

# TP6: Tri rapide en place

## Étude théorique

Algorithme de partition entre deux indices :

---

**Algorithme 1** : `partition_entre( $T, n, a, b$ )`

---

**Entrée(s)** :  $T$  tableau de taille  $n$ ,  $a, b \in \llbracket 0, n-1 \rrbracket$  avec  $a \leq b$

**Sortie(s)** :  $j \in \llbracket a, b \rrbracket$  tel que l'ancienne valeur de  $T[a]$  est maintenant à l'indice  $j$ , et  $\forall x \in T[a..j], x \leq T[j]$  et  $\forall x \in T[j..b], x > T[j]$

```

1  $i \leftarrow a + 1$ ;
2  $s \leftarrow b$ ;
  // Invariant I:  $\forall k \in \llbracket a, i-1 \rrbracket, T[k] \leq T[a]$ 
  // Invariant S:  $\forall k \in \llbracket s+1, b \rrbracket, T[k] > T[a]$ 
3 tant que  $i \leq s$  faire
4   si  $T[i] \leq T[a]$  alors
5      $i \leftarrow i + 1$ ;
6   sinon
7     Échanger  $T[i]$  et  $T[s]$ ;
8      $s \leftarrow s - 1$ ;
9 Échanger  $T[a]$  et  $T[i-1]$ ;
10 retourner  $i-1$ 
```

---

(Complexité :  $\mathcal{O}(b-a)$ ).

Algorithme de tri rapide :

---

**Algorithme 2** : `tri_rapide_entre( $T, n, a, b$ )`

---

**Entrée(s)** :  $T$  tableau de taille  $n$ ,  $a, b \in \llbracket 0, n-1 \rrbracket$  avec  $a \leq b$

**Sortie(s)** :  $T$  est modifié de sorte que  $T[a..b]$  est trié.

```

1 si  $b - a \leq 0$  alors
  // Tableau à 0 ou 1 éléments
2   retourner ;
3  $p \leftarrow \text{partition\_entre}(T, n, a, b)$ ;
4  $\text{tri\_rapide\_entre}(T, n, a, p-1)$ ;
5  $\text{tri\_rapide\_entre}(T, n, p+1, b)$ ;
```

---

Pour trier un tableau en entier, il suffit d'appeler l'algorithme précédent sur tout l'intervalle des indices :

---

**Algorithme 3** : `tri_rapide( $T, n$ )`

---

**Entrée(s)** :  $T$  tableau de taille  $n$

**Sortie(s)** :  $T$  est trié.

```

1  $\text{tri\_rapide\_entre}(T, n, 0, n-1)$ ;
```

---

Notons que chaque appel récursif prend un espace mémoire constant dans la pile d'appel. La hauteur maximale de la pile d'appel sera donc proportionnelle à la profondeur maximale de récursion. Par exemple, pour un tableau où le pivot choisi coupe systématiquement en deux parties parfaitement égales, l'arbre d'appel sera équilibré, et aura au plus  $\log_2(n) + 1$  appels de fonction imbriqués. La taille de la pile est donc au plus en  $\mathcal{O}(\log n)$

En revanche, pour un tableau de la forme  $[1, 2, \dots, n]$ , où le pivot coupe en deux parties très inégales, nous avons vu que l'on a  $n$  appels qui s'empilent les uns sur les autres pour trier des tableaux de tailles  $n, n-1, \dots, 1$ . Dans ce cas, la taille de la pile est en  $\mathcal{O}(n)$ .

## Implémentation en C et chronométrage

Lorsque l'on implémente des algorithmes de tri, on peut toujours commencer par faire une fonction d'échange, ce qui évite d'avoir à faire la manœuvre d'échange à chaque fois :

```
1 /* Échange T[i] et T[j] */
2 void swap(int* T, int i, int j){
3     int tmp = T[i];
4     T[i] = T[j];
5     T[j] = tmp;
6 }
```

Fonction qui vérifie si un tableau est trié dans l'ordre croissant :

```
1 /* Renvoie true si T est trié, false sinon.
2    n est la taille de T */
3 bool is_sorted(int* T, int n){
4     for (int i = 0; i < n-1; ++i){
5         if(T[i] < T[i+1]){
6             return false;
7         }
8     }
9     return true;
10 }
```

