Searching Algorithms

Objectives

- Describe what a searching algorithm is
- Implement linear search on arrays
- Implement binary search on sorted arrays
- Implement a naive string searching algorithm
- Implement the KMP string searching algorithm

How do we search?

Given an array, the simplest way to search for an value is to look at every element in the array and check if it's the value we want.

JavaScript has search!

There are many different search methods on arrays in JavaScript:

- indexOf
- includes
- find
- findIndex

But how do these functions work?

Linear Search

Let's search for 12:

```
[ 5, 8, 1, 100, 12, 3, 12 ]
```

12!

Linear Search Pseudocode

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

Linear Search

BIG O

O(1)	O(n)	O(n)
Best	Average	Worst

YOUR TURN

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!

Divide and Conquer

Let's search for 15:

[~~1, 3, 4, 6, 8, 9, 11~~, 12, 15, 16, ~~17, 18, 19~~]

Binary Search Pseudocode

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

YOUR TURN

NOW LET'S DO IT RECURSIVELY!

WHAT ABOUT BIG O?

O(log n)

Worst and Average Case

O(1)

Best Case

Suppose we're searching for 13

[2,4,5,9,11,14,15,19,21,25,28,30,50,52,60,63]

[2,4,5,9,11,14,15]

[11,14,15]

[11]

NOPE, NOT HERE!

16 elements = 4 "steps"

To add another "step", we need to double the number of elements

Let's search for 32

[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,32,35]

[~~1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16~~,17,18,19,20,21,22,23,24,25,26,27,28,29,30,32,35]

[~~1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24~~,25,26,27,28,29,30,32,35]

[~~1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28~~,29,30,32,35]

[~~1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30~~,32,35]

[~~1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30~~,32,35]

32 elements = 5 "steps" (worst case)

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

Naive String Search

Long string:

wowomgzomg

Short string:

omg

wowomgzomg

Pseudocode

- Loop over the longer string
- Loop over the shorter string
- If the characters don't match, break out of the inner loop
- If the characters do match, keep going
- If you complete the inner loop and find a match, increment the count of matches
- Return the count

YOUR TURN

KMP String Search

- The Knutt-Morris-Pratt algorithm offers an improvement over the naive approach
- Published in 1977
- This algorithm more intelligently traverses the longer string to reduce the amount of redundant searching

KMP Example

Long string:

`lolomglololroflol`

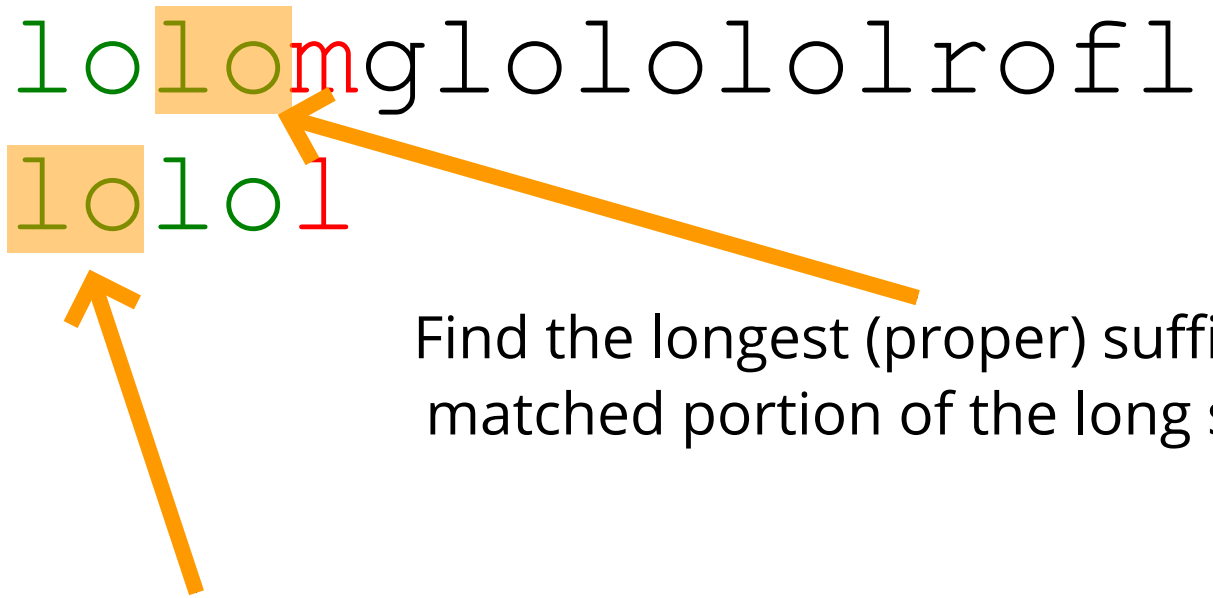
Short string:

`lolol`

`lolomglolololrofl`

KMP - WTF

How do you know how far to traverse?



Prefixes and Suffixes

- In order to determine how far we can shift the shorter string, we can *pre-compute* the length of the longest (proper) suffix that matches a (proper) prefix
- This tabulation should happen before you start looking for the short string in the long string

Prefixes and Suffixes

Example

Short string	1	o	1	o	1
Longest prefix suffix match	0	0	1	2	3

Prefixes and Suffixes

Another Example

a	b	a	c	a	b	a	b	d	a
0	0	1	0	1	2	3	2	0	1

Building the Table

```
function matchTable(word) {
  let arr = Array.from({ length: word.length }).fill(0);
  let suffixEnd = 1;
  let prefixEnd = 0;
  while (suffixEnd < word.length) {
    if (word[suffixEnd] === word[prefixEnd]) {
      // we can build a longer prefix based on this suffix
      // store the length of this longest prefix
      // move prefixEnd and suffixEnd
      prefixEnd += 1;
      arr[suffixEnd] = prefixEnd;
      suffixEnd += 1;
    } else if (word[suffixEnd] !== word[prefixEnd] && prefixEnd !== 0) {
      // there's a mismatch, so we can't build a larger prefix
      // move the prefixEnd to the position of the next largest prefix
      prefixEnd = arr[prefixEnd - 1];
    } else {
      // we can't build a proper prefix with any of the proper suffixes
      // let's move on
      arr[suffixEnd] = 0;
      suffixEnd += 1;
    }
  }
  return arr;
}
```

KMP - FTW!

```
function kmpSearch(long, short) {
  let table = matchTable(short);
  let shortIdx = 0;
  let longIdx = 0;
  let count = 0;
  while (longIdx < long.length - short.length + shortIdx + 1) {
    if (short[shortIdx] !== long[longIdx]) {
      // we found a mismatch :(
      // if we just started searching the short, move the long pointer
      // otherwise, move the short pointer to the end of the next potential prefix
      // that will lead to a match
      if (shortIdx === 0) longIdx += 1;
      else shortIdx = table[shortIdx - 1];
    } else {
      // we found a match! shift both pointers
      shortIdx += 1;
      longIdx += 1;
      // then check to see if we've found the substring in the large string
      if (shortIdx === short.length) {
        // we found a substring! increment the count
        // then move the short pointer to the end of the next potential prefix
        count++;
        shortIdx = table[shortIdx - 1];
      }
    }
  }
  return count;
}
```

Big O of Search Algorithms

Linear Search - **O(n)**

Binary Search - **O(log n)**

Naive String Search - **O(nm)**

KMP - **O(n + m)** time, **O(m)** space

Recap

- Searching is a very common task that we often take for granted
- When searching through an unsorted collection, linear search is the best we can do
- When searching through a sorted collection, we can find things very quickly with binary search
- KMP provides a linear time algorithm for searches in strings