

Performances des algorithmes

S2-2-1 ALGO-II Algorithmes

Germain Gondor

LYCÉE CARNOT - DIJON, 2023 - 2024

Sommaire

- 1 Introduction
- 2 La preuve de terminaison
- 3 La preuve de correction
- 4 La complexité des algorithmes

Sommaire

- 1 Introduction
 - Problématique
 - Cas de l'industrie
- 2 La preuve de terminaison
- 3 La preuve de correction
- 4 La complexité des algorithmes

Sommaire

1 Introduction

- Problématique
- Cas de l'industrie

2 La preuve de terminaison

3 La preuve de correction

4 La complexité 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, ainsi que l'espace mémoire utilisé.

Pourquoi faire ?

Pourquoi ces études sont-elles intéressantes ?

Pourquoi faire ?

Pourquoi ces études sont-elles intéressantes ?

On ne peut prouver un théorème par des exemples.

EXEMPLE : affirmer *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 *les nombres se terminant par 9 sont divisibles par 3* est faux.

Il est vrai que les cas 9, 39, 69 et 1239 fonctionnent mais 19 ne fonctionne pas.

Comment faire alors les tests permettant de mettre en lumière qu'une assertion ou un programme ne fonctionne pas ?

Sommaire

1 Introduction

- Problématique
- Cas de l'industrie

2 La preuve de terminaison

3 La preuve de correction

4 La complexité des algorithmes

Cas de l'industrie

La preuve de programme n'est pas 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.

Cas de l'industrie

La preuve de programme n'est pas 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.

En réalité, dans l'industrie, on fait ce qu'on appelle des tests (ou benchmark) : des scénarios sont créés sur lesquels on teste les logiciels. On en déduit alors une fiabilité du logiciel qui sert d'argument de vente. Cette phase de tests fait parfois apparaître des erreurs étonnantes.

On trouve toutes sortes d'anecdotes sur Internet :

On trouve toutes sortes d'anecdotes sur Internet :

- **problème de terminaison** : Amazon et le livre qui valait 23,7 million de dollars

On trouve toutes sortes d'anecdotes sur Internet :

- **problème de terminaison** : Amazon et le livre qui valait 23,7 million de dollars
- **problème de correction** : Sonde Mars Climate Orbiter

On trouve toutes sortes d'anecdotes sur Internet :

- **problème de terminaison** : Amazon et le livre qui valait 23,7 million de dollars
- **problème de correction** : Sonde Mars Climate Orbiter
- **problème de complexité** : Portail France.fr

On trouve toutes sortes d'anecdotes sur Internet :

- **problème de terminaison** : Amazon et le livre qui valait 23,7 million de dollars
- **problème de correction** : Sonde Mars Climate Orbiter
- **problème de complexité** : Portail France.fr
- **problème de calcul numérique** : Anti-missile Patriot

Sommaire

- 1 Introduction
- 2 **La preuve de terminaison**
 - Tester n'est pas prouver
 - Conjecture de Syracuse
 - Terminaison d'un algorithme
- 3 La preuve de correction
- 4 La complexité des algorithmes

Sommaire

- 1 Introduction
- 2 La preuve de terminaison**
 - Tester n'est pas prouver
 - Conjecture de Syracuse
 - Terminaison d'un algorithme
- 3 La preuve de correction
- 4 La complexité des algorithmes

Tester n'est pas prouver

Il ne s'agit pas de lancer l'implémentation d'un algorithme pour vérifier qu'il se termine. Quand bien même on aurait effectué un très grand nombre d'exécutions ayant toutes terminées, on ne pourrait pas pour autant affirmer que l'algorithme se termine. Il reste toujours la possibilité qu'un ensemble improbable de paramètres fasse échouer le système, et l'entraîne dans une exécution interminable. C'est un **problème de terminaison**.

Tester n'est pas prouver

Il ne s'agit pas de lancer l'implémentation d'un algorithme pour vérifier qu'il se termine. Quand bien même on aurait effectué un très grand nombre d'exécutions ayant toutes terminées, on ne pourrait pas pour autant affirmer que l'algorithme se termine. Il reste toujours la possibilité qu'un ensemble improbable de paramètres fasse échouer le système, et l'entraîne dans une exécution interminable. C'est un **problème de terminaison**.

Le fond des choses est en fait bien plus compliqué qu'il n'y paraît. Un algorithme simple tel que le calcul de la suite de Syracuse pose encore aujourd'hui des problèmes aux mathématiciens.

Sommaire

- 1 Introduction
- 2 La preuve de terminaison**
 - Tester n'est pas prouver
 - Conjecture de Syracuse
 - Terminaison d'un algorithme
- 3 La preuve de correction
- 4 La complexité des algorithmes

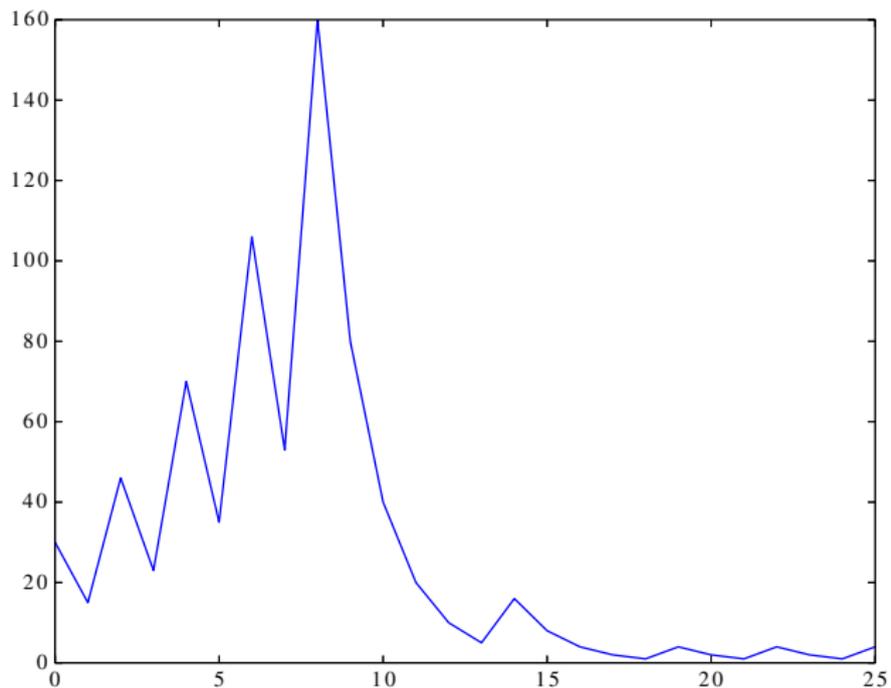
Conjecture de Syracuse

La suite de Syracuse d'un nombre entier N est définie par récurrence de la manière suivante :

$$u_0 = N$$

et pour tout entier $n > 0$:

$$u_{n+1} = \begin{cases} \frac{u_n}{2} & \text{si } u_n \text{ est pair} \\ 3 \cdot u_n + 1 & \text{si } u_n \text{ est impair} \end{cases}$$



$N=30$

Algorithm 1 Suite de Syracuse

donnée: N ; un entier

résultat: syr ; la liste des termes de la suite de Syracuse jusqu'à ce qu'un terme soit égal à 1.

