Informatique pour tous - PC*

Algorithmes de tri

Florent Pompigne
pompigne@crans.org

Lycée Buffon

année 2017/2018

Plan

Motivation

Le tri est un moyen classique de structurer des données.

Motivation

Le tri est un moyen classique de structurer des données.

Exemples :

Motivation

Le tri est un moyen classique de structurer des données.

Exemples :

- Les mots dans un dictionnaire

Motivation

Le tri est un moyen classique de structurer des données.

Exemples :

- Les mots dans un dictionnaire
- Les dossiers dans une administration

Motivation

Le tri est un moyen classique de structurer des données.

Exemples :

- Les mots dans un dictionnaire
- Les dossiers dans une administration

L'intérêt est de pouvoir rechercher un élément de façon beaucoup plus efficace, grâce au principe de la recherche dichotomique

Motivation

Le tri est un moyen classique de structurer des données.

Exemples :

- Les mots dans un dictionnaire
- Les dossiers dans une administration

L'intérêt est de pouvoir rechercher un élément de façon beaucoup plus efficace, grâce au principe de la recherche dichotomique (pour rappel, de complexité logarithmique, au lieu de la complexité linéaire d'une recherche dans une liste non triée).

Spécification

Un **algorithme de tri** est un algorithme :

Spécification

Un **algorithme de tri** est un algorithme :

- prenant en argument une liste d'éléments comparables, c'est-à-dire appartenant à un ensemble totalement ordonné (par exemple, des nombres ou des mots) ;

Spécification

Un **algorithme de tri** est un algorithme :

- prenant en argument une liste d'éléments comparables, c'est-à-dire appartenant à un ensemble totalement ordonné (par exemple, des nombres ou des mots) ;
- renvoyant une liste formée des mêmes éléments, mais rangés dans l'ordre croissant.

Spécification

Un **algorithme de tri** est un algorithme :

- prenant en argument une liste d'éléments comparables, c'est-à-dire appartenant à un ensemble totalement ordonné (par exemple, des nombres ou des mots) ;
- renvoyant une liste formée des mêmes éléments, mais rangés dans l'ordre croissant.

Il existe beaucoup d'algorithmes différents répondant à cette spécification, dont on peut notamment comparer les complexités.

Spécification

Un **algorithme de tri** est un algorithme :

- prenant en argument une liste d'éléments comparables, c'est-à-dire appartenant à un ensemble totalement ordonné (par exemple, des nombres ou des mots) ;
- renvoyant une liste formée des mêmes éléments, mais rangés dans l'ordre croissant.

Il existe beaucoup d'algorithmes différents répondant à cette spécification, dont on peut notamment comparer les complexités. Dans la suite, on étudiera trois d'entre eux : le tri par insertion, le tri fusion et le tri rapide.

Spécification

Un **algorithme de tri** est un algorithme :

- prenant en argument une liste d'éléments comparables, c'est-à-dire appartenant à un ensemble totalement ordonné (par exemple, des nombres ou des mots) ;
- renvoyant une liste formée des mêmes éléments, mais rangés dans l'ordre croissant.

Il existe beaucoup d'algorithmes différents répondant à cette spécification, dont on peut notamment comparer les complexités. Dans la suite, on étudiera trois d'entre eux : le tri par insertion, le tri fusion et le tri rapide.

Un algorithme de tri est dit **en place** s'il modifie la liste argument plutôt qu'en renvoyer une nouvelle, et qu'il n'utilise qu'une quantité constante de mémoire par ailleurs.

Plan

Principe : On insère un à un les éléments dans la liste des éléments déjà triés.

Principe : On insère un à un les éléments dans la liste des éléments déjà triés.

Algorithme

Procédure tri_par_insertion
Entrées une liste L
Traitement
Soit n la longueur de L
Pour i *de 1 à* $n-1$ **faire**
    On insère $x = L[i]$ dans la sous-liste déjà triée à sa gauche.
    Pour cela, on échange x avec l'élément immédiatement à gauche, jusqu'à ce que celui-ci soit inférieur à x, ou que x soit devenu le premier élément.
FinPour

Correction

La correction de l'algorithme se démontre en utilisant pour la boucle `Pour` l'invariant I : ($L[0 : i]$ est triée et L contient les mêmes éléments qu'au départ).

Correction

La correction de l'algorithme se démontre en utilisant pour la boucle `Pour` l'invariant I : ($L[0 : i]$ est triée et L contient les mêmes éléments qu'au départ).

- I est clairement vrai au début de la première itération, puisque $L[0 : 1]$ est une liste à un élément, donc triée.

