Outils Formels pour l'Informatique

Jean-Marc Fédou
fedou@unice.fr

Présentation

Ce cours abordera :

* les mathématiques pour l'informatique (mathématiques discrètes, logique, dénombrement)
* l'étude des performances des ordinateurs (complexité)
* la formalisation des objets informatiques (induction, langages formels, théorie des graphes)

Bibliographie

* Concepts fondamentaux de l'informatique
  Aho & Ullman, éd. Dunod.

* Mathématiques pour l'informatique
  A. Arnold & I. Guessarian, éd. EdiScience.

* Méthodes mathématiques pour l'informatique
  J. Vélu, éd. Dunod.

* Introduction à l'algorithmique
  Cormen, Leiserson & Rivest, éd. Dunod.

* Mathématiques discrètes et informatique,
  N.H. Xuong, éd. Masson.

Organisation du semestre

* Début des cours le mercredi 18 Septembre
* Début des TD le mercredi 18 Septembre
    * Interrogations écrites de 5 mn en TD, sur le cours et les exercices simples de la feuille de TD en début de deuxième séance

* **FAIRE** les exercices d'échauffement
* **APPRENDRE** le cours
* **FAIRE** ou **CHERCHER** les autres exercices

AVANT le TD

Evaluation

* Deux devoirs surveillés (coeff 1/3 chacun)
  * Novembre (à confirmer)
  * Janvier (à confirmer)
* Moyenne des interrogations écrites en TD (coeff 1/3)

Supports

* Supports disponibles sur ma page web
  http://www.i3s.unice.fr/~fedou
* Transparents
* Feuilles de TD

Emploi du temps

* Cours Mercredi 8h00-9h30h, en amphi de chimie Intervenant Jean-Marc Fédou

* TD Groupe 1
  -le mercredi de 9h45 à 11h15 en M-0-1 Intervenant J-M Fédou
  -le vendredi de 9h45 à 11h15 en M-3-4, Intervenant J-M Fédou

* TD Groupes 2
  - le mercredi de 11h30 à 13h en M-0-1, Intervenant J. Coupechoux
  -le vendredi de 8h00 à 9h30 en M-3-4, Intervenant J. Coupechoux

Motivations

Mathématiques discrètes et Informatique

L'informatique ne manipule que des suites FINIES de 0 et de 1

* **Textes**. Une page A4 comporte environ 2500 caractères, chacun codé sur 8 bits en ISO 8859-1 (Latin1)

* **Nombres**. On a l'habitude de manipuler les nombres entiers sous leur forme décimale. Dans un ordinateur, ils sont codés en binaire. Si l'on regroupe les bits par paquets de 4, on obtient la représentation hexadécimale.

* **Images**. Une image standard contient aujourd'hui 8 millions de pixels

* **Sons**. Un morceau de musique de 6mn au format MPeg 3 nécessite 8 Mégaoctets

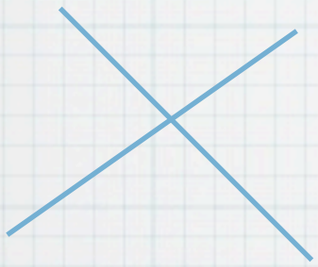
* **Vidéos**. Une vidéo de 6mn sur youtube nécessite 15 Mo. Chaque minute, plus de 24 h de vidéos sont postées sur youtube ...

* **Programmes**. Le programme qui a permis d'écrire ce document a une taille de 283,7 Mégaoctets

