



Information, Calcul et Communication Module 1 : Calcul

Leçon I.3 : Complexité des Algorithmes

J.-C. Chappelier, J. Sam (slides)

Objectifs de la leçon

Dans la leçon précédente, nous avons vu les composantes d'un algorithme, en particulier les instructions de contrôle, et comment celles-ci étaient mise en œuvre dans des algorithmes simples.

Nous cherchons maintenant à caractériser les algorithmes du point de vue de leur **coût calcul**, c'est-à-dire l'ordre de grandeur du *nombre d'opérations* requises pour obtenir le résultat.

☞ C'est dans ce sens que nous parlons de l'ordre de **complexité** d'un algorithme

Les objectifs de cette leçon sont de :

- Formaliser l'évaluation du coût calcul
- Identifier les principales classes d'ordre de complexité
- Pour cela, exploiter deux familles d'algorithmes: la **recherche** et le **tri**
- Aborder la stratégie de **conception top-down** sur l'algorithme de tri

Plan

- Evaluation du coût calcul sur deux variantes d'un algorithme de recherche linéaire
- Evaluation du coût calcul d'une recherche dans un ensemble ordonné: la dichotomie
- Ordre de complexité d'un algorithme (notation $O(\dots)$)
- Ordre de complexité d'un problème: le tri
- Stratégie de conception top-down du tri par insertion

Recherche

Exemple : recherche d'un élément x dans un ensemble E

AVANT TOUT : Spécification claire du problème :

- E peut-il être vide ?
 E varie-t-il pendant la recherche ?
 E est-il ordonné ?

Recherche

Exemple : recherche d'un élément x dans une liste E

Considérons par exemple les deux algorithmes suivants :

appartient1

entrée : x, E non vide

sortie : $x \in E ?$ (booléen)

$i \leftarrow 1$

Répéter

Si $x = E[i]$

sortir : Vrai

$i \leftarrow i + 1$

$t \leftarrow \text{taille}(E)$

Tant que $i \leq t$

sortir : Faux

appartient2

entrée : x, E non vide

sortie : $x \in E ?$ (booléen)

$t \leftarrow \text{taille}(E)$

Pour i de 1 à t

Si $x = E[i]$

sortir : Vrai

sortir : Faux

Recherche

Exemple : recherche d'un élément x dans une liste E

Considérons par exemple les deux algorithmes suivants :

appartient1
entrée : x, E non vide sortie : $x \in E ?$
$i \leftarrow 1$ Répter Si $x = E[i]$ sortir : Vrai $i \leftarrow i + 1$ $t \leftarrow$ taille (E) Tant que $i \leq t$ sortir : Faux

appartient2
entrée : x, E non vide sortie : $x \in E ?$
$t \leftarrow$ taille (E) Pour i de 1 à t Si $x = E[i]$ sortir : Vrai sortir : Faux

autre algorithme (qui calcule la taille de E)

Complexité d'un algorithme

1) ces algorithmes sont ils **corrects** ?

- se terminent ils pour tous les cas ?
- donnent ils ce que l'on veut ?

2) lequel des deux est le **plus efficace** ?


- notion de **complexité** d'un algorithme

complexité (**temporelle** dans le **pire des cas**) : nombre d'instructions *élémentaires* nécessaires à un algorithme pour donner la réponse dans *le pire des cas*.

C'est une fonction de la *taille de l'entrée*

Complexité d'un algorithme : exemple

2^e question : lequel est le **plus efficace** ?

Notons n la taille de E et comptons combien d'instructions élémentaires chaque algorithme nécessite dans le pire des cas  $C1(n)$ et $C2(n)$

Pour l'algorithme **appartient1**(x, E) :

$i \leftarrow 1$

Répéter

Si $x = E[i]$

sortir : *Vrai*

$i \leftarrow i + 1$

$t \leftarrow \text{taille}(E)$

Tant que $i \leq t$

sortir : *Faux*

- | | | |
|---|--|--|
| 1 | affectation de la valeur 1 à la variable i | |
| 2 | accès au i -ème élément de E et
comparaison de cet élément avec x | |
| 3 | incrémentement de i (de 1) | |
| 4 | calcul de la taille de E | |
| 5 | vérification de la condition ($i \leq t$) et
retour en 2 | |

1 instruction


2 instructions

1 instruction

$T(n)$ instructions

1 instruction

Dans le *pire des cas*, les étapes 2 à 5 sont faites autant de fois qu'il y a d'éléments dans E , donc n fois.

 $C1(n) = 1 + n(T(n) + 4)$

Complexité d'un algorithme : exemple

Pour l'algorithme **appartient2**(x, E) :

$t \leftarrow \mathbf{taille}(E)$	1	calcul de la taille de E	$T(n)$ instructions
Pour i de 1 à t	2	affectation de la valeur 1 à i	1 instruction
Si $x = E[i]$	3	vérification de la condition ($i \leq t$)	1 instruction
sortir : <i>Vrai</i>	4	accès au i^{e} élément de E et comparaison de cet élément avec x	2 instructions
	5	incrémentement de i (de 1) et retour en 3	1 instruction
sortir : <i>Faux</i>			

Dans le *pire des cas*, les étapes 3 à 5 seront faites autant de fois qu'il y a d'éléments dans E , donc n fois.

👉 $C2(n) = T(n) + 1 + 4n$

Complexité d'un algorithme : exemple

Supposons (raisonnablement) que le calcul de la taille de E se fait en $T(n) = a + b \cdot n$ instructions (avec $b \geq 0$, mais éventuellement nul).

On aurait alors :

$$C1(n) = 1 + (a + 4)n + bn^2$$

$$C2(n) = 1 + a + (4 + b)n$$

Si $b > 0$ (i.e. non nul), alors l'algorithme 1 est donc **beaucoup plus lent** (pour de grands ensembles) !

Peut-on faire (nettement) mieux que l'algorithme 2 ?

