Qu'est ce qu'un algorithme ?

1 Définition

Un algorithme est une suite finie d'opérations ou règles à appliquer dans un ordre déterminé, à un nombre fini de données, pour arriver à en un nombre fini d'étapes, à un certain résultat et cela indépendamment des données.

Mots clés :

-> **démarche**
-> **prépare une programmation fiable** (sans garantir que ce soit la meilleure solution toutefois)

Inconvénients :

-> démarche pas forcément unique
-> on se limite souvent aux "bonnes pratiques"
-> peut-être assimilé à une recette

Pour écrire un algorithme il faut connaître les **outils** à utiliser

On parle de **langage algorithmique** :

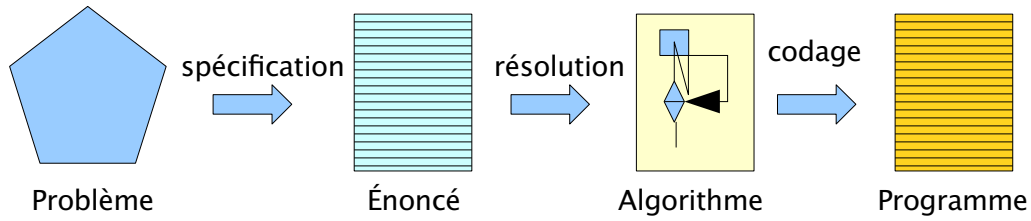
Ordinogramme : représentation **graphique**
- Une action est décrite dans un rectangle
- une condition est décrite dans un losange
- l'ordre des actions est décrit à l'aide de flèches

Note : peu intéressant dès que le programme est complexe

Pseudo-code : représentation **textuelle**
- écrit en langage naturel
- ne comporte que des éléments syntaxiques génériques

Permet de résoudre des problèmes compliqués <-- on étudiera ce langage

Graphiquement :



De façon très simplifiée, 4 étapes seulement.

En réalité: un **planning** est associé en parallèle, un **cahier des charges**, des **revues de projet** ...

2) Exemple de réalisation d'une application informatique

Écriture d'un cahier des charges avec le client

-> version initiale (discussion, temps)
-> version finale

Note: un cahier des charges est un contrat, et chaque mot doit être justifié.

-> cerner les besoins, les moyens (financement)
-> penser à la formation des utilisateurs
-> penser à une éventuelle migration
-> signature : elle engage les deux parties

Écriture du programme informatique

- Analyse fonctionnelle
- Décomposition en programmes à réaliser

Pour chaque programme:

-> étudier un algorithme et le finaliser
-> écrire le programme
-> le tester (démarche qualité)

TANT QUE tests non tous satisfaisants
    corriger les erreurs
FIN TANT QUE

Tester l'application complète et la mettre au point
Faire vérifier par le client
SI nécessaire et raisonnable
    faire modifications /corrections
Fin SI

Livraison et mise en place
Formation utilisateur
Facturation

Algorithme / Programme C

```
count, val_max entier

saisir (val_maxi)

pour compteur = 1 à val_max
        afficher (count)
        incrémenter count
fin pour count
```

```
#include <stdlib.h>
#include <stdio.h>

int saisir_val_max( int value );
int main (void)
{
   long val_max;
   long count;
   for ( count = 1 ; count < val_max  ; count ++ )
   printf("Compteur : %d\n", count );
   return EXIT_SUCCESS;
}

int saisir_val_max( value)
{
  fscanf("Entrer valeur maxi : %d\n",&value);
  return value;
}
```

Note: l'**algorithme** met en avance le **concept**, l'essentiel de la méthode, qui pourra être **implémentée** en utilisant un langage de programmation.

