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 $\qquad (2^k - 1) + 1 \to 0$

$i \leftarrow 0$
Tant que $i < k$ et $T_{[i]} = 1$:
    $T_{[i]} \leftarrow 0$
    $i \leftarrow i + 1$
Si $i < k$: $T_{[i]} \leftarrow 1$
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* $O(k)$?

- $01...11 \to 10...00$: demande effectivement k *inversions* de bits
- $10...00 \to 10...01$: 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 $k \to$ chaque Incrément coûte $O(k)$
- On effectue N Incréments $\to$ coût global $O(Nk)$
- Remarque : si $N \ll 2^k$, chaque Incrément coûte $O(\log N) \to O(N \log N)$

Analyse *amortie*

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

$\to$ 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 $T \rightarrow O(k)$
- 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

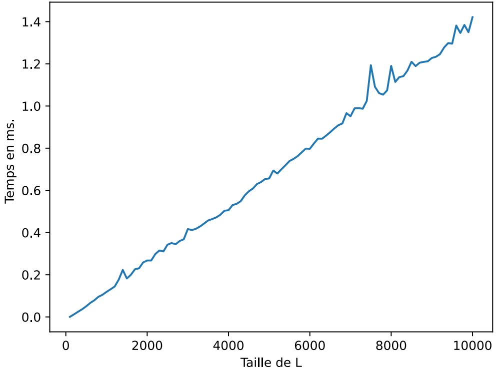
- Calcul direct d'un coût amorti pour chaque appel
- Preuve globale que le coût amorti défini est *valide*
- Forme d'analyse pire cas avec une notion de coût modifiée

Utilisation principale : structures de données

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

Exemple des `list` en Python

```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 $O(1) \to$ liste chaînée ?
- Accès à $L_{[i]}$ en temps $O(1) \to$ 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 $O(k) \to$ 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 $\to N = 2^k$
- Taille de T doublée quand n est une puissance de 2

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

$$= n_{i-1} + 3 - 2N_{i-1} + N_{i-1} = n_{i-1} + 3 - N_{i-1} = 3$$

✓

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 $\quad n_i = n_{i-1} - 1$

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 pas 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

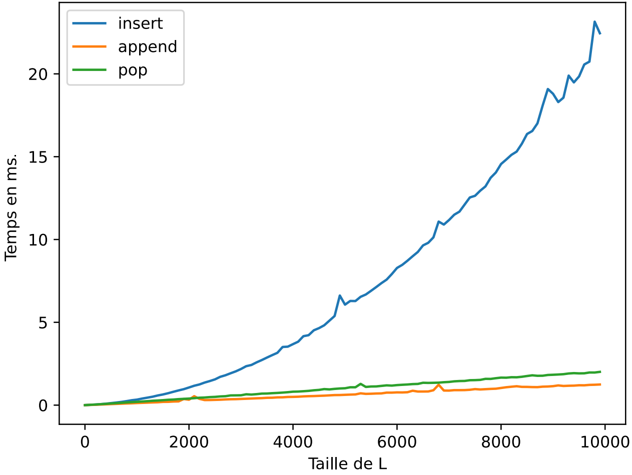
Complexité amortie

- 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