

INF120 - Algorithmique

Page web du cours : <http://tinyurl.com/INF120-2011S1>

• Contact

- Courriel : philippe.gambette@gmail.com
(INF120 doit apparaître dans le sujet du courriel)
- Avant ou après le cours

• Matériel

- Ordinateur portable : interdit pendant les TP, a priori inutile en cours et TD.
- Pas de téléphone portable pendant cours/TD/TP

• Déroulement des enseignements

- Pause (méritée) de 5 minutes entre deux heures de TD
- Passages au tableau pour les corrections d'exercices en TD

• Sources

Le livre de Java premier langage, A. Tasso
<http://www.pise.info/algo/introduction.htm>
Cours INF120 de J.-G. Luque
<http://xkcd.com>, <http://xkcd.free.fr>

• Notes et devoirs

- Interrogations QCM en début de cours ou TD
(signalement des absences pour rattrapage, voir intranet)
- Un devoir maison

• Note finale

- Prévision : environ 2/3 “compétences”, environ 1/3 “motivation”
- Compétences : 2/3 devoir final (11 janvier 2012), 1/3 QCM
- Motivation : devoir maison, exercices, TP

• Exercices supplémentaires d'entraînement


- Sur demande, par courriel
- Sur demande, possibilité d'organiser une séance d'exercices ou de préparation au devoir final.

<http://serecom.univ-tln.fr/cours/index.php/Algorithmie>
Cours de J. Henriet :
<http://julienhenriet.olymp-network.com/Algo.html>

A quoi sert un algorithme ?

- À décrire les étapes de résolution d'un problème :
 - de façon structurée et compacte (méthode **facile à comprendre, facile à transmettre**)
 - à partir d'opérations de base (méthode **adaptée aux moyens à disposition, adaptée aux connaissances de celui qui l'utilise**)
 - indépendamment d'un langage de programmation (méthode **adaptée pour des problèmes qui se traitent sans ordinateur, compréhensible sans apprendre un langage de programmation**)

(“étapes” aussi appelées “pas de l'algorithme”)

- Problème : données en entrée  résultat en sortie

- L' « *algorithme des crêpes* »

Ingrédients : beurre, oeufs, sachets de sucre vanillé, farine, lait, sel

Récipients : saladier, verre mesureur, poêle, assiette,

Opérations de base : **mettre dans un récipient**, **mélanger**, **attendre pendant ... minutes**, **retourner**, **laisser cuire pendant ... minutes**

Algorithme des crêpes :

Mettre 4 oeufs **dans** le saladier

Mettre 1 sachet de sucre vanillé **dans** le saladier

Mettre 250 g de farine **dans** le verre mesureur

Mettre le contenu du verre mesureur **dans** le saladier

Mettre 0,5 litre de lait **dans** le verre mesureur

Mettre le contenu du verre mesureur **dans** le saladier

Mettre 50 grammes de beurre **dans** la poêle

Laisser cuire la poêle **pendant 1 minute**

Mettre le contenu de la poêle **dans** le saladier

Mélanger le contenu du saladier

Attendre pendant 60 minutes

Mettre 5 grammes de beurre **dans** la poêle

Algorithmes sans ordinateurs :

- Euclide (vers -300) : calcul du PGCD de 2 nombres
- Al-Khwarizmi (825) : résolution d'équations
- Ada Lovelace (1842) : calcul des nombres de Bernoulli sur la *machine analytique* de Charles Babbage

Laisser cuire la poêle **pendant 0.5 minute**

Tant que le saladier n'est pas vide :

Mettre 5 cL du contenu du saladier **dans** le verre mesureur

Mettre le contenu du verre mesureur **dans** la poêle

Laisser cuire la poêle **pendant 2 minutes**

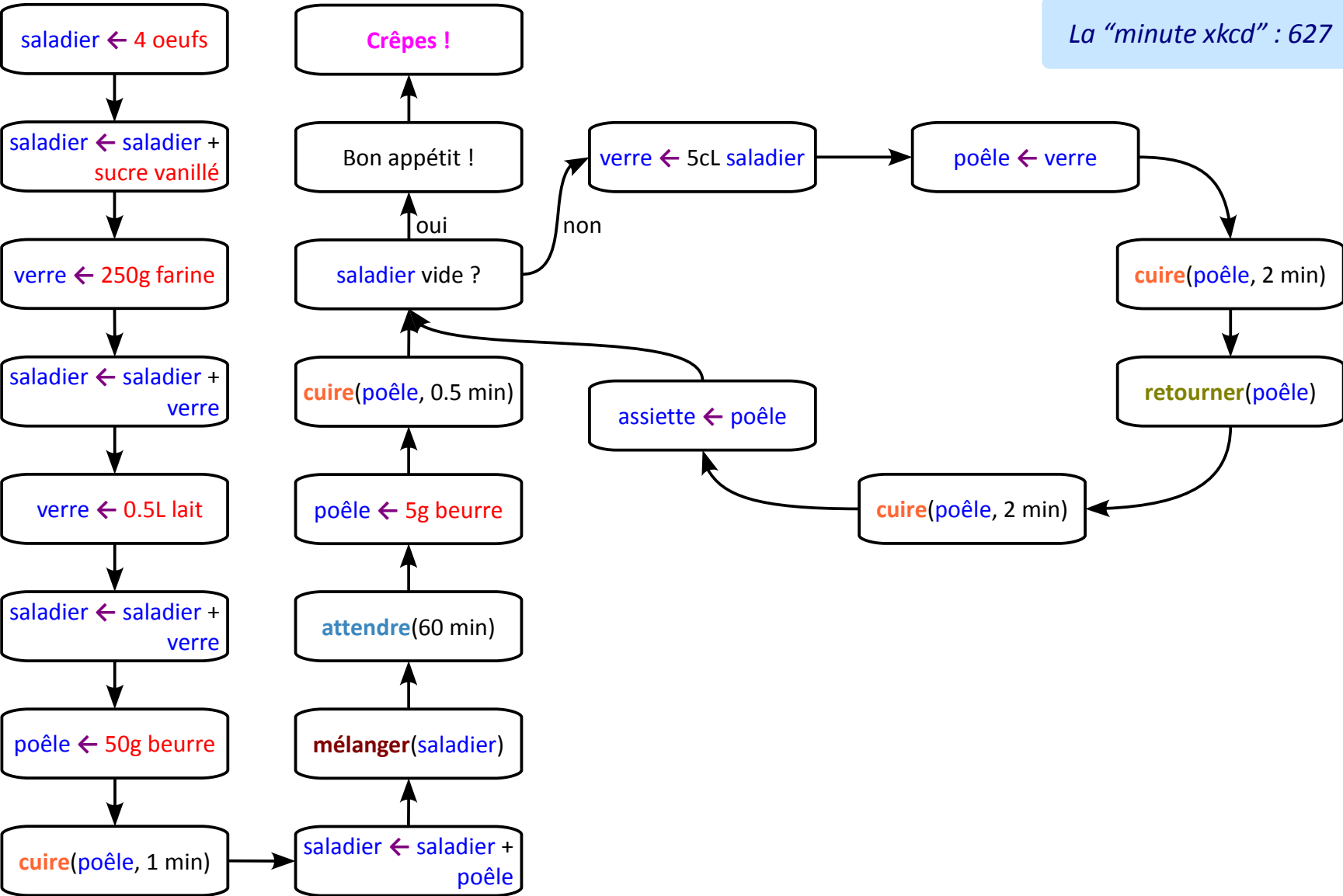
Retourner le contenu de la poêle

Laisser cuire la poêle **pendant 2 minutes**

Mettre le contenu de la poêle **dans** l'assiette

Organigramme de la recette des crêpes

La "minute xkcd" : 627



Enjeux algorithmiques : terminaison, correction, complexité

Terminaison : L'algorithme se termine-t-il ?

