1 Fonctions

Lorsque l'on écrit des programmes longs, on divise le code en blocs afin de mieux s'y retrouver. De plus, on se rend souvent compte que plusieurs blocs sont identiques ou presque. Les ***fonctions*** permettent de factoriser le code,en regroupant un bloc de code ayant une fonction spécifique sous un nom réutilisable.

Par exemple, écrivons un programme qui calcule le maximum entre 4 nombres a, b, c, d. L'idée est de regarder a, puis b, puis c, puis d, et de garder en mémoire le plus grand vu jusqu'à présent, dans une variable `max_temp` (comme **temp**oraire).

```c
#include <stdio.h>

int main(){
  float a = 3, b = -2, c = 0.5, d = 5 ;
  float max_temp = a;

  if (b > max_temporaire){
    max_temp = b;
  }
  if (c > max_temporaire){
    max_temp = c;
  }
  if (d > max_temporaire){
    max_temp = d;
  }

  printf("Le maximum de %f, %f, %f et %f est %f\n", a, b, c, d, max_temp);
  return 0;
}
```

On s'aperçoit vite que le code est très redondant, et effectue la même opération 3 fois de suite. L'utilisation de ***fonctions*** va permettre de réduire cette redondance et d'améliorer l'organisation. Écrivons une nouvelle version du code qui utilise une **fonction** `max2`, qui sert à calculer le maximum entre deux nombres :

```c
#include <stdio.h>

/* calcule et renvoie le maximum entre x et y */
float max2(float x, float y){
  if (x > y){
    return x;
  } else {
    return y;
  }
}

int main(){
  float a = 3, b = -2, c = 0.5, d = 5 ;
  float max_temp = a;

  max_temp = max2(b, max_temp);
  max_temp = max2(c, max_temp);
  max_temp = max2(d, max_temp);

  printf("Le maximum de %f, %f, %f et %f est %f\n", a, b, c, d, max_temp);
  return 0;
}
```

La fonction `max2` permet de calculer le maximum entre deux flottants. Elle **renvoie** une valeur (avec le mot clé **return**), donc l'instruction `max_temporaire = max2(b, max_temporaire);` a pour effet de stocker dans la variable `max_temporaire` le résultat de la fonction.

Les fonctions permettent d'améliorer l'organisation et la lisibilité du code, et le rendent plus ***modulaire***. Par exemple, si l'on veut que notre programme affiche le minimum plutôt que le maximum, il suffit de changer le sens de l'inégalité dans la fonction `max2`. En éliminant la redondance, on diminue le risque d'introduire des bugs.

A Syntaxe des fonctions

Définition 1

La syntaxe d'une fonction est la suivante :

```
type_retour nom_fonction(type_1 param_1, ..., type_k param_k){
  /* corps de
  la fonction */
  return valeur_retour;
}
```

On spécifie donc le type de la valeur calculée par la fonction, appelé ***type de retour*** et le type de chacun des ***paramètres*** de la fonction. Le n-uplet (`type_1`, ..., `type_k`, `type_retour`) s'appelle la ***signature*** de la fonction, ou par abus de langage son type.
On notera parfois `type_1` $\times \ldots \times$ `type_k` $\to$ `type_retour`.

Exercice 1

Quelle est la signature de la fonction suivante ?

```
float bla(int a, int b, float c){
  /*
    ...
   */
}
```

Dans le code d'une fonction, utiliser l'instruction ***return*** fera sortir de la fonction, et retourner à l'endroit où la fonction a été appelée, avec la valeur spécifiée prête à être utilisée ou stockée dans une variable.
Une fonction peut avoir plusieurs instructions ***return*** pas nécessairement à la fin du code.

Exemple 1

```
/* renvoie 1 si a divise b, 0 sinon */
int divise(int a, int b){
  if (b%a == 0){
    return 1;
  } else {
    return 0;
  }
}
```

En revanche, toutes les instructions return doivent retourner une valeur du type déclaré pour la fonction. Dans l'exemple précédent, `1` et `0` sont bien de type `int`.

Certaines fonction font quelque chose mais ne calculent pas de valeur de retour particulière.

Exemple 2

```c
// affiche le double de a
void afficher_double(float a){
  float b = 2 * a;
  printf("%d\n", b);
}
```

La fonction `afficher_double` écrit dans le terminal mais ne renvoie pas le résultat qu'elle a calculé : il n'y a pas de ***return***. Dans ce cas là, le type de retour de la fonction est ***void***. Ce type signifie "aucune valeur". On appelle parfois **procédure** les fonctions ayant le type de retour `void`.

NE PAS CONFONDRE AFFICHER ET RENVOYER !

Lorsqu'une fonction affiche quelque chose dans le terminal, la valeur affichée peut être lue par la personne qui a lancé le programme, mais cette valeur est ensuite perdue et devient inutilisable. Si une fonction calcule une valeur qui sera utile au reste du programme, elle doit la **renvoyer** avec le mot clé **return**.

Exemple 3

Le code suivant ne fonctionne pas :

```c
#include <stdio.h>

int ajoute_un(int x){
  printf("%d\n", 1+x);
}

int main(){
  int y = ajoute_un(3);
  printf("%d\n", y); // à l'exécution, affiche n'importe quoi
}
```

En effet, à la compilation, on voit apparaître un avertissement : la fonction `ajout_un` est sensée renvoyer un `int`, mais elle n'a pas d'instruction `return`. La valeur affichée n'est **pas** renvoyée, on ne peut plus la récupérer une fois l'exécution de la fonction terminée.

Documentation Lorsqu'on écrit une fonction, on doit **SYSTEMATIQUEMENT** inclure un **commentaire de documentation**, qui décrit ce que la fonction fait. Un commentaire de fonction doit faire apparaître le nom de tous les paramètres et expliquer ce que renvoie la fonction, ce qu'elle affiche, ce qu'elle modifie, etc... Cette description doit être précise, et utile : quelqu'un qui le le commentaire de documentation d'une fonction doit pouvoir l'utiliser même s'il n'a pas lu le code de la fonction.

Exemple 4

Une fonction et son commentaire :

```c
/* renvoie x/y si y est non nul, sinon
   affiche un message d'avertissement et renvoie 0 */
float safe_division(float x, float y){
  if (y == 0){
    printf("Attention: division par 0\n");
    return 0;
  } else {
    return x/y;
  }
}
```

B Appel de fonction

Lorsque l'on utilise une fonction que l'on a défini, on dit que l'on ***appelle*** la fonction.

Exemple 5

```c
#include <stdio.h>

// renvoie la moyenne de a et b
float moyenne(float a, float b){
  return (a+b)/2;
}

int main(){
  float x = 18.77, y = 15.36;

  float c = moyenne(2*x, y+3);

  return 0;
}
```

On dit que l'on ***appelle*** `moyenne` sur `x` et `y`. On dit également que `x` et `y` sont les ***arguments*** de la fonction.

NE PAS CONFONDRE PARAMÈTRE ET ARGUMENT !

Les paramètres désignent les variables utilisées dans la définition de la fonction, tandis que les arguments désignent les valeurs fournies en entrée quand on appelle la fonction

Dans l'exemple au dessus, `a` et `b` sont les paramètres de la fonction `moyenne`, et lorsqu'on appelle la fonction dans le main, les arguments sont les valeurs de `2*x` et `y+3`.

2 Portée des variables

Définition 2

La portée d'une variable est l'ensemble des lignes d'un programme où cette variable existe, c'est à dire où l'on peut y faire référence.

En C, les variables peuvent être locales, auquel cas leur portée se limite à une unique fonction, ou globale, auquel cas leur portée est l'ensemble du programme.

