Comparative Analysis of Search Time Complexity Using Linear, Binary, and Hash Methods on GBK-Encoded Mandarin Characters

Raynard Fausta - 13524052
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: raynrdfaustasmlone@gmail.com , 13524052@std.stei.itb.ac.id

***Abstract*—This article compares and analyzes three search methods—linear search, binary search, and hash search—based on its time complexity when implemented to Mandarin characters encoded with GBK. An array of 1,000 Mandarin characters is used as the dataset for the search algorithms. Analysis of time complexity is done based on the search method implemented in the algorithms. Every algorithm is implemented in Python language to ensure fair condition for comparative analysis. Time measurements are collected with high-precision timers to assess the execution performance. The analysis results confirm the theoretical claims: linear search demonstrates O(n) complexity, binary search O(log n) starting from the second search, and hash search O(1). Although, actual execution time may vary from the theoretical claims as shown in the testing. This article offers insight for the development of Mandarin-to-other-languages dictionary software.**

Keywords—Mandarin; time complexity; search algorithm; linear search; hash search; binary search; encoding; GBK

I. Introduction

Mandarin is a very distinct language compared to other world languages which are mostly alphabetical languages. Mandarin is a logographic language, in which symbols are used to represent the meaning of the words entirely and have no direct correlation with the words' pronunciation.

As China is rising to be world's leading economy and technological leader, the use of Mandarin has been increasing year by year. In fact, according to research conducted by International Center for Language Studies in August 2024, Mandarin was ranked second as the world's most spoken language after English. Mandarin was spoken by 199 million people as a foreign language in 84 countries.

As the world is experiencing enormous effects of globalization, languages need to be encoded and digitally represented in order for them to be transferred across computers. The very first encoding method is the ASCII (American Standard Code for Information Interchange) method which uses binary code to represent a single character. ASCII is effective to encode most of world's languages as they are alphabetical. However, Mandarin is not an alphabetical language, therefore the ASCII method fails to encode Mandarin character.

In 1980, as an effort for the Chinese to join the globalization trend, they developed the GB2312(国家标准 2312/Guojia Biaozhun) which could encode 6763 Mandarin character(汉字).

However, GB2312 did not manage to encode some rare characters from Chinese names or classic Mandarin script. Therefore, the Chinese developed the GBK (国家标准扩展/Guojia Biaozhun Kuozhan). GBK then occupied critical role in Mandarin encoding development, addressing the limitations in the previous GB2312 standard and facilitating the transition to Unicode-compatible GB18030.

As Mandarin is accelerating to become the world's most used language, the most efficient search algorithm for Mandarin character is needed to enhance the speed of data processing and transferring. Therefore, the author is interested in conducting comparative analysis of search time complexity using linear, binary, and hash methods on GBK-encoded Mandarin characters.

II. Theoretical Background

A. Algorithm

Algorithms are sets of procedure or instruction used to solve a problem or to perform a computation.

Algorithms have several advantages such as consistent results for every repetition, scalability, which algorithms can handle large data and solve complex problems, automation, and standardization, which algorithms can be standardized and shared among people.

A good algorithm must be correct for every circumstance and efficient in solving a problem. Efficiency of algorithms can be measured from the usage of memory space or time taken to solve a problem. An efficient algorithm must solve a problem using as little memory space as possible and complete it in the shortest possible time. Both memory space and time requirement to solve a problem are measured from the size of the input.

B. Algorithm Complexity

Algorithm Complexity is the amount of memory space and time to solve a problem. There are two kinds of algorithms

complexity, time complexity and space complexity. The complexity of an algorithm is represented by a function which describes the algorithm's efficiency based on the size of data to be processed by the algorithm.

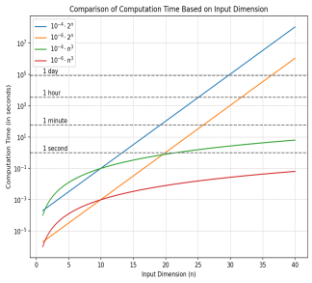

Figure 1. Comparison of Computation Time Based on Input Dimension [Source: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/25-Kompleksitas-Algoritma-Bagian1-2024.pdf]

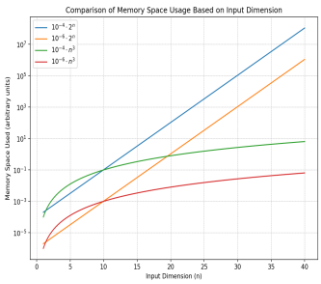

Figure 2. Comparison of Memory Space Usage Based on Input Dimension

C. Time Complexity

Time Complexity is the amount of time needed by the algorithm to solve a problem. This amount of time is not measured based on the execution time because different computers might take different time for an operation, as different computers use different machine languages. Different compilers also produce different machine languages.

Time Complexity is measured by calculating the amount of iteration and operations executed by the algorithms. Not all operations are calculated in time complexity.

Some of the typical operations which are calculated are:

- Write and Read Operation
  Example: **input(a)**, **print(a)** in Python
- Arithmetic Operation
  Example: n is **a+b** in Prolog
- Assignment Operation
  Example: int **a=0**; in C language
- Comparison Operation
  Example: **x>y** in most programming languages
- Array Accessing Operation
  Example: n=**a[0]** in Python
- Function or Procedure Calling Operation
  Example: **fungsikuadrat(2)**, **sort(array)** in C language
- Etc.

