

Compression d'entiers par Bit Packing

Rapport de projet

Romain STEFANI

M1 Master Informatique
Université Côte d'Azur

Année universitaire 2025-2026

Software Engineering

2 novembre 2025

Table des matières

1	Introduction	3
1.1	Contexte du projet	3
1.2	Objectif général	3
1.3	Contraintes et organisation	3
2	Pourquoi compresser des entiers ?	3
3	Méthodes de compression implémentées	4
3.1	Le principe du Bit Packing	4
3.2	BASIC et NO_OVERLAP	5
3.2.1	Principe commun	5
3.2.2	La différence : gestion du chevauchement	6
3.2.3	Accès direct avec get()	6
3.2.4	Décompression	7
3.3	OVERFLOW	7
3.3.1	Problématique des outliers	7
3.3.2	Principe de la zone de débordement (Overflow)	7
3.3.3	Choix du nombre de bits optimal	8
3.3.4	Accès direct et décompression	8
3.3.5	Cas d'usage	8
4	Architecture logicielle	8
4.1	Structure générale du code	8
4.2	Le pattern Factory	9
4.3	L'interface BitPacking	9
4.4	Gestion des erreurs	9
5	Processus de développement	10
5.1	BASIC – Développement de la méthode de base	10
5.1.1	Phase de réflexion initiale	10
5.1.2	Choix du sens de remplissage	10
5.1.3	Première version fonctionnelle	11

5.1.4	Phase d'optimisation	11
5.2	NO_OVERLAP – Adaptation de la méthode de base	12
5.3	OVERFLOW – Développement de la méthode sophistiquée	12
5.3.1	Réflexions initiales et prototypes	12
5.3.2	Solution finale : calcul du coût total	13
5.3.3	Développement de compress(), get() et decompress()	13
6	Benchmark et analyse de performance	14
6.1	Protocole de mesure	14
6.1.1	Phase de warm-up	14
6.1.2	Phase de mesure	14
6.1.3	Distributions testées	14
6.1.4	Tailles testées	15
6.2	Résultats et comparaisons	15
6.2.1	Cas uniforme (0-100) – 10 000 éléments	15
6.2.2	Cas avec outliers (2% grandes valeurs) – 10 000 éléments	16
6.2.3	Cas grandes valeurs (0-100 000) – 10 000 éléments	16
6.2.4	Cas d'usage recommandé pour chaque méthode	16
6.3	Impact des outliers	17
6.4	Analyse de la latence de rentabilité	17
6.4.1	Calcul de la latence de rentabilité	17
6.4.2	Résultats pour le cas uniforme (10 000 éléments, 0-100)	18
6.4.3	Résultats pour le cas avec outliers (10 000 éléments, 2% grandes valeurs)	18
6.4.4	Interprétation pour les réseaux réels	18
6.4.5	Choix de la méthode selon le réseau	18
7	Conclusion	19

1 Introduction

1.1 Contexte du projet

J'ai réalisé ce projet dans le cadre du cours de Software Engineering du Master 1 Informatique à l'Université Côte d'Azur. L'objectif était d'étudier et d'implémenter des méthodes de compression d'entiers par Bit Packing pour optimiser la transmission de données sur les réseaux.

Le problème de base est simple : quand on transmet des tableaux d'entiers sur un réseau, Java utilise systématiquement 32 bits par entier, même si la valeur stockée n'a besoin que de quelques bits. Cette approche gaspille beaucoup de bande passante quand on transmet des données.

1.2 Objectif général

L'objectif de ce projet était de développer trois méthodes différentes de Bit Packing dont le fonctionnement sera expliqué dans la suite du rapport. La contrainte principale était de préserver un accès direct aux éléments compressés : contrairement aux méthodes de compression classiques qui nécessitent de décompresser tout le tableau pour accéder à un élément, mes algorithmes devaient permettre d'accéder au i -ème élément sans avoir à décompresser l'ensemble.

Au-delà de l'implémentation, je devais également mettre en place un benchmark pour mesurer les performances de chaque méthode. L'objectif était de déterminer dans quels contextes chaque méthode est la plus efficace et de calculer les seuils de rentabilité réseau au-delà desquels la compression devient avantageuse.

1.3 Contraintes et organisation

Le projet a été développé en Java et hébergé sur GitHub. L'architecture devait respecter les bonnes pratiques du génie logiciel, notamment en utilisant un pattern Factory pour instancier les trois méthodes de compression (BASIC, NO_OVERLAP et OVERFLOW) via une interface commune `BitPacking`.

Cette interface définit trois méthodes essentielles : `compress(int[] array)` pour compresser un tableau d'entiers, `decompress(int[] compressedArray)` pour récupérer le tableau original, et `get(int[] compressedArray, int i)` pour accéder directement à un élément sans décompression complète.

2 Pourquoi compresser des entiers ?

Quand on transmet des données sur un réseau, il y a toujours un compromis entre le temps de traitement local (compression/décompression) et le temps de transmission. Avec

la représentation standard en Java, chaque entier consomme 32 bits, même si la valeur stockée est petite.

Prenons un exemple concret : un tableau de 10 000 entiers avec des valeurs entre 0 et 100. En représentation standard, ce tableau nécessite 320 000 bits de transmission. Mais pour représenter des valeurs jusqu'à 100, on n'a besoin que de 7 bits ($2^7 = 128$). Avec du Bit Packing, le même tableau pourrait ne consommer que 70 000 bits (plus quelques bits de métadonnées), soit une réduction de plus de 78% du volume.

Par contre, la compression et la décompression prennent du temps. La question devient donc : à partir de quelle latence réseau la compression devient-elle rentable ? C'est une des questions auxquelles le benchmark devra répondre.

