Complexité en moyenne; complexité amortie

Étude sur des exemples

1. Complexité en moyenne
2. Complexité amortie

Présentation

Dans ce chapitre on étudie :

- La notion de *complexité en moyenne* à travers l'exemple classique du tri rapide.

Présentation

Dans ce chapitre on étudie :

- La notion de *complexité en moyenne* à travers l'exemple classique du tri rapide.
- La notion de *coût amorti* (par la méthode du potentiel) à travers l'exemple tout aussi classique du *compteur binaire*.

Crédits

- Wikipedia (Analyse amortie)

Crédits

- Wikipedia (Analyse amortie)
- « Option informatique MPSI - MP/MP* » Roger MANSUY (Vuibert)

Crédits

- Wikipedia (Analyse amortie)
- ≪ Option informatique MPSI - MP/MP* ≫ Roger MANSUY (Vuibert)
- ≪ Algorithmique - 3ème édition Cours avec 957 exercices et 158 problèmes ≫ Thomas H. Cormen, Charles Leiserson, Ronald Rivest, Clifford Stein (Dunod)

Crédits

- Wikipedia (Analyse amortie)
- « Option informatique MPSI - MP/MP* » Roger MANSUY (Vuibert)
- « Algorithmique - 3ème édition Cours avec 957 exercices et 158 problèmes » Thomas H. Cormen, Charles Leiserson, Ronald Rivest, Clifford Stein (Dunod)
- « Informatique - MP2I/MPI - CPGE 1re et 2e années » Balabonski Thibaut, Conchon Sylvain, Filliâtre Jean-Christophe, Nguyen Kim, Sartre Laurent (ellipse)

1 Complexité en moyenne

2 Complexité amortie

Tri rapide

- Dans le tri rapide, on sépare l'entrée en deux listes en comparant à un élément pivot, on les trie séparément et on les fusionne par concaténation.

Tri rapide

- Dans le tri rapide, on sépare l'entrée en deux listes en comparant à un élément pivot, on les trie séparément et on les fusionne par concaténation.
- Dans le tri rapide, la fonction de division est délicate et celle de fusion triviale. C'est le contraire pour le tri fusion.

Tri rapide

- Dans le tri rapide, on sépare l'entrée en deux listes en comparant à un élément pivot, on les trie séparément et on les fusionne par concaténation.
- Dans le tri rapide, la fonction de division est délicate et celle de fusion triviale. C'est le contraire pour le tri fusion.
- On donne une version pour les listes. On peut en donner une version *en place* pour des tableaux.

Tri rapide

- Dans le tri rapide, on sépare l'entrée en deux listes en comparant à un élément pivot, on les trie séparément et on les fusionne par concaténation.
- Dans le tri rapide, la fonction de division est délicate et celle de fusion triviale. C'est le contraire pour le tri fusion.
- On donne une version pour les listes. On peut en donner une version *en place* pour des tableaux.
- Mis au point en 1960 par Tony Hoare, alors étudiant en visite à l'université d'État de Moscou.

Tri rapide

- Dans le tri rapide, on sépare l'entrée en deux listes en comparant à un élément pivot, on les trie séparément et on les fusionne par concaténation.
- Dans le tri rapide, la fonction de division est délicate et celle de fusion triviale. C'est le contraire pour le tri fusion.
- On donne une version pour les listes. On peut en donner une version *en place* pour des tableaux.
- Mis au point en 1960 par Tony Hoare, alors étudiant en visite à l'université d'État de Moscou.
- Possède une variante *Quick select* pour le calcul du k-ième élément d'une liste sans procéder au tri préalable (application : calcul de la médiane).

Tri rapide

Partition

```
let rec partition l p = match l with
  | [] -> [],[]
  | t::q -> let (l1,l2)= partition q p in
    if t<=p then (t::l1,l2) else (l1,t::l2);;
```

- Terminaison :

Tri rapide

Partition

```
let rec partition l p = match l with
  | [] -> [],[]
  | t::q -> let (l1,l2)= partition q p in
    if t<=p then (t::l1,l2) else (l1,t::l2);;
```

- Terminaison :
  - Les cas de bases terminent ;

Tri rapide

Partition

```
let rec partition l p = match l with
  | [] -> [],[]
  | t::q -> let (l1,l2)= partition q p in
    if t<=p then (t::l1,l2) else (l1,t::l2);;
```

- Terminaison :
  - Les cas de bases terminent ;
  - et l'appel interne se fait avec une liste strictement plus courte que la liste initiale ;

Tri rapide

Partition

```
let rec partition l p = match l with
  | [] -> [],[]
  | t::q -> let (l1,l2)= partition q p in
    if t<=p then (t::l1,l2) else (l1,t::l2);;
```

- Terminaison :
  - Les cas de bases terminent ;
  - et l'appel interne se fait avec une liste strictement plus courte que la liste initiale ;
  - et il n'y a pas de boucle ou d'appel à une fonction qui ne termine pas.

Tri rapide

Partition

```
let rec partition l p = match l with
  | [] -> [],[]
  | t::q -> let (l1,l2)= partition q p in
    if t<=p then (t::l1,l2) else (l1,t::l2);;
```

- Terminaison :
  - Les cas de bases terminent ;
  - et l'appel interne se fait avec une liste strictement plus courte que la liste initiale ;
  - et il n'y a pas de boucle ou d'appel à une fonction qui ne termine pas.
  - C'est tout bon !

Tri rapide

Partition

```
let rec partition l p = match l with
  | [] -> [],[]
  | t::q -> let (l1,l2)= partition q p in
    if t<=p then (t::l1,l2) else (l1,t::l2);;
```

- Terminaison :
  - Les cas de bases terminent ;
  - et l'appel interne se fait avec une liste strictement plus courte que la liste initiale ;
  - et il n'y a pas de boucle ou d'appel à une fonction qui ne termine pas.
  - C'est tout bon !
- Complexité temporelle : dans tous les cas de la forme $C_n = C_{n-1} + 1$ pour une liste de taille n. Linéaire.

Tri rapide

Partition : correction par récurrence

```
let rec partition l p = match l with
  | [] -> [],[]
  | t::q -> let (l1,l2)= partition q p in
    if t<=p then (t::l1,l2) else (l1,t::l2);;
```

Montrons que la fonction retourne deux listes dont la 1ere contient tous les éléments de ℓ plus petits que le pivot, les éléments strictement plus grands étant dans la seconde.

- Cas de base : OK de façon évidente. On suppose correct le code pour une taille de liste $\leq n$. Soit `l` telle que $|\ell| = n + 1$.

Tri rapide

Partition : correction par récurrence

```
let rec partition l p = match l with
  | [] -> [],[]
  | t::q -> let (l1,l2)= partition q p in
    if t<=p then (t::l1,l2) else (l1,t::l2);;
```

Montrons que la fonction retourne deux listes dont la 1ere contient tous les éléments de ℓ plus petits que le pivot, les éléments strictement plus grands étant dans la seconde.

- Cas de base : OK de façon évidente. On suppose correct le code pour une taille de liste $\leq n$. Soit `l` telle que $|\ell| = n + 1$.
- Par HR, `partition q p` partitionne correctement `q` en ℓ_1, ℓ_2. Supposons aussi $t \leq p$. Alors dans le tuple retourné :

Tri rapide

Partition : correction par récurrence

```
let rec partition l p = match l with
  | [] -> [],[]
  | t::q -> let (l1,l2)= partition q p in
    if t<=p then (t::l1,l2) else (l1,t::l2);;
```

Montrons que la fonction retourne deux listes dont la 1ere contient tous les éléments de ℓ plus petits que le pivot, les éléments strictement plus grands étant dans la seconde.

