StackAbuse

Search Algorithms in Python

Guest Contributor

Introduction

Searching for data stored in different data structures is a crucial part of pretty much every single application.

There are many different algorithms available to utilize when searching, and each have different implementations and rely on different data structures to get the job done.

Being able to choose a specific algorithm for a given task is a key skill for developers and can mean the difference between a fast, reliable and stable application and an application that crumbles from a simple request.

Membership Operators

Algorithms develop and become optimized over time as a result of constant evolution and the need to find the most efficient solutions for underlying problems in different domains.

One of the most common problems in the domain of Computer Science is searching through a collection and determining whether a given object is present in the collection or not.

Almost every programming language has its own implementation of a basic search algorithm, usually as a function which returns a `Boolean` value of `True` or `False` when an item is found in a given collection of items.

In Python, the easiest way to search for an object is to use Membership Operators - named that way because they allow us to determine whether a given object is a member in a collection.

These operators can be used with any iterable data structure in Python, including Strings, Lists, and Tuples.

- `in` - Returns `True` if the given element is a part of the structure.
- `not in` - Returns `True` if the given element is not a part of the structure.

```
>>> 'apple' in ['orange', 'apple', 'grape']
True
>>> 't' in 'stackabuse'
True
>>> 'q' in 'stackabuse'
False
>>> 'q' not in 'stackabuse'
True
```

Membership operators suffice when all we need to do is find whether a substring exists within a given string, or determine whether two Strings, Lists, or Tuples intersect in terms of the objects they hold.

In most cases we need the position of the item in the sequence, in addition to determining whether or not it exists; membership operators do not meet this requirement.

There are many search algorithms that don't depend on built-in operators and can be used to search for values faster and/or more efficiently. In addition, they can yield more information, such as the position of the element in the collection, rather than just being able to determine its existence.

Linear Search

Linear search is one of the simplest searching algorithms, and the easiest to understand. We can think of it as a ramped-up version of our own implementation of Python's `in` operator.

The algorithm consists of iterating over an array and returning the index of the first occurrence of an item once it is found:

```
def LinearSearch(lys, element):
    for i in range (len(lys)):
        if lys[i] == element:
            return i
    return -1
```

So if we use the function to compute:

```
>>> print(LinearSearch([1,2,3,4,5,2,1], 2))
```

Upon executing the code, we're greeted with:

```
1
```

This is the index of the first occurrence of the item we are searching for - keeping in mind that Python indexes are 0-based.

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 `lys`.

Linear search is not often used in practice, because the same efficiency can be achieved by using inbuilt methods or existing operators, and it is not as fast or efficient as other search algorithms.

Linear search is a good fit for when we need to find the first occurrence of an item in an unsorted collection because unlike most other search algorithms, it does not require that a collection be sorted before searching begins.

Binary Search

Binary search follows a divide and conquer methodology. It is faster than linear search but requires that the array be sorted before the algorithm is executed.

Assuming that we're searching for a value `val` in a sorted array, the algorithm compares `val` to the value of the middle element of the array, which we'll call `mid`.

- If `mid` is the element we are looking for (best case), we return its index.

- If not, we identify which side of `mid` `val` is more likely to be on based on whether `val` is smaller or greater than `mid`, and discard the other side of the array.
- We then recursively or iteratively follow the same steps, choosing a new value for `mid`, comparing it with `val` and discarding half of the possible matches in each iteration of the algorithm.

The binary search algorithm can be written either recursively or iteratively. Recursion is generally slower in Python because it requires the allocation of new stack frames.

Since a good search algorithm should be as fast and accurate as possible, let's consider the iterative implementation of binary search:

```
def BinarySearch(lys, val):
    first = 0
    last = len(lys)-1
    index = -1
    while (first <= last) and (index == -1):
        mid = (first+last)//2
        if lys[mid] == val:
            index = mid
        else:
            if val<lys[mid]:
                last = mid -1
            else:
                first = mid +1
    return index
```

If we use the function to compute:

```
>>> BinarySearch([10,20,30,40,50], 20)
```

We get the result:

```
1
```

Which is the index of the value that we are searching for.

