Conception des algorithmes
Stratégies Gauche-droite / Droite-gauche

Lundi 12 Février 2018

Michael FRANÇOIS

francois@esiea.fr

https://francois.esiea.fr

Conception des algorithmes

• Il existe de nombreuses techniques pour concevoir un algorithme. Dans les prochains cours, chaque technique sera étudiée en détail et des exemples seront donnés pour chacune d'entre-elles :

- méthode **diviser-pour-régner** (Séparer le problème en plusieurs sous-problèmes "indépendants" semblables au problème initial mais de taille moindre, résoudre ensuite les sous-problèmes puis combiner leurs solutions pour produire la solution du problème de départ)
- méthode de **programmation dynamique** (Comme la méthode diviser-pour-régner, elle résout des problèmes en combinant des solutions de sous-problèmes. Elle s'applique même lorsque les sous-problèmes possèdent des sous-problèmes en communs. Elle s'applique généralement aux problèmes d'optimisation)
- algorithmes **gloutons** (Un algorithme glouton fait un choix localement optimal dans l'espoir que ce choix aboutisse vers une solution qui est globalement optimale)

• L'objectif est de donner des méthodes pour que vous soyez capables de concevoir vos propres algorithmes lorsque le besoin se fera sentir.

• Évidemment nous ne négligerons pas la pratique et nous (vous) programmerons(ez) ces exemples d'algorithmes.

Programmation itérative / récursive

Programmation itérative

- Un **programme itératif** permet de répéter/réitérer une ou plusieurs instructions un certain nombre de fois.

- Une boucle est utilisée pour répéter les actions ainsi qu'une variable qui s'incrémentera à chaque passage, jusqu'à ce que la condition d'arrêt soit satisfaite.

Exemple :

```
#include <stdio.h>

int FACTORIELLE(int N)
{
   int i, Fact;
   Fact=1;
   if (N==0) return Fact;
   for (i=2; i<=N; i++)
   {
     Fact = Fact*i;
   }
   return Fact;
}

int main(int argc, char** argv)
{
  int N = 1;

  while (N != 0)
  {
    printf("Saisir un nombre : ");
    scanf("%d", &N);
    printf("%d! = %d\n", N, FACTORIELLE(N));
  }
  return 0;
}
```

Exécution :

```
$ gcc -o EXEC test.c -Wall
$ ./EXEC
Saisir un nombre : 2
2! = 2
Saisir un nombre : 3
3! = 6
Saisir un nombre : 9
9! = 362880
Saisir un nombre : 0
0! = 1
$
```

Programmation récursive

- La récursion est un concept fondamental en mathématique/informatique.
- Un **programme récursif** est tout simplement un programme qui s'appelle lui même, ou qu'une fonction est récursive si elle est définie en référence à elle-même.
- Une condition de terminaison est nécessaire pour indiquer au programme à ne plus faire appel à lui même ou à la fonction à ne plus être définie en référence à elle-même.
- Le prochain cours sera totalement dédié à la récursivité.

Exemple :

```c
#include <stdio.h>

int FACTORIELLE(int N)
{
   if (N==0) return 1;
   return N*FACTORIELLE(N-1);
}

int main(int argc, char** argv)
{
  int N = 1;

  while (N != 0)
  {
    printf("Saisir un nombre : ");
    scanf("%d", &N);
    printf("%d! = %d\n", N, FACTORIELLE(N));
  }
  return 0;
}
```

Exécution :

```
$ gcc -o EXEC test.c -Wall
$ ./EXEC
Saisir un nombre : 3
3! = 6
Saisir un nombre : 10
10! = 3628800
Saisir un nombre : 0
0! = 1
$ _
```

Stratégies Gauche-droite / Droite-gauche

Stratégies gauche-droite / droite-gauche

- Lorsqu'on manipule des entiers, des vecteurs ou plus généralement des listes, il existe au moins deux manières de parcourir les données :
  - on peut parcourir la liste de **gauche à droite** (indice 0 à n-1) ;
  - ou la parcourir de **droite à gauche** (indice n-1 à 0).
- Nous n'obtenons généralement pas du tout le même algorithme.

