IV. Conception de programmes (2/2)

Les algorithmes des fonctions des types

IV.1 Méthode de conception des algorithmes

Pour étudier les grands principes de conception des algorithmes, nous nous appuierons sur la résolution d'un problème classique : *le tri*.

Définition : *Tri d'entier*

// Données : une séquence de n entiers $(a_1, a_2, \ldots, a_n)$

// Résultat : une permutation $(b_1, b_2, \ldots, b_n)$ de la séquence $(a_1, a_2, \ldots, a_n)$ telle que $b_1 \leqslant b_2 \leqslant \ldots \leqslant b_n$

La définition d'un problème reste général et un algorithme doit s'attacher à résoudre ce problème quelques soient les données correspondants aux spécifications du problèmes. On parle d'*instance* du problème.

Définition : *instance d'un problème*

Une *instance* d'un problème est constitué d'un ensemble de données satisfaisant les contraintes imposées dans l'énoncé du problème.

Définition : *algorithme correct*

On dit qu'un algorithme est *correct* si pour toutes les instances d'un problème il se termine et fournit une solution correcte. On dit alors que l'algorithme *résout* le problème

IV.1.1 Approche incrémentale

L'approche *incrémentale* consiste à prendre les données une à une et à les insérer dans la solution en cours de construction.

Une fois que toutes les données auront donc été traitées une à une, on obtient la solution finale.

C'est une approche assez naturelle et qui donnera la plupart du temps un *algorithme itératif*.

On appelle aussi ce type d'algorithme, *algorithme glouton*.

Tri par insertion

principe : insérer les éléments un à un et augmenter ainsi à chaque étape la taille de la séquence triée

```
func tri_insertion(S : [Int](n)) -> [Int](n)
  #donnée   : une séquence S de n nombres entiers
  #résultat : la séquence triée par ordre croissant
  for j in 1..<n do // indice du nombre à trier -> à
  insérer
    c=S[j]   // extraction du nombre en cours d'insertion
    i=j-1    // examen de la séquence en partant du dernier
    while(i≥0) and (S[i]>c) do // tant que les valeurs sont
    ≥
      S[i+1]=S[i] // on décale la séquence d'un rang
      i=i-1       // on passe au nombre précédant
    endw
    S[i+1]=c // on insère c à la dernière position examinée
```

IV.1.2 Diviser pour mieux régner

principe : diviser le problème initial en sous-problèmes similaires afin de devoir résoudre un problème de taille moindre. Ensuite, il suffit de recombiner les solutions obtenues sur les sous-problèmes. Cette fusion des résultats est plus simple que de calculer le résultat directement.

Si la taille du sous-problème est encore trop importante, on peut appliquer à nouveau la méthode en diviser les sous-problèmes en sous-sous-problèmes de taille moindre. Cela revient à appliquer à nouveau la stratégie diviser pour mieux régner sur les sous-problèmes $\longrightarrow$ l'algorithme s'utilise donc lui-même sur les sous-problèmes définis $\Longrightarrow$ *algorithme récursif*

Par exemple pour le tri, l'algorithme devient :

1. ***diviser :*** diviser la séquence `S` de n nombres en deux sous-séquences de $n/2$ nombres ;
2. ***régner :*** trier chaque sous-séquence
   - 2.1. si trier la sous-séquence est encore trop difficile, alors appliquer à nouveau la méthode : diviser-régner-combiner
     $\longrightarrow$ appel récursif à l'algorithme de tri;
   - 2.2. si la sous-séquence est facile à trier (sous-séquence de taille 1 ou 2), alors la trier
3. ***combiner :*** fusionner les deux sous-séquences triées pour produire la séquence triée.

Tri par fusion

```
Proc tri-fusion(S : [Int](n), p : Int, r : Int)
  //Pre:   S suite de n nombre entiers
           0<=p<=r<=n
  //Données:  S, séquence de nombres de taille n
            p l'indice à partir duquel le tri se fait
            r l'indice de la 1ère donnée à ne pas trier
  if (p<r) then
    q=(p + r)/2              //diviser
    tri-fusion(S,p,q+1) //régner : appel récursif sur la 1ère
moitié
    tri-fusion(S,q+1,r) //régner : appel récursif sur la 2nde
moitié
    fusionner(S,p,q,r)  //combiner : fusionner les deux listes
triées
  endif
endProc
```

IV.2 Analyser l'efficacité d'un algorithme

Concevoir un algorithme nécessite d'avoir une stratégie mais aussi de s'assurer que l'algorithme est

- correcte : il faudra donc être capable de montrer qu'il est juste et donc prouver sa justesse
- efficace : mesurer son efficacité pour éviter de concevoir des algorithmes qui résolvent le problème après plusieurs heures/jours/semaines d'exécution.