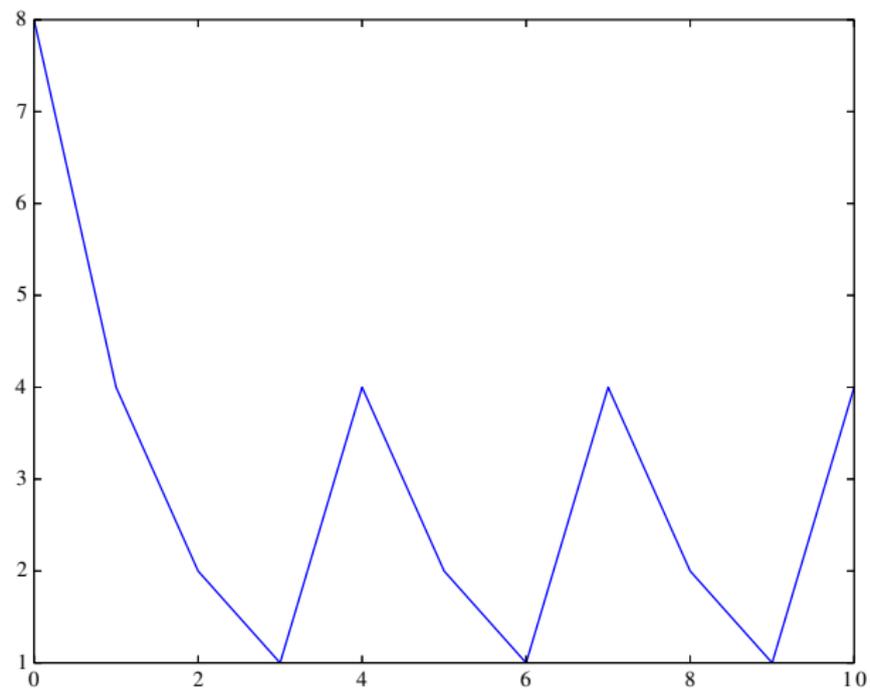
Syracuse(N)

```
1:  $\text{syr} \leftarrow [N]$ 
2:  $i \leftarrow 0$ 
3: tant que  $\text{syr}[i] \neq 1$  faire
4:   si  $\text{syr}[i] \% 2 == 0$  alors
5:      $\text{syr} \leftarrow \text{syr} + [\text{syr}[i] / 2]$ 
6:   sinon
7:      $\text{syr} \leftarrow \text{syr} + [3 * \text{syr}[i] + 1]$ 
8:   fin si
9:    $i \leftarrow i + 1$ 
10: fin tant que
11: renvoi:  $\text{syr}$ 
```

Empiriquement, cet algorithme se termine.

Empiriquement, cet algorithme se termine.

On sait montrer qu'une fois arrivée à 1, la suite calculée a alors un cycle d'ordre 3. Mais on ne sait pas montrer que ce cycle est toujours atteint.

 $N=8$

Sommaire

- 1 Introduction
- 2 La preuve de terminaison**
 - Tester n'est pas prouver
 - Conjecture de Syracuse
 - Terminaison d'un algorithme
- 3 La preuve de correction
- 4 La complexité des algorithmes

Définition

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 composition de deux instructions qui se terminent.

Problématique

Le **problème** vient des boucles *Tant que*.

Problématique

Le **problème** vient des boucles *Tant que*.

L'exemple de la suite de Syracuse le souligne : une boucle *Tant que* s'arrête lorsqu'une condition d'arrêt est réalisée, contrairement à la boucle *Pour* qui s'arrête lorsqu'un nombre prédéfini de répétitions ont été effectuées.

Fonction PGCD

PROBLÈME: *Comment déterminer le Plus Grand Diviseur Commun (PGCD) de deux entiers dont on ne connaît pas la factorisation ?*

Fonction PGCD

PROBLÈME: *Comment déterminer le Plus Grand Diviseur Commun (PGCD) de deux entiers dont on ne connaît pas la factorisation ?*

D'après l'algorithme ci-contre, on définit la fonction $f(u, v) = u \wedge v$.

Fonction PGCD

PROBLÈME: *Comment déterminer le Plus Grand Diviseur Commun (PGCD) de deux entiers dont on ne connaît pas la factorisation ?*

D'après l'algorithme ci-contre, on définit la fonction $f(u, v) = u \wedge v$. Cette fonction est strictement positive et décroissante : il ne peut donc y avoir une infinité d'itérations. On remarque aussi qu'elle tend vers deux fois la valeur finale de u (ou v).

Fonction PGCD

PROBLÈME: *Comment déterminer le Plus Grand Diviseur Commun (PGCD) de deux entiers dont on ne connaît pas la factorisation ?*

D'après l'algorithme ci-contre, on définit la fonction $f(u, v) = u \wedge v$. Cette fonction est strictement positive et décroissante : il ne peut donc y avoir une infinité d'itérations. On remarque aussi qu'elle tend vers deux fois la valeur finale de u (ou v).

REMARQUE : On pourrait aussi raisonner avec la fonction $g(u, v) = \max(u, v)$.

Algorithm 2 Fonction PGCD

donnée: a et b; deux entiers naturels non nuls

résultat: u; le PGCD de a et b

PGCD(a,b)

1: $u \leftarrow a$

2: $v \leftarrow b$

3: **tant que** $u \neq v$ **faire**

4: **si** $u > v$ **alors**

5: $u \leftarrow u - v$

6: **sinon**

7: $v \leftarrow v - u$

8: **fin si**

9: **fin tant que**

renvoi: u

Fonction PGCD : algorithme d'Euclide

PROBLÈME:

Fonction PGCD : algorithme d'Euclide

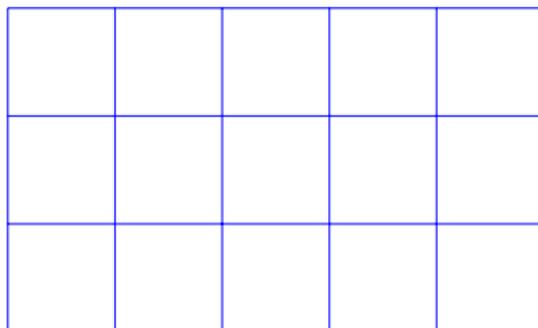
PROBLÈME:

dans la tradition grecque, en comprenant un nombre entier comme une longueur, un couple d'entiers comme un rectangle, leur PGCD est la longueur du côté du plus grand carré permettant de carreler entièrement ce rectangle.

Fonction PGCD : algorithme d'Euclide

PROBLÈME:

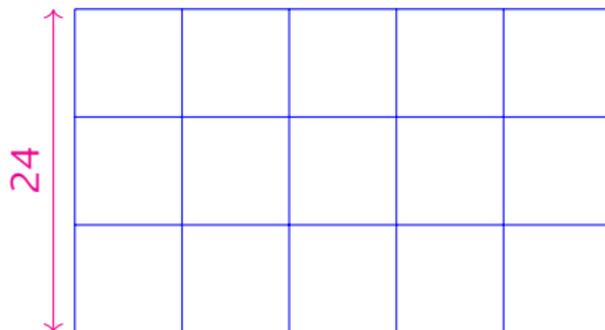
dans la tradition grecque, en comprenant un nombre entier comme une longueur, un couple d'entiers comme un rectangle, leur PGCD est la longueur du côté du plus grand carré permettant de carreler entièrement ce rectangle.



