Chap. 3 – Analyse amortie, analyse d'algorithmes probabilistes

HAI503I – Algorithmique 4

Bruno Grenet

Université de Montpellier – Faculté des Sciences

Incrémenter un entier de 0 à $2^k - 1$

Représentation

- Tableau T de k bits (ou *mot binaire de longueur k*)
- Entier N représenté : $\sum_{i=0}^{k-1} T_{[i]} 2^i$

Incrément(T):

Entrée: Tableau T de taille k représentant un entier N

Sortie: Le même T, représentant $N + 1$ *modulo* 2^k $\qquad (2^k - 1) + 1 \to 0$

1. $i \leftarrow 0$
2. Tant que $i < k$ et $T_{[i]} = 1$:
3. $\quad T_{[i]} \leftarrow 0$
4. $\quad i \leftarrow i + 1$
5. Si $i < k$: $T_{[i]} \leftarrow 1$
6. Renvoyer T

Propriétés d'Incrément

Correction

- Si T représente N, alors après Incrément, T représente $N' = N + 1 \bmod 2^k$

Preuve

- Si $N = 2^k - 1$, $T_{[i]} = 1$ pour tout i et après incrément $T_{[i]} = 0$ pour tout i
- Sinon, soit i tel que $T_{[i]} = 0$ et $T_{[j]} = 1$ pour $j < i$:
  - Après Incrément : $T_{[i]} = 1$, $T_{[j]} = 0$ pour $j < i$ et $T_{[k]}$ inchangé pour $k > i$
  - Donc $N' = N + 2^i - \sum_{j<i} 2^j = N + 1$

Complexité

- Incrément a complexité $O(k)$

Preuve

- Pire cas $\rightarrow$ on parcourt une fois tout le tableau T

Peut-on dire mieux ?

La complexité d'Incrément est-elle *vraiment* $O(k)$?

- $01\ldots11 \to 10\ldots00$: demande effectivement k *inversions* de bits
- $10\ldots00 \to 10\ldots01$: ne demande qu'une inversion de bit !

Comment prendre en compte les variations ?

- Les Incréments peuvent coûter 1, 2, …, k
- Lesquels sont les plus *fréquents* ?

$\to$ Fixer une suite d'Incréments

Suite d'Incréments

On incrémente T de 0 à $N-1$: quel est le coût *global* ?

Analyse *pire cas*

- T est de taille $k \rightarrow$ chaque Incrément coûte $O(k)$
- On effectue N Incréments $\rightarrow$ coût global $O(Nk)$
- Remarque : si $N \ll 2^k$, chaque Incrément coûte $O(\log N) \rightarrow O(N \log N)$

Analyse *amortie*

- $T_{[0]}$ est inversé à chaque fois
- $T_{[1]}$ est inversé une fois sur deux
- ...
- $T_{[k-1]}$ est inversé une fois sur 2^{k-1}

$\rightarrow$ Coût global : $\sum_{i=0}^{k-1} \lfloor \frac{N}{2^i} \rfloor < N \sum_{i=0}^{+\infty} \frac{1}{2^i} = 2N$

Bilan sur Incrément

Coût d'un appel à Incrément

- Pire cas : on doit parcourir tout le tableau $T \to O(k)$
- On ne peut pas dire mieux *a priori* !

Coût de N appels à Incréments

- Pire cas : $N \times O(k) = O(Nk)$
- Coût global : $O(N)$ car certains Incréments peu chers
- Remarque : valable aussi pour N Incréments quelconques

Coût *amorti* d'Incrément

Le coût amorti de l'algorithme Incrément est $O(1)$ par appel à Incrément

Analyse pire cas et analyse amortie

Scénario

- Algorithme Algo de complexité $C(n)$ pour une entrée de taille n, *dans le pire cas*
- Séquence de N appels à Algo : coût $c_i \leq C(n)$ sur l'entrée n°i

Deux analyses possibles

- Analyse pire cas : le coût global est borné par $N \times C(n)$
- Analyse amortie : le coût global est $\leq \sum_{i=1}^{N} c_i$

Remarques

- L'analyse pire cas reste valide ; l'analyse amortie est meilleure
- Estimation directe du coût c_i difficile, voire impossible
- Plusieurs méthodes d'analyse :
  - méthode de l'agrégat
  - méthode de l'accompte
  - méthode du potentiel

Méthode de l'agrégat

Idée : si le coût global pour N appels est $C^{tot}(N)$, le coût amorti est $C^{tot}(N)/N$

- Agrégat : mot compliqué pour une idée simple $\rightarrow$ on somme les coûts et on divise

Mise en œuvre

- Regarder globalement les N appels comme une seule exécution
- Regrouper des opérations venant de différents appels pour mieux compter

Exemple pour Incrément

- Compter le nombre total d'inversions du bit $T_{[0]}$, du bit $T_{[1]}$, etc.

