Éléments de Programmation + Cours 03 - Boucles

Romain Demangeon

LU1IN011 - Section ScFo 11 + 13 + ...

26/09/2022

SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS DEPUIS 1257

Expressivité et Terminaison

Expressivité

L'expressivité d'un langage, c'est l'ensemble des fonctions mathématiques qui peuvent être calculées par des fonctions informatiques (programmes) écrites dans ce langage.

- Jusqu'ici, trois instructions:
  - `return`,
  - affectation `=`,
  - alternative `if :` / `else :`.
- Expressivité limitée (calculatrice): primalité ?
- Implémentation de formules mathématiques (conditionnées).

Terminaison

Une fonction (informatique) `f` termine sur l'entrée `e` quand l'exécution de `f(e)` finit par s'arrêter. Elle termine quand elle termine sur toutes ses entrées.

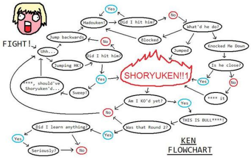
- Toutes nos fonctions terminent trivialement.

Que nous manque t-il ?

Objectif

Pouvoir répéter une action *aussi longtemps que nécessaire*.

- Analogie culinaire:
  - monter des blancs en neige,
  - cuire un gâteau.
- Répéter des actions similaires, potentiellement différentes.
- Comment exprimer aussi longtemps que nécessaire ?
- Terminaison ?



Calcul de la somme des premiers entiers

Problème

Calculer la somme des n premiers entiers.

Calcul de la somme des premiers entiers

Problème

Calculer la somme des n premiers entiers.

(*willing suspension of disbelief:* "Gauss n'a jamais existé: $\sum_{i=1}^{n} i = \frac{n.(n+1)}{2}$ est inconnue.")

Par exemple, si n vaut 5

```python
def somme5() -> int:
    """retourne la somme des 5 premiers entiers naturels."""

    return 1 + 2 + 3 + 4 + 5

# Test
assert somme5() == 15
```

Calcul de la somme des premiers entiers

Problème

Calculer la somme des n premiers entiers.

(*willing suspension of disbelief:* "Gauss n'a jamais existé: $\sum_{i=1}^{n} i = \frac{n.(n+1)}{2}$ est inconnue.")

Par exemple, si n vaut 5

```
def somme5() -> int:
    """retourne la somme des 5 premiers entiers naturels."""

    return 1 + 2 + 3 + 4 + 5

# Test
assert somme5() == 15
```

- Malaise:
  - solution spécifique à $n = 5$.
    - définir une fonction pour chaque entier.
  - Ecriture fastidieuse quand $n = 100000$.
- On voudrait:
  - une définition générale `def somme(n : int) ->int`,

Calcul de la somme des premiers entiers (II)

- Calculs répétitifs: nombre d'étapes n'est pas fixe.
- Math: formule générale $\sum_{i=1}^{n} i = 1 + 2 + \cdots + n$.
  - n est paramètre de la formule (pas i).
- A la main, approche itérative:
  - la somme s vaut 0 initialement,
  - on ajoute le premier entier 1, s vaut 1,
  - on ajoute l'entier suivant 2, s vaut 3,
  - on ajoute l'entier suivant 3, s vaut 6,
  - on ajoute l'entier suivant 4, s vaut 10,
  - on ajoute l'entier suivant 5, s vaut 15,
  - on a atteint la borne 5, on s'arrête et la somme vaut 15.

```python
def somme_ite5() -> int:
    """retourne la somme des 5 premiers entiers naturels."""

    s : int = 0 # valeur temporaire de la somme

    s = s + 1
    s = s + 2
    s = s + 3
    s = s + 4
    s = s + 5
    return s
```

Calcul de la somme des premiers entiers (III)

- Même traitement à chaque étape.
  - Une instruction: affecter une nouvelle valeur à la variable `s`.
- Idée:
  - une variable i initialisée à 1 pour représenter, successivement les entiers de 1 à n.
  - une variable s initialisée à 0 pour représenter la somme des entiers jusqu'à l'entier courant.
  - a chaque étape (en tout n fois):
    - affecter à s sa valeur courante augmentée de i,
    - faire passer i à l'entier suivant (incrémentation).
  - arrêter quand on a fait n étapes avec n paramètre.
- Outil fourni dans tout (?) langage de prog.: les boucles.

Calcul de la somme des premiers entiers (IV)

- Boucle `while`: répète un bloc d'instruction tant qu'une certaine condition (expression booléenne) est vérifiée.
- Ici, principe de répétition:
  - répéter tant que `i <= 5` (condition) le bloc d'instructions suivant:
    1. Instr. ajouter *s* le contenu de *i* (`s = s + i`).
    2. Instr. incrémenter *i* (`i = i + 1`).

```python
def somme_while5() -> int:
    """retourne la somme des 5 premiers entiers naturels."""

    i : int = 1 # entier courant

    s : int = 0 # la somme cumulee

    while i <= 5:
        s = s + i
        i = i + 1

    return s

# Test
assert somme_while5() == 1 + 2 + 3 + 4 + 5
```

- Terminaison ?: `i` vaut `6` au 5eme tour, condition fausse, on sort.

SORBONNE UNIVERSITÉ CRÉATEURS DE FUTURS DEPUIS 1257

Syntaxe du *while*

- Syntaxe:

```
while cond:
    instruction_1
    instruction_2
    ...
    instruction_n
```

  - `cond` est la condition de la boucle, c'est une expression booléenne
  - ```
        instruction_1
        instruction_2
        ...
        instruction_n
    ```

    est le corps de la boucle (ce qui est répété).
  - le corps est défini par l'indentation:

    ```
    while cond
        instruction_1
        instruction_2
        ...
        instruction_n
    instruction_n1
    ```

    ici, `instruction_n1` ne fait par partie du corps de la boucle, elle n'est pas répétée, elle est exécutée en sortie de la boucle.
    ```

Calcul de la somme des premiers entiers (V)

- Grâce au `while`: écriture synthétique de `somme5`.
- Généraliser la somme: n comme paramètre, pour remplacer `5`.

```python
def somme_entiers(n : int) -> int:
    """Precondition: n >= 1
    retourne la somme des n premiers entiers naturels."""

    i : int = 1 # entier courant, en commencant par 1

    s : int = 0 # la somme cumulee

    while i <= n:
        s = s + i
        i = i + 1

    return s
```

Exercices

Somme des carrés.

Donner une fonction `somme_carres` qui prend en entrée un entier naturel `n` et renvoie la somme des carrées des entiers de 0 jusque `n`.

- Spécification:

Exercices

Somme des carrés.

Donner une fonction `somme_carres` qui prend en entrée un entier naturel `n` et renvoie la somme des carrées des entiers de 0 jusque `n`.

- Spécification: `def somme_carres(n : int) ->int`
- Précondition:

Exercices

Somme des carrés.

Donner une fonction `somme_carres` qui prend en entrée un entier naturel `n` et renvoie la somme des carrées des entiers de 0 jusque `n`.

- Spécification: `def somme_carres(n : int) ->int`
- Précondition: `n >= 0`
- Algorithme:

Exercices

Somme des carrés.

Donner une fonction `somme_carres` qui prend en entrée un entier naturel `n` et renvoie la somme des carrées des entiers de 0 jusque `n`.

- Spécification: `def somme_carres(n : int) ->int`
- Précondition: `n >= 0`
- Algorithme: ajouter incrémentalement le carré de chaque entier, en partant de `0` et en s'arrêtant à `n`
- Implémentation:

Exercices

Somme des carrés.

Donner une fonction `somme_carres` qui prend en entrée un entier naturel `n` et renvoie la somme des carrées des entiers de 0 jusque `n`.

- Spécification: `def somme_carres(n : int) ->int`
- Précondition: `n >= 0`
- Algorithme: ajouter incrémentalement le carré de chaque entier, en partant de `0` et en s'arrêtant à `n`
- Implémentation:

```python
def somme_carres(n : int) -> int:
    """Precondition : n >= 0 """

    i : int = 0
    s : int = 0

    while i <= n:
        s = s + i * i
        i = i + 1
    return s
```

- Validation:

Exercices

Somme des carrés.

Donner une fonction `somme_carres` qui prend en entrée un entier naturel `n` et renvoie la somme des carrées des entiers de 0 jusque `n`.

- Spécification: `def somme_carres(n : int) ->int`
- Précondition: `n >= 0`
- Algorithme: ajouter incrémentalement le carré de chaque entier, en partant de `0` et en s'arrêtant à `n`
- Implémentation:

```python
def somme_carres(n : int) -> int:
  """Precondition : n >= 0 """

  i : int = 0
  s : int = 0

  while i <= n:
      s = s + i * i
      i = i + 1
  return s
```

- Validation:

```python
#Test
assert somme_carres(4) == 30
```

Exercices (II)

Somme des entiers impairs.

Donner une fonction `somme_impairs` qui prend en entrée un entier naturel `n` et renvoie la somme des entiers impairs compris entre 0 et `n` (inclus).

- Spécification:

Exercices (II)

Somme des entiers impairs.

Donner une fonction `somme_impairs` qui prend en entrée un entier naturel `n` et renvoie la somme des entiers impairs compris entre 0 et `n` (inclus).

- Spécification: `def somme_impairs(n : int) ->int`
- Précondition:

Exercices (II)

Somme des entiers impairs.

Donner une fonction `somme_impairs` qui prend en entrée un entier naturel `n` et renvoie la somme des entiers impairs compris entre 0 et `n` (inclus).

- Spécification: `def somme_impairs(n : int) —>int`
- Précondition: `n >= 0`
- Algorithme:

Exercices (II)

Somme des entiers impairs.

Donner une fonction `somme_impairs` qui prend en entrée un entier naturel `n` et renvoie la somme des entiers impairs compris entre 0 et `n` (inclus).

- Spécification: `def somme_impairs(n : int) −>int`
- Précondition: `n >= 0`
- Algorithme:
  - parcourir incrémentalement les entiers, partant de `0` et en s'arrêtant à `n`, et n'ajouter que ceux impairs.
  - parcourir de 2 en 2, en partant de `1` et en s'arrêtant à `n`.
  - parcourir incrémentalement les entiers de `0` à `(n − 1) // 2`, et ajouter le successeur de leur double à chaque étape.
- Implémentation:

```python
def somme_impairs1(n : int) -> int:
    """Precondition : n >= 0 """

    i : int = 0
    s : int = 0
    while i <= n:
        if i % 2 == 1:
            s = s + i
        i = i + 1
    return s
```

```python
def somme_impairs2(n : int) -> int:
    """Precondition : n >= 0 """
    i : int = 1
    s : int = 0
    while i <= n:
        s = s + i
        i = i + 2
    return s
```

