INF4100
Laboratoire no. 3

30/01/08

(Diviser-pour-régner, `mpd`, temps d'exécution et opérations barométriques)

1. Utilisation de `mpd`, temps d'exécution, etc.

Le fichier `sommes.mpd` contient un programme MPD qui définit trois versions d'une fonction pour calculer la somme des entiers compris entre 1 et `n`. Ces fonctions calculent donc, de façon explicite, la valeur de $\sum_{i=1}^{\mathtt{n}} i$.

Ces trois versions sont les suivantes :

- `somme0` : Fonction itérative.
- `somme1` : Fonction récursive avec *récursion de queue*.
- `somme2` : Fonction récursive avec *récursion dichotomique* équilibrée — décomposition en deux sous-problèmes de taille équivalente.

Le programme comporte trois arguments et produit comme résultat la somme calculée, *ainsi* que le temps d'exécution requis pour exécuter le programme. Les trois arguments sont les suivants :

- `n` : Valeur utilisée comme argument pour la fonction de sommation — le résultat devrait donc toujours être $\frac{\mathtt{n}(\mathtt{n}+1)}{2}$.
- `aExecuter` : Numéro de la fonction qu'on désire exécuter — donc 0, 1 ou 2.
- `nbExecutions` : Nombre de fois (consécutives) où la fonction sélectionnée doit être appelée avant d'imprimer le temps d'exécution — cet argument est nécessaire parce que le temps pour une seule exécution peut parfois être très petit.

Pour cet exercice, vous devez ...

a. Complétez le code MPD de la fonction auxiliaire `somme2Rec` appelée par la fonction `somme2`.

b. Comparez le temps d'exécution pour diverses valeurs des arguments du programme. Qu'en concluez-vous quant à la performance, et le bon fonctionnement, de ces diverses fonctions? (Essayez des petites *et* des grandes valeurs pour `n`!)

   Note : Le temps d'exécution est mesuré à l'aide de l'opération prédéfinie `age()`, qui retourne un entier indiquant, en millisecondes, le temps écoulé depuis que le programme a amorcé son exécution.

c. Choisissez une opération barométrique appropriée pour ces fonctions. Ensuite, modifiez le programme de façon *à compter et à imprimer, en plus du temps d'exécution*, le nombre d'opérations barométriques exécutées par chacune des fonctions. Qu'en concluez-vous?

Suggestion : Introduisez une variable *globale* (par ex., `nbOpsBaros`), que vous initialisez à 0 dans le corps du programme principal, et que vous incrémentez à chaque fois qu'une opération barométrique appropriée est exécutée.

Est-il possible de modifier quelque peu le programme de façon à ce que les trois fonctions génèrent *exactement le même nombre d'opérations barométriques*? Si oui, comment?

d. Modifiez le programme façon à introduire un *seuil* pour la version récursive dichotomique — c'est-à-dire que si le problème devient suffisamment simple, alors un calcul *itératif* (via un appel à la fonction itérative appropriée) est exécuté plutôt qu'un calcul récursif (plutôt qu'un autre appel récursif).

Quel est alors l'effet sur le temps d'exécution? Sur le nombre total d'opérations barométriques?

2. Procédure `fusionner`

a. Soit le tableau suivant :

```
int T[8] = (21, 20, 14, 3, 78, 82, 95, 60);
```

Donnez le contenu de `T` *après chaque appel de la procédure* `fusionner` suite à un appel «`trier( T, 8 )`» de la procédure de tri par fusion (Algorithme 2, p. 33 du cahier de notes de cours).

b. Expliquez ce que fait la procédure suivante — on suppose que `n` est une puissance de 2 (i.e., $n = 2^k$ pour un $k \geq 0$) et que `lg(n)` calcule le logarithme base 2 de `n` :

```
procedure proc( ref int A[*], int n )
{
  for [i = 1 to lg(n)] {
    int dist = 2**(i-1);
    for [j = 1 to n by 2*dist] {
      fusionner( A, j, j+dist-1, j+2*dist-1 );
    }
  }
}
```

Note : La clause `by v` indique le pas d'itération (*iteration step*) de la variable index de la boucle `for`. Par exemple :

```
for [i = 1 to 10 by 3] => i=1, i=4, i=7 et i=10

for [i = 1 to 10 by 4] => i=1, i=5 et i=9
```