Premiers programmes en C

Guillaume Rousseau
MP2I Lycée Pierre de Fermat
guillaume.rousseau@ens-lyon.fr

12 septembre 2024

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.

Exemple 1. On veut écrire 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){
  /* CODE 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`.

Exemple 2. 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. Par exemple :

```
// 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. Au dessus par exemple, `1` et `0` sont bien de type `int`.

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

```
1 // affiche le double de a
2 void afficher_double(float a){
3   float b = 2 * a;
4   printf("%d\n", b);
5 }
```

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 :

```
1  #include <stdio.h>
2
3  int ajoute_un(int x){
4          printf("%d\n", 1+x);
5          // le compilateur affiche un avertissement:
6          // la fonction est de type float mais n'a pas de return
7  }
8
9  int main(){
10         int y = ajoute_un(3);
11         printf("%d\n", y); // à l'exécution, affiche n'importe quoi
12 }
```

Commentaires Lorsqu'on écrit une fonction, on doit **SYSTEMATIQUEMENT** inclure un commentaire pour expliquer ce qu'elle 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...

Exemple 4. Une fonction et son commentaire :

```
1  /* renvoie x/y si y est non nul, sinon
2     affiche un message d'avertissement et renvoie 0 */
3  float safe_division(float x, float y){
4    if (y == 0){
5      printf("Attention: division par 0\n");
6      return 0;
7    } else {
8      return x/y;
9    }
10 }
```

B Appel de fonction

Lorsque l'on utilise une fonction que l'on a défini, on dit que l'on ***appelle*** la fonction. Par exemple, dans le code suivant :

```
1  #include <stdio.h>
2
3  // renvoie la moyenne de a et b
4  float moyenne(float a, float b){
5    return (a+b)/2;
6  }
7
8  int main(){
9    float x = 18.77, y = 15.36;
10
11   float c = moyenne(2*x, y+3);
12
13   return 0;
14 }
```

On dit que l'on ***appelle*** `moyenne` sur `x` et `y`. On dit également que `x` et `y` sont les ***arguments*** de la fonction. ***Attention*** à bien faire la différence entre les ***paramètres*** de la fonction, ici `a` et `b`, qui sont les noms utilisés dans la définition de la fonction, et les ***arguments***, ici `x` et `y`, qui sont les valeurs sur lesquelles la fonction est appelée.

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` :

```
1  #include <stdio.h>
2
3  float moyenne(float a, float b){
4    return (a+b)/2;
5  }
6
7  int main(){
8    float x = 18.77, y = 15.36,
9
10   float c = moyenne(2*x, y+3);
11
12   return 0;
13 }
```

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 5. 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 6. 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. Modifions le code précédent en renommant le paramètre de la fonction `modifier` :

```c
#include <stdio.h>

void modifier(int x){
  x = 0;
}

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

Rien ne change, car le `x` de la fonction `modifier` et le `x` de la fonction `main` n'ont rien à voir. En fait, on peut imaginer qu'elles s'appellent `x_modifier` et `x_main`.

En réalité, nous avons vu dans le TP1 une fonction qui modifie ses arguments : ***scanf***. En effet, `scanf` stocke dans ses arguments les valeurs qu'elle lit dans le terminal. 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 7. En Python, les listes sont passées par adresse, et sont donc modifiables par les fonctions.

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 while

Comme son nom l'indique, la boucle ***while*** 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 while 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.

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

Algorithme 1 : Factorielle

Entrée(s) : n entier positif
Sortie(s) : $n!$ la factorielle de n

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

FIGURE 1 – Calcul de la `factorielle`

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

FIGURE 2 – État mémoire lors de l'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. Les colonnes correspondront à l'avancement du programme, c'est à dire à la ligne de code exécutée courante. On a une ligne du tableau par variable, et on indique dans chaque case la valeur de la variable de la ligne, à l'instant correspondant à la colonne. Il peut aussi être utile de rajouter une ligne dans laquelle on note si la condition de boucle est vérifiée ou non (voir Fig. 2).

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 while en C est donc très simple :

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

Utilisons les boucles while sur quelques exemples.

Exemple 8. 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 le python, le C, le OCaml, etc... Cette puissance d'expressivité vient à un prix : une boucle while peut ne jamais terminer.