Implémentation du tri par insertion :

```
1 /* Trie T, tableau de taille n */
2 void tri_insertion(int* T, int n){
3     for (int i = 0; i < n; ++i){
4         for (int j = i; j > 0 && T[j] < T[j-1]; j--){
5             swap(T, j, j-1);
6         }
7     }
8 }
```

Pour le tri rapide, on implémente les trois algorithmes étudiés dans la première partie du TP :

```
1 /* Partitionne T[a..b] selon T[a]
2    Renvoie un indice p tel que:
3    - T[p] contient l'ancienne valeur de T[a]
4    - Toutes les valeurs <= T[p] sont entre a et p-1
5    - Toutes les valeurs > T[p] sont entre p+1 et b
6    */
7 int partition(int* T, int a, int b){
8     assert(b-a+1>=2);
9     int pivot = T[a];
10    int i = a+1;
11    int s = b;
12    while(i<=s){
```

```

13     if (T[i] > pivot){
14         swap(T, i, s);
15         s--;
16     } else {
17         i++;
18     }
19 }
20 swap(T, i-1, a);
21 return i-1;
22 }
23
24 /* Trie T[a..b] */
25 void tri_rapide_entre(int* T, int a, int b){
26     if (b-a <= 0){
27         return;
28     }
29     int p = partition(T, a, b);
30     tri_rapide_entre(T, a, p-1);
31     tri_rapide_entre(T, p+1, b);
32 }
33
34 /* Trie T, tableau de taille n */
35 void tri_rapide(int* T, int n){
36     tri_rapide_entre(T, 0, n-1);
37 }

```

Comparons les performances sur des tableaux aléatoires. Voici une fonction remplissant un tableau avec des valeurs aléatoires :

```

1  /* TESTS SUR DES TABLEAUX ALEATOIRES */
2  /* Initialise les n premières cases de T en y écrivant
3   * des entiers aléatoires entre 0 et v_max inclus */
4  void random_tab(int* T, int n, int v_max){
5      for (int i = 0; i < n; ++i){
6          T[i] = rand()%(v_max+1);
7      }
8  }

```

On peut alors tester les deux tris et chronométrer le temps comme demandé dans l'énoncé :

```

1  /* Trie nb_tests tableaux générés aléatoirement de taille n,
2   en utilisant le tri rapide. Renvoie le temps moyen pris. */
3  float test_rand_rapide(int n, int nb_tests){
4      int* T = malloc(n*sizeof(int));
5      clock_t deb = clock();
6      for (int i = 0; i < nb_tests; ++i) {
7          random_tab(T, n, 2*n);
8          tri_rapide(T, n);
9          assert(is_sorted(T, n));
10     }
11     clock_t fin = clock();
12     float total = (float) 1000.0*(fin-deb)/CLOCKS_PER_SEC;
13     return total / nb_tests;
14 }

```

Notons que l'on chronomètre le temps total pris, plutôt que de chronométrer individuellement chaque appel et de sommer les temps. La fonction `clock()` n'est pas infiniment précise, et procéder ainsi permet de limiter les erreurs d'imprécision. On mesure aussi techniquement le temps de génération et de vérification, mais ils sont négligeables quand  $n$  grandit.

## ☆ Compétences numériques ☆ : Comparaison des performances

Passons en Python. Voici un code lisant dans deux fichiers les temps pris par le tri rapide et par le tri par insertion :

```
1 import matplotlib.pyplot as plt
2 import math
3
4 def lire_valeurs(fn):
5     """
6     Lit une liste de flottants dans fn et la renvoie
7     """
8     # ouverture en mode lecture
9     f = open(fn, "r")
10    # chaîne de caractère avec tout le contenu du fichier
11    contenu = f.read()
12    # sépare en sous-chaînes selon les espaces et les '\n'
13    nombres = contenu.split()
14    # transforme chaque sous-chaîne en flottant
15    nombres = list(map(float, nombres))
16    # fermeture du fichier
17    f.close()
18    return nombres
19
20
21 def comparer_tris():
22     """
23     Trace les courbes des temps d'exécution des tris rapide et par insertion.
24     """
25     liste_n = list(range(100, 4001, 100)) # de 100 à 4000 par pas de 100
26     valeurs_rapide = lire_valeurs("valeurs/rand_rapide.txt")
27     valeurs_insertion = lire_valeurs("valeurs/rand_insertion.txt")
28
29     plt.plot(liste_n, valeurs_rapide, 'r-', label="Tri rapide")
30     plt.plot(liste_n, valeurs_insertion, 'b-', label="Tri insertion")
31
32     # légendes des courbes / axes
33     plt.xlabel("Taille du tableau")
34     plt.ylabel("Temps (ms)")
35     plt.legend()
36
37     plt.savefig("comparaison_random.png")
```

(N'oubliez pas de légender vos courbes et vos axes!)

