Chapitre 1

Preuves de programmes

Notions	Commentaires
Terminaison. Correction partielle. Correction totale. Variant. Invariant.	La correction est partielle quand le résultat est correct lorsque l'algorithme s'arrête, la correction est totale si elle est partielle et si l'algorithme termine.

Extrait de la section 1.1 du programme officiel de MP2I : « Algorithmes et programmes ».

Notions	Commentaires
Spécification des données attendues en entrée, et fournies en sortie/retour.	On entraîne les étudiants à accompagner leurs programmes et leurs fonctions d'une spécification. Les signatures des fonctions sont toujours précisées.

Extrait de la section 1.2 du programme officiel de MP2I : « Discipline de programmation ».

Sommaire

0 Spécification d'une fonction

Rappels du Chapitre 0 (Introduction) :

- La signature d'une fonction est la donnée de son identifiant, du type de chacune de ses entrées et du type de sa sortie.
- Le prototype d'une fonction est la même chose à ceci près que l'on indique en plus le nom de chacun des variables.

```c
bool est_positif(int x) {
  return (x >= 0);
}
```

- Signature : `est_positif` : `int` $\longrightarrow$`bool`
- Prototype : `est_positif` : `(x : int)` $\longrightarrow$`bool`

Remarque.

- La façon d'écrire les prototypes « à la mathématiques » ci-dessus ressemble à ce que l'on utilisera en OCaml.
- En C, il est valide d'annoncer qu'une fonction existera [1] en déclarant uniquement sa signature ou son protoype : `bool est_positif(int);` ou `bool est_positif(int x);` .

Définition 1 (Spécification).
La **spécification** d'une fonction est la description de ce qu'elle est censée faire, c'est à dire qu'elle indique :

- Entrées :
  - pour chacune d'entre elles, son type.
  - les éventuelles conditions supplémentaires que ces entrées doivent respecter. Par exemple : « x, un entier > 3 » ou encore « T, un tableau d'entiers triés par ordre croissant ».
    De telles conditions sont appelées des **pré-conditions**.
- Sortie :
  - le type de la valeur renvoyée.
  - les conditions attendues sur la valeur renvoyée (généralement en lien avec les entrées).
  - les éventuelles modifications que fait la fonction sur « le monde extérieur », c'est à dire sur des variables (ou objets mémoire) que la fonction n'a pas créé elle-même : modification d'un argument passé par référence, modification d'une variable globale, enregistrement d'un fichier sur le disque dur, affichage sur un écran, etc. On parle d'**effets secondaires** (« *side effects* »).

On appelle **post-conditions** la donnée des conditions attendues sur la valeur renvoyée et des effets secondaires.

Remarque.

- On peut voir la spécification comme un contrat entre l'utilisateur de la fonction et celle-ci : l'utilisateur de la fonction s'engage à vérifier les pré-conditions, en échange de quoi la fonction s'engage à vérifier les post-conditions.
  Si les pré-conditions ne sont pas remplies lors d'un appel, la fonction n'est tenue à rien et peut renvoyer tout et n'importe quoi. Cependant, il est pertinent de plutôt lever un message d'erreur et interrompre le programme dans ce cas (beaucoup plus simple à déboguer !).
- Quand il n'y a pas d'effets secondaires, on ne les spécifie pas (au lieu de spécifier leur absence).
- L'anglicisme « effets de bord » pour « effets secondaires » existe.
- On essaye de limiter les effets secondaires au strict nécessaire, et on veille à toujours être exhaustif quand on les liste. Il s'agit de l'une des sources de bogues les plus communes !

1. On s'engage à en fournir le code source tôt ou tard.

Exemple.

```c
/** Renvoie le maximum de deux entiers
 * Entrées : a et b deux entiers relatifs
 * Sortie : le maximum de a et b
 */
int max(int a, int b) {
  if (a > b) {
    return a;
  }
  else {
    return b;
  }
}
```

La fonction `max` est bien spécifiée et respecte cette spécification.

```c
/** Renvoie le maximum d'un tableau
 * Entrées : T, un tableau de L entiers
 *           L, un entier positif
 * Sorties : le maximum des L premières cases de T
 * Eff. Sec. : euh?...
 */
int apres_moi_le_deluge(int L, int T[]) {
  for (int i = 1; i < L; i = i+1) {
    T[i] = max(T[i-1], T[i]);
  }
  // à la fin, la dernière case est
  // plus grande que toutes les autres
  return T[L-1];
}
```

La fonction `apres_moi_le_deluge` a des effets secondaires mal spécifiés… et pire encore : absolument évitables !