Méthode de l'accompte

Idée : payer plus que le *vrai* coût à certains appels, et moins à d'autres

- Accompte : on imagine que les coûts sont de l'argent, et le compte doit être en positif

Mise en œuvre

- À chaque appel,
  - fixer une taxe à payer (éventuellement nulle pour certains appels)
  - utiliser l'accompte pour payer le coût de l'appel
- L'accompte doit toujours rester positif
- Coût amorti par opération : taxe maximale payée
- Remarque : plus difficile que l'agrégat, mais plus puissant

Exemple pour Incrément

- Chaque passage de bit de 0 à 1 coûte 2, et chaque passage de 1 à 0 est gratuit
- À chaque appel : prélèvement de 1 par inversion de bits
- Coût amorti : 2

Méthode du potentiel

Idée : associer aux appels les plus chers une augmentation de *potentiel*

- Potentiel : métaphore de l'*énergie potentielle* en physique

Mise en œuvre

- Définir une *fonction potentiel* $\Phi \geq 0$ sur l'objet manipulé
  - Valeur initiale Φ_0
  - Valeur après i appels : $\Phi_i \geq \Phi_0$
- Si le coût d'un appel est c_i, son *coût amorti* est $a_i = c_i + \Phi_i - \Phi_{i-1}$
- Le coût total amorti de N appels est $\sum_{i=1}^{N} a_i = \sum_{i=1}^{N} c_i + \Phi_N - \Phi_0$

Exemple pour Incrément

- Potentiel du tableau T : $\Phi(T) =$ nombre de 1 dans T
- Si Incrément(T) remet ℓ bits à 0 :
  - coût $c_i = \ell + 1$
  - différence de potentiel : $\Phi_i - \Phi_{i-1} = \ell - 1$
  - coût amorti : $\ell + 1 - (\ell - 1) = 2$

Bilan sur les trois méthodes

Techniques plus ou moins faciles

- Méthode de l'agrégat : idée la plus évidente… mais demande une compréhension globale
- Méthodes de l'accompte et du potentiel : plus difficile à mettre en œuvre, mais compréhension *locale*

Idées communes aux méthodes de l'accompte et du potentiel

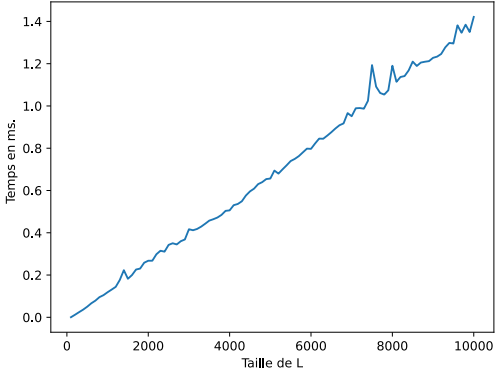
- Calcul direct d'un coût amorti pour chaque appel
- Preuve globale que le coût amorti défini est *valide*
- Forme d'analyse pire cas avec une notion de coût modifiée

Utilisation principale : structures de données

- Ensemble d'algorithmes de manipulation de la structure (ajout, suppression, etc.)
- Coûts variables → analyse amortie pour avoir un *coût moyen par opérations*

Exemple des `list` en Python

```
def test(n):
    L = []
    for i in range(n): L.append(i)
    for i in range(n//2): L[i], L[n-i-1] = L[n-i-1], L[i]
```



Quelle structure de données ?

- Ajout en fin de liste en $O(1)$ $\rightarrow$ liste chaînée ?
- Accès à $L_{[i]}$ en temps $O(1)$ $\rightarrow$ tableau ?

