Introduction à la complexité

Jeudi 4 mai

Option Informatique

Ecole Alsacienne

Avant de commencer

- Prochaines séances
  - 11 mai : *Tris et complexité*
  - 18 mai : Devoir sur table
  - 25 mai : Pont de l'Ascension
  - 1er juin : Introduction au développement web
  - 8 juin : dernière séance

- TD à rendre pour le 28 mai
  - *Trolls & Châteaux*
  - *Conversions et encodages*
  - *Méthodes de chiffrement*

Plan

1. Ordres de grandeur
2. Définitions et notations
3. Premiers exemples
4. Complexité et récursivité
5. Exercices

Ordres de grandeur

ORDRES DE GRANDEUR

- *A combien estimez-vous les quantités suivantes ?*
  - Nombre de gênes de l'être humain : 30 000
  - Nombre de cheveux sur la tête : 125 000
  - Nombre de livres et d'imprimés à la BNF : 54 millions
  - Nombre d'être humains : 7,4 milliards
  - Nombre de neurones dans un cerveau humain : $\mathbf{10^{11}}$
  - Nombre de cellules dans le corps humain : $\mathbf{10^{14}}$
  - Nombre d'insectes sur Terre : $\mathbf{10^{18}}$
  - Nombre d'atomes dans le corps humain : $\mathbf{7 \times 10^{27}}$
  - Nombre d'atomes constituant la Terre : $\mathbf{10^{50}}$
  - Nombre de particules dans l'Univers : $\mathbf{10^{80}}$
  - Nombre de parties possibles aux échecs : $\mathbf{10^{123}}$

ORDRES DE GRANDEUR

- *Quel est l'âge de l'univers ?*
  - Environ 15 milliards d'années
  - Soit environ $\mathbf{5 \times 10^{17}}$ secondes

- *Combien d'opérations élémentaires un ordinateur peut-il faire par seconde ?*
  - Les processeurs actuels font tous au moins 1 Ghz
  - Rappel : 1 Hz = "1 fois par seconde"
  - Donc quelques milliards d'opérations par seconde

- *Combien de temps faut-il à un ordinateur pour compter de 1 en 1 jusqu'à* $\mathbf{10^{27}}$ *?*
  - *Deux fois l'âge de l'univers (*$\mathbf{10^{18}}$ secondes)
  - Explication :
    - En une seconde, il compte de 1 à $\mathbf{10^9}$
    - En dix secondes, il compte de 1 à $\mathbf{10 \times 10^9 = 10^{10}}$
    - En mille secondes, il compte de 1 à $\mathbf{1000 \times 10^9 = 10^{12}}$
    - En $\mathbf{10^{18}}$ secondes, il compte de 1 à $\mathbf{10^{18} \times 10^9 = 10^{27}}$

Comparaisons entre fonctions

- *Lequel de ces nombres est le plus grand ?*

(quand n prend des valeurs très élevées)

  - $1000 \times n$ *ou* n^2
  - n *ou* $2 \times n$
  - n *ou* n^2
  - n^{10} *ou* 2^n

L'énigme du nénuphar

- Le premier janvier, un étang contient un unique nénuphar.
- Chaque nuit, chaque nénuphar donne naissance à un nouveau nénuphar.
- Chaque nénuphar recouvre une petite surface du lac (toujours la même)

- **Question** : *Sachant que la moitié du lac est recouverte le 31 janvier, quand le lac entier sera-t-il entièrement recouvert ?*

La fonction exponentielle

- Notation : l'exponentielle de n se note e^n ou $\exp(n)$

- Informellement :
  - "*plus n est grand, plus e^n monte vite*"
  - "*plus n est grand, plus e^n est plus grand que e^{n-1}*

- Propriétés mathématiques :
  - $\exp(n+m) = \exp(n) \times \exp(m)$
  - $\exp(0) = 1$
  - La dérivée de l'exponentielle est l'exponentielle : $\exp'(x) = \exp(x)$

La fonction 2^n

- Lien avec les nénuphars : 2^n se comporte comme e^n

- Informellement :
  - "*plus n est grand, plus 2^n monte vite*"
  - "*plus n est grand, plus 2^n est plus grand que 2^{n-1}*

- Propriétés mathématiques :
  - $2^{n+m} = 2^n \times 2^m$
  - $2^0 = 1$

- S'il s'est écoulé n jours depuis le premier janvier, l'étang contient 2^n nénuphars

La fonction logarithme

- On peut voir la fonction **logarithme** comme l'inverse de la fonction exponentielle :

$$\log(\exp(x)) = x = \exp(\log(x))$$

- On parle souvent du **logarithme en base 2**, qui est tel que :

