EIDGENÖSSISCHE TECHNISCHE HOCHSCHULE – LAUSANNE
POLITECNICO FEDERALE – LOSANNA
SWISS FEDERAL INSTITUTE OF TECHNOLOGY – LAUSANNE

Faculté Informatique et Communications
Cours de Programmation I, sections MA et PH
Chappelier J.-C.

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

NOM : Hanon Ymous
(000000) #0000
Place : 1

Programmation I (SMA/SPH) : SÉRIE NOTÉE

23 novembre 2017

INSTRUCTIONS (à lire attentivement)

IMPORTANT ! Veuillez suivre les instructions suivantes à la lettre sous peine de voir votre série annulée dans le cas contraire.

1. Vous disposez d'une heure quarante-cinq minutes pour faire cet examen (10h15 - 12h00).
2. Vous devez **écrire à l'encre noire ou bleu foncée**, pas de crayon ni d'autre couleur.
   N'utilisez **pas non plus de stylo effaçable** (perte de l'information à la chaleur).
3. Vous avez droit à toute documentation papier.
   En revanche, vous ne pouvez pas utiliser d'ordinateur personnel, ni de téléphone portable, ni aucun autre matériel électronique.
4. Répondez aux questions directement sur la donnée ; utilisez aussi le verso des feuilles, **MAIS** n'utilisez *que* le verso de la feuille sur laquelle se trouve la question, et non **pas** celui de la feuille précédente !
   Ne joignez aucune feuilles supplémentaires ; **seul ce document sera corrigé**.
5. Lisez attentivement et *complètement* les questions de façon à ne faire que ce qui vous est demandé. Si l'énoncé ne vous paraît pas clair, ou si vous avez un doute, demandez des précisions à l'un des assistants.
6. L'examen comporte deux exercices indépendants, qui peuvent être traités dans n'importe quel ordre, mais qui ne rapportent pas la même chose (les points sont indiqués, le total est de 72). Tous les exercices comptent pour la note finale.

Question 1 — Vérification de Sudoku [sur 23 points]

Le cadre de cet exercice est celui de la vérification de grilles de Sudoku. En fait, nous allons ici nous attaquer à un sous-problème un peu plus simple, dont le principe est le suivant : étant donnée une grille de dimensions $N \times N$, trouver si elle est dans une configuration valide, où une configuration est valide si :

- toutes les lignes de la grille contiennent exactement les nombres $1, 2, 3..., N$ (sans répétition, donc) ;
- idem pour les colonnes.

Par exemple, parmi les configurations ($N = 4$ ici) :

1	2	3	4
3	4	1	2
2	3	4	1
4	1	2	3

1	2	3	4
3	4	1	2
3	2	4	1
4	1	2	3

1	2	3	4
3	4	4	2
2	3	1	1
4	1	2	3

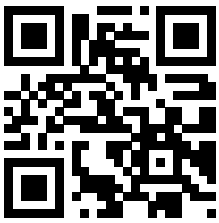
seule la première est correcte ; la seconde grille n'est pas valide en raison de la répétitions de 3 dans la première colonne et de 2 dans la deuxième ; et la troisième grille n'est pas valide par exemple parce que 4 est répété deux fois en seconde ligne.

Question 1.1 – Structures de données [sur 2 points]

Étant donnée la déclaration suivante :

```
constexpr size_t N = 4;
```

définissez (ici) les types `Line` et `Sudoku` ; une `Line` étant un tableau de `N` entiers et un `Sudoku` étant un tableau de `N` `Line`s.



Question 1.2 – Test de lignes [sur 12 points]

Écrivez une fonction `test_line` qui prend une `Line` en paramètre et retourne `true` ou `false` en fonction que cette ligne est valide ou non ; c.-à-d. en fonction qu'elle contient, ou non, exactement une fois chaque valeur entre 1 et `N`.

Écrivez ensuite une fonction `test_lines` (avec un 's') qui teste si toutes les lignes d'un `Sudoku` sont valides ou non.

suite au dos ☞

Question 1.3 – Test d'*une* colonne [sur 6 points]

Écrivez une fonction `test_column` qui prend un `Sudoku` et un indice de colonne en paramètres et retourne `true` ou `false` en fonction que la colonne correspondante est valide ou non ; c.-à-d. qu'elle contient, ou non, exactement une fois chaque valeur entre 1 et `N`.

