10

Searching

An important operation for all data structures is searching for elements from a collection of data. There are various methods to search for an element in data structures; in this chapter, we shall explore the different strategies that can be used to find elements in a collection of items.

Data elements can be stored in any kind of data structure, such as an array, link list, tree, or graph; the search operation is very important for many applications, mostly whenever we want to know if a particular data element is present in an existing list of data items. In order to retrieve the information efficiently, we require an efficient search algorithm.

In this chapter, we will learn about the following:

- Various search algorithms
- Linear search algorithm
- Jump search algorithm
- Binary search algorithm
- Interpolation search algorithm
- Exponential search algorithm

Let us start with an introduction to searching and a definition and then look at the linear search algorithm.

Introduction to searching

A search operation is carried out to find the location of the desired data item from a collection of data items. The search algorithm returns the location of the searched value where it is present in the list of items and if the data item is not present, it returns `None`.

Efficient searching is important to efficiently retrieve the location of the desired data item from a list of stored data items. For example, we have a long list of data values, such as `{1, 45, 65, 23, 65, 75, 23}`, and we want to see if `75` is present in the list or not. It becomes important to have an efficient search algorithm when the list of data items becomes large.

There are two different ways in which data can be organized, which can affect how a search algorithm works:

- First, the search algorithm is applied to a list of items that is already sorted; that is, it is applied to an ordered set of items. For example, `[1, 3, 5, 7, 9, 11, 13, 15, 17]`.
- The search algorithm is applied to an unordered set of items, which is not sorted. For example, `[11, 3, 45, 76, 99, 11, 13, 35, 37]`.

We will first take a look at linear searching.

Linear search

The search operation is used to find out the index position of a given data item in a list of data items. If the searched item is available in the given list of data items, then the search algorithm returns the index position where it is located; otherwise, it returns that the item is not found. Here, the index position is the location of the desired item in the given list.

The simplest approach to search for an item in a list is to search linearly, in which we look for items one by one in the whole list. Let's take an example of six list items `{60, 1, 88, 10, 11, 100}` to understand the linear search algorithm, as shown in *Figure 10.1*:

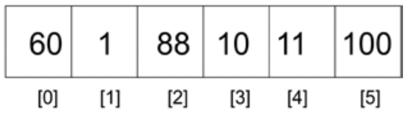

Figure 10.1: An example of linear search

The preceding list has elements that can be accessed through the index. To find an element in the list, we can search for the given element linearly one by one. This technique traverses the list of elements by using the index to move from the beginning of the list to the end. Each element is checked, and if it does not match the search item, the next item is examined. By hopping from one item to the next, the list is traversed sequentially. We use list items with integer values in this chapter to help you understand the concept, since integers can be compared easily; however, a list item can hold any other data type as well.

The linear search approach depends on how the list items are stored in memory—whether they are already sorted in order or they are not sorted. Let's first see how the linear search algorithm works if the given list of items is not sorted.

Unordered linear search

The unordered linear search is a linear search algorithm in which the given list of date items is not sorted. We linearly match the desired data item with the data items of the list one by one till the end of the list or until the desired data item is found. Consider an example list that contains the elements `60`, `1`, `88`, `10`, and `100`—an unordered list. To perform a `search` operation on such a list, one proceeds with the first item and compares that with the search item. If the search item is not matched, then the next element in the list is checked. This continues till we reach the last element in the list or until a match is found.

In an unordered list of items, the search for the term `10` starts from the first element and moves to the next element in the list. Thus, firstly `60` is compared with `10`, and since it is not equal, we compare `66` with the next element `1`, then `88`, and so on till we find the search term in the list. Once the item is found, we return the index position of where we have found the desired item. This process is shown in *Figure 10.2*:

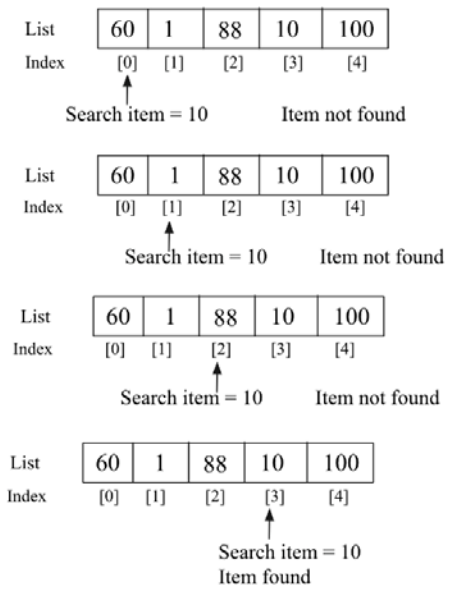

Figure 10.2: Unordered linear search

Here is the implementation in Python for the linear search on an unordered list of items:

```
def search(unordered_list, term):
    for i, item in enumerate(unordered_list):
        if term == unordered_list[i]:
            return i
    return None
```

The `search` function takes two parameters; the first is the list that holds the data, and the second parameter is the item that we are looking for, called the **search term**. On every pass of the `for` loop, we check if the search term is equal to the indexed item. If this is true, then there is a match, and there is no need to proceed further with the search. We return the index position where the searched item is found in the list. If the loops run to the end of the list with no match found, then `None` is returned to signify that there is no such item in the list.

We can use the following code snippet to check if a desired data element is present in the given list of data items:

```
list1 = [60, 1, 88, 10, 11, 600]

search_term = 10
index_position = search(list1, search_term)
print(index_position)

list2 = ['packt', 'publish', 'data']
search_term2 = 'data'
Index_position2 = search(list2, search_term2)
print(Index_position2)
```

The output of the above code is as follows:

```
3
2
```

In the output of the above code, firstly, the index position `3` is returned when we search for data element `10` in `list1`. And secondly, index position `2` is returned when data item `'data'` is searched for in `list2`. We can use the same algorithm for searching a non-numeric data item from a list of non-numeric data items in Python, since string elements can also be compared similarly to numeric data in Python.

When searching for any element from an unordered list of items, in the worst case the desired item may be in the last position or may not be present in the list. In this situation we will have to compare the search item with all the elements of the list, i.e. `n` times if the total number of data items in the list is `n`. Thus, the unordered linear search has a worst-case running time of `O(n)`. All the elements may need to be visited before finding the search term. The worst-case scenario will be when the search term is located at the last position of the list.

Next, we discuss how the linear search algorithm works if the given list of data items is already sorted.

