Devoir #1 — INF600A (gr. 20)

Automne 2018

Date de remise électronique : lundi 15 octobre, 12h00 (midi)
Date de remise du listing papier : mardi 16 octobre, 13h30.
Aucune remise électronique ne sera acceptée après lundi 15 octobre, 12h00 (midi).
Note : La solution sera publiée sur la page Web du cours dans l'après-midi du 15 octobre!

Note : Le labo du jeudi 4 octobre sera consacré au devoir et c'est moi qui serai présent pour répondre à vos question, vous aider à déboguer...

Script `bash` pour gestion d'une cave à vins

Ce que vous devez faire

Pour ce travail, vous devez développer un script `bash`, `gv.sh`, qui permet de gérer une cave à vins.[1] Plus précisément, **un squelette pour ce script** vous est fourni **et vous devez le compléter**.

Divers autres fichiers vous sont aussi fournis. Pour obtenir le script et ces fichiers, exécutez la comande suivante:

```
git clone http://www.labunix.uqam.ca/~tremblay_gu/git/GestionVins.git
```

Note : La première chose à faire une fois ce dépôt cloné est d'exécuter la commande «`make perms`» pour assurer **que** `gv.sh` **devienne exécutable** :

```
$ cd GestionVins
$ make perms
```

La figure 1 présente la liste des commandes acceptées par le script, liste produite par la commande «`./gv.sh aide`».

Exemples d'exécution

La figure 2 présente quelques exemples d'exécution des fonctionnalités de base de ce script. Ces exemples d'exécution sont obtenus par des cibles `ex_*` du `makefile` décrit plus bas. Dans chaque cas, l'exemple d'exécution s'effectue à partir d'une base de données «nouvelle» («fraîche») conservée dans le dépôt par défaut, soit le fichier «`.vins.txt`». Ce dépôt contient la description de quatre (4) vins et est créé à partir d'un fichier texte **prédéfini** — le fichier `vins.txt.init`, qui ne doit jamais être modifié![2] C'est la cible `ex_init`, dont dépend chaque exemple, qui crée ce dépôt initial, assurant ainsi que chaque exemple s'exécute dans un contexte indépendant des autres exemples.

[1] Un script `bash` n'est pas l'outil qu'on utiliserait typiquement pour développer un tel programme. Par contre, comme vous le constaterez, c'est possible de le faire, et ce de façon relativement rapide et concise, en utilisant les outils de base d'Unix tels que `grep`, `sed`, `awk`, combinés par l'intermédiaire de pipelines. Une fois que vous aurez complété ce travail, essayez d'imaginer combien de temps (et de lignes de code) il vous aurait fallu pour développer un programme équivalent... en Java!?

[2] Vous pouvez consulter ce fichier pour voir le format attendu pour le «vrai» dépôt de données, un fichier texte avec séparateurs de type CSV = *Comma Separated Values*.

Notez que ces exemples sont là pour vous aider à *amorcer* le développement du script `gv.sh`, avec des cas simples... et sans erreur — *happy paths*! **Ces exemples ne sont pas des tests!** Pour connaitre la spécification à respecter pour chaque commande, vous devez consulter les en-têtes des fonctions associées aux diverses commandes dans le script `gv.sh` et examiner les diverses suites et cas de tests, que vous trouverez dans les fichiers `Tests/*_test.rb` (voir plus bas).

```
$ ./gv.sh aide
NOM
  ./gv.sh -- Script pour gestion d'une cave a vins

SYNOPSIS
  ./gv.sh [--depot=fich|-] commande [options-commande] [argument...]

COMMANDES
  aide          - Emet la liste des commandes
  ajouter       - Ajoute un vin
  init          - Cree une nouvelle base de donnees
                  (dans './.vins.txt' si --depot n'est pas specifie)
  lister        - Liste les vins selon differents formats
  noter         - Attribue une note et un commentaire a un vin
                  (qui n'a pas encore ete note)
  selectionner  - Selectionne les vins matchant divers motifs/criteres
  supprimer     - Supprime un vin (qui n'a pas encore ete note)
  trier         - Trie selon divers criteres
```

