Algorithmes du programme

Première et terminale

Ce livret rassemble les algorithmes que le programme de NSI demande de savoir écrire, compléter ou dérouler, en première et en terminale. Chacun reprend l'implémentation du cours correspondant. Pour chaque algorithme sont donnés la capacité du programme qu'il illustre, son principe, son code, des exemples d'appel avec leur résultat, et son coût, avec l'invariant ou le variant quand le programme le demande. Les capacités citées en italique sont celles des programmes officiels (bulletin officiel spécial n° 1 du 22 janvier 2019 pour la première, n° 8 du 25 juillet 2019 pour la terminale). Tous les codes et tous les résultats affichés ont été exécutés et vérifiés.

① Parcours séquentiel d'un tableau

1.1 Recherche d'une occurrence

Programme de première : écrire un algorithme de recherche d'une occurrence sur des valeurs de type quelconque.

On compare chaque élément à la valeur cherchée et on s'arrête dès qu'on la trouve.

```
def chercher(liste, x):
    """Renvoie True si x est dans liste, False sinon."""
    for element in liste:
        if element == x:
            return True
    return False
```

```
>>> chercher([3, 8, 1, 9], 1)
True
>>> chercher(["pomme", "kiwi"], "poire")
False
```

Coût. Linéaire, en $O(n)$ pour une liste de n éléments : dans le pire cas, tous les éléments sont examinés.

1.2 Recherche d'un extremum

Programme de première : écrire un algorithme de recherche d'un extremum.

On garde en mémoire le plus grand élément rencontré, ou son indice, et on le remplace chaque fois qu'on trouve plus grand.

```
def maximum(liste):
    """Renvoie le plus grand élément d'une liste non vide."""
    maxi = liste[0]
    for element in liste:
        if element > maxi:
            maxi = element
    return maxi

def indice_maximum(liste):
    """Renvoie l'indice du plus grand élément d'une liste non vide."""
    indice_max = 0
    for i in range(len(liste)):
        if liste[i] > liste[indice_max]:
            indice_max = i
    return indice_max
```

```
>>> maximum([12, 7, 19, 3])
19
>>> indice_maximum([12, 7, 19, 3])
2
```

Coût. Linéaire, en $O(n)$.

1.3 Calcul d'une moyenne

Programme de première : écrire un algorithme de calcul d'une moyenne.

On accumule la somme des éléments, puis on la divise par leur nombre.

```
def moyenne(liste):
    """Renvoie la moyenne des nombres d'une liste non vide."""
    somme = 0
    for element in liste:
        somme = somme + element
    return somme / len(liste)
```

```
>>> moyenne([12, 15, 9, 14])
12.5
```

Coût. Linéaire, en $O(n)$.

② Tris par insertion et par sélection

2.1 Tri par insertion

Programme de première : écrire un algorithme de tri ; décrire un invariant de boucle qui prouve la correction du tri par insertion.

On prend les éléments un par un et on insère chacun à sa place dans la partie déjà triée, à sa gauche, en décalant vers la droite les éléments plus grands.

```
def tri_insertion(L):
    """Trie la liste L par ordre croissant et la renvoie."""
    for i in range(1, len(L)):
        en_cours = L[i]
        j = i
        # décalage vers la droite des éléments plus grands que en_cours
        while j > 0 and L[j - 1] > en_cours:
            L[j] = L[j - 1]
            j -= 1
        L[j] = en_cours
    return L
```

```
>>> tri_insertion([5, 2, 9, 1, 7])
[1, 2, 5, 7, 9]
```

Coût. Quadratique, en $O(n^2)$ dans le pire cas, pour une liste rangée dans l'ordre décroissant ; linéaire si la liste est déjà triée.

Invariant. Au début de l'étape i, les éléments `L[0]` à `L[i - 1]` sont triés. C'est vrai pour $i = 1$, chaque étape le conserve, donc la liste entière est triée à la fin.

2.2 Tri par sélection

Programme de première : écrire un algorithme de tri ; décrire un invariant de boucle qui prouve la correction du tri par sélection.

