

Programmation et IA

Cours 1

Jean-Jacques Lévy

`jean-jacques.levy@inria.fr`

`http://jeanjacqueslevy.net/prog-ia-25`

Plan

- machines informatiques
- systèmes informatiques
- programmes
- langages de programmation
- la programmation
- un langage pour le cours: Python 3
- premiers programmes

dès maintenant: **télécharger Python 3 en** `http://www.python.org`

les systèmes
informatiques

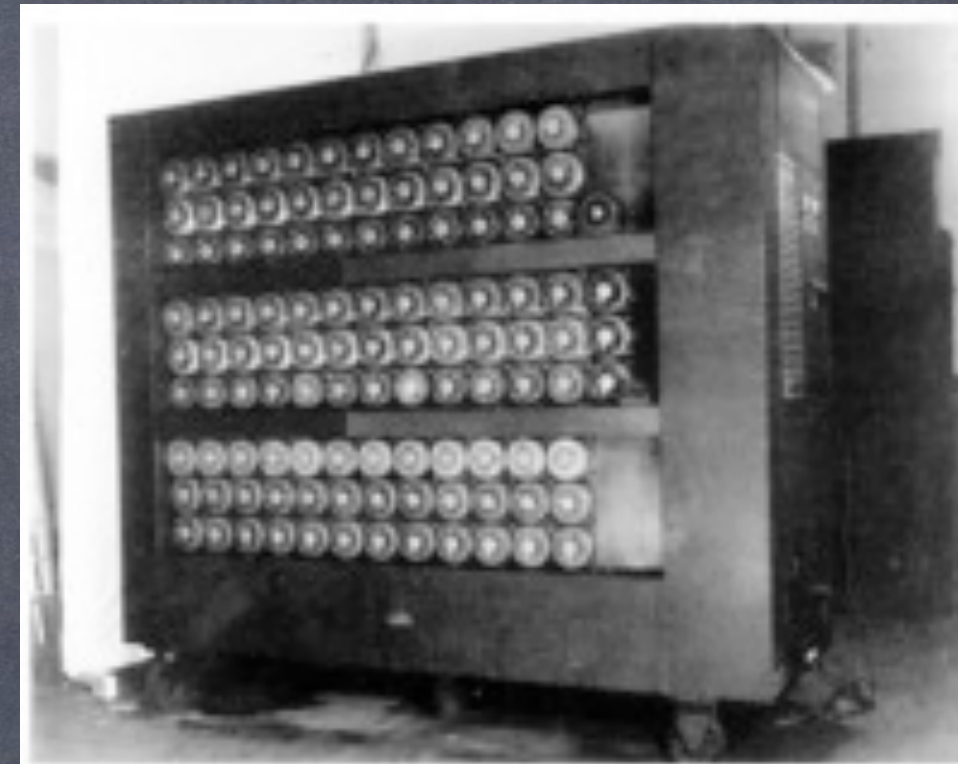
G
e
r
m
a
n
y



Enigma



Lorenz

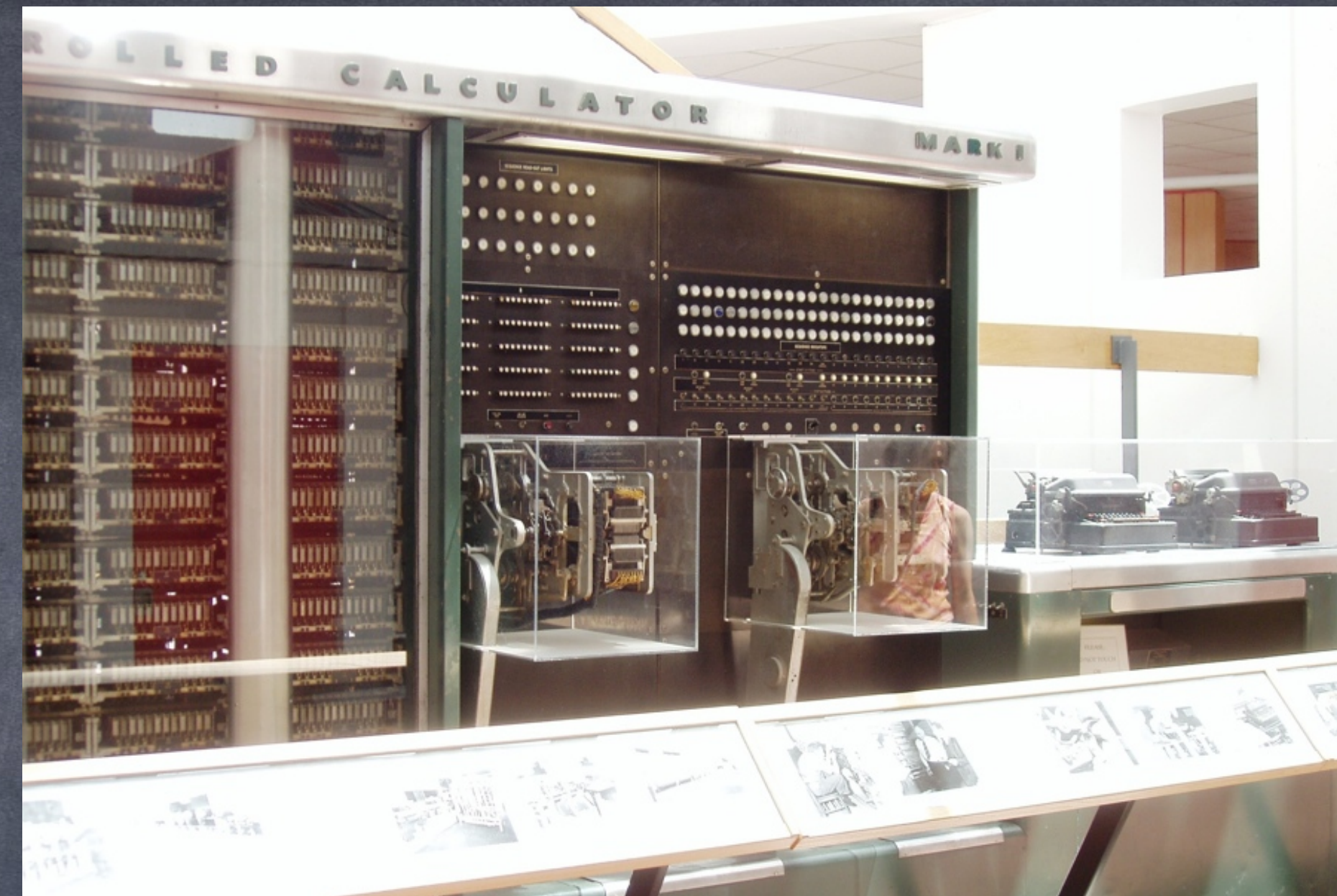
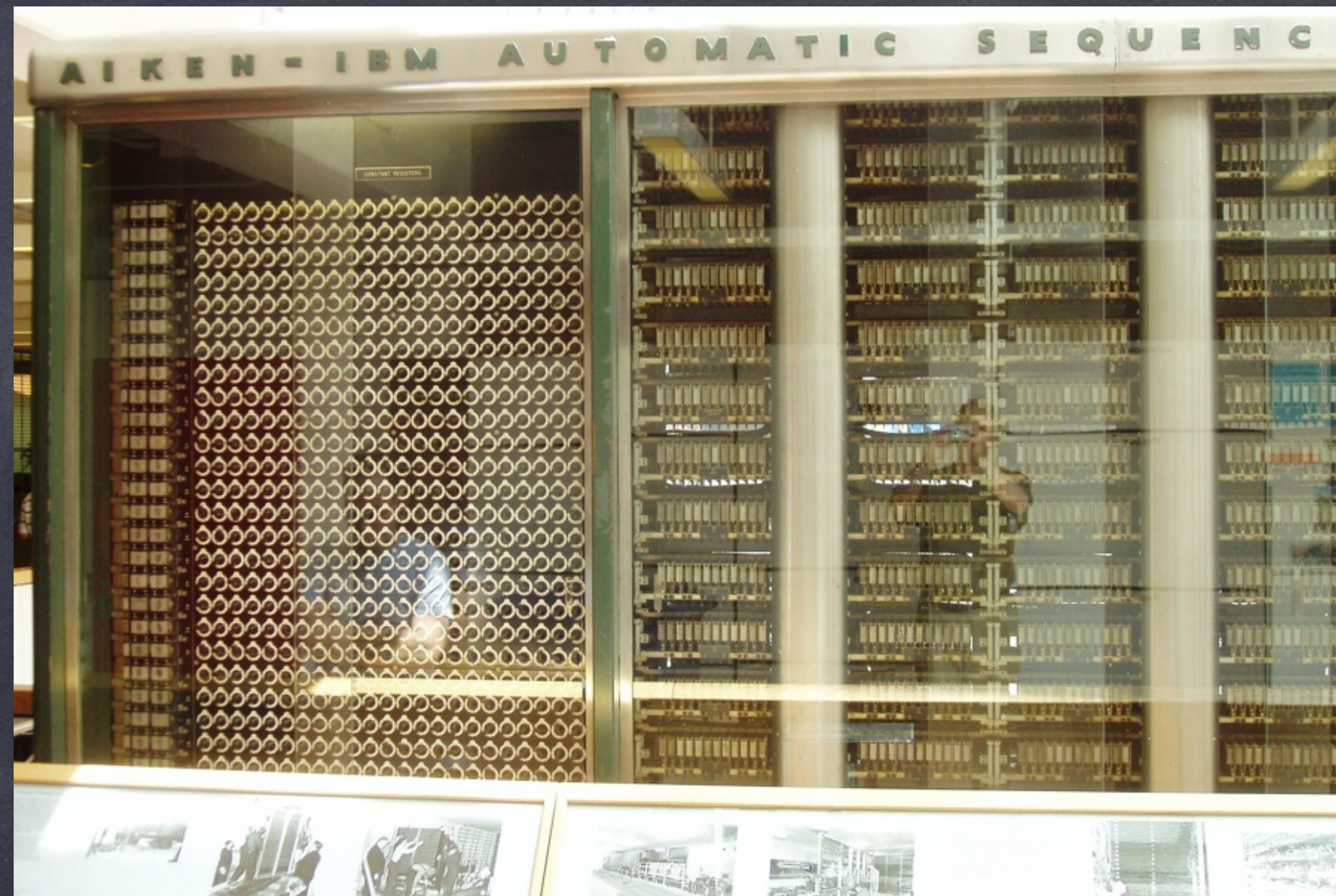


