RAPPORT DE PROJET

BIBLIOTHÈQUE DE COMPRESSION BITPACKING

Préparé par :
BENADY Semy (22207203)

Sous la direction de :
Monsieur Régin J.C.

Université Côte d'Azur

2 novembre 2025

Table des matières

1 INTRODUCTION

Dans le cadre de la matière de Génie Logiciel, ce projet aborde le problème central de la transmission de tableaux d'entiers sur Internet : la compression de données (Bit Packing).

L'objectif principal est d'implémenter plusieurs variantes d'un algorithme de compression et décompression d'entiers, puis d'en analyser les performances et la rentabilité.

Plus précisément, il s'agissait de :

1. **Concevoir une structure logicielle modulaire et extensible.**
2. **Implémenter trois approches distinctes** du Bit Packing :
   - `NoCrossing` (V1) : Sans chevauchement de mots mémoire.
   - `Crossing` (V2) : Avec chevauchement (densité maximale).
   - `Overflow` (V3) : Une méthode hybride optimisée pour les distributions de données mixtes (petits et grands nombres).
3. **Garantir l'accès direct** à l'élément i (fonction `get(i)`) sans nécessiter une décompression complète.
4. **Mesurer le temps d'exécution** (Compression, Décompression, Get) via des bancs d'essai (benchmarks) rigoureux.
5. **Évaluer la rentabilité** de chaque méthode.

Ce rapport détaille l'architecture logicielle choisie, qui a été conçue comme une bibliothèque robuste. Il présente les stratégies de résolution pour chaque algorithme, les défis de conception surmontés, et conclut par une analyse comparative des performances basée sur les benchmarks exécutés.

2 ARCHITECTURE LOGICIELLE

L'architecture du projet a été pensée pour passer d'un simple script à une bibliothèque logicielle modulaire, professionnelle et extensible. La philosophie de conception centrale est que **le tableau compressé (`int[]`) est autosuffisant** : il contient un en-tête (header) binaire avec toutes les métadonnées nécessaires à sa propre lecture.

L'architecture repose sur plusieurs piliers : le patron de conception **Stratégie** (Strategy Pattern), l'**Abstraction** de la logique de bas niveau, et des **Fabriques** (Factories) pour le découplage.

Voici une analyse détaillée de chaque classe du projet.

2.1 Le Cœur de l'API (L'interface publique)

Ces classes définissent le contrat public de la bibliothèque.

2.1.1 IntCompressor.java

- **Rôle** : C'est le contrat public et le cœur du Strategy Pattern. Elle définit ce qu'un "compresseur" doit savoir faire, sans imposer de détails d'implémentation.

- **Méthodes Clés** :
  - `int[] compress(int[] src)` : Prend un tableau source et retourne un **nouveau** tableau `int[]` autosuffisant, qui contient à la fois l'en-tête et les données compressées.
  - `void decompress(int[] compressed, int[] dst)` : Prend le tableau compressé et un tableau de destination (`dst`) pré-alloué. Ce choix de conception évite une allocation mémoire (coûteuse) dans la méthode de décompression.
  - `int get(int[] compressed, int index)` : La fonction d'accès direct. Elle lit l'en-tête de `compressed` et ne récupère que les bits correspondant à l'élément `index`, garantissant un accès en $O(1)$.

2.1.2 CompressionType.java

- **Rôle** : Définit de manière sécurisée les trois stratégies disponibles : `CROSSING`, `NO_CROSSING`, `OVERFLOW`.
- **Justification** : J'ai utilisé un `enum` car c'est une pratique d'ingénierie logicielle supérieure à l'utilisation de `String` (comme "overlap"). Cela garantit la sécurité de type (le compilateur empêche les fautes de frappe) et rend le code (comme les `switch` des factories) plus propre et plus sûr.

2.2 La Hiérarchie d'Implémentation (Les Stratégies)

Ces classes contiennent la logique métier de chaque algorithme.

