
ITC - 2^o année - Cours 3

Prog Dyn : reconstruction de solutions

I Rappel : rendu de monnaie

1 - Problème du plus petit nombre de pièces à rendre

Entrées :

- Un entier naturel **a_rendre** de centimes à rendre
- Une liste **valeurs_pieces** d'entiers naturels représentant les différentes valeurs possibles de chaque pièce qu'on peut utiliser pour rendre la monnaie.

Sortie : Le nombre minimal de pièces à utiliser pour rendre **a_rendre** centimes.

2 - Résolution par programmation dynamique descendante

```
def rendu_memo(a_rendre, valeurs_possibles):
    memo = { 0: 0 }
    def aux(i):
        if i in memo:
            return memo[i]
        memo[i] = min([1 + aux(i - val) for val in valeurs_possibles if val <= i])
        return memo[i]
    # Fin de la fonction aux
    return aux(a_rendre)
```

II Reconstruction de solution

3 - Différence entre coût d'une solution, et une solution Pour le rendu de monnaie, au lieu de s'intéresser à **combien** de pièces on peut rendre au minimum, on peut s'intéresser à **quelles** pièces il faut rendre, pour rendre le moins de pièces possibles.

Cela revient à changer le problème algorithmique qu'on considère, vers le problème suivant :

Entrées :

- Un entier naturel **a_rendre** de centimes à rendre
- Une liste **valeurs_pieces** d'entiers naturels représentant les différentes valeurs possibles de chaque pièce qu'on peut utiliser pour rendre la monnaie.

Sortie : Une liste des valeurs des pièces rendues, pour rendre **a_rendre** centimes en le moins de pièces possibles. La liste en sortie peut contenir des répétitions. Chaque valeur de la liste en sortie doit appartenir à **valeurs_possibles**.

4 - Utilité de connaître le coût pour obtenir une solution de coût minimale Supposons qu'on veut rendre 79 centimes avec des pièces de 1, 3, 6, 12, 24 et 30 centimes. Pour savoir quelle est la première pièce à rendre, on peut dresser le tableau suivant :

Valeur première pièce	1	3	6	12	24	30
Reste à rendre	78	76	73	67	55	49
Combien de pièces pour rendre le reste?	3	5	4	4	3	3

Le tableau nous permet de conclure que la première pièce à rendre doit être soit une pièce de 1, soit une pièce de 24, soit une pièce de 30.

On a donc besoin de connaître le coût des sous-problème pour construire une solution optimale.

Le processus de construire une solution optimale à partir des coûts s'appelle la **reconstruction de solution**.

III Exemple de reconstruction de solution pour le rendu de monnaie

5 - Fonction renvoyant le dictionnaire de mémorisation On commence par coder une fonction qui prend en entrée le nombre de centimes à rendre et la liste des valeurs possibles des pièces, et qui renvoie le **dictionnaire de mémorisation** qu'aurait rempli la fonction mémorisée d'évaluation du coût

```
def memo_rendu(a_rendre, valeurs_pieces):
    memo = { 0: 0 }
    def aux(i):
        if i in memo:
            return memo[i]
        memo[i] = min([1+aux(i - v) for v in valeurs_pieces if v <= i])
        return memo[i]
    # Fin de la fonction aux
    aux(a_rendre)
    return memo
```

6 - Reconstruction de solution pour le rendu de monnaie On se sert de la fonction du paragraphe précédent pour coder une fonction **pieces_rendues** qui renvoie la liste des valeurs des pièces à rendre. Par exemple, l'appel **pieces_rendues(79, [1, 3, 6, 12, 24, 30])** renvoie la liste [1, 24, 24, 30]. La fonction **pieces_rendues** fait appel à une fonction auxiliaire **indice_minimum** qui prend en entrée une liste et renvoie un **indice** de la liste auquel on peut trouver une occurrence du minimum de la liste.

```
def indice_minimum(liste):
    assert(len(liste) != 0)
    indice = 0
    valeur = liste[0]
    for i in range(len(liste)):
        if valeur > liste[i]:
            indice = i
            valeur = liste[i]
    return indice

def pieces_rendues(a_rendre, valeurs_pieces):
    memo = memo_rendu(a_rendre, valeurs_pieces)
    pieces_rendues = []
    while a_rendre > 0:
        coûts_restes = [memo[a_rendre - v] for v in valeurs_pieces if v <= a_rendre]
        indice_min = indice_minimum(coûts_restes)
        piece_choisie = valeurs_pieces[indice_min]
        pieces_rendues.append(piece_choisie)
```

```

a_rendre = a_rendre - piece_choisie
return pieces_rendues

```

IV Reconstruction de solution - Problème de la grenouille

7 - Rappel : le problème de la grenouille On a n pierres alignées, ayant différentes hauteurs $h_0, \dots, h_{n-1}$. Une grenouille commence à la pierre d'indice 0 et veut se rendre jusqu'à la pierre d'indice $n-1$. À chaque étape, si la grenouille est sur la pierre d'indice i , elle peut

- soit se rendre sur la pierre d'indice $i+1$ avec un coût $|h_{i+1} - h_i|$
- soit se rendre sur la pierre d'indice $i+2$ avec un coût $|h_{i+2} - h_i|$

Le problème algorithmique de la grenouille est alors donné par :

Entrée : une liste `hauteurs` contenant des entiers qui représentent les hauteurs des différentes pierres
Sortie : Le coût minimal en lequel la grenouille peut aller de la pierre 0 à la pierre $n-1$

8 - Fonction renvoyant le dictionnaire de mémorisation du problème de la grenouille

```

def memo_grenouille(hauteurs):
    n = len(hauteurs)
    memo = {}
    def aux(depart):
        if depart in memo:
            return memo[depart]
        if depart == n-1:
            memo[depart] = 0
            return memo[depart]
        if depart == n-2:
            memo[depart] = abs(hauteurs[n-2] - hauteurs[n-1])
            return memo[depart]
        saut1 = abs(hauteurs[depart] - hauteurs[depart + 1])
        saut2 = abs(hauteurs[depart] - hauteurs[depart + 2])
        reste1 = aux(depart + 1)
        reste2 = aux(depart + 2)
        memo[depart] = min(saut1 + reste1, saut2 + reste2)
        return memo[depart]
    # Fin de la fonction aux
    aux(0)
    return memo

```

9 - Exercice : reconstruction de solution Codez une fonction `longueurs_sauts` qui prend en entrée la liste des hauteurs des pierres, et qui renvoie une liste de 1s et de 2s représentant la suite de sauts que doit faire la grenouille pour aller de la première à la dernière pierre avec un coût minimal.

V Distance d'édition entre deux mots

10 - Contexte Imaginez que vous êtes un programme d'édition de texte, et que vous voulez proposer des corrections orthographiques à la personne qui est en train d'écrire. Vous disposez donc d'un dictionnaire avec tous les mots qui existent, et si la personne écrit un mot qui n'est pas dans le dictionnaire, vous suspectez qu'il y a une erreur d'orthographe. Mais comment savoir quel mot

proposer à la place ? Vous souhaitez proposer les mots du dictionnaire qui *ressemblent le plus* au mot tapé. Vous avez donc besoin d'une **mesure de à quel point deux mots se ressemblent**.

11 - Définition de la distance d'édition Pour mesurer la distance entre deux mots, l'informaticien russe Vladimir Levenshtein a proposé en 1965 de compter le nombre minimal d'opérations nécessaires pour passer de l'un à l'autre, si on ne s'autorise que les opérations suivantes :

- supprimer une lettre
- ajouter une lettre
- remplacer une lettre

On appelle cette mesure la *distance d'édition* ou encore la *distance de Levenshtein*.

12 - Exemple Pour passer de « Doikstre » à « Dijkstra », le plus simple est de

- Supprimer le 'o'
- Ajouter un 'j' entre le 'i' et le 'k'
- Remplacer le 'e' par un 'a'

La distance d'édition entre « Doikstre » et « Dijkstra » est donc de 3.

13 - Exercice de compréhension - Définition de la distance d'édition

Quelle est la distance d'édition entre « Levenshtein » et « Livonsteine » ?

V.1 Notations préliminaires

14 - Notation - Indice et tranche Soit u un mot. On numérote les lettres des mots en commençant à l'indice 0. On note $u[i]$ la lettre à l'indice i dans le mot u , et on note $u[i : j]$ la partie du mot u qui va de la lettre d'indice i incluse à la lettre d'indice j exclue. Par exemple, « Dijkstra »[0] = 'D' et « Dijkstra »[1 : 4] = « ijk ».

15 - Exercice

1. Que vaut « Levenshtein »[1 : 5] ?
2. Que vaut « Levenshtein »[5] ?

16 - Notation pour la distance d'édition, et sa matrice Soient u et v deux mots dont on veut calculer la distance d'édition. Soit n la longueur de u et m la longueur de v .

- On note $\text{dist}(u, v)$ la distance d'édition entre u et v .
- On introduit une matrice D d'entiers de taille $n \times m$, définie par :

$$\forall i \in \llbracket 0, n \rrbracket, \forall j \in \llbracket 0, m \rrbracket, D_{i,j} = \text{dist}(u[0 : i], v[0 : j])$$

17 - Exercice Si u = « Doikstre » et v = « Dijkstra », que vaut $D_{8,8}$?

V.2 Calcul de la distance d'édition

18 - Exercice - Relation récursive Soient u et v les deux mots entre lesquels on veut calculer la distance d'édition. Soient n et m leurs longueurs respectives. Soient $i \in \llbracket 0, n \rrbracket$ et $j \in \llbracket 0, m \rrbracket$ deux indices.

Exprimez $D[i, j]$ en fonction de $D[i-1, j-1]$, $D[i-1, j]$ et $D[i, j-1]$. Vous distinguerez plusieurs cas, selon que i vaille 0 ou pas, que j vaille 0 ou pas, et que $u[i-1]$ soit égal à $v[j-1]$ ou pas.

19 - Exercice - Code python Codez une fonction `distance` qui prend en entrée deux chaînes de caractères, et qui renvoie en sortie la distance d'édition entre elles.