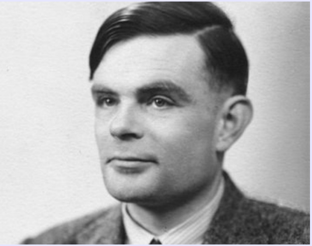
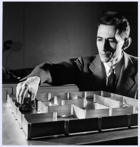
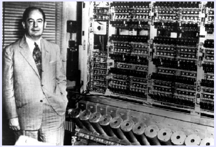




Information, Calcul et Communication

Module 1 : Calcul

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Leçon I.1 : Calcul et Algorithmes I

J.-C. Chappelier & J. Sam

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Objectifs de la leçon

Dans la leçon précédente, nous avons vu combien l'informatique est devenue centrale à notre civilisation

- accéleration(s)
- ominprésente dans tous les domaines de l'économie
- 4^{e} pilier de notre culture

Une question est maintenant de savoir comment *traiter*/*manipuler* toutes ces information(s).

☞ C'est tout l'objet du **calcul** informatique.

Les objectifs de cette leçon sont de :

- Formaliser ces calculs : notion d'algorithme
- Présenter les « ingrédients de base » des algorithmes
- Introduire quelques principales familles d'algorithmes : recherche, tri, plus court chemin
- Calculer (et exprimer) la complexité d'un algorithme

Qu'est-ce que l'Informatique ?

« *Science du* **traitement automatique de l'information** *(tri, transmission, utilisation), mis en œuvre sur des* **ordinateurs**. »

($\simeq$ Petit Robert)

Objectifs : permettre, à l'aide d'**ordinateurs**,

- la *simulation* de *modèles* et l'*optimisation* de solutions
- l'*automatisation* d'un certain nombre de tâches
- l'*organisation*, le *transfert* et la *recherche* d'information

☞ Qu'est-ce qu'un ordinateur ?

Qu'est-ce qu'un ordinateur ?

[plus dans les leçons III.1 et III.2]

Un ordinateur est un exemple d'**automate programmable**.

Un **automate** est un dispositif capable d'assurer, sans intervention humaine, un enchaînement d'opérations correspondant à la réalisation d'une tâche donnée.

Exemples : montre, « ramasse-quilles », ...

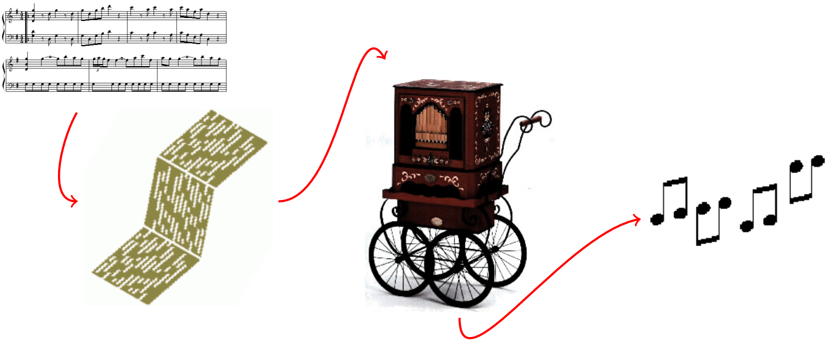
Un automate est **programmable** lorsque la nature de la *tâche* qu'il est capable de réaliser peut être *modifiée* à volonté.

Dans ce cas, la description de la tâche à réaliser se fait par le biais d'un **programme**, c.-à-d. une séquence d'instructions et de données susceptibles d'être comprises et exécutées par l'automate.

Exemples : métier à tisser Jacquard, orgue de barbarie, ... et l'ordinateur !

☞ **Formalisation :** *Machine de Turing universelle* [leçon I.3]

Exemple d'automate programmable



«**PROGRAMME**» :

Conception : quelles notes enchaîner ?

Réalisation : percer les trous aux bons endroits

Exécution : tourner la manivelle

Résultat : mélodie

Programmation de l'automate

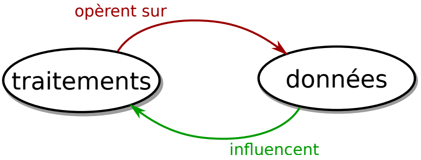
Un ordinateur doit donc permettre la *description* des différents traitements que l'on veut automatiser, des modèles que l'on veut simuler, des informations que l'on veut rechercher, ...

