CONTRÔLE TERMINAL DÉCEMBRE 2024

documents autorisés : une feuille A4 manuscrite recto-verso
Durée : 2h

Ex 1 (5 pts) : Cours

1) Donner, en Python, une implémentation la plus efficace possible de la recherche dichotomique d'un élément x dans une liste L. Donner sa classe de complexité temporelle. A quelle grande famille d'algorithme appartient-elle ?
2) Dans quel cas est-il préférable d'utiliser la programmation dynamique plutôt que l'approche récursive "diviser pour régner" pour résoudre un problème ?
3) Donner les avantages et inconvénients de la programmation dynamique par rapport à une stratégie gloutonne pour résoudre un problème.

Ex 2 (2,5 pts) : Complexités

1) Les deux fonctions Python ci-contre (prenant en entrée une chaîne de caractères) effectuent la même tâche. Laquelle ?

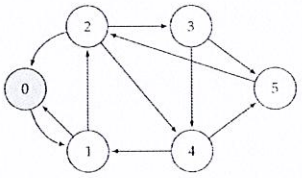
2) Evaluer la classe de complexité temporelle de chacune d'entre-elles, en justifiant.

```python
def fct1(s):
    res = ''
    for i in range(len(s)):
        res = s[i] + res
    return res

def fct2(s):
    n = len(s)
    if (n <= 1):
        return s
    else:
        return fct2(s[n//2:n]) + fct2(s[0:n//2])
```

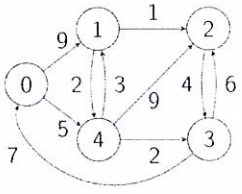
Ex 3 (3 pts) : Parcours

1) On souhaite effectuer un parcours en largeur (BFS) du graphe ci-contre, à partir du sommet 0. De quelle(s) structure(s) de données a-t-on besoin pour effectuer ce type de parcours ? Pourquoi ?
2) Effectuer "à la main" un tel parcours, en faisant apparaître l'état de chaque structure de données lors de chaque itération, comme en TD.



Ex 4 (3 pts) : Graphe orienté pondéré

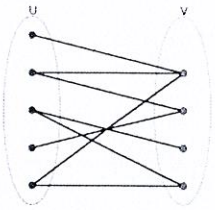
Effectuer "à la main" le déroulement de l'algorithme de Dijkstra sur le graphe pondéré ci-contre, en donnant les distances de poids minimal de chaque sommet au sommet 0, ainsi que le chemin associé.
On utilisera un tableau, comme fait en CM et en TD.



Ex 5 (3 pts) : Graphe biparti

Un graphe est dit "biparti" si ses sommets peuvent être divisés en deux ensembles disjoints U et V tel que chaque arête du graphe relie un sommet de U à un sommet de V.

Ci-contre un exemple de graphe biparti.



Voici une stratégie pour savoir si un graphe n'est pas biparti :

lors d'un parcours en largeur (BFS), attribuer une étiquette (U ou V) à chaque sommet : pour chaque sommet `s` *visité, on attribue à ses voisins (non encore visités) l'étiquette "opposée" à celle de* `s`*. Si deux sommets voisins ont à un moment donné la même étiquette, alors le graphe n'est pas biparti.*

Ecrire une fonction Python `est_biparti(G)` prenant en entrée un graphe `G` représenté par sa liste d'adjacence, et indiquant si ce graphe est biparti ou non, en suivant cette stratégie.

Ex 6 (3,5 pts) : Problème de la plus longue sous séquence commune (plssc)

Rappel du TD 8 : Le problème de la plus longue sous-séquence commune consiste à trouver un mot `x` (le plus grand possible) constitué de lettres communes à deux mots `s` et `t`, rencontrées dans le même ordre quand on parcourt ces mots de gauche à droite. Ces lettres ne sont pas forcément consécutives dans les mots.

Par exemple, la plus longue sous-séquence commune à '`abcde`' et '`bice`' est '`bce`', de longueur 3.

On peut déterminer cette plus longue sous-séquence commune de cette manière :

— si `s` ou `t` est vide, `x` est vide

— si les dernières lettres de `s` et `t` sont les mêmes, alors `x` termine par ce caractère

— sinon `x` est la plus longue des 2 plus longues sous-séquences communes suivantes : celle de `s` et `t` privée de sa dernière lettre, et celle de `t` et `s` privée de sa dernière lettre.

Nous avons vu dans le TD 8 qu'on pouvait obtenir la séquence recherchée par programmation dynamique. Nous adoptons une autre approche ici.

1) A partir de la caractérisation décrite ci-dessus, écrire une fonction Python **récursive** `plssc(s,t)` qui retourne la **longueur** de la plus longue sous-séquence commune de deux mots `s` et `t`.

2) On peut éviter les appels récursifs redondants de la fonction précédente et donc améliorer son efficacité en adoptant la technique de la **mémoïsation**, déjà utilisée en CM/TD pour les termes de la suite de Fibonacci et le problème du nombre minimal de pièces à rendre dans le rendu de monnaie.

Il faut ici mémoriser la longueur de la plus longue sous-séquence commune à `s` et `t` pour chaque couple constitué des longueurs de `s` et `t`. On utilisera donc ici un **dictionnaire** dont les clés sont ces couples (initialisé à vide avant la fonction), pour stocker ces longueurs.

Ecrire une fonction `plssc_mem(s,t)` implémentant cette technique.