Étude de la hauteur moyenne d'un arbre binaire de recherche

1 Arbre binaire de recherche : définition, implémentation, construction

Un arbre binaire de recherche (ABR) est un arbre binaire particulier. Un arbre binaire est un graphe orienté vérifiant les conditions suivantes :

- il n'y a pas de cycle,
- il existe un et un seul sommet qui n'a pas d'antécédent (i.e. aucun arc n'arrive dans ce sommet) : c'est la racine de l'arbre,
- tous les sommets non-racines ont exactement un antécédent (i.e. il existe exactement un arc qui arrive dans ce sommet) : c'est le père du sommet,
- tous les sommets ont au plus deux arcs sortants : s'ils existent, les sommets cibles de ces arcs sont les fils du sommet.

Dans un arbre binaire de recherche, un sommet fils a toujours une orientation : on parle de fils gauche et fils droit. Pour un sommet qui n'a qu'un fils, ce fils peut être droit ou gauche (même dans ce cas, son orientation doit être précisée). Pour un sommet qui a deux fils, il y a forcément un fils gauche et un fils droit. De plus, chaque sommet a une étiquette (dans ce projet les étiquettes seront des éléments de $\mathbb{R}_+$). Pour qu'un tel arbre soit un ABR, il faut que la condition suivante soit vérifiée : pour tout sommet de l'arbre, l'étiquette du sommet doit être supérieure à toutes celles de son fils gauche et inférieure à toutes celles de son fils droit. Deux ABR sont donnés en exemple à la Figure 1. Dans tout ce projet, vous pouvez supposer que toutes les étiquettes des ABR sont deux-à-deux distinctes (en particulier, vous ne serez pas sanctionnés si vos programmes ne fonctionnent pas sur les arbres où une même étiquette apparait plusieurs fois).

Les ABR sont des structures données triées (en un certain sens), ce qui rend la recherche de donnée efficace, car il n'est souvent pas nécessaire de parcourir l'intégralité de la structure pour chercher une donnée précise mais seulement une branche de l'arbre : par exemple si vous cherchez l'étiquette 3 dans l'arbre de gauche de la Figure 1, vous comparez d'abord 3 à l'étiquette de la racine 2, puis vous allez dans la branche droite (car $3 > 2$), puis dans la branche gauche (car $3 < 4$), pour trouver le sommet d'étiquette 3.

Définition. *On appelle hauteur d'un arbre la longueur de sa plus longue branche (i.e. le plus long chemin sur le graphe partant de la racine). Si A est un arbre, on notera $h(A)$ sa hauteur.*

Par exemple, dans la Figure 1, la hauteur respective des deux ABR est 2 et 4.

La notion de hauteur d'un arbre est importante car la complexité de beau-

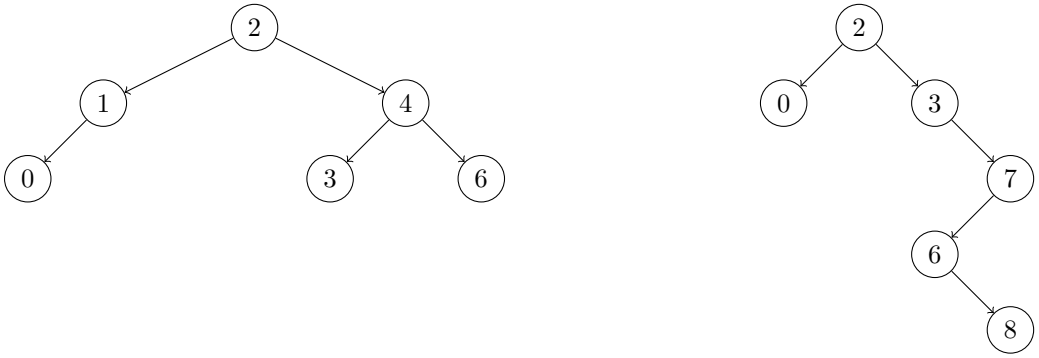

FIGURE 1 – Deux exemples d'ABR

coup d'algorithmes dépend de la hauteur et non du nombre de sommets. Le but de ce projet est de montrer que, en moyenne, la hauteur d'un ABR à n sommets construit aléatoirement est de l'ordre de $\ln n$. La question **T1** suivante montre que c'est l'ordre de grandeur optimal.

T1. Montrer que, pour tout arbre binaire A à n sommets, $\log_2(n+1) - 1 \leq h(A) \leq n - 1$, où $\log_2$ désigne le logarithme binaire.

Une manière simple d'implémenter les ABR en python est d'utiliser une matrice $n \times 3$ (avec n le nombre de sommets de l'ABR) : la matrice m d'un ABR est définie de la manière suivante

- $m[0][0]$ est l'étiquette de la racine,
- $m[0][1]$ et $m[0][2]$ sont les "positions" respectives des fils gauche et droit de la racine dans la matrice
- pour $1 \leq i \leq n-1$, $m[i][0]$ est l'étiquette du sommet en position i,
- $m[i][1]$ et $m[i][2]$ sont les "positions" respectives des fils gauche et droit du sommet en "position" i.

