Alexis Malzieu - 1917652
Nicole Joyal - 1794431

POLYTECHNIQUE MONTRÉAL
LE GÉNIE EN PREMIÈRE CLASSE

1. Git

E1.1 Mise en place de votre répertoire Git (3 pts)

Finalement, l'un des 2 coéquipiers doit exécuter la commande «git log » et:

a) copier la sortie provenant du terminal dans le rapport sous la question E1.1. [/2]
Vous devriez ainsi avoir toute l'information concernant les nouveaux fichiers ajoutés au répertoire. Si ce n'est pas le cas, demandez de l'aide au chargé de laboratoire. Répondez également à la question

```
commit df64716cde1fe01dfedd65dccee71ecfd0b703f5 (HEAD -> master,
origin/master, origin/HEAD)
Author: alexismalzieu <alex.malzieu@gmail.com>
Date:   Mon Jan 29 12:52:05 2018 -0500

    init

commit 1b2a6fdc629acea28d1672e5941d0912728eeb7f
Author: admin <gigl-technique@polymtl.ca>
Date:   Tue Sep 11 15:51:58 2012 -0400

    Commit administratif

commit 70996bca0add50159f527a3495c0e19d965081c1
Author: Luc Lalonde <Luc.Lalonde@polymtl.ca>
Date:   Tue Sep 11 15:31:01 2012 -0400

    Premier commit

commit df64716cde1fe01dfedd65dccee71ecfd0b703f5 (HEAD -> master,
origin/master, origin/HEAD)
Author: alexismalzieu <alex.malzieu@gmail.com>
Date:   Mon Jan 29 12:52:05 2018 -0500

    init

diff --git a/data/quantium_algo.txt b/data/quantium_algo.txt
new file mode 100755
index 0000000..1be9e84
--- /dev/null
+++ b/data/quantium_algo.txt
```

b) quelle est la différence entre les commandes «git log» et «git log -p» ? [/1]

git log : affiche tous les commits précédemment effectués à partir du plus récent.
git log -p : affiche tous les changements et modifications effectués lors des commits.

E1.2 Modification 1 (10 pts)

Les deux coéquipiers reçoivent la tâche de modifier trois fichiers du code source, soit les fichiers: «HashMap.h», «HashMap.cpp» et «main.cpp». Vous devez donc réaliser les étapes suivantes:

1. L'Équipier 1 dé-commente la signature de la méthode int compteur(const std::string& key) dans le fichier «HashMap.h»
2. Ensuite, il dé-commente cette méthode en bas du fichier «hashMap.cpp».
3. Compilez manuellement le programme pour vérifier si le programme compile bien : « g++ -o pari *.cpp ».
4. Suite à la modification, l'équipier 1 doit exécuter la commande «git status». Répondez par la suite aux questions suivantes:
    - Copiez et collez à votre rapport la sortie du terminal correspondante à l'exécution de la commande.[/3]

```
[nijoya@l4714-22 src]$ g++ -o pari *.cpp
[nijoya@l4714-22 src]$ git status
Sur la branche master
Votre branche est à jour avec 'origin/master'.
Modifications qui ne seront pas validées :
  (utilisez "git add <fichier>..." pour mettre à jour ce qui sera
validé)
  (utilisez "git checkout -- <fichier>..." pour annuler les
modifications dans la copie de travail)

    modifié :      HashMap.cpp
    modifié :      HashMap.h

Fichiers non suivis:
  (utilisez "git add <fichier>..." pour inclure dans ce qui sera
validé)

    HashMap.o
    pari

aucune modification n'a été ajoutée à la validation (utilisez "git add"
ou "git commit -a")
```

    - Pourquoi le nom de l'exécutable, HashMap.h et HashMap.cpp sont écris en rouge dans la console?
      Car ces fichiers ont été modifiés mais pas encore commit.

5. Pour terminer, l'équipier en question exécute les commandes: «git pull» ,«git add HashMap.h HashMap.cpp suivi de «git commit -m "Votre Message"» et «git push». À ce point, une nouvelle révision de «HashMap.h» et «HashMap.cpp» devrait être créée.

