Table of Contents

1 Contrôle continu de mars 2017

1.1 Exercice 1

1.1.1 Q :

Ecrire l'en-tête d'une fonction `nbOcc( )` qui compte le nombre d'occurrences d'une valeur donnée dans un ensemble de n valeurs. On se pla valeurs est un tableau de caractères `s[n]`, i.e. une chaîne de caractères de longueur n.

```
def nbOcc(c, s, n):
    '''compte et retourne le nbre d occurrences res
    du caractère c dans la chaine de caractère s de longueur n
    entrées
    . c : char , caratère recherché
    . s : tableau de char (ou string) de longueur n
    . n : longueur de s
    retourne l entier res
    '''
```

1.1.2 Q :

Ecrire la fonction `nbOcc( )` : solution itérative.

In [22]:
```
def nbOcc(c, s, n):
    '''compte et retourne le nbre d occurrences res
    du caractère c dans la chaine de caractère s de longueur n'''
    res = 0
    for i in range(n):
        if s[i] == c:
            res = res + 1
    return res
```

1.1.3 Q :

Utiliser `nbOcc( )` pour compter les nombres d'occurrences suivantes :
a. nombre de `a` dans `licence` ?
b. nombre de `i` dans `licence` ?
c. nombre de `c` dans `licence` ?
Les nombres correspondants seront stockées dans des variables `n0, n1, n2` qui seront ensuite affichées.

In [23]:
```
''' Utilise nbOcc( ) pour compter les nombres d'occurrences de lettre dans mot'''
mot = "licence"

n0 = nbOcc('a', mot, 7)
n1 = nbOcc('i', mot, 7)
n2 = nbOcc('c', mot, 7)

print(n0, n1, n2)
```

```
0 1 2
```

1.1.4 Q :

Justifier le choix de la structure de contrôle utilisée.

On utilise une boucle `for ... in range[n]` car le nombre d'itérations pour parcourir la chaîne de caractères est connu a priori : il vaut n : l

1.1.5 Q :

En déduire la complexité asymptotique, en temps, de `nbOcc`. Préciser la mesure et le paramètre de complexité associés.

- mesure : on compte le nombre de comparaisons (`if` dans le corps de la boucle).
- paramètre de complexité : la longueur de la chaîne, soit n.
- analyse :
  - nombre de comparaisons `if` dans le corps de la boucle = 1
  - corps de boucle exécuté n fois **exactement**.

Donc la complexité asymptotique est linéaire en la longueur de la chaîne, soit $\theta(n)$.

Remarque Ici le nombre fixé d'itérations de la boucle `for` permet d'écrire un équivalent asymptotique ($\theta()$) de la complexité.

1.1.6 Q* :

Ecrire `nbOccRec( )`, version récursive de `nbOcc( )`.

In [24]:
```
def nbOccRec(c, s, n):
    '''compte et retourne le nbre d occurrences
    du caractère c dans la chaine de caractère s de longueur n.
    - (une) version recursive
    entrées:
    c : char
    s : tableau de char (string)
    n : int '''
    # terminaison quand s est vide i.e; de longueur 0
    if n == 0:
        return 0
    else:
        # test occurrence sur dernier caractere de s : en place n-1
        # puis appel recursif sur chaine qui précède : de longueur n-1
        if s[n-1] == c:
            return nbOccRec(c, s, n-1) + 1
        else:
            return nbOccRec(c, s, n-1)
```

1.1.7 Q :

Utiliser `nbOccRec( )` pour compter les nombres d'occurrences suivantes :
a. nombre de `a` dans `licence` ?
b. nombre de `i` dans `licence` ?
c. nombre de `c` dans `licence` ?
Les nombres corresponsdants seront stockées dans des variables `n0, n1, n2` qui seront ensuite affichées.

In [25]:
```
''' Utilise nbOccRec( ) pour compter les nombres d'occurrences de lettre dans mot'''
mot = "licence"

nr0 = nbOccRec('a', mot, 7)
nr1 = nbOccRec('i', mot, 7)
nr2 = nbOccRec('c', mot, 7)

print(nr0, nr1, nr2)
```

```
0 1 2
```

Remarque : c'est le même appel que pour la version itérative (à l'identificateur de la fonction près).

1.1.8 Q * :

En notant `nbOccRec` par `f` (pour faire plus court), exhiber l'évolution de la pile des appels à `f` pour les décomptes suivants :

1. `nbOccRecPrint('e', "ete", 3)`
2. `nbOccRecPrint('a', "ete", 3)`

Cette question est traitée automatiquement avec les questions/réponses supplémentaires suivantes.

1.1.9 Q** :

Proposer une fonction `nbOccRecPrint( )` : une version recursive avec affichage pour exhiber l'évolution de la pile des appels.

In [26]:
```
def nbOccRecPrint(c, s, n, pile):
    '''compte et retourne le nbre d occurrences res
    du caractère c dans la chaine de caractère s de longueur n.
    - version recursive avec affichage pour exhiber la pile :
    . on affiche l état de la pile après chaque modification de la pile
    . on empile `f(n)` pour un appel de `f(n)`
    . on dépile le haut de la pile pour un return
    '''
    if n == 0:
        return 0
    else:
        pile.append('f' + str(n-1))
        print(pile)
        res = nbOccRecPrint(c, s, n-1, pile)
        if s[n-1] == c:
            pile.pop(len(pile)-1)
            print(pile)
            return res + 1
        else:
            pile.pop(len(pile)-1)
            print(pile)
            return res
```

L'exécution suivante exhibe les 2 évolutions demandées de la pile des appels (et une supplémentaire).

