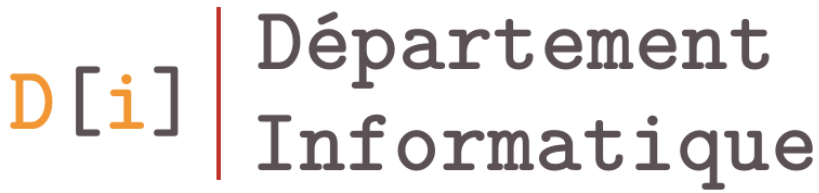
Vérification

Équipe pédagogique du L2 Algorithmique

L2 Informatique, Université de Toulouse



Plusieurs Modèles de Vérification

- **Vérification Dynamique :** Permet de détecter des instances du problème qui ne sont pas satisfaites par le programme. Elle ne permettra pas, en général, de vérifier que le programme satisfait toutes les instances du problème.
  1. Le Test
  2. assert
- **Vérification Statique :** Elle permet de vérifier un programme, indépendamment de toute instance, en fonction de sa spécification :
  1. Pour une spécification en triplet de Hoare :

```
/*PreCondition : P*/
S
/*PostCondition : Q*/
```

  2. la vérification est formelle :

$$P \rightarrow wp("S", Q)$$

Nous nous intéressons aux méthodes de test classique et à l'évaluation des assertion pendant l'exécution du programme :

- **TEST :**
  - Produire les jeux de tests qui permettent de garantir qu'un algorithme satisfait toutes les exigences du problème.
  - Nous ne pouvons ni traiter tous les cas de tests possibles ni restreindre le test à certaines classes de cas qui s'avèrent suffisantes.
  - Il s'agit d'étudier soit les traces d'exécution, en pas à pas, du programme (avec des *printf* par exemple), soit les résultats calculés par le programme.
  - Produire un tableau de situation, si la taille des données le permet et/ou le nombre d'étapes de calcul est raisonnable.

- **assert**
  - Améliorer l'exploitation des tests en s'appuyant sur des assertions qui annotent le programme,
  - Méthode de débogage qui s'appuie sur les pré et post conditions (comme assertions).
  - Nous nous intéressons à une méthode utilisant la fonction 'assert', de la bibliothèque C dans le programme ayant comme argument les assertions qui commentent le programme,
  - Le problème de la génération ou des choix des cas de test reste problématique.
  - Les assertions seront exprimées par des expressions booléennes d'une façon directe en langage C.
  - Les expressions avec quantificateurs nécessitent l'écriture de fonctions.

Exemple

Ecrire, pour un nombre naturel N, la spécification d'un programme qui calcule la plus grande puissance de 2 strictement inférieure à N.

- **Comprendre**
  - Nous dénotons ce nombre par *pgn* (représente la sortie ou le résultat)
  - N est une entrée qui doit être >1, car si $N \leq 1$,alors *pgn* n'existe pas
  - pgn est la puissance de 2 qui est la plus grande $<$ N (à satisfaire par le programme)
  - pgn en résultat, est un entier puissance de 2 et toute puissance de 2 $>$ *pgn* , est supérieure à N
  - $N = 50$ $pgn = 32$
  - $N = 49$ $pgn = 32$

- **spécification formelle**

/* Pré condition :$N \in \mathbb{N}^* \wedge N > 1$ */
PgPuis2Inf(N, &pgn)
/* Post condition : $(\exists I : I \in \mathbb{N} \wedge pgn = 2^I \wedge 2^I < N \leq 2^{I+1})$ */

- **Vérification avec des assert** :
  - Le programme est commenté par des pré et post conditions
  - Les boucles sont annotées avec des invariants.
  - Nous annotons le programme avec des `assert` qui prennent en argument les assertions exprimées en langage C.
  - Pour l'exemple ci-dessous nous proposons un invariant devant être vérifié avant la boucle et à la fin de chaque itération. Pour ce faire
  - Nous utilisons une fonction pour calculer la puissance :

```
int puiss2(int N) {
    if (N==0) return 1;
    return 2*puiss2(N-1);
}
```

- **La post condition en assert :**

```
assert(i>=0 && pgn=puiss2(i) && pgn<N && N<=puiss2(i+1) );
```