Some typical algorithms contain typical operations such as:

- Searching algorithm involves comparison operations
- Sorting algorithms involves comparison and swap operations
- Multiplication of matrixes algorithm involves arithmetic operations
- Value calculation of a polynomial involves arithmetic operations

There are three types of time complexities:

- $T_{max}(n)$: time complexity for the worst case which algorithms perform the maximum amount of work
  $T_{max}(n)=n$
- $T_{min}(N)$: time complexity for the best case which algorithms perform the minimum amount of work
  $T_{min}(n)=1$
- $T_{average}(N)$: time complexity for the average case

$$T_{average}(n)= \frac{1+2+3+4+..+n}{n} = \frac{\frac{1}{2}*n*(n+1)}{n} = \frac{n+1}{2}$$

or

$$T_{average}(n)= \sum_{j=1}^{n} T_j \, P(a[j]=x) = \sum_{j=1}^{n} T_j \frac{1}{n}$$
$$= \frac{1}{n}\sum_{j=1}^{n} T_j = \frac{1}{n}\sum_{j=1}^{n} j$$
$$= \frac{1}{n}\left(\frac{n*(n+1)}{2}\right) = \frac{n+1}{2}$$

The performance of algorithms can only be determined when the algorithm input size is large. Hence, asymptotic time complexity, an algorithmic time complexity notation used to describe the performance of algorithms for large input size, is needed.

There are 3 kinds of asymptotic time complexity:

1. Big-O Notation (Big-O)
   Big-O is used to compare several algorithms for a specific problem and determine the best algorithm based on time complexity.

Definition:
A function T(n) is O(f(n)), which f(n) represents the asymptotic upper bound of T(n), if for some constant c and n:

$T(n) \leq c*f(n)$, for $n \geq n_0$

f(n) is the upper bound of T(n) for large input size

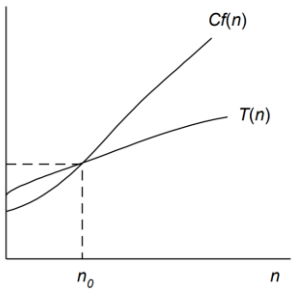

Figure 3. Illustration for Big-O Definition [Source https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/26-Kompleksitas-Algoritma-Bagian2-2024.pdf]

There are unlimited combinations of c and n value that satisfy $T(n) \leq c*f(n)$, however only one pair is needed to satisfy the Big-O definition.

1, n, $n^2, n^3, .., log\ n, n\ log\ n, 2^n, n!$ are some typical functions for f(n).

Theorem:
If $T(n)= a_m n^m + a_{m-1} n^{m-1} + .. + a_1 n + a_0$ is a polynomial with degree ≤ m, then $T(n)=O(n^m)$

Hence, finding the term with the greatest degree is sufficient to determine the Big-O notation.

Generalization from the first Theorem:

1. Exponential dominates every polynomial ($y^n > n^p, y > 1$
2. Polynomial dominates ln(n) ($n^p > ln\ n$)

3. All logarithmic functions grow at the same rate (a log(n) = b log(n))
4. n log n grows at faster rate than n, but slower than n^2

Theorem:
If $T_1(n)=O(f(n))$ and $T_2(n)=O(g(n))$, then:
a. $T_1(n) + T_2(n) = O(f(n)) + O(g(n)) = O(\max(f(n), g(n))$
b. $T_1(n) * T_2(n) = O(f(n)) * O(g(n)) = O(f(n) * g(n))$
c. $O(c*f(n)) = O(f(n))$, c is constant
d. $f(n) = O(f(n))$

Algorithm Group	Type
O(1)	Constant
O(log n)	Logarithmic
O(n)	Linear
O(n log n)	Logarithmic Linear
$O(n^2)$	Quadratic
$O(n^3)$	Cubic
$O(2^n)$	Exponential
O(n!)	Factorial

Table 1. Grouping of algorithms based on Big-O Notation [Source: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/25-Kompleksitas-Algoritma-Bagian1-2024.pdf]

The spectrum of algorithm time complexity, in order:

$$\underbrace{1 < \log n < n \log n < n^2 < n^3 < \ldots}_{\text{Polynomial algorithm (good)}} < \underbrace{2^n < n!}_{\text{Exponential algorithm (bad)}}$$

1	log n	n	n log n	n^2	n^3	2^n	n!
1	0	1	0	1	1	2	1
1	1	2	2	4	8	4	2
1	2	4	8	16	64	16	24
1	3	8	24	64	512	256	362880
1	4	16	64	256	4096	65536	20922789888000
1	5	32	160	1024	32768	4294967296	263130836933693530167218012160000000

Table 2. Value of Each Complexity Function with Varying n [Source: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/25-Kompleksitas-Algoritma-Bagian1-2024.pdf]

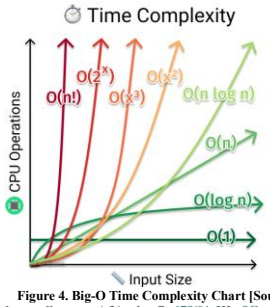

Figure 4. Big-O Time Complexity Chart [Source: https://miro.medium.com/v2/resize:fit:678/0*sHLx8GgoVyc4Ku2c.png]

- O(1)

The execution time of algorithm is constant and does not depend on input size. It is found in algorithms whose instructions are only executed once.

