codecademy

Searching and Sorting

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.

```
Create a variable called max_value_i
Set max_value_index to the index of
  For each element in the search lis
    if element is greater than the e
      Set max_value_index equal to t
return max_value_index
```

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 pseudocode block demonstrates a function that performs a linear search. The relevant index is returned if the target is found and -1 with a message that a value is not found if it is not.

```
For each element in the array
  if element equal target value then
    return its index
  if element is not found, return
    “Value Not Found” message
```

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, a message as well as `-1` is returned as well.

```
For each element in the array
  if element equal target value then
    print success message
    return its index
  if element is not found
    print Value not found message
    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.

```
For each element in the searchList
  if element equal target value then
    Add its index to a list of occur
if the list of occurrences is empty
  raise ValueError
otherwise
  return the list occurrences
```

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

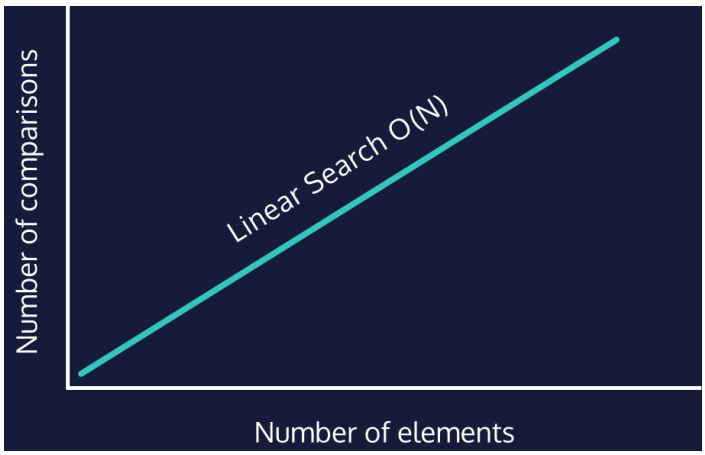
The best-case performance for the Linear Search algorithm is when the search item appears at the beginning of the list and is O(1). The worst-case performance is when the search item appears at the end of the list or not at all. This would require N comparisons, hence, the worse case is O(N).

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.



Java Linear Search Algorithm

Linear search will start with the first element and check if it is a match for our target element, and will continue

the search till it finds a match. The steps are:
Step 1: Examine the current element in the list.
Step 2: If the current element is equal to the target value, stop.
Step 3: If the current element is not equal to the target value, check the next element in the list.
Continue steps 1 - 3 until the element is found or the end of the list is reached.

```
for each element in the search_list
    if element equal target value th
       print location and return ind
if element is not found then
    print message not found and retu
```

Return Value of a Linear Search

A method 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, a message indicating the value was not found along with returning a `-1`.

```
class LinearSearch
{
  public static int linearSearch(int[]
arr, int target)
  {
      // loop through the input array
        // check if array at current
index is equal to `target`
          // if true
          System.out.println("Element is
present at index " + i);
          return i;
      // if we finish looping through our
array and the element is not found
      System.out.println("Element is not
present in the array");
      return -1;
  }
  public static void main(String args[])
  {
      int arr[] = { 2, 3, 4, 10, 40 };
      int target = 10;
      linearSearch(arr, target);
  }
}
```

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).

2D Array Linear Search

In Java, a linear search on a 2D array is usually accomplished with nested for loops.

```
for each row in the search_list
  for each column of that row
    if element at that row and colum
      print the index of the row and
   print message that element is not
```

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;
}
```

Base case in a binary search using recursion

In a recursive binary search, there are two cases for which that is no longer recursive. One case is when the middle is equal to the target. The other case is when the search value is absent in the list.

```
binary_search(sorted_list, left_poin
  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 pointe
```

