❑ **Exercice 1** : *complexité des algorithmes de recherche*
Dans tout cet exercice, on exprime la complexité en nombre de comparaisons effectués par l'algorithme. Et note t la longueur du texte et m la longueur du motif.

1. Montrer que la recherche naïve effectue entre t et $t \times m$ comparaisons et donner des exemples dans laquelle ces deux bornes sont atteintes.
2. Déterminer de même les bornes du nombre de comparaisons effectuées par l'algorithme de Boyer-Moore-Hoorspool et donner un exemple où elles sont atteintes.

❑ **Exercice 2** : *Algorithme naïf avec motif à caractère unique*

1. Montrer qu'il est possible d'améliorer l'algorithme de recherche naïf si on suppose que tous les caractères du motifs apparaissent une seule fois dans le motif.
   ⊗ Faire par exemple la recherche de `abcd` dans le texte `abceababccabcdabb` et observer ce qu'il se passe lorsqu'on trouve un début de correspondance.
2. Donner une implémentation en C ou en OCaml de ce nouvel algorithme.

❑ **Exercice 3** : *Dérouler l'algorithme de Boyer-Moore-Hoorspool*

1. Donner la table de décalage du motif `tele`
2. Donner les étapes du déroulement de l'algorithme de Boyer-Moore-Hoorspool pour recherche toutes les occurrences de ce motif dans le texte : `le telephone ou la television`

❑ **Exercice 4** : *Rabin-Karp à deux motifs*

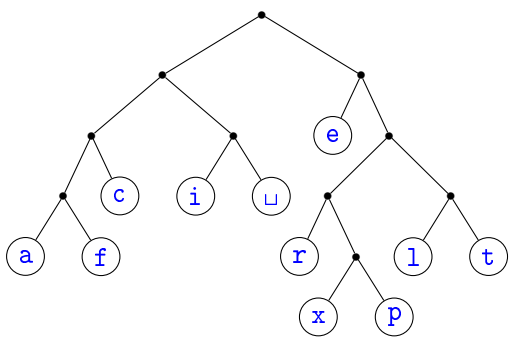
1. Ecrire une fonction `hash : string -> int` qui effectue la moyenne des code ASCII de la chaine donnée en argument en leur affectant pour coefficient leur position dans la chaine (le coefficient du premier caractère est 1).
2. Ecrire l'implémentation de l'algorithme de Rabin-Karp avec cette fonction de hachage afin de rechercher un motif dans un texte.
3. Adapter votre algorithme de façon à pourvoir rechercher les occurences de deux motifs `m1` et `m2` qu'on suppose de même longueur.
4. Proposer une nouvelle modification si les motifs sont de taille différentes.

❑ **Exercice 5** : *dérouler l'algorithme de Huffman*
On considère la phrase : « *comprendre cet algorithme* ».

1. Faire une tableau indiquant pour chaque caractère son nombre d'occurrence dans la phrase.
2. Dérouler l'algorithme de Huffman afin de construire l'arbre de codage préfixe.
3. Donner les codes de chaque caractère.
4. Déterminer le taux de compression de l'algorithme sur cet exemple.

❑ **Exercice 6** : *Quelques arbres de Huffmann*

1. Donner l'arbre de Huffman obtenu pour un texte contenant 10 caractères ayant tous le même nombre d'occurences n.
2. Donner l'arbre de Huffman obtenu pour un texte contenant 7 caractères dont les nombres d'occurences sont 1, 2, 4, 8, 16, 32 et 64.
3. On donne l'arbre de Huffmann suivant :



Donner les codes des caractères présents dans l'arbre.

4. Donner le texte dont la compression est :
   `11101001111011101111010111101110110101011000010100011001100010000001010111010`
5. Calculer la taille initiale du texte et la taille du texte compressé.

❑ **Exercice 7** : *Séquence génétique*

On rappelle que le principe de la compression LZW est d'attribuer un code aux préfixes rencontrés lors de la lecture du texte à compresser de façon à disposer d'un code compact si ce code se présente à nouveau. Initialement, seuls les codes de l'alphabet (usuellement les caractères ASCII) sont présents dans la table de codage.

Ici, on s'intéresse à des chaines de caractères représentant des séquences génétique codés sur l'alphabet `{A, C, G, T}`, par souci de simplicité on attribue initialement les codes `A` $\to 0$, `C` $\to 1$, `G` $\to 2$ et `T` $\to 3$. Et on souhaite compresser la séquence `ATGTGATGTCCT`. Le début de l'algorithme va donc consister à attribuer un nouveau code au premier préfixe non encore codé qui apparaît lors de la lecture du texte. Et donc, ici, on attribue le code `4` au prefixe `AT` et on emmettra le code de `A` c'est à dire 0.

1. Poursuivre le déroulement de cet algorithme en complétant le tableau suivant :

Position dans le texte	Code émis	Nouveau préfixe ajouté
`A`TGTGATGTCCT	0	`AT` $\to 4$

2. Quel est le taux de compression obtenu (la taille d'un code est un octet) ?
3. Décompresser le texte T codé par suite de codes `[3; 1; 4; 6; 5; 2; 0; 2 ]` sur ce même alphabet en expliquant comment est reconstruit le dictionnaire de décompression.

❑ **Exercice 8** : *Algorithme* LZW

1. Décrire le fonction de l'algorithme LZW sur une chaine qui contient n caractères identiques.
2. On considère la compression d'un texte contenant 5 caractères différents. Donner une chaîne de caractères de longueur 20 pour laquelle l'algorithme LZW ne réduit pas la taille du résultat par rapport à l'entrée.

❑ **Exercice 9** : *diminution de la taille*

Peut-on écrire un proramme qui prend en entrée un fichier et renvoie une version compressée (sans perte d'information) de taille *strictement* inférieure de ce fichier ?