UNIVERSITÉ CÔTE D'AZUR

Software Engineering Project 2025

Data Compression for Faster Transmission

Rapport de projet

Implémentation d'un algorithme de Bit Packing

Auteur :

Benjamin Foulcher

Encadrant :

JC Regin

Université Côte d'Azur – Master Informatique
Année universitaire 2025–2026

Table des matières

1. Introduction

1.1. Contexte du projet

Dans le domaine de l'informatique et des communications numériques, la transmission rapide et efficace de données est devenue un enjeu majeur. Les systèmes d'échange d'informations doivent souvent envoyer de grandes quantités de données, parfois dans des environnements où la bande passante ou la latence constituent des contraintes fortes. Réduire la taille des données avant de les transmettre permet donc de gagner du temps et d'optimiser les ressources utilisées.

Ce projet, proposé dans le cadre du cours de *Software Engineering Project 2025*, porte sur un cas concret de compression de tableaux d'entiers. L'idée est d'explorer des méthodes simples mais efficaces permettant de représenter plusieurs entiers dans un espace mémoire réduit, sans perdre la capacité d'accéder directement à chaque élément. Cette approche s'inspire des techniques de compression dites *bit packing*, très utilisées dans les bases de données, les moteurs de recherche ou encore dans les algorithmes de traitement du signal.

1.2. Objectif général

L'objectif principal du projet est de concevoir et d'implémenter un système de compression d'entiers qui diminue la quantité de données à transmettre tout en conservant un accès direct et rapide à n'importe quel élément. Contrairement aux techniques de compression classiques qui nécessitent de décompresser entièrement les données avant de les utiliser, la méthode de *bit packing* permet d'accéder à une valeur précise à partir de sa position, même dans un flux compressé.

Trois versions de cette méthode ont été développées :

- une première version qui compresse sans chevauchement, en regroupant les valeurs par blocs complets de 32 bits ;
- une seconde version qui autorise le chevauchement des bits entre deux blocs pour un meilleur taux de compression ;
- une troisième version intégrant une zone de débordement (*overflow area*) pour traiter séparément les valeurs exceptionnellement grandes, afin de ne pas dégrader le taux de compression global.

En complément de l'implémentation, le projet impose de mesurer les temps d'exécution des fonctions de compression, de décompression et d'accès, et d'établir un protocole

expérimental permettant d'évaluer la rentabilité de la compression selon la latence de transmission.

1.3. Structure du rapport

Le présent rapport est structuré de la manière suivante :

- Le **chapitre 2** décrit les principes du *bit packing*, les opérations binaires utilisées et la conception logicielle du projet.
- Le **chapitre 3** présente le protocole de mesure, les résultats expérimentaux et l'étude du seuil de rentabilité.
- Le **chapitre 4** discute les observations, les limites rencontrées et les pistes d'amélioration possibles.
- Enfin, le **chapitre 5** conclut sur les apports du projet et propose quelques ouvertures pour des extensions futures.

2. Méthodologie et conception

2.1. Principe du bit packing

La compression par *bit packing* repose sur une idée simple : plutôt que de stocker chaque entier sur 32 bits par défaut, on utilise uniquement le nombre de bits réellement nécessaires pour le représenter. Si toutes les valeurs d'un tableau sont comprises entre 0 et 4095, par exemple, il suffit de 12 bits pour chacune d'elles. Regrouper ces valeurs sur un espace mémoire continu permet donc de réduire considérablement la taille totale à transmettre, tout en évitant toute perte d'information.

L'approche consiste à "empiler" les bits des entiers les uns à la suite des autres dans des blocs de 32 bits. Chaque fois qu'un bloc est rempli, on passe au suivant. Le processus inverse, la décompression, consiste à lire successivement les bons segments de bits et à les retransformer en entiers.

Deux variantes principales ont été codées :

- **sans chevauchement** : chaque valeur est entièrement contenue dans un bloc de 32 bits ;
- **avec chevauchement** : une valeur peut être partagée entre deux blocs pour améliorer le taux de compression.

2.1.1. Représentation binaire des entiers

En mémoire, un entier n'est qu'une suite de bits (0 et 1). Par exemple, le nombre 13 s'écrit `1101` en binaire. Pour un ordinateur, cela correspond à des puissances de deux :

$$13 = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$

Chaque bit représente donc une valeur selon sa position. En exploitant cette représentation, on peut manipuler directement les bits d'un entier à l'aide d'opérations logiques, ce qui rend possible l'écriture de plusieurs entiers dans une seule zone mémoire.

La représentation binaire a un avantage majeur : elle permet d'agir très finement sur les données. En décalant, masquant ou combinant certains bits, on peut coder ou décoder les valeurs sans conversion coûteuse.

2.1.2. Motivation de la compression par bits

Le stockage et la transmission d'un grand nombre d'entiers sont des opérations fréquentes, notamment dans les domaines du traitement de données, des réseaux et des jeux en ligne. Chaque entier stocké sur 32 bits représente une certaine redondance, car peu de valeurs utilisent réellement toute cette plage. Si l'on sait que les valeurs sont bornées (par exemple entre 0 et 4095), utiliser 32 bits est un gaspillage de 20 bits par élément.