👉 oui, si l'ensemble est **ordonné**

Plan

- Evaluation du coût calcul sur deux variantes d'un algorithme de recherche linéaire
- Evaluation du coût calcul d'une recherche dans un ensemble ordonné: la dichotomie
- Ordre de complexité d'un algorithme (notation $O(\dots)$)
- Ordre de complexité d'un problème: le tri
- Stratégie de conception top-down du tri par insertion

Dichotomie (algorithme récursif)

appartient_D

entrée : x, E **ordonné**

sortie : $x \in E ?$ (booléen)

Si E est vide

sortir : Faux

Si E est réduit à 1 seul élément e

Si $x = e$ Alors **Sortir** Vrai

Sinon **Sortir** Faux

découper E en deux parties E_1 et E_2

Si $x \leq \max(E_1)$

sortir : **appartient_D**(x, E_1)

Sinon

sortir : **appartient_D**(x, E_2)

la sortie est Vrai ou Faux (booléen)

moitié-moitié

☞ Cet algorithme s'appelle la **recherche par dichotomie**

Exemple de recherche par dichotomie dans un ensemble ordonné

$E =$ abaque
 abasourdi
 babouin
 baobab
 blanc
 bleu
 zoulou

$x =$ bleu

Si E est vide

sortir : Faux

Si E est réduit à 1 seul élément e

Si $x = e$ Alors **Sortir** Vrai

Sinon **Sortir** Faux

découper E en deux parties E_1 et E_2

Si $x \leq \max(E_1)$

sortir : **appartient_D**(x, E_1)

Sinon

sortir : **appartient_D**(x, E_2)

appartient_D(x, E) = ??

$E =$ abaque
 abasourdi
 babouin
 baobab
 blanc
 bleu
 zoulou

$x =$ bleu

E n'est pas vide ni réduit à un élément

Si E est vide

sortir : Faux

Si E est réduit à 1 seul élément e

Si $x = e$ Alors **Sortir** Vrai

Sinon **Sortir** Faux

découper E en deux parties E_1 et E_2

Si $x \leq \max(E_1)$

sortir : **appartient_D**(x, E_1)

Sinon

sortir : **appartient_D**(x, E_2)

appartient_D(x, E) = ??

$E_1 =$ abaque
 abasourdi
 babouin
 baobab

$x =$ bleu

$E_2 =$ blanc
 bleu
 zoulou

E n'est pas vide ni réduit à un élément

donc on découpe E en (par exemple) E_1 et E_2 comme ci-dessus

Si E est vide

sortir : Faux

Si E est réduit à 1 seul élément e

Si $x = e$ Alors **Sortir** Vrai

Sinon **Sortir** Faux

découper E en deux parties E_1 et E_2

Si $x \leq \max(E_1)$

sortir : **appartient_D**(x, E_1)

Sinon

sortir : **appartient_D**(x, E_2)

appartient_D(x, E) = ??

$E_1 =$ abaque
 abasourdi
 babouin
 baobab

$x =$ bleu

$E_2 =$ blanc
 bleu
 zoulou

est-ce que $x \leq \max(E_1)$, c.-à-d. est-ce que «bleu» vient avant «baobab» ?

👉 non

Si E est vide

sortir : Faux

Si E est réduit à 1 seul élément e

Si $x = e$ Alors **Sortir** Vrai

Sinon **Sortir** Faux

découper E en deux parties E_1 et E_2

Si $x \leq \max(E_1)$

sortir : $\text{appartient_D}(x, E_1)$

Sinon

sortir : $\text{appartient_D}(x, E_2)$

$$\text{appartient_D}(x, E) = \text{appartient_D}(x, E_2)$$

$E_1 =$ abaque
 abasourdi
 babouin
 baobab

$x =$ bleu

$E_2 =$ blanc
 bleu
 zoulou

est-ce que $x \leq \max(E_1)$, c.-à-d. est-ce que «bleu» vient avant «baobab» ?

☞ non

donc la solution est la même que $\text{appartient_D}(x, E_2)$

Si E est vide

sortir : Faux

Si E est réduit à 1 seul élément e

Si $x = e$ Alors **Sortir** Vrai

Sinon **Sortir** Faux

découper E en deux parties E_1 et E_2

Si $x \leq \max(E_1)$

sortir : **appartient_D**(x, E_1)

Sinon

sortir : **appartient_D**(x, E_2)

$$\text{appartient_D}(x, E) = \text{appartient_D}(x, E_2)$$

$E_1 =$ abaque
 abasourdi
 babouin
 baobab

$x =$ bleu

$E_{2.1} =$ blanc
 bleu

$E_{2.2} =$ zoulou

E_2 n'est pas vide ni réduit à un élément

donc on découpe E_2 en (par exemple) $E_{2.1} =$ blanc bleu et $E_{2.2} =$ zoulou

Si E est vide

sortir : Faux

Si E est réduit à 1 seul élément e

Si $x = e$ Alors **Sortir** Vrai

Sinon **Sortir** Faux

découper E en deux parties E_1 et E_2

Si $x \leq \max(E_1)$

sortir : $\text{appartient_D}(x, E_1)$

Sinon

sortir : $\text{appartient_D}(x, E_2)$

$$\text{appartient_D}(x, E) = \text{appartient_D}(x, E_{2.1})$$

$E_1 =$ abaque
 abasourdi
 babouin
 baobab

$x =$ bleu

$E_{2.1} =$ blanc
 bleu

$E_{2.2} =$ zoulou

est-ce que $x \leq \max(E_{2.1})$?

👉 oui

donc la solution est la même que $\text{appartient_D}(x, E_{2.1})$

Si E est vide

sortir : Faux

Si E est réduit à 1 seul élément e

Si $x = e$ Alors **Sortir** Vrai

Sinon **Sortir** Faux

découper E en deux parties E_1 et E_2

Si $x \leq \max(E_1)$

sortir : **appartient_D**(x, E_1)

Sinon

sortir : **appartient_D**(x, E_2)

$$\text{appartient_D}(x, E) = \text{appartient_D}(x, E_{2.1})$$

$E_1 =$ abaque
 abasourdi
 babouin
 baobab

$x =$ bleu

$E_{2.1} =$ blanc
 bleu

$E_{2.2} =$ zoulou

$E_{2.1}$ n'est pas vide ni réduit à un élément,

Si E est vide

sortir : Faux

Si E est réduit à 1 seul élément e

Si $x = e$ Alors **Sortir** Vrai

Sinon **Sortir** Faux

découper E en deux parties E_1 et E_2

Si $x \leq \max(E_1)$

sortir : **appartient_D**(x, E_1)

Sinon

sortir : **appartient_D**(x, E_2)

$$\text{appartient_D}(x, E) = \text{appartient_D}(x, E_{2.1})$$

$E_1 =$ abaque
 abasourdi
 babouin
 baobab

$E_{2.1.1} =$ blanc

$E_{2.1.2} =$ bleu

$E_{2.2} =$ zoulou

$x =$ bleu

$E_{2.1}$ n'est pas vide ni réduit à un élément,
 donc on découpe $E_{2.1}$ en $E_{2.1.1} =$ blanc et $E_{2.1.2} =$ bleu

Si E est vide

sortir : Faux

Si E est réduit à 1 seul élément e

Si $x = e$ Alors **Sortir** Vrai

Sinon **Sortir** Faux

découper E en deux parties E_1 et E_2

Si $x \leq \max(E_1)$

sortir : **appartient_D**(x, E_1)

Sinon

sortir : **appartient_D**(x, E_2)

$$\text{appartient_D}(x, E) = \text{appartient_D}(x, E_{2.1})$$

$E_1 =$ abaque
 abasourdi
 babouin
 baobab

$E_{2.1.1} =$ blanc

$E_{2.1.2} =$ bleu

$E_{2.2} =$ zoulou

$x =$ bleu

est-ce que $x \leq \max(E_{2.1.1})$, c.-à-d. est-ce que «bleu» vient avant «blanc» ?

