Algorithmique Avancée

Preuve et complexité

1.

Boucle while : Après la kième itération les k dernières cases du tableau sont triées et supérieures aux autres cases (elles sont donc disposées dans leur position finale).

Boucle for : Après la ième itération, l'élément d'indice i est supérieur à tous ceux qui le précèdent.

2.

On peut prouver inductivement l'invariant de boucle for :

A la ième itération, l'élément d'indice (i-1) est supérieur à tous ceux qui le précèdent. Si il est plus grand que le suivant on l'échange sinon il reste.

S'il y a échange alors le nouvel élément d'indice i est supérieur au nouvel élément d'indice i-1 et par récurrence est supérieur à tous ceux qui précèdent

S'il n'y a pas échange alors l'élément d'indice i est supérieur à l'élément d'indice i-1 qui par construction est supérieur à tous les éléments précédents.

L'invariant de boucle for est donc vrai.

Comme on itère sur tous les éléments pas encore triés du tableau, on itère sur le plus grand restant et on le place à la fin (invariant de boucle for) donc après la kième itération on a trié à la fin les k éléments les plus grands du tableau.

La boucle while itère sur tous les k une et une seule fois (k+=1 à la fin de la boucle). Donc le tableau finit par être triés après au maximum n itérations.

On a la correction et la terminaison.

3. ON fait des échanges pairs à pairs. Ainsi pour tout couple (i,j) tel que 0<=i<j<n, si A[i]>A[j], il y aura échange. Aussi on cherche la probabilité de l'évènement (0<=i<j<n, si A[i]>A[j]).

On peut placer à l'indice i un nombre k compris entre 1 et n et à l'indice j un nombre k' compris entre 1 et n k exclus, La probabilité que A[i]>A[j] est de k/n. Pour chaque couple (i,j) tel que 0<=i<j<n on a alors une probabilité de p= \sum_{k=1}^{n} \fract{k}{n²} = \fract{k}{n}

Et donc l'espérance du nombre d'échange en moyenne est E = \sum_{i=[1..n]} \sum_{j=1..n} \ fract {k} {n} = \fract{n+1}{2} * \fract{k}{n}

4. Moins il y a de « sections ordonnées » dans le tableau, plus il y aura d'appel à la boucle while. Ainsi le pire cas est la permutation [n..1]

Conception d'algorithmes

5.

explication : un algorithme en divide and conquer divise le tableau en plusieurs sous tableaux où le cas est plus simple à prédire. On pourrra ainsi revenir au cas simple de 1 tableau puis on porra remonter au tableaux de 2 et 3 cases, à partir de ces tableaux on pourra construire les tableaux de 4 5 et 6 cases etc. Jusqu'à pouvoir recontruire le tableau de départ. Le sous tableau ayant la plus grande somme sera alors le résultat

6.

7.

```
def hanoi(n):
"""
n est le nombre de disque
"""
    solved = {[swap(0,2)]} # réponse triviale pour un disque
    for i in range(1,n+1):
        solved[i] = solved[i-1] # on déplace les i - 1 plus petits disques grâce à la méthode connue
        solved[i] += [swap(0,1)] # on place le plus gros disque sur l'intermédiaire
        solved[i] += reverse(solved[i-1]) # on ramène les i - 1 plus petits disques au point de départ
        solved[i] += [swap(1,2)] # on place le plus gros disque sur le final
        solved[i] += solved[i-1] # on termine en mettant les plus petits disques sur le point d'arrivée
    return solved[n]

```

explication :

On notera sur l'exemple ci-dessus que le programme vise à tranférer les disques de la tige 0 vers la tige 2.

On commence par la possibilité la plus simple, celle où il n'y a qu'un seul disque. La solution est immédiate, il suffit de placer le disque unique sur la tige 2.

Pour la suite on suppose avoir résolu le problème pour i disques.

Pour i+1 disques on commence par déplacer les i disques sur la tige cible grâce à la méthode déjà connue grâce à notre mémoïsation.

Puis on déplace le plus gros disque désormais libéré vers la tige intermédiaire.

Si on sait faire le déplacement dans un sens on sait le faire dans l'autre donc on peut ramener la pile de i disques sur la tige 2 à la tige 0.

On peut alors passer le plus gros disque sur la tige. On se retrouve dans la situation pour i disques que l'on sait résoudre.

On note que l'on pourrait simplifier les instructions si on pouvait choisir sur quel tige on réalise le transfert. On pourrait alors faire le transfert de i disques du départ vers l'intermédiaire puis le i+1e disque du départ vers le final et finalement le transfert des i disques de l'intermédiaire vers le final.

8.

Grâce à la méthode de programmation dynamique ascendante on ne réalise qu'une seule fois chaque calcul. On passe donc seulement une fois dans la boucle : complexité temporelle en O(n). Par contre ces performances temporelles ont un coup : une complexité en mémoire en O(n).

Structure de données

9.

```
def join(A1, A2, x):
    A = newAVL()
    A.valeur = x
    A.gauche = A1
    A.droite = A2
    h1 = height(A1)
    h2 = height(A2)
    while not done:
    if abs(h2 - h1) < 2:
        done = True
    else
        if h2 > h1:
            while abs(height(focus.gauche) - height(focus.droite)) >= 2:
                A = rotate_left(A)
            focus = A.gauche
        else:
            while abs(height(focus.gauche) - height(focus.droite)) >= 2:
                A = rotate_right(A)
            focus = A.droite
    return A
```

On commence par faire un ABR à partir de A1, x et A2.

Une fois l'ABR créé, on veut en faire un AVL, donc dont la hauteur du fils gauche diffère de maximum 1 de la hauteur du fils droit.

Pour parvenir à cela on réalise des rotations jusqu 'à ce que ce soit le cas.

Ce faisant on risque d'avoir déséquilibré les arbres fils, aussi on refait sur le fils.