Série N°02 – Algorithmes de tri

Exercice 01 : Tri un peu spécial (compétition – solve it 2023)

Ecrire une fonction ***« valeurMax »*** qui prend comme paramètre un tableau ***« T »*** d'entiers positifs (ou nuls), et qui réorganise ses cases tel que le tableau forme la plus grande valeur possible. Vu que la valeur maximale peut être très grande, retournez un tableau de caractères (string) plutôt qu'un entier.

Exemples d'entrées et de sorties :

Entrée : T = {1, 2, 3, 4, 5, 6, 7, 8, 9, 0}
Sortie : "9876543210"

Entrée : T = {54, 14, 89, 2, 301, 45, 7}
Sortie : "8975432143012"

La fonction doit avoir la signature suivante :

```
char* valeurMax(int T[], int n);
```

Indication : utiliser n'importe quel algorithme de tri (idéalement « quick-sort » ou « merge-sort »), mais il faut bien choisir le comparateur

```
int compare(int a, int b) {
    // Placer les nombres dans des ordres différents puis les comparer
    char concat1[40], concat2[40];
    sprintf(concat1, "%d%d", a, b);
    sprintf(concat2, "%d%d", b, a);
    return strcmp(concat1, concat2);
    // Si la valeur retournée > 0 alors « concat1 » est supérieur
    // (« a » doit précéder « b » dans l'ordre)
    // Si la valeur retournée < 0 alors « concat1 » est supérieur
    // (« b » doit précéder « a » dans l'ordre)
}

void bubbleSort(int T[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (compare(T[j], T[j + 1]) < 0) {
                // Si T[j + 1] doit précéder T[j] on permute les valeurs.
                swap(&T[j], &T[j + 1]);
            }
        }
```

```
    }
}

char* valeurMax(int T[], int n) {
    bubbleSort(T, n);
    // Allouer un espace mémoire pour la chaîne résultante
    char *result = malloc(20 * n);
    result[0] = '\0'; // Initialiser la chaîne
    // Concaténer les nombres dans l'ordre
    for (int i = 0; i < n; i++) {
        sprintf(result, "%s%d", result, T[i]);
    }
    return result; // Retourner la chaîne résultante
}
```

Exercice 02 : La valeur minimale possible

Ecrire une fonction ***« minMemeTaille »*** qui prend comme paramètre un nombre entier, et qui réorganise ses chiffres tel que :

- La valeur du résultat est la plus petite possible en conservant les mêmes chiffres.
- Il ne doit pas y avoir de zéro en début de nombre (sauf si le nombre est égal à 0).

Indications

- Penser à utiliser une version légèrement modifiée d'un algorithme de tri (supposez que vous avez déjà un algorithme de tri implémenté).
- Attention aux nombres négatifs (dans ce cas, vous devez maximiser la valeur absolue).

```
// Fonction pour comparer deux caractères pour qsort (tri croissant)
int compareAscending(const void *a, const void *b) {
    return (*(char *)a - *(char *)b);
}

// Fonction pour comparer deux caractères pour qsort (tri décroissant)
int compareDescending(const void *a, const void *b) {
    return (*(char *)b - *(char *)a);
}

int minMemeTaille(int n) {
    // Si n est 0, retourner 0 directement
    if (n == 0) {
        return 0;
    }
    // Vérifier si le nombre est négatif
    bool is_negative = (n < 0);
    n = abs(n); // Travailler avec la valeur absolue
```

```
    // Convertir le nombre en chaîne pour manipuler ses chiffres
    char digits[11]; // Pour un int, il y a au maximum 10 chiffres + '\0'
    sprintf(digits, "%d", n);
    // Obtenir la longueur de la chaîne
    int len = strlen(digits);
    // Trier les chiffres
    if (is_negative) {
        // Si le nombre est négatif, trier dans l'ordre décroissant
        qsort(digits, len, sizeof(char), compareDescending);
    } else {
        // Si le nombre est positif, trier dans l'ordre croissant
        qsort(digits, len, sizeof(char), compareAscending);
        // S'assurer qu'il n'y ait pas de zéro en tête
        if (digits[0] == '0') {
            for (int i = 1; i < len; i++) {
                // Échanger le premier zéro avec le premier chiffre non nul
                if (digits[i] != '0') {
                    char temp = digits[0];
                    digits[0] = digits[i];
                    digits[i] = temp;
                    break;
                }
            }
        }
    }
    // Reconstruire le nombre à partir des chiffres tries
    // atoi converte une chaine à un entire.
    int result = atoi(digits);
    return is_negative ? -result : result;
}
```

