**Exercice 1 (6 points)**

On considére les fonctions A, B et C définies ce-dessous, déterminer la complexité en nombre d’”opération” **en justifiant votre réponse**

A (int n){

while (1<n){

operation;

n=n/2;

}

}

B (int n){

int i;

for (i=0; i<n\*n; i++)

operation;

}

C (int n){

int i;

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

A(n);

}

B(n);

}

Dans cet exercice, nous devons déterminer la complexité en nombre d'opérations pour les fonctions A, B et C. Voici les résultats :

Fonction A :

La fonction A utilise une boucle while qui se répète tant que la condition n > 1 est vraie. À chaque itération de la boucle, une opération est effectuée, suivie d'une division par 2 de la variable n.

La complexité de la fonction A dépend du nombre d'itérations effectuées dans la boucle while. Dans le pire des cas, la boucle se répète jusqu'à ce que n atteigne 1. Cela se produit lorsque n est une puissance de 2. Par conséquent, le nombre d'itérations est log₂(n).

La complexité totale de la fonction A est donc de O(log(n)) en nombre d'opérations.

Fonction B :

La fonction B utilise une boucle for qui s'exécute de 0 à n². À chaque itération de la boucle, une opération est effectuée.

Le nombre total d'opérations dépend de la valeur de n. La boucle s'exécute n² fois, ce qui donne une complexité de O(n²) en nombre d'opérations.

Fonction C :

La fonction C appelle la fonction A(n) dans une boucle for qui s'exécute n fois. Ensuite, elle appelle la fonction B(n).

La complexité de la fonction A(n) est O(log(n)) en nombre d'opérations, comme nous l'avons déterminé précédemment.

La complexité de la fonction B(n) est O(n²) en nombre d'opérations, comme nous l'avons également déterminé précédemment.

La boucle for qui appelle la fonction A(n) s'exécute n fois, et la fonction B(n) est appelée une seule fois.

La complexité totale de la fonction C est donc de O(n log(n) + n²) en nombre d'opérations.

**Exercice 2 : (6 points)**

On cherche la somme d’un tableau T de n éléments entiers

1. 1. Ecrivez un algorithme de type diviser-pour-régner qui résout ce problème
2. 2. Analysez sa complexité

* Donner l’équation récurrente estimant la complexité
* Déduire l’ordre de complexité

1. 3. Comparez la complexité de l’algorithme

Exercice 2:

Algorithme de type "diviser-pour-régner" pour la somme d'un tableau T de n éléments entiers :

Voici un algorithme de type "diviser-pour-régner" pour résoudre le problème de la somme d'un tableau T :

arduino

Copy code

int somme\_tableau(int T[], int debut, int fin) {

if (debut == fin) {

return T[debut];

} else {

int milieu = (debut + fin) / 2;

int somme\_gauche = somme\_tableau(T, debut, milieu);

int somme\_droite = somme\_tableau(T, milieu + 1, fin);

return somme\_gauche + somme\_droite;

}

}

Analyse de la complexité :

L'algorithme utilise une approche de diviser-pour-régner, où le tableau est divisé en deux parties de taille équivalente jusqu'à atteindre des sous-tableaux de taille 1. Ensuite, les sommes des sous-tableaux sont combinées pour obtenir la somme totale du tableau.

Le cas de base se produit lorsque le sous-tableau ne contient qu'un seul élément, ce qui se fait en temps constant O(1).

Dans le cas général, l'algorithme divise le tableau en deux parties et effectue deux appels récursifs sur les sous-tableaux. Par conséquent, le temps de calcul est réparti équitablement entre les deux sous-tableaux.

Équation récurrente de la complexité :

Soit T(n) la complexité de l'algorithme pour un tableau de taille n.

T(n) = 2T(n/2) + O(1)

La récurrence représente la division du tableau en deux parties de taille équivalente et les appels récursifs sur les sous-tableaux.

Déduire l'ordre de complexité :

En résolvant l'équation récurrente, nous obtenons que la complexité de l'algorithme est de O(n), c'est-à-dire linéaire en fonction de la taille du tableau.

Comparaison de la complexité de l'algorithme :

L'algorithme de type "diviser-pour-régner" pour la somme d'un tableau a une complexité de O(n), ce qui signifie qu'il est linéaire en fonction de la taille du tableau. En comparaison, la complexité de l'algorithme de tri à bulles (sujet A, exercice 2) est de O(n^2), ce qui est quadratique.

Ainsi, l'algorithme de type "diviser-pour-régner" pour la somme du tableau est plus efficace en termes de complexité que l'algorithme de tri à bulles.

**Exercice 3: (8 points)**

Soit la fonction récursive “**Traiter**” :

int Traiter (int n)

{

if (n==0) return 1;

if (n==1) return 0;

else

return Traiter (n-1) + 2\*Traiter(n-2);

}

1.Donner la trace d’exécution de la fonction “**Traiter**” lorsqu’on lui fournit comme entrée n=4

2.Quel est le type de cette récursivité ?

3.Donner l’équation de récurrence de la fonction “**Traiter**”

4.Quel est le polynôme caractéristique de cette fonction ?

5.Calculer la complexité de “**Traiter**” tout en détaillant les étapes de calcul

6.Déduire la classe de complexité de “**Traiter**”

Exercice 3:

Trace d'exécution de la fonction "Traiter" avec n=4 :

Voici la trace d'exécution de la fonction "Traiter" lorsque n=4 :

Traiter(4)

= Traiter(3) + 2 \* Traiter(2)

= (Traiter(2) + 2 \* Traiter(1)) + 2 \* (Traiter(1) + 2 \* Traiter(0))

= ((Traiter(1) + 2 \* Traiter(0)) + 2 \* 0) + 2 \* (0 + 2 \* 1)

= (0 + 2 \* 1) + 2 \* (0 + 2 \* 1)

= 2 + 2 \* 2

= 2 + 4

= 6

Type de récursivité :

La fonction "Traiter" utilise une récursivité multiple, car elle appelle à la fois Traiter(n-1) et Traiter(n-2) dans la branche else.

Équation de récurrence de la fonction "Traiter" :

L'équation de récurrence pour la fonction "Traiter" est donnée par :

T(n) = T(n-1) + 2 \* T(n-2)

Polynôme caractéristique de la fonction "Traiter" :

Pour déterminer le polynôme caractéristique, nous remplaçons T(n) par x^n dans l'équation de récurrence et résolvons l'équation pour x.

x^n = x^(n-1) + 2 \* x^(n-2)

En simplifiant par x^(n-2), nous obtenons :

x^2 = x + 2

Le polynôme caractéristique est donc x^2 - x - 2 = 0.

Calcul de la complexité de la fonction "Traiter" :

La complexité de la fonction "Traiter" dépend du nombre d'appels récursifs effectués et du temps de calcul pour chaque appel.

En analysant la fonction, nous pouvons voir que pour chaque valeur de n, deux appels récursifs sont effectués sur les valeurs n-1 et n-2. Ainsi, le nombre total d'appels récursifs est égal au nombre de termes dans la suite de Fibonacci jusqu'à n.

La complexité de la fonction "Traiter" est donc exponentielle, et elle peut être approximée par O(2^n) en termes de complexité en nombre d'opérations.

Classe de complexité de la fonction "Traiter" :

Comme mentionné précédemment, la complexité de la fonction "Traiter" est exponentielle, O(2^n). Elle augmente rapidement avec la valeur de n. Par conséquent, la classe de complexité de la fonction "Traiter" est exponentielle.