Les tableaux dynamiques

Idée de base

- Structure de donnée sous-jacente : un tableau
- Deux tailles :
  - taille effective N du tableau en mémoire
  - nombre n d'éléments stockés

Conditions à respecter

- Il faut toujours $N \geq n$ pour avoir assez de place
- Il ne faut pas $N \gg n$: utilisation de place inutile

Objectifs

- Assurer $N = O(n) \rightarrow$ en pratique $n \leq N \leq 4n$
- Accès à un élément en temps $O(1)$: immédiat
- Ajout et suppression en fin de tableau en $O(1)$

Ajout et suppression

Ajout d'un élément x à la fin

- Si $N > n$: $T_{[n]} \leftarrow x$; $n \leftarrow n + 1$
- Sinon, doubler la taille de T :
  - Nouveau tableau U de taille $2N$
  - Recopie de T dans U
  - Ajout de x à U

Suppression d'un élément x à la fin

- Pas de difficulté : $n \leftarrow n - 1$
- Pour éviter $N \gg n$, il faut (parfois) réduire la taille de T
  - Idée 1 : si $n < N/2$ on réduit de moitié $\rightarrow$ mauvaise idée !
  - Idée 2 : si $n < N/4$ on réduit de moitié $\rightarrow$ bonne idée !

Remarque

- N toujours ≥ 1
  - Suppression du dernier élément : pas de modification de T

Les algorithmes

AJOUT(T, N, n, x) :

1. Si $n < N$:
2. $\quad T_{[n]} \leftarrow x$
3. $\quad n \leftarrow n + 1$
4. $\quad$ Renvoyer (T, N, n)
5. $U \leftarrow$ tableau de taille $2N$
6. Pour $i = 0$ à $N - 1$: $U_{[i]} \leftarrow T_{[i]}$
7. $U_{[n]} \leftarrow x$
8. $(N, n) \leftarrow (2N, n + 1)$
9. Renvoyer (U, N, n)

SUPPRESSION(T, N, n) :

1. Si $n = 1$ ou $n > N/4$:
2. $\quad n \leftarrow n - 1$
3. $\quad$ Renvoyer (T, N, n)
4. $U \leftarrow$ tableau de taille $N/2$
5. Pour $i = 0$ à $n - 2$: $U_{[i]} \leftarrow T_{[i]}$
6. $(N, n) \leftarrow (N/2, n - 1)$
7. Renvoyer (U, N, n)

Dans le pire cas, AJOUT et SUPPRESSION effectuent chacun $O(n)$ affectations

Analyse amortie 1 : uniquement des Ajouts

Coût de m Ajouts dans un tableau initialement vide ?

Analyse pire cas

- Un Ajout dans un tableau de taille k coûte $O(k) \rightarrow$ coût total $O(m^2)$

Méthode de l'agrégat

- Deux types de coût :
  - Affectations $T_{[n]} \leftarrow x$ quand on Ajoute x
  - *Réaffectations* quand on double la taille de T
- $N = 1$ initialement, et on double la taille quand nécessaire $\rightarrow N = 2^k$
- Taille de T doublée quand n est une puissance de 2

$\rightarrow$ coût total des réaffectations : $\sum_{k=1}^{\lfloor \log m \rfloor} 2^k < 2^{\lfloor \log m \rfloor + 1} \leq 2m$

Théorème

Le coût amorti de m Ajouts dans un tableau initialement vide est de 3 affectations par opération

Analyse amortie 2 : Ajouts et Suppressions

Coût de m opérations Ajout/Suppression dans un tableau initialement vide ?

Notations

Après la $i^{\text{ème}}$ opération,

- n_i : nombre d'élément dans le tableau
- N_i : taille du tableau
- $\alpha_i = n_i/N_i$: *coefficient de remplissage*
- c_i : coût de la $i^{\text{ème}}$ opération (nombre d'affectations)

Fonction potentiel

$$\Phi_i = \begin{cases} 2n_i - N_i & \text{si } \alpha_i \geq \frac{1}{2} \\ N_i/2 - n_i & \text{si } 0 < \alpha_i \leq \frac{1}{2} \\ 0 & \text{si } \alpha_i = 0 \end{cases}$$

Objectif : Montrer que le coût amorti $a_i = c_i + \Phi_i - \Phi_{i-1}$ de chaque opération est constant