Exercice. Parmi les 4 fonctions `maxi_vX` ci-dessous, lesquelles respectent leur spécification ?

```c
/** Renvoie le maximum des premières cases d'un tableau
 * Entrées : n un entier >= 1
 *           T un tableau d'au moins n entiers
 * Sortie : l'élément maximal de T
 */
int maxi_v0(int n, int T[]) {
  int sortie = T[0];
  int i = 0;
  while (i < n) {
    if (T[i] > sortie) {
      sortie = T[i];
      i = i+1;
    }
  }
  return sortie;
}
```

```c
/** Renvoie le maximum des premières cases d'un tableau
 * Entrées : n un entier >= 1
 *           T un tableau d'au moins n entiers
 * Sortie : l'élément maximal de T
 */
int maxi_v1(int n, int T[]) {
  int sortie = 666;
  for (int i = 0; i < n; i = i+1) {
    sortie = max(sortie, T[i]);
  }
  return sortie;
}
```

```c
/** Renvoie le maximum des
 *  premières cases d'un
    ↪ tableau
 * Entrées : n un entier >= 1
 *           T un tableau d'au
    ↪ moins n entiers
 * Sortie : l'élément maximal
    ↪ de T
 */
int maxi_v2(int n, int T[]) {
  int sortie = T[0];
  for (int i = 1; i < n; i =
   ↪ i+1) {
    sortie = max(sortie,
     ↪ T[i]);
  }
  return sortie;
}
```

```c
/** Renvoie le maximum des
 *  premières cases d'un
    ↪ tableau
 * Entrées : n un entier >= 1
 *           T un tableau d'au
    ↪ moins n entiers
 * Sortie : l'élément maximal
    ↪ de T
 */
int maxi_v3(int n, int T[]) {
  int sortie = T[0];
  int i = 0;
  while (i < n) {
    sortie = max(sortie,
     ↪ T[i]);
    i = i+1;
  }
  return sortie;
}
```

1 Terminaison

Convention 2 (Notation prime).
Lorsque l'on analyse une boucle, si x désigne une valeur au début d'une itération, on notera x' la valeur associée en fin d'itération (c'est à dire juste avant le début de l'itération suivante).
De même lorsque l'on analyse une fonction récursive : le prime désigne la valeur au début de l'appel récursif suivant.

Cette convention est personnelle, et même si elle est partagée par plusieurs de mes collègues, je vous recommande très chaudement de la rappeler en début d'une copie de concours avant de l'utiliser.

1.0 Variants et preuves de terminaison

Définition 3 (Terminaison).
On dit qu'une fonction termine si pour toutes entrées vérifiant les pré-conditions de la fonction, la suite d'instruction exécutée par la fonction lors d'un appel sur ces entrées est finie.

Remarque. En pratique, on aimerait aussi que la suite d'instruction soit finie même si, par erreur, on ne remplit pas les pré-conditions…

Définition 4 (Variant - semestre 1).
Un **variant de boucle** (TantQue ou Pour) est une quantité :

- entière
- minorée (resp. majorée) tant que l'on ne quitte pas la boucle.
- strictement décroissante (resp. strictement croissante) d'une itération sur l'autre.

Remarque. Le terme quantité est volontairement général : il peut s'agir de la valeur d'une variable, de la différence de deux variables, ou de toute autre opération savante impliquant au moins une variable.

Exemple.

```c
int somme(int n) {
  int s = 0;
  int i = 0;
  while (i < n+1) {
    s = s+i;
    i = i+1;
  }
  return s;
}
```

Dans la boucle `while` de fonction `somme` ci-contre, la quantité i est un variant. En effet, elle est :

- entière car `i` est un entier.
- strictement croissante car si l'on note `i'` la valeur de `i` à la fin d'une itération, on a `i'` $=$ `i` $+ 1$.
- majorée car d'après la condition de boucle, `i` $<$ `n` $+ 1$ (où `n` ne varie jamais).

Théorème 5 (Preuves de terminaison).
Toute boucle qui admet un variant termine.
Toute fonction dont toute les boucles terminent *et* dont tous les appels de fonction terminent termine.

