IFT339 - Tableaux et gestion de mémoire dynamique

Les circonstances ayant été très peu favorables, les exercices 1 à 3a ont été complétés pour vous. Vous êtes invités à analyser le code du header Tableau et à vous assurer de bien comprendre son fonctionnement. Plusieurs concepts s'appliquent aux structures de données en général et reviendront tout au long du cours.

Il vous est fortement suggéré de compléter les exercices 3b, 3c et 3d pour mettre en pratique la compréhension des exigences et vos habiletés à conçevoir une solution appropriée dans un objectif d'optimalité algorithmique.

La série d'exercices suivante demande l'analyse et la modification du programme C++ qui l'accompagne. Portez une attention particulière à l'utilisation des pointeurs et à la gestion de la mémoire tout au long des exercices. Ces connaissances vous sont requises pour assurer votre réussite de ce cours.

Conseil du jour : la plupart des éditeurs de code permettent de commenter un bloc entier de code surligné avec le raccourci ctrl-k-c, ou de décommenter avec ctrl-k-u.

Exercice 1

a) Observez le code de la section *Exercice 1*. Ce code cause un plantage à l'exécution. Pourquoi ?

b) Implémentez la fonction de surcharge de l'opérateur = dans le header tableau.h.

c) L'implémentation de la fonction de surcharge de l'opérateur = n'est pas suffisante. Pourquoi ?

d) Implémentez la fonction de surcharge du constructeur par copie dans le header tableau.h.

Exercice 2

Décommentez la dernière ligne de la section *Exercice 2* et observez le code de cette section. Implémentez la méthode nécessaire au comportement attendu dans le header tableau.h. Ce comportement est le suivant ; l'instruction "*tableau1 + tableau2*" doit retourner un nouveau tableau représentant la concaténation des deux tableaux en entrée, soit qui, pour les instances en exemple, contient les éléments de *tableau1* suivis des éléments de *tableau2*. Conséquemment, la taille du tableau retourné est la somme des tailles des deux tableaux concaténés.

Exercice 3

Implémentez la ou les fonctions nécessaires à la résolution des problèmes suivants, et calculez la complexité algorithmique de vos implémentations. Des instances sont fournies en exemple, mais il

est attendu que vos solutions soient valides et correctes pour toute instance du type spécifié. Vous ne devez utiliser que la structure de la classe Tableau, et ne pouvez pas instancier de nouvelles structures telles un vecteur ou une table de hashage.

a) Implémentez la fonction *min_max* qui reçoit en entrée un tableau d'entiers *tab* et retourne une paire (i, j) d'indices telle que la différence entre $tab[i]$ et $tab[j]$ est maximum. Utilisez l'instance *tableau* en exemple.

b) Implémentez la fonction *buy_low_sell_high* qui reçoit en entrée un tableau d'entiers *tab* et retourne une paire (i, j) d'indices telle que la différence entre $tab[i]$ et $tab[j]$ est maximum, **et** tel que $i < j$. Utilisez l'instance *tableau* en exemple.

c) Implémentez la fonction *is_anagram* qui reçoit deux tableaux *tab*1 et *tab*2 de *char*, et retourne *true* si le contenu de ces tableaux représente des anagrammes, c'est-à-dire, qu'il y a une façon de réordonner les éléments de *tab*2 pour obtenir un tableau identique à *tab*1, et vice-versa. Les instances *tableau3* et *tableau4* sont fournies en exemple et représentent un anagramme. Pour cette question, vous pouvez conçevoir vos propres méthodes utilitaires si nécessaire, ou utiliser celles fournies par la librairie standard.
Indice : une nouvelle fonction dans le header, ainsi que std : :sort() pourraient s'avérer utiles.

d) Implémentez la fonction *missing_int* qui reçoit un tableau *tab* de n entiers contenant tous les entiers compris entre 0 et n sauf un déterminé arbitrairement, et retourne un *int* représentant cet entier manquant. Il est garanti que chaque élément du tableau *tab* est unique. Les instances *tableau5*, *tableau6* et *tableau7* sont fournies en exemple.