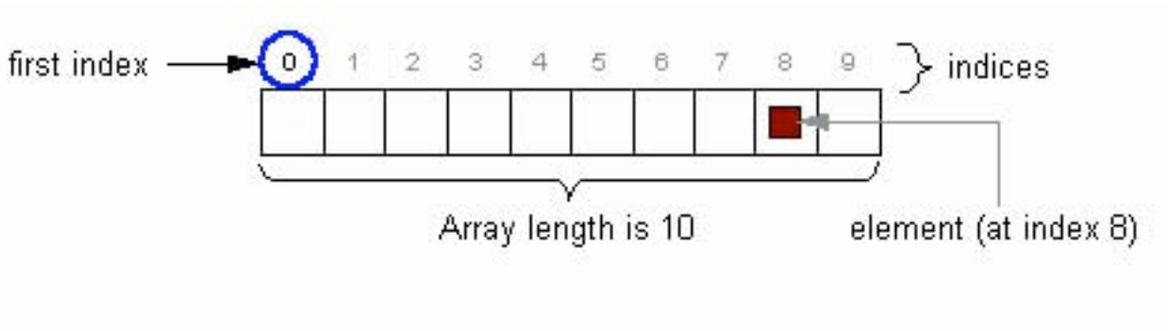
Programmation par Objets - Printemps 2010
Frédéric Mallet & Jean-Paul Roy
L1 Info&Math - Faculté des Sciences de Nice

http://deptinfo.unice.fr/~roy

Une collection de taille fixe

Les tableaux



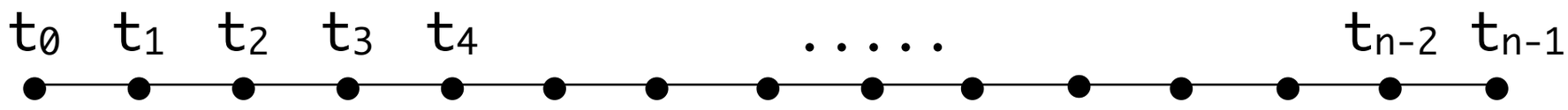
Pourquoi des collections ?

- Il est souvent nécessaire de stocker de nombreuses données dans des **collections** : bibliothèques, sécurité sociale, cartes d'étudiants, albums de photos, relevés météo, etc.

- Des centaines, des milliers, des millions d'éléments ! Il est ainsi hors de question de créer une variable par élément.

- La **taille** (nombre d'éléments) d'une collection peut être **fixe** (les lettres de l'alphabet) ou **variable** (ma collection de DVD).

- Nous allons étudier les **tableaux** qui sont des collections :
  - **de taille fixe** : on choisit un nombre d'éléments et on s'y tient.
  - dont les éléments sont **tous de même type**, et **numérotés**.



Les collections de type **tableau**

• Comme la plupart des langages, Java propose des **tableaux** : ce sont des collections d'éléments **numérotés**, **de même type**, **en nombre fixé** une fois pour toutes !

• On pourra donc seulement :

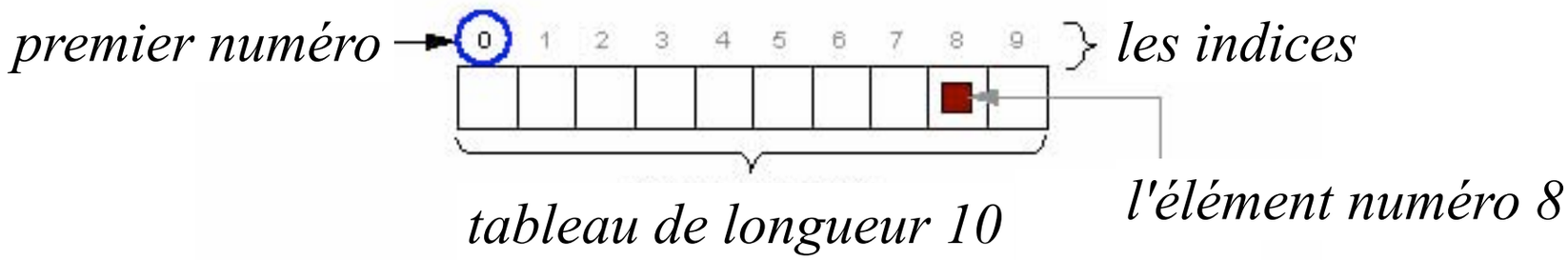
- **consulter** directement l'élément numéro k
- **modifier** directement l'élément numéro k

COMPLEXITE : Il est garanti que le **temps d'accès à l'élément numéro k** du tableau ne dépend pas de k (il a lieu **en temps constant**).

