Rappels

Rezak AZIZ

CNAM

Méthodologie de résolution de problème

Comment partir d'un problème (Exercice) vers un Programme ?

Du problème au programme

Démarche générale

Problème $\Rightarrow$ Analyse $\Rightarrow$ Algorithme $\Rightarrow$ Programme $\Rightarrow$ Résultat

Du problème au programme

Démarche générale

Problème ⇒ Analyse ⇒ Algorithme ⇒ Programme ⇒ Résultat

- Tout programme commence par une **analyse du problème**.
- L'algorithme est la **traduction logique** de cette analyse.
- Le programme n'est qu'une **traduction technique** de l'algorithme dans un langage.
- Enfin, on vérifie le résultat à travers des tests et des corrections.

Du problème au programme

Démarche générale

Problème ⇒ Analyse ⇒ Algorithme ⇒ Programme ⇒ Résultat

- Tout programme commence par une **analyse du problème**.
- L'algorithme est la **traduction logique** de cette analyse.
- Le programme n'est qu'une **traduction technique** de l'algorithme dans un langage.
- Enfin, on vérifie le résultat à travers des tests et des corrections.

Objectif

Comprendre avant de coder : la qualité du programme dépend de la qualité de l'analyse et de la conception.

Étape 1 : Analyse du problème

- Lire et comprendre l'énoncé du problème.
- Identifier clairement :
  - les **données d'entrée**,
  - les **résultats attendus (sorties)**,
  - les **traitements nécessaires**.
- Déterminer la logique générale de résolution.

Étape 1 : Analyse du problème

- Lire et comprendre l'énoncé du problème.
- Identifier clairement :
  - les **données d'entrée**,
  - les **résultats attendus (sorties)**,
  - les **traitements nécessaires**.
- Déterminer la logique générale de résolution.

Bon réflexe

- Reformuler le problème avec vos propres mots.
- Décomposer le problème en sous-problèmes simples.
- Utiliser des schémas ou exemples pour clarifier les idées.

Étape 1 : Analyse du problème

- Lire et comprendre l'énoncé du problème.
- Identifier clairement :
  - les **données d'entrée**,
  - les **résultats attendus (sorties)**,
  - les **traitements nécessaires**.
- Déterminer la logique générale de résolution.

Bon réflexe

- Reformuler le problème avec vos propres mots.
- Décomposer le problème en sous-problèmes simples.
- Utiliser des schémas ou exemples pour clarifier les idées.

Attention

Ne pas se précipiter vers le code ! Une mauvaise analyse conduit toujours à une mauvaise solution.

Étape 2 : Conception de l'algorithme

Définition

Un **algorithme** est une suite d'actions **finites et ordonnées**, qui, correctement exécutées, produisent le résultat attendu.

Étape 2 : Conception de l'algorithme

Définition

Un **algorithme** est une suite d'actions **finites et ordonnées**, qui, correctement exécutées, produisent le résultat attendu.

- Décrire les actions à réaliser sous forme de pseudo-code ou d'étapes numérotées.
- Chaque action doit être :
  - claire et non ambiguë,
  - réalisable par la machine,
  - limitée dans le temps.
- L'algorithme doit traiter le cas général et les cas particuliers.

Étape 2 : Conception de l'algorithme

Définition

Un **algorithme** est une suite d'actions **finites et ordonnées**, qui, correctement exécutées, produisent le résultat attendu.

- Décrire les actions à réaliser sous forme de pseudo-code ou d'étapes numérotées.
- Chaque action doit être :
  - claire et non ambiguë,
  - réalisable par la machine,
  - limitée dans le temps.
- L'algorithme doit traiter le cas général et les cas particuliers.

Astuce pédagogique

Toujours tester mentalement l'algorithme sur un exemple concret avant de le coder.

Problème

Soit un tableau T à une dimension ayant n valeurs aléatoires. On désire connaître les modes (valeurs fréquentes) de tableau T.

Problème - Etape 1 : Analyser le problème

Problème : Soit un tableau T à une dimension ayant N valeurs aléatoires. On désire connaître les modes (valeurs fréquentes) de tableau T.

Analyse :

- **Entrée :** n la taille du tableau.
- **Sortie :** Afficher une liste de valeurs les plus fréquentes.
- **Idée :** pour un tableau T de n entiers, compter combien de fois chaque valeur apparaît puis garder celles avec le plus grand compteur.

