Agenda

- **Quick Sort**
- **Counting Sort**
- **Radix Sort**
- **What is Searching algorithm?**
- **Intro to Linear Search**
- **Intro to Binary Search**

Quick Sort

- Quick Sort is a popular and efficient sorting algorithm. It is also, however, one of the less intuitive!
- Quick Sort takes in some data, puts it in order, then spits it out.

- Quick Sort is an in-place, unstable, and comparison-type algorithm.

> Note : “in-place algorithm” - an algorithm which transforms input using no auxiliary data structure. However, a small amount of extra storage space is allowed for auxiliary variables.

Unstable means that two elements with equal values can appear in different order in the sorted output compared with their order in the unsorted inp array.

For example, if we wanted to sort:

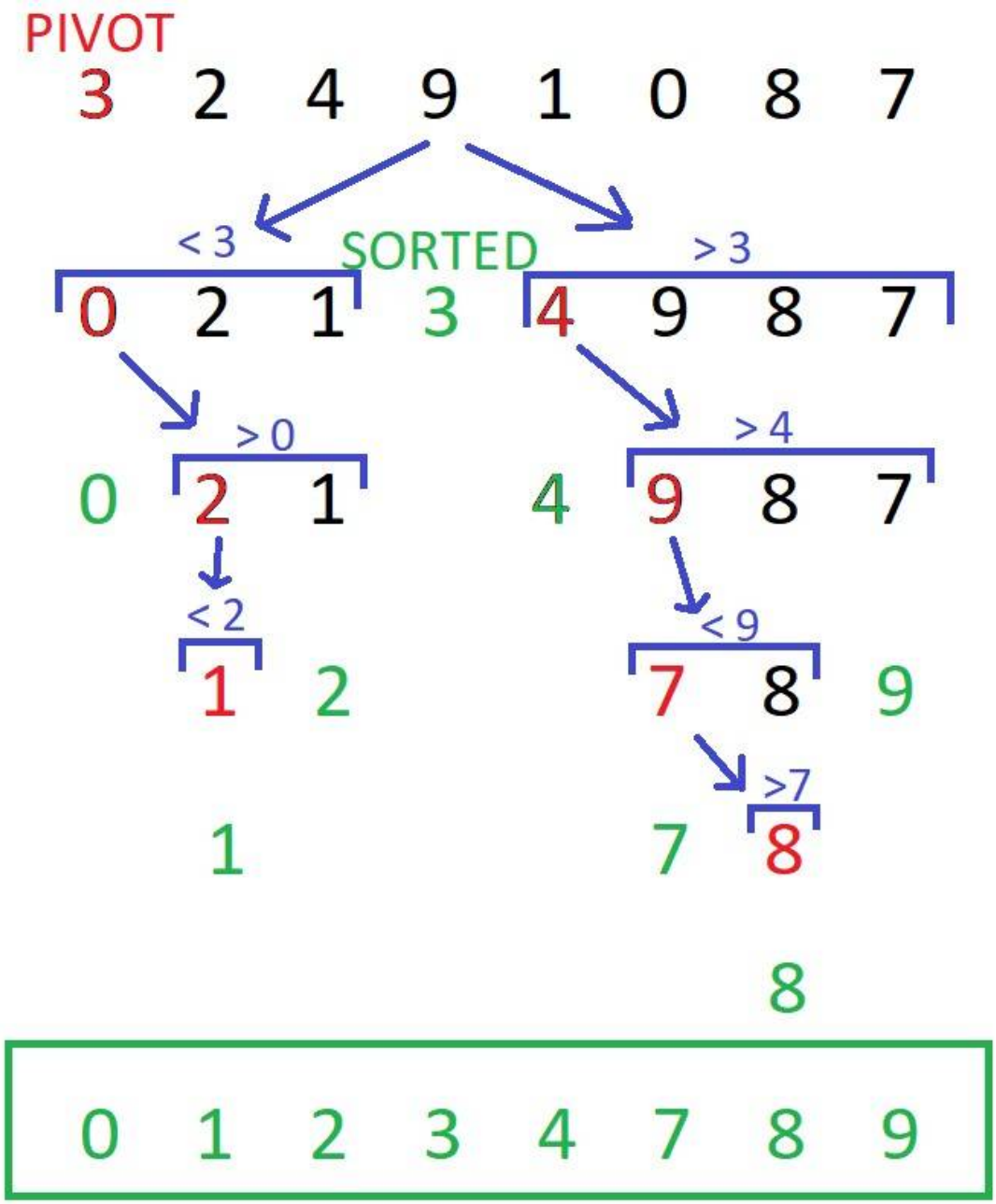
```
[“Cherries“, “Blackberries”, “Apples”, “Bananas”]
```