Si E est vide

sortir : Faux

Si E est réduit à 1 seul élément e

Si $x = e$ Alors **Sortir** Vrai

Sinon **Sortir** Faux

découper E en deux parties E_1 et E_2

Si $x \leq \max(E_1)$

sortir : **appartient_D**(x, E_1)

Sinon

sortir : **appartient_D**(x, E_2)

$$\text{appartient_D}(x, E) = \text{appartient_D}(x, E_{2.1.2})$$

$E_1 =$ abaque
 abasourdi
 babouin
 baobab

$E_{2.1.1} =$ blanc

$E_{2.1.2} =$ bleu

$E_{2.2} =$ zoulou

$x =$ bleu

est-ce que $x \leq \max(E_{2.1.1})$, c.-à-d. est-ce que «bleu» vient avant «blanc» ?

☞ non

donc la solution est la même que **appartient_D**($x, E_{2.1.2}$)

Si E est vide

sortir : Faux

Si E est réduit à 1 seul élément e

Si $x = e$ Alors **Sortir** Vrai

Sinon **Sortir** Faux

découper E en deux parties E_1 et E_2

Si $x \leq \max(E_1)$

sortir : **appartient_D**(x, E_1)

Sinon

sortir : **appartient_D**(x, E_2)

$$\text{appartient_D}(x, E) = \text{appartient_D}(x, E_{2.1.2})$$

$E_1 =$ abaque
 abasourdi
 babouin
 baobab


$E_{2.1.1} =$ blanc

$E_{2.1.2} =$ bleu

$E_{2.2} =$ zoulou

$x =$ bleu

$E_{2.1.2}$ est réduit à un élément

Est-ce x ?  oui

Si E est vide

sortir : Faux

Si E est réduit à 1 seul élément e

Si $x = e$ Alors **Sortir** Vrai

Sinon **Sortir** Faux

découper E en deux parties E_1 et E_2

Si $x \leq \max(E_1)$

sortir : **appartient_D**(x, E_1)

Sinon

sortir : **appartient_D**(x, E_2)

appartient_D(x, E) = **oui** !

$E_1 =$ abaque
 abasourdi
 babouin
 baobab


$E_{2.1.1} =$ blanc

$E_{2.1.2} =$ bleu

$E_{2.2} =$ zoulou

$x =$ bleu

$E_{2.1.2}$ est réduit à un élément

Est-ce x ?  oui

donc la solution est « **oui** »

Complexité ?

Quel est le nombre d'opérations nécessaires pour une recherche par dichotomie dans le pire des cas ?

Si l'élément recherché est au « milieu » du « milieu » du ... « milieu » du « milieu » de l'ensemble, il faudra répéter la boucle de découpage en **deux** autant de fois qu'on découpe l'ensemble.

On va donc boucler autant de fois qu'on peut diviser n par 2 et obtenir un résultat avec **une partie entière non nulle**.

Combien de fois qu'on peut diviser n par 2 ?

👉 $\log_2 n$

Coût calcul très avantageux par rapport à la recherche linéaire car **$\log_2 (n) \ll n$**

Rappels :

$$2^y = n \Rightarrow y = \log_2(n)$$

$$\log_2(x) = \frac{\ln(x)}{\ln(2)}$$

Plan

- Evaluation du coût calcul sur deux variantes d'un algorithme de recherche linéaire
- Evaluation du coût calcul d'une recherche dans un ensemble ordonné: la dichotomie
- **Ordre de complexité d'un algorithme (notation $O(\dots)$)**
- Ordre de complexité d'un problème: le tri
- Stratégie de conception top-down du tri par insertion

Complexité : notation $O(\dots)$

Pour comparer des algorithmes, ce qui nous intéresse c'est de savoir comment leur **complexité évolue en fonction de la taille des données** en entrée.

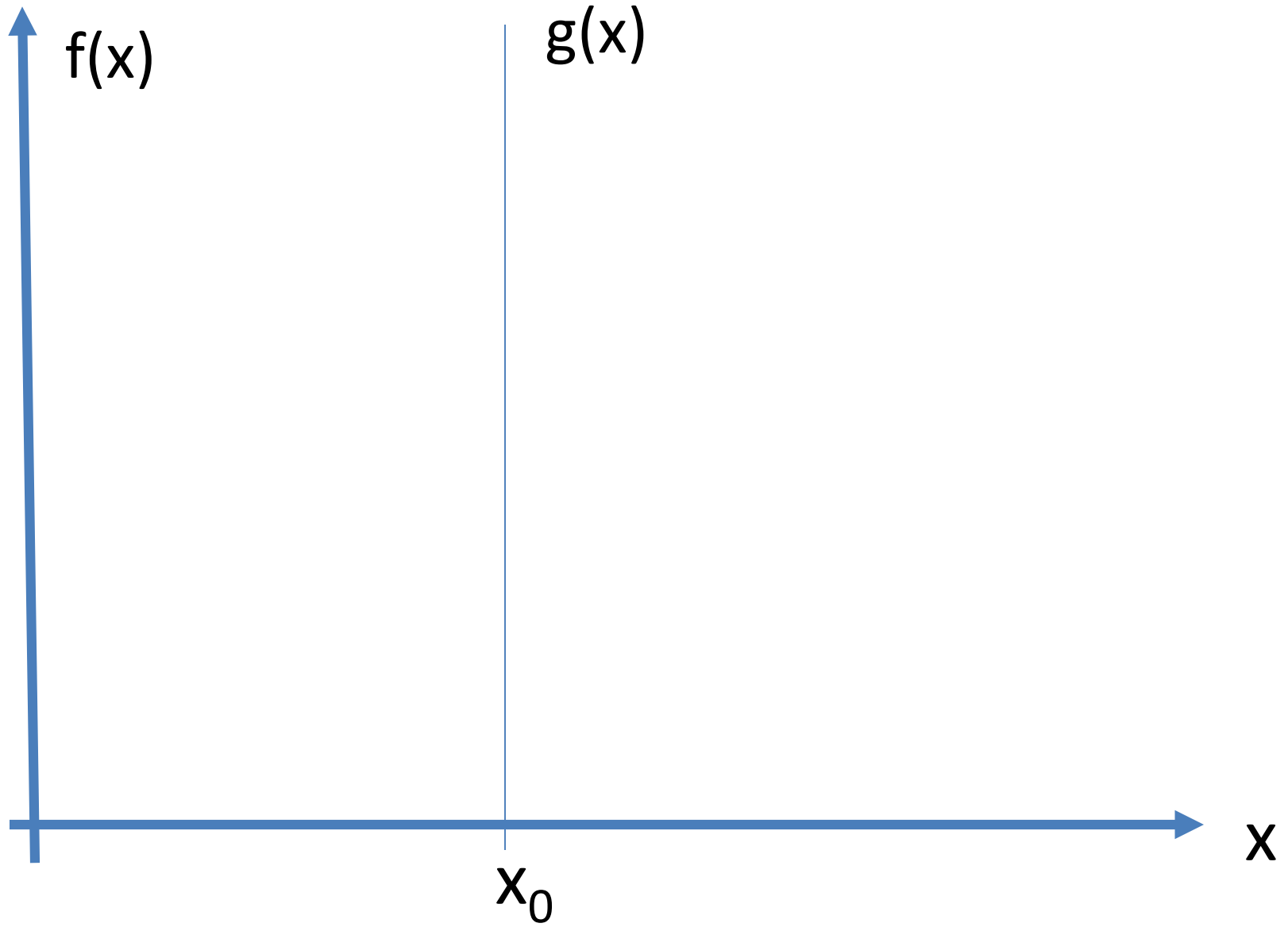
Pour cela, on effectue des comparaisons sur les **ordres de grandeur asymptotiques** (quand la taille des données en entrée tend vers l'infini) Ces ordres de grandeur sont généralement notés en utilisant la notation de Landau $O(\dots)$

