INF1600

Travail pratique 3

Lien C++ et Assembleur

Département de Génie Informatique et Génie Logiciel
Polytechnique Montréal

1 Introduction et sommaire

Ce travail pratique a pour but d'approfondir vos connaissances sur l'assembleur IA-32, toujours selon la syntaxe AT&T. Il sera question dans ce travail pratique d'écrire des programmes impliquant la notion de récursivité. Par la suite, vous implémenterez des méthodes de classes dont le comportement attendu vous est fourni sous forme de code C++. La dernière partie de ce travail pratique abordera finalement la notion de classe Parent et classe Enfant.

1.1 Remise

Voici les détails concernant la remise de ce travail pratique :

- Méthode : sur Moodle, une seule remise par équipe, incluant un **rapport PDF** et les **fichiers sources** que vous modifiez en un seul fichier compressé. Une pénalité de 0,5 pt sera appliquée si le rapport est remis sous un format différent ou si votre code ne figure pas dans des fichiers séparés (**un fichier assembleur par exercice**).
- Format du rapport PDF : Incluez une page titre où doivent figurer les noms et matricules des deux membres de l'équipe, votre groupe de laboratoire, le nom et le sigle du cours, la date de remise et le nom de l'École. Dans une seconde page, incluez le barème de la section 1.2. Finalement, pensez à incluez les réponses aux questions et des captures d'écran si requis.
- Format des fichiers sources : Modifiez les fichiers assembleurs demandés dans les dossiers désignés à chaque question, puis compresser le tout en un seul fichier source <.zip> que vous devez nommer comme suit :
  `<matricule1>-<matricule2>-<tp3>.<zip>`

Attention :

L'équipe de deux que vous avez formé pour le TP1 est la même pour ce TP et la suite des TPs de cette session.

1.2 Barème

Les travaux pratiques 1 à 5 sont notés sur 4 points chacun, pour un total de 20/20. Le TP3 est noté selon le barème suivant. Reproduisez ce tableau dans le document PDF que vous remettrez.

TP 3			**/4,00**
Suite de Syracuse			/1,00
	Q1	/0,40	
	Q2	/0,40	
	Q3	/0,20	
Quicksort			/1,50
	Q1	/1,50	
REER			/1,50
	Q1	/0,25	
	Q2	/0,25	
	Q3	/0,25	
	Q4	/0,25	
	Q5	/0,50	

2 Suite de Syracuse

Pour ce premier exercice, il sera question d'implémenter une fonction permettant de calculer la [suite de Syracuse] d'un nombre entier $N > 0$. Cette suite est définie de la manière suivante :

$$u_0 = N$$

Et pour tout entier naturel n: $u_{n+1} = \begin{cases} \frac{u_n}{2} \; si \; u_n \; est \; pair \\ 3u_n + 1 \; si \; u_n \; est \; impair \end{cases}$

Suite de Syracuse pour $N = 15$

u_0	u_1	u_2	u_3	u_4	u_5	u_6	u_7	u_8	u_9	u_{10}	u_{11}	u_{12}	u_{13}	u_{14}	u_{15}	u_{16}	u_{17}	u_{18}	u_{19}	u_{20}	
15	46	23	70	35	106	53	**160**	80	40	**20**	**10**	5	16	8	4	2	**1**	4	2	**1**	…

Suite de Syracuse pour N = 15, Wikimedia Commons

La conjecture de Syracuse affirme que pour tout entier $N > 0$, il existe un indice N tel que $u_n = 1$.

Vous devez vous arrêtez d'itérez lorsque $u_n = 1$. Dans l'exemple de la suite pour $N = 15$ ci-haut, vous vous arrêteriez à u_{17} (au premier cycle trivial).

Pour exécuter votre code assembleur, nous avons fait appel aux fonctions correspondantes dans le fichier `syracuse/main.c`. **Vous ne devez pas modifier ce fichier**. Pour lancer le programme, exécutez les commandes suivantes dans le terminal, en vous trouvant dans le répertoire courant de l'exercice :

```
$ make clean
$ make
$ ./syracuse
```

2.1 Partie 1 : Suite de Syracuse itérative

Q1/ À l'aide du jeu d'instructions IA-32, complétez le programme assembleur `syracuse_s_iter.s` qui doit calculer et retourner la suite de Syracuse de la valeur u_0 passée en paramètre, de façon **itérative**. La fonction correspondante en C se trouve dans le fichier `syracuse/main.c`. Vous pouvez vous inspirer de cette fonction.

Lorsque vous lancez le programme, la version en C fournie est exécutée en simultané avec votre implémentation en assembleur. Comparez les valeurs affichées dans le terminal pour vérifier le bon fonctionnement de votre implémentation.