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 binarySearchRecu
middle, last, target);
    }
}
```

Binary Search Sorted Dataset

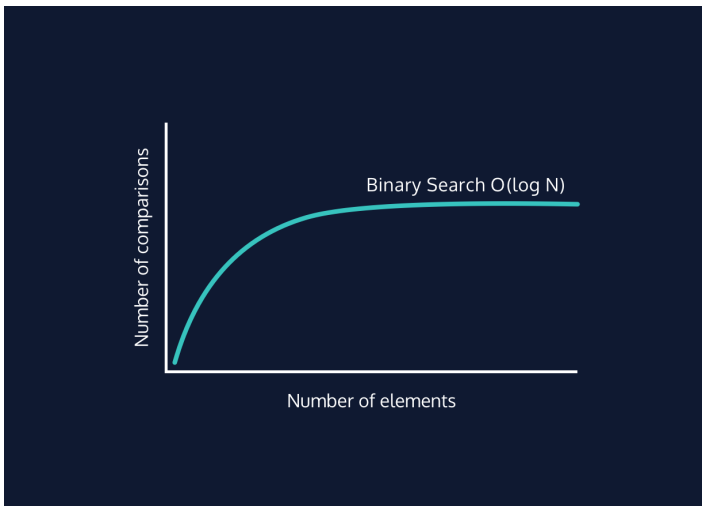
Binary search performs the search for the target within a sorted array. Hence, to run a binary search on a dataset, it must be sorted prior to performing it.

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.

Iterative Binary Search Middle Index

The iterative binary search method take a `sorted array`, and `target`, as arguments. When the method runs the first time the `left`, the first index of the input array is `0`, while the `right`, the last index of the input array, is equal to its length - 1. The `mid` is the middle index of the input array. Now, the algorithm runs a while loop comparing the `target` with the array value of the `mid` index of the input array.

```
class BinarySearch
{
  public static int binarySearch(int[]
arr, int target)
  {
     int left = 0;
     int right = array.length - 1;

    while (left <= right)
    {

      int mid = (left + right) / 2;

      if (target == array[mid]) {
        return mid;
      }
     // code to run if `target` is
greater or lesser than `array[mid]`
    }

    return -1;
  }
}
```

Iterative Binary Search Algorithm

The iterative `binarySearch()` algorithm is implemented with the following steps:

- Accepts a sorted array and target value as parameters
- Sets a `left` integer to `0` and `right` integer to `arr.length`
- Executes a `while` loop as long as `right` is greater than `left`
- Inside the `while` loop, finds the value in the array at the middle index, equal to the average of `left` and `right`
- If the current mid-value equals the target value, return the middle index
- If the target value is greater than the value being checked, set `left` equal to the middle index + 1
- If the target value is less than the value being checked, set `right` equal to the middle index
- Outside of the `while` loop, return `-1`

```
public class BinarySearch {
  public static int search(int[] arr, int target) {
    int left = 0;
    int right = arr.length;
    while (left < right) {
      int mid = Math.floorDiv(left + right, 2);
      int midValue = arr[mid];
      if (midValue == target) {
        return mid;
      } else if (midValue < target) {
        left = mid + 1;
      } else {
        right = mid;
      }
    }

    return -1;
  }
}
```

Selection Sort Algorithm

Selection sort works by repeatedly finding the minimum element of an unsorted array, and moving that element to the end of the "sorted" part of the array. It "selects" the smallest element and places it in the correct spot.

```
class Main {
  public static void selectionSort (int arr[]) {

    int size = arr.length;

    for (int i = 0; i < size - 1; i++) {

      int currentMinimumIndex = i;
      for (int j = i + 1; j < arr.length; j++) {
        if (arr[j] < arr[currentMinimumIndex]) {
          currentMinimumIndex = j;
        }
      }

      int temp = arr[currentMinimumIndex];
      arr[currentMinimumIndex] = arr[i];
```

```
      arr[i] = temp;
    }
  }

  public static void main(String args[])
{
  }
}
```

Selection Sort Nested Loops

Selection sort makes use of nested for loops. The inner for loop finds the smallest number from the section of the unsorted array and moves it to the end of the sorted section of the array. The outer for loop makes this process happen n times - once for each element that needs to be put into the sorted section of the array

