Haramaya University

School of Graduate Studies

Department of Computer Science

Comparative Analysis on the Performance of Selected Searching Algorithm on Afan Oromo Text File

M.Sc. Experimental Research

Abrahim Abdo and Ketema Dirba

College: Computing and Informatics

Department: Computer Science

Program: Research Methodology

Major Advisor: Mesfin K. (PhD)

23 October 2021

Haramaya, Oromia, Ethiopia

Table of Contents

1. Introduction 2

1.1 Objective 2

2. Literature Review 3

3. Experimental setup 4

3.1 Proposed Approach 5

4. Data Collection & Analysis 9

4.1 Binary search 10

4.2 Hash Table 11

5. Data Interpretation 13

6. Conclusion 14

7. **References** i

8. Annex ii

1. Introduction

Searching is a process of checking and finding an element from a list of elements, It's important to any user who are dealing with computer and in current daily public service, in order to identify and understand the sensitive texts or existence of data in their storage medium e.g., religious texts, laws, health and rules are often used the number of searching method to identify the data they are looking for is in database or not [1]. There is a number of methods for search i.e., linear search, binary search, interpolation search and hashing function [2]. Linear search is a simple approach where each item of an array is checked sequentially until the searched item is found.

Binary search work on a sorted list of items, it finds the middle index M by dividing the search interval by two. If the value of the searched item is less than the item at M, the item can only be found in the lower half of the interval [3]. A different approach hash table to searching calculates the position of the key in the table based on the value of the key. The value of the key is the only indication of the position. When the key is known, the position in the table can be accessed directly, without making any other preliminary tests, as required in a binary search or when searching a tree.

Selecting the best searching algorithm is one of the most subjective issues in different application area. Selecting a particular algorithms is depends on the nature of the problems and the number of factor that affect the performance of the algorithm. In algorithm analysis and design the most important resource to analyze when selecting a particular algorithms is generally the running time and amount of memory it needs. Several factors affect the running time of a program, the main factors are the algorithm used and the input to the algorithm. The other resource to analyze is space complexity of an algorithm or program is the amount of memory it needs to run the program [4]. Performance Analysis is that, there may be more than one approach (or algorithm) to solve a problem but, the best algorithm in efficiency to solve a given problem is one that requires less space in memory and takes less time to complete its execution. However, in practice, it is not always possible to achieve both of these objectives. One algorithm may require more space but less time to complete its execution while the other algorithm requires less time space but takes more time to complete its execution [1].

Afan Oromo is one of the major African languages that is widely spoken and used in most parts of Ethiopia and some parts of other neighbor countries like Kenya and Somalia [5]. It is used by Oromo people, who are the largest ethnic group in Ethiopia. Currently, Afan Oromo is an official language of Oromia regional state (which is the largest Regional State among the current Federal States in Ethiopia). Being the official language, it has been used as medium of instruction for primary and junior secondary schools of the region. Afan Oromo Text is ANSI Code characters that adopted form Latin, we can use UTF-8 mode for Afan Oromo text file reading in python.

In this paper we need to perform experimental research method on Afan Oromo text file by selecting tow searching algorithm namely binary search and hashing. Data to be used for this experiment is three Afan Oromo text file, that have different size for the first file its size is to be expected 5MB, 20MB and 50MB respectively. The paper is organized as follows, after this introduction the literature review is presented in section 2, then section 3 provides the experimental setup, section 4 present the conducted data analysis, then section 5 present the results and discussion and finally section 6 is concludes the paper. At the end of paper the proposed prototype will be presented.

1.1 Objective

Our main objective is to make performance analysis in this three text by developing experimental setup, as the size of text file become larger or as the text file size become large and larger, which a searching algorithms will do better performance under a certain given assumptions. When we analyze an algorithm in case of time complexity it depends on the input data, there are three cases. In the best case, in the average case, and in the worst case, the amount of time a program might be expected to take on best, average, and worst possible input data respectively.

Then we will start by setting the Hypothesis:

In Binary search it finds the middle index M by dividing the search interval by two. Observe that in each comparison the size of the search area is reduced by half. Hence in the worst case, at most log2 n comparisons required. Time Complexity in the average case is almost approximately equal to the running time of the worst case. When we analyze an algorithm in case of space complexity it depends on the approach used by algorithm or program it needs the amount of memory to run the program. The major difference between the iterative and recursive version

of Binary Search is that iterative version has a space complexity of O(1). In case of hash, the search time is reduced from O(log n), as in a binary search, to 1 or at least O(1); regardless of the number of elements being searched, the run time is always the same. But this is just an ideal, and in real applications, this ideal can only be approximated. Space complexity in a hash table, it use hash function values and slots. Consequently, the space complexity of every reasonable hash table is O(n).

2. Literature Review

In [3] the authors presents a new technique of efficient search of a sequence of words in a large text file using the concept of hashing and linked list. The hash table is built by organizing the words of the original text file in order. The number of occurrences and positions of each word and positions of its next words in the original file are also stored to the record of each word in the hash table. The hash function is used only to locate the upper and lower bounds in the hash table of the first word under search. These bounds are the locations of the first and the last words in the hash table starting with the first character of the word under search. Then a binary-search like algorithm is used to match the word in the table. Total memory requirement is calculated by the difference of total word in hash table and the repeated word in the hash table times the average word. However, in their approach they were using linked list with hash table and searching for sentences in hash table is compression of each charterers in the hash table after sorted. In reality the searching sentences in the text means that all words are finding in the ordered they were in original text file not in alphabetical order.