2.2 Partie 2 : Suite de Syracuse récursive

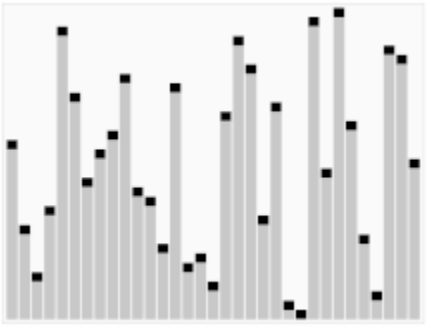
Q2/ Décommentez les lignes correspondantes aux appels des fonctions récursives dans le fichier `main.c` pour exécuter votre code et le code fourni.

À l'aide du jeu d'instructions IA-32, complétez le programme assembleur `syracuse_s_rec.s` qui doit calculer et retourner la suite de Syracuse de la valeur u_0 passée en paramètre de façon **récursive**. La fonction correspondante en C se trouve dans le fichier `syracuse/main.c`. **Une note de 0 sera attribuée à cette question si l'implémentation n'est pas récursive**. Comparez les valeurs affichées dans le terminal pour vérifier le bon fonctionnement de votre implémentation.

Q3/ Dans le fichier `syracuse/main.c`, remplacez la valeur de `u0` par `1000002743`. Quel comportement observez-vous ? Justifier le résultat produit par le programme, en vérifiant que le comportement est identique entre les implémentations en Assembleur et les implémentations en C. Quel est le nombre d'itérations produit ?

3 Quicksort

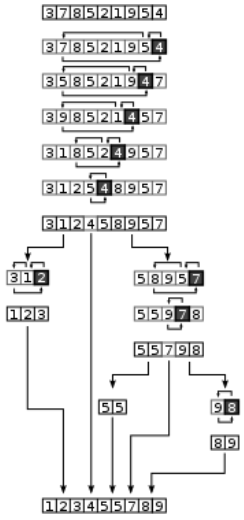
Quicksort est un algorithme de tri très commun développé par Tony Hoare en 1959 pour trier un tableau donné.



Visualisation animée de Quicksort, les lignes horizontales étant pivots, Wikimedia Commons

Quicksort est algorithme de type diviser pour régner. Il peut être implémenté de manière **récursive** pour un tableau **d'entiers** (int) comme suit :

1. Si la taille du tableau est inférieure à 2, nous retournons immédiatement. Aucune manipulation n'est à faire. Cette vérification est faite avec la constante `CUTOFF`.
2. Dans le cas contraire, un pivot doit être choisi. Pour notre algorithme, nous allons prendre la méthode de partitionnement *median of three*. Cette méthode est déjà implémentée pour vous.
3. Réordonner les éléments du tableau de façon à ce que ceux inférieurs au pivot se retrouvent à gauche de ce dernier, alors que ceux supérieurs se retrouvent à droite. Quant aux éléments égaux au pivot, le choix vous revient de les placer soit à gauche ou à droite.
4. Appliquer récursivement l'algorithme en divisant pour régner. Appliquer `quicksort` pour les éléments à gauche du pivot, donc de l'index 0 à l'index du $pivot - 1$. Appliquer également `quicksort` pour les éléments à droite du pivot, c'est-à-dire de l'index $pivot + 1$ à l'index *n*, soit la taille du tableau.



Exemple de quicksort pour un tableau de 9 éléments, Wikimedia Commons.

Les méthodes `final`, `printT`, `swapRefs` et `medianOfThree` sont déjà implémentées pour vous. Vous devez utiliser `medianOfThree` et `swapRefs` dans votre implémentation assembleur. Vous devez **uniquement** écrire l'équivalence assembleur de **`quicksort`**.

Pour exécuter votre code assembleur, nous avons fait appel aux fonctions correspondantes dans le fichier `quicksort/main.c`. **Vous ne devez pas modifier ce fichier**. Pour lancer le programme, exécutez les commandes suivantes dans le terminal, en vous trouvant dans le répertoire courant de l'exercice :

```
$ make clean
$ make
$ ./quicksort
```

Q1/ À l'aide du jeu d'instructions IA-32, complétez le programme assembleur `quicksort_s.s` qui implémenter **un comportement identique** à la fonction équivalente en C `quicksort` se trouvant dans le fichier `quicksort/main.c`. **Une note de 0 sera attribuée à cette question si l'implémentation n'est pas récursive**. Comparez les valeurs affichées dans le terminal pour vérifier le bon fonctionnement de votre implémentation.

