TP d'algorithmique avancée
Corrigé du TP 1 : complexité et temps d'exécution

Jean-Michel Dischler et Frédéric Vivien

Recherche simultanée du minimum et du maximum

1. Écrivez un programme qui implémente d'une part l'algorithme naïf de recherche simultanée du minimum et du maximum, et d'autre part l'algorithme optimal vu en TD (si besoin est, le corrigé du TD est disponible à l'URL : `http://icps.u-strasbg.fr/~vivien/Enseignement/Algo-2001-2002/index.html`).

   Comparez les temps d'exécution des deux algorithmes (par exemple en utilisant la fonction `clock`). Qu'observez-vous ? Qu'en concluez-vous ?

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

int main(int argc, char ** argv)
{
  int i;
  long int n,* tableau, min, max;
  long int debut, apres_premier, apres_second;

  /*Initialisations*/
  if (argc!=2){
    fprintf(stderr,"Le programme prend
       obligatoirement un argument\ n");
    return(-1);
  }
  n = atol(argv[1]);
  tableau = malloc(n*sizeof(long int));
  for(i=0; i<n; i++) tableau[i] = random();
  debut  = clock();


  /*Premier algorithme*/
  min = tableau[0];
  max = tableau[0];
  for(i=1; i<n; i++){
    if (tableau[i] > max) max=tableau[i];
    if (tableau[i] < min) min=tableau[i];
  }
  apres_premier = clock();

  /*Second algorithme*/
  if (tableau[1]>tableau[0]){
    min = tableau[0];
    max = tableau[1];
  }
  else{
    min = tableau[1];
    max = tableau[0];
  }

  for(i=2; i<n-1; i+=2){
    if (tableau[i+1]>tableau[i]){
      if (tableau[i+1]>max) max=tableau[i+1];
      if (tableau[i]  <min) min=tableau[i];
    }
    else{
      if (tableau[i]  >max) max=tableau[i];
      if (tableau[i+1]<min) min=tableau[i+1];
    }
  }
  if ((n%2)!=0){
      if (tableau[n] > max) max=tableau[n];
      if (tableau[n] < min) min=tableau[n];
  }
  apres_second  = clock();

  /* Affichage des temps d'execution*/
  fprintf(stderr,"Premier  algorithme: %.3lf\n
          Deuxieme algorithme: %.3lf\n",
          (1.0*(apres_premier-debut))
             /CLOCKS_PER_SEC,
          (1.0*(apres_second-apres_premier))
             /CLOCKS_PER_SEC);
  return 0;
}
```

Le résultat dépend de l'ordinateur sur lequel ces fonctions sont testées, mais la version de complexité la plus faible n'est pas plus rapide. En effet, son code est plus compliqué et met en œuvre des opérations

dont le coût est comparable à celui des fonctions de comparaisons (seules prises en compte lors du calcul de la complexité).

2. Récrivez le programme précédent en remplaçant les comparaisons au moyen des opérateurs $>$ et $<$ par des appels à une fonction `compare(a, b)` qui renvoie la valeur du test « `a` > `b` ».
   Qu'observez-vous ? Qu'en concluez-vous ?

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

int compare(long int a, long int b)
{
  int i, j =0;
  for(i=0; i<1000; i++) j++;

  return (a > b);
}

int main(int argc, char ** argv)
{
  int i;
  long int n, * tableau, min, max;
  long int debut, apres_premier,
           apres_second;

  /*Initialisations*/
  if (argc!=2){
    fprintf(stderr,"Le programme prend
       obligatoirement un argument\n");
    return(-1);
  }
  n = atol(argv[1]);
  tableau = malloc(n*sizeof(long int));
  for(i=0; i<n; i++) tableau[i]=random();
  debut = clock();

  /*Premier algorithme*/
  min = tableau[0];
  max = tableau[0];
  for(i=1; i<n; i++){
    if (compare(tableau[i], max))
       max = tableau[i];
    if (compare(min, tableau[i]))
       min = tableau[i];
  }
  apres_premier = clock();

  /* Second algorithme*/
  if (compare(tableau[1], tableau[0])){
    min = tableau[0];
    max = tableau[1];
  }
  else{
    min = tableau[1];
    max = tableau[0];
  }

  for(i=2; i<n-1; i+=2){
    if(compare(tableau[i+1],tableau[i])){
      if (compare(tableau[i+1], max))
         max = tableau[i+1];
      if (compare(min, tableau[ i ]))
         min = tableau[i];
    }
    else{
      if (compare(tableau[ i ], max))
         max = tableau[i];
      if (compare(min, tableau[i+1]))
         min = tableau[i+1];
    }
  }
  if ((n%2)!=0){
      if (compare(tableau[n], max))
         max = tableau[n];
      if (compare(min, tableau[n]))
         min = tableau[n];
  }
  apres_second  = clock();

  /* Affichage des temps d'execution*/
  fprintf(stderr,"
     Premier  algorithme: %.3lf\n
     Deuxieme algorithme: %.3lf\n",
     (1.0*(apres_premier-debut))
        /CLOCKS_PER_SEC,
     (1.0*(apres_second-apres_premier))
        /CLOCKS_PER_SEC);
  return 0;
}
```

