Terminaison de Programme

Objectif : Déterminer si un programme termine son exécution ou boucle indéfiniment.

```
int f(int n) {
    if (n % 2 == 0) f(n);
    else return 1;
}


int g(int n) {
    if (n == 0) return 1;
    else return n*g(n-1);
}
```

Dans cet exemple on voit bien que la fonction `f` ne termine pas toujours alors que la fonction `g` si.

I - Terminaison d'un programme récursif

Important : Un programme récursif se démontre par récurrence.

Énoncé : $\forall n\in\N$ g(n) termine.

Initialisation : Le programme termine directement (c'est le cas de base).

Hérédité : Pour n > 0, le programme appelle g(n-1), qui termine par HR puis g(n) termine.

Suite de Fibonacci

```
/// Suite de Fibonacci
int fibo(int n) {
    if (n==0 || n==1) return 1;
    else return fibo(n-1) + fibo(n-2);
}
```

Énoncé : $\forall n \in \N$ fibo(n) termine.

Initialisation : Pour n = 0 ou n = 1 le programme termine (cas de base).

Hérédité : Pour n > 1, on suppose que fibo(n-1) et fibo(n-2) terminent, or fibo(n) n'effectue que ces deux appels, donc fibo(n) termine également.

Suite de Syracuse

```
/// Suite de Syracuse
// pré-cond : n > 0
int syracuse(int n) {
    if (n==1) return 0;
    else if (n % 2 == 0)
        return 1 + syracuse(n/2);
    else
return 1 + syracuse(3*n + 1);
}
```

On est bloqué puisque 3n+1 < n. Il s'agit encore à ce jour d'un problème ouvert, on ne sait pas si ce programme termine.

Triangle de Pascal

Pré-condition : $n\in\N$ et $k \in [\![0, n]\!]$

```
/// Triangle de Pascal
int binom(int k, int n) {
    if (k == 0 || n == 0) return 1;
    else return binom(n-1, k-1) + binom(n-1, k);
}
```

Nous allons procéder à une récurrence sur n

Initialisation : n = 0, le programme termine.

Hérédité : n > 0, plusieurs cas :

- Si k = 0 le programme termine.
- Si k > 0 alors $n - 1 \geq 0$ et $k - 1 \geq 0$ donc ces valeurs (et k) respectent la pré-condition, donc les fonctions appelées récursivement terminent par HR.

II - Terminaison de programmes impératifs

Les questions de terminaisons ne se poseront que pour les boucles en impératif. Pour montrer la terminaison d'une boucle on exhibe un variant de boucle.

C'est une quantité mathématique définie en utilisant les variables du programme.

- Cette quantité est à valeurs dans $\N$
- Cette quantité décroît strictement dans lors d'une exécution de boucle.

Recherche dichotomique

Pré-condition : a est trié.

```
/// Recherche dichotomique
int mem(int len, int* a, int elt) {
    int i = 0;
    int j = len;
    while (i < j) {
        int mid = (j+i)/2;
        if (a[mid] == elt) return mid;
        if (a[mid] > elt) j = mid;
        else i = mid+1;
    }
    return -1;
}
```

Variant de boucle choisi : `j-i`.

Début de boucle : on a les variables i et j.

Après un tour de boucle i et j valent i' et j'.

Si a[mid] = elt : le programme termine.

SI a[mid] > elt : $i' = i$ et $j' = \lfloor (i+j)/2 \rfloor$

- Si $i+j$ est pair alors $j'-i' = (i+j)/2 - i = (j-i)/2 < j-i$ car $j-i > 0$
- si $i+j$ est impair alors $j'-i' = (i+j-1)/2 - i = (j-i-1)/2 < j-i$

Si a[mid] < elt alors les calculs sont analogues.

Donc $j-i$ décroît strictement dans $\N$, elle atteindra donc 0 ce qui provoquera l'arrêt de la boucle.

Cas des boucles for

Une boucle for peut toujours se traduire par une boucle while.

Pour `for(int i = a; i < b; i++)` on a toujours `b-i-a` comme variant. En principe elle termine toujours (sauf si mal écrite) mais par précaution on donnera le variant.

Exemple de traduction d'une boucle for en boucle while :

```
int a,b;

/// Boucle for
for(int i = a; i < b; i++) foo();


/// Boucle while associée
int i = a;
while (i < b) {
    foo();
    i++;
}
```