Question 1.4 – Test complet [sur 3 points]

On suppose que la définition de la fonction `test_columns` (avec un 's' ; laquelle teste si toutes les colonnes d'un `Sudoku` sont valides ou non) vous est donnée (vous n'avez pas à la coder !).

Écrivez pour finir une fonction qui vérifie si un `Sudoku` est valide (c.-à-d. que toutes ses lignes et toutes ses colonnes sont valides au sens défini ci-dessus).

Question 2 — Gestionnaire d'intervalles [sur 49 points]

Le but du programme considéré dans cette question est de représenter des réunions d'intervalles *bornés* (pas infinis), sans redondance ni recouvrement dans ces représentations. Par exemple pour représenter la réunion d'intervalles $[0, 2] \cup [1, 4[\cup [5, 6] \cup [6, 10]$, on stockera seulement $[0, 4[$ et $[5, 10]$.

Chaque intervalle individuel sera représenté par un début et une fin, de type `double`, ainsi que deux booléens indiquant si ces bornes sont incluses ou non dans l'intervalle (bords ouverts ou fermés).

Une réunion d'intervalles devra pouvoir contenir un nombre arbitraire d'intervalles (dans les limites de la mémoire disponible sur la machine, évidemment !). L'ordre des intervalles dans une réunion d'intervalles n'importe pas : on pourra représenter la réunion $[0, 4[\cup [5, 10]$ soit comme « $[0, 4[$ et $[5, 10]$ », soit comme « $[5, 10]$ et $[0, 4[$ ».

Question 2.1 – Structures de données [sur 3 points]

Commencez par définir (ici) les structures de données suivantes :

- `Intervalle`, structure de données pour représenter un intervalle comme expliqué ci-dessus ;
- `Reunion_Intervalles`, une structure de données pour stocker un ensemble (ou une *réunion*) d'intervalles.

Question 2.2 – Affichage [sur 4.5 points]

Écrivez (au dos) une fonction d'affichage pour les intervalles : `void affiche(Intervalle);`

Cette fonction affichera :

- le mot « `VIDE` » si l'intervalle est vide ; on supposera exister (voir plus loin) une fonction `est_vide` permettant de tester si un intervalle est vide ou non ;
- soit `[`, soit `]` en fonction que le début (borne inférieure) est inclus ou non dans l'intervalle ;
- les deux bornes, séparées par une virgule ;
- soit `]`, soit `[` en fonction que la fin (borne supérieure) est incluse ou non.

Exemple d'affichage : `[0, 4[`

suite au dos ☞

Question 2.3 – Fonction outil : égalité de `double` [sur 4 points]

Comme il est déconseillé de tester directement l'égalité de deux `double`, nous vous demandons d'écrire ici une fonction `egaux` qui teste si deux `double` (pas d'intervalle ici !) sont égaux à une précision donnée près (c.-à-d. si la valeur absolue de leur différence est inférieure à cette précision). La précision par défaut pour cette fonction doit être de 10^{-10} (qui s'écrit « `1e-10` » en C++).

Question 2.4 – Opérations sur les intervalles [sur 38.5 points]

On veut pouvoir effectuer les actions suivantes, détaillées plus loin :

1. tester si un intervalle est vide ;
2. tester si un nombre donné est dans un intervalle ;
3. savoir si deux intervalles sont joignables en un seul ;
4. calculer la fusion (= réunion) de deux intervalles joignables (laquelle est un intervalle) ;
5. ajouter un intervalle à une réunion d'intervalles, tout en garantissant la représentation non-redondante décrite en début d'exercice ;
6. faire la réunion de deux réunions d'intervalles, c.-à-d. ajouter une réunion d'intervalles à une autre réunion d'intervalles.

Les quatre premières fonctions ainsi que la dernière sont assez simples à écrire et font l'objet de la sous-question suivante. La cinquième fonction est traitée ensuite.

Question 2.4.1 – Cinq fonctions simples [sur 23.5 points]

[3.5 points] Commencez par coder (ici) la fonction `est_vide` permettant de savoir si un intervalle passé en paramètre est vide ou non. Un intervalle sera considéré comme vide si sa fin est strictement inférieure à son début, ou alors si les deux bornes sont égales (au sens défini en Question 2.3) et ne sont pas incluses (aucune des deux).

suite au dos ☞

[6 points] Codez ensuite (ici) la fonction `contient` permettant de tester si une valeur de type `double` est contenue dans un intervalle.
A noter qu'un intervalle vide ne contient de toutes façons rien.

