Introduction à la Programmation des Algorithmes

3.2. Langage C – Fonctions récursives

François Fleuret

https://fleuret.org/11x001/

UNIVERSITÉ
DE GENÈVE

On peut définir des fonctions mathématiques de manière récursive.

Par exemple la fonction factorielle:

$$f(n) = \begin{cases} 1 & \text{si } n <= 0, \\ n \cdot f(n-1) & \text{sinon.} \end{cases}$$

Des constructions similaires sont possibles en programmation.

```c
#include <stdio.h>

int factorielle(int n) {
  if(n <= 0) return 1;
  else return n * factorielle(n - 1);
}

int main(void) {
  for(int k = 0; k <= 5; k++) {
    printf("%d! = %d\n", k, factorielle(k));
  }
  return 0;
}
```

affiche

```
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
```

Cela nous donne par substitution,

```
factorielle(4)
4 * factorielle(3)
4 * 3 * factorielle(2)
4 * 3 * 2 * factorielle(1)
4 * 3 * 2 * 1 * factorielle(0)
4 * 3 * 2 * 1 * 1
12 * 2 * 1 * 1
24 * 1 * 1
24 * 1
24
```

Une fonction récursive doit intégrer une **condition d'arrêt** qui retourne une valeur sans appel récursif.

```
1  int factorielle(int n) {
2    if(n <= 0) return 1;
3    else return n * factorielle(n - 1);
4  }
```

Une fonction récursive doit intégrer une **condition d'arrêt** qui retourne une valeur sans appel récursif.

```
1  int factorielle(int n) {
2    if(n <= 0) return 1;
3    else return n * factorielle(n - 1);
4  }
```

Sans une telle condition, ou si elle n'est jamais réalisée, la fonction ne s'arrête pas. Par exemple si nous écrivons une fonction pour calculer $\sum_{k=1}^{n} k$:

```
1  int somme(int n) {
2    if(n == 1) return 1;
3    else return n + somme(n - 1);
4  }
```

Une fonction récursive doit intégrer une **condition d'arrêt** qui retourne une valeur sans appel récursif.

```
1  int factorielle(int n) {
2    if(n <= 0) return 1;
3    else return n * factorielle(n - 1);
4  }
```

Sans une telle condition, ou si elle n'est jamais réalisée, la fonction ne s'arrête pas. Par exemple si nous écrivons une fonction pour calculer $\sum_{k=1}^{n} k$:

```
1  int somme(int n) {
2    if(n == 1) return 1;
3    else return n + somme(n - 1);
4  }
```

Elle ne s'arrête pas si elle est appelée avec `n<=0`.

Une bonne habitude de programmation est de vérifier que les valeurs des arguments ont bien les propriétés attendues et à réagir en conséquence si ce n'est pas le cas. Une mesure brutale consiste à appeler `abort`, définie dans `stdlib.h`, qui crash le programme.

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

int somme(int n) {
  if(n <= 0) abort();
  if(n == 1) return 1;
  else return n + somme(n - 1);
}

int main(void) {
  printf("%d\n", somme(10));
  printf("%d\n", somme(-3));
  return 0;
}
```

affiche

```
55
Aborted (core dumped)
```

Il arrive que l'on veuille programmer plusieurs fonctions qui s'utilisent mutuellement. On parle alors de **récursion mutuelle**.

On doit alors déclarer l'une d'elle avant de les définir.

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

int impair(int n);

int pair(int n) {
  if(n < 0) abort();
  else if(n == 0) return 1;
  else return impair(n-1);
}

int impair(int n) {
  if(n < 0) abort();
  else if(n == 0) return 0;
  else return pair(n-1);
}

int main(void) {
  printf("%d %d %d\n", pair(13), pair(4), impair(6));
  return 0;
}
```

Une convention importante en C pour l'évaluation d'opérateurs booléens est que les opérandes ne sont elles-mêmes calculées que si cela est nécessaire. On parle d'évaluation **paresseuse**.

Par exemple pour évaluer

```
0 && f(x)
```

comme l'évaluation se fait de gauche à droite, il n'est pas nécessaire de calculer `f(x)`.

Une convention importante en C pour l'évaluation d'opérateurs booléens est que les opérandes ne sont elles-mêmes calculées que si cela est nécessaire. On parle d'évaluation **paresseuse**.

Par exemple pour évaluer

```
0 && f(x)
```

comme l'évaluation se fait de gauche à droite, il n'est pas nécessaire de calculer `f(x)`.