Cette description se fait en combinant :

- des données, qui permettent la représentation des objets du monde réel dans l'ordinateur ;
- opérations/traitements, qui permettent de manipuler les données et de modéliser les actions du monde réel.

Qu’est-ce que la programmation ? (résumé)

Programmer c’est décomposer la **tâche** à automatiser sous la forme d’une séquence d’instructions (**traitements**) et de **données** adaptées à l’automate utilisé.



Formalisation des traitements : **algorithmes**

☞ distinguer formellement les bons traitements des mauvais

Formalisation des données : **structures de données abstraites**

☞ distinguer formellement les bonnes structures de données des mauvaises

La **conception** consiste à choisir les bons algorithmes et bonnes structures de données pour résoudre un problème donné.

Algorithme $\neq$ Programme

Un algorithme est **indépendant du langage de programmation** dans lequel on va l'exprimer **et de l'ordinateur** utilisé pour le faire tourner.

C'est une description abstraite des étapes conduisant à la solution d'un problème.

algorithme = partie conceptuelle d'un programme (indépendante du langage)

programme = implémentation (réalisation) de l'**algorithme**, dans un langage de programmation et sur un système particulier.

Algorithme : un concept central

PageRank® : algorithme fondamental permettant au moteur de recherche Google de classer les pages web en fonction de leur popularité

Idée de base : le rang d'une page web (son importance) est mesuré en utilisant le nombre des autres pages la citant et leur rang. (définition par *récurrence*)

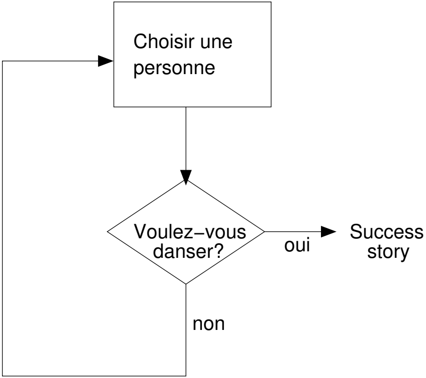
☞ L'algorithme est une valeur en soi (marque déposée)

☞ La clé historique du succès de Google

Voir https://www.youtube.com/watch?v=wR0wVxK3m_o

Algorithmes : introduction

« Voulez-vous danser ? » : premier algorithme :



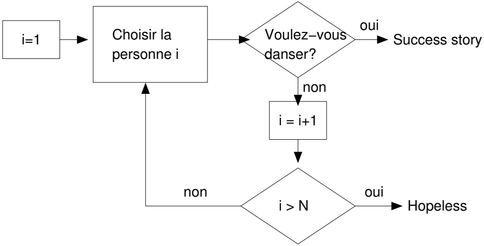
Données :

- Personne
- Ensemble de N personnes

☞ Il n'est pas garanti que l'algorithme puisse se terminer !

Algorithmes : introduction

« Voulez-vous danser ? » : deuxième algorithme :



Données :

- Personne
- Ensemble **ordonné** de N personnes

- les données sont *structurées*
  ☞ lien algorihtme / représentation des données
- l'algorithme se **termine** nécessairement (au pire N essais successifs)
- … mais il n'est pas sûr qu'il soit le plus *efficace* possible !

Qu'est ce qu'un algorithme ?

Algorithme ?

☞ moyen pour un humain de représenter (pour un autre humain ou une machine) la résolution par calcul d'un **problème**

« *spécification d'un schéma de calcul sous forme d'une suite d'opérations élémentaires obéissant à un enchaînement déterminé* » [Encyclopedia Universalis]

Les algorithmes existent depuis bien avant les ordinateurs : déjà dans l'Antiquité (e.g. *division égyptienne*, *algorithme d'Euclide*)

Origine du nom : mathématicien persan Al-Khawarizmi du 9e siècle, surnommé « le père de l'algèbre ».