Il existe plusieurs implémentations d'un même ABR, car le choix des "positions" dans la matrice est arbitraire (sauf pour la racine qui est toujours en position 0). Une absence de fils sera signalée par un -1 (voir exemple ci-dessous). Par exemple, les deux matrices suivantes sont des implémentations en python de l'ABR de gauche de la Figure 1 :

```
m1 = [[2, 1 , 2 ],
      [1, 3 , -1],
      [4, 4 , 5 ],
      [0, -1, -1],
      [3, -1, -1],
      [6, -1, -1]]

m2 = [[2, 5 , 2 ],
      [0, -1, -1],
```

```
[4, 4 , 3 ],
[6, -1, -1],
[3, -1, -1],
[1 , 1 , -1]]
```

S1 (facultative). Écrire un programme pour représenter graphiquement un ABR (en utilisant ce programme sur les matrices $m1$ et $m2$ ci-dessus, votre programme doit afficher quelque chose qui ressemble à l'ABR de gauche de la Figure 1). Cette question est relativement libre, même des programmes qui font un affichage "minimal" seront récompensés.
Indice : vous pouvez utiliser le package "graphics.py" (voir par exemple `https://anh.cs.luc.edu/handsonPythonTutorial/graphics.html`), "matplotlib.pyplot" (voir `https://matplotlib.org/3.5.3/api/_as_gen/matplotlib.pyplot.html`), "networkx" (voir `https://networkx.org/`), ou autre (dans ce cas, il faudra écrire le lien de la documentation ou d'un tutoriel concis).

S2. Écrire un programme qui teste si un arbre binaire (implémenté comme ci-dessus) est un ABR. Votre programme n'a pas à vérifier que l'entrée correspond bien à l'implémentation d'un arbre binaire.
Utiliser votre programme sur les arbres $m1$ et $m2$ définies ci-dessus.

S3. Écrire un programme qui étant donné une étiquette et un ABR, teste si l'étiquette est présente dans l'arbre. Votre programme doit renvoyer la position de l'étiquette si elle est présente, et -1 si elle est absente (la position est à comprendre au sens de l'implémentation décrite plus haut).
Tester votre programme pour chercher les étiquettes 4 et 5 dans l'arbre $m1$.

S4. Écrire un programme qui calcule la hauteur d'un arbre binaire. Utiliser votre programme pour calculer les hauteurs des arbres $m1$ et $m2$.
Indice : vous pouvez commencer par écrire un programme récursif qui étant donné un arbre et une position d'un sommet, calcule le plus long chemin partant de ce sommet.

Il existe une manière simple d'insérer une nouvelle étiquette dans un ABR, tout en conservant la structure d'ABR. Par exemple, si vous voulez insérer l'étiquette 5 dans l'ABR de gauche de la Figure 1, il suffit de parcourir l'arbre en partant de la racine et en respectant les ordres relatives des étiquettes déjà présentes jusqu'à trouver de la place : comme $5 > 2$, il faut d'abord aller dans la branche de droite, puis encore dans celle de droite (car $5 > 4$), puis dans celle de gauche (car $5 < 6$) qui est vide, il suffit alors d'insérer 6 pour obtenir l'ABR de la Figure 2.

S5. Écrire un programme qui étant donné une étiquette et un ABR, modifie l'ABR en lui insérant la nouvelle étiquette. L'arbre ainsi modifié doit toujours être un ABR.
Utiliser votre programme pour insérer l'étiquette 5 dans l'arbre $m1$.

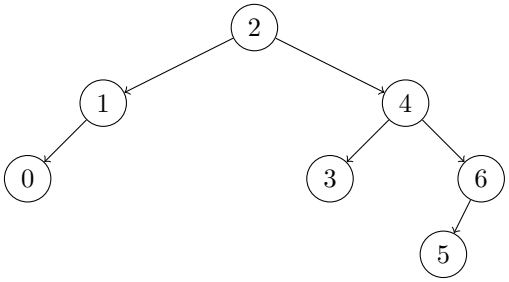

FIGURE 2 – Exemple d'insertion

Si vous voulez utiliser un ABR pour ranger des données (dans ce projet, nous supposerons que les données sont les étiquettes pour simplifier) qui arrivent les unes après les autres, vous devrez construire votre ABR au fur et à mesure que les données arrivent. L'ordre d'arrivée des données sera défini par une permutation. Dans la suite, nous notons $\mathfrak{S}(A, B)$ l'ensemble des bijections de A dans B (où A, B sont des ensembles finis de même cardinal), et $\mathfrak{S}_n$ l'ensemble des permutations de $[\![0, n-1]\!]$.

Definition. *Pour $\sigma \in \mathfrak{S}_n$, nous notons A_n^σ l'ABR obtenu dont l'étiquette de la racine est $\sigma(0)$, et où on insère successivement $\sigma(1)$, puis $\sigma(2)$,... jusqu'à $\sigma(n-1)$.*

Par ailleurs, nous représenterons n'importe quel $\sigma \in \mathfrak{S}_n$ comme le $n-$uplet (que l'on implémentera par une List en python) de la manière suivante :

$$\sigma = [\sigma(0), \sigma(1), ..., \sigma(n-1)].$$

Par exemple, l'ABR de la Figure 2 est l'arbre A_7^σ avec la permutation σ suivante (d'autres permutations permettent d'obtenir le même arbre)

$$\sigma = [2, 1, 4, 6, 3, 0, 5]$$

S6. Écrire un programme qui construit A_n^σ en fonction de σ (le paramètre σ est donc une List, et votre programme doit renvoyer un ABR implémenté comme expliqué plus haut).
Utiliser votre programme pour construire l'ABR A_7^σ pour la permutation σ ci-dessus.

