RAPPORT DU PROJET DE BITPACKING – SOFTWARE ENGINEERING 2025 - OUAHRANI KHALDI Sofiane

1. INTRODUCTION

Dans le cadre de la matière de Génie Logiciel, ce projet donné 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 :
    - **Aligned (V1)** : Sans chevauchement.
    - **Overlap (V2)** : Avec chevauchement.
    - **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 passer par une décompression.
4. Mesurer le temps d'exécution (Compression, Décompression, Get).
5. Évaluer la rentabilité de chaque méthode.

   Ce rapport détaille l'architecture logicielle choisie, les stratégies de résolution pour chaque algorithme, les problèmes d'ingénierie rencontrés, et conclut par une analyse comparative des performances basée sur les benchmarks exécutés.

2. ARCHITECTURE LOGICIELLE

J'ai conçu l'architecture logicielle pour qu'elle soit simple, modulaire et évolutive.
Elle repose sur une classe abstraite commune (*BitPacking*), un Design Pattern "*Factory*" (*CompressionFactory*) et un lanceur (*Main*) sans oublier bien sûr les 3 classes héritant de *BitPacking* : *BitPackingAligned*, *BitPackingOverlap* et *BitPackingOverflow*.

2.1. BitPacking.java (Classe Abstraite)

Cette classe est le socle du projet. Elle centralise la logique partagée et définit le contrat pour toutes les implémentations.

Variables d'état :

- *INT_BITS : Constante (32 bits).*
- *k : Le nombre de bits (K) requis pour la compression.*
- *originalSize : La taille du tableau d'origine.*

Méthodes *abstraites* :

- *compress(int[] input)*
- *decompress(int[] compressedArray)*
- *get(int[] compressedArray, int i)*

Méthodes partagées :

- *calculateK(int[] input) :* Cette fonction détermine la valeur maximale du tableau d'entrée et calcule le k minimal nécessaire en utilisant Integer.numberOfLeadingZeros(maxVal).
- *ResultWithTime<T>* (ma classe interne générique) : J'ai implémenté cette classe pour résoudre un problème de performance. Auparavant, je devais appeler *compress()* une fois pour obtenir le résultat, puis *timeCompress()* une seconde fois pour obtenir le temps, ce qui doublait le coût CPU. Cette classe encapsule le résultat et le temps d'exécution en un seul objet, optimisant ainsi le chronométrage.
- *timeCompress(), timeDecompress(), timeGet()* : Ces méthodes utilisent *System.nanoTime()* et la classe *ResultWithTime<T>* pour chronométrer l'exécution des méthodes abstraites.

2.2. CompressionFactory.java

Cette classe implémente le **Design Pattern Factory**, comme demandé dans l'énoncé.

- **Rôle :** Elle découple le code client (Main.java) des implémentations concrètes.
- **Constantes :** Définit des chaînes statiques (TYPE_ALIGNED, TYPE_OVERLAP, TYPE_OVERFLOW) pour éviter les erreurs de saisie.
- **Méthode createCompressor(String type) :** Une méthode statique qui utilise un switch pour instancier et retourner la classe de compression demandée (BitPackingAligned, BitPackingOverlap ou BitPackingOverflow).

