GRAVALLON Guillaume

LANUSSE Quentin

TP ALGO/PROG

Exercice : Élection dans une liste chaînée circulaire

a) **Proposer un type explicite pour représenter une liste simplement chaînée circulaire.**

Notre liste simplement chaînée est représentée par la classe « CircularList » qui utilise une classe interne « Cell ».

b) **Pour l'exercice, il n'est pas nécessaire que le type que vous avez choisi puisse représenter une liste vide, pourquoi ?**

Il n'est pas nécessaire de pouvoir représenter une liste vide car, s'il n'y a pas de candidats, il n'y a pas d'élection. Cette liste n'a donc pas d'intérêt sans valeurs. Néanmoins dans un soucis de qualité de code, notre liste vide est représentée comme une liste de valeur « null » et les cas sont gérés dans les différentes méthodes.

c) **Implémenter l'algorithme d'élection proposé. Il doit renvoyer le candidat élu et afficher à l'écran les candidats éliminés au fur et à mesure. (L'affichage pourra être commenté lorsque vous ferez des tests sur de grandes instances.)**

Cet algorithme a été implémenté dans la classe « CircularList » sous la forme de la méthode « elect() » qui prend comme paramètre k, et un booléen permettant d'avoir ou non l'affichage de chaque étape.

d) **Tester sur l'exemple donné ci-dessus**

Le test est effectué dans le main de notre programme, grâce à une instance de la classe « TestElection » sur laquelle nous appelons la méthode « testExemple() ». En voici l'affichage :

3 is eliminated...

6 is eliminated...

2 is eliminated...

7 is eliminated...

5 is eliminated...

1 is eliminated...

4 is elected !

e) **Faire un programme qui affiche le candidat élu pour k qui vaut 2, 3, 5 puis 10 et dans les cas où 10^3 10^4, 10^5, puis 10^6 candidats se présentent.**

Le test est effectué dans le main de notre programme, grâce à une instance de la classe « TestElection » sur laquelle nous appelons la méthode « testElection ()» avec les différentes valeurs demandées. Voici un tableau récapitulatif des résultats obtenus.

	10^3	**10^4**	**10^5**	**10^6**
2	977	3617	68929	951425
3	604	2692	92620	637798
5	763	646	40333	718997
10	63	9143	77328	630538

f) **Étudier le coût de façon exacte puis en utilisant une notation de Landau, en précisant s'il s'agit d'un pire cas, meilleur cas ou cas moyen. Expliquer le résultat. Indication : pour vous aider, introduisez dans votre algorithme une variable qui compte le nombre de tours de boucle et faites des tests systématiques, par exemple pour n = 100 et k variant de 1 à 200.**

En étudiant le coût de l'algorithme on le détermine à (n-1)(k-1), ce qui est confirmé par une variable comptant les tours de boucle. Cela s'explique naturellement : nous effectuons notre boucle jusqu'à n'avoir plus qu'un candidat, moment où aucune boucle n'est effectuée. Nous allons donc forcément faire n-1 fois le traitement. Celui-ci boucle k-1 fois pour effectuer le déplacement nécessaire dans la liste. Nous avons donc n-1 fois k-1 traitement, d'où le coût constaté.

En notation Landau le coût est donc de O(kn) (cout moyen).

Problème : comparaison de tris simples

Il s'agit d'implémenter puis de tester trois algorithmes de tris (non récursifs) sur des tableaux d'entiers afin de pouvoir comparer leurs performances.

Ces trois algorithmes ont été largement étudiés et il en existe de multiples implémentations sur internet.

Vous pouvez reprendre ce que bon vous semble à condition de citer vos sources proprement ! Il vous est demandé par contre de bien avoir compris le principe de chacun des algorithmes.

3.1 Tri par insertion dichotomique

Le premier algorithme est un tri par insertion, comme présenté en cours, mais améliorée de la façon suivante : à chaque fois qu'un élément doit être inséré dans la tranche des éléments triés, la recherche de la position où insérer l'élément se fait grâce à une recherche dichotomique.

3.2 Tri bulle mélangé (shaker sort)

Le deuxième algorithme est une variante du tri bulle. Le tri bulle pourrait s'implémenter par cette fonction :

```
Void triBulle(array(i n t) RW t) {
        f o r( i n t i = 0; i< t.length; i++)
                f o r( i n t j = t.length - 1; j > i; j--)
                        i f (t[j] < t[j - 1])
                                swap(t, j - 1, j);
}
```

Le tri bulle mélange ou *shaker sort* fonctionne comme le tri bulle en alternant les passes de gauche à droite et de droite à gauche.

3.3 Shell sort

Le troisième algorithme, le *shell sort* est une variante du tri par insertion qui permet des permutations entre des éléments éventuellement non adjacents du tableau. Voici comment fonctionne ce tri.

Soit *k* une constante entière positive plus petite que la taille du tableau. On considère le sous-ensemble des éléments à trier composé des éléments d'indices 0, *k*, *2k* ,3*k*... et on applique le principe du tri par insertion (sans dichotomie), en place, sur ces éléments. (Cela revient à faire + *k* pour aller à l'indice suivant, au lieu de + 1 dans le tri par insertion.)

On applique ensuite ce principe successivement sur les éléments d'indices 1, 1+k, 1+2k, 1+3k..., puis 2, 2+k, 2+2k, 2+3k..., etc. jusqu'à (k-1), (k-1)+k, (k-1)+2k, (k-1)+3k...

(Bien entendu, pour *k* valant 1, on retrouve l'algorithme de tri par insertion.)

L'algorithme de *shell sort* est l'application répétée de cette procédure en suivant une séquence de valeurs de *k* bien choisie.

3.2 Travail à faire

Question 2

a) **Implémenter l'algorithme correspondant.**

Les trois algorithmes sont implémentés dans la classe « SortClass ».

b) **Faire des tests fonctionnels simples.**

Les tests sont effectués dans le main du programme.

c) **Mettre en place un moyen de compter le nombre de comparaisons et le nombre d'affectations réalisée, et un moyen de mesurer le temps d'exécution.**
Au sein du code des trois algorithmes, nous avons intégré trois variables. Une mesure le temps d'exécution de la méthode, les deux autres sont des compteurs d'affectation et de comparaison.

Question 3

a) **Implémenter une fonction qui remplit aléatoirement et uniformément un tableau de taille fixée de façon à se rapprocher de l'hypothèse d'équirépartition utilisée pour les coûts en moyenne.**
Cette méthode est implémentée dans « SortClass » et s'appelle « fillList() ».

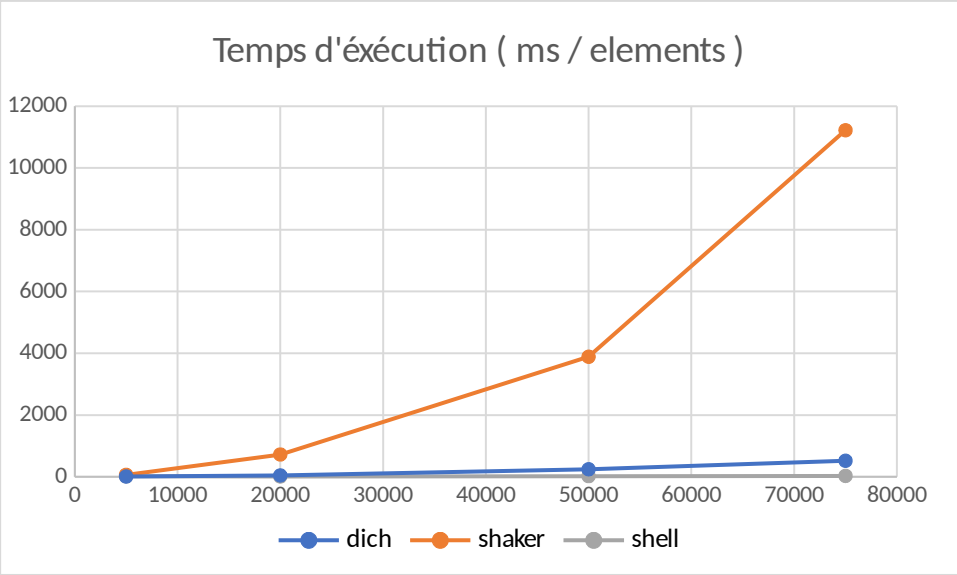
b) **Comparer les trois solutions sur des tailles croissantes d'instance, en termes de nombre d'affectations, nombre de comparaisons et temps d'exécution.**