Correction

La correction de l'algorithme se démontre en utilisant pour la boucle `Pour` l'invariant I : ($L[0 : i]$ est triée et L contient les mêmes éléments qu'au départ).

- I est clairement vrai au début de la première itération, puisque $L[0 : 1]$ est une liste à un élément, donc triée.
- Si I est vrai au début d'une itération, il est encore vrai à la fin de cette itération, en supposant que l'étape d'insertion est elle-même correcte.

Correction

La correction de l'algorithme se démontre en utilisant pour la boucle `Pour` l'invariant I : ($L[0 : i]$ est triée et L contient les mêmes éléments qu'au départ).

- I est clairement vrai au début de la première itération, puisque $L[0 : 1]$ est une liste à un élément, donc triée.
- Si I est vrai au début d'une itération, il est encore vrai à la fin de cette itération, en supposant que l'étape d'insertion est elle-même correcte.
- En sortie de boucle, i vaut n, donc I implique bien que L est triée.

Complexité

- Dans le pire cas, une itération de la boucle `Pour` nécessite i comparaisons et i échanges, quand x est déplacé jusqu'au début de L.

Complexité

- Dans le pire cas, une itération de la boucle `Pour` nécessite i comparaisons et i échanges, quand x est déplacé jusqu'au début de L.

  Le nombre de comparaisons et d'échanges au total est alors de $\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$, d'où une complexité dans le pire cas en $O(n^2)$.

Complexité

- Dans le pire cas, une itération de la boucle `Pour` nécessite i comparaisons et i échanges, quand x est déplacé jusqu'au début de L.

  Le nombre de comparaisons et d'échanges au total est alors de $\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$, d'où une complexité dans le pire cas en $O(n^2)$.

  Ce pire cas est atteint quand la liste à trier est initialement décroissante.
- Dans le meilleur cas, il n'y a qu'une comparaison par itération et aucun échange (x reste à sa place), la complexité est donc en $O(n)$.

Complexité

- Dans le pire cas, une itération de la boucle `Pour` nécessite i comparaisons et i échanges, quand x est déplacé jusqu'au début de L.

  Le nombre de comparaisons et d'échanges au total est alors de $\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$, d'où une complexité dans le pire cas en $O(n^2)$.

  Ce pire cas est atteint quand la liste à trier est initialement décroissante.
- Dans le meilleur cas, il n'y a qu'une comparaison par itération et aucun échange (x reste à sa place), la complexité est donc en $O(n)$.

  Ce meilleur cas est atteint quand la liste à trier est déjà triée.
- On peut montrer que la complexité en moyenne est en $O(n^2)$.

Plan

Principe : On coupe la ljste à trier en 2, on trie chaque moitié par récurrence et on fusionne les résultats.

Fusion

Fonction fusion
Entrées Deux listes $L1$ et $L2$ supposées triées
Traitement
Soit $n1$ la longueur de $L1$, $n2$ celle de $L2$
Soit L initialisée comme liste vide
Soient $i1, i2$ initialisés à 0
Tant que $i1 + i2 < n1 + n2$ **faire**
    Si $i2 = n2$ *ou* $(i1 < n1$ *et* $L1[i1] < L2[i2])$ **alors**
        On ajoute $L1[i1]$ à la fin de L
        On incrémente $i1$
    sinon
        On ajoute $L2[i2]$ à la fin de L
        On incrémente $i2$
    FinSi
FinTantque
Sortie Renvoyer L

Tri fusion

Fonction tri_fusion
Entrée Une liste L
Traitement
Soit n la longueur de L
Si $n \leq 1$ **alors**
| Soit $R = L$
sinon
| Soit $m = n//2$
| Soit $L1 = tri\_fusion(L[0 : m])$
| Soit $L2 = tri\_fusion(L[m : n])$
| Soit $R = fusion(L1, L2)$
FinSi
Sortie Renvoyer R

Correction

Correction

- La correction de `fusion` se démontre en utilisant pour la boucle `tant que` l'invariant I : (L est triée et contient les mêmes éléments que $L1[0 : i1] + L2[0 : i2]$).