Pour deux fonctions f et g de \mathbb{R} dans \mathbb{R} , on écrit :

$$f \in O(g)$$

si et seulement si

$$\exists c > 0 \exists x_0 \forall x > x_0 |f(x)| \leq c \cdot |g(x)|$$



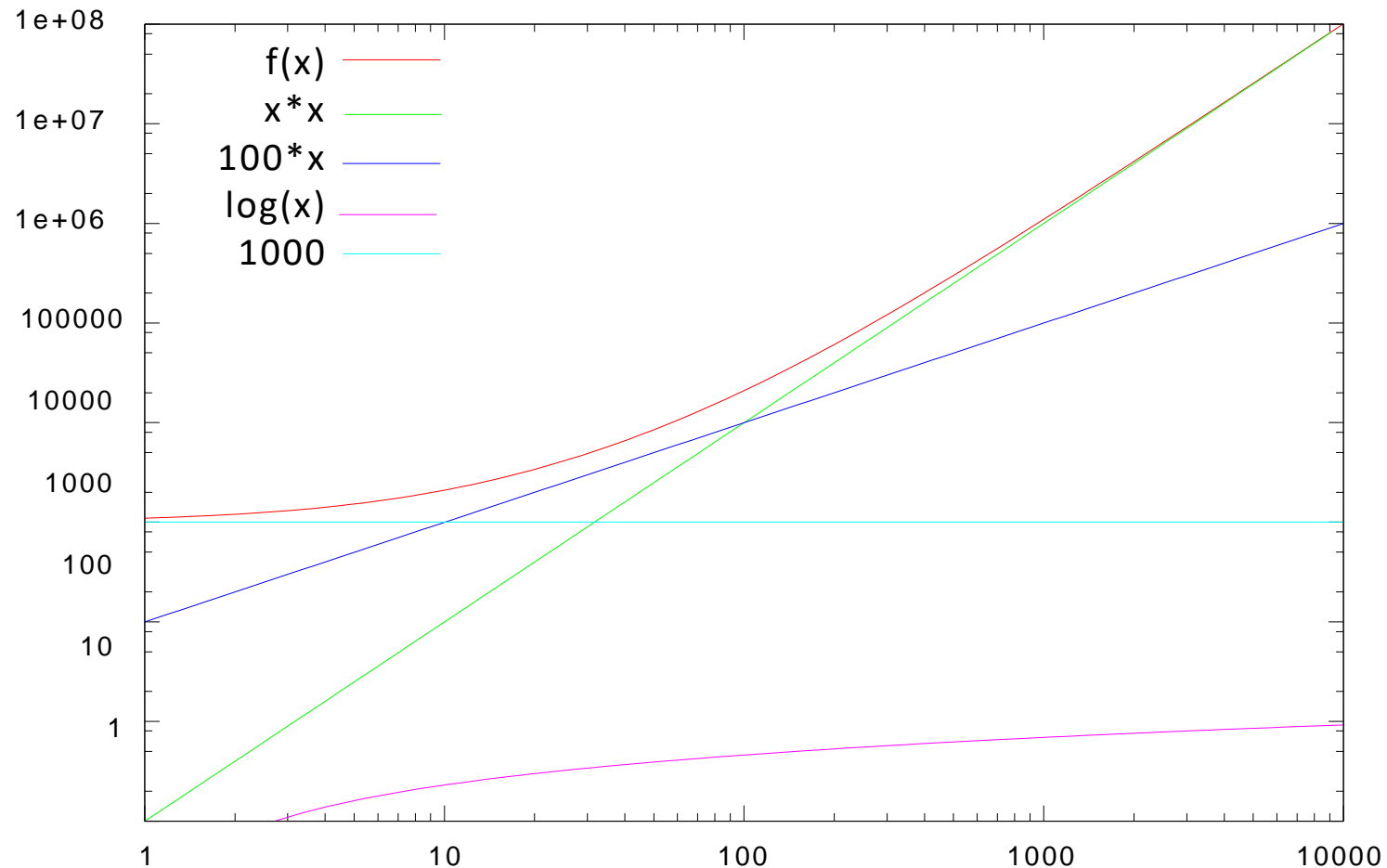
Notation « grand O » : exemple

$$f(n) = n^2 + 100n + \log n + 1000$$

n	$f(n)$	n^2		$100n$		$\log n$		1000	
		valeur	%	valeur	%	valeur	%	valeur	%
1	1'101	1	0.1	100	9.1	0	0	1000	90.82
10	2'101	100	4.8	1'000	47.6	1	0.0	1000	47.6
100	21'002	10'000	47.6	10'000	47.6	2	0.0	1000	4.8
10^3	1'101'003	10^6	90.8	10^5	9.1	3	0.0	1000	0.1
10^4	101'001'004	10^8	99.0	10^6	1.0	4	0.0	1000	0.0
...									

