Chapter 5

**Searching Algorithms**

Linear search, Binary search & String search

**5.1 Objectives**

1. *Describe* what a *searching algorithm* is
2. Implement Linear *search* on arrays
3. Implement Binary *search* on sorted arrays
4. Implement a NaiveString *searching* algorithm
5. Implement the KMPString *searching* algorithm

* Search a string from a list of string. JS built-in function is **list.indexOf("string")**. But we're gonna built one of our own.
* Our goal: We are going to search a string inside an array of string. In JS there is indexOf() method to do this kind of job. And indexOf() uses a searching algorithm.
* We are gonna build this kind of functions to search arrays and strings.
* Usage: It is useful to search database. To find a user with a specific user name. Or for a "valid state" to check if an user inputs a valid states in USA.



USA\_states = ["Alabama", "Alaska", "American Samoa", "Arizona", "Arkansas", "California", "Colorado", "Connecticut", "Delaware", "District of Columbia", "Federated States of Micronesia", "Florida", "Georgia", "Guam", "Hawaii", "Idaho", "Illinois", "Indiana", "Iowa", "Kansas", "Kentucky", "Louisiana", "Maine", "Marshall Islands", "Maryland", "Massachusetts", "Michigan", "Minnesota", "Mississippi", "Missouri", "Montana", "Nebraska", "Nevada", "New Hampshire", "New Jersey", "New Mexico", "New York", "North Carolina", "North Dakota", "Northe rn Mariana Islands", "Ohio", "Oklahoma", "Oregon", "Palau", "Pennsylvania", "Puerto Rico", "Rhode Island", "South Carolina", "South Dakota", "Tennessee", "Texas", "Utah", "Vermont", "Virgin Island", "Virginia", "Washington", "West Virginia", "Wisconsin", "Wyoming"];

console.**log**(USA\_states);

**5.2 Linear Search**

* How do we search? Normally we look at every element of an array and we check if it's the value we want. We can use this if we have an un-sorted array/data.
* But in case of sorted data, (alphabetic or numeric), we have a better approach.

/\* sorted array \*/

USA\_states = ["Alabama", "Alaska", "American Samoa", "Arizona", "Arkansas", "California", "Colorado", "Connecticut", "Delaware", "District of Columbia", "Federated States of Micronesia", "Florida", "Georgia", "Guam", "Hawaii", "Idaho", "Illinois", "Indiana", "Iowa", "Kansas", "Kentucky", "Louisiana", "Maine", "Marshall Islands", "Maryland", "Massachusetts", "Michigan", "Minnesota", "Mississippi", "Missouri", "Montana", "Nebraska", "Nevada", "New Hampshire", "New Jersey", "New Mexico", "New York", "North Carolina", "North Dakota", "Northe rn Mariana Islands", "Ohio", "Oklahoma", "Oregon", "Palau", "Pennsylvania", "Puerto Rico", "Rhode Island", "South Carolina", "South Dakota", "Tennessee", "Texas", "Utah", "Vermont", "Virgin Island", "Virginia", "Washington", "West Virginia", "Wisconsin", "Wyoming"];

console.**log**(USA\_states);

/\* unsorted array \*/

rand\_names = ["tommy", "monkeygurl", "dineshl23", "dineshl234", "patrick33", "cats4lyfe", "timothy", "tom", "tommyl", "tomm\_y", "tom\_my","i\_hate\_cats", "pickle\_luvr", "pickle\_hater", "dog\_guy", "q", "juan987"]

console.**log**(rand\_names);

* Linear search: Checking each & every element of an array is called "***linear-search***". Following JS functions uses ***Linear Search***, i.e. following functions looks each & every elements of an array.
* There are many different search methods on arrays in JavaScript:

1. indexOf
2. includes
3. find
4. findlndex

* Either we *start from beginning* and moving *forward* or *start from the end* and moving *backward*. And in both case we look at *each element* of the *array*. It's the usual *search technique* that we use in our *everyday life*.
* Linear Search Pseudocode:

1. This function accepts an *array* and a *value*
2. Loop through the array and *check* if the *current array element* is equal to the *value*
3. If it is, *return* the *index* at which the element is found
4. If the value is never found, *return -1*

function **linearSearch**(arr, val){

    for(var i = 0; i < arr.length; i++){

        if(arr[i] === val) return i;

    }

    return -1;

}

**linearSearch**([34,51,1,2,3,45,56,687], 100)

**===** means equal value and equal type

**Practiced Version**

function **is\_in\_arr**(arr, val){

    for(var i = 0; i < arr.length; i++){

        if(arr[i]===val) return i;

        // else return -1; // it cause error, do not use this, ends the loop immediately

    }

    return -1;

}

Array could be array of strings.