À l'étape i, on cherche le plus petit élément de la partie non triée et on l'échange avec l'élément d'indice i.

```
def echanger(L, i, j):
    """Échange les éléments d'indices i et j de la liste L."""
    L[i], L[j] = L[j], L[i]

def tri_selection(L):
    """Trie la liste L par ordre croissant et la renvoie."""
    for i in range(len(L) - 1):
        indice_du_min = i
        for j in range(i + 1, len(L)):
            if L[j] < L[indice_du_min]:
                indice_du_min = j
        echanger(L, i, indice_du_min)
    return L
```

```
>>> tri_selection([5, 2, 9, 1, 7])
[1, 2, 5, 7, 9]
```

Coût. Quadratique, en $O(n^2)$, quelle que soit la liste.

Invariant. Au début de l'étape i, les éléments `L[0]` à `L[i - 1]` sont les i plus petits éléments de la liste, rangés dans l'ordre croissant.

③ Recherche dichotomique

3.1 Recherche dans une liste triée

Programme de première : montrer la terminaison de la recherche dichotomique à l'aide d'un variant de boucle.

On compare la valeur cherchée à l'élément du milieu de la zone de recherche ; selon le résultat, on s'arrête ou on continue dans la moitié gauche ou dans la moitié droite.

```
def recherche_dichotomique(liste, element):
    """Renvoie True si element est dans liste (triée), False sinon."""
    debut = 0
    fin = len(liste) - 1
    while debut <= fin:
        centre = (debut + fin) // 2
        if liste[centre] == element:
            return True
        elif liste[centre] < element:
            debut = centre + 1
        else:
            fin = centre - 1
    return False
```

```
>>> recherche_dichotomique([2, 5, 8, 12, 16, 23], 12)
True
>>> recherche_dichotomique([2, 5, 8, 12, 16, 23], 7)
False
```

Coût. Logarithmique, en $O(\log_2 n)$: la zone de recherche est divisée par deux à chaque tour.

Variant. L'entier `fin - debut` diminue strictement à chaque tour qui ne trouve pas l'élément ; il finit par devenir négatif, et la boucle s'arrête.

④ Algorithmes gloutons

4.1 Rendu de monnaie

Programme de première : résoudre un problème grâce à un algorithme glouton.

On rend toujours la plus grande pièce possible, tant qu'elle ne dépasse pas la somme qui reste à rendre.

```
def rendu_monnaie(montant, pieces):
    """Renvoie la liste des pièces rendues pour montant (en centimes)."""
    pieces_triees = sorted(pieces, reverse=True)
    resultat = []
    reste = montant
    for piece in pieces_triees:
        while reste >= piece:
            resultat.append(piece)
            reste -= piece
    return resultat
```

```
>>> rendu_monnaie(678, [1, 2, 5, 10, 20, 50, 100, 200])
[200, 200, 200, 50, 20, 5, 2, 1]
>>> rendu_monnaie(6, [1, 3, 4])
[4, 1, 1]
```

Coût. Celui du tri des pièces, puis une étape par pièce rendue.

Remarque. Avec les pièces en euros, le glouton rend toujours le nombre minimal de pièces. Ce n'est pas vrai pour tous les systèmes : avec des pièces de 1, 3 et 4, il rend trois pièces pour 6, alors que deux suffisent $(3 + 3)$.

4.2 Sac à dos

Programme de première : résoudre un problème grâce à un algorithme glouton.

Chaque objet est un triplet (nom, valeur, poids). On range les objets par rapport valeur sur poids décroissant, puis on prend chaque objet s'il tient encore dans le sac.

```
def rapport(o):
    """Rapport valeur/poids de l'objet o = (nom, valeur, poids)."""
    return o[1] / o[2]

def sac_a_dos_01(poids_max, objets):
    """Renvoie les objets choisis et leur valeur totale."""
    tries = sorted(objets, key=rapport, reverse=True)
    poids_actuel = 0
    valeur_totale = 0
    sac = []
    for nom, valeur, poids in tries:
        if poids_actuel + poids <= poids_max:
            sac.append((nom, valeur, poids))
            poids_actuel += poids
            valeur_totale += valeur
    return sac, valeur_totale
```

```
>>> sac_a_dos_01(10, [("A", 7, 6), ("B", 5, 5), ("C", 5, 5)])
([('A', 7, 6)], 7)
```

Coût. Celui du tri, en $O(n \log n)$, suivi d'un parcours linéaire.

Remarque. Le glouton ne donne pas toujours la meilleure solution. Dans l'exemple, il prend A (valeur 7) et le sac ne peut plus rien recevoir, alors que B et C ensemble valent 10.

⑤ Algorithme des k plus proches voisins

5.1 Prédire une classe

Programme de première : écrire un algorithme qui prédit la classe d'un élément en fonction de la classe majoritaire de ses k plus proches voisins.

Les données connues sont des couples (point, classe). On calcule la distance de chacune au point étudié, on garde les k plus proches, puis on renvoie la classe la plus fréquente parmi elles.

```
from math import sqrt

def distance(p, q):
    """Distance entre deux points du plan p = (x, y) et q = (x, y)."""
    return sqrt((p[0] - q[0]) ** 2 + (p[1] - q[1]) ** 2)

def premier(couple):
    return couple[0]

def k_plus_proches_voisins(donnees, cible, k):
    """donnees : liste de couples (point, classe).
    Renvoie les k couples (distance, classe) les plus proches de cible."""
    distances = []
    for point, classe in donnees:
        distances.append((distance(point, cible), classe))
    distances = sorted(distances, key=premier)
    return distances[:k]

def classe_majoritaire(voisins):
    """Renvoie la classe la plus fréquente parmi les couples (distance, classe)."""
    effectifs = {}
    for d, classe in voisins:
        if classe in effectifs:
            effectifs[classe] = effectifs[classe] + 1
        else:
            effectifs[classe] = 1
    meilleure = None
    maxi = 0
    for classe in effectifs:
        if effectifs[classe] > maxi:
            maxi = effectifs[classe]
            meilleure = classe
    return meilleure
```

```
>>> donnees = [((1, 1), "rouge"), ((2, 1), "rouge"), ((1, 3), "rouge"),
...            ((5, 4), "bleu"), ((4, 5), "bleu"), ((3, 3), "bleu")]
>>> classe_majoritaire(k_plus_proches_voisins(donnees, (2, 2), 3))
'rouge'
>>> classe_majoritaire(k_plus_proches_voisins(donnees, (4, 3), 3))
'bleu'
```

Coût. Celui du tri des n distances, en $O(n \log n)$. Le TP de première extrait plutôt k fois la plus petite distance, en $O(k \times n)$, ce qui est avantageux quand k est petit.

⑥ Piles et files

6.1 Pile et file avec une liste Python

Programme de terminale : distinguer des structures par le jeu des méthodes qui les caractérisent ; écrire plusieurs implémentations d'une même structure de données.

La pile ajoute et retire au même bout de la liste ; la file ajoute à la fin et retire au début.

```
class Pile:
    def __init__(self):
        self.contenu = []

    def est_vide(self):
        return len(self.contenu) == 0

    def empiler(self, element):
        self.contenu.append(element)

    def depiler(self):
        if self.est_vide():
            raise ValueError("pile vide")
        return self.contenu.pop()

class File:
    def __init__(self):
        self.contenu = []

    def est_vide(self):
        return len(self.contenu) == 0

    def enfiler(self, element):
        self.contenu.append(element)

    def defiler(self):
        if self.est_vide():
            raise ValueError("file vide")
        return self.contenu.pop(0)
```

```
>>> p = Pile()
>>> f = File()
>>> for x in [1, 2, 3]:
...     p.empiler(x)
...     f.enfiler(x)
...
>>> p.depiler()
3
>>> f.defiler()
1
```

Coût. Empiler, dépiler et enfiler en temps constant ; défiler en $O(n)$ avec une liste Python, car tous les éléments sont décalés. L'implémentation du cours par liste chaînée défile en temps constant.