Preuve de l'analyse amortie

Le coût amorti $a_i = c_i + \Phi_i - \Phi_{i-1}$ de la $i^{\text{ème}}$ opération est ≤ 3 pour tout i

Ajout : $n_i = n_{i-1} + 1$

1. $\alpha_{i-1}, \alpha_i < 1/2$: $1 + \left(N_i/2 - n_i\right) - \left(N_{i-1}/2 - n_{i-1}\right) = 0$
2. $\alpha_{i-1} < 1/2 \leq \alpha_i$: $1 + \left(2n_i - N_i\right) - \left(N_{i-1}/2 - n_{i-1}\right) = 3 + 3n_{i-1} - \frac{3}{2}N_{i-1} \leq 3$
3. $\alpha_{i-1}, \alpha_i \geq 1/2$ mais pas doublement : $1 + \left(2n_i - N_i\right) - \left(2n_{i-1} - N_{i-1}\right) = 3$
4. $\alpha_{i-1} = 1$ et $\alpha_i \geq 1/2$: $\left(n_{i-1}+1\right) + \left(2n_i - N_i\right) - \left(2n_{i-1} - N_{i-1}\right)$
   $= n_{i-1} + 3 - 2N_{i-1} + N_{i-1} = n_{i-1} + 3 - N_{i-1} = 3$

✓

Preuve de l'analyse amortie

Le coût amorti $a_i = c_i + \Phi_i - \Phi_{i-1}$ de la $i^{\text{ème}}$ opération est ≤ 3 pour tout i

SUPPRESSION $n_i = n_{i-1} - 1$

1. $\alpha_i, \alpha_{i-1} \geq 1/2$: $0 + (2n_i - N_i) - (2n_{i-1} - N_{i-1}) = -2$
2. $\alpha_{i-1} \geq 1/2 > \alpha_i$: $0 + (N_i/2 - n_i) - (2n_{i-1} - N_{i-1}) = 1 - 3n_{i-1} + \frac{3}{2}N_{i-1} \leq 1$
3. $\alpha_{i-1}, \alpha_i < 1/2$ mais de réduction : $0 + (N_i/2 - n_i) - (\frac{N_{i-1}}{2} - n_{i-1}) = 1$
4. $\alpha_{i-1} = 1/4$, $\alpha_i \leq 1/2$: $n_i + (N_i/2 - n_i) - (\frac{N_{i-1}}{2} - n_{i-1}) = n_{i-1} - \frac{N_{i-1}}{4} = 0$

Bilan sur les tableaux dynamiques

Principes

- Tableau de taille variable
  - Mémoire *allouée* supérieure à celle utilisée
  - Remplissage : $\frac{1}{4} \leq \alpha \leq \frac{1}{2}$
  - Taille doublée ou divisée par deux quand nécessaire
- Accès direct et AJOUT *en fin de tableau* en temps constant

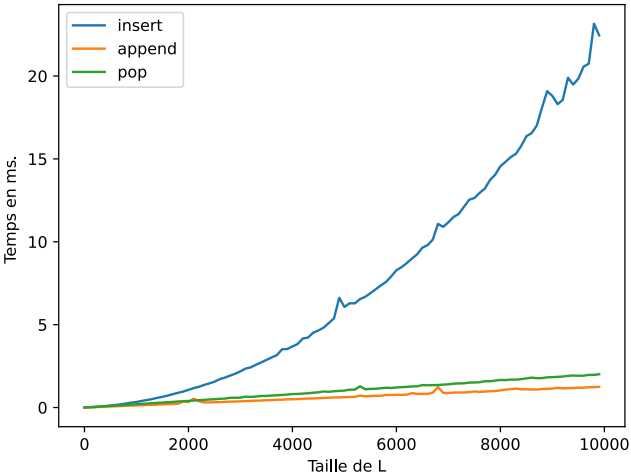
Complexité amortie

- Chaque opération coûte ≤ 3 affectations $\rightarrow$ coût constant par opération
- Mais tout de même : si on connaît à l'avance la taille, coût triplé !

