Faculté des Sciences
Aix*Marseille Université

Mise à Niveau
Etude expérimentale de différents algorithmes de tris

ALLAM Thomas, BERNARDINI Mickael
L3 Informatique – Groupe 1

I. Algorithmes et complexités

Tri par casier

```
int getMax(int* array, int arraysize) {
     int max = 0;
     int i;
     for (i = 0; i < arraysize; ++i)
               if (array[i] > max)
                         max = array[i];
     return max;
}
```

Dans un premier temps l'algorithme du tri par casier parcourt le tableau à trier, à la recherche du plus grand élément. Soit une complexité en O(n).

```
void counting_sort(int *array, int taille, int min, int max)
{
     int i, j, z;
     int range = max - min + 1;
     int *count = malloc(range * sizeof(*array));

     for(i = 0; i < range; i++) count[i] = 0;
     for(i = 0; i < taille; i++) count[ array[i] - min ]++;

     for(i = min, z = 0; i <= max; i++) {
          for(j = 0; j < count[i - min]; j++) {
               array[z++] = i;
          }
     }
     free(count);
}
```

Après avoir trouvé l'élément maximal du tableau à trier, il faut construire un second tableau dont la taille correspond à la valeur maximal trouvé précédemment puis initialiser chaque case à 0. Soit une complexité en O(max) où max est la valeur la plus grande.

Ensuite on parcourt le tableau à trier en incrémentant pour chaque valeur la case dans le deuxième tableau, la valeur du premier tableau correspond à un indice dans le second tableau. Soit une complexité en O(n).

Pour finir on parcourt les différentes cases du tableau et on reconstruit le tableau initial en le classant. Soit une complexité en O(max).

Au final on a une complexité en nombre d'opérations qui est en O(max + n). La complexité est linéaire et dépend de la valeur max.

Tri par arbre binaire de recherche

```
void tri_arbre_binaire_de_recherche(int* array, int arraysize)
{
     struct Noeud *tree;
     int i;
     for (i = 0; i < arraysize; ++i)
          insertion(&tree, array[i]);
}
```

Dans un premier temps l'algorithme du tri par arbre binaire de recherche déclare un arbre vide puis parcourt l'ensemble des n valeurs du tableau à trier et les insèrent dans l'arbre. Soit une complexité en O(n).

```
void insertion(struct Noeud **noeud, int m){
     if (*noeud == NULL)
     {
          *noeud = malloc(sizeof(struct Noeud));
          (*noeud) -> donnee = m;
          (*noeud) -> fils_G = NULL;
          (*noeud) -> fils_D = NULL;
     }
     else
     {
          if (m < (*noeud) -> donnee)
               insertion(&((*noeud) -> fils_G), m);
          else
               insertion(&((*noeud) -> fils_D), m);
     }
}
```

Ensuite l'arbre est initialisé avec la première valeur du tableau à trier, qui correspond à la racine, puis les éléments suivants sont insérés soit du côté gauche de l'arbre soit du côté droit en appel récursif. L'insertion des éléments d'un côté ou de l'autre dépend de grosseur de l'élément par rapport à la racine. Soit une complexité en O(n.log(n)).

Au final on a une complexité en O(n.log(n)). La complexité dépend de la taille du tableau à trier cependant la construction d'un arbre binaire de recherche nécessite un espace mémoire important.

Tri par insertion séquentielle dans une liste chainée

```
void tri_insertion_sequentielle_liste_chainee(int* array, int
arraysize)
{
	struct Chainon *liste = NULL;
	int i;
	for (i = 0; i < arraysize; ++i)
	{
		liste = insertion_dans_liste(liste, array[i]);
	}
	for (i = 0; liste != NULL; ++i)
	{
		array[i] = liste -> donnee;
		liste = liste -> p_suivant;
	}
	liberer(&liste);
}
```

Dans un premier temps l'algorithme du tri par insertion séquentielle dans une liste chainée crée un premier chainon vide, qui représente la liste chainée dans son état initial, puis insère un à un les éléments du tableau à trier. Soit une complexité en O(n).

```
struct Chainon *insertion_dans_liste(Chainon *liste, int
valeur) {
	struct Chainon *p_nouvelle = malloc(sizeof *liste);
	if (p_nouvelle == NULL)
		exit(EXIT_FAILURE);
	else {
		struct Chainon *p_tmp = NULL;
		struct Chainon *p_liste = liste;
		p_nouvelle -> donnee = valeur;
		while(p_liste != NULL && p_liste -> donnee <= valeur)
		{
			p_tmp = p_liste;
			p_liste = p_liste -> p_suivant;
		}
		p_nouvelle -> p_suivant =  p_liste;
		if (p_tmp != NULL)
			p_tmp -> p_suivant = p_nouvelle;
		else
			liste = p_nouvelle;
	}
	return liste;
}
```

