Structures de données : cours 1 – Analyse d’algorithmes

Elena Leroux

Description du cours

- **Objectifs du cours :**
  - Acquérir les fondements théoriques et pratiques des structures de données et des algorithmes qui leur sont associés.
  - L'accent sera mis sur les propriétés fondamentales de ces structures et l'étude de leur complexité, sans toutefois négliger les aspects reliés à leurs implantations.

Description du cours

- **Objectifs du cours :**
  - Acquérir les fondements théoriques et pratiques des structures de données et des algorithmes qui leur sont associés.
  - L'accent sera mis sur les propriétés fondamentales de ces structures et l'étude de leur complexité, sans toutefois négliger les aspects reliés à leurs implantations.

- **Prérequis :**
  - Vous avez des connaissances dans les structures des données de base.
  - Java servira de langage de programmation pour les TD de mise en œuvre des structures de données. Vous êtes donc supposés savoir déjà programmer en Java.

Bibliographie

- **Structures de données :**

  - Michael Goodrich and Roberto Tamassia, « Data Structures and Algorithms in Java » , John Wiley & Sons, 2006.
  - Thomas Cormen, Charles Leiserson, Ronald Rivest « Introduction à l’algorithmique », Dunod, 1994.
  - Mark Allen Weiss, « Data Structures and Problem Solving Using Java », Addison-Wesley, 1998.
  - Michael Goodrich and Roberto Tamassia, « Algorithm Design : Foundations, Analysis, and Internet Examples », John Wiley & Sons.

Bibliographie

- **Structures de données :**
  - Michael Goodrich and Roberto Tamassia, « Data Structures and Algorithms in Java » , John Wiley & Sons, 2006.
  - Thomas Cormen, Charles Leiserson, Ronald Rivest « Introduction à l'algorithmique », Dunod, 1994.
  - Mark Allen Weiss, « Data Structures and Problem Solving Using Java », Addison-Wesley, 1998.
  - Michael Goodrich and Roberto Tamassia, « Algorithm Design : Foundations, Analysis, and Internet Examples », John Wiley & Sons.

- **Java :**
  - Kathy Sierra and Bert Bates, « Head First Java », O'Reilly, 2005.
  - David Arnow and Gerald Weiss, « Introduction to Programming Using Java: An Object-Oriented Approach », Addison-Wesley, 2000.

Organisation du cours

- **CM 1 :** Complexité et analyse d’un algorithme

Organisation du cours

- **CM 1 :** Complexité et analyse d’un algorithme
- **CM 2 :** Structures de données linéaires
  - piles, files, listes chaînées : différentes implantations et applications
  - tables de hachage

Organisation du cours

- CM 1 : Complexité et analyse d'un algorithme
- CM 2 : Structures de données linéaires
  - piles, files, listes chaînées : différentes implantations et applications
  - tables de hachage
- CM 3 : Structures de données arborescentes
  - applications et implantations
  - algorithmes de parcours
  - arbres binaires de recherche
  - arbres équilibrés :
    - AVL, rouge et noire

Organisation du cours

- CM 1 : Complexité et analyse d’un algorithme
- CM 2 : Structures de données linéaires
  - piles, files, listes chaînées : différentes implantations et applications
  - tables de hachage
- CM 3 : Structures de données arborescentes
  - applications et implantations
  - algorithmes de parcours
  - arbres binaires de recherche
  - arbres équilibrés :
    - AVL, rouge et noire
- CM 4 : Graphes
  - applications et implantations
  - algorithmes de parcours, recherche du plus court chemin, arbre de poids minimum, etc.

Organisation du cours

- CM 1 : Complexité et analyse d'un algorithme
- CM 2 : Structures de données linéaires
  - piles, files, listes chaînées : différentes implantations et applications
  - tables de hachage
- CM 3 : Structures de données arborescentes
  - applications et implantations
  - algorithmes de parcours
  - arbres binaires de recherche
  - arbres équilibrés :
    - AVL, rouge et noire
- CM 4 : Graphes
  - applications et implantations
  - algorithmes de parcours, recherche du plus court chemin, arbre de poids minimum, etc.
- CM 5 : NP-complétude

Contact

- **E-mail :** elena.leroux@univ-ubs.fr

Plan

- Algorithmique
- Pseudocode
- Différentes techniques de preuve
- Preuve d’un algorithme
- Complexité d’un algorithme
- Analyse asymptotique d’un algorithme
- Résumé

Algorithmique (1/2)

- **Conception des méthodes pour la résolution d'un problème donné :**
  - On dispose descriptions :
    - du problème,
    - des données d'entrée et
    - des résultats attendus.

Algorithmique (1/2)

- **Conception des méthodes pour la résolution d'un problème donné :**
  - On dispose descriptions :
    - du problème,
    - des données d'entrée et
    - des résultats attendus.
  - On décrit une ou plusieurs méthodes pour résoudre le problème donné.

Algorithmique (1/2)

- **Conception des méthodes pour la résolution d'un problème donné :**
  - On dispose descriptions :
    - du problème,
    - des données d'entrée et
    - des résultats attendus.
  - On décrit une ou plusieurs méthodes pour résoudre le problème donné.

algorithmes

Algorithmique (1/2)

- **Conception des méthodes pour la résolution d’un problème donné :**
  - On dispose descriptions :
    - du problème,
    - des données d’entrée et
    - des résultats attendus.
  - On décrit une ou plusieurs méthodes pour résoudre le problème donné.
  - On montre que ces méthodes :
    - se terminent et
    - répondent au problème.

algorithmes

Algorithmique (1/2)

- **Conception des méthodes pour la résolution d'un problème donné :**
  - On dispose descriptions :
    - du problème,
    - des données d'entrée et
    - des résultats attendus.
  - On décrit une ou plusieurs méthodes pour résoudre le problème donné.
  - On montre que ces méthodes :
    - se terminent et
    - répondent au problème.

algorithmes

- **Calcul de complexité des méthodes :**
  - **en temps du calcul,**
  - en espace mémoire utilisé.

Algorithmique (1/2)

- **Conception des méthodes pour la résolution d'un problème donné :**
  - On dispose descriptions :
    - du problème,
    - des données d'entrée et
    - des résultats attendus.
  - On décrit une ou plusieurs méthodes pour résoudre le problème donné.
  - On montre que ces méthodes :
    - se terminent et
    - répondent au problème.

algorithmes

- **Calcul de complexité des méthodes :**
  - **en temps du calcul,**
  - en espace mémoire utilisé.
- **Réalisation des méthodes :**
  - Organisation des données.

Algorithmique (1/2)

- **Conception des méthodes pour la résolution d'un problème donné :**
  - On dispose descriptions :
    - du problème,
    - des données d'entrée et
    - des résultats attendus.
  - On décrit une ou plusieurs méthodes pour résoudre le problème donné.
  - On montre que ces méthodes :
    - se terminent et
    - répondent au problème.

algorithmes

- **Calcul de complexité des méthodes :**
  - **en temps du calcul,**
  - en espace mémoire utilisé.
- **Réalisation des méthodes :**
  - Organisation des données. → choix d'une **structure de données**

Algorithmique (1/2)

- **Conception des méthodes pour la résolution d'un problème donné :**
  - On dispose descriptions :
    - du problème,
    - des données d'entrée et
    - des résultats attendus.
  - On décrit une ou plusieurs méthodes pour résoudre le problème donné.
  - On montre que ces méthodes :
    - se terminent et
    - répondent au problème.

algorithmes

- **Calcul de complexité des méthodes :**
  - **en temps du calcul,**
  - en espace mémoire utilisé.
- **Réalisation des méthodes :**
  - Organisation des données.
  - Implémentation des méthodes.

choix d'une **structure de données**

Algorithmique (2/2)

- **Vocabulaire :**
  - **Un algorithme :** une méthode qui résout le problème donné, pas par pas et dans un temps fini.

Algorithmique (2/2)

- **Vocabulaire :**
  - **Un algorithme :** une méthode qui résout le problème donné, pas par pas et dans un temps fini.
  - **Un programme :** est la transcription d'un algorithme dans un langage formel, c'est-à-dire, où toutes les instructions sont spécifiées sans aucune ambiguïté.

Algorithmique (2/2)

- **Vocabulaire :**
  - **Un algorithme :** une méthode qui résout le problème donné, pas par pas et dans un temps fini.
  - **Un programme :** est la transcription d'un algorithme dans un langage formel, c'est-à-dire, où toutes les instructions sont spécifiées sans aucune ambiguïté.
  - **Un entrée :** une valeur qu'un algorithme prend en entrée. Cette valeur doit être choisie à partir d'un ensemble défini.

Algorithmique (2/2)

- **Vocabulaire :**
  - **Un algorithme :** une méthode qui résout le problème donné, pas par pas et dans un temps fini.
  - **Un programme :** est la transcription d'un algorithme dans un langage formel, c'est-à-dire, où toutes les instructions sont spécifiées sans aucune ambiguïté.
  - **Un entrée :** une valeur qu'un algorithme prend en entrée. Cette valeur doit être choisie à partir d'un ensemble défini.
  - **Une sortie :** une solution du problème de départ.

Algorithmique (2/2)

- **Vocabulaire :**
  - **Un algorithme :** une méthode qui résout le problème donné, pas par pas et dans un temps fini.
  - **Un programme :** est la transcription d'un algorithme dans un langage formel, c'est-à-dire, où toutes les instructions sont spécifiées sans aucune ambiguïté.
  - **Un entrée :** une valeur qu'un algorithme prend en entrée. Cette valeur doit être choisie à partir d'un ensemble défini.
  - **Une sortie :** une solution du problème de départ.

- **Propriétés des algorithmes :**
  - **La finitude :** l'algorithme doit produire la sortie souhaitée en un nombre **fini** (mais peut-être très grand) **d'étapes**, quelque soit l'entrée.

Algorithmique (2/2)

- **Vocabulaire :**
  - **Un algorithme :** une méthode qui résout le problème donné, pas par pas et dans un temps fini.
  - **Un programme :** est la transcription d'un algorithme dans un langage formel, c'est-à-dire, où toutes les instructions sont spécifiées sans aucune ambiguïté.
  - **Un entrée :** une valeur qu'un algorithme prend en entrée. Cette valeur doit être choisie à partir d'un ensemble défini.
  - **Une sortie :** une solution du problème de départ.

- **Propriétés des algorithmes :**
  - **La finitude :** l'algorithme doit produire la sortie souhaitée en un nombre **fini** (mais peut-être très grand) **d'étapes**, quelque soit l'entrée.
  - **L'efficacité :** chaque étape de l'algorithme doit pouvoir s'exécuter dans un temps fini.

Algorithmique (2/2)

- **Vocabulaire :**
  - **Un algorithme :** une méthode qui résout le problème donné, pas par pas et dans un temps fini.
  - **Un programme :** est la transcription d'un algorithme dans un langage formel, c'est-à-dire, où toutes les instructions sont spécifiées sans aucune ambiguïté.
  - **Un entrée :** une valeur qu'un algorithme prend en entrée. Cette valeur doit être choisie à partir d'un ensemble défini.
  - **Une sortie :** une solution du problème de départ.

- **Propriétés des algorithmes :**
  - **La finitude :** l'algorithme doit produire la sortie souhaitée en un nombre **fini** (mais peut-être très grand) **d'étapes**, quelque soit l'entrée.
  - **L'efficacité :** chaque étape de l'algorithme doit pouvoir s'exécuter dans un temps fini.
  - **La généralité :** l'algorithme s'applique à tous les problèmes d'une forme désirée.

Plan

- Algorithmique
- Pseudocode
- Différentes techniques de preuve
- Preuve d'un algorithme
- Complexité d'un algorithme
- Analyse asymptotique d'un algorithme
- Résumé

Pseudocode

- Description de haut niveau d'un algorithme.
- Plus structuré que la prose anglaise, française, …
- Moins détaillé qu'un programme.
- Notation préférée pour la description des algorithmes.
- Cache tous les détails de la programmation.

- **Exemple :**

  Trouver l'élément maximal dans le tableau des entiers.

```
Algorithm arrayMax(A,n)
  Input array A of n integers
  Output maximum element of A
Begin
  currentMax ←A[0];
  for i ← 1 to n do
    if A[i]>currentMax then
      currentMax ← A[i];
    end;
  end;
  return currentMax;
End.
```

Détails de pseudocode

- Déclaration d'une méthode :

```
Algorithm name_of_method(arg1,[arg2,…,argn])
  Input …
  Output …
Begin
  …
End.
```

Détails de pseudocode

- Déclaration d'une méthode :

```
Algorithm name_of_method(arg1,[arg2,…,argn])
  Input …
  Output …
Begin
  …
End.
```

- Appel d'une méthode :

```
var.name_of_method(arg1,[arg2,…,argn]);
```

Détails de pseudocode

- Déclaration d'une méthode :

```
Algorithm name_of_method(arg1,[arg2,…,argn])
  Input …
  Output …
Begin
  …
End.
```

- Appel d'une méthode :

```
var.name_of_method(arg1,[arg2,…,argn]);
```

- Structures de contrôle :
  - `if … then … [else …] end;`
  - `while … do … end;`
  - `repeat … until … end;`
  - `for … do … end;`

Détails de pseudocode

- Déclaration d'une méthode :

```
Algorithm name_of_method(arg1,[arg2,…,argn])
  Input …
  Output …
Begin
  …
End.
```

- Appel d'une méthode :

```
var.name_of_method(arg1,[arg2,…,argn]);
```

- Structures de contrôle :
  - `if … then … [else …] end;`
  - `while … do … end;`
  - `repeat … until … end;`
  - `for … do … end;`

- Expressions :
  - ← **affectation** (comme « = » en Java);
  - = **test d'égalité** (comme « == » en Java);
  - n^2 **exposants** et autres formatages mathématiques autorisés.

Détails de pseudocode

- Déclaration d'une méthode :

```
Algorithm name_of_method(arg1,[arg2,…,argn])
   Input …
   Output …
Begin
   …
End.
```

- Appel d'une méthode :

```
var.name_of_method(arg1,[arg2,…,argn]);
```

- Structures de contrôle :
  - `if … then … [else …] end;`
  - `while … do … end;`
  - `repeat … until … end;`
  - `for … do … end;`

- Expressions :
  - ← **affectation** (comme « = » en Java);
  - = **test d'égalité** (comme « == » en Java);
  - n^2 **exposants** et autres formatages mathématiques autorisés.

- Return value :

```
return expression;
```

Opérations élémentaires

- Opérations de base effectuées par l'algorithme :
  - évaluation d'une expression,
  - affectation d'une valeur à une variable,
  - appel à une méthode,
  - accès à une cellule de tableau,
  - etc.

- Les opérations de base sont indépendantes du langage de programmation choisi.

- Le temps d'exécution d'une opération de base est constant.

- Les opérations de base sont facilement identifiables dans un pseudocode.

Plan

- Algorithmique
- Pseudocode
- Différentes techniques de preuve :
  - Preuve par exemple
  - Preuve par contre-exemple
  - Preuve par contrapositive
  - Preuve par contradiction
  - Preuve par induction
- Preuve d'un algorithme
- Complexité d'un algorithme
- Analyse asymptotique d'un algorithme
- Résumé

Différentes techniques de preuves :

preuve par exemple

Proposition :

Il y a des nombres supérieurs à un qui sont égaux au produit de la somme et du produit de leurs chiffres.

$n_1 n_2 \ldots n_p = (n_1 + n_2 + \ldots + n_p) \cdot (n_1 \cdot n_2 \cdot \ldots \cdot n_p)$

Preuve (par exemple) :

$135 = (1+3+5) \cdot (1 \cdot 3 \cdot 5) = 9 \cdot 15$

$144 = (1+4+4) \cdot (1 \cdot 4 \cdot 4) = 9 \cdot 16$

Différentes techniques de preuves :

preuve par exemple

Proposition :

Il y a des nombres supérieurs à un qui sont égaux au produit de la somme et du produit de leurs chiffres.

$$n_1 n_2 \ldots n_p = (n_1 + n_2 + \ldots + n_p) \cdot (n_1 \cdot n_2 \cdot \ldots \cdot n_p)$$

Preuve (par exemple) **:**

$135 = (1+3+5) \cdot (1 \cdot 3 \cdot 5) = 9 \cdot 15$

$144 = (1+4+4) \cdot (1 \cdot 4 \cdot 4) = 9 \cdot 16$ ■

Plan

- Algorithmique
- Pseudocode
- Différentes techniques de preuve :
  - Preuve par exemple
  - Preuve par contre-exemple
  - Preuve par contrapositive
  - Preuve par contradiction
  - Preuve par induction
- Preuve d'un algorithme
- Complexité d'un algorithme
- Analyse asymptotique d'un algorithme
- Résumé

Différentes techniques de preuves :

preuve par contre-exemple

Proposition :

Tous les nombres de la forme $2^i - 1$ sont premiers.

Différentes techniques de preuves :

preuve par contre-exemple

Proposition :

Tous les nombres de la forme 2^i-1 sont premiers.

Preuve (par contre-exemple) **:**

$2^4-1 = 15 = 3 \cdot 5$

$\blacksquare$

Plan

- Algorithmique
- Pseudocode
- Différentes techniques de preuve :
  - Preuve par exemple
  - Preuve par contre-exemple
  - Preuve par contrapositive
  - Preuve par contradiction
  - Preuve par induction
- Preuve d'un algorithme
- Complexité d'un algorithme
- Analyse asymptotique d'un algorithme
- Résumé

Différentes techniques de preuves :

preuve par contrapositive

Principe :

Soient p et q deux prédicats, alors $(p \rightarrow q) \Leftrightarrow (\neg q \rightarrow \neg p)$. Parfois, si la proposition donnée a la forme $(p \rightarrow q)$, il est plus facile de prouver $(\neg q \rightarrow \neg p)$. Cette preuve s'appelle preuve par contrapositive.

Proposition :

Soient a et b entiers. Si $a \bullet b$ est pair, alors a est pair ou b est pair.

Preuve (par contrapositive) **:**

Si a est impair et b est impair, alors $a \bullet b$ est impair.

