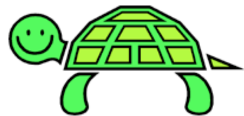
Programmation par Objets - Printemps 2010
Frédéric Mallet & Jean-Paul Roy
L1 Info&Math - Faculté des Sciences de Nice

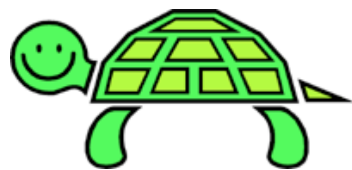
http://deptinfo.unice.fr/~roy

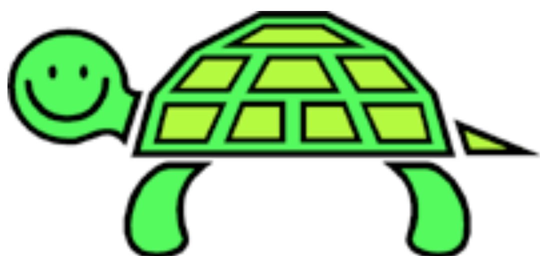
Cours n°3

Calculs répétitifs (1)

la récurrence







Qu'est-ce qu'un calcul **récursif** ?

- Comment calculer la factorielle n! d'un entier $n \geq 0$?

  Par une **récurrence** (comme en maths) :

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

$$0! = 1$$
$$n! = n \times (n-1)! \quad \text{si } n > 0$$

```
void setup() {
  println("fac(10) = " + fac(10));
}
```

```
fac(10) = 3628800
```

- Méthode **récursive** ⇔ programmée **par récurrence**.

http://fr.wikipedia.org/wiki/Raisonnement_par_récurrence

- Par quel mécanisme l'ordinateur peut-il **comprendre une récurrence** ?
- Examinons ce que nous ferions à la main :

pour calculer	*je dois calculer*	*et ensuite je devrai*
fac(4)	fac(3)	multiplier par 4
fac(3)	fac(2)	multiplier par 3
fac(2)	fac(1)	multiplier par 2
fac(1)	fac(0)	multiplier par 1
fac(0)	1	faire toutes les multiplications en attente !

- Le mécanisme consiste donc à **mettre plein de calculs en attente**, puis à les effectuer lorsqu'on aboutit à un certain cas élémentaire.

- Intellectuel non ?...

- Comme les matheux, les programmeurs peuvent utiliser ce schéma de calcul. Mais ils en ont un autre : l'itération.

- Un autre exemple récursif : **calcul de x^n**.

VERSION 1 : par une **récurrence usuelle** (passage de n-1 à n)

```
float puissance(float x, int n) {
  if (n == 0) {
    return 1;
  } else {
    if (n > 0) {
      return x * puissance(x,n-1);
    } else {
      return 1 / puissance(x,-n);
    }
  }
}
```

$x^0 = 1$

$x^n = x \times x^{n-1}$ si $n > 0$

$x^n = 1/x^{-n}$ si $n < 0$

```
float puissance(float x, int n) {
  if (n == 0)  return 1;
  if (n > 0) return x * puissance(x,n-1);
  return 1 / puissance(x,-n);
}
```

Version plus agréable à lire mais plus dangeureuse (`else` *implicites à cause des* `return` *!*)

VERSION 2 : par une **récurrence dichotomique** (passage de n/2 à n)

$$x^0 = 1 \quad \text{si } n = 0$$
$$x^n = 1/x^{-n} \quad \text{si } n < 0$$
$$x^n = (x^{n/2})^2 \quad \text{si } n > 0 \text{ et pair}$$
$$x^n = x(x^{n/2})^2 \quad \text{si } n > 0 \text{ et impair}$$

$2^{11} = 2 * (2^{10})^2$

```
float puissanceDicho(float x, int n) {
  if (n == 0) return 1;
  if (n < 0) return 1 / puissanceDicho(x,-n);
  float hr = puissanceDicho(x,n/2);    // hyp. de récurrence
  if (n % 2 == 0) return hr * hr;
  return x * hr * hr;
}
```

- Dans la version 1, **l'hypothèse de récurrence** était : je suppose que je sais calculer `puissance(x,n-1)`. Dans la version 2, elle devient : je suppose que je sais calculer `puissance(x,n/2)`.

- Pour n > 0, les transformations $n \mapsto n-1$ et $n \mapsto n/2$ font converger n vers 0 (cas de base).