- Cas de base : OK de façon évidente. On suppose correct le code pour une taille de liste $\leq n$. Soit `l` telle que $|\ell| = n + 1$.
- Par HR, `partition q p` partitionne correctement `q` en ℓ_1, ℓ_2. Supposons aussi $t \leq p$. Alors dans le tuple retourné :
  - On ajoute t à la liste ℓ_1 des éléments de q plus petits que p. On obtient exactement tous les éléments de ℓ plus petits que le pivot.

Tri rapide

Partition : correction par récurrence

```
let rec partition l p = match l with
  | [] -> [],[]
  | t::q -> let (l1,l2)= partition q p in
    if t<=p then (t::l1,l2) else (l1,t::l2);;
```

Montrons que la fonction retourne deux listes dont la 1ere contient tous les éléments de ℓ plus petits que le pivot, les éléments strictement plus grands étant dans la seconde.

- Cas de base : OK de façon évidente. On suppose correct le code pour une taille de liste $\leq n$. Soit `l` telle que $|\ell| = n+1$.
- Par HR, `partition q p` partitionne correctement `q` en ℓ_1, ℓ_2. Supposons aussi $t \leq p$. Alors dans le tuple retourné :
  - On ajoute t à la liste ℓ_1 des éléments de q plus petits que p. On obtient exactement tous les éléments de ℓ plus petits que le pivot.
  - La liste ℓ_2 est constituée exactement des éléments de q strictement plus grands que p ; donc aussi (puisque $t \leq p$) exactement ceux de ℓ.

Tri rapide

Partition : correction par récurrence

```
let rec partition l p = match l with
  | [] -> [],[]
  | t::q -> let (l1,l2)= partition q p in
    if t<=p then (t::l1,l2) else (l1,t::l2);;
```

Montrons que la fonction retourne deux listes dont la 1ere contient tous les éléments de ℓ plus petits que le pivot, les éléments strictement plus grands étant dans la seconde.

- Cas de base : OK de façon évidente. On suppose correct le code pour une taille de liste $\leq n$. Soit `l` telle que $|\ell| = n+1$.
- Par HR, `partition q p` partitionne correctement `q` en ℓ_1, ℓ_2. Supposons aussi $t \leq p$. Alors dans le tuple retourné :
  - On ajoute t à la liste ℓ_1 des éléments de q plus petits que p. On obtient exactement tous les éléments de ℓ plus petits que le pivot.
  - La liste ℓ_2 est constituée exactement des éléments de q strictement plus grands que p ; donc aussi (puisque $t \leq p$) exactement ceux de ℓ.
  - Correction OK. Le cas $t > p$ est laissé au lecteur.

Tri rapide

Partition : correction par induction

```
let rec partition l p = match l with
  | [] -> [],[]
  | t::q -> let (l1,l2)= partition q p in
    if t<=p then (t::l1,l2) else (l1,t::l2);;
```

Montrons que la fonction retourne deux listes dont la 1ere contient tous les éléments de ℓ plus petits que le pivot, les éléments strictement plus grands étant dans la seconde.

- Cas de base : OK de façon évidente.

Tri rapide

Partition : correction par induction

```
let rec partition l p = match l with
  | [] -> [],[]
  | t::q -> let (l1,l2)= partition q p in
    if t<=p then (t::l1,l2) else (l1,t::l2);;
```

Montrons que la fonction retourne deux listes dont la 1ere contient tous les éléments de ℓ plus petits que le pivot, les éléments strictement plus grands étant dans la seconde.

- Cas de base : OK de façon évidente.
- Hérédité : Supposons que `partition q p` partitionne correctement `q` en ℓ_1, ℓ_2. Supposons aussi $t \leq p$. Alors dans le tuple retourné :

Tri rapide

Partition : correction par induction

```
let rec partition l p = match l with
  | [] -> [],[]
  | t::q -> let (l1,l2)= partition q p in
    if t<=p then (t::l1,l2) else (l1,t::l2);;
```

Montrons que la fonction retourne deux listes dont la 1ere contient tous les éléments de ℓ plus petits que le pivot, les éléments strictement plus grands étant dans la seconde.

- Cas de base : OK de façon évidente.
- Hérédité : Supposons que `partition q p` partitionne correctement `q` en ℓ_1, ℓ_2. Supposons aussi $t \leq p$. Alors dans le tuple retourné :
  - On ajoute t à la liste ℓ_1 des éléments de q plus petits que p. On obtient exactement tous les éléments de ℓ plus petits que le pivot.

Tri rapide

Partition : correction par induction

```
let rec partition l p = match l with
  | [] -> [],[]
  | t::q -> let (l1,l2)= partition q p in
    if t<=p then (t::l1,l2) else (l1,t::l2);;
```

Montrons que la fonction retourne deux listes dont la 1ere contient tous les éléments de ℓ plus petits que le pivot, les éléments strictement plus grands étant dans la seconde.

- Cas de base : OK de façon évidente.
- Hérédité : Supposons que `partition q p` partitionne correctement `q` en ℓ_1, ℓ_2. Supposons aussi $t \leq p$. Alors dans le tuple retourné :
  - On ajoute t à la liste ℓ_1 des éléments de q plus petits que p. On obtient exactement tous les éléments de ℓ plus petits que le pivot.
  - La liste ℓ_2 est constituée exactement des éléments de q strictement plus grands que p ; donc aussi (puisque $t \leq p$) exactement ceux de ℓ.

Tri rapide

Partition : correction par induction

```
let rec partition l p = match l with
  | [] -> [],[]
  | t::q -> let (l1,l2)= partition q p in
    if t<=p then (t::l1,l2) else (l1,t::l2);;
```

Montrons que la fonction retourne deux listes dont la 1ere contient tous les éléments de ℓ plus petits que le pivot, les éléments strictement plus grands étant dans la seconde.

- Cas de base : OK de façon évidente.
- Hérédité : Supposons que `partition q p` partitionne correctement `q` en ℓ_1, ℓ_2. Supposons aussi $t \leq p$. Alors dans le tuple retourné :
  - On ajoute t à la liste ℓ_1 des éléments de q plus petits que p. On obtient exactement tous les éléments de ℓ plus petits que le pivot.
  - La liste ℓ_2 est constituée exactement des éléments de q strictement plus grands que p ; donc aussi (puisque $t \leq p$) exactement ceux de ℓ.
  - Correction OK. Le cas $t > p$ est laissé au lecteur.

Tri rapide

```
let rec tri_rapide l = match l with
    | [] -> []
    | t::q -> let(l1,l2)= partition q t in
      (tri_rapide l1)@(t::(tri_rapide l2));;
```

Terminaison :

- Variant $|\ell|$.

Tri rapide

```
let rec tri_rapide l = match l with
    | [] -> []
    | t::q -> let(l1,l2)= partition q t in
      (tri_rapide l1)@(t::(tri_rapide l2));;
```

Terminaison :

- Variant $|\ell|$.
- Le cas de base termine (envoie de la liste vide).

Tri rapide

```
let rec tri_rapide l = match l with
    | [] -> []
    | t::q -> let(l1,l2)= partition q t in
      (tri_rapide l1)@(t::(tri_rapide l2));;
```

Terminaison :

- Variant $|\ell|$.
- Le cas de base termine (envoie de la liste vide).
- L'appel à `partition` termine (déjà vu).