Problème - Etape 1 : Analyser le problème

Problème : Soit un tableau T à une dimension ayant N valeurs aléatoires. On désire connaître les modes (valeurs fréquentes) de tableau T.

Analyse :

- **Entrée :** n la taille du tableau.
- **Sortie :** Afficher une liste de valeurs les plus fréquentes.
- **Idée :** pour un tableau T de n entiers, compter combien de fois chaque valeur apparaît puis garder celles avec le plus grand compteur.

Pas trivial à résoudre !

Problème - Etape 1.1 : Décomposer le problème

Principe

Un problème complexe se résout plus facilement lorsqu'il est divisé en **sous-problèmes simples et précis**.

Plusieurs problèmes à résoudre :

1. Initialiser Aléatoirement le tableau T
2. Etant donné un tableau T, compter le nombre d'occurrences (cardinalités) de chaque valeur de T et renvoyer un tableau *Card* de ces cardinalités.
3. Etant donné un tableau *Card*, Trouver la valeur maximale *max*.
4. Etant donné un tableau *Card*, Afficher tout les index dont la valeur est égale à une valeur *max*.

Problème - Etape 2: Conception de l'algorithme

Il suffit de combiner les sous problème dans un algorithmes

```
ALGORITHME ModesTab
Variables n : ENTIER, T : Tableau d'entiers
DEBUT
  Lire(n)
  T := initTab(n)
  cardinalite := card(T, n)
  maxCard = maxTab(cardinalite)
  afficherCasesEgales(cardinalite, maxFreq)
FIN
```

Problème - Etape 2: Conception de l'algorithme

Il suffit de combiner les sous problème dans un algorithmes

```
ALGORITHME ModesTab
Variables n : ENTIER, T : Tableau d'entiers
DEBUT
  Lire(n)
  T := initTab(n)
  cardinalite := card(T, n)
  maxCard = maxTab(cardinalite)
  afficherCasesEgales(cardinalite, maxFreq)
FIN
```

Maintenant il faut résoudre les sous problèmes un à un.

Sous Problèmes 1 : initTab - Étape 1 et 2

Problème : Initialiser Aléatoirement le tableau T de n
Analyse :

- **Entrée :** n la taille du tableau, un tableau T.
- **Sortie :** un tableau T de n entiers.
- **Idée :** Parcourir le tableau et à chaque case mettre un nombre aléatoire.

Sous Problèmes 1 : initTab - Étape 1 et 2

Problème : Initialiser Aléatoirement le tableau T de n
Analyse :

- **Entrée :** n la taille du tableau, un tableau T.
- **Sortie :** un tableau T de n entiers.
- **Idée :** Parcourir le tableau et à chaque case mettre un nombre aléatoire.

Le problème est simple.

```
FONCTION initTab(n: ENTIER): Tableau
Variables T : Tableau de n d'entiers
DEBUT
  POUR chaque valeur T[i] FAIRE
      Lire(T[i])
  FPOUR
  RETOURNER T
FIN
```

Sous Problèmes 2 card(T,n) - Étape 1 et 2

Problème : Etant donné un tableau T de taille n, compter le nombre d'occurrences (cardinalités) de chaque valeur de T et renvoyer un tableau *Card* de ces cardinalités.

Analyse :

- **Entrée :** T un tableau d'entiers, n la taille de tableau.
- **Sortie :** un tableau *Card* de cardinalités .
- **Idée :** Chaque case du tableau *Card* correspond à un compteur de la valeur correspondant à son index.

Plusieurs problèmes à résoudre :

1. Connaitre la taille de tableau des cardinalités. Revient à trouver le max d'un tableau (Même principe que le sous problème 3).
2. Initialiser le tableau à 0. (Même principe que sous problème 1).
3. Remplir le tableau des cardinalités

Sous Problèmes 2 card(T,n) - Étape 1 et 2

Problème : Etant donné un tableau T de taille n, compter le nombre d'occurrences (cardinalités) de chaque valeur de T et renvoyer un tableau *Card* de ces cardinalités.

Analyse :

- **Entrée :** T un tableau d'entiers, n la taille de tableau.
- **Sortie :** un tableau *Card* de cardinalités .
- **Idée :** Chaque case du tableau *Card* correspond à un compteur de la valeur correspondant à son index.

