# Algorithmique
# M2 Data Science
## cours 1 : une introduction

Vincent Runge

Section 1

# Introduction

# Algorithmique et algorithme

Définitions :

**Algorithmique** : science de la conception et de l'analyse des algorithmes

**Algorithme** : suite finie et non ambiguë d'opérations permettant de résoudre un problème

- Il prend en **entrée** une valeur (ou un ensemble de valeurs) et donne en **sortie** une valeur (ou un ensemble de valeurs).

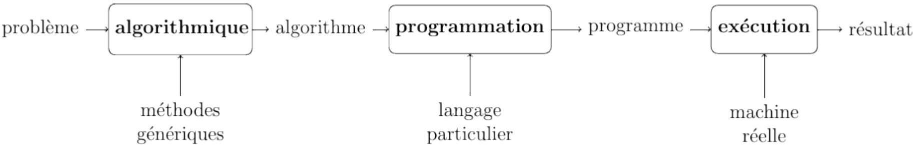



**Figure 1:** Ne pas confondre algorithmique, algorithme, programmation, programme

# Algorithmique et algorithme

1. IIIème siècle avant J.-C. (Grèce) : algorithme d'Euclide pour le calcul du pgcd (1er algorithme connu)
2. Le terme **algorithme** vient de la déformation du nom du mathématicien perse Al-Khwârizmî (780-850)
3. Le concept d'algorithmie, sera théorisé par Alan Turing en 1936 avec la machine de Turing

*"Computer science is not about machines, in the same way that astronomy is not about telescopes"* (Michael Fellows)

# Algorithmique et algorithme

Un bon algorithme :

1. Un algorithme **correct**
    - l'algorithme se termine en un temps fini (*preuve de terminaison*)
    - l'algorithme produit la bonne sortie/réponse (*preuve de correction*)
2. Un algorithme **efficace**
    - détermination du temps d'exécution nécessaire et de l'espace mémoire nécessaire à la résolution du problème (en fonction de la taille de l'entrée).
    - pour un problème donné, *plusieurs algorithmes* (ou aucun) sont possibles: on recherche alors *le ou les plus efficaces*

# (1) L'algorithme correct

Pour les suites de Collatz (appelé aussi suites de Syracuse), on ne sait toujours pas si **cet algorithme** simple se termine quel que soit *n*:

```
collatz(n:int)
  afficher n;
  si n == 1 alors stop, sinon
  si n%2 == 0 alors collatz(n/2), sinon collatz(3*n+1), finsi
  finsi
```

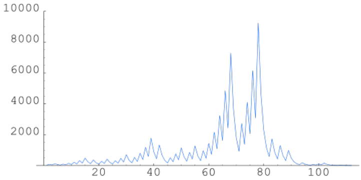



**Figure 2:** Représentation graphique de la suite collatz 27

# (1) L'algorithme correct

Pour les suites de Collatz (appelé aussi suites de Syracuse), on ne sait toujours pas si **ce programme R** simple se termine quel que soit $n$:

```
collatz <- function(n)
{
  print(n)
  if(n == 1){return("stop")}
  if(n%%2 == 0){collatz(n/2)}else{collatz(3*n+1)}
}
```

Ni évaluer le nombre d'opérations nécessaires pour terminer (s'il termine)

Pour votre culture (un article sur la conjecture de Collatz)

# Un cadre plus large : l'informatique théorique

complexité des algorithmes $\subset$ algorithmique $\subset$ informatique théorique

1. Algorithme **correct** : notions de machine de Turing, de calculabilité, problème de l'arrêt. . .
2. Algorithme **efficace : la complexité algorithmique, la théorie de la complexité**

Ce cours porte exclusivement sur le point 2

Section 2

# Complexité des algorithmes

# Mesurer le temps d'exécution

1. Le temps d'exécution **dépend souvent de la forme de l'entrée** (par exemple dans le cas d'un algorithme de tri, si le tableau est déja trié)
2. On cherche **une fonction T(n)** représentant le temps d'exécution de l'algorithme en fonction de la taille de l'entrée $n$ (par exemple la taille d'un vecteur)
3. Le calcul exact est impossible, on cherche alors des ordres de grandeur pour:

- le meilleur des cas
- **le pire des cas**
- le cas moyen (il faut connaître la distribution statistique des entrées)

# Exemple du tri par insertion

**Algorithm** TRI-INSERTION (A)

```
1: for j ← 2 to n do
2:    key ← A[j]
3:    i ← j − 1
4:    while i > 0 and A[i] > key do
5:       A[i + 1] ← A[i]
6:       i ← i − 1
7:    end while
8:    A[i + 1] ← key
9: end for
```

- Soit $c_i$ le coût d'exécution (en temps) de chaque ligne
- Soit $t_j$ le nombre de fois que la boucle while est exécutée pour l'indice j

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1)$$

$$+ \sum_{j=2}^{n} \Big( c_4 t_j + c_5(t_j - 1) + c_6(t_j - 1) \Big) + c_8(n-1)$$

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1)$$
$$+ \sum_{j=2}^{n} \Big( c_4 t_j + c_5(t_j - 1) + c_6(t_j - 1) \Big) + c_8(n-1)$$

1. **Cas le plus favorable :** le tableau d'entrée est déjà trié et donc pour tout $j$ dans $\{2, ..., n\}$ on a $t_j = 1$. Temps d'exécution :

$$T(n) = (c_1 + c_2 + c_3 + c_4 + c_8)n - (c_2 + c_3 + c_4 + c_8)$$

   Temps sous la forme $an + b$ : **fonction linéaire** en $n$.

2. **Cas le plus défavorable :** la tableau d'entrée est trié en ordre décroissant donc pour tout $j$ dans $\{2, ..., n\}$ on a $t_j = j$. Temps d'exécution :

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1)$$
$$+ c_4(\frac{n(n+1)}{2} - 1) + c_5 \frac{n(n-1)}{2} + c_6 \frac{n(n-1)}{2}) + c_8(n-1)$$

   Temps sous la forme $an^2 + bn + c$ : **fonction quadratique** en $n$

# Code R du tri par insertion

```
insertionsort_function <- function(v)
{
  for (j in 2:length(v))
  {
    key <- v[j]
    i <- j - 1
    while (i > 0 && v[i] > key)
    {
      v[i + 1] = v[i]
      i <- i - 1
    }
    v[i + 1] <- key
  }
  return(v)
}
```

# tests avec les données $n, n-1, \ldots, 2, 1$

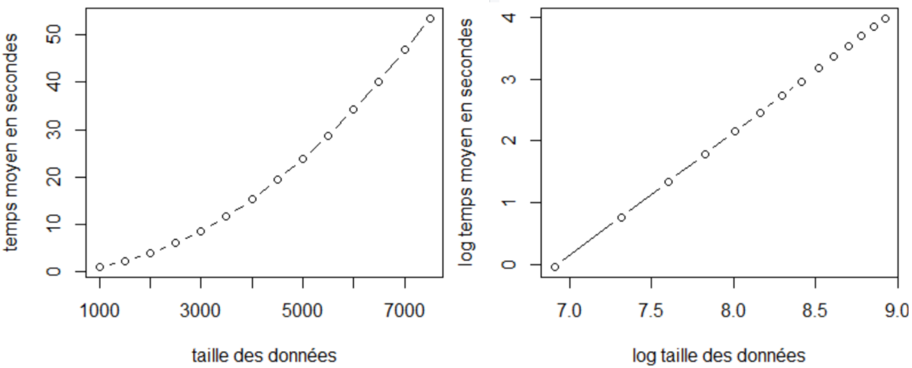



**Figure 3:** Moyenne du temps d'exécution sur 50 simulations pour chaque n avec les données n,n-1,. . . ,2,1. La régression linéaire sur le logarithme des données donne un coefficient directeur de 1.997