Example:

```
if (a>b):
    maks=a
else :
    maks=b
```

Figure 5. Example of Program with O(1) Complexity

- O(log n)

Algorithm with slower grow rate than n. It is found in algorithms that transform complex problems into several smaller subproblems with equal size.

Example:

```
function binarySearch(array, value) {
  var low = 0;
  var high = array.length - 1;
  var middle;

  while (low <= high) {
    middle = (low + high) / 2;
    if (array[middle] > value) {
      high = middle - 1;
    } else if (array[middle] < value) {
      low = middle + 1;
    } else {
      return middle;
    }
  }
}
```

Figure 6. Example of Program with O(log n) Complexity [Source: https://miro.medium.com/v2/resize:fit:868/1*WbGV7i6CyOQ5jt6hRnK2Zw.png]

- O(n)

Algorithm whose execution time grow linearly with respect to input size. Each input element undergoes a similar process.

Example:

```
function linearSearch(array,target) {
    for (var i = 0; i < array.length; i++) {
        if (array[i]===target) {
            return true
        }
    }

    return false
}
```

Figure 7. Example of Program with O(n) Complexity [Source: https://res.cloudinary.com/practicaldev/image/fetch/s--TFKIRFBW--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https://dev-to-uploads.s3.amazonaws.com/i/znn2e6wn2u9zkbiexflg.png]

- O(n log n)

Algorithm that divides complex problems into smaller subproblems and solves them independently and combining all the subproblems' solutions. It is found in algorithms that solve a problem with divide and conquer.

Example:

```
function mergeSort(array) {
  // Nothing to Sort here
  if (array.length === 1) {
    return array;
  }
  // Split Array into right and left
  const middle = Math.floor(array.length / 2);
  const left = array.slice(0, middle);
  const right = array.slice(middle);

  return merge(mergeSort(left), mergeSort(right));
}
```

Figure 8. Example of Program with O(n log n) Complexity [Source: https://media2.dev.to/dynamic/image/width=800%2Cheight=%2Cfit=scale-down%2Cgravity=auto%2Cformat=auto/https%3A%2F%2Fdev-to-uploads.s3.amazonaws.com%2Fi%2Fzkwfjqivcqr8ezewkk34.png]

- $O(n^2)$

Algorithm whose execution time grows quadratically with respect to input size. It is found in algorithms that process input elements in 2 nested loops.

Example:

```
void selection_sort (int A[ ], int size) {

    int minimum;

    for(int i = 0; i < size-1 ; i++)  {

        minimum = i ;

        for(int j = i+1; j < size ; j++ ) {
            if(A[ j ] < A[ minimum ])  {
            minimum = j ;
            }
        }
      swap ( A[ minimum ], A[ i ]) ;
    }
}
```

Figure 9. Example of Program with O(n²) Complexity [Source: https://miro.medium.com/v2/resize:fit:609/1*1pEjE1Bau34XnwOWbbwkog.png]

- $O(n^3)$

Algorithm that processes input elements in 3 nested loops.

Example:

```
SQUARE-MATRIX-MULTIPLY(A, B)
1  n = A.rows
2  let C be a new n × n matrix
3  for i = 1 to n
4      for j = 1 to n
5          c_ij = 0
6          for k = 1 to n
7              c_ij = c_ij + a_ik · b_kj
8  return C
```

Figure 10. Example of Program with O(n³) Complexity [Source: https://shivathudi.github.io/assets/square-multiply.png]

- $O(2^n)$

Algorithms that are categorized as brute-force method. It involves trials and errors in solving the problem.

Example:

```
# Input size of the set
size_of_set = int(input('Enter size of set: '))

# Input elements of the set (user input)
set1 = [input(f"Enter {i+1} element: ") for i in range(size_of_set)]

# Initialize the result with an empty subset
result = [[]]

# Generate all subsets
for i in set1:
    n = len(result)
    for j in range(n):
        r = result[j] + [i]
        result.append(r)

# Print all subsets
for element in result:
    print(element)
```

Figure 11. Example of Program with O(2ⁿ) Complexity [Source: https://allinpython.com/power-set-program-in-python/]

- O(n!)

Algorithm that processes each input element and links it with n-1 other elements.

Example:

Algorithm 2: Algorithm to compute an optimal solution to the TSP-D

Data: A set of locations V and a depot v_0

Result: A table $D(S, v)$ with optimal TSP-D paths that start at the depot v_0, end at location v and covers all locations in S.

```
D ← ∞ ;
D(∅, v_0) ← 0 ;
Precompute D_sv ;
for i = 1, ..., |V| do
    for U ⊆ V where |U| = i do
        for T ⊆ V \ U do
            for (u, w) ∈ U × V do
                z ← D(U, u) + D_sv(T ∪ {w}, u, w) ;
                if z < D(U ∪ {u, w} ∪ T, w) then
                    D(U ∪ {u, w} ∪ T, w) ← z ;
return D ;
```

Figure 12. Example of Program with O(n!) Complexity [Source: https://onlinelibrary.wiley.com/cms/asset/2ea50bb4-a37f-4f45-bab0-04265338b894/net21864-gra-0002.png]

2. Big-Omega Notation (Big-Ω)

Definition:

A function T(n) is Ω(g(n)), which g(n) represents the asymptotic lower bound of T(n), if for some constant c and n:

$$T(n) \geq c*g(n), \text{ for } n \geq n_0$$

3. Big-Theta Notation (Big- Θ)

If T(n) = Θ(h(n)), T(n)'s order is h(n).

