Data Structures and algorithms

Search Algorithms

Godofredo Avena

Nov 9, 2023

Search Algorithms

Search algorithms are techniques used to find elements in a data structure, such as an array or a list.

Different search algorithms have different time complexities, which affect their efficiency.

The choice of search algorithm depends on factors like the data structure, its size, and the specific problem requirements.

- Linear Search
- Binary Search
- Ternary Search
- Exponential Search
- Interpolation Search
- Jump Search

Time Complexities

- **O(1)** - Constant time: The algorithm takes a constant amount of time regardless of the input size.
- **O(log N)** - Logarithmic time: The algorithm's time complexity increases with the logarithm of the input size.
- **O(N)** - Linear time: The algorithm's time complexity increases linearly with the input size.
- **O(N log N)** - Linearithmic time: The algorithm's time complexity increases linearly and logarithmically with the input size.
- **O(N^2)** - Quadratic time: The algorithm's time complexity increases quadratically with the input size.

Space Complexities

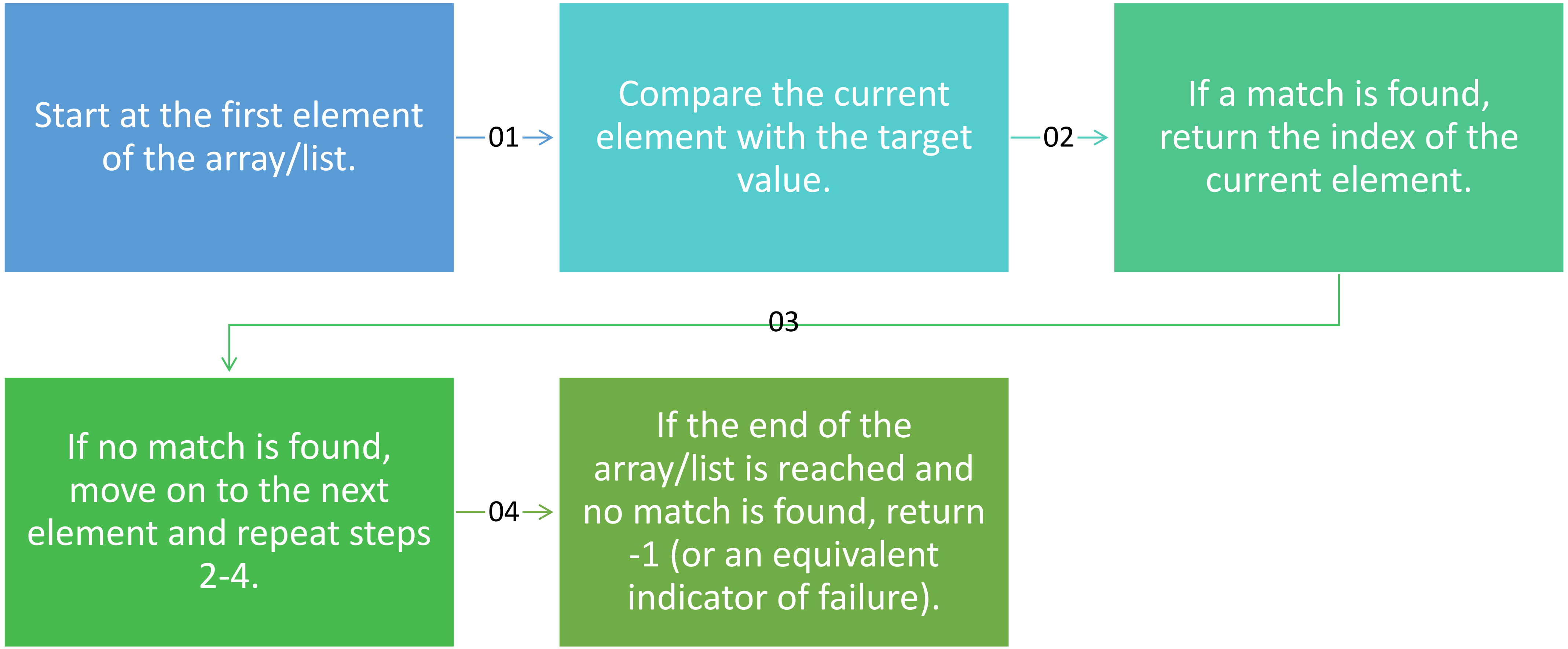
- **O(1)** - Constant space: The algorithm uses a constant amount of memory regardless of the input size.
- **O(log N)** - Logarithmic space: The algorithm's memory usage increases with the logarithm of the input size.
- **O(N)** - Linear space: The algorithm's memory usage increases linearly with the input size.
- **O(N^2)** - Quadratic space: The algorithm's memory usage increases quadratically with the input size.