2.2.1 BitPackingBase.java

- **Rôle** : Sert de classe mère commune aux trois stratégies pour partager le code et éviter la duplication (principe DRY - Don't Repeat Yourself).
- **Méthodes Clés** :
  - `protected int computeKAuto(int[] src)` : Analyse le tableau source (`src`) pour trouver la valeur maximale et en déduire le nombre de bits (k) minimal requis pour la représenter. Elle utilise `32 - Integer.numberOfLeadingZeros(max)` pour un calcul efficace.
  - `protected int[] allocWithHeader(int headerWords, int dataBits)` : Méthode utilitaire qui alloue le tableau `int[]` de sortie. Elle calcule la taille totale nécessaire (en `long` pour éviter les débordements) pour l'en-tête et les données, en arrondissant au mot de 32 bits supérieur.

2.2.2 BitPackingCrossing.java

- **Rôle** : Implémente la compression la plus dense, où les entiers de k bits peuvent être "à cheval" sur deux mots de 32 bits.
- **Fonctionnement** :
  - `compress(src)` : Calcule k, alloue `out`, écrit l'en-tête, puis boucle de $i = 0$ à n. Pour chaque entier, il calcule sa position absolue en bits (`bitPos = baseBit + i * k`) et **délègue** l'écriture binaire complexe à `BitIO.writeBitsLSB(out, bitPos, k, src[i])`.
  - `get(compressed, index)` : Lit k de l'en-tête, calcule le `bitPos` absolu, et **délègue** la lecture à `BitIO.readBitsLSB(compressed, bitPos, k)`.

2.2.3 BitPackingNoCrossing.java

- **Rôle** : Implémente la compression alignée. Chaque entier de k bits est garanti de tenir dans un seul mot de 32 bits.
- **Fonctionnement** :
  - `compress(src)` : Calcule k et `elementPerWord = 32 / k`. Il alloue `out` et écrit l'en-tête. En bouclant, il calcule le `wordIndex` et le `bitOffset` pour chaque i, puis **délègue** l'écriture à la version optimisée `BitIO.writeBitsInWordLSB(...)`.
  - `get(compressed, index)` : Applique la même logique de calcul d'index et appelle `BitIO.readBitsInW` pour une lecture rapide.
  - **Compromis** : Cette méthode est plus rapide en `get` (car elle ne lit qu'un mot mémoire), mais elle gaspille de l'espace ($32 \pmod k$ bits sont perdus par mot).

2.2.4 BitPackingOverflow.java

- **Rôle** : Stratégie adaptative "intelligente" pour les données hétérogènes (99% de petites valeurs, 1% de "outliers" très grands).
- **Structure des données** :

```
[HEADER] [ZONE PRINCIPALE (Flags + Payloads)] [ZONE OVERFLOW (Valeurs
↪  brutes)]
```

- **Méthodes Clés** :
  - `public static BestKResult findBestK(int[] array, int n)` : Le "cerveau" de l'algorithme. Cette méthode teste tous les k possibles de 1 à `maxBits`. Pour chaque k, elle calcule le coût total en bits :

$$\text{Coût} = (n \times \text{bitsParEntrée}) + (m \times \text{bitsParOutlier})$$

    où m est le nombre d'outliers et $\text{bitsParEntrée} = 1(\text{flag}) + \max(k, \log_2(m))$. Elle retourne le k qui minimise ce coût total.
  - `compress(src)` :
    1. Appelle `findBestK` pour obtenir le plan de compression optimal.
    2. Alloue `out` et écrit un en-tête détaillé.
    3. Boucle sur `src` : Si v est "normal", il écrit `(FLAG=0 | VALEUR)` dans la zone principale. Si v est un "outlier", il l'écrit d'abord dans la zone `overflow`, puis écrit `(FLAG=1 | INDEX_OVERFLOW)` dans la zone principale.
  - `get(compressed, index)` : Lit l'entrée de la zone principale à `index`. Il vérifie le bit de flag : si c'est `0`, il retourne le payload ; si c'est 1, il utilise le payload comme index pour aller lire la valeur brute dans la zone overflow.

2.3 Les Piliers Techniques (Infrastructure)

Ces classes `final` fournissent les services de bas niveau essentiels à l'architecture.