Algorithme : exemples

<u>Exemples :</u>

- algorithmes de tri (d'objets, e.g. cartes à jouer)
- chemin le plus rapide pour venir à l'EPFL depuis chez soi (ou pour trouver le trajet le moins cher pour aller en vacances)
- algorithme d'Euclide (plus grand diviseur commun)
- résoudre une équation
- PageRank, EdgeRank, ...
- ...

Qu’est ce qu’un algorithme ?

Algorithme : composition d’un ensemble fini d’opérations élémentaires bien définies (déterministes) opérant sur un nombre fini de données et effectuant un traitement bien défini :

- suite finie de règles à appliquer,
- dans un ordre déterminé,
- à un nombre fini de données,
- si possible, se terminant (i.e. arriver, en un nombre fini d’étapes, à un résultat, et cela quelles que soient les données traitées).

Un algorithme peut être

- *séquentiel* : ses opérations s’exécutent en séquence
- *parallèle* : certaines de ses opérations s’exécutent en parallèle, simultanément
- *réparti* : certaines de ses opérations s’exécutent sur plusieurs machines (répartition géographique)

Définition formelle des algorithmes

Formalisation : dans les années (19)30 par des mathématiciens :
Gödel, Turing, Church, Post, Kleene, ...

☞ *fonctions « calculables »* et *machines de Turing* : abstraction mathématique des notions de **traitement** (suite d'opérations élémentaires), de **problème** et d'**algorithme**.

(cf leçon I.3)

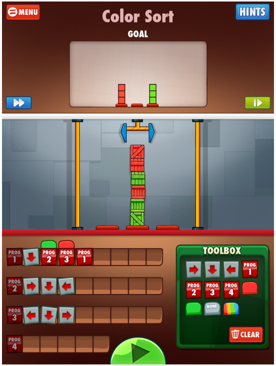
☞ Mais comment concrètement créer un algorithme ?

Plan

- Formaliser ces calculs : notion d'algorithme
- Présenter les « ingrédients de base » des algorithmes
- Quelques familles d'algorithmes

Comment créer un algorithme... sans le savoir

Exemple : le jeu Cargo-Bot : vous devez expliquer à un bras articulé comment l'on peut passer d'un état initial à un état final, en utilisant une séquence limitée d'instructions :



 pour prendre ou poser un cube

 pour déplacer le bras une fois vers la droite

 pour déplacer le bras une fois vers la droite *si* il porte un cube bleu

Vidéo : http://www.youtube.com/watch?v=i56nbx_6nLc

Comment créer un algorithme... sans le savoir

En résolvant un des problèmes de Cargo-Bot, vous concevez un algorithme :

- suite finie de règles à appliquer,
- dans un ordre déterminé,
- à un nombre fini de données (les cubes),
- si possible, se terminant.

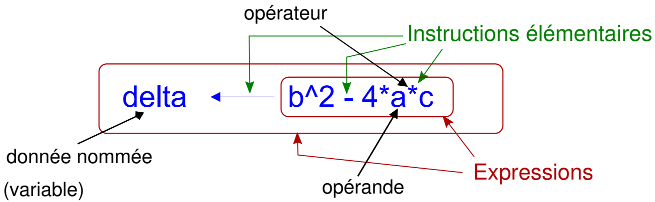
Données et instructions dans un algorithme

Un algorithme travaille sur des données qu'il utilise et/ou modifie.

☞ il doit **mémoriser** ces données, en les associant à un *nom*, pour pouvoir les retrouver au moment où elles lui sont nécessaires.

Une donnée nommée est ce que l'on appelle une *variable* dans un algorithme.

Les **traitements** sont associés à la notion d'**instructions** et d'**expressions**.



Instruction élémentaire

Certaines instructions sont dites **élémentaire** : « *atome de calcul* »

Une instruction élémentaire est une instruction dont le coût d'exécution est **constant** (ne dépend pas de la donnée que l'on manipule).

Exemples :

- instruction élémentaire : associer une donnée de base (comme un nombre) à un nom (variable)

  delta ⟵ b^2 - 4*a*c

- instruction non élémentaire : compter le nombre de caractères contenus dans une phrase (dépend de la longueur de la phrase).

Structures de contrôle

Pour pouvoir exprimer des traitements intéressants/complexes, un algorithme ne peut se réduire à une séquence linéaire d'instructions.

☞ **structures de contrôle**

Une structure de contrôle sert à **modifier l'ordre linéaire d'exécution** d'un programme.

- ☞ faire exécuter à la machine des tâches de façon *répétitive*, ou *en fonction de certaines conditions* (ou les deux).

Algorithmes : briques de base

Les différentes structures de contrôle

On distingue 3 types de structures de contrôle :

les branchements conditionnels : *si ... alors ...*

Si $\Delta = 0$
  $x \leftarrow -\frac{b}{2}$
Sinon
  $x_1 \leftarrow \frac{-b-\sqrt{\Delta}}{2}, \quad x_2 \leftarrow \frac{-b+\sqrt{\Delta}}{2}$

les boucles conditionnelles : *tant que ...*

Tant que réponse non valide
  poser la question

les itérations : *pour ... allant de ... à ...* , *pour ... parmi ...*

$$x = \sum_{i=1}^{5} \frac{1}{i^2}$$

$x \leftarrow 0$
Pour i de 1 à 5
  $x \leftarrow x + \frac{1}{i^2}$

Les différentes structures de contrôle

On distingue 3 types de structures de contrôle :

les branchements conditionnels : *si ... alors ...*

les boucles conditionnelles : *tant que ...*

les itérations : *pour ... allant de ... à ...* , *pour ... parmi ...*

Note : on peut toujours (évidemment !) faire des itérations en utilisant des boucles conditionnelles :

$x \leftarrow 0$
$i \leftarrow 1$
Tant que $i \leq 5$
  $x \leftarrow x + \frac{1}{i^2}$
  $i \leftarrow i + 1$

mais conceptuellement il y a une différence (notions d'ordonnancement, d'ensemble, d'itérateur).

Premier exemple concret

On veut écrire l'algorithme permettant de résoudre (dans $\mathbb{R}$) une équation du second degré de type :

$$x^2 + b\,x + c = 0$$

Pour b et c fixés, les solutions réelles d'une telle équation sont :

$$\begin{cases} \left\{ \frac{-b+\sqrt{\Delta}}{2}, \frac{-b-\sqrt{\Delta}}{2} \right\} & \text{si } \Delta > 0 \\ \left\{ \frac{-b}{2} \right\} & \text{si } \Delta = 0 \\ \emptyset & \text{sinon} \end{cases}$$

avec $\Delta = b^2 - 4\,c$

Conception de l'algorithme ?

☞ Facile dans un cas aussi simple (déjà bien formalisé au départ) mais peut devenir (très) complexe (☞ prochaine leçon)

Premier exemple concret : Formalisation des traitements

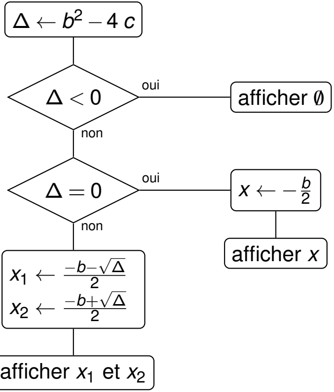
☞ **Algorithme** :

Second degré

entrée : b, c

sortie : $\{x \in \mathbb{R} : x^2 + b\,x + c = 0\}$

$\Delta \leftarrow b^2 - 4\,c$
Si $\Delta < 0$
    sortir : $\emptyset$
Sinon
    Si $\Delta = 0$
        $x \leftarrow -\frac{b}{2}$
        sortir : $\{x\}$
    Sinon
        $x_1 \leftarrow \frac{-b-\sqrt{\Delta}}{2}$,
        $x_2 \leftarrow \frac{-b+\sqrt{\Delta}}{2}$
        sortir : $\{x_1, x_2\}$



Algorithmes : conclusion

On attend d'un algorithme qu'il se termine, produise le résultat correct (solution du problème) pour toute entrée.

Difficulté de l'Informatique (science) : assurer que l'algorithme est correct pour toute entrée.

On ne peut pas vérifier par des essais (empirisme) : on ne pourra jamais tester tous les cas

☞ vérification par preuves mathématiques

Importance d'un travail soigneux et mûrement réfléchi !