Definition:

A function T(n) is Θ(h(n)), means h(n) represents both asymptotic upper and lower bound for T(n), if:

$$T(n) = O(h(n)) \text{ and } T(n) = \Omega(h(n)).$$

Theorem:

If $T(n) = a_m n^m + a_{m-1} n^{m-1} + .. + a_1 n + a_0$ is a polynomial with degree ≤ m, then T(n)'s order is n^m

D. Mandarin As Logographic Language

Mandarin characters (汉字) use strokes(笔画), which must be written in specific order(笔顺), to represent the meaning of a word.

序号	笔画	名称	例字	序号	笔画	名称	例字
1	丶	点	主	17	乙	横折弯钩	九
2	一	横	三	18	𠃌	横撇弯钩	那
3	丨	竖	十	19	𠃍	横折折折钩	奶
4	丿	撇	入	20	𠃎	竖折折钩	与
5	乀	捺	人	21	𠃏	竖弯	电
6	𠂉	横撇	有	22	𠃑	横折弯	没
7	𠂇	横点	女	23	𠃒	横折	口
8	亅	竖提	长	24	𠃓	竖折	山
9	𠂆	横折提	语	25	𠃔	撇折	云
10	𠄌	弯钩	了	26	𠃕	横撇	水
11	亅	竖钩	小	27	𠃖	横折折撇	及
12	𠃋	竖弯钩	乳	28	𠃗	竖折撇	专
13	𠃊	斜钩	戏	29	𠃘	撇斜钩	风
14	𠄎	卧钩	心	30	𠃙	竖折折	直
15	𠃌	横钩	写	31	𠃚	横折折	凹
16	𠃍	横折钩	力	32	𠃛	横折折折	凸

Figure 13. Table of 32 Mandarin Basic Strokes（笔画） [Source: https://www.hypy.com.cn/upfile/image/20210629/20210629174069056905.jpg]

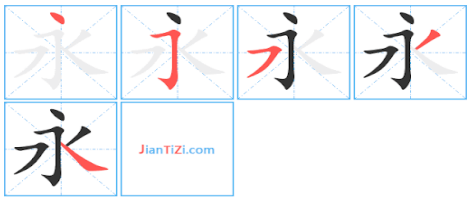

Figure 14. Stroke Order of a Mandarin Character [Source: https://bishun.ivtool.com/7738.htm]

There are six basics of how a Mandarin character is written:

1. 象形(Pictographs)

Mandarin characters are stylized or simple representations of the object.

Figure 15. Illustration of 象形(Pictographs) [Source: https://www.mandarinblueprint.com/blog/chinese-characters/#:~:text=Chinese%20characters%20are%20not%20words,but%20many%20others%20are%20not]

2. 指事(Ideographs)

Mandarin characters represent abstract concepts that could be understood intuitively.

Example:

上-above, 中-middle, 一-one

Figure 16. Illustration 指事(Ideographs) [Source: https://www.mandarinblueprint.com/blog/chinese-characters/#:~:text=Chinese%20characters%20are%20not%20words,but%20many%20others%20are%20not]

3. 会意(Compound Ideographs)

Mandarin characters are combinations of two or more Mandarin characters.

Example:

家-house, 尖-tip, 休-rest

Figure 17. Illustration of 会意(Compound Ideographs) [Source: https://www.mandarinblueprint.com/blog/chinese-characters/#:~:text=Chinese%20characters%20are%20not%20words,but%20many%20others%20are%20not]

4. 形声(Phonetic-Semantic Compounds)

Mandarin characters are combinations of semantic (represents meaning) and phonetic components (represents pronunciation).

Example:

- 炮 pào – firecracker, cannon ((火-fire) semantic; (包- bāo) phonetic)
- 晴 qíng – sunny, clear ((日-sun) semantic; (青- qīng) phonetic)

5. 转注(Transfer characters)

Mandarin characters are modified to create new Mandarin characters.

Example:

考 kǎo (to verify) and 老 lǎo (old)

6. 假借(Loan Characters)

Mandarin characters appear in other characters, which are unrelated, but with similar pronunciation.

Example:

哥 gē (older brother) and 歌 gē (song)

As Mandarin is not an alphabetical language, the organization of Mandarin characters in dictionary are based on 部首(radical) or 汉语拼音(Han Yu Pin Yin).

Figure 18. Most Common Radical [Source: https://img.alicdn.com/imgextra/i2/859515618/O1CN01fqZT9O1rN5nzGnvcL_!!0-item_pic.jpg_q50s50.jpg]

汉语拼音(Han Yu Pin Yin) is used to pronounce each Mandarin characters.

Figure 19. Mandarin Pin Yin Alphabet [Source: https://i.pinimg.com/736x/d2/81/3e/d2813ec3a9adf5a993b56b6d443e862f.jpg]

E. GBK Encoding

GBK(国家标准扩展/ Guojia Biaozhun Kuozhan), established in 1985, is an expansion of GB2312 which was only able to encode 6763 Mandarin character. However, it failed to encode some rare characters from Chinese names or classic Mandarin script. Both GB2312 and GBK Encoding were built with ASCII as their foundation of development. GBK Encoding then maintained backward compatibility with GB2312 encoding.