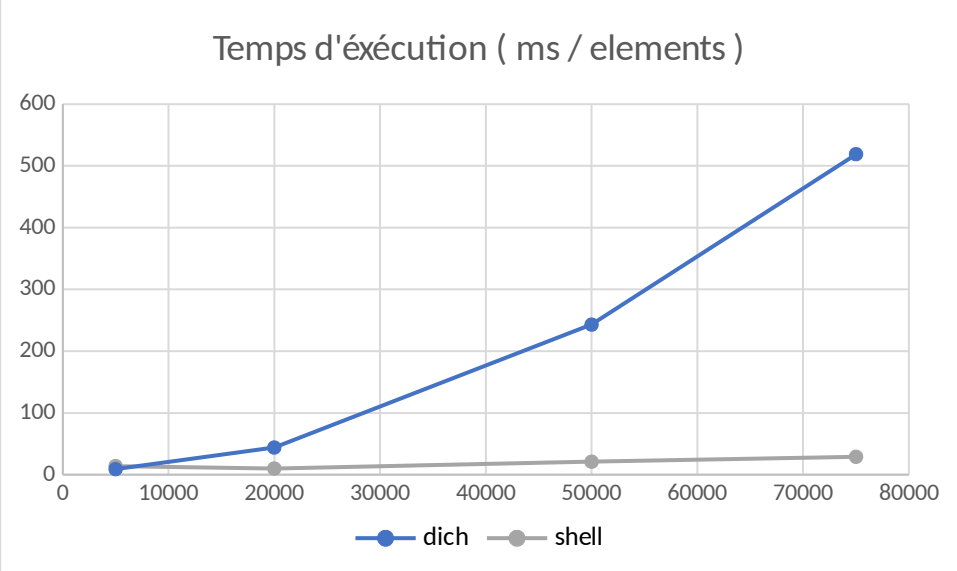
Voir graphes.

c) **Pour le *shell sort*, on pourra utiliser les séquences de valeurs de *k* suivantes : 1, 4, 13, 40, 121, 364, 1093, 3280, 9841…, ou 1, 8, 23, 77, 281, 1073, 4193, 16577…, ou encore 1, 2, 3, 4, 6, 9, 8, 12, 18, 27, 16, 24, 36, 54, 81, …**[1]

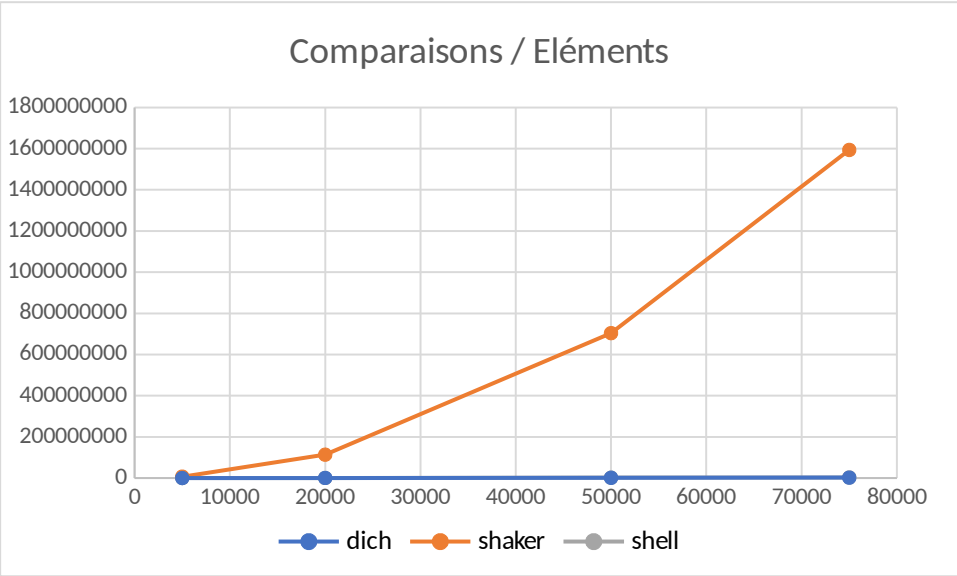
La séquence que nous avons choisie est la première proposée par Shell, qui est une division successive par 2.