2.3.1 Headers.java

- **Rôle** : Gère le format binaire de l'en-tête. Il rend le tableau `int[]` autosuffisant et remplace l'ancienne classe `BitPackedArray`.
- **Attributs Clés** :
  - `HEADER_WORDS = 5` : Définit la taille de l'en-tête (20 octets).
  - `MAGIC = 0x1A2B3C4D` : Un "nombre magique" pour valider qu'un `int[]` est bien un tableau compressé par cette bibliothèque.
- **Méthodes Clés** :
  - `static void write(...)` : Écrit les 5 mots de l'en-tête (`MAGIC`, n, mode, k, métadonnées packées) au début du tableau `out`.
  - `static void checkMagic(int[] in)` : Appelé par toutes les méthodes de lecture pour valider le `MAGIC`.
  - `static int n(int[] in)`, `static CompressionType mode(int[] in)`, ... : "Getters" qui lisent et décodent les métadonnées depuis l'en-tête.

2.3.2 BitIO.java

- **Rôle** : Abstrait **toute** la manipulation binaire de bas niveau (Principe de Responsabilité Unique). Les stratégies lui délèguent le travail.
- **Méthodes Clés** :
  - `static void writeBitsLSB(int[] words, int bitPos, int bitLen, int value)` : Écrit `bitLen` bits à une position absolue `bitPos`. Gère le cas de **chevauchement** en calculant `first` (bits dans `words[wordIndex]`) et `rest` (bits dans `words[wordIndex + 1]`).
  - `static int readBitsLSB(...)` : Fait l'opération inverse, en lisant sur un ou deux mots et en "recollant" les bits.
  - `static void writeBitsInWordLSB(...)` / `static int readBitsInWordLSB(...)` : Versions **optimisées** pour `NoCrossing`. Elles partent du principe que l'opération se fait sur un seul mot.

2.4 Les Points d'Entrée (Fabriques et Clients)

Ces classes sont les points d'entrée pour utiliser la bibliothèque ou la tester.

2.4.1 CompressorFactory.java

- **Rôle** : Le point d'entrée principal et moderne de la bibliothèque (Patron de Fabrique).
- **Méthode Clé** : `public static IntCompressor create(CompressionType type)`
- **Fonctionnement** : Un simple `switch` sur l'**enum** `CompressionType` qui instancie et retourne la stratégie demandée.

2.4.2 BitPackingFactory.java

- **Rôle** : Assurer la **rétrocompatibilité** avec l'ancien `Main.java` qui utilise des `String` en français.
- **Fonctionnement** : Implémente le **Patron d'Adaptateur**.
- **Méthode Clé** : `public static IntCompressor createBitPacking(String type)`
  - Prend un `String` (ex: "Chevauchement").
  - Le "traduit" en enum (ex: `CompressionType.CROSSING`).
  - Appelle la **nouvelle** `CompressorFactory` et retourne le résultat.

2.4.3 Main.java

- **Rôle** : Un banc d'essai interactif complet pour tester une configuration à la fois.
- **Fonctionnement** : La classe `main` est un simple coordinateur. Le travail est délégué à des **classes internes** statiques :
  - `UI` : Gère tout l'affichage et le dialogue avec l'utilisateur.
  - `DataGenerator` : Gère la création des données de test.
  - `BenchmarkRunner` : Gère le protocole de test (avec "warmups").
  - `InputHelper` / `FormatHelper` : Gèrent la validation des saisies et le formatage des résultats (notamment `median`, plus robuste que la moyenne).

2.4.4 AutomatedBenchmark.java

- **Rôle** : Un banc d'essai **non interactif** conçu pour **comparer** les 3 stratégies.
- **Fonctionnement** :
  - `runAllScenarios()` : Définit 3 scénarios de test fixes (k=9, k=8, outliers).
  - `runScenario(...)` : Pour un scénario donné, il **boucle sur les 3 `CompressionType`** et exécute les benchmarks (temps et taille) pour chacun, affichant un tableau comparatif.

2.4.5 SmokeTest.java

- **Rôle** : Test de non-régression (ou "test de fumée").
- **Fonctionnement** : Crée un petit tableau, boucle sur les 3 stratégies, compresse, décompresse, et vérifie que `Arrays.equals(original, recovered)` est `true`.

