IN 101
18 décembre 2009

Contrôle

ENSTA ParisTech

NOM : **Prénom :**

- tous les documents (poly, slides, TDs, livres, brouillon du voisin...) sont **interdits**.
- les réponses doivent toutes tenir dans les cases prévues dans l'énoncé (pas de feuilles supplémentaires).
- le code des algorithmes/fonctions pourra être donné soit en C soit en pseudo-code. En C la syntaxe n'a pas besoin d'être exacte, mais le code doit être présenté clairement (accolades, indentation, écriture lisible...). En pseudo-code, aucune syntaxe n'est imposée, mais tout devra être suffisamment détaillé pour ne pas laisser de place aux ambiguïtés (écrire "parcourir les sommets du graphe G" n'est pas assez précis, mais "parcourir les voisins du sommet S" est correct).
- le barème actuel de ce contrôle aboutit à une note sur 95 points. La note finale sur 20 sera dérivée de la note sur 95 grâce à une fonction croissante qui n'est pas encore définie (cela ne sert à rien de la demander).
- n'oubliez pas de remplir votre nom et votre prénom juste au dessus de ce cadre.

Exercice 1 : QCM

(20 points)

Chaque bonne réponse rapporte 1 point. Chaque mauvaise réponse enlève 1 point. Si vous n'êtes pas certains de votre réponse, ne répondez pas au hasard, la note totale peut être négative !

1.a] Laquelle de ces complexités de ne fait pas partie des complexités dites *polynomiales* ?

☐ $\Theta(\log n)$, ☐ $\Theta(n^4)$, ☐ $\Theta(2^{\log n})$, ☐ $\Theta(n^{\log n})$.

1.b] Quelle est la complexité du meilleur algorithme pour le calcul de la puissance n-ième d'une matrice 2×2 d'entiers (en supposant que les calculs d'entiers se font en temps constant) ?

☐ $\Theta(\log n)$, ☐ $\Theta(n)$, ☐ $\Theta(n \log n)$, ☐ $\Theta(n^3)$.

1.c] La *programmation dynamique* permet d'améliorer certain algorithmes en :

☐ diminuant leur complexité temporelle et augmentant leur complexité spatiale,
☐ augmentant leur complexité temporelle et augmentant leur complexité spatiale,
☐ diminuant leur complexité temporelle et diminuant leur complexité spatiale,
☐ augmentant leur complexité temporelle et diminuant leur complexité spatiale.

1.d] Combien de comparaisons fera un algorithme de tri par insertion (bien implémenté) si on lui passe en argument un tableau de n entiers déjà trié ?

☐ $\log n$, ☐ $n - 1$, ☐ $2n$, ☐ $\frac{n(n-1)}{2}$.

1.e] Pour laquelle de ces entrées un algorithme de tri fusion fera-t-il le moins de comparaisons ?

☐ des entiers déjà triés,
☐ des entiers inversement triés,
☐ $\frac{n}{2}$ entiers croissants puis $\frac{n}{2}$ décroissants,
☐ le même nombre dans tous les cas.

1.f] Laquelle de ces conditions pourrait-elle permettre d'avoir un algorithme de tri *par comparaisons* de n entiers ayant une complexité en moyenne inférieure à $\Theta(n \log n)$?

- ☐ les entiers à trier sont uniquement entre 0 et 100000,
- ☐ on accepte d'avoir un entier qui n'est pas à sa place à la fin de l'algorithme,
- ☐ l'ordre des entiers est fortement biaisé,
- ☐ les $\frac{n}{2}$ premiers entiers sont déjà triés.

1.g] Lequel de ces algorithmes de tri peut se faire *en place* et a une complexité de $\Theta(n \log n)$ dans le pire cas ?

- ☐ le tri rapide,
- ☐ le tri à bulle,
- ☐ le tri par paquets,
- ☐ le tri par tas.

1.h] On range n références dans une table de hachage utilisant une fonction de hachage de $[1, m]$ dans $[1, k]$. En supposant que les références sont réparties uniformément dans la table, quel est le coût moyen de la suppression d'une référence ?