A Variable locale

Reprenons l'exemple de la fonction `moyenne` :

```c
#include <stdio.h>

float moyenne(float a, float b){
  return (a+b)/2;
}

int main(){
  float x = 18.77, y = 15.36,

  float c = moyenne(2*x, y+3);

  return 0;
}
```

Les variables `a` et `b` n'existent que dans le corps de la fonction moyenne. On dit que ce sont des variables ***locales*** : impossible d'y accéder depuis le main. De même, les variables x, y et c sont des variables locales de la fonction `main` : impossible d'y accéder depuis le reste du code.

Exemple 6

Si l'on essaie de compiler le code suivant :

```c
#include <stdio.h>

void definition_x(){
  int x = 0;
}


int main(){
  definition_x();
  printf("%d\n", x);
  return 0;
}
```

On obtient une erreur :

In function 'main' : error : 'x' undeclared

Le compilateur nous averti que l'on utilise une variable qui n'a pas été déclarée avant. Donc, la déclaration dans la fonction `definition_x` est bien ***locale*** !

B Passage par valeur

Lorsque l'on appelle une fonction sur des arguments, les valeurs des arguments sont copiées et collées dans les paramètres. Ainsi, ***un appel de fonction ne peut pas modifier ses arguments***. On parle de ***passage par valeur***.

Exemple 7

Si l'on compile et exécute le code suivant :

```c
#include <stdio.h>

// modifie y pour lui assigner 0 ?
void modifier(int y){
  y = 0;
}

int main(){
  int x = 5;
  modifier(x);
  printf("%d\n", x);
  return 0;
}
```

Le programme va afficher 5, car la variable x du main n'aura pas été modifiée.

Remarque 1

Même en changeant le nom du paramètre de `modifier` en `int x`, rien ne change. En effet, le `x` de la fonction `modifier` et le `x` de la fonction `main` n'ont rien à voir : ce sont deux variables locales de fonctions distinctes. On pourrait presque imaginer qu'elles s'appellent `x_modifier` et `x_main`.

En réalité, nous avons vu dans le TP1 une fonction capable de modifier ses arguments, à savoir ***scanf***. Cette fonction lit dans le terminal, et stocke ce qu'elle lit dans les variables que l'on passe en argument. Ce comportement est lié au fait que l'on écrive les arguments `&x`, `&y` ... au lieu de simplement `x`, `y`... Le symbole `&` permet d'obtenir l'adresse d'une variable, c'est à dire le numéro de sa case mémoire (Cf. chapitre 3). Dans cette situation, on parle de ***passage par adresse***, car on fournit à la fonction l'adresse de la variable, permettant ainsi une modification : la fonction peut aller directement écraser ce qu'il y a écrit à l'emplacement physique de la variable.

Exemple 8

En Python, les listes sont passées par adresse, et sont donc modifiables par les fonctions.

Remarque 2

Techniquement, l'utilisation de `&x` reste du passage par valeur, car on donne à la fonction `scanf` la valeur "adresse de `x`", qu'elle utilise pour aller modifier le contenu de `x` !

C Variables globales

On peut déclarer des variables qui sont partagées entre les différentes fonctions. On dit que ce sont des variables ***globales***. En C, il suffit de mettre la déclaration de variable ***hors*** du corps des fonctions, comme suit :

```c
#include <stdio.h>

int variable_globale;

void modifier(){
  variable_globale = 0;
}


int main(){
  variable_globale = 5;
  printf("%d\n", variable_globale);
  modifier();
  printf("%d\n", variable_globale);
  return 0;
}
```

Dans ce programme, la variable `variable_globale` est bien partagée entre les fonctions, et donc le programme affiche bien 5 puis 0.

Fonction VS fonction Le terme ***fonction*** fait évidemment penser aux fonctions mathématiques. Ce sont bien deux concepts distincts. Notamment, en mathématiques, une fonction renvoie toujours le même résultat lorsqu'on l'applique aux mêmes arguments. En informatique, ce n'est pas le cas (ex : l'utilisation d'aléatoire), et une fonction peut même modifier ses arguments, ce qui n'aurait aucun sens en mathématiques. De plus, une fonction mathématique décrit la valeur qu'elle calcule, mais pas toujours la manière de la calculer. En informatique, une fonction est une suite d'instructions, c'est à dire l'implémentation d'un ***algorithme***.

3 Boucles

Les boucles sont un des outils principaux en programmation. Ce sont elles qui permettent de faire des programmes complexes et intéressants. Le principe d'une boucle est de répéter plusieurs instructions, un certain nombre de fois, ou jusqu'à ce qu'une certaine condition soit satisfaite.

A Boucle tant-que

Comme son nom l'indique, une boucle ***tant-que*** va servir à exécuter des lignes de code tant qu'une certaine condition est vérifiée. Voyons sur un exemple simple comment utiliser une telle boucle

On rappelle que la factorielle de n, notée $n!$, est le produit de tous les entiers entre 1 et n.

$$n! = \prod_{i=1}^{n} i$$

Utilisons une boucle tant-que pour écrire une fonction calculant la factorielle d'un entier n. L'idée va être de créer une variable i qui va varier entre 1 et n, ainsi qu'une variable r (comme

résultat) que l'on va multiplier par i à chaque étape. Donc, la boucle s'arrêtera lorsque i dépasse n strictement. On copie en fait le comportement naturel d'une personne à qui on demanderait de calculer une factorielle [1].

Écrivons d'abord l'algorithme en pseudo-code :

Algorithme 1 : `factorielle(n)`

Entrée(s) : n entier positif
Sortie(s) : $n!$

```
1 r ← 1;
2 i ← 1;
3 tant que i ≤ n faire
4 |   r ← resultat × i;
5 |   i ← i + 1;
6 retourner r
```

ligne	1	2	3	4	5	3	4	5	3	4	5	3	6
i	?	1	1	1	2	2	2	3	3	3	4	4	4
r	1	1	1	1	1	1	2	2	2	6	6	6	6
$i \leq n$	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✗

FIGURE 1 – Tableau d'exécution de `factorielle(3)`

Le bloc d'instruction commençant par **Tant que** $i \leq n$ **faire** s'appelle une boucle ***tant que***, ou boucle ***while***. La partie $i \leq n$ s'appelle la **condition de boucle**. Pour exécuter une boucle while, on regarde si la condition est vérifiée, et si c'est le cas on exécute les instructions internes de la boucle. On revérifie alors la condition, et ainsi de suite. La boucle s'arrête lorsque la condition n'est plus vérifiée pour la première fois.

Simulons une exécution de ce code à la main. On représente l'exécution d'un algorithme ou d'un programme par un **tableau d'exécution**. Les colonnes correspondent à l'avancement du programme, c'est à dire à la dernière ligne de code exécutée, et on a une ligne du tableau par variable. Dans chaque case, on indique la valeur de la variable de la ligne, à l'instant correspondant à la colonne (voir Fig. 1). Il peut aussi être utile de rajouter une ligne dans laquelle on note si la condition de boucle est vérifiée ou non. De manière générale, on peut rajouter toutes les informations dont on peut avoir besoin.

Maintenant, implémentons cet algorithme en C :

```c
int factorielle(int n){
  int r = 1;
  int i = 1;
  while (i <= n){
    r = r * i;
    i = i+1;
  }
  return r;
}
```

La syntaxe d'une boucle tant-que en C est donc assez simple :

```c
while (CONDITION){
  /* CODE */
}
```

1. On parle d'une personne qui aime les maths. Le comportement naturel d'une personne à qui on demanderait de calculer une factorielle est la fuite instantanée.

Exemple 9

Pour $n \in \mathbb{N}$, on appelle **racine entière** de n l'entier $k \in \mathbb{N}$ le plus grand possible tel que $k^2 <= n$ (autrement dit $k = \lfloor \sqrt{n} \rfloor$. Tentons d'écrire un algorithme calculant la racine entière d'un nombre en utilisant une boucle while. L'idée est de tester tous les entiers $k \in \mathbb{N}$ jusqu'à ce que k^2 dépasse strictement n. On sait alors que l'on peut renvoyer $k-1$ car on a $(k-1)^2 \leq n < k^2$.