The action that the algorithm performs next in each iteration is one of several possibilities:

- Returning the index of the current element
- Searching through the left half of the array
- Searching through the right half of the array

We can only pick one possibility per iteration, and our pool of possible matches gets divided by two in each iteration. This makes the time complexity of binary search *O(log n)*.

One drawback of binary search is that if there are multiple occurrences of an element in the array, it does not return the index of the first element, but rather the index of the element closest to the middle:

```
>>> print(BinarySearch([4,4,4,4,4], 4))
```

Running this piece of code will result in the index of the middle element:

```
1
```

For comparison performing a linear search on the same array would return:

```
0
```

Which is the index of the *first* element. However, we cannot categorically say that binary search does not work if an array contains the same element twice - it can work just like linear search and return the first occurrence of the element in some cases.

If we perform binary search on the array `[1,2,3,4,4,5]` for instance, and search for 4, we would get `3` as the result.

Binary search is quite commonly used in practice because it is efficient and fast when compared to linear search. However, it does have some shortcomings, such as its reliance on the `//` operator. There are many other *divide and conquer* search algorithms that are derived from binary search, let's examine a few of those next.

Jump Search

Jump Search is similar to binary search in that it works on a sorted array, and uses a similar *divide and conquer* approach to search through it.

It can be classified as an improvement of the linear search algorithm since it depends on linear search to perform the actual comparison when searching for a value.

Given a sorted array, instead of searching through the array elements incrementally, we search in *jumps*. So in our input list `lys`, if we have a jump size of *jump* our algorithm will consider elements in the order `lys[0]`, `lys[0+jump]`, `lys[0+2jump]`, `lys[0+3jump]` and so on.

With each jump, we store the previous value we looked at and its index. When we find a set of values where `lys[i]` <element< `lys[i+jump]`, we perform a linear search with `lys[i]` as the left-most element and `lys[i+jump]` as the right-most element in our search set:

```
import math

def JumpSearch (lys, val):
    length = len(lys)
    jump = int(math.sqrt(length))
    left, right = 0, 0
    while left < length and lys[left] <= val:
        right = min(length - 1, left + jump)
        if lys[left] <= val and lys[right] >= val:
            break
        left += jump;
    if left >= length or lys[left] > val:
        return -1
    right = min(length - 1, right)
    i = left
    while i <= right and lys[i] <= val:
        if lys[i] == val:
            return i
        i += 1
    return -1
```

Since this is a complex algorithm, let's consider the step-by-step computation of jump search with this input:

```
>>> print(JumpSearch([1,2,3,4,5,6,7,8,9], 5))
```

- Jump search would first determine the jump size by computing `math.sqrt(len(lys))`. Since we have 9 elements, the jump size would be √9 = 3.
- Next, we compute the value of the `right` variable, which is the minimum of the length of the array minus 1, or the value of `left+jump`, which in our case would be 0+3= 3. Since 3 is smaller than 8 we use 3 as the value of `right`.
- Now we check whether our search element, 5, is between `lys[0]` and `lys[3]`. Since 5 is not between 1 and 4, we move on.

- Next, we do the calculations again and check whether our search element is between `lys[3]` and `lys[6]`, where 6 is 3+jump. Since 5 is between 4 and 7, we do a linear search on the elements between `lys[3]` and `lys[6]` and return the index of our element as:

```
4
```

The time complexity of jump search is *O(√n)*, where *√n* is the jump size, and *n* is the length of the list, placing jump search between the linear search and binary search algorithms in terms of efficiency.

The single most important advantage of jump search when compared to binary search is that it does not rely on the division operator (`/`).

In most CPUs, using the division operator is costly when compared to other basic arithmetic operations (addition, subtraction, and multiplication), because the implementation of the division algorithm is iterative.