• Notation Java : si `T` est un type, alors `T[]` dénotera l'ensemble des tableaux dont les éléments sont de type `T`

`int[]`	`boolean[]`	`Cercle[]`	`float[]`
tableau d'entiers	*tableau de booléens*	*tableau de cercles*	*tableau de réels*

Déclaration d'un tableau d'entiers



- Soit `tab` un **nouveau tableau de 10 entiers** :

```
int[] tab = new int[10];
```

- Affichons la **longueur** du tableau `tab` :

```
println("Longueur  = " + tab.length);
```

```
Longueur = 10
```

- Nous commençons tout doucement à pénétrer dans l'univers Java pour lequel un tableau est un **objet**. Les variables propres à un objet sont nommées des **champs** (**attributs**, **propriétés**) de cet objet. La longueur est un champ d'un tableau.

- La notation pointée `objet.champ` permet d'accéder au champ d'un objet. Ici `tab.length` se lit : la **longueur du tableau** `tab`.

Remplissage d'un tableau d'entiers

- L'instruction `int[] tab = new int[10];` déclare une nouvelle variable `tab` de type `int[]` dont les éléments sont initialisés à `0`.

- Pour y placer des éléments de notre choix, nous pouvons :

  - **les énumérer à la déclaration** s'il y en a peu et qu'ils sont tous connus :

```
int[] mesures = {12, 8, 10, 14, 5, 13, 1, 6, 0, 8};
```

  - utiliser une **boucle de remplissage**. Mettons-y par exemple les `10` premiers nombres impairs :

```
int[] mesures = new int[10];
for (int i = 0; i < 10; i = i + 1) {
  mesures[i] = 2 * i + 1;
}
```

- La notation `t[i]` dénote donc **l'élément numéro i** d'un tableau `t`.

→ $0 \leq i < t.length$

Les éléments d'un tableau sont des variables !

- Plus précisément, ce sont des **variables indexées**. Les matheux parlent d'une ***suite finie*** $t_0, t_1, t_2, \ldots, t_{n-1}$. Les programmeurs parlent d'un tableau de longueur n, d'éléments `t[0]`, `t[1]`, `t[2]`, ..., `t[n-1]`.

- Comme toutes les variables, on peut donc les modifier (en respectant leur type) :

```
mesures[i] = 2 * mesures[i];
```

Je double la valeur de la mesure numéro i

- On veillera à bien respecter les **valeurs légales des indices** : si un tableau `t` est de longueur `10`, de demandez pas `t[-1]` ou `t[10]` !!!

- Sinon vous risquez une erreur à l'exécution et... COUIC !

java.lang.ArrayIndexOutOfBoundsException

Affichage d'un tableau

- L'instruction `println` n'aime pas trop afficher un tableau :

```
int[] notes = new int[5];
println("notes = " + notes);
```

Run

```
notes = [I@22d5b5
```

- En *Processing* (mais *pas en Java*), vous pouvez demander simplement `println(notes)` qui affichera son contenu à la verticale ! A EVITER !

- On préfèrera plutôt un **affichage à l'horizontale** que l'on se programme soi-même avec une boucle `for` :

```
int[] notes = {6, 3, 8, 2, 3};
for (int i = 0; i < notes.length; i = i + 1) {
  print(notes[i] + " ");
}
println();        //  pour aller à la ligne en bout de ligne !
```

Quelques Algorithmes de base dans les Tableaux...

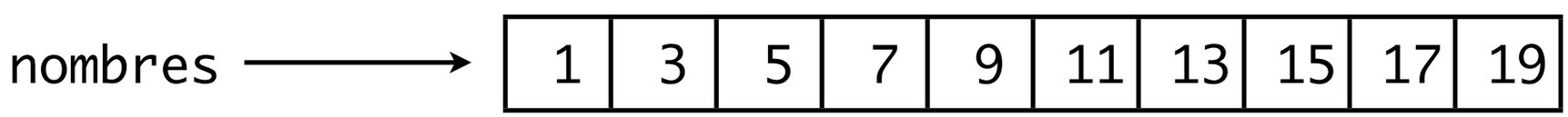
Il est souhaitable que vous connaissiez ces algorithmes quasiment par coeur, en les comprenant bien entendu.
Sachez que la majorité des problèmes sont des ***déformations de problèmes déjà vus****...*

Somme des éléments d'un tableau

- C'est un exemple de **parcours exhaustif** : on doit explorer **tout** le tableau, de gauche à droite par exemple.
- Comme avec l'affichage à l'horizontale...