Plan

- Formaliser ces calculs : notion d'algorithme
- Présenter les « ingrédients de base » des algorithmes
- Introduire quelques principales familles d'algorithmes :
  - recherche
  - tri
  - plus court chemin

Algorithme : exemples

Exemples :

- algorithmes de tri (d'objets, e.g. cartes à jouer)
- chemin le plus rapide pour venir à l'EPFL depuis chez soi (ou pour trouver le trajet le moins cher pour aller en vacances)
- algorithme d'Euclide (PGDC)
- résoudre une équation
- PageRank, EdgeRank, ...
- ...

☞ Comment, à partir d'un problème concret, trouver une solution ?

Types de problèmes algorithmiques

On retrouve trois grands types de problèmes (parmi d'autres) :

- Recherche
- Tri
- Calcul du plus court chemin

Recherche

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

AVANT TOUT : Spécification claire du problème et de l'algorithme voulu :

- E peut-il être vide ?
  E varie-t-il pendant la recherche ?
  E est-il ordonné ?
- algorithme :
  - séquentiel ?
    p.ex. : recherche d'un mot dans le dictionnaire
  - parallèle ?
    p.ex. : recherche d'un élève dans la salle
  - réparti ?
    p.ex. : recherche d'un objet perdu sur le campus
    $\longrightarrow$ demander à chaque concierge

Recherche

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

Considérons par exemple les deux algorithmes suivants :

appartient1

entrée : x, E
sortie : $x \in E$?

$i \longleftarrow 1$
Répéter
  Si $x = E[i]$
    sortir : $x \in E$
  $i \longleftarrow i + 1$
  $t \longleftarrow$ ***taille***(E)
jusqu'à $i > t$
sortir : $x \notin E$

appartient2

entrée : x, E
sortie : $x \in E$?

$t \longleftarrow$ ***taille***(E)
Pour *i de 1 à t*
  Si $x = E[i]$
    sortir : $x \in E$
sortir : $x \notin E$

autre algorithme (qui calcule la taille de E)

Complexité d'un algorithme

Première question : ces algorithmes sont ils **corrects** ?

- se terminent ils pour tous les cas ?
- donnent ils ce que l'on veut ?
  (p.ex. quid de **appartient1** si E est vide ?
  quid de **appartient2** si E est modifié pendant le parcours ?)

☞ démonstrations mathématiques

2e question : lequel des deux est le **plus efficace** ?

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

complexité (temporelle pire 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) :

1	affectation de la valeur 1 à la variable i	1 instruction
2	accès au i-ème élément de E et comparaison de cet élément avec x	2 instructions
3	incrément de i (de 1)	1 instruction
4	calcul de la taille de E	$T(n)$ instructions
5	vérification de la condition $(i < t)$ et retour en 2	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.

☞ $C1(n) = 1 + n\Big(T(n) + 4\Big)$

Complexité d'un algorithme : exemple

Pour l'algorithme appartient2(x, E) :

1	calcul de la taille de E	$T(n)$ instructions
2	affectation de la valeur 1 à i	1 instruction
3	vérification de la condition $(i < t)$	1 instruction
4	accès au i^{e} élément de E et comparaison de cet élément avec x	2 instructions
5	incrément de i (de 1) et retour en 3	1 instruction

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.

☞ $C2(n) = T(n) + 1 + 4\,n$

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 :

$$C1(n) = 1 + (a + 4)\, n + b\, n^2$$

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

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

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

☞ oui, si la liste est **ordonnée**

Dichotomie

appartient_D
entrée : x, E ***ordonnée***   sortie : $x \in E$?
Si *E est vide*   ***sortir :*** $x \notin E$ ***Si*** *E est réduit à 1 seul élément e*   ***sortir :*** $x = e$ *?* *(i.e. « vrai » si $x = e$ et « faux » sinon)*   *découper E en deux sous-ensembles non vides et disjoints E_1 et E_2* *(le plus optimal étant au milieu de E)*   ***Si*** $x \leq \max(E_1)$ ***sortir : appartient_D***(x, E_1) ***Sinon***   ***sortir : appartient_D***(x, E_2)

☞ Cet algorithme s'appelle la recherche par dichotomie

Exemple de recherche par dichotomie dans une liste ordonnée

$$\text{appartient\_D}(x, E) = ??$$

$E =$ abaque
abasourdi
babouin
baobab
blanc
bleu
zoulou

$x =$ bleu

(place pour prendre des notes)

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 la liste, il faudra répéter la boucle de découpage en deux autant de fois.

