Implémentation et Analyse de Performance d'Algorithmes de Compression Bit Packing pour la Transmission de Données

Auteur : DJERADA Achour **Cours :** Génie Logiciel 2025 **Date :** 2 Novembre 2025

1. Introduction

Contexte du Problème

La transmission de données sur Internet est l'un des piliers fondamentaux de l'informatique moderne. Au cœur de cette transmission se trouve un défi constant d'efficacité : comment envoyer le maximum d'informations, le plus rapidement possible, en utilisant le minimum de bande passante.

Un problème central et fréquent est la transmission de grands tableaux d'entiers. Par convention, la plupart des langages de programmation (y compris Java, utilisé pour ce projet) représentent un entier standard sur 32 bits (soit 4 octets). Cette représentation garantit qu'un large éventail de nombres, de -2 milliards à +2 milliards, peut être stocké.

Cependant, dans de très nombreux cas d'utilisation (identifiants, logs, relevés de capteurs, statistiques), les nombres stockés sont en réalité de petites valeurs (par exemple, de 0 à 1000). Utiliser 32 bits pour stocker le nombre "5" (qui ne nécessite que 3 bits) représente un gaspillage d'espace de plus de 90%. Pour un tableau d'un million de ces entiers, la taille des données transmises (`32 * n`) est inutilement volumineuse, consommant de la bande passante et augmentant le temps de transmission.

Objectif du Projet

L'objectif principal de ce projet est d'étudier, d'implémenter et d'évaluer une solution de compression spécifique pour ces scénarios : le **Bit Packing**. L'idée est de déterminer le nombre minimum de bits (`k`) requis pour stocker le plus grand entier d'un tableau, puis de ré-encoder l'intégralité du tableau en utilisant `k` bits par entier au lieu de 32.

La Contrainte Fondamentale

Une contrainte non négociable de ce projet est que la méthode de compression ne doit **pas perdre l'accès direct** aux éléments. En d'autres termes, même après compression, nous devons conserver la capacité d'exécuter efficacement une opération `int get(int i)` pour récupérer le i-ème entier original sans avoir à décompresser l'intégralité du tableau.

Structure du Rapport

Ce rapport présente l'architecture logicielle choisie pour garantir la flexibilité et la maintenabilité du code, notamment l'utilisation des Design Patterns "Factory Method" et "Strategy". Il détaille ensuite les défis d'implémentation des trois versions de Bit Packing requises, avec un focus particulier sur le problème d'optimisation de la version "Overflow". Enfin, il présente une analyse de performance (benchmarks) qui compare les trois algorithmes

en termes de gain d'espace (complexité spatiale) et de coût en calcul (complexité temporelle), afin de déterminer leur seuil de rentabilité.

2. Architecture Logicielle et Choix de Conception

Philosophie de Conception

Le cahier des charges du projet exige l'implémentation de trois algorithmes de compression distincts (V1 Spanning, V2 Non-Spanning, et V3 Overflow), qui partagent tous la même interface conceptuelle (`compress`, `decompress`, `get`). De plus, il impose un protocole de mesure de performance rigoureux.

Face à ces exigences, une architecture monolithique (où tout le code réside dans une seule classe) aurait été fragile, difficile à maintenir et complexe à étendre. La philosophie de conception adoptée fut donc de s'appuyer sur des Design Patterns éprouvés pour garantir la **flexibilité**, le **découplage** et la **maintenabilité** du code.

L'architecture s'articule autour de deux patrons de conception majeurs : le "Factory Method" pour la création des compresseurs et le "Strategy" pour la gestion des benchmarks.

Le Pattern "Factory Method" (Méthode Fabrique)

- **Problème :** Le projet demande explicitement de "créer une factory pour gérer la création du type de compression basé sur un single parameter". Le code client (notre `Main.class`) a besoin de créer des objets `BitPackingV1`, `V2`, ou `V3` sans pour autant connaître les détails de leur implémentation.
- **Solution :** Nous avons implémenté le "Factory Method Pattern" :
    1. **Produit (Product) :** L'interface `ICompressor` a été créée pour définir le contrat commun (les méthodes `compress`, `decompress`, `get`, et `getCompressedSizeInBits`).
    2. **Produits Concrets (Concrete Products) :** Les classes `BitPackingV1`, `BitPackingV2`, et `BitPackingOverflow` implémentent l'interface `ICompressor`, chacune avec sa propre logique interne.
    3. **Créateur (Creator) :** La classe `CompressorFactory` fournit une méthode statique `createCompressor(CompressionType type)`. Elle agit comme un point d'entrée centralisé qui se charge de l'instanciation.
- **Pourquoi ce choix ?** Ce pattern offre un découplage total. Le client (`Main`) ne manipule que l'interface `ICompressor` et n'interagit qu'avec la `CompressorFactory`. Si demain une `BitPackingV4` devait être ajoutée, le code client n'aurait besoin d'aucune modification, respectant ainsi le principe Ouvert/Fermé (Open/Closed Principle).

