TD : HACHAGE POLYNOMIAL

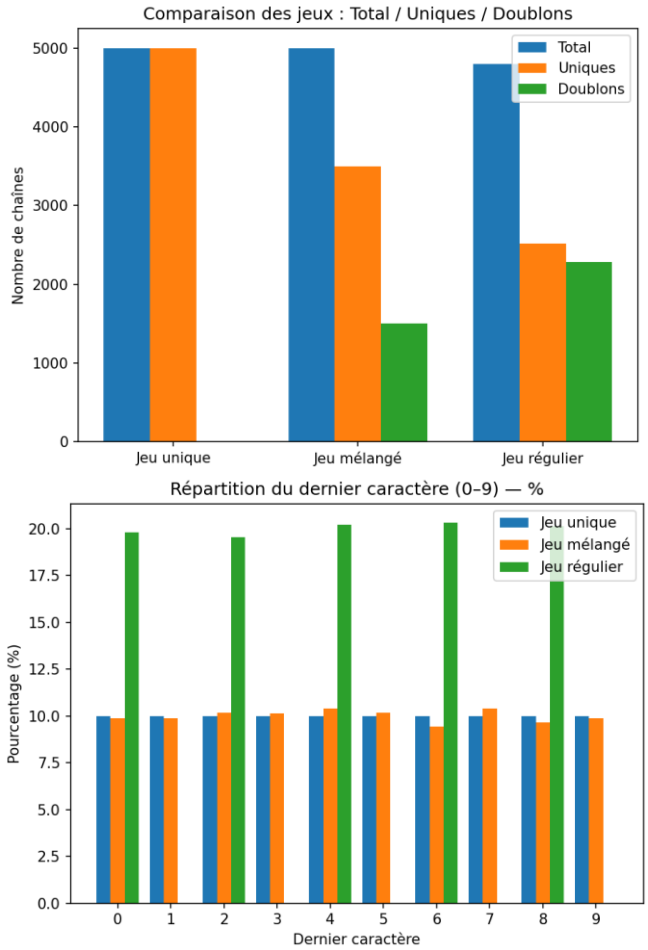
Dans ce TD, vous allez concevoir vos propres tables de hachage afin de comprendre concrètement comment un hachage polynomial répartit des chaînes de caractères et pourquoi de « mauvais » choix de paramètres (base, taille de table paire ou puissance de 2) peuvent cibler des régularités dans les données et des emplacements vides.

Vous implémenterez la stratégie par chaînage en instrumentant chaque opération (insertion, recherche, suppression). Vous mettrez également en œuvre une politique de redimensionnement automatique (seuil 70 %).

I) JEUX DE DONNÉES UTILISÉ

Il y a trois listes de jeux de données sous forme de chaines de caractères :

- `jeu_unique` : contient 5000 clés distinctes,
- `jeu_melange` : contient 5000 clés avec 30 % de doublons
- `jeu_regulier` : contient environ 5000 clés avec de nombreuses fins identiques sur les derniers caractères pairs.



Vous pouvez visualiser ces données à l'aide des fonctions `Affiche_Repartion_Donnees()` et `Affiche_Repartion_Derniers_Caracteres()` qui sont commentées par défaut dans le fichier source.

II) RAPPELS SUR LE HACHAGE POLYNOMIAL

Le principe du hachage polynomial est d'encoder une chaîne de caractère en une valeur numérique (code de hachage) en utilisant la valeur du code ASCII/Unicode de chaque caractère. Elle consiste à itérer sur les caractères un par un et de maintenir une somme courante. À chaque caractère, on multiplie la somme par une constante, on ajoute le nouveau caractère, puis, si nécessaire, on prend un modulo pour éviter les dépassements :