Si nous définissons

```
1  int divise(int a, int b) {
2    printf("Est-ce que %d divise %d?\n", b, a);
3    return a%b == 0;
4  }
```

```c
int main(void) {
  printf("calcul de q\n");
  int q = divise(10, 5) && divise(20, 4);
  printf("calcul de r\n");
  int r = divise(10, 3) && divise(20, 4);
  printf("calcul de t\n");
  int t = divise(10, 3) || divise(20, 4);
  printf("calcul de u\n");
  int u = divise(10, 2) && divise(20, 4) && divise(6, 4) && divise(15, 3);
  return 0;
}
```

affiche

```
calcul de q
Est-ce que 5 divise 10?
Est-ce que 4 divise 20?
calcul de r
Est-ce que 3 divise 10?
calcul de t
Est-ce que 3 divise 10?
Est-ce que 4 divise 20?
calcul de u
Est-ce que 2 divise 10?
Est-ce que 4 divise 20?
Est-ce que 4 divise 6?
```

Cela nous permet par exemple de re-écrire

```
1  int pair(int n) {
2    if(n < 0) abort();
3    return n == 0 || impair(n-1);
4  }
5
6  int impair(int n) {
7    if(n < 0) abort();
8    return n > 0 && pair(n-1);
9  }
```

Notez que ces fonctions étaient des exemples récursifs très inefficaces. En pratique on écrirait:

```
1  int pair(int n) {
2    if(n < 0) abort();
3    return n%2 == 0;
4  }
5
6  int impair(int n) {
7    if(n < 0) abort();
8    return n%2 == 1;
9  }
```

Récursivité terminale

Nous avons vu que les variables locales sont réservées sur une “pile” en mémoire. Ce mécanisme est le même pour les variables locales à une fonction.

En plus des variables, quand une fonction est appelée, l'ordinateur mémorise sur cette pile des informations nécessaires pour savoir où continuer l'exécution quand la fonction se termine.

Cela signifie qu'en pratique un programme occupe pour cela une quantité de mémoire proportionnelle au nombre d'appels de fonctions imbriqués.

Cela peut poser problème si on fait beaucoup d'appels récursifs

```c
#include <stdio.h>

/* retourne le nombre de diviseurs de n qui sont >= k */
int nb_diviseurs(int n, int k) {
  if(k > n) return 0;
  int nb = nb_diviseurs(n, k + 1);
  if(n % k == 0) nb++;
  return nb;
}

int main(void) {
  printf("%d\n", nb_diviseurs(245, 1));
  printf("%d\n", nb_diviseurs(317322, 1));
  printf("%d\n", nb_diviseurs(8133453, 1));
  return 0;
}
```

affiche

```
6
36
Segmentation fault (core dumped)
```

Ce problème peut être évité dans une fonction **si aucune opération n'est nécessaire entre le retour de l'appel récursif et le retour de la fonction elle même.**

Puisque l'ordinateur n'a pas besoin de repasser par la fonction après l'appel, il peut revenir directement après le premier appel à cette fonction récursive, et ne conserve rien sur la pile après chaque appel récursif.

On parle alors de fonction **récursive terminale.**

Voici une version récursive terminale:

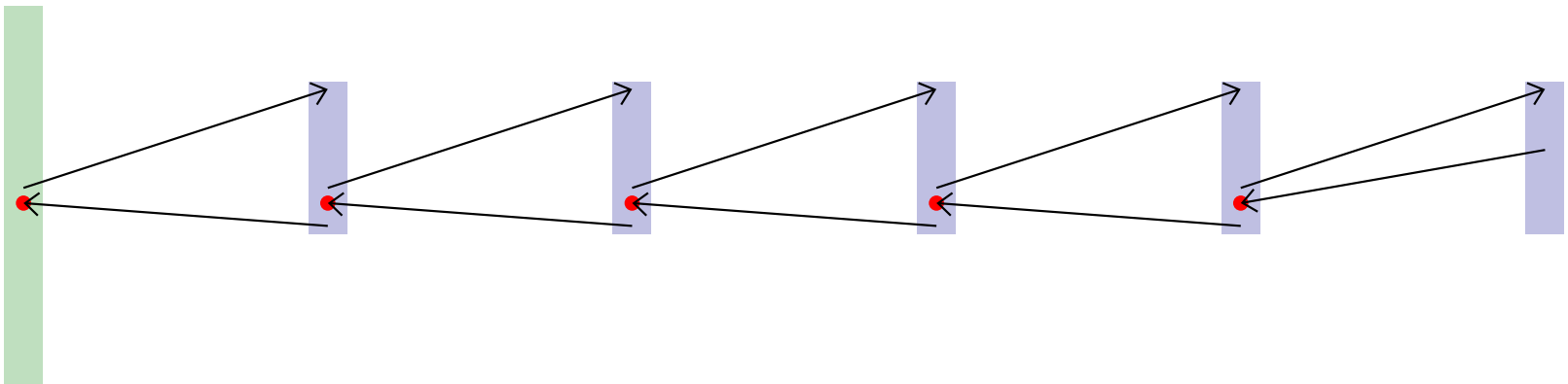
```
1   #include <stdio.h>
2
3   /* est-ce que n a un diviseur >= k ? */
4   int a_un_diviseur(int n, int k) {
5     if(k > n) return 0;
6     if(n%k == 0) return 1;
7     return a_un_diviseur(n, k+1);
8   }
9
10  int main(void) {
11    printf("%d\n", a_un_diviseur(479001599, 2));
12    return 0;
13  }
```