6. L'Équipier 2 doit par la suite, sous son répertoire et sans faire un «git pull», modifier le fichier
7. SomeKeyHash.cpp» dans la définition de la fonction «hash()». En particulier, il faut remplacer l'implémentation actuelle (qui toujours retourne 1) par l'implémentation djb2 de la page http://www.cse.yorku.ca/~oz/hash.html . Il faut quelques modifications pour que cela marche dans le contexte du logiciel pari. Compilez manuellement pour tester.
8. Afin de propager sa modification, l'Équipier 2 exécute les commandes: «git add» suivi de «git pull» «git commit -m "Votre Message"» et «git push» et une nouvelle révision de «SomeKeyHash.cpp» est ainsi créée. Répondez par la suite aux questions suivantes:

    - Copiez la sortie du terminal correspondante à l'exécution de la commande «git pull».[/3]

      ```
      Updating df64716..26c48ed
      Fast-forward
       src/HashMap.cpp |   3 +--
       src/HashMap.h   |   2 +-
       src/HashMap.o   | Bin 0 -> 49464 bytes
       src/pari        | Bin 0 -> 33328 bytes
       4 files changed, 2 insertions(+), 3 deletions(-)
       create mode 100644 src/HashMap.o
       create mode 100755 src/pari
      ```

    - Est-ce que git a détecté un conflit? Pourquoi (pas)? [/2]
      Non car les 2 équipiers n'ont pas travaillé sur les mêmes fichiers

    - Finalement, l'Équipier 2 doit exécuter la commande «git pull» (pour être certain que le «log» soit à jour) suivi de «git log --graph --decorate --all --oneline» et copier la sortie correspondante au rapport. S'il n'y pas de différence entre ce «log» et celui de E1.1, veuillez consulter le chargé de laboratoire. [/1]

      ```
      $ git log --graph --decorate --all --oneline
      * 26c48ed (HEAD -> master, origin/master, origin/HEAD)
      Decommente de la fonction compteur
      * df64716 init
      * 1b2a6fd Commit administratif
      * 70996bc Premier commit
      ```

Alexis Malzieu - 1917652
Nicole Joyal - 1794431

POLYTECHNIQUE MONTRÉAL
LE GÉNIE EN PREMIÈRE CLASSE

E1.3 Modification 2 (10 pts)

Vous êtes maintenant affectés à différentes tâches de modifications sur le fichier «main.cpp». Pour mieux avancer, les deux équipiers répartissent leur travail.

1. L'Équipier 1 et L'Équipier 2 prennent le soin de faire, en tant que bonne pratique, un «git pull» avant de débuter leurs travaux sur le fichier en question.

2. L'Équipier 1 doit modifier et sauvegarder le fichier «main.cpp» pour que la fonction main() prend comme argument le nom d'un fichier (un std::string) et peut imprimer chaque mot du fichier. Mettez cette implémentation avant la ligne « //utilisation normale », et gardez le reste de l'implémentation existante de la méthode main() comme tel. Regardez les articles suivants pour inspiration: http://www.cplusplus.com/articles/DEN36Up4/ et http://stackoverflow.com/questions/20372661/read-word-by-word-from-file-in-c .

3. Suite à la modification, l'équipier en question doit exécuter la commande «git status». Répondez par la suite aux questions suivantes:
4.
    - Copiez la sortie du terminal correspondante à l'exécution de la commande. [/2]

```
[nijoya@l4714-22 src]$ git status
Sur la branche master
Votre branche est à jour avec 'origin/master'.
Modifications qui ne seront pas validées :
  (utilisez "git add <fichier>..." pour mettre à jour ce qui
sera validé)
  (utilisez "git checkout -- <fichier>..." pour annuler les
modifications dans la copie de travail)

    modifié :      main.cpp
    modifié :      pari

Fichiers non suivis:
  (utilisez "git add <fichier>..." pour inclure dans ce qui sera
validé)

    main.o
    quantium_algo.txt

aucune modification n'a été ajoutée à la validation (utilisez
"git add" ou "git commit -a")
```

    - Compilez le code source et copiez la sortie du programme. [/1]

```
g++ -o pari *.cpp
\.pari quantium_algo.txt
<affiche tout le contenu du fichier txt>
```

5. En entretemps, l'Équipier 2 travaille aussi dans le fichier «main.cpp», mettant à jour les commentaires vides (/**/) au-dessus de la méthode main(). Ces commentaires doivent expliquer le but du logiciel pari.

