

Algorithmique et Programmation avancée

Diviser pour régner

Application au problème du tri

Le problème du tri est le suivant:

Étant donné une liste de n nombres $[a_1, a_2, \dots, a_n]$, déterminer une permutation $[a'_1, a'_2, \dots, a'_n]$ de la liste telle que $[a'_1 \leq a'_2 \leq \dots \leq a'_n]$. Ainsi à partir de la suite $[31, 41, 59, 26, 41, 58]$, un algorithme de tri correct produit en sortie la suite $[26, 31, 41, 41, 58, 59]$.

Il existe plusieurs façons de concevoir un tel algorithme:

- Le "brut force" qui consiste à tester toutes les permutations possible de la liste. $O(n!)$
- La méthode incrémentale: le tri par insertion. $O(n^2)$
- La méthode "Diviser pour régner" (Divide and conquer): Le tri-fusion. $O(n \log n)$
- Et bien d'autres ...

Le tri par insertion

Le tri par insertion est efficace lorsqu'il s'agit de trier un petit nombre d'éléments. Il s'inspire de la façon dont on trie des cartes à jouer.

On peut voir ça ainsi: Les nombres non triés sont les cartes en main droite, la main gauche est vide. On place au fur et à mesure les cartes de la main droite, à la bonne place dans la main gauche.

Exemple 1. $[5, 2, 4, 6, 1, 3]$

Pseudo-code:

triInsertion(li):

#Entrées: Liste $[a_1, a_2, \dots, a_n]$

#Sorties: Permutation $[a'_1, a'_2, \dots, a'_n]$ triée

Pour j allant de 2 à n

$clé = li[j]$ #Carte à placer

$i = j - 1$

 Tant que $i > 0$ et $li[i] > clé$

$li[i + 1] = li[i]$ #On décale les cartes déjà triées

$i = i - 1$

$li[i + 1] = clé$ #On place la carte

On peut dénombrer le nombre d'étapes: Pour chaque j , les affectation prennent $\Theta(1)$ étapes (un nombre constant), et la boucle "tant que" compte t_j étapes où t_j correspond au nombre de cartes déplacées pour placer la j -ème, on a donc $t_j = O(n)$, et l'algorithme ne dépasse pas $O(n^2)$ étapes. Inversement on constate que si la liste est initialement triée à l'envers, alors $t_j = j$ et on dénombre au moins $\sum_{j=2}^n j = \Theta(n^2)$ étapes. Il existe une instance du problème (liste triée à l'envers) pour lequel la complexité est $\Omega(n^2)$. On en déduit que la complexité du tri par insertion est $\Theta(n^2)$ dans le pire des cas.

Le tri-fusion

Beaucoup d'algorithmes utilisent la récursivité: pour résoudre un problème, ils s'appellent eux-mêmes pour traiter des sous-problèmes similaires. C'est le cas des algorithmes "diviser pour régner". Le paradigme "diviser pour régner" implique trois étapes:

- Diviser le problème en un certain nombre de sous-problèmes qui sont des instances plus petites du même problème.

- Régner sur les sous-problèmes en les résolvant de manière récursive. Si la taille d'un problème est suffisamment petite, on peut le résoudre directement.
- Combiner les solutions des sous-problèmes pour produire la solution du problème initial.

Dans le cas du tri fusion, on aura:

- Diviser la suite de n éléments à trier en deux sous-suite de $n/2$ éléments chacune.
- Régner/Trier les deux sous-suites de manière récursive en utilisant le tri fusion
- Combiner/Fusionner les deux sous-suite triées pour produire la sous-suite triée.

La récursivité stoppe quand la séquence à trier a une longueur 1 puisqu'une suite de longueur 1 est déjà triée.

Illustration

On a le pseudo code suivant (composé de deux fonctions):

```

fusion(G,D):
#Entrées: 2 Listes  $G$  et  $D$  triées
#Sorties: la liste triée composée des éléments de  $G$  et de  $D$ 
    Si une liste est vide, renvoyer l'autre
    Si  $G[1] < D[1]$ 
        renvoyer la liste qui commence par  $G[1]$  suivie de fusion( $G'$ , $D$ )
    Sinon
        renvoyer la liste qui commence par  $D[1]$  suivie de fusion( $G$ , $D'$ )

```

```

triFusion(li):
#Entrées: Liste  $[a_1, a_2, \dots, a_n]$ 
#Sorties: Permutation  $[a'_1, a'_2, \dots, a'_n]$  triée
    Si la liste est de taille 1, renvoyer la liste # initialisation
    Sinon:
        Diviser  $li$  en deux moitiés
        Appliquer triFusion sur chaque des moitiés
        Renvoyer la fusion des deux

```

On peut alors décrire le temps d'exécution de l'algorithme par une récurrence: Soit $T(n)$ le temps d'exécution d'un problème de taille n , chaque étape diviser génère deux sous-suites de taille $n/2$. L'étape fusion prend $\Theta(n)$ étapes. La récurrence peut s'exprimer ainsi:

$$T(n) = \begin{cases} \Theta(1) & \text{pour } n = 1 \\ 2T(n/2) + \Theta(n) & \text{sinon} \end{cases}$$

On la résout par un arbre récursif: on considère que chaque étape basique prend un temps c .

On obtient une complexité en $\Theta(n \log n)$.

TD/TP: Algorithmique et Programmation avancée

2: Diviser pour régner

1: Un peu de code

A partir des pseudo-codes du cours, coder en Python3 les 2 algorithmes de tri proposés.

2: Arbres récursifs

Utiliser des arbres récursifs pour déterminer une bonne borne asymptotique supérieure pour les récurrences :

$$T(n) = 3T(n/2) + n.$$

$$T(n) = T(n/2) + n^2.$$

$$T(n) = 4T(n/2 + 2) + n.$$

$$T(n) = 2T(n - 1) + 1.$$

On supposera que n est une puissance de 2.

3: Sous tableau maximal

On considère un tableau de n entiers relatifs, par exemple:

A	13	-3	-25	20	-3	-16	-23	18	20	-7	12	-5	-22	15	-4	7
---	----	----	-----	----	----	-----	-----	----	----	----	----	----	-----	----	----	---

Il peut modéliser, par exemple, des variations en bourse au cours du temps. En gardant cet exemple, on suppose que l'on cherche la période de temps sur laquelle le gain serait maximal, ie, le sous-tableau connexe de somme maximale.

1: Expliciter la méthode "brut force". Calculer sa complexité (borne sup et inf). Écrire son pseudo code. Le coder en Python3.

2: Utiliser la méthode diviser pour régner pour trouver un algorithme plus efficace. Calculer sa complexité. Écrire son pseudo code. Le coder en Python3.

3: Trouver un algorithme encore plus efficace. Calculer sa complexité. Écrire son pseudo code. Le coder en Python3.

Correction

2: Arbres récursifs

$$T(n) = 3T(n/2) + n \rightarrow O(n^{\log_2 3})$$

$$T(n) = T(n/2) + n^2 \rightarrow O(n^2).$$

$$T(n) = 4T(n/2 + 2) + n \rightarrow O(n^2).$$

$$T(n) = 2T(n - 1) + 1 \rightarrow O(2^n).$$