- Lorsque l'un des deux algorithmes est obtenu en renversant le sens de parcours de l'autre, *Knuth* propose de définir les deux algorithmes obtenus comme duaux.

Nous retiendrons cette définition par la suite.

- En pratique, entre deux algorithmes duaux, l'un est plus rapide que l'autre (pas forcément en théorie).
- Ainsi, dans tous les cas, il faut toujours se poser la question de savoir si on parcourt la liste dans le ≪bon sens≫ (*i.e.* le sens de parcours des données).
- Pour illustrer notre propos, nous allons étudier le problème du calcul du miroir d'un nombre.

Miroir d'un nombre

- Soit un nombre entier $n > 0$ (en base 10), où

$$n = d_k d_{k-1} d_{k-2} \cdots d_2 d_1 d_0$$

avec $d_k \neq 0$. L'image miroir de n est définie par :

$$miroir(n) = d_0 d_1 d_2 \cdots d_{k-2} d_{k-1} d_k$$

- Exemples :
  - miroir(1234567) = 7654321
  - miroir(519754224) = 422457915.
- Quoi faire ? Deux stratégies possibles :
  1. parcours **Droite-Gauche** des chiffres de n ;
  2. parcours **Gauche-Droite** des chiffres de n ;

Première stratégie : parcours «Gauche-Droite» des chiffres de n

- $1 \rightarrow 1$
- $12 \rightarrow 21$
- $123 \rightarrow 321$
- $1234 \rightarrow 4321$
- $12345 \rightarrow 54321$
- $123456 \rightarrow 654321$
- $1234567 \rightarrow 7654321$

Remarque :
Lorsque le problème du nombre miroir est posé aux étudiants, très souvent leur première idée est d'écrire une fonction qui calcule le nombre k de chiffres du nombre n. Ils proposent alors en grande majorité un algorithme qui, par exemple pour $n = 123$, revient à calculer successivement :
1, 21 puis 321.
Une fois k connu, on calcule d'abord d_k et on itère. C'est bien alors un parcours ≪Gauche-Droite≫ des chiffres que nous faisons.

Mais pour calculer le nombre miroir de n faut-il **nécessairement connaître** k le nombre de chiffres de n ??????

Remarque :
Lorsque le problème du nombre miroir est posé aux étudiants, très souvent leur première idée est d'écrire une fonction qui calcule le nombre k de chiffres du nombre n. Ils proposent alors en grande majorité un algorithme qui, par exemple pour $n = 123$, revient à calculer successivement :
1, 21 puis 321.
Une fois k connu, on calcule d'abord d_k et on itère. C'est bien alors un parcours ≪Gauche-Droite≫ des chiffres que nous faisons.

Mais pour calculer le nombre miroir de n faut-il **nécessairement connaître** k le nombre de chiffres de n ?????? $\Rightarrow$
Non !!!

EXERCICES : (À faire en TD)

- Écrire une fonction permettant de déterminer de manière itérative, la longueur k d'un nombre.
- Écrire une fonction permettant de calculer de manière itérative le miroir d'un nombre. Le balayage sera fait bien-sur de gauche à droite.

Deuxième stratégie : "algorithme dual" parcours ≪Droite-Gauche≫ des chiffres de n

• Que donne l'algorithme dual, à savoir le parcours du nombre n de la droite vers la gauche ?

• Il peut sembler curieux d'opter pour une stratégie ≪Droite-Gauche≫ pour ce problème, mais voyons pourquoi :

- Si $n = yz$ alors $miroir(n) = (z \times 10) + y$
- Si $n = xyz$ alors $miroir(n) = (zy \times 10) + x = (miroir(yz) \times 10) + x$
  $$= ((z \times 10) + y) \times 10 + x$$
  $$= z \times 100 + y \times 10 + x$$
- De même, si $n = wxyz$ alors $miroir(n)$ est égal à $[\ (z \times 10 + y) \times 10 + x\ ] \times 10 + w$

Remarque : et donc c'est toujours le calcul de $z \times 10$ qui semble être effectué en premier.

Prenons l'exemple de $n = xyz$ dont le miroir corresponds à : $z \times 10^2 + y \times 10 + x$.

