Algorithmie-Principes

Septembre 2023

© pdareys@free.fr

ESGI école supérieure de génie informatique

Sigma Web

Programme

- Introduction
- Variables/Opérateurs
- Structures conditionnelles et itératives
- Fonctions/Récursivité
- Structures de données spécifiques
- Persistance (fichiers, Bd)
- Programmation Objet

Introduction

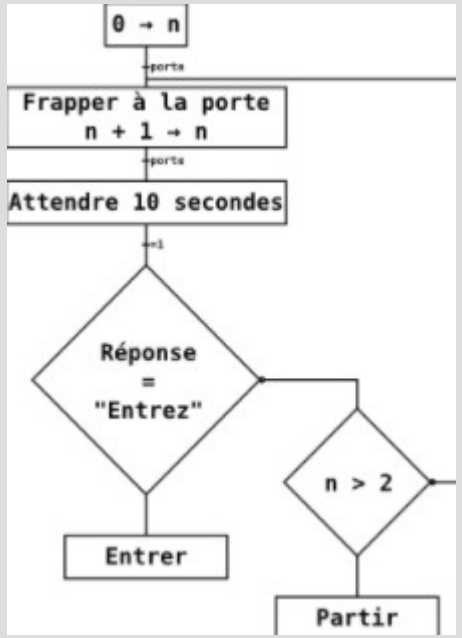
- Définition
  - « Suite d'opérations appliquées à des données pour résoudre avec **certitude** un problème en un **nombre fini d'étapes** »

    Exemple :
    - Traitement mathématique
    - Recette de cuisine

  - Un algorithme a une Complexité : notée *O(f(n))*
    - où f(n) est la fonction mathématique associée à la quantité d'information manipulée par l'algo
    - Ex : double boucle de 1 à n → $f(n) = n^2$

Introduction

- Représentation
  - 1/ Graphique :
    - Utilisé dans les années 70
    - Rapide et facile à lire
    - Très pratique pour des petits algorithmes
      - Mais…
    - Éloigné des langages



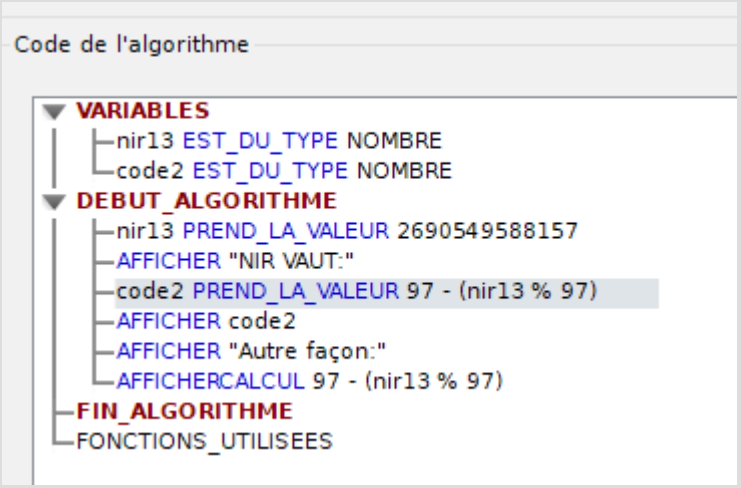
Introduction

- Représentation
  - 2/ Pseudo-code :
    - Indépendant de tout langage
    - Proche des langages
      - Mais…
    - Non standardisé
    - Rien ne dit que « ça marche »

```
VAR a,b,c
a=3
b=5
c=a+b
OUTPUT c
```

Introduction

- Représentation
  - 3/ Utilisation d'outils :
    Ex : *AlgoBox*
    - Indépendant des langages



Introduction

- Choix pour la formation
  - Utilisation du Javascript
  - Avantages
    - Le langage le plus utilisé ! (dans de nombreux frameworks modernes)
    - Interprété
    - Déploiement très facile
  - Liste d'outils :
    - Un navigateur (avec debugger intégré) comme **Google Chrome**, Firefox, ...
    - Un outil de dev (**Vscode**, Notepad ++)

Variables/Opérateurs

- Variables
  - Permet le stockage des données initiales et résultats
    - Type « simple » (entier, flottant, booléen, chaîne, …)
    - Type « complexe » (structure appropriée, objet)
    - Type tableau
  - En JS (autotypé) :
    - Pas de déclaration de type (mot-clé : **var**, ou let)
      Ex : let age=30 ;// age supposé entier par la suite
    - Objet : {nom : ‘Durand’, prenom : ‘Paul’}
    - Tableau : []

Variables/Opérateurs

- Variables
  - Choix du type
    - Dans la table des symboles il faut savoir combien d'octets sont réservés pour chaque variable
    - Fixer le type évite des confusions
  - Table des symboles

```
var n=2;
```

Table des symboles	
ID	@
n	0xA0

Mémoire	
@	Valeur
0xA0	2

⚠ En réalité, la table contient également d'autres informations (scope...).

