esprit Se former autrement HONORIS UNITED UNIVERSITIES	**Examen** Semestre : 1 Session : Principale
Module : RO-Complexité Enseignant(s): Équipe RO-Complexité	
Documents autorisés : NON ; Nombre de pages : 2 ; Internet autorisée : NON Calculatrice autorisé : NON	
Date : 21/01/2022 à 11H : 00	Durée : 1H30

Exercice 1 (6 points):

Pour chaque algorithme proposé ci-dessous, calculer l'ordre de complexité en nombre d'opération : Afficher(∗).

Algorithme 1
For (i = n; i>1; i=i/2)
For (j = 0; j<n; j++)
Afficher(∗)
EndFor
EndFor
	Algorithme 2
	For (i=1; i<n ; i++)
	For (j=1; j< n; j=j*2)
	Afficher(∗)
	EndFor
	EndFor

Algorithme 3
For (i=1; i<n ; i++)
For (j=1; j< i; j++)
Afficher(∗)
EndFor
EndFor
	Algorithme 4
	For (i=1; i<n ; i=i*2)
	For (j=1; j< n; j++)
	Afficher(∗)
	EndFor
	EndFor

Algorithme 5
For (i=1; i<n ; i++)
For (j=1; j< n; j++)
For(k=1; k<j ; k++)
Afficher(∗)
EndFor
EndFor
EndFor
	Algorithme 6
	For (i=1; i<n ; i++)
	For (j=1; j< i; j++)
	For(k=1; k<j ; k++)
	Afficher(∗)
	EndFor
	EndFor
	EndFor

Exercice 2 : (6 points)

Le principe de l'algorithme de tri à bulles est de comparer deux à deux les éléments consécutifs d'un tableau et d'effecteur une permutation si les deux éléments ne sont

pas dans le bon ordre. On continue ce processus jusqu'à ce qu'il n'y ait plus de permutation. Le pseudo-code de l'algorithme est comme suit :

```
void Tri_Bulles(int T[], int n)
{
  int i, j, tmp;
  for (i=0 ; i < n-1; i++)
  {
    for (j=0 ; j < n-i-1; j++)
    {
      if (T[j] > T[j+1])
      {
        tmp = T[j];
        T[j] = T[j+1];
        T[j+1] = tmp;
      }
    }
  }
}
```

1. Calculer la complexité au meilleur et au pire des cas pour :
   a) le nombre de comparaisons entre éléments du tableau
   b) le nombre d'échanges d'éléments du tableau
2. Comparer les deux complexités de a) et b) dans le meilleur des cas.
3. Comparer les deux complexité de a) et b) dans le pire des cas.

Exercice 3: (8 points)

Un entier est divisible par 11 si et seulement si la somme alternée de ses chiffres est nulle.

Exemple : 121 est divisible par 11 car 1-2+1=0.

On considère alors la fonction récursive « **somme_alt**» qui permet de calculer la somme alternée des chiffres d'un entier donné sous la forme d'un tableau T d'entier contenant ses chiffres.

```
int somme_alt(int T[], int n){

 if(n==0){
    return 0;
 }
 if(n%2==0)
    return somme_alt(T, n-1)-T[n-1];
 else
    return somme_alt(T, n-1)+T[n-1];


}
```

Le paramètre n représente la taille du tableau.

1. La récursivité est-elle simple, multiple ou imbriquée ?
2. Cette récursivité est-elle terminale ?
3. Dérécursiver la fonction « **somme_alt** »

4. On s'intéresse à déterminer la complexité de la fonction « **somme_alt** » en fonction des opérations arithmétiques et logiques.

a. Donner l'équation récurrente de complexité relative à « **somme_alt »**

a. Déterminer alors la classe de complexité de « **somme_alt** »

Correction

Exercice 1 :