Tri rapide

```
let rec tri_rapide l = match l with
    | [] -> []
    | t::q -> let(l1,l2)= partition q t in
      (tri_rapide l1)@(t::(tri_rapide l2));;
```

Terminaison :

- Variant $|\ell|$.
- Le cas de base termine (envoie de la liste vide).
- L'appel à `partition` termine (déjà vu).
- Seulement deux appels internes, tous deux effectués avec des listes de taille strictement inférieure à $|\ell|$ (puisque $|\ell_1| + |\ell_2| = |\ell| - 1$).

Tri rapide

```
let rec tri_rapide l = match l with
    | [] -> []
    | t::q -> let(l1,l2)= partition q t in
      (tri_rapide l1)@(t::(tri_rapide l2));;
```

Terminaison :

- Variant $|\ell|$.
- Le cas de base termine (envoie de la liste vide).
- L'appel à `partition` termine (déjà vu).
- Seulement deux appels internes, tous deux effectués avec des listes de taille strictement inférieure à $|\ell|$ (puisque $|\ell_1| + |\ell_2| = |\ell| - 1$).
- Terminaison OK.

Tri rapide : correction

Preuve par induction

```
let rec tri_rapide l = match l with
    | [] -> []
    | t::q -> let(l1,l2)= partition q t in
      (tri_rapide l1)@(t::(tri_rapide l2));;
```

Correction : Cas de base : OK. Supposons que les deux appels internes retournent une version triée de `l1` et de `l2`.

- Par correction de `partition` : `l1` contient les éléments de `q` plus petits ou égaux au pivot et `l2` les éléments strictement plus grands.

Tri rapide : correction

Preuve par induction

```
let rec tri_rapide l = match l with
    | [] -> []
    | t::q -> let(l1,l2)= partition q t in
      (tri_rapide l1)@(t::(tri_rapide l2));;
```

Correction : Cas de base : OK. Supposons que les deux appels internes retournent une version triée de `l1` et de `l2`.

- Par correction de `partition` : `l1` contient les éléments de `q` plus petits ou égaux au pivot et `l2` les éléments strictement plus grands.
- La concaténation retournée

Tri rapide : correction

Preuve par induction

```
let rec tri_rapide l = match l with
    | [] -> []
    | t::q -> let(l1,l2)= partition q t in
      (tri_rapide l1)@(t::(tri_rapide l2));;
```

Correction : Cas de base : OK. Supposons que les deux appels internes retournent une version triée de `l1` et de `l2`.

- Par correction de `partition` : `l1` contient les éléments de `q` plus petits ou égaux au pivot et `l2` les éléments strictement plus grands.
- La concaténation retournée
  - est triée : d'abords les éléments plus petits que le pivot rangés dans l'ordre, puis le pivot, puis les élements plus grands que le pivot rangés dans l'ordre.

Tri rapide : correction

Preuve par induction

```
let rec tri_rapide l = match l with
    | [] -> []
    | t::q -> let(l1,l2)= partition q t in
      (tri_rapide l1)@(t::(tri_rapide l2));;
```

Correction : Cas de base : OK. Supposons que les deux appels internes retournent une version triée de `l1` et de `l2`.

- Par correction de `partition` : `l1` contient les éléments de `q` plus petits ou égaux au pivot et `l2` les éléments strictement plus grands.
- La concaténation retournée
  - est triée : d'abords les éléments plus petits que le pivot rangés dans l'ordre, puis le pivot, puis les élements plus grands que le pivot rangés dans l'ordre.
  - contient exactement tous les éléments de `q` (puisque la réunion de `l1` et `l2` les contient) plus l'élément manquant `t`.

Tri rapide : correction

Preuve par induction

```
let rec tri_rapide l = match l with
    | [] -> []
    | t::q -> let(l1,l2)= partition q t in
      (tri_rapide l1)@(t::(tri_rapide l2));;
```

Correction : Cas de base : OK. Supposons que les deux appels internes retournent une version triée de `l1` et de `l2`.

- Par correction de `partition` : `l1` contient les éléments de `q` plus petits ou égaux au pivot et `l2` les éléments strictement plus grands.
- La concaténation retournée
  - est triée : d'abords les éléments plus petits que le pivot rangés dans l'ordre, puis le pivot, puis les élements plus grands que le pivot rangés dans l'ordre.
  - contient exactement tous les éléments de `q` (puisque la réunion de `l1` et `l2` les contient) plus l'élément manquant `t`.
  - La concaténation est donc la version triée de `l`. Hérédité OK

Tri rapide : correction

Preuve par récurrence

Correction : Cas de base : OK. Supposons que le tri soit correct pour des tailles de listes $\leq n$. On effectue `tri_rapide l` avec $|\ell| = n + 1$. Par HR, les deux appels internes sur ℓ_1 et ℓ_2 retournent une version triée de `l1` et de `l2`.

- Par correction de `partition` : `l1` contient les éléments de `q` plus petits ou égaux au pivot et `l2` les éléments strictement plus grands.

Tri rapide : correction

Preuve par récurrence

Correction : Cas de base : OK. Supposons que le tri soit correct pour des tailles de listes $\leq n$. On effectue `tri_rapide l` avec $|\ell| = n + 1$. Par HR, les deux appels internes sur ℓ_1 et ℓ_2 retournent une version triée de `l1` et de `l2`.

- Par correction de `partition` : `l1` contient les éléments de `q` plus petits ou égaux au pivot et `l2` les éléments strictement plus grands.
- La concaténation retournée

Tri rapide : correction

Preuve par récurrence

Correction : Cas de base : OK. Supposons que le tri soit correct pour des tailles de listes $\leq n$. On effectue `tri_rapide l` avec $|\ell| = n + 1$. Par HR, les deux appels internes sur ℓ_1 et ℓ_2 retournent une version triée de `l1` et de `l2`.

- Par correction de `partition` : `l1` contient les éléments de `q` plus petits ou égaux au pivot et `l2` les éléments strictement plus grands.
- La concaténation retournée
  - est triée : d'abords les éléments plus petits que le pivot rangés dans l'ordre, puis le pivot, puis les éléments strictement plus grands que le pivot rangés dans l'ordre.

Tri rapide : correction

Preuve par récurrence

Correction : Cas de base : OK. Supposons que le tri soit correct pour des tailles de listes $\leq n$. On effectue `tri_rapide l` avec $|\ell| = n + 1$. Par HR, les deux appels internes sur ℓ_1 et ℓ_2 retournent une version triée de `l1` et de `l2`.

- Par correction de `partition` : `l1` contient les éléments de `q` plus petits ou égaux au pivot et `l2` les éléments strictement plus grands.
- La concaténation retournée
  - est triée : d'abords les éléments plus petits que le pivot rangés dans l'ordre, puis le pivot, puis les éléments strictement plus grands que le pivot rangés dans l'ordre.
  - contient exactement tous les éléments de `q` (puisque `l1 @ l2` les contient) plus l'élément manquant `t`.

Tri rapide : correction

Preuve par récurrence

Correction : Cas de base : OK. Supposons que le tri soit correct pour des tailles de listes $\leq n$. On effectue `tri_rapide l` avec $|\ell| = n + 1$. Par HR, les deux appels internes sur ℓ_1 et ℓ_2 retournent une version triée de `l1` et de `l2`.

- Par correction de `partition` : `l1` contient les éléments de `q` plus petits ou égaux au pivot et `l2` les éléments strictement plus grands.
- La concaténation retournée
  - est triée : d'abords les éléments plus petits que le pivot rangés dans l'ordre, puis le pivot, puis les éléments strictement plus grands que le pivot rangés dans l'ordre.
  - contient exactement tous les éléments de `q` (puisque `l1 @ l2` les contient) plus l'élément manquant `t`.
  - La concaténation est donc la version triée de `l`. Hérédité OK.

