MINISTÈRE
DE L'ÉDUCATION
NATIONALE,
DE L'ENSEIGNEMENT
SUPÉRIEUR
ET DE LA RECHERCHE

Liberté
Égalité
Fraternité

Concours externe BAC + 3 du CAPES

Cafep-Capes

Section Numérique et sciences informatiques

1) Exemple de sujet pour la première épreuve d'admissibilité
2) Extrait de l'arrêté du 17 avril 2025

Les épreuves des concours externes du Capes et du Cafep-Capes BAC +3 sont déterminées dans l'arrêté du 17 avril 2025 fixant les modalités d'organisation du concours externe du certificat d'aptitude au professorat de l'enseignement du second degré, publié au Journal Officiel du 19 avril 2025, qui fixe les modalités d'organisation du concours et décrit le schéma des épreuves.

CAPES BAC +3

Sujet 0 / Première épreuve d'admissibilité

Ce sujet comporte deux parties indépendantes.

Partie 1. Palindrome

On dit qu'un mot est un **palindrome** s'il se lit de la même façon de gauche à droite et de droite à gauche. Par exemple *kayak*, *radar*, *été* ou *ada* sont des palindromes. Afin de rendre la vie plus palindromique, on souhaite transformer des mots (chaînes de caractères) en palindromes en ajoutant des lettres aux bons endroits. Par exemple, à partir du mot *aab*, on rajoute *baa* à la fin du mot pour former le palindrome *aabbaa*. On a rajouté 3 lettres, mais en rajoutant *b* devant le mot on obtient *baab* qui est également un palindrome de taille plus petite.

L'objectif de l'exercice est de trouver le nombre minimum de lettres à insérer dans un mot pour former un palindrome, puis de déterminer les lettres à insérer et les positions où elles seront insérées.

La fonction `palindrome_string` ci-dessous propose de tester si un mot est un palindrome en comparant le mot avec le mot retourné. L'opération `"".join(li)` effectue la concaténation des caractères de la liste de caractères `li` sans séparateur. L'opérateur `==` permet de tester l'égalité de deux chaînes de caractères.

```
def palindrome_string(mot):
    """
    Teste si mot est un palindrome.
    Paramètre: mot (str): chaine de caractères
    Renvoie: (bool): True (mot est un palindrome)
                     False (mot n'est pas un palindrome)
    """
    li = list(mot)
    n = len(li)
    for k in range(n):
        li[k], li[n-1-k] = li[n-1-k], li[k]
    mot_retourne = "".join(li)
    return mot ==  mot_retourne
```

Cette fonction `palindrome_string(mot)` renvoie la valeur `True` systématiquement, même si le `mot` n'est pas un palindrome.

Q. 1 Identifier l'erreur algorithmique dans la fonction `palindrome_string` et modifier cette fonction afin que le résultat soit correct.

La fonction précédente nécessite le calcul d'un nouveau mot (le mot retourné). On propose une autre fonction, `palindrome_indice`, testant si un mot est un palindrome et utilisant uniquement les indices des caractères du mot donné en paramètre.

```
def palindrome_indice(mot):
    """
    Teste si mot est un palindrome.
    Paramètre: mot (str): chaine de caractères
    Renvoie: (bool): True (m est un palindrome)
                     False (m n'est pas un palindrome)
    """
    n = len(mot)
    i = 0
    j = n - 1
    # point d'observation 1
    while i < j and mot[i] == mot[j]:
        # point d'observation 2
        i = i + 1
        j = j - 1
    # point d'observation 3
    return i >= j
```

Q. 2 Faire la trace de l'exécution de cette fonction aux points d'observation 1, 2 et 3 sur les chaînes de caractères suivantes : `"rever"`, `"elle"` et `"raler"` (les accents ont été supprimés pour rendre l'analyse plus simple).

Q. 3 Démontrer que la fonction `palindrome_indice` est correcte. Pour cela, on proposera, en le justifiant, un invariant de l'itération, puis un variant de l'itération et on conclura à la preuve de la correction.

Q. 4 Évaluer le coût au pire et au mieux de la fonction `palindrome_indice` en fonction de la taille du mot, le coût considéré étant le nombre de tests d'égalité de caractères effectués.

On veut maintenant écrire et étudier une version récursive de la fonction.

Q. 5 Écrire une fonction récursive `palindrome_rec` qui teste si un mot est un palindrome.

Q. 6 Démontrer que la fonction récursive proposée est correcte et évaluer son coût au pire et au mieux, toujours en termes de nombre de comparaisons de caractères.

On cherche maintenant à insérer des lettres dans le mot afin de le transformer en palindrome.

Q. 7 Déterminer combien de lettres il faut au minimum dans les mots suivants pour les transformer en palindromes : `"baba"`, `"reperer"` et `"beaute"`.

On considère un mot de longueur n dont les lettres sont indicées de 0 à $n-1$. Pour $0 \leqslant i \leqslant j < n$, on note `palindrome_min_lettres(mot, i, j)` le nombre minimal de lettres à ajouter pour transformer le sous-mot `mot[i]mot[i+1]...mot[j]` en palindrome.

Q. 8 Exprimer une relation de récurrence sur les valeurs `palindrome_min_lettres(mot, i, j)` avec $0 \leqslant i \leqslant j < n$. On précisera les conditions limites lorsque $i = j$.

Indication : on pourra distinguer les cas `mot[i]` $=$ `mot[j]` et `mot[i]` $\neq$ `mot[j]`.