# Quelques algorithmes de tri (source wikipédia)

Tableau comparatif des tris procédant par comparaisons

| Nom | Cas optimal | Cas moyen | Pire des cas | Complexité spatiale | Stable |
|---|---|---|---|---|---|
| Tri rapide | $n \log n$ | $n \log n$ | $n^2$ | $\log n$ en moyenne, $n$ dans le pire des cas ; variante de Sedgewick : $\log n$ dans le pire des cas | Non |
| Tri fusion | $n \log n$ | $n \log n$ | $n \log n$ | $n$ | Oui |
| Tri par tas | $n \log n$ | $n \log n$ | $n \log n$ | $1$ | Non |
| Tri par insertion | $n$ | $n^2$ | $n^2$ | $1$ | Oui |
| Introsort | $n \log n$ | $n \log n$ | $n \log n$ | $\log n$ | Non |
| Tri par sélection | $n^2$ | $n^2$ | $n^2$ | $1$ | Non |
| Timsort | $n$ | $n \log n$ | $n \log n$ | $n$ | Oui |
| Tri de Shell | $n$ | $n \log^2 n$ ou $n^{3/2}$ | $n \log^2 n$pour la meilleure suite d'espacements connue | $1$ | Non |
| Tri à bulles | $n$ | $n^2$ | $n^2$ | $1$ | Oui |
| Tri arborescent | $n \log n$ | $n \log n$ | $n \log n$ (arbre équilibré) | $n$ | Oui |
| Smoothsort | $n$ | $n \log n$ | $n \log n$ | $1$ | Non |
| Tri cocktail | $n$ | $n^2$ | $n^2$ | $1$ | Oui |
| Tri à peigne | $n$ | $n \log n$ | $n^2$ | $1$ | Non |
| Tri pair-impair | $n$ | $n^2$ | $n^2$ | $1$ | Oui |

(Un tri est dit stable s'il préserve l'ordonnancement initial des éléments que l'ordre considère comme égaux)

Mais où sont passées les constantes $c_i$ ???

# Quel algorithme choisir?

- On peut se baser sur la connaissance de la forme des vecteurs d'entrée ("presque" triées? de nombreuses répétitions?...)
- Ou sur la *distribution* des vecteurs d'entrée (vecteur d'entiers? bornées?). Dans ce cas, des tris en temps linéaire existent (*radix sort, counting sort, bucket sort*)
- On doit parfois tenir compte de la potentielle contrainte d'espace mémoire

# Comparaisons asymptotiques (notation de Landau) : définitions

- $u_n = O(v_n)$ s'il existe $M > 0$ tel que un $\frac{u_n}{v_n} \leq M$ à partir d'un certain rang
- $u_n = \Omega(v_n)$ s'il existe $m > 0$ tel que un $\frac{u_n}{v_n} \geq m$ à partir d'un certain rang
- $u_n = \Theta(v_n)$ s'il existe $m, M > 0$ tels que $m \leq \frac{u_n}{v_n} \leq M$ à partir d'un certain rang.

**On utilisera (comme dans la plupart des cours) la notation $O$ pour en fait signifier $\Theta$ (mais pas toujours... c'est souvent au lecteur de comprendre le contexte)**

Pour l'algorithme de tri par insertion, on a le temps $T(n) = O(n^2)$

(et $T(n) = \Omega(n)$ si la liste est déjà triée)

# Les coût élémentaires

Chacune des opérations élémentaires a une certaine durée d'exécution :

- l'affectation
- les comparaisons
- les opérations arithmétiques

D'où la nécessité d'avoir **une constante pour chaque type d'opération** pour un calcul exact : $c+$, $c-$, $c*$, $c<$,...

Complexité = sommer tous les temps d'exécution des différentes opérations effectuées lors de l'exécution de l'algorithme.

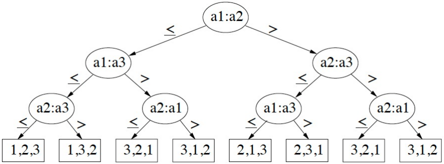
**MAIS** modifier strictement positivement les constantes n'a pas d'effet sur le comportement asymptotique de la complexité ($O$, $\Theta$, $\Omega$).

**Conclusion :**

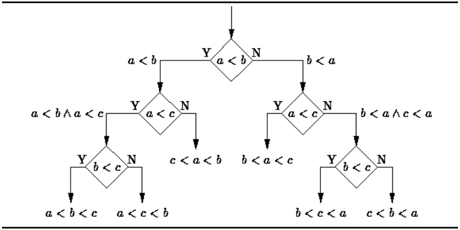
On considère $c = 1$ pour tous les calculs de complexité.