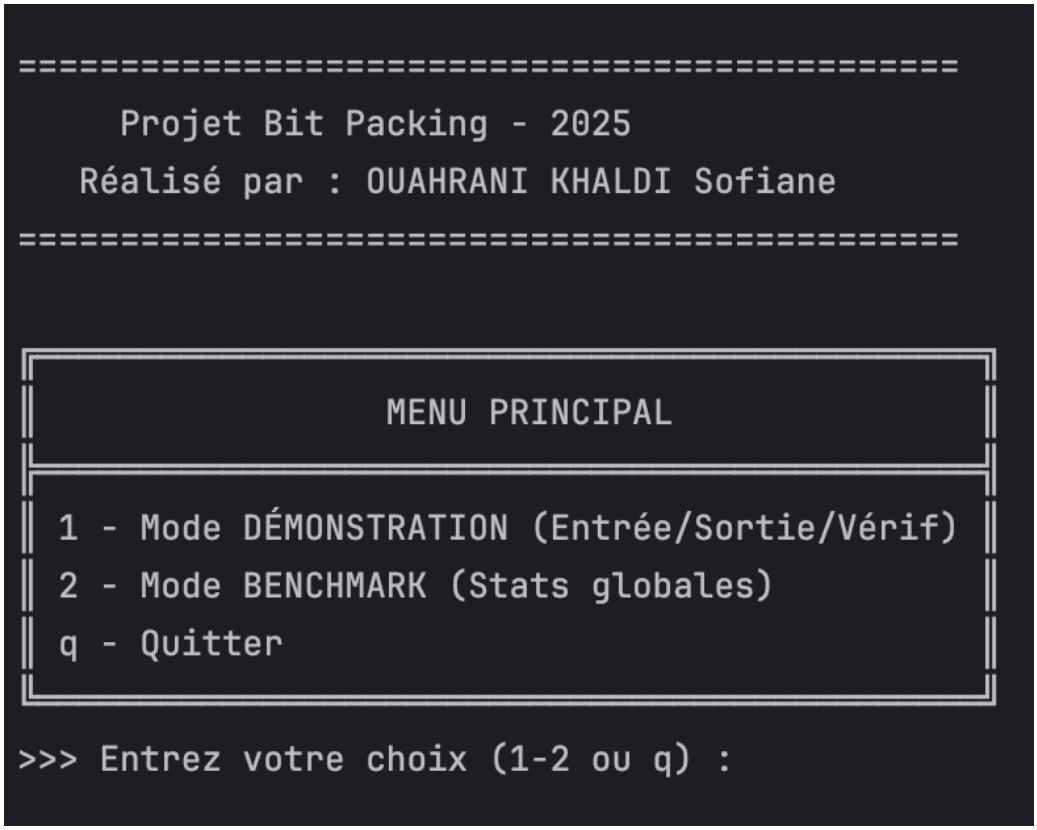
2.3. Le Lanceur : Main.java

Cette classe sert d'interface utilisateur en console et de lanceur pour le protocole de test.

J’ai fait en sorte qu’elle soit interactive et facile à comprendre.

- **Mode DÉMONSTRATION :** Permet à l'utilisateur de fournir un tableau personnalisé, de choisir l'un des trois algorithmes, et d'exécuter un cycle complet. Il affiche les métriques (taille, taux, temps CPU) et le verdict de rentabilité.
- **Mode BENCHMARK :** Exécute un protocole de test automatisé sur les trois algorithmes. Il utilise plusieurs tailles de tableaux (de 10 à 1 000 000) et trois distributions de données (petits nombres, mixtes, grands nombres) pour générer une analyse comparative complète.
- **Calcul de Rentabilité :** Implémente la logique de calcul du seuil de rentabilité en comparant le Coût CPU Total (*Tc + Td)* au Gain Temps Réseau (*T_std - T_comp*), basé sur un débit réseau simulé (*NETWORK_SPEED_MBPS*).

Voici comment l'interface ce présente et un exemple de choix du mode d'exécution :



Mode DEMONSTRATION sélectionné :

```
╔═══════════════════════════════════════════╗
║                MENU PRINCIPAL             ║
╠═══════════════════════════════════════════╣
║ 1 - Mode DÉMONSTRATION (Entrée/Sortie/Vérif)
║ 2 - Mode BENCHMARK (Stats globales)       ║
║ q - Quitter                               ║
╚═══════════════════════════════════════════╝
>>> Entrez votre choix (1-2 ou q) : 1

--- Mode DÉMONSTRATION ---

[1] Choisissez le mode de compression :
 1 - Aligned (V1 - Sans chevauchement)
 2 - Overlap (V2 - Avec chevauchement)
 3 - Overflow (V3)
>>> Entrez votre choix (1-3) : 1

--- Entrée personnalisée ---
Entrez les entiers séparés par des espaces (ex: 10 5000 3 1): 1 2 3 4

Mode sélectionné : ALIGNED
```