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

- De façon analogue, le **logarithme en base 10** vérifie :

$$\log_{10}(10^n) = n$$

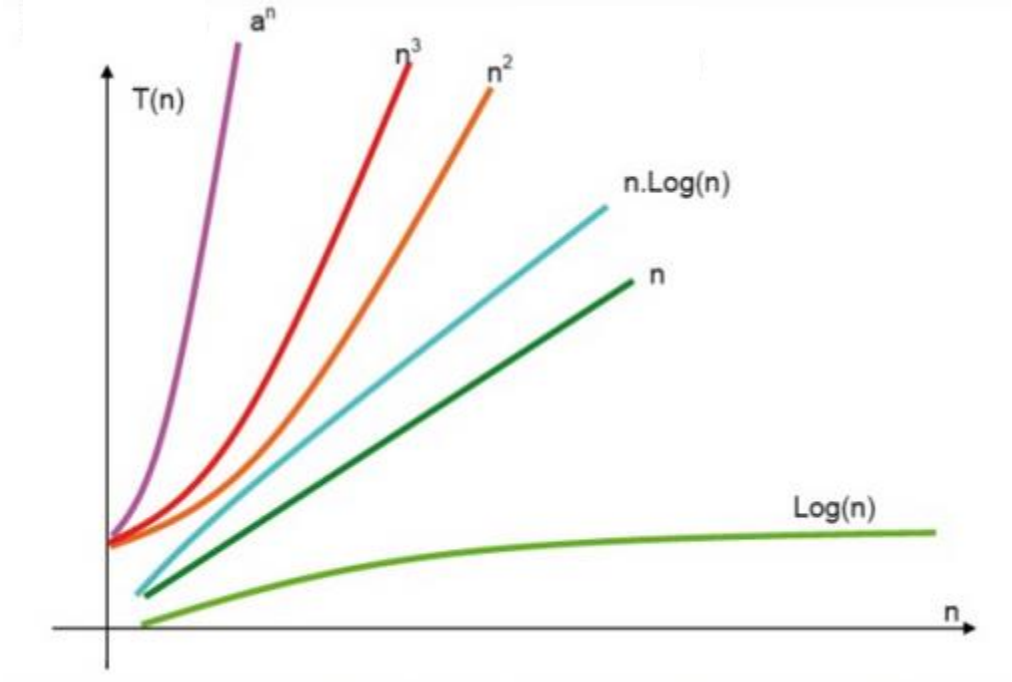
- « Le logarithme est aussi lent que l'exponentielle est rapide »

COMPARAISONS ENTRE FONCTIONS

n	n^2	n^3	n^{10}	2^n	$\log_{10}(n)$
1	1	1	1	2	0
2	4	8	1024	4	0,30103
3	9	27	59049	8	0,47712125
4	16	64	1048576	16	0,60205999
5	25	125	9765625	32	0,69897
10	100	1000	1E+10	1024	1
20	400	8000	1,024E+13	1048576	1,30103
50	2500	125000	9,7656E+16	1,1259E+15	1,69897
100	10000	1000000	1E+20	1,2677E+30	2
500	250000	125000000	9,7656E+26	3,273E+150	2,69897
1000	1000000	1000000000	1E+30	1,072E+301	3

Notation : $x\text{E+}k = x \times 10^k$

COMPARAISONS ENTRE FONCTIONS



Une simple feuille de papier

- On prend une feuille de papier très grande.

- *Quelle épaisseur obtient-on si on la plie 42 fois ?*

Plus que la distance Terre-Lune !

  - Distance Terre-Lune : 384 467 km, soit environ $3{,}8 \times 10^8$ m
  - Epaisseur d'une feuille de papier : un peu plus de 10^{-4} m
  - $10^{-4} \times \underbrace{2 \times 2 \times \cdots \times 2}_{42 \text{ fois}} = 10^{-4} \times 2^{42} \approx 4{,}4 \times 10^8$

