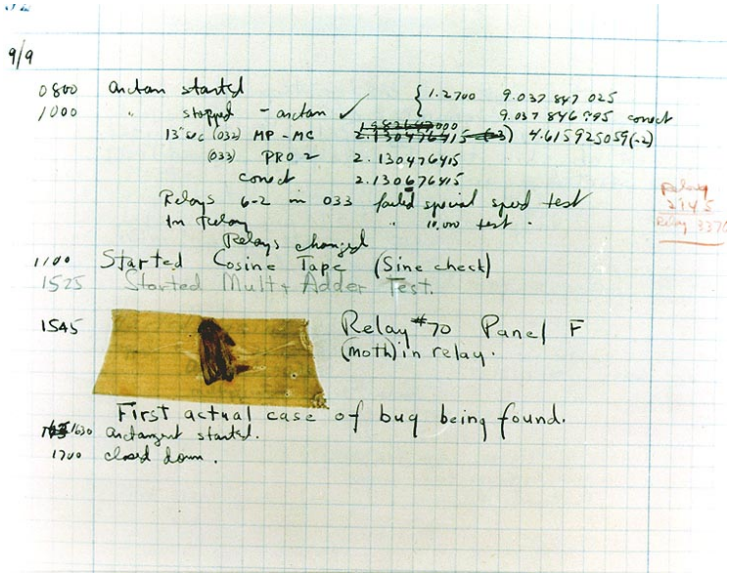
LABO 11 – DEBUGGING ET PROFILING
Informatique 1

Hes·so VALAIS WALLIS
Haute Ecole d'Ingénierie
Hochschule für Ingenieurwissenschaften

But du laboratoire (2 périodes)

1. Le but de ce laboratoire est de prendre en main des outils de *debug* et de *profiling*, deux outils importants pour la programmation.
2. Le *debugger* aide l'utilisateur à trouver les erreurs dans son programme (bugs) en lui permettant d'exécuter le programme pas à pas et de suivre en même temps la valeur de chaque variable. Le terme de bug – qui signifie insecte en anglais - pour désigner une erreur dans un programme est très ancien. Il trouverait ses sources (d'après la légende) dans la présence d'un insecte ayant causé un problème dans un calculateur Aiken Relay Mark II, à l'université de Harvard, le 9 septembre 1945.



3. Dans la deuxième partie du labo, vous allez réaliser une opération nommée *profiling* qui permet de mettre rendre visible le temps nécessaire à l'exécution des différentes parties d'un programme. Cet outil aide ainsi à optimiser un programme en pointant l'endroit où il consomme le plus de ressources.
4. La durée estimée pour réaliser ce laboratoire est de **deux périodes**. Si le temps imparti ne devait pas suffire, vous êtes invités à terminer le travail en dehors des heures de cours.
5. Vous pouvez trouver cette donnée sous forme électronique sur le site web du cours http://inf1.begincoding.net. Vous y trouverez également le corrigé de ce labo la semaine prochaine.

Partie 1 – Outils de debug

Il existe deux types d'erreurs en programmation. Les premières erreurs sont celles qui sont décelées par le compilateur. Même si elles ne sont pas toujours faciles à résoudre, leur détection, ainsi que leur localisation est grandement facilitée par l'outil d'environnement de développement. Le deuxième type d'erreurs correspond à celles qui apparaissent lors de l'exécution du programme. Il est généralement plus difficile de remédier à ces erreurs, car par exemple, elles peuvent ne pas se répéter. La plupart des environnements de programmation offrent cependant des outils de *debug*. Par exemple, *Eclipse* offre une perspective *Debug* qui permet d'accéder facilement à toutes les informations utiles aidant au suivi du programme et à la résolution des bugs.

Pour ouvrir cette perspective, on peut passer par le menu *Window* et choisir la perspective *Debug*. Normalement, cette perspective est automatiquement ouverte lorsqu'un programme est lancé en mode *Debug*. Pour lancer un programme en mode debug, il suffit de sélectionner *Debug as* dans le menu *Run*. Ce mode permet d'exécuter le programme pas à pas (*step over*) et d'entrer dans les méthodes (*step into*). En parallèle, il permet aussi de suivre les variables existantes à un moment du programme, ainsi que leur valeur.

Pour finir, la dernière notion importante est le *breakpoint*. Un *breakpoint* est un endroit où l'exécution du programme va être arrêtée pour pouvoir la suivre pas-à-pas. Les *breakpoints* peuvent être inclus dans le code de multiples façon, en cliquant dans la marge du code, ou en allant dans le menu *Run* et en sélectionnant *Toggle breakpoint*. Il est également possible de spécifier des breakpoints conditionnels où votre programme ne s'arrêtera que si certaines conditions sont remplies.