Mode BENCHMARK sélectionné :

```
╔═══════════════════════════════════════════╗
║                MENU PRINCIPAL             ║
╠═══════════════════════════════════════════╣
║ 1 - Mode DÉMONSTRATION (Entrée/Sortie/Vérif)
║ 2 - Mode BENCHMARK (Stats globales)       ║
║ q - Quitter                               ║
╚═══════════════════════════════════════════╝
>>> Entrez votre choix (1-2 ou q) : 2

--- Mode BENCHMARK (Aléatoire & Interactif) ---
Exécution de tests complets sur 5 tailles de tableaux : [10, 100, 1000, 50000, 1000000]

=== TEST #1 | Taille : 10 éléments | Petits nombres uniquement | MaxVal : 255 ===
=> Taille originale : 40 octets
=> Temps de transmission standard : 32 000 ns

============================================
RÉSULTATS DIAGNOSTIC : ALIGNED
============================================
   K : 8 bits (Max Val: 255)
   Taille Org. : 40 octets (320 bits)
   Taille Comp.: 12 octets (96 bits)
   Taux de Compression : 70,00%
```

3. STRATEGIES DE RESOLUTION

J'ai implémenté les trois stratégies de compression distinctes, chacune héritant de *BitPacking*, suivantes :

3.1. V1 : BitPackingAligned (Sans Chevauchement)

C'est l'implémentation la plus simple, utilisant la convention LSB-first (de droite à gauche).

- **Logique compress() :**
    1. Calcule *ITEMS_PER_CONTAINER = 32 / K*.
    2. Alloue le tableau de sortie (*ceil(originalSize / ITEMS_PER_CONTAINER)*).
    3. La logique de bourrage (est gérée par la condition *if (bitOffset + K > INT_BITS),* qui force le passage au conteneur suivant (*outputIndex++)* et réinitialise *bitOffset = 0*.
    4. J'utilise un décalage vers la gauche et un OU binaire pour l'écriture : *compressedArray[outputIndex] |= (inputVal << bitOffset);*
- **Logique get(i) :**
    1. Calcule l'adresse : *indexContainer = i / itemsPerContainer*; et *bitOffset = (i % itemsPerContainer) * K;*.
    2. L'extraction se fait par un décalage à droite et un masquage pour ne garder que le résultat voulu : *(compressedArray[indexContainer] >>> bitOffset) & MASK_K_BITS;*.
- **Optimisation :** J'ai rajouté cette petite optimisation qui me créait un coût CPU inutile : Si K > 16 bits (K > INT_BITS / 2), ITEMS_PER_CONTAINER vaut 1. Le taux de compression devient ainsi 0%. J'ai donc ajouté un "chemin rapide" qui retourne input directement si K > 16, évitant ce coût CPU inutile.

3.2. V2 : BitPackingOverlap (Avec Chevauchement)

Cette version maximise la densité en permettant aux paquets d'être coupés sur deux conteneurs. Elle utilise également la convention LSB-first.

- **Logique compress() :**
    1. Optimisation : J'ai géré le cas K = 32, pour lequel le chevauchement est inutile. La méthode retourne ainsi input directement.
    2. La taille du tableau est calculée sur le nombre total de bits *(ceil(originalSize * K / 32))*.
    3. **Cas 1 (Pas de chevauchement)** : if (bitOffset + K <= INT_BITS) donc insertion simple LSB-first (value << bitOffset) avec un décalage vers la gauche encore.
    4. **Cas 2 (Chevauchement)** : Le paquet est coupé en 2 parties : bits1 (poids faible) et bits2 (poids fort).
        - *partie1 = value & masque(bits1)* est insérée à bitOffset.
        - *partie2 = value >>> bits1* (le reste) est insérée à l'offset 0 du conteneur suivant.
        - Le nouvel bitOffset est mis à jour à bits2 pour bien insérer les prochains entiers.

- **Logique get(i) :**
    1. Calcule la position absolue : bitPosition = i * K.
    2. Calcule l'adresse : *indexContainer = bitPosition / 32*; et *bitOffset = bitPosition % 32*;.
    3. **Cas 1 (Pas de chevauchement)** : *if (bitOffset + K <= INT_BITS)*. Extraction simple (identique à Aligned).
    4. **Cas 2 (Chevauchement)** :
        - Lit partie1 (poids faible) à la fin de container[j].
        - Lit partie2 (poids fort) au début de container[j+1].
        - Recombine en suivant bien le décalage : result = partie1 | (partie2 << bits1);
- **Logique decompress() :** J'ai choisi pour la lisibilité et la robustesse, d'appeler get(i) en boucle, bien que ce soit moins performant qu'une lecture séquentielle .

