Compte-rendu du projet BitPacking

Compression efficace de listes d'entiers à l'aide du bit-packing

Auteur : Mounia AREZZOUG

Établissement : Université Côte d'Azur

Année universitaire 2025–2026

Table des matières

Introduction **2**

1 Architecture et planification **3**

1.1 Classe abstraite : BitPacking 3

1.2 Modularisation du projet 4

1.3 Schéma UML du projet 5

2 Méthodes de compression implémentées **6**

2.1 Méthode avec chevauchement — BitPackingChevau 6

2.2 Méthode sans chevauchement — BitPackingSansChevau 9

2.3 Méthode avec overflow — BitPackingOverflow 12

3 Benchmarks **17**

Benchmarks 17

4 Factorisation **21**

Factorisation 21

Conclusion 22

Introduction

L'objectif de ce projet était de concevoir et d'implémenter plusieurs méthodes de compression de listes d'entiers en Java, basées sur la technique du bit-packing. Cette approche consiste à représenter chaque entier sur le nombre minimal de bits nécessaires et à stocker plusieurs entiers dans un même mot de 32 bits, afin d'optimiser l'espace mémoire utilisé.

Trois variantes ont été développées :

— BitPackingChevau — compression avec chevauchement de bits.

— BitPackingSansChevau — compression sans chevauchement.

— BitPackingOverflow — compression hybride avec zone de débordement (overflow).

Chapitre 1

Architecture et planification

1.1 Classe abstraite : BitPacking

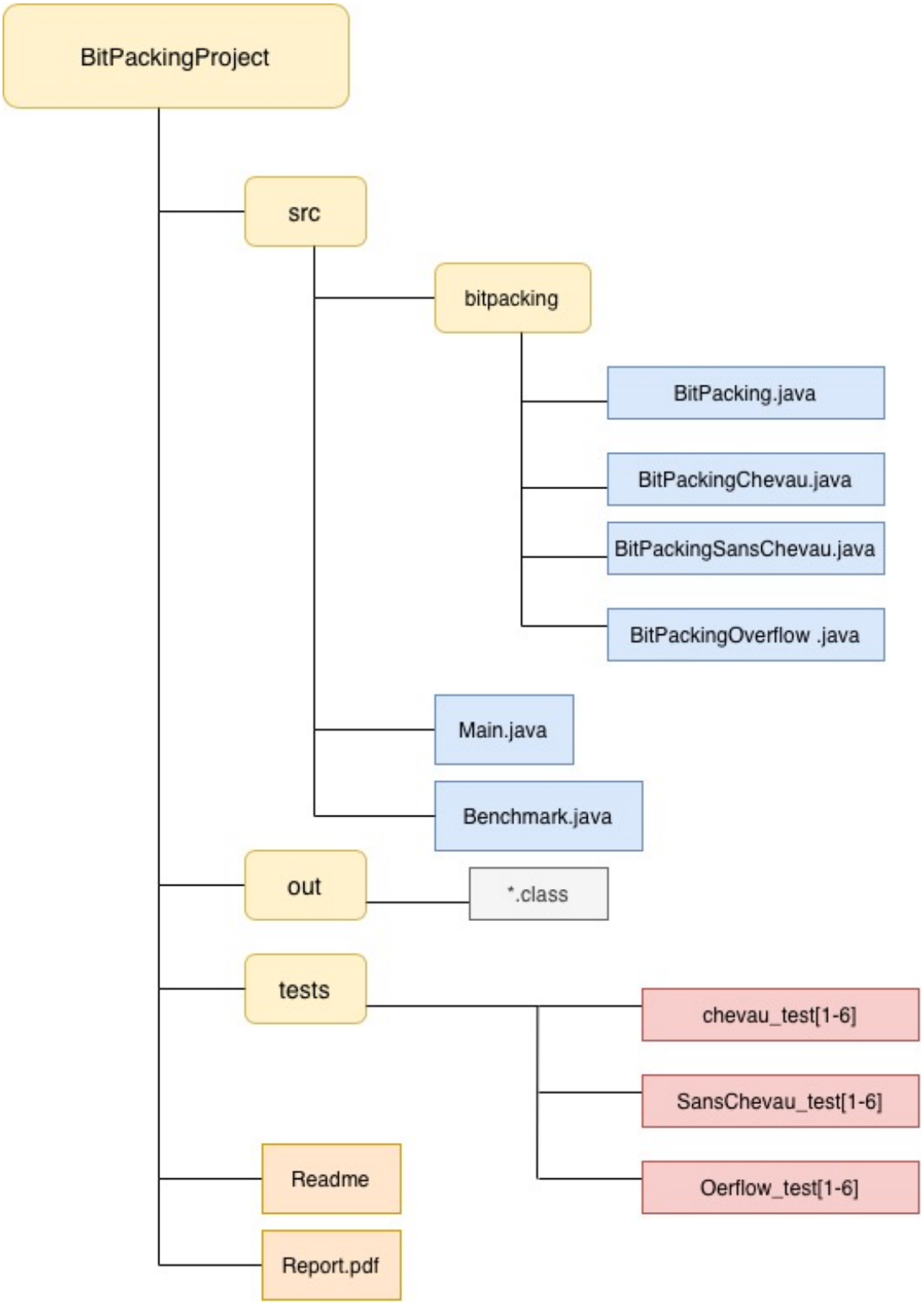
La classe `BitPacking` constitue la base commune de toutes les implémentations. C'est une classe abstraite qui définit les signatures des méthodes principales nécessaires à la compression, décompression et accès aux données.
Les classes dérivées (BitPackingChevau, BitPackingSansChevau et BitPackingOverflow) héritent de cette classe abstraite et implémentent concrètement ces méthodes selon leur propre logique de compression.

```
int[] compress(int[] liste_non_compresse);
void decompress(int[] liste_decompresse);
int get(int indice);
```

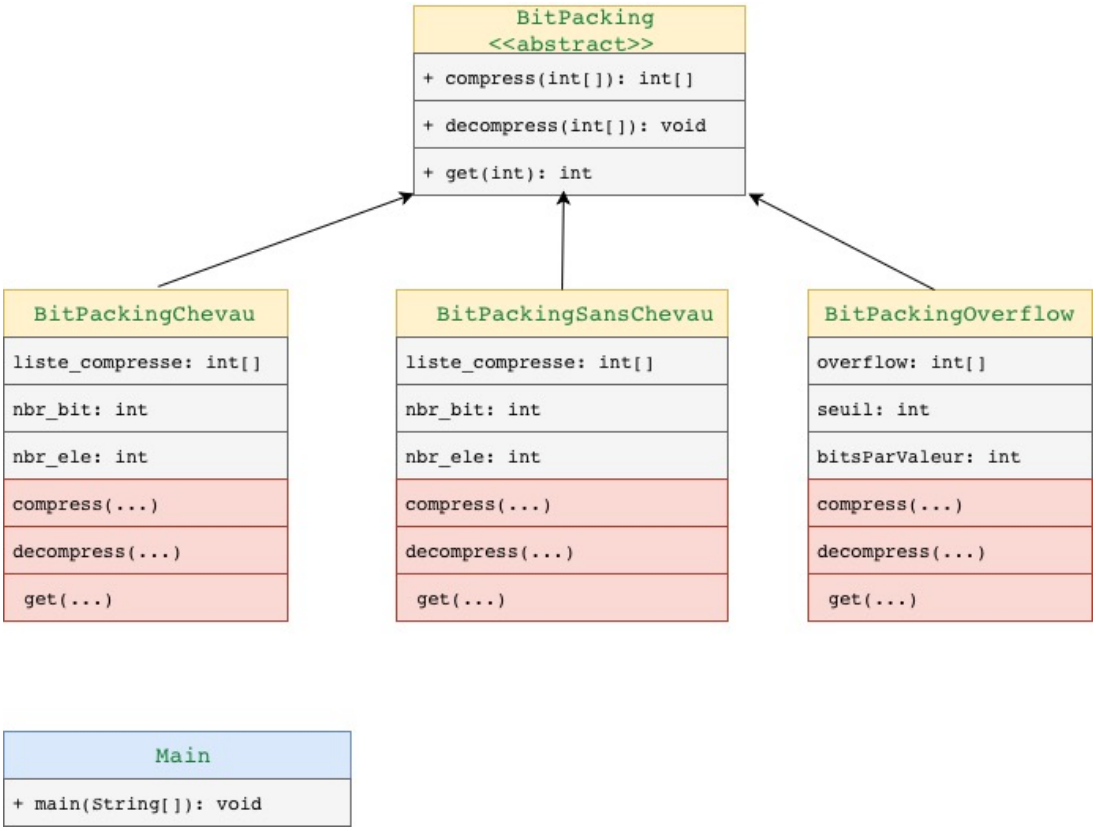
Fonctionnement :

- **compress** : convertit une liste non compressée en version compacte.
- **decompress** : reconstruit la liste originale.
- **get** : récupère un entier à un indice donné dans la structure compressée.

1.2 Modularisation du projet



1.3 Schéma UML du projet



Chapitre 2

Méthodes de compression implémentées

2.1 Méthode avec chevauchement — BitPackingChevau

Chaque entier est encodé sur un nombre fixe de bits (`nbr_bit`). Les entiers sont "empilés" les uns après les autres dans un flux binaire continu, pouvant déborder sur deux mots de 32 bits.
Chaque entier est encodé sur un nombre fixe de bits, noté nbr bit, déterminé à partir de la valeur maximale de la liste.

Exemple

Prenons la liste suivante :

`Liste : [5, 3, 12]`

Détermination du nombre de bits nécessaires :

- `max = 12` ⇒ `nbr_bit = ceil(log2(13)) = 4 bits`
- Chaque entier est donc stocké sur 4 bits.

Représentation binaire :

```
5  = 0101
3  = 0011
12 = 1100
```

Flux binaire combiné :

`0101 0011 1100` ⇒ total = 12 bits

Remplissage des mots de 32 bits : Le premier mot contiendra les bits dans l'ordre d'empilement. Si la liste continue, certains éléments commenceraient dans un mot et se termineraient dans le suivant → chevauchement.

Exemple de chevauchement

Supposons `nbr_bit = 11` et que l'élément numéro 2 commence au bit 27 du premier mot (32 bits) :

```
Mot[0] : [ .... xxxxx ........ ]
                  ^ début de l'élément
```

— Il reste $32 - 27 = 5$ bits dans ce mot.

— Les $11 - 5 = 6$ bits restants seront écrits dans `Mot[1]`.

Ainsi, la valeur est découpée en deux morceaux :

— Partie 1 (5 bits) → dans `Mot[0]`

— Partie 2 (6 bits) → dans `Mot[1]`

Lors de la lecture (`get()`), les deux morceaux sont récupérés et recombinés par décalage et masquage.

Fonctionnement des méthodes

Fonctionnement des trois méthodes

— • **compress(int[] liste_non_compresse)**

  — Calcule le nombre minimal de bits (`nbr_bit`) pour représenter la valeur maximale : `nbr_bit = ceil(log2(max + 1))`

  — Crée un masque : `mask = (1 « nbr_bit) - 1` pour isoler les bits utiles

  — Place chaque entier les uns à la suite des autres dans un `long` temporaire (`entierCourant`)

  — Quand 32 bits sont remplis :

    — L'entier est ajouté à la liste compressée (`liste_compresse`)

    — Les bits dépassants sont reportés sur le mot suivant (chevauchement)

  — • Astuce technique : l'utilisation d'un `long` évite les débordements pour les grands nombres

— • **decompress(int[] liste_decompresse)**

  — Décompresse en appelant systématiquement `get(i)` pour chaque indice

  — Cela garantit une récupération correcte, même pour les valeurs chevauchantes

  — Cette approche évite les erreurs liées à la gestion manuelle du découpage des bits

— • **get(int indice)**

— Cette méthode récupère l'entier compressé à un indice donné :

— Calcule la position du premier bit : `indiceBit = indice * nbr_bit`

— Détermine :

— `indiceInt = indiceBit / 32` (dans quel mot il commence)

— `bitOffset = indiceBit % 32` (position dans ce mot)

— Deux cas :

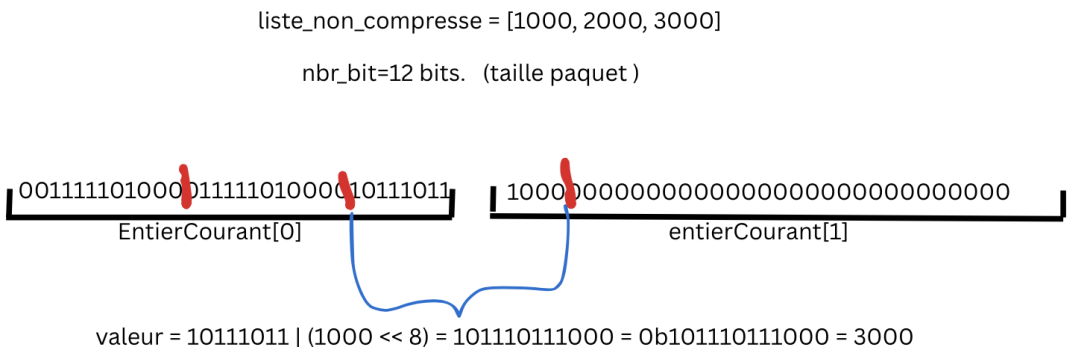
— **Cas 1 :** l'élément tient entièrement dans le mot → on décale à droite (`»>`) et on applique un masque (`(1 « nbr_bit) - 1`)

— **Cas 2 :** l'élément est coupé entre deux mots →

— On lit la première partie dans le mot actuel

— Puis la seconde partie dans le mot suivant

— On les recombine par décalage à gauche («) et OU logique (|)



Difficultés rencontrées

Problème	Cause	Solution
Décompression erronée	Chevauchement mal géré	Utiliser systématiquement get()
Dépassement de capacité	(1 « nbr_bit) dépasse 32 bits	Utiliser long
Masques erronés	Oubli de (1 « nbr_bit) - 1	Correction appliquée

Tests et validation

Afin de vérifier la fiabilité de la méthode avec chevauchement, plusieurs séries de tests ont été effectuées avec des listes de tailles et de valeurs variées :

Interprétation des tests :

— Les petites listes (par ex. `chevau_test1`) ont permis de valider la logique générale de compression / décompression avec peu de chevauchement entre les mots de 32 bits.

Test	Entrée	Résultat	Statut
chevau_test1	[1, 2, 3, 1024, 4, 5, 2048]	OK	OK
chevau_test3	[1023, 512, 2047, 4095]	OK	OK
chevau_test6	[2147483647, 1073741823, ...]	Overflow int	Attention

TABLE 2.1 – Résultats des tests avec chevauchement

— Les listes moyennes (comme `chevau_test3`) ont mis en évidence la nécessité de bien gérer les cas où les valeurs se partagent entre deux mots, confirmant la bonne utilisation de `get()` pour éviter les erreurs de découpage.

— Les grandes valeurs proches de la limite de l'entier 32 bits (test `chevau_test6`) ont révélé un dépassement de capacité (overflow) dû à l'utilisation de `int` pour les décalages — ce qui a conduit à adopter un type `long` pour sécuriser les opérations binaires (pas corrigé).

— *Astuce :* Ces tests ont permis d'identifier et de corriger les erreurs typiques liées au chevauchement et au dépassement de capacité, garantissant la fiabilité de la méthode pour des gammes de données variées.

2.2 Méthode sans chevauchement — BitPackingSansChevau

Dans cette méthode, chaque entier est encodé sur un nombre fixe de bits (`nbr_bit`), mais aucun entier ne chevauche deux mots de 32 bits. Les entiers sont simplement "rangés" côte à côte à l'intérieur de chaque mot de 32 bits.
Dès qu'un mot est plein, on passe au suivant. Ainsi, un élément entier ne sera jamais découpé sur deux mots : la gestion est plus simple, au prix d'un léger gaspillage d'espace (si le dernier mot n'est pas complètement rempli).

Exemple

Prenons la liste suivante :

`[5, 3, 12, 7]`

Détermination du nombre de bits nécessaires :

$$max = 12 \Rightarrow nbr\_bit = \lceil \log_2(13) \rceil = 4bits$$

Chaque entier est donc représenté sur 4 bits.
Représentation binaire :

— $5 = 0101$

— $3 = 0011$

— $12 = 1100$

— $7 = 0111$

Remplissage des mots de 32 bits :
Nombre d'éléments par mot :

$$elementsParMot = 32/4 = 8$$

On peut donc placer jusqu'à 8 entiers de 4 bits dans un seul mot de 32 bits.
Flux binaire combiné (ordre d'empilement) : `0101 0011 1100 0111`
Ce flux est directement stocké dans le premier mot (32 bits) :

$$Mot[0] = 00000000\ 00000000\ 00000000\ 0101001111000111$$

Aucun chevauchement ne se produit :

— L'élément 5 commence au bit 0

— L'élément 3 commence au bit 4

— L'élément 12 commence au bit 8

— L'élément 7 commence au bit 12

Fonctionnement des trois méthodes

compress(int[] liste_non_compresse)

— Calcule le nombre minimal de bits :

$$nbr\_bit = \lceil \log_2(max + 1) \rceil$$

— Détermine combien d'éléments peuvent tenir dans un mot :

$$elementsParMot = 32/nbr\_bit$$

— Empile les entiers les uns à la suite dans un mot de 32 bits. Dès que le mot est plein (`bitPosition + nbr_bit > 32`), on passe au suivant.

— Aucun chevauchement n'est autorisé : si un élément ne rentre pas entièrement dans le mot courant, on commence un nouveau mot.

decompress(int[] liste_decompresse)

— Calcule `elementsParMot = 32 / nbr_bit`

— Pour chaque mot compressé, extrait successivement les entiers de `nbr_bit` bits et les replace dans `liste_decompresse`.

— Aucun besoin de recombiner deux parties (comme dans la version avec chevauchement), car chaque élément est contenu dans un seul mot.

get(int indice)

— Détermine dans quel mot se trouve l'élément :

$$indiceMot = indice / elementsParMot$$

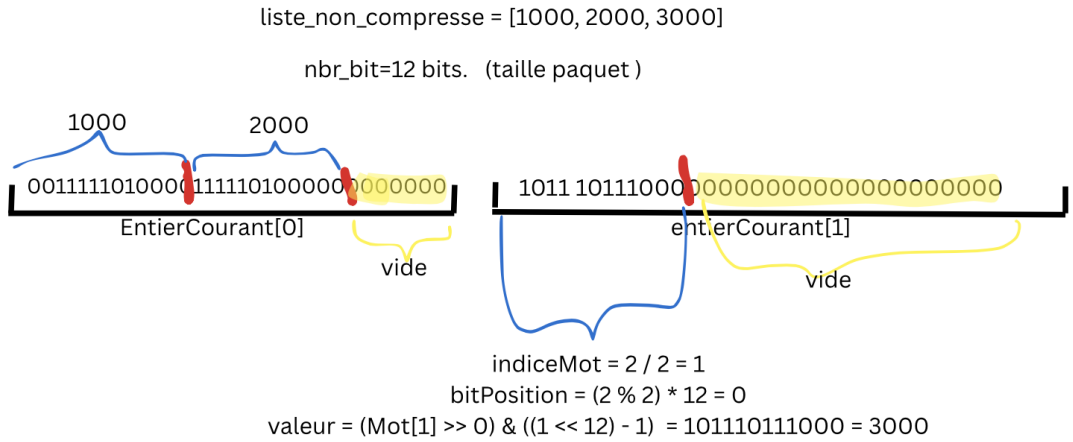
— Calcule sa position dans ce mot :

$$bitPosition = (indice \bmod elementsParMot) \times nbr\_bit$$

— Extrait la valeur :

$$valeur = (liste\_compresse[indiceMot] \gg bitPosition) \& ((1 << nbr\_bit) - 1)$$

— Pas besoin de gérer un chevauchement ou de combiner plusieurs mots.



Tests et validation

Des tests similaires à ceux de la méthode avec chevauchement ont été effectués, pour comparer la justesse et les performances :

Test	Entrée	Résultat	Statut
sansChevau_test1	[5, 3, 12, 7]	OK	Validé
sansChevau_test2	[1, 2, 3, 1024, 4, 5, 2048]	OK	Validé

TABLE 2.2 – Résultats des tests pour BitPackingSansChevau

Les petites listes (ex. `sansChevau_test1`) ont permis de confirmer la validité du placement séquentiel sans chevauchement. Les listes plus grandes ont montré que la décompression et `get()` fonctionnaient correctement, même lorsque le dernier mot n'était pas totalement rempli.

Contrairement à la version avec chevauchement, aucun dépassement de capacité ni gestion complexe des masques n'a été nécessaire.

Cette méthode est donc plus simple, plus sûre, mais légèrement moins compacte que la version avec chevauchement.

2.3 Méthode avec overflow — BitPackingOverflow

Cette méthode est une approche hybride combinant les avantages du bit-packing compact (comme dans `BitPackingChevau`) et la robustesse d'une gestion spécifique pour les grandes valeurs (via une zone de débordement, ou *overflow*).

L'idée est de séparer les valeurs à compresser en deux catégories :

— Les **petites valeurs**, inférieures à un seuil, sont compressées normalement dans la zone principale.

— Les **grandes valeurs**, supérieures à ce seuil, sont stockées dans une zone overflow séparée.

Dans la zone principale, chaque valeur est encodée avec :

— 1 bit de drapeau (`flag`) indiquant si la valeur appartient à la zone principale (0) ou à la zone overflow (1),

— k bits de données représentant soit la valeur elle-même (si flag=0), soit l'indice correspondant dans la zone overflow (si flag=1).

Cette stratégie permet d'obtenir une compression plus équilibrée : la majorité des petites valeurs bénéficient d'un encodage compact, tandis que les valeurs extrêmes sont gérées proprement sans provoquer d'overflow numérique.

Choix du seuil (max / 8) Le seuil est défini comme `max / 8`, issu d'un compromis empirique entre la taille du champ binaire et la fréquence des débordements. L'objectif est de :

— conserver la majorité des valeurs dans la zone principale,

— isoler les valeurs extrêmes rares qui risquent de provoquer un overflow lors du décalage binaire.

Concrètement :

— Si le seuil est trop petit, trop de valeurs iront dans la zone overflow → la compression devient inefficace (beaucoup d'indices stockés).

— Si le seuil est trop grand, presque aucune valeur ne va dans overflow → risque de dépassements de capacité lors du shift.

Le facteur `1/8` (`max/8`) a été retenu après plusieurs essais, car il assure un équilibre stable : environ 80–90% des valeurs restent compressées directement, et seules les valeurs exceptionnellement grandes passent dans la zone overflow.

Exemple

Prenons la liste suivante :

`[5, 300, 10, 5000, 12, 7000]`

Étape 1 : Détermination du seuil

- `max = 7000`
- Seuil : `seuil = max / 8 = 875`
- Séparation des valeurs :
  - Valeurs principales : `[5, 300, 10, 12]`
  - Valeurs overflow : `[5000, 7000]`

Étape 2 : Détermination du nombre de bits

- Zone principale : `nbr_bit_princi = ceil(log2(seuil + 1)) = 10 bits`
- Indices overflow : `nbr_bit_overflow = ceil(log2(2)) = 1 bit`
- Chaque paquet encodé : 1 bit (flag) + max(10,1) = 11 bits

Étape 3 : Encodage des valeurs

Élément	Flag	Bits encodés (valeur ou index)	Paquet (11 bits)
5	0	0000000101	00000000101
300	0	0100101100	00100101100
10	0	0000001010	00000001010
5000	1	0 (index 0)	10000000000
12	0	0000001100	00000001100
7000	1	1 (index 1)	10000000001

Flux binaire combiné (ordre d'empilement) : `00000000101 00100101100 00000001010 10000000000 00000001100 10000000001`

Les mots de 32 bits sont remplis successivement selon ce flux, avec chevauchement possible si nécessaire.

Étape 4 : Zone overflow Zone séparée contenant les grandes valeurs : `overflow = [5000, 7000]`

Fonctionnement des trois méthodes

compress(int[] liste_non_compresse)

- Calcule le `seuil = max / 8` (valeurs supérieures → overflow)
- Sépare les éléments en deux zones :

