Introduction à la complexité

Mardi 23 Avril

Option Informatique

Ecole Alsacienne

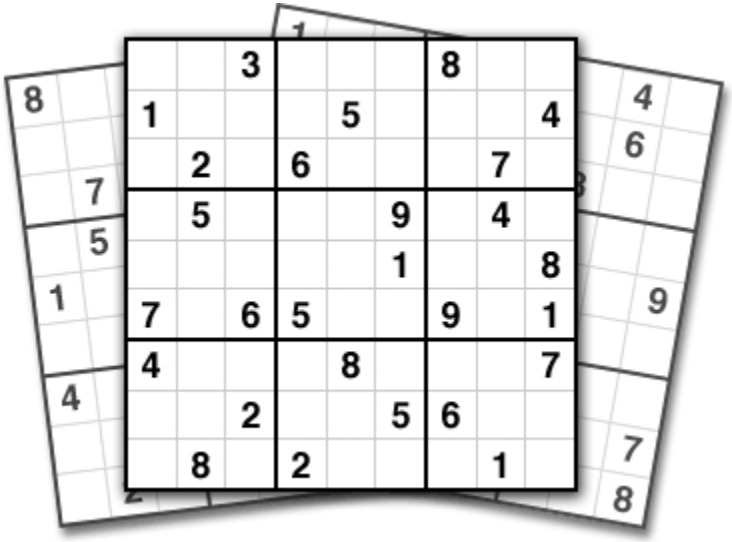
Absence du 9 avril

- Toutes mes excuses

- Séance de rattrapage : jeudi 23 mai de 14h à 16h

Mini-projet de programmation

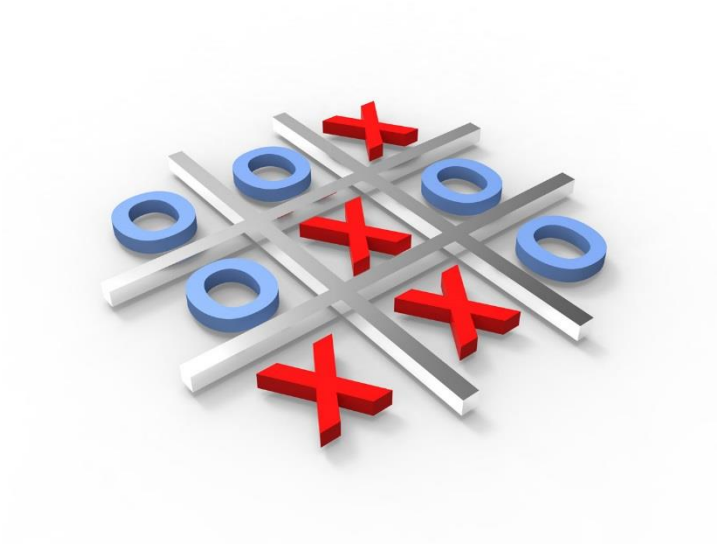
Piste 1 : Sudoku



- Résolution manuelle
- Résolution automatique
- Génération automatique

MINI-PROJET DE PROGRAMMATION

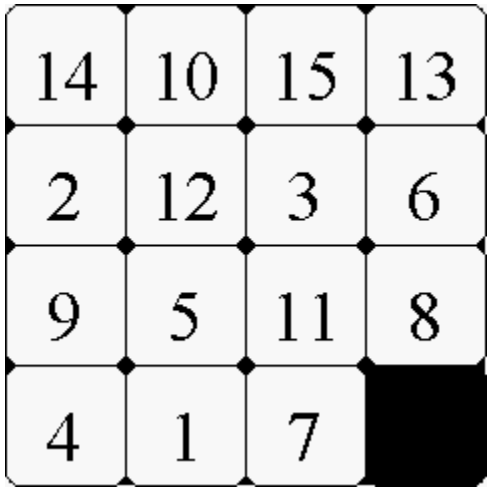
Piste 2 : Tic-Tac-Toe



- Joueur contre joueur
- Joueur contre aléatoire
- Joueur contre IA

Mini-projet de programmation

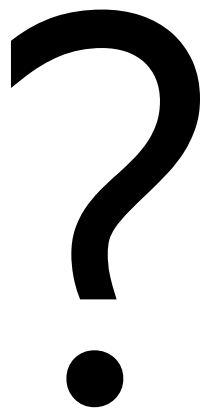
Piste 3 : Taquin



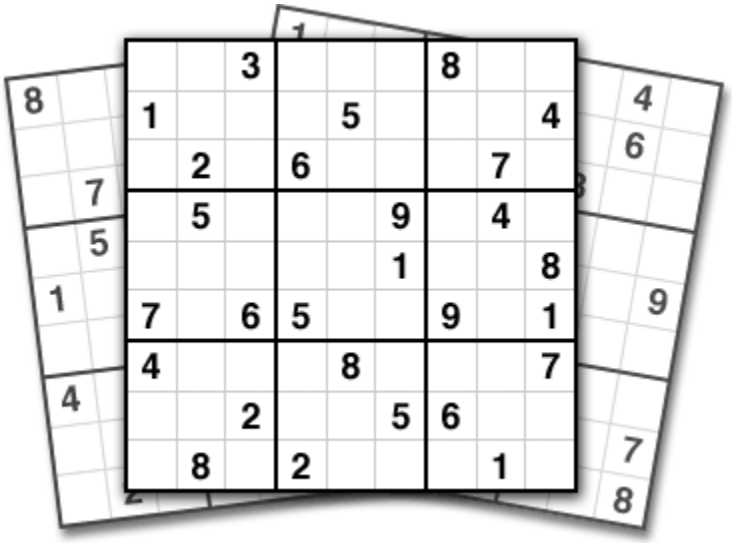
- Résolution manuelle
- Génération automatique
- Résolution automatique

Mini-projet de programmation

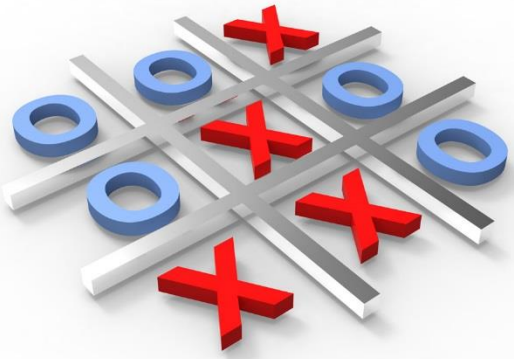
Piste 4



Mini-projet de programmation



14	10	15	13
2	12	3	6
9	5	11	8
4	1	7	



?

Quel projet préférez-vous ?

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 : 35 millions
  - Nombre d'être humains : 7 milliards
  - Nombre de neurones dans un cerveau humain : 10^{11}
  - Nombre de cellules dans le corps humain : 10^{14}
  - Nombre d'insectes sur Terre : 10^{18}
  - Nombre d'atomes dans le corps humain : 7×10^{27}
  - Nombre d'atomes constituant la Terre : 10^{50}
  - Nombre de particules dans l'Univers : 10^{80}
  - Nombre de parties possibles aux échecs : 10^{120}

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 secondes ?*
  - 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)

Comparaisons entre fonctions

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

(quand n des 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 :