Démonstration. Traitons les deux points séparément :

- Considérons une boucle qui admet un variant V. Quitte à adapter la preuve, supposons ce variant str. décroissant minoré et notons m un tel minorant. Numérons les itérations de la boucle : $0, 1, \ldots$ et notons $V_0, V_1, \ldots$ les valeurs du variant V au début de chacune de ces itérations.
  Comme la suite des V_i est strictement décroissante et que ses termes sont des entiers, on pour tout $i \in \mathbb{N}$: $V_i - 1 \geq V_{i+1}$. Il s'ensuit que si les itérations existent au moins jusqu'au rang $i = V_0 - m + 1$, on a $V_{V_0-m-1} \leq m - 1$. Or cela est impossible car m est un minorant : en particulier, les itérations s'arrêtent avant d'atteindre ce rang. C'est à dire que la boucle termine.
- Les seules instructions vues jusqu'à présent succeptibles de ne pas terminer sont les boucles. Si toutes les boucles de la fonction termine, et que toutes les fonctions appelées (donc totues *leurs*) boucles terminent, la fonction termine bien.
  NB : nous verrons dans quelques mois la non-terminaison des appels récursifs… mais c'est un cas particulier de la non-terminaison d'un appel de fonction. L'énoncé du théorème est donc valide.

□

Exemple. La fonction `somme` de l'exemple précédent termine.

Exemple.

```c
int sup_log_2(int x) {
  int n = 0;
  int p = 1;
  while (p < x) {
    p = 2*p;
    n = n+1;
  }
  return n;
}
```

Dans la boucle `while` de fonction `sup_log_2` ci-contre, la quantité $V =$ `x`$-$`p` est un variant. En effet, elle est :

- entière car `x` et `p` sont des entiers.
- strictement décroissante. En effet, comme `p` $>$ 0 (il l'est au début de la première itération et n'est ensuite que multiplié par 2), on a `p'` $>$ `p`. Or, `x'` $=$ `x`, donc $V' < V$.
- minorée car d'après la condition de boucle, $0 <$ `x` $-$ `p`.

Donc la boucle termine. Comme la fonction n'a pas d'autres boucles ni d'appels de fonctions, elle termine.

Profitons de cet exemple pour mentionner une autre façon de comprendre $\lceil \log_2 x \rceil$: c'est le nombre de fois qu'il faut diviser x par 2 pour atteindre 1 ou moins. C'est donc réciproquement l'exposant de la plus petite puissance de 2 qui atteint x ou plus.

Remarque.

- Étant donné une fonction, il peut parfois être impossible de prouver qu'elle termine et impossible de prouver qu'elle ne termine pas. On termes ~~savants~~ de MPI, on dit que « le problème de l'arrêt n'est pas décidable ».

  Ce n'est pas un problème qui se pose à notre modeste niveau.
- Les preuves de variant vous rappellent peut-être les récurrences mathématiques : il y a un lien profond entre les deux que nous apercevrons plus en détails au second semestre.

1.1 Cas particulier des boucles Pour

Pour étudier la terminaison d'une boucle Pour, on la déplie ainsi :

```
1 pour i allant de 0 inclu
    à n exclu faire                    1 i ←0
2 |  corpsDeLaBoucle          ⟼       2 tant que i < n faire
                                       3 |  corpsDeLaBoucle
                                       4 |  i ← i + 1
```

C'est à dire que l'on se ramène à la boucle `while` dont le `for` est un raccourci. On fai tbien attention au fait que :

- Le compteur de boucle (i ci-dessus) est initialisé juste avant la première itération.
- La mise à jour du compteur a lieu à la toute fin d'une itération.

Remarque.

- Parfois, la mise à jour du compteur n'est pas une simple incrémentation de 1. Ca ne change rien, on procède pareil.
- Dans certains langages (dont le C), il est possible que le corps de la boucle Pour modifie le compteur. C'est une mauvaise pratique qu'il ne faut pas faire : il faut utiliser un `while` dans ce cas !
- Je parlerais de « **vraie boucle Pour** » pour désigner une boucle où le compteur n'est pas modifié par le corps, où sa mise à jour est une incrémentation d'une quantité constante (pas forcément 1), et où le compteur est borné par une quantité fixée (cohérente avec la monotonie du compteur).

  Ce n'est pas un terme canonique, et je vous conseille de le redéfinir si vous souhaitez l'utiliser.

Proposition 6 (Terminaison des vraies Pour).
Une « vraie boucle Pour » admet toujours un variant. En particulier, elle termine.

