Université des Sciences et de la Technologie Houari Boumediène
Faculté d'Electronique et d'Informatique
Département d'Informatique
LMD Master 1ère Année IL 2011/12
Module ''Algorithmique Avancée et Complexité''
Date : 12/01/2012

Corrigé de l'examen
<Corrigé préparé par le responsable du module, Mr ISLI>

Exercice 1 (NP-complétude : 7 points=<(3+1+1)+1+1>) :

On considère le problème de décision CYCLE_HAM suivant :

- **Description :** un graphe G
- **Question :** G admet-il un cycle hamiltonien ? un cycle hamiltonien est un cycle qui part d'un sommet initial, visite chacun des autres sommets exactement une fois, avant de revenir sur le sommet initial

1. Le but de cette première question est de trouver un algorithme polynômial de validation pour le problème CYCLE_HAM ci-dessus, que vous appellerez **validation_ch**. Il vous est demandé de procéder comme suit :
   a. Donnez l'algorithme sous forme d'une fonction booléenne dont il est important que vous expliquiez les paramètres (*numérotez les lignes de votre algorithme*).
   b. Calculez le nombre d'opérations élémentaires de l'algorithme en fonction d'une taille n à préciser. Appelez ce nombre T(n).
   c. Montrez que $T(n)=O(n^k)$, pour une certaine constante k à préciser.
2. Quel le nombre total de certificats possibles pour l'algorithme de validation **validation_ch** ? Expliquez.
3. Donnez un algorithme testant si la réponse d'une instance du problème CYCLE_HAM est OUI.

Solution :

1.
   a. L'algorithme de validation est comme suit. Il est écrit sous forme d'une fonction booléenne à trois entrées n, C et ch. La paire (n,C) donne le codage de l'instance du problème :

- L'instance est un graphe G=<V,E> (V={u[1],…,u[n]} est l'ensemble des sommets de G, de cardinal n, et E l'ensemble de ses arêtes)
- C est la matrice d'adjacence de G, de taille n*n, booléenne
  - C[i][j]=1 si et seulement si i=j ou (u[i],u[j]) est arête de G

L'entrée ch est un certificat consistant en un tableau de taille n, dont les éléments sont tous différents et appartiennent à l'ensemble {1,…,n} (le certificat est un tableau de permutation de taille n). L'algorithme de validation retournera VRAI ssi pour tout i allant de 1 à n-1, (u[c[i]],u[c[i+1]]) est arête de G ; et, en même temps, (u[c[n]],u[c[1]]) est aussi arête de G (ce qui équivaut à dire que (u[c[1]],…,u[c[n]]) est cycle hamiltonien de G).

Si un entier est représenté sur p bits, la paire (n,C) peut être vue comme un mot de $\{0,1\}^*$ de longueur $p*(n^2+1)$: les p premiers bits (0 ou 1) coderont le nombre n de sommets de l'instance, les p*n suivants coderont la 1ère ligne de la matrice C, …, les p*n derniers bits coderont la toute dernière ligne de la matrice C. Mais on peut faire

mieux : voir la paire (n,C) comme un mot de $\{0,1\}^*$ de longueur p+n^2 : C étant booléenne, on peut coder chacun de ses éléments avec un unique bit, et non avec p bits.

```
Booléen validation_ch(n,C,ch)
    début
    1. Pour i=1 à n-1 faire
    2.     Si C[u[ch[i]],u[ch[i+1]]]=0 alors retourner FAUX finsi
    3. Fait
    4. Si C[u[ch[n]],u[ch[1]]]=0 alors retourner FAUX finsi
    5. Retourner VRAI
    fin
```

b. Le nombre T(n) d'opérations élémentaires de l'algorithme est clairement un polynôme de degré 1 en n : le pire cas correspond ici au cas où l'algorithme retourne VRAI

Instruction	(Majorant du) nombre d'opérations élémentaires
1	3*n
2	1*(n-1)
4	1
5	1