Figure 1: Liste des commandes telle qu'affichée par «`./gv.sh aide`».

```
$ make ex_lister
# Il devrait y avoir 4 vins: 1, 2, 4 et 5 -- avec notes et commentaires.
./gv.sh lister
  1 [rouge - 26.65$]: Chianti Classico 2015, Volpaia (10/06/18) => 4\
    {Fonce, dense, opaque. Aromes fruits noirs. Tannins charnus.}
  2 [rouge - 26.65$]: Chianti Classico 2014, Volpaia (10/06/18) =>  {}
  4 [blanc - 16.50$]: Alsace 2016, Pfaff (03/07/18) =>  {}
  5 [ rose - 18.50$]: Cotes de Provence 2017, Roseline (03/07/18) => 3\
    {Frais, leger.}
#
# Il devrait y avoir 4 vins: 1, 2, 4 et 5.
./gv.sh lister --court
  1 [26.65$]: Chianti Classico 2015, Volpaia
  2 [26.65$]: Chianti Classico 2014, Volpaia
  4 [16.50$]: Alsace 2016, Pfaff
  5 [18.50$]: Cotes de Provence 2017, Roseline

$ make ex_ajouter
./gv.sh ajouter Beaujolais 2017 Foillard 22.00
# Il devrait y avoir 5 vins, le 5e (no. 6) etant le Beaujolais
./gv.sh lister
  1 [rouge - 26.65$]: Chianti Classico 2015, Volpaia (10/06/18) => 4\
    {Fonce, dense, opaque. Aromes fruits noirs. Tannins charnus.}
  2 [rouge - 26.65$]: Chianti Classico 2014, Volpaia (10/06/18) =>  {}
  4 [blanc - 16.50$]: Alsace 2016, Pfaff (03/07/18) =>  {}
  5 [ rose - 18.50$]: Cotes de Provence 2017, Roseline (03/07/18) => 3\
    {Frais, leger.}
  6 [rouge - 22.00$]: Beaujolais 2017, Foillard (19/09/18) =>  {}

$ make ex_trier
# Les 4 vins devraient etre affiches en ordre inverse de numero.
./gv.sh trier --reverse
5:03/07/18:rose:Cotes de Provence:2017:Roseline:18.50:3:Frais, leger.
4:03/07/18:blanc:Alsace:2016:Pfaff:16.50::
2:10/06/18:rouge:Chianti Classico:2014:Volpaia:26.65::
1:10/06/18:rouge:Chianti Classico:2015:Volpaia:26.65:4:Fonce [...] charnus.
```

Figure 2: Exemples d'exécution illustrant quelques-unes des fonctionnalités de base. **Note :** Dans ces exemples, les sauts de ligne avec «\» ne sont pas significatifs ; ils ont été ajoutés uniquement pour des fins de mise en page dans le présent document. Idem pour «`[...]`».

Catégories de commandes

Les commandes (si on ignore `init`) se décomposent en trois grandes catégories :

1. Les commandes qui modifient la base de données : `ajouter`, `noter` et `supprimer`.

   Ces commandes lisent les données dans le fichier texte (le dépôt) qui représente la BD et modifient le fichier. De plus, lorsqu'aucune erreur n'est signalée, elles sont silencieuses, i.e., elles n'émettent rien sur `stdout`.

2. La commande `lister`, qui reçoit des enregistrements décrivant des vins et affiche sur `stdout` diverses informations et selon divers formats.

   Cette commande peut utiliser *i*) les données provenant directement du dépôt; ou *ii*) des données provenant de `stdin`, dans le même format que celui du dépôt. Elle émet ensuite sa sortie sur `stdout`.

3. Les commandes qui effectuent du filtrage ou du réordonnancement sur des enregistrements décrivant des vins : `selectionner` et `trier`.[3]

   Ces commandes peuvent utiliser *i*) les données provenant directement du dépôt; ou *ii*) des données provenant de `stdin`, dans le même format que celui du dépôt. Elles émettent ensuite les enregistrements sélectionnés ou réordonnés sur `stdout`.

Ces deux derniers types de commande, qui peuvent utiliser `stdin` comme source des enregistrements, peuvent être combinés **à l'aide de pipelines**, comme l'illustre les exemples présentés dans la figure 3. Notez que l'utilisation de `stdin` peut se faire avec l'option «`-depot=-`» ou plus simplement en indiquant «`-`» comme premier argument au script.