- Comment récupérer le chiffre z ?
- Comment se débarrasser du chiffre z pour obtenir $n = xy$?

- Prenons l'exemple de $n = xyz$ dont le miroir corresponds à : $z \times 10^2 + y \times 10 + x$.
  - $n\%10$ permet simplement d'obtenir le chiffre z,
  - $n/10$ permet d'obtenir l'autre partie du nombre n à savoir ici xy.

- En clair, sans connaître le nombre de chiffres constituant n, obtenir le chiffre de poids le plus faible (*i.e.* le plus à droite) de n est simple : c'est $n\%10$. Et $n/10$ permet d'écarter/d'enlever le chiffre le plus à droite de notre nombre n.

- La stratégie ≪Droite-Gauche≫ semble donc a priori plus adaptée à notre problème !!!

Avec 4 chiffres :

• Pour $n = wxyz$, le miroir de n est :

$$z \times 10^3 + y \times 10^2 + x \times 10 + w$$

que l'on peut réécrire sous la forme :

$$[\ (z \times 10 + y) \times 10 + x\ ] \times 10 + w$$

• On appelle forme de Hörner l'expression du polynôme correspondant au nombre entier $zyxw$ en base 10.

- Ainsi, dans le cas où :

$$n = d_k d_{k-1} d_{k-2} \cdots d_2 d_1 d_0$$

alors $miroir(n)$ est donné par :

$$d_0 d_1 d_2 \cdots d_{k-2} d_{k-1} d_k$$

ce qui signifie en fait que :

$$miroir(n) = d_0 \times 10^k + d_1 \times 10^{k-1} + \cdots + d_{k-1} \times 10 + d_k$$

- On en déduit ainsi que l'algorithme de Hörner peut être utilisé pour calculer le nombre miroir.

Quelques exemples :

- $12 \rightarrow 21 = 2 \times 10 + 1$
- $123 \rightarrow 321 = [((3 \times 10) + 2) \times 10] + 1$
- $1234 \rightarrow 4321 = [((4 \times 10 + 3) \times 10) + 2] \times 10 + 1$
- $12345 \rightarrow 54321 = [((5 \times 10 + 4) \times 10 + 3) \times 10 + 2] \times 10 + 1$
- $123456 \rightarrow 654321 = ...$
- $1234567 \rightarrow 7654321 = ...$
- etc.

- L'algorithme de Hörner donne :

$$miroir(n) = [\ \cdots ((d_0 \times 10) + d_1) \times 10 + \cdots d_{k-1}\ ] \times 10 + d_k$$

- On voit alors que les calculs peuvent se faire plus facilement, et plus naturellement, dans ≪l'ordre d'arrivée≫ des chiffres $d_0, d_1, \ldots$;

- L'algorithme peut commencer sans même connaître le nombre de chiffres qui composent n.

- Le test d'arrêt est simple : tant qu'il reste ≪quelque chose de non nul≫ on continu le traitement.

- Il est conseillé d'utiliser la structure de contrôle `while`, lorsqu'on ne connaît pas à l'avance le nombre d'itérations à effectuer. Dans le cas où ce dernier est connu, on peut utiliser la boucle `for`.

Code C :

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

int MIROIR(int n)
{
   int mir, nbre;
   mir=0; nbre=n;
   while (nbre > 0)
   {
      mir = mir*10 + (nbre % 10);
      nbre = nbre/10;
   }
   return mir;
}

int main(int argc, char** argv)
{
  int n = 1;

  while (n != 0)
  {
    printf("Saisir un nombre : ");
    scanf("%d", &n);
    printf("%d --------> %d\n", n, MIROIR(n));
  }
  return 0;
}
```

Exécution :