3.3. V3 : BitPackingOverflow

C'est l'implémentation la plus complexe, conçue pour les tableaux hétérogènes (petits + grands entiers). Elle utilise la logique LSB-first et Overlap.

- **Structure de Sortie (3 Zones) :**
    1. **Header (2 INTs)** : Stocke les métadonnées (K', K_overflow, K_main, K_ref, originalSize, overflow_start_index).
    2. **Main Data Area** : Stocke les paquets de K_main bits (= Flag + Payload).
    3. **Overflow Area** : Stocke les valeurs brutes des grands nombres. J'ai choisi de la séparer de la Main Data Area en la faisant commencer au conteneur suivant le dernier utilisé par la Main Data Area , ce qui peut créer un bourrage inutile mais je tenais a séparer ces deux zones.
- **Logique compress() (que j'ai réalisé en 6 Étapes majeures) :**
    1. **Analyse (*analyzeAndSetK*)** : Fonction statique qui teste tous les K' possibles (1 à 32) et choisit celui qui minimise le coût total en bits (Main + Overflow).
    2. **Séparation** : J'ai choisi de créer trois listes (petits nombres, grands nombres, flags) pour maintenir l'ordre original en m'y retrouvant.
    3. **Calcul des K finaux** : Calcule K_ref (basé sur le nombre d'overflows) et K_main (basé sur 1 + max(K', K_ref)) pour gérer le cas où les index ont besoin de plus de bits pour être représenté que les petits nombres.
    4. **Allocation** : Calcule la taille totale requise pour les 3 zones.
    5. **Écriture** : Écrit d'abord la Zone Overflow , puis réinitialise les pointeurs de suivi (state) et écrit la Zone Principale (paquets Flag+Payload).
    6. **Écriture Header** : Écrit les métadonnées dans compressedArray[0] et [1].
- **Logique get(i) (Navigation dans les 3 Zones) :**
    1. **Lecture Header** : Appelle *readHeader*() pour initialiser l'état (K', K_main, etc.).
    2. **Lecture Main Data** : Calcule l'adresse du i-ème paquet (*HEADER_SIZE + i * K_main)* et utilise *readOverlap* pour extraire le paquet mainPayloadWithFlag.
    3. **Analyse Flag** : Isole le Flag.
    4. **Retour** : Si Flag=0, retourne le Payload. Si Flag=1, utilise le Payload comme index pour lire la valeur brute dans l'Overflow Area (grâce à un second appel à readOverlap).

4. DEFIS DE CONCEPTION ET PROBLEMES RENCONTRES

Durant ce projet, J'ai rencontré plusieurs défis qui ont nécessité des choix de conception spécifiques :

- **PB 1 : Choix du sens écriture/lecture des bits :**
  - **Problème :** Lors de la conception initiale de mon projet, j'écrivais/lisais mes bits de gauche à droite .Ce qui compliquais mes logiques d'écriture/lecture. Par exemple pour me situer je commençais à *32 – K_ offset* au lieu de simplement *Offset*. Et d'autres erreurs lors de la recherches d'un entier.
  - **Solution :** Après m'être rendu compte à quel point cela était incensé/compliqué pour rien, j'ai adopté la convention (plus naturelle ducoup) LSB-First, c'est-à-dire « de droite à gauche ».
- **PB 2 : Indépendance des Données :**
  - **Problème :** Dans *BitPackingOverflow* les méthodes get() et decompress() ne peuvent pas fonctionner si elles sont appelées par une nouvelle instance de la classe, car l'état (this.k, this.k_main, etc.) n'est pas initialisé (il l'est uniquement dans compress()).
  - **Solution :** J'ai donc rendu les données auto-suffisantes. Le Header est écrit par compress(). Les méthodes get() et decompress() appellent obligatoirement readHeader(compressedArray) au début pour s'auto-configurer à partir des données reçues.
- **PB 3 : Gestion des Pointeurs Multiples (Overflow)**
  - **Problème :** L'écriture des Zones Principale et Overflow nécessitait deux jeux de pointeurs (index et bitOffset). L'utilisation de variables d'instance (globales) aurait créé des problèmes de concurrence (thread-safety) et des effets de bord.
  - **Solution :** J'utilise donc un tableau d'état local (int[] state = {index, offset}) passé par référence aux méthodes utilitaires (writeOverlap, readOverlap). Chaque zone gère son propre curseur de manière isolée.
- **PB 4 : Uniformité du Payload (Overflow)**
  - **Problème :** Le Payload de la Main Data doit être uniforme, mais K' (taille valeur) et K_ref (taille index) peuvent différer.
  - **Solution :** J'ai donc fixé la taille du Payload à max(K', K_ref). Le paquet le plus petit (valeur ou index) reçoit des zéros de padding pour atteindre la taille uniforme, garantissant un adressage i * K_main fiable.
- **PB 5 : Optimisation de decompress() (:**
  - **Problème :** Le problème que j'ai remarqué étaitt que l'appel en boucle de get(i) dans decompress() faisait « beau » certes mais était très lent, car readHeader() était appelé N fois.
  - **Solution :** J'ai donc surchargé la méthode get(i, boolean headerDefined). La boucle decompress() appelle readHeader() une seule fois, puis appelle get(i, true) pour sauter les lectures inutiles du Header.

