

## SUJET + CORRIGE

### Avertissement

- La plupart des questions sont indépendantes.
- **À chaque question, vous pouvez au choix répondre par un algorithme ou bien par un programme python.**
- 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	Bonus Points	Score
Mise en bouche	7	0	
Algorithmes de rang	13	1	
Total:	20	1	

### Exercice 1 : Mise en bouche

(7 points)

(a) (1 point) Deux nombres sont opposés si leur somme est égale à 0. Deux nombres sont inverses si leur produit est égal à 1. Écrire un algorithme `sontInvOuOpp(a,b)` où `a` et `b` sont deux nombres, qui retourne `Vrai` si `a` et `b` sont inverses ou opposés, `Faux` sinon.

**Solution:** Deux solutions parmi d'autres.

```
def sontInvOuOpp(a ,b):
    return a+b==0 or a*b==1
```

**Algorithme 1:** `SontInvOuOpp(a,b)`

**Données** : Deux nombres `a` et `b`

**retourner** (`a+b=0`) OU (`a*b=1`);

(b) (2 points) Écrire un algorithme `existeInvOuOppConsecutifs(T)` où `T` est un tableau de nombres, qui retourne `Vrai` si `T` contient deux nombres **consécutifs** opposés ou inverses, `Faux` sinon.

**Solution:** Deux solutions parmi d'autres.

```
def existeInvOuOppConsecutifs(T):
    for i in range(len(T)-1):
        if sontInvOuOpp(T[ i ] ,T[ i +1 ]):
            return True
    return False
```

**Algorithme 2:** `ExisteInvOuOppConsecutifs(T)`

**Données** : Un tableau `T` de nombres

```
pour i=0 à len(T)-2 faire
    si sontInvOuOpp(T[i],T[i+1]) alors
        retourner True;
    retourner False;
```

(c) (2 points) Écrire un algorithme `existeInvOuOpp(T)` où  $T$  est un tableau de nombres, qui retourne **Vrai** si  $T$  contient deux nombres, **ayant des indices différents**, opposés ou inverses, **Faux** sinon.

**Solution:** Deux solutions parmi d'autres.

```
def existeInvOuOpp(T):
    for i in range(len(T) - 1):
        for j in range(i + 1, len(T)):
            if sontInvOuOpp(T[i], T[j]):
                return True
    return False
```

---

**Algorithme 3: ExisteInvOuOpp( $T$ )**

---

**Données :** Un tableau  $T$  de nombres

**pour**  $i=0$  à  $\text{len}(T)-2$  **faire**

**pour**  $j=i+1$  à  $\text{len}(T)-1$  **faire**

**si** `sontInvOuOpp( $T[i], T[j]$ )` **alors**

**retourner** `True`;

**retourner** `False`;

---

(d) (2 points) Écrire un algorithme `nbInvOuOpp(T)` où  $T$  est un tableau de nombres, qui retourne le nombre de paires d'indices  $(i, j)$  telles que : d'une part  $i < j$ ; d'autre part  $T[i]$  et  $T[j]$  soient des nombres opposés ou inverses.

**Solution:** Deux solutions parmi d'autres.

```
def nbInvOuOpp(T):
    nb = 0
    for i in range(len(T) - 1):
        for j in range(i + 1, len(T)):
            if sontInvOuOpp(T[i], T[j]):
                nb = nb + 1
    return nb
```

---

**Algorithme 4: NbInvOuOpp( $T$ )**

---

**Données :** Un tableau  $T$  de nombres

`nb`  $\leftarrow 0$ ;

**pour**  $i=0$  à  $\text{len}(T)-2$  **faire**

**pour**  $j=i+1$  à  $\text{len}(T)-1$  **faire**

**si** `sontInvOuOpp( $T[i], T[j]$ )` **alors**

`nb`  $\leftarrow$  `nb` + 1;

**retourner** `nb`;

---

**Exercice 2 : Algorithmes de rang**

**(13 points)**

Le problème de la sélection consiste à trouver dans un tableau de nombres l'élément dit de rang  $i$ .