The Bombe

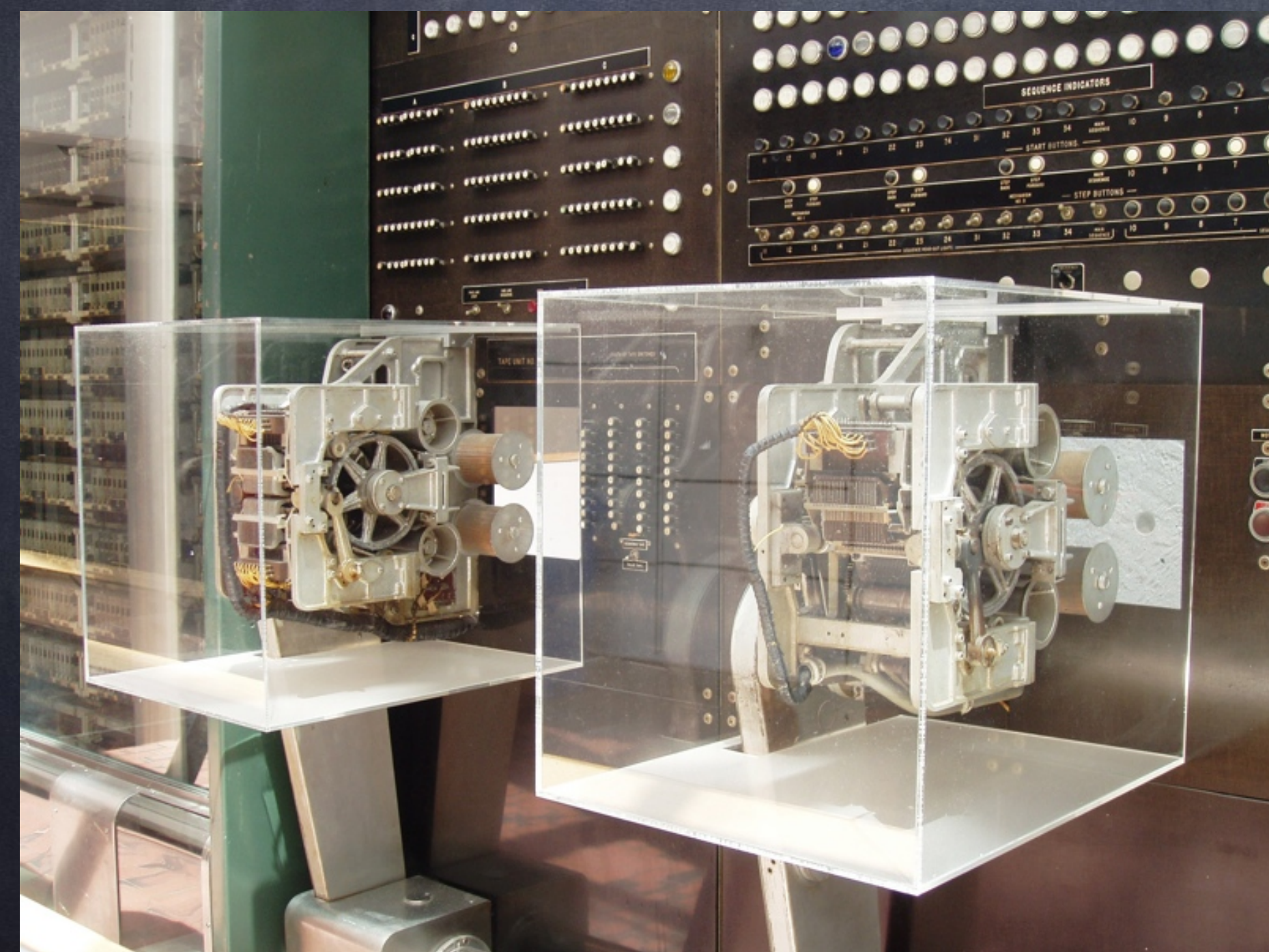


Colossus

B
l
e
t
c
h
l
e
y
P
a
r
k



Mark I, Harvard



Mark II,
Harvard





DEC vax 11/780



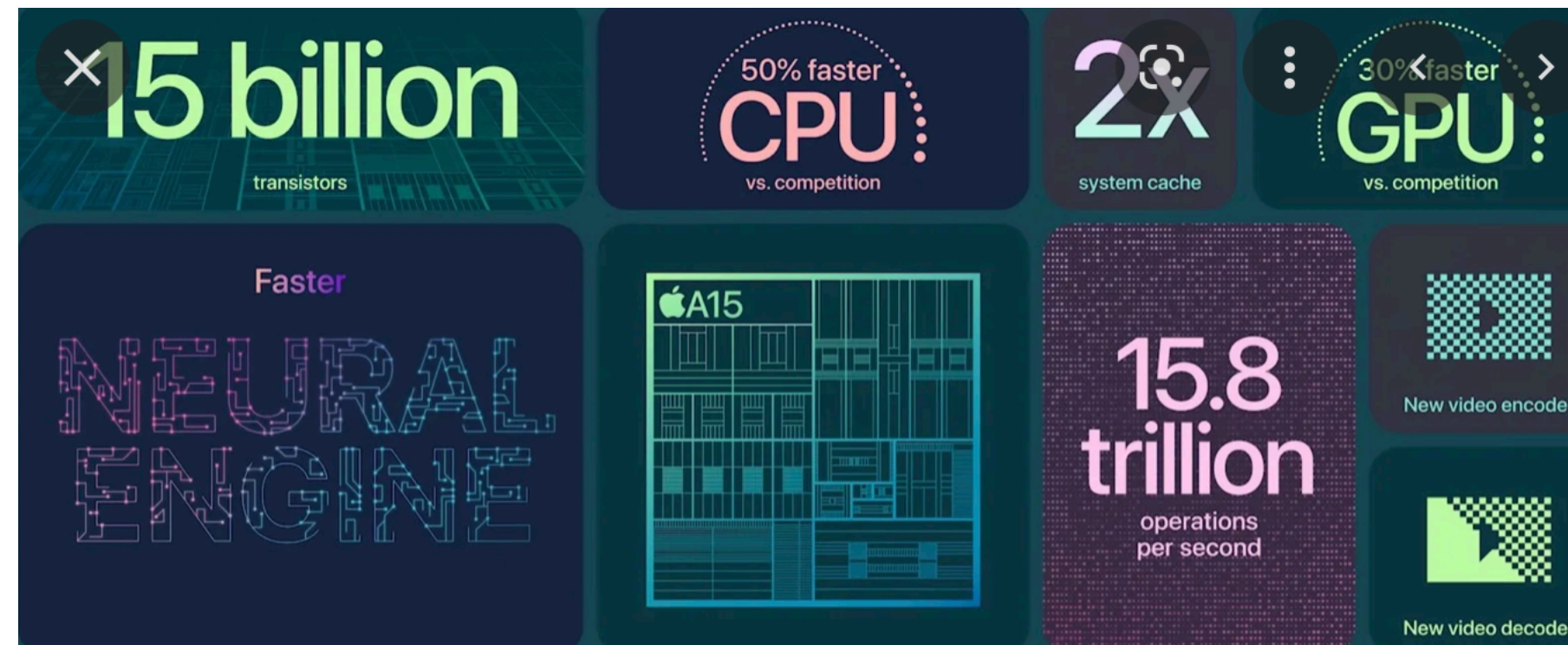
CII iris 80



hp2621 et colorix

Machines informatiques

- les ordinateurs modernes sont miniaturisés — micro processeurs (*chips*)



- 15 milliards de transistors avec un canal à 3nm (10^{-9})