into alphabetical order by first letter, the output could be:

```
[“Apples”, “Bananas”, “Blackberries”, “Cherries”]
```

Quick Sort logic

Since Quick Sort is one of the less intuitive sorting algorithms, we will go over each step slowly to make things easy to follow. If it doesn’t quite mak sense after the first run-through, then go over it again and it should begin to sink in – at least that’s what I found. Stick with it!



- we start with the first element of the array (known as "the pivot") and place all of the numbers smaller than it to its left, and all the numbers larger than it to its right. The pivot will then be in its correct place.
- There is then a left sub-array, [0, 2, 1], and a right sub-array, [4, 9, 8, 7]. We now need to sort these arrays. The pivot is chosen to be the first element of each array. This process is repeated until the left and right sub-arrays are sorted and each number is in its correct place.
- **The best-case time complexity is of O(nlog(n)) – linearithmic time complexity.** The log(n) comes from the number of decompositions (divisions into subarrays) that have to be done. Then we have O(n) comparisons per decomposition.
- **The worst-case time complexity is of O(n2) – quadratic time complexity.** The worst case is when we have a sorted array and we start from the smallest or largest value; this requires n decompositions, and with n comparisons per decomposition, this results in O(n2). So, the Big O of Quick Sort is n2 – quadratic time complexity.

Quick Sort has a space complexity of O(log(n)). If implemented recursively, this is due to the stack frames that have to be stored.

Time Complexity (Best)	O(nlog(n))
Time Complexity (Average)	O(nlog(n))
Time Complexity (Worst)	O(n^2)
Space Complexity	O(log(n))
In-place/out-of-place	In-place

Time Complexity (Best)	O(nlog(n))
Stability?	Unstable
Comparison Sort?	Comparison

```
// A utility function to swap two elements
function swap(arr, i, j) {
  const temp = arr[i];
  arr[i] = arr[j];
  arr[j] = temp;
}

/* This function takes last element as pivot, places
 the pivot element at its correct position in sorted
 array, and places all smaller (smaller than pivot)
 to left of pivot and all greater elements to right
 of pivot */
function partition(arr, low, high) {

  // pivot
  const pivot = arr[high];

  // Index of smaller element and
  // indicates the right position
  // of pivot found so far
  let i = (low - 1);

  for (let j = low; j <= high - 1; j++) {

      // If current element is smaller
      // than the pivot
      if (arr[j] < pivot) {

          // Increment index of
          // smaller element
          i++;
          swap(arr, i, j);
      }
  }
  swap(arr, i + 1, high);
  return (i + 1);
}

/* The main function that implements QuickSort
        arr[] --> Array to be sorted,
        low --> Starting index,
        high --> Ending index
*/
function quickSort(arr, low, high) {
  if (low < high) {

      // pi is partitioning index, arr[p]
      // is now at right place
      const pi = partition(arr, low, high);

      // Separately sort elements before
      // partition and after partition
      quickSort(arr, low, pi - 1);
      quickSort(arr, pi + 1, high);
  }
}
```

```
// Driver Code
const arr = [10, 7, 8, 9, 1, 5];
const n = arr.length;

quickSort(arr, 0, n - 1);
console.log(arr);
```