# Exemple du tri par comparaison: $T(n) = \Omega(n \log(n))$

- Seule opération utilisée : la comparaison $(<, >, \leq, \geq)$
- Trier $=$ trouver une permutation des indices 2 par 2 qui amène au vecteur trié.
- tri par comparaisons successives se modélise comme un **arbre binaire**. Chaque nœud de l'arbre correspondant à une comparaison entre deux éléments
- On a au moins $n!$ feuilles



# Exemple du tri par comparaison: $T(n) = \Omega(n \log(n))$




- la hauteur $h$ de l'arbre = le temps min d'exécution
- hauteur $h$ = au plus $2^h$ feuilles, on résout:

$$2^h = 2^{T(n)} \geq n!$$

$$T(n) \geq \log_2 \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

$$= \frac{1}{2}\log_2(2\pi n) + n\log_2(n) - n\log_2(e) \in \Omega(n\log(n))$$

# Deux complexités

- Complexité d'**un algorithme**: Borne supérieure asymptotique du temps (complexité au pire)
- Complexité d'**un problème**: La meilleure des pires complexités des algorithmes répondant au problème

Exemple:

1. Complexité du tri par insertion : $O(n^2)$
2. Complexité du problème de tri : $O(n\log(n))$

# Ordre de grandeur en temps d'exécution (1)

**Ordre de grandeur du temps nécessaire à l'exécution d'un algorithme d'un type de complexité**

| Temps | Type de complexité | Temps pour n = 5 | Temps pour n = 10 | Temps pour n = 20 | Temps pour n = 50 | Temps pour n = 250 | Temps pour n = 1 000 | Temps pour n = 10 000 | Temps pour n = 1 000 000 | Problème exemple |
|---|---|---|---|---|---|---|---|---|---|---|
| $O(1)$ | complexité constante | 10 ns | 10 ns | 10 ns | 10 ns | 10 ns | 10 ns | 10 ns | 10 ns | accès à une cellule de tableau |
| $O(\log(n))$ | complexité logarithmique | 10 ns | 10 ns | 10 ns | 20 ns | 30 ns | 30 ns | 40 ns | 60 ns | recherche dichotomique |
| $O(\sqrt{n})$ | complexité racinaire | 22 ns | 32 ns | 45 ns | 71 ns | 158 ns | 316 ns | 1 µs | 10 µs | test de primalité naïf |
| $O(n)$ | complexité linéaire | 50 ns | 100 ns | 200 ns | 500 ns | 2.5 µs | 10 µs | 100 µs | 10 ms | parcours de liste |
| $O(n \log(n))$ | complexité linéarithmique | 40 ns | 100 ns | 260 ns | 850 ns | 6 µs | 30 µs | 400 µs | 60 ms | tris par comparaisons optimaux (comme le tri fusion ou le tri par tas) |
| $O(n^2)$ | complexité quadratique (polynomiale) | 250 ns | 1 µs | 4 µs | 25 µs | 625 µs | 10 ms | 1 s | 2.8 heures | parcours de tableaux 2D |
| $O(n^3)$ | complexité cubique (polynomiale) | 1.25 µs | 10 µs | 80 µs | 1.25 ms | 156 ms | 10 s | 2.7 heures | 316 ans | multiplication matricielle naïve |
| $O(n!)$ | complexité factorielle | 1.2 µs | 36 ms | 770 ans | $10^{48}$ ans | ... | ... | ... | ... | problème du voyageur de commerce avec une approche naïve |

# Ordre de grandeur en temps d'exécution (2)

## Temps approximatif de calcul

Hypothèses : Donnée de taille $n = 10^6$, 1 milliard d'opérations par seconde, le terme dans le $O$ donne le nombre d'opérations.

- $O(1)$ : 1 ns
- $O(\ln(n))$ : 15 ns
- $O(n)$ : 1 ms
- $O(n \ln n)$ : 15 ms
- $O(n^2)$ : 15 min
- $O(n^3)$ : 30 ans
- $O(2^n)$ : $10^{300000}$ milliards d'années !