In [6] they proposed Search algorithm using blocks and words prefixes**.** The algorithm depends on the concept of word prefix, the letters at the beginning of the word. The idea of this algorithm is that words with length three or four are not stored in the search blocks and words with length more than four are stored in blocks ordered by alphabet. And when they made a search for a word, they do not make the search from the beginning of the block, they make it from the first word that has the same prefix as the search word in the suitable block. They were used execution time and number of comparisons as performance parameters to compare between the performances of them. However, this algorithm is not a completed because they try to delete the word in the original file for their practical purpose and limited on the length of words. Moreover, they was not mention about the space complexity yet in their works.

In [2] problem is a special problem of string matching since they search a sequence of words (sentence) in a large database. They used another approach instead of binary search a quasi-Newton method i.e., Secant method. Their working procedure are, first, text sentences are organized in higher order sequence according to Unicode values as science, they were solve Arabic text that are needed to change into Unicode. The algorithm was tested on a database containing 6236 sentences. The sentences are sorted and stored in 29 files (as 29 letters exist in Arabic). The sentences starting with (Alif) the first Arabic alphabets, are stored in F1.txt; the sentences starting with Baa (ب) the second Arabic alphabet are stored in F2.txt; and so on. However, this algorithm is used to sort the sentence and apply searching for sentences in the file, they did not concerns about, when we went find specific word in the file. And also a large number of file that means, the sentences are stored in 29 files (as 29 letters exist in Arabic).

3. Experimental setup

In our Python experimental setup, we can use UTF-8 mode for Afan Oromo text file reading in python. The iterative approach is used for implementing the binary searching algorithm. The proposed Approach search of a phrase in a large original text file using hashing and python is demonstrated. The following table describe the hardware, software specification we want to use for implementing the experimental setup.

System Property	Value	Software used	Value
Brand	Dell	Framework	PyCharm
Model	Latitude E7440	Version	Community Edition 2021.1
System type	X64-based PC	Programing Language	Python 3.9
Operating system	Windows 10 Pro	Notepad++	Text editor
Processor	Intel® core™ i5-4310U CPU @ 2.00GHz	MS-Word, MS-Exel	Report write Analysis data
Main memory	4GB	EdrawMax	Drawing flowchart
		Library	Json ,Time Numpy,Statistics
		Encoding type	Default UTF-8

Table 1: Hardware and **Software** Used

File organization and source, Afan Oromo Text is ANSI Code characters that adopted form Latin. In the table-1 we try to mention our machine hardware capability, taking into account the system main memory for processing large file, the following table-2 is description of the file with the amount of memory being considered for our RAM processing a large text file.

File type	Property	Source
File1.txt	Afan Oromo text file 5MB	Afan Oromo News ,books, MSc thesis ,and any Afan Oromo text form internet and BBC,VOA and FBC Afan Oromo news etc.
File2.txt	Afan Oromo text file 16MB	
File3.txt	Afan Oromo text file 32MB	
Temporary file	For sorting we have temporary file for each text file hash table construct and reporting the complexity for analysis	This Temporary file produced by program during runtime

Table 2: File organization and source.

3.1 Proposed Approach

The original text File1 is organized as hash table $Hash_t$ using the scheme presented by flowcharts in Fig-1 shows the flowchart of reading and storing all the words of File1 in a temporary file. First, three empty files Temp1, Temp2 and Temp3 are created. For each file, all the words are read and stored in a Temp1 and the words are storing and arranged in Temp2. For each file the approach will applied before searching started but, for binary search we need only temporary sorted file not hash table and also binary search is used to find a given word in phrase at a time. But for hash table we can apply a given phrase at time or even sentences. A summary of the terms used in the following figures are now identified.

- File1: the large text-file being considered.
- Temp1: the hash-table to be constructed using defined parameters.
- Temp2,: temporary sorted of Temp1.
- Temp3: used for recording data in running time and space
- Hasht: the hash-table to be constructed using defined parameters.

- I: index number, where i<: n.
- S: sentence/word-sequence with p words to be searched from within F1.
- Phrase: text the sentence/word-sequence S to be searched from within F1.
- WJ: first word in S to be searched
- Wj1: first character in the first word to be searched when j=1,2,3,….p
- Index values are determined to set the values U and L for the first index and least index respectively.

