The Divide-and-Conquer and Sequential Algorithms: Retrieving Information About Chapters In The Noble Quran

DIMAS ATHA PUTRA[1*], MUHAMMAD FAISAL[2]
[1,2]Infomatics Engineering, Faculty of Science and Technology Maulana Malik Ibrahim Malang State Islamic University of Malang
[*]*Corresponding Author, [1]230605110052@student.uin-malang.ac.id, [2]mfaisal@ti.uin-malang.ac.id*

Abstract

Reciting the Quran is one of the most commonly done thing Muslims do. Young Muslims learn how to correctly recite the Quran since elementary school. Usually, the teacher would ask to open a chapter in the Quran, to be studied. Big portion of these teachers do not make it easy on how to find the chapters. Usually they just say the title of the chapter, or which part—The Quran is divided into thirty parts—that chapter is in. In this paper, we utilize the binary-search and linear-search algorithm to help students easily find information about Quran chapters, just by looking for their titles. The implementation process involves learning the algorithms and that involves creating a flowchart representation of the algorithms, using the free software Flowgorithm. The algorithms will then be translated into a programming language, and in this example, we build a program from the ground up with R, the data-science focused programming language, utilizing the algorithms. As a result, we were able to build a simple web-app program that can take a keyword for a title a user might want to look up, search through a dataset for the chapter, and return a set of information from that dataset. In conclusion, the algorithms proved worth implementing for a simple program like this one, where searching through a dataset is an imminent task.

Keywords*—search, binary, sequential, algorithm, Quran, chapters.*

I. INTRODUCTION

Searching as an activity has been one of the most significant thing the human race do (Andersson, 2017:1). It is almost ninety percent of the thing people do in their everyday life. Moms who need a new recipe for the family to enjoy, can just search for a recipe in seconds. Students who needs a little help on their academic problems, can search away for an answer, or at least, a way to an answer. Doctors who needs a refresh on how to actually carry out an operation to replace a patient's heart, can just look up for a video on how to do it, in full, for free. A believer with their various Holy Book or Scripture, can look up any chapters or verses and their meaning. Muslims and The Noble Quran, for example.

Muslims in this day and age know about the Quran, and all about its lore. Not many Muslims know everything about the chapters, though. And that is fine. Though, in some cases, Muslims need to know about some portion of information about the chapters. In some classes, when the teacher asks to open a chapter to be studied, they say just the name of the chapter, and sometimes the *juz* (parts—the Quran is divided into thirty parts) that chapter is in. And, due to the Quran to have been published, in different ways and styles—some are easier for new students, the other are really artistic—that new students face difficulties to see if they have got to the right chapter, and they will need to find out the number of that chapter. A little help is needed to find information about chapters, like which number is it in the Quran, just by looking for the name of that chapter.

II. THEORY OF THE ALGORITHMS

There are quite the ways researchers have discovered to search for something in an array of data. Two primitive algorithms are known as sequential search, also well known as the linear-search algorithm; and the many divide-and-conquer algorithms, one of which is known widely as the binary-search algorithm (Parmar & Kumbharana, 2015:13). Linear-search is simply going through every each of the data in the array, and comparing it with the data that is the "keyword." If a match is found, the search is done, which data in the array is the match is returned. Else, the search is over, but nothing is returned. This is the easiest to implement, yet it is pretty slow (Beck & Beck, 1992:112). Because, it has to go through the array of data, one by one, from index 0 to 1. If the array to work with has the length of, say, 20 data, and the match of a keyword chosen would be at index 17, it will have to go all the way to index 17 to get to a returnable (Charalambous & Conn, 1978:168).

the search algorithm: binary search.

Binary-search is about turning the data array bi. This means, dividing it into two. One side will be smaller than the other. The data that is being looked

up could be on either side. This would also mean, sorting the data array first (Keller, 1979:35). For example, let us have an array {0, 1, 4, 5, 7, 10, 11, 14, 15, 16.}.

0 1 4 5 7 10 11 14 15 16

And we need to search for an 11, this means dividing the array into two, i.e. {0, 1, 4, 5, 7}, and {10, 11, 14, 15, 16}.

0 1 4 5 7 10 11 14 15 16

Then, use either the last item of the first array or the first item of the second array, to compare with the keyword, 11. If we use the former, in this case it's 7, that number is less than 11. Therefore, the first array can just be forgotten altogether, because we know, the keyword would never be in that array.

0 1 4 5 7 10 **11** 14 15 16
less than 11

0 1 4 5 7 10 **11** 14 15 16

We would then split the second array into two, i.e. {10, 11, 14}, and {15, 16}.

0 1 4 5 7 10 **11** 14 15 16

Let's use the former method for this one. We compare 14 with 11, which would obviously result in 14 being the bigger number than 11.

0 1 4 5 7 10 **11** 14 15 16
bigger than 11

0 1 4 5 7 10 **11** 14 15 16

Therefore, the second array here is eliminated. 11 is not there. We keep doing this, splitting, comparing, and eliminating irrelevant data, until we find that, 11 is indeed in the array, and it's at index 6.

0 1 4 5 7 10 **11** 14 15 16

This method features more complex outside things to work, like sorting. But it is generally faster. Splitting fields of work into two, means it generally works twice as fast. Some studies show that it can even get to about 5000 times faster than other algorithms, such as sequential searching (Keller, 1979:36).

the matching algorithm: linear search.