On voit que passer d'une complexité quadratique $O(n^2)$ à $O(n \log(n))$ peut avoir une conséquence pratique très importante.

# Ordre de grandeur en temps d'exécution (2)

Sur un exemple

```
devtools::install_github("vrunge/M2algorithmique")
library(M2algorithmique)

n <- 10^6
v <- n:1

system.time(insertion_sort_Rcpp(v))[[1]]
system.time(heap_sort_Rcpp(v))[[1]]
```

On obtient 202.524 secondes pour *insertion_sort_Rcpp* et 0.15 seconde pour *heap_sort_Rcpp*.

Section 3

# Quelques exemples

# Exemple 1 : les opérations mathématiques élémentaires

1. **La multiplication de deux nombres à n-digit** peut se faire en temps $\mathbf{O}(\mathbf{n}\log(\mathbf{n}))$ (à l'école on apprend la méthode en $O(n^2)$)
2. **La multiplication de deux matrices de taille n** peut se faire en temps $\mathbf{O}(\mathbf{n}^{\mathbf{2.373}})$ par l'agorithme de Coppersmith-Winograd (à l'université on apprend la méthode en $O(n^3)$)

Voir la page wikipédia

Ces remarques justifient (parfois, souvent) l'utilisation de **librairies d'algèbre linéaire** pour effectuer ces opérations efficacement. (**Armadillo** ou **Eigen** pour le C++ par exemple : l'optimisation est aussi en terme de gestion optimale des ressources de votre machine (calcul multithread))

# Exemple 1 : les opérations mathématiques élémentaires

**Multiplication matricielle :**

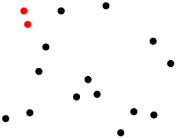
(source wiki)

*The optimal number of field operations needed to multiply two square $n \times n$ matrices up to constant factors is* ***still unknown****. This is* ***a major open question*** *in theoretical computer science.*

*As of December 2020, the matrix multiplication algorithm with best asymptotic complexity runs in $O(n^{2.3728596})$* ***time****, given by Josh Alman and Virginia Vassilevska Williams. However, this and similar improvements to Strassen are not used in practice, because they are* ***galactic algorithms****: the constant coefficient hidden by the Big O notation is so large that they are only worthwhile for matrices that are too large to handle on present-day computers.*

# Exemple 2 : la géométrie algorithmique

Soient $n$ points dans le plan euclidien. **Déterminer les deux point les plus proches** (*closest pair of points problem*)



- Il existe une solution évidente en $O(n^2/2)$ (calculer toutes les paires)
- Mais on peut faire mieux !!! => En $O(n \log(n))$

On va regarder et commenter ensemble le preuve en image !

# Exemple 3 : Machine Learning / IA (a)

# Exemple 3 : Machine Learning / AI (b)

| Algorithm | Classification/Regression | Training | Prediction |
| --- | --- | --- | --- |
| Decision Tree | C+R | $O(n^2p)$ | $O(p)$ |
| Random Forest | C+R | $O(n^2pn_{trees})$ | $O(pn_{trees})$ |
| Random Forest | R Breiman implementation | $O(n^2pn_{trees})$ | $O(pn_{trees})$ |
| Random Forest | C Breiman implementation | $O(n^2\sqrt{p}n_{trees})$ | $O(pn_{trees})$ |
| Extremly Random Trees | C+R | $O(npn_{trees})$ | $O(npn_{trees})$ |
| Gradient Boosting ( $n_{trees}$) | C+R | $O(npn_{trees})$ | $O(pn_{trees})$ |
| Linear Regression | R | $O(p^2n+p^3)$ | $O(p)$ |
| SVM (Kernel) | C+R | $O(n^2p+n^3)$ | $O(n_{sv}p)$ |
| k-Nearest Neighbours (naive) | C+R | $-$ | $O(np)$ |
| Nearest centroid | C | $O(np)$ | $O(p)$ |
| Neural Network | C+R | ? | $O(pn_{l_1}+n_{l_1}n_{l_2}+\dots)$ |
| Naive Bayes | C | $O(np)$ | $O(p)$ |

