Définitions

- Un algorithme est une suite d'instructions et d'opérations permettant de résoudre un problème.

Définitions

- Un algorithme est une suite d'instructions et d'opérations permettant de résoudre un problème.

  Par exemple pour résoudre le problème du tri d'une liste de valeurs, on peut utiliser l'algorithme du tri par insertion. Cela consiste à prendre une à une chaque valeur et à l'insérer au bon emplacement dans une liste initialement vide.

Définitions

- Un algorithme est une suite d'instructions et d'opérations permettant de résoudre un problème.

  Par exemple pour résoudre le problème du tri d'une liste de valeurs, on peut utiliser l'algorithme du tri par insertion. Cela consiste à prendre une à une chaque valeur et à l'insérer au bon emplacement dans une liste initialement vide.

- Un programme est la traduction d'un algorithme dans un langage informatique. On dit qu'un programme est la mise en oeuvre d'un algorithme.

Définitions

- Un algorithme est une suite d'instructions et d'opérations permettant de résoudre un problème.

  Par exemple pour résoudre le problème du tri d'une liste de valeurs, on peut utiliser l'algorithme du tri par insertion. Cela consiste à prendre une à une chaque valeur et à l'insérer au bon emplacement dans une liste initialement vide.

- Un programme est la traduction d'un algorithme dans un langage informatique. On dit qu'un programme est la mise en oeuvre d'un algorithme.

  L'algorithme du tri par insertion peut être écrit en Python, mais aussi dans un autre langage de programmation.

Définitions

- Un algorithme est une suite d'instructions et d'opérations permettant de résoudre un problème.

  Par exemple pour résoudre le problème du tri d'une liste de valeurs, on peut utiliser l'algorithme du tri par insertion. Cela consiste à prendre une à une chaque valeur et à l'insérer au bon emplacement dans une liste initialement vide.

- Un programme est la traduction d'un algorithme dans un langage informatique. On dit qu'un programme est la mise en oeuvre d'un algorithme.

  L'algorithme du tri par insertion peut être écrit en Python, mais aussi dans un autre langage de programmation.

- L'algorithme (la méthode) est donc indépendante du langage utilisé pour le mettre en oeuvre sur un ordinateur.

Définitions

- Un algorithme est une suite d'instructions et d'opérations permettant de résoudre un problème.

  Par exemple pour résoudre le problème du tri d'une liste de valeurs, on peut utiliser l'algorithme du tri par insertion. Cela consiste à prendre une à une chaque valeur et à l'insérer au bon emplacement dans une liste initialement vide.
- Un programme est la traduction d'un algorithme dans un langage informatique. On dit qu'un programme est la mise en oeuvre d'un algorithme.

  L'algorithme du tri par insertion peut être écrit en Python, mais aussi dans un autre langage de programmation.
- L'algorithme (la méthode) est donc indépendante du langage utilisé pour le mettre en oeuvre sur un ordinateur.
- Pour décrire l'algorithme on utilise parfois un *pseudo langage*, permettant de donner la suite des instructions, tests et opérations de l'algorithme.

Exemples

Algorithme : Algorithme mystère

Entrées : $n \in \mathbb{N}, m \in \mathbb{N}$
Sorties : $?$

```
1 r ← 0
2 tant que m > 0 faire
3 |   m ← m − 1
4 |   r ← r + n
5 fin
6 return r
```

Exemples

Algorithme : Algorithme mystère

Entrées : $n \in \mathbb{N}, m \in \mathbb{N}$
Sorties : $?$

```
1 r ← 0
2 tant que m > 0 faire
3 |   m ← m − 1
4 |   r ← r + n
5 fin
6 return r
```

1. Faire fonctionner cet algorithme avec $n = 3$ et $m = 4$.

Exemples

Algorithme : Algorithme mystère

Entrées : $n \in \mathbb{N}, m \in \mathbb{N}$
Sorties : ?

```
1 r ← 0
2 tant que m > 0 faire
3 |   m ← m − 1
4 |   r ← r + n
5 fin
6 return r
```

1. Faire fonctionner cet algorithme avec $n = 3$ et $m = 4$.
2. Quel est le rôle de cet algorithme ? Proposer un nom pour cet algorithme.

