

ISIMA PREMIÈRE ANNÉE

COMPTE-RENDU DE TP
STRUCTURES DE DONNÉES

Dérécursification à l'aide d'une pile

Benjamin BARBESANGE
Pierre-Loup PISSAVY
Groupe G21

Enseignant :
Michelle CHABROL

mars 2015



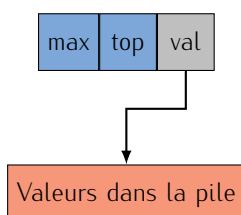
1 | Présentation

Le but de ce TP est de dérécurser une fonction à l'aide d'une pile. Les fonctions de gestion d'une pile seront ainsi créées. Les opérations suivantes sont permises avec la pile :

- Initialiser la pile,
- Libérer la pile,
- Tester si la pile est vide,
- Tester si la pile est pleine,
- Retourner l'élément en haut de la pile,
- Afficher l'élément en haut de la pile,
- Insérer un élément dans la pile.

1.1 Structure de données employée

(a) Structure utilisée



(b) Code

```
20 typedef struct _stack_t {
21     int      max; /* nombre max d'elements dans la pile */
22     int      top; /* position de l'element en tete de pile */
23     datatype *val; /* tableau des valeurs de la pile */
24 } stack_t;
```

FIGURE 1.1 – Structure de pile et code correspondant

1.2 Organisation du code source

Nous avons défini deux modules, le premier contient une fonction sous forme récursive ainsi que sa version sous forme itérative. Nous disposons également d'un module permettant de gérer une pile, qui est ainsi utilisé lors de la dérécursivation de la fonction.

1.2.1 Gestion de la pile

- `src/stack.h`
- `src/stack.c`

1.2.2 Fonction récursive

- `src/truc.h`
- `src/truc.c`

1.2.3 Programme principal

- `src/main.c`

2 | Détails du programme

2.1 Gestion de la pile

Code C

```
1  /* stack.h
2  Header
3
4  -----| DERECURSIVATION DE FONCTION PAR PILE |-----
5
6  BARBESANGE Benjamin,
7  PISSAVY Pierre-Loup
8
9  ISIMA 1ere Annee, 2014-2015
10 */
11
12 #ifndef __STACK__H
13 #define __STACK__H
14
15 #include <stdio.h>
16 #include <stdlib.h>
17
18 typedef int datatype; /* permet d'utiliser des types differents avec la pile */
19
20 typedef struct _stack_t {
21     int max; /* nombre max d'elements dans la pile */
22     int top; /* position de l'element en tete de pile */
23     datatype *val; /* tableau des valeurs de la pile */
24 } stack_t;
25
26 int init(stack_t *,int);
27 void supp(stack_t *);
28 int empty(stack_t);
29 int full(stack_t);
30 int pop(stack_t *, datatype *);
31 int top(stack_t *, datatype *);
32 int push(stack_t *, datatype);
33
34 #endif
```

```

1  /* stack.c
2  Fonctions de gestion de la structure de pile
3
4  -----| DERECURSIVATION DE FONCTION PAR PILE |-----
5
6  BARBESANGE Benjamin,
7  PISSAVY Pierre-Loup
8
9  ISIMA 1ere Annee, 2014-2015
10 */
11 #include "stack.h"
12
13 /* int init(stack_t *p, int n)
14 Fonction d'initialisation de la pile avec une taille max
15
16 Entrees :
17     *p : pointeur sur la pile
18     n : taille maximum de la pile
19
20 Sortie :
21     int : code d'erreur
22         1 si aucune erreur
23         0 si erreur de creation de la pile
24 */
25 int init(stack_t *p, int n) {
26     int ret = 1;
27     p->max = n;
28     p->top = -1;
29     p->val = (int*) malloc(n*sizeof(int));
30     if (p->val == NULL) {
31         ret = 0;
32     }
33     return ret;
34 }
35
36 /* void supp(stack_t *p)
37 Fonction de suppression de la pile
38
39 Entree :
40     *p : pointeur sur la tete de la pile
41
42 Sortie :
43     Aucune
44 */
45 void supp(stack_t *p) {
46     free(p->val);
47 }
48
49 /* int empty(stack_t *p)
50 Teste si la pile est vide ou non
51
52 Entree :
53     p : tete de la pile
54
55 Sortie :

