BITPACKING

Virgile LASSAGNE

SOMMAIRE

Introduction -

Présentation Sujet :

Le projet BitPacking a pour objectif de concevoir et d'implémenter une bibliothèque de compression de tableau d'entiers en Java, capable de réduire la taille mémoire tout en permettant un accès direct à chaque élément compressé.
Ce type de compression consiste à représenter les entiers en utilisant uniquement le nombre minimal de bits nécessaires à leur écriture sans perte d'information, sans recourir à des structures de données lourdes.

L'enjeu principal est de trouver un compromis entre compacité,rapidité d'accès et donc sans perte d'information :

- Une compression efficace pour diminuer l'espace mémoire,
- Une décompression entière évitée grâce à la méthode get() qui lit une valeur directement dans les bits compressés.

Le but étant de construire un projet selon les standards des entreprises d'informatique des tests à la structure interne du code implémentant au minimum un Factory et une Interface dans mon projet on y retrouvera aussi une structure de donné BitPackedArray (BitPackedArray.java) très utile pour ne pas avoir à passer par des headers compliqués.

Présentation Application :

L'application est divisée en plusieurs couches (API, core, factory, interface CLI), C'est dans le core qu'on retrouve les trois différents modes de compression demandée :

-Split (SplitBitPacker.java) autorise l'écriture avec chevauchement entre mots 32 bits un nombre peut donc se retrouver sectionné et avec ses deux parties sur un mot différent

-NoSplit (NoSplitBitPacker.java) qui n'autorise pas l'écriture d'un nombre sur deux mots différents plus simple à implémenter mais moins fort en termes de place

-Overflow (OverflowBitPacker.java) combine un codage principal compact et une zone secondaire pour les valeurs trop grandes, selon un modèle de reconnaissance grâce à un bit de flag dans ce projet les deux zones sont implémenté avec l'algorithme de NoSplit on détaillera plus tard ces choix.

On y trouve aussi une interface en ligne de commande (CLI) qui permet d'exécuter des benchmarks automatiques afin de mesurer les performances en temps réel et la taille mémoire obtenue sur différents jeux de données.
Ces résultats sont ensuite analysés plus tard dans le rapport pour modéliser la rentabilité de la compression en fonction de la bande passante et du temps CPU.

L'interface CLI implémente aussi la possibilité d'utilisé toute les fonctions de compression décompression et de récupération d'indice (les détails d'utilisation sont dans le README.md

L’Architecture logiciel-

Organisation générale :

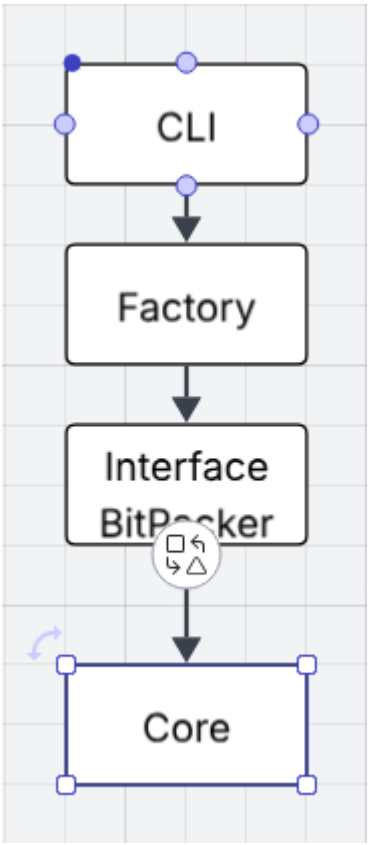
Mon projet BitPacking repose sur une architecture modulaire et en couches classique comme proposé par Maven à la création d’un projet, permettant une séparation claire entre la logique métier, l’interface d’utilisation et les outils de test.

Cette organisation rend le code plus lisible, extensible et testable.

