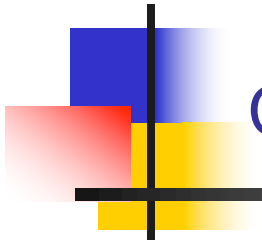
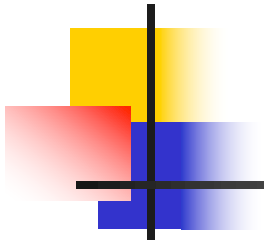


Chapitre 9

L'analyse amortie



Introduction

- Il arrive que les analyses en meilleur cas, en pire cas et en cas moyen ne soient pas suffisantes pour nous donner une idée exacte du temps d'exécution d'un algorithme.
- Supposons qu'un algorithme ait des efficacités en meilleur cas et pire cas de $C_{best}(n)$ et $C_{worst}(n)$ et que cet algorithme est appelé *k* fois. Nous pourrions être tenté de dire que l'algorithme qui en résulte a une efficacité de $k \times C_{best}(n)$ et $k \times C_{worst}(n)$ en meilleur et en pire cas.
- Or, lorsque l'on manipule une structure de données, il n'est pas certain que le meilleur cas ou le pire cas peuvent survenir lors de *k* appels successifs. C'est alors que l'on a recours à l'analyse amortie.
- L'analyse amortie permet d'amortir un appel coûteux, souvent le pire cas de l'algorithme, sur les appels précédents.
- Commençons par un exemple concret: l'analyse d'une fonction qui ajoute un élément à un vecteur.

Une classe vecteur

```
class Vecteur {
private:
  int* elements_;
  unsigned int capacite_;       // Capacite du vecteur elements_
  unsigned int taille_;         // Nombre d'elements ajoutes

public:
  Vecteur()
    : capacite_(1),
      taille_(0)
  { elements_ = new int[capacite_]; }

  // Ajoute un element a la fin du vecteur
  void ajoute(int element) {
    assert(taille_ <= capacite_);

    if (taille_ == capacite_) {
      capacite_ = 2 * capacite_;                // Double la capacite du vecteur
      int* t = new int[capacite_];
      for (unsigned int i = 0; i < taille_; i++) // Recopie chaque element
        t[i] = elements_[i];                    // dans le nouveau vecteur
      delete [] elements_;                      // Supprime l'ancien vecteur
      elements_ = t;                            // Utilise le nouveau vecteur
    }
    elements_[taille_++] = element;
  }

  // Retourne le i eme element du vecteur
  int operator[](unsigned int i) const {
    return elements_[i];
  }

  virtual ~Vecteur() {
    delete [] elements_;
  }
};
```

Analyse de la méthode `ajoute`

- La méthode `ajoute` écrit un élément dans le vecteur. Si ce vecteur est rempli à pleine capacité, alors on alloue un espace mémoire deux fois plus grand dans lequel on recopie les éléments du vecteur déjà ajoutés. Avec cette capacité doublée, il est maintenant possible d'ajouter l'élément au vecteur.
- Nous dénotons n la taille du vecteur avant l'ajout, c'est-à-dire l'équivalent de `taille_` avant l'appel à `ajoute`.
- L'opération de base est la copie d'un élément dans un vecteur. Elle se retrouve sur les lignes `t[i] = elements_[i]` et `elements[taille++] = element`.
- En meilleur cas, le vecteur n'est pas rempli à pleine capacité et la méthode `ajoute` s'exécute en temps $C_{best}(n) = 1 \in \Theta(1)$.
- En pire cas, le vecteur est rempli à pleine capacité et les n éléments du vecteur doivent être recopié afin d'ajouter le nouvel élément. $C_{worst}(n) = n + 1 \in \Theta(n)$.

Analyse d'appels successifs

```
void ajouteKElements(int k) {
  Vecteur entiers;
  for (int i = 1; i <= k; i++)
    entiers.ajoute(i);
}
```

- **Question**: Si on ajoute k éléments à un vecteur vide par le biais de k appels successifs à la méthode `ajoute`, devons-nous conclure que ces appels s'effectueront dans ces temps?

$$C_{best}(k) = \sum_{i=1}^{k} 1 = k \in \Theta(k)$$

$$C_{worst}(k) = \sum_{i=1}^{k} (i+1) = \frac{k(k+1)}{2} + k \in \Theta(k^2)$$

- **Réponse**: Non! Car nous ne pouvons pas affirmer que le meilleur cas ou le pire cas se produira k fois de façon successive.
- Alors analysons le temps d'exécution d'une séquence de k appels à la méthode `ajoute` sur un vecteur originellement vide.