Initialisation d'un tableau d'entiers :

```
int[] nombres = new int[10];
for (int i = 0; i < nombres.length; i = i + 1) {
  nombres[i] = 2 * i + 1;
}
```



Calcul et affichage de la somme de ses éléments :

```
int somme = 0;
for (int i = 0; i < nombres.length; i = i + 1) {
  somme = somme + nombres[i];
}
println("Somme des éléments : " + somme);
```

On écrit plutôt une méthode !

- Plutôt qu'un cas particulier, on programme le cas général dans une méthode qui prendra un tableau d'entiers quelconque t :

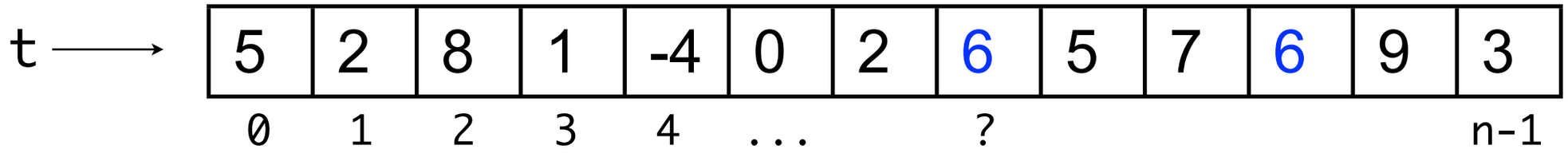
```
int sommeTableau(int[] t) {
  int somme = 0;
  for (int i = 0; i < t.length; i = i + 1) {
    somme = somme + t[i];
  }
  return somme;
}
```

- C'est au programme qui utilisera cette méthode de décider s'il veut ou non afficher le résultat :

```
int[] nombres = {7,9,3,2,5,4,6};
println("Somme des éléments : " + sommeTableau(nombres));
```

Recherche **séquentielle** dans un tableau

• PROBLEME : Etant donné un tableau t (par exemple d'entiers), chercher si un entier donné x est un élément du tableau, et à quelle position.



• Si le tableau n'a aucune particularité, la seule solution consiste à tout balayer **en séquence** de gauche à droite pour trouver x.

• Commençons par chercher le **numéro de la première occurrence de x, ou bien -1** (indice inexistant) si x n'est pas un élément du tableau.

J'en suis à l'indice i.
Si i est trop grand, échec.
Sinon si t[i] est égal à x, gagné !
sinon passer à i+1.

Parcours non exhaustif !

- L'algorithme précédent peut se programmer par **récurrence sur i** :

```
int rechercheSeq1(int x, int[] t) {
  return rechercheSeq1(x, 0, t);
}

int rechercheSeq1(int x, int i, int[] t) {
  // je cherche à partir de l'indice i
  if (i >= t.length) return -1;
  if (t[i] == x) return i;
  return rechercheSeq1(x,i + 1,t);
}
```

- Exécution :

```
void setup() {
  int[] t = {5, 2, 8, 1, -4, 0, 2, 6, 5, 7, 6, 9, 3};
  print("On travaille sur le tableau : ");
  afficheTableau(t);
  print("Version recursive. Je cherche l'indice de 6 dans le tableau : ");
  int indice = rechercheSeq1(6,t);
  println(indice);
  print("Indice de 14 dans le tableau : " + rechercheSeq1(14,t));
}
```

```
On travaille sur le tableau : 5 2 8 1 -4 0 2 6 5 7 6 9 3
Version recursive. Je cherche l'indice de 6 dans le tableau : 7
Indice de 14 dans le tableau : -1
```

• L'algorithme se programme aussi **de manière itérative**. Attention de ne pas accéder à une case inexistante du tableau !

```
int rechercheSeq2(int x, int[] t) {
  int i = 0;
  while (i < t.length && t[i] != x) {
    i = i + 1;
  }
  // donc ici i >= t.length || t[i] == x
  if (i < t.length) return i;
  else return -1;
}
```