Les courbes tracées :

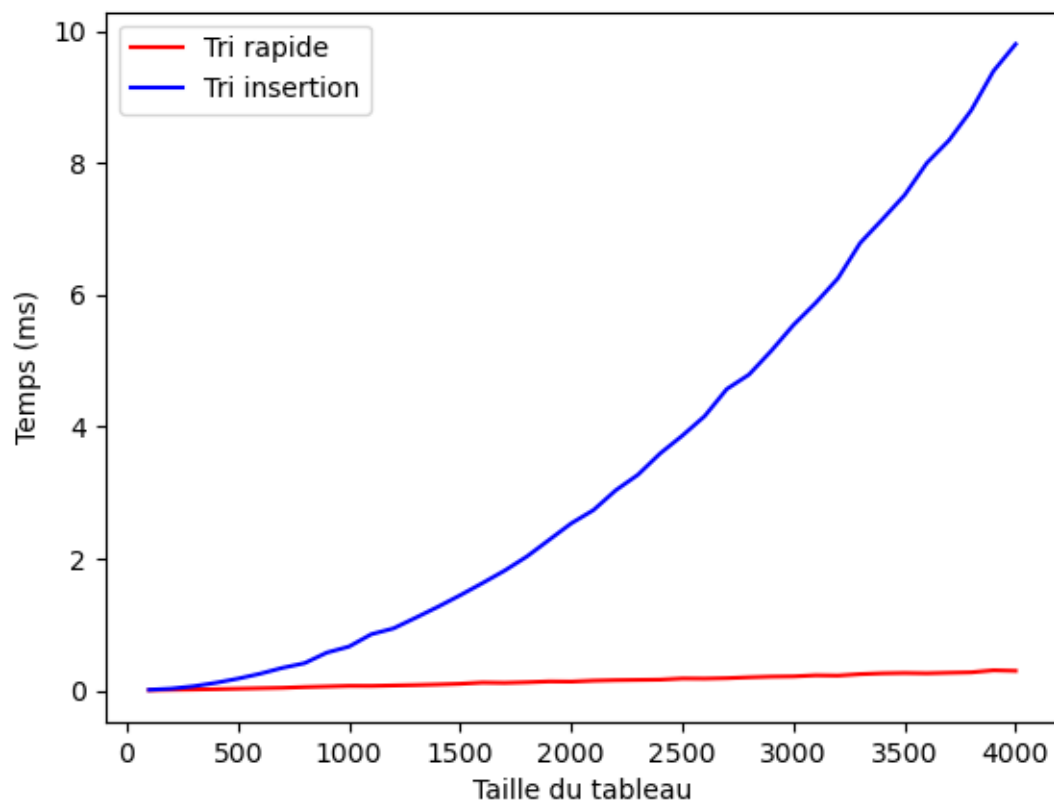


FIGURE 1 – Comparaison des deux tris sur des tableaux aléatoires

Le tri rapide est tellement rapide qu'il semble complètement plat ! Afin de vérifier que les courbes ont l'allure qu'on attend ( $n \log n$  pour le tri rapide,  $n^2$  pour le tri par insertion), on peut diviser les temps obtenus par les fonctions attendues, ce qui donne les courbes suivantes :