Analyse d'appels successifs

- Le pire cas se produit lorsque n est une puissance de deux. Il se produit donc $\lfloor \log(k-1) \rfloor$ fois en k appels successifs.
- Nous démontrons ici que le temps d'exécution de la fonction `ajouteKElements` est linéaire.

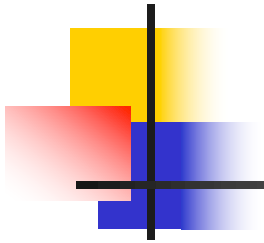
$$C_{\text{ajoute}}(n) = \begin{cases} n+1 & \text{si } n \text{ est une puissance de 2} \\ 1 & \text{sinon} \end{cases}$$

$$\begin{aligned} C_{\text{ajouteKElements}}(k) &= \sum_{n=0}^{k-1} C_{\text{ajoute}}(n) \\ &= k + \sum_{i=0}^{\lfloor \log_2(k-1) \rfloor} 2^i \\ &= k + 2^{\lfloor \log_2(k-1) \rfloor + 1} - 1 \end{aligned}$$

$$C_{\text{ajouteKElements}}(k) \leq k + 2^{\log_2(k-1)+1} - 1 = k + 2 \cdot 2^{\log_2(k-1)} - 1 = k + 2(k-1) - 1 \leq 3k \in O(k)$$

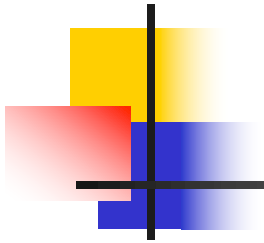
$$C_{\text{ajouteKElements}}(k) \geq k + 2^{\log_2(k-1)} - 1 = 2k - 2 \in \Omega(k)$$

$$C_{\text{ajouteKElements}}(k) \in \Theta(k)$$



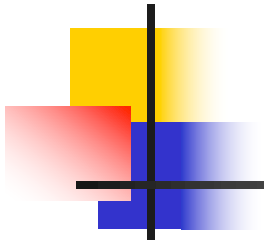
Conclusion de l'analyse

- Le fait qu'ajouter k éléments à un vecteur vide se fait en temps linéaire laisse croire que la fonction `ajouteElement` s'exécute en temps constant.
- Nous allons démontrer que le temps d'exécution d'`ajoueElement` peut être **amorti** sur les appels **précédents** et ainsi s'exécuter en temps constant.



L’analyse amortie

- L’analyse amortie consiste à analyser une fonction en permettant de comptabiliser du temps de calcul sur des appels antérieurs.
- Les temps de calcul passés sur les instances du pire cas pourront donc être amoindris sur les meilleures instances.
- Note: Il n’est pas permis d’amortir les temps d’exécution sur les appels futurs car rien ne garanti qu’une fonction sera appelée à nouveau.



La méthode du comptable

- La méthode du comptable est une technique d'analyse amortie qui consiste à attribuer à l'algorithme une banque de temps initialement vide: $B = 0$.
- Si l'algorithme s'exécute en $C_{réel}(n)$, on charge $C_{amorti}(n)$ à l'utilisateur de la fonction. La banque de temps est mise à jour par l'opération $B \leftarrow B + C_{amorti}(n) - C_{réel}(n)$.
- La banque de temps ne doit, en aucun cas, accumuler une dette: $B \geq 0$.
- Pour obtenir une bonne analyse amortie, il faut surcharger l'utilisateur ($C_{amorti}(n) > C_{réel}(n)$) quand le meilleur cas se produit et le sous-charger ($C_{amorti}(n) < C_{réel}(n)$) lorsque le pire cas se produit.
- Au final, l'efficacité amortie de l'algorithme est donnée par $C_{amorti}(n)$.