[3]Le résultat produit par la commande `trier` est, par défaut, trié en ordre de numéro des vins. Toutefois, de façon générale, rien n'assure que les enregistrements soient triés ainsi, même s'ils proviennent directement du dépôt.

```
$ ./gv.sh selectionner --bus
1:10/06/18:rouge:Chianti Classico:2015:Volpaia:26.65:4:Fonce...charnus.
5:03/07/18:rose:Cotes de Provence:2017:Roseline:18.50:3:Frais...leger.

$ ./gv.sh lister --court
  1 [26.65$]: Chianti Classico 2015, Volpaia
  2 [26.65$]: Chianti Classico 2014, Volpaia
  4 [16.50$]: Alsace 2016, Pfaff
  5 [18.50$]: Cotes de Provence 2017, Roseline

$ ./gv.sh selectionner --bus | ./gv.sh --depot=- lister --court
  1 [26.65$]: Chianti Classico 2015, Volpaia
  5 [18.50$]: Cotes de Provence 2017, Roseline

$ ./gv.sh selectionner --bus |
  ./gv.sh - trier --prix |
  ./gv.sh - lister --court
  5 [18.50$]: Cotes de Provence 2017, Roseline
  1 [26.65$]: Chianti Classico 2015, Volpaia
```

Figure 3: Exemples illustrant l'utilisation de pipelines de commandes. (Les sauts de ligne ne sont pas significatifs dans les commandes; ils sont présents uniquement pour la mise en forme.)

Tests d'acceptation

Commande	base	intermediaire	avance
`ajouter`	N	N, E	B
`init`	N	N, E	
`lister`	N	N, E	
`noter`	N	E	
`selectionner`	N	N,E	
`supprimer`	N	E	
`trier`	N	N, E	B

N	Cas «normaux»
E	Cas avec erreurs
B	Bonus

Tableau 1: Les différentes commandes et les types de tests pour ces commandes.

Le tableau 1 présente les diverses commandes et indique ce qui est testé dans chaque groupe de tests — voir la description des tests présentée plus bas.

Quelques remarques en lien avec les tests :

- Vous pouvez désactiver — *temporairement!* — un test dans un fichier `Tests/*_test.rb` en changeant l'identificateur «`it_`» par «`_it_`». Vous pouvez faire de même pour un groupe de tests en changeant l'identificateur «`describe`» par «`_describe`».
- Plusieurs des tests **intermédiaires** portent sur des cas d'erreurs. Pour plus de détails sur les erreurs et messages qui doivent être signalés — **sur `stderr` et non sur `stdout`** — voir les cas de test où le symbole `:intermediaire` apparaît et avec des appels à la méthode `genere_erreur`. Et voir aussi l'en-tête des diverses fonctions.
- Dans les appels à `genere_erreur`, l'argument entre les barres obliques spécifie une *expression régulière Ruby* qui doit être matchée par le message d'erreur (sur `stderr`). Cette vérification porte sur *quelques éléments clés* du message et la casse est ignorée; il y a donc une certaine «flexibilité» quant au contenu exact du message.
- Notez que si une commande reçoit des arguments en trop, alors aucune sortie (`stdout`) ne doit être générée ; seul le message d'erreur approprié doit être émis (`stderr`).
- **Note importante** : **N'utilisez pas d'accents dans vos messages,** car cela peut créer des problèmes de portabilité entre machines. En fait, n'utilisez pas d'accents non plus dans vos commentaires!

Ce qui vous est fourni

Les fichiers fournis sont les suivants :

1. `gv.sh` : Le script `bash` à compléter.

2. Un fichier `bd-vins.sh` contenant diverses constantes et fonctions liées à la représentation (format) de la base de données textuelle.

   Ces éléments vous sont fournis tout d'abord pour assurer que vous utilisiez tous la même (et bonne) représentation interne, tel que requis pour les tests. Ils pourront aussi vous être utiles pour faciliter l'écriture de votre code — et éviter, le plus possible, l'utilisation de «constantes magiques».

   Note importante : Si désiré, vous pouvez **ajouter** des éléments dans ce fichier ; par contre, **vous ne devez pas modifier** les éléments déjà présents!

3. Un répertoire `Tests` contenant divers fichiers définissant des **tests d'acceptation**.

   Ces tests sont écrits en `Ruby`, à l'aide de `minitest`. Le fichier `test_helper.rb` définit des méthodes auxiliaires alors que les fichiers `*_test.rb` définissent diverses suites de test — par ex., le fichier `ajouter_test.rb` contient la suite de tests pour la commande `ajouter`.