Preuve de terminaison :

- critère qui diminue à chaque boucle de l'algorithme
→ quantité de pâte à crêpes
- plus formellement : suite à **valeurs entières positives, strictement décroissante** à chaque boucle de l'algorithme

La "minute mathématique"

Théorème :
Toute suite à valeurs entières positives, strictement décroissante, ne peut prendre qu'un nombre fini de valeurs.

Application :
Trouver la suite qui convient pour prouver la terminaison d'un algorithme.

La "minute votes SMS"

Problème : aller en voiture de Châtelet à la Tour Montparnasse

Algorithme "du repère visuel" :
A tout instant on sait où se trouve la Tour Montparnasse
→ prendre la rue qui s'en rapproche le plus

NE TERMINE PAS NECESSAIREMENT

Correction : L'algorithme donne-t-il le résultat attendu ?

Preuve de correction :

- « invariant » : propriété vraie tout au long de l'algorithme
 - vraie à la première étape
 - si vraie à une étape, vraie à l'étape suivante
- ⇒ vrai à la fin

En pratique, pour débiter :

- vérifier sur les "cas de base"
- vérifier sur des exemples aléatoires

Complexité : Combien de temps l'algorithme prend-il pour se terminer ?

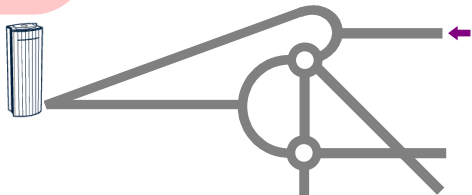
Théorie de la complexité :

- évaluer le nombre d'opérations en fonction de la taille du problème, dans le pire cas
- prouver qu'on ne peut pas utiliser moins d'opérations pour résoudre le problème, dans le pire cas

En pratique :

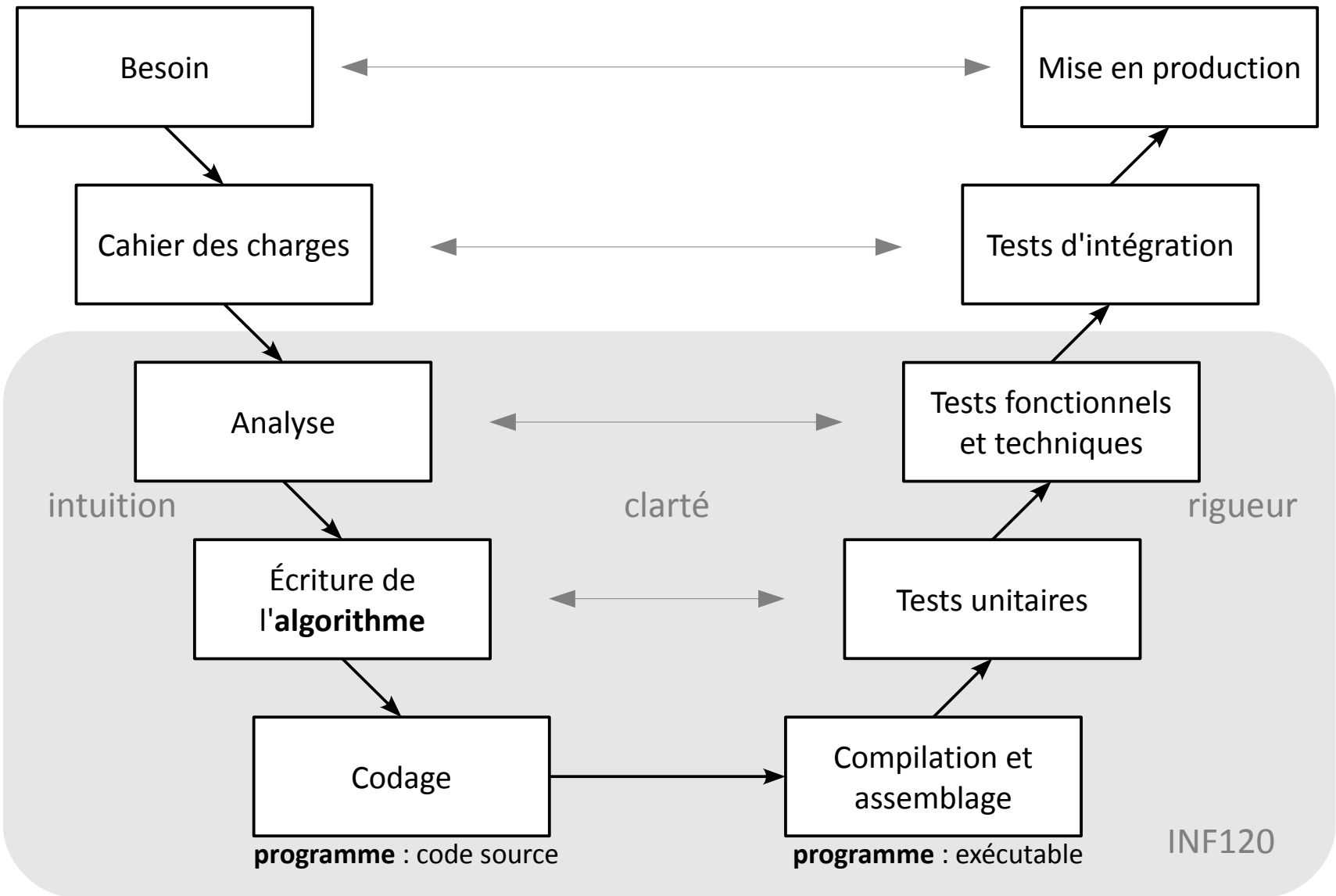
- vérifier sur des exemples aléatoires
- connaître les cas difficiles

Exige-t-on que tout bon algorithme se termine ?
Oui en général (exceptions : algorithme de contrôle d'un ascenseur, de transactions financières, ...)

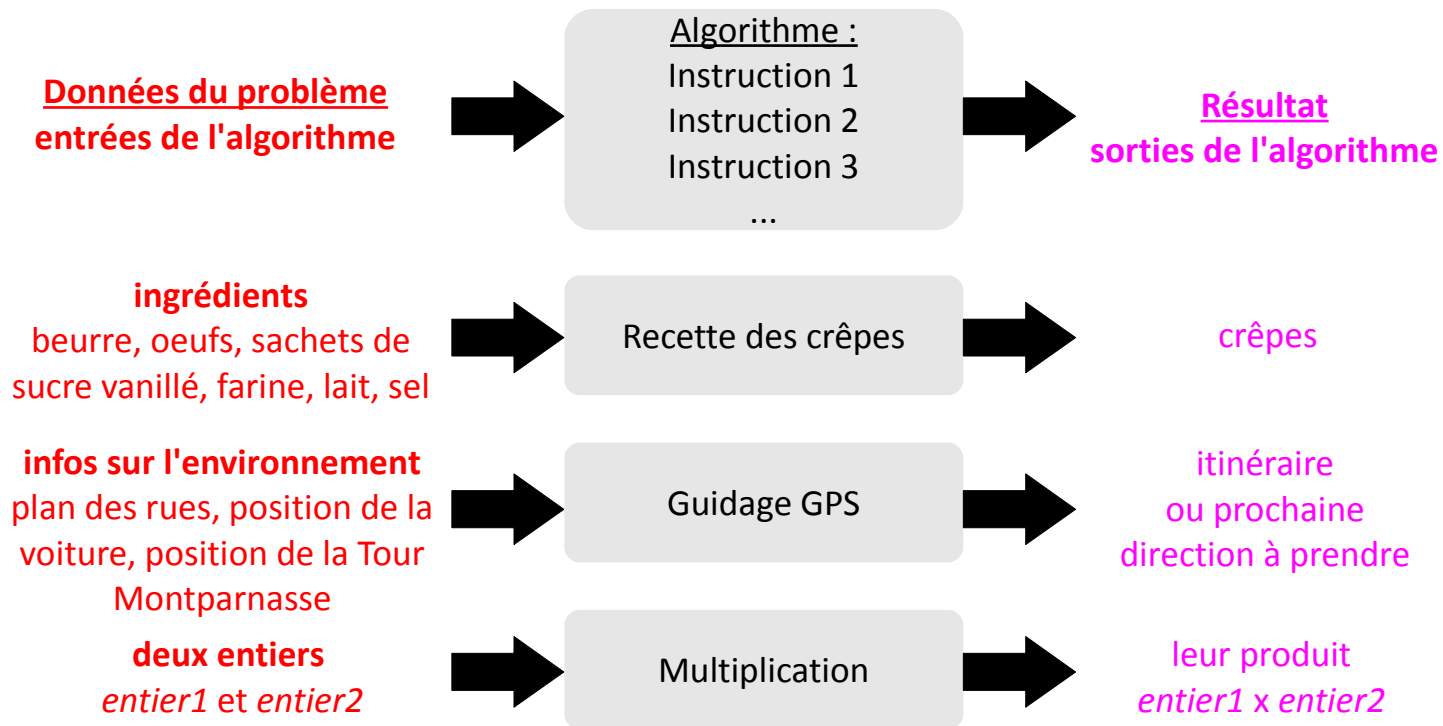


algorithme **glouton** :
toujours choisir le profit maximum.
Pas toujours la meilleure stratégie.

De l'algorithme au programme



Composants d'un algorithme



Divers **types d'instructions** d'un algorithme :

- déclaration d'un algorithme
- appel d'un algorithme
- déclaration d'une **variable**
- **affectation** d'une **variable**
- **entrées** / **sorties**
- **boucle**
- test

Variables et affectation

Dans un algorithme, une **variable** possède :

- un **nom**,
- une **valeur**,
- un **type** (ensemble des valeurs que peut prendre la variable).

La **valeur** d'une variable :

- est **fixe à un moment donné**,
- peut **changer au cours du temps**.

L'**affectation** change la valeur d'une variable :

- $a \leftarrow 5$:
 - la variable a prend la valeur 5
 - la valeur précédente est perdue (“écrasée”)
- $a \leftarrow b$:
 - la variable a prend la valeur de la variable b
 - la valeur précédente de a est perdue (“écrasée”)
 - la valeur de b n'est pas modifiée
 - a et b doivent être de même type
(ou de type compatible)

Dans un **algorithme**, choisir pour les variables :

- un nom composé de **lettres** et éventuellement de **chiffres**
- un nom **expressif**, par exemple :
 - *chaîne, requête1...* pour une chaîne de caractères
 - *n, a, compteur, nbOperations, longueur...* pour un entier
 - *x, y, température* pour un réel
 - *estEntier, testEntier, trouvé...* pour un booléen
- un nom **assez court** (il faut l'écrire !)
- éviter les **noms réservés** : *pour, tant que, si...*

Dans un **programme** :

- **éviter** les lettres accentuées et la ponctuation
- préférer l'**anglais** si votre code source est diffusé largement
- être **expressif** et **lisible** :
 - *est_entier* ou *estEntier* plutôt que *estentier*

Votre code sera relu, par vous ou par d'autres...

En **Java** le nom de variable doit commencer par une lettre, de préférence minuscule. Il peut contenir des lettres, des chiffres, et les symboles \$ et _. Il est sensible à la casse (majuscules/minuscules).

Mon premier vrai algorithme

Je connais l'algorithme d'**addition** de deux entiers positifs.
Comment écrire un algorithme de **multiplication** de deux entiers ?

Intuition : $5 \times 3 = \underbrace{5 + 5 + 5}_{3 \text{ fois}}$

$entier1 \times entier2 = \underbrace{entier1 + entier1 + entier1 + \dots + entier1}_{entier2 \text{ fois}}$

Multiplication :

- Entrées :** deux entiers *entier1* et *entier2*
- Sorties :** le **produit** de *entier1* et *entier2*
- Variables :** entiers *compteur* et *produit*

```
Début
  compteur ← 0
  produit ← 0
  Tant que compteur < entier2 faire :
    produit ← addition(produit, entier1)
    compteur ← addition(compteur, 1)
  Fin tant que
  renvoyer produit
Fin
```

Terminaison ? Correction ?

Essayer avec *entier1* = 5 et *entier2* = 3

Preuve dans un prochain cours...

Valeurs avant la *i*-ième boucle *Tant que* :

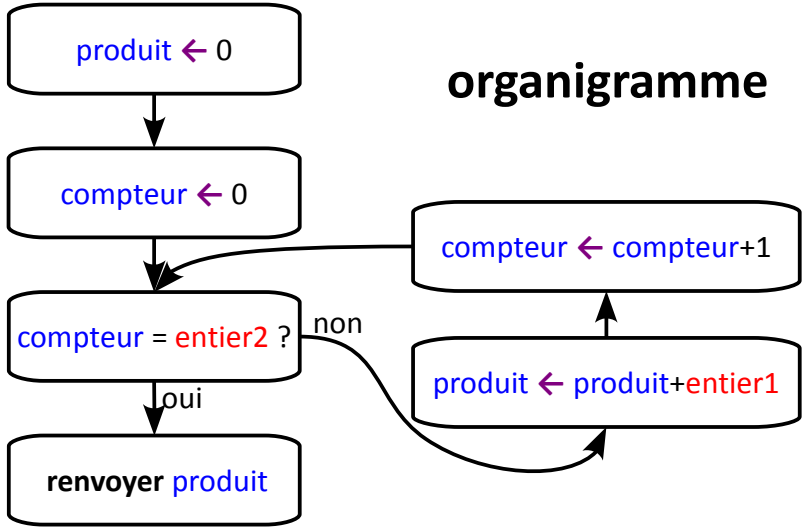
<i>i</i>	1	2	3	4	5
<i>compteur</i>	0	1	2	3	-
<i>produit</i>	0	5	10	15	-