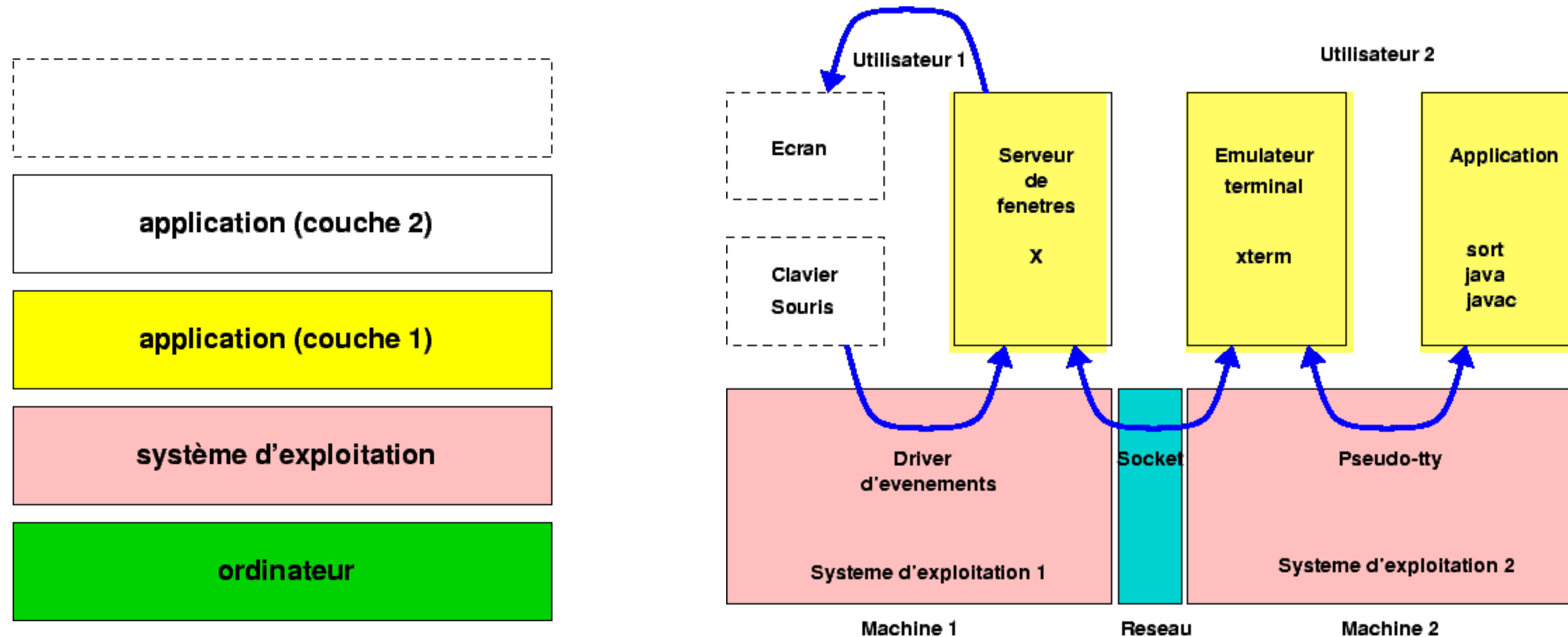


iPhone 14

macbook 2022

Systemes informatiques

- Windows, MacOS, Unix, Linux, ios, android, GFS, ...
- faire marcher l'ordinateur avec ses périphériques (écran, disques, clavier, souris, utilisateurs, réseau, ...)
- les OS sont d'une complexité extrême (~100 millions de lignes de code)



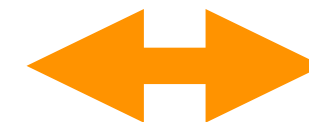
*OS = operating system
= système d'exploitation*

- au 15ème siècle, on construisait des cathédrales
- au 20ème siècle, on a construit des systèmes informatiques

Machines informatiques

- les premiers vrais ordinateurs en 1960: IBM 704, 7040, 360/370, ...
- le langage des machines est exprimé en **binaire**: 0 ou 1 [pour des raisons électriques]

```
mac$ od -x a.out | head -4
00000000      0162      0006      8f30      2e89      6000      0000      0060      0000
00000020      0038      0007      010b      020a      4000      0000      2000      0000
00000040      0000      0000      0140      0040      0000      0040      0000      1000
00000060      2000      1000      fffe      f3ff      0000      0000      1013      0000
mac$ od -x a.out | tail -5
01164000      0000      0028      0216      0000      3030      0040      1046      0000
01164020      0000      0023      021c      0000      1580      1000      f0c5      ffff
01164040      0000      0023      0220      0000      12e0      1000      f341      ffff
01164060      0000      0023      0228      0000      12e4      1000      f341      ffff
```



```
mac$ cat t.c
#include <stdio.h>

char s[100] = "voici une chaîne de caractères" ;

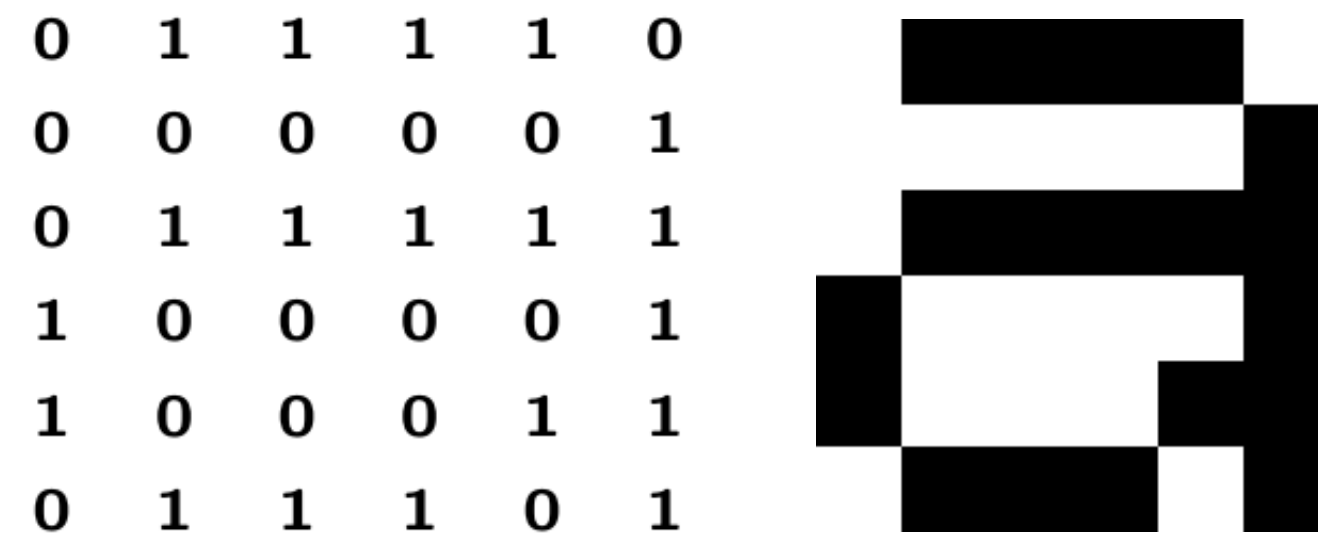
main()
{
    printf ("%s\n", s);
}
```

- ici 40256 octets = 322048 bits
- dans ce binaire, il y a des commandes [opérations à effectuer] et des données
- ce binaire est chargé dans la mémoire de l'ordinateur
- et l'ordinateur exécute les commandes du binaire
- chaque opération prend quelques micro-secondes

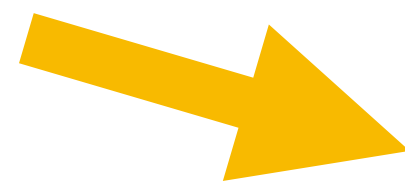
*bits = binary digit
octets = 8 bits = bytes*

Exemples de problèmes non triviaux

- afficher des lettres sur l'écran
- dessiner rapidement des figures sur l'écran (vecteurs, cercles, ovales, *splines*)
- écrire efficacement sur les disques
- ne pas perdre les messages envoyés ou reçus sur le réseau
- sauvegarder les données (*backups*)



