Laboratoire 1 – Analyses asymptotiques et profilage d'algorithmes

ELE440 – Algorithmes (Automne 2017)

I. Objectifs

Ce laboratoire porte sur l'analyse asymptotique des algorithmes (théorique et expérimentale). Noter que l'ensemble du laboratoire se fait sur une plate-forme Linux. Les objectifs sont :

1. de comprendre et d'appliquer les méthodes d'analyse asymptotique de la performance des algorithmes ;
2. de se familiariser avec les outils de profilage d'algorithmes pour l'analyse de la performance des algorithmes dans la pratique.

II. Première partie (40 %)

Dans le laboratoire « introduction au système linux », nous avons vu comment observer un processus (un programme) qui s'exécute sur la machine à partir du terminal. Cependant, si l'on connaît l'algorithme d'un programme, il est possible de comprendre son fonctionnement. Dans cette première partie du laboratoire, il s'agit d'étudier un algorithme et de prédire son comportement. Vous allez devoir implémenter puis analyser un algorithme dont la fonctionnalité n'est pas connue à l'avance.

1. Implémentation

T1 – Travail d'implémentation: (5 %)

Écrire un programme en langage C qui implémente l'algorithme **Algo1**, dont le pseudo-code est fourni en annexe. L'algorithme prend en entré un tableau T1[1 … N] de taille N et retourne en sortie un tableau T2[1 … N] de la même taille.

T2 – Travail d'analyse : (15 %)

L'analyse de la performance de l'algorithme **Algo1** s'effectue de deux façons (théorique et expérimentale) et utilise le principe des baromètres dans les deux cas. Les indications ci-dessous servent à répondre aux questions de la section 2.

Analyse théorique :

À l'aide des techniques d'analyse récursive et de la technique des baromètres, déterminer l'ordre asymptotique de l'algorithme **Algo1**. Indiquer les lignes dans le pseudo-code où il faut placer les baromètres.

Analyse expérimentale :

Le but de l'analyse expérimentale est de mettre en évidence le lien qui existe ou non entre la performance (en termes de temps de calcul) de l'algorithme et les caractéristiques des données traitées. Les caractéristiques des données qui nous intéressent sont la taille **N** des tableaux, le rang **R** des données et le degré de désordre du tableau **D**. La performance est mesurée, quant à elle, en ajoutant des compteurs aux instructions identifiées comme « baromètres ».

La procédure de test suggérée s'exprime selon l'algorithme suivant :

Soit : $N = 1000 \times K$ avec $K = [10, 20, 30, ..., 100]$

$R = [10^2 , 10^4 , 10^6 , 10^8]$

$D = [0, 25, 50, 75, 100]$ %

Créer un fichier de résultats

```
1   Pour i de 1 à 10
2   n = N[i]
3       Pour j de 1 à 4
4           r = R[j]
5           Pour k de 1 à 5
6               d = D[k]
7               Pour m de 1 à 10 (nombre d'itérations de chaque test)
8                   T1 = Générer_données (n, r, d)
9                   (T2, Emesuré) = algo1(T1, n)
10              Sauvegarder (n, r, d et les 10 Emesuré) dans un fichier de résultats
11
12
```

Ainsi, l'algorithme est testé un total de 2000 fois.

Note : Les fonctions « Générer_données » et « Sauvegarder » sont fournies avec l'énoncé dans le fichier `resources.c`.

2. Questions (20 %)

Q1 – Que fait l'algorithme Algo1 ?

Q2 – Quel est l'ordre asymptotique théorique de Algo1 ? Les ordres Ω et O ainsi que l'ordre Θ (si possible) sont déterminés par la méthode d'analyse récursive de votre choix (arbre récursif, équation charactéristique, méthode générale). Pour y arriver, les temps de calcul minimum et maximum (E_{min} , E_{max}) de l'Algo2, ainsi que son ordre asymptotique, sont déterminés selon la méthode des baromètres. devez justifier vos calculs et résultats.

Q3 – Analyser les résultats de la série de tests à l'aide du programme de votre choix (ex.: Excel, MATLAB, OOCalc, ...) afin de déterminer la relation asymptotique entre la performance de l'algorithme et les caractéristiques N, R et D des données. L'analyse est effectuée de la façon suivante :

Tracer les 3 graphiques suivants : f versus N, f versus R, f versus D, où f = performance

- Sur chaque graphique, tracer les courbes Emin , Emax et Emoyen
- Pour chaque graphique, déterminer l'ordre asymptotique de Emin, Emax et Emoyen
- De ces résultats, tirer des conclusions quant à l'ordre asymptotique de l'algorithme

III. Deuxième partie (50 %)

Note : il faut suivre toutes les étapes de cette partie pour répondre aux questions de la section 5.

Une autre façon d'analyser la performance des algorithmes est d'utiliser un approche par **profilage**. Un profileur est un programme externe qui permet de collecter des informations sur un programme cible, lors de son exécution. Il existe deux types de profileurs :

- Le premier type analyse les performances du programme à partir d'un fichier exécutable. Ce type de profilage est limité puisqu'il ne permet pas de faire une analyse de la complexité algorithmique à partir du fichier binaire. Toutefois, il peut s'avérer intéressant d'avoir des information sur l'exécution d'un programme, comme la consommation de l'espace en mémoire ou l'utilisation du processeur.
- Le deuxième type de profileur permet d'effectuer une analyse du code source dans le but de fournir des statistiques sur le déroulement de l'algorithme. Il permet notamment de compter le nombre d'appels aux fonctions, l'espace mémoire utilisé par le programme et le temps de calcul de chaque fonction. Ces informations offrent plus de détails sur les performances d'un programme.

1. Introduction à gprof

Dans cette deuxième partie, on s'intéresse au deuxième types de profileurs. Par défaut, linux offre un profileur permettant d'analyser du code C/C++. Ce profileur doit être configuré (activé) lors de la compilation du code. Cette configuration permet d'ajouter des instructions spécifiques dans le code source du programme. Ces instructions permettent au programme de générer des données statistiques lors de son exécution.

Soit l'exemple écrit en code C et qui est fourni avec l'énoncé de ce laboratoire dans les fichiers `prof_main.c` et `prof_tache.c`. Ce code permet d'exécuter 3 fonctions : `fonction_1`, `fonction_2` et `fonction_3`. Chaque fonction appelle une ou plusieurs fois les tâches

`tache_A` et `tache_B` (voir Figure 1).

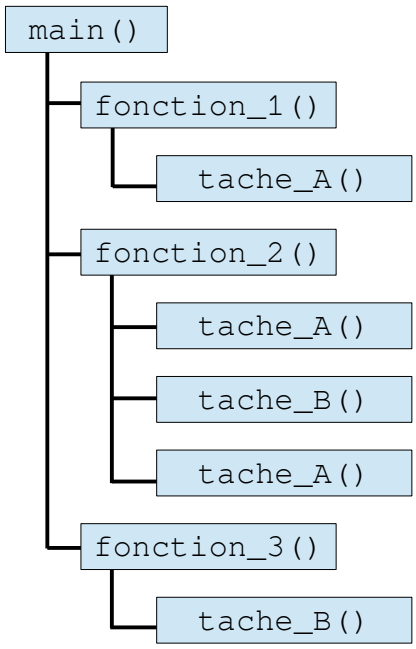

Figure 1 : Arborescence d'exécution du programme.

Chaque fonction ainsi que chaque tâche nécessitent un certain temps d'exécution. Afin d'identifier quelle partie du code est la plus coûteuse en temps de calcul, il est intéressant de profiler ce code. Le profilage se déroule en 3 étapes :

- Étape 1 : Compiler le programme avec l'option de profilage

  Afin de permettre au profileur **gprof** d'analyser un code, il est nécessaire de compiler le code avec l'option `-pg`. Cette option permet d'ajouter des lignes de code spécifiques afin d'enregistrer des statistiques utiles lors de son exécution (par ex. nombre d'appel aux fonctions, durée d'exécution des fonctions, etc.). La commande de compilation est :

  ```
  $ gcc -pg prof_main.c prof_taches.c -o programme
  ```

- Étape 2 : Exécuter le programme

  La deuxième étape consiste à exécuter le programme.

  ```
  $ ./programme
  ```

  À la fin de l'exécution, un fichier nommé `gmon.out` est automatiquement généré. Ce fichier contient les informations statistiques sur l'exécution du programme.

- Étape 3 : Générer le fichier des statistiques

Pour pouvoir interpréter le fichier des statistiques, il faut utiliser la commande `gprof`. Cette commande prend en entrée le programme à analyser `programme` et le fichier de statistiques généré `gmon.out`, et affiche à l'écran les informations sous un format compréhensible. La commande suivante permet de récupérer ces informations dans un fichier texte `analyse.txt`.

```
$ gprof ./programme gmon.out > analyse.txt
```

2. Interpréter le fichier de profilage de gprof (5 %)

Par défaut, gprof produit deux format de profilage : profilage global et le graphe des appels.

- Profilage global : le temps total d'exécution de chaque fonction du programme est résumé dans un premier tableau appelé *flat profile* (Figure 2). Chaque ligne du tableau désigne une fonction du programme (les lignes sont triées par ordre décroissant de temps d'exécution).

  Chaque colonne du tableau indique une statistique :

  - % : pourcentage du temps d'exécution total passé dans la fonction ;
  - cumulative (s) : cumul du temps d'exécution passé sur l'ensemble des fonctions précédentes dans le tableau ;
  - self (s) : temps d'exécution pour tout les appels de la fonction ;
  - calls : nombre d'appel à la fonction ;
  - self (s/call) : temps moyen d'exécution de la fonction pour chaque appel ;
  - total (s/call) : temps d'exécution total pour chaque appel de la fonction incluant ses descendants ;
  - name : nom de la fonction.

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 64.58      1.54     1.54        2     0.77     0.77  tache_B
 35.46      2.39     0.85        3     0.28     0.28  tache_A
  0.84      2.41     0.02        1     0.02     1.36  fonction_2
  0.00      2.41     0.00        1     0.00     0.28  fonction_1
  0.00      2.41     0.00        1     0.00     0.77  fonction_3
```

Figure 2. Exemple de tableau de profilage global généré par gprof.

- Graphe des appels : un deuxième tableau appelé *call graph* permet d'afficher le temps d'exécution de chaque fonction en incluant les fonctions *enfants*, appelées à partir de celle-

ci. Ces informations sont pratiques pour détecter le cas d'une fonction qui, elle-même n'est pas très lente, mais fait appelle à d'autres fonctions très coûteuses en temps de calcul (Figure 3). De la même façon que pour le profilage global, chaque colonne correspond à une statistique. Cependant, cette fois, chaque groupe de lignes séparées par les symboles « --- » désigne l'exécution d'une fonction qui est spécifiée par un index unique ([1], [2], etc). Les fonctions qui sont au dessus de cette fonction correspondent à ses parents (les fonctions qui ont fait appel à la fonction indexée) et les fonctions qui sont en dessous correspondent à ses enfants (les fonctions appelées par la fonction indexée).

```
index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.00    2.41                 main [1]
                0.02    1.34       1/1           fonction_2 [3]
                0.00    0.77       1/1           fonction_3 [5]
                0.00    0.28       1/1           fonction_1 [6]
-----------------------------------------------
                0.77    0.00       1/2           fonction_2 [3]
                0.77    0.00       1/2           fonction_3 [5]
[2]     64.0    1.54    0.00       2          tache_B [2]
-----------------------------------------------
                0.02    1.34       1/1           main [1]
[3]     56.3    0.02    1.34       1          fonction_2 [3]
                0.77    0.00       1/2           tache_B [2]
                0.56    0.00       2/3           tache_A [4]
-----------------------------------------------
                0.28    0.00       1/3           fonction_1 [6]
                0.56    0.00       2/3           fonction_2 [3]
[4]     35.1    0.85    0.00       3          tache_A [4]
-----------------------------------------------
                0.00    0.77       1/1           main [1]
[5]     32.0    0.00    0.77       1          fonction_3 [5]
                0.77    0.00       1/2           tache_B [2]
-----------------------------------------------
                0.00    0.28       1/1           main [1]
[6]     11.7    0.00    0.28       1          fonction_1 [6]
                0.28    0.00       1/3           tache_A [4]
-----------------------------------------------
```

Figure 3. Exemple de tableau de graphe des appels généré par gprof.

3. Visualiser le graphe des appels (5 %)

Le fichier `analyse.txt` contient beaucoup d'informations, mais il n'est pas très évident à visualiser. Il existe des outils de visualisation qui permettent de générer une figure à partir du graphe des appels. Par exemple, le programme `gprof2dot` permet de transformer un fichier de sortie `gprof` en une image. Pour cela, exécuter la commande suivante :

```
$ gprof2dot analyse.txt | dot -Tpng -o image.png
```

Une image `image.png` est générée (Figure 4). Elle permet de voir les appels entre les fonctions ainsi que le temps d'exécution total de chaque fonction.

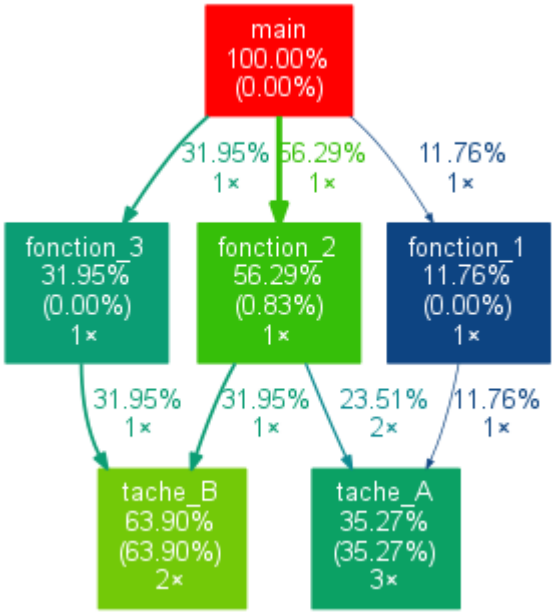

Figure 4. Illustration visuelle du graphe des appels : `image.png`

Notez que lorsque le temps d'exécution d'une fonction est trop petit, `gprof2dot` ne génère pas la représentation de cette fonction dans le graphique. Ceci, afin de n'afficher que les fonctions coûteuses en temps de calcul, et permet une meilleure visibilité. Il est possible de forcer l'affichage de toutes les fonctions avec les options `-n0` et `-e0`, la commande devient alors :

```
$ gprof2dot -n0 -e0  analyse.txt | dot -Tpng -o output.png
```

4. Analyse d'un code plus complexe (40 %)

Soit deux implémentations de l'algorithme de tri par base « Radix » donnée dans les fichiers `tri_radix_1.c` et `tri_radix_2.c`. À l'aide du profileur gprof, analyser le code des deux implémentations indépendamment et répondre aux questions.

5. Questions (travail à faire)

Q1 – Générer le fichier `analyse.txt` de profilage gprof pour le code source des fichiers `prof_main.c` et `prof_tache.c` et générer la figure du graphe des appels associé (Figure 4).

Q2 – Expliquer le principe de fonctionnement de l'algorithme de tri par base (Radix) en utilisant vos propres mots (0.5 à 1 page max).

Q3 – Analyser le code source des deux implémentations de l'algorithme de tri par base `tri_radix_1.c` et `tri_radix_2.c` et donner la principale différence entre les deux

implémentations.

Q4 – À l'aide de la commande `gprof`, générer les fichiers `analyse_radix_1.txt` et `analyse_radix_2.txt` des codes sources `tri_radix_1.c` et `tri_radix_2.c` respectivement, ainsi que les figures des graphes des appels associées (afficher toutes les fonctions sur les graphes des appels).

Q5 – Déterminer la (les) fonction(s) qui seraient coûteuses en terme de temps de calcul. Expliquer son (leurs) fonctionnement(s) (1 page max).

Q6 – Laquelle des deux implémentations est la plus efficace en terme de temps de calcul ? Justifier votre réponse.

Q7 – Est-il possible d'améliorer ces implémentation ? Si oui, comment ?

IV. Livrables

Rapport de laboratoire (12 %)

Le rapport est écrit dans vos mots et doit contenir les sections suivantes (les longueurs des sections sont suggérées surtout à titre indicatif; l'important est de fournir les informations demandées et de le faire de façon concise):

1. Introduction (maximum 1 page)

   Contient une description des buts du laboratoire et des objectifs visés. Sert à introduire les sections qui suivent.

2. Première partie

   Contient les réponses aux questions Q1 à Q3 de la première partie.

   - Pour l'analyse théorique (Q2) : fournir le détail de l'analyse théorique de l'algorithme et justifier les conclusions de l'analyse. Indiquer les baromètres retenus (et non ceux qui ont été éliminés). Il faut distinguer clairement les formules exactes (ex. nombre d'exécutions, E) et les formules asymptotiques.
   - Pour l'analyse expérimentale (Q3) : donner une courte description du protocole de test. fournir le détail de l'analyse des résultats des tests, comprenant les graphiques, l'interprétation des graphiques et les conclusions quant aux ordres asymptotiques de l'algorithme. Les graphiques doivent comporter des axes clairement étiquetés ainsi que des légendes si nécessaire.

3. Deuxième partie

   Contient la réponse aux questions Q1 à Q7. Indiquer pour chaque question la (les)

commande(s) utilisée(s) pour obtenir les résultats.

4. Conclusion (1 page max)

   Conclut quant à l'atteinte des objectifs de départ. Fait aussi une comparaison entre les résultats des analyses théorique et expérimentale ainsi que l'analyse par profilage.

5. Références

   Contient les références de vos sources. Dans le corps du rapport, vous devez également mettre un renvoi après chaque élément emprunté, ex. [1], [2]...

   Remarque: Si une information provient de l'énoncé de laboratoire ou du matériel de cours, il n'est pas nécessaire de citer cette référence.

   Si le document cité est un volume :

   1. De Garmo, E.P., Sullivan, W.G. & Bontadelli, J.A. (1989). Engineering Economy (8e ed.). New York : MacMillan.

   Si le document cité provient d'un site internet :

   2. École de technologie supérieure. Politique d'éthique de la recherche avec des êtres humains, [En ligne]. http://www.etsmtl.ca/SG/Politique/polethsh.pdf (Consulté le 14 novembre 2000).

   Si le document cité est un article de périodique :

   3. Gargour, C.S., Ramachandran, V., Bogdadi, G. (1991). Design of Active RC and Switched Capacitor Filters Having Variable Magnitude Characteristics Using a Unified Approach. J. of Computers and Electrical Engineering, 17(1), 11-12.

Code source

Vous devez fournir au chargé de laboratoire **l'ensemble des fichiers exécutables**. De plus, il faut aussi remettre l'ensemble des **codes sources**, **des figures générées** ainsi que vos **fichiers de résultats de l'analyse expérimentale** et de profilage, en version électronique.

Annexe

```
1   Fonction Algo1(T[0 ... N-1], N)
2       Si (N>=2)
3           m = N/2
4           G = créer_tableau(m entiers)
5           D = créer_tableau((N - m) entiers)
6
7           Pour i de 0 à m-1
8               G[i] = T[i]
9           Fin Pour
10
11          Pour i de 0 à N-m-1
12              D[i] = T[m+i]
13          Fin Pour
14
15          G = Algo1(G, m)
16          D = Algo1(D, N-m)
17          T = Algo2(T, G, m, D, N-m)
18
19          détruire_tableau(G)
20          détruire_tableau(D)
21      Fin Si
22
23      Retourner T
24  Fin fonction
```

(algo2 se retrouve sur la prochaine page)

```
22  Fonction Algo2(T[0 ... N-1], G[0 ... a-1], a, D[0 ... b-1], b)
23      i = 0
24      j = 0
25      k = 0
26      Tant que (i < a) et (j < b)
27              Si (G[i] < D[j])
28                      T[k] = G[i]
29                      i = i + 1
30              Sinon
31                      T[k] = D[j]
32                      j = j + 1
33              Fin Si
34              k = k + 1
35      Fin Tant que
36
37      Tant que (i < a)
38              T[k] = G[i]
39              i = i + 1
40              k = k + 1
41      Fin Tant que
42
        Tant que (j < b)
                T[k] = D[j]
                j = j + 1
                k = k + 1
          Fin Tant que

       Retourner T
    Fin fonction
```