Analyse de `ajouteElement` par la méthode du comptable

- Lorsque la capacité n du tableau doit être doublée, il y a n/2 éléments qui sont copiés dans un nouveau tableau pour la première fois et n/2 éléments qui avaient déjà été recopiés lors de la dernière redimension.
- Nous fixons la fonction $C_{amorti}(n) = 2$ si $n = 0$ et $C_{amorti}(n) = 3$ sinon.
  - De ces unités de temps, une est utilisée pour ajouter l'élément.
  - Une unité est mise en banque afin de pouvoir copier le nouvel élément lorsque la capacité du tableau devra être doublée.
  - Si $n > 0$, la troisième unité est mise en banque pour copier un élément qui est déjà dans le tableau.
- Ainsi, lorsqu'on ajoute les éléments au tableau, on prend en considération le coût pour copier les n/2 anciens éléments et les n/2 nouveaux éléments dans le tableau à capacité doublée.
- Nous avons

$$C_{\text{réel}}(n) = C_{\text{ajoute}}(n) = \begin{cases} n+1 & \text{si } n \text{ est une puissance de } 2 \\ 1 & \text{sinon} \end{cases}$$

Trace de `ajouteElement`

- `ajouteElement`(1)

  $B \leftarrow B + C_{amorti}(0) - C_{réel}(0) = 0+2-1 = 1$
- `ajouteElement`(2)

  $B \leftarrow B + C_{amorti}(1) - C_{réel}(1) = 1+3-2 = 2$
- `ajouteElement`(3)

  $B \leftarrow B + C_{amorti}(2) - C_{réel}(2) = 2+3-3 = 2$
- `ajouteElement`(4)

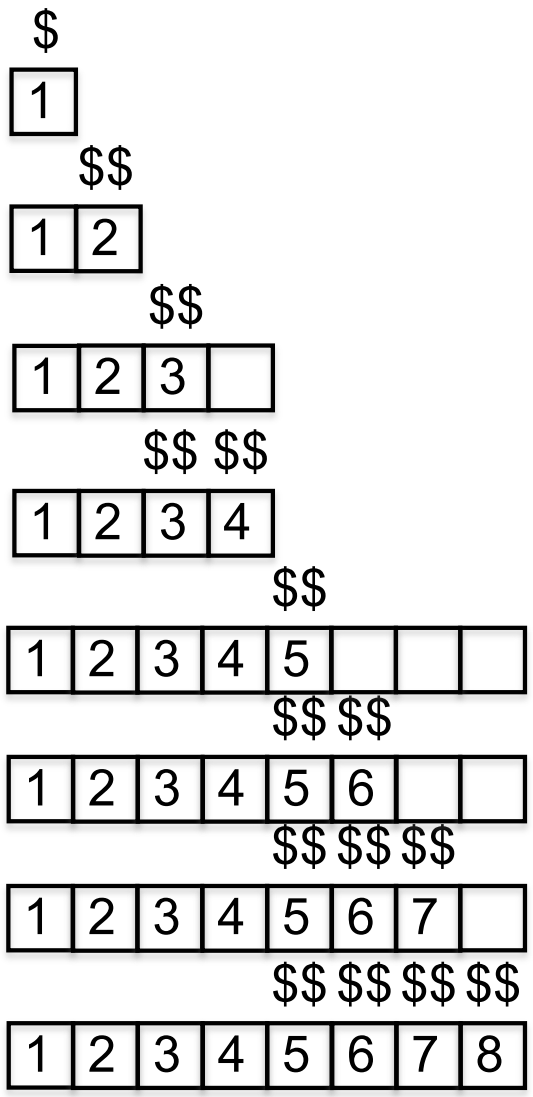
  $B \leftarrow B + C_{amorti}(3) - C_{réel}(3) = 2+3-1 = 4$
- `ajouteElement`(5)

  $B \leftarrow B + C_{amorti}(4) - C_{réel}(4) = 4+3-5 = 2$
- `ajouteElement`(6)

  $B \leftarrow B + C_{amorti}(5) - C_{réel}(5) = 2+3-1 = 4$
- `ajouteElement`(7)

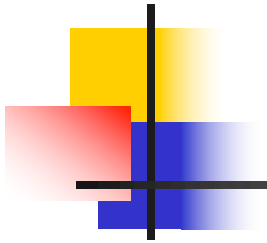
  $B \leftarrow B + C_{amorti}(6) - C_{réel}(6) = 4+3-1 = 6$
- `ajouteElement`(8)