Pour cet exercice, du fait que les indices d'un tableau  $T$  sont compris entre 0 et `longueur(T)-1`, nous admettrons que l'élément de rang 0 est le plus petit élément du tableau, et que l'élément de rang `longueur(T)-1` est le plus grand.

**Exemple :** Soit  $T = [8, 6, 53, 8, 2, 9, 3, 10]$ , alors :

- Les éléments de rang  $< 0$  sont indéfinis.
- L'élément de rang 0 est 2.
- L'élément de rang 1 est 3.
- L'élément de rang 2 est 6.
- L'élément de rang 3 est 8.
- L'élément de rang 4 est 8.
- L'élément de rang 5 est 9.
- L'élément de rang 6 est 10.
- L'élément de rang 7 est 53.
- Les éléments de rang  $> 7$  sont indéfinis.

**Remarque 1 :** Une solution simple au problème de la sélection consiste à utiliser un algorithme quelconque de tri, puis de retourner l'élément de rang souhaité.

---

#### Algorithme 5: Rang(T,rang)

---

**Données :** Un tableau T de nombres, et rang un entier

**Résultat :** Si rang est un indice, alors T[rang] après avoir trié T

```
si rang<0 OU rang≥longueur(T) alors
  retourner nil;
  Trier(T);
  retourner T[rang];
```

---

**Remarque 2 :** Il est facile de se persuader qu'il n'est pas utile de trier *tout* le tableau pour avoir une solution au problème de la sélection. Dans cet exercice, nous allons adapter des algorithmes de tri vus en cours afin d'obtenir des algorithmes de rang plus *efficaces* que le précédent.

Dans toute la suite de l'exercice, vous pourrez utiliser la fonction classique **Echange(T,i,j)** qui échange les valeurs du tableau T indiquées par i et j.

---

#### Algorithme 6: Echange(T,i,j)

---

**Données :** Un tableau T de nombres, et deux indices i et j

**Résultat :** T[i] et T[j] échangés  
 $aux \leftarrow T[i];$   
 $T[i] \leftarrow T[j];$   
 $T[j] \leftarrow aux;$

---

```
def echange(T, i, j):
    TMP = T[i]
    T[i] = T[j]
    T[j] = TMP
```

(a) Solution adaptée du tri par sélection vu en cours.

---

#### Algorithme 7: TriSelection(T)

---

**Données :** Un tableau T de nombres

**Résultat :** Le tableau T trié en ordre croissant

```
pour i=0 à longueur(T)-1 faire
  iMin ← i;
  pour j=i+1 à longueur(T)-1 faire
    si T[j] < T[iMin] alors
      iMin ← j;
  si i ≠ iMin alors
    Echange(T,i,iMin);
```

---

```
def triSelection(T):
    for i in range(len(T)):
        iMin = i
        for j in range(i+1, len(T)):
            if T[j] < T[iMin]:
                iMin = j
        if iMin != i:
            echange(T, i, iMin)
```

Il semble évident qu'une fois la valeur désirée *bien placée* dans le tableau, il est inutile de continuer le tri.

i. (2 points) Écrire un algorithme `rangSelection(T, r)` fortement inspiré de l'algorithme ou du programme python `triSelection(T)` qui résout le problème de la sélection. Ne pas oublier de s'assurer que le rang désiré correspond à un indice du tableau.

**Solution:** Deux solutions parmi d'autres.

```
def rangSelection (T, r):
    if r<0 or r>=len (T):
        return None
    for i in range (r+1):
        iMin = i
        for j in range (i+1, len (T)):
            if T[j]<T[iMin]:
                iMin = j
        if iMin!=i:
            echange (T, i, iMin)
    return T[r]
```

---

**Algorithme 8:** RangSelection(T,r)

---

**Données :** Un tableau T de nombres, et un indice r

**Résultat :** L'élément de rang r du tableau T

si  $r < 0$  OU  $r \geq \text{longueur}(T)$  alors

    retourner nil;

pour  $i=0$  à  $r$  faire

$iMin \leftarrow i$ ;

    pour  $j=i+1$  à  $\text{longueur}(T)-1$  faire

        si  $T[j] < T[iMin]$  alors

$iMin \leftarrow j$ ;

        si  $i \neq iMin$  alors

            Echange(T,i,iMin);

retourner T[r];

---

ii. (1 point) Compléter le tableau des complexités en fonction de  $n=\text{longueur}(T)$  et du rang  $r$ .