Fonction PGCD : algorithme d'Euclide

PROBLÈME:

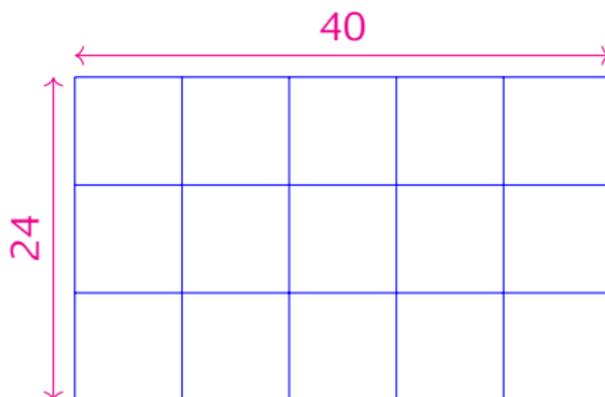
dans la tradition grecque, en comprenant un nombre entier comme une longueur, un couple d'entiers comme un rectangle, leur PGCD est la longueur du côté du plus grand carré permettant de carreler entièrement ce rectangle.



Fonction PGCD : algorithme d'Euclide

PROBLÈME:

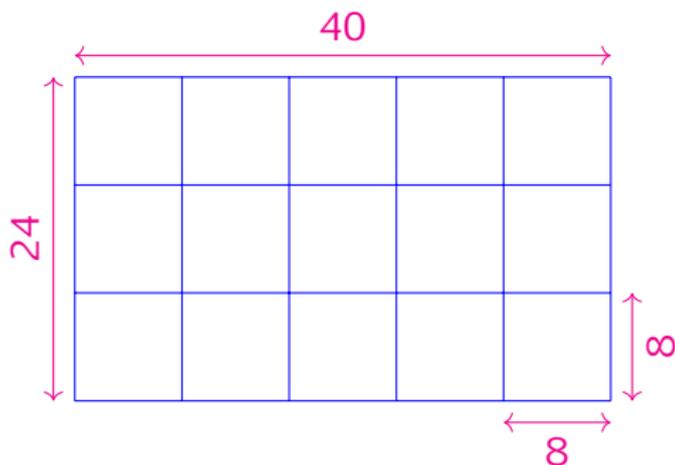
dans la tradition grecque, en comprenant un nombre entier comme une longueur, un couple d'entiers comme un rectangle, leur PGCD est la longueur du côté du plus grand carré permettant de carrelé entièrement ce rectangle.



Fonction PGCD : algorithme d'Euclide

PROBLÈME:

dans la tradition grecque, en comprenant un nombre entier comme une longueur, un couple d'entiers comme un rectangle, leur PGCD est la longueur du côté du plus grand carré permettant de carreler entièrement ce rectangle.



Algorithm 3 Fonction PGCD : algorithme d'Euclide

donnée: a et b; deux entiers naturels non nuls

résultat: le PGCD de a et b

Euclide_PGCD(a,b)