3 Méthodes de compression implémentées

3.1 Le principe du Bit Packing

Le Bit Packing repose sur une idée simple : au lieu d'utiliser systématiquement 32 bits pour chaque entier, on calcule le nombre de bits réellement nécessaires pour représenter les valeurs du tableau, puis on stocke les entiers en utilisant uniquement ces bits.

Le principe de la méthode se décompose en plusieurs étapes :

1. **Calcul du nombre de bits** : En fonction de la méthode et des éléments du tableau, on veut calculer le nombre de bits qu'on va utiliser pour stocker nos entiers. Java fournit la méthode `Integer.numberOfLeadingZeros()` qui compte le nombre de zéros en tête de la représentation binaire. On peut ainsi calculer précisément pour un entier :

$$\text{bits nécessaires} = 32 - \text{numberOfLeadingZeros}(\text{value}) \quad (1)$$

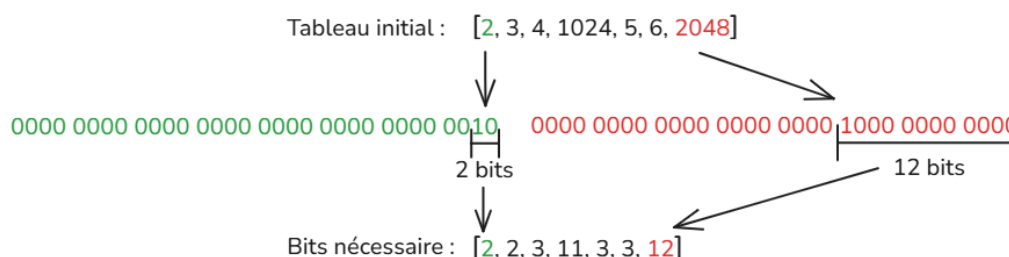


FIGURE 1 – Calcul du nombre de bits nécessaires

2. **Stockage compact** : Au lieu d'allouer 32 bits par élément, on utilise uniquement le nombre de bits calculé. Les éléments sont stockés les uns à la suite des autres dans un tableau d'entiers 32 bits, en « tassant » les valeurs.
3. **Position calculable** : Pour pouvoir accéder directement au i -ème élément, chaque valeur est stockée à une position calculable :

$$\text{position bits} = i \times \text{bits nécessaires} \quad (2)$$

Cela permet de retrouver n'importe quel élément sans avoir à tout décompresser.

4. **Métadonnées** : Le premier entier du tableau compressé contient les métadonnées : la taille du tableau original et le nombre de bits utilisés par élément. Ces informations sont indispensables pour la décompression et l'accès direct.

Les trois méthodes utilisent le même format pour stocker les métadonnées dans le premier entier du tableau compressé. J'ai optimisé cette structure pour utiliser efficacement les 32 bits disponibles :

- 26 bits pour la taille du tableau (bits 6 à 31) : permet de stocker jusqu'à 67 millions d'éléments
- 6 bits pour le nombre de bits nécessaires (bits 0 à 5) : permet de stocker des valeurs jusqu'à 63 bits, donc suffisant car notre maximum est à 32.

Cette répartition se fait avec l'opération :

```
1 header[0] = (arraySize << 6) | (bitsNeeded & 0x3F)
```

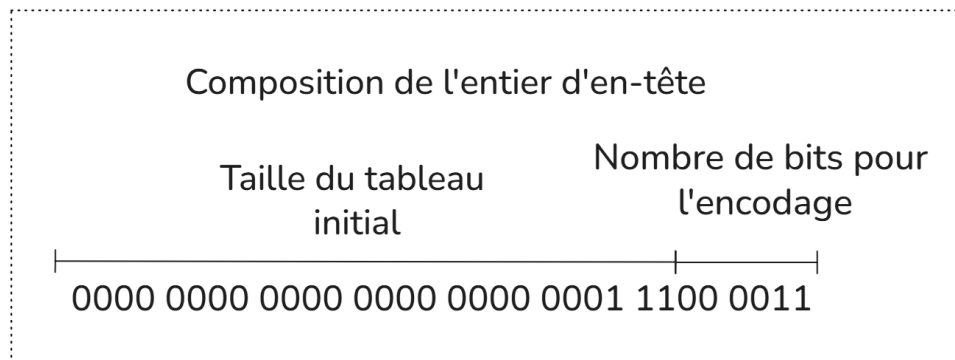


FIGURE 2 – Représentation du header avec répartition des bits

3.2 BASIC et NO_OVERLAP

Ces deux méthodes partagent le même principe de base et se différencient uniquement dans la gestion du chevauchement des éléments sur les entiers du tableau compressé.

3.2.1 Principe commun

Le fonctionnement de ces deux méthodes commence de la même manière. On parcourt d'abord tout le tableau pour identifier la valeur maximale. Cette valeur est celle qui nécessite le plus grand nombre de bits pour être représentée. Une fois cette valeur trouvée, on calcule le nombre de bits nécessaires.

Par exemple, si la valeur maximale du tableau est 100, on a besoin de 7 bits car $2^7 = 128$. Tous les entiers du tableau original seront donc représentés sur 7 bits dans le tableau compressé, même s'ils nécessitent moins de bits individuellement.

Les éléments sont ensuite stockés les uns à la suite des autres dans le tableau compressé. Chaque élément occupe exactement le nombre de bits calculé. On peut calculer la position

de chaque élément avec la formule :

$$\text{position} = i \times \text{bits_nécessaires} \quad (3)$$

3.2.2 La différence : gestion du chevauchement

Le problème survient quand un élément compressé ne rentre pas complètement dans la fin de l'entier 32 bits actuel. Imaginons qu'on utilise 12 bits par élément. Le premier élément occupera les bits 0 à 11 du premier entier, le deuxième les bits 12 à 23, mais le troisième devrait occuper les bits 24 à 35. Or, un entier ne fait que 32 bits.