Démonstration. D'après la définition de vraie boucle Pour, le compteur est un variant immédiat. □

Exemple.

```c
int sommebis(int n) {
  int s = 0;
  for (int i = 0; i < n+1; i = i+1) {
    s = s+i;
  }
  return s;
}
```

La boucle pour de fonction `sommebis` ci-contre est une « vraie boucle Pour », c'est à dire que son compteur i est entier, strictement croissante (car `i'` $= \texttt{i} + 1$) et borné par $\texttt{n} + 1$ constant. La boucle termine donc.
Puisqu'il n'y a pas d'autres boucles ni d'appels de fonctions, il s'ensuit que `sommebis` termine.

1.2 Fonctions récursives

Pour prouver la terminaison d'une fonction récursive, on doit montrer que la suite d'appels récursifs termine. Pour cela, on utilise un **variant d'appels récursifs** : c'est comme un variant, sauf qu'on demande la stricte monotonie non pas d'une itération à l'autre mais d'un appel à l'autre. La minoration correspond en général au cas de base de la fonction récursive.

Exemple.

```c
/** Renvoie x^n
 * Entrée : x un entier
 *          n entier >= 0
 * Sortie : x^n
 */
int exp_rap(int x, int n) {
  assert(n >= 0);

  /* Cas de base */
  if (n == 0) { return 1; }
  else if (n == 1) { return x; }

  /* Sinon, cas récursifs */
  int x_puiss_demi = exp_rap(x, n/2);
  if (n % 2 == 0) {
    return x_puiss_demi * x_puiss_demi;
  } else {
    return x * x_puiss_demi * x_puiss_demi;
  }
}
```

Prouvons que cette fonction récursive termine. Commençons par remarquer qu'elle ne contient ni boucles ni appels à une autre fonction : il suffit de montrer que la suite d'appels récursifs termine. Pour cela, prouvons que la quantité `n` est un variant d'appels récursifs :

- `n` est un entier (ligne 179).
- `n` est minoré par 0 d'après les pré-conditions. Notons que si `n` atteint 0 ou 1 alors la fonction termine aux lignes 183-184.
- `n` décroit strictement d'un appel au suivant. d'après le code et le point précédent la fonction peut s'appeler récursivement si $\texttt{n} > 2$. Dans ce cas, elle s'appelle récursivement avec l'argument `n'` qui vaut $\frac{\texttt{n}}{2}$. On a bien $\texttt{n'} < \texttt{n}$.

Comme la suite d'appels récursifs admet un variant, elle termine, et d'après l'analyse initiale la fonction `exp_rap` termine.

Remarque.

- Dans cet exemple, le variant est un argument. Mais cela peut aussi être la différence de plusieurs arguments (par exemple pour une dichotomie) ou le nombre d'éléments d'une liste.
- Au second semestre, nous verrons les *inductions structurelles* qui permettent souvent de prouver une terminaison récursive de manière plus proche du code, et donc plus simple à comprendre.
- Il est souvent pertinent de faire la minoration avant la stricte décroissance. Par exemple, dans cette preuve, cela m'a permet d'utiliser implicitement le fait que $\texttt{n} \geq 0$ ce qui était nécessaire pour conclure que $\texttt{n'} < \texttt{n}$.
- S'il y a plusieurs appels récursifs, il faut prouver la terminaison de chacun d'entre eux et donc prouver la stricte monotonie pour chacun d'entre eux.

2 Correction

2.0 Correction partielle

Définition 7 (Correction partielle).
On dit qu'une fonction est **partiellement correcte** si pour toutes entrées vérifiant les pré-conditions et sur lesquelles la fonction termine, le résultat de l'appel de la fonction vérifie les post-conditions.

Remarque.

- C'est équivalent à « pour toutes entrées vérifiant les pré-conditions, soit l'appel de la fonction ne termine pas soit son résultat vérifie les post-conditions ».
- Comme pour la terminaison, on voudrait que la fonction soit bien élevée même lorsque ses préconditions ne sont pas remplies : le mieux est que dans ce cas elle termine rapidement en renvoyant un message d'erreur (et n'ait pas d'effets secondaires ‼).

2.0.0 Sauts conditionnels

Pour prouver la correction partielle d'une fonction qui ne contient que des déclarations/modifications de variables et des sauts conditionnels, il suffit de faire une disjonction de cas sur les `if-then-else`.

