Knuth Morris Pratt (KMP)

In 1977, Donald Knuth, Vaughan Pratt, and James Morris published *Fast Pattern Matching In Strings* using observations found in the pattern **m** to make fast comparisons. KMP work similarly to brute-force by starting comparisons from the front of text **n** and pattern **m**. Specifically, we have **i** and **j** to index into **n** and **m** respectively and keep track of what characters we're comparing.

In KMP, when there is a mismatched character **n[i] != m[j]**, instead of shifting our pattern by one and start matching the pattern over again as in brute force, we tell ourselves "I want to keep **i** where it's at in **n[i]**. How can I shift my pattern to keep matching at **n[i]**?"

In the below example, we mismatch at B != D

n =	A	B	A	B	A	B	A	D		
m =	A	B	A	B	A	D				

I want to keep comparing at the B in **n**, so let's shift our pattern some amount.

n =	A	B	A	B	A	B	A	D		
m =	A	B	A	B	A	D				
		A	B	A	B	A	D			

When we shift over our pattern once, we end up trying to compare B and A. We cannot continue our comparisons here because characters before B in **n** do not match up with the characters before A in **m**. BABA != ABAB. Let's keep shifting.

n =	A	B	A	B	A	B	A	D		
m =	A	B	A	B	A	D				
		A	B	A	B	A	D			
			A	B	A	B	A	D		

Here we can attempt to compare B in **n** and B in **m**. In this case we do have a match, which is good. What's more important are the characters before the two B's.

n =	A	B	**A**	**B**	**A**	B	A	D		
m =	A	B	A	B	A	D				
		A	B	A	B	A	D			
			A	**B**	**A**	B	A	D		

We can see that the three characters before our B's are exactly the same. Because we know that these two substrings are identical, we don't have to compare anything before our **n[i]** (which we promised wouldn't move backwards.

The pattern shifting rule is this: if we have a mismatch where **n[i]** != **m[j]**, we will shift over **m** such

- The first k characters in **m** match the k characters before **n[i]**. **m[0 .. k-1]** == **n[i-k ... i-1]** for the largest k possible. We match the longest prefix in **m** to the longest suffix in **n[0...i]**.

Failure Table

For a mismatch between **n[i]** and **m[j],** the failure table tells us how much we should shift our pattern by to have a prefix and suffix alignment. We use a pre-processed table to calculate these values. The table has an entry for every character in **m**. The value we store at an **m[i],** where i {0 … m.length-1}, is the length of the longest prefix **m[0 … k-1]** that matches with **m[i - k - 1 … i].** The first character always has a value of 0.

A_0	B_1	A_2	B_3	A_4	D_5
0	0	1	2	3	0

```
Algorithm failureTable(pattern)
  FTable[0] ← 0
  i ← 1
  j ← 0
  while i < m
    // we have matched j + 1 chars
    if pattern[i] = pattern[j] then
       FTable[i] ← j + 1
       i ← i + 1
       j ← j + 1
    // use failure function to shift pattern
    else if j > 0 then
      j ← FTable[j - 1]
    // no prefix match
    else
      FTable[i] ← 0
      i ← i + 1
  return FTable[]
```

KMP

```
Algorithm KMPMatch(T, P)
  F ← failureFunction(P)
  i ← 0
  j ← 0
  while i < n
    if T[i] = P[j]
      if j = m - 1
        return i - j // match
      else
      i ← i + 1
      j ← j + 1
    else if j > 0
      j ← F[j - 1]
    else
      i ← i + 1
  return -1 // no match
```

This algorithm runs in **O(n + m)**. This is because our **n[i]** never goes backwards in our comparisons. It will only go forward, which will cover the length of **n**. The while loop only does two things.

1. Increases **i** s.t. our **n.**.
2. Shifts our pattern over by at least 1.

Our while loop will run at most **2n** iterations, so we **O(n)** for the loop. The failure table is calculated in **O(m)**, so we get **O(n + m).**

A	**B**	**A**	**C**	**A**	**B**	**A**	**B**	**A**	**A**	**B**	**A**	**A**	**B**	**A**	**C**	**A**	**B**	**B**	**B**			
A	B	A	**A**	B	A	C	A	B														
		A	**B**	A	A	B	A	C	A	B												
			A	B	A	A	B	A	C	A	B											
				A	B	A	**A**	B	A	C	A	B										
						A	B	A	A	B	A	C	A	B								
									A	B	A	A	B	A	C	A	B					