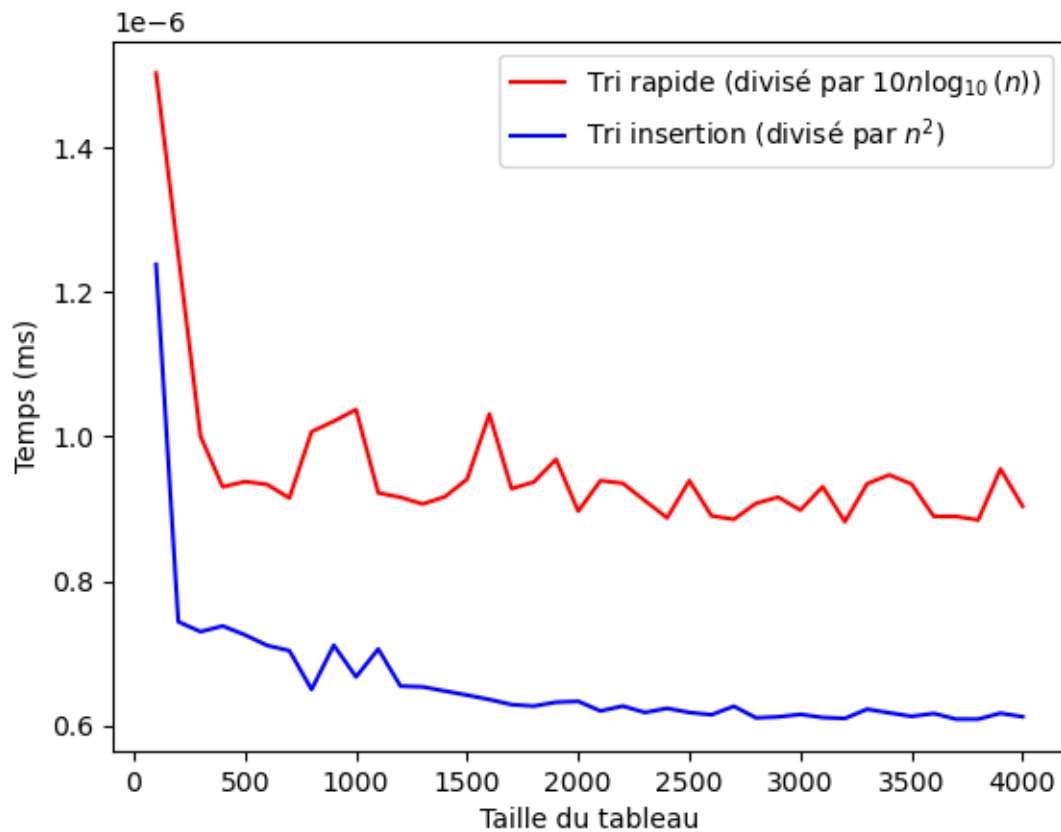


FIGURE 2 – Vérification des allures des complexités

On voit que les deux courbes restent relativement constantes, ce qui confirme que les tris ont bien les complexités attendues. Finalement, le temps d'exécution du tri par insertion semble prendre un temps compris entre  $C_1 n^2$  et  $C_2 n^2$  avec  $C_1 = 6 \times 10^{-10} s$  et  $C_2 = 8 \times 10^{-10} s$ , tandis que le tri rapide semble prendre entre  $C_3 n \log n$  et  $C_4 n \log n$  avec  $C_3 = 8 \times 10^{-9} s$  et  $C_4 = 1.2 \times 10^{-8} s$ .

## Pour aller plus loin : choix du pivot

Pour implémenter le choix de pivot proposé, on peut passer par la fonction suivante :

```

1  /* Renvoie l'indice de la médiane de T[a], T[b] et T[(a+b)/2*/
2  int mediane(int* T, int a, int b){
3      int m = (a+b)/2;
4      if (T[a] <= T[m] && T[m] <= T[b] || T[b] <= T[m] && T[m] <= T[a]){
5          return m;
6      } else if (T[m] <= T[a] && T[a] <= T[b] || T[b] <= T[a] && T[a] <= T[m]){
7          return a;
8      } else {
9          return b;
10     }
11 }
```

Une fois que l'on a identifié la médiane, il suffit de la mettre au début de la zone à partitionner, et on peut alors réutiliser le même algorithme qu'avant :

```

1  /* Partitionne T[a..b] selon la valeur médiane
2     entre T[a], T[b] et T[m] où m est le milieu de [a,b] */
3  int partition_médiane(int* T, int a, int b){
4      int med = mediane(T, a,b);
5      // on met la médiane en T[a], puis on peut
6      // réutiliser le même algorithme qu'avant
7      swap(T, med, a);
8      return partition(T, a, b);
9  }
10
11
12 /* Trie T[a..b] */
13 void tri_médiane_entre(int* T, int a, int b){
14     if (b-a <= 0){
15         return;
16     }
17     int p = partition_médiane(T, a, b);
18     tri_médiane_entre(T, a, p-1);
19     tri_médiane_entre(T, p+1, b);
20 }
21
22 /* Trie T, tableau de taille n */
23 void tri_médiane(int* T, int n){
24     tri_médiane_entre(T, 0, n-1);
25 }
```

Il faut bien penser à renommer tous les appels récursifs !