There is another algorithm that will be used. To compare the keyword with the value of the array is usually with an is-it-equal-to operation (Lin, 2019:5). In this paper, we will use a string-matching algorithm to compare for the two data. If the data contains the keyword, the keyword is treated like it is equal to the data.

The algorithm is a direct derivative of the sequential-search algorithm. A keyword is used to go through the array of data in a linear fashion (hence the name "linear-search," one character of the keyword at a time. If the each of the characters in the string matches sequentially, data is found to contain the keyword. If within the sequence there is a character that does not match, we keep going through the data, but we start again at the first character of the keyword.

Let "`substrings`" be a string to be compared with (the data), and "`str`" be the string that is to compare (the key). We immediately know visually, that the string of characters "str" exists in the string of characters "sub**str**ings." But how can a computer know that?

We start going through the data string. We will compare the first character of the data with the first character of the key. If it match, we will continue with the second character of the data with the second character of the string. If not, we go through the third character of the data, but we go back with the first character of the key. Let's go with the "`substrings`" as the data and "`str`" as the key.

data[0]
s u b s t r i n g s
s t r
key[0]

We loop through with the length of the first data as the driver of when to stop the looping. Then, we compare if the first character of the data and the first character of the keyword matches.

data[0]
s u b s t r i n g s
s t r
key[0]

The first index of the strings match, we are now at the next index for both data and the key.

data[1]
s **u** b s t r i n g s
s **t** r
key[1]

No match. We keep going through the data but we restart the key index to one.

data[1]
s **u** b s t r i n g s
s t r
key[0]

data[2]
s u **b** s t r i n g s
s t r
key[0]

data[3]

s u b **s** t r i n g s
s t r
key[0]

data[3]
s u b **s** t r i n g s
s t r
key[0]

A match! We keep going through and check if the strings match consecutively from this point on.

data[4]
s u b s **t** r i n g s
s **t** r
key[1]

data[5]
s u b s t **r** i n g s
s t **r**
key[2]

For each comparison, we need to check if the amount of consecutive matches equals to the length of the key string. We do not need to check for a character that is beyond the key string provided. In this case, we are at index 2, which means there were three consecutive matches with the data. Since there are three characters in the keyword, we stop the whole loop and take a direct conclusion, which is that, the data string contains the key string.

III. METHODOLOGY

The purpose of this paper is, essentially, to document the steps it takes to utilize a search algorithm to solve a very specific problem. Applying the binary-search algorithm into an application involves not only learning and designing the algorithm, but also implementing it in some way or another. The workflow is more or less, this flowchart.

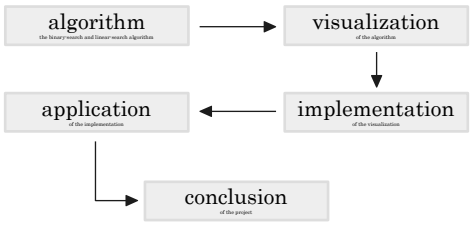

Fig. 1 Workflow for the writing of this paper

In this paper, we have learned the theory of how binary-search and linear-search works. We will then need to apply this theory as an algorithm. We will represent the algorithm as a flowchart, as a plan on how we are going to implement it. This flowchart will come in handy as we try to translate the algorithm into a programming language of our choice.

With the algorithm in a language that we can work with, we can now try and build an actual program that utilizes the algorithm. In this case, it is about utilizing a binary-search to retrieve usable information about Quran chapters. Something like: what number in the Quran is it, how many verses are in it; and some knowledge about its nature, like where it was revelated, and some stories surrounding it. This information comes from a database provided for free by the good folks at Quran.com.

IV. FLOWGORITHM

The designing of this algorithm will be done in the infamous Flowgorithm software, a flowcharting tool designed by Devin Cook. It has been used by lecturers all over the world (Gajewski & Smyrnova-Trybulska, 2018:394). It is a simple, free-to-access-and-use software in which people can create simple programs by defining an algorithm in the form of a flowchart (Kouroma, 2016:1).

The flowchart for the search algorithm that was designed for this paper is the following.

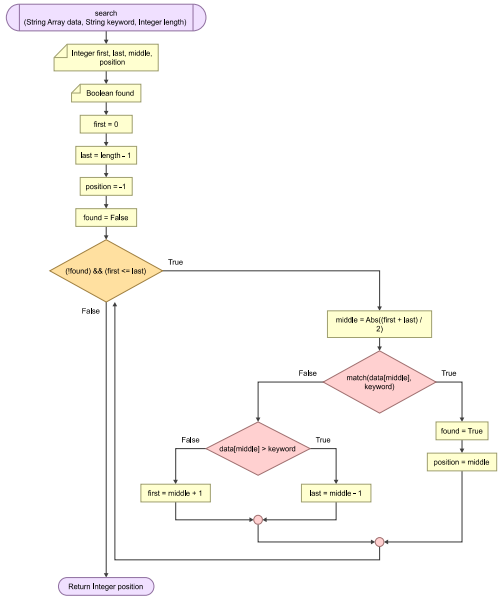

Fig. 2 Binary-search algorithm flowchart

This flowchart is how the simple binary-search algorithm would look like. It first takes in an array of data that is sorted first, because this algorithm was designed for sorted array of data (Biedl et al., 2004:231). It then roughly looks for the data in the middle of the array. If the data is bigger compared to the keyword, the side of bigger array will not be used and we focus on the side of smaller array (Lin, 2019:1).

