Devoir noté – Gestion d'intervalles

Jean-Cédric Chappelier

Version 1.0 du 1er mars

(du 14 mars 2022, 12h00, au 27 mars, 23h59.)

I. Introduction et instructions générales

Ce devoir noté consiste à écrire un programme pouvant effectuer diverses manipulations sur des réunions d'intervalles bornés (pas infinis, donc).

Nous vous demandons d'écrire tout votre code dans *un seul* fichier nommé `intervalles.c` sur la base d'un fichier fourni, à compléter.

Indications : Si un comportement ou une situation donnée n'est pas définie dans la consigne ci-dessous, vous êtes libre de définir le comportement adéquat. On considérera comme comportement adéquat toute solution qui ne viole pas les contraintes données et qui ne résulte pas en un crash du programme.

Instructions :

1. Cet exercice doit être réalisé **individuellement !** L'échange de code relatif à cet exercice noté est **strictement interdit !** Cela inclut la diffusion de code sur le forum ou sur tout site de partage.

   Le code rendu doit être le résultat de *votre propre production.* Le plagiat de code, de quelque façon que de soit et quelle qu'en soit la source sera considéré comme de la tricherie (et peut même, suivant le cas, être illégal et passible de poursuites pénales).

   En cas de tricherie, vous recevrez la note « NA » pour l'entièreté de la branche et serez de plus dénoncés et punis suivant l'ordonnance sur la discipline.

2. Vous devez donc également garder le code produit pour cet exercice dans un endroit à l'accès strictement personnel.

   Le fichier (source !) `intervalles.c` à fournir comme rendu de ce devoir ne devra plus être modifié après la date et heure limite de rendu.

3. Veillez à rendre du code *anonyme* : **pas** de nom, ni de numéro SCIPER, ni aucun identifiant personnel d'aucune sorte !

4. Utilisez le site Moodle du cours pour rendre votre exercice.

   Vous avez jusqu'au 27 mars, 23:59 (heure du site Moodle du cours faisant foi) pour soumettre votre rendu. **Aucun délai supplémentaire ne sera accordé.**

Je tiens également à souligner que l'objectif de ce devoir n'est pas tant de vous évaluer que de vous entraîner. C'est pour cela que le coefficient de ce devoir est assez faible (10%). C'est également dans cet esprit qu'il a été conçu : de façon très progressive (de plus en plus difficile). Utilisez le donc comme un entraînement indicatif (sur lequel vous aurez un retour) en travaillant à votre niveau dans une durée qui vous semble raisonnable pour les crédits de ce cours (au lieu d'essayer de tout faire dans un temps déraisonnable). Ce coefficient (1 sur 10) n'est pas représentatif de la charge de travail (qui peut beaucoup varier à ce niveau d'un(e) étudiant(e) à l'autre), mais bien pour vous donner l'occasion d'avoir un premier retour sur votre niveau de programmation en C, sans prendre trop de « risques ».

II. Cadre général

Le but du programme considéré dans ce devoir 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).

Comme nous n'avons pas encore vu en cours l'allocation dynamique, une réunion d'intervalles devra pouvoir contenir *au maximum* `MAX_INTERVALS` (fourni) intervalles. Une telle une réunion d'intervalles aura par contre connaissance du nombre exact d'intervalles qu'elle contient.

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 $[0, 4[$ ».

II.1 Structures de données [3 points]

Commencez par définir les structures de données suivantes :

- `Interval`, structure de données pour représenter un intervalle comme expliqué ci-dessus ;
- `Interval_union`, une structure de données pour stocker un ensemble (une *réunion*) d'intervalles. Comme expliqué plus haut, cette structure de donnée sera de taille maximale fixe (pas d'allocation dynamique dans ce devoir) et stockera également le nombre d'intervalles effectivement utilisés (parmi le

max possible). Par convention, les intervalles effectivement utilisés seront stockés à partir du début (indice 0) et sans « trou » (pas de saut dans les indices des intervalles utilisés). L'ordre des intervalles utilisé, par contre, n'importe pas.

II.2 Affichage [8 points]

Écrivez la fonction `print_interval()` d'affichage pour les intervalles. Cette fonction doit afficher :

- le mot « `VIDE` » si l'intervalle est vide ; on supposera exister (voir plus loin) une fonction `is_empty()` 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 (format : `%g`) ;
- soit `]`, soit `[` en fonction que la fin (borne supérieure) est incluse ou non.

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

Écrivez ensuite la fonction `print_union()` d'affichage pour les réunion d'intervalles. Cette fonction doit afficher:

- les mots « `REUNION VIDE` » si la réunion ne contient aucun intervalle ;
- chacun des intervalles, séparés par « `U` ».

Exemple d'affichage: `[0, 4[ U [5, 10]`. Notez qu'il n'y a pas de « `U` » tout seul tout devant, ni à la fin.