Hachage polynomial

1. Encoder chaque caractère en un entier (ASCII/Unicode)
2. Itérer sur la chaîne : on maintient une valeur k
3. À chaque caractère de code x, on met à jour :

$$k_i = (k_{i-1} \cdot B + x_i) \bmod M$$

B : base (une constante > taille de l'alphabet, ex. 131 ou 257, en évitant les puissances de 2).
M = modulo (grand entier pour éviter les débordements et réduire les collisions) (on peut prendre un grand nombre premier M = 1 000 000 007, ou bien s'appuyer sur le débordement 64 bits non signé en prenant M = 2^{64})

4. Pour obtenir un numéro de compartiment entre 0 et n-1, on peut appliquer ensuite la fonction de compression à base de modulo :

$$h(k) = k \bmod n$$

III) HACHAGE POLYNOMIAL

Dans cette partie, on utilise le fichier « TD1.1_CodeHash.py »

III.1. Code de hachage

Écrire la fonction `hash_poly(s : str, B : int, M :int)` qui prend en entrée une chaîne de caractères `s`, une base `B` et un modulo M et qui retourne un entier en utilisant la méthode du hachage polynomial. Utiliser la fonction `ord()` afin d'avoir la valeur Unicode d'un caractère.

Vérifier : `hash_poly("abc",257,10**9+7) = 6432038`
`hash_poly("",257,101) = 0`

III.2. Effet d'une régularité des données avec un mauvais choix de (B, M)

On prend M=32 et B=128.

1. Calculer les codes de hachage de « abc », « bc » et « c ». Que remarquez-vous ? Expliquer.
2. Afficher la distribution des codes de hachage des jeux de données avec la fonction `Distribution_Code_Hash(valeurs, M, B)` (implémentée dans la bibliothèque `malib_td1`)
3. Comparer avec M=33 et B=128.

IV) TABLE DE HACHAGE AVEC CHAÎNAGE

Dans cette partie, on utilise le fichier « TD1.2_TableHachage_Chainage.py ».

IV.1. Fonction de hachage {code hash + compression}

1. Recopier votre code de la fonction `hash_poly()` dans le fichier.

2. Écrire la fonction de compression `f_Compression(code, n)` prenant en entrée le code du hash, le nombre de compartiments n de la table de hachage et retournant l'indice de l'emplacement où stocker la valeur.

3. Écrire la fonction `f_Hachage(cle, B, M, n)` permettant d'appliquer la fonction de hachage {hash _code + compression} sur une clé de type chaîne de caractère et de retourner le hash correspondant.

   Tester avec n = 10007 : `f_Hachage("abc",257,10**9+7,10000+7) = 7544`
   `f_Hachage("abcdefg",257,10**9+7,10000+7) = 6625`

4. Créer la table de hachage vide : `table_hachage = [[] for i in range(n)]`

IV.2. Insertion d'un élément

1. Écrire la fonction `f_Insert(element, table)` qui ajoute un élément (chaîne de caractères) par chaînage et retourne le facteur de charge courant. Dans cette version, on accepte que des doublons puissent être stockés.

Vous pourrez utiliser une variable globale `n_total` pour suivre le nombre total d'éléments enregistrés dans la table.

Tester :
```
>>> f_Insert("user8346", table_hachage)
9.9930048965724e-05
>>> table_hachage[3514]
['user8346']
>>> f_Insert("user9490", table_hachage)
0.000199860097931448
>>> table_hachage[3514]
['user8346', 'user9490']
>>> f_Insert("user9490", table_hachage)
0.00029979014689717197
>>> table_hachage[3514]
['user8346', 'user9490', 'user9490']
```

2. Modifier votre fonction pour gérer les doublons et faire en sorte que la table de hachage n'en contienne pas.

Tester pour vérifier que la gestion des doublons fonctionne correctement.

IV.3. Recherche d'un élément

La position d'un élément dans la table de hachage est donnée par une liste de la forme :

`[indice_compartiment, indice_debordement]`

… où `indice_compartiment` donne la position de la valeur recherchée dans la table de hachage et `indice_debordement` la position dans la sous-liste où sont enregistrées les collisions.