La compression par bits permet :
- de réduire la taille du message à transmettre ;
- de diminuer le temps de transmission sur le réseau ;
- de limiter la mémoire utilisée, sans devoir recourir à un algorithme complexe comme ceux utilisés pour les fichiers textes ou images.

Dans le cadre de ce projet, cette méthode permet de tester différents compromis entre compacité, vitesse et facilité d'accès.

2.1.3. Avantage : accès direct à un élément compressé

Une caractéristique essentielle de ce projet est de maintenir un **accès direct** aux données compressées. Concrètement, pour retrouver la valeur d'indice i, il suffit de :

1. calculer la position de départ du i-ème entier dans le flux compressé ;
2. identifier le bloc de 32 bits concerné ;
3. extraire les k bits correspondant à la valeur ;
4. les reconstruire à l'aide de décalages et de masques binaires.

Ce principe garantit une lecture rapide, même sans décompression complète, ce qui est particulièrement utile dans les systèmes où seules certaines valeurs doivent être consultées.

En résumé, la méthode du *bit packing* combine trois avantages : une réduction notable de la taille mémoire, un traitement simple à implémenter, et la conservation d'un accès aléatoire efficace aux données.

2.2. Analyse des opérations binaires utilisées

Le *bit packing* repose sur la manipulation directe des bits pour écrire et relire les entiers à partir de blocs de 32 bits. Pour cela, plusieurs opérations binaires fondamentales sont utilisées. Elles permettent de positionner correctement chaque valeur dans la structure compressée et de la récupérer ensuite sans perte d'information.

2.2.1. Décalage à gauche et à droite

Les décalages binaires (« et ») sont des opérations qui déplacent l'ensemble des bits d'un entier vers la gauche ou vers la droite. Chaque décalage d'une position correspond à une multiplication ou une division par deux.

— `x « n` : décale les bits de `x` vers la gauche de `n` positions. Cela revient à multiplier la valeur par 2^n et à libérer des bits à droite pour insérer d'autres valeurs.
— `x » n` : décale les bits de `x` vers la droite de `n` positions. Cela permet d'extraire des informations situées dans les bits de poids faible.

Ces décalages sont indispensables pour "caler" correctement les valeurs lors de la compression. Par exemple, pour placer un entier de 12 bits à partir de la 13e position dans un bloc de 32 bits, on effectue un décalage vers la gauche de 12 bits :

```
val « 12
```

et on combine ensuite le résultat avec le bloc déjà existant.

De même, lors de la décompression, un décalage inverse (») permet d'aligner les bits avant d'appliquer un masque pour récupérer uniquement la partie utile.

2.2.2. Opérateurs logiques (ET, OU, masques)

Les opérateurs logiques bit à bit permettent de manipuler sélectivement certaines zones de bits.

— **OU binaire (`|`)** : combine deux valeurs en positionnant à 1 les bits qui sont à 1 dans l'une ou l'autre. Utilisé pour "fusionner" une nouvelle valeur dans un bloc existant.

$$0000\,1111 \mid 1111\,0000 \;=\; 1111\,1111$$

— **ET binaire (`&`)** : conserve uniquement les bits à 1 dans les deux valeurs. Sert principalement à isoler certaines parties d'un nombre selon un **masque**.

$$1011\,0101 \;\&\; 0000\,1111 \;=\; 0000\,0101$$

— **Masques binaires** : un masque est une valeur spéciale utilisée pour sélectionner des bits précis. Par exemple, pour récupérer les 12 bits de poids faible d'une valeur, on crée un masque :

```
mask = (1 « 12) - 1
```

Ce masque vaut `0000 1111 1111 1111` et permet de faire :

```
val & mask
```

afin de ne garder que la partie utile de l'entier.

Ces opérateurs sont très rapides car ils sont directement pris en charge par le processeur, ce qui rend cette méthode de compression efficace même sur de grands volumes de données.

2.2.3. Illustration schématique

Le principe peut se représenter schématiquement ainsi :

Bloc 1 (32 bits)	**Bloc 2 (32 bits)**	**Bloc 3 (32 bits)**	**...**
`[0-11]`	`[12-23]`	`[24-31 + 0-3]`	`[4-15]`

Chaque cellule représente un segment de bits correspondant à un entier. Lorsque les entiers ne tombent pas "juste" sur les bornes de 32 bits, une partie de la valeur déborde sur le bloc suivant. Cette organisation demande une bonne gestion des décalages et des masques pour garantir que les bits s'empilent et se récupèrent dans le bon ordre.

En résumé, l'utilisation des décalages et des opérateurs logiques permet de manipuler directement les bits sans conversion, ce qui rend la compression rapide, contrôlable et entièrement réversible.

2.3. Structure logicielle du projet

L'implémentation du projet suit une organisation modulaire inspirée de la logique objet. Chaque fonctionnalité principale est isolée dans un fichier distinct pour faciliter la maintenance, la lecture et les tests. Cette séparation permet aussi d'ajouter ou de modifier une version de compression sans impacter les autres.