affiche

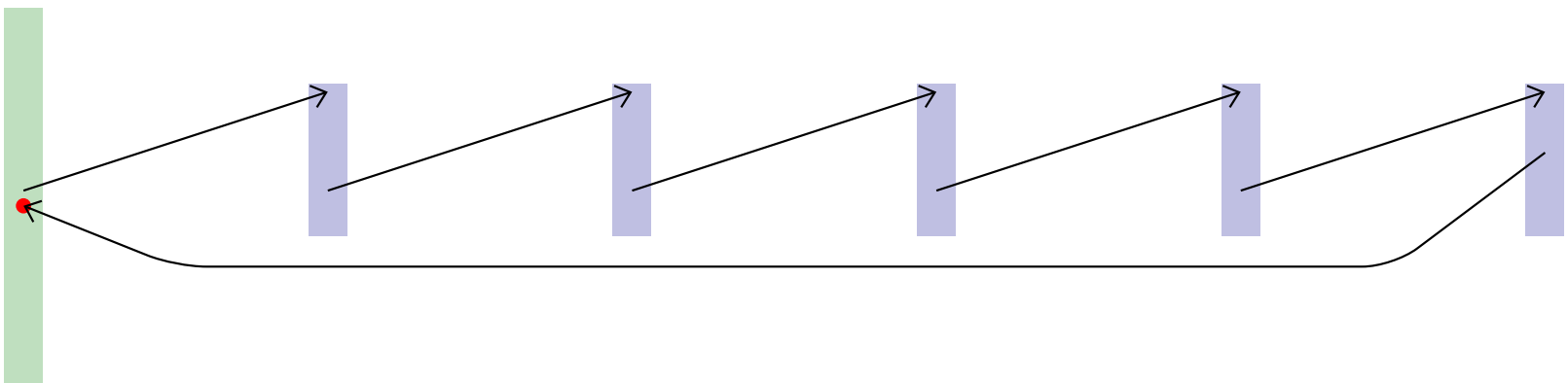
0

La fonction `a_un_diviseur` appelée ligne 11 entraînera un grand nombre d'appels récursifs, mais dès que l'un des returns des lignes 5 ou 6 sera exécuté, le programme pourra retourner directement à la ligne 11, sans repasser un grand nombre de fois ligne 7.

Récursion



Récursion terminale



Il est généralement possible de re-écrire une fonction récursive non-terminale sous une forme récursive terminale en rajoutant des arguments qui représentent des quantités accumulées.

Cette fonction n'était pas récursive terminale, d'où le crash quand elle a été appelée avec une grosse valeur:

```
1  int nb_diviseurs(int n, int k) {
2    if(k > n) return 0;
3    int nb = nb_diviseurs(n, k + 1);
4    /* ici nb est le nombres de diviseurs de n qui sont >= k+1 */
5    if(n % k == 0) nb++;
6    /* ici nb est le nombres de diviseurs de n qui sont >= k */
7    return nb;
8  }
```

Par exemple:

```c
#include <stdio.h>

/* retourne nb + le nombre de diviseurs de n qui sont >= k de façon à
ce que si on l'appelle avec nb = le nombre de diviseurs de n qui sont
< k elle retourne le nombre total de diviseurs de n */

int nb_diviseurs(int n, int k, int nb) {
  if(k > n) return nb;
  /* ici nb est le nombre de diviseurs de n qui sont < k */
  if(n % k == 0) nb++;
  /* ici nb est le nombre de diviseurs de n qui sont <= k */
  return nb_diviseurs(n, k + 1, nb);
}

int main(void) {
  printf("%d\n", nb_diviseurs(245, 1, 0));
  printf("%d\n", nb_diviseurs(317322, 1, 0));
  printf("%d\n", nb_diviseurs(8133453, 1, 0));
  return 0;
}
```

affiche

```
6
36
16
```

On peut cacher les arguments supplémentaires en créant une fonction qui exécute le premier appel récursif:

```c
/* retourne nb + le nombre de diviseurs de n qui sont >= k */
int nb_diviseurs_rec(int n, int k, int nb) {
  if(k > n) return nb;
  if(n % k == 0) nb++;
  return nb_diviseurs_rec(n, k + 1, nb);
}

int nb_diviseurs(int n) {
  return nb_diviseurs_rec(n, 1, 0);
}
```

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

float racine_rec(float x, float a, float b, float eps) {
  float c = (a + b) / 2;
  if(b - a < eps) return c;
  else
    if(c * c >= x)
      return racine_rec(x, a, c, eps);
    else
      return racine_rec(x, c, b, eps);
}

float racine(float x) {
  if(x < 0) abort();
  return racine_rec(x, 0, x+1, 1e-6);
}

int main(void) {
  printf("%f\n", racine(2));
  return 0;
}
```

affiche

```
1.414213
```

Fin