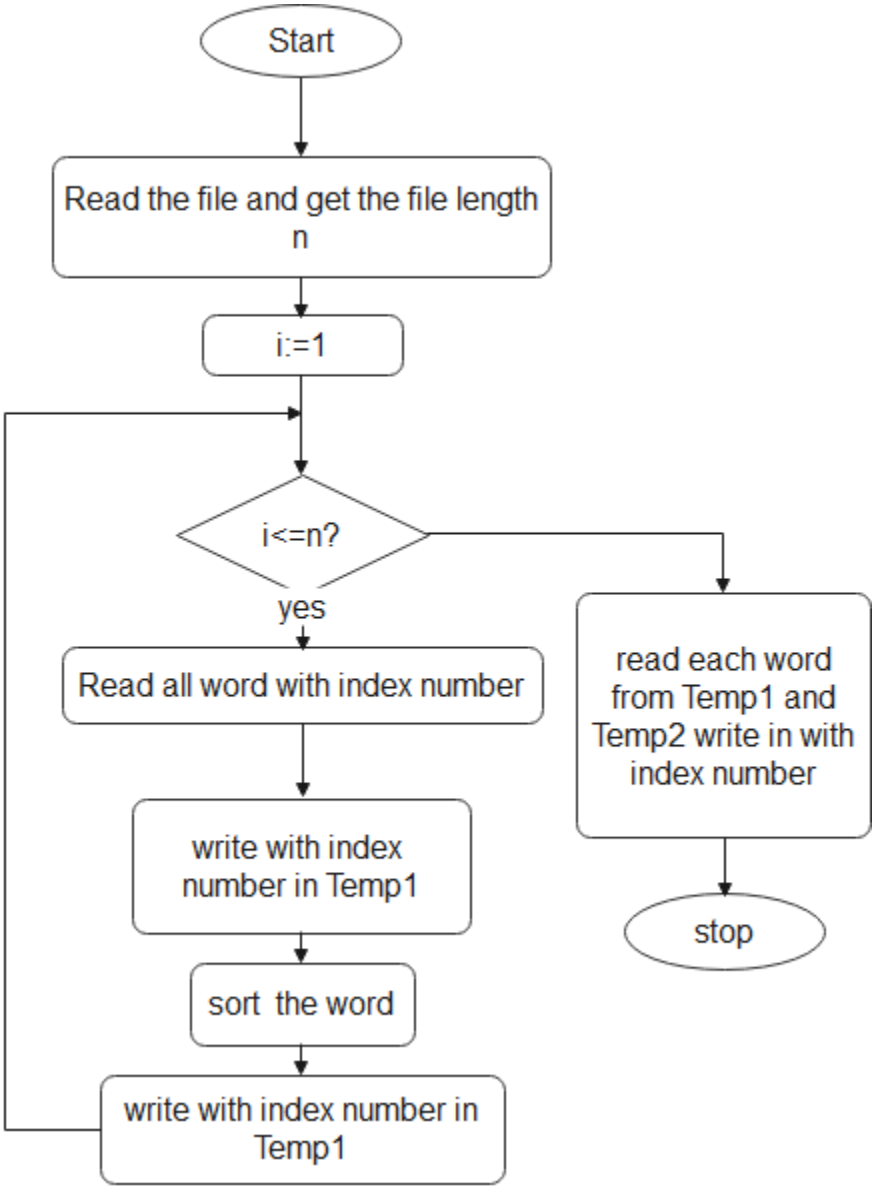

Figure 1: Sorted file that contain word by alphabetical with their index position.

Then after file have been sorted we can construct hash table from sorted temporary file and phrase needed to be searched in the original file. Fig-2 shows the flowchart of building complete hash table $Hash_t$ form temporary sorted Temp2 by adding the information of the locations of current and next-words and the number of occurrences of the current word in Temp2 and storing all the information as a record in the final hash-table, $Hash_t$. When building $Hash_t$, all the words in Temp2 are read until the end of the file is reached. Suppose that the current word which is read is Wj. Then the matching with W is searched in the sorted document Temp2. The number of matches (occurrences) F and the location P of each occurrence and the index value I of the next word of each occurrence, along with the index value of the current word and the word itself are stored as a single record in the hash table $Hash_t$. This is performed for all the words of Temp1 and finally File1 is completed.