2.3.1. Organisation des fichiers

Le projet est structuré de la manière suivante :

```
bitpacking/
|-- base.py               # Classe de base commune à toutes les variantes
|-- bitpacking_v1.py       # Version 1 : compression alignée
|-- bitpacking_v2.py       # Version 2 : compression avec chevauchement
|-- bitpacking_overflow.py # Version 3 : avec zone de débordement
|-- factory.py            # Sélection dynamique de la version
\-- benchmark/
\-- test.py           # Mesures de performance
```

Cette structure reflète les trois versions demandées dans le sujet. Le fichier `base.py` contient la classe mère `BitPacking`, qui définit les éléments communs à toutes les implémentations : gestion du nombre de bits, stockage des blocs et interface standard (`compress()`, `decompress()`, `get()`). Les classes filles (`BitPackerV1`, `BitPackerV2`, `BitPackerOverflo` redéfinissent la logique spécifique à chaque méthode.

2.3.2. Classes principales

- **BitPacking (classe mère)** Fournit la structure de base : initialisation du nombre de bits par valeur, stockage de la taille du tableau et gestion des blocs compressés.
- **BitPackerV1** Implémente la version la plus simple, où chaque entier est inséré séquentiellement dans un bloc de 32 bits sans chevauchement. C'est la version la plus rapide à exécuter, mais elle n'exploite pas toujours toute la capacité des blocs.

— **BitPackerV2** Permet le chevauchement entre deux blocs. Lorsqu'une valeur dépasse la limite d'un bloc, la partie restante est stockée dans le bloc suivant. Cette méthode demande plus d'opérations binaires, mais offre un meilleur taux de compression.
— **BitPackerOverflow** Introduit une zone de débordement pour les valeurs exceptionnelles. Si une valeur nécessite plus de bits que la taille moyenne (k), elle est remplacée par un pointeur vers la zone de débordement, où la vraie valeur est stockée intégralement. Cette stratégie évite de pénaliser l'ensemble du tableau à cause de quelques valeurs extrêmes.
— **Factory** La classe ou fonction `make_bitpacker()` centralise la création des objets. Elle reçoit un paramètre indiquant la version souhaitée (`"1"`, `"2"` ou `"overflow"`) et retourne automatiquement l'instance correspondante. Cette approche rend le code plus propre et simplifie les tests comparatifs.
— **Benchmark** Contient le code de mesure des temps d'exécution. Il génère des tableaux d'entiers aléatoires, exécute chaque méthode, mesure le temps moyen de compression, décompression et accès, puis calcule le ratio de compression.

2.3.3. Paramètres et variables importantes

Plusieurs paramètres influencent directement la compression :

— **k** : nombre de bits utilisés pour représenter un entier. Il est choisi selon la valeur maximale du tableau à compresser :

$$k = \lceil \log_2(\max(array) + 1) \rceil$$

Plus k est petit, meilleur est le taux de compression.
— **mask** : masque binaire utilisé pour isoler les k bits d'une valeur. Il se définit par `mask = (1 « k) - 1`. Ce masque est utilisé lors de la compression et de la décompression pour extraire la partie utile des entiers.
— **shift** : position actuelle du prochain bit à écrire dans le bloc de 32 bits. Lorsqu'il atteint ou dépasse 32, un nouveau bloc est créé.
— **data** : liste contenant les blocs compressés (chaque bloc étant un entier de 32 bits). Cette structure représente le flux compressé final.
— **overflow_limit** : seuil utilisé dans la version avec débordement pour décider si une valeur doit être envoyée dans la zone spéciale.

En résumé, l'organisation du code suit une logique claire : une classe abstraite commune, trois variantes concrètes, une fabrique pour leur création, et un module séparé pour les tests. Cette architecture simple mais structurée facilite à la fois la compréhension du projet et l'évaluation des performances.

2.4. Implémentation des trois approches

Les trois approches implémentées suivent le même principe général : compresser un tableau d'entiers en représentant chaque valeur sur un nombre réduit de bits. Elles se distinguent

principalement par la manière dont elles organisent ces bits dans les blocs de 32 bits et par la façon dont elles gèrent les cas particuliers. Chaque version a été pensée pour explorer un compromis différent entre simplicité, performance et efficacité de compression.

2.4.1. Version 1 : compression alignée (BitPackerV1)

La première approche, appelée **BitPackerV1**, est la plus simple et la plus directe. Elle écrit les entiers les uns à la suite des autres dans des blocs de 32 bits, sans jamais chevaucher deux blocs. Lorsqu'un bloc ne dispose plus de suffisamment de place pour contenir entièrement un entier, il est complété avec des zéros et un nouveau bloc est créé pour la suite.

Cette méthode est très intuitive et rapide à exécuter, car elle évite toute gestion complexe de débordement. Chaque entier commence toujours à une position bien définie à l'intérieur d'un bloc. L'inconvénient principal est qu'une partie des bits peut rester inutilisée à la fin de certains blocs, ce qui réduit légèrement le taux de compression obtenu.

