

UNITE IN3ST02 TD5 - Corrigé

Exercice 2.1 - Tri rapide (Quick Sort)

Invariant possible pour une résolution directe :

$$I(k, m) = (t[i..k-1] \leq t[k]=e_1 < t[k+1..m])$$

L'inconvénient de cet invariant est qu'il conduit à des permutation circulaire à 3 éléments. On peut éviter cela par une résolution en deux temps :

$$\begin{aligned} & (t=[e_1, e_{i+1}, \dots, e_j] \wedge i < j) \\ & \{ \text{pré-séparer}(t, i, j, k) \} \\ & (t[i+1..j]=\Pi(e_{i+1}, \dots, e_j) \wedge t[i+1..k] \leq t[i]=e_1 < t[k+1..j]) \\ & \{ \text{échanger } t[i] \text{ et } t[k] \} \\ & (t[i..j]=\Pi(e_1, \dots, e_j) \wedge t[i..k-1] \leq t[k]=e_1 < t[k+1..j]) \end{aligned}$$

Réolvons pré-séparer :

Invariant :

$$I1(k, m) = (t[i+1..k] \leq t[i]=e_1 < t[k+1..m])$$

Condition d'arrêt : $m=j$

car alors $I1(k, m) = \text{post-condition}$

Corps de boucle

- $(I1(k, m) \wedge t[m+1] \leq t[i])$
 $\{ \text{Echanger } t[k+1] \text{ et } t[m+1] \}$
 $(I1(k+1, m+1))$
- $(I1(k, m) \wedge t[m+1] > t[i]) \{ \} (I1(k, m+1))$

Initialisations :

$$\begin{aligned} k & \leftarrow i \\ m & \leftarrow i \\ // I(k, m) \end{aligned}$$

On prouve aisément que l'algorithme est de complexité linéaire.

Exercice 2.2 - Tri par fusion

Invariant :

$$I(k_1, k_2, k) = ((t[0..k-1]=t_1[0..k_1-1] \cup t_2[0..k_2-1]) \wedge t[0..k-1] \text{ trié })$$

Condition d'arrêt : $k_1=n_1$ ou $k_2=n_2$

Pour atteindre la propriété $I(k_1, k_2, k) = \text{post-condition}$, il restera alors à compléter $t[k..n_1+n_2-1]$ par les éléments du tableau t_1 ou t_2 qui n'a pas été complètement vidé par la boucle.

Corps de boucle

- $(I(k_1, k_2, k) \wedge t_1[k_1] > t_2[k_2])$
 $\{ t[k] \leftarrow t_2[k_2] \}$
 $(I(k_1, k_2+1, k+1))$
- $(I(k_1, k_2, k) \wedge t_1[k_1] \leq t_2[k_2])$
 $\{ t[k] \leftarrow t_1[k_1] \}$
 $(I(k_1+1, k_2, k+1))$

Initialisations

$$\begin{aligned} k_1 & \leftarrow 0 \\ k_2 & \leftarrow 0 \\ k & \leftarrow 0 \\ // I(k_1, k_2, k) \end{aligned}$$

On prouve aisément que l'algorithme est de complexité linéaire.

Exercice 2.3 - Tri par tas (Heap Sort)

2.3.1-

Fonction estFeuille(i, k) : booléen
retourner $(2*i+1 \geq k)$

Fonction estInterne1(i, k) : booléen
retourner $(2*i+1 = k-1)$

Fonction estInterne2(i, k) : booléen
retourner $(2*i+2 < k)$

2.3.2-

Fonction estDominant(t, i, k) : booléen
 $fd \leftarrow 2*(i+1)$ // fils droit
 $fg \leftarrow fd - 1$ // fils gauche
 si $fd < k$ alors retourner $(t[i] \geq t[fg] \text{ et } t[i] \geq t[fd])$ // cas 2 fils
 si $fg < k$ alors retourner $(t[i] \geq t[fg])$ // cas 1 fils
 retourner vrai // cas 0 fils

FinFonction.

2.3.3 - rétablirTas(t, j, k)

Procédure « rétablir $t[j..k-1]$ est un tas » sachant que tout nœud autre que j du sous-arbre de racine j de l'abpp $t[0..k-1]$ est dominant

Invariant

$$I(j') = (\text{ tous les nœuds du sous-arbre de racine } j \text{ de l'abpp } t[0..k-1] \text{ sont dominants sauf éventuellement le nœud } j')$$

Condition d'arrêt : j' est dominant dans $t[0..k-1]$
 car alors : $I(j') \wedge \text{estDominant}(t, j', k)$
 $\rightarrow t[j..k-1]$ est un tas dans l'abpp $t[0..k-1]$

Corps de boucle

$$// I(j') \wedge j' \text{ non dominant}$$

```

jStar ← indice du fils de j', dans t[0..k-1], de clé max
permuter(t[j'], t[jStar]) ;
// I(jStar)
j' ← jStar
// I(j')

```

Initialisation

```

j' ← j
// I(j, k)

```

Complexité

Au pire, la valeur initiale de t[j] chemine jusqu'à une feuille de l'arbre. La longueur parcourue est en $O(\lfloor \lg(k-j) \rfloor)$. Cette longueur est le nombre d'itérations du corps de boucle, lui-même en temps constant. La procédure rétablirTas est donc de complexité logarithmique.