1: **répéter**

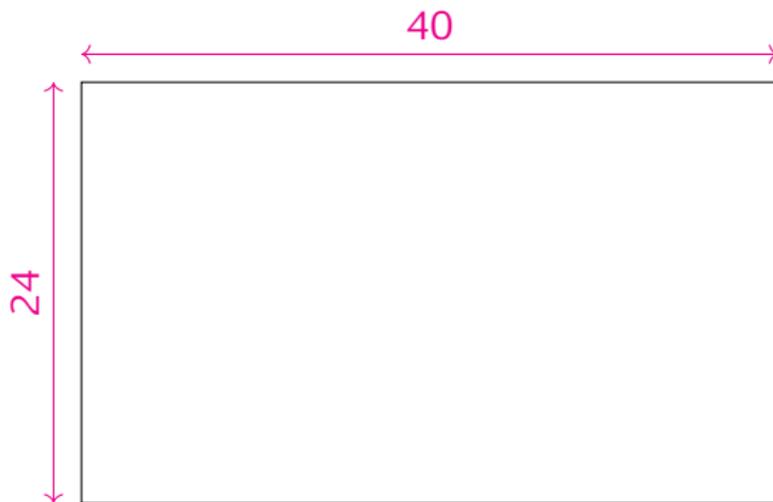
2: $r \leftarrow a \bmod b$

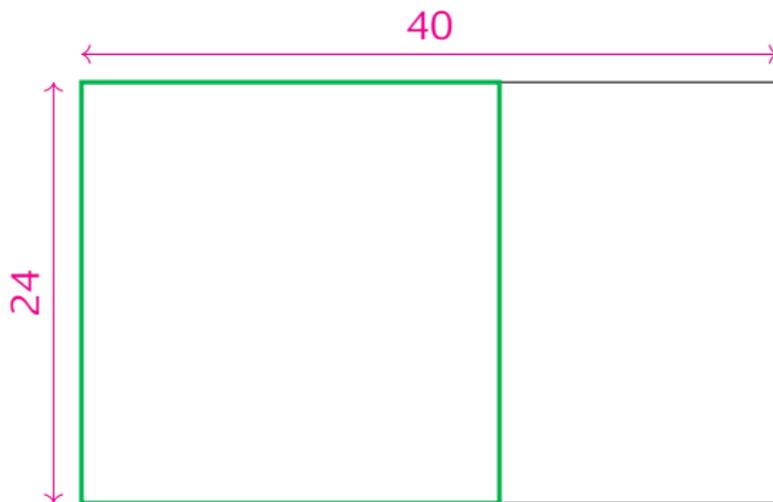
3: $a \leftarrow b$

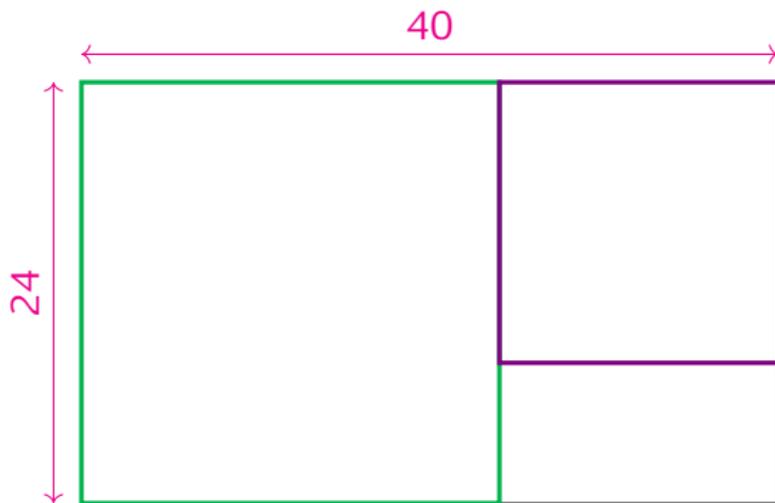
4: $b \leftarrow r$

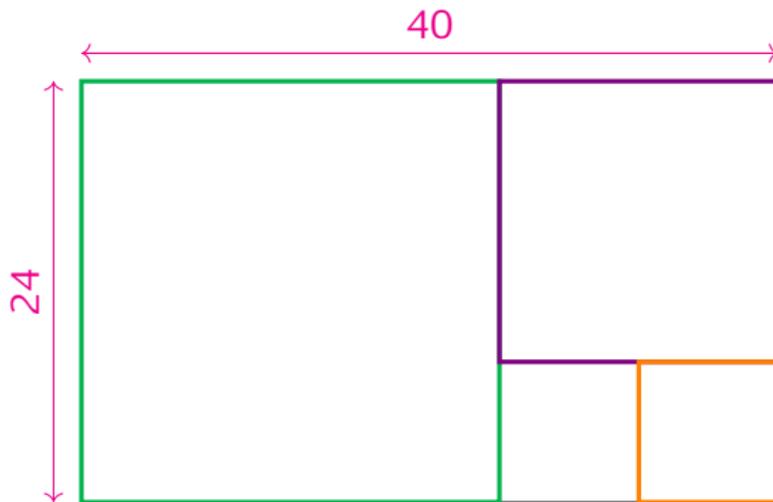
5: **jusqu'à** $r == 0$

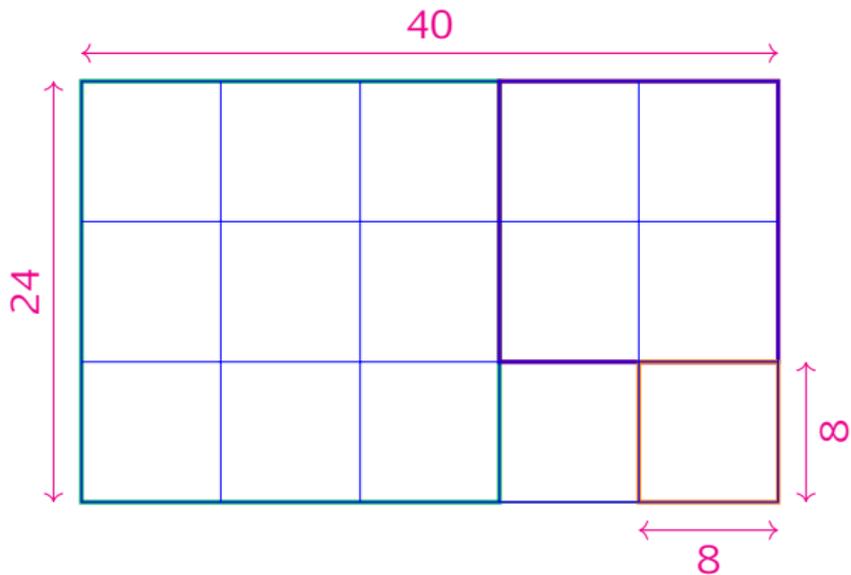
renvoi: a











A la $n^{\text{ième}}$ itération, on a la relation $a_n = q_n \cdot b_n + r_n$ où q_n est un entier.

L'algorithme d'Euclide entraîne cette définition de la suite (a_n) : pour tout n tel que a_{n+1} soit un entier positif, il existe un entier q_n tel que :
 $a_n = q_n \cdot a_{n+1} + a_{n+2}$.

De plus, $0 \leq a_{n+2} < a_{n+1}$ pour tout n tel que a_{n+1} est non nul.

La suite d'entiers naturels est donc strictement décroissante (tant qu'elle est non nulle) à partir du rang 1, et vaut donc 0 à un certain rang.

Il en est donc de même pour la suite (r_n) .

REMARQUE : L'algorithme d'Euclide (algorithme 3) présente l'avantage d'avoir une complexité en temps d'exécution moindre par rapport à l'algorithme initial (algorithme 2).

Variante de boucle

Pour faire une preuve de terminaison, on cherche donc ce que l'on appelle un **variant de boucle** ou encore une **fonction de terminaison**, c'est à dire une suite d'entiers strictement croissante majorée ou de suite d'entiers strictement décroissante minorée.

REMARQUE : le nombre de valeurs prises par la fonction de terminaison est une bonne estimation du nombre de répétitions effectuées lors de l'exécution. Si l'on trouve un majorant de cette fonction, on aura donc un majorant du nombre de tours effectués par la boucle quelque soit l'entrée de l'algorithme. C'est donc un premier pas vers l'étude de la complexité.

Sommaire

- 1 Introduction
- 2 La preuve de terminaison
- 3 La preuve de correction**
 - Problème de correction
 - Invariant de boucle
 - Cas des algorithmes récursifs
- 4 La complexité des algorithmes

Sommaire

- 1 Introduction
- 2 La preuve de terminaison
- 3 La preuve de correction**
 - Problème de correction
 - Invariant de boucle
 - Cas des algorithmes récursifs
- 4 La complexité des algorithmes

Problème de correction

On peut montrer qu'un algorithme se termine bien, sans pour autant qu'il ne donne toujours le bon résultat : c'est un **problème de correction**.

EXEMPLE : Fonction PGCD : algorithme d'Euclide, structure itérative (algorithme 3)

On rappelle qu'à la $n^{\text{ième}}$ itération, on a la relation

$$a_n = q_n \cdot b_n + r_n = q_n \cdot a_{n+1} + a_{n+2}$$

où q_n est un entier.

A la dernière itération N , $r_N = a_{N+2} = 0$. On a donc $a_N = q_N \cdot a_{N+1}$. La relation précédente montre que a_{N+1} divise a_N .

On écrit ensuite que $a_{N-1} = q_{N-1} \cdot a_N + a_{N+1} = (q_{N-1} \cdot q_N + 1) \cdot a_{N+1}$.

On en déduit que a_{N+1} divise aussi a_{N-1} ; puis, de même, et par récurrence, que a_{N+1} divise tous les termes de la suite (a_n) ; en particulier les premiers termes a et b . a_{N+1} est donc bien un diviseur commun de a et b .

Réciproquement, tout diviseur commun de a et b divise aussi tous les termes de la suite (a_n) ; donc en particulier a_{N+1} . a_{N+1} est donc un diviseur commun de a et b , que divise tout autre diviseur commun; c'est bien le $PGCD(a, b)$.

Le PGCD de a_n et a_{n+1} pour tout n entier positif non nul, est ce que l'on appelle un **invariant de boucle**. A chaque itération, c'est une propriété qui est **conservée**.

Sommaire

- 1 Introduction
- 2 La preuve de terminaison
- 3 La preuve de correction**
 - Problème de correction
 - Invariant de boucle
 - Cas des algorithmes récursifs
- 4 La complexité des algorithmes

Invariant de boucle

Un **invariant de boucle** est une propriété :

Invariant de boucle

Un **invariant de boucle** est une propriété :

- qui est vérifiée après la phase d'initialisation,

Invariant de boucle

Un **invariant de boucle** est une propriété :

- qui est vérifiée après la phase d'initialisation,
- qui reste vraie après l'exécution d'une itération,

Invariant de boucle

Un **invariant de boucle** est une propriété :

- qui est vérifiée après la phase d'initialisation,
- qui reste vraie après l'exécution d'une itération,
- qui, conjointement à la condition d'arrêt, permet de montrer que le résultat attendu est bien celui qui est calculé.

Sommaire

- 1 Introduction
- 2 La preuve de terminaison
- 3 La preuve de correction**
 - Problème de correction
 - Invariant de boucle
 - Cas des algorithmes récursifs
- 4 La complexité des algorithmes

Cas des algorithmes récursifs

Pour prouver la correction d'un **algorithme récursif**, on adopte un **raisonnement par récurrence**.

La preuve s'effectue par induction (récurrence) sur la taille du problème à résoudre.

- la base de la récurrence est celle de la récursivité.
- l'étape de la récurrence : on suppose que l'algorithme est correct pour les appels récursifs précédents, et on montre qu'il fonctionne correctement pour l'appel actuel.

EXEMPLE : Fonction PGCD : algorithme d'Euclide, structure récursive

Algorithm 4 Fonction PGCD : algorithme d'Euclide récursifs

donnée: a et b; deux entiers naturels non nuls tels que $a \geq b$

résultat: le PGCD de a et b

Euclide_PGCD_récurif(a,b)

- 1: **si** $b == 0$ **alors**
 - 2: **renvoi:** a
 - 3: **sinon**
 - 4: **renvoi:** Euclide_PGCD_récurif(b, a mod b)
 - 5: **fin si**
-

- au dernier appel N de la fonction *Euclide_PGCD_récurif*, $b_N = 0$.
 a_N est donc le $PGCD(a_N, 0)$.

