# SEARCHING

ALGORITHMS IN JAVA

Sercan Külcü | Algorithms In Java | 10.05.2023

# Contents

# Introduction

Searching algorithms are used to find the presence or location of a specific element within a collection of data. Here are some commonly used searching algorithms:

- Linear Search: This is the simplest searching algorithm that sequentially checks each element of the collection until a match is found or the end of the collection is reached. It has a time complexity of O(n), where n is the number of elements in the collection. Linear search is applicable to both sorted and unsorted data.
- Binary Search: Binary search is an efficient algorithm for finding an element in a sorted collection. It works by repeatedly dividing the search interval in half until the element is found or determined to be absent. It has a time complexity of O(log n) since it eliminates half of the remaining elements at each step. Binary search can only be used on sorted data.
- Hashing: Hashing is a technique that uses a hash function to map keys to array indices, allowing for constant-time average case search complexity. It is commonly used in data structures like hash tables. However, in the worst case, hashing can have a time complexity of O(n) due to collisions.
- Jump Search: Jump search is an algorithm that works on sorted data. It jumps ahead by a fixed number of steps and then performs a linear search to find the desired element. It is generally faster than linear search but slower than binary search. Its time complexity is O(sqrt(n)).
- Interpolation Search: Interpolation search is an improved version of binary search that works well on uniformly distributed sorted data. Instead of always dividing the search interval in half, it estimates the position of the desired element based on the values of the endpoints. It has an average time complexity of O(log log

n) for uniformly distributed data, but it can degrade to O(n) in certain cases.

- Exponential Search: Exponential search is a combination of binary search and linear search. It starts with a small range and doubles it until the desired element is potentially within that range. It then performs a binary search within that range. It has a time complexity of O(log n) in the average case.

These are some commonly used searching algorithms, each with its own advantages and applicable scenarios. The choice of the searching algorithm depends on factors such as the nature of the data, whether it is sorted or unsorted, and the desired time complexity.

# Linear search

Linear search is a simple algorithm for finding an element in a list. It works by sequentially checking each element in the list until the target element is found. If the target element is not found, the algorithm returns -1.

Linear search is an inefficient algorithm, with a time complexity of O(n), where n is the number of elements in the list. However, it is a simple algorithm to implement, and it can be used to find an element in a list even if the list is not sorted.

The linear search algorithm works as follows:

- Start at the beginning of the list.
- Compare the current element to the target element.
- If the current element is equal to the target element, return the index of the current element.
- Otherwise, move to the next element in the list.
- Repeat steps 2-4 until the end of the list is reached.
- If the target element is not found, return -1.

The linear search algorithm can be implemented in Java as follows:

```
public class LinearSearch {


 public static int linearSearch(int[] list, int target) {

  int index = -1;


  for (int i = 0; i < list.length; i++) {

   if (list[i] == target) {

    index = i;
```

```
        break;
      }
    }

    return index;
  }

  public static void main(String[] args) {
    // Create a list of integers
    int[] list = {5, 3, 1, 2, 4};

    // Search for the element 3 in the list
    int index = linearSearch(list, 3);

    // Print the index of the element 3
    System.out.println(index);
  }
}
```

This code will print the following output:

```
2
```

Advantages and Disadvantages

The linear search algorithm has the following advantages:

- It is simple to implement.
- It can be used to find an element in a list even if the list is not sorted.

The linear search algorithm has the following disadvantages:

- It is inefficient, with a time complexity of O(n).
- It can be slow for large lists.

Linear search is a simple algorithm for finding an element in a list. It is inefficient, but it can be used to find an element in a list even if the list is not sorted.

# Binary search

Binary search is a search algorithm that finds the position of a target value within a sorted array. The algorithm works by repeatedly dividing the array in half and searching the half that contains the target value. The search continues until the target value is found or until the entire array has been searched.

Binary search is an efficient algorithm, with a time complexity of O(log n), where n is the number of elements in the array. This means that the time it takes to search the array grows logarithmically with the number of elements.

The binary search algorithm works as follows:

- Start with the middle element of the array.
- Compare the target value to the middle element.
- If the target value is equal to the middle element, return the index of the middle element.
- If the target value is less than the middle element, search the left half of the array.
- If the target value is greater than the middle element, search the right half of the array.
- Repeat steps 2-5 until the target value is found or until the entire array has been searched.
- If the target value is not found, return -1.