Ordered linear search

If the data elements are already arranged in a sorted order, then the linear search algorithm can be improved. The linear search algorithm in a sorted list of elements has the following steps:

1. Move through the list sequentially
2. If the value of a search item is greater than the object or item currently under inspection in the loop, then quit and return `None`

In the process of iterating through the list, if the value of the search term is less than the current item in the list, then there is no need to continue with the search. Let's consider an example to see how this works. Let's say we have a list of items `{2, 3, 4, 6, 7}` as shown in *Figure 10.3*, and we want to search for term `5`:

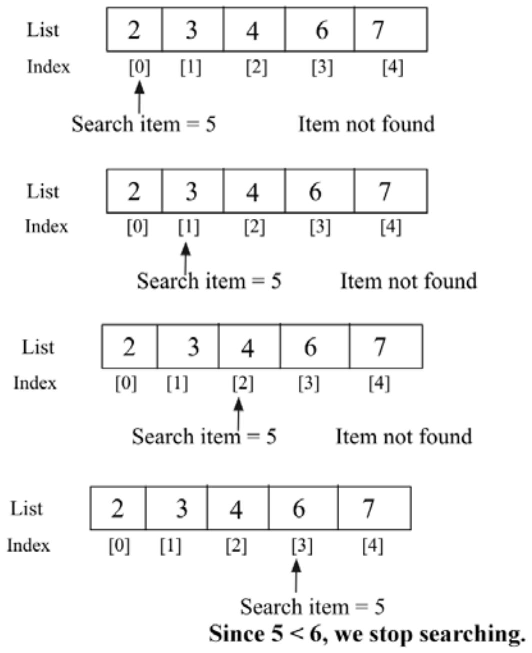

Figure 10.3: Example of ordered linear search

We start the `search` operation by comparing the desired search element `5` with the first element; no match is found. We continue on to compare the search element with the next element, i.e. `3`, in the list. Since it also does not match, we move on to examine the next element, i.e. `4`, and since it also does not match, we continue searching in the list, and we compare the search element with the fourth element, i.e. `6`. This also does not match the search term. Since the given list is already sorted in ascending order and the value of the search item is less than the fourth element, the search item cannot be found in any later position in the list. In other words, if the current item in the list is greater than the search term, then it means there is no need to further search the list, and we stop searching for the element in the list.

Here is the implementation of the linear search when the list is already sorted:

```
def search_ordered(ordered_list, term):
     ordered_list_size = len(ordered_list)
     for i in range(ordered_list_size):
          if term == ordered_list[i]:
              return i
          elif ordered_list[i] > term:
              return None
     return None
```

In the preceding code, the `if` statement now caters to checking if the search item is found in the list or not. Then, `elif` tests the condition where `ordered_list[i] > term`. We stop searching if the comparison evaluates to `True`, which means the current item in the list is greater than the search element. The last line in the method returns `None` because the loop may go through the list and still the search item is not matched in the list.

We use the following code snippet to use the search algorithm:

```
list1 = [2, 3, 4, 6, 7]

search_term = 5
index_position1 = search_ordered(list1, search_term)

if index_position1 is None:
    print("{} not found".format(search_term))
else:
    print("{} found at position {}".format(search_term, index_position1))


list2 = ['book','data','packt', 'structure']

search_term2 = 'structure'
index_position2 = search_ordered(list2, search_term2)

if index_position2 is None:
    print("{} not found".format(search_term2))
else:
    print("{} found at position {}".format(search_term2, index_position2))
```

The output of the above code is as follows:

```
5 not found
structure found at position 3
```

In the output of the above code, firstly, the search item `5` is not matched in the given list. And for the second list of non-numeric data elements, the string structure is matched at index position `3`. Hence, we can use the same linear search algorithm for searching a non-numeric data item from an ordered list of data items, so the given list of data items should be sorted similarly to a contact list on a phone.

In the worst-case scenario, the desired search item will be present in the last position of the list or will not be present at all. In this situation, we will have to trace the complete list (say `n` elements). Thus, the worst-case time complexity of an ordered linear search is `O(n)`.

Next, we will discuss the jump search algorithm.

Jump search

The **jump search** algorithm is an improvement over linear search for searching for a given element from an ordered (or sorted) list of elements. This uses the divide-and-conquer strategy in order to search for the required element. In linear search, we compare the search value with each element of the list, whereas in jump search, we compare the search value at different intervals in the list, which reduces the number of comparisons.

In this algorithm, firstly, we divide the sorted list of data into subsets of data elements called blocks. Within each block, the highest value will lie within the last element, as the array is sorted. Next, in this algorithm, we start comparing the search value with the last element of each block. There can be three conditions:

1. If the search value is less than the last element of the block, we compare it with the next block.
2. If the search value is greater than the last element of the block, it means the desired search value must be present in the current block. So, we apply linear search in this block and return the index position.
3. If the search value is the same as the compared element of the block, we return the index position of the element and we return the candidate.

Generally, the size of the block is taken as $\sqrt{n}$, since it gives the best performance for a given array of length `n`.

In the worst-case situation, we will have to make *n/m* number of jumps (here, `n` is the total number of elements, and *m* is the block size) if the last element of the last block is greater than the item to be searched, and we will need *m* - 1 comparisons for linear search in the last block. Therefore, the total number of comparisons will be ((*n/m*) + *m* - 1), which will minimize when *m* = √*n* . So the size of the block is taken as √*n* since it gives the best performance.

Let's take an example list `{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11}` to search for a given element (say `10`):

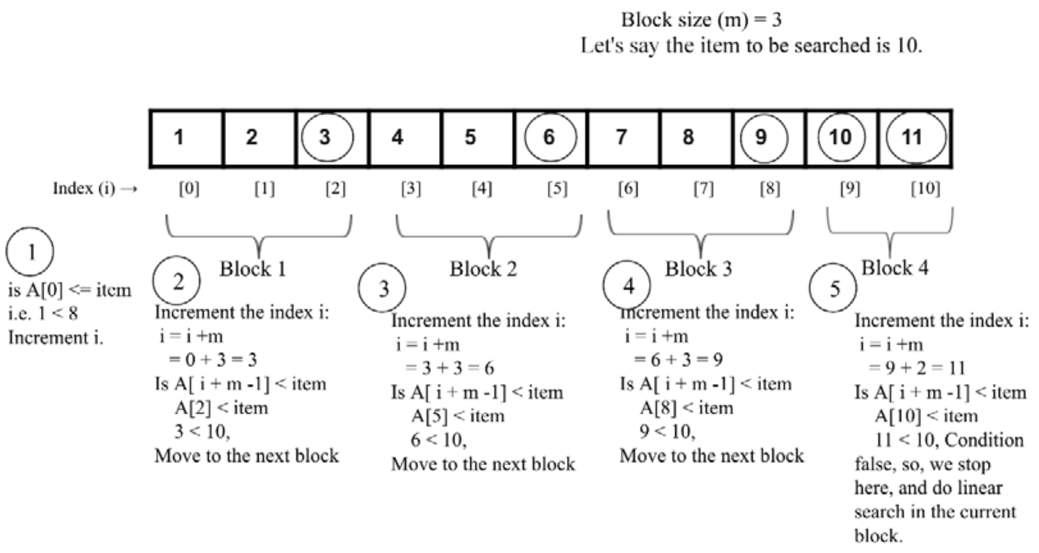

Figure 10.4: Illustration of the jump search algorithm

In the above example, we find the desired element `10` in `5` comparisons. Firstly, we compare the first value of the array with the desired item `A[0] <= item`; if it is true, then we increase the index by the block size (this is shown in *step 1* in *Figure 10.4*). Next, we compare the desired item with the last element of each block. If it is greater, then we move to the next block, such as from block 1 to block 3 (this is shown in *steps 2*, *3*, and *4* in *Figure 10.4*).