Le fait de remplacer l'invocation directe de la comparaison (via les opérateurs $<$ et $>$) par un appel de fonction augmente considérablement le coût d'exécution de ces tests. Quand le coût des tests devient nettement plus important que celui des autres opérations, les programmes se comportent comme le prédit leur complexité ***en nombre de comparaisons****.*

Calcul de x^n

1. Écrivez un programme qui implémente d'une part l'algorithme naïf de calcul de x^n et d'autre part la « méthode binaire » vue en cours (si besoin est, le cours est disponible à l'URL : `http://icps.u-strasbg.fr/~vivien/Enseignement/Algo-2001-2002/index.html`).

   Pour implémenter cet algorithme, vous avez besoin de pouvoir récupérer le i^{e} bit d'un entier. Pour ce faire vous pouvez utiliser les décallages : $n >> i$ est équivalent à une division par 2^i et amène donc le i^{e} bit en position de poids faible ; et l'opérateur « & » qui est le *et* logique bit à bit.

   Comparez les temps d'exécution des deux algorithmes (par exemple en utilisant la fonction `clock`). Qu'observez-vous ? Qu'en concluez-vous ?

```
#include <stdlib.h>
#include <time.h>
#include <stdio.h>

int main(int argc, char ** argv)
{
  int i, rang;
  long int n, x, resultat;
  long int debut, apres_premier,
           apres_second;

  /* Initialisations*/
  if (argc!=3){
    fprintf(stderr,"Le programme prend
       obligatoirement deux arguments.\n");
  return(-1);
  }
  n = atol(argv[1]); x = atoi(argv[2]);

  /* Premier algorithme */
  debut = clock();
  resultat = x;
  for(i=2; i<=n; i++) resultat = x * resultat;
  fprintf(stderr,"resultat= %ld\n", resultat);
  apres_premier = clock();

  /* Deuxieme algorithme */
  resultat = x;
  /*Calcul du rang du bit de poids fort*/
  rang = sizeof(long int)*8-1;
  while(((n >> rang)==0)&&(rang>0)) rang--;

  for(rang = rang-1; rang >=0; rang--){
    if ((n >> rang)&1){
      resultat = resultat * resultat;
      resultat = x*resultat;
    }
    else{
      resultat = resultat * resultat;
    }
  }

  fprintf(stderr,"resultat = %ld\n",
     resultat);
  apres_second = clock();

  /* Affichage des temps d'execution*/
  fprintf(stderr,"
     Premier  algorithme: %.3lf\n
     Deuxieme algorithme: %.3lf\n",
     (1.0*(apres_premier-debut))
        /CLOCKS_PER_SEC,
     (1.0*(apres_second-apres_premier))
        /CLOCKS_PER_SEC);
  return 0;
}
```

   Les temps d'exécution reflète bien ici la complexité telle qu'elle a été calculée en cours.

