CORRECTION DU TP2

Tables d'entiers en C - implementation, allocation dynamique et analyse de complexite

Unite	4R-IN1 - Algorithmique et structures de donnees en C
Sujet de reference	TP2 : Table (d'entiers)
Langage / compilation	C90, gcc, -Wall -Wextra -Wpedantic -O2
Contenu du paquet	Code commente, tests, Makefile, benchmark CSV et ce rapport PDF

Objectif de la correction

Fournir une implementation complete de l'API demandee, mais aussi un outil fiable pour etudier la complexite : maintien d'un tableau trie, recherche dichotomique, redimensionnement avec realloc, tri par insertion, quicksort et choix du pivot.

La correction conserve les fonctions du sujet, ajoute des tests automatiques et propose un benchmark qui compare les memes tableaux pour les quatre tris. Les mesures sont fournies a titre pedagogique ; l'analyse s'appuie en priorite sur le nombre de comparaisons et d'ecritures plutot que sur des temps d'execution tres courts.

Principales invariants

- Capacite : `size >= 1` et `0 <= n <= size`.
- Donnees : les `n` premieres cases contiennent les elements valides ; les autres cases sont reservees.
- Ordre : `insert` et `del` preservent l'ordre croissant ; les doublons sont autorises.
- Allocation : `realloc` passe toujours par un pointeur temporaire afin de ne pas perdre l'ancien bloc en cas d'echec.
- Robustesse : les fonctions verifient les pointeurs et les bornes principales ; une table vide a une capacite minimale de 1.

Point de vigilance : l'implementation traite explicitement RATIO=100. L'ajout redimensionne aussi lorsque le tableau est plein, sans quoi la prochaine insertion accederait hors limites.

1. Structure et operations

Le type Table contient un pointeur vers le tableau dynamique, sa capacite et le nombre d'elements valides. Cette separation est essentielle : `size` est une capacite de stockage, tandis que n est la taille logique des donnees.

```
typedef struct Table {
    int *data;
    int size;
    int n;
} Table;
```

Redimensionnement

Lors d'un ajout, la table est agrandie si l'occupation courante depasse RATIO pour cent ou si elle est pleine. La capacite double en temps normal. Avant une insertion, toute reallocation reussie preserve les elements deja presents ; si elle echoue, insert renvoie -1 sans modifier la table. Apres suppression, la capacite est reduite de moitie quand l'occupation passe sous RATIO/2 pour cent. Une reallocation de reduction qui echoue n'annule pas la suppression.

Operations principales

Fonction	Resume / convention	Complexite
new_table / free_table	Alloue la structure et son tableau, puis libere les deux blocs.	O(1) hors cout allocateur
print_table / is_sorted	Affiche les n valeurs / verifie l'ordre croissant.	O(n)
dicho_search	Recherche la premiere occurrence dans une table triee.	O(log n)
insert	Trouve la position, decale les valeurs puis insere ; realloc occasionnel.	O(n), avec decalages
del	Retire une occurrence, decale les valeurs suivantes ; tente de reduire la capacite.	O(n), avec decalages
random_inserts	Effectue nb insertions aleatoires au plus.	O(nb * n) au pire
shuffle	Melange les n valeurs valides avec Fisher-Yates.	O(n)
del_mutiple / clean	Compacte le tableau en un passage puis tente une reduction.	O(n) chacun

Cas des doublons

insert place une nouvelle occurrence apres celles de meme valeur. dicho_search et del ciblent la premiere occurrence. del_mutiple (orthographe de l'enonce) retire toutes les occurrences ; del_multiple est aussi fourni comme alias. clean suppose le tableau trie et ne conserve qu'une occurrence de chaque valeur.

2. Tris et instrumentation

Tri par insertion

A chaque iteration, la valeur courante est memorisee puis les valeurs plus grandes sont decalees vers la droite. Le tri est simple et stable. Il effectue O(n) comparaisons sur un tableau deja trie et $O(n^2)$ comparaisons/deplacements dans le pire cas, par exemple sur un tableau inverse.

Quicksort

La partition choisit un pivot, le place temporairement au debut, puis utilise deux indices qui se rapprochent : l'indice gauche avance sur les valeurs inferieures ou egales au pivot, l'indice droit recule sur les valeurs superieures ou egales. Le pivot est place a sa position finale, puis les deux sous-tableaux sont tries recursivement. Les bornes des boucles sont explicites afin d'eviter les acces hors tableau.