4. Un fichier `makefile` définissant diverses cibles :

   - `make` : Exécute la cible par défaut, spécifiée par la variable `WIP` — W*ork*-I*n*-P*rogress*. Initialement, cette cible par défaut est `wip_ex`, avec l'exemple pour la commande `lister` — `WIP=wip_ex` et `COMMANDE=lister`.
   - `make wip_ex` : Exécute `gv.sh` avec l'exemple pour la commande indiquée par la variable `COMMANDE` — dont la valeur initiale est `lister`.

     Les cibles suivantes sont aussi définies : `ex_ajouter`, `ex_lister`, `ex_noter`, `ex_supprimer`, `ex_selectionner`, `ex_trier`. D'autres cibles avec ces mêmes noms mais suffixés par un «+» pour des exemples plus avancés sont aussi définies : `ex_{ajouter,lister,selectionner,trier}+`.

     Vous pouvez (**devriez!**) modifier la variable `COMMANDE` au fur et à mesure où vous développez une nouvelle commande. De cette façon, il vous suffira d'exécuter «`make`» pour lancer l'exécution de l'exemple.

     Remarque : Tel qu'indiqué plus haut, ces cibles représentent des **exemples** d'exécution. Ces exemples ont pour but de vous aider lors de l'écriture initiale du script et *ne constituent pas des tests* — **il n'y a pas de vérification des résultats obtenus par rapport aux résultats attendus**. Les véritables tests sont plutôt définis par les cibles décrites au point suivant.

- `make wip_test` : Exécute les différents tests pour la commande indiquée par `COMMANDE`.

  Trois catégories — trois **niveaux** — de tests sont définies :

  - `base`
  - `intermediaire`
  - `avance`

  C'est la variable `NIVEAU` qui spécifie le niveau désiré. Tous les niveaux sont testés si la valeur de cette variable est «`tous`» ou si elle n'est pas définie.

- `make test` : Exécute l'ensemble des tests, donc pour toutes les commandes et pour le niveau par défaut spécifié dans le `makefile`.

  On peut aussi exécuter tous les tests pour tous les niveaux comme suit :

  ```
  make test NIVEAU=tous
  ```

  Ou encore exécuter un test pour une commande et un niveau spécifiques :

  ```
  make test_lister NIVEAU=base
  ```

- `make remise` : Remet la copie électronique du travail — **voir plus bas**.

Ce que vous devez remettre

Remise électronique

La remise électronique doit se faire au plus tard le lundi 15 octobre à 12h00 (midi).

Plus précisément, vous devez remettre une version électronique **de votre code** en exécutant la commande suivante sur `java.labunix.uqam.ca` et ce **à partir du répertoire** `GestionVins` (qui contient le script `gv.sh`, le répertoire `.git`, etc.) :

```
make remise
```

Important : Auparavant, **vous devez modifier la variable** `CODES_PERMANENTS` dans le `makefile` pour indiquer votre/vos code/s permanent/s (seul/équipe).

Remise papier

Vous devez remettre, **sous forme papier**, le listing de votre fichier `gv.sh`. Vous pourrez effectuer cette remise en main propre **au début du cours du 16 octobre**, juste avant l'examen, et ce même si la date limite officielle pour la remise électronique est le lundi 15 octobre à 12h00 (midi).[4]

Pour effectuer la remise de votre listing, utilisez **la page avec grille de correction** disponible à la fin du présent document.

[4] Si les versions papier et électronique diffèrent, **c'est la version électronique qui sera corrigée!**

Remarques sur la correction

- Si le fichier remis contient des erreurs de syntaxe, alors vous perdrez une très grande (la majeure!) partie des points — voir la dernière page pour la grille de correction.
- J'exécuterai votre script sur la machine `java.labunix.uqam.ca`. Si vous développez votre script sur votre machine personnelle, assurez-vous qu'il fonctionne correctement sur `java` avant de le remettre. **S'il ne fonctionne pas sur `java`, tant pis!**
- J'effectuerai la vérification du bon fonctionnement *avec (possiblement) d'autres tests* que ceux qui vous sont fournis — donc avec des **tests privés additionnels.**
- Une partie des points sera accordée pour la *qualité* et le style de votre code : présentation, clarté et simplicité, structure, choix des identificateurs, mise en page, etc. Deux principes de base : KISS (= simplicité) et **DRY** (= pas de code dupliqué).[5]
- Une partie des points sera aussi accordée pour l'utilisation de `git`![6]

  Donc, utilisez `git` pour gérer l'évolution de votre projet et éviter sa «régression». Notamment, lorsque vous avez réussi à faire fonctionner une partie de votre code — par exemple, lorsque l'exemple ou les tests d'une commande s'exécutent avec succès — faites un `commit`! Et j'examinerai l'historique des *commits* pour m'assurer que *vous avez utilisé* `git` *de façon appropriée* — donc il n'y a pas un seul «gros» `commit` fait juste avant la remise finale!