Exemple 9. Le programme suivant ne termine jamais :

```
1 int main(){
2   while (1 == 1){
3     ; // instruction vide
4   }
5   return 0;
6 }
```

Les boucles while sont l'outil principal de tous les programmes que l'on utilise dans la vie de tout les jours, car la plupart des programmes 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;
```

Par exemple, lorsque vous ouvrez Firefox, le programme lit votre clavier et votre souris en permanence afin de vous permettre de vous déplacer sur les pages, de cliquer sur des liens, de rentrer des URL, etc...

Exemple 10. 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.

```
1 int main(){
2   int x = 0;
3   printf("Entrez un entier pour avoir sa racine entière (-1 pour quitter).\n");
4   while (x >= 0){
5     scanf("%d", &x);
6     printf("%d\n", racine_entiere(x));
7   }
8   return 0;
9 }
```

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 $x_0, \dots, x_{n-1}$

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

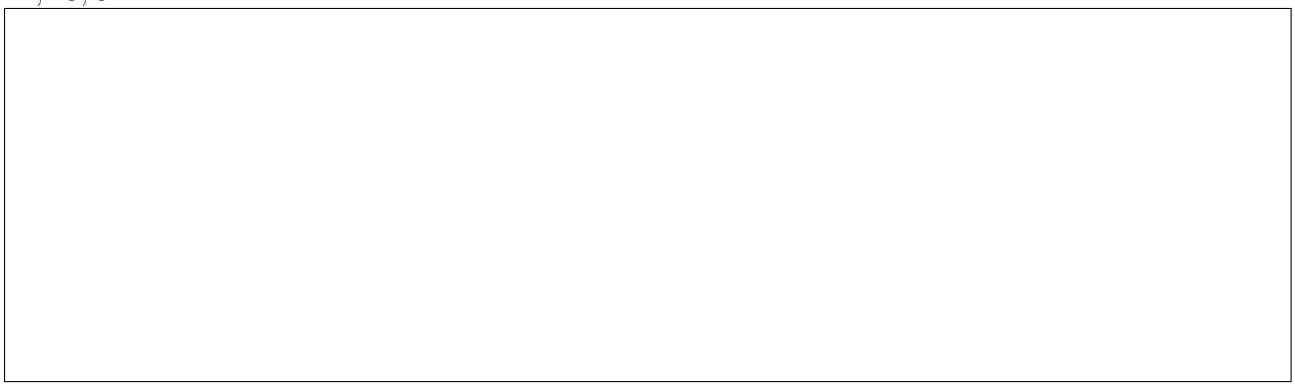
Un autre exemple : on veut calculer la somme de n réels. On utilise la même méthode : une variable s stockant la somme partielle des éléments déjà vus.

Algorithme 4 : Somme

Entrée(s) : $n \in \mathbb{N}^*, (x_0, \ldots, x_{n-1}) \in \mathbb{N}^n$
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 1. Écrire l'algorithme de calcul du maximum ci-dessus en utilisant une boucle pour.

Exercice 2. 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 11. 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);
}
```

Exercice 3. Écrire une fonction C qui teste si un nombre est premier en utilisant une boucle for, puis une fonction C qui prend en entrée un entier $n \in \mathbb{N}$ et affiche les n premiers nombres premiers, en utilisant une boucle while.

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 12. 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 :

