SINGAPORE INSTITUTE OF TECHNOLOGY

INF1008

Data Structures & Algorithms

Part 2 Assignment

Group 10

Name	Student ID
Wong Yok Hung	2202391
Christopher Kok	2203503
Chong Hou Wei	2201783
Vivian Ng	2203557
Methinee Ang	2202781

Contents

Chapter 1

Task 1

1.1 User Guide

To compile the executable, the user must have an updated version of rustlang's cargo tool installed (if they would like to conduct tests on the algorithm as well as run it). The user then just needs to input into the command line: `cargo run <phone number> <phone number>  <phone number>`. Alternatively, they may run the following on the command line should the program already be compiled: `qn_1.exe <phone number> <phone number>  <phone number>`. The program will then output the median phone number from the given input phone numbers. Note that phone number arguments with whitespace in them must be delimited by quotation marks.

Example `cargo run 123-456-7890 "(323) 456-7890" +1 223-456-7890 1-322-345-7890 "322 555 0000"`

1.2 Structure/Design

The program takes in an input vector of strings, after which it will clean the input string into phone numbers of length 10, then parsed as an integer. If the input vector is empty, the program will terminate. The program will also ignore any phone numbers that have fewer than 10 digits or more than 11 digits to account for the US country code. This process is $O(n)$, as it must iterate across the entire vector once. Next, the vector is passed into a function to find the median values. If the length of the array is an even number, the algorithm runs the quickselect algorithm twice, otherwise once.

The quickselect algorithm is based on the quicksort algorithm, where it only recurses on the slice of the vector where the desired k-th smallest value is after the vector is partitioned by a pivot [1]. In the case of this program, k was chosen to be $\lceil n/2 \rceil$[1]. This leads to quickselect having an average case time complexity of $O(n)$, with the worst case of $O(n^2)$, compared to quicksort's average and worst case time complexity of $O(n \log n)$ and $O(n^2)$ respectively. Due to this $O(n)$ average case, we opted to use a vector as the data structure over other data structures like linked lists or tree-based structures, which at least have an $O(n \log n)$ total time complexity.

1.3 Limitations

Despite the $O(n)$ average case time complexity, it shares the same $O(n^2)$ worst case time complexity due to its similarities with quicksort, where a non-optimally chosen pivot leads to $O(n^2)$. This may be alleviated by using hybrid algorithms such as introselect [2], which uses both the quickselect and median of medians algorithm, depending on which algorithm is expected to perform better for the given input slice.

Due to our implementation of quickselect's inability to select more than one k, the quickselect algorithm must be run twice if the input array's length is an even number, doubling the runtime of the algorithm.

[1] In the case of an even numbered length input, $k_1 = n/2, k_2 = n/2 + 1$

1.4 Testing

Four main tests are conducted to verify the correctness of the algorithm and its performance. The program passes all tests.

- Testing to find the median value for an unsorted array against the value of the median index of a sorted copy of the unsorted array. This array has a random length and randomly chosen integer values. This test is run 100 times.
- Testing that quickselect finds the correct value in the worst case scenario (a sorted array)
- Testing that quickselect finds the median value for an unsorted array of randomly generated 10-digit phone numbers. This test is run 100 times.
- Comparing the efficiency of the algorithm against an implementation of quicksort. Like test 2, the array consists of randomly generated phone numbers, and the comparisons are conducted from an array length of 1 to 100,000. To aid in completing the test faster, the comparisons for each array length are split among all logical processors of the user's CPU. For slices of the arrays that are the same length, their comparisons are run on the same thread to ensure there is no variance between the conditions of the test for both algorithms. Each test is then run 5 times and a mean of the time taken for both algorithms to find the k-th smallest value is recorded to a file and is shown below:

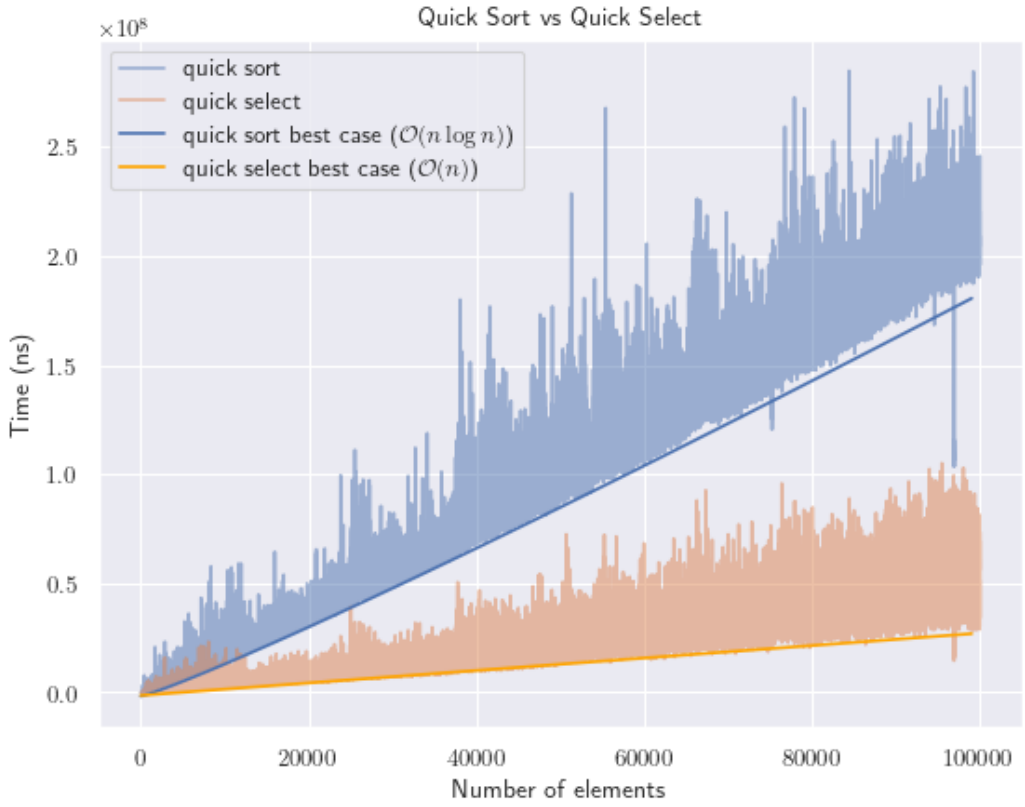


Figure 1.1: Quick Sort vs Quick Select time efficiency

Chapter 2

Task 2

2.1 User Guide

The program takes 3 positional arguments:

1. A filename of a file which contains the list of phone numbers.
2. A target phone number that the user would like to find the nearest K numbers for. Note that if the phone number contains whitespace, it must be delimited by quotation marks.
3. A number K for which the program will find the nearest K phone numbers to argument 2. If $K \leq 0$, the program panics with the error message, "k must be greater than 0".

Example A user may run the program in one of the following manners:

- `cargo run <filename> <target number> <K>`
- `Question_2_rust.exe <filename> <target number> <K>`

2.2 Structure/Design

The phone numbers are first stored in a vector, then are sanitized one by one to ensure only digits remain, then parsed as an integer. Any invalid phone numbers are ignored in this process. Additionally, the program will assert that $k > 0$. This step is $O(n)$, as it must iterate through the entire input list. The numbers are then stored in a hashmap as keys, where the values are the number of times the number has appeared in the input vector. This step is also $O(n)$, as it iterates through the vector which contains the phone numbers. We selected a hashmap as the data structure due to its $O(1)$ element insertion and retrieval time complexity. Figure 2.1 shows an example on how this hashmap is constructed, where the keys are the phone numbers and the values are the count of each number that appears in the input vector.

The absolute difference between each number and the target number is then stored in a BTreeMap, with the values being the number themselves. The BTreeMap was chosen as it has two properties that benefit us. Firstly, due to it being a form of a B-Tree, the keys are sorted by value. Secondly, the B-Tree makes each node contain $B - 1$ to $2B - 1$ elements in a contiguous array for some choice of B, which improves cache efficiency [3]. Construction of a BTreeMap is $O(n \log n)$. Refer to Figure 2.2 for an example of how the BTreeMap may look like.

We then iterate up to K[1] phone numbers of the BTreeMap. As search through a BTreeMap is $O(\log n)$, this process has the worst case of $O(K \log n)$, where there are K unique closest numbers to the target number. If $K > n$, then the time complexity is $O(n \log n)$. Subsequently, the program will print each of the closest K phone numbers as many times as they appeared in the input vector by checking their values in the hashmap, which would cumulatively have $O(K)$ time complexity. Overall, the time complexity of the program is $O(n \log n)$.

[1] $K + 1$ if there is a tie in the last 2 numbers

numbers {}

Key	Value
10	1
13	5
14	20
11	2
5	3
12	2
⋮	⋮

Adding a number that is not in the hashmap

Key	Value
10	1
13	5
14	20
11	2
5	3
12	2
6	1
⋮	⋮

Adding a number that is already in the hashmap

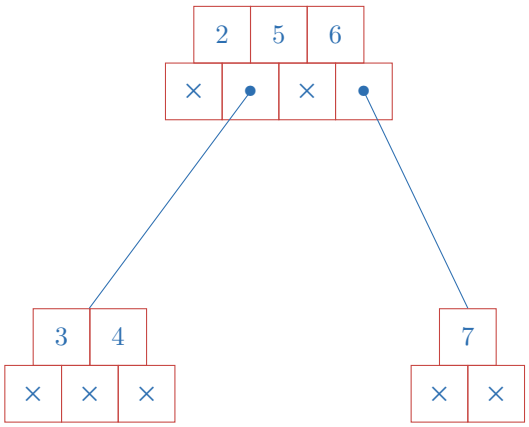
Key	Value
10	1
13	5
14	20
11	2
5	3
12	3
6	1
⋮	⋮

Figure 2.1: Constructing the numbers hashmap (using 2-digit numbers)

differences {}

Key	Value	
2	10	6
5	13	
6	14	
3	11	5
4	12	
7	15	
⋮	⋮	

(a) BTreeMap keys and values



(b) BTreeMap keys tree structure

Figure 2.2: BTreeMap

2.3 Limitations

As the time complexity for the algorithm is $O(n \log n)$, it performs similarly to just running a simple quicksort on the input, then printing K unique phone numbers closest to the target number, which would be far less complex to implement compared to having a BTreeMap in the implementation.

2.4 Testing

The following tests are conducted, and the program passes all.

1. Testing that the sanitization of phone numbers work correctly on a wrongly randomly generated phone number and a valid generated phone number. This process is repeated 100 times.
2. Testing that the algorithm can select the closest K values from a predefined array.
3. Testing that the algorithm will return the same values as a naive solution of this problem for a randomly generated array of random length, with random 10-digit phone numbers as the elements.
4. Comparing the efficiency of the algorithm against solution that uses quicksort. The array consists of randomly generated phone numbers, and the comparisons are conducted from an array length of 1 to 10,000. To aid in completing the test faster, the comparisons for each array length are split among all logical processors of the user's CPU. For slices of the arrays that are the same length, their comparisons are run on the same thread to ensure there is no variance between the conditions of the test for both algorithms. Each test is then run 5 times and a mean of the time taken for both algorithms to find the K-th closest phone numbers is recorded to a file and is shown below:

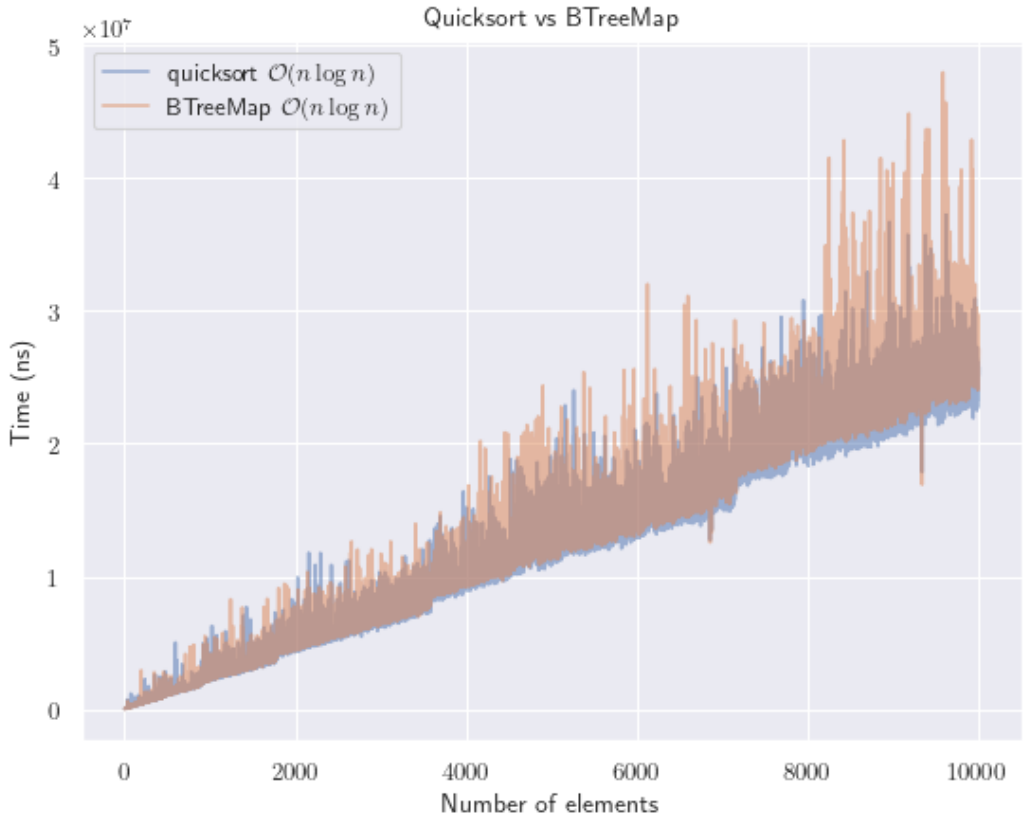


Figure 2.3: Quicksort implementation vs BTreeMap implementation time efficiency

Chapter 3

Bibliography

[1] C. A. R. Hoare, "Algorithm 65: Find," *Commun. ACM*, vol. 4, no. 7, pp. 321–322, Jul. 1961, ISSN: 0001-0782, 1557-7317. DOI: 10.1145/366622.366647. [Online]. Available: https://dl.acm.org/doi/10.1145/366622.366647 (visited on 03/20/2023).

[2] D. R. Musser, "Introspective sorting and selection algorithms," *Softw: Pract. Exper.*, vol. 27, no. 8, pp. 983–993, Aug. 1997, ISSN: 0038-0644, 1097-024X. DOI: 10.1002/(SICI)1097-024X(199708)27:8<983::AID-SPE117>3.0.CO;2-#. [Online]. Available: https://onlinelibrary.wiley.com/doi/10.1002/%28SICI%291097-024X%28199708%2927%3A8%3C983%3A%3AAID-SPE117%3E3.0.CO%3B2-%23 (visited on 03/20/2023).

[3] RustLang. "BTreeMap in std::collections - rust," std - Rust. (May 15, 2015), [Online]. Available: https://doc.rust-lang.org/stable/std/collections/struct.BTreeMap.html (visited on 03/18/2023).