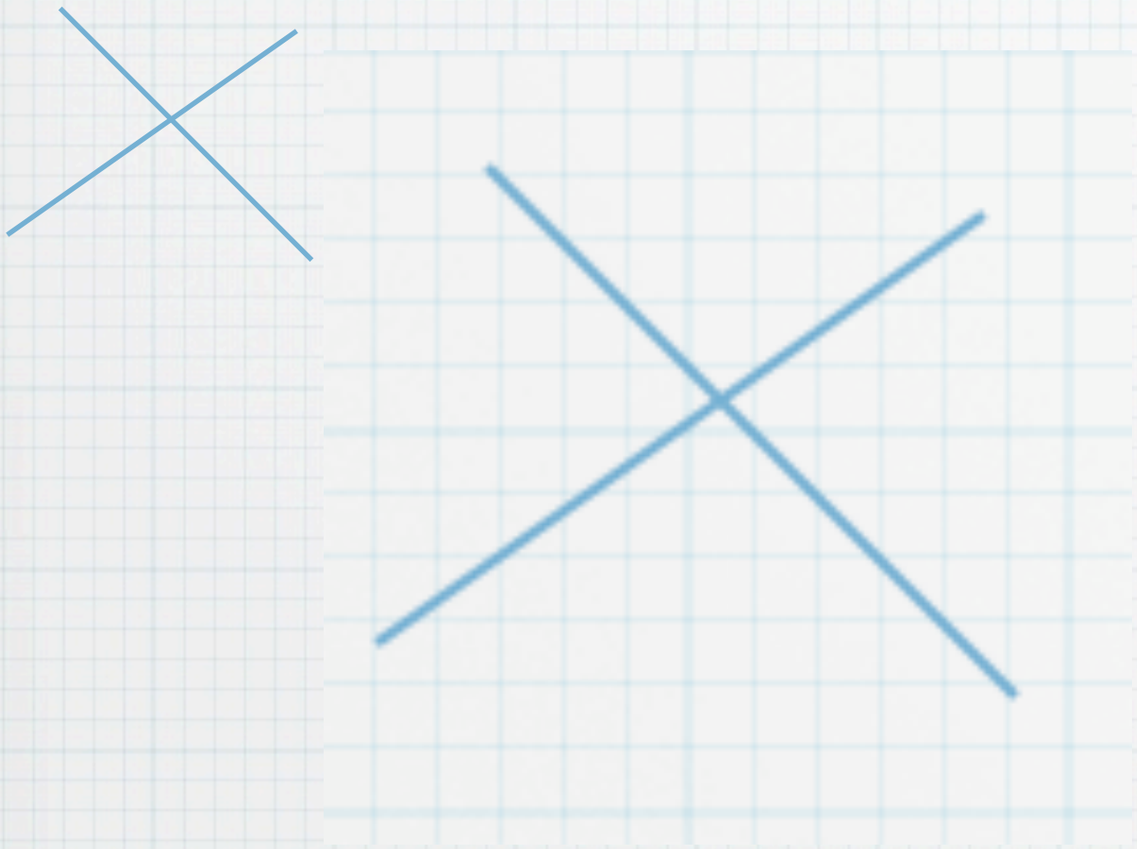
Motivations

Mathématiques discrètes et Informatique

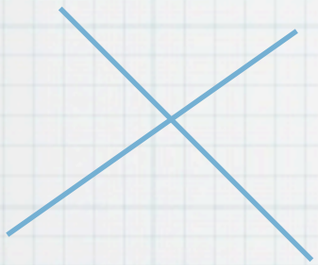
La géométrie de l'ordinateur n'est pas euclidienne