Commençons par analyser l'efficacité de nos deux algorithmes de tri pour les comparer.

IV.2.1 Analyse de l'efficacité du tri par insertion

Le temps pris par l'algorithme dépend du nombre de données dans la séquence `s`

En général le temps d'exécution d'un algorithme croît avec la taille des données en entrée à examiner.

L'efficacité d'un algorithme s'exprime donc en fonction de la ***taille d'entrée*** de l'algorithme

Pour chaque problème, il faut donc commencer par spécifier ce que représente cette taille d'entrée :

- nombre de données à examiner
- nombre de bits du codage de la donnée,
- nombre de chaque type de données (*n* arêtes et *m* sommets)

Pour analyser l'efficacité d'un algorithme, on considère que chaque instruction est une ***opération élémentaire*** et on examine combien de fois elle s'exécute : on compte ***le nombre total d'opérations élémentaires***

```
for j in 1…S.count do // n
  c=S[j]    // n-1
  i=j-1       // n-1
  while(i≥0) and (S[i]>c) do  // t_j pour chaque j variant de
  1 à n
  // soit au total ∑_{j=1}^{n-1} t_j
    S[i+1]=S[i] // ∑_{j=1}^{n-1} (t_j − 1) (même raisonnement)
    i=i-1         // ∑_{j=1}^{n-1} (t_j − 1) (même raisonnement)
  ends
  S[i+1]=c // n-1
endfor
```

Le temps d'exécution en fonction de n est alors :

$$T(n) = n + (n-1) + (n-1) + \sum_{j=1}^{n-1} t_j + \sum_{j=1}^{n-1} (t_j - 1) + \sum_{j=1}^{n-1} (t_j - 1) + (n-1)$$

Le temps d'exécution dépendra donc essentiellement des valeurs t_j, elles-mêmes dépendant de l'ordre des données de la séquence d'entrée `s`.

On dit que le temps d'exécution dépend de la nature de l'entrée du problème.

Plusieurs hypothèses :

- `s` est déjà triée,
- `s` est triée en ordre inverse,
- `s` est rangée dans un ordre quelconque.

`S` est déjà trié

Alors `S[j]<=c` pour chaque `j=1,…,len(S)` et t_j`=1` pour toutes les valeurs de `j`. Soit

$$T(n) = n + 4(n-1) + n = 6n - 4$$

Le cas où `S[]` est déjà trié est le meilleur cas pour cet algorithme et `T(n)` est une fonction ***linéaire*** de n.

On dit que l'algorithme est *linéaire en la taille de la donnée*

`S[ ]` est trié en ordre inverse

On doit à chaque fois comparer chaque élément de la séquence déjà triée avec c et donc `t_j=j.` Sachant que

$$\sum_{j=1}^{n-1} j = \frac{n(n-1)}{2} \text{ et } \sum_{j=1}^{n-1} (j-1) = \frac{(n-1)(n-2)}{2}$$

alors

$$T(n) = n + (n-1) + (n-1) + \frac{n(n-1)}{2} + \frac{(n-1)(n-1)}{2} + \frac{(n-1)(n-2)}{2} + (n-1)$$

Soit

$$T(n) = \frac{1}{2}(3n^2 + n + 2)$$

C'est le pire cas pour cet algorithme.

On dira que la *complexité de l'algorithme* dans le ***pire cas*** est *quadratique* en la taille de la donnée.

IV.2.2 Analyse de l'efficacité du tri par fusion

On supposera que la taille du problème initial est une puissance de 2.