```python
def somme_impairs3(n : int) -> int:
    """Precondition : n >= 0 """
    i : int = 0
    s : int = 0
    while i <= (n - 1) // 2:
        s = s + 2 * i + 1
        i = i + 1
    return s
```

- Validation:

Exercices (II)

Somme des entiers impairs.

Donner une fonction `somme_impairs` qui prend en entrée un entier naturel `n` et renvoie la somme des entiers impairs compris entre 0 et `n` (inclus).

- Spécification: `def somme_impairs(n : int) —>int`
- Précondition: `n >= 0`
- Algorithme:
  - parcourir incrémentalement les entiers, partant de `0` et en s'arrêtant à `n`, et n'ajouter que ceux impairs.
  - parcourir de 2 en 2, en partant de `1` et en s'arrêtant à `n`.
  - parcourir incrémentalement les entiers de `0` à `(n — 1) // 2`, et ajouter le successeur de leur double à chaque étape.
- Implémentation:

```
def somme_impairs1(n : int) —> int:
    """Precondition : n >= 0 """

    i : int = 0
    s : int = 0
    while i <= n:
        if i % 2 == 1:
            s = s + i
        i = i + 1
    return s
```

```
def somme_impairs2(n : int) —> int:
    """Precondition : n >= 0 """
    i : int = 1
    s : int = 0
    while i <= n:
        s = s + i
        i = i + 2
    return s
```

```
def somme_impairs3(n : int) —> int:
    """Precondition : n >= 0 """
    i : int = 0
    s : int = 0
    while i <= (n — 1) // 2:
        s = s + 2 * i + 1
        i = i + 1
    return s
```

- Validation:`assert somme_impairs(5) == 9`

SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS DEPUIS 1257

Exercices (III)

Racine cubique approchée.

Donner une fonction `racine_cubique_entiere` qui prend en entrée un entier naturel `n` et renvoie la partie entière de sa racine cubique.

- Spécification:

Exercices (III)

Racine cubique approchée.

Donner une fonction `racine_cubique_entiere` qui prend en entrée un entier naturel `n` et renvoie la partie entière de sa racine cubique.

- Spécification: `def racine_cubique_entiere(n : int) ->int:`
- Hypothèse:

Exercices (III)

Racine cubique approchée.

Donner une fonction `racine_cubique_entiere` qui prend en entrée un entier naturel `n` et renvoie la partie entière de sa racine cubique.

- Spécification: `def racine_cubique_entiere(n : int) ->int:`
- Hypothèse: `n >= 0`
- Algorithme:
  - Problème: pas de primitive pour faire directement la racine cubique.

Exercices (III)

Racine cubique approchée.

Donner une fonction `racine_cubique_entiere` qui prend en entrée un entier naturel `n` et renvoie la partie entière de sa racine cubique.

- Spécification: `def racine_cubique_entiere(n : int) ->int:`
- Hypothèse: `n >= 0`
- Algorithme:
  - Problème: pas de primitive pour faire directement la racine cubique.
  - Solution: on parcourt incrémentalement tous les entiers et on les élève au cube, on s'arrête quand on dépasse `n`.
  - Différence: on ne sait pas à l'avance combien de tours de boucle on va faire.
- Implémentation et Validation:

```python
def racine_cubique_entiere(n : int) -> int:
    """Precondition : n >= 0 """
    racine : int = 0
    while (racine ** 3) <= n:
        racine = racine + 1
    return racine - 1

assert racine_cubique_entiere(30) == 3
```

Exercices (III)

Racine cubique approchée.

Donner une fonction `racine_cubique_entiere` qui prend en entrée un entier naturel `n` et renvoie la partie entière de sa racine cubique.

- Spécification: `def racine_cubique_entiere(n : int) ->int:`
- Hypothèse: `n >= 0`
- Algorithme:
  - Problème: pas de primitive pour faire directement la racine cubique.
  - Solution: on parcourt incrémentalement tous les entiers et on les élève au cube, on s'arrête quand on dépasse `n`.
  - Différence: on ne sait pas à l'avance combien de tours de boucle on va faire.
- Implémentation et Validation:

```python
def racine_cubique_entiere(n : int) -> int:
    """Precondition : n >= 0 """
    racine : int = 0
    while (racine ** 3) <= n:
        racine = racine + 1
    return racine - 1

assert racine_cubique_entiere(30) == 3
```

- `return int(n ** (1 / 3))` fonctionne . . . (intérêt pédagogique nul)

Principe d'interprétation du `while`

Pour interpréter

```
while cond:
    instruction_1
    ...
    instruction_n
instruction_apres_1
...
```

1. on évalue `cond`
2. si la valeur de `cond` n'est pas `False`, on interprète en entier:

   ```
   instruction_1
   ...
   instruction_n
   ```

   et on revient en 1.
3. si la valeur de `cond` est `False`, on sort de la boucle et on interprète la suite:

   ```
   instruction_apres_1
   ...
   ```

Simulation de boucle

Tables de simulation

1. Fixer les valeurs des paramètres (on simule sur un exemple précis)
2. Fixer les valeurs des variables non modifiées par la boucle.
3. Créer un tableau avec:
   3.1 une colonne tour de boucle,
   3.2 une colonne par variable modifiée par la boucle.
4. Remplir une ligne *entrée* avec les valeurs avant la boucle.
5. Décider s'il y a un tour de boucle en évaluant la condition.
6. Si oui, remplir une nouvelle ligne avec les valeurs en fin de tour.
7. Sinon, on écrit *(sortie)* au dernier tour.

Simulation de boucle: Exemple

```
def somme_entiers(n : int) -> int:
    """Precondition: n >= 1
    retourne la somme des n premiers entiers naturels."""

    i : int = 1 # entier courant, en commencant par 1

    s : int = 0 # la somme cumulee

    while i <= n:
        s = s + i
        i = i + 1

    return s
```

somme_entiers(5)

Simulation de boucle: Exemple

```
def somme_entiers(n : int) -> int:
    """Precondition: n >= 1
    retourne la somme des n premiers entiers naturels."""

    i : int = 1 # entier courant, en commencant par 1

    s : int = 0 # la somme cumulee

    while i <= n:
        s = s + i
        i = i + 1

    return s
```

`somme_entiers(5)`

tour de boucle	variable s	variable i
entrée	0	1
1	1	2
2	3	3
3	6	4
4	10	5
5 (sortie)	15	6

Utilisation de print

- **print** permet de tracer (obtenir une trace) des boucles,.
- on obtient exactement une simulation.

```python
def somme_entiers_tracee(n : int) -> int:
    """Precondition: n >= 1
    retourne la somme des n premiers entiers naturels."""

    i : int = 1 # compteur
    s : int = 0 # somme

    print("======================")
    print("s en entree vaut ", s)
    print("i en entree vaut ", i)
    while i <= n:
        s = s + i
        i = i + 1
        print("--------------------")
        print("s apres le tour vaut ", s)
        print("i apres le tour vaut ", i)
    print("--------------------")
    print("sortie")
    print("======================")

    return s
```

Suites récursives

Définition

Une suite récursive $(u_n)_{n\in\mathbb{N}}$ est définie par un premier terme k et une fonction de récursion f. On note:

$$\begin{cases} u_0 & = & k \\ u_{n+1} & = & f(u_n) \quad \text{pour } n \in \mathbb{N} \end{cases}$$

Exemple

$(u_n)_{n\in\mathbb{N}}$ définie par $\begin{cases} u_0 & = & 7 \\ u_{n+1} & = & 2 * u_n + 3 \quad \text{pour } n \in \mathbb{N} \end{cases}$

- Objectif: définir une fonction `valeur_u(n)` renvoyant la valeur du `n`-eme terme de la suite u_n donnée en exemple.
- problème similaire au précédent:
  - boucle `while` avec un compteur `i` et une accumulation `u`.

Suites Récursives (II)

```
def suite_u(n : int) -> int:
    """Precondition: n >= 0
    retourne la valeur au rang n de la suite U."""

    u : int = 7  # valeur au rang 0

    i : int = 0  # initialement rang 0

    while i < n:
        u = 2 * u + 3
        i = i + 1

    return u
```

Simulation suite_u(6)

Suites Récursives (II)

```
def suite_u(n : int) -> int:
    """Precondition: n >= 0
    retourne la valeur au rang n de la suite U."""

    u : int = 7  # valeur au rang 0

    i : int = 0  # initialement rang 0

    while i < n:
        u = 2 * u + 3
        i = i + 1

    return u
```

Simulation `suite_u(6)`

tour de boucle	variable u	variable i
entrée	7	0
1	17	1
2	37	2
3	77	3
4	157	4
5	317	5
6 (sortie)	637	6

Suites Récursives (III)

- Généraliser: définir `suite_rec(n,f,k)` qui renvoie le `n`-ième terme de la suite de premier terme `k` et de fonction de récursion `f`.
- Difficulté: fonction de type `Callable[[int], int]` en paramètre.
- Ordre supérieur:
  - fonctions comme paramètre ou résultat de fonction
  - pas au programme de LU1IN001 (système de types d'ordre 1).
  - style de programmation fonctionnelle (Cours 11, LU2IN019, LI101).
- Présent dans l'informatique moderne (par exemple, dans le Web).

```python
def suite_rec(n : int, f : Callable[[int], int], k : int):
    """ Precondition: n >= 0
    retourne la valeur au rang n de la suite recursive
    de premier terme k et de fonction de recursion f."""

    u : int = k  # valeur au rang 0
    i : int = 0  # initialement rang 0

    while i < n:
        u = f(u)
        i = i + 1
    return u
```

Somme et produit des termes d'une suite

Objectif

Calcul des sommes et produits partiels des termes d'une suite.

Suite dyadique:

$\forall n \in \mathbb{N},\ u_n = \frac{1}{2^n}$

$\forall n \in \mathbb{N},\ S_n = \sum_{k=0}^{n} u_k = \sum_{k=0}^{n} \frac{1}{2^k}$

Somme et produit des termes d'une suite

Objectif

Calcul des sommes et produits partiels des termes d'une suite.

Suite dyadique:

$\forall n \in \mathbb{N},\ u_n = \frac{1}{2^n}$

$\forall n \in \mathbb{N},\ S_n = \sum_{k=0}^{n} u_k = \sum_{k=0}^{n} \frac{1}{2^k}$

```python
def somme_partielle_u(n : int) -> float:
    """ Precondition: n >= 0
    retourne le n-ieme terme de la somme partielle  :
    1 + 1/2 + 1/4 + ... + (1/2)^n"""

    s : float = 0.0  # la somme vaut 0 initialement
    k : int = 0    # on commence au rang 0

    while k <= n:
        s = s + ((1/2) ** k)
        k = k + 1
    return s
```

Somme et produit des termes d'une suite (II)

Factorielle:

$\forall n \in \mathbb{N}^*,\ n! = \prod_{k=1}^{n} k$

Somme et produit des termes d'une suite (II)

Factorielle:

$$\forall n \in \mathbb{N}^*,\ n! = \prod_{k=1}^{n} k$$

```
def factorielle(n : int) -> int:
    """Precondition : n > 0
    retourne le produit factoriel n!"""

