Diviser pour régner

Les algorithmes de tri

Présentation générale

- Méthode de résolution des problèmes en fournissant un algorithme récursif
- La structure générale se subdivise en 3 étapes

1. La décomposition du problème en un certain nombre de sous-problèmes de taille réduite
2. Les appels récursifs: appliquer récursivement la fonction sur chacune des nouvelles entrées et retourner les k solutions $s_1, \ldots, s_k$
3. La reconstitution des solutions partielles aux sous-problèmes en la solution *s* au problème

Multiplication des grands entiers

- Entier représenté sur des centaines d'octets
- Opérations élémentaires
  - Lecture d'1 bit
  - Modification d'1 bit
  - L'accès au bit suivant
  - Suppression du bit de poids faible (division par 2 noté n>>1)
  - Insertion d'1 nouveau bit de poids faible (multiplication par 2 noté 2<<1)

Pour ajouter le bit 0 derriere un nombre (a) en binaire on le multiplie par deux en decimale (2a) multiplier, et pour ajouer 1 on le multiplie par deux puis on ajoute 1 (2a+1).

Il vient donc que pour lui enlever son bit de poids faible il faut le diviser par deux.

Premier algorithme

```
Function produit(a,b:entier):entier
  resultat←0;
  tantque b ≠ 0 faire
        si estImpair(b) alors
                resultat ← addition(resultat,a);
        a<<1; -- on multiplie a par 2
        b>>1; -- on divise b par 2
  fintq
  retourner resultat;
```

Analyse de l'algorithme

- La fonction addition est linéaire en la taille des entrées
- Le produit est quadratique
  - $\Theta(\text{taille(a)}\times\text{taille(b)})=\Theta(n^2)$ avec n=taille(a)=taille(b)
- Peut-on faire mieux?

$$a = a_{n-1} \dots a_0 = \sum_{k=0}^{n-1} a_k 2^k = \sum_{k=0}^{\frac{n}{2}-1} a_k 2^k + \sum_{k=\frac{n}{2}}^{n-1} a_k 2^k = \sum_{k=0}^{\frac{n}{2}-1} a_k 2^k + 2^{\frac{n}{2}} \times \sum_{k=0}^{n-1-\frac{n}{2}} a_{k+\frac{n}{2}} 2^k$$

$$= \quad a_2 + 2^{\frac{n}{2}} \times a_1 \; avec \; a_2 = \sum_{k=0}^{\frac{n}{2}-1} a_k 2^k = a_{\frac{n}{2}} \dots a_0 \quad et$$

$$= \sum_{k=0}^{n-1-\frac{n}{2}} a_{k+\frac{n}{2}} 2^k = \sum_{k=\frac{n}{2}}^{n-1} a_k 2^{k-\frac{n}{2}} = a_{n-1} \dots a_{n-\frac{n}{2}}$$

Seconde version

- n=max(taille(a),taille(b))
- On suppose n pair
- $a = a_1 2^{n/2} + a_2$
  - a_1 est composé des n/2 bits de poids faible
  - a_2 des n/2 bits de poids fort
- Décomposons b de la même façon que a
- $a\times b=a_1b_12^n+(a_1b_2+a_2b_1)2^{n/2}+a_2b_2$
- $a\times b= (a_1+a_2)(b_1+b_2)2^{n/2}+a_1b_1(2^n-2^{n/2})+a_2b_2(1-2^{n/2})$

Seconde version

- Pour multiplier a par b il faut
- 1. Décomposer l'entrée (a,b) en trois nouvelles entrée

$$(a_1 + a_2, b_1 + b_2) \quad (a_1, b_1) \quad (a_2, b_2)$$

- 2. Appliquer récursivement le produit sur chacune des entrées.
- 3. Recomposer le résultat s de la manière suivante

$$s = s_1 \cdot 2^{n/2} + s_2 \cdot 2^n - s_2 \cdot 2^{n/2} + s_3 - s_3 \cdot 2^{n/2}$$

Analyse de la complexité

- Décomposition $\Theta(n)$
- 3 appels récursifs sur des problèmes de taille n/2
- Reconstitution de la solution finale
  - 2 additions, 2 soustractions, 4 décalages droits
  - O(n) pour chaque opération => $\Theta(n)$
  - pour la reconstitution d'où f(n) = n + 3×f(n/2) et f(1)=1

$$f(n) = n + 3\left(\frac{n}{2} + 3\left(\frac{n}{2^2} + \ldots\right)\right) = n\left(\left(\frac{3}{2}\right)^0 + \left(\frac{3}{2}\right)^1 + \ldots\right) \approx n \cdot \left(\frac{3}{2}\right)^{\ln_2(n)} = n \cdot n^{\ln(3/2)/\ln 2}$$

$$f(n) = n^{\ln_2(3)} \approx n^{1,58}$$

Une autre forme de "master théorem".