/\* *sorted array* \*/

USA\_states = ["Alabama", "Alaska", "American Samoa", "Arizona", "Arkansas", "California", "Colorado", "Connecticut", "Delaware", "District of Columbia", "Federated States of Micronesia", "Florida", "Georgia", "Guam", "Hawaii", "Idaho", "Illinois", "Indiana", "Iowa", "Kansas", "Kentucky", "Louisiana", "Maine", "Marshall Islands", "Maryland", "Massachusetts", "Michigan", "Minnesota", "Mississippi", "Missouri", "Montana", "Nebraska", "Nevada", "New Hampshire", "New Jersey", "New Mexico", "New York", "North Carolina", "North Dakota", "Northe rn Mariana Islands", "Ohio", "Oklahoma", "Oregon", "Palau", "Pennsylvania", "Puerto Rico", "Rhode Island", "South Carolina", "South Dakota", "Tennessee", "Texas", "Utah", "Vermont", "Virgin Island", "Virginia", "Washington", "West Virginia", "Wisconsin", "Wyoming"];

console**.**log(USA\_states);

/\* *unsorted array* \*/

rand\_names = ["tommy", "monkeygurl", "dineshl23", "dineshl234", "patrick33", "cats4lyfe", "timothy", "tom", "tommyl", "tomm\_y", "tom\_my","i\_hate\_cats", "pickle\_luvr", "pickle\_hater", "dog\_guy", "q", "juan987"]

console**.**log(rand\_names);

function is\_in\_arr(arr, val){

    for(var i = 0; i < arr**.**length; i++){

        if(arr[i]===val) return i;

        // *else return -1; // it cause error, do not use this*

    }

    return -1;

}

console**.**log(is\_in\_arr(USA\_states, "Idaho"));

console**.**log(is\_in\_arr(rand\_names, "tom"));

* Big – O of linear search: O(n). It is linear.



* Best Case: we find the match at the beginning of the array.
* Average Case: we find the match in the middle of the array before the last element.
* Worst Case: we find the match at the end of the array.
* But in general it's complexity is **O(n)**. It works best when we're dealing with "Unsorted-Array".

**5.3 Binary Search**

* Binary search is a much faster form of search
* Rather than eliminating one element at a time, you can eliminate half of the remaining elements at a time
* Binary search only works on sorted arrays!
* So the data has to be sorted. If it's numbers lowest to highest or highest to lowest or strings alphabetical. There has to be an order to it.
* Suppose we want to find: "Oregon".
* We pick a *mid point* and compare with the given value,
* then we *eliminate* the *half* of the array.

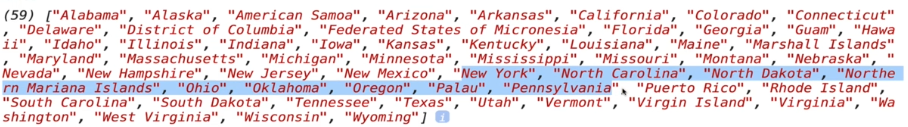


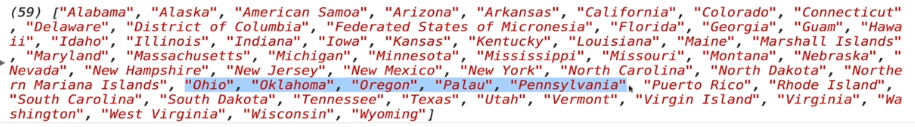
* We eliminate the first portion.

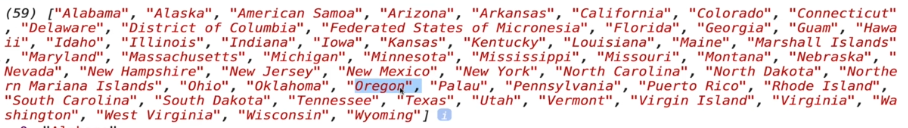


* We repeat the process, and the selected portion of the array gets small.