Summary

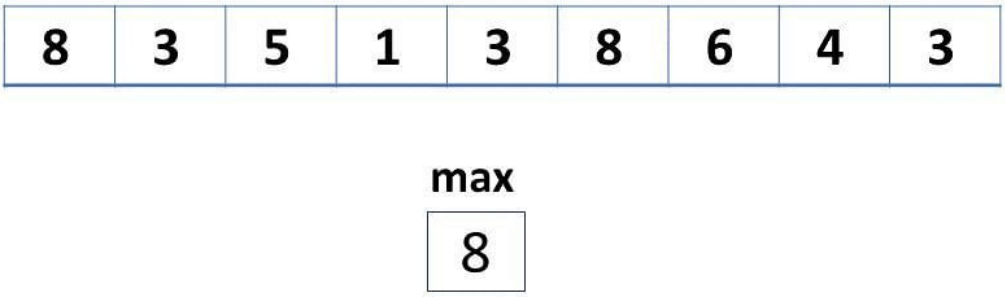
Quick Sort is very good at sorting large arrays, but Insertion Sort is often a better choice for small arrays. Knowing this, you could make a optimization: the input array length could be checked before sorting; if the array is small, perform Insertion Sort; if the array is large, perform Quick Sort.

Counting Sort

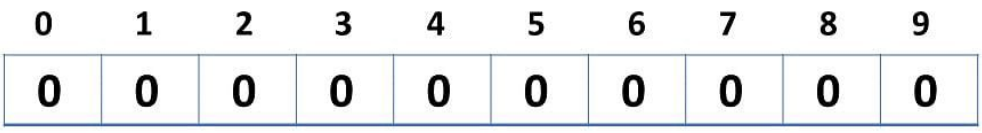
- Counting sort is an integer sorting algorithm used in computer science to collect objects according to keys that are small positive integers.
- It works by determining the positions of each key value in the output sequence by counting the number of objects with distinct key values and applying prefix sum to those counts.
- Because its running duration is proportional to the number of items and the difference between the maximum and minimum key values, it is only suited for direct usage when the number of items is not much more than the variation in keys.
- It's frequently used as a subroutine in radix sort, a more efficient sorting method for larger keys.

Iterating through the input, counting the number of times each item appears, and utilizing those counts to compute each item's index in the final, sorte array is how counting sort works.

- Consider a given array that needs to be sorted. First, you'll have to find the largest element in the array and set it to the max.



- To store the sorted data, you will now initialize a new count array with length "max+1" and all elements set to 0.

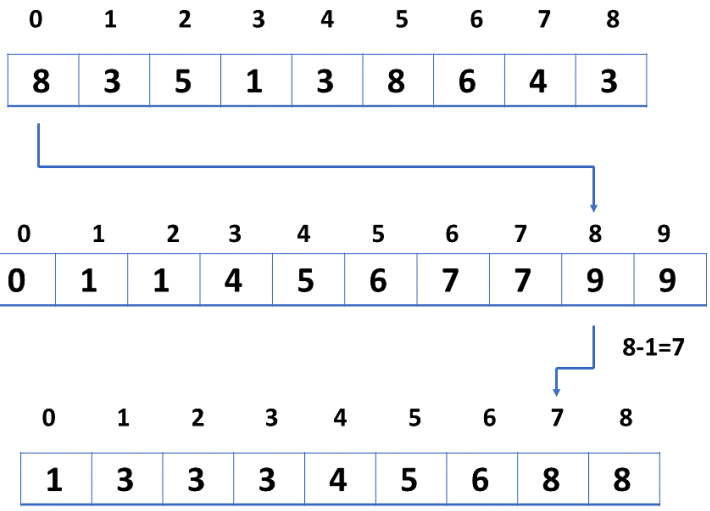


- To store the sorted data, you will now initialize a new count array with length "max+1" and all elements set to 0.

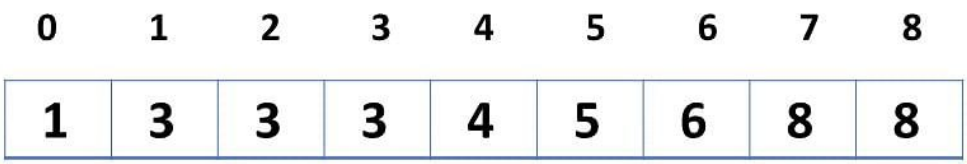
- Later, as shown in the figure, you will store elements of the given array with the corresponding index in the count array.

