Qualité de développement

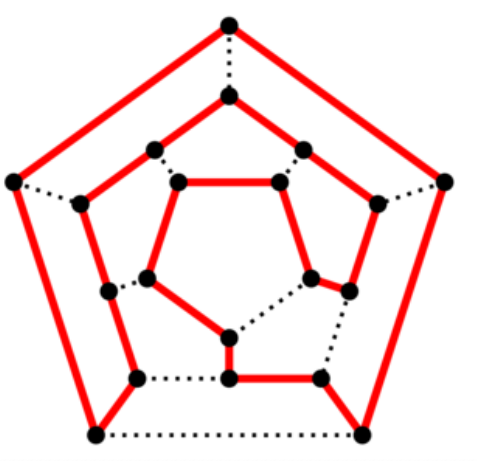
CM4-1 : Algorithmique « avancée »

Mickaël Martin Nevot

V2.0.1

Qualité de développement

Notions de base



- **Terminaison** :
  - Assurance de finir en temps fini
- **Correction** :
  - Garantie d'une solution correcte
  - Solution au problème posé
- **Complétude** :
  - Proposition de solutions sur un espace de problème donné
- **Déterminisme** :
  - Se comporte de façon prévisible (un ensemble de données particulier produira toujours le même résultat)

Programmation modulaire

- Regroupement de codes sources en **unités de travail logiques**
- **Encapsulation**
- **Réutilisabilité** et partage du code facilités
- Facilitation de réalisation de bibliothèques
- **Généricité** possible

Couplage et cohésion

- **Couplage** :
  - Interconnexion de deux classes (modules)
  - **Couplage fort** : une modification d'une classe nécessite la modification de l'autre
  - **Couplage faible** : communication de données uniquement grâce à des « interfaces »
- **Cohésion** :
  - Relations des données au sein d'une même classe
  - Centré sur un but : se concentre sur tâche unique et précise

Programmation par contrat

- Paradigme
- But principal : **réduire le nombre de bogues**
- Précise les **responsabilités** entre le client et le fournisseur
- **Pas de vérification** de validité des règles
- **Assertions** :
  - Énoncé considéré ou présenté comme vrai
  - Écrites dans le code source en commentaires
  - Donnent la sémantique du programme

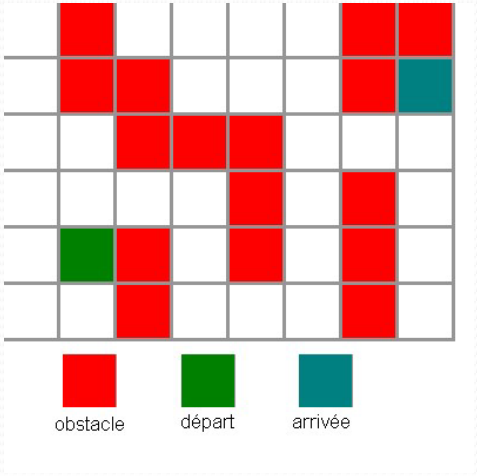
Type d'assertions

- **Précondition** : Doit garantir un traitement possible et sans erreur
  - Doit être vérifiée **avant** un traitement par le **client**
- **Postcondition** :
  - Doit être garantie **après** un traitement par le **fournisseur**
- Invariant :
  - Doit être **toujours vrai**
    (durant toute / une partie d'une application)

```
/**
 * Fonction calculant la racine carrée de la valeur de x.
 *
 * Précondition : x ≥ 0.
 * Postcondition : résultat ≥ 0 et si x ≠ 1 alors résultat ≠ x.
 */
public int sqrt() { … }
```

Efficacité d'un algorithme

- Particularités de l'ordinateur
- Performance en moyenne (**évaluation asymptotique**)
- **Complexité** :
  - L'évolution du nombre d'instructions de base en fonction de la quantité de données à traiter
  - Quantité de mémoire nécessaire pour effectuer les calculs



Complexité

- Comptabiliser le **nombre d'étapes d'un algorithme**
- Calculer dans **le pire des cas**
- Notation : $O(n)$ avec n étant le nombre d'étapes