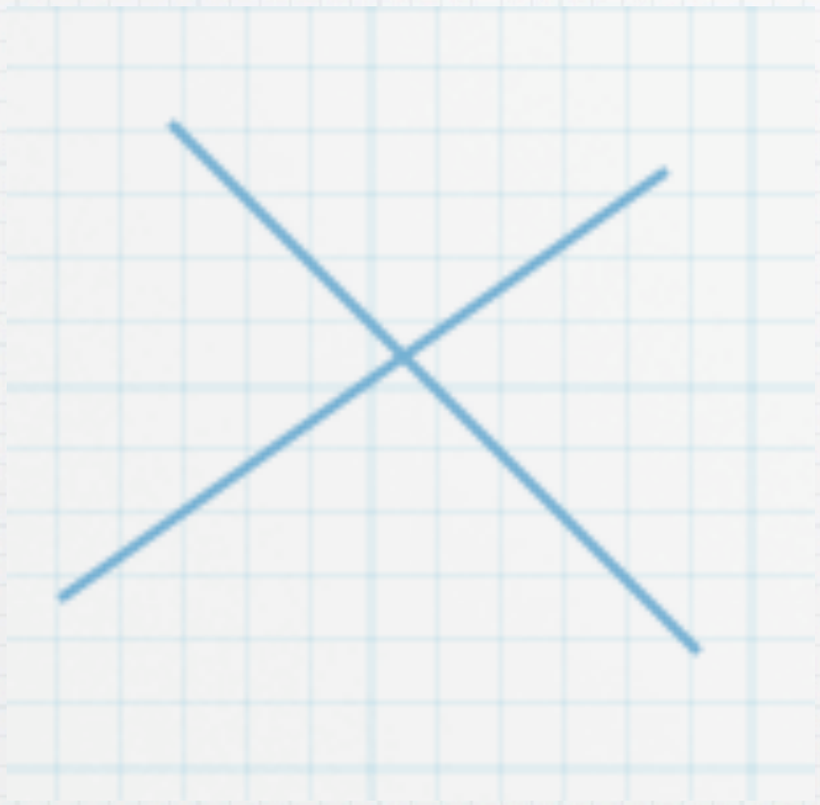


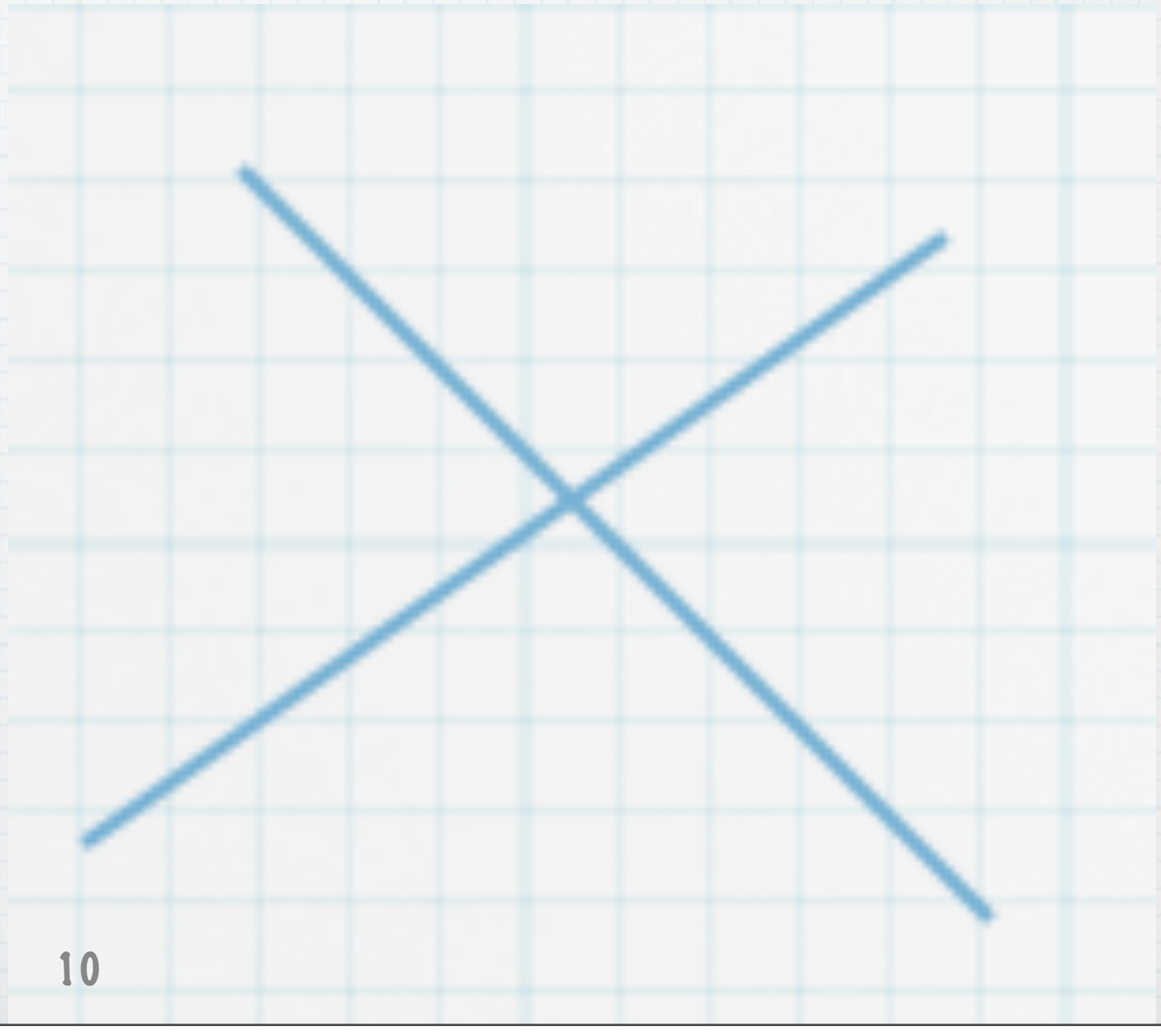
La géométrie de l'ordinateur n'est pas euclidienne