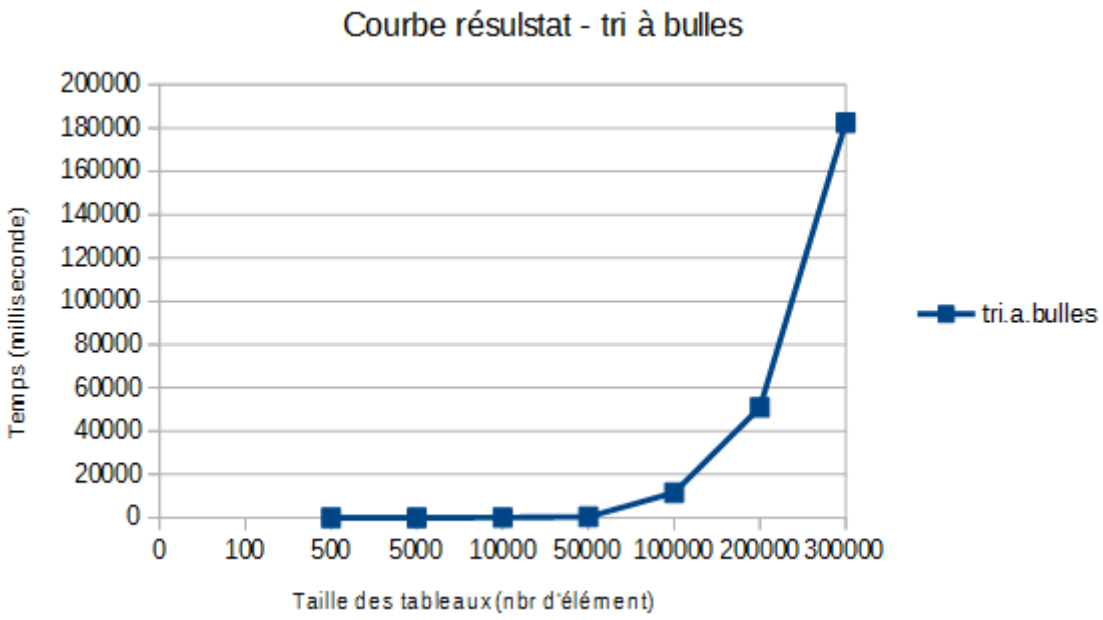
Les éléments sont insérés un a un directement à la bonne place : si l'élément courant est plus grand que le précédent alors on crée un nouveau en fin de liste puis on l'insère, au contraire si l'élément est plus petit on décale tous les chainons de une position puis on insère l'élément a la bonne place. Soit une complexité en O(n).

Au final on a une complexité en $O(n^2)$. La complexité dépend de la taille du tableau en entré ainsi que de l'ordre dans lequel sont initialement rangé les éléments.