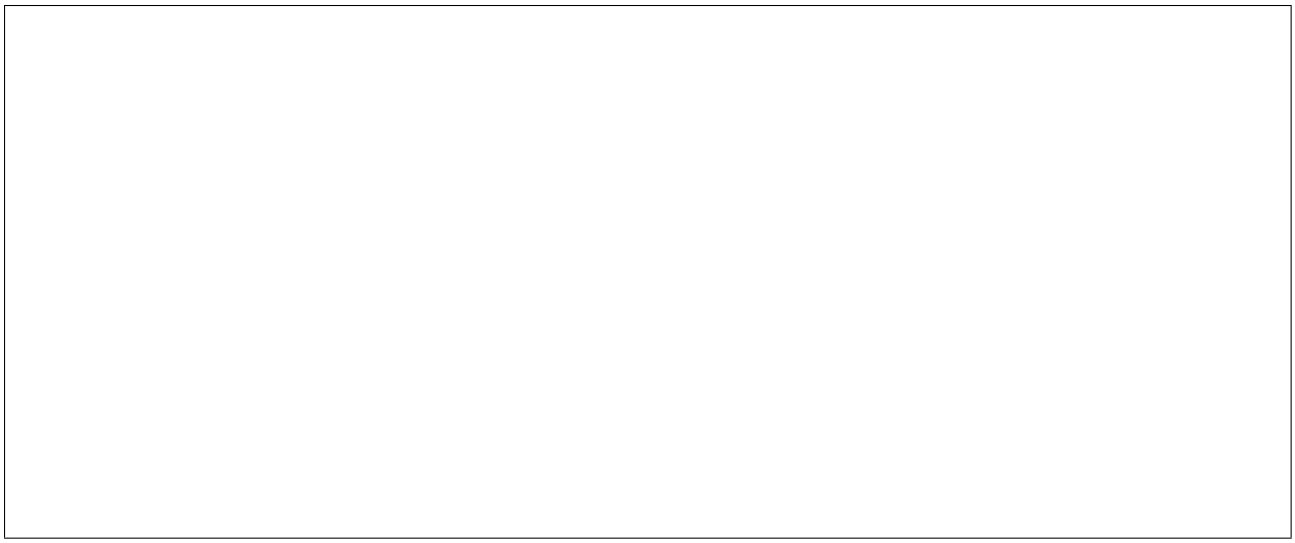
$$\begin{aligned} u_0 &= x \\ u_{n+1} &= \begin{cases} \dfrac{u_n}{2} & \text{si } u_n \text{ est pair,} \\ 3u_n + 1 & \text{si } u_n \text{ est impair.} \end{cases} \end{aligned}$$

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)` :

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

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)` :

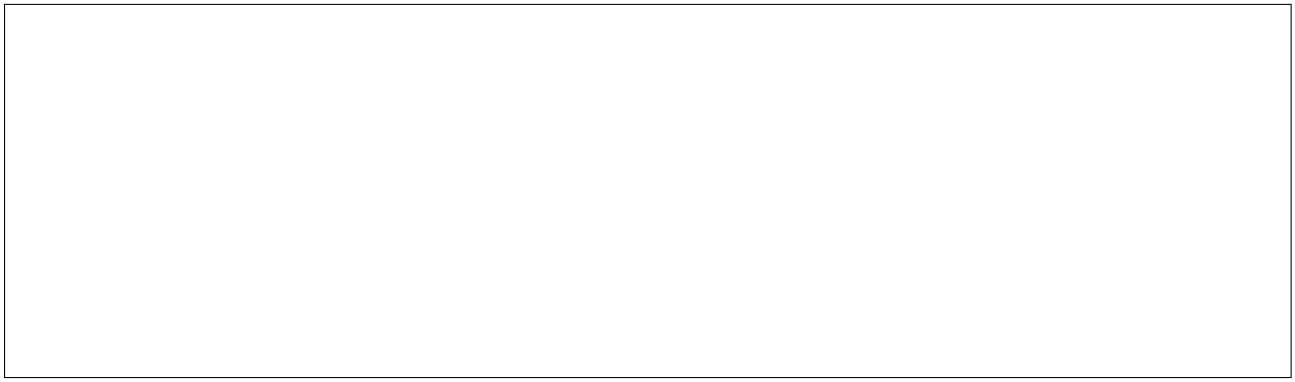


On appelle ce type de schéma un ***arbre d'appel de fonction***.

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 stockant 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 logique, sur des algorithmes, et comment utiliser ce formalisme pour écrire des programmes plus sûrs.

A Terminaison

Définition 3. On dit qu'un algorithme ***termine*** si pour toute entrée, il s'arrête au bout d'un nombre fini (mais dépendant de l'entrée) d'étapes.

Exemple 13. Considérons les deux algorithmes suivants :

Algorithme 7 : Boucle infinie

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

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

Algorithme 8 : 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, et donc l'algorithme ne peut pas sortir de la boucle.

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

Il semble donc que l'algorithme termine ***car*** la quantité $b - a$ décroit de 1 en 1 et que la condition de boucle est équivalente à $b - a = 0$. 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 des hypothèses à notre algorithme, par exemple demander que a soit inférieur à b :

Algorithme 9 : Boucle finie... ?