Dans la suite, nous aurons parfois besoin de construire des ABR dont l'ensemble des étiquettes n'est pas de la forme $[\![0, n-1]\!]$. Étant donné deux ensembles d'entiers de cardinal $n \in \mathbb{N}^*$, $E = \{e_0, ..., e_{n-1}\}$ et $F = \{f_0, ..., f_{n-1}\}$ (de telle sorte que $e_0 < e_1 < ... < e_{n-1}$), on définit l'ABR A_n^σ comme l'ABR initialement vide dans lequel on insère successivement $\sigma(e_0)$, puis $\sigma(e_1)$,... jusqu'à $\sigma(e_{n-1})$. Autrement dit, les éléments de F sont les étiquettes de A_n^σ, et l'ordre relatif des éléments de E définit l'ordre d'insertion des étiquettes (qui est lui-même permuté par σ).

Pour montrer formellement certains résultats de la Section 3, nous aurons besoin de la définition technique suivante.

Definition. *Soient* $E = \{e_0, ..., e_{n-1}\}$ *et* $F = \{f_0, ..., f_{n-1}\}$ *deux ensembles d'entiers de cardinal fini* n *avec* $e_0 < ... < e_{n-1}$ *et* $f_0 < ... < f_{n-1}$. *Si* $\sigma \in \mathfrak{S}(E, F)$, *alors pour tout* $0 \leq i \leq n-1$, *il existe un unique* $0 \leq j_i \leq n-1$ *telle que* $\sigma(e_i) = f_{j_i}$. *On note alors* $\tilde{\sigma}$ *l'élément de* $\mathfrak{S}_n$ *défini par* $\tilde{\sigma}(i) = j_i$. *Autrement dit, si on identifie chaque* e_i *avec* i, *et chaque* f_j *avec* j, *cela revient à identifier* σ *avec* $\tilde{\sigma}$ *:* $\sigma(e_i) = f_{\tilde{\sigma}(i)}$

Remarque. Si $\sigma \in \mathfrak{S}(E, F)$ (avec les mêmes notations que dans la Définition ci-dessus), alors A_n^σ est exactement l'ABR $A_n^{\tilde{\sigma}}$ où chaque étiquette j est remplacée par l'étiquette f_j.

2 Quelques propriétés de permutations aléatoires

Remarque. *Dans cette section (et la suivante), les arguments combinatoires devront être justifiés en explicitant des bijections avec des ensembles dont le cardinal est connu. Vous pouvez admettre que* $|\mathfrak{S}_n| = n!$ *et que le nombre de parties à* k *éléments d'un ensemble à* n *éléments est* $\binom{n}{k} = \frac{n!}{k!(n-k)!}$ *(si* $0 \leq k \leq n$*).*

Pour calculer la hauteur moyenne des ABR de taille n, nous allons supposer que l'ordre d'arrivée des données suit la loi uniforme : on s'intéresse donc à l'espérance de la variable aléatoire $\mathbb{E}[h(A_n^\sigma)]$ où σ est une v.a. sur $\mathfrak{S}_n$ de loi $\mathcal{U}(\mathfrak{S}_n)$.

T2. Soit σ une v.a. sur $\mathfrak{S}_n$ de loi $\mathcal{U}(\mathfrak{S}_n)$. Montrer que pour tout entier $0 \leq k \leq n-1$, $\sigma(k)$ est une v.a. de loi $\mathcal{U}([\![0, n-1]\!])$.

T3. Soit σ une v.a. sur $\mathfrak{S}_n$ de loi $\mathcal{U}(\mathfrak{S}_n)$. Montrer que, pour tout $1 \leq k \leq n-1$, pour tout entiers de $[\![0, n-1]\!]$ deux-à-deux distincts $j_0, ..., j_k$,

$$\mathbb{P}(\sigma(k) = j_k | \sigma(0) = j_0, ..., \sigma(k-1) = j_{k-1}) = \frac{1}{n-k}. \tag{1}$$

T4. Soit σ une v.a. sur $\mathfrak{S}_n$ telle que la loi de $\sigma(0)$ est $\mathcal{U}([\![0, n-1]\!])$ et telle que (1) soit vraie pour tout $1 \leq k \leq n-1$ et tout $j_0, ..., j_k$ distincts deux-à-deux. Montrer que σ suit la loi $\mathcal{U}(\mathfrak{S}_n)$.

S7. Exploiter **T4** pour écrire un programme qui simule une variable aléatoire de loi $\mathcal{U}(\mathfrak{S}_n)$, et l'implémenter (le temps d'exécution de la méthode ne doit pas être de l'ordre factorielle n...).
Utiliser votre programme pour tracer l'histogramme de la répartition empirique de la longueur du cycle de $\sigma \sim \mathcal{U}(\mathfrak{S}_n)$ contenant 0, pour $n = 50$ avec un échantillon de taille 1000.

S8. En déduire une méthode pour simuler l'ABR A_n^σ où σ suit la loi $\mathcal{U}(\mathfrak{S}_n)$, et l'implémenter.
Utiliser votre programme pour tracer l'histogramme de la répartition empirique

de l'étiquette du fils gauche de la racine de A_n^σ avec $\sigma \sim \mathcal{U}(\mathfrak{S}_n)$ (si la racine n'a pas de fils gauche, nous dirons que la valeur de l'étiquette est -1), pour $n = 50$ avec un échantillon de taille 1000.

3 Étude de la hauteur moyenne d'un ABR

Let but principal de cette section est de montrer le résultat suivant

Théorème. *Soit σ une v.a. sur $\mathfrak{S}_n$ de loi $\mathcal{U}(\mathfrak{S}_n)$. Alors*

$$\mathbb{E}[h(A_n^\sigma)] \leq 3\log_2(n+3),$$

où $\log_2$ désigne le logarithme binaire.

