# Informatique pour tous - MP*

# Rappels :
# Preuves de terminaison et de correction
# Complexité temporelle

Florent Pompigne
pompigne@crans.org

Lycée Buffon

année 2017/2018

# Plan

# Terminaison d'une boucle while

Pour démontrer qu'une boucle while termine, on utilise un **variant**. C'est une expression faisant intervenir les variables du programme et qui vérifie les deux conditions suivantes :

# Terminaison d'une boucle while

Pour démontrer qu'une boucle while termine, on utilise un **variant**. C'est une expression faisant intervenir les variables du programme et qui vérifie les deux conditions suivantes :

- Au début de chaque itération, le variant est à valeur dans $\mathbb{N}$.

# Terminaison d'une boucle while

Pour démontrer qu'une boucle while termine, on utilise un **variant**. C'est une expression faisant intervenir les variables du programme et qui vérifie les deux conditions suivantes :

- Au début de chaque itération, le variant est à valeur dans $\mathbb{N}$.
- Entre deux itérations successives de la boucle, la valeur du variant a diminué strictement (d'où le nom de variant).

# Terminaison d'une boucle while

Pour démontrer qu'une boucle while termine, on utilise un **variant**. C'est une expression faisant intervenir les variables du programme et qui vérifie les deux conditions suivantes :

- Au début de chaque itération, le variant est à valeur dans $\mathbb{N}$.
- Entre deux itérations successives de la boucle, la valeur du variant a diminué strictement (d'où le nom de variant).

**Exemple** :

```
def somme(L):
    res = 0
    i = 0
    while i<len(L):
        res = res + L[i]
        i = i+1
    return res
```

# Invariants de boucle

Pour démontrer qu'une boucle while ou for a le comportement attendu, on utilise un **invariant de boucle**. C'est une proposition qui vérifie les conditions suivantes :

# Invariants de boucle

Pour démontrer qu'une boucle while ou for a le comportement attendu, on utilise un **invariant de boucle**. C'est une proposition qui vérifie les conditions suivantes :

- L'invariant est vrai avant d'entrer dans la boucle

# Invariants de boucle

Pour démontrer qu'une boucle while ou for a le comportement attendu, on utilise un **invariant de boucle**. C'est une proposition qui vérifie les conditions suivantes :

- L'invariant est vrai avant d'entrer dans la boucle
- Si l'invariant est vrai au début d'une itération, alors il reste vrai à la fin de cette itération (d'où le nom d'invariant).

# Invariants de boucle

Pour démontrer qu'une boucle while ou for a le comportement attendu, on utilise un **invariant de boucle**. C'est une proposition qui vérifie les conditions suivantes :

- L'invariant est vrai avant d'entrer dans la boucle
- Si l'invariant est vrai au début d'une itération, alors il reste vrai à la fin de cette itération (d'où le nom d'invariant).
- Lorsque la boucle se termine, l'invariant permet de déduire la correction du programme.

# Invariants de boucle

Pour démontrer qu'une boucle while ou for a le comportement attendu, on utilise un **invariant de boucle**. C'est une proposition qui vérifie les conditions suivantes :

- L'invariant est vrai avant d'entrer dans la boucle
- Si l'invariant est vrai au début d'une itération, alors il reste vrai à la fin de cette itération (d'où le nom d'invariant).
- Lorsque la boucle se termine, l'invariant permet de déduire la correction du programme.

**Exemple** :

```
def somme(L):
    res = 0
    i = 0
    while i<len(L):
        res = res + L[i]
        i = i+1
    return res
```

# Plan

# Principe

# Principe

- mesure le coût en temps d'un algorithme.

# Principe

- mesure le coût en temps d'un algorithme.
- se donne en fonction d'un entier caractéristique de ou des entrée(s) (par exemple, la longueur de la liste argument)

# Principe

- mesure le coût en temps d'un algorithme.
- se donne en fonction d'un entier caractéristique de ou des entrée(s) (par exemple, la longueur de la liste argument)
- se calcule dans le pire cas pour une entrée de cette taille.

# Principe

- mesure le coût en temps d'un algorithme.
- se donne en fonction d'un entier caractéristique de ou des entrée(s) (par exemple, la longueur de la liste argument)
- se calcule dans le pire cas pour une entrée de cette taille.
- se calcule asymptotiquement, avec la notation $O(f(n))$. Par exemple, un algorithme qui nécessite $4n^2 + 3n + 18$ opérations est en $O(n^2)$.

# Principe

- mesure le coût en temps d'un algorithme.
- se donne en fonction d'un entier caractéristique de ou des entrée(s) (par exemple, la longueur de la liste argument)
- se calcule dans le pire cas pour une entrée de cette taille.
- se calcule asymptotiquement, avec la notation $O(f(n))$. Par exemple, un algorithme qui nécessite $4n^2 + 3n + 18$ opérations est en $O(n^2)$.
- On considère que les **opérations élémentaires** sont en temps constant (ie en $O(1)$).

# Principe

- mesure le coût en temps d'un algorithme.
- se donne en fonction d'un entier caractéristique de ou des entrée(s) (par exemple, la longueur de la liste argument)
- se calcule dans le pire cas pour une entrée de cette taille.
- se calcule asymptotiquement, avec la notation $O(f(n))$. Par exemple, un algorithme qui nécessite $4n^2 + 3n + 18$ opérations est en $O(n^2)$.
- On considère que les **opérations élémentaires** sont en temps constant (ie en $O(1)$).
- Lorsqu'on enchaîne deux algorithmes, leurs complexités s'ajoutent. Ainsi, un algorithme constitué d'une partie en $O(n^3)$ et d'une autre en $O(n^2)$ est en $O(n^3 + n^2) = O(n^3)$.

# Principe

- mesure le coût en temps d'un algorithme.
- se donne en fonction d'un entier caractéristique de ou des entrée(s) (par exemple, la longueur de la liste argument)
- se calcule dans le pire cas pour une entrée de cette taille.
- se calcule asymptotiquement, avec la notation $O(f(n))$. Par exemple, un algorithme qui nécessite $4n^2 + 3n + 18$ opérations est en $O(n^2)$.
- On considère que les **opérations élémentaires** sont en temps constant (ie en $O(1)$).
- Lorsqu'on enchaîne deux algorithmes, leurs complexités s'ajoutent. Ainsi, un algorithme constitué d'une partie en $O(n^3)$ et d'une autre en $O(n^2)$ est en $O(n^3 + n^2) = O(n^3)$.
- Par conséquent, la complexité d'une boucle est la somme des complexités de chaque itération. Lorsque chaque itération a la même complexité, celle-ci est donc multipliée par le nombre d'itérations.

# Exemples

```
def ???_???????(n):
    for i in range(2, int(math.sqrt(n))+1):
        if n%i==0:
            return False
    return True
```

# Exemples

```
def ???_???????(n):
    for i in range(2, int(math.sqrt(n))+1):
        if n%i==0:
            return False
    return True


def ??????????(A):
    n = len(A)
    for i in range(n):
        for j in range(i):
            A[i][j] , A[j][i] = A[j][i] , A[i][j]
```

# Exercice

Déterminer ce que calcule le programme suivant. Prouver sa terminaison et sa correction. Calculer sa complexité.

```
def ????????????????_??????(k,n):
    res = 1
    while n>0:
        if n%2==1:
            res = res*k
        k = k**2
        n = n//2
    return res
```