C'est ici que les deux méthodes diffèrent :

Méthode BASIC : On autorise le chevauchement. Le troisième élément sera réparti sur deux entiers : les bits 24 à 31 du premier entier et les bits 0 à 3 du second entier. Cette approche maximise l'utilisation de l'espace mais complique légèrement l'extraction.

Méthode NO_OVERLAP : On refuse le chevauchement. Quand on détecte qu'un élément ne peut pas tenir complètement dans l'entier actuel, on passe directement à l'entier suivant en laissant des bits inutilisés. Dans notre exemple, le troisième élément serait stocké entièrement dans le second entier (bits 0 à 11), laissant les bits 24 à 31 du premier entier vides.

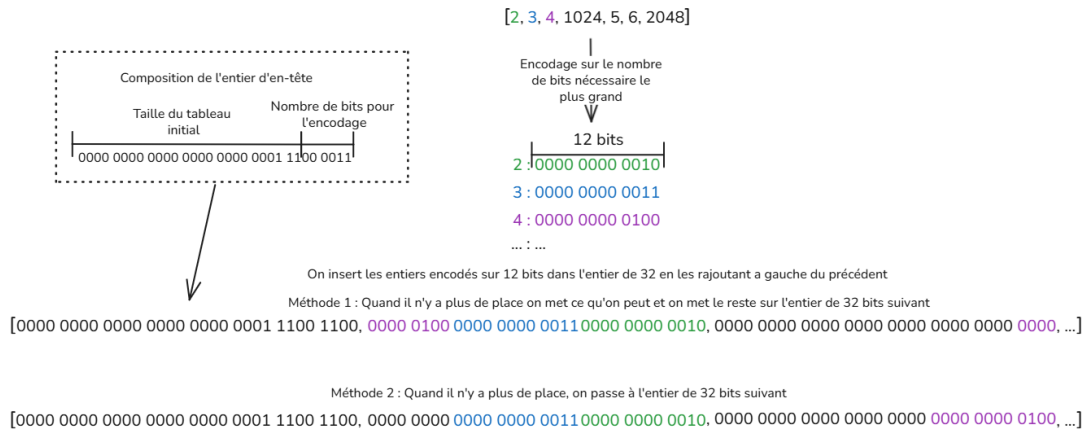


FIGURE 3 – Comparaison des méthodes BASIC et NO_OVERLAP

3.2.3 Accès direct avec `get()`

La méthode `get(int[] compressedArray, int i)` permet d'accéder directement au i -ème élément sans décompresser tout le tableau. Elle calcule d'abord la position en bits de l'élément recherché, puis identifie dans quel entier il se trouve.

Pour la méthode BASIC, il faut gérer deux cas : soit l'élément tient dans un seul entier et on l'extrait, soit il chevauche deux entiers et il faut combiner les bits des deux entiers.

Pour NO_OVERLAP, c'est plus simple : on sait que l'élément est forcément dans un seul entier, donc il nous suffit de l'extraire directement.

3.2.4 Décompression

La méthode `decompress()` reconstruit le tableau original en appelant simplement `get()` pour chaque index de 0 à la taille du tableau. Elle exploite donc directement le mécanisme d'accès direct.

3.3 OVERFLOW

La méthode OVERFLOW est plus sophistiquée. Elle a été conçue pour gérer efficacement les tableaux qui contiennent des valeurs qui sortent du lot (outliers).

3.3.1 Problématique des outliers

Imaginons un tableau de 10 000 entiers où 9 800 valeurs sont entre 0 et 100 (nécessitant 7 bits) mais 200 valeurs sont autour de 1 000 000 (nécessitant 20 bits). Avec BASIC ou NO_OVERLAP, on serait obligé d'utiliser 20 bits pour tous les 10 000 éléments, ce qui gaspillerait énormément d'espace sur les 9 800 petites valeurs.

3.3.2 Principe de la zone de débordement (Overflow)

L'idée est de séparer le tableau compressé en deux zones. La première zone, appelée zone principale, stocke la majorité des valeurs avec un nombre réduit de bits. La seconde zone, appelée zone de débordement (overflow), stocke les quelques outliers en 32 bits.

Pour distinguer les deux cas, on ajoute un bit de flag à chaque élément de la zone principale. Si le flag vaut 0, la valeur est stockée directement. Si le flag vaut 1, la valeur est en réalité un index qui pointe vers un élément de la zone overflow.

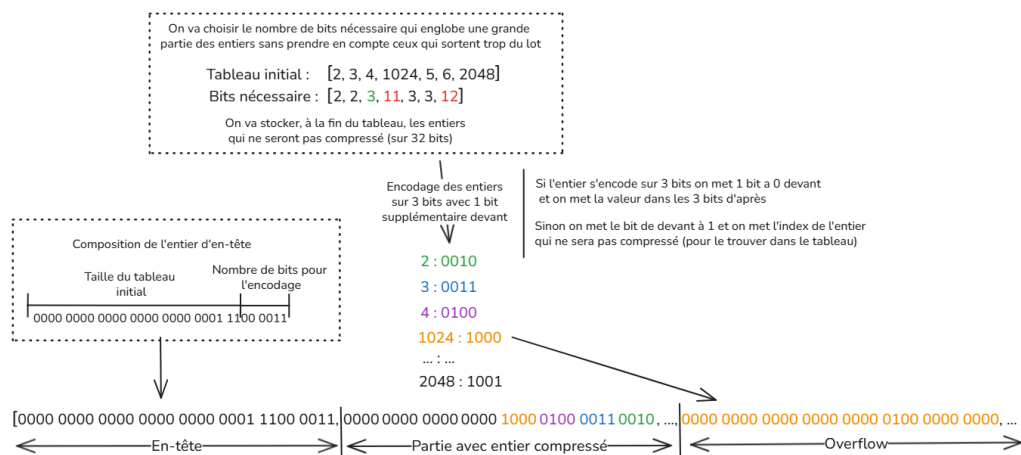


FIGURE 4 – Structure de la méthode OVERFLOW avec zone de débordement

3.3.3 Choix du nombre de bits optimal

La principale problématique de cette méthode est : combien de bits faut-il utiliser pour la zone principale ? Si on choisit trop peu de bits, beaucoup d'éléments iront en overflow. Si on en choisit trop, on gaspille de l'espace.

J'ai implémenté un algorithme qui teste différentes valeurs et calcule pour chacune le coût total en bits. Le coût prend en compte à la fois l'espace utilisé par la zone principale (chaque élément prend le nombre de bits choisi plus 1 bit de flag) et l'espace utilisé par la zone overflow (32 bits par élément en overflow).

L'algorithme sélectionne le nombre de bits qui minimise ce coût total. Il vérifie également qu'on peut représenter l'index de débordement avec le nombre de bits choisi : si on a 300 éléments en overflow, il faut au moins 9 bits pour représenter les index de 0 à 299.

3.3.4 Accès direct et décompression

La méthode `get(int[] compressedArray, int i)` calcule la position de l'élément dans la zone principale, extrait la valeur compressée (flag + valeur), puis vérifie le flag. Si le flag est à 0, elle retourne la valeur directement. Si le flag est à 1, elle utilise la valeur comme index pour aller chercher l'élément réel dans la zone overflow.

La décompression fonctionne de la même manière que le `get()` mais pour chaque élément.

3.3.5 Cas d'usage

Cette méthode est particulièrement efficace quand le tableau contient des outliers. Elle permet d'économiser beaucoup plus d'espace que BASIC et NO_OVERLAP dans ce scénario. Par contre, le temps de compression est plus long car il faut calculer le nombre de bits optimal.

4 Architecture logicielle

4.1 Structure générale du code

Le projet est organisé autour d'une architecture modulaire qui respecte les principes du génie logiciel. J'ai structuré le code en plusieurs classes, chacune ayant une responsabilité claire :

- **Interface BitPacking** : définit les fonctions que toutes les méthodes de compression doivent respecter
- **Trois classes d'implémentation** : `BasicBitPacking`, `NoOverlapBitPacking` et `OverflowBitPacking`
- **Factory** : `BitPackingFactory` pour instancier les méthodes
- **Main** : interface utilisateur en ligne de commande
- **Benchmark** : système de mesure de performances

Cette organisation permet une séparation claire des responsabilités et facilite l'ajout de nouvelles méthodes de compression si besoin.

4.2 Le pattern Factory

J'ai utilisé le pattern Factory pour gérer la création des différentes implémentations de Bit Packing. Ce choix présente plusieurs avantages :

Extensibilité : Si je veux ajouter une nouvelle méthode de compression, il suffit de créer une nouvelle classe implémentant l'interface `BitPacking` et d'ajouter un cas dans la Factory. Le reste du code n'a pas besoin d'être modifié.

Séparation des responsabilités : Le code qui utilise les méthodes de compression n'a pas besoin de connaître les détails de construction. Il demande simplement une instance via la Factory en spécifiant le type désiré (`BASIC`, `NO_OVERLAP` ou `OVERFLOW`).

Centralisation : Toute la logique de création est centralisée dans un seul endroit, ce qui facilite la maintenance. Si je veux modifier la façon dont une méthode est instanciée, je n'ai qu'un seul endroit à modifier.

La Factory propose deux méthodes : une qui accepte un enum (type-safe) et une autre qui accepte une String (plus pratique pour l'interface utilisateur). La seconde convertit la String en enum puis utilise la première méthode avec.

4.3 L'interface BitPacking

L'interface `BitPacking` définit trois méthodes essentielles que toutes les implémentations doivent fournir :

`compress(int[] array)` : prend un tableau d'entiers en entrée et retourne le tableau compressé. Cette méthode effectue l'analyse nécessaire (trouver la valeur max, calculer les bits nécessaires, etc.) puis construit le tableau compressé avec les métadonnées en tête.

`decompress(int[] compressedArray)` : prend un tableau compressé et retourne le tableau original. Cette méthode extrait d'abord les métadonnées du header, puis reconstruit le tableau original élément par élément.

`get(int[] compressedArray, int i)` : permet l'accès direct au *i*-ème élément du tableau original sans avoir à décompresser l'ensemble. C'est cette méthode qui rend le Bit Packing intéressant pour les cas où on a besoin d'accéder à quelques éléments spécifiques plutôt qu'à tout le tableau.

Cette interface garantit que les trois méthodes peuvent être utilisées de la même manière. Le benchmark peut ainsi tester les trois implémentations avec exactement le même code.

4.4 Gestion des erreurs

J'ai mis en place une gestion d'erreurs pour traiter les cas que le système ne peut pas gérer. Trois types d'exceptions sont levées :

IllegalArgumentException pour les valeurs négatives : Le système ne supporte pas les entiers négatifs car la méthode `numberOfLeadingZeros()` ne fonctionne pas correctement avec eux. Les nombres négatifs sont représentés d'une certaine façon en Java qui fait qu'ils commencent toujours par un bit à 1. La méthode retournerait systématiquement 0, ce qui fausserait complètement le calcul du nombre de bits nécessaires. Plutôt que de produire des résultats incorrects, le système lève une exception explicite dès qu'une valeur négative est détectée.