Exemples

Algorithme : Algorithme mystère

Entrées : $n \in \mathbb{N}, m \in \mathbb{N}$
Sorties : ?

```
1 r ← 0
2 tant que m > 0 faire
3 |   m ← m − 1
4 |   r ← r + n
5 fin
6 return r
```

1. Faire fonctionner cet algorithme avec $n = 3$ et $m = 4$.
2. Quel est le rôle de cet algorithme ? Proposer un nom pour cet algorithme.
3. Donner une implémentation en Python de cet algorithme.

Algorithmique

Dans l'étude des algorithmes (algorithmique), on s'intéresse aux trois problèmes suivants :

Algorithmique

Dans l'étude des algorithmes (algorithmique), on s'intéresse aux trois problèmes suivants :

1. terminaison : peut-on garantir que l'algorithme se termine en un temps fini ? (ne concerne que les algorithmes récursifs ou avec des boucles non bornées)

Algorithmique

Dans l'étude des algorithmes (algorithmique), on s'intéresse aux trois problèmes suivants :

1. terminaison : peut-on garantir que l'algorithme se termine en un temps fini ? (ne concerne que les algorithmes récursifs ou avec des boucles non bornées)
2. correction : peut-on garantir que l'algorithme fournit la réponse attendue ?

Algorithmique

Dans l'étude des algorithmes (algorithmique), on s'intéresse aux trois problèmes suivants :

1. terminaison : peut-on garantir que l'algorithme se termine en un temps fini ? (ne concerne que les algorithmes récursifs ou avec des boucles non bornées)
2. correction : peut-on garantir que l'algorithme fournit la réponse attendue ?
3. complexité : evolution du temps d'exécution de l'algorithme en fonction de la taille des données (prédiction de temps de calcul, comparaisons des performances d'algorithmes résolvant le même problème.)

Algorithmique

Dans l'étude des algorithmes (algorithmique), on s'intéresse aux trois problèmes suivants :

1. terminaison : peut-on garantir que l'algorithme se termine en un temps fini ? (ne concerne que les algorithmes récursifs ou avec des boucles non bornées)
2. correction : peut-on garantir que l'algorithme fournit la réponse attendue ?
3. complexité : evolution du temps d'exécution de l'algorithme en fonction de la taille des données (prédiction de temps de calcul, comparaisons des performances d'algorithmes résolvant le même problème.)

L'algorithme étudié doit avoir une spécification précise (entrées, sorties, préconditions, postconditions, effets de bord). On parle d'algorithmes (et non de programmes) car ces questions sont indépendantes de l'implémentation dans un langage de programmation quelconque.

Exemple

Dans le cas de l'algorithme permettant de multiplier deux entiers positifs m et n sans utiliser l'opérateur de multiplication, on veut donc :

Exemple

Dans le cas de l'algorithme permettant de multiplier deux entiers positifs m et n sans utiliser l'opérateur de multiplication, on veut donc :

- *Prouver* que cet algorithme s'arrête après un nombre fini d'étapes (terminaison).

Exemple

Dans le cas de l'algorithme permettant de multiplier deux entiers positifs m et n sans utiliser l'opérateur de multiplication, on veut donc :

- *Prouver* que cet algorithme s'arrête après un nombre fini d'étapes (terminaison).
- *Prouver* que le résultat renvoyé est bien $n \times m$ (correction).

Exemple

Dans le cas de l'algorithme permettant de multiplier deux entiers positifs m et n sans utiliser l'opérateur de multiplication, on veut donc :

- *Prouver* que cet algorithme s'arrête après un nombre fini d'étapes (terminaison).
- *Prouver* que le résultat renvoyé est bien $n \times m$ (correction).
- Donner une mesure de l'évolution du nombre d'opérations réalisés par l'algorithme en fonction de la taille des entrées n et m (complexité).