Autres utilisations

- Création de pile $\rightarrow$ idem !
- Création de file $\rightarrow$ travail supplémentaire, *cf* TD

Performance des `list` Python



- Insertion en début de tableau
- Insertion en fin de tableau
- Suppression en fin de tableau

Conclusion sur l'analyse amortie

Technique avancée d'analyse d'algorithmes

- Dépasser l'analyse *pire cas*
- Prendre en compte les variations de temps entre différents appels

Trois techniques

- Méthode de l'agrégat
- Méthode de l'accompte
- Méthode du potentiel

Utilisation principale : structures de données

- Chaque opération *peut* coûter cher
- Mais peu d'opérations coûtent cher
- Si on utilise plusieurs fois la structure de donnée $\rightarrow$ coût amorti faible

Problème de la sélection

QuickSelect(T, k) :

Entrées : Tableau T de taille n d'entiers tous distincts ; entier k entre 1 et n
Sortie : Le $k^{\text{ème}}$ plus petit élément $T^{(k)}$ de T

1. $p \leftarrow T_{[i]}$ avec i choisi aléatoirement entre 0 et $n-1$ *(pivot)*
2. $n_0 \leftarrow$ nombre d'élément $< p$ dans T *(boucle Pour)*
3. Si $n_0 = k-1$: Renvoyer p
4. Si $n_0 \geq k$:
5. $\quad T_0 \leftarrow$ tableau des éléments de T qui sont $< p$ *(boucle Pour)*
6. $\quad$ Renvoyer QuickSelect(T_0, k)
7. $T_1 \leftarrow$ tableau des éléments de T qui sont $> p$ *(boucle Pour)*
8. Renvoyer QuickSelect$(T_1, k - n_0 - 1)$

Correction

$$T^{(k)} = \begin{cases} p & \text{si } n_0 = k-1 \\ T_0^{(k)} & \text{si } n_0 \geq k \\ T_1^{(k-n_0-1)} & \text{sinon} \end{cases}$$

Complexité de QuickSelect

Analyse en pire cas

- Deux boucles en $O(n)$ + appel récursif sur un tableau de taille $\leq n-1$
- $C_n \leq C_{n-1} + O(n) \to C_n = O(n^2)$ *nombre de comparaisons*

Peut-on dire mieux ?

- Appel récursif en taille $n-1$ si $T_{[i]}$ est toujours le minimum ou toujours le maximum
- Est-ce que ça arrive *en pratique* ?
  - Quelle est la taille *moyenne* (espérance) du tableau pour l'appel récursif ?
  - Quelle est l'espérance du nombre de comparaisons effectuées ?

Théorème

Soit C_n le nombre de comparaisons effectuées par QuickSelect(T, k) où T est de taille n.
Alors $\mathbb{E}[C_n] \leq 4n$, quelque soit k.

Preuve par récurrence

- C_n = nb de comparaisons $\quad \mathbb{E}[C_n] \leq 4n \quad n=1$ ✓

- $C_n \rightsquigarrow n$ + appel récursif $= n + C_t$ où t = taille du tableau de l'appel récursif
  $\leq n + 4t$

- t ? - Si $p = T(k)$: $t = 0$
  $p = T(j)$ avec $j > k$: $t = j-1$
  $p = T(j)$ avec $j < k$: $t = n-j$
  $\Big\} \; t \leq \max(j-1, n-j)$

- $\mathbb{E}[C_n] = \sum_{j=1}^{n} \underbrace{\mathbb{E}[C_n \mid p = T(j)]}_{\leq n + 4\max(j-1, n-j)} \underbrace{\Pr[p = T(j)]}_{1/n} \leq n + \frac{4}{n} \underbrace{\sum_{j=1}^{n} \max(j-1, n-j)}_{S}$

Supp n pair

$$S = \sum_{j=1}^{n/2} n-j + \sum_{j=n/2+1}^{n} j-1 = \sum_{\ell=n/2}^{n-1} \ell + \sum_{\ell=n/2}^{n-1} \ell = 2\left(\sum_{\ell=1}^{n-1} \ell - \sum_{\ell=1}^{n/2-1} \ell\right) = n(n-1) - \frac{n}{2}\left(\frac{n}{2}-1\right)$$
$$= \frac{3n^2}{4} - \frac{n}{2}$$

$$\mathbb{E}[C_n] \leq n + \frac{4}{n}\left(\frac{3n^2}{4} - \frac{n}{2}\right) = n + 3n - 2 < 4n \quad ✓$$

Bilan sur QuickSelect

Nombre de comparaisons

- Si on est **très** malchanceux, QuickSelect effectue $\sim \frac{1}{2}n^2$ comparaisons
- L'espérance du nombre de comparaisons effectuées est $\leq 4n = O(n)$

Que veut dire *malchanceux* ?