Remplir ce tableau, c'est **faire la trace de l'algorithme** : savoir à chaque moment la valeur de toutes les variables

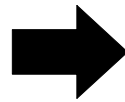
Dictionnaire pseudo-code / Java

	Pseudo-code	Java
Déclaration d'un algorithme	Addition Entrées : entiers i et j Sortie : somme $i+j$ (entier) Début ... renvoyer ... Fin	public static int Addition (int i , int j){ ... return ... }
Déclaration d'une variable	Variables : entier i	int i ;
Affectation	$i \leftarrow 1$	$i = 1$;
Test	Si $i=1$ alors ... Sinon ... FinSi	if ($i==1$){ ... } else { ... }
Boucle	Tant que $i<3$: ... Fin TantQue	while ($i<3$) { ... }

De l'organigramme au code Java

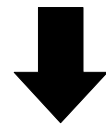


organigramme



Multiplication : **pseudo-code**

Entrées : deux entiers *entier1* et *entier2*
Sorties : le *produit* de *entier1* et *entier2*
Variables : entiers *compteur* et *produit*
Début
compteur ← 0
produit ← 0
Tant que *compteur* < *entier2* faire :
 produit ← addition(*produit*, *entier1*)
 compteur ← addition(*compteur*, 1)
Fin tant que
renvoyer produit
Fin



```
public static int multiplication(int entier1, int entier2){  
  
int compteur=0;  
int produit=0;  
while (compteur<entier2){  
    produit=produit+entier1;  
    compteur=compteur+1;  
}  
return produit;  
  
}
```

code Java

Codage binaire

La "minute votes SMS"

Programme Java :

```
import java.io.*;
public class Boucle{
    public static void main(String [] arg){
        int i=1;
        while(i>0){
            i=i*2;
        }
        System.out.print("J'ai fini !");
    }
}
```

Le programme affiche "J'ai fini" parce que les entiers "int" Java ne sont pas de "vrais entiers" mais des entiers entre -2 147 483 648 et 2 147 483 647

Résultat du programme :

2	
4	131072
8	262144
16	524288
32	1048576
64	2097152
128	4194304
256	8388608
512	16777216
1024	33554432
2048	67108864
4096	134217728
8192	268435456
16384	536870912
32768	1073741824
65536	-2147483648
	J'ai fini!!

La "minute mathématique"

Pour le stockage comme pour le traitement d'instructions, il est nécessaire que toutes les données traitées par un ordinateur soient codées en **binaire**, par des **0** et des **1**.

Exemple de nombre entier en binaire : 1101100001101

0	0	0	1	1	0	1	1	0	0	0	0	1	1	0	1
2^{15}	2^{14}	2^{13}	2^{12}	2^{11}	2^{10}	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

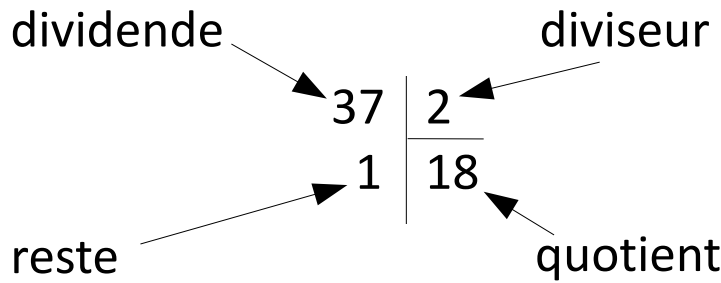
$$2^{12} + 2^{11} + 2^9 + 2^8 + 2^3 + 2^2 + 2^0 = 4096 + 2048 + 512 + 256 + 8 + 4 + 1 = 6925$$

Codage binaire

Pour le stockage comme pour le traitement d'instructions, il est nécessaire que toutes les données traitées par un ordinateur soient codées en **binaire**, par des **0** et des **1**.

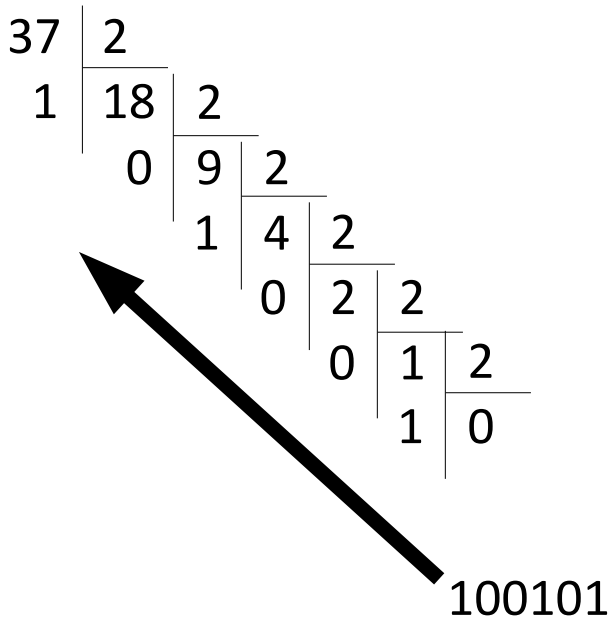
Exemple : écrire 37 en binaire ?

Division euclidienne :



en Java : $1=37\%2$

en Java : $18=37/2$



Pour faire son geek :

- compter sur ses doigts en binaire, jusqu'à 2^{10}
- faire des estimations de nombres données en binaire : $2^{10} = 1024$ donc $2^{10} \approx 1000$