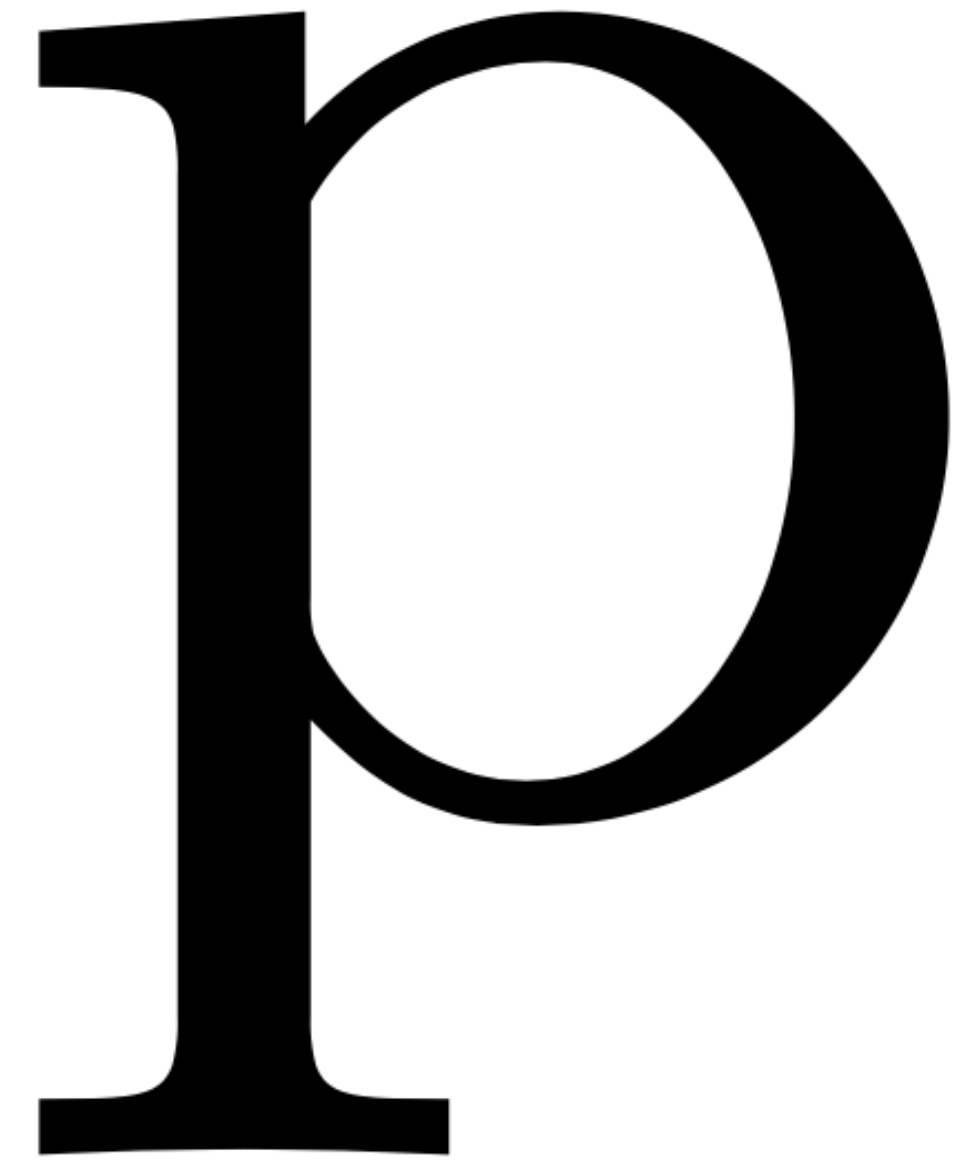
lettre 'a' = tableau de 0 et 1



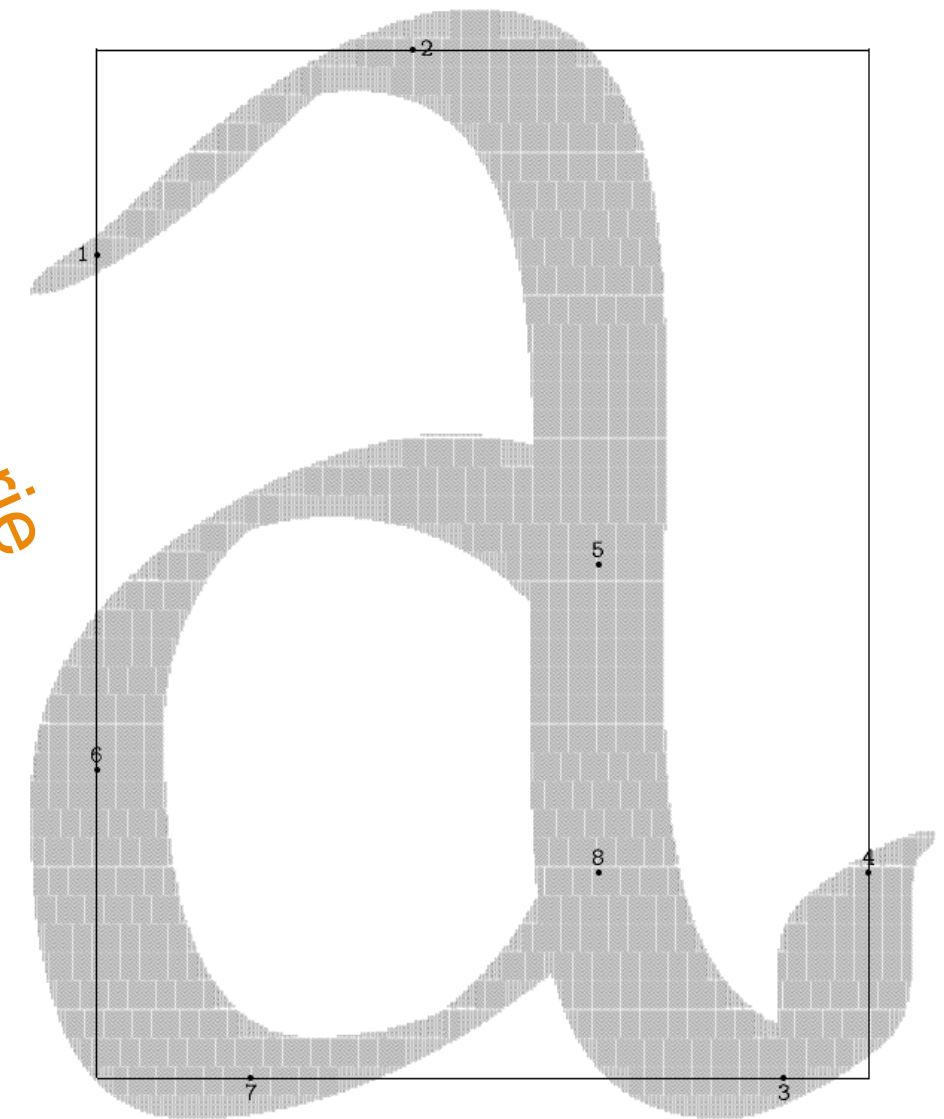
écran *bitmap*

écran = grand tableau de 0 et 1 (taille = *résolution* de l'écran)

metafont = réinvention de l'imprimerie



METAFONT output 2012.05.07:1643 Page 1 Character 97



Algorithmes

- algorithmes **linéaires** (en complexité $O(n)$)
 - recherche du maximum dans un tableau
- algorithmes **polynomiaux** (en complexité $O(n^2)$, $O(n^3)$, ...)
 - tri Bulle, tri sélection
- algorithmes **exponentiels** (en complexité $O(2^n)$, $O(2^{2^n})$, ...)
 - exploration exhaustive (les 8 reines)
- algorithmes sub-linéaires (en complexité $O(\log n)$)
 - recherche dichotomique en table
- algorithmes sub-quadratiques (en complexité $O(n \log n)$)
 - tri fusion