Entrée(s) : a et b deux entiers avec $a \leq b$
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;
```

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

— $b - a$ est positif au début de l'algorithme

— $b - a$ décroît de 1 à chaque tour de boucle

— $b - a$ ne peut pas devenir négatif, car dès que cette quantité vaut 1, on sort de la boucle.

Ces trois conditions garantissent que la boucle termine.

Cette méthode peut être généralisée :

Définition 4. Un ***variant*** de boucle est une quantité obtenue 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.

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 des **variants de boucles**.

Étudions quelques algorithmes et tentons d'exhiber, pour chacun, un variant de boucle permettant de montrer la terminaison.

Exponentiation rapide On considère le problème suivant : étant donné $x \in \mathbb{R}$ et $n \in N$, comment calculer la valeur de x^n ?

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

Algorithme 10 : Exponentiation naïve

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

```
1 result ← 1;
2 tant que n > 0 faire
3 |   result ← result × x ;
4 |   n ← n − 1 ;
5 retourner result
```

Ici, n est un variant de boucle. En effet, $n \in \mathbb{N}$ à tout moment, et n décroit strictement à chaque passage dans la boucle. On peut calculer le nombre exact de passages dans la boucle : il y en a n.

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

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

Par exemple pour calculer 2^{11} :

A partir de cette remarque, on en déduit l'algorithme d'exponentiation rapide :

Algorithme 11 : 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
```

Faisons tourner l'algorithme à la main pour $x = 2$ et $n = 11$:

Ici, n est bien un variant de boucle. En effet, lorsque $n > 0$, que n soit pair ou impair, on a bien $\lfloor \frac{n}{2} \rfloor < n$. Donc, comme il existe un variant de boucle, l'algorithme termine bien.

Exemple 14 (Descente en quinquonce)**.** On considère l'algorithme suivant :

Algorithme 12 : Quinquonce

Entrée(s) : $a \in \mathbb{N}$
Sortie(s) : Rien

```
1 b ← a + 1;
2 tant que a > 0 faire
3     si a == b alors
4         a ← a − 1;
5     sinon
6         b ← b − 1;
```

Cet algorithme fait décroitre a et b en décalé :

Ici, ni a ni b ne sont des variants de la boucle tant que. En effet, aucun des deux ne décroit strictement à chaque passage. En revanche, $a + b$ est bien un variant :

— $a + b$ est entier.

— $a + b$ est toujours positif ou nul.

— $a + b$ décroit strictement à chaque passage : On se place au début d'un passage et on montre le résultat par disjonction de cas. Notons a' et b' les valeurs de a et b à la fin du passage.
  — Si la condition du If est vérifiée, alors $a' = a - 1$ et $b' = b$;
  — Sinon, alors $a' = a$ et $b' = b - 1$.

  Dans tous les cas, $a' + b' = (a + b) - 1$.

La boucle tant que admet un variant de boucle, donc elle termine, et donc l'algorithme termine également.

Théorème de l'arrêt L'étude de la terminaison d'un programme se fait au cas par cas, en exhibant un bon variant de boucle. Il n'existe pas de méthode systématique permettant de montrer la terminaison d'un programme. En fait, il y a un résultat très fort 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.

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

B Préconditions, assertions

Nous avons vu au cours de ce chapitre plusieurs exemples de fonctions qui peuvent renvoyer des résultats incohérents ou même ne jamais finir si elles ne sont pas appelées sur les bons arguments. Par exemple, l'implémentation suivante de la fonction factorielle ne termine pas si on l'appelle sur un entier négatif :

```
1 // calcule la factorielle de n
2 int factorielle(int n){
3   if (n == 0){
4     return 1;
5   } else {
6     return n * factorielle(n-1);
7   }
8 }
```

Une solution est de modifier la condition et de renvoyer 1 si $n \leq 0$ et pas seulement si $n = 0$. Cependant, calculer la factorielle d'un entier négatif n'a pas de sens, il n'y a donc pas de raison de renvoyer 1 plutôt qu'autre chose. Dans ce cas, on peut spécifier dans la description de la fonction que son argument doit forcément être positif :

```
1 // calcule la factorielle de n un entier naturel
2 int factorielle(int n){
3   ...
4 }
```

Une personne qui lirait le code de cette fonction saurait alors qu'elle ne peut pas être utilisée a priori sur des entiers négatifs, car son comportement n'est pas spécifié. Le fait que n doit être positif est une ***précondition*** de la fonction.