    k : int = 1   # on demarre au rang 1

    f : int = 1   # factorielle au rang 1

    while k <= n:
        f = f * k
        k = k + 1

    return f
```

Somme et produit des termes d'une suite (III)

- Objectif; calculer la somme des n-premiers termes d'une suite récursive à partir de son élément initial et de sa fonction de récursion.
- $S_n = \sum_{k=0}^{n} u_k = u_0 + f(u_0) + f(f(u_0)) + \dots$

Somme et produit des termes d'une suite (III)

- Objectif; calculer la somme des n-premiers termes d'une suite récursive à partir de son élément initial et de sa fonction de récursion.
- $S_n = \sum_{k=0}^{n} u_k = u_0 + f(u_0) + f(f(u_0)) + \dots$

```python
def somme_suite_rec(n : int, f : Callable[[float], float], k : float):
    """ Precondition: n >= 0
    renvoie la valeur de la somme partielle des n premiers termes
    de la suite recursive de premier terme k
    et de fonction de recursion f"""

    i : int = 0 # iterateur
    u : int = k # premier terme de la suite
    s : int = k # somme accumulee

    while i < n:
        u = f(u)
        s = s + u
        i = i + 1
    return s
```

Calcul du PGCD

Problème

Calculer le plus grand commun diviseur de deux entiers positifs.

Méthode standard

- `pgcd` doit calculer le pgcd de ses paramètres.
- deux paramètres `a` et `b`, entiers tels que `a>= b >= 0`.
- résultat est un entier.
-

```python
def pgcd(n : int, m : int) -> int:
    """Precondition: n >= m > 0
    Retourne le plus grand commun diviseur de n et m."""
```

- Comment calculer le résultat ?
  - Trouver un algorithme pour résoudre le problème.

Calcul du PGCD: Rappels

- si $(k, n) \in \mathbb{N}^2$, k divise n s'il existe $m \in \mathbb{N}$ tel que $k.m = n$.
  - 3 divise 12 (car $3.4 = 12$).
  - 5 ne divise pas 12.
  - 42 divise 0 (car $42.0 = 0$).
- l'ensemble des diviseurs de $n \in \mathbb{N}$, noté $\texttt{div}(n)$, est l'ensemble des entiers de $\mathbb{N}$ qui divisent n.
  - $\texttt{div}(12) = \{1, 2, 3, 4, 6, 12\}$
  - $\texttt{div}(9) = \{1, 3, 9\}$
  - $\texttt{div}(13) = \{1, 13\}$
  - $\texttt{div}(0) = \mathbb{N}$
- l'ensemble des diviseurs communs de $n \in \mathbb{N}$ et de $m \in \mathbb{N}$, noté $\texttt{div}(n, m)$, est l'intersection des diviseurs de n et m (i.e. $\texttt{div}(n) \cap \texttt{div}(m)$)
  - $\texttt{div}(12, 9) = \{1, 3\}$
  - $\texttt{div}(12, 0) = \{1, 2, 3, 4, 6, 12\}$
- le *pgcd* de $n \in \mathbb{N}^*$ et de $m \in \mathbb{N}$, noté $\texttt{pgcd}(n, m)$, est le plus grand diviseur commun à n et m (i.e. $\texttt{max}(\texttt{div}(n, m))$)
  - $\texttt{pgcd}(12, 9) = 3$
  - $\texttt{pgcd}(12, 0) = 12$

Solution Naïve

- Utilise les ensembles (Cours 09) et les compréhensions (Cours 10)

```
def diviseurs(n : int) -> Set[int]:
    """Precondition : n > 0"""
    return {k for k in range(1, n + 1) if n % k == 0}

def max_ensemble(E : Set[int]) -> int:
    """Precondition: E != set()
    Precondition: les elements de E sont positifs"""
    m : int = -1
    e : int
    for e in E:
        if e > m:
            m = e
    return m

def pgcd_naif(n : int ,m : int) -> int:
    """Precondition: n > 0, m >= 0"""

    if m == 0:
        return n
    else:
        return max_ensemble(diviseurs(n) & diviseurs(m))

assert pgcd_naif(12, 9) == 3
```

- Meilleur algorithme ?

Algorithme d'Euclide

- Soit $n \in \mathbb{N}^*$ et $m \in \mathbb{N}$, la division euclidienne de n par m est l'unique couple $(q, r) \in \mathbb{N} \times [\![0, n-1]\!]$ tel que $n = q.m + r$
  - q est le quotient, obtenu avec `n // m`,
  - r est le reste, obtenu avec `n % m`,
  - avec 12 et 9 on a $(1, 3)$ car $12 = 1.9 + 3$
  - avec 12 et 6 on a $(2, 0)$ car $12 = 2.6 + 0$
- Propriété: Si (q, r) est la division euclidienne de n par m, alors $\mathtt{pgcd}(n, m) = \mathtt{pgcd}(m, r)$.
- Algorithme d'Euclide: Pour calculer $\mathtt{pgcd}(n, m)$:
  1. si m est 0, le pgcd est n,
  2. sinon
     2.1 on calcule r le reste de la division euclidienne de m par n
     2.2 on calcule $\mathtt{pgcd}(m, r)$. (récursion)
- Terminaison: on sait que $r < m$, donc "quelque chose" (ici la somme des deux nombres) décroît à chaque étape.

Algorithme d'Euclide: Exemples

- le pgcd de 56 et 42 est le pgcd de 42 et 14 ($56 = 42 * 1 + 14$)
- le pgcd de 42 et 14 est le pgcd de 14 et 0 ($42 = 14 * 3 + 0$)
- le pgcd de 14 et 0 est 14.

le pgcd de 56 et 42 est 14.

- le pgcd de 4199 et 1530 est le pgcd de 1530 et 1139 ($4199 = 1530 * 2 + 1139$)
- le pgcd de 1530 et 1139 est le pgcd de 1139 et 391 ($1540 = 1139 * 1 + 391$)
- le pgcd de 1139 et 391 est le pgcd de 391 et 357 ($1139 = 391 * 2 + 357$).
- le pgcd de 391 et 357 est le pgcd de 357 et 34 ($391 = 357 * 1 + 34$).
- le pgcd de 357 et 34 est le pgcd de 34 et 17 ($357 = 34 * 10 + 17$).
- le pgcd de 34 et 17 est le pgcd de 17 et 0 ($34 = 17 * 10 + 0$).
- le pgcd de 17 et 0 est 17.

le pgcd de 4199 et 1530 est 17.

Algorithme d'Euclide: Implémentation

```
def pgcd(n : int, m : int) -> int:
    """Precondition: n >= m > 0
    Retourne le plus grand commun diviseur de n et m."""
```

- Variables:

Algorithme d'Euclide: Implémentation

```
def pgcd(n : int, m : int) -> int:
    """Precondition: n >= m > 0
    Retourne le plus grand commun diviseur de n et m."""
```

- Variables: deux variables `d` et `r` pour stocker les deux nombres à chaque étape.
  - elles contiennent initialement `n` et `m`.
- Condition de la boucle:

Algorithme d'Euclide: Implémentation

```python
def pgcd(n : int, m : int) -> int:
    """Precondition: n >= m > 0
    Retourne le plus grand commun diviseur de n et m."""
```

- Variables: deux variables `d` et `r` pour stocker les deux nombres à chaque étape.
  - elles contiennent initialement `n` et `m`.
- Condition de la boucle: continuer à faire des divisions euclidiennes tant que `r` est différent de `0`
  - sinon, le résultat est `d`
- Corps de la boucle:

Algorithme d'Euclide: Implémentation

```python
def pgcd(n : int, m : int) -> int:
    """Precondition: n >= m > 0
    Retourne le plus grand commun diviseur de n et m."""
```

- Variables: deux variables `d` et `r` pour stocker les deux nombres à chaque étape.
  - elles contiennent initialement `n` et `m`.
- Condition de la boucle: continuer à faire des divisions euclidiennes tant que `r` est différent de `0`
  - sinon, le résultat est `d`
- Corps de la boucle: mettre `d % r` dans `r` et `r` dans `d`.
  - ```python
        r = d % r
        d = r
    ```
    ```

Algorithme d'Euclide: Implémentation

```python
def pgcd(n : int, m : int) -> int:
    """Precondition: n >= m > 0
    Retourne le plus grand commun diviseur de n et m."""
```

- Variables: deux variables `d` et `r` pour stocker les deux nombres à chaque étape.
  - elles contiennent initialement `n` et `m`.
- Condition de la boucle: continuer à faire des divisions euclidiennes tant que `r` est différent de `0`
  - sinon, le résultat est `d`
- Corps de la boucle: mettre `d % r` dans `r` et `r` dans `d`.
  - ```python
        r = d % r
        d = r
    ```
  - Problème: instructions exécutées en séquence (`r` change)
    ```

Algorithme d'Euclide: Implémentation

```python
def pgcd(n : int, m : int) -> int:
    """Precondition: n >= m > 0
    Retourne le plus grand commun diviseur de n et m."""
```

- Variables: deux variables `d` et `r` pour stocker les deux nombres à chaque étape.
  - elles contiennent initialement `n` et `m`.
- Condition de la boucle: continuer à faire des divisions euclidiennes tant que `r` est différent de `0`
  - sinon, le résultat est `d`
- Corps de la boucle: mettre `d % r` dans `r` et `r` dans `d`.
  - ```python
        r = d % r
        d = r
    ```
  - Problème: instructions exécutées en séquence (`r` change)
  - Solution: variable temporaire (pour la future valeur de `r`):

    ```python
        temp = d % r
        d = r
        r = temp
    ```
    ```

Algorithme d'Euclide: Implémentation

```python
def pgcd(n : int, m : int) -> int:
    """Precondition: n >= m > 0
    Retourne le plus grand commun diviseur de n et m."""
```

- Variables: deux variables `d` et `r` pour stocker les deux nombres à chaque étape.
  - elles contiennent initialement `n` et `m`.
- Condition de la boucle: continuer à faire des divisions euclidiennes tant que `r` est différent de `0`
  - sinon, le résultat est `d`
- Corps de la boucle: mettre `d % r` dans `r` et `r` dans `d`.
  - ```python
    r = d % r
    d = r
    ```
  - Problème: instructions exécutées en séquence (`r` change)
  - Solution: variable temporaire (pour la future valeur de `r`):

    ```python
    temp = d % r
    d = r
    r = temp
    ```
  - Cours 07: `d, r = r, d % r`
    ```

Algorithme d'Euclide: Implémentation (II)

```
def pgcd(n : int, m : int) -> int:
    """Precondition: n >= m > 0
    Retourne le plus grand commun diviseur de n et m."""

    d : int = n
    r : int = m
    temp :int = 0   #  variable temporaire

    while r != 0:
        temp = d % r
        d = r
        r = temp
    return d
```

Simulation de `pcgd(56, 42)`

Algorithme d'Euclide: Implémentation (II)