Tâche 1

1) Démarrez *Eclipse* et créez un nouveau projet.
2) Téléchargez les données du laboratoire depuis le site du cours et copier les fichiers source dans le répertoire *src* du projet que vous venez de créer.
3) Exécutez *NumberAnalyzer.* La méthode *main* de cette classe crée tout d'abord un tableau de nombres aléatoires et cherche ensuite la position du premier élément dupliqué.
4) Exécutez plusieurs fois le programme et vérifiez son résultat. Exécutez maintenant le programme en mode *Debug* (en mettant un breakpoint au début du main en faisant un clic sur la barre à côté des numéros de ligne) et suivez les variables au fur et à mesure des instructions exécutées. Vous pouvez passer à la ligne suivante à l'aide de la touche F6 ou de l'icône 
5) Essayez maintenant d'introduire un breakpoint conditionnel à la ligne 28 de manière à ce que celui-ci ne se déclenche qu'à la dixième itération.
6) Essayez d'ajouter d'autres breakpoints avec d'autres conditions.
7) Modifiez maintenant les paramètres utilisés lors de la création du tableau. Créez maintenant un tableau de 10 éléments contenant des nombres de 0 à 50.
8) Exécuter plusieurs fois le programme. Que se passe-t-il ?
9) En vous aidant du debugger, suivez le déroulement du programme et trouvez l'erreur contenue dans ce programme.

Partie 2 – Profilage du code

Le but du profilage est d'identifier les ressources (processeur ou mémoire) nécessaires à la réalisation des instructions d'un programme. Cela permet de cibler les régions du code les plus actives et de se focaliser ainsi sur ces régions pour optimiser le code. Il existe de multiples outils gratuits de profiling permettant d'automatiser cela mais, dans ce labo, vous allez utiliser une méthode simple de mesure de temps afin de mesurer le temps total passé dans les différentes méthodes. Ce résultat est très intéressant pour se faire une bonne idée des instruments consommant du temps dans un programme.

La deuxième partie du TP va se focaliser sur les comparaisons entre trois structures de données : les tableaux statiques, les tableaux dynamiques (*vector*) et les listes chaînées (*linked list*). Une grande partie du code a été déjà réalisée et se situe dans cinq fichiers différents : `Launcher.java` qui contient le `main` de votre pogramme, `DynamicStructureComparison.java` qui contient les tests de comparaison des structures de données et pour finir les classes `SampleArray`, `SampleVector` et `SampleLinkedList`. Ces trois dernières classes construisent une structure de données à partir d'un tableau et contiennent deux méthodes : `increment` et `insertZero`, qui permettent de tester respectivement l'accès aux éléments de la structure, ainsi que l'insertion d'un élément au milieu de la structure.

Tâche 2

1) La classe `SampleArray` est complète. Complétez maintenant les classes `SampleVector` et `SampleLinkedList` en implémentant le constructeur ainsi que les méthodes `increment` et `insertZero` pour les deux classes, afin qu'elles aient un comportement similaire à leur équivalent dans `SampleArray`.
2) Observez la classe `DynamicStructureComparison` et comprenez ce que font les méthodes `testInsertion` et `testAccess`. Observez aussi comment est utilisée la méthode `System.nanoTime()`.
3) Observez maintenant la classe `Launcher` et comprenez les différentes étapes du *main*.
4) Les paramètres que nous allons varier maintenant sont la taille du tableau aléatoire créé, la position d'accès ainsi que le nombre de répétition exécutées lors du test d'insertion et du test d'accès. Réalisez les différents tests décrits dans la table ci-dessous en modifiant les valeurs des différents paramètres et notez les résultats.
5) Faites le lien entre ces résultats et la théorie du cours sur les structures dynamiques. Pouvez-vous expliquer ce que vous voyez ?
6) Selon vous, quelle est la différence entre les listes chaînées offertes par Java et celles que vous avez créées dans le labo précédent (si vous l'avez déjà fait) ?

	1	2	3	4	5	6	7	8
Test	Accès	Accès	Accès	Accès	Insertion	Insertion	Insertion	Insertion
Nb Eléments	5'000'000	5'000'000	5'000'000	1'000	50'000	500'000	500'000	1'000
Position	0	2'500'000	4'999'999	500	0	0	499'999	500
Répétitions	100	100	100	10000	1'000	1'000	1'000	1'000
Temps avec array								
Temps avec Vector								
Temps Linked List								