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

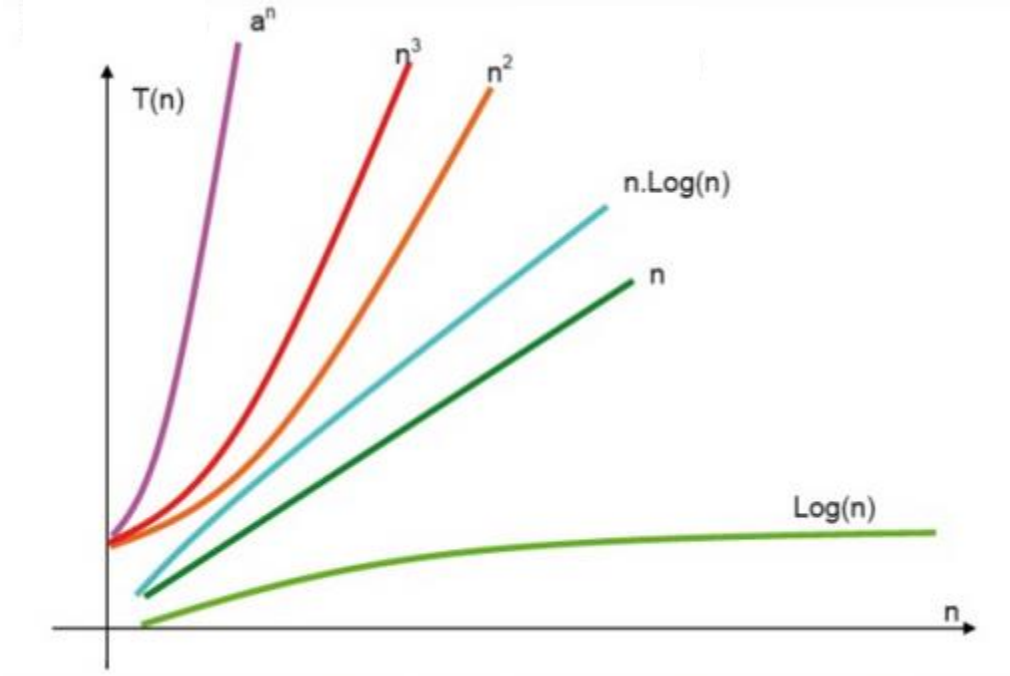
- « Le logarithme est aussi lent que l'exponentielle est rapide »

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
  - $2^{42} \approx 4{,}4 \times 10^{12}$

- *Et si on la plie 50 fois ?*

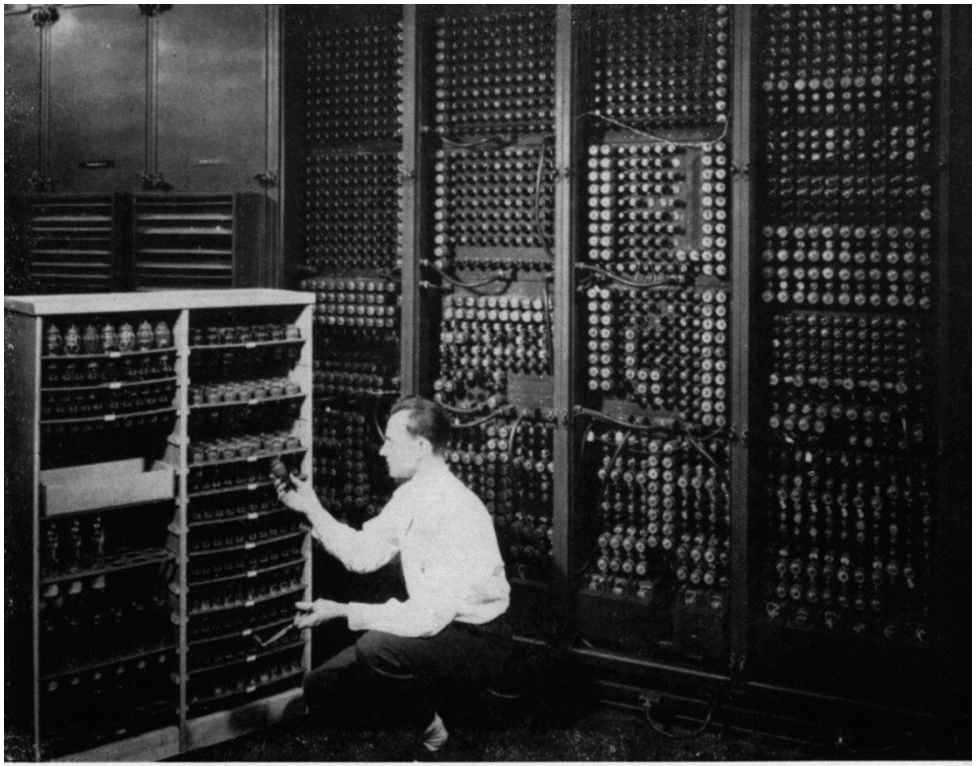
Pratiquement la distance Terre-Soleil !