T(n)=4n+1

c. T(n)=4n+1=O(n) : trivial

2. Le nombre total de certificats possibles pour l'algorithme de validation est clairement le nombre de permutations de n éléments, c'est-à-dire n!
Explication : si on construit un certificat ch en choisissant d'abord ch[1] puis ch[2} … et enfin ch[n] alors il y a n façons de chosir ch[1], n-1 façons de choisir ch[2], …, 1 façon de choisir ch[n]. Il y a donc n*(n-1)*…*1=n ! façons de choisir un certificat ch.
3. Ici, nous donnons un algorithme (itératif) naïf calculant la réponse d'une instance du problème CYCLE_HAM donnée par la paire (n,C). L'algorithme parcourt les différents certificats de la façon suivante. Pour chacun des entiers i de 0 à n^n-1, il écrit i dans la base n et met le résultat dans un tableau T de taille n (T[1] et T[n] étant, respectivement, les positions de poids le plus fort et de poids le plus faible). Il incrémente chaque élément de T de 1 pour que les éléments soient tous dans l'intervalle 1..n (les indices du graphe représenté par l'instance). Il teste ensuite si le tableau T obtenu est un tableau de permutation. Si oui, il appelle l'algorithme de validation validation_ch avec les paramètres n, C et T : si validation_ch retourne VRAI, le tableau T est un certificat validant l'instance et l'algorithme retourne OUI. Si aucun i de 0 à n^n-1 ne donne lieu à un tel tableau, l'algorithme invalide l'instance en retournant NON.

```
Fonction reponse_instance(n,C)
début
Pour i=0 à n^n-1 faire
        T=base_n(i,n)
            //la fonction base_n(i,n) écrit i dans la base n et retourne le résultat
            //sous forme d'un tableau de taille n : i=T[1].n^(n-1)+…+T[n].n^0
        Pour i=1 à n faire T[i]=T[i]+1 fait
            //on incrémente de 1 chaque élément du tableau pour avoir tous
            //les éléments dans l'intervalle 1..n
        Si tableau_permutation(T) et validation_ch(n,C,T) alors retourner OUI finsi
fait
retourner NON
fin
```

Exercice 2 <fusion : 4 points>: Fusion de deux listes chaînées triées : donnez un algorithme polynômial de fusion de deux listes chaînées triées, le résultat de la fusion devant être une liste chainée triée. Expliquez la polynômialité de l'algorithme.

Solution :

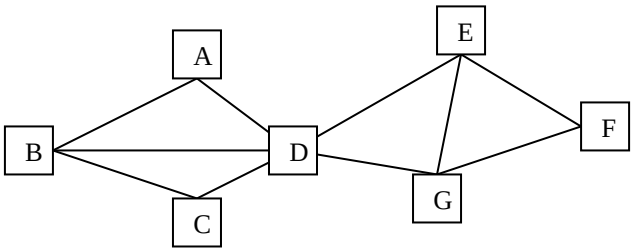
La fonction fusion(tete1,tete2) ci-dessous fusionne deux listes chaînées triées, de têtes tete1 et tete2 respectivement. Le résultat de la fusion est une liste chaînée triée de tête tete. La fonction n'a besoin d'aucune création dynamique d'élément de liste : elle commence par « zigzaguer » entre les deux listes chaînées jusqu'à atteindre la fin d'une des deux ; puis elle continue le parcours de celle dont elle n'a pas encore atteint la fin. Elle retourne le pointeur tete.

```
fusion(tete1,tete2)              //les deux listes sont supposées triées par ordre croissant …
début  si tete1=NIL alors retourner tete2 finsi
       si tete2=NIL alors retourner tete1 finsi
              //si on arrive à ce point, aucune des deux listes n'est vide
       si tete1.clef≤tete2.clef alors
              tete=tete1
              t1=tete1.suiv
              t2=tete2
       sinon
              tete=tete2
              t1=tete1
              t2=tete2.suiv
       finsi
       t=tete
       tant que t1≠NIL et t2≠NIL faire
              si t1.clef≤t2.clef       alors
                     t.suiv=t1 ; t=t1
                     t1=t1.suiv      //on avance dans la liste 1 …
              sinon
                     t.suiv=t2 ; t=t2
                     t2=t2.suiv      //on avance dans la liste 2 …
              finsi
       fait
              //Au moins une des deux listes a été parcourue jusqu'à la fin …
              //Si les deux listes ont été parcoures jusqu'à la fin, on affecte NIL à t.suiv …
              //Sinon, on « colle » le reste de la liste dont le parcours n'a pas encore pris fin …
       si t1≠NIL alors t.suiv=t1
       sinon
              si t2≠NIL alors t.suiv=t2 sinon t.suiv=NIL finsi
       finsi
              //La fusion a pris fin …
              //La fonction retourne le pointeur sur l'élément de tête de la liste résultante
       retourner tete
fin
```