- *Et si on la plie 51 fois ?*

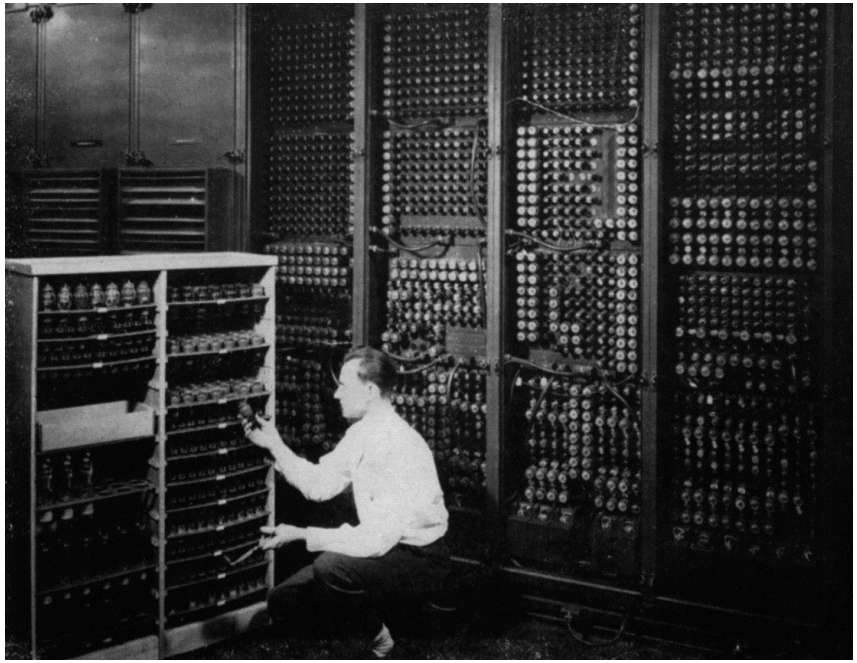
Plus que la distance Terre-Soleil !

  - Distance Terre-Soleil : 149 597 870 km, soit environ $1{,}5 \times 10^{11}$ m
  - $10^{-4} \times \underbrace{2 \times 2 \times \cdots \times 2}_{51 \text{ fois}} = 10^{-4} \times 2^{51} \approx 2{,}3 \times 10^{11}$

Définitions et notations

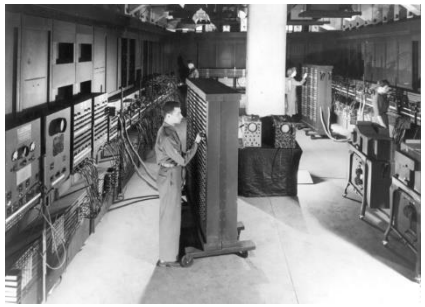
Petit voyage dans le temps

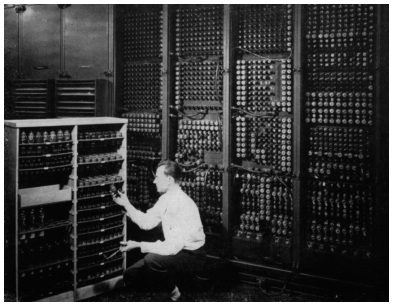
- *Qu'est-ce que c'est ?*



- Le premier ordinateur entièrement électronique
  - Date de présentation au public : 1946
  - *Electronic Numerical Integrator Analyser and Computer* (ENIAC)

Petit voyage dans le temps





- Caractéristiques
  - Dimensions : 2,4 x 0,9 x 30,5 mètres
  - Superficie : 167 m² (presque la taille d'un terrain de tennis)
  - Poids : 30 tonnes
  - Consommation électrique : 150 kilowatts (plusieurs dizaines de foyers)

- Composant principal : le tube à vide
  - Cause la plus fréquente de pannes : les insectes (*bug* en anglais)
  - Plus longue période de calcul sans panne : 116 heures (en 1954)

- Temps de calcul

	Multiplication de nombres à 10 chiffres	Calcul d'une trajectoire de tir
Calcul manuel	Plusieurs minutes	2,6 jours
ENIAC	0,001 s	3 s
Ordinateur de bureau des années 2000	30 ns	36 µs