  - Distance Terre-Soleil : 149 597 870 km, soit environ $1{,}5 \times 10^{11}$ m
  - $2^{50} \approx 1{,}1 \times 10^{15}$

Définitions et notations

Petit voyage dans le temps

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



Replacing a bad tube meant checking among ENIAC's 19,000 possibilities.

- *Que se passe-t-il ?*

Lois de Moore

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

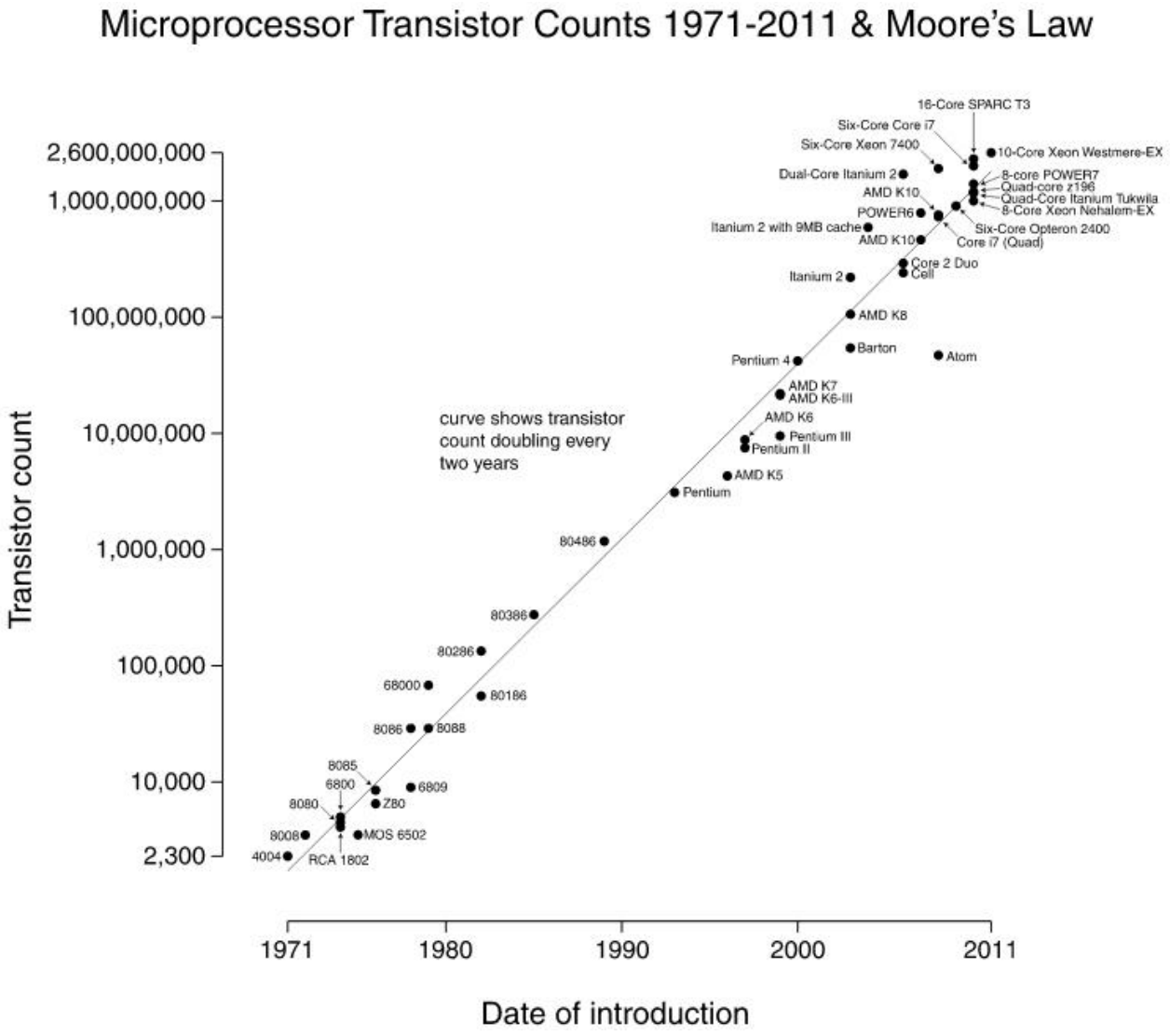
- **Première loi de Moore (1965)**

  La complexité des semiconducteurs 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



