String Searching

String searching is a classical problem in computer science. You're given a long string text (referred to as n) and a string pattern (referred to as m) and you want to find the pattern inside of the text. This text and pattern is comprised of characters from a finite set (the alphabet Σ). In our examples, we will use the English alphabet as our set of characters, but other alphabets include the binary alphabet (Σ = {0, 1}) and DNA bases (Σ = {A, C, T, G}).

When performing string search, the cost of our algorithm is the sum of character comparisons. Because the algorithm running time factors in both the sizes of our text and pattern, we include both **n** and **m** in our Big-O analysis: O(nm), O(n + m), etc.

String searching is used in DNA sequencing, text editor, online searching, etc.

Brute Force String Search

The "Brute Force" method is the naive method of string searching. Given your text **n** and pattern **m**, you align the beginning of n and m and see if **n[0] == m[0]**. If they match, you then check **n[1] == m[1]**, then **n[2] == m[2]**, and so on.

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

If there is a mismatch (**n[i] != m[i]**), then you shift m over by one character so now the compare **n[1] == m[0].**

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

You continue these search patterns until you find your string. **Perform the rest of the brute force string search algorithm below. Mark characters that matched.**

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

This runs in worst case **O(mn)** given such a case:

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

We would compare the entire pattern every shift.

Boyer-Moore

In 1977, Robert Boyer and J Moore published *A Fast String Searching Algorithm* with some key intuition on string searching:

1. Given a mismatch at **n[i]** and **m[j],** if **n[i]** does not appear in **m**, then one can shift all of **m** passed **n[i]**.
2. If **n[i]** does appear in **m** as character **c,** align **n[i]** with the last occurrence of **c** in **m**. If you have already passed **c**, shift over by 1.
3. To have the above work, start comparisons at the end of the pattern instead of the beginning.

Both of these intuitions allow us to skip many more unnecessary comparisons. The first intuition states that if a character we're matching with doesn't appear in our pattern, there's no point in shifting our pattern over by 1 if there's a mismatch. Because shifting over by 1 will yield us another mismatch with this character, we can completely shift over this character. E.g.

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

In the above case we mismatch with D. Because D doesn't appear in **m**, we can completely shift **m** passed D.

The second intuition states that if we mismatch with a character **c** in the text and **c** appears earlier in our pattern, we can align **c** with the **last occurance** of the same character in our pattern.

n =	A		P	A	**T**	T	E	R	N
m =	R	I	**T**	H	M				
			R	I	T	H	M		

In the above case we mismatch with T in our text and M in our pattern. We do have a last occurrence of T earlier in our pattern, so we can align the two T's with each other. From here we would start comparing from the end again. By aligning these characters, we can guarantee a match with those characters and increase our chances for a pattern match.

Sometimes we may have already passed the last occurence of our mismatched character. In such a case, we will shift over by 1.

n =	A	B	A	C	A	A	B	R	N
m =	A	B	A	C	A	B			
		A	B	A	C	A	B		
NO	A	B	A	C	A	B			
YES			A	B	A	C	A	B	

In the second iteration of comparisons, we mismatch with A in **n** and C in **m**. We look for the last occurrence of A in our pattern, but we've already passed this. Trying to align this old A would shift our pattern backwards. Therefore, we shift over by 1.

Boyer-Moore Last Table

To know how much to shift our pattern over by, we perform preprocessing over our pattern and create a **Last Table**. This table is a mapping between characters and numbers.

For our pattern **ABACAB**, our last table would look like:

chars	**A**	**B**	**C**	*
Last Occ.	4	5	3	-1

Underneath, complete the code that would create such a table.

```
int[] createLastTable(String pattern) {
  int[] table = new int[Characters.MAX_VALUE];









  return table;
}
```

```
procedure BoyerMoore(text, pattern)
  lastTable ← BoyerMooreLastTable(pattern)
  i ← 0
  while i <= length of text - length of pattern
    j ← length of pattern - 1
    while j >= 0 and text[i + j] = pattern[j]
      j ← j - 1
    end while
    if j = -1 then
      return i
    else
      shiftedIndex ← lastTable[text[i + j]]
      if shiftedIndex < j then
        i ← i + (j - shiftedIndex)
      else i ← i + 1
      end if
    end if
  end while
  return -1
end procedure
```

Boyer-Moore performs in O(**nm**) where **n** is the length of the text and **m** is the length of the pattern. Such is the case.

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

Rabin Karp

In 1987, Richard Karp and Michael Rabin wrote a paper *Rolling Hash (Rabin-Karp Algorithm)* that used the idea of hashing to perform pattern matching. Remember that a hash function will **map a String to a number**. Because of the property of hash functions, these mappings should be bijective(1 to 1). Therefore, two completely different strings should not output the same hash value. However, with the limitations of computers and hash functions, two strings may produce the same hash value. In such a case, we must make sure the two strings are identical, so then we compare characters. If our strings produce the same hash value and have same characters, then the two strings match. If our strings produce the same hash value and have mismatching characters, then they are not the same.

Of course with string searching, our strings we want to compare are the pattern **m** and substring of the text **n**. Using our idea of hashing, we will perform hash(**m**) and hash(**n[0 ... m.length-1]**) . The question is what is our hash function?

Rolling Hash

Let's say **m = ABACAB** and **n = ABABACAB** such that:

n =	A	B	A	B	A	C	A	B
m =	A	B	A	C	A	B		

To calculate the hash, the following formula is used:

$$\sum_{i=0}^{j} text[i] \ * BASE^{j-i}$$

where j is the number of letters in text, and BASE is a prime number.
With BASE = 101, **Hash(m)** would yield:

$$65 * 101^5 \ + \ 66 * 101^4 \ + \ 65 * 101^3 \ + \ 67 * 101^2 \ + \ 65 * 101^1 \ + \ 66 * 101^0$$

= **690092178694**

Because we're comparing our pattern to the beginning of our text, we'll hash an equal length of the beginning of the text **ABABAC**.
Hash(**n[0 ... m.length-1**) would yield:

$$65 * 101^5 \ + \ 66 * 101^4 \ + \ 65 * 101^3 \ + \ 66 * 101^2 \ + \ 65 * 101^1 \ + \ 67 * 101^0$$

= **690092168494**

These two hash values don't match, so we know for sure the strings don't match. We continue by shifting our pattern over by 1.

n =	A	B	A	B	A	C	A	B
m =	A	B	A	C	A	B		
		A	B	A	C	A	B	

And we would hash this new substring of text **BABACA.**

$$66 * 101^5 \ + \ 65 * 101^4 \ + \ 66 * 101^3 \ + \ 65 * 101^2 \ + \ 67 * 101^1 \ + \ 65 * 101^0$$

= **700499228894**

However, performing this hash on a new substring everytime we shift over is expensive. An **O(m)** hash function done **O(n-m)** times will yield an **O(nm)** algorithm... and if our hash function is bad or we produce the same hash values every time **O(m)** comparison, we'll get an **O(mnm)** algorithm.

Look at the first substring of **n** we hashed and the next substring.

A	B	A	B	A	C	A	B
A	B	A	B	A	C	A	B

The difference between the substrings we hash is the first character **(A)** is removed and the next character **(A)** is appended. Mathematically, our hash expressions share similar statements:

hash("ABABAC")

$$65 * 101^5 + 66 * 101^4 + 65 * 101^3 + 66 * 101^2 + 65 * 101^1 + 67 * 101^0$$

hash("BABACA")

$$66 * 101^5 + 65 * 101^4 + 66 * 101^3 + 65 * 101^2 + 67 * 101^1 + 65 * 101^0$$

$$= 101 * (66 * 101^4 + 65 * 101^3 + 66 * 101^2 + 65 * 101^1 + 67 * 101^0)$$
$$+ 65 * 101^0$$

Both expressions share

$$(66 * 101^4 + 65 * 101^3 + 66 * 101^2 + 65 * 101^1 + 67 * 101^0)$$

Using this observation, we can write a mathematical expression and code