1) 1ere boucle « for » comporte log2(n) itérations car on divise successivement n par 2 jusqu'à atteindre 1
   2eme boucle « for » comporte n itérations
   Comme les deux boucles sont imbriquées alors C(n)=n*log2(n)=O (n log(n))
2) 1ere boucle for : n-1 itérations
   2eme boucle for : log2(n) itérations
   Comme les deux boucles sont imbriquées alors C(n)= (n-1)*log2(n)= O (n log(n))
3) 1ere boucle for : n-1 itérations
   2eme boucle for : i-1 itérations avec i-1<n
   Comme les deux boucles sont imbriquées alors C(n) <= (n-1)*n
   D'où C(n)= O (n²)
4) 1ere boucle for : log2(n) itérations
   2eme boucle for : n-1 itérations
   Comme les deux boucles sont imbriquées alors C(n)= (n-1)*log2(n)= O (n log(n))
5) 1ere boucle for : n-1 itérations
   2eme boucle for : n-1 itérations
   3eme boucle for : j-1 itérations avec j-1<n
   Comme les trois boucles sont imbriquées alors C(n) <= (n-1)²*n
   D'où C(n)= O (n^3)
6) 1ere boucle for : n-1 itérations
   2eme boucle for : i-1 itérations avec i-1<n
   3eme boucle for : j-1 itérations avec j-1<i<n
   Comme les trois boucles sont imbriquées alors C(n) <= (n-1)*n²
   D'où C(n)= O (n^3)

Exercice 2 :

1) a) Dans tous les cas la comparaison var être faite, donc la complexité au pire et au meilleur sont égales au produit du nombre d'itérations des boucles « for »
   1ere boucle for : n-1 itérations
   2eme boucle for : n-i-1 itérations avec n-i-1<n
   Comme les deux boucles sont imbriquées alors C(n) <= (n-1)*n
   Ainsi la complexité au pire et au meilleur en nombre de comparaisons sont égales à O (n²)

b) On fait un échange entre T[j] et T [j+1] que lorsque T[j]>T [j+1]
   Donc si le tableau est trié dans l'ordre croissant on ne va pas faire d'échanges
   La complexité au meilleur est nulle (C(n)=0).

Si le tableau est trié dans l'ordre décroissant alors la condition T[j]>T [j+1] est toujours vrai et dans ce cas la complexité d'échanges est égale à celle des comparaisons à O près

D'où la complexité au pire en nombre d'échanges est O (n^2)

2) Au meilleur des cas l'algorithme fait 0 échange et O (n^2) comparaisons, donc l'algorithme est plus efficace en échange
3) Au pire des cas la complexité en nombre de comparaisons et en nombre d'échanges sont égales à O près.

Exercice 3 :

1) « somme_alt » appelle récursivement elle-même une seule fois, donc on a une récursivité simple.

2) Cette récursivité est non terminale car l'appel récursif est poursuivi par une opération d'addition.

3) Comme la récursivité est non terminale, on doit utiliser une pile pour sauvegarder le traitement à faire une fois qu'on atteint la condition d'arrêt.

Dans la pile on va sauvegarder ces paramètres :

si n est pair **si n est impair**

T[0]
-T[1]
T[2]
.
.
.
-T[n-2]
T[n-1]

T[0]
-T[1]
T[2]
.
.
.

T[n-2]
-T[n-1]

Ainsi l'algorithme dérécursivé est donné par :

```
int somme_alt(int T[],int n){
    stack<int> P;
    P.init() ;
    While(n>0){
      If(n%2==0)
        P.push((-1)*T[n-1])
      Else
        P.push(T[n-1]) ;
      n-- ;
    }
    Int s=0 ;
    While(not P.empty()){
      s=s+P.pop() ;
    }
    Return s ;
}
```

4) a) 2 comparaisons + 1 division + 3 addition = 6 opérations, donc

$$C(n) = \{1\,6 \; + C(n-1) \qquad si\, n = 0\, sinon$$

b) C(n) est une suite arithmétique de 1er terme 1 et de raison 6
Donc C(n)=1+6*n=O(n)