5. Pour propager ses modifications, l'Équipier 1 exécute les mêmes actions que décrites précédemment (git add, git commit -m, git push) Après, pour propager les modifications, l'Équipier 2 exécute les mêmes actions que décrites précédemment Répondez par la suite aux questions suivantes:

```
$ git push
To https://githost.gi.polymtl.ca/git/log1000-77
 ! [rejected]        master -> master (fetch first)
error: failed to push some refs to
'https://githost.gi.polymtl.ca/git/log1000-77'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

- Copiez la sortie du terminal correspondante à l'exécution des commandes «git fetch ; git log --graph --decorate --all --oneline». [/2]

```
* b94c469 (HEAD -> master) commentaire du main
| * 278d8b1 (origin/master, origin/HEAD) Lecture du fichier dans
main()
|/
* 26c48ed Decommente de la fonction compteur
* df64716 init
* 1b2a6fd Commit administratif
* 70996bc Premier commit
```

- Est-ce qu'il y aura un conflit lors d'un merge? Pourquoi (pas)? Si nécessaire, utilisez « git diff ..origin/master» pour décider. [/2]

Oui il y a un conflit car les équipiers ont tous les deux travaillé sur le fichier main.cpp.

- Maintenant faites « git merge », et copiez la sortie dans le rapport. [/2]

```
$ git merge
git loAuto-merging src/main.cpp
CONFLICT (content): Merge conflict in src/main.cpp
Automatic merge failed; fix conflicts and then commit the result.
```

- Finalement, l'Équipier 2 doit exécuter la commande «git push» suivi de «git log » et copier la sortie correspondante au rapport . S'il n'y pas de différence entre ce «log» et celui de E1.2, veuillez consulter le chargé de laboratoire. [/1]

```
$ git log --graph --decorate --all
--oneline
*   d86e299 (HEAD -> master) Résolution du conflit
|\
| * 278d8b1 (origin/master, origin/HEAD) Lecture du fichier dans
main()
* | b94c469 commentaire du main
|/
* 26c48ed Decommente de la fonction compteur
* df64716 init
* 1b2a6fd Commit administratif
* 70996bc Premier commit
```

E1.4 Modification 3 (10 pts)

On est presque là. Les deux équipiers font individuellement la révision cruciale pour finir le pari. Malheureusement, ils ne savent pas ce que l'autre équipier est en train de changer, ce qui pourrait être risquant.

1. L'Équipier 1 et L'Équipier 2 prennent le soin de faire, en tant que bonne pratique, un «git pull» avant de débuter leurs travaux sur le fichier en question.

2. L'Équipier 1 entame la première tâche de déclarer un HashMap « map » au début de main() (la ligne sous la signature de la fonction) et d'utiliser la méthode compteur() de la classe HashMap dans la boucle "for" ajouté pendant E 1.3 2) pour compter le nombre d'occurrences de chaque mot. Ensuite, enlevez l'ancien code de la méthode main() à partir de « //utilisation normale » jusqu'à la fin de la méthode. Compilez le code source résultant.

3. Pour propager ses modifications, l'Équipier1 exécute les mêmes actions que décrites précédemment en E1.2 3).

4. L'Équipier 2 décide en entretemps de déclarer un HashMap « mymap » au début de main() et de le parcourir dans une boucle for juste après la ligne « //utilisation normale » pour trouver et imprimer le mot avec le nombre d'occurrences le plus élevé. Utilisez la méthode getKeys() pour obtenir toutes les clefs du HashMap (utilisez vector<string> comme si c'était un tableau, la taille est disponible via la méthode size()), puis utilisez la boucle pour trouver la clef avec la valeur la plus élevée. L'Équipier 2 assume que l'Équipier 1 est en train d'écrire le code source qui remplira « mymap ». Notez que l'Équipier 2 n'enlève pas l'ancien code de main().

5. Pour propager ses modifications, l'Équipier 2 exécute les mêmes actions que décrites précédemment.(git add, git...)

   a. Copiez la sortie du terminal correspondante à l'exécution de la commande «git pull». [/3]

```
remote: Counting objects: 9, done.
```

```
remote: Compressing objects: 100% (5/5),
done.
remote: Total 5 (delta 4), reused 0 (delta 0)
Unpacking objects: 100% (5/5), done.
From https://githost.gi.polymtl.ca/git/log1000-77
   d86e299..5412f41  master     -> origin/master
warning: Cannot merge binary files: src/pari (HEAD vs.
5412f41abc50be006f12de562981d044c7dab5a7)
Auto-merging src/pari
CONFLICT (content): Merge conflict in src/pari
Auto-merging src/main.cpp
CONFLICT (content): Merge conflict in src/main.cpp
Automatic merge failed; fix conflicts and then commit the result.
```

6. L'équipier 2 doit suivre les étapes suivante: [/5]

- **git branch nouvelle-branche** (cette commande c'est pour créer une nouvelle branche git autre que MASTER sur laquelle on se trouve actuellement).

- faire un **git checkout nouvelle-branche** (pour changer de branche, maintenant on va se trouver sur la branche nouvelle-branche, de ce fait toutes les modifications qui seront fait seront visible juste sur cette branche et pas sur la branche principale **master**).

- se rendre sur le dossier du git(log1000-XX/TP1 ce chemin est un exemple et peut être différent d'une équipe a l'autre) et créer un fichier texte (textSurMaBranche.txt)

- faire un **git (add textSurMaBranche.txt, commit -m "ajouter du fichier text sur la nouvelle branche")** (pour ajouter et propager le changement, l'ajout du fichier .txt" seulement sur la branche nouvelle-branche alors le changement n'est toujours pas visible sur la branche **master**)

- **git checkout master** (pour revenir a branche **master** et la on va découvrir que les changement ne sont pas visible par les utilisateurs du master même si c'est le même utilisateur qui a fait le changement sur la nouvelle branche il sera dans l'incapacité de voir les changements sur cette nouvelle branche).

- faire un **git merge nouvelle-branche** (on fait fusionner la branche créé avec la branche principale master pour propager les modifications précédemment faites sur la branche nouvelle-branche).

Utilitée de cette exercise :
https://www.occitech.fr/blog/2014/12/un-modele-de-branches-git-efficace/ 7. Décrire en quelques ligne que c'est il passé:

a. Est-ce que Git a détecté un conflit? Pourquoi (pas)? Utilisez "git log --graph --decorate --all --oneline" et "git diff ..origin/master" pour supporter votre explication. [/3]

Non car le fichier créé dans la nouvelle branche n'influe pas sur les autres fichiers du master. Il sera donc ajouté normalement au projet.

b. Dans le cas où vous rencontrez une situation conflictuelle, comment pensez-vous régler cette dernière si on veut que le logiciel résultant réussit à résoudre le pari avec l'étudiant? Est-ce que ça peut être fait automatiquement? Pourquoi (pas)? Si oui, résolvez le conflit. [/3]

Lors d'un conflit, c'est à nous de choisir la version du code que l'on veut garder.
On peut lister les conflits avec $ git diff.
Si ce sont des conflits de code, ce n'est pas automatisable.
Si ce sont des conflits de fichiers entiers, il peut être envisageable d'utiliser un .gitignore

c. Finalement, l'Équipier 2 doit exécuter la commande «git push» suivi de «git log -v» et copier la sortie correspondante au rapport. S'il n'y pas de différence entre ce «log» et celui de E1.3, veuillez consulter le chargé de laboratoire. [/1]

```
commit 69047d18cdc91e839d09d87c2e49b3db322ea80a (HEAD -> master, origin/master, origin/HEAD)
Merge: be9c8c3 a94b92b
Author: alexismalzieu <alex.malzieu@gmail.com>
Date:   Mon Jan 29 15:23:21 2018 -0500