Exemple introductif

Algorithme : Multiplier sans utiliser $*$

Entrées : $n \in \mathbb{N}, m \in \mathbb{N}$
Sorties : nm

```
1 r ← 0
2 tant que m > 0 faire
3 |   m ← m − 1
4 |   r ← r + n
5 fin
6 return r
```

Exemple introductif

Algorithme : Multiplier sans utiliser $*$

Entrées : $n \in \mathbb{N}, m \in \mathbb{N}$
Sorties : nm

```
1 r ← 0
2 tant que m > 0 faire
3 |   m ← m − 1
4 |   r ← r + n
5 fin
6 return r
```

1. Montrer que dans la boucle "tant que" :

Exemple introductif

Algorithme : Multiplier sans utiliser $*$

Entrées : $n \in \mathbb{N}, m \in \mathbb{N}$
Sorties : nm

```
1 r ← 0
2 tant que m > 0 faire
3 |   m ← m − 1
4 |   r ← r + n
5 fin
6 return r
```

1. Montrer que dans la boucle "tant que" :
   - m ne prend que des valeurs entières

Exemple introductif

Algorithme : Multiplier sans utiliser $*$

Entrées : $n \in \mathbb{N}, m \in \mathbb{N}$
Sorties : nm

```
1 r ← 0
2 tant que m > 0 faire
3 |   m ← m − 1
4 |   r ← r + n
5 fin
6 return r
```

1. Montrer que dans la boucle "tant que" :
   - m ne prend que des valeurs entières
   - m prend des valeurs positives

Exemple introductif

Algorithme : Multiplier sans utiliser $*$

Entrées : $n \in \mathbb{N}, m \in \mathbb{N}$
Sorties : nm

```
1 r ← 0
2 tant que m > 0 faire
3 |   m ← m − 1
4 |   r ← r + n
5 fin
6 return r
```

1. Montrer que dans la boucle "tant que" :
   - m ne prend que des valeurs entières
   - m prend des valeurs positives
   - les valeurs prises par m sont **strictement** décroissantes.

Définitions

Définitions

- On dit qu'un algorithme termine lorsqu'il renvoie un résultat en un nombre fini d'étapes quels que soient les valeurs des entrées.

Définitions

- On dit qu'un algorithme termine lorsqu'il renvoie un résultat en un nombre fini d'étapes quels que soient les valeurs des entrées.
- Un variant de boucle est une quantité :

Définitions

- On dit qu'un algorithme termine lorsqu'il renvoie un résultat en un nombre fini d'étapes quels que soient les valeurs des entrées.
- Un variant de boucle est une quantité :
  1. à valeurs entières,

Définitions

- On dit qu'un algorithme termine lorsqu'il renvoie un résultat en un nombre fini d'étapes quels que soient les valeurs des entrées.
- Un variant de boucle est une quantité :
  1. à valeurs entières,
  2. positives,

Définitions

- On dit qu'un algorithme termine lorsqu'il renvoie un résultat en un nombre fini d'étapes quels que soient les valeurs des entrées.
- Un variant de boucle est une quantité :
  1. à valeurs entières,
  2. positives,
  3. qui décroît *strictement* à chaque passage dans la boucle.

Définitions

- On dit qu'un algorithme termine lorsqu'il renvoie un résultat en un nombre fini d'étapes quels que soient les valeurs des entrées.
- Un variant de boucle est une quantité :
  1. à valeurs entières,
  2. positives,
  3. qui décroît *strictement* à chaque passage dans la boucle.

Propriété

Si une boucle admet un variant, alors cette boucle termine.

Définitions

- On dit qu'un algorithme termine lorsqu'il renvoie un résultat en un nombre fini d'étapes quels que soient les valeurs des entrées.
- Un variant de boucle est une quantité :
  1. à valeurs entières,
  2. positives,
  3. qui décroît *strictement* à chaque passage dans la boucle.

Propriété

Si une boucle admet un variant, alors cette boucle termine.

Exemple

m est un variant de boucle de l'algorithme de multiplication ci-dessus et donc cet algorithme termine.