Lois de Moore

- Gordon Earle Moore, un des trois fondateurs d'Intel

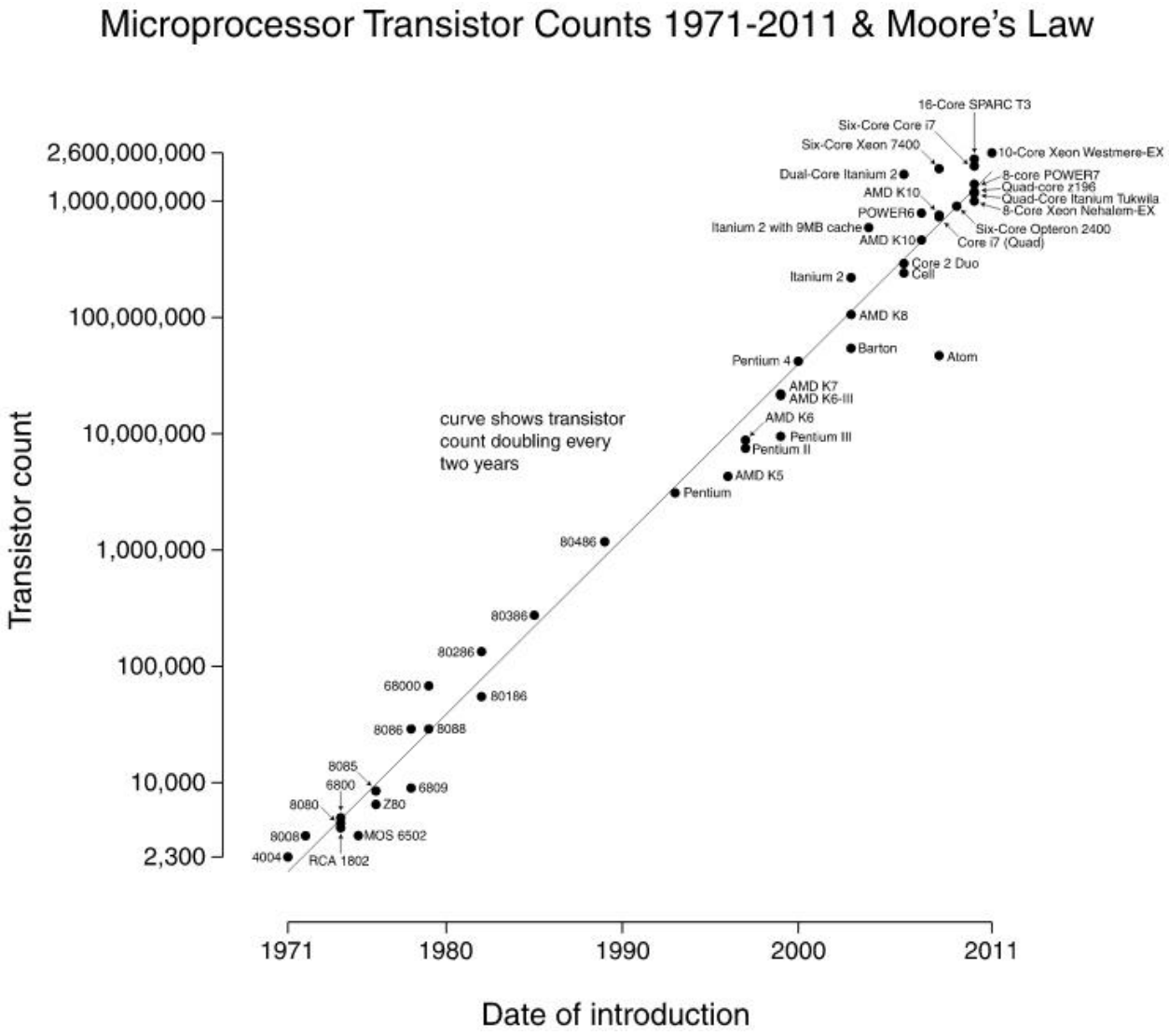
- Première loi de Moore (1965)

  La complexité des semi-conducteurs proposés en entrée de gamme double tous les ans à cout constant

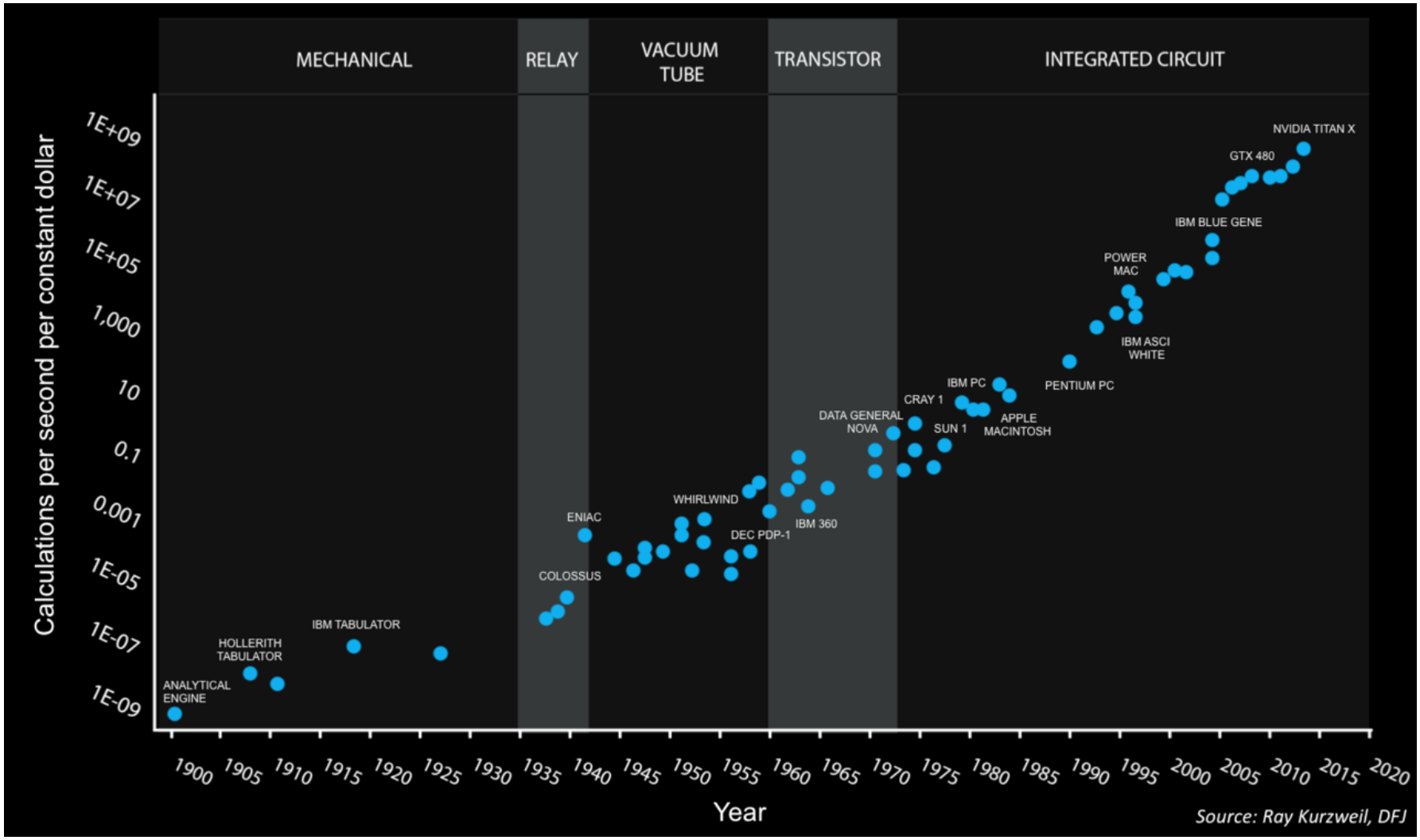
- Seconde loi de Moore (1975)

