codecademy

Linear & Binary Search

Searching for smallest or largest value using linear search

Linear search can be used to search for the smallest or largest value in an unsorted list rather than searching for a match. It can do so by keeping track of the largest (or smallest) value and updating as necessary as the algorithm iterates through the dataset.

```
function getHighestValueUsingLinearSearch(array) {
  // highest value is the first value to start with
  let highest = array[0];

  for(let i = 1; i < array.length; i++) {
    if(highest < array[i]) {
      highest = array[i];
    }
  }
  return highest;
}
```

Linear Search best case

For a list that contains n items, the best case for a linear search is when the target value is equal to the first element of the list. In such cases, only one comparison is needed. Therefore, the best case performance is O(1).

Linear Search Complexity

Linear search runs in linear time and makes a maximum of n comparisons, where n is the length of the list.
Hence, the computational complexity for linear search is O(N).
The running time increases, at most, linearly with the size of the items present in the list.

Linear Search expressed as a Function

A linear search can be expressed as a function that compares each item of the passed dataset with the target value until a match is found.
The given JavaScript code block demonstrates a function that performs a linear search. The relevant index is returned if the target is found and -1 if it is not.

```
function linearSearch(array, target) {
  for(let i = 0; i < array.length; i++) {
    if (target == array[i]) {
      return i;
    }
  }
  return -1;
}
```

Return value of a linear search

A function that performs a linear search can return a message of success and the index of the matched value if the search can successfully match the target with an element of the dataset. In the event of a failure, an indication is returned as well.

For instance, in the given code block, the index `i` will be returned if a value is matched with the target. The value `-1` will be returned if there was no match.

```
function linearSearch(array, target) {
  for (let i = 0; i < array.length; i++) {
    if (target == array[i]) {
      console.log("Target found!");
      return i;
    }
  }
  console.log("Target not found!");
  return -1;
}
```

Modification of linear search function

A linear search can be modified so that all instances in which the target is found are returned. This change can be made by not 'breaking' when a match is found.

```
function linearSearch(array, target) {
  let hit_array = [];
  for (let i = 0; i < array.length; i++) {
    if (target == array[i]) {
      hit_array.push(i);
    }
  }
  return hit_array;
}
```

Linear search

Linear search sequentially checks each element of a given list for the target value until a match is found. If no match is found, a linear search would perform the search on all of the items in the list.

For instance, if there are `n` number of items in a list, and the target value resides in the `n-5` th position, a linear search will check `n-5` items total.

Linear search as a part of complex searching problems

Despite being a very simple search algorithm, linear search can be used as a subroutine for many complex searching problems. Hence, it is convenient to implement linear search as a function so that it can be reused.

Linear Search Best and Worst Cases

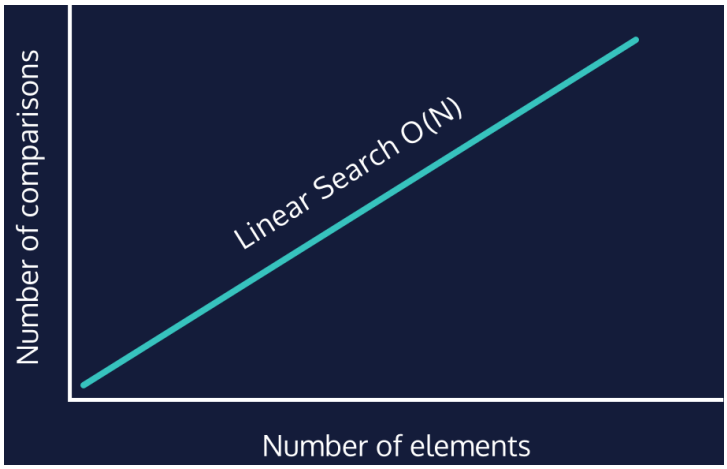
The best-case performance for the Linear Search algorithm is when the search item appears at the beginning of the list and is O(1). The worst-case performance is when the search item appears at the end of the list or not at all. This would require N comparisons, hence, the worse case is O(N).

Linear Search Average Runtime

The Linear Search Algorithm performance runtime varies according to the item being searched. On average, this algorithm has a Big-O runtime of O(N), even though the average number of comparisons for a search that runs only halfway through the list is N/2.

Linear Search Runtime

The Linear Search algorithm has a Big-O (worst case) runtime of O(N). This means that as the input size increases, the speed of the performance decreases linearly. This makes the algorithm not efficient to use for large data inputs.



Complexity of Binary Search

A dataset of length n can be divided *log n* times until everything is completely divided. Therefore, the search complexity of binary search is O(log n).

Iterative Binary Search

A binary search can be performed in an iterative approach. Unlike calling a function within the function in a recursion, this approach uses a loop.

```
function binSearchIterative(target, array, left, right) {
  while(left < right) {
    let mid = (right + left) / 2;
    if (target < array[mid]) {
      right = mid;
    } else if (target > array[mid]) {
      left = mid;
    } else {
      return mid;
    }
  }
  return -1;
}
```

Updating pointers in a recursive binary search

In a recursive binary search, if the value has not been found then the recursion must continue on the list by updating the left and right pointers after comparing the target value to the middle value.
If the target is less than the middle value, you know the target has to be somewhere on the left, so, the right pointer must be updated with the middle index. The left pointer will remain the same. Otherwise, the left pointer must be updated with the middle index while the right pointer remains the same. The given code block is a part of a function `binarySearchRecursive()` .

```
function binarySearchRecursive(array, first, last, target) {
  let middle = (first + last) / 2;
  // Base case implementation will be in here.

  if (target < array[middle]) {
    return binarySearchRecursive(array, first, middle, target);
  } else {
    return binarySearchRecursive(array, middle, last, target);
  }
}
```