[5] De façon informelle = Approche «WTF/min» = plus je mets de rouge parce que je ne comprends pas le code ou parce que le code est mal écrit, plus la note est faible ☹

[6] Pour des informations sur `git` : `http://www.labunix.uqam.ca/~tremblay_gu/INF600A/Liens` `http://www.labunix.uqam.ca/~tremblay_gu/INF600A/Materiel/git.pdf`

Remarques et suggestions

- Pour simplifier le problème, vous pouvez supposer que les options des commandes sont fournies dans l'ordre indiqué dans la documentation de la fonction pour la commande — cf. fichier `gv.sh`. Toutefois, un **bonus** pourra être accordé si votre script peut traiter les options *dans n'importe quel ordre*.

- La commande Unix `sort` possède de nombreuses options. Deux qui pourraient vous être utiles sont les suivantes : «`--field-separator=`» et «`--key=`» : voir figure 4

```
$ cat foo.txt
abc,123,444
def,456,111
aaa,789,333

$ sort --field-separator=, --key=2,2 foo.txt
abc,123,444
def,456,111
aaa,789,333

$ sort -t, -k3,3 foo.txt
def,456,111
aaa,789,333
abc,123,444
```

Figure 4: Exemples illustrant les options `--field-separator=` (`-t`) et `--key=` (`k`) de la commande `sort`.

- La commande Unix `head -n`, comme vue dans les notes de cours, permet de sélectionner les `n` premières lignes d'un fichier. Il existe aussi une option «-c» qui permet d'obtenir les premiers **caractères**, par exemple :

  ```
  $ echo abcde | head -c1
  a$ echo abcde | head -c3
  abc$
  ```

- Voir la fonction `debug` qui vous est fournie. Voir aussi les options «`-x`» et «`-v`» de la commande `bash`.

- Quelques autres suggestions sur le style (et la propriété DRY) :
  - Pour respecter les principes KISS et DRY, **n'hésitez pas à introduire des fonctions auxiliaires** — un exemple : une fonction qui retourne l'enregistrement (le vin) associé à un numéro de vin!
  - Profitez du fait que de nombreux tests sont disponibles pour effectuer du **refactoring**, i.e., modifier votre code — lorsqu'il fonctionne! — pour en améliorer la qualité interne.
  - Une façon «élégante» de traiter/signaler les erreurs est d'utiliser, de façon uniforme, le patron de code suivant avec garde, où `erreur` est une fonction — qui vous est fournie, donc utilisez-la! — qui signale l'erreur puis termine l'exécution du script :

    ```
    [[ condition_erreur ]] && erreur "Message..."
    ```

    Quelques exemples :

    ```
    [[ -z $x ]] && erreur "x est vide"
    # A ce point-ci, on sait que $x n'est pas vide!

    [[ $z == 0 ]] && erreur "z est 0"
    # A ce point-ci, on sait que z != 0!
    ```

Informations complémentaires (26 septembre 2018)

- Le message affiché par la commande `aide` est le suivant :

  ```
  ./gv.sh [--depot=fich|-] commande [options-commande] [argument...]
  ```

  Ceci signifie que «`--depot`» est une option **globale**, donc pouvant apparaitre **avant** n'importe quelle commande. (On parle d'une «option globale» si cette option s'utilise avec n'importe quelle commande, par opposition à une «option locale» si cette option est spécifique à une commande.)

  Comme on peut spécifier cette option pour toutes les commandes, ce n'est pas la fonction associée à la commande qui fait le traitement de cette option, mais bien le **programme principal**. Plus précisément, il communique cette information aux fonctions par l'intermédiaire de la variable (globale) `le_depot`. C'est pourquoi `le_depot` n'est pas — et n'a pas besoin d'être — un argument des fonctions `ajouter`, `supprimer`, `noter`, `trier`, etc.