The binary search algorithm can be implemented in Java as follows:

```
public class BinarySearch {


  public static int binarySearch(int[] list, int target) {

    int low = 0;
```

```
  int high = list.length - 1;

  while (low <= high) {
    int mid = (low + high) / 2;

    if (list[mid] == target) {
      return mid;
    } else if (list[mid] < target) {
      low = mid + 1;
    } else {
      high = mid - 1;
    }
  }

  return -1;
}

public static void main(String[] args) {
  // Create a sorted array of integers
  int[] list = {1, 2, 3, 4, 5};

  // Search for the element 3 in the list
```

```
    int index = binarySearch(list, 3);


    // Print the index of the element 3
    System.out.println(index);
  }
}
```

This code will print the following output:

```
2
```

Advantages and Disadvantages

The binary search algorithm has the following advantages:

- It is efficient, with a time complexity of O(log n).
- It can be used to find an element in a list even if the list is not sorted.

The binary search algorithm has the following disadvantages:

- It can only be used to find elements in sorted arrays.
- It is not as simple to implement as some other search algorithms.

Binary search is an efficient algorithm for finding an element in a sorted array. It is not as simple to implement as some other search algorithms, but it is a powerful tool for searching sorted data.

# Hashing

Hashing is a technique for storing and retrieving data based on a key. A hash function takes an input value, called the key, and produces an output value, called the hash code. The hash code is then used to store the data in a hash table.

Hash tables are a data structure that stores data in a way that allows for efficient lookups. When data is stored in a hash table, it is hashed using a hash function. The hash code is then used to find the location of the data in the hash table.

Hashing is a powerful technique for storing and retrieving data. It is used in a variety of applications, including:

- Data structures: Hashing is used to implement data structures such as hash tables, hash sets, and hash maps.
- Algorithms: Hashing is used in a variety of algorithms, such as the quicksort algorithm and the merge sort algorithm.
- Cryptography: Hashing is used in cryptography to create secure hash functions, such as MD5 and SHA-1.

How Hashing Works

Hashing works by using a hash function to convert a key into a hash code. The hash code is then used to find the location of the data in the hash table.

The hash function is a mathematical function that takes an input value and produces an output value. The hash function is designed to be fast and efficient. It should also be evenly distributed, so that all possible hash codes are equally likely.

The hash table is a data structure that stores data in a way that allows for efficient lookups. The hash table is divided into a number of buckets. The hash code is used to determine which bucket the data should be stored in.

When data is stored in a hash table, it is first hashed using a hash function. The hash code is then used to find the bucket where the data should be stored. The data is then stored in the bucket.

When data is retrieved from a hash table, it is first hashed using a hash function. The hash code is then used to find the bucket where the data should be stored. The data is then retrieved from the bucket.

Advantages of Hashing

Hashing has a number of advantages, including:

- Efficiency: Hashing is a very efficient technique for storing and retrieving data.
- Space efficiency: Hashing can be used to store data in a compact space.
- Flexibility: Hashing can be used to store a variety of data types.

Disadvantages of Hashing

Hashing also has a number of disadvantages, including:

- Collisions: Hash collisions can occur when two different keys produce the same hash code.
- Space complexity: Hash tables can be inefficient for storing large amounts of data.
- Security: Hashing can be used to create secure hash functions, but it is not a foolproof security technique.

Hashing is a powerful technique for storing and retrieving data. It is used in a variety of applications, including data structures, algorithms, and cryptography. Hashing has a number of advantages, including efficiency, space efficiency, and flexibility. However, it also has a number of disadvantages, including collisions, space complexity, and security.

Here is an implementation of a hash map in Java:

public class HashMap<K, V> {

```
private static final int DEFAULT_CAPACITY = 16;
private static final float DEFAULT_LOAD_FACTOR = 0.75f;

private int capacity;
private float loadFactor;
private Entry<K, V>[] table;
private int size;

public HashMap() {
   this(DEFAULT_CAPACITY, DEFAULT_LOAD_FACTOR);
}

public HashMap(int capacity) {
   this(capacity, DEFAULT_LOAD_FACTOR);
}

public HashMap(int capacity, float loadFactor) {
   this.capacity = capacity;
   this.loadFactor = loadFactor;
   table = new Entry[capacity];
}
```

```
public void put(K key, V value) {
    if (key == null) {
        throw new NullPointerException();
    }

    int hash = key.hashCode();
    int index = hash % capacity;

    for (Entry<K, V> entry : table[index]) {
        if (entry.key.equals(key)) {
            entry.value = value;
            return;
        }
    }

    table[index] = new Entry<>(key, value);
    size++;

    if (size > loadFactor * capacity) {
        resize();
    }
```

```
}

public V get(K key) {
    if (key == null) {
        return null;
    }

    int hash = key.hashCode();
    int index = hash % capacity;

    for (Entry<K, V> entry : table[index]) {
        if (entry.key.equals(key)) {
            return entry.value;
        }
    }

    return null;
}

public boolean containsKey(K key) {
    return get(key) != null;
}
```

```
public boolean containsValue(V value) {
    for (Entry<K, V> entry : table) {
        if (entry.value.equals(value)) {
            return true;
        }
    }

    return false;
}

public void remove(K key) {
    if (key == null) {
        return;
    }

    int hash = key.hashCode();
    int index = hash % capacity;

    for (Entry<K, V> entry : table[index]) {
        if (entry.key.equals(key)) {
            table[index] = null;
```

```
            size--;
            break;
        }
    }
}

public int size() {
    return size;
}

public void clear() {
    for (int i = 0; i < capacity; i++) {
        table[i] = null;
    }

    size = 0;
}

private void resize() {
    int newCapacity = capacity * 2;
    Entry<K, V>[] newTable = new Entry[newCapacity];
```

```
    for (Entry<K, V> entry : table) {
        if (entry != null) {
            int hash = entry.key.hashCode();
            int index = hash % newCapacity;
            newTable[index] = entry;
        }
    }

    table = newTable;
    capacity = newCapacity;
}

private static class Entry<K, V> {

    K key;
    V value;

    public Entry(K key, V value) {
        this.key = key;
        this.value = value;
    }
}
```

```
}
```

This implementation of a hash map uses a linked list to store entries that have the same hash code. This allows for efficient lookups even when there are collisions.

# Jump search

Jump search is a search algorithm that finds the position of a target value within a sorted array. The algorithm works by repeatedly jumping over large sections of the array, until it reaches a section that is likely to contain the target value. The search continues until the target value is found or until the entire array has been searched.

Jump search is an efficient algorithm, with a time complexity of $O(\sqrt{n})$, where n is the number of elements in the array. This means that the time it takes to search the array grows logarithmically with the square root of the number of elements.

The jump search algorithm works as follows:

- Start with the first element of the array.
- Calculate the jump step, which is the square root of the number of elements in the array.
- Jump ahead by the jump step.
- Compare the element at the new position to the target value.
- If the element is equal to the target value, return the index of the element.
- If the element is less than the target value, repeat steps 2-5.
- If the element is greater than the target value, return -1.

The jump search algorithm can be implemented in Java as follows:

```
public class JumpSearch {


  public static int jumpSearch(int[] list, int target) {

    int n = list.length;


    // Calculate the jump step
```

```
    int jumpStep = (int) Math.floor(Math.sqrt(n));

    // Start at the first element of the array
    int current = 0;

    while (current < n) {
      // If the element at the current position is equal to the target value, return the index of the element
      if (list[current] == target) {
        return current;
      }

      // If the element at the current position is less than the target value, jump ahead by the jump step
      if (list[current] < target) {
        current += jumpStep;
      }
    }

    // If the target value is not found, return -1
    return -1;
  }
```

```
  public static void main(String[] args) {
    // Create a sorted array of integers
    int[] list = {1, 2, 3, 4, 5};

    // Search for the element 3 in the list
    int index = jumpSearch(list, 3);

    // Print the index of the element 3
    System.out.println(index);
  }
}
```

This code will print the following output:

```
2
```

Advantages and Disadvantages

The jump search algorithm has the following advantages:

- It is efficient, with a time complexity of O(√n).
- It can be used to search large arrays.

The jump search algorithm has the following disadvantages:

- It is not as simple to implement as some other search algorithms.
- It is not as efficient as some other search algorithms for small arrays.

Jump search is a powerful algorithm for searching sorted arrays. It is efficient and can be used to search large arrays. However, it is not as simple to implement as some other search algorithms.

# Interpolation search

Interpolation search is a search algorithm that finds the position of a target value within a sorted array. The algorithm works by comparing the target value to the middle element of the array. If the target value is equal to the middle element, the algorithm returns the index of the middle element. If the target value is less than the middle element, the algorithm searches the left half of the array using the same method. If the target value is greater than the middle element, the algorithm searches the right half of the array using the same method. The search continues until the target value is found or until the entire array has been searched.

Interpolation search is an efficient algorithm, with a time complexity of O(log(log(n))), where n is the number of elements in the array. This means that the time it takes to search the array grows logarithmically with the logarithm of the number of elements.