```
def pgcd(n : int, m : int) -> int:
    """Precondition: n >= m > 0
    Retourne le plus grand commun diviseur de n et m."""

    d : int = n
    r : int = m
    temp :int = 0   #  variable temporaire

    while r != 0:
        temp = d % r
        d = r
        r = temp
    return d
```

Simulation de `pcgd(56, 42)`

tour de boucle	variable temp	variable q	variable r
entrée	0	56	42
1	14	42	14
2 (sortie)	0	14	0

Boucles imbriquées: couples d'entiers

Problème

Pour un entier positif n fixé, combien y existe t'il de couples d'entiers (i, j) tels que $i + j$ soit divisible par 3.

- pour $n = 0$ on en a 1: $(0, 0)$.
- pour $n = 1$ on en a 1: $(0, 0)$.
- pour $n = 2$ on en a 3: $(0, 0), (1, 2), (2, 1)$.

- Algorithme: impossible avec une boucle.
  - il faut faire varier i et j indépendamment.
  - pour chaque valeur de i, on parcourt toutes les valeurs possibles de j
  - espace quadratique
- Solution: boucles imbriquées.

Boucles imbriquées: couples d'entiers (II)

```
def nombre_couples(n : int) -> int:
    """Precondition : n >= 0
    calcule le nombre de couple (i,j) tels que i.j est divisible par 3"""

    i : int = 0
    j : int = 0
    nb : int = 0

    while i <= n:
        j = 0
        while j <= n:
            if (i + j) % 3 == 0:
                nb = nb + 1
            j = j +1
        i = i + 1
    return nb
```

- l'indentation est cruciale,
- on ne doit pas oublier de remettre `j` à `0`

Boucles imbriquées: couples d'entiers (III)

Simulation de `nombre_couples(2)`

tour de boucle externe	tour de boucle interne	variable i	variable j	variable nb
entrée	-	0	0	0
1	entrée	0	0	0
1	1	0	1	1
1	2	0	2	1
1	3 (sortie)	0	3	1
2	entrée	1	0	1
2	1	1	1	1
2	2	1	2	1
2	3 (sortie)	1	3	2
3	entrée	2	0	2
3	1	2	1	2
3	2	2	2	3
3	3 (sortie)	2	3	3
3 (sortie)	-	3	3	3

- Une colonne par boucle, classées de l'extérieur vers l'intérieur.
- Simulation multiples : pas au programme des examens.
  - mais les boucles imbriquées, oui.
- Facilement traçable.

Zéro d'une fonction sur intervalle (I)

Problème

Décider si une fonction des entiers f s'annule sur l'intervalle entier $[\![a; b]\!]$.

Méthode standard

- la fonction `annule` doit décider si une fonction est égale à `0.0` sur un entier `x` compris entre deux bornes.
- trois arguments: une fonction `f` de type `Callable[[int], float]`, une borne inférieure `a` entière et une borne supérieure `b` entière.
- un résultat booléen.
- Algorithme: utiliser une boucle pour calculer successivement toutes les valeurs de `f` sur les entiers entre `a` et `b`
  - l'itérateur `x` va commencer à `a` puis être incrémenté successivement jusque valoir `b`.

Zéro d'une fonction sur un intervalle (II)

```
def annulation(a : int, b : int, f : Callable[[int], float]) -> bool:
    """ Precondition : a <= b
    Retourne True si la fonction f  s'annule sur l'intervalle [a;b]."""

    x : int = a  # element courant, au debut de l'intervalle

    while (x <= b):
        if f(x) == 0.0:
            return True  # la fonction s'annule !
        else:  # sinon on continue avec l'element suivant
            x = x + 1

    return False  # on sait ici que la fonction ne s'annule pas
```

- Il faut des fonctions utilisables en argument, par exemple:

```
def parabole(x : float) -> float:
    """ Calcule la valeur de X^2+X-6    """

    return x * x + x - 6

assert annulation(0, 10, parabole) == True
assert annulation(10, 20, parabole) == False
```

- Sous-typage contravariant avec l'argument:
  Comme on a `float` ⊆ `int`, on a `int ->float` ⊆ `float->float`

Typage

Typage

Donner un type à une expression c'est indiquer la nature d'une expression.

- Objectifs:
  - Vérifier les appels de fonctions.
  - Valider le code (homogénéité).
  - Gérer la mémoire.
- Typage plus ou moins forts
  - OCaml: `float_of_int(x) +. 2.3`
  - Javascript: `(2 + 3) + " saucisses"`
- Typage explicite: le programmeur doit lui-même indiquer les types (déclarations).
- Typage implicite: le type est inféré par un programme (algorithme d'unification).

Sous-Typage

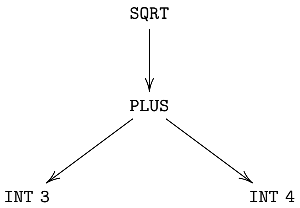
Définition

Un type A est un sous-type de B si toutes les expressions (les objets) de type A sont aussi de type B.

- - `int` est un sous-type de `float`.
  - "entier naturel" est un sous-type de "entier".
  - "poisson" est un sous-type de "animal".
- Si on a besoin d'une expression de type B, et que A est un sous-type de B, on peut prendre une expression de type A.
  - si f prend un entier, je peux calculer $f(3)$.
  - si j'ai besoin d'un animal, je peux prendre un poisson.
- Attention au sens:
  - si f prend un entier naturel, je ne peux pas (forcément) calculer $f(-3)$.
  - si j'ai besoin d'un poisson, je ne peux pas (forcément) prendre un serpent.
- Dans les signatures des fonctions: + général pour les paramètres, + particulier pour le résultat.
- Héritage dans les langages objets (11).

Grammaires d'expressions

- Compilation: domaine de l'informatique qui s'intéresse à la traduction d'un langage dans un autre.
- Fondement de la programmation: traduction d'un langage "compréhensible" (Python) en langage machine.
  - point de détail: Python est interprété et non compilé.
- Analyse lexicale: séparation du code en jetons.
  - `math.sqrt(3 + 4)` $\rightarrow$ reconnaitre `sqrt`, 3, 4, l'opérateur +, les parenthèses.
- Analyse syntaxique: organisation des jetons en arbre syntaxique.
  -



Grammaires d'expressions (II)

- Les Grammaires permettent d'exprimer le code reconnaissable par le compilateur/l'interprêteur.
- Définition à l'aide de "graines" `S ::= E1 | E2 | ...  | EN`
  - formellement, point fixe d'une fonction (théorème de Knaster-Tarski).
- Grammaire des entiers `N ::= 0 | Succ(N)`
- Grammaire de l'arithmétique `N ::= 0 | Succ(N) | Plus(N,N) | Sous(N,N) | Mult(N,N)`
- Grammaire de la Carte de référence.

Effets de bords

Définition

Un effet de bord est une instruction d'une fonction qui modifie un état (la mémoire, l'affichage) autre que la valeur de retour de la fonction.

- souvent son interprétation n'a pas d'effet direct sur le calcul.

- Affichage: `print` est un effet de bord, elle affiche sur la sortie standard.
- la modification de fichiers (''disque dur'') est un effet de bord.
- Nécessaires, mais difficile à analyser.
  - idempotence des fonctions ?
- `print` fait un effet de bord: affichage à l'écran
  - utile pour connaître les valeurs intermédiaires des variables.

Valeurs Intermédiaires

```
def essai_var3(x : int) -> int:

    n : int = 0
    print("la valeur de n est:", format(n))

    m : int = x
    print("la valeur de n est:", format(n), "la valeur de m est:", format(m))

    n = m + x
    print("la valeur de n est:", format(n), "la valeur de m est:", format(m))
    m = n + 1
    print("la valeur de n est:", format(n), "la valeur de m est:", format(m))
    n = m + x
    print("la valeur de n est:", format(n), "la valeur de m est:", format(m))
    return n
```

- A utiliser en TME.

Fonctions partielles

- primitive `print`:
  - utilisation courante: afficher des chaînes de caractères.
  - peut contenir des expressions de différents types.
  - la valeur de retour de `print` est

Fonctions partielles

- primitive `print`:
  - utilisation courante: afficher des chaînes de caractères.
  - peut contenir des expressions de différents types.
  - la valeur de retour de `print` est Rien

Fonctions partielles

- primitive **`print`**:
  - utilisation courante: afficher des chaînes de caractères.
  - peut contenir des expressions de différents types.
  - la valeur de retour de **`print`** est Rien (en `Python`: `None`).
  - le type de `None` est

Fonctions partielles

- primitive **print**:
  - utilisation courante: afficher des chaînes de caractères.
  - peut contenir des expressions de différents types.
  - la valeur de retour de **print** est Rien (en `Python`: `None`).
  - le type de `None` est `None`:
- Fonctions qui n'ont pas de valeur de retour:

```
def affiche_trois_fois(n : int) -> None:
    print(n)
    print(n)
    print(n)

assert affiche_trois_fois(10) == None
```

  - Est-ce vraiment des fonctions ?
  - Plus tard dans l'UE, types optionnels
    - renvoyer soit un entier (quand ça "marche"), soit rien (quand ce n'est pas possible)

Différence entre `print` et `return`

- `print` est une instruction qui affiche la valeur d'une expression sur la sortie standard.
- `return` renvoie la valeur de son argument à l'appelant.
  - Si l'appelant est le *top-level* de `mrpython`, il affiche la valeur qu'il reçoit.
  - Si l'appelant est une expression, il utilise cette valeur.

```
def h2(x : int) -> int:
    return x + 1

def h3(x : int) -> None:
    print(x + 1)
```

Comparer les expressions `1 + 2 * h2(10)` et `1 + 2 * h3(10)`

Variables Globales

- On peut affecter des variables en dehors des fonctions ("globales").
  - elle doivent être déclarées.
- Ces variables ne sont pas accessibles dans les fonctions.
- Ces variables ne sont pas modifiables.
- Ces variables sont, en fait, des constantes.
- Utiles pour les tests et les essais.
  - surtout avec des structures de données (cours 05-10).