Proposition

- Soit $f : \mathrm{N} \rightarrow R^{+}$

$$\begin{cases} f(n_0) = d \\ f(n) = af(n/b) + cn^k \end{cases} \text{avec } n_0 \geq 1, b \geq 2 \quad \text{et des réels } k \geq 0, a > 0, c > 0, d > 0$$

$n > n_0$ et n/n_0 est une puissance de b

Alors on a:

$$f(n) = \begin{cases} \theta(n^k) \text{ si } a < b^k \\ \theta(n^k \log n) \text{ si } a = b^k \\ \theta(n^{\log_b a}) \text{ si } a > b^k \end{cases}$$

Tri Rapide (Quicksort)

- Il est fondé sur le paradigme diviser-pour-régner
- les trois étapes du processus diviser-pour-régner sont employées pour trier un sous-tableau typique A[p . . r].

1. Partitionnement/Diviser
   - Le tableau A[p . . r] est partitionné (réarrangé) en deux sous-tableaux (éventuellement vides) A[p . . q − 1] et A[q + 1. . r] tels que chaque élément de A[p . . q − 1] soit inférieur ou égal à A[q] qui, lui-même, est inférieur ou égal à chaque élément de A[q + 1. . r].
   - L'indice q est calculé dans le cadre de cette procédure de partitionnement.
2. Régner
   - Les deux sous-tableaux A[p . . q−1] et A[q+1 . . r] sont triés par des appels récursifs au tri rapide.
3. Combiner
   - Comme les sous-tableaux sont triés sur place, aucun travail n'est nécessaire pour les recombiner : le tableau A[p . . r] tout entier est maintenant trié.

Algorithme

- TRI-RAPIDE(*A, p, r*)
  1 **si** $p < r$ **alors**
  2 $q \leftarrow$ PARTITION(*A, p, r*)
  3 TRI-RAPIDE(*A, p, q* − 1)
  4 TRI-RAPIDE(*A, q* + 1*, r*)
- Pour trier un tableau *A* entier, l'appel initial est TRI-RAPIDE(*A,* 1*, longueur*[*A*]).

Partitionnement

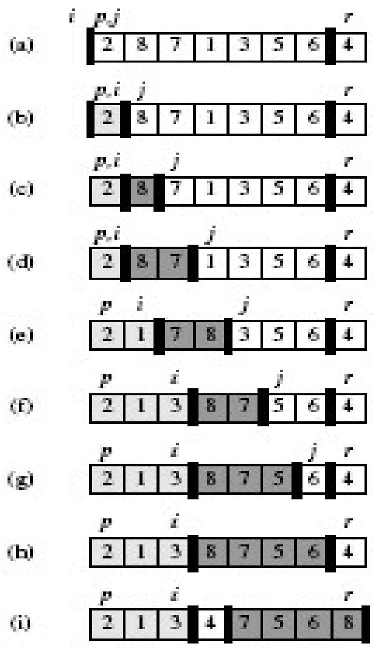
- Le point principal de l'algorithme est la procédure PARTITION, qui réarrange le sous-tableau A[p . . r] sur place.
- PARTITION(A, p, r)
- 1 x ← A[r]
- 2 i ← p − 1
- 3 **pour** j ← p à r − 1 **faire**
- 4 **si** A[j] ≤ x **alors**
- 5 i ← i + 1
- 6 permuter A[i] ↔ A[j]
- fsi
- fpour
- 7 permuter A[i + 1] ↔ A[r]
- 8 retourner i + 1

Analyse de l'algorithme Partition

- Au début de chaque itération de la boucle des lignes 3–6, pour tout indice *k*,
- 1) Si $p \leq k \leq i$, alors $A[k] \leq x$.
- Sur les lignes 7–8, l'élément pivot est permuté de façon à aller entre les deux partitions.
- 2) Si $i + 1 \leq k \leq j - 1$, alors $A[k] > x$.
- 3) Si $k = r$, alors $A[k] = x$.

Exercice

- Appliquer le partitionnement sur le tableau suivant
- (2 8 7 1 3 5 6 4)



Performance du tri rapide

- cas le plus défavorable
  - la routine de partitionnement produit un sous problème à $n-1$ éléments et une autre avec 0 élément.
  - Supposons que ce partitionnement non équilibré survienne à chaque appel récursif. Le partitionnement coûte $\Theta(n)$
  - l'appel récursif sur un tableau de taille 0 rend la main sans rien faire, $T(0) = \Theta(1)$ et la récurrence pour le temps d'exécution est
    - $\mathbf{T(n)} = T(n-1)+T(0)+\Theta(n) = \mathbf{T(n-1)+\Theta(n)= \Theta(n^2)}$.