Exercice 03 : La valeur manquante

Soit ***T*** un tableau d'entier de taille ***n***. On suppose que ***T*** contient tous les entiers de ***0*** à ***n*** sauf un. L'objectif de cet exercice est d'écrire une fonction qui renvoie la valeur manquante.

- La solution naïve est de parcourir le tableau et de tester pour chaque entier de ***0*** à ***n*** s'il est présent dans le tableau. Donner l'implémentation de cette solution. Quelle est sa complexité ?

```
// Solution naïve : O(n²)
int valeurManquanteNaive(int T[], int n) {
    for (int i = 0; i <= n; i++) {
        bool found = false;
        for (int j = 0; j < n; j++) {
            if (T[j] == i) {
                found = true;
                break;
            }
        }
        if (!found) {
            return i; // Retourner l'entier manquant
        }
    }
    // Cette instruction finale est inatteignable
    // car il y toujours une valeur manquante
    // Mais le langage C oblige un retour non conditionné
    return -1;
}
```

- Comment peut-on réduire la complexité de cet algorithme ? Implémenter cette solution.

```
// Fonction de comparaison pour qsort
int compare(const void *a, const void *b) {
    return (*(int *)a - *(int *)b);
}

// Fonction pour trouver la valeur manquante
int valeurManquanteParTri(int T[], int n) {
    // Trier le tableau
    qsort(T, n, sizeof(int), compare);
    // Vérifier l'écart avec l'indice attendu
    for (int i = 0; i < n; i++) {
        if (T[i] != i) {
            return i; // Le premier écart indique la valeur manquante
        }
    }
    // Si aucun écart n'est trouvé, la valeur manquante est n
    return n;
}

// Meilleur solution : O(n)

int valeurManquanteOptimisee(int T[], int n) {
    // Calcul de la somme attendue de 0 à n
    int sommeAttendue = n * (n + 1) / 2;
    // Calcul de la somme réelle des éléments du tableau
    int sommeReelle = 0;
    for (int i = 0; i < n; i++) {
        sommeReelle += T[i];
```

```
    }
    // La valeur manquante est la différence entre les deux
    return sommeAttendue - sommeReelle;
}
```

Exercice 04 : Chercher deux nombres avec une somme donnée

L'objectif de cet exercice est d'écrire une fonction ***« somme2 »*** qui prend comme argument un tableau ***T*** et un entier S et qui retourne vrai si le tableau T contient deux nombres dont la somme est égale à S. Par exemple, si T = [1, 2, 3, 4, 5] et S = 7, la fonction doit retourner vrai car 3 + 4 = 7.

- Commencer par implémenter une version naïve de cette fonction qui teste toutes les paires possibles de nombres dans le tableau. Quelle est la complexité de cette fonction ?

```
bool somme2Naive(int T[], int n, int S) { // O(n²)
    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j < n; j++) {
            if (T[i] + T[j] == S) {
                return true; // Une paire trouvée
            }
        }
    }
    return false; // Aucune paire trouvée
}
```

- Comment peut-on utiliser les algorithmes de tri pour améliorer la complexité de cette fonction en complexité sous-quadratique ? Implémenter cette version.

```
// Fonction de comparaison pour qsort
int compare(const void *a, const void *b) {
    return (*(int *)a - *(int *)b);
}

bool somme2Optimisee(int T[], int n, int S) { // O(n log(n))
    // Trier le tableau
    qsort(T, n, sizeof(int), compare);
    // Deux indices : un au début, l'autre à la fin
    int i = 0, j = n - 1;
    while (i < j) {
        int somme = T[i] + T[j];
        if (somme == S) {
            return true; // Une paire trouvée
        } else if (somme < S) {
            i++; // Augmenter la somme
        } else {
```

```
            j--; // Réduire la somme
        }
    }
    return false; // Aucune paire trouvée
}
```