```
nombre : int = 42

def increm(x : int) -> int:
    return x + 1

assert increm(nombre) == 43

def ajoute_n(x : int) -> int:
    return x + nombre # ERREUR

nombre = nombre + 1 # ERREUR
```

SORBONNE UNIVERSITÉ CRÉATEURS DE FUTURS DEPUIS 1257

Modèle mémoire

- Mémoire est un espace indicé:
  - chaque "tiroir" a une taille et une adresse.
- une variable, c'est un nom pour l'adresse d'un tiroir,
  - une table de symboles lie noms et adresses.
- deux "zones" de mémoire:
  - le tas: où vivent les variables globales, les données, les objets, les fonctions (le code),
  - la pile: qui sert à l'execution de fonction.
    - contient les variables locales et les arguments,
    - durée de vie limitée,
    - cas des fonctions qui appellent d'autres fonctions
- En LU1IN002: modèle mémoire formel.

Décidabilité

Définition

Un problème de décision est décidable quand il existe un algorithme pour le résoudre.

- Problème de décision: résultat booléen.
- Solution: un unique algorithme qui marche dans tous les cas particuliers.
- Primalité d'un entier.
  - décider si un entier est premier $\mathtt{div}(n) = \{1, n\}$
  - entrée: un entier, résultat: booléen.
  - algorithme: crible d'Eratosthene (par exemple)

Décidabilité informatique vs. Décidabilité logique

- Décidabilité logique: une formule est décidable (par rapport a un système logique) quand il existe une preuve (dans le système logique) de sa vérité ou de sa fausseté.
- En fait c'est pareil (Curry-Howard).

Indécidabilité

- Il existe des problèmes indécidables.
  - des problèmes qu'aucun algorithme ne peut résoudre.

Indécidabilité

- Il existe des problèmes indécidables.
  - des problèmes qu'aucun algorithme ne peut résoudre.

Théorème: Incomplétude de Gödel

Tout système logique un peu intéressant contient au moins une formule indécidable.

- "un peu intéressant": contient l'arithmétique de Peano $(0, \mathtt{S}, +, .)$
- Logique $\rightarrow$ Informatique: il existe des problèmes indécidables dès que l'expressivité est suffisante.
- Utilisée (de manière discutable) en philosophie (cf. Debray, Bouveresse)
- Exemple: Correspondance de Post
  - Instance: dominos, chacun en quantité illimitée: $\left(\frac{a}{baa}\right)\left(\frac{ab}{aa}\right)\left(\frac{bba}{bb}\right)$
  - Question: existe t-il une suite (finie) de dominos telle que le mot lu au-dessus est le même que le mot lu en dessous ?

Indécidabilité

- Il existe des problèmes indécidables.
  - des problèmes qu'aucun algorithme ne peut résoudre.

Théorème: Incomplétude de Gödel

Tout système logique un peu intéressant contient au moins une formule indécidable.

- "un peu intéressant": contient l'arithmétique de Peano $(0, \mathtt{S}, +, .)$
- Logique $\rightarrow$ Informatique: il existe des problèmes indécidables dès que l'expressivité est suffisante.
- Utilisée (de manière discutable) en philosophie (cf. Debray, Bouveresse)
- Exemple: Correspondance de Post
  - Instance: dominos, chacun en quantité illimitée: $\left(\frac{a}{baa}\right)\left(\frac{ab}{aa}\right)\left(\frac{bba}{bb}\right)$
  - Question: existe t-il une suite (finie) de dominos telle que le mot lu au-dessus est le même que le mot lu en dessous ?
  - Ici: $\left(\frac{bba}{bb}\right)\left(\frac{ab}{aa}\right)\left(\frac{bba}{bb}\right)\left(\frac{a}{baa}\right)$, mot *bbaabbbaa*
  - Il n'existe pas d'algorithme qui prend en entrée un jeu de domino et décide la question.
    - PCP est indécidable.

Terminaison

- Une boucle s'arrête quand sa condition est fausse.
  - peut-on être sur qu'elle sera forcément fausse au bout d'un certain temps?

```python
def infini() -> int:
    """compte pendant l'eternite et renvoie 1 ensuite  """

    i : int = 0 # compteur

    while True:
        i = i + 1
    return 1
```

-

```python
def somme_entiers2(n : int) -> int:
    """retourne la somme des n premiers entiers naturels.  """
    i : int = 1 # entier courant, en commencant par 1
    s : int = 0 # la somme cumulee
    while i <= n:
        s = s + i
        i = i - 1
    return s
```

- Peut-on détecter les programmes divergents ?
- La Terminaison est t-elle décidable ?

Problème de l'arrêt

- Supposons qu'on a la fonction (non-typée) suivante:

```
def arret(fonc, argu ):
    """renvoie True si l'appel de fonction fonc avec l'argument argu termine, False sinon."""
```

- On définit alors:

```
def diago(f):
  i = 0
  if arret(f,f):
      while True:
          i = i + 1
  else:
      return i
```

- Que dire de `diago(diago)` ?

Problème de l'arrêt

- Supposons qu'on a la fonction (non-typée) suivante:

```python
def arret(fonc, argu ):
    """renvoie True si l'appel de fonction fonc avec l'argument argu termine, False sinon."""
```

- On définit alors:

```python
def diago(f):
    i = 0
    if arret(f,f):
        while True:
            i = i + 1
    else:
        return i
```

- Que dire de `diago(diago)` ?
  - si `diago(diago)` s'arrête, c'est que `arret(diago,diago)` vaut `False`. Contradiction !
  - si `diago(diago)` ne s'arrête pas, c'est que `arret(diago,diago)` vaut `True`. Contradiction !
  - On a montré par l'absurde, qu'il n'existe pas de fonction `arret`.
- La terminaison d'un programme est indécidable.
  - énormes conséquences pour l'informatique.

Comparer des programmess

Question fondamentale

Qu'est-ce qu'un bon programme ?

et aussi "Qu'est ce qu'un meilleur programme ?"

Propriétés

- Correction: Est-ce que le programme calcule la bonne fonction ?
- Terminaison: Est-ce que le programme finit toujours pas renvoyer une valeur ?
- Efficacité: Est-ce que le programme est rapide et économe en mémoire ?

Correction

Définition

Un programme `f` est *correct* vis à vis d'une fonction $\mathcal{F}$, quand à chaque calcul `f(x)` pour `x` satisfaisant les hypothèses, si le programme renvoie `v`, alors $\mathcal{F}(x) = v$ (avec (`x`,`v`) représentations de (x, v)).

```python
def somme_entiers(n : int) -> int:
    """ Precondition : n >= 0
    renvoie la somme des entiers jusque n inclus"""

    s : int = 0

    i : int = 0

    while i <= n:
        s = s + i
        i = i + 1

    return i
```

```python
def somme_entiers(n : int) -> int:
    """ Precondition : n >= 0
    renvoie la somme des entiers jusque n inclus"""

    s : int = 0

    i : int = 0

    while i < n:
        s = s + i
        i = i + 1

    return s
```

- Propriété indécidable.
- Analyses statiques des programmes pour les vérifier.
  - types (par exemple la signature), modèle.
- Tests, Simulations: pas de preuve, peut convaincre.

SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS DEPUIS 1257

Terminaison

Définition

Un programme `f` *termine sur l'entrée* `e` quand l'exécution de `f(e)` finit par s'arrêter. Il *termine* quand il termine sur toutes ses entrées qui satisfont ses hypothèses.

```python
def somme_entiers(n : int) -> int:
    """ Precondition : n >= 0
    renvoie la somme des entiers jusque n inclus"""

    s : int = 0

    i : int = 0

    while i <= n:
        s = s + i
        i = i - 1

    return s
```

```python
def somme_entiers(n : int) -> int:
    """ Precondition : n >= 0
    renvoie la somme des entiers jusque n inclus"""

    s : int = 0

    i : int = 0

    while i <= i:
        s = s + i
        i = i + 1

    return s
```

- Propriété indécidable.
- Analyses statiques.
    - types du λ-calcul, analyses de boucles.
- Terminaison à la volée (stopper les calculs trop longs).

Efficacité

Définition

Un programme `f` est *plus efficace en moyenne* qu'un programme `g`:

- `f` et `g` calculent la même fonction mathématique.
- sur toutes les entrées d'une même taille, en moyenne, `f` utilise moins d'opérations élémentaires que `g`.

- entrée de même taille ?
- opérations élémentaires ?
  - affectations, comparaisons, multiplications, ...
- complexité en espace (utilisation de mémoire)

```
def somme_entiers(n : int) -> int:
    """ Precondition : n >= 0
    renvoie la somme des entiers jusque n inclus"""

    s : int = 0
    i : int = 0
    while i <= n:
        s = s + i
        i = i + 1
    return s
```

```
def somme_entiers(n : int) -> int:
    """ Precondition : n >= 0
    renvoie la somme des entiers jusque n inclus"""

    return n * (n + 1) / 2
```

Efficacité (II)

Définition

Un programme `f` est *plus efficace dans le pire des cas* que `g`:

- `f` et `g` calculent la même fonction mathématique.
- sur toutes les entrées d'une même taille, dans le pire des cas, `f` utilise moins d'opérations élémentaires que `g`.

```
def mention(m : int) -> str:
    """ Precondition : m >= 0 and m <= 20
    renvoie la mention du bac associee a la moyenne m"""

    if m <= 10:
        return ''Elimine''
    else:
        if m <= 12:
            return ''Passable''
        else:
            if m <= 14:
                return ''Assez Bien''
            else:
                if m <= 16:
                    return ''Bien''
                else:
                    return ''Tres bien''
```

```
def mention(m : int) -> str:
    """ Precondition : m >= 0 and m <= 20
    renvoie la mention du bac associee a la moyenne m'

    if m <= 14:
        if m <= 12:
            if m <= 10:
                return ''Elimine''
            else:
                return ''Passable''
        else:
            return ''Assez Bien''
    else:
        if m <= 16:
            return ''Bien''
        else:
            return ''Tres bien''
```

Correction

- On veut étudier la correction d'une fonction *Python 101*.
- On peut distinguer deux choses:
  - la correction de l'algorithme,
  - la correction de l'implémentation.