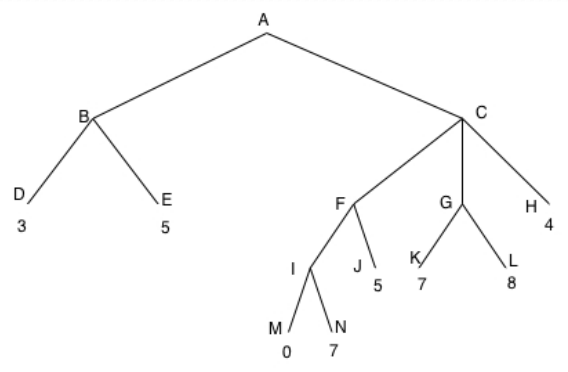
```
// Complexité O(n)
for (int i = 0 ; i < n ; ++i) { … }

// Complexité O(n²) : avec m ≤ n
for (int i = 0 ; i < m ; ++i) {
    for (int j = 0 ; j < n ; ++j) { … }
}

// Complexité O(log(n))
… // Fonction Dichotomie
```

Complexité : comparatif

Notation	Complexité	Temps n = 10	Temps n = 10000	Temps n = 1000000	Exemple
$O(1)$	Constante	10 ns	10 ns	10 ns	Index tableau
$O(\log n)$	Logarithmique	10 ns	40 ns	60 ns	Dichotomie
$O(n)$	Linéaire	100 ns	100 µs	10 ms	Parcours liste
$O(n \log n)$	Quasi-linéaire	100 ns	400 µs	60 ms	Tri fusion
$O(n^2)$	Quadratique	1 µs	1 s	2.8 h	Tri par insertion
$O(n^3)$	Cubique	10 µs	2.7 h	316 ans	
$O(n^{\log n})$	Quasi-polynomiale	100 ns	3.2 ans	10^{20} ans	
$O(e^n)$	Exponentielle	10 µs	…	…	DPFP
$O(n!)$	Factorielle	36 ms	…	…	Voyageur de commerce

Heuristiques

- Cas d'utilisation :
  - **Complexité trop grande**
  - Impossibilité d'obtenir un résultat en temps raisonnable
- Rechercher la solution la plus proche possible d'une solution optimale par essais successifs
- Pas d'exhaustivité possible : il faut faire des **choix** !
- **Choix** généralement très dépendant du problème



Cela s'appelle une heuristique

Algorithmes de tri

- **Classification** (permet de choisir un algorithme)
  - Complexité algorithmique :
    - Tri optimaux : $O(n \log n)$
  - Caractère en place :
    - Modifie directement la structure (le tableau)
  - Caractère stable :
    - Préserve l'ordre « relatif »

Une liste à trier :	(4, 1) (3, 7) (3, 1) (5, 6)
Triée par la première valeur :	(3, 7) (3, 1) (4, 1) (5, 6)
Et pas :	(3, 1) (3, 7) (4, 1) (5, 6)