- au dernier appel N de la fonction *Euclide_PGCD_récurusif*, $b_N = 0$. a_N est donc le $PGCD(a_N, 0)$.
- on suppose qu'au $n^{\text{ième}}$ appel de la fonction *Euclide_PGCD_recurusif*, a_n est le $PGCD(a_n, b_n)$.

- au dernier appel N de la fonction *Euclide_PGCD_récurusif*, $b_N = 0$. a_N est donc le $PGCD(a_N, 0)$.
- on suppose qu'au $n^{\text{ième}}$ appel de la fonction *Euclide_PGCD_récurusif*, a_N est le $PGCD(a_n, b_n)$.
- à l'appel $n - 1$ de la fonction *Euclide_PGCD_récurusif* :

$$a_{n-1} = q_{n-1} \cdot b_{n-1} + r_{n-1} = q_{n-1} \cdot a_n + b_n$$

Or a_N étant le $PGCD(a_n, b_n)$, il est aussi plus grand diviseur de (a_{n-1}, a_n) , c'est-à-dire que $a_N = PGCD(a_{n-1}, a_n)$ et donc $a_N = PGCD(a_{n-1}, b_{n-1})$.

- au dernier appel N de la fonction *Euclide_PGCD_récurusif*, $b_N = 0$. a_N est donc le $PGCD(a_N, 0)$.
- on suppose qu'au $n^{\text{ième}}$ appel de la fonction *Euclide_PGCD_récurusif*, a_N est le $PGCD(a_n, b_n)$.
- à l'appel $n - 1$ de la fonction *Euclide_PGCD_récurusif* :

$$a_{n-1} = q_{n-1} \cdot b_{n-1} + r_{n-1} = q_{n-1} \cdot a_n + b_n$$

Or a_N étant le $PGCD(a_n, b_n)$, il est aussi plus grand diviseur de (a_{n-1}, a_n) , c'est-à-dire que $a_N = PGCD(a_{n-1}, a_n)$ et donc $a_N = PGCD(a_{n-1}, b_{n-1})$.

- on en déduit par récurrence que $a_N = PGCD(a, b)$.

Sommaire

- 1 Introduction
- 2 La preuve de terminaison
- 3 La preuve de correction
- 4 La complexité des algorithmes**
 - Complexité en temps et en mémoire
 - Opérations élémentaires
 - Classes de complexité
 - Méthodes de calcul de la complexité

Sommaire

- 1 Introduction
- 2 La preuve de terminaison
- 3 La preuve de correction
- 4 **La complexité des algorithmes**
 - Complexité en temps et en mémoire
 - Opérations élémentaires
 - Classes de complexité
 - Méthodes de calcul de la complexité

Complexité en temps et en mémoire

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

Complexité en temps et en mémoire

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

Cette efficacité peut être évaluée par :

Complexité en temps et en mémoire

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 entrées.

Complexité en temps et en mémoire

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 entrées.
- les ressources nécessaires à son exécution : c'est la **complexité en mémoire** (cf cours Pivot de Gauss).

On se focalise donc par la suite sur l'étude la complexité en temps d'exécution, i.e. en **nombre d'opérations élémentaires**.

On se focalise donc par la suite sur l'étude la complexité en temps d'exécution, i.e. en **nombre d'opérations élémentaires**.

Nous étudions donc le nombre d'opérations élémentaires de manière asymptotique avec les notions mathématiques de Landau : $O()$, $\Theta()$.

On se focalise donc par la suite sur l'étude la complexité en temps d'exécution, i.e. en **nombre d'opérations élémentaires**.

Nous étudions donc le nombre d'opérations élémentaires de manière asymptotique avec les notions mathématiques de Landau : $O()$, $\Theta()$.

On suppose que $f(n) \geq 0$ et $g(n) \geq 0$ pour tout entier n :

On se focalise donc par la suite sur l'étude la complexité en temps d'exécution, i.e. en **nombre d'opérations élémentaires**.

Nous étudions donc le nombre d'opérations élémentaires de manière asymptotique avec les notions mathématiques de Landau : $O()$, $\Theta()$.

On suppose que $f(n) \geq 0$ et $g(n) \geq 0$ pour tout entier n :

- f est appelée un *grand \hat{O}* de g et on note $f = O(g)$ lorsqu'il existe un rang n_0 et une constante $c > 0$ tels que $\forall n \geq n_0, f(n) \leq c.g(n)$

On se focalise donc par la suite sur l'étude la complexité en temps d'exécution, i.e. en **nombre d'opérations élémentaires**.

Nous étudions donc le nombre d'opérations élémentaires de manière asymptotique avec les notions mathématiques de Landau : $O()$, $\Theta()$.

On suppose que $f(n) \geq 0$ et $g(n) \geq 0$ pour tout entier n :

- f est appelée un *grand O* de g et on note $f = O(g)$ lorsqu'il existe un rang n_0 et une constante $c > 0$ tels que $\forall n \geq n_0, f(n) \leq c.g(n)$
- f est appelée un *theta* de g et on note $f = \Theta(g)$ lorsqu'il existe un rang n_0 et une constante $c_1, c_2 > 0$ tels que $\forall n \geq n_0, c_1.g(n) \leq f(n) \leq c_2.g(n)$

On se focalise donc par la suite sur l'étude la complexité en temps d'exécution, i.e. en **nombre d'opérations élémentaires**.

Nous étudions donc le nombre d'opérations élémentaires de manière asymptotique avec les notions mathématiques de Landau : $O()$, $\Theta()$.

On suppose que $f(n) \geq 0$ et $g(n) \geq 0$ pour tout entier n :

- f est appelée un *grand O* de g et on note $f = O(g)$ lorsqu'il existe un rang n_0 et une constante $c > 0$ tels que $\forall n \geq n_0, f(n) \leq c.g(n)$
- f est appelée un *theta* de g et on note $f = \Theta(g)$ lorsqu'il existe un rang n_0 et une constante $c_1, c_2 > 0$ tels que $\forall n \geq n_0, c_1.g(n) \leq f(n) \leq c_2.g(n)$

Tout le travail de l'étude de complexité repose donc sur l'identification des opérations non élémentaires et sur le calcul de leur nombre.

Sommaire

- 1 Introduction
- 2 La preuve de terminaison
- 3 La preuve de correction
- 4 La complexité des algorithmes**
 - Complexité en temps et en mémoire
 - Opérations élémentaires
 - Classes de complexité
 - Méthodes de calcul de la complexité

Opérations élémentaires

On considère comme élémentaires les opérations :

Opérations élémentaires

On considère comme élémentaires les opérations :

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

Opérations élémentaires

On considère comme élémentaires les opérations :

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

Opérations élémentaires

On considère comme élémentaires les opérations :

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

Opérations élémentaires

On considère comme élémentaires 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

Opérations élémentaires

On considère comme élémentaires 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
- taille d'un tableau

Opérations élémentaires

On considère comme élémentaires 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
- taille d'un tableau
- La méthode append sur une liste Python peut être considérée comme une opération basique.

Opérations élémentaires

On considère comme élémentaires 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
- 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*

Opérations élémentaires

On considère comme élémentaires 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
- 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

Opérations élémentaires

On considère comme élémentaires 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
- 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

Si l'on note $T(P)$ la complexité d'une instruction P , alors pour deux instructions P et Q , on a : $T(P; Q) = T(P) + T(Q)$.

