# Search Smarter, Not Harder: A Comparative Study of Search Algorithms in C++

Amr Eid , Ismail ElDahshan, Mohamed Azouz, Omar Mohammed

Department of Computer Science and Engineering, The American University in Cairo

CSCE 1102: Fundamentals of Computing II Lab

Dr. Esraa Hamdi

May 13, 2023

# TABLE OF CONTENTS

# ABSTRACT

This comparative study explores different search algorithms in C++ and their time complexities. The four main search algorithms discussed are Binary Search, Jump Search, Exponential Search, and Hashing. Binary search is efficient for sorted arrays with a time complexity of O(log n), while Jump Search divides the data into blocks for faster searching with a time complexity of O(n/m + (m-1)). Exponential Search is useful for unbounded data sets with a time complexity of O(log n). Hashing provides constant time complexity O(1) in the best case, but it can vary based on the load factor. The study conducts experiments to analyze the execution time and scaling behavior of each algorithm for different input sizes, aiming to identify their efficiency and help choose the appropriate algorithm for specific cases.

- Keywords: Searching Algorithms, Hashing Search, Binary Search, Jump Search, Exponential Search, Time complexity.

# INTRODUCTION

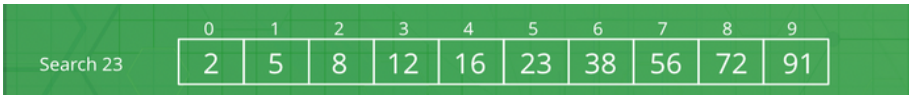
In computer science, thinking of the term “search” guides you towards an important topic: searching algorithms. But what exactly is a searching algorithm? A searching algorithm is an algorithm that is used to search for or extract an element from any data structure that it is contained in. In C++, there are four main search algorithms: Binary Search, Jump Search, Exponential Search, and Hashing. Binary search is used in sorted arrays, it performs searching by repeatedly dividing the search interval in half until the target element is found. Binary search has a time complexity of O(log n), which makes it much faster especially for large arrays. As the size of the array increases, the time it takes to perform a binary search increases logarithmically. Jump search has a very similar use to binary search since it is used in sorted arrays, which also makes it very efficient for large sets of data. It performs its function by dividing the data into blocks and then jumping through these blocks until the target element is found. The optimal size of a block to be jumped is ($\sqrt{n}$). This makes the time complexity of Jump Search $O(\sqrt{n})$ , which makes it slower than binary search. The exponential search, which is a variant of the binary search, is useful for unbounded or infinite data sets. It involves two steps: first it finds the range in which the element is present, then it performs in the range found . Exponential search has a time complexity of O(log n), which is the same as the binary search time complexity, since it is a variant of it, thus making it very efficient for large data sets. Among the four searching algorithms, the only searching method that does not require the array to be sorted is searching through Hash Tables. Hash Tables have a complexity of O(1) in its best case, However, This complexity may vary depending on the load factor. The load factor is the number of occupied spots in the array over the array size. The worst case of searching through hash tables is searching for an element that does not even exist in an array of load factor 1, in this case, the complexity would be O(N). Each of these searching algorithms has specific cases to be applied in, advantages that help distinguish each searching algorithm from the other and specific time complexities as well. Over the course of the paper,

the four searching algorithms will be discussed in detail, in order for you to be able to distinguish and choose the appropriate searching algorithm for the case at hand.

## BACKGROUND

### Binary Search:

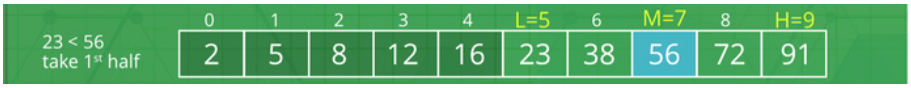
Searching for a desired value in a sorted array can be made more efficient using binary search, a fundamental algorithm technique. By reducing the search space in half with every comparison, binary search employs a divide-and-conquer approach, allowing for a time complexity of O(log n) for any given array of size n.




*Array of elements. Search key = 23*


*$1_{st}$ iteration of the algorithm*

The binary search begins by comparing the target value to the array's middle element, completing the search if these two values are the same. Conversely, should the target value be less than the middle element, the only array area that needs to be searched is the left half. We can limit our search if the middle element exceeds the target value; in this case, we only have to search the left half of the array. We repeat this procedure on the appropriate half until we discover the target value or conclude it is not present in the array.




*$2_{nd}$ iteration of the algorithm*


*$3_{rd}$ iteration of the algorithm. Search key is found*

Some drawbacks accompany the potent binary search algorithm. Its power is restricted since a prerequisite for the binary search to function properly is that the array is sorted. When the array isn't sorted, sorting it must first be done, leading to a time complexity of O(n log n). Also, the binary search may not be the optimal pick for small arrays since the benefits of the algorithm's efficiency might be counteracted by the overhead of space division during the search.

## Jump Search:

Jump search is one of many search algorithms which can be used to find an element in a sorted array. Jump search is similar to linear search, however it uses a jump value to narrow the part of the array being searched. The first step in the algorithm is determining the jump (or step) value. Jump search needs a jump value as it uses it to make sure it only has to search a small block of the array rather than the whole array. It will first check for the key at the index 0, then 0 + m (where m is the step), and keep going until array[km] > key or until the end of the array is reached and no element greater than the key was found. Once an element is found which is greater than the key, a linear search is performed between the index at which that element is present and the last index searched.

| 3 | 14 | 19 | 23 | 29 | 32 | 34 | 38 | 46 | 47 | 51 | 53 | 58 | 72 | 83 | 88 | 92 | 94 | 99 | 102 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] | [9] | [10] | [11] | [12] | [13] | [14] | [15] | [16] | [17] | [18] | [19] |

*Figure showing the array we will use*

For the above array, there would initially be a variable (low) set to 0, and high which will also be set to 0. The element at the index high would be compared with the key. If they are equal, then the index 0 would be returned. If not, low is set to equal high,and high is incremented by the value of the step. For this example, we will set m equal to 4, and we will set the key as 51.

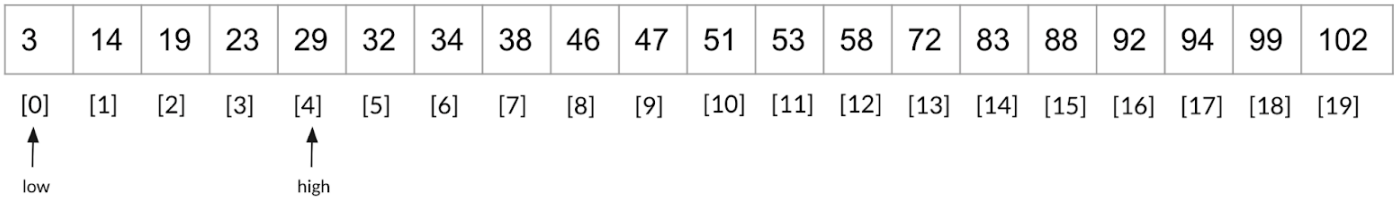



*Figure showing step 1 of Jump Search*

We will then repeat the same process: compare the value array[high] to the key. Since it is still not greater than or equal, we will set low = high (now 4) and then increment high by the step value.

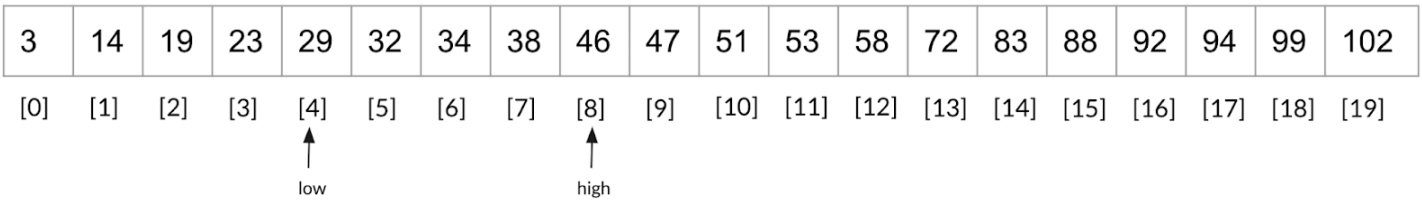



*Figure showing step 2 of Jump Search*

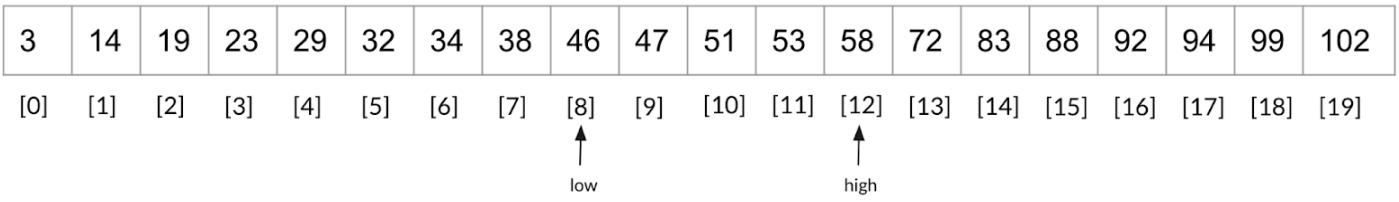



*Figure showing step 3 of Jump Search*

Since the element at the current index, 12, is greater than our key, 51, we will then perform a linear search starting from the index low + 1 as the value at index low has already been searched.

For an array of size n and with a jump value of m, the number of blocks can be calculating by dividing n by m (n/m). Once the correct block is found, a linear search will be performed on the number of elements in that block (m) except the first value as it has already been checked. This means we will search in (m-1) elements. This brings the time complexity of the Jump Search algorithm to O(n/m + (m-1)).

A very important step in jump search is optimizing the jump value. By conducting some basic mathematical calculations using differentiation, it was found that the optimal jump value is the square root of n, where n is the size of the array.

Jump search is very similar to linear search. The main advantage it has over linear search is that it will divide the elements of the array into blocks, which means it will not be searching in all the elements of the array. There are many other search algorithms, such as binary search, which are much more efficient. For large arrays, jump search could slow down a program significantly. It also only works for sorted arrays, which for extremely large data sets, may be inconvenient.

## **Exponential Search:**

A sorted, unbounded list can be searched for a specific input value (the search "key") using exponential search. There are two steps to the algorithm. The first step establishes the area where the search key would fall if it were included in the list. On this range, a binary search is carried out in the second stage. Exponential search finds the range by, exponentially increasing the powers of the base two. The search is carried out on the

blocks that have been divided using pow(2, k), where k is an integer higher than or equal to 0. The current block is subjected to binary search once the element at location pow(2, n) is greater than the key element.

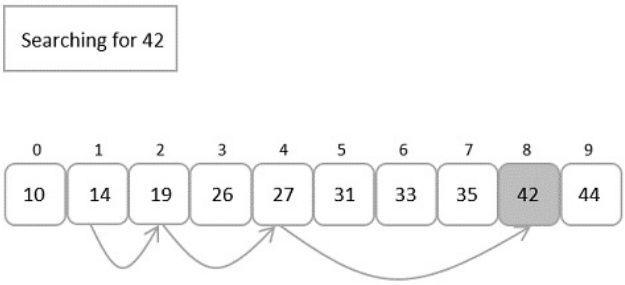



*Figure representation of how Exponential search works if the key element is found directly.*

In some cases, the element is not directly found and thus sometimes binary search will need to be employed.

For example, let's search for the position of element 81 in the given array.

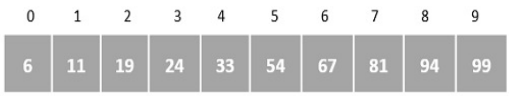



*Figure showing given array that is gonna be used for exponential search*

First step, comparing the array's initial element to element 81, the key element. The array's initial element is 6, but the key element to be searched is 81; as a result, the jump begins at the first index because no match was discovered.

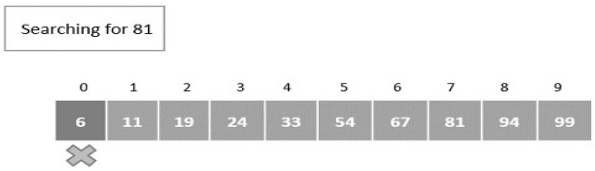



*Figure explaining step 1 of exponential search*

Second step, the key element is compared with the element in the first index after initialising i = 1. The key element in this case does not match the element in the first index. As a result, it is increased by an exponential factor in powers of 2. The element in the second index is compared with the key element after the index has been increased to 2^m = 2^1. It is again increased because there is still no match.

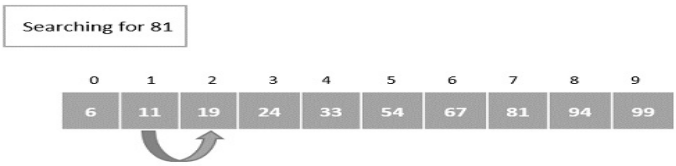



*Figure explaining step 2 of exponential search*

Third step, once more, the index is increased in powers of 2. 2^2 = 4, thus, the element in the fourth index is compared to the key element and no match has yet been discovered.

*Figure explaining step 3 of exponential search*

Fourth step, once more, the index is increased exponentially. This time, comparing between the element in the eighth index and the key element, again a match is not discovered. The element in the eighth index, however, is bigger than the key element. Thus, the current block of elements is subjected to the binary search algorithm.

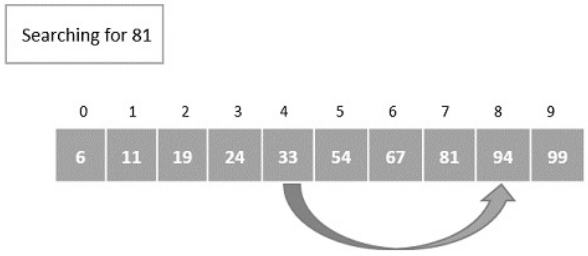



*Figure explaining step 4 of exponential search*

Fifth step, since the element we are standing at is greater than the key element, we can now implement binary search. The current block of elements includes the elements in the indices [4, 5, 6, 7].

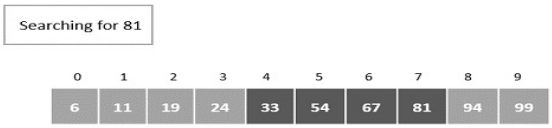



*Figure explaining step 5 of exponential search*

Binary search is now applied on these elements,where the mid is calculated to be the 5th element.

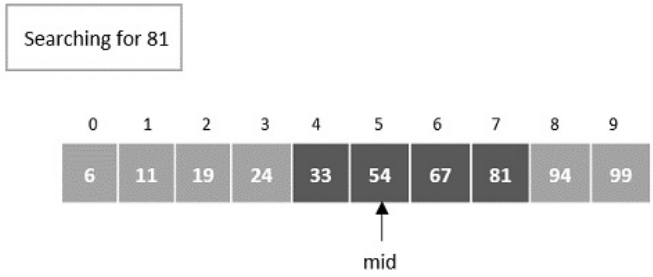



*Figure explaining step 5 of exponential search*

Sixth step, since the match cannot be made at the mid, it follows that the key element is larger than the mid. As a result, the search is conducted in the block's right half.

The mid is now the sixth element.

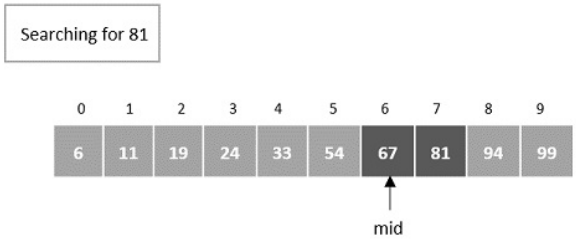



*Figure explaining step 6 of exponential search*

Final step, It now searches on the right half of the mid element because the element is still not discovered at the sixth element. The mid is then allocated at the seventh element. There, the element is located.

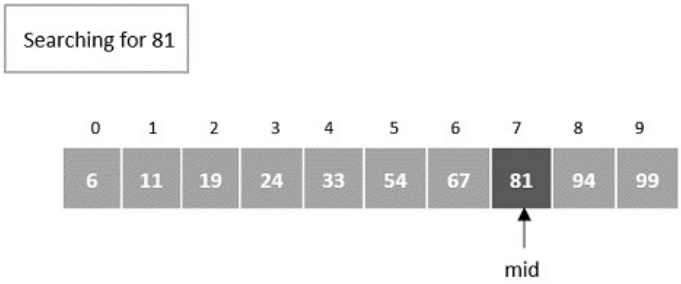


*Figure explaining final step of exponential search*

Since exponential search employs binary search, it accompanies the same drawbacks. Its power is restricted since a prerequisite for the binary search to function properly is that the array is sorted. Not only that, exponential search is much more complex than other searching algorithms since it requires handling the doubling of the index and performing a binary search within a specific range. This additional complexity can make the implementation more error-prone and harder to understand.

Table 1: Steps of Exponential Search

| **Step** | **Description** |
|---|---|
| Step 1 | Start with the first element of the array |
| Step 2 | Check if the target element is found at the current position. If yes, the search is successful |
| Step 3 | If the target element is greater, double the range size |
| Step 4 | Perform a binary search within the current range |
| Step 5 | If the target element is found, the search is successful. If not, repeat steps 3 to 5 until the range exceeds the array size |
| Step 6 | Perform a binary search within the final range to pinpoint the exact location of the target element |

*Table summarizing exponential search*

## Hash Tables:

Hash Table is a data structure which stores data in an associative manner. In a hash table, data is stored in an array format, where each data value has its own unique index value. Access of data becomes very fast if we know the index of the desired data. Thus, it becomes a data structure in which insertion and search operations are very fast irrespective of the size of the data. Hash Table uses an array as a storage medium and uses hash technique to generate an index where an element is to be inserted or is to be located from.

## Hashing Process:

Hashing is a technique to convert a range of key values into a range of indexes of an array. For example, if the array is of size N, the hashing function should return a number between 0 and N-1. A simple hashing function would return the Key modulus the size of the array.

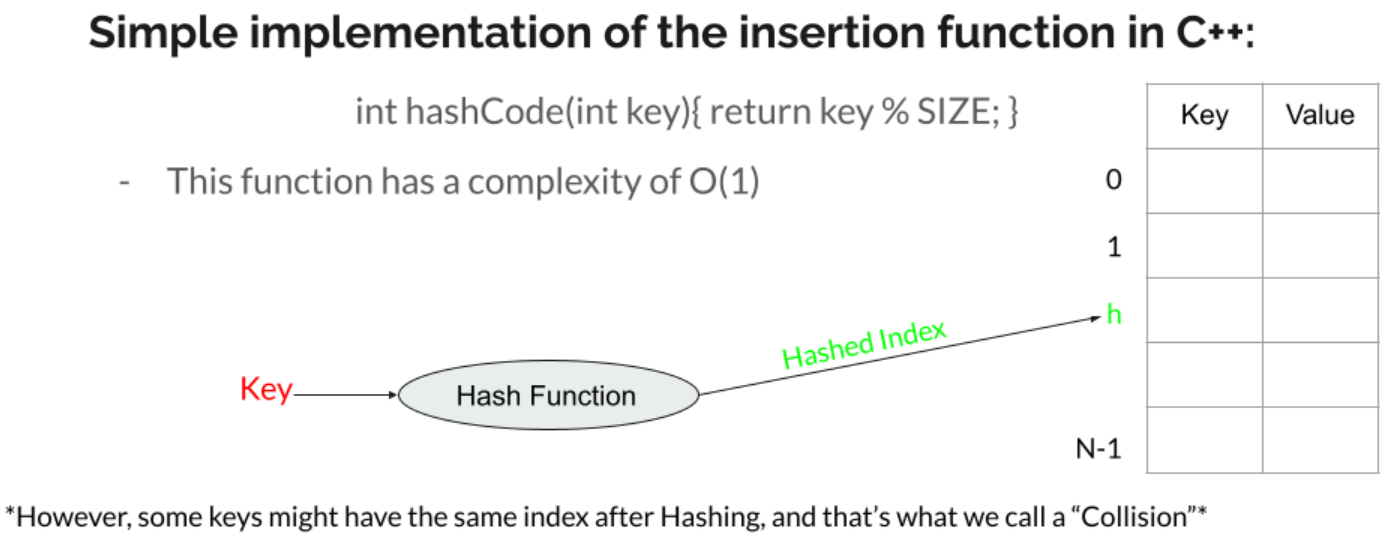



*Figure showing the Hashing process*

**Collisions:**

Collision happens when the hashing function returns the same value for different keys.

For Example :

Array Size : 8  1st Key : 8  2ndKey : 16

8%8=0  and  16%8=0

There are two types of collision handling : Open Addressing and Chaining. The simplest form of open addressing is linear probing.It always puts the item in the next unoccupied index of the array. If location **(h)** is occupied, the next index to discover is **(h=(h+1)%arrSize;).** In Chaining, each slot in the hash table contains a linked list of key-value pairs that hash to that index. When a collision occurs, the new key-value pair is added to the linked list at that index. Chaining is more memory efficient than Open Addressing.

**Searching Process:**

I. Open Addressing

- In this type of search, we'll assume that collisions are handled through Open Addressing.

II. Chaining

- In this type of search, we'll assume that collisions are handled through Chaining.

In the case of open addressing, the size of the array is set to 20, and the hashing algorithm used is (Key % Size). We employ linear probing to handle collisions, where the next available slot is found by incrementing the index

using the formula (h = (h+1)%ArraySize).

We will now perform a search for the following keys in the table:

- Key 44

During the search process, an empty location in the array is denoted by the value -1. By applying the hashing algorithm and the linear probing technique, we will determine the location of each key within the array.

| Key: | 80 | 21 | 20 | 18 | 24 | 5 | -1 | 47 | 87 | -1 | 50 | 71 | -1 | -1 | 34 | 14 | 76 | -1 | 98 | 59 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Index: | [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] | [9] | [10] | [11] | [12] | [13] | [14] | [15] | [16] | [17] | [18] | [19] |

*Figure showing the array to be implemented*

- Searching For 44 in the array :

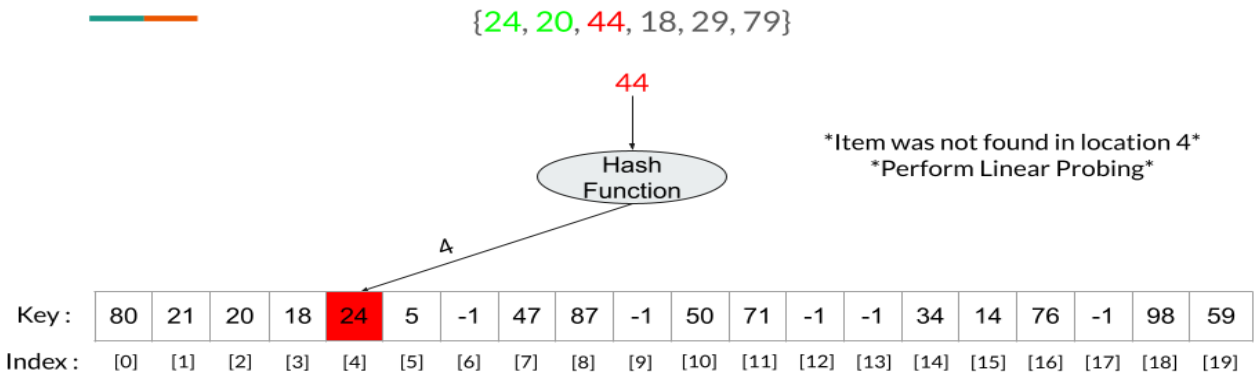



*Figure 1 showing step 1 of searching for 44*

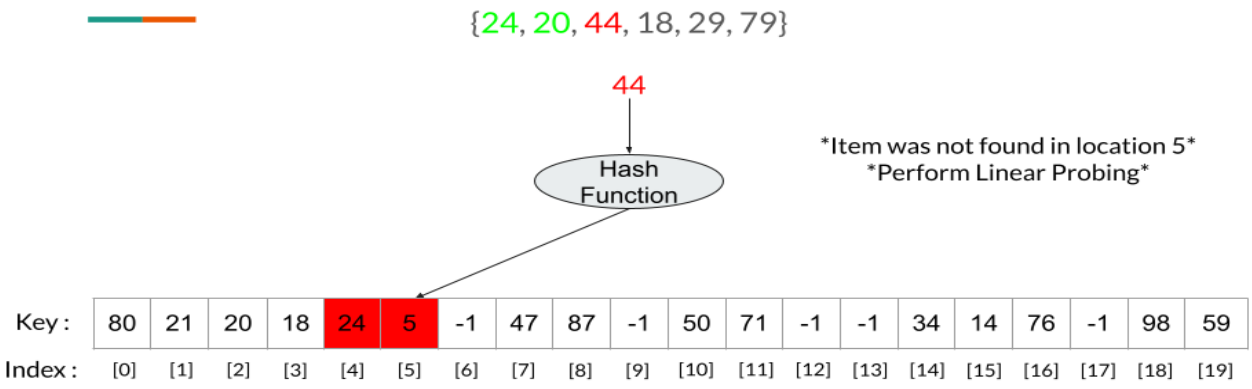



*Figure 1 showing step 2 of searching for 44*

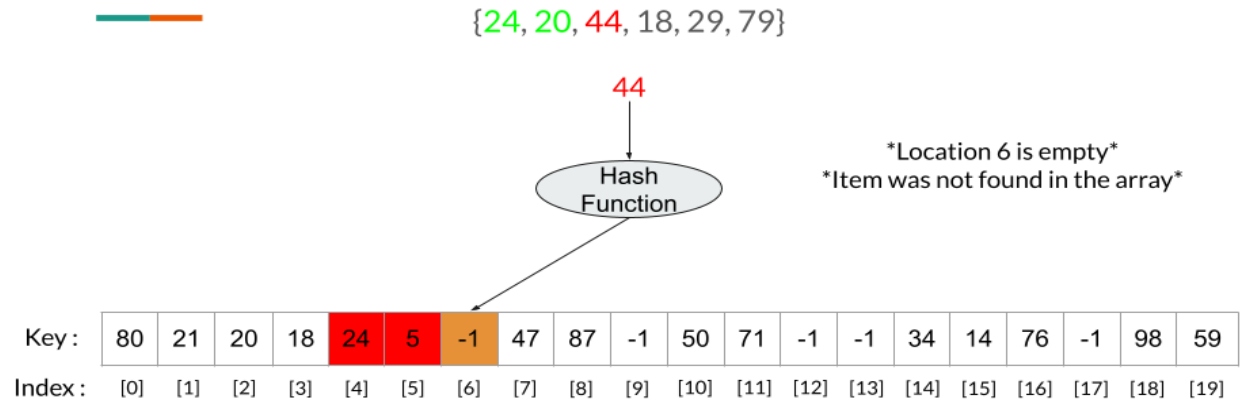



*Figure 1 showing step 3 of searching for 44*

Initially, the search examined location 4 in the hash table. However, the desired item was not found at this position.As a result, the algorithm performed linear probing and moved to the next position, which was location 5. Unfortunately, item 44 was still not found at this location.The algorithm continued with linear probing and moved to the subsequent position, which was location 6. At this point, it was observed that location 6 was empty, indicating that no item was present there.Since the item was not found in any of the examined locations, and the current probed location is empty, it can be concluded that item 44 is not present in the hash table because if it were to exist, it would have been in that empty location.

## **Searching in Chaining:**

We'll be searching through an array of linked lists. Each index of that array will host a linked list of keys that share the same hashed value. Once we get our hashed value, we go to that index in the array. We then perform a linear search on that specific linked list to find the key we're looking for.

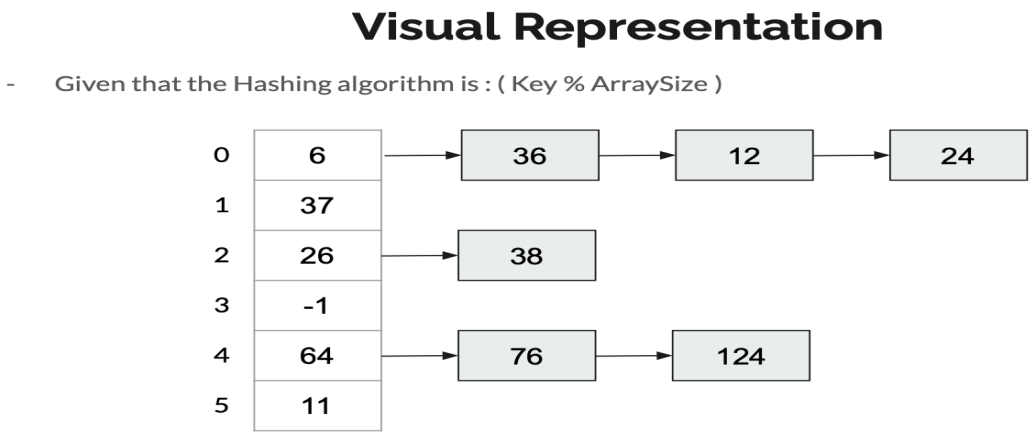
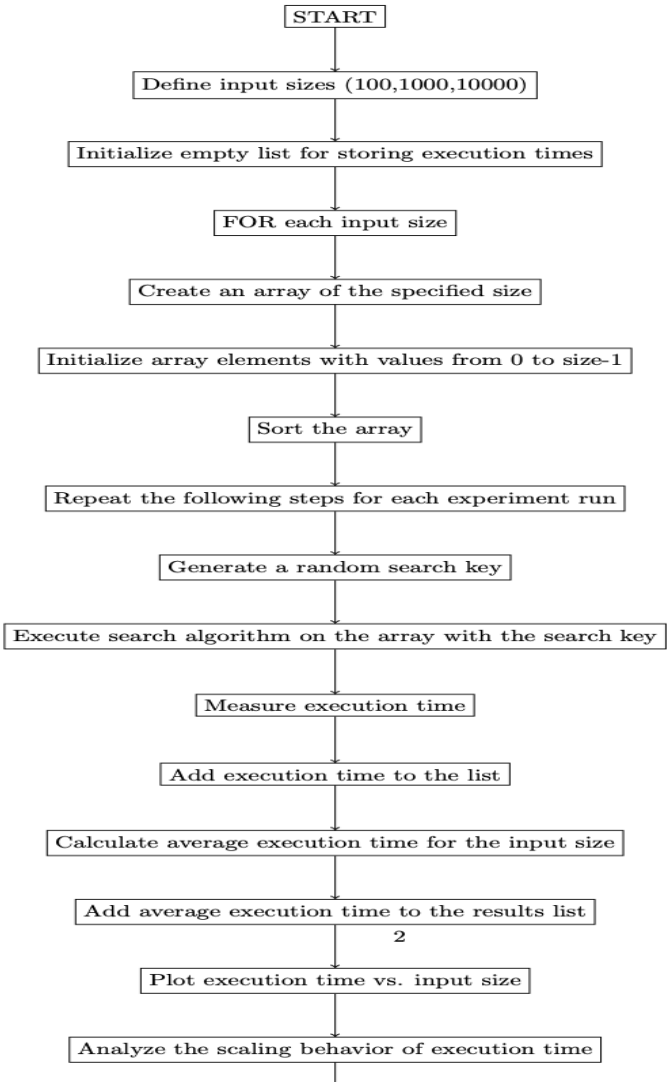



*Figure showing visual representation of chaining.*

## PROPOSED APPROACH

Our study aims to explore the time complexity of the following search algorithms empirically in order to find the efficiency of each searching algorithm. To reach this objective, we will conduct experiments to gauge the execution time of the search algorithm for various input sizes and analyze the results to identify the algorithm's time complexity. We have selected a range of input sizes- small arrays containing 100 elements and larger arrays with 10000 elements for this investigation. During this process, we aim to scrutinize the scaling behavior of the algorithmic execution time depending on the input size. Using a loop, we'll initialize each element of an array with values from 0 to size-1, ensuring it is sorted and easily searchable with each algorithm. We'll generate a random integer for the search key to test the algorithm, avoiding biased values. This method will ensure that the search key is not biased toward any particular value and that the algorithm is tested for various inputs. Finally, to ensure statistical significance and accuracy, we will repeat the experiment multiple

times for each input size and take the average execution time. We will repeat the experiment at least three times for each input size.




*Flowchart explaining the approach in the research*

## Results & Evaluation

## Binary Search:

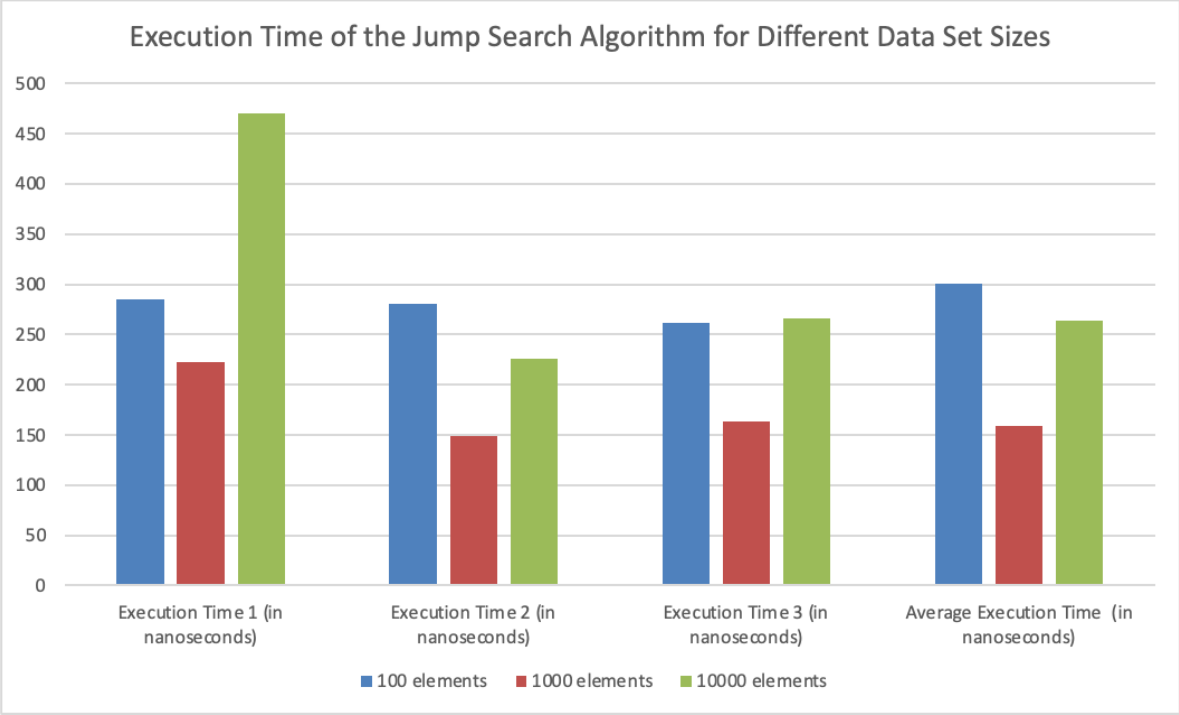
The results indicate that binary search is most effective for larger data sets, evidenced by the algorithm's logarithmic time complexity growth rate. Specifically, when searching for keys within arrays of size 100, 1000, and 10000, the binary search had an average execution time of (764, 97, 542) nanoseconds. Notably, despite its efficient performance in large inputs contexts, reflecting a strength in the algorithm, significant variation was observed in execution times which may result from technical fluctuations in the system. Overall, the results suggest that the binary search algorithm is suitable for searching through large sorted arrays with logarithmic time complexity.

| Number of Elements | Execution Time 1 (in nanoseconds) | Execution Time 2 (in nanoseconds) | Execution Time 3 (in nanoseconds) | Average Execution Time (in nanoseconds) |
|---|---|---|---|---|
| 100 | 750 | 667 | 875 | 764 |
| 1000 | 83 | 125 | 83 | 97 |
| 10000 | 417 | 750 | 458 | 542 |


## Jump Search:

The results obtained through running the experiment on C++ are quite unexpected. The algorithm performed best, consistently, for the array of size 1000, whereas it was slowest for an array of size 100, the smallest array in the experiment. This is unexpected as I would have expected the execution time to increase with the size of the array. However we can see that is not the case as the 10,000 element arrays were searched faster than the 100 element arrays. Jump search has a time complexity of O(n/m+m-1) which puts it as more efficient than linear search, however it is less efficient than binary search.

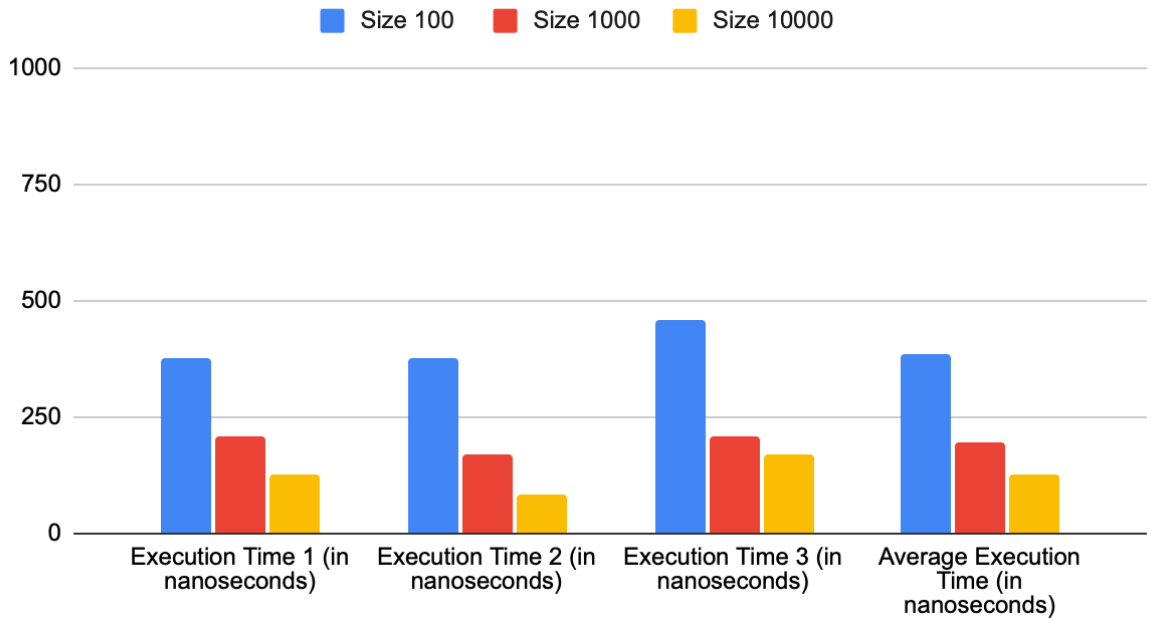
| Number of Elements | Execution Time 1 (in nanoseconds) | Execution Time 2 (in nanoseconds) | Execution Time 3 (in nanoseconds) | Average Execution Time (in nanoseconds) |
|---|---|---|---|---|
| 100 | 304 | 280 | 287 | 301 |
| 1000 | 153 | 149 | 159 | 159 |
| 10000 | 232 | 226 | 266 | 264 |




## Exponential Search:

Through the given results of running the exponential search in c++, it is seen that as the number of elements increases, the execution time generally decreases. This is expected as exponential search has a time complexity of O(log N). As the array size grows, the time taken for the search operation logarithmically increases at a slower rate. Additionally, the execution times are relatively low for all the tested cases of sizes 100,1000,10000 which is seen through the average execution times, 386,194, and 125 nanoseconds, respectively for each of the array sizes. This indicates that exponential search is efficient for searching in sorted arrays, even with a larger number of elements. The average execution times are within a reasonable range, suggesting that exponential search performs well.

| Number of Elements | Execution Time 1 (in nanoseconds) | Execution Time 2 (in nanoseconds) | Execution Time 3 (in nanoseconds) | Average Execution Time (in nanoseconds) |
|---|---|---|---|---|
| 100 | 375 | 375 | 459 | 386 |
| 1000 | 208 | 167 | 208 | 194 |
| 10000 | 125 | 83 | 167 | 125 |




## Hashing Search:

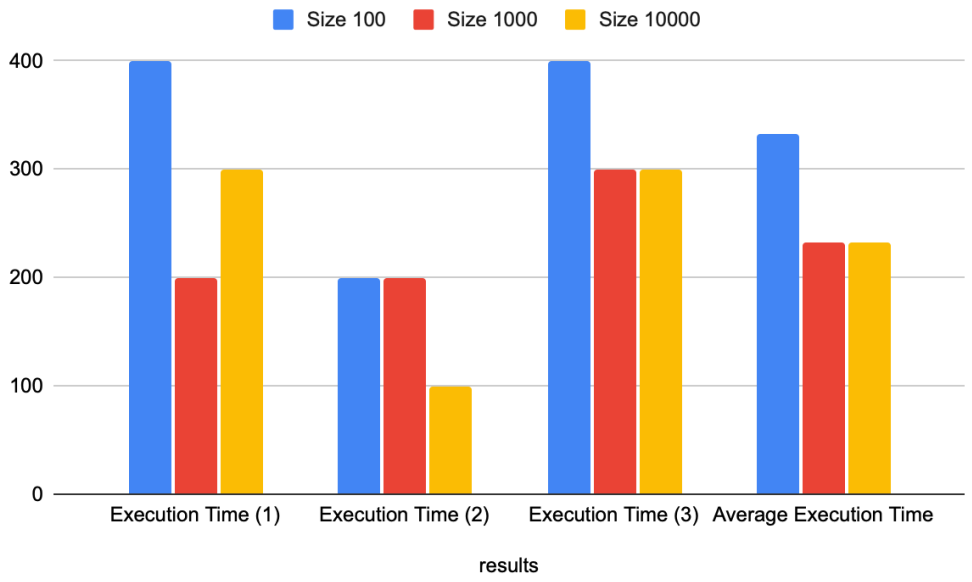
The results for hashing search are made up of 3 different cases:

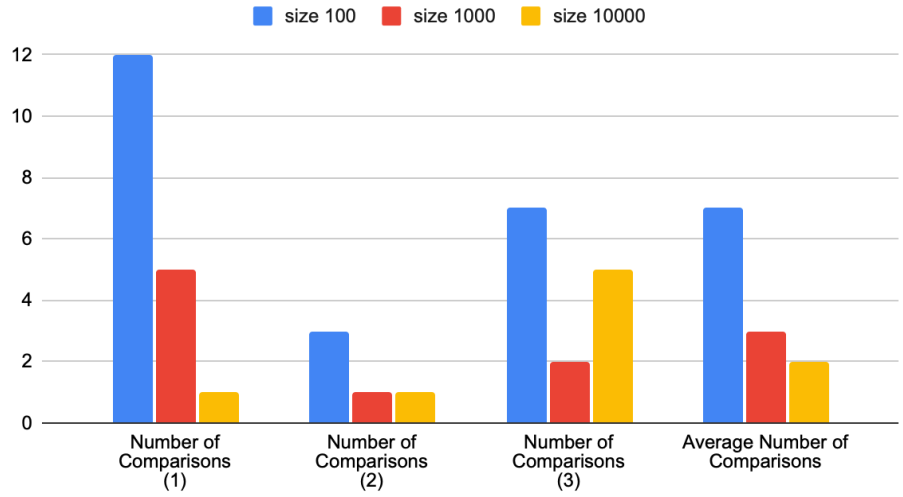
- **Case 1 : Array is Full : (Worst Case)**
  - **The Hash Table is using Linear Probing to handle collisions.**
  - **The execution time is in nanoseconds.**

*COMPLEXITY*

| Array Size | Execution Time (1) | Execution Time (2) | Execution Time (3) | Average Execution Time | Number of Comparisons (1) | Number of Comparisons (2) | Number of Comparisons (3) | Average Number of Comparisons |
|---|---|---|---|---|---|---|---|---|
| 100 | 400 | 200 | 400 | 333 | 12 | 3 | 7 | 7 |
| 1000 | 200 | 200 | 300 | 233 | 5 | 1 | 2 | 3 |
| 10000 | 300 | 100 | 300 | 233 | 1 | 1 | 5 | 2 |

Average Time : 266 nanoseconds
Average Number of Comparisons : 4





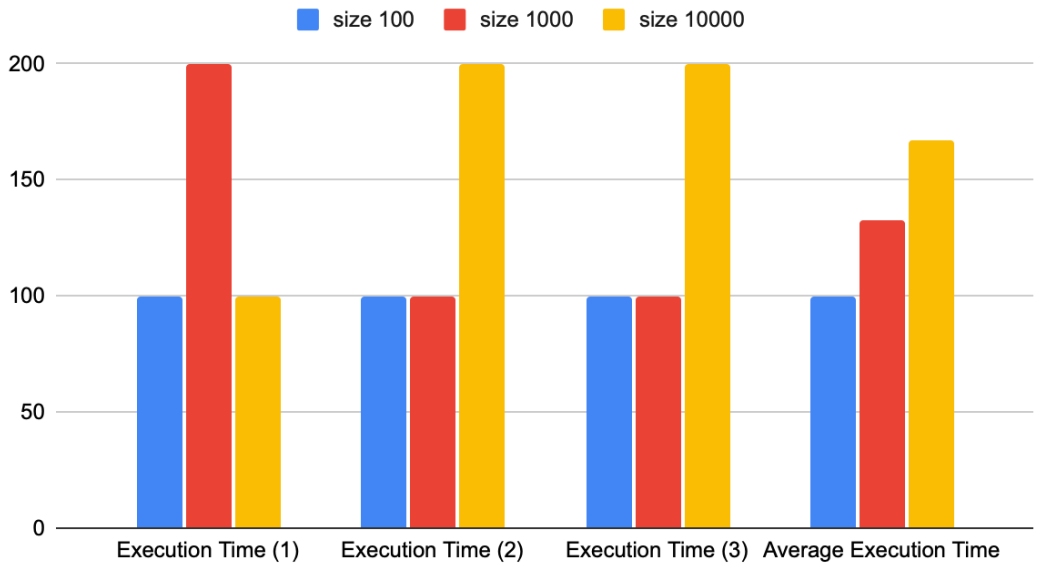

results case 1

- **Case 2 : Array is approximately 50% Full : (Average Case)**
  - **The Hash Table is using Linear Probing to handle collisions.**
  - **The execution time is in nanoseconds.**

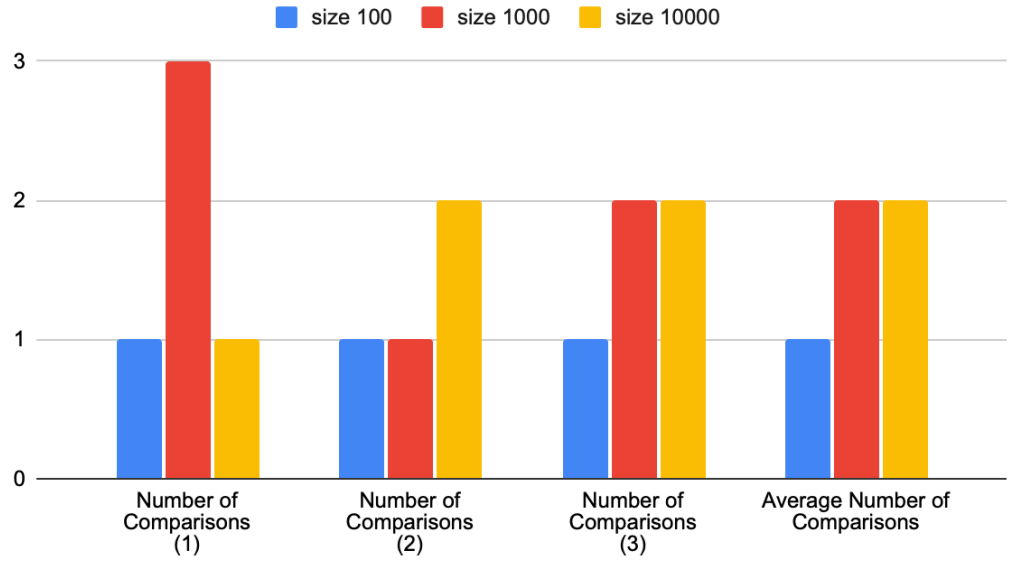
*COMPLEXITY*

| Array Size | Execution Time (1) | Execution Time (2) | Execution Time (3) | Average Execution Time | Number of Comparisons (1) | Number of Comparisons (2) | Number of Comparisons (3) | Average Number of Comparisons |
|---|---|---|---|---|---|---|---|---|
| 100 | 100 | 100 | 100 | 100 | 1 | 1 | 1 | 1 |
| 1000 | 200 | 100 | 100 | 133 | 3 | 1 | 2 | 2 |
| 10000 | 100 | 200 | 200 | 167 | 1 | 2 | 2 | 2 |

Average Time : 133 nanoseconds
Average Number of Comparisons : 2




results case 2



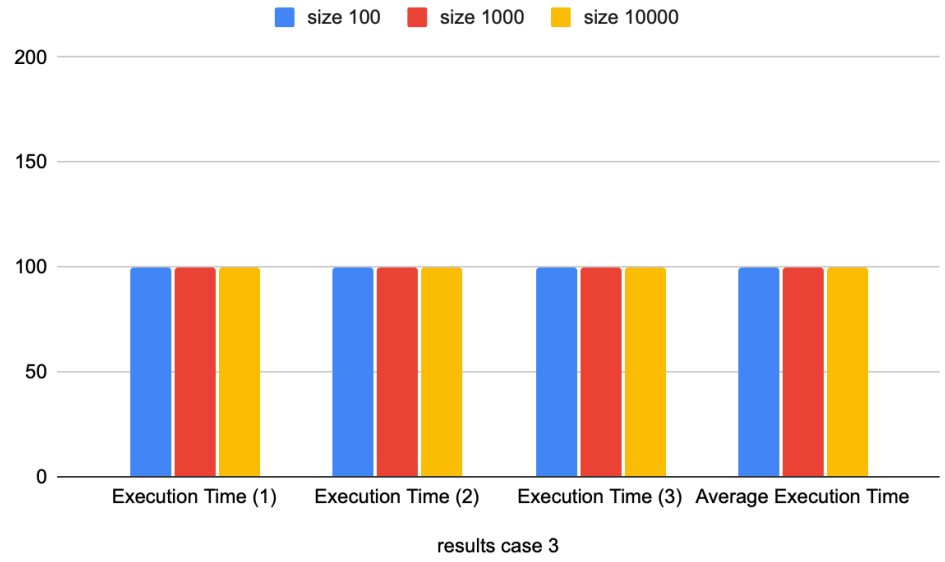

results case 2

- **Case 3 : Array is approximately 25% Full : (Best Case)**
  - **The Hash Table is using Linear Probing to handle collisions.**
  - **The execution time is in nanoseconds.**

