Algorithmique

Lycée Jean Moulin : NSI

Introduction

L'informatique est structurée par quatre concepts :

Introduction

L'informatique est structurée par quatre concepts :

- **L'algorithme :**

Introduction

L'informatique est structurée par quatre concepts :

- **L'algorithme :** méthode opérationnelle permettant de résoudre un problème.

Introduction

L'informatique est structurée par quatre concepts :

- **L'algorithme :** méthode opérationnelle permettant de résoudre un problème.
- **La machine :**

Introduction

L'informatique est structurée par quatre concepts :

- **L'algorithme :** méthode opérationnelle permettant de résoudre un problème.
- **La machine :** système physique doté de fonctionnalités.

Introduction

L'informatique est structurée par quatre concepts :

- **L'algorithme :** méthode opérationnelle permettant de résoudre un problème.
- **La machine :** système physique doté de fonctionnalités.
- **Le langage :**

Introduction

L'informatique est structurée par quatre concepts :

- **L'algorithme :** méthode opérationnelle permettant de résoudre un problème.
- **La machine :** système physique doté de fonctionnalités.
- **Le langage :** moyen de communication entre l'informaticien et la machine.

Introduction

L'informatique est structurée par quatre concepts :

- **L'algorithme :** méthode opérationnelle permettant de résoudre un problème.
- **La machine :** système physique doté de fonctionnalités.
- **Le langage :** moyen de communication entre l'informaticien et la machine.
- **L'information :**

Introduction

L'informatique est structurée par quatre concepts :

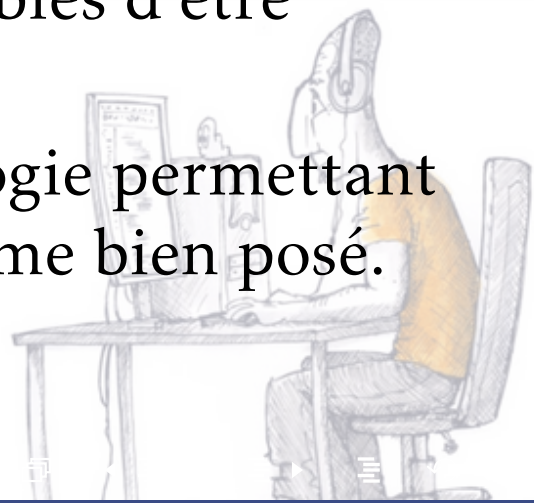
- **L'algorithme :** méthode opérationnelle permettant de résoudre un problème.
- **La machine :** système physique doté de fonctionnalités.
- **Le langage :** moyen de communication entre l'informaticien et la machine.
- **L'information :** données symboliques susceptibles d'être traitées par une machine.

Introduction

L'informatique est structurée par quatre concepts :

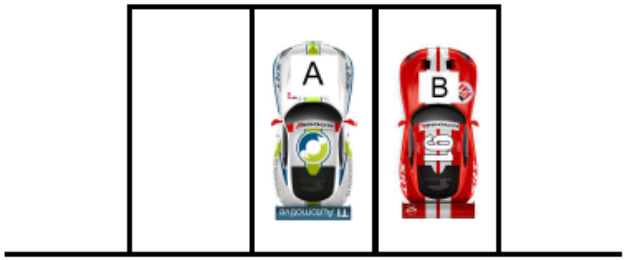
- **L'algorithme :** méthode opérationnelle permettant de résoudre un problème.
- **La machine :** système physique doté de fonctionnalités.
- **Le langage :** moyen de communication entre l'informaticien et la machine.
- **L'information :** données symboliques susceptibles d'être traitées par une machine.

On se propose de mettre en place une méthodologie permettant de mettre sur le papier la résolution d'un problème bien posé.



Poser proprement le problème

Le problème : Permuter deux voitures sur un parking de trois places numérotées P_1, P_2 et P_3, sans gêner la circulation. La voiture A est sur l'emplacement P_2, la voiture B est sur l'emplacement P_3.

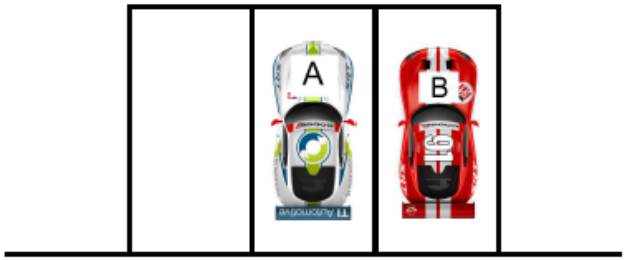


Avant permutation

Après permutation

Poser proprement le problème

Le problème : Permuter deux voitures sur un parking de trois places numérotées P_1, P_2 et P_3, sans gêner la circulation. La voiture A est sur l'emplacement P_2, la voiture B est sur l'emplacement P_3.



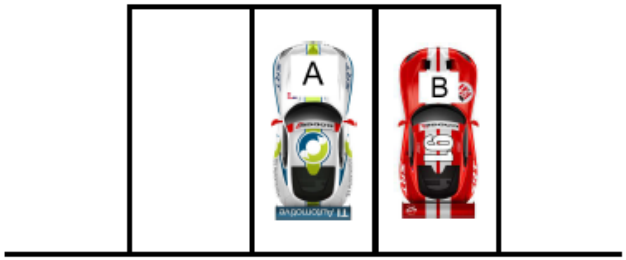
Avant permutation

Après permutation

Un premier algorithme (*naïf*) :

Poser proprement le problème

Le problème : Permuter deux voitures sur un parking de trois places numérotées P_1, P_2 et P_3, sans gêner la circulation. La voiture A est sur l'emplacement P_2, la voiture B est sur l'emplacement P_3.



Avant permutation

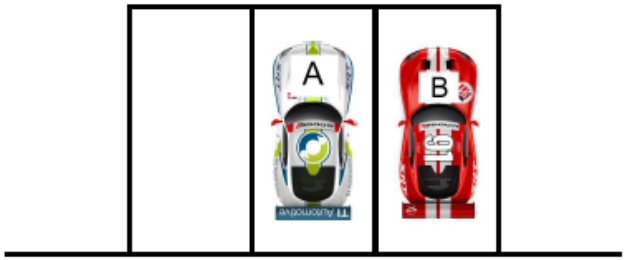
Après permutation

Un premier algorithme (*naïf*) :

- déplacer la voiture B de P_3 à P_1.

Poser proprement le problème

Le problème : Permuter deux voitures sur un parking de trois places numérotées P_1, P_2 et P_3, sans gêner la circulation. La voiture A est sur l'emplacement P_2, la voiture B est sur l'emplacement P_3.



Avant permutation

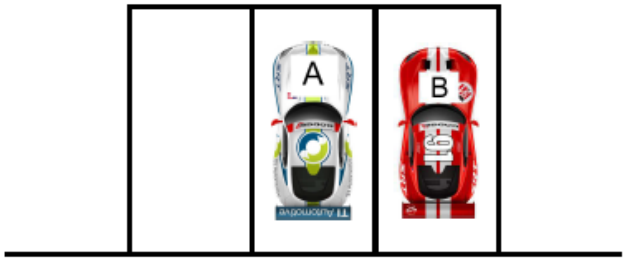
Après permutation

Un premier algorithme (*naïf*) :

- déplacer la voiture B de P_3 à P_1.
- déplacer la voiture A de P_2 à P_3.

Poser proprement le problème

Le problème : Permuter deux voitures sur un parking de trois places numérotées P_1, P_2 et P_3, sans gêner la circulation. La voiture A est sur l'emplacement P_2, la voiture B est sur l'emplacement P_3.



Avant permutation

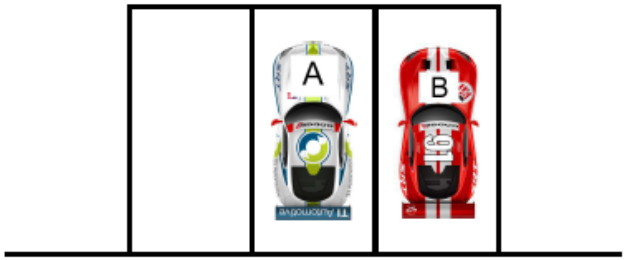
Après permutation

Un premier algorithme (*naïf*) :

- déplacer la voiture B de P_3 à P_1.
- déplacer la voiture A de P_2 à P_3.
- déplacer la voiture B de P_1 à P_2.

Poser proprement le problème

Le problème : Permuter deux voitures sur un parking de trois places numérotées P_1, P_2 et P_3, sans gêner la circulation. La voiture A est sur l'emplacement P_2, la voiture B est sur l'emplacement P_3.



Avant permutation

Après permutation

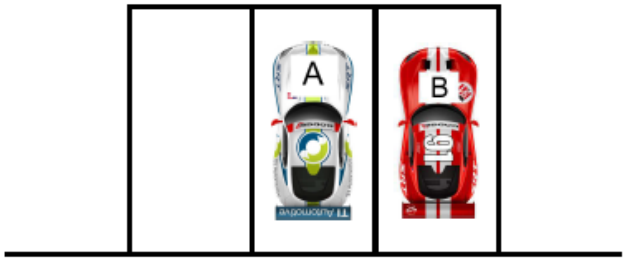
Un premier algorithme (*naïf*) :

- déplacer la voiture B de P_3 à P_1.
- déplacer la voiture A de P_2 à P_3.
- déplacer la voiture B de P_1 à P_2.

Des imprévus

Poser proprement le problème

Le problème : Permuter deux voitures sur un parking de trois places numérotées P_1, P_2 et P_3, sans gêner la circulation. La voiture A est sur l'emplacement P_2, la voiture B est sur l'emplacement P_3.



Avant permutation

Après permutation

Un premier algorithme (*naïf*) :

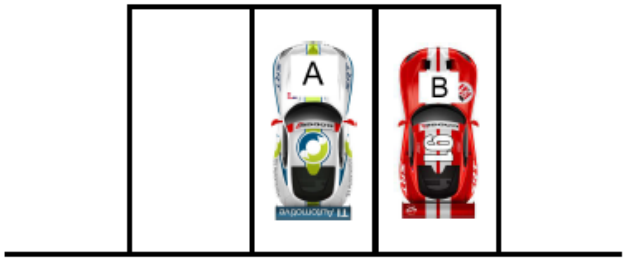
- déplacer la voiture B de P_3 à P_1.
- déplacer la voiture A de P_2 à P_3.
- déplacer la voiture B de P_1 à P_2.

Des imprévus

- P_1 est-il libre ?

Poser proprement le problème

Le problème : Permuter deux voitures sur un parking de trois places numérotées P_1, P_2 et P_3, sans gêner la circulation. La voiture A est sur l'emplacement P_2, la voiture B est sur l'emplacement P_3.



Avant permutation

Après permutation

Un premier algorithme (*naïf*) :

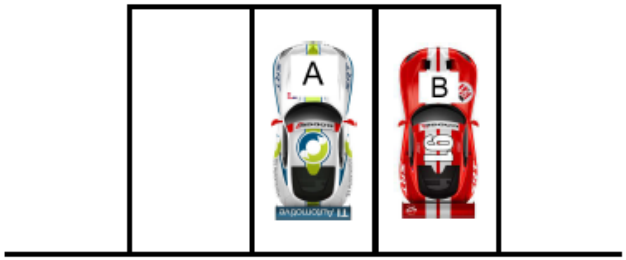
- déplacer la voiture B de P_3 à P_1.
- déplacer la voiture A de P_2 à P_3.
- déplacer la voiture B de P_1 à P_2.

Des imprévus

- P_1 est-il libre ?
- les voitures sont-elles en état de marche ?

Poser proprement le problème

Le problème : Permuter deux voitures sur un parking de trois places numérotées P_1, P_2 et P_3, sans gêner la circulation. La voiture A est sur l'emplacement P_2, la voiture B est sur l'emplacement P_3.



Avant permutation

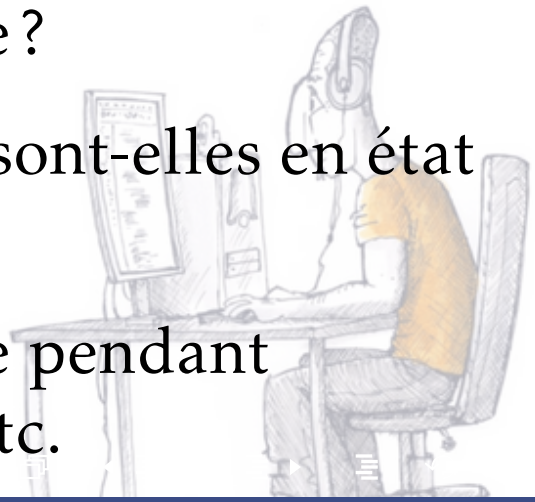
Après permutation

Un premier algorithme (*naïf*) :

- déplacer la voiture B de P_3 à P_1.
- déplacer la voiture A de P_2 à P_3.
- déplacer la voiture B de P_1 à P_2.

Des imprévus

- P_1 est-il libre ?
- les voitures sont-elles en état de marche ?
- P_3 est-il libre pendant l'échange ? etc.



Poser proprement le problème

Le problème est en partie mal posé, il faut préciser les choses :

Poser proprement le problème

Le problème est en partie mal posé, il faut préciser les choses :

- **Données :**

Poser proprement le problème

Le problème est en partie mal posé, il faut préciser les choses :

- **Données :** Parking de trois emplacements, numérotés P_1, P_2 et P_3. Deux voitures en état de marche : A sur P_3 et B sur P_2.

Poser proprement le problème

Le problème est en partie mal posé, il faut préciser les choses :

- **Données :** Parking de trois emplacements, numérotés P_1, P_2 et P_3. Deux voitures en état de marche : A sur P_3 et B sur P_2.
- **Hypothèses générales :**

Poser proprement le problème

Le problème est en partie mal posé, il faut préciser les choses :

- **Données :** Parking de trois emplacements, numérotés P_1, P_2 et P_3. Deux voitures en état de marche : A sur P_3 et B sur P_2.
- **Hypothèses générales :** Un seul conducteur réalise la permutation. Celui-ci a son permis de conduire et les clés des deux voitures. Lorsqu'une voiture est en mouvement, aucune autre voiture ne vient sur le parking. Une éventuelle autre voiture ne reste qu'un temps fini sur un emplacement.

Poser proprement le problème

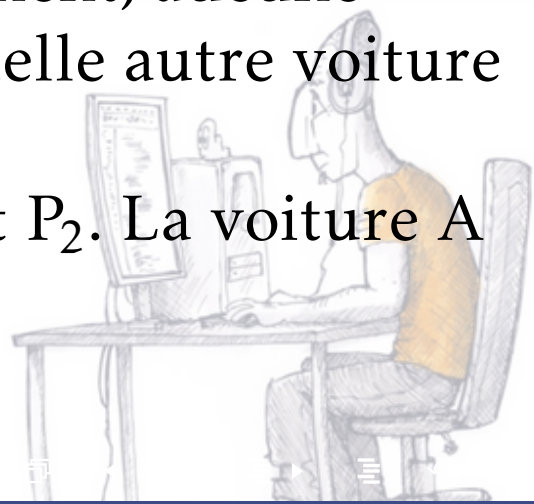
Le problème est en partie mal posé, il faut préciser les choses :

- **Données :** Parking de trois emplacements, numérotés P_1, P_2 et P_3. Deux voitures en état de marche : A sur P_3 et B sur P_2.
- **Hypothèses générales :** Un seul conducteur réalise la permutation. Celui-ci a son permis de conduire et les clés des deux voitures. Lorsqu'une voiture est en mouvement, aucune autre voiture ne vient sur le parking. Une éventuelle autre voiture ne reste qu'un temps fini sur un emplacement.
- **Résultats :**

Poser proprement le problème

Le problème est en partie mal posé, il faut préciser les choses :

- **Données :** Parking de trois emplacements, numérotés P_1, P_2 et P_3. Deux voitures en état de marche : A sur P_3 et B sur P_2.
- **Hypothèses générales :** Un seul conducteur réalise la permutation. Celui-ci a son permis de conduire et les clés des deux voitures. Lorsqu'une voiture est en mouvement, aucune autre voiture ne vient sur le parking. Une éventuelle autre voiture ne reste qu'un temps fini sur un emplacement.
- **Résultats :** La voiture B est sur l'emplacement P_2. La voiture A est sur l'emplacement P_3.



Un algorithme possible

- Aller au volant de la voiture B (emplacement P_3).

Un algorithme possible

- Aller au volant de la voiture B (emplacement P_3).
- Répéter attendre jusqu'à emplacement P_1 libre.

Un algorithme possible

- Aller au volant de la voiture B (emplacement P_3).
- Répéter attendre jusqu'à emplacement P_1 libre.
- Déplacer la voiture B de P_3 à P_1.

Un algorithme possible

- Aller au volant de la voiture B (emplacement P_3).
- Répéter attendre jusqu'à emplacement P_1 libre.
- Déplacer la voiture B de P_3 à P_1.
- Aller au volant de la voiture A (emplacement P_2).

Un algorithme possible

- Aller au volant de la voiture B (emplacement P_3).
- Répéter attendre jusqu'à emplacement P_1 libre.
- Déplacer la voiture B de P_3 à P_1.
- Aller au volant de la voiture A (emplacement P_2).
- Répéter attendre jusqu'à emplacement P_3 libre.

Un algorithme possible

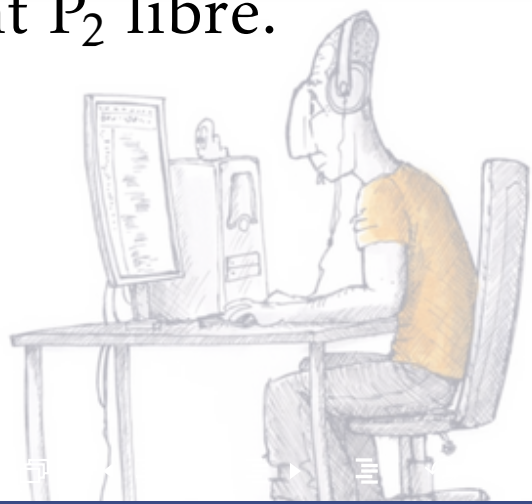
- Aller au volant de la voiture B (emplacement P_3).
- Répéter attendre jusqu'à emplacement P_1 libre.
- Déplacer la voiture B de P_3 à P_1.
- Aller au volant de la voiture A (emplacement P_2).
- Répéter attendre jusqu'à emplacement P_3 libre.
- Déplacer la voiture A de P_2 à P_3.

Un algorithme possible

- Aller au volant de la voiture B (emplacement P_3).
- Répéter attendre jusqu'à emplacement P_1 libre.
- Déplacer la voiture B de P_3 à P_1.
- Aller au volant de la voiture A (emplacement P_2).
- Répéter attendre jusqu'à emplacement P_3 libre.
- Déplacer la voiture A de P_2 à P_3.
- Aller au volant de la voiture B (emplacement P_1).

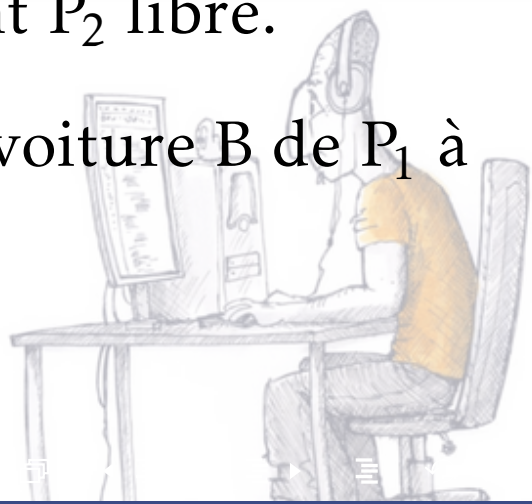
Un algorithme possible

- Aller au volant de la voiture B (emplacement P_3).
- Répéter attendre jusqu'à emplacement P_1 libre.
- Déplacer la voiture B de P_3 à P_1.
- Aller au volant de la voiture A (emplacement P_2).
- Répéter attendre jusqu'à emplacement P_3 libre.
- Déplacer la voiture A de P_2 à P_3.
- Aller au volant de la voiture B (emplacement P_1).
- Répéter attendre jusqu'à emplacement P_2 libre.



Un algorithme possible

- Aller au volant de la voiture B (emplacement P_3).
- Répéter attendre jusqu'à emplacement P_1 libre.
- Déplacer la voiture B de P_3 à P_1.
- Aller au volant de la voiture A (emplacement P_2).
- Répéter attendre jusqu'à emplacement P_3 libre.
- Déplacer la voiture A de P_2 à P_3.
- Aller au volant de la voiture B (emplacement P_1).
- Répéter attendre jusqu'à emplacement P_2 libre.
- Déplacer la voiture B de P_1 à P_2.



Un autre exemple

Le problème : Résoudre l'équation $ax = b$.

Un autre exemple

Le problème : Résoudre l'équation $ax = b$.
L'algorithme qui nous vient à l'esprit :

Un autre exemple

Le problème : Résoudre l'équation $ax = b$.
L'algorithme qui nous vient à l'esprit :

Si a n'est pas nul **alors**
  la solution est $x = \frac{b}{a}$
Sinon
  Il n'y a pas de solution

Un autre exemple

Le problème : Résoudre l'équation $ax = b$.

L'algorithme qui nous vient à l'esprit :

Si a n'est pas nul **alors**
  la solution est $x = \frac{b}{a}$
Sinon
  Il n'y a pas de solution

Là encore, l'énoncé n'est pas satisfaisant :

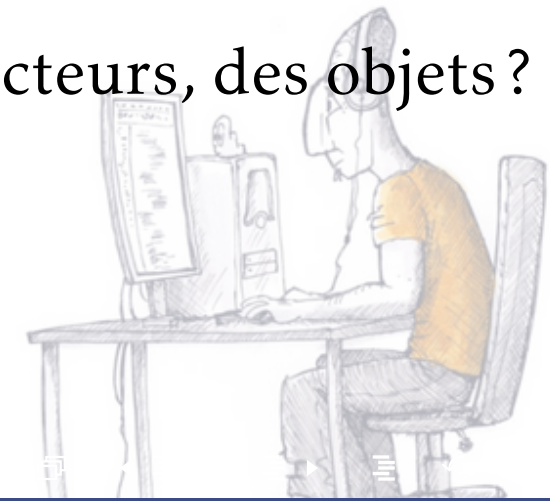
Un autre exemple

Le problème : Résoudre l'équation $ax = b$.
L'algorithme qui nous vient à l'esprit :

Si a n'est pas nul **alors**
  la solution est $x = \frac{b}{a}$
Sinon
  Il n'y a pas de solution

Là encore, l'énoncé n'est pas satisfaisant :

- Les variables x, a et b sont-elles réelles, des vecteurs, des objets ?



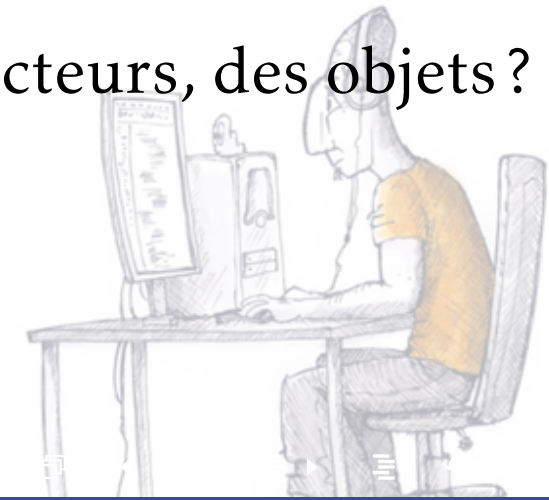
Un autre exemple

Le problème : Résoudre l'équation $ax = b$.
L'algorithme qui nous vient à l'esprit :

Si a n'est pas nul **alors**
  la solution est $x = \frac{b}{a}$
Sinon
  Il n'y a pas de solution

Là encore, l'énoncé n'est pas satisfaisant :

- Les variables x, a et b sont-elles réelles, des vecteurs, des objets ?
- Qui est l'inconnue ?



Un autre exemple

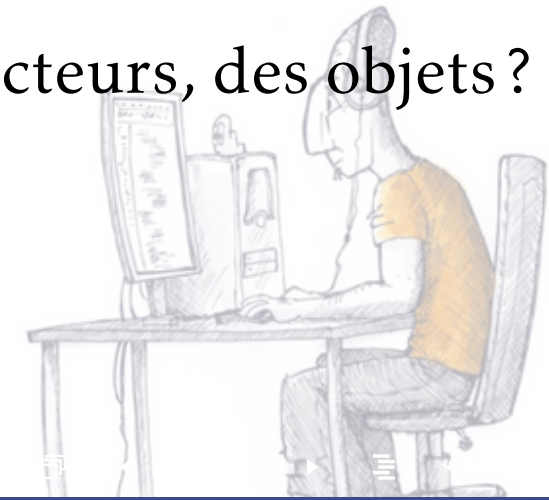
Le problème : Résoudre l'équation $ax = b$.
L'algorithme qui nous vient à l'esprit :

Si a n'est pas nul **alors**
  la solution est $x = \frac{b}{a}$
Sinon
  Il n'y a pas de solution

Là encore, l'énoncé n'est pas satisfaisant :

- Les variables x, a et b sont-elles réelles, des vecteurs, des objets ?
- Qui est l'inconnue ?

Reformulons le problème :



Un autre exemple

Le problème : Résoudre l'équation $ax = b$.

L'algorithme qui nous vient à l'esprit :

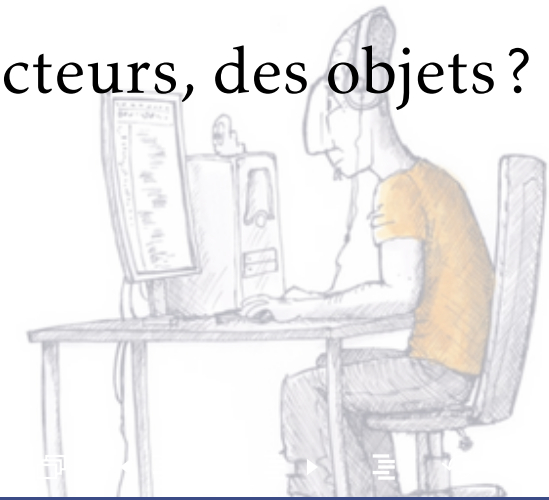
Si a n'est pas nul **alors**
  la solution est $x = \frac{b}{a}$
Sinon
  Il n'y a pas de solution

Là encore, l'énoncé n'est pas satisfaisant :

- Les variables x, a et b sont-elles réelles, des vecteurs, des objets ?
- Qui est l'inconnue ?

Reformulons le problème :

Soient $a \in \mathbb{R}^*$ et $b \in \mathbb{R}$, trouver x tel que $ax = b$



Méthodologie

Un bon algorithme doit :

Méthodologie

Un bon algorithme doit :

- être clair

Méthodologie

Un bon algorithme doit :

- être clair
- fournir un résultat satisfaisant

Méthodologie

Un bon algorithme doit :

- être clair
- fournir un résultat satisfaisant
- être adapté au public visé

Méthodologie

Pour cela, il est nécessaire de :

Un bon algorithme doit :

- être clair
- fournir un résultat satisfaisant
- être adapté au public visé

Méthodologie

Un bon algorithme doit :

- être clair
- fournir un résultat satisfaisant
- être adapté au public visé

Pour cela, il est nécessaire de :

- poser correctement le problème

Méthodologie

Un bon algorithme doit :

- être clair
- fournir un résultat satisfaisant
- être adapté au public visé

Pour cela, il est nécessaire de :

- poser correctement le problème
- rechercher une méthode de résolution

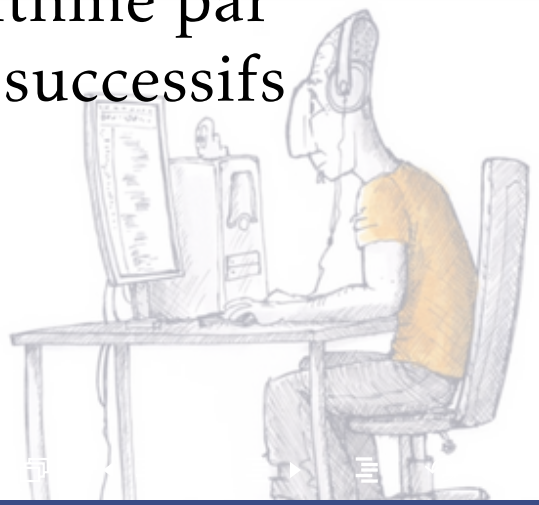
Méthodologie

Un bon algorithme doit :

- être clair
- fournir un résultat satisfaisant
- être adapté au public visé

Pour cela, il est nécessaire de :

- poser correctement le problème
- rechercher une méthode de résolution
- écrire l'algorithme par raffinements successifs



Pseudo-langage

Afin d'uniformiser l'écriture des algorithmes, un pseudo-langage (non normalisé) contenant l'indispensable est utilisé :

Pseudo-langage

Afin d'uniformiser l'écriture des algorithmes, un pseudo-langage (non normalisé) contenant l'indispensable est utilisé :

- Des données :

Pseudo-langage

Afin d'uniformiser l'écriture des algorithmes, un pseudo-langage (non normalisé) contenant l'indispensable est utilisé :

- Des données :

- Des variables :

Pseudo-langage

Afin d'uniformiser l'écriture des algorithmes, un pseudo-langage (non normalisé) contenant l'indispensable est utilisé :

- Des données :
- Des variables :
- Des opérateurs :

Pseudo-langage

Afin d'uniformiser l'écriture des algorithmes, un pseudo-langage (non normalisé) contenant l'indispensable est utilisé :

- Des données :
- Des variables :
- Des opérateurs :
- Des expressions :

Pseudo-langage

Afin d'uniformiser l'écriture des algorithmes, un pseudo-langage (non normalisé) contenant l'indispensable est utilisé :

- Des données :
- Des variables :
- Des opérateurs :
- Des expressions :
- Des instructions :