- **PB 6 : Bourrage Inter-Zone (Compromis de Design)**
    - **Problème :** L'adressage statique de l'Overflow Area (HEADER_SIZE_INTS + mainDataSizeInts) crée un bourrage si la Main Data n'utilise pas entièrement son dernier conteneur.
    - **Solution (Choix) :** J'avoue j'ai conservé ce bourrage. Pour l'éliminer aurait je pourrait stocker un bitOffset de départ dans le Header, ce qui peux un peu complexifier l'adressage. J'ai préféré la simplicité et la robustesse de l'adressage statique.

5. COMPARAISON DES METHODES (Via l'analyse des Benchmarks)

J'ai procédé à l'exécution du Mode Benchmark (*Main*.java) sur un tas de scénarios (variant la taille de 10 à 1 000 000 et la distribution des K) ce qui a révélé les tendances suivantes :

Tableau 1 : Taux de Compression Moyen (Gain de Place)

Distribution	Aligned (V1) Taux %	Overlap (V2) Taux %	Overflow (V3) Taux %
Petits Nombres (K<17)	**75.00%**	**75.00%**	71.87%
Mixtes	0.00%	46.00%	**66.00%**
Grands Nombres (K>=17)	0.00%	**46.88%**	43.75%

Analyse Taux :

1. **V1 (Aligned)** est parfait si K divise 32, mais s'effondre à 0% de compression dès que K > 16.

   Par exemple :

```
=== TEST #1 | Taille : 10 éléments | Petits nombres uniquement | MaxVal : 255 ===
=> Taille originale : 40 octets
=> Temps de transmission standard : 32 000 ns

============================================
RÉSULTATS DIAGNOSTIC : ALIGNED
============================================
   K : 8 bits (Max Val: 255)
   Taille Org. : 40 octets (320 bits)
   Taille Comp.: 12 octets (96 bits)
   Taux de Compression : 70,00%

--- Coût CPU et Rentabilité ---
   Coût CPU Total : 22 125 ns
   Gain Temps Réseau : 22 400 ns
   T_Get (ns) : 875

✅ RENTABLE : Le gain de temps réseau dépasse le coût du CPU.
```

```
=== TEST #2 | Taille : 10 éléments | Petits + Grands nombres | MaxVal : 75 653 ===
=> Taille originale : 40 octets
=> Temps de transmission standard : 32 000 ns

============================================
RÉSULTATS DIAGNOSTIC : ALIGNED
============================================
   K : 17 bits (Max Val: 75 653)
   Taille Org. : 40 octets (320 bits)
   Taille Comp.: 40 octets (320 bits)
   Taux de Compression : 0,00%

--- Coût CPU et Rentabilité ---
   Coût CPU Total : 6 792 ns
   Gain Temps Réseau : 0 ns
   T_Get (ns) : 250

❌ NON RENTABLE : Le coût du CPU est trop élevé pour ce gain.
```

2. **V2 (Overlap)** maintient un taux de 47% même si K > 16. C'est le meilleur choix pour les données homogènes de grande taille.