[4 points] Continuez en codant (ici) la fonction `sont_joignables` permettant de tester si l'union de deux intervalles est représentable avec un seul intervalle ou non. Ce test de jointure est assez simple : il suffit que l'une des borne d'un des intervalles soit contenue dans l'autre intervalle.

Par exemple, $[1, 3]$ et $[2, 4]$ sont joignables, car $[1, 3] \cup [2, 4] = [1, 4]$, mais $[1, 2[$ et $]2, 3]$ ne le sont pas car 2 n'est présent dans aucun des deux intervalles. En revanche, $[1, 2[$ et $[2, 3[$ sont joignables (et donnent $[1, 3[$).

suite au dos ☞

[6 points] Continuez en codant (ici) la fonction `fusion` permettant de calculer l'union de deux intervalles supposés joignables (on ne le testera pas ici). Cette fonction retourne donc un `Intervalle`).

La réunion de deux intervalles joignables est très simple à calculer : c'est simplement l'intervalle dont le début est le plus petit des deux débuts et la fin la plus grandes des deux fins.

Quant à l'appartenance de ces deux bornes : elles sont présentes dans l'intervalle résultat si elles le sont dans (au moins) l'un des deux intervalles à fusionner. Voir les exemples précédents (`sont_joignables`) pour illustration.

Remarque : En C++, les fonctions `max()` et `min()` existent dans la bibliothèque `algorithm` (mais on ne vous demande pas les `#include` ici !).

[4 points] En supposant qu'existe déjà la fonction `ajouter` permettant d'ajouter *un* intervalle à une réunion d'intervalles en la modifiant (fonction numéro 5 dans la liste page 7), terminez cette sous-question en codant (ici) la fonction `ajouter` (même nom) permettant d'ajouter *une réunion d'intervalles* à une autre réunion d'intervalles.

Question 2.4.2 – Ajouter *un* intervalle [sur 14 points]

Pour finir, codez (en face, page suivante) la fonction `ajouter` permettant d'ajouter *un* intervalle à une réunion d'intervalles (en la modifiant).

L'algorithme d'ajout d'un intervalle à une réunion existante n'est pas complètement trivial. Si l'intervalle à ajouter n'est pas vide (sinon il n'y a rien à faire !), il faut parcourir tous les intervalles déjà stockés, et fusionner tous les intervalles qui sont rendus joignables par le nouvel intervalle. Si le nouvel intervalle n'est joignable avec aucun intervalle existant, il suffit alors simplement de l'ajouter à la réunion d'intervalles.

Par exemple, si `reu_invs` représente la réunion d'intervalles $[-1,2] \cup [3,5] \cup [6,8[$ et `inv` représente l'intervalle $[0,4]$, l'appel `ajouter(reu_invs, inv)` devra modifier `reu_invs` pour qu'il représente la réunion $[-1,5] \cup [6,8[$.

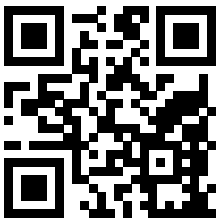
L'algorithme pourrait fonctionner comme suit sur cet exemple :

1. $[-1,2]$ est joignable à $[0,4]$ et donne $[-1,4]$; `reu_invs` contient alors à ce stade $[-1,4]$, $[3,5]$ et $[6,8[$;
2. $[-1,4]$ est lui-même joignable à $[3,5]$ et donne $[-1,5]$; `reu_invs` contient alors à ce stade $[-1,5]$ et $[6,8[$;
3. $[-1,5]$ n'est pas joignable à $[6,8[$.

Autre exemple : l'appel `ajouter(reu_invs, inv2)` où `inv2` représente $[-3,-2]$ et `reu_invs` représente la réunion d'intervalles $[-1,5] \cup [6,8[$, devra modifier `reu_invs` pour qu'il représente $[-1,5] \cup [6,8[\cup [-3,-2]$: il suffit de rajouter `inv2` à la fin de `reu_invs` (rappel : l'ordre n'importe pas) puisque `inv2` n'est joignable à aucun des intervalles de `reu_invs`.

Remarque : pour simplifier la tâche, on supposera que la fonction suivante, permettant de supprimer un intervalle donné par sa position dans une réunion d'intervalles, existe déjà :

```
void supprimer(Reunion_Intervalles& intervalles, size_t position);
```



(si nécessaire, place supplémentaire pour répondre ; mais
veuillez clairement indiquer le numéro de la question à laquelle vous répondez ici)