Algorithmes et programmation

- bien réfléchir avant de se lancer dans la programmation
- la **partie algorithmique** est souvent une petite partie de la programmation
- la partie principale de la programmation consiste en **l'organisation des différents modules**
- par exemple en **séparant** bien les interfaces (entrée-sortie) du corps du programme
- en général, on n'y arrive pas au premier coup —> **plusieurs versions**
- quand le programme grossit et aussi quand plusieurs auteurs, la **gestion des versions** est critique

Intelligence artificielle

- c'est principalement un problème de **statistiques**
- il y a une phase d'**apprentissage** avec un grand nombre de données
- une phase ultérieure d'optimisation pour **approximer** le résultat demandé
- l'IA bénéficie de la gestion maintenant possible d'un grand nombre de données (*big data*)
- et de **processeurs** de calcul ultra-performants (*GPU, neural engines, TPU, ..*)
- on verra les réseaux de neurones dans la 2ème partie du cours
- les **algorithmes** et les **structures de données** sont toujours indispensables
- il y a toujours un bel avenir pour les programmeurs!

Quelques rappels

- le cours utilise le langage Python et l'environnement Visual Studio Code. (`vscode`)
- et pour la partie IA, les bibliothèques Numpy, Tensorflow et Jax

programmation en Python

Python

- utiliser un système intégré (**Visual Studio** ou autre)
- ou utiliser une simple fenêtre **terminal** [le plus rapide pour démarrer]
- avec un éditeur de texte (Emacs, VI, TextEdit, ..)

- sur la fenêtre terminal, on tape:

```
mac$ python
Python 3.7.9 (default, Sep  6 2020, 13:20:25)
[Clang 11.0.3 (clang-1103.0.32.62)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

- et on peut fonctionner en mode calculette:

```
>>> 23 + 42
65

>>> 438 * 234
102492

>>> (438 * 234) + 35
102527

>>> ((438 * 234) + 35 / 3)
102503.66666666667
```


Débuts en Python

- mode calculette

```
>>> x = 45
>>> 3 * x + 2
137
```

- avec des chaînes de caractères

```
>>> print ("Bonjour les amis !")
Bonjour les amis !
>>> y = "Bonjour les amis"
>>> y + " et la famille"
'Bonjour les amis et la famille'
>>> z = " !"
>>> y + " et la famille" + z
'Bonjour les amis et la famille !'
```

```
>>> s = "Bonjour les amis !"
>>> s[0]
'B'
>>> s[1]
'o'
>>> s[2]
'n'
>>> s[9]
'e'
>>> s[-1]
'!'
>>> s[-2]
','
>>> s[-3]
's'
```


Débuts en Python

- définir des fonctions

```
>>> def double (x) :  
...     return (2 * x)  
...  
>>> double(3)  
6
```

```
>>> def concatene (s1, s2) :  
...     return (s1 + " " + s2)  
...  
>>> concatene ("Hello", "World !")  
'Hello World !'
```

Débuts en Python

- calcul du PGCD (algorithme d'Euclide) de deux nombres entiers m et n
- écrire la date de Pâques de 2021 à 2050
- calculer la suite de Syracuse. Vérifier son résultat jusqu'à 1000

$$u_{n+1} = \begin{cases} 3u_n + 1 & \text{si } u_n \text{ impair} \\ u_n/2 & \text{si } u_n \text{ pair} \end{cases}$$

Exercices du cours 1

- PGCD par l'algorithme d'Euclide

```
def pgcd (m, n) :  
    while m != n :  
        if m > n :  
            m = m - n  
        else:  
            n = n - m  
    return m
```

← suppose $m > 0$ et $n > 0$

- suite de Syracuse

```
def syracuse (n) :  
    while n != 1 :  
        print (n, end = ' ')  
        if n % 2 == 0 :  
            n = n // 2  
        else :  
            n = 3 * n + 1  
    print (n)
```

← impression sur la même ligne

```
>>> syracuse(19)  
19 58 29 88 44 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1
```

Listes

- les listes de Python sont des tableaux dynamiques modifiables

```
>>> x = [1, 3, 4, 2, 3, 5]
>>> type (x)
<class 'list'>
>>> x[0]
1
>>> x[1]
3
>>> x[3]
2
```

```
>>> x[3] = 99
>>> print (x)
[1, 3, 4, 99, 3, 5]
>>> len (x)
6
```

← modification du 4ème élément

← longueur de la liste

```
>>> x.append (9)
>>> x
[1, 3, 4, 2, 3, 5, 9]
```

- les listes de Python ne sont pas forcément homogènes

```
>>> y = [1, 3, 4, "kludge", 2, 3]
>>> print (y)
[1, 3, 4, 'kludge', 2, 3]
```

différent des listes de Lisp,
Ocaml, Haskell, Java, ...

Listes

- itération sur une liste (tableau)

```
>>> s = 0
>>> for m in x :
...     s = s + m
...
>>> s
27
```

← itération sur tous les éléments de x

```
>>> x
[1, 3, 4, 2, 3, 5, 9]
>>> max = -1
>>> for m in x :
...     if m > max :
...         max = m
...
>>> max
9
```

← itération sur tous les éléments de x

- idem avec une fonction

```
>>> def sum_of (x) :
...     r = 0
...     for m in x :
...         r = r + m
...     return r
...
>>> sum_of (x)
27
>>> a = [-3, 42, 23, 11, -30]
>>> sum_of (a)
43
```

Listes

- itération sur une liste (tableau)

```
>>> x
[1, 3, 4, 2, 3, 5, 9]
>>> max = -1
>>> for m in x :
...     if m > max :
...         max = m
...
>>> max
9
```

 itération sur tous les éléments de x

Exercice: valeur de `max_of ([ ])` ?

Exercice: écrire la fonction `min_of (x)`

Exercice: écrire la fonction `max_of ([ ])`
qui marche aussi sur tous les types ordonnés

Intervalle — Tranche — n-uplet

- intervalles (*range*)

```
>>> for i in range (0, 10):  
...     print (i)  
...  
0  
1  
2  
3  
4  
5  
6  
7  
8  
9
```

← **intervalle semi-ouvert** →