Variables/Opérateurs

- Opérateurs
  - Affectation
    - =

      Ex : *a = 3*
    - Bien comprendre l'affectation
      - Une expression comme celle-ci : *c=a+b* est partagée en 2 parties
        - L'expression à droite (*right-value*) est évaluée
        - Le résultat est affecté à la variable à gauche (*left-value*)

Variables/Opérateurs

- Opérateurs
  - Opérateurs arithmétiques

+	Addition entre nombres, ex : 2+3 (=5)
-	Soustraction, ex : 3-2 (=1)
*	Multiplication, ex : 3*2 (=6)
/	Division, ex : 3/2 (=1.5)
%	Modulo, ex : 3%2 (=1)

  - Opérateurs de concaténation :

    Ex : ‘bon’ + ‘jour’ = ‘bonjour’

Structures conditionnelles

- « IF/ELSE»
  - Syntaxe

```
if (exp1) {
   // Code exécuté si exp1 vraie.
}
else if (exp2) {
   // Code exécuté si exp1 fausse mais exp2 vraie.
}
else{
   //  Code exécuté si exp1 et exp2 fausses.
}
```

- « if » ne peut apparaître qu'une seule fois
- « else » 0 ou 1 fois
- « else if » 0 ou n fois
- Les expressions sont évaluées à *true* ou *false* (vrai/faux)
- Test multiples : il existe la clause *switch (case)*

Structures conditionnelles

- Opérateurs de comparaison (en JS)

==	Egalité simple (même valeur)
!=	Non-égalité simple
===	Egalité totale (même valeur, même type)
!==	Non-égalité totale
<	Strictement inférieur
>	Strictement supérieur
<=	Inférieur ou égal
>=	Supérieur ou égal

- Opérateurs logiques

!	Non	Négation
&&	Et	Intersection
||	Ou inclusif	Réunion

- D'où le tableau

A	B	A AND B	A OR B
VRAI	VRAI	V	V
VRAI	FAUX	F	V
FAUX	FAUX	F	F

Structures de contrôle

- *while* ou *do … while*
  - Syntaxe :

    while (exp) {
    // exécutée tant que exp est vraie
    }
  - Utile quand on ne sait pas à priori combien de *tours* seront nécessaires
  - *do … while* intéressant si on est pas sûr que d'emblée la condition est vraie

  Ex :

```
var n=0;
while(n<10){
  console.log(n); // 0 2 4 6 8
  n=n+2;
}
```

Structures de contrôle

- *for*
  - Syntaxe :

```
for (expInit ; expCond; expAction) {
    // exécutée tant que expCond est vraie
    }
```

  - Le plus utilisé
  - Le plus performant
  - Ruptures *: break* pour sortir de la boucle, *continue* pour passer au tour suivant (pas très utilisé)

  Ex :

```
for(var n=0;n<10;n=n+2){
  console.log(n); // 0 2 4 6 8
}
```

Fonctions

- USAGE
  - Sous-programme pouvant être appelé par le programme principal ou une autre fonction
  - Evite les redondances et simplifie la maintenance
    - Isole une séquence d'instruction
    - Remplace chaque occurrence de cette séquence par un appel
    - On a donc un seul endroit à « mettre à jour »
  - Permet la modularité :
    - Du développement collaboratif
    - Des tests
    - Du deboggage
    - Par la création de bibliothèques (api)

Fonctions

- Paramètres
  - Peut recevoir des paramètres (entrée)
  - En général doit retourner **un** résultat (sortie)
    Ex :

```
function add(x,y){
  var z=x+y;
  return z; // Retourne une copie de c.
}

function main(){
  var a=3,b=5;
  var c=add(a,b);
  console.log(c); // 8
}
```

  - Fonctionnement :
    - a, b initialisées
    - Exécution transférée à *add()*
      - a copiée dans x, b dans y
      - x+y transféré à z
    - L'exécution revient dans le main
    - La valeur de z est copiée dans c qui est affichée

Fonctions

- Portée (scope)
  - Dans l'entête de la fonction, on parle de paramètres *formels* (à l'appel ce sont des paramètres *effectifs*)
  - Les variables introduites dans une fonction ainsi que les paramètres sont **locales** à la fonction, donc **détruites** à la sortie
  - Il est possible de définir des variables **globales** (donc accessibles partout)

    Ex (en JS) :

```
var a=1,b=2,c=3;

function main(){
  var b=9;
  console.log(a); // 1
  console.log(b); // 9
  console.log(c); // 3
}
```

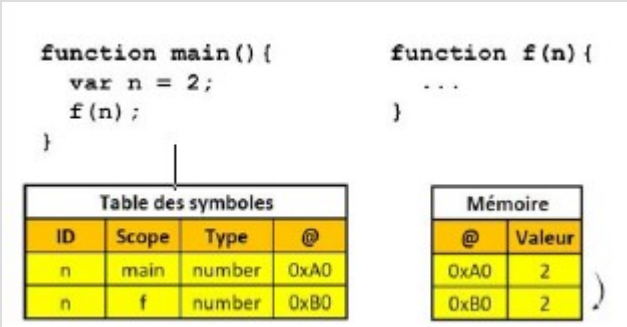
⚠ Dans cet exemple, **la variable locale b masque la variable globale b**, elle ne l'écrase pas.