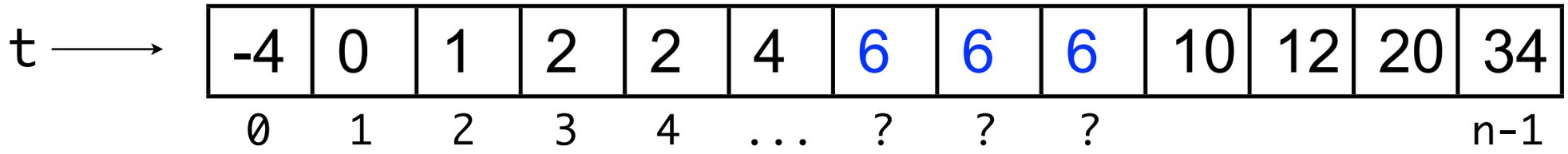
• Il est souvent agréable dans les tableaux d'utiliser une boucle `for` de parcours exhaustif quitte à s'échapper avec `break` ou `return`.
Cela garantit de ne jamais sortir du tableau !

```
int rechercheSeq3(int x, int[] t) {
  for (int i = 0; i < t.length; i = i + 1) {
    if (t[i] == x) return i;
  }
  return -1;
}
```

Recherche **dichotomique** dans un tableau **trié**

• Une **dichotomie** (coupure en deux) est intéressante si le tableau est **trié** (par exemple en ordre croissant).

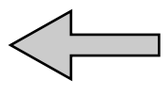
• Stratégie : je regarde le milieu. Si c'est l'élément cherché, j'ai gagné ! Sinon il me suffit de **continuer la recherche dans une seule moitié du tableau**...



J'en suis à travailler entre les indices a et b.
S'il n'y a plus d'indices, échec !
Sinon soit m le milieu de [a,b].
Si t[m] est l'élément cherché, gagné !
Sinon, si x < t[m], recherche à gauche dans [a,m-1]
Sinon, recherche à droite dans [m+1,b].

• L'algorithme précédent peut se programmer par **récurrence sur la longueur de l'intervalle de recherche [a,b]** :

```
int rechercheDicho1(int x, int[] t) {
  return rechercheDicho1(x,0,t.length-1,t);
}

int rechercheDicho1(int x, int a, int b, int[] t) {
  // je cherche dans l'intervalle d'indices [a,b]
  if (a > b) return -1;
  int m = (a + b) / 2;
  if (t[m] == x) return m;
  if (x < t[m]) return rechercheDicho1(x,a,m-1,t);
  return rechercheDicho1(x,m+1,b,t);
}
```



• Je divise donc la longueur de l'intervalle de recherche par deux chaque fois, et je tombe sur une COMPLEXITE LOGARITHMIQUE !
En passant de 1000 éléments à 1000000 éléments, le temps de calcul ne fait que doubler...

• L'algorithme se programme aussi **itérativement** (cf TD)...

Recherche du plus grand élément

- D'un tableau non trié bien entendu...

```
int maximum1(int[] t) {
  return maximum1(0,t);
}
```

récursif

```
int maximum1(int i, int[] t) {
  // je cherche le maximum depuis l'indice i
  if (i == t.length-1) return t[t.length - 1];
  return max(t[i], maximum1(i+1,t));
}
```

Je parcours le tableau en mettant à jour une variable `maxi` *qui contient le maximum jusqu'à présent.*

```
int maximum2(int[] t) {
  int maxi = t[0];
  for (int i = 1; i < t.length; i = i + 1) {
    if (t[i] > maxi) {
      maxi = t[i];
    }
  }
  return maxi;
}
```