Algorithme 2 : Racine entière

Entrée(s) : n entier positif
Sortie(s) : k la racine entière de n

$k \leftarrow 0$;
tant que $k^2 \leq n$ **faire**
  $k = k + 1$;
retourner $k - 1$

Exécutons cet algorithme sur $n = 21$:

Implémentons maintenant cet algorithme en C :

```c
/* renvoie le plus grand entier k tel que k^2 <= n
   lorsque n est un entier positif */
int racine_entiere(int n){
  int k = 0;
  while (k*k <= n){
    k = k+1;
  }
  return k-1;
}
```

A partir du moment ou un langage de programmation peut faire des boucles while, il est aussi puissant que Python, C, OCaml, etc... Cette puissance d'expressivité vient à un prix : une boucle while peut ne jamais terminer.

Exemple 10

Le programme suivant ne termine jamais :

```c
int main(){
  while (1 == 1){
    ; // instruction vide
  }
  return 0;
}
```

Notons que les logiciels que l'on utilise dans la vie de tous les jours tournent en boucle à l'infini tant qu'on ne les ferme pas (navigateur, jeu vidéo, traitement de texte...), et ont donc la structure suivante :

```
tant que la fenêtre est ouverte faire
    Récupérer la saisie de l'utilisateur;
    Mettre à jour l'état du programme et l'afficher à l'écran;
```

Exemple 11

Écrivons un programme qui permet de calculer la racine entière de tous les nombres que rentre l'utilisateur. On décide arbitrairement que lorsque l'utilisateur rentre un nombre négatif, le programme se termine.

```c
int main(){
  int x = 0;
  printf("Entrez un entier positif (-1 pour quitter).\n");
  while (x >= 0){
    scanf("%d", &x);
    printf("%d\n", racine_entiere(x));
  }
  return 0;
}
```

B Boucle for

Dans de nombreux cas, on utilise les boucles pour inspecter et traiter un nombre fini mais connu à l'avance d'objets. Par exemple, écrivons un algorithme qui permet de trouver le maximum de n entiers (pour $n \in \mathbb{N}^*$), en s'inspirant de l'algorithme vu au tout début du chapitre : on utilise une variable r qui stocke le maximum des valeurs déjà vues :

Algorithme 3 : Recherche de maximum

Entrée(s) : $n \in \mathbb{N}^*, (x_0, \dots, x_{n-1}) \in \mathbb{N}^n$
Sortie(s) : Maximum de $x_0, \dots, x_{n-1}$

```
r ← x_0 // maximum vu pour l'instant
i ← 1 // premier indice pas encore testé
tant que i < n faire
    si x_i > r alors
        r ← x_i;
    i ← i + 1;
retourner r
```

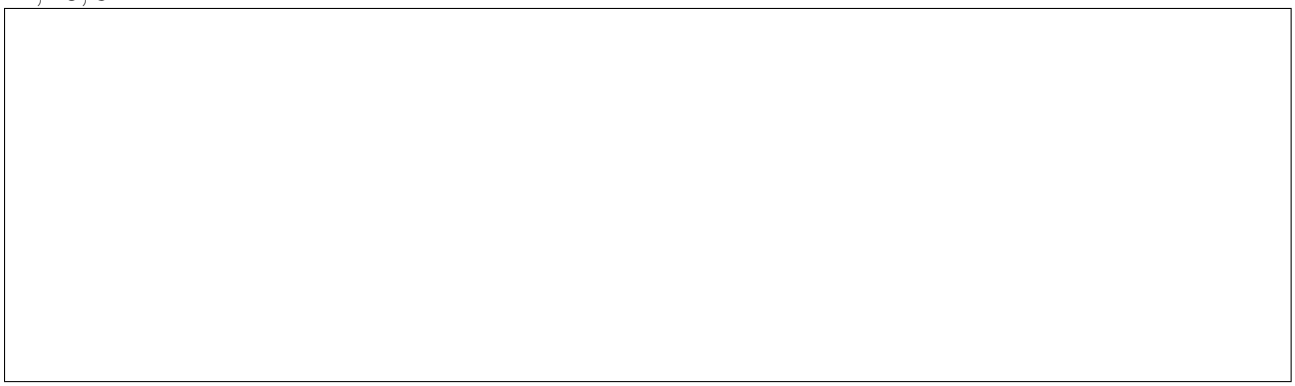
Un autre exemple : on veut calculer la somme de n réels. On utilise ici une variable s stockant la somme partielle des éléments déjà vus.

Algorithme 4 : Somme

Entrée(s) : $n \in \mathbb{N}^*, (x_0, \dots, x_{n-1}) \in \mathbb{N}^n$
Sortie(s) : $x_0 + \cdots + x_{n-1}$

```
1 r ← 0// somme des éléments vus pour l'instant
2 i ← 0// premier indice pas encore vu
3 tant que i < n faire
4 |   r ← r + x_i;
5 |   i ← i + 1;
6 retourner r
```

Exécutons ce dernier algorithme sur un exemple, en prenant $n = 4$ et $x_0, x_1, x_2, x_3 = 12, 21, 13, 8$:



On remarque que la boucle a une structure très simple : on parcourt chaque x_i en incrémentant i de 1 à chaque fois. Ce modèle de boucle revient très souvent en informatique, si bien que l'on utilise une autre syntaxe pour les réaliser : les **boucles pour**. Par exemple, le calcul de la somme peut se réécrire :

```
1 pour i = 0 à n − 1 faire
2 |   r ← r + x_i;
3 retourner r
```

La syntaxe générale des boucles pour est :

```
1 pour i = debut à fin par increment faire
2 |   Corps de la boucle;
```

Pour exécuter cette boucle, on initialise i à la valeur **debut**, puis on exécute une première fois le corps de la boucle. On augmente ensuite i de **increment**, et on exécute à nouveau le corps de la boucle. On continue ainsi jusqu'à ce que i ait dépassé la valeur **fin**. Tout se passe comme si on avait écrit :

```
1 i = debut;
2 tant que i ≤ fin faire
3 |   Corps de la boucle;
4 |   i ← i + increment;
```

Exercice 2

Réécrire l'algorithme de calcul du maximum ci-dessus en utilisant une boucle pour.

Exercice 3

On veut compter le nombre de nombres premiers entre deux entiers $a \leq b \in \mathbb{N}$:

Algorithme 5 : Compte premier

Entrée(s) : $a, b \in \mathbb{N}$ avec $a \leq b$
Sortie(s) : $|\{p \in [\![a, b]\!], p \text{ premier }\}|$

```
1  p ← a // varie entre a et b
2  n ← 0 // compte le nombre de nombres premiers vus
3  tant que p ≤ b faire
       // Tester si p est premier en testant tous les diviseurs
          potentiels entre 2 et p − 1
4      est_premier ← vrai;
5      d ← 2// potentiel diviseur
6      tant que d < p faire
7          si d divise p alors
8              est_premier ← faux;
9          d = d + 1;
10     si est_premier alors
11         n ← n + 1;
12     p ← p + 1;
13 retourner n
```

Réécrire cet algorithme à l'aide de boucles pour

Un point **essentiel** est que les valeurs **debut**, **fin** et **increment** sont fixes, et ne changent pas au cours de la boucle.

Une boucle ***for*** est donc un cas particulier de boucle ***while***. En fait, les boucles for sont strictement moins puissantes que les boucles while : un langage de programmation qui n'aurait que des boucles for ne permettrait pas de faire la même chose que le C, le python, etc...