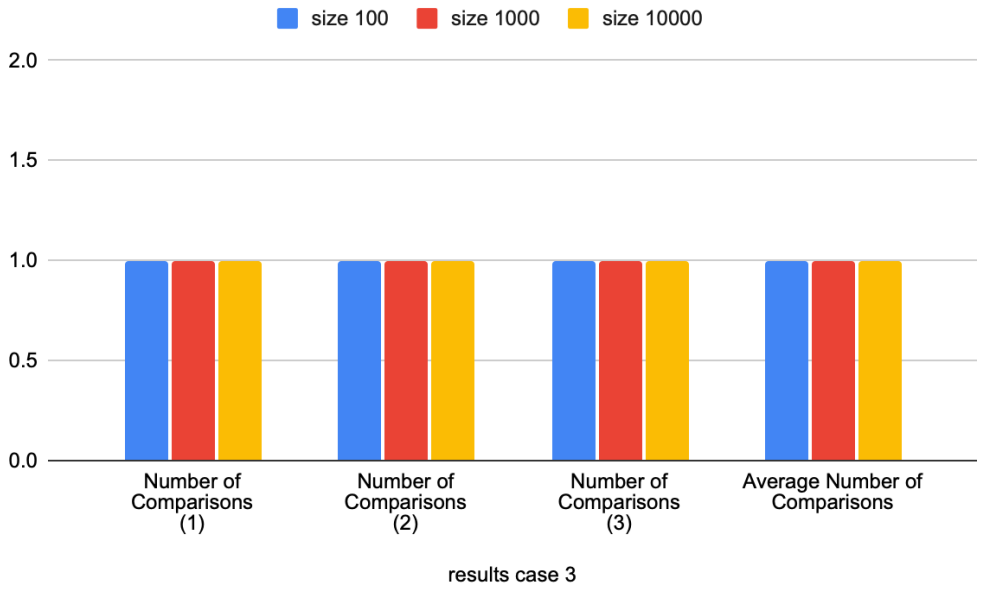
*COMPLEXITY*

| Array Size | Execution Time (1) | Execution Time (2) | Execution Time (3) | Average Execution Time | Number of Comparisons (1) | Number of Comparisons (2) | Number of Comparisons (3) | Average Number of Comparisons |
|---|---|---|---|---|---|---|---|---|
| 100 | 100 | 100 | 100 | 100 | 1 | 1 | 1 | 1 |
| 1000 | 100 | 100 | 100 | 100 | 1 | 1 | 1 | 1 |
| 10000 | 100 | 100 | 100 | 100 | 1 | 1 | 1 | 1 |

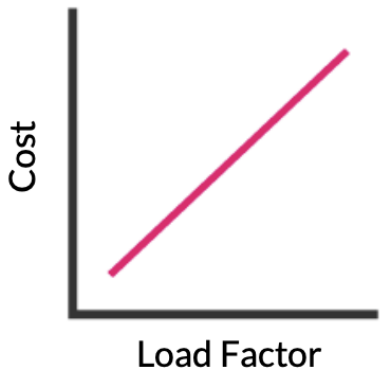
Average Time : 100 nanoseconds
Average Number of Comparisons : 1




results case 3




Thus, from the results above we can conclude that the cost of searching in a Hash Table is independent of the array size. Additionally, the Load Factor is what affects the searching cost, as the Load Factor increases, the number of collisions increases. As the number of collisions increases, the searching cost increases. So, the Load Factor is directly proportional to the Searching cost.



## Discussion

The presented results demonstrate the effectiveness of binary search in searching through large sorted arrays with logarithmic time complexity growth rate (O log N). This complexity is evidenced by the algorithm's efficient performance when searching for keys within arrays of size 100, 1000, and 10000, with an average execution time of (764, 97, 542) nanoseconds, respectively. However, despite its efficient performance, there was significant variation in execution times that may result from technical fluctuations in the system. Overall, the results suggest that binary search is most effective for larger data sets, making it suitable for practical applications in contexts where large sorted arrays are standard.

In contrast, the results obtained by running the experiment on C++ indicate that the jump search algorithm performed best for the size 1000, whereas it was slowest for the smallest array size of 100. This result is unexpected, as one would expect execution time to increase with the size of the array due to the time complexity of O(N/M + (M-1)). Nevertheless, the execution times for 10,000 element arrays were faster than those for 100 element arrays, indicating that the algorithm's performance may vary depending on the size of the data set. It would be needed to be tested with a more extensive data set repetitively to make sure of these findings as they were unpredicted.

Regarding exponential search, the results show that it is efficient for searching in sorted arrays, even with a more significant number of elements. As the number of components increases, the execution time generally decreases due to the algorithm's logarithmic time complexity of O(log N). The execution times are relatively low for all the tested cases of sizes 100, 1000, and 10000, with average execution times of 386, 194,

and 125 nanoseconds, respectively. Indicates that exponential search is an effective alternative to binary search for searching in sorted arrays.

Hash tables have a complexity of O(1) in their best case, which may vary depending on the load factor. The load factor represents the number of occupied spots over the array size. The worst-case scenario for searching hash tables is looking for an element that does not exist in a load factor 1 array, which results in an O(N) complexity. As a result, the efficiency of hash tables in actual applications is determined by their load factor and data set size. The findings indicate that hash tables can be a more efficient option to binary and exponential search in some situations, mainly when dealing with big data sets, even more significant than 10000 - outside the paper's experiment scope.

## Conclusion

In conclusion, it is evident from the results that each method differs in its strengths and limitations according to the size of the array being searched. The hash table search performed best for the smallest array size of 100, whereas the binary search excelled for the largest array of 1000. The exponential search proved the most efficient for the largest array size of 10000. Despite these variations, it is evident that binary or exponential search algorithms are the optimal choices for searching through sorted arrays with logarithmic time complexity growth rates; the exponential search algorithm showed a consistent decrease in execution time as the number of elements in the array increased. Hash tables were also quite effective in their best cases, with a complexity of O(1); however, their performance depends on the load factor. To get the fastest execution time, consider the array's size when choosing the best searching algorithm.

# References

Design and analysis - exponential search. Tutorials Point. (n.d.). https://www.tutorialspoint.com/design_and_analysis_of_algorithms/design_and_analysis_of_algorithms_exponential_search.htm

GeeksforGeeks. (2023, April 17). *Binary search - data structure and algorithm tutorials*. GeeksforGeeks. https://www.geeksforgeeks.org/binary-search/

GeeksforGeeks. (2023b, May 3). *Measure execution time of a function in C++*. GeeksforGeeks. https://www.geeksforgeeks.org/measure-execution-time-function-cpp/

GeeksforGeeks. (2023, March 28). *Exponential search*. GeeksforGeeks. https://www.geeksforgeeks.org/exponential-search/

GeeksforGeeks. (2023, April 25). *Jump search*. GeeksforGeeks. https://www.geeksforgeeks.org/jump-search/

Fulber-Garcia, V. (2023, March 11). *Understanding hash tables*. Baeldung on Computer Science. https://www.baeldung.com/cs/hash-tables

# Appendix

Binary search

Code used for the experiment.

For measuring execution time, help was used from: GeeksforGeeks. (2023b, May 3). *Measure execution time of a function in C++*. GeeksforGeeks. https://www.geeksforgeeks.org/measure-execution-time-function-cpp/

```
#include <chrono>

#include <iostream>

#include <cstdlib>

#include <ctime>

using namespace std::chrono;

using namespace std;

const int MAX_SIZE = 10000;

int arr[MAX_SIZE];

void generate_array(int size) {

  srand(time(NULL));

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

```
    arr[i] = rand() % 10001;
  }
  sort(arr, arr + size);
}


int binary_search(int arr[], int size, int target) {
  int left = 0;
  int right = size - 1;
  while (left <= right) {
    int mid = (left + right) / 2;
    if (arr[mid] == target) {
      return mid;
    } else if (arr[mid] < target) {
      left = mid + 1;
    } else {
      right = mid - 1;
    }
  }
  return -1;
```

```
}

long long run_experiment(int size) {

  // Create a sorted array of size elements

  int* arr = new int[size];

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

      arr[i] = i;

  }



  // Generate a random key to search for

  int key = rand() % size;



  // Measure the execution time of binary search

  auto start = std::chrono::high_resolution_clock::now();

  int index = binary_search(arr, size, key);

  auto end = std::chrono::high_resolution_clock::now();

  auto duration = std::chrono::duration_cast<std::chrono::nanoseconds>(end - start).count();



  // Clean up the array and return the execution time

  delete[] arr;
```

```
  return duration;

}



int main()

{

  long long time1 = run_experiment(100);

  long long time2 = run_experiment(1000);

  long long time3 = run_experiment(10000);



  cout << "Execution time for 100 elements: " << time1 << " nanoseconds" << endl;

  cout << "Execution time for 1000 elements: " << time2 << " nanoseconds" << endl;

  cout << "Execution time for 10000 elements: " << time3 << " nanoseconds" << endl;



  return 0;

}
```

Exponential Search:

Code used for experiment:

For measuring execution time, help was used from: GeeksforGeeks. (2023b, May 3). *Measure execution time of a function in C++*. GeeksforGeeks. https://www.geeksforgeeks.org/measure-execution-time-function-cpp/

```
#include <iostream>

#include <chrono>

using namespace std;

int exponentialSearch(int arr[], int size, int target) {

if (arr[0] == target) {

    return 0;

  }



  int i = 1;

  while (i < size && arr[i] <= target) {

    i *= 2;

  }
```

```
  int left = i / 2;
  int right = std::min(i, size - 1);


  // Perform exponential search within the identified range
  while (left <= right) {
    int mid = left + (right - left) / 2;
    if (arr[mid] == target) {
      return mid;  // Target found at index mid
    } else if (arr[mid] < target) {
      left = mid + 1;  // Search in the right half
    } else {
      right = mid - 1;  // Search in the left half
    }
  }


  return -1;  // Target not found
}


long long run_experiment(int size) {
```

```
    // Create a sorted array of size elements
    int* arr = new int[size];
    for (int i = 0; i < size; i++) {
        arr[i] = i;
    }

    // Generate a random key to search for
    int key = rand() % size;

    auto start = std::chrono::high_resolution_clock::now();
    int index = exponentialSearch(arr, size, key);
    auto end = std::chrono::high_resolution_clock::now();
    auto duration = std::chrono::duration_cast<std::chrono::nanoseconds>(end - start).count();

    // Clean up the array and return the execution time
    delete[] arr;
    return duration;
}
```

```
int main()

{

  long long time1 = run_experiment(100);

  long long time2 = run_experiment(1000);

  long long time3 = run_experiment(10000);



  cout << "Execution time for 100 elements: " << time1 << " nanoseconds" << endl;

  cout << "Execution time for 1000 elements: " << time2 << " nanoseconds" << endl;

  cout << "Execution time for 10000 elements: " << time3 << " nanoseconds" << endl;



  return 0;

}
```

Jump Search:

Code used for experiment:

For measuring execution time, help was used from: GeeksforGeeks. (2023b, May 3). *Measure execution time of a function in C++*. GeeksforGeeks. https://www.geeksforgeeks.org/measure-execution-time-function-cpp/

```
#include <chrono>

#include <iostream>

#include <cstdlib>

#include <ctime>

#include <cmath>



using namespace std::chrono;

using namespace std;



const int MAX_SIZE = 10000;



int arr[MAX_SIZE];
```

```
void generate_array(int size) {
  srand(time(NULL));
  for (int i = 0; i < size; i++) {
     arr[i] = rand() % 10001;
  }
  sort(arr, arr + size);
}

int jumpsearch(int arr[], int size, int target)
{
  int m = sqrt(size);
  int low = 0;
  int high = 0;

  for(int i=0;i<size/m;i++)
  {
     if(arr[high] == target)
        return high;
```

```
    else if(arr[high] > target)
    {
        for(int j=low;j<high;j++)
        {
            if(arr[j]==target)
                return j;
        }
    }
    else
    {
        low = high;
        high += m;
    }
  }
  return -1;
}
long long run_experiment(int size) {
  // Create a sorted array of size elements
  int* arr = new int[size];
```

```
  for (int i = 0; i < size; i++) {
      arr[i] = i;
  }

  // Generate a random key to search for
  int key = rand() % size;

  // Measure the execution time of binary search
  auto start = std::chrono::high_resolution_clock::now();
  int index = jumpsearch(arr, size, key);
  auto end = std::chrono::high_resolution_clock::now();
  auto duration = std::chrono::duration_cast<std::chrono::nanoseconds>(end - start).count();

  // Clean up the array and return the execution time
  delete[] arr;
  return duration;
}

int main()
```

```
{

  long long time1 = run_experiment(100);

  long long time2 = run_experiment(1000);

  long long time3 = run_experiment(10000);



  cout << "Execution time for 100 elements: " << time1 << " nanoseconds" << endl;

  cout << "Execution time for 1000 elements: " << time2 << " nanoseconds" << endl;

  cout << "Execution time for 10000 elements: " << time3 << " nanoseconds" << endl;



  return 0;

}
```

Hash Tables:

1) Header File:

```
// File: hashTable.h

// Definition of Hash Table Template Class

#ifndef HASH_TABLE_H

#define HASH_TABLE_H

// Specification of the class

template <class keyType, class dataType>



class hashTable

{

 public:



   // Member Functions

  // Constructor with two arguments,

   // Table size is nelements, k is "empty" value of keyType
```

```
hashTable(int nelements, const keyType &k);

~hashTable();                // Destructor



// Functions Prototype Definitions



bool tableIsEmpty() const;    // return True if table is empty

bool tableIsFull() const;     // return True if table is full

int  occupancy() const;          // return no. of occupied slots

// insert key and data at a hashed slot, return true if successful

bool insert(const keyType &, const dataType & );



// Search the table for the slot that matches key.

 // If found, return True, set current position to slot

bool search(const keyType & ,int &Counter);       // Search given key



 void updateData(const dataType & );        // Update data part of current slot

void retrieveData(dataType &) const;   // Retrieve data part of current slot

void traverse() const;               // Traverse whole table
```

```
private:



 // Slot Class

    class slot

    {

    public:

       keyType key;      // key

       dataType data;    // Data

    }; // end of class slot declaration



 slot *T;              // Pointer to Storage Array

 int h;               // Index to a slot

 int MaxSize, csize;         // Maximum and Current Sizes

 keyType Empty;         // Key value to consider as "Empty"



 // Private Member function

 int hash(const keyType & ) const;     // Hashing function

}; // End of class hashTable definition

#endif // HASH_TABLE_H
```

```
#include "hashTable.cpp"
```

2) Implementation File:

```
// File:hashTable.cpp

// hashTable Class implementation file



#include <iostream>

using namespace std;



// Constructor with two arguments,

// Table size is nelements, k is "empty" value of keyType

template <class keyType, class dataType>

hashTable<keyType, dataType>::hashTable(int nelements, const keyType &k)

{  MaxSize = nelements; T = new slot[MaxSize];

  Empty = k;

  for(int i = 0; i < MaxSize; i++) T[i].key = Empty;

  h = -1;  csize = 0;

}
```

```
// Destructor

template <class keyType, class dataType>

hashTable<keyType, dataType>::~hashTable()

{ delete [] T;}



// return True if table is empty

template <class keyType, class dataType>

bool hashTable<keyType, dataType>::tableIsEmpty() const

{

  return (csize == 0);

}



// return True if table is full

template <class keyType, class dataType>

bool hashTable<keyType, dataType>::tableIsFull() const

{

  return (csize == MaxSize);

}
```

```
// return no. of occupied slots

template <class keyType, class dataType>

int hashTable<keyType, dataType>::occupancy() const

{

  return csize;

}




// insert key and data at a hashed slot, return true if successful

template <class keyType, class dataType>

bool hashTable<keyType, dataType>::insert(const keyType &k, const dataType &d)

{

  if (!tableIsFull())

  {

     h = hash(k);

     while(T[h].key != Empty)

        h = (h+1) % MaxSize;

     T[h].key = k;  T[h].data = d; csize++;
```

```
    return true;
  }
  else return false;
}


// Search the table for the slot that matches key.
// If found, return True, set current position to slot
template <class keyType, class dataType>
bool hashTable<keyType, dataType>::search(const keyType &k , int &Counter)
{
  Counter=0;
  if(!tableIsEmpty())
  {
    h = hash(k); int start = h;
    for ( ; ; )
    {
      if (T[h].key == Empty) return false;
      if (k == T[h].key)
        {
```

```
            Counter++;

            return true;

          }

          else

          {

            Counter++;

          }

        h = (h+1) % MaxSize;

        if (h == start) return false;

     }

  }

  else return false;

}




// Update the data part of the current slot

template <class keyType, class dataType>

void hashTable<keyType, dataType>::updateData(const dataType &d )

{
```

```
  if ((h >=0)&&(h < MaxSize)) T[h].data = d;

}



// Retrieve the data part of the current slot

template <class keyType, class dataType>

void hashTable<keyType, dataType>::retrieveData(dataType &d) const

{

  if ((h >= 0)&&(h < MaxSize)) d = T[h].data;

     else d = T[0].data;

}



// Traverse whole table

template <class keyType, class dataType>

void hashTable<keyType, dataType>::traverse() const

{

  for(int i = 0; i < MaxSize; i++)

     cout << T[i].key << endl;

}
```

```
// Private Hashing Function

template <class keyType, class dataType>

int hashTable<keyType, dataType>::hash(const keyType &k ) const

{

  return (k % MaxSize);

}
```

3) Main Program:

```
#include <iostream>

#include <chrono>

#include <stdlib.h>

#include <ctime>

#include "hashTable.h"

using namespace std;

int main()

{

   int N=10000,Counter=0;        // Change N to change the array size

   hashTable<int,int> H(N,-1);
```

```
  srand(time(0));

  for(int i=0;i<N;i+=4)      // To have a full array use (i++)

//To have a 50% full array use (i+=2)

//To have a 25% full array use (i+=4)

  {

     H.insert(rand()%(N/4),i);

  }

  auto start_time = std::chrono::high_resolution_clock::now(); // get start time

  H.search(rand()%(N/4),Counter);

  auto end_time = std::chrono::high_resolution_clock::now(); // get end time

  auto duration_ns = std::chrono::duration_cast<std::chrono::nanoseconds>(end_time - start_time); // calculate time duration in
nanoseconds

  cout << "Time taken: " << duration_ns.count() << " ns" << std::endl;

  cout<<"Number  of Comparisons is : "<<Counter<<endl;

}
```