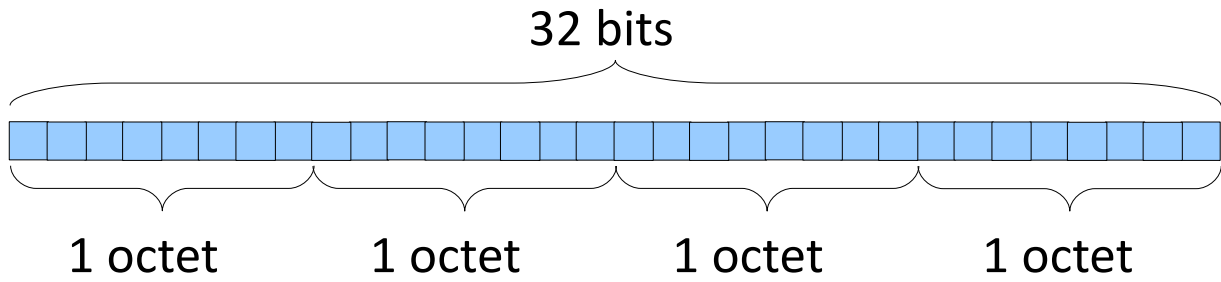
$2^{32} \approx 4$ milliards

$2^{32} = 4\ 294\ 967\ 296$

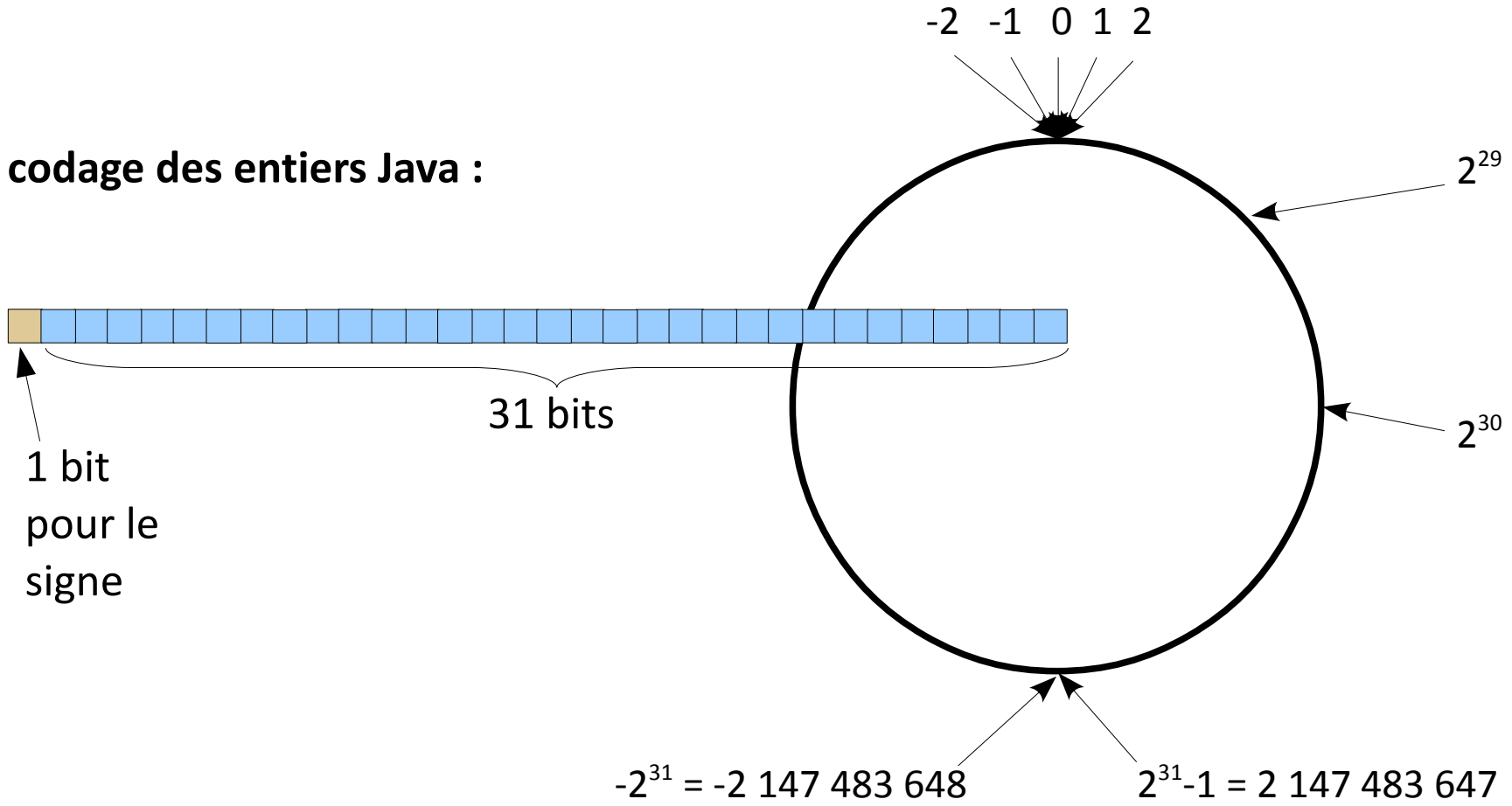
La "minute xkcd" : 953

Codage des entiers 32 bits

- 1 bit = 0 ou 1
- 1 octet = 8 bits
- 1 Ko (kiloctet) = 1024 octets
- 1 Mo (mégaoctet) = 1024 Ko (disquette)
- 1 Go (gigaoctet) = 1024 Mo (carte mémoire, 2h de vidéo en DivX)
- 1 To (téraoctet) = 1024 Go (disque dur externe)



Le codage des entiers Java :



Autres codages

• Chaînes de caractères

- ASCII : 7 bits, caractères simples codés de 32 à 127
- ANSI : 8 bits, caractères simples codés de 32 à 127, caractères accentués de 128 à 255
- UTF-8 : de 1 à 4 octets

• Couleurs d'une image

- RGB : "red, blue, green", 1 octet pour chacun :
- valeurs entre 0 et 255
- codage hexadécimal avec 2 symboles

La "minute culturelle"

Hexadécimal : en base 16 (ἕξάς : six, decem : dix)

Codé par les chiffres de 0 à 9 et les lettres A B C D E F

	A	B	C	D	E	F
	↑	↑	↑	↑	↑	↑
	10	11	12	13	14	15

- Deux symboles pour un octet :
 16^2 valeurs possibles = 256

- Utilisé pour coder les couleurs en HTML :
couleur="#RRGGBB"
rouge="#FF0000", vert="#00FF00"
#800080 ?

Les booléens

- Opérations sur les booléens :
et, ou, non

ET	VRAI	FAUX
VRAI	VRAI	FAUX
FAUX	FAUX	FAUX

OU	VRAI	FAUX
VRAI	VRAI	VRAI
FAUX	VRAI	FAUX

NON	VRAI	FAUX
	FAUX	VRAI

Les opérations de base en Java

- **Type entier `int`**

`+`, `-`, `*`, `/` (division entière), `%` (reste modulo), `^` (puissance)

- **Type flottant `float`, ou `double` (plus précis)**

`+`, `-`, `*`, `/`

- **Type booléen `boolean`**

`false` (faux), `true` (vrai), `&&` (et), `||` (ou), `!` (non)

- **Type chaîne de caractères `String`**

`+` (concaténation : `"INF"+"120"="INF120"`)

Compétences acquises en INF120

Connaître les éléments de base d'un algorithme

en pseudo-code

Un algorithme résout un problème, en fournissant un **résultat en sortie** à partir de **données en entrée**.

Entrées : entiers *i* et *j*
Sortie : somme *i+j* (entier)

Pour cela, il utilise plusieurs types d'**instructions** : *une instruction par ligne !*

- des **affectations** dans des **variables** (mémoires) $\text{produit} \leftarrow \text{entier1} + \text{produit}$
variable valeur
- des **boucles** **Tant que ... faire : ... Fin TantQue**
- des **appels** à d'autres algorithmes $\text{Addition}(3,5)$ { entrées de l'algorithme entre parenthèses, respectant l'ordre de déclaration des entrées
- des **tests** **Si ... alors : ... sinon : ... FinSi**
- des **"lectures"** d'entrées et **"renvois"** de sorties *renvoyer *i+j**

Chaque **variable** a un **nom**. On doit :

- la **déclarer** en définissant son **type** (ensemble de valeurs possibles) *Variables : entiers *a* et *b**
type
 - puis l'**initialiser** (lui donner une **valeur**) $a \leftarrow 2$
 $b \leftarrow a+2$
- avant de l'utiliser.