Further, when the desired search element becomes smaller than the last element of a block, we stop incrementing the index position and then we do the linear search in the current block. Now, let us discuss the implementation of the jump searching algorithms. Firstly, we implement the linear search algorithm, which is similar to what we discussed in the previous section.

It is given again here for the sake of completeness as follows:

```
def search_ordered(ordered_list, term):
    print("Entering Linear Search")
    ordered_list_size = len(ordered_list)
    for i in range(ordered_list_size):
        if term == ordered_list[i]:
            return i
        elif ordered_list[i] > term:
            return -1
    return -1
```

In the above code, given an ordered list of elements, it returns the index of the location where a given data element is found in the list. It returns `-1` if the desired element is not found in the list. Next, we implement the `jump_search()` method as follows:

```
def jump_search(ordered_list, item):
    import math
    print("Entering Jump Search")
    list_size = len(ordered_list)
    block_size = int(math.sqrt(list_size))
    i = 0
    while i != len(ordered_list)-1 and ordered_list[i] <= item:
        print("Block under consideration - {}".format(ordered_list[i:
i+block_size]))
        if i+ block_size > len(ordered_list):
            block_size =  len(ordered_list) - i
            block_list = ordered_list[i: i+block_size]
            j = search_ordered(block_list, item)
            if j == -1:
                print("Element not found")
                return
            return i + j
        if ordered_list[i + block_size -1] == item:
            return i+block_size-1
        elif ordered_list[i + block_size - 1] > item:
            block_array = ordered_list[i: i + block_size - 1]
            j = search_ordered(block_array, item)
            if j == -1:
                print("Element not found")
                return
            return i + j
        i += block_size
```

In the above code, firstly we assign the length of the list to the variable `n`, and then we compute the block size as $\sqrt{n}$. Next, we start with the first element, index `0`, and then continue searching until we reach the end of the list.

We start with the starting index `i = 0` with a block of size m, and we continue incrementing until the window reaches the end of the list. We compare whether `ordered_list [I + block_size -1] == item`. If they match, it returns the index position `(i+ block_size -1)`. The code snippet for this is as follows:

```
        if ordered_list[i+ block_size -1] == item:
            return i+ block_size -1
```

If `ordered_list [i+ block_size -1] > item`, we proceed to carry out the linear search algorithm inside the current block `block_array = ordered_list [i : i+ block_size-1]`, as follows:

```
        elif ordered_list[i+ block_size -1] > item:
            block_array = ordered_list[i: i+ block_size -1]
            j = search_ordered(block_array, item)
            if j == -1:
                print("Element not found")
                return
            return i + j
```

In the above code, we use the linear search algorithm in the subarray. It returns `-1` if the desired element is not found in the list; otherwise, the index position of `(i + j)` is returned. Here, `i` is the index position until the previous block where we may find the desired element and `j` is the position of the data element within the block where the desired element is matched. This process is also depicted in *Figure 10.5*.

In this figure, we can see that `i` is in index position 5, and then `j` is the number of elements within the final block where we find the desired element, i.e. 2, so the final returned index will be `5 + 2 = 7`:

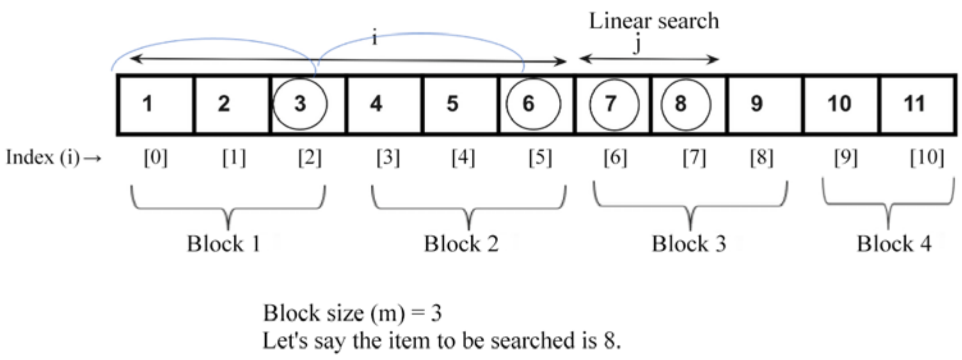

Figure 10.5: Demonstration of index position i and j for the search value 8

Further, we need to check for the length of the last block since it may have a number of elements less than the block size. For example, if the total number of elements is 11, then in the last block we will have 2 elements. So, we check if the desired search element is present in the last block, and if so we should update the starting and ending index as follows:

```
        if i+ block_size > len(ordered_list):
            block_size =  len(ordered_list) - i
            block_list = ordered_list[i: i+block_size]
            j = search_ordered(block_list, item)
            if j == -1:
                print("Element not found")
                return
            return i + j
```

In the above code, we search for the desired element using the linear search algorithm.

Finally, if `ordered_list[i+m-1] < item`, then we move to the next iteration, and update the index by adding the block size to the index as `i += block_size`.

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

The output of the above code snippet is:

```
Entering Jump Search
Block under consideration - [1, 2, 3]
Block under consideration - [4, 5, 6]
Block under consideration - [7, 8, 9]
Entering Linear Search
7
```

In the above output, we can see the steps for how we searched for element `10` in the given list of elements.

Thus, jump search performs linear search on a block, so first it finds the block in which the element is present and then applies linear search within that block. The size of the block depends on the size of the array. If the size of the array is `n`, then the block size may be $\sqrt{n}$. If it does not find the element in that block, it moves to the next block. The jump search first finds out in which block the desired element may be present. For a list of `n` elements, and a block size of *m*, the total number of jumps possible will be *n/m* jumps. Let's say the size of the block is $\sqrt{n}$; thus, the worst-case time complexity will be $O(\sqrt{n})$.

Next, we will discuss the binary search algorithm.

Binary search

The **binary search** algorithm finds a given item from the given sorted list of items. It is a fast and efficient algorithm to search for an element; however, one drawback of this algorithm is that we need a sorted list. The worst-case running time complexity of a binary search algorithm is `O(logn)` whereas for linear search it is `O(n)`.

The binary search algorithm works as follows. It starts searching for the item by dividing the given list in half. If the search item is smaller than the middle value then it will look for the searched item only in the first half of the list, and if the search item is greater than the middle value it will only look at the second half of the list. We repeat the same process every time until we find the search item, or we have checked the whole list. In the case of a non-numeric list of data items, for example, if we have string data items, then we should sort the data items in alphabetical order (similar to how a contact list is stored on a phone).

