Exercices avec le logiciel R

Épreuve d'Informatique Filière 4: DEUG Sciences de la Vie. Parcours 4 (BioMathématiques)

J.R. Lobry

Contrôle terminal -Juin 1997

Documents visés autorisés

1 Introduction

On reprend ici un vieil exercice de programmation en langage C et on s'amuse à comparer avec une solution dans le langage S implémenté dans R.

2 Sujet

Une intéressante propriété du nombre 6174 a été remarquée par D.R. Kaprekar [1]. Soit N_1 un nombre entier positif de 4 chiffres. Formez à partir de ses chiffres deux nombres : le plus grand avec ses chiffres écrits dans l'ordre descendant, et le plus petit avec ses chiffres écrits dans l'ordre ascendant, et calculez la différence N_2 de ces deux nombres. Traitez N_2 de la même manière et formez la séquence N_1, N_2, N_3, ..., N_i. Les deux propriétés suivantes sont caractéristiques de cet algorithme :

1. Le nombre 6174 est un invariant, parce que $7641 - 1467 = 6174$.
2. Tous les autres nombres à 4 chiffres conduisent au même résultat après i opérations, $i \leq 8$. Exemple :

```
N1 = 3015    5310 - 0135 = 5175 = N2
             7551 - 1557 = 5994 = N3
             9954 - 4599 = 5355 = N4
             5553 - 3555 = 1998 = N5
             9981 - 1899 = 8082 = N6
             8820 - 0288 = 8532 = N7
             8532 - 2358 = 6174 = N8
```

Bien entendu, nous devons exclure les nombres formés de 4 chiffres identiques.

On demande d'écrire en langage C un programme qui demande à l'utilisateur un nombre entier de 4 chiffres et affiche la suite N_1, N_2, N_3, ..., N_i correspondante.

3 Solution en C

Le programme ci-après fait un peu plus que ce qui était demandé : il passe en revue tous les cas possibles pour vérifier que l'on a bien convergence vers 6174 dans tous les cas. La sortie du programme est la suivante :

```
#0000 0000
#0001 0999 8991 8082 8532 6174
#0002 1998 8082 8532 6174
#0003 2997 7173 6354 3087 8352 6174
#0004 3996 6264 4176 6174
#0005 4995 5355 1998 8082 8532 6174
#0006 5994 5355 1998 8082 8532 6174
#0007 6993 6264 4176 6174
#0008 7992 7173 6354 3087 8352 6174
[...]
#7889 1998 8082 8532 6174
#7899 2088 8532 6174
#7999 1998 8082 8532 6174
#8888 0000
#8889 0999 8991 8082 8532 6174
#8899 1089 9621 8352 6174
#8999 0999 8991 8082 8532 6174
#9999 0000

/*****************************************************************************

                                KAPREKAR

A simple program to study convergence to kaprekar constant for all integers
in the range 0000-9999.

*****************************************************************************/

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

typedef enum BOOLEEN { FAUX, VRAI } booleen;
typedef enum CHOIX_TRI { ASCENDENT, DESCENDENT } choix_tri;

#define N 4              /* Number of digits in a kaprekar number */
const int K_MIN = 0000;  /* Min int value for a kaprekar number */
const int K_MAX = 9999;  /* Max int value for a kaprekar number */

typedef int kaprekar[4] ;
const kaprekar k_const = { 6, 1, 7, 4 };
const kaprekar k_nul = { 0, 0, 0, 0 };

#define NOT_KPR_MSG "Not a Kaprekar number"
#define NULL_PTR_MSG "Null pointer"

/*****************************************************************************

                              CHECK_KAPREKAR

Return VRAI if input x is a correct kaprekar number. For beeing a correct
kaprekar number, all positions should be in the range [0-9], and the
pointer should nor be the null pointer.

Abort with message on stderr if uncorrect.

*****************************************************************************/
void check_kaprekar( const kaprekar x , char *from_msg)
{
  booleen is_kaprekar = VRAI;
  int i;

  if( x == NULL )
  {
    fprintf(stderr, "Error: %s from %s\n", NULL_PTR_MSG, from_msg);
    exit(EXIT_FAILURE);
```

```
  }

  for( i = 0 ; i < N ; i++)
    if( x[i] < 0 || x[i] > 9 )
    {
      is_kaprekar = FAUX;
      break;
    }

  if( is_kaprekar == FAUX )
  {
    fprintf(stderr, "Error: %s from %s\n", NOT_KPR_MSG, from_msg);
    exit(EXIT_FAILURE);
  }

}

/*****************************************************************************

                                K_TO_INT

Convert the kaprekar number x in its int equivalent. Abort if input x is not
a correct kaprekar number.

*****************************************************************************/
int k_to_int( const kaprekar x )
{
  int resultat;
  check_kaprekar(x, "x in k_to_int");

  resultat = 1000*x[0] + 100*x[1] + 10*x[2] + x[3];

  return( resultat );
}


/*****************************************************************************

                                INT_TO_K

Convert the input int number x in its kaprekar representation.
The result is returned in resultat.

*****************************************************************************/
void int_to_k( int x, kaprekar resultat )
{
  int denom = pow(10, N-1);
  int i;

  if( x < K_MIN || x > K_MAX )
  {
    fprintf(stderr,
    "Error : "NOT_KPR_MSG" for argumnent x in function int_to_k\n");
    exit(EXIT_FAILURE);
  }

  for( i = 0 ; i < N ; i++ , denom /= 10 )
  {
    resultat[i] =  x/denom;
    x -= resultat[i]*denom;
  }

  check_kaprekar( resultat, "resultat in int_to_k after conversion");

}

/*****************************************************************************

                                K_EGAL

Returns VRAI if both arguments are equals.

*****************************************************************************/
booleen k_egal( const kaprekar x, const kaprekar y)
```

```
{
  int i;

  check_kaprekar(x, "x in k_egal");
  check_kaprekar(y, "y in k_egal");

  for( i = 0 ; i < N ; i++)
   if( x[i] != y[i] )
    return(FAUX);

  return(VRAI);
}

/*****************************************************************************

                                K_SET

Set a kaprekar number value

*****************************************************************************/
void k_set( kaprekar x, int x0, int x1, int x2, int x3)
{
  x[0]=x0;
  x[1]=x1;
  x[2]=x2;
  x[3]=x3;
  check_kaprekar(x, "x in k_set");
}

/*****************************************************************************

                                K_INF

Returns VRAI if x < y.

*****************************************************************************/
booleen k_inf( const kaprekar x, const kaprekar y)
{
  check_kaprekar(x, "x in k_inf");
  check_kaprekar(y, "y in k_inf");
  if( k_to_int(x) < k_to_int(y) )
    return(VRAI);
  else
    return(FAUX);
}

/*****************************************************************************

                                K_CPY

Copy kaprekar number from into to.

*****************************************************************************/
void k_cpy( const kaprekar from, kaprekar to)
{
  int i;

  check_kaprekar(from, "from in k_cpy");

  for( i = 0 ; i < N ; i++)
    to[i] = from[i];

  check_kaprekar(to, "to in k_cpy");

}

/*****************************************************************************

                                K_SORT

This function sorts a kaprekar number x by increasing or decreasing digits
according to the value of flag choix (ASCENDENT or DESCENDENT).

*****************************************************************************/
```

```
void k_sort( kaprekar x, choix_tri choix)
{
  int cmp_ascendent(int *x, int *y);
  int cmp_descendent(int *x, int *y);

  check_kaprekar(x, "x in k_sort before qsort");
  if( choix == ASCENDENT )
  {
    qsort( x, N, sizeof(int), (int(*)(const void*, const void*))cmp_ascendent );
  }
  else
  {
    qsort( x, N, sizeof(int), (int(*)(const void*, const void*))cmp_descendent);
  }

  check_kaprekar(x, "x in k_sort after qsort");
}

int cmp_ascendent(int *x, int *y)
{
  return( *x - *y);
}

int cmp_descendent(int *x, int *y)
{
  return( *y - *x );
}

/*****************************************************************************

                                K_SUB

Compute x minus y and return the result in resultat.

*****************************************************************************/
void k_sub( const kaprekar x, const kaprekar y, kaprekar resultat)
{
  check_kaprekar(x, "x in k_sub");
  check_kaprekar(y, "y in k_sub");
  check_kaprekar(resultat, "resultat in k_sub before computation");

  if( k_inf(x, y) )
  {
    fprintf(stderr, "Warning: x less than y in k_sub\n");
    fprintf(stderr, "x = %d y = %d\n", k_to_int(x), k_to_int(y) );
  }

  int_to_k( k_to_int(x) - k_to_int(y), resultat );

  check_kaprekar(resultat, "resultat in k_sub after computation");
}

/*****************************************************************************

                                K_NEXT

Compute the next Kaprekar number.
Return FAUX is if Kaprekar constant or 0000 is encountered

*****************************************************************************/
booleen k_next(kaprekar x)
{
  kaprekar x_asc, x_des;
  check_kaprekar(x, "x in k_next before computing");
  k_cpy(x, x_asc);
  k_cpy(x, x_des);
  k_sort(x_asc, ASCENDENT);
  k_sort(x_des, DESCENDENT);
  k_sub(x_des, x_asc, x);
  check_kaprekar(x, "x in k_next after computing");
  if( k_egal(x, k_const) || k_egal(x, k_nul) )
    return(FAUX);
  else
    return(VRAI);
```

```
}

/*****************************************************************************

                                K_PRINT

*****************************************************************************/
void k_print(const kaprekar x)
{
  int i;
  check_kaprekar(x, "x in k+print");
  for( i = 0 ; i < N ; i++ )
    printf("%1d",x[i]);
  printf(" ");
}

/*****************************************************************************

                                MAIN

*****************************************************************************/
int main(void)
{
  int i,j,k,l;
  kaprekar graine;
  booleen continuer;

  for( i = 0 ; i < 10 ; i++ )
    for( j = i ; j < 10 ; j++ )
      for( k = j ; k < 10 ; k++ )
        for( l = k ; l < 10 ; l++ )
        {
          k_set(graine, i, j, k, l);
          printf("#");
          k_print( graine );
         do
         {
           continuer = k_next(graine);
           k_print( graine );
         }
         while( continuer );
         printf("\n");
      }
  return( EXIT_SUCCESS );
}
```