**Rappel :** Les complexités ne dépendent pas de valeurs particulières des paramètres  $n$  et  $r$ , mais de valeurs particulières contenues dans le tableau.

**Solution:**

	TriSelection(T)	RangSelection(T,r)
Temps (meilleur des cas)	$\Omega(n^2)$	$\Omega(n \times r)$
Temps (pire des cas)	$\mathcal{O}(n^2)$	$\mathcal{O}(n \times r)$
Espace (meilleur des cas)	$\Omega(1)$	$\Omega(1)$
Espace (pire des cas)	$\mathcal{O}(1)$	$\mathcal{O}(1)$

**Non demandé :** Il est facile d'améliorer (un peu) la solution en sélectionnant les valeurs minimales (comme ici) lorsque  $r < n/2$ , et en sélectionnant les valeurs maximales lorsque  $r \geq n/2$ . Les complexités s'expriment alors en remplaçant  $r$  par  $\min(r, n - r)$ .

(b) Solution adaptée du tri à bulle vu en cours.

```
def triBulle (T):
    for i in range (len (T)-1, 0, -1):
        for j in range (i):
            if T[j]>T[j+1]:
                echange (T, j, j+1)
```

---

**Algorithme 9:** TriBulle(T)

---

**Données :** Un tableau T de nombres

**Résultat :** Le tableau T trié en ordre croissant

pour  $i=\text{len}(T)-1$  à 1 décroissant faire
   
    pour  $j=0$  à  $i-1$  faire
   
        si  $T[j] > T[j+1]$  alors
   
            Echange(T,j,j+1);

Il semble évident qu'une fois la valeur désirée *bien placée* dans le tableau, il est inutile de continuer le tri.

i. (2 points) Écrire un algorithme `rangBulle(T,r)` fortement inspiré de l'algorithme ou du programme python `triBulle(T)` qui résout le problème de la sélection. Ne pas oublier de s'assurer que le rang désiré correspond à un indice du tableau.

**Solution:** Deux solutions parmi d'autres.

```
def rangBulle(T, r):
    if r<0 or r>=len(T):
        return None
    for i in range(len(T)-1, r-1, -1):
        for j in range(i):
            if T[j]>T[j+1]:
                echange(T, j, j+1)
    return T[r]
```

---

**Algorithme 10: RangBulle(T,r)**


---

**Données :** Un tableau T de nombres et un indice r

**Résultat :** L'élément de rang r du tableau T

si  $r < 0$  OU  $r \geq \text{longueur}(T)$  alors

    retourner nil;

    pour  $i = \text{len}(T)-1$  à r, décroissant faire

        pour  $j = 0$  à  $i-1$  faire

            si  $T[j] > T[j+1]$  alors

                Echange(T, j, j+1);

    retourner T[r];

---

ii. (1 point) Compléter le tableau des complexités en fonction de  $n = \text{longueur}(T)$  et du rang  $r$ .

**Solution:**

	TriBulle(T)	RangBulle(T,r)
Temps (meilleur des cas)	$\Omega(n^2)$	$\Omega(n \times (n - r))$
Temps (pire des cas)	$\mathcal{O}(n^2)$	$\mathcal{O}(n \times (n - r))$
Espace (meilleur des cas)	$\Omega(1)$	$\Omega(1)$
Espace (pire des cas)	$\mathcal{O}(1)$	$\mathcal{O}(1)$

**Non demandé :** Il est facile d'améliorer (un peu) la solution en faisant monter les grosses bulles (comme ici) lorsque  $r \geq n/2$ , et en faisant descendre les petites bulles lorsque  $r < n/2$ . Les complexités s'expriment alors en remplaçant  $n - r$  par  $\min(r, n - r)$ .

(c) Solution adaptée du tri rapide vu en cours.

Soit la variante suivante de l'algorithme de partition basée sur l'algorithme du drapeau Hollandais vu en cours.