Définition 5. Une précondition est une hypothèse qui doit être vérifiée au début d'une fonction pour assurer son bon fonctionnement.

Afin d'obliger une précondition à être satisfaite, on peut utiliser des ***assertions***. En C, la fonction `assert` permet de tester une condition, et d'arrêter le programme immédiatement si celle-ci n'est pas vérifiée. Par exemple :

```
1  // calcule la factorielle de n un entier naturel
2  int factorielle(int n){
3    assert(n >= 0);
4
5    if (n == 0){
6      return 1;
7    } else {
8      return n * factorielle(n-1);
9    }
10 }
```

Si l'on essaie maintenant d'appeler la fonction sur une entrée négative, on obtiendra une **erreur d'assertion**.

Cette utilisation des assertions permet de garantir que les préconditions sont satisfaites, et facilite donc le débogage.

C Correction

Définition 6. La ***spécification*** d'un algorithme ou d'un programme, c'est la description de sa sortie en fonction de ses entrées. Cette spécification prend en compte les préconditions.

Exemple 15. Regardons la spécification d'algorithme suivante :

Algorithme 13 : Test de primalité

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

Rien ne dit ce qui est sensé se passer lorsque l'entrée est négative : sans voir le code, on ne peut pas présupposer du comportement sur ces entrées. En revanche, sur les entrées positives, le comportement est clairement défini.

Définition 7. On dit qu'un algorithme est ***partiellement correct*** s'il correspond bien à sa spécification. On dit qu'un algorithme est ***totalement correct*** s'il est partiellement correct et s'il termine toujours.

Faire l'étude de la correction d'un algorithme ou d'un programme, c'est vérifier qu'il est bien correct, ou partiellement correct.

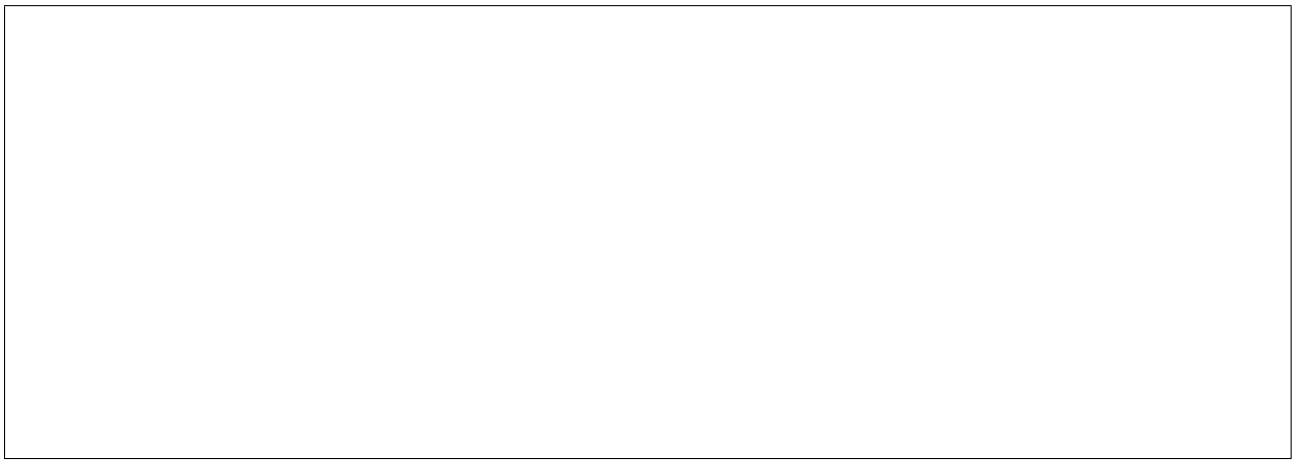
Exemple 16. On reprend le calcul de la factorielle :

Algorithme 14 : Factorielle

Entrée(s) : n entier positif
Sortie(s) : $n! = 1 \times 2 \times \dot{\times} n$

```
1 r ← 1;
2 pour i = 1 à n faire
3 |  r ← r × i;
4 retourner r
```

On veut montrer formellement que cet algorithme calcule bien $n! = 1 \times 2 \times \cdots \times n$.
Exécutons quelques étapes de l'algorithme à la main pour le calcul de 7!.



Pour $k \in [\![0, n]\!]$, et x une variable de notre programme, notons x_k la valeur de x après avoir effectué k passages de boucle. Par exemple, on a $i_0 = 1, i_1 = 2, \ldots$ et $r_0 = 1, r_1 = 1, r_2 = 2, r_3 = 6, \ldots$.

On peut donc émettre l'hypothèse que, pour tout $k \in [\![0, n]\!]$, on a :

$$i_k = k + 1 r_k = k!$$

On peut même le montrer ! Montrons par récurrence que l'on a $\forall k \in [\![0, n]\!], P(k)$: "$i_k = k+1$ et $r_k = k!$".

— Initialisation : Au bout de 0 tours, c'est à dire avant même de rentrer dans la boucle, on a bien $r_0 = 1$ et $i_0 = 1$, ce qui est précisément $P(0)$.

— Hérédité : On suppose $P(k)$ pour un entier $k \in [\![0, n-1]\!]$ quelconque. Montrons que l'on a aussi $P(k+1)$. Par HR, on a $i_k = k+1$ et $r_k = k!$ qui sont les deux valeurs de i et r après k passages. Étudions l'effet du $+1$-ème passage. La seule instruction du corps de la boucle modifie r en lui multipliant i, ce qui signifie que l'on a $r_{k+1} = r_k \times i_k$, et donc $r_{k+1} = k! \times (k+1) = (k+1)!$. De plus, après le corps de la boucle, i est incrémenté de 1 en prévision du prochain tour, et donc $i_{k+1} = i_k + 1 = k+2$. On a bien montré $P(k+1)$.

Ainsi, on en déduit qu'au bout du n-ème et dernier passage, on a $r_n = n!$. Ainsi, l'algorithme renvoie bien $n!$, il correspond donc à sa spécification et donc en sortie de boucle, on a bien $r = n!$: l'algorithme est correct car il répond à sa spécification !

Cette méthode, consistant à exhiber une propriété valide tout au long de la boucle, s'appelle la méthode des invariants de boucle.

Définition 8. Un ***invariant de boucle*** d'une boucle d'un programme est une propriété P formulée à partir des variables du programme, telle que :

— P est vraie à l'initialisation de la boucle

— Si P est vraie au début d'un passage de boucle, P est vraie à la fin de ce passage.

Correction des boucles for vs. while Dans la preuve précédente, on a un peu triché : on a dit qu'à la fin du corps de boucle, i est incrémenté en prévision du tour suivant. En réalité, pour une boucle pour, ce n'est pas clair ce qui arrive à i après le tout dernier passage, car la portée de l'indice de boucle est restreinte à la boucle. En pratique, il est toujours plus simple de travailler avec des boucles while. Lorsque l'on écrit un algorithme et que l'on veut en prouver la correction, on l'écrira en remplaçant les boucles pour par des boucles tant que !

L'utilité des invariants de boucle provient de la propriété suivant :

Proposition 1. Soit P un invariant d'une boucle ***tant que***. Alors P est vraie à chaque fois que l'on teste la condition de boucle. En particulier, si la boucle se termine, P est vraie **à la sortie de la boucle**.

Exemple 17. Appliquons cette méthode sur l'algorithme précédent de calcul de factorielle. On commence par transformer la boucle for en boucle while :

Algorithme 15 : Factorielle

Entrée(s) : n entier positif
Sortie(s) : $n! = 1 \times 2 \times \cdots \times n$

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

Il faut maintenant trouver un invariant de boucle qui aidera à montrer qu'en sortie de l'algorithme, on a bien $r = n!$. Le raisonnement par récurrence que l'on a fait plus haut prouve que la propriété $P : r = (i-1)!$ est un invariant de boucle. Réécrivons la preuve, dans le style d'une **preuve d'invariant**.

— En entrée de boucle, $r = 1 = 0! = (i-1)!$. P est vérifiée.

— Supposons P vraie au début d'un passage de boucle, montrons qu'elle est vraie à la fin du passage.
Notons r' et i' les valeurs de r et i à la fin du passage.
On a $r' = r \times i$ et $i' = i + 1$.
De plus, par hypothèse, $r = (i-1)!$.
Donc, $r' = (i-1)! \times i = i! = (i'-1)!$.
Autrement dit, P' est vérifiée à la fin du passage.

Pour conclure, il suffit de remarquer que lorsque l'on sort de la boucle, la condition de boucle devient fausse, on a donc $i > n$. De plus, i avance de 1 en 1, donc forcément $i = n + 1$. A cet instant, on a donc $r = (n+1-1)! = n!$. L'algorithme renvoie donc $n!$, et est donc correct : le résultat renvoyé correspond bien à la spécification.

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 un invariant de boucle qui permet de prouver cette formule. Cette étape s'effectue en généralisant la formule obtenue à l'étape 1, en faisant intervenir les variables qui apparaissent dans les boucles. Pour aider à trouver cet invariant on peut exécuter l'algorithme à la main sur des exemples et voir comment les différentes variables évoluent.
3. Montrer que la propriété énoncée est bien un invariant de boucle
   (a) Montrer que la propriété est vraie en entrée de boucle
   (b) Montrer que la propriété se transmet d'un passage à l'autre. Pour cela, on se place au début d'un passage, en supposant la propriété vraie, puis on montre que la propriété est toujours vraie à la fin du passage, avec les valeurs des variables mises à jour
4. Pour conclure, on regarde ce que dit l'invariant de boucle à la sortie de la boucle. Pour cela, il faut utiliser la condition de boucle.

UN INVARIANT 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 18. Voici une preuve (**fausse**) de l'hypothèse de Riemann :

Démonstration. On considère l'algorithme suivant :

Algorithme 16 : `Resoudre_Riemann`

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

```
// Invariant de boucle: 1 + 1 = 2
tant que 1 = 0 faire
    Se tourner les pouces;
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 **toujours** 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.

Exercice 5. Montrons la correction de l'algorithme suivant :

Algorithme 17 : `est_premier`

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