Notation « grand O » : exemple (suite)

$$f(n) = n^2 + 100n + \log n + 1000$$



Comparaison d'algorithme

En pratique pour mesurer la complexité d'un algorithme, on utilise évidemment **la plus petite des fonctions majorantes possibles $g()$**

Exemples :

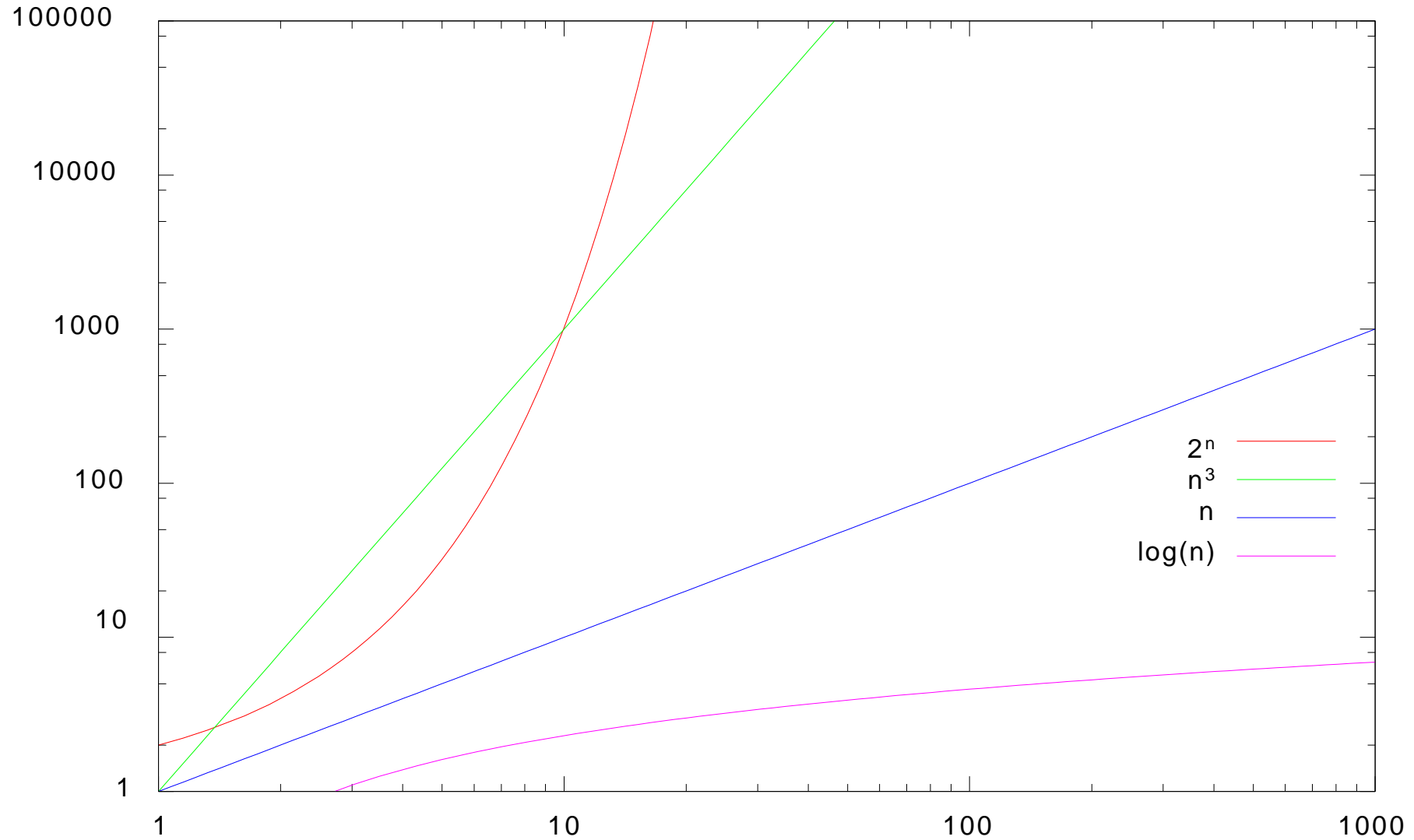
n est $O(n^2)$ mais n est aussi $O(n)$

12 est $O(n^2)$, $O(n)$, mais surtout $O(1)$

Différentes classes de complexité permettent alors de caractériser les algorithmes (n représentant la taille d'entrée) :

- .. complexité **constante $O(1)$** : cas idéal ! aucune influence sur l'algorithme
- .. complexité **logarithmique $O(\log n)$**
- .. complexité **linéaire $O(n)$**
- .. complexité **quasi-linéaire $O(n \log(n))$**
- .. complexité **polynomiale $O(n^2)$, ... $O(n^k)$**
- .. complexité **exponentielle $O(2^n)$** , le pire des cas

Comparaison

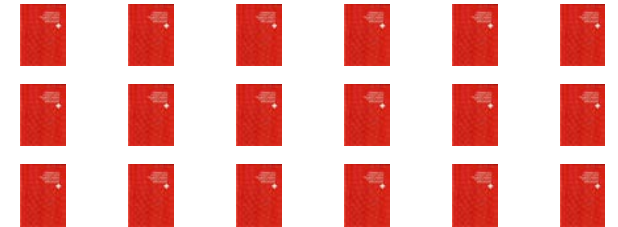


Comparaison

Si la police devrait contrôler les papiers de tous les Lausannois(es),
il y aurait une file continue de 175 km
à peu près une file linéaire continue jusqu'à Zürich !



Si elle ne doit en contrôler que le **log** : que 18 passeports !!
(log en base 2)



Algorithmes de recherche dans une liste

Pour résumer sur les algorithmes de recherche:

- **si la liste n'est pas ordonnée** : recherche exhaustive terme à terme, complexité linéaire ($O(n)$, où n est la taille de la liste)
- **si la liste est ordonnée** : recherche par dichotomie, complexité logarithmique ($O(\log n)$)

👉 importance de la *modélisation des données*

- Ici si la liste est triée : solution moins complexe en temps
- ***mais quelle est la complexité du tri ?...***
- Notez cependant que le tri n'est fait qu'une seule fois avant toutes les recherches !

Plan

- Evaluation du coût calcul sur deux variantes d'un algorithme de recherche linéaire
- Evaluation du coût calcul d'une recherche dans un ensemble ordonné: la dichotomie
- Ordre de complexité d'un algorithme (notation $O(\dots)$)
- **Ordre de complexité d'un problème: le tri**
- Stratégie de conception top-down du tri par insertion

Les tris

Le problème du tri est un problème intéressant en tant que tel et un bon **exemple de problème** pour lequel il existe de **nombreux algorithmes**.

