Corrections des Fiches de TD
R. Mandiau

Fiche TD 1

Exercice 1 : **Tri par sélection.** On considère le tri suivant de n nombres rangés dans un tableau A : on commence par trouver le plus petit élément de A et on le permute avec $A[1]$. On continue de cette manière pour les $n-1$ premiers éléments de A. Écrire le pseudo-code pour cet algorithme. Donner le temps d'exécution pour le cas optimal et le pire des cas.

Exercice 2 : Le **Tri à bulles** est un algorithme défini par (Cf. annexe pour une version optimisée) :

Algorithme 1 : Tri_Bulles(A,n)

```
   Entrées : A : tableau[1..MAX] d'entier ; n : entier ;
1.1 début
1.2   pour i ← 1 à n faire
1.3     pour j ← n à i + 1 faire
1.4       si (A[j] < A[j − 1]) alors
1.5         swap(A[j], A[j − 1]) ;
```

1. Quel est le temps d'exécution du tri à bulles dans le cas le plus défavorable ?
2. Quelles sont ses performances, comparées au tri par insertion et au tri fusion ?
3. Quel est le temps d'exécution du tri cocktail dans le pire des cas (optionnel) ?

Exercice 3 : Le tri par insertion peut être exprimé sous la forme d'une procédure récursive de la manière suivante. Pour trier $A[1..n]$, on trie récursivement $A[1..n-1]$ puis on insère $A[n]$ dans le tableau trié $A[1..n-1]$. Écrire une récurrence pour le temps d'exécution de cette version récursive du tri par insertion.

Exercice 4 : Déterminer la complexité dans le cas favorable et le cas défavorable pour la recherche d'un palindrome.

Exercice 5 : **Recherche séquentielle** : Déterminer les complexités (cas favorable, cas moyen, cas défavorable) pour l'algorithme de recherche séquentielle d'un élément dans une liste.

Exercice 1 : **Tri par sélection.** On considère le tri suivant de n nombres rangés dans un tableau A : on commence par trouver le plus petit élément de A et on le permute avec $A[1]$. On continue de cette manière pour les $n-1$ premiers éléments de A. Écrire le pseudo-code pour cet algorithme. Donner le temps d'exécution pour le cas optimal et le pire des cas.

Le tri par sélection est décrit par l'algorithme 2.

Algorithme 2 : Tri_selection(A,n)

```
   Entrées : A : tableau[1..MAX] d'entier ; n : entier ;
   Sorties : A : tableau[1..MAX] d'entier ;
   Données : i, j, ind : entier ;
2.1 début
2.2     pour i ← 1 à n − 1 faire
2.3         ▷ Recherche valeur min.
2.4         ind ← i ;
2.5         pour j ← i + 1 à n faire
2.6             si (A[j] < A[ind]) alors
2.7                 ind ← j ;
2.8         ▷ Permutation si nécessaire
2.9         si (ind ≠ i) alors
2.10            swap (A[i], A[ind]) ;
```

Notons que le cas favorable considère la liste triée en ordre croissant. Dans le cas défavorable, la liste est généralement triée en sens inverse. Toutefois, cette méthode de tri ne vérifie pas ce principe général. En effet, elle utilise une liste définie telle que le nombre d'exécutions de la ligne 2.7 soit maximisé :

A[1]	A[2]	A[3]	A[4]	A[5]	coût de 2.7
20	40	30	10	0	
20	40	30	10	0	2
0	40	30	10	20	2
0	**10**	30	40	20	1
0	**10**	**20**	40	30	1
0	**10**	**20**	**30**	**40**	**6**

Nous estimons pour chaque opération fondamentale de cette méthode, le coût et le nombre d'itérations dans les cas favorable et défavorable (Tableau 1).

TABLE 1 – Algorithme et analyse pour le tri sélection

Algorithme	*coût*	*Fav.*	*Defav.*
Algorithme 3 : Tri_selection(A,n)			
Entrées : A : tableau[1..MAX] d'entier ; n : entier ;			
Sorties : A : tableau[1..MAX] d'entier ;			
Données : i, j, ind : entier ;			
3.1 **début**			
3.2 **pour** $i \leftarrow 1$ **à** $n-1$ **faire**	c_1	$(n-1)+1$	$(n-1)+1$
3.3 ▷ Recherche valeur min.	0	-	-
3.4 $ind \leftarrow i$;	c_2	$n-1$	$n-1$
3.5 **pour** $j \leftarrow i+1$ **à** n **faire**	c_3	$c_{iter} + c_{sortie}$	$c_{iter} + c_{sortie}$
3.6 **si** $(A[j] < A[ind])$ **alors**	c_4	c_{iter}	c_{iter}
3.7 $ind \leftarrow j$;	c_5	0	c_{iter}
3.8 ▷ Permutation si nécessaire	0	-	-
3.9 **si** $(ind \neq i)$ **alors**	c_6	$(n-1)$	$(n-1)$
3.10 **swap**$(A[i], A[ind])$;	c_7	0	$(n-1)$

Le tableau 2 estime le comportement de la variable j :
— un nombre d'itérations (désigné par c_{iter}), et
— la sortie de la structure Pour (c_{sortie}).

TABLE 2 – Déroulement du tri par sélection pour les variables i et j

Variable i	Variable j	
	Nombre d'itérations	Sortie
	c_{iter}	c_{sortie}
1	$(n-1)$	1
2	$(n-2)$	1
3	$(n-3)$	1
...	...	1
$n-1$	1	1
Cumul	$1+2+\ldots+(n-1)$	$1+1+\ldots+1$

Or nous savons que : $\sum_{i=1}^{n-1} i = \frac{n\cdot(n-1)}{2}$ et $\sum_{i=1}^{n-1} 1 = n-1$.

TABLE 3 – Algorithme et analyse pour le tri sélection

	Algorithme	*coût*	*Fav.*	*Defav.*
	Algorithme 4 : Tri_selection(A,n)			
	Entrées : A : tableau[1..MAX] d'entier ; n : entier ;			
	Sorties : A : tableau[1..MAX] d'entier ;			
	Données : i, j, ind : entier ;			
4.1	**début**			
4.2	**pour** $i \leftarrow 1$ **à** $n-1$ **faire**	c_1	n	n
4.3	▷ Recherche valeur min.	0	-	-
4.4	$ind \leftarrow i$;	c_2	$n-1$	$n-1$
4.5	**pour** $j \leftarrow i+1$ **à** n **faire**	c_3	$\frac{n^2-n}{2} + (n-1)$	$\frac{n^2-n}{2} + (n-1)$
4.6	**si** $(A[j] < A[ind])$ **alors**	c_4	$\frac{n^2-n}{2}$	$\frac{n^2-n}{2}$
4.7	$ind \leftarrow j$;	c_5	0	$\frac{n^2-n}{2}$
4.8	▷ Permutation si nécessaire	0	-	-
4.9	**si** $(ind \neq i)$ **alors**	c_6	$(n-1)$	$(n-1)$
4.10	**swap**($A[i]$, $A[ind]$) ;	c_7	0	$(n-1)$

Note importante : Pour l'instruction de la ligne 4.7 concernant le coût c_5, nous avons ici considéré une borne supérieure. En fait, en pratique, le nombre d'itérations dans le cas défavorable n'est pas aussi élevé.

$T_{Fav}(n) = c_1 \cdot n + c_2 \cdot (n-1) + c_3 \left(\frac{n^2-n}{2} + (n-1)\right) + c_4 \cdot \frac{n^2-n}{2} + c_6 \cdot (n-1)$.

Nous en déduisons ainsi $T_{Fav}(n) = \left(\frac{c_3}{2} + \frac{c_4}{2}\right) \cdot n^2 + \left(c_1 + c_2 + \frac{c_3}{2} - \frac{c_4}{2} + c_6\right) \cdot n + (-c_2 - c_3 - c_6)$. Pour conclure, nous pouvons admettre que $T_{Fav}(n) = \Omega(n^2)$.

Un travail similaire pour le cas défavorable : $T_{Defav}(n) = c_1 \cdot n + c_2 \cdot (n-1) + c_3 \left(\frac{n^2-n}{2} + (n-1)\right) + c_4 \cdot \frac{n^2-n}{2} + c_5 \cdot \frac{n^2-n}{2} + c_6 \cdot (n-1) + c_7 \cdot (n-1)$.

$$T_{Defav}(n) = \left(\frac{c_3}{2} + \frac{c_4}{2} + \frac{c_5}{2}\right) \cdot n^2 + \left(c_1 + c_2 + \frac{c_3}{2} - \frac{c_4}{2} - \frac{c_5}{2} + c_6 + c_7\right) \cdot n + (-c_2 - c_3 - c_6 - c_7)$$

Nous pouvons également conclure que la complexité dans le pire des cas est en $O(n^2)$.

En résumé, la complexité temporelle est $T(n) = \Theta(n^2)$.

Question supplémentaire ... et quid de la ligne 4.9 ?

Rappel :
— $T_{Fav}(n) = \left(\frac{c_3}{2} + \frac{c_4}{2}\right) \cdot n^2 + \left(c_1 + c_2 + \frac{c_3}{2} - \frac{c_4}{2} + c_6\right) \cdot n + (-c_2 - c_3 - c_6)$
— $T_{Defav}(n) = \left(\frac{c_3}{2} + \frac{c_4}{2} + \frac{c_5}{2}\right) \cdot n^2 + \left(c_1 + c_2 + \frac{c_3}{2} - \frac{c_4}{2} - \frac{c_5}{2} + c_6 + c_7\right) \cdot n + (-c_2 - c_3 - c_6 - c_7)$

En terme pratique

Mesure de performance : Nous considérons les durées (temps CPU), mesure utilisée mais qui reste critiquable.

Courbe de tendance : Nous considérons que chaque coût c_i est équivalent (coût unitaire). Le principe est d'ajouter un compteur qui sera incrémenté à chaque opération fondamentale.

— $T_{Fav}(n) = n^2 + 3 \cdot n - 3$
— $T_{Defav}(n) = \frac{3}{2} \cdot n^2 + \frac{7}{2} \cdot n - 4$

Comme nous avons pris une hypothèse majorant le coût de la ligne 4.7 (correspondant au coût c_5). L'évaluation va donc différer entre sa mesure pratique et la courbe de tendance pour le cas défavorable (pour le cas favorable, il y a bien correspondance entre la mesure pratique et la courbe de tendance). Par exemple, nous pouvons estimer le cas défavorable pour $n = 5$ à $T_{Defav}(n) = \frac{3}{2} \cdot n^2 + \frac{7}{2} \cdot n - 4 = 51$ (le cas favorable $T_{Fav}(n) = n^2 + 3 \cdot n - 3 = 37$).

Cas	A[1]	A[2]	A[3]	A[4]	A[5]	coût pratique	évaluation
Favorable	10	20	30	40	50	37	37
Défavorable	20	40	30	10	0	47	51

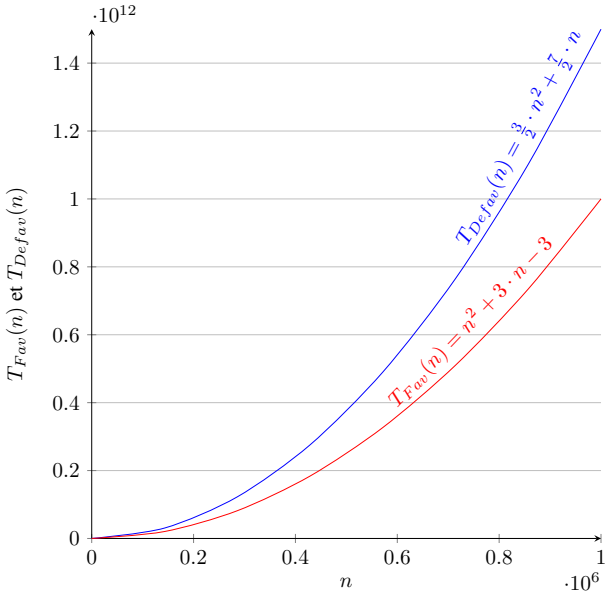

FIGURE 1 – Courbe de tendance cas Favorable/ Défavorable pour le tri sélection

Exercice (à faire) Compléter l'exercice

1. Nous pouvons améliorer l'estimation de la borne supérieure en évaluant plus précisément la ligne 4.7 (constante c_5).
2. Comparer avec la version récursive

Exercice 2 : Le **Tri à bulles** est un algorithme défini par :

Algorithme 5 : Tri_Bulles(A,n)

```
Entrées : A : tableau[1..MAX] d'entier ; n : entier ;
5.1 début
5.2     pour i ← 1 à n faire
5.3         pour j ← n à i + 1 faire
5.4             si (A[j] < A[j − 1]) alors
5.5                 permuter(A[j], A[j − 1]) ;
```

1. Quel est le temps d'exécution du tri à bulles dans le cas le plus défavorable ?
2. Quelles sont ses performances, comparées au tri par insertion et au tri fusion ?
3. Quel est le temps d'exécution du tri cocktail dans le pire des cas (optionnel) ?

Question 1 : Montrons que la complexité est quadratique dans le cas favorable (Tableau 4).

TABLE 4 – Cas Favorable pour le tri à Bulles

	Algorithme	*coût*	*Nombre de fois*
5.1	**début**		
5.2	**pour** $i \leftarrow 1$ **à** n **faire**	c_1	$n+1$
5.3	**pour** $j \leftarrow n$ **à** $i+1$ ***par pas*** -1 **faire**	c_2	$\sum_{i=1}^{n}(n-i) + \sum_{i=1}^{n} 1$
5.4	**si** $(A[j] < A[j-1])$ **alors**	c_3	$\sum_{i=1}^{n}(n-i)$
5.5	**swap**$(A[j], A[j-1])$;	c_4	0

Cas Favorable : Nous pouvons alors estimer le nombre d'opérations du tri à bulles dans le cas favorable par : $T_{Fav}(n) = c_1 \cdot (n+1) + c_2 \cdot (\sum_{i=1}^{n}(n-i) + \sum_{i=1}^{n} 1) + c_3 \cdot \sum_{i=1}^{n}(n-i) + c_4 \cdot 0$.

Nous savons que $\sum_{i=1}^{n}(n-i) = \frac{n\cdot(n-1)}{2}$. La somme initiale devient en effectuant les remplacements des différents sommes : $T_{Fav}(n) = c_1 \cdot (n+1) + c_2 \cdot \left(\frac{n\cdot(n-1)}{2} + n\right) + c_3 \cdot \frac{n\cdot(n-1)}{2}$. Après factorisation, nous obtenons le résultat suivant pour la complexité : $T_{Fav}(n) = n^2 \cdot \left(\frac{c_2}{2} + \frac{c_3}{2}\right) + n \cdot \left(c_1 + \frac{c_2}{2} - \frac{c_3}{2}\right) + c_1$. Pour conclure, la complexité temporelle du tri à bulles dans le cas favorable est donc en $T_{Fav}(n) = \Omega\left(n^2\right)$.

Cas défavorable : L'algorithme tri à bulles se comporte de la manière similaire pour le cas favorable et le cas défavorable, excepté pour la ligne .5. Dans le pire des cas, cette ligne est évaluée par le nombre d'itérations (i.e. $\sum_{i=1}^{n}(n-i)$) multiplié par la complexité de la procédure **swap**. Le coût de cette procédure avait été évalué en coût constant. Nous en déduisons que cet algorithme est estimé en $\sum_{i=1}^{n}(n-i)$ (en remplacement de 0 du Tableau 4). Il est alors possible de montrer que la complexité de l'algorithme de tri est en : $T_{Defav}(n) = n^2 \cdot \left(\frac{c_2}{2} + \frac{c_3}{2} + \frac{c_4}{2}\right) + n \cdot \left(c_1 + \frac{c_2}{2} - \frac{c_3}{2} - \frac{c_4}{2}\right) + c_1$.

Le temps d'exécution est donc une fonction quadratique. La complexité temporelle du tri à bulles dans le cas défavorable est en $T_{Defav}(n) = O\left(n^2\right)$.

Exercice 3 : Le tri par insertion peut être exprimé sous la forme d'une procédure récursive de la manière suivante. Pour trier $A[1..n]$, on trie récursivement $A[1..n-1]$ puis on insère $A[n]$ dans le tableau trié $A[1..n-1]$. Écrire une récurrence pour le temps d'exécution de cette version récursive du tri par insertion.

Faisons une petite digression en rappelant quelques éléments du cours. Le tri par insertion est défini par l'algorithme 6.

Algorithme 6 : Tri_insertion (A,n)

Entrées : A : tableau[$1..MAX$] d'entier ; n entier ;
Sorties : A : tableau[$1..MAX$] d'entier ;
Données : i, j, cle : entier ;

```
6.1 début
6.2   pour i ← 2 à n faire
6.3     cle ← A[i];
6.4     ▷ Insérer A[i] dans la liste triée A[1..i − 1]
6.5     j ← i − 1;
6.6     tant que (j > 0) et (A[j] > cle) faire
6.7       A[j + 1] ← A[j];
6.8       j ← j − 1;
6.9     A[j + 1] ← cle;
```

TABLE 5 – Algorithme du tri par insertion et son analyse

	Algorithme	*coût*	*Fav.*	*Defav.*
6.1	**début**			
6.2	**pour** $i \leftarrow 2$ **à** n **faire**	c_1	n	n
6.3	$cle \leftarrow A[i]$;	c_2	$n-1$	$n-1$
6.4	▷ Insérer $A[i]$ dans la liste triée			
6.5	$j \leftarrow i-1$;	c_3	$n-1$	$n-1$
6.6	**tant que** $(j > 0)$ ***et*** $(A[j] > cle)$ **faire**	c_4	$n-1$	$\frac{n\cdot(n-1)}{2} + (n-1)$
6.7	$A[j+1] \leftarrow A[j]$;	c_5	0	$\frac{n\cdot(n-1)}{2}$
6.8	$j \leftarrow j-1$;	c_6	0	$\frac{n\cdot(n-1)}{2}$
6.9	$A[j+1] \leftarrow cle$;	c_7	$n-1$	$n-1$

Rappelons que pour un algorithme quelconque de tri, le meilleur des cas est caractérisé par une liste initiale supposée déjà triée.

Cas favorable : Montrons que la complexité est linéaire dans le cas favorable (Tableau 5) :

Nous cumulons les coûts correspondants pour chaque opération : $T_{Fav}(n) = c_1 \cdot n + c_2 \cdot (n-1) + c_3 \cdot (n-1) + c_4 \cdot (n-1) + c_7 \cdot (n-1)$. En factorisant, nous obtenons : $T_{Fav}(n) = (c_1 + c_2 + c_3 + c_4 + c_7) \cdot n - (c_2 + c_3 + c_4 + c_7)$. Nous admettons que le temps d'exécution peut se définir par une fonction linéaire de la forme $T_{Fav}(n) = a \cdot n + b$, avec $a = c_1 + c_2 + c_3 + c_4 + c_7$ et $b = c_2 + c_3 + c_4 + c_7$. La complexité temporelle dans le cas favorable est en $\Omega(n)$.

Cas défavorable : Montrons que la complexité est quadratique dans le cas défavorable (Tableau 6). Le cas défavorable correspond à une liste triée inversée.

TABLE 6 – Déroulement du tri par insertion pour les variables i et j

Variable i	Variable j du tant que	
	Nombre d'itérations	Sortie
2	1	1
3	2	1
...	...	1
n	$(n-1)$	1
Cumul	$1 + 2 + \ldots + (n-1)$	$1 + 1 + \ldots + 1$

Or nous savons que : $\sum_{i=2}^{n}(i-1) = \frac{n\cdot(n-1)}{2}$ et $\sum_{i=2}^{n} 1 = n-1$.

Nous pouvons alors estimer le nombre d'opérations du tri par insertion dans le cas défavorable par : $T_{Defav}(n) = c_1 \cdot n + c_2 \cdot (n-1) + c_3 \cdot (n-1) + c_4 \cdot (\frac{n\cdot(n-1)}{2} + (n-1)) + c_5 \cdot \frac{n\cdot(n-1)}{2} + c_6 \cdot \frac{n\cdot(n-1)}{2} + c_7 \cdot (n-1)$. Après factorisation, nous obtenons le résultat suivant pour la complexité : $T_{Defav}(n) = (\frac{c_4}{2} + \frac{c_5}{2} + \frac{c_6}{2}) \cdot n^2 + (c_1 + c_2 + c_3 + \frac{c_4}{2} - \frac{c_5}{2} - \frac{c_6}{2} + c_7) \cdot n - (c_2 + c_3 + c_4 + c_7)$. Le temps d'exécution est alors une fonction quadratique $T_{Defav}(n) = a \cdot n^2 + b \cdot n + c$, avec les constantes suivantes : $a = \frac{c_4}{2} + \frac{c_5}{2} + \frac{c_6}{2}$, $b = c_1 + c_2 + c_3 + \frac{c_4}{2} - \frac{c_5}{2} - \frac{c_6}{2} + c_7$ et $c = -(c_2 + c_3 + c_4 + c_7)$. Le tri par insertion a ainsi un temps d'exécution dans le cas le plus défavorable de $O(n^2)$.

Pour conclure, le tri par insertion a une complexité temporelle entre $\Omega(n)$ et $O(n^2)$.

Revenons donc à l'exercice, nous aimerions écrire l'algorithme dans sa version récursive. Le tri récursif par insertion se définit à l'aide d'un algorithme, qui permet d'effectuer ses différents appels récursifs et d'insérer chaque élément $A[n]$ dans le tableau trié. L'algorithme 7 décrit ce tri récursif. L'algorithme 8 se propose une insertion d'un élément à sa bonne place dans la liste.

Algorithme 7 : Tri_inser_rec (A, n)

Entrées : tableau[1..MAX] d'entier ; n : entier ;
Sorties : tableau[1..MAX] d'entier ;

```
7.1 début
7.2     si n > 1 alors
7.3         Tri_inser_rec (A, n − 1) ;
7.4         Inserer (A, n) ;
```

Algorithme 8 : Inserer (A, ind)

Entrées : A : tableau[1..MAX] d'entier ; ind : entier ;
Sorties : A : tableau[1..MAX] d'entier ;
Données : i, val : entier ;

```
8.1 début
8.2     si A[ind] < A[ind − 1] alors
8.3         val ← A[ind] ;
8.4         i ← ind ;
8.5         tant que (i > 0) et (val ≤ A[i − 1]) faire
8.6             A[i] ← A[i − 1] ;
8.7             i ← i − 1 ;
8.8         A[i] ← val ;
```

Nous pouvons donc affirmer que l'algorithme 8 a une complexité en $\Theta(n)$.

L'algorithme du tri par insertion dans sa version récursive, peut se définir par la relation de récurrence : $T(n) = T(n-1) + \Theta(n)$. Il est donc possible de proposer et de vérifier la conjecture $O(n^2)$.

Prouvons que $T(n) \leq c \cdot n^2$?

Supposons $T(n-1) \leq c \cdot (n-1)^2$. Nous avons : $T(n) = T(n-1) + \Theta(n)$ qui conduit à $T(n) \leq c \cdot (n-1)^2 \leq c \cdot n^2$ pour $n \geq 2$.

— $n = 2 : T(2) = T(1) + 1 = 1 + 1 \leq c \cdot 2^2$

— $n = 3 : T(3) = T(2) + 2 = 4 \leq c \cdot 3^2$

Prenons $c = 1$ et $n_0 = 2$ pour vérifier l'inégalité et donc la conjecture.

Il serait intéressant de comparer les versions itérative et récursive. Qu'en pensez vous ?

Notre approximation précédente n'est plus suffisante. Nous devons proposer une analyse approfondie des opérations fondamentales.

TABLE 7 – Algorithme Tri_inser_rec et son analyse

Algorithme	*coût*	*Fav.*	*Defav.*
Algorithme 9 : Tri_inser_rec (A, n)			
Entrées : A : tableau$[1..MAX]$ d'entier ; n : entier ;			
Sorties : A : tableau$[1..MAX]$ d'entier ;			
9.1 **début**			
9.2 **si** $n > 1$ **alors**	C_1	n	n
9.3 Tri_inser_rec $(A, n-1)$;	C_2	$n-1$	$n-1$
9.4 Inserer (A, n) ;	-	-	-

TABLE 8 – Algorithme Inserer et son analyse

Algorithme	*coût*	*Fav.*	*Defav.*
Algorithme 10 : Inserer (A, ind)			
Entrées : A :tableau$[1..MAX]$ d'entier ; ind : entier ;			
Sorties : A : tableau$[1..MAX]$ d'entier ;			
Données : i, val : entier ;			
10.1 **début**			
10.2 **si** $A[ind] < A[ind-1]$ **alors**	c_1	$n-1$	$n-1$
10.3 $val \leftarrow A[ind]$;	c_2	0	$n-1$
10.4 $i \leftarrow ind$;	c_3	0	$n-1$
10.5 **tant que** $(i > 0)$ ***et*** $(val \leq A[i-1])$ **faire**	c_4	0	$\frac{n\cdot(n-1)}{2} + (n-1)$
10.6 $A[i] \leftarrow A[i-1]$;	c_5	0	$\frac{n\cdot(n-1)}{2}$
10.7 $i \leftarrow i-1$;	c_6	0	$\frac{n\cdot(n-1)}{2}$
10.8 $A[i] \leftarrow val$;	c_7	0	$n-1$

Le tableau de l'algorithme **Inserer** définit les coûts cumulés des opérations lors des différents appels.

Pour un tableau de n éléments :

Cas Favorable : $T_{Fav}(n) = c_1 \cdot (n-1) + C_1 \cdot n + C_2 \cdot (n-1)$; d'où la complexité dans le cas favorable est en $(c_1 + C_1 + C_2) \cdot n - (c_1 + C_2)$.

Cas Défavorable : $T_{Defav}(n) = c_1 \cdot (n-1) + c_2 \cdot (n-1) + c_3 \cdot (n-1) + c_4 \cdot \left(\frac{n\cdot(n-1)}{2} + (n-1)\right) + c_5 \cdot \frac{n\cdot(n-1)}{2} + c_6 \cdot \frac{n\cdot(n-1)}{2} + c_7 \cdot (n-1) + C_1 \cdot n + C_2 \cdot (n-1)$; d'où la complexité dans le cas défavorable est en $T_{Defav}(n) = \left(\frac{c_4}{2} + \frac{c_5}{2} + \frac{c_6}{2}\right) \cdot n^2 + \left(c_1 + c_2 + c_3 + \frac{c_4}{2} - \frac{c_5}{2} - \frac{c_6}{2} + c_7 + C_1 + C_2\right) \cdot n + (-c_1 - c_2 - c_3 - c_4 - c_7 - C_2)$.

Courbe de Tendance

Algorithme	$T_{Fav}(n)$	$T_{Defav}(n)$
version itérative	$5 \cdot n - 4$	$\frac{3}{2} \cdot n^2 + \frac{7}{2} \cdot n - 4$
version récursive	$3 \cdot n - 2$	$\frac{3}{2} \cdot n^2 + \frac{11}{2} \cdot n - 6$

Les courbes de tendance mettent en évidence l'efficacité de la version récursive sur la version itérative.

En termes de mesures de performance (mesure du temps que met l'algorithme à s'exécuter), les ordres de grandeurs sont très proches et ne devraient pas influencer les mesures. De plus, attention aux nombres d'appels récursifs qui dépend fortement de la « mémoire fixe réservée » (le nombre d'appels récursifs n'est pas infini sur une machine).

fiche de TD 2

Exercice 1 : Bien que dans le pire des cas, le tri par fusion s'exécute en $\Theta(n.\log_2(n))$ et le tri par insertion en $\Theta(n^2)$, les facteurs constants du tri par insertion le rendent plus rapide pour n petit. Il est naturel d'utiliser le tri par insertion à l'intérieur du tri par fusion lorsque les sous-problèmes deviennent suffisamment petits. On va étudier la modification suivante du tri par fusion : $\frac{n}{k}$ sous-listes de longueur k sont triées via le tri par insertion, puis fusionnées à l'aide du mécanisme de fusion classique, k est une valeur à déterminer.

1. Montrer que les $\frac{n}{k}$ sous-listes, chacune de longueur k, peuvent être triées via le tri par insertion avec un temps $\Theta(n.k)$ dans le cas le plus défavorable.
2. Montrer que les sous-listes peuvent être fusionnées en n : $\Theta(n.\log_2(\frac{n}{k}))$ dans le cas le plus défavorable.
3. Sachant que l'algorithme modifié s'exécute en n : $\Theta(n.k + n.\log_2(\frac{n}{k}))$ dans le cas le plus défavorable, quelle est la plus grande valeur asymptotique (notation Θ) de k en tant que fonction de n, pour lequel l'algorithme modifié a le même temps d'exécution asymptotique que le tri par fusion classique.
4. Comment doit-on choisir k en pratique ?

Exercice 2 : Déterminer la complexité temps de la procédure de Hanoï (inventé par Edouard Lucas, 1892)

Procédure Hanoi(n,A,B,C)

```
   Entrées : n : entier, A,B,C : caractères
-.1 début
-.2   // A : départ, B : intermédiaire, C : Arrivée ;
-.3   si (N = 1) alors
-.4     Ecrire(" Transférer de A vers C ")
-.5   sinon
-.6     Hanoi (n − 1, A, C,B) ;
-.7     Ecrire (" Transferer de A vers C ") ;
-.8     Hanoi (n − 1,B,A,C);
```

Exercice 3 : **Recherche Dichotomique et variante** En supposant une liste A triée d'éléments, on peut comparer le milieu de la séquence avec la valeur v recherchée, et supprimer la moitié de la séquence de la suite des opérations. La recherche dichotomique est un algorithme qui répète cette procédure, en divisant par deux à chaque fois la taille de la partie restante de la liste.

1. Écrire le pseudo code (itératif ou récursif) de la recherche dichotomique. Expliquez pourquoi le temps d'exécution de la recherche dichotomique dans le cas le plus défavorable est $\Theta(\log_2(n))$? Question facultative : temps d'exécution de la recherche dans le cas favorable ?
2. Déterminer la complexité temporelle pour la recherche trichotomique (algorithme identique à la recherche dichotomique mais avec une décomposition en 3 partitions correspondant à trois sous-listes).
3. Effectuer la même étude mais en supposant k partitions ?
4. On souhaite une performance de la recherche " k-tomique " au moins c fois plus rapide que pour une recherche dichotomique simple. Déterminer la valeur de k pour des valeurs de $c = 2$ et $c = 10$?

Exercice 1 : Bien que dans le pire des cas, le tri par fusion s'exécute en $\Theta(n.\log_2(n))$ et le tri par insertion en $\Theta(n^2)$, les facteurs constants du tri par insertion le rendent plus rapide pour n petit. Il est naturel d'utiliser le tri par insertion à l'intérieur du tri par fusion lorsque les sous-problèmes deviennent suffisamment petits. On va étudier la modification suivante du tri par fusion : $\frac{n}{k}$ sous-listes de longueur k sont triées via le tri par insertion, puis fusionnées à l'aide du mécanisme de fusion classique, k est une valeur à déterminer.

1. Montrer que les $\frac{n}{k}$ sous-listes, chacune de longueur k, peuvent être triées via le tri par insertion avec un temps $\Theta(n.k)$ dans le cas le plus défavorable.
2. Montrer que les sous-listes peuvent être fusionnées en n : $\Theta(n.\log_2(\frac{n}{k}))$ dans le cas le plus défavorable.
3. Sachant que l'algorithme modifié s'exécute en n : $\Theta(n.k + n.\log_2(\frac{n}{k}))$ dans le cas le plus défavorable, quelle est la plus grande valeur asymptotique (notation Θ) de k en tant que fonction de n, pour lequel l'algorithme modifié a le même temps d'exécution asymptotique que le tri par fusion classique.
4. Comment doit-on choisir k en pratique ?

Question 1. Montrer que les $\frac{n}{k}$ sous-listes, chacune de longueur k, peuvent être triées via le tri par insertion avec un temps $\Theta(n.k)$ dans le cas le plus défavorable.

Une liste de k éléments est trié dans le cas défavorable en $T(k) = O(k^2)$ (Cf cours). Les $\frac{n}{k}$ sous-listes de k éléments sont triées en $T(n,k) = O(\frac{n \cdot k^2}{k}) = O(n \cdot k)$.

Note : Le passage $O(.)$ en $\Theta(.)$ n'est pas ici gênant.

Question 2. Montrer que les sous-listes peuvent être fusionnées en n : $\Theta(n.\log_2(\frac{n}{k}))$ dans le cas le plus défavorable.

L'idée est d'appliquer le tri fusion (de complexité $\Theta(n \cdot \log_2(n))$) sur des listes triées, chacune de longueur k (grossièrement, les appels récursifs s'arrêtent des l'obtention d'une liste de longueur k). Nous en déduisons que la complexité est donc $T(n,k) = \Theta(n \cdot (\log_2(n) - \log_2(k)))$.
d'où : $T(n,k) = \Theta(n \cdot \log_2(\frac{n}{k}))$.

Question 3. Sachant que l'algorithme modifié s'exécute en n : $\Theta(n.k + n.\log_2(\frac{n}{k}))$ dans le cas le plus défavorable, quelle est la plus grande valeur asymptotique (notation Θ) de k en tant que fonction de n, pour lequel l'algorithme modifié a le même temps d'exécution asymptotique que le tri par fusion classique.

La complexité de l'algorithme modifié est donc en $T(n,k) = \Theta(n \cdot k + n \cdot \log_2(\frac{n}{k}))$.

Question 4. On vous demande de choisir k en pratique tel que l'algorithme modifié a le même temps d'exécution asymptotique que le tri par fusion classique.

$T(n,k) \leq T_{Fusion}(n)$, i.e. $n \cdot k + n \cdot \log_2(\frac{n}{k}) \leq n \cdot \log_2(n)$ (à la constante multiplicative pres).

$$\begin{aligned} n \cdot k + n \cdot \log_2(\tfrac{n}{k}) &\leq n \cdot \log_2(n) \\ k + \log_2(\tfrac{n}{k}) &\leq \log_2(n) \\ k &\leq \log_2(k) \end{aligned}$$

pas de solution pour k.

Lorsque $k = 1$, l'algorithme dégrade légèrement le comportement de l'algorithme : nous passons de $n \cdot \log_2(n)$ (tri fusion) à $n + n \cdot \log_2(n)$ (algo. modifié).

Pour conclure :

— L'analyse a été effectué sans produire le pseudo-code ou le code l'algorithme modifié.

— Malgré une idée très séduisante (*en théorie*), elle n'a malheureusement pas de sens *en pratique*.

L'algorithme 11 est une adaptation du tri par insertion ; tandis que l'algorithme 12 est le tri fusion modifiée.

Algorithme 11 : Tri_insertion_bis (A,g, d)

Entrées : A : tableau[$1..MAX$] d'entier ; g, d entier ;
Sorties : A : tableau[$1..MAX$] d'entier ;
Données : i, j, cle : entier ;

11.1 **début**
11.2 **pour** $i \leftarrow g$ **à** d **faire**
11.3 $cle \leftarrow A[i]$;
11.4 ▷ `Insérer` $A[i]$ `dans la liste triée` $A[1..i-1]$
11.5 $j \leftarrow i-1$;
11.6 **tant que** $(j >= 0)$ ***et*** $(A[j] > cle)$ **faire**
11.7 $A[j+1] \leftarrow A[j]$;
11.8 $j \leftarrow j-1$;
11.9 $A[j+1] \leftarrow cle$;

Algorithme 12 : Tri_Fusion_modif (A,p,r,k)

Entrées : A : $tableau$[$1..MAX$] d'entier ; p, r,k : entier ;
Sorties : A : $tableau$[$1..MAX$] d'entier ;
Données : q : entier ;

12.1 **début**
12.2 **si** $r - p > k$ **alors**
12.3 $q \leftarrow \lfloor \frac{p+r}{2} \rfloor$;
12.4 Tri_Fusion_modif (A,p,q,k) ;
12.5 Tri_Fusion_modif (A,$q+1$, r,k) ;
12.6 Fusion (A,p,q,r) ;
12.7 **sinon**
12.8 Tri_insertion_bis (A,p, r) ;

Dans le cas défavorable :

n	Tri Fusion	Tri Fusion - insertion			
		$k=1$	$k=2$	$k=100$	$k=10000$
0	1	2	2	2	2
10	322	413	349	187	187
100	5320	19834	18938	15352	15352
1000	72844	$1.56466 \cdot 10^6$	$1.56812 \cdot 10^6$	$1.52771 \cdot 10^6$	$1.5035 \cdot 10^6$
10000	931684	$1.5087 \cdot 10^8$	$1.50812 \cdot 10^8$	$1.50457 \cdot 10^8$	$1.50035 \cdot 10^8$
100000	$1.13136 \cdot 10^7$	$1.50109 \cdot 10^{10}$	$1.50099 \cdot 10^{10}$	$1.50064 \cdot 10^{10}$	$1.50028 \cdot 10^{10}$
300000	$3.67543 \cdot 10^7$	$1.35035 \cdot 10^{11}$	$1.35033 \cdot 10^{11}$	$1.35023 \cdot 10^{11}$	$1.3501 \cdot 10^{11}$

Exercice 2 : Déterminer la complexité temps de la procédure de Hanoï (inventé par Edouard Lucas, 1892)

Algorithme 13 : Hanoi(n,A,B,C)

Entrées : n : entier, A,B,C : caractères

```
13.1 début
13.2   ▷ A : départ, B : intermédiaire, C : Arrivée ;
13.3                                                  ▷ condition en Θ(1)
13.4   si (N = 1) alors
13.5     Ecrire(" Transférer de A vers C ")
13.6   sinon
13.7     Hanoi (n − 1, A, C,B) ;                      ▷ T(n − 1)
13.8     Ecrire (" Transferer de A vers C ") ;
13.9     Hanoi (n − 1,B,A,C) ;                        ▷ T(n − 1)
```

Rappelons que la primitive *Ecrire* n'est pas considérée comme une opération fondamentale. Nous pouvons facilement caractériser la relation de récurrence pour la procédure **Hanoi** :

$$T(n) = \begin{cases} \Theta(1) & \text{si } n = 1, \\ 2 \cdot T(n-1) + \Theta(1) & \text{sinon} \end{cases}$$

Posons la suite (U_n) telle que

$$T(n) = \begin{cases} 1 & \text{si } n = 1, \\ 2 \cdot U_{n-1} + 1 & \text{sinon} \end{cases}$$

Petit rappel :
— $\sum_{i=0}^{n} x^i = \frac{x^{n+1}-1}{x-1}$
— $\sum_{i=0}^{\infty} x^i = \frac{1}{1-x}$, pour $|x| < 1$

$$\begin{aligned} U_n &= 2 \cdot U_{n-1} + 1 \\ U_n &= 2 \cdot (2 \cdot U_{n-2} + 1) + 1 = 2^2 \cdot U_{n-2} + 2 + 1 \\ U_n &= 2^2 \cdot (2 \cdot U_{n-3} + 1) + 2 + 1 = 2^3 \cdot U_{n-3} + 2^2 + 2 + 1 \\ &\dots \\ &\dots \\ U_n &= 2^n \cdot U_1 + \sum_{i=0}^{n-1} 2^i \\ U_n &= 2^n + \sum_{i=0}^{n-1} 2^i \end{aligned}$$

$$\begin{aligned} U_n &= 2^n + \tfrac{1-2^n}{1-2} \\ U_n &= 2^n + 2^n - 1 \\ U_n &= 2^{n+1} - 1 \end{aligned}$$

Nous en déduisons que la complexité temporelle de Hanoi est en $T(n) = \Theta(2^{n+1})$.

Signification : E. Lucas précisait dans son histoire de moines bouddhistes déplaçant un disque par an, que la fin du monde serait prévue pour $n = 64$ disques.

Le nombre d'opérations fondamentales est de $2^{65} - 1 \approx 36 \cdot 10^{18}$.
Si nous supposons un disque par an, nous devons attendre approximativement 10^{15} siècles!!!

Prenons une hypothèse d'un déplacement d'un disque par seconde (*moine turbo*!!!). Pour chaque siècle $T = 3600 \cdot 24 \cdot 365 \cdot 100 \approx 3.15 \cdot 10^9$. La fin du monde serait donc prévu : $\frac{2^{n+1}-1}{T}$, i.e. approx. dans $11 \cdot 10^9$ siècles.

Exercice 3 : **Recherche Dichotomique et variantes**

En supposant une liste A triée d'éléments, on peut comparer le milieu de la séquence avec la valeur v recherchée, et supprimer la moitié de la séquence de la suite des opérations. La recherche dichotomique est un algorithme qui répète cette procédure, en divisant par deux à chaque fois la taille de la partie restante de la liste.

1. Écrire le pseudo code (itératif ou récursif) de la recherche dichotomique. Expliquez pourquoi le temps d'exécution de la recherche dichotomique dans le cas le plus défavorable est $\Theta(\log_2(n))$?

   Question facultative : temps d'exécution de la recherche dans le cas favorable ?
2. Déterminer la complexité temporelle pour la recherche trichotomique (algorithme identique à la recherche dichotomique mais avec une décomposition en 3 partitions correspondant à trois sous-listes).
3. Effectuer la même étude mais en supposant k partitions ?
4. On souhaite une performance de la recherche " k-tomique " au moins c fois plus rapide que pour une recherche dichotomique simple. Déterminer la valeur de k pour des valeurs de $c = 2$ et $c = 10$?

question 1. Écrire le pseudo code (itératif ou récursif) de la recherche dichotomique. Expliquez pourquoi le temps d'exécution de la recherche dichotomique dans le cas le plus défavorable est $\Theta(\log_2(n))$?

Question facultative : temps d'exécution de la recherche dans le cas favorable ?

Algorithme 14 : Dicho(A,g,d,val)

```
Entrées : A : tableau[1.. MAX] d'entier ; g,d,val : entier ;
Sorties : booleen ;
Données : pivot : entier ;
14.1  début
14.2      si (g < d) alors
14.3          pivot ← ⌊(g+d)/2⌋ ;
14.4          si (val = A[pivot]) alors
14.5              retourner VRAI ;
14.6          sinon
14.7              si (val < A[pivot]) alors
14.8                  retourner Dicho(A,g,pivot-1,val) ;
14.9              sinon
14.10                 retourner Dicho(A,pivot+1,d,val) ;
14.11     sinon
14.12         retourner FAUX ;
```

La relation de récurrence est immédiate : $T(n) = T(\frac{n}{2}) + \Theta(1)$. Nous pouvons montrer la conjecture $T(n) = \Theta(\log_2(n))$ est vérifiée. Notons que l'application du théorème de la méthode générale donne également une complexité temporelle $T(n) = \Theta(\log_2(n))$.

question 2. Déterminer la complexité temporelle pour la recherche trichotomique (algorithme identique à la recherche dichotomique mais avec une décomposition en 3 partitions correspondant à trois sous-listes).

Algorithme 15 : Tricho(A,g,d,val)

```
          Entrées : A : tableau[1..MAX] d'entier ; g,d,val : entier ;
          Sorties : booleen ;
          Données : pivot1, pivot2 : entier ;
15.1      début
15.2          si (g < d) alors
15.3              pivot1 ← ⌊(g+d)/3⌋ ; pivot2 ← ⌊(pivot1+d)/3⌋ ;
15.4              si (val = A[pivot1]) ou (val = A[pivot2]) alors
15.5                  retourner VRAI ;
15.6              sinon
15.7                  si (val < A[pivot1]) alors
15.8                      retourner Tricho(A,g,pivot1-1,val) ;
15.9                  sinon
15.10                     si (val > A[pivot2]) alors
15.11                         retourner Tricho(A,pivot2+1,d,val) ;
15.12                     sinon
15.13                         retourner Tricho(A,pivot1+1,pivot2-1,val) ;
15.14         sinon
15.15             retourner FAUX ;
```

$T(n) = T(\frac{n}{3}) + \Theta(1)$: Il est alors possible de montrer la conjecture $T(n) = \Theta(\log_3(n))$.

question 3 Effectuer la même étude mais en supposant k partitions ?

$T(n,k) = T(\frac{n}{k}) + \Theta(1)$: Nous pouvons en déduire que pour k partitions, $T(n,k) = \Theta(\log_k(n))$.

Notons que l'algorithme est plus compliqué à implémenter. Nous n'avons pas eu besoin de le décrire en pseudo-code pour déterminer sa complexité.

question 4. On souhaite une performance de la recherche " k-tomique " au moins c fois plus rapide que pour une recherche dichotomique simple. Déterminer la valeur de k pour des valeurs de $c = 2$ et $c = 10$?

$$\begin{array}{ll} c \cdot T(n,k) & \leq T_{Dicho}(n) \\ c \cdot \log_k(n) & \leq \log_2(n) \\ c \cdot \frac{\log_{10}(n)}{\log_{10}(k)} & \leq \frac{\log_{10}(n)}{\log_{10}(2)} \\ c \cdot \log_{10}(2) & \leq \log_{10}(k) \\ \log_{10}(2^c) & \leq \log_{10}(k) \\ 2^c & \leq k \end{array}$$

Si $c = 2$, $k \geq 4$ (de même pour $c = 10$, $k \geq 1024$).

Essayons de proposer une interprétation du résultat :

Algorithme	Nbre Oper. Fondam.
$T_{Dicho}(n)$	29.9
$T(n, 1024)$	2.9

Par hypothèse, considérons un ordinateur qui effectue 10^6 opérations à la seconde. Nous comparons deux algorithmes qui s'exécutent en 29.9 à 2.9 μ-seconde (pour $n = 10^6$). Malgré une augmentation de la performance théorique de l'algorithme, la différence en pratique ne sera pas visible. Il y a donc aucun intérêt à proposer des algorithmes k-tomiques.

Essayons d'appliquer le théorème de la méthode générale

Rappel du cours : Soient les constantes $a \geq 1$, $b > 1$ et une fonction $f(n)$. Soit $T(n)$ défini par la récurrence $T(n) = a \cdot T(\frac{n}{b}) + f(n)$. $T(n)$ peut alors être borné asymptotiquement de la façon suivante :

1. Si $f(n) = O(n^{\log_b(a)-\epsilon})$ pour une constante $\epsilon > 0$ alors $T(n) = \Theta(n^{\log_b(a)})$
2. Si $f(n) = \Theta(n^{\log_b(a)})$ alors $T(n) = \Theta(n^{\log_b(a)} \cdot \log_2(n))$
3. Si $f(n) = \Omega(n^{\log_b(a)+\epsilon})$ pour une constante $\epsilon > 0$ et $a \cdot f(\frac{n}{b}) \leq c \cdot f(n)$ pour une constante $c < 1$ et n *grand* alors $T(n) = \Theta(f(n))$

En effet, l'application du cas 2 du théorème sur la relation de récurrence $T(n, k) = T(\frac{n}{k}) + \Theta(1)$: $a = 1$, $b = k$ et $f(n) = 1$. En effet, la comparaison entre les grandeurs $n^{\log_b(a)}$ et $f(n)$ sont égales, ie $\Theta(n^{\log_b(a)}) = \Theta(n^{\log_k(1)}) = f(n)$. Notons ainsi que le théorème de la méthode générale conclut pour toutes les valeurs de k à $T(k, n) = \log_2(n)$.

Ce résultat n'est pas ce que nous avons conjecturé initialement ? Pourquoi ?