```

```

56     int : boolean
57         0 si la pile n'est pas vide
58         1 si la pile est vide
59 */
60 int empty(stack_t p) {
61     return (p.top == -1)?1:0;
62 }
63
64 /* int full(stack_t p)
65 Teste si la pile est pleine ou non
66
67 Entree :
68     p : tete de la pile
69
70 Sortie :
71     int : boolean
72         0 si la pile n'est pas pleine
73         1 si la pile est pleine
74 */
75 int full(stack_t p) {
76     return (p.top == p.max-1)?1:0;
77 }
78
79 /* int pop(stack_t *p, datatype *v)
80 Recupere le premier element de la pile (et l'enleve) et retourne un code d'erreur
81
82 Entree :
83     *p : pointeur sur la tete de la pile
84     *v : pointeur sur un element du type de la pile, variable en I/O
85
86 Sortie :
87     int : code d'erreur
88         0 si rien n'est retourne dans la variable v
89         1 si on a recupere l'element en tete
90 */
91 int pop(stack_t *p, datatype *v) {
92     int ok = 0;
93     if (!empty(*p)) {
94         *v = p->val[p->top];
95         ok = 1;
96         p->top--;
97     }
98     return ok;
99 }
100
101 /* int top(stack_t *p, datatype *v)
102 Retourne l'element en tete de la pile (sans l'enlever) et retourne un code d'erreur
103
104 Entree :
105     *p : pointeur sur la tete de la pile
106     *v : pointeur sur un element du type de la pile, variable en I/O
107
108 Sortie :
109     int : code d'erreur
110         0 si rien n'est retourne
111         1 si on recupere l'element en tete

```

```

112 */
113 int top(stack_t *p, datatype *v) {
114     int ok = 0;
115     if (!empty(*p)) {
116         *v = p->val[p->top];
117         ok = 1;
118     }
119     return ok;
120 }
121
122 /* int push(stack_t *p, datatype v)
123    Insere un element en tete de la pile
124
125    Entree :
126        *p : pointeur sur la tete de la pile
127        v : element a inserer dans la pile
128
129    Sortie :
130        int : code d'erreur
131            0 si l'element n'est pas ajoute dans la pile
132            1 si l'element est ajoute dans la pile
133 */
134 int push(stack_t *p, datatype v) {
135     int ok = 0;
136     if (!full(*p)) {
137         p->top++;
138         p->val[p->top] = v;
139         ok = 1;
140     }
141     return ok;
142 }

```

2.2 Fonction récursive

Code C

```

1  /* truc.h
2  Header
3
4  -----| DERECURSIVATION DE FONCTION PAR PILE |-----
5
6  BARBESANGE Benjamin,
7  PISSAVY Pierre-Loup
8
9  ISIMA 1ere Annee, 2014-2015
10 */
11 #ifndef __TRUC__H
12 #define __TRUC__H
13     #include <stdio.h>
14     #include <stdlib.h>
15     #define N 10
16     int TRUC(int, int);
17     int truc_iter(int, int);
18 #endif