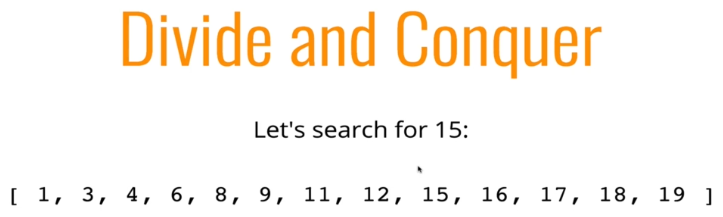




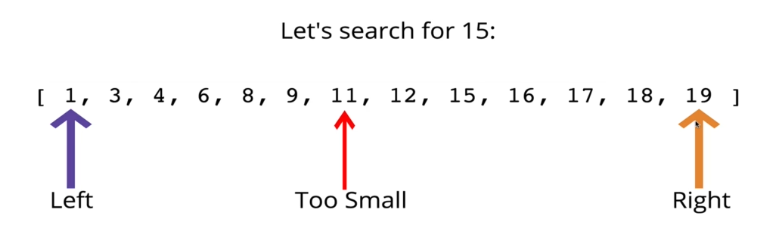


* Idea of the implementation: The idea is dividing and conquering.

1. So we *split* *up* the array into *two pieces*.
2. We pick a *pivot point* in the middle usually and we check the *left side* and the *right side* and see where piece we're looking for.



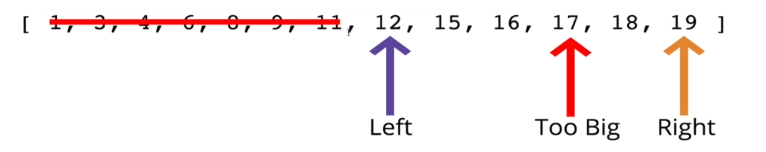
* Example: So let's search for 15 in this list. Remember it has to be sorted. This doesn't work at all. If it's not sorted. We're going to call three points left and right and mid-point.



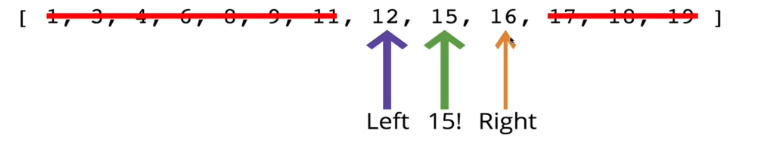
We're going to *change* the *window* of where we're searching. We can throw out *half of the array* roughly every time.

We're going to start with the ***left*** at ***1*** and ***right*** at the ***end*** ***19*** and then we're going to pick somewhere in the middle. If there is fraction we can round it.

* We're going to pick 11 here as mid point.
* We check is 15 greater than 11 or less than 11.
* Since 15 is greater than 11, we eliminate all of that to the left and we make the new left 12. And the new right is 19.
* Again we set the mid pivot point, which is 17.
* We check is 15 greater than or less than 17.
* Since 15 is less than 17, we can eliminate 17 18 19 (right side).



* Now we have left at 12 right at 16 and then the middle is 15 which is the only choice and 15 equal to 15.
* So it only took three iterations. And much faster than linear search it would took 9 - iterations.



* If we have a Sorted Array, numbers or strings or whatever it is as long as we can *compare* easily to check if something is *greater* than or *less* than, we can implement Binary Search which is much faster.
* Binary Search Pseudocode:

1. This function *accepts* a *sorted array* and a value
2. Create a *left pointer* at the *start* of the array, and a *right pointer* at the *end* of the array
3. While the left pointer comes before the right pointer:
   1. Create a *pointer* in the *middle*
   2. If you find the value you want, *return* the *index*
   3. If the value is *too small*, move the *left* pointer *up*
   4. If the value is *too large*, move the *right* pointer *down*
4. If you never find the value, return -1.

**Practiced version**

// *Practiced version*

**function** **binarySearch**(arr, elem){

**var** start=0;

**var** end=arr**.**length-1;

**var** mid=Math**.floor**((start+end)/2);

**while**((arr[mid] **!=** elem) **&&** (start **<=** end)){

**if**(elem **>** arr[mid]){

            start = mid + 1;

        } **else**{

            end = mid -1;

        }

        mid = Math**.floor**((start+end)/2);

    }

**if**(arr[mid]**===**elem) **return** mid;

**return** -1;

}

* Notice: in the while loop in the joint condition ((arr[mid] **!=** elem) **&&** (start **<=** end)), **(start <= end)** is used to exit the while loop, unless we end up with a infinite-loop. The loop ends when **(start <= end)** is false.
* Also if we found the value, **(arr[mid] != elem)** gets false and it ends the while-loop.
* Original solution:

// *Original Solution*

**function** **binarySearch**(arr, elem) {

**var** start=0;

**var** end=arr**.**length-1;

**var** middle=Math**.floor**((start+end)/2);

**while**(arr[middle] **!==** elem **&&** start **<=** end) {

**if**(elem **<** arr[middle]){

            end = middle - 1;

        } **else** {

            start = middle + 1;

        }

        middle = Math**.floor**((start + end) / 2);

    }

**if**(arr[middle] **===** elem){

**return** middle;

    }

**return** -1;

}

**Refactored solution**

// Refactored Version

function **binarySearch**(arr, elem) {

    var start = 0;

    var end = arr.length - 1;

    var middle = Math.**floor**((start + end) / 2);

    while(arr[middle] !== elem && start <= end) {

        if(elem < arr[middle]) end = middle - 1;

        else start = middle + 1;

        middle = Math.**floor**((start + end) / 2);

    }

    return arr[middle] === elem ? middle : -1;

}

**binarySearch**([2,5,6,9,13,15,28,30], 103)

* elem ? middle : -1; uses the ternary conditional operator. It is a alternative to if-else statement.

Binary Search BIG O:



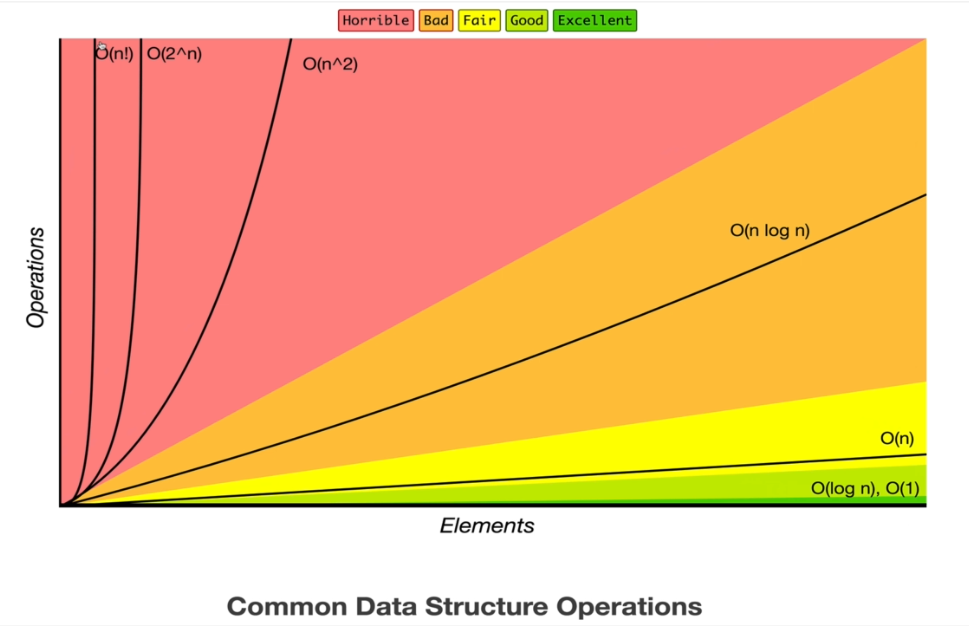
|  |  |
| --- | --- |
| * Now 4 steps is the worst case for 16 elements. * If we search for 9, it would be 2 step. |  |

After we double the size of the array, it took 5-steps.

**Time-complexity is actually**

Notice in our above 2 cases: = worst case for 16 element.

= worst case for 32 element.



* Note: Notice that, is nearly good as **O(n)**, so it is one of the best case.

**5.4 Naive String Search**

Searching for substrings in a larger string.