itératif

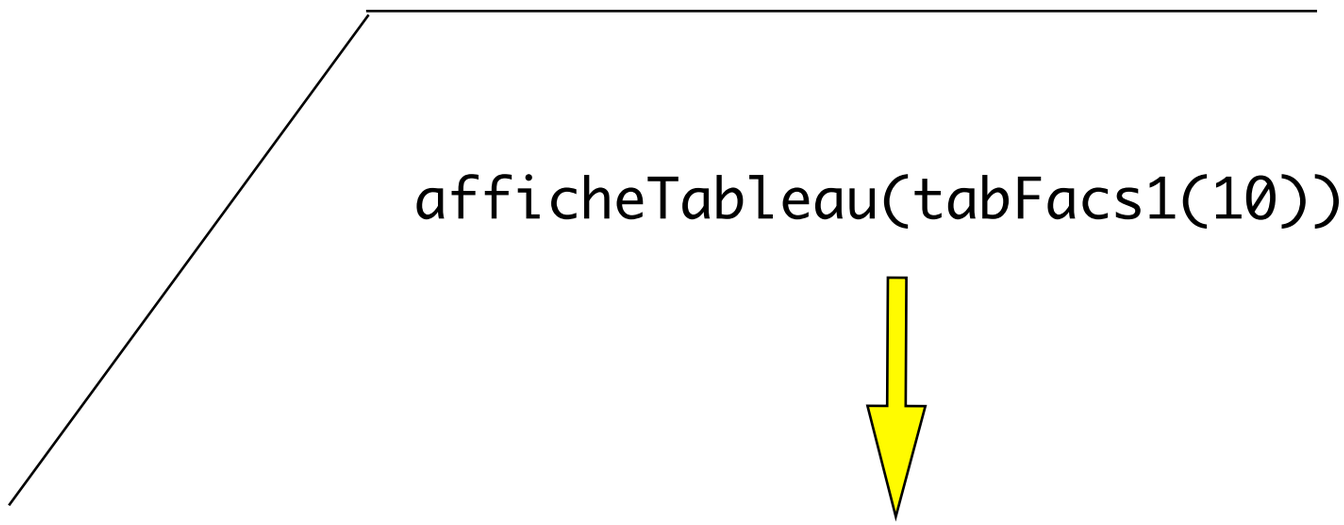
Rendre un tableau en résultat d'une fonction

- Jusqu'ici nos fonctions prenaient un tableau en argument. Elles peuvent aussi **rendre un tableau en résultat**.

- Exemple : **calculer le tableau des factorielles** de `0!` jusqu'à `n!`

```
int[] tabFacs1(int n) {          // n ≥ 0, on utilise une fonction fac
  int[] res = new int[n+1];
  for (int i = 0; i <= n; i = i + 1) {
    res[i] = fac(i);
  }
  return res;
}

int fac(int n) {
  if (n <= 0) return 1;
  return n * fac(n-1);
}
```

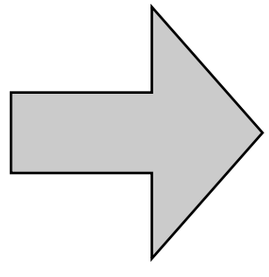
```
affiche Tableau(tabFacs1(10));
```



```
1 1 2 6 24 120 720 5040 40320 362880 3628800
```

- L'utilisation de la méthode `fac` n'est pas une très bonne idée sur le plan de la complexité, puisque l'algorithme est alors en **O(n²)**...

`fac(1)` ⟶ *1 multiplication*
`fac(2)` ⟶ *2 multiplications*
`fac(3)` ⟶ *3 multiplications*
...
`fac(n)` ⟶ *n multiplications*



`tabFacs(n)` ⟶ *n(n+1)/2 multiplications*

- Une **optimisation** consiste à profiter du calcul de `(k-1)!` pour calculer `k!`, d'où une complexité qui devient **O(n)**...

```
int[] tabFacs2(int n) {          // n ≥ 0, sans utiliser de fonction fac
  int[] res = new int[n+1];
  int f = 1;
  for (int i = 0; i <= n; i = i + 1) {
    res[i] = f;                  // f = i!
    f = f * (i + 1);             // f = (i + 1)!
  }
  return res;
}
```

Animation d'une onde sinusoïdale

• Profitons de *Processing*. Visualisons le sinus en associant à chaque ligne horizontale du canvas un niveau de gris (0 = noir, 255 = blanc).

• Si `i` varie dans `[0;height[`, alors `2πi/height` varie dans `[0;2π[` et la valeur absolue de son sinus varie dans `[0,1]`. Remplissons le tableau `onde` dans la fonction `setup()` :

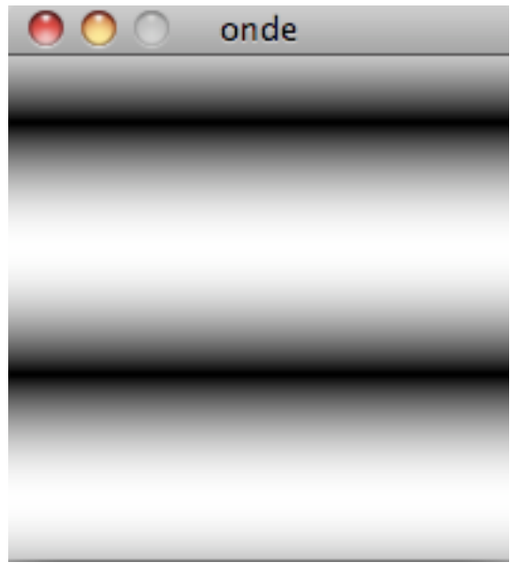
```
float[] onde;      // on n'aura sa longueur qu'après l'instruction size !
int y;             // l'ordonnée courante

void setup() {
  size(200,200);
  y = 0;
  onde = new float[height];
  for (int i = 0; i < height; i = i + 1) {
    onde[i] = abs(sin(2 * PI * i / height)) * 255;
  }
}
```

`tab[i] ∈ [0,255]`

- Dans la fonction `draw()`, à chaque image, on efface le canvas, puis on affiche le tableau sous la forme de niveaux de gris, à partir de l'ordonnée y, qui est incrémentée.

- On obtient ainsi l'illusion d'une **onde animée** :

```
void draw() {
  background(255,255,255);
  // affichage du tableau à partir de l'ordonnée y
  for (int i = 0; i < height; i = i + 1) {
    float gris = onde[i];
    stroke(gris, gris, gris);
    line(0,(i+y) % height,width,(i+y) % height);
  }
  y = (y + 1) % height;
}
```

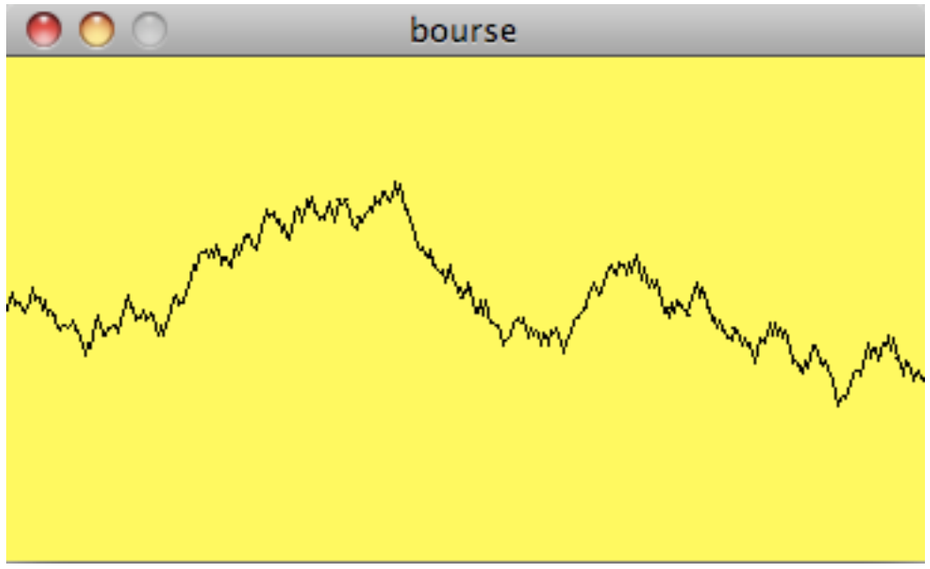


- Le fait de calculer le tableau dans la fonction `setup()` évite de lourds calculs trigonométriques à chaque image de l'animation...

Fluctuation aléatoire d'un cours de bourse

- Chez les boursiers, **l'analyse technique** consiste à étudier les propriétés graphiques de l'évolution du cours de bourse d'une action.

- Ecrivons un programme qui génère aléatoirement l'évolution d'un cours de bourse, sur 365 jours.



x ∈ [0,364]

- Nous utilisons un tableau `bourse` de longueur 365, et `bourse[x]` contient le cours au jour x. Le cours initial est à la demi-hauteur, et le cours du jour `x+1` est issu d'une petite variation du cours du jour x.

```
float[] bourse;

void generation() {                                    // les cours sur une année
  bourse[0] = height/2;
  for (int x = 1; x < width; x = x + 1) {
    bourse[x] = bourse[x-1] + random(10) - 5;          // variation journalière
  }
}

void plot() {                                          // tracé de la courbe
  for (int x = 0; x < width-1; x = x + 1) {            // comme suite...
    line(x, bourse[x], x+1, bourse[x+1]);              // ... de segments
  }
}

void setup() {
  size(365,200);
  background(255,255,0);
  bourse = new float[width];
  generation();
  plot();
}
```