Fonctions

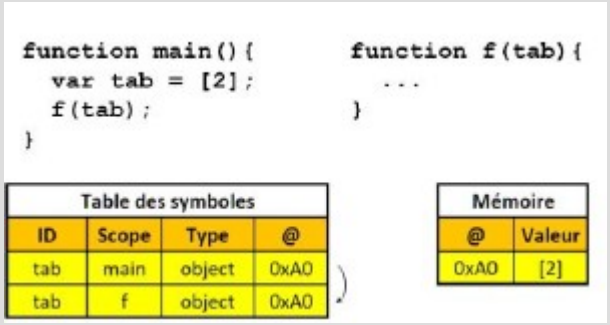
- Passage par valeur ou référence
  - Par valeur : la valeur est copiée dans le paramètre formel (en général concerne les types primitifs comme entier, flottant, ...)
  - Par référence : l'adresse du paramètre effectif est copiée dans dans le paramètre formel (les objets, les tableaux)

Fonctions

- Passage par valeur ou référence
  - Par valeur :



  - Par référence :



Fonctions

- Fonction récursive
  - Elle fait appel à elle-même
  - Fait appel à une valeur du paramètre en général inférieure à la valeur du paramètre précédent
  - Il ne faut pas oublier un test de sortie (attention au débordement

  Ex : la factorielle (n ! = n*n-1*...*1)

```
function fact(n) {
    if (n == 1)
        return 1 ; // sortie impérative, risque de débordement !
  return n*fact(n-1) ;
}
```

Fonctions

- Fonction récursive
  - Certains problèmes sont très difficiles à résoudre en dehors d'une fonction récursive
    Ex : tours de Hanoï

Algo :

```
# n : nombre de disques.
# d : numéro de la tour de départ
# i : numéro de la tour intermédiaire
# a : numéro de la tour d'arrivée
FUNCTION hanoi(n,d,i,a):
  IF n!=0:
    hanoi(n-1,d,a,i)
    Déplacer le disque de la tour #d à la tour #a
    OUTPUT le contenu des tours
    hanoi(n-1,i,d,a)
```

Fonctions

- Closures
  - Fonction définie dans une autre fonction
  - Les fonctions de tri sont souvent de ce type
    Ex :

```
function add(n){
  function addPlus(k){
    return n+k;
  }
  return addPlus;
}

function main(){
  var add3=add(3);
  console.log(add3(5)) // 8
}
```

La closure addPlus() n'est pas accessible directement.

Fonctions

- Closures
  - Lambdas : closure anonyme
    ex :

```
function add(n){
  return function(k){
    return n+k;
  }
}

function main(){
  var add3=add(3);
  console.log(add3(5)) // 8
}
```

⚠ Les lambdas sont constamment utilisées comme fonctions *callback* en programmation événementielle, spécialement en JavaScript.

  - Fonctions anonymes (js) :
    data.forEach (element => {console.log (element)}) ;

Structure de données

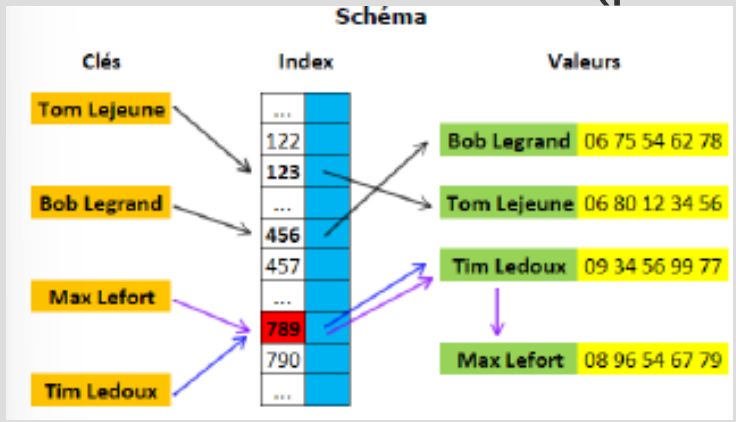
- Tableaux
  - Séquence ordonnée dans un unique bloc mémoire
    - Représenté par une variable indicée
    - Selon les langages :
      - Données homogènes ou hétérogènes (js)
      - Une ou plusieurs dimensions
      - L'indice en général commence à 0 (js)
      - Taille fixe ou dynamique (js)
      - En général déclaré avec des []