Exemple 12

Essayez de remplacer la boucle while par une boucle ***for*** dans l'algorithme suivant :

Algorithme 6 : N-ème nombre premier

Entrée(s) : $n \in \mathbb{N}^*$
Sortie(s) : n-ème nombre premier

```
1  p ← 0;
2  c ← 0// compte le nombre de nombres premiers vus
3  tant que c ≤ 0 faire
4      si p est premier alors
5          c ← c + 1;
6          si c = n alors
7              retourner p
8      p ← p + 1;
```

Intuitivement, on utilise une boucle while lorsque l'on ne sait pas combien de tours de boucle on devra effectuer.

C Boucles for en C

En C, le mot clé `for` permet de faire des boucles. Utilisons-le pour écrire une fonction C qui prend en entrée un entier $n \in \mathbb{N}^*$ et affiche les entiers de 1 à n :

```c
void compteur(int n){
  for (int i = 1; i <= n; i = i+1){
    printf("%d\n", i);
  }
}
```

La syntaxe d'une boucle ***for*** en C est la suivante :

```c
for(initialisation; condition; increment){
  /*
   * CORPS DE LA BOUCLE
   */
}
```

Une telle boucle comporte 4 éléments :

1. L'initialisation de la boucle : c'est le code exécuté la première fois que la boucle est lancée. Dans l'exemple précédent, l'initialisation est `int i = 1`.
2. La condition de boucle : cette condition est évaluée à chaque début de boucle. Si elle est vérifiée, on continue, sinon on sort immédiatement de la boucle. Dans l'exemple, c'est `i <= n`.
3. L'incrément : c'est une instruction qui est exécutée à chaque fin de boucle. C'est elle qui sert à faire progresser la boucle. Au dessus, l'incrément est `i = i + 1;`
4. Le corps de la boucle : C'est les instructions qui sont exécutées lors de chaque passage dans la boucle. Au dessus, le corps ne contient qu'une instruction : `printf("%dn", i);`. Le corps de la boucle est délimité par des accolades {, }.

Malgré leur nom, les boucles for du C ne sont pas équivalentes aux boucles for algorithmiques. En fait, elles sont aussi puissante que les boucles while. En effet, les deux codes suivants sont équivalents :

```c
while(condition){
  /*
   * CORPS DE LA BOUCLE
   */
}
```

```c
for( ; condition; ){
  /*
   * CORPS DE LA BOUCLE
   */
}
```

En pratique cependant, on utilisera presque toujours les boucles for en C en utilisant la syntaxe suivante afin de copier les boucles for algorithmiques :

```c
for(int INDICE = VAL_DEBUT; INDICE < VAL_FIN; INDICE = INDICE + INCREMENT){
  /*
   * CORPS DE LA BOUCLE
   */
}
```

Très souvent l'incrément vaut 1, on peut alors utiliser une syntaxe plus courte : en C, `i++` est équivalent à écrire `i=i+1`. On peut donc écrire :

```c
for (int i = 0; i < n; i++){
  printf("%d\n", i);
}
```

4 Fonctions récursives

Rien n'interdit une fonction de s'auto-appeler dans son propre code ! On dit d'une telle fonction qu'elle est récursive.

Exemple 13

Soit $x \in \mathbb{N}$. La suite de Syracuse de x est une suite $(u_n)_{n\in\mathbb{N}}$ à valeurs dans $\mathbb{N}$ définie par :

$$\begin{aligned} u_0 &= x \\ u_{n+1} &= \begin{cases} \dfrac{u_n}{2} & \text{si } u_n \text{ est pair,} \\ 3u_n + 1 & \text{si } u_n \text{ est impair.} \end{cases} \end{aligned}$$

Codons une fonction qui calcule le n-ième terme de la suite de Syracuse d'un entier x. L'idée est de remplacer dans la définition les u_n par des `syracuse(x, n)` :

```c
int syracuse(int x, int n){
  if (n == 0){
    return x;
  } else {
    if (syracuse(x,n-1)%2 == 0){
      return syracuse(x,n-1)/2;
    } else {
      return 3 * syracuse(x,n-1) + 1;
    }
}
```

Ce code comporte un très gros problème. En effet regardons ce qu'il se passe si l'on appelle `syracuse(x, 5)`. Le programme commence par calculer `syracuse(x, 4)` pour tester la condition du if. Puis, que la condition soit vérifiée ou non, il recalcule `syracuse(x, 4)` afin de renvoyer le résultat. Donc, chaque appel à la fonction cause deux appels récursifs !

Traçons un schéma montrant tous les appels récursifs qui sont créées lorsque l'on exécute `syracuse(x, 4)` :

On appelle ce type de schéma un ***arbre d'appel de fonction***.

On observe donc une explosion exponentielle du nombre d'appels : pour calculer `syracuse(x, n)`, il faudra de l'ordre de 2^n appels de fonction au total. On peut rectifier ce problème en sto-

ckant la valeur de `syracuse(x, n-1)` dans une **variable temporaire** :

```
1  int syracuse(int x, int n){
2    if (n == 0){
3      return x;
4    } else {
5      int precedent = syracuse(x, n-1);
6      if (precedent%2 == 0){
7        return precedent/2;
8      } else {
9        return 3 * precedent + 1;
10     }
11 }
```

Avec cette modification, On ne fait plus qu'un seul appel récursif, et donc le nombre total d'appels lors du calcul de `syracuse(x, n)` est de l'ordre de n, ce qui est beaucoup plus efficace :

Exercice 4

Tracer l'arbre d'appel de `syracuse(5, -1)`. Que peut-on dire de l'exécution ?

5 Étude de programmes

Lorsque l'on écrit un programme informatique, on doit s'assurer que ce programme se comporte "comme il faut". L'étude des algorithmes et des programmes sert à formaliser cette notion. Nous allons voir dans cette partie comment écrire des preuves, et raisonner de manière formelle, sur des algorithmes, et comment utiliser ce formalisme pour écrire des programmes plus sûrs. Comme exemple filé, nous allons étudier l'algorithme suivant, déterminer intuitivement ce qu'il fait, et faire une **preuve** que notre intuition est correcte :

Algorithme 7 : `mystere`(n, m)

Entrée(s) : $n, m \in \mathbb{N}$
Sortie(s) : ???

```
1 N ← n;
2 R ← 0;
3 tant que N > 0 faire
4 |   N ← N − 1;
5 |   R ← R + m;
6 retourner R
```

A Spécification

Définition 3

La **spécification** d'un algorithme, d'une fonction, d'un programme, est la description de sa sortie en fonction de ses entrées. Lorsque les entrées doivent vérifier certaines hypothèses pour que l'exécution se passe correctement, on appelle ces hypothèses des **pré-conditions**.

Par exemple, voici une spécification d'algorithme :