Correction

- La correction de `fusion` se démontre en utilisant pour la boucle `tant que` l'invariant I : (L est triée et contient les mêmes éléments que $L1[0 : i1] + L2[0 : i2]$).
- La correction de `tri_fusion` se montre par récurrence forte sur la longueur n de L :

Correction

- La correction de `fusion` se démontre en utilisant pour la boucle `tant que` l'invariant I : (L est triée et contient les mêmes éléments que $L1[0 : i1] + L2[0 : i2]$).
- La correction de `tri_fusion` se montre par récurrence forte sur la longueur n de L :
  - Si $n \leq 1$, on renvoie L qui est déjà triée

Correction

- La correction de `fusion` se démontre en utilisant pour la boucle `tant que` l'invariant I : (L est triée et contient les mêmes éléments que $L1[0 : i1] + L2[0 : i2]$).
- La correction de `tri_fusion` se montre par récurrence forte sur la longueur n de L :
  - Si $n \leq 1$, on renvoie L qui est déjà triée
  - Si $n > 1$, alors $n//2 < n$ et $n - (n//2) < n$, donc par hypothèse de récurrence $L1$ et $L2$ sont triées, et l'appel de `fusion` sur ces listes renvoie bien une liste triée contenant les mêmes éléments que L.

Complexité

Complexité

- Dans fusion, chaque itération de la boucle est en $O(1)$. De plus, $i1$ ou $i2$ est incrémenté jusqu'à ce que $i1 = n1$ et $i2 = n2$, il y a donc $n1 + n2$ itérations et la complexité totale est en $O(n1 + n2)$.

Complexité

- Dans fusion, chaque itération de la boucle est en $O(1)$. De plus, $i1$ ou $i2$ est incrémenté jusqu'à ce que $i1 = n1$ et $i2 = n2$, il y a donc $n1 + n2$ itérations et la complexité totale est en $O(n1 + n2)$.
- La complexité de `tri_fusion` est donc soumise à la relation de récurrence

$$C(n) = 2C(n/2) + O(n)$$

Cette complexité est donc en $O(n \ln(n))$ (dans le pire comme dans le meilleur cas, et donc également en moyenne).

Plan

Principe : On choisit arbitrairement un élément de la liste à trier, le pivot, on répartit les autres éléments en deux listes selon qu'ils soient inférieurs ou supérieurs au pivot, on trie ces deux listes par récurrence et on concatène les résultats.

Répartition

Fonction répartition
Entrées Une liste L
Traitement
Soit n la longueur de L
Soit $p = L[0]$
Soient $L1$ et $L2$ initialisées comme listes vides
Pour *chaque élément x de* $L[1 : n]$ **faire**
    Si $x < p$ **alors**
        On ajoute x à $L1$
    sinon
        On ajoute x à $L2$
    FinSi
FinPour
Sortie Renvoyer le triplet $L1, p, L2$

Tri rapide

Fonction tri_rapide
Entrée Une liste L
Traitement
Soit n la longueur de L
Si $n \leq 1$ **alors**
  Soit $R = L$
sinon
  Soient $L1, p, L2 =$repartition(L)
  Soit $R1 =$tri_rapide$(L1)$
  Soit $R2 =$tri_rapide$(L2)$
  Soit R obtenu en concaténant $R1, [p]$ et $R2$
FinSi
Sortie Renvoyer R

Correction

Correction

- La correction de `répartition` se démontre aisément avec l'invariant I : ($\forall x \in L1, x \leq p$ et $\forall x \in L2, x \geq p$ et $L1 + L2$ contient les mêmes éléments que la portion de L parcourue).

Correction

- La correction de `répartition` se démontre aisément avec l'invariant I : ($\forall x \in L1, x \leq p$ et $\forall x \in L2, x \geq p$ et $L1 + L2$ contient les mêmes éléments que la portion de L parcourue).
- La correction de `tri_rapide` se démontre par récurrence forte sur la longueur n de L :

Correction

- La correction de `répartition` se démontre aisément avec l'invariant I : ($\forall x \in L1, x \leq p$ et $\forall x \in L2, x \geq p$ et $L1 + L2$ contient les mêmes éléments que la portion de L parcourue).
- La correction de `tri_rapide` se démontre par récurrence forte sur la longueur n de L :
  - Si $n \leq 1$, on renvoie L qui est déjà triée.

Correction

- La correction de `répartition` se démontre aisément avec l'invariant I : ($\forall x \in L1, x \leq p$ et $\forall x \in L2, x \geq p$ et $L1 + L2$ contient les mêmes éléments que la portion de L parcourue).
- La correction de `tri_rapide` se démontre par récurrence forte sur la longueur n de L :
  - Si $n \leq 1$, on renvoie L qui est déjà triée.
  - Si $n > 1$, alors p existe bien, et $L1$ et $L2$ sont de longueurs strictement inférieures à n. Par hypothèse de récurrence, $R1$ et $R2$ sont donc triées, et $R1 + [p] + R2$ est aussi triée d'après la spécification de `répartition`.