La géométrie de l'ordinateur n'est pas euclidienne



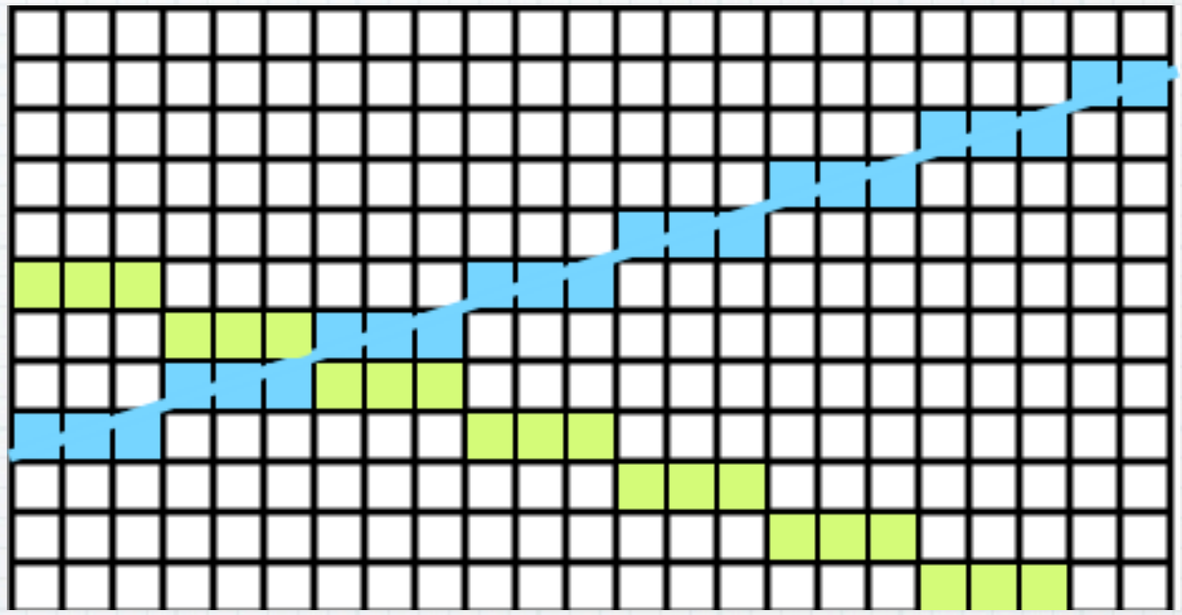




La géométrie informatique n'est pas euclidienne

Deux droites non parallèles peuvent

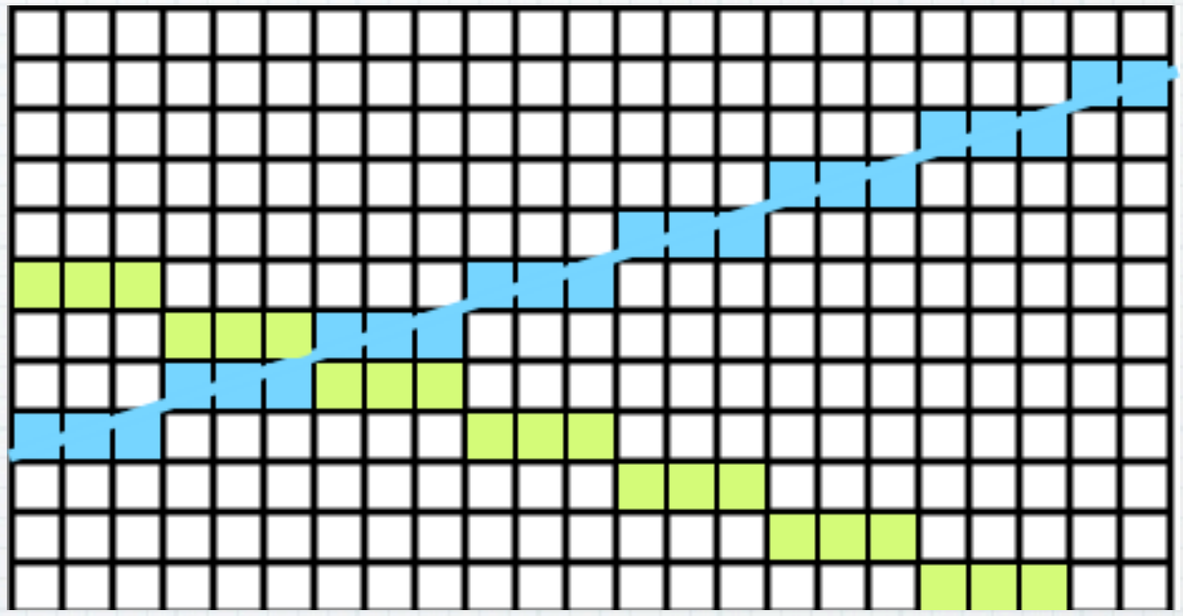
- n'avoir aucun points d'intersection



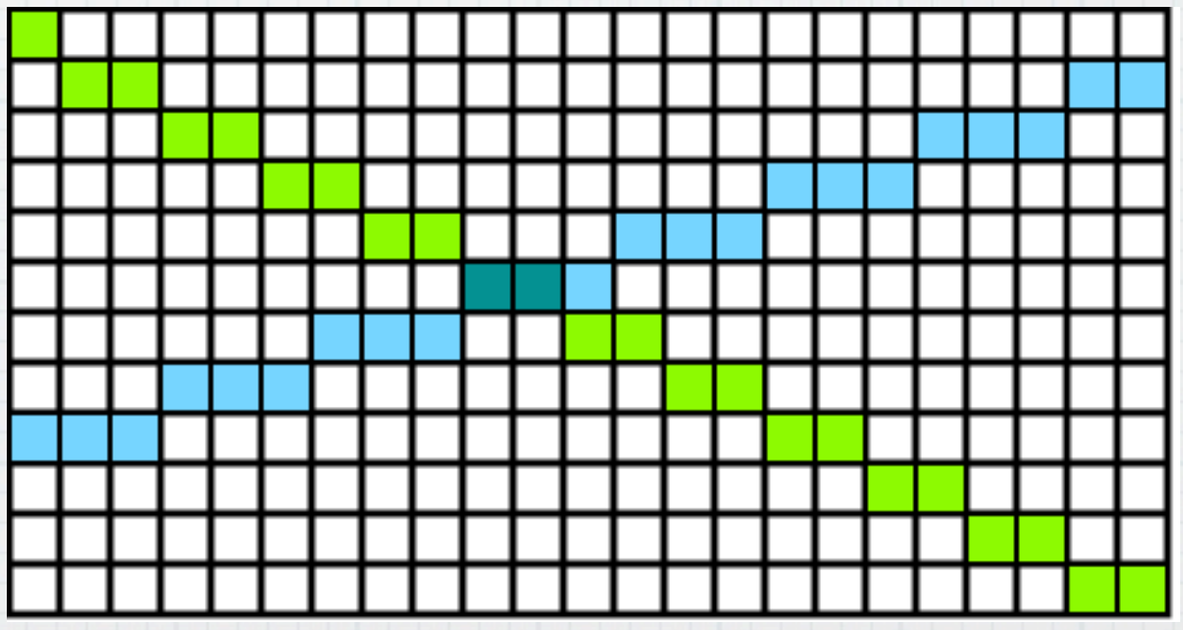
La géométrie informatique n'est pas euclidienne

Deux droites non parallèles peuvent

- n'avoir aucun points d'intersection



- ou peuvent en avoir plusieurs



Limites de l'informatique

* Certains problèmes n'ont pas de solutions, comme le problème de l'arrêt d'un programme.
* D'autres ont des solutions, mais algorithmiquement inefficaces (Sac à dos, Voyageur de commerce)
* Certains algorithmes sont meilleurs que d'autres

Langages

* Au début étaient les cartes perforées, l'assembleur et ce n'est que dans les années 50-60 que sont apparus des langages de programmation plus pratiques à utiliser

  * La théorie des langages est à la base des compilateurs ou interpréteurs que vous utilisez : analyse lexicale, analyse syntaxique, analyse sémantique ...

* La théorie des langages a des applications dans bien d'autres domaines

  * Cryptographie, codes

  * Compression

  * Biologie (Etude du génome)

Plan du cours

1. Complexité des algorithmes :
2. Ensembles et dénombrabilité :
3. Relations, fonctions et ordres :
4. Récurrence et induction :
5. Dénombrement :
6. Suites récurrentes :
7. Langages rationnels :
8. Automates finis :
9. Logique :
10. Graphes

1. Complexité des algorithmes

Motivation

* Un algorithme est une procédure finie et mécanique de résolution d'un problème.
  * Exemples : les algorithmes d'Euclide, l'algorithme de Dijkstra ...
* Un algorithme doit se terminer sur toutes les données possibles du problème et doit fournir une solution correcte dans chaque cas.
* Pour résoudre informatiquement un problème donné, on implante donc un algorithme sur un ordinateur.