Algorithme 8 : Test de primalité

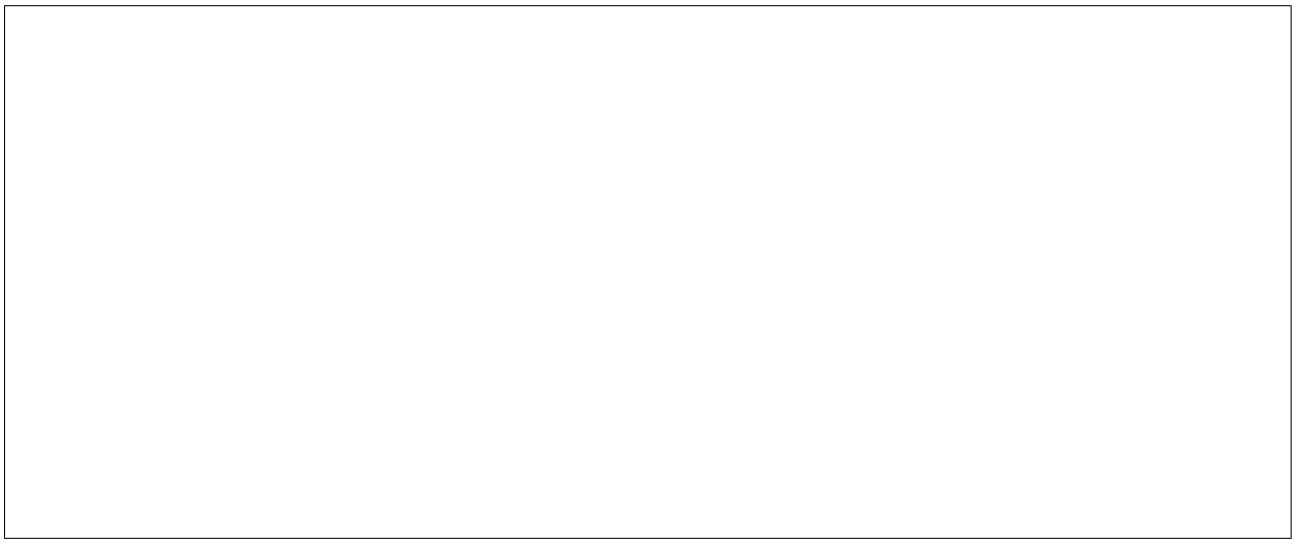
Entrée(s) : $n \in \mathbb{N}$
Sortie(s) : Oui si n est premier, Non sinon

Notons que l'on ne sait pas ce que renvoie l'algorithme si on lui fournit en entrée un entier négatif : il pourrait tourner à l'infini, renvoyer Oui, Non, lever un message d'erreur, etc... Le comportement d'un programme est uniquement défini là où les pré-conditions sont remplies.

Définition 4

On dit qu'un algorithme est **partiellement correct** s'il correspond bien à sa spécification, et qu'il est **totalement correct** si, de plus, il termine sur toutes les entrées valides (i.e. vérifiant les pré-conditions).

Faire **l'étude de correction** d'un programme, c'est montrer qu'il est correct. Revenons à l'algorithme `mystere`. Avant de pouvoir étudier sa correction, il faut déterminer sa spécification, c'est à dire ce qu'il fait. Pour cela, tentons de l'exécuter sur l'entrée $n = 3, m = 5$, et dressons un tableau d'exécution :



Il apparaît que cet algorithme semble réaliser une multiplication : il renvoie $n \times m$. De plus, on voit la manière dont le résultat est construit : la variable N sert à décompter, et la variable R accumule le résultat.

On peut donc imaginer que la spécification de l'algorithme `mystere` est :

Algorithme 9 : `mystere`(n, m)

Entrée(s) : $n, m \in \mathbb{N}$
Sortie(s) : $n \times m$

Notons qu'avec cette spécification, on ne peut multiplier que des entiers positifs. En réalité, l'algorithme fonctionne aussi si m est négatif, en revanche lorsque n est négatif, le résultat est systématiquement 0.

Invariant de boucle

Il nous reste maintenant à montrer que l'algorithme est correct. Pour cela, il faut montrer que pour $n, m \in \mathbb{N}$:

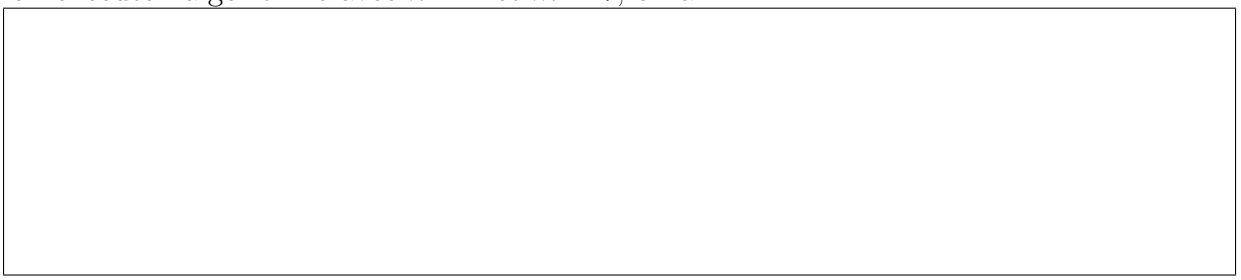
— l'exécution termine toujours ;

— le résultat est toujours $n \times m$

Pour cela, nous allons devoir apprendre à énoncer et prouver des propriétés sur les algorithmes. Pour le moment, on manipule principalement des algorithmes constitués de boucles. Une boucle peut avoir toutes sortes de propriétés intéressante, que l'on exprime en fonction des variables et du nombre de tours de boucle effectués. Autrement dit, on étudie les **suites** (au sens mathématique) des valeurs prises par les variables du programme. Par exemple, dans l'algorithme `mystere`, on peut étudier les suites $(n_k)_{k\in\mathbb{N}}, (m_k)_{k\in\mathbb{N}}, (N_k)_{k\in\mathbb{N}}, (R_k)_{k\in\mathbb{N}}$ avec n_k la valeur prise par n après k tours de boucles, m_k la valeur prise par m après k tours de boucle, et ainsi de suite.

Exemple 14

Si l'on exécute l'algorithme avec $n = 4$ et $m = 7$, on a :



On fait bien la différence entre une variable et la suite des valeurs qu'elle contient. Une variable au sens informatique est un objet totalement différent d'une variable mathématique : en informatique, une variable est une boîte avec un nom, dont le contenu change au cours du temps.

Notons que ces suites ne sont pas vraiment infinies, car les boucles s'arrêtent généralement après un nombre fini de tours, mais on peut étudier les termes qui sont bien définis, et qui correspondent aux vrais tours de boucles.

Étudions ces suites. Il est assez clair que $(n_k)_{k\in\mathbb{N}}$ est une suite constante égale à l'entrée n_0, puisque n n'est jamais modifiée dans le programme.
Nous avons remarqué expérimentalement que N sert de décompte. On peut donc émettre l'hypothèse que, pour $k \in \mathbb{N}$ un nombre valide de tours de boucles, on a $N_k = n - k$.

On peut même le montrer assez simplement en raisonnant par récurrence !

Pour $k \in \mathbb{N}$, notons $P(k)$ la propriété : "si la boucle tant-que effectue au moins k passages, alors $N_k = n - k$". Montrons par récurrence que $\forall k \in \mathbb{N}, P(k)$:

— Initialisation : Pour $k = 0$: la boucle exécute forcément au moins 0 passages, et N_0 est la valeur prise par N initialement, avant même de rentrer dans la boucle. Autrement dit, $N_0 = n = n - 0$: $P(0)$ est vraie.

— Hérédité : supposons $P(k)$ vraie, et supposons que la boucle effectue un $k+1$-ème passage. On peut exprimer N_{k+1} en fonction de N_k en suivant les instructions exécutées par la boucle, et il apparaît que $N_{k+1} = N_k - 1$. Or, par HR, $N_k = n - k$. Donc $N_{k+1} = N - k - 1 = N - (k + 1)$, d'où $P(k + 1)$.

Par récurrence, nous avons bien montré que pour $k \in \mathbb{N}$, si N_k est défini, alors $N_k = n - k$.

Définition 5

Une propriété exprimée à l'aide des variables d'un programme et étant vraie tout au long de l'exécution d'une boucle s'appelle un **invariant** de cette boucle.

La preuve précédente a permis de montrer l'invariant "$N_k = n - k$" (où k est le nombre de tours effectués).

Nous allons utiliser les invariants pour montrer la terminaison et la correction de l'algorithme `mystere`.

B Terminaison

Considérons les deux algorithmes suivants :

Algorithme 10 : Boucle infinie

Entrée(s) : x un entier
Sortie(s) : Rien

```
1 tant que 0 = 0 faire
2     Ne rien faire;
```

Algorithme 11 : Boucle finie... ?