The cost by itself is very small, but when the number of elements to search through is very large, and the number of division operations that we need to perform increases, the cost can add up incrementally. Therefore jump search is better than binary search when there is a large number of elements in a system where even a small increase in speed matters.

To make jump search faster, we could use binary search or another internal jump search to search through the blocks, instead of relying on the much slower linear search.

Fibonacci Search

Fibonacci search is another divide and conquer algorithm which bears similarities to both binary search and jump search. It gets its name because it uses Fibonacci numbers to calculate the block size or search range in each step.

Fibonacci numbers start with zero and follow the pattern *0, 1, 1, 2, 3, 5, 8, 13, 21...* where each element is the addition of the two numbers that immediately precede it.

The algorithm works with three Fibonacci numbers at a time. Let's call the three numbers `fibM`, `fibM_minus_1`, and `fibM_minus_2` where `fibM_minus_1` and `fibM_minus_2` are the two numbers immediately before `fibM` in the sequence:

```
fibM = fibM_minus_1 + fibM_minus_2
```

We initialize the values to 0,1, and 1 or the first three numbers in the Fibonacci sequence to avoid getting an index error in the case where our search array `lys` contains a very small number of items.

Then we choose the smallest number of the Fibonacci sequence that is greater than or equal to the number of elements in our search array `lys`, as the value of `fibM`, and the two Fibonacci numbers immediately before it as the values of `fibM_minus_1` and `fibM_minus_2`. While the array has elements remaining and the value of `fibM` is greater than one, we:

- Compare `val` with the value of the block in the range up to `fibM_minus_2`, and return the index of the element if it matches.
- If the value is greater than the element we are currently looking at, we move the values of `fibM`, `fibM_minus_1` and `fibM_minus_2` two steps down in the Fibonacci sequence, and reset the index to the index of the element.
- If the value is less than the element we are currently looking at, we move the values of `fibM`, `fibM_minus_1` and `fibM_minus_2` one step down in the Fibonacci sequence.

Let's take a look at the Python implementation of this algorithm:

```python
def FibonacciSearch(lys, val):
    fibM_minus_2 = 0
    fibM_minus_1 = 1
    fibM = fibM_minus_1 + fibM_minus_2
    while (fibM < len(lys)):
        fibM_minus_2 = fibM_minus_1
        fibM_minus_1 = fibM
        fibM = fibM_minus_1 + fibM_minus_2
    index = -1;
    while (fibM > 1):
        i = min(index + fibM_minus_2, (len(lys)-1))
        if (lys[i] < val):
            fibM = fibM_minus_1
            fibM_minus_1 = fibM_minus_2
            fibM_minus_2 = fibM - fibM_minus_1
            index = i
        elif (lys[i] > val):
            fibM = fibM_minus_2
            fibM_minus_1 = fibM_minus_1 - fibM_minus_2
            fibM_minus_2 = fibM - fibM_minus_1
        else :
            return i
    if(fibM_minus_1 and index < (len(lys)-1) and lys[index+1] == val):
        return index+1;
    return -1
```

If we use the FibonacciSearch function to compute:

```
>>> print(FibonacciSearch([1,2,3,4,5,6,7,8,9,10,11], 6))
```

Let's take a look at the step-by-step process of this search:

- Determining the smallest Fibonacci number greater than or equal to the length of the list as `fibM` ; in this case, the smallest Fibonacci number meeting our requirements is 13.
- The values would be assigned as:
    - fibM = 13
    - fibM_minus_1 = 8
    - fibM_minus_2 = 5
    - index = -1
- Next, we check the element `lys[4]` where 4 is the minimum of -1+5 . Since the value of `lys[4]` is 5, which is smaller than the value we are searching for, we move the Fibonacci numbers *one* step down in the sequence, making the values:
    - fibM = 8
    - fibM_minus_1 = 5
    - fibM_minus_2 = 3
    - index = 4
- Next, we check the element `lys[7]` where 7 is the minimum of 4+3. Since the value of `lys[7]` is 8, which is greater than the value we are searching for, we move the Fibonacci numbers *two* steps down in the sequence.