Supposons $a=2 \bullet i+1$ et $b=2 \bullet j+1$. Alors :

$a \bullet b = 4 \bullet i \bullet j + 2 \bullet i + 2 \bullet j + 1 = 2 \bullet (2 \bullet i \bullet j + i + j) + 1$.

Par conséquent, $a \bullet b$ est impair.

Différentes techniques de preuves :

preuve par contrapositive

Principe :

Soient p et q deux prédicats, alors $(p \rightarrow q) \Leftrightarrow (\neg q \rightarrow \neg p)$. Parfois, si la proposition donnée a la forme $(p \rightarrow q)$, il est plus facile de prouver $(\neg q \rightarrow \neg p)$. Cette preuve s'appelle preuve par contrapositive.

Proposition :

Soient a et b entiers. Si $\underbrace{a \cdot b \text{ est pair}}_{p}$, alors $\underbrace{a \text{ est pair ou } b \text{ est pair}}_{q}$.

$p \rightarrow q$

Preuve (par contrapositive) **:**

Si $\underbrace{a \text{ est impair et } b \text{ est impair}}_{\neg q}$, alors $\underbrace{a \cdot b \text{ est impair}}_{\neg p}$.

$\neg q \rightarrow \neg p$

Supposons $a=2 \cdot i+1$ et $b=2 \cdot j+1$. Alors :

$a \cdot b = 4 \cdot i \cdot j + 2 \cdot i + 2 \cdot j + 1 = 2 \cdot (2 \cdot i \cdot j + i + j) + 1$.

Par conséquent, $a \cdot b$ est impair. ■

Plan

- Algorithmique
- Pseudocode
- Différentes techniques de preuve :
  - Preuve par exemple
  - Preuve par contre-exemple
  - Preuve par contrapositive
  - Preuve par contradiction
  - Preuve par induction
- Preuve d'un algorithme
- Complexité d'un algorithme
- Analyse asymptotique d'un algorithme
- Résumé

Différentes techniques de preuves :

preuve par contradiction

Principe :

Démontrer la vérité d’une proposition en prouvant l’absurdité de la proposition complémentaire (ou contraire). Cette preuve s’appelle preuve par contradiction.

Différentes techniques de preuves :

preuve par contradiction

Principe :

Démontrer la vérité d'une proposition en prouvant l'absurdité de la proposition complémentaire (ou contraire). Cette preuve s'appelle preuve par contradiction.

Proposition :

Soient a et b entiers. Si $a \cdot b$ est impair, alors a est impair et b est impair.

Différentes techniques de preuves :

preuve par contradiction

Principe :

Démontrer la vérité d'une proposition en prouvant l'absurdité de la proposition complémentaire (ou contraire). Cette preuve s'appelle preuve par contradiction.

Proposition :

Soient `a` et `b` entiers. Si `a•b` est impair, alors `a` est impair et `b` est impair.

Preuve (par contradiction) **:**

Soit `a • b` impair.

Supposons, sans perte de généralité, que `a` est pair. Alors `a=2 • i`. Par conséquent, `a • b=2 • i • b`. C'est-à-dire, `a • b` est pair, ce qui constitue une contradiction.

Différentes techniques de preuves :

preuve par contradiction

Principe :

Démontrer la vérité d'une proposition en prouvant l'absurdité de la proposition complémentaire (ou contraire). Cette preuve s'appelle preuve par contradiction.

Proposition :

Soient `a` et `b` entiers. Si `a•b` est impair, alors `a` est impair et `b` est impair.

Preuve (par contradiction) **:**

Soit `a • b` impair.

Supposons, sans perte de généralité, que `a` est pair. Alors `a=2 • i`. Par conséquent, `a • b=2 • i • b`. C'est-à-dire, `a • b` est pair, ce qui constitue une contradiction.

Par conséquent, `a` est impaire et `b` est impaire.

■

Plan

- Algorithmique
- Pseudocode
- Différentes techniques de preuve :
  - Preuve par exemple
  - Preuve par contre-exemple
  - Preuve par contrapositive
  - Preuve par contradiction
  - Preuve par induction
- Preuve d'un algorithme
- Complexité d'un algorithme
- Analyse asymptotique d'un algorithme
- Résumé

Différentes techniques de preuves :

preuve par induction (1/2)

Principe :

Soit `P` une proposition sous un ensemble de `n` éléments.

Si `P` satisfait les conditions suivantes :

- `P` est vraie pour le premier élément,
- si `P` est vraie pour le `n`-ème élément, alors elle est vraie pour le `(n+1)`-ème élément.

alors on peut conclure que la proposition `P` est vraie pour n'importe quel `n`.

Différentes techniques de preuves :

preuve par induction (1/2)

Principe :

Soit P une proposition sous un ensemble de n éléments.

Si P satisfait les conditions suivantes :

- P est vraie pour le premier élément,
- si P est vraie pour le n-ème élément, alors elle est vraie pour le $(n+1)$-ème élément.

alors on peut conclure que la proposition P est vraie pour n'importe quel n.

Proposition :

Prouver que la proposition S_n telle que $1+2+3+\ldots+n = (n(n+1))/2$ est vraie pour tout $n \geq 1$.

Différentes techniques de preuves :

preuve par induction (1/2)

Principe :

Soit P une proposition sous un ensemble de n éléments.

Si P satisfait les conditions suivantes :

- P est vraie pour le premier élément,
- si P est vraie pour le n-ème élément, alors elle est vraie pour le $(n+1)$-ème élément.

alors on peut conclure que la proposition P est vraie pour n'importe quel n.

Proposition :

Prouver que la proposition S_n telle que $1+2+3+\ldots+n = (n(n+1))/2$ est vraie pour tout $n \geq 1$.

Preuve (par induction sur n) **:**

Cas de base :

Soit $n=1$, alors $1=(1(1+1))/2=2/2=1$. Puisque le deux parties de S_n sont égales alors la première condition est vérifiée.

Différentes techniques de preuves :

preuve par induction (2/2)

Hypothèse d'induction :

`1+2+3+…+n = (n(n+1))/2` est vraie jusqu'au rang `n`.

Différentes techniques de preuves :

preuve par induction (2/2)

Hypothèse d'induction :

`1+2+3+…+n = (n(n+1))/2` est vraie jusqu'au rang `n`.

Pas d'induction :

On souhaite prouver que :

`1+2+3+…+(n+1) = ((n+1)((n+1)+1))/2`

Différentes techniques de preuves :

preuve par induction (2/2)

Hypothèse d'induction :

`1+2+3+…+n = (n(n+1))/2` est vraie jusqu'au rang `n`.

Pas d'induction :

On souhaite prouver que :

`1+2+3+…+(n+1) = ((n+1)((n+1)+1))/2`

`1+2+3+…+n+(n+1) = ((n+1)(n+2))/2`

Différentes techniques de preuves :

preuve par induction (2/2)

Hypothèse d'induction :

`1+2+3+…+n = (n(n+1))/2` est vraie jusqu'au rang `n`.

Pas d'induction :

On souhaite prouver que :

`1+2+3+…+(n+1) = ((n+1)((n+1)+1))/2`

`1+2+3+…+n+(n+1) = ((n+1)(n+2))/2`

On utilise l'hypothèse d'induction :

`(n(n+1))/2+(n+1) = ((n+1)(n+2))/2`

Différentes techniques de preuves :

preuve par induction (2/2)

Hypothèse d'induction :

$1+2+3+\ldots+n = (n(n+1))/2$ est vraie jusqu'au rang n.

Pas d'induction :

On souhaite prouver que :

$1+2+3+\ldots+(n+1) = ((n+1)((n+1)+1))/2$

$1+2+3+\ldots+n+(n+1) = ((n+1)(n+2))/2$

On utilise l'hypothèse d'induction :

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

On simplifie :

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

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

$(n^2+3n+2)/2 = ((n+1)(n+2))/2$

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

Différentes techniques de preuves :

preuve par induction (2/2)

Hypothèse d'induction :

`1+2+3+…+n = (n(n+1))/2` est vraie jusqu'au rang `n`.

Pas d'induction :

On souhaite prouver que :

`1+2+3+…+(n+1) = ((n+1)((n+1)+1))/2`

`1+2+3+…+n+(n+1) = ((n+1)(n+2))/2`

On utilise l'hypothèse d'induction :

`(n(n+1))/2+(n+1) = ((n+1)(n+2))/2`

On simplifie :

`(n(n+1))/2 + 2(n+1)/2 = ((n+1)(n+2))/2`

`(n(n+1)+2(n+1))/2 = ((n+1)(n+2))/2`

$(n^2+3n+2)/2 = ((n+1)(n+2))/2$

`((n+1)(n+2))/2 = ((n+1)(n+2))/2`

On a montré que S_{n+1} est aussi vraie.

Différentes techniques de preuves :

preuve par induction (2/2)

Hypothèse d'induction :

$1+2+3+\ldots+n = (n(n+1))/2$ est vraie jusqu'au rang n.

Pas d'induction :

On souhaite prouver que :

$1+2+3+\ldots+(n+1) = ((n+1)((n+1)+1))/2$

$1+2+3+\ldots+n+(n+1) = ((n+1)(n+2))/2$

On utilise l'hypothèse d'induction :

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

On simplifie :

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

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

$(n^2+3n+2)/2 = ((n+1)(n+2))/2$

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

On a montré que S_{n+1} est aussi vraie.

Conclusion :

Puisque les deux conditions ont été vérifiées alors on peut dire que la propriété S_n est vraie pour tout $n \geq 1$.

Plan

- Algorithmique
- Pseudocode
- Différentes techniques de preuve
- Preuve d'un algorithme
- Complexité d'un algorithme
- Analyse asymptotique d'un algorithme
- Résumé

Preuve d'un algorithme

Preuve d'un algorithme

- Un algorithme **A** résout le problème **P** si pour tous énoncés **E** de **P**, qui sont stockés dans le sous-ensemble des variables d'entrée de l'algorithme **A** :
  - la suite d'opérations exécutées est finie (condition de terminaison).
  - le sous-ensemble de variables de sortie contient le résultat associé à l'énoncé **E** (condition de validité).

- Prouver un algorithme **A**, c'est démontrer mathématiquement les conditions de terminaison et de validité de cet algorithme.

Exemple :

somme des éléments d'un tableau

Exemple :

somme des éléments d'un tableau

- Soit **A** un tableau de **n** entiers. Écrire deux algorithmes itératif et récursif qui calculent la somme des éléments de **A**. Prouver ces algorithmes.

- **Algorithme itératif :**

```
Algorithm sum1(A[n],n)
  Input array A of n integers
  Output sum of all elements of A
Begin
  sum ← 0;
  for i ← 1 to n do
    sum ← sum + A[i];
  end;
  return sum;
End.
```

Algorithme récursif :

Exemple :

somme des éléments d'un tableau

- Soit **A** un tableau de **n** entiers. Écrire deux algorithmes itératif et récursif qui calculent la somme des éléments de **A**. Prouver ces algorithmes.

- **Algorithme itératif :**

```
Algorithm sum1(A[n],n)
  Input array A of n integers
  Output sum of all elements of A
Begin
  sum ← 0;
  for i ← 1 to n do
    sum ← sum + A[i];
  end;
  return sum;
End.
```

Algorithme récursif :

```
Algorithm sum2(A[n],n)
  Input array A of n integers
  Output sum of all elements of A
Begin
  if n ≤ 0 then
    return 0;
  else
    return A[n] + sum2(A[n-1],n-1);
  end;
End.
```

Exemple :

preuve de l'algorithme itératif (1/3)

- **Terminaison de l'algorithme :**

  La preuve de la terminaison de l'algorithme itérative est triviale puisque :

  - la boucle est exécutée **n** fois,
  - le corps de la boucle contient les opérations élémentaires qui s'exécutent en un temps fini.

```
Algorithm sum1(A[n],n)
  Input array A of n integers
  Output sum of all elements of A
Begin
  sum ← 0;
  for i ← 1 to n do
    sum ← sum + A[i];
  end;
  return sum;
End.
```

Exemple :

preuve de l'algorithme itératif (1/3)

- **Terminaison de l'algorithme :**

  La preuve de la terminaison de l'algorithme itérative est triviale puisque :
  - la boucle est exécutée **`n`** fois,
  - le corps de la boucle contient les opérations élémentaires qui s'exécutent en un temps fini.

```
Algorithm sum1(A[n],n)
   Input array A of n integers
   Output sum of all elements of A
Begin
   sum ← 0;
   for i ← 1 to n do
      sum ← sum + A[i];
   end;
   return sum;
End.
```

- **Validité de l'algorithme :**

  On doit prouver la proposition suivante :

  « *à la fin de la* ***`i`****-ème itération du boucle, la variable* ***`sum`*** *contient la somme des* ***`i`*** *premiers éléments du tableau* ***`A`*** ».

Exemple :

preuve de l'algorithme itératif (1/3)

```
Algorithm sum1(A[n],n)
   Input array A of n integers
   Output sum of all elements of A
Begin
   sum ← 0;
   for i ← 1 to n do
      sum ← sum + A[i];
   end;
   return sum;
End.
```

- **Terminaison de l'algorithme :**

  La preuve de la terminaison de l'algorithme itérative est triviale puisque :
  - la boucle est exécutée **`n`** fois,
  - le corps de la boucle contient les opérations élémentaires qui s'exécutent en un temps fini.

- **Validité de l'algorithme :**

  On doit prouver la proposition suivante :

  « *à la fin de la **`i`**-ème itération du boucle, la variable **`sum`** contient la somme des **`i`** premiers éléments du tableau **`A`*** ».

  Preuve (par induction sur **`i`**) **:**

  Notons sum_i la valeur de la variable **`sum`** à la fin de la **`i`**-ème itération et $sum_0=0$ sa valeur initiale.

Exemple :

preuve de l'algorithme itératif (1/3)

```
Algorithm sum1(A[n],n)
  Input array A of n integers
  Output sum of all elements of A
Begin
  sum ← 0;
  for i ← 1 to n do
    sum ← sum + A[i];
  end;
  return sum;
End.
```

- **Terminaison de l'algorithme :**

  La preuve de la terminaison de l'algorithme itérative est triviale puisque :
  - la boucle est exécutée `n` fois,
  - le corps de la boucle contient les opérations élémentaires qui s'exécutent en un temps fini.

- **Validité de l'algorithme :**

  On doit prouver la proposition suivante :

  « à la fin de la `i`-ème itération du boucle, la variable `sum` contient la somme des `i` premiers éléments du tableau `A` ».

  Preuve (par induction sur `i`) **:**

  Notons sum_i la valeur de la variable `sum` à la fin de la `i`-ème itération et $sum_0=0$ sa valeur initiale.
  - Cas de base :

    Soit `i=1`, alors la proposition est trivialement vraie car à l'issue de la première itération `sum` (initialisée à `0`) contient la valeur de `A[1]` :

    $sum_1=sum_0+A[1]=A[1]$.

Exemple :

preuve de l'algorithme itératif (2/3)

- **Hypothèse de l'induction :**
  On suppose que la proposition

  est vraie pour l'itération `i`.

```
Algorithm sum1(A[n],n)
  Input array A of n integers
  Output sum of all elements of A
Begin
  sum ← 0;
  for i ← 1 to n do
    sum ← sum + A[i];
  end;
  return sum;
End.
```

Exemple :

preuve de l'algorithme itératif (2/3)

- **Hypothèse de l'induction :**

  On suppose que la proposition

  est vraie pour l'itération `i`.

- **Pas de l'induction :**

  On souhaite prouver que :

```
Algorithm sum1(A[n],n)
  Input array A of n integers
  Output sum of all elements of A
Begin
  sum ← 0;
  for i ← 1 to n do
    sum ← sum + A[i];
  end;
  return sum;
End.
```

Exemple :

preuve de l'algorithme itératif (2/3)

- **Hypothèse de l'induction :**

  On suppose que la proposition

  est vraie pour l'itération **i**.

- **Pas de l'induction :**

  On souhaite prouver que :

  On sait que :

```
Algorithm sum1(A[n],n)
  Input array A of n integers
  Output sum of all elements of A
Begin
  sum ← 0;
  for i ← 1 to n do
    sum ← sum + A[i];
  end;
  return sum;
End.
```

Exemple :

preuve de l'algorithme itératif (2/3)

- **Hypothèse de l'induction :**

  On suppose que la proposition

  est vraie pour l'itération **`i`**.

- **Pas de l'induction :**

  On souhaite prouver que :

  On sait que :

  Alors par hypothèse d'induction on a :

```
Algorithm sum1(A[n],n)
  Input array A of n integers
  Output sum of all elements of A
Begin
  sum ← 0;
  for i ← 1 to n do
    sum ← sum + A[i];
  end;
  return sum;
End.
```

Exemple :

preuve de l'algorithme itératif (2/3)

- **Hypothèse de l'induction :**

  On suppose que la proposition

  est vraie pour l'itération `i`.

- **Pas de l'induction :**

  On souhaite prouver que :

  On sait que :

  Alors par hypothèse d'induction on a :

  On obtient donc que :

```
Algorithm sum1(A[n],n)
  Input array A of n integers
  Output sum of all elements of A
Begin
  sum ← 0;
  for i ← 1 to n do
    sum ← sum + A[i];
  end;
  return sum;
End.
```

Exemple :

preuve de l'algorithme itératif (2/3)

```
Algorithm sum1(A[n],n)
  Input array A of n integers
  Output sum of all elements of A
Begin
  sum ← 0;
  for i ← 1 to n do
    sum ← sum + A[i];
  end;
  return sum;
End.
```

- **Hypothèse de l'induction :**

  On suppose que la proposition

  est vraie pour l'itération `i`.

- **Pas de l'induction :**

  On souhaite prouver que :

  On sait que :

  Alors par hypothèse d'induction on a :

  On obtient donc que :

- **Conclusion :**

  On a prouvé que vraie initialement, la proposition est vraie à chaque itération de boucle. Cette proposition est donc un invariant de l'algorithme. On parle aussi d'invariant de boucle.

Exemple :

preuve de l'algorithme itératif (3/3)

- **Conclusion :**

On a montré :

- que l'algorithme itératif se termine et
- qu'à la fin de la **n**-ème itération, il aura bien calculé la somme de **n** éléments du tableau **A**.