```
$
$ gcc -o EXEC test.c -Wall
$ ./EXEC
Saisir un nombre : 12345
12345 --------> 54321
Saisir un nombre : 865219
865219 --------> 912568
Saisir un nombre : 424845681356742l5242
-1 --------> 0
Saisir un nombre : 0
0 --------> 0
$
```

Code C modifié:

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

int MIROIR(int n)
{
   int mir, nbre;
   mir=0; nbre=n;
   printf("%d <--------> %d\n", nbre, mir);
   while (nbre > 0)
   {
      mir = mir*10 + (nbre % 10);
      nbre = nbre/10;
      printf("%d <--------> %d\n", nbre, mir);
   }
   return mir;
}

int main(int argc, char** argv)
{
  int n = 1;

  while (n != 0)
  {
    printf("Saisir un nombre : ");
    scanf("%d", &n);
    MIROIR(n);
  }
  return 0;
}
```

Exécution :

```
$
$ gcc -o EXEC test.c -Wall
$ ./EXEC
Saisir un nombre : 12345
12345 <--------> 0
1234 <--------> 5
123 <--------> 54
12 <--------> 543
1 <--------> 5432
0 <--------> 54321
Saisir un nombre : 842156
842156 <--------> 0
84215 <--------> 6
8421 <--------> 65
842 <--------> 651
84 <--------> 6512
8 <--------> 65124
0 <--------> 651248
Saisir un nombre : 0
0 <--------> 0
$
```

Exemple de parcours gauche-droite/droite-gauche sur un tableau d'entiers

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

void REMPLISSAGE_TAB(int TAB[])
{
  int i;
  for (i=0; i<100; i++)
  {
    TAB[i] = rand()%1000; /*valeurs aléatoires entre 0 et 999*/
  }
}

int main(int argc, char** argv)
{
  int compt, val;
  int TAB[100]; /*Déclaration d'un tableau de taille 100*/
  REMPLISSAGE_TAB(TAB); /*Remplissage du tableau par des valeurs aléatoires*/

  compt=0; val = 94; /*Initialisation du compteur et de la variable val*/
  while (TAB[compt] != val && compt < 100) /* Parcours Gauche-->Droite */
  {compt++;}
  printf("G->D | Valeur %d trouvée après %d itération(s) \n", val, compt);

  compt=99; /*Réinitialisation du compteur*/
  while (TAB[compt] != val && compt >=0) /* Parcours Droite-->Gauche */
  {compt--;}
  printf("D->G | Valeur %d trouvée après %d itération(s) \n", val, 99-compt);
  return 0;
}
```

Exécution avec $val = 94$:

```
$ gcc -o EXEC test.c -Wall
$ ./EXEC
G->D | Valeur 94 trouvée après 98 itération(s)
D->G | Valeur 94 trouvée après 1 itération(s)
$
```

Exécution avec $val = 27$:

```
$ gcc -o EXEC test.c -Wall
$ ./EXEC #ici val=27
G->D | Valeur 27 trouvée après 11 itération(s)
D->G | Valeur 27 trouvée après 88 itération(s)
$
```

Remarque : on voit que pour cet exemple, quand $val = 94$ le parcours **droite-gauche** est plus efficace, mais lorsque $val = 27$ c'est le parcours **gauche-droite** qui est le plus efficace. Dans le cas où les valeurs du tableau sont renouvelées constamment, les performances des deux parcours restent à peu prêt équilibrées.

EXERCICE : donné au contrôle TD le 23/03/2016

```
répéter
    Affecter 1 à T
    pour i de 0 à n−2 inclus faire
        si (TAB[i] > TAB[i+1]) alors
            Échanger TAB[i] et TAB[i+1]
            Affecter 0 à T
        fin
    fin
tant que T est égale à 0;
```

Quel est le rôle de cet algorithme ? Donner la version droite-gauche de cet algorithme.

EXERCICE : donné au contrôle TD le 23/03/2016

Rép : version droite-gauche du tri à bulles

répéter
  Affecter `1` à `T`
  pour *i de n-1 à 1 inclus* **faire**
    si *(TAB[i] < TAB[i-1])* **alors**
      Échanger `TAB[i]` et `TAB[i-1]`
      Affecter `0` à `T`
    fin
  fin
tant que *T est égale à 0;*

Bibliographie

- R. ERRA, "Cours 3 d'informatique", 1A-S2 2013-2014 ESIEA.
- R. Sedgewick, "Algorithmes en langage C", 2005, DUNOD.