• Intéressons-nous à la **COMPLEXITE DU CALCUL** : quel est le nombre d'opérations effectuées par chaque algorithme ?

VERSION 1 : par une **récurrence usuelle** (passage de n-1 à n)

Le nombre d'opérations est clairement d'ordre n. On dit qu'on a un **coût O(n)**.

$$x^0 = 1$$
$$x^n = x \times x^{n-1} \quad si\ n > 0$$

La signification mathématique précise de O(n) vous sera expliquée au semestre 2.

VERSION 2 : par une **récurrence dichotomique** (passage de n/2 à n)

Le nombre d'opérations est du même ordre que le nombre de fois qu'on peut diviser n par 2 avant de tomber sur 0.

$$x^0 = 1 \quad si\ n = 0$$
$$x^n = (x^{n/2})^2 \quad si\ n > 0\ et\ pair$$
$$x^n = x(x^{n/2})^2 \quad si\ n > 0\ et\ impair$$

Or $n \approx 2^a$ se résoud en $a \approx \log_2 n$

L'algorithme par dichotomie a donc un **coût O(log n)**.

• Un exemple encore plus récursif : la **suite de Fibonacci**. Ce que les matheux nomment une **récurrence double** :

```
int fib(int n) {                    // n >= 0
  if (n < 2) return n;
  return fib(n-1) + fib(n-2);
}
```

$$f_0 = 0$$
$$f_1 = 1$$
$$f_n = f_{n-1} + f_{n-2} \quad si \quad n \geq 2$$

```
void timeFib(int n) {
  float start = millis();
  int res = fib(n);
  println("fib(" + n + ") = " + res + " (Time = " + (millis() - start) + "ms)");
}
```

```
timeFib(20);
timeFib(30);
timeFib(40);
```

```
fib(20) = 6765 (Time = 1.0 ms)
fib(30) = 832040 (Time = 20.0 ms)
fib(40) = 102334155 (Time = 2376.0 ms)
```

Ouf !

Complexité exponentielle !

Apprendre à raisonner par récurrence

- Discipline de longue haleine, vitale aussi bien pour le mathématicien (preuves) que pour le programmeur (algorithmes) !

- L'idée consiste à **ramener le problème courant à un problème identique mais portant sur des données plus petites**.

- Cette idée est un cas particulier d'une idée plus générale qui fait la gloire des scientifiques et des militaires :

DIVISER POUR REGNER

- La bonne approche consiste en effet (presque toujours) à **casser le problème en sous-problèmes plus simples** !

- Récurrence : le sous-problème est identique au problème de départ !

EXEMPLE DETAILLE DE RECURRENCE SIMPLE

- Soit à calculer la somme des entiers de l'intervalle [350, 2010]. On commence par **généraliser le problème** en travaillant dans un intervalle [a,b] quelconque. Soit donc à calculer S(a,b) = a + (a+1) + ... + b

$$S(a,b) = \sum_{a \leq k \leq b} k$$

- Je raisonne par **récurrence sur l'entier a** en distinguant le **cas de base** et le **cas général** :

Cas de base : si a > b. Alors S(a,b) = 0 car {k : a ≤ k ≤ b} est vide !

Cas général : si a ≤ b. Alors S(a,b) = a + (a+1) + ... + b = a + S(a+1,b)

sous-problème identique !

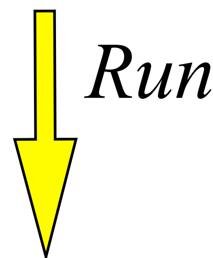
- On a donc réussi à **ramener un calcul S(...) à un autre calcul S(...) plus simple** : c'est l'essence de la récurrence !

• Il reste maintenant à traduire cette **analyse récursive** en un **programme** exécutable :

```
int sommeEntiers(int a, int b) {
  if (a > b) return 0;
  return a + sommeEntiers(a+1,b);
}
```

$$S(a,b) = 0 \quad si\ a > b$$
$$S(a,b) = a + S(a+1,b) \quad si\ a \leq b$$

```
println("sommeEntiers(350,2010) = " + sommeEntiers(350,2010));
```



Run

```
sommeEntiers(350,2010) = 1959980
```

• L'erreur courante consiste à poser comme cas de base a = b. Le programme devient alors faux (si a > b).

• La complexité est en O(b-a) opérations.

Diverses formes de récurrence

➊ Dans tous les cas, il faut trouver **le cas de base**, vers lequel le cas général va converger.

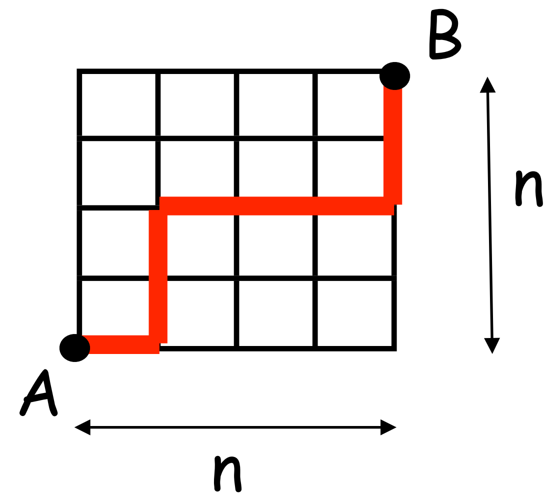
➋ Trois manières de formuler **l'hypothèse de récurrence** :

simple : on passe de n-1 à n. C'est en général ce que font les matheux, et aussi souvent les programmeurs.

dichotomique : on passe de n/2 à n. Bonne accélération d'algorithmes pour les programmeurs.

forte : on suppose que l'on sait calculer pour 0,1,2,...,n-1 et l'on montre que l'on sait alors calculer pour n. La récurrence double en est un cas particulier !

EXEMPLE DETAILLE DE RECURRENCE DOUBLE



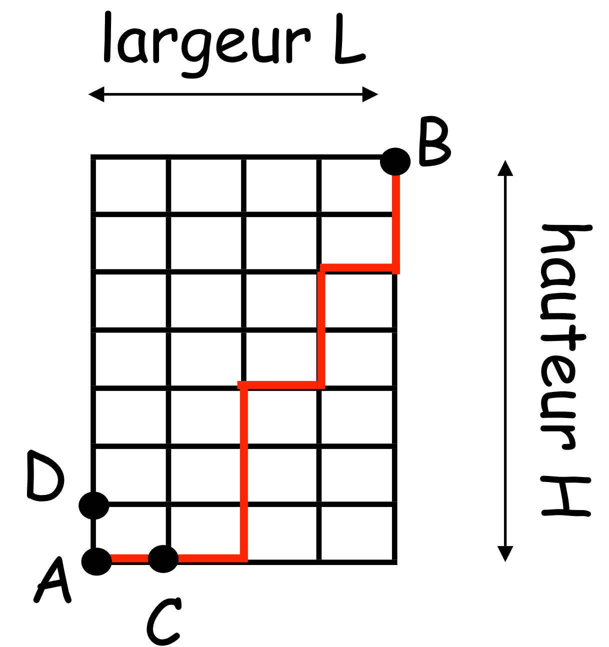
- Un robot parcourt un **monde carré de côté n**.
Il doit se rendre du point A au point B en suivant un chemin de longueur minimale 2n.
Combien de chemins possibles ?

- Je tente une récurrence simple :

```
int nbCheminsCarre(int n) {
  if(n == 0) return 1;
  int hypoRec = nbCheminsCarre(n - 1);
  ???
}
```

- Même si je suppose que je sais résoudre le problème dans un monde plus petit de côté n-1, je n'arrive pas à en déduire la solution pour un monde de côté n. **La récurrence simple résiste !**

• Je vais donc **relaxer les données** et supposer que je travaille dans un rectangle de largeur L et de hauteur H.



• Pour aller de A vers B, je dois passer par C ou par D... et je me retrouve encore dans un problème rectangulaire plus simple ! *D'où la récurrence...*

```
int nbCheminsRect(int largeur, int hauteur) {
  if(largeur == 0 || hauteur == 0) return 1;
  return nbCheminsRect(largeur - 1,hauteur)        // par C
       + nbCheminsRect(largeur,hauteur - 1);       // par D
}
```

• Et le monde carré n'est alors qu'un cas particulier de monde rectangulaire !

```
int nbChemins(int n) {           // le carré comme cas particulier !
  return nbCheminsRect(n,n);
}
```

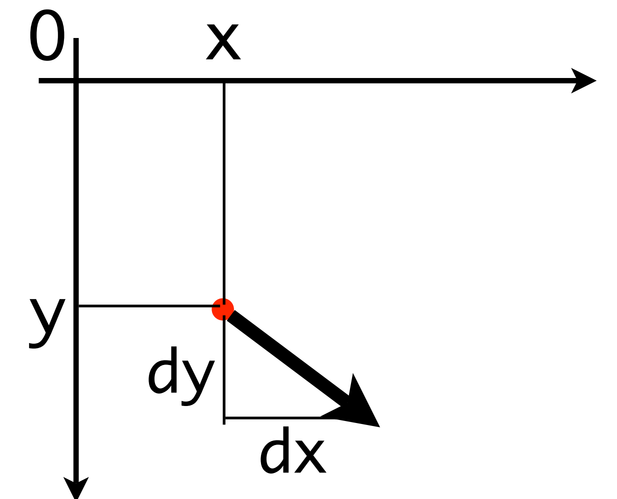
Et le graphisme dans tout ça ?...

• Il va remontrer le bout du nez avec la **TORTUE** !

• Nous allons modéliser cet animal virtuel, capable de deux actions élémentaires dans un canvas :

avancer d'une distance d	`avance(d);`
tourner à droite d'un angle a donné	`droite(a);`

• La bestiole sera à tout moment repérée dans le système orthonormé du canvas, par sa **position** **(x,y)** et son **vecteur-vitesse** **(dx,dy)**. Lorsqu'elle fait un pas, elle effectue une translation de vecteur (dx,dy).



• Notre programme va donc travailler sur 4 **champs**, qui seront des nombres approchés x, y, dx, dy.

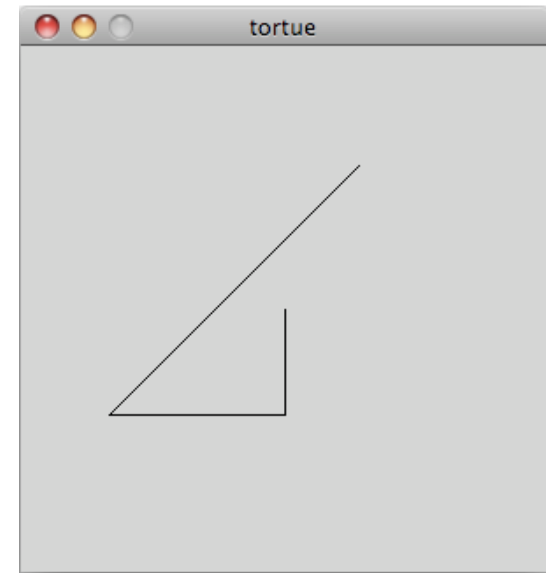
```
float x, y, dx, dy;                  // position et vitesse de la tortue

void avance(float d) {
  // avance d'une distance d dans la direction du vecteur vitesse
  float xSuiv = x + d * dx;          // calcul de la position suivante
  float ySuiv = y + d * dy;          //    dans deux variables locales xSuiv, ySuiv
  line(x,y,xSuiv,ySuiv);             // tracé d'un segment
  x = xSuiv;                         // mise à jour de la position
  y = ySuiv;
}

void droite(float a) {
  // mise à jour du vecteur vitesse par rotation d'angle a.
  // Attention à l'orientation des axes informatiques !
  float cosa = cos(a);               //    Pour ne pas les recalculer !
  float sina = sin(a);
  float dxSuiv = dx * cosa - dy * sina;
  float dySuiv = dx * sina + dy * cosa;
  dx = dxSuiv;
  dy = dySuiv;
}
```

rotation d'angle a

$$z' = e^{ia} z$$

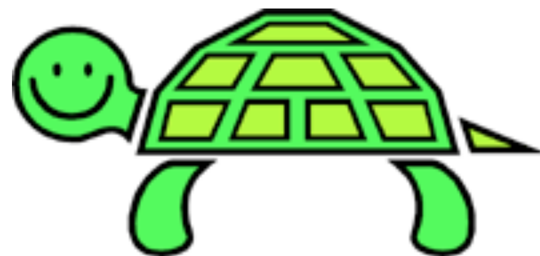
$$x' + iy' = (\cos a + i \sin a)(x + iy)$$

• Testons notre programme jusqu'à présent, n'attendons pas d'avoir écrit des centaines de lignes :




```
void setup() {
  size(300,300);
  x = width/2;          // point de départ au centre
  y = height/2;
  dx = 0;               // vitesse initiale vers le Sud
  dy = 1;
  avance(60); droite(PI/2); avance(100); droite(3*PI/4); avance(200);
}
```

• Cette idée de tortue pour dessiner a été popularisée avec le langage LOGO, très utilisé par les profs d'écoles avec les enfants, pour à la fois leur enseigner la programmation, et les spatialiser.