Compétences acquises en INF120

Connaître les éléments de base d'un algorithme

en Java

Un algorithme résout un problème, en fournissant un **résultat en sortie** à partir de **données en entrée**.

```
public static int Addition(int i, int j){...}
```

instructions terminées par ";" type de sortie entrées précédées de leur type

Pour cela, il utilise plusieurs types d'**instructions** :

- des **affectations** dans des **variables** (mémoires) `produit = entier1+produit`

variable valeur

- des **boucles** `while (...){...}`

- des **appels** à d'autres algorithmes `Addition(3, 5)` { entrées de l'algorithme entre parenthèses, respectant l'ordre de déclaration des entrées

- des **tests** `if(...){...}else{...}`

- des "**lectures**" d'entrées et "**renvois**" de sorties `return i+j;`

Chaque **variable** a un **nom**. On doit :

- la **déclarer** en définissant son **type** (ensemble de valeurs possibles)

```
int a, b;
```

- puis l'**initialiser** (lui donner une **valeur**)

```
a = 2
```

type

avant de l'utiliser.

```
b = a+2
```

Méthodologie pour l'algorithmique

Savoir lire et comprendre un algorithme

Premiers éléments à identifier :

- qu'est-ce que l'algorithme **prend en entrée** ? **Combien** de variables, de quels **types** ?
- qu'est-ce que l'algorithme **renvoie en sortie** ? **Rien** ? Ou bien **une** variable ? De quel **type** ?

Ensuite :

- quels sont les autres **algorithmes appelés** par l'algorithme ?

Enfin :

- faire la **trace** de l'algorithme, c'est-à-dire l'essayer sur un **exemple**
(... ou plusieurs pour passer au moins une fois par toutes les instructions de l'algorithme)
et voir ce que valent **toutes les variables à chaque étape**
(et noter ces valeurs dans un tableau contenant une ligne par variable et une colonne par étape),
- noter en particulier le **résultat obtenu en sortie** pour une **entrée testée**.

Méthodologie pour l'algorithmique

Savoir concevoir un algorithme pour résoudre un problème

Premiers éléments à identifier :

- quels sont les **outils à disposition** ? (pour ces outils : données en entrée, type de données en entrée, résultat en sortie, type de résultat en sortie, résultat attendu sur un exemple...)
- quel est le **comportement attendu** pour mon algorithme ? (données en entrée, type de données en entrée, résultat en sortie, type de résultat en sortie, résultat attendu sur un exemple...)

Ensuite, résoudre le problème en utilisant ces outils :

- comment résoudre le problème **étape par étape** ? (essayer sur l'exemple testé)
- est-ce que les **outils à disposition** sont **utilisables** pour réaliser chaque étape ?

Enfin :

- comment **structurer** l'utilisation des outils à disposition ? (**combinaison** des différents outils à l'intérieur de structure de **boucles**, de **tests**, utilisation d'un **organigramme**...)
- comment **décomposer** le problème ? (et **reformuler** chaque sous-problème pour le résoudre avec les outils à disposition, écrire un algorithme par sous-problème)

La "minute xkcd" : 722

Le toit avec une pente à 26,565°

Instructions Java :

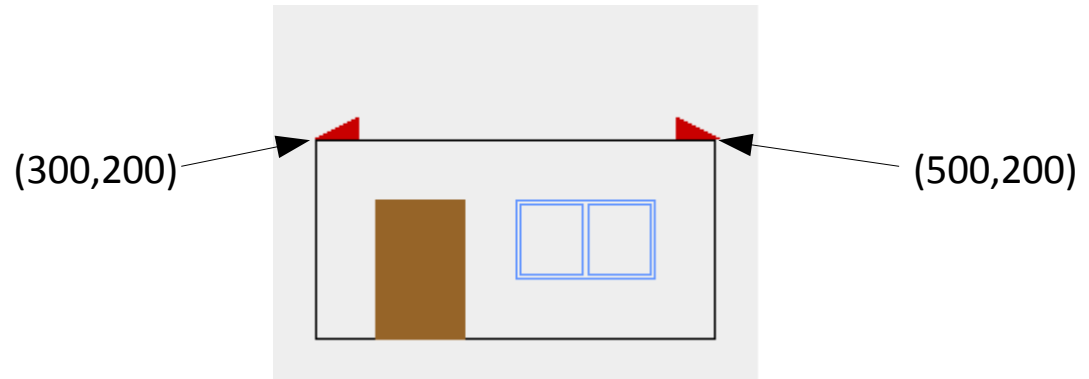
```
i=0;
rouge=couleurRGB(200,0,0);
while (i<51){
    dessineLigne(g,300+2*i,199-i,
        300+2*i,199,rouge);
    dessineLigne(g,300+2*i+1,199-i,
        300+2*i+1,199,rouge);
    dessineLigne(g,500-2*i,199-i,
        500-2*i,199,rouge);
    dessineLigne(g,500-2*i+1,199-i,
        500-2*i+1,199,rouge);
    i=i+1;
}
```

Faire la trace de l'algorithme : $i=10$

dessineLigne(g,320,189,320,199,rouge)

dessineLigne(g,321,189,321,199,rouge)

...



Traduction en pseudo-code :

Algorithme **DessineToit()**

Variables : entier i , couleur *rouge*

Début

$i \leftarrow 0$

rouge \leftarrow couleurRGB(200,0,0)

Tant que $i < 51$ faire :

dessineLigne(300+2*i*,199-*i*,300+2*i*,199,rouge)

dessineLigne(301+2*i*,199-*i*,301+2*i*,199,rouge)

dessineLigne(500-2*i*,199-*i*,500-2*i*,199,rouge)

dessineLigne(501-2*i*,199-*i*,501-2*i*,199,rouge)

$i \leftarrow i+1$

Fin TantQue

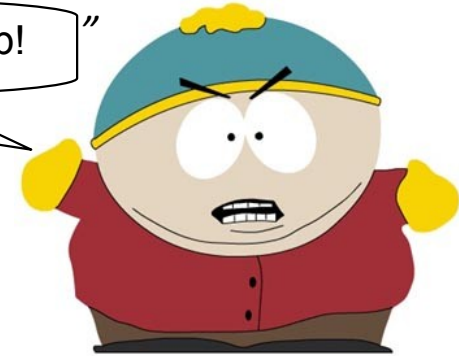
Fin

Le nombre d'utilisations d'un mot dans un texte

Écrivez un algorithme **TrouveMot** qui prend en entrée une chaîne de caractères *mot* et une chaîne de caractères *texte*, et compte le nombre de fois que *mot* apparaît dans *texte*.

Exemple : *mot* = "f***"

texte = "Nobody's f***in' screaming, Craig! Wake the f*** up!"



Algorithme **TrouveMot** :

Entrées : chaîne de caractères *mot*, chaîne de caractères *texte*

Sorties : nombre de fois (entier) que *mot* apparaît dans *texte*

Début

$i \leftarrow 1$

$compteur \leftarrow 0$

Tant que $i < \text{Longueur}(\text{texte}) - \text{Longueur}(\text{mot}) + 2$

faire :

Si $\text{mot} = \text{SousChaîne}(\text{texte}, i,$

$i + \text{Longueur}(\text{mot}) - 1)$ alors :

$compteur \leftarrow compteur + 1$

FinSi

$i \leftarrow i + 1$

Fin TantQue

renvoyer *compteur*

Fin

La "minute culturelle"

Comment compter **plusieurs mots** dans un texte ?