Entrée(s) : a et b deux entiers
Sortie(s) : La différence entre b et a

```
1 diff ← 0;
2 tant que b ≠ a faire
3     b ← b − 1;
4     diff ← diff + 1;
5 retourner diff;
```

Le premier algorithme ne termine pas. En effet, la condition $0 = 0$ est toujours réalisée, l'algorithme ne sort donc jamais de la boucle.

Le deuxième algorithme semble terminer. En effet, si l'on simule son exécution pour $b = 10$, $a = 5$, on a :

Il semble donc que l'algorithme termine, et que l'on en a même une **preuve** : la quantité $b - a$ décroit de 1 en 1, et atteint donc forcément 0, faisant sortir de la boucle dont la condition est $b \neq a$. Cependant, si l'on prend comme entrées $b = 8$, $a = 15$, on voit que la quantité $b - a$ peut être négative initialement, et donc décroître sans jamais passer par 0.

Pour faire en sorte que le deuxième algorithme termine, on pourrait échanger a et b s'ils ne sont pas dans le bon ordre. On peut aussi rajouter une pré-condition :

Algorithme 12 : Boucle finie... ?

Entrée(s) : a et b deux entiers avec $a \leq b$
Sortie(s) : La différence entre b et a

Lorsque la condition $b > a$ est vérifiée au départ de l'algorithme, on peut garantir que :

1. $b - a$ est positif et entier au début de l'algorithme
2. $b - a$ décroît de 1 à chaque tour de boucle

3. $b-a$ ne peut pas devenir strictement négatif, car dès que cette quantité atteint 0, on sort de la boucle.

Ces trois conditions garantissent que la boucle termine. Cette méthode peut être généralisée :

Définition 6

Un ***variant*** de boucle est une quantité exprimée à partir des variables d'un algorithme, qui est :

— entière

— positive durant toute l'exécution de l'algorithme

— **strictement** décroissante à chaque passage dans la boucle.

Ne pas confondre invariant de boucle et variant de boucle !

— un **invariant** est une **propriété** qui est toujours vraie, donc invariante au fil de l'exécution de la boucle.

— un **variant** est une **quantité** qui devient de plus en plus petite, donc variante avec l'exécution de la boucle.

Théorème 1

Si l'on peut exhiber un variant pour une boucle, alors cette boucle termine toujours.

Démonstration. Par l'absurde : si une boucle admettant un variant de boucle ne termine pas, alors en considérant les valeurs du variant de boucle à chaque passage, on peut construire une suite d'entiers positifs strictement décroissante, ce qui est absurde (principe de descente infinie de Fermat). □

Ainsi, la principale méthode pour prouver la terminaison d'un programme va être de chercher un variant pour chacune de ses boucles.

Montrons la terminaison de l'algorithme `mystere`. On cherche une quantité qui est **entière**, **positive**, **strictement décroissante**. Ici, la variable N semble être un bon candidat. Montrons tout de même formellement que c'est bien un variant de la boucle tant-que :

— Entière et positive : Montrons l'invariant "$N \in \mathbb{N}$". Pour $k \in \mathbb{N}$, on pose N_k la valeur de N après k tours de boucles. Montrons par récurrence que pour tout $k \in \mathbb{N}$, lorsque N_k est bien définie, $N_k \in \mathbb{N}$:

  — Initialisation : $N_0 = n \in \mathbb{N}$

  — Hérédité : Soit $k \in \mathbb{N}$, tel que $N_k \in \mathbb{N}$. Supposons que l'on effectue un $k+1$-ème passage. Alors, cela signifie que la condition de boucle est vérifiée, donc que $N_k > 0$. Ainsi, $N_{k+1} = N_k - 1$ est bien positif, et toujours entier (car N_k l'est par HR).

  On a bien montré que N reste entière et positive au long de l'exécution.

— Strictement décroissante : nous avons déjà montré que, pour $k \in \mathbb{N}$, si l'algorithme a effectué au moins $k+1$ tours, alors $N_{k+1} = N_k - 1$: N est bien strictement décroissante au cours de l'exécution !

Ainsi, N est un variant de l'unique boucle de l'algorithme `mystere`, ce qui garantit la terminaison.

C Preuve de correction

L'objectif de la correction est de montrer que le comportement d'un algorithme correspond bien à la spécification annoncée. Pour l'algorithme mystère, il faut montrer que `mystere`$(n, m) = nm$ pour tout $n, m \in \mathbb{N}$. Notons que l'algorithme termine par une instruction "**Return** R". Il faut donc réussir à montrer qu'en sortie de la boucle, on a :

$$R = nm$$

Généralement, pour montrer la correction d'un algorithme, on cherche à exprimer et montrer un invariant qui généralise le résultat final que l'on cherche. Ici, on va chercher à généraliser la formule ci-dessus. On pourrait par exemple se demander ce que vaut R, de manière générale, au cours de la boucle. Nous avons vu expérimentalement que R augmente de m en m, on pourrait donc proposer l'hypothèse suivante : $R_k = km$. Cette formule colle bien avec les valeurs obtenues en simulant l'exécution à la main, il reste à la montrer.

Montrons par récurrence que pour $k \in \mathbb{N}$, si l'algorithme effectue k tours de boucles, alors $R_k = km$:

— Initialisation : Pour $k = 0$, en entrée de boucle, on a bien $R_0 = 0 = 0 \times m$

— Hérédité : Soit $k \in \mathbb{N}$ tel que $R_k = km$. Supposons que l'on effectue un $k+1$-ème passage de boucle : on a $R_{k+1} = R_k + m = km + m$ (par HR), donc $R_{k+1} = (k+1)m$.

D'où le résultat annoncé.

Il reste alors une dernière étape : étudier la **sortie** de boucle. On sait que l'on sort de la boucle quand $N_k \leq 0$, mais on sait aussi (on l'a montré) que N est toujours positive. Autrement dit, lorsque l'on sort de la boucle, on a $N_k = 0$, soit $n - k = 0$, soit $k = n$. Ainsi, en sortie de boucle, $R_k = km = nm$, ce qui montre bien la correction partielle. Comme on a montré la terminaison précédemment, on peut dire que l'algorithme est totalement correct !

D Méthode générale

Lorsque l'on veut montrer la correction d'un algorithme, il faut :

1. Identifier la spécification précise de l'algorithme, et l'exprimer comme une formule mathématique utilisant les différentes entrées et variables de l'algorithme.
2. Déterminer des invariants pour les différentes boucles. Cette étape s'effectue en cherchant des propriétés simples sur les variables, ainsi qu'en cherchant à généraliser la formule obtenue à l'étape 1. Il peut être utile d'exécuter l'algorithme à la main sur des exemples pour voir apparaître des motifs, des formules plus ou moins évidentes.
3. Faire les preuves d'invariants. Pour chaque propriété :
   (a) Énoncer la preuve par récurrence proprement, en posant bien les suites des variables.
   (b) Montrer l'initialisation. Pour cela, on se place à l'entrée de la boucle, c'est à dire au tout premier instant où l'on teste la condition.
   (c) Montrer que la propriété se transmet d'un passage à l'autre. Pour cela, on exprime les valeurs x_{k+1} en fonction des x_k, et on vérifie que la propriété reste vraie. Comme en cours de maths, lorsque l'on utilise l'hypothèse de récurrence, on pense bien à le dire !
4. Se placer à la fin de la boucle, en niant la condition, et à l'aide des invariants, conclure sur la propriété de l'étape 1.

Les étapes 1, 3, et 4 sont **faciles**, on peut presque les faire automatiquement sans trop réfléchir. Par exemple, pour montrer qu'une propriété est un invariant (étape 3), il suffit de faire une preuve par récurrence où l'on vérifie qu'on retombe bien sur nos pieds après un passage de boucle.
Seule l'étape 2, qui consiste à trouver des invariants adéquats, est difficile, car il faut parfois avoir l'intuition de quels invariants vont être nécessaires, et de comment les exprimer.

UN INVARIANT SEUL NE SUFFIT PAS A MONTRER LA CORRECTION

Attention : la dernière étape n'est PAS facultative ! La phrase "L'algorithme a un invariant de boucle donc il est correct" n'a aucun sens ! !

Exemple 15

Voici une preuve (**fausse**) de l'hypothèse de Riemann :

Démonstration. On considère l'algorithme suivant :

Algorithme 13 : `Resoudre_Riemann`

Entrée(s) :
Sortie(s) : Oui si l'hypothèse de Riemann est vraie, Non sinon

```
// Invariant de boucle: 1 + 1 = 2
1 tant que 1 = 0 faire
2     Se tourner les pouces;
3 retourner Oui
```

L'algorithme termine car on ne rentre pas dans la boucle puisque $1 = 0$. De plus, il est clair que $1 + 1 = 2$ est un invariant de boucle. L'algorithme a un invariant de boucle donc il est correct. Or l'algorithme renvoie Oui : d'après sa spécification l'hypothèse de Riemann est donc vraie. □

Cette preuve ne fonctionne pas car même si $1+1 = 2$ est un invariant de boucle, il n'y a pas de lien avec la spécification de l'algorithme. Il faut **faire le lien entre l'invariant et la spécification**, et pour ça, il faut préciser **ce que dit l'invariant en sortie de boucle**. En sortie de boucle, la négation de la condition de boucle devient vraie, ce qui ajoute de l'information à notre invariant.

E Étude de cas : exponentiation

Étudions quelques algorithmes et tentons d'exhiber, pour chacun, un variant de boucle permettant de montrer la terminaison. On considère le problème suivant : étant donné $x \in \mathbb{R}$ et $n \in N$, comment calculer la valeur de x^n ?

Exponentiation naïve

Exercice 5

L'algorithme naïf consiste à multiplier 1 par x, n fois d'affilée :

Algorithme 14 : `expo_naive`(x, n)

Entrée(s) : $x \in \mathbb{R}$, $n \in \mathbb{N}$
Sortie(s) : x^n

```
1 result ← 1;
2 i ← 0;
3 tant que i < n faire
4 |   result ← result × x ;
5 |   i ← i + 1 ;
6 retourner result
```

Commençons par la terminaison :

Q1. Montrer que $n - i$ est un variant de boucle.

Q2. Est-il vrai que $n - i$ est strictement positif tout au long de la boucle ?

Passons à la correction :

Q3. Exprimer la correction de l'algorithme `expo_naive` en complétant la phrase suivante : "En sortie de la boucle tant-que, result = ...".

Q4. Proposer une formule liant directement result, x et i.

Q5. Montrer la formule proposée.

Q6. En se plaçant à la sortie de la boucle, conclure.

Exercice 6

Il existe une méthode plus efficace pour calculer des puissances : ***l'exponentiation rapide***. On remarque :

$$x^n = \begin{cases} 1 & \text{si } n = 0 \\ x & \text{si } n = 1 \\ (x^2)^{\frac{n}{2}} & \text{si } n > 0 \text{ est pair,} \\ x \times (x^2)^{\frac{n-1}{2}} & \text{si } n > 0 \text{ est impair.} \end{cases}$$

Par exemple pour calculer 2^{11}, on voit que :

$$\begin{aligned} 2^{11} &= 2^5 \times 2^5 \times 2 \\ 2^5 &= 2^2 \times 2^2 \times 2 \\ 2^2 &= 2 \times 2 \end{aligned}$$

Autrement dit, il suffit de faire 5 multiplications, soit deux fois moins qu'avec la méthode naïve ! Le principe de l'algorithme d'exponentiation rapide est de calculer des puissances de x de plus en plus grandes (en suivant des puissances de 2 : $x^1, x^2, x^4, x^8, \dots$ [a]) et de les utiliser pour calculer x^n.

Algorithme 15 : `expo_rapide`(x, n)

Entrée(s) : $x \in \mathbb{R}$, $n \in \mathbb{N}$
Sortie(s) : x^n

```
1 result ← 1;
2 accu ← x;
3 N ← n;
4 tant que N > 0 faire
5     si N est impair alors
6         result ← accu × result ;
7     N ← N//2 // division entière: ⌊N/2⌋
8     accu ← accu × accu ;
9 retourner result
```

a. Cette méthode revient en fait à écrire n en base 2 !

Q1. Faire tourner l'algorithme à la main pour $x = 2$, $n = 11$, vérifier que l'on trouve bien 2048.

Q2. Montrer que pour tout $a \in \mathbb{N}^*$, $\lfloor \frac{a}{2} \rfloor < a$. On pourra faire une disjonction de cas selon la parité de a.

Q3. Vérifier que N est un variant de boucle.

Q4. Exprimer la correction de l'algorithme à l'aide d'une phrase du type "en sortie de la boucle, ..."

Pour $k \in \mathbb{N}$, on note r_k, a_k, N_k les valeurs respectives de result, accu et N après k tours de boucle.

Q5. Proposer une formule liant directement x^n, ${a_k}^{N_k}$ et r_k, et la montrer par récurrence.

Q6. Conclure.

Q7. Combien de tours de boucle peut-on effectuer dans le pire cas, en fonction de n ?

Théorème de l'arrêt L'étude de la terminaison et de la correction d'un programme se fait au cas par cas, et il n'existe pas de méthode systématique. Il y a même un résultat fondamental à ce sujet en informatique :

Théorème 2: Théorème de l'arrêt

Il est impossible d'écrire un algorithme qui :

— Prend en entrée le code source d'un programme

— Renvoie en sortie ***vrai*** si le programme termine, ***faux*** sinon.

Démonstration. En spé ! □

Ce théorème rend impossible la vérification systématique et automatisée de programme, et motive donc l'utilisation de méthodes comme les variants de boucle.

6 Test de programmes

Une fois que l'on a écrit un programme, vérifié sa terminaison et garanti sa correction sous certaines pré-conditions, il reste tout de même à vérifier qu'il n'y a pas de coquilles ou de bugs, et que le code correspond bien à l'algorithme théorique derrière. C'est à ça que servent les ***tests*** de programmes.

Tester un programme, c'est vérifier (en partie) son bon comportement. Un type de test standard est le ***test unitaire***.

Définition 7

Effectuer un ***test unitaire*** sur programme, c'est exécuter ce programme sur une entrée prédéfinie pour laquelle on connaît le résultat, et vérifier que la sortie correspond bien au résultat attendu.

Lorsque l'on écrit une fonction, on doit toujours fournir des tests unitaires. Un ensemble de tests unitaires s'appelle une ***batterie*** de tests.

Exemple 16

Voici une batterie de tests qui permettrait de tester une fonction de test de primalité :

```
1  // Renvoie true si n est premier, false sinon
2  bool est_premier(int n){
3    ...
4  }
5
6  void test_est_premier(){
7    assert(est_premier(2));
8    assert(est_premier(3));
9    assert(est_premier(31));
10   assert(!est_premier(4));
11   assert(!est_premier(1));
12   assert(!est_premier(0));
13   assert(!est_premier(-23));
14 }
```

Cette batterie de test est conçue pour tester plusieurs cas différents : des nombres premiers, et des nombres pas premiers pour différentes raisons (divisible par autre chose que 1 et eux-même, négatif, égal à 1...).

Une batterie de tests doit être construite de manière à tester le plus de cas possibles, afin de bien détecter les défauts éventuels. Il existe plusieurs méthodes permettant de construire de bonnes batteries de tests.

A Graphe de flot d'exécution

Nous avons déjà exécuté des algorithmes à la main, afin de comprendre leur comportement. Le flot d'exécution d'un programme est la manière dont les instructions s'enchaînent lors de l'exécution, en prenant en compte les différentes branches d'un if-else, les boucles for et while...

On peut représenter ce flot par un ***graphe de flot de contrôle***.

Définition 8

Le ***graphe de flot de contrôle*** (ou ***C***ontrol-***F***low ***G***raph) d'un programme est une représentation graphique de ce programme, constitué :

— De blocs d'instructions élémentaires

— De conditions

— De flèches reliant les blocs et les conditions dans l'ordre du programme

D'une condition, il y a toujours deux flèches qui partent : une étiquetée ***vrai*** et une étiquetée ***faux***. Les conditions servent à représenter d'une part les if-else, d'autre part les boucles.

Exemple 17

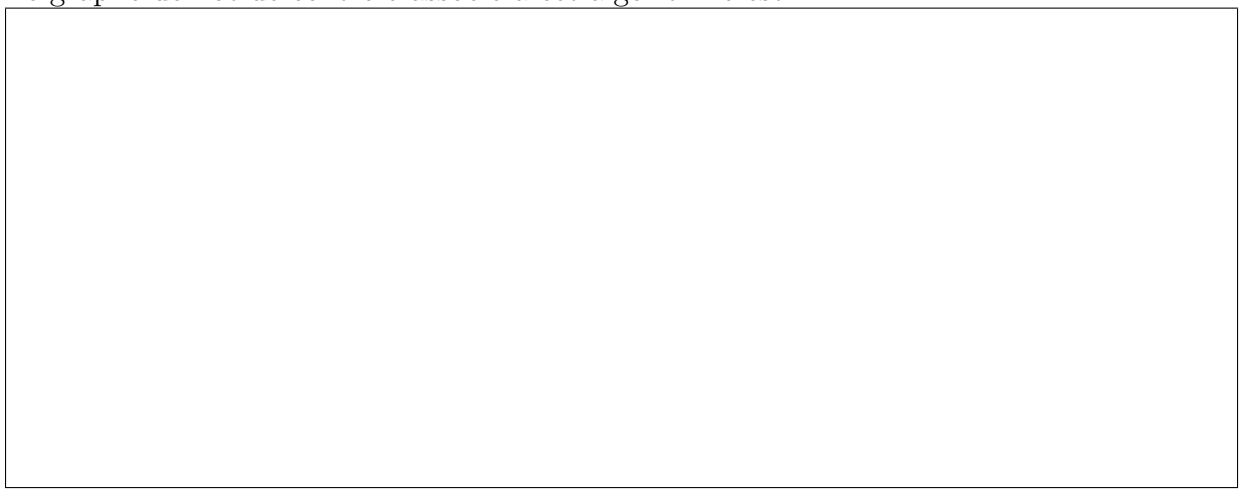
On reprend l'algorithme d'exponentiation rapide :

Algorithme 16 : Exp. rapide

Entrée(s) : $x \in \mathbb{R}$, $n \in \mathbb{N}$
Sortie(s) : x^n

```
1 result ← 1;
2 accu ← x;
3 tant que n > 0 faire
4 |   si n%2 = 1 alors
5 |   |   result ← accu × result ;
6 |   n ← n//2;
7 |   accu ← accu × accu ;
8 retourner result
```

Le graphe de flot de contrôle associé à cet algorithme est :



Dans ce graphe, on peut représenter une exécution par un chemin suivant certaines flèches. Par exemple, traçons sur le CFG précédent l'exécution pour $x = 2$ et $n = 7$.

Le CFG sert entre autres à déterminer la pertinence d'une batterie de tests. Idéalement, on voudrait qu'une batterie de tests essaie tous les chemins possibles dans le CFG. Cependant ce n'est jamais possible en pratique : dès qu'une boucle apparaît, le nombre de chemins devient infini, et même sans boucle il peut être bien trop grand pour tous les tester.

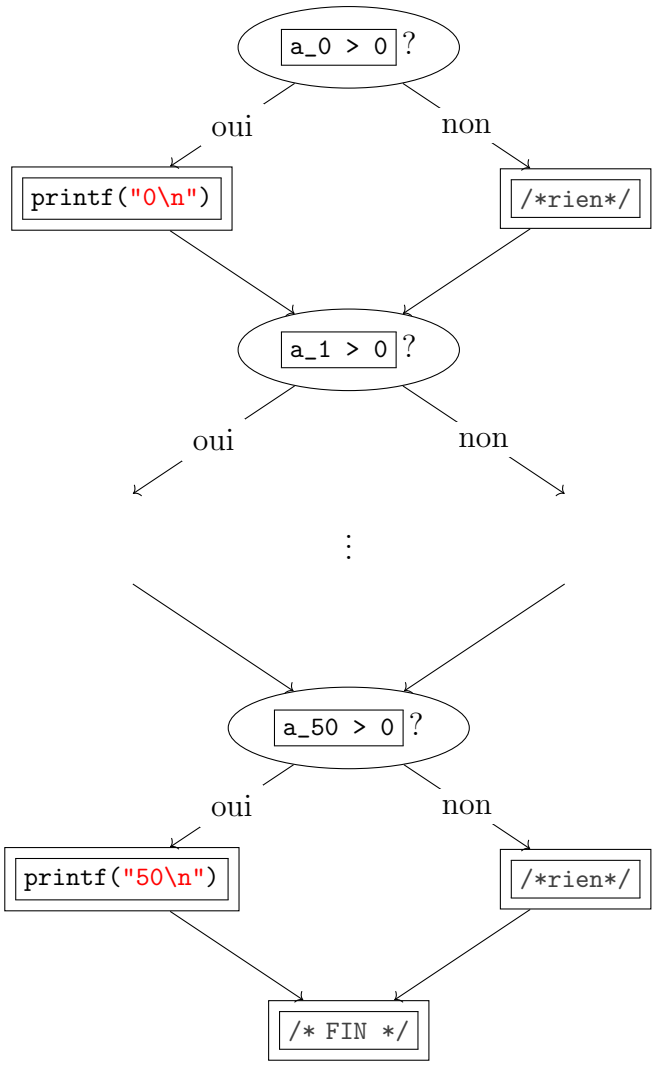
Exemple 18

Considérons le code suivant, et son CFG :

```c
#include <stdio.h>

int main(){
  int a_0;
  scanf("%d", &a_0);
  int a_1;
  scanf("%d", &a_1);
  /* ... */
  int a_50;
  scanf("%d", &a_50);


  if (a_0 > 0){
    printf("0\n");
  } else {
    // rien
  }
  /* ... */
  if (a_50 > 0){
    printf("50\n");
  } else {
    // rien
  }
}
```



Dans ce CFG, il y a 2^{51} chemins d'exécutions, bien trop pour les essayer tous un par un. En revanche, on voit que tester avec tous les a_i à 0 puis avec tous les a_i à 1 permet de tester toutes les branches.

Définition 9

- On dit qu'une batterie de tests réalise une ***couverture par sommets*** du graphe de flot de contrôle d'un programme s'il visite chaque bloc de code et chaque condition du CFG.
- On dit qu'une batterie de tests réalise une ***couverture par arêtes*** du graphe de flot de contrôle d'un programme s'il visite chaque flèche du CFG.

Par exemple, le test $(x = 2, n = 7)$ à lui tout seul couvre les sommets du CFG de l'algorithme d'exponentiation rapide, mais pas les arêtes. Si l'on rajoute par exemple $(x = 2, n = 2)$, on couvre bien les arêtes ET les sommets.

Remarque 3

Si une batterie de tests couvre par arêtes un CFG, alors elle le couvre aussi par sommets

Attention, avoir une batterie de tests qui couvre toutes les arêtes du CFG d'un programme ne signifie pas qu'elle détectera tous les bugs ! Mais elle aura plus de chance de les détecter qu'une batterie de tests qui ne couvre que partiellement le CFG.