Let's understand the binary search algorithm with an example. Suppose we have a book with 1,000 pages, and we want to reach page number 250. We know that every book has its pages numbered sequentially from `1` upward. So, according to the binary search analogy, we first check forsearch item 250, which is less than the midpoint, which is 500. Thus, we search for the required page only in the first half of the book.

We again find the midpoint of the first half of the book, using page 500 as a reference we find the midpoint, 250. That brings us closer to finding the 250th page. Then we find the required page in the book.

Let's take another example to understand the workings of binary search. We want to search for item 43 from a list of 12 items, as shown in *Figure 10.6*:

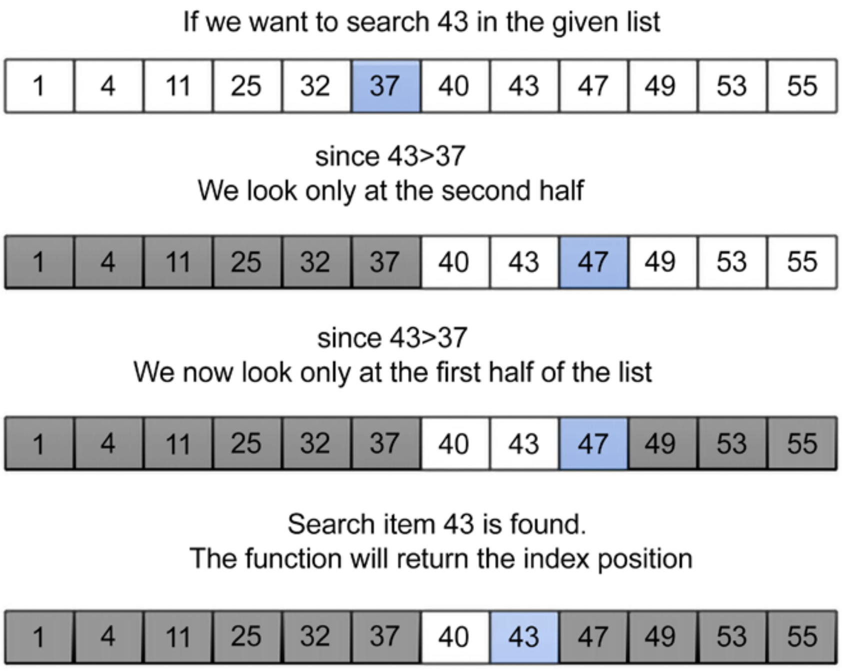

Figure 10.6: Working of binary search

We start searching for the item by comparing it to the middle item of the list, which is 37 in the example. If the value of the search item is less than the middle value, we only look at the first half of the list; otherwise, we will look at the other half. So, we only need to search for the item in the second half. We follow the same procedure until we find search item 43 in the list. This process is shown in the *Figure 10.6*.

The following is an implementation of the binary search algorithm on an ordered list of items:

```
def binary_search_iterative(ordered_list, term):
    size_of_list = len(ordered_list) - 1
    index_of_first_element = 0
    index_of_last_element = size_of_list
    while index_of_first_element <= index_of_last_element:
        mid_point = (index_of_first_element + index_of_last_element)/2
        if ordered_list[mid_point] == term:
            return mid_point
        if term > ordered_list[mid_point]:
            index_of_first_element = mid_point + 1
        else:
            index_of_last_element = mid_point - 1
    if index_of_first_element > index_of_last_element:
        return None
```

We'll explain the above code using a list of sorted elements `{10, 30, 100, 120, 500}`. Now let's assume we have to find the position where item `10` is located in the list shown in *Figure 10.7*:

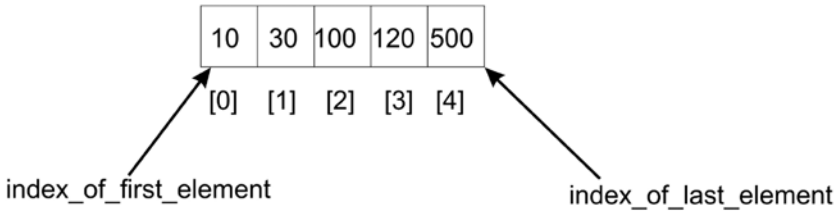

Figure 10.7: Sample list of five items

Firstly, we declare two variables, i.e. `index_of_first_element` and `index_of_last_element`, which denote the starting and ending index positions in the given list. Next, the algorithm uses a `while` loop to iteratively adjust the limits in the list within which we have to find a search item. The terminating condition to stop the `while` loop is that the difference between the starting index, `index_of_first_element`, and the `index_of_last_element` index should be positive.

The algorithm first finds the midpoint of the list by adding the index of the first element (i.e. `0` in this case) to the index of the last element (which is `4` in this example) and dividing it by `2`. We get the middle index, `mid_point`:

```
mid_point = (index_of_first_element + index_of_last_element)/2
```

In this case, the index of the midpoint is `2`, and the data item stored at this position is `100`. We compare the midpoint element with the search item `10`.

Since these do not match, and the search item `10` is less than the midpoint, the desired search item should lie in the first half of the list, thus, we adjust the index range of `index_of_first_element` to `mid_point-1`, which means the new search range becomes `0` to `1`, as shown in *Figure 10.8*:

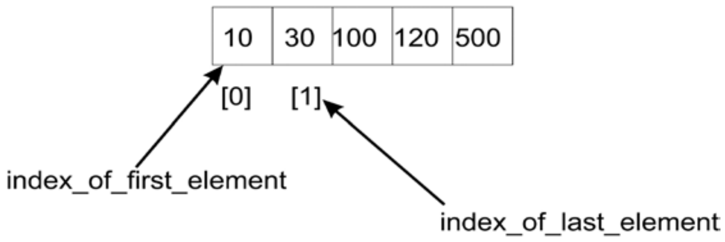

Figure 10.8: Index of first and last elements for the first half of the list

However, if we had been searching for `120`, as `120` would have been greater than the middle value (`100`), we would have searched for the item in the second half of the list, and as a result, we would have needed to change the list index range to be `mid_point +1` to `index_of_last_element`. In that case the new range would have been `(3, 4)`.

So, with the new indexes of the first and last elements, i.e. `index_of_first_element` and `index_of_last_element`, now being `0` and `1` respectively, we compute the midpoint `(0 + 1)/2`, which equals `0`. The new midpoint is `0`, so we find the middle item and compare it with the search item, which yields the value `10`. Now, our search item is found, and the index position is returned.

Finally, we check if `index_of_first_element` is less than `index_of_last_element` or not. If this condition fails, it means that the search term is not in the list.

We can use the below code snippet to search for a term/item in the given list:

```
list1 = [10, 30, 100, 120, 500]

search_term = 10
index_position1 = binary_search_iterative(list1, search_term)
if index_position1 is None:
    print("The data item {} is not found".format(search_term))
else:
    print("The data item {} is found at position {}".format(search_term,
index_position1))



list2 = ['book','data','packt', 'structure']
```

```
search_term2 = 'structure'
index_position2 = binary_search_iterative(list2, search_term2)
if index_position2 is None:
    print("The data item {} is not found".format(search_term2))
else:
    print("The data item {} is found at position {}".format(search_term2,
index_position2))
```

The output of the above code is as follows:

```
The data item 10 is found at position 0
The data item structure is found at position 3
```

In the above code, firstly we check the search term `10` in the list, and we get the correct location, i.e. index position `0`. Further, we check the index position of the string structure in the given sorted list of data items, and we get the index position 3.

The implementation that we have discussed is based on an iterative process. However, we can also implement it using the recursive method, in which we recursively shift the pointers that point to the beginning (or starting) and end of the search list. See the following code for an example of a recursive implementation of the binary search algorithm:

```
def binary_search_recursive(ordered_list, first_element_index, last_
element_index, term):

    if (last_element_index < first_element_index):
        return None
    else:
        mid_point = first_element_index + ((last_element_index - first_
element_index) // 2)

        if ordered_list[mid_point] > term:
            return binary_search_recursive (ordered_list, first_element_
index, mid_point-1, term)
        elif ordered_list[mid_point] < term:
            return binary_search_recursive (ordered_list, mid_point+1,
last_element_index, term)
        else:
            return mid_point
```

A call to this recursive implementation of the binary search algorithm and its output is as follows:

```
list1 = [10, 30, 100, 120, 500]

search_term = 10
index_position1 =  binary_search_recursive(list1, 0, len(list1)-1, search_
term)
if index_position1 is None:
    print("The data item {} is not found".format(search_term))
else:
    print("The data item {} is found at position {}".format(search_term,
index_position1))


list2 = ['book','data','packt',  'structure']

search_term2 = 'data'
index_position2 = binary_search_recursive(list2, 0, len(list1)-1, search_
term2)
if index_position2 is None:
    print("The data item {} is not found".format(search_term2))
else:
    print("The data item {} is found at position {}".format(search_term2,
index_position2))
```

The output of the above code is as follows:

```
The data item 10 is found at position 0
The data item data is found at position 1
```

Here, the only distinction between the recursive binary search and the iterative binary search is the function definition and also the way in which `mid_point` is calculated. The calculation for `mid_point` after the `((last_element_index - first_element_index)//2)` operation must add its result to `first_element_index`. That way, we define the portion of the list to attempt the search.

In binary search, we repeatedly divide the search space (i.e. the list in which the desired item may lie) in half. We start with the complete list, and in each iteration, we compute the middle point; we only consider half the list to search for the item and the other half of the list is ignored. We repeatedly check until the value is found, or the interval is empty. Therefore, at each iteration, the size of the array reduces by half; for example, at iteration 1, the size of the list is `n`, in iteration 2, the size of the list becomes $n/2$, in iteration 3 the size of the list becomes $n/2^2$, and after *k* iterations the size of the list becomes $n/2^k$. At that time the size of the list will be equal to `1`. That means:

```
=>  n/2^k = 1
```

Applying the `log` function on both sides:

```
=> log₂(n) = log₂(2k)
=> log₂(n) = k log₂(2)
=> k = log₂(n)
```

Hence, the binary search algorithm has the worst-case time complexity of `O(log n)`.

Next, we will discuss the interpolation search algorithm.

Interpolation search

The binary search algorithm is an efficient algorithm for searching. It always reduces the search space by half by discarding one half of the search space depending on the value of the search item. If the search item is smaller than the value in the middle of the list, the second half of the list is discarded from the search space. In the case of binary search, we always reduce the search space by a fixed value of half, whereas the interpolation search algorithm is an improved version of the binary search algorithm in which we use a more efficient method that reduces the search space by more than half after each iteration.

The interpolation search algorithm works efficiently when there are uniformly distributed elements in the sorted list. In a binary search, we always start searching from the middle of the list, whereas in the interpolation search we compute the starting search position depending on the item to be searched. In the interpolation search algorithm, the starting search position is most likely to be close to the start or end of the list; if the search item is near the first element in the list, then the starting search position is likely to be near the start of the list and if the search item is near the end of the list, then the starting search position is likely to be near the end of the list.

It is quite similar to how humans perform a search on any list of items. It is based on trying to make a good guess of the index position where a search item is likely to be found in a sorted list of items.

It works in a similar way to the binary search algorithm except for the method to determine the splitting criteria to divide the data in order to reduce the number of comparisons. In the case of a binary search, we divide the data into equal halves and in the case of an interpolation search, we divide the data using the following formula:

$$mid = low\_index + \frac{(upper\_index - low\_index)}{(list[upper\_index] - list[low\_index])} * (search\_term - list[low\_index])$$

In the preceding formula, `low_index` is the lower-bound index of the list, which is the index of the smallest value, and `upper_index` denotes the index position of the highest value in the list. The `list[low_index]` and `list[upper_index]` are the lowest and highest values respectively in the list. The `search_value` variable contains the value of the item that is to be searched.

Let's consider an example to understand how the interpolation search algorithm works using the following list of seven items:

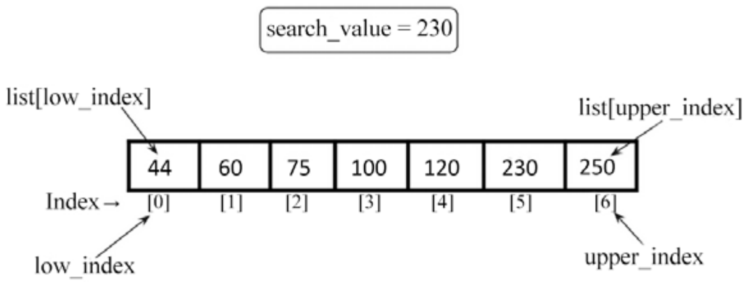

Figure 10.9: Example of interpolation search

Given the list of seven items, `44`, `60`, `75`, `100`, `120`, `230`, and `250`, the `mid` point can be computed using the above mentioned formula with the following values:

```
list1 = [4,60,75,100,120,230,250]
low_index = 0
upper_index = 6
list1[upper_index] = 250
list1[low_index] = 44
search_value = 230
```

Putting the values of all the variables in the formula, we get:

```
mid = low_index +  ((upper_index - low_index)/ (list1[upper_index] -
list1[low_index])) * (search_value - list1[low_index])
=> 0 + [(6-0)/(250-44)] * (230-44)
=> 5.41
=> 5
```

The `mid` index is 5, in the case of an interpolation search, so the algorithm starts searching from the index position 5. So, this is how we compute the midpoint from which we start searching for the given element.