```
// Tester tous les diviseurs potentiels entre 2 et p − 1
d ← 2;
// Invariant: aucun nombre entre 2 et d − 1 ne divise p
tant que d < p et p%d ≠ 0 faire
    d ← d + 1;
si d = p alors
    retourner Oui
sinon
    retourner Non
```

Question 1. Donner un variant de la boucle while et prouver la terminaison de cet algorithme

Question 2. Traduire l'invariant proposé en une formule mathématique.

Question 3. Montrer que l'invariant donné en commentaire en est bien un, et l'utiliser pour prouver la correction de l'algorithme.

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 9. 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 19. 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 10. 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.

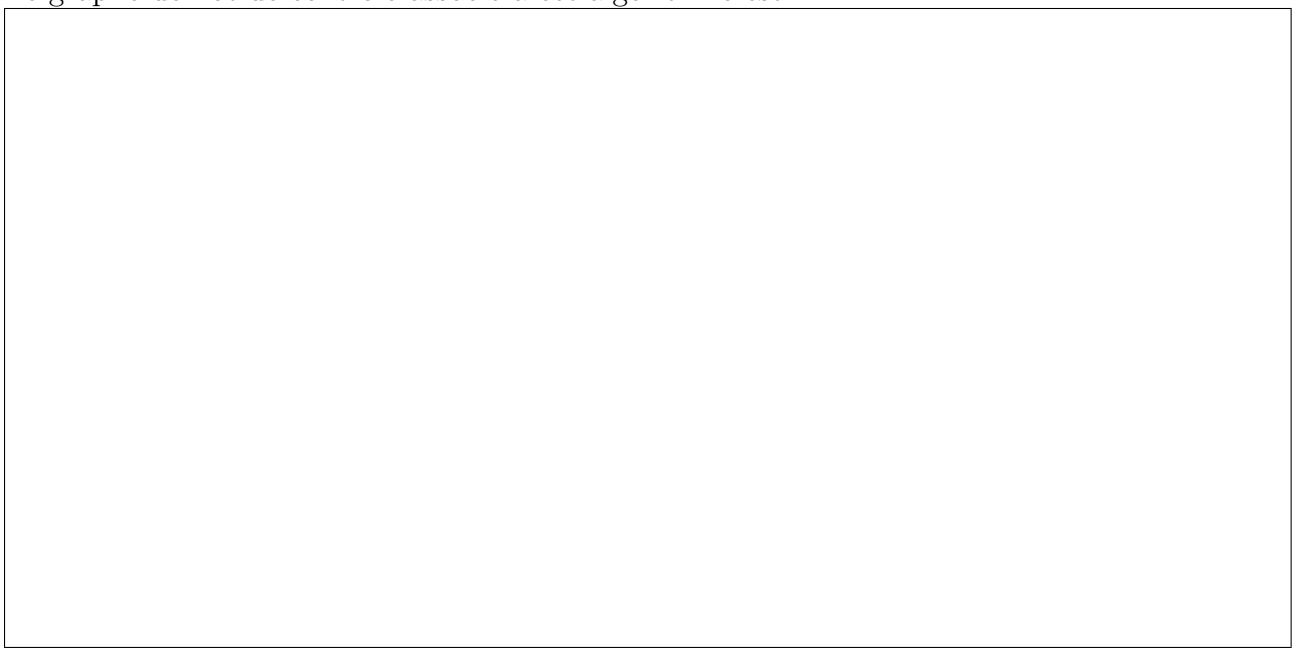
Exemple 20. On reprend l'algorithme d'exponentiation rapide :

Algorithme 18 : 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 $n = 7$.

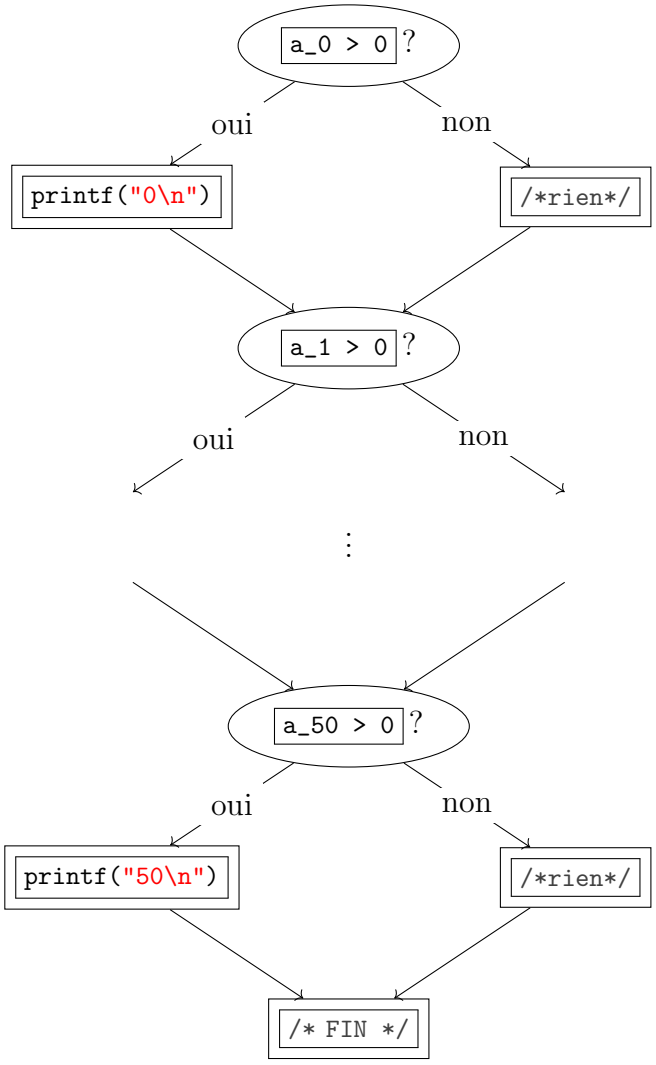
Le CFG sert entre autres à déterminer la pertinence d'une batterie de tests. Idéalement, on voudrait qu'une batterie de tests essaie tous les chemins possibles dans le CFG. Cependant ce n'est jamais possible en pratique : dès qu'une boucle apparaît, le nombre de chemins devient infini, et même sans boucle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

Exemple 21. 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 11.

— 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 2. 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.