   Exemple sur un tableau d'1 millions d'éléments :

```
==========================================
RÉSULTATS DIAGNOSTIC : OVERLAP
==========================================
   K : 17 bits (Max Val: 131 070)
   Taille Org. : 4 000 000 octets (32 000 000 bits)
   Taille Comp.: 2 125 000 octets (17 000 000 bits)
   Taux de Compression : 46,88%

--- Coût CPU et Rentabilité ---
   Coût CPU Total : 11 446 917 ns
   Gain Temps Réseau : 1 500 000 000 ns
   T_Get (ns) : 2 917

✅ RENTABLE : Le gain de temps réseau dépasse le coût du CPU.
```

3. **V3 (Overflow)** quant à elle, excelle sur les données mixtes (66-69%), car analyzeAndSetK trouve un K' optimal, battant V2.

Exemple : Pour une exécution sur un très grand tableau (1M) d'entiers mixtes on a par exemple un gain réseau 15 fois grand que le coût CPU !!!

```
==========================================
RÉSULTATS DIAGNOSTIC : OVERFLOW
==========================================
   K : 17 bits (Max Val: 131 065)
   Taille Org. : 4 000 000 octets (32 000 000 bits)
   Taille Comp.: 2 177 108 octets (17 416 864 bits)
   Taux de Compression : 45,57%
   [i] K' (pour les petits nombres) : 16 bits

--- Coût CPU et Rentabilité ---
   Coût CPU Total : 94 831 708 ns
   Gain Temps Réseau : 1 458 313 600 ns
   T_Get (ns) : 2 709

✅ RENTABLE : Le gain de temps réseau dépasse le coût du CPU.
```

Tableau 2 : Coût CPU Total Moyen (Vitesse)

Taille N	Aligned (V1) T_CPU (ns)	Overlap (V2) T_CPU (ns)	Overflow (V3) T_CPU (ns)
10	**10 K**	26 K	63 K
100	**16 K**	71 K	244 K
1 000	**85 K**	208 K	836 K
50 000	**1.2 M**	2.7 M	14.6 M
1 000 000	**13.4 M**	6.6 M	69.7 M

Analyse de la Vitesse :

1. **V1 (Aligned)** est le plus rapide sur les petits K.
2. **V3 (Overflow)** est massivement plus lent (10x à 20x) que V1/V2, à cause du coût de l'analyse analyzeAndSetK.

Ce que je recommende (Choix de l'Algorithme) :

- **Cas 1 : Petits** K **(K <= 16)**
  - **Choix : V1 (Aligned)**.
  - Il offre le même Taux de Compression que V2 (75% pour K=8 par exemple) mais avec un Coût CPU 40% inférieur. Il est le plus rentable.
- **Cas 2** : **Grands K (K > 16) avec des données Homogènes**
  - **Choix : V2 (Overlap)**.
  - La V1 Aligned échoue (Taux 0%). V2 Overlap maintient un taux de 47% pour un coût CPU très bas (6.6M ns), surpassant V3 (qui est 10x plus lent).
- **Cas 3 : Données Mixtes**
  - **Choix : V3 (Overflow)**.
  - C'est le seul algorithme conçu pour ce cas. Sur les données mixtes, V3 atteint en moyenne 69% de compression, là où V2 est bloqué à 46%. Le coût CPU élevé est justifié par le gain de place nettement supérieur.

PS : EN GENERAL Il faut noter que sur de très petits tableaux, bien qu'on ait une bonne compression on peut se retrouver avec une non rentabilité puisque la transmission des données étant très rapide sur ce genre de tableaux, le coût CPU dépasse le gain de temps de transmission obtenu par la compression.

Voici un exemple illustratif :

Bien qu'on ait un taux de compression de **70%** on observe que le coût CPU est légèrement supérieur au gain de temps réseau.

```
=== TEST #1 | Taille : 10 éléments | Petits nombres uniquement | MaxVal : 255 ===
=> Taille originale : 40 octets
=> Temps de transmission standard : 32 000 ns

==========================================
RÉSULTATS DIAGNOSTIC : ALIGNED
==========================================
   K : 8 bits (Max Val: 255)
   Taille Org. : 40 octets (320 bits)
   Taille Comp.: 12 octets (96 bits)
   Taux de Compression : 70,00%

--- Coût CPU et Rentabilité ---
   Coût CPU Total : 23 667 ns
   Gain Temps Réseau : 22 400 ns
   T_Get (ns) : 1 667

❌ NON RENTABLE : Le coût du CPU est trop élevé pour ce gain.
```