IllegalArgumentException pour les tableaux trop grands : La structure de métadonnées utilise 26 bits pour stocker la taille du tableau, ce qui limite la capacité à 67 millions d'éléments environ. Si un tableau plus grand est fourni, une exception est levée avant le début de la compression pour éviter un débordement silencieux qui produirait des données corrompues.

IndexOutOfBoundsException pour les accès invalides : La méthode `get()` vérifie que l'index demandé est bien compris entre 0 et la taille du tableau. Un index négatif ou trop grand lève une exception appropriée.

5 Processus de développement

5.1 BASIC – Développement de la méthode de base

5.1.1 Phase de réflexion initiale

J'ai commencé le développement par la méthode BASIC. Ma première réflexion s'est portée sur les quatre étapes du Bit Packing décrites en section 3.1.

La première étape consistait à calculer le nombre de bits nécessaires. Pour cela, j'ai parcouru le tableau pour trouver la valeur maximale, puis utilisé la méthode `Integer.numberOfLeadingZeros()` de Java qui compte le nombre de zéros en tête de la représentation binaire.

5.1.2 Choix du sens de remplissage

La réflexion suivante a été déterminante pour toute la suite du projet : dans quel sens remplir les entiers lors du stockage ? On peut remplir de droite à gauche (du bit de poids faible vers le bit de poids fort) ou de gauche à droite (du bit de poids fort vers le bit de poids faible).

Techniquement, on peut calculer la position d'un élément dans les deux cas. Si on remplit de gauche à droite, la formule serait $\text{position} = (n - 1 - i) \times \text{bitsNécessaires}$ où n est le nombre total d'éléments. Le problème c'est qu'au moment de récupérer un élément avec `get()`, il faut connaître n pour faire ce calcul, ce qui ajoute une dépendance supplémentaire.

J'ai donc choisi de remplir de droite à gauche, c'est-à-dire du bit de poids faible vers le bit de poids fort. Le premier élément est stocké dans les bits de poids faible, le deuxième juste après, et ainsi de suite. Avec cette approche, la position de chaque élément est calculable

directement : $\text{position} = i \times \text{bitsNécessaires}$. La méthode `get()` devient plus simple et plus rapide car on n'a pas besoin de connaître le nombre total d'éléments ni de calculer une position.

5.1.3 Première version fonctionnelle

Une fois le sens de remplissage choisi, j'ai développé une première version fonctionnelle des trois méthodes principales : `compress()`, `get()` et `decompress()`. J'ai testé cette version manuellement en vérifiant la représentation binaire des entiers compressés pour m'assurer que les valeurs étaient bien stockées aux bonnes positions.

La méthode `compress()` devait gérer le cas où un élément chevauche deux entiers. Par exemple, avec 12 bits par élément, le troisième élément commence au bit 24 du premier entier et se termine au bit 4 du second. J'ai donc implémenté deux cas : un cas simple où l'élément tient dans un seul entier, et un cas complexe où il faut répartir les bits sur deux entiers consécutifs.

5.1.4 Phase d'optimisation

Une fois la version fonctionnelle validée, j'ai entamé une phase d'optimisation. L'objectif était de rendre le code le plus efficient possible, tant au niveau de l'architecture que des opérations utilisées.

Optimisation des opérations arithmétiques Une optimisation a été de remplacer certaines opérations arithmétiques par des opérations binaires. Cette optimisation est possible parce que 32 est une puissance de 2 ($32 = 2^5$).

Pour le modulo, l'opération $x \% 32$ peut être remplacée par $x \& 31$. Pourquoi ça marche ? Faire un modulo 32 revient à ne garder que le reste de la division par 32, c'est-à-dire les 5 bits de poids faible du nombre (car $32 = 2^5$). En binaire, 31 s'écrit 11111 (cinq bits à 1). Donc l'opération $x \& 31$ masque tous les bits sauf les 5 de poids faible, ce qui donne exactement le même résultat que $x \% 32$. Cette opération binaire est nettement plus rapide que le modulo arithmétique.

Pour la division, l'opération $x / 32$ peut être remplacée par $x \gg 5$. Diviser un nombre par 32 revient à décaler sa représentation binaire de 5 positions vers la droite (car $32 = 2^5$). Par exemple, $100/32 = 3$, et en binaire : $100 = 1100100$, et $1100100 \gg 5 = 11 = 3$. Cette opération de décalage est beaucoup plus rapide qu'une division.

Ces optimisations s'appliquent uniquement pour les puissances de 2. Pour un modulo 30 ou une division par 30, on ne pourrait pas utiliser ces raccourcis car 30 n'est pas une puissance de 2.

Autres optimisations J'ai également réduit le nombre de variables temporaires. Au lieu de stocker des résultats intermédiaires qui ne servent qu'une fois, je calcule directement les valeurs là où elles sont utilisées. Par exemple, plutôt que `numInts = (totalBits`

+ 31) / 32 suivi de `result = new int[1 + numInts]`, j'ai simplifié en `result = new int[1 + ((arraySize * bitsNeeded + 31) » 5)]`.

Enfin, j'ai utilisé l'opérateur `|=` plutôt que de réaffecter. Au lieu de `result[i] = result[i] | value`, j'écris `result[i] |= value`. C'est plus concis et légèrement plus rapide car on évite une lecture puis une écriture.

5.2 NO_OVERLAP – Adaptation de la méthode de base

Une fois BASIC optimisé, j'ai développé NO_OVERLAP. Cette méthode est en réalité plus simple à réaliser que BASIC car elle élimine la complexité du chevauchement.