Exemple

Algorithme : Nombre de chiffres en base 10

Entrées : $n \in \mathbb{N}$

Sorties : p nombre de chiffres de n dans son écriture en base 10.

```
1 si n = 0 alors
2 |   return 1
3 fin
4 p ← 0
5 tant que n > 0 faire
6 |   p ← p + 1
7 |   n ← ⌊n/10⌋
8 fin
9 return p
```

Exemple

Algorithme : Nombre de chiffres en base 10

Entrées : $n \in \mathbb{N}$

Sorties : p nombre de chiffres de n dans son écriture en base 10.

```
1 si n = 0 alors
2 |   return 1
3 fin
4 p ← 0
5 tant que n > 0 faire
6 |   p ← p + 1
7 |   n ← ⌊n/10⌋
8 fin
9 return p
```

1. Donner une implémentation en Python de cet algorithme.

Exemple

Algorithme : Nombre de chiffres en base 10

Entrées : $n \in \mathbb{N}$

Sorties : p nombre de chiffres de n dans son écriture en base 10.

```
1 si n = 0 alors
2 |   return 1
3 fin
4 p ← 0
5 tant que n > 0 faire
6 |   p ← p + 1
7 |   n ← ⌊n/10⌋
8 fin
9 return p
```

1. Donner une implémentation en Python de cet algorithme.
2. Prouver la terminaison de cet algorithme.

Implémentation en Python

```
1  def nbchiffres(n:int)->int:
2      '''Renvoie le nombres de chiffres de n en base 10'''
3      assert n>=0, "Le nombre doit être positif"
4      if n==0:
5          return 1
6      p = 0
7      while n>0:
8          p = p + 1
9          n = n//10
10     return p
```

Preuve de terminaison

Montrons que n est un variant de la boucle `tant que` de l'algorithme :

1. n ne prend que des valeurs entières, en effet, $n \in \mathbb{N}$ en entrée et $\left\lfloor \frac{n}{10} \right\rfloor$ est entier.

Preuve de terminaison

Montrons que n est un variant de la boucle `tant que` de l'algorithme :

1. n ne prend que des valeurs entières, en effet, $n \in \mathbb{N}$ en entrée et $\left\lfloor \frac{n}{10} \right\rfloor$ est entier.
2. n est positif par condition d'entrée dans la boucle.

Preuve de terminaison

Montrons que n est un variant de la boucle `tant que` de l'algorithme :

1. n ne prend que des valeurs entières, en effet, $n \in \mathbb{N}$ en entrée et $\left\lfloor \frac{n}{10} \right\rfloor$ est entier.
2. n est positif par condition d'entrée dans la boucle.
3. n décroît strictement à chaque passage dans la boucle car comme $n > 0$, $\left\lfloor \frac{n}{10} \right\rfloor < n$

Exercice

1. Ecrire un algorithme qui prend en entrée un entier $n > 1$ et renvoie le premier diviseur strictement supérieur à 1 de cet entier. Par exemple pour $n = 7$ l'algorithme renvoie 7 et pour $n = 15$, l'algorithme renvoie 3.

Exercice

1. Ecrire un algorithme qui prend en entrée un entier $n > 1$ et renvoie le premier diviseur strictement supérieur à 1 de cet entier. Par exemple pour $n = 7$ l'algorithme renvoie 7 et pour $n = 15$, l'algorithme renvoie 3.
2. Donner une implémentation en Python de cet algorithme sous la forme d'une fonction dont on précisera soigneusement la spécification.

Exercice

1. Ecrire un algorithme qui prend en entrée un entier $n > 1$ et renvoie le premier diviseur strictement supérieur à 1 de cet entier. Par exemple pour $n = 7$ l'algorithme renvoie 7 et pour $n = 15$, l'algorithme renvoie 3.
2. Donner une implémentation en Python de cet algorithme sous la forme d'une fonction dont on précisera soigneusement la spécification.
3. Prouver la terminaison de cet algorithme.

Exemple introductif

Algorithme : Multiplier sans utiliser $*$

Entrées : $n \in \mathbb{N}, m \in \mathbb{N}$
Sorties : nm

```
1 r ← 0
2 tant que m > 0 faire
3 |   m ← m − 1
4 |   r ← r + n
5 fin
6 return r
```

On note m_0 la valeur initiale de m et on considère la propriété suivante suivante noté I : « $r = (m_0 - m)n$ ». Montrer que cette propriété est vraie :

Exemple introductif

Algorithme : Multiplier sans utiliser $*$

Entrées : $n \in \mathbb{N}, m \in \mathbb{N}$
Sorties : nm

```
1 r ← 0
2 tant que m > 0 faire
3 |   m ← m − 1
4 |   r ← r + n
5 fin
6 return r
```