GBK Encoding uses double byte encoding scheme with a code range from 8140 to FEFE, however any combinations with 7F ending are not supported. GBK is able to encode 21,003 Mandarin characters, an increase from the previous 6763 characters. GBK encoding also covers 23940 code positions and supports CJK(Chinese, Japanese, Korean) Han characters.

GBK Encoding has similar codepoints to ASCII and GB2312, as shown below:

- 1-byte codes: {0x00-0x7F}
  Identical to ASCII codes

- 2-byte codes: [0x81-0xFE][0x40-0x7E] and [0x81-0xFE] [0x80-0xFE]
  GB2312 and additional characters

In Ascii code, its most significant bit in the first byte is 0, system will then process only one byte. If the most significant bit in the first byte is 1, then it is Mandarin character. Therefore, determining whether a byte is ASCII or Mandarin is easy.

The system will then process those 2 bytes by performing a lookup for (B1, B2) pair in the GBK mapping table.

GBK Map consists of 5 levels. Each contains specific ranges for lead and trail bytes.

Level	Lead Byte Range	Trail Byte Range	Potential Code Point
GBK/1	0xA1–0xA9	0xA1–0xFE	846
GBK/2	0xB0–0xF7	0xA1–0xFE	6768
GBK/3	0x81–0xA0	0x40–0xFE	6080
GBK/4	0xAA–0xFE	0x40–0xFE, except 7F	8160
GBK/5	0xA8–0xA9	0x40–0xA0, except 7F	192

Table 3. Level in GBK Encoding [Source: *Chinese Internal Code Specification (GBK) [Standard No. GBK n0278-1995]*]

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
9040	丗	丘	丙	业	丛	东	丝	丞	丢	两	严	丧	个	丫	中	丰
9050	串	临	丸	丹	为	主	丽	举	丿	乂	乃	久	么	义	之	乌
9060	乐	乔	乍	乒	乓	乎	乏	乐	乒	乓	乔	乖	乘	乙	乜	九
9070	乞	也	习	乡	书	乩	买	乱	乳	乾	了	予	争	事	二	
9080	于	亏	云	互	亓	五	井	亚	些	亟	亡	亢	交	亥	亦	产
9090	亨	亩	享	京	亭	亮	亲	亳	亵	人	亿	什	仁	仂	仃	仄
90A0	仅	仆	仇	仉	今	介	仍	从	仑	仓	仔	仕	他	仗	付	仙
90B0	仝	仞	仟	代	令	以	仨	仪	们	仫	仰	仲	仳	仵	件	价
90C0	任	份	仿	企	伈	伉	伊	伍	伎	伏	伐	休	众	优	伙	会
90D0	伛	伞	伟	传	伢	伤	伥	伦	伧	伪	伫	伭	伯	估	伴	伶
90E0	伸	伺	似	伽	伾	佁	佃	但	位	低	住	佐	佑	体	何	佗
90F0	佘	余	佚	佛	作	佝	佞	佟	你	佣	佤	佥	佩	佬	佯	

Figure 20. GBK MAP for 0x9040-0x90FF [Source: https://www.khngai.com/chinese/charmap/tblgbk.php?page=1]

III. METHOD

A program in Python will be implemented to analyze search time complexity for GBK-Encoded Mandarin characters using linear, hash, and binary search. The program will have an array of 1,000 Mandarin characters as its pre-condition and has been encoded with GBK encoding. The program receives input from users in Mandarin characters and performs searches using linear, binary, and hash methods. The corresponding GBK encoding will be used as the key during the process.

A time complexity analysis(theoretical) for the program is done to determine the Big-O notation for each method which will then be used to determine the most efficient search method. Program will also measure the execution time(empirical) for each search method to strengthen the hypothesis claim based on time complexity analysis.

To further strengthen the understanding of Mandarin character searching, a simple Mandarin-English dictionary program for beginner speakers in Python language is implemented using the most efficient search method among the three methods.

A. Dataset Preparation

1,000 Simplified Mandarin characters(汉字) will be placed as elements of array in Python programming languages. There is no specific category for assigning those 1,000 characters into the array. Array assigning is assisted by ChatGPT to ensure correctness and efficiency.

Source: https://www.chinesereadersguild.com/most-frequent-chinese-characters-1-1000/

B. GBK Encoding

Every Mandarin character will be encoded with GBK encoding by the Python program and stored accordingly in another array before proceeding to the search algorithm.

Python has a built-in GBK encoding system. Mandarin characters contain 2 bytes in its GBK encoding. The first byte (high byte) is left-shifted 8 times and is operated with “or” operators with the second byte (low byte), in order for them to be combined as an integer.

```
def gbk_encoding(array):
    encoded = []
    for hz in array:
        gbk_bytes = hz.encode('gbk')
        val = (gbk_bytes[0] << 8) | gbk_bytes[1]
        encoded.append(val)
    return encoded
```

```
seasons = ["春", "夏", "秋", "冬"]
```

```
D:\Phyton\Matematika Diskrit>python testing.py
gbk_seasons = [0xB4BA, 0xCFC4, 0xC7EF, 0xB6AC]
```

Figure 21. GBK Encoding in Python

C. Search Method Implementations

In every method, the same initial states occur in which every Mandarin character in initial array has been encoded with GBK encoding and results are stored in a separate array that only store GBK values before search is performed.