Le principe reste le même au début : parcourir le tableau pour trouver la valeur maximale et calculer le nombre de bits nécessaires. La différence se situe dans le stockage. Au lieu de laisser les éléments chevaucher deux entiers, je calcule combien d'éléments peuvent tenir dans un entier : `elementsPerInt = 32/bitsNeeded`.

Quand j'arrive à la fin d'un entier et que l'élément suivant ne peut pas y tenir complètement, je passe simplement à l'entier suivant en laissant des bits inutilisés. Cela simplifie grandement la méthode `get()` : je suis certain que chaque élément est contenu dans un seul entier, donc je n'ai plus besoin de gérer le cas du chevauchement.

Cette simplicité a un coût : on perd quelques bits par entier (les bits inutilisés en fin d'entier). Le benchmark permettra de déterminer si cela vaut le coup.

5.3 OVERFLOW – Développement de la méthode sophistiquée

La méthode OVERFLOW a représenté le plus gros défi du projet. Toute la complexité résidait dans la recherche du nombre de bits optimal pour la zone principale.

5.3.1 Réflexions initiales et prototypes

Ma première idée était de calculer le nombre de « bits perdus » pour chaque configuration possible. J'appelle « bit perdu » un bit qui est utilisé pour un élément alors qu'il n'est pas strictement nécessaire. Par exemple, si un élément nécessite 5 bits mais qu'on utilise 7 bits pour tous les éléments, on perd 2 bits par élément.

J'avais en tête qu'en augmentant ou en baissant le nombre de bits, on pouvait faire varier ce nombre de bits perdus et trouver un optimal. Mon idée initiale était d'utiliser une approche dichotomique pour trouver rapidement cet optimal sans avoir à tester toutes les valeurs possibles.

Mais j'ai vite réalisé que cette approche ne fonctionnerait pas car la fonction de coût n'est pas monotone. Prenons un exemple avec un tableau contenant :

- 100 éléments nécessitant 3 bits
- 200 éléments nécessitant 7 bits
- 50 éléments nécessitant 10 bits
- 300 éléments nécessitant 15 bits

- 10 éléments nécessitant 25 bits

On aurait en bits perdus pour chaque k :

- $k = 3$: 13 610 bits perdus
- $k = 7$: 9 850 bits perdus
- $k = 10$: 10 230 bits perdus
- $k = 15$: 3 930 bits perdus

On voit que l'on a une courbe qui baisse puis remonte puis rebaisse, la recherche dichotomique pourrait donc se tromper dans certains cas. J'ai donc dû abandonner cette approche et il faudra donc tester toutes les valeurs de k présentes dans le tableau.

5.3.2 Solution finale : calcul du coût total

J'ai donc changé d'approche pour tester toutes les valeurs uniques. Au lieu de parcourir le tableau entier pour chaque candidat testé, j'ai optimisé en calculant directement le coût total pour chaque taille de bits effectivement présente dans le tableau.

L'algorithme fonctionne ainsi :

1. Je calcule d'abord pour chaque élément du tableau le nombre de bits qu'il nécessite individuellement
2. Je trie ces valeurs pour obtenir une liste ordonnée
3. Pour chaque valeur k distincte dans ce tableau trié, je calcule combien d'éléments iraient en overflow si on utilisait k bits pour la zone principale, puis j'en déduis le coût total

Le calcul du coût total est crucial et il ne faut rien oublier :

$$\text{coût_total} = n \times (k + 1) + \text{overflow_count} \times 32 \quad (4)$$

Où n est le nombre total d'éléments, k est le nombre de bits testé pour la zone principale, le $+1$ est essentiel car il représente le bit de flag qui indique si un élément est en overflow ou non, overflow_count est le nombre d'éléments qui nécessitent strictement plus de k bits, et ces éléments en overflow sont stockés en 32 bits complets.

L'algorithme vérifie également une contrainte importante : il faut pouvoir représenter l'index de débordement avec k bits. Par exemple, si on a 300 éléments en overflow, il faut au moins 9 bits car avec 8 bits on ne peut représenter que 256 valeurs ($2^8 = 256$) ce qui est insuffisant, alors qu'avec 9 bits on peut représenter jusqu'à 512 valeurs ($2^9 = 512$), ce qui couvre bien les 300 index nécessaires (de 0 à 299). Cette vérification évite des cas où l'algorithme choisirait un k trop petit pour encoder les index.

5.3.3 Développement de `compress()`, `get()` et `decompress()`

Une fois l'algorithme de recherche du k optimal fonctionnel, j'ai développé les méthodes de compression et décompression. Le principe est similaire à BASIC et NO_OVERLAP, avec quelques différences :

Pour `compress()` : je dois d'abord appeler `findBestBitSize()` pour déterminer le k optimal, puis construire le tableau avec la zone principale et la zone overflow. Chaque élément de la zone principale utilise $k + 1$ bits (dont 1 bit de flag).

Pour `get()` : je dois extraire l'élément compressé, vérifier le bit de flag, et soit retourner la valeur directement (`flag = 0`), soit l'utiliser comme index pour aller chercher dans la zone overflow (`flag = 1`).

Pour `decompress()` : je vais fonctionner comme le `get()` mais pour chaque élément.

6 Benchmark et analyse de performance

6.1 Protocole de mesure

6.1.1 Phase de warm-up

Avant toute mesure, j'exécute 200 itérations de compression sans mesurer le temps. Cette phase de warm-up en Java permet au compilateur d'optimiser le code. Les premières exécutions d'une méthode sont toujours plus lentes car le code est interprété.