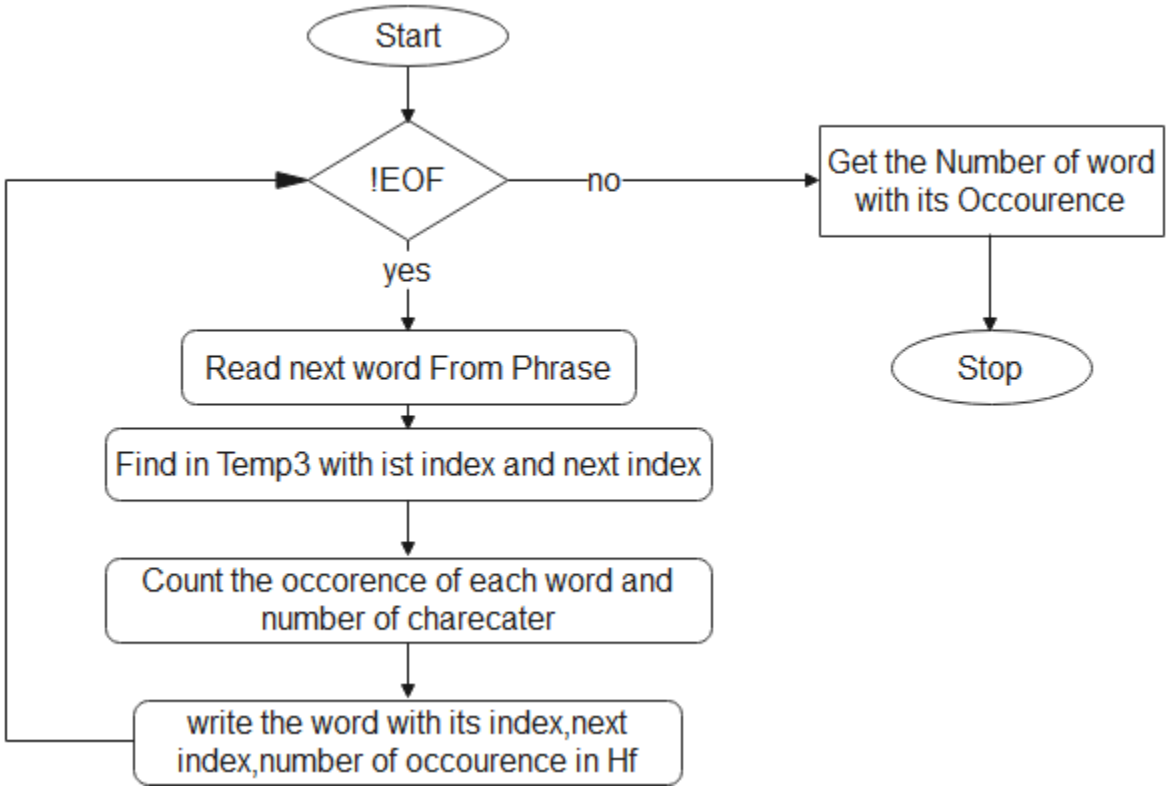

Figure 2: Constructed hash table from sorted temporary file and phrase

The flowchart shown in Figure 3 provides the search mechanism of the first word Wl of a sentence S in the hash table $Hash_t$. First index values are determined to set the values of the upper U and lower L bounds. When the first word W1 is matched with the first word W1 in hash table, then the index of W1 is taken as L. Similarly, the next word W1 is matched with the next word

W1 in hash table, then the index of W1 is taken as U. then if U is the next index of the first word its information is retrieved from hash table.

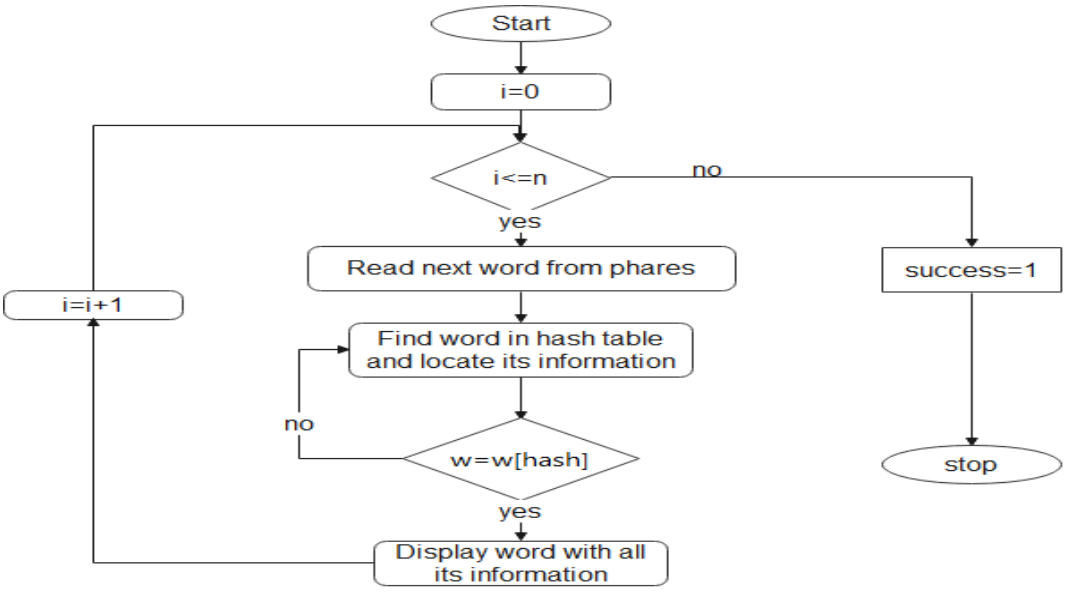

Figure 3: search the words form hash table.

