Université de Technologie de Compiègne — Documents non autorisés
Examen médian NF16, Automne 2009 — Durée 2 heures

Dans un souci d'équité, il ne sera pas répondu à des questions pendant la durée de l'épreuve (sauf aux étudiants étrangers ayant des difficultés pour comprendre le français qui seront aidés pour la compréhension du sujet). Si l'énoncé vous semble comporter une imprécision, faites un choix pour la lever que vous indiquerez clairement dans votre copie. Assurez-vous par une relecture attentive de l'énoncé que la réponse à votre question n'y figure pas. **Chaque partie est à rédiger sur une copie séparée, un point vous sera retranché sinon**.

Exercice I (copie numéro I : 4 points). On me demande de calculer C_{100}^{50} mais j'ai oublié mes cours de maths et j'ai oublié la définition et la formule des C_n^p. Tout ce dont je me souviens est que C_n^p est défini pour $0 \leq p \leq n$, que C_n^0 vaut 1 ainsi que C_n^n, et que l'on a la propriété : $C_n^p = C_{n-1}^p + C_{n-1}^{p-1}$ quand p ne vaut ni 0 ni n. J'utilise la fonction suivante :

```
fonction cnp (n, p : Naturel) : Naturel
si p = 0 ou n = p alors      retourner 1
sinon      retourner cnp (n - 1, p) + cnp (n - 1, p - 1)
finsi
```

Mais cette fonction présente un défaut majeur. Lequel ? Donner une autre version qui n'a pas ce défaut en n'utilisant aucune autre propriété des C_n^p que celles données dans l'énoncé.

Exercice II (copie numéro II). Partie I : 8 points. Les listes écologiques.

Soit les déclarations suivantes :

```
typedef struct cell {int val;  struct cell * suc;} T_cell;
typedef struct ecolist {int taille, trie;   T_cell * t_l, * t_l_recup;} T_ecolist;
```

Et les fonctions :

```
void echange(int *a, int *b) { int inter;  inter = *a;  *a=*b;  *b=inter;}
T_cell * creer_cell()   {  return  (T_cell *)malloc(sizeof(T_cell)); }
```

Le champ **t_l** pointe sur la cellule de tête de la liste utile et **t_l_recup** pointe sur la cellule de tête de la liste de « récupération ». Initialement ces listes sont vides (la valeur est à NULL). Quand on ajoute une donnée, on crée par défaut une cellule avec la fonction **creer_cell()**, et ainsi de suite tant que l'on ne supprime pas de cellule de la liste dont la tête est **t_l.** Cependant, quand on supprime une cellule de la liste **t_l** alors elle est détachée de la liste **t_l** et attachée en tête de la cellule **t_l_recup**. Quand on ajoutera une nouvelle donnée dans **t_l**, il faudra donc d'abord vérifier que la liste **t_l_recup** est vide. Si elle est vide on créera une nouvelle cellule, mais si elle ne l'est pas on détachera la cellule de tête de la liste **t_l_recup** pour l'utiliser.

Une liste écologique de type **T_ecolist** gère donc **deux** listes : une « utile » dont la tête est le champ **t_l** et une de cellule détruites qui peuvent être recyclées **t_l_recup**.

Vous remarquerez les champs **taille** qui contient la taille courante de la liste **t_l** et le champ **trie** qui vaut 0 si la liste **t_l** n'est pas triée et 1 si elle est triée.

Question 1 (1 point) : Ecrire la fonction dont l'entête est **T_ecolist * creer_eco_list();** qui crée dynamiquement une variable structurée de type **T_ecolist** et l'initialise avec **taille**=0, **trie**=0, **t_l** = NULL et **t_l_recup** = NULL.

Question 2 (1,5 points) : Ecrire les fonctions dont les entêtes sont **T_cell * detacher_recup(T_ecolist * l)** et **T_cell * detacher(T_ecolist * l)** qui retournent un pointeur vers la cellule détachée. On suppose dans les

deux cas que les pointeurs correspondant ne sont pas à NULL. Les champs **t_l,** et **t_l_recup** seront correctement mis à jour dans chaque cas.

Question 3 (1,5 points) : Ecrire les fonctions dont les entêtes sont **void attacher_recup(T_ecolist * l, T_cell * c)** et **void attacher(T_ecolist * l, T_cell * c)** qui attache la cellule **c** en tête de la liste de récupération dans le premier cas et en tête de la liste utile dans le second. Les champs **t_l**, et **t_l_recup** seront correctement mis à jour dans chaque cas.

Question 4 (1,5 points) : Ecrire les fonctions dont les entêtes sont **int ajouter(T_ecolist * l, int val) et int retirer(T_ecolist * l, int * val)**. Elles utilisent les fonctions précédentes. La première ajoute une nouvelle cellule en tête de la liste utile **t_l** avec la valeur val en recyclant en priorité une cellule de la liste **t_l_recup**. La seconde retire la cellule de tête de la liste utile **t_l** et l'ajoute en tête de la liste **t_l_recup**, le deuxième paramètre permettra de récupérer la valeur **val** détruite. Elles retourneront la valeur 1 si l'opération s'est bien passée, la valeur 0 sinon.