```
>>> for i in range (10):  
...     print (i)  
...  
0  
1  
2  
3  
4  
5  
6  
7  
8  
9
```

- abréviation et pas dans les intervalles

```
>>> range (10)  
range(0, 10)  
>>> for i in range (0, 10, 2) :  
...     print (i)  
...  
0  
2  
4  
6  
8
```

← **2 est le pas**

Exercice: quel est le sens de `range (10, 0, -1)` ?

Listes

- idem avec une fonction

```
>>> def max_of (x) :  
    r = x[0]  
    for m in x[1:] :  
        if m > r :  
            r = m  
    return r
```

```
>>> max_of (x)  
9  
>>> max_of (['jj', 'andre', 'paul'])  
'paul'
```

← si x est une liste non vide →

```
>>> def max_of (x) :  
    r = x[0]  
    for i in range (1, len(x)) :  
        if x[i] > r :  
            r = x[i]  
    return r
```

```
>>> max_of (x)  
9  
>>> max_of (['jj', 'andre', 'paul'])  
'paul'
```


Intervalle — Tranche — n-uplet

- tranche (*slice*)

```
>>> x
[1, 3, 4, 2, 3, 5, 9]
>>> x[3:6]
[2, 3, 5]
>>> x[3:6:2]
[2, 5]
>>> x[3:]
[2, 3, 5, 9]
>>> x[:6]
[1, 3, 4, 2, 3, 5]
>>> x[::2]
[1, 4, 3, 9]
```

 **intervalle semi-ouvert**

- un n-uplet (*tuple*) est une liste non modifiable

```
>>> b = (9, "novembre", 1989)
>>> b[0]
9
>>> b[1]
'novembre'
>>> b[2]
1989
```

- une chaîne de caractère est une liste de caractères non modifiables

```
>>> s = "abcdefghijklmnopq"
>>> s[3:10]
'defghij'
```

Valeur d'un tableau — Alias

Exercice: le programme suivant est-il correct ?

```
def copy (a, b, i, j, l) :  
    for k in range (l) :  
        b [j + k] = a [i + k]
```



on suppose :

$$0 \leq i$$

$$0 \leq j$$

$$0 \leq l$$

$$i + l < m = \text{len}(a)$$

$$j + l < n = \text{len}(b)$$

Valeur d'un tableau — Alias

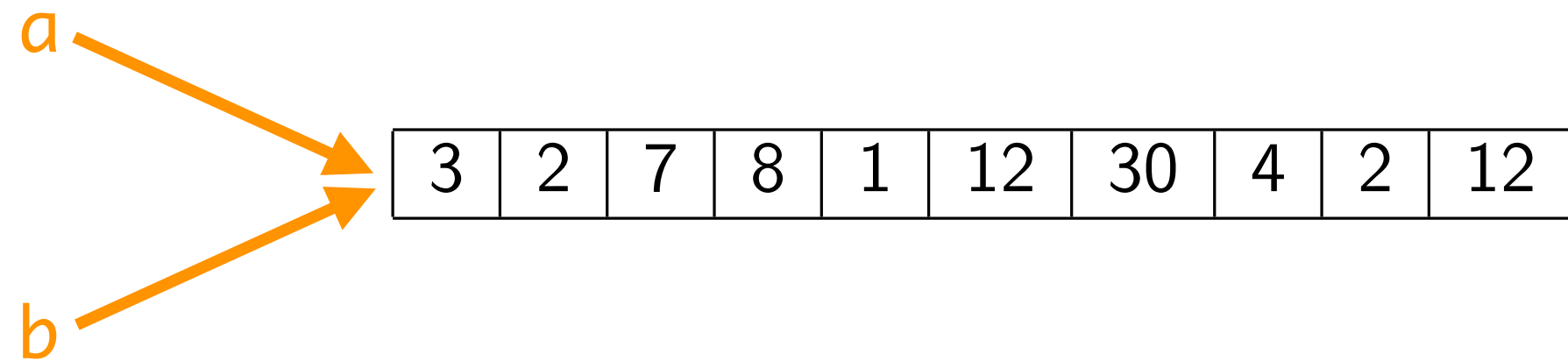
- soient 2 tableaux **a** et **b**

```
a = [3, 2, 7, 8, 1, 12, 30, 4, 2, 12]
b = a
```

→ **a**[2] = 888

```
print (a)
[3, 2, 888, 8, 1, 12, 30, 4, 2, 12]
```

```
print (b)
[3, 2, 888, 8, 1, 12, 30, 4, 2, 12]
```

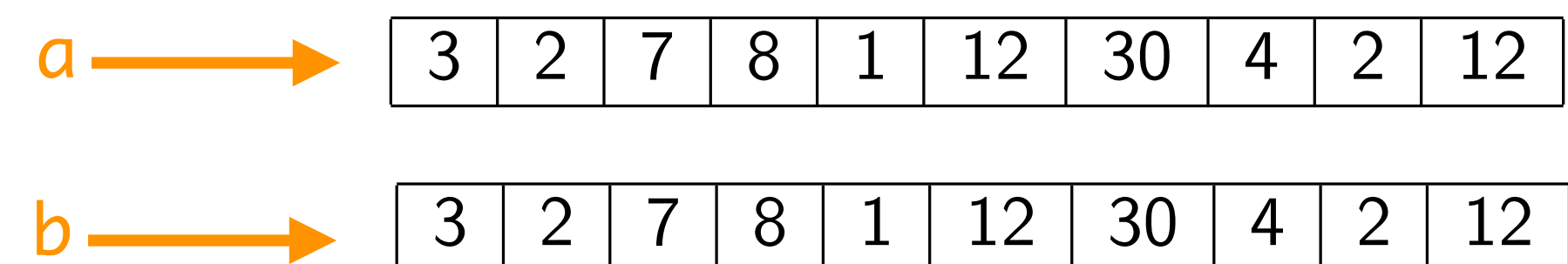


```
a = [3, 2, 7, 8, 1, 12, 30, 4, 2, 12]
b = [3, 2, 7, 8, 1, 12, 30, 4, 2, 12]
```

→ **a**[2] = 888

```
print (a)
[3, 2, 888, 8, 1, 12, 30, 4, 2, 12]
```

```
print (b)
['jj', 'andre', 'paul']
```



- les variables **a** et **b** sont 2 alias d'un même tableau

- les variables **a** et **b** sont 2 tableaux distincts

- la valeur de **a** ou de **b** est l'adresse mémoire de son premier élément

données modifiables et alias sont sources de bugs

tableaux
multi-
dimensionnels

Tableaux multi-dimensionnels

- une matrice est une liste de listes