Méthode naïve :

appliquer l'algorithme **TrouveMot** pour chaque mot à compter
autant de lectures du texte que de mots distincts à compter

Méthode astucieuse **plus rapide** :

faire une lecture du texte en construisant un dictionnaire
compter tous les mots au fur et à mesure
ensuite, pour avoir le nombre d'apparitions d'un mot,
simplement aller voir dans le dictionnaire
(pas besoin de reparcourir le texte)

Les tableaux

Les tableaux sont des variables qui contiennent **plusieurs variables de même type**, stockées chacune dans une des cases du tableau.

en pseudo-code

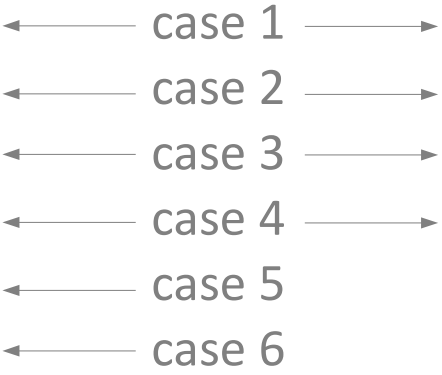
Variables : *tableau1*, un tableau d'entiers,
tableau2, un tableau de chaînes de caractères

Par exemple,

Un **tableau d'entiers** :

Un **tableau de chaînes de caractères** :

4
5
1
23
8
9



"chaine1"
"chaine2"
"blabla"
"toto"

longueur 4

longueur d'un tableau = nombre de cases

Longueur(tableau2)

longueur 6

```
tableau1=NouveauTableau(6)  
Case(tableau1,1)←4  
Case(tableau1,2)←5  
Case(tableau1,3)←1  
Case(tableau1,4)←23  
Case(tableau1,5)←8  
Case(tableau1,6)←9
```

```
tableau2=NouveauTableau(4)  
Case(tableau2,1)←"chaine1"  
Case(tableau2,2)←"chaine2"  
Case(tableau2,3)←"blabla"  
Case(tableau2,4)←"toto"
```

Les tableaux

Les tableaux sont des variables qui contiennent **plusieurs variables de même type**, stockées chacune dans une des cases du tableau.

en Java

Déclaration des variables de tableau (tableau d'entiers, et tableau de chaînes de caractères) :

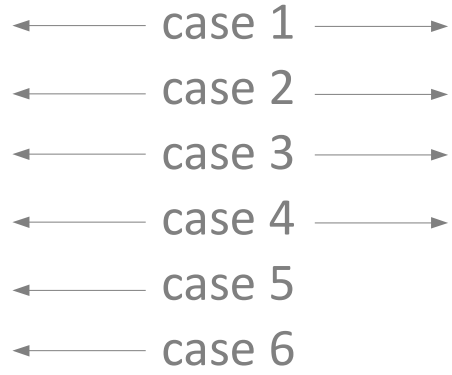
```
int[] tableau1; String[] tableau2;
```

Par exemple,

Un **tableau d'entiers** :

Un **tableau de chaînes de caractères** :

4
5
1
23
8
9



"chaine1"
"chaine2"
"blabla"
"toto"

longueur d'un tableau = nombre de cases
`tableau2.length`

longueur 4

longueur 6

```
tableau1=new int[6];  
tableau1[0]=4;  
tableau1[1]=5;  
tableau1[2]=1;  
tableau1[3]=23;  
tableau1[4]=8;  
tableau1[5]=9;
```

```
tableau2=new String[4];  
tableau2[0]="chaine1";  
tableau2[1]="chaine2";  
tableau2[2]="blabla";  
tableau2[3]="toto";
```

Attention, cases du tableau `t` numérotées de 0 à `t.length-1` en Java.

Les tableaux

Pour lire le contenu d'un tableau...

il faut une **boucle pour aller lire chaque case** !

Si le tableau a été prévu trop court au début, **impossible de changer sa longueur...** il faut une boucle pour le recopier dans un tableau plus grand !

Possibilité de créer des **tableaux de tableaux...**

L'algorithme de base, le **parcours du tableau**, pour visiter chaque case :

Variables : tableau d'entiers *tab*, entier *i*

$i \leftarrow 1$

Tant que $i < \text{Longueur}(tab)+1$ faire :

 [des choses avec la *i*-ième case du tableau **Case**(*tab*,*i*)...]

$i \leftarrow i+1$

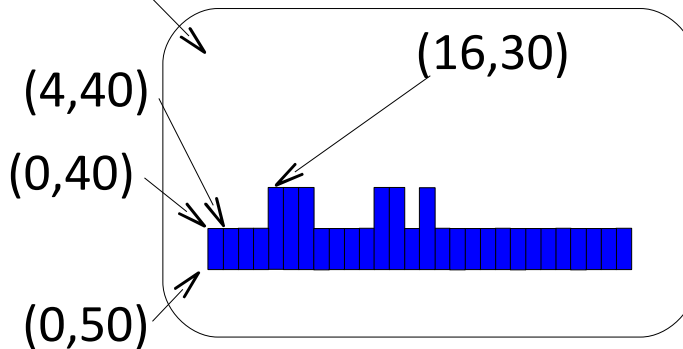
Fin Tant que

Graphique du nombre d'apparitions des mots dans un texte

Un tableau
de chaînes de
caractères *Mots*

j	1
ai	1
cueilli	1
ce	1
brin	2
de	2
bruyère	2
l	1
automne	1
est	1
morte	1
souviens	2
t	2
en	1
nous	2
ne	1
verrons	1
plus	1
sur	1
terre	1
odeur	1
du	1
temps	1
et	1
toi	1
que	1
je	1
attends	1

(0,0) Résultat voulu :



Algorithme **DessineHistogramme**

Entrée : tableau de chaînes de caractères *Mots* et
tableau d'entiers *NbApparitions*.

Variable : entier *compteur*

Début

compteur ← 1

Tant que *compteur* < **Longueur**(*Mots*)+1 faire :

dessineRectanglePlein(*compteur**4-4,
50-10***Case**(*NbApparitions*,*compteur*),
4,10***Case**(*NbApparitions*,*compteur*),
couleurRGB(0,0,255))

compteur ← 1 + *compteur*

Fin TantQue

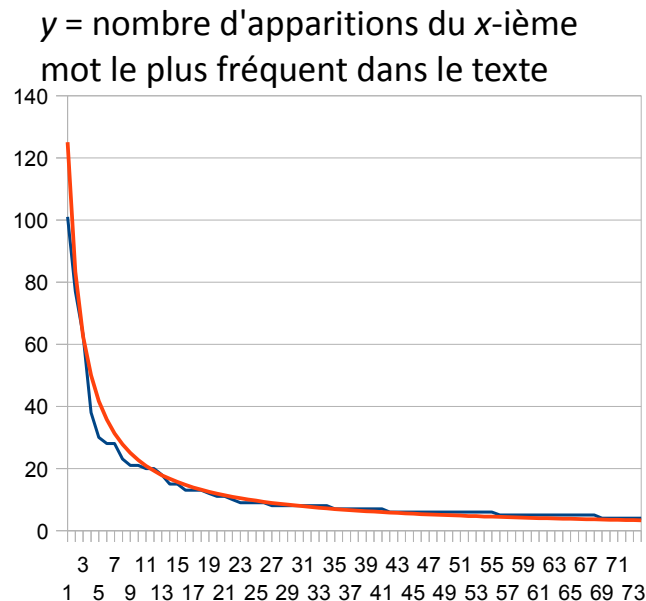
Fin

Un tableau d'entiers *NbApparitions*

Le nombre d'apparitions d'un mot dans un texte

La "minute mathématique"

La loi de Zipf prédit la courbe du nombre d'apparitions des mots les plus fréquents d'un texte.



