

TCP2101 - ALGORITHM DESIGN AND ANALYSIS

Trimester 2 19/20

TC02

Lecturer :

Mr. Bau Yoon Teck

Group Member :

1. Foo Fang Jee 1171100973
2. Justin Tey 1161104265
3. Leong Kah Ming 1171100884
4. Michelle Chai Mei Wei 1171101517

[Question 3 – Checklist Table 4](#_Toc16975)

[Quick-select algorithm 5](#_Toc26282)

[1. Array size test (k = 80) 5](#_Toc23539)

[2. Pivot test (worst case scenario, with array size = 10000) 6](#_Toc6267)

[3. Test of different cases (same set of array size = 10000) 7](#_Toc877)

[Merge-sort algorithm 8](#_Toc16358)

[1. Array size test 8](#_Toc13758)

[2. Test of different cases (same set of array size = 10000) 8](#_Toc10461)

[Comparative analysis 10](#_Toc3643)

[Conclusion 13](#_Toc5496)

[Reference 14](#_Toc21426)

# Question 3 – Checklist Table

|  |  |  |  |
| --- | --- | --- | --- |
| **No.** | **Tasks** | **Mark** | **Checklist**  **(Yes/no) and person(s) to work on this task** |
| 1 | Quick-select algorithm for searching the *k*-th element.  - Correct implementation of searching *k*-th element using quick-select algorithm.  - Output the intermediate results for a random array of 10 elements for tutor to inspect the correctness of the algorithms. | 4 | Yes  (Fang Jee, Michelle) |
| 2 | Test the Quick-select algorithm with  - Different array sizes (10,000, 100,000, 1,000,000, etc.) that can show significant results.  - Different pivot (random pivot vs fixed pivot).  - Different cases (e.g. best, average, and worst). | 4 | Yes  (Fang Jee, Michelle) |
| 3 | Merge-sort algorithm for searching the *k*-th element.  - Correct implementation of searching *k*-th element using merge-sort algorithm.  - Output the intermediate results for a random array of 10 elements for tutor to inspect the correctness of algorithms. | 4 | Yes  (Justin, Kah Ming) |
| 4 | Test the Merge-sort algorithm with  - Different array sizes (10,000, 100,000, 1,000,000, etc.) that can show significant results.  - Test the Merge-sort algorithm with different cases (e.g. best, average, and worst). | 4 | Yes  (Justin, Kah Ming) |
| 5 | - Include the above experiment results that can be used to perform a comparative analysis (such as drawing the graphs for comparison) between the two algorithms (Quick-select & Merge-sort) in the report.  - Conclude your findings in the report. | 4 | Yes  (Fang Jee , Justin, Kah Ming, Michelle) |

# Quick-select algorithm

The way that the quick-select algorithm implemented is similar to the quick-sort algorithm. To begin, the last element of the input array will be taken as the pivot to do partition. Using that pivot, the array will be partition into three sections which is Lesser, Equal, and Greater. If the k element that we are trying to look for is equal to the pivot location, the value of the element k will be print. Else if the k is smaller than the pivot location then the partition will occur again in the Lesser section. Else if the k is bigger than the pivot location then the partition will occur again in the Greater section.

Fixed Pivot:  
Please input the kth smallest element: 5 (actual index is 4)

Unsorted array: 42 68 35 1 70 25 79 59 63 65

Pivot = 9: 65

Pivot = 6: 65

42 35 1 25 59 63 65 70 68 79

Pivot = 5: 63

Pivot = 5: 63

42 35 1 25 59 63 65 70 68 79

Pivot = 4: 59

Pivot = 4: 59

42 35 1 25 59 63 65 70 68 79

The 5th smallest element is: 59

Sorted array: 42 35 1 25 59 63 65 70 68 79

Random pivot:

Please input the kth smallest element: 5 (actual index is 4)

Unsorted array: 42 68 35 1 70 25 79 59 63 65

Pivot = 5: 25

Pivot = 1: 25

1 25 35 42 70 65 79 59 63 68

Pivot = 5: 65

Pivot = 6: 65

1 25 35 42 59 63 65 70 79 68

Pivot = 4: 59

Pivot = 4: 59

1 25 35 42 59 63 65 70 79 68

The 5th smallest element is: 59

Sorted array: 1 25 35 42 59 63 65 70 79 68

## & 2.2 Array size test (k = 80): Fixed pivot

To test the performance of quick-select algorithm with fixed pivot implementation, we will be using four data size which is 10000, 100000, 1000000, 500000. Each input size will be tested 5 times to find for the element in 80 and recorded in seconds.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Array size, *n*** | **10,000** | **100,000** | **1,000,000** | **5,000,000** |
| Attempt 1 (s) | 0.000999 | 0.001983 | 0.033009 | 0.170038 |
| Attempt 2 (s) | 0.000992 | 0.002001 | 0.033166 | 0.179039 |
| Attempt 3 (s) | 0.001001 | 0.002001 | 0.034009 | 0.182043 |
| Attempt 4 (s) | 0.000984 | 0.003 | 0.036008 | 0.183031 |
| Attempt 5 (s) | 0.001001 | 0.002 | 0.035008 | 0.17803 |
| Average Duration (s) | 0.0009954 | 0.002197 | 0.03424 | 0.1784362 |

## Pivot test (worst case scenario, with array size = 10000)

In order to tackle worst case in quick-select algorithm, random pivot can be apply. In this test we will be comparing the different in duration for both random and fixed pivot during the worst case with a 10000 input size. Each case will be tested 5 times.

|  |  |  |
| --- | --- | --- |
| **Pivot selection** | **Random** | **Fixed** |
| Attempt 1 (s) | 0.001001 | 0.113026 |
| Attempt 2 (s) | 0.001 | 0.112027 |
| Attempt 3 (s) | 0.001 | 0.112024 |
| Attempt 4 (s) | 0.001 | 0.114027 |
| Attempt 5 (s) | 0.001 | 0.112027 |
| Average Duration (s) | 0.001 | 0.112626 |

## Test of different cases (same set of array size = 10000)

In this test we will be testing different cases that will be occur in quick-select algorithm with fixed pivot implementation. Best case will be getting the k-element in one try. Average case will be a random input data. While the worst will be finding the k-element after n-1 try. To test this we will be using input size of 10000 and each case will be tested 5 times.

|  |  |  |  |
| --- | --- | --- | --- |
| **Case** | **Best** | **Average** | **Worst** |
| Attempt 1 (s) | 0 | 0  (k=37) | 0.112036 |
| Attempt 2 (s) | 0 | 0.001 (k=1238) | 0.112026 |
| Attempt 3 (s) | 0.001001 | 0.001001 (k=4632) | 0.112036 |
| Attempt 4 (s) | 0 | 0  (k=9574) | 0.112024 |
| Attempt 5 (s) | 0.000998 | 0.001 (k=58) | 0.111036 |
| Average duration (s) | 0.0003998 | 0.0006002 | 0.1118316 |

# Merge-sort algorithm

The way that the merge-sort algorithm implemented is by taking the input array and partition them into left and right until one element has left for both sides. Then the comparison of both left and right will start by choosing the smallest from both sides till the largest, which will be call merging. This will occur until the input array is correctly sorted.

Which element would you like to find in the array? 5 ( actual index = 4 )

Before sort: 42 68 35 1 70 25 79 59 63 65

42 68 35 1 70 25 79 59 63 65

35 42 68 1 70 25 79 59 63 65

35 42 68 1 70 25 79 59 63 65

1 35 42 68 70 25 79 59 63 65

1 35 42 68 70 25 79 59 63 65

1 35 42 68 70 25 59 79 63 65

1 35 42 68 70 25 59 79 63 65

1 35 42 68 70 25 59 63 65 79

1 25 35 42 59 63 65 68 70 79

After sorted: 1 25 35 42 59 63 65 68 70 79

Element in position 5 is 59

## Array size test

To test the performance of merge-sort algorithm, we will be using four data size which is 10000, 100000, 1000000, 500000. Each input size will be tested 5 times and recorded in seconds.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Array size, *n*** | **10,000** | **100,000** | **1,000,000** | **5,000,000** |
| Attempt 1 (s) | 0.007011 | 0.077005 | 55.4115 | 1191.9 |
| Attempt 2 (s) | 0.007002 | 0.071027 | 54.8905 | 1197.78 |
| Attempt 3 (s) | 0.007002 | 0.072017 | 56.7984 | 1179.17 |
| Attempt 4 (s) | 0.006010 | 0.071016 | 56.687 | 1183.14 |
| Attempt 5 (s) | 0.007001 | 0.072005 | 56.7014 | 1177.73 |
| Average Duration (s) | 0.006805 | 0.072614 | 56.09776 | 1185.944 |

## Test of different cases (same set of array size = 10000)

In this test we will be testing merge-sort algorithm in different cases. Best case will be having a sorted array as input. Average case will be a random input data. While the worst case will be input that have all the odd number on the left and even number on the right in sorted form. To test this we will be using input size of 10000 and each case will be tested 5 times.

|  |  |  |  |
| --- | --- | --- | --- |
| **Case** | **Best** | **Average** | **Worst** |
| Attempt 1 (s) | 0.001997 | 0.002995 | 0.005027 |
| Attempt 2 (s) | 0.001999 | 0.002992 | 0.006028 |
| Attempt 3 (s) | 0.001992 | 0.003987 | 0.004984 |
| Attempt 4 (s) | 0.001998 | 0.00399 | 0.005027 |
| Attempt 5 (s) | 0.001998 | 0.002975 | 0.004986 |
| Average duration (s) | 0.0019968 | 0.0033878 | 0.0052104 |

# Comparative analysis

# 

Figure : Running time test for different array sizes

According to the graph above, the merge-sort algorithm’s running time increases exponentially after the array size of 1mil, where it takes around 56s to complete, and around 20mins for the array size of 5mil. On the other hand, quick-select algorithm runs quickly, less than 1s for all the different array sizes. The result reflects that time complexity of quick-select algorithm is less than merge-sort algorithm; time complexity of quick-select algorithm is O(*n*), while time complexity of merge-sort algorithm is O(*n lg* *n*).

# 

Figure 2: Different pivot selection for quick-select algorithm

For quick-select algorithm, we use the worst-case scenario to test the usage of fixed pivot in comparison to random pivot. The fixed pivot takes around 0.112626s, while random pivot runs at less than 0.001s. Using a random pivot, the algorithm performs 99.11% faster as compared to using a fixed pivot, which is the last element of the partition in this case. In conclusion, random pivot runs a lot faster than fixed pivot when the array size is 10,000 in a worst-case scenario.

# 

Figure 3: Different cases analysis

For quick-select algorithm, it performs at a similar speed for best and average case, where the time complexity is given by O(*n*). For the worst case, quick-select algorithm performs significantly slower as compared to the other two cases, where the time complexity is given by O(*n*2); the time difference between best case and worst case is 0.1114318s, which is roughly 278 times of the best case’ time of 0.0003998s.

Unlike quick-select algorithm, merge-sort algorithm performs at a similar speed for best, average, and worst case, where the time complexity is given by O(*n lg* *n*). The time difference between best case and worst case is only 0.0032136s which is, which is 1.6 times of the best case’ time of 0.00019968s. In comparison to quick-select algorithm, the worst case of merge-sort algorithm performs 95.3% faster than quick-select algorithm.

# Conclusion

In terms of improvement for our algorithms, for the partition code of quick-select algorithm, the partition through using three sequences (L, E, G) could be used instead to achieve a better representation of the sorted array.

In conclusion, quick-select and merge-sort algorithm both have their pros and cons. Both of them can manage to produce an output almost instantaneously when the input data is small but as the input data grow quick-select will outperform merge-sort. But this theory will be overthrown will happen when the input data is the worst case and a fixed pivot is used as it will cost O(n) for quick-select while O(n log n) for merge-sort. However, it is possible to improve the time complexity to O(n) by implementing a random pivot rather than a fixed pivot. At the end of the day, if we are looking at just the k element in the input data, quick-select will be a better algorithm compared to merge-sort as it will stop as soon as it finds k, but merge-sort will be able to provide a fully sorted array.

# Reference

1. <https://www.geeksforgeeks.org/quickselect-algorithm/>

2. <https://stackoverflow.com/questions/10846482/quickselect-algorithm-understanding>

3. <https://stackoverflow.com/questions/24594112/when-will-the-worst-case-of-merge-sort-occur>

4. <https://www.geeksforgeeks.org/merge-sort/>

5. TCP2101 Lecture 07 Sorting and Selection Algorithms

6. TCP2101 Lecture 08 Divide & Conquer