  Par contre, comme indiqué p. 4, le (pseudo-)dépôt «`-`», qui correspond à `stdin`, ne peut pas être utilisé avec certaines commandes, qui doivent donc faire les vérifications appropriées.

  Voir aussi plus bas (p. 15) pour une remarque concernant l'utilisation du nom de fichier «`-`» avec la commande `cat`.

- `ajouter appellation millesime nom prix`
  - L'`appellation` est une chaine, possiblement avec des blancs, mais sans le séparateur («`:`»).
  - Le `millesime` est l'année de production du vin, donc un nombre indiquant une année «valide», disons à 4 chiffres — 1998, 2010, 2016, etc.
  - Le `nom` du vin est une chaine, possiblement avec des blancs, sans «`:`».
  - Le `prix` est un montant, avevc deux (2) chiffres après le point — donc ce pourrait être plus de 100.00.

  Suite à l'ajout du vin dans le dépôt, la date d'achat est indiquée dans l'enregistrement. Cette date est sous la forme «`JJ/MM/AA`».

  Voir la commande Unix `date`, plus spécifiquement : `date "+%d/%m/%y"`.

- `selectionner --bus`

  Les vins qui ont été bus/notés, donc pour lesquels il y a une note et un commentaire.

  `selectionner --non-bus`

  Les vins n'ont pas encore été bus/notés, donc pour lesquels n'y a ni note, ni commentaire.

  Note : Un commentaire est une chaine de caractères quelconques mais, pour simplifier, sans le séparateur.

- `lister [--court|--long|--format=un_format]`

  `un_format` est une chaine dans laquelle des spécifications de format peuvent apparaitre. Ces spécifications sont les suivantes:

  ```
  %I => numero (ID)
  %D => date-achat
  %T => type
  %A => appellation
  %M => millesime
  %N => nom
  %P => prix
  %n => note
  %c => commentaire
  ```

  Cette chaine peut aussi contenir d'autres éléments, comme on peut le voir dans l'un des cas de tests, par exemple :

  ```
  --format="Note pour %I => %n"
  ```

  Le saut de ligne final est **implicite**. Si un saut de ligne apparait explicitement à la fin de la chaine de format, alors deux sauts de ligne sont émis, par exemple :

  ```
  $ ./gv.sh lister --format="%I => %N\n"
  1 => Volpaia

  2 => Volpaia

  4 => Pfaff

  5 => Roseline

  $
  ```

- `trier --cle=CLE`

  `CLE` est une lettre utilisable comme dans `format` pour la commande `lister` — mais sans «%»!

  Par exemple, les commandes suivantes sont équivalentes:

`trier --cle=I`	`trier --numero`	`trier`
`trier --cle=D`	`trier --date-achat`	
`trier --cle=M`	`trier --millesime`	

  Pour simplifier, vous pouvez supposer qu'il n'y a qu'une seule clé spécifiée, i.e., `CLE` est une (et une) seule lettre parmi celles des spécifications de format.

Information utiles au sujet de la commande `cat`

Le symbole «-» peut être utilisé comme nom de fichier pour `stdin`, par exemple:

```
$ cat foo.txt
123
***

$ cat bar.txt
xxx
---

 $ cat - <foo.txt
123
***

$ cat foo.txt | cat bar.txt -
xxx
---
123
***
```

Travail remis à Guy Tremblay

INF600A-20 : Devoir #1

Remise électronique (`oto`) : lundi 15 octobre, 12h00 (midi)

Remise listing papier (en classe) : mardi 16 octobre, 13h30

Ne pas mettre dans une enveloppe

Nom	
Prénom	
Code permanent	

Nom	
Prénom	
Code permanent	

	`base`	`intermediaire`	`avance`
Commande			
`ajouter`	/3	/8	/1
`init`	/6	/2	
`lister`	/2	/10	
`noter`	/1	/6	
`selectionner`	/4	/6	
`supprimer`	/1	/5	
`trier`	/4	/3	/4

Bon fonctionnement avec tests de base		/25
Bon fonctionnement avec tests intermédiaires		/15
Utilisation appropriée de `git`		/5
Qualité générale du code		/5
Bon fonctionnement avec tests avancés (**bonus**)		/5
Total		/50
Note globale		**/ 10**