Examen : Structure de donnée

Dr. Jehan-Antoine Vayssade

24 mars 2025

Instructions

— **Durée :** 1 heure 30 min
— **Note totale / maximale :** 20 points
— **Des points bonus peuvent être cachés !**
— **Les réponses doivent être argumentées**
— **Prenez uniquement un stylo**
— **Pour la rédaction des codes :** ils peuvent être rédigés dans des langages du même paradigme que C, tels que Python, C++ ou Java. (Attention aux coûts cachés).

1 Définition (3 points)

— Expliquez le concept de complexité asymptotique.
— Définissez les trois notations asymptomatiques.
— Quels sont les avantages et inconvénients de ces notations ?
— A quoi correspond la sémantique axiomatique ?
— Qu'est ce que cela permet de faire de plus ou de moins ?

2 Comparaison et swap (1 point)

Dans les algorithmes de tri, on sépare parfois la complexité temporelle en deux, notamment pour le nombre de comparaisons et le nombre de swaps. Expliquez cette distinction.

3 Analyse de récurrence (2 point)

Analysez la complexité temporelle et spatiale de la fonction récursive suivante. Donnez la notation asymptotique.

```
1    int mystery(int n) {
2        if (n <= 1) return n;
3        return mystery(n / 2) + n;
4    }
```

4 Rotation dans une liste chaînée (2 point)

Écrivez une fonction en C pour effectuer une rotation à droite d'une liste simplement chaînée (le dernier noeud devient le premier). Définissez la structure de données utilisée. Quelle est la complexité ?

5 Traversée d'arbre binnaire (3 point)

La traversée d'un arbre binaire peut s'effectuer selon trois méthodes : Pre-order, In-order et Post-order. Voici un exemple de signature de fonction pour effectuer cette traversée, complétez-la pour les différents cas de figure. Donnez un exemple d'arbre et de sortie en utilisant les trois fonctions.

```
1    typedef struct Node {
2        char data;
3        struct Node *left, *right;
4    } Node;
5
6    void binnairy_tree_order_traversal(Node *tree)
7    {
8        if(!tree) return;
9        // TODO
10       printf(``%c '', tree->data);
11       // TODO
12   }
```

6 Optimisation (3 points)

Analysez la complexité du code de tri boustrophédon ci-dessous, en termes de temps (comparaison et swap) et d'espace (pire cas et meilleur cas). Expliquez comment la complexité pourrait être améliorée. Proposez des suggestions pour réduire le nombre de comparaisons et de swaps inutiles. Modifiez le code pour optimiser ces aspects et donnez les nouvelles complexités.

```
1  void boustrophedonsort(void *data, int count, size_t size, CompareFunc
       compare) {
2      int start = 0, end = count - 1;
3      char *cdata = (char*)data;
4
5      while (end != start)
6      {
7          for (int i = start; i < end; ++i)
8              if (compare(cdata + i * size, cdata + (i + 1) * size) > 0)
9                  memswap(cdata + i * size, cdata + (i + 1) * size, size
                       );
10         end--;
11
12         if(end == start)
13             break;
14
15         for (int i = end; i >= start; --i)
16             if (compare(cdata + (i - 1) * size, cdata + i * size) > 0)
17                 memswap(cdata + (i - 1) * size, cdata + i * size, size
                       );
18         start++;
19     }
20 }
```

7 Composantes connexes (2 points)

Une composante connexe dans un graphe non orienté est un sous-graphe où chaque paire de noeuds est connectée par un chemin. En d'autres termes, tous les noeuds d'une composante connexe sont accessibles les uns aux autres via des arêtes.

Considérons un graphe non orienté avec deux groupes de noeuds séparés. Chaque groupe forme une composante connexe, car les noeuds à l'intérieur d'un groupe sont tous connectés, mais il n'existe aucun chemin entre les deux groupes.

Comment pouvez-vous procéder pour compter le nombre de composantes connexes ? Sur quels algorithmes pouvez-vous vous appuyer ? Sachant que cette fonction est disponible, proposez un pseudo-code pour compter les composantes connexes dans une matrice d'adjacence. Sur quelle structure de données cet algorithme repose-t-il ?

8 Min-Heap (4 points)

Un min-heap est une structure de données arborescente complète qui satisfait deux propriétés fondamentales : la forme et la propriété du tas. Donc (1) un min-heap est un arbre binaire complet, ce qui signifie qu'il est rempli à tous les niveaux, sauf peut-être le dernier niveau, qui est rempli de gauche à droite. Cela implique que tous les noeuds internes ont deux enfants, sauf ceux du dernier niveau, qui peuvent avoir zéro ou un enfant droit. Et (2) la propriété du tas stipule que la valeur de chaque noeud parent est inférieure ou égale à celle de ses enfants. Cela garantit que le minimum de l'arbre se trouve toujours à la racine.

La constitution d'un tas à partir d'un tableau existant implique plusieurs étapes clés :

— Initialisation : On commence avec un tableau d'éléments non triés.
— Construction du Tas : On itère à travers le tableau, en commençant par le dernier noeud non-feuille (c'est-à-dire le parent du dernier noeud) et en remontant vers la racine. Pour chaque noeud, on applique l'opération sieving pour s'assurer que la propriété du tas est respectée.
— Sieving : Cette opération consiste à comparer la valeur du noeud actuel avec celles de ses enfants. Si un enfant a une valeur inférieure, on échange le noeud avec cet enfant et on répète l'opération récursivement (ou mieux itérativement) jusqu'à ce que la propriété du tas soit satisfaite.

Code de base

Pour vous aidez, vous pouvez utilisé ces fonctions, et complétez les parties manquantes :

```
1  void swap(int* a, int* b);
2
3  int sieving(int* array, int element, int max) { # TODO }
4
5  int main() {
6      int arr[] = {10, 5, 15, 2, 20, 30};
7      int n = sizeof(arr) / sizeof(arr[0]);
8
9      for (int i = #TODO#)
10         sieving(arr, i, n);
11
12     return 0;
13 }
```

Travail demandé

1. Proposez un exemple de min-heap (tableau + petit diagramme).
2. Complétez la boucle `for`, telle qu'expliquée pour la **constitution du tas**.
3. Définissez la position des enfants en fonction de l'indice i, soit (left et right).
4. Écrivez la fonction de **réorganisation du tas** (sieving).
5. Donnez la **complexité temporelle** (comparaisons et swaps) de sieving.
6. Donnez la **complexité spatiale** de sieving.
7. Utilisez cette fonction sieving pour construire une stratégie de tri.
8. Déduisez la complexité de cet algorithme de tri.