The interpolation search algorithm works as follows:

1. We start searching for the given search value from the midpoint (we have just seen how to compute it).
2. If the search value matches the value stored at the index of the midpoint, we return this index position.
3. If the search value does not match the value stored at the midpoint, we divide the list into two sublists, i.e. a higher sublist and lower sublist. The higher sublist has all the elements with higher index values than the midpoint, and the lower sublist has all the elements with lower index values.
4. If the search value is greater than the value of the midpoint, we search the given search value in the higher sublist and ignore the lower sublist.
5. If the search value is lower than the value of the midpoint, we search the given search value in the lower sublist and ignore the higher sublist.
6. We repeat the process until the size of the sublists is reduced to zero.

Let us understand the implementation of the interpolation search algorithm. Firstly, we define the `nearest_mid()` method, which computes the midpoint as follows:

```
def nearest_mid(input_list, low_index, upper_index, search_value):
       mid = low_index + (( upper_index - low_index)/(input_list[upper_
index] - input_list[low_index])) * (search_value - input_list[low_index])
       return int(mid)
```

The `nearest_mid` function takes, as arguments, the lists on which to perform the search. The `low_index` and `upper_index` parameters represent the bounds in the list within which we are hoping to find the search term. Furthermore, `search_value` represents the value being searched for.

In an interpolation search, the midpoint is generally more to the left or right. This is caused by the effect of the multiplier being used when dividing to obtain the midpoint. The implementation of the interpolation algorithm remains the same as that of the binary search except for the way we compute the midpoint.

In the following code, we provide the implementation of the interpolation search algorithm:

```
def interpolation_search(ordered_list, search_value):
    low_index = 0
    upper_index = len(ordered_list) - 1
    while low_index <= upper_index:
        mid_point = nearest_mid(ordered_list, low_index, upper_index,
search_value)
        if mid_point > upper_index or mid_point < low_index:
            return None
        if ordered_list[mid_point] == search_value:
            return mid_point
        if search_value > ordered_list[mid_point]:
            low_index = mid_point + 1
        else:
            upper_index = mid_point - 1
    if low_index > upper_index:
        return None
```

In the above code, we initialize the `low_index` and `upper_index` variables for the given sorted list. We firstly compute the midpoint using the `nearest_mid()` method.

The computed midpoint using the `nearest_mid` function may produce values that are greater than `upper_bound_index` or lower than `lower_bound_index`. When this occurs, it means the search term, `term`, is not in the list. None is, therefore, returned to represent this.

Next, we match the search value with the value stored at the midpoint, i.e. `ordered_list[mid_point]`. If that matches, the index of the midpoint is returned; if it does not match, then we divide the lists into higher and lower sublists, and we readjust `low_index` and `upper_index` so that the algorithm will focus on the sublist that is likely to contain the search term similar to what we did in the binary search:

```
        if search_value > ordered_list[mid_point]:
            low_index = mid_point + 1
        else:
            upper_index = mid_point - 1
```

In the above code, we check if the search value is greater than the value stored at `ordered_list[mid_point]`, then we only adjust the `low_index` variable to point to the `mid_point + 1` index.

Let's see how this adjustment occurs. Suppose we want to search for `190` in the given list in *Figure 10.10*, then the midpoint will be `4` as per the above formula. Then we compare the search value (i.e. `190`) with the value stored at the midpoint (i.e. `120`). Since the search value is greater, we search for the element in the higher sublist, and readjust the `low_index` value. This is shown in *Figure 10.10*:

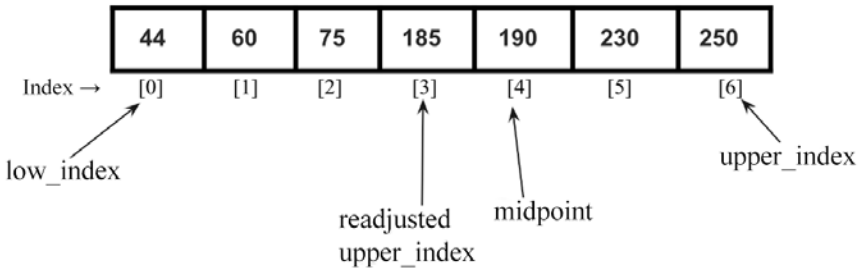

Figure 10.10: Readjustment of the low_index when the value of the search item is greater than the value at the midpoint

On the other hand, if the value of the search term is less than the value stored at `ordered_list[mid_point]`, then we only adjust the `upper_index` variable to point to the index `mid_point - 1`. For example, if we have the list shown in *Figure 10.11*, and we want to search for `185`, then the midpoint will be `4` as per the formula.

Next, we compare the search value (i.e. 185) with the value stored at the midpoint (i.e. 190). Since the search value is less as compared to `ordered_list[mid_point]`, we search for the element in the lower sublist, and readjust the `upper_index` value. This is shown in *Figure 10.11*:

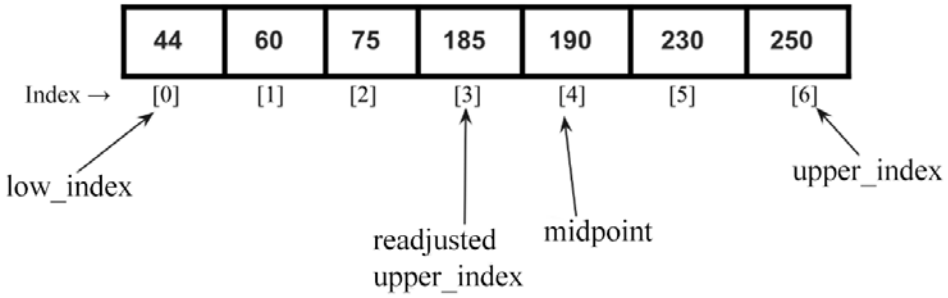

Figure 10.11: Readjustment of the upper_index when the search item is less than the value at the midpoint

The following code snippet can be used to create a list of elements `{44, 60, 75, 100, 120, 230, 250}`, in which we want to search for `120` using the interpolation search algorithm.

```
list1 = [44, 60, 75, 100, 120, 230, 250]
a = interpolation_search(list1, 120)
print("Index position of value 2 is ", a)
```

The output of the above code is as follows:

```
Index position of value 2 is  4
```

Let's use a more practical example to understand the inner workings of both the binary search and interpolation algorithms.

Consider for example the following list of elements:

```
[ 2, 4, 5, 12, 43, 54, 60, 77]
```

At index 0, the value 2 is stored, and at index 7, the value 77 is stored. Now, assume that we want to find element 2 in the list. How will the two different algorithms go about it?

If we pass this list to the `interpolation search` function, then the `nearest_mid` function will return a value equal to `0` using the formula of `mid_point` computation, which is as follows:

```
mid_point = 0 + [(7-0)/(77-2)] * (2-2)
          = 0
```

As we get the `mid_point` value `0`, we start the interpolation search with the value at index `0`. Just with one comparison, we have found the search term.