```
public static void selectionSort (int
arr[]) {

    //for loop to loop through all
unsorted elements
    for (int i = 0; i < size - 1; i++) {
        // for loop to find the smallest
value of unsorted elements
        for (int j = i + 1; j <
arr.length; j++) {
            // Code that keeps track of
the smallest value
        }
        // Code that swaps the smallest
value to the correct place
    }

  }
```

Selection Sort Runtime

Nested loops are generally an indicator of quadratic complexity. This means that as the number of elements n increases, the running time increases quadratically. This means that if n doubles, we know that sorting time will quadruple, resulting in a runtime of O(n^2) due to the nested loop structure.

```
  public static void selectionSort (int
arr[]) {

    // nested for loop indicates runtime
of O(n^2)
    for (int i = 0; i < size - 1; i++) {
        for (int j = i + 1; j <
arr.length; j++) {
        }
    }
  }
```

Selection Sort Number of Comparison Statements

The number of comparison statements made by selection sort is less than n^2. This is because the inner for loop that searches for the smallest remaining value in the unsorted section of the list does not have to search through all n items. It only needs to search through the remaining unsorted values. Nevertheless, the runtime remains O(n^2).

```
class Main {
  public static void selectionSort (int
arr[]) {

    // outer for loop makes n - 1
comparisons
    for (int i = 0; i < size - 1; i++) {

     // inner for loop only searches
remaining unsorted values
      for (int j = i + 1; j < arr.length;
j++) {
       // code that keeps track of the
smallest value
      }
     // code that swaps elements
  }
}

  public static void main(String args[])
{
  }
}
```

Selection Sort Best and Worst Case Comparisons

In both best case and worst case scenarios, selection sort makes approximately n^2 comparisons. Even if the list is already sorted, the inner for loop will have to make n comparisons to find the smallest remaining unsorted item.

```
// worst case scenario
int[] inputArray1 = { 19, 15, 12, 7 };
selectionSort(inputArray1);
// determine 1st index
  // compare 19 to 15, then 15 to 12,
then 12 to 7
  // 7 is the lowest value, swap with 19
  // 3 comparisons were made

// best case scenario
int[] inputArray2 = { 7, 12, 15, 19 };
selectionSort(inputArray2);
// determine 1st index
  // compare 7 to 12, then 7 to 15, then
7 to 19
```

```
  // 7 is the lowest, no swap
  // 3 comparisons were made
```

Selection Sort Swap

Sorting occurs as selection sort swaps the element in the first position of the unsorted sub-list with the element with the lowest value in the remainder of the unsorted sub-list.

```
public static void selectionSort (int
arr[]) {

  // for loop to loop through all
unsorted elements
  for (int i = 0; i < size - 1; i++) {
      // for loop to find the smallest
value of unsorted elements
      for (int j = i + 1; j < arr.length;
j++) {
           // Code that keeps track of
the smallest value
      }
      // Code that swaps the smallest
value to the correct place
      int temp =
arr[currentMinimumIndex];
      arr[currentMinimumIndex] = arr[i];
      arr[i] = temp;
  }
}
```

Insertion Sort Algorithm

Insertion sort algorithm works by repeatedly taking the next unsorted item in an unsorted list and inserting that item into the correct location in a sorted list.

```
class InsertionSort {
  public static void sort(int[] array) {
   for (int i = 1; i < array.length; i++)
{
     int current = array[i];
     int j = i -1;
     while (j >= 0 && array[j] > current)
{
       array[j + 1] = array[j];
       j--;
     }
     array[j+1] = current;

   }
```

```
  }
  public static void main(String[] args)
{
  }
}
```

Insertion Sort Nested Loops

Insertion sort uses nested loops. The inner loop iterates through the list of items that are already sorted looking for the correct place to insert the most recent unsorted item. This loop happens n times - once for each item that needs to be inserted.

```
class InsertionSort {
  public static void sort(int[] array) {
   // outer loop iterates through input
array starting with second element
   for (int i = 1; i < array.length; i++)
{
     // store value of current element

     // inner loop
     while (j >= 0 && array[j] > current)
{
       // compare current element to
predecessor(s)
     }
     // move the greater element(s) one
position to make space for swapped
element
   }
  }
}
```

Insertion Sort Best Case Runtime

In the case of an ascending array, the best case scenario, the runtime of insertion sort is O(n). As each element gets picked for insertion into the sorted list, it will only take one comparison to find the correct place to insert the new item. 1 comparison will happen n times, for a total runtime of O(n).

```
int[] ascendingNumbers = {2, 4, 6, 8};
insertionSort(ascendingNumbers);
// sort second element - 4 is greater
than 2 - no shift or further comparisons
// sort third element - 6 is greater than
4 - no shift or further comparisons
// sort fourth element - 8 is greater
than 6 - no shift or further comparisons
```

Java Insertion Sort Runtime

Nested loops are generally an indicator of quadratic complexity. This means that as the number of elements

```
class InsertionSort {
  public static void sort(int[] array) {
```

n increases, the running time increases quadratically. This means that if n doubles, we know that sorting time will quadruple, resulting in a runtime of O(n^2) due to the nested loop structure.

```
    for (int i = 1; i < array
{
      // nested loop structure indicates
O(n^2)
      while (j >= 0 && array[j] > current)
{
          // compare current element to
predecessor
      }
      // code to swap
    }
  }
}
```

Insertion Sort Worst Case

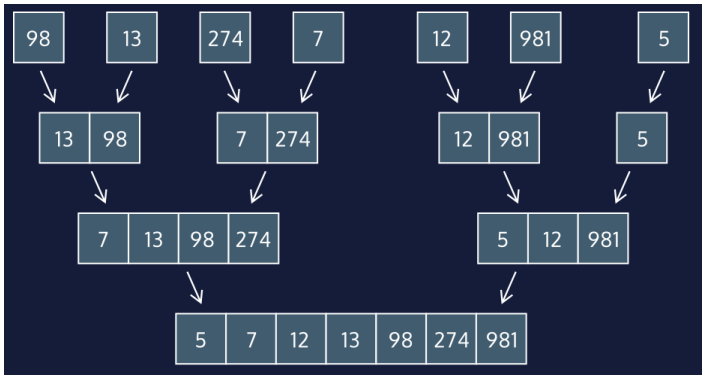
In the case of a descending array, our worst case scenario, insertion sort will have to make approximately n^2 comparisons. This happens when the list to sort is in perfect reverse order. All n items that need to be inserted into the sorted section of the list will be compared to every item in the already sorted section of the list.

```
int[] descendingNumbers = {8, 6, 4, 2};
insertionSort(descendingNumbers);
// result of first sort - {6, 8, 4, 2}
// result of second sort - {4, 6, 8, 2}
// comparisons to sort last element - 2
is less than 8, 2 is less than 6, 2 is
less than 4, insert 2
```

Merge Sort Merging

Merge Sort is a divide and conquer algorithm. It consists of two parts:

```
1) splitting the original list into smaller s
2) merging back the presorted 1-element lists
```



The merging portion is iterative and takes 2 sublists. The first element of the left sublist is compared to the first element of the right sublist. If it is smaller, it is added to a new sorted list, and removed from the left sublist. If it is bigger, the first element of the right sublist is added instead to the sorted list and then removed from the right sublist. This is repeated until either the left or right sublist is empty. The remaining non-empty sublist is appended to the sorted list.

Big-O Runtime for Merge Sort

The Merge Sort algorithm is divided into two parts. The first part repeatedly splits the input list into smaller lists to eventually produce single-element lists. The best, worst and average runtime for this part is Θ(log N). The second part repeatedly merges and sorts the single-element lists to twice its size until the original input size

is achieved. The best, worst and average runtime for this part is Θ(N). Therefore, the combined runtime is Θ(N log N).