Trois choix de pivot sont fournis : premier element, element central de l'intervalle, ou position aleatoire. Le pivot central est une position centrale, pas la mediane calculee des valeurs. Pour un tableau trie, le pivot au debut devient systematiquement un mauvais pivot et produit une partition tres desequilibree.

Complexites attendues

Algorithme / cas	Meilleur ou typique	Pire cas
Tri par insertion	O(n) si deja trie ; souvent $O(n^2)$ sur aleatoire	$O(n^2)$
Quicksort	O(n log n) en moyenne si les partitions restent equilibrees	$O(n^2)$
Quicksort + pivot debut	Souvent O(n log n) sur entree aleatoire	$O(n^2)$ sur entree triee ou inversement triee

Interpreteur des compteurs

`SortStats` mesure les comparaisons entre valeurs, les echanges de deux cases distinctes et les ecritures dans le tableau. Le tri par insertion effectue des decalages, pas des echanges : son compteur swaps reste donc a zero. Pour comparer les deux familles de tris, il faut regarder les comparaisons et les ecritures, et ne pas se limiter aux echanges. Le temps CPU est conserve dans le CSV, mais il est beaucoup plus sensible au bruit pour des tris qui durent quelques microsecondes.

La macro DEBUG=1 affiche les statistiques sur stderr, de maniere a ne pas polluer le CSV de benchmark. Les memes statistiques peuvent etre lues dans le programme via last_sort_stats().

3. Mesures experimentales

Le benchmark compare les quatre algorithmes sur des tailles de 100, 200, 400, 800 et 1600 elements. Chaque point est une moyenne sur 10 essais. Pour chaque essai, les quatre algorithmes recoivent exactement les memes donnees ; trois distributions sont testees : aleatoire, deja triee et en ordre inverse.

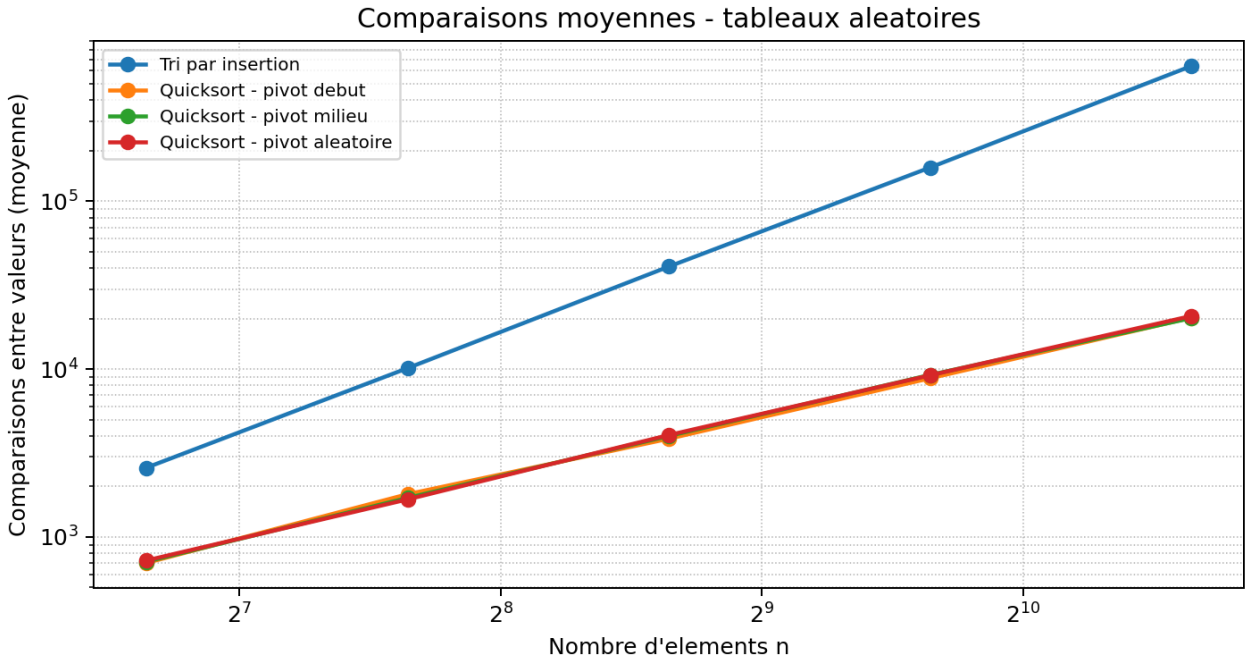


Figure 1 - Comparaisons moyennes sur des tableaux aleatoires. L'axe vertical logarithmique permet de montrer simultanement la croissance quadratique du tri par insertion et celle, plus proche de n log n sur ces donnees, des variantes de quicksort.

