**Exercice 1 (6 points):**

Pour chaque algorithme proposé ci-dessous, calculer l’ordre de complexité.

**Algo1**

Pour i=1 à n faire Pour j=1 à n faire

s=j+i

Fin Pour Fin Pour

**Algo2**

Pour i=1 à n-1 faire Pour j=1 à i faire

k=1

Tant que k<m faire

S=i+j

k=k+1

Fin Tant que

Fin Pour Fin Pour

**Algo3**

Pour i=5 à n-5 faire Pour j=i-5 à i+5 faire

S=S+i

Fin Pour Fin Pour

**Algo4**

i=n

Tant que i>1 faire

Ecrire (“ i= “, i)

i=i/2

Fin Tant que

**Algo5**

Pour i=1 à n faire Pour j=1 à faire

y(i)=y(i)+A(i,j)\*x(j)

Fin Pour Fin Pour

**Algo6**

Pour i=1 à n faire Pour j=i à min(n,i+1) faire

y(i) = y(i) + A(i,j)\*x(j)

Fin Pour Fin Pour

Algo1:

L'algorithme a deux boucles imbriquées. La première boucle s'exécute n fois et la deuxième boucle s'exécute également n fois pour chaque itération de la première boucle. Par conséquent, le nombre total d'opérations est de l'ordre de n \* n, soit O(n^2).

Algo2:

L'algorithme a une boucle externe qui s'exécute n-1 fois et une boucle interne qui s'exécute i fois pour chaque itération de la boucle externe. Le nombre total d'opérations est la somme des itérations des boucles pour toutes les valeurs de i de 1 à n-1. En simplifiant, cela donne (n-1) \* n / 2. Donc l'ordre de complexité est O(n^2).

Algo3:

L'algorithme a une boucle externe qui s'exécute n-9 fois et une boucle interne qui s'exécute 11 fois pour chaque itération de la boucle externe. Le nombre total d'opérations est la somme des itérations des boucles pour toutes les valeurs de i de 5 à n-5. En simplifiant, cela donne (n-9-5+1) \* 11, soit (n-13) \* 11. Donc l'ordre de complexité est O(n).

Algo4:

L'algorithme a une boucle tant que qui s'exécute tant que la condition i>1 est vérifiée. À chaque itération, i est divisé par 2. Le nombre total d'opérations dépend du nombre d'itérations nécessaires pour que i atteigne la valeur 1. Si nous considérons que i est initialement égal à n, alors le nombre d'opérations est approximativement log2(n). Donc l'ordre de complexité est O(log n).

Algo5:

L'algorithme a une boucle externe qui s'exécute n fois et une boucle interne qui s'exécute indépendamment de n. Le nombre total d'opérations dépend de la taille du tableau x(j) et du nombre d'opérations effectuées à chaque itération de la boucle interne. Sans plus d'informations sur les dimensions des tableaux et les opérations effectuées, il n'est pas possible de déterminer l'ordre de complexité avec précision.

Algo6:

L'algorithme a une boucle externe qui s'exécute n fois et une boucle interne qui s'exécute de i à min(n, i+1). Le nombre total d'opérations dépend de la valeur de i à chaque itération de la boucle externe. La boucle interne s'exécute au maximum 2 fois pour chaque itération de la boucle externe. Par conséquent, le nombre total d'opérations est de l'ordre de n \* 2, soit O(n).

En résumé:

Algo1: O(n^2)

Algo2: O(n^2)

Algo3: O(n)

Algo4: O(log n)

Algo5: Dépend des dimensions des tableaux et des opérations effectuées.

Algo6: O(n)

**Exercice 2: (7 points)**

Soit la fonction récursive “**FCT**”

Fonction FCT(T: tab, n,m: entier): booléen

Début

Si n>0 Alots

Si(m=T[n]) alors

FCT <- vrai

FinSi

FCT <- FCT(T,n-1,m)

Sinon

FCT <- faux

FinSi

Fin

1.Que faire la fonction “**FCT**”? sachant que **T** est un tableau de taille **n** et que la première case dans le tableau est d’indice 1 et la dernière case est d’indice n

2.Quel est le type de récursivité de la fonction “**FCT**”?

3.On cherche à calculer la complexité en nombre de comparaison. Donner l’équation récurrente de la fonction “**FCT**”?

4.Détailler le calcul de complexité en résolvant l’équation de récurrence obtenue dans la question 3

5.Déduire la classe de complexité de la fonction “**FCT**”

La fonction "FCT" recherche si la valeur m est présente dans le tableau T. Elle parcourt le tableau de la dernière case (indice n) à la première case (indice 1) et vérifie si la valeur m correspond à celle de l'élément en cours.

Le type de récursivité de la fonction "FCT" est récursivité terminale, car toutes les appels récursifs sont suivis d'un retour de valeur sans autre opération.

L'équation récurrente de la fonction "FCT" pour la complexité en nombre de comparaisons est la suivante :

C(n) = C(n-1) + 1, si n > 0

C(n) = 0, si n = 0

Pour résoudre cette équation récurrente, nous pouvons la développer en remplaçant récursivement C(n-1) par C(n-2), C(n-2) par C(n-3), et ainsi de suite, jusqu'à atteindre le cas de base C(0) = 0.

C(n) = C(n-1) + 1

= (C(n-2) + 1) + 1

= C(n-2) + 2

= C(n-3) + 3

= ...

= C(0) + n

Puisque C(0) = 0, nous avons finalement C(n) = n.

La classe de complexité de la fonction "FCT" est linéaire, car le nombre de comparaisons est proportionnel à la taille du tableau n. Donc la complexité est O(n).

**Exercice 3: (7 points)**

On considère A[] un tableau d’entiers, on veut chercher la différence A[j] - A[i] maximale entre deux éléments A[i] et A[j] tel que A[i] < A[j] et i<j

Exemples :

Input= A[1,4,9,5,3,7], Output: 8

Input= A[9,8,1,6,3,2], Output: 5

Input= A[9,8,6,3,2] = [], Output: -1 (tableau décroissant)

Pour résoudre ce problème, on considère l’algorithme récursif “**maxDifference**” qui suit le principe diviser pour régner, son code est le suivant :

int maxDifference(int A[], int l, int r)

{

if (1>=r) return -1;

int maxDiff=-1;

int mid=l+(r-1)/2;

int leftMaxDiff=maxDifference(A,l,mid);

int rightMaxDiff=maxDifference(A,mid+1,r);

int minLeft=findMin(A,l,mid);

int maxRight=findMax(A,mid+1,r);

maxDiff=max(max(leftMaxDiff,rightMaxDiff),maxRight-minLeft);

return maxDiff;

}

1.On suppose que les fonctions “findmin et “findmax” sont de complexité au pire O(n), donner alors l’équation récurrente de la fonction “**maxDifference**”

2.Montrer que la complexité de cet algorithme est O(nLog(n))

3.Cet algorithme est-t-il considéré comme rapide ? Justifier votre réponse

4.Soient “**maxDifference2**” et “**maxDifference3**” deux autres algorithmes qui permettent de résoudre le même problème que “**maxDifference**” et ils ont comme complexité respectivement O(n²) et O(n3). Comparer les trois algorithmes en termes de performance

L'équation récurrente de la fonction "maxDifference" est la suivante :

T(n) = 2 \* T(n/2) + O(n)

Cela est dû au fait que l'algorithme divise le tableau en deux parties égales à chaque étape récursive, ce qui contribue à une complexité de O(n/2) = O(n) pour les fonctions "findMin" et "findMax". De plus, il effectue deux appels récursifs sur des sous-tableaux de taille n/2.

Pour montrer que la complexité de cet algorithme est O(n log(n)), nous pouvons utiliser la méthode du théorème maître. Dans l'équation récurrente, a = 2 (le nombre d'appels récursifs), b = 2 (la taille relative des sous-problèmes), et f(n) = O(n) (la complexité des opérations effectuées en dehors des appels récursifs).

En utilisant le théorème maître, nous avons:

log(b,a) = log(2,2) = 1

Puisque f(n) = O(n) et log(b,a) = 1, la complexité de l'algorithme est O(n log(n)).

Oui, cet algorithme est considéré comme rapide. Une complexité de O(n log(n)) est meilleure que O(n^2) et O(n^3) pour des entrées de taille importante. Cela signifie que l'algorithme a une meilleure performance en termes de temps d'exécution par rapport à ces deux autres algorithmes.

En termes de performance :

L'algorithme "maxDifference" avec une complexité de O(n log(n)) est le plus efficace parmi les trois, car il divise le problème en sous-problèmes de taille réduite et combine les résultats de manière optimale.

L'algorithme "maxDifference2" avec une complexité de O(n^2) est moins performant, car il effectue des comparaisons répétées pour chaque paire d'éléments du tableau.

L'algorithme "maxDifference3" avec une complexité de O(n^3) est le moins performant, car il implique une boucle à trois niveaux qui examine toutes les combinaisons possibles d'éléments du tableau.

En résumé, l'algorithme "maxDifference" est le plus rapide parmi les trois proposés pour résoudre le problème de recherche de la différence maximale entre deux éléments du tableau qui satisfont certaines conditions.