On a donc prouvé l'algorithme itératif.

```
Algorithm sum1(A[n],n)
   Input array A of n integers
   Output sum of all elements of A
Begin
   sum ← 0;
   for i ← 1 to n do
      sum ← sum + A[i];
   end;
   return sum;
End.
```

Exemple :

preuve de l'algorithme récursif (1/2)

- **Terminaison de l'algorithme :**

  Preuve (par induction sur **n**) :

  - Cas de base :

    Soit **n=0**, alors l'algorithme **sum2** se termine immédiatement de façon évidente.

```
Algorithm sum2(A[n],n)
  Input array A of n integers
  Output sum of all elements of A
Begin
  if n ≤ 0 then
    return 0;
  else
    return A[n] + sum2(A[n-1],n-1);
  end;
End.
```

Exemple :

preuve de l'algorithme récursif (1/2)

- **Terminaison de l'algorithme :**

Preuve (par induction sur `n`) :

- **Cas de base :**

  Soit `n=0`, alors l'algorithme `sum2` se termine immédiatement de façon évidente.

- **Hypothèse d'induction :**

  On suppose que l'algorithme `sum2` avec un paramètre `n` se termine.

```
Algorithm sum2(A[n],n)
   Input array A of n integers
   Output sum of all elements of A
Begin
   if n ≤ 0 then
      return 0;
   else
      return A[n] + sum2(A[n-1],n-1);
   end;
End.
```

Exemple :

preuve de l'algorithme récursif (1/2)

- **Terminaison de l'algorithme :**

Preuve (par induction sur `n`) :

- Cas de base :

  Soit `n=0`, alors l'algorithme `sum2` se termine immédiatement de façon évidente.

- Hypothèse d'induction :

  On suppose que l'algorithme `sum2` avec un paramètre `n` se termine.

```
Algorithm sum2(A[n],n)
   Input array A of n integers
   Output sum of all elements of A
Begin
   if n ≤ 0 then
      return 0;
   else
      return A[n] + sum2(A[n-1],n-1);
   end;
End.
```

- Pas de l'induction :

  Si on appel l'algorithme `sum2` avec le paramètre `n+1`, alors :

  - on appel l'algorithme `sum2` avec le paramètre `n` (qui par hypothèse de l'induction se termine),
  - on ajoute au résultat de cet appel la valeur de `A[n+1]` et
  - on termine.

Exemple :

preuve de l'algorithme récursif (1/2)

- **Terminaison de l'algorithme :**

Preuve (par induction sur `n`) :

- **Cas de base :**

  Soit `n=0`, alors l'algorithme `sum2` se termine immédiatement de façon évidente.

- **Hypothèse d'induction :**

  On suppose que l'algorithme `sum2` avec un paramètre `n` se termine.

```
Algorithm sum2(A[n],n)
  Input array A of n integers
  Output sum of all elements of A
Begin
  if n ≤ 0 then
    return 0;
  else
    return A[n] + sum2(A[n-1],n-1);
  end;
End.
```

- **Pas de l'induction :**

  Si on appel l'algorithme `sum2` avec le paramètre `n+1`, alors :

  - on appel l'algorithme `sum2` avec le paramètre `n` (qui par hypothèse de l'induction se termine),
  - on ajoute au résultat de cet appel la valeur de `A[n+1]` et
  - on termine.

- **Conclusion :**

  Donc, quelle que soit la valeur du paramètre `n`, l'appel de la fonction `sum2` se termine.

■

Exemple :

preuve de l'algorithme récursif (2/2)

- **Validité de l'algorithme :**

  On note sum_n la valeur retournée par l'appel de l'algorithme **sum2** avec le paramètre **n**.

  Cet algorithme récursif reproduit la relation mathématique de récurrence suivante :

```
Algorithm sum2(A[n],n)
  Input array A of n integers
  Output sum of all elements of A
Begin
  if n ≤ 0 then
    return 0;
  else
    return A[n] + sum2(A[n-1],n-1);
  end;
End.
```

On montre facilement par induction (de la même façon que pour l'algorithme itératif) que la solution de cette expression récursive est :

Exemple :

preuve de l'algorithme récursif (2/2)

- **Validité de l'algorithme :**

  On note **sum**$_n$ la valeur retournée par l'appel de l'algorithme **sum2** avec le paramètre **n**.

  Cet algorithme récursif reproduit la relation mathématique de récurrence suivante :

```
Algorithm sum2(A[n],n)
  Input array A of n integers
  Output sum of all elements of A
Begin
  if n ≤ 0 then
    return 0;
  else
    return A[n] + sum2(A[n-1],n-1);
  end;
End.
```

On montre facilement par induction (de la même façon que pour l'algorithme itératif) que la solution de cette expression récursive est :

- **Conclusion :**

  On a donc montré :

  - que l'algorithme récursif se termine et
  - qu'il calcule bien la somme des éléments du tableau **A**.

  On a donc prouvé l'algorithme récursif.

Plan

- Algorithmique
- Pseudocode
- Différentes techniques de preuve
- Preuve d'un algorithme
- Complexité d'un algorithme
  - Qualité d'un algorithme
  - Temps d'exécution et complexité d'un algorithme
  - Différentes méthodes pour mesurer la complexité
  - Complexité asymptotique
  - Intuitions de notations asymptotiques
  - Propriétés importantes de notations asymptotiques
  - Classes principales de complexité
- Analyse asymptotique d'un algorithme
- Résumé

Qualité d'un algorithme

- Il y a différentes mesures de la qualité d'un algorithme, par exemple :
  - temps pour le programmer
  - temps pour le corriger/généraliser
  - temps pour l'exécuter
  - espace requis pour l'exécuter
- Difficile (souvent impossible) d'obtenir un algorithme optimal selon toutes ces caractéristiques.
- Compromis en faveur du temps d'exécution (plus objectif).
- Espace mémoire est parfois (mais rarement) un facteur important.

Plan

- Algorithmique
- Pseudocode
- Différentes techniques de preuve
- Preuve d'un algorithme
- Complexité d'un algorithme
  - Qualité d'un algorithme
  - Temps d'exécution et complexité d'un algorithme
  - Différentes méthodes pour mesurer la complexité
  - Complexité asymptotique
  - Intuitions de notations asymptotiques
  - Propriétés importantes de notations asymptotiques
  - Classes principales de complexité
- Analyse asymptotique d'un algorithme
- Résumé

Calculer le temps de l'exécution

- Il y a plusieurs facteurs qui influencent le temps de l'exécution d'un algorithme :
  - quantité et valeurs des données,
  - type de la machine,
  - langage de programmation,
  - qualité du code,
  - qualité du compilateur / interpréteur,
  - etc.

- La complexité en temps d'un algorithme est une mesure de l'ordre de grandeur du temps de l'exécution de l'algorithme en fonction (uniquement) de la taille des données d'un problème.

Mesurer la complexité

- **Il y a différentes mesures de la complexité d'un algorithme :**
  - **complexité dans le meilleur des cas :** analyse sur les meilleurs données d'un algorithme.
    - peu pertinente et donc pas intéressante.

Mesurer la complexité

- **Il y a différentes mesures de la complexité d'un algorithme :**
  - **complexité dans le meilleur des cas :** analyse sur les meilleurs données d'un algorithme.
    - peu pertinente et donc pas intéressante.
  - **complexité en moyenne :** analyse statistique en fonction d'une distribution particulière dans l'espace des problèmes.
    - très intéressante mais souvent difficile à obtenir.

Mesurer la complexité

- **Il y a différentes mesures de la complexité d'un algorithme :**
  - **complexité dans le meilleur des cas :** analyse sur les meilleurs données d'un algorithme.
    - peu pertinente et donc pas intéressante.
  - **complexité en moyenne :** analyse statistique en fonction d'une distribution particulière dans l'espace des problèmes.
    - très intéressante mais souvent difficile à obtenir.
  - **complexité dans le pire des cas :** analyse sur les plus mauvaises données d'un algorithme.
    - pertinente,
    - plus facile à analyser,
    - cruciale pour plupart d'applications : applications de finances, jeux, robotique, …

Mesurer la complexité

- **Il y a différentes mesures de la complexité d'un algorithme :**
  - **complexité dans le meilleur des cas :** analyse sur les meilleurs données d'un algorithme.
    - peu pertinente et donc pas intéressante.
  - **complexité en moyenne :** analyse statistique en fonction d'une distribution particulière dans l'espace des problèmes.
    - très intéressante mais souvent difficile à obtenir.
  - **complexité dans le pire des cas :** analyse sur les plus mauvaises données d'un algorithme.
    - pertinente,
    - plus facile à analyser,
    - cruciale pour plupart d'applications : applications de finances, jeux, robotique, …

- **On privilégie la complexité dans le pire des cas parce que l'analyse est rigoureuse et simple.**

Mesurer la complexité :

notions formelles

- Soit `T(algo,d)` le temps d'exécution de l'algorithme `algo` appliqué aux données `d` de taille `n`.

Mesurer la complexité :

notions formelles

- Soit `T(algo,d)` le temps d'exécution de l'algorithme `algo` appliqué aux données `d` de taille `n`.

  Alors :

  - Complexité dans le pire des cas :

Mesurer la complexité :
notions formelles

- Soit `T(algo,d)` le temps d'exécution de l'algorithme `algo` appliqué aux données `d` de taille `n`.

  Alors :
  - Complexité dans le pire des cas :
  - Complexité dans le meilleur des cas :

Mesurer la complexité :

notions formelles

- Soit `T(algo,d)` le temps d'exécution de l'algorithme `algo` appliqué aux données `d` de taille `n`.

  Alors :
  - Complexité dans le pire des cas :
  - Complexité dans le meilleur des cas :
  - Complexité en moyenne :

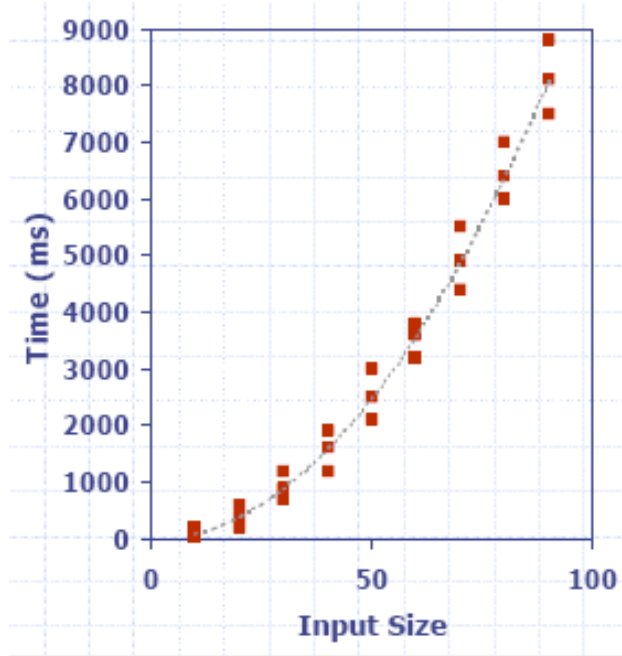
    où `p(d)` est la probabilité d'avoir l'entrée `d`.

Plan

- Algorithmique
- Pseudocode
- Différentes techniques de preuve
- Preuve d'un algorithme
- Complexité d'un algorithme
  - Qualité d'un algorithme
  - Temps d'exécution et complexité d'un algorithme
  - Différentes méthodes pour mesurer la complexité :
    - Méthode 1 : études expérimentales
    - Méthode 2 : analyse théorique
  - Complexité asymptotique
  - Intuitions de notations asymptotiques
  - Propriétés importantes de notations asymptotiques
  - Classes principales de complexité
- Analyse asymptotique d'un algorithme
- Résumé

Méthode 1 : études expérimentales

- Implémenter l'algorithme en Java (ou autre langage de programmation).
- Faire fonctionner le programme avec des entrées de taille et de composition différentes.
- Utiliser une méthode comme `System.currentTimeMillis()` pour obtenir une mesure réelle du temps d'exécution.
- Dessiner le graphique des résultats.



Limitation de la méthode 1

- On doit implémenter l'algorithme.
  - On veux connaître la complexité en temps d'un algorithme avant de l'implémenter, question de sauver du temps et de l'argent.
  - Les résultats trouvés ne sont pas représentatifs.
- Pour comparer différents algorithmes qui résolvent le même problème on doit utiliser le même environnement (hardware, software).

Plan

Méthode 2 : analyse théorique

- Dans cette méthode on écrit les algorithmes en pseudocode (description de haut niveau), on ne les implémente pas.
- Caractérise le temps d'exécution comme une fonction de la taille de l'entrée `n`.
- Prend en compte toutes les entrées possibles.
- Indépendant de l'environnement utilisé (hardware, software).

Compter les opérations élémentaires

- En expectant le pseudocode d'un algorithme on peut déterminer le nombre maximum, minumum d'opérations élémentaires exécuter par un algorithme, comme une fonction de la taille de l'entrée.

```
Algorithm arrayMax(A,n)
  Input array A of n integers
  Output maximum element of A
Begin
  currentMax ← A[0];
  for i ← 1 to n-1 do
    if A[i]>currentMax then
      currentMax ← A[i];
    end;
    {increment counter i}
  end;
  return currentMax;
End.
```

Compter les opérations élémentaires

- En expectant le pseudocode d'un algorithme on peut déterminer le nombre maximum, minumum d'opérations élémentaires exécuter par un algorithme, comme une fonction de la taille de l'entrée.

nombre d'opérations

```
Algorithm arrayMax(A,n)
  Input array A of n integers
  Output maximum element of A
Begin
  currentMax ← A[0];
  for i ← 1 to n-1 do
    if A[i]>currentMax then
      currentMax ← A[i];
    end;
    {increment counter i}
  end;
  return currentMax;
End.
```

Compter les opérations élémentaires

- En expectant le pseudocode d'un algorithme on peut déterminer le nombre maximum, minumum d'opérations élémentaires exécuter par un algorithme, comme une fonction de la taille de l'entrée.

```
Algorithm arrayMax(A,n)                         nombre d'opérations
  Input array A of n integers
  Output maximum element of A
Begin
  currentMax ← A[0];                         2
  for i ← 1 to n-1 do
    if A[i]>currentMax then
      currentMax ← A[i];
    end;
    {increment counter i}
  end;
  return currentMax;
End.
```

Compter les opérations élémentaires

- En expectant le pseudocode d'un algorithme on peut déterminer le nombre maximum, minumum d'opérations élémentaires exécuter par un algorithme, comme une fonction de la taille de l'entrée.

```
Algorithm arrayMax(A,n)                          nombre d'opérations
  Input array A of n integers
  Output maximum element of A
Begin
  currentMax ← A[0];                         2
  for i ← 1 to n-1 do                        1+2(n-1)
    if A[i]>currentMax then
      currentMax ← A[i];
    end;
    {increment counter i}
  end;
  return currentMax;
End.
```

Compter les opérations élémentaires

- En expectant le pseudocode d'un algorithme on peut déterminer le nombre maximum, minumum d'opérations élémentaires exécuter par un algorithme, comme une fonction de la taille de l'entrée.

```
Algorithm arrayMax(A,n)
  Input array A of n integers
  Output maximum element of A
Begin
  currentMax ← A[0];
  for i ← 1 to n-1 do
    if A[i]>currentMax then
      currentMax ← A[i];
    end;
    {increment counter i}
  end;
  return currentMax;
End.
```

nombre d'opérations

2
1+2(n-1)
2(n-1)

Compter les opérations élémentaires

- En expectant le pseudocode d'un algorithme on peut déterminer le nombre maximum, minumum d'opérations élémentaires exécuter par un algorithme, comme une fonction de la taille de l'entrée.

```
Algorithm arrayMax(A,n)
  Input array A of n integers
  Output maximum element of A
Begin
  currentMax ← A[0];
  for i ← 1 to n-1 do
    if A[i]>currentMax then
      currentMax ← A[i];
    end;
    {increment counter i}
  end;
  return currentMax;
End.
```

nombre d'opérations

2
1+2(n-1)
2(n-1)
2(n-1)

Compter les opérations élémentaires

- En expectant le pseudocode d'un algorithme on peut déterminer le nombre maximum, minumum d'opérations élémentaires exécuter par un algorithme, comme une fonction de la taille de l'entrée.

Pseudocode	nombre d'opérations
Algorithm arrayMax(*A*, *n*)	
Input array *A* of *n* integers	
Output maximum element of *A*	
Begin	
currentMax ← *A*[0];	2
for *i* ← 1 **to** *n*−1 **do**	1+2(n-1)
if *A*[*i*]>*currentMax* **then**	2(n-1)
currentMax ← A[*i*];	2(n-1)
end;	
{increment counter i}	2(n-1)
end;	
return *currentMax*;	
End.	

Compter les opérations élémentaires

- En expectant le pseudocode d'un algorithme on peut déterminer le nombre maximum, minumum d'opérations élémentaires exécuter par un algorithme, comme une fonction de la taille de l'entrée.

	nombre d'opérations
Algorithm arrayMax(*A*, *n*)	
Input array *A* of *n* integers	
Output maximum element of *A*	
Begin	
currentMax ← *A*[0];	2
for *i* ← 1 **to** *n*−1 **do**	1+2(n-1)
if *A*[*i*]>*currentMax* **then**	2(n-1)
currentMax ← A[*i*];	2(n-1)
end;	
{increment counter i}	2(n-1)
end;	1
return *currentMax*;	
End.	

Compter les opérations élémentaires

- En expectant le pseudocode d'un algorithme on peut déterminer le nombre maximum, minumum d'opérations élémentaires exécuter par un algorithme, comme une fonction de la taille de l'entrée.

Algorithme	nombre d'opérations
Algorithm arrayMax(*A*, *n*)	
Input array *A* of *n* integers	
Output maximum element of *A*	
Begin	
currentMax ← *A*[0];	2
for *i* ← 1 **to** *n*−1 **do**	1+2(n-1)
if *A*[*i*]>*currentMax* **then**	2(n-1)
currentMax ← A[*i*];	2(n-1)
end;	
{increment counter i}	2(n-1)
end;	
return *currentMax*;	1
End.	

Total au pire :

$T_{max} = 8n-4$

Compter les opérations élémentaires

- En expectant le pseudocode d'un algorithme on peut déterminer le nombre maximum, minumum d'opérations élémentaires exécuter par un algorithme, comme une fonction de la taille de l'entrée.

```
Algorithm arrayMax(A,n)
  Input array A of n integers
  Output maximum element of A
Begin
  currentMax ← A[0];
  for i ← 1 to n-1 do
    if A[i]>currentMax then
      currentMax ← A[i];
    end;
    {increment counter i}
  end;
  return currentMax;
End.
```

nombre d'opérations

2
1+2(n-1)
2(n-1)

2(n-1)
1

Total au pire :
T_{max} = 8n-4

Total au mieux :
T_{min} = 6n-2

Compter les opérations élémentaires

- En expectant le pseudocode d'un algorithme on peut déterminer le nombre maximum, minumum d'opérations élémentaires exécuter par un algorithme, comme une fonction de la taille de l'entrée.

```
Algorithm arrayMax(A,n)
  Input array A of n integers
  Output maximum element of A
Begin
  currentMax ← A[0];
  for i ← 1 to n-1 do
    if A[i]>currentMax then
      currentMax ← A[i];
    end;
    {increment counter i}
  end;
  return currentMax;
End.
```

nombre d'opérations

Total au pire :
$T_{max} = 8n-4$

Total au mieux :
$T_{min} = 6n-2$

Compter les opérations élémentaires

- En expectant le pseudocode d'un algorithme on peut déterminer le nombre maximum, minumum d'opérations élémentaires exécuter par un algorithme, comme une fonction de la taille de l'entrée.

```
Algorithm arrayMax(A,n)
  Input array A of n integers
  Output maximum element of A
Begin
  currentMax ← A[0];
  for i ← 1 to n-1 do
    if A[i]>currentMax then
      currentMax ← A[i];
    end;
    {increment counter i}
  end;
  return currentMax;
End.
```

nombre d'opérations

Total au pire :
$T_{max} = 8n-4$

Total au mieux :
$T_{min} = 6n-2$

Compter les opérations élémentaires

- En expectant le pseudocode d'un algorithme on peut déterminer le nombre maximum, minumum d'opérations élémentaires exécuter par un algorithme, comme une fonction de la taille de l'entrée.

```
Algorithm arrayMax(A,n)
  Input array A of n integers
  Output maximum element of A
Begin
  currentMax ← A[0];
  for i ← 1 to n-1 do
    if A[i]>currentMax then
      currentMax ← A[i];
    end;
    {increment counter i}
  end;
  return currentMax;
End.
```

nombre d'opérations

$T_{max} = 8n-4$

Total au mieux :
$T_{min} = 6n-2$

Compter les opérations élémentaires

- En expectant le pseudocode d'un algorithme on peut déterminer le nombre maximum, minumum d'opérations élémentaires exécuter par un algorithme, comme une fonction de la taille de l'entrée.

```
Algorithm arrayMax(A,n)
  Input array A of n integers
  Output maximum element of A
Begin
  currentMax ← A[0];
  for i ← 1 to n-1 do
    if A[i]>currentMax then
      currentMax ← A[i];
    end;
    {increment counter i}
  end;
  return currentMax;
End.
```

nombre d'opérations

Total au mieux :
$T_{min} = 6n-2$

Compter les opérations élémentaires

- En expectant le pseudocode d'un algorithme on peut déterminer le nombre maximum, minumum d'opérations élémentaires exécuter par un algorithme, comme une fonction de la taille de l'entrée.

```
Algorithm arrayMax(A,n)
  Input array A of n integers
  Output maximum element of A
Begin
  currentMax ← A[0];
  for i ← 1 to n-1 do
    if A[i]>currentMax then
      currentMax ← A[i];
    end;
    {increment counter i}
  end;
  return currentMax;
End.
```

nombre d'opérations

Total au mieux :
$T_{min} = 6n-2$

Compter les opérations élémentaires

- En expectant le pseudocode d'un algorithme on peut déterminer le nombre maximum, minumum d'opérations élémentaires exécuter par un algorithme, comme une fonction de la taille de l'entrée.

```
Algorithm arrayMax(A,n)
  Input array A of n integers
  Output maximum element of A
Begin
  currentMax ← A[0];
  for i ← 1 to n-1 do
    if A[i]>currentMax then
      currentMax ← A[i];
    end;
    {increment counter i}
  end;
  return currentMax;
End.
```

nombre d'opérations

$T_{min} = 6n-2$

Compter les opérations élémentaires

- En expectant le pseudocode d'un algorithme on peut déterminer le nombre maximum, minumum d'opérations élémentaires exécuter par un algorithme, comme une fonction de la taille de l'entrée.

```
Algorithm arrayMax(A,n)
  Input array A of n integers
  Output maximum element of A
Begin
  currentMax ← A[0];
  for i ← 1 to n-1 do
    if A[i]>currentMax then
      currentMax ← A[i];
    end;
    {increment counter i}
  end;
  return currentMax;
End.
```

nombre d'opérations

Plan

- Algorithmique
- Pseudocode
- Différentes techniques de preuve
- Preuve d'un algorithme
- Complexité d'un algorithme
  - Qualité d'un algorithme
  - Temps d'exécution et complexité d'un algorithme
  - Différentes méthodes pour mesurer la complexité :
  - Complexité asymptotique
  - Intuitions de notations asymptotiques
  - Propriétés importantes de notations asymptotiques
  - Classes principales de complexité
- Analyse asymptotique d'un algorithme
- Résumé

Complexité asymptotique

- Pour ne retenir que les caractéristiques essentielles d'une complexité, et rendre ainsi son calcul simple mais indicatif, il est légitime d'ignorer toute constante pouvant apparaître lors du décompte du nombre de fois qu'une instruction est exécutée.

Complexité asymptotique

- Pour ne retenir que les caractéristiques essentielles d'une complexité, et rendre ainsi son calcul simple mais indicatif, il est légitime d'ignorer toute constante pouvant apparaître lors du décompte du nombre de fois qu'une instruction est exécutée.

- **Exemple :**

  Si

  T_{max}`(arrayMax,n) = 8n-4`,

  alors on dira que la complexité dans le pire de cas de cet algorithme est égale à `n`.

Complexité asymptotique

- Pour ne retenir que les caractéristiques essentielles d'une complexité, et rendre ainsi son calcul simple mais indicatif, il est légitime d'ignorer toute constante pouvant apparaître lors du décompte du nombre de fois qu'une instruction est exécutée.

- **Exemple :**

  Si

  T_{max}`(arrayMax,n) = 8n-4`,

  alors on dira que la complexité dans le pire de cas de cet algorithme est égale à `n`.

- Le résultat obtenu à l'aide de ces simplifications représente ce qu'on appelle la complexité asymptotique de l'algorithme considéré.

Complexité asymptotique

- Pour ne retenir que les caractéristiques essentielles d'une complexité, et rendre ainsi son calcul simple mais indicatif, il est légitime d'ignorer toute constante pouvant apparaître lors du décompte du nombre de fois qu'une instruction est exécutée.

- **Exemple :**

  Si

  T_{max}`(arrayMax,n) = 8n-4`,

  alors on dira que la complexité dans le pire de cas de cet algorithme est égale à `n`.

- Le résultat obtenu à l'aide de ces simplifications représente ce qu'on appelle la complexité asymptotique de l'algorithme considéré.

- La complexité asymptotique d'un algorithme décrit le comportement de celui-ci quand la taille `n` des données du problème traité devient de plus en plus grande, plutôt qu'une mesure exacte du temps d'exécution.

Plan

- Algorithmique
- Pseudocode
- Différentes techniques de preuve
- Preuve d'un algorithme
- Complexité d'un algorithme
  - Qualité d'un algorithme
  - Temps d'exécution et complexité d'un algorithme
  - Différentes méthodes pour mesurer la complexité :
  - Complexité asymptotique :
    - Notation du O
    - Notation du Ω
    - Notation du Θ
  - Intuitions de notations asymptotiques
  - Propriétés importantes de notations asymptotiques
  - Classes principales de complexité
- Analyse asymptotique d'un algorithme
- Résumé

Notations du grand-O et de la borne supérieure asymptotique

- **Définitions :**
  - Pour une fonction $g(n)$ donnée, on note $O(g(n))$ l'ensemble de fonctions $f(n)$ telles qu'il existe des constantes $c > 0$ et $n_0 \geq 1$ telles que

    $0 \leq f(n) \leq c \cdot g(n)$ pour tout $n \geq n_0$.

  - Si une fonction $f(n) \in O(g(n))$, on dit que $g(n)$ est une **borne supérieure asymptotique** pour $f(n)$.

    On note : $f(n) = O(g(n))$.

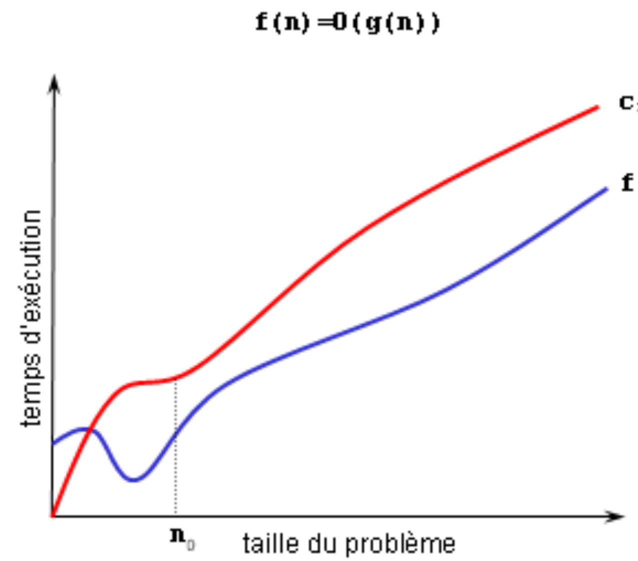
Notations du grand-O et de la borne supérieure asymptotique

- **Définitions :**
  - Pour une fonction $g(n)$ donnée, on note $O(g(n))$ l'ensemble de fonctions $f(n)$ telles qu'il existe des constantes $c > 0$ et $n_0 \geq 1$ telles que

    $0 \leq f(n) \leq c \bullet g(n)$ pour tout $n \geq n_0$.

  - Si une fonction $f(n) \in O(g(n))$, on dit que $g(n)$ est une **borne supérieure asymptotique** pour $f(n)$.

    On note : $f(n) = O(g(n))$.



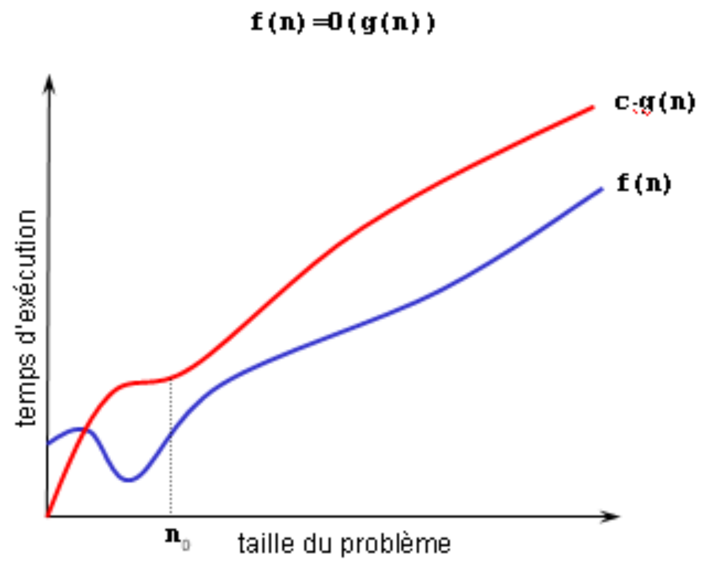
Notations du grand-O et de la borne supérieure asymptotique

- **Définitions :**
  - Pour une fonction $g(n)$ donnée, on note $O(g(n))$ l'ensemble de fonctions $f(n)$ telles qu'il existe des constantes $c > 0$ et $n_0 \geq 1$ telles que

    $0 \leq f(n) \leq c \bullet g(n)$ pour tout $n \geq n_0$.

  - Si une fonction $f(n) \in O(g(n))$, on dit que $g(n)$ est une **borne supérieure asymptotique** pour $f(n)$.

    On note : $f(n)=O(g(n))$.



- **Signification :**
  - Pour toutes les grandes entrées (c'est-à-dire $n \geq n_0$), on est assuré que l'algorithme ne prend pas plus de $c \bullet g(n)$ étapes.

Notations du grand-O et de la borne supérieure asymptotique

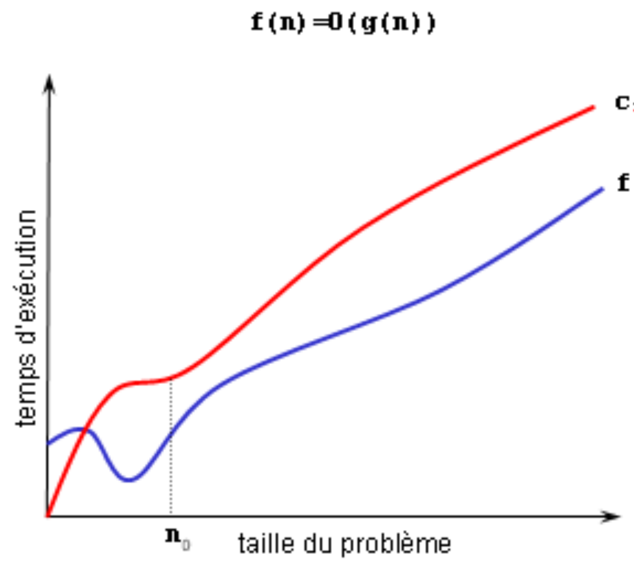
- **Définitions :**
  - Pour une fonction $g(n)$ donnée, on note $O(g(n))$ l'ensemble de fonctions $f(n)$ telles qu'il existe des constantes $c > 0$ et $n_0 \geq 1$ telles que

    $0 \leq f(n) \leq c \cdot g(n)$ pour tout $n \geq n_0$.

  - Si une fonction $f(n) \in O(g(n))$, on dit que $g(n)$ est une **borne supérieure asymptotique** pour $f(n)$.

    On note : $f(n) = O(g(n))$.

- **Signification :**
  - Pour toutes les grandes entrées (c'est-à-dire $n \geq n_0$), on est assuré que l'algorithme ne prend pas plus de $c \cdot g(n)$ étapes.
  - La borne supérieure est utilisée pour borner le temps d'exécution d'un algorithme dans le pire des cas.



Exemples

- $2 \cdot n+10 \in O(n)$
  - On doit trouver des constantes $c>0$ et $n_0 \geq 1$ telles que $2 \cdot n+10 \leq c \cdot n$ pour $n \geq n_0$.
    - $2 \cdot n+10 \leq c \cdot n$
    - $(c-2) \cdot n \geq 10$
    - $n \geq 10/(c-2)$
  - On peut par exemple choisir $c=3$ et $n_0=10$.
- $n^2 \notin O(n)$
  - On doit trouver des constantes $c>0$ et $n_0 \geq 1$ telles que $n^2 \leq c \cdot n$ pour $n \geq n_0$.
    - $n^2 \leq c \cdot n$
    - $n \leq c$
  - Cette inégalité ne peut pas être satisfaite car c doit être constante.
- $3 \cdot n^3+20 \cdot n^2+5 \in O(n^3)$
  - On doit trouver des constantes $c>0$ et $n_0 \geq 1$ telles que $3 \cdot n^3+20 \cdot n^2+5 \leq c \cdot n^3$ pour $n \geq n_0$.
  - On peut par exemple choisir $c=2$ et $n_0=21$.
- $3 \cdot \log(n)+5 \in O(\log(n))$
  - On doit trouver des constantes $c>0$ et $n_0 \geq 1$ telles que $3 \cdot \log(n)+5 \leq c \cdot \log(n)$ pour $n \geq n_0$.
  - On peut par exemple choisir $c=8$ et $n_0=2$.

Règles d'utilisation de grand-O

- Si `f(n)` est un polynôme de degré `d`, alors `f(n)` appartient à $O(n^d)$, c'est-à-dire que :
  - on « oublie » les termes de plus petit ordre,
  - on remplace le facteur du terme de plus haut rang par un.

Règles d'utilisation de grand-O

- Si $f(n)$ est un polynôme de degré d, alors $f(n)$ appartient à $O(n^d)$, c'est-à-dire que :
  - on « oublie » les termes de plus petit ordre,
  - on remplace le facteur du terme de plus haut rang par un.

Exemple :

- Si $f(n)=3n^3+2n^2+1$ alors $f(n) \in O(n^3)$.

Règles d'utilisation de grand-O

- Si $f(n)$ est un polynôme de degré d, alors $f(n)$ appartient à $O(n^d)$, c'est-à-dire que :
  - on « oublie » les termes de plus petit ordre,
  - on remplace le facteur du terme de plus haut rang par un.

  Exemple :
  - Si $f(n)=3n^3+2n^2+1$ alors $f(n) \in O(n^3)$.

- On utilise toujours la plus petite classe de fonctions.

  Exemple :
  - On dit « $f(n)=2n$ appartient à $O(n)$ » et non « $f(n)=2n$ appartient à $O(n^2)$ ».

Règles d'utilisation de grand-O

- Si $f(n)$ est un polynôme de degré d, alors $f(n)$ appartient à $O(n^d)$, c'est-à-dire que :
  - on « oublie » les termes de plus petit ordre,
  - on remplace le facteur du terme de plus haut rang par un.

  Exemple :
  - Si $f(n)=3n^3+2n^2+1$ alors $f(n) \in O(n^3)$.

- On utilise toujours la plus petite classe de fonctions.

  Exemple :
  - On dit « $f(n)=2n$ appartient à $O(n)$ » et non « $f(n)=2n$ appartient à $O(n^2)$ ».

- On utilise toujours l'expression de la fonction la plus simple d'une classe.

  Exemple :
  - On dit « $f(n)=3n+5$ appartient à $O(n)$ » et non « $f(n)=3n+5$ appartient à $O(3n)$ ».

Plan

- Algorithmique
- Pseudocode
- Différentes techniques de preuve
- Preuve d'un algorithme
- Complexité d'un algorithme
  - Qualité d'un algorithme
  - Temps d'exécution et complexité d'un algorithme
  - Différentes méthodes pour mesurer la complexité :
  - Complexité asymptotique :
    - Notation du O
    - Notation du Ω
    - Notation du Θ
  - Intuitions de notations asymptotiques
  - Propriétés importantes de notations asymptotiques
  - Classes principales de complexité
- Analyse asymptotique d'un algorithme
- Résumé

Notations du grand-Ω et de la borne inférieure asymptotique

- **Définitions :**
  - Pour une fonction $g(n)$ donnée, on note $\Omega(g(n))$ l'ensemble de fonctions $f(n)$ telles qu'il existe des constantes $c > 0$ et $n_0 \geq 1$ telles que

    $0 \leq c \cdot g(n) \leq f(n)$ pour tout $n \geq n_0$.

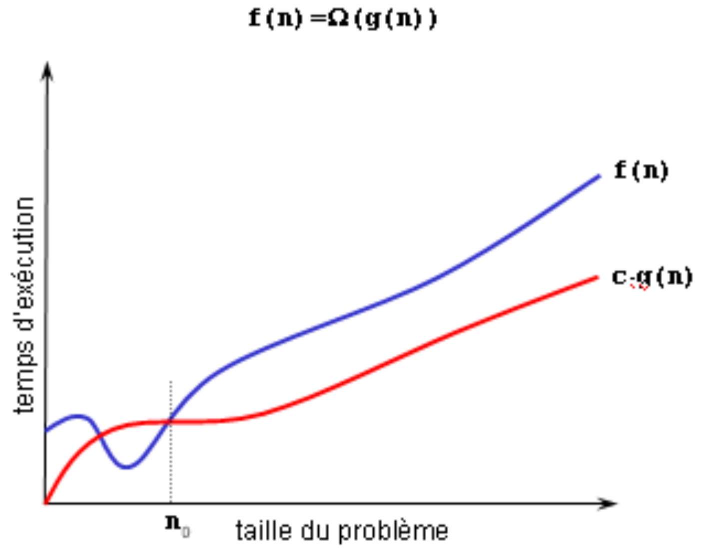
Notations du grand-Ω et de la borne inférieure asymptotique

- **Définitions :**
  - Pour une fonction $g(n)$ donnée, on note $\Omega(g(n))$ l'ensemble de fonctions $f(n)$ telles qu'il existe des constantes $c > 0$ et $n_0 \geq 1$ telles que

    $0 \leq c \cdot g(n) \leq f(n)$ pour tout $n \geq n_0$.

  - Si une fonction $f(n) \in \Omega(g(n))$, on dit que $g(n)$ est une **borne inférieure asymptotique** pour $f(n)$.

    On note : $f(n)=\Omega(g(n))$.



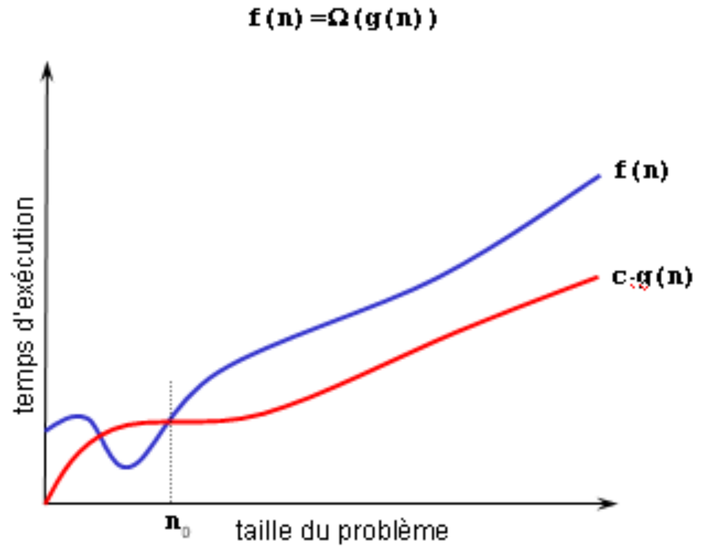
Notations du grand-Ω et de la borne inférieure asymptotique

- **Définitions :**
  - Pour une fonction $g(n)$ donnée, on note $\Omega(g(n))$ l'ensemble de fonctions $f(n)$ telles qu'il existe des constantes $c > 0$ et $n_0 \geq 1$ telles que

    $0 \leq c \cdot g(n) \leq f(n)$ pour tout $n \geq n_0$.

  - Si une fonction $f(n) \in \Omega(g(n))$, on dit que $g(n)$ est une **borne inférieure asymptotique** pour $f(n)$.

    On note : $f(n)=\Omega(g(n))$.



- **Signification :**
  - Pour toutes des grandes entrées (c'est-à-dire $n \geq n_0$), l'exécution de l'algorithme nécessite au moins $c \cdot g(n)$ étapes.

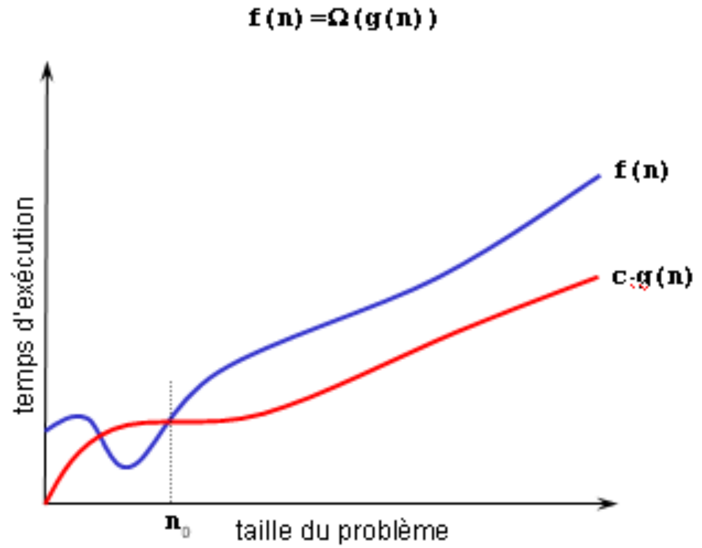
Notations du grand-Ω et de la borne inférieure asymptotique

- **Définitions :**
  - Pour une fonction $g(n)$ donnée, on note $\Omega(g(n))$ l'ensemble de fonctions $f(n)$ telles qu'il existe des constantes $c > 0$ et $n_0 \geq 1$ telles que

    $0 \leq c \bullet g(n) \leq f(n)$ pour tout $n \geq n_0$.

  - Si une fonction $f(n) \in \Omega(g(n))$, on dit que $g(n)$ est une **borne inférieure asymptotique** pour $f(n)$.

    On note : $f(n)=\Omega(g(n))$.



- **Signification :**
  - Pour toutes des grandes entrées (c'est-à-dire $n \geq n_0$), l'exécution de l'algorithme nécessite au moins $c \bullet g(n)$ étapes.
  - La borne inférieure est utilisée pour borner le temps d'exécution d'un algorithme dans le meilleur des cas.

Exemple

- Soit $f(n)=c_1 \cdot n^2+c_2 \cdot n$ où c_1 et c_2 sont deux constantes positives, alors $f(n) \in \Omega(n^2)$.
  - On doit trouver des constantes $c>0$ et $n_0 \geq 1$ telles que $c_1 \cdot n^2+c_2 \cdot n \geq c \cdot n^2$ pour $n \geq n_0$.
  - Si $c=c_1$, alors inégalité $c_1 \cdot n^2+c_2 \cdot n \geq c_1 \cdot n^2$ est vraie pour tout $n \geq 1$.
  - On peut donc choisir $c=c_1=3$ et $n_0=1$.

Plan

- Algorithmique
- Pseudocode
- Différentes techniques de preuve
- Preuve d'un algorithme
- Complexité d'un algorithme
  - Qualité d'un algorithme
  - Temps d'exécution et complexité d'un algorithme
  - Différentes méthodes pour mesurer la complexité :
  - Complexité asymptotique :
    - Notation du O
    - Notation du Ω
    - Notation du Θ
  - Intuitions de notations asymptotiques
  - Propriétés importantes de notations asymptotiques
  - Classes principales de complexité
- Analyse asymptotique d'un algorithme
- Résumé

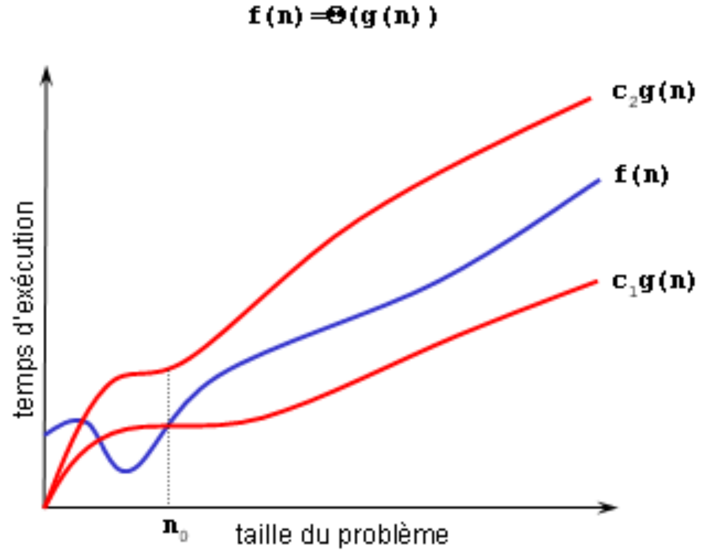
Notations du grand-Θ et de la borne asymptotique

- **Définitions :**
  - Pour une fonction $g(n)$ donnée, on note $\Theta(g(n))$ l'ensemble de fonctions $f(n)$ telles qu'il existe des constantes $c_1 > 0$, $c_2 > 0$ et $n_0 \geq 1$ telles que $0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ pour tout $n \geq n_0$.

Notations du grand-Θ et de la borne asymptotique

- **Définitions :**
  - Pour une fonction $g(n)$ donnée, on note $\Theta(g(n))$ l'ensemble de fonctions $f(n)$ telles qu'il existe des constantes $c_1 > 0$, $c_2 > 0$ et $n_0 \geq 1$ telles que $0 \leq c_1 \bullet g(n) \leq f(n) \leq c_2 \bullet g(n)$ pour tout $n \geq n_0$.
  - Si une fonction $f(n) \in \Theta(g(n))$, on dit que $g(n)$ est une **borne asymptotique** pour $f(n)$.

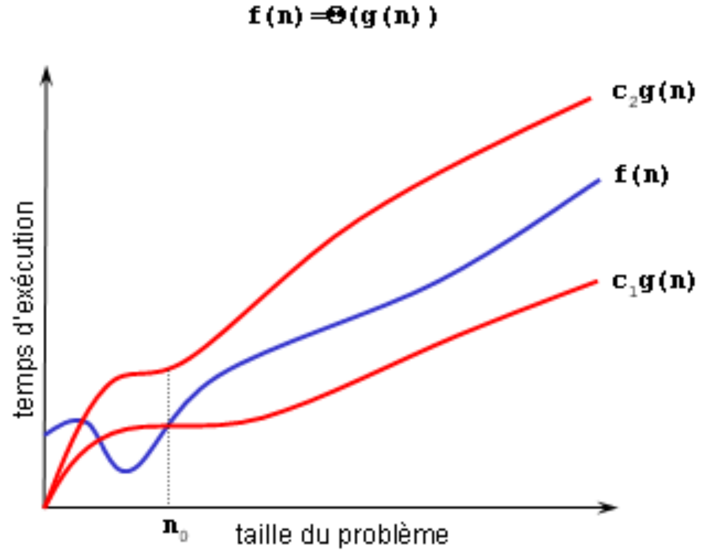
    On note : $f(n)=\Theta(g(n))$.



Notations du grand-Θ et de la borne asymptotique

- **Définitions :**
  - Pour une fonction $g(n)$ donnée, on note $\Theta(g(n))$ l'ensemble de fonctions $f(n)$ telles qu'il existe des constantes $c_1 > 0$, $c_2 > 0$ et $n_0 \geq 1$ telles que $0 \leq c_1 \bullet g(n) \leq f(n) \leq c_2 \bullet g(n)$ pour tout $n \geq n_0$.
  - Si une fonction $f(n) \in \Theta(g(n))$, on dit que $g(n)$ est une **borne asymptotique** pour $f(n)$.

    On note : $f(n)=\Theta(g(n))$.



- **Signification :**
  - Le temps d'exécution d'un algorithme $f(n)$ est dans $\Theta(g(n))$ s'il est à la fois dans $O(g(n))$ et dans $\Omega(g(n))$.

Exemple

- Soit $f(n)=1/2 \cdot n^2-3 \cdot n$, alors $f(n) \in \Theta(n^2)$.

Exemple

- Soit $f(n)=1/2 \cdot n^2-3 \cdot n$, alors $f(n) \in \Theta(n^2)$.
  - On doit trouver des constantes $c_1>0$, $c_2>0$ et $n_0 \geq 1$ telles que
    $c_1 \cdot n^2 \leq 1/2 \cdot n^2-3 \cdot n \leq c_2 \cdot n^2$ pour $n \geq n_0$.

Exemple

- Soit $f(n)=1/2 \cdot n^2-3 \cdot n$, alors $f(n) \in \Theta(n^2)$.
  - On doit trouver des constantes $c_1>0$, $c_2>0$ et $n_0 \geq 1$ telles que
    $c_1 \cdot n^2 \leq 1/2 \cdot n^2-3 \cdot n \leq c_2 \cdot n^2$ pour $n \geq n_0$.
  - La division de cette inégalité par n^2 donne :
    $c_1 \leq 1/2-3/n \leq c_2$

Exemple

- Soit $f(n)=1/2 \cdot n^2-3 \cdot n$, alors $f(n) \in \Theta(n^2)$.
  - On doit trouver des constantes $c_1>0$, $c_2>0$ et $n_0 \geq 1$ telles que
    $c_1 \cdot n^2 \leq 1/2 \cdot n^2-3 \cdot n \leq c_2 \cdot n^2$ pour $n \geq n_0$.
  - La division de cette inégalité par n^2 donne :
    $c_1 \leq 1/2-3/n \leq c_2$
  - On peut s'arranger pour que le membre droit de l'inégalité, $1/2-3/n \leq c_2$, soit valide pour n'importe quelle valeur de $n \geq 1$ en choisissant $c_2 \geq 1/2$.

Exemple

- Soit $f(n)=1/2 \bullet n^2-3 \bullet n$, alors $f(n) \in \Theta(n^2)$.
  - On doit trouver des constantes $c_1>0$, $c_2>0$ et $n_0 \geq 1$ telles que
    $c_1 \bullet n^2 \leq 1/2 \bullet n^2-3 \bullet n \leq c_2 \bullet n^2$ pour $n \geq n_0$.
  - La division de cette inégalité par n^2 donne :
    $c_1 \leq 1/2-3/n \leq c_2$
  - On peut s'arranger pour que le membre droit de l'inégalité, $1/2-3/n \leq c_2$, soit valide pour n'importe quelle valeur de $n \geq 1$ en choisissant $c_2 \geq 1/2$.
  - On peut aussi s'arranger pour que le membre gauche de l'inégalité, c'est-à-dire $c_1 \leq 1/2-3/n$, soit valide pour n'importe quelle valeur de $n \geq 7$ en choisissant $c_1 \geq 1/14$.

Exemple

- Soit $f(n)=1/2 \cdot n^2-3 \cdot n$, alors $f(n) \in \Theta(n^2)$.
  - On doit trouver des constantes $c_1>0$, $c_2>0$ et $n_0 \geq 1$ telles que
    $c_1 \cdot n^2 \leq 1/2 \cdot n^2-3 \cdot n \leq c_2 \cdot n^2$ pour $n \geq n_0$.
  - La division de cette inégalité par n^2 donne :
    $c_1 \leq 1/2-3/n \leq c_2$
  - On peut s'arranger pour que le membre droit de l'inégalité, $1/2-3/n \leq c_2$, soit valide pour n'importe quelle valeur de $n \geq 1$ en choisissant $c_2 \geq 1/2$.
  - On peut aussi s'arranger pour que le membre gauche de l'inégalité, c'est-à-dire $c_1 \leq 1/2-3/n$, soit valide pour n'importe quelle valeur de $n \geq 7$ en choisissant $c_1 \geq 1/14$.
  - On peut donc vérifier que $1/2 \cdot n^2-3 \cdot n \in \Theta(n^2)$ en prenant $c_1=1/14$, $c_2=1/2$ et $n_0=7$.

Exemple

- Soit $f(n)=1/2 \cdot n^2-3 \cdot n$, alors $f(n) \in \Theta(n^2)$.
  - On doit trouver des constantes $c_1>0$, $c_2>0$ et $n_0 \geq 1$ telles que
    $c_1 \cdot n^2 \leq 1/2 \cdot n^2-3 \cdot n \leq c_2 \cdot n^2$ pour $n \geq n_0$.
  - La division de cette inégalité par n^2 donne :
    $c_1 \leq 1/2-3/n \leq c_2$
  - On peut s'arranger pour que le membre droit de l'inégalité, $1/2-3/n \leq c_2$, soit valide pour n'importe quelle valeur de $n \geq 1$ en choisissant $c_2 \geq 1/2$.
  - On peut aussi s'arranger pour que le membre gauche de l'inégalité, c'est-à-dire $c_1 \leq 1/2-3/n$, soit valide pour n'importe quelle valeur de $n \geq 7$ en choisissant $c_1 \geq 1/14$.
  - On peut donc vérifier que $1/2 \cdot n^2-3 \cdot n \in \Theta(n^2)$ en prenant $c_1=1/14$, $c_2=1/2$ et $n_0=7$.
- Soit $f(n)=6n^3$, alors $f(n) \notin \Theta(n^2)$.