- **L'invariant en assert :**

```
assert(i>=0 && pgn=puiss2(i) && pgn<N);
```

Plus Grande Puissance de 2 < N

Exemple

```
int main(){
    int i, pgn, N;
    printf("donner␣un␣entier␣naturel>1␣N=");
    scanf("%d", &N);
    assert(N>1);
    i=0;
    pgn=1;
    assert(i>=0 && pgn==puiss2(i) && pgn<N);
    while(pgn*2 < N){
        ++i;
        pgn = pgn*2;
        assert(i>=0 && pgn==puiss2(i) && pgn<N);
    };
    assert(i>=0 && pgn==puiss2(i) && pgn<N && N<=puiss2(i+1) );
    printf("pgn=%d\n", pgn);
    return 0;
}
```

Remarque

Dans l'exemple précédent nous avons utilisé la fonction assert pour évaluer les assertions. Ce sont des expressions logiques. Le programme s'arrête avec un message dans le cas où leur évaluation donne la valeur FAUX sinon l'exécution continue. Cette méthode de test est bien plus intéressante que l'exécution du programme "pas à pas" car elle évite la construction d'un tableau de situation interminable. Elle permettra de détecter les erreurs concernant la boucle à travers l'évaluation des pré et post conditions.
Pour exprimer les arguments d'assert, il est souvent nécessaire de définir des fonctions booléennes auxiliaires. Le problème de la vérification de ces fontions va se poser aussi.
Cette méthode reste insatisfaisante car elle est dynamique. On dépend de chaque cas de test.

Vérification statique : Triplet de Hoare

A la base nous spécifions un programme par le Triplet de la logique de Hoare :

```
/*PreCondition:  P*/
    S
 /*PostCondition: Q*/
```

- Si la PreCondition P est satisfaite, alors l'action S s'exécute et se termine en un état vérifiant la PostConndition Q.
- Hoare a proposé une méthode de vérification formelle en caractérisant l'affectation et les structures de contrôles par des règles. Ces règles concernent seulement la **correction partielle**.

La vérification selon la méthode de Hoare est une réponse à la question : Est ce que /*P*/S/*Q*/ est vraie ? Ceci pourrait s'alourdir car :

- Pour chaque Q, il y a plusieurs P tels que /*P*/ S /*Q*/ est valide
- Pour chaque P, il y a plusieurs Q tels que /*P*/ S /*Q*/ est valide

Vérification statique : wp de Dijkstra

- Dijkstra a proposé une méthode fondée sur les *wp* (weakest precondition) ou (plus faible précondition : **PFP** = $wp(\text{“}S\text{”}, Q)$), qui conduit à calculer, pour une postcondition Q (objectif) une **précondition unique**.
- Il s'agit de la plus faible pré condition **PFP** à partir de laquelle, si on exécute **S**, alors **S** se termine dans un état qui satisfait **Q**.
- Le *wp* est décrit par des règles de calcul. Le but est de pouvoir, à travers l'application de ces règles, faire disparaître les instructions pour obtenir une formule de la logique du 1er ordre (**PFP**).
- Le problème de la vérification d'un programme devient un problème de vérification d'une formule logique :

$$P \rightarrow PFP \text{ en d'autres termes : } P \rightarrow wp(\text{“}S\text{”}, Q)$$

Pour ce faire, il faut définir des règles qui sont associées aux différentes instructions et structures de contrôle de base (correction partielle) ainsi que les règles concernant la terminaison.

wp d'une Affectation Règle 1

- $wp("x = e", Q) \equiv Q(x \leftarrow e) \wedge calculable(e)$
- on remplace les occurences libres de x dans Q par l'expression e
- l'expression e doit être calculable (diviseur non nul par exemple)

1. Vérifier le triplet : $/*x > 0*//x = x - 1/*x > 0*/$
   - **Vérifier** $(x > 0) \rightarrow wp("x = x - 1", x > 0)$,
   - **Calculer wp :** $wp("x = x - 1", x > 0) \equiv x - 1 > 0$
   - $(x > 0) \rightarrow (x - 1 > 0)$ qui est $\bot$
   - Pour $x = 1$: La prémisse est $\top$ et la conclusion de l'implication est $\bot$.
   - $x = x - 1$ ne satisfait pas la spécification.
2. Vérifier : $/*x > 0*//x = x - 1/*x \geq 0*/$
   - **Vérifier** $(x > 0) \rightarrow wp("x = x - 1", x \geq 0)$.
   - **Calculer wp :** $wp("x = x - 1", x > 0) \equiv x - 1 \geq 0$
   - $(x > 0) \rightarrow (x - 1 \geq 0)$ qui est $\top$
   - $x = x - 1$ satisfait la spécification.