3 STRATÉGIES DE RÉSOLUTION ET IMPLÉMENTATION

L'un des choix de conception majeurs de ce projet a été d'abstraire la manipulation binaire complexe dans la classe `BitIO`. Grâce à cela, les trois classes de stratégie (`Crossing`, `NoCrossing`, `Overflow`) ne contiennent que la logique "métier" (comment les données sont organisées) et délèguent l'écriture/lecture de bas niveau.

J'ai implémenté toutes les stratégies en suivant la convention LSB-first (Least Significant Bit first), où l'écriture se fait du bit 0 vers le bit 31 à l'intérieur d'un mot `int`.

3.1 V1 : `BitPackingNoCrossing` (Sans Chevauchement)

Cette stratégie est la plus simple. Elle privilégie la vitesse d'accès (`get`) au détriment de l'espace de stockage.

- **Principe** : Garantir que chaque entier de k bits est stocké **entièrement** dans un seul mot de 32 bits. Il ne peut jamais être "à cheval" sur deux mots.
- **Compromis (Inconvénient)** : Cette approche génère du gaspillage. Pour $k = 9$, on ne peut stocker que 3 entiers par mot (27 bits), gaspillant ainsi $32 \pmod 9 = 5$ bits par mot.
- **Logique `compress(src)`** :
  1. Je calcule d'abord le nombre d'entiers par mot : `elementsPerWord(k)`.
  2. J'alloue le tableau de sortie et j'écris l'en-tête (via `Headers.write`).
  3. Je boucle de $i = 0$ à n. Pour chaque i, je calcule ses coordonnées via les *helpers* :
     - `wordIndexFor(i, ...)` (le mot où il sera stocké).
     - `bitOffsetFor(i, ...)` (le décalage de début dans ce mot).
  4. Je **délègue** l'écriture à la méthode optimisée de `BitIO` : `BitIO.writeBitsInWordLSB(out, baseWord() + wordIndex, bitOffset, k, src[i])`.
- **Logique `get(compressed, index)`** :
  1. La lecture est le miroir exact de l'écriture. Je lis k depuis l'en-tête.
  2. Je calcule `elementsPerWord`, `wordIndex` et `bitOffset` en utilisant les mêmes *helpers*.
  3. Je **délègue** la lecture à `BitIO.readBitsInWordLSB(...)`.
- **Conclusion** : L'implémentation est simple et `get` est très rapide car il n'implique qu'une seule lecture en mémoire (`words[wordIndex]`) et une seule opération binaire.

3.2 V2 : `BitPackingCrossing` (Avec Chevauchement)

Cette stratégie privilégie la densité de stockage maximale.

- **Principe** : Traiter le tableau de sortie comme un flux de bits continu. Un entier de k bits est simplement écrit à la suite du précédent, quitte à ce qu'il commence dans `words[j]` et se termine dans `words[j+1]`.
- **Compromis (Inconvénient)** : L'accès `get` peut être légèrement plus lent, car il doit potentiellement lire *deux* mots en mémoire (`words[j]` et `words[j+1]`) et recombiner les morceaux.
- **Logique `compress(src)`** :
  1. Je calcule k (via `BitPackingBase`).
  2. J'alloue le tableau et j'écris l'en-tête.
  3. Je boucle de $i = 0$ à n. La logique d'adressage est ici plus simple :
     - Je calcule la position **absolue** en bits : `bitPos = baseBit + (i * k)`.
  4. Je **délègue** tout le travail complexe à `BitIO` : `BitIO.writeBitsLSB(out, bitPos, k, src[i])`. C'est `BitIO` qui se charge de calculer le `wordIndex`, l'offset, et de gérer le cas du chevauchement (`first` et `rest`).

- **Logique `get(compressed, index)`** :
  1. Je lis k de l'en-tête.
  2. Je calcule la même position absolue : `bitPos = baseBit + (index * k)`.
  3. Je **délègue** la lecture complexe à `BitIO.readBitsLSB(...)`.
- **Conclusion** : Grâce à l'abstraction de `BitIO`, la logique de cette stratégie est devenue la plus simple à écrire (une simple multiplication), tout en étant la plus dense en stockage.

3.3 V3 : `BitPackingOverflow` (Gestion de Débordement)

C'est la stratégie la plus complexe et la plus "intelligente", conçue pour les jeux de données hétérogènes (ex: 99% de petites valeurs, 1% de "outliers" très grands).