Pseudo-langage

Afin d'uniformiser l'écriture des algorithmes, un pseudo-langage (non normalisé) contenant l'indispensable est utilisé :

- Des données :
- Des variables :
- Des opérateurs :
- Des expressions :
- Des instructions :
- Des fonctions :

Pseudo-langage

- Des données : Ce sont des valeurs extérieures introduites dans l'algorithme. Une constante est une valeur non modifiable (π,...)
- Des variables :

- Des opérateurs :

- Des expressions :

- Des instructions :

- Des fonctions :

Pseudo-langage

- Des données : Ce sont des valeurs extérieures introduites dans l'algorithme. Une constante est une valeur non modifiable (π,...)
- Des variables : Elles sont modifiables, elles possèdent un type, contiennent une valeur et sont identifiées par un nom explicite
- Des opérateurs :

- Des expressions :

- Des instructions :

- Des fonctions :

Pseudo-langage

- Des données : Ce sont des valeurs extérieures introduites dans l'algorithme. Une constante est une valeur non modifiable (π,...)
- Des variables : Elles sont modifiables, elles possèdent un type, contiennent une valeur et sont identifiées par un nom explicite
- Des opérateurs : Chaque type de donnée admet un jeu d'opérateurs adapté (arithmétiques (+,-..), relationnels (==,>,..) et logiques (et, ou,..)
- Des expressions :

- Des instructions :

- Des fonctions :

Pseudo-langage

- Des données : Ce sont des valeurs extérieures introduites dans l'algorithme. Une constante est une valeur non modifiable (π,...)
- Des variables : Elles sont modifiables, elles possèdent un type, contiennent une valeur et sont identifiées par un nom explicite
- Des opérateurs : Chaque type de donnée admet un jeu d'opérateurs adapté (arithmétiques (+,-..), relationnels (==,>,..) et logiques (et, ou,..)
- Des expressions : b/a est une expression numérique et $a \neq 0$ est une expression booléenne
- Des instructions :

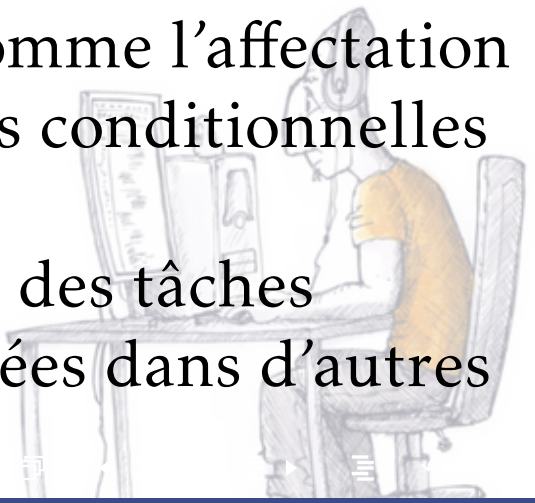
- Des fonctions :

Pseudo-langage

- Des données : Ce sont des valeurs extérieures introduites dans l'algorithme. Une constante est une valeur non modifiable (π,...)
- Des variables : Elles sont modifiables, elles possèdent un type, contiennent une valeur et sont identifiées par un nom explicite
- Des opérateurs : Chaque type de donnée admet un jeu d'opérateurs adapté (arithmétiques (+,-..), relationnels (==,>,..) et logiques (et, ou,..)
- Des expressions : b/a est une expression numérique et $a \neq 0$ est une expression booléenne
- Des instructions : Elles peuvent être simples comme l'affectation (x← b/a) ou structurées comme les instructions conditionnelles (if, else) et répétitives(for ou while)
- Des fonctions :

Pseudo-langage

- Des données : Ce sont des valeurs extérieures introduites dans l'algorithme. Une constante est une valeur non modifiable (π,...)
- Des variables : Elles sont modifiables, elles possèdent un type, contiennent une valeur et sont identifiées par un nom explicite
- Des opérateurs : Chaque type de donnée admet un jeu d'opérateurs adapté (arithmétiques (+,-..), relationnels (==,>,..) et logiques (et, ou,..)
- Des expressions : b/a est une expression numérique et $a \neq 0$ est une expression booléenne
- Des instructions : Elles peuvent être simples comme l'affectation (x← b/a) ou structurées comme les instructions conditionnelles (if, else) et répétitives(for ou while)
- Des fonctions : Elles permettent d'automatiser des tâches répétitives, d'ajouter de la clarté et d'être utilisées dans d'autres algorithmes



Principe de bonne programmation

Tous les algorithmes que nous avons vus sont assez courts

Principe de bonne programmation

Tous les algorithmes que nous avons vus sont assez courts
Dans l'industrie, il arrive que des équipes produisent des codes de millions de lignes, dont certains mettent en jeux des vies (aéronautique, nucléaire...)

Principe de bonne programmation

Tous les algorithmes que nous avons vus sont assez courts
Dans l'industrie, il arrive que des équipes produisent des codes de millions de lignes, dont certains mettent en jeux des vies (aéronautique, nucléaire...)
Toute la difficulté réside dans l'écriture de programmes sûrs.

Principe de bonne programmation

Tous les algorithmes que nous avons vus sont assez courts
Dans l'industrie, il arrive que des équipes produisent des codes de millions de lignes, dont certains mettent en jeux des vies (aéronautique, nucléaire...)
Toute la difficulté réside dans l'écriture de programmes sûrs. On constate que lorsqu'un algorithme est simple, le risque d'erreur est moindre..

Principe de bonne programmation

Tous les algorithmes que nous avons vus sont assez courts
Dans l'industrie, il arrive que des équipes produisent des codes de millions de lignes, dont certains mettent en jeux des vies (aéronautique, nucléaire...)
Toute la difficulté réside dans l'écriture de programmes sûrs. On constate que lorsqu'un algorithme est simple, le risque d'erreur est moindre..
C'est avec ce principe essentiel que se basent les bonnes méthodes

Principe de bonne programmation

Tous les algorithmes que nous avons vus sont assez courts
Dans l'industrie, il arrive que des équipes produisent des codes de millions de lignes, dont certains mettent en jeux des vies (aéronautique, nucléaire...)
Toute la difficulté réside dans l'écriture de programmes sûrs. On constate que lorsqu'un algorithme est simple, le risque d'erreur est moindre..
C'est avec ce principe essentiel que se basent les bonnes méthodes

Ainsi, tout problème compliqué doit être découpé en sous-problèmes simples

Principe de bonne programmation

Tous les algorithmes que nous avons vus sont assez courts
Dans l'industrie, il arrive que des équipes produisent des codes de millions de lignes, dont certains mettent en jeux des vies (aéronautique, nucléaire...)
Toute la difficulté réside dans l'écriture de programmes sûrs. On constate que lorsqu'un algorithme est simple, le risque d'erreur est moindre..
C'est avec ce principe essentiel que se basent les bonnes méthodes

Ainsi, tout problème compliqué doit être découpé en sous-problèmes simples

Chaque module ou sous-programme est alors testé et validé séparément.
Tous les modules doivent aussi être largement documentés.

Exemple : PGCD de deux entiers

Problème : Étant donné deux entiers naturels non nuls, déterminer leur PGCD

Exemple : PGCD de deux entiers

Il existe plusieurs méthodes pour déterminer le PGCD de deux entiers :

Exemple : PGCD de deux entiers

Par soustractions :

385-210=175

210-175=35

175-35=140

140-35=105

105-35=70

70-35=35

35-35=0

PGCD→ 35

Exemple : PGCD de deux entiers

Par soustractions :

385-210=175
210-175=35
175-35=140
140-35=105
105-35=70
70-35=35
35-35=0
PGCD→ 35

Avec leur décomposition en produit :

$385 = 7 \times 6 \times 5$
$210 = 7 \times 5 \times 11$
PGCD→ $7 \times 5 = 35$

Exemple : PGCD de deux entiers

Par soustractions :

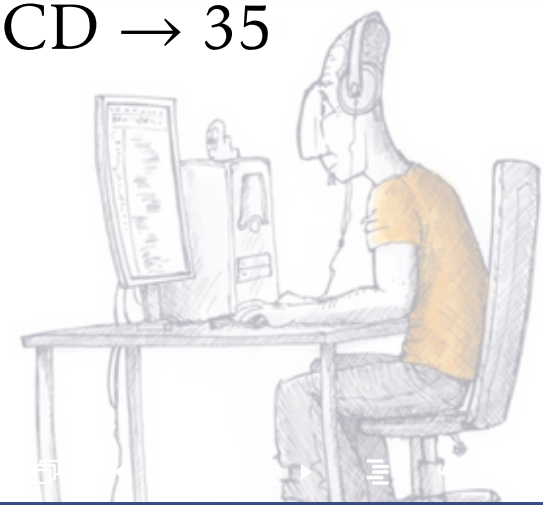
385-210=175
210-175=35
175-35=140
140-35=105
105-35=70
70-35=35
35-35=0
PGCD→ 35

Avec leur décomposition en produit :

$385 = 7 \times 6 \times 5$
$210 = 7 \times 5 \times 11$
PGCD→ $7 \times 5 = 35$

L'algorithme d'Euclide :

$385 = 1 \times 210 + 175$
$210 = 1 \times 175 + 35$
$175 = 5 \times 35 + 0$
PGCD → 35



Exemple : PGCD de deux entiers

Algorithme de la 1ère méthode :

Données : :(a,b)$\in \mathbb{N}^* \times \mathbb{N}^*$

```
fonction pgcd(a,b)
u ← a
v ← b
Tant que u ≠ v faire
    Si u > v alors
        u ← u-v
    Sinon
        v ← v-u

retourner u
```

Exemple : PGCD de deux entiers

Algorithme de la 1ère méthode :

Données : :$(a,b) \in \mathbb{N}^* \times \mathbb{N}^*$

fonction pgcd(a,b)
u ← a
v ← b
Tant que u ≠ v **faire**
    Si u > v **alors**
        u ← u-v
    Sinon
        v ← v-u

retourner u

Sa traduction en Python :

```
def pgcd(a,b):
    '''
    a , b deux entiers naturels non nuls
    retourne le PGCD de a et b
    '''
    u=a
    v=b
    while u!=v:
        if u>v:
            u=u-v
        else:
            v=v-u
    return u
```

Analyse des algorithmes

Problématique

Analyse des algorithmes

Problématique

Problème : Comment s'assurer qu'un algorithme se termine, donne des résultats corrects et est efficace ?

Analyse des algorithmes

Problématique

Problème : Comment s'assurer qu'un algorithme se termine, donne des résultats corrects et est efficace ?
L'analyse des algorithmes peut se faire suivant trois axes :

Analyse des algorithmes

Problématique

Problème : Comment s'assurer qu'un algorithme se termine, donne des résultats corrects et est efficace ?
L'analyse des algorithmes peut se faire suivant trois axes :

- **La terminaison :** il s'agit de vérifier que l'algorithme ne tourne pas de manière infinie

Analyse des algorithmes

Problématique

Problème : Comment s'assurer qu'un algorithme se termine, donne des résultats corrects et est efficace ?
L'analyse des algorithmes peut se faire suivant trois axes :

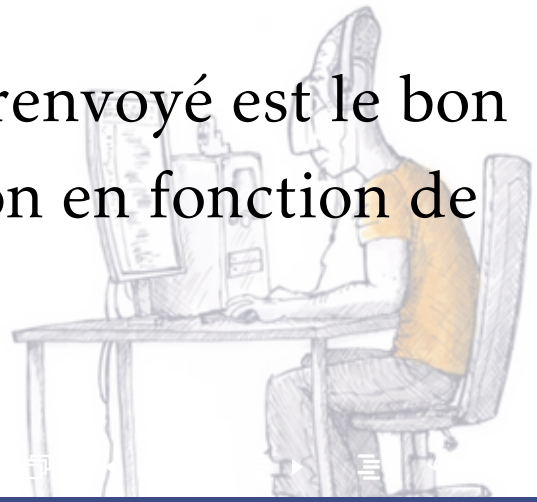
- **La terminaison :** il s'agit de vérifier que l'algorithme ne tourne pas de manière infinie
- **La correction :** il faut s'assurer que le résultat renvoyé est le bon

Analyse des algorithmes

Problématique

Problème : Comment s'assurer qu'un algorithme se termine, donne des résultats corrects et est efficace ?
L'analyse des algorithmes peut se faire suivant trois axes :

- **La terminaison :** il s'agit de vérifier que l'algorithme ne tourne pas de manière infinie
- **La correction :** il faut s'assurer que le résultat renvoyé est le bon
- **La complexité :** on spécifie le temps d'exécution en fonction de la taille des données en entrée



Pourquoi faire ?

Pourquoi ces études sont-elles intéressantes ?

On ne peut prouver un théorème par des exemples.

Pourquoi faire ?

Pourquoi ces études sont-elles intéressantes ?

On ne peut prouver un théorème par des exemples.
Exemple :affirmer que les nombres se terminant par 9 sont divisibles par 3 est faux.

Pourquoi faire ?

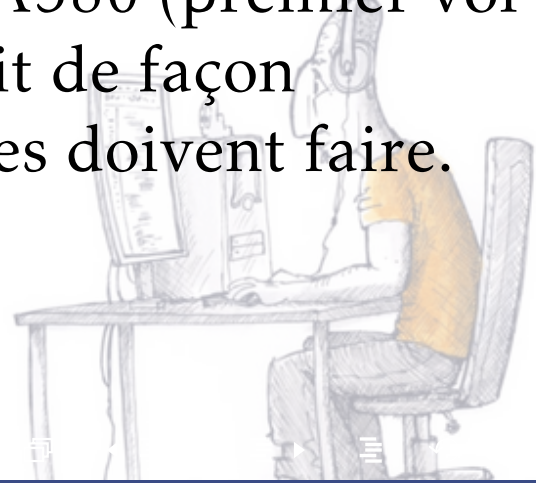
Pourquoi ces études sont-elles intéressantes ?

On ne peut prouver un théorème par des exemples.
Exemple :affirmer que les nombres se terminant par 9 sont divisibles par 3 est faux.
c'est vrai pour 9, 39, 69 et 1239 mais faux pour 19

Pourquoi faire ?

Pourquoi ces études sont-elles intéressantes ?

La preuve de programme n'est pas encore très développée en informatique industrielle. Le logiciel de vol de l'A380 (premier vol en 2005) est le premier à avoir été prouvé : on sait de façon certaine que les commandes de vol font ce qu'elles doivent faire. Ce n'était pas le cas sur les avions précédents.



Quelques anecdotes...

On trouve toutes sortes d'anecdotes sur Internet :

Quelques anecdotes...

On trouve toutes sortes d'anecdotes sur Internet :

- Problème de terminaison : Le livre qui valait 23,7 million de dollars (Amazon)

Quelques anecdotes...

On trouve toutes sortes d'anecdotes sur Internet :

- Problème de terminaison : Le livre qui valait 23,7 million de dollars (Amazon)
- Problème de correction : Le crash de la sonde Mars Climate Orbiter (problème d'unité de mesure entre deux parties du programme)

Quelques anecdotes...

On trouve toutes sortes d'anecdotes sur Internet :

- Problème de terminaison : Le livre qui valait 23,7 million de dollars (Amazon)
- Problème de correction : Le crash de la sonde Mars Climate Orbiter (problème d'unité de mesure entre deux parties du programme)
- Problème de complexité : Lors de sa mise en ligne le portail France.fr n'avait pas prévu l'afflux de visiteurs...

La terminaison d'un algorithme

Définition : Terminaison d'un algorithme

Un algorithme se termine, si son exécution sur machine s'arrête toujours quelque soit la nature des données en entrée

La terminaison d'un algorithme

Définition : Terminaison d'un algorithme

Un algorithme se termine, si son exécution sur machine s'arrête toujours quelque soit la nature des données en entrée

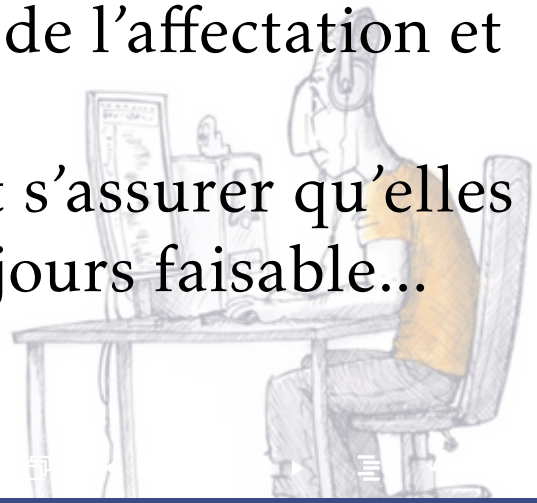
Certaines instructions se terminent sans même que l'on ait besoin de se poser de question. C'est le cas par exemple de l'affectation et de la boucle bornée for.

La terminaison d'un algorithme

Définition : Terminaison d'un algorithme

Un algorithme se termine, si son exécution sur machine s'arrête toujours quelque soit la nature des données en entrée

Certaines instructions se terminent sans même que l'on ait besoin de se poser de question. C'est le cas par exemple de l'affectation et de la boucle bornée for.
Le problème vient des boucles «Tant que», il faut s'assurer qu'elles se terminent et nous verrons que ce n'est pas toujours faisable...



Terminaison : exemple 1

Données : $n \in \mathbb{N}$

S$\leftarrow$ 0
Pour i allant de 0 à n **faire**
    S=S+i

retourner S

Terminaison : exemple 1

Données : $n \in \mathbb{N}$

S$\leftarrow$ 0
Pour i allant de 0 à n **faire**
    S=S+i

retourner S

La terminaison :

Terminaison : exemple 1

Données : $n \in \mathbb{N}$

S$\leftarrow$ 0
Pour i allant de 0 à n **faire**
    S=S+i

retourner S

La terminaison :
Les affectations ne posent pas de problème et la boucle est bornée

Terminaison : exemple 1

Données : $n \in \mathbb{N}$

S$\leftarrow$ 0
Pour i allant de 0 à n **faire**
    S=S+i

retourner S

La terminaison :
Les affectations ne posent pas de problème et la boucle est bornée
L'algorithme se termine bien...

Terminaison : exemple 2

Données : $n \in \mathbb{N}$

S $\leftarrow$ 0
i $\leftarrow$ 0
Tant que $i <= n$ **faire**
    S=S+i
    i=i+1

retourner S

Terminaison : exemple 2

La terminaison :

Données : $n \in \mathbb{N}$

S $\leftarrow$ 0
i $\leftarrow$ 0
Tant que $i <= n$ **faire**
    S=S+i
    i=i+1

retourner S

Terminaison : exemple 2

Données : $n \in \mathbb{N}$

S $\leftarrow$ 0
i $\leftarrow$ 0
Tant que $i <= n$ **faire**
    S=S+i
    i=i+1

retourner S

La terminaison :
Les affectations ne posent pas de problème

Terminaison : exemple 2

Données : $n \in \mathbb{N}$

S ← 0
i ← 0
Tant que $i <= n$ **faire**
    S=S+i
    i=i+1

retourner S

La terminaison :
Les affectations ne posent pas de problème
Pour savoir si la boucle Tant que se termine, il faut exhiber ce que l'on appelle un **variant de boucle**, c'est à dire une suite d'entiers strictement croissante majorée (ou strictement décroissante minorée)

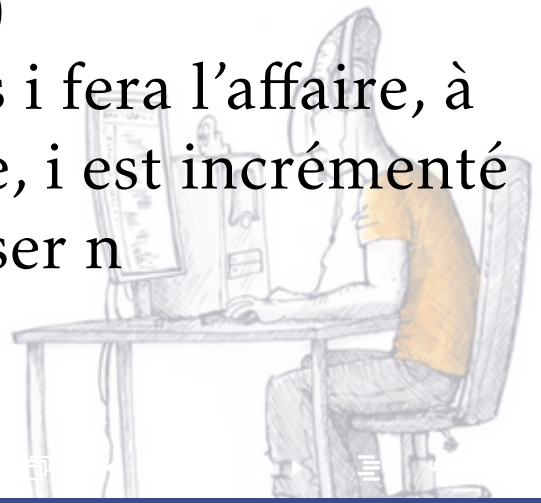
Terminaison : exemple 2

Données : $n \in \mathbb{N}$

S ← 0
i ← 0
Tant que $i <= n$ **faire**
    S=S+i
    i=i+1

retourner S

La terminaison :
Les affectations ne posent pas de problème
Pour savoir si la boucle Tant que se termine, il faut exhiber ce que l'on appelle un **variant de boucle**, c'est à dire une suite d'entiers strictement croissante majorée (ou strictement décroissante minorée)
La suite des itérateurs i fera l'affaire, à chaque tour de boucle, i est incrémenté de 1 et ne peut dépasser n



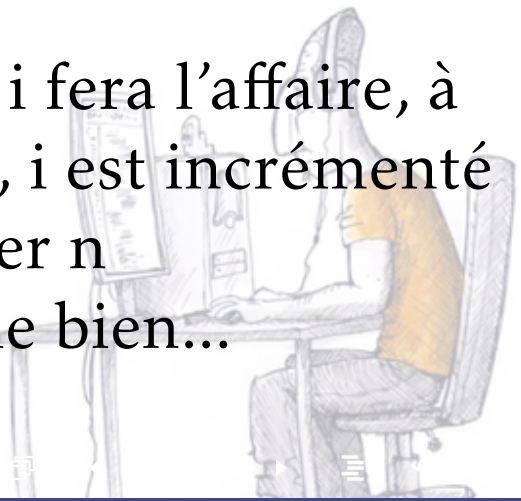
Terminaison : exemple 2

Données : $n \in \mathbb{N}$

S ← 0
i ← 0
Tant que $i <= n$ **faire**
    S=S+i
    i=i+1

retourner S

La terminaison :
Les affectations ne posent pas de problème
Pour savoir si la boucle Tant que se termine, il faut exhiber ce que l'on appelle un **variant de boucle**, c'est à dire une suite d'entiers strictement croissante majorée (ou strictement décroissante minorée)
La suite des itérateurs i fera l'affaire, à chaque tour de boucle, i est incrémenté de 1 et ne peut dépasser n
L'algorithme se termine bien...



Terminaison : exemple 3

Données : $(a,b) \in \mathbb{N}^* \times \mathbb{N}^*$

```
fonction pgcd(a,b)
u ← a
v ← b
Tant que u ≠ v faire
    Si u > v alors
        u ← u-v
    Sinon
        v ← v-u

retourner u
```

Terminaison : exemple 3

Données : $(a,b) \in \mathbb{N}^* \times \mathbb{N}^*$

```
fonction pgcd(a,b)
u ← a
v ← b
Tant que u ≠ v faire
    Si u > v alors
        u ← u-v
    Sinon
        v ← v-u
retourner u
```

La terminaison :

Terminaison : exemple 3

Données : $(a,b) \in \mathbb{N}^* \times \mathbb{N}^*$

fonction pgcd(a,b)
u ← a
v ← b
Tant que u ≠ v **faire**
    Si u > v **alors**
        u ← u-v
    Sinon
        v ← v-u

retourner u

La terminaison :
Les affectations ne posent pas de problème

Terminaison : exemple 3

Données : (a,b)$\in \mathbb{N}^* \times \mathbb{N}^*$

fonction pgcd(a,b)
u $\leftarrow$ a
v $\leftarrow$ b
Tant que u $\neq$ v **faire**
    Si u > v **alors**
        u $\leftarrow$ u-v
    Sinon
        v $\leftarrow$ v-u

retourner u

La terminaison :
Les affectations ne posent pas de problème
À chaque tour de boucle, on calcule la différence $u - v$ ou $v - u$.

Terminaison : exemple 3

Données : (a,b)$\in \mathbb{N}^* \times \mathbb{N}^*$

```
fonction pgcd(a,b)
u ← a
v ← b
Tant que u ≠ v faire
    Si u > v alors
        u ← u-v
    Sinon
        v ← v-u
retourner u
```

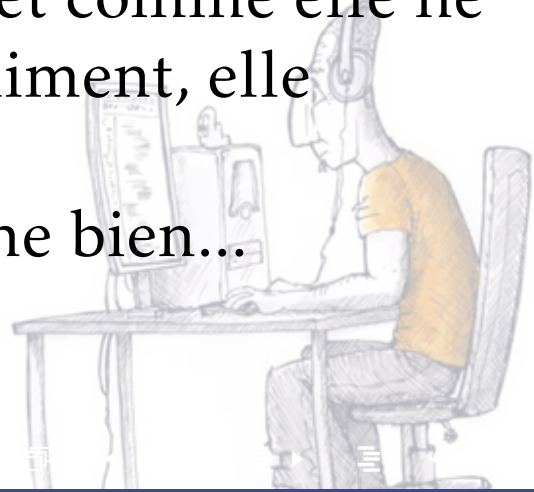
La terminaison :
Les affectations ne posent pas de problème
À chaque tour de boucle, on calcule la différence $u - v$ ou $v - u$.
La suite abs(u-v) est une suite d'entiers strictement décroissante et positive (donc minorée par 0) et comme elle ne peut décroître indéfiniment, elle atteindra donc 0

Terminaison : exemple 3

Données : (a,b)$\in \mathbb{N}^* \times \mathbb{N}^*$

```
fonction pgcd(a,b)
u ← a
v ← b
Tant que u ≠ v faire
    Si u > v alors
        u ← u-v
    Sinon
        v ← v-u
retourner u
```

La terminaison :
Les affectations ne posent pas de problème
À chaque tour de boucle, on calcule la différence $u - v$ ou $v - u$.
La suite abs(u-v) est une suite d'entiers strictement décroissante et positive (donc minorée par 0) et comme elle ne peut décroître indéfiniment, elle atteindra donc 0
L'algorithme se termine bien...



Terminaison : exemple 4 (la suite de Syracuse)

La suite de Syracuse d'un nombre entier N > 0 est définie par :

$$\begin{cases} u_0 = \mathrm{N} > 0 \\ u_{n+1} = \dfrac{u_n}{2} \text{ si } u_n \text{ est pair} \\ u_{n+1} = 3u_n + 1 \text{ sinon} \end{cases}$$

La conjecture de Syracuse, affirme que quelque soit l'entier N > 0 la suite aboutit sur le nombre 1 !

Il n'existe aucune preuve mathématique de ce résultat

Terminaison : exemple 4 (la suite de Syracuse)

Données : :N ∈ $\mathbb{N}^*$

```
fonction syracuse(N)
u ← N
Tant que u ≠ 1 faire
    Si u est pair alors
        u ← u/2
    Sinon
        u ← 3*u+1
retourner u
```

Terminaison : exemple 4 (la suite de Syracuse)

Données : :N ∈ $\mathbb{N}^*$

```
fonction syracuse(N)
u ← N
Tant que u ≠ 1 faire
    Si u est pair alors
        u ← u/2
    Sinon
        u ← 3*u+1
retourner u
```

La terminaison :

Terminaison : exemple 4 (la suite de Syracuse)

Données : :N ∈ $\mathbb{N}^*$

```
fonction syracuse(N)
u ← N
Tant que u ≠ 1 faire
    Si u est pair alors
        u ← u/2
    Sinon
        u ← 3*u+1
retourner u
```

La terminaison :
Comme il n'existe pas de preuve mathématique, et même si tous les nombres testés aboutissent bien sur 1

Terminaison : exemple 4 (la suite de Syracuse)

Données : :N ∈ $\mathbb{N}^*$

```
fonction syracuse(N)
u ← N
Tant que u ≠ 1 faire
    Si u est pair alors
        u ← u/2
    Sinon
        u ← 3*u+1
retourner u
```

La terminaison :
Comme il n'existe pas de preuve mathématique, et même si tous les nombres testés aboutissent bien sur 1
On ne peut prouver la terminaison...

La correction d'un algorithme

Définition : Correction d'un algorithme

Un algorithme est correct, si son exécution sur machine donne toujours le bon résultat quelque soit l'entrée

La correction d'un algorithme

Définition : Correction d'un algorithme

Un algorithme est correct, si son exécution sur machine donne toujours le bon résultat quelque soit l'entrée

Pour montrer qu'un algorithme est correct, on utilise ce que l'on appelle un **invariant de boucle**

La correction d'un algorithme

Définition : Correction d'un algorithme

Un algorithme est correct, si son exécution sur machine donne toujours le bon résultat quelque soit l'entrée

Pour montrer qu'un algorithme est correct, on utilise ce que l'on appelle un **invariant de boucle**
Un invariant de boucle est une propriété :

- Qui est vérifiée après l'initialisation

La correction d'un algorithme

Définition : Correction d'un algorithme

Un algorithme est correct, si son exécution sur machine donne toujours le bon résultat quelque soit l'entrée

Pour montrer qu'un algorithme est correct, on utilise ce que l'on appelle un **invariant de boucle**
Un invariant de boucle est une propriété :

- Qui est vérifiée après l'initialisation
- Qui reste vraie après l'exécution d'une itération

La correction d'un algorithme

Définition : Correction d'un algorithme

Un algorithme est correct, si son exécution sur machine donne toujours le bon résultat quelque soit l'entrée

Pour montrer qu'un algorithme est correct, on utilise ce que l'on appelle un **invariant de boucle**
Un invariant de boucle est une propriété :

- Qui est vérifiée après l'initialisation
- Qui reste vraie après l'exécution d'une itération
- Qui permet de montrer que le résultat attendu est bien celui qui est calculé

Correction : exemple 1

Données : $n \in \mathbb{N}$

S$\leftarrow$ 0
Pour i allant de 0 à n **faire**
    S=S+i

retourner S

Correction : exemple 1

La correction :

Données : $n \in \mathbb{N}$

S$\leftarrow$ 0
Pour i allant de 0 à n **faire**
  S=S+i

retourner S

Correction : exemple 1

La correction :
On considère la propriété : S contient la somme des entiers jusqu'à n