Complexité

Complexité

- On montre facilement que `répartition` a une complexité en $O(n)$ dans le meilleur comme dans le pire cas.

Complexité

- On montre facilement que `répartition` a une complexité en $O(n)$ dans le meilleur comme dans le pire cas.
- Dans le pire cas, on a toujours l'une des listes $L1$ et $L2$ vide et l'autre contenant tous les éléments de L sauf p. La complexité de `tri_fusion` est alors soumise à la relation de récurrence

$$C(n) = C(n-1) + O(n)$$

La complexité dans le pire cas est donc en $O(n^2)$.

Complexité

- On montre facilement que `répartition` a une complexité en $O(n)$ dans le meilleur comme dans le pire cas.
- Dans le pire cas, on a toujours l'une des listes $L1$ et $L2$ vide et l'autre contenant tous les éléments de L sauf p. La complexité de `tri_fusion` est alors soumise à la relation de récurrence

$$C(n) = C(n-1) + O(n)$$

  La complexité dans le pire cas est donc en $O(n^2)$.
- Dans le meilleur cas, les listes $L1$ et $L2$ sont toujours de tailles égales, et la complexité de `tri_fusion` est donc soumise à la relation de récurrence

$$C(n) = 2C(n/2) + O(n)$$

  La complexité dans le meilleur cas est donc en $O(n \ln(n))$.

Complexité

- On montre facilement que `répartition` a une complexité en $O(n)$ dans le meilleur comme dans le pire cas.
- Dans le pire cas, on a toujours l'une des listes $L1$ et $L2$ vide et l'autre contenant tous les éléments de L sauf p. La complexité de `tri_fusion` est alors soumise à la relation de récurrence

$$C(n) = C(n-1) + O(n)$$

  La complexité dans le pire cas est donc en $O(n^2)$.
- Dans le meilleur cas, les listes $L1$ et $L2$ sont toujours de tailles égales, et la complexité de `tri_fusion` est donc soumise à la relation de récurrence

$$C(n) = 2C(n/2) + O(n)$$

  La complexité dans le meilleur cas est donc en $O(n \ln(n))$.
- On peut montrer que la complexité en moyenne est en $O(n \ln(n))$.

Plan

Tri	Pire cas	Meilleur cas	En moyenne
Insertion	$O(n^2)$	$O(n)$	$O(n^2)$
Fusion	$O(n\ln(n))$	$O(n\ln(n))$	$O(n\ln(n))$
Rapide	$O(n^2)$	$O(n\ln(n))$	$O(n\ln(n))$

Tri	Pire cas	Meilleur cas	En moyenne
Insertion	$O(n^2)$	$O(n)$	$O(n^2)$
Fusion	$O(n\ln(n))$	$O(n\ln(n))$	$O(n\ln(n))$
Rapide	$O(n^2)$	$O(n\ln(n))$	$O(n\ln(n))$

En pratique, dans sa variante optimisée en place, le tri rapide est le tri le plus souvent utilisé, sa complexité moyenne étant meilleure que celle du tri fusion d'un facteur constant.

Tri	Pire cas	Meilleur cas	En moyenne
Insertion	$O(n^2)$	$O(n)$	$O(n^2)$
Fusion	$O(n\ln(n))$	$O(n\ln(n))$	$O(n\ln(n))$
Rapide	$O(n^2)$	$O(n\ln(n))$	$O(n\ln(n))$

En pratique, dans sa variante optimisée en place, le tri rapide est le tri le plus souvent utilisé, sa complexité moyenne étant meilleure que celle du tri fusion d'un facteur constant.

Le tri fusion a l'avantage de garantir une complexité en $O(n\ln(n))$ sur toute entrée.

Tri	Pire cas	Meilleur cas	En moyenne
Insertion	$O(n^2)$	$O(n)$	$O(n^2)$
Fusion	$O(n\ln(n))$	$O(n\ln(n))$	$O(n\ln(n))$
Rapide	$O(n^2)$	$O(n\ln(n))$	$O(n\ln(n))$

En pratique, dans sa variante optimisée en place, le tri rapide est le tri le plus souvent utilisé, sa complexité moyenne étant meilleure que celle du tri fusion d'un facteur constant.

Le tri fusion a l'avantage de garantir une complexité en $O(n\ln(n))$ sur toute entrée.

Le tri par insertion est efficace dans un contexte où les listes traitées sont déjà presque triées. Il est également plus efficace que les autres tris sur les listes de petite taille.