Pour montrer ce résultat, il est utile de remarquer que l'ABR A_n^σ (où σ est un élément quelconque de $\mathfrak{S}_n$) s'écrit comme en Figure 3, où v_σ est une étiquette, et A_-^σ et A_+^σ sont des ABR (éventuellement vides).

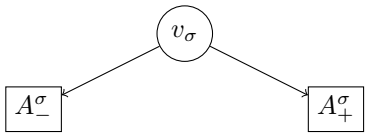

FIGURE 3 – Représentation de A_n^σ

Dans la suite, on note K_σ le nombre de sommets dans l'arbre A_-^σ.

T5. Soit $\sigma \in \mathfrak{S}_n$. Exprimer le nombre de sommets de A_+^σ en fonction de n et K_σ. Exprimer v_σ et K_σ en fonction de σ.

T6. Si σ est un v.a. de loi $\mathcal{U}(\mathfrak{S}_n)$, quelle est la loi de K_σ ?

T7. Soit $\sigma \in \mathfrak{S}_n$. Montrer qu'il existe $l_\sigma \in \mathbb{N}$ et $\sigma_- \in \mathfrak{S}(E_\sigma, [\![0, l_\sigma - 1]\!])$ (où E_σ est un ensemble de cardinal l_σ à déterminer) tels que $A_-^\sigma = A_{l_\sigma}^{\sigma_-}$.
Indice : Rappelons que la notation A_n^τ (pour des $\tau \in \mathfrak{S}(E, F)$) est définie à la fin de la Section 1. Avant de faire cette question, assurez-vous d'avoir bien compris la Remarque de la fin de la Section 1 (il s'agit principalement d'un jeu de notation).

T8. Soit σ une v.a. de loi $\mathcal{U}(\mathfrak{S}_n)$. Montrer que, pour tout entier $0 \leq k \leq n-1$ et $\gamma \in \mathfrak{S}_k$,

$$\mathbb{P}(\{\tilde{\sigma}_- = \gamma\} \cap \{K_\sigma = k\}) = \frac{1}{k!} \times \frac{1}{n}.$$

Rappel : pour $\tau \in \mathfrak{S}(A, B)$, $\tilde{\tau}$ est définie à la fin de la Section 1.

T9. Soit $k \in [\![0, n-1]\!]$. Montrer que, conditionnellement à $\{K_\sigma = k\}$, la variable $h(A_-^\sigma)$ a même loi que $h(A_k^\tau)$ (avec τ une variable aléatoire de loi $\mathcal{U}(\mathfrak{S}_k)$). Via un raisonnement similaire, que dire de la loi de $h(A_+^\sigma)$ conditionnellement

à $\{K_\sigma = k\}$?

T10. Exprimer $h(A_n^\sigma)$ en fonction de $h(A_-^\sigma)$ et $h(A_+^\sigma)$. En déduire que la suite $h_n = \mathbb{E}\left[2^{h(A_n^\sigma)}\right]$ (où $\sigma \sim \mathcal{U}(\mathfrak{S}_n)$) satisfait :

$$h_n \leq \frac{4}{n}\sum_{k=0}^{n-1} h_k.$$

Indice : Vous pouvez partitionner l'univers en fonction de la valeur de K_σ, et admettre la formule de l'espérance totale (i.e. si A est un événement tel que ni A ni A^c ne sont négligeables, et si X une v.a.r., alors $\mathbb{E}[X] = \mathbb{P}(A)\mathbb{E}[X|A] + \mathbb{P}(A^c)\mathbb{E}[X|A^c]$ où $\mathbb{E}[\bullet|A]$ désigne l'espérance associée à la mesure de probabilités $\mathbb{P}(\bullet|A)$).

T11. Montrer par récurrence forte que, pour tout entier positif n, $h_n \leq \binom{n+3}{3}$.
Indice : vous pouvez utiliser la formule du triangle de Pascal $\binom{m}{j} = \binom{m+1}{j+1} - \binom{m}{j+1}$

T12. Démontrer le théorème énoncé au début de cette section.
Indice : Commencer par déduire de **T11** que $h_n \leq (n+3)^3$.

S9. Donner une valeur approchée de $\mathbb{E}[h(A_{10}^\sigma)]$ où σ est une v.a. de loi $\mathcal{U}(\mathfrak{S}_{10})$. On garantira que l'erreur commise est inférieure 1, avec probabilité supérieure à 0.9.
Indice : on pourra utiliser la borne (grossière) suivante $\mathbb{E}[h(A_n^\sigma)^2] \leq n^2$

S10. Tracer l'histogramme de la répartition empirique de $h(A_{100}^\sigma)$ (où σ est une v.a. de loi $\mathcal{U}(\mathfrak{S}_{100})$). Vous pouvez prendre un échantillon de taille 1000.

4 Application au tri de données

Un ABR est une structure dont on peut se servir pour stocker des données triées. Une conséquence simple (que nous ne chercherons pas à démontrer ici) du théorème de la section précédente, est que la complexité moyenne de l'algorithme de construction d'un ABR est inférieure à $3n\log_2(n+3) = \Theta(n\ln n)$ (ce qui est l'ordre de grandeur de la complexité optimale pour les problèmes de tri), en supposant que l'ordre d'insertion des données est uniforme sur l'ensemble des permutations. Toutefois il existe des cas où cette complexité est $\Theta(n^2)$. En particulier, si les données sont fournies par une personne malveillante (i.e. une personne faisant exprès de choisir une "mauvaise" permutation σ de l'ordre des données), elle peut forcer une implémentation naïve du tri à avoir une complexité de l'ordre de n^2 (ce qui est a priori le cas du programme de la question **S6**).