On note m_0 la valeur initiale de m et on considère la propriété suivante suivante noté I : « $r = (m_0 - m)n$ ». Montrer que cette propriété est vraie :

1. avant d'entrée dans la boucle,

Exemple introductif

Algorithme : Multiplier sans utiliser $*$

Entrées : $n \in \mathbb{N}, m \in \mathbb{N}$
Sorties : nm

```
1 r ← 0
2 tant que m > 0 faire
3 |   m ← m − 1
4 |   r ← r + n
5 fin
6 return r
```

On note m_0 la valeur initiale de m et on considère la propriété suivante suivante noté I : « $r = (m_0 - m)n$ ». Montrer que cette propriété est vraie :

1. avant d'entrée dans la boucle,
2. qu'elle reste vraie à chaque tour de boucle.

Exemple introductif

Algorithme : Multiplier sans utiliser $*$

Entrées : $n \in \mathbb{N}, m \in \mathbb{N}$
Sorties : nm

```
1 r ← 0
2 tant que m > 0 faire
3 |   m ← m − 1
4 |   r ← r + n
5 fin
6 return r
```

On note m_0 la valeur initiale de m et on considère la propriété suivante suivante noté I : « $r = (m_0 - m)n$ ». Montrer que cette propriété est vraie :

1. avant d'entrée dans la boucle,
2. qu'elle reste vraie à chaque tour de boucle.

Que peut-on en conclure ?

Définitions

- Un invariant de boucle est une propriété qui :

Définitions

- Un invariant de boucle est une propriété qui :
  - est vraie à l'entrée dans la boucle (initialisation),

Définitions

- Un invariant de boucle est une propriété qui :
  - est vraie à l'entrée dans la boucle (initialisation),
  - reste vraie à chaque itération si elle l'était à l'itération précédente (conservation).

Définitions

- Un invariant de boucle est une propriété qui :
  - est vraie à l'entrée dans la boucle (initialisation),
  - reste vraie à chaque itération si elle l'était à l'itération précédente (conservation).
- Un algorithme est dit partiellement correct lorsqu'il renvoie la réponse attendue quand il se termine.

Définitions

- Un invariant de boucle est une propriété qui :
  - est vraie à l'entrée dans la boucle (initialisation),
  - reste vraie à chaque itération si elle l'était à l'itération précédente (conservation).
- Un algorithme est dit partiellement correct lorsqu'il renvoie la réponse attendue quand il se termine.
- Un algorithme est dit totalement correct lorsqu'il est partiellement correcte et que sa terminaison est prouvée.

Définitions

- Un invariant de boucle est une propriété qui :
  - est vraie à l'entrée dans la boucle (initialisation),
  - reste vraie à chaque itération si elle l'était à l'itération précédente (conservation).
- Un algorithme est dit partiellement correct lorsqu'il renvoie la réponse attendue quand il se termine.
- Un algorithme est dit totalement correct lorsqu'il est partiellement correcte et que sa terminaison est prouvée.

Définitions

- Un invariant de boucle est une propriété qui :
  - est vraie à l'entrée dans la boucle (initialisation),
  - reste vraie à chaque itération si elle l'était à l'itération précédente (conservation).
- Un algorithme est dit partiellement correct lorsqu'il renvoie la réponse attendue quand il se termine.
- Un algorithme est dit totalement correct lorsqu'il est partiellement correcte et que sa terminaison est prouvée.

Prouver la correction d'un algorithme

On utilise un invariant de boucle qui en sortie de boucle fournit une propriété permettant de montrer la correction.
La méthode est similaire à une récurrence mathématique.

Exemple

Algorithme : Nombre de chiffres en base 10

Entrées : $n \in \mathbb{N}$

Sorties : p nombre de chiffres de n dans son écriture en base 10.

```
1 si n = 0 alors
2 |   return 1
3 fin
4 p ← 0
5 tant que n > 0 faire
6 |   p ← p + 1
7 |   n ← ⌊n/10⌋
8 fin
9 return p
```

Prouver que cet algorithme est correct.