2. Pour pouvoir comparer effectivement les deux algorithmes, à la question précédente, il vous a fallu utiliser des valeurs de la puissance, n, relativement élevées. De ce fait, les résultats des calculs étaient forcément faux et généraient des dépassement de capacités (*overflow*), à moins de n'essayer de ne calculer que des puissances de 0, 1 ou -1.

   Pour remédier à ce problème, récrivez votre programme en utilisant la librairie *gmp* de GNU qui permet de faire du calcul en précision arbitraire. Pour ce faire, vous avez uniquement besoin d'inclure le fichier de déclaration idoine (`gmp.h`), de lier votre application avec la librairie *ad-hoc* (`gmp`), et d'utiliser les fonctions et constructions appropriées, celles listées ci-dessous suffisant amplement :
   - `mpz_t q` : déclare l'entier `q` en précision arbitraire ;
   - `mpz_init(mpz_t q)` : effectue l'initialisation de l'entier `q`, cette initialisation est indispensable ;
   - `mpz_set_ui(mpz_t q, unsigned long x)` : affecte la valeur de `x` à `q` ;
   - `mpz_mul(mpz_t a, mpz_t b, mpz_t c)` : effectue la multiplication de `b` et de `c` et stocke le résultat dans `a`.

– `mpz_out_str(FILE * stream, int base, mpz_t q)` : affiche sur le flot de sortie `stream` (typiquement `stderr` ou `stdout`) la valeur de l'entier `q` exprimée dans la base `base` qui doit être un entier compris entre 2 et 32.

Comparez les temps d'exécution des deux algorithmes. Qu'observez-vous ? Qu'en concluez-vous ?

```
#include <stdlib.h>
#include <time.h>
#include <stdio.h>
#include <gmp.h>


int main(int argc, char ** argv)
{
  int i, rang;
  long int x, n;
  mpz_t resultat, mpg_x;
  long int debut, apres_premier,
           apres_second;

  /*Initialisations*/
  if (argc!=3){
    fprintf(stderr,"Le programme prend
       obligatoirement deux arguments.\n");
    return(-1);
  }
  n = atol(argv[1]);
  x = atoi(argv[2]);
  mpz_init(resultat);
  mpz_init(mpg_x);
  mpz_set_ui(mpg_x, x);

  /* Premier algorithme */
  debut  = clock();
  mpz_set_ui(resultat, x);
  for(i=2; i<=n; i++)
     mpz_mul(resultat, mpg_x, resultat);
  fprintf(stderr,"resultat = ");
  mpz_out_str(stderr, 10, resultat);
  fprintf(stderr,"\n");
  apres_premier = clock();

  /* Deuxieme algorithme */
  mpz_set_ui(resultat, x);
  /*Calcul rang du bit de poids fort*/
  rang = sizeof(long int)*8-1;
  while(((n >> rang)==0)&&(rang>0)) rang--;

  for(rang = rang-1; rang >=0; rang--){
    if ((n >> rang)&1){
      mpz_mul(resultat, resultat, resultat);
      mpz_mul(resultat, mpg_x, resultat);
    }
    else{
      mpz_mul(resultat, resultat, resultat);
    }
  }
  fprintf(stderr,"resultat = ");
  mpz_out_str(stderr, 10, resultat);
  fprintf(stderr,"\n");

  apres_second  = clock();

  /* Affichage des temps d'execution*/
  fprintf(stderr,"
     Premier  algorithme: %.3lf\n
     Deuxieme algorithme: %.3lf\n",
     (1.0*(apres_premier-debut))
        /CLOCKS_PER_SEC,
     (1.0*(apres_second-apres_premier))
        /CLOCKS_PER_SEC);

return 0;
}
```

On retrouve les différences prédites par les études de complexité. Cependant, ici, le coût d'une multiplication n'est plus constant, il dépend de la taille des opérandes. De ce fait, les multiplications sont très rapidement plus coûteuses (car elles ont lieu sur les valeurs réelles, et donc sur des nombres plus grands), la différence entre les algorithmes apparaît plus rapidement, c'est-à-dire pour des valeurs de la puissance plus petite, ce qui donne l'illusion d'une différence d'efficacité moindre.