Conséquences

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

- Voici deux programmes :

```
let prog1() =
    let total = ref 0 in
    for i = 0 to 10000000
    do
         total :=
           !total + 1;
    done;
    !total;
```

```
let prog2() =
    let total = ref 0 in
    for i = 0 to 10
    do
         total :=
           !total + 1000000;
    done;
    !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 algorithme 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
  - ```
    let programme_idiot_1(n)=
      for i = 0 to (n - 1)
           do
           done;
    ```
    Linéaire
  - ```
    let programme_idiot_2(n)=
      for i = 0 to (n*n - 1)
           do
           done;
    ```
    Quadratique
  - ```
    let programme_idiot_3(n)=
      for i = 0 to 0
           do
           done;
    ```
    Constant
    ```

Exécution en temps constant

- On dit qu'une opération s'exécute en **temps constante** 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
- : unit = ()
```

Exemple 1 : afficher les nombre de 1 à n

- Code :

```
let exemple1 n =
  for i = 1 to n                    Linéaire
  do
    print_int i;
    print_newline();                Constant
  done;;
```

- 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
- : unit = ()
```

EXEMPLE 2 : TABLE DE MULTIPLICATION

- Code :

```
let exemple2 n =
  for i = 1 to n
  do
    for j = 1 to n
    do
      print_int (i*j);
      print_string "\t"
    done;
    print_newline();
  done;;
```

Quadratique

Linéaire

Constant

- Complexité : $O(n^2)$

EXEMPLE 3 : TRI PAR INSERTION

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

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

EXEMPLE 3 : TRI PAR INSERTION

- Trouver l'indice du maximum :

```
let trouver_indice_du_maximum v indice_max =
  let indice_max_courant = ref 0 in
  let max_courant = ref v.(0) in
  for i = 1 to indice_max
  do
    if (v.(i)> !max_courant)
    then
      begin
          max_courant := v.(i);
          indice_max_courant := i
      end;
  done;
  !indice_max_courant;;
```

Constant

Linéaire

Constant

Constant

- Complexité : $O(n)$

EXEMPLE 3 : TRI PAR INSERTION

- Tri par insertion

```
let tri_par_insertion v =
  for indice_max = Array.length(v)-1 to 1
  do
    let max_sous_vecteur =                    Linéaire
       trouver_indice_du_maximum v indice_max in
    let stock = v.(max_sous_vecteur) in
    v.(max_sous_vecteur) <- v.(indice_max);
    v.(indice_max) <- stock;                  Constant
  done;;                                      Quadratique
```

- Complexité : $O(n^2)$

Complexité et récursivité

RAPPEL : FONCTION RÉCURSIVE

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

- Modèle classique :

```
let f(n) =
    if (n=0)
    then
          // Renvoyer une valeur
    else
          let resultat_appel_recursif = f(n-1) in
          g(resultat_appel_recursif);;
```

EXEMPLE SIMPLE : LA FACTORIELLE

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

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

- Définition récursive :

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

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

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 OCaml :

```
let rec fact n =
    if (n=0)
    then
          1
    else
          n*fact(n-1)
```

EXEMPLE SIMPLE : LA FACTORIELLE

- En OCaml :

```
let rec fact n =
    if (n=0)
    then
          1
    else
          n*fact(n-1)
```

- Complexité : $O(n)$

Exemple un peu moins simple : la boule de cristal

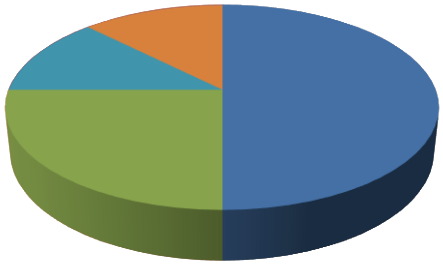
Rappelez-vous…

- 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

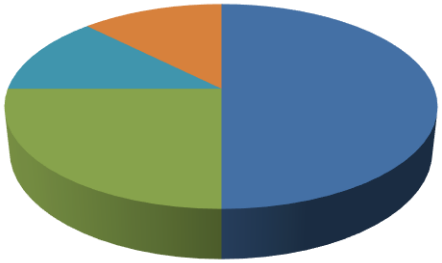
EXEMPLE UN PEU MOINS SIMPLE : LA BOULE DE CRISTAL

- 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 proportionel 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 un vecteur v ?