Question 5 (1 point) : Ecrire la fonction « bulle », qui parcourt une unique fois la liste utile d'une liste écologique et qui échange les contenus val de deux cellules consécutives si la valeur de la première est supérieure à celle de la seconde. Si aucun échange n'a lieu alors la liste est triée et on mettra à jour le champ **trie** de la liste écologique transmise. Peut-on affirmer que la liste est forcément triée après l'usage de la fonction « bulle » ? Quelle est la complexité de « bulle » en fonction du champ taille ?

Question 6 (1,5 points) : Ecrire la fonction « cherche » qui recherche la cellule qui contient la valeur x au sein d'une liste écologique. Pour optimiser les recherches on commence par utiliser « bulle » si la liste n'est pas triée. Ensuite, si la liste est triée on en tient compte pour arrêter la recherche au bon moment. Si la liste est triée et si x n'est pas dans la liste **t_l,** quelle condition permet d'arrêter la recherche ? Quelle est la complexité de cette fonction dans tous les cas ? En fin de compte est-ce que cette façon de faire améliore la complexité au pire cas ?

Partie II (8 points) : La recherche de la kième plus petite valeur d'un tableau **T** est un problème classique en algorithmique. On dit que l'on cherche la valeur de **rang k**. Dans cette partie nous vous proposons d'étudier deux algorithmes qui permettent de résoudre le problème. Nous considérons un tableau **T** de **n** nombres. Les indices du tableau **T** vont de **0** à **n-1** comme en langage C. Nous disposons de l'algorithme echange(a,b) qui échange le contenu des variables a et b.

Soit l'algorithme suivant :

```
entier partition(T : tableau, debut : entier, fin : entier)
  pivot, i, pivotNouveauIndice : entier

  pivotNouveauIndice := fin
 si (debut=fin) retourner debut
 sinon
  pivot := T[debut]
  i := debut+1
  // point d'observation 1
  tant_que  (i ≠ pivotNouveauIndice) faire
      si (T[i]>=pivot)
      alors
          echange(T[pivotNouveauIndice], T[i])
          pivotNouveauIndice := pivotNouveauIndice - 1
      sinon  i := i+1
      fin_si
      // point d'observation 2
   fin_tant_que
  si (T[pivotNouveauIndice]>pivot)
  alors
      pivotNouveauIndice := pivotNouveauIndice - 1
```

```
      echange(T[debut], T[pivotNouveauIndice])
   sinon echange(T[debut],T[pivotNouveauIndice])
   fin_si
  // point d'observation 3
  retourner pivotNouveauIndice
 fin_si
```

Cet algorithme comporte en commentaire trois points d'observation, où l'on désire observer l'état des variables **i, pivotNouveauIndice** et du tableau **T** en entier.

Attention on suppose que cet algorithme est invoqué avec debut <= fin. **Le cas debut > fin est donc impossible.**

Question 1 (2 points) : faire fonctionner l'algorithme sur le tableau T = {707, 703, 706, 709, 710, 703, 710, 710, 711, 704} avec debut = 0 et fin = 9. Aux points d'observation que valent les variables i, pivotNouveauIndice et que contient le tableau T ?

Question 2 (2 points) : prouver la terminaison de cet algorithme sachant que debut <= fin quand on l'invoque.

Question 3 (1 point) : donner sa complexité en argumentant simplement.

Nous introduisons maintenant l'algorithme récursif :

```
entier selection(T : tableau,  debut : entier, fin : entier, ind_rang : entier)

 entier pivotNouveauIndice, pivotIndice : entier

 pivotNouveauIndice = partition(T, debut, fin)
 // point  d'observation 1
 si (ind_rang = pivotNouveauIndice)
        retourner T[rang]
 sinon
  si (ind_rang < pivotNouveauIndice)
      retourner selection(T, debut , pivotNouveauIndice-1, ind_rang)
  sinon
      retourner selection(T, pivotNouveauIndice+1, fin , ind_rang)
```

Nous ne considérons plus les points d'observation de l'algorithme partition.

En revanche, nous introduisons le point d'observation 1 au sein de l'algorithme **selection**. Nous voulons y observer les contenus des variables **debut**, **fin**, **rang**, **pivotNouveauIndice** et le contenu du tableau **T**.

Attention : les tableaux sont indicés à partir de l'indice 0.
Le paramètre **ind_rang** est l'indice du tableau dans lequel nous aurons la valeur telle que :
o toutes les valeurs du tableau T comprises dans les indices {debut, debut+1, ... , ind_rang-1} seront strictement inférieures à la valeur T[ind_rang]
o toutes les valeurs du tableau comprises dans les indices {ind_rang+1, ind_rang+1, ... , fin} seront supérieures ou égales à la valeur rangée dans T[ind_rang].

Question 4 (1,5 points) : Soit T = {7, 4, 11, 9, 10, 3, 3, 10, 12, 6}, debut = 0, fin = 9, ind_rang = 3, donner les valeurs des variables observées et le contenu du tableau au point d'observation 1 ainsi que la valeur retournée quand on invoque **selection(T , 0, 9, 3).**

Question 5 (1,5 points) : Considérons que la taille d'un tableau T est un nombre pair n. Donner la complexité de l'algorithme selection si on l'applique avec ind_rang = n/2 ET si le tableau T est déjà trié par ordre croissant mais qu'on ne le sait pas. Expliquer.