- Espérance valable *quelque soit l'entrée* (T et k)
  - Pas de chance ou malchance par rapport à l'entrée
  - La probabilité porte sur les choix aléatoires de l'algorithme
- On peut être *parfois* malchanceux au cours de l'algorithme
  - Pas besoin d'avoir de la chance à chaque étape…
  - … simplement de ne pas être très malchanceux à chaque étape
- Exemple : proba. de faire toujours le pire choix $= 1/n!$
  - si $n = 10$: $1/3\,628\,800$
  - si $n = 100$: $< 1/10^{157}$

QuickSelect est toujours correct, l'espérance de sa complexité est $O(n)$

Coupe minimale dans un graphe

Définition

- Une *coupe* dans un graphe $G = (S, A)$ est une partition de $S = S_1 \sqcup S_2$
- La *taille* de la coupe $S_1 \sqcup S_2$ est le nombre d'arêtes entre S_1 et S_2 :
  $|\{u_1u_2 \in A : u_1 \in S_1, u_2 \in S_2\}|$

Problème de la coupe minimale

Entrée : Graphe $G = (S, A)$
Sortie : Coupe $S = S_1 \sqcup S_2$ de taille minimale

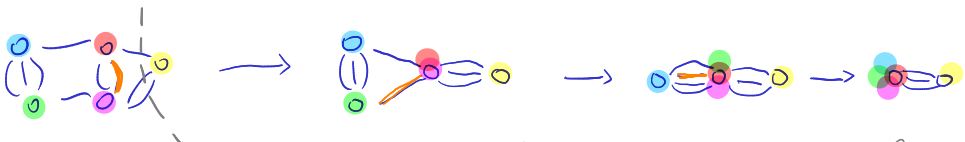
Généralisation nécessaire : multigraphes

- Un *multigraphe* est un graphe qui autorise plusieurs arêtes entre 2 sommets
- Coupe et problème de la coupe minimale définis de manière équivalente

Algorithme probabiliste pour la coupe minimale

CoupeMin(G):

1. Tant que G possède au moins 3 sommets :
2. Choisir une arête de G, aléatoirement
3. *Contracter* l'arête G
4. Renvoyer la coupe définie par les deux sommets restants



Complexité de CoupeMin

G : multigraphe à n sommets

Fait admis

Si G est représenté par listes d'adjacence, la *contraction* d'une arête peut s'effectuer en temps $O(n)$

Théorème

L'algorithme CoupeMin renvoie une coupe de G en temps $O(n^2)$

Preuve

- À chaque itération, on contracte une arête $\rightarrow$ le nombre de sommets diminue de 1
- Le nombre d'itérations est donc $\leq n-2$
- La complexité totale est $O(n^2)$

La complexité ne dépend pas des choix probabilistes !

Correction de CoupeMin

Pourquoi cet algorithme fonctionnerait-il ?

Lemme de correction

L'algorithme CoupeMin appliqué à un multigraphe à n sommets renvoie une coupe *minimale* avec probabilité $\geq \frac{2}{n(n-1)}$

Remarque

- Cette probabilité est très petite !
- Exemple pour $n = 100$, $\frac{2}{n(n-1)} \simeq 0,02\%$…

Preuve du lemme de correction

- Soit C^* une coupe minimale : CoupeTin renvoie $C^* \iff$ aucune arête de C^* n'est contractée pendant l'algo.

- On se place à une étape où il reste k sommets et aucune arête de C^* n'a été contractée.

  La proba d'en contracter une maintenant est $\leq 2/n$.

  $m = \#$ arêtes dans C^*. Alors le graphe possède $\geqslant \frac{nk}{2}$ arêtes.

  Car si tous les sommets ont degré $\geqslant m$.