Cas défavorable vs tri par insertion

- Le temps d'exécution du tri rapide n'est donc pas meilleur, dans le cas le plus défavorable, que celui du tri par insertion.
- En outre, ce temps d'exécution de $\Theta(n^2)$ se produit quand le **tableau d'entrée est déjà complètement trié**
- Dans cette même situation le **tri par insertion** s'exécute en un temps $O(n)$.

Cas favorable

- PARTITION produit deux sous-problèmes de taille non supérieure à *n/*2
- La récurrence du temps d'exécution est alors $T(n) \leq 2T(n/2) + \Theta(n)$,
- la solution en est $T(n) = O(n \lg n)$.

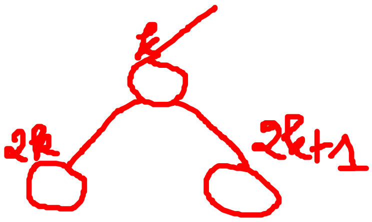
Tri par Tas

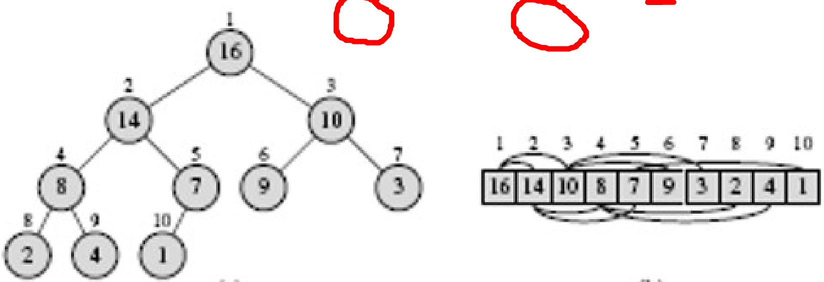
- Rappeler la structure d'arbre et la représentation sous forme d'un tableau
- Revenir sur le propriétés des arbres complets
- Algorithme du tri pas tas
  - Construction du tas
  - Manipulation du tas

Définition

- La structure de tas (binaire) est un tableau qui peut être vu comme un arbre binaire presque complet
- Chaque nœud de l'arbre correspond à un élément du tableau qui contient la valeur du noeud.
- Un tableau A représentant un tas est un objet ayant deux attributs :
  - longueur[A], nombre d'éléments du tableau,
  - et taille[A], nombre d'éléments du tas rangés dans le tableau A.
  - A[1 . . longueur[A]] contient des nombres valides,
  - Aucun élément après A[taille[A]], où taille[A] ≤ longueur[A], n'est un élément du tas.

Exemple de tas





(a) (b)

Fonctions usuelles

- La racine de l'arbre est $A[1]$
- Étant donné l'indice i d'un noeud,
  - PARENT(i) : **retourner** $i/2$
    - décaler i d'une position binaire vers la droite
  - GAUCHE(i): **retourner** $2i$
    - décaler simplement d'une position vers la gauche la représentation binaire de i
  - DROITE(i): **retourner** $2i+1$
    - décaler d'une position vers la gauche la représentation binaire de i et en ajoutant un 1 comme bit de poids faible
- Exercice de programmation:
  - Écrire chacune de ces fonctions en langage C

Propriété des tas

- Dans un ***tas max***, la ***propriété de tas max*** est que, pour chaque noeud i autre que la racine, $A[\text{PARENT}(i)] \geq A[i]$,
  - En d'autres termes, la valeur d'un noeud est au plus égale à celle du parent.
  - Ainsi, le plus grand élément d'un tas max est stocké dans la racine,
  - et le sous-arbre issu d'un certain noeud contient des valeurs qui ne sont pas plus grandes que celle du nœud lui-même.
- Un ***tas min*** est organisé en sens inverse ; la ***propriété de tas min*** est que, pour chaque noeud i autre que la racine, $A[\text{PARENT}(i)] \leq A[i]$.
  - Le plus petit élément d'un tas min est à la racine.

Hauteur d’un tas

- la ***hauteur*** d’un noeud dans un tas se définit comme le nombre d’arcs sur le chemin simple le plus long reliant le noeud à une feuille
- On définit la hauteur du tas comme étant la hauteur de sa racine.
- Comme un tas de *n* éléments est basé sur un arbre binaire complet, sa hauteur est $\Theta(\lg n)$

CONSERVATION DE LA STRUCTURE DE TAS

- ENTASSER-MAX est un sous-programme qui prend en entrée un tableau *A* et un indice *i*.
- Quand ENTASSER-MAX est appelée, on suppose que les arbres binaires enracinés en GAUCHE(*i*) et DROITE(*i*) sont des tas max, mais que *A*[*i*] peut être plus petit que ses enfants, violant ainsi la propriété de tas max.
- Le rôle de ENTASSER-MAX est de faire «descendre» la valeur de *A*[*i*] dans le tas max de manière que le sous-arbre enraciné en *i* devienne un tas max.

Algorithme

- ENTASSER-MAX(A, i)
- $l \leftarrow$ GAUCHE(i)
- $r \leftarrow$ DROITE(i)
- **si** $l \leq taille[A]$ et $A[l] > A[i]$
-     **alors** $max \leftarrow l$
-     **sinon** $max \leftarrow i$
- **si** $r \leq taille[A]$ et $A[r] > A[max]$
-     **alors** $max \leftarrow r$
- **si** $max \neq i$
-     **alors** échanger $A[i] \leftrightarrow A[max]$
-         ENTASSER-MAX(A, max)

Exercice

- Illustrer l'action de ENTASSER-MAX(A, 3) sur le tableau A = [27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0].

Analyse de l'efficacité

- en un noeud *i* donné est le temps $\Theta(1)$ nécessaire pour corriger les relations entre les éléments $A[i]$, $A[\text{GAUCHE}(i)]$, et $A[\text{DROITE}(i)]$, plus le temps d'exécuter ENTASSERMAX sur un sous-arbre enraciné sur l'un des enfants du noeud *i*.
- Les sous-arbres des enfants ont chacun une taille au plus égale à $2n/3$ (le pire des cas survient quand la dernière rangée de l'arbre est remplie exactement à moitié), et le temps d'exécution
- de la procédure ENTASSER-MAX peut donc être décrit par la récurrence $T(n) \leq T(2n/3) + \Theta(1)$.
- La solution de cette récurrence, d'après le théorème général est $T(n) = O(\lg n)$.
- On peut également caractériser le temps d'exécution de ENTASSER-MAX sur un noeud de hauteur *h* par $O(h)$.
  - En effet, la descente s'effectue sur une branche de l'arbre
  - Or nous avons vu que la hauteur h de l'arbre est le plus long chemin de la racine à une feuille où h=lg n

CONSTRUCTION D'UN TAS

- On peut utiliser la procédure ENTASSER-MAX à l'envers pour convertir un tableau A[1 . . n], avec n = length[A], en tas max.
- En remarquant que les éléments du sous-tableau A[(n/2 + 1) . . n] sont tous des feuilles de l'arbre
  - chacun est initialement un tas à 1 élément.
- La procédure CONSTRUIRE-TAS-MAX parcourt les autres noeuds de l'arbre et appelle ENTASSER-MAX pour chacun.

Algorithme

- CONSTRUIRE-TAS-MAX(A)
- 1 *taille*[A] ← *longueur*[A]
- 2 **pour** i ← *longueur*[A]/2 **jusqu'à** 1
- 3 **faire** ENTASSER-MAX(A, i)
- Simuler l'exécution de Construire-Tas sur le tableau A=[4 1 3 2 16 9 10 14 8 7]
- Temps d'exécution (majorant)
  - Chaque appel à ENTASSER-MAX coûte $O(\lg n)$,
  - il existe $O(n)$ appels de ce type.
  - Le temps d'exécution est donc $O(n \lg n)$.
  - Ce majorant, quoique correct, n'est pas asymptotiquement serré.

ALGORITHME DU TRI PAR TAS

- TRI-PAR-TAS(A)
- 1 CONSTRUIRE-TAS-MAX(A)
- 2 **pour** i ← *longueur*[A] **jusqu'à** 2
- 3 **faire** échanger $A[1] \leftrightarrow A[i]$
- 4 *taille*[A] ← *taille*[A] − 1
- 5 ENTASSER-MAX(A, 1)
- La procédure TRI-PAR-TAS prend un temps $O(n\lg n)$
  - l'appel à CONSTRUIRE-TAS-MAX prend un temps $O(n)$
  - chacun des $n - 1$ appels à ENTASSER-MAX prend un temps $O(\lg n)$.

A éviter

- **Quand ne pas utiliser l'approche diviser-pour-regner (avec récursivité)**
- Pour que l'approche diviser-pour-regner *avec* récursivité conduise à une solution efficace,
- il ne faut pas que l'une ou l'autre des conditions suivantes survienne :
  1. Un problème de taille *n* est décompose en deux ou plusieurs sous-problèmes eux-même de taille presque *n* (par ex., *n* -1).
  2. Un problème de taille *n* est décompose en *n* sous problèmes de taille *n/c* (pour une constante $c \geq 2$).

Travaux Pratiques

- Programmer les algorithmes vus en cours
  - Tri rapide (par segmentation)
  - Tri par Tas
  - Recherche dichotomique dans un tableau trié
- Contraintes
  - Langage de programmation: le C
  - Délai de réalisation: 14 jours
  - Livrables: ***code sources, jeux de test, un fichier readme*** qui détaille l'exploitation de votre programme