DESCRIPTION DU PROGRAMME

On peut diviser son fonctionnement en deux grandes étapes.

1. L'étape LZ77 : "La détection de motifs"

Au lieu de lire chaque lettre une par une, le programme agit comme un détective qui cherche des déjà-vus.

- La fenêtre glissante (Sliding Window) : Le programme garde en mémoire les 16 384 derniers octets lus. C'est sa zone de recherche.
- La Table de Hachage : Pour aller très vite (surtout sur 200 Mo), il utilise une table de hachage. Elle indexe chaque triplet de lettres. S'il voit "abc", il regarde instantanément s'il a déjà vu "abc" dans sa fenêtre.
- Le Remplacement : S'il trouve une répétition (par exemple, une phrase de 20 lettres déjà vue plus tôt), il ne réécrit pas la phrase. Il écrit un Jeton (Token) :
    - offset : "Recule de X octets."
    - len : "Copie les Y prochains caractères."
    - char : "Puis ajoute cette lettre."

2. L'étape Huffman (via Serialize) : "L'économie de bits"

Une fois que le LZ77 a fini, nous avons une longue liste de LZTokens. Mais ces jetons sont encore un peu lourds. C'est là qu'intervient le codage entropique.

- Analyse des fréquences : Le programme repère quels jetons reviennent le plus souvent.
- Raccourcis binaires : Au lieu d'utiliser la même taille pour tous les jetons, l'algorithme donne des codes très courts (ex: 2 ou 3 bits) aux jetons fréquents et des codes plus longs aux jetons rares.
- Résultat : C'est cette étape qui permet de "tasser" les 112 Mo issus du LZ77 pour tomber à 53,43 Mo. C'est le principe utilisé par le format ZIP ou GZIP.

3. Pourquoi est-ce si efficace sur le fichier ?

Le taux de compression exceptionnel qu’on as obtenu (73,29% de réduction) s'explique par deux facteurs :

1. Redondance élevée : Le fichier texte contient probablement beaucoup de mots ou de structures répétées.
2. Fenêtre large : En utilisant une fenêtre de 16 Ko, le programme est capable de lier des répétitions très éloignées les unes des autres.

ALGORTITHME UTILISER

1. L'Algorithme LZ77 (Lempel-Ziv 1977)

Son rôle est de détecter et de supprimer la redondance textuelle.

Le concept de la "Fenêtre Glissante" (Sliding Window)

L'algorithme parcourt le fichier et garde en mémoire les 16 384 derniers caractères lus (la fenêtre). Pour chaque nouvelle séquence de caractères, il regarde s'il l'a déjà vue dans cette fenêtre.

- Si oui : Il remplace la séquence par un Pointeur (ou Jeton). Ce jeton dit simplement : "Recule de X caractères et recopie les Y prochains".
- Si non : Il écrit le caractère tel quel (on appelle cela un "Littéral").

    - offset : Distance en arrière vers le début du motif.
    - len : Nombre de caractères à copier.
    - char : Le caractère qui casse la répétition (permet d'avancer d'au moins 1 cran à chaque tour).

2. Le Codage Entropique (Huffman / Serialization)

C'est ici que se produit la "magie" qui a fait passer ton fichier de 112 Mo à 53,43 Mo.

Une fois que le LZ77 a transformé le texte en une liste de jetons, certains jetons sont beaucoup plus fréquents que d'autres (par exemple, le jeton représentant un espace ou la lettre 'e').

- Le principe de Huffman : Au lieu d'utiliser 8 bits pour chaque caractère, l'algorithme attribue des codes de longueur variable.
    - Les symboles très fréquents sont codés sur 2 ou 3 bits.
    - Les symboles rares sont codés sur 10 ou 12 bits.

ARCHITECTURE DU SYSTEME

1. Couche d'Entrée (Input Layer)

- Composant : Lecteur de flux (File Reader).
- Rôle : Charge le fichier de 200 Mo en mémoire sous forme d'un vecteur d'octets (Vector{UInt8}).
- Contrainte : Vérifie la barrière de sécurité des 100 Mo avant d'autoriser le passage à la couche suivante.