  $\Rightarrow$ proba $\leq \frac{m}{nk/2} = \frac{2}{n}$ ☑

- Proba de ne jamais contracter une arête de $C^* \geqslant \left(1-\frac{2}{n}\right)\left(1-\frac{2}{n-1}\right)\cdots\left(1-\frac{2}{3}\right)$

  $= \frac{n-2}{n}\,\frac{n-3}{n-1}\,\frac{n-4}{n-2}\cdots\frac{1}{3} = \frac{2\cdot 1}{n(n-1)}$

Répétitions de CoupeMin

Probabilité de *succès* très faible $\rightarrow$ on répète l'algorithme pour améliorer cette probabilité

Lemme de répétition

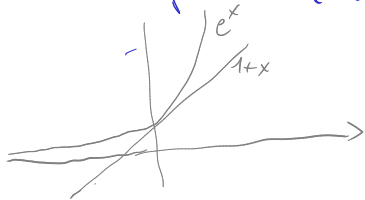
Si on répète N fois CoupeMin et qu'on garde la plus petite coupe renvoyée, cette coupe est minimale avec probabilité $\geq 1 - e^{-2N/n(n-1)}$

Remarques

- Si on répète $N = 2n^2$ fois l'algorithme, on obtient
  - une complexité $O(n^4)$
  - une coupe minimale avec probabilité $\geq 1 - e^{-4} \simeq 98\%$

Preuve du lemme de répétition

- Au $i^{ème}$ essai, proba de ne pas trouver une coupe minimale est $< 1 - \frac{2}{n(n-1)}$
- Proba de ne jamais trouver une coupe minimale est $\leq \left(1 - \frac{2}{n(n-1)}\right)^N$
- Maths : $\boxed{1 + x \leq e^x}$



$$1 - \frac{2}{n(n-1)} \leq e^{-2/n(n-1)}$$

↳ Proba de ne jamais réussir est $\leq e^{-2N/n(n-1)}$

Bilan sur CoupeMin

Complexité

- Un appel à CoupeMin coûte toujours $O(n^2)$
- On a besoin de $O(n^2)$ *répétitions* $\rightarrow O(n^4)$

Correction

- Un appel à CoupeMin renvoie une coupe minimale avec proba. $\geq \frac{2}{n(n-1)}$
- N appels à CoupeMin renvoient une coupe minimale avec proba. $\geq 1 - e^{-2N/n(n-1)}$
- cn^2 appels à CoupeMin renvoient une coupe minimale avec proba. $\geq 1 - e^{-2c}$

CoupeMin répété est toujours en temps $O(n^4)$, et correct avec très bonne probabilité

Définition

Un algorithme probabiliste est un algorithme qui effectue des choix aléatoires au cours de son exécution.

Remarque

- Choix aléatoires : accès à un générateur de bits aléatoires, ou d'entiers aléatoires, etc.

Analyse d'un algorithme probabiliste

- Le comportement d'un algorithme probabiliste est une *expérience probabiliste*
- Le résultat renvoyé (*correction*) ou le nombre d'opérations effectuées (*complexité*) peuvent dépendre des choix aléatoires $\rightarrow$ deux familles d'algorithmes

Les algorithmes de type *Las Vegas*

Définition

- Un algorithme probabiliste est de type Las Vegas si
  - son résultat ne dépend pas des choix aléatoires
  - sa complexité dépend des choix aléatoires
- Étude de la complexité :
  - Modélisée par une variable aléatoire
  - Calcul de l'espérance de la variable aléatoire
- Exemple : QuickSelect

Signification de l'espérance de la complexité

« L'espérance de la complexité est $O(n^2)$ »

- On *s'attend* à avoir une exécution en temps $O(n^2)$
- Si on exécute l'algorithme de nombreuses fois (sur la même entrée), le temps de calcul moyen sera $O(n^2)$

Las Vegas : « toujours correct, souvent rapide »

Les algorithmes de type *Monte Carlo*

Définition

- Un algorithme probabiliste est de type Las Vegas
  - si sa complexité ne dépend pas des choix aléatoires
  - son résultat dépend des choix aléatoires
- Étude de la correction :
  - *probabilité de succès* : probabilité que l'algorithme soit correct
- Exemple : Coupe Tim répété

Améliorer la probabilité de succès

Si la probabilité de succès est p et qu'on répète N fois l'algorithme

- probabilité qu'une (au moins) des répétitions soit correcte est $1-(1-p)^N \geq 1-e^{-pN}$

Monte Carlo : « toujours rapide, souvent correct »

Bilan sur les algorithmes probabilistes

Pourquoi des algorithmes probabilistes ?