    - fibM = 3
    - fibM_minus_1 = 2
    - fibM_minus_2 = 1
    - index = 4
- Now we check the element `lys[5]` where 5 is the minimum of 4+1 . The value of `lys[5]` is 6, which *is* the value we are searching for!

The result, as expected is:

```
5
```

The time complexity for Fibonacci search is *O(log n)*; the same as binary search. This means the algorithm is faster than both linear search and jump search in most cases.

Fibonacci search can be used when we have a very large number of elements to search through, and we want to reduce the inefficiency associated with using an algorithm which relies on the division operator.

An additional advantage of using Fibonacci search is that it can accommodate input arrays that are too large to be held in CPU cache or RAM, because it searches through elements in increasing step sizes, and not in a fixed size.

Exponential Search

Exponential search is another search algorithm that can be implemented quite simply in Python, compared to jump search and Fibonacci search which are both a bit complex. It is also known by the names *galloping search*, *doubling search* and *Struzik search*.

Exponential search depends on binary search to perform the final comparison of values. The algorithm works by:

- Determining the range where the element we're looking for is likely to be
- Using binary search for the range to find the exact index of the item

The Python implementation of the exponential search algorithm is:

```
def ExponentialSearch(lys, val):
    if lys[0] == val:
        return 0
    index = 1
    while index < len(lys) and lys[index] <= val:
        index = index * 2
    return BinarySearch( arr[:min(index, len(lys))], val)
```

If we use the function to find the value of:

```
>>> print(ExponentialSearch([1,2,3,4,5,6,7,8],3))
```

The algorithm works by:

- Checking whether the first element in the list matches the value we are searching for - since `lys[0]` is 1 and we are searching for 3, we set the index to 1 and move on.
- Going through all the elements in the list, and while the item at the index'th position is less than or equal to our value, exponentially increasing the value of `index` in multiples of two:
    - index = 1, `lys[1]` is 2, which is less than 3, so the index is multiplied by 2 and set to 2.
    - index = 2, `lys[2]` is 3, which is equal to 3, so the index is multiplied by 2 and set to 4.
    - index = 4, `lys[4]` is 5, which is greater than 3; the loop is broken at this point.
- It then performs a binary search by slicing the list; `arr[:4]` . In Python, this means that the sub list will contain all elements up to the 4th element, so we're actually calling:

```
>>> BinarySearch([1,2,3,4], 3)
```

which would return:

```
2
```

Which is the index of the element we are searching for in both the original list, and the sliced list that we pass on to the binary search algorithm.

Exponential search runs in *O(log i)* time, where *i* is the index of the item we are searching for. In its worst case, the time complexity is *O(log n)*, when the last item is the item we are searching for (*n* being the length of the array).

Exponential search works better than binary search when the element we are searching for is closer to the beginning of the array. In practice, we use exponential search because it is one of the most efficient search algorithms for unbounded or infinite arrays.

Interpolation Search

Interpolation search is another divide and conquer algorithm, similar to binary search. Unlike binary search, it does not always begin searching at the middle. Interpolation search calculates the probable position of the element we are searching for using the formula:

```
index = low + [(val-lys[low])*(high-low) / (lys[high]-lys[low])]
```

Where the variables are:

- lys - our input array
- val - the element we are searching for
- index - the probable index of the search element. This is computed to be a higher value when val is closer in value to the element at the end of the array (`lys[high]`), and lower when val is closer in value to the element at the start of the array (`lys[low]`)
- low - the starting index of the array
- high - the last index of the array

The algorithm searches by calculating the value of `index` :

- If a match is found (when `lys[index] == val`), the index is returned
- If the value of `val` is less than `lys[index]` , the value for the index is re-calculated using the formula for the left sub-array
- If the value of `val` is greater than `lys[index]` , the value for the index is re-calculated using the formula for the right sub-array

Let's go ahead and implement the Interpolation search using Python:

```
def InterpolationSearch(lys, val):
    low = 0
    high = (len(lys) - 1)
    while low <= high and val >= lys[low] and val <= lys[high]:
        index = low + int(((float(high - low) / ( lys[high] - lys[low])) * (
        if lys[index] == val:
            return index
        if lys[index] < val:
            low = index + 1;
        else:
            high = index - 1;
    return -1
```

If we use the function to compute:

```
>>> print(InterpolationSearch([1,2,3,4,5,6,7,8], 6))
```

Our initial values would be:

- val = 6,
- low = 0,
- high = 7,
- lys[low] = 1,
- lys[high] = 8,
- index = 0 + [(6-1)*(7-0)/(8-1)] = 5

Since `lys[5]` is 6, which is the value we are searching for, we stop executing and return the result:

```
5
```

If we have a large number of elements, and our index cannot be computed in one iteration, we keep on re-calculating values for *index* after adjusting the values of *high* and *low* in our formula.

The time complexity of interpolation search is *O(log log n)* when values are uniformly distributed. If values are not uniformly distributed, the worst-case time complexity is *O(n)*, the same as linear search.

Interpolation search works best on uniformly distributed, sorted arrays. Whereas binary search starts in the middle and always divides into two, interpolation search calculates the likely position of the element and checks the index, making it more likely to find the element in a smaller number of iterations.

Why Use Python For Searching?

Python is highly readable and efficient when compared to older programming languages like Java, Fortran, C, C++ etc. One key advantage of using Python for implementing search algorithms is that you don't have to worry about casting or explicit typing.

In Python, most of the search algorithms we discussed will work just as well if we're searching for a String. Keep in mind that we do have to make changes to the code for algorithms which use the search element for numeric calculations, like the interpolation search algorithm.

Python is also a good place to start if you want to compare the performance of different search algorithms for your dataset; building a prototype in Python is easier and faster because you can do more with fewer lines of code.

To compare the performance of our implemented search algorithms against a dataset, we can use the time library in Python:

```
import time

start = time.time()
# call the function here
end = time.time()
print(start-end)
```

Conclusion

There are many possible ways to search for an element within a collection. In this article, we attempted to discuss a few search algorithms and their implementations in Python.

Choosing which algorithm to use is based on the data you have to search through; your input array, which we've called `lys` in all our implementations.

- If you want to search through an unsorted array or to find the *first* occurrence of a search variable, the best option is linear search.
- If you want to search through a sorted array, there are many options of which the simplest and fastest method is binary search.
- If you have a sorted array that you want to search through without using the division operator, you can use either jump search or Fibonacci search.
- If you know that the element you're searching for is likely to be closer to the start of the array, you can use exponential search.
- If your sorted array is also uniformly distributed, the fastest and most efficient search algorithm to use would be interpolation search.

If you're not sure which algorithm to use with a sorted array, just try each of them out along with Python's time library and pick the one that performs best with your dataset.

#python #algorithms

Last Updated: September 26th, 2019

Was this article helpful? ☆☆☆☆☆

You might also like...

- Guide to the K-Nearest Neighbors Algorithm in Python and Scikit-Learn
- Quicksort in Python
- Big O Notation and Algorithm Analysis with Python Examples
- Bucket Sort in Python
- Insertion Sort in Python

Guest Contributor *Author*

Course

Graphs in Python - Theory and Implementation

#python #data structures #algorithms #computer science

Graphs are an extremely versatile data structure. More so than most people realize! Graphs can be used to model practically anything, given their nature of...

Details →

Course

Data Visualization in Python with Matplotlib and Pandas

#python #pandas #matplotlib

Data Visualization in Python with Matplotlib and Pandas is a course designed to take absolute beginners to Pandas and Matplotlib, with basic Python knowledge, and...

David Landup

Details →