Somme géométrique

PROBLÈME: *On souhaite calculer la somme des n premiers entiers.*

Somme géométrique

PROBLÈME: *On souhaite calculer la somme des n premiers entiers.*

Algorithm 6 Somme des n premiers entiers

donnée: n , un entier

résultat: S ; la somme des n premiers entiers

Somme(n)

1: $S \leftarrow 0$

2: **pour** i de 1 à n **faire**

3: $S \leftarrow S + i$

4: **fin pour**

5: **renvoi:** S

Somme géométrique

PROBLÈME: *On souhaite calculer la somme des n premiers entiers.*

Algorithm 7 Somme des n premiers entiers

donnée: n , un entier

résultat: S ; la somme des n premiers entiers

Somme(n)

1: $S \leftarrow 0$

2: **pour** i de 1 à n **faire**

3: $S \leftarrow S + i$

4: **fin pour**

5: **renvoi:** S

Dans l'algorithme ci-dessus, on ne considère que le coût t de la somme $S+i$ effectuée à chaque itération de la boucle *Pour*.

Somme géométrique

PROBLÈME: *On souhaite calculer la somme des n premiers entiers.*

Algorithm 8 Somme des n premiers entiers

donnée: n , un entier

résultat: S ; la somme des n premiers entiers

Somme(n)

- 1: $S \leftarrow 0$
 - 2: **pour** i de 1 à n **faire**
 - 3: $S \leftarrow S + i$
 - 4: **fin pour**
 - 5: **renvoi:** S
-

Dans l'algorithme ci-dessus, on ne considère que le coût t de la somme $S+i$ effectuée à chaque itération de la boucle *Pour*.

Ainsi le coût total est de la forme $C(n) = n.t$. On dira que la complexité est du type : $C(n) = \Theta(n)$.

Somme géométrique

PROBLÈME: *On souhaite calculer la somme des n premiers entiers.*

Algorithm 9 Somme des n premiers entiers

donnée: n , un entier

résultat: S ; la somme des n premiers entiers

Somme(n)

- 1: $S \leftarrow 0$
 - 2: **pour** i de 1 à n **faire**
 - 3: $S \leftarrow S+i$
 - 4: **fin pour**
 - 5: **renvoi:** S
-

Dans l'algorithme ci-dessus, on ne considère que le coût t de la somme $S+i$ effectuée à chaque itération de la boucle *Pour*.

Ainsi le coût total est de la forme $C(n) = n.t$. On dira que la complexité est du type : $C(n) = \Theta(n)$.

Elle croît linéairement avec le nombre d'entiers n .

Sommaire

- 1 Introduction
- 2 La preuve de terminaison
- 3 La preuve de correction
- 4 La complexité des algorithmes**
 - Complexité en temps et en mémoire
 - Opérations élémentaires
 - Classes de complexité
 - Méthodes de calcul de la complexité

Classes de complexité

On définit différentes classes de complexité :

Notation	Type de complexité
$\Theta(1)$	Complexité constante
$\Theta(\ln(n))$	Complexité logarithmique
$\Theta(n)$	Complexité linéaire
$\Theta(n \cdot \ln(n))$	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



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

Opération Python	Coût moyen
Copy	$O(n)$
Append	$O(1)$
Insert	$O(n)$
Obtenir un élément	$O(1)$
Changer un élément	$O(1)$
Supprimer un élément	$O(n)$
Itération	$O(n)$
Obtenir une sélection	$O(k)$
Supprimer une sélection	$O(n)$
Changer une sélection	$O(k+n)$
Extend	$O(k)$
Trier	$O(n \cdot \log(n))$
Multiplier	$O(nk)$
x in s	$O(n)$
$\min(s)$, $\max(s)$	$O(n)$
Length	$O(1)$

Il apparaît deux choses :

Il apparaît deux choses :

- il faut éviter les complexités supérieures à la complexité polynomiale d'ordre 2;

Il apparaît deux choses :

- il faut éviter les complexités supérieures à la complexité polynomiale d'ordre 2;
- il ne faut absolument pas compter sur les avancées technologiques !

Nous étudierons, lorsque l'exemple s'y prêtera, trois types de complexités en temps :

Nous étudierons, lorsque l'exemple s'y prêtera, trois types de complexités en temps :

- **la complexité dans le pire cas, C_{max}** : c'est un majorant du temps d'exécution sur toutes les entrées possibles d'une même taille ; la complexité dans le pire cas apporte une notion de sécurité sur le temps d'exécution ;

Nous étudierons, lorsque l'exemple s'y prêtera, trois types de complexités en temps :

- **la complexité dans le pire cas**, C_{max} : c'est un majorant du temps d'exécution sur toutes les entrées possibles d'une même taille ; la complexité dans le pire cas apporte une notion de sécurité sur le temps d'exécution ;
- **la complexité en moyenne**, C_{moy} : il s'agit d'une moyenne sur le temps d'exécution sur toutes les entrées possibles d'une même taille en considérant une distribution équiprobable ; elle représente le comportement moyen d'un algorithme ;

Nous étudierons, lorsque l'exemple s'y prêtera, trois types de complexités en temps :

- **la complexité dans le pire cas**, C_{max} : c'est un majorant du temps d'exécution sur toutes les entrées possibles d'une même taille ; la complexité dans le pire cas apporte une notion de sécurité sur le temps d'exécution ;
- **la complexité en moyenne**, C_{moy} : il s'agit d'une moyenne sur le temps d'exécution sur toutes les entrées possibles d'une même taille en considérant une distribution équiprobable ; elle représente le comportement moyen d'un algorithme ;
- **la complexité dans le meilleur cas**, C_{min} : elle est peu utile, facile à calculer, mais n'apporte qu'une borne inférieure sur les temps d'exécution.

Sommaire

- 1 Introduction
- 2 La preuve de terminaison
- 3 La preuve de correction
- 4 La complexité des algorithmes**
 - Complexité en temps et en mémoire
 - Opérations élémentaires
 - Classes de complexité
 - Méthodes de calcul de la complexité

Méthode itérative

On **somme** membre à membre et on en déduit la complexité :

EXEMPLE : avec une récurrence $C(n) = C(n/2) + 1$

Méthode itérative

On **somme** membre à membre et on en déduit la complexité :

EXEMPLE : avec une récurrence $C(n) = C(n/2) + 1$

Méthode itérative

On **somme** membre à membre et on en déduit la complexité :

EXEMPLE : avec une récurrence $C(n) = C(n/2) + 1$

$$1 \quad | \quad C(n) = C(n/2) + 1$$

Méthode itérative

On **somme** membre à membre et on en déduit la complexité :

EXEMPLE : avec une récurrence $C(n) = C(n/2) + 1$

$$\begin{array}{l|l} 1 & C(n) = C(n/2) + 1 \\ 2 & C(n/2) = C(n/4) + 1 \end{array}$$

Méthode itérative

On **somme** membre à membre et on en déduit la complexité :

EXEMPLE : avec une récurrence $C(n) = C(n/2) + 1$

$$\begin{array}{l|l} 1 & C(n) = C(n/2) + 1 \\ 2 & C(n/2) = C(n/4) + 1 \\ & \dots \end{array}$$

Méthode itérative

On **somme** membre à membre et on en déduit la complexité :

EXEMPLE : avec une récurrence $C(n) = C(n/2) + 1$

$$\begin{array}{l|l} 1 & C(n) = C(n/2) + 1 \\ 2 & C(n/2) = C(n/4) + 1 \\ & \dots \\ i & C(n/(2^{i-1})) = C(n/(2^i)) + 1 \end{array}$$