On va donc boucler autant de fois qu'on peut diviser n par 2.

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

☞ $\log_2 n$

Note : vous verrez aussi en leçon II.3 combien cette idée de dichotomie est lié à la notion d'*information* !

Rappels :

$$2^y = x \Longrightarrow y = \log_2(x)$$

$$\log_2(x) = \frac{\log_{10}(x)}{\log_{10}(2)}$$

Complexité : notation $\mathcal{O}(\ldots)$

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 $\mathcal{O}(\ldots)$

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

$$f \in \mathcal{O}(g)$$

si et seulement si

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

Dans ce cas, on dit que f est « **en grand O** » de g.

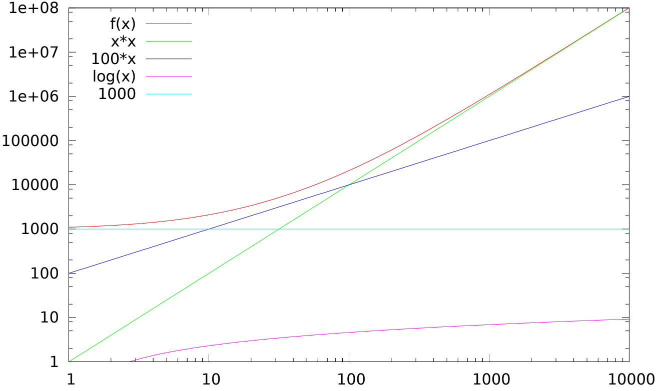
Notation « grand O » : exemple

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

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 **le plus petit des « grands O » possibles** (souvent noté Θ, cf annexes)

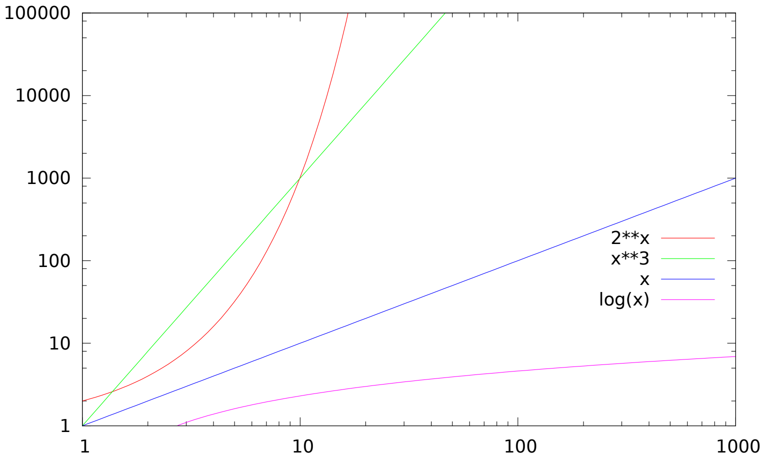
Exemples :

$$n \text{ est } \mathcal{O}(n^2) \text{ mais } n \text{ est aussi } \mathcal{O}(n)$$
$$12 \text{ est } \mathcal{O}(n^2),\ \mathcal{O}(n), \text{ mais surtout } \mathcal{O}(1)$$

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

- complexité constante $\mathcal{O}(1)$: le nombre d'éléments n'a pas d'influence sur l'algorithme
- complexité logarithmique $\mathcal{O}(\log n)$
- complexité linéaire $\mathcal{O}(n)$
- complexité quasi-linéaire $\mathcal{O}(n \log(n))$
- complexité polynomiale $\mathcal{O}(n^2)$, ... $\mathcal{O}(n^k)$
- complexité exponentielle $\mathcal{O}(2^n)$

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 continue jusqu'à Zürich !

Si elle ne doit en contrôler que le log :

que 18 passeports à contrôler ! !