  $B \leftarrow B + C_{amorti}(7) - C_{réel}(7) = 6+3-1 = 8$



La banque de temps n'est jamais en faillite

- **Théorème**: En tout temps, nous avons $B \geq 0$.
- **Démonstration**: Soit B_i la banque de temps après avoir ajouté i éléments. Nous avons vu lors de la trace que $B_0 \geq 0$ et $B_1 \geq 0$.
- **Hypothèse d'induction**: Supposons que $B_i \geq 0$ pour $0 \leq i \leq k$.
- **Étape d'induction**: Nous démontrons que $B_{k+1} \geq 0$.
  - Si k n'est pas une puissance de 2, le tableau ne doit pas être redimensionné. Ce k+1 ième appel ajoute deux unités de temps à la banque de temps. $B_{k+1} = B_k + 2 \geq 2$.
  - Si k est une puissance de 2, il faut redimensionner le tableau. La dernière fois que le tableau a été redimensionné, nous avions $B_{k/2+1} \geq 0$. Les $k/2 - 1$ derniers appels ont ajouté $(k/2 - 1) \times 2$ unités de temps à la banque. Ce k+1 ième appel retranche $k + 1 - 3$ unités de temps car $C_{réel}(k) = k + 1$ et $C_{amorti}(k) = 3$. Nous obtenons
    $B_{k+1} = B_{k/2+1} + (k/2 - 1) \times 2 - (k + 1 - 3) = B_{k/2+1} \geq 0$.
- Ce qu'il fallait démontrer.



Remarques

- Nous avons démontré que charger 3 unités de temps est suffisant pour ne pas mener la banque de temps en faillite.
- Puisque $C_{amorti}(n) \leq 3$ pour tout n, nous avons $C_{amorti}(n) \in O(1)$.
- Puisque $C_{amorti}(n) \geq C_{best}(n) \in \Omega(1)$, nous avons $C_{amorti}(n) \in \Theta(1)$.

- La quantité de temps en banque et le temps accumulé ou dépensé n'a pas à être calculé par l'algorithme.
- Ces temps sont seulement considérés lors de l'analyse.
- On ne retrouve donc pas la variable B dans le code de l'algorithme.

- Il faut choisir une fonction $C_{amorti}(n)$ suffisamment grande pour ne pas que la banque de temps soit en déficit. Cependant, elle doit être la plus petite possible pour bien démontrer l'efficacité de l'algorithme.
- Idéalement, $C_{amorti}(n)$ a le même ordre de croissance que $C_{best}(n)$.

La méthode de la fonction potentiel

- Avec la méthode du comptable, on décide combien une fonction devrait charger en temps et la banque de temps B s'ajuste en conséquence.
- Avec la méthode de la fonction potentiel, on décide plutôt de combien de temps nous devrions avoir en banque et le temps d'exécution amortie s'ajuste en conséquence.
- Une fonction potentiel est une fonction $\Phi(d)$ que l'on associe à l'état d'une structure de données d. Elle doit être non-négative et $\Phi(d) = 0$ lorsque la structure de données d est dans son état initial
  Ex: un vecteur vide, un arbre vide, un vecteur nul, etc…
- Si un algorithme prend en entrée une structure de données d_{i-1} et la modifie sous la forme d_i en un temps d'exécution $C_{réel}(d_{i-1})$, alors cet algorithme aura un temps d'exécution amorti

$$C_{amorti}(d_{i-1}) = C_{réel}(d_{i-1}) + \Phi(d_i) - \Phi(d_{i-1})$$

Séquence d'appels

- L'analyse amortie nous donne une borne supérieure sur le temps d'exécution réel d'une séquence d'exécution.

$$\sum_{i=1}^{k} C_{\text{amorti}}(d_{i-1}) = \sum_{i=1}^{k} \left(C_{\text{réel}}(d_{i-1}) + \phi(d_i) - \phi(d_{i-1})\right)$$

$$= \phi(d_k) - \phi(d_0) + \sum_{i=1}^{k} C_{\text{réel}}(d_{i-1}) \qquad \text{somme télescopique}$$

$$\geq \sum_{i=1}^{k} C_{\text{réel}}(d_{i-1}) \qquad \text{car } \phi(d_k) \geq 0 \text{ et } \phi(d_0) = 0$$

- Il convient donc de choisir la fonction potentiel la plus *petite* possible.
- Si la valeur $\Phi(d_i)$ est excessivement grande, cela voudrait dire qu'on emmagasine trop de temps dans la banque de temps et que ce temps chargé ne sera jamais dépensé.

