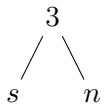
UE INF 152, Algorithmique et Programmation

Éléments de correction de la Session 1 2008-2009 et aide à la préparation de la Session 2.

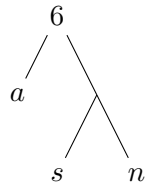
Exercice 1 1. Si on sait répondre à la question en bas du transparent 183, on sait citer un algorithme ayant une complexité asymptotique indépendante des entrées (ne dépendant que de la taille du tableau, et non de la distribution des éléments du tableau d'entrée).

Il suffit aussi de lire le transparent 203 pour pouvoir donner une deuxième solution.

2. Voir transparents 176 à 178, ou exercice 3, question 4.
3. Voir les derniers transparents du cours. On classe les lettres du texte `ananas` par fréquence croissante. On obtient 3 arbres ayant un seul sommet (où l'on inscrit les lettres aux feuilles et la fréquence à la racine) : $(s, 1)$, $(n, 2)$, et $(a, 3)$. On regroupe les lettres les moins fréquentes en un arbre, en faisant la somme de leurs fréquences. Il reste deux arbres : l'arbre $(a, 3)$ (une racine portant la lettre a et la fréquence 3), et l'arbre



On regroupe ces deux arbres en un seul, ce qui donne par exemple :



Pour obtenir le code de chaque lettre, on parcourt le chemin de la racine à la feuille portant la lettre, en écrivant 0 quand on prend une arête gauche et 1 quand on prend une arête droite. Ainsi, a est codé par `0`, s par `10`, et n par `11`. Donc, `ananas` est codé par `011011010`.

Remarque. Il y a d'autres codages que l'on peut obtenir, car l'algorithme laisse des choix. Par exemple, lorsqu'on regroupe deux arbres en un seul, on peut choisir quel arbre mettre comme fils gauche et quel arbre mettre comme fils droit de la racine. Par exemple, on aurait pu obtenir l'arbre symétrique au précédent.

Exercice 2 1. Beaucoup de copies présentent des fonctions erronées car l'énoncé n'a pas été correctement interprété : on demande de calculer le nombre d'éléments de `t` appartenant à l'intervalle `]a,b]` où `a` et `b` peuvent être des nombres beaucoup plus grands que la taille du tableau `t`.

Par exemple, si `t=[100,200,200,200,500,100,300,500]` et $a = 200$ et $b = 500$ la fonction retournera 3 car le tableau contient trois éléments entre `a` exclu et `b` inclus (précisement `t[6]` qui vaut 300, `t[4]` et `t[7]` qui valent 500).

Très souvent les fonctions erronées consideraient `a` et `b` comme étant des indices et ont comptabilisé le nombre d'éléments compris entre les deux indices (calcul qui ne necessite pas le passage du tableau en paramètre ! !).

Pour répondre correctement à cette question il faut donc parcourir le tableau et tester, pour chaque élément, s'il est supérieur à `a` et inférieur ou égal à `b`. Une variable (initialisée correctement !) sera utilisée pour comptabiliser les éléments qui satisfont le test et sera renvoyée par la fonction.

Puisque la fonction "visite" une seule fois chaque élément du tableau, la complexité est linéaire par rapport à la taille du tableau (on peut aussi dire que si n est la taille du tableau, la complexité est en $O(n)$, ou de l'ordre de grandeur de n).

2.
```
def nb_occurrences(t,x):
    nb = 0
    for y in t:
        if y == x:
            nb += 1
    return nb
```

ou bien

```
def nb_occurrences(t,x):
    nb = 0
    for i in range(len(t)):
        if t[i] == x:
            nb += 1
    return nb
```

La fonction effectue autant de comparaisons qu'il y a d'éléments dans `t`.

3. (a) Le programme affiche le tableau [0,2,3,1,0,2]. Sur l'exemple la fonction `nb_occurrences` est appelée 6 fois, car le maximum de `t` est 5 et l'appel à la fonction a lieu pour chaque valeur entre 0 et 5.

   (b) La fonction `f` retourne un tableau contenant à la case d'indice i la fréquence de la valeur i dans `t` : le tableau `[1,2,2,2,5,1,3,5]` ne contient pas la valeur 0, donc la case d'indice 0 du tableau [0,2,3,1,0,2] vaut 0, le tableau `[1,2,2,2,5,1,3,5]` contient deux fois la valeur 1, donc la case d'indice 1 du tableau [0,2,3,1,0,2] vaut 2, le tableau `[1,2,2,2,5,1,3,5]` contient trois fois la valeur 2, donc la case d'indice 2 du tableau [0,2,3,1,0,2] vaut 3, et ainsi de suite.

   (c) Durant une exécution de la fonction `f` la fonction `nb_occurrences` est appelée `maximum+1` fois. À chaque appel elle effectue autant de comparaisons qu'il y a d'éléments dans `t` : le nombre total de comparaisons effectuées par l'ensemble des appels est donc

   `len(t)` × `(maximum +1)`.

   (d)
```
def f_bis(t):
    maximum = max(t)
    tab = [0]*(maximum + 1)
    for x in t:
        tab[x] += 1
    return tab
```

   ou bien

```
def f_bis(t):
    maximum = max(t)
    tab = [0]*(maximum + 1)
    for i in range(len(t)):
        tab[t[i]] += 1
    return tab
```

4. Voir les transparents de cours 223 et 224 et faire tourner la fonction `Tri_comptage` sur le tableau `t=[1,2,2,2,5,1,3,5]`. On pourra vérifier que le tableau retourné par la fonction `f` est le même que le tableau appelé `C` dans le transparent 224.

5.
```
def cumul(v):
    R = [0]*len(v)
    R[0] = v[0]
    for i in range(1, len(v)):
        R[i] = R[i-1] + v[i]
    return R
```

6.
```
def nb_elements_bis(a,b,c):
    if b < len(c):
        return c[b]-c[a]
    return c[-1]-c[a]
```

En supposant que la longueur de `c` est calculée en temps constant, cette fonction a complexité constante (c.à.d. ne dépendant pas de la taille du tableau `c` car elle effectue toujours une comparaison et une soustraction). On peut dire aussi que la fonction est en $O(1)$ (voir transparent 166).

Exercice 3 1. On parcourt le tableau `T` en recherchant `x` :

```
def appartient(x, T):
    for y in T:
        if x == y:
            return True
    return False
```

2. Ne pas oublier de tester d'abord les cas de base de la récursion :

```
def appartient_rec(x, T, i, j):
    if i > j:
        return False
    if T[i] == x:
        return True
    return appartient_rec(x, T, i+1, j)

def appartient_bis(x, T):
    return appartient_rec(x, T, 0, len(T)-1)
```

3. On parcourt `T1` et pour chacun de ses éléments, on teste s'il est dans `T2` :

```
def card_intersection(T1,T2):
    card = 0
    for x in T1:
        if appartient(x,T2):
            card +=1
    return card
```

Complexité : pour chaque élément de `T1`, on parcourt, dans le cas le pire, le tableau `T2` tout entier. La complexité est donc `len(T1)` × `len(T2)`.

4. On fait une recherche dichotomique : on compare l'élément cherché avec celui en milieu du tableau. S'il est égal, on retourne `True`. Et comme le tableau est trié dans l'ordre décroissant :
   – S'il est plus petit, on le cherche dans le 1/2 tableau de droite.
   – S'il est plus grand, on le cherche dans le 1/2 tableau de gauche.

```
def appartient_tab_trie_aux(x, T, i, j):
    if i > j:
        return False
    milieu = (i+j)/2
    if x == T[milieu]:
        return True
    if x < T[milieu]:
        return appartient_tab_trie_aux(x, T, milieu+1, j)
    return appartient_tab_trie_aux(x, T, i, milieu-1)

def appartient_tab_trie(x,T)    :
    return appartient_tab_trie_aux(x, T, 0, len(T)-1)
```

Comme la taille du tableau dans lequel on cherche est divisée par 2 à chaque fois, la complexité est $O(\log(n))$ où n est la taille du tableau `T`.

5. On parcourt les tableaux de gauche à droite :

```
def intersection(T1,T2):
    R = []
    i,j,n1,n2 = 0,0,len(T1),len(T2)
    while i < n1 and j < n2:
        if T1[i] == T2[j]:
            R += [T1[i]]
            i += 1
            j += 1
        elif T1[i] < T2[j]:
            j += 1
        else:
            i += 1
    return R
```

À chaque itération de la boucle `while`, au moins l'un des deux indices `i` ou `j` est incrémenté. Comme on sort de cette boucle dès que `i` ou `j` atteignent la taille de `T1` ou `T2`, respectivement, la complexité est $O(\texttt{len(T1)} + \texttt{len(T2)})$ dans le cas le pire.