Pour un tableau de 10 éléments mixtes par exemple, en utilisant overflow on a bien un taux de compression de 40% mais le coût CPU est bien trop élevé ce qui montre bien que la rentabilité pour des méthodes coûteuse se fait à partir d'un certain nombre d'éléments / ou si on a un tableau de minuscules valeurs et un/plusieurs nombre représentable sur un k « démesuré » :

```
==========================================
RÉSULTATS DIAGNOSTIC : OVERFLOW
==========================================
   K : 9 bits (Max Val: 94 777)
   Taille Org. : 40 octets (320 bits)
   Taille Comp.: 24 octets (192 bits)
   Taux de Compression : 40,00%
   [i] K' (pour les petits nombres) : 8 bits

--- Coût CPU et Rentabilité ---
   Coût CPU Total : 169 459 ns
   Gain Temps Réseau : 12 800 ns
   T_Get (ns) : 4 500

❌ NON RENTABLE : Le coût du CPU est trop élevé pour ce gain.
```

Voici également un tableau comparatif moyen que j'ai réalisé en fonction de la taille des tableaux :

Taille N	**Distribution**	**Aligned (V1) Taux de compression%**	**Overlap (V2) Taux de compression%**	**Overflow (V3) Taux de compression %**
10	Petits Nombres	**70.00%**	**70.00%**	50.00%
10	Mixtes	0.00%	**40.00%**	**40.00%**
10	Grands Nombres	0.00%	**40.00%**	20.00%
100	Petits Nombres	**75.00%**	**75.00%**	69.00%
100	Mixtes	0.00%	46.00%	**66.00%**
100	Grands Nombres	0.00%	**46.00%**	41.00%
1 000	Petits Nombres	**75.00%**	**75.00%**	71.60%
1 000	Mixtes	0.00%	46.80%	**69.10%**
1 000	Grands Nombres	0.00%	**46.80%**	43.50%
50 000	Petits Nombres	**75.00%**	**75.00%**	71.87%
50 000	Mixtes	0.00%	46.87%	**56.86%**

Taille N	Distribution	Aligned (V1) Taux de compression%	Overlap (V2) Taux de compression%	Overflow (V3) Taux de compression %
50 000	Grands Nombres	0.00%	**46.87%**	43.75%
1 000 000	Petits Nombres	**75.00%**	**75.00%**	71.87%
1 000 000	Mixtes	0.00%	**46.88%**	45.58%
1 000 000	Grands Nombres	0.00%	**46.88%**	43.75%

6. CONCLUSION

Ce projet m'a permis de valider l'efficacité de trois architectures de Bit Packing.
L'analyse des benchmarks démontre qu'il n'existe pas de "meilleure" solution universelle ; le choix dépend entièrement de la distribution des données d'entrée.

La V1 (Aligned) est la plus rapide pour les données simples.

La V2 (Overlap) offre la meilleure densité pour les données homogènes complexes (K > 16).

Enfin, la V3 (Overflow) est la solution la plus robuste pour les données hétérogènes (outliers), bien que son coût CPU soit significativement plus élevé en raison de l'analyse heuristique des coûts.

Le projet a également souligné l'importance des choix de conception pour garantir la robustesse, l'efficacité et la performance des algorithmes.

Je vous remercie d'avoir pris le temps de lire ce rapport et j'espère que mes efforts et le travail réalisé tout au long de ce projet auront su en refléter toute la rigueur et l'implication.

7. Références

Voici plusieurs supports qui m'ont servi lors de la réalisation du ce projet :

- **Calcul de K (Integer.numberOfLeadingZeros) :**
  - https://www.geeksforgeeks.org/java/integer-numberofleadingzeros-method-in-java-with-example/
- **Conception des Classes Génériques (ResultWithTime) :**
  - https://www.jmdoudoux.fr/java/dej/chap-generique.htm
- **Opérations binaires en Java :**
  - https://www.geeksforgeeks.org/java/bitwise-operators-in-java/
- **Calcul du temps :**
  - https://koor.fr/Java/API/java/lang/System/nanoTime.wp
- **Convention bits :**
  - https://fr.wikipedia.org/wiki/Num%C3%A9rotation_des_bits#:~:text=Le%20LSB%20est%20parfois%20appel%C3%A9,bit%20le%20plus%20%C3%A0%20gauche%20.
- **La liste des exceptions (Java) :**
  - https://www.w3schools.com/java/java_ref_errors.asp