1. Écrire la fonction `f_Recherche(element, table)` qui retourne la position de l'élément dans la table, s'il existe, et qui retourne `None` sinon.

Tester :
```
>>> f_Recherche("user8346", table_hachage)
[3514, 0]
>>> f_Recherche("user9490", table_hachage)
[3514, 1]
>>> f_Recherche("abc", table_hachage)
-1
```

IV.4. Suppression d'un élément

1. Écrire la fonction `f_Supprime(element, table)` qui supprime un élément de la table, et qui retourne le facteur de charge après la suppression ou retourne -1 si l'élément n'a pas été trouvé.

Tester :
```
>>> table_hachage[3514]
['user8346', 'user9490']
>>> f_Supprime("user9490", table_hachage)
>>> 0.00029979014689717197
>>> table_hachage[3514]
['user8346']
>>> f_Supprime("abs", table_hachage)
-1
```

IV.5. Distribution du nombre de collisions

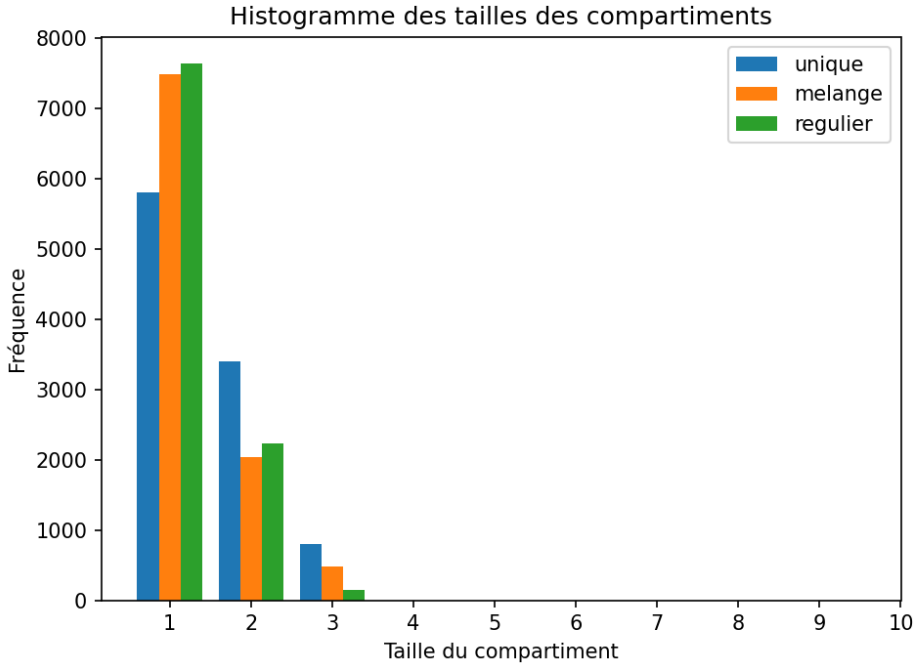
1. Écrire une fonction `f_RemplirTable(jeu,table)` qui insère la totalité des données d'un jeu dans une table.

2. Remplir trois tables : `table_jeu_unique` (avec le jeu `jeu_unique`), `table_melange` (avec le jeu de données `jeu_melange`) et `table_regulier` (avec le `jeu_regulier`).

3. Afficher la distribution du nombre de collisions de chaque table avec la fonction `Distribution_Compartiments(nbr_par_case)` implémentée dans la librairie `malib_td1`. Ici `nbr_par_case` est la liste des nombres d'élément par compartiment.