Correction

Idée : à chaque tour de boucle, on incrémente p et n perd un chiffre. La somme de p et du nombre de chiffres de n est donc constante. C'est l'invariant de boucle !

Correction

Idée : à chaque tour de boucle, on incrémente p et n perd un chiffre. La somme de p et du nombre de chiffres de n est donc constante. C'est l'invariant de boucle !

Le cas $n = 0$ est trivial, on note n_0 la valeur initiale de n et on suppose donc $n_0 > 0$, on considère la propriété I : « $c(n_0) = p + c(n)$ ». Où $c(m)$ vaut 0 si $m = 0$ et le nombre de chiffres de m en base 10 sinon.

Correction

Idée : à chaque tour de boucle, on incrémente p et n perd un chiffre. La somme de p et du nombre de chiffres de n est donc constante. C'est l'invariant de boucle !

Le cas $n = 0$ est trivial, on note n_0 la valeur initiale de n et on suppose donc $n_0 > 0$, on considère la propriété I : « $c(n_0) = p + c(n)$ ». Où $c(m)$ vaut 0 si $m = 0$ et le nombre de chiffres de m en base 10 sinon.

- initialisation : I est vraie avant d'entrer dans la boucle puisque $p = 0$ et $n = n_0$.

Correction

Idée : à chaque tour de boucle, on incrémente p et n perd un chiffre. La somme de p et du nombre de chiffres de n est donc constante. C'est l'invariant de boucle !

Le cas $n = 0$ est trivial, on note n_0 la valeur initiale de n et on suppose donc $n_0 > 0$, on considère la propriété I : « $c(n_0) = p + c(n)$ ». Où $c(m)$ vaut 0 si $m = 0$ et le nombre de chiffres de m en base 10 sinon.

- initialisation : I est vraie avant d'entrer dans la boucle puisque $p = 0$ et $n = n_0$.
- conservation : on suppose I vraie du début d'un tour de boucle et on note n' (resp. p') la valeur de n (resp p') après ce tour de boucle

Correction

Idée : à chaque tour de boucle, on incrémente p et n perd un chiffre. La somme de p et du nombre de chiffres de n est donc constante. C'est l'invariant de boucle !

Le cas $n = 0$ est trivial, on note n_0 la valeur initiale de n et on suppose donc $n_0 > 0$, on considère la propriété I : « $c(n_0) = p + c(n)$ ». Où $c(m)$ vaut 0 si $m = 0$ et le nombre de chiffres de m en base 10 sinon.

- initialisation : I est vraie avant d'entrer dans la boucle puisque $p = 0$ et $n = n_0$.
- conservation : on suppose I vraie du début d'un tour de boucle et on note n' (resp. p') la valeur de n (resp p') après ce tour de boucle
  $p' + c(n') = p + 1 + c\left(\left\lfloor \frac{n}{10} \right\rfloor\right)$

Correction

Idée : à chaque tour de boucle, on incrémente p et n perd un chiffre. La somme de p et du nombre de chiffres de n est donc constante. C'est l'invariant de boucle !

Le cas $n = 0$ est trivial, on note n_0 la valeur initiale de n et on suppose donc $n_0 > 0$, on considère la propriété I : « $c(n_0) = p + c(n)$ ». Où $c(m)$ vaut 0 si $m = 0$ et le nombre de chiffres de m en base 10 sinon.

- initialisation : I est vraie avant d'entrer dans la boucle puisque $p = 0$ et $n = n_0$.
- conservation : on suppose I vraie du début d'un tour de boucle et on note n' (resp. p') la valeur de n (resp p') après ce tour de boucle
  $p' + c(n') = p + 1 + c\left(\left\lfloor \frac{n}{10} \right\rfloor\right)$
  $p' + c(n') = p + 1 + c(n) - 1$

Correction

Idée : à chaque tour de boucle, on incrémente p et n perd un chiffre. La somme de p et du nombre de chiffres de n est donc constante. C'est l'invariant de boucle !

Le cas $n = 0$ est trivial, on note n_0 la valeur initiale de n et on suppose donc $n_0 > 0$, on considère la propriété I : « $c(n_0) = p + c(n)$ ». Où $c(m)$ vaut 0 si $m = 0$ et le nombre de chiffres de m en base 10 sinon.