Spécification du problème :

On considère une structure de données contenant des éléments que l'on peut **comparer** (entre eux : *relation d'ordre* totale sur l'ensemble des éléments)

On dira qu'un ensemble de données est **trié** si ses éléments sont disposés par **ordre croissant** lorsque l'on itère sur la structure de donnée.

Les tris

Par exemple : on cherche à trier un tableau d'entiers.



Remarques :

- un tri ne supprime pas les doublons
- quelque soit l'algorithme de tri, un ensemble de données vide ou réduit à un seul élément est déjà trié !...

On parle de **tri interne** (ou « sur/en place », par opposition à **tri externe**) lorsque l'on peut effectuer le tri en mémoire centrale, sans utiliser de support extérieur (fichier).

Algorithmes de tri

Il existe un grand nombre d'algorithmes de tri :

- récursif
- par insertion
- par sélection
- tri shaker
- tri de Shell
- tri tournois
- tri fusion
- tri par tas
- quick sort (« tri rapide »)
- ...

Comparaison des méthodes de tri

Soit n le nombre de données à trier.

	Complexité	
	moyenne	pire cas
par sélection	$O(n^2)$	$O(n^2)$
par insertion	$O(n^2)$	$O(n^2)$
de Shell	–	$O(n^{1.5})$
quick sort	$O(n \log n)$	$O(n^2)$
par tas	$O(n \log n)$	$O(n \log n)$

Mais en **pratique** : à partir de quelle taille les méthodes simples deviennent-elles vraiment plus mauvaises que les méthodes sophistiquées (quick sort ou tri par tas) ?

Conclusions sur les tris : comparaison (2)

En pratique ?

Cela dépend de nombreux facteurs, mais en général on peut dire que **pour moins d'une centaine** d'éléments les tris sophistiqués n'en valent pas la peine.

Par ailleurs, expérimentalement le **quick sort** est 2 à 3 fois plus rapide que le tri par tas

Dans le cas de listes presque triées, les tris par insertion sont efficaces

Le **tri bulles**, très simple à écrire, est le moins bon des tris : **à proscrire** (sauf à des fins pédagogiques)

<http://www.sorting-algorithms.com/>

Un premier exemple : le tri par insertion

Le principe du **tri par insertion** est extrêmement simple :

Un **élément mal placé** dans le tableau va systématiquement être inséré à sa « **bonne place** » dans le tableau.

tri insertion

entrée : *un tableau (d'objets que l'on peut comparer)*

Sortie : *le tableau trié*

Tant que il y a un **élément mal placé**
on cherche sa **bonne place**
on **déplace** l'élément à sa **bonne place**

« **élément mal placé** » = tout élément du tableau
strictement plus petit que son prédécesseur.

Exemple de déroulement du tri par insertion

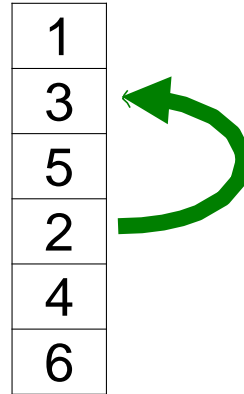
1
3
5
2
4
6

Exemple de déroulement du tri par insertion

1
3
5
2
4
6

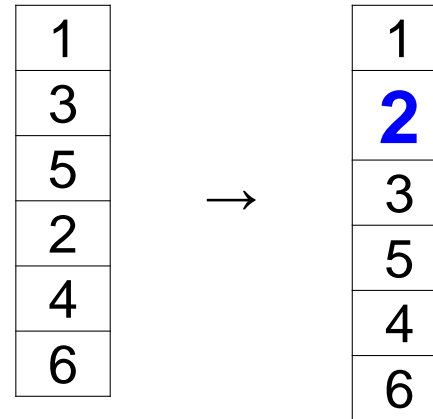
Tant que il y a un élément mal placé
on cherche sa bonne place
on déplace l'élément à sa bonne place

Exemple de déroulement du tri par insertion



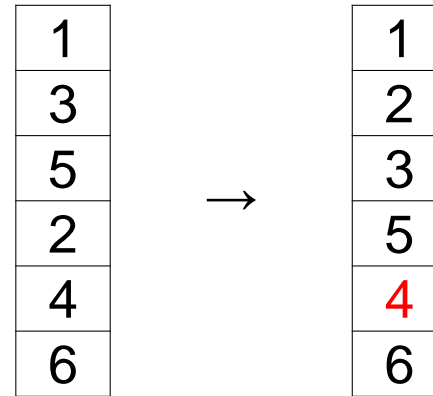
Tant que il y a un élément mal placé
on cherche sa bonne place
on déplace l'élément à sa bonne place

Exemple de déroulement du tri par insertion



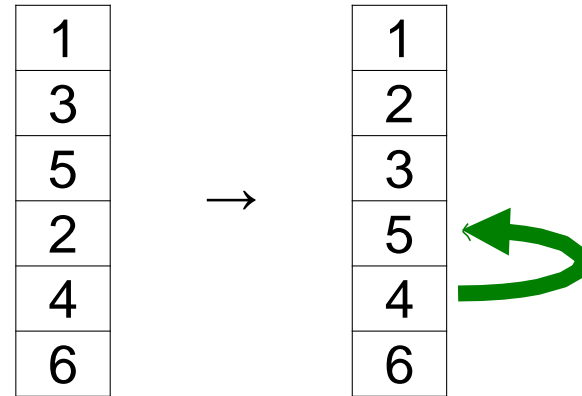
Tant que il y a un élément mal placé
on cherche sa bonne place
on déplace l'élément à sa bonne place

Exemple de déroulement du tri par insertion



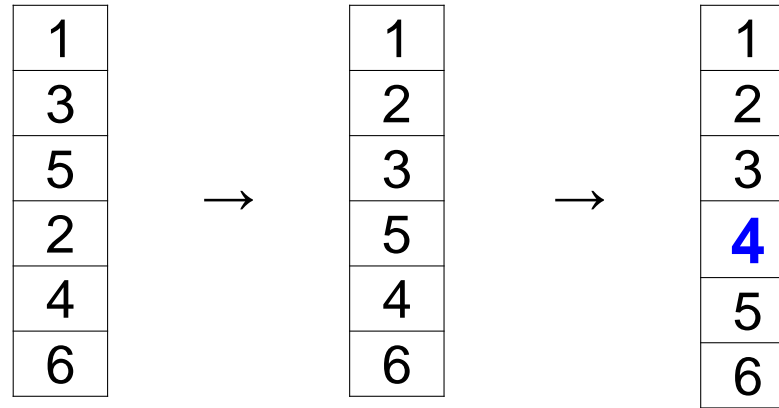
Tant que il y a un élément mal placé
on cherche sa bonne place
on déplace l'élément à sa bonne place

Exemple de déroulement du tri par insertion



Tant que il y a un élément mal placé
on cherche sa bonne place
on déplace l'élément à sa bonne place

Exemple de déroulement du tri par insertion



Tant que il y a un élément mal placé
on cherche sa bonne place
on déplace l'élément à sa bonne place

Plan

- Evaluation du coût calcul sur deux variantes d'un algorithme de recherche linéaire
- Evaluation du coût calcul d'une recherche dans un ensemble ordonné: la dichotomie
- Ordre de complexité d'un algorithme (notation $O(\dots)$)
- Ordre de complexité d'un problème: le tri
- **Stratégie de conception top-down du tri par insertion**

Conception d'algorithmes

Comment **concevoir** un algorithme permettant de résoudre un problème donné ?