3. Commenter les résultats obtenus.

Observation attendue :

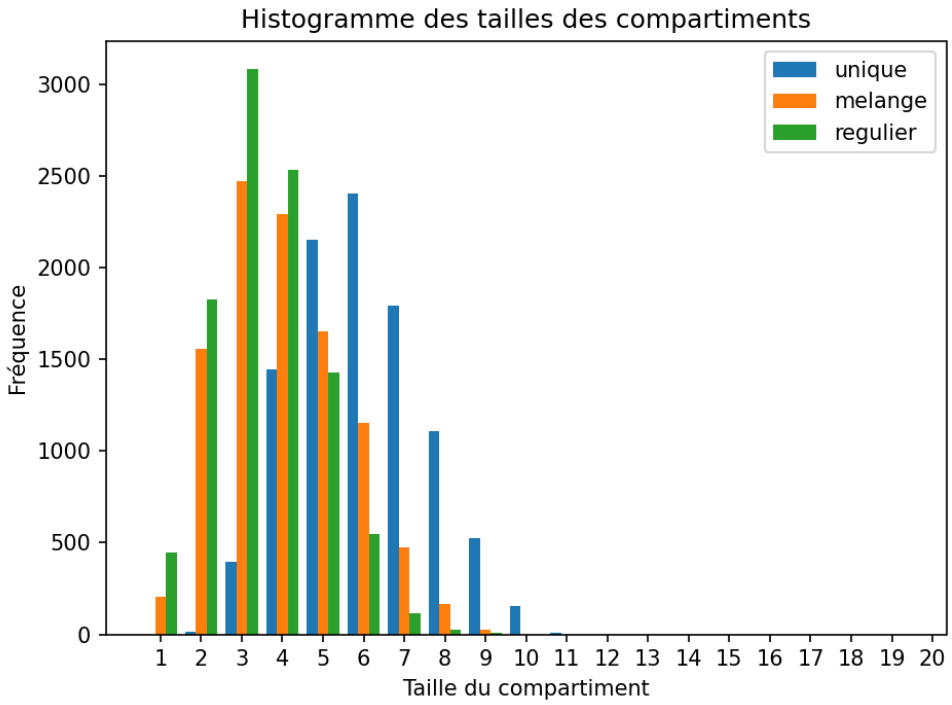


IV.6. Gestion du facteur de charge

1. Augmenter le nombre de données de chaque jeu en modifiant la taille des jeux de données x10.

2. Afficher la distribution des du nombre de collisions de chaque table comme précédemment.

On observe que les compartiments se remplissent davantage et que comme précédemment le jeu unique concentre davantage de collisions que les autres jeux :



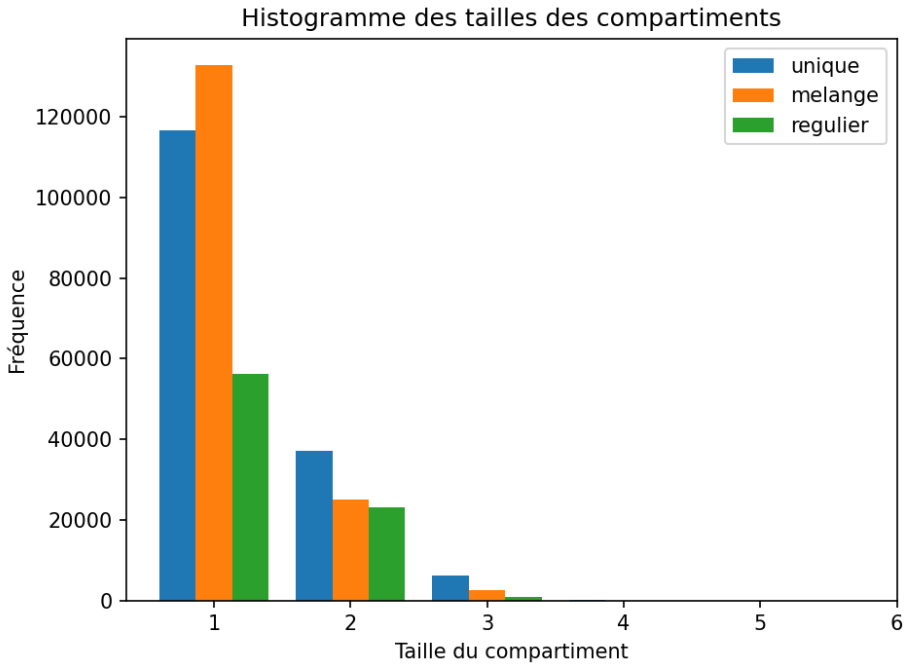
3. Écrire la fonction `f_NbrPremier(val)` qui retourne le premier nombre premier strictement supérieur à la valeur de `val`.

```
Tester :    >>> f_NbrPremier(256)          >>> f_NbrPremier(10000)
            257                           10007
```

