TP7

Notes : Pour cette séance, notre notation suppose que le premier index d'un tableau est 0

Numéro 1

Problème 7.13 b), p.252, livre Brassard-Bratley: Donnez un algorithme pour la fusion de deux listes triées U et V qui n'utilise pas de sentinels et fonctionne en temps linéaire en la somme des tailles de U et V.

Solution

Plutôt que de rajouter un élément avec une valeur infinie (sentinelles) à la fin des tableaux U et V, nous traitons explicitement les cas où $i == m$ et $j == n$. L'algorithme modifié est le suivant:

Code

```
def merge(U,V):
    i,j,m,n = 0,0, len(U), len(V)
    T = [0 for x in range(m+n)]
    for k in range(m+n):
        if i == m  :
            T[k] = V[j]
            j += 1
        elif j == n :
            T[k] = U[i]
            i += 1
        else :
            if U[i] < V[j] :
                T[k] = U[i]
                i += 1
            else :
                T[k] = V[j]
                j += 1
    return T
```

Numéro 2

Problème 7.14, p.252, livre Brassard-Bratley : Nous avons vu en classe un algorithme de fusion capable de fusionner deux listes triées U et V en temps linéaire en la somme des tailles de U et V. Donnez un autre algorithme de fusion fonctionnant aussi en temps linéaire mais n'utilisant pas de liste auxiliaire: les sections $T[1..k]$ et $T[k+1..n]$ d'une liste sont triées indépendamment et vous devez trier la liste complète $T[1..n]$ en utilisant seulement un nombre fixe de mémoire supplémentaire

Solution

Ce problème est plus complexe que l'on pourrait le croire. Voici l'algorithme, initialement proposé, qui n'utilise qu'une quantité constante de mémoire additionnelle pour fusionner deux liste triées $T[1..k]$ et $T[k+1..n]$.

Code - Premier essai

```
def fusion_special1(T,k):
    i,j,n = 0,k, len(T)
    while (i < j and j < n):
        if T[i] <= T[j] :
            i += 1
        else :
            jj = j
            t = T[i]
            while (j < n and T[j] < t) :
                transpose(T,i,j)
                i += 1
                j += 1
            if (i < jj ) :
                j = jj
            else :
                j -=1
    return T
```

Solution (suite)

Malheureusement, bien que cette procédure fonctionne sur bon nombre d'entrées, elle peut ne pas fonctionner lorsque l'entrée contient des doublons. Par exemple, certaines instances comme `0 2 4 1 1 3`. Dans ce cas, le résultat est `0 1 1 2 4 3`. Un autre contre-exemple est `2 3 3 1 1 2`.

Code - Deuxième essai

Problème : Complexité un peu plus que linéaire

```
def fusion_special2(T,k):
    i,j,n = 0,k, len(T)

    while (i < j and j <n):
        if T[i] <= T[j] :
            i += 1
        else :
            jj,t = j, T[i]
            transpose(T,i,jj)
            while (jj+1 < n and T[jj+1] < t):
                transpose(T,jj,jj+1)
                jj +=1
            i +=1
    return T
```

Solution (suite)

Une solution à ce problème est disponible dans l'article suivant :

Huang & Michael A. Langston, *Practical in-place merging*. Commun. ACM vol. 31 issue 3, p. 348–362, 1988.

Numéro 3

Problème 7.20, p.253, livre Brassard-Bratley:} Une liste T contient n éléments. Vous voulez trouver le m plus petits éléments oèu m est beaucoup plus petit que n. Est-ce que vous aller

1. Ordonnez T et retourner les m premiers éléments
2. Appelez Sélection(T, i), pour $1 \leq i \leq m$, ou bien
3. Utilisez une autre méthode?

Justifiez votre réponse.

Solution

Nous proposons une troisième alternative. Elle consiste à appeler Sélection(T, m) puis à parcourir linéairement T et de stocker toutes les valeurs (distinctes) plus petites ou égales à la valeur obtenue via Sélection(T, m). L'algorithme tri finalement ces valeurs. Voici son pseudocode:

Code

```
def selection_m(T,m):
    n = len(T)
    selection(T,m)
    T[:m].sort()
    return T[:m]
```

Solution (suite)

Le temps de calcul des algorithmes est respectivement de $O(n \log n)$, $O(mn)$ et $O(n + m \log m)$. Supposons que $m \in O(\log^c n)$ pour $c \in \mathbb{N}$ dans le but de formaliser le concept de m beaucoup plus petit que n, nous avons donc les temps de calculs suivants:

	$c = 0$	$c = 1$	$c \geq 2$
Choix a	$n \log n$	$n \log n$	$n \log n$
Choix b	n	$n \log n$	$n \log^c n$
Choix c	n	n	n

Ainsi, dans tous ces cas, le choix c est préférable.

Numéro 4

Problème 7.21, p.253, livre Brassard-Bratley:} Une liste T contient n éléments. Vous voulez trouver les éléments des rangs $\lceil \frac{n}{2} \rceil$, $\lceil \frac{n}{2} \rceil + 1$, ..., $\lceil \frac{n}{2} \rceil + m - 1$. Est-ce que vous aller

a) Ordonnez T et retourner les éléments appropriés

b) Appelez Sélection(T, i) m fois, pour, pour $\lceil \frac{n}{2} \rceil \leq i \leq \lceil \frac{n}{2} \rceil + m - 1$, ou bien

c) Utilisez une autre méthode?

Justifiez votre réponse.

Solution

Nous proposons une troisième alternative. Notons que l'algorithme `Sélection`(T, i) modifie T durant son exécution de telle sorte que $T[i]$ est la i $\{$ème$\}$ plus petite valeur, $T[j] \leq T[i]$ pour tout $j \leq i$ et $T[k] \geq T[i]$ pour tout $k > i$. Nous utilisons ce fait pour construire un algorithme. Nous utilisons la notation $T[i : j]$ pour décrire un sous-tableau T' de taille $j - i + 1$ contenant les valeurs $T[i]$ à $T[j]$.

Code

```
def selection_m(T,m):
    n = len(T)
    i = ceil(n/2.0)
    selection(T,i)
    selection(T[i:],m-1)
    T[i:i+m].sort()
    return T[i:i+m]
```

Solution (suite)

Le temps de calcul des algorithmes est respectivement de $O(n \log n)$, $O(mn)$ et $O(n + m \log m)$. Supposons que $m \in O(\log^c n)$ pour $c \in \mathbb{N}$ dans le but de formaliser le concept de m beaucoup plus petit que n, nous avons donc les temps de calculs suivants:

	$c = 0$	$c = 1$	$c \geq 2$
Choix a	$n \log n$	$n \log n$	$n \log n$
Choix b	n	$n \log n$	$n \log^c n$
Choix c	n	n	n

Ainsi, dans tous ces cas, le choix c est préférable.

Numéro 5

Problème: Démontrez que $\{$*quicksort*$\}$ *prend, en moyenne, un temps* $\mathcal{O}(n \log n)$ *pour ordonner* n *éléments.*

Solution

Voir autre document