The program will receive a Mandarin character as input. The program then encodes the input(target) into GBK value.

- Linear Search

The program iterates every value in the array which stored the encoded GBK value. When the target value in GBK matches the one of the array element value, iterations will be stopped and "Found" message will be printed. Otherwise, there is no match for the target value after iterating every element and "Not Found" message will be printed.

```
target=input("Input a Han Zi (to end program input no) : ")

while (target!="no"):
    target_gbk=target.encode('gbk')

    target_val = (target_gbk[0] << 8) | target_gbk[1]
                 (function) def perf_counter() -> float
    found=0      Performance counter for benchmarking.
    start = time.perf_counter()
    for value in common_array_gbk:
        if (value==target_val):
            found=1
            break
    end = time.perf_counter()


    if found==1:
        print('Found')
    else:
        print('Not found')

    print(f"Search time: {end - start:.16f} seconds\n")
```

Figure 22. Linear Search Implementation in Python

➢ Binary Search

Array which stored GBK values will be sorted with merge sort algorithm ascendingly as array needs to be sorted before performing binary search. Left marker is assigned with the index 0 and right marker with array length-1. Search starts by comparing the value in the middle index with target value and the range is adjusted accordingly by shifting either the left marker or the right marker. Process repeats until the left marker is greater than right marker or target value has matched an array element.

"Found" message is printed when there is a match for target value otherwise "Not Found" is printed.

```
target=input("Input a Han Zi (to end program input no) : ")
target_gbk=target.encode('gbk')
target_val = (target_gbk[0] << 8) | target_gbk[1]
found=0

left=0
right=len(common_array_gbk)-1

start = time.perf_counter()
mergeSort(common_array_gbk)
while(left<=right):
    mid = (left+right)//2
    if (common_array_gbk[mid]==target_val):
        found=1
        break
    if (common_array_gbk[mid]>target_val):
        right=mid-1
    if(common_array_gbk[mid]<target_val):
        left=mid+1
end = time.perf_counter()

if found==1:
        print('Found')
else:
    print('Not found')

print(f"Search time: {end - start:.16f} seconds\n")
```

Figure 23. Binary Search with Merge Sort Implementation in Python

➢ Hash Search

Program builds a hash table with GBK value as the key and value is assigned with True as dummy because program only checks the target input's existence in the array. The program proceeds to look-up the target value in the hash table.

"Found" message is printed when there is a match for target value in the hash table otherwise "Not Found" is printed.

```
target=input("Input a Han Zi (to end program input no) : ")
target_gbk=target.encode('gbk')
target_val = (target_gbk[0] << 8) | target_gbk[1]

start = time.perf_counter()

#Hash Table Build
hash_table = {val: True for val in common_array_gbk}

if target_val in hash_table:
    print("Found")
else:
    print("Not found")
end = time.perf_counter()
print(f"Search time: {end - start:.16f} seconds\n")

target=input("Input a Han Zi (to end program input no) : ")
```

Figure 24. Hash Search Implementation in Python

D. Search Time Measurement

Time library is imported into the Python code to measure the search time. Start time marker recorded immediately before the search is about to start and the end time marker recorded immediately after the search has ended.

In linear search method, the start time marker is placed before program goes into iteration and the end time marker is placed after the iterations stop.

In binary search method, the start time marker is placed before the program sorts the array of GBK value and end time marker is placed after the iterations stop.

In hash search method, the start time marker is placed before the programs build hash table and then end time marker is placed after the checking of target value in the hash table.

The difference between the end time and start time is the program execution time.

E. Empirical Testing

Mandarin Character	GBK Encoding Value
试	0xCAD4
印	0xD3A1
快	0xBFEC
蓝	0xC0B6
火	0xBBF0
茶	0xB2E8
灯	0xB5C6
巧	0xC7E3
我	0xCED2
爱	0XB0AE
隆	0xC1FA

Table 4. Inputs for Empirical Testing

Each algorithm will be tested with the same 10 inputs sequentially and its execution time will be recorded and compared with other algorithms. Recorded time execution has the precision of 16 decimal places.

IV. Results and Discussion

A. Algorithm Time Complexity Analysis (Theoretical)

1. Linear Search

Linear search iterates each element of array until it finds the matching value for the target GBK value or reaches the end of the array. In each iteration, one comparison operation (if (value==target_val)) is done to check whether the current element of array has matched the target GBK value.

In its best case the minimum iteration needed is only 1 iteration or $T_{min}(n)=1$. In this case, the first element directly matches the target GBK value.

In its worst case the maximum iteration needed is n iteration(s) which n is the number of elements in the array. In this case, matching target value is found at the very last element of array.

The average case for linear search with n element(s) is:

$$T_{avg}(n) = \frac{1+2+3+4..+n}{n} = \frac{\frac{1}{2}*(n)*(n+1)}{n} = \frac{n+1}{2}$$

Linear search Big-O asymptotic time complexity is:

$T(n) = \frac{n+1}{2} \leq 2 * n$ for C = 2, f(n) = n and $n \geq 1$

Hence, the Big-O complexity for linear search is O(n).

2. Binary Search

Arrays are sorted with merge sort algorithms before binary search algorithm is implemented.