Cette version sert de base de référence : elle permet de mesurer les gains réels apportés par les variantes plus optimisées.

2.4.2. Version 2 : compression chevauchante (BitPackerV2)

La deuxième approche, **BitPackerV2**, améliore la précédente en autorisant le chevauchement des valeurs entre deux blocs successifs. Lorsqu'un entier ne tient pas entièrement dans l'espace restant du bloc courant, la partie excédentaire est écrite au début du bloc suivant. Ce fonctionnement garantit une utilisation maximale de chaque bit disponible.

Le principe nécessite toutefois une gestion plus fine des décalages et des masques. À chaque ajout, le programme doit :

1. écrire la portion de la valeur qui rentre dans le bloc courant ;
2. stocker la partie restante dans le bloc suivant, à la bonne position ;
3. conserver la cohérence des indices pour la lecture et la décompression.

Cette méthode augmente légèrement le temps de calcul, mais elle améliore sensiblement le taux de compression, car aucun espace n'est perdu entre les entiers. C'est un bon compromis entre performance et efficacité mémoire.

2.4.3. Version 3 : gestion des débordements (BitPackerOverflow)

La troisième version, **BitPackerOverflow**, introduit une idée supplémentaire : la gestion d'une *zone de débordement* (overflow area). Cette zone sert à stocker séparément les valeurs exceptionnelles qui nécessitent plus de bits que la moyenne du tableau.

En pratique, la compression principale utilise un nombre de bits k' déterminé par la majorité des valeurs. Lorsqu'un entier dépasse cette limite, il n'est pas écrit directement dans le flux compressé : on insère à la place un code spécial composé d'un bit d'indicateur

(par exemple un 1 en tête) suivi d'un identifiant pointant vers la position réelle de la valeur dans la zone de débordement.

Exemple : si la majorité des entiers se codent sur 3 bits, mais qu'un petit nombre d'entre eux nécessite 11 bits, le flux compressé pourra ressembler à ceci :

0-1, 0-2, 1-0, 0-3, 0-4, 1-1

où le premier bit indique s'il s'agit d'une valeur classique (0) ou d'un pointeur vers la zone de débordement (1).

Cette approche permet d'éviter qu'une seule valeur très grande ne fasse grimper le nombre de bits nécessaires pour tout le tableau. Elle améliore donc le taux de compression sur les données hétérogènes, au prix d'une légère complexité supplémentaire dans la gestion des indices.

En résumé :
- **BitPackerV1** privilégie la simplicité et la vitesse.
- **BitPackerV2** optimise l'espace en utilisant chaque bit libre.
- **BitPackerOverflow** gère les valeurs anormales sans pénaliser les autres.

Ces trois variantes permettent de comparer concrètement les différents compromis possibles entre rapidité d'exécution et efficacité de compression.

2.5. Fonctions principales du code

L'implémentation repose sur trois fonctions essentielles : `compress()`, `decompress()` et `get()`. Elles constituent le cœur du mécanisme de compression et de décompression. Une structure complémentaire, appelée `factory`, permet de créer dynamiquement la version de compresseur adaptée au besoin.

2.5.1. `compress()`

La fonction `compress()` reçoit en entrée un tableau d'entiers et construit une version compressée sous forme d'une liste de blocs de 32 bits. Pour chaque entier, la fonction calcule le nombre de bits nécessaires (`k`), applique un masque pour isoler cette portion, puis insère la valeur dans le bloc courant en effectuant les décalages appropriés.

Lorsqu'un bloc atteint 32 bits, il est ajouté à la liste `data`, et un nouveau bloc est ouvert. Dans la version chevauchante (`V2`), une partie de la valeur peut être répartie sur deux blocs. Dans la version avec débordement (`Overflow`), la fonction vérifie si la valeur dépasse le seuil `overflow_limit` et la redirige vers la zone de débordement si nécessaire.

En résumé, cette fonction traduit le tableau original en une séquence binaire compacte tout en mémorisant la structure nécessaire pour le décodage.

2.5.2. `decompress()`

La fonction `decompress()` reconstruit le tableau initial à partir de la séquence compressée. Elle lit chaque bloc de 32 bits, en extrait les valeurs successives selon le nombre de bits `k`, puis recompose les entiers d'origine grâce à des opérations inverses à celles utilisées pour la compression.

Dans le cas de la version chevauchante, la fonction gère automatiquement les valeurs "coupées" sur deux blocs en combinant leurs parties manquantes à l'aide de décalages et d'opérations logiques. Pour la version `Overflow`, elle vérifie si une valeur correspond à un pointeur vers la zone de débordement et remplace alors ce pointeur par la valeur réelle.

Cette étape permet de vérifier la réversibilité complète du processus : la séquence compressée doit pouvoir être entièrement restituée sans perte.

2.5.3. `get()`

La fonction `get(i)` est un des points essentiels du projet : elle permet de récupérer directement la $i^{\text{ème}}$ valeur compressée sans décompresser le tableau complet.