Tri rapide : complexité en nombre de comparaisons

Cas où l'une des listes est toujours vide

- Supposons que l'une des listes retournée par partition soit vide à chaque fois (cela se produit par exemple si la liste est déjà triée).

Tri rapide : complexité en nombre de comparaisons

Cas où l'une des listes est toujours vide

- Supposons que l'une des listes retournée par partition soit vide à chaque fois (cela se produit par exemple si la liste est déjà triée).
- On désigne par T_n la complexité temporelle en nombre de comparaisons pour une liste triée de taille n.

Tri rapide : complexité en nombre de comparaisons

Cas où l'une des listes est toujours vide

- Supposons que l'une des listes retournée par partition soit vide à chaque fois (cela se produit par exemple si la liste est déjà triée).
- On désigne par T_n la complexité temporelle en nombre de comparaisons pour une liste triée de taille n.
- Alors T_n vérifie une relation de la forme $T_n = T_{n-1} + n - 1$ ($n-1$: nombre exact de comparaisons dans `partition`) et $T_0 = 0$ (aucune comparaison).
  Remarquons au passage que $T_1 = 0$ (aucune comparaison) et que $T_{1-1} + 1 - 1 = 0 = T_1$.

Tri rapide : complexité en nombre de comparaisons

Cas où l'une des listes est toujours vide

- Supposons que l'une des listes retournée par partition soit vide à chaque fois (cela se produit par exemple si la liste est déjà triée).
- On désigne par T_n la complexité temporelle en nombre de comparaisons pour une liste triée de taille n.
- Alors T_n vérifie une relation de la forme $T_n = T_{n-1} + n - 1$ ($n-1$: nombre exact de comparaisons dans `partition`) et $T_0 = 0$ (aucune comparaison).
  Remarquons au passage que $T_1 = 0$ (aucune comparaison) et que $T_{1-1} + 1 - 1 = 0 = T_1$.
- Complexité en somme des premiers entiers soit $O(n^2)$.

Tri rapide : complexité en nombre de comparaisons

Cas où l'une des listes est toujours vide

- Supposons que l'une des listes retournée par partition soit vide à chaque fois (cela se produit par exemple si la liste est déjà triée).
- On désigne par T_n la complexité temporelle en nombre de comparaisons pour une liste triée de taille n.
- Alors T_n vérifie une relation de la forme $T_n = T_{n-1} + n - 1$ ($n-1$: nombre exact de comparaisons dans `partition`) et $T_0 = 0$ (aucune comparaison).
  Remarquons au passage que $T_1 = 0$ (aucune comparaison) et que $T_{1-1} + 1 - 1 = 0 = T_1$.
- Complexité en somme des premiers entiers soit $O(n^2)$.
- On montre dans le transparent suivant que ce cas est bien le pire, c'est à dire que la complexité C_n en nombre de comparaisons est la pire si le pivot est à une extrémité et donc que $C_n = T_n$.

Tri rapide : complexité en nombre de comparaisons

Cas où l'une des listes est toujours vide (1)

- HR(n) : pour tout $q \leq n$: 1) T_q est la pire complexité et 2)

$$\forall k \in \{1, \ldots, n\}, \; T_{k-1} + T_{n-k} + n - 1 \leq T_n \text{ (égalité si } k = 1)$$

Tri rapide : complexité en nombre de comparaisons

Cas où l'une des listes est toujours vide (1)

- HR(n) : pour tout $q \leq n$: 1) T_q est la pire complexité et 2)

$$\forall k \in \{1, \ldots, n\}, \; T_{k-1} + T_{n-k} + n - 1 \leq T_n \text{ (égalité si } k = 1)$$

- Cas de base $n = 1$. $T_0 = 0 = T_1$, $k \in \{1\}$. Alors $T_{1-1} + T_{1-1} + 1 - 1 = 0 \leq T_1$. Et T_1 est la pire cpx. HR(1) OK.

Tri rapide : complexité en nombre de comparaisons

Cas où l'une des listes est toujours vide (1)

- HR(n) : pour tout $q \leq n$: 1) T_q est la pire complexité et 2)

$$\forall k \in \{1, \ldots, n\}, \; T_{k-1} + T_{n-k} + n - 1 \leq T_n \text{ (égalité si } k = 1)$$

- Cas de base $n = 1$. $T_0 = 0 = T_1$, $k \in \{1\}$. Alors $T_{1-1} + T_{1-1} + 1 - 1 = 0 \leq T_1$. Et T_1 est la pire cpx. HR(1) OK.
- Si HR(n) est vérifiée. Soit $k \in \{1, \ldots, n+1\}$

$$\begin{aligned} T_{k-1} + T_{n+1-k} + n & \underbrace{=}_{\text{def. de } T} \\ T_{k-1} + T_{n-k} + (n-k) + n - 1 + 1 & \underbrace{\leq}_{HR(n)} \end{aligned}$$

$T_n + (n - k + 1) \leq T_n + n = T_{n+1}$: Point 2 OK.

Tri rapide : complexité en nombre de comparaisons

Cas où l'une des listes est toujours vide (2)

- Soit ℓ une liste quelconque de longueur $n+1$ partitionnée lors de l'appel `partition q` en deux sous-listes ℓ_1, ℓ_2 de longueur k et $n-k$ ($k \geq 0$). Notons C_{n+1}^{ℓ} la complexité exacte en nombre de comparaisons de l'appel `tri_rapide l` ; $C_k^{\ell_1}$ (resp. $C_{n-k}^{\ell_2}$) les complexités de `tri_rapide l1` (resp. `tri_rapide l2`).

Tri rapide : complexité en nombre de comparaisons

Cas où l'une des listes est toujours vide (2)

- Soit ℓ une liste quelconque de longueur $n+1$ partitionnée lors de l'appel `partition q` en deux sous-listes ℓ_1, ℓ_2 de longueur k et $n-k$ ($k \geq 0$). Notons C^{ℓ}_{n+1} la complexité exacte en nombre de comparaisons de l'appel `tri_rapide l` ; $C^{\ell_1}_{k}$ (resp. $C^{\ell_2}_{n-k}$) les complexités de `tri_rapide l1` (resp. `tri_rapide l2`).
- Si ℓ_i est vide, $C^{\ell}_{n+1} = C^{\ell_j}_{n} + n \leq T_n + n$ par HR.1 (puisque $|\ell_i| \leq n$) donc $C^{\ell}_{n+1} \leq T_{n+1}$.