4 Solution en C (obfuscated)

Ce programme fait la même chose que le précédent, mais le code source est nettement moins bien lisible.

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#define H printf
#define I sprintf
#define J sscanf
#define L *(char *)
#define K const void *
#define C(D) strncpy(D,_2,5);
#define E(F,G) for(F=G;F<10;F++)
#define A(B) _1*=-1;qsort(B,4,sizeof(char),_);
int _1=-1,i,j,k,l,x,y;char _2[5],_3[5],_4[5];
int _(K a,K b){return(_1*(L a-L b));}
main(){ E(i,0) E(j,i) E(k,j) E(l,k) {
I(_2,"%1d%1d%1d%1d\0",i,j,k,l);H("#%s",_2);
do { C(_3) C(_4) A(_3) A(_4) J(_3,"%d",&x);
J(_4,"%d",&y);I(_2,"%04d",y-x);H(" %s",_2);
if(y-x==6174)break;} while(y-x);H("\n");}}
```

5 Solution en S

```
nextKaprekar <- function(input) {
    if (!is.character(input))
        stop("string expected")
    x <- as.integer(strsplit(input, split = "")[[1]])
    if (!all(is.finite(x)))
        stop("non numeric digits in input")
    if (length(x) != 4)
        stop("wrong number of digits in input")
    petit <- as.integer(paste(sort(x), collapse = ""))
    grand <- as.integer(paste(sort(x, decreasing = TRUE), collapse = ""))
    return(formatC(grand - petit, width = 4, flag = "0"))
}
nextKaprekar("6174")
```

```
[1] "6174"
```

```
iterateKaprekar <- function(seed) {
    cat("#", seed, " ", sep = "")
    repeat {
        seed <- nextKaprekar(seed)
        cat(seed, " ", sep = "")
        if (seed == "6174" || seed == "0000") {
            cat("\n")
            break
        }
    }
}
iterateKaprekar("3015")
```

```
#3015 5175 5994 5355 1998 8082 8532 6174
```

```
for (seed in formatC(0:10, width = 4, flag = 0)) iterateKaprekar(seed)
```

```
#0000 0000
#0001 0999 8991 8082 8532 6174
#0002 1998 8082 8532 6174
#0003 2997 7173 6354 3087 8352 6174
#0004 3996 6264 4176 6174
#0005 4995 5355 1998 8082 8532 6174
#0006 5994 5355 1998 8082 8532 6174
#0007 6993 6264 4176 6174
#0008 7992 7173 6354 3087 8352 6174
#0009 8991 8082 8532 6174
#0010 0999 8991 8082 8532 6174
```

Références

[1] D. R. Kaprekar. An interesting property of the number 6174. *Scripta Mathematika*, 15 :244–245, 1955.