Examen de Programmation C/Java

– Examen (**1**) –

Vendredi 15 décembre, 2017
Durée: 3 heures

Résumé

Ce sujet comprend deux parties, l'une à traiter en C et l'autre en Java. À l'issue de l'examen vous devrez envoyer votre travail par e-mail dans une archive contenant le code source des programmes à l'adresse `<emmanuel.fleury@u-bordeaux.fr>`.
L'archive devra avoir la forme suivante (remplacez les mots entre `<...>` par votre nom/prénom) :

```
<NOM>_<Prenom>-examen/
    +-- fairylights/
    +-- java/
```


1 Programmation C : Fairylights (12 points)

1.1 Mise en place du projet

Sachant qu'il n'y a aucune dépendance à une bibliothèque particulière, le but de cette section est de réaliser un build-system avec l'outil `make` qui permette de compiler l'exécutable demandé.

Questions

1. Mettre en place la structure complète des sources du programme comme suit :

```
NOM_Prenom-examen/
    +-- fairylights/
          +-- fairylights.c
          `-- Makefile
```

2. Créez le fichier `Makefile` et écrivez les cibles suivantes :
   - `all` : Lance la compilation de l'exécutable `fairylights` ;
   - `fairylights` : Compile le fichier source en un exécutable ;
   - `clean` : Nettoie le répertoire de tous les fichiers superflus et des fichiers crées par la compilation.
3. Au sein de `Makefile` utilisez (et positionnez) correctement les variables classiques, c'est à dire : `CFLAGS`, `CPPFLAGS` et `LDFLAGS`. Ainsi que la cible spéciale `.PHONY`.

1.2 Problème principal

Comme chaque année les fêtes approchent et l'atelier du Père Noël est en pleine effervescence, les elfes travaillent dur pour livrer les cadeaux à temps. Cependant, ils adorent parier sur tout et n'importe quoi pendant leurs pauses (ils jouent des friandises d'Halloween qu'ils ont pu conserver jusque là).

Dans leur salle de pause, trône un immense sapin orné d'une guirlande électrique de Noël. Cette guirlande a la propriété de s'allumer suivant un schéma différent chaque jour (un schéma correspond à la séquence de lampes qui sont soit allumées, soit éteintes).

Les elfes ont pris l'habitude de parier sur le schéma qui va sortir le jour suivant. Mais, pour simplifier, ils considèrent que le schéma est un nombre entier codé en binaire. Lorsque la lampe est éteinte, elle est

représentée par un '0' et lorsqu'elle est allumée par un '1'. Ils parient en donnant des entiers positifs. Et, ils gagnent si l'un d'eux est exactement le schéma qui sort le jour suivant.

Évidemment, très vite, le générateur pseudo-aléatoire de schémas est devenu crucial pour eux et, suite à des tricheries de la part du responsable du générateur pseudo-aléatoire l'année passée, l'elfe en chef a dû changer un peu les règles. À présent, ce sont deux elfes qui gèrent chacun une source pseudo-aléatoire qui produisent chacun un entier (`x` et `y`) chaque matin. Puis, on effectue un `&`-bit-à-bit des deux pour obtenir le schéma final qui sera appliqué à la guirlande.

Alabaster est un elfe que rien n'arrête. Il a réussi à soudoyer chacun des elfes qui gèrent les deux générateurs pseudo-aléatoires pour qu'ils lui donnent les intervalles `[0,X-1]` et `[0,Y-1]` sur lesquels seront choisis les deux nombres pseudo-aléatoires. Chaque jour, il reçoit donc deux bornes supérieures (`X` et `Y`) pour l'aider à faire ses paris. Sa stratégie consiste à parier sur tous les nombres inférieurs à un nombre `Z` (`[0,Z-1]`) qu'il choisi en fonction de `X` et `Y`. Alabaster cherche donc à évaluer le nombre de combinaisons qui peuvent le faire gagner en fonction de `X`, `Y` et `Z`.

— **Entrée** : La première ligne de l'entrée donne le nombre de cas de test, `T`. Puis, les `T` cas suivent. Chaque cas de test est une suite de trois nombres : `X`, `Y` et `Z` qui représentent les deux intervalles des générateurs pseudo-aléatoires et l'intervalle sur lequel à joué Alabaster.

— **Sortie** : Pour chaque cas, la sortie sera une ligne contenant `"Case #x: "` (`x` commence à 1) suivi par le nombre de paires possibles qui feront gagner Alabaster.

— **Limites** : Le nombre de cas de test (`T`) ne dépassera jamais 100. Ensuite, pour les petits jeux de données, on a $1 \leq$ `X`, `Y`, `Z` ≤ 1000. Et pour les grands jeux de données, on a $1 \leq$ `X`, `Y`, `Z` $\leq 10^9$.

Fichier d'entrée

```
5
3 4 2
4 5 2
7 8 5
45 56 35
103 143 88
```

Sortie sur `stdout`

```
Case #1: 10
Case #2: 16
Case #3: 52
Case #4: 2411
Case #5: 14377
```

Figure 1 – Exemple de fichier d'entrées/sorties.