Méthode itérative

On **somme** membre à membre et on en déduit la complexité :

EXEMPLE : avec une récurrence $C(n) = C(n/2) + 1$

$$\begin{array}{l|l} 1 & C(n) = C(n/2) + 1 \\ 2 & C(n/2) = C(n/4) + 1 \\ & \dots \\ i & C(n/(2^{i-1})) = C(n/(2^i)) + 1 \\ & \dots \end{array}$$

Méthode itérative

On **somme** membre à membre et on en déduit la complexité :

EXEMPLE : avec une récurrence $C(n) = C(n/2) + 1$

1	$C(n) = C(n/2) + 1$
2	$C(n/2) = C(n/4) + 1$
	...
i	$C(n/(2^{i-1})) = C(n/(2^i)) + 1$
	...
m	$C(1) = \cancel{C(0)} + 1$

Méthode itérative

On **somme** membre à membre et on en déduit la complexité :

EXEMPLE : avec une récurrence $C(n) = C(n/2) + 1$

1	$C(n) = C(n/2) + 1$
2	$C(n/2) = C(n/4) + 1$
	...
i	$C(n/(2^{i-1})) = C(n/(2^i)) + 1$
	...
m	$C(1) = \cancel{C(0)} + 1$
	$C(n) = m$

Méthode itérative

On somme membre à membre et on en déduit la complexité :

EXEMPLE : avec une récurrence $C(n) = C(n/2) + 1$

1	$C(n) = C(n/2) + 1$
2	$C(n/2) = C(n/4) + 1$
	...
i	$C(n/(2^{i-1})) = C(n/(2^i)) + 1$
	...
m	$C(1) = \cancel{C(0)} + 1$
	$C(n) = m$

Or m est telle que $\frac{n}{2^{m-1}} = 1$ ce qui entraîne $n = 2^{m-1}$ soit $m = \log_2 n + 1$

Méthode itérative

On somme membre à membre et on en déduit la complexité :

EXEMPLE : avec une récurrence $C(n) = C(n/2) + 1$

1	$C(n) = C(n/2) + 1$
2	$C(n/2) = C(n/4) + 1$
	...
i	$C(n/(2^{i-1})) = C(n/(2^i)) + 1$
	...
m	$C(1) = C(1) + 1$
	$C(n) = m$

Or m est telle que $\frac{n}{2^{m-1}} = 1$ ce qui entraîne $n = 2^{m-1}$ soit $m = \log_2 n + 1$

$$\Rightarrow C(n) = O(\ln(n))$$

EXEMPLE : avec une récurrence $C(n) = C(n - 1) + 1$

EXEMPLE : avec une récurrence $C(n) = C(n - 1) + 1$

EXEMPLE : avec une récurrence $C(n) = C(n - 1) + 1$

$$1 \mid \quad C(n) = C(n - 1) + 1$$

EXEMPLE : avec une récurrence $C(n) = C(n - 1) + 1$

$$\begin{array}{l|l} 1 & C(n) = C(n-1) + 1 \\ 2 & C(n-1) = C(n-2) + 1 \end{array}$$

EXEMPLE : avec une récurrence $C(n) = C(n-1) + 1$

$$\begin{array}{l|l} 1 & C(n) = C(n-1) + 1 \\ 2 & C(n-1) = C(n-2) + 1 \\ & \dots \end{array}$$

EXEMPLE : avec une récurrence $C(n) = C(n - 1) + 1$

$$\begin{array}{l|l} 1 & C(n) = C(n-1) + 1 \\ 2 & C(n-1) = C(n-2) + 1 \\ & \dots \\ i & C(n-(i-1)) = C(n-i) + 1 \end{array}$$

EXEMPLE : avec une récurrence $C(n) = C(n - 1) + 1$

$$\begin{array}{l|l} 1 & C(n) = C(n-1) + 1 \\ 2 & C(n-1) = C(n-2) + 1 \\ & \dots \\ i & C(n-(i-1)) = C(n-i) + 1 \\ & \dots \end{array}$$

EXEMPLE : avec une récurrence $C(n) = C(n - 1) + 1$

1	$C(n) = C(n - 1) + 1$
2	$C(n - 1) = C(n - 2) + 1$
	...
i	$C(n - (i - 1)) = C(n - i) + 1$
	...
n	$C(n - (n - 1)) = \cancel{C(0)} + 1$

EXEMPLE : avec une récurrence $C(n) = C(n-1) + 1$

$$\begin{array}{l|l}
 1 & C(n) = C(n-1) + 1 \\
 2 & C(n-1) = C(n-2) + 1 \\
 & \dots \\
 i & C(n-(i-1)) = C(n-i) + 1 \\
 & \dots \\
 n & C(n-(n-1)) = \cancel{C(0)} + 1 \\
 \hline
 & C(n) = n
 \end{array}$$

$$\Rightarrow C(n) = O(n)$$

Méthode par substitution

on vérifie une **intuition** :

Méthode par substitution

on vérifie une **intuition** :

- hypothèse par intuition $C(n) = g(n)$

Méthode par substitution

on vérifie une **intuition** :

- hypothèse par intuition $C(n) = g(n)$
- démontrer que $a.g(n/b) + f(n) = g(n)$ avec $g(1) = c$ en fixant les constantes

Exemple : $C(n) = C(n/2) + 1$ avec $C(1) = 1$

Exemple : $C(n) = C(n/2) + 1$ avec $C(1) = 1$

- intuition : $C(n) = \Theta(\ln(n))$

Exemple : $C(n) = C(n/2) + 1$ avec $C(1) = 1$

- intuition : $C(n) = \Theta(\ln(n))$
- on pose alors : $C(n) = a \cdot \log_2(n) + c$ avec a et c à déterminer.

Exemple : $C(n) = C(n/2) + 1$ avec $C(1) = 1$

- intuition : $C(n) = \Theta(\ln(n))$
- on pose alors : $C(n) = a \cdot \log_2(n) + c$ avec a et c à déterminer.
- on cherche a et c par substitution

Exemple : $C(n) = C(n/2) + 1$ avec $C(1) = 1$

- intuition : $C(n) = \Theta(\ln(n))$
- on pose alors : $C(n) = a \cdot \log_2(n) + c$ avec a et c à déterminer.
- on cherche a et c par substitution

$$C(n) = a \cdot \log_2(n) + c$$

$$C(n/2) = a \cdot \log_2(n/2) + c = a \cdot \log_2(n) - a \cdot \log_2(2) + c$$

Exemple : $C(n) = C(n/2) + 1$ avec $C(1) = 1$

- intuition : $C(n) = \Theta(\ln(n))$
- on pose alors : $C(n) = a \cdot \log_2(n) + c$ avec a et c à déterminer.
- on cherche a et c par substitution

$$C(n) = a \cdot \log_2(n) + c$$

$$C(n/2) = a \cdot \log_2(n/2) + c = a \cdot \log_2(n) - a \cdot \log_2(2) + c$$

$$C(n) = C(n/2) + 1$$

$$\Rightarrow a \cdot \log_2(n) + c = a \cdot \log_2(n) - a + c + 1$$

$$\Rightarrow a = 1$$

Exemple : $C(n) = C(n/2) + 1$ avec $C(1) = 1$

- intuition : $C(n) = \Theta(\ln(n))$
- on pose alors : $C(n) = a \cdot \log_2(n) + c$ avec a et c à déterminer.
- on cherche a et c par substitution

$$C(n) = a \cdot \log_2(n) + c$$

$$C(n/2) = a \cdot \log_2(n/2) + c = a \cdot \log_2(n) - a \cdot \log_2(2) + c$$

$$C(n) = C(n/2) + 1$$

$$\Rightarrow a \cdot \log_2(n) + c = a \cdot \log_2(n) - a + c + 1$$

$$\Rightarrow a = 1$$

Or $1 = C(1) = \log_2(1) + c \Rightarrow c = 1$. Ainsi :

$$C(n) = \log_2(n) + 1$$

Méthode générale

on utilise les propriétés des **suites récurrentes linéaires**.

