Questions écrites (50 points)

Question #1

a) **ALGORITHME** InverserMoitierGauche(tableau)

```
TailleTableau ← longeur(tableau)                    // Ligne 1
      Si  tailleTableau mod 2 = 0                   // Ligne 2
            limiteTableau ← tailleTableau/2         // Ligne 3
      Sinon
            limiteTableau ← (tailleTableau/2) +1    // Ligne 4
      Fin Si
      i ← 0                                         // Ligne 5
      Tant que i < (limiteTableau/2)                // Ligne 6
            indiceElementEchanger ← limiteTableau – i – 1 // Ligne 7
            temp ← tableau[i]                       // Ligne 8
            tableau[i]  ← tableau[indiceElementEchanger] // Ligne 9
            tableau[indiceElementEchanger] ← temp   // Ligne 10
            i ← i+1                                 // Ligne 11
      Fin Tant que
FIN
```

b. Dans la fonction suivante, InverserMoitierGauche, nous aurons une complexité temporelle de O(n) en termes de Big-O.
Cela est obtenu en effectuant une analyse de la fonction et en examinant chaque opération élémentaire. Si nous penons la ligne 1 à 4 (ligne numérotée à la question '1.a)'), se sont toutes des opérations s'effectuant en temps constant soit O(1).
En ce qui concerne la ligne 5, nous faisons une affectation simple donc, O (1) aussi. Dans le cas de la ligne 6 à 11, la boucle, exécute des itérations en fonction de la valeur de *limiteTableau / 2* et dans le pire cas elle s'approche de n/2 +1 donc, la boucle va itérer n/4 fois. Étant donné que le reste des opérations à l'intérieur de la boucle sont de O (1) et que la boucle s'effectue en n/4 pire temps, alors la ligne 6 à 11 est de O(n). La partie de l'algorithme la plus couteuse est donc, la boucle qui parcours les éléments du tableau. **L'algorithme 'InverserMoitierGauche' sera donc, de complexité temporelle de l'ordre de O(n).**

c. Dans le cas de l'algorithme précédent et de la complexité spatiale, on se concentre sur l'espace mémoire supplémentaire qu'il est requis au-delà de l'entrée elle-même. Donc, la mémoire consommée autre que la mémoire d'entrée comme le nombre d'éléments n utilisés. Notre algorithme utilise un nombre fixe de variables telles que *tailleTableau, limiteTableau, i, indiceElementEchanger et temp*. Chacune d'entre elles va occuper un espace constant. De plus, nous n'utilisons aucune structure de données supplémentaire tels qu'un tableau afin de faire la copie ou l'utilisation d'une pile, afin d'effectuer nos opérations. Seulement le tableau passé en paramètre sera utilisé et manipulé. Prenant en considération que la complexité spatiale de l'algorithme est dominée par

l'utilisation d'un nombre fixe de variables supplémentaires, qui ne dépendent pas de la taille du tableau d'entrée. Cela veut dire que **la complexité spatiale sera de temps constant donc, Big-O de O (1).** Donc, quel que soit la taille d'entrée, l'algorithme nécessite une quantité fixe d'espace mémoire.

Question #2

F(n)	**Big-O**
$n^5 \log n$ est $O(n^7)$	**Vrai:** Par définition de Big-O, il faut trouver une constante, réelle, c > 0 et une constante entière $n_0 \geq 1$ telle que $n^5 \log(n) \leq c * n^7$ pour $n \geq 1$ Il est possible de simplifier cette équation en divisant chaque côté de l'inégalité par n^5, on obtient, $\log(n) \leq c * n^2$ pour $n \geq 1$ Si nous prenons c = 1 et $n_0 = 1$, nous satisferons la clause ci-dessus et de même pour tout c > 0 *et* $n_0 \geq 1$, nous avons le terme log(n) qui croit beaucoup moins rapidement que le terme $c * n^2$. Donc, si cette clause est prouvée il va de soit pour $n^5 \log(n) \leq c * n^7$ et par le fait même cette fonction est O (n^7).
$1o^8n^2 + 5n^4 + 5000000n^3 + n$ est $\Theta(n^4)$	**Vrai:** Par définition, il faut prouver que f(n) = $10^8 * n^2 + 5n^4 + 5000000n^3 + n$ est de O (n^4) et Ω (n^4) à la fois pour être Θ (n^4). Par définition de Big-O, il faut trouver une constante, réelle, c > 0 et une constante entière $n_0 \geq 1$ telle que $10^8 * n^2 + 5n^4 + 5000000n^3 + n \leq c * n^4$ pour $n \geq 1$ Si nous prenons c=6 et $n_0 = 1$, nous satisferons la clause ci-dessus que pour tout $n \geq n_0$, f(n) sera inférieur ou égale à c * n^4. Donc, il est possible de dire que f(n) est O (n^4). Maintenant il ne reste plus qu'a prouver que f(n) est aussi Ω (n^4). Par définition, il faut prouver que f(n) est de et Ω (n^4) et donc, il faut trouver une constante, réelle, c > 0 et une constante entière $n_0 \geq 1$ telle que $10^8 * n^2 + 5n^4 + 5000000n^3 + n \geq c * n^4$ Étant donné que le terme $5n^4$ est le terme dominant dans f(n), la fonction ne sera jamais inférieure à $5n^4$ pour tout $n \geq 1$. Nous pouvons donc, choisir c=5, puisque $5n^4$ est déjà une partie de f(n), on s'assure ainsi que $f(n) \geq c * n^4$ pour tout $n \geq 1$. Donc, f(n) est Big-Omega de Ω (n^4). Par conséquent, f(n) satisfait à la fois O (n^4) et Ω (n^4), nous concluons que f(n) est Big-thêta Θ (n^4).

