TP2 : Performances d'un programme

1 Objectifs

Lorsque l'on doit produire un programme en entreprise, le produit désiré est décrit dans un document nommé *spécification*, lui-même obtenu à partir des *exigences* exprimées par le demandeur. Les exigences tout comme les spécifications sont séparées en deux types principaux : les exigences/spécifications fonctionnelles, et les exigences/spécifications non-fonctionnelles.

Les *exigences fonctionnelles* décrivent le comportement attendu du programme, généralement sous la forme de pré-conditions et de post-conditions et de *cas d'utilisation*. L'autre type d'exigence contient le plus souvent des exigences sur les *performances* du programme : temps d'exécution, espace mémoire ou espace disque utilisé.

Au cours de cette séance, nous allons tenter d'évaluer, de manière expérimentale, le coût en temps d'exécution de l'algorithme de tri par insertion (vu en cours).

2 Préambule : programme fourni

Le programme Ada `testtri.adb` contient :
— un préambule contenant la déclaration de l'ensemble des paquetages utiles pour la séance courante (clauses `with` et `use`) ;
— la déclaration de la constante `NMAX` (taille maximale du tableau), du type `Intervalle` (type des entiers du tableau) ;
— la déclaration du tableau à trier `A` ;
— la procédure `TriParInsertion`, contenant l'implémentation du tri par insertion ;
— la procédure `initialise_tableau`, à compléter, permettant d'initialiser le tableau `A` ;
— la procédure `testUnTri` initialise le tableau, puis le trie, en mesurant le temps d'exécution du tri. Ce temps d'exécution est ensuite affiché.
— le corps du programme, à compléter.

3 Remplir un tableau avec des valeurs aléatoires

3.1 Génération aléatoire et pseudo-aléatoire

En informatique, pour simuler le hasard, on utilise des générateurs de nombres pseudo-aléatoires : *Un générateur de nombres pseudo-aléatoires, pseudorandom number generator (PRNG) en anglais, est un algorithme qui génère une séquence de nombres présentant certaines propriétés du hasard. Par exemple, les nombres sont supposés être approximativement indépendants les uns des autres, et il est potentiellement difficile de repérer des groupes de nombres qui suivent une certaine règle (comportements de groupe)* (cf. wikipedia).

En pratique, les PRNG sont initalisée grâce à une *seed* (semence, graine). Toute la suite de nombres produite par le générateur découle de façon déterministe de la valeur de la graine.

3.2 Aspects pratiques Ada

Dans les sources qui vous sont fournies, le paquetage `NombreAleatoire` est une instance du paquetage générique `Ada.Numerics.Discrete_Random` pour l'intervalle d'entier `VALEUR_MAX` compris entre 1 et `NMAX`. L'initialisation de ce paquetage se fait à en appelant la procédure `NombreAleatoire.Reset(Semence)`. `Semence` est de type `Ada.Numerics.Discrete_Random.Generator` ; c'est un type qui cache l'état interne du générateur. Les valeurs de la suite de nombres pseudo-aléatoires dépendent donc de celui-ci.

L'obtention du prochain nombre aléatoire se fait en appelant la fonction `NombreAleatoire.Random(Semence)`.

Exercice 1

Dans le code qui vous est fourni, complétez la procédure `initialise_tableau` avec l'initialisation des `N` premières case du tableau avec des valeurs aléatoires.

4 Mesure expérimentale du temps d'exécution

Protocole expérimental

Comme pour toute mesure expérimentale, il faut préciser ce qu'on appelle le ***protocole*** de l'expérience. Ce protocole consiste à décrire précisément l'expérience qu'on va effectuer, en en fixant à l'avance tous les paramètres. Le choix des paramètres de l'expérience est effectué en fonction de l'objectif de l'expérience : il s'agit d'avoir la garantie que l'analyse des résultats sera pertinente, par rapport à l'objectif fixé. Ce n'est pas en général un problème simple, et il est courant qu'une expérience doive être renouvelée avec un nouveau protocole.