L'algorithme est plus facile à lire, et plus concis

3) Méthodologie de construction d'un algorithme

Problème : son **énoncé** doit être sans ambiguïté

Exemple : déterminer le maximum entre deux entiers donnés

Instance d'un problème : soit un doublet d'entiers

Programme :

Caractérisé par :
- un **ensemble** de données, et un ensemble de **résultats**
- une solution informatique : description d'**actions** à décrire dans un certain **ordre** et dans un certain **langage de programmation**

Bien analyser et comprendre le problème
Identifier les données fournies : entrées
Préciser les résultats attendus : sorties
Élaborer le processus et transformation des entrées en vue d'obtenir les sorties attendues

Objectifs :

Programmation structurée

convivialité : ne pas négliger l'interface utilisateur (on ne parle pas de l'apparence ici, i.e. look)
modularité: on n'écrit plus de gros programmes
lisibilité: commentaires, choix judicieux des identificateurs, implémentation judicieuse ..etc
maintenabilité: un programme bien conçu, est facile à maintenir
réutilisabilité: une conception de qualité permet de réutiliser certaines parties
extensibilité: prvoir dès la conception d'ajouter des nouvelles fonctionnalités (futures versions)

-> PORTABILITÉ : un programme écrit correctement en langage C est réutilisable avec la plus grande partie des systèmes d'exploitations les plus connus (Linux, Windows, Mac OS X, Solaris, ...)

Structure d'un algorithme

Tout algorithme ne peut utiliser que 3 types de structures différentes:

- la séquence
- l'alternative
- l'itération

La séquence :

C'est une suite d'instructions, exécutées séquentiellement, i.e. linéairement à la suite les unes des autres.

Remarques:

- Une **séquence** est traduite en **langage C** par une suite d'instructions simples enchaînées dans **un bloc, qui peut ne compter qu'une ligne. On parle alors d'instruction simple.**

- **instruction simple: redirection entre un périphérique d'entrée et un périphérique de sortie**

Exemples de redirections possibles:

- **affectation** : le périphérique d'entrée est le clavier, celui de sortie une variable (ou une constante) écrit dans la mémoire du programme en cours d'exécution

- **affichage**: le périphérique d'entrée est l'espace mémoire du programme, celui de sortie, l'écran
- acquisition: périphérique d'entrée est le clavier, en sortie, l'espace mémoire du programme
- autres cas: entrée : fichier, sortie écran
- .... etc

- lors du débogage, l'instruction « **s** » exécute un bloc complet d'instructions, et passe au bloc ou à la structure suivante, alors que l'instruction « **n** » exécute **une seule instruction**, en respectant l'ordre défini dans le bloc en cours d'exécution.

L'alternative

Toute alternative est basée sur une expression booléenne.

Une expression booléenne est une équation logique qui ne peut prendre que deux valeurs: VRAI ou FAUX.

La valeur FAUX est representée par un 0 (zéro) en langage C
La valeur vraie, par un entier non nul

Cette expression booléenne pourra être construite à l'aide de :

- variables
- valeurs
- expressions mathématiques
- comparaisons (vrai si égal, vrai si inférieur ou égal .. etc)

Cas possibles: == , > , < , != , >= , <=

- opérateurs booléens: ET, OU, NOT

- code de retour d'une fonction **(attention à complémenter la valeur de retour !)**

À savoir : ET est prioritaire sur OU , et NOT est prioritaire par rapport à ET

Exemples:

```
if ( 0 == variable )  /*  evite l'erreur classique:  if ( variable = 0 )  qui est une affectation */
```

```
if 1 /* toujours vrai */

if ( 0 == sin(x) )

if ( a == 1 )
  {
    snprintf(stdout, sizeof(buff), "chaine %s " , string);
    for (i = 0 ; i < 10 ; i+=1 )
      {
          compteur += 1;
      }

  }

if ( 0 != erreur )
  EXIT_FAILURE;   /*  pas de { } si une seule ligne, améliore la lisibilité */


if defined (MACOSX)  &&  !defined (LINUX ) || defined ( QUARTZ)
  {
     .....
  }

if   !( strcmp ( string1, string2 ))  /* si strcmp() est vraie, elle retourne 0  */
    {
     /*  cas ou string1 est égale à string2 */
     .....
    }
else  /*  strings différentes  */
  {

  }
```

si NOT ((a ET b) OU c) pourrait s'écrire sans parenthèses : si NOT a ET b OU C

-> **mais il vaut mieux en mettre** , ce qui constitue une **bonne habitude**

Rappel: Théorèmes de De Morgan :

Première forme : $\overline{A+B}=\overline{A}\cdot\overline{B}$

Seconde forme : $\overline{A\cdot B}=\overline{A}+\overline{B}$

Exercice: