Correction colle d'algorithmique des données n°1

Exercice 1 : Questions de cours

Question 1

1. *Algorithme* : Séquence d'opérations élémentaires non ambigües permettant de résoudre un problème.
   Programme : Algorithme traduit au moyen d'un **langage de programmation.**
2. $\log(10^n) < n\log(n) < 3^{\ln(n)} < n^{10} < 10^n$

Question 2

1. Cf. cours
2. C(n) = a * C(n/k) + O(n) :
   - $a > k : C(n) = O(n^{\log_k(a)})$
   - $a = k : C(n) = O(n\log(n))$
   - $a < k : C(n) = O(n)$

Question 3

1. Structure de données en *FIFO* : le premier élément entré dans le file sera la premier sorti.
2. Structure de données en *LIFO* : le dernier élément entré dans le file sera la premier sorti.

Question 4

1. C'est la méthode du tri naturel des jeux de carte : on insère chaque carte à sa position jusqu'à avoir atteint la fin du paquet.
   En moyenne : $O(n^2)$
   En pire cas : $O(n^2)$, si le tableau est trié à l'envers
2. Utilise un pivot pour séparer les éléments qui sont plus petits et ceux plus grands.
   En moyenne : $O(n\log(n))$
   En pire cas ; $O(n^2)$, si le tableau est déjà trié

Exercice 2 : Calcul de la complexité

Question 1

L'opération fondamentale ici est l'addition.

Question 2

C(n) = 5n = O(n)
Explication : 5 passages dans la boucle interne : O(1), n passages dans la boucle externe : O(n)

Question 3

Résultat de l'ordre de O(n²)
Explication :

$$\sum_{i=1}^{n-i} i + \sum_{i=1}^{n-i}(i+1) + \sum_{i=1}^{n-i}(i+2) + \sum_{i=1}^{n-i}(i+3) + \sum_{i=1}^{n-i}(i+4) = 4\left(\sum i + (n-1) + 2(n-1) + 3(n-1) + 4(n-1)\right) = 2(n-1)n + 10(n-1)$$

Question 4

Dans chaque boucle :

- f(n) en O(n)
- 1 addition : O(1)

On a f(n) tours de boucle, soit O(n²) itérations. Soit **C(n) = O(n³)**

Question 5

On calcule la somme des i de 0 à n^2, soit la somme des i de 0 à m, avec m = n^2, ce qui donne une complexité en O(n^4)

Question 6

```c
int g (int n) {
    int y = 0;
    int i;
    int max = f(n);
    for (i = 0; i < max; i++) {
        y += i;
    }
    return y;
}
```

Exercice 3 : Vecteur creux

Question 1

```c
double access (const struct array *vector, size_t index) {
    size_t i;

    for (i = 0; i < vector->size; i++) {
        if (vector->data[i].index == index) {
            return vector->data[i].value;
        }
    }

    return 0; // si la valeur n'est pas présente, c'est qu'elle est nulle
}
```

`access` est de complexité O(n).

Question 2

On ajoute à la fin du tableau dynamique, opération en O(1) **amorti** (complexité de l'ajout dans un tableau dynamique).

Question 3

On a au minimum du O(n) pour le parcours du tableau afin de vérifier si l'indice existe déjà.
L'ajout de la valeur est ensuite en O(1) (amorti suivant les cas, si on atteint la capacité du tableau).
On a donc une complexité en O(n).

Question 4

0 <= `result->size` <= `right->size` + `left->size`donc la taille maximale est obtenue quand tous les indices sont différents.

Question 5

On a 2 double boucles, qui parcourent chacun des vecteurs creux à ajouter : O(n^2)

Question 6

On utilise un **algorithme de recherche dichotomique**, en O(log(n)).

Question 7

On recherche si l'indice est présent : $O(\log(n))$

- s'il est présent, on ne fait rien : $O(1)$
- sinon, on l'insère de manière à garder le tableau trié : $O(n)$

Question 8

```c
void add (struct array *result, const struct array *left, const struct array *right) {
    size_t i, j, k;

    // on s'assure que le vecteur de destination est bien vide
```

```c
    if (result->data != NULL) {
        free(result->data);
    }
    result->capacity = right->size + left->size;
    result->data = calloc(result->capacity, sizeof(struct coord));

    // tant qu'on n'est pas arrivé au bout des deux vecteurs
    while ((j < left->size) || (k < right->size)) {
        while ((k == right->size || j < left->size) && (left->data[j].index < right->data[k].index)) {
            // les indices de left sont plus petits que ceux de right : on les ajoute
            result->data[i] = left->data[j];
            i++;
            j++;
        }

        while ((k < right->size || j == left->size) && (right->data[k].index < left->data[j].index)) {
            // on fait l'inverse
            result->data[i] = right->data[k];
            i++;
            j++;
        }

        while (right->data[k].index = left->data[j].index) {
            // les indices de left et right sont égaux
            double value = left->data[j].value + right->data[k].value;
            if (value != 0) {
                // on ajoute un des index
                result->data[i].index = left->data[j].index;
                result->data[i].value = value;
                i++;
            }
        }
    }
}
```

Question 9

Cette fonction `add` est en O(n).

Question 10

On trie les tableaux avant :

- tri en $O(n \log(n))$
- `add` en $O(n)$

Soit une complexité totale en $O(n \log(n))$.

Question pour les vainqueurs

Quelle structure choisir pour implanter les vecteurs creux ?

En choisissant des tableaux triés, on obtient des complexités plus faibles partout, c'est donc la meilleure solution.
Autrement, on pourrait utiliser un arbre binaire de recherche, trié selon les indices (avec `access` en $O(n \log(n))$, `set` en $O(n \log(n))$, `add` en $O(n)$).

Pour un vecteur dense, on a `access` en $O(1)$, `set` en $O(1)$ et `add` en $O(k)$, mais avec un coût important en mémoire.