- initialisation : I est vraie avant d'entrer dans la boucle puisque $p = 0$ et $n = n_0$.
- conservation : on suppose I vraie du début d'un tour de boucle et on note n' (resp. p') la valeur de n (resp p') après ce tour de boucle
  $p' + c(n') = p + 1 + c\left(\left\lfloor \frac{n}{10} \right\rfloor\right)$
  $p' + c(n') = p + 1 + c(n) - 1$
  $p' + c(n') = c(n_0)$, car I est vraie à l'entrée de la boucle.

Correction

Idée : à chaque tour de boucle, on incrémente p et n perd un chiffre. La somme de p et du nombre de chiffres de n est donc constante. C'est l'invariant de boucle !

Le cas $n = 0$ est trivial, on note n_0 la valeur initiale de n et on suppose donc $n_0 > 0$, on considère la propriété I : « $c(n_0) = p + c(n)$ ». Où $c(m)$ vaut 0 si $m = 0$ et le nombre de chiffres de m en base 10 sinon.

- initialisation : I est vraie avant d'entrer dans la boucle puisque $p = 0$ et $n = n_0$.
- conservation : on suppose I vraie du début d'un tour de boucle et on note n' (resp. p') la valeur de n (resp p') après ce tour de boucle
  $p' + c(n') = p + 1 + c\left(\left\lfloor \frac{n}{10} \right\rfloor\right)$
  $p' + c(n') = p + 1 + c(n) - 1$
  $p' + c(n') = c(n_0)$, car I est vraie à l'entrée de la boucle.

En sortie de boucle, on a $n' = 0$ et donc $p = c(n_0)$ et l'algorithme est correct.

Exemple

Algorithme : Premier diviseur

Entrées : $n \in \mathbb{N}, n > 1$

Sorties : d premier diviseur de n strictement supérieur à 1.

```
1 d ← 2
2 tant que n mod d ≠ 0 faire
3 |   d ← d + 1
4 fin
5 return d
```

Prouver que cet algorithme est correct.

Exercice 1 : fonction mystère

On considère la fonction Python ci-dessous :

```python
def mystere(a: int, b: int) -> int:
    assert (a >= 0 and b > 0)
    q = 0
    while (a-b >= 0):
        a = a - b
        q = q + 1
    return q
```

1. Proposer un nom et une spécification pour cette fonction

Exercice 1 : fonction mystère

On considère la fonction Python ci-dessous :

```python
def mystere(a: int, b: int) -> int:
    assert (a >= 0 and b > 0)
    q = 0
    while (a-b >= 0):
        a = a - b
        q = q + 1
    return q
```

1. Proposer un nom et une spécification pour cette fonction
2. Prouver sa terminaison

Exercice 1 : fonction mystère

On considère la fonction Python ci-dessous :

```python
def mystere(a: int, b: int) -> int:
    assert (a >= 0 and b > 0)
    q = 0
    while (a-b >= 0):
        a = a - b
        q = q + 1
    return q
```

1. Proposer un nom et une spécification pour cette fonction
2. Prouver sa terminaison
3. Prouver sa correction par rapport à la spécification proposée à la question 1.

Exercice 2 : exponentiation rapide

On donne ci-dessous l'algorithme d'exponentiation rapide en version itérative :

Algorithme : Exponentiation rapide

Entrées : $a \in \mathbb{R}, n \in \mathbb{N}$

Sorties : a^n

```
1 p ← 1
2 tant que n ≠ 0 faire
3 |   si n est impair alors
4 |   |   p ← p × a
5 |   fin
6 |   a ← a ∗ a
7 |   n ← ⌊n/2⌋
8 fin
9 return p
```

1. Donner les valeurs prises par a, n et p lorsqu'on fait fonctionner cet algorithme avec $a = 2$ et $n = 13$.
2. Donner une implémentation de cet algorithme en Python.
3. Prouver que cet algorithme termine.
4. Prouver qu'il est correct.
   En notant a_0 (resp. n_0) la valeur initiale de a (resp. n), on pourra prouver l'invariant $p \times a^n = a_0^{n_0}$.