Comparons les performances :

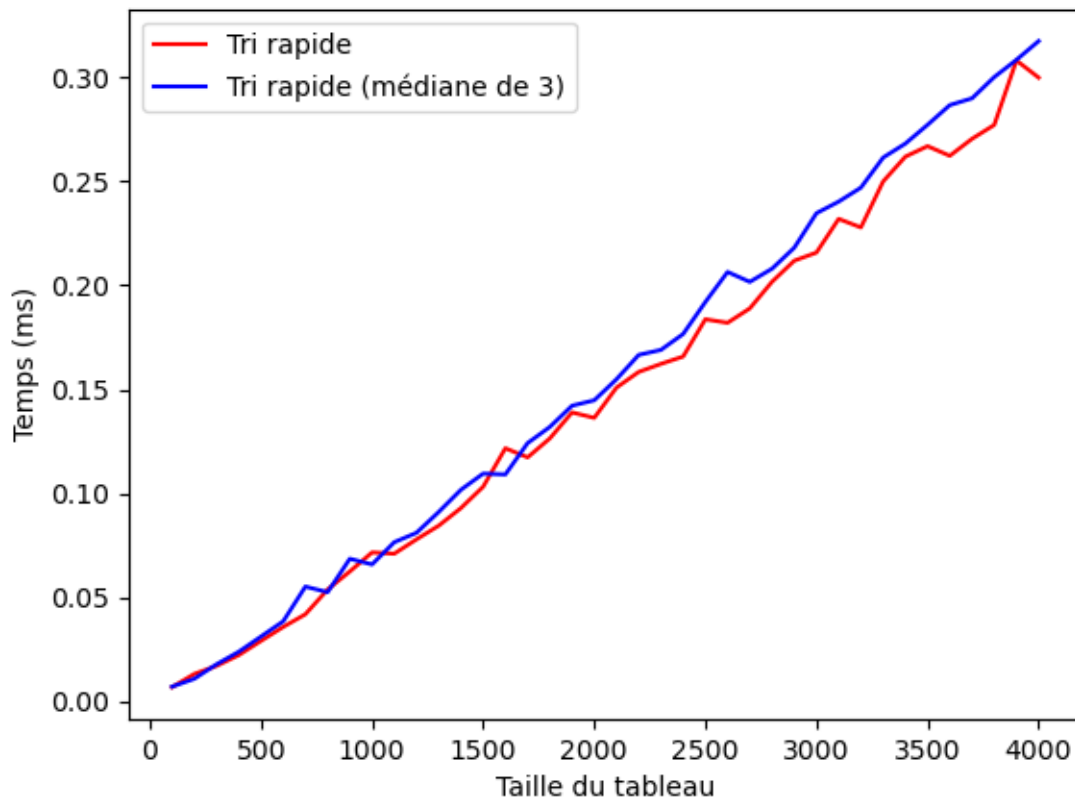


FIGURE 3 – Comparaison du tri rapide classique avec la version utilisant la médiane de trois.

On voit que les résultats ne sont pas du tout meilleurs ! C'est assez logique, car l'objectif de cette méthode de choix de pivot est d'éviter les cas où le premier élément du tableau fait une partition inégale, mais pour un tableau généré aléatoirement, ce cas arrive rarement.

Tentons de comparer les performances sur une autre famille de tableaux, par exemple sur des tableaux décroissants. Nous avons vu que le tri rapide basique prend  $\Theta(n^2)$  sur cette famille de tableaux, mais avec la méthode de choix de pivot proposée, on devrait se ramener à du  $\mathcal{O}(n \log n)$ .