Tri rapide : complexité en nombre de comparaisons

Cas où l'une des listes est toujours vide (2)

- Soit ℓ une liste quelconque de longueur $n+1$ partitionnée lors de l'appel `partition q` en deux sous-listes ℓ_1, ℓ_2 de longueur k et $n-k$ ($k \geq 0$). Notons C^{ℓ}_{n+1} la complexité exacte en nombre de comparaisons de l'appel `tri_rapide l` ; $C^{\ell_1}_{k}$ (resp. $C^{\ell_2}_{n-k}$) les complexités de `tri_rapide l1` (resp. `tri_rapide l2`).
- Si ℓ_i est vide, $C^{\ell}_{n+1} = C^{\ell_j}_{n} + n \leq T_n + n$ par HR.1 donc $C^{\ell}_{n+1} \leq T_{n+1}$.
- Si $|\ell_1| \times |\ell_2| \neq 0$ alors $1 \leq k \leq n-1$ et :

$$\begin{aligned} C^{\ell}_{n+1} &= C^{\ell_1}_{k} + C^{\ell_2}_{n-k} + n \\ &\underbrace{\leq}_{\text{HR}(n)} T_k + T_{n-k} + n \underbrace{=}_{\substack{k'=k+1 \\ k' \in [\![2,n]\!]}} T_{k'-1} + T_{n+1-k'} + n \\ &\leq T_{n+1} \text{ (d'après la preuve du transparent précédent)} \end{aligned}$$

Tri rapide : complexité en nombre de comparaisons

Cas où l'une des listes est toujours vide (2)

- Soit ℓ une liste quelconque de longueur $n+1$ partitionnée lors de l'appel `partition q` en deux sous-listes ℓ_1, ℓ_2 de longueur k et $n-k$ ($k \geq 0$). Notons C_{n+1}^{ℓ} la complexité exacte en nombre de comparaisons de l'appel `tri_rapide l` ; $C_k^{\ell_1}$ (resp. $C_{n-k}^{\ell_2}$) les complexités de `tri_rapide l1` (resp. `tri_rapide l2`).
- Si ℓ_i est vide, $C_{n+1}^{\ell} = C_n^{\ell_j} + n \leq T_n + n$ par HR.1 donc $C_{n+1}^{\ell} \leq T_{n+1}$.
- Si $|\ell_1| \times |\ell_2| \neq 0$ alors $1 \leq k \leq n-1$ et :

$$\begin{aligned} C_{n+1}^{\ell} &= C_k^{\ell_1} + C_{n-k}^{\ell_2} + n \\ &\underbrace{\leq}_{\text{HR}(n)} T_k + T_{n-k} + n \underbrace{=}_{\substack{k'=k+1 \\ k' \in [\![2,n]\!]}} T_{k'-1} + T_{n+1-k'} + n \\ &\leq T_{n+1} \text{ (d'après la preuve du transparent précédent)} \end{aligned}$$

- Ainsi, ℓ étant quelconque, T_{n+1} est la pire complexité possible. IZP !

Complexité moyenne en nombre de comparaisons

- La *complexité moyenne* $C(n)$ en nombre de comparaisons du tri rapide pour une liste de taille n est la moyenne des complexités pour les différentes positions du pivot. Elle vérifie (si la position finale du pivot est équiprobable) :

$$\begin{aligned} C(n) &= \frac{1}{n}\sum_{k=0}^{n-1}(C(k)+C(n-k-1)+n-1) \\ &= n-1+\frac{1}{n}\sum_{k=0}^{n-1}(C(k)+C(n-k-1)) \end{aligned}$$

On constate que chaque terme apparaît deux fois.

Complexité moyenne en nombre de comparaisons

- La *complexité moyenne* $C(n)$ du tri rapide pour une liste de taille n est la moyenne des complexités pour les différentes positions du pivot. Elle vérifie (si la position finale du pivot est équiprobable) :

$$\begin{aligned} C(n) &= \frac{1}{n}\sum_{k=0}^{n-1}(C(k)+C(n-k-1)+n-1) \\ &= n-1+\frac{1}{n}\sum_{k=0}^{n-1}(C(k)+C(n-k-1)) \end{aligned}$$

  On constate que chaque terme apparaît deux fois.

- Alors $C(n) = n-1+\frac{2}{n}\sum_{k=0}^{n-1} C(k)$ donc $nC(n) = 2\sum_{k=0}^{n-1} C(k) + n(n-1)$

Complexité moyenne en nombre de comparaisons

- La *complexité moyenne* $C(n)$ du tri rapide pour une liste de taille n est la moyenne des complexités pour les différentes positions du pivot. Elle vérifie (si la position finale du pivot est équiprobable) :

$$\begin{aligned} C(n) &= \frac{1}{n}\sum_{k=0}^{n-1}(C(k)+C(n-k-1)+n-1) \\ &= n-1+\frac{1}{n}\sum_{k=0}^{n-1}(C(k)+C(n-k-1)) \end{aligned}$$

  On constate que chaque terme apparaît deux fois.

- Alors $C(n) = n-1+\frac{2}{n}\sum_{k=0}^{n-1} C(k)$ donc
  $nC(n) = 2\sum_{k=0}^{n-1} C(k) + n(n-1)$
- Donc
  $nC(n)-(n-1)C(n-1) = 2C(n-1)+n(n-1)-(n-1)(n-2)$
  puis $nC(n)-(n+1)C(n-1) = 2(n-1)$

Complexité moyenne du tri rapide

- Il vient

$$\frac{C(n)}{n+1} - \frac{C(n-1)}{n} = \frac{2(n-1)}{n(n+1)}$$

$$= \frac{4}{n+1} - \frac{2}{n} \text{ (décomp. en éléments simples)}$$

Complexité moyenne du tri rapide

- Il vient

$$\frac{C(n)}{n+1} - \frac{C(n-1)}{n} = \frac{2(n-1)}{n(n+1)}$$
$$= \frac{4}{n+1} - \frac{2}{n} \text{ (décomp. en éléments simples)}$$

- Par télescopage, et puisque $C(0) = 0$

$$\frac{C(n)}{n+1} - \frac{C(0)}{1} = \frac{C(n)}{n+1} = 4\sum_{k=1}^{n} \frac{1}{k+1} - 2\sum_{k=1}^{n} \frac{1}{k}$$
$$= 4\sum_{k=2}^{n+1} \frac{1}{k} - 2\sum_{k=1}^{n} \frac{1}{k}$$
$$= 2\sum_{k=2}^{n} \frac{1}{k} + \frac{4}{n+1} - 2$$
$$= 2\sum_{k=1}^{n} \frac{1}{k} + \frac{4}{n+1} - 4 \leq 2 \times \underbrace{\sum_{k=1}^{n} \frac{1}{k}}_{\sim \ln(n)} + \underbrace{\frac{n+1}{4}}_{O(1/n)}$$

Complexité moyenne du tri rapide

- on sait que $\sum_{k=1}^{n} \frac{1}{k} = \ln n + \gamma + O(1/n)$ où γ est la constante d'Euler ($\gamma \simeq 0.577$, cf. cours de maths).

Complexité moyenne du tri rapide

- on sait que $\sum_{k=1}^{n} \frac{1}{k} = \ln n + \gamma + O(1/n)$ où γ est la constante d'Euler ($\gamma \simeq 0.577$, cf. cours de maths).
- Alors
$$\frac{C(n)}{n+1} = (2\ln n + 2\gamma + O(1/n)) + \frac{4}{n+1} - 4 = 2\ln n + 2\gamma + O(1/n) - 4.$$

Complexité moyenne du tri rapide

- on sait que $\sum_{k=1}^{n} \frac{1}{k} = \ln n + \gamma + O(1/n)$ où γ est la constante d'Euler ($\gamma \simeq 0.577$, cf. cours de maths).
- Alors
$$\frac{C(n)}{n+1} = (2\ln n + 2\gamma + O(1/n)) + \frac{4}{n+1} - 4 = 2\ln n + 2\gamma + O(1/n) - 4.$$
- Donc $C(n) = O(n \log n)$.

Complexité moyenne du tri rapide

- on sait que $\sum_{k=1}^{n} \frac{1}{k} = \ln n + \gamma + O(1/n)$ où γ est la constante d'Euler ($\gamma \simeq 0.577$, cf. cours de maths).
- Alors
$$\frac{C(n)}{n+1} = (2\ln n + 2\gamma + O(1/n)) + \frac{4}{n+1} - 4 = 2\ln n + 2\gamma + O(1/n) - 4.$$
- Donc $C(n) = O(n \log n)$.
- On peut minorer de la même façon (en exo : utiliser $\sum_{k=1}^{n} \frac{1}{k} > \ln n$ apcr), donc $C_n = \Theta(n \log n)$.

