

Les méthodes de conception des algorithmes

Rappels

Rappels sur les structures de données :

- piles, files, listes, ABR
- Arbres 2-3-4, ARN, B-Arbres
- Binomial Trees
- Tas binômiaux, tas binaires, tas de Fibonacci
- Tables de hachage
- SD pour les ensembles (listes, graphes)

On peut classer les SD dans deux catégories, les SD orientées opération de dictionnaire, ou les structures de contrôle.

1 le paradigme “Divide and Conquer”

C'est une classe de conception des algorithmes qui procède de la manière suivante :

1. On divise l'entrée en plusieurs entrées plus petites
2. On résout les sous-problèmes séparément d'une manière récursive
3. On combine les sous-résultats pour former le résultat global.

C'est une méthode simple est performante.

Pour analyser le coût d'un algorithme basé sur ce paradigme, on a recours aux équations de récurrence, c'est à dire le coût d'une instance est déterminé en fonction des instances plus petites.

1.1 Exemple du tri fusion

A la base le tri est une opération quadratique. En utilisant un tri fusion - basé sur ce paradigme -, on réduit le coût de $O(n^2)$ en $O(n \cdot \log(n))$.

Pour exemple d'application, de nombreux sites utilisent une technique appelée technique des filtres collaboratifs (collaborative filtering), pour comparer nos préférences avec celles des autres utilisateurs.

Pour vous proposer des nouveaux choix en fonction des choix de ces utilisateurs (qui vous ressemblent).

Le but qui nous intéresse ici est d'implémenter des outils de méta-recherche pour résoudre une requête en faisant appel à plusieurs sites simultanément. On synthétise ensuite les résultats pour déceler les similarités et les différences et classer les résultats des différents moteurs de recherche.

On peut exprimer ce problème de la manière suivante : Soient la séquence suivante de n nombres :

$$a_1, \dots, a_n$$

tous différents.

On veut définir une mesure qui permet de déterminer de combien cette séquence est loin de la séquence quand elle est ordonnée d'une manière ascendante.

$$0 < a_1 < \dots < a_n$$

Le coût de l'algorithme est $O(n \cdot \log(n))$.

Il s'agit de compter les inversions (i.e. permutations). 2 indices i et j forment une inversion.

Si $a_j < a_i$ et $i < j$, c'est à dire si a_i et a_j ne sont pas ordonnés, il y a C_2^n paires à examiner.

On doit exprimer chaque couple (a_i, a_j) et déterminer s'il constitue une inversion, cela coûte $O(n^2)$, mais on peut l'améliorer en $O(n \cdot \log(n))$.

En utilisant le paradigme D&C :

1. On divise la liste en deux sous-listes
2. On compte le nombre d'inversions dans les deux sous-listes séparément
3. On compte les inversions lorsque les éléments des paires (a_i, a_j) appartiennent aux deux sous-listes.

Compter_Par_D&C(L)

Si L contient 1 seul élément Alors

Arrêter

Sinon

Diviser L en deux sous-listes A et B

A contient les premiers $\lfloor n/2 \rfloor$ éléments et B les suivants

r_A ← Compter_Par_D&C(A)

r_B ← Compter_Par_D&C(B)

r ← Combiner_Et_Compter(A,B)

Finsi

Retourner r

Fin

Combiner_Et_Compter(A,B)

CurseurA ← Debut(A)

CurseurB ← Debut(B)

```
CptInversion <- 0
TantQue les deux listes sont non vides Faire
  a <- valeur(CurseurA)
  b <- valeur(CurseurB)
  Si b < a Alors
    CptInversion <- CptInversion + taille restante de A
  Finsi
  faire avancer le curseur des listes sur le plus petit (?)
FintantQue
Fin
```