The function defined here takes three arguments: data array, keyword, and the length of the array. In the function environment, we are to declare four integer variables: `first`, `last`, `middle`,

`position`. The `first` variable is the index of the smallest item in the array (given that the array was sorted in ascending order to begin with). The `last` variable is the index of the biggest item in the array. The `position` variable is used to store the resulting index later. And finally, the `found` boolean variable, just as a switch for the upcoming *while-loop* utilized in this function. We are now ready to start a looping session.

The looping is started if the `found` variable is `FALSE` and the value of `first` is less or equal to `last`. While both of those conditions are met, we start with taking the index of the middle of the range, in this case, for the first loop, it's the entire array. That is by adding `first` and `last`, and dividing the result by 2. The result of that will be made absolute, so that it stays a positive value—because that is how indices are. Then, we do a bit of comparison operation. If the data at that index doesn't match the given keyword, we do another comparison: which one is larger, that data or the keyword. This is why the data array needs to be sorted first (Ahmad et al., 2013:1375). If the data is larger, we set `first` to be the value of `middle + 1`. If the keyword is larger, we set `last` to be the value of `middle - 1`. This process is iterated over and over until, if the data at index `middle` matches the keyword we give, the variable `found` is set to `TRUE` and the position of that data in the array is recorded in the `position` variable.

This algorithm is defined as a function. This function will later be called in a main environment.

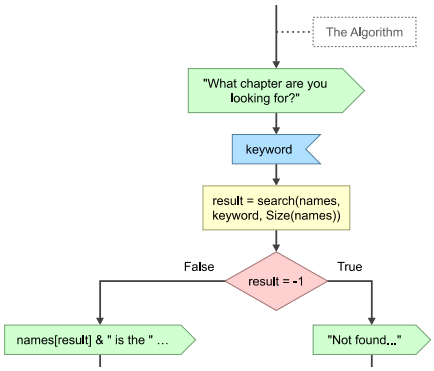

Fig. 3 The algorithm defined as a function, of which returnable value is used in the main function.

For example, in the flowchart in Fig. 2, the value of the variable `position` is the returnable value. It is the index of the match in the array of data, which is set to `-1` before the searching begins. Which means, it will return `-1` if no match was found throughout the searching.

If we go back to *Fig. 2*, you can see that the second algorithm we talked about is utilized in the first comparison in the process. That is, using the string-matching algorithm that is based on a linear-search algorithm, to see if the keyword exists in the data. A linear process in a binary process. There are many, way better string-matching algorithms out there, like the Aho—Corasick algorithm, invented by, well, Alfred Aho and Margaret Corasick (1975). Though very efficient, this algorithm is intended for many key strings, and it is not very easy to implement. We only need to check for one for each comparison, and the implementation should be as simple as it gets for such a simple program. Therefore, we will use a linear-search-like algorithm to pull this off.

The flowchart for the string-matching algorithm is designed like the following.

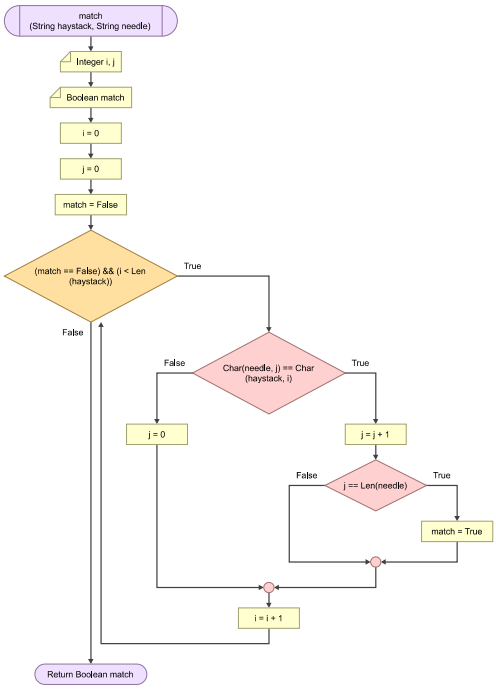

Fig. 4 String-matching algorithm flowchart based on the sequential-searching algorithm

This algorithm will make sure that the closest match to the data is accounted. We don't need to search entirely for the "algorithm" by giving the query string "algorithm." Just with "algo," we can find the right data. These flowchart will be available as a supplement, see Appendix 1.

V. IMPLEMENTATION

The algorithms will be implemented in a web-app, written in R, with the classic Shiny framework. R is a programming language, designed with data science and data scientists in mind. This is the gist of what Wickham & Grolemund (2017:xiii) stated:

> *We think R is a great place to start your data science journey because it is an environment designed from the ground up to support data science. R is not just a programming*

language, but it is also an interactive environment for doing data science.

R was designed by Ross Ihaka and Robert Gentleman, data scientists and professors from the University of Auckland, New Zealand (Peng, 2016:6). The R programming language was a derivative of the S language—designed by Rick Becker, John Chambers, and Allan Wilks—and the Scheme language—designed by Guy Lewis Steele Jr. and Gerald Jay Sussman (Ihaka & Gentleman, 1996:299).

the functions.