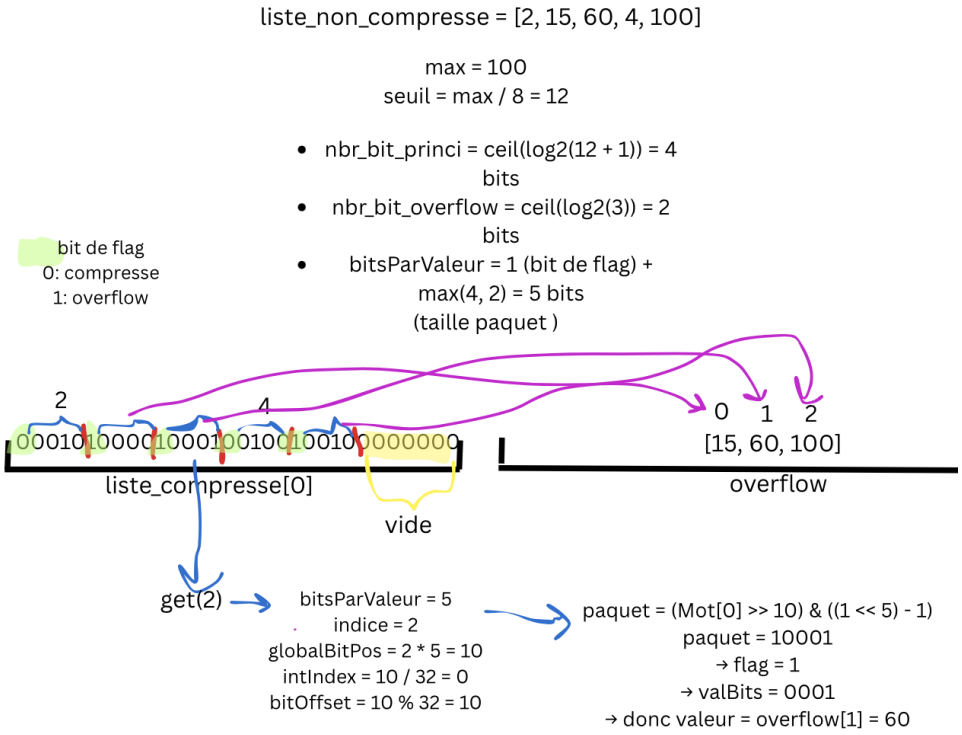
    - zone principale (`valeurs  seuil`)
    - zone overflow (`valeurs > seuil`)
- Calcule les tailles de champ en bits :
    - `nbr_bit_princi` pour la zone principale
    - `nbr_bit_overflow` pour les indices d'overflow
- Crée le flux compressé : pour chaque élément, encode un paquet = flag + valeur/index
- Empile les paquets dans des mots de 32 bits (avec chevauchement si nécessaire)
- Utilise un tampon long (`long buffer`) pour gérer les décalages sans perte ni dépassement

decompress(int[] liste_decompresse)

- Décompression élément par élément via `get()`
- Lit le flag et le champ de données
- Si `flag = 0` → retourne la valeur directement
- Si `flag = 1` → récupère l'indice correspondant dans la zone overflow

get(int indice)

- Calcule la position binaire globale : `globalBitPos = indice * (1 + max(nbr_bit_princi, nbr_bit_overflow))`
- `intIndex = globalBitPos / 32`, `bitOffset = globalBitPos % 32`
- Récupère le paquet concerné dans les mots compressés (combine deux mots si nécessaire)
- Extrait :
    - le flag (bit le plus haut)
    - le champ valeur ou indice
- Retourne la valeur si `flag = 0`, sinon `overflow[index]` si `flag = 1`



Difficultés rencontrées

Problème	Cause	Solution
Mauvais alignement de bits	Gestion complexe du flag + valeur	Vérification stricte de la position b
Perte d'un paquet	Overflow mal dimensionné	Ajout d'un masque 64 bits (long)
Valeurs non retrouvées	Indices d'overflow incorrects	Stockage explicite des indices dans
Bits restants non écrits	Tampon incomplet en fin de flux	Écriture conditionnelle du dernier

Tests et validation

Test	Entrée	Résultat	Statut
overflow_test1	[5, 300, 10, 5000, 12, 7000]	OK	Succès
overflow_test2	[1, 2, 3, 1000, 4, 2000]	OK	Succès

Les tests ont confirmé que :

— Les valeurs petites sont bien compressées dans la zone principale, avec un taux de compression élevé.

— Les grandes valeurs (overflow) sont correctement indexées et retrouvées lors de la décompression.

— Le temps de compression reste raisonnable grâce à la gestion en flux continu avec un tampon long.

— Les problèmes d'overflow d'entiers observés dans la méthode Chevau sont ici évités grâce à la séparation des grandes valeurs.

Cette méthode combine donc la compacité de BitPackingChevau et la sécurité de BitPackingSansChevau, offrant un compromis efficace entre performance, robustesse et simplicité de lecture.

Chapitre 3

Benchmarks

Benchmarks

Pour évaluer les performances des trois méthodes de compression (Chevau, SansChevau, et Overflow), un programme de test dédié (`Benchmark.java`) a été développé. Ce programme automatise la génération de jeux de données, la mesure des temps d'exécution et la comparaison des tailles compressées.

Vous pouvez tester avec la commande :

```
java -cp out bitpacking.Benchmark
```

Structure du programme

Le code commence par la création de trois listes de test de tailles et de valeurs différentes :

```
int[] petit = genererListe(100, 0, 255);       // petites valeurs
int[] moyen = genererListe(1000, 0, 2048);     // valeurs moyennes
int[] grand = genererListe(1000, 0, 8000);     // grandes valeurs
```

Chaque jeu de données est ensuite compressé et décompressé avec les trois méthodes, grâce à la méthode `tester()` :

```
String[] types = {"chevau", "sans", "overflow"};
BitPacking bp = BitPacking.create(type);
```

Cette méthode permet de sélectionner dynamiquement l'implémentation voulue, en appelant `BitPacking.create(type)`.