```
def aire_triangle(a : float, b : float, c : float):
    """ Precondition : (a>0) and (b>0) and (c>0)
        Precondition : les cotes a, b et c definissent bien un triangle.

    retourne l'aire du triangle dont les cotes sont de longueur a, b, et c."""

    s : float = (a + b + c) / 2

    return math.sqrt(s * (s - a) * (s - b) * (s - c))
```

- Cas de `aire_triangle`:
  - algorithme:
    - preuve mathématique de la formule.
  - implémentation:
    - on suppose l'interpréteur correct (ce n'est pas toujours le cas),
    - on vérifie que l'expression calcule bien la formule.
- Fonction avec boucle ?

Correction et boucles

```python
def puissance(x : float, n : int) -> float:
    """ Precondition : n>=0
    retourne la valeur de x eleve a la puissance n."""

    res : float = 1 # valeur de x^0

    i : int = 1 # compteur

    while i != n + 1:
        res = res * x
        i = i + 1
    return res
```

Simulation `puissance(2,5)`

Correction et boucles

```python
def puissance(x : float, n : int) -> float:
    """ Precondition : n>=0
    retourne la valeur de x eleve a la puissance n."""

    res : float = 1 # valeur de x^0

    i : int = 1 # compteur

    while i != n + 1:
        res = res * x
        i = i + 1
    return res
```

Simulation `puissance(2,5)`

tour de boucle	variable res	variable i
entrée	1	1
1	2	2
2	4	3
3	8	4
4	16	5
5 (sortie)	32	6

Correction et boucles

```python
def puissance(x : float, n : int) -> float:
    """ Precondition : n>=0
    retourne la valeur de x eleve a la puissance n."""

    res : float = 1 # valeur de x^0

    i : int = 1 # compteur

    while i != n + 1:
        res = res * x
        i = i + 1
    return res
```

Simulation `puissance(2,5)`

tour de boucle	variable res	variable i
entrée	1	1
1	2	2
2	4	3
3	8	4
4	16	5
5 (sortie)	32	6

La fonction est-elle correcte ?

Invariant

Malaise

- la simulation nous prouve formellement que `puissance` est correcte quand elle est appelée sur les arguments `2, 5`.
- la simulation ne nous dit rien, formellement, sur le cas général.
- la simulation nous suggère informellement une méthode générale:
  - à chaque étape, on voit qu'on multiplie `res` par `x`,
  - on s'arrête après `n` étapes.

Définition

Un *invariant de boucle* est une expression logique:

- Qui est vraie en entrée de boucle.
- Qui est vraie après chaque tour de boucle.

- on ne dit rien de l'invariant au cours du calcul d'une boucle.
- on veut des invariants utiles: pas `i >= 1` ni `True`.
- l'invariant est une expression logique (pas `Python`)

Invariant (II)

Qu'est ce qu'un invariant utile ?

- il faut que "(Invariant) + (Sortie de boucle) $\Rightarrow$ Correction"

Méthode Standard

1. Comprendre le problème posé.
   - exprimer la correction.
2. Simuler la fonction (plusieurs fois).
   - intuition de ce qui reste vrai à chaque tour de boucle.
   - relation entre les variables et arguments.
3. Expérience et vision mathématique.
   - penser à ce qui entraîne la correction.

Invariant pour `puissance`

Invariant (II)

Qu'est ce qu'un invariant utile ?

- il faut que "(Invariant) + (Sortie de boucle) $\Rightarrow$ Correction"

Méthode Standard

1. Comprendre le problème posé.
   - exprimer la correction.
2. Simuler la fonction (plusieurs fois).
   - intuition de ce qui reste vrai à chaque tour de boucle.
   - relation entre les variables et arguments.
3. Expérience et vision mathématique.
   - penser à ce qui entraîne la correction.

Invariant pour `puissance`

$$\mathtt{x}^{\mathtt{i}-1} = \mathtt{res}$$

Invariant (III)

Simulation avec invariant de `puissance(2, 5)`

Invariant (III)

Simulation avec invariant de `puissance(2, 5)`

tour de boucle	**variable res**	**variable i**	**Invariant** $\mathtt{x}^{\mathtt{i-1}} = \mathtt{res}$
entree	1	1	$2^{1-1} = 1$**(Vrai)**
1	2	2	$2^{2-1} = 2$ **(Vrai)**
2	4	3	$2^{3-1} = 4$ **(Vrai)**
3	8	4	$2^{4-1} = 8$ **(Vrai)**
4	16	5	$2^{5-1} = 16$ **(Vrai)**
5 (sortie)	32	6	$2^{6-1} = 32$ **(Vrai)**

Cas général

On veut montrer que l'invariant reste vrai à chaque tour de boucle:

- Preuve par récurrence:
  - Invariant vrai en entrée.
  - On suppose que l'invariant est vrai au début d'un tour de boucle, on montre qu'il est vrai à la fin du tour de boucle.
  - par récurrence, l'invariant est vrai en sortie de boucle.

Invariant (IV) - Preuve Formelle

Montrons, par récurrence, que $\mathtt{x}^{\mathtt{i-1}} = \mathtt{res}$ est toujours vrai en sortie de boucle:

- On a $\mathtt{x}^{1\,-\,1} = \mathtt{1}$, donc l'invariant est vrai en entrée de boucle.
- On appelle x, n les valeurs de $\mathtt{x}$ et $\mathtt{n}$, on appelle i, r les valeurs de $\mathtt{i}$ et $\mathtt{res}$ au début du tour et i', r' leurs valeurs en fin de tour.
  - Supposons que l'invariant est vrai au début d'un tour. On a $x^{i-1} = r$.
  - en regardantle corps de la boucle, on sait que:
    - $i' = i + 1$,
    - $r' = r * x$,
  - on calcule $r' = r * x = x^{i-1} * x = x^{i} = x^{i'-1}$
  - et l'invariant est vrai à la fin du tour.
- par récurrence, l'invariant est toujours vrai en fin de tour, donc en sortie de boucle (si jamais ça arrive, cf. terminaison).

On a montré que notre invariant est invariant. Est-il utile ?

Invariant (V) - Preuve Formelle

Montrons que "(Invariant) + (Sortie de boucle) $\Rightarrow$ Correction"

- En sortie de boucle, l'invariant est vrai (cf. slide précédent): donc $x^{i-1} = res$ (1)
- En sortie de boucle, la condition de boucle est fausse (par définition): donc $i = n + 1$ (2)
- De (1) et (2) on déduit: $x^n = res$
- La fonction renvoie la valeur de `res`, donc elle est correcte (pour la fonction mathématique "puissance").

Aux Examens de LU1IN001

- Recherche d'invariant: pas vraiment (QCM).
- Preuve d'invariance: non (L2 Info).
  - Test de l'invariant dans une simulation.
- Preuve de correction en sortie de boucle: oui.

Terminaison

- La méthode de correction suppose que la fonction termine (que la sortie de boucle existe).
- Comment prouver qu'une fonction avec un `while` termine ?
  - Pas de preuve générale (indécidabilité).
  - Idée: montrer que quelque chose décroît strictement à chaque tour.
  - Corollaire de *Bolzano-Weierstrass* (caractérisation des espaces métriques compacts): il n'existe pas de suite infinie d'entiers naturels strictement décroissante.

Définition

Un *variant de boucle* est une expression arithmétique:

- Qui est un entier naturel positif en entrée de boucle.
- Qui décroit **strictement** à chaque tour de boucle.
- Qui vaut 0 en sortie de boucle.

Trouver un bon variant:

- même méthode que pour l'invariant,
- intuition de ce qui décroît.

Variant

Variant pour `puissance`:

Variant

Variant pour `puissance`: `n - i` $+ 1$

Simulation de `puissance(2, 5)` avec variant

Variant

Variant pour `puissance`: `n - i` $+ 1$

Simulation de `puissance(2, 5)` avec variant

tour de boucle	**variable res**	**variable i**	**Variant `n - i` + 1**
entree	1	1	$5 - 1 + 1 = 5$
1	2	2	$5 - 2 + 1 = 4$
2	4	3	$5 - 3 + 1 = 3$
3	8	4	$5 - 4 + 1 = 2$
4	16	5	$5 - 5 + 1 = 1$
5 (sortie)	32	6	$5 - 6 + 1 = 0$

Cas général

On veut montrer qu'une expression est un variant:

- Montrer que sa valeur est un entier positif en entrée de boucle.
- Montrer que sa valeur décroit strictement entre le début et la fin d'un tour de boucle.
- Montrer qu'on sort de la boucle quand il vaut 0.

Variant (II) - Preuve formelle

- Montrons que $\texttt{n} - \texttt{i} + 1$ est bien un variant de boucle:
  - Appelons n la valeur de `n` et i_0 la valeur de `i` en entrée de boucle.
  - En entrée l'expression $n - i_0 + 1 = n - 1 + 1$ vaut n qui est un entier positif.
  - Appelons i la valeur de `i` au début d'un tour et i' la valeur de `i` à la fin d'un tour.
    - on sait que $i' = i + 1$,
    - donc $n - i' + 1 = n - (i + 1) + 1 = n - i < n - i + 1$
    - le variant décroit scrictement à chaque tour de boucle.
  - quand le variant vaut 0, on a $n - i + 1 = 0$ soit $i = n + 1$ ce qui correspond à la condition de sortie de boucle.
- Comme la fonction admet un variant de boucle, elle termine.

Aux Examens de LU1IN001

- Recherche de variant: oui.
- Preuve de terminaison: non (simulation).
  - Test du variant sur une simulation.

Efficacité

Points à examiner en LU1IN001

- Calculs redondants.
- Raccourcis logiques.
- Algorithme plus efficace.

Contexte

- Conjecture empirique de Moore: *la densité des transistors double tous les deux ans*.
  - croissance exponentielle de la puissance de calcul
- Taille des données croît aussi (graphismes).
- Méthodes basées sur la non-efficacité des programmes: cryptographie.
- Classes de complexité des fonctions mathématiques: $\mathcal{P}$, $\mathcal{NP}$, ...
  - $f \in \mathcal{P}$ si il existe un programme `f` et un polynôme P tel que pour tout x `f(x)` calcule $f(x)$ en temps $t \leq P(|x|)$.

Factorisation

```
def aire_triangle(a : float, b : float, c : float) -> float:
    """ Precondition : (a>0) and (b>0) and (c>0)
    Precondition : les cotes a, b, et c definissent
    bien un triangle.

    donne l'aire du triangle dont les cotes
    sont de longueur a, b, et c."""

    return math.sqrt(((a + b + c) / 2)
                     * (((a + b + c) / 2) - a)
                     * (((a + b + c) / 2) - b)
                     * (((a + b + c) / 2) - c))
```

```
def aire_triangle(a : float, b : float, c : float) -> float:
    """ Precondition : (a>0) and (b>0) and (c>0)
    Precondition : les cotes a, b, et c definissent
    bien un triangle.

    retourne l'aire du triangle dont les cotes
    sont de longueur a, b, et c."""

    p : float = (a + b + c) / 2   # demi-perimetre

    return math.sqrt(p * (p - a) * (p - b) * (p - c))
```

- Calcul `(a + b + c) / 2` répété 4 fois.
- Factorisation du calcul grâce à une variable.
- Complexité en espace / Coût d'une affectation.

Sortie anticipée (I)

On veut calculer le plus petit diviseur non-trivial (différent de 1) d'un entier naturel.

```
def plus_petit_diviseur(n : int) -> int:
    """ retourne le plus petit diviseur non-trivial de n"""

    d : int = 0 # pas encore trouve

    m : int = 2

    while m <= n:
        if (d == 0) and (n % m == 0):
            d = m
        m = m + 1
    return d
```

Sortie anticipée (I)

On veut calculer le plus petit diviseur non-trivial (différent de 1) d'un entier naturel.

