Correction de l'Examen d'Algorithmique Avancée et Complexité

7-01-2025

Exercice 1 : Notation asymptotique et classification de fonctions (6 points)

Analyse des fonctions (5 points)

Fonction	Terme dominant	Notation asymptotique $O(\cdot)$	Classe de complexité
$f_1(n) = 3n + 5$	$3n$	$O(n)$	Linéaire
$f_2(n) = 4n^2 + 2n + 10$	$4n^2$	$O(n^2)$	Quadratique
$f_3(n) = n \log n + 50$	$n \log n$	$O(n \log n)$	Quasi-linéaire
$f_4(n) = 2^n + n^5$	2^n	$O(2^n)$	Exponentielle
$f_5(n) = \log_2 n + 5n^3$	$5n^3$	$O(n^3)$	Cubique
$f_6(n) = \log_2 n + 2$	$\log_2 n$	$O(\log n)$	Logarithmique

Classement des fonctions selon la complexité asymptotique (1 point)

$f_6(n) = O(\log n) < f_1(n) = O(n) < f_3(n) = O(n \log n) < f_5(n) = O(n^3) < f_2(n) = O(n^2) < f_4(n) = O(2^n)$

—

Exercice 2 : Tri rapide (QuickSort) (6 points)

Question 1 : Principe du tri rapide (1.5 point)

Le tri rapide (QuickSort) est une méthode de tri basée sur le paradigme *diviser pour régner*. L'algorithme fonctionne comme suit :

- Il sélectionne un pivot parmi les éléments de la liste.
- Il partitionne la liste en deux sous-listes : les éléments inférieurs au pivot et les éléments supérieurs ou égaux au pivot.
- Il applique récursivement le tri rapide sur les deux sous-listes.
- Lorsque les sous-listes ont une taille inférieure ou égale à 1, elles sont déjà triées.

Question 2 : Code C de l'algorithme (1.5 point)

Voici le code C correspondant à l'algorithme de tri rapide :

```c
#include <stdio.h>

// Fonction pour echanger deux elements
void echanger(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

```c

// Fonction pour partitionner le tableau
int partition(int arr[], int low, int high) {
    int pivot = arr[high];  // Choisir le dernier element comme pivot
    int i = low - 1;        // Indice de l'element plus petit

    for (int j = low; j < high; j++) {
        if (arr[j] < pivot) {
            i++;
            echanger(&arr[i], &arr[j]);
        }
    }
    echanger(&arr[i + 1], &arr[high]);
    return (i + 1);
}

// Fonction recursive pour le tri rapide
void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);  // Indice de partition
        quickSort(arr, low, pi - 1);         // Tri des elements avant la
           partition
        quickSort(arr, pi + 1, high);        // Tri des elements apres la
           partition
    }
}
```

Listing 1: Code C de QuickSort

Question 3 : Complexité temporelle (3 points)

Pire cas (1.5 point) Dans le pire cas, le pivot choisi est toujours l'un des éléments extrêmes (le plus petit ou le plus grand). L'algorithme doit alors trier une sous-liste de taille $n-1$ à chaque étape. L'équation de récurrence devient :

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

La résolution de cette équation donne une complexité temporelle de $O(n^2)$.

Meilleur cas (1.5 point) Dans le meilleur cas, le pivot divise parfaitement la liste en deux sous-listes égales à chaque étape. L'équation de récurrence devient :

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

La résolution de cette équation donne une complexité temporelle de $O(n \log n)$.

—

Exercice 3 : Programmation dynamique – Problème du plus grand carré blanc (7 points)

Question 1 : Formulation de l'équation récursive (1.5 point)

Soit $dp[i][j]$ la taille du plus grand carré entièrement constitué de 1 se terminant en position (i, j). L'équation récursive est donnée par :

$$dp[i][j] = \begin{cases} 0 & \text{si } M[i][j] = 0 \\ \min(dp[i-1][j], dp[i][j-1], dp[i-1][j-1]) + 1 & \text{si } M[i][j] = 1 \end{cases}$$

Question 2 : Code C de l'approche récursive avec mémoïsation (1.5 point)

```c
#include <stdio.h>
#include <string.h>

#define MAX 100

int memo[MAX][MAX];

// Fonction recursive avec memoiisation
int maxSquareRecursive(int M[MAX][MAX], int i, int j) {
    if (i < 0 || j < 0) return 0;
    if (memo[i][j] != -1) return memo[i][j];
    if (M[i][j] == 0) return memo[i][j] = 0;
    memo[i][j] = 1 +
    min(min(maxSquareRecursive(M, i-1, j),
    maxSquareRecursive(M, i, j-1)),
    maxSquareRecursive(M, i-1, j-1));
    return memo[i][j];
}
```

Listing 2: Code C de l'approche récursive avec mémoïsation

Question 3 : Code C de l'approche itérative (du bas vers le haut) (1.5 point)

```c
#include <stdio.h>
#define MAX 100

int maxSquareBottomUp(int M[MAX][MAX], int n) {
    int dp[MAX][MAX] = {0};
    int maxSize = 0;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (M[i][j] == 1) {
                if (i == 0 || j == 0)
                    dp[i][j] = 1;
                else
                    dp[i][j] = 1 +
                    min(min(dp[i-1][j], dp[i][j-1]),
                    dp[i-1][j-1]);

                if (dp[i][j] > maxSize)
                    maxSize = dp[i][j];
            }
        }
    }
    return maxSize;
}
```

Listing 3: Code C de l'approche itérative

Question 4 : Analyse de la complexité (2.5 points)

Complexité temporelle (1 point) La complexité temporelle des deux approches est $O(n^2)$, car chaque cellule de la matrice est visitée une seule fois.

Complexité spatiale (1.5 point) La complexité spatiale est également $O(n^2)$ en raison de l'utilisation d'une matrice auxiliaire dp de taille $n \times n$.