T13. Pour n'importe quel entier $n \geq 1$, expliciter un élément σ de $\mathfrak{S}_n$ tel que $h(A_n^\sigma) = n-1$.

T14. Soient σ, τ deux v.a. indépendantes définies respectivement sur $\mathfrak{S}(E,E)$ et $\mathfrak{S}([\![0,n-1]\!],E)$ (où E est un ensemble de cardinal n) telles que la loi de σ est $\mathcal{U}(\mathfrak{S}(E,E))$. Quelle est la loi de $\sigma \circ \tau$?

S11. Écrire un programme qui étant donné une "List" de données, construit un ABR contenant ces données. La hauteur moyenne de cet ABR doit être de l'ordre de $\log_2 n$ (où n est le nombre de données), et votre programme doit être robuste à la "malveillance" (i.e. la hauteur moyenne de votre ABR ne doit pas dépendre de la permutation donnée en entrée).
Utiliser votre programme pour construire un ABR dont les étiquettes sont les 100 nombres premiers les plus petits (*Indice :* le centième nombre premier est 541). Tracer l'histogramme de la répartition empirique de la hauteur de cet ABR pour un échantillon de taille 1000 (et comparé le avec celui de **S10**).

Propriétés spatiales des tables de hachage universel

1 Table de hachage : présentation

En informatique, il est souvent utile de pouvoir manipuler un ensemble de données (que l'on peut implémenter comme une "List" en Python) pouvant exécuter les opérations de bases suivantes (entre autres...) :

— accès à un élément déjà présent,
— modification d'un élément déjà présent
— ajout d'un nouvel élément.

Dans beaucoup de langages de programmation (notamment parmi ceux plus bas niveau que Python), il y a deux implémentations "naturelles" du type "List" de Python : les tableaux et les listes chaînées (ou piles). Si on dispose de n données, l'avantage des tableaux est que les opérations "accès à un élément" et "modification d'un élément" sont rapides (le cout est proche de $O(1)$), mais que l'opération "ajout d'un élément" est lente (de l'ordre de $O(n)$), ce qui est exactement l'inverse des listes chaînées. Le but de ce projet est d'étudier un compromis entre les tableaux et les listes chaînées : les tables de hachage.

Dans les trois premières sections du projet, nous définissons formellement une table de hachage comme un tableau contenant des listes chaînées, ce que l'on implémentera en Python comme une List de List. Chaque liste chaînée sera appelée une alvéole. Le nombre d'alvéoles d'une table de hachage donnée est non-modifiable (en particulier, si vos programmes le modifient, alors ils ne seront pas corrects) : vous ne pouvez pas ajouter d'alvéole. Mais vous pouvez ajouter des éléments dans les alvéoles déjà présentes. Autrement dit, vous n'avez pas le droit d'utiliser la commande "append" pour ajouter une alvéole, mais vous avez le droit de l'utiliser pour ajouter un élément dans une alvéole. Ci-dessous, t est une table de hachage implémentée en python à 4 alvéoles : la première contient les éléments 2,8,6, la deuxième est vide, la troisième contient seulement l'élément 1, et la quatrième contient les éléments 5,4,15,9.

```
t = [[2,8,6],
     [],
     [1],
     [5,4,15,9]]
```

Remarque. *Pour créer une table de hachage m à 4 alvéoles (par exemple) initialement vides (sans utiliser la fonction append), vous pouvez utiliser le code ci-dessous*

```
t0 = [[] for i in range(4)]
```

Étant donné un ensemble de données $\mathcal{D}$, pour construire une table de hachage à m alvéoles contenant n données $d_0, ..., d_{n-1} \in \mathcal{D}$, il faut définir ce que l'on appelle une fonction de hachage $h : \mathcal{D} \to [\![0, m-1]\!]$, qui étant donné une donnée d à stocker dans la table, renvoit l'alvéole $h(d)$ dans laquelle d sera insérée (on dit que d est hachée dans l'alvéole $h(d)$). Par exemple, avec l'exemple de la table de hachage t ci-dessus, on a $m = 4$ alvéoles, $\mathcal{D} = \mathbb{N}$ et la fonction de hachage h utilisée vérifie : $h(2) = h(8) = h(6) = 0$, $h(1) = 2$ et $h(5) = h(4) = h(15) = h(9) = 3$.

S1. Écrire un programme qui, étant donné une table de hachage, une fonction de hachage et une donnée, modifie la table en insérant la donnée dans la bonne alvéole. Vous pouvez supposer que les données sont de type Int.
Appliquer votre programme pour insérer la donnée 7 via la fonction de hachage $h(n) = \lfloor n/4 \rfloor$ (où $\lfloor x \rfloor$ est la partie entière de x) dans la table de hachage suivante (et afficher la table ainsi obtenue)

```
[[1,3,2,0],
[6],
[],
[],
[17,18]]
```

S2. Écrire un programme qui étant donné une table de hachage (implémentée comme expliqué précédemment), une fonction de hachage et une donnée, teste si la donnée est présente dans la table. Vous pouvez supposer que les données sont de type Int.
Vérifier votre programme en testant successivement la présence des données 7, 15 et 18 dans la table obtenue à la fin de **S1**.