Pour cela, la fonction calcule :

1. la position exacte du premier bit correspondant à l'élément i ;
2. le bloc de 32 bits dans lequel ce bit se situe ;
3. la portion du bloc à extraire (et éventuellement celle du bloc suivant en cas de chevauchement) ;
4. la valeur finale après application du masque et des décalages nécessaires.

Cette méthode assure un accès aléatoire rapide, ce qui distingue cette approche de la plupart des techniques de compression traditionnelles. Le coût d'accès reste constant, quel que soit le nombre d'éléments du tableau.

2.5.4. Structure de la `factory`

Afin de simplifier la gestion des trois versions de compression, une structure de type *factory* a été mise en place. Elle permet de créer dynamiquement la bonne classe de compresseur selon le paramètre choisi par l'utilisateur. L'avantage de cette approche est d'éviter les conditions complexes dans le code principal : la sélection du mode est centralisée.

Exemple simplifié de logique :

```
def make_bitpacker(mode, bits, overflow_limit=None):
if mode == "simple":
return BitPackerV1(bits)
elif mode == "overlap":
return BitPackerV2(bits)
elif mode == "overflow":
return BitPackerOverflow(bits, overflow_limit)
```

Cette structure rend le programme évolutif : une nouvelle méthode de compression pourrait être ajoutée sans modifier le reste du code.

2.6. Choix d'implémentation et contraintes

Les choix effectués lors du développement visent à équilibrer la clarté du code, la rapidité d'exécution et la fidélité aux contraintes du sujet.

2.6.1. Accès direct et compromis mémoire

Le principal objectif était de conserver la possibilité d'un accès direct à chaque valeur compressée. Ce choix implique de maintenir une structure logique claire dans le flux compressé, même si cela réduit légèrement le taux de compression. Les approches à chevauchement et débordement offrent de meilleures performances en taille, mais demandent davantage de calculs et donc un temps de compression plus élevé.

Ce compromis entre mémoire et temps d'accès est central dans la conception du projet : selon les cas d'usage, on peut privilégier la vitesse (version 1) ou la compacité (version 2 ou overflow).

2.6.2. Limites et simplifications

Plusieurs simplifications ont été assumées pour concentrer le projet sur son objectif principal :

- la taille des blocs est fixée à 32 bits, correspondant à un entier standard en Python ;
- la taille du nombre de bits (`k`) est déterminée à partir de la valeur maximale du tableau et reste constante pour tout le tableau ;
- aucune gestion d'erreur spécifique n'a été ajoutée pour les cas extrêmes (tableau vide, valeurs négatives, dépassements supérieurs à 32 bits).

Ces choix facilitent la compréhension du code et garantissent une implémentation fonctionnelle en un temps limité, conformément au cadre du projet.

2.6.3. Gestion des entiers positifs uniquement

Le sujet précisait que la prise en charge des entiers négatifs relevait du bonus. Dans cette version, seules les valeurs positives sont supportées, ce qui simplifie le calcul du nombre de bits nécessaires et évite d'avoir à gérer la représentation en complément à deux.

Toutefois, une extension possible serait d'ajouter un bit de signe, ou d'utiliser un encodage décalé (offset) pour convertir les entiers négatifs en valeurs positives avant la compression. Cela permettrait de conserver la compatibilité avec les méthodes existantes sans en modifier la logique de base.

3. Protocole expérimental et benchmarks

3.1. Mise en place du protocole de mesure

L'objectif de cette phase est d'évaluer les performances des trois approches de compression en termes de temps d'exécution et de gain en taille mémoire. Pour obtenir des résultats représentatifs, il a été nécessaire de concevoir un protocole expérimental simple mais rigoureux, permettant de comparer les méthodes dans les mêmes conditions.

3.1.1. Principe de chronométrage avec `time.perf_counter()`

Pour mesurer précisément le temps d'exécution des fonctions, la bibliothèque standard `time` de Python a été utilisée, et plus précisément la fonction `time.perf_counter()`. Cette fonction est adaptée aux mesures de performance car elle offre une résolution élevée (de l'ordre de la microseconde) et inclut le temps CPU et le temps d'attente du système d'exploitation.

Le principe est le suivant :

```
start = time.perf_counter()
# appel de la fonction a mesurer
end = time.perf_counter()
duration = end - start
```

Cette mesure est appliquée à trois opérations principales :

- `compress()` — temps nécessaire à la construction du flux compressé ;
- `decompress()` — temps de reconstruction du tableau initial ;
- `get()` — temps moyen d'accès à un élément situé au milieu du tableau.

Pour garantir la fiabilité des résultats, chaque mesure est effectuée plusieurs fois sur les mêmes données, puis moyennée afin de réduire l'impact d'éventuelles fluctuations dues à la charge du système.

3.1.2. Structure du script de test

Les tests de performance sont regroupés dans un fichier dédié, placé dans le répertoire `benchmark/`. Ce script automatise l'exécution des trois versions du compresseur sur des tableaux de tailles croissantes et affiche les résultats sous forme lisible.

Extrait simplifié du code de test :

```
for n in [1000, 5000, 10000, 50000]:
arr = [random.randint(0, 4095) for _ in range(n)]
for mode in ["1", "2", "overflow"]:
bp = make_bitpacker(mode, bits=12, overflow_limit=8)
t_comp, _ = measure_time(bp.compress, arr)
t_decomp, _ = measure_time(bp.decompress)
t_get, _ = measure_time(bp.get, n // 2)
```

Les résultats sont ensuite affichés sous forme de tableau, indiquant :

— le mode de compression testé ;
— les temps moyens de compression, décompression et accès ;
— le ratio de compression obtenu (taille compressée / taille initiale).

Cette organisation permet de lancer l'ensemble des benchmarks d'un seul appel, et de comparer facilement les performances relatives des trois méthodes.

3.1.3. Justification de l'échantillon de tailles testées

Les tailles de tableaux choisies (1 000, 5 000, 10 000 et 50 000 entiers) visent à représenter plusieurs ordres de grandeur sans allonger excessivement les temps d'exécution. Elles permettent d'observer à la fois le comportement des algorithmes sur des petits volumes (où le surcoût de la compression peut dominer) et sur des volumes plus importants (où le gain en transmission devient significatif).

Ces valeurs ont également été sélectionnées de façon à rester manipulables dans un environnement standard, sans nécessiter d'optimisations matérielles spécifiques. Le but n'est pas d'atteindre des performances absolues, mais d'obtenir une comparaison cohérente entre les trois méthodes sur des conditions identiques.

Ainsi, ce protocole expérimental fournit une base fiable pour analyser les compromis entre vitesse, taux de compression et accessibilité directe aux données.

3.2. Présentation des résultats

Les mesures ont été réalisées sur un poste personnel équipé d'un processeur Intel Core i7 et de 16 Go de mémoire vive, en environnement Python 3.11. Chaque test a été répété plusieurs fois pour obtenir des résultats moyens représentatifs, en limitant l'influence de la charge du système. Les trois méthodes de compression (`simple`, `overlap` et `overflow`) ont été évaluées sur des tableaux de 1 000, 5 000, 10 000 et 50 000 entiers.

3.2.1. Tableaux de mesure

Le tableau ci-dessous présente une synthèse des résultats observés. Les temps de compression et de décompression sont exprimés en millisecondes (ms), tandis que le temps d'accès direct via `get()` est exprimé en microsecondes (µs).

Mode	Taille (n)	Compression (ms)	Décompression (ms)	Accès (µs)
simple	1 000	0.22	0.33	1.5
overlap	1 000	0.22	0.35	324.0
overflow	1 000	0.50	0.56	1.6
simple	5 000	0.70	0.98	1.9
overlap	5 000	1.82	1.28	1257.2
overflow	5 000	2.25	2.08	2.4
simple	10 000	1.41	1.99	1.7
overlap	10 000	1.54	2.63	2595.8
overflow	10 000	5.41	4.03	1.7
simple	50 000	7.18	10.38	2.9
overlap	50 000	7.12	12.62	13677.3
overflow	50 000	24.37	20.79	4.6

TABLE 3.1 – Résultats du benchmark sur différentes tailles de tableaux.

Le ratio de compression reste constant à 0.375 pour l'ensemble des tests, ce qui est cohérent avec l'utilisation de 12 bits par valeur (soit $12/32 = 0.375$). Cela confirme la stabilité du mécanisme de compression, indépendamment du volume de données traité.

3.2.2. Analyse comparative des trois versions

L'analyse des résultats met en évidence des comportements distincts entre les trois modes :

- La **version simple** est la plus rapide et la plus régulière. Son temps de compression et de décompression croît linéairement avec la taille du tableau, tout en maintenant un temps d'accès direct très faible (autour de 2 µs). Elle offre un bon équilibre entre performance et simplicité.
- La **version chevauchante (overlap)** montre des temps de compression similaires à la version simple, mais un temps d'accès anormalement élevé. Ce comportement s'explique par la complexité des décalages binaires à travers deux blocs successifs : la fonction `get()` doit combiner des portions de bits réparties sur plusieurs entiers, ce qui multiplie les opérations nécessaires. Ce surcoût est particulièrement visible sur les grands tableaux (jusqu'à 13 ms pour 50 000 éléments).
- La **version avec débordement (overflow)** est logiquement la plus lente sur la compression, car elle doit vérifier chaque valeur et gérer une zone mémoire séparée. Cependant, son temps d'accès reste très proche de celui de la version simple, ce qui montre que la gestion des pointeurs vers la zone de débordement n'impacte pas significativement la lecture.

Ainsi, la méthode la plus efficace dépend du contexte : - pour un usage où la rapidité d'accès prime, la version simple est la plus adaptée ; - pour des données hétérogènes contenant des valeurs extrêmes, la version overflow devient préférable ; - la version overlap n'est intéressante que dans les cas où le gain de place justifie la complexité supplémentaire.

3.2.3. Temps de compression, décompression et accès

Les mesures confirment que le temps de compression et de décompression évolue de manière linéaire avec la taille du tableau ($O(n)$). Les temps restent faibles pour toutes les versions : même sur 50 000 éléments, la compression s'effectue en moins de 25 ms.