☐ $\Theta(\frac{n+m}{k})$, ☐ $\Theta(n \times \frac{m}{k})$, ☐ $\Theta(\frac{m}{k})$, ☐ $\Theta(\frac{n}{k})$.

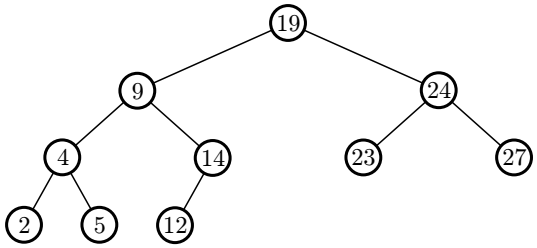

FIGURE 1 – Un arbre.

1.i] Combien l'arbre de la FIGURE 1 contient-il de nœuds et de feuilles ?

- ☐ 10 nœuds, 5 feuilles,
- ☐ 4 nœuds, 6 feuilles,
- ☐ 5 nœuds, 5 feuilles,
- ☐ 10 nœuds, 6 feuilles.

1.j] Dans quel ordre s'affichent les nœuds de l'arbre de la FIGURE 1 si on les affiche à l'aide d'un parcours *préfixe* ?

- ☐ 2, 4, 5, 9, 12, 14, 19, 23, 24, 27,
- ☐ 2, 5, 4, 12, 14, 9, 23, 27, 24, 19,
- ☐ 19, 9, 4, 2, 5, 14, 12, 24, 23, 27,
- ☐ 27, 24, 23, 19, 9, 14, 12, 5, 4, 2.

1.k] Si on effectue une rotation *à droite* à la racine de l'arbre de la FIGURE 1, quel nœud sera le fils gauche du nœud 19 après la rotation ?

☐ 9, ☐ 14, ☐ 23, ☐ 12.

1.l] Considérons que l'arbre de la FIGURE 1 est un arbre AVL. Quel nœud ne peut-on pas supprimer sans avoir à effectuer un rotation pour rééquilibrer l'arbre ?

- ☐ le nœud 23,
- ☐ le nœud 14,
- ☐ le nœud 9,
- ☐ aucun des nœuds ne déséquilibre l'arbre si on l'enlève.

1.m] Combien de fils possède le plus petit nœud d'un tas ?

☐ 0, ☐ 0 ou 1, ☐ 1 ou 2, ☐ 0, 1 ou 2.

```
1  int func(int n) {
2    int i,j,k = 0;
3    for (i=2; i<n; i++) {
4      for (j=2; j*j<=i; j++) {
5        if (i%j == 0) {
6          break;
7        }
8      }
9      if (j*j > i) {
10       k = k+i;
11     }
12   }
13   return k;
14 }
```

1.n] Que calcule la fonction `func` ?

- ☐ la somme des entiers de 2 à n,
- ☐ la somme des nombres premiers plus petits que n,
- ☐ la somme des diviseurs premiers de n,
- ☐ la somme des entiers plus petits que $\sqrt{n}$.

1.o] Quelle est la complexité en moyenne de la fonction `func` ?

- ☐ $\Theta(\log n)$,
- ☐ $\Theta(n)$,
- ☐ entre $\Theta(n)$ et $\Theta(n^2)$,
- ☐ $\Theta(n^2)$.

1.p] La matrice d'adjacence d'un graphe non-orienté est toujours :

☐ diagonale, ☐ symétrique, ☐ triangulaire, ☐ de rang plein.

1.q] Un parcours en profondeur permet de faire le tri topologique d'un graphe. Pour que cela puisse fonctionner, le graphe doit être :

☐ connexe, ☐ non-orienté, ☐ réflexif, ☐ sans cycles.

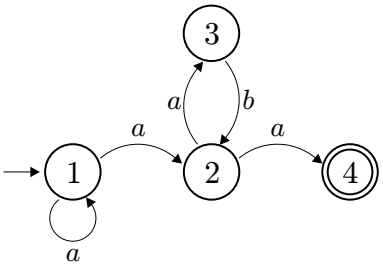

FIGURE 2 – Un automate.