  Le nombre de transistors des microprocesseurs double tous les deux ans

Lois de Moore

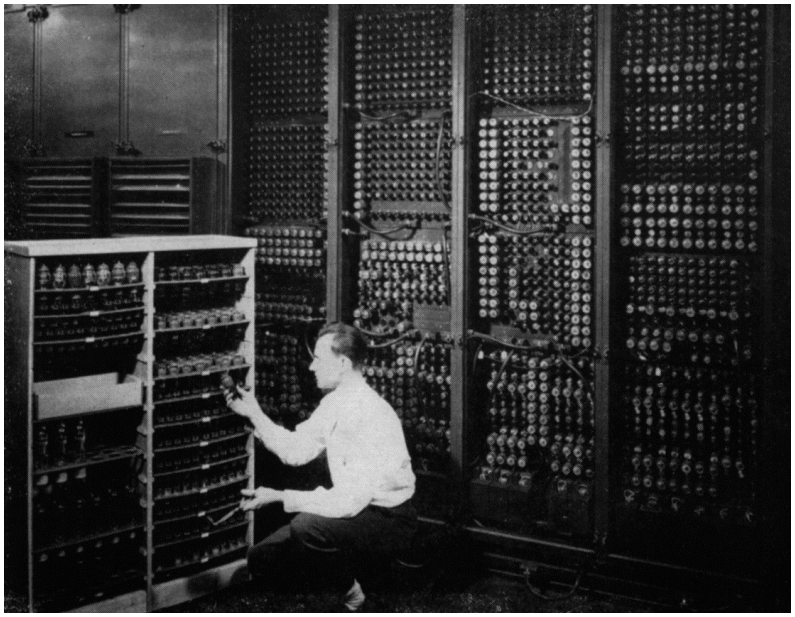


Lois de Moore



Lois de Moore

- **Conséquence** : Compter en combien de temps s'exécute un programme n'a pas vraiment de sens.



Lois de Moore

- **Question :** *Combien aurait couté un iPhone 5S s'il avait été produit en 1991 ?*
  Au moins 3,5 millions de dollars (2,6 millions d'euros)

- Mémoire
  - Cout d'un disque dur d'1 Go en 1991 : 10 000 dollars
  - Cout d'1 Go de mémoire Flash : 45 000 dollars
  - Cout de 32 Go de mémoire Flash : 1,4 millions dollars
- Processeur
  - Cout du millions d'instructions par secondes (MIPS) : 30 dollars
  - MIPS du microprocesseur l'iPhone 5S : 20 000
  - Cout du microprocesseur : 600 000 dollars
- Connectivité et Bande passante
  - Cout pour un kilo-octet par seconde : 100 dollars
  - Capacité d'un iPhone 5S : plus de 15 Mo/s
  - Cout de la connectivité et bande passante : 1,5 millions de dollars
- A l'époque, cette technologie n'aurait pas tenu dans un réfrigérateur