Exemple

- Soit $f(n)=1/2 \cdot n^2-3 \cdot n$, alors $f(n) \in \Theta(n^2)$.
  - On doit trouver des constantes $c_1>0$, $c_2>0$ et $n_0 \geq 1$ telles que
    $c_1 \cdot n^2 \leq 1/2 \cdot n^2-3 \cdot n \leq c_2 \cdot n^2$ pour $n \geq n_0$.
  - La division de cette inégalité par n^2 donne :
    $c_1 \leq 1/2-3/n \leq c_2$
  - On peut s'arranger pour que le membre droit de l'inégalité, $1/2-3/n \leq c_2$, soit valide pour n'importe quelle valeur de $n \geq 1$ en choisissant $c_2 \geq 1/2$.
  - On peut aussi s'arranger pour que le membre gauche de l'inégalité, c'est-à-dire $c_1 \leq 1/2-3/n$, soit valide pour n'importe quelle valeur de $n \geq 7$ en choisissant $c_1 \geq 1/14$.
  - On peut donc vérifier que $1/2 \cdot n^2-3 \cdot n \in \Theta(n^2)$ en prenant $c_1=1/14$, $c_2=1/2$ et $n_0=7$.
- Soit $f(n)=6n^3$, alors $f(n) \notin \Theta(n^2)$.
  - On doit trouver des constantes $c_1>0$, $c_2>0$ et $n_0 \geq 1$ telles que
    $c_1 \cdot n^2 \leq 6n^3 \leq c_2 \cdot n^2$ pour $n \geq n_0$.

Exemple

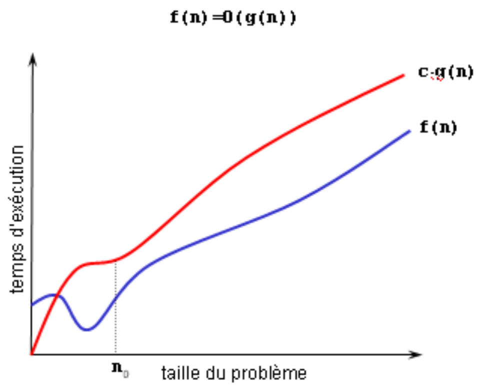
- Soit $f(n)=1/2 \cdot n^2-3 \cdot n$, alors $f(n) \in \Theta(n^2)$.
  - On doit trouver des constantes $c_1>0$, $c_2>0$ et $n_0 \geq 1$ telles que
    $c_1 \cdot n^2 \leq 1/2 \cdot n^2-3 \cdot n \leq c_2 \cdot n^2$ pour $n \geq n_0$.
  - La division de cette inégalité par n^2 donne :
    $c_1 \leq 1/2-3/n \leq c_2$
  - On peut s'arranger pour que le membre droit de l'inégalité, $1/2-3/n \leq c_2$, soit valide pour n'importe quelle valeur de $n \geq 1$ en choisissant $c_2 \geq 1/2$.
  - On peut aussi s'arranger pour que le membre gauche de l'inégalité, c'est-à-dire $c_1 \leq 1/2-3/n$, soit valide pour n'importe quelle valeur de $n \geq 7$ en choisissant $c_1 \geq 1/14$.
  - On peut donc vérifier que $1/2 \cdot n^2-3 \cdot n \in \Theta(n^2)$ en prenant $c_1=1/14$, $c_2=1/2$ et $n_0=7$.
- Soit $f(n)=6n^3$, alors $f(n) \notin \Theta(n^2)$.
  - On doit trouver des constantes $c_1>0$, $c_2>0$ et $n_0 \geq 1$ telles que
    $c_1 \cdot n^2 \leq 6n^3 \leq c_2 \cdot n^2$ pour $n \geq n_0$.
  - Si on simplifie le membre droit de l'inégalité, c'est-à-dire $6n^3 \leq c_2 \cdot n^2$, on obtient $n \leq c_2/6$. Cette inégalité ne peut pas être satisfaite car c_2 doit être constante.

Plan

- Algorithmique
- Pseudocode
- Différentes techniques de preuve
- Preuve d'un algorithme
- Complexité d'un algorithme
  - Qualité d'un algorithme
  - Temps d'exécution et complexité d'un algorithme
  - Différentes méthodes pour mesurer la complexité
  - Complexité asymptotique
  - Intuitions de notations asymptotiques
  - Propriétés importantes de notations asymptotiques
  - Classes principales de complexité
- Analyse asymptotique d'un algorithme
- Résumé

Intuitions de notations asymptotiques

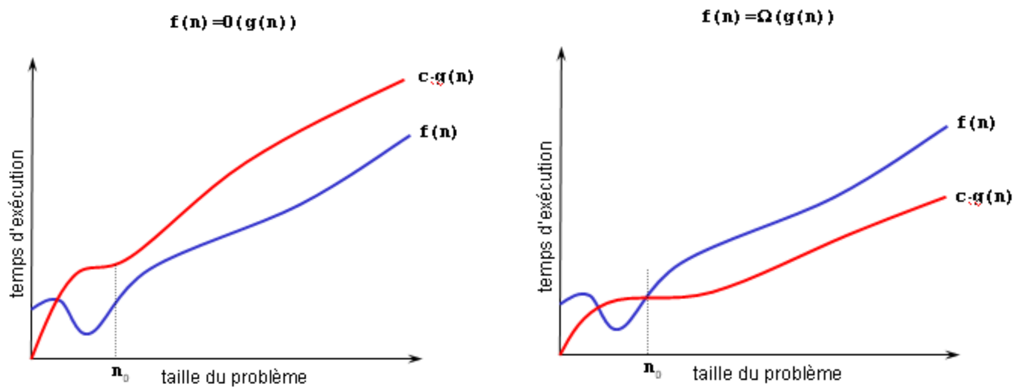
- **Grand-O :**
  - `f(n)` appartient à l'ensemble `O(g(n))` si `f(n)` est asymptotiquement plus petite ou égale à `g(n)`.



Intuitions de notations asymptotiques

- **Grand-O :**
  - `f(n)` appartient à l'ensemble `O(g(n))` si `f(n)` est asymptotiquement plus petite ou égale à `g(n)`.

- **Grand-Ω :**
  - `f(n)` appartient à l'ensemble `Ω(g(n))` si `f(n)` est asymptotiquement plus grande ou égale à `g(n)`.

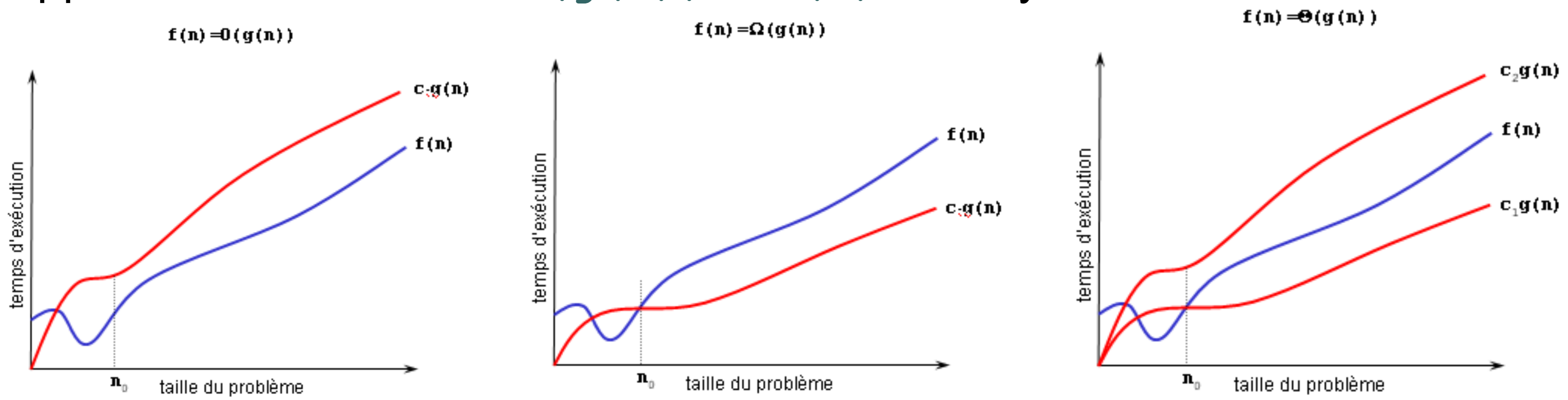


Intuitions de notations asymptotiques

- **Grand-O :**
  - `f(n)` appartient à l'ensemble `O(g(n))` si `f(n)` est asymptotiquement plus petite ou égale à `g(n)`.

- **Grand-Ω :**
  - `f(n)` appartient à l'ensemble `Ω(g(n))` si `f(n)` est asymptotiquement plus grande ou égale à `g(n)`.

- **Grand-Θ :**
  - `f(n)` appartient à l'ensemble `Θ(g(n))` si `f(n)` est asymptotiquement égale à `g(n)`.



Plan

- Algorithmique
- Pseudocode
- Différentes techniques de preuve
- Preuve d'un algorithme
- Complexité d'un algorithme
  - Qualité d'un algorithme
  - Temps d'exécution et complexité d'un algorithme
  - Différentes méthodes pour mesurer la complexité
  - Complexité asymptotique
  - Intuitions de notations asymptotiques
  - Propriétés importantes de notations asymptotiques
  - Classes principales de complexité
- Analyse asymptotique d'un algorithme
- Résumé

Propriétés importantes de notations asymptotiques

- **Réflexivité :**
  - $f(n) \in O(f(n))$
  - $f(n) \in \Omega(f(n))$
  - $f(n) \in \Theta(f(n))$

Propriétés importantes de notations asymptotiques

- **Réflexivité :**
  - $f(n) \in O(f(n))$
  - $f(n) \in \Omega(f(n))$
  - $f(n) \in \Theta(f(n))$

- **Symétrie :**
  - $f(n) \in O(g(n)) \Leftrightarrow g(n) \in O(f(n))$
  - $f(n) \in \Omega(g(n)) \Leftrightarrow g(n) \in \Omega(f(n))$
  - $f(n) \in \Theta(g(n)) \Leftrightarrow g(n) \in \Theta(f(n))$

Propriétés importantes de notations asymptotiques

- **Réflexivité :**
  - $f(n) \in O(f(n))$
  - $f(n) \in \Omega(f(n))$
  - $f(n) \in \Theta(f(n))$

- **Symétrie :**
  - $f(n) \in O(g(n)) \Leftrightarrow g(n) \in O(f(n))$
  - $f(n) \in \Omega(g(n)) \Leftrightarrow g(n) \in \Omega(f(n))$
  - $f(n) \in \Theta(g(n)) \Leftrightarrow g(n) \in \Theta(f(n))$

- **Symétrie transposée :**
  - $f(n) \in O(g(n)) \Leftrightarrow g(n) \in \Omega(f(n))$

Propriétés importantes de notations asymptotiques

- **Réflexivité :**
  - $f(n) \in O(f(n))$
  - $f(n) \in \Omega(f(n))$
  - $f(n) \in \Theta(f(n))$

- **Symétrie :**
  - $f(n) \in O(g(n)) \Leftrightarrow g(n) \in O(f(n))$
  - $f(n) \in \Omega(g(n)) \Leftrightarrow g(n) \in \Omega(f(n))$
  - $f(n) \in \Theta(g(n)) \Leftrightarrow g(n) \in \Theta(f(n))$

- **Symétrie transposée :**
  - $f(n) \in O(g(n)) \Leftrightarrow g(n) \in \Omega(f(n))$

- **Transitivité :**
  - Si $f(n) \in O(g(n))$ et $g(n) \in O(h(n))$, alors $f(n) \in O(h(n))$.
  - Si $f(n) \in \Omega(g(n))$ et $g(n) \in \Omega(h(n))$, alors $f(n) \in \Omega(h(n))$.
  - Si $f(n) \in \Theta(g(n))$ et $g(n) \in \Theta(h(n))$, alors $f(n) \in \Theta(h(n))$.

Propriétés importantes de notations asymptotiques

- **Addition :**
  - Si $f_1(n) \in O(g_1(n))$ et $f_2(n) \in O(g_2(n))$, alors :
    - $(f_1+f_2)(n) = O(\max(g_1(n),\ g_2(n)))$
    - $(f_1+f_2)(n) = O(g_1(n)+g_2(n))$

Propriétés importantes de notations asymptotiques

- **Addition :**
  - Si $f_1(n) \in O(g_1(n))$ et $f_2(n) \in O(g_2(n))$, alors :
    - $(f_1+f_2)(n) = O(\max(g_1(n), g_2(n)))$
    - $(f_1+f_2)(n) = O(g_1(n)+g_2(n))$

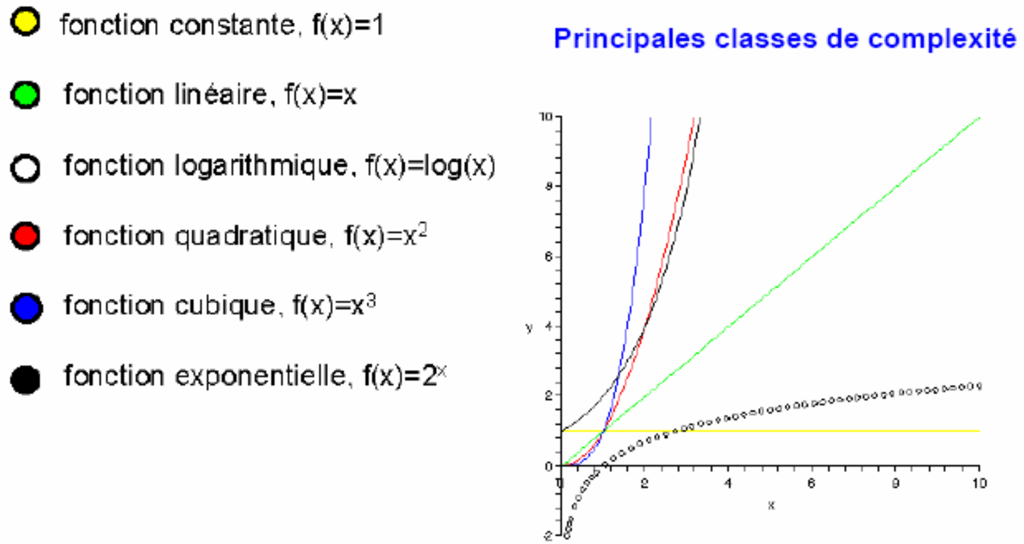
- **Multiplication :**
  - Si $f_1(n) \in O(g_1(n))$ et $f_2(n) \in O(g_2(n))$, alors :
    - $(f_1 \cdot f_2)(n) = O(g_1(n) \cdot g_2(n))$

Plan

- Algorithmique
- Pseudocode
- Différentes techniques de preuve
- Preuve d'un algorithme
- Complexité d'un algorithme
  - Qualité d'un algorithme
  - Temps d'exécution et complexité d'un algorithme
  - Différentes méthodes pour mesurer la complexité
  - Complexité asymptotique
  - Intuitions de notations asymptotiques
  - Propriétés importantes de notations asymptotiques
  - Classes principales de complexité
- Analyse asymptotique d'un algorithme
- Résumé

Classes principales de complexité

- **constante** $\approx$ **`1`** **:**
  - une opération élémentaire,
  - une séquence d'opérations indépendantes des données d'entrée.
- **logarithme** $\approx$ **`log(n)`** **:**
  - la hauteur d'un arbre équilibré,
  - recherche binaire.
- **linéaire** $\approx$ **`n`** **:**
  - parcours des données.
- **N-log-N** $\approx$ **`n•log(n)`** **:**
  - combinaison des deux précédents,
  - tri.
- **quadratique** $\approx$ n^2 **:**
  - deux boucles imbriquées,
  - énumération de tous les couples.
- **cubique** $\approx$ n^3 **:**
  - trois boucles imbriquées.
- **exponentielle** $\approx$ 2^n **:**
  - combinaisons,
  - énumération de toutes les combinaisons (sommes géométriques).



Plan

- Algorithmique
- Pseudocode
- Différentes techniques de preuve :
- Preuve d'un algorithme
- Complexité d'un algorithme
- Analyse asymptotique d'un algorithme
- Résumé

Analyse asymptotique d’un algorithme

- L’analyse asymptotique d’un algorithme détermine le temps d’exécution de cet algorithme dans le notation grand-O.

Analyse asymptotique d’un algorithme

- L’analyse asymptotique d’un algorithme détermine le temps d’exécution de cet algorithme dans le notation grand-O.

- **Pour réaliser l’analyse asymptotique d’un algorithme :**
  - On calcule le nombre d’opérations primitives exécutées par l’algorithme. Ce nombre d’opérations est exprimé sous la forme d’une fonction qui dépend du nombre d’entrées.
  - Puis, on représente cette fonction dans la notation de grand-O.

Analyse asymptotique d'un algorithme

- L'analyse asymptotique d'un algorithme détermine le temps d'exécution de cet algorithme dans le notation grand-`O`.

- **Pour réaliser l'analyse asymptotique d'un algorithme :**
  - On calcule le nombre d'opérations primitives exécutées par l'algorithme. Ce nombre d'opérations est exprimé sous la forme d'une fonction qui dépend du nombre d'entrées.
  - Puis, on représente cette fonction dans la notation de grand-`O`.

- **Exemple :**
  - L'algorithme `arrayMax` exécute `8n-4` opérations primitives dans le pire des cas.
  - On dit que l'algorithme `arrayMax` s'exécute en temps `O(n)`.

Analyse asymptotique d’un algorithme

- L’analyse asymptotique d’un algorithme détermine le temps d’exécution de cet algorithme dans le notation grand-`O`.

- **Pour réaliser l’analyse asymptotique d’un algorithme :**
  - On calcule le nombre d’opérations primitives exécutées par l’algorithme. Ce nombre d’opérations est exprimé sous la forme d’une fonction qui dépend du nombre d’entrées.
  - Puis, on représente cette fonction dans la notation de grand-`O`.

- **Exemple :**
  - L’algorithme `arrayMax` exécute `8n-4` opérations primitives dans le pire des cas.
  - On dit que l’algorithme `arrayMax` s’exécute en temps `O(n)`.