The implementation starts by defining the algorithms as functions in R. The codes will slightly differ from the flowchart, because of the difference in how Flowgorithm and R as programming languages work. Defining the function in R can be done like so:

```
binary_search <- function(dataset, keyword, indices) {
    # Sort the data if indices are included
    dataset <- data.frame( "data" = dataset)
    if (!missing(indices)) {
        dataset <- data.frame( "data" = dataset, "indices" = indices)
    }

    for (i in 1:length(dataset$data)) {
        dataset$data[i] <- gsub( " ", "", gsub("'", "", gsub("-", "", tolower(dataset$data[i]))))
    }

    dataset <- dataset[order(dataset$data),]

    # Search
    first <- 1
    last <- length(dataset$data)
    position <- -1
    middle <- 0
    found <- FALSE

    while ((!found) && (first <= last)) {
        middle <- round(abs((first + last) /  2))

        set <- dataset$data[middle]
        key <- gsub( " ", "", gsub("'", "", gsub("-", "", tolower(keyword))))

        if (match_string(set, key)) {
            found <- TRUE
            position <- middle
        } else if (match_string(key, set)) {
            found <- TRUE
            position <- middle
        } else if (set > key) {
            last <- middle - 1
        } else {
            first <- middle + 1
        }
    }
    if (!missing(indices) && position != -1) {
        return (dataset$indices[position])
    } else {return (position)}
}
```

Fig. 5 The algorithm implemented as an R function

For the function, the dataset will contain two set of data: an array of names that will be traversed through, and an array of the indices each name is assigned to. The indices is made to be an array, because the names will be sorted alphabetically, and we want the indices each names is intact. The function includes sorting the dataset first, because binary-searching is designed to work on sorted set of data.

```
# Sort the data
dataset <- data.frame( "data" = dataset)
if (!missing(indices)) {
    dataset <- data.frame(
        "data" = dataset,
        "indices" = indices
        )
}

for (i in 1:length(dataset$data)) {
    dataset$data[i] <- gsub( " ", "",
                            gsub( "'", "",
                                 gsub( "-", "",
                                      tolower(dataset$data[i])
                                      )
                                 )
                            )
}

dataset <- dataset[order(dataset$data),]
```

Fig. 6 Preparation before the search: normalization on the names and then sorting the data by those names

Before sorting begins, we do a little normalization on the names. The names of the Quran chapters usually includes special characters like the dash (-), the apostrophe (') and some has the whitespace. We need to remove that from the equation, so that these characters, if input, does not matter. We also need to make all the characters to be lowercase. This makes the binary-search algorithm case-insensitive. For the normalizations to work as intended, they will also be applied to the keyword given (Suárez-Paniagua & Casey, 2021:805).

Once sorting is done, the next step is to actually start searching. The variables we talked about, namely: `first`, `last`, `position`, `middle`, `found`, is defined and assigned to a default value. Just like in the flowchart, albeit one little change is made. R's indices start from 1 (Venables et al., 2009:26).

```
dataset <- c( "apple", 2, "banana", 4, 5)
dataset[ 1 ]


## [1] "apple"


dataset[ 2 ]


## [1] "2"
```

Fig. 7 R counts from 1

So, the value defined for `first` is set to `1`. R also calculates number differently. Every integer, when put into operations will be treated as real floating-point values. In classic programming languages, 5 divided with 2 will produce the number 2. It is supposed to be 2.5, but since both numbers are treated as an integer, the result will be an integer, and any remainder is pretty much just gone. In R, numbers are treated as it its: a number. 5 divided by 2 will result in 2.5, as it should. Except,

we want that integer-only treatment applied for calculating indices. In the *while-loop*, we need to calculate the index of the data that is in the middle of the array in the range from the value of `first` to `last`. This is done by summing up `first` and `last`, and dividing the result by 2. The value should be the index of the median of the data. Now, in case the result of it would contain decimals, we would need to round it up to get the integer of the number. We also need to make the value absolute, because there are no negative indices.

```
while ((!found) && (first <= last)) {
    middle <- round(abs((first + last) /  2))
```

To improve the readibility of the code and increase the efficiency, we can set a variable for the strings that we are going to go through. Putting the same function each time we need the same thing is a waste of valuable space and time. We can call the function once, put what it returns in a variable, and use that variable each time we need that very value.

```
set <- dataset$data[middle]
key <- gsub(" ", "",
            gsub("'", "",
                 gsub("-", "", tolower(keyword))
                 )
            )
```

Now we can start with the search. We will use traditional *if-else* blocks for this part. We first check if the the data string contains the keyword string. If a user is looking for the word "hopeful," typing "hope" shall count as a match. In some cases, we shall also check the opposite. Does the keyword string contain the data string? Say, a user does not know about how one shall spell "hopeful," and inputs "hopefull," instead. In this case, checking for the keyword string in a data string will give way to an error because there are more characters in the keyword string than the data string. In our simple string-matching algorithm, this scenario will return the value `FALSE`, as in the word "hopefull" does not show up in the word "hopeful." The opposite is true, however. The word "hopeful" exists in "hopefull." Therefore, we also need to check for the opposite of finding the needle in a haystack. This is one of the things that were not in the flowchart, and were eventually added during implementation.

```
if (match_string(set, key)) {
    found <- TRUE
    position <- middle
} else if (match_string(key, set)) {
    found <- TRUE
    position <- middle
} else if (set > key) {
    last <- middle - 1
} else {
    first <- middle + 1
}
```