Une limite de la loi de Moore

- En 2017, on est capable de produire des composants de seulement 14 nanomètres, voire un peu moins
- D'ici 2020, on aura peut-être atteint 2 ou 3 nanomètres.
- Il devient difficile de réduire ainsi la taille des transistors :
  - 2 à 3 nanomètres, c'est la taille d'une dizaine d'atomes
  - Cela devient très couteux
  - On entre alors dans le domaine de la physique quantique
- On est sans doute en train de se heurter à une limite de la loi de Moore, mais d'autres pistes d'innovation existent
  - Des matériaux moins couteux, plus « propres »
  - D'autres axes de réflexions possibles

Rapidité d'exécution d'un programme

- **Constat** : Compter en combien de temps s'exécute un programme n'a pas vraiment de sens, mais certains programmes sont plus « rapides » que d'autres.

- Voici deux programmes :

```
def prog1():
    total = 0
    for i in range(10000000):
        total = total + 1
    return total
```

```
def prog2():
    total = 0
    for i in range(10):
        total = total + 1000000
    return total
```

- Quelque soit la machine utilisée, le second est un million de fois plus rapide que le premier.

Définition

- La **complexité en temps** d'un programme est
  - Une estimation du temps qu'il met à s'exécuter
  - En fonction de la taille des arguments passés en entrée
  - A une constante près

- La **complexité en mémoire** d'un programme est
  - Une estimation de l'espace dont il a besoin pour s'exécuter
  - En fonction de la taille des arguments passés en entrée
  - A une constante près

- **Remarque** : On se concentrera dans un premier temps sur la complexité en temps.

"En fonction de la taille des arguments"

- Exemple

```
def programmeIdiot1(n):
    x = 1
    for i in range(n):
        x = 2
    return x
```

Linéaire

```
def programmeIdiot2(n):
    x = 1
    for i in range(n*n):
        x = 2
    return x
```

Quadratique

```
def programmeIdiot3(n):
    x = 1
    for i in range(2):
        x = 2
    return x
```

Constant

Exécution en temps constant

- On dit qu'une opération s'exécute en **temps constant** quand elle s'exécute toujours dans le même temps, quelque soit la taille de l'entrée :

$$Temps\ de\ calcul(n)\ \approx Constante$$

- **Remarque** : on considère que la plupart des opérations de base d'un langage de programmation sont en temps constant.

- **Exemples** :
  - Comparer deux nombres
  - Afficher un nombre
  - Récupérer la valeur d'une variable

- Notation : $O(1)$

Complexité linéaire

- On dit qu'un algorithme a une **complexité linéaire** quand il s'exécute dans un temps proportionnel à la taille de l'entrée :

$$Temps\ de\ calcul(n)\ \approx Constante\ \times\ n$$

- Exemples :
  - n
  - $10000 \times n$
  - $\frac{n}{2}$

- Notation : $O(n)$

COMPLEXITÉ QUADRATIQUE

- On dit qu'un algorithme a une **complexité quadratique** quand il s'exécute dans un temps proportionnel au carré de la taille de l'entrée :

$$Temps\ de\ calcul(n)\ \approx Constante\ \times\ n^2$$

- Exemples :
  - n^2
  - $10 \times n^2$
  - $\frac{n^2}{1000000}$

- Notation : $O(n^2)$

Complexité polynomiale

- On dit qu'un algorithme a une **complexité polynomiale** quand il s'exécute dans un temps proportionnel à une puissance de la taille de l'entrée :

$$Temps\ de\ calcul(n)\ \approx Constante\ \times\ n^k$$

(k ne dépendant pas de n)

- Exemples :
  - n^5
  - n^2
  - n

- Notation : $O(n^k)$

Complexité exponentielle

- On dit qu'un algorithme a une **complexité exponentielle** quand il s'exécute dans un temps qui augmente exponentiellement avec la taille de l'entrée, ce qui peut s'écrire :

$$Temps\ de\ calcul(n) \approx Constante \times k^n$$

(k ne dépendant pas de n)

- Exemple :
  - 2^n : le temps de calcul double chaque fois que n augmente de 1

- Notation : $O(2^n)$

Complexité logarithmique

- On dit qu'un algorithme a une **complexité logarithmique** quand il s'exécute dans un temps proportionnel au logarithme de la taille de l'entrée :

$$Temps\ de\ calcul(n) \approx Constante \times \log(n)$$

- Notation : $O(\log(n))$

RAPPEL : TABLEAU DES COMPARAISONS

n	n^2	n^3	n^{10}	2^n	$\log_{10}(n)$
1	1	1	1	2	0
2	4	8	1024	4	0,30103
3	9	27	59049	8	0,47712125
4	16	64	1048576	16	0,60205999
5	25	125	9765625	32	0,69897
10	100	1000	1E+10	1024	1
20	400	8000	1,024E+13	1048576	1,30103
50	2500	125000	9,7656E+16	1,1259E+15	1,69897
100	10000	1000000	1E+20	1,2677E+30	2
500	250000	125000000	9,7656E+26	3,273E+150	2,69897
1000	1000000	1000000000	1E+30	1,072E+301	3

- Meilleure sera la complexité, plus on pourra traiter des entrées de taille importante

Exemples simples

EXEMPLE 1 : AFFICHER LES NOMBRE DE 1 À n

- **But :** Ecrire un programme qui prend en entrée un entier n, qui affiche la liste des entiers entre 1 et n, et ne renvoie rien.

- **Exemple :**

```
exemple1(5)
1
2
3
4
5
```

EXEMPLE 1 : AFFICHER LES NOMBRE DE 1 À n

- Code :

```
def exemple1(n):
    for i in range(1,n+1)        Linéaire
        print(i)                 Constant
```

- Complexité : $O(n)$

EXEMPLE 2 : TABLE DE MULTIPLICATION

- **But :** Ecrire un programme qui prend en entrée un entier n, qui affiche une table de multiplication avec n lignes et n colonnes.

- **Exemple :**

```
exemple2(4)
1       2       3       4
2       4       6       8
3       6       9       12
4       8       12      16
```

EXEMPLE 2 : TABLE DE MULTIPLICATION

- Code :

```
def exemple2(n):
    for i in range(1,n+1):
        for j in range(1,n+1):
            print(i*j, end = "\t")
        print()
```

Quadratique

Linéaire

Constant

- Complexité : $O(n^2)$

Complexité et récursivité

RAPPEL : FONCTION RÉCURSIVE

- **Définition** : Une fonction récursive est une fonction qui s'appelle elle-même

- Modèle classique :

```
def f(n):
    if (n==0):
        # Renvoyer une valeur
    else:
        resultatAppelRecursif = f(n-1)
        g(resultatAppelRecursif)
```

EXEMPLE SIMPLE : LA FACTORIELLE

- **Définition** : La fonction factorielle est définie par

$$n! = factorielle(n) = \begin{cases} 1 & si \quad n = 0 \\ 1 \times 2 \times \cdots \times n & si\ n > 0 \end{cases}$$

- Définition récursive :

$$n! = factorielle(n) = \begin{cases} 1 & si \quad n = 0 \\ factorielle\ (n-1) \times n & si\ n > 0 \end{cases}$$

- Remarques :
  - $n!$ est plus rapide que n^k
  - $n!$ est plus rapide que 2^n

EXEMPLE SIMPLE : LA FACTORIELLE

- Définition récursive :

$$n! = factorielle(n) = \begin{cases} 1 \quad si \quad n = 0 \\ factorielle\,(n-1) \times n \quad sinon \end{cases}$$

- En Python :

```
def fact(n):
    if (n==0):
        return 1
    else:
        return n*fact(n-1)
```

Exemple simple : la factorielle

- En Python :

```
def fact(n):
    if (n==0):
        return 1
    else
        return n*fact(n-1)
```

- Analyse
  - Notons $C(n)$ le nombre d'opération élémentaires effectuées dans le calcul de `fact(n)`
  - $C(0) = 2$ (un test et le renvoi d'une valeur)
  - Si $n > 0$, $C(n) = 3 + C(n-1)$ (un test, une multiplication, et un renvoi)
  - $C(n) = 3 + 3 + C(n-2) = 3 + 3 + \ldots + C(0) = 3n + 2$

- Complexité : $O(n)$

Exemple un peu moins simple : la boule de cristal

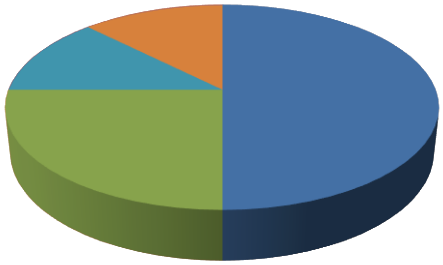
Rappelez-vous…

- Donnez moi un chiffre entre 1 et 1 000
- Je vous le retrouve en 10 questions binaires (oui – non)

Exemple un peu moins simple : la boule de cristal

- Principe :

A chaque étape, on divise par deux
la taille de l'intervalle de recherche.