- Now, you will change the count array by adding the previous counts to produce the cumulative sum of an array, as shown below:

- Because the original array has nine inputs, you will create another empty array with nine places to store the sorted data, place the elements in their correct positions, and reduce the count by one.



- As a result, the sorted array is:



Complexities of Counting Sort:

Time Complexity	O(n + k)
Time Complexity (Best)	When all elements are same
Time Complexity (Average)	When n and k are equally dominant
Time Complexity (Worst)	When data is skewed and range is large
Space Complexity	O(k)
Stability?	Stable

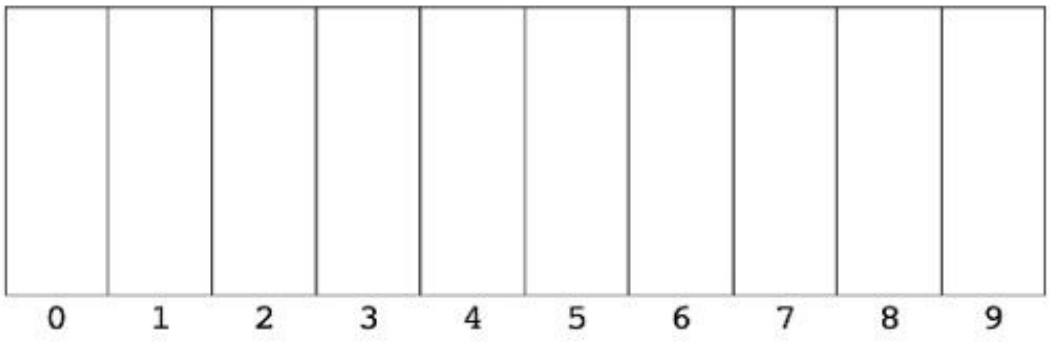
> Note: n is the number of elements and k is the range of elements(k = largest element - smallest element).

Radix Sort

Radix Sort is a unique sorting algorithm: it sorts an array without making any comparisons between elements.

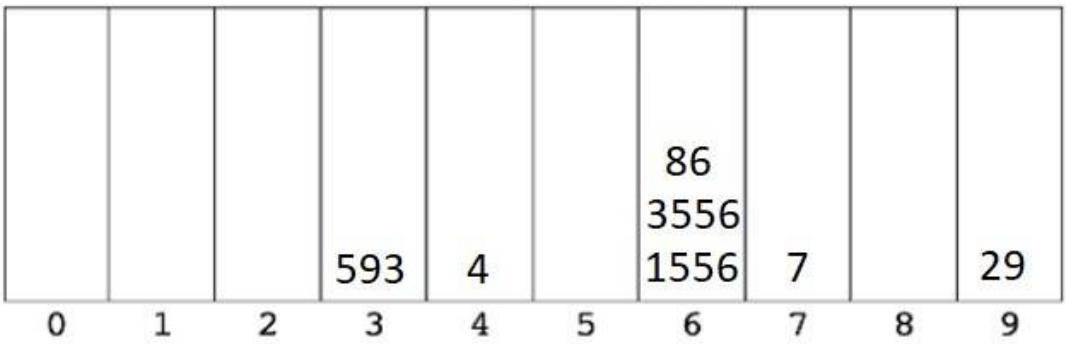
Radix Sort exploits the fact the size of a number is encoded in the number of digits – more digits means a bigger number.

Each digit in a number can be one of ten values: 0-9. So, we need to create ten buckets – one for each value:



Ten buckets for sorting base 10 numbers.

First, Radix Sort looks at the right-most digit of each number, and puts each number in the correct bucket. If we are sorting the array [1556, 4, 3556, 59 29, 86, 7], this would result in:



- Putting these numbers back in there new order we'd get [593, 4, 1556, 3556, 86, 7, 29].
- As you can see, the elements are now in order by their right-most digit.

- Radix Sort would then look at the second digit from the right, put them in the correct buckets, and reorder. Here's how it would look: [4, 7, 29, 1556, 3556, 86, 593]
- Then after looking at the third digit from the right: [4, 7, 29, 86, 1556, 3556, 593]
- And after looking at the fourth digit from the right: [4, 7, 29, 86, 593, 1556, 3556] => The array is sorted!
- **The time complexity of Radix Sort** depends on the number of digits in the longest number, k, and the length of the input array, n.
- For every loop iteration up to k, we have to loop over all of the numbers in the array; therefore, the time complexity of Radix Sort is O(k* n).
- **The space complexity of Radix Sort is O(n + d)** , where n is the length of the input array, and d is the amount of values each digit could be – in our case, 0 to 9, so d is 10.
- However, **Radix Sort is not as popular as Merge or Quick Sort due to it being more "specialized"**. You can throw integers, floats and numbers at Merge/Quick Sort and they will sort them without issue. But Radix Sort would need to be tweaked if, for example, we wanted to sort alphabetically, instead of just sorting numbers - 26 buckets to sort alphabetically, instead of 10 buckets for numbers.

O(kn)	O(kn)
Time Complexity (Average)	O(kn)
Time Complexity (Worst)	O(kn)
Space Complexity	O(d + n)
In-place/out-of-place	Out-of-place
Stability?	Unstable
Comparison Sort?	Non-comparison Sort

```
function getMax(arr,n)
{
    let mx = arr[0];
        for (let i = 1; i < n; i++)
            if (arr[i] > mx)
                mx = arr[i];
        return mx;
}

// A function to do counting sort of arr[] according to
    // the digit represented by exp.
function countSort(arr,n,exp)
{
    const output = new Array(n); // output array
        let i;
        const count = new Array(10);
        for(let i=0;i<10;i++)
            count[i]=0;

        // Store count of occurrences in count[]
        for (i = 0; i < n; i++)
            count[Math.floor(arr[i] / exp) % 10]++;

        // Change count[i] so that count[i] now contains
        // actual position of this digit in output[]
        for (i = 1; i < 10; i++)
            count[i] += count[i - 1];

        // Build the output array
        for (i = n - 1; i >= 0; i--) {
            output[count[Math.floor(arr[i] / exp) % 10] - 1] = arr[i];
            count[Math.floor(arr[i] / exp) % 10]--;
        }
```

```
        // Copy the output array to arr[], so that arr[] now
        // contains sorted numbers according to current digit
        for (i = 0; i < n; i++)
            arr[i] = output[i];
}

// The main function to that sorts arr[] of size n using
    // Radix Sort
function radixsort(arr,n)
{
    // Find the maximum number to know number of digits
        const m = getMax(arr, n);

        // Do counting sort for every digit. Note that
        // instead of passing digit number, exp is passed.
        // exp is 10^i where i is current digit number
        for (let exp = 1; Math.floor(m / exp) > 0; exp *= 10)
            countSort(arr, n, exp);
}

/* Driver Code */
const arr=[170, 45, 75, 90, 802, 24, 2, 66];
const n = arr.length;

// Function Call
radixsort(arr, n);
console.log(arr);
```

Sorting Algorithms Comparison Table

Algorithm	Time Complexity (Best)	Time Complexity (Average)	Time Complexity (Worst)	Space Complexity	Stability
Bubble Sort	O(n)	O(n^2)	O(n^2)	O(1)	Stable
Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)	Unstable
Insertion Sort	O(n)	O(n^2)	(n^2)	O(1)	Stable
Merge Sort	O(nlog(n))	O(nlog(n))	O(nlog(n))	O(n)	Stable
Quick Sort	O(nlog(n))	O(nlog(n))	O(n^2)	O(log(n))	Unstable
Radix Sort	O(kn)	O(kn)	O(kn)	O(d + n)	Stable

Algorithm	Time Complexity (Best)	Time Complexity (Average)	Time Complexity (Worst)	Space Complexity	Stability
Bubble Sort	O(n)	O(n^2)	O(n^2)	O(1)	Stable
Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)O(n^2)	Unstable
Insertion Sort	O(n)	O(n^2)	(n^2)	O(1)	Stable
Merge Sort	O(nlog(n))	O(nlog(n))	O(nlog(n))	O(n)	Stable
Quick Sort	O(nlog(n))	O(nlog(n))	O(n^2)	O(log(n))	Unstable
Radix Sort	O(kn)	O(kn)	O(kn)	O(d + n)	Stable

What is Searching algorithm?

- As a programmer, you want to find the best solution to a problem so that your code is not only correct but also efficient. Choosing a suboptimal algorithm could mean a longer completion time, increased code complexity, or worse, a program that crashes.
- You may have used a search algorithm to locate items in a collection of data. The JavaScript language has several methods, like `find`, to locate items in an array. However, these methods use linear search. A linear search algorithm starts at the beginning of a list and compares each element with the search value until it is found.

- This is fine when you have a small number of elements. But when you are searching large lists that have thousands or millions of elements, you need a better way to locate items. This is when you would use binary search.

Linear Search

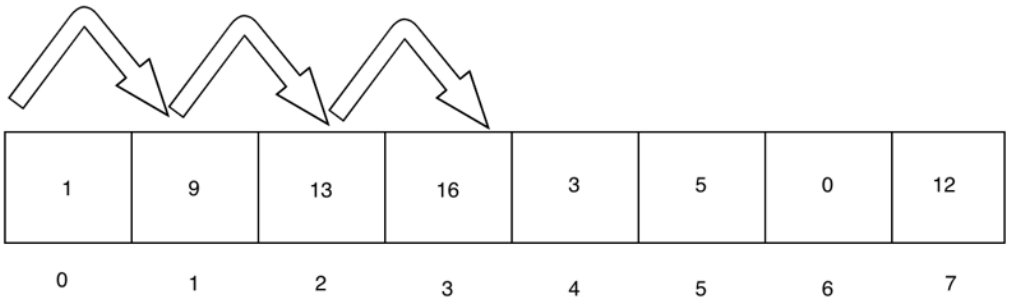
We will begin by explaining how to implement linear search in JavaScript. We will create a function called `linearSearch` that accepts a value that an integer or string and an array as parameters. The function will search every element in the array for the value and return the position of the value the array if it is found. If the value is not in the array, it will return `-1` .

Pseudocode for Linear Search

```
Set found to false
Set position to -1
Set index to 0
while found is false and index < number of elements
    if list[index] is equal to search value
        Set found to true
        Set position to index
    else Add 1 to index
return position
```

Step-by-Step Explanation of Linear Search

Imagine our input to the linear search is `[1,9,13,16,3,4,0,12]` . If the value we're searching for is `16` , the above logic would return `3` . And, if w search for `11` , the above logic will return `-1` . Let's break it down.



We initialize the algorithm by setting foundto false, positionto -1, and indexto 0. Then we iterate :

Step	index	list[index]	position	found
1	0	1	-1	false
2	1	9	-1	false
3	2	13	-1	false
4	3	16	3	true

If we follow the above logic for an element that is not present in the array, you'll see the code returning -1, since `found = false` , and `position -1` till the end.

JavaScript Implementation of Linear Search

Here is a JavaScript implementation of the linear search algorithm:

```
function linearSearch(value, list) {
    let found = false;
    let position = -1;
    let index = 0;

    while(!found && index < list.length) {
        if(list[index] == value) {
```

```
            found = true;
            position = index;
        } else {
            index += 1;
        }
    }
    return position;
}
```

Properties of Linear Search

It is important to note that the linear search algorithm does not need to use a sorted list. Also, the algorithm can be customized for use in differe scenarios like searching for an array of objects by key. If you have an array of customer data that includes keys for the first and last name, you could te if the array has a customer with a specified first name. In that case, instead of checking if `list[index]` is equal to our search value, you would chec for `list[index].first` .

Time Complexity of Linear Search

The best-case time complexity is achieved if the element searched is the first element in the list. Now, the search would complete with a sing comparison. Hence, the best-case time complexity would be **O(1)**.

The worst-case time complexity occurs if the element searched is the last element or is not present in the list. In this case, the search has to compare a elements in the array. We say that the input data has length **n**, which means the overall time complexity is **O(n)** due to the n comparisons made.

In the example above, I used the `linearSearch` function on an array with eight elements. In the worst case, when the search value is not in the list is at the end of the list, the function would have to make eight comparisons. Because our array is so small, there is no need to optimize using a differe algorithm. However, beyond a certain point, it is no longer efficient to use a linear search algorithm, and that's when using a binary search algorith would be better.

The average time complexity of linear search is also **O(n)**.

Space Complexity of Linear Search

The overall space complexity of this algorithm is equivalent to the size of the array. Hence, **O(n)**. You don't need to reserve any additional space f completing this algorithm.

Binary Search

Imagine you are playing a number guessing game. You are asked to guess a number between 1 and 100. If your number is too high or too low, you w get a hint.

What would your strategy be? Would you choose numbers randomly? Would you start with 1, then 2, and so on until you guessed correctly? Even if yo had unlimited guesses, you want to make the correct guess in as few tries as possible. Therefore, you might start by guessing 50. If the number is highe you could guess 75. If it is lower, then that means the number is between 50 and 75, and you would choose a number that's in the middle. You would g on like this until you arrived at the correct number. This is similar to how binary search works.

There are two ways of implementing binary search:

- iterative method
- recursive method

Pseudocode for Iterative Binary Search

Here's some pseudocode that expresses the binary search using the iterative method:

```
do until the low and high pointers have not met or crossed
    mid = (low + high)/2
    if (x == arr[mid])
        return mid
    else if (x > arr[mid])
        low = mid + 1
    else
        high = mid - 1
```

Pseudocode for Recursive Binary Search

Here is the pseudocode for implementing the binary search using the recursive method.

```
binarySearch(arr, x, low, high)
    if low > high
        return False
    else
        mid = (low + high) / 2
        if x == arr[mid]
            return mid
        else if x > arr[mid]
            return binarySearch(arr, x, mid + 1, high)
        else
            return binarySearch(arr, x, low, mid - 1)
```

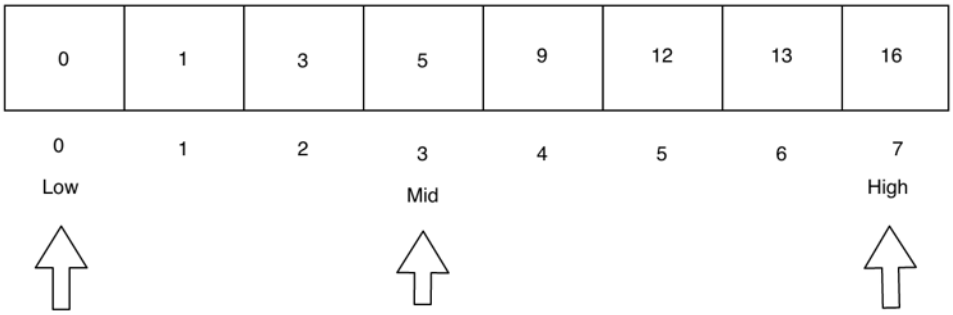
Regardless of the technique used, the binary search algorithm always uses the divide and conquer approach.

Step-by-Step Explanation

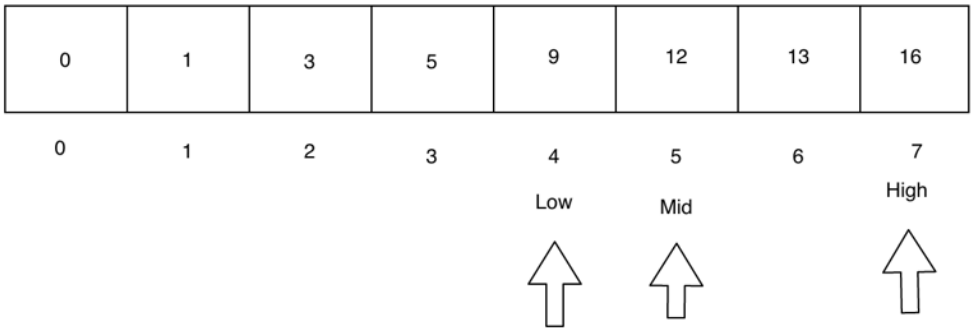
Let's consider an array `[1,9,13,16,3,5,0,12]` where the `searchValue` is `13` .

We start with a sorted array as required to perform a binary search. Note that sorting the array is expensive, but once it's done, the array can b efficiently searched many times.

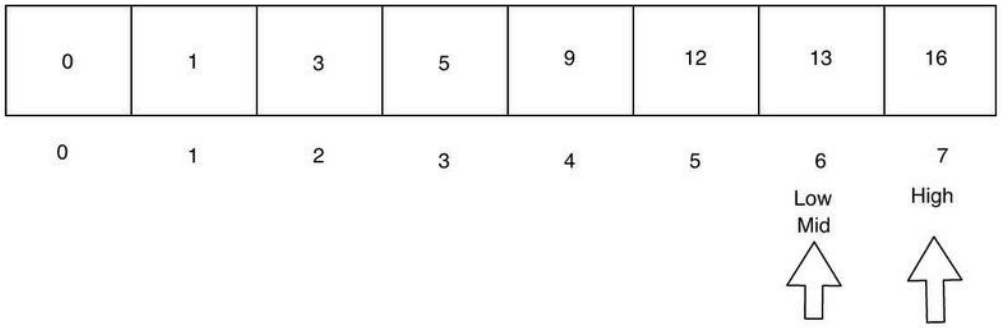
Now, the high and low pointers will be set to the first and last elements respectively. The mid pointer is set to the index given by (low - high) / 2.



Since searchValue > mid, we need to search the right side of the array. So we set the lowpointer to be just after mid, and midis reset to be betwee the lowand highpointers.



Again, the target value is towards the right of the array. Once again, we adjust the low and high pointers. Now the low and mid pointers are the same.



Now, the searchValueis found at mid, which means we have reached the end of our search!

Step	low	mid	high	list[mid]
1	0	3	7	5
2	4	5	7	9
3	6	6	7	13

JavaScript Implementation of Binary Search

Now let's code the binary search algorithm in JavaScript!

We'll create a function, `binarySearch` , that accepts a value and an array as parameters. It will return the index where the value occurs in the list found. If the value is not found, it returns -1. This is our implementation written in JavaScript:

```
//note that list must be sorted for this function to work
function binarySearch(value, list) {
    let low = 0;    //left endpoint
    let high = list.length - 1;   //right endpoint
    let position = -1;
    let found = false;
    let mid;

    while (found === false && low <= high) {
        mid = Math.floor((low + high)/2);
        if (list[mid] == value) {
            found = true;
            position = mid;
        } else if (list[mid] > value) {  //if in lower half
            high = mid - 1;
        } else {  //in in upper half
```

```
                low = mid + 1;
            }
        }
        return position;
    }
```

Time Complexity

One of the main reasons why we use binary search for finding elements in an array would be its time complexity. In a best-case scenario, the tim complexity for Binary Search is **O(1)**. This happens when the element in the middle of the array matches the search element.

At worst, the time complexity for searching an element using binary search is **O(log n)**—much less than **O(n)** for large values of n. To get an idea of ho much slower growing **log(n)** is than **n**, here is a table of typical values of **log(n)**.

n	n
1	1
8	3
1024	10
1,000,000	19.9
1,000,000,000,000,000,000	59.8

So, as you can see, the bigger n gets, the more the improvement of a binary search over a linear search.

The average ****case time complexity is also **O(log n)**, for searching an item using binary search.

Space Complexity

The space complexity for implementing the binary search is again **O(n)**.

Properties of Binary Search

Unlike linear search, binary search uses a sorted list. That lets us use a "divide and conquer" algorithm to solve the problem.

Conclusion

- we saw how to implement a linear search and a binary search algorithm. The linear search algorithm is simpler and doesn't require a sorted array. However, it is inefficient to use with larger arrays. In the worst case, the algorithm would have to search all elements making n comparisons (where n is the number of elements).
- The binary search algorithm, on the other hand, requires you to sort the array first and is more complicated to implement. However, it is more efficient even when considering the cost of sorting. For example, an array with 10 elements would make at most 4 comparisons for a binary search vs. 10 for a linear search—not such a big improvement. However, for an array with 1,000,000 elements, the worst case in binary search is only 20 comparisons. That's a huge improvement over linear search!

Thank You !