- Chaque étape de diviser produit alors deux sous-séquences d'une taille `n/2`.
- L'étape diviser consiste à calculer l'indice du milieu de la séquence : temps en `θ(1)`.
- L'étape régner consiste à résoudre récursivement les deux sous-problèmes. Soit `T(n)` la fonction exprimant le temps de l'algorithme tri-fusion, l'étape régner nécessite `2×T(n/2)`
- L'étape combiner, c'est à dire l'exécution de la fonctions `fusionner`, est clairement en `θ(n)`.

Finalement le temps d'exécution est donné par la fonction de récurrence :

$$T(n) = \begin{cases} \Theta(1) \text{ si } n = 1 \\ 2 * T\left(\frac{n}{2}\right) + \Theta(1) \text{ si } n > 1 \end{cases}$$

$$T(n) = \Theta(n \log_2 n) + \Theta(n) = \Theta(n log_2 n)$$

Conclusion : le tri par fusion est plus efficace que le tri par insertion dès que *n* est assez grand.

Conclusion

L'analyse d'un algorithme peut donc dépendre des instances du problème. En général, on se place toujours dans le pire des cas. Les arguments pour le choix de cette méthodes sont :

- le temps d'exécution dans le pire des cas constitue une borne supérieure du temps d'exécution ;
- pour certains algorithmes, le pire des cas arrive assez souvent;
- assez souvent le cas moyen est à peu près aussi mauvais que le pire des cas .

Lorsqu'on analyse des algorithmes, on s'intéresse essentiellement à l'***ordre de grandeur*** du temps d'exécution.

IV.3 Prouver un algorithme

Tester un programme est *insuffisant* pour prouver qu'il est juste.

Le test d'un programme ne peut ***démontrer qu'une chose*** : qu'il est faux!

Pour prouver un algorithme, deux possibilités :

- faire une preuve « *manuelle* » sur le modèle des démonstrations en mathématiques, en utilisant des assertions bien définies et bien placées ;
- faire une preuve « *formelle* » avec l'utilisation de règles de déduction précises et l'aide d'un programme d'assistance aux preuves.

IV.3.1 Principe d'une preuve

Il faut apprendre à raisonner sur le texte d'un algorithme afin de gagner du temps sur les tests !

spécifier correctement et ***suffisamment*** l'algorithme

mettre des ***commentaires pertinents*** qui feront le lien entre la preuve et la conception de l'algorithme ➔ amélioration de la *lisibilité* de l'algorithme

Il est souvent possible de prouver l'algorithme et de s'assurer :

- de sa *correction* ;
- de sa *robustesse* ;

La justesse n’est pas la seule qualité attendue d’un algorithme. Un bon algorithme a les qualités suivantes :

Justesse : obtient le(s) résultat(s) recherché(s) en un temps fini ;

Correction : des spécifications claires permettant de s'assurer du bon emploi de l'algorithme et de s’assurer de sa *justesse* ;

Lisibilité : qui facilite la compréhension, la maintenance et l'évolution du programme correspondant ;

Efficacité : qui limite au maximum l'utilisation des ressources mémoires et processeurs.

Robustesse : son exécution résiste à tous les cas d’utilisation, ce qui nécessite la prévision des cas de mauvaise utilisation ;

Définition : *Spécification*

La *spécification* précise les *données* avec les *pre-conditions* que l'on exige d'elles, ainsi que les *résultats* avec les *post-conditions* que l'algorithme doit assurer.

Une spécification définit implicitement un ***contrat*** :

> *si les pre-conditions sont vérifiées avant l'appel, alors les post-conditions sont vérifiées au retour de l'appel du sous-programme*

En conséquence

- ***l'appelant*** doit satisfaire les pre-conditions mais peut alors supposer que les post-conditions sont vérifiées.
- ***l'appelé*** peut supposer les pre-conditions vraies mais doit s'assurer de la justesse des post-conditions

Pour prouver un algorithme, il suffit alors de montrer que le contrat est respecté et pour cela montrer que la suite d'opérations permet d'obtenir le bon résultat.

Idée générale : un algorithme est une séquence d'états de la mémoire permettant de passer d'un état initial à un état final :

$\text{État}_{\text{initial}} \longrightarrow$ ***Algorithme*** $\longrightarrow \text{État}_{\text{final}}$

La spécification d'un programme sera décrit par :

- des propriétés vérifiées par $\text{État}_{\text{initial}}$
- des propriétés vérifiées par $\text{État}_{\text{final}}$
- des relations entre données et résultats
- la descriptions des états du programme par des ***assertions***

Définition : *Assertion*

Une *assertion* est une ***expression*** de type ***Booléen*** permettant de vérifier certaines conditions de validité de l'algorithme.

