GTU Department of Computer Engineering
CSE 222/505 - Spring 2022
Homework 6 Report

Emirkan Burak Yılmaz
1901042659

1. SYSTEM REQUIREMENTS

Q1. Hash Table Implementations

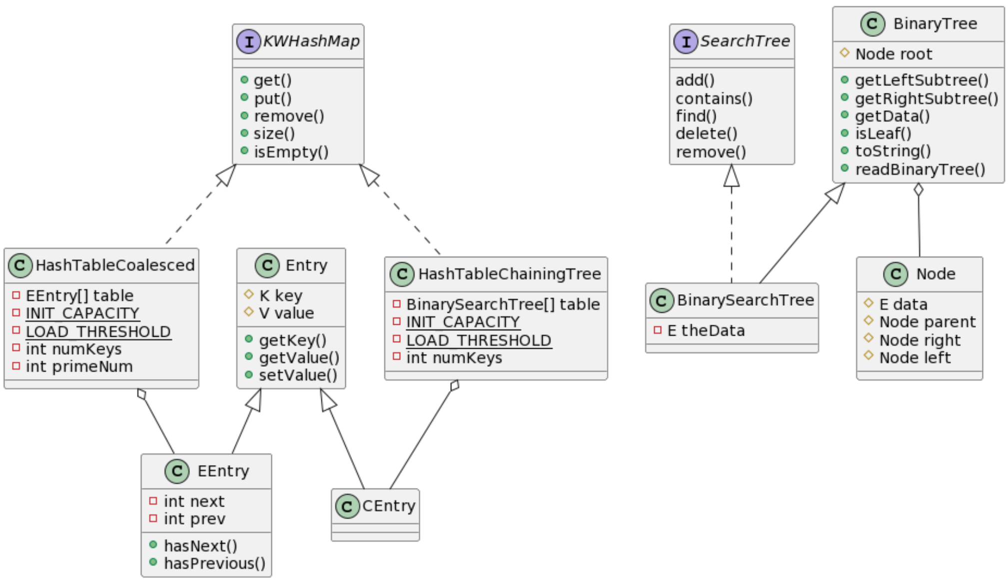
- **I.** Hash table classes should implement KWHashMap interface.
- **II.** Two different implementations: separate chaining and hybrid of open addressing and chaining.
- **III.** In First implementation separate chaining should be used with binary search trees to chain items mapped on the same table slot.
- **IV.** In second implementation, a hashing technique that is combination of the double hashing and coalesced hashing should be used to map the items.
- **V.** Explain advantages and disadvantages of coalesced chaining over standard open addressing and chaining method.
- **VI.** Explain advantages and disadvantages of double hashing over standard open addressing and chaining method.
- **VII.** Hash table implementations should be tested empirically. There should be 100 randomly generated data sets for different set sizes small (100), medium (1000), and large (10000).
- **VIII.** Empirical results should be obtained as the average of 100 independent runs for two hash table implementations.
- **IX.** Compare the performance of the two implementations by using test result.

Q2. Sorting Algorithms

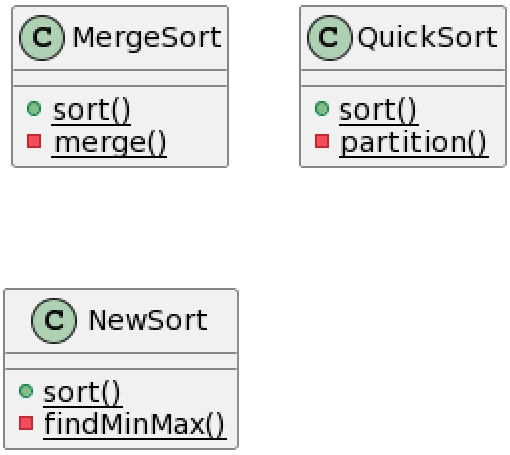
- I. Implement and evaluate the merge sort algorithm.
- II. Implement and evaluate the quick sort algorithm.
- III. Implement and evaluate the new sort algorithm.
- IV. Test sorting algorithms both empirically and theoretically. There should be 1000 randomly generated data sets for different set sizes small (100), medium (1000), and large (10000).
- V. Empirical results should be obtained as the average of 1000 independent runs for each algorithm and problem size individually.
- VI. The execution time complexity should be analyzed theoretically.
- VII. The consistency between empirical and theoretical analysis should be evaluated.
- VIII. Performance of the sorting algorithms should be compared and explained by tables and diagrams.

2. CLASS DIAGRAMS

Q1. Hash Table Implementations



Q2. Sorting Algorithms



3. PROBLEM SOLUTION APPROACH

Q1. Hash Table Implementations

HashTableChainingTree is an implementation of KWHashMap interface for use of hash table. The class has an inner class CEntry (Comparable Entry) to keep the key-value pairs. Binary search trees are used to chain the entries. So, entries are kept İn hash table which is an array of binary search trees. İn each new entry addition, class makes sures that the load factor below load threshold. If it's above, then the table dynamically grows with method rehash.

HashTableCoalesced is an implementation of KWHashMap interface for use of hash table. The class uses special hashing technique which is combination of double hashing and chaining. With this hashing technique the advantages of changing are gained and the disadvantages of wasting memory by keeping buckets empty are prevented. The class has EEntry (Enhanced Entry) which keeps the index of next and previous entry at the hash table. Chains are implemented by using previous and next field of the entries.

The colliding items are linked to each other and searching a key is done by traversing links at the chains. First collision index found by double hashing then if there is chain, that chain is traversed to find the target entry. Adding new entry is done by traversing chain till reaching last entry at the chain and determined the next position by double hashing. Removing an entry is done by firstly founding place of the entry and if it’s head of the chain then the next entry copied the place of target entry, and the place of next entry is set empty, and links are readjusted. If the target entries are at the end or middle of the list, they are removed, and the links of previous and next entries are readjusted properly.

- **Advantages and Disadvantages of Coalesced Chaining**

Coalesced hashing, also called coalesced chaining avoids the effects of primary and secondary clustering, and as a result can take advantage of the efficient search algorithms for separate chaining. If the chains are short, this strategy is very efficient for both time and space complexity. However, chains can be long, and this may reduce the run-time performance.

- **Advantages and Disadvantages of Double Hashing**

Normally if two keys have same hash value then both follow same probe sequence, and this causes clusters to form. However double hashing offers better resistance against clustering over the other variants linear probing and quadratic probing. A major reason for this is use of dual functions. Dual functions enable double hashing to perform a more uniform distribution of keys, resulting in lower collision. As a downside it's requires more calculation because of the dual hash functions.

Q2. Sorting Algorithms

Merge Sort

- Merge sort is an recursive algorithm which sorts the given array by splitting the array two halves and sort them separatly. After two halves become sorted merge them two sort whole array.
- To analyze the algorithm it should be note that merge sort is not in-place algorithm. It requires extra space to split and sort two halves seperatly. That's because the number of copy operation is an important aspect that should be consern. On the other hand the number of comparison should be analyzed for a detailed analyzes.

⇒ merge operation in terms of # of comparison

$M_b(n) = \frac{n}{2}$ → Every item in the first halve is smaller than the second halve. That's beause all the items are selected first array.

$M_w(n) = n-1$ → In each iteration smaller item selected equally in two halves. That's beause there needs to be $n-1$ comparison.

$\boxed{\text{merge} : M(n) = \Theta(n)}$

⇒ # of copy operation

$C(n) = 2C(\frac{n}{2}) + n$ → whole array copied into two half sizes array. , $C(1) = 0$

$C(n) = 2(2C(\frac{n}{4}) + \frac{n}{2}) + n = 2^2 C(\frac{n}{2^2}) + 2n$

$C(n) = 4(2C(\frac{n}{8}) + \frac{n}{4}) + 2n = 2^3 C(\frac{n}{2^3}) + 3n$

$C(n) = 2^k C(\frac{n}{2^k}) + kn$ $\quad 2^k = n$

$C(n) = n \log n$ $\quad k = \log n$

$\boxed{C(n) = \Theta(n \log n)}$

$\Rightarrow$ # of comparison

$$C(n) = 2C\left(\frac{n}{2}\right) + n \quad \text{merge operation}$$

$$C(n) = 2\left(2C\left(\frac{n}{4}\right) + \frac{n}{2}\right) + n = 2^2 C\left(\frac{n}{4}\right) + 2n$$

$$C(n) = 2^k C\left(\frac{n}{2^k}\right) + kn$$

$$C(n) = n\log n$$

$$\boxed{C(n) = \Theta(n\log n)}$$

- So both # of comparison and # of copy operation takes $\Theta(n\log n)$, Overall time complexity of merge sort algorithm is $\Theta(n\log n)$

$$\boxed{T(n) = \Theta(n\log n)}$$

New Sort

- New sort is a recursive algorithm which sorts the array by finding minimum and maximum items and placed them at head and tail of the current arrays.

- Algorithm uses another recursive algorithm which finds minimum and maximum items. In each recursive call problem size reduced by 2 till it becomes 0. Since algorithm reduced the array size by 2, overall recursive calls become $\frac{n}{2}$. So if we assume comparison between terms are constant time, then the algorithm time complexity become $\Theta(n)$.

$$T(n) = \sum_{i=1}^{n/2} \frac{i}{2} = \frac{1}{2}\sum_{i=1}^{n/2} i = \frac{1}{2} \cdot \frac{(n/2).(n/2+1)}{2} = \frac{n^2+2n}{16}$$

finding min and max items

in each recursive call head position increases by one and tail position decreases by one. So overall $n/2$ recursive call will be made.

$$\boxed{T(n) = \Theta(n^2)}$$

Quick sort

- Quick sort is an recursive algorithm which sorts the array by partition. In each recursive call, a pivot value is selected and the array is rearranged as every item smaller than pivot value is at the left and every item larger than pivot value is at the right side of the pivot position. An this process of swapping items to split the array is called partition. After partition is done the place of pivot item is it's sorted position because everything its left is smaller and everything its right is larger than itself. By using some technique left and right sides can be sorted recursively.

- Partition takes linear time since every item should be compared with the pivot value. So to analyze overall run-time complexity partition can be used.

- Selection of pivot value is very important, if the pivot value is extreme value (either min or max term), the array splitted with only one item one side and the rest is other side. With such a selection the recursion depth become n. However if the pivot value is selected as array is splitted equally, then recursion depth will be logn.

$T_w(n) = T(n-1) + \underbrace{n}_{\text{partition}}, \quad T_w(0) = 0$

$T_w(n) = T(n-2) + n-1 + n$

$T_w(n) = T(n-k) + (n-k+1) + \dots + n \quad k=n$

$T_w(n) = 1+2+3+ \dots + n$

$T_w(n) = \frac{n.(n+1)}{2}$

$\boxed{T_w(n) = \Theta(n^2)}$

$T_b(n) = 2T(\frac{n}{2}) + \underbrace{n}_{\text{partition}}$

$T_b(n) = 2^k T(\frac{n}{2^k}) + kn$

$\boxed{T_b(n) = \Theta(n \log n)}$

- $T(n) = O(n^2)$
  $T(n) = \Omega(n \log n)$

- In average case algorithm works with $\Theta(n \log n)$ time complexity, since the probability of gettin extreme value on partition is very low.

sorting algorithm	time complexity
Merge sort	$\Theta(n\log n)$
New sort	$\Theta(n^2)$
Quick sort	best : $\Theta(n\log n)$, worst : $\Theta(n^2)$

- So New sort always works in quadratic time thats because it's the worst one. Merge sort always works $n\log n$ time complexity which makes it choiceable in terms of stabity especially for real-time processes. Quick sort works with $n\log n$ time complexity in average case. However in worst case it works in quadratic time. That's because its not apporiate selection for real time processes.

4. TEST CASES

Q1.

i. HashTableChainingTree

Test ID	Test Case	Test Steps	Test Data	Expected Results	Actual Results	Pass/Fail
T1	Put a key-value pair	1. Create a hash table. 2. Call method put with key and value.	keys = {3, 12, 13, 25, 23, 51}	If the key is already then modifies the existing value, otherwise inserts the new key-value pair.	As Expected	PASS
T2	Get the value associated with the given key	Call method get with the key value on existing hash table.	keys = {3, 12, 13, 25, 23, 51, 134, 124, 34, 53}	If the key exist returns the value associated with its key.	As Expected	PASS
T3	Remove a key-value pair	Call method remove with the key value on existing hash table.	keys = {3, 1, 12, 13, 25, 23, 51, 12, 324, -12}	If the key exist then removes the pair and returns its value, otherwise returns null. As result all the entries are removed.	As Expected	PASS

ii. HashTableCoalesced

Test ID	Test Case	Test Steps	Test Data	Expected Results	Actual Results	Pass/Fail
T1	Put a key-value pair	1. Create a hash table. 2. Call method put with key and value.	keys = {3, 12, 13, 25, 23, 51}	If the key is already then modifies the existing value, otherwise inserts the new key-value pair.	As Expected	PASS
T2	Get the value associated with the given key	Call method get with the key value on existing hash table.	keys = {3, 12, 13, 25, 23, 51, 134, 124, 34, 53}	If the key exist returns the value associated with its key.	As Expected	PASS
T3	Remove a key-value pair	Call method remove with the key value on existing hash table.	keys = {3, 1, 12, 13, 25, 23, 51, 12, 324, -12}	If the key exist then removes the pair and returns its value, otherwise returns null. As result all the entries are removed.	As Expected	PASS

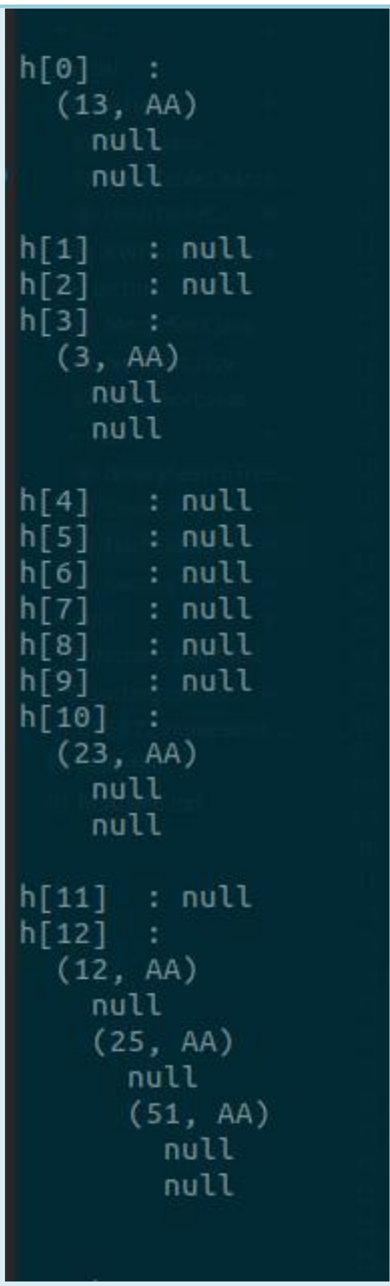
Q2.Sorting Algorithms

Test ID	Test Case	Test Steps	Test Data	Expected Results	Actual Results	Pass/Fail
T1	Test Merge Sort	1. Create a random data set (array). 2. Call sort method of MergeSort class.	A random generated data set which sizes 100.	Given array is sorted as ascending order.	As Expected	PASS
T2	Test New Sort	1. Create a random data set (array). 2. Call sort method of NewSort class.	A randomly generated data set which sizes 100.	Given array is sorted as ascending order.	As Expected	PASS
T3	Test Merge Sort	1. Create a random data set (array). 2. Call sort method of QuickSort class.	A randomly generated data set which sizes 100.	Given array is sorted as ascending order.	As Expected	PASS
T4	Run Time Comparison	1. Create different data sets for different sizes by randomly. 2. Measure total execution time of sorting all the data sets for 3 different sorting algorithm.	100 random data set for each set sizes with sizes small (100), medium (1000) large (10000).	Execution times QuickSort is fastest and NewSort is slowest.	As Expected	PASS

5. RUNNING AND RESULTS

Q1. Hash Table Implementations

i. HashTableChainingTree

<table>
<tr><th>Test ID</th><th>Test Result</th></tr>
<tr><td>T1</td><td>
h[0] :
(13, AA)
null
null

h[1] : null
h[2] : null
h[3] :
(3, AA)
null
null

h[4] : null
h[5] : null
h[6] : null
h[7] : null
h[8] : null
h[9] : null
h[10] :
(23, AA)
null
null

h[11] : null
h[12] :
(12, AA)
null
(25, AA)
null
(51, AA)
null
null</td></tr>
</table>

T2	Get key 3 : SUCCESFULL Get key 12 : SUCCESFULL Get key 13 : SUCCESFULL Get key 25 : SUCCESFULL Get key 23 : SUCCESFULL Get key 51 : SUCCESFULL Get key 134 : FAIL Get key 124 : FAIL Get key 34 : FAIL Get key 53 : FAIL Remove 3
T3	h[0] : null h[1] : null h[2] : null h[3] : null h[4] : null h[5] : null h[6] : null h[7] : null h[8] : null h[9] : null h[10] : null h[11] : null h[12] : null

ii. HashTableCoalesced

Test ID	Test Result
T1	HashValue Key Next Prev 0 null null 1 null null 2 null null 3 23 null 8 4 12 5 null 5 13 null 4 6 51 null null 7 3 null null 8 25 3 null 9 null null --- numKeys: 6 | capacity: 10 | load factor: 0.60
T2	Get key 3 : SUCCESFULL Get key 12 : SUCCESFULL Get key 13 : SUCCESFULL Get key 25 : SUCCESFULL Get key 23 : SUCCESFULL Get key 51 : SUCCESFULL Get key 134 : FAIL Get key 124 : FAIL Get key 34 : FAIL Get key 53 : FAIL

T3

```
HashValue  Key       Next      Prev

0          null      null
1          null      null
2          null      null
3          null      null
4          null      null
5          null      null
6          null      null
7          null      null
8          null      null
9          null      null
-----------------------------------------------------
numKeys: 0    | capacity: 10    | load factor: 0.00
```

Average run-time comparison with HashTableChainingTree and HasTableCoalesced with three different set sizes.

```
------------------------ HashTableChaningTree --------------------------------
Method                 Small(100)             Medium(1000)           Large(10000)

Put                    148                    228                    742
Get                    42                     120                    731
Remove                 58                     91                     367
-----------------------------------------------------------------------------
------------------------ HashTableCoalesced ----------------------------------
Method                 Small(100)             Medium(1000)           Large(10000)

Put                    77                     280                    2028
Get                    77                     283                    1667
Remove                 26                     37                     408
-----------------------------------------------------------------------------
```

Q2. Sorting Algorithms

Test ID	Test Result
T1	----------------- Test MergeSort ----------------- Unsorted arr = {308 , 226 , 324 , 644 , 835 , 313 , 829 , 289 , 467 , 517 , 468 , 707 , 468 , 715 , 193 , 110 , 300 , 542 , 675 , 474 , 500 , 57 , 374 , 19 , 611 , 377 , 209 , 156 , 803 , 357 , 253 , 8 , 465 , 178 , 874 , 751 , 267 , 15 , 780 , 785 , 898 , 410 , 58 , 186 , 511 , 638 , 287 , 418 , 730 , 778 , 409 , 943 , 797 , 814 , 247 , 378 , 839 , 36 , 656 , 171 , 244 , 689 , 911 , 754 , 42 , 927 , 794 , 275 , 398 , 512 , 793 , 157 , 445 , 185 , 883 , 175 , 55 , 545 , 100 , 959 , 925 , 804 , 572 , 897 , 147 , 952 , 961 , 184 , 201 , 395 , 228 , 12 , 931 , 391 , 81 , 272 , 168 , 643 , 656 , 294 } Sorted arr = {8 , 12 , 15 , 19 , 36 , 42 , 55 , 57 , 58 , 81 , 100 , 110 , 147 , 156 , 157 , 168 , 171 , 175 , 178 , 184 , 185 , 186 , 193 , 201 , 209 , 226 , 228 , 244 , 247 , 253 , 267 , 272 , 275 , 287 , 289 , 294 , 300 , 308 , 313 , 324 , 357 , 374 , 377 , 378 , 391 , 395 , 398 , 409 , 410 , 418 , 445 , 465 , 467 , 468 , 468 , 474 , 500 , 511 , 512 , 517 , 542 , 545 , 572 , 611 , 638 , 643 , 644 , 656 , 656 , 675 , 689 , 707 , 715 , 730 , 751 , 754 , 778 , 780 , 785 , 793 , 794 , 797 , 803 , 804 , 814 , 829 , 835 , 839 , 874 , 883 , 897 , 898 , 911 , 925 , 927 , 931 , 943 , 952 , 959 , 961 }

T2

```
----------------- Test NewSort -----------------
Unsorted arr = {308 , 226 , 324 , 644 , 835 , 313 , 829 , 289 , 467
, 517 , 468 , 707 , 468 , 715 , 193 , 110 , 300 , 542 , 675 , 474
, 500 , 57 , 374 , 19 , 611 , 377 , 209 , 156 , 803 , 357 , 253
, 8 , 465 , 178 , 874 , 751 , 267 , 15 , 780 , 785 , 898 , 410
, 58 , 186 , 511 , 638 , 287 , 418 , 730 , 778 , 409 , 943 , 797
, 814 , 247 , 378 , 839 , 36 , 656 , 171 , 244 , 689 , 911 , 754
, 42 , 927 , 794 , 275 , 398 , 512 , 793 , 157 , 445 , 185 , 883
, 175 , 55 , 545 , 100 , 959 , 925 , 804 , 572 , 897 , 147 , 952
, 961 , 184 , 201 , 395 , 228 , 12 , 931 , 391 , 81 , 272 , 168
, 643 , 656 , 294 }

Sorted arr = {8 , 12 , 15 , 19 , 36 , 42 , 55 , 57 , 58 ,
81 , 100 , 110 , 147 , 156 , 157 , 168 , 171 , 175 , 178 , 184 ,
185 , 186 , 193 , 201 , 209 , 226 , 228 , 445 , 247 , 253 , 267 ,
272 , 275 , 287 , 517 , 294 , 300 , 308 , 313 , 324 , 357 , 374 ,
377 , 378 , 391 , 395 , 398 , 409 , 467 , 465 , 418 , 410 , 468 ,
468 , 474 , 500 , 511 , 512 , 542 , 545 , 572 , 611 , 638 , 643 ,
644 , 289 , 656 , 656 , 675 , 689 , 707 , 715 , 244 , 730 , 751 ,
754 , 778 , 780 , 785 , 793 , 794 , 797 , 803 , 804 , 814 , 829 ,
835 , 839 , 874 , 883 , 897 , 898 , 911 , 925 , 927 , 931 , 943 ,
952 , 959 , 961 }
```

T3

```
----------------- Test QuickSort -----------------
Unsorted arr = {308 , 226 , 324 , 644 , 835 , 313 , 829 , 289 , 467
, 517 , 468 , 707 , 468 , 715 , 193 , 110 , 300 , 542 , 675 , 474
, 500 , 57 , 374 , 19 , 611 , 377 , 209 , 156 , 803 , 357 , 253
, 8 , 465 , 178 , 874 , 751 , 267 , 15 , 780 , 785 , 898 , 410
, 58 , 186 , 511 , 638 , 287 , 418 , 730 , 778 , 409 , 943 , 797
, 814 , 247 , 378 , 839 , 36 , 656 , 171 , 244 , 689 , 911 , 754
, 42 , 927 , 794 , 275 , 398 , 512 , 793 , 157 , 445 , 185 , 883
, 175 , 55 , 545 , 100 , 959 , 925 , 804 , 572 , 897 , 147 , 952
, 961 , 184 , 201 , 395 , 228 , 12 , 931 , 391 , 81 , 272 , 168
, 643 , 656 , 294 }

Sorted arr = {8 , 12 , 15 , 19 , 36 , 42 , 55 , 57 , 58 ,
81 , 100 , 110 , 147 , 156 , 157 , 168 , 171 , 175 , 178 , 184 ,
185 , 186 , 193 , 201 , 209 , 226 , 228 , 244 , 247 , 253 , 267 ,
272 , 275 , 287 , 289 , 294 , 300 , 308 , 313 , 324 , 357 , 374 ,
377 , 378 , 391 , 395 , 398 , 409 , 410 , 418 , 445 , 465 , 467 ,
468 , 468 , 474 , 500 , 511 , 512 , 517 , 542 , 545 , 572 , 611 ,
638 , 643 , 644 , 656 , 656 , 675 , 689 , 707 , 715 , 730 , 751 ,
754 , 778 , 780 , 785 , 793 , 794 , 797 , 803 , 804 , 814 , 829 ,
835 , 839 , 874 , 883 , 897 , 898 , 911 , 925 , 927 , 931 , 943 ,
952 , 959 , 961 }
```

T4

Sorting Algorithm	Small(100)	Medium(1000)	Large(10000)
Merge Sort	314	867	3006
New Sort	68	1700	146739
Quick Sort	28	178	1366