- Algorithmes souvent simples et efficaces
- Parfois : meilleure complexité que les algorithmes déterministes
- *Et pourquoi pas ?* $\rightarrow$ en pratique, ils fonctionnent très bien !

Analyse des algorithmes probabilistes

- Modélisation probabiliste, avec variable aléatoire
- *Las Vegas* : étude de l'espérance de la complexité
- *Monte Carlo* : étude de la probabilité de succès

Et ensuite ?

- *Atlantic City* : « souvent correct, souvent rapide »
- Algorithmes quantiques : généralisation des algorithmes probabilistes

Algorithmes probabilistes parfois difficiles à analyser… mais indispensables !

Le tri rapide

TriRapide(T) :

1. Si taille(T) $= 1$: renvoyer T
2. $p \leftarrow T_{[i]}$ avec i choisi aléatoirement entre 0 et $n-1$ *(pivot)*
3. $n_p \leftarrow$ nombre d'indices i tels que $T_{[i]} = p$ *(boucle Pour)*
4. $T_0 \leftarrow$ tableau des éléments de T qui sont $\leq p$ *(boucle Pour)*
5. $T_1 \leftarrow$ tableau des éléments de T qui sont $> p$ *(boucle Pour)*
6. $T_0 \leftarrow$ TriRapide(T_0)
7. $T_1 \leftarrow$ TriRapide(T_1)
8. Renvoyer la concaténation T_0, n_p fois p, et T_1

Correction (idée)

- Par récurrence sur la taille n de T ($n = 1$: ok)
- T_0 et T_1 sont de taille $< n \rightarrow T_0$ et T_1 sont correctement triés
- donc le tableau renvoyé est correctement trié

Espérance du nombre de comparaisons

Théorème

L'espérance du nombre de comparaisons effetuées par TriRapide est $O(n \log n)$

Notations

- $T^{(i)}$: $i^{\text{ème}}$ plus petit élément de T
- $X_{ij} = \begin{cases} 1 & \text{si } T^{(i)} \text{ est comparé à } T^{(j)} \text{ au cours de l'algorithme} \\ 0 & \text{sinon} \end{cases}$
- X : nombre total de comparaisons $\rightarrow X = \sum_{i<j} X_{ij}$

Lemme

Pour $1 \leq i < j \leq n, \mathbb{E}[X_{ij}] = \Pr[X_{ij} = 1] = 2/(j - i + 1)$

Preuve du théorème

- $\mathbb{E}[X] = \sum_{i<j} \mathbb{E}[X_{ij}] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} = \sum_{i=1}^{n} \sum_{k=2}^{n-i+1} \frac{2}{k} \leq \sum_{i=1}^{n} \sum_{k=1}^{n} \frac{2}{k}$
- $\sum_{k=1}^{n} 1/k = O(\log n)$
- Donc $\mathbb{E}[X] = O(n \log n)$

Preuve du lemme

Bilan sur le TriRapide

Propriétés du TriRapide

- L'algorithme est toujours correct
- L'espérance de sa complexité est $O(n \log n)$
- Type d'algorithme probabiliste : Las Vegas

Comportement pratique

- Le TriRapide est efficace en pratique, s'il est implanté *en place*
- Tri *stable*

Conclusion générale

Analyse amortie

- Analyse de plusieurs exécutions consécutives d'un même algorithme
- Complexité amortie = temps *moyen* pris par les exécutions successives
- Trois techniques de preuve : agrégat, accompte et potentiel

Analyse d'algorithmes probabilistes

- Analyse d'algorithmes qui font des choix aléatoires
- Étude de l'espérance de la complexité ou de la probabilité de succès
- Comportement *moyen* vis-à-vis des choix aléatoires

Conclusion générale

Analyse amortie

- Analyse de plusieurs exécutions consécutives d'un même algorithme
- Complexité amortie = temps *moyen* pris par les exécutions successives
- Trois techniques de preuve : agrégat, accompte et potentiel

Analyse d'algorithmes probabilistes

- Analyse d'algorithmes qui font des choix aléatoires
- Étude de l'espérance de la complexité ou de la probabilité de succès
- Comportement *moyen* vis-à-vis des choix aléatoires

Analyse en moyenne

- Analyse du comportement d'algorithmes sur des *entrées aléatoires*
- Calcul de l'espérance de la complexité sur une entrée aléatoire
- Question subtile : quelle distribution sur les entrées ?