TP1 : Ensembles

Préliminaires.
Le TP est à effectuer en C++ à l'aide de vos outils habituels.

(1) **Environnement de travail.** Dans le dossier `Documents`, créez un dossier `MathInfo`.

(2) Téléchargez depuis le site l'archive `TP1.zip` (c'est un dossier compressé qui contient plusieurs documents).

(3) Ouvrez l'archive et cliquez sur **Extraire** : extrayez les fichiers vers le dossier `Documents/MathInfo`.

(4) Ouvrez le fichier `TP1.cpp` avec `Code::Blocks` (bouton droit > ouvrir avec > autre application > `Code::Blocks`)

(5) **Configurer votre compilateur C++.** Dans `Code::Blocks`, suivez les instructions suivantes :

```
Settings -> Compiler -> Global compiler settings -> Compiler Flags :
Have g++ follow the C++11 ISO C++ language standard
```

Si vous utilisez un autre logiciel, faites en sorte que votre commande de compilation soit :

```
g++ -Wall -std=c++11 -o "%e" "%f"
```

(6) **Compiler le fichier** `TP1.cpp`. Vérifier que vos options de compilation sont bien réglées en compilant le fichier `TP1.cpp`. Aucune erreur ne doit apparaître à la compilation. Ce fichier contient **des fonctions à compléter**, ainsi que **des fonctions de tests**, et une fonction `main` à adapter à votre avancement. À la fin du TP, toutes les instructions de la fonction `main` doivent pouvoir s'exécuter sans erreur.

(7) **Rappel C++**

Vous n'étiez pas à UPsud l'année dernière et n'avez jamais fait de C++ ? Pas de panique ! Le fichier `baseCpp.cpp` est là pour vous aider. Si vous venez du langage C, la transition sera très facile. Si vous venez d'un autre langage, ça prendra un peu plus de temps mais vous y arriverez très bien. Faites savoir à votre chargé de TP que vous débutez en C++, nous sommes là pour vous aider. Les cours de L1 de Nicolas Thiéry peuvent vous aider.

```
// Créer un tableau vide
vector<int> t = vector<int>();

// Créer un tableau avec une taille donnée
vector<int> t = vector<int>(5);

// Ajouter un élément à la fin d'un tableau
t.push_back(3);

// Obtenir la taille d'un tableau
t.size();
```

Exercice 1 (Mise en jambe)**.**
Lors de ce TP, nous allons modéliser des opérations de base sur les ensembles. Nous nous limitons aux **ensembles d'entier** que nous représenterons en C++ par un tableau sous forme de `vector<int>`. Quelques règles :

— Chaque nombre ne doit apparaître qu'une seule fois.
— L'ordre des nombres n'a pas de signification (en particulier, plusieurs tableaux peuvent représenter le même ensemble).

(1) Lesquels de ces tableaux suivent nos conventions ? Pour chaque tableau correct, donner l'ensemble représenté.

2

1	2

1	1

5	2	8	1	9

2	1	8	5	9

3	6	3	5	4

1	2	3	4	5

Dans la suite des exercices, on supposera toujours que les tableaux ont déjà le bon format.

(2) Compléter la fonction `afficheEnsemble` qui prend en argument un ensemble sous forme de `vector<int>` et l'affiche. Tester la fonction grâce à la première instruction de `main` qui appelle `testAffiche`. **Remarque : on suppose que le tableau reçu en argument a le format correct, on demande simplement de l'afficher (pas de le corriger).**

(3) Compléter la fonction `affiche123` qui affiche toutes les façons possible de représenter l'ensemble $\{1, 2, 3\}$.

Exercice 2 (Ensembles inclus, ensembles égaux)**.**
Dans cet exercice, on va chercher à implanter un test d'égalité pour deux ensembles. Étant donné qu'un même ensemble peut être représenté par plusieurs tableaux différents, un **test d'égalité sur les tableaux ne suffit pas**. Nous utiliserons la caractérisations suivante :

$$A = B \Leftrightarrow (A \subset B) \text{ et } (B \subset A)$$

(1) Compléter la fonction `appartient` qui teste l'appartenance d'un élément à un ensemble et la tester grâce à la fonction fournie.

(2) Utiliser cette fonction pour implanter la fonction `inclus` qui teste l'inclusion d'un ensemble dans un autre. La tester avec la fonction fournie.

(3) Enfin, utiliser la fonction `inclus` et la **caractérisation** donnée en début d'exercice pour compléter la fonction `egal` et la tester avec la fonction fournie.

Exercice 3 (Opérations élémentaires)**.**

(1) Compléter la fonction `ajoute`. **Rappel : un ensemble ne doit pas contenir plusieurs fois la même valeur !** Tester avec la fonction fournie.

(2) Compléter les fonctions `unionEns`, `intersectionEns`, `soustractionEns`, `differenceEns` et les tester avec les fonctions fournies (tester à chaque fois !). **Pensez à utiliser les fonctions des questions précédentes !**

Aide par l'exemple
Union : $\{3,1,2,7\} \cup \{2,4,3,5,8\} = \{3,1,2,7,4,5,8\}$
Intersection : $\{3,1,2,7\} \bigcap \{2,4,3,5,8\} = \{3,2\}$
Soustraction : $\{3,1,2,7\} \setminus \{2,4,3,5,8\} = \{1,7\}$
Différence symétrique $\{3,1,2,7\} \triangle \{2,4,3,5,8\} = \{1,7,4,5,8\}$

Exercice 4 (Ensemble des parties)**.**
Dans cet exercice, on se propose de calculer l'ensemble des parties d'un ensemble d'entier. C'est-à-dire que nous allons afficher **tous les sous-ensembles** d'un ensemble donné. Pour cela, nous utilisons une correspondance entre les **entiers de 0 à** 2^n-1 et les 2^n sous-ensemble d'un ensemble à n éléments. On se sert de **l'écriture binaire des entiers.**

Soit E un ensemble de taille n représenté par le tableau V et k un entier tel que $0 \leq k < 2^n$. Alors le sous-ensemble e_k de E donné par k est obtenu par la règle suivante : $V[i] \in e_k$ ssi le bit en position i de k est 1 (le bit de poids faible est en position 0). Par exemple, si $E = \{2,1,3\}$, $V = [2,1,3]$ et $k = 6$. En binaire, k s'écrit 110. Son bit en position 0 (bit de poids faible) est 0 donc $V[0] = 2 \notin e_6$. Son bit en position 1 est 1 donc $V[1] = 1 \in e_6$. Son bit en position 2 est 1 donc $V[2] = 3 \in e_6$. Au final, $e_6 = \{1,3\}$.

La fonction `bitn` vous est donnée : elle retourne vrai si bit de poids i de n est 1.

(1) **Entraînement** Soit $V = [3,6,5,8,9,1,4]$ un tableau représentant un ensemble E, donner le sous-ensemble e_{99}.
(2) Compléter la fonction `sousEnsemble` et la tester avec la fonction donnée.
(3) Modifier la fonction `testSousEnsemble` pour ajouter un test correspondant au calcul fait dans la question 1.
(4) Compléter les fonctions `nbSousEnsembles` puis `afficheSousEnsembles` et les tester.

Exercice 5 (Aller plus loin *clubsuit*)**.**

(1) Quelles sont les complexités des fonctions `appartient`, `ajout`, `union`, `intersection`, `soustraction` et `difference` ?
(2) S'il était possible de tester l'appartenance et d'ajouter en $O(1)$, comment cela affecterait-il les complexités des autres fonctions ?
(3) Effectuer une recherche pour trouver des *pistes* et des *mots clés* sur les structures de données adaptées à une telle implantation.