Exemple: Incrémentation d'un compteur

- Cette fonction prend en entrée un vecteur de bits représentant un entier et incrémente cet entier.
- Le bit le plus faible est à la position A[0] du tableau et le bit le plus fort à la position A[n-1]. Lorsque l'on écrit le nombre en binaire, on doit donc lister les bits de A[n-1] à A[0].

```
ALGORITHME Incremente(A[0..n-1])
assertion: ∀j, A[j] ∈ {0, 1}
i ← 0
tant que i < n et A[i] = 1 faire
    A[i] ← 0
    i ← i + 1
si i = n alors ajoute une entrée au vecteur A
A[i] ← 1
```

Analyse de la fonction Incremente

- La taille de l'instance est n, soit celle du vecteur A
- L'opération de base est la mise à zéro d'un bit du vecteur A
- Le temps d'exécution de l'algorithme est égal à la position du premier bit éteint, ou n si tous les bits sont allumés.

$$C_{\text{réel}}(A) = \min\{i \mid A[i] = 0\} \cup \{n\}$$

- Dans le meilleur des cas, A[0] = 0 et l'algorithme n'éteint aucun bit: $C_{best}(n) = 0 \in \Theta(1)$.
- Dans le pire cas, le vecteur ne contient que des 1 et l'algorithme éteint tous les bits: $C_{worst}(n) = n \in \Theta(n)$.
- Procédons à une analyse amortie en utilisant la fonction de potentielle $\Phi(A)$ qui retourne le nombre de bits allumés dans le vecteur A.

$$\phi(A) = \sum_{i=0}^{n-1} A[i]$$

- Notons d'abord que $\Phi(A) = 0$ lorsque A est le vecteur nul. Notre analyse supposera que le vecteur est initialement nul.

Analyse de la fonction Incremente

- Soit A le vecteur passé à la fonction Incrémente et soit A' le vecteur après avoir été incrémenté. Nous avons la relation suivante.

$$\phi(A') = \phi(A) - C_{\text{réel}}(A) + 1$$

- Le nombre de bits allumés dans A' est égal
  - au nombre de bits allumés dans A;
  - moins ceux qui ont été éteints;
  - plus celui qui a été allumé.
- Par substitution, on obtient le temps d'exécution amorti.

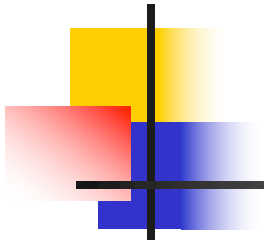
$$\begin{aligned} C_{\text{amorti}}(A) &= C_{\text{réel}}(A) + \phi(A') - \phi(A) \\ &= C_{\text{réel}}(A) + (\phi(A) - C_{\text{réel}}(A) + 1) - \phi(A) \\ &= 1 \\ &\in \Theta(1) \end{aligned}$$

- Ce temps est indépendant de la taille de l'instance, nous avons donc $C_{\text{amorti}}(n) \in \Theta(1)$.

Trace de la fonction incrémente

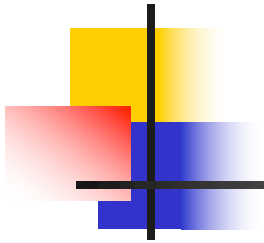
- Soit A_{i-1} le vecteur passé à l'appel i et A_i le vecteur retourné par cet appel. A_0 est le vecteur nul.

i	A_{i-1}	A_i	$\phi(A_{i-1})$	$\phi(A_i)$	$C_{réel}(A_{i-1})$	$C_{amorti}(A_{i-1})$
1	0000	0001	0	1	0	1
2	0001	0010	1	1	1	1
3	0010	0011	1	2	0	1
4	0011	0100	2	1	2	1
5	0100	0101	1	2	0	1
6	0101	0110	2	2	1	1
7	0110	0111	2	3	0	1
8	0111	1000	3	1	3	1



Conclusion

- L'analyse amortie permet de mieux répartir les temps d'exécution entre les pires cas et les meilleurs cas.
- Il est résulte une efficacité qui peut être utilisée pour analyser une séquence d'appels à un algorithme.
- Cette analyse nécessite d'être créatif en:
  - proposant une fonction C_{amorti} qui épargne du temps dans une banque de temps ou qui en retire sans jamais créer un déficit.
  - ou en proposant une fonction potentiel ϕ non négative initialement nulle qui mappe une structure de données à son potentiel.



Lecture

- Cormens, Leiserson, Rivest, Stein, Introduction to algorithms, 3rd edition.
  Chapitre 17: Amortized analysis
- Goodrich, Tamassia, Algorithm design and applications.
  Section 1.4: Amortization