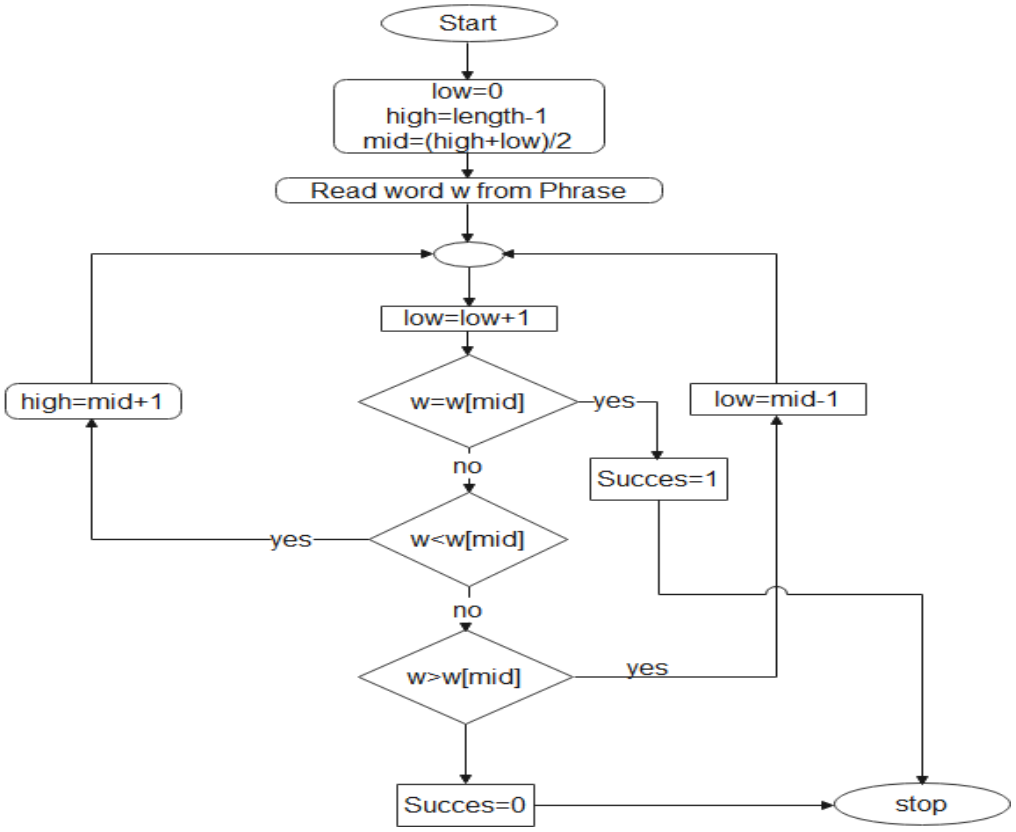

Figure 4: Binary search for the word to be searched after file has been sorted.

Now, an algorithm that employs the binary search method is used to determine the exact matching of WI in sorted Temp2 file. In this method, a midpoint m is calculated by dividing the sum of lowest and highest index, the highest index is the length of the file minus one and lowest index is index at zero. Then, the corresponding word W at the index m is extracted. If ASCII value of any character in W is less than corresponding value in W J, L is set as mid+ 1. If ASCII value of any character in W is greater than corresponding value in W J, U is set as mid-1.This process is repeated until L<U or W = W l. For the former case, success: = 0 meaning that the word W l is not found in Temp1. In the latter case, success is set as 1 i.e., WI is found in Temp1.

Note that: binary search method is used to determine the exact matching of word in hash table and it's also standalone method in this paper.

4. Data Collection & Analysis

When we analyze an algorithm in case of complexity it depends on the approach used by algorithm or program it needs the CPU execution or running time and amount of memory to run the program. In this paper for binary search algorithm iterative approach is were used for implementation. To make a performance analysis we were started with sorting the data by using python. Then, for each file testing is performed at beast, average and worst case to analysis the performance of the algorithm. The following table-3 is a running time collected during experimental test on three Afan Oromo text file while the size of file is increasing from 5MB,16MB and 32MB respectively.

File	**Time complexity**		
	Beast case	Average	Worst
File1	0.036980628967285156	0.5117030143737793	0.836519718170166
File2	0.06249594688415527	1.9842398166656494	3.1872758865356445
File3	0.2031102180480957	4.484098672866821	5.390216827392578

Table 3: Binary search recorded time complexity for three file.

4.1 Binary search

Binary search is a search algorithm that finds the position of a key or target value within an array. So, the most important and critical to know is the worst case that means when the data looking for is at the end of file or absolutely not found in the list. The following fig-5 is indicating the performance of the algorithm as a size of file is increasing

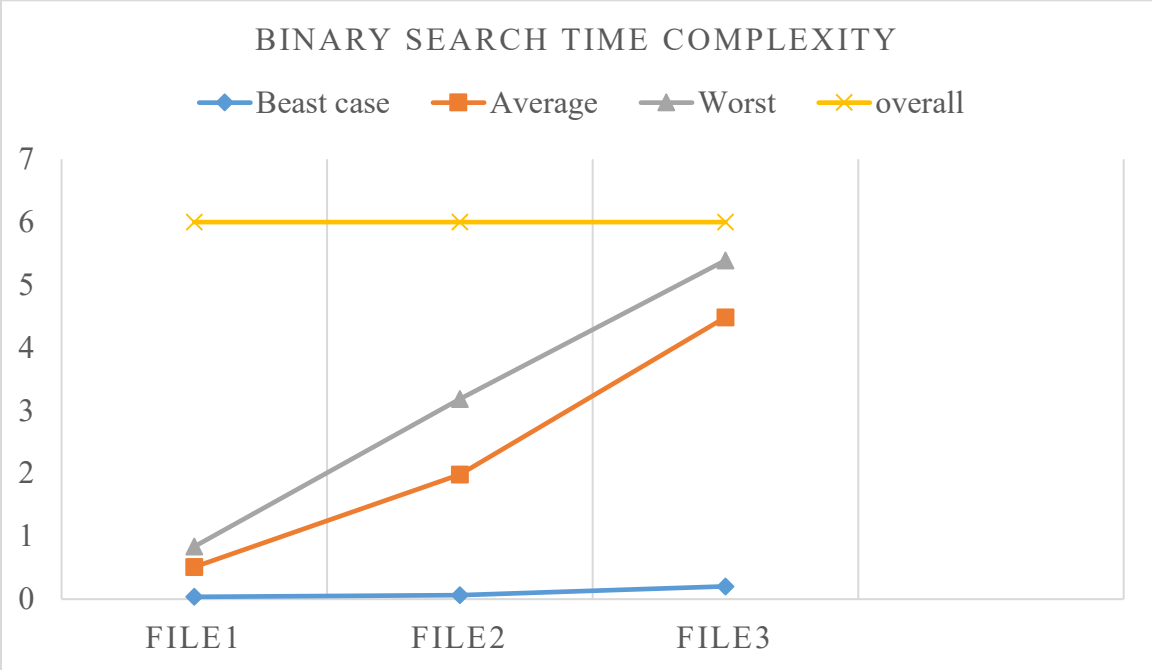

Figure 5: the performance of binary search algorithm as a size of file is increasing

In the above fig-5, it's clear that a running time of binary search for best case is when we find at the begging of file for all three file, where mid is the length of the file reduced by half. So, Time complexity is O (1), almost for all three file regardless of their size. Worst case: when we find the data at the end or not found. So, Time complexity is O (log n)). Form the figure as a file size is increasing the upper bound of O (1) is to at overall that is means O(n),but not exceeding the upper bound in any means. Time Complexity in the average case is almost equal to the running time of the worst case in the above figure. As a size of file is became large and large the performance of the algorithm is decreasing.