L'accès direct via `get()` est globalement constant et rapide pour les versions simple et overflow, ce qui valide l'un des objectifs principaux du projet : *pouvoir accéder à un élément compressé sans décompresser l'ensemble du flux.*

Les écarts mesurés entre les trois méthodes proviennent donc avant tout du coût des opérations binaires spécifiques à chaque implémentation, et non d'un problème d'algorithme global. Globalement, la méthode du *bit packing* démontre une excellente efficacité pour la transmission et le stockage d'entiers, avec une réduction de taille constante et un temps de traitement négligeable.

3.3. Étude du seuil de rentabilité

L'un des objectifs du projet était de déterminer dans quelles conditions la compression devient réellement avantageuse. En effet, compresser et décompresser un tableau demande du temps de calcul ; il est donc pertinent de savoir à partir de quel point le gain sur le temps de transmission compense ce coût supplémentaire.

3.3.1. Formule du seuil

On peut modéliser le temps total nécessaire pour envoyer un tableau d'entiers avec ou sans compression à l'aide de la relation suivante :

$$t_{\text{comp}} + t_{\text{decomp}} + r \cdot t_{\text{transmit}} < t_{\text{transmit}}$$

où :

- t_{comp} est le temps de compression du tableau ;
- t_{decomp} est le temps de décompression côté réception ;
- t_{transmit} est le temps de transmission d'un tableau non compressé ;
- r est le ratio de compression, c'est-à-dire la proportion de la taille compressée sur la taille initiale.

La compression devient rentable lorsque la somme du temps de traitement (compression + décompression) est inférieure au temps gagné sur la transmission :

$$t_{\text{gain}} = (1 - r) \cdot t_{\text{transmit}} - (t_{\text{comp}} + t_{\text{decomp}}) > 0$$

Dans ce cas, le système transmet plus vite en compressant qu'en envoyant directement les données brutes.

3.3.2. Interprétation des résultats

En utilisant les mesures obtenues, on observe que les temps de compression et de décompression sont de l'ordre de quelques millisecondes au maximum, tandis que le ratio de compression reste constant à 0.375. Cela signifie que les données compressées représentent environ 37,5 % de la taille originale.

Ainsi, la compression devient avantageuse dès que le temps de transmission du tableau initial dépasse quelques millisecondes. Prenons un exemple : si un tableau non compressé met 20 ms à être transmis, alors le même tableau compressé (avec $r = 0.375$) ne nécessite que :

$$t_{\text{total}} = t_{\text{comp}} + t_{\text{decomp}} + 0.375 \times 20 \approx 2 + 7.5 = 9.5\,\text{ms}$$

soit un gain d'environ 10 ms par transfert.

Ce calcul montre que même pour des volumes modestes, la compression par bit packing permet un gain tangible dès que la latence réseau dépasse quelques millisecondes. Plus la taille des données augmente ou plus la bande passante est limitée, plus l'intérêt de la compression devient évident.

4. Discussion et analyse critique

4.1. Comparaison entre les versions

Les trois versions développées permettent de mieux comprendre les compromis possibles entre simplicité, efficacité et flexibilité dans la compression par bits.

La **version simple** se distingue par sa rapidité et sa fiabilité. Elle est parfaitement adaptée aux cas où la bande passante n'est pas le principal facteur limitant. Son fonctionnement purement séquentiel en fait une solution robuste, simple à maintenir et très prévisible en termes de temps d'exécution.

La **version chevauchante** (`overlap`) cherche à optimiser l'espace mémoire utilisé en remplissant intégralement les blocs de 32 bits. Cette stratégie améliore légèrement le taux de compression théorique, mais au prix d'un accès beaucoup plus lent, surtout pour la fonction `get()`. Cette différence montre que l'optimisation mémoire peut parfois se faire au détriment de la performance pratique.

Enfin, la **version avec débordement** (`overflow`) offre un compromis intéressant pour les ensembles de données très hétérogènes. Elle permet de ne pas pénaliser tout le tableau lorsqu'une petite minorité de valeurs dépasse le nombre de bits habituel. Bien qu'un peu plus lente à compresser, elle conserve un accès rapide et préserve un bon ratio global.

4.2. Limites observées

Malgré les résultats positifs, plusieurs limites ont été identifiées au cours du développement :

- Les opérations binaires deviennent plus complexes dès qu'une valeur s'étend sur plusieurs blocs, ce qui augmente le risque d'erreurs de manipulation et le temps de calcul.
- Le code actuel ne prend pas encore en compte les entiers négatifs, ce qui limite son usage à des données strictement positives.
- Le protocole expérimental reste local et ne simule pas de véritable transmission réseau : les temps de transfert utilisés sont estimés plutôt que mesurés sur une connexion réelle.
- Le ratio de compression est fixe dans les tests (12 bits), ce qui ne reflète pas toujours la variabilité des données du monde réel.

