**Exercice 1 : (6points)**

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

Algorithme 1 :

Le premier algorithme effectue une boucle externe de n à 1, avec une division par 2 à chaque itération. À l'intérieur de cette boucle, il y a une boucle interne de 0 à n. La complexité totale de l'algorithme est donc de **O(n log n)**.

Algorithme 2 :

Le deuxième algorithme effectue une boucle externe de 1 à n et une boucle interne de 1 à n avec une multiplication par 2 à chaque itération. La complexité totale de l'algorithme est donc de **O(n log n)**.

Algorithme 3 :

Le troisième algorithme effectue une boucle externe de 1 à n et une boucle interne de 1 à i. À chaque itération de la boucle externe, la boucle interne s'exécute i fois. La complexité totale de l'algorithme est donc de O(1 + 2 + 3 + ... + n-1), ce qui équivaut à **O(n^2)**.

Algorithme 4 :

Le quatrième algorithme effectue une boucle externe de 1 à n avec une multiplication par 2 à chaque itération, et une boucle interne de 1 à n. La complexité totale de l'algorithme est donc de **O(n log n)**.

Algorithme 5 :

Le cinquième algorithme effectue une boucle externe de 1 à n, une boucle interne de 1 à n, et une boucle interne de 1 à j. À chaque itération de la boucle externe, les deux boucles internes s'exécutent n fois chacune. La complexité totale de l'algorithme est donc de **O(n^3)**.

Algorithme 6 :

Calculer cette somme peut être assez complexe et dépend du domaine spécifique des indices. Pour une analyse simplifiée, nous pouvons supposer que i, j et k sont égaux à n. Dans ce cas, la complexité de l'algorithme 6 serait de **O(n^3)**..

**Exercice 2 : (6 points)**

Le principe de l’algorithme de trie à bulles est de comparer deux à deux les éléments consécutifs d’un tableau et d’effectuer 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 :
   1. Le nombre de comparaisons entre éléments du tableau
   2. 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 complexités de a. et b. dans le pire des cas

**Pour l'algorithme de tri à bulles présenté dans l'exercice 2, nous allons calculer la complexité au meilleur et au pire des cas pour le nombre de comparaisons entre les éléments du tableau et le nombre d'échanges d'éléments du tableau. Ensuite, nous comparerons les deux complexités dans le meilleur des cas et dans le pire des cas.**

**Complexité au meilleur des cas :**

**Dans le meilleur des cas, cela signifie que le tableau est déjà trié. Dans ce cas, aucune permutation n'est nécessaire, et l'algorithme effectuera uniquement les comparaisons entre les éléments du tableau.**

**Le nombre de comparaisons entre les éléments est donné par la somme de 1 à n-1 :**

**C(n) = 1 + 2 + 3 + ... + (n-1) = (n-1) \* n / 2**

**Le nombre d'échanges d'éléments est égal à zéro puisqu'aucun échange n'est nécessaire.**

**Complexité au pire des cas :**

**Dans le pire des cas, cela signifie que le tableau est trié dans l'ordre inverse, ce qui nécessite de permuter tous les éléments jusqu'à obtenir un tableau trié.**

**Le nombre de comparaisons entre les éléments est toujours la même somme de 1 à n-1 :**

**C(n) = 1 + 2 + 3 + ... + (n-1) = (n-1) \* n / 2**

**Le nombre d'échanges d'éléments correspond également à la somme de 1 à n-1 :**

**E(n) = 1 + 2 + 3 + ... + (n-1) = (n-1) \* n / 2**

Comparaison des complexités dans le meilleur des cas et le pire des cas :

Dans le meilleur des cas et le pire des cas, la complexité pour le nombre de comparaisons entre les éléments et le nombre d'échanges d'éléments est la même, ce qui est cohérent avec le comportement de l'algorithme de tri à bulles.

Ainsi, dans les deux cas, la complexité pour le nombre de comparaisons entre les éléments est de O(n^2) et la complexité pour le nombre d'échanges d'éléments est également de O(n^2).

**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 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
   1. Donner l’équation récurrente de complexité relative à “**somme\_alt**”
   2. Déterminer alors la classe de complexité de “**somme\_alt**”

Exercice 3:

La récursivité est-elle simple, multiple ou imbriquée ?

La récursivité dans la fonction "somme\_alt" est simple, car la fonction s'appelle elle-même une seule fois dans chaque branche conditionnelle.

Cette récursivité est-elle terminale ?

La récursivité dans la fonction "somme\_alt" n'est pas terminale, car les appels récursifs sont suivis d'opérations arithmétiques ou logiques avant de retourner la valeur.

Dérécursiver la fonction "somme\_alt":

Pour dérécursiver la fonction "somme\_alt", nous pouvons utiliser une approche itérative pour calculer la somme alternée. Voici une version dérécursivée de la fonction :

int somme\_alt(int T[], int n) {

int somme = 0;

for (int i = 0; i < n; i++) {

if (i % 2 == 0) {

somme -= T[i];

} else {

somme += T[i];

}

}

return somme;

}

Complexité de la fonction "somme\_alt":

a. Équation récurrente de complexité relative à "somme\_alt":

La fonction "somme\_alt" effectue une boucle itérative de 0 à n, où n est la taille du tableau. À chaque itération, une opération arithmétique est effectuée. Par conséquent, l'équation récurrente de complexité relative à "somme\_alt" est :

T(n) = T(n-1) + 1

b. Classe de complexité de "somme\_alt":

En résolvant l'équation récurrente, nous obtenons :

T(n) = T(n-1) + 1

= T(n-2) + 1 + 1

= T(n-3) + 1 + 1 + 1

= ...

= T(0) + n

Donc, la complexité de la fonction "somme\_alt" est linéaire, c'est-à-dire O(n), où n est la taille du tableau.