* First we implement naive string search.
* Then we're going to see a better implementation.
* Naive String Search:
* Suppose you want to *count* the *number of times* a *smaller* string appears in a *longer* string
* A straightforward approach involves checking *pairs of characters* individually

|  |  |
| --- | --- |
| * Firstly we if we want to find a single character, we can use "***Linear Search***". * Searching for a pattern Pseudocode:  1. Loop over the longer string 2. Loop over the shorter string 3. If the characters don't match, break out of the inner loop 4. If the characters do match, keep going 5. If you complete the inner loop and find a match, increment the count of matches 6. Return the count |  |
| * short[j] !== long[i+j] Checks if the string characters match. * j === short.length – 1 Checks the *small-string* is appeared in the *longer-string*, it checks if the *index* of *small-string* is reached to the end. | function **naiveSearch**(long, short){      var count = 0;      for(var i = 0; i < long.length; i++){          for(var j = 0; j < short.length; j++){             if(short[j] !== long[i+j]) break;             if(j === short.length - 1) count++;          }      }      return count;  }  **naiveSearch**("lorie loled", "lol") |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |

**How we built the code:**

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
| Adding ***count*** variable |  |

// Walkthrough version

function **naiveSearch**(long, short){

    var count = 0;

    for(var i = 0; i < long.length; i++){

        for(var j = 0; j < short.length; j++){

            console.**log**(short[j], long[i+j])

            if(short[j] !== long[i+j]){

                console.**log**('BREAK!');

                break;

            }

            if(j === short.length - 1){

                console.**log**("FOUND ONE!");

                count++;

            }

        }

    }

    return count;

}

**naiveSearch**("lorie loled", "lol")

**5.5 KMP (Knuth\_Morris\_Pratt) String searching algorithm**

// *Knuth-Morris-Pratt algorithm*

// *Write a function which accepts a string and a pattern, and returns the index*

// *at which the value exists. If the pattern does not exist in the string, return -1.*

// *Time Complexity - O(n + m),* *Space complexity - O(n)*

**function** **buildArrayPattern**(pattern) {

**const** arrayPattern=[0];

**let** prefix=0;

**let** suffix=1;

**while** (arrayPattern**.**length **<** pattern**.**length) {

**if** (pattern[prefix] **===** pattern[suffix]) {

      arrayPattern**.push**(prefix + 1);

      prefix++;

      suffix++;

    } **else** **if** (prefix **===** 0) {

      arrayPattern**.push**(0);

      suffix++;

    } **else** {

      prefix = arrayPattern[prefix - 1];

    }

  }

**return** arrayPattern;

}

**function** **kmp**(text, pattern) {

**const** arrayPattern= **buildArrayPattern**(pattern);

**let** textIndex=0;

**let** patternIndex=0;

**while** (textIndex **<** text**.**length) {

**if** (text[textIndex] **===** pattern[patternIndex]) {

      if (patternIndex === pattern.length - 1) {

        return textIndex - pattern.length + 1;

      } else {

        textIndex++;

        patternIndex++;

      }

    } else if (patternIndex === 0) {

      textIndex++;

    } else {

      patternIndex = arrayPattern[patternIndex - 1];

    }

  }

  return -1;

}

console.log(kmp('dabcdabcyabkglabcdcxabcdabcdabcy', 'abcdabcy')); // 1

console.log(kmp('dabcdabcyabkglabcdcxabcdabcdabcy', 'abcdabcyd')); // -1

function kmpCounter(text, pattern) {

  const arrayPattern = buildArrayPattern(pattern);

  let textIndex = 0;

  let patternIndex = 0;

  let counter = 0;

  while (textIndex < text.length) {

    if (text[textIndex] === pattern[patternIndex]) {

      if (patternIndex === pattern.length - 1) {

        textIndex++;

        patternIndex = 0;

        counter++;

      } else {

        textIndex++;

        patternIndex++;

      }

    } else if (patternIndex === 0) {

      textIndex++;

    } else {

      patternIndex = arrayPattern[patternIndex - 1];

    }

  }

  return counter;

}

console.log(kmpCounter('dabcdabcyabkglabcdcxabcdabcdabcy', 'abcdabcy')); // 2

console.log(kmpCounter('dabcdabcyabkglabcdcxabcdabcdabcy', 'abcdabcyd')); // 0

* Description: Given a text ***txt[0..n-1]*** and a pattern ***pat[0..m-1]***, write a function ***search(char pat[], char txt[])*** that prints all occurrences of ***pat[]*** in ***txt[]***. You may assume that ***n > m***.
* Examples:

|  |  |
| --- | --- |
| Input: txt[] = "THIS IS A TEST TEXT"  pat[] = "TEST"  Output: Pattern found at index 10  Input: txt[] = "AABAACAADAABAABA"  pat[] = "AABA"  Output: Pattern found at index 0  Pattern found at index 9  Pattern found at index 12 | C:\Users\Rumaan\Pictures\PatternSearching21-300x164.png |