Il n'y a malheureusement pas de méthode miracle ni de recette toute faite pour construire des solutions algorithmiques à un problème donné.

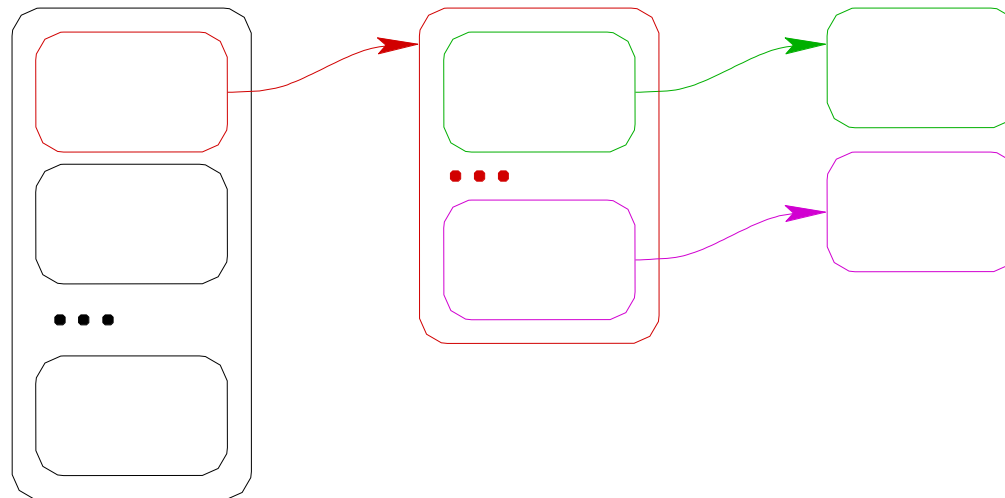
Il existe cependant plusieurs **méthodes de résolution**, c'est-à-dire des *schémas d'élaboration de solutions*.

Plusieurs de ces méthodes suivent ce que l'on appelle **une approche descendante** (« *top-down* », procède par *analyse*), par opposition à ascendante (« *bottom-up* », procède par *synthèse*).

Approche descendante

Résoudre un problème par une **approche descendante** consiste à **décomposer** le problème général en **sous-problèmes** plus spécifiques, lesquels seront chacun décomposés en problèmes encore plus spécifiques, etc. (raffinements successifs)

Une telle analyse du problème se fait à l'aide de **blocs imbriqués** correspondant chacun à des résolutions de plus en **plus spécifiques**, décrites par des algorithmes de plus en plus spécialisés.



Exemple

Algorithme de tri par insertion

On découpe le problème en sous-problèmes :

exemple:

tri insertion
entrée : <i>un tableau (d'objets que l'on peut comparer)</i> sortie : <i>le tableau trié</i>
Tant que il y a un élément mal placé on cherche sa bonne place on déplace l'élément à sa bonne place

1
3
5
2
4
6

Chaque **sous-problème** étant ensuite spécifié plus clairement puis résolu.

Tri par insertion : résolution détaillée

Le sous-problème *rechercher un élément mal placé*

entrée : un tableau **tab** (de taille N)

sortie : position du 1^{er} élément mal rangé dans l'ordre croissant
car *strictement plus petit que son prédécesseur*

On parcourt le tableau à partir du 2^{ième} élément
car c'est le premier à avoir un prédécesseur :

Pour **pos** de 2 à N

Si **tab**[**pos**] < **tab**[**pos**-1]

sortir: **pos**

sortir: **1**

Exemple:

tab	pos :
1	
3	2
5	3
2	4
4	5
6	6

s'il n'y a pas d'élément mal placé on retourne la position **1** car, dans cette boucle **Pour**, l'élément de position **1** ne peut pas être mal placé.

Tri par insertion : résolution détaillée (2)

Le sous-problème *trouver la bonne place*

entrée : un tableau **tab** et l'entier **pos**, position d'un élément mal placé

sortie : la bonne position **pos_ok** de l'élément mal placé.

posons **Valeur_mal_placee** \leftarrow **tab**[**pos**]


On recherche sa « bonne position » **pos_ok**

en reculant dans le tableau de (**pos -1**) jusqu'au second élément

pos_ok \leftarrow **pos -1**

Tant que (**pos_ok** \geq 2 et **Valeur_mal_placee** $<$ **tab**[**pos_ok -1**])
pos_ok \leftarrow **pos_ok -1**

Exemple:

tab	<i>position</i>	
1	1	
3	2	
5	3	 pos_ok
2	4	pos
4	5	
6	6	

Remarque:

la condition pour continuer la boucle comporte deux termes qui doivent tous les deux être VRAI.

On quitte la boucle dès que l'un des deux est FAUX

Tri par insertion : résolution détaillée (3)

Le sous-problème *déplacer un élément*

entrée : un tableau **tab**, une position de départ **pos** et une position finale **pos_ok**

On doit déplacer l'élément de la position **pos** dans **tab** à la position **pos_ok**.