Plusieurs problèmes à résoudre :

1. Connaitre la taille de tableau des cardinalités. Revient à trouver le max d'un tableau (Même principe que le sous problème 3).
2. Initialiser le tableau à 0. (Même principe que sous problème 1).
3. Remplir le tableau des cardinalités

```
FONCTION card(T: Tableau, n: ENTIER): Tableau
Variable Taille: ENTIER
Variable cardinalite : Tableau[Taille] ENTIERS
DEBUT
  Taille := Max(T)+1
  initTabValeur(cardinalite,0)
  POUR chaque valeur T[i] FAIRE
      cardinalite[T[i]]++
  FPOUR
  RETOURNER cardinalite
FIN
```

Sous Problèmes 3 maxTab - Étape 1 et 2

Problème : Etant donné un tableau *Card*, Trouver la valeur maximale *max*.
Analyse :

- **Entrée :** T un tableau d'entiers, n la taille de tableau.
- **Sortie :** *max* le maximum du tableau.
- **Idée :** parcourir le tableau et extraire le *max*

Le problème est simple

Sous Problèmes 3 maxTab - Étape 1 et 2

Problème : Etant donné un tableau *Card*, Trouver la valeur maximale *max*.
Analyse :

- **Entrée :** T un tableau d'entiers, n la taille de tableau.
- **Sortie :** *max* le maximum du tableau.
- **Idée :** parcourir le tableau et extraire le *max*

Le problème est simple

```
FONCTION maxTab(T: Tableau, n: ENTIER): ENTIER
Variable maximum: ENTIER
DEBUT
  maximum := T[0]
  POUR chaque valeur T[i] FAIRE
     SI T[i] > maximum ALORS
        maximum :=  T[i]
     FSI
  FPOUR
  RETOURNER maximum
FIN
```

Sous Problèmes 4 afficherCasesEgales - Étape 1 et 2

Problème : Etant donné un tableau *Card*, Afficher tout les index dont la valeur est égale à une valeur v.

Analyse :

- **Entrée :** *Card* un tableau d'entiers, n la taille de tableau, v la valeur des index à afficher.
- **Sortie :** Affichage à l'écran.
- **Idée :** parcourir le tableau *Card* et afficher les index des valeurs qui sont égales à v

Le problème est simple

Sous Problèmes 4 afficherCasesEgales - Étape 1 et 2

Problème : Etant donné un tableau *Card*, Afficher tout les index dont la valeur est égale à une valeur *v*.

Analyse :

- **Entrée :** *Card* un tableau d'entiers, *n* la taille de tableau, *v* la valeur des index à afficher.
- **Sortie :** Affichage à l'écran.
- **Idée :** parcourir le tableau *Card* et afficher les index des valeurs qui sont égales à *v*

Le problème est simple

```
FONCTION afficherCasesEgales(T: Tableau, n: ENTIER,
DEBUT
  POUR chaque valeur T[i] FAIRE
     SI T[i] = v ALORS
        ECRIRE(i)
     FSI
  FPOUR
FIN
```

Étape 3 : Passage à la programmation

Traduction de l'algorithme

- Choisir un langage de programmation adapté (C, Java, Python...).
- Traduire chaque étape de l'algorithme en instructions du langage.

Étape 3 : Passage à la programmation

Traduction de l'algorithme

- Choisir un langage de programmation adapté (C, Java, Python...).
- Traduire chaque étape de l'algorithme en instructions du langage.

Validation

- Exécuter le programme sur plusieurs jeux de test :
  - Tableau avec un seul mode.
  - Tableau avec plusieurs modes.
  - Tableau avec toutes les valeurs différentes.
- Vérifier les erreurs de logique et d'affichage.

Rappel sur les Classes et Objets

Comment partir d'un problème (Exercice) vers un Programme ?

Rappel : Classes et Objets en Java

Pourquoi les classes et les objets ?

- En programmation orientée objet, on modélise le **monde réel** sous forme d'objets.
- Chaque objet possède :
  - un **état** (ses données internes),
  - un **comportement** (les actions qu'il peut effectuer),
  - une **identité** (il est unique, même s'il ressemble à un autre).
- Une **classe** est un modèle ou un moule qui décrit la structure commune à plusieurs objets.

Exemple

Une classe `CompteBancaire` décrit un compte :

- Attributs : `solde`, `titulaire`
- Méthodes : `deposer()`, `retirer()`, `afficherSolde()`

Chaque objet `CompteBancaire` aura son propre solde et titulaire.

Structure type d'une classe Java

```
public class CompteBancaire {
    // 1. Attributs
    private String titulaire;
    private double solde;

    // 2. Constructeur
    public CompteBancaire(String t) {
        this.titulaire = t;
        this.solde = 0.0;
    }

    // 3. Méthode
    public void afficherSolde() {
        System.out.println(titulaire +
          " : " + solde + " €");
    }
}
```

Éléments essentiels :

- Les attributs décrivent l'état de l'objet.
- Le constructeur initialise les valeurs au moment de la création.
- Les méthodes définissent le comportement.
- L'objet est créé avec le mot-clé `new`.

Exemple d'utilisation :

```
CompteBancaire c1 = new CompteBancaire("Alice");
c1.afficherSolde(); // Alice : 0.0 €
```

Encapsulation : protéger l'état interne

Principe

- On rend les attributs **privés** pour les protéger.
- On contrôle l'accès via des **getters** et **setters**.
- On garantit ainsi la cohérence de l'objet (ex. pas de coordonnées négatives, pas de solde négatif...).

Mauvaise pratique :

```
class Compte {
    double solde; // public !
}
Compte c = new Compte();
c.solde = -5000; // Incohérent !
```

Bonne pratique :

```
class Compte {
    private double solde;
    public void deposer(double m) {
        if (m > 0) solde += m;
    }
    public double getSolde() {
        return solde;
    }
}
```

Getters et Setters

Rôle :

- Lire et modifier les attributs privés.
- Ajouter des règles de validation (ex : bornes, cohérence).
- Séparer l'**interface publique** de l'**implémentation interne**.

```
public class CompteBancaire {
    private String titulaire;
    private double solde;

    public String getTitulaire() {
        return titulaire;
    }

    public double getSolde() {
        return solde;
    }

    public void setTitulaire(String t) {
        this.titulaire = t;
    }
}
```

Getters et Setters

Rôle :

- Lire et modifier les attributs privés.
- Ajouter des règles de validation (ex : bornes, cohérence).
- Séparer l'**interface publique** de l'**implémentation interne**.

```
public class CompteBancaire {
    private String titulaire;
    private double solde;

    public String getTitulaire() {
        return titulaire;
    }

    public double getSolde() {
        return solde;
    }

    public void setTitulaire(String t) {
        this.titulaire = t;
    }
}
```

Exemple d'utilisation :

```
CompteBancaire c = new CompteBancaire("Alice");
System.out.println(c.getTitulaire());
```

Constructeurs

Définition

Un **constructeur** est une méthode spéciale :

- appelée automatiquement avec `new`,
- porte le même nom que la classe,
- n'a pas de type de retour,
- sert à initialiser les attributs.

```
public class CompteBancaire {
    private String titulaire;
    private double solde;

    // Constructeur
    public CompteBancaire(String t, double s) {
        this.titulaire = t;
        this.solde = s;
    }
}
```

```
// Utilisation
CompteBancaire c =
    new CompteBancaire("Alice", 100.0);
```

Classes de test

Pourquoi tester ?

- Vérifier que le comportement réel correspond au comportement attendu.
- Identifier rapidement les erreurs dans les méthodes.

```
public class TestCompte {
    public static void main(String[] args) {
        CompteBancaire c = new CompteBancaire("Alice", 100);
        c.deposer(50);
        System.out.println("Test solde : " +
          (c.getSolde() == 150 ? "[ok]" : "[erreur]"));
    }
}
```

Idée clé : Une bonne classe se vérifie par des tests simples et automatiques.

Attributs et méthodes de classe

Principe

- Un attribut `static` est partagé par toutes les instances.
- Il appartient à la classe, et non à un objet particulier.
- Exemple : compter le nombre de comptes créés.

```
public class CompteBancaire {
    private static int compteur = 0;
    private String titulaire;

    public CompteBancaire(String t) {
        this.titulaire = t;
        compteur++;
    }

    public static int getCompteur() {
        return compteur;
    }
}
```

```
public class TestStatic {
    public static void main(String[] args) {
        new CompteBancaire("Alice");
        new CompteBancaire("Bob");
        System.out.println(
            CompteBancaire.getCompteur()
        );
        // affiche 2
    }
}
```

Vous avez maintenant toutes les bases !

Place à la pratique avec le **TP 11 : Classes et Tests.**