UNIVERSITÉ BORDEAUX 1 Sciences Technologies DISVE – Licence	Année : 2011/2012 Semestre 2 **PARCOURS :** Licence LIMI201 & LIMI211 **UE J1MI2013 :** Algorithmes et Programmes **Épreuve :** Devoir Surveillé Terminal **Date :** Lundi 11 juin 2012 **Heure :** 8 heures 30 **Durée :** 1 heure 30 Documents : non autorisés Épreuve de M. Alain Griffault

SUJET + CORRIGE

Avertissement

- La plupart des questions sont indépendantes.
- Le sujet est sans doute un peu long.
- Les indentations des fonctions écrites en `Python` doivent être respectées.
- L'espace laissé pour les réponses est suffisant (sauf si vous utilisez ces feuilles comme brouillon, ce qui est fortement déconseillé).

Question	Points	Score
Récursivité	9	
Le tri par base	11	
Variante tri rapide	10	
Complexité	10	
Total:	40	

Exercice 1 : Récursivité (9 points)

Le nombre de combinaisons C_n^p représente le nombre de sous-ensembles de cardinal p d'un ensemble de cardinal n. Il est défini par $C_n^p = 1$ si $p = 0$ ou si $p = n$, et par $C_n^p = C_{n-1}^{p-1} + C_{n-1}^p$ dans le cas général. Une propriété intéressante pour calculer les combinaisons est : $C_n^p = \frac{n \times C_{n-1}^{p-1}}{p}$.

(a) Soit la fonction `combRec(n,p)` où `n` et `p` sont deux entiers positifs tels que $p \leq n$.

```
def combRec(n,p):
    if (p == 0 or n == p):
        return 1
    else:
        return n * combRec(n-1,p-1) // p
```

i. (1½ points) Compléter le tableau suivant pour simuler la pile d'exécution lors de l'appel `combRec(7,3)`.

Solution:

appels successifs ↑ ↓ fin des appels

combRec	4	0	1
combRec	5	1	5
combRec	6	2	15
combRec	7	3	35
fonction	n	p	retour

Soit les fonctions `combRecV2(n,p)` et `combRecUV(n,p,u,v)` où `n` et `p` sont deux entiers positifs tels que $p \leq n$, et `u` et `v` deux paramètres supplémentaires.

```
def combRecUV(n,p,u,v):
    if (p == 0 or n == p):
        return 1
    else:
        return n * combRecUV(n-1,p-1,u*n,v*p) // p

def combRecV2(n,p):
    return combRecUV(n,p,1,1)
```

ii. (1½ points) Compléter le tableau suivant afin de simuler la pile d'exécution lors d'un appel à `combRecV2(7,3)`.

Solution:

fonction	n	p	u	v	retour
combRecUV	4	0	210	6	1
combRecUV	5	1	42	6	5
combRecUV	6	2	7	3	15
combRecUV	7	3	1	1	35
combRecV2	7	3			35

(b) (3 points) Si vous n'avez pas fait d'erreur lors de vos simulations, vous devez constater que dans le second tableau, la valeur $\frac{u}{v}$ lors du dernier appel récursif est égal au résultat final. C'est donc qu'il est inutile de propager le résultat jusqu'à l'appel initial.
Écrire une fonction python récursive terminale `combRecAux(n,p,u,v)` qui calcule le nombre de combinaisons C_n^p, et préciser l'appel initial à cette fonction dans une fonction python `combRecTerm(n,p)`.

Solution:

```
def combRecAux(n,p,u,v):
    if (p == 0 or n == p):
        return u // v
    else:
        return combRecAux(n-1,p-1,u*n,v*p)

def combRecTerm(n,p):
    return combRecAux(n,p,1,1)
```

(c) (3 points) Écrire une fonction python itérative `combIter(n,p)`, dérivée *automatiquement* de la fonction `combRecAux(n,p,u,v)` qui calcule le nombre de combinaisons C_n^p.