4 REER[1]

Dans le cadre de cet exercice, il sera question d'implémenter des fonctions financières guidant votre retraite, notamment :

1. Le salaire final que vous allez gagner après 16 années de service au moment de prendre votre retraite. Ce montant est calculé de la façon suivante :

$$\boldsymbol{salaireFinal} = salaireDebut\left(1 + t_{\%augmentationSalariale}\right)^{anneesAvantRetraite-1}$$

2. Le montant que vous devez avoir accumulé au jour de la retraite si vous avez simplement besoin de 60 % de votre salaire final (montant calculé à la question 1) à chaque année de votre retraite. Ce montant est calculé de la façon suivante :

$$salaireRetraite = salaireFinal * t_{\%salaireVoulu}$$

$$\boldsymbol{montantAccumuler} = salaireRetraite\left(\frac{(1 + t_{\%intérêts})^{anneesDeRetraite} - 1}{t_{\%intérêts}(1 + t_{\%intérêts})^{anneesDeRetraite}}\right)$$

3. Le montant à épargner à chaque année de votre carrière pour réussir à amasser le montant à la question 2.

$$\boldsymbol{montantEpargne} = montantAccumuler\left(\frac{t_{\%intérêts}}{(1 + t_{\%intérêts})^{anneesAvantRetraite} - 1}\right)$$

4. Le montant à investir aujourd'hui pour obtenir le montant à la question 2 au lieu d'investir à chaque année.

$$\boldsymbol{montantInvesti} = montantAccumuler(1 + t_{\%intérêts})^{-anneesAvantRetraite}$$

Complétez l'exercice en sachant que vous êtes à 32 années de la retraite et que votre salaire de départ est de 79 520 $ (avec une augmentation annuelle de 4 %). Le taux d'intérêts annuel moyen est fixe à 8 %. Vous avez seulement besoin de 64 % de votre plus grand salaire réalisé pendant votre carrière à chaque année de votre retraite. Vous voulez vivre de vos rentes pendant 16 années.

5. Enfin, puisque le taux d'intérêts annuel moyen varie selon chaque type de compte REER, vous devez implémenter le taux d'intérêt le plus pessimiste possible en tenant compte de l'inflation. Cela dit, il ne s'agira plus d'un taux fixe à 8 %, mais plutôt de 4 %. Vous devez implémenter la fonction `montantAInvestirMaintenant` qui retourne le montant restant moyen de l'encaisse d'un client suite à l'ouverture d'un REER qui maximise l'investissement. Ce solde peut s'avérer négatif. Dans un tel cas, le client doit emprunter un tel montant envers des créanciers pour investir le tout en un seul versement. Nous estimons qu'un client moyen possède 50 000 $ en liquidités.

Pour exécuter votre code assembleur, nous avons fait appel aux fonctions correspondantes dans le fichier `main.c`. **Vous ne devez pas modifier ce fichier**.

Pour lancer le programme, exécutez les commandes suivantes dans le terminal, en vous trouvant dans le répertoire courant de cette première partie:

```
$ make clean
$ make
$ ./reer
```

[1] Exercice inspiré de Moulay Huard, 2022.

Q1/ À l'aide du jeu d'instructions IA-32, complétez le programme assembleur `salaireFinalAsm.s` qui doit implémenter **un comportement identique** à la fonction correspondante en C++ `Reer::salaireFinal()` se trouvant dans le fichier `reer/reer.cpp`.

Q2/ À l'aide du jeu d'instructions IA-32, complétez le programme assembleur `montantAmasseFinalAvantRetraiteAsm.s` qui doit implémenter **un comportement identique** à la fonction correspondante en C++ `Reer::montantAmasseFinalAvantRetraite()` se trouvant dans le fichier `reer/reer.cpp`. Votre programme doit appeler la méthode `salaireFinal()`.

Q3/ À l'aide du jeu d'instructions IA-32, complétez le programme assembleur `montantAEpargnerChaqueAnneeAsm.s` qui doit implémenter **un comportement identique** à la fonction correspondante en C++ `Reer::montantAEpargnerChaqueAnnee()` se trouvant dans le fichier `reer/reer.cpp`. Votre programme doit appeler la méthode `montantAmasseFinalAvantRetraite()`.

Q4/ À l'aide du jeu d'instructions IA-32, complétez le programme assembleur `montantAInvestirMaintenantAsm.s` qui doit implémenter **un comportement identique** à la fonction correspondante en C++ `Reer::montantAInvestirMaintenant()` se trouvant dans le fichier `reer/reer.cpp`. Votre programme doit appeler la méthode `montantAmasseFinalAvantRetraite()`.

Q5/ À l'aide du jeu d'instructions IA-32, complétez le programme assembleur `montantAInvestirMaintenantCompteAsm.s` qui doit implémenter **un comportement identique** à la fonction correspondante en C++ `Compte:: montantAInvestirMaintenant()` se trouvant dans le fichier `reer/compte.cpp`. Votre programme doit appeler la méthode `Reer:: salaireFinal()`.