n^n est O(n!)	**Faux:** Par définition, de Big-O, il faut trouver une constante, réelle, c > 0 et une constante entière $n_0 \geq 1$ telle que $n^n \leq c * n!$ pour $n \geq 1$ Dans note cas, n^n crois beaucoup plus rapidement que $n!$ Donc, il ne sera jamais possible de trouver un c suffisamment grand afin de respecter la clause. Par conséquent, pour tout $n \geq 1$ n^n va être strictement plus grand que $n!$ donc, notre fonction n'est pas O (n!).
$0.01n^2 + 0.0000001n^4$ est $\Theta(n^2)$	**Faux :** Pour des grandes valeurs de n, le terme $0.0000001n^4$ croîtra plus vite que $0.01n^2$, rendant $0.01n^2+0.0000001n^4$ asymptotiquement équivalent à n^4.
$1000000n^2 + 0.0000001n^5$ est $\Omega(n^3)$	**Faux :** Ici, $0.0000001n^5$ deviendra le terme dominant pour de très grandes valeurs de n, ce qui signifie que l'expression est $\Omega(n^5)$,
n! est $\Omega(2^n)$	**Vrai :** n! croît effectivement plus vite que 2^n pour de grandes valeurs de n, donc n! est bien $\Omega(2n)$

Question #3

a. Pour la complexité temporelle Big-O (Pire cas) : ce cas se produit lorsque le tri du tableau nécessite le nombre maximal de permutations pour chaque élément. C'est à dire, qu'à chaque appel récursif, l'algorithme effectue potentiellement des permutations pour chaque élément du tableau. Ainsi la complexité **Big-O en pire cas serait $O(n^2)$**.
Pour la complexité temporelle Big-Omega (meilleur cas) : le meilleur cas se produit si le tableau A est déjà trié en entrée. Dans ce cas, le tableau A est parcouru une seule fois dans chaque boucle *while* (avant et arrière) sans effectuer une seule permutation. Les deux boucles vérifient simplement que chaque paire d'éléments adjacents n'ont besoin d'aucune permutation. Et ainsi, la variable '*done*' reste à '*true*' et aucun appel récursif de l'algorithme n'est effectué. Dans ce cas, **Big-Omega en meilleur cas serait Ω(n)**

b. Trace de MonAlgorithme pour un tableau A = (6, 12, 7, 3, 5) :

À l'initialisation :

A	**n**	**j**	**done**
(6,12, 7, 3, 5)	**5**	**0**	**true**

	j	**A[j-1]**	**A[j]**	**A[j+1]**	**A après permutation**	**done**
Boucle while (1)						
Iter 1	0	-	6	12	*Pas de permutation*	true
Iter 2	1	-	12	7	A = (6,7,12, 3, 5)	false
Iter 3	2	-	12	3	A = (6,7,3, 12, 5)	false
Iter 4	3	-	12	5	A = (6,7,3,5,12)	false
Boucle while (2)						
Iter 1	4	5	12	-	Pas de permutation	false
Iter 2	3	3	5	-	Pas de permutation	false

Iter 3	2	7	3	-	A = (6,3,7,5,12)	false
Iter 4	1	6	3	-	A = (3,6,7,5,12)	false

***done* = *'false'* = > 1er appel récursif de MonAlgorithme(A,n) :**

A	**n**	**j**	**done**
(3,6,7,5,12)	**5**	**0**	**true**

	j	**A[j-1]**	**A[j]**	**A[j+1]**	**A après permutation**	**done**
Boucle while (1)						
Iter 1	0	-	3	6	*Pas de permutation*	true
Iter 2	1	-	6	7	*Pas de permutation*	true
Iter 3	2	-	7	5	A = (3,6,5,7,12)	false
Iter 3	3	-	7	12	*Pas de permutation*	false
Boucle while (2)						
Iter 1	4	7	12	-	Pas de permutation	false
Iter 2	3	5	7	-	Pas de permutation	false
Iter 3	2	6	5	-	A = (3,5,6,7,12)	false
Iter 4	1	3	5	-	*Pas de permutation*	false

Résultat final : A = (3,5,6,7,12)

c. 'MonAlgorithme' est un algorithme de tri croissant par comparaison. La première étape consiste à déplacer les éléments les plus grands du tableau vers la fin de celui-ci (comparaison de gauche vers la droite). Ensuite, il compare les éléments adjacents dans la direction opposée, et effectue les permutations nécessaires pour déplacer les plus petits éléments vers le début du tableau. Enfin, si au moins une permutation a eu lieu lors de ces deux premières étapes, l'algorithme s'appelle lui-même récursivement pour effectuer un nouveau "cycle" de comparaisons et de permutations. Ce processus se répète jusqu'à ce que le tableau en entrée soit entièrement trié.

d. Afin d'améliorer le temps d'exécution, envisager d'autres algorithmes de tri comme le tri rapide, le tri fusion, ou le tri par tas permettra d'offrir de meilleures performances en moyenne et dans le pire des cas (Big-O), notamment pour de grands ensembles de données.
De façon générale, une bonne optimisation serait la division du tableau en deux à chaque appel. Cela s'inspire des stratégies "Diviser pour régner", typique des algorithmes de tri ci-haut. Cette méthode implique de diviser le tableau en sous-tableaux plus petits, de les trier indépendamment, puis de fusionner les sous-tableaux triés en un seul tableau trié.

e. MonAlgorithme est un exemple de récursivité linéaire, et non terminale. Chaque appel récursif n'est pas la dernière opération effectuée ; il y a une vérification de condition (*if (!done)*) qui précède l'appel récursif, et le retour de l'appel récursif lui-même n'est pas immédiatement retourné.

Questions de programmation (50 points)

Les premiers nombres de la séquence CalcNbImpairs sont :
1, 1, 1, 3, 5, 9, 17, 31, 57, 105, 193, 355, 653, 1201, 2209, 4063, 7473, 13745, 25281, 46499, ...

Ces nombres commencent par trois valeurs prédéterminées (1,1,1) et chaque valeur suivante est la somme des trois précédentes. Nous pouvons donc poser la définition récursive suivante :

Cas de base :

- C1 = 1
- C2 = 1
- C3 = 1

Cas récursif : $C_N = C_{N-1} + C_{N-2} + C_{N-3}$ pour $N > 3$

Dans cette partie, nous allons concevoir un algorithme récursif qui définit le calculateur CalcNbImpairs.

a) Lorsqu'on appelle *CalcNbImpairs*, si N est strictement inférieur à 4, il y a un seul appel (le cas de base lui-même), donc k=1. Si N est supérieur ou égal à 4, la fonction se rappelle elle-même trois fois, donc nous avons : **k(N) = k(N−1) + k(N−2) + k(N−3) + 1**

```
Algorithme CalcNbImpairsMultp (N) :

   Entrée : un entier N supérieur ou égal à 1
   Sortie : le N^ieme nombre du calculateur
   k ← 0    // compteur nombre d'appels

   k ← k + 1
   Si N < 4 alors
      Retourner 1
   Sinon
      Retourner CalcNbImpairsMultp(N-1) + CalcNbImpairsMultp(N-2) + CalcNbImpairsMultp(N-3)
```

Bien que simple, cette approche peut conduire à une complexité temporelle exponentielle en raison des appels récursifs multiples. Notamment pour des valeurs élevées de N. Examinons cela en fonction du nombre d'appels récursif k par *CalcNbImpairsMultp(N)* :

$k_1 = 1 \quad k_2 = 1 \quad k_3 = 1$
$k_4 = k_3 + k_2 + k_1 + 1 = 1 + 1 + 1 + 1 = 4$
$k_5 = k_4 + k_3 + k_2 + 1 = 4 + 1 + 1 + 1 = 7$
$k_6 = k_5 + k_4 + k_3 + 1 = 7 + 4 + 1 + 1 = 13$
$k_7 = k_6 + k_5 + k_4 + 1 = 13 + 7 + 4 + 1 = 25$
$k_8 = k_7 + k_6 + k_5 + 1 = 25 + 13 + 7 + 1 = 46$

On note alors que k_N triple au moins toutes les deux fois - avec une croissance exponentielle puisque le nombre global d'appels récursifs pour un N donné vérifie bien que $k_N > 2^{N/2}$
Cette explosion du nombre d'appels résulte en une complexité temporelle de l'**ordre de O (3^N)**.

Cependant, il est possible d'améliorer les performances de l'algorithme *CalcNbImpair* en utilisant une approche itérative ou une approche récursive linéaire.

L'algorithme *CalcNbImpairsLineaire(N)* utilisera la récursivité linéaire : la fonction contient au plus un appel récursif. Cette approche permettra de réduire le nombre d'appels par calcul, et par conséquent réduire la complexité à l'**ordre de O(N)**.
Ce deuxième algorithme résout plus efficacement les goulots d'étranglement du premier en éliminant la redondance des calculs grâce à une transmission des données de façon linaire. Cette version optimisée conduit à une réduction drastique du nombre total d'opérations. Par conséquent, l'algorithme utilisant la récursivité linaire sera beaucoup plus efficace pour traiter des grandes valeurs de N.

```
Algorithme CalcNbImpairsLineaire (N) :

   Entrée : un entier N supérieur ou égal à 1
   Sortie : le N^ieme nombre du calculateur

   Fonction calcNbImpairs(N)
      Si N < 4
         Retourner 1
      Fin Si
      Retourner calcNbImpairsRecursive(1, 1, 1, N, 4)
   Fin Fonction

   Fonction calcNbImpairsRecursive(premier, deuxieme, troisieme, N, compteur)
      K ← K + 1
      Si compteur == N
          Retourner premier + deuxieme + troisieme
      Fin Si
      Retourner
        calcNbImpairsRecursive(deuxieme, troisieme, premier + deuxieme + troisieme, N,
compteur + 1)
      Fin Fonction
```

- calcNbImpairs est la méthode qui démarre le processus de calcul pour un entier *N*. Si *N* est strictement inférieur à 4, la méthode retourne directement 1 – sinon elle appelle calcNbImpairsRecursive avec les trois premiers termes initiaux étant 1, et commence le calcul à partir du 4ème terme.
- calcNbImpairsRecursive est la méthode récursive qui effectue réellement le calcul. Elle prend les trois derniers termes calculés (premier, deuxieme, troisieme), le terme cible N, et le compteur actuel compteur qui est initialisé à 4 pour le premier appel. Lorsque le compteur atteint N, elle retourne la somme des trois derniers termes. Sinon, elle se rappelle elle-même avec les termes mis à jour et incrémente le compteur.

b)

Dans le cas de notre algorithme *CalcNbImpair,* aucune des deux versions implémentées en java n'utilise de récursivité terminale, a proprement parlé. Afin que la fonction soit considérée comme telle, la derniere instruction doit correspondre à l'appel récursif. Dans le cas de nos deux algorithmes, la dernière instruction correspond à la somme des termes : le calcul **premier + deuxième + troisième** est d'abord effectué avant l'appel récursif de la fonction, avec ses résultats.

Pour être considérée comme une fonction récursive, notre fonction devrais effectuer les calculs nécessaires en premier. Donc, il faudrait initier les résultats au préalable. Cependant, cela implique ll'utilisation temporaire d'un minimum de 2 nouvelles variables, ce qui à son tour requiert un certain temps à chaque répétition. Cela explique pourquoi nous observons des temps d'exécution similaires entre *CalcNbImpairLineaire* et *CalcNbImpairLineaireTerminal*.

```
Algorithme CalcNbImpairsLineaireTerminal (N) :

   Entrée : un entier N supérieur ou égal à 1
   Sortie : le N^ieme nombre du calculateur


   Fonction calcNbImpairs(N)
      Si N < 4
         Retourner 1
      Fin Si
      Retourner calcNbImpairsRecursive(1, 1, 1, N, 4)
   Fin Fonction


   Fonction calcNbImpairsRecursive(premier, deuxieme, troisieme, N, compteur)
      K ← K + 1
      Si compteur == N
          Retourner premier + deuxieme + troisieme
      Fin Si
      nouveauTroisieme ← premier + deuxieme + troisieme
      nouveauCompteur ← compteur + 1
      Retourner
        calcNbImpairsRecursive(deuxieme,troisieme,nouveauTroisieme,nouveauCompteur)
      Fin Fonction
```

En terne de complexité temporelle, l'algorithme CalcNbImpairLineaireTerminal est également **O(N)**.

La version Java sera présente dans le fichier Zip, aussi contenant les autres fichiers en lien avec projet. De plus, les fichiers .txt avec les résultats obtenus de chaque expérimentation seront contenus dans le fichier Zip. Il est également possible d'obtenir à nouveau les fichiers .txt en exécutant la fonction main de la classe CalculTempExecutionAlgorithme.