Solution:

```
def combIter(n,p):
    u = 1
    v = 1
    while True:
        if (p == 0 or n == p):
            return u // v
        else:
            # n,p,u,v = n-1,p-1,u*n,v*p
            u = u*n
            v = v*p
            n = n-1
            p = p-1
```

Exercice 2 : Le tri par base (11 points)

Nous souhaitons trier un tableau de personnes suivant leurs dates de naissance. Nous allons étudier deux approches : La première utilise un algorithme de tri *classique* qui utilise une fonction de comparaison de deux personnes ; et la seconde est une adaptation de l'algorithme de tri par base qui a été beaucoup utilisé autrefois pour trier les cartes perforées. Dans la suite, nous supposerons que les personnes sont stockées sous forme de tableaux.

(a) (2 points) Compléter la fonction python `neAvant(P1,P2)` de comparaison de deux personnes `P1` et `P2` qui retourne `Vrai` si `P1` est née avant ou le même jour que `P2`, et `Faux` sinon.

Cette fonction `neAvant(P1,P2)` pourra être utilisée dans n'importe lequel des algorithmes de tri vus en cours pour trier un tableau de `N` personnes suivant leur date de naissance. Il suffira de remplacer les tests `T[i]<=T[j]` par `neAvant(T[i],T[j])`.

Solution: Trois solutions (un prédicat, des branchements, une itération) parmi d'autres.

```
# p1 et p2 sont deux tableaux de 3 entiers
# p1[0] et p2[0] contiennent les jours de naissance
# p1[1] et p2[1] contiennent les mois de naissance
# p1[2] et p2[2] contiennent les annees de naissance

def neAvant(p1,p2):
    return ((p1[2] < p2[2]) or
            ((p1[2] == p2[2]) and (p1[1] < p2[1])) or
            ((p1[2] == p2[2]) and (p1[1] == p2[1]) and (p1[0] <= p2[0])))

def neAvantV1(p1,p2):
    if p1[2]==p2[2]:
        if p1[1]==p2[1]:
            return p1[0] <= p2[0]
        return p1[1] < p2[1]
    return p1[2] < p2[2]

def neAvantV2(p1,p2):
    for i in range(2,-1,-1):
        if p1[i]<p2[i]:
            return True
        if p1[i]>p2[i]:
            return False
    return True
```

(b) Le tri par base des dates de naissances de personnes.

Définition (rappel) : Un algorithme de tri est *stable* si l'ordre des indices de deux valeurs égales est inchangé dans le tableau trié. Par exemple si $T[5] = T[36]$ dans le tableau non trié, alors si i et j sont les nouveaux indices de $T[5]$ et de $T[36]$ dans le tableau trié, alors $i < j$.

Le tri par base consiste à appliquer une suite de tris stables sur des parties de l'objet à trier. Pour une date de naissance, il faut effectuer séquentiellement trois tris *stables* :

1. Trier (avec un tri stable) suivant le jour de naissance.
2. Trier (avec un tri stable) suivant le mois de naissance.
3. Trier (avec un tri stable) suivant l'année de naissance.

i. (2 points) Soit le tableau suivant de trois personnes avec leurs dates de naissances.

indice	0	1	2
jour	14	22	22
mois	3	1	3
année	2004	2004	2001

Compléter le tableau suivant après chacune des trois étapes.

Solution:

	Après le tri stable jours de naissance			Après le tri stable mois de naissance			Après le tri stable années de naissance		
indice	0	1	2	0	1	2	0	1	2
jour	14	22	22	22	14	22	22	22	14
mois	3	1	3	1	3	3	3	1	3
année	2004	2004	2001	2004	2004	2001	2001	2004	2004

ii. (2 points) Il est connu que le tri par base n'est pas correct si le tri utilisé n'est pas *stable*. Au choix, justifier cette affirmation, ou bien simuler sur l'exemple précédent un tri par base utilisant le tri par sélection.

Solution: Par exemple, la dernière étape peut donner avec un tri non stable :

Après un tri **par sélection non stable**.

indice	0	1	2
jour	22	14	22
mois	3	3	1
année	2001	2004	2004

Ce qui démontre que le tri base doit n'utiliser que des tris stables.

iii. (1 point) Donner la liste des tris stables vus en cours.

Solution:

```
def triStable(t,clef):
    # clef indice le second indice a utiliser pour le tri
    # N'importe lequel des 3 tris stables suivants peut etre utilise
    # triInsertion(t,clef)
    triBulle(t,clef)
    # triFusion(t,clef)
```

iv. (2 points) Compléter la fonction python `triBase(t)` pour trier un tableau de `N` personnes suivant leurs dates de naissances. Vous pourrez utiliser (sans en donner le code) une fonction `triStable(t,clef)` qui trie un tableau `t` à deux dimensions en fonction des valeurs `t[i][clef]`. Par exemple si `t` est un tableau de personnes, l'appel `triStable(t,1)` trie les personnes en fonction des mois de naissances.

Solution:

```
def triBase(t):
    # t[i][0] est le jour de naissance de la personne t[i]
    # t[i][1] est le mois de naissance de la personne t[i]
    # t[i][2] est l'annee de naissance de la personne t[i]
    for clef in range(3):
        triStable(t,clef)
```

v. (2 points) Donner et justifier la complexité de votre algorithme `triBase` en fonction de l'algorithme de tri *stable* utilisé.

Solution: Le tri appelle un nombre de fois déterminé (ici 3) un algorithme de tri stable. La complexité du tri par base est donc celle du tri stable utilisé soit :
- $\Theta(n^2)$ pour les tris par insertion et à bulles.
- $\Theta(nlog_2(n))$ pour le tri par fusion.

Exercice 3 : Variante tri rapide (10 points)

(a) (2 points) Justifier la validité de la fonction python `partitionner(t,g,d)` suivante, dans laquelle les itérations `Répéter` de l'algorithme 3 du tri rapide vu en cours sont remplacées par des itérations `Tant que`.

Solution: Les deux structures de programme suivantes sont équivalentes.

Algorithm 1 `Répéter ...jusqu'à`

1: **repeat**
2: S
3: **until** P

$\equiv$

Algorithm 2 `Tant que ...`

1: S
2: **while** $\neg P$ **do**
3: S
4: **end while**

Algorithm 3 `Partitionner(t,g,d)`

Condition : $0 \leq g < d \leq len(t) - 1$
1: $pivot \leftarrow t[g]$
2: $i \leftarrow g - 1$
3: $j \leftarrow d + 1$
4: **while true do**
5: **repeat**
6: $j \leftarrow j - 1$
7: **until** $t[j] <= pivot$
8: **repeat**
9: $i \leftarrow i + 1$
10: **until** $t[i] >= pivot$
11: **if** $i < j$ **then**
12: $Echange(t, i, j)$
13: **else**
14: **return** j
15: **end if**
16: **end while**
Garantie : $\forall k \leq j, t[k] \leq pivot$
$\forall k > j, t[k] \geq pivot$
Complexité : $\Theta(d - g)$

Algorithm 4 `TriRapideRec(t,g,d)`

Condition : $0 \leq g < d \leq len(t) - 1$
1: **if** $g < d$ **then**
2: $m \leftarrow Partitionner(t, g, d)$
3: $TriRapideRec(t, g, m)$
4: $TriRapideRec(t, m + 1, d)$
5: **end if**
Garantie : $\forall k \in [g, d[, t[k] \leq t[k + 1]$
Complexité : $\Omega((d - g)log_2(d - g))$,
$\mathcal{O}((d - g)^2)$

Algorithm 5 `TriRapide(t)`

1: $TriRapideRec(t, 0, len(t) - 1)$
Complexité : $\Omega(nlog_2(n)), \mathcal{O}(n^2)$
avec $n = len(t)$

```
1  def echange(t,i,j):
2      TMP = t[i]
3      t[i] = t[j]
4      t[j] = TMP
5
6  def partitionner(t,g,d):
7      pivot = t[g]
8      i = g-1
9      j = d+1
10     while True:
11         j -= 1
12         while t[j] > pivot:
13             j -= 1
14         i += 1
15         while t[i] < pivot:
16             i += 1
17         if i < j:
18             echange(t,i,j)
19         else:
20             return j
21
22 def triRapideRec(t,g,d):
23     if g < d:
24         m = partitionner(t,g,d)
25         triRapideRec(t,g,m)
26         triRapideRec(t,m+1,d)
27
28 def triRapide(t):
29     triRapideRec(t,0,len(t)-1)
```

(b) Soient l'algorithme du drapeau hollandais vu en cours et une écriture possible en python.