Cet algorithme *partitionne* le tableau en trois zones : la première contient des valeurs strictement inférieures à la valeur du pivot ; la seconde contient des valeurs égales à la valeur du pivot ; et la troisième des valeurs strictement supérieures à la valeur du pivot.

```

def troisPartitionner(T,g,d):
    pivot = T[g]
    i = g
    j = i
    k = d
    while j <= k:
        if T[j] == pivot:
            j += 1
        elif T[j] < pivot:
            echange(T,i,j)
            i += 1
            j += 1
        else:
            echange(T,j,k)
            k -= 1
    return i,j,k

```

**Algorithme 11:** TroisPartitionner(T,g,d)

**Données :** Un tableau T de nombres, et deux indices g et d

**Résultat :** i,j,k tel que  $T[g..i-1] < T[i..k] = \text{pivot} < T[k+1..d]$

```

pivot ← T[g];
i ← g;
j ← i;
k ← d;
tant que  $j \leq k$  faire
    si  $T[j] = \text{pivot}$  alors
         $j \leftarrow j+1;$ 
    sinon si  $T[j] < \text{pivot}$  alors
        Echanger(T,i,j);
         $j \leftarrow j+1;$ 
         $i \leftarrow i+1;$ 
    sinon
        Echanger(T,j,k);
         $k \leftarrow k-1;$ 
retourner i,j,k;

```

**Rappel :** La complexité, vue en cours, de `troisPartitionner(T,g,d)` est  $\Theta(d - g + 1)$ .

i. (1 point) Compléter le tableau suivant afin de simuler l'exécution de `troisPartitionner`.

$$T = [17, 3, 21, 13, 17, 25, 4], g = 0, d = 6$$

**Solution:**

	Temps →						
pivot	17						
couple d'indices échangés			(0,1)	(2,6)	(1,2)	(2,3)	(5,5)
i	0		1		2	3	
j	0	1	2		3	4	5
k	6			5			4

ii. (2 points) Cette version *améliorée* du tri rapide tire profit des trois zones, en ne faisant pas d'appel récursif sur la zone intermédiaire, car les valeurs de cette zone sont correctement placées.

```

def triRapideRec(T, g, d):
    if g < d:
        i, j, k = troisPartitionner(T, g, d)
        triRapideRec(T, g, i-1)
        triRapideRec(T, k+1, d)

def triRapide(T):
    triRapideRec(T, 0, len(T)-1)

```

---

**Algorithme 12:** TriRapide-Rec(T,g,d)

**Données :** Un tableau T de nombres, et deux indices g et d

**Résultat :** Le tableau T[g..d] trié en ordre croissant

**si**  $g < d$  **alors**

(i,j,k)  $\leftarrow$  troisPartitionner(T,g,d);  
TriRapideRec(T,g,i-1);  
TriRapideRec(T,k+1,d);

---

**Algorithme 13:** TriRapide(T)

**Données :** Un tableau T de nombres

**Résultat :** Le tableau T trié en ordre croissant

TriRapideRec(T,0,longueur(T)-1);

---

Écrire des algorithmes `rangRapide(T,r)` et `rangRapideRec(T,g,d,r)` fortement inspirés des algorithmes `triRapide(T)` et `triRapideRec(T,g,d)`, qui résolvent le problème de la sélection. Ne pas oublier de s'assurer que le rang désiré correspond à un indice du tableau.

**Solution:** Deux solutions parmi d'autres.

```

def rangRapideRec(T, g, d, r):
    if g < d:
        i, j, k = troisPartitionner(T, g, d)
        if r < i:
            rangRapideRec(T, g, i-1, r)
        elif r > k:
            rangRapideRec(T, k+1, d, r)

def rangRapide(T, r):
    if r < 0 or r >= len(T):
        return None
    rangRapideRec(T, 0, len(T)-1, r)
    return T[r]

```

---

**Algorithme 14:** RangRapideRec(T,g,d,r)

**Données :** Un tableau T de nombres, trois indices g, d et r

**Résultat :** Positionne l'élément de rang r du tableau T

**si**  $g < d$  **alors**

(i,j,k)  $\leftarrow$  troisPartitionner(T,g,d);  
**si**  $r < i$  **alors**  
    RangRapideRec(T,g,i-1,r);  
**sinon si**  $r > k$  **alors**  
    RangRapideRec(T,k+1,d,r);