Linear Search Algorithm

- The linear search algorithm is a simple and straightforward technique for searching an element in an array or list.
- It sequentially checks each element of the array or list until a match is found or the entire array/list has been searched.
- Suitable for small or unsorted data sets

- **Time Complexity:** O(n), where n is the number of elements in the array/list.
- **Space Complexity:** O(1)

Linear Algorithm Steps


Start at the first element of the array/list.
01
Compare the current element with the target value.
02
If a match is found, return the index of the current element.
03
If no match is found, move on to the next element and repeat steps 2-4.
04
If the end of the array/list is reached and no match is found, return -1 (or an equivalent indicator of failure).

Linear Search Python code

```python
def linear_search(arr, target):
    for i, element in enumerate(arr):
        if element == target:
            return i

    return -1
```

Binary Search Algorithm

- Binary search algorithm is an efficient search technique used for finding the index of a specific element in a sorted array.
- It works by dividing the search space in half and then recursively searching in the relevant half.
- The time complexity of the binary search algorithm is O(log2 N).

- The complexity of Binary Search Technique
- **Time Complexity:** O(1) for the best case. O(log2 n) for average or worst case.
- **Space Complexity:** O(1)

Binary Search Algorithm Steps

1. Define the search space.

2. Find the middle element of the search space.

3. Compare the middle element with the target value.

4. Update the search space based on the comparison and repeat steps 2-4 until the desired value is found or the search space becomes empty.

Binary Search Python code

```
def binary_search(arr, target):
    low, high = 0, len(arr) - 1
    while low <= high:
        mid = (low + high) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            low = mid + 1
        else:
            high = mid - 1
    return -1
```

Ternary Search Algorithm

- Ternary search algorithm is an efficient search technique used for finding the minimum or maximum value of a unimodal function or the index of a specific element in a sorted array.
- It works by dividing the search space into three equal parts and then recursively searching in the relevant part.
- Like the binary search, it also separates the lists into sub-lists. This procedure divides the list into three parts using two intermediate mid values. As the lists are divided into more subdivisions, so it reduces the time to search a key value.
- Time Complexity: O(log3 n)
- Space Complexity: O(1)

Ternary Search Algorithm Steps

1. Define the search space.

2. Divide the search space into three equal parts.

3. Determine which part of the search space contains the desired value.

4. Repeat steps 2-3 on the selected part until the desired value is found or the search space becomes too small.

Ternary Search Python code

```python
def ternary_search(arr, target, left, right):
    if left <= right:
        mid1 = left + (right - left) // 3
        mid2 = right - (right - left) // 3

        if arr[mid1] == target:
            return mid1
        elif arr[mid2] == target:
            return mid2
        elif target < arr[mid1]:
            return ternary_search(arr, target, left, mid1 - 1)
        elif target > arr[mid2]:
            return ternary_search(arr, target, mid2 + 1, right)
        else:
            return ternary_search(arr, target, mid1 + 1, mid2 - 1)

    return -1
```

Exponential Search Algorithm

- Exponential Search, also known as Exponential Binary Search or Doubling Search or galloping search, is an efficient search algorithm used to search for a specific element in a sorted array.
- It is particularly useful for large arrays, as it combines the efficiency of Binary Search with a faster initial search phase.
- The algorithm works by first finding the appropriate search interval in which the target element may be found, and then performing a Binary Search within that interval.

- **Time Complexity:** O(1) for the best case. O(log2 n) for average or worst case.
- **Space Complexity:** O(1)

Exponential Search Algorithm Steps

01

Define the search space.

02

Exponentially increase the search space until the desired element is within the range.

03

Perform binary search within the search range to find the index of the desired element.

Exponential Search Python code

```
def binary_search(arr, target, left, right):
    while left <= right:
        mid = left + (right - left) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return -1

def exponential_search(arr, target):
    if arr[0] == target:
        return 0
    i = 1
    n = len(arr)
    while i < n and arr[i] <= target:
        i *= 2
    return binary_search(arr, target, i // 2,
min(i, n) - 1)
```

Interpolation Search Algorithm

- For the interpolation searching technique, the procedure will try to locate the exact position using interpolation formula. After finding the estimated location, it can separate the list using that location. As it tries to find exact location every time, so the searching time reduces.

