

UE ALGO5 — TD2 — Séance 1 : Mesure expérimentale de la complexité

Objectifs

À la fin de cette séance, vous devriez être capable de :

- instrumenter un programme pour évaluer expérimentalement son temps d'exécution ;
 - faire des choix pertinents pour analyser les traces de cette instrumentation ;
 - réaliser l'APNÉE 1.
-

Introduction/Rappels

La complexité (en temps) d'un algorithme est une **fonction** qui exprime **une mesure du temps d'exécution** (ou «coût») d'un algorithme en fonction de la «taille» des données auxquelles on l'applique.

On peut distinguer : complexité dans les cas favorables, défavorables, en moyenne.

La complexité en moyenne est souvent difficile à calculer/approcher de manière théorique :

- il faut des hypothèses probabilistes sur les données
- la formule obtenue peut être compliquée...

Une approche possible : mesurer **expérimentalement** cette complexité, sur une **implémentation** donnée de l'algorithme, et sur un ensemble de données «types» (benchmark), ou générées aléatoirement.

Comment effectuer la mesure ?

- Mesurer directement le temps d'exécution ?

Problème : il faut être capable d'éliminer tout le «bruit» introduit par le système d'exploitation (allocation mémoire, accès aux disques, gestion de processus, etc.) pour obtenir des résultats «fiables», c'est-à-dire comparables d'une exécution à une autre, indépendamment du système ou de la machine utilisée.

- Il est préférable de revenir à une approche plus proche de la définition «théorique» de la complexité, c'est-à-dire de mesurer un nombre d'opérations élémentaires significatives (par exemple : comparaisons, affectations, nombre d'appels à certaines fonctions, d'accès à des éléments de tableaux, etc.).

On modifie donc le programme pour mesurer/compter, sur chaque exécution, le nombre d'opérations élémentaires qui nous intéresse (instrumentation du code).

Remarques :

- en modifiant ainsi le programme on augmente certainement son temps d'exécution...
 - il faut bien sûr choisir la (ou les) bonne(s) opérations élémentaires (celles qui sont significatives...).
-

Exercice 1.

On considère la fonction C suivante, dans laquelle T et N sont des variables externes (initialisées dans d'autres parties du programme) :

```
1 void f() {  
    int x, i, s;  
3    float m;  
    i = 0 ; s = 0 ;  
5    while (i < N) {  
        i = i + 1 ;  
7        x = T[i] ;  
        if (x != -1) { s = s + x ; }  
9    } ;  
    m = s / i ;  
11    printf ("%f\n", m);  
}
```

- Q1. Modifiez cette fonction pour afficher le nombre d'opérations arithmétiques (addition, division, etc.) qu'elle effectue lors de chaque exécution.
- Q2. Modifiez cette fonction pour afficher également le nombre de comparaisons (\neq , $<$) qu'elle effectue lors de chaque exécution.

Corrigé —

```
void f() {
2   int x, i, s;
   float m;

4   int cpt_arith, cpt_comp;

6   cpt_arith = 0;
8   cpt_comp = 0;

10  i = 0 ; s = 0 ;
   while (i<N) {
12     cpt_comp = cpt_comp + 1;
       cpt_arith = cpt_arith + 1;
14     i = i+1 ;
       x = T[i] ;
16     cpt_comp = cpt_comp + 1;
       if (x != -1) {
18         cpt_arith = cpt_arith + 1;
           s = s + x ;
20     }
       };
22  cpt_comp = cpt_comp + 1; // dernière itération
   m = s/i ;
24  cpt_arith = cpt_arith + 1;
   printf ("%f\n",m);
26  printf ("nombre_d'opérations_arithmétiques_=%d\n",cpt_arith);
   printf ("nombre_de_comparaisons_=%d\n",cpt_comp);
28 }
```

Exercice 2.

On considère le programme suivant, dans lequel g , $g1$ et $g2$ sont des fonctions externes :

```
int main() {
2   double x, y ;
   scanf ("%f%f", &x, &y) ;
4   while (x != 0) {
       if ((g(x)>0) && (x != y)) { x = g1(x) ; } else { y = g2(y) ; }
6   } ;
   printf ("%d\n",x);
8   return 0;
}
```

- Q1. Modifiez ce programme pour afficher le nombre de comparaisons entre x et y qu'il effectue lors de chacune de ses exécutions.

Corrigé —

Problème : il n'est pas facile de savoir quand l'expression $x \neq y$ est évaluée (à cause de la sémantique de l'opérateur $\&\&$ du C qui correspond à un «et puis»).

Il faudrait donc tester la condition $g(x) > 0$ avant d'incrémenter le compteur, du style :

```
1 if (g(x)>0) cpteur = cpteur + 1; /* instrumentation */  
2 if ((g(x)>0) && (x != y) ... /* suite du programme ... */
```

Mais c'est dangereux, on ne sait pas ce que fait g (effets de bords)!

Il est préférable de réécrire une fonction $\text{diff}(x,y)$ qui fait la comparaison et incrémente un compteur :

```
if (g(x)>0) && diff(x,y) ...
```

avec

```
1 int diff (int x, int y) {  
    cpteur = cpteur + 1 ;  
3    return (x != y) ;  
}
```

En résumé : une solution «propre» pour instrumenter consiste à utiliser un ensemble de primitives dédiées à la gestion du compteur, bien identifiées, indépendantes de celles du programme cible :

—init_compteur
—incr_compteur

(ou directement $\text{cpteur} = \text{cpteur} + 1$ pour éviter l'utilisation explicite de pointeurs pour passer un paramètre résultat, merci le C!)

—affiche_compteur

Pour compter le nombre d'occurrence d'une opération élémentaire on peut également la remplacer par une fonction dédiée qui :

—incrémente le compteur (par effet de bord)

—a le même effet que l'opération élémentaire qu'elle remplace (comparaison, affectation, etc.)

Exercice 3.

Q1. Écrivez un programme C qui effectue un parcours séquentiel d'un tableau T de N valeurs entières ($N > 0$) et affiche la valeur maximale de ce tableau (N est une constante, on ne détaille pas l'initialisation de T).

Corrigé —

```
int main() {
2   int i ;
   int max ;

4   initialisation() ; /* initialisation du tableau T */

6   max = T[0] ;
8   for (i=1 ; i<N ; i++) {
   if (T[i] > max) {
10      max = T[i] ;
   }
12 }
   printf("%d\n", max) ;
14 }
```

Q2. On s'intéresse à une fonction de coût f_c qui associe à chaque exécution de ce programme le *nombre de fois* où un «nouveau» maximum a été obtenu lors du parcours. Instrumentez le programme pour afficher à la fin de chacune de ces exécutions la valeur de cette fonction de coût.

Corrigé —

```
int main() {
2   int i ;
   int max ;
4   int compteur = 0 ;

6   initialisation() ; /* initialisation du tableau T */

8   max = T[0] ;
   for (i=1 ; i<N ; i++) {
10      if (T[i] > max) {
   max = T[i] ;
12      compteur = compteur + 1 ;
   }
14 }
   printf("le maximum est %d\n", max) ;
16   printf("un nouveau maximum a été obtenu %d fois\n", compteur) ;
   return 0 ;
18 }
```

Q3. En fonction de N, quelle est la valeur minimale que l'on peut obtenir pour f_c ? Quelle est la valeur maximale? Donnez des contenus possibles pour le tableau T pour lesquels ces valeurs sont atteintes.

Corrigé —

- Valeur minimale pour f_c : 0 (si le maximum est en tête)
 - Valeur maximale pour f_c : $N - 1$ (si le tableau est trié en ordre croissant)
-

Q4. On souhaite mesurer expérimentalement la *valeur moyenne* de f_c en fonction de N. Comment peut-on

s'y prendre?

Corrigé —

1. Déterminer la «taille» des données, c'est-à-dire le paramètre à faire varier pour mesurer le coût : ici c'est N ...
2. Il faut choisir les données sur lesquelles expérimenter. Ce peut être :
 - des données aléatoires, selon une loi de probabilité
 - des données «réelles», correspondant à des exemples concrets (benchmark)
 - des données choisies pour illustrer des cas favorables, des cas défavorables.
3. Faire une première série de mesures pour regarder comment se comporte l'algorithme, et déterminer l'intervalle sur lequel il va être pertinent de travailler. On peut par exemple commencer par faire varier N de 100 en 100 puis s'apercevoir que le temps d'exécution est très court (ou très lent), que les valeurs obtenues varient peu (ou énormément)...
4. Une fois cet «intervalle de travail» déterminé, on peut refaire une série de mesures plus précises qui produisent un «tableau de mesure».

Remarque : il faut automatiser autant que possible cette partie (par exemple, produire les résultats dans un fichier).

5. Il faut ensuite exploiter les résultats, par exemple en traçant la courbe (ou un histogramme), et en le comparant avec une courbe ou un histogramme connu (que l'on suppose proche du résultat attendu).
 6. Il faut enfin **interpréter** les résultats : par exemple, sont-ils conformes au résultat théorique attendu? S'il s'agit de comparer deux coûts (de deux algorithmes par exemple), que peut-on conclure de la comparaison des deux courbes? etc.
-