Search Algorithms

H2 Computing 2020

First edition (2020)

Introduction

A search for an element involves matching attributes (usually the name or value) against one or more criteria for each element in a collection, and returning the index of the element. If the array is sorted according to the required criteria, the task becomes easier.

Regardless of whether the array is sorted or unsorted, the time taken to search for the matching element increases with the number of elements in the array to be searched.

We will cover 3 search algorithms here: the Linear Search, the Hash Table Search, and the Binary Search.

Search algorithms for unsorted arrays

Linear Search

The Linear Search is the simplest algorithm among the three. It involves matching each element of the array against the search criteria, and returning the first match.

Pseudocode for Linear Search

```
FUNCTION linearSearch(array, target)
FOR i in 0 to array.LENGTH-1
    if array[i] == target
    RETURN i
ENDFOR
ENDFUNCTION
```

Exercises

How would you modify the above function to return the first element *smaller* than `target`?

```
FUNCTION linearSearch(array, target)
FOR i = 0 to array.LENGTH-1
    if array[i] < target
    RETURN i
ENDFOR
ENDFUNCTION
```

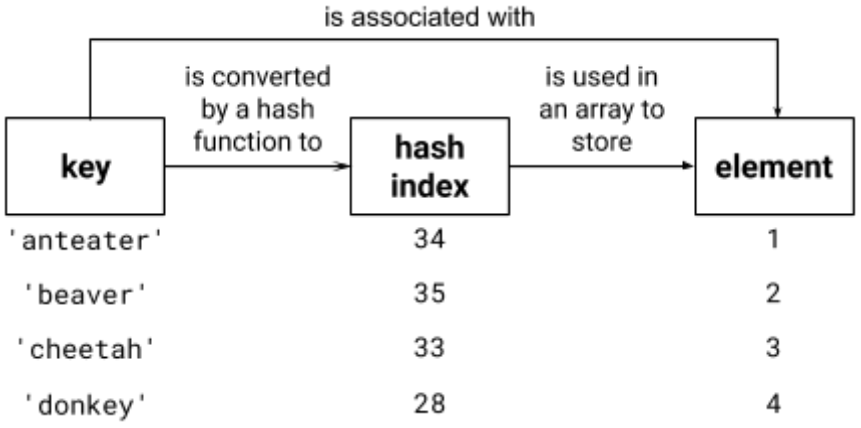
How would you modify the above function to return the first element whose `name` **attribute** matches `target`? (Assume that the array elements are objects with this attribute.)

```
FUNCTION linearSearch(array, target)
FOR i in 0 to array.LENGTH-1
    if array[i].name == target.name
    RETURN i
ENDFOR
ENDFUNCTION
```

Hash Table Search

The Hash Table Search takes advantage of a phenomenon: retrieving an element using its index is *much faster* than searching for an element that matches a criterion. We would be able to retrieve a value from an array using a **key** much more quickly than searching for the index of the value. By constructing an array to follow special rules, we can use a lookup instead of a search to return the target value.

Instead of storing an element by index, we **map a unique key to each element** in a hash table. This unique key is converted into a hash index by a **hash function**.



Hash functions

> **hash** (*noun*)
> a mixture, as of things used before in different forms; rehash

A perfect hash function:

1. Takes an **input key** of *any length*
2. And returns an **output hash index** of *fixed size*

A hash index:

- Is **unique** to the key
- for two keys **is the same if the keys are equal**

To be hashable, a key:

- must be **immutable** (so that its hashed index is always the same)

A perfect hash table creates a one-to-one (bijective) mapping between key and element.

Hashing in Python

Python provides a `hash()` function that takes in any input key and returns an `int` value.

```
>>> hash('anteater')
1655975154239661184
>>> hash('beaver')
-3003524256904139215
>>> hash('cheetah')
5626345346233235883
>>> hash('donkey')
-3056701103000864772
```

These integer values are too large to be used directly as indexes, so we have to reduce them to a suitable size. For instance, if we expect up to 100 elements to be stored, we would convert the integers to values between 0 and 99 inclusive. We can do this by:

1. taking each integer modulo 50[1] (to get indexes between 0 to 49)
2. adding 50 to the index if the integer is positive[2]

The following Python code demonstrates how a `customHash` function can do this:

```
def customHash(string):
    index = hash(string)
    if index < 0:
        index = index % 50       # index 0-49
    else:
        index = index % 50 + 50  # index 50-99
    return index
>>> customHash('anteater')
34
>>> customHash('beaver')
35
>>> customHash('cheetah')
33
>>> customHash('donkey')
28
```

[1] Note that the modulo (%) operator in Python always returns a positive value.
[2] This allows us to use indexes 0-49 for negative integers, and indexes 50-99 for positive integers.

Creating a `HashTable` class

With the unique key generated by `customHash`, we can now store values into an array using their key.

Since we do not want this array being accessed directly by the user, we **abstractise** its implementation by **encapsulating** it into a class.

```
class HashTable:
    def __init__(self, size):
        self.size = size
        self.__data =  [None] * self.size

    def __repr__(self):
        return f'HashTable({self.__data})'

    def __hash(self, string):
        index = hash(string)
        if index < 0:
            index = index % (self.size // 2)
        else:
            index = index % (self.size // 2) + (self.size // 2)
        return index

    def add(self, key, element):
        index = self.__hash(key)
        self.__data[index] = element

    def get(self, key):
        index = self.__hash(key)
        return self.__data[index]
```

Storing and retrieving values from `HashTable`

Once we initialise the hash table:

```
>>> animals = HashTable(100)
```

We can store keys along with values using the defined `.add()` method:

```
>>> animals.add('anteater', 1)
>>> animals.add('beaver', 2)
>>> animals.add('cheetah', 3)
>>> animals.add('donkey', 4)
```

And we can retrieve the values using the keys using the defined `.get()` method:

```
>>> animals.get('anteater')
1
>>> animals.get('beaver')
2
>>> animals.get('cheetah')
3
>>> animals.get('donkey')
4
```

(optional) Python dunders for working with keys and values

Python has `__setitem__` and `__getitem__` dunders for working with keys and values, which we can use to replace `.add()` and `.get()` respectively

```
def __setitem__(self, key, element):
    index = self.__hash(key)
    self.__data[index] = element

def __getitem__(self, key):
    index = self.__hash(key)
    return self.__data[index]
```

Implementing these dunders allows us to use a more convenient syntax to add key-value pairs to the hash table:

```
>>> animals['anteater'] = 1
>>> animals['beaver'] = 2
>>> animals['cheetah'] = 3
>>> animals['donkey'] = 4
```

And we can use a similar syntax to retrieve values using keys:

```
>>> animals['anteater']
1
>>> animals['beaver']
2
>>> animals['cheetah']
3
>>> animals['donkey']
4
```

This lets us define classes that work like Python `dicts`.

Resolving hash table collisions (out of syllabus)

While a *perfect* hash function should give a unique key for every input key, in reality we know that is not possible. Even a very good hash function will occasionally give identical indexes for two different keys. This is known as a **hash table collision**.

This is likely to happen with:

- a low-quality hash function
- a very small hash table

Following from the earlier example, if we initialise a very small hash table:

```
>>> smallhashtable = HashTable(4)
```

We will end up with the following set of key-value pairs, and the hash index associated with the key:

key	hash index	value
`'anteater'`	`0`	1
`'beaver'`	3	2
`'cheetah'`	`0`	3
`'donkey'`	3	4

If we store `'anteater'` key with value 1, followed by `'cheetah'` key with value 3, the second value of 3 will overwrite the existing value of 1 at index `0`.

A number of strategies exist to resolve such collisions and allow the hash table to prevent key-value pairs from being overwritten. Here, we only focus on three strategies: open addressing, separate chaining, and hash table resizing.

Disambiguation

In the hash table below, two key-value pairs have been stored in indexes 0 and 3:

index	key-value
0	`'anteater': 1`
1	
2	
3	`'beaver': 2`

If we attempt to store another key-value pair `'cheetah': 3`, `'cheetah'` returns a hash index of 0, which will collide with the existing key-value of `'anteater': 1`. We need a way to clarify which key a certain value is mapped to; this is known as **disambiguation**. Storing the key with the value allows us to disambiguate by checking if the retrieved key is the same.

Collision strategy 1: Open addressing

In the open addressing strategy, if a hash table collision occurs resulting in an index being unavailable, a **rehashing function** is used to generate another index. A simple rehashing function that can be used is to simply increment the index by 1 until an empty slot is found.

Using the open addressing strategy with a rehashing function that increments the index by 1, we would rehash `'cheetah'` to give us an index of 1. After storing `'cheetah': 3` at this rehashed index, the table now looks like this:

index	key-value
0	`'anteater': 1`
1	`'cheetah': 3`
2	
3	`'beaver': 2`

When we attempt to retrieve the value associated with the `'cheetah'` key:

1. `'cheetah'` is hashed to the index 0, and the key-value pair at index 0 is retrieved.
2. It is discovered that the retrieved key is not `'cheetah'`. Rehashing `'cheetah'` gives us a new index of 1.
3. The key-value pair at index 1 is retrieved. Since the retrieved key matches `'cheetah'`, the associated value 3 is returned.

Collision strategy 2: Separate chaining

In the separate chaining strategy, key-value pairs that collide (i.e. have the same index) are simply stored under the same index as a collection. After storing `'cheetah': 3` at the same index 0, the table now looks like this:

index	key-value
0	`'anteater': 1, 'cheetah': 3`
1	
2	
3	`'beaver': 2`

When we attempt to retrieve the value associated with the `'cheetah'` key:

4. `'cheetah'` is hashed to the index 0, and the first key-value pair at index 0 is retrieved.
5. It is discovered that the retrieved key is not `'cheetah'`. The second key-value pair at index 0 is retrieved.
6. Since this retrieved key matches `'cheetah'`, the associated value 3 is returned.

Collision strategy 3: Hash Table resizing

In the resizing strategy, ifa collision happens, the hash table is expanded in the hope that with more available indexes, the likelihood of a hash collision is lowered. This also means that all the keys need to be rehashed, and their values moved to the rehashed index.

By expanding the hash table to 20 slots, we get the following key-value pairs and hashes:

key	hash index	value
`'anteater'`	6	1
`'beaver'`	13	2
`'cheetah'`	2	3
`'donkey'`	15	4

Thus, the hash table collision is resolved. This strategy can be computationally expensive when resizing large hash tables.

Search algorithms for sorted arrays

Binary Search

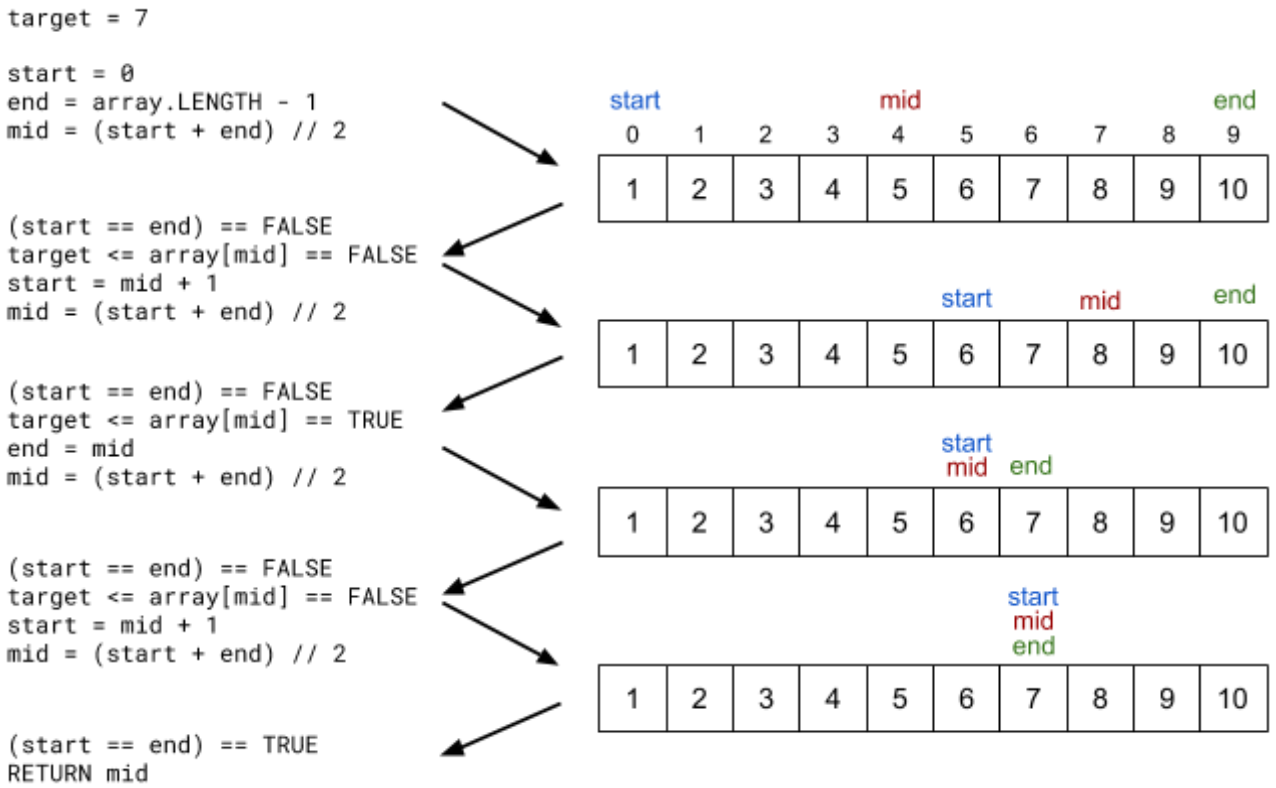
> **binary** (*adjective*)
> relating to, composed of, or involving two things.

The Binary Search algorithm is so named, not because it involves binary numbers, but it involves splitting the search space (i.e. the subarray to be searched) in two each time.

If we assume that the array is sorted, we can divide the array in the middle into two subarrays: a less-than-middle subarray, and a greater-than-middle subarray. The element we are looking for must thus be in either subarray, and we may deduce which it is in by checking if it is less than or greater than the middle element.

Rather than continuously dividing the array, which would make it difficult to keep track of the original index, we just keep track of three values that demarcate the current search space: `start`, `mid` (middle), and `end`.

Illustration of Binary Search



Pseudocode for Binary Search (iterative)

```
FUNCTION binarySearch(array, target)
start = 0
end = array.LENGTH - 1
mid = (start + end) // 2
WHILE (start != end)
    IF target <= array[mid] THEN
        end = mid
    ELSE
        start = mid + 1
    ENDIF
    mid = (start + end) // 2
ENDWHILE
RETURN mid
ENDFUNCTION
```

Time Complexity of Search Algorithms

Time complexity of Linear Search

Linear search involves an element-by-element lookup and comparison. In the worst-case scenario (i.e. the target element is the last element), every element will have to be compared. An n-element array will thus require *up to* n comparisons if sorted, and n comparisons if unsorted. **The worst-case time complexity of linear search is thus O(n)**.

Time complexity of Hash Table Search

The Hash Table algorithm involves the following steps:

1. Hashing the key to get the element index
2. Retrieving the element from its hashed index

Regardless of the number of elements in the hash table, the two steps are always the same. Thus, **the (theoretical) time complexity of hash table search is O(1)**.

Practically speaking, this is true only if the entire hash table can be stored in random-access memory (RAM). If the hash table is so large that it cannot be stored entirely in RAM, parts of it have to be stored on a physical disk (hard disk or solid state drive), and retrieval of elements from disk will be significantly slower than retrieval from physical memory.

Hash table collisions also worsen the time complexity of hash table search. Disambiguation through the chaining method involves iterating through a list of values for each hashed index: an O(n) operation. Rehashing the entire table is also an O(n) operation. Thus, it is important to pick a hashing function that minimises hash table collisions as much as possible.

Time complexity of Binary Search

The binary search algorithm is iterative, with each iteration involving:

1. Checking if the `start` and `end` indexes are equal.
2. Comparing the `target` with the middle element,
3. Updating the `start` or `end` index,
4. Recalculating the `mid` index

We will ignore steps 1 and 3 as they are relatively fast and do not significantly affect the execution time of the algorithm.

In essence, each iteration splits an n-element search space (between `start` to `end` indexes) into an $\frac{n}{2}$-element search space, until `start` and `end` are equal (a 1-element search space).

Iteration	Elements in search space
0	n
1	$n/2$
2	$n/4$
3	$n/8$
⋮	⋮
$\log_2 n$	1

Thus, a search through an n-element array involves $log_2 n$ operations. **The time complexity of binary search is O(log n)**.