---

**Algorithme 15:** RangRapide(T,r)

**Données :** Un tableau T de nombres, et un indice r

**Résultat :** L'élément de rang r du tableau T

**si**  $r < 0$  **OU**  $r \geq \text{longueur}(T)$  **alors**  
    **retourner** nil;  
    RangRapideRec(T,0,longueur(T)-1,r);  
    **retourner** T[r];

---

iii. (1 point) Compléter le tableau des complexités en fonction de `n=longueur(T)` et du rang `r`.

**Solution:**

	TriRapide(T)	RangRapide(T,r)
Temps (meilleur des cas) (Toutes les valeurs identiques)	$\Omega(n)$	$\Omega(n)$
Temps (pire des cas) (tableau trié)	$\mathcal{O}(n^2)$	$\mathcal{O}(n \times r)$
Espace (meilleur des cas)	$\Omega(1)$	$\Omega(1)$
Espace (pire des cas)	$\mathcal{O}(n)$	$\mathcal{O}(r)$

(d) La solution *naturelle* au problème de sélection basé sur le tri rapide est une solution récursive *terminale*.

i. (2 points) Écrire un algorithme `rangRapideIteratif(T,r)` obtenu *automatiquement* à partir de votre solution à la question précédente (qui doit donc être récursive terminale).

**Solution:** Deux solutions parmi d'autres.

```
def rangRapideIteratif(T, r):
    if r < 0 or r >= len(T):
        return None
    g = 0
    d = len(T) - 1
    while True:
        i, j, k = troisPartitionner(T, g, d)
        if r < i:
            d = i - 1
        elif r > k:
            g = k + 1
        else:
            return T[r]
```

**Algorithme 16:** RangRapideIteratif(T,r)

**Données :** Un tableau T de nombres, et un indice r

**Résultat :** L'élément de rang r du tableau T

si  $r < 0$  OU  $r \geq \text{longueur}(T)$  alors

  | retourner nil;

g  $\leftarrow 0$ ;

d  $\leftarrow \text{longueur}(T) - 1$ ;

tant que True faire

  | (i,j,k)  $\leftarrow$  troisPartitionner(T,g,d);

  | si  $r < i$  alors

    | d  $\leftarrow i - 1$ ;

  | sinon si  $r > k$  alors

    | g  $\leftarrow k + 1$ ;

  | sinon

    | retourner T[r];

ii. (1 point) Compléter le tableau des complexités en fonction de `n=longueur(T)` et du rang `r`.

**Solution:**

	TriRapide(T)	RangRapideIteratif(T,r)
Temps (meilleur des cas) (Toutes les valeurs identiques)	$\Omega(n)$	$\Omega(n)$
Temps (pire des cas) (tableau trié)	$\mathcal{O}(n^2)$	$\mathcal{O}(n \times r)$
Espace (meilleur des cas)	$\Omega(1)$	$\Omega(1)$
Espace (pire des cas)	$\mathcal{O}(n)$	$\mathcal{O}(1)$

**Non demandé :** En réalité le pire des cas est soit un tableau trié, ce qui donne une complexité en  $\mathcal{O}(n \times r)$ , soit un tableau trié en ordre décroissant qui donne une complexité en  $\mathcal{O}(n \times (n - r))$ . Ainsi la complexité dans le pire des cas est en  $\mathcal{O}(n \times \max(r, n - r))$ .

(e) (1 point (bonus)) En pratique, le cas *standard* correspond à un tableau initial non trié, et ayant peu de valeurs répétées. L'algorithme de partitionnement retourne alors souvent trois zones telles

que l'intermédiaire est *petite* et à peu près au centre du tableau.

Compléter le tableau en fonction de  $n=\text{longueur}(T)$  et du rang  $r$  pour ce cas *moyen*.

**Solution:**

	TriRapide(T)	RangRapide(T,r)	RangRapideIteratif(T,r)
Temps moyen (zone 2 petite et centrée)	$(n \times \log_2(n))$	$(n)$	$(n)$