Search

Notes from Charlie Obimbo and revised by Yan Yan.

Contents

1. Definition
2. Linear Search
3. Binary Search
4. Interpolation Search
5. Recursive Algorithm
6. Time Complexity Analysis

Learning Objectives

1. Describe the process of linear search, binary search, and interpolation search
2. Define recursive algorithm and apply it to some search algorithms
3. Analyze and calculate the best-case and worst-case complexities of search algorithms

Search: Definition

- Sometimes referred to as seek, a search is a function or process of finding letters, words, files, web pages, or other data.
- An **algorithm** is a sequence of steps for accomplishing a task
- An algorithm's **runtime** is the time the algorithm takes to execute.

Linear Search

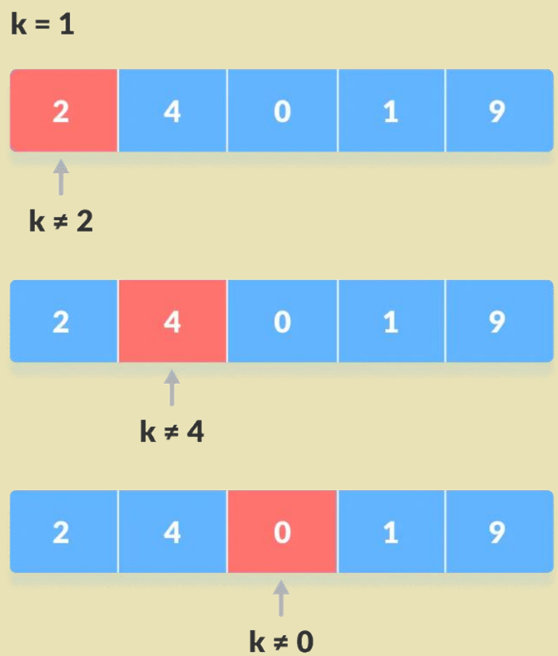
- Linear search is a search algorithm that starts from the beginning of a list, and checks each element until the search key is found or the end of the list is reached
  - mostly used to search an unordered list of elements

- **Linear Search** Example:
- Find the position of k=1 in the array:

2	4	0	1	9

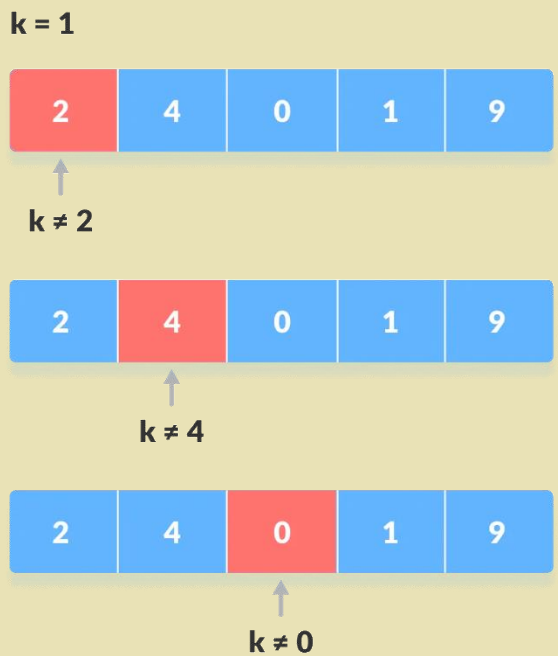
- Example:
- Find the position of k=1 in the array:

- Start from the first element, compare k with each element x:

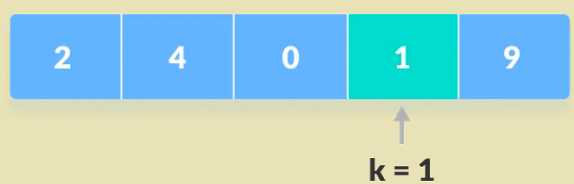


- Example:
- Find the position of k=1 in the array:

- Start from the first element, compare k with each element x:



- If x == k, return the index



Function

```
LinearSearch(numbers, numbersSize, key) {
      for (int i = 0; i < numbersSize; ++i) {
         if (numbers[i] == key) {
            return i;
         }
      }
            return -1; // not found
}
```

Linear Search

- An algorithm typically uses a number of steps proportional to the size of the input.
- For a list with **n** elements, linear search requires **at most n** comparisons. The algorithm is said to require "on the order" of **n** comparisons.
- Time Complexity O(n)

Linear Search Exercise

- Given a list of 10,000 elements, and if each comparison takes 2 µs, what is the fastest possible runtime for linear search? What's the longest possible runtime?

Exercise is from the textbook 2.1 Searching and algorithms

Linear Search Exercise

- Given a list of 10,000 elements, and if each comparison takes 2 µs, what is the fastest possible runtime for linear search? What's the longest possible runtime?

2 µs, and 20,000 µs

- How to improve the search efficiency?

Exercise is from the textbook 2.1 Searching and algorithms

Binary Search

- If the list is **sorted** and directly accessible (such as an **array**), we could use the binary search to speed up
  - Search starts with the middle element
  - If the search key is found, the algorithm returns the matching location.
  - If the search key is not found, the algorithm repeats the search on the remaining left sublist or the remaining right sublist

Binary Search

- Eliminate half of the search space each step
- The search terminates when the element is found or the search space is empty (element not found).

Binary Search

- If initial length of array $=n$
  - Iteration 1 - Length of array$=n/2$
  - Iteration 2 - Length of array $=(n/2)/2=n/ 2^2$
  - …
  - Iteration k - Length of array $=n/2^k$
  - After k iterations, the size of the array becomes 1
  - Time Complexity $O(\log_2 n)$.

- **Binary Search** Example:
- Find the position of k=23 in the array:

2	5	8	12	16	23	38	56	72	91

- **Binary Search** Example:
- Find the position of k=23 in the array:

2	5	8	12	16	23	38	56	72	91

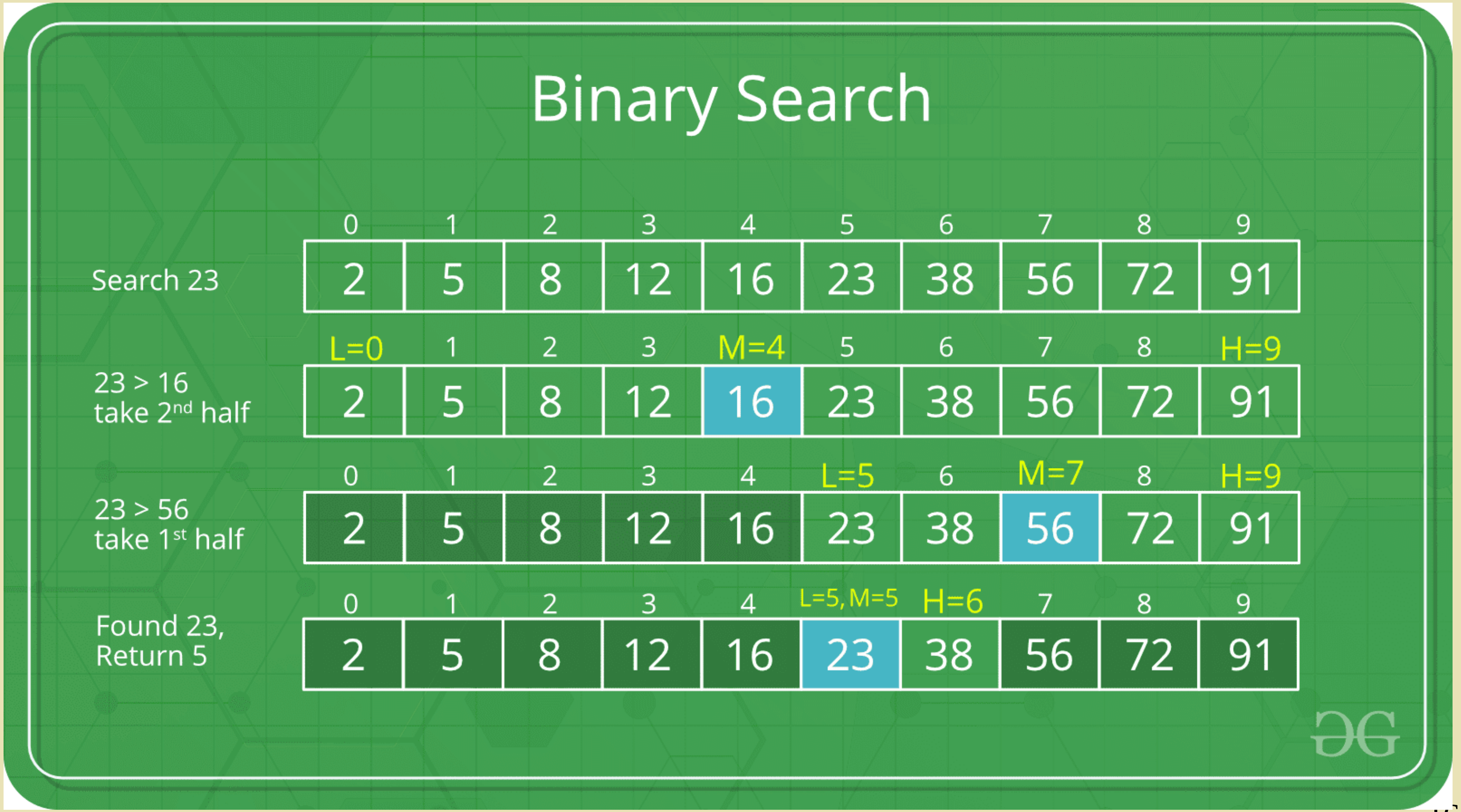


Figure is from https://www.geeksforgeeks.org/complexity-analysis-of-binary-search/

```
BinarySearch(numbers, numbersSize, key) {
   int mid = 0, low = 0, high;
   high = numbersSize – 1;
   while (high >= low) {
      mid = (high + low) / 2;
      if (numbers[mid] < key) {
         low = mid + 1;
      }
      else if (numbers[mid] > key) {
         high = mid – 1;
      }
      else {
         return mid;
      }
   }
    return -1; // not found
}
```

Binary Search

- Q: What is the best case and worst case?
- Best O(1)
- Worst $O(\log_2 n)$.

Interpolation Search

- Improved version of binary search for uniformly distributed elements
  - "Guess" the position of the searched elements
  - The calculation is done based on the values at the bounds of the search space and the value to be searched.
    - Usually called a prob
- If prob is the searched element, return; or narrow down the search space base on the prob

Interpolation Search

- Example A=[1,3,5,7,9,11], x=3
- $prob = low + \frac{(x - A[low])(high - low)}{(A[high] - A[low])}$
- $prob = 0 + \frac{(3-1)(5-0)}{(11-1)} = 1$

Interpolation Search Algorithm

- 1. Initialize low and high indices to the start and end of the array, respectively.
- 2. Calculate the probe position using the interpolation formula.
- 3. Compare the probe element with the target element.
  - a). If they are equal, the search is successful.
  - b). If the probe element is greater, update the high index to the probe position minus one.

https://www.javatpoint.com/interpolation-search

Interpolation Search Algorithm Cont'd

   – c.) If the probe element is smaller, update the low index to the probe position plus one.

- 4. Repeat steps 2-3 until the target element is found or the low index exceeds the high index.

https://www.javatpoint.com/interpolation-search

```c
// If x is present in a sorted integer array, then returns index of it, else returns -1.

int interpolationSearch(int arr[], int arr_size, int x)
{
        int pos;
        // Initialize the lower and higher positions for the search
        int lo=0, hi=arr_size-1;
        // Perform interpolation search while the search space is valid
        while (lo <= hi && x >= arr[lo] && x <= arr[hi]) {
                // Probing the position with keeping uniform distribution in mind.
                pos = lo + (((double)(hi - lo) / (arr[hi] - arr[lo])) * (x - arr[lo]));

                // Condition of target found
                if (arr[pos] == x)
                        return pos;

                // If x is larger, x is in right sub array
                else if (x>arr[pos])
                        lo= pos + 1;
                // If x is smaller (x<arr[pos]), x is in left sub array
                else
                        hi = pos – 1;
        }
        return -1;
}
```

https://www.w3resource.com/c-programming-exercises/searching-and-sorting/c-search-and-sorting-exercise-19.php/

Recursive Algorithm

- A recursive algorithm is an algorithm that breaks the problem into *smaller subproblems* and applies the algorithm itself to solve the smaller subproblems.
  - Binary search

Binary Search Recursive Algorithm

```
BinarySearch(numbers, low, high, key) {
   if (low > high)
      return -1

   mid = (low + high) / 2
   if (numbers[mid] < key) {
      return BinarySearch(numbers, mid + 1, high, key)
   }
   else if (numbers[mid] > key) {
      return BinarySearch(numbers, low, mid - 1, key)
   }
   return mid
}
```


Figure is from the textbook Figure 2.8.2: BinarySearch recursive algorithm.

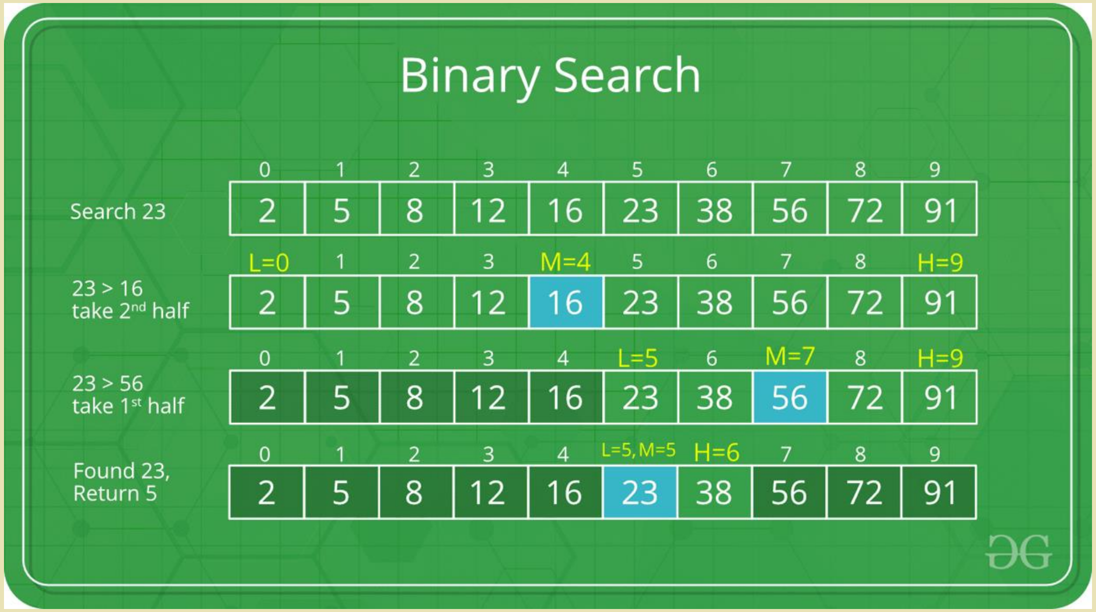
Interpolation Search Recursive

```
// If x is present in arr[0..n-1], then returns index of it, else returns -1.
int interpolationSearch(int arr[], int lo, int hi, int x)
{
        int pos;
        // array is sorted
        if (lo <= hi && x >= arr[lo] && x <= arr[hi]) {
                // Probing the position with keeping uniform distribution in mind.
                pos = lo + (((double)(hi - lo) / (arr[hi] - arr[lo])) * (x - arr[lo]));

                // Condition of target found
                if (arr[pos] == x)
                        return pos;

                // If x is larger, x is in right sub array
                if (arr[pos] < x)
                        return interpolationSearch(arr, pos + 1, hi, x);

                // If x is smaller, x is in left sub array
                if (arr[pos] > x)
                        return interpolationSearch(arr, lo, pos - 1, x);
        }
        return -1;
}
```


https://www.geeksforgeeks.org/interpolation-search/

Binary Search Time Complexity

- Key: how many times (x) we need to call the recursive function BinarySearch?



Binary Search Time Complexity

- Assume array length is **n,** each time we divide **n** by 2

$$(\frac{1}{2})^x n = 1$$

$$2^x = n$$

$$\mathrm{x} = \log_2 n$$

That's about this lecture!