II.3 Fonction outil : égalité de `double` [4 points]

Comme il est déconseillé de tester directement l'égalité de deux `double`, nous vous demandons d'écrire une fonction `are_equal()` qui teste si deux `double` (pas d'intervalle ici !) sont égaux à une précision `DBL_EPSILON` (fournie) près, c.-à-d. si la valeur absolue de leur différence est inférieure à cette précision.

II.4 Opérations sur les intervalles [40 points]

On veut pouvoir effectuer les actions suivantes, détaillées ci-dessous :

- tester si un intervalle est vide (`is_empty()`) ;
- tester si un nombre donné est dans un intervalle (`contains()`) ;
- savoir si deux intervalles sont joignables en un seul (`can_be_joined()`) ;
- calculer l'intervalle correspondant à la fusion de deux intervalles joignables (`join()`) ;
- supprimer un intervalle d'une réunion d'intervalles (`remove_interval()`) ;

- (`add()`) ajouter un intervalle à une réunion d'intervalles, tout en garantissant la représentation non-redondante décrite en début d'exercice ;
- (`merge()`) faire la réunion de deux réunions d'intervalles, c.-à-d. ajouter une réunion d'intervalles à une autre réunion d'intervalles.

Les cinq premières fonctions ainsi que la dernière sont assez simples à écrire et font l'objet des sous-sections suivantes. La sixième fonction (`add()`), plus difficile, est traitée en dernier dans une section spécifique.

Note: pour plus de détails sur les arguments et valeur de retours des fonctions, voir également les commentaires dans le code C fourni.

II.4.1 `is_empty()` [3 points]

Commencez par coder la fonction `is_empty()` 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 et ne sont pas incluses (aucune des deux).

Notez donc bien qu'avec une telle *convention*, les intervalles $]4, 4]$, $[4, 4[$ et $[4, 4]$ sont en fait le même intervalle (réduit au seul point 4). L'intervalle $]4, 4[$ est, par contre, l'intervalle vide.

II.4.2 `contains()` [6 points]

Codez ensuite la fonction `contains()` permettant de tester si une valeur de type `double` est contenue dans un intervalle donné.

A noter qu'un intervalle vide ne contient de toutes façons rien.

II.4.3 `can_be_joined()` [4 points]

Continuez en codant la fonction `can_be_joined()` permettant de tester si deux intervalles on une intersection vide ou non (deux tels intervalles sont alors représentables en un seul intervalle). 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 $[1, 3[$).

II.4.4 `join()` [6 points]

Continuez en codant (ici) la fonction `join()` permettant de calculer l'union de deux intervalles supposés joignables (on ne le testera **pas** ici ; si ce n'est pas le cas, le comportement est indéfini (« *undefined behaviour* »)). Cette fonction retourne donc un `Interval`.

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 `fmax()` et `fmin()` existent dans la bibliothèque `math`.

II.4.5 `remove_interval()` [4 points]

Implémentez la fonction `remove_interval()` qui supprime le i-ième intervalle d'une réunion d'intervalles (en la modifiant, donc); et ne fait rien si i n'est pas valide.

Il faudra veiller à ce que la réunion d'intervalle reste conforme aux conventions de représentation: qu'il n'y ait pas de «$_{\text{trou}}$» (saut d'indice) dans la liste des intervalles stockés.

II.4.6 `merge()` [4 points]

En supposant qu'existe déjà la fonction `add()` permettant d'ajouter *un* intervalle à une réunion d'intervalles en la modifiant (voir section suivante), codez la fonction `merge()` permettant d'ajouter *une réunion d'intervalles* (deuxième argument) à une autre réunion d'intervalles (premier argument, modifié(e)).

II.5 Ajouter *un* intervalle [sur 14 points]

Pour finir, codez la fonction `add()` 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_intvs` représente la réunion d'intervalles $[-1, 2] \cup [3, 5] \cup [6, 8[$ et `intv` représente l'intervalle $[0, 4]$, l'appel `add(reu_intvs, intv)` devra modifier `reu_intvs` pour qu'il représente la réunion $[-1, 5] \cup [6, 8[$.

L'algorithme pourrait fonctionner comme suit sur cet exemple :

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

- $[-1, 5]$ 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`.

III. Conseils et tests

N'hésitez pas à créer d'autres fonctions utilitaires si nécessaire.

Tout votre code et toutes vos fonctions doivent être robustes tant que faire se peut (c.-à-d. sauf avis contraire).

Pensez à tester correctement chacune des fonctionnalités implémentées à **chaque** étape et **avant** de passer à l'étape suivante. Cela vous évitera bien des tracas.

Si vous avez `diff` (ou `colordiff`) d'installé(s), vous pouvez même comparer le résultat :

```
./intervalles > output.txt
diff -wB example1.out output.txt
```