⑦ Arbres binaires

7.1 Taille et hauteur

Programme de terminale : calculer la taille et la hauteur d'un arbre.

Un arbre est soit vide (`None`), soit un nœud avec une valeur, un sous-arbre gauche et un sous-arbre droit. La taille est le nombre de nœuds ; la hauteur vaut -1 pour l'arbre vide et 0 pour une racine seule.

```
class Arbre:
    def __init__(self, valeur, gauche=None, droit=None):
        self.valeur = valeur
        self.gauche = gauche
        self.droit = droit

def taille(arbre):
    if arbre is None:
        return 0
    return 1 + taille(arbre.gauche) + taille(arbre.droit)

def hauteur(arbre):
    if arbre is None:
        return -1
    return 1 + max(hauteur(arbre.gauche), hauteur(arbre.droit))
```

```
>>> a = Arbre(8, Arbre(3, Arbre(1), Arbre(6)), Arbre(10, None, Arbre(14)))
>>> taille(a)
6
>>> hauteur(a)
2
```

Coût. Linéaire, en $O(n)$: chaque nœud est visité une fois.

7.2 Parcours en profondeur et en largeur

Programme de terminale : parcourir un arbre de différentes façons (ordres infixe, préfixe ou suffixe ; ordre en largeur d'abord).

En profondeur, on traite la racine avant ses sous-arbres (préfixe), entre eux (infixe) ou après eux (suffixe). En largeur, on traite les nœuds niveau par niveau, à l'aide d'une file.

```
def prefixe(arbre):
    if arbre is not None:
        print(arbre.valeur, end=' ')
        prefixe(arbre.gauche)
        prefixe(arbre.droit)

def infixe(arbre):
    if arbre is not None:
        infixe(arbre.gauche)
        print(arbre.valeur, end=' ')
        infixe(arbre.droit)

def suffixe(arbre):
    if arbre is not None:
        suffixe(arbre.gauche)
        suffixe(arbre.droit)
        print(arbre.valeur, end=' ')

def largeur(arbre):
    if arbre is None:
        return
    file = [arbre]
    while file != []:
        noeud = file.pop(0)
        print(noeud.valeur, end=' ')
        if noeud.gauche is not None:
            file.append(noeud.gauche)
        if noeud.droit is not None:
            file.append(noeud.droit)
```

```
>>> a = Arbre(8, Arbre(3, Arbre(1), Arbre(6)), Arbre(10, None, Arbre(14)))
>>> prefixe(a)
8 3 1 6 10 14
>>> infixe(a)
1 3 6 8 10 14
>>> suffixe(a)
1 6 3 14 10 8
>>> largeur(a)
8 3 10 1 6 14
```

Coût. Linéaire, en $O(n)$.

7.3 Arbre binaire de recherche

Programme de terminale : rechercher une clé dans un arbre de recherche, insérer une clé.

Dans un arbre binaire de recherche, les valeurs du sous-arbre gauche sont plus petites que la racine et celles du sous-arbre droit plus grandes. La recherche et l'insertion descendent donc d'un seul côté à chaque étape.

```
def rechercher(arbre, x):
    if arbre is None:
        return False
    if x == arbre.valeur:
        return True
    elif x < arbre.valeur:
        return rechercher(arbre.gauche, x)
    else:
        return rechercher(arbre.droit, x)

def inserer(arbre, x):
    """Insère x et renvoie l'arbre obtenu."""
    if arbre is None:
        return Arbre(x)
    if x < arbre.valeur:
        arbre.gauche = inserer(arbre.gauche, x)
    elif x > arbre.valeur:
        arbre.droit = inserer(arbre.droit, x)
    return arbre
```

```
>>> a = None
>>> for x in [8, 3, 10, 1, 6, 14, 4, 7]:
...     a = inserer(a, x)
...
>>> rechercher(a, 6)
True
>>> rechercher(a, 5)
False
>>> infixe(a)
1 3 4 6 7 8 10 14
```

Coût. En $O(h)$, où h est la hauteur de l'arbre : de l'ordre de $\log_2 n$ pour un arbre équilibré, mais $n-1$ pour un arbre filiforme.