- **Principe** : Au lieu d'utiliser le k maximal (défini par l'outlier), je choisis un k optimal pour la majorité des données. Les valeurs qui dépassent ce k sont stockées dans une "zone de débordement" séparée.
- **Structure des données** : Le tableau de sortie est divisé en trois zones :
  1. `[EN-TÊTE]` (5 mots)
  2. `[ZONE PRINCIPALE]` (Contient n paquets)
  3. `[ZONE OVERFLOW]` (Contient m valeurs brutes, où m est le nombre d'outliers)
- **Logique `findBestK(src)`** (le "cerveau" de l'algorithme) :
  1. J'ai implémenté une heuristique qui teste tous les k possibles (de 1 à `maxBits`).
  2. Pour chaque k, elle simule le coût total en bits :
     - Elle compte le nombre d'outliers m (valeurs $> (1 \ll k) - 1$).
     - Elle calcule les bits requis pour *indexer* ces outliers : `indexBits = log2(m)`.
     - Un paquet dans la zone principale doit contenir un **flag** (1 bit) et un **payload** (soit la valeur, soit l'index). J'ai fait le choix de conception de rendre ce payload uniforme : `payloadBits = max(k, indexBits)`.
     - La taille d'une entrée principale est donc `bitsParElement = 1 + payloadBits`.
     - Le coût total (calculé en `long`) est : `Coût = (n * bitsParElement) + (m * bitsParOverflo`
  3. Cette méthode retourne le k (et les métadonnées associées) qui **minimise** ce coût total.
- **Logique `compress(src)`** :
  1. Appelle `findBestK` pour obtenir le plan de compression (k, `bitsParElement`, etc.).
  2. Alloue le tableau `out` et écrit un en-tête détaillé (via `Headers.write`).
  3. Boucle sur `src` :
     - **Si v est "normal"** ($v \leq$ maxMain) : J'écris `(FLAG=0 | v)` dans la zone principale, en utilisant `BitIO.writeBitsLSB`.
     - **Si v est un "outlier"** :
       (a) J'écris d'abord la valeur brute v dans la zone `overflow` (calculée via le *helper* `overflowBitPos`).

(b) J'écris `(FLAG=1 | overflowIndex)` dans la zone principale.

(c) J'incrémente `overflowIndex`.

- **Logique `get(compressed, index)`** :
  1. Lit l'en-tête détaillé (via `Headers.n`, `Headers.bitsPerElement`, etc.).
  2. Calcule le `bitPos` de l'entrée `index` dans la zone principale (via le *helper* `mainBitPos`).
  3. Lit le paquet de `bitsParElement` (via `BitIO.readBitsLSB`).
  4. J'isole le bit de flag (via le *helper* `flagMask`).
  5. **Si `flag == 0`** : Je retourne le payload (isolé avec `indexMask`), qui est la valeur.
  6. **Si `flag == 1`** : J'utilise le payload comme `overflowIndex` pour calculer une nouvelle position dans la zone overflow (via `overflowBitPos`), et je lis la valeur brute à cet endroit.

4 DÉFIS DE CONCEPTION ET SOLUTIONS APPORTÉES

Durant le développement de la bibliothèque, plusieurs défis de conception ont été rencontrés. L'objectif a été de les résoudre en améliorant la robustesse et la maintenabilité du code, tout en conservant une performance optimale.

4.1 Centralisation de la logique de validation

- **Problème:** Initialement, la logique de validation des tableaux (vérification des nuls, valeurs négatives, calcul de k) était dispersée. Cela entraînait une duplication de code et des responsabilités peu claires.
- **Solution:** La logique de validation a été rationalisée. La logique partagée (comme `computeKAuto` qui détermine le k maximal) a été déplacée dans la classe de base `BitPackingBase`. Cela a permis de rendre les implémentations plus cohérentes et centralisées.

