

□

La fin de la démonstration est du calcul :

$$\mathbb{E}[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} = \sum_{i=1}^{n} \sum_{k=2}^{n-i+1} \frac{2}{k} \leq \sum_{i=1}^{n} \sum_{k=1}^{n} \frac{2}{k} = 2nH_n$$

où $H_n = \sum_{k=1}^{n} 1/k$ est le $n^{\text{ème}}$ *nombre harmonique*. Puisque $H_n = \ln(n) + \Theta(1)$, on en déduit que $\mathbb{E}[X] \leq 2n \ln(n) + \Theta(n) = O(n \log n)$.

2.3 Au delà de l'espérance

Le résultat précédent informe que, quelque soit le tableau en entrée, l'espérance du temps de calcul de QUICKSORT est bonne : les cas patologiques en $O(n^2)$ sont finalement rares. Cependant, il ne s'agit que de l'espérance et on ne connaît pas précisément la distribution des temps de calculs, pour une entrée donnée. Il se pourrait très bien qu'avec probabilité $1/5$, le temps de calcul soit $\geq n^2/5$ (chiffres fantaisistes). Cela signifierait que sur une entrée donnée, le calcul prendrait beaucoup de temps avec une probabilité non négligeable. On souhaite au contraire montrer que la probabilité que le calcul soit lent est en fait négligeable.

La première approche est d'utiliser la borne de Markov : pour toute variable aléatoire X, $\mathbb{P}[X \geq \lambda \mathbb{E}[X]] \leq 1/\lambda$. Ici, cela signifie par exemple qu'avec probabilité $\geq 9/10$, l'algorithme n'effectue pas plus de 10 fois

plus de comparaisons que la borne du lemme précédent. Cependant, cette borne est assez faible : elle n'exclue pas qu'avec une probabilité $1/10$ (donc non négligeable !), l'algorithme prenne un temps $O(n^2)$.

L'objectif de cette partie est de démontrer le résultat plus fort suivant à l'aide de la borne de Chernoff.

Théorème 2.4. *Soit T un tableau de taille n. Alors* QUICKSORT(T) *effectue* $\leq 20n\ln n = O(n\log n)$ *comparaisons avec probabilité au moins* $1-1/n$.

Idée de la démonstration. Cette démonstration est plus conceptuelle que celle de la partie précédente. On considère l'arbre de récursion de l'exécution de l'algorithme sur une entrée donnée, dont chaque nœud est étiqueté par le sous-tableau courant : la racine est étiquetée avec le tableau T en entrée, ses trois fils par les tableaux B et H et le tableau contenant un seul élément $T[i]$, etc. L'objectif est de montrer qu'avec grande probabilité, la profondeur de cet arbre est bornée par $O(\log n)$. Pour cela, on montre que la taille des tableaux diminue suffisamment vite, avec grande probabilité, le long de chaque chemin de la racine à une feuille.

□

Démonstration. Fixons donc un tableau T, et intéressons-nous à l'arbre de récursion de l'algorithme. On remarque qu'il y a exactement n feuilles dans cet arbre puisque ce sont les n sous-tableaux de taille 1 possibles.

On fixe un élément z quelconque du tableau T. Le sous-tableau ne contenant que z étiquette une feuille : on s'intéresse au chemin entre la racine et cette feuille. Notre objectif est de montrer qu'avec grande probabilité, ce chemin est de longueur au plus $c\ln n$ (pour une constante c qui sera fixée plus tard). Notons M la variable aléatoire qui mesure la longueur de ce chemin. Numérotons de 0 (la racine) à M (la feuille) les nœuds de ce chemin et notons t_i la taille du tableau qui étiquette le nœud i (les t_i sont donc aussi des variables aléatoires). Les t_i vérifient $t_i < t_{i-1}$ pour tout i, $t_0 = n$ et $t_m = 1$. Par convention, on définit $t_i = 1$ pour tout $i > M$. Alors pour tout m, $\mathbb{P}[M > m] = \mathbb{P}[t_m > 1]$. Notre objectif est donc de montrer que t_i *est* souvent très inférieur* à t_{i-1}, ce qui impliquera que pour m pas trop grand, $t_m \leq 1$.

Pour formaliser ces notions, on appelle *équilibrée* une partition de T en deux sous-tableaux B et H telle que $|T|/4 \leq |B|, |H| \leq 3|T|/4$. On remarque que si la partition du tableau étiquetant le nœud i est équilibrée, $t_{i+1} \leq (3/4)t_i$ par définition. La probabilité qu'une partition soit équilibrée est $1/2$: en effet, la partition est équilibrée si et seulement si le pivot choisi est parmi les $n/2$ éléments $T_{(n/4)}, \ldots, T_{(3n/4)}$ (sur les n choix possibles de pivot). Autrement dit, en définissant la variable aléatoire Z_i qui vaut 1 si $t_{i-1}/4 \leq t_i \leq (3/4)t_{i-1}$ et 0 sinon, $\mathbb{P}[Z_i = 1] = 1/2$. On définit également pour tout ℓ la variable aléatoire $Z_{(\ell)} = Z_1 + \cdots + Z_\ell$ qui compte le nombre de partitions équilibrées sur le chemin de la racine vers le nœud ℓ. On remarque que si $Z_{(\ell)} \geq \ln(n)/\ln(4/3)$, $t_\ell \leq 1$ (donc $t_\ell = 1$) : s'il y a eu k partitions équilibrées entre la racine et le nœud ℓ, la taille du tableau étiquetant ce nœud est au plus $(3/4)^k n$ et $(3/4)^{\ln(n)/\ln(4/3)} = 1/n$. On en déduit [5] que $\mathbb{P}[t_m > 1] \leq \mathbb{P}\left[Z_{(m)} < \ln(n)/\ln(4/3)\right]$.

On cherche donc à borner la probabilité que $Z_{(m)} < \ln(n)/\ln(4/3)]$. On utilise la borne de Chernoff. L'espérance de $Z_{(m)}$ est $\mathbb{E}\left[Z_{(m)}\right] = \sum_{i\leq m}\mathbb{E}[Z_i] = m/2$ car les Z_i sont des variables aléatoires à valeur dans $\{0,1\}$ et que $\mathbb{P}[Z_i = 1] = 1/2$.

Si on pose $m = c\ln n$ pour une constante c à fixer,

$$\mathbb{P}\left[Z_{(c\ln n)} < \frac{\ln n}{\ln 4/3}\right] = \mathbb{P}\left[Z_{(c\ln n)} = \frac{c\ln n}{2}\cdot\frac{2}{c\ln 4/3}\right] \leq e^{-\frac{c\ln n}{4}(1-\frac{2}{c\ln 4/3})^2} = \left(\frac{1}{n}\right)^{\frac{c}{4}(1-\frac{2}{c\ln 4/3})^2}$$

5. De manière intuitive, si un évènement E implique un évènement F, alors $\mathbb{P}[E] \leq \mathbb{P}[F]$. De manière formelle, $\mathbb{P}[E] = \mathbb{P}[E|F]\mathbb{P}[F] + \mathbb{P}[E|\neg F]\mathbb{P}[\neg F] \leq \mathbb{P}[F]$, puisque $\mathbb{P}[E|F] \leq 1$ et $\mathbb{P}[E|\neg F] = \mathbb{P}[E\wedge\neg F]/\mathbb{P}[\neg F] = 0$.

(en utilisant $e^{-\ln n} = 1/n$). Dès que l'exposant $\frac{c}{4}(1-2/c\ln(4/3))^2$ dépasse 2 (ce qui arrive pour $c \geq 20$), on obtient $\mathbb{P}[m > c\ln n] \leq 1/n^2$.

En conclusion,

$$\mathbb{P}[M > 20\ln n] = \mathbb{P}[t_{20\ln n} > 1] \leq \mathbb{P}\left[Z_{(20\ln n)} < \frac{\ln n}{\ln 4/3}\right] \leq 1/n^2.$$

Puisque cette borne est valable pour chacun des n chemins entre la racine et une feuille, l'inégalité de Boole permet de conclure que la probabilité que l'un des chemins dépasse la longueur $20\ln n$ est au plus $1/n$.

Pour conclure, chaque élément se trouve dans au plus un sous-tableau à chaque profondeur de l'arbre. Donc le nombre total de comparaisons effectuées à une profondeur fixée est au plus $n-1$. Ainsi, le nombre total de comparaison est au plus $20(n-1)\ln n$ avec probabilité $\geq 1-1/n$.

□

Remarque. Le résultat obtenu est-il réellement meilleur que celui obtenu très facilement avec l'inégalité de Markov ? La réponse est oui ! Pour un tableau de taille 1 000 000, le théorème 2.4 montre qu'avec probabilité au moins 99,9999%, l'algorithme effectue au plus $20n\ln n$ comparaisons. Avec l'inégalité de Markov, on n'obtient la même borne qu'avec probabilité 90% environ.

Exercice. Redémontrer que l'espérance du nombre de comparaisons effectué par QuickSort est $O(n\log n)$, en utilisant le lemme précédent.

3 Récapitulatif : classification des algorithmes probabilistes

Les algorithmes QUICKSELECT et QUICKSORT répondent toujours correctement, mais leurs temps de calcul sont des variables aléatoires. À l'inverse, le temps de calcul de RANDMINCUT est garanti, mais son résultat n'est correct qu'avec une certaine probabilité.