This code can be optimized to get rid of the repetition. We can set the first *if* statement to check both the conditions, whether if the data string contains the keyword string, or if the keyword string contains the data string. This can be done in R using the widely used `OR` operator. The `OR` operator—or more commonly utilized with the double vertical bar characters (||)—evaluates the boolean values of the left operand and the right operand (Venables et al., 2009:39). If one of those operands comes out as `TRUE`, the block is executed. If none of the operands is `TRUE`, the block is skipped, and we go to the *else* statement.

```
if ((match_string(set, key)) || (match_string(key, set))) {
    found <- TRUE
    position <- middle
} else if (set > key) {
    last <- middle - 1
} else {
    first <- middle + 1
}
```

Now, we will look at that `match_string()` function utilized in the *if-else* blocks. This function is an implementation of the sequential-search-like algorithm we designed earlier, which can be seen as its flowchart form in *Fig. 4*. The function in R is as follows:

```
match_string <- function(haystack, needle) {
    # Convert string to vector (not necessary but makes things easier)
    haystack <- strsplit(haystack,  "")[[1]]
    needle <- strsplit(needle,  "")[[1]]

    i <- 1
    j <- 1
    match <- FALSE

    while ((match == FALSE) && (i < length(haystack))) {
        if (needle[j] == haystack[i]) {
            j <- j + 1
            if (j == length(needle)) {
                match <- TRUE
            }
        } else {
            j <- 1
        }
        i <- i + 1
    }

    return (match)
}
```

Fig. 8 The string-matching algorithm defined as an R function

The two arguments plugged into this function will be converted into R vector objects. This is because, R does not treat string objects like arrays, as how its cousing Python would. In Python, getting the first character of the string "hopeful" is simply:

```
text = "hopeful"
text[0]


## 'h'
```

In R, the string will need to be converted to a vector. This is done by splitting the string by a character, and in this case, that character is just an empty string, because every character will need to be split. So, the string "hopeful" will become an array-

like object that contains each character as the elements, namely {"h", "o", "p", "e", "f", "u", "l"}. R can do the indexing thing with this kind of object. This is not really necessary, though improves efficiency and readibility.

```
text <- strsplit( "hopeful", "")[[1]]
text


## [1] "h" "o" "p" "e" "f" "u" "l"


text[ 1]


## [1] "h"
```

Now we are ready to traverse through the characters of the two strings. We shall start a loop while the a `match` has not been found, and as long as we are traversing within the data string. The `match` boolean variable is made to keep track of if the matching completes. The expression

```
i < length("data string")
```

will keep the loop within the count of the amount of character in the data string. Within the *while-loop*, each loop will add `1` to `i`, to count how many loops has been going on. This i variable will also be used to get the character of the data string at index `i`. In short, the loop keeps going as long as two conditions are met: a match has not been found, and we are still traversing the data string. If either of those two conditions is false, the loop stops, and the function returns the value of the current match variable. If it `FALSE`, no match is found, and the opposite is true.

```
while ((match == FALSE) && (i < length(haystack))) {
    if (needle[j] == haystack[i]) {
        j <- j + 1
        if (j == length(needle)) {
            match <- TRUE
        }
    } else {
        j <- 1
    }
    i <- i + 1
}
```

This *while-loop* will do what we learned in the "theory of the algorithms" chapter. For each the characters in the haystack string, we check if the first character of the needle string matches. If it does, we go to the next character in the haystack string, and check if the second character of the needle string matches. Each time a match happens, we keep going with our haystack string as usual, but we check for the next character in the needle string. When it does not match, still going with the haystack string as usual, we go back to the first character of the needle string to be checked. The while-loop completes if there the amount of consecutive matches equals to the amount of characters in the needle string. It basically means every character in the needle string, with their respective order, exists in the data string. This will set the value of `match` to `TRUE`, thus the *while-loop* can be exit.

For this program, we also added a few little functions to help with the algorithms. These functions are not in the algorithms, but it is still an algorithm, so we will talk about it briefly. These functions are defined solely for this program, to provide for the varying types of information that is provided. Because one of the information that is to be retrieved about the chapters in the Quran is the number of order it is, we need to make the number ordinal, i.e., adding "st", "nd", "rd", and "th" suffixes to numbers. This function works simply by checking the value of the ones in a number. If it is 3, the suffix is "rd". If it is 2, the suffix is "nd". If it is 1, the suffix is "1st". Other than that, it's all "th". And one more thing to add for the 1-2-3 condition, if the tens place is 1, the suffix will remain "th". The function is defined in R like so:

```
make_ordinal <- function(number) {
    suffixes <- c( "st", "nd", "rd")

    number <- as.character(number)
    len <- nchar(number)
    last_digit <- substr(number, len, len)
    suffix <- "th"

    before_last_digit <- "0"
    if (len > 1) {before_last_digit = substr(number, len - 1, len - 1)}
    if ((last_digit %in% c("1", "2", "3")) &&
        (before_last_digit != "1"))
        {suffix <- suffixes[as.integer(last_digit)]}

    print(suffix)

    return (paste(number, suffix, sep = ""))
}
```

Fig. 9 Number to ordinal number converter function

One more function defined to improve the quality of output by the program is the very simple `capitalize()` function. This function will simply make the first character of the string given to be capitalized. The function works by taking the first character, capitalizing it, and attaching it back with a substring that starts from the second character until the last character in the string.

```
capitalize <- function(string) {
    return (paste(sep = "",
        toupper(substr(string, 1, 1)),
        substr(string, 2, nchar(string))))
}
```

Fig. 10 Function to capitalize the first character of a string

the dataset.

Algorithms that we need for this program have been successfully implemented. We are now ready to utilize these algorithms and build our program. The data that we are going to go through is provided for free by the good folks over at Quran.com through their API. The base URL for the API, at the time of writing, is

`https://api.quran.com/api/v4`

To fetch the list of chapters of the Quran in the database, we add `/chapters` at the end.

`https://api.quran.com/api/v4/chapters`

The response of this request will be in a form of a JSON file, a standard format for RESTful APIs.


```
{
    "chapters": [
      {
        "id": 1,
        "revelation_place": "makkah",
        "revelation_order": 5,
        "bismillah_pre": false,
        "name_simple": "Al-Fatihah",
        "name_complex": "Al-Fātiĥah",
        "name_arabic": "الفاتحة",
        "verses_count": 7,
        "pages": [
          1,
          1
        ],
        "translated_name": {
          "language_name": "english",
          "name": "The Opener"
        }
      },
      {
        "id": 2,
        "revelation_place": "madinah",
        "revelation_order": 87,
        "bismillah_pre": true,
        "name_simple": "Al-Baqarah",
        "name_complex": "Al-Baqarah",
        "name_arabic": "البقرة",
        "verses_count": 286,
        "pages": [
          2,
          49
        ],
        "translated_name": {
          "language_name": "english",
          "name": "The Cow"
        }
      },
      {
        "id": 3,
        "revelation_place": "madinah",
        "revelation_order": 89,
        "bismillah_pre": true,
        "name_simple": "Ali 'Imran",
        "name_complex": "Āli `Imrān",
        "name_arabic": "آل عمران",
        "verses_count": 200,
        "pages": [
          50,
          76
        ],
        "translated_name": {
          "language_name": "english",
          "name": "Family of Imran"
        }
      },
      {
        "id": 4,
        "revelation_place": "madinah",
        "revelation_order": 92,
        "bismillah_pre": true,
        "name_simple": "An-Nisa",
        "name_complex": "An-Nisā",
        "name_arabic": "النساء",
        "verses_count": 176,
        "pages": [
          77,
          106
        ],
        "translated_name": {
          "language_name": "english",
          "name": "The Women"
        }
      },
}
```


To get a value, we have to go down the JSON tree. For example, to get to the English name of the first chapter in the Quran.com, we will need go from `chapters`, first index, `translated_name`, and then `name`, of which the value is "The Opener." Which is indeed the English translation of Al-Fātiĥah, the 1st chapter in the Noble Quran. Where was it revelated? We can find out by going from chapters, first index, revelation_place. So, if we want to give out the information of the order it exists in the chapter in the Quran, and the place of revelation, the result would be,

"Al-Fātiĥah is the 1st chapter in The Noble Quran, revelated in Makkah."

Note that we will make use of the information of the index to get the ordinal number, and our `make_ordinal()` function will come in handy. The raw value of the `revelation_place` is "makkah," and we can use our `capitalize()` function to make it capitalized.

The entire JSON response from `/chapter` is imported in R as a data frame object, using the `fromJSON()` function provided by the `jsonlite` package. And for the `binary_search()` function, we extract the list of simple names and their indices, to be sorted in said function.

```
quran <- fromJSON( "https://api.quran.com/api/v3/chapters/" )

chapters <- tibble(
    indices = quran$chapters$chapter_number,
    chapters = quran$chapters$name_simple)
chapters <- chapters[order(chapters$chapters),]
```

	id	revelation_place	revelation_order	bismillah_pre	name_simple	name_complex	name_arabic	verses_count	pages
	<int>	<chr>	<int>	<lgl>	<chr>	<chr>	<chr>	<int>	<list>
1	1	makkah	5	FALSE	Al-Fatihah	Al-Fātiḥah	الفاتحة	7	<int [2]>
2	2	madinah	87	TRUE	Al-Baqarah	Al-Baqarah	البقرة	286	<int [2]>
3	3	madinah	89	TRUE	Ali 'Imran	Āli `Imrān	آل عمران	200	<int [2]>
4	4	madinah	92	TRUE	An-Nisa	An-Nisā	النسآء	176	<int [2]>
5	5	madinah	112	TRUE	Al-Ma'idah	Al-Mā'idah	المائدة	120	<int [2]>
6	6	makkah	55	TRUE	Al-An'am	Al-'An`ām	الأنعام	165	<int [2]>
7	7	makkah	39	TRUE	Al-A'raf	Al-'A`rāf	الأعراف	206	<int [2]>
8	8	madinah	88	TRUE	Al-Anfal	Al-'Anfāl	الأنفال	75	<int [2]>
9	9	madinah	113	FALSE	At-Tawbah	At-Tawbah	التوبة	129	<int [2]>
10	10	makkah	51	TRUE	Yunus	Yūnus	يونس	109	<int [2]>

1-10 of 114 rows | 1-10 of 12 columns Previous 1 2 3 4 5 6 ... 12 Next

Fig. 11 The JSON tree as an R data frame

The data frame contains many other data frame, like a tree having branches. This mimics the structure of a JSON file, albeit being a little bit different. Each item in a JSON file contains a set of data, be it a value like a string, or another set of data. R data frames make use of the similarity of data items in a JSON dataset have, and creates a table that uses the keys as the category.

These data will be used in this layout, a standard Shiny application layout.

Fig. 12 Web-app layout

Keyword for the searching algorithm will be taken through the left sidebar input field. We put a prompt so users will know that they are supposed to put the name of a chapter they are looking for in this field. The keyword they typed in will then be used in the binary search algorithm.

```
ui <- fluidPage(
    tags$head(
        tags$link(rel = "stylesheet", type = "text/css", href = "style.css")),

    div(style = "max-width: 1000px; margin: 0 auto; text-align: justify;" ,
        div(style = "font-size: 2em",
            titlePanel(
                div( "Chapters in The Noble Quran" , class = "title"),
                windowTitle = "Chapters in The Noble Quran"
                ),
            div(
                span(style = "h4", "by Dimas Atha Putra" ),
                span(style = "font-size: .5em" , "230605110052" )
                )
            ),

        hr(),

        sidebarLayout(
            sidebarPanel(
                textInput( "keyword", "What chapter?" )
            ),

            # Show a plot of the generated distribution
            mainPanel(
                div(style= "max-width: 1000px; margin: 0 auto; text-align: center;" ,
                    uiOutput( "found")
                )
            )
        ),
        uiOutput( "info")
    ),
)
```

Fig. 13 Web-app layout code

This is the Shiny code that would be behind the example layout in Fig. 12. The input field to get the keyword from the user will be put in a variable named "keyword. This variable will be used throughout the server function to get the information about the Quran using the database that was fetched from the API.

```
output$found <- renderUI({
    keyword <- "Maryam"
    if (input$keyword != "") {
        keyword <- input$keyword
    }

    pos <- binary_search(chapters$chapters, keyword, chapters$indices)

    if (pos != -1) {
        chapter_name <- quran$chapters$name_complex[pos]
        chapter_number <- quran$chapters$chapter_number[pos]
        chapter_verses <- quran$chapters$verses_count[pos]
        chapter_revelated <- quran$chapters$revelation_order[pos]
        chapter_town <- quran$chapters$revelation_place[pos]

        div(style = 'font-family: "Courier Prime"' ,
            h3(paste(chapter_name, ", the ", make_ordinal(chapter_number), " chapter in The Noble Quran." , sep = "")),
            p(paste( "Consists of ", chapter_verses, " verses, it is the ", make_ordinal(chapter_revelated), " revelation in the town of ", capitalize(chapter_town), sep = "" ))
        )
    } else {
        div(style = 'font-family: "Courier Prime"' ,
            h3("Chapter not found."),
            p("Try guessing for the correct transliteration of the chapter. It should be \"Al-Fatihah\" instead of \"AlFatihah\"")
        )
    }
})
```

This will render the part of the layout right beside the text input: the main panel. This part contains the name of the chapter in the correct transliteration, the order of where it is in the Quran, the amount of verses in it, the order of revelation, and the place of revelation. Right below these two panels, will be rendered a full description about the chapter. An Arabic name of the chapter will also be displayed. To get these information we use the

`https://api.quran.com/api/v4/chapters/[i]/info`

API to get the deeper information about each chapters. This will contain a short description, and a long description. These data are utilized like so.

```
output$info <- renderUI({
    keyword <- "Maryam"
    if (input$keyword != "") {
        keyword <- input$keyword
    }

    pos <- binary_search(chapters$chapters, keyword, chapters$indices)

    if (pos != -1) {
        chapter_info <- fromJSON(paste( "https://api.quran.com/api/v3/chapters/" , pos, "/info", sep = ""))
        div(
            div(style = 'font-family: "Amiri"; font-size: 10vw; text-align: center' , quran$chapters$name_arabic[pos]),
            HTML(gsub("\\\\", "", chapter_info$chapter_info$text))
        )
    }
})
```

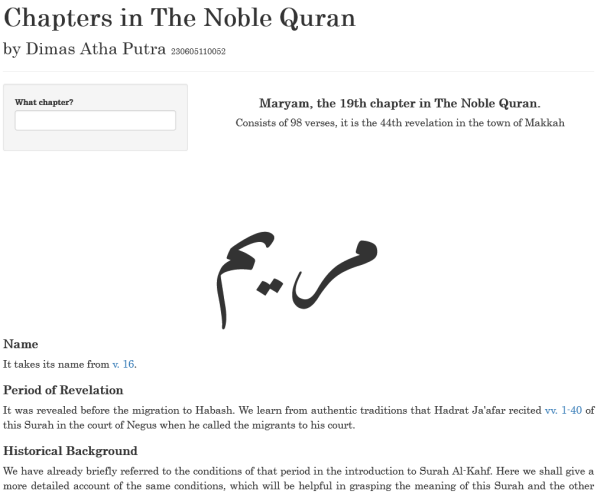

Fig. 14 Rendered web-app

The web-app is available for use, see Appendix 02. Searching through it is very simple: just put the name of the chapter, and it will get the information of the chapter that it thinks you are looking for. For example here, the chapter "Āli `Imrān" is to be looked up.