1 Complexité en moyenne

2 Complexité amortie

Présentation

- L'*analyse amortie* est une méthode d'évaluation de la complexité temporelle des opérations sur une structure de données (tableaux, piles, files, arbres...).

Présentation

- L'*analyse amortie* est une méthode d'évaluation de la complexité temporelle des opérations sur une structure de données (tableaux, piles, files, arbres...).
- Elle consiste principalement à majorer le coût cumulé d'une suite d'opérations pour attribuer à chaque opération la moyenne de cette majoration, en prenant en compte le fait que les cas chers surviennent rarement et isolément et compensent les cas bon marché.

Exemple du compteur binaire

- On veut représenter, pour un entier n, tous les tableaux possibles de bouléens de longueur n.

Exemple du compteur binaire

- On veut représenter, pour un entier n, tous les tableaux possibles de bouléens de longueur n.
- Un tel tableau `t` peut être vu comme la représentation binaire d'en entier positif sur n bits. Il y en a 2^n. On choisit la version *little endian* : bit de poids faible en `t[0]`. Donc 1 s'écrit

1	0	0	0

.

Exemple du compteur binaire

- On veut représenter, pour un entier n, tous les tableaux possibles de bouléens de longueur n.
- Un tel tableau `t` peut être vu comme la représentation binaire d'en entier positif sur n bits. Il y en a 2^n. On choisit la version *little endian* : bit de poids faible en `t[0]`. Donc 1 s'écrit | 1 | 0 | 0 | 0 |.
- Pour avoir tous les tableaux, on part du tableau entièrement nul et on lui applique itérativement un incrément de 1 (donc $2^n - 1$ fois) :

```
1  void incr(bool c[], int n){
2    int i=0;
3    while (i<n && c[i]==1)
4     { c[i] = 0; i++;}// propagation de la retenue
5    if (i<n) c[i] = 1;
6  }
```

Exemple du compteur binaire

- On veut représenter, pour un entier n, tous les tableaux possibles de bouléens de longueur n.
- Un tel tableau `t` peut être vu comme la représentation binaire d'en entier positif sur n bits. Il y en a 2^n. On choisit la version *little endian* : bit de poids faible en `t[0]`. Donc 1 s'écrit | 1 | 0 | 0 | 0 |.
- Pour avoir tous les tableaux, on part du tableau entièrement nul et on lui applique itérativement un incrément de 1 (donc $2^n - 1$ fois) :

```
1  void incr(bool c[], int n){
2    int i=0;
3    while (i<n && c[i]==1)
4     { c[i] = 0; i++;}// propagation de la retenue
5    if (i<n) c[i] = 1;
6  }
```

- Nous voulons mesurer la complexité de cette fonction en nombre d'accès aux éléments du tableau.

Analyse grossière

Pour un tableau de taille n.

- La complexité dépend du nombre de tour dans la boucle `while`.

Analyse grossière

Pour un tableau de taille n.

- La complexité dépend du nombre de tour dans la boucle `while`.
- Meilleur cas : si `t[0]==0`, alors pas de passage dans la boucle, deux accès seulement (en lecture puis écriture).

Analyse grossière

Pour un tableau de taille n.

- La complexité dépend du nombre de tour dans la boucle `while`.
- Meilleur cas : si `t[0]==0`, alors pas de passage dans la boucle, deux accès seulement (en lecture puis écriture).
- Pire cas : tous les bits à 1. Pour chacun des n passages, un accès en lecture puis un en écriture. $2n$ accès .

Analyse grossière

Pour un tableau de taille n.

- La complexité dépend du nombre de tour dans la boucle `while`.
- Meilleur cas : si `t[0]==0`, alors pas de passage dans la boucle, deux accès seulement (en lecture puis écriture).
- Pire cas : tous les bits à 1. Pour chacun des n passages, un accès en lecture puis un en écriture. $2n$ accès .
- Dans le pire cas, la complexité est en $O(n)$ pour un tableau.

Analyse grossière

Pour un tableau de taille n.

- La complexité dépend du nombre de tour dans la boucle `while`.
- Meilleur cas : si `t[0]==0`, alors pas de passage dans la boucle, deux accès seulement (en lecture puis écriture).
- Pire cas : tous les bits à 1. Pour chacun des n passages, un accès en lecture puis un en écriture. $2n$ accès .
- Dans le pire cas, la complexité est en $O(n)$ pour un tableau.
- Majorer le coût des 2^n appels à `incr` : $2n2^n = n2^{n+1}$.
  C'est trop imprécis car passer n fois dans la boucle arrive exceptionnellement.

Exemple

Numéro	Paramètre	Nombre de tours
0	0000 . . . 0	0
1	1000 . . . 0	1
2	0100 . . . 0	0
3	1100 . . . 0	2
4	0010 . . . 0	0
5	1010 . . . 0	1
6	0110 . . . 0	0
7	1110 . . . 0	3
8	0001 . . . 0	0
9	1001 . . . 0	1
10	0101 . . . 0	0
11	1101 . . . 0	2
12	0011 . . . 0	0
13	1011 . . . 0	1
14	0111 . . . 0	0
15	1111 . . . 0	4

Total : 14 passages dans `while`

Sur 16 appels consécutifs à `incr` :

- Un appel sur 2 : 0 tour de boucle ;

Exemple

Numéro	Paramètre	Nombre de tours
0	0000 . . . 0	0
1	1000 . . . 0	1
2	0100 . . . 0	0
3	1100 . . . 0	2
4	0010 . . . 0	0
5	1010 . . . 0	1
6	0110 . . . 0	0
7	1110 . . . 0	3
8	0001 . . . 0	0
9	1001 . . . 0	1
10	0101 . . . 0	0
11	1101 . . . 0	2
12	0011 . . . 0	0
13	1011 . . . 0	1
14	0111 . . . 0	0
15	1111 . . . 0	4

Total : 14 passages dans `while`

Sur 16 appels consécutifs à `incr` :

- Un appel sur 2 : 0 tour de boucle ;
- Un appel sur 4 : 1 tours de boucle ;

Exemple

Numéro	Paramètre	Nombre de tours
0	0000 . . . 0	0
1	1000 . . . 0	1
2	0100 . . . 0	0
3	1100 . . . 0	2
4	0010 . . . 0	0
5	1010 . . . 0	1
6	0110 . . . 0	0
7	1110 . . . 0	3
8	0001 . . . 0	0
9	1001 . . . 0	1
10	0101 . . . 0	0
11	1101 . . . 0	2
12	0011 . . . 0	0
13	1011 . . . 0	1
14	0111 . . . 0	0
15	1111 . . . 0	4

Total : 14 passages dans `while`

Sur 16 appels consécutifs à `incr` :

- Un appel sur 2 : 0 tour de boucle ;
- Un appel sur 4 : 1 tours de boucle ;
- Un appel sur 8 : 2 tours de boucle ;

Exemple

Numéro	Paramètre	Nombre de tours
0	0000 . . . 0	0
1	1000 . . . 0	1
2	0100 . . . 0	0
3	1100 . . . 0	2
4	0010 . . . 0	0
5	1010 . . . 0	1
6	0110 . . . 0	0
7	1110 . . . 0	3
8	0001 . . . 0	0
9	1001 . . . 0	1
10	0101 . . . 0	0
11	1101 . . . 0	2
12	0011 . . . 0	0
13	1011 . . . 0	1
14	0111 . . . 0	0
15	1111 . . . 0	4