Binary Search

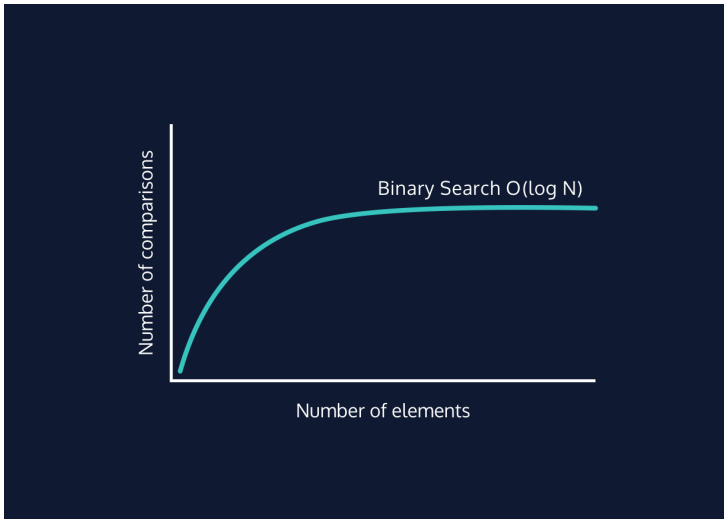
Binary search performs the search for the target within a sorted array. Hence, to run a binary search on a dataset, it must be sorted prior to performing it.

Operation of a Binary Search

The binary search starts the process by comparing the middle element of a sorted dataset with the target value for a match. If the middle element is equal to the target value, then the algorithm is complete. Otherwise, the half in which the target cannot logically exist is eliminated and the search continues on the remaining half in the same manner.
The decision of discarding one half is achievable since the dataset is sorted.

Binary Search Performance

The binary search algorithm takes time to complete, indicated by its `time complexity`. The worst-case time complexity is `O(log N)`. This means that as the number of values in a dataset increases, the performance time of the algorithm (the number of comparisons) increases as a function of the base-2 logarithm of the number of values.
Example: Binary searching a list of 64 elements takes at MOST `log2(64)` = 6 comparisons to complete.



Binary Search

The binary search algorithm efficiently finds a goal element in a sorted dataset. The algorithm repeatedly compares the goal with the value in the middle of a subset of the dataset. The process begins with the whole dataset; if the goal is smaller than the middle element, the algorithm repeats the process on the smaller (left) half of the dataset. If the goal is larger than the middle element, the algorithm repeats the process on the larger (right) half of the dataset. This continues until the goal is reached or there are no more values.

Throwing Exception in Linear Search

The linear search function may throw a `ValueError` with a message when the target value is not found in the search list. Calling the linear search function inside a `try` block is recommended to catch the `ValueError` exception in the `except` block.

```
def linear_search(lst, match):
  for idx in range(len(lst)):
    if lst[idx] == match:
      return idx
    else:
      raise ValueError("{0} not in
list".format(match))

recipe = ["nori", "tuna", "soy sauce",
"sushi rice"]
ingredient = "avocado"
try:
  print(linear_search(recipe, ingredient))
except ValueError as msg:
  print("{0}".format(msg))
```

Find Maximum Value in Linear Search

The Linear Search function can be enhanced to find and return the maximum value in a list of numeric elements. This is done by maintaining a variable that is compared to every element and updated when its value is smaller than the current element.

```
def find_maximum(lst):
  max = None
  for el in lst:
    if max == None or el > max:
      max = el
  return max
test_scores = [88, 93, 75, 100, 80, 67,
71, 92, 90, 83]
print(find_maximum(test_scores)) # returns
100
```

Linear Search Multiple Matches

A linear search function may have more than one match from the input list. Instead of returning just one index to the matched element, we return a list of indices. Every time we encounter a match, we add the index to the list.

```
def linear_search(lst, match):
  matches = []
  for idx in range(len(lst)):
    if lst[idx] == match:
      matches.append(idx)
  if matches:
    return matches
  else:
    raise ValueError("{0} not in
list".format(match))

scores = [55, 65, 32, 40, 55]
print(linear_search(scores, 55))
```

Raise Error in Linear Search

A Linear Search function accepts two parameters:

```
1) input list to search from
2) target element to search for in
the input list
```

If the target element is found in the list, the function returns the element index. If it is not found, the function raises an error. When implementing in Python, use the `raise` keyword with `ValueError()`.

```
def linear_search(lst, match):
  for idx in range(len(lst)):
    if lst[idx] == match:
      return idx
  raise ValueError('Sorry, {0} is not
found.'.format(match))
```

Recursive Binary Search

Binary search can be implemented using recursion by creating a function that takes in the following arguments: a sorted list, a left pointer, a right pointer, and a target. The base cases must account for when the left and right pointers are equal, as well as when the target is found, in which case the the index of the array is returned. Initially, set the left pointer to index 0 of the list and right pointer to the last index. If middle value > target, right pointer = middle value. If middle value < target, left pointer = middle value. Call the binary search function with the properly adjusted pointers.

```
def binary_search(sorted_list,
left_pointer, right_pointer, target):
  if left_pointer >= right_pointer:
    #base case 1
        #mid_val and mid_idx defined here
  if mid_val == target:
    #base case 2
  if mid_val > target:
    #recursive call with left pointer
  if mid_val < target:
    #recursive call with right pointer
```