Space complexity of a binary Search, in iterative approach we have only the current element that are being compared with element in the data, if the current is not matched, the current element is discarded and take next element. So space complexity of binary search iterative

approach is the current element in byte (Size of a Value). So if we want to search word like "**Kitaabonni** ", each characters occupies 4 bytes then it will consume at least 40 bytes. Space complexity of a binary Search is O (1), it would mean that the space consumption grows linearly with the amount of elements in it.

4.2 Hash Table

For hash table first after sort the file we can start with building the hash table from phrase to be searched. Now we will show an example of how the hash table Hasht can be built from the following Phrase. That a file contains the following text.

Kitaabonni Afaan

We need to find a function h that can transform a particular key K, be it a string, number, record, or the like, into an index in the table used for storing items of the same type as K. To create a hash function, which is always the goal, the table has to contain at least the same number of positions as the number of elements being hashed. The following is a constructed hash table to apply the hash function method search for a given phrase.

Index no I	Word W	No of char N1	No ofoccur F	Pos of first W L	Pos of next W U
0	Kitaabonni	10	75	1017996	1017997
1	Afaan	3	285	1017997	1017998

Table 4: the constructed hash table from a given phrase.

Now the contents of Temp2 is expanded by adding the number of occurrences F and their locations of the sorted word W and the index value of next word U for each occurrence. Location of a word is given by the position in the phrase W. for the word "Kitaabonni" appears 75 times in the text. Hence the number of occurrence F= 75. It first appears with word position L= 1017996. The next word is at index I= 0 It first appears with word position L = 1017997, the next words have F= 285 number of occurrence, to find the exact location of the given phrase in the original file index, we compere the first word last index and the next word first index then if their

index is consecutive to each other like as in the original file then that index is returned else the words are there but not continuously putted all together in the original file. The following table-5 is the performed test that recorded running time of the Hash function test that is developed in python prototype.

File	Time complexity		
	Test1	Test2	Test3
File1	0.01565837860107422	0.015582799911499023	0.0029935836791992188
File2	0.03128314018249512	0.003998517990112305	0.031241893768310547
File3	0.04697227478027344	0.0781252384185791	0.0156252384185791

Table 5: hash table recorded time complexity for three file

It is obvious, in hash table there is no difference in beast case, average case and worst case, as a size of file is increasing hash table do the same performance in time complexity. The following fig-6 is the performance of hash table as a file size is increasing.

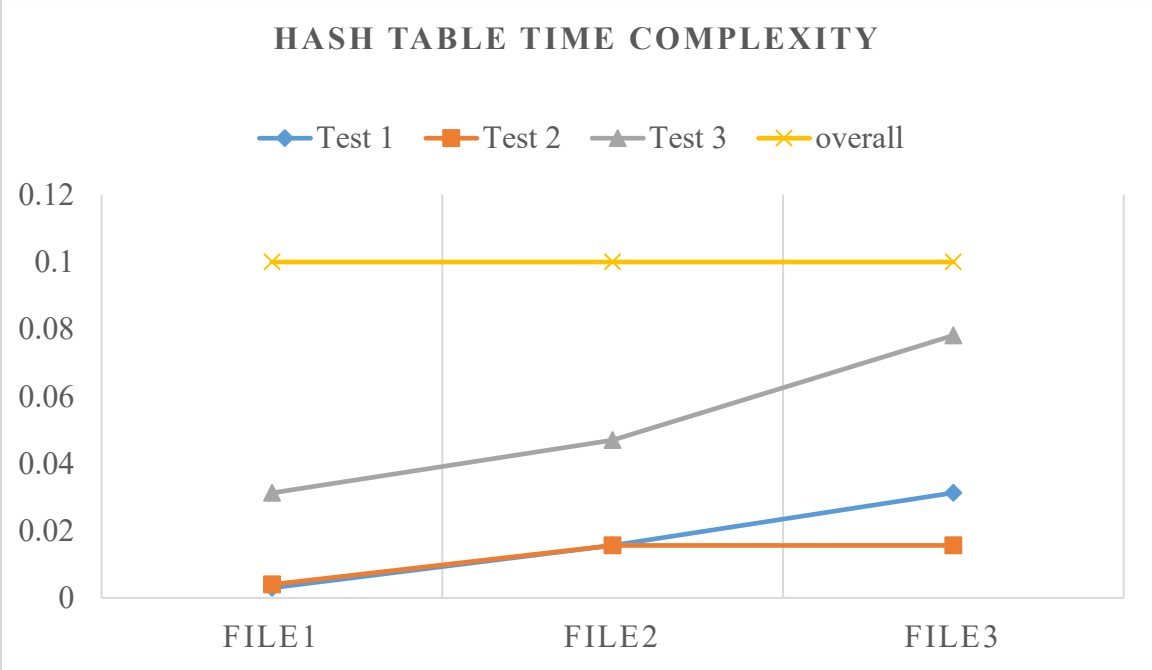

Figure 6: the performance of hash function algorithm as a size of file is increasing

We perform testing by selecting data at the begging, at the middle and at the end of file for making analysis. The reason is that as a file size is increasing the performance of hash function algorithm is not affected. Therefore, hash table is not analyzed in terms of best, average and worst case. In fig-6 Test1 and test2 is become overlapped at file1 and file2, however Test3 is making anther divergence as a file size is increasing. Test3 is different in other test not by the size of file but in case of the number of occurrence word given in a phrase, then in order to understand its overall performance we were used descriptive statistics for hash table mean, median and mode 0.02683, 0.01566 and 0.04697 respectively. The overall case in the figure is indicate that the mean of hash table performance is 0.02683,so the algorithm is grow as much to overall, that means typical order of upper bound of O(1),but not exceeding its hash table time complexity O(1).