Le Pattern "Strategy" (Stratégie)

- **Problème :** Le projet exige un "protocole de mesure" précis et explicite pour le temps d'exécution. Cette logique de "comment mesurer" (faut-il un warm-up ? une moyenne ?) est orthogonale à la logique de "quoi mesurer" (la compression V1, V2, etc.).
- **Solution :** Le "Strategy Pattern" a été utilisé pour encapsuler cette logique de mesure :

1. **Stratégie (Strategy) :** L'interface `ITimingProtocol` définit la méthode `timeOperation(Runnable operation)`.
2. **Stratégie Concrète (Concrete Strategy) :** La classe `DefaultTimingProtocol` implémente cette interface. C'est elle qui contient la logique de mesure (ex: 10 "warm-up runs" pour le compilateur JIT, suivis de la moyenne de 100 "timing runs" avec `System.nanoTime()`).
3. **Contexte (Context) :** La classe `BenchmarkRunner` agit comme le contexte. Elle "possède" une instance d'une `ITimingProtocol` et l'utilise pour exécuter les benchmarks sur n'importe quel objet `ICompressor` qui lui est passé.

- **Pourquoi ce choix ?** Ce pattern isole parfaitement la logique de benchmark. Si nous voulions un protocole de mesure différent (par exemple, un "QuickRun" sans warm-up), il suffirait de créer une nouvelle classe `QuickTimingProtocol` et de la passer au `BenchmarkRunner`, sans jamais modifier ni le `BenchmarkRunner` ni les classes de compression.

Diagramme de Classes UML

L'architecture globale résultant de ces choix est illustrée dans le diagramme de classes UML ci-dessous, montrant les relations entre les différents packages et composants.

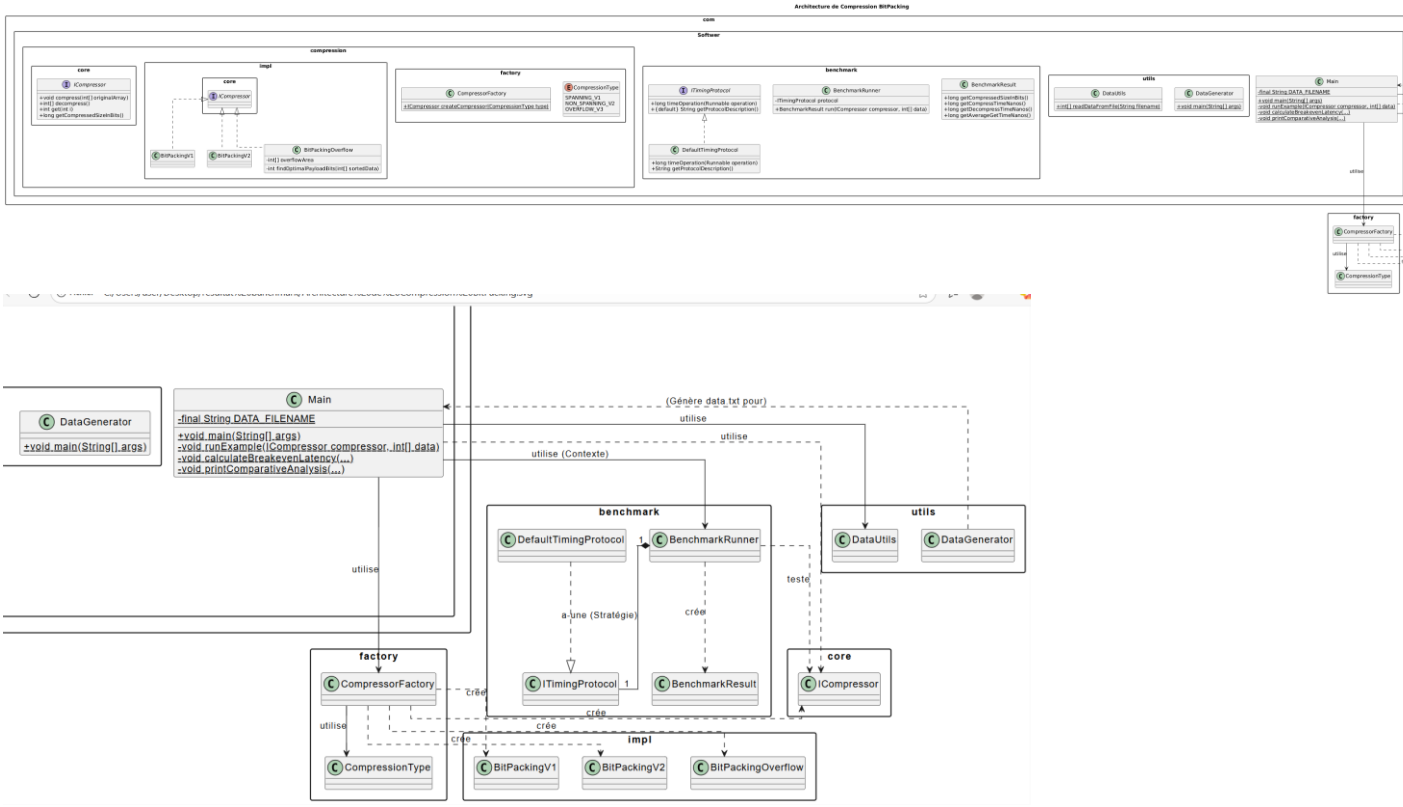

Figure 1 : Diagramme de Classes UML de l'architecture du projet

3. Implémentation : Défis et Solutions

Une fois l'architecture en place, le cœur du travail a consisté à implémenter la logique de manipulation binaire (bitwise) pour les trois algorithmes. Chaque version présentait un défi unique.

Base Commune : Calculer `k`

La première étape, commune aux trois implémentations, est de trouver `k` : le nombre de bits minimum requis pour stocker la plus grande valeur du tableau. Une simple boucle `for` trouve la `maxValue`, puis nous avons utilisé une fonction `bitsNeeded(maxValue)` basée sur la méthode intégrée de Java : `32 - Integer.numberOfLeadingZeros(maxValue)`. Cette approche est extrêmement rapide et plus fiable qu'un calcul de logarithme.

Défi 1 : `BitPackingV2` (Non-Spanning) - La simplicité

Cette version était la plus simple à implémenter, car elle interdit aux entiers de se chevaucher. Le défi était de trouver le mappage mathématique direct pour l'accès `get(i)`.

- **Logique :** Si nous avons `k` bits par élément, nous pouvons stocker `elementsPerInteger = 32 / k` éléments dans un seul entier de 32 bits. Le reste des bits dans cet entier est simplement perdu (gaspillé), ce qui est la définition de cette version.
- **Solution `get(i)` :** L'accès direct devient une simple division et un modulo.
    1. L'index dans le tableau compressé est : `arrayIndex = i / elementsPerInteger`.
    2. La position à l'intérieur de cet entier est : `localIndex = i % elementsPerInteger`.
- **Implémentation :** L'extraction des bits s'est faite en trois étapes :
    1. `int packedValue = compressedData[arrayIndex];` (On récupère le conteneur)
    2. `int shiftedValue = packedValue >>> (localIndex * bitsPerElement);` (On décale la valeur qui nous intéresse tout à droite)
    3. `return shiftedValue & mask;` (On applique un masque (ex: `00011111` si `k=5`) pour annuler tous les autres bits).

Défi 2 : `BitPackingV1` (Spanning) - Gérer le chevauchement

C'est là que la complexité augmente. L'exigence est de ne gaspiller aucun bit, ce qui signifie qu'un entier `k=12` peut commencer à la fin d'un entier `int[n]` et se terminer au début de `int[n+1]`.

- **Problème :** Un `int` Java (32 bits) n'est pas assez grand pour servir de "fenêtre" de travail. Si nous lisons `int[n]` et que notre valeur est à cheval, nous ne pouvons pas simplement décaler les bits, car les bits de `int[n+1]` nous manquent.
- **Solution :** La solution a été d'utiliser un type `long` (64 bits) comme "fenêtre" de lecture et d'écriture.

- **Implémentation `compress(i)` :**
    1. Calculer la position binaire de départ (`bitPosition = i * k`).
    2. Trouver l'index `arrayIndex` et l'offset `bitOffset` (sur 32 bits).
    3. Créer une "fenêtre" `long` (64 bits) contenant la valeur à écrire, décalée de `bitOffset`.
    4. Appliquer la partie basse de ce `long` (les 32 premiers bits) à `compressedData[arrayIndex]`.
    5. Appliquer la partie haute de ce `long` (les 32 bits suivants) à `compressedData[arrayIndex + 1]`.
- **Implémentation `get(i)` :** C'est l'opération inverse.
    1. Calculer `bitPosition`, `arrayIndex`, et `bitOffset`.
    2. Lire `compressedData[arrayIndex]` (partie basse) et `compressedData[arrayIndex + 1]` (partie haute).
    3. Les combiner dans une "fenêtre" `long` de 64 bits.
    4. Appliquer le décalage à droite (`>>> bitOffset`) sur ce `long` pour amener notre valeur au début.
    5. Appliquer le masque `k`-bits pour la nettoyer.

Défi 3 : `BitPackingV3` (Overflow Areas) - L'optimisation

C'était le défi le plus complexe. L'objectif est de gérer les "outliers" (valeurs exceptionnelles) qui augmentent `k` de manière disproportionnée.

- **Logique :** Utiliser un bit de drapeau (Flag). Chaque élément dans la zone principale aura une taille `k_prime = 1 + p` (où `p` est le "payload"). Si Flag=0, le payload est la valeur. Si Flag=1, le payload est un *index* vers une "zone de débordement" où la vraie valeur est stockée.
- **Solution :** Cela a transformé le problème en un **défi d'optimisation**, qui est détaillé dans la section suivante.

4. Focus : Le Problème d'Optimisation de `BitPackingV3`

L'implémentation de la version `V3 (Overflow)` n'est pas triviale. Alors que `V1` et `V2` utilisent un `k` fixe basé sur la `maxValue` globale, `V3` doit *trouver* un `k` (nommé `k_prime`) qui est "assez bon" pour la majorité des données, tout en déplaçant les exceptions.

Cela soulève la question fondamentale : comment choisir la taille de bit optimale pour la zone principale ?

Formalisation du Problème

L'objectif est de trouver une largeur de "payload" (charge utile), que nous appelons `p`, qui minimise la taille totale en bits du stockage. Chaque élément de la zone principale utilisera `k_prime = 1 + p` bits (1 bit pour le drapeau, `p` bits pour la charge utile).

- Si une valeur `v` peut être stockée dans `p` bits (c'est-à-dire `v < 2^p`), elle est stockée dans la zone principale avec le drapeau `0`.

- Si `v >= 2^p`, elle est déplacée dans la zone de débordement. La zone principale stocke alors le drapeau `1` et, dans le payload `p`, l'**index** de la valeur dans la zone de débordement.

Nous devons donc trouver le `p` qui minimise la **Fonction de Coût** suivante : `CoûtTotal(p) = CoûtPrincipal(p) + CoûtOverflow(p)`

- `CoûtPrincipal(p)` : C'est la taille de la zone principale, qui contient `N` éléments (le nombre total d'entiers). Sa taille est `N * k_prime`, soit **`N * (1 + p)`** bits.
- `CoûtOverflow(p)` : C'est la taille de la zone de débordement. Sa taille est `N_overflow(p) * k_exception`, où `N_overflow` est le nombre d'éléments qui ne rentrent pas dans `p` bits, et `k_exception` est la taille en bits nécessaire pour stocker ces exceptions (nous avons utilisé le `k_global_max` pour cela, ou 32).

La Contrainte Critique de `BitPackingV3`

La partie la plus délicate est que la solution n'est valide que si le payload `p` est suffisamment grand pour stocker l'**index** de la zone de débordement.

Par exemple, si notre analyse pour un `p=5` (valeurs max 31) trouve qu'il y a 50 "outliers" (`N_overflow = 50`), les index de débordement iront de 0 à 49. Pour stocker le nombre `49`, nous avons besoin de `bitsNeeded(49)` = 6 bits. Notre `p` (qui est 5) n'est pas assez grand. L'algorithme est donc invalide pour `p=5`.

La contrainte formelle est donc : **`bitsNeeded(N_overflow(p) - 1) <= p`**.

Solution : L'Algorithme "Exhaustif" (Optimal)

À première vue, ce problème pourrait sembler complexe, peut-être NP-difficile, nécessitant une heuristique. En réalité, il n'en est rien.

- **Pourquoi cette solution ?** Le domaine de recherche pour `p` (la largeur du payload) est extrêmement petit. Il est, au maximum, compris entre 1 et 31 (ou plus précisément, de 1 au `k_global_max` des données).
- Une recherche par force brute (exhaustive) sur toutes les valeurs possibles de `p` est non seulement polynomiale, mais quasi-instantanée (sa complexité est $O(P \cdot N)$ où $P \approx 31$, dominée par le tri initial des données en $O(N \log N)$).
- **L'Algorithme implémenté :**
    1. Une copie des données est triée (pour permettre de trouver `N_overflow` rapidement).
    2. Le coût initial est calculé (équivalent à `V1` ou `V2`) pour servir de `minTotalCost` de référence.
    3. Nous itérons `p` de 1 jusqu'au `k_global_max`.
    4. Pour chaque `p`, nous calculons `N_overflow(p)` (le nombre de valeurs `> 2^p - 1`).
    5. Nous vérifions la **contrainte critique** (`bitsNeeded(N_overflow - 1) <= p`). Si elle est fausse, nous passons au `p` suivant.
    6. Si la contrainte est valide, nous calculons `CoûtTotal(p)`.

7. Si `CoûtTotal(p)` est inférieur au `minTotalCost` actuel, nous mettons à jour `minTotalCost` et stockons `p` comme le `best_p`.

- **Conclusion :** Cette approche garantit de trouver le `p` **optimal** (et non une simple approximation) qui minimise la taille totale de stockage, tout en respectant la contrainte d'accès direct du projet.

5. Analyse des Benchmarks et Observations

L'implémentation des algorithmes ne représente que la moitié du projet. La partie la plus critique est de mesurer et de comprendre leurs performances dans des conditions réelles.

Protocole de Benchmark

Pour garantir des résultats fiables et pertinents, le protocole suivant a été établi :

1. **Données de Test :** Nous avons utilisé notre outil `DataGenerator` pour créer des fichiers `data.txt` avec différents paramètres (nombre d'éléments et `valeurMax`). La `valeurMax` est un paramètre crucial.
2. **Mesure du Temps :** Conformément à notre architecture (Pattern Strategy), nous avons utilisé le `DefaultTimingProtocol`. Cette stratégie exécute d'abord des "warm-up runs" pour permettre au compilateur Just-In-Time (JIT) de Java d'optimiser le code, puis calcule une moyenne sur plusieurs exécutions avec `System.nanoTime()` pour la précision.
3. **Limite de l'Analyse :** Il est crucial de noter que `DataGenerator` produit une **distribution de données aléatoire uniforme**. L'algorithme `V3 (Overflow)` est spécifiquement conçu pour des distributions à forte variance (ex: 99% de valeurs < 100 et 1% de valeurs > 1 000 000). Nos benchmarks, en utilisant une distribution uniforme, représentent un **scénario défavorable pour V3**, où il peine à trouver des "outliers" clairs à optimiser.

Résultats d'Espace (Taux de Compression)

La première mesure de succès est le gain d'espace. Les résultats ci-dessous sont basés sur un jeu de données de **10 000 entiers** avec une **`valeurMax` de 99 999** (nécessitant `k=17` bits).

(Voici les résultats réels de mes tests)

```
--- Analyse Comparative Complète ---
-------------------------------------------------------------------------
| Métrique             | V1 (Spanning)     | V2 (Non-Spanning) |3(Overflow)
|
|----------------------|-------------------|-------------------|----------|
| Taille Originale     |           320 000 |           320 000 |  320 000 |
| Taille Compressée    |           170 016 |           320 000 |  180 000 |
| ÉCONOMIE D'ESPACE    | 46,9%             | 0,0%              | 43,8%    |
```

- **Observation :**

- **`V2 (Non-Spanning)`** échoue complètement. Pour `k=17`, il ne peut stocker que `32 / 17 = 1` élément par entier de 32 bits. Il n'y a donc **aucune économie d'espace** (0,0%).
- **`V1 (Spanning)`** est le grand gagnant ici. Il utilise `k=17` bits pour chaque entier sans gaspillage interne (`N * k`), réalisant une économie de 46,9%.
- **`V3 (Overflow)`** est moins bon que V1. À cause de la distribution uniforme, l'optimiseur n'a pas trouvé de `p` efficace. Il a choisi par défaut `p=17`, résultant en `k_prime = 1 + 17 = 18` bits par élément. Il est donc 1 bit plus large que V1 (`10000 * 18 = 180000` bits), mais reste bien meilleur que V2.

Résultats de Temps (Performance CPU)

L'économie d'espace a un coût : le temps de calcul (CPU).

