CORRIGÉ DU DS 2

le 14/12/2024
Durée : 2h

Le devoir est constitué d'un exercice (sur 5 points) et d'un problème.

Exercice : Prévention des collisions aériennes

1.
```
SELECT id_aero
FROM aeroport
WHERE pays = 'France' ;
```

2.
```
SELECT COUNT(*)
FROM vol
WHERE jour='2016-05-02' AND heure<'12:00' ;
```

3.
```
SELECT V.id_vol
FROM vol AS V JOIN aeroport AS A
              ON V.depart=A.id_aero
WHERE A.ville='Paris' AND V.jour='2016-05-02' ;
```

4. Cette requête fournit les numéros des vols partant d'un aéroport en France et arrivant dans un aéroport en France le 2 mai 2016.

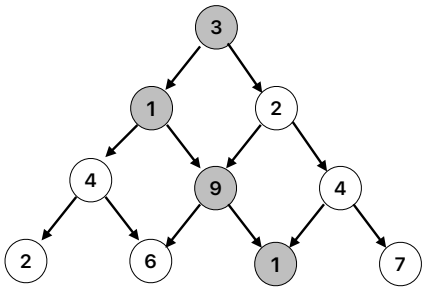
5.
```
SELECT V1.id_vol,V2.id_vol
FROM vol AS V1 JOIN  vol AS V2
               ON V1.depart=V2.arrivee AND V2.depart=V1.arrivee
                                       AND V1.niveau=V2.niveau
                                       AND V1.jour=V2.jour
WHERE V1.id_vol<V2.id_vol ;
```

La dernière condition est faite pour éviter les doublons (on a supposé que les chaînes de caractères peuvent être comparées).

Problème

I. Présentation

II. Représentation des données du problème



1.

Le score de ce chemin est $3+1+9+1$, c'est à dire 14.

2.
```
def est_chemin(ch,p):
    n = len(p)
    if len(ch) != n:
        return False
    elif ch[0] != 0:
        return False
    else :
        for i in range(n-1):
            if not (ch[i+1] == ch[i] or ch[i+1] == ch[i] + 1):
                return False

        return True
```

3.
```
def score(ch,p):
    n = len(p) # ou len(ch)
    S = 0
    for i in range(n):
        S += p[i][ch[i]]
    return S
```

4. À chaque étape de la construction du chemin on a 2 choix, ce qui fait 2^{n-1} chemins dans la pyramide. À chacune de ces étapes il faut additionner le score du disque. Puis ensuite il faut trouver le score maximal parmi les 2^{n-1} chemins. Cela donne à cette méthode une complexité exponentielle.

III. Stratégie gloutonne

5.
```
def score_glouton(p):
    n = len(p)
    i = 0 # niveau
    pos = 0 # position dans le niveau
    ch = [pos]
    score = p[i][pos]
    while i+1 < n:
        if p[i+1][pos] < p[i+1][pos+1]:
            pos = pos + 1
        # sinon, pos ne change pas
        i += 1
        ch.append(pos)
        score += p[i][pos]
    return score
```

6. Par cette stratégie, on obtient dans l'exemple de la figure 1 le score de $1+5+3+2+3$, c'est à dire 14.
Un autre chemin permet d'obtenir le score $1+2+4+5+6=18$.
On n'obtient donc par le score maximal par cette stratégie.

7. Pour cette méthode, on effectue une boucle de longueur $n-1$ avec à chaque étape une comparaison et une addition.
On a donc une complexité linéaire en n.

IV. Algorithme récursif naïf

8.
```
def scmax(p,i,j):
    n = len(p)
    if i == n-1:
        return p[i][j]
    else :
        return p[i][j] + max(scmax(p,i+1,j), scmax(p,i+1,j+1))
```

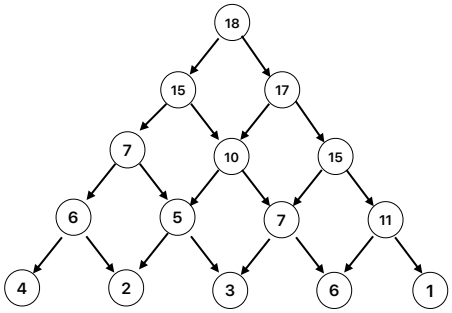
9.
```
def calcul_score_max(p):
    return scmax(p,0,0)
```