Space complexity of a hash table is depends on the number of elements it currently stores and in real world also on the actual implementation. For in a constructed (Hasht) hash table a file size is 285+75=360 ,then the total memory requirement for the hash and phrase is (28647*4)= 1440 bytes while the for phrase to searched is (10*4) +(5*4) = 60bytes with other information being recorded in Hasht. A hash table typically has a space complexity of O (n).

5. Data Interpretation

A numerous method have been introduced in this problem area in last 10 years to make analysis, which searching algorithm is do have better Performance as a size of file is increasing. For this purpose, we try to review a number of related work on a Performance Analysis of binary search and hash table algorithm.

In [3] the authors presents a new technique of efficient search of a sequence of words in a large text file using the concept of hashing and linked list. They were make performance analysis in hash table by developing prototype. They were come up with result for selected algorithm in case of time complexity and space, total memory requirement is presented. However, in their approach searching for sentences in hash table is compression of each charterers in the hash table after file has been sort. In reality the searching sentences in the text means that all words are finding in the ordered they were in original text file not in alphabetical order, so after build the hash table we can find by their index rather than compering every words for searching. In [6] the proposed Search algorithm using blocks and words prefixes**.** The algorithm depends on the concept of word prefix, the letters at the beginning of the word. However, this algorithm is not a complete and

limited the length of words. Moreover, they was not mention about the space complexity, yet in their works.

In our proposed experimental setup we have been select tow searching algorithm namely binary search and hash function, the algorithm is implemented by python programing language by selecting the general problem solving algorithm divide and conquer because of efficient using our machine resource. For binary search iterative approach is were used to overcome the space complexity problem. However, the simplest and most straightforward way of solving a problem may not be sometimes the best one. The binary search was have a better space complexity because of the iterative approach we have been used but there is a divergence in a time complexity as a size of file is increasing. Another, approach was hash table that is do the same performance always regardless of size file. Even if the hash table do better in time, after hash table is constructed we need to use binary search for exact match of a given phrase or sentence. The typical orders it shows that the typical orders is increasing as the N become large, some of the algorithm is good when N is small, but other may not [7]. $O(1) < O(\log n) < O(n) < O(n\log n) < O(n2) < O(n3) < O(2n)$.

In general our proposed hash table have better performance in time and binary search have a better space complexity and also a binary search time Complexity in the average case is almost equal to the running time of the worst case.

6. Conclusion

In this paper, we illustrate the process of developing experimental for selected searching algorithm namely binary search and hash function. We also conduct a set of experiments to compare the performance of binary search and hash table. The result indicate that, binary search have a better space complexity because of the iterative approach we have been used, but there is a divergence in a time complexity as a size of file is increasing. Another, approach was hash table that have the same performance regardless of file size. We think this paper is suited for people to choice a particular algorithm they are interested in. However, the simplest and most straightforward way of solving a problem may not be sometimes the best one. The binary search have a better space complexity while hash table have better performance in time complexity. Besides, we will continue our research in learning more efficient searching algorithm. Since searching text in large file problem is involved in many application domains.

7. References

[1] A. Drozdek, Data Structures and Algorithms in C++,Fourth Edition, Cengage Learning., 2012 .

[2] Muhammad Nomani Kabir* , "Towards Optimal Search: An Efficient Search Algorithm for Arabic Text using Secant Method," in *2nd Borneo International Conference on Applied Mathematics and Engineering (BICAME)*, Kuantan, Malaysia, 2018.

[3] Muhammad N. Kabirl, Yasser M. Alginahil.3 and Omar Tayan1.Z, "Effcient Search of a sequence of words in a Large Text File," IEEE, Madinah, Saudi Arabia, 2014.

[4] V. V. Das, Principles of Data Structures using C and C++, New Delhi,India: New Age International(P)Ltd, 2006.

[5] Girma Debele Dinegde and Martha Yifiru Tachbelie, "Afan Oromo News Text Summarizer," *International Journal of Computer Applications (0975 – 8887),* Vols. Volume 103 – No.4,, pp. 1-6, October 2014.

[6] Khalid Thabit and Sumaia M. AL-Ghuribi*, "A new search algorithm for documents using blocks and words prefixes," *Academic Journal,* vol. 8(16), no. 27, pp. 40-648, April, 2013.

[7] T. H. Cormen*, Introduction to Algorithm, Londan,England: MIT Press: , 1990.

[8] Maryam Yammahi*, "An efficient technique for searching very large files with fuzzy criteria using the Pigeonhole Principle," *Article in Proceedings of the IEEE · ,* July 2014.