- **Time Complexity:** O(log2(log2 n)) for the average case, and O(n) for the worst case (when items are distributed exponentially)
- **Space Complexity:** O(1)

Interpolation Search Algorithm Steps

1. Define the search space.
2. Estimate the position of the target element using the linear interpolation formula.
3. Compare the estimated position with the target value.
4. Update the search space based on the comparison and repeat steps 2-4 until the desired value is found or the search space becomes empty.

Interpolation Search Python code

```
def interpolation_search(arr, target):
    low, high = 0, len(arr) - 1

    while low <= high and target >= arr[low] and target <= arr[high]:
        if low == high:
            if arr[low] == target:
                return low
            return -1
        #linear interpolation formula
        pos = low + ((target - arr[low]) * (high - low)) // (arr[high] - arr[low])

        if arr[pos] == target:
            return pos
        elif arr[pos] < target:
            low = pos + 1
        else:
            high = pos - 1

    return -1
```

Jump Search Algorithm

- Jump search is a search algorithm that works by first checking the first element of a sorted array, then skipping ahead by a fixed step size until it finds an element that is greater than the target value. Once the algorithm has found an element that is greater than the target, it performs a linear search between the previous index and the current index to find the target value.

- Time Complexity: $O(\sqrt{n})$
- Space Complexity: $O(1)$

step-by-step breakdown of how jump search works:

Determine the block size: The algorithm starts by determining the block size to jump ahead by. The optimal block size is sqrt(n), where n is the length of the array being searched.

Jump ahead by block size: Starting at index 0, the algorithm checks if the value at index 0 is equal to the target value. If it is, the algorithm stops and returns the index. If the value at index 0 is not equal to the target value, the algorithm jumps ahead by the block size and checks the value at that index.

Repeat until target value is found: The algorithm continues to jump ahead by the block size until it finds a value that is greater than the target value. Once a value greater than the target value is found, the algorithm performs a linear search between the previous index and the current index to find the target value.

Jump Search Python code

```python
def jump_search(arr, target):
    n = len(arr)
    jump = int(math.sqrt(n))
    left, right = 0, 0

    while right < n and arr[right] < target:
        left = right
        right += jump

    for i in range(left, min(right, n)):
        if arr[i] == target:
            return i

    return -1
```

Questions

Thank you!

Group Laboratory Exercise - Search Algorithms
Deadline : 11:29 pm, Nov 19, 2023

Objective: To implement and compare the performance of various search algorithms on different data sets.

Materials:

Computer with Python installed (https://www.python.org/downloads/)
Integrated development environment (IDE) for writing Python code (https://code.visualstudio.com/download)
Download code: "git clone https://github.com/gavena/PUPSearchAlgo.git" or
https://github.com/gavena/PUPSearchAlgo/archive/refs/heads/main.zip

Instructions:

Part 1: Implementing Search Algorithms

Implement all search algorithms in Python (code already provided) :

- Linear search
- Binary search
- Jump search
- Exponential search
- Interpolation search
- Ternary Search

Note: You can refer to the earlier explanations and the code provided for each algorithm to help you with the implementation.

Part 2: Generating Test Data

Create python code three different data sets for testing the search algorithms:

a. Small data set: Generate a sorted list of 100 integers.
b. Medium data set: Generate a sorted list of 1,000 integers.
c. Large data set: Generate a sorted list of 10,000 integers.

Choose a random target element from each data set to search for.

Part 3: Testing and Comparing Search Algorithms

For each search algorithm, create a code to measure the time taken to find the target element in each data set. A code has been given but you can change it to get a more efficient answer.

Record the execution times for each algorithm and data set combination.