The interpolation search algorithm works as follows:

- Start with the middle element of the array.
- Calculate the interpolation factor, which is the difference between the target value and the middle element divided by the difference between the largest element in the array and the middle element.
- Use the interpolation factor to find the index of the element that is expected to contain the target value.
- Compare the element at the calculated index to the target value.
- If the element is equal to the target value, return the index of the element.
- If the element is less than the target value, search the right half of the array using the same method.
- If the element is greater than the target value, search the left half of the array using the same method.
- If the target value is not found, return -1.

The interpolation search algorithm can be implemented in Java as follows:

```
public class InterpolationSearch {


  public static int interpolationSearch(int[] list, int target) {
    int low = 0;
    int high = list.length - 1;


    while (low <= high) {
      // Calculate the interpolation factor
      int interpolationFactor = (target - list[low]) / (list[high] - list[low]);


      // Use the interpolation factor to find the index of the element that is expected to contain the target value
      int index = low + interpolationFactor * (high - low);


      // Compare the element at the calculated index to the target value
      if (list[index] == target) {
        return index;
      } else if (list[index] < target) {
        low = index + 1;
      } else {
        high = index - 1;
```

```
    }
   }

   // If the target value is not found, return -1
   return -1;
  }

  public static void main(String[] args) {
   // Create a sorted array of integers
   int[] list = {1, 2, 3, 4, 5};

   // Search for the element 3 in the list
   int index = interpolationSearch(list, 3);

   // Print the index of the element 3
   System.out.println(index);
  }
}
```

This code will print the following output:

```
2
```

Advantages and Disadvantages

The interpolation search algorithm has the following advantages:

- It is efficient, with a time complexity of O(log(log(n))).
- It can be used to search large arrays.
- It is more efficient than binary search for arrays that are uniformly distributed.

The interpolation search algorithm has the following disadvantages:

- It is not as simple to implement as some other search algorithms.
- It is not as efficient as some other search algorithms for arrays that are not uniformly distributed.

Interpolation search is a powerful algorithm for searching sorted arrays. It is efficient and can be used to search large arrays. However, it is not as simple to implement as some other search algorithms.

# Exponential search

Exponential search is a search algorithm that finds the position of a target value within a sorted array. The algorithm works by repeatedly dividing the array in half and searching the half that is most likely to contain the target value. The search continues until the target value is found or until the entire array has been searched.

Exponential search is an efficient algorithm, with a time complexity of O(log(log(n))), where n is the number of elements in the array. This means that the time it takes to search the array grows logarithmically with the logarithm of the number of elements.

The exponential search algorithm works as follows:

- Start with the entire array.
- Calculate the exponential factor, which is 2 raised to the power of the number of elements in the array divided by 2.
- Divide the array into two halves, one containing the elements at indexes 0 to the exponential factor - 1 and the other containing the elements at indexes exponential factor to n - 1.
- Compare the target value to the middle element of the first half.
- If the target value is equal to the middle element, return the index of the middle element.
- If the target value is less than the middle element, search the first half using the same method.
- If the target value is greater than the middle element, search the second half using the same method.
- If the target value is not found, return -1.

The exponential search algorithm can be implemented in Java as follows:

public class ExponentialSearch {

```
public static int exponentialSearch(int[] list, int target) {
  int low = 0;
  int high = list.length - 1;

  while (low <= high) {
    // Calculate the exponential factor
    int exponentialFactor = (int) Math.pow(2, high - low + 1);

    // Divide the array into two halves
    int middle = (low + high) / 2;
    int firstHalf = middle - exponentialFactor + 1;
    int secondHalf = middle + exponentialFactor - 1;

    // Compare the target value to the middle element of the first half
    if (list[firstHalf] == target) {
      return firstHalf;
    } else if (list[firstHalf] < target) {
      low = firstHalf + 1;
    } else {
      high = secondHalf - 1;
    }
  }
```

```
    // If the target value is not found, return -1
    return -1;
  }

  public static void main(String[] args) {
    // Create a sorted array of integers
    int[] list = {1, 2, 3, 4, 5};

    // Search for the element 3 in the list
    int index = exponentialSearch(list, 3);

    // Print the index of the element 3
    System.out.println(index);
  }
}
```

This code will print the following output:

```
2
```

Advantages and Disadvantages

The exponential search algorithm has the following advantages:

- It is efficient, with a time complexity of O(log(log(n))).
- It can be used to search large arrays.

- It is more efficient than binary search for arrays that are not uniformly distributed.

The exponential search algorithm has the following disadvantages:

- It is not as simple to implement as some other search algorithms.
- It is not as efficient as some other search algorithms for arrays that are uniformly distributed.

Exponential search is a powerful algorithm for searching sorted arrays. It is efficient and can be used to search large arrays. However, it is not as simple to implement as some other search algorithms.