[9] "The Exact Online String Matching Problem: A Review of the Most Recent Results," *ACM Comput. Surv. 45, 2, Article 13,* p. 42, February 2013.

[10] S.-e. Yang, "A SEARCH ALGORITHM AND DATA STRUCTURE FOR AN EFFICIENT INFORMATION SYSTEM," Madison, Wisconsin.

[11] YUAN CAO*, "Binary Hashing for Approximate Nearest Neighbor Search on Big Data: A Survey," *Open Access IEEE,* Vols. VOLUME 6, , pp. 2039-2054, 2018.

8. Annex

Code

```
#This is the overall python library being used,their purpose is presented in the code

import json
import mmap
import os
import heapq as hp
import time as t
import numpy as np
import statistics as stat


        this method is used to sort the word for both algorithm
def Sortfile():
   with open('file3.txt.', 'r', encoding='UTF-8') as f: #open the file for reading all word
      datafile = f.readlines()
      data = len(datafile)
      print(data)
      value = []
      with open('temp1.txt', 'w', encoding='UTF-8') as f1:   #open the file for writing
         for line in datafile:
            value = line.split()
            json_str = json.dumps(value)              #json is used to process the data after reading it in a list
            response = json.loads(json_str)
            lan = len(value)
            for i in range(len(value)):
               print(response[i], ',', i, file=f1) #writing all word with its original index in to Temp1 temporary file
   with open('temp1.txt', 'r', encoding='UTF-8') as f1:
      sortd = f1.readlines()
      sorted_word = sorted(sortd)                              #sort all word and write in Temp2
      with open('temp2.txt', 'w', encoding='UTF-8') as f2:
         for l in sorted_word:
            value, index = l.strip().split(',', 1)
            print(value, ',', index, file=f2)

        #binary searching is started here

def binary_search(output, low, avarege, high, phares):
   start = t.time()                                          #this is the time where binary search is started for
   value = phares.strip().split(' ')
   valulang = len(value)
   for k in range(valulang):
      while low < high:
         if phares[0] == output[avarege][0]:            #compering the first char of Phrase
            if output[low].find(value[k]) != -1:         #camper the words with in original file
               print(output[low])
               print(t.time() - start, ',', low, file=f3)     #Report the running Time by current time –start time
                resource_usage()                          #call a resource_usage method for processing
               break
            else:
               low = low + 1
         elif phares[0] < output[avarege][0]: #compering the first char of Phrase if the words are not in mid
            if output[low].find(v[k]) != -1:       #camper the words with in original file
```

```
            print(output[low])
            print(t.time() - start, ',', low, file=f3) #Report the running Time by current time –start time
            resource _usage()                          #call a resource_usage method for processing
            break
         else:
            low = low + 1
         high = avarege - 1
         continue
      else:
         low = avarege
         avarege = high
         binary_search(output, low, avarege, high, phares)  #else call again method by second half
         break

def Build_hash(phares):                                    #start to buld hash table
   f4 = open(file='hashtime.txt', mode='w', encoding='UTF-8')
   with open('temp2.txt', 'r', encoding='UTF-8') as f2:
      outputs = f2.readlines()
      lengs = len(outputs)
      value = phares.strip().split(' ')
      valulang = len(value)
      with open('Hasht.txt', 'w', encoding='UTF-8') as f3:
         for k in range(valulang):
            count = 0
            indext = []
            for i in range(lengs):
               if outputs[i].find(value [k]) != -1:
                  word, findex= outputs[i].strip().split(',', 2)
                  print(findex, file=f3)
                  print(word, ',', findex,  file=f4)
                  count = count + 1                        #count No of occurrence word
               else:
                  continue
            print(value [k], count, file=f4)
            print(value [k], count, file=f5)


def searchs(phares):                       #searching is started here for a given phrase
   start = t.time()
   with open('Hasht.txt', 'r', encoding='UTF-8') as f3:
      fanal = f3.readlines()
      fanallang = len(fanal)
      finalsorted = sorted(fanal)
      end = fanallang - 1
      d = 0
      while d < end:
         if int(finalsorted[d + 1]) == int(finalsorted[d]) + 1:  #binary search mothed is used for exact macth for the index of words
            print(t.time() - start, file=f5)
            value= phares.strip().split(' ')
            valulang = len(value)
            for k in range(valulang):
               print(value[k], finalsorted[d])
            break
         else:
            d = d + 1
```

```
def resource_usage():                #used to Report the resources and calculate the average mean,…
    f3 = open(file='temp3.txt', mode='r', encoding='UTF-8')
 f5 = open(file='hello.txt', mode='a', encoding='UTF-8')
    memoryu = f3.readlines()
    h= []
    for l in memoryu:
        value, key = l.split(',',1)
        print(np.round(value,2,2), key)
        indexs=int(value)
        hp.heappush(h, indexs)
    mean_ value=stat.mean(h)
    median_ value=stat.median(h)
    mode_ value=stat.mode(h)
  print(mean_ value, median_ value, mode_ value,file=f5)

if __name__ == "__main__":  #main method is started this is the first code to be run
 f5 = open(file='hello.txt', mode='a', encoding='UTF-8')
f2= open(file='temp2.txt', mode ='w', encoding='UTF-8')
Sortfile()
phares = 'Kitaabonni Afaan '
output = f2.readlines()
leng = len(output)
print(leng)
avarege = leng // 2
low = 0
high = leng - 1
binary_search(output,low,avarege,high, phares)
build_hash(phares)
searchs(phares)
```