Search Algorithms

Objectives:

- Learn how to implement search algorithms
  - Linear Search
  - Binary Search
- Able to explain search algorithm
- Search in data from a file and write search result to a file

1. Introduction

A **search algorithm** is a computation technique to retrieve information from some data structure.

By using suitable **encodings within the data structure** and **searching methods**, searching could be made **time** or **space** efficient.

Example - Yellow Pages Example

Records in printed Yellow Pages (*data structure*) are sorted lexicographically (*encoded*) according to names (such as company names, phone registration names).

- Thus, we could look up (*search*) someone's phone number given his or her name.
- However, it is very time consuming to find out someone's name if a phone number is given.

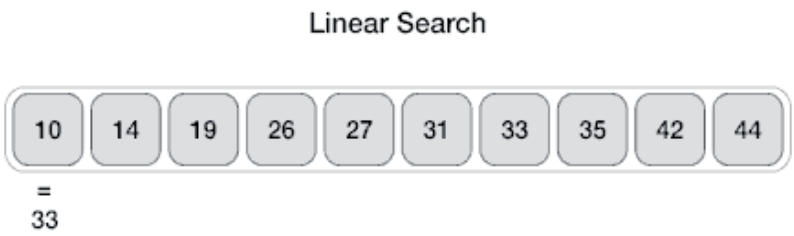
Categories of Algorithm

Based on the type of search method, these algorithms are generally classified into two categories:

Sequential Search:

- The list or array is traversed sequentially and every element is checked.
- The list is commonly unsorted.

Example: Linear Search for value of 33



Interval Search:

- These algorithms are specifically designed for searching in **sorted** data-structures.

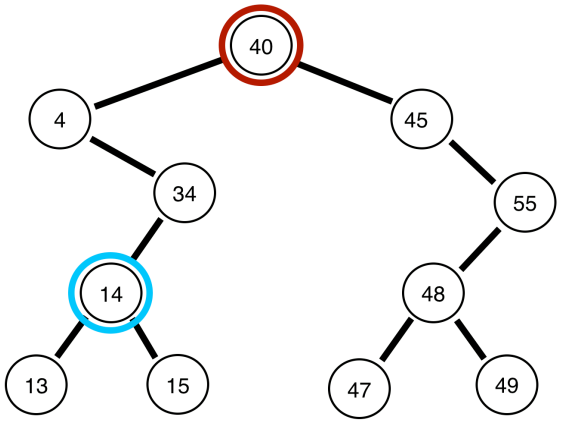
- These type of searching algorithms are much more efficient than Linear Search as they repeatedly target the center of the search structure and divide the search space in half.

Example 1: Binary Search for value of 47

Search for 47

0	4	7	10	14	23	45	47	53

Example 2: Search for value of 14 in a Binary Search Tree



Common Search Algorithms

- Linear Search
- Binary Search

Recap: Built-in Search Functions

We have been performing searching in Python. Here are some recaps.

Example:

- Generate a list `s` with 10 random integers between 1 and 10.

In [1]:

```
import random

# Generate 10 random integers between 1 and 20 (inclusive).
s = [random.randint(1,10) for i in range(10)]
print(s)
```

```
[6, 4, 1, 3, 3, 8, 6, 6, 8, 3]
```

Membership Operators

Python's membership operators, `in` and `not in`, test for membership in a sequence, such as strings, lists, or tuples.

- It returns boolean value `True` or `False`.

In [2]:

```
x = 10
print(x in s)

y = 5
print(y not in s)
```

```
False
True
```

Search an Item in List

List provides a method `index()` which searches an element in the list and returns its index.

- If the same element is present more than once, the method returns the index of the first occurrence of the element.
- If element is not found, it raises an exception.

Example:

- Ask user for an integer and check if it exists in the list.

In [3]:

```
x = int(input('Enter an integer [1-10]: '))

try:
    idx = s.index(x)
    print('Found at index =', idx)
except:
    print('Not found')
```

```
Enter an integer [1-10]: 3
Found at index = 3
```

Search Multiple Items in List

List comprehension can be used to perform search and return index values if value exists in a list.

In [4]:

```
x = int(input('Enter an integer [1,10]: '))

y = [ i for i,v in enumerate(s) if v == x]
if y:
    print('Value', x, 'found at index ', y)
else:
    print('Not found')
```

```
Enter an integer [1,10]: 3
Value 3 found at index  [3, 4, 9]
```

Search Key in Dictionary

To search in keys of a dictionary, simply use its `get()` method. It returns its value if the key exists in dictionary, else it returns `None`.

In [5]:

```
d = {'a':1, 'b':2, 'c':3}

x = d.get('b')
y = d.get('d')
print(x, y)
```

```
2 None
```

Search Value in Dictionary

It's also common to search through values of a dictionary. To do that, apart from using `for-loop`, we can also make use of list comprehension.

In [6]:

```
d = {'a':1, 'b':2, 'c':3, 'd':1}

x = [k for k,v in d.items() if v == 1]
print(x)
```

```
['a', 'd']
```

2. Linear Search

Linear search searches an item in a given list **sequentially** till the end of the collection.

- It is one of the simplest searching algorithms
- It commonly uses `for-loop` or `while-loop` to iterate through the collection.

Exercise:

Implement a function `linear_search(array, val)` which searches the list `array` for a value `val`. It returns first index if the value is found, else it return `-1`.

Hint: How to index of a value in a `for-loop` ?

In [7]:

```
def linear_search(array, val):
    for index, element in enumerate(array):
        if element == val:
            return index
    return -1
```

In [8]:

```
## TEST CODE
print(s)
idx = linear_search(s,5)
print(idx)
```

```
[6, 4, 1, 3, 3, 8, 6, 6, 8, 3]
-1
```

Time Complexity

In linear search, all items are searched one-by-one to find the required item.

If the sample size is `n` ,

- The best-case lookup to find an item is 1, i.e., the item is at the head of the list.
- The worst-case lookup to find an item is `n` , i.e. the item is at the end of the list.
- The average lookup to find an item is `n/2` .

The time complexity of linear search is `O(n)` , meaning that the time taken to execute increases with the number of items in our input list.

Find Average Execution Time using `%%timeit`

Jupyter Notebook provides a magic function `%timeit` and `%%timeit` to time a code execution.

- `%timeit` is used to time a single line of statement
- `%%timeit` is used to time all codes in a cell. `%%timeit` must be placed at first line of cell.

In [11]:

```
import random
array = [random.randint(1,100) for i in range(20)]
```

In [12]:

```
%timeit linear_search(array, 5)
```

```
2.94 µs ± 1.12 µs per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

In [13]:

```
%%timeit
linear_search(array, 5)
```

```
1.65 µs ± 201 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

3. Binary Search

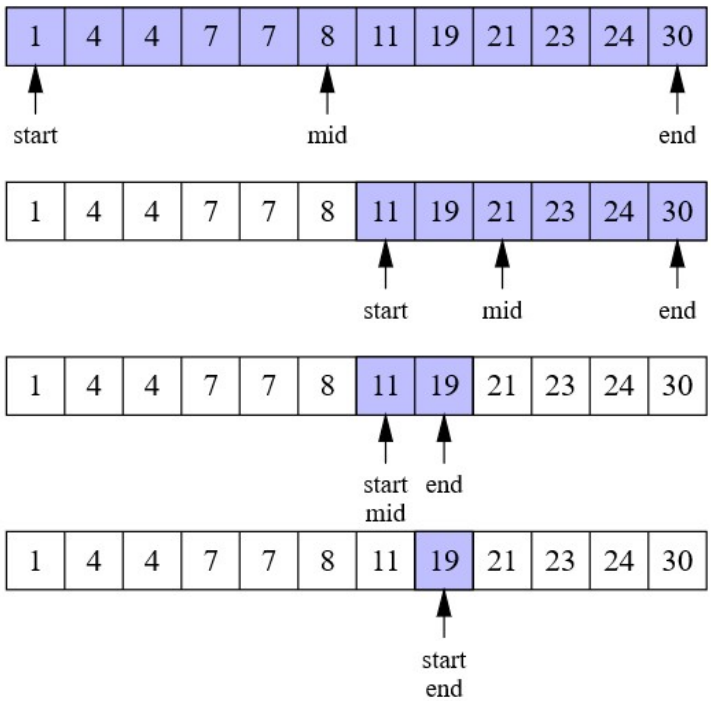
Binary Search uses a **divide and conquer** methodology. It is faster than linear search. But it requires the array to be **sorted**.

How it Works?

- First check the MIDDLE element in the list.
- If it is the value we want, we can stop.
- If it is HIGHER than the value we want, we repeat the search process with the portion of the list BEFORE the middle element.
- If it is LOWER than the value we want, we repeat the search process with the portion of the list AFTER the middle element.

Example:

Find value 19 in a sorted list of integers.



The binary search algorithm can be written using either iterative loops or recursive function.

Implementation with Loops

We uses `while-loop` since we need to shift two pointers, `left` and `right`, until `left` is greater than `right`.

In [ ]:

```
def binary_search_iterative(arr,target):
    left=0
    right=len(arr)-1

    while left<=right:
        mid=(left+right)//2
        if arr[mid]==target:
            # bingo
            return True
        else:
            if target>arr[mid]:
                # move left pointer
                left = mid+1
            else:
                # move right pointer
                right = mid-1

    return False
```

Implementation with Recursive Function

- The base case of the function is when the input array is empty.
- The recursion happens and we narrow the problem into half of the array.

In [ ]:

```
def binary_search_recursive(arr,target):
    # Base Case!
    if len(arr) == 0:
        return False

    # Recursive Case
    mid = len(arr)//2
    if arr[mid]==target:
        return True
    else:
        # Call again on second half
        if target<arr[mid]:
            return rec_bsearch(arr[:mid],target)
        # Or call on first half
        else:
            return rec_bsearch(arr[mid+1:],target)
```

Time Complexity

If the same size is `n`,

- The best-case lookup is 1.
- The worst-case loopkup is `log(n) / log(2)`.
- Every iteration, the sample size is halved: `n/2`, `n/4`, ... which become 1 after `log(n) / log(2)` iterations.