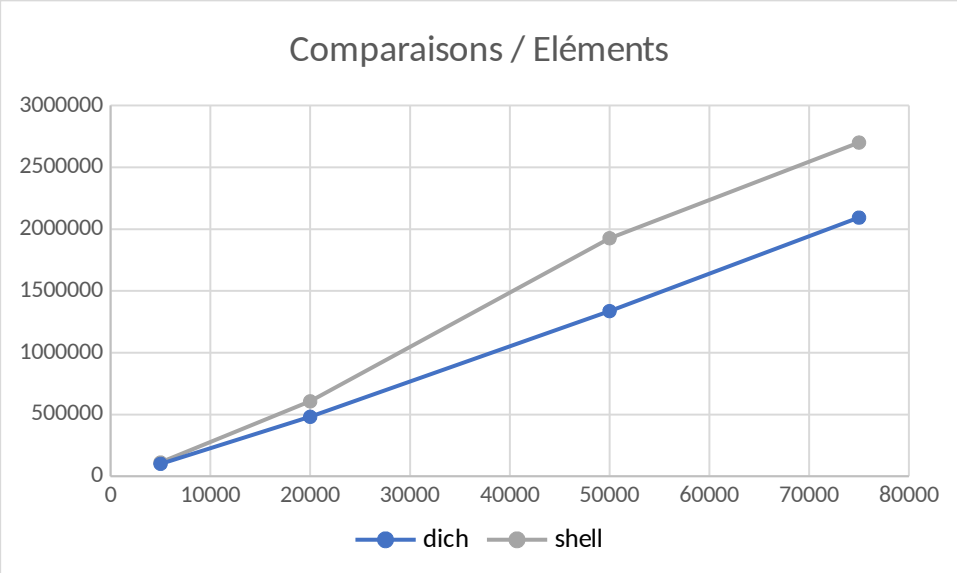
d) **Mettre les résultats sous forme graphique ! Et commenter brièvement les résultats remarquables.**