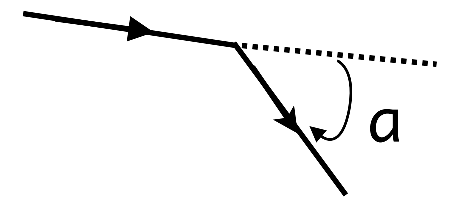


• Mettez les mains dans cet embryon de géométrie tortue, et ajoutez toutes les fonctionnalités que vous voudrez ! Soyez créatifs !

• Par exemple, programmons le dessin à la tortue d'un **polygone régulier de n côtés**, chaque côté ayant une longueur c.

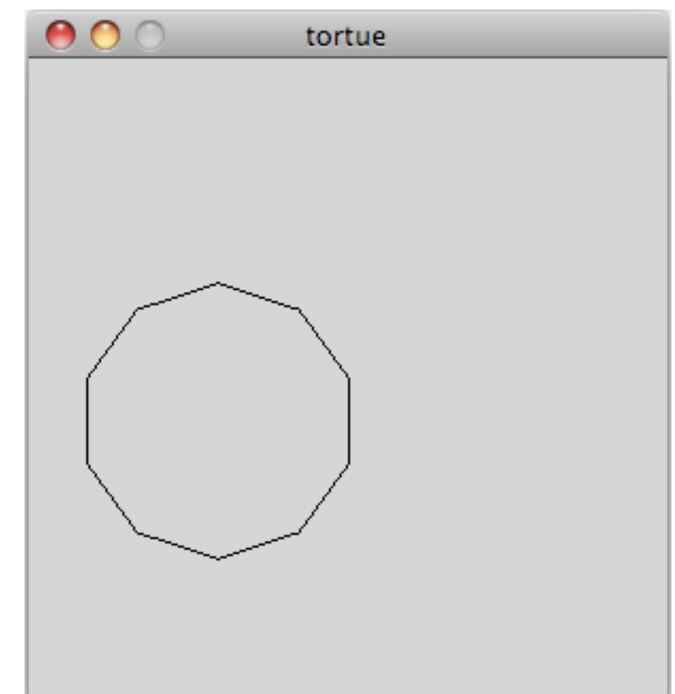
• La récurrence simple sur n ne fonctionne pas : je ne sais pas déduire un polygone régulier de n côtés à partir d'un polygone régulier de n-1 côtés. Je vais plutôt raisonner par **récurrence sur le nombre de côtés déjà tracés** ! Soit a l'angle dont tourne la tortue à chaque côté.

```
void polyRec(float a, int nbCotes, float c) {
  if(nbCotes == 0) return;
  avance(c);
  droite(a);
  polyRec(a, nbCotes - 1, c);
}
```



• La méthode finale `poly(nbCotes, c)` s'en déduit en cas particulier.

```
void poly(int nbCotes, float c) {
  polyRec(2*PI/nbCotes,nbCotes,c);
}
```



Variables Locales et Champs

• Dans une première approche, un programme informatique comprend deux sortes de variables :

- les **variables locales**, qui n'ont de sens que dans une portion limitée du programme. Les paramètres sont des variables locales à une méthode.

- les **champs**, qui existent dans la totalité du texte du programme.

• Les variables `x`, `y`, `dx`, `dy` étaient des ***champs***. Toutes les méthodes du programme peuvent s'en servir, y faire référence, les modifier.

• Les variables `xSuiv`, `ySuiv` de la méthode `avance()` étaient ***locales à cette fonction***, de même que son paramètre `d`. Les autres méthodes n'y ont pas accès, elles ont même le droit d'avoir leurs propres variables locales de même nom !

- Plus précisément, une variable locale est déclarée au sein d'un bloc.
- RAPPEL : un bloc est une suite d'instructions entre accolades :

```
{
    avance(50);
    droite(PI/3);
    float dist = sqrt(234);
    avance(dist);
    println("Fini !");
}
```

La variable `dist` *n'existe que dans cette portion du bloc !*

- **Une variable locale dans un bloc ne peut être référencée qu'entre sa déclaration et la fin du bloc**. Autre exemple typique :

```
if (x > 0) {
    int y = x + 1;
    ...
} else {
    ...
}
```

La variable y n'existe que dans ce bloc !

Ici y n'existe pas !