Part 4: Analysis and Conclusion

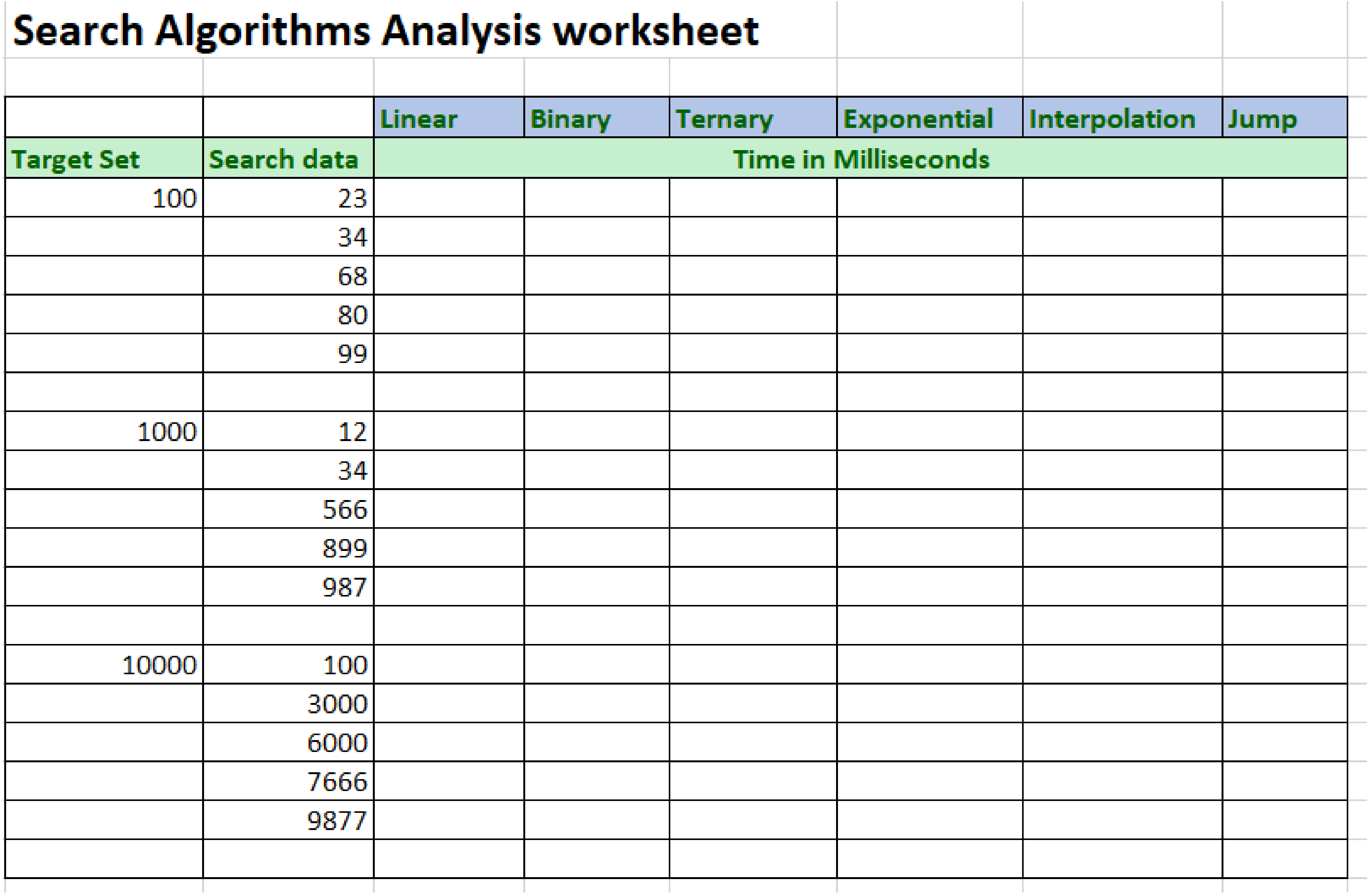
Create a table or graph to visualize the performance of each search algorithm on the different data sets.

Analyze the results and answer the following questions:

- a. Which search algorithm performed the best overall?
- b. Did any search algorithms perform better on specific data sets?
- c. How did the size of the data set affect the performance of the search algorithms?
- d. Write a brief conclusion summarizing your findings

Deliverables:

Python code for each search algorithm (already provided) and the python code on how to generated dataset and executing the data set.
Table and graph comparing the performance of the search algorithms on the different data sets (see sample)
Analysis and conclusion of the exercise



Search Algorithms Analysis worksheet

		Linear	Binary	Ternary	Exponential	Interpolation	Jump
Target Set	Search data	Time in Milliseconds					
100	23						
	34						
	68						
	80						
	99						
1000	12						
	34						
	566						
	899						
	987						
10000	100						
	3000						
	6000						
	7666						
	9877						