- **Remarque :**
  - Lorsque l'on compte les opérations primitives, on peut négliger les facteurs constants ainsi que les termes de plus bas ordre car de toutes façons ils vont être ignorés.

Plan

- Algorithmique
- Pseudocode
- Différentes techniques de preuve :
- Preuve d’un algorithme
- Complexité d’un algorithme
- Analyse asymptotique d’un algorithme
  - Règles pour dériver la complexité d’un algorithme
  - Exemples
- Résumé

Règles pour dériver la complexité d'un algorithme (1/3)

- **Règle 1** :
  - La complexité d'un ensemble d'instructions est la somme des complexités de chacune d'elles.

Règles pour dériver la complexité d'un algorithme (1/3)

- **Règle 1** :
  - La complexité d'un ensemble d'instructions est la somme des complexités de chacune d'elles.
- **Règle 2** :
  - Les opérations élémentaires telles que :
    - affectation,
    - test,
    - accès à un tableau,
    - opérations logiques et arithmétiques,
    - lecture ou écriture d'une variable simple,
    - etc.

    comptent la même chose. Elles sont en $O(1)$ (ou en $\Theta(1)$).
  - Une suite donnée d'opérations élémentaires compte pour $O(1)$.

Règles pour dériver la complexité d'un algorithme (2/3)

- **Règle 3 (instructions de contrôle) :**
  - Instruction `if` :
    - Prendre le maximum entre le bloc d'instructions de `then` et celui de `else`.

Règles pour dériver la complexité d'un algorithme (2/3)

- **Règle 3 (instructions de contrôle) :**
  - Instruction `if` :
    - Prendre le maximum entre le bloc d'instructions de `then` et celui de `else`.
  - Instruction `switch` :
    - Prendre le maximum parmi les complexités des blocs d'instructions des différents cas de cette instruction.

Règles pour dériver la complexité d'un algorithme (2/3)

- **Règle 3 (instructions de contrôle) :**
  - Instruction `if` :
    - Prendre le maximum entre le bloc d'instructions de `then` et celui de `else`.
  - Instruction `switch` :
    - Prendre le maximum parmi les complexités des blocs d'instructions des différents cas de cette instruction.

- **Règle 4 (instructions de répétition) :**
  - Boucle `for` :
    - La complexité de la boucle `for` est calculée par la complexité du corps de cette boucle multipliée par le nombre de fois où elle est répétée.

Règles pour dériver la complexité d'un algorithme (2/3)

- **Règle 3 (instructions de contrôle) :**
  - Instruction `if` :
    - Prendre le maximum entre le bloc d'instructions de `then` et celui de `else`.
  - Instruction `switch` :
    - Prendre le maximum parmi les complexités des blocs d'instructions des différents cas de cette instruction.

- **Règle 4 (instructions de répétition) :**
  - Boucle `for` :
    - La complexité de la boucle `for` est calculée par la complexité du corps de cette boucle multipliée par le nombre de fois où elle est répétée.
  - Boucle `while` :
    - En règle générale, pour déterminer la complexité d'une boucle `while`, il faudra avant tout déterminer le nombre de fois où cette boucle est répétée, ensuite le multiplier par la complexité du corps de cette boucle.

Règles pour dériver la complexité d’un algorithme (3/3)

- **Règle 5 (procédures et fonctions) :**
  - La complexité des procédures et fonctions est déterminée par celle de leur corps. Notons qu’on fait la distinction entre les fonctions récursives et celles qui ne le sont pas :

Règles pour dériver la complexité d'un algorithme (3/3)

- **Règle 5 (procédures et fonctions) :**
  - La complexité des procédures et fonctions est déterminée par celle de leur corps. Notons qu'on fait la distinction entre les fonctions récursives et celles qui ne le sont pas :
    - Dans le cas de fonctions **récursives**, le temps de calcul est exprimé comme une relation de récurrence.

Règles pour dériver la complexité d'un algorithme (3/3)

- **Règle 5 (procédures et fonctions) :**
  - La complexité des procédures et fonctions est déterminée par celle de leur corps. Notons qu'on fait la distinction entre les fonctions récursives et celles qui ne le sont pas :
    - Dans le cas de fonctions **récursives**, le temps de calcul est exprimé comme une relation de récurrence.
    - Pour les fonctions **non récursives**, leur complexité temporelle se calcule en sachant que l'appel à une fonction prend un temps constant en $O(1)$ (ou en $\Theta(1)$).

Plan

- Algorithmique
- Pseudocode
- Différentes techniques de preuve :
- Preuve d'un algorithme
- Complexité d'un algorithme
- Analyse asymptotique d'un algorithme
  - Règles pour dériver la complexité d'un algorithme
  - Exemples :
    - Produit de deux matrices
    - Recherche dichotomique d'un élément dans un tableau
    - Factorielle
- Résumé

Exemple 1 :

produit de deux matrices (1/2)

- Soient `A[n,p]=(a`$_{ij}$`)` et `B[p,m] =(b`$_{ij}$`)` deux matrices, alors le produit entre `A[n,p]` et `B[p,m]` est la matrice `C[n,m] =(c`$_{ij}$`)` où :

- **Exemple :**

A[3,2] B[2,4] C[3,4]

Exemple 1 :

produit de deux matrices (2/2)

```
Algorithm matrixMultiplication(A[n,p],B[p,m],C[n,m], n,p,m)

  Input arrays A, B, C of n×p, p×m, n×m integers
  Output array C that is the product of the arrays A et B

Begin

  for i←1 to n do
    for j ← 1 to m do
      sum ← 0;
      for k ← 1 to p do
        sum ← sum+A[i,k]•B[k,j];
      end;
      C[i,j] ← sum;
    end;
  end;
  return C;

End.
```

Exemple 1 :

produit de deux matrices (2/2)

```
Algorithm matrixMultiplication(A[n,p],B[p,m],C[n,m], n,p,m)

  Input arrays A, B, C of n×p, p×m, n×m integers
  Output array C that is the product of the arrays A et B

Begin

  for i←1 to n do
    for j ← 1 to m do
      sum ← 0;
      for k ← 1 to p do
        sum ← sum+A[i,k]•B[k,j];
      end;
      C[i,j] ← sum;
    end;
  end;
  return C;

End.
```

O(n • m • p) car la boucle est itérée n fois.

Exemple 1 :

produit de deux matrices (2/2)

```
Algorithm matrixMultiplication(A[n,p],B[p,m],C[n,m], n,p,m)

  Input arrays A, B, C of n×p, p×m, n×m integers
  Output array C that is the product of the arrays A et B

Begin

  for i←1 to n do
    for j ← 1 to m do
      sum ← 0;
      for k ← 1 to p do
        sum ← sum+A[i,k]•B[k,j];
      end;
      C[i,j] ← sum;
    end;
  end;
  return C;

End.
```

$O(n \cdot m \cdot p)$ est la complexité de l'algorithme.

Exemple 1 :

produit de deux matrices (2/2)

```
Algorithm matrixMultiplication(A[n,p],B[p,m],C[n,m], n,p,m)

  Input arrays A, B, C of n×p, p×m, n×m integers
  Output array C that is the product of the arrays A et B

Begin

  for i←1 to n do
    for j ← 1 to m do
      sum ← 0;
      for k ← 1 to p do
        sum ← sum+A[i,k]•B[k,j];
      end;
      C[i,j] ← sum;
    end;
  end;
  return C;

End.
```

O(n • m • p) est la complexité de l'algorithme.

Remarque : Pour cet algorithme, il n'y pas lieu de distinguer les différentes complexités. Dans tous les cas, nous aurons à effectuer ce nombre d'opérations.

Plan

- Algorithmique
- Pseudocode
- Différentes techniques de preuve :
- Preuve d’un algorithme
- Complexité d’un algorithme
- Analyse asymptotique d’un algorithme
  - Règles pour dériver la complexité d’un algorithme
  - Exemples :
    - Produit de deux matrices
    - Recherche dichotomique d’un élément dans un tableau
    - Factorielle
- Résumé

Exemple 2 :

recherche dichotomique d'un élément dans un tableau (1/2)

- Soient `A[n]` un tableau de `n` entiers déjà triés et `e` un entier. Trouver la position de l'élément `e` dans le tableau `A[n]` en utilisant la recherche dichotomique.

- **Exemple :**

`A[10]`

`e`=7

1	2	3	4	5	6	7	8	9	10
2	4	5	7	8	9	10	15	20	21

Exemple 2 :
recherche dichotomique d'un élément dans un tableau (1/2)

- Soient `A[n]` un tableau de `n` entiers déjà triés et `e` un entier. Trouver la position de l'élément `e` dans le tableau `A[n]` en utilisant la recherche dichotomique.

- **Exemple :**

`A[10]`

`e`=7

1	2	3	4	5	6	7	8	9	10
2	4	5	7	8	9	10	15	20	21

`inf` `sup`

Exemple 2 :
recherche dichotomique d'un élément dans un tableau (1/2)

- Soient **`A[n]`** un tableau de **`n`** entiers déjà triés et **`e`** un entier. Trouver la position de l'élément **`e`** dans le tableau **`A[n]`** en utilisant la recherche dichotomique.

- **Exemple :**

`A[10]`

1	2	3	4	5	6	7	8	9	10
2	4	5	7	8	9	10	15	20	21

`e`=7

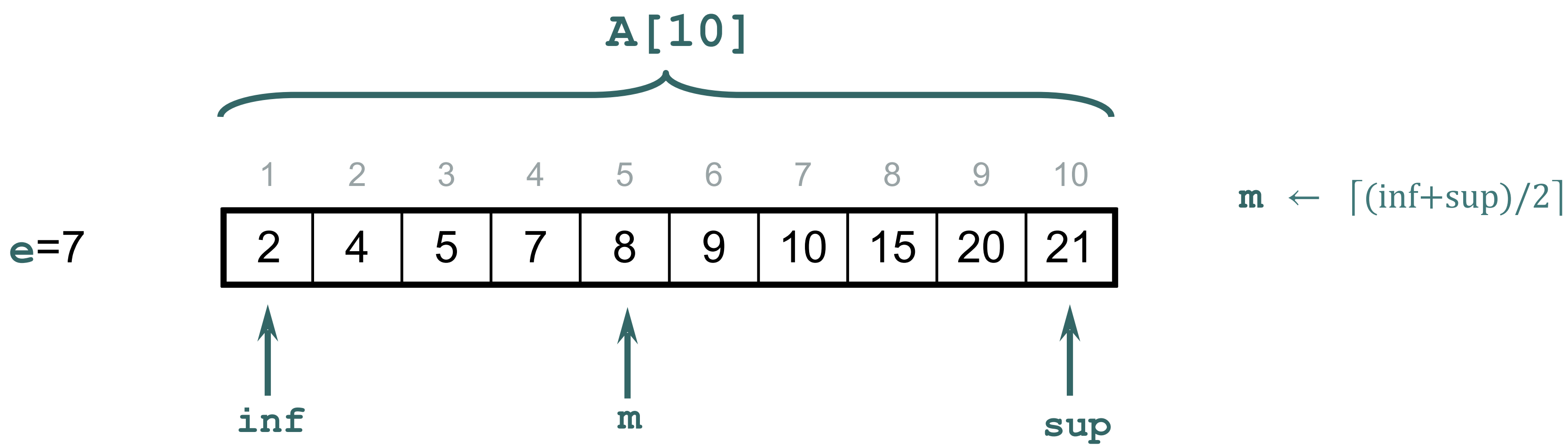
`m` $\leftarrow \lceil (\mathrm{inf}+\mathrm{sup})/2 \rceil$

`inf` **`sup`**

Exemple 2 :
recherche dichotomique d'un élément dans un tableau (1/2)

- Soient `A[n]` un tableau de `n` entiers déjà triés et `e` un entier. Trouver la position de l'élément `e` dans le tableau `A[n]` en utilisant la recherche dichotomique.

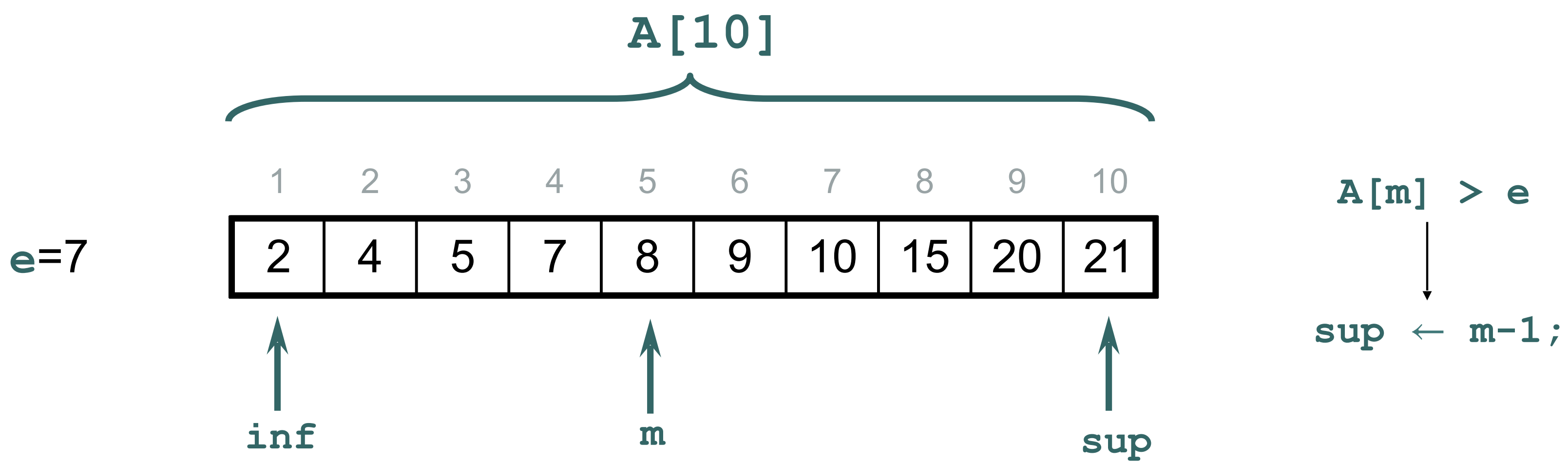
- **Exemple :**



Exemple 2 :
recherche dichotomique d'un élément dans un tableau (1/2)

- Soient `A[n]` un tableau de `n` entiers déjà triés et `e` un entier. Trouver la position de l'élément `e` dans le tableau `A[n]` en utilisant la recherche dichotomique.

- **Exemple :**



Exemple 2 :
recherche dichotomique d'un élément dans un tableau (1/2)

- Soient `A[n]` un tableau de `n` entiers déjà triés et `e` un entier. Trouver la position de l'élément `e` dans le tableau `A[n]` en utilisant la recherche dichotomique.

- **Exemple :**

`A[10]`

`e`=7

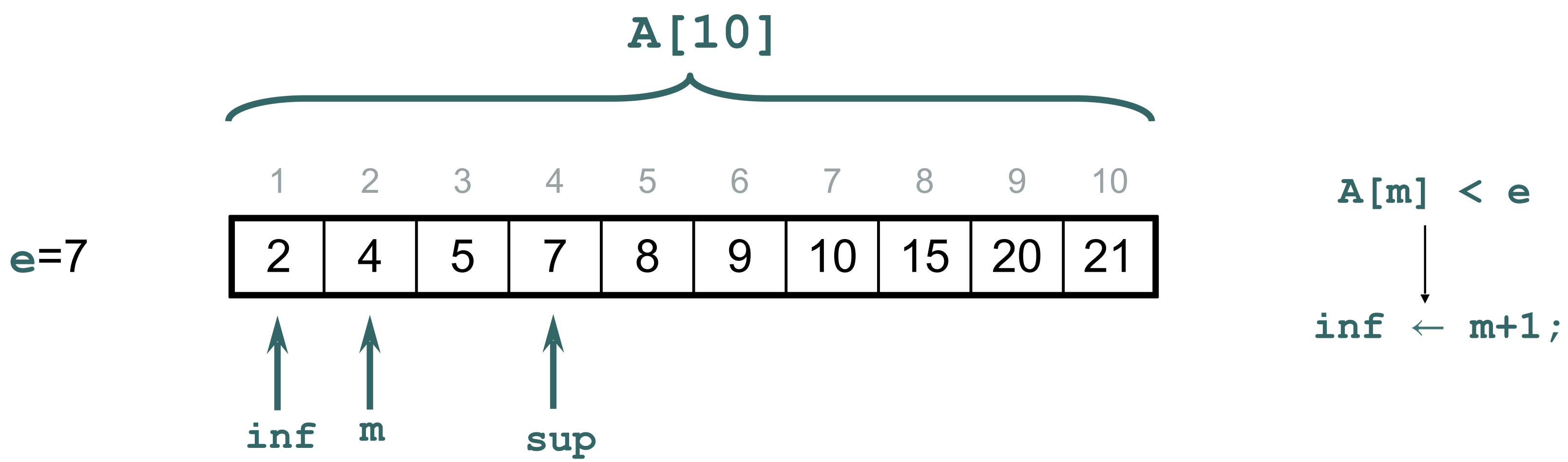
1	2	3	4	5	6	7	8	9	10
2	4	5	7	8	9	10	15	20	21

`inf` `sup`

Exemple 2 :
recherche dichotomique d'un élément dans un tableau (1/2)

- Soient `A[n]` un tableau de `n` entiers déjà triés et `e` un entier. Trouver la position de l'élément `e` dans le tableau `A[n]` en utilisant la recherche dichotomique.

- **Exemple :**



Exemple 2 :
recherche dichotomique d'un élément dans un tableau (1/2)

- Soient **`A[n]`** un tableau de **`n`** entiers déjà triés et **`e`** un entier. Trouver la position de l'élément **`e`** dans le tableau **`A[n]`** en utilisant la recherche dichotomique.

- **Exemple :**

`A[10]`

`e`=7

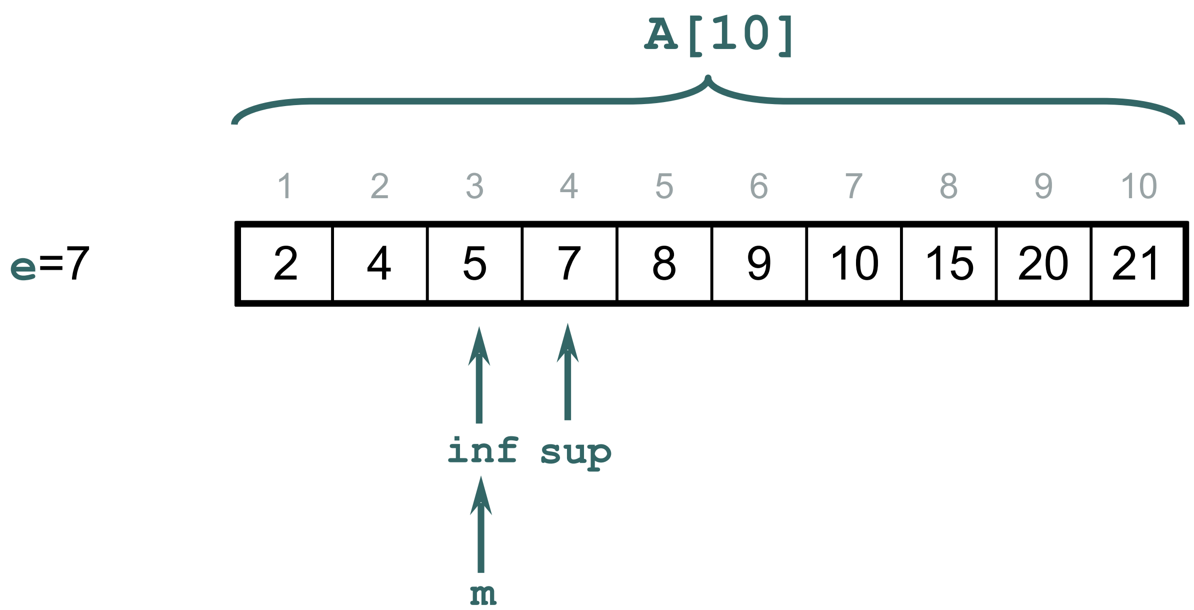
1	2	3	4	5	6	7	8	9	10
2	4	5	7	8	9	10	15	20	21

`inf` **`sup`**

Exemple 2 :
recherche dichotomique d'un élément dans un tableau (1/2)

- Soient **`A[n]`** un tableau de **`n`** entiers déjà triés et **`e`** un entier. Trouver la position de l'élément **`e`** dans le tableau **`A[n]`** en utilisant la recherche dichotomique.

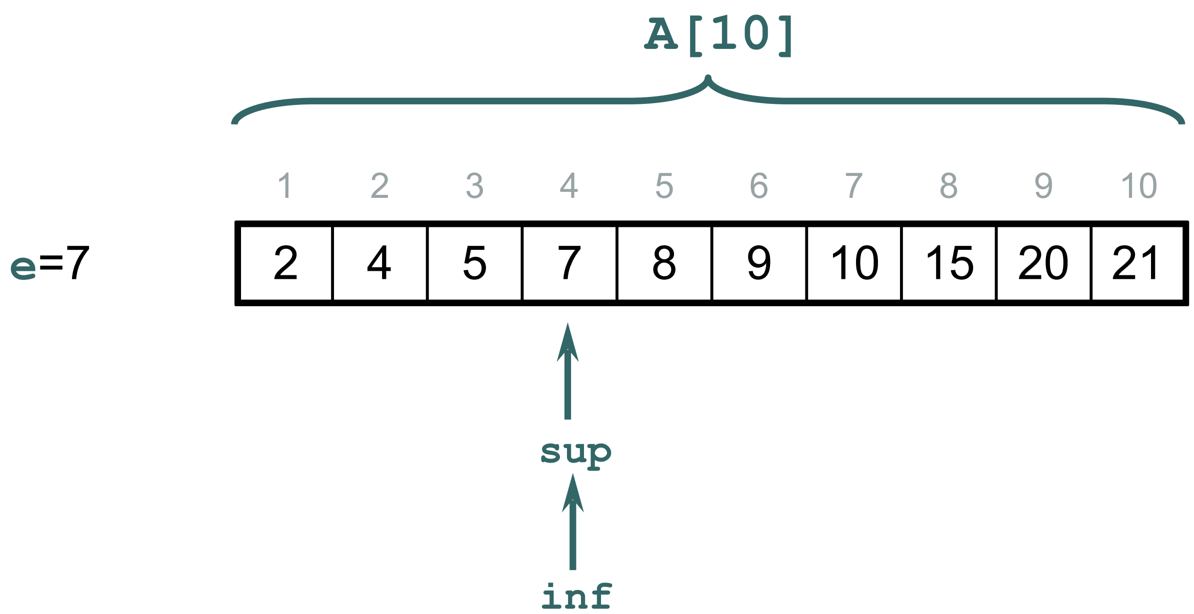
- **Exemple :**



Exemple 2 :
recherche dichotomique d'un élément dans un tableau (1/2)

- Soient `A[n]` un tableau de `n` entiers déjà triés et `e` un entier. Trouver la position de l'élément `e` dans le tableau `A[n]` en utilisant la recherche dichotomique.

- **Exemple :**



Exemple 2 :

recherche dichotomique d'un élément dans un tableau (1/2)

- Soient **`A[n]`** un tableau de **`n`** entiers déjà triés et **`e`** un entier. Trouver la position de l'élément **`e`** dans le tableau **`A[n]`** en utilisant la recherche dichotomique.

- **Exemple :**

`A[10]`

	1	2	3	4	5	6	7	8	9	10
`e`=7	2	4	5	7	8	9	10	15	20	21

Exemple 2 :
recherche dichotomique d'un élément dans un tableau (1/2)

```
Algorithm search(A[n], n, e)
  Input array A of n integers, integer e
  Output position of the integer e in the array A
Begin
  inf ← 1; sup ← n; found ← false;
  while sup ≥ inf and !found do
    m ← ⌈(inf+sup)/2⌉
    if e=A[m] then
      found ← true;
    else if e<A[m] then
           sup ← m-1;
         else
           inf ← m+1;
         end;
    end;
  end;


  if found=true then
    return -1;
  end;
  return m;
End.
```

Complexité dans le meilleur de cas :

Il n'est pas difficile de voir que le cas favorable se présente quand la valeur recherchée `e` est au milieu du tableau `A`. Autrement dit, la boucle `while` ne sera itérée qu'une seule fois. Dans ce cas, l'algorithme aura effectué un nombre constant d'opérations. C'est-à-dire la complexité de l'algorithme est `O(1)`.

Exemple 2 :

recherche dichotomique d'un élément dans un tableau (1/2)

```
Algorithm search(A[n], n, e)
  Input array A of n integers, integer e
  Output position of the integer e in the array A
Begin
  inf ← 1; sup ← n; found ← false;
  while sup ≥ inf and !found do
    m ← ⌈(inf+sup)/2⌉
    if e=A[m] then
      found ← true;
    else if e<A[m] then
           sup ← m-1;
         else
           inf ← m+1;
         end;
    end;
  end;


  if found=true then
    return -1;
  end;
  return m;
End.
```

Complexité dans le pire de cas :

Ce cas se présente quand l'élément **`e`** n'existe pas. Dans ce cas, la boucle **`while`** sera itérée jusqu'à ce que la variable **`sup<inf`**. Le problème est de savoir combien d'itérations sont nécessaires pour que cette condition soit vérifiée. Pour le savoir, il suffit de constater, qu'après chaque itération, l'ensemble de recherche est divisé par deux.

Exemple 2 :
recherche dichotomique d'un élément dans un tableau (1/2)

```
Algorithm search(A[n], n, e)
  Input array A of n integers, integer e
  Output position of the integer e in the array A
Begin
  inf ← 1; sup ← n; found ← false;
  while sup ≥ inf and !found do
    m ← ⌈(inf+sup)/2⌉
    if e=A[m] then
      found ← true;
    else if e<A[m] then
           sup ← m-1;
         else
           inf ← m+1;
         end;
    end;
  end;


  if found=true then
    return -1;
  end;
  return m;
End.
```

Complexité dans le pire de cas :

Ce cas se présente quand l'élément **e** n'existe pas. Dans ce cas, la boucle **while** sera itérée jusqu'à ce que la variable **sup<inf**. Le problème est de savoir combien d'itérations sont nécessaires pour que cette condition soit vérifiée. Pour le savoir, il suffit de constater, qu'après chaque itération, l'ensemble de recherche est divisé par deux.

Nombre d'itérations	Intervalle de recherche
0	n
1	n/2
2	n/4
3	n/8
…	…
k	$n/2^k$

Exemple 2 :

recherche dichotomique d'un élément dans un tableau (1/2)

```
Algorithm search(A[n], n, e)
  Input array A of n integers, integer e
  Output position of the integer e in the array A
Begin
  inf ← 1; sup ← n; found ← false;
  while sup ≥ inf and !found do
    m ← ⌈(inf+sup)/2⌉
    if e=A[m] then
      found ← true;
    else if e<A[m] then
           sup ← m-1;
         else
           inf ← m+1;
         end;
    end;
  end;


  if found=true then
    return -1;
  end;
  return m;
End.
```

Complexité dans le pire de cas :

Ce cas se présente quand l'élément **`e`** n'existe pas. Dans ce cas, la boucle **`while`** sera itérée jusqu'à ce que la variable **`sup<inf`**. Le problème est de savoir combien d'itérations sont nécessaires pour que cette condition soit vérifiée. Pour le savoir, il suffit de constater, qu'après chaque itération, l'ensemble de recherche est divisé par deux.

On arrêtera les itérations de la boucle **`while`** dès que la condition suivante sera vérifiée :

$$n/2^k = 1$$

$$k = O(\log(n))$$

Autrement dit, la complexité de cet algorithme dans le pire de cas est `O(log(n))`.

Plan

- Algorithmique
- Pseudocode
- Différentes techniques de preuve :
- Preuve d'un algorithme
- Complexité d'un algorithme
- Analyse asymptotique d'un algorithme
  - Règles pour dériver la complexité d'un algorithme
  - Exemples :
    - Produit de deux matrices
    - Recherche dichotomique d'un élément dans un tableau
    - Factorielle
- Résumé

Exemple 3 :
factorielle

- Soit `n` un entier. Calculer la factorielle de `n`.

  Rappel : `n! = n • (n-1)•(n-2) • 2 • 1`.

- **Exemple :**

  5! = 5 • 4 • 3 • 2 • 1 = 120

- **Algorithme :**

Exemple 3 :
factorielle

- Soit `n` un entier. Calculer la factorielle de `n`.

  Rappel : `n! = n • (n-1)•(n-2) • 2 • 1`.

- **Exemple :**

  5! = 5 • 4 • 3 • 2 • 1 = 120

- **Algorithme :**

```
Algorithm factorial(n)
  Input integer n
  Output factorial of n
Begin
  if n < 2 then
    return 1;
  else
    return n • factorial(n-1);
  end;
End.
```

Exemple 3 :

factorielle

- Soit **n** un entier. Calculer la factorielle de **n**.

  Rappel : $n! = n \cdot (n-1) \cdot (n-2) \cdot 2 \cdot 1.$

- **Exemple :**

  $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$

- **Algorithme :**

```
Algorithm factorial(n)
  Input integer n
  Output factorial of n
Begin
  if n < 2 then
    return 1;
  else
    return n • factorial(n-1);
  end;
End.
```

Complexité :

Pour déterminer la complexité de cet algorithme, nous allons **déterminer le nombre de fois où il fait appel à lui-même**. Une fois ce nombre connu, il est alors facile de déterminer sa complexité.

Exemple 3 :
factorielle

- Soit **n** un entier. Calculer la factorielle de **n**.

  Rappel : $n! = n \bullet (n-1) \bullet (n-2) \bullet 2 \bullet 1$.

- **Exemple :**

  $5! = 5 \bullet 4 \bullet 3 \bullet 2 \bullet 1 = 120$

- **Algorithme :**

```
Algorithm factorial(n)
  Input integer n
  Output factorial of n
Begin
  if n < 2 then
    return 1;
  else
    return n • factorial(n-1);
  end;
End.
```

Complexité :

Pour déterminer la complexité de cet algorithme, nous allons **déterminer le nombre de fois où il fait appel à lui-même**. Une fois ce nombre connu, il est alors facile de déterminer sa complexité.

1. Dans le corps de cette fonction, il y a:
   - un test,
   - un appel à elle même,
   - une soustraction,
   - une multiplication,
   - une opération de sortie.

   En tout, pour chaque exécution de cette fonction, il y a 5 opérations élémentaires qui sont exécutées pour $n>2$.

Exemple 3 :
factorielle

- Soit `n` un entier. Calculer la factorielle de `n`.

  Rappel : `n! = n • (n-1)•(n-2) • 2 • 1`.

- **Exemple :**

  5! = 5 • 4 • 3 • 2 • 1 = 120

- **Algorithme :**

```
Algorithm factorial(n)
  Input integer n
  Output factorial of n
Begin
  if n < 2 then
    return 1;
  else
    return n • factorial(n-1);
  end;
End.
```

2. Soit `f(n)` la complexité de l'algorithme `factorial(n)`. Alors :
   - `f(n-1)=factorial(n-1)`,
   - `f(n)=f(n-1)+5` si `n>2`

   La dernière équation est connue sous le nom d'**équation de récurrence**.

Exemple 3 :
factorielle

- Soit `n` un entier. Calculer la factorielle de `n`.

  Rappel : `n! = n • (n-1)•(n-2) • 2 • 1`.

- **Exemple :**

  5! = 5 • 4 • 3 • 2 • 1 = 120

- **Algorithme :**

```
Algorithm factorial(n)
  Input integer n
  Output factorial of n
Begin
  if n < 2 then
    return 1;
  else
    return n • factorial(n-1);
  end;
End.
```

2. Soit `f(n)` la complexité de l'algorithme `factorial(n)`. Alors :
   - `f(n-1)=factorial(n-1)`,
   - `f(n)=f(n-1)+5` si `n>2`

   La dernière équation est connue sous le nom d'**équation de récurrence**.

Exemple 3 :
factorielle

- Soit `n` un entier. Calculer la factorielle de `n`.

  Rappel : `n! = n • (n-1)•(n-2) • 2 • 1`.

- **Exemple :**

  5! = 5 • 4 • 3 • 2 • 1 = 120

- **Algorithme :**

```
Algorithm factorial(n)
  Input integer n
  Output factorial of n
Begin
  if n < 2 then
    return 1;
  else
    return n • factorial(n-1);
  end;
End.
```

2. Soit `f(n)` la complexité de l'algorithme `factorial(n)`. Alors :
   - `f(n-1)=factorial(n-1)`,
   - `f(n)=f(n-1)+5` si `n>2`

   La dernière équation est connue sous le nom d'**équation de récurrence**.

   Pour connaître `f(n)`, il y a lieu de passer à la résolution d'équation de récurrence comme suit :

```
f(n)   = f(n-1) + 5
f(n-1) = f(n-2) + 5
f(n-2) = f(n-3) + 5
…
f(2)   = f(1) + 5
```

Exemple 3 :

factorielle

- Soit `n` un entier. Calculer la factorielle de `n`.

  Rappel : `n! = n • (n-1)•(n-2) • 2 • 1`.

- **Exemple :**

  5! = 5 • 4 • 3 • 2 • 1 = 120

- **Algorithme :**

```
Algorithm factorial(n)
  Input integer n
  Output factorial of n
Begin
  if n < 2 then
    return 1;
  else
    return n • factorial(n-1);
  end;
End.
```

2. Soit `f(n)` la complexité de l'algorithme `factorial(n)`. Alors :
   - `f(n-1)=factorial(n-1)`,
   - `f(n)=f(n-1)+5` si `n>2`

   La dernière équation est connue sous le nom d'**équation de récurrence**.

   Pour connaître `f(n)`, il y a lieu de passer à la résolution d'équation de récurrence comme suit :

```
f(n)   = f(n-1) + 5
f(n-1) = f(n-2) + 5
f(n-2) = f(n-3) + 5
…
f(2)   = f(1) + 5
```

   En additionnant membre à membre, on arrive à : `f(n)=f(1)+5(n-1)=2+5(n-1)=5n-3`.

Exemple 3 :

factorielle

- Soit `n` un entier. Calculer la factorielle de `n`.

  Rappel : `n! = n • (n-1)•(n-2) • 2 • 1`.

- **Exemple :**

  5! = 5 • 4 • 3 • 2 • 1 = 120

- **Algorithme :**

```
Algorithm factorial(n)
  Input integer n
  Output factorial of n
Begin
  if n < 2 then
    return 1;
  else
    return n • factorial(n-1);
  end;
End.
```

2. Soit `f(n)` la complexité de l'algorithme `factorial(n)`. Alors :
   - `f(n-1)=factorial(n-1)`,
   - `f(n)=f(n-1)+5` si `n>2`

   La dernière équation est connue sous le nom d'**équation de récurrence**.

   Pour connaître `f(n)`, il y a lieu de passer à la résolution d'équation de récurrence comme suit :

```
f(n)   = f(n-1) + 5
f(n-1) = f(n-2) + 5
f(n-2) = f(n-3) + 5
…
f(2)   = f(1) + 5
```

   En additionnant membre à membre, on arrive à : `f(n)=f(1)+5(n-1)=2+5(n-1)=5n-3`.

   C'est-à-dire que la complexité de l'algorithme est `O(n)`.

Plan

- Algorithmique
- Pseudocode
- Différentes techniques de preuve
- Preuve d’un algorithme
- Complexité d’un algorithme
- Analyse asymptotique d’un algorithme
- Résumé

Résumé

- **Une algorithme doit être :**
  - correct (conditions de terminaison et validité sont vérifiées) et
  - exécuté en temps raisonnable.
- **La complexité d’un algorithme :**
  - est relative à une ou des opérations fondamentales de l’algorithme,
  - dépend de la taille des données et de leur configuration.
- **La performance des machines ne change pas l'efficacité d’un algorithme.**