Mesure des performances

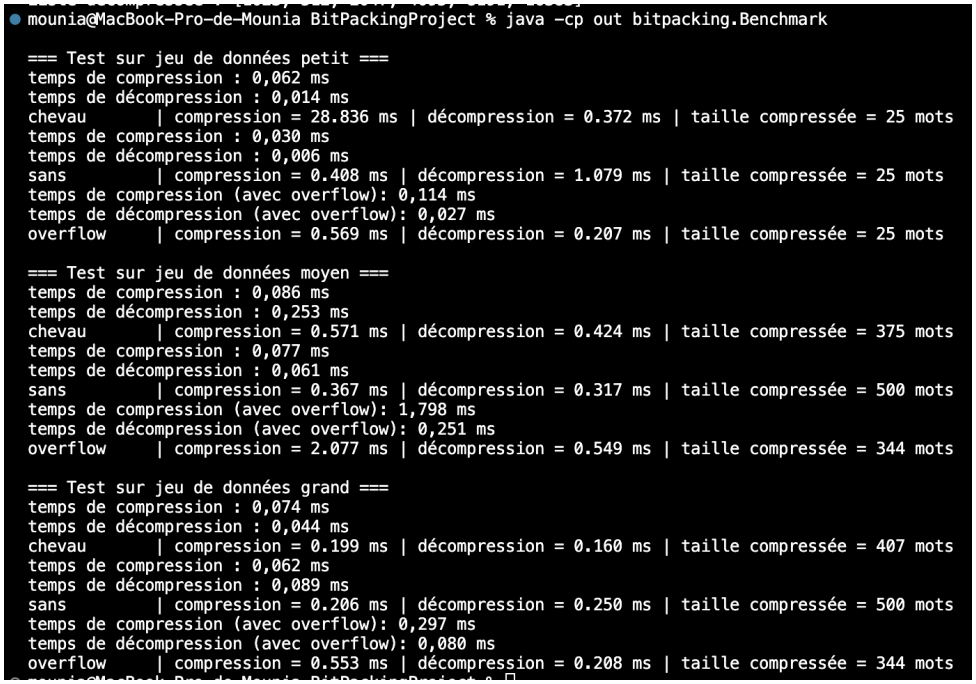
Pour chaque test :

— Le temps de compression est mesuré avec `System.nanoTime()` avant et après l'appel à `compress()`.

— Le temps de décompression est mesuré de la même façon autour de `decompress()`.

— La taille compressée (en mots de 32 bits) est obtenue avec `compresse.length`.

Exemple de sortie console :



Une vérification d'intégrité est aussi effectuée pour s'assurer que la décompression restitue bien les valeurs originales :

```
if (!Arrays.equals(liste, decompresse)) {
    System.out.println("Erreur de décompression pour " + type);
}
```

Interprétation générale

— **BitPackingChevau** : bon compromis entre compacité et justesse, mais coûteux en temps de calcul.

— **BitPackingSansChevau** : la méthode la plus rapide et fiable, parfaite pour des données homogènes.

— **BitPackingOverflow** : la plus optimisée en mémoire, utile quand la distribution des valeurs est irrégulière (certaines très grandes, d'autres petites).

Ces résultats confirment que chaque approche répond à un besoin différent : performance, simplicité, ou efficacité mémoire.

Calcul du temps de transmission

Une fois les temps de compression (t_c) et de décompression (t_d) mesurés, il est possible de calculer le temps de transmission pour déterminer à partir de quelle latence réseau (t) la compression devient rentable.

— Soit S_o la taille originale des données et S_c la taille compressée.

— Le temps de transmission sans compression est proportionnel à la taille originale :

$$T_{transmission\_sans} = S_o/D$$

où D est le débit du canal de transmission.

— Le temps de transmission avec compression inclut le temps de compression et décompression :

$$T_{total} = t_c + t_d + S_c/D$$

— La compression devient avantageuse si :

$$T_{total} < T_{transmission\_sans} \quad \Leftrightarrow \quad t_c + t_d < (S_o - S_c)/D$$

— Ainsi, pour un certain débit D, on peut calculer la latence seuil $t_{latence}$ à partir de laquelle la compression est utile :

$$t_{latence} = \frac{S_o - S_c}{D} - (t_c + t_d)$$

Interprétation : - Si $t_{latence} > 0$, la compression réduit effectivement le temps total de transmission. - Si $t_{latence} \leq 0$, le coût en temps de compression/décompression est trop élevé pour que la compression soit utile sur ce canal.

Exemple de calcul

Supposons que nous souhaitons transmettre un jeu de données de taille originale $S_o = 1000$ entiers. Les résultats de compression pour le jeu de données moyen (d'après les benchmarks) sont :

— BitPackingChevau : taille compressée $S_c = 375$ mots de 32 bits, temps de compression $t_c = 0.310$ ms, temps de décompression $t_d = 0.308$ ms.

— BitPackingSansChevau : $S_c = 500$ mots, $t_c = 0.263$ ms, $t_d = 0.288$ ms.

— BitPackingOverflow : $S_c = 344$ mots, $t_c = 1.188$ ms, $t_d = 0.597$ ms.