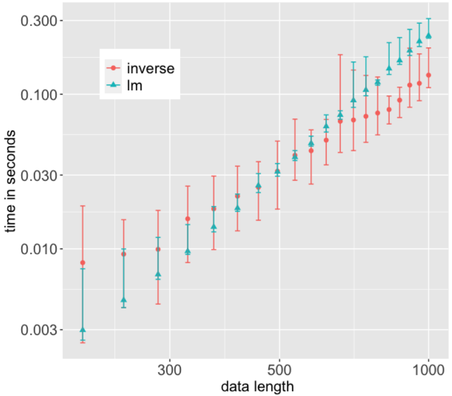
# Exemple 3 : Machine Learning / AI (c)

Pour la régression linéaire : $O(p^2n + p^3)$ pourquoi?

- La solution est $\hat{\beta} = (X^TX)^{-1}X^TY$
- Le coût du calcul de la matrice de "variance-covariance" $X^TX$ est le plus important : $O(p^2n)$ (en effet $n \geq p$ en régression multiple)

# Exemple 3 (bis) : tests sur la régression linéraire multiple

Régression avec n = p. On doit observer une complexité $O(n^3)$ (au plus)

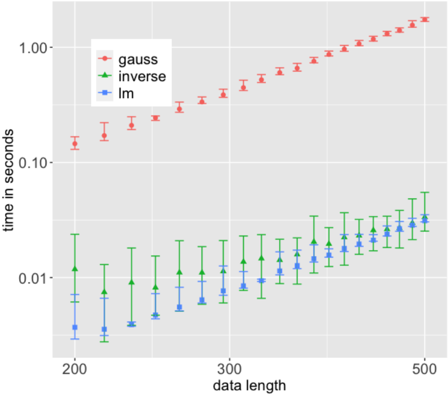



```
> LMres_inv$coefficients ### methode matricielle
(Intercept)      log(n)
 -14.843334    1.845123
> LMres_lm$coefficients ### fonction lm
(Intercept)      log(n)
 -20.767822    2.797414
```

Comment étudier la complexité d'un algorithme *inconnu* (qu'on n'a pas soit même implémenté par des simulations)? (*reverse engineering*)

# Exemple 3 (ter) : tests sur la régression linéaire multiple

On obtient $O(n^r)$ avec $r < 3$, pourquoi??




idem + *gaussian elimination*

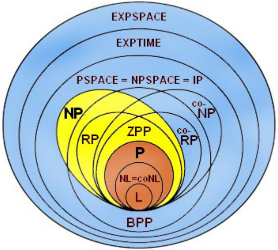
```
> LMres_inv$coefficients
(Intercept)      log(n)
 -13.054286    1.534123
> LMres_lm$coefficients
(Intercept)      log(n)
 -19.353397    2.550344
> LMres_gauss$coefficients
(Intercept)      log(n)
 -16.487962    2.738269
```

Section 4

# Million dollar baby ou de la théorie de la complexité

# Théorie de la complexité

Les algorithmes peuvent se ranger en des **classes de complexité**. Il en existe des centaines. . . On peut les représenter avec un diagramme de Venn.




voir aussi Complexity Zoo et Complexity diagram

---

(Problème de décision : algorithme dont la résponse est oui ou non)

Les deux plus connues :

- la **classe de complexité P** des problèmes de décision admettant un algorithme de résolution s'exécutant en temps polynomial
- la **classe de complexité NP** des problèmes de décision dont la vérification du résultat demande un temps polynomial

# Million dollar baby ou P = NP ?

Le problème : Est-ce que les algorithmes de la **classe de complexité NP** admettent une stratégie de résolution en temps polynomial?

«Tout ce que l'on peut vérifier facilement, peut-il être découvert aisément?»

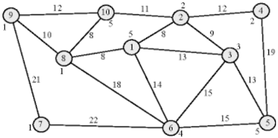
Clay Mathematics Institute

L'Institut de mathématiques Clay offre $10^6$\$ à quiconque sera en mesure de démontrer $P = NP$ ou $P \neq NP$ ou de démontrer que ce n'est pas démontrable.