Dans le premier cas, les 10 paires qui peuvent faire gagner Alabaster sont : $(0,0)$, $(0,1)$, $(0,2)$, $(0,3)$, $(1,0)$, $(1,1)$, $(1,2)$, $(1,3)$, $(2,0)$, $(2,1)$. Notez que $(0,1)$ et $(1,0)$ sont différents. Aussi, le couple $(2,2)$ peut sortir mais donnera un 2 qui est plus grand que 0 ou 1 (les nombres plus petits que 2).

Enfin, vous trouverez des fichiers d'exemples plus aboutis à l'URL suivante :

`http://www.labri.fr/~fleury/courses/programming/exam/`

Questions

1. Commencez par écrire la partie du programme qui ouvre un fichier pour récupérer les données du problème. On suppose que l'interface utilisateur du programme se comportera comme suit :

   ```
   $> ./fairylights
   fairylights: error: No input file given !
   $> ./fairylights fairylights-xsmall.in
   Case #1: XXX
   Case #2: XXX
   ...
   ```

   En cas d'absence du fichier, le programme doit terminer en renvoyant une erreur sur `stderr` et avec un code de retour valant `EXIT_FAILURE`.

Pour le reste du parsing, on supposera que nous avons toujours affaire à des fichiers “parfaits”, sans erreur de syntaxe, ni oubli de la part de l'utilisateur. Inutile, donc, de passer du temps à essayer d'être robuste lorsque vous parsez le contenu du fichier. Par contre, on s'attend à ce que vous détectiez l'absence du fichier, ou un problème lors de l'ouverture (en lecture) de celui-ci.

Il vous faut donc, dans l'ordre, vérifier la présence d'un argument sur la ligne de commande, ouvrir le fichier en lecture (et vérifier qu'il a bien été ouvert), récupérer le nombre de cas, puis faire en sorte de récupérer les éléments de chaque cas.

2. Commencez par vous attaquer aux petits jeux de données (`small`) qui sont faisables en force brute.

3. Les grands jeux de données (`large`) sont plus difficiles à calculer car ils ne sont pas abordables par force brute pure (à cause de la combinatoire qui est derrière). Il va falloir avoir recours à quelques astuces. Notamment, il y a un biais dans les choix de `X`, `Y` `Z` qui rend les choses bien plus faciles. Très souvent, dans les cas qui devraient être difficiles à calculer on a `Z > X` ou `Z > Y`, or dans ces cas le nombre de combinaisons possibles est donné par `X * Y`. Dans tous les autres cas, on peut recourir à la force brute en essayant d'éviter les cas qui seront trivialement faux (trouvez la formule pour cela).

   Programmez la résolution du problème en utilisant l'algorithme suggéré. La clarté du code, son efficacité ainsi que les commentaires que vous y mettrez seront aussi évalués.

2 Programmation Java (8 points)

Questions

1. Questions de cours (*Indication : on rendra ici un simple fichier ascii qui contiendra les réponses*) :

   (a). Dans un diagramme UML, quelle propriété importante est commune entre les *liens d'implémentation* (*implement links*) et les liens d'héritage (*inheritance links*) (outre le fait que leurs arcs se terminent graphiquement par des flèches triangulaires).

   (b). À propos de la distinction entre les *associations* (*own links*), donner un exemple d'agrégation et un exemple de composition (évidemment différents de ceux du cours), et les expliquer.

   (c). Expliquer brièvement et précisément l'interêt de typer par des interfaces les variables qui dénotent des objets, et non de les typer par des classes.

2. Voici les deux définitions de l'interface classique de Java qui permet de définir des types d'instances ordonnées (la première est celle de l'API pour Java < 5, et la seconde est celle Java ≥ 5) :

```java
public interface Comparable {
    public int compareTo(Object x);
}
```

Listing 1 : L'interface `Comparable` en Java < 5.

```java
public interface Comparable<T> {
    public int compareTo(T x);
}
```

Listing 2 : L'interface `Comparable` en Java ≥ 5.

Reconsidérer l'exercice 2 de la fiche 4 qui portait sur des formes géométriques (`Shape`) comparables, et dont le corrigé utilisait la seconde définition, i.e. `Comparable<T>`. Adapter ce corrigé – disponible sur le site du cours – en utilisant seulement la première définition, i.e. `Comparable`, et expliquer ses inconvénients par rapport à la version initiale (*Indication : on rendra ici le code source complet ; les explications demandées se trouveront en commentaires dans ce code source*).

3. (a). Concevoir une architecture orientée objet pour la simulation d'un système sécurisé de contrôle d'aiguillages associés à un ensemble de lignes de trains. Représenter votre conception au moyen d'un diagramme UML de classes simplifié ("*UML as sketch*") en utilisant le logiciel `dia` (*Indication : on rendra ici le diagramme* `dia`).

   (b). Mettre en œuvre de manière minimale votre conception ci-dessus. (*Indication : on rendra ici le répertoire du code source*).