I - Définitions

Définition Une *boucle* est une répétition d'instructions. Chaque passage dans la boucle est une *iteration*.

Rappel En C, deux mots clés permettent de définir des boucles : `for` et `while`. Pas d'inquiétude, il est aussi possible de définir des boucles en OCaml, comme nous le verrons plus tard.

Définition Un **variant** de boucle est une quantité : **entière**, **minorée**, qui décroît **strictement** à chaque passage dans la boucle.

Définition Un *invariant de boucle* est une propriété qui est vraie à l'initialisation d'une boucle et qui est conservée à l'issue d'un passage dans la boucle (si elle était vraie avant le passage).

II - Terminaison

Propriété Si une boucle admet un variant de boucle alors elle ne peut pas être infinie (donc elle termine).

A_ Exemple de l'algorithme d'Euclide

Définissons en C une fonction utilisant l'algorithme d'Euclide pour calculer le PGCD de deux entiers a et $b > 0$.

```
1  int pgcd(int a, int b)
2  {
3    assert (b > 0);
4
5    int tmp;
6    while (b != 0)
7    {
8      tmp = b;
9      b = a % b;
10     a = tmp;
11   }
12   return a;
13 }
```

La *précondition* $b > 0$ est vérifiée ici à la **ligne 3**, en utilisant la fonction `assert`.

Montrons que `b` est un *variant* de la boucle `while`.
`b` est :

- **une quantité entière ?** `b` est un entier
- **minorée ?** `b` est initialement strictement supérieur à 0 et l'exécution de la boucle s'arrête si `b = 0`
- **strictement décroissante ?** à chaque passage dans la boucle, `b` prend la valeur de `a % b`, c'est-à-dire le reste de la division euclidienne de `a` par `b`, quantité qui est, quelle que soit la valeur de `a`, **strictement** inférieure à `b`.

Ainsi, la boucle `while` admet un variant donc son exécution se termine, d'après la propriété ci-dessus.

B_ Exemple de la recherche dichotomique

Nous nous intéressons à la recherche d'un élément entier dans un tableau contenant des entiers en ordre croissant. Un exemple de code pour la fonction `recherche_dicho` est donné page suivante.
Les préconditions de la boucle sont :

- le tableau `tableau` contient des éléments en ordre trié
- l'indice `a` est un indice valide dans le tableau
- l'indice `a` est strictement inférieur à la valeur de `b` dont la valeur maximum possible est l'indice max du tableau + 1.

La fonction `recherche_dicho` contient une boucle `while`.

Montrons que la quantité représentée par `b - a` est un variant de la boucle `while`.

- `b` et `a` sont des entiers, `b - a` est un entier,
- la précondition impose que `a < b`, donc `b - a > 0` et la boucle s'arrête si `a = b`, donc si `b - a = 0`, la quantité `b - a` est donc minorée par 0,
- soit a et b vérifiant les préconditions : quelles sont les valeurs de a' et b' au prochain passage dans la boucle ? D'après le code, on calcule $i = \left\lfloor \frac{a+b}{2} \right\rfloor$. Puisque $a < b$, alors $i < b$ et $i \geq a$. Dans le cas où la condition du test de la **ligne 10** n'est pas remplie, le test de la ligne **ligne 15** implique deux cas possibles :
  - lorsque le test amène en **ligne 17**, alors $a' = a$ et $b' = i$. On a alors $b' - a' = i - a < b - a$,

- lorsque le test amène en **ligne 21**, alors $a' = i+1$ et $b' = b$. On a alors $b' - a' = b - (i+1) = b - i - 1$, or $i \geq a$ donc $b' - a' < b - a$.

Si la condition du test de la **ligne 8** est vérifiée, alors la boucle s'arrête.

Ainsi dans tous les cas, soit l'exécution de la boucle s'arrête, soit la quantité $b - a$ décroît strictement. La quantité `b - a` est donc un variant de la boucle. Cette dernière s'arrête.

```
1  int recherche_dicho(const int *tableau, int n, int e)
2  {
3    int res = -1;
4    bool trouve = false;
5    int a = 0;
6    int b = n;
7    while (a < b && !trouve)
8    {
9      int i = (a + b)/2;
10     if (tableau[i] == e)
11     {
12       trouve = true;
13       res = i;
14     }
15     else if (tableau[i] > e)
16     {
17       b = i;
18     }
19     else
20     {
21       a = i+1;
22     }
23   }
24   return res;
25 }
```

C_ Remarques

L'analyse effectuée pour les boucles est très proche des raisonnements effectués pour garantir la terminaison d'une fonction récursive. Il est tout à fait possible de parler de *variant* lors de l'étude d'une fonction récursive si on a repéré une quantité entière, minorée et strictement décroissante... ou une quantité entière, majorée et strictement croissante.

III - Correction

Prouver la correction d'un algorithme comportant une boucle nécessite de montrer qu'une propriété intéressante est un *invariant* de la boucle.

Le raisonnement sera encore très proche de ce que nous avons fait pour prouver la correction des fonctions récursives.
On démontre que la propriété $\mathcal{I}$ est vraie juste avant la boucle, c'est l'initialisation.
Puis on démontre que si la propriété est vraie en début d'itération alors elle reste vraie à la fin de l'itération, c'est l'hérédité.
On conclut en utilisant l'invariant après arrêt de l'exécution de la boucle.