Algorithme	Comparaisons	Echanges	Ecritures
Tri par insertion	640716	0	640723
Quicksort - pivot debut	20274	3779	7558
Quicksort - pivot milieu	20167	4593	9185
Quicksort - pivot aleatoire	20752	4571	9142

Sur les entrees aleatoires de taille 1600, le tri par insertion effectue environ 644 000 comparaisons, contre environ 20 000 pour chaque variante de quicksort. Ce resultat illustre l'ecart entre $O(n^2)$ et $O(n \log n)$ sur cet echantillon ; il ne constitue pas une preuve mathematique et les constantes dependent de l'implementation.

4. Sensibilite au choix du pivot

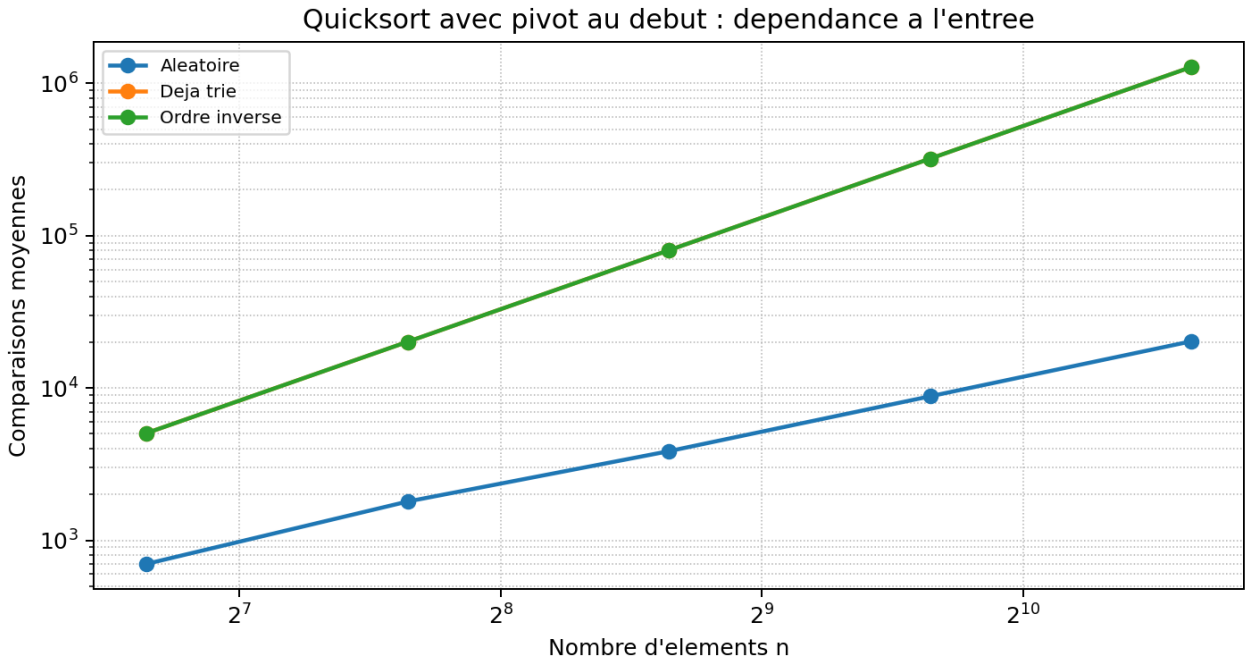


Figure 2 - Pour quicksort avec pivot au debut, le nombre de comparaisons devient quadratique sur les tableaux deja tries et inverses. L'algorithme reste correct, mais le choix du pivot degrade fortement ses performances.

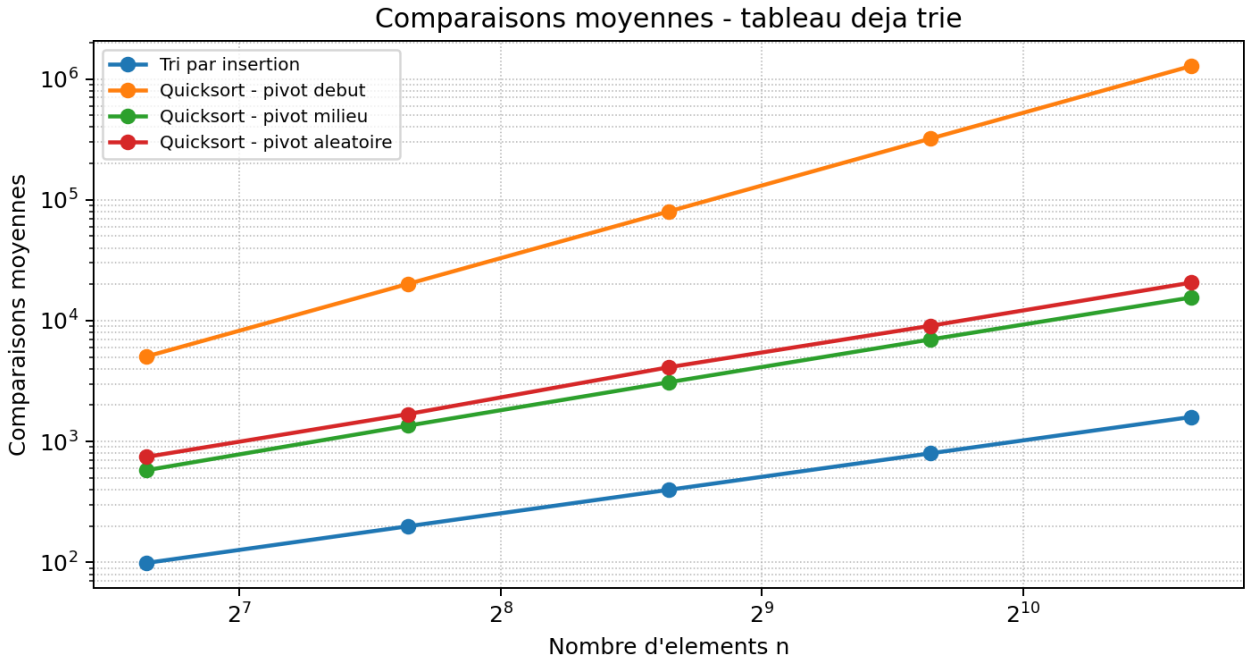


Figure 3 - Sur une entree deja triee, le tri par insertion est lineaire. Le quicksort avec pivot au debut est quadratique ; les pivots milieu et aleatoire conduisent ici a des partitions plus equilibrees.

5. Compiler, tester et reproduire

Compilation et tests automatiques

```
make clean
make
make test
./tp2 --demo
```

Les tests couvrent notamment l'insertion dans une table vide, l'ordre trie, les doublons, la recherche, les suppressions simples et multiples, le nettoyage des doublons et les quatre tris. La correction a ete compilee en C90 avec avertissements actives. Des tests supplementaires ont compare les tris a un resultat de reference sur des tableaux aleatoires contenant des doublons. Les tests ont egalement ete executes avec RATIO=1 et RATIO=100.

Benchmark

```
make benchmark
# ou, pour modifier taille maximale et nombre d'essais :
./tp2 --benchmark 3200 20 > benchmark.csv
```

Le benchmark fixe une graine et reutilise les memes donnees pour les quatre tris : les compteurs d'operations sont donc reproductibles. Les temps CPU mesures avec clock() fluctuent d'une execution a l'autre et leur precision est limitee pour les petites tailles. Les compteurs logiciels doivent servir de base principale a l'interpretation.

Configuration RATIO / DEBUG

```
gcc -std=c90 -Wall -Wextra -Wpedantic -O2 -DRATIO=75 table.c test.c -o tp2
make clean
make CFLAGS='-std=c90 -Wall -Wextra -Wpedantic -O2 -DDEBUG=1'
```

Notes pedagogiques

- Le comptage d'echanges ne suffit pas pour comparer insertion sort et quicksort ; le premier decale les elements.
- Le temps mesure depend de la machine, du compilateur, de l'optimisation et du bruit d'execution. Conserver les memes conditions lors d'une comparaison.
- Le quicksort recursif peut consommer une profondeur de pile O(n) dans le pire cas. Le pivot aleatoire reduit le risque sans supprimer theoriquement le pire cas quadratique.

Conclusion : cette correction sert a la fois de reference fonctionnelle et de support de mesure. Les invariants de representation et les cas limites (table vide, doublons, echec d'allocation, capacite pleine) sont aussi importants que l'algorithme de tri lui-meme.