Algorithmes diviser-pour-régner

Idée: On prend un problème et on le casse en sous-problèmes. On résout les sous-problèmes (possiblement en les cassant de nouveau) et on combine les résultats pour obtenir la solution au problème original.

C'est une approche de haut en bas.

- Construction générale: Si l'instance à résoudre est suffisamment petite, on utilise un algorithme classique pour la résoudre, sinon, on la décompose en morceaux (si possible de même taille) et on rappelle l'algorithme sur ces morceaux pour ensuite les recombiner.

Algorithmes diviser-pour-régner - efficacité

3 conditions pour avoir un algorithme diviser-pour-régner efficace:

1) Bien décider quand utiliser l'algorithme simple sur de petites instances plutôt que les appels récursifs

2) La décomposition d'une instance en sous-instances et la recombinaison des sous-solutions doivent être efficaces

3) Les sous-instances doivent être, autant que possible, environ de la même taille

Algorithmes diviser-pour-régner - complexité

Souvent la complexité en temps d'un algorithme diviser-pour-régner sur une instance de taille *n* peut s'écrire comme:

$$T(n) = bT(n/b) + g(n)$$

où $g(n)$ est le temps prit pour casser et reconstruire.

Si on peut montrer que $g(n) \in \Theta(n^k)$ pour un certain k alors le théorème suivant nous donne automatiquement la complexité de l'algorithme:

Soit $T : \mathbb{N} \longrightarrow \mathbb{R}^+$ une fonction éventuellement non décroissante telle que

$$T(n) = \ell T(\frac{n}{b}) + cn^k, \forall n > n_0,$$

où $\frac{n}{n_0}$ est une puissance de b. Alors,

$$T(n) \quad \in \quad \begin{cases} \Theta(n^k) & \text{si } \ell < b^k \\ \Theta(n^k \log_b n) & \text{si } \ell = b^k \\ \Theta(n^{\log_b \ell}) & \text{si } \ell > b^k \end{cases}$$

Algorithmes diviser-pour-régner - recherche binaire

Problème: Soit L[1 .. n] une liste d'entiers triée en ordre croissant i.e. on a $L[i] \leq L[j], \quad \forall 1 \leq i \leq j \leq n$. Le problème consiste à trouver un entier x dans la liste, ou, si l'entier x n'est pas dans la liste, de trouver la position où il devrait être inséré.

Exemple: Trouver 7 dans la liste triée suivante:

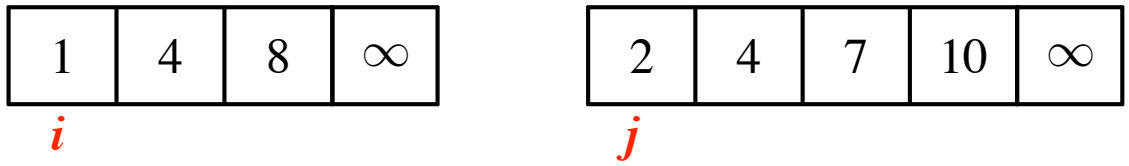
1	2	3	4	5	6	7	8	9	10	11
-5	-2	0	3	8	8	9	12	12	26	31
i					*k*					*j*

Algorithmes diviser-pour-régner - trier

Problème: Trier une liste d'entiers L[1 .. n]

Solution Tri-fusion: Diviser la liste en deux. Trier ces parties par appels récursifs et fusionner les solutions de chaque partie en faisant attention de conserver l'ordre.

Fusion:



Algorithmes diviser-pour-régner - tri rapide (Quicksort)

Idée: L'idée est de choisir un élément du tableau L comme pivot de mettre les éléments plus grand que ce pivot à sa droite et les autres éléments à sa gauche. Par appels récursifs, on ordonne ensuite les éléments de chaque côté du pivot.

Pour balancer la taille des sous-parties à trier, on aimerait utiliser la médiane des éléments comme pivot.

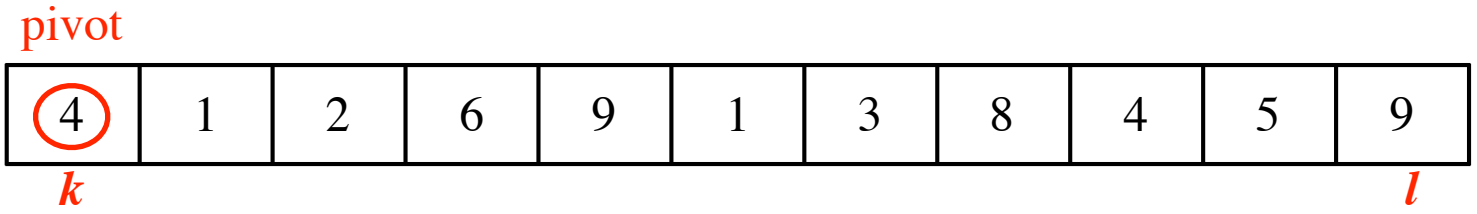
- Trop coûteux

Donc, on choisit arbitrairement le pivot en espérant le mieux.

Pour que tri rapide soit compétitif, il faut absolument un pivotage linéaire avec une très petite constante cachée

Algorithmes diviser-pour-régner - tri rapide (Quicksort)

Pivotage:



- Deux pointeurs k, initialisé à 1, l initialisé à taille(L).
- Bouger k vers la droite jusqu'à un élément > pivot
- Bouger l vers la gauche jusqu'à un élément $\leq$ pivot
- Échanger L[k] et L[l] et répéter tant que $k < l$
- Échanger pivot et L[l]