Définition : *Assertion d'un algorithme*

Une assertion est un énoncé

- placé en un point du texte du programme
- exprimant des propriétés et/ou des relations entre les valeurs des variables du programme ***vérifiées*** chaque fois que l'exécution du programme atteint ce point.

IV.3.2 Preuve de correction partielle

Prouver un programme se réalise alors en deux étapes :

1. faire une *preuve de correction partielle*, en s'appuyant sur les spécifications et des assertions exprimant les états de l'algorithme
2. et faire une *preuve de terminaison*

Définition : *Preuve de correction partielle*

Une preuve de correction partielle consiste à

- caractériser l'effet de chaque instruction ;
- combiner ces effets (instructions composées) ;
- vérifier par des assertions l'évolution de l'état du programme.

IV.3.2.1 Exemple de l'algorithme de division euclidienne


```
func divEuclide(X,Y:Int) -> (Int, Int)
// calcule le quotient Q et le reste R de X/Y
// Pre : ?
// Post: return X/Y
  Q : Int = 0
  R : Int = X
  while R > Y do
    R = R - Y
    Q = Q + 1
  endw
  return (Q,R)
endfunc
```

Comment vérifier que cet algorithme de division euclidienne est juste ?

On peut faire quelques tests pour vérifier si l'algorithme n'est pas faux :

- `X=13` et `Y=5` : `Q=2` et `R=3`, pas d'erreur détectée
- `X =13` et `Y=-5` : l'algorithme boucle
- `X=-13` et `Y=5` : `Q=0` et `R=-12`, l'algorithme est faux
- `X=-13` et `Y=-5` : `Q=2` et `R=-3`, l'algorithme est faux
- `X=13` et `Y=0` : l'algorithme boucle

De ces tests, on peut facilement déduire que X et Y doivent être strictement positifs.

Peut-on maintenant garantir que le résultat est juste?

Il faut exprimer par une *assertion* le fait que la fonction calcule le quotient `Q` et le reste `R` de la division euclidienne de `X` par `Y`, c'est à dire que $(X = Q \times Y + R)$ et $(R < Y)$ à la fin de l'algorithme

```
func divEuclide(X,Y:Int) -> (Int, Int)
// calcule le quotient Q et le reste R de X/Y
// Pre: X,Y >0
// Post:  return X/Y
  Q : Int = 0
  R : Int = X
  //
  while R≥Y do
    R = R - Y //
    Q = Q + 1 //
    //
  endw
  // Assertion: (X = Y × Q + R) and (R < Y)
  return (Q,R)
endfunc
```

Il suffit de prouver l'assertion finale pour prouver que l'algorithme est juste

Cette assertion $(X = Y \times Q + R)$ et $(R < Y)$ qui exprime le résultat va devenir notre *invariant* et va nous aider à prouver notre algorithme. Il ne fait qu'exprimer le résultat en cours de calcul : $X = Q \times Y + R$

```
func divEuclide(X,Y:Int) -> (Int, Int)
// calcule le quotient Q et le reste R de X/Y
// Pre: X,Y >0
// Post:return X/Y
  Q : Int = 0
  R : Int = X
  // (X = Y × Q + R) : propriété (Invariant) vraie au départ
  while R≥Y do
    R = R - Y // on perturbe l'invariant (on fait avancer l'algorithme)
    Q = Q + 1 // on doit rétablir l'invariant
    // (X = Y × Q + R) : invariant toujours vrai ?
  endw
  // invariant (X = Y × Q + R) + condition de fin (R < Y)
  // donne l'assertion finale : (X = Y × Q + R) et (R < Y)
  return (Q,R)
endfunc
```

CQFD.

L'assertion $(X = Y \times Q + R)$ de `divEuclide` est un invariant de boucle. Son expression est `True` quelque soit l'itération.