Méthode générale

on utilise les propriétés des **suites récurrentes linéaires**.

On peut toujours écrire une relation de récurrence de la forme :

$$C(1) = c$$

$$C(n) = a.C(n/b) + f(n)$$

Relation Solution	Exemples

Relation Solution	Exemples
$C(n) = C(n-1) + b$ $C(n) = C(0) + b.n = \Theta(n)$	factorielle, recherche séquentielle (balayage) récursive dans un tableau

Relation Solution	Exemples
$C(n) = C(n-1) + b$ $C(n) = C(0) + b.n = \Theta(n)$	factorielle, recherche séquentielle (balayage) récursive dans un tableau
$C(n) = a.C(n-1) + b, a \neq 1$ $C(n) = a^n \cdot \left(C(0) - \frac{b}{1-a} \right) + \frac{b}{1-a}$ $= \Theta(a^n)$	répétition a fois d'un traitement sur le résultat de l'appel récursif

Relation Solution	Exemples
$C(n) = C(n-1) + b$ $C(n) = C(0) + b.n = \Theta(n)$	factorielle, recherche séquentielle (balayage) récursive dans un tableau
$C(n) = a.C(n-1) + b, a \neq 1$ $C(n) = a^n \cdot \left(C(0) - \frac{b}{1-a} \right) + \frac{b}{1-a}$ $= \Theta(a^n)$	répétition a fois d'un traitement sur le résultat de l'appel récursif
$C(n) = C(n-1) + a.n + b$ $C(n) = C(0) + a.n.(n+1)/2 + n.b = \Theta(n^2)$	traitement en coût linéaire avant l'appel récursif, tri à bulle

Relation Solution	Exemples
$C(n) = C(n-1) + b$ $C(n) = C(0) + b.n = \Theta(n)$	factorielle, recherche séquentielle (balayage) récursive dans un tableau
$C(n) = a.C(n-1) + b, a \neq 1$ $C(n) = a^n \cdot \left(C(0) - \frac{b}{1-a} \right) + \frac{b}{1-a}$ $= \Theta(a^n)$	répétition a fois d'un traitement sur le résultat de l'appel récursif
$C(n) = C(n-1) + a.n + b$ $C(n) = C(0) + a.n.(n+1)/2 + n.b = \Theta(n^2)$	traitement en coût linéaire avant l'appel récursif, tri à bulle
$C(n) = C(n/2) + b$ $C(n) = C(1) + b.\log_2(n) = \Theta(\ln(n))$	élimination de la moitié des éléments en temps constant avant l'appel récursif, recherche dichotomique récursive, exponentiation rapide

Relation Solution	Exemples
$C(n) = C(n-1) + b$ $C(n) = C(0) + b.n = \Theta(n)$	factorielle, recherche séquentielle (balayage) récursive dans un tableau
$C(n) = a.C(n-1) + b, a \neq 1$ $C(n) = a^n \cdot \left(C(0) - \frac{b}{1-a} \right) + \frac{b}{1-a}$ $= \Theta(a^n)$	répétition a fois d'un traitement sur le résultat de l'appel récursif
$C(n) = C(n-1) + a.n + b$ $C(n) = C(0) + a.n.(n+1)/2 + n.b = \Theta(n^2)$	traitement en coût linéaire avant l'appel récursif, tri à bulle
$C(n) = C(n/2) + b$ $C(n) = C(1) + b.\log_2(n) = \Theta(\ln(n))$	élimination de la moitié des éléments en temps constant avant l'appel récursif, recherche dichotomique récursive, exponentiation rapide
$C(n) = a.C(n/2) + b, a \neq 1$ $C(n) = n^{\log_2(a)} \cdot \left(C(1) - \frac{b}{1-a} \right) + \frac{b}{1-a}$ $= \Theta(n^{\log_2(a)})$	répétition a fois d'un traitement sur le résultat de l'appel récursif dichotomique

Relation Solution	Exemples
$C(n) = C(n-1) + b$ $C(n) = C(0) + b.n = \Theta(n)$	factorielle, recherche séquentielle (balayage) récursive dans un tableau
$C(n) = a.C(n-1) + b, a \neq 1$ $C(n) = a^n \cdot \left(C(0) - \frac{b}{1-a} \right) + \frac{b}{1-a}$ $= \Theta(a^n)$	répétition a fois d'un traitement sur le résultat de l'appel récursif
$C(n) = C(n-1) + a.n + b$ $C(n) = C(0) + a.n.(n+1)/2 + n.b = \Theta(n^2)$	traitement en coût linéaire avant l'appel récursif, tri à bulle
$C(n) = C(n/2) + b$ $C(n) = C(1) + b.\log_2(n) = \Theta(\ln(n))$	élimination de la moitié des éléments en temps constant avant l'appel récursif, recherche dichotomique récursive, exponentiation rapide
$C(n) = a.C(n/2) + b, a \neq 1$ $C(n) = n^{\log_2(a)} \cdot \left(C(1) - \frac{b}{1-a} \right) + \frac{b}{1-a}$ $= \Theta(n^{\log_2(a)})$	répétition a fois d'un traitement sur le résultat de l'appel récursif dichotomique
$C(n) = C(n/2) + a.n + b$ $C(n) = \Theta(n)$	traitement linéaire avant l'appel récursif dichotomique

Relation Solution	Exemples
$C(n) = C(n-1) + b$ $C(n) = C(0) + b.n = \Theta(n)$	factorielle, recherche séquentielle (balayage) récursive dans un tableau
$C(n) = a.C(n-1) + b, a \neq 1$ $C(n) = a^n \cdot \left(C(0) - \frac{b}{1-a} \right) + \frac{b}{1-a}$ $= \Theta(a^n)$	répétition a fois d'un traitement sur le résultat de l'appel récursif
$C(n) = C(n-1) + a.n + b$ $C(n) = C(0) + a.n.(n+1)/2 + n.b = \Theta(n^2)$	traitement en coût linéaire avant l'appel récursif, tri à bulle
$C(n) = C(n/2) + b$ $C(n) = C(1) + b.\log_2(n) = \Theta(\ln(n))$	élimination de la moitié des éléments en temps constant avant l'appel récursif, recherche dichotomique récursive, exponentiation rapide
$C(n) = a.C(n/2) + b, a \neq 1$ $C(n) = n^{\log_2(a)} \cdot \left(C(1) - \frac{b}{1-a} \right) + \frac{b}{1-a}$ $= \Theta(n^{\log_2(a)})$	répétition a fois d'un traitement sur le résultat de l'appel récursif dichotomique
$C(n) = C(n/2) + a.n + b$ $C(n) = \Theta(n)$	traitement linéaire avant l'appel récursif dichotomique
$C(n) = 2.C(n/2) + a.n + b$ $C(n) = \Theta(n.\ln(n))$	traitement linéaire avant double appel récursif dichotomique, tri fusion