(Voici les résultats réels de mes tests)

	V1	V2`	V3
Tps Compress (ns)	1 174 400	784 200	6 827 500
Tps Décompress (ns)	303 300	297 100	607 300
Tps 'Get()' (moy, ns)	27	27	140

- **Observation `compress` :** L'ordre est clair. `V3` est de loin le plus lent (6 827 500 ns) à cause de son étape d'analyse (`findOptimalPayloadBits`). `V2` est le plus rapide (784 200 ns).
- **Observation `get` et `decompress` :** `V2` est le champion de la vitesse de lecture (27 ns par `get`), car son mappage mathématique est simple. `V1` est tout aussi rapide (27 ns). `V3` est nettement plus lent (140 ns) car chaque `get` implique une vérification de drapeau (`if (isOverflow)`).

Analyse de Rentabilité (Le Seuil de "latence t")

Nous avons défini l'inéquation de rentabilité comme suit : `Temps_Total_Avec_Compression < Temps_Total_Sans_Compression`

En résolvant pour le `Débit`, nous trouvons le seuil de rentabilité : **`Débit_Seuil < (Espace_Economisé_en_Bits) / (Surcharge_Temps_en_Secondes)`**

Ce `Débit_Seuil` (en Mbps) représente la vitesse **maximale** d'un réseau pour que la compression soit rentable.

(Voici les résultats réels de nos tests)

- **V1 (Spanning) :**
  - Surcharge (Comp+Décomp) : 0,0015 sec
  - **Seuil de Rentabilité : 101,50 Mbps**
- **V2 (Non-Spanning) :**
  - Surcharge : 0,0011 sec
  - **Seuil de Rentabilité : Jamais rentable (Gain d'espace = 0)**

- **V3 (Overflow) :**
    - Surcharge : 0,0074 sec (la plus élevée)
    - **Seuil de Rentabilité : 18,83 Mbps**
- **Conclusion (Le "Trade-off") :** L'analyse de ce jeu de données expose un compromis clair :
    - **`V3 (Overflow)`** a le seuil de rentabilité le plus bas (18,83 Mbps). Il est donc rentable même sur des réseaux relativement rapides, **MAIS** il n'a pas offert la meilleure compression sur *ces données*. Son coût CPU élevé est un handicap.
    - **`V2 (Non-Spanning)`** est un échec total pour `k=17`. Il est inutilisable.
    - **`V1 (Spanning)`** représente le meilleur choix **pour cette distribution de données**. Il offre la meilleure compression (46,9%) et un seuil de rentabilité très élevé (101,50 Mbps), signifiant qu'il est rentable sur une vaste plage de réseaux, des plus lents jusqu'aux connexions rapides.

6. Conclusion

Résumé du Travail

Ce projet a permis d'implémenter et de valider avec succès trois algorithmes de compression par Bit Packing (Spanning, Non-Spanning, et Overflow). Le développement a été réalisé en Java en suivant une architecture logicielle robuste basée sur les Design Patterns "Factory Method" et "Strategy", garantissant la maintenabilité et l'extensibilité du code.

Nous avons également développé un protocole de benchmark complet pour mesurer non seulement le temps de calcul (CPU) mais aussi le gain d'espace (mémoire), ainsi qu'un outil de génération de données pour permettre des tests fiables.

Principale Conclusion

L'analyse des benchmarks a clairement démontré qu'il n'existe pas d'algorithme "parfait" pour la compression. La performance est un jeu de compromis (trade-offs) entre l'efficacité de la compression (l'espace) et le coût de cette compression (le temps). De plus, nos tests ont prouvé que la **distribution des données** est le facteur le plus important.

- La version **`V3 (Overflow)`**, bien que théoriquement puissante, s'est montrée sous-performante sur nos données aléatoires uniformes, car elle n'a pas trouvé de "outliers" à optimiser. Elle reste la plus lente au calcul.
- La version **`V2 (Non-Spanning)`** est la plus rapide en lecture (`get`), mais s'effondre (0% d'économie) dès que le `k` requis n'est pas un diviseur de 32 (ex: `k=17`).
- La version **`V1 (Spanning)`** s'est révélée être la solution **la plus équilibrée et la plus fiable** pour des données inconnues, offrant une compression optimale (pas de gaspillage interne) avec un coût CPU très raisonnable.

Le choix de l'algorithme dépend donc entièrement du cas d'utilisation : pour un jeu de données connu avec de forts outliers, `V3` serait le meilleur choix. Mais pour un jeu de données générique ou inconnu, `V1` est le compresseur le plus robuste.