  * Mais, pour un problème donné, il existe bien souvent plusieurs algorithmes.
  * Y a-t-il un intérêt à choisir ? et si oui comment choisir ?

* En pratique, il n'est même pas suffisant de détenir un algorithme.

  * Il existe des problèmes pour lesquels on a des algorithmes, mais qui restent « informatiquement non résolus ».
  * C'est parce que les temps d'exécution sont vite exorbitants.
  * On cherche alors des heuristiques pour abaisser ces temps de calcul.

Exemples d'algorithmes

Multiplication d'entiers

* Algorithme classique

```
      235142
   x    8712
  ---------
     470284
    235142
   1645994
  1881136
  ---------
  2048557104
```

Multiplication d'entiers

* Algorithme «à la russe»

~~235142~~	~~8712~~
~~470284~~	~~4356~~
~~940568~~	~~2178~~
1881136	1089
~~3762272~~	~~544~~
~~7524544~~	~~272~~
~~15049088~~	~~136~~
~~30098176~~	~~68~~
~~60119635 2~~	~~34~~
120392704	17
~~240785408~~	~~8~~
~~481570816~~	~~4~~
~~963141632~~	~~2~~
1926283264	1
2048557104	

Calcul du PGCD

* Décomposition en nombres premiers
  * p = 235142 = 2x...
  * q = 8712 = $2^3 \times 3^2 \times 11$

* Algorithme d'Euclide

	26	1	105	4	
235142	8712	8630	82	20	2
8630	82	20	2	0	

Ecriture d'un nombre en binaire

255	2
1	127
	1

127	2
63	
1	

63	2
31	
1	

31	2
15	
1	

15	2
7	
1	

7	2
3	
1	

3	2
1	
1	

1	2
0	
1	

L'écriture binaire du nombre entier 255 est 11111111

Complexité

* Le mot **complexité** recouvre en fait deux réalités :

  * la **complexité des algorithmes**. C'est l'étude de l'efficacité comparée des algorithmes. On mesure ainsi le temps et aussi l'espace nécessaire à un algorithme pour résoudre un problème.
    Cela peut se faire de façon expérimentale ou formelle.

  * la **complexité des problèmes**

* La complexité des algorithmes a abouti à une classification des problèmes en fonction des performances des meilleurs algorithmes connus qui les résolvent.

* Techniquement, les ordinateurs progressent de jour en jour. Cela ne change rien à la classification précédente. Elle a été conçue indépendamment des caractéristiques techniques des ordinateurs.

* Les domaines alliés de la complexité et de la preuve de programmes utilisent toutes les notions que nous verrons dans ce cours.

Temps d'exécution d'un programme

* On implante un algorithme dans un langage de haut niveau pour résoudre un problème donné.
* Le temps d'exécution du programme dépend alors :
  * des données du problème pour cette exécution
  * de la qualité du code engendré par le compilateur
  * de la nature et de la rapidité des instructions offertes par l'ordinateur (facteur 1 à 1000)
  * de l'efficacité de l'algorithme
  * ... et aussi de la qualité de la programmation !
* A priori, on ne peut pas mesurer le temps de calcul sur toutes les entrées possibles. Il faut trouver une autre méthode d'évaluation. L'idée est de s'affranchir des considérations subjectives (programmeur, matériel...). Pour cela, on utilise une grandeur n pour « quantifier » les entrées et on calcule les performances théoriques uniquement en fonction de n.

Taille des entrées

* En premier lieu, il faut évaluer la taille des données nécessaires à l'algorithme. On les appelle les **entrées** ou les **instances**. La taille n va dépendre du codage de ces entrées
  Par exemple, en binaire, il faut $\log_2(n)$ bits pour coder l'entier n.

* En pratique, on choisira comme taille la ou les dimensions les plus significatives. Voici quelques exemples, selon que le problème est modélisé par :

  * des **polynômes** : le degré, le nombre de coefficients ≠ 0

  * des **matrices** m x n : max (m,n), m.n, m+n

  * des **graphes** : nombre de sommets, d'arêtes, produit des deux

  * des **arbres** : comme les graphes et la hauteur, l'arité

  * des **tableaux**, des **fichiers** : nombre de cases, d'éléments.

  * des **mots** : leur longueur

* C'est ici que l'on s'aperçoit que le choix de telle ou telle structure de données n'est pas anodin quant à l'efficacité d'un algorithme.

Repérer les opérations fondamentales

* C'est la nature du problème qui fait que certaines opérations deviennent plus fondamentales que d'autres dans un algorithme.
* Leur nombre intervient alors principalement dans l'étude de la complexité de l'algorithme.
* Avec un peu d'habitude, on les repère :

Recherche d'un élément dans une liste, un tableau, un arbre ...	comparaisons
Tri d'un tableau, d'un fichier...	comparaisons déplacements
Multiplication de polynômes, de matrices, de grands entiers	additions multiplications

Coût des opérations

* Dans un ordinateur , toutes les opérations n'ont pas le même coût.
    * Exemple : multiplication et addition.
    * C'est peu étonnant. La mutiplication est bien une généralisation de l'addition. Pourquoi coûterait-elle pareil ?
* Seules les opérations élémentaires sont à égalité de coût.
* Pour simplifier, on peut faire l'hypothèse que toutes les opérations ont un coût uniforme (même si cela manque souvent de réalisme). Ce coût est alors constant car il ne dépend plus de rien.
* Pour un programme donné, les profileurs permettent de savoir quelles sont les instructions qui prennent le plus de temps (règle des 90-10).

Evaluation des coûts en séquentiel

* Dans un programme strictement séquentiel,les « boucles » sont disjointes ou emboîtées. Ceci exclut toute récursivité.

* Les temps d'exécution s'additionnent :

  * choix : (si C alors J sinon K)

$$T(n) = TC(n) + \max \{ TJ(n), TK(n) \}$$

  * itération bornée
    (pour i de j à k faire B)
    où Tentête est mis pour l'affectation de l'indice de boucle et le test de continuation.

$$T(n) = (k-j+1).(Tentête(n) + TB(n)))$$

* Le cas des boucles imbriquées se déduit de cette formule.

  * itération non bornée

    * (tant que C faire B)

    * (répéter B jusqu'à C)

$$T(n) = \#boucles . (TB(n) + TC(n)) + TC(n)$$

$$T(n) = \#boucles . (TB(n) + TC(n))$$

Le nombre de boucles #boucles s'évalue inductivement. appel de procédures : il n'y a pas d'appel récursif on peut ordonner les procédures de sorte que la ième ait sa complexité qui ne dépend que des procédures j, avec j<i.

Evaluation des coûts en récursif

* On sait que les algorithmes du type « diviser pour régner » donnent lieu à des programmes récursifs.
* Comment évaluer leur coût ? Il faut trouver :
  * la relation de récurrence associée
  * la base de la récurrence, en examinant les cas d'arrêts de la récursion
  * et aussi la solution de cette équation
* Techniques utilisées : résolution des équations de récurrence ou de partition classique ou par les séries génératrices.
* Exemples
  * Le temps T(n) de la recherche dichotomique dans un tableau de n cases est
    $T(n) = 1 + \log_2(n)$
  * La complexité en temps pour résoudre le problème des tours de Hanoï est
    $T(n) = 2^n - 1$

Variantes du temps d'exécution

* Temps d'exécution dans le pire des cas $T_{pire}(n) = max_n \{ T(t), |t|=n \}$
  * Exemple : le cas le pire pour le tri rapide (quicksort) est quand le tableau est déjà trié, le cas le pire pour le tri par sélection est quand le tableau est trié dans l'ordre inverse.
* Temps d'exécution en moyenne $T_{moy}(n) = \Sigma_{|t|=n} p(t) \ . \ T(t)$
  avec P(n) une loi de probabilité sur les entrées. Fréquemment, on utilise la loi de probabilité uniforme. T(n) peut aussi être une valeur moyenne.
  $T_{moy\&unif}(n) = (1/\#n) \quad \Sigma_{|t|=n} T(t)$
* Temps d'exécution dans le meilleur des cas $T_{meilleur}(n) = min_n \{ T(t), |t|=n \}$

Estimation asymptotique

* En complexité, on ne considère que des fonctions positives de IN dans IR. On ne veut pas évaluer précisément les temps d'exécution (d'autant que ça dépend des machines...). On se contente donc d'estimations.

* On dit que f positive est asymptotiquement majorée (ou dominée) par g et on écrit

$f = O(g)$

quand il existe une constante c, c>0, telle que pour tout n assez grand, on a : $f(n) \leq c\, g(n)$

* On définit symétriquement la minoration de f par g : $f = \Omega(g)$.

* On dit que f est du même ordre de grandeur que g et on écrit

$f = \Theta(g)$ quand $f = O(g)$ et $g = O(f)$

quand il existe deux constantes c_1 et c_2, telles que pour un n assez grand, on a :

$c_1\, g(n) \leq f(n) \leq c_2\, g(n)$

Ordres de grandeurs

* du plus petit au plus grand :

$O(1)$	$O(\log(n))$	$O(n)$	$O(n.\log(n))$	$O(n^2)$	$O(n^3)$	...	$O(2^n)$	$O(n!)$

* Attention, la notation $f = O(g)$ ne dénote pas une égalité mais plutôt l'appartenance de f à la classe des fonctions en $O(g)$.
  * Exemple : $4n^2+n = O(n^2)$ et $n^2-3 = O(n^2)$ sans que pour n assez grand on ait $4n^2+n = n^2-3$

* un algorithme est dit **polynomial** s'il existe un entier p tel que son coût est $O(n^p)$ avec $p>1$.