4.2 Classe `Main.java` monolithique

- **Problème:** La classe `Main.java` était devenue monolithique. Elle gérait à la fois l'interface utilisateur (lecture des entrées, affichage des menus), la génération des données de test, l'exécution du protocole de benchmark (warmups, mesures) et le formatage des résultats. Cela la rendait difficile à lire, à déboguer et à maintenir.
- **Solution:** J'ai appliqué une Séparation des Responsabilités (Separation of Concerns). La classe a été divisée en plusieurs classes internes statiques, chacune avec un rôle unique : `UI` (gestion de l'affichage), `DataGenerator` (génération des données), `BenchmarkRunner` (exécution des tests), `InputHelper` et `FormatHelper` (utilitaires). La méthode `main` est désormais un simple coordinateur de ces composants, ce qui améliore radicalement la lisibilité.

4.3 Duplication de la logique d'adressage

- **Problème:** À l'intérieur des stratégies comme `BitPackingOverflow` et `BitPackingNoCrossing`, les calculs d'adressage (trouver le bon mot, le bon offset, appliquer les masques) étaient répétés dans les méthodes `compress`, `decompress`, et `get`.

- **Solution:** Cette logique complexe a été extraite dans des méthodes utilitaires (*helpers*) privées et bien nommées (ex: `mainBitPos`, `flagMask` pour `Overflow`; `wordIndexFor`, `bitOffsetFor` pour `NoCrossing`). Ce changement a éliminé la duplication, rendu les méthodes publiques beaucoup plus claires et a centralisé la logique métier en un seul endroit par classe.

4.4 Indépendance des données (Format autosuffisant)

- **Problème:** Une conception précédente liait les données compressées à un objet "wrapper" Java. Cet objet contenait les métadonnées ('n', 'k') et une référence à l'instance de compression. Ce format n'était pas portable : il était impossible de sauvegarder le `int[]` sur disque ou de l'envoyer sur le réseau et de le relire, car les métadonnées étaient perdues.
- **Solution:** La conception a été modifiée pour rendre le tableau `int[]` compressé totalement **autosuffisant**. J'ai implémenté la classe `Headers.java`, qui écrit un en-tête binaire standardisé (5 entiers) au tout début du tableau. Cet en-tête contient un `MAGIC` (pour vérification), la taille 'n', le mode de compression, et le 'k' utilisé. Ainsi, les méthodes `decompress` et `get` n'ont besoin de rien d'autre que le tableau `int[]` lui-même pour fonctionner.

4.5 Sécurisation contre le Débordement d'Entier (Integer Overflow)

Problème Initial Un problème critique a été identifié lorsque les calculs de taille de tableau dépassaient la limite maximale d'un `int` (2,147,483,647). En Java, un `int` qui dépasse cette limite "boucle" et devient négatif. Lorsque j'essayais d'utiliser ces nombres négatifs pour allouer des tableaux, le programme plantait avec une `ArrayIndexOutOfBoundsException` ou une `NegativeArraySizeException`.

Corrections Apportées Pour robustifier la bibliothèque, j'ai implémenté plusieurs corrections, principalement dans `BitPackingBase.java`.

4.5.1 Sécurisation de `allocWithHeader` (dans `BitPackingBase.java`)

Problème: La méthode d'allocation d'origine effectuait le calcul de `totalBits` en `int`, ce qui provoquait un débordement silencieux.

```
protected int[] allocWithHeader(int headerWords, int dataBits) {
    if (headerWords < 0) headerWords = 0;
    int totalBits = headerWords * 32 + dataBits;  // □ DANGER : peut
    ↪  déborder !
    int words = (totalBits + 31) / 32;
    return new int[words];
}
```

Solution: J'ai modifié la méthode pour utiliser des `long` pour tous les calculs intermédiaires. J'ai également ajouté des gardes-fous pour vérifier que le résultat final pouvait tenir dans un `int` avant de tenter l'allocation, fournissant des messages d'erreur clairs.

```
protected int[] allocWithHeader(int headerWords, int dataBits) {
    if (headerWords < 0) headerWords = 0;
```

```
    if (dataBits < 0) throw new IllegalArgumentException("dataBits cannot
    ↪  be negative: " + dataBits);

    // □ SÉCURISÉ : utilise long pour éviter les débordements
    long totalBitsLong = (long) headerWords * 32L + (long) dataBits;
    if (totalBitsLong > Integer.MAX_VALUE) {
        throw new IllegalArgumentException("Data too large: total bits
        ↪  would be " + totalBitsLong +
            " which exceeds maximum supported size of " +
            ↪  Integer.MAX_VALUE + " bits");
    }

    long wordsLong = (totalBitsLong + 31L) / 32L;
    if (wordsLong > Integer.MAX_VALUE) {
        throw new IllegalArgumentException("Array too large: would need "
        ↪  + wordsLong +
            " words which exceeds maximum array size");
    }

    int words = (int) wordsLong;
    return new int[words];
}
```