```
def plus_petit_diviseur(n : int) -> int:
    """ retourne le plus petit diviseur non-trivial de n"""

    d : int = 0 # pas encore trouve

    m : int = 2

    while m <= n:
        if (d == 0) and (n % m == 0):
            d = m
        m = m + 1
    return d
```

```
def plus_petit_diviseur(n : int) -> int:
    """ retourne le plus petit diviseur non-trivial de

    d : int = 0 # pas encore trouve

    m : int = 2

    while (d == 0) and (m <= n):
        if (d == 0) and (n % m == 0):
            d = m
        m = m + 1
    return d
```

- On évite (quand *n* n'est pas premier) beaucoup de tours de boucle inutiles.
- Raffinement de la condition de la boucle.

Sortie anticipée (II)

```
def plus_petit_diviseur(n : int) -> int:
    """ retourne le plus petit diviseur non-trivial de n"""

    d : int = 0 # pas encore trouve

    m : int = 2

    while m <= n:
        if (d == 0) and (n % m == 0):
            return m
        m = m + 1
    return d
```

- Sortie directe de la fonction (avec return).
- **return** fait sortir de la fonction, donc *a fortiori* de toutes les boucles.
- Analyse de terminaison ?

Compter l'efficacité

- Définir ce que l'on compte:
  - dépend du contexte:
    - avion:

Compter l'efficacité

- Définir ce que l'on compte:
  - dépend du contexte:
    - avion: nombre d'opérations numériques.
    - micro-onde:

Compter l'efficacité

- Définir ce que l'on compte:
  - dépend du contexte:
    - avion: nombre d'opérations numériques.
    - micro-onde: espace mémoire.
    - jeux vidéo:

Compter l'efficacité

- Définir ce que l'on compte:
  - dépend du contexte:
    - avion: nombre d'opérations numériques.
    - micro-onde: espace mémoire.
    - jeux vidéo: appels aux fonctions de la la librairie graphique.
    - appli web:

Compter l'efficacité

- Définir ce que l'on compte:
  - dépend du contexte:
    - avion: nombre d'opérations numériques.
    - micro-onde: espace mémoire.
    - jeux vidéo: appels aux fonctions de la la librairie graphique.
    - appli web: envoi et réception de messages asynchrones (AJAX).
  - temps:
    - multiplications, comparaisons, appels à des primitives, ...
  - espace:
    - nombre de variables, appels de fonctions.
- Definir par rapport à quoi on compte:
  - dans quel cas ?
    - en moyenne, pire des cas, ...
  - taille des arguments:
    - longueur d'une liste,
    - taille d'un entier (son $\texttt{log}_2$, correspondant à la mémoire qu'il prend)
- On s'intéresse souvent à la complexité asymptotique:
  - $\frac{n.(n+1)}{2}$ est similaire à $3.n^2 + 180.n$.
  - $n.\texttt{log}_2(n)$ est meilleur que $\frac{n.(n+1)}{2}$.

Efficacité d'algorithmes

```python
def puissance(x : float, n : int) -> float:
    """ Precondition : n >= 0
    retourne x eleve a la puissance n."""

    res : float = 1

    i : int = 1

    while i != (n + 1):
        res = res * x
        i = i + 1
    return res
```

- Autant de multiplications que la valeur n
- Peut-on faire mieux ?

Efficacité d'algorithmes

```
def puissance(x : float, n : int) -> float:
    """ Precondition : n >= 0
    retourne x eleve a la puissance n."""

    res : float = 1

    i : int = 1

    while i != (n + 1):
        res = res * x
        i = i + 1
    return res
```

- Autant de multiplications que la valeur n
- Peut-on faire mieux ?

- $x^n = \begin{cases} x^{\lfloor n/2 \rfloor} * x^{\lfloor n/2 \rfloor} & \text{si } n \text{ est pair} \\ x^{\lfloor n/2 \rfloor} * x^{\lfloor n/2 \rfloor} * x & \text{sinon} \end{cases}$

```
def puissance_rapide(x : float, n ; int) -> float:
    """ Precondition : n >= 0
    donne x eleve a la puissance n."""

    res : float = 1

    acc : float = x

    i : int = n

    while i > 0:
        if i % 2 == 1:
            res = res * acc
        acc = acc * acc
        i = i // 2
    return res
```

- Est-ce que ça calcule vraiment x^n ?

Efficacité d'algorithmes

```python
def puissance(x : float, n : int) -> float:
    """ Precondition : n >= 0
    retourne x eleve a la puissance n."""

    res : float = 1

    i : int = 1

    while i != (n + 1):
        res = res * x
        i = i + 1
    return res
```

- Autant de multiplications que la valeur n
- Peut-on faire mieux ?

- $x^n = \begin{cases} x^{\lfloor n/2 \rfloor} * x^{\lfloor n/2 \rfloor} & \text{si } n \text{ est pair} \\ x^{\lfloor n/2 \rfloor} * x^{\lfloor n/2 \rfloor} * x & \text{sinon} \end{cases}$

```python
def puissance_rapide(x : float, n ; int) -> float:
    """ Precondition : n >= 0
    donne x eleve a la puissance n."""

    res : float = 1

    acc : float = x

    i : int = n

    while i > 0:
        if i % 2 == 1:
            res = res * acc
        acc = acc * acc
        i = i // 2
    return res
```

- Est-ce que ça calcule vraiment x^n ?
- Preuve de correction

Puissance rapide - Terminaison

Variant ?

Puissance rapide - Terminaison

Variant `i`.

Simulation `puissance_rapide(2,10)` avec variant

Puissance rapide - Terminaison

Variant `i`.

Simulation `puissance_rapide(2,10)` avec variant

tour de boucle	**variable res**	**variable acc**	**Variant `i`**
entree	1	2	10
1	1	4	5
2	4	16	2
3	4	256	1
4 (sortie)	1024	65536	0

Preuve Formelle

- i_0 valeur initiale de `i` vaut n valeur de `n`.
- i valeur de `i` en début de boucle, i' valeur en fin de boucle:
  - $i' = \lfloor i/2 \rfloor < i$
- quand $i = 0$ on sort de la boucle (condition fausse)

Ainsi, `puissance_rapide` termine.

Puissance rapide - Correction

Invariant ?

Puissance rapide - Correction

Invariant $\texttt{res} = \frac{x^n}{acc^i}$.

Puissance rapide - Correction

Invariant $\texttt{res} = \frac{x^n}{acc^i}$.

tour de boucle	variable res	variable acc	variable i	Invariant $\texttt{res} = \frac{x^n}{acc^i}$
entree	1	2	10	$1 = \frac{1024}{2^{10}}$ **(Vrai)**
1	1	4	5	$1 = \frac{1024}{4^5}$ **(Vrai)**
2	4	16	2	$4 = \frac{1024}{16^2}$ **(Vrai)**
3	4	256	1	$4 = \frac{1024}{256^1}$ **(Vrai)**
4 (sortie)	1024	65536	0	$1024 = \frac{1024}{65536^0}$ **(Vrai)**

- Soit n, x les valeurs de `n`, `x`, invariant vrai en entrée: $1 = \frac{x^n}{x^n}$.
- Soit i, a, r les valeurs de `i`, `acc`, `res` en début de tour et i', a', r' leurs valeurs en fin de tour.
  - Supposons l'invariant vrai en début de tour: $r = \frac{x^n}{a^i}$
  - Si $i = 2 * p$, alors $i' = p$, $r' = r$, $a' = a * a$, on a $r' = \frac{x^n}{a^{2p}} = \frac{x^n}{(a*a)^p} = \frac{x^n}{a'^{i'}}$ et l'invariant reste vrai.
  - Si $i = 2 * p + 1$, alors $i' = p$, $r' = r * a$, $a' = a * a$, on a $r' = a * \frac{x^n}{a^{2p+1}} = a * \frac{x^n}{(a*a)^p * a} = \frac{x^n}{a'^{i'}}$ et l'invariant reste vrai.
- Par récurrence, invariant vrai en sortie, quand $\texttt{i} = 0$ soit $\texttt{res} = \frac{x^n}{\texttt{acc}^0}$
  La fonction renvoie bien x^n.

Complexité

```python
def puissance(x : float, n : int) -> float:
    """ Precondition : n >= 0
    donne x eleve a la puissance n."""

    res : Number = 1

    i : int = 1

    while i != (n + 1):
        res = res * x
        i = i + 1
    return res
```

- Autant de multiplications que la valeur `n`.
- On fait une de multiplication à chaque tour de boucle.
- On fait n tours de boucle.
- Dans tous les cas, $n \times 1 = n$ multiplications.

Complexité

```python
def puissance(x : float, n : int) -> float:
    """ Precondition : n >= 0
    donne x eleve a la puissance n."""

    res : Number = 1

    i : int = 1

    while i != (n + 1):
        res = res * x
        i = i + 1
    return res
```

- Autant de multiplications que la valeur n.
- On fait une de multiplication à chaque tour de boucle.
- On fait n tours de boucle.
- Dans tous les cas, $n \times 1 = n$ multiplications.

```python
def puissance_rapide(x : float, n : int):
    """ Precondition : n >= 0
    donne x eleve a la puissance n."""

    # res : Number
    res = 1

    # val : Number
    acc = x

    # i : int
    i = n

    while i > 0:
        if i % 2 == 1:
            res = res * acc
        acc = acc * acc
        i = i // 2
    return res
```

- Combien de multiplications ?

Complexité

```python
def puissance(x : float, n : int) -> float:
    """ Precondition : n >= 0
    donne x eleve a la puissance n."""

    res : Number = 1

    i : int = 1

    while i != (n + 1):
        res = res * x
        i = i + 1
    return res
```

- Autant de multiplications que la valeur n.
- On fait une de multiplication à chaque tour de boucle.
- On fait n tours de boucle.
- Dans tous les cas, $n \times 1 = n$ multiplications.

```python
def puissance_rapide(x : float, n : int):
    """ Precondition : n >= 0
    donne x eleve a la puissance n."""

    # res : Number
    res = 1

    # val : Number
    acc = x

    # i : int
    i = n

    while i > 0:
        if i % 2 == 1:
            res = res * acc
        acc = acc * acc
        i = i // 2
    return res
```

- Combien de multiplications ?
- On fait une ou deux multiplications à chaque tour de boucle.
- On fait

Complexité

```
def puissance(x : float, n : int) -> float:
    """ Precondition : n >= 0
    donne x eleve a la puissance n."""

    res : Number = 1

    i : int = 1

    while i != (n + 1):
        res = res * x
        i = i + 1
    return res
```

- Autant de multiplications que la valeur `n`.
- On fait une de multiplication à chaque tour de boucle.
- On fait n tours de boucle.
- Dans tous les cas, $n \times 1 = n$ multiplications.

```
def puissance_rapide(x : float, n : int):
    """ Precondition : n >= 0
    donne x eleve a la puissance n."""

    # res : Number
    res = 1

    # val : Number
    acc = x

    # i : int
    i = n

    while i > 0:
        if i % 2 == 1:
            res = res * acc
        acc = acc * acc
        i = i // 2
    return res
```

- Combien de multiplications ?
- On fait une ou deux multiplications à chaque tour de boucle.
- On fait $\texttt{log}_2(n)$ tours de boucle.
- Dans le pire des cas, $2.\texttt{log}_2(n)$ multiplications.

Tracer la complexité

- On peut stocker le nombre d'opérations élémentaires que fait une fonction dans une variable.
  - et l'afficher avec **print** juste avant la sortie de la fonction.