6.1.2 Phase de mesure

Une fois le warm-up effectué, je mesure précisément trois opérations :

La compression : j'exécute 500 compressions du même tableau et je mesure le temps de chaque itération avec `System.nanoTime()`. Je calcule ensuite la moyenne pour obtenir un temps de compression représentatif.

La décompression : même principe avec 500 décompressions du tableau compressé.

L'accès direct avec `get()` : pour mesurer cette opération, j'effectue 500 itérations où chaque itération contient 10 accès aléatoires à différents index du tableau. Au total, 5000 appels à `get()` sont mesurés.

6.1.3 Distributions testées

J'ai testé trois scénarios représentatifs qui couvrent différents cas d'usage :

Distribution uniforme (0-100) : tous les éléments sont entre 0 et 100, nécessitant 7 bits. C'est le cas optimal pour le Bit Packing car toutes les valeurs sont petites.

Distribution avec outliers (2% grandes valeurs) : 98% des valeurs sont entre 0 et 100, mais 2% sont entre 100 et 1 000 000. Ce scénario teste l'efficacité de la méthode OVERFLOW face aux outliers.

Distribution grandes valeurs (0-100 000) : toutes les valeurs sont entre 0 et 100 000, nécessitant 17 bits. C'est le cas défavorable où la compression apporte moins de gain.

6.1.4 Tailles testées

Pour chaque distribution, j'ai testé trois tailles de tableau : 1 000, 10 000 et 100 000 éléments. Cela permet de voir comment les méthodes se comportent à différentes échelles.

6.2 Résultats et comparaisons

Les résultats du benchmark révèlent des différences entre les trois méthodes selon le type de données. (La méthode OVERFLOW peut avoir besoin de plus d'entiers que la méthode BASIC avec pourtant le même nombre de bits pour les entiers compressés car il stock en plus le flag)

6.2.1 Cas uniforme (0-100) – 10 000 éléments

Méthode	Compress. (μ s)	Décompress. (μ s)	get() (ns)	Taille compressée	Gain espace
BASIC	17.0	18.5	35	2189 ints 249 952 bits	78.1%
NO_OVERLAP	11.7	32.9	36	2501 ints 239 968 bits	75.0%
OVERFLOW	85.7	23.8	55	2501 ints 239 968 bits	75.0%

TABLE 1 – Résultats pour distribution uniforme (0-100)

Dans ce cas, NO_OVERLAP est la plus rapide en compression mais plus lente en décompression. BASIC offre un meilleur équilibre entre vitesse et gain d'espace. OVERFLOW est nettement plus lente en compression à cause de l'algorithme `findBestBitSize()`, mais n'apporte pas de gain supplémentaire car il n'y a pas d'outliers.

6.2.2 Cas avec outliers (2% grandes valeurs) – 10 000 éléments

Méthode	Compress. (μs)	Décompress. (μs)	get() (ns)	Taille compressée	Gain espace
BASIC	23.3	25.8	56	6251 ints 119 968 bits	37.5%
NO_OVERLAP	12.9	31.8	28	10001 ints -32 bits	-0.01%
OVERFLOW	75.0	15.6	27	3950 ints 193 600 bits	60.5%

TABLE 2 – Résultats pour distribution avec outliers (2% grandes valeurs)

Dans ce cas, les différences sont plus importantes. NO_OVERLAP ne compresse quasiment pas (elle perd même quelques bits à cause des métadonnées). BASIC fait mieux mais reste limitée. OVERFLOW économise 193 600 bits contre 119 968 pour BASIC, soit un gain de 73 632 bits. Le temps de compression d'OVERFLOW est environ 3 fois plus long que BASIC, mais la décompression est plus rapide. Le `get()` est également plus rapide avec OVERFLOW qu'avec BASIC.

6.2.3 Cas grandes valeurs (0-100 000) – 10 000 éléments

Méthode	Compress. (μs)	Décompress. (μs)	get() (ns)	Taille compressée	Gain espace
BASIC	16.9	16.9	24	5314 ints 149 952 bits	46.9%
NO_OVERLAP	12.4	31.0	22	10001 ints -32 bits	-0.01%
OVERFLOW	73.4	23.7	40	5626 ints 139 968 bits	43.7%

TABLE 3 – Résultats pour distribution grandes valeurs (0-100 000)

Dans ce cas, BASIC et OVERFLOW offrent des gains similaires. NO_OVERLAP ne compresse toujours pas à cause de la limite d'éléments par entier. BASIC est la plus rapide en compression et décompression. Le `get()` est le plus rapide avec NO_OVERLAP malgré l'absence de compression.

6.2.4 Cas d'usage recommandé pour chaque méthode

BASIC est un bon choix par défaut. Elle offre un gain d'espace correct dans tous les cas et des temps de compression et décompression corrects également. Cette méthode va être

utile quand on ne connaît pas la distribution des données à l'avance ou quand on veut un comportement prévisible.

NO_OVERLAP est intéressante uniquement pour des valeurs très petites où elle peut offrir des temps de compression courts. Cependant, dès que les valeurs nécessitent plus de 10 bits, elle devient inefficace car elle ne peut plus stocker plusieurs éléments par entier. Son `get()` reste rapide dans tous les cas grâce à l'absence de chevauchement.

OVERFLOW est très utile quand le tableau contient des outliers. Le gain d'espace est bien meilleur que les autres méthodes pour ce cas. Bien que le temps de compression soit plus long, la décompression et le `get()` restent meilleurs. Cette méthode va être utile quand on sait que les données contiennent quelques valeurs aberrantes parmi beaucoup de petites valeurs.

6.3 Impact des outliers

Les tests montrent l'importance du choix de la méthode en fonction de la distribution des données.

Sur les données uniformes, les trois méthodes se comportent de manière similaire en termes de gain d'espace. BASIC et OVERFLOW économisent environ 78% de l'espace, tandis que NO_OVERLAP économise 75%.