4.5.2 Amélioration des Messages d'Erreur dans `computeKAuto`

Pour aider au diagnostic, j'ai amélioré le message d'erreur lors de la détection d'une valeur négative (souvent un symptôme d'un débordement en amont).

Problème:

```
if (v < 0) throw new IllegalArgumentException("Negative value not
↪  supported: " + v);
```

Solution:

```
if (v < 0) throw new IllegalArgumentException("Negative value not
↪  supported: " + v +
    ". This may be caused by integer overflow when creating the input
    ↪  data.");
```

Justification: Le nouveau message guide l'utilisateur vers la cause probable (un débordement lors de la création des données d'entrée) au lieu d'un simple plantage.

4.5.3 Ajout d'un Test de Validation (`OverflowTest.java`)

Pour valider ces corrections de manière robuste, j'ai créé un nouveau fichier de test, `OverflowTest.java` (non inclus dans ce rapport). Ce test vérifie spécifiquement les cas limites :

- Test avec des valeurs proches de `Integer.MAX_VALUE`.
- Test avec des données intentionnellement trop grandes pour être compressées (pour vérifier que l'exception est levée).

- Test avec des valeurs négatives pour confirmer que l'erreur est gérée.

Résumé des Gains de Robustesse

- **Technique:** J'utilise désormais des `long` pour tous les calculs de taille intermédiaires et je vérifie les dépassements avant de convertir en `int`.
- **Fonctionnel:** Au lieu de plantages mystérieux (comme `NegativeArraySizeException`), la bibliothèque lève désormais des `IllegalArgumentException` avec des messages clairs, expliquant que les données sont trop volumineuses.

5 ANALYSE DES BENCHMARKS ET RECOMMANDATIONS

Pour évaluer le compromis performance/taille de chaque stratégie, j'ai utilisé le banc d'essai `AutomatedBenchmark.java`. Ce script exécute trois scénarios de test sur un million d'entiers ($n = 1,000,000$), ciblant les cas d'utilisation spécifiques de chaque algorithme.

L'analyse de ces scénarios révèle qu'il n'existe pas de "meilleur" compresseur universel ; le choix optimal dépend entièrement de la distribution des données d'entrée.

5.1 Scénario 1 : Données Uniformes (k=9, Cas "Imparfait")

- **Contexte** : Données homogènes où $k = 9$ bits. C'est un cas "imparfait" car 32 n'est pas un multiple de 9 ($32 \pmod 9 = 5$).
- **Analyse des performances** :
  - **Taille et Taux de Compression** : `Crossing` (V2) est le grand gagnant. Il ne perd aucun bit et produit la taille la plus compacte, affichant le meilleur taux de compression (ex: ≈ 3.55x). `NoCrossing` (V1) gaspille 5 bits par mot, ce qui entraîne une taille finale significativement plus grande et un taux de compression plus faible (ex: ≈ 2.9x).
  - **Vitesse** : `NoCrossing` (V1) est le plus rapide sur `get(i)` (lecture simple dans un mot). `Crossing` (V2) est légèrement plus lent en `get(i)` car il doit parfois lire sur deux mots. `Overflow` (V3) est inutile ici et son analyse `findBestK` lui fait perdre du temps en compression.

5.2 Scénario 2 : Données Uniformes (k=8, Cas "Parfait")

- **Contexte** : Données homogènes où $k = 8$ bits. C'est le cas "parfait" car 32 est un multiple de 8 ($32 \pmod 8 = 0$).
- **Analyse des performances** :
  - **Taille et Taux de Compression** : `NoCrossing` (V1) et `Crossing` (V2) sont **exactement identiques** en taille. Puisque k s'aligne parfaitement, aucun bit n'est gaspillé. Les deux affichent un taux de compression optimal de ≈ 4.0x (32 bits / 8 bits).
  - **Vitesse** : `NoCrossing` (V1) est le **vainqueur absolu**. Sa logique de `get` est la plus simple (pas de vérification de chevauchement), ce qui la rend marginalement plus rapide.