```

```

1  /* truc.c
2     Fonction recursive et son equivalent en iteratif
3
4     -----| DERECURSIVATION DE FONCTION PAR PILE |-----
5
6     BARBESANGE Benjamin,
7     PISSAVY Pierre-Loup
8
9     ISIMA 1ere Annee, 2014-2015
10  */
11
12  #include "truc.h"
13  #include "stack.h"
14
15  /* tableau des valeurs pour la decomposition, premiere valeur inutile */
16  int P[N+1] = {0,1,3,2,0,5,2,7,1,9,1};
17
18  /* int TRUC(int S, int I)
19     Fonction sous forme recursive qui affiche la decomposition de
20     S en I entiers pris a partir d'un tableau d'entiers (defini ici en statique)
21
22     Entrees :
23         int S : Nombre a decomposer
24         int I : Nombre d'entiers utilises pour decomposer S
25
26     Sortie :
27         int : entier sous forme de booleen
28             0 si on a pas pu decomposer S exactement
29             1 sinon
30  */
31  int TRUC(int S, int I) {
32      if (S == 0) {
33          return 1;
34      } else if (S < 0 || I > N) {
35          return 0;
36      } else if (TRUC(S-P[I],I+1)) {
37          printf("%d\n",P[I]);
38          return 1;
39      } else {
40          return TRUC(S,I+1);
41      }
42  }
43
44  /* int truc_iter(int s, int i)
45     Meme fonction qu'au dessus, mais sous forme iterative
46
47     Entrees :
48         int s : Nombre a decomposer
49         int i : Nombre d'entiers utilises pour decomposer S
50
51     Sortie :
52         int : entier sous forme de booleen
53             0 si on a pas pu decomposer S exactement
54             1 sinon
55  */

```



```

56 int truc_iter(int s, int i) {
57     int sl = s;
58     int il = i;
59     int r = 0;
60     stack_t p;
61     if (init(&p,N)) {
62         do {
63             while (sl > 0 && il <= N) {
64                 push(&p,il);
65                 sl -= P[il];
66                 ++il;
67             }
68             if (sl == 0) {
69                 r = 1;
70                 while (!empty(p)) {
71                     pop(&p,&il);
72                     sl += P[il];
73                     printf("%d\n",P[il]);
74                 }
75             } else {
76                 r = 0;
77                 if (!empty(p)) {
78                     pop(&p,&il);
79                     sl += P[il];
80                     ++il;
81                 }
82             }
83         } while (!empty(p));
84         supp(&p);
85     }
86     return r;
87 }

```

2.3 Programme principal

Code C

```
1  /* main.c
2     Fonction principale du programme, pour les tests
3
4     -----| DERECURSIVATION DE FONCTION PAR PILE |-----
5
6     BARBESANGE Benjamin,
7     PISSAVY Pierre-Loup
8
9     ISIMA 1ere Annee, 2014-2015
10 */
11
12 #include "truc.h"
13 #include "stack.h"
14
15 extern int P[];
16
17 void pile_test(int);
18 void afficher_P();
19
20 int main(int argc, char *argv[]) {
21     FILE *f;
22     char buf[100];
23     int i, s, n;
24     if (argc > 1) {
25         /* Lecture de fichier de commandes */
26         f = fopen(argv[1], "r");
27         if (f) {
28             while (!feof(f)) {
29                 buf[0] = '\0';
30                 fgets(buf, 100, f);
31                 switch (buf[0]) {
32                     case 'p': /* pile */
33                         sscanf(&buf[1], "%d", &n);
34                         pile_test(n);
35                         break;
36                     case 't': /* truc iteratif */
37                         afficher_P();
38                         sscanf(&buf[1], "%d %d", &s, &i);
39                         printf("Reponse: %s\n", (truc_iter(s, i)) ? "Vrai" : "Faux");
40                         break;
41                     case 'T': /* truc recursif */
42                         afficher_P();
43                         sscanf(&buf[1], "%d %d", &s, &i);
44                         printf("Reponse: %s\n", (TRUC(s, i)) ? "Vrai" : "Faux");
45                         break;
46                     case '#': /* texte */
47                         printf("%s", &buf[1]);
48                         break;
49                     default:
50                         puts("\n");
51                 }
52             }
53         } else {
```

```

54     fprintf(stderr, "Fichier invalide\n");
55 }
56 } else {
57     /* Fonctions de tests de base */
58     printf("Test de pile:\n");
59     pile_test(4);
60     printf("Fonction recursive:\n");
61     TRUC(9,1);
62     printf("Fonction iterative:\n");
63     truc_iter(9,1);
64 }
65 return 0;
66 }
67
68
69 void pile_test(int n) {
70     stack_t p;
71     int i = 0;
72     if (init(&p,n)) {
73         while (push(&p,i)) {
74             printf("Empiler: %d\n",i);
75             ++i;
76         }
77         while (pop(&p,&i)) {
78             printf("Depiler: %d\n",i);
79         }
80         supp(&p);
81     }
82 }
83
84 void afficher_P() {
85     int i;
86     printf("Tableau P : ");
87     for (i = 1; i <= N; ++i) {
88         printf("%d ",P[i]);
89     }
90     printf("\n");
91 }

```

3 | Principes et lexiques des fonctions

Dans cette partie, sont décrits les algorithmes de principe associés aux fonctions écrites en langage C, ainsi qu'un lexique concernant les variables intermédiaires des fonctions.

Les lexiques des variables d'entrée, sortie et entrée/sortie sont disponibles dans le code source directement.

3.1 Gestion de la pile

La gestion de la pile s'effectue grâce aux fichiers `stack.c` et `stack.h`.

3.1.1 init

Algorithme init

Début

```
On initialise le code de retour à 1; [ Il n'y a aucune erreur ]
On initialise la taille max de la pile;
On initialise l'indice du haut de la pile à -1; [ Pour indiquer qu'elle est vide ]
On alloue l'espace de la pile;
Si l'allocation n'est pas réussie Alors
    Le code d'erreur passe à 0;
FinSi;
Retourner code d'erreur;
```

Fin

Lexique :

```
ret: code d'erreur, 0 si il y a une erreur de création de la pile, 1 sinon
```

3.1.2 supp

Ici, nous libérons simplement de tableau de valeurs de la pile, puisque celui-ci est alloué dynamiquement lors de la création.

3.1.3 empty

Cette fonction teste simplement si l'indice du haut de la pile est -1 , ce qui veut dire qu'il n'y a aucun élément dans la pile. Ainsi la valeur 1 sera retournée. Sinon la valeur 0 est retournée.

3.1.4 full

Cette fonction vérifie si l'indice de l'élément en haut de la pile est égal à la taille max de la pile (moins 1, car les tableaux commencent à 0). Si c'est le cas, on renvoie 1 pour signaler que la pile est pleine, et 0 sinon.

3.1.5 pop

Algorithme pop

Début

```
On initialise le code de retour à 0; [ On n'a pas dépilé ]  
Si la pile n'est pas vide Alors  
    On dépile l'élément dans une variable en Input/Output;  
    Le code de retour passe à 1; [ On a dépilé et récupéré l'élément ]  
    On modifie l'indice de l'élément en haut de la pile; [ On retranche 1 à l'indice précédent ]  
FinSi;  
Retourner code de retour;
```

Fin

Lexique :

ok: code de retour, 0 si on n'a pas dépilé, 1 si on a dépilé la valeur en haut de la pile

3.1.6 top

Algorithme top

Début

```
On initialise le code de retour à 0; [ Pas d'élément lu ]  
Si la pile n'est pas vide Alors  
    On copie la tête de pile dans une variable en I/O;  
    Le code d'erreur passe à 1; [ On a dépilé ]  
FinSi;  
Retourner code de retour;
```

Fin

Lexique :

| ok: code de retour, 0 si on n'a pas récupéré l'élément en haut de la pile, 1 si on l'a récupéré

3.1.7 push

Algorithme push

Début

| On initialise le code de retour à 0; [L'élément n'est pas ajouté dans la pile]

| Si la pile n'est pas pleine Alors

| | On incrémente l'indice de l'élément en haut de la pile;

| | On place l'élément dans le tableau de la pile, à l'indice précédemment modifié;

| | Le code de retour passe à 1; [L'élément est empilé]

| FinSi;

| Retourner code de retour;