4. Écrire la fonction `f_Insert_avec_charge(element, table)` qui insère un élément en maintenant un facteur de charge inférieur à 70% (en doublant la capacité n au plus proche nombre premier supérieur à 2·nlors d'un dépassement).

5. Écrire une fonction `f_RemplirTableAvecCharge(jeu,table)` qui insère la totalité des données d'un jeu dans une table avec gestion du facteur de charge.

6. Refaire l'expérience de remplissage des trois tables en utilisant cette fois-ci la fonction `f_Insert_avec_charge(element, table)` et vérifier que la répartition des données dans les compartiments s'améliore.



Vous pouvez faire la suite si vous avez encore du temps !

IV.7. Complexité des opérations d'insertion

L'utilisation des listes dans nos fonctions d'insertion ne permet pas de prendre en compte les délais d'insertion pour mesurer la complexité en temps car les listes python sont optimisées et gérées dynamiquement.

Pour avoir une meilleure image de ce que représente en réalité le temps d'insertion dans la table de hachage, nous allons nous baser sur le nombre de valeurs déjà enregistrées dans un compartiment de la table de hachage avant l'insertion d'une nouvelle valeur.

L'objectif est de remplir une table de taille n = 10007 avec 300 000 valeurs issues du jeu unique, et d'enregistrer le nombre de cases du compartiment où cette valeur sera placée à chaque tentative d'insertion. Cette métrique nous donnera une idée de la complexité du temps d'insertion dans la table de hachage car la complexité dépend du temps mis lors de la vérification des doublons (et plus il y a de cases remplies dans un compartiment, plus cette vérification sera longue).

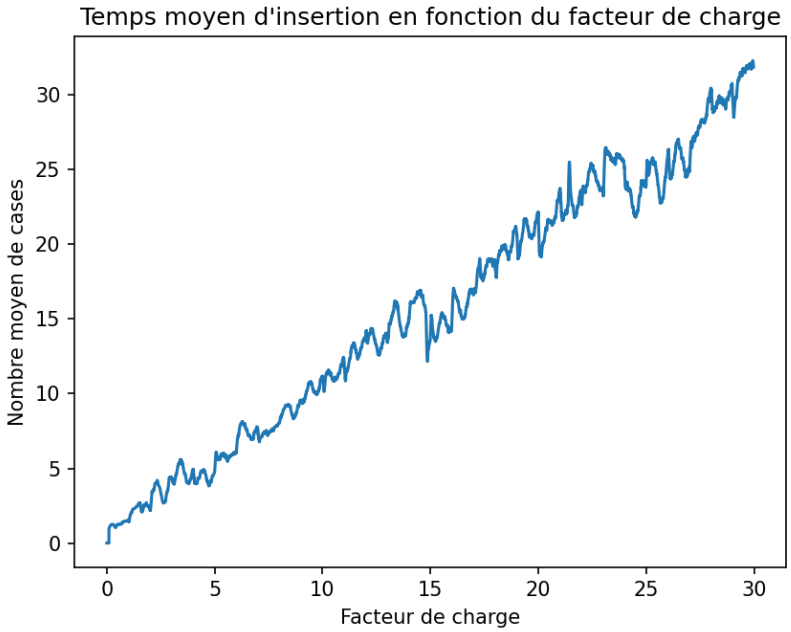
1. Créer un jeu de données de 100k valeurs en modifiant la taille du jeu unique.

2. Créer une fonction `f_CountRemplirTable(jeu, table)` qui permet de remplir la table en renvoyant deux listes : `alpha` et `count`, contenant pour la première la valeur du facteur de charge à chaque insertion d'un élément, et pour la seconde le nombre de cases déjà remplies dans le compartiment cible lors d'une insertion.

3. Créer une fonction `f_MoyenneMobile(liste, fenetre)` qui retourne une liste contenant les moyennes des valeurs contenues dans la liste, sur une fenêtre de taille donnée.
   Vous pouvez utiliser `np.mean(liste[x:x+y])` pour calculer la moyenne des données de la liste sur l'intervalle [x,x+y].
   Vous ferez attention à ce que la liste retournée ait la même longueur que la liste traitée (ajouter des 0 au début de la liste avant de la retourner).

4. Afficher les courbes de la moyenne mobile (prendre par exemple 1000) du nombre de cases lors des opérations d'insertion en fonction du taux de charge de la table.

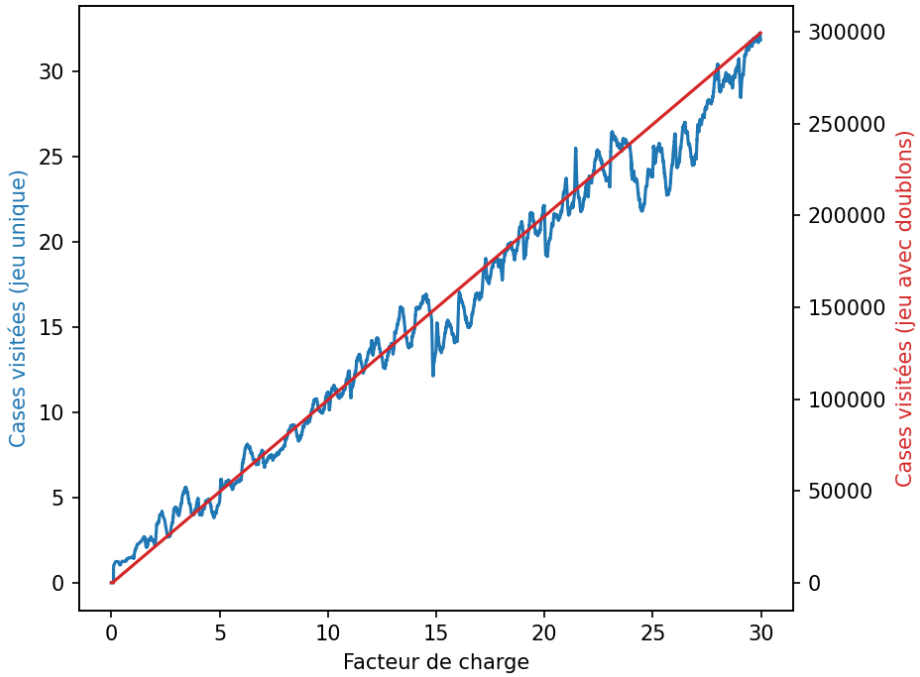
Résultat attendu (sur 300k valeurs) :



Cette courbe semble être en O(1 + α). Pour la comparer à ce qu'on aurait dans le pire des cas, on va refaire la même mesure mais sur un jeu de donné qui ne contient que des doublons (pour forcer les collisions).

5. Créer un jeu de la même taille de précédemment, et utiliser la fonction d'insertion qui ne gère pas les doublons pour remplir la table (afin de forcer le remplissage du compartiment) en effectuant les mêmes mesures que précédemment. Afficher les courbes dans les deux cas.

Résultat attendu :



Quelle est la complexité de l'opération d'insertion dans le pire des cas ?

6. La fonction de hachage utilisée garantit-elle une complexité en $O(1 + \alpha)$ sur l'insertion comme le ferait une fonction universelle ?