In [27]:
```
nn = nbOccRecPrint('e', "ete", 3, [])
print(nn)
nn = nbOccRecPrint('a', "ete", 3, [])
print(nn)
# pour le plaisir de regarder python bosser :)
nr2 = nbOccRecPrint('c', "licence", 7, [])
print(nr2)
```

```
['f2']
['f2', 'f1']
['f2', 'f1', 'f0']
['f2', 'f1']
['f2']
[]
2
['f2']
['f2', 'f1']
['f2', 'f1', 'f0']
['f2', 'f1']
['f2']
[]
0
['f6']
['f6', 'f5']
['f6', 'f5', 'f4']
['f6', 'f5', 'f4', 'f3']
['f6', 'f5', 'f4', 'f3', 'f2']
['f6', 'f5', 'f4', 'f3', 'f2', 'f1']
['f6', 'f5', 'f4', 'f3', 'f2', 'f1', 'f0']
['f6', 'f5', 'f4', 'f3', 'f2', 'f1']
['f6', 'f5', 'f4', 'f3', 'f2']
['f6', 'f5', 'f4', 'f3']
['f6', 'f5', 'f4']
['f6', 'f5']
['f6']
[]
2
```

1.2 Exercice 2

1.2.1 Q : fonction minIt() qui calcule le min d'un tableau d'enteirs : version itérative

In [28]:
```
def minIt(t, n):
    '''calcule le min d'un tableau d'entiers : version itérative
    entrées : t : tableau d int
    n: int, sa taille
    retourne int
    '''
    res = t[0]
    for i in range(n):
        if t[i] < res:
            res = t[i]
    return res
```

Exemple d'appels :

In [29]:
```
tt = [8, 2, 4, 1, 5]
m = minRec(tt, 0, len(tt))
print(m)

tt_up = [i for i in range(10)]
tt_do = [i for i in range(9, 0, -1)]
m = minIt(tt_up, 10)
print("min de", tt_up, "= ", m)

mm = minIt(tt_do, len(tt_do)) # remarquer l utilisation de la fonction len()
print("min de", tt_do, "= ", mm)
```

```
1
min de [0, 1, 2, 3, 4, 5, 6, 7, 8, 9] =  0
min de [9, 8, 7, 6, 5, 4, 3, 2, 1] =  1
```

1.2.2 Q :Complexité asymptotique de cette solution itérative

- mesure : on compte le nombre de comparaisons (`if` dans le corps de la boucle).
- paramètre de complexité : la longueur de la chaîne, soit n.
- analyse :
  - nombre de comparaisons `if` dans le corps de la boucle = 1
  - corps de boucle exécuté n fois **exactement**.

Donc la complexité asymptotique est linéaire en la longueur de la chaîne, soit $\theta(n)$.

Remarque Ici le nombre fixé d'itérations de la boucle `for` permet d'écrire un équivalent asymptotique ($\theta()$) de la complexité.

1.2.3 Q :

Propriété récursive du min d'un tableau connaissant une fonction min(a,b).

Propriété récursive : min(a0, a1, ... a_n-1) = min(a0, min(a1, a2, ..., a_n-1))
Cas terminal avec la fonction min(a,b) quand la longueur du tableau == 2

Remarque : la propriété précédente exhibe le besoin de gérer les indices de début des sous-tableaux gauche au fur et à mesure des appels ré
fonction récursive doit inclure cet indice (en plus de la taille du sous-tableau).

1.2.4 Q (*):

In [30]:
```
def minRec(t, g, d):
    '''min de t entre les indices g et d-1
    version récursive
    entrées.
    t : tableau d int
    g, d : int
    retourne int'''
    if  d - g == 2: # terminaison : 2 élements dans le sous-tab
        return min(t[g], t[d-1])
    else:
        return min(t[g], minRec(t, g+1, d))
```

In [31]:
```
'''exemple d appels'''
tt = [8, 2, 4, 1, 5]
m = minRec(tt, 0, len(tt))
print(tt, "--> ", m)

tt_up = [i for i in range(10)]
m = minRec(tt_up, 0, len(tt_up))
print(tt_up, "--> ",m)

tt_do = [i for i in range(9, 0, -1)]
m = minRec(tt_do, 0, len(tt_do))
print(tt_do, "--> ",m)
```

```
[8, 2, 4, 1, 5] -->  1
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9] -->  0
[9, 8, 7, 6, 5, 4, 3, 2, 1] -->  1
```

1.2.5 Complexité de la version itérative

La complexité de ce min récursif est aussi linéaire : l'écriture récursive ne fait que remplacer la boucle des itérations de minIt(). La comparaisor min() : 1 comparaison par appel à min(). Il y a exactement (n-1) appels à min() donc la complexité est bien $\theta(n)$.

In [ ]: