**Module 19. Searching Algorithms**

**Learning Objectives**

* Understand the concept of searching algorithms
* Gain familiarity with various searching techniques such as linear search, binary search
* Learn the implementation details of each searching algorithm
* Understand time complexity of searching algorithms

# **Linear Search**

Linear search, also known as sequential searching, is a straightforward algorithmic approach. It begins from one end of a list and inspects each element sequentially until the desired element is located. It represents the most basic form of searching algorithm.

1. How it works?

Let's say we have a list of elements, and we're attempting to find a specific element, which we'll refer to as the "key."

A white square with black numbers

Description automatically generated

The following steps are as follows to find key = 3.

Step 1: We begin by comparing the first element in the list, which is '5', with the key. Since the key, which is 3, is not equal to 5, we move on to the next element. (3 ≠ 5)

A white square with black numbers

Description automatically generated

3 ≠ 5

Step 2: Next, we compare the second element in the list, which is '8', with the key. Since the key, which is 3, is not equal to 8, we proceed to the next element. (3 ≠ 8)

A white square with black numbers

Description automatically generated

3 ≠ 8

Step 3: Next, we compare the third element in the list, which is '10', with the key. Since the key, which is 3, is not equal to 10, we proceed to the next element. (3 ≠ 10)

A white square with black numbers

Description automatically generated

3 ≠ 10

Step 4: Next, we compare the fourth element in the list, which is '3', with the key. Since the key, which is 3, is equal to 3, we found the element.

A white square with black numbers

Description automatically generated

key

Otherwise, if the key is not found, it indicates that the key does not exist within the list.

1. Linear Search Implementation

(Example 1) Search for a number.

def linearSearch(nList, k):

    for i in range(0, len(nList)):

        if (nList[i] == k):

            return i

    return -1

def main():

    list = [5, 8, 10, 3, 99, 55]

    key = 3

    result = linearSearch(list, key)

    if result != -1:

        print(key, "is found at index", result)

    else:

        print(key, "is not found.")

if \_\_name\_\_ == "\_\_main\_\_":

    main()

(Output)

3 is found at index 3

(Example 2) Search for a string.

def linearSearch(nList, k):

    for i in range(0, len(nList)):

        if (nList[i] == k):

            return i

    return -1

def main():

    list = ['John', 'Tom', 'Joshua', 'Lee', 'Davis']

    key = 'Lee'

    result = linearSearch(list, key)

    if result != -1:

        print(key, "is found at index", result)

    else:

        print(key, "is not found.")

if \_\_name\_\_ == "\_\_main\_\_":

    main()

(Output)

Lee is found at index 3

1. Time Complexity

In the linear Search, every element should be compared at least once. In the best-case scenario, the complexity is when the key is found at the first index. Conversely, in the worst-case scenario, the complexity is when the key is either found at the last index or not found at all.

# **Binary Search**

Binary search is another widely used approach for searching within a list of values. However, for binary search to function correctly, it requires that the elements in the list are already ordered. Assume that the list is in ascending order.

1. How it works

To begin, we must designate three indices: 'low', 'mid', and 'high'. 'low' corresponds to the smallest element, 'high' to the largest, and 'mid' is calculated as "(low + high) / 2". We are attempting to find a specific element, which we'll refer to as the "key."

A black background with a black square

Description automatically generated with medium confidence

A white rectangular object with black numbers

Description automatically generated

Low

high

mid

(Example 1) The following steps are as follows to find key = 31.

Step 1: We begin by comparing the 'mid' element in the list, which is '19', with the key. Since the key, which is 31, is not equal to 19 and greater than 19.

31 ≠ 19

A white rectangular object with black numbers

Description automatically generated

high = 6

low = 0

mid = 3

In Step 2, since the key (31) is greater than 19, we don't need to search the half of the list that is less than the key. Therefore, we update the 'low' and 'mid' indices as follows: 'low = mid + 1 = 3 + 1 = 4', so 'low' indicates 23. 'mid = (low + high) / 2 = (4 + 6) / 2 = 5', which indicates 31. 'high' remains unchanged.

key

A white rectangular object with black numbers

Description automatically generated

mid = 5

low = 4

high = 6

Step 3: Moving forward, we assess the 'mid' element within the list, denoted as '31', in comparison to the key value, which is also 31. Given that the key matches the value of 31, we have successfully identified the desired element.

(Example 2) The following steps are as follows to find key = 10.

Step 1: We begin by comparing the 'mid' element in the list, which is '19', with the key. Since the key, which is 10, is not equal to 19 and less than 19.