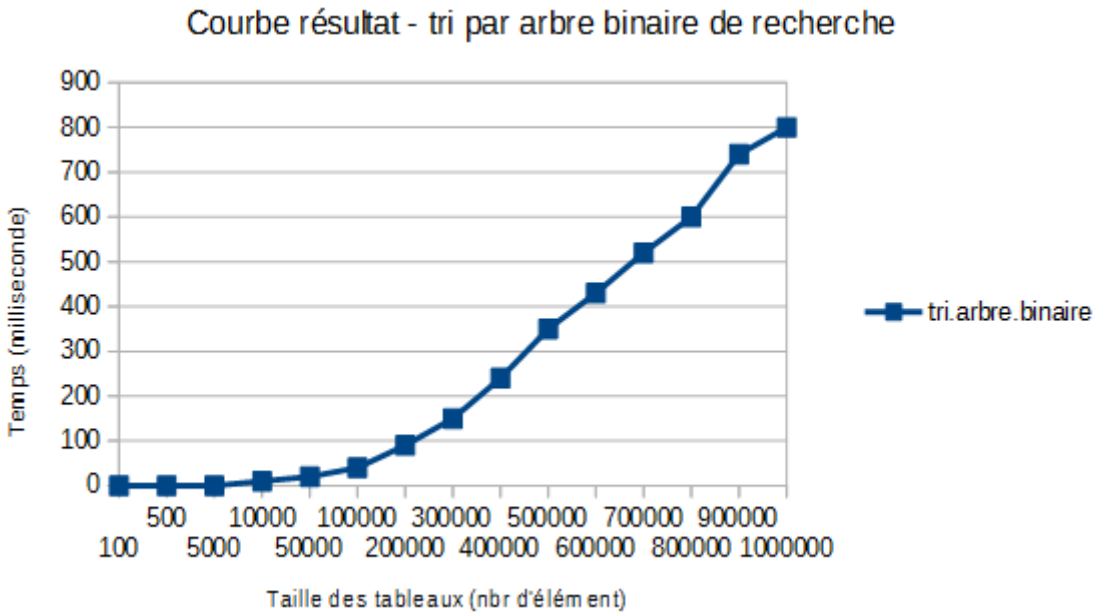
II. Exploitation des résultats

Tous les tests ont été réalisés 20 fois sur différentes tailles de tableau

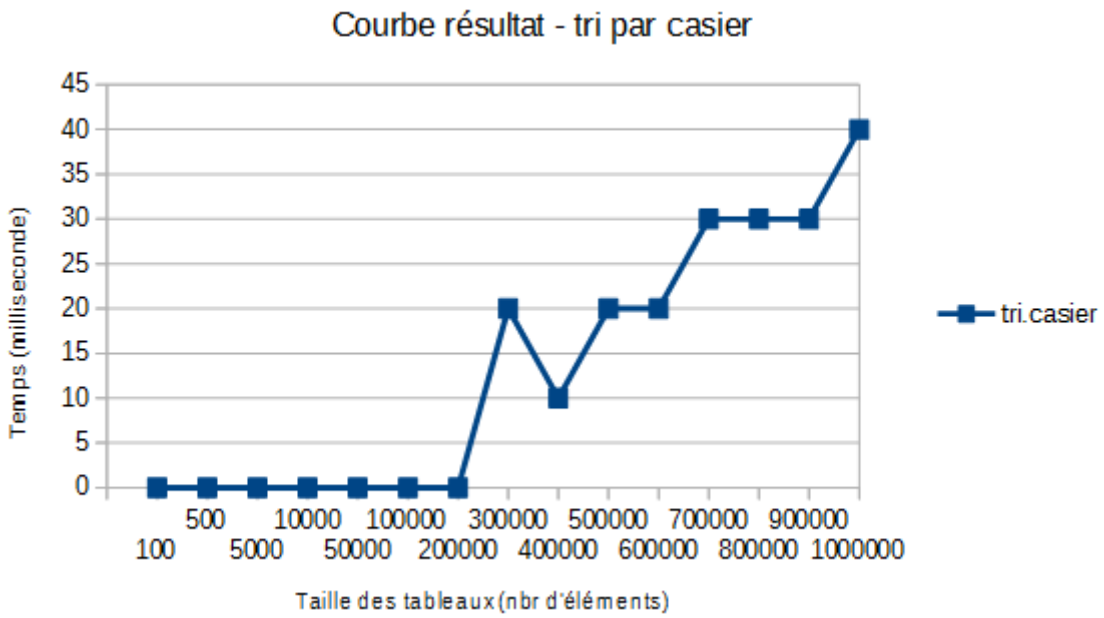
Tri à bulles



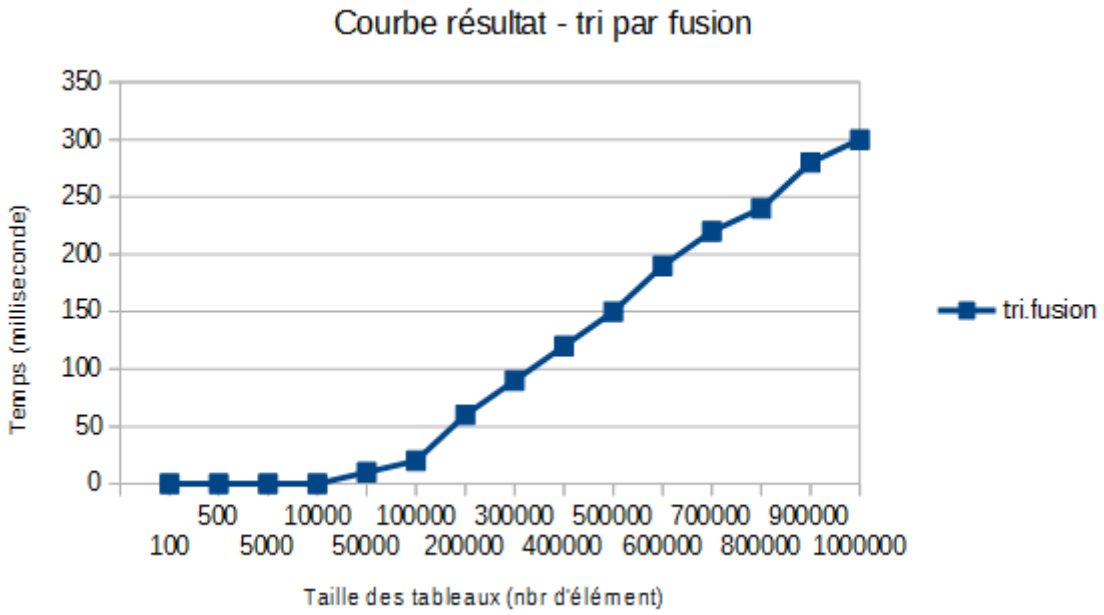
Tri par arbre binaire de recherche



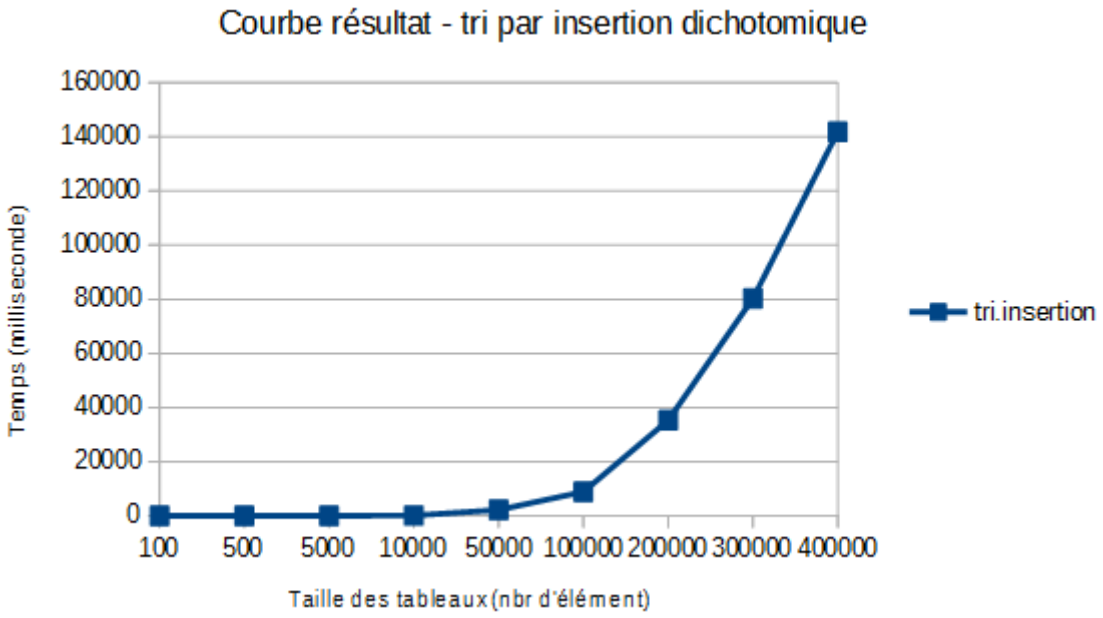
Tri par casier



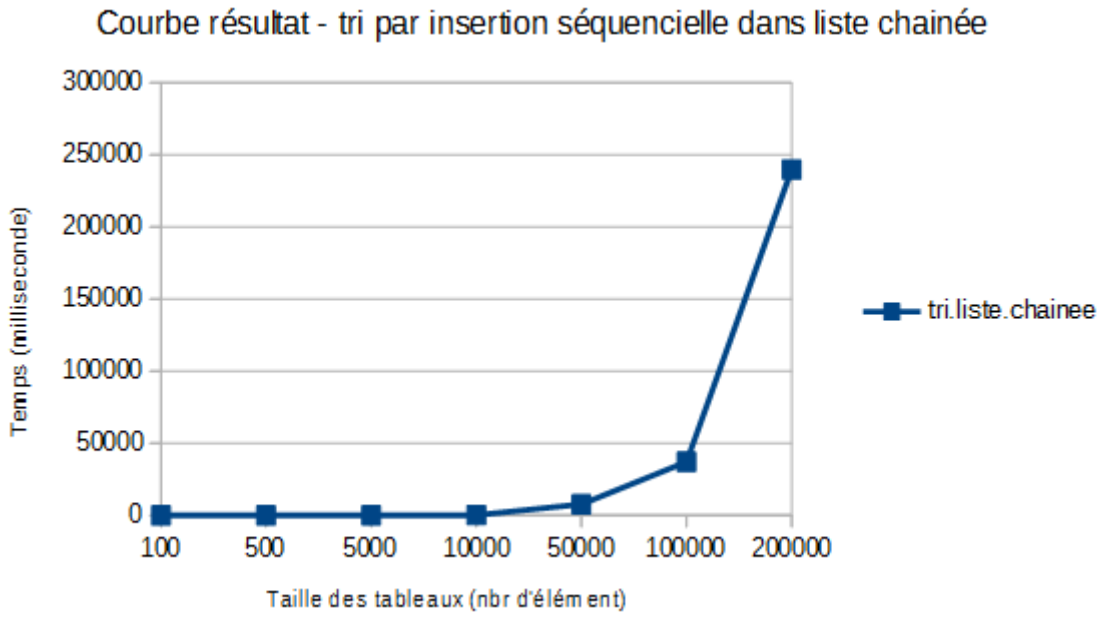
Tri par fusion



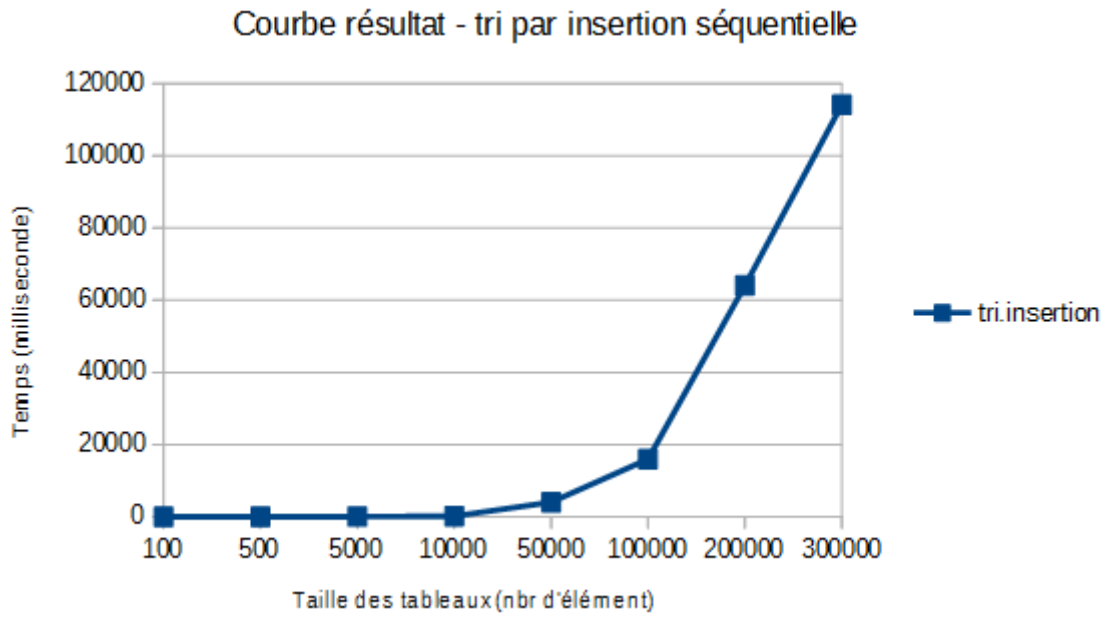
Tri par insertion dichotomique



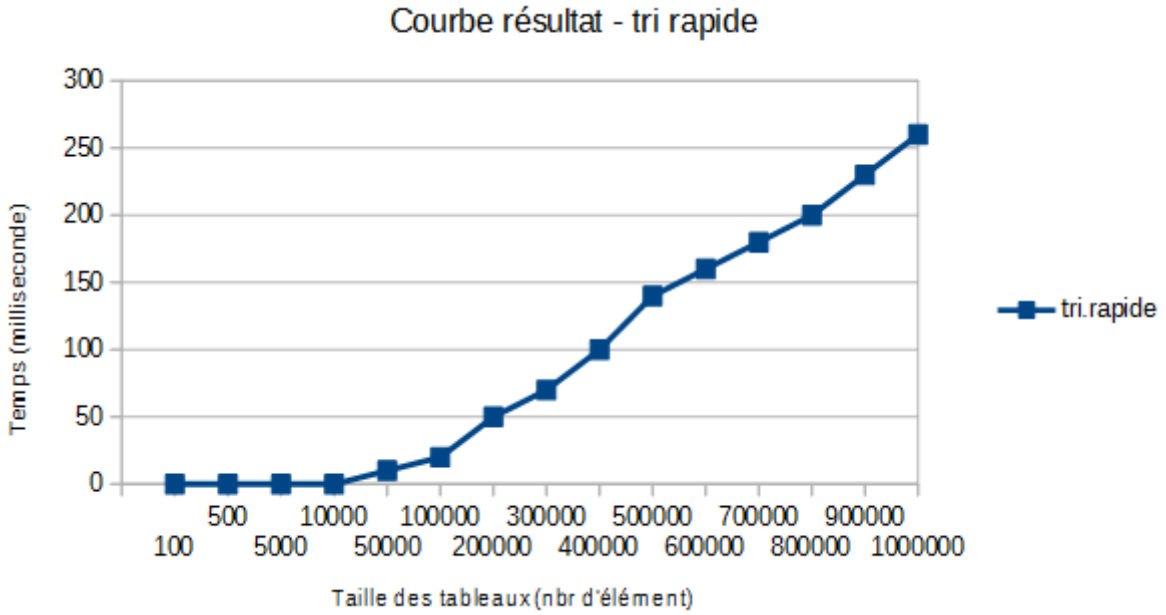
Tri par fusion



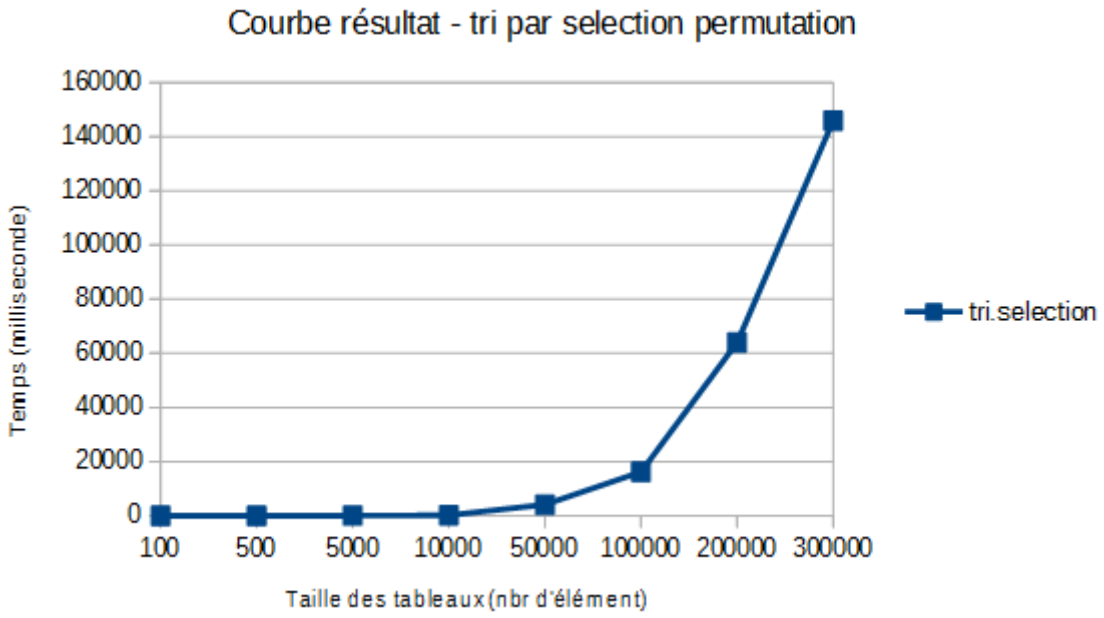
Tri par insertion séquentielle



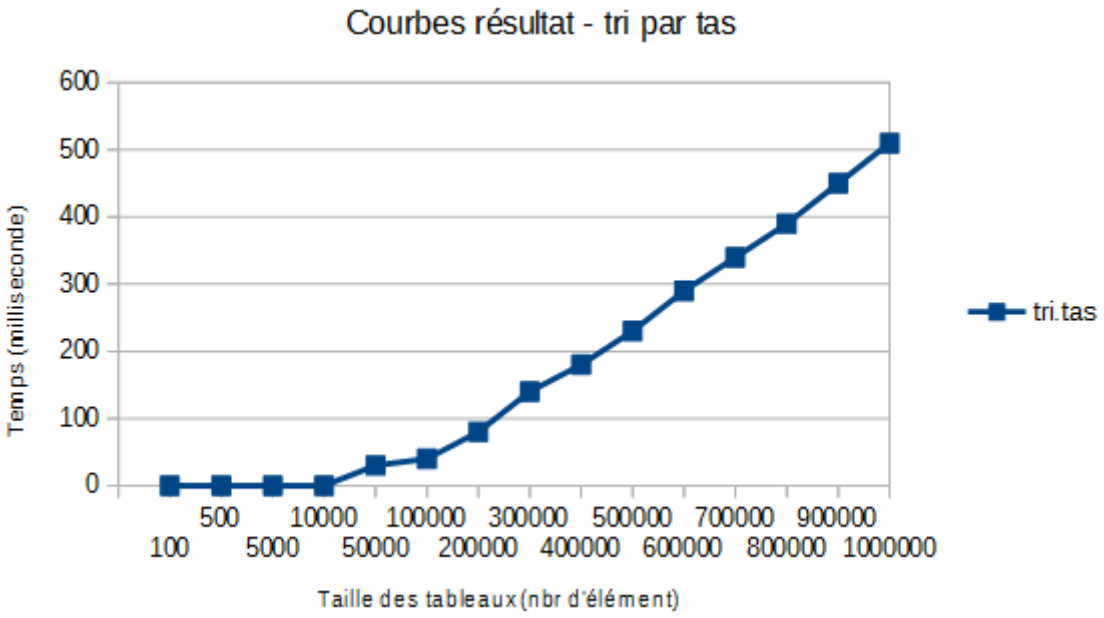
Tri rapide



Tri par sélection permutation



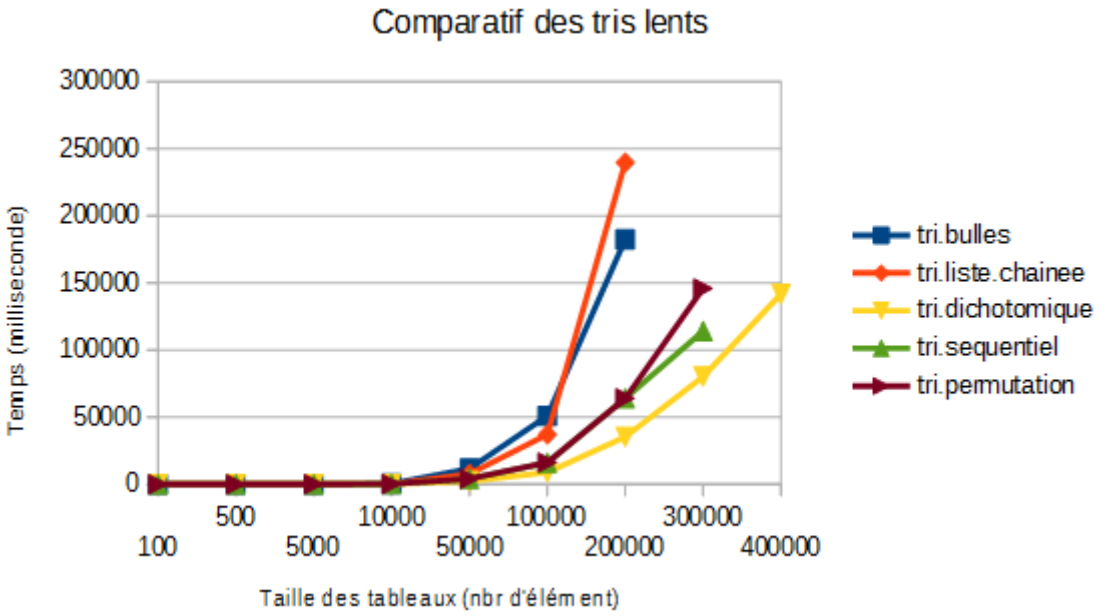
Tri par tas



III. Comparatifs

Tris lents

	100	500	5000	10000	50000	100000	200000	300000	400000
Tri à bulles	0	0	110	450	11630	50950	182470		
Tri par liste chainée	0	0	50	170	7690	37080	239750		
Tri insertion dichotomique	0	0	20	90	2190	8870	35300	80280	141850
Tri insertion séquentielle	0	0	40	160	4020	15960	64060	114190	
Tri sélection permutation	0	0	40	170	4100	16210	64010	145920	

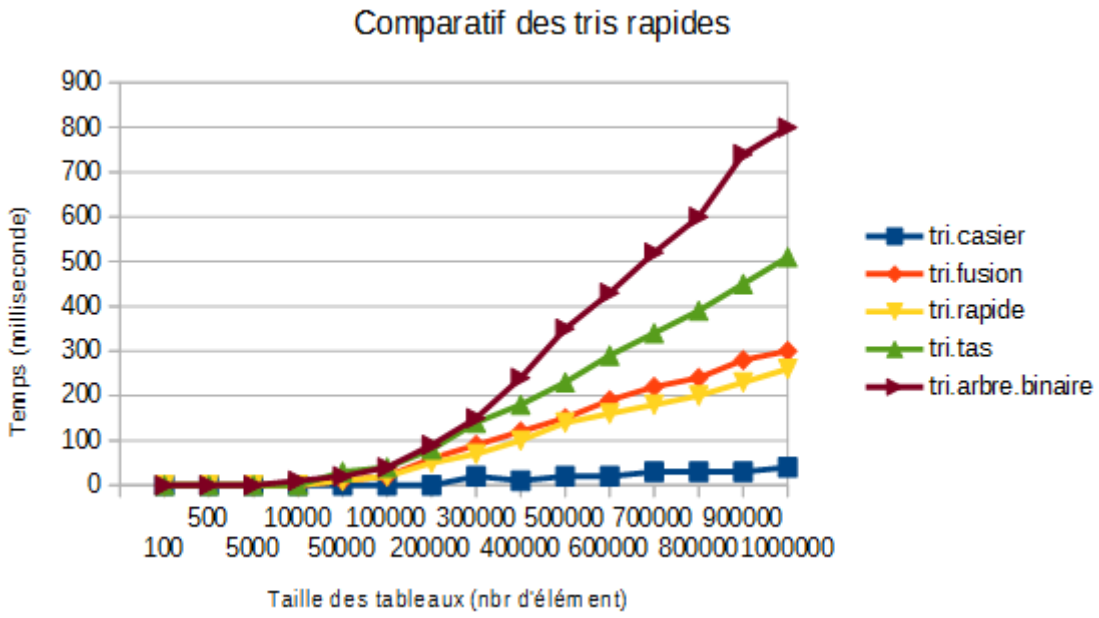


Les tris dits lents sont : le tri à bulles, le tri par insertion séquentielle dans liste chainée, le tri par insertion dichotomique, le tri par insertion séquentielle et le tri par sélection permutation.
Ces tris n'ont pas réussi à réaliser tous les tests dans un intervalle de 5 minutes.
Comme on peut le voir sur les résultats obtenue : le tri le plus rapide ici est le tri par insertion dichotomique qui réalisé les même tris plus rapidement que les autres et a également réalisé un tri en plus.
Par opposition on peut voir ici que le tri le plus lent est le tri par insertion séquentielle dans liste chainée.

Tris rapides

	100	500	5000	10000	50000	100000	200000	300000	400000
Tri casier	0	0	0	0	0	0	0	20	10
Tri fusion	0	0	0	0	10	20	60	90	120
Tri rapide	0	0	0	0	10	20	50	70	100
Tri par tas	0	0	0	0	30	40	80	140	180
Tri arbre	0	0	0	10	20	40	90	150	240

500000	600000	700000	800000	900000	1000000
20	20	30	30	30	40
150	190	220	240	280	300
140	160	180	200	230	260
230	290	340	390	450	510
350	430	520	600	740	800

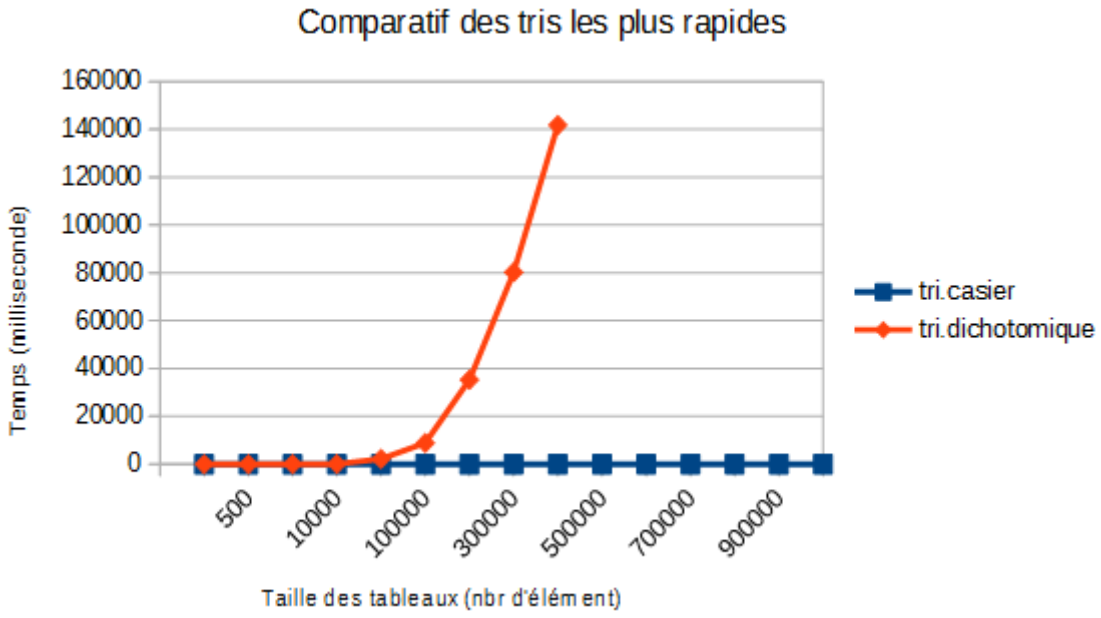


Les tris dits rapides sont : le tri par casier, le tri fusion, le tri rapide, le tri par tas et le tri par arbre binaire de recherche.
Ces tris ont réussi à réaliser tous les tests dans un intervalle inférieur à 5 minutes.
Comme on peut le voir sur les résultats obtenue : le tri le plus rapide ici est le tri par casier qui a réalisé tous ses tris en moins de 100 millisecondes
Par opposition on peut voir ici que le tri le plus lent est le tri par arbre binaire de recherche.

Comparaison des tris les plus rapide

	100	500	5000	10000	50000	100000	200000	300000	400000
Tri casier	0	0	0	0	0	0	0	20	10
Tri insertion dichotomique	0	0	20	90	2190	8870	35300	80280	141850

500000	600000	700000	800000	900000	1000000
20	20	30	30	30	40



Comme on peut le voir ici, le tri par casier réalise tous ses tris en un temps instantané par rapport au tri par insertion dichotomique.
Cela est dû à leurs différentes implémentations et donc complexités. En effet le tri par insertion dichotomique a une complexité en $O(n^2)$ contrairement au tri par casier qui est en $O(m + n)$ ou encore que le tri rapide en $O(n.log(n))$.
Cependant le tri par insertion dichotomique reste très efficace sur des entrées de petite taille ou lorsque le tableau est presque triée ($O(n)$ affectation).

IV. Conclusion

Grace au programme de test réalisé, nous avons pu évaluer la complexité théorique et pratique de dix algorithmes de tri. Ces tests ont prouvés que ces deux complexités sont liées. En effet un algorithme a une complexité théorique élevé plus il sera lent à l'exécution.