2.3.4 - construireTas(t, n)**Invariant**

$I(i) = (\forall j \in [i..n-1] \Rightarrow \text{racineTas}(t, j, n))$

Condition d'arrêt : $i = 0$

car alors : $I(i) \rightarrow t[0..n-1]$ est un tas

Corps de boucle

```

// I(i) ∧ i ≠ 0
i ← i-1
// I(i+1) ∧ i ≥ 0
rétablirTas(t, i, n)
// I(i)

```

Initialisation

```

i ← ⌊n/2⌋ // n ≥ 1
// I(i) car toutes les feuilles sont trivialement dominantes

```

Remarque :

la procédure rétablirTas ne peut pas être appliquée de la racine vers les feuilles car sa pré-condition n'a aucune raison d'être vérifiée partout

Complexité

Chaque appel rétablirTas est en $O(\lg n)$, et il existe $\theta(n)$ appels de ce type. Le temps d'exécution est donc en $O(n \lg n)$. Mais ce majorant, quoique correct, n'est pas asymptotiquement très serré.

On peut trouver un majorant plus fin en observant que le temps d'exécution de rétablirTas sur un nœud dépend de la hauteur h de ce nœud dans l'arbre (on rappelle que la hauteur d'un nœud est le nombre d'arcs du chemin le plus long qui relie ce nœud à une feuille), et que les hauteurs de la plupart des nœuds sont faibles. On s'appuie également sur les propriétés d'un tas de n éléments :

- sa hauteur est $\lfloor \lg n \rfloor$
- le nombre de nœuds à la hauteur h est au plus de $\lceil n / 2^{h+1} \rceil$

Le temps requis par rétablirTas appelé sur un nœud de hauteur h étant $O(h)$, le coût total de la procédure construireTas est :

$$\begin{aligned} & \text{Somme de } h=0 \text{ à } \lfloor \lg n \rfloor \text{ des } (\lceil n / 2^{h+1} \rceil O(h)) \\ & = O(n * \text{Somme de } h=0 \text{ à } \lfloor \lg n \rfloor \text{ des } (h / 2^h)) \end{aligned}$$

Sachant que lorsque $|x| < 1$:

$$\text{Somme de } h=0 \text{ à } +\infty \text{ des } x^k = 1 / (1-x)$$

on en déduit, en dérivant et en multipliant par x, que :

$$\text{Somme de } h=0 \text{ à } +\infty \text{ des } kx^k = x / (1-x)^2$$

et donc, en prenant $x=1/2$, que :

$$\text{Somme de } h=0 \text{ à } +\infty \text{ des } h / 2^h = 2$$

le temps d'exécution de construireTas peut donc être borné par :

$$\begin{aligned} & O(n * \text{Somme de } h=0 \text{ à } \lfloor \lg n \rfloor \text{ des } h / 2^h) \\ & = O(n * \text{Somme de } h=0 \text{ à } +\infty \text{ de } h / 2^h) \\ & = O(n) \end{aligned}$$

L'algorithme construireTas est donc de complexité linéaire.

2.3.5 - triParTas(t, n)**Invariant**

$I(i) = ((t[0..i-1] \text{ est un tas}) \wedge (t[0..i-1] \leq t[i..n-1]) \wedge (t[i..n-1] \text{ trié}))$

Procédure triParTas(t[0..n-1])

```

construireTas(t,n) ; // O(n)
i ← n
// I(i)
tant que i > 1 faire
// I(i) ∧ i > 1
i ← i-1
// I(i+1) ∧ i ≥ 1
permuter(t[0], t[i])
// t[0..i-1] ≤ t[i..n-1] ∧ t[i..n-1] trié ∧ (tout nœud
// de t[1..i-1] est racine d'un tas dans t[0..i-1])
rétablirTas(t, 0, i) // O(lg(i))
// I(i)
finTantQue
// t[0..n-1] trié

```

Complexité

$$\begin{aligned} T_{\text{boucle tant-que}}(n) & \leq T_{\text{test}} + T_{i-1} + T_{\text{permuter}} + (a + b * \lg(n-1)) + T_{\text{test}} \\ & \quad + (a + b * \lg(n-2)) + T_{\text{test}} \\ & \quad + \dots \\ & \quad + (a + b * \lg(1)) + T_{\text{test}} \\ & = A + n * B + b * (\lg(1) + \lg(2) + \dots + \lg(n-1)) \\ & = A + n * B + b * \lg(1 * 2 * \dots * (n-1)) = A + n * B + b * \lg((n-1)!) \end{aligned}$$

Comme $n! = (2\pi n)^{1/2} (n/e)^n (1 + O(1/n))$ [formule de Stirling]

il s'en suit :

$$T_{\text{boucle tant-que}}(n) \leq \alpha + \beta.n + \delta.\lg(n) + \gamma.n.\lg(n)$$

La procédure trierParTas est donc en $O(n.\lg(n))$. Les algorithmes de tri comparatifs optimaux étant en $\Omega(n.\lg(n))$, il s'en suit que le tri par tas est en $\theta(n.\lg(n))$ et est donc optimal.

Exercice 2.4 - Tri de liste par éclatement-fusion

Les procédures fendre et fusion sont de complexité linéaire. Donc, le temps d'exécution de l'algorithme triFusion appliqué à une liste de n éléments se définit par la relation de récurrence :

$$T(0) = T(1) = a$$

$$T(n>1) = 2 T(n/2) + b + c*n$$

Le développement de cette récurrence conduit à la forme suivante :

$$T(n>1) = A + B n + C n \lg(n)$$

L'algorithme de triFusion est donc en $\theta(n.\lg(n))$