10 ≠ 19

A white rectangular object with black numbers

Description automatically generated

low = 0

mid = 3

high = 6

In Step 2, since the key (10) is less than 19, we don't need to search the half of the list that is greater than the key. Therefore, we update the 'mid' and 'high' indices as follows: 'high = mid - 1 = 3 - 1 = 2', so 'high' indicates 15. 'mid = (low + high) / 2 = (0 + 2) / 2 = 1', which indicates 12. 'low' remains unchanged.

10 ≠ 12

A white rectangular object with black numbers

Description automatically generated

low = 0

high = 2

mid = 1

Step 3: Next, we compare the 'mid' element in the list, which is '12', with the key. Since the key, which is 10, is not equal to 12 and less than 12. And update indices as below: 'high = mid - 1 = 1 - 1 = 0', so 'high' indicates 10. 'mid = (low + high) / 2 = (0 + 1) / 2 = 0', which indicates 10. 'low' remains unchanged.

A white square with black numbers

Description automatically generated

low = mid = high = 0

Step 4: Moving forward, we assess the 'mid' element within the list, denoted as '10', in comparison to the key value, which is also 10. Given that the key matches the value of 10, we have successfully identified the desired element.

1. Binary Search Implementation

(Example 1) Search for a number.

def binarySearch(nList, k):

    low = 0

    high = len(nList) - 1

    while low <= high:

        mid = low + (high - low)//2

        if nList[mid] == k:

            return mid

        elif nList[mid] < k:

            low = mid + 1

        else:

            high = mid - 1

    return -1

def main():

    list = [10, 12, 15, 19, 23, 31, 45]

    key = 31

    result = binarySearch(list, key)

    if result != -1:

        print(key, "is found at index", result)

    else:

        print(key, "is not found.")

if \_\_name\_\_ == "\_\_main\_\_":

    main()

(Output)

31 is found at index 5

(Example 2) Search for a string.

def binarySearch(nList, k):

    low = 0

    high = len(nList) - 1

    while low <= high:

        mid = low + (high - low)//2

        if nList[mid] == k:

            return mid

        elif nList[mid] < k:

            low = mid + 1

        else:

            high = mid - 1

    return -1

def main():

    list = ['Davis', 'John', 'Joshua', 'Lee', 'Tom']

    key = 'Davis'

    result = binarySearch(list, key)

    if result != -1:

        print(key, "is found at index", result)

    else:

        print(key, "is not found.")

if \_\_name\_\_ == "\_\_main\_\_":

    main()

(Output)

Davis is found at index 0

1. Time Complexity

Let T(n) represent the time complexity for a binary search on a list of n elements. With each comparison, the number of elements is halved.

|  |  |  |
| --- | --- | --- |
| Depth | The number of elements | Complexity |
| 0 |  |  |
| 1 |  |  |
| 2 |  |  |
| 3 |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  | 1 |  |

Until the key value is found, indicating that only one element remains, the relationship holds true, where represents the initial number of elements and represents the number of comparisons made. By solving for in terms of , we find .

Therefore, the number of comparisons made is logarithmic with respect to the initial number of elements, leading to a time complexity of

**Summary**

1. Linear searching algorithm sequentially checks each element in a list until the target element is found.
2. Linear searching is suitable for small lists or unsorted data.
3. Time complexity of linear searching is O(n).
4. Binary searching algorithm requires a sorted list.
5. Binary searching algorithm divides the list in half at each step and compares the target element with the middle element to determine if it lies in the first or second half of the list.
6. Time complexity of binary searching algorithms is O(log n).

**Programming Exercises**

**Exercise 1: Find Maximum Element in a List**

Write a Python function that uses linear search to find the maximum element in a list.

**Exercise 2: Find Duplicates in a List**

Write a Python function that uses searching algorithms to find duplicate elements in a list and returns a list of duplicates.

**Exercise 3: Search for a Key in a Dictionary**

Write a Python function that takes a dictionary and a key as input and checks if the key exists using linear search.

**Exercise 4: Check if a String is a Palindrome**

Implement a Python function that uses two pointers to check if a given string is a palindrome by comparing characters from the start and end of the string.

**Exercise 5: Search for a Word in a Text File**

Implement a Python program that reads a text file and searches for a specific word, counting how many times it appears.

**Exercise 6: Phone Book Search**

Implement a simple phone book application using Python dictionaries. Write functions to add contacts, search for a contact by name, and delete contacts. Utilize linear search for searching contacts by name.

**Exercise 7: Application of Searching Algorithms**

Given a list of integers, implement a function find\_pair\_sum(arr, target\_sum) that uses any of the searching algorithms to find a pair of numbers in the list that sum up to the target\_sum. Return the indices of the pair if found, otherwise return None.