Données : $n \in \mathbb{N}$

S$\leftarrow$ 0
Pour i allant de 0 à n **faire**
  S=S+i

retourner S

Correction : exemple 1

Données : $n \in \mathbb{N}$

S$\leftarrow$ 0
Pour i allant de 0 à n **faire**
    S=S+i

retourner S

La correction :
On considère la propriété : S contient la somme des entiers jusqu'à n
Lors de l'initialisation, S contient 0 donc S contient bien la somme des entiers jusqu'à 0

Correction : exemple 1

Données : $n \in \mathbb{N}$

S$\leftarrow$ 0
Pour i allant de 0 à n **faire**
  S=S+i

retourner S

La correction :
On considère la propriété : S contient la somme des entiers jusqu'à n
Lors de l'initialisation, S contient 0 donc S contient bien la somme des entiers jusqu'à 0
Pour i=k, S contient la somme des k premiers entiers, à l'étape suivante i=k+1 S contiendra la somme des k+1 premiers entiers

Correction : exemple 1

Données : $n \in \mathbb{N}$

S$\leftarrow$ 0
Pour i allant de 0 à n **faire**
  S=S+i

retourner S

La correction :
On considère la propriété : S contient la somme des entiers jusqu'à n
Lors de l'initialisation, S contient 0 donc S contient bien la somme des entiers jusqu'à 0
Pour i=k, S contient la somme des k premiers entiers, à l'étape suivante i=k+1 S contiendra la somme des k+1 premiers entiers
Puisque l'algorithme se termine pour k=n, et que S contient la somme des n premiers entiers, l'algorithme est donc correct

Correction : exemple 2

Données : $(a,b) \in \mathbb{N}^* \times \mathbb{N}^*$

```
fonction pgcd(a,b)
u ← a
v ← b
Tant que u ≠ v faire
    Si u > v alors
        u ← u-v
    Sinon
        v ← v-u
retourner u
```

Correction : exemple 2

Données : (a,b)$\in \mathbb{N}^* \times \mathbb{N}^*$

```
fonction pgcd(a,b)
u ← a
v ← b
Tant que u ≠ v faire
    Si u > v alors
        u ← u-v
    Sinon
        v ← v-u
retourner u
```

La correction :

Correction : exemple 2

Données : (a,b)$\in \mathbb{N}^* \times \mathbb{N}^*$

```
fonction pgcd(a,b)
u ← a
v ← b
Tant que u ≠ v faire
    Si u > v alors
        u ← u-v
    Sinon
        v ← v-u

retourner u
```

La correction :
On considère la propriété :
pgcd(a,b)=pgcd(u,v)

Correction : exemple 2

Données : (a,b)$\in \mathbb{N}^* \times \mathbb{N}^*$

```
fonction pgcd(a,b)
u ← a
v ← b
Tant que u ≠ v faire
    Si u > v alors
        u ← u-v
    Sinon
        v ← v-u
retourner u
```

La correction :
On considère la propriété :
pgcd(a,b)=pgcd(u,v)
Un théorème de mathématique nous assure que :

$$pgcd(u,v) = pgcd(u-v,v) = pgcd(u,v-u)$$

Correction : exemple 2

Données : (a,b)$\in \mathbb{N}^* \times \mathbb{N}^*$

```
fonction pgcd(a,b)
u ← a
v ← b
Tant que u ≠ v faire
    Si u > v alors
        u ← u-v
    Sinon
        v ← v-u
retourner u
```

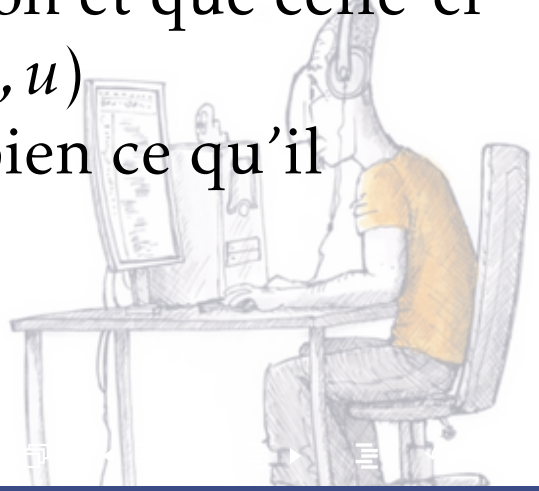
La correction :
On considère la propriété :
pgcd(a,b)=pgcd(u,v)
Un théorème de mathématique nous assure que :
$pgcd(u,v) = pgcd(u-v,v) = pgcd(u,v-u)$
Comme c'est exactement ce qu'il se passe à chaque itération et que celle-ci se termine sur $pgcd(u,u)$

Correction : exemple 2

Données : (a,b)$\in \mathbb{N}^* \times \mathbb{N}^*$

```
fonction pgcd(a,b)
u ← a
v ← b
Tant que u ≠ v faire
    Si u > v alors
        u ← u-v
    Sinon
        v ← v-u
retourner u
```

La correction :
On considère la propriété :
pgcd(a,b)=pgcd(u,v)
Un théorème de mathématique nous assure que :
$pgcd(u,v) = pgcd(u-v,v) = pgcd(u,v-u)$
Comme c'est exactement ce qu'il se passe à chaque itération et que celle-ci se termine sur $pgcd(u,u)$
L'algorithme calcule bien ce qu'il promet...



Complexité d'un algorithme

En plus d'être correct, on demande à un algorithme d'être efficace.

Complexité d'un algorithme

En plus d'être correct, on demande à un algorithme d'être efficace.
Cette efficacité peut être évaluée par :

Complexité d'un algorithme

En plus d'être correct, on demande à un algorithme d'être efficace. Cette efficacité peut être évaluée par :

- le temps que prend l'exécution de l'algorithme : c'est la **complexité en temps**. Le temps d'exécution dépend de la taille des données fournies en entrée.

Complexité d'un algorithme

En plus d'être correct, on demande à un algorithme d'être efficace. Cette efficacité peut être évaluée par :

- le temps que prend l'exécution de l'algorithme : c'est la **complexité en temps**. Le temps d'exécution dépend de la taille des données fournies en entrée.
- les ressources nécessaires à son exécution : c'est **la complexité en mémoire**

Complexité d'un algorithme

En plus d'être correct, on demande à un algorithme d'être efficace. Cette efficacité peut être évaluée par :

- le temps que prend l'exécution de l'algorithme : c'est la **complexité en temps**. Le temps d'exécution dépend de la taille des données fournies en entrée.
- les ressources nécessaires à son exécution : c'est **la complexité en mémoire**

On s'intéressera à la complexité en temps qui correspond au nombre d'opérations élémentaires

Complexité - opérations élémentaires

Nous étudions donc le nombre d'opérations élémentaires

Complexité - opérations élémentaires

Nous étudions donc le nombre d'opérations élémentaires
On considère comme élémentaire les opérations :

Complexité - opérations élémentaires

Nous étudions donc le nombre d'opérations élémentaires
On considère comme élémentaire les opérations :

- addition, soustraction, multiplication, division, modulo sur des entiers ou des flottants

Complexité - opérations élémentaires

Nous étudions donc le nombre d'opérations élémentaires
On considère comme élémentaire les opérations :

- addition, soustraction, multiplication, division, modulo sur des entiers ou des flottants
- affectation simple

Complexité - opérations élémentaires

Nous étudions donc le nombre d'opérations élémentaires
On considère comme élémentaire les opérations :

- addition, soustraction, multiplication, division, modulo sur des entiers ou des flottants
- affectation simple
- accès aux éléments d'un tableau, modification d'un éléments d'un tableau et taille d'un tableau

Complexité - opérations élémentaires

Nous étudions donc le nombre d'opérations élémentaires
On considère comme élémentaire les opérations :

- addition, soustraction, multiplication, division, modulo sur des entiers ou des flottants
- affectation simple
- accès aux éléments d'un tableau, modification d'un éléments d'un tableau et taille d'un tableau
- La méthode append sur une liste Python peut être considérée comme une opération basique

Complexité - opérations élémentaires

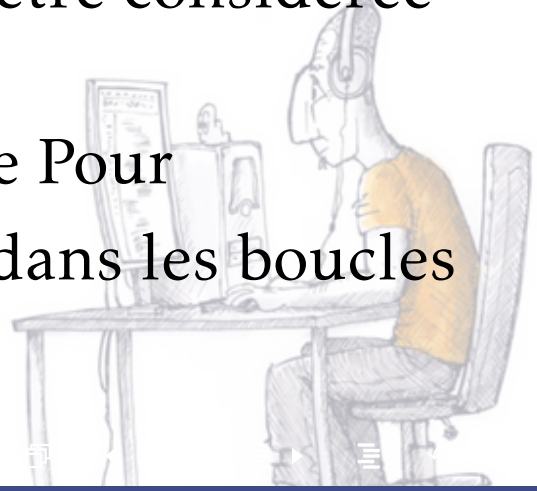
Nous étudions donc le nombre d'opérations élémentaires
On considère comme élémentaire les opérations :

- addition, soustraction, multiplication, division, modulo sur des entiers ou des flottants
- affectation simple
- accès aux éléments d'un tableau, modification d'un éléments d'un tableau et taille d'un tableau
- La méthode append sur une liste Python peut être considérée comme une opération basique
- la gestion de la variable de boucle d'une boucle Pour

Complexité - opérations élémentaires

Nous étudions donc le nombre d'opérations élémentaires
On considère comme élémentaire les opérations :

- addition, soustraction, multiplication, division, modulo sur des entiers ou des flottants
- affectation simple
- accès aux éléments d'un tableau, modification d'un éléments d'un tableau et taille d'un tableau
- La méthode append sur une liste Python peut être considérée comme une opération basique
- la gestion de la variable de boucle d'une boucle Pour
- les tests élémentaires, utilisés principalement dans les boucles Tant que et les structures conditionnelles



Complexité - Les types de complexité

Notation : $O(n)$ ou $\Theta(n)$

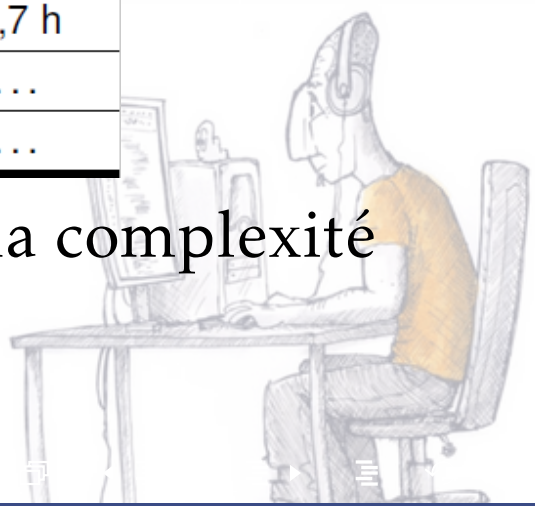
Notation	Type de complexité
$\Theta(1)$	Complexité constante
$\Theta(\ln(n))$	Complexité logarithmique
$\Theta(n)$	Complexité linéaire
$\Theta(n.\ln(n))$	Complexité quasi-linéaire
$\Theta(n^2)$	Complexité quadratique
$\Theta(n^3)$	Complexité cubique
$\Theta(n^p)$	Complexité polynomiale
$\Theta(n^{\ln(n)})$	Complexité quasi-polynomiale
$\Theta(2^n)$	Complexité exponentielle
$\Theta(n!)$	Complexité factorielle

Complexité - Les types de complexité

En s'appuyant sur une base de 10^9 opérations par secondes, le tableau ci-dessous donne le temps d'exécution des différentes complexités citées précédemment.

n	5	10	20	50	250	10^3	10^4
1	10 ns	10 ns	10 ns	10 ns	10 ns	10 ns	10 ns
$\log(n)$	10 ns	10 ns	10 ns	20 ns	30 ns	30 ns	40 ns
$O(n)$	50 ns	100 ns	200 ns	500 ns	2,5 μs	10 μs	100 μs
$n.\log(n)$	50 ns	100 ns	200 ns	501 ns	2,5 μs	10 μs	100.5 μs
n^2	250 ns	1 μs	4 μs	25 μs	625 μs	10 ms	1 s
n^3	1,25 μs	10 μs	80 μs	1,25 ms	156 ms	10 s	2,7 h
2^n	320 ns	10 μs	10 ms	130 jours	10^{59} ans	...	...
$n!$	1,2 μs	36 ms	770 ans	10^{45} ans	...	...	...

Il faut donc éviter les complexités supérieures à la complexité polynomiale d'ordre 2...



Complexité : exemple 1

Données : $n \in \mathbb{N}$

S$\leftarrow$ 0
Pour i allant de 0 à n **faire**
  S=S+i

retourner S

Complexité : exemple 1

La complexité :

Données : $n \in \mathbb{N}$

S$\leftarrow$ 0
Pour i allant de 0 à n **faire**
  S=S+i

retourner S

Complexité : exemple 1

Données : $n \in \mathbb{N}$

S← 0
Pour i allant de 0 à n **faire**
  S=S+i

retourner S

La complexité :
S ← 0 une opération

Complexité : exemple 1

Données : $n \in \mathbb{N}$

S$\leftarrow$ 0
Pour i allant de 0 à n **faire**
  S=S+i

retourner S

La complexité :
S $\leftarrow$ 0 une opération
Il y a $n+1$ additions et $n+1$ affectations

Complexité : exemple 1

Données : $n \in \mathbb{N}$

S$\leftarrow$ 0
Pour i allant de 0 à n **faire**
  S=S+i

retourner S

La complexité :
S $\leftarrow$ 0 une opération
Il y a $n+1$ additions et $n+1$ affectations
Il y a également les n incrémentations de la boucle for

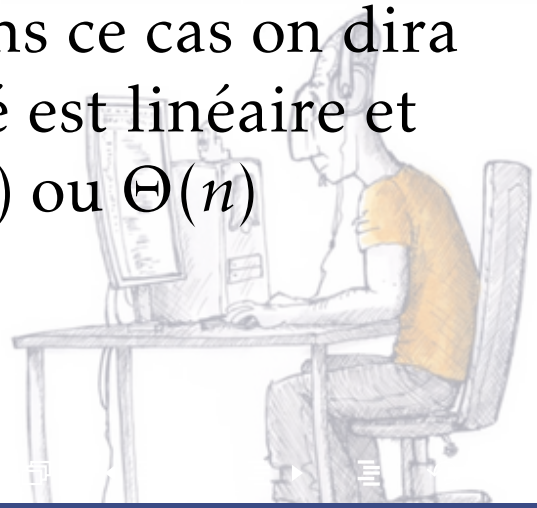
Complexité : exemple 1

Données : $n \in \mathbb{N}$

S$\leftarrow$ 0
Pour i allant de 0 à n **faire**
  S=S+i

retourner S

La complexité :
S $\leftarrow$ 0 une opération
Il y a $n+1$ additions et $n+1$ affectations
Il y a également les n incrémentations de la boucle for
Il y a au total $3n+3$ opérations élémentaires. Dans ce cas on dira que la complexité est linéaire et cela se note : $O(n)$ ou $\Theta(n)$



Complexité : exemple 2

Données : $(a,b) \in \mathbb{N}^* \times \mathbb{N}^*$

```
fonction pgcd(a,b)
u ← a
v ← b
Tant que u ≠ v faire
    Si u > v alors
        u ← u-v
    Sinon
        v ← v-u

retourner u
```

Complexité : exemple 2

Données : $(a,b) \in \mathbb{N}^* \times \mathbb{N}^*$

```
fonction pgcd(a,b)
u ← a
v ← b
Tant que u ≠ v faire
    Si u > v alors
        u ← u-v
    Sinon
        v ← v-u
retourner u
```

La complexité :

Complexité : exemple 2

Données : $(a,b) \in \mathbb{N}^* \times \mathbb{N}^*$

```
fonction pgcd(a,b)
u ← a
v ← b
Tant que u ≠ v faire
    Si u > v alors
        u ← u-v
    Sinon
        v ← v-u
retourner u
```

La complexité :
2 affectations

Complexité : exemple 2

Données : (a,b)$\in \mathbb{N}^* \times \mathbb{N}^*$

```
fonction pgcd(a,b)
u ← a
v ← b
Tant que u ≠ v faire
    Si u > v alors
        u ← u-v
    Sinon
        v ← v-u
retourner u
```

La complexité :
2 affectations
Le pire des cas possible est $a = n$ et $b = 1$

Complexité : exemple 2

Données : (a,b)$\in \mathbb{N}^* \times \mathbb{N}^*$

```
fonction pgcd(a,b)
u ← a
v ← b
Tant que u ≠ v faire
    Si u > v alors
        u ← u-v
    Sinon
        v ← v-u
retourner u
```

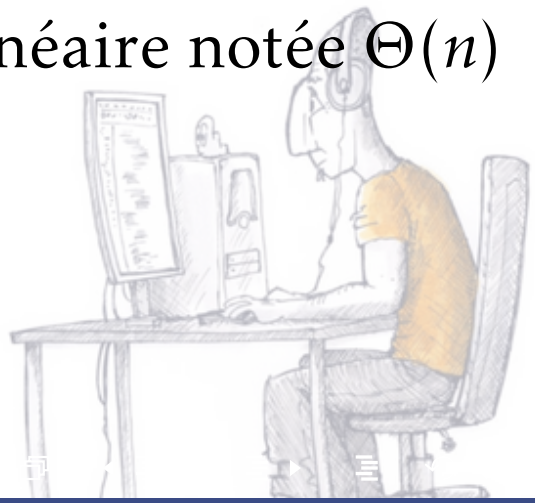
La complexité :
2 affectations
Le pire des cas possible est $a = n$ et $b = 1$
Il y aura n tests (Tant que), n test (Si), n soustractions et n affectations

Complexité : exemple 2

Données : (a,b)$\in \mathbb{N}^* \times \mathbb{N}^*$

```
fonction pgcd(a,b)
u ← a
v ← b
Tant que u ≠ v faire
    Si u > v alors
        u ← u-v
    Sinon
        v ← v-u
retourner u
```

La complexité :
2 affectations
Le pire des cas possible est $a = n$ et $b = 1$
Il y aura n tests (Tant que), n test (Si), n soustractions et n affectations
Au total $4n + 2$ opérations élémentaires
soit une complexité linéaire notée $\Theta(n)$ ou $O(n)$



Complexité : exemple 3

Données : n $n \in \mathbb{N}$

S$\leftarrow$ 0
Pour i allant de 1 à n **faire**
    Pour j allant de 1 à n **faire**
        S=S+i+j
retourner S

Complexité : exemple 3

La complexité :

Données : n $n \in \mathbb{N}$

S$\leftarrow$ 0
Pour i allant de 1 à n **faire**
    Pour j allant de 1 à n **faire**
        S=S+i+j
retourner S

Complexité : exemple 3

La complexité :
1 affectation

Données : n $n \in \mathbb{N}$

S$\leftarrow$ 0
Pour i allant de 1 à n **faire**
    Pour j allant de 1 à n **faire**
        S=S+i+j
retourner S

Complexité : exemple 3

Données : n $n \in \mathbb{N}$

S$\leftarrow$ 0
Pour i allant de 1 à n **faire**
    Pour j allant de 1 à n **faire**
        S=S+i+j
retourner S

La complexité :
1 affectation
Pour chaque i on fait $2n$ additions et n affectations

Complexité : exemple 3

Données : n $n \in \mathbb{N}$

S$\leftarrow$ 0
Pour i allant de 1 à n **faire**
    Pour j allant de 1 à n **faire**
        S=S+i+j
retourner S

La complexité :
1 affectation
Pour chaque i on fait $2n$ additions et n affectations
Il y aura donc $n \times (3n) + 1$ opérations élémentaires

Complexité : exemple 3

Données : n $n \in \mathbb{N}$

S$\leftarrow$ 0
Pour i allant de 1 à n **faire**
    Pour j allant de 1 à n **faire**
        S=S+i+j
retourner S

La complexité :
1 affectation
Pour chaque i on fait $2n$ additions et n affectations
Il y aura donc $n \times (3n) + 1$ opérations élémentaires
De plus chaque incrémentation de i ou j compte pour une opération élémentaire soit : $n \times n$ opérations supplémentaires
Au total il y a : $4n^2 + 1$ opérations élémentaires. Soit une complexité quadratique notée $\Theta(n^2)$ ou $O(n^2)$