```
>>> a = [[1,2], [3,4]]
```

```
>>> a[0][0]
```

```
1
```

```
>>> a[0][1]
```

```
2
```

```
>>> a[1][0]
```

```
3
```

```
>>> a[1][1]
```

```
4
```

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

	0		1	
a	1	2	3	4
	0	1	0	1

- une itération sur les matrices

```
def print_matrix (a) :  
    for line in a :  
        for elt in line :  
            print (elt, end = ' ' )  
        print ()
```

← impression sur une ligne

```
def print_matrix (a) :  
    for line in a :  
        for elt in line :  
            print ("%2d " %elt, end = ' ' )  
        print ()
```

Tri

- on cherche le minimum et on le met en tête..
et on recommence à partir du deuxième élément, etc...

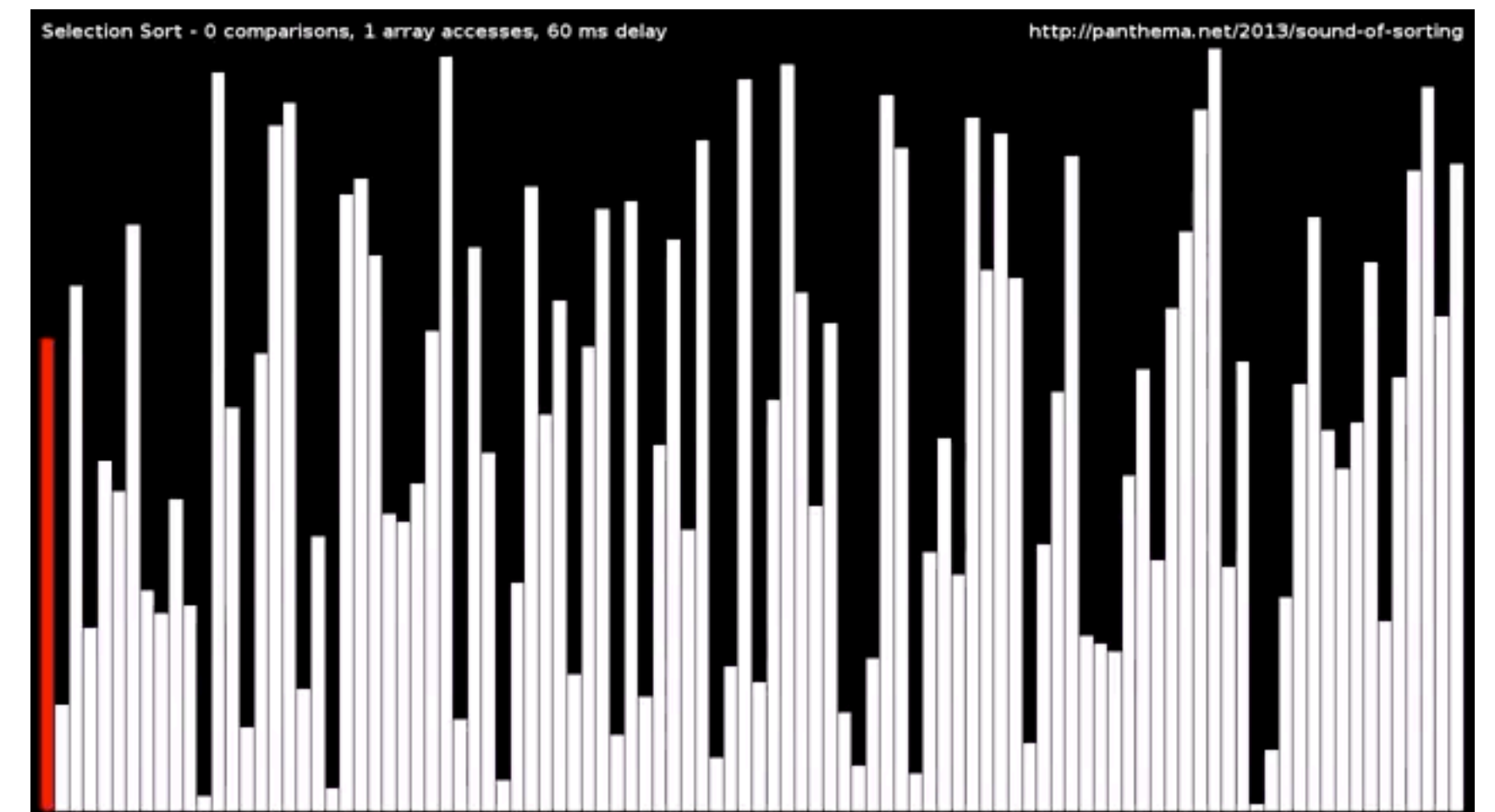
```
def tri_selection (a) :  
    n = len (a)  
    for i in range (n-1) :  
        jmin = i;  
        for j in range(i+1, n) :  
            if a[j] < a[jmin] :  
                jmin = j  
        t = a[i]; a[i] = a[jmin]; a[jmin] = t
```

← échanger les valeurs de $a[i]$ et $a[jmin]$

← on suppose $\text{len}(a) > 0$

```
def index_min_of (a) :  
    imin = 0  
    for i in range (1, len(a)):  
        if a[i] < a[imin] :  
            imin = i  
    return imin
```

<http://visualgo.net/en/sorting>



Ensembles — Compréhension

- les ensembles sont des listes non ordonnées d'éléments tous distincts

```
print ({10, 2, 3} == {3, 2, 10})  
→ True
```

```
print ({10, 2, 3} == {5, 2, 10})  
→ False
```

- on peut générer des tableaux, listes, ensembles, dictionnaires avec la notation compréhensive

```
a = [x**2 for x in range(21) if x*2 < 21]  
print (a)  
→ [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

```
b = {2 * x for x in range (20)}  
print (b)  
→ {0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38}
```

```
c = {x : x**2 for x in range (10)}  
print (c)  
→ {0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}
```


Modules

- les fonctions ou données (de librairie..) sont regroupées en modules
- par exemple, on charge le module `random` avec

```
import random
```

- notation qualifiée avec nom de module `random.sample`

```
def rand_array (n, p) :  
    return random.sample (range(p), n)
```

- on peut raccourcir la notation qualifiée `rd.sample` le nom du module avec

```
import random as rd
```

- on peut utiliser la notation simple `sample` sans le nom du module avec

```
from random import *
```

- la liste des modules disponibles s'obtient avec

```
help()  
modules  
.  
.  
.  
quit
```

Modules

Exercice: faire sa propre bibliothèque de modules, par exemple en y incorporant `sum_of ([ ])` et `max_of ([ ])` ?

- on crée un fichier `myLib.py`

```
import myLib
```

- ou

```
import myLib as my
```

- ou

```
from myLib import *
```

Prochain cours

- réviser les classes et objets
- classes et objets
- structures de données
- structures arborescentes
- parcours d'arbres