A. Exemple de l'algorithme d'Euclide

Rappelons les propriétés suivantes du `pgcd` pour deux entiers u et v positifs ou nuls et $u \bmod v$, le reste de la division euclidienne de u par v :

$$\text{pgcd}(u,0) = u \quad (1)$$
$$\text{pgcd}(u,v) = \text{pgcd}(v, u \bmod v) \quad (2)$$

Invariant de boucle PGCD(a, b) = PGCD(a_0, b_0) où a_0 et b_b0 sont les valeurs de a et b à l'appel de la fonction. *Il est toujours possible de copier les valeurs d'entrées dans de nouvelles variables, pour faciliter le raisonnement.*

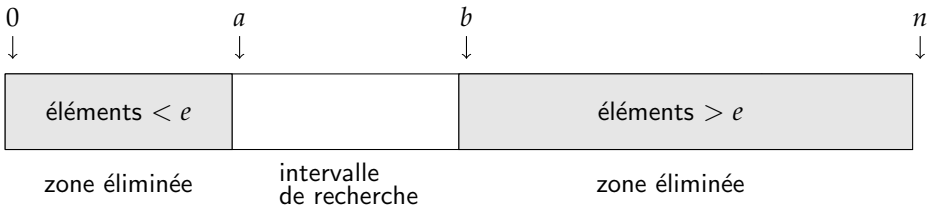
Initialisation Avant la boucle, a et b n'ont pas été modifiées et valent a_0 et b_0 donc PGCD(a, b) = PGCD(a_0, b_0).

Hérédité Supposons que PGCD(a, b) = PGCD(a_0, b_0) à l'entrée de la boucle. Alors, d'après le code, en sortie de boucle on a $a' = b$ et $b' = a \bmod b$. D'après la propriété (2), PGCD(b, $a \bmod b$) = PGCD(a, b), donc PGCD(a', b') = PGCD(a, b) et par hypothèse de récurrence, PGCD(a', b') = PGCD(a_0, b_0).

Conclusion L'invariant étant conservé, à la suite de la dernière itération, on a PGCD(a, b) = PGCD(a_0, b_0). Or puisque la boucle se termine, on a $b = 0$ et d'après la propriété (1), a = PGCD(a_0, b_0).

B. Exemple de la recherche dichotomique

Voici une représentation sur un tableau de taille n de l'effet de la recherche dichotomique en cours d'exécution.



Si à la sortie de la boucle `res != -1`, alors `res` contient l'indice d'un élément du tableau dont la valeur est égale à l'élément recherché `e`.
Il faut montrer que lorsque le résultat est `-1`, la valeur de e ne se trouvait pas dans le tableau.
Nous allons prouver l'invariant suivant : « avant l'indice `a`, tous les éléments du tableaux sont strictement inférieurs à l'élément e cherché et à partir de la position `b`, tous les éléments du tableau sont strictement supérieurs à l'élément e cherché ».

Initialisation Avant le premier passage dans la boucle, d'après le code on a `a = 0` et `b = n`. Dans ce tableau d'indices compris dans l'intervalle $[0; n-1]$, il n'y a pas d'élément avant 0, ni d'élément à partir de n. L'élément cherché ne peut donc se trouver ni dans la partie d'indices inférieurs à a ni dans la partie de positions supérieures ou égales à b. Donc l'invariant est vérifié.

Hérédité Supposons l'invariant vérifié à l'entrée d'une boucle. Alors les éléments avant l'indice a sont strictement inférieurs à l'élément cherché e et les éléments à partir de l'indice b sont strictement supérieurs à l'élément e cherché. Supposons que l'élément au milieu i de l'intervalle ne soit pas égal à l'élément e cherché. Deux cas sont possibles :

- si l'élément `tableau[i]` est strictement supérieur à `e`, alors puisque le tableau est en ordre croissant, l'élément `tableau[i]` est inférieur ou égal à tous les éléments d'indice supérieurs à `i` et l'élément cherché ne peut pas se trouver dans la partie comprise entre l'indice `i` inclus et l'indice `b`. Par hypothèse de récurrence, l'élément cherché ne peut pas se trouver dans la partie située à partir de l'indice `b` donc il ne peut pas se trouver dans toute la partie située à partir de l'indice `i`. Or d'après le code, à la fin de l'itération on a $b' = i$ et $a' = a$. Puisque les éléments dans la partie située avant l'indice `a` sont par hypothèse de récurrence strictement inférieurs à l'élément cherché, alors les éléments d'indice inférieur à `a'` sont strictement inférieurs à l'élément e et les éléments d'indice supérieur ou égal à `b'` sont strictement supérieurs à e. Donc les propriétés de l'invariant sont conservées.
- si l'élement `tableau[i]` est strictement inférieur à `e` alors puisque le tableau est en ordre croissant et par hypothèse de récurrence comme précédemment, l'élément cherché est strictement supérieur à tous les éléments situés de l'indice 0 à l'indice `i` inclus. À la fin de l'itération on a $a' = i + 1$ et $b' = b$. Par hypothèse de récurrence, les élémens d'indice supérieur ou égal à `b` sont strictement supérieurs à l'élément cherché. Encore une fois, les éléments situés avant `a'` sont strictement inférieurs à e et les éléments situés après `b'` sont strictement supérieurs à b et les propriétés de l'invariant sont conservés.

Conclusion Si l'élément n'est pas trouvé, nous avons montré que l'exécution de la boucle s'arrête lorsque $a = b$. Dans ce cas, il n'y a plus d'éléments à considérer dans le tableau, l'intervalle $[[a, a[[$ étant vide. Et puisque l'invariant est vérifié, l'élément cherché est strictement supérieur à tous les éléments de l'intervalle $[0, a-1]$ et strictement inférieurs à tous les éléments situés dans l'intervalle $[a, n-1]$. On peut donc affirmer que l'élément cherché ne se trouvait pas dans le tableau, le résultat `-1` est correct.