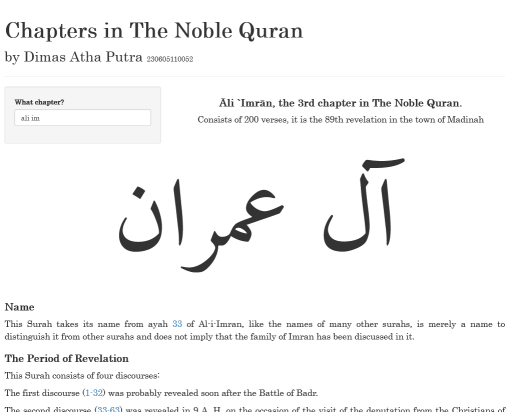

Fig. 15 Looking up Āli `Imrān

Though users will probably need to look the chapters up with the full name and correct transliterations, this web-app makes it that the search algorithm is insensitive to those specifications. This is because of the string-matching algorithm we implemented earlier, getting the search to the closest item, which in this case happens to be the correct one.

VI. CONCLUSION

There are many search and string-matching algorithm that were developed over the years. The most simplest one were discussed in this paper. These primitive approach prove to still work very efficiently, especially for beginner programmers, where one would build a program of which the tasks involves taking a keyword and doing a search through a dataset with it. We successfully implemented binary-search and linear-search as one algorithm, and using that to search through a dataset provided by an API, and using said API, return a set of data as an output. The linear-search algorithm is implemented as a string-matching algorithm, of which is used in the binary-search algorithm to compare between the keyword and the actual data that is traversed in a dataset. By getting the index of the data that is found, we can use it to find out more about stuff about that data in the same API. This simple app can hopefully be useful in teaching-learning environments, or even for anyone who want to learn about the chapters in The Noble Quran.

REFERENCES

Ahmad, M., Ikram, A. A., Wahid, I., & Salam, A. (2013). Efficient Sort Using Modified Binary Search-a New Way to Sort. *World Applied Sciences Journal*, *28*(10), 1375-1378.

Aho, A. V., & Corasick, M. J. (1975). Efficient String Matching: An Aid to Bibliographic Search. *Commun. ACM*, *18*(6), 333–340. https://doi.org/10.1145/360825.360855

Andersson, C. (2017). "Google is not fun": an investigation of how Swedish teenagers frame online searching. *Journal of Documentation*, *73*(6), 1244-1260.

Beck, A., & Beck, M. (1992). The revenge of the linear search problem. *SIAM journal on control and optimization*, *30*(1), 112-122.

Biedl, T., Chan, T., Demaine, E. D., Fleischer, R., Golin, M., King, J. A., & Munro, J. I. (2004). Fun-sort—or the chaos of unordered binary search. *Discrete Applied Mathematics*, *144*(3), 231-236.

Charalambous, C., & Conn, A. R. (1978). An Efficient

Method to Solve the Minimax Problem Directly. SIAM *Journal on Numerical Analysis, 15*(1), 162–187. http://www.jstor.org/stable/2156570

Gajewski, R. R., & Smyrnova-Trybulska, E. (2018). Algorithms, Programming, Flowcharts and Flowgorithm. *E-Learning and Smart Learning Environment for the Preparation of New Generation Specialists*, 393-408.

Ihaka, R., & Gentleman, R. (1996). R: a language for data analysis and graphics. *Journal of computational and graphical statistics*, *5*(3), 299-314.

Keller, Kenneth H. (1979). *Computation cost functions for divide-and-conquer algorithms*. *11*(1), 27–32. https://jstor.org/stable/community.30005202

Kourouma, M. K. (2016). Capabilities and Features of Raptor, Visual Logic, and Flowgorithm for Program Logic and Design.

Lin, A. (2019). Binary search algorithm. *WikiJournal of Science*, *2*(1), 1-13.

Parmar, V. P., & Kumbharana, C. K. (2015). Comparing linear search and binary search algorithms to search an element from a linear list implemented through static array, dynamic array and linked list. *International Journal of Computer Applications*, *121*(3).

Peng, R. D. (2016). *R programming for data science* (pp. 86-181). Victoria, BC, Canada: Leanpub.

Suárez-Paniagua, V., & Casey, A. (2021). BERT and Approximate String Matching for Automatic Recognition and Normalization of Professions in Spanish Medical Documents. *IberLEF@ SEPLN*, 803–813.

Venables, W. N., Smith, D. M., Team, R. D. C., & others. (2009). *An introduction to R*. Network Theory Limited Bristol.

Wickham, H., Çetinkaya-Rundel, M., & Grolemund, G. (2023). *R for data science*. " O'Reilly Media, Inc.".

APPENDICES

01	Repository	*https://github.com/contrapoetra/chapters-of-the-quran/*
02	Web-app (serverless)	*https://contrapoetra.github.io/chapters-of-the-quran/*
03	Web-app	*https://contrapoetra.shinyapps.io/chapters-of-the-quran/*