Définition 3.1. Un algorithme de type *Las Vegas* est un algorithme probabiliste dont le résultat est toujours correct, mais dont le temps de calcul est une variable aléatoire.

Un algorithme de type *Monte Carlo* est un algorithme probabiliste dont le temps de calcul est garanti, mais dont le résultat n'est correct qu'avec une certaine probabilité.

En résumé, un algorithme de type Las Vegas est « toujours correct, probablement rapide » alors qu'un algorithme de type Monte Carlo est « toujours rapide, probablement correct ». Ainsi, QUICKSELECT et QUICKSORT sont des algorithmes de type Las Vegas alors que RANDMINCUT est un algorithme de type Monte Carlo.

Remarque. On peut raffiner la classification, en distinguant plusieurs types d'algorithmes Monte Carlo. Les premiers ne se trompent *que d'un côté* : par exemple, c'est le cas de RANDMINCUT qui ne peut pas trouver une coupe trop petite, ou d'un problème de décision [6] qui n'aurait que des faux positifs mais pas de faux négatif [7], ou l'inverse. Les seconds sont ceux qui peuvent se tromper *des deux côtés*, c'est-à-dire avoir à la fois des faux positifs ou des faux négatifs. Ces différentes classifications, dans le cadre des algorithmes en temps polynomial, sont capturés par différentes *classes de complexité* : ZPP correspond aux algorithmes Las Vegas, BPP aux algorithmes Monte Carlo avec erreur des deux côtés, RP aux

6. Un problème de décision est un problème qui n'admet que deux réponses possibles, soit « oui » soit « non ».
7. On parle de faux positif si la réponse est « non » mais que l'algorithme répond « oui », et de faux négatif dans le cas contraire.

algorithmes Monte Carlo n'ayant que des faux négatifs, et coRP à ceux n'ayant que des faux positifs. Après formalisation de ces classes, on peut montrer que $\mathsf{ZPP} \subset \mathsf{RP}, \mathsf{coRP} \subset \mathsf{BPP}$. Se reporter au cours de complexité algorithmique pour plus de détails.

Comme on l'a vu dans le cas de RANDMINCUT, la probabilité de succès d'un algorithme de type Monte Carlo peut en général être amplifiée de manière exponentiellement rapide en répétant l'algorithme plusieurs fois. De la même manière, il est souvent possible de transformer un algorithme de type Monte Carlo en un algorithme de type Las Vegas : lorsqu'on sait vérifier (rapidement) si le résultat obtenu est correct, il suffit de répéter l'algorithme tant qu'un résultat correct n'est pas obtenu.

Les lemmes suivants quantifient ces remarques. Le premier lemme montre essentiellement qu'un algorithme Monte Carlo de complexité $T(n)$ et ayant une probabilité de succès p peut être transformé en un algorithme Las Vegas dont l'espérance du temps de calcul est $(1/p)T(n)$.

Lemme 3.2. *Supposons qu'un algorithme de type Monte Carlo renvoie une réponse correcte avec probabilité $p > 0$. Alors l'espérance du nombre de répétitions nécessaires de l'algorithme pour obtenir une réponse correcte est $1/p$.*

Démonstration. On note X la variable aléatoire qui détermine le nombre d'exécutions avant d'obtenir une bonne réponse, et on cherche à calculer $\mathbb{E}[X]$. On donne trois démonstrations.

1. Puisque chaque exécution est indépendante, $\mathbb{P}[X = k] = p(1-p)^{k-1}$: il faut avoir obtenu $k-1$ mauvaises réponses, puis la bonne réponse. L'espérance du nombre de répétitions est donc $\sum_{k\geq 1} kp(1-p)^{k-1} = 1/p$ (calcul laissé en exercice).
2. On obtient soit la bonne réponse au premier essai (avec probabilité p), soit il faut recommencer et l'espérance du nombre d'étapes restantes est encore $\mathbb{E}[X]$ puisque les exécutions sont indépendantes. On obtient l'équation $\mathbb{E}[X] = p \times 1 + (1-p) \times (1 + \mathbb{E}[X])$ dont la résolution donne $\mathbb{E}[X] = 1/p$.
3. On peut formaliser le raisonnement précédent avec la formule des espérances totales : $\mathbb{E}[X] = \mathbb{P}[X = 1]\mathbb{E}[X|X = 1] + \mathbb{P}[X > 1]\mathbb{E}[X|X > 1]$. On retrouve l'équation ci-dessus en notant que $\mathbb{P}[X = 1] = p$, $\mathbb{E}[X|X = 1] = 1$, $\mathbb{P}[X > 1] = 1-p$ et $\mathbb{E}[X|X > 1] = 1 + \mathbb{E}[X]$.

□

Remarque. La dernière égalité $\mathbb{E}[X|X > 1] = 1 + \mathbb{E}[X]$ mérite justification. Intuitivement, cela est justifié par le fait que les exécutions étant indépendantes, la première exécution incorrecte ne change rien à la probabilité d'obtenir des exécutions correctes dans l'avenir. Formellement, cela découle de l'égalité $\mathbb{P}[X \geq k|X > 1] = \mathbb{P}[X \geq k-1]$. Si on suppose l'égalité,

$$\mathbb{E}[X|X > 1] = \sum_{k>0} \mathbb{P}[X \geq k|X > 1] = \sum_{k>0} \mathbb{P}[X \geq k-1] = 1 + \sum_{k>0} \mathbb{P}[X \geq k] = 1 + \mathbb{E}[X].$$

L'égalité elle-même provient de la définition de probabilité conditionnelle : $\mathbb{P}[X \geq k|X > 1] = \mathbb{P}[X \geq k \wedge X > 1]/\mathbb{P}[X > 1]$. Comme $k > 1$, $\mathbb{P}[X \geq k \wedge X > 1] = \mathbb{P}[X \geq k] = (1-p)^{k-1}$ puisque $X \geq k$ signifie simplement que les $(k-1)$ premières exécutions étaient incorrectes, et $\mathbb{P}[X > 1] = 1-p$. Ainsi, $\mathbb{P}[X \geq k|X > 1] = (1-p)^{k-2} = \mathbb{P}[X \geq k-1]$.

Ce second lemme permet d'améliorer la probabilité de succès par répétition de l'algorithme de type Monte Carlo. En fonction des situations, on utilisera l'un ou l'autre des énoncés de ce lemme.

Lemme 3.3. *Supposons qu'un algorithme de type Monte Carlo renvoie un résultat correct avec probabilité p, et qu'on effectue N exécutions indépendantes de l'algorithme.*

— *La probabilité qu'une des exécutions renvoie un résultat correct est $1-(1-p)^N \geq 1-e^{-pN}$.*

— *L'espérance du nombre d'exécutions correctes est pN.*
— *Si $p > 1/2$, la probabilité que la majorité des exécutions soient correctes est $\geq 1 - e^{-pN/2}$.*

Démonstration. Pour $k = 1$ à N, soit X_k la variable aléatoire qui indique si la $k^{\text{ème}}$ exécution de l'algorithme est correcte : $X_k = 1$ si le résultat est correct, et $X_k = 0$ sinon. Les variables aléatoires $X_1, \ldots, X_N$ sont indépendantes puisque les exécutions sont elles-mêmes indépendantes.

La probabilité que toutes les exécutions renvoient une erreur est

$$\mathbb{P}\left[\bigwedge_{k=1}^{N} X_k = 1\right] = \prod_{k=1}^{N} \mathbb{P}[X_k = 1] = (1-p)^N \leq e^{-pN}$$

où l'inégalité est déduite de l'inégalité $1 + x \leq e^x$. Cela démontre le premier point.

Pour le deuxième, on utilise simplement la linéarité de l'espérance sur la variable aléatoire $X = X_1 + \cdots + X_N$ qui compte le nombre d'exécutions correctes : $\mathbb{E}[X] = \sum_k \mathbb{E}[X_k] = pN$.

Pour le dernier point, on utilise l'inégalité de Chernoff simplifiée $\mathbb{P}[X \leq (1-\delta)\mathbb{E}[X]] \leq e^{-\mathbb{E}[X]/2}$i, valable pour $0 \leq \delta \leq 1$, à nouveau avec la variable aléatoire $X = X_1 + \cdots + X_N$. On obtient

$$\mathbb{P}[X \leq N/2] = \mathbb{P}[X \leq (1-(1-1/2p))\mathbb{E}[X]] \leq e^{-\mathbb{E}[X]/2} = e^{-pN/2}.$$

□

Exercice. Les deux questions portent sur le troisième point du lemme.

1. Pourquoi doit-on supposer $p > 1/2$? Donner deux explications, intuitive et mathématique.
2. Quelle borne obtient-on si on utilise la version plus précise de l'inégalité de Chernoff ?

À retenir :

Las Vegas : toujours correct, souvent rapide $\rightarrow$ espérance du temps de calcul
Monte Carlo : toujours rapide, souvent correct $\rightarrow$ probabilité de succès
Monte Carlo $\Longrightarrow$ Las Vegas (sous certaines hypothèses)
Amplification : répéter un algorithme Monte Carlo améliore la probabilité de succès
Bornes (Markov, Chernoff, ...) : plus précis que l'espérance