Fonctionne pour n'importe quel texte assez long...

La boucle “for” / “Pour tout...”

La boucle “for” / “Pour tout”

Une boucle pour **parcourir tous les entiers entre deux valeurs entières.**

Algorithme **DessineHistogramme**

Entrée : tableau de chaînes de caractères *Mots* et tableau d'entiers *NbApparitions*.

Variable : entier *compteur*

Début

compteur ← 1

Tant que *compteur* < **Longueur**(*Mots*)+1 **faire :**

dessineRectanglePlein(*compteur**4-4,
 50-10***Case**(*NbApparitions*,*compteur*),
 4,10***Case**(*NbApparitions*,*compteur*),
 couleurRGB(0,0,255))

compteur ← 1 + *compteur*

Fin TantQue

Fin

En Java :

```
int compteur;  
compteur=1;  
while (compteur<mots.length+1) {  
    ...  
}
```

Algorithme **DessineHistogramme**

Entrée : tableau de chaînes de caractères *Mots* et tableau d'entiers *NbApparitions*.

Variable : entier *compteur*

Début

Pour *compteur* de 1 à **Longueur**(*Mots*) **faire :**

dessineRectanglePlein(*compteur**4-4,
 50-10***Case**(*NbApparitions*,*compteur*),
 4,10***Case**(*NbApparitions*,*compteur*),
 couleurRGB(0,0,255))

compteur ← 1 + *compteur*

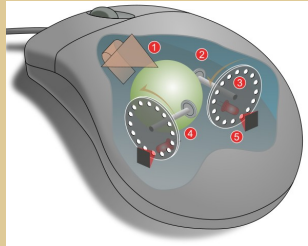
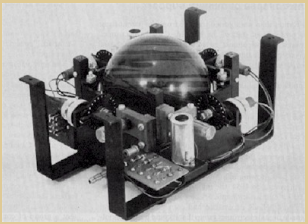
Fin Pour

Fin

*déclaration +
initialisation* *condition d'arrêt* *mise à jour*

```
for (int compteur=1; compteur<mots.length+1; compteur++) {  
    ...  
}
```

Souris et autres périphériques d'entrée



La "minute culturelle"

L'invention de la souris

1952 Trackball (boule de commande)
Tom Cranston et Fred Longstaff
(Marine Royale Canadienne)

1963 Souris mécanique
Douglas Engelbart et Bill English
(Stanford Research Institute)

1977 Souris optique
Jean-Daniel Nicoud et André Guignard
(Ecole polytechnique fédérale de Lausanne)

Entrées-sorties dans la communication ordinateur – utilisateur
Quel **type de données** utiliser en **algorithmique** pour coder les entrées-sorties ?

Périphérique	Type de données transmises
Clavier	chaîne de caractères
Souris à 1 bouton	deux entiers (abscisse et ordonnée) + un booléen (clic ou pas)
Webcam	image, donc tableau de tableaux de couleurs RGB
Kinect	image + tableau de tableaux d'entiers (profondeur)
Ecran	Si ligne de commande : chaîne de caractères Si interface graphique : image, donc tableau de tableaux de couleurs RGB

La "minute xkcd" : 243

Crédits :
http://en.wikipedia.org/wiki/File:DATAR_trackball.jpg
http://cerncourier.com/cws/article/cern/28358/1/cernbooks2_12-00
 Jeremymkemp, Pbroks13,
http://fr.wikipedia.org/wiki/Fichier:Mouse_mechanism_diagram.svg
 Stéphane Magnenat (User:Nct)
<http://en.wikipedia.org/wiki/File:SmakyMouseAG.jpeg>

Instructions d'entrée-sortie en Java et pseudo-code

Entrées clavier

en pseudo-code

chaîne de caractères



reponseALaQuestion(*questionAAfficher*)

affiche la question *questionAAfficher* et renvoie une chaîne de caractères.

Exemple : **reponseALaQuestion**("Quel est votre nom") me laisse taper mon nom au clavier et renvoie "Gambette"

en Java

```
Scanner lectureClavier = new Scanner(System.in);  
String stringLu = lectureClavier.next();
```

LectureClavier.next() renvoie une chaîne de caractères.

```
int intLu = lectureClavier.nextInt();
```

LectureClavier.nextInt() renvoie un entier.

Sorties écran (ligne de commande)

en pseudo-code

chaîne de caractères



Affiche(*chaineAAfficher*)

affiche la chaîne de caractères *chaineAAfficher* et ne renvoie rien.

en Java

```
String chaineAAfficher="blabla";  
System.out.println(chaineAAfficher);  
affiche la chaîne de caractères questionAAfficher,  
puis retourne à la ligne, mais ne renvoie rien.
```

Les fonctions

La "minute mathématique"

fonction	exemple	entrées possibles	sortie
cosinus	$\text{cosinus}(1.047)=0.5$	flottant	flottant
somme	$\text{somme}(2,3)=5$	2 entiers 2 flottants	entier flottant
opposé	$\text{opposé}(4)=-4$	entier flottant	entier flottant
inverse	$\text{inverse}(10)=0.1$	flottant	flottant
différence	$\text{différence}(2,3)=-1$	2 entiers 2 flottants	entier flottant
estPositif	$\text{estPositif}(-5)=\text{FAUX}$	entier	booléen
partieEntière	$\text{partieEntière}(5.6)=5$	flottant	entier
moyenne	$\text{moyenne}(2,4,6)=4$	3 flottants	flottant
min	$\text{min}(\{6,2,4,3\})=2$	tableau d'entiers tableau de flottants	entier flottant

Attention : point au lieu de virgule dans les flottants (pour différencier de la virgule qui sépare les entrées d'une fonction)

Les fonctions

Les **fonctions** = les **algorithmes**

Zéro, un ou plusieurs paramètres en entrée...

Une sortie (ou aucune : `void` en Java)...

En Java :

Toujours une fonction `main()` **qui ne renvoie rien** et qui **prend en entrée les paramètres du programme**

Déclaration des fonctions après la fonction `main()`

Visibilité des variables : toute variable déclarée à l'intérieur d'une fonction n'est valable **que dans cette fonction** et **ne peut pas être utilisée ailleurs**.

Variables « **locales** »

```
public class TP1{
    public static void main(String[] arg) {
        int i,j;
        i=5;
        j=34;
        System.out.print("i+1="+inc(i)+" , i="+i+" ,
            j="+j+" , somme : "+addition(i,j));
    }
    public static int inc(int i){
        i=i+1;
        return i;
    }
    public static int addition(int i, int j){
        return i+j;
    }
}
```

pas la même variable *i* même si elles ont la même valeur !

appel de la fonction addition

type de variable renvoyée par la fonction

ligne de déclaration de la fonction addition

définition de la fonction addition

entrées de la fonction précédées de leur type