Remarque. Le parcours infixe d'un arbre binaire de recherche donne ses valeurs dans l'ordre croissant.

⑧ Graphes

8.1 Parcours en profondeur et en largeur

Programme de terminale : parcourir un graphe en profondeur d'abord, en largeur d'abord.

Le graphe est un dictionnaire qui associe à chaque sommet la liste de ses voisins. La liste `deja_vus` marque les sommets rencontrés, pour ne pas les traiter deux fois. Le parcours en profondeur avance tant qu'il peut ; le parcours en largeur traite les voisins directs avant les voisins des voisins, à l'aide d'une file.

```
def parcourir_profondeur_recursif(G, S, deja_vus):
    deja_vus.append(S)
    print(S, end=" ")
    for voisin in G[S]:
        if voisin not in deja_vus:
            parcourir_profondeur_recursif(G, voisin, deja_vus)

def parcourir_largeur(G, S, deja_vus):
    file_a_traiter = [S]
    deja_vus.append(S)
    while file_a_traiter != []:
        S = file_a_traiter.pop(0)
        print(S, end=" ")
        for voisin in G[S]:
            if voisin not in deja_vus:
                deja_vus.append(voisin)
                file_a_traiter.append(voisin)
```

```
>>> G = {"A": ["B", "C"], "B": ["A", "D"], "C": ["A", "D", "E"],
...      "D": ["B", "C"], "E": ["C"]}
>>> parcourir_profondeur_recursif(G, "A", [])
A B D C E
>>> parcourir_largeur(G, "A", [])
A B C D E
```

Coût. Chaque sommet est traité une fois et chaque liste de voisins parcourue une fois. Avec une liste `deja_vus`, chaque test d'appartenance coûte jusqu'au nombre de sommets ; un dictionnaire de marquage le ramènerait à un temps constant.

8.2 Recherche d'un chemin

Programme de terminale : chercher un chemin dans un graphe.

On fait un parcours en largeur depuis le départ en notant, pour chaque sommet découvert, le sommet d'où l'on vient. Si l'arrivée est atteinte, on remonte ces prédécesseurs pour reconstruire le chemin. Le parcours en largeur donne un chemin qui a le moins d'arêtes possible.

```
def parcourir(G, S, D, deja_vus):
    file_a_traiter = [S]
    deja_vus[S] = None
    while file_a_traiter != []:
        S = file_a_traiter.pop(0)
        if S == D:
            return True
        for voisin in G[S]:
            if voisin not in deja_vus:
                deja_vus[voisin] = S        # prédécesseur de voisin
                file_a_traiter.append(voisin)
    return False

def recherche_chemin(G, S, D):
    M = {}          # prédécesseurs
    R = []          # chemin, construit à l'envers
    if parcourir(G, S, D, M):
        R.append(D)
        while D != S:
            D = M[D]
            R.append(D)
    R.reverse()
    return R
```

```
>>> G = {"A": ["B", "C"], "B": ["A", "D"], "C": ["A", "D", "E"],
...      "D": ["B", "C"], "E": ["C"]}
>>> recherche_chemin(G, "A", "E")
['A', 'C', 'E']
>>> recherche_chemin(G, "B", "E")
['B', 'A', 'C', 'E']
```

Coût. Celui d'un parcours en largeur.

8.3 Présence d'un cycle

Programme de terminale : repérer la présence d'un cycle dans un graphe.

Pour un graphe non orienté et connexe, on fait un parcours en profondeur avec une pile. Si l'on dépile un sommet déjà marqué, c'est qu'il a été atteint par deux arêtes différentes : le graphe contient un cycle.

```
def possede_cycle(G, S, deja_vus):
    pile_a_traiter = [S]
    while pile_a_traiter != []:
        S = pile_a_traiter.pop()
        if S in deja_vus:
            return True
        deja_vus.append(S)
        for voisin in G[S]:
            if voisin not in deja_vus:
                pile_a_traiter.append(voisin)
    return False
```

```
>>> G = {"A": ["B", "C"], "B": ["A", "D"], "C": ["A", "D", "E"],
...      "D": ["B", "C"], "E": ["C"]}
>>> arbre = {"A": ["B", "C"], "B": ["A"], "C": ["A", "D"], "D": ["C"]}
>>> possede_cycle(G, "A", [])
True
>>> possede_cycle(arbre, "A", [])
False
```

Coût. Celui d'un parcours en profondeur.

Attention. Cette méthode ne convient pas à un graphe orienté : avec les arcs A vers B, A vers C et C vers B, le sommet B est empilé deux fois et la fonction répond qu'il y a un cycle, alors qu'il n'y en a pas.

⑨ Diviser pour régner

9.1 Tri fusion

Programme de terminale : écrire un algorithme utilisant la méthode « diviser pour régner ».

On coupe la liste en deux moitiés, on trie chacune par un appel récursif, puis on fusionne les deux listes triées en une seule. Dans `fusion`, `resultat.extend(L)` ajoute à la fin de `resultat` tous les éléments de `L`, dans l'ordre.

```
def fusion(gauche, droite):
    """Fusionne deux listes triées en une seule liste triée."""
    resultat = []
    i, j = 0, 0
    while i < len(gauche) and j < len(droite):
        if gauche[i] <= droite[j]:
            resultat.append(gauche[i])
            i += 1
        else:
            resultat.append(droite[j])
            j += 1
    resultat.extend(gauche[i:])
    resultat.extend(droite[j:])
    return resultat

def tri_fusion(tab):
    """Renvoie une liste triée contenant les éléments de tab."""
    if len(tab) <= 1:
        return tab
    milieu = len(tab) // 2
    gauche = tri_fusion(tab[:milieu])
    droite = tri_fusion(tab[milieu:])
    return fusion(gauche, droite)
```

```
>>> tri_fusion([38, 27, 43, 3, 9, 82, 10])
[3, 9, 10, 27, 38, 43, 82]
```

Coût. En $O(n \log n)$, quelle que soit la liste : $\log_2 n$ niveaux de découpage, et un travail linéaire à chaque niveau.

9.2 Exponentiation rapide

Programme de terminale : écrire un algorithme utilisant la méthode « diviser pour régner ».

Pour calculer a^n, on calcule une seule fois $a^{n/2}$ et on l'élève au carré, en multipliant encore par a si n est impair.

```
def puissance(a, n):
    """Calcule a puissance n par exponentiation rapide."""
    if n == 0:
        return 1
    if n % 2 == 0:
        demi = puissance(a, n // 2)
        return demi * demi
    else:
        demi = puissance(a, (n - 1) // 2)
        return a * demi * demi
```

```
>>> puissance(3, 13)
1594323
>>> puissance(2, 10)
1024
```

Coût. En $O(\log_2 n)$ multiplications, contre $n - 1$ pour le calcul naïf.

10 Programmation dynamique

10.1 Rendu de monnaie optimal

Programme de terminale : utiliser la programmation dynamique pour écrire un algorithme. Le rendu de monnaie est l'un des deux exemples cités.

`t[i][j]` est le nombre minimal de pièces pour rendre j avec les i premières valeurs de pièces. On remplit le tableau ligne par ligne : soit on n'utilise pas la pièce v, soit on l'utilise une fois de plus à partir de `t[i][j - v]`. Chaque sous-problème n'est résolu qu'une seule fois.

```
def rendu_dynamique(monnaie, s):
    """Nombre minimal de pièces de monnaie pour rendre la somme s."""
    n = len(monnaie)
    INF = float("inf")
    t = [[INF] * (s + 1) for _ in range(n + 1)]
    for i in range(n + 1):
        t[i][0] = 0
    for i in range(1, n + 1):
        v = monnaie[i - 1]
        for j in range(1, s + 1):
            t[i][j] = t[i - 1][j]          # sans la pièce v
            if j >= v and t[i][j - v] + 1 < t[i][j]:
                t[i][j] = t[i][j - v] + 1  # avec une pièce v de plus
    return t[n][s]
```

```
>>> rendu_dynamique([1, 3, 4], 6)
2
>>> rendu_dynamique([1, 2, 5, 10, 20, 50, 100, 200], 678)
8
```

Coût. En $O(n \times s)$: une case du tableau par valeur de pièce et par somme.

Remarque. Là où le glouton rend trois pièces pour 6 avec les pièces 1, 3 et 4, la programmation dynamique trouve l'optimum, deux pièces.

10.2 Alignement de séquences : distance d'édition

Programme de terminale : utiliser la programmation dynamique pour écrire un algorithme. L'alignement de séquences est l'un des deux exemples cités.

La distance d'édition est le nombre minimal d'insertions, de suppressions et de substitutions pour passer d'un mot à l'autre. `t[i][j]` est la distance entre les i premières lettres de `a` et les j premières lettres de `b` ; chaque case se déduit de trois cases déjà calculées.

```
def distance_edition(a, b):
    n, m = len(a), len(b)
    t = [[0] * (m + 1) for _ in range(n + 1)]
    for i in range(n + 1):
        t[i][0] = i
    for j in range(m + 1):
        t[0][j] = j
    for i in range(1, n + 1):
        for j in range(1, m + 1):
            cout = 0 if a[i - 1] == b[j - 1] else 1
            t[i][j] = min(t[i - 1][j] + 1,          # suppression
                          t[i][j - 1] + 1,          # insertion
                          t[i - 1][j - 1] + cout)   # substitution
    return t[n][m]
```

```
>>> distance_edition("chien", "niche")
4
>>> distance_edition("algorithme", "logarithme")
3
```

Coût. En $O(n \times m)$ pour des mots de longueurs n et m.

11 Recherche textuelle

11.1 Algorithme de Boyer-Moore, version de Horspool

Programme de terminale : étudier l'algorithme de Boyer-Moore pour la recherche d'un motif dans un texte.

On compare le motif au texte de droite à gauche. En cas d'échec, on décale le motif selon le caractère du texte aligné avec la fin du motif : la table de décalages, calculée une fois, donne pour chaque caractère du motif sa distance à la fin ; un caractère absent du motif permet de sauter toute la longueur du motif.

```
def pre_traitement(motif):
    """Table des décalages : {caractère: décalage}."""
    m = len(motif)
    decalages = {}
    for i in range(m - 1):      # tous les caractères sauf le dernier
        decalages[motif[i]] = m - 1 - i
    return decalages

def recherche_boyer_moore(texte, motif):
    """Renvoie la liste des indices où motif apparaît dans texte."""
    n = len(texte)
    m = len(motif)
    decalages = pre_traitement(motif)
    indices = []
    i = 0
    while i <= n - m:
        j = m - 1
        while j >= 0 and texte[i + j] == motif[j]:
            j -= 1
        if j < 0:
            indices.append(i)
            i += 1
        else:
            c = texte[i + m - 1]
            if c in decalages:
                i += decalages[c]
            else:
                i += m
    return indices
```

```
>>> pre_traitement("chat")
{'c': 3, 'h': 2, 'a': 1}
>>> recherche_boyer_moore("un chat et un chaton", "chat")
[3, 14]
```

Coût. En $O(n \times m)$ comparaisons dans le pire cas, pour un texte de longueur n et un motif de longueur m. Quand les caractères du texte sont variés, les sauts écartent la plupart des positions : il faut bien moins de comparaisons qu'avec la recherche naïve, surtout pour un motif long. Le programme précise que l'étude de ce coût ne peut pas être exigée.