5.3 Scénario 3 : Données Hétérogènes (Outliers)

- **Contexte** : Données mixtes, avec 99% de petites valeurs ($k \approx 6$) et 1% de "outliers" très grands ($k \approx 20$).
- **Analyse des performances** :
  - **Taille et Taux de Compression** : `Overflow` (V3) est le **vainqueur incontesté**. Les stratégies V1 et V2 sont non adaptatives : elles doivent utiliser le k maximal ($k = 20$) pour *tous* les éléments, résultant en un taux de compression faible (≈ 1.6x). `Overflow`, lui, utilise son heuristique pour choisir un k optimal (ex: $k = 6$ ou 7) et ne stocke que les 1% d'outliers. Son taux de compression est massivement meilleur (ex: ≈ 4.2x), surpassant même les scénarios uniformes.
  - **Vitesse** : `Overflow` (V3) est de loin le plus lent en **compression** à cause du coût de l'analyse `findBestK`. Cependant, ce coût est un investissement unique qui est presque toujours justifié par le gain de place. En `get(i)`, sa performance est excellente (presque aussi rapide que `Crossing` pour les 99% de cas où `flag == 0`).

5.4 Recommandations : Quelle stratégie choisir ?

Basé sur cette analyse, voici mes recommandations :

1. **Cas 1 : Données Hétérogènes (Outliers)**
   - **CHOIX : `Overflow` (V3)**
   - **Justification :** C'est le seul algorithme conçu pour ce cas. Si vos données contiennent ne serait-ce que 0.1% de valeurs aberrantes qui faussent le k maximal, `Overflow` offrira un gain de place spectaculaire. Le coût élevé de la compression est un compromis acceptable pour cet énorme gain en taille.
2. **Cas 2 : Données Homogènes, k "Parfait" (k divise 32)**
   - **CHOIX : `NoCrossing` (V1)**
   - **Justification :** Dans ce scénario (ex: $k = 8$ ou $k = 16$), `NoCrossing` offre la **même taille** que `Crossing` (un taux de ≈ 4.0x pour $k = 8$) mais avec des opérations `get` et `decompress` plus rapides, car sa logique d'adressage est plus simple (garantie de lire un seul mot mémoire).
3. **Cas 3 : Données Homogènes, k "Imparfait" (k ne divise pas 32)**
   - **CHOIX : `Crossing` (V2)**
   - **Justification :** Dans ce cas (ex: $k = 9$, $k = 11$, $k = 17$), `NoCrossing` gaspille trop d'espace (ex: taux de ≈ 2.9x pour $k = 9$). `Crossing` est la meilleure solution car il garantit la densité de stockage maximale (taux de ≈ 3.55x pour $k = 9$) avec des performances de lecture/écriture très rapides.

6 CONCLUSION

Ce projet m'a permis d'implémenter et de valider en profondeur l'efficacité de trois architectures de Bit Packing distinctes.

L'analyse des benchmarks, rendue possible par la création d'un banc d'essai automatisé, démontre qu'il n'existe pas de "meilleure" solution universelle. Le choix optimal de la stra-

tégie dépend entièrement de la distribution des données d'entrée : `NoCrossing` pour la vitesse sur des k parfaits, `Crossing` pour la densité sur des k imparfaits, et `Overflow` pour la robustesse face aux données hétérogènes.

Au-delà des algorithmes, ce projet a souligné l'importance cruciale des choix d'ingénierie logicielle. La transition d'un prototype vers une bibliothèque robuste – en implémentant une API claire (`IntCompressor`), en abstaitsant la logique complexe (`BitIO`), et en assurant l'indépendance des données (`Headers`) – a été fondamentale pour créer un code maintenable, réutilisable et robuste face aux défis de conception comme les débordements d'entiers.

J'espère que ce rapport et le travail réalisé tout au long de ce projet auront su en refléter la rigueur et l'implication.