Complexité de l'algorithme :

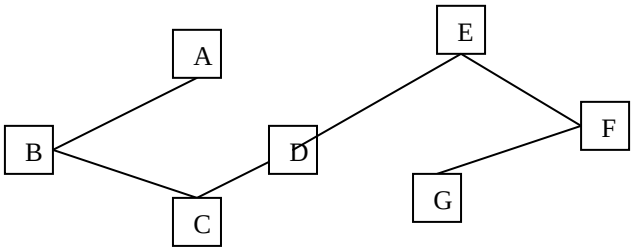
L'algorithme fait un parcours de la liste L1 et un parcours de la liste L2. Si n est le maximum des tailles de L1 et L2, le nombre f(n) d'opérations élémentaires de l'algorithme peut être majoré par un polynôme de degré 1 en n. L'algorithme de fusion est donc linéaire : f(n)=O(n).

Exercice 3 (parcours en profondeur d'abord : 4 points=2+2) : Donnez une forêt de recouvrement issue d'un parcours en profondeur d'abord du graphe suivant, de même que l'ordre dans lequel le parcours considéré a visité les sommets du graphe :



Solution 1 :

Forêt (arbre car graphe connexe) de recouvrement :

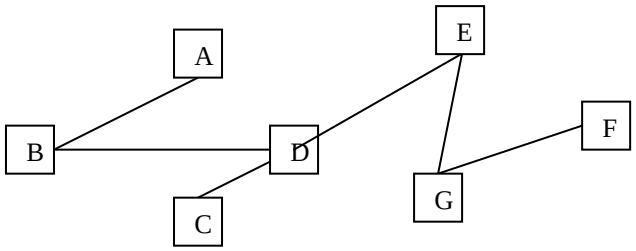


Ordre de visite des sommets : A, B, C, D, E, F, G (A le premier, G le dernier)

Explication : Ici, le parcours commence au niveau du sommet A et descend jusqu'au sommet G via, respectivement, les arêtes (A,B), (B,C), (C,D), (D,E), (E,F) et (F,G). Arrivé sur le sommet G, le parcours trouve que tous les sommets voisins de G ont déjà été visités, donc la descente en profondeur ne peut plus continuer, et les arêtes (G,D) et (G,E) ne feront pas partie de l'arbre de recouvrement. Le parcours fait donc un retour arrière (backtracks) et remonte sur le sommet le plus récemment visité qui est ici F : F n'a plus de voisin non encore visité. Le parcours remonte sur E, et le scénario se répète : E n'a plus de voisin non encore visité ((l'arête (E,G) ne sera pas exploré car G déjà visité). Retour arrière donc, sur le sommet D : les arêtes (D,A) et (D,B) ne feront pas partie de l'arbre de recouvrement, les sommets A et B ayant déjà été visités. Retour sur C puis sur B : l'arête (B,A) ne fera pas partie de l'arbre de recouvrement, le sommet D ayant déjà été visité. Retour arrière sur A : le sommet voisin D ayant déjà été visité, l'arête (A,D) ne fera pas partie de l'arbre de recouvrement. Il n'est plus possible de faire retour arrière : le parcours prend fin.

Solution 2 :

Forêt (arbre) de recouvrement :



Ordre de visite des sommets : A, B, D, C, E, G, F ou A, B, D, E, G, F, C

D'autres solutions existent !

Exercice 4 (dérécursivation : 5 points=2+3) : Dérécursivation du parcours en profondeur d'abord d'un arbre binaire : le cas préfixe.

1. Donnez un algorithme récursif de parcours préfixe d'un arbre binaire.
2. Dérécursivation du parcours : donnez un algorithme itératif équivalent à l'algorithme récursif de la question 1.

Solution :

1. Algorithme récursif :

```
Préfixe(A)
début
    Si A≠NIL alors
            Afficher(A.clef);
            Préfixe(A.sa-gauche);
            Préfixe(A.sa-droit);
    finsi
fin
```

2. Algorithme itératif équivalent :

```
Préfixe_it(A)
début
    EMPILER(P,A)
    Tant que (non PILE-VIDE(P)) faire
            B=DEPILER(P)
            Si B≠NIL alors
                    Afficher(B.clef);
                    EMPILER(P,B.sa-droit);
                    EMPILER(P,B.sa-gauche);
                            //on empile le sous-arbre droit d'abord puis le sous-arbre gauche !
            finsi
    fait
fin
```