Problème algorithmique : étant donné une chaîne de caractères `s`, renvoyer la plus longue sous-chaîne palindrome dans `s`.

Exemple concret :

- **Entrée :** `s = "babad"`
- **Sortie :** `"bab"` (Note : `"aba"` est aussi une réponse valide).
- **Entrée :** `s = "cbbd"`
- **Sortie :** `"bb"`

Solution 1

Le code ci-dessous propose une solution brute-force qui teste toutes les sous-chaînes possibles de la chaîne d'entrée, vérifie si elles sont des palindromes, et conserve la plus longue trouvée.

```python
def is_palindrome(substring):
    """
    Vérifie si une chaîne est un palindrome.
    """
    for i in range(len(substring)):
        if substring[i] != substring[-(i+1)]:
            return False
    return True

def longestPalindrome(s):
    """
    Trouve la plus longue sous-chaîne palindrome dans s.
    """
    longest = 0
    longest_substring = ""

    # Générer toutes les sous-chaînes possibles
    for i in range(len(s) + 1):
        for j in range(i, len(s) + 1):
            substring = s[i:j]

            # Vérifier si la sous-chaîne est un palindrome
            if is_palindrome(substring):

                # Mettre à jour si c'est la plus longue trouvée
                if len(substring) > longest:
                    longest = len(substring)
                    longest_substring = substring

    print(f"La plus longue sous-chaîne palindrome est : '{longest_substring}'")
    return longest_substring
```

Solution 2

Cette seconde solution est également une approche par force brute, très similaire à la première. Elle génère toutes les sous-chaînes possibles et vérifie pour chacune si elle est un palindrome, en conservant la plus longue trouvée.

Une légère optimisation a été apportée à la fonction `is_palindrome` : elle ne parcourt que la moitié de la chaîne de caractères, car vérifier la symétrie sur une moitié suffit à déterminer si c'est un palindrome.

```python
def is_palindrome(substring):
    #  Inutile de faire le test deux fois
    for i in range(len(substring) // 2):
        if substring[i] != substring[-(i+1)]:
            return False
    return True

def longestPalindrome(s):
    n = len(s)
    longest_substring = ""

    # On parcourt toutes les substrings possibles
    for i in range(n):
        for j in range(i, n):
            substring = s[i:j+1]


            if is_palindrome(substring):
                if len(substring) > len(longest_substring):
                    longest_substring = substring

    print(longest_substring)
    return longest_substring
```

Solution 3

Une troisième approche beaucoup plus optimisée. Au lieu de générer toutes les sous-chaînes, elle part du principe que chaque palindrome a un centre. Ce centre peut être un seul caractère (pour les palindromes de longueur impaire comme `"aba"`) ou l'espace entre deux caractères (pour les longueurs paires comme `"abba"`).

L'algorithme parcourt la chaîne une seule fois. Pour chaque position, il la considère comme un centre potentiel et "étend" la vérification vers l'extérieur (gauche et droite) tant que les caractères correspondent. Il fait cela pour les deux cas (longueur paire et impaire) et conserve le plus long palindrome trouvé. Bien que sa complexité temporelle soit toujours $O(N^2)$, cette méthode est nettement plus rapide en pratique que les approches par force brute.

```python
def expand_around_center(s, left, right):
    # renvoie le plus long palindrome a partir d un caractere , O(n)
    while left >= 0 and right < len(s) and s[left] == s[right]:
        left -= 1
        right += 1
    return s[left + 1:right]


def longestPalindrome(s):

    if not s:
        return ""

    longest_substring = ""

    for i in range(len(s)):
        # distinction des cas pairs et impairs
        odd_palindrome = expand_around_center(s,i, i)
        even_palindrome = expand_around_center(s,i, i + 1)

        longest_substring = max(longest_substring, odd_palindrome, even_palindrome, key=len)

    print(longest_substring)
    return longest_substring
```

Solution 4

Cette méthode s'appuie sur des opérations matricielles avec NumPy : au lieu de tester chaque centre séparément en Python, on traduit la séquence en un vecteur d'entiers puis on effectue des comparaisons décalées et des opérations booléennes sur des tableaux entiers.

Encodage des symboles. Pour pouvoir comparer rapidement des éléments quelconques (caractères d'une chaîne ou éléments d'une séquence de `hashable`) avec les opérateurs vectoriels de NumPy, la fonction `_encode_labels` mappe chaque symbole sur un entier $0, \ldots, \sigma-1$. Le résultat est un tableau entier `lab` de longueur N qui sert de représentation compacte et comparable des symboles originaux.

Comparaisons par décalages (shifts). Un palindrome centré en c de rayon k impose que, pour tout $t \in \{1, \ldots, k\}$, on ait $\texttt{lab}[c-t] = \texttt{lab}[c+t]$ (cas impair) ou $\texttt{lab}[c-t+1] = \texttt{lab}[c+t]$ (cas pair). Plutôt que de tester chaque centre séparément, on calcule pour chaque pas k un masque booléen `eq` qui compare en une seule opération vectorielle deux tranches décalées de `lab` (par exemple `lab[:-2*k]` et `lab[2*k:]`). Ce masque indique simultanément, pour tous les centres valides à ce pas, si la paire distante k correspond.

Accumulation des runs. Pour savoir si un palindrome s'étend d'au moins k autour d'un centre donné, on maintient un tableau booléen `run_odd` (resp. `run_even`) qui représente si toutes les comparaisons pour les pas $1, \ldots, k$ ont été vraies. À chaque itération sur k on met à jour ce masque (en excluant soigneusement les indices hors-bord) puis on l'ajoute à un accumulateur entier `R_odd` (resp. `R_even`) : cet accumulateur compte le nombre de pas consécutifs satisfaits, soit littéralement le rayon du palindrome pour chaque centre.

Reconstruction du meilleur palindrome. Après avoir traité tous les pas k, les rayons sont convertis en longueurs via $L_{odd} = 2R_{odd} + 1$ et $L_{even} = 2R_{even}$. On identifie la longueur maximale parmi les cas pairs et impairs, on récupère le centre correspondant et son rayon, puis on reconstruit les indices de début et de fin du meilleur palindrome. Si l'entrée était une chaîne, la méthode renvoie aussi la sous-chaîne correspondante ; sinon elle renvoie `"None"`.

Complexité et limites. L'algorithme réalise en pratique $O(N^2)$ comparaisons élémentaires (pour tous les pas k et centres valides), mais ces comparaisons et masques sont vectorisés et exécutés en C par NumPy, ce qui réduit fortement le coût d'interprétation Python. La mémoire principale reste $O(N)$ pour `lab`, `R_*` et `run_*`, toutefois des tableaux temporaires `eq` de taille $\approx N$ sont alloués à chaque pas — attention donc aux très longues séquences.

```python
def _encode_labels(seq):
    m = {}
    lab = np.empty(len(seq), dtype=np.int32)
    for i, x in enumerate(seq):
        lab[i] = m.setdefault(x, len(m))
    return lab

```

```python
def longest_palindrome_matrix(seq):
    """
    Matrix/shift-based longest palindromic substring.
    Returns: (start, end_exclusive, length, substring_if_str_else_None)
    """
    N = len(seq)
    is_str = isinstance(seq, str)
    if N == 0: return (0, 0, 0, "" if isinstance(seq, str) else None)
    lab = _encode_labels(seq if not is_str else seq)

    # 1) Cas impair
    R_odd = np.zeros(N, dtype=np.int32)       # radius per center
    run_odd = np.zeros(N, dtype=bool)
    for k in range(1, N):                     # k = radius step
        m = N - 2 * k
        if m <= 0:
            break
        eq = (lab[:-2*k] == lab[2*k:])        # eq for centers k..N-k-1
        if k == 1:
            run_odd.fill(False)
            run_odd[k:N-k] = eq
        else:
            run_odd[:k] = False
            run_odd[N-k:] = False
            run_odd[k:N-k] &= eq
        R_odd += run_odd

    L_odd = 2 * R_odd + 1
    odd_len = int(L_odd.max())
    odd_center = int(L_odd.argmax())
    odd_r = int(R_odd[odd_center])

    # 2) Cas pair
    if N >= 2:
        R_even = np.zeros(N - 1, dtype=np.int32)
        run_even = np.zeros(N - 1, dtype=bool)
        for k in range(1, N // 2 + 1):
            m = N - 2 * k + 1
            if m <= 0:
                break
            eq = (lab[:m] == lab[2*k - 1:])   # centres (k-1)..(N-k-1)
            if k == 1:
                run_even.fill(False)
                run_even[k-1:N-k] = eq
            else:
                run_even[:k-1] = False
                run_even[N-k:] = False
                run_even[k-1:N-k] &= eq
            R_even += run_even

        L_even = 2 * R_even
        even_len = int(L_even.max()) if L_even.size else 0
        even_center = int(L_even.argmax()) if L_even.size else 0
        even_r = int(R_even[even_center]) if R_even.size else 0
    else:
        even_len = 0
        even_center = 0
        even_r = 0
```

```
66
67      # 3) Sélectionner les meilleurs indices et les reconstruire
68      if odd_len >= even_len:
69          start = odd_center - odd_r
70          end = odd_center + odd_r + 1
71          best_len = odd_len
72      else:
73          c = even_center
74          r = even_r
75          start = c - r + 1
76          end = c + r + 1
77          best_len = even_len
78
79      sub = seq[start:end] if is_str else None
80      return (start, end, best_len, sub)
```

Solution 5

Manacher transforme le problème en un cadre unifié pour palindromes de longueur paire et impaire grâce à un séparateur (#) inséré entre les caractères et des sentinelles aux extrémités. On maintient :

— un tableau P où $P[i]$ est le *rayon* (nombre de caractères appariés de part et d'autre) du plus long palindrome centré en i dans la chaîne transformée,

— un centre courant c et une frontière droite r du palindrome le plus à droite déjà trouvé,

— la symétrie : pour un index i à l'intérieur de $[\cdot, r)$, son miroir par rapport à c est $i' = 2c - i$, ce qui donne une borne initiale $P[i] \geq \min(r - i, P[i'])$.

On étend ensuite naïvement autour de i au-delà de r si possible. Le tout est linéaire, car chaque caractère ne déclenche qu'un nombre amorti constant d'extensions.

```python
def longest_palindrome(s: str) -> str:
    if not s: return ""
    t = '^#' + '#'.join(s) + '#$'       # sentinelles + séparateurs
    P, c, r = [0]*len(t), 0, 0
    for i in range(1, len(t)-1):
        if i < r: P[i] = min(r - i, P[2*c - i])
        while t[i + P[i] + 1] == t[i - P[i] - 1]:
            P[i] += 1
        if i + P[i] > r: c, r = i, i + P[i]
    m, i = max((n, i) for i, n in enumerate(P))
    return s[(i - m)//2 : (i + m)//2]
```

Prétraitement : on construit `t = '^#' + '#'.join(s) + '#$'` afin de gérer uniformément les palindromes pairs et impairs, tout en évitant les tests de bornes.

Structures (`P, c, r`) :

— `P[i]` = rayon du plus long palindrome centré en `i` dans `t`,

— `c` = centre du palindrome actif le plus à droite,

— `r` = position de sa frontière droite.

Initialisation par symétrie (`if i < r: ...`) : si `i` est sous la cloche courante, on initialise `P[i]` à $\min(r - i, P[2 * c - i])$, grâce au miroir $i' = 2c - i$.

Extension (boucle `while`) : on agrandit le palindrome centré en `i` tant que les caractères situés de part et d'autre du centre sont égaux.

Mise à jour (`if i + P[i] > r: ...`) : si l'on dépasse `r`, le palindrome courant devient la nouvelle cloche `(c, r)`.

Extraction du résultat : on prend le maximum `m` de `P` (le rayon) et son centre `i`. Dans la chaîne originale `s`, l'indice de début vaut $(i - m)//2$ et la fin $(i + m)//2$. On renvoie donc `s[(i - m)//2 : (i + m)//2]`.

Remarques sur la complexité de cet algo

L'initialisation par miroir ne sous-estime jamais le vrai rayon car elle est bornée par `r - i`, et elle ne nécessite des extensions que *au-delà* de `r`. Le maximum de `P` permet d'identifier un centre optimal, et la conversion d'indices est correcte puisque chaque caractère de `s` correspond à une position impaire de `t`. Concernant la complexité, l'algorithme s'exécute en temps $O(n)$ et utilise un espace $O(n)$, avec $n = |\mathtt{s}|$. Chaque position augmente `r` au plus une seule fois, et les comparaisons effectuées hors symétrie sont amorties.