On the other hand, the binary search algorithm needs three comparisons to arrive at the search term, as illustrated in *Figure 10.12*:

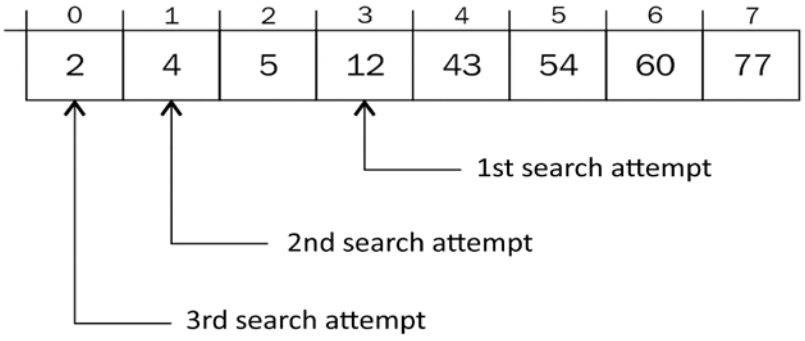

Figure 10.12: Three comparisons are required to search for the item using the binary search algorithm

The first `mid_point` value calculated is `3`. The second `mid_point` value is `1` and the last `mid_point` value where the search term is found is `0`. So, we reach the desired search item in three comparisons whereas in interpolation search we find the desired item on the first attempt.

The interpolation search algorithm works well when the data set is sorted, and uniformly distributed. In this case, the average case time complexity is `O(log(log n))` in which `n` is the length of the array. Moreover, if the dataset is randomized, in that case, the worst-case time complexity of the interpolation search algorithm will be `O(n)`. So, interpolation search may work better than binary search if the given data is uniformly distributed.

Exponential search

Exponential search is another search algorithm that is mostly used when we have large numbers of elements in a list. Exponential search is also known as galloping search and doubling search. The exponential search algorithm works in the following two steps:

1. Given a sorted array of `n` data elements, we first determine the subrange in the original list where the desired search item may be present
2. Next, we use the binary search algorithm to find out the search value within the subrange of data elements identified in *step 1*

Firstly, in order to find out the subrange of data elements, we start searching for the desired item in the given sorted array by jumping 2^i elements every iteration. Here, `i` is the value of the index of the array. After each jump, we check if the search item is present between the last jump and the current jump. If the search item is present then we use the binary search algorithm within this subarray, and if it is not present, we move the index to the next location. Therefore, we first find the first occurrence of an exponent `i` such that the value at index 2^i is greater than the search value. Then, the 2^i becomes the lower bound and 2^i-1 becomes the upper bound for this range of data elements in which the search value will be present. The exponential search algorithm is defined as follows:

1. First, we check the first element `A[0]` with the search element.
2. Initialize the index position `i=1`.
3. We check two conditions: (1) if it is the end of the array or not (i.e. 2^i < len(A)), and (2) if `A[i] <= search_value`). In the first condition, we check if we have searched the complete list, and we stop if we have reached the end of the list. In the second condition, we stop searching when we reach an element whose value is greater than the search value, because it means the desired element will be present before this index position (since the list is sorted).
4. If either of the above two conditions is true, we move to the next index position by incrementing `i` in powers of 2.
5. We stop when either of the two conditions of *step 3* is satisfied.
6. We apply the binary search algorithm on the range 2^i//2 to min (2^i, len(A)).

Let's take an example of a sorted array of elements `A = {3, 5, 8, 10, 15, 26, 35, 45, 56, 80, 120, 125, 138}` in which we want to search for the element `125`.

We start with comparing the first element at index i = 0, i.e. A[0] with the search element. Since A[0] < `search_value`, we jump to the next location 2^i with i = 0, since A[2^0] < `search_value`, the condition is true, hence we jump to the next location with i = 1 i.e. A[22^1] < `search_value`. We again jump to the next location 2^i with i = 2, since A[2^2] < `search_value`, the condition is true. We iteratively jump to the next location until we complete searching the list or the search value is greater than the value at that location, i.e. A[2^i] < len(A) or A[2^i] <= `search_value`. Then we apply the binary search algorithm on the range of the subarray. The complete process for searching a given element in the sorted array using the exponential search algorithm is depicted in *Figure 10.13*:

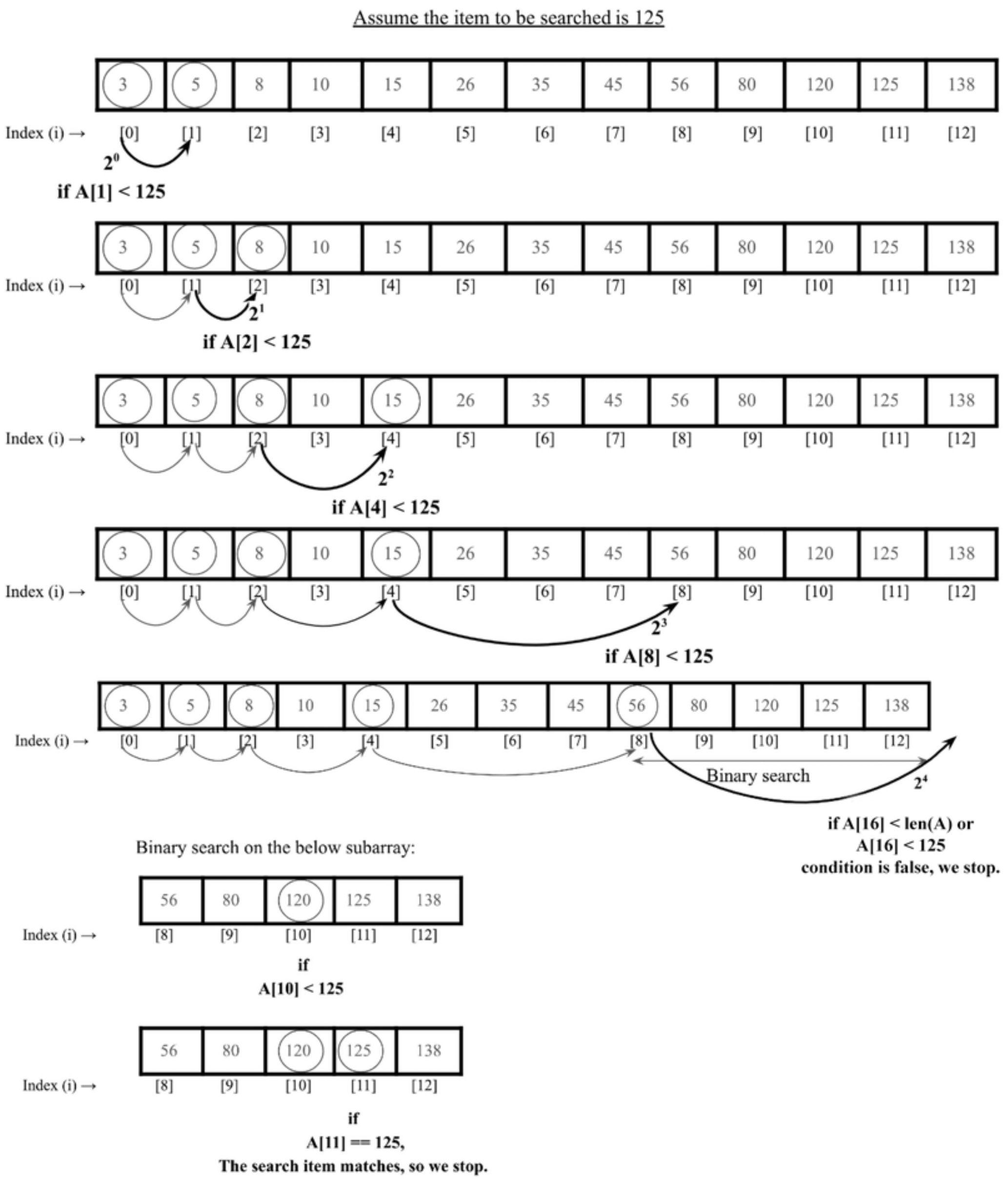
Assume the item to be searched is 125
3
5
8
10
15
26
35
45
56
80
120
125
138
Index (i) →
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
[9]
[10]
[11]
[12]
2^0
if A[1] < 125
2^1
if A[2] < 125
2^2
if A[4] < 125
2^3
if A[8] < 125
Binary search
2^4
if A[16] < len(A) or A[16] < 125 condition is false, we stop.
Binary search on the below subarray:
if A[10] < 125
if A[11] == 125, The search item matches, so we stop.