Algorithm 6 `Drapeau(t)`

Condition : $\forall i \in [0, len(t)-1], t[i] \in [0,2]$

1: $i \leftarrow 0$
2: $j \leftarrow i$
3: $k \leftarrow len(t)-1$
4: **while** $(j \leq k)$ **do**
5: **if** $t[j] = 1$ **then** // 1 : Blanc
6: $j \leftarrow j+1$
7: **else if** $t[j] = 0$ **then** // 0 : Bleu
8: $Echange(t, i, j)$
9: $i \leftarrow i+1$
10: $j \leftarrow j+1$
11: **else** // 2 : Rouge
12: $Echange(t, j, k)$
13: $k \leftarrow k-1$
14: **end if**
15: **end while**
16: **return** i, j, k

Garantie : $\forall l \in [0, i[, t[l] = 0$
$\forall l \in [i, j[, t[l] = 1$
$\forall l \in]k, len(t)-1], t[l] = 2$

Complexité : $\Theta(n)$, avec $n = len(t)$

```
1  def drapeau(t):
2      # t est un tableau de 0, 1 ou 2
3      # 0 : Bleu, 1 : Blanc, 2 : Rouge
4      i = 0
5      j = i
6      k = len(t) - 1
7      while j <= k:
8          if t[j] = 1:
9              j += 1
10         else:
11             if t[j] == 0:
12                 echange(t,i,j)
13                 i += 1
14                 j += 1
15             else:
16                 echange(t,j,k)
17                 k -= 1
18     return i,j,k
```

i. (1½ points) Soit `t` un tableau d'entiers compris entre `0` et `2`. Soient `g` et `d` deux indices tels que $0 \leq g < d \leq (len(t)-1)$. Écrire un algorithme ou bien un programme python `drapeauIntervalle(t,g,d)` qui trie les variables `t[i]` pour les indices `i` compris dans l'intervalle fermé $[g, d]$, et laisse les autres variables du tableau inchangées. *Vous* **devez** *n'indiquer que les différences avec l'algorithme 6 ou avec le programme python* `drapeau`.

Solution: Les lignes 4, 5 et 6 sont remplacées par :

```
i = g
j = i
k = d
```

ii. (1½ points) Soit `t` un tableau d'entiers quelconques. Soient `g` et `d` deux indices tels que $0 \leq g < d \leq (len(t)-1)$. Écrire un algorithme ou bien un programme python `drapeauPartition(t,g,d)` qui retourne trois indices `i,j,k` tels que $\forall l \in [0, i[, t[l] < pivot$, $\forall l \in [i, j[, t[l] = pivot$ et $\forall l \in]k, len(t)-1], t[l] > pivot$ où $pivot$ est une valeur quelconque du tableau entre les indices `g` et `d`. *Vous* **devez** *n'indiquer que les différences avec l'algorithme 6 ou avec le programme python* `drapeau`.

Solution: Les lignes 4, 5 et 6 sont remplacées par :

```
i = g
j = i
k = d
pivot = t[g]
```

Les lignes 8 et 11 sont remplacées par :

```
if t[j] == pivot:
    if t[j] < pivot:
```

(c) i. (2 points) Soit `t` un tableau d'entiers. Soient `g` et `d` tels que $0 \leq g < d \leq (len(t)-1)$. Écrire un algorithme ou un programme python récursif `triRapideDrapeauRec(t,g,d)`

et un algorithme ou un programme python `triRapideDrapeau(t)` qui optimisent les algorithmes 4 et 5 de tri rapide en tirant profit de la fonction `drapeauPartition(t,g,d)`.

Solution:

```
def triRapideDrapeauRec(t,g,d):
    if g < d:
        i,j,k = drapeauPartition(t,g,d)
        triRapideDrapeauRec(t,g,i-1)
        triRapideDrapeauRec(t,k+1,d)

def triRapideDrapeau(t):
    triRapideDrapeauRec(t,0,len(t)-1)
```

ii. (1½ points) Soit `t` un tableau d'entiers dont toutes les valeurs sont identiques. Donner et justifier le nombre d'appels récursifs de votre algorithme `triRapideDrapeauRec(t,g,d)` pour l'appel `triRapideDrapeau(t)`.

Solution: Pour ce tableau particulier, `triRapideDrapeauRec(t,0,len(t)-1)` va appeler `drapeauPartition(t,0,len(t)-1)` qui retournera `0,len(t),len(t)-1`. Il y aura ensuite un appel récursif à `triRapideDrapeauRec(t,0,-1)` qui terminera aussitôt, suivi d'un appel récursif à `triRapideDrapeauRec(t,len(t),len(t)-1)` qui terminera également aussitôt.

iii. (1½ points) Donner et justifier les complexités de votre algorithme `triRapideDrapeau(t)`.

Solution:
- Meilleur des cas (toutes les valeurs sont égales) : $\Omega(n)$.
- Pire des cas (tableau trié de valeurs différentes) : $\mathcal{O}(n^2)$.

Exercice 4 : Complexité (10 points)

(a) Soit la fonction `indiceTrie(t)` où `t` est un tableau de `n` entiers :

```
def indiceTrie(t):
    i = 0
    while ((i+1) < len(t)) and (t[i] <= t[i+1]):
        i += 1
    return i
```

i. (1 point) Donner le nombre d'affectations effectuées sur la variable `i` dans la fonction `indiceTrie(t)` en fonction de la valeur de retour `i` ?

Solution:
- Nombre d'affectations : `i+1`

ii. (1 point) Déduire les complexités en nombre d'affectations de la fonction `indiceTrie(t)` en fonction de `n` (taille du tableau `t`) ?

Solution:
- Meilleur des cas : $\Omega(1)$.
- Pire des cas : $\mathcal{O}(n)$.

(b) (2 points) Soit la fonction `tri(t,g,d)` où `t` est un tableau de `n` entiers et `g` et `d` deux entiers compris entre `0` et `n-1`. `tri(t,g,d)` utilise la fonction classique `echange(t,i,j)` de la page 5.

```
def tri(t,g,d):
    if g >= 0 and g < d and d <= (len(t)-1):
        for i in range(g,d):
            for j in range(i+1,d+1):
```

```
    if t[i] > t[j]:
        echange(t,i,j)
```

Donner et justifier les complexités en nombre d'affectations sur les variables `t[k]` de la fonction `tri(t,g,d)` en fonction des paramètres `g` et `d` ?

Solution:
- Meilleur des cas : 0 affectation soit $\Omega(1)$. Cas du tableau trié.
- Pire des cas : $3(d-g)^2$ affectations soit $\mathcal{O}((d-g)^2)$. Cas du tableau trié à l'envers.

(c) Soit la fonction `triErrone(t)` où `t` est un tableau de `n` entiers :

```
def triErrone(t):
    i = indiceTrie(t)
    tri(t,i+1,len(t)-1)
```

i. (2 points) Donner le nombre d'affectations effectuées sur la variable `i` et sur les variables `t[k]` du tableau lors d'un appel à la fonction `triErrone(t)` en fonction de la variable `i` et de `n` (taille du tableau `t`) ?

Solution:
- Nombre minimum d'affectations : i
- Nombre maximum d'affectations : $i + 3 * (n-i)^2$

ii. (2 points) Déduire les complexités en nombre d'affectations de la fonction `triErrone(t)` en fonction de `n` (taille du tableau `t`) ?

Solution:
- Meilleur des cas : $\Omega(1)$.
- Pire des cas : $\mathcal{O}(n^2)$.

iii. (2 points) Expliquer pourquoi la fonction `triErrone(t)` peut ne pas trier le tableau `t`.

Solution: La fonction `triErrone(t)` retourne un tableau constitué de deux suites croissantes : la première va des indices `0` à `i`, et la seconde des indices `i+1` à `len(t)-1`. De plus si `i` n'est pas égal à `len(t)-1`, nous savons que $t[i+1] < t[i]$. Dans ce cas, le tableau `t` n'est pas trié.