Algorithmes de tri

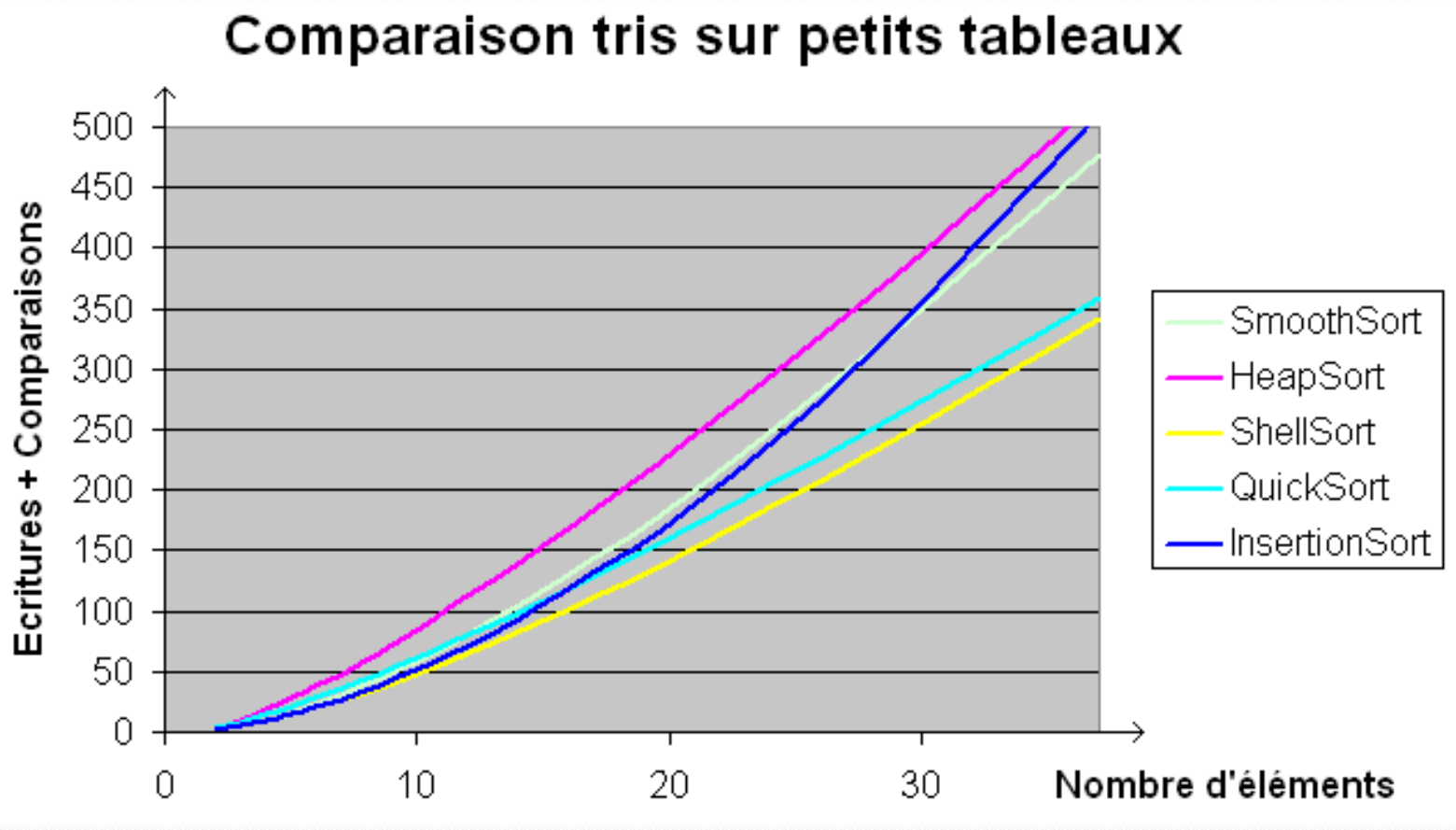
- Algorithmes simples :
  - Tri à bulles $O(n^2)$ (amusant mais pas efficace)
  - Tri par sélection $O(n^2)$ (< 7 éléments)
  - **Tri par insertion** $O(n^2)$ (< 15 éléments)
- Algorithmes élaborés :
  - Tri de Shell $O(n \log^2 n)$ (tri par insertion++)
  - Tri fusion $O(n \log n)$
  - **Tri rapide** $O(n \log n)$ ($O(n^2)$ dans le pire des cas)
  - Tri par tas $O(n \log n)$ (si $O(n^2)$ pour tri rapide)
  - **Smoothsort** $O(n \log n)$ (tri par tas++, « presque » trié)

Algorithmes de tri

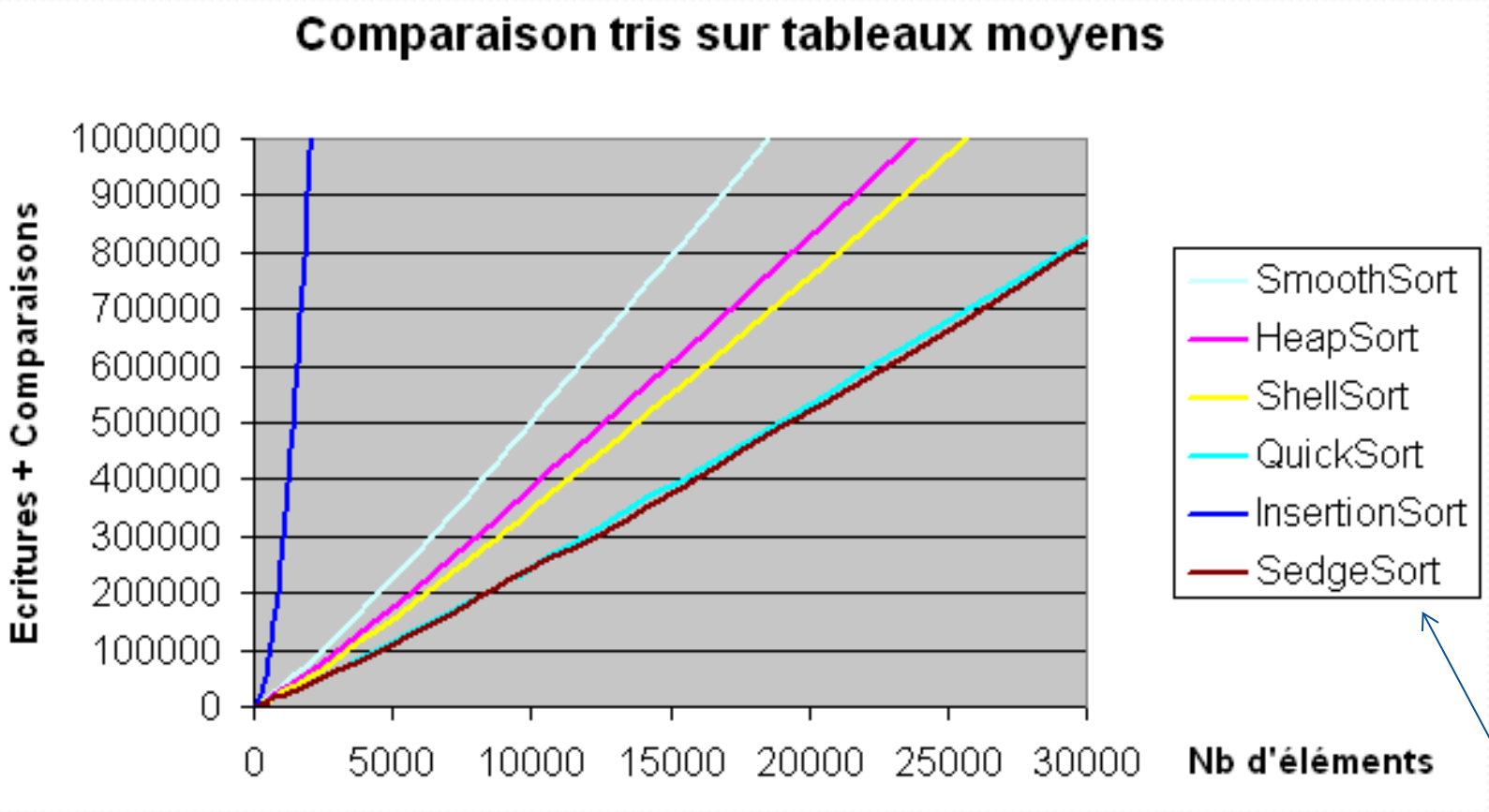
- Algorithmes simples :
  - Tri à bulles $O(n^2)$
  - Tri par sélection $O(n^2)$
  - **Tri par insertion** (Insertionsort) $O(n^2)$
- Algorithmes élaborés :
  - Tri de Shell (Shellsort) $O(n \log^2 n)$
  - Tri fusion $O(n \log n)$
  - **Tri rapide** (Quicksort) $O(n \log n)$
  - Tri par tas (Heapsort) $O(n \log n)$
  - **Smoothsort** $O(n \log n)$

	En place	Stable
Tri à bulles	✓	✓
Tri par sélection	✓	✗
Tri par insertion (Insertionsort)	✓	✓
Tri de Shell (Shellsort)	✓	✗
Tri fusion	✗	✓
Tri rapide (Quicksort)	✓	✗
Tri par tas (Heapsort)	✓	✗
Smoothsort	✓	✗

Algorithmes de tri



Algorithmes de tri



Variante du tri rapide prenant en compte le principe de Sedgewick

Tri : bonus

- http://youtu.be/lyZQPjUT5B4

Modèles de conception

- **Optimiser le temps** de développement
- **Augmenter la qualité** d'une application
- **Minimiser les interactions** entre modules/classes
- Familles :
  - **Construction** :
    - Instanciation/configuration des classes et objets
  - **Structuraux** :
    - Organisation et groupement des classes
  - **Comportementaux** :
    - Collaboration inter-objet et distribution des responsabilités

Quelques modèles de conception

- Singleton
- Fabrique abstraite

Construction

- Objet composite
- Façade
- Décorateur

Structuraux

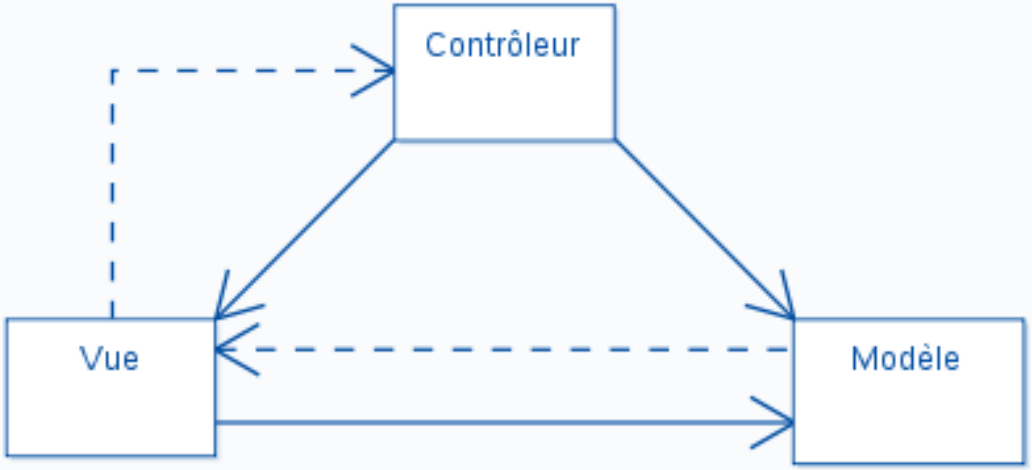
- Observateur
- Stratégie
- Visiteur

Comportementaux

- Injection de contrôle
- Objet d'accès aux données (DAO)

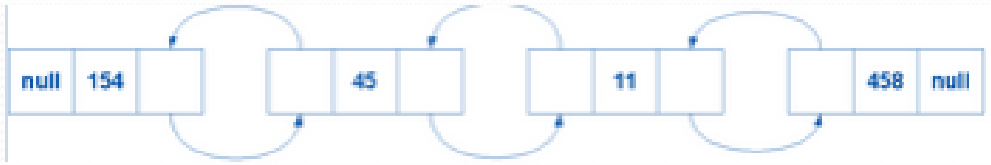
Modèle-vue-contrôleur

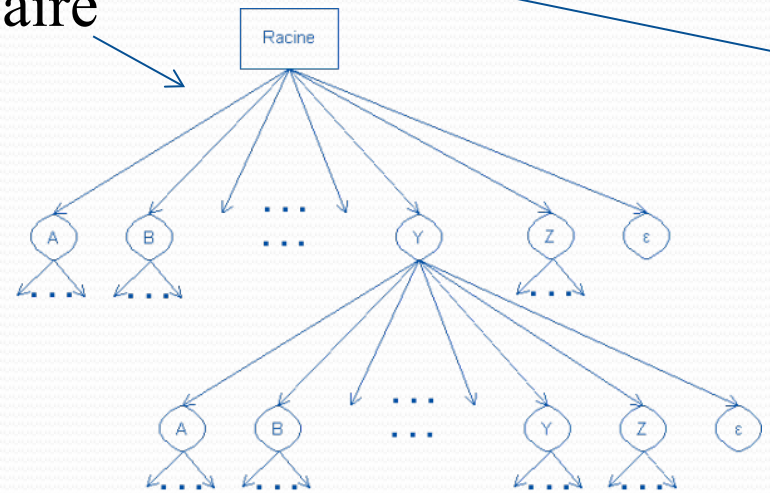
- **Architecture** et **méthode de conception** d'IHM
- **Modèle** : données et leur manipulation
- **Vue** : élément de l'interface graphique
- **Contrôleur** : orchestre les actions, synchronise
- **MVC 2** : un seul contrôleur

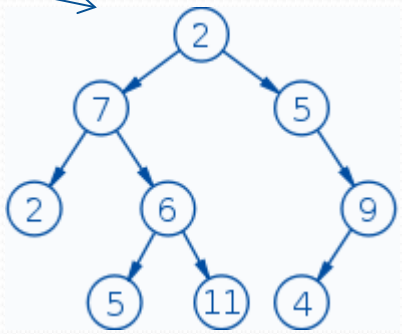


Récursivité structurelle

- **Liste chaînée** (collection ordonnée) :
  - Simplement chaînée
  - Doublement chaînée
- **Arbre** :
  - Binaire
  - N-aire







Optimisation de code

- Ne faire qu'**une fois que le programme fonctionne** et répond aux spécifications fonctionnelles
- Choisir un algorithme de **complexité inférieure**
- Choisir des **structures de données** adaptées
- Bien ordonner les instructions
- Bien utiliser le langage de programmation choisi
- Bien utiliser les bibliothèques du langage
- Penser à l'optimisation automatique (compilateur)
- Utiliser un langage de bas niveau pour les points critiques

Aller plus loin

- Classe de complexité
- Tri utilisant la structure des données, volumineux, bande, Introsort
- Dérécursivité
- Récursivité croisée
- Co-variance/contra-variance
- Théorie des graphes
- Algorithmes évolutionnistes
- Réseaux de neurones

Liens

- Documents classiques :
  - Livres :
    - N.H. Xuong. *Mathématiques discrètes et informatique.*
    - Gamma, Helm, Johnson, Vlissides. *Design Patterns.*

Crédits

Auteur

Mickaël Martin Nevot
mmartin.nevot@gmail.com

Carte de visite électronique

Relecteurs

Cours en ligne sur : www.mickael-martin-nevot.com