Ces limites ne remettent pas en cause la validité du projet, mais indiquent des pistes claires d'amélioration pour une version plus complète.

4.3. Perspectives d'amélioration

Plusieurs pistes pourraient être envisagées pour aller plus loin :

- **Optimisation des accès** : améliorer la fonction `get()` dans la version chevauchante pour réduire le coût des décalages multiples.
- **Compression adaptative** : ajuster dynamiquement le nombre de bits k selon la distribution des valeurs plutôt que de l'imposer globalement à tout le tableau.
- **Parallélisation** : exécuter la compression et la décompression sur plusieurs cœurs en divisant le tableau en segments indépendants, afin de réduire les temps de traitement.
- **Simulation réseau réelle** : introduire un module de test avec latence artificielle ou sockets pour mesurer le gain de temps dans un contexte de transmission concret.
- **Interface utilisateur minimale** : proposer une petite interface en ligne de commande permettant de choisir le mode, la taille du tableau et de visualiser les résultats directement.

Ces évolutions permettraient de rendre le projet plus complet et de le rapprocher d'un outil utilisable pour des applications de compression légère ou de stockage optimisé.

4.4. Gestion optionnelle des entiers négatifs

Le sujet proposait en bonus la gestion des entiers négatifs, qui n'a pas été intégrée dans cette première version. Ce cas particulier introduit des complications supplémentaires, car la représentation binaire classique utilise le *complément à deux*, ce qui rend le calcul du nombre de bits non trivial.

Plusieurs approches sont envisageables :

- **Ajout d'un bit de signe** : réserver un bit supplémentaire pour indiquer si le nombre est positif ou négatif. Cela augmente légèrement la taille moyenne, mais garde la logique actuelle inchangée.
- **Décalage d'origine** : transformer toutes les valeurs négatives en positives avant compression, par exemple en ajoutant une constante égale à la valeur absolue du minimum du tableau.
- **Encodage différentiel** : stocker non pas les valeurs absolues mais les différences entre éléments successifs, ce qui réduit souvent la plage de valeurs à représenter.

5. Conclusion

5.1. Synthèse du travail

Ce projet avait pour objectif d'explorer et d'implémenter différentes méthodes de compression d'entiers par *bit packing*, afin de réduire le volume de données à transmettre tout en conservant un accès direct à chaque élément. Trois approches ont été développées : une version alignée, une version chevauchante et une version avec débordement. Le travail a également intégré un protocole de mesure pour évaluer leurs performances en temps d'exécution et en taux de compression.

5.2. Résultats obtenus

Les tests ont montré que la compression est très efficace pour des données bornées, avec un ratio stable autour de 0.375, soit une réduction d'environ 60 % de la taille initiale. La version simple s'est révélée la plus rapide, tandis que la version overflow offre un meilleur équilibre sur des jeux de données hétérogènes. L'accès direct aux éléments, même en mode compressé, reste rapide et constant, validant ainsi le principal objectif du projet.

5.3. Ouvertures possibles

Plusieurs pistes peuvent prolonger ce travail : ajouter la gestion des entiers négatifs, introduire une compression adaptative selon la distribution des valeurs, ou simuler un véritable contexte de transmission réseau pour mesurer le gain réel sur des flux distants. Une version parallèle ou partiellement vectorisée pourrait également accélérer le traitement sur des volumes plus importants.

A. Annexes

A.1. Extraits de code commentés

Pour illustrer le fonctionnement complet d'une implémentation, cette annexe présente le code intégral de la version alignée (`BitPackingV1`). Elle comprend les trois fonctions principales du projet : `compress()`, `decompress()` et `get()`. Cette version constitue la base de référence sur laquelle les deux autres variantes ont été développées.

Listing A.1 – Exemple d'implémentation de `BitPackingV1`

```
from .base import BitPacking

class BitPackingV1(BitPacking):
    def compress(self, arr):
        packed = []
        bits_in_int = 32
        mask = (1 << self.k) - 1
        current_val = 0
        shift = 0

        for num in arr:
            current_val |= (num & mask) << shift
            shift += self.k

            if shift >= bits_in_int:
                packed.append(current_val)
                shift -= bits_in_int
                current_val = (num >> (self.k - shift)) if shift > 0
                    else 0

        if shift > 0:
            packed.append(current_val)

        self.data = packed
        self.original_size = len(arr)
        return packed

    def decompress(self):
        result = []
        mask = (1 << self.k) - 1
        bits_in_int = 32

        for i in range(self.original_size):
            index = (i * self.k) // bits_in_int
            offset = (i * self.k) % bits_in_int
```

```
            val = (self.data[index] >> offset) & mask
            result.append(val)
        return result

    def get(self, i):
        bits_in_int = 32
        mask = (1 << self.k) - 1
        idx = (i * self.k) // bits_in_int
        offset = (i * self.k) % bits_in_int
        return (self.data[idx] >> offset) & mask
```

A.2. Exemple de sortie du benchmark

```
Mode: overflow
Compress time:   23.24 ms
Decompress time: 19.77 ms
Get time:        3.40 µs
Compression ratio: 0.375
```