The average, best, and worst case for merge sort are the same because merge sort continuously divides arrays into 2 parts, despite array might have been sorted in the middle of the process. It then merges all the subarrays in a process that involves every element.

Merge sort recursively divides arrays into half and the total operation for this process is log (n). Each subarray is then merged in a process that touches every element, and this process involves n operation(s). The total operation for merge sort is n * log (n). Hence, the time complexity(T(n)) for merge sort is n log n.

After the array is sorted, binary search recursively divides arrays into intervals with left and right marker until the middle element eventually matches the target GBK value or the last element to be proceeded.

$$1 = \frac{N}{2^x}$$
$$2^x = N$$
$$x \log 2 = \log N$$
$$x = \log_2 N$$

N= array size; x=number of processes

Hence, the time complexity(T(n)) for binary search is log n.

For the first search, array must be sorted before the search hence the time complexity for the first search is

$T_{first}(n) = T_{sort}(n) + T_{search}(n) = n \log n + \log n$

The Big-O asymptotic time complexity for the first search is:

$T_{first}(n) = n \log n + \log n \leq 2 * (n \log n)$ for C = 2, f(n) = n log n, and $n \geq 1$

Hence, the Big-O complexity for the first search is O(n log n).

However, starting the second search, algorithms do not perform sorting algorithms. Therefore, the time complexity is reduced to T(n) = log n and its Big-O complexity is

$T_{second}(n) = \log n \leq \log n$ for C = 1, f(n) = log n, and $n \geq 1$

Hence, the Big-O complexity starting the second search is O(log n).

3. Hash Search

Elements of array are mapped into a hash table by Python dictionary which GBK values are assigned as keys and values are assigned True as dummy. GBK value is ensured to be unique as each Mandarin character has its own GBK encoded value. Python then internally applies its hash function to manage key placement and lookup efficiency. This process is done n times as it has to map every element in the array and assigns them with True; hence it has the time complexity(T(n)) of n.

Search is then performed by looking up whether the target GBK value is in the hash table in 1 process. Therefore, the time complexity(T(n)) for this process is 1.

For the first search, elements must be mapped into hash table before the hash table lookup, hence the time complexity for the first search is

$T_{first}(n) = T_{map}(n) + T_{lookup}(n) = n + 1$

The Big-O asymptotic time complexity for the first search is:

$T_{first}(n) = n + 1 \leq 2 * n$ for C = 2, f(n) = n, and $n \geq 1$

Hence, the Big-O complexity for the first search is O(n).

However, starting the second search, algorithms do not perform mapping algorithms. Therefore, the time complexity is reduced to T(n)= 1 and its Big-O complexity is

$T_{second}(n) = 1 \leq 1$ for C=1, f(n)= 1, and $n \geq 1$

Hence, the Big-O complexity starting the second search is O(1).

Theoretically the most efficient search method starting for the second search determined based on the time complexity is :

Hash Search O(1) > Binary Search O(log n) > Linear Search O(n)

Hash search is theoretically the most efficient search method with the Big-O complexity of 1, meaning the time needed to complete the computation is similar for every number of inputs and does not depend on input size.

B. Empirical Testing Results

Each input is run with the same Python interpreter.

Mandarin Character	Execution Time (s)
试	0.0000633000163361
印	0.0001500000362284
快	0.0000565000227652
蓝	0.0001567000290379
火	0.0000839999993332
茶	0.0001755999983288
灯	0.0001760000013746
巧	0.0001110999728553
我	0.0000062999897636
爱	0.0000184000236914

Table 5. Linear Search Testing Execution Time

Mandarin Character	Execution Time (s)
试	0.0048413000185974
印	0.0000170999555849
快	0.0000746999867260
蓝	0.0000155000016093
火	0.0000185000244528
茶	0.0000219999928959
灯	0.0000166000099853
巧	0.0000147999962792
我	0.0000171000137925
爱	0.0000181000214070

Table 6. Binary Search Testing Execution Time

Mandarin Character	Execution Time (s)
试	0.0006209000130184
印	0.0002829000004567
快	0.0002565999748185
蓝	0.0002490999759175
火	0.0002980999997817
茶	0.0003375000087544
灯	0.0003679000074044
巧	0.0002389000146650
我	0.0002282999921590
爱	0.0003244999679737

Table 7. Hash Search Testing Execution Time

Average time for Linear Search: 0.0000997900089715 s (99.7 μs)
Average time for Binary Search: 0.0005055700021330 s (505.6 μs)
Average time for Hash Search: 0.0003204699954949 s (320.4 μs)

Empirically the fastest search method is the linear search method with average of 99.7 μs, followed by hash search with average of 320.4 μs, and binary search with average of 505.6 μs.

Empirical results strengthen the claim that time complexity for the first binary search is more complex than the second binary search. As O(n log n) is a more complex algorithm than O(log n).

Time taken for the first binary search: 4.84 ms
Time take for the second binary search: 17μs

Hash search complexity for the first search is also proven to be more complex than the second search. The first search requires additional time to construct the hash table (O(n)), while the next searches benefit from constant lookups time(O(1)).