(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 ($\mathcal{O}(n)$, où n est la taille de la liste)
- si la liste *ordonnée* : recherche par dichotomie, complexité logarithmique ($\mathcal{O}(\log n)$)

☞ importance de la *modélisation des données*

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

Les tris

Les méthodes de tris sont très **importantes en pratique** non seulement en soi, mais aussi parce qu'elle interviennent dans beaucoup d'autres algorithmes.

Elles sont par exemple nécessaires pour faire une recherche efficace.

Le problème du tri est également 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 abstraite 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 dans la machine, sans utiliser de support extérieur (fichier).

Un premier exemple : le tri par insertion

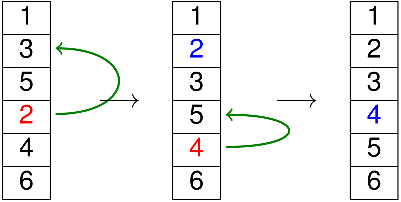
Le principe du **tri par insertion** est extrêmement simple :

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*

Par « *élément mal placé* », on entend ici tout élément du tableau strictement plus petit que son prédécesseur.

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

Algorithmes de tri

Il existe un grand nombre d'algorithmes de tri :

- récursif
- 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	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$
insertion	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$
de Shell	–	$\mathcal{O}(n^{1.5})$
quick sort	$\mathcal{O}(n \log n)$	$\mathcal{O}(n^2)$
par tas	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$

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)

Problème de plus court chemin

Une troisième famille classique de problèmes très répandus est celle des **plus court chemins**

Note : « plus court » en un certain sens qui peut aussi être « moins cher », « plus rapide », ... ☞ notion de fonction de coût / pondération

Exemples :

- calcul du chemin le plus rapide entre *toutes* les gares du réseau CFF (2 à 2)

  (Nous verrons dans la prochaine leçon une solution à ce problème)
- calcul du chemin le plus rapide entre *une* gare *donnée* et toutes les *autres* gares
- calcul du chemin le plus rapide entre *deux* gares *données*

mais aussi (!) :

- résoudre le Rubik's cube en un nombre minimum de mouvements
- transcrire un texte lu (reconnaissance de la parole)
- corriger les erreurs dans une communication satellite (codes de convolution)

Ce que j'ai appris aujourd'hui

Dans cette leçon, vous avez

- appris ce qu'est un **algorithme** et ses principaux constituants
- appris à comparer l'efficacité de deux algorithmes : **complexité**
- vu trois familles de problèmes typiques en Informatique (recherche, tris, plus court chemin)
- 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 :

- comprendre les problèmes de base de l'Informatique (recherche, tris, plus court chemin)
- construire des algorithmes simples pour des problèmes simples typiques
- calculer la complexité d'algorithmes simples

La suite

La prochaine leçon :

- présentera les *stratégies* de conception d'algorithme :
  comment concevoir un algorithme face à un problème complexe ?
- algorithme(s) de plus court chemin

Annexe mathématique

Pour information (hors cours) :

Dans les notations asymptotiques, on utilise aussi souvent :

- Ω : $f \in \Omega(g) \iff g \in \mathcal{O}(f)$
- Θ : $f \in \Theta(g) \iff f \in \mathcal{O}(g)$ <u>et</u> $f \in \Omega(g)$

Exemples :

$f(x) = ax^2 + bx + c$, $a > 0$

$f \in \mathcal{O}(x^2)$, mais aussi $f \in \mathcal{O}(x^3)$

$f \in \Omega(x^2)$, mais aussi $f \in \Omega(x)$

$f \in \Theta(x^2)$

Annexe informatique

Pour information (hors cours) :

En fait, pour un nombre entier n représenté en binaire, les opérations arithmétiques (et comparaison) ne sont pas en temps constant, mais ont les complexités suivantes (si la valeur de n peut croitre à l'infini, donc pas de représentation de taille fixe ; cf leçon I.4) :

comparaison	$\log(n)$
addition	$\log(n)$
multiplication	$\simeq \log(n) \log(\log(n)) \log(\log(\log(n)))$
division	$\mathcal{O}(\text{multiplication})$
racine carrée	$\mathcal{O}(\text{multiplication})$