



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)`.

↑						↓
	combRecUV	7	3	1	1	
	combRecV2	7	3			
	fonction	n	p	u	v	retour

(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)`.

(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  $C_n^p$ .

**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 supposons 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])`.

```
# 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 années de naissance
```

- (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.

indice	Après le tri stable des jours de naissance			Après le tri stable des mois de naissance			Après le tri stable des années de naissance		
	0	1	2	0	1	2	0	1	2
jour									
mois									
année									

- 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.
- iii. (1 point) Donner la liste des tris stables vus en cours.
- 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.
- ```
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):
```
- v. (2 points) Donner et justifier la complexité de votre algorithme `triBase` en fonction de l'algorithme de tri *stable* utilisé.

**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**.

**Algorithm 3 Partitionner(t,g,d)****Condition :**  $0 \leq g < d \leq \text{len}(t) - 1$ 

```

1: pivot ← t[g]
2: i ← g - 1
3: j ← d + 1
4: while true do
5:   repeat
6:     j ← j - 1
7:   until t[j] ≤ pivot
8:   repeat
9:     i ← 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 :  $\forall k \leq j, t[k] \leq \text{pivot}$ 
 $\forall k > j, t[k] \geq \text{pivot}$ 
Complexité :  $\Theta(d - g)$ 

```

**Algorithm 4 TriRapideRec(t,g,d)****Condition :**  $0 \leq g < d \leq \text{len}(t) - 1$ 

```

1: if g < d then
2:   m ← 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 :**  $\Omega(n \log_2(n)), \mathcal{O}(n^2)$ avec  $n = \text{len}(t)$ 

```

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
12         j -= 1
13         while t[j] > pivot:
14             j -= 1
15         i += 1
16         while t[i] < pivot:
17             i += 1
18
19     if i < j:
20         echange(t, i, j)
21     else:
22         return j
23
24 def triRapideRec(t, g, d):
25     if g < d:
26         m = partitionner(t, g, d)
27         triRapideRec(t, g, m)
28         triRapideRec(t, m + 1, d)
29
30 def triRapide(t):
31     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( $\mathbf{t}$ )
 

---

**Condition :**  $\forall i \in [0, \text{len}(\mathbf{t}) - 1], t[i] \in [0, 2]$

```

1:  $i \leftarrow 0$ 
2:  $j \leftarrow i$ 
3:  $k \leftarrow \text{len}(\mathbf{t}) - 1$ 
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 :**  $\forall l \in [0, i], t[l] = 0$

$\forall l \in [i, j], t[l] = 1$

$\forall l \in [k, \text{len}(\mathbf{t}) - 1], t[l] = 2$

**Complexité :**  $\Theta(n)$ , avec  $n = \text{len}(\mathbf{t})$

---

```

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

```

i. (1½ points) Soit  $\mathbf{t}$  un tableau d'entiers compris entre 0 et 2. Soient  $g$  et  $d$  deux indices tels que  $0 \leq g < d \leq (\text{len}(\mathbf{t}) - 1)$ . Écrire un algorithme ou bien un programme python `drapeauIntervalle( $\mathbf{t}, g, d$ )` qui trie les variables  $\mathbf{t}[i]$  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`.

ii. (1½ points) Soit  $\mathbf{t}$  un tableau d'entiers quelconques. Soient  $g$  et  $d$  deux indices tels que  $0 \leq g < d \leq (\text{len}(\mathbf{t}) - 1)$ . Écrire un algorithme ou bien un programme python `drapeauPartition( $\mathbf{t}, g, d$ )` 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  $\forall l \in [k, \text{len}(\mathbf{t}) - 1], t[l] > pivot$  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`.



**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` ?

– Nombre d'affectations :

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

– Meilleur des cas :

– Pire des cas :

(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` ?

– Meilleur des cas :

– Pire des cas :

(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`) ?

– Nombre minimum d'affectations :

– Nombre maximum d'affectations :

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

– Meilleur des cas :

– Pire des cas :

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