S3. Écrire un programme qui, étant donné une List d'entiers compris entre 0 et 19, construit la table de hachage à 5 alvéoles associée à la fonction de hachage $h(n) = \lfloor n/4 \rfloor$, telle que chaque donnée ne soit présente qu'une fois dans la table (même si la donnée est présente en double dans la List). Il ne sera pas nécessaire de vérifier que les entiers de la List sont compris entre 0 et 19.
Utiliser votre programme pour constuire la table (que vous afficherez) otenue par insertion successives des données suivante (dans cet ordre) : 5, 1, 7, 6, 5, 9, 15, 0, 18.

Dans toute la suite, on supposera que les tables de hachages ne contiennent pas de doublon.

La recherche d'une donnée d dans une table de hachage a l'avantage que vous n'avez pas besoin de rechercher parmi toutes les données stockées, mais seulement parmi celles qui sont hachées dans l'alvéole $h(d)$. C'est donc sans surprise que les complexités moyennes des algorithmes sur les tables de hachage dépendent du facteur de remplissage $\alpha = n/m$ (avec n le nombre de données stockées et m le nombre d'alvéoles). Dans une table de hachage "bien construite", toutes les alvéoles ont une taille proche de α. Quand m est petit (par exemple $m = 1$ et $\alpha \gg 1$), on ne profite pas de la structure de la table de hachage pour

optimiser le temps d'exécution des programmes. Inversement, si $m \gg n$ (et donc $\alpha \ll 1$), il y a beaucoup d'alvéoles vides, ce qui est un gachis de ressources en espace.

Le but du projet est d'étudier des méthodes de construction de tables de hachage qui ont de bonnes propriétés en moyenne.

2 Hypothèse de hachage uniforme simple

Dans toute cette section, nous noterons $h : \mathcal{D} \to [\![0, m-1]\!]$ la fonction de hachage (déterministe), et $D_0, ..., D_{n-1}$ les données (v.a. sur $\mathcal{D}$) qui sont insérées successivement dans une table de hachage à m alvéoles initialement vide. Soit T la table de hachage ainsi obtenue. Pour $0 \leq j \leq m-1$, nous noterons N_j la taille de l'alvéole j :

$$N_j = |\{0 \leq i \leq n-1 \ : \ h(D_i) = j\}|.$$

Définition. *Nous dirons que $(T, D_0, ..., D_{n-1})$ satisfait l'hypothèse de hachage uniforme simple (HHUS) si les variables $h(D_i)$ $(0 \leq i \leq n-1)$ sont i.i.d. de loi $\mathcal{U}([\![0, m-1]\!])$, et si, avec probabilité 1, les variables D_i $(0 \leq i \leq n-1)$ sont distinctes deux-à-deux.*

En théorie, la seconde condition de l'HHUS n'est pas nécessaire, mais elle permet d'éviter des difficultés techniques sans grande importance dans les questions de cette section.

T1. Supposons que $D_0, ..., D_{n-1}$ sont i.i.d. de loi $\mathcal{U}([0,1[)$, et $h : d \in [0,1[\mapsto \lfloor dm \rfloor$. Montrer que l'HHUS est vérifiée.

S4. La table de hachage obtenue à **T1** est aléatoire (car les données sont des v.a.). Écrire un programme qui simule cette table de hachage aléatoire avec $m = 5$ alvéoles en y insérant $n = 1000$ données i.i.d. de loi $\mathcal{U}([0,1[)$. Utiliser votre programme pour simuler une telle table, puis afficher la taille de ses alvéoles à l'aide d'un histogramme.

T2. Supposons que l'HHUS est vérifiée. Pour $0 \leq i, j \leq n-1$, donner la probabilité que les données D_i et D_j soient hachées dans la même alvéole. Donner la loi de N_j $(0 \leq j \leq m-1)$.
<u>*Indice :*</u> Pour la deuxième partie de la question, commencer par montrer que, pour tout $0 \leq j \leq m-1$,

$$N_j = \sum_{i=0}^{n-1} 1_{\{h(D_i)=j\}}.$$

Dans les deux questions suivantes, nous considérons l'algorithme de recherche de données de **S2**. Pour estimer le nombre de comparaisons que doit effectuer

cet algorithme pour tester si une donnée d est présente dans la table T, il y a deux scénarios :

- soit d n'est pas dans la table, alors le nombre de comparaisons est le nombre total de données insérées dans l'alvéole $h(d)$ (car il faut parcourir toute l'avéole pour être sûr que d n'est pas présente),
- soit d est dans la table, alors le nombre de comparaisons est le nombre de données qui ont été insérées dans l'alvéole $h(d)$ avant d, ce qui inclut d (car une fois que d est trouvée, on peut arrêter la recherche).

T3. Supposons que l'HHUS est vérifiée. Soit D une v.a. sur $\mathcal{D}$ qui est presque sûrement différente de toutes les D_i $(0 \leq i \leq n-1)$. Quelle est l'espérance du nombre de comparaisons de données qui devront être faîtes par l'algorithme de **S2** qui teste si D est présente dans la table H ?
Indice : commencer par donner le nombre exact de comparaisons (qui est aléatoire) en utilisant l'indice de **T2**.

T4. Supposons que l'HHUS est vérifiée. Soit I une v.a. sur $[\![0, n-1]\!]$. Quelle est l'espérance du nombre de comparaisons qui devront être faîtes par l'algorithme de **S2** qui teste si D_I est présente dans la table H ?
Indice : même indice que pour **T3**.

3 Méthode de hachage universel

Étant donné un ensemble $\mathcal{D}$ et un entier $m \geq 1$, on dit qu'une collection finie de fonctions $h_j : \mathcal{D} \to [\![0, m-1]\!]$ $(0 \leq j \leq q-1)$ est une collection de fonctions de hachage universelle si, pour tout $d \neq d'$, le nombre de fonctions h_j $(0 \leq j \leq q-1)$ telles que $h_j(d) = h_j(d')$ est inférieur à q/m.

Dans toute cette section, nous noterons $h_0, ..., h_{q-1}$ une collection de fonctions de hachage universelle (déterministe), et $d_0, ..., d_{n-1}$ les données (éléments déterministes de $\mathcal{D}$) qui sont insérées successivement dans une table de hachage à m alvéoles initialement vide, où la fonction de hachage utilisée est h_I avec I v.a. de loi $\mathcal{U}([\![0, q-1]\!])$ (c'est la même fonction h_I qui est utilisée pour hacher toutes les données d_i dans la table). Nous noterons T la table ainsi obtenue. Nous supposons que les données d_i $(0 \leq i \leq n-1)$ sont deux-à-deux distinctes. Pour $0 \leq j \leq m-1$, nous notons encore N_j la taille de l'alvéole j

$$N_j = |\{0 \leq i \leq n-1 \ : \ h_I(d_i) = j\}|.$$

Nous appellerons méthode de hachage universel, cette méthode de construction.

T5. Soit $d \in \mathcal{D}$ différente des données $d_0, ..., d_{n-1}$. Montrer que l'espérance du nombre de comparaisons qui devront être faîtes par l'algorithme qui teste si d est présente dans la table H est inférieure au facteur de remplissage $\alpha = n/m$.

T6. Soit $i \in [\![0, n-1]\!]$. Montrer que l'espérance du nombre de comparaisons qui devront être faîtes par l'algorithme qui teste si d_i est présente dans la table H est inférieure à $1 + \alpha$.

3.1 Application avec des données de $(\mathbb{Z}_p)^l$

Dans la suite, pour tout entier $p \geq 2$, nous notons $\mathbb{Z}_p$ (aussi courament noté $\mathbb{Z}/p\mathbb{Z}$) l'ensemble des classes d'équivalence pour la relation de congruence modulo p : deux entiers sont équivalents si et seulement s'ils ont le même reste dans la division euclidienne par p. On admettra que si p est un nombre premier, alors $\mathbb{Z}_p$ (muni de l'addition et de la multiplication habituelles) est un corps.

T7 (facultative). Soient $l \in \mathbb{N}^*$, p un nombre premier et $\mathcal{D} = (\mathbb{Z}_p)^l$. Pour $a \in \mathbb{Z}_p$, soit

$$h_a : d = (d_0, ..., d_{l-1}) \in \mathcal{D} \mapsto \sum_{j=0}^{l-1} d_j a^j \in \mathbb{Z}_p.$$

Étant donné deux éléments distincts b, c de $\mathcal{D}$, montrer qu'il y a au plus $l - 1$ éléments $a \in \mathbb{Z}_p$ tels que $h_a(b) = h_a(c)$.

T8. Soient p un nombre premier et $\mathcal{D} = (\mathbb{Z}_p)^2$. Donner un entier m et une collection de fonctions de hachage universelle.

S5. Écrire un programme qui, étant donné une List d'éléments de $(\mathbb{Z}_{11})^2$ (un élément de $(\mathbb{Z}_{11})^2$ pourra être implémenté par une List de longueur 2 ou par un couple), construit une table de hachage par une méthode de hachage universelle. Utiliser votre programme pour construire la table (puis afficher la) contenant les données suivantes

```
dS5 = [[5,8],[0,0],[3,1],[10,5],[6,2],[1,5]]
```

3.2 Contrôle du nombre de collisions

Par définition, les collisions dans une table de hachage sont les paires de données hachées dans une même alvéole. L'ensemble $\mathcal{C}$ des collisions est formellement décrit de la façon suivante

$$\mathcal{C} = \{\{k, l\} \subseteq [\![0, n-1]\!] \ : \ k \neq l \text{ et } h_I(d_k) = h_I(d_l)\}.$$

On notera X le nombre de collisions dans la table T.

T9. Supposons $m \geq n^2$. Montrer que l'espérance du nombre de collisions X dans la table T est strictement inférieure à $1/2$. En déduire que la probabilité qu'il n'y ait aucune collision est strictement supérieure à $1/2$.
Indice : On pourra exploiter (et réarranger la somme)

$$X = \sum_{\{k,l\} \subseteq [\![0,n-1]\!]} 1_{\{k,l\} \in \mathcal{C}}.$$

T10. Si $m \geq n^2$, étant donné n données $d_0, ..., d_{n-1}$, on cherche une fonction de hachage h parmi $h_0, ..., h_{q-1}$ qui n'a aucune collision. Pour ça, on en choisit une au hasard (uniformément) parmi les fonctions $h_0, ..., h_{q-1}$ jusqu'à en trouver une sans collision (si la fonction choisie a une collision, il faut en rechoisir une aléatoirement indépendamment des tirages précédents). Donner la loi du nombre de tirages que vous devrez faire avant de trouver une fonction de hachage sans collision (le résultat dépend de la probabilité d'avoir une collision : $\mathbb{P}(X \geq 1)$, que l'on ne cherchera pas à expliciter). Montrer que la probabilité que le nombre de tirages soit strictement supérieur à k, est inférieure à 2^{-k}.

S6. En déduire un programme qui, étant donné une List de $n \leq 14$ éléments de $(\mathbb{Z}_{211})^2$, construit une table de hachage à $m = 211$ alvéoles sans aucune collision (i.e. toutes les alvéoles sont de taille 0 ou 1).
Utiliser votre programme pour construire une table sans collision contenant les données suivantes

```
dS6 = [[5,8],[0,0],[3,1],[10,5],[6,2],[1,5],[4,2],[5,7],[3,5],[6,9],[0,2]]
```

L'avantage de la table de hachage construite à **S6** (qui suit la méthode de **T10**) est que toutes les alvéoles sont de tailles 0 ou 1. L'inconvénient est l'espace gâché : le facteur de remplissage est $1/n$ qui peut être proche de zéro. Dans la section suivante, nous verrons comment exploiter **T10** sans avoir un facteur de remplissage qui tend vers zéro quand n tend vers l'infini.

4 Méthode de hachage parfait

L'idée du hachage parfait est de construire une table de hachage dont les alvéoles sont des tables de hachage sans collision. Il y a donc deux niveaux de hachage. La fonction de hachage pour la table principale sera choisie selon une méthode de hachage universel (comme décrit dans la section précédente). Pour chaque alvéole j, il faut choisir une fonction de hachage $h^{[j]}$ pour garantir l'absence de collision (en suivant les questions **T10** et **S6**).

Pour construire une table de hachage selon la méthode du hachage parfait, il faut connaître dès le départ l'ensemble des données à stocker. Une fois la table construite, il n'est pas possible d'insérer a posteriori de nouveaux éléments tout en garantissant l'absence de collisions au second niveau de hachage.

Nous notons m le nombre d'alvéoles de la table de hachage, $d_0, ..., d_{n-1}$ les données à stocker, et N_j le nombre de données hachées dans l'alvéole j. Avant d'insérer les données dans la table à m alvéoles, il faut choisir le nombre d'alvéoles M_j de l'avéole j (car chaque alvéole est une table de hachage) : les nombres M_j ne sont pas modifiables.

Le but de la question suivante est d'écrire un programme qui calcule pour chaque alvéole j, le nombre de données N_j qui seront hachées dans cet alvéole.

S7. Écrire un programme qui, étant donné un entier m, une List de données et une fonction de hachage à valeurs dans $[\![0, m-1]\!]$, renvoit une List l de taille m telle que $l[j]$ est le nombre de données hachées dans l'alvéole j (votre programme n'est pas censé construire de table de hachage).
Afficher sous la forme d'un histogramme la taille des alvéoles de la table construite à **S5** (sans re-construire la table).

Dans le contexte du hachage parfait, on connait à l'avance le nombre de données N_j stockées dans chaque alvéole j. Le principe de la méthode de hachage parfait est de garantir l'absence de collision au sein des alvéoles. Une condition nécessaire évidente pour avoir cette propriété est "pour tout $0 \leq j \leq m-1$, $M_j \geq N_j$" (par le principe des tiroirs). Toutefois, si les fonctions de hachage sont choisies parmi une collection universelle $\mathcal{H}$, le choix de $M_j = N_j$ ne permet pas de garantir qu'il existe une fonction $h^{[j]} \in \mathcal{H}$ telle que $h^{[j]}$ n'a pas de collision pour les N_j données à insérer dans l'alvéole j (qui contient M_j alvéoles). Il faut donc choisir des tailles d'alvéoles M_j apprioriées.

Conformément à ce qui a été montré à la question **T10**, nous choisirons d'implémenter l'alvéole j par une table de hachage de taille $M_j = N_j^2$ associée à une fonction de hachage $h^{[j]}$ choisie de telle sorte à ce qu'il n'y ait aucune collision dans la table de hachage de l'alvéole j. La taille totale de la table de hachage est alors $\sum_{j=0}^{m-1} N_j^2$.

On peut penser qu'il y a un gachis d'espace important (car l'alvéole j est de taille N_j^2 pour stocker N_j données). En particulier, si toutes les n données sont hachées dans la même alvéole de la table principale, alors la taille totale est n^2, et le facteur de remplissage est $1/n$ comme dans la question **S6**.

Les deux questions suivantes montrent que la taille totale de la table est "souvent" d'ordre n.

T11. En supposant que $m \geq n$, montrer que l'espérance de la taille totale (i.e. la somme des tailles des alvéoles) de la table construite par hachage parfait est strictement inférieure à $2n$.
Indice : Montrer que $N_j^2 = N_j + 2X_j$ avec X_j le nombre de collisions dans la table de hachage principal venant de l'alvéole j.

T12. En supposant que $m \geq n$, montrer que, pour tout $0 < \varepsilon < 1$, il existe un entier $t(\varepsilon)$ tel que, avec probabilité supérieure à $1-\varepsilon$, la taille totale de la table de hachage ne dépasse pas $nt(\varepsilon)$.

S8. Donner une valeur approchée de l'espérance de la taille totale de la table de hachage à $m = 11$ alvéoles construite par la méthode de hachage parfait (la fonction de hachage principale est choisie uniformément parmi les fonctions h_a définies à **T7** avec $p = 11$ et $l = 2$) via la liste de données `lEx` définie ci-dessous (qui sont des éléments de $\mathbb{Z}_{11}$). On garantira que l'erreur commise est inférieure 4, avec probabilité supérieure à 0.5.
Indice : démontrer (et exploiter) que la variance de la taille totale de la table

est inférieure à $m \cdot n^4$ (c'est une borne relativement grossière)

```
lEx = [[5,8],[0,0],[3,1],[10,5],[6,2],[1,5],[4,7],[2,2],[10,7],[5,4]]
```

S9. Tracer un histogramme de la répartition empirique de la taille totale de la table de hachage (avec les mêmes paramètres que dans la question **S8**). Vous pouvez prendre un échantillon de taille 10000.