Ici, l'objectif annoncé de l'expérience est ***d'estimer le temps moyen d'exécution du tri par insertion***. Ce temps dépend de la taille `N` des données, mais aussi de leur valeur (en général, dans le cas d'un tri, de l'ordre initial de ces valeurs dans le tableau à trier). Dans un premier temps, on conserve l'hypothèse probabiliste de valeurs aléatoires, fixées par l'implémentation de `initialise_tableau`. Il restera donc à l'expérimentateur à **fixer les valeurs de** `N` pour lesquelles il décide d'estimer ce coût.

La manière d'estimer le coût moyen, pour une valeur de `N`, n'est pas totalement explicitée : la méthode standard [1] consiste à effectuer $X(\texttt{N})$ observations indépendantes, et à calculer la moyenne arithmétique des valeurs observées. Pour simplifier on choisira ici une valeur de $X(\texttt{N})$ **constante** (ne dépendant pas de `N`), que l'on notera `X` dans la suite.

Comment compléter le protocole de l'expérience, c'est-à-dire décider quelles valeurs de `N` considérer, et choisir la constante `X` ? Il est logique de penser que plus on considère de valeurs de `N`, et plus on fait d'observations pour chaque valeur, plus les résultats seront pertinents. Mais il y a le problème du coût de l'expérience : ici, la ressource ≪temps≫ est limitée, il faut donc faire un compromis. Une solution consiste à réaliser une ≪maquette≫ d'expérience, pour se fixer les idées aussi bien sur le temps nécessaire à effectuer les mesures que sur la variabilité des valeurs observées pour une valeur de `N` (par exemple en se limitant à quelques valeurs de `N` et en choisissant un `X` petit).

Exercice 2 : choix de `X`

Nous désirons maintenant lancer le tri $X(\texttt{N})$ **un nombre constant de fois X, sur un tableau à chaque fois rempli avec de nouvelles valeurs aléatoires**.

1. Modifiez la procedure `testUnTri` pour qu'elle effectue `X` fois l'initialisation (avec des valeurs aléatoires différentes !) et le tri. Dans un premier temps, on pourra prendre comme valeur $\texttt{X} = 10$.
2. Recompilez le programme, testez son exécution pour quelques valeurs de `N`.

Exercice 3 : faire varier `N`

Un autre paramètre important est `N`, soit la taille du tableau à trier, et la variation de cette taille entre `X` expériences pour un `N` donné. En effet, de trop petites valeurs de `N` feront que les temps obtenus seront trop proches de 0 pour permettre une quelconque interprétation. De même, en prenant différentes valeurs de `N` trop proches les unes des autres, la différence entre les temps obtenus risque de ne pas être assez significative.

1. Modifiez le programme principal pour qu'il effectue plusieurs mesures successives (une ≪mesure≫ consistant en un appel à `testUnTri`, c'est-à-dire `X` initialisations/tris du tableau), pour différentes valeurs de `N` prises dans un ensemble de valeurs prédéfinies.
2. Établissez l'ensemble de valeurs pertinentes pour `N`.

Exercice 4 : réaliser les mesures

1. Compilez puis exécuter votre programme. Notez le temps d'exécution moyen obtenu pour chaque valeur de `N`.
2. Reportez ces résultats dans un repère (en indiquant les valeur de `N` en abscisses et le temps moyen correspondant en ordonnées) et tracez l'allure de la courbe obtenue. De quelle façon cette courbe évolue-t-elle en fonction de `N` (est-elle monotone ? croissante ou décroissante ? linéaire ? quadratique ? ...) ?

1. Il existe d'autres méthodes que nous n'évoquerons pas ici.

5 Mesure par instrumentation du code source

Le temps d'exécution d'un programme est potentiellement dépendant de plusieurs paramètres :
— la configuration physique de la machine,
— le système d'exploitation et des différents paramètres possibles,
— l'overhead des tâches systèmes,
— la charge extérieure (celle de l'OS, des autres utilisateurs),
— des valeurs des éventuels paramètres du programme lui-même,
— des données d'entrées.

Les 3 premiers points (configuration, OS) sont difficiles à faire varier/à contrôler, car ils nécessitent d'avoir plusieurs machines ou de modifier l'OS. La charge extérieure peut être partiellement contrôlée, mais cela est difficile, par exemple dans des environnements partagés. De plus, si le temps d'exécution est très court, il est difficile de faire des mesures pertinentes, car le bruit induit par les autres paramètres peut être plus important que le temps d'exécution que l'on désire mesurer.

Le temps (physique) d'exécution n'est donc pas forcément une bonne mesure du coût en «temps» d'un programme. Une autre manière de mesurer ce «temps» est de compter les opérations (comparaisons, affectation, ...) que l'on fait dans le programme. On peut pour cela réaliser une *instrumentation du code*[2], c'est-à dire de le modifier afin d'obtenir, à l'exécution, des informations sur les aspects non-fonctionnels du programme.

Nous souhaitons maintenant connaître ***le nombre moyen de comparaison de valeurs*** faites au cours d'un tri du tableau, en fonction de la taille de ce tableau.

Exercice 5 : instrumentation du tri par insertion

Modifiez la procédure `TriParInsertion` afin de compter le nombre de comparaison effectuées au cours d'un tri. Modifiez `testUnTri` afin d'afficher ce nombre pour chaque valeur de `N` testée.

Exercice 6 : mesures par instrumentation

Reprenez l'exercice 4 avec le programme instrumenté.

6 Pour aller plus loin... (partie facultative)

Exercice 7 : utilisation d'un logiciel de tracé de courbes

On propose maintenant d'utiliser un logiciel de tracé de courbes, le logiciel `gnuplot`. On donne ci-dessous quelques indications pour utiliser ce logiciel, sachant que des informations complémentaires sont disponibles sur `http://www.gnuplot.info/help.html`.

— Format des données : les couples de coordonnées des points à afficher doivent être fournies dans un fichier texte, un couple par ligne, séparés par un (ou plusieurs) espace(s). Ces coordonnées peuvent être entières ou réelles. Pour produire un tel fichier une solution simple consiste à *rediriger* les sorties de votre programme : `./testtri > resultat`
Attention, *toutes* les sorties sont redirigées (y compris les messages destinés à l'utilisateur). Il faut donc modifier le programme pour que l'affichage ne comporte que les données à tracer.
— Le logiciel `gnuplot` se lance simplement avec la commande `gnuplot` qui a pour effet de démarrer une session interactive. La commande `quit` permettra de terminer cette session.
— Une fois une session interactive démarrée il est possible de tracer la courbe correspondant aux points fournis dans le fichier `resultat` :

```
gnuplot> plot "resultat" with lines
```

On peut également afficher une seconde courbe dans le même repère, par exemple :

```
gnuplot> replot log(x)
```

— Enfin, il est possible de diriger la sortie de `gnuplot` dans un fichier PNG :

```
gnuplot> set term png
gnuplot> set out "resultat.png" ; plot "resultat" with lines
```

2. il s'agit de la méthode la plus simple. Des outils tels que *valgrind* (dédié au *profilage* d'un programme en terme de mémoire), et des *debuggers* tels que *gdb* permettent aussi dans une certaine mesure de compter le nombre d'opérations.

Exercice 8 : hypothèse sur les données

On peut refaire les mesures des exercices 4 et 6 en changeant l'hypothèse sur les données du tableau. Par exemple, plutôt que des données aléatoires, on peut intialiser le tableau :

— avec des données constantes, ou prises dans un intervalle petit par rapport à la taille du tableau ;
— avec des données déjà triées ;
— avec des données «presque» déjà triées…