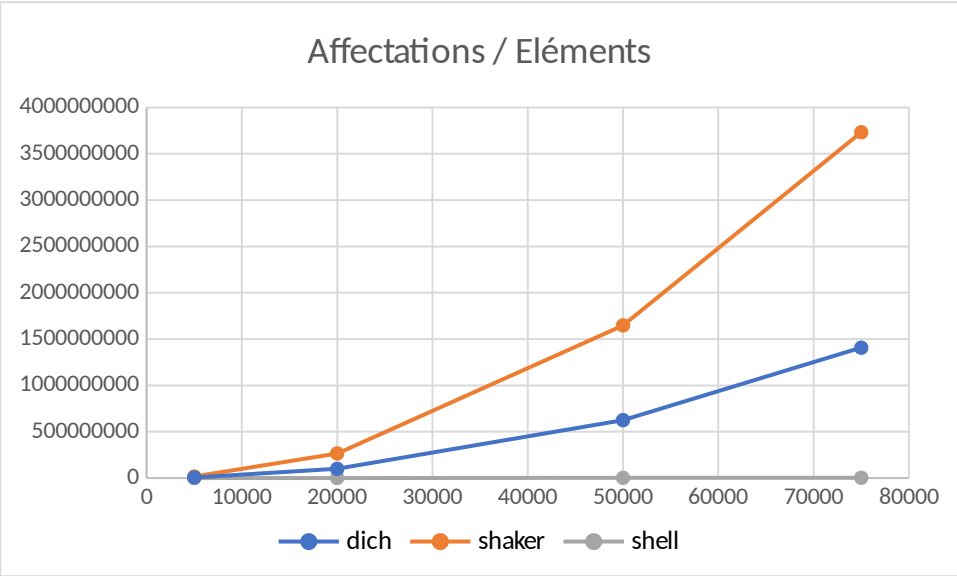


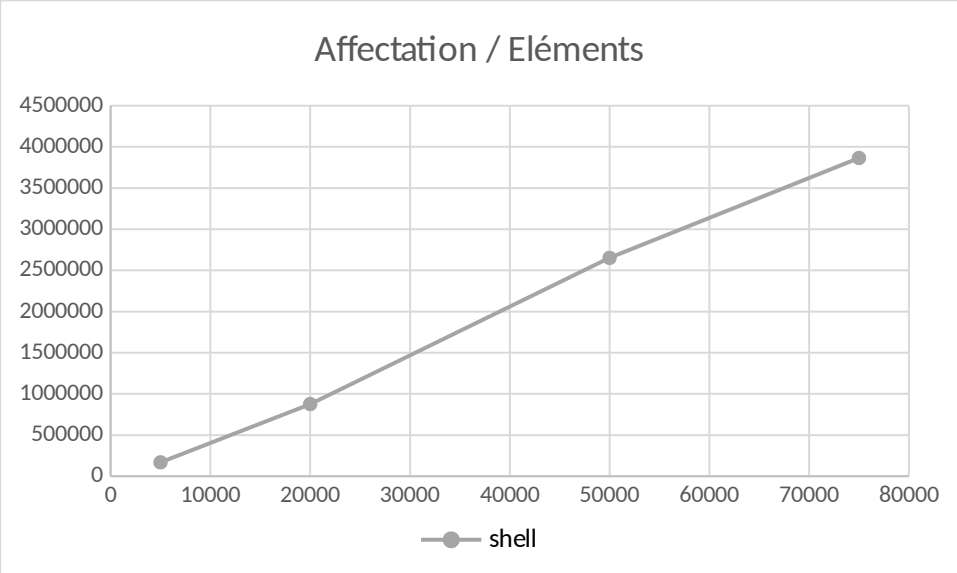
On remarque immédiatement que le shaker est le plus long à s'exécuter. Il apparaît bien à part des deux autres méthodes sur le graphe. Son temps d'exécution augmente très rapidement quand le nombre d'éléments augmente. En zoomant sur les deux autres méthodes nous pouvons voir que le tri par dichotomie augmente moins rapidement que le shaker et toujours avec un temps moindre. Le shell quant à lui augmente très lentement, de façon assez linéaire sur le graphe.





De manière assez identique aux graphes de temps, le shaker semble isolé pour le nombre de comparaisons. Il est une nouvelle fois bien au-dessus des deux autres méthodes et augmente de façon très rapide quand le nombre d'éléments augmente. En zoomant sur le tri par insertion dichotomique et le tri shell, on remarque que les deux semblent assez proches et augmentent plutôt linéairement, mais le tri shell semble se détacher au fur et à mesure de l'augmentation du nombre d'éléments.





Une nouvelle fois le shaker est le moins performant. Il augmente là encore de façon très rapide et dans des valeurs bien supérieures aux autres tris. Le tri dichotomique augmente moins rapidement et dans des valeurs moindres. Cependant les deux méthodes sont bien au-dessus du tri shell en terme de nombre d'affectations. En zoomant sur celui-ci on remarque une progression linéaire.

Question 4

a) **Pour chaque tri, donner quelques arguments simples qui expliquent pourquoi l'algorithme fonctionne (en quoi est-ce que c'est trié ?).**

- Tri par insertion dichotomique

Sur le fonctionnement de son tri, le tri par insertion dichotomique n'est pas différent du tri par insertion « classique ». On considère notre tableau en deux parties, l'une triée et l'autre non, on parcourt la partie non triée du tableau en rangeant l'élément courant dans la partie triée à sa place en décalant les éléments. Ainsi la partie triée grandit jusqu'à être tout le tableau.

La différence avec le tri par Insertion « classique » est la recherche de la bonne position pour l'élément. En effet plutôt que de parcourir la partie triée du tableau jusqu'à trouver l'indice qui nous intéresse, on fait une recherche dichotomique.

- Tri bulle mélangé

L'algorithme employé ici est très simple, on parcourt le tableau dans un sens en comparant les éléments un à un. Si les deux éléments ont besoin d'être échangés, on les échange sinon on passe aux suivants. Une fois le parcours complété, si le tableau n'est pas trié, on répète la manœuvre dans l'autre sens, et ce jusqu'à ce que l'on ne fasse plus aucun échange durant un parcours.

On voit bien ici qu'à chaque parcours, les grands éléments sont approchés de la fin du tableau tandis que les petits sont approchés du début. On peut considérer le tableau réparti en 3 zones, la zone centrale est non triée et les deux autres sont triées. A chaque parcours complet du tableau, on amène soit le maximum/minimum (selon le sens du parcours) de la zone non triée dans la zone triée correspondante. Étant donné qu'il s'agit à chaque fois des extrêmes, dès qu'il atteint une zone triée, un élément est à sa position finale. Ainsi la zone centrale diminue à chaque parcours et le tableau fini trié. Évidemment à chaque boucle parcourant le tableau afin de swapper les éléments, on ne parcourt que la zone centrale pour une question de rapidité.

- Shell sort

Pour réaliser le tri Shell, on considère une suite de valeur k allant jusqu'à 1 (nous avons choisi la suite de division successive de n par 2 soit : n/2, n/4, ..., 1). Pour chaque valeur de k, nous réalisons un tri par insertion sur k sous-tableaux de taille n/k, les sous-tableaux sont composées de la façon suivante :

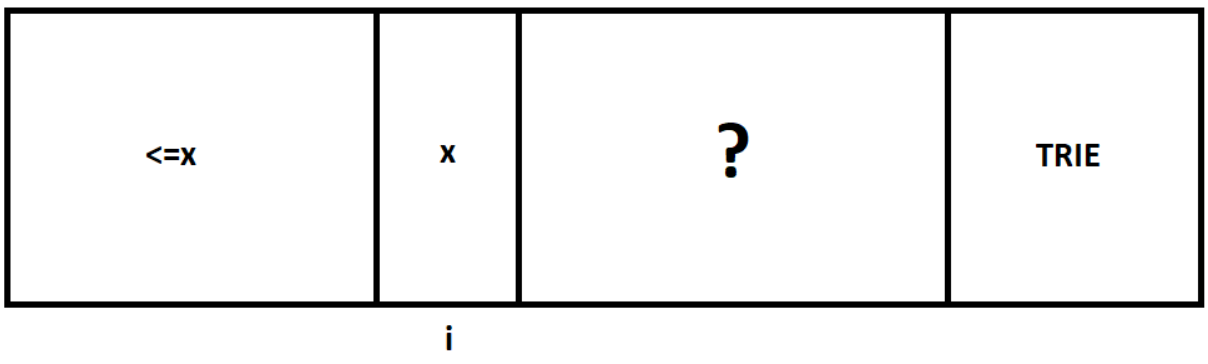
- L1 : [0, k, 2k, ...]
- L2 : [1, k+1, 2k+1, ...]
- L3 : [1, k+2, 2k+2, ...]
- Etc.

Une fois ces sous-tableaux triées, on change de valeur de k jusqu'à obtenir 1, on finira ainsi par un tri par insertion sur le tableau complet (ce qui nous garantit un tableau trié).

L'idée est de faire des tris par insertion sur des tableaux presque triés de plus en plus grands, ainsi chaque tri par insertion a une complexité minimale ≈O(n).

b) **En particulier, pour le tri bulle mélangé, commencer par étudier des invariants de boucle du tri bulle (simple), puis s'en servir pour expliquer sa variante.**

Voici l'invariant du tri bulle sous forme schématique :



Lors d'un tri bulle classique, on considère deux parties dans le tableau, la première non triée et la seconde triée. A Chaque boucle, on part de l'indice i=0 du tableau et on parcoure le tableau jusqu'au dernier élément non trié d'indice k. Durant le parcours du tableau, on compare la valeur à l'indice i et la valeur à l'indice i+1. Si t[i]>t[i+1] on échange les deux valeurs. Sinon on avance simplement. De cette façon, on fait remonter l'élément dont la valeur est la plus élevée jusqu'à la fin de la partie non triée. On réduit ainsi cette partie.

L'invariant est donc le fait que à chaque itération t[i] est le maximum de t[0…i]. Dans le tri Shaker sort, on parcourt le tableau dans les deux sens, l'invariant devient donc le maximum ou le minimum (selon le sens) de la partie centrale non triée du tableau.

On peut poser une variable idInf (indice le plus faible de la partie non triée) et idSup (indice le plus élevé de la partie non triée), ainsi t[i] est maximum de t[idInf…i] ou minimum de t[i…idSup] selon le sens de parcours.

c) **En particulier, pour le *shell sort*, quelle propriété simple doit avoir la séquence des valeurs de *k* pour garantir le tri ?**

Elle doit se terminer par 1 pour assurer que le tri se termine car la dernière passe sera ainsi un tri par insertion (rapide du fait des passes précédentes).

Question 5

a) **Etudier le coût en temps de chacun des algorithmes. Bien préciser si vous parlez de pire cas, de meilleur cas ou de cas moyen. Pour un pire cas ou un meilleur cas, il faut préciser possible sur quelle instance il se manifeste.**

Tri par insertion dichotomique :

Comme pour le tri par insertion, on traite les n éléments du tableau. On boucle donc n fois. Cette boucle effectue un traitement qui cherche la position d'insertion. Mais ici la recherche n'a pas un coût de n mais de $\log_2(n)$ car nous utilisons une recherche dichotomique. Ainsi le coût est en $O(n \log_2(n))$ (cas moyen) .

Tri Shell :

Pour cet algo, on choisit une façon de déterminer une suite de valeurs allant de n à 1. Par exemple nous avons décidé de diviser successivement n par 2 jusqu'à 1. Ces valeurs sont nommées k.

Pour chaque valeur k, on sépare le tableau en k sous-listes que l'on trie. Ces tris sont des tris par insertion que l'on traite ici dans leur meilleur cas, à savoir un tableau presque trié. De fait le coût de chacun de ces tris est O(x) avec x taille du tableau où x vaut donc ici (n/k), soit le nombre d'éléments de nos sous-listes.

On effectue donc k tris de coût (n/k). Soit un cout total de O(n) pour chaque k.

Ayant choisi de diviser n par 2 jusqu'à une valeur de 1 nous allons avoir $\log_2(n)$ valeurs de k.

Soit un coût final de $O(n \log_2(n))$.
Nous nous plaçons ici dans le meilleur des cas, les autres cas étant plus complexes et dépendant de plus de paramètres.

Tri Shaker :

On parcourt une première fois tout le tableau.
Au meilleur cas le tableau est déjà trié. On a donc une complexité en O(n).
En cas moyen (comme en pire) on fait n passes sur n-i éléments (où i est le nombre d'éléments déjà triés sur lesquels on ne repasse pas). On a donc une complexité en $O(n^2)$.

b) **Pour le tri par insertion avec dichotomie, identifier si la dichotomie fait gagner en performance et si oui en quoi.**

Elle fait gagner en performance dans le cas moyen car on a besoin de moins de comparaisons pour déterminer un indice d'insertion.

c) **Comparer les résultats théoriques obtenus avec les performances mesurées précédemment.**

Tri par insertion dichotomique :

Dans le cas moyen du tri par insertion avec recherche dichotomique, nous observons une complexité $O(n\log_2(n))$. Cela correspond aux résultats obtenus graphiquement, où nous remarquions une évolution des coûts quasi-linéaire en fonction du nombre d'élément.

Tri Shell :

Nous remarquons une courbe quasiment similaire à celle trouvée pour le tri par insertion dichotomique, à ceci près que le tri Shell est beaucoup plus performant en nombre d'affectations et en durée que le tri par insertion dichotomique. Cette inclinaison quasi-linéaire de courbe correspond également à la complexité $n\log_2(n)$ trouvée précédemment.

Shaker :

Le Shaker Sort quant à lui est bien moins performant que les deux autres et a une complexité supérieure ($O(n^2)$). Ce qui correspond à nouveau aux courbes de la question 3, où nous avions trouvé une croissance quadratique du coût de l'algorithme en fonction de la taille du tableau.

Question 6

a) **Le tri par insertion dichotomique améliore-t-il le tri par insertion ?**

Dans un cas moyen, la recherche dichotomique améliore le tri par insertion classique car elle permet de réduire le nombre de comparaison, ainsi, si le coût moyen reste le même ($O(n\log_2(n))$), l'algorithme devient plus performant.

Remarque : Cela n'est pas nécessairement le cas si le tableau est déjà presque trié, dans ce cas là, un tri par insertion classique aura besoin de très peu de comparaison (parfois même moins qu'avec une recherche dichotomique).

Ainsi, si la recherche dichotomique n'améliore pas toujours le tri par insertion, elle ne peut cependant pas le rendre significativement moins performant.

b) **Le tri bulle mélangé améliore-t-il le tri bulle ?**

Le tri bulle mélangé est une simple optimisation qui permet de plus rapidement replacer en début de tableau des petits éléments qui se trouvaient en fin. Comme pour la recherche dichotomique pour le tri par insertion, cela ne change pas la complexité de l'algorithme mais permet tout de même de gagner en performance dans le cas ou des valeurs faibles se trouvent en fin de tableau.

c) **Le *shell sort* améliore-t-il le tri par insertion ?**

Le shell sort consiste à trier le tableau en plusieurs fois en le considérant séparé de différentes façons en sous-tableaux en fonction de valeurs déterminées. Il faut alors réaliser un tri par insertion sur chacun des sous-tableaux de façon à les faire sur des tableaux presque triés afin d'obtenir un cas quasi-idéal pour chaque tri par insertion. De la même façon que pour les deux cas précédents, s'il n'améliore pas toujours significativement l'algorithme de base, il ne peut pas faire beaucoup moins bien. Typiquement un tri par insertion classique ira plus vite sur un tableau déjà trié ou presque (cas que l'on exploite donc au maximum dans le tri shell).