wp d'une Affectation d'un tableau

- La **Règle 1** n'est pas adéquate quand nous l'appliquons sur des variables composées ou indicées comme les cases d'un tableau.
- $wp("a[i] = e;", Q)$ pose problème.
- Vérifions $wp("a[2] = 5", i = 2 \wedge a[i] = 6)$ La **Règle 1** ne peut pas exploiter le fait que a[i] et a[2] peuvent être identiques.
- $wp("a[2] = 5", i = 2 \wedge a[i] = 6)$ devrait nous donner : $(5 = 6)$ ce qu'on peut constater comme faux. Pour conditionner cette règle, nous utilisons l'affectation globale d'un tableau pour modifier une case.

- Nous introduisons la notation $(a; i : e)$ représentant le tableau modifié
- Règle de calcul : $(a; i : e)[j] = a[j]$ si $i \neq j$ sinon e
- Désormais, l'instruction $a[i] = e$ s'écrira $a = (a; i : e)$

- $wp("a[2] = 5", i = 2 \wedge a[i] = 6) \equiv wp("a = (a; 2 : 5)", i = 2 \wedge a[i] = 6)$
- Selon cette écriture la **Règle 1** est applicable en remplaçant dans la post condition a par sa valeur $(a; 2 : 5)$ pour obtenir :
- $wp("a = (a; 2 : 5)", i = 2 \wedge a[i] = 6) \equiv i = 2 \wedge (a; 2 : 5)[i] = 6 \equiv (5 = 6)$ ce qui montre que l'affectation ne satisfait pas la post condition.

wp d'une Séquence Règle 2

- $wp("s_1; s_2", Q) \equiv wp("s_1", wp("s_2", Q))$
- On évalue les *wp* de droite à gauche

- Vérifier la séquence suivante :
  /* $z * x^y = A^B$ */
  `z=z*x;`
  `y=y-1;`
  /* $z * x^y = A^B$ */
  Vérifier : $(z * x^y = A^B) \rightarrow wp("z = z * x; y = y - 1;", z * x^y = A^B)$
  Calcul du wp :
  $wp("z = z * x;", wp("y = y - 1;", z * x^y = A^B)) \equiv$
  $wp("z = z * x;", z * x^{(y-1)} = A^B) \equiv$
  $(z * x * x^{(y-1)} = A^B) \equiv$
  $(z * x^y = A^B)$ qui est égale à la prémisse ci dessus : $\top$

- Vérifier la séquence suivante :

  /* $(A, B \in \mathbb{N} \wedge x = A \wedge y = B)$ */

  ```
  x=x+y;
  y=x-y;
  x=x-y;
  ```

  $/ * x = B \wedge y = A * /$

  Vérifier : $(A, B \in \mathbb{N} \wedge x = A \wedge y = B) \rightarrow$
  $wp("x = x + y; y = x - y; x = x - y;", x = B \wedge y = A)$

  Calcul du wp :

  $wp("x = x + y; y = x - y; x = x - y;", x = B \wedge y = A) \equiv$
  $wp("x = x + y", wp("y = x - y", wp("x = x - y", x = B \wedge y = A))) \equiv$
  $wp("x = x + y", wp("y = x - y", x - y = B \wedge y = A)) \equiv$
  $wp("x = x + y", x - (x - y) = B \wedge x - y = A) \equiv$
  $x + y - (x + y - y) = B \wedge x + y - y = A \equiv$
  $y = B \wedge x = A$

  La vérification se ramène à la vérification de la formule :
  $(A, B \in \mathbb{N} \wedge x = A \wedge y = B) \rightarrow (y = B \wedge x = A)$
  $\top$

wp de l'instruction if Règle 3

- $wp("if(b)\ s_1; else\ s_2;", Q) \equiv (b \rightarrow wp("s_1", Q) \wedge \neg b \rightarrow wp("s_2", Q))$
- $wp("if(b)\ s;", Q) \equiv (b \rightarrow wp("s", Q) \wedge \neg b \rightarrow Q)$

La vérification d'une instruction if :

```
/*P*/
if (b)
     S1;
else
     S2;
/* Q*/
```

$P \rightarrow wp("if(b)\ S1; else\ S2;", Q)$

$\equiv$

$P \rightarrow (b \rightarrow wp("S1", Q) \wedge \neg b \rightarrow wp("S2", Q))$

$\equiv$

$(P \wedge b \rightarrow wp("S1", Q)) \wedge (P \wedge \neg b \rightarrow wp("S2", Q))$

- **valeur absolue**

$/*x = A \wedge A \in \mathbb{Z}\ */$

```
if (x<0)
    x=-x;
```

$/*\ x = |A|\ */$

  - **Vérifier :** $x = A \wedge A \in \mathbb{Z} \rightarrow wp(\text{“}if\,(x < 0)x = -x;\text{”}, x = |A|)$
  - **Calcul du wp :** $wp(\text{”}if\ (x < 0)\ x = -x;\text{”}, x = |A|)$
    - ★ $(x < 0 \rightarrow wp(\text{”}x = -x\text{”}, x = |A|)) \wedge (x \geq 0 \rightarrow x = |A|)$
    - ★ $(x < 0 \rightarrow -x = |A|) \wedge (x \geq 0 \rightarrow x = |A|)$
  - Le triplet est donc équivalent à
    $x = A \wedge A \in \mathbb{Z} \rightarrow (x < 0 \rightarrow -x = |A|) \wedge (x \geq 0 \rightarrow x = |A|)$
    $\equiv$
    $x = A \wedge A \in \mathbb{Z} \rightarrow (A < 0 \rightarrow -A = |A|) \wedge (A \geq 0 \rightarrow A = |A|)$
    $\equiv \top$ par définition de la valeur absolue.

- **Calculer dans (x,y) le min et le max des entiers A et B**

```
/*A, B ∈ ℤ*/
 if (A<B){
     x=A;
     y=B;
 }
 else {
     x=B;
     y=A;
 }
/* x ≤ y ∧ (x, y) = permutation(A, B) */
```

- **Vérifier :** $(A, B \in \mathbb{Z}) \rightarrow (wp(\text{"}if\ (A < B)\ \{x = A;\ y = B;\}\ else\ \{x = B;\ y = A;\}\text{"}, x \leq y \wedge (x, y) = permutation(A, B))$
- **Calcul du wp :**
  - $(A < B \rightarrow wp(\text{"}x = A;\ y = B;\text{"}, x \leq y \wedge (x, y) = permutation(A, B)))$
    $\wedge (A \geq B \rightarrow wp(\text{"}x = B;\ y = A\text{"}, x \leq y \wedge (x, y) = permutation(A, B)))$
  - $(A < B \rightarrow A \leq B \wedge (A, B) = permutation(A, B))$ **(a)**
    $\wedge\ (A \geq B \rightarrow B \leq A \wedge (B, A) = permutation(A, B))$ **(b)**

  $\equiv$

- $(A, B \in \mathbb{Z}) \rightarrow (\mathbf{a}) \wedge (\mathbf{b}) \equiv \top$.

Les Boucles while

```
/*P*/
init;
while (b)
    S;
/*Q*/
```

Pour simplifier l'expression du *wp* d'une boucle nous utilisons une approche fondée sur les notions d'**invariant** et de **variant** qui séparent la vérification en deux parties (correction partielle et terminaison). Le chapitre suivant abordera différentes stratégies pour construire l'invariant et le variant.

- **Correction Partielle** si la boucle se termine alors elle se termine correctement. L'invariant de la boucle **I**, qui est utilisé dans ce cadre, exprime, intuitivement, la propriété intermédiaire à satisfaire par chaque itération. Ceci nous a conduit à proposer trois règles simples pour vérifier la correction partielle :

- **W1** $P \rightarrow wp(\text{"}init\text{"}, I)$ ce qui veut dire I est $\top$ avant la boucle
- **W2** $I \wedge b \rightarrow wp(\text{"}S\text{"}, I)$ I est conservé par chaque itération S
- **W3** $I \wedge \neg b \rightarrow Q$ A la sortie de la boucle son objectif est atteint

- **La boucle annotée par l'invariant (correction partielle) :**

```
/*P*/

init;

/*I*/

while (b)  {

   /*I and b*/

       S ;

   /*I*/
   }

/*I and not b*/

/*Q*/
```

Vérification Plus Grande Puissance de 2 < N

- **Programme annoté**

$/*\ N \in \mathbb{N}^* \wedge N > 1\ */$ **(w1)**

```
i=0;
pgn=1;
```

$/*\ (i \in \mathbb{N} \wedge pgn = 2^i \wedge pgn < N\ */$

```
while (pgn*2 < N) {
```

$/*\ (i \in \mathbb{N} \wedge pgn = 2^i \wedge pgn < N \wedge pgn * 2 < N\ */$ **(w2)**

```
    ++i;
    pgn = pgn*2;
```

$/*\ (i \in \mathbb{N} \wedge pgn = 2^i \wedge pgn < N\ */$

```
};
```

$/*\ (i \in \mathbb{N} \wedge pgn = 2^i \wedge pgn < N \wedge pgn * 2 \geq N\ */$ **(w3)**

$/*\ (\exists I \in \mathbb{N} \wedge pgn = 2^I \wedge pgn < N \leq 2^{I+1})\ */$

Correction partielle :

- **Vérifier W1 :**
  $N \in \mathbb{N}^* \wedge N > 1 \rightarrow wp(" i = 0;\ pgn = 1;", i \in \mathbb{N} \wedge pgn = 2^i \wedge pgn < N)$
  - **Calcul de wp :** $wp(" i = 0;\ pgn = 1;", i \in \mathbb{N} \wedge pgn = 2^i \wedge pgn < N)$
  - $(0 \in \mathbb{N} \wedge 1 = 2^0 \wedge 1 < N)$

  $\equiv$
- $N \in \mathbb{N}^* \wedge N > 1 \rightarrow (0 \in \mathbb{N} \wedge 1 = 2^0 \wedge 1 < N)$
  - les deux premiers conjoints, $(0 \in \mathbb{N} \wedge 1 = 2^0)$ sont $\top$ par l'arithmétique
  - le troisième conjoint, $1 < N$, est $\top$ par la prémisse,
  - **w1** $\equiv \top$

Correction partielle :

- **Vérifier W2 :**
  $(i \in \mathbb{N} \wedge pgn = 2^i \wedge pgn < N \wedge pgn * 2 < N \rightarrow$
  $wp(" ++i;\ pgn = pgn * 2", i \in \mathbb{N} \wedge pgn = 2^i \wedge pgn < N)$
  - **Calcul de wp :** $((i+1) \in \mathbb{N} \wedge pgn * 2 = 2^{i+1} \wedge pgn * 2 < N)$
  - $((i+1) \in \mathbb{N} \wedge pgn * 2 = 2^{i+1} \wedge pgn * 2 < N)$

  $\equiv$
- $(i \in \mathbb{N} \wedge pgn = 2^i \wedge pgn < N \wedge pgn * 2 < N \rightarrow$
  $((i+1) \in \mathbb{N} \wedge pgn * 2 = 2^{i+1} \wedge pgn * 2 < N)$
  - $(i+1) \in \mathbb{N} \equiv \top$ car dans la prémisse on a $i \in \mathbb{N}$ le successeur de i aussi $\in \mathbb{N}$
  - $pgn * 2 = 2^{i+1} \equiv \top$ car par la prémisse on a $pgn = 2^i$ en multipliant par 2 les deux membres on conserve l'équation
  - $pgn * 2 < N$ par la prémisse
- **w2**$\equiv \top$
- **Vérifier W3 : Il suffit de prendre I=i**
  $(i \in \mathbb{N} \wedge pgn = 2^i \wedge pgn < N \wedge pgn * 2 \geq N) \rightarrow$
  $(\exists I \in \mathbb{N} \wedge pgn = 2^I \wedge pgn < N \leq 2^{I+1}) \equiv \top$ car $pgn * 2 = 2^I * 2$