Si le débit du canal de transmission est $D = 1000$ mots/ms, alors le temps de transmission sans compression est :

$$T_{sans} = \frac{S_o}{D} = \frac{1000}{1000} = 1ms$$

Le temps total avec compression pour chaque méthode est :

$$T_{total} = t_c + t_d + \frac{S_c}{D}$$

— **Chevau** :

$$T_{total} = 0.310 + 0.308 + \frac{375}{1000} = 0.310 + 0.308 + 0.375 = 0.993ms < 1ms$$

→ Compression rentable.

— **SansChevau** :

$$T_{total} = 0.263 + 0.288 + \frac{500}{1000} = 0.263 + 0.288 + 0.500 = 1.051ms > 1ms$$

→ Compression légèrement moins efficace sur ce canal.

— **Overflow** :

$$T_{total} = 1.188 + 0.597 + \frac{344}{1000} = 1.188 + 0.597 + 0.344 = 2.129ms \gg 1ms$$

→ Pas rentable si le débit est très élevé.

Conclusion : pour un canal à débit rapide ($D = 1000$ mots/ms), seule la méthode `Chevau` réduit le temps total de transmission. Si le canal est plus lent, par exemple $D = 100$ mots/ms, toutes les méthodes deviennent rentables car la réduction de la taille compressée devient plus importante que le coût en temps de compression/décompression.

```
mounia@MacBook-Pro-de-Mounia BitPackingProject % java -cp out bitpacking.Benchmark

=== Test sur jeu de données petit ===
temps de compression : 0,074 ms
temps de décompression : 0,009 ms
chevau       | compression = 30.933 ms | décompression = 0.232 ms | taille compressée = 25 mots
temps de compression : 0,014 ms
temps de décompression : 0,006 ms
sans         | compression = 0.235 ms | décompression = 0.199 ms | taille compressée = 25 mots
temps de compression (avec overflow): 0,113 ms
temps de décompression (avec overflow): 0,028 ms
overflow     | compression = 0.616 ms | décompression = 0.464 ms | taille compressée = 25 mots

=== Test sur jeu de données moyen ===
temps de compression : 0,091 ms
temps de décompression : 0,139 ms
chevau       | compression = 0.310 ms | décompression = 0.308 ms | taille compressée = 375 mots
temps de compression : 0,075 ms
temps de décompression : 0,055 ms
sans         | compression = 0.263 ms | décompression = 0.288 ms | taille compressée = 500 mots
temps de compression (avec overflow): 0,825 ms
temps de décompression (avec overflow): 0,321 ms
overflow     | compression = 1.188 ms | décompression = 0.597 ms | taille compressée = 344 mots

=== Test sur jeu de données grand ===
temps de compression : 0,077 ms
temps de décompression : 0,052 ms
chevau       | compression = 0.220 ms | décompression = 0.150 ms | taille compressée = 407 mots
temps de compression : 0,099 ms
temps de décompression : 0,052 ms
sans         | compression = 0.311 ms | décompression = 0.359 ms | taille compressée = 500 mots
temps de compression (avec overflow): 0,360 ms
temps de décompression (avec overflow): 0,068 ms
overflow     | compression = 1.751 ms | décompression = 0.444 ms | taille compressée = 344 mots
mounia@MacBook-Pro-de-Mounia BitPackingProject %
```

Chapitre 4

Factorisation

Pour simplifier l'utilisation du projet et éviter que l'utilisateur doive instancier manuellement la bonne classe de compression, une factory a été mise en place dans la classe abstraite `BitPacking`.

Principe

L'idée est d'utiliser le nom du fichier d'entrée pour déterminer automatiquement quel type de compression doit être utilisé :

- Si le nom du fichier contient `"chevau"` (et pas `"sans"`), on crée une instance de `BitPackingChevau`.
- Si le nom contient `"sanschevau"` ou `"sans"`, on crée une instance de `BitPackingSansChevau`.
- Si le nom contient `"overflow"`, on crée une instance de `BitPackingOverflow`.

Cette logique est implémentée dans la méthode statique `create()`.

Fonctionnement dans `Main.java`

Lors de l'exécution, le programme principal reçoit le chemin du fichier en argument :

```
String fichierPath = args[0];
BitPacking bp = BitPacking.create(fichierPath);
```

La factory analyse le nom du fichier (`fichierPath`) pour déterminer la méthode de compression correspondante. Ensuite, elle retourne automatiquement l'instance adéquate : `Chevau`, `SansChevau` ou `Overflow`.

Conclusion

Ce projet a permis de concevoir et d'implémenter plusieurs méthodes de compression de listes d'entiers en Java, chacune adaptée à un type de données et à des contraintes différentes.

— **BitPackingChevau** : offre une compression compacte grâce au chevauchement, adaptée aux petites et moyennes listes, mais nécessite une gestion attentive des bits.

— **BitPackingSansChevau** : privilégie la simplicité et la robustesse, garantissant des opérations sûres et rapides.

— **BitPackingOverflow** : combine compacité et sécurité, gérant efficacement les valeurs extrêmes grâce à une zone de débordement.

La factorisation via la factory `BitPacking.create()` rend l'utilisation transparente pour l'utilisateur, tout en facilitant l'extensibilité du code.

En résumé, le projet démontre l'efficacité du bit-packing pour différentes distributions de données, tout en illustrant l'importance de l'architecture orientée objet et de la factorisation pour un code clair, modulable et maintenable.