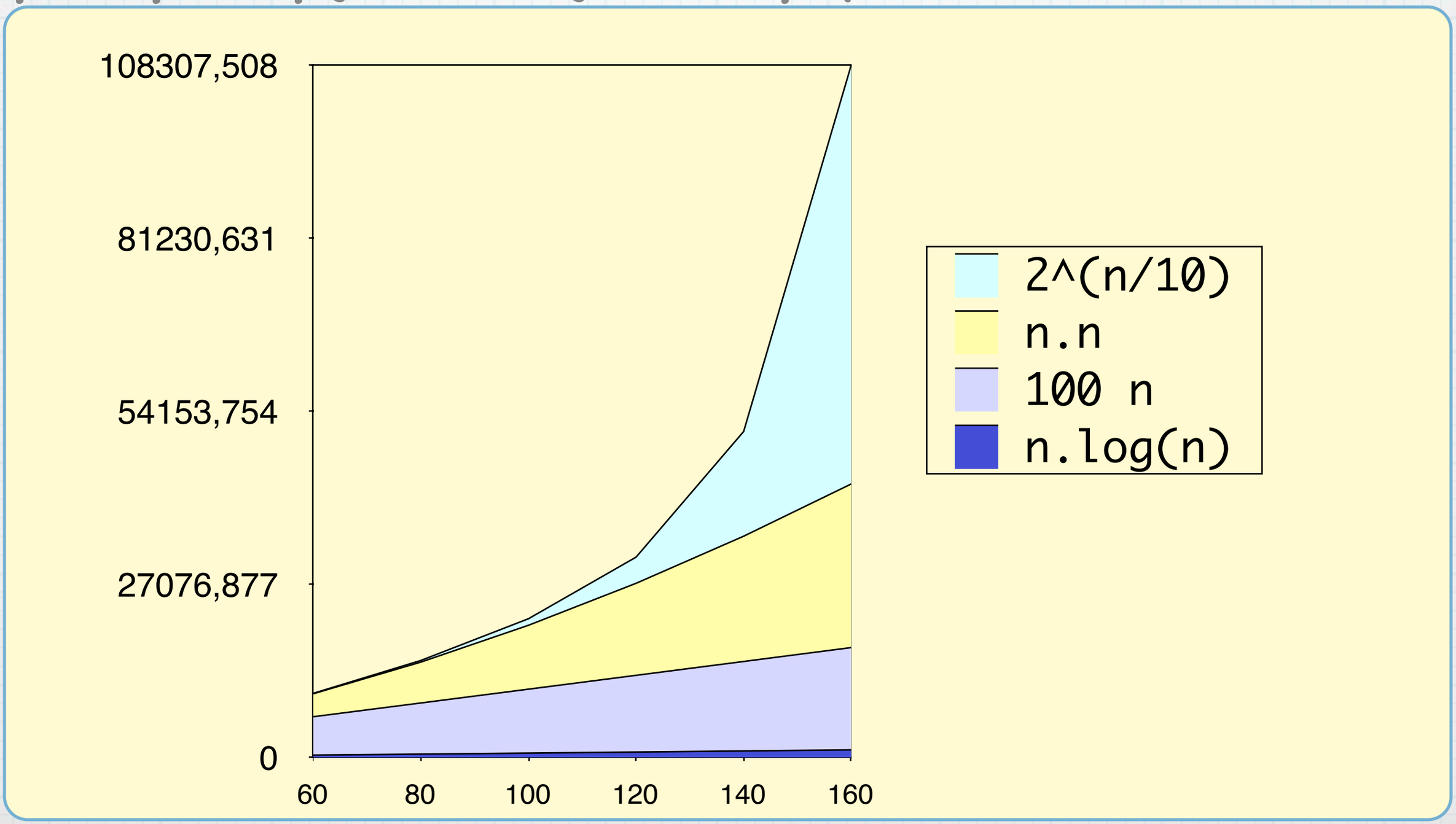
Ordres de grandeurs

$O(1)$	constant
$O(\log(n))$	logarithmique
$O(n)$	linéaire
$O(n \log(n))$	n log(n)
$O(n^2)$	quadratique
$O(n^3)$	cubique
$O(2^n)$	exponentiel

Commentaires

En pratique,

* un algorithme à une complexité exponentielle est inutilisable.
* pour n pas trop grand, les algorithmes polynomiaux sont encore efficaces.



Commentaires

* Si on considère qu'une instruction est réalisée en 10^{-8}s, la taille des données que peut traiter un algorithme en un temps donné est fourni par le tableau suivant

temps	O(1)	O(log(n))	O(n)	O(n.log(n))	$O(n^2)$	$O(n^3)$	...	$O(2^n)$	O(n!)
1 sec									
1 mn					80000				
1h									
1 mois									

Commentaires

* Si on considère qu'une instruction est réalisée en 10^{-8}s, le temps necessaire au traitement d'une donnée de taille n est fourni par le tableau suivant

Taille	O(1)	O(log(n))	O(n)	O(n.log(n))	$O(n^2)$	$O(n^3)$	...	$O(2^n)$	O(n!)
10									
100									
1000									
10 000									
1000000									

Algorithme de recherche d'un élément dans un tableau

```
int trouve(int x, int[] t) {
  for(int i=0, i< t.length, i++)
  {
    if (t[i] == x) return i;
  }
  return -1;
}
```

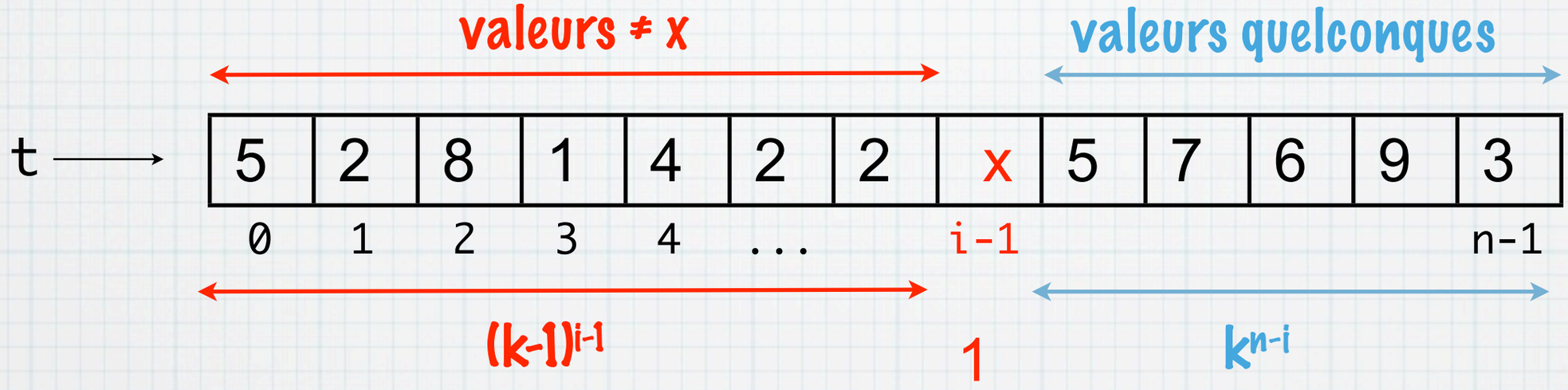
* Complexité dans le meilleur des cas O(1)
* Complexité dans le pire des cas O(n)
* Complexité en moyenne

* On suppose que les entrées du tableau sont des entiers choisis aléatoirement entre 1 et k. Chaque entier a une probabilité 1/k d'apparaître
* Il y a donc k^n tableaux de longueur n, chacun apparaissant avec une probabilité $1/k^n$

$$C_{moy}(n) = \Sigma_{|t|=n}\ p(t)\ .\ C(t) = \Sigma_{|t|=n}\ C(t)/k^n$$

Algorithme de recherche d'un élément dans un tableau

* L'algorithme a une complexité égale à **i** chaque fois que



* Et la complexité de l'algorithme est égale à **n** lorsque x n'est pas élément du tableau

* On en déduit que

$$T_{moy}(n) = \frac{n\,(k-1)^n + \sum_{i=1..n} i\,(k-1)^{i-1}\,k^{n-i}}{k^n} = k\,[1-(1-1/k)^n]$$

Exemples d'algorithmes

Otaku, Cedric's weblog

* Here is an interesting coding challenge: write a counter function that counts from 1 to max but only returns numbers whose digits don't repeat.

* For example, part of the output would be:
  - 8, 9, 10, 12 (11 is not valid)
  - 98, 102, 103 (99, 100 and 101 are not valid)
  - 5432, 5436, 5437 (5433, 5434 and 5435 are not valid)

* Also :
  - Display the biggest jump (in the sequences above, it's 4: 98 -> 102)
  - Display the total count of numbers
  - Give these two values for max=10000

* Post your solution in the comments or on your blog. All languages and all approaches (brute force or not) are welcome.

* I'm curious to see if somebody can come up with a clever solution (I don't have one)...

* Solutions disponibles à l'adresse http://beust.com/weblog/archives/000493.html

Challenge

* Ecrire un programme qui
  * calcule, dans l'ordre, les entiers n'ayant pas de chiffres répétés dans leur écriture décimale

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100
101	102	103	104	105	106	107	108	109	110	111	112	113	114	115	116	117	118	119	120
121	122	123	124	125	126	127	128	129	130	131	132	133	134	135	136	137	138	139	140
141	142	143	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159	160
161	162	163	164	165	166	167	168	169	170	171	172	173	174	175	176	177	178	179	180
181	182	183	184	185	186	187	188	189	190	191	192	193	194	195	196	197	198	199	200

  * Le nombre de ces entiers (le plus grand est 9876543210)
  * et qui retourne aussi la plus grande différence entre deux entiers consécutifs de la liste

Solutions concises

* Ruby

```
(98..103).select { |x| x.to_s.size == x.to_s.split(//).uniq.size }
```

* Python

```
(i for i in xrange(start, end) if len(str(i)) == len(set(str(i)))):
```

* Scala

```
for (i <- 1 to 100000; s = i.toString; if HashSet[Char](s:_*).size ==s.size) println(i)
```

* J ... oui c'est un vrai langage !

```
f =: [:(#;[:>./2-~/\])(#~([:*/[:~:":)"0)
```

Solutions concises

27 heures

* Ruby

```
(98..103).select { |x| x.to_s.size == x.to_s.split(//).uniq.size }
```

* Python

```
(i for i in xrange(start, end) if len(str(i)) == len(set(str(i)))):
```

* Scala

```
for (i <- 1 to 100000; s = i.toString; if HashSet[Char](s:_*).size ==s.size) println(i)
```

* J ... oui c'est un vrai langage !

```
f =: [:(#;[:>./2-~/\])(#~([:*/[:~:":)"0)
```

Une solution concise et efficace

```
public class BeustSequence {

 public static void findAll(long max, Listener listener) {
   for (int length = 1; length < 11; length++) {
     if (find(1, length, 1, 0, max, listener)) return;
   }
  }
   * @param used digit bitfield
   * @param max value
   * @param listener hears results
   * @return true if we reached max, false otherwise
   */
}
```

```
private static boolean find(int first, int remaining, long value,
    int used, long max, Listener listener) {
        for (int digit = first; digit < 10; digit++, value++) {
            int mask = 1 << digit;
            if ((used & mask) == 0) {
            if (remaining == 1) {
                if (value > max) return true;
                listener.hear(value);
          } else
        if (find(0, remaining - 1, value * 10, used | mask, max,
            listener)) {
          return true;
        } } }
      return false;
    }}
```

Une solution concise et efficace

```
public class BeustSequence {

 public static void findAll(long max, Listener listener) {
   for (int length = 1; length < 11; length++) {
     if (find(1, length, 1, 0, max, listener)) return;
   }
 }

   * @param used digit bitfield
   * @param max value
   * @param listener hears results
   * @return true if we reached max, false otherwise
   */
}
```

34 ms

```
private static boolean find(int first, int remaining, long value,
    int used, long max, Listener listener) {
      for (int digit = first; digit < 10; digit++, value++) {
        int mask = 1 << digit;
        if ((used & mask) == 0) {
        if (remaining == 1) {
        if (value > max) return true;
          listener.hear(value);
      } else
    if (find(0, remaining - 1, value * 10, used | mask, max,
        listener)) {
      return true;
    } } }
  return false;
}}
```