2. Couche d'Analyse et Indexation (Hashing Engine)

C'est le "cerveau" qui accélère le traitement.

- Algorithme : Table de hachage (Hash Map).

- Fonctionnement : Elle indexe des séquences de 3 octets. Au lieu de scanner tout le passé, l'algorithme consulte cette table pour trouver instantanément la dernière position connue d'une séquence.

3. Couche de Compression Spatiale (LZ77 Engine)

C'est ici que la redondance textuelle est éliminée.

- Mécanisme : Fenêtre glissante (Sliding Window) de 16 Ko.
- Transformation : Le texte est converti en une liste de structures LZToken.
    - Si un motif est trouvé : (Distance, Longueur, Prochain Caractère).
    - Si aucun motif n'est trouvé : (0, 0, Caractère Brut).

4. Couche de Compression Statistique (Entropy Layer)

C'est la couche qui a permis d'atteindre les 53,43 Mo.

- Algorithme : Codage de longueur variable (type Huffman).
- Rôle : Analyse la fréquence d'apparition des jetons.
- Optimisation : Elle réduit le coût binaire des jetons les plus fréquents. Par exemple, un jeton très courant ne prendra que quelques bits, alors qu'un jeton rare en prendra plus. Julia utilise ici la sérialisation optimisée pour compacter ces objets en mémoire.

5. Couche de Sortie (Output Layer)

- Composant : Binary Streamer.
- Rôle : Écrit les données finales sur le disque avec l'extension `.lzh`.
- Sécurité : Utilise la fonction flush() pour garantir que la mémoire tampon est bien vidée sur le disque physique.

RESULTAT EXPERIMENTAUX

Voici la resultat apres compression de quelque fichier :

Taille du fichier (Mo)	Temps de compression (Sec)	Taux de compression (%)	Taille final (Mo)
200	0,684	73,23	53,53
400	1,309	73,24	107,04
600	1,933	73,23	160,6
800	2,771	73,23	214,15
1000	3,167	73,23	274,09
1200	3,781	73,24	328,87

ANALYSE ET DISCUSSION

1. Analyse Technique du Programme

La Stratégie d'Écriture (Buffering)

Contrairement à une approche naïve qui consisterait à créer une chaîne de caractères géante en mémoire avant de l'enregistrer, ce programme utilise une écriture par blocs.

- Pourquoi ? Charger 1,2 Go de texte en RAM avant l'écriture pourrait ralentir le système, voire faire planter le script si la mémoire disponible est limitée.
- Fonctionnement : On génère un paragraphe, on l'écrit immédiatement sur le disque, puis on vide la mémoire pour le paragraphe suivant. La consommation de RAM reste donc constante (quelques kilo-octets), qu'on génère 50 Mo ou 10 Go.

Précision des Tailles (Binaire vs Décimal)

Le programme utilise le standard binaire informatique (1 Mo = 1024^2 octets).

- Pour 1,2 Go, le calcul est : 1,2 * 1024 * 1024 * 1024 octets.
- L'utilisation de len(paragraphe.encode('utf-8')) est cruciale. En Python, une chaîne de caractères (str) n'a pas la même taille en mémoire qu'une fois convertie en octets sur le disque, surtout avec des caractères spéciaux. Cette méthode garantit une précision au centième de Mo près.

2. Discussion sur les Cas d'Usage

Pourquoi générer de tels volumes de texte "artificiel" ? Voici les applications principales :

Domaine	Utilité
Tests de Performance	Vérifier comment un logiciel de traitement de texte ou un éditeur (comme VS Code ou Notepad++) réagit face à un fichier de 1 Go.
Big Data	Simuler des jeux de données pour tester des algorithmes de recherche (Elasticsearch, Grep) ou de tri.
Compression	Évaluer l'efficacité des algorithmes (ZIP, GZIP, 7z). Notez que comme notre texte est répétitif, il se compressera très bien.
Réseau	Tester la vitesse de transfert de fichiers entre un serveur et un client sans utiliser de données sensibles.