Q. 9 Compléter la fonction récursive `palindrome_min_lettres` donnée ci-dessous. Les "..." peuvent représenter une expression ou une ou plusieurs ligne.

```
def palindrome_min_lettres(mot, i, j):
    """
    Renvoie le nombre minimal de lettres à insérer au sous-mot de mot de
    l'indice i à l'indice j (inclus) pour le transformer en palindrome.
    """
    if  ... : # cas de base
        return ...
    else: # cas général
        ...
        return ...
```

Q. 10 Dessiner l'arbre des appels associé à `palindrome_min_lettres("aabb",0,3)`.

Q. 11 Calculer le coût au pire et au mieux de l'appel `palindrome_min_lettres(mot, 0, n-1)`, en nombre de comparaisons de caractères, où `n` est la taille de `mot`.

L'arbre des appels met en évidence des appels redondants. Pour palier ce problème, on se propose d'utiliser une structure de donnée auxiliaire (par exemple un dictionnaire en Python) permettant d'associer à un ensemble de valeurs de paramètres `(mot, i, j)` la valeur de retour de l'appel de fonction `palindrome_min_lettres(mot, i, j)`.

Q. 12 Réécrire la fonction `palindrome_min_lettres(mot, i, j, d)` avec comme paramètre supplémentaire `d` la structure auxiliaire contenant les valeurs déjà calculées par la fonction lors de précédents appels. Préciser l'initialisation de la variable `d`.

Q. 13 Calculer le coût au pire et au mieux de l'appel `palindrome_min_lettres(mot, 0, n-1, d)`, en nombre de comparaisons de caractères, où `n` est la taille de `mot`.

Q. 14 Modifier la fonction `palindrome_min_lettres` pour qu'elle renvoie tous les palindromes de taille minimale contenant le mot initial.

Partie 2. Codage binaire d'arbres binaires

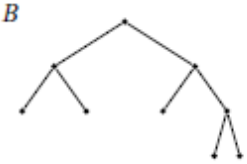
On considère des arbres binaires complets, c'est à dire que tout nœud est soit une feuille soit un nœud interne avec un sous-arbre gauche **et** un sous-arbre droit. La taille d'un arbre est son nombre de nœuds. À un arbre A on associe une séquence de bits $S(A)$, appelée *code* de A, définie de la manière suivante :

$$S(A) = \begin{cases} 1 & \text{si } A \text{ est réduit à une feuille,} \\ 0\,S(A_1)\,S(A_2) & \text{si } A_1 \text{ et } A_2 \text{ sont les sous-arbres gauche et droit de } A. \end{cases}$$

Q. 15 Donner la valeur de $S(A)$ et $S(B)$ pour les deux arbres suivants :

arbre A

arbre B



Q. 16 Dessiner les arbres correspondant aux séquences $m_1 = 000111011$ et $m_2 = 0101011$.

Pour m une séquence de bits on note $N_0(m)$ le nombre de bits "0" dans la séquence et $N_1(m)$ le nombre de bits "1".

Q. 17 Pour un arbre A de taille n, interpréter le nombre de 0 et de 1 dans la séquence $S(A)$. Formuler et justifier la relation qu'il y a entre le nombre de 0 et le nombre de 1.

Q. 18 Donner une séquence de bits qui n'est pas le code d'un arbre binaire complet, justifier.

Q. 19 Décrire un algorithme de codage d'un arbre binaire complet.

Pour une séquence m, on notera $|m|$ la longueur de la séquence. Les éléments de m sont repérés par de indices entre 1 et $|m|$. Pour un indice i dans cet intervalle, on désigne par $m[i]$ l'élément de m en position i. Pour deux indices i et j tels que $i \leqslant j$, on note $m[i \dots j]$ la sous-séquence $m[i]m[i+1]\dots m[j]$, de longueur $j - i + 1$.

On suppose disposer d'une fonction *fin-sous-arbre*(m, i) qui prend en arguments m, une séquence de bits, et un indice i et qui renvoie l'indice j tel que $m[i \dots j]$ est le code d'un arbre binaire complet, ou 0 s'il n'existe pas de tel indice. Par exemple, avec $m = 000111011$ et $i = 3$, la fonction renvoie 5 parce que la sous-séquence 011 correspond à un arbre binaire complet.

Q. 20 Décrire un algorithme de décodage qui à partir d'un code d'arbre reconstruit l'arbre correspondant, en utilisant la fonction *fin-sous-arbre*.

On cherche maintenant à réaliser la fonction *fin-sous-arbre*.

Q. 21 Soit A un arbre binaire complet. Posons $m = S(A)$. Montrer que pour tout indice $i < |m|$ on a $N_0(m[1 \dots i]) \geqslant N_1(m[1 \dots i])$.

Q. 22 En déduire que la valeur que doit renvoyer *fin-sous-arbre*(m, i) est le plus petit entier j tel que $m[i \dots j]$ contient strictement plus de 1 que de 0.

Q. 23 Écrire l'algorithme associé à *fin-sous-arbre*(m, i).

Synthèse

Q. 24 Montrer que si deux arbres A et B vérifient $S(A) = S(B)$ alors A et B sont le même arbre, c'est à dire que le code est non-ambigu.

Réglementation de la première épreuve d’admissibilité

Extrait de l’annexe de l’arrêté du 17 avril 2025 fixant les modalités d'organisation du concours externe du certificat d'aptitude au professorat de l'enseignement du second degré, publié au Journal Officiel du 19 avril 2025

A. – Epreuves d’admissibilité

1° Première épreuve d'admissibilité.

L'épreuve consiste en l'analyse et la résolution d'un ou plusieurs problèmes. L'épreuve évalue les connaissances disciplinaires ainsi que les capacités de raisonnement, d'argumentation et de programmation du candidat.

Durée : quatre heures.

Coefficient 3.

L'épreuve est notée sur 20. Une note globale égale ou inférieure à 5 est éliminatoire ;