    Merge branch 'nouvelle-branche'

commit a94b92be5aed9583bb36de1bb5070c91a11e6061 (nouvelle-branche)
Author: alexismalzieu <alex.malzieu@gmail.com>
Date:   Mon Jan 29 15:23:00 2018 -0500

    ajouter du fichier text sur la nouvelle branche
```

d. [POINT BONUS] Quel est le mot le plus populaire dans « quantium algo » ? La bonne réponse doit montrer la sortie de votre programme, incluant la fréquence du mot gagnant. [/1]

```
Mot avec le plus d'occurence : "the", 889 fois.
```

Alexis Malzieu - 1917652
Nicole Joyal - 1794431

2. Make

À ce stade du TP, vous avez acquis et validé vos connaissances sur la gestion de votre entrepôt de versions Git. Ayant eu des problèmes à manuellement compiler, générer et installer des programmes et des fichiers, vous voulez absolument un build system automatique et efficace. Pour faire le tout, vous devez utiliser un fichier Makefile.

Documentation

D'abord, suivez attentivement ce tutoriel sur la base et les méthodes d'optimisation d'un Makefile:

https://www.youtube.com/watch?v=0YpgXKkNs04&list=PLyexDug1zljFuP8tC-PfhMIXtTKu1vFd3

E2.1 Éléments de construction d'un exécutable (5 pts)

Avant même de rédiger votre Makefile, vous devez connaître l'ensemble des éléments nécessaires pour les deux phases de construction du logiciel: 1. compiler et 2. installer le système.

1. Pendant la compilation, l'exécutable que vous devez créer sera constitué de l'ensemble de code source, tenant compte des relations #include dans les fichiers code source. Le nom de l'exécutable est **pari.** Pour simplifier les choses, il vaut mieux sauvegarder tous les fichiers générés pendant la compilation dans un dossier séparé avec le nom **build/** (à côté des dossiers src/ et data/) **. On appellera cette phase « compile ».**

2. La deuxième phase du build (appelé « install ») crée un dossier **site/** , puis copie les fichiers générés par la compilation (de **build/**) ainsi que des fichiers .txt de **data/** (comme le livre « quantium_algo»[2]) vers le dossier **site/** .

Les deux équipiers veulent que le build soit automatisé complètement (sans activités manuelles à faire) et sera efficace, par exemple:

- Si on change le code source, les phases de compilation et installation devront être refaites pour que le système reste cohérent. Ce cas peut être testé avec la commande «touch src/HashMap.h; make» dans le terminal.

- Également, si on change un des fichiers .txt dans le dossier **data** , l'installation doit être refaite, mais sans recompilation du code source. Ce cas peut être testé avec la commande «touch data/quantium_algo.txt; make» dans le terminal.

Vous devez écrire de manière hiérarchique le graphe de dépendance des phases et des fichiers nécessaires pour exécuter le build comme décrit ci-dessus. Appelez la cible principale « all » (une convention populaire), c.-à-d. si on appelle « make » sans spécifier la cible, « all » sera choisie automatiquement. Écrivez votre réponse dans votre fichier de réponses. À noter que même si actuellement vous n'avez pas les fichiers de dépendance .o, vous devez tout de même planifier leur intégration dans le graphe. Des cibles "phony", s'il y a lieu, doivent aussi figurer dans le graphe. [/5]

Petit exemple de réponse:

hello:
-start.o
    -start.cpp
    -hello.h
-hello.o
-hello.cpp
-hello.h

Ce que l'on peut voir dans cet exemple est que l'exécutable hello est fait du code compilé de start.o et hello.o. Le code source start.cpp appelle des fonctions de hello.o, la raison pour laquelle main.o dépend de hello.h. Si deux cibles dépendent d'une même cible, chacune doit mentionner cette dépendance dans votre graphe textuel.

all:

-main.o	*-hashmap.o*	*-someKeyHash.o*
-main.cpp	*-hashmap.cpp*	*-someKeyHash.cpp*
-hashmap.h	*-hashmap.h*	*-someKeyHash.h*
	-someKeyHash.h	
	-hashNode.h	

E2.2 Création du Makefile et exécution du programme (15 pts)

Vous devez maintenant rédiger un vrai **Makefile** en fonction des dépendances de construction que vous avez énumérées à l'étape précédente. Ne vous attardez pas à optimiser le script du **Makefile** . Suivez les étapes suivantes (astuce: ajoutez une commande "echo [un mot identifiant le règle]" dans chaque liste de commandes pour indiquer si les commandes de ce règle ont été exécutées):

1. Faire une mise-à-jour de votre copie locale («git pull») du répertoire Git d'équipe à partir du terminal de l'un des équipiers.

Alexis Malzieu - 1917652
Nicole Joyal - 1794431

2. Créez et modifiez un fichier **Makefile** avec un éditeur de texte et sauvegardez-le sous le dossier TP1/ (**pas** dans src/). **N'oubliez pas les bonnes extensions de fichiers et d'utiliser le compilateur g++ .** [/7]

3. Exécutez la commande ***make*** .

```
alexismalzieu (master *) log1000-77
$ make
g++ -o build/main.o -c src/main.cpp
making main.o
g++ -o build/hashMap.o -c src/hashMap.cpp
making hashMap.o
g++ -o build/SomeKeyHash.o -c src/SomeKeyHash.cpp
making SomeKeyHash.o
g++ -o site/pari build/main.o build/hashMap.o build/SomeKeyHash.o
making pari
```

4. Simulez des changements d'un fichier avec les deux commandes «touch ...» mentionnées ci-dessus. Copiez les deux sorties dans le rapport. [/3]

```
alexismalzieu (master *) log1000-77
$ touch src/HashMap.h
alexismalzieu (master *) log1000-77
$ make
g++ -o build/main.o -c src/main.cpp
making main.o
g++ -o build/hashMap.o -c src/hashMap.cpp
making hashMap.o
g++ -o site/pari build/main.o build/hashMap.o build/SomeKeyHash.o
making pari
```

5. Faire «git add» du fichier **Makefile** , suivi d'un «git commit» et «git push» dans le répertoire Git.

Astuce:

Pour rendre le TP plus facile, ajoutez et complétez les deux cibles suivantes pour enlever les fichiers générés:

clean:
    #enlevez les fichiers générés

mrproper: clean
    #enlevez les dossiers générés

Alexis Malzieu - 1917652
Nicole Joyal - 1794431

6. Expliquer le rôle et le fonctionnement des symbols, variables et commandes présents dans le makefile suivant: [/5]

```
CC=gcc
CFLAGS=-I/usr/local/include -Wall -pipe
LDFLAGS=-lMesaGL -L/usr/local/lib
RM=/bin/rm
MAKE=/usr/bin/make
MAKEDEPEND=/usr/X11R6/bin/makedepend

SRC= a.c \
     b.c \
     c.c
OBJ=$(subst .c,.o,$(SRC))

SUBDIR= paf pof


.SUFFIXES: .c
.c.o:
        $(CC) -c $(CFLAGS) $<

all:
        for i in $(SUBDIRS); do (cd $$i; $(MAKE) all); done
        $(MAKE) monProgramme

monProgramme: $(OBJ)
        $(CC) -o $@ $(LDFLAGS) $^

clean:
        $(RM) -f $(OBJ) core *~
        for i in $(SUBDIRS); do (cd $$i; $(MAKE) clean); done

depend:
        $(MAKEDEPEND) -- $(CFLAGS) -- $(SRC)
```

Chaque variable peut être utilisée dans les lignes de commandes du makefile. Il suffit de les insérer comme ceci : $(nom_variable). Le champ de texte associé sera alors utilisé dans la commande. Les variables ci-dessus sont des variables à utilisation conventionnel

CC : compilateur utilisé (ici GCC)
CFLAGS : options de compilation
LDFLAGS : options de linking
MAKE : localise l'exécutable make dans le système (path). Utilisé pour faire un appel récursif de make dans le makefile.
MAKEDEPEND : path de makedepend dans le système. Outils pour gérer les dépendances en c.
SRC : contient tous les fichiers sources .c du projet
OBJ : contient tous les fichiers .o du projet (qui seront donc générés)
OBJ=$(subst .c, .o, $(SRC)) : Le nom des fichiers .o seront obtenu en prenant le nom des fichiers contenus dans SRC en substituant l'extension .c en .o
SUBDIR : ensemble des sous-fichiers du projet

Partie 2 : [/6]

Pour vous familiariser encore plus avec git , les commandes de compilation et pour se familiariser avec le code source du logiciel open source Ring[3] ,vous allez compiler une de ses parties en suivant les étapes ci-dessous :

Alexis Malzieu - 1917652
Nicole Joyal - 1794431

1. Sortez du dossier que vous avez utilisé comme clône
de votre répertoire git dans la partie 1, et créez un nouveau dossier (qui ne fera pas partie de votre propre répertoire) dans lequel vous allez exécuter cette commande : git clone https://gerrit-ring.savoirfairelinux.com/ring-daemon

Cela va vous permettre d'avoir une copie de la composante du logiciel ring sur laquelle vous allez travailler dans le TP2.

2. Pour cela, vous allez maintenant commencer les étapes de compilation :

cd contrib
mkdir native
cd native
../bootstrap
make

```
libs/libopendht_la-crypto.o
clang: warning: -ljsoncpp: 'linker' input unused [-Wunused-command-line-argument]
clang: warning: -lrestbed: 'linker' input unused [-Wunused-command-line-argument]
clang: warning: -ljsoncpp: 'linker' input unused [-Wunused-command-line-argument]
clang: warning: -lrestbed: 'linker' input unused [-Wunused-command-line-argument]
crypto.cpp:26:10: fatal error: 'nettle/gcm.h' file not found
#include <nettle/gcm.h>
         ^~~~~~~~~~~~~~
1 error generated.
make[2]: *** [libopendht_la-crypto.lo] Error 1
make[1]: *** [install-recursive] Error 1
make: *** [.opendht] Error 2
```

a. Maintenant exécutez la commande **«time make »** et insérez la capture de la sortie. [/2]

```
clang: warning: -ljsoncpp: 'linker' input unused [-Wunused-command-line-argument]
clang: warning: -lrestbed: 'linker' input unused [-Wunused-command-line-argument]
clang: warning: -ljsoncpp: 'linker' input unused [-Wunused-command-line-argument]
clang: warning: -lrestbed: 'linker' input unused [-Wunused-command-line-argument]
crypto.cpp:26:10: fatal error: 'nettle/gcm.h' file not found
#include <nettle/gcm.h>
         ^~~~~~~~~~~~~~
1 error generated.
make[2]: *** [libopendht_la-crypto.lo] Error 1
make[1]: *** [install-recursive] Error 1
make: *** [.opendht] Error 2

real    0m22.878s
user    0m12.802s
sys     0m6.479s
```

b. Changez le nom de l'attribut privé callIDSet_
partout dans les fichiers src/account.h et src/account.cpp. Puis, exécutez une deuxième fois la commande **«time make »** et insérez la capture de la sortie. [/2]

```
clang: warning: -ljsoncpp: 'linker' input unused [-Wunused-command-line-argument]
clang: warning: -lrestbed: 'linker' input unused [-Wunused-command-line-argument]
clang: warning: -ljsoncpp: 'linker' input unused [-Wunused-command-line-argument]
clang: warning: -lrestbed: 'linker' input unused [-Wunused-command-line-argument]
crypto.cpp:26:10: fatal error: 'nettle/gcm.h' file not found
#include <nettle/gcm.h>
         ^~~~~~~~~~~~~~
1 error generated.
make[2]: *** [libopendht_la-crypto.lo] Error 1
make[1]: *** [install-recursive] Error 1
make: *** [.opendht] Error 2

real    0m23.741s
user    0m12.926s
sys     0m7.028s
```

c. Est ce qu'il y a une différence de temps entre les deux exécutions de la commande? Pourquoi (pas)? [/2]

Il n'y a pas tellement de différence entre les 2 temps d'exécution.

- *Le fait qu'il y ait une légère différence s'explique par le fait que les fichiers ont été modifiés et donc que la compilation doit être refaite. En effet, à la question b. la commande make avait déjà été lancée juste avant sans aucune modification entre eux. Les 2 fichiers en question ne sont donc pas entrés en jeu.*
- *Le fait qu'il n'y ait pas de grosse différence est que la différence entre ces 2 exécutions vient uniquement de ces 2 fichiers.*