1.r] Quel est le langage reconnue par l'automate non-déterministe de la FIGURE 2 ?

☐ $a^+a(ba)^*$, ☐ $a^*(ab)^*a$, ☐ $a^*a(ab|a)^*$, ☐ $a^+a(ab)^*a$.

1.s] Quel est le nombre maximum d'états que contient l'automate obtenu en déterminisant un automate non-déterministe à n états ?

☐ $2n-1$, ☐ n^2-1, ☐ 2^n-1, ☐ $n!-1$.

1.t] Pour que l'insertion soit la moins coûteuse possible dans une table de hachage, il faut que son facteur de remplissage soit :

- ☐ le plus petit possible,
- ☐ le plus grand possible,
- ☐ le plus proche de 1 possible,
- ☐ cela n'a pas d'importance.

Exercice 2 : Arbres pour la représentation d'expressions arithmétiques (20 points)

On s'intéresse dans cet exercice à des arbres représentants des expressions arithmétiques. Les nœuds internes de l'arbre peuvent contenir les symboles `+`, `-`, `*` ou `/` et les feuilles contiennent soit un nombre, soit le symbole `x`. Un nœud contenant un `+` représente l'expression de la somme entre le fils gauche et le fils droit de ce nœud. Il en est de même pour les autre symboles. Nous allons étudier comment implémenter de tels arbres pour évaluer une expression en une valeur de `x` donnée, ou pour dériver une expression par rapport à `x`.

(1 pts) **2.a]** Dessinez l'arbre correspondant à l'expression $\frac{x+2}{3-x}$.

(1 pts) **2.b]** Comment s'affiche l'expression correspondant à cet arbre si on en affiche les nœuds à l'aide d'un parcours postfixe ?

On utilise la structure suivante pour représenter un nœud de l'arbre :

```
1 typedef struct t_node {
2   int val;
3   char symbol;
4   struct t_node* fg;
5   struct t_node* fd;
6 } node;
```

On choisit la convention de représenter un entier par un nœud dont le `symbol` est `'n'` et `val` vaut l'entier en question. Les autres nœuds sont représentés uniquement par `symbol` qui contient `'+'`, `'-'`, `'*'`, `'/'` ou `'x'`.

(3 pts) **2.c]** Écrivez une fonction *`void`* `affiche(`*`node*`* `A)` qui affiche l'arbre d'une expression dans l'ordre standard, avec des parenthèses. L'expression $\frac{x+2}{3-x}$ devra s'afficher `((x+2)/(3-x))`. Si vous écrivez votre fonction en C, rappelez vous que `%c` permet à `printf` d'afficher un *`char`* directement.

↷

↶

(5 pts) **2.d**] Écrivez maintenant une fonction `double eval(node* A, double v)` qui évalue l'expression de l'arbre `A` quand `x` vaut la valeur `v`.

(2 pts) **2.e**] Avant de commencer à dériver un arbre, il va être utile de pouvoir copier un arbre complet. Écrivez pour cela une fonction récursive `node* clone(node* A)` qui recopie l'arbre `A` et retourne un pointeur sur la racine de la copie. Chaque nœud du nouvel arbre doit être une copie de nœud.

(8 pts) **2.f]** Écrivez maintenant une fonction `node* deriv(node* A)` qui retourne un pointeur vers la racine d'un nouvel arbre correspondant à la dérivée de l'expression représentée par `A` par rapport à `x`.

Exercice 3 : Hauteur d'arbres AVL (8 points)

Dans cet exercice, nous cherchons à prouver que la hauteur maximale d'une arbre AVL contenant n nœuds est bien $\Theta(\log n)$.

(2 pts) **3.a]** Donnez la définition d'un arbre AVL.

(1 pts) **3.b]** Dessinez des arbres AVL de hauteur 1, 2 et 3 contenant un nombre minimal de nœud.

(3 pts) **3.c]** Expliquez comment construire, de façon récursive, un arbre AVL de hauteur h contenant le moins de nœuds possible. Déduisez-en une formule de récurrence calculant le nombre de nœuds minimum d'un arbre AVL de hauteur h.

(2 pts) **3.d]** Déduisez-en une borne sur la hauteur d'un arbre AVL contenant n nœuds.

Exercice 4 : Routeur et QOS (25 points)

Dans cet exercice, nous nous intéressons à l'implémentation d'un routeur. Il s'agit ici d'un routeur très basique, puisque tout ce qu'il sait faire c'est stocker les paquets qui arrivent, puis les ressortir dans un ordre donné. Un paquet sera représenté par la structure suivante :

```
1 typedef struct {
2   int num;
3   int port;
4   int ip;
5   char* data;
6 } packet;
```

Le champ `num` correspond à un numéro de série du paquet, ajouté automatiquement par le routeur quand le paquet arrive. On considère qu'un paquet arrivé après un autre aura toujours un numéro de série supérieur. Les champs `port` et `ip` correspondent à l'adresse IP et au port vers lequel le paquet vont être envoyé. Le champ `data` contient les données du paquet. La mémoire du routeur est composée d'un tableau `packet* tab` pouvant contenir 1024 paquets et d'un peu de place pour stocker toutes les variables dont vous aurez besoin.

(2 pts) **4.a]** Pour l'instant le routeur est un simple relais : il ne regarde pas ce qui est à l'intérieur des paquets et ce contente de stocker les paquets quand ils arrivent et de les ressortir dans l'ordre de leur arrivée. Quelle est la structure de donnée la mieux adaptée pour l'implémentation d'un tel routeur ? Quelles sont les complexités des opérations d'insertion et d'extraction dans cette structure ?

(5 pts) **4.b]** Écrivez le code des deux fonctions `void insert(packet* p)` et `packet* extract()` qui vont réciproquement ajouter un paquet dans le tableau `tab` et extraire le premier paquet arrivé encore en attente. Attention, le tableau a une capacité maximale de 1024 éléments, cela veut dire qu'il ne peut pas y avoir plus de 1024 paquets en attente au même moment, mais le routeur doit être capable de faire transiter plus de 1024 paquets l'un après l'autre quand même. Notez que vous pouvez utiliser des variables globales pour représenter par exemple le nombre de paquets en attente dans le routeur.

↷

↶

Afin de permettre de traiter en priorité certains paquets par rapport à d'autres, nous décidons d'implémenter une forme de QOS (Quality of Service). Il s'agit d'attribuer à certains ports une priorité plus élevée qu'à d'autres. Ainsi, en donnant une priorité élevée au port 80, tout le trafic HTTP sera prioritaire par rapport au trafic peer-to-peer, par exemple. Pour cela, nous ajoutons un tableau `int* qos` qui contient les priorités de chacun des ports : l'instruction `qos[80] = 20` attribue la priorité 20 au port 80.

(2 pts) **4.c**] On veut implémenter un routeur avec QOS qui peut donc mettre en attente les paquets qu'il reçoit et à tout moment extraire le paquet ayant la priorité la plus élevée. On veut aussi que deux paquets ayant la même priorité sortent dans l'ordre de leur arrivée. Quelle structure de donnée est la mieux adaptée pour implémenter un tel routeur avec QOS ? Quelles sont alors les complexités des opérations d'insertion et d'extraction (en fonction du nombre n de paquets en attente) ?

(6 pts) **4.d]** Écrivez les fonctions *void* `insert_qos(`*packet**` p)` et *packet** `extract_qos()` qui vont réciproquement ajouter un paquet dans le tableau `tab` et extraire le paquet le plus prioritaire (et le premier arrivé pour des paquets de même priorité). Encore une fois, vous pouvez utiliser des variables globales si vous en avez besoin.

(2 pts) **4.e]** Afin de simplifier la configuration du routeur, au lieu d'attribuer directement une priorité à chaque port, on veux pouvoir définir un ensemble de règles de la forme : "le port 80 est prioritaire sur le port 21", "le port 443 est prioritaire sur le port 21", "le port 22 est prioritaire sur le port 80". Chaque règle lie 2 numéros de ports en disant que l'un des deux ports doit avoir une priorité plus élevée que l'autre. Quelle méthode proposeriez vous pour attribuer automatiquement les priorités des ports en fonction de cet ensemble de règle ? Quelle est la complexité de votre méthode ?

(8 pts) **4.f]** Écrivez (en pseudo-code, en C cela serait un peu lourd...) un algorithme qui prend en argument deux tableaux de `n` entiers `sup` et `inf` (correspondants aux règles "le port `sup[i]` est prioritaire sur le port `inf[i]`") et qui initialise le tableau `qos` avec des priorités qui suivent ces règles. On donnera une priorité encore plus faible à tous les ports qui n'interviennent dans aucune règle. Vous pouvez écrire des fonctions annexes si vous avez besoin et, encore une fois, vous pouvez utiliser des variables globales (à énumérer en dehors des fonctions).

↶

Exercice 5 : Problème de sac à dos (15 points)

En informatique, un problème de sac à dos est un problème où, étant donné un ensemble de n entiers (n objets ayant des volumes différents ou non), on veut sélectionner un sous-ensemble de ces entiers dont la somme vaut exactement une valeur S cible (on veut remplir exactement un sac à dos de volume S). On se donne donc en entrée un tableau `int* tab` de n entiers, triés en ordre croissant, et un entier `int S`. On veut calculer un tableau `int* present` de n valeurs dans $\{0, 1\}$, 1 si l'on prend l'entier de `tab` correspondant et 0 sinon. À la fin, on veut :

$$\sum_{\texttt{i}=0}^{n-1} \texttt{tab}[\texttt{i}] \times \texttt{present}[\texttt{i}] = \texttt{S}.$$

(2 pts) **5.a]** Décrivez un algorithme qui essaye de résoudre ce problème (mais n'y arrive pas forcément) en choisissant toujours l'entier qui approche le plus la somme de `S`. Comment appelle-t-on un tel algorithme ? Quelle est sa complexité dans le pire cas ?

(5 pts) **5.b]** On veut modifier l'algorithme précédent pour que quand il ne trouve pas de solution au problème, il revienne en arrière et, à une étape précédente, essaye un entier qui approche un peu moins la somme totale de `S`. Écrivez un algorithme récursif qui fait cela et remplit le tableau `present` au fur et à mesure.

(1 pts) **5.c]** Quelle est la complexité dans le pire cas de l'algorithme précédent ?

(5 pts) **5.d]** On suppose maintenant que n est assez grand, mais que la somme S que l'on vise est relativement petite (mais quand même plus grande que chacun des entiers de `tab`). La méthode précédente ne tire pas profit de la petite valeur de S. Écrivez une fonction dont la complexité ne doit pas dépasser $\Theta(nS)$ qui cherche s'il existe une solution au problème de sac à dos posé (la fonction doit juste renvoyer 1 ou 0, selon qu'une solution existe ou pas).
Inspirez-vous des méthodes de programmation dynamique.

(2 pts) **5.e]** Comment peut-on modifier l'algorithme précédent pour aussi connaître la solution s'il en existe une ? On ne veut pas pour autant que la complexité spatiale de l'algorithme ne dépasse $\Theta(S)$.

Exercice 6 : Un peu de dessin pour occuper le temps (7 points)

(3 pts) **6.a]** Dessinez un automate deterministe pour la recherche du motif *abacabc*. On veut un automate qui reste dans l'état final une fois le motif rencontré.

(2 pts) **6.b]** Dessinez un automate non-deterministe reconnaissant le langage $\mathcal{L} = (a^+b|ac)^*a$.

(2 pts) **6.c]** Déterminisez l'automate précédent.

Ne pas retourner.

Il est interdit de regarder le contenu de ce document avant d'y avoir été invité.

Toute violation de cette interdiction sera sévèrement punie.

Ne pas retourner.