Figure 10.13: Illustration of the exponential search algorithm

Now, let us discuss the implementation of the exponential search algorithm. Firstly, we implement the binary search algorithm, which we have already discussed in the previous section, but for the completeness of this algorithm it is given again as follows:

```
def binary_search_recursive(ordered_list, first_element_index, last_
element_index, term):

    if (last_element_index < first_element_index):
        return None
    else:
        mid_point = first_element_index + ((last_element_index - first_
element_index) // 2)

        if ordered_list[mid_point] > term:
            return binary_search_recursive (ordered_list, first_element_
index, mid_point-1, term)
        elif ordered_list[mid_point] < term:
            return binary_search_recursive (ordered_list, mid_point+1,
last_element_index, term)
        else:
            return mid_point
```

In the above code, given the ordered list of elements, it returns the index of the location where the given data element is found in the list. It returns `None` if the desired element is not found in the list. Next, we implement the `exponential_search()` method as follows:

```
def exponential_search(A, search_value):
    if (A[0] == search_value):
        return 0
    index = 1
    while index < len(A) and A[index] < search_value:
        index *= 2
    return binary_search_recursive(A, index // 2, min(index, len(A) - 1),
search_value)
```

In the above code, firstly, we compare the first element `A[0]` with the search value. If it matches then the index position `0` is returned. If that does not match, we increase the index position to 2^0, i.e. 1. We check `A[1] < search_value`. Since the condition is true, we jump to the next location 2^1, i.e. we compare `A[2] < search_value`. Since the condition is true, we move to the next location.

We iteratively increase the index position in the power of 2 until the stop condition is satisfied:

```
    while index < len(A) and A[index] < search_value:
        index *= 2
```

Finally, when the stopping criteria are met, we use the binary search algorithm to search for the desired search value within the subrange as follows:

```
    return binary_search_recursive(A, index // 2, min(index, len(A) - 1), 
search_value)
```

Finally, the `exponential_search()` method returns the index position if the search value is found in the given array; otherwise, `None` is returned.

```
print(exponential_search([1,2,3,4,5,6,7,8,9, 10, 11, 12, 34, 40], 34))
```

The output of the above code snippet is:

```
12
```

In the above output, we get index position `12` for the search item `34` in the given array.

The exponential search is useful for very large-sized arrays. This is better than binary search because instead of performing a binary search on the complete array, we find a subarray in which the element may be present and then apply binary search, so it reduces the number of comparisons.

The worst-case time complexity of exponential search is O($\log_2 i$), where `i` is the index where the element to be searched is present. The exponential search algorithm can outperform binary search when the desired search element is present at the beginning of the array.

We can also use exponential search to search in bounded arrays. It can even out-perform binary search when the target is near the beginning of the array, since exponential search takes `O(log(i))` time whereas the binary search takes `O(log n)` time, where `n` is the total number of elements. The best-case complexity of exponential search is `O(1)`, when the element is present at the first location of the array.

Next, let us discuss how to decide which search algorithm we should choose for a given situation.

Choosing a search algorithm

Now that we've covered the different types of search algorithms, we can look into which ones work better and in what situations. The binary search and interpolation search algorithms are better in performance compared to both ordered and unordered linear search functions. The linear search algorithm is slower because of the sequential probing of elements in the list to find the search term.

Linear search has a time complexity of `O(n)`. The linear search algorithm does not perform well when the given list of data elements is large.

The binary search operation, on the other hand, slices the list in two anytime a search is attempted. On each iteration, we approach the search term much faster than in a linear strategy. The time complexity yields `O(log n)`. The binary search algorithm performs well but the drawback of it is that it requires a sorted list of elements. So, if the given data elements are short and unsorted then it is better to use the linear search algorithm.

Interpolation search discards more than half of the list of items from the search space, and this gives it the ability to get to the portion of the list that holds a search term more efficiently. In the interpolation search algorithm, the midpoint is computed in such a way that it gives a higher probability of obtaining the search term faster. The average-case time complexity of interpolation search is `O(log(log n))`, whereas the worst-case time complexity of the interpolation search algorithm is `O(n)`. This shows that interpolation search is better than binary search and provides faster searching in most cases.

Therefore, if the list is short and unsorted, then the linear search algorithm is suitable, and if the list is sorted and not very big then the binary search algorithm can be used. Further, the interpolation search algorithm is good to use if the data elements in the list are uniformly distributed. If the list is very large, then the exponential search algorithm and jump search algorithm can be used.

Summary

In this chapter, we discussed the concept of searching for a given element from a list of data elements. We discussed several important search algorithms, such as linear search, binary search, jump search, interpolation search, and exponential search. The implementations of these algorithms were discussed using Python in detail. We will be discussing sorting algorithms in the next chapter.

Exercise

1. On average, how many comparisons are required in a linear search of `n` elements?
2. Assume there are eight elements in a sorted array. What is the average number of comparisons that will be required if all the searches are successful and if the binary search algorithm is used?
3. What is the worst-case time complexity of the binary search algorithm?
4. When should the interpolation search algorithm perform better than the binary search algorithm?

Join our community on Discord

Join our community's Discord space for discussions with the author and other readers:

`https://packt.link/MEvK4`