Exemple. Prouvons la correction partielle de la fonction ci-dessous :

```c
/** Renvoie le maximum de deux entiers
 * Entrées : a et b deux entiers relatifs
 * Sortie : le maximum de a et b
 */
int max(int a, int b) {
  if (a > b) {
    return a;
  }
  else {
    return b;
  }
}
```

Notons tout d'abord que comme annoncé, la fonction n'a pas d'effets secondaires (les deux ragument sont passés par valeur). Pour prouver la post-condition de la sortie, procédons comme le code (ligne 9) par disjonction de cas :

- Si `a` $>$ `b`, on entre dans le `then` ligne 10 : on renvoie `a`, qui est bien $\max(\texttt{a}, \texttt{b})$ comme annoncé.
- Sinon, on entre dans le `else` (ligne 13) : on renvoie `b`, qui est bien $\max(\texttt{a}, \texttt{b})$ comme annoncé.

La fonction est donc partiellement correcte.

Remarque. Il est rare que, comme dans cet exemple, les sauts conditionnels demande la majeure partie du travail de la preuve : c'est généralement un point facile à prouver sur lequel on passe rapidement pour ne pas alourdir la rédaction.

2.0.1 Invariants de boucle

Convention 8 (Somme vide).
Lorsque l'on écrit $\sum_{k=a}^{b}$ ..., on considère que si $a > b$ alors la somme est vide est vaut 0.
De même avec le symbole produit Π.

Définition 9 (Invariant de boucle).
Un **invariant de boucle** (TantQue ou Pour) est une propriété logique qui :

- est vraie juste avant la toute première itération de la boucle (on appelle cela l'**initialisation**).
- si elle est vraie au début d'une itération, est vraie à la fin de cette même itération (on parle de **conservation**).

Remarque.

- Un invariant, comme un variant, doit impliquer des variables de la fonction pour être utile.
- C'est en fait une récurrence ! L'avantage de ce formalisme-ci par rapport à une récurrence sur les itérations est que l'on a pas besoin d'introduire un numéro d'itération superflu. L'autre avantage est que l'on compare le début d'une itération à la fin de celle-ci, là où une récurrence compererait au début de l'itération suivante... laquelle n'existe peut-être pas.
- Comme pour les preuves de terminaison, on utilise la notation prime pour marquer une quantité en fin d'itération.
- Pour prouver la conservation dans une boucle `while`, il est souvent utile d'utilise le fait que l'on est encore en train d'itérer la boucle et que donc la condition de la boucle est vraie : elle est une hypothèse que l'on peut appliquer.
- Une *très* bonne pratique est de ne pas seulement prouver que quelque chose est un invariant, mais aussi de préciser ce qu'il prouve à la fin de la dernière itération. On nomme cette étape la **conclusion**.

Exemple.

```c
/** Somme des n premiers entiers.
 * Entrées : n >= 0
 * Sortie : somme des n premiers entiers de N*.
 * Eff. Sec : aucun.
 */
int somme(int n) {
  int s = 0;
  int i = 0;
  while (i < n+1) {
    s = s+i;
    i = i+1;
  }
  return s;
}
```

Montrons que I : «`s` $= \sum_{k=0}^{\texttt{i}-1} k$» est un invariant.

- Initialisation : avant la première itération de la boucle, on a `s` $= 0$. On a également `i` $- 1 < 0$, d'où la somme de I est vide et vaut donc 0. On a bien $0 = 0$: l'invariant est initialisé.
- Conservation : supposons l'invariant I vrai vrai au début d'une itération quelconque et montrons-le vrai à la fin de cette itération, c'est à dire montrons que I' : « `s'` $= \sum_{k=0}^{\texttt{i'}-1} k$ ».
  On a :

$$\begin{cases} \texttt{s'} = \texttt{s} + \texttt{i} & \text{(ligne 108)} \\ \texttt{i'} = \texttt{i} + 1 & \text{(ligne 109)} \end{cases}$$

Or d'après I, on a $\texttt{s} = \sum_{k=0}^{\texttt{i}-1} k$. Donc :

$$\begin{aligned} \texttt{s'} &= \texttt{s} + \texttt{i} \\ &= \sum_{k=0}^{\texttt{i}-1} k + \texttt{i} && \text{d'après } I \\ &= \sum_{k=0}^{\texttt{i}} k \\ &= \sum_{k=0}^{\texttt{i'}-1} k && \text{car } \texttt{i'} = \texttt{i} + 1 \end{aligned}$$

On a prouvé que I' est vrai : l'invariant se conserve.

- Conclusion : On a prouvé que I est un invariant. En particulier, comme à la fin de la dernière itération on a $\texttt{i'} = \texttt{n} + 1$, on a en sortie de boucle :

$$\begin{aligned} \texttt{s} &= \sum_{k=0}^{\texttt{n}+1-1} k \\ &= \sum_{k=0}^{\texttt{n}} k \end{aligned}$$

Remarque.