- **Algorithme** :

```
let appartient x v =
    let n = Array.length v in
    let trouve = ref false in
    let indice_actuel = ref 0 in
    while ((!not trouve) && (!indice_actuel < n)
    do
        if (v.(!indice_actuel) = x)
        then trouve := true;
    done;
    !trouve;;
```

- **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** :

```
let rec supprimer_doublons l =
   if (l = [])
   then []
   else
      begin
         let t = List.hd l in
         let q = List.tl l in
         if (List.mem t q)
         then supprimer_doublons q
         else t :: (supprimer_doublons q)
      end;;
```

SUPPRIMER LES DOUBLONS

- Question : Quelle est la complexité d'un algorithme :
  - Prenant en argument une liste d'entiers `l` compris entre 0 et 99
  - Renvoyant une liste `l2` correspondant à `l` sans doublons

- Algorithme :

```
let supprimer_doublons_0_100 l =

   let trouve = Array.make 100 false in

   let l2 = ref l in
   while (!l2 <> [])
   do
      let t = List.hd !l2 in
      let q = List.hd !l2 in
      trouve.(t) <- true;
      l2 := q
   done;

   let l3 = ref [] in
   for i = 0 to 99
   do
      if (trouve.(i))
      then l3 := i :: !l3;
   done;
   !l3;;
```

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.

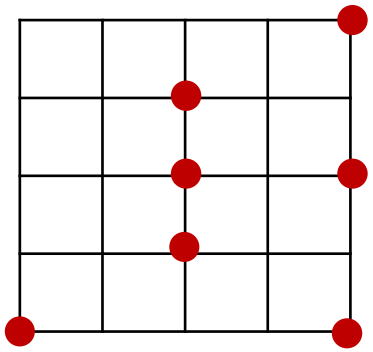
Recherche de milieux

- **En entrée :** une série de points du plan définis par leurs coordonnées :

$$\{ (x_i, y_i) \}_{1 \leq i \leq n}$$

- **Question :** Trouver la liste des points qui sont des milieux, c'est-à-dire situés exactement entre deux autres points

- **Exemples** :
  - (2,2) milieu de (2,1) et (2,3)
  - (2,2) milieu de (0,0) et (4,4)
  - (4,2) milieu de (4,0) et (4,4)
  - (2,1) milieu de (0,0) et (4,2)



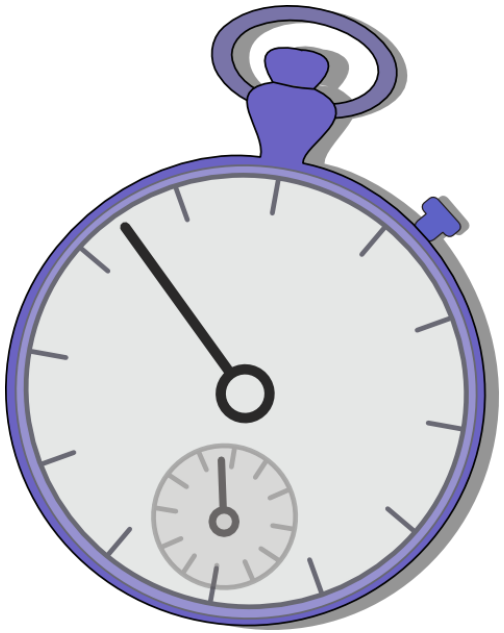
Prochaine séance

Mardi 23 Avril 2013

[TD] La complexité en pratique

n

2^n



n^2

$n \times \log n$