Fin

Lexique :

| ok: code de retour, 0 si on n'a pas empilé, 1 si on a empilé la valeur

3.2 Dérécursivation de la fonction

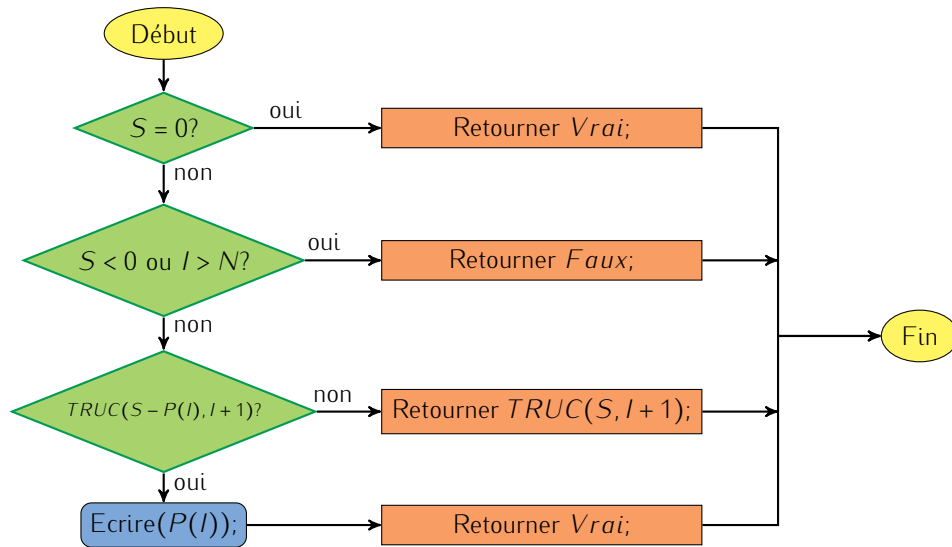
La fonction récursive ainsi que sa version itérative se trouvent dans les fichiers `truc.c` et `truc.h`.

3.2.1 TRUC

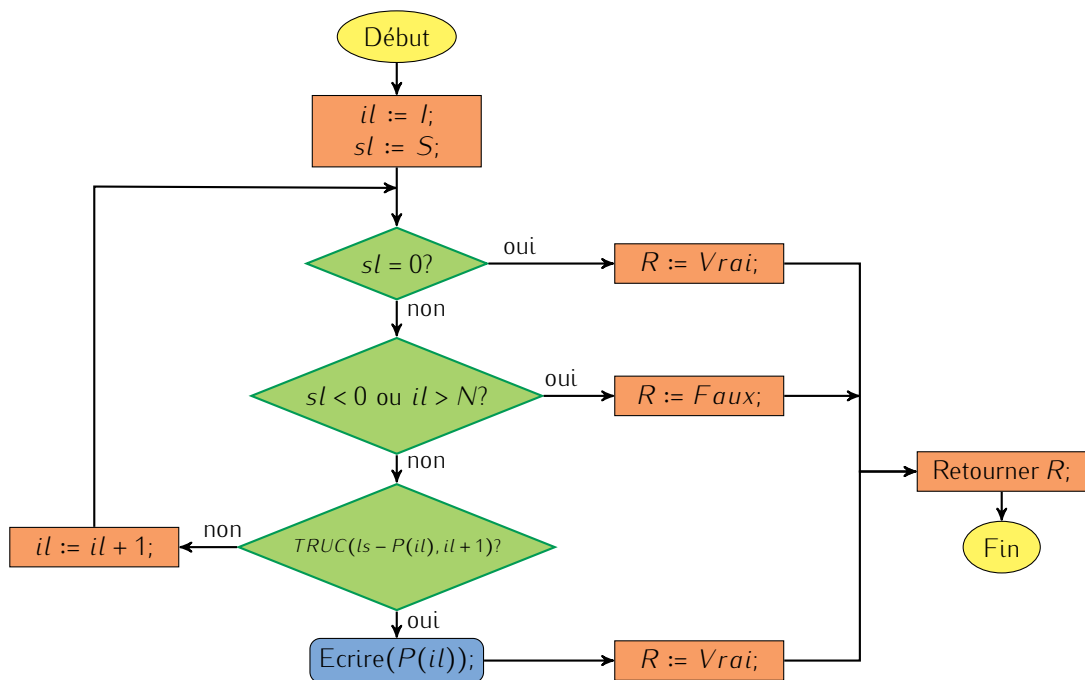
Cette fonction étant l'énoncé du TP, nous ne détaillerons ainsi ni le principe ni les variables utilisées dans cet algorithme.