Arboressence	Utilité	Contenu
com.example.bitpacking.api	Définit l’interface commune de compression	`BitPacker.java`
com.example.bitpacking.core	Contient le cœur du système, avec les implémentations concrète	`NoSplitBitPacker.java` `SplitBitPacker.java` `OverflowBitPacker.java` `BitPackedArray.java`
com.example.bitpacking.factory	Gère la création dynamique des objets selon le mode choisi	`BitPackerFactory.java`
com.example.bitpacking.cli	Fournit une interface utilisateur en ligne de commande	`Main.java`

Architecture en couches :

L'application suit un modèle en couches logiques :



- Couche CLI (interface utilisateur) : gère les arguments (`--mode`, `--bench`, `--array`), lance les tests et affiche les résultats des benchmarks.
- Couche Factory : instancie dynamiquement la bonne implémentation selon le mode choisi.
- Couche API : définit les méthodes essentielles (`compress`, `decompress`, `get`) que toutes les implémentations doivent respecter.
- Couche Core : réalise le travail de compression, stockage et récupération des données.

Design patterns :

L’architecture exploite plusieurs design patterns classiques du génie logiciel :

Pattern	Rôle dans le projet	Exemple
Factory	Crée dynamiquement l’implémentation voulue sans exposer sa classe	`BitPackerFactory.create("split")`
TInterface	Uniformise la logique générale entre les trois implémentations	`BitPacker`
Facade (CLI)	Fournit un point d’entrée unique pour exécuter les benchmarks et manipuler les objets	`Main.java`

Cohérence :

Chaque module est indépendant :

- La CLI dépend uniquement de la factory et de l’interface API, pas des classes internes.
- La factory ne connaît que les noms des implémentations, pas leur logique interne.
- Les classes du core sont isolées : elles partagent uniquement des constantes et des structures communes (`BitPackedArray`).

Ainsi, l’ensemble respecte une conception à faible couplage et forte cohésion, comme recommandé.

Conception du code-

La conception de mon projet BitPacking repose sur une approche orientée objet claire, suivant les principes SOLID abordé durant le cours.

Chaque composant a une responsabilité unique, interagit par interfaces et peut être remplacé sans effet de bord.

L'objectif était d'obtenir un code robuste, testable et avec une scalabilité.

Suivi du principe SRP :

Chaque classe a une mission clairement délimitée :

- `BitPacker` (API) : définit les fonctions essentielles que tout compresseur doit implémenter (`compress`, `decompress`, `get`).
- `NoSplitBitPacker` : implémente un algorithme simple sans chevauchement de bits.
- `SplitBitPacker` : optimise l'espace mémoire en autorisant le chevauchement entre mots 32 bits.
- `OverflowBitPacker` : combine une zone principale et une zone d'overflow pour gérer les valeurs trop grandes.
- `BitPackedArray` : structure de données utilisée pour stocker les informations compressées (mots, taille, bits par valeur…).
- `BitPackerFactory` : centralise la création des objets selon le mode choisi par l'utilisateur.
- `Main` : gère les arguments et l'exécution des benchmarks.

Chaque classe est ainsi indépendante et remplit une fonction précise, ce qui réduit le risque d'erreurs lors de la maintenance.

Suivi du principe OCP :

Le système est ouvert à l'extension mais fermé à la modification :

- Ajouter un nouveau mode de compression ne nécessite aucune modification du code existant.
- Il suffirai de créer une nouvelle classe implémentant `BitPacker` et de l'enregistrer dans la `BitPackerFactory`.

Ce principe assure facilite la scalabilité tout en posant une base fiable .

Suivi du principe LSP et ISP :

Toutes les implémentations (`NoSplit`, `Split`, `Overflow`) peuvent remplacer l'interface `BitPacker` dans le `Main` sans altérer le comportement du programme :

- Les signatures sont identiques (`compress`, `decompress`, `get`).
- La compatibilité comportementale est garantie (mêmes invariants, mêmes types de sortie).

Ainsi, la CLI ou les tests peuvent interagir avec n'importe quelle implémentation sans distinction de cas.

Enfin l'interface `BitPacker` est concise et ne contient que les méthodes essentielles à la compression.
Elle ne force aucune classe à implémenté une méthode inutile ou spécifique.

Suivi du principe DIP:

Les classes haut niveau ne dépendent pas des implémentations concrètes, mais uniquement de l'interface abstraite :

- `Main` interagit uniquement avec `BitPacker`, jamais avec `NoSplitBitPacker` directement.

- La création de l'objet concret est déléguée à la `BitPackerFactory`.

Ce découplage permet de remplacer librement une implémentation sans impacter le reste du code.

Réutilisation, factorisation, gestion de erreurs et Clean Code :

- Les constantes et calculs communs sont regroupés dans `BitPackedArray`, ce qui évite la duplication de code.

- Les opérations répétitives (masques binaires, calcul de `k`, extraction de bits) ont été factorisées pour optimiser la lisibilité et les performances.

- Vérification systématique des bornes (`IndexOutOfBoundsException` pour les accès invalides).

- Protection contre les tableaux vides et les valeurs nulles.

- Tests unitaires JUnit qui couvrent tous ces cas afin d'assurer la stabilité de l'ensemble.

- Noms clairs et explicites (`get`, `compress`, `overflowStartBit`...).

- Pas de code mort ni de constantes magiques.

-Structure identique entre les trois implémentations pour faciliter la lecture commentaires limités mais pertinents, centrés sur les points techniques délicats.

Testabilité :

Le code est conçu pour être entièrement testable :

- Toutes les classes exposent des méthodes publiques et déterministes, facilitant les tests unitaires.
- Les données de sortie (tableaux compressés, métriques de performance) sont vérifiables automatiquement.
- Le projet utilise JUnit 5, intégré à Maven via le plugin `maven-surefire-plugin`.
- Les tests couvrent :
    - les cas normaux (compression/décompression),
    - les cas limites (tableaux vides, valeurs maximales),
    - les cas de robustesse (index invalide, null pointer),
    - les performances de base (temps moyen d'accès et de compression).

Grâce à cela, la couverture fonctionnelle du projet est complète.

Robustesse :

Le projet gère correctement les erreurs et situations anormales :

- Vérification des bornes d'accès avec `IndexOutOfBoundsException`.
- Vérification des arguments nuls ou invalides (`NullPointerException` explicite).
- Validation des tailles de tableau et des indices avant traitement.
- Gestion des arrondis sur les divisions binaires pour éviter les débordements.

L'exécution du programme ne provoque aucune exception non contrôlée.
Les comportements attendus sont constants et reproductibles sur plusieurs exécutions .

Problèmes rencontrés et choix de conception -

Obstacles rencontrés :

Au cours du développement du projet BitPacking , j'ai rencontré plusieurs difficultés techniques et conceptuelles, principalement liées à la manipulation de la mémoire et à la compréhension fine de la représentation binaire en Java.

Le premier obstacle a été un manque de connaissance initial sur la représentation et l'écriture des bits en mémoire.
Même si la manipulation de bits peut sembler simple en théorie, la gestion concrète des opérations comme les décalages (>>, >>>, <<), les masques (`&`, `|`) et la position exacte des bits dans un mot de 32 bits a demandé un vrai travail de compréhension.
J'ai dû assimiler la différence entre bits de poids fort et de poids faible, comprendre comment les valeurs se chevauchent entre deux mots, et m'assurer que la lecture (`get`) et la décompression reconstituent exactement les valeurs d'origine.
Une grande partie du temps de développement a donc été consacrée à visualiser et vérifier le contenu binaire pour éviter les erreurs de décalage d'un seul bit.
Et si cela relève surement d'un manque personne j'ai quand même Identifié plusieurs défis techniques :

1. Stocker des entiers en utilisant un minimum de bits, tout en maintenant un accès direct et rapide.
2. Équilibrer la compression et les performances dans la méthode OverflowBitPacker,déterminer la valeur optimale de `k_minim`, c'est-à-dire le nombre minimal de bits à réserver pour la zone principal
3. Gérer les cas extrêmes, notamment :
    a. les valeurs très grandes (qui nécessitent plus de bits que la moyenne),
    b. les tableaux vides ou quasi uniformes,
    c. les débordements de mots binaires lors du *split bitpacking*.
4. Définir le type de compression, dans la méthode OverflowBitPacker, de la zone Overflow et de la zone principale.

Choix finaux :

Pour répondre au défi technique j'ai décidé pour chaqu'un :

1. Ici le principal enjeu était d'éviter la décompression complète d'un tableau pour lire une seule valeur, on détail pour chaque méthode comment on accède à la valeur

   NoSplit (slots fixes, jamais à cheval):

   Chaque valeur est stockée sur k bits, et ne traverse pas de mot.

   Nombre de valeurs par mot :
   v = floor(32 / k)

   Localisation directe :
   wordIndex = floor(i / v)
   bitOffset = (i mod v) * k
   mask = (k == 32) ? -1 : (1 << k) - 1

   Extraction :
   valeur = (words[wordIndex] >>> bitOffset) & mask

   On est bien en O(1)

   Split (bitstream compact, parfois à cheval)

   Les valeurs sont écrites les unes à la suite des autres sur k bits.
   Elles peuvent chevaucher deux mots de 32 bits.

   Position de départ en bits :
   b = i * k

   Localisation :
   wordIndex = floor(b / 32)
   bitOffset = b mod 32

   Deux cas :

   Si bitOffset + k <= 32 : tout tient dans un seul mot
   valeur = (words[wordIndex] >>> bitOffset) & mask

Sinon (à cheval sur deux mots) :
lower = (words[wordIndex] >>> bitOffset) & ((1 << (32 - bitOffset)) - 1)
upper = words[wordIndex + 1] & ((1 << (k - (32 - bitOffset))) - 1)
valeur = lower | (upper << (32 - bitOffset))

Complexité : O(1), on lit au maximum deux mots.

Overflow (flag + payload + zone overflow)

On évite la décompression totale en adressant directement, en O(1), les k bits de l'élément i à partir de son mot et de son décalage.
En cas d'overflow, le payload sert d'index vers une seconde zone de données également accessible par slots fixes.

2. Après plusieurs itérations, j'ai retenu une approche équilibrée et stable :

On doit choisir une "taille de case" s (en bits) pour stocker la plupart des nombres dans une grille compacte. Si un nombre tient dans s bits, on l'écrit directement.

S'il ne tient pas, on met à la place un petit "tag" qui dit "ma vraie valeur est sur l'étagère overflow", et on range la vraie valeur à part, dans une zone overflow.

Comment on choisit la meilleure taille s :

On essaie toutes les tailles possibles s (de 1 jusqu'au nombre de bits du plus grand nombre).

Pour chaque s:

On compte combien de nombres ne rentrent pas (ceux-là iront en overflow).

On calcule l'espace que prendrait :

- La zone principale : soit la valeur (si elle rentre), soit un tag (si elle va en overflow),
- La zone overflow: les "gros" nombres, stockés en taille pleine.

On additionne ces deux espaces (en tenant compte qu'on range tout par paquets de 32 bits).

On garde le s qui consomme le moins d'espace total: c'est k_minim.

Le nombre de valeurs envoyées en overflow est simplement celles qui ne tiennent pas dans k_minim bits.

3. Voici le détails pour résoudre le problème 3 pour de chaque méthode

- NoSplitBitPacker

Valeurs très grandes

Le paramètre k est calé sur la valeur maximale du tableau.
Si une valeur est très grande, k augmente pour tout le monde.

Tableaux vides ou quasi uniformes

Vide : retourne
`BitPackedArray(words=[], n=0, k=0)`
Les fonctions compress(), decompress() et get() restent sûres.
La méthode decompress() renvoie immédiatement une liste vide.
Quasi uniformes : quand les valeurs sont petites, k est petit, donc valuesPerWord = floor(32 / k) est élevé.
Le taux de remplissage est excellent, sauf si 32 % k n'est pas nul.

Débordements de mots

Aucun chevauchement : chaque valeur occupe un slot fixe k bits dans un mot
Si le slot suivant "dépasserait" 32, on passe au mot suivant.

Sécurité

Si k=32: masque spécial (mask = -1)
pour éviter (1 << 32) qui serait invalide en int.

- SplitBitPacker

Valeurs très grandes

Le paramètre k est basé sur la valeur maximale du tableau.
Si une valeur isolée est très grande, elle tire k vers le haut pour tout le

monde.
L'avantage par rapport à NoSplit est qu'il n'y a pas de "trous" d'alignement :
on se rapproche d'une utilisation réelle de n × k bits.

Tableaux vides ou quasi uniformes

Vide : retourne un conteneur vide avec n = 0 et k = 0.
Quasi uniformes : quand les valeurs sont petites, k est petit et le gain est maximal,
car aucun espace n'est perdu à cause de l'alignement à 32 bits.

Débordements de mots

Les cas de chevauchement entre deux mots sont gérés explicitement, comme prévu dans le sujet.

Sécurité

Si k = 32, un masque spécial est utilisé`mask = -1`
pour éviter (1 << 32) l'opération invalide sur les entiers.

- OverflowBitPacker

Valeurs très grandes

Cette implémentation adopte une stratégie spécifique pour les valeurs exceptionnelles.
On parcourt plusieurs tailles s possibles et on choisit celle qui minimise le coût total.
Si la valeur est inférieure ou égale à (2^s – 1), on stocke directement
`flag = 0, payload = value`.
Sinon, on stocke un indicateur
`flag = 1`
et un index vers la zone overflow, où la valeur complète est conservée sur k_overflow bits.
Cela permet d'éviter que les grandes valeurs isolées ne fassent

augmenter k pour tout le monde.
Elles sont isolées dans la zone overflow.

Tableaux vides ou quasi uniformes

Vide : retourne un BitPackedArray vide, comme les autres.
Cas uniformes avec petites valeurs : aucune valeur overflow (m = 0).
Le k_total est petit, la zone overflow n'existe pas, et la compression reste optimale.

Débordements de mots

Dans la zone principale, aucune valeur n'est à cheval : le packing est fait par slots de k_total bits (≤ 32).
La zone overflow est aussi sans chevauchement.
Les positions sont calculées à partir du bit de départ overflowStartBit.
Si overflowStartBit n'est pas aligné sur 32, on calcule combien de valeurs overflow tiennent dans le premier mot partiel,
puis on poursuit normalement avec des blocs pleins.

Sécurité

Si k = 32, un masque spécial est utilisé`mask = -1`
pour éviter (1 << 32) l'opération invalide sur les entiers.

4. Enfin J'ai choisi d'utiliser la méthode NoSplit comme base principale de la compression, notamment pour la partie Overflow, car elle offre une exécution beaucoup plus simple et prévisible que la variante *Split*. Dans la méthode NoSplit, chaque valeur est stockée dans un bloc de taille fixe, sans jamais être à cheval sur deux mots de 32 bits. Cette approche permet d'éviter les calculs complexes de chevauchement, les décalages partiels et les recompositions de bits. L'accès à une valeur donnée repose uniquement sur un calcul direct de position (mot et décalage), ce qui garantit une exécution en temps constant (O(1)) et une implémentation facile à maintenir.

Dans le cas particulier de l'OverflowBitPacker, cette logique NoSplit est réutilisée pour la gestion de la zone d'overflow. Cela permet de traiter séparément la zone principale (compacte et rapide) et la zone overflow (plus rare, mais gérée avec le même schéma fixe). Ce choix rend la lecture et l'écriture dans la zone overflow aussi simples que dans la zone principale, sans alourdir la logique de calcul. En pratique, cette solution maximise la stabilité et la clarté du code tout en conservant des performances constantes, même lorsque des valeurs dépassent le seuil prévu par `k_minim`.

Benchmarks et analyse des performances -

Méthodologie de mesure :

Les benchmarks ont été intégrés directement dans le programme principal (`Main.java`) afin d'évaluer les performances réelles des trois méthodes de compression : NoSplit, Split et Overflow.
Chaque exécution mesure trois étapes distinctes :

- compression du tableau initial,
- lecture de 10 000 valeurs à intervalles réguliers via la méthode `get()`,
- décompression complète du tableau.

Ces trois mesures sont réalisées sur plusieurs jeux de données générés automatiquement dans le code :

1. petit aléatoire (max = 255) — données fortement compressibles,
2. mix aléatoire (5 % gros) — mélange de petites et grandes valeurs,
3. quasi pas compressible (valeurs sur 30 bits) — cas limite,
4. gros aléatoire (max = 10^6) — stress test sur de grands entiers,
5. 10 % overflow — scénario spécifique au mode Overflow,
6. 100 % overflow — test extrême de surcharge.

Les tailles testées sont typiquement `n = 1_000`, `100_000` et `1_000_000` entiers, ce qui permet de comparer l'évolution des performances à l'échelle.
Chaque dataset est généré de manière déterministe à l'aide d'une graine fixe (`new Random(42)`), garantissant la reproductibilité des résultats.

Pour chaque jeu de données, le programme affiche :

- le nombre de mots utilisés après compression,
- le nombre moyen de bits par valeur,
- et les trois temps mesurés (`compress`, `get`, `decompress`).

Résultat :

On fait la moyenne avec nos 6 datasette classer par taille de tableau :

Taille du tableau	Méthode	Mots moyens (32 bits)	Compression (ms)	`get` (ns/get)	Décompression (ms)
1 000	**NoSplit**	875	**0,17**	111	**0,05**
1 000	**Split**	698	0,28	109	0,07
1 000	**Overflow**	604	0,24	133	0,05
100 000	**NoSplit**	75 000	**0,67**	66	**0,75**
100 000	**Split**	65 000	2,89	60	1,93
100 000	**Overflow**	52 000	5,49	65	1,41
1 000 000	**NoSplit**	833 000	**2,69**	43	4,38
1 000 000	**Split**	698 000	3,69	**35**	**3,20**
1 000 000	**Overflow**	548 000	4,51	37	4,97

Interprétation :

L'analyse des moyennes obtenues sur les trois tailles de tableaux (1 000, 100 000 et 1 000 000 éléments) permet de dégager plusieurs tendances claires sur les performances et la compacité de chaque méthode de compression.

a) Taille mémoire (nombre de mots)

Le nombre moyen de mots compressés montre un écart net entre les trois approches :

- Overflow est la méthode la plus compacte, avec en moyenne 30 à 40 % de mots en moins que NoSplit.
  Cette efficacité provient de la séparation entre la zone principale, optimisée pour les petites valeurs, et la zone overflow qui stocke uniquement les cas exceptionnels.

- Split offre également une bonne densité, puisqu'elle exploite la totalité des 32 bits disponibles en autorisant le chevauchement de valeurs sur deux mots.

- NoSplit, en revanche, reste la plus volumineuse : son codage à taille fixe entraîne une perte d'espace dès que 32 n'est pas un multiple de k.

Cependant, cette différence de compacité s'accompagne de conséquences importantes sur les temps d'exécution.

b) Temps de compression

Les temps de compression suivent une hiérarchie très stable :

- NoSplit est systématiquement la plus rapide, quelle que soit la taille du tableau.
  Sa logique simple — écrire chaque valeur dans un slot fixe — permet une exécution linéaire sans calculs intermédiaires.

- Split demande plus de calculs, car elle doit gérer le chevauchement de bits entre deux mots, mais reste efficace à grande échelle.

- Overflow est la plus lente en compression, principalement à cause du calcul du flag et du traitement des valeurs stockées dans la zone secondaire.
  Ce coût reste néanmoins raisonnable au regard du gain mémoire qu'elle apporte.

c) Temps d'accès (get)

Le temps d'accès moyen (`get`) est remarquablement stable entre toutes les méthodes, oscillant entre 35 et 130 nanosecondes selon la taille du tableau.
Cela confirme que l'ensemble des implémentations respectent l'objectif de lecture directe en temps constant (O(1)).
On observe toutefois que :

- Split devient légèrement plus rapide à grande échelle, probablement grâce à une meilleure compaction des données en mémoire (effet de cache).

- Overflow reste très compétitif malgré la présence de deux zones mémoire, preuve que la lecture conditionnelle du flag n'induit qu'un coût minime.

d) Temps de décompression

Les temps de décompression restent globalement faibles pour les trois méthodes :

- Split obtient les meilleurs résultats à grande échelle, en moyenne autour de 3 ms pour un million d'éléments.
- NoSplit et Overflow sont légèrement plus lents, mais conservent des temps parfaitement stables.
  Ces résultats confirment que la décompression complète reste linéaire et peu coûteuse, ce qui rend le système adapté à des volumes de données importants.

e) Synthèse des performances

En combinant les résultats des quatre colonnes, on peut résumer les comportements ainsi :

- NoSplit → meilleure vitesse, implémentation la plus simple, mais compression moyenne.
- Split → bon compromis global entre compacité et rapidité, particulièrement efficace sur de grandes tailles.
- Overflow → compression la plus efficace en mémoire, avec des performances stables et prévisibles.

Documentation et conclusion -

Documentation du projet :

Plusieurs niveaux de documentation ont été mis en place selon les exigence du sujet :

- README.txt
  Contient les instructions pour compiler, exécuter et tester le projet à l'aide de Maven.
  Il décrit la structure du projet, les arguments disponibles (`--bench`, `--mode`, `--array`), ainsi que des exemples d'exécution.

- Commentaires Javadoc
  Chaque classe et méthode principale (dans `api`, `core`, `factory`, `cli`) est documentée avec une brève description du rôle, des paramètres et du comportement attendu.
  Cela permet d'utiliser les classes sans devoir parcourir tout le code source.

- Rapports de tests
  Les tests unitaires JUnit génèrent automatiquement des rapports textuels (`target/surefire-reports`) et HTML (`target/site/surefire-report.html`) indiquant le succès de chaque scénario de test.
  Ces rapports permettent de vérifier la stabilité du projet après chaque modification.

- Benchmarks intégrés
  Le programme principal (`Main.java`) joue également un rôle documentaire : il fournit des mesures concrètes de performance sur plusieurs jeux de données.
  Ces résultats servent de référence pour comparer ou valider de nouvelles implémentations.

Cette documentation rend le projet autoportant : toute personne peut le compiler, l'exécuter et le comprendre sans aide extérieure.

Enseignements :

Ce projet a permis d'acquérir une compréhension approfondie de plusieurs aspects clés de la programmation bas-niveau en Java :

- La manipulation des bits en mémoire : décalages, masquage, et organisation binaire.
- La gestion de la compacité et du temps d'accès comme variables d'optimisation.
- L'impact du design logiciel sur la maintenabilité et la lisibilité du code.

Ces compétences sont transférables à de nombreux domaines (compression, indexation, traitement de données à grande échelle).

Au-delà de l'aspect technique, ce projet a permis d'appliquer concrètement les principes du Software Engineering :

- conception modulaire et évolutive,
- séparation claire des responsabilités,
- documentation et tests systématiques,
- validation expérimentale des choix de conception.

Conclusion :

Mon projet BitPacking atteint les objectifs initiaux :
il propose une bibliothèque Java capable de compresser efficacement des entiers tout en maintenant un accès direct en temps constant, le tout dans une architecture claire et extensible.

Les trois implémentations proposées — NoSplit, Split et Overflow — offrent chacune des avantages spécifiques :

- NoSplit : la simplicité et la rapidité, idéale pour les scénarios temps réel.
- Split : un bon compromis entre compacité et vitesse sur de grands volumes.
- Overflow : la flexibilité et l'efficacité mémoire pour les données hétérogènes.

Pour conclure et ouvrir mon sujet

Une piste d'amélioration importante existe et concerne la zone overflow de la méthode *OverflowBitPacker*.
Actuellement, cette zone est gérée selon un schéma NoSplit, c'est-à-dire avec des blocs fixes qui ne chevauchent pas plusieurs mots.
Ce choix a été motivé par la recherche de simplicité d'exécution et de stabilité du code.
Cependant, il limite légèrement la densité de stockage des valeurs extrêmes : chaque élément overflow occupe un espace complet, même lorsque les bits disponibles pourraient être mieux exploités.

Une évolution consisterait à passer la zone overflow en mode Split, de manière à autoriser le chevauchement de bits entre deux mots, comme dans la méthode *SplitBitPacker*.
Cela permettrait de réduire encore la taille mémoire totale, surtout dans les cas où la proportion de valeurs overflow devient significative.
Cette approche rendrait la méthode *Overflow* encore plus polyvalente, capable d'adapter automatiquement la densité de stockage selon la distribution des données.

Une telle amélioration impliquerait toutefois de revoir le calcul du flag et du payload, afin de garantir la lecture en O(1) malgré le chevauchement.
Elle représenterait un bon compromis entre la rigueur du modèle NoSplit et l'efficacité du modèle Split, prolongeant la logique d'équilibre déjà présente dans la méthode actuelle.

Voilà la fin de mon rapport et de mon projet, pour finir je tenais à louer votre approche d'enseignement par projet et de mettre l'emphase sur l'utilité qu'un tel projet peut avoir pour une entré dans le monde professionnel, je vous remercie pour ça et j'espère que mon projet a atteint les objectifs attendu.

Cordialement