Terminaison

- Nous cherchons à construire un ensemble muni d'un ordre bien fondé : $(E, \prec)$, ce qui signifie que E ne contient pas de chaînes infinies strictement décroissantes.
- Ceci permet d'exprimer que toute progression de la boucle correspond à une chaîne qui décroît selon l'ordre bien fondé, et donc que la boucle termine. Nous utilisons la notion de *variant* :
  - Souvent on se trouve dans les entiers naturels où toute chaîne décroissante est finie : le *variant* est une expression entière > 0.
  - Il suffit de montrer que ce variant décroit à chaque itération de la boucle. Nous utilisons une variable auxiliaire t pour sauvegarder la valeur du variant au début du corps de la boucle. La démarche s'exprime en deux règles :

- **W4** $(I \wedge b) \rightarrow variant \in E$
  Ceci signifie qu'au début d'une itération le variant n'a pas atteint 0 dans le cas des entiers.
- **W5** $(I \wedge b \wedge t = variant) \rightarrow wp(''S\text{``}, variant \prec t)$
  A chaque fin d'itération le *variant* décroit strictement.

(correction totale = correction partielle + terminaison) :

```
/*P*/
init;
/*I  */
while (b) {
    /*I and b  */    (w4)
    /* 0 < variant*/
    t=variant
    /*I and b and  t= variant*/    (w5)
    S ;
    /*I and variant<t*/
    }
/*I and not b*/
/*Q*/
```

- Considérons l'exemple précédent **après sa correction partielle**.
  - proposer un *variant* peut être compliqué voire sans réponse connue même pour des problèmes simples.
  - Il est construit à partir des variables manipulées dans la boucle.
  - Dans notre cas, nous sommes dans $\mathbb{N}$ le *variant* est une expression entière bornée par 0 qui décroit à chaque itération.
- Comme *variant* nous proposons l'expression $N - pgn * 2$.
- Le schéma du programme complété par les assertions de la correction totale :

/* $N \in \mathbb{N} \ \wedge \ N > 1$ */
i=0 ; pgn=1 ;
/* $(i \in \mathbb{N} \wedge pgn = 2^i \wedge pgn < N$ */
while (pgn*2 < N) {
/* $(i \in \mathbb{N} \wedge pgn = 2^i \wedge pgn < N \wedge pgn * 2 < N) \wedge N - pgn * 2 > 0$ */
t = N-pgn*2 ; ++i ; pgn = pgn*2 ;
/* $(i \in \mathbb{N} \wedge pgn = 2^i \wedge pgn < N) \wedge \ t > \underline{N - pgn * 2}$ */
} ;
/* $(i \in \mathbb{N} \wedge pgn = 2^i \wedge pgn < N \wedge pgn * 2 \geq N$ */
/* $(\exists I \in \mathbb{N} \wedge pgn = 2^I \wedge pgn < N \leq 2^{I+1})$ */

Vérifier la terminaison de la boucle

- **Vérifier W4 :**
  $(i \in \mathbb{N} \wedge pgn = 2^i \wedge pgn < N \wedge pgn * 2 < N \rightarrow N - pgn * 2 > 0$
  $\equiv \top$
  car $N - pgn * 2 > 0$ est obtenue directement à partir de la prémisse $pgn * 2 < N$
- **Vérifier W5**
  $(i \in \mathbb{N} \wedge pgn = 2^i \wedge pgn < N \wedge pgn * 2 < N \wedge t = N - pgn * 2) \rightarrow$
  $wp(" ++i;\ pgn = pgn * 2", t > N - pgn * 2)$
  - **Calcul du wp :** $t > N - pgn * 2 * 2)$

  $\equiv$
- $(i \in \mathbb{N} \wedge pgn = 2^i \wedge pgn < N \wedge pgn * 2 < N \wedge t = N - pgn * 2 \rightarrow$
  $t = N - pgn * 2 > N - pgn * 2 * 2)$ car $pgn = 2^i \geq 1$
- **w5** $\equiv \top$

Exercice 1

Il s'agit d'écrire un programme qui calcule la position la plus proche d'un entier X, dans un tableau trié T[N]. X se trouve dans l'intervalle entre le premier élément du tableau et le dernier élément de ce tableau : [T[0] , T[N-1] [. Par position la plus proche nous entendons l'indice pos qui vérifie : $T[pos] \leq X < T[pos+1]$.

Comprendre

- Nous constatons que N doit être > 1, car X appartient à un intervalle non vide avec T[0] < T[N-1]
- le tableau doit être trié (T,$\leq$), avec au moins $T[0] < T[N-1]$. La position la plus proche ne peut pas être $N-1$
- Soit N=8, $T = \{0, 2, 2, 3, 5, 5, 5, 6\}$.
  On peut proposer pour X l'une des valeurs 0, 1, 2, 3 4, 5. Si X=5, pos est la dernière position de 5 dans T càd 6.
  Nous constatons qu'il peut y avoir une infinité de cas de tests avec des tableaux de taille > 1. Il est donc nécessaire d'exprimer les pré et post conditions d'une façon formelle ce qui couvre tous les cas de test.

Spécification et Modèles de solution

- **Spécification :**
  $/*N > 1 \wedge (\forall I : 0 \leq I \leq N-2 \rightarrow T[I] \leq T[I+1]) \wedge T[0] \leq X < T[N-1]*/$
  PosProche (T, X, &pos)
  $/*0 \leq pos \leq N-2 \wedge T[pos] \leq X < T[pos+1]*/$
- **Modèles de Solution** Nous pouvons proposer trois modèles de recherche de la position la plus proche de X dans T :
  1. Recherche linéaire à partir de la position 0
  2. Recherche linéaire à partir de N-2
  3. Recherche dichotomique
- Le troisième modèle est le plus rapide. Le modèle 2 permettra de s'arrêter dès qu'on trouve un nombre $\leq$ X. Il paraît plus avantageux que la solution 1 qui conduit, dans le cas où X est égal à une case du tableau à effectuer plusieurs itérations inutiles, comme par exemple avec X=5. Nous avons choisi, pour cet exemple, le modèle 2.

Vérification

$/ * N > 1$
$\wedge(\forall I : 0 \leq I \leq N-2 \rightarrow T[I] \leq T[I+1]) \wedge T[0] \leq X < T[N-1] * /$
pos=N-2 ;
/*.....*/
while (T[pos] > X){
/*.....*/
- -pos ;
/*.....*/
}
/*.....*/
$/ * 0 \leq pos \leq N-2 \wedge T[pos] \leq X < T[pos+1] * /$

- Invariant :$0 \leq pos \leq N-2 \wedge X < T[pos+1] \wedge$ précondition du programme
- **Q1** Compléter les assertions dans le programme ci-dessus
- **Q2** Vérifier la correction partielle de la boucle
- **Q3** Proposer un *variant* et vérifier la terminaison

UNIVERSITE TOULOUSE III PAUL SABATIER
Faculté des Sciences et d'Ingénierie

Exercice 2

Considérons le programme suivant :

/* $N \in \mathbb{N}$ */

```
     i=0 ;  c=0 ;  k=1 ;  m=6 ;
```

/* $(c = i^3) \wedge (0 \leq i \leq N) \wedge (k = 3 * i^2 + 3 * i + 1) \wedge (m = 6 * i + 6)$*/

```
while (i<=N) {
     /* ..................... */
     t=variant;
     c = c+ k ; k= k + m ; m = m+6 ;
     i++ ;
     /* ..................... */
     };
  /*  ..................... */
}
```

/*$c = N^3$ */

- **Q1.** Compléter les commentaires en .....
- **Q2.** Vérifier (correction partielle) que le programme calcule dans la variable c la valeur de N^3. Si erreur, proposer une modification du programme et vérifier.
- **Q3.** Correction totale : proposer un *variant* et vérifier la terminaison

Exercice 3 : "Mini Syracuse"

Nous souhaitons vérifier un programme qui calcule les termes d'une suite qu'on nomme **Mini Syracuse** qui commence par une valeur $U_0 = N \in \mathbb{N}^*$:

$$U_i = \begin{cases} N & \text{si } i = 0 \\ U_{i-1}/2 & \text{si } i > 0 \wedge U_{i-1} \text{ est paire} \\ U_{i-1} + 1 & \text{si } i > 0 \wedge U_{i-1} \text{ est impair} \end{cases}$$

On peut considérer le programme suivant :

```
/*N ∈ ℕ**/
y=N;
/* ⊤ */
 while (y>1)
    if (y%2==0)
        y=y/2;
    else
        y=y+1;
/*y = 1*/
```

- Invariant Triviale : $\top$
- **Q1** Vérifier la correction partielle
- **Q2.** Proposer un *variant* et vérifier la terminaison