Nous allons dérécurser cette fonction. Il s'agit d'un traitement simplifié du problème du sac à dos.

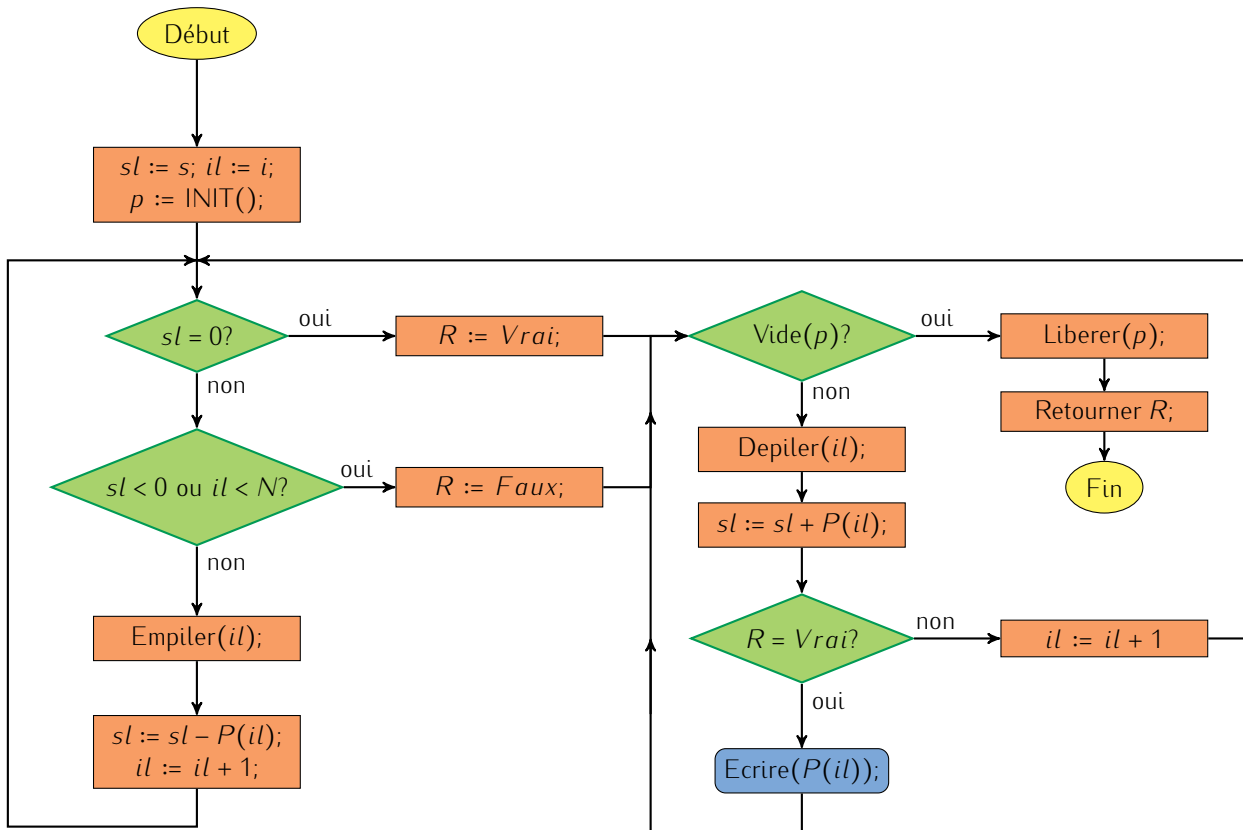
Logigramme initial



Suppression des appels terminaux



Suppression des appels non-terminaux



3.2.2 truc_iter

Algorithme truc_iter (Principe)

Début

Copie des paramètres d'entrée dans des variables locales, sl et il;
Initialisation de la pile de la même taille que le tableau statique;

Si l'initialisation a réussi Alors

Répéter

TantQue $sl > 0$ Et $il \leq N$ Faire

On envoie il dans la pile;

$sl := sl - P(il)$;

On incrémente il;

FinTantQue;

Si $sl = 0$ Alors

Le booléen de retour est à Vrai;

TantQue la pile n'est pas vide Faire

On récupère il à partir de la pile;

On ajoute $P(il)$ à sl;

On affiche $P(il)$;

FinTantQue;

Sinon

Le booléen de retour est à Faux;

Si la pile n'est pas vide Alors

On récupère il à partir de la pile;

On ajoute $P(il)$ à sl;

On incrémente il;

FinSi;

FinSi;

TantQue la pile n'est pas vide fait;

On supprime la pile;

FinSi;

Retourner Booléen de retour;

Fin

Lexique :

sl: copie locale du nombre s passé en paramètre. Représente le nombre à décomposer

il: copie locale du nombre i passé en paramètre. Représente le nombre d'entiers du tableau à utiliser pour décomposer s

r: booléen de retour, indique 1 si on a obtenue la somme s, 0 sinon

p: pile

P: tableau d'entiers, défini statiquement

N: taille du tableau P

4 | Compte rendu d'exécution

4.1 Makefile

```
1 #Compilateur et options de compilation
2 CC=gcc
3 CFLAGS=-Wall -ansi -pedantic -Wextra -g
4
5 #Fichiers du projet
6 SOURCES=main.c stack.c truc.c
7 OBJECTS=$(SOURCES:.c=.o)
8
9 EXEC=prog
10
11 $(EXEC): $(OBJECTS)
12     $(CC) $(CFLAGS) $^ -o $(EXEC)
13
14 .c.o:
15     $(CC) -c $(CFLAGS) $.c
16
17 clean:
18     rm $(OBJECTS) $(EXEC)
```

4.2 Jeux de tests

Exécution du programme avec le fichier suivant :

```
Test
#1) Test de pile avec 5 éléments
p 5

#Comparons les resultats donnees par les deux fonctions

#2) TRUC(15,1)
T 15 1

#3) truc(15,1)
t 15 1

#Exemples de resolutions impossibles

#4) truc(15,7)
```

t 15 7

#5) TRUC(256,1)

T 256 1

Resultat

1) Test de pile avec 5 éléments

Empiler: 0

Empiler: 1

Empiler: 2

Empiler: 3

Empiler: 4

Depiler: 4

Depiler: 3

Depiler: 2

Depiler: 1

Depiler: 0

Comparons les resultats donnees par les deux fonctions

2) TRUC(15,1)

Tableau P : 1 3 2 0 5 2 7 1 9 1

1

1

2

5

0

2

3

1

Reponse: Vrai

3) truc(15,1)

Tableau P : 1 3 2 0 5 2 7 1 9 1

1

1

2

5

0

2

3

1

Reponse: Vrai

Exemples de resolutions impossibles

4) truc(15,7)

Tableau P : 1 3 2 0 5 2 7 1 9 1

Reponse: Faux

5) TRUC(256,1)

Tableau P : 1 3 2 0 5 2 7 1 9 1

Reponse: Faux

Passage dans l'outil valgrind

```
Resultat Valgrind
==7438== Memcheck, a memory error detector
==7438== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==7438== Using Valgrind-3.10.0.SVN and LibVEX; rerun with -h for copyright info
==7438== Command: ../src/prog
==7438==
Test de pile:
Empiler: 0
Empiler: 1
Empiler: 2
Empiler: 3
Depiler: 3
Depiler: 2
Depiler: 1
Depiler: 0
Fonction recursive:
1
2
0
2
3
1
Fonction iterative:
1
2
0
2
3
1
==7438==
==7438== HEAP SUMMARY:
==7438==    in use at exit: 0 bytes in 0 blocks
==7438==   total heap usage: 2 allocs, 2 frees, 56 bytes allocated
==7438==
==7438== All heap blocks were freed -- no leaks are possible
==7438==
==7438== For counts of detected and suppressed errors, rerun with: -v
==7438== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```