- J'ai volontairement beaucoup détaillé les étapes de calcul de cette preuve, afin de limiter les difficultés calculatoires. On aurait pu aller plus vite.
- Toutefois, je vous demande de ne pas aller plus vite que moi sur la phrase d'introduction de la conservation : « *supposons l'invariant [...] vrai au début d'une itération quelconque et montrons-le vrai à la fin de cette itération, c'est à dire montrons que [...]'* ». En effet, je veux que vous ayez chacun des éléments de cette phrase sous les yeux avant de vous lancer dans la conservation en elle-même. [2]

Proposition 10.
Pour prouver la correction partielle d'une fonction qui contient des boucles, on fait appel à de sinvariants de boucle afin de prouver qu'une certaine propriété est vraie au moment de la sortie de la boucle.
Autrement dit, les invariants jouent le rôle d'un lemme : ils *ne* prouvent *pas* la correction partielle, mais on espère pouvoir utiliser leur résultat pour en déduire la correction partielle.

Exemple. Dans l'exemple précédent, l'invariant choisi prouve qu'en sortie de boucle, on a $\texttt{s} = \sum_{k=0}^{\texttt{n}} k$. Comme `s` n'est pas modifié entre la sortie de la boucle et le `return s` , on peut conclure que la sortie renvoyée vérifié la condition annoncée. De plus, il n'y a pas d'effets secondaires (une seule entrée, passée par valeur) comme annoncés : les post-conditions sont vérifiées, donc la fonction est partiellement correcte.

Exemple. Prouvons la correction partielle de la fonction ci-dessous. On utilisera la propriété suivante pour qualifier $\lceil \log_2 x \rceil$ (avec $x \geq 1$) : c'est l'unique entier ℓ tel que $2^{\ell-1} < x \leq 2^{\ell}$.

2. Comme, peut-être, lors de votre apprentissage des récurrences en Terminale.

```c
/** Partie entière supérieure de log_2(x)
 * Entrées : x entier >= 1
 * Sortie : la partie entière supérieure du
 *          logarithme en base 2 de x
 * Eff. Sec : aucun.
 */
int sup_log_2(int x) {
  int n = 0;
  int p = 1;
  while (p < x) {
    p = 2*p;
    n = n+1;
  }
  return n;
}
```

On veut prouver l'encadrement voulu pour la ligne 136, c'est à dire en fin de boucle `while`.

Puisque l'on a quitté la boucle, on a alors $\mathtt{p} \geq \mathtt{x}$. En déroulant les étapes à la main, on se rend compte que I : « $\mathtt{p} = 2^{\mathtt{n}}$ » est un invariant [3] : nous allons donc le prouver. Nous aurons aussi besoin de l'invariant J : « $\frac{\mathtt{p}}{2} < \mathtt{x}$ » qui donnera l'autre moitié de l'encadrement voulu [4].

- Initialisation : avant la première itération, on a p = 1 et n = 0, or $1 = 2^0$ donc J est bien initialisé. De plus, d'après les pré-conditions on a $\mathtt{x} \geq 1$. Comme $\frac{\mathtt{p}}{2} = \frac{1}{2} < 1$, on a bien l'initialisation de I.
- Conservation : supposons les invariants I et J vrais au début d'une itération quelconque et montrons-les vrais à la fin de cette itération, c'est à dire montrons que I' : « $\mathtt{p} = 2^{\mathtt{n}}$ » et que J' : « $\frac{\mathtt{p}'}{2} < \mathtt{x}$ ».
  On a :
  $$\begin{cases} \mathtt{p}' = 2\mathtt{p} & \text{(ligne 133)} \\ \mathtt{n}' = \mathtt{n} + 1 & \text{(ligne 134)} \end{cases}$$
  En appliquant I dans la définition de `p'`, on obtient :
  $$\mathtt{p}' = 2 \times 2^{\mathtt{n}} = 2^{\mathtt{n}+1} = 2^{\mathtt{n}'}$$
  C'est à dire I'. Pour prouver J', utilisons le fait qu'au début de chaque itération on a $\mathtt{p} < \mathtt{x}$. Donc d'après al définition de `p'` :
  $$\frac{\mathtt{p}'}{2} = \mathtt{p} < \mathtt{x}$$
  C'est à dire J'. Les deux invariants se conservent.
- Conclusion : On a prouvé que I et J sont des invariants. En particulier, en sortie de boucle on a $\mathtt{p} = 2^{\mathtt{n}}$ et $\mathtt{x} > \frac{\mathtt{p}}{2} = 2^{\mathtt{n}-1}$.

L'évaluation de J en sortie de boucle nous donne la minoration de l'encadrement attendu. Pour obtenir la majoration, remarquons qu'on quitte la boucle car $\mathtt{p} \geq \mathtt{x}$, c'est à dire d'après I car $2^{\mathtt{n}} \geq \mathtt{x}$.

On a ainsi prouvé l'encadrement demandé : la fonction renvoie bien $\lceil \log_2 \mathtt{x} \rceil$. Remarquons enfin qu'elle n'a aucun effets secondaires comme annoncé : la fonction est donc partiellement correcte.

Remarque. Les invariants peuvent aussi servir à prouver une terminaison (on prouve que la minoration/majoration du variant est un invariant) ou des non-terminaisons (on prouve que la condition d'un `while` est toujours fausse).

3. Qui sera très utile puisque l'on veut encadrer x par des puissances de 2 : nous avons la majoration grâce à cet invariant !
4. Celui-ci est peut-être un peu moins évident : on peut le trouver en essayant de conclure avec uniquement I, on se rend alors compte qu'il nous manque la moitié de l'encadrement mais qu'elle forme un invariant pas méchant.

2.0.2 Cas particulier des boucles Pour

Comme pour la terminaison, on déplie les boucles Pour :

```
1 pour i allant de 0 inclu
    à n exclu faire
2 |   corpsDeLaBoucle
```

⟶

```
1 i ←0
2 tant que i < n faire
3 |   corpsDeLaBoucle
4 |   i ← i + 1
```

En particulier :

- Le compteur est créé juste avant la première itération, et l'initialisation des invariants se fait donc avec cette valeur.
- Le compteur est mis à jour juste avant la fin de l'itération, et donc sa valeur en fin d'itération (`i'`) n'est plus celle en début (`i`).
- On quitte la boucle car le compteur dépasse : en sortie de boucle, le compteur contient la valeur mise à jour du compteur (dans l'exemple ci-dessus, `i` vaut `n` en sortie de boucle).
  Attention, cette valeur mise à jour n'est pas forcément la borne du for : pensez par exemple à `for (int i = 0; i < 3; i = i+2)` .
  On se contente généralement d'affirmer cette valeur en sortie de boucle au lieu de la prouver.

Exemple. Prouvons la correction partielle de la fonction ci-dessous :

```c
148 int exp_simple(int a, int n) {
149   int p = 1;
150   for (int i = 0; i < n; i = i+1) {
151     p = a*p;
152   }
153   return p;
154 }
```

On veut prouver qu'en sortie de boucle, $\mathtt{p} = \mathtt{a}^{\mathtt{n}}$. Pour cela, montrons que I : « $\mathtt{p} = \mathtt{a}^{\mathtt{i}}$ » est un invariant.

- Initialisation : Juste avant la première itération, on a $\mathtt{p} = 1$ et $\mathtt{i} = 0$. On a bien $1 = \mathtt{a}^0$, d'où l'initialisation.
- Hérédité : supposons l'invariant I vrai au début d'une itération quelconque et montrons-le vrai à la fin de cette itération, c'est à dire montrons que I' : « $\mathtt{p}' = \mathtt{a}^{\mathtt{i}'}$ ».
  Or :

$$\begin{aligned} \mathtt{p}' &= a \times \mathtt{p} && \text{d'après ligne 151} \\ &= \mathtt{a} \times \mathtt{a}^{\mathtt{i}} && \text{d'après } I \\ &= \mathtt{a}^{\mathtt{i}+1} \\ &= \mathtt{a}^{\mathtt{i}'} && \text{d'après ligne 150} \end{aligned}$$

  C'est à dire I' : l'invariant est conservé.
- Conclusion : On a prouvé que I est un invariant. En particulier, comme en sortie de boucle $\mathtt{i} = \mathtt{n}$, on a alors :

$$\mathtt{p} = \mathtt{a}^{\mathtt{n}}$$

C'est exactement la condition attendue sur `p`. De plus, la fonction n'a pas d'effets secondaires, comme annoncé. Elle est donc partiellement correcte.

2.1 Fonctions récursives

Pour les fonctions récursives, on prouve les invariants par... récurrence ! Il s'agit de prouver que la spécification est vérifiée, en supposant que l'appel récursif la vérifie déjà (hérédité). Il faut bien sûr également prouver que les cas de base la vérifient (initialisation).

Exemple.

```c
/** Renvoie x^n
 * Entrée : x un entier
 *          n entier >= 0
 * Sortie : x^n
 */
int exp_rap(int x, int n) {
  assert(n >= 0);

  /* Cas de base */
  if (n == 0) { return 1; }
  else if (n == 1) { return x; }

  /* Sinon, cas récursifs */
  int x_puiss_demi = exp_rap(x, n/2);
  if (n % 2 == 0) {
    return x_puiss_demi * x_puiss_demi;
  } else {
    return x * x_puiss_demi * x_puiss_demi;
  }
}
```

Prouvons que cette fonction est partiellement correcte. Commençons par noter qu'elle n'a pas d'effets secondaires, comme demandé.

Montrons par récurrence forte sur $n \in \mathbb{N}$ la propriété H_n : « pour tout x entier, `exp_rap(x,n)` renvoie x^n ».

- Initialisation : si $n = 0$ (resp. si $n = 1$), la fonction renvoie 1 qui vaut bien x^0 (resp. x qui vaut bien x^1). D'où l'initialisation.
- Hérédité : soit n entier strictement supérieur à 1 tel que la propriété soit vraie jusqu'au rang $n-1$. Montrons-la vraie au rang n, c'est à dire montrons que `exp_rap(x,n)` renvoie x^n.
  Comme $n > 1$, l'exécution ne rentre pas dans les `if-else` (ni dans le `assert`) et on va donc en ligne 187. Par hypothèse de récurrence, `x_puiss_demi` contient $x^{\lfloor \frac{n}{2} \rfloor}$.
  Or, d'après le cours de maths :
  - Si n est pair, on a $\lfloor \frac{n}{2} \rfloor = \frac{n}{2}$. Donc :
    $$x^{\lfloor \frac{n}{2} \rfloor}.x^{\lfloor \frac{n}{2} \rfloor} = x^n$$
  - Sinon, on a $\lfloor \frac{n}{2} \rfloor = \frac{n-1}{2}$. Donc :
    $$x.x^{\lfloor \frac{n}{2} \rfloor}.x^{\lfloor \frac{n}{2} \rfloor} = x.x^{n-1} = x^n$$

  Or, cette disjonction de cas et ces calculs sont exactement les liens 188-192 du code : la fonction renvoie donc bien dans tous les cas x^n. D'où l'hérédité.
- Conclusion : On a prouvé par récurrence forte que pour tout n, pour tout x, la fonction renvoie bien le résultat annoncé.

Les post-conditions sont donc vérifiées : la fonction est partiellement correcte.

Remarque. S'il y a plusieurs appels récursifs, on doit utiliser l'hypothèse de récurrence sur chacun d'entre eux.

2.2 Correction totale

Définition 11 (Correction totale).
On dit qu'une fonction est **totalement correcte** lorsqu'elle termine et qu'elle est partiellement correcte.

Exemple.

- La fonction `somme` des exemples précédents est totalement correcte : on a prouvé sa terminaison page 19 et correction aux pages 23 et 24.
- La fonction `sup_log_2` aussi : terminaison page 19 et correction page 24.
- La fonction `exp_rap` aussi : terminaison page 21 et correction page 27.
- La fonction `max` termine car elle ne contient ni boucle ni appels de fonction, et on a prouvé sa correction page 22 : elle est aussi totalement correcte.
- La fonction `patience` ci-dessous est partiellement correcte mais ne termine pas, donc n'est pas totalement correcte :

```c
/** Renvoie true
  * Entrée : rien
  * Sortie : true
  */
bool patience(void) {
        while (true) {
                int sert_a_rien = 0;
        }
        return true;
}
```

Remarque. La correction totale (que ce soit la terminaison ou la correction partielle) d'une fonction dépend de la correction totale des fonctions qu'elle appelle ! Il faut donc commencer par les prouver (ou les admettre s'il s'agit de fonctions fournies par des librairies que l'on a pas codées ou par l'énoncé.)

Exercice. Prouver que la fonction `maxi_v3` ci-dessous est totalement correcte :

```c
/** Renvoie le maximum des
 *  premières cases d'un tableau
 * Entrées : n un entier >= 1
 *           T un tableau d'au moins n entiers
 * Sortie : l'élément maximal de T
 */
int maxi_v3(int n, int T[]) {
  int sortie = T[0];
  int i = 0;
  while (i < n) {
    sortie = max(sortie, T[i]);
    i = i+1;
  }
  return sortie;
}
```