Mais en présence d'outliers, les différences deviennent importantes. NO_OVERLAP ne compresse pas car les grandes valeurs forcent l'utilisation de beaucoup de bits, et sans le chevauchement on se retrouve avec un élément par entier. BASIC parvient à compresser mais perd beaucoup d'efficacité car elle doit utiliser autant de bits pour les petites valeurs que pour les grandes. OVERFLOW résout ce problème en séparant les outliers dans une zone dédiée, ce qui permet de conserver un bon taux de compression.

Le coût en temps de compression d'OVERFLOW (3 à 4 fois plus long que BASIC) est compensé par le fait que la décompression et le `get()` restent rapides et un bien meilleur gain de place. Si on doit transmettre les données sur un réseau, le temps gagné grâce à la réduction de taille peut largement compenser ce surcoût de compression.

6.4 Analyse de la latence de rentabilité

Le benchmark permet de répondre à la question centrale : à partir de quelle latence réseau la compression devient-elle rentable ?

6.4.1 Calcul de la latence de rentabilité

Pour qu'une compression soit avantageuse, le temps gagné sur la transmission doit compenser le surcoût de compression et décompression. La latence de rentabilité se calcule ainsi :

$$\text{latence_rentabilité} = \frac{\text{temps_compression} + \text{temps_décompression}}{\text{bits_économisés}} \quad (5)$$

Cette formule donne le temps de transmission par bit au-delà duquel la compression est rentable. On peut la convertir en débit :

$$\text{débit_rentabilité} = \frac{1}{\text{latence_rentabilité}} \quad (6)$$

6.4.2 Résultats pour le cas uniforme (10 000 éléments, 0-100)

- **BASIC** : $(17.0 + 18.5) \mu s / 249\,952 \text{ bits} = 0.142 \text{ ns/bit} \rightarrow 7\,042 \text{ Mbps}$
- **NO_OVERLAP** : $(11.7 + 32.9) \mu s / 239\,968 \text{ bits} = 0.186 \text{ ns/bit} \rightarrow 5\,376 \text{ Mbps}$
- **OVERFLOW** : $(85.7 + 23.8) \mu s / 239\,968 \text{ bits} = 0.456 \text{ ns/bit} \rightarrow 2\,193 \text{ Mbps}$

6.4.3 Résultats pour le cas avec outliers (10 000 éléments, 2% grandes valeurs)

- **BASIC** : $(23.3 + 25.8) \mu s / 119\,968 \text{ bits} = 0.409 \text{ ns/bit} \rightarrow 2\,444 \text{ Mbps}$
- **NO_OVERLAP** : ne compresse pas, non pertinent
- **OVERFLOW** : $(75.0 + 15.6) \mu s / 193\,600 \text{ bits} = 0.468 \text{ ns/bit} \rightarrow 2\,137 \text{ Mbps}$

6.4.4 Interprétation pour les réseaux réels

Ces chiffres permettent de déterminer sur quels types de réseau la compression est rentable :

Réseau local rapide (10 Gbps) : la compression n'est pas rentable dans le cas uniforme, le temps de transmission serait plus court sans compression. Par contre, avec des outliers, OVERFLOW peut être intéressante car le gain d'espace est important.

LAN Gigabit (1000 Mbps) et en dessous – WiFi (100-300 Mbps), Internet domestique (50-100 Mbps), 4G/5G (10-100 Mbps) : la compression devient rentable pour toutes les méthodes dans tous les cas.

6.4.5 Choix de la méthode selon le réseau

Sur réseau rapide ($> 5 \text{ Gbps}$), si on doit compresser, privilégier BASIC pour son temps de traitement court dans le cas uniforme, et OVERFLOW dans le cas avec outliers malgré son temps de compression plus long car le gain d'espace compense.

Sur réseau moyen (100 Mbps - 1 Gbps), toutes les méthodes sont rentables. Le choix dépend uniquement de la distribution des données : BASIC par défaut, OVERFLOW si outliers.

Sur réseau lent ($< 100 \text{ Mbps}$), la compression est indispensable. OVERFLOW est recommandée car même si elle prend plus de temps à compresser, le gain d'espace est tel que le temps de transmission total reste meilleur.

7 Conclusion

Ce projet m'a permis de développer et d'analyser trois méthodes de compression par Bit Packing : **BASIC**, **NO_OVERLAP** et **OVERFLOW**. Chacune a ses avantages selon le contexte d'utilisation.

BASIC offre un bon compromis général avec des performances équilibrées et un gain d'espace correct dans tous les cas. C'est le choix par défaut quand on ne connaît pas la distribution des données.

NO_OVERLAP simplifie l'accès aux éléments en évitant le chevauchement, ce qui accélère le `get()`. Par contre, elle devient inefficace dès que les valeurs nécessitent plus de 10 bits.

OVERFLOW est indispensable en présence d'outliers. Elle économise plus d'espace que les autres méthodes dans ce scénario, malgré un temps de compression plus long.

Le benchmark a permis de calculer les latences de rentabilité pour chaque méthode. Sur des réseaux de moins de 5 Gbps, la compression est largement rentable. Le choix de la méthode dépend alors uniquement de la distribution des données : **BASIC** par défaut, **OVERFLOW** si des outliers sont présents.

Ce projet m'a également permis d'approfondir mes connaissances en génie logiciel, notamment l'utilisation du pattern Factory, la conception d'interfaces, l'optimisation du code et l'importance des benchmarks pour évaluer objectivement les performances.

Le code source complet du projet est disponible sur GitHub :

<https://github.com/RomainStefani04/SoftwareEngineering>