# Le problème du voyageur de commerce

## Un problème d'optimisation combinatoire NP-complet

Étant donné une liste de villes, et des distances entre toutes les paires de villes, déterminer **un plus court chemin** qui visite chaque ville une et une seule fois et qui termine dans la ville de départ.



- Problème de **classe NP** pour le problème de décision (réponse T or F): *Existe-t-il un chemin plus court qu'une longueur donnée L?*
- On ne connait pas d'algorithme de résolution en temps polynomial
- Pour $n$ villes, on a $\frac{(n-1)!}{2}$ chemins à visiter
- La **programmation dynamique** permet de réduire la complexité en temps à $O(n^2 2^n)$

# Le problème du voyageur de commerce



<u>Propriétés :</u>

- Tous les problèmes NP se ramènent aux problèmes NP-complets
- Trouver un algorithme polynomial pour un problème NP-complet ou prouver qu'il n'en existe pas permet de savoir si $P = NP$ ou $P \neq NP$.

# from NP to P !!!

PRIMES complexité en temps $O(n^{12})$ (prouvé en 2002) puis $O(n^6)$ (prouvé en 2006) pour tester la primalité d'un entier.

page wikipédia

## PRIMES is in P

Manindra Agrawal    Neeraj Kayal
Nitin Saxena*

Department of Computer Science & Engineering
Indian Institute of Technology Kanpur
Kanpur-208016, INDIA
Email: {manindra,kayaln,nitinsa}@iitk.ac.in

**Abstract**

We present an unconditional deterministic polynomial-time algorithm that determines whether an input number is prime or composite.

# Références web

Science étonnante (Nos algorithmes pourraient-ils être plus rapides?)

voyageur de commerce le jeu!!!

# L'analyse de la complexité

- $\neq$ théorie de la complexité (avec ses classes de complexité P, NP... et donc ses **classes d'algorithmes**)
- $=$ étude formelle de la quantité de ressources (temps et/ou d'espace) nécessaire à l'exécution d'un algorithme

Section 5

# Programme et évaluation de ce cours

# Programme

L'objet de ce cours :

- paradigmes algorithmiques (récursif, programmation dynamique. . . )
- la complexité des algorithmes (temps d'exécution et mémoire)
- structures de données (tableaux, listes chaînées, arbres. . . )

6 cours de 3h.

1. Présentation, code R, package R, Rcpp
2. Diviser pour régner (la récursion)
3. Structures de données (tableaux, listes chaînées, arbres)
4. Programmation dynamique
5. Algorithmes non-exacts
6. Algorithmes non-exacts suite

# Evaluation : un projet par groupe

Chaque *groupe de 3-4 étudiants* choisit un *problème d'algorithmique* à traiter

**Avec les éléments suivants :**

- présentation de la problématique
- solution naïve
- solution(s) améliorée(s)
- présentation du package
- évaluation sur des exemples

Pour chaque algorithme (écrit en R **et** en C++) il faudra

1. Réaliser une analyse des temps de calcul théorique et vérifier ces résultats avec le code.
2. Comparer les temps de calcul entre code R et code C++ pour un même algorithme

# Notre outil : Package R avec C++ (Rcpp)

Chaque groupe devra réaliser

- Un package Rcpp (R avec C++)
- Sa mise à disposition sur github
- Une présentation scientifique

**À installer sur votre ordinateur :** R, Rstudio, compilateur gcc

**Package R à installer :**

- knitr, rmarkdown, markdown, prettydoc
- Rcpp
- devtools, roxygen2
- ggplot2, microbenchmark

# TP R/Rcpp (first and last)

## R, Rstudio, R packages, Rcpp, algorithmes de tri

1. On va commencer par apprendre **un peu de R**
   - écrire des fonctions
   - créer un package R
   - évaluer la vitesse des fonctions créées
2. À cela on va rajouter les **codes équivalents en C++ avec le package Rcpp**
3. Puis comparer les temps entre toutes les fonctions sur des **simulations**

=> **Ecrire au moins 2 algorithmes de tri**