      Ex :

```
var tab,i;
tab=[1,2,3];
tab[1]=5;
for(i=0;i<tab.length;i++){
  console.log(tab[i]); // 1 5 3
}
```

Structure de données

- Tables de hachage
  - Séquence non ordonnée de paires clé/valeur (ou tableau associatif ou annuaire)
  - Une fonction de hachage transforme les clés pour produire un entier utilisé comme index puis une autre fonction résout les éventuelles collisions (pas de bijection donc)



Structure de données

- Type spécifique
  - Les impératifs algorithmiques amènent à définir d'autres types de données liés au projet

- Ex :

```
const PERSON = {     // une personne avec des attributs (syntaxe KamelCase !)
    firstName: "",
    lastName: "",
    age: 0,
    isLogged: false};
```

- Créer une personne :

```
let p = Object.create (PERSON) ;
p.firstName = 'Paul' ;  // affecter le prénom
p.age = ...
```

Persistance

- Fichiers
  - Il s'agit du mécanisme responsable de la sauvegarde et restitution des données
  - Quand il s'agit de fichiers, il y a généralement un lien entre un nom logique et un nom physique, puis le code se base sur le nom logique
  - Exemple de code (nécessite *node.js*)

```
const fs = require("fs");

fs.open("sample.txt", "w", (err, file) => {
   if (err) throw err;
   console.log(file);
});
```

Persistance

- Fichiers
  - Les actions courantes :
    - Lecture (indexée ou séquentielle)
    - Écriture
    - Changement de droits
    - Renommage
    - Copie
    - ...

Persistance

- Bases de données
  - Il existe des bases de données relationnelles (*SQL*) ou des bases de données *NoSQL* (MongoDb, se base sur du JSON)
  - Alternative aux fichiers
  - mais nécessite un serveur supplémentaire et donc des tâches d'administration supplémentaires

Persistance

- Bases de données
  - Exemple de code (nécessite *node.js*)

```
const mysql = require ('mysql');

// Vous devez d'abord créer une connexion à la base de données
// Assurez-vous de remplacer «utilisateur» et «mot de passe» par les valeurs correctes
const con = mysql.createConnection ({
  hôte: «localhost»,
  utilisateur: 'utilisateur',
  mot de passe: 'mot de passe',
});

con.connect ((err) => {
  si (err) {
    console.log ('Erreur de connexion à Db');
    revenir;
  }
  console.log ('Connexion établie');
});
```

Programmation Objet

- Intérêt
  - Qualité du code (maintenance)
- Les 4 bénéfices
  - Encapsulation (sécurité)
  - Abstraction
  - Héritage
  - Polymorphisme
- A manipuler :
  - la classe = attributs + comportements

Programmation Objet

Exemple

```
class Personnage {          // commence par une Majuscule
  constructor(nom, sante, force) {
    this.nom = nom;         // initialisation des Attributs (ici : nom, santé, force)
    this.sante = sante;     // this est l'objet courant
    this.force = force;
  }
  // Renvoie la description du personnage
  decrire() {           // exemple de Méthode (on ne met pas function)
    return this.nom + ' a: ' +  this.force + ' points de force';
  }
}
```

Invocation :

```
const aurora = new Personnage("Aurora", 150, 25);
                    // "Aurora a : 150 points de vie, 25 en force et 0 points d'expérience"
console.log(aurora.decrire());
```

Programmation Objet

- Héritage
  - Ex :

```
class Car extends Vehicle {
  constructor (name, color) {
    super(name) ;  // super invoque le parent
    this.color = color ; // rajout attribut color : la voiture est un véhicule et en plus a une couleur !
  }
  describe () {
    return super.describe() + this.color ;
  }
}
```

Programmation Objet

- Exceptions
  - La gestion des erreurs dans un programme DOIT se faire à l'aide d'exceptions, dans le but de décharger la partie « métier » (les calculs, les data, ..) d'éventuels cas d'erreur
  - Ex : (bloc *try/catch/throw*)

```
try {
 let a=10 ; let b=0 ;
 If (b == 0)
     throw ('Interdit de diviser par 0 !!') ;        // l'instruction throw redirige vers le bloc catch
                                                        'le plus proche'
}
catch(err) {
  console.log (err);  // affiche le message !!
}
```

Conclusion

- Les problèmes algorithmiques nécessitent une phase de conception approfondie
- Au codage se rajoutent des tests (unitaires, d'intégration, de validation)
  - Il existe des outils de génération de test dans la plupart des langages

- ----------
- Q/R ???

FIN