Questions posées	0	1	2	3	4	5	6	7	8	9	10
Intervalle	1024	512	256	128	64	32	16	8	4	2	1

EXEMPLE UN PEU MOINS SIMPLE : LA BOULE DE CRISTAL

- En pseudo-code :

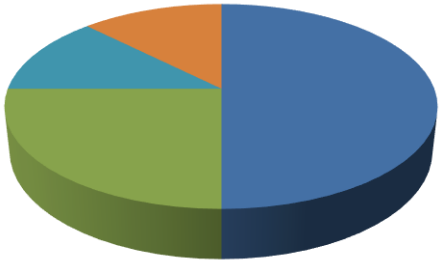
```
TrouverNombreMystere(min, max) =
    Si (min = max)
    Alors
        La réponse est min
    Sinon
        milieu = (min+max)/2
        Si (nombreMystere ≤ milieu)
        Alors
            TrouverNombreMystere(min, milieu)
        Sinon
            TrouverNombreMystere(milieu+1, max)
    Fin Si
```

Exemple un peu moins simple : la boule de cristal

- **Question :** Quelle est la complexité de ce programme ?

- **Remarque** : On peut supposer d'abord que n est une puissance de 2 : $n = 2^k$

- **Réponse :** Le temps de calcul est proportionnel à k, et donc à $\log_2(n)$



Exercices

Recherche dans un vecteur

- **Question** : Quelle est la complexité d'un algorithme qui recherche si un élément x appartient à une liste v ?

- **Algorithme** :

- **Remarque** : sauf indication contraire, on s'intéresse toujours à la complexité dans le pire des cas

Supprimer les doublons

- **Question** : Quelle est la complexité d'un algorithme :
  - Prenant en argument une liste d'entiers `l`
  - Renvoyant une liste `l2` correspondant à `l` sans doublons

- **Algorithme** :

Le plus long palindrome

- **Définition :** Un **palindrome** est une chaîne de caractères qu'on peut lire de gauche à droite ou de droite à gauche.

- **Exemples** :
  - "Bob"
  - "Kayak"
  - "La mariée ira mal"
  - "Zeus a été à Suez"
  - "Engage le jeu que je le gagne" (Alain Damasio, la Horde du Contrevent)

- **But :** Ecrire une fonction qui prend en argument une chaîne de caractère et qui renvoie le plus long palindrome qu'elle contient.

Tri par sélection

- Principe
  - On trouve le plus grand élément de la liste $l[0 \ldots n-1]$
  - On le met dans la dernière case
  - On trouve le plus grand élément de la sous-liste $l[0 \ldots n-2]$
  - On le met dans l'avant dernière case
  - Etc.

- But
  - Implémentez cette méthode
  - Déduisez-en sa complexité

- Fonction auxiliaire à écrire
  - Trouver l'indice du maximum d'un sous-liste

Tri par sélection

- Trouver l'indice du maximum :

- Complexité : $O(n)$

Tri par sélection

- Tri par sélection

- Complexité : $O(n^2)$

Tri à bulles

- Principe
  - On parcourt la liste du début à la fin
  - Si on a `l[i] > l[i+1]`, alors on inverse ces deux éléments (`l[i]` "remonte")
  - Si on arrive à la fin de la liste sans qu'il y ait d'échange, c'est que la liste est triée
  - Sinon on refait un passage
  - Etc.

- But
  - Implémentez cette méthode
  - Déduisez-en sa complexité

Tri par insertion

- Aussi appelé « tri du joueur de cartes »

- Principe
  - Le joueur a dans sa main des cartes déjà triées
  - Il reçoit une nouvelle carte
  - Il l'insère au bon endroit dans sa main
  - Il reçoit une nouvelle carte
  - etc.
- But
  - Implémentez cette méthode
  - Déduisez-en sa complexité

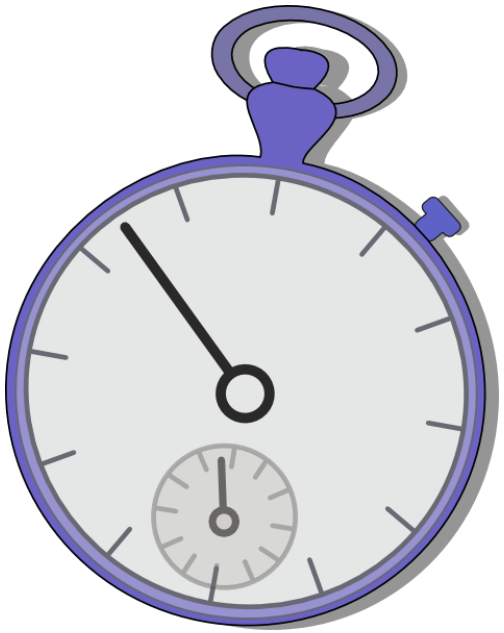
Prochaine séance

Jeudi 11 avril

Tris et complexité

n

2^n



n^2

$n \times \log n$