Total : 14 passages dans `while`

Sur 16 appels consécutifs à `incr` :

- Un appel sur 2 : 0 tour de boucle ;
- Un appel sur 4 : 1 tours de boucle ;
- Un appel sur 8 : 2 tours de boucle ;
- Un appel sur 16 : 3 tours de boucle ;

Cadre d'étude de complexité amortie

On effectue une étude de complexité amortie pour des algorithmes qui

- ont une faible complexité pour de nombreuses entrées ;

Cadre d'étude de complexité amortie

On effectue une étude de complexité amortie pour des algorithmes qui

- ont une faible complexité pour de nombreuses entrées ;
- sont coûteux pour certaines entrées (peu nombreuses en proportion) ;

Cadre d'étude de complexité amortie

On effectue une étude de complexité amortie pour des algorithmes qui

- ont une faible complexité pour de nombreuses entrées ;
- sont coûteux pour certaines entrées (peu nombreuses en proportion) ;
- vérifient que dans toute séquence d'appel à l'algorithme, les entrées coûteuses interviennent assez rarement pour que le coût moyen reste faible.

Différentes méthodes

- Il y a 3 méthodes usuelles d'analyse amortie : la méthode de l'agrégat, la méthode comptable et la méthode du potentiel (seule étudiée ici).

Différentes méthodes

- Il y a 3 méthodes usuelles d'analyse amortie : la méthode de l'agrégat, la méthode comptable et la méthode du potentiel (seule étudiée ici).
- Méthode du potentiel : A chaque entrée possible x, on associe un nombre positif ou nul $\phi(x)$ dit *potentiel*. Il représente un coût latent dans l'entrée mais pas encore réalisé.

Différentes méthodes

- Il y a 3 méthodes usuelles d'analyse amortie : la méthode de l'agrégat, la méthode comptable et la méthode du potentiel (seule étudiée ici).
- Méthode du potentiel : A chaque entrée possible x, on associe un nombre positif ou nul $\phi(x)$ dit *potentiel*. Il représente un coût latent dans l'entrée mais pas encore réalisé.
- Une séquence d'un algorithme ayant de bonnes propriétés de complexité amortie alterne entre :

Différentes méthodes

- Il y a 3 méthodes usuelles d'analyse amortie : la méthode de l'agrégat, la méthode comptable et la méthode du potentiel (seule étudiée ici).
- Méthode du potentiel : A chaque entrée possible x, on associe un nombre positif ou nul $\phi(x)$ dit *potentiel*. Il représente un coût latent dans l'entrée mais pas encore réalisé.
- Une séquence d'un algorithme ayant de bonnes propriétés de complexité amortie alterne entre :
  - de (nombreuses) opérations de faible coût faisant monter progressivement le potentiel,

Différentes méthodes

- Il y a 3 méthodes usuelles d'analyse amortie : la méthode de l'agrégat, la méthode comptable et la méthode du potentiel (seule étudiée ici).
- Méthode du potentiel : A chaque entrée possible x, on associe un nombre positif ou nul $\phi(x)$ dit *potentiel*. Il représente un coût latent dans l'entrée mais pas encore réalisé.
- Une séquence d'un algorithme ayant de bonnes propriétés de complexité amortie alterne entre :
  - de (nombreuses) opérations de faible coût faisant monter progressivement le potentiel,
  - et de (peu nombreuses) opérations de coût élevé faisant diminuer brusquement le potentiel.

Différentes méthodes

- Il y a 3 méthodes usuelles d'analyse amortie : la méthode de l'agrégat, la méthode comptable et la méthode du potentiel (seule étudiée ici).
- Méthode du potentiel : A chaque entrée possible x, on associe un nombre positif ou nul $\phi(x)$ dit *potentiel*. Il représente un coût latent dans l'entrée mais pas encore réalisé.
- Une séquence d'un algorithme ayant de bonnes propriétés de complexité amortie alterne entre :
  - de (nombreuses) opérations de faible coût faisant monter progressivement le potentiel,
  - et de (peu nombreuses) opérations de coût élevé faisant diminuer brusquement le potentiel.
- Dans l'exemple du compteur binaire, le potentiel est poportionnel au nombre de 1 dans le tableau.

Coût amorti d'une opération

Définition

Soit une opération *op* dont l'éxécution produit une sortie x_s à partir d'une entrée x_e. Le *coût réel* C de *op* est la complexité temporelle de son exécution. Son *coût amorti* A est la somme du coût réel et de la variation de potentiel :

$$A = C + \phi(x_s) - \phi(x_e).$$

- Dans cette définition, les *entrées* et *sorties* représentent au sens large ce qui est donné à l'algorithme (paramètres, état de la mémoire au début de l'appel) et ce qu'il produit (résultats renvoyés, état de la mémoire en sortie).

Coût amorti d'une opération

Définition

Soit une opération *op* dont l'éxécution produit une sortie x_s à partir d'une entrée x_e. Le *coût réel* C de *op* est la complexité temporelle de son exécution. Son *coût amorti* A est la somme du coût réel et de la variation de potentiel :

$$A = C + \phi(x_s) - \phi(x_e).$$

- Dans cette définition, les *entrées* et *sorties* représentent au sens large ce qui est donné à l'algorithme (paramètres, état de la mémoire au début de l'appel) et ce qu'il produit (résultats renvoyés, état de la mémoire en sortie).
- Une *séquence d'opérations consécutives* est une suite d'appels à un algorithme de sorte que chaque sortie d'un appel soit l'entrée du suivant.

Coût amorti d'une opération

Définition

Soit une opération *op* dont l'éxécution produit une sortie x_s à partir d'une entrée x_e. Le *coût réel* C de *op* est la complexité temporelle de son exécution. Son *coût amorti* A est la somme du coût réel et de la variation de potentiel :

$$A = C + \phi(x_s) - \phi(x_e).$$

- Dans cette définition, les *entrées* et *sorties* représentent au sens large ce qui est donné à l'algorithme (paramètres, état de la mémoire au début de l'appel) et ce qu'il produit (résultats renvoyés, état de la mémoire en sortie).
- Une *séquence d'opérations consécutives* est une suite d'appels à un algorithme de sorte que chaque sortie d'un appel soit l'entrée du suivant.
- On montre que le coût réel est toujours inférieur au coût amorti.

Théorème d'amortissement

Théorème

Soit une suite de n opérations

$$x_0 \xrightarrow{op_1} x_1 \xrightarrow{op_2} x_2 \ldots \xrightarrow{op_n} x_n$$

à partir d'une entrée x_0 telle que $\phi(x_0) = 0$. En notant C_i le coût réel de l'opération op_i et A_i son coût amorti, on a la relation d'amortissement suivante :

$$\sum_{i=1}^{n} C_i \leq \sum_{i=1}^{n} A_i.$$

Preuve du th. d'amortissement

Démonstration.

On a

$$\begin{aligned}
\sum_{i=1}^{n} A_i &= \sum_{i=1}^{n} (C_i + \phi(x_i) - \phi(x_{i-1})) \\
&= (\sum_{i=1}^{n} C_i) + \underbrace{\phi(x_n)}_{\geq 0} - \underbrace{\phi(x_0)}_{=0} \\
&\geq \sum_{i=1}^{n} C_i.
\end{aligned}$$

Corollaire au th. d'amortissement

Corollaire

Avec les notations du th. d'amortissement, si le coût amorti est borné par une constante k, alors la complexité moyenne au sein d'une séquence arbitraire de n opérations est bornée par k.