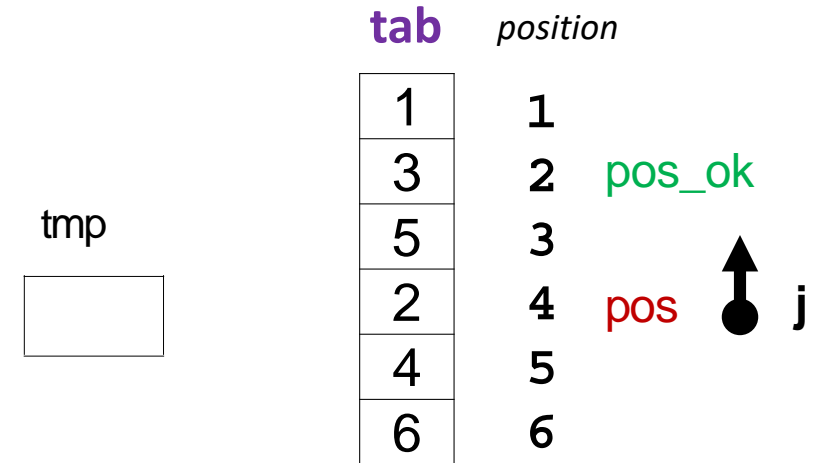
On peut effectuer cette opération par **décalages successifs** à l'aide d'une boucle **Pour**:

$$\text{tmp} \leftarrow \text{tab}[\text{pos}]$$

Pour j de **pos** à (**pos_ok** + 1)
 $\text{tab}[j] \leftarrow \text{tab}[j-1]$

$$\text{tab}[\text{pos_ok}] \leftarrow \text{tmp}$$

exemple:

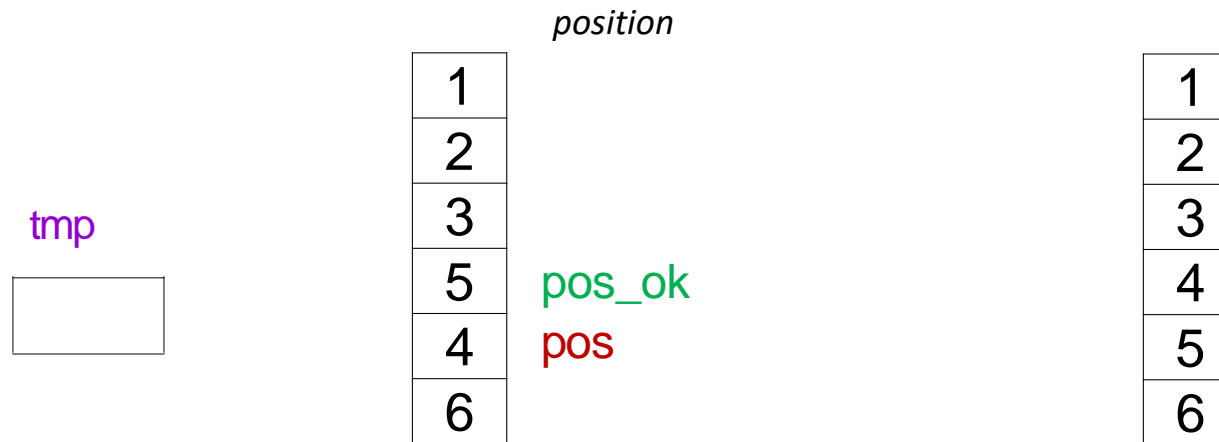


On utilise une variable tmp pour ne pas perdre la valeur **tab[pos]**.

Tri par insertion : résolution détaillée (4)

l'algorithme se poursuit tant qu'un élément mal placé est trouvé dans le tableau

exemple précédent (traitement de l'élément suivant mal placé):



Améliorations

1. Pour *rechercher le prochain élément mal placé*, ce n'est pas la peine de recommencer du début (position 2) à chaque fois. On peut continuer juste après *la dernière position mal placée*.
2. On pourrait *trouver la bonne place* et *déplacer l'élément* à cette place *en même temps* (i.e. en *une seule* itération)

Exemple précédent à faire soi-même:

Pour pos de 2 à N (= taille du tableau)

$tmp \leftarrow tab[pos]$

$j \leftarrow pos$

Tant que $j \geq 2$ et $tmp < tab[j-1]$

$tab[j] \leftarrow tab[j-1]$

$j \leftarrow j-1$

$tab[j] \leftarrow tmp$

pos

	1	1	1
2	3	2	2
3	5	3	3
4	2	5	4
5	4	4	5
6	6	6	6

tmp

tmp

exemple sur le pire des cas: liste à l'envers

Pour **pos** de 2 à N (= taille du tableau)

tmp ← tab[**pos**]

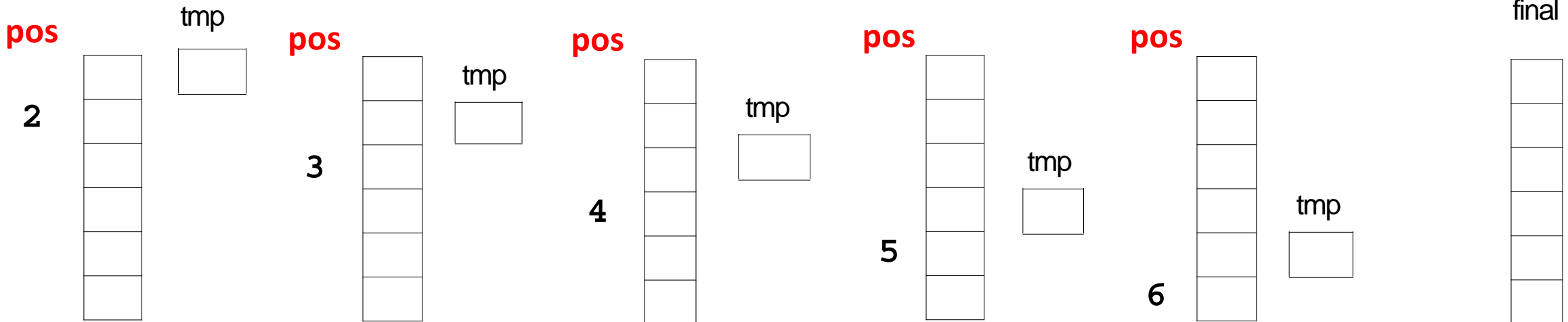
j ← **pos**

Tant que j ≥ 2 **et** tmp < tab[j-1]

tab[j] ← tab[j-1]

j ← j-1

tab[j] ← tmp



Ce que j'ai appris aujourd'hui

Dans cette leçon, vous avez

- appris à comparer l'efficacité de deux algorithmes : **complexité**
- vu deux familles de problèmes typiques en Informatique (recherche, tris)
- vu combien *algorithme* et représentations des *données* sont liés :
 - recherche linéaire dans une liste non ordonnée
 - versus recherche dichotomique dans une liste ordonnée

Vous pouvez maintenant :

- décrire certains problèmes de base de l'Informatique (recherche, tris)
- construire des algorithmes simples pour des problèmes simples typiques
- calculer la complexité d'algorithmes simples

La suite

La prochaine leçon présentera :

- la stratégie *Divide & Conquer*
- Autres illustrations avec l'approche récursive : force et faiblesse
- La programmation dynamique
 - Exemple: Algorithme(s) de plus court chemin