10. À chaque appel à cette fonction on lance de manière récursive deux appels. Il faut n étapes pour arriver à un cas de base. La complexité de cette fonction sera donc de l'ordre de 2^n.
On peut aussi formaliser cela grâce à la relation de récurrence entre le nombre d'opérations nécessaires à la résolution du problème pour une pyramide de hauteur i (c'est à dire pour un disque du niveau $n-1-i$), noté C_i, et le nombre d'opérations nécessaires à la résolution du problème pour une pyramide de taille $i-1$:

$$C_i = 2C_{i-1} + 2 \ .$$

On a compté deux opérations en plus des appels récursifs : une addition et une comparaison.
Cette relation de récurrence donne une complexité de l'ordre de 2^n.

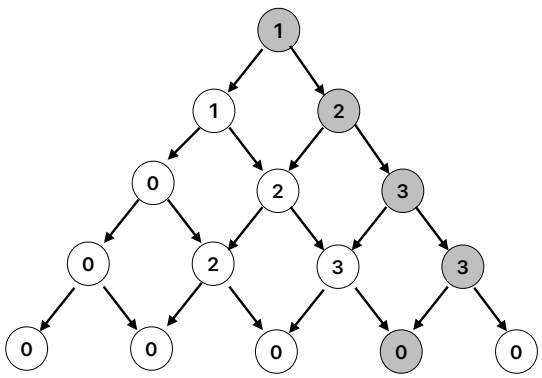
V. Programmation dynamique

11.


12.
```
def pyramide_nulle(n):
    l = []
    for i in range(n):
        l.append([0 for j in range(i+1)])
    return l
```

13.
```
def prog_dyn(p):
    n = len(p)
    s = pyramide_nulle(n)
    # remplissage du dernier rang (base) :
    for j in range(n):
        s[n-1][j] = p[n-1][j]
    i = n-2
    while i >= 0:
        for j in range(i+1):
            s[i][j] = p[i][j] + max(s[i+1][j],s[i+1][j+1])
        i -= 1
    return  s[0][0]
```

14. La boucle externe est de longueur $n-1$.
Lorsque l'on remplit le niveau i, la boucle interne est de longueur $i+1$.
À chaque étape de cette boucle on fait une addition et une comparaison (pour obtenir le maximum).
La complexité totale est donc en $\mathcal{O}(n^2)$.

15.


16.
```
def prog_dyn_suiv(p):
    n = len(p)
    s = pyramide_nulle(n)
    suiv = pyramide_nulle(n)
    # remplissage du dernier rang (base) de s:
       #(le dernier rang de suiv reste \‘a 0)
    for j in range(n):
        s[n-1][j] = p[n-1][j]
    i = n-2
    while i >= 0:
        for j in range(i+1):
            if s[i+1][j] < s[i+1][j+1] :
                s[i][j] = p[i][j] + s[i+1][j+1]
                suiv[i][j] = j+1
            else :
                s[i][j] = p[i][j] + s[i+1][j]
                suiv[i][j] = j
        i -= 1
    return  s[0][0] , suiv
```

17. À partir de la pyramide `suiv` que l'on a obtenue à la question 15, on reconstruit le chemin de score maximal de la manière suivante :

    - on part du niveau 0 ;
    - à chaque étape on descend d'un niveau en allant dans la position sur le niveau suivant qui est indiquée dans la case de `suiv` où l'on se trouve.

    Sur notre exemple on obtient le chemin `[0,1,2,3,3]`

18.
```
def reconstruit(suiv):
    n = len(suiv)
    j = 0
    ch = [j]
    for i in range(n-1):
        j = suiv[i][j]
        ch.append(j)
    return ch
```

VI. Résolution par mémoïzation

19.
```
def score_max_memo(p):
    n = len(p)
    s = pyramide_nulle(n)
    def remplissage_s(i,j):
        if s[i][j] == 0:
            if i == n-1:
                s[i][j] = p[i][j]
            else:
                remplissage_s(i+1,j)
                remplissage_s(i+1,j+1)
                s[i][j] = p[i][j] + max(s[i+1][j], s[i+1][j+1])
    remplissage_s(0,0)
    return s[0][0]
```