Un invariant de boucle exprime la propriété suivante :

```
// P                          // P
Instruction I      ⟹          While C:
// P                          Instruction I
                             // P and !C
```

Il faut donc arriver à prouver qu'il reste vrai malgré les instructions *I* qui modifient l'état de l'algorithme.

le problème de l'affectation c'est le changement d'état de la variable :

```
// P(X,…)
X = Expression ;
// P'(X,…) ???
```

Exemple : Affectation et assertions (1/2)

```
// X = Y × Q + R, R ⩽ Y; pre-assertion
R = R-Y; // affectation ⟹ changement d'état
// ??? post-assertion
```

Exemple : Affectation et assertions (2/2)

```
// X = Y × (Q + 1) + R, R ⩾ 0; pre-assertion
Q = Q+1; // affectation ⟹ changement d'état
// ??? post-assertion
```

On peut utiliser des indices k pour faciliter les preuves :

- avant la boucle : $Q_O = 0,\ R_0 = X,\ X = Y \times Q_O + R_O$
- à la k[ième] itération, on écrira :

```
// Assertion : X = Y × Q_{k-1} + R_{k-1}
R=R-Y // ⟹ R_K = R_{k-1} − Y ⟶ on perturbe
Q=Q+1 // ⟹ Q_K = Q_{k-1} + 1 ⟶ on essaie de rétablir
// a-t-on toujours l'invariant ?
// R_K = R_{k-1} − Y ⟹ R_{k-1} = R_k + Y
// Q_K = Q_{k-1} + 1 ⟹ Q_{k-1} = Q_k − 1
// X = Y × Q_{k-1} + R_{k-1} ⟺ X = Y × (Q_k − 1) + (R_k + Y)
// ⟺ X = Y × Q_k − Y + R_k + Y ⟺ X = Y × Q_k + R_k ⟶ l'invariant est
rétabli
```

On a donc bien rétabli l'invariant, la propriété reste donc vraie et par induction, elle est vraie à la fin de l'algorithme → CQFD

IV.3.2.2 Comment écrire une itération juste et la prouver ?

En pratique, lorsqu'on écrit une itération et que l'on veut prouver sa justesse, il faut :

- avoir une idée !
- se placer à l'itération k et déterminer l'invariant
- écrire sous forme de commentaire, et si possible sous forme d'assertion, l'invariant de boucle
- en déduire :
  - la condition d'arrêt
  - les initialisations
  - les actions à effectuer après la boucle si besoin est...
- prendre l'habitude d'écrire sous forme de commentaire en toute lettre la condition d'arrêt

IV.3.3 Preuve de terminaison

Définition : *Preuve de terminaison*

Une preuve de terminaison consiste à montrer que l'exécution de l'algorithme s'arrête.

Dans le cas d'une ***séquence d'instruction sans rupture de séquence***, prouver que l'algorithme se termine est trivial.

Dans le cas d'un *algorithme itératif,* il est souvent facile de montrer qu'un algorithme se termine.

Dans le cas d'un ***algorithme récursif*** il faut montrer qu'il s'arrête et donc que la suite des appels permettra bien d'atteindre la condition d'arrêt

$\longrightarrow$ démonstration par récurrence sur les variables faisant partie de la condition d'arrêt.

Preuve de terminaison des boucles

Il n'est pas toujours évident de prouver qu'une boucle se termine.

Principe de base

➔ associer à la boucle une *suite strictement décroissante finie*.

Au minimum

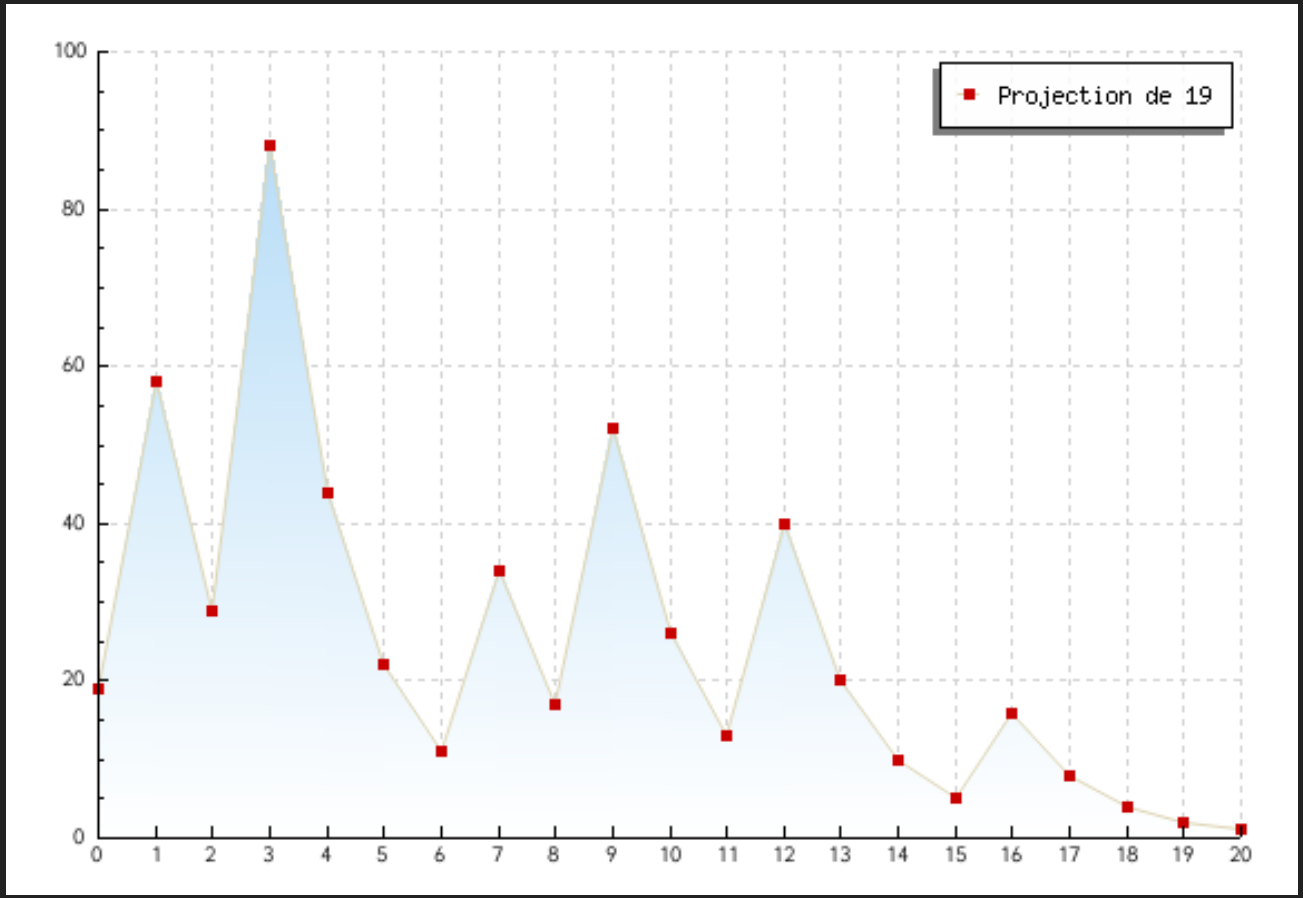
➔ vérifier que la condition d'arrêt est modifiée dans le corps de la boucle

Mais il existe des cas où ce n'est pas facile....

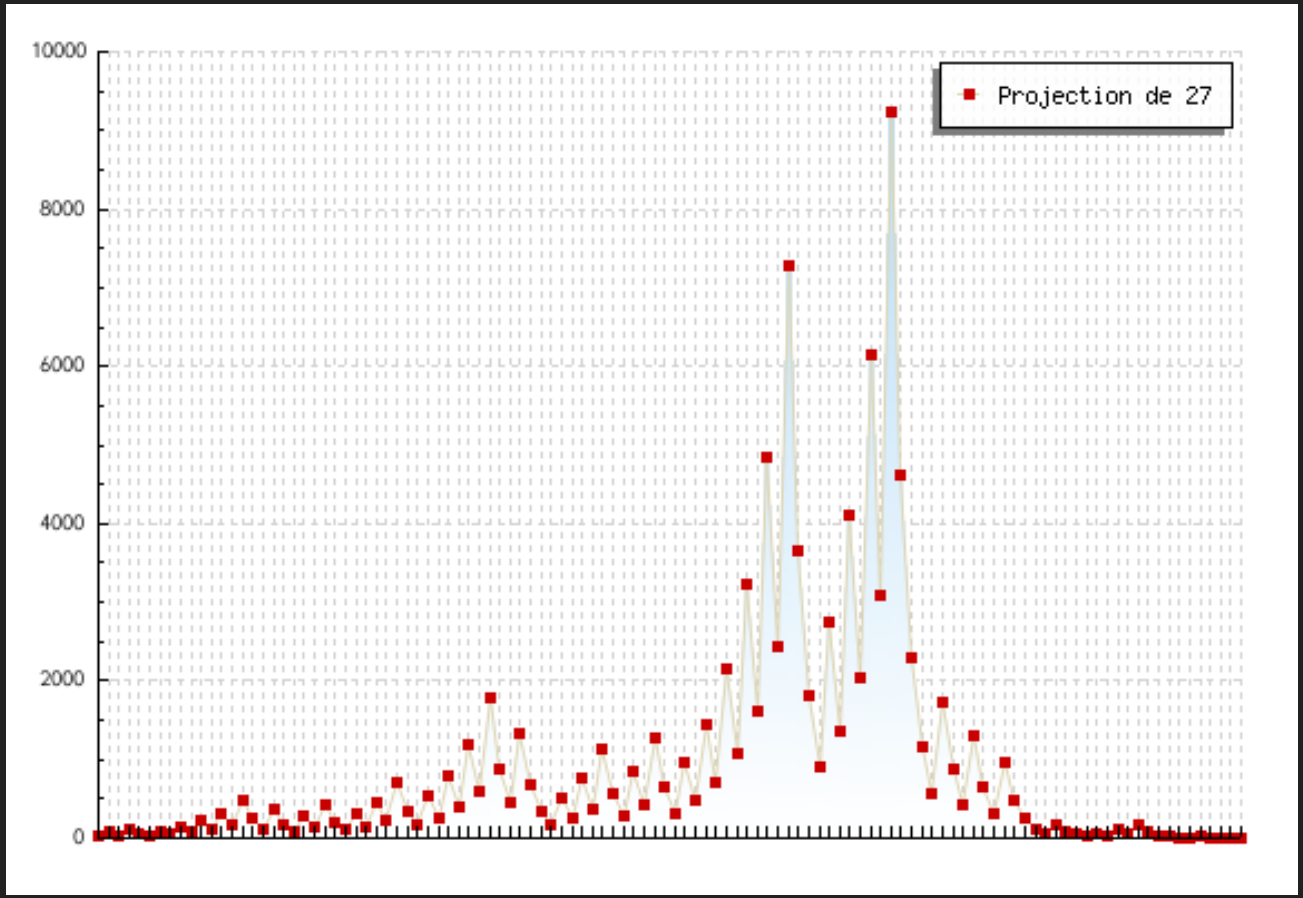
Peut-on prouver que l'algorithme ci-dessous se termine ?

Suite de Syracuse

```
// Données : N : int
// Résultat: suite d'entiers Nk, k=int(0)
Nk : [Int](maxint)
Nk[0]= N
while Nk[k]!=1 do
  k=k+1
  if Nk[k-1]%2 == 0 then
    Nk[k] = Nk[k-1]/2
  else
    Nk[k] = 3×Nk[k-1]+1
  endif
endw
```



Preuve : terminaison



Preuve : terminaison

IV.3.4 Conclusion

Pour prouver un algorithme, il faudra donc faire des preuves de ***correction partielle*** associées à des ***preuves de terminaison***.

De plus, pour chaque boucle, il faut :

- faire figurer l'invariant sous forme de commentaire
- penser à vérifier la terminaison
- pour raisonner sans se tromper, il est prudent, après la fin de boucle, d'écrire la condition d'arrêt sous forme de commentaire.

Pour chaque algorithme, il est donc impératif de faire figurer sa spécification sous forme de commentaires

- pour les boucles non triviales, ***faire figurer l'invariant*** dans le source sous forme de commentaire
- penser à ***vérifier la terminaison***
- pour raisonner sans se tromper : ***écrire la condition d'arrêt*** sous forme de commentaire
- dans les cas compliqués, commenter les `if..then..else`.

Aide à la vérification : utiliser les assertions lorsque le langage le permet, sinon les écrire sous forme de commentaire.