Démonstration.

$$\sum_{i=1}^{n} C_i \underbrace{\leq}_{\text{par th. d'amortissement}} \sum_{i=1}^{n} A_i \leq k \times n.$$

Retour au compteur binaire

Complexité en nombre d'accès aux cases du tableau :

- Soit k le nombre de 1 consécutifs depuis la position 0. L'appel `incr(c,n)` réalise :

Retour au compteur binaire

Complexité en nombre d'accès aux cases du tableau :

- Soit k le nombre de 1 consécutifs depuis la position 0. L'appel `incr(c,n)` réalise :
  - $2k + 1$ accès mémoire si $k < n$.

Retour au compteur binaire

Complexité en nombre d'accès aux cases du tableau :

- Soit k le nombre de 1 consécutifs depuis la position 0. L'appel `incr(c,n)` réalise :
  - $2k + 1$ accès mémoire si $k < n$.
  - $2n$ accès si $k = n$.

Retour au compteur binaire

Complexité en nombre d'accès aux cases du tableau :

- Soit k le nombre de 1 consécutifs depuis la position 0. L'appel `incr(c,n)` réalise :
  - $2k+1$ accès mémoire si $k < n$.
  - $2n$ accès si $k = n$.
- On définit le potentiel $\phi(c)$ du tableau c par

$$\phi(c) = \alpha \times \underbrace{(\text{nombre de 1 dans } c)}_{|c|_1} \text{ avec } \alpha \text{ à déterminer}$$

Retour au compteur binaire

Complexité en nombre d'accès aux cases du tableau :

- Soit k le nombre de 1 consécutifs depuis la position 0. L'appel `incr(c,n)` réalise :
  - $2k+1$ accès mémoire si $k < n$.
  - $2n$ accès si $k = n$.
- On définit le potentiel $\phi(c)$ du tableau c par

$$\phi(c) = \alpha \times \underbrace{(\text{nombre de 1 dans } c)}_{|c|_1} \text{ avec } \alpha \text{ à déterminer}$$

- Si c_i possède k bits 1 consécutifs depuis la position 0, alors soit k' tel que $|c|_1 = k + k'$. Une application de `incr`, tranforme c_i en c_{i+1}.

Retour au compteur binaire

Complexité en nombre d'accès aux cases du tableau :

- Soit k le nombre de 1 consécutifs depuis la position 0. L'appel `incr(c,n)` réalise :
  - $2k+1$ accès mémoire si $k < n$.
  - $2n$ accès si $k = n$.
- On définit le potentiel $\phi(c)$ du tableau c par

$$\phi(c) = \alpha \times \underbrace{(\text{nombre de 1 dans } c)}_{|c|_1} \text{ avec } \alpha \text{ à déterminer}$$

- Si c_i possède k bits 1 consécutifs depuis la position 0, alors soit k' tel que $|c|_1 = k + k'$. Une application de `incr`, tranforme c_i en c_{i+1}.
- Le delta de potentiel est

$$\phi(c_{i+1}) - \phi(c_i) = \begin{cases} \alpha(k'+1) - \alpha(k+k') = \alpha - \alpha k \text{ si } k \neq n \\ 0 - \alpha k \text{ si } k = n \text{ car alors } c_{i+1} = 00\ldots0 \end{cases}$$

Retour au compteur binaire

Complexité en nombre d'accès aux cases du tableau :

- Alors

$$A_{i+1} = C_{i+1} + \phi(c_{i+1}) - \phi(c_i) = \begin{cases} \text{si } k \neq n : (2k+1) + \alpha - \alpha k \\ \text{si } k = n : 2k - \alpha k \end{cases}$$

Retour au compteur binaire

Complexité en nombre d'accès aux cases du tableau :

- Alors

$$A_{i+1} = C_{i+1} + \phi(c_{i+1}) - \phi(c_i) = \begin{cases} \text{si } k \neq n : (2k+1) + \alpha - \alpha k \\ \text{si } k = n : 2k - \alpha k \end{cases}$$

- En choisissant $\alpha = 2$, le coût amorti devient <u>constant</u> :

$$A_{i+1} = C_{i+1} + \phi(c_{i+1}) - \phi(c_i) = \begin{cases} \text{si } k \neq n : (2k+1) + 2 - 2k = 3 \\ \text{si } k = n : 2k - 2k = 0 \end{cases}$$

Retour au compteur binaire

Complexité en nombre d'accès aux cases du tableau :

- Alors

$$A_{i+1} = C_{i+1} + \phi(c_{i+1}) - \phi(c_i) = \begin{cases} \text{si } k \neq n : (2k+1) + \alpha - \alpha k \\ \text{si } k = n : 2k - \alpha k \end{cases}$$

- En choisissant $\alpha = 2$, le coût amorti devient <u>constant</u> :

$$A_{i+1} = C_{i+1} + \phi(c_{i+1}) - \phi(c_i) = \begin{cases} \text{si } k \neq n : (2k+1) + 2 - 2k = 3 \\ \text{si } k = n : 2k - 2k = 0 \end{cases}$$

- On en déduit que le coût amorti est bornée par 3. D'après le corollaire du th. d'amortissement, toute séquence d'incrémentation commençant en $00\ldots0$ (de potentiel nul) réalise en moyenne moins de 3 opérations d'accès au tableau par appel à `incr`.

Complexité moyen vs complexité amortie

- La complexité moyenne est une moyenne (donc calculée avec des probabilités) :
  Elle donne :

Complexité moyen vs complexité amortie

- La complexité moyenne est une moyenne (donc calculée avec des probabilités) :
  Elle donne :
  - Une complexité supposée représentative du plus grand nombre d'entrée ;

Complexité moyen vs complexité amortie

- La complexité moyenne est une moyenne (donc calculée avec des probabilités) :
  Elle donne :
  - Une complexité supposée représentative du plus grand nombre d'entrée ;
  - Sans apporter aucune garantie sur la complexité d'une opération particulière ni même sur une séquence particulière d'opérations.

Complexité moyen vs complexité amortie

- La complexité moyenne est une moyenne (donc calculée avec des probabilités) :
  Elle donne :
  - Une complexité supposée représentative du plus grand nombre d'entrée ;
  - Sans apporter aucune garantie sur la complexité d'une opération particulière ni même sur une séquence particulière d'opérations.
- La complexité amortie apporte une borne garantie à toute séquence d'opération.

Complexité moyen vs complexité amortie

- La complexité moyenne est une moyenne (donc calculée avec des probabilités) :
  Elle donne :
  - Une complexité supposée représentative du plus grand nombre d'entrée ;
  - Sans apporter aucune garantie sur la complexité d'une opération particulière ni même sur une séquence particulière d'opérations.
- La complexité amortie apporte une borne garantie à toute séquence d'opération.
  - Son calcul n'utilise pas de probabilité ;

Complexité moyen vs complexité amortie

- La complexité moyenne est une moyenne (donc calculée avec des probabilités) :
  Elle donne :
  - Une complexité supposée représentative du plus grand nombre d'entrée ;
  - Sans apporter aucune garantie sur la complexité d'une opération particulière ni même sur une séquence particulière d'opérations.
- La complexité amortie apporte une borne garantie à toute séquence d'opération.
  - Son calcul n'utilise pas de probabilité ;
  - Elle ne dit rien de la complexité d'une opération particulière mais assure un équilibre à toute séquence d'opérations.