Time taken for the first hash search: 620.9 μs
Time taken for the second hash search: 282.9 μs
Time taken for the third hash search: 256.6 μs

V. CONCLUSION

The experimental comparative analysis successfully proved the theoretical characteristics of the three search methods. Linear search is proven to be consistent with its theoretical O(n) complexity as the algorithms' time complexity increases with respect to the input size. Binary search time complexity for the first search is O(n log n) causing it to be more complex at first, however starting from the second search the complexity drops to O(log n), making it an efficient search method for large input size. Hash search is a very consistent searching method as it is not affected with the input size (O(1)), however the first search might be more complex as it needs to build the hash table which has the complexity of O(n). The characteristic of hash complexity is displayed from constant execution time starting from the second search.

Practically, the performance of each algorithm may vary based on input size, input frequency, and processor specifications. Therefore, while hash search is ideal for handling large-scale input size and frequent lookups, binary search offers a good balance between speed and setup complexity. Linear search remains reliable for small input size. Overall, this article underscores the importance of algorithm selection in handling Mandarin character data effectively.

As GBK are able to encode 21,003 characters, the most efficient search method to be implemented in Mandarin to other languages dictionary software development is hash search. With a time complexity of O(1), hash search offers

fast and consistent lookup performance regardless of the Mandarin character which is being searched. As GBK encoded values are also unique, it can be utilized as keys in building hash table.

VI. Appendix

Video Link: https://drive.google.com/file/d/10J-EFp0Mix4iaG1zlqmICsy1mQ1Vvn0-/view?usp=sharing

Duration: 32 minutes 42 seconds

Programs used in Analysis: https://github.com/RayapSunggal/13524052-Raynard-Fausta/tree/main

VII. Acknowledgment

The author sincerely thanks God Almighty for providing the strength and opportunity to complete this article successfully. The author also extends his deep appreciation and gratitude to Mr. Arrival Dwi Sentosa, S.Kom., M.T., lecture of K02 IF1220 Discrete Mathematics, whose semester-long support enabled the smooth completion of this article. The author would also like to extend his personal appreciation to Wowkie Zhang (阳光彩虹小白马) and to EggPlantEgg (浪子回头), whose music has accompanied and uplifted the author during the preparation of this article

VIII. References

[1] Awati, Rahul and Peter Loshin. 2025. *ASCII (American Standard Code for Information Interchange)*. TechTarget. Retrieved June 16, 2025, from https://www.techtarget.com/whatis/definition/ASCII-American-Standard-Code-for-Information-Interchange

[2] 计算中心. 2015. *GB2312-80 编码表(汉字机内码)*. 成都信息工程大学. Retrieved June 16, 2025, from http://jszx.cuit.edu.cn/NewsCont.asp?type=1009&id=20566

[3] 计算中心. 2015. *GBK 编码*. 成都信息工程大学. Retrieved June 16, 2025, from http://jszx.cuit.edu.cn/NewsCont.asp?bm=00&type=1009&id=20567

[4] Anubhavgoel1. 2023. *Definition, Types, Complexity and Examples of Algorithm*. GeeksforGeeks. Retrieved June 16, 2025 from https://www.geeksforgeeks.org/what-is-an-algorithm-definition-types-complexity-examples/

[5] Krantz, Tom. 2025. *What is a brute force attack?*. IBM. Retrieved June 17, 2025 from https://www.ibm.com/think/topics/brute-force-attack

[6] Munir, Rinaldi. 2025. *Kompleksitas algoritma (Bagian 1)*. Informatika STEI ITB. Retrieved June 17, 2025 from https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/25-Kompleksitas-Algoritma-Bagian1-2024.pdf

[7] Munir, Rinaldi. 2025. *Kompleksitas algoritma (Bagian 2)*. Informatika STEI ITB. Retrieved June 17, 2025 from https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/26-Kompleksitas-Algoritma-Bagian2-2024.pdf

[8] *Understanding Chinese Characters: the Basics You Need to Know*. (n.d.). MANDARIN BLUEPRINT. Retrieved June 17, 2025 from https://www.mandarinblueprint.com/blog/chinese-characters/#:~:text=Chinese%20characters%20are%20not%20words,but%20many%20others%20are%20not

[9] *ASCII，GBK，和 Unicode 的 UTF-8，UTF-16，UTF-32 阐述*. (n.d.). 城南花已开. Retrieved June 17, 2025 from https://www.cnblogs.com/cnhyk/p/12284020.html

[10] National Information Technology Standardization Technical Committee of China. 1995. *Chinese Internal Code Specification (GBK) [Standard No. GBK n0278-1995]*. China Electronics Standardization Institute.

[11] Ďurišinová, Martina. 2025. *Understanding The Different Types of Search Algorithms*. Luigi's Box. Retrieved June 18, 2025 from https://www.luigisbox.com/blog/types-of-search-algorithms/

[12] *Hashing in Data Structure*. 2025. GeeksforGeeks. Retrieved June 18, 2025 from https://www.geeksforgeeks.org/dsa/hashing-data-structure/

[13] *Merge Sort Algorithm*. (n.d.). Progamiz. Retrieved June 18, 2025 from https://www.programiz.com/dsa/merge-sort

[14] *Hashing in Data Structure*. 2025. GeeksforGeeks. Retrieved June 18, 2025 from https://www.geeksforgeeks.org/dsa/hashing-data-structure/

[15] *Dictionaries in Python*. 2025. GeeksforGeeks. Retrieved June 18, 2025 from https://www.geeksforgeeks.org/python/python-dictionary/

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 20 Juni 2025

Raynard Fausta
13524052