```python
def puissance_complex(x : float, n : int):
    """ Precondition : n>= 0 """

    res : float = 1

    i : int = 1

    nb_mult : int = 0

    while i <= n:
        res = res * x
        nb_mult = nb_mult + 1
        i = i + 1

    print("Nombre de multiplications =", nb_mult)

    return res
```

```python
def puissance_rapide_complex(x : float, n : int) -> int:
    """ Precondition : n >= 0 """

    res : float = 1

    acc : float = x

    i : int = n

    nb_mult : int = 0

    while i > 0:
        if i % 2 == 1:
            res = res * acc
            nb_mult = nb_mult + 1

        acc = acc * acc
        nb_mult = nb_mult + 1
        i = i // 2

    print("Nombre de multiplications = ",nb_mult)
    return res
```

Récursion

- Autre approche (que la boucle) pour les calculs répétitifs.
- Fonction qui s'utilise elle-même.
- Dans le corps d'une fonction `f` on peut appliquer une fonction `g`
  - exemple: `divise` dans `est_premier`
  - et quand `g` = `f` ?

Récursion

- Autre approche (que la boucle) pour les calculs répétitifs.
- Fonction qui s'utilise elle-même.
- Dans le corps d'une fonction `f` on peut appliquer une fonction `g`
  - exemple: `divise` dans `est_premier`
  - et quand `g` = `f` ?

```
def factorielle(n : int) -> int:
    """ Precondition : n >= 0
    donne la factorielle de n."""

    if n <= 1:
        return 1
    else:
        return n * factorielle(n - 1)
```

- Proche de la définition mathématique.
- Correction ? Terminaison ?
- Pas particulièrement efficace (récursivité terminale).
- Pas au programme de LU1IN001 (cf. LU2IN019, LI101).

Exercice: Racine Cubique

```python
def racine_cubique_entiere(n : int) -> int:
    """ Precondition : n >= 0 """

    racine : int  = 0
    while (racine ** 3) <= n:
        racine = racine + 1
    return racine - 1
```

- Variant ?

Exercice: Racine Cubique

```python
def racine_cubique_entiere(n : int) -> int:
    """ Precondition : n >= 0 """

    racine : int  = 0
    while (racine ** 3) <= n:
        racine = racine + 1
    return racine - 1
```

- Variant $\mathtt{max}(n - racine^3, 0)$,
- Invariant ?

SORBONNE UNIVERSITÉ CRÉATEURS DE FUTURS DEPUIS 1257

Exercice: Racine Cubique

```
def racine_cubique_entiere(n : int) -> int:
    """ Precondition : n >= 0 """

    racine : int  = 0
    while (racine ** 3) <= n:
        racine = racine + 1
    return racine - 1
```

- Variant $\mathtt{max}(n - racine^3, 0)$,
- Invariant "$(racine - 1)^3 \leq n$",
- Correction ?

Exercice: Racine Cubique

```python
def racine_cubique_entiere(n : int) -> int:
    """ Precondition : n >= 0 """

    racine : int  = 0
    while (racine ** 3) <= n:
        racine = racine + 1
    return racine - 1
```

- Variant $\mathtt{max}(n - racine^3, 0)$,
- Invariant "$(racine - 1)^3 \leq n$",
- Correction en fin de boucle, $racine^3 > n$ donc $(racine - 1)^3 \leq n < racine^3$.
- Complexité (en comparaison, pire des cas) ?

Exercice: Racine Cubique

```python
def racine_cubique_entiere(n : int) -> int:
    """ Precondition : n >= 0 """

    racine : int  = 0
    while (racine ** 3) <= n:
        racine = racine + 1
    return racine - 1
```

- Variant $\mathtt{max}(n - racine^3, 0)$,
- Invariant "$(racine - 1)^3 \leq n$",
- Correction en fin de boucle, $racine^3 > n$ donc $(racine - 1)^3 \leq n < racine^3$.
- Complexité (en comparaison, pire des cas)
  - pire des cas: indifférent.
  - complexité: $n^{\frac{1}{3}}$

Exercice: PGCD

```
def pgcd(n : int, m : int) -> int:
    """ Precondition : n >= m>= 0 """

    d : int = n

    r : int = m

    # temp : int
    temp = 0

    while r != 0:
        temp = d % r
        d = r
        r = temp

    return d
```

- Variant ?

Exercice: PGCD

```
def pgcd(n : int, m : int) -> int:
    """ Precondition : n >= m>= 0 """

    d : int = n

    r : int = m

    # temp : int
    temp = 0

    while r != 0:
        temp = d % r
        d = r
        r = temp

    return d
```

- Variant r,
- Invariant ?

Exercice: PGCD

```
def pgcd(n : int, m : int) -> int:
    """ Precondition : n >= m>= 0 """

    d : int = n

    r : int = m

    # temp : int
    temp = 0

    while r != 0:
        temp = d % r
        d = r
        r = temp

    return d
```

- Variant r,
- Invariant "$\texttt{div}(\texttt{n}, \texttt{m}) = \texttt{div}(\texttt{d}, \texttt{r})$",
- Correction ?

Exercice: PGCD

```
def pgcd(n : int, m : int) -> int:
    """ Precondition : n >= m>= 0 """

    d : int = n

    r : int = m

    # temp : int
    temp = 0

    while r != 0:
        temp = d % r
        d = r
        r = temp

    return d
```

- Variant r,
- Invariant "$\mathtt{div}(\mathtt{n}, \mathtt{m}) = \mathtt{div}(\mathtt{d}, \mathtt{r})$",
- Correction en fin de boucle, $r = 0$ donc $\mathtt{div}(\mathtt{n}, \mathtt{m}) = \mathtt{div}(\mathtt{d}, 0) = \mathtt{div}(\mathtt{d})$, puis passage au $\mathtt{max}$.
- Complexité (en modulo, pire des cas) ?

Exercice: PGCD

```python
def pgcd(n : int, m : int) -> int:
    """ Precondition : n >= m>= 0 """

    d : int = n

    r : int = m

    # temp : int
    temp = 0

    while r != 0:
        temp = d % r
        d = r
        r = temp

    return d
```

- Variant r,
- Invariant "$\texttt{div}(\texttt{n}, \texttt{m}) = \texttt{div}(\texttt{d}, \texttt{r})$",
- Correction en fin de boucle, $r = 0$ donc $\texttt{div}(\texttt{n}, \texttt{m}) = \texttt{div}(\texttt{d}, 0) = \texttt{div}(\texttt{d})$, puis passage au $\texttt{max}$.
- Complexité (en modulo, pire des cas)
  - pire des cas: nombres consécutifs de Fibonacci.
  - complexité: inférieure à $k.\texttt{log}_2(m) + 1$ avec $k \approx 2.0781$

Correction de programmes

- Domaine Systèmes Embarqués: comment être sûr que l'avion ne va pas s'écraser ? (ou que le missile va s'écraser)
  - Le problème de la correction d'un programme est indécidable.
  - Deux approches:
    - le Test: soumettre le programme à des jeux de test couvrant.
    - la Vérification: regarder le code du programme, prouver la correction.
  - Avantages
    - Test: peu coûteux, pas d'accès au code.
    - Vérification: à faire une seule fois, garantie formelle.
  - Inconvénients
    - Test: pas de garantie (cas improbables mais possibles ?).
    - Vérification: très difficile.
  - En pratique: beaucoup de test, mais de plus en plus de vérification.

Vérification

- Systèmes de Types: des types garantissent certaines propriétés: correction, terminaison, absence de blocage, ...
  - exemple: typeur de *MrPython*
- Modèles: considérer l'espace d'état (les configurations possibles de la mémoire) et le diviser en zone.
  - montrer que certaines zones sont inatteignables.

Vérification automatique

- Des assistants de preuves (Coq, Isabelle) peuvent prouver des programmes.
- Code basé sur les isomorphismes de Curry-Howard.

formules	$\leftrightarrow$	types
preuves	$\leftrightarrow$	programmes

- Utile pour vérifier les interpréteurs et compilateurs des langages de programmation.

Définition

Un modèle de calcul est un langage formel de termes liés par une relation de réduction.

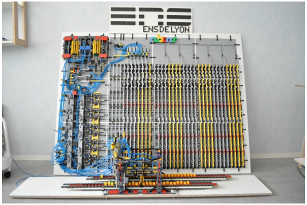
- syntaxe: langage de termes, construit depuis des briques de base (Python 101: instructions et expressions, "arbres" du Cours01)
- sémantique: règles de réduction (Python 101: principes d'interprétation et d'évaluation)

Exemples de modèles de calcul

- Fonctions Récursives (Kleene 1940):
  - fonctions des entiers définies récursivement.
  - syntaxe: définition, successeur, projection, composition, récursion, minimisation.
  - exemple: $\mathtt{Add}(a, b) = \mathbf{R}^2[\mathbf{U}_1{}^1, \mathbf{S}_1{}^3(\mathtt{Succ}, \mathbf{U}_2{}^3)]$
  - sémantique: remplacement en utilisant des égalités.

Modèles de Calcul

- Lambda-Calcul (Church 1930):
  - tout est fonction, calcul = substitution de variables.
  - syntaxe: $M, N ::= x \mid \lambda x.M \mid M\ N$.
  - exemple: $\texttt{Add}(a, b) = \lambda abfe.(a\ f)\ (b\ f\ e)$
  - sémantique: β-reduction: $(\lambda x.M)\ N \to M[N/x]$
  - origine du `lambda` de Python.
- Machine de Turing (Turing 1930):
  - modèle opérationnel de calcul: "ordinateur".
  - syntaxe: un ruban (de 0 et de 1), une tête de lecture, un état.
  - sémantique: règles, en fonction de la position de la tête et de la case du ruban, on modifie (ou non) la case et on se déplace à gauche ou à droite
  - en Lego (par les étudiants de l'ENS de Lyon)



Trois manières différentes de calculer.

Thèse de Church

(Rappel) Expressivité: ensembles des fonctions mathématiques atteignables par un modèle de calcul.

Théorème

Les machines de Turing, le λ-calcul de Church et les fonctions récursives de Kleene:

1. ont la même expressivité,
2. qui est la notion naturelle du calcul.

- partie mathématique (1.): preuves formelles, encodages.
- partie philosophique (2.): notion de "calcul du cerveau humain".
- introduction de la Turing-complétude (Turing-puissance): "être aussi expressif que ces 3 modèles".
  - les langages usuels sont (évidemment) Turing-puissant.
  - notre langage (à quatre instructions) est Turing-puissant.

Conclusion

- boucle: instruction `while`.
- simulation de boucles.
- boucles imbriquées.
- correction (invariant)
- terminaison (variant)
- efficacité
- Culture Générale :
  - Problème de l'arrêt
  - Thèse de Church

Conclusion (II)

TD-TME 03

- Thèmes 03 et 04 du cahier d'exercices.

Activité 03

- Validation

Cours 04 - 04/10/2021

- "ces séquences qui nous gouvernent"