En C :

```

1  /* Initialise les n premières cases de T en y écrivant
2     n-1, n-2, n-3, ..., 1, 0
3  */
4  void décroissant(int* T, int n){
5      for (int i = 0; i < n; ++i){
6          T[i] = n-i-1;
7      }
8  }

```

Puis on réécrit des variantes des fonctions de test qui utilisent cette fonction pour générer les tableaux, par exemple pour le tri rapide :

```

1  float test_decr_rapide(int n, int nb_tests){
2      int* T = malloc(n*sizeof(int));
3      clock_t deb = clock();
4      for (int i = 0; i < nb_tests; ++i) {
5          décroissant(T, n);
6          tri_rapide(T, n);
7          assert(is_sorted(T, n));

```



```
8 | }  
9 | clock_t fin = clock();  
10 | float total = (float) 1000.0f*(fin-deb)/CLOCKS_PER_SEC;  
11 | return total / nb_tests;  
12 | }
```

Voici les courbes obtenues :

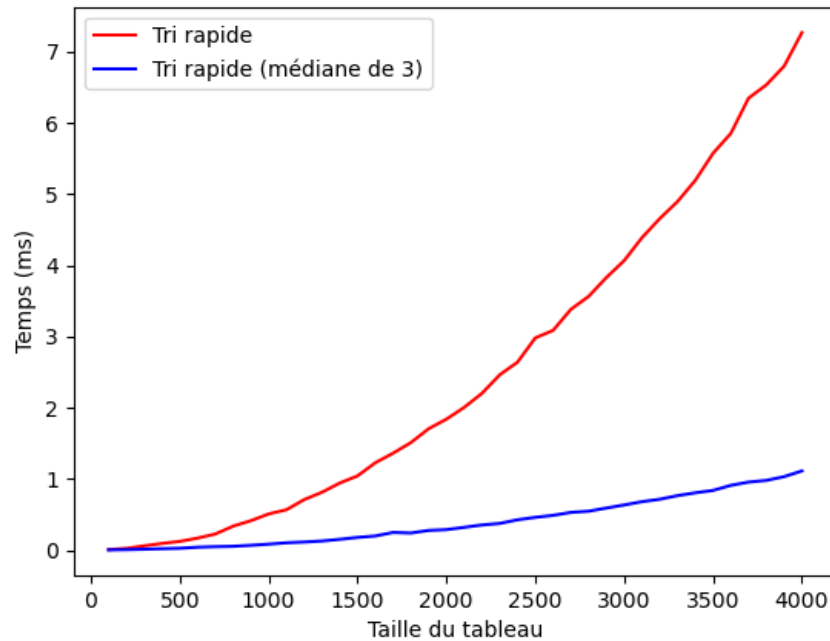


FIGURE 4 – Comparaison du tri rapide classique avec la version utilisant la médiane de trois, sur des tableaux décroissants.

C'est bien meilleur !

## Pour aller encore plus loin : Tri hybride

Tri hybride proposé par le sujet ( $K$  est une variable globale) :

```

1  /* Divise T[a..b] en blocs de tailles <= K, tels que
2     chaque bloc contient globalement les bonnes valeurs
3     pour trier T[a..b], mais pas forcément dans l'ordre.
4     ex: T = [9, 1, 3, 4, 2, 1, 6, 4, 7] et K = 3:
5     après hybrid_semi_sort, T pourra contenir:
6     [1, 2, 1, 3, 4, 4, 9, 6, 7] */
7  void hybrid_semi_sort(int* T, int a, int b){
8      if (b-a+1 <= K){
9          return;
10     }
11     int p = partition_mediane(T, a, b);
12     hybrid_semi_sort(T, a, p-1);
13     hybrid_semi_sort(T, p+1, b);
14 }
15
16 /* Trie T, tableau de taille n */
17 void tri_hybride(int* T, int n){
18     hybrid_semi_sort(T, 0, n-1);
19     tri_insertion(T, n);
20 }
```

La phase de tri par insertion est assez rapide, car elle revient à trier individuellement chacun des blocs. En effet, chaque élément du tableau est déjà dans le bon bloc de taille  $K$  après avoir appliqué le semi-tri rapide. Donc, la complexité de cette deuxième phase est de  $\mathcal{O}(\frac{n}{K}K^2)$  (il y a  $\frac{n}{K}$  blocs, chacun pouvant se trier en  $\mathcal{O}(K^2)$ ). Autrement dit,  $\mathcal{O}(Kn)$ . De plus, si le tableau est quasiment trié en entrée, on peut s'attendre à ce que la phase de tri par insertion soit très rapide, car il n'y aura que peu d'insertions à réaliser.

Passons à l'évaluation. Nous allons tester les performances du tri hybride sur des tableaux déjà triés, où  $\frac{n}{10}$  échanges aléatoires ont été effectuées.

```

1  /* Génère un tableau trié, puis échange n/10 paires de cases au hasard.
2     */
3  void quasi_trie(int* T, int n){
4      for (int i = 0; i < n; ++i){
5          T[i] = i;
6      }
7      for (int k = 0; k < n/10; ++k){
8          int i = rand()%n;
9          int j = rand()%n;
10         swap(T, i, j);
11     }
12 }
```

On peut alors tester le tri hybride avec le même schéma que précédemment :

```

1  float test_trie_hybride(int n, int nb_tests){
2      int* T = malloc(n*sizeof(int));
3      clock_t deb = clock();
4      for (int i = 0; i < nb_tests; ++i) {
5          quasi_trie(T, n);
6          tri_hybride(T, n);
7          assert(is_sorted(T, n));
8      }
9      clock_t fin = clock();
10     float total = (float) 1000.0f*(fin-deb)/CLOCKS_PER_SEC;
11     return total / nb_tests;
12 }
```

Le code suivant lance cette fonction en fixant  $K = 5, 15, 25, \dots, 495$  :

```

1 // tris hybrides sur les tableaux quasi triés
2 for (K = 5; K <= 500; K = K + 10){
3     char nom_fichier[50];
4     sprintf(nom_fichier, "valeurs/trie_hybride_%d.txt", K);
5
6     FILE* f = fopen(nom_fichier, "w");
7     for (int n = 10000; n <= 100000; n = n+10000){
8         float th = test_trie_hybride(n, nb_tests);
9         fprintf(f, "%f\n", th);
10    }
11    printf("Créé: %s\n", nom_fichier);
12    fclose(f);
13 }

```

La fonction `sprintf` est très pratique : elle fonctionne exactement comme `printf` ou `fprintf`, mais écrit le texte généré directement dans une chaîne de caractère. C'est particulièrement utile pour générer des noms de fichiers comme fait ci-dessus.

Traçons quelques unes des courbes, en comparant au tri avec choix du pivot par médiane de trois :

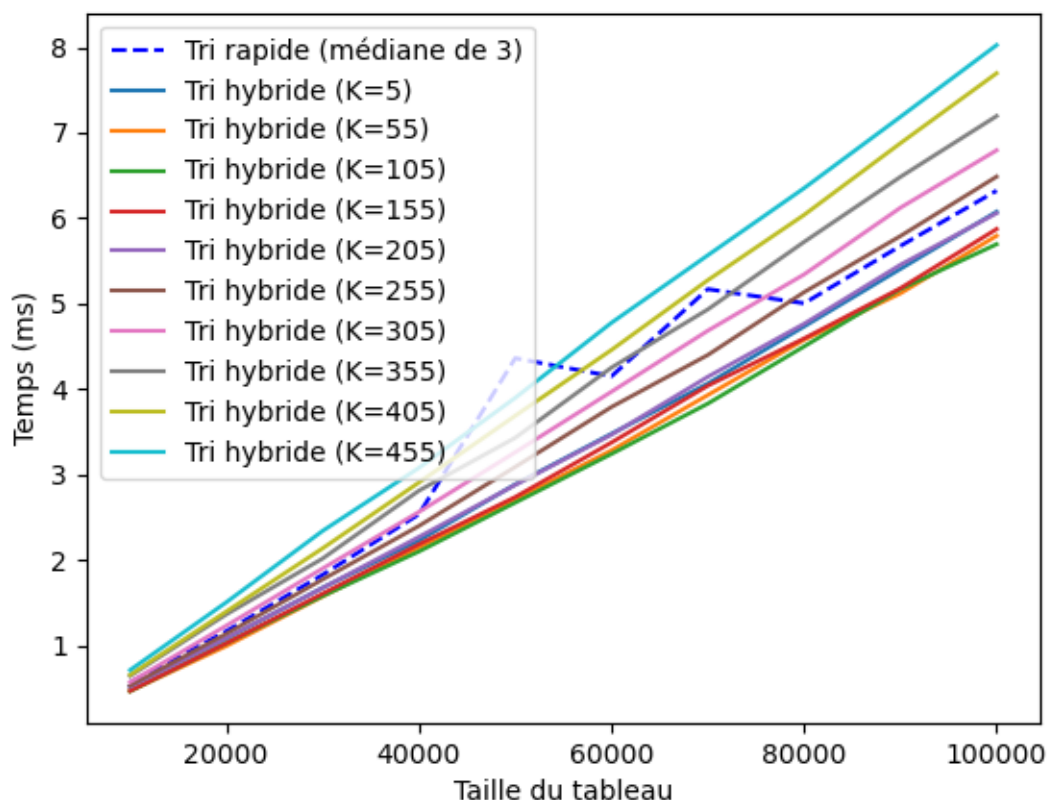


FIGURE 5 – Comparaison de tris hybrides sur des tableaux presque triés.

On voit que pour plusieurs valeurs de  $K$  (entre 55 et 155 sur le graphique), le tri hybride est légèrement plus efficace que le tri rapide simple.

## BONUS : pointeurs de fonctions

Revenons sur le code C qui teste les tris. Nous disposons de plusieurs algorithmes de tris, et de plusieurs méthodes de génération de tableaux. Si l'on veut faire des tests complets, il faut, pour chaque fonction de génération  $X$ , et pour chaque tri  $Y$ , créer une fonction `test_X_Y(int n, int nb_tests)` qui lance les tests correspondants. On aimerait bien pouvoir passer  $X$  et  $Y$  en paramètre afin de ne pas écrire 12 fois la même fonction (si l'on a 4 algos de tri et 3 fonctions de génération).

Une première solution est d'avoir une fonction intermédiaire qui prend en entrée une chaîne de caractères et qui applique l'algorithme de tri associé :

```
1 void trier(int* T, int n, char* algorithme){
2     if (strcmp(algorithme, "insertion") == 0){
3         tri_insertion(T, n);
4     } else if strcmp(algorithme, "rapide") == 0){
5         tri_rapide(T, n);
6     }
7     // etc...
8 }
```

En faisant la même chose avec les fonctions de génération, on arrive à paramétrer la situation, on pourrait écrire une fonction `float tester(char* gen, char* tri, int n, int nb_tests)` qui fait `nb_tests` sur des tableaux de taille `n`, en générant les tableaux selon `gen` et en les triant selon `tri`, qui sont des chaînes de caractères.

On peut même faire mieux ! En C, **une fonction est un pointeur** vers la zone mémoire qui contient son code compilé. On peut donc faire une fonction qui **prend en paramètre une fonction** :

```
1 /* Trie nb_test tableaux de taille n générés par gen,
2    en utilisant l'algorithme tri. */
3 float test_tri(void gen(int*, int), void tri(int*, int), int n, int nb_tests){
4     int* T = malloc(n*sizeof(int));
5     clock_t deb = clock();
6     for (int i = 0; i < nb_tests; ++i) {
7         gen(T, n);
8         tri(T, n);
9         assert(is_sorted(T, n));
10    }
11    clock_t fin = clock();
12    float total = (float) 1000.0f*(fin-deb)/CLOCKS_PER_SEC;
13    return total / nb_tests;
14 }
```

Dans cette fonction, le premier paramètre `gen` est de type `void (int*, int)` : c'est une fonction qui prend en entrée un pointeur/tableau et un entier, et qui ne renvoie rien. Dans le main, on peut aussi créer un tableau contenant nos fonctions de génération, et un autre contenant nos fonctions de tri, et lancer tous les tests avec une seule boucle :

```
1 /* Algos de tri */
2 void (*tris[4]) (int*, int) = {
3     tri_insertion,
4     tri_rapide,
5     tri_mediane,
6     tri_hybride
7 };
8 char* noms_tris[4] = {"Insertion", "Rapide", "Médiane", "Hybride"};
```

```
9
10 /* Générateurs de tableaux */
11 void (*gens[3]) (int*, int) = {random_tab, decroissant, quasi_trie};
12 char* noms_gens[3] = {"Aléatoire", "Décroissant", "Quasi-trié"};
13
14
15 for (int i = 0; i < 3; ++i){
16     printf("Tableaux %s\n", noms_gens[i]);
17     for (int n = 1000; n <= 10000; n=n+1000){
18         printf("n = %d\n", n);
19         for (int j = 0; j < 4; ++j){
20             float t = test_tri(gens[i], tris[j], n, nb_tests);
```

Le code complet est dans le fichier `sort_func.c` de l'archive du corrigé.

La syntaxe est assez moche, surtout au niveau des types : C n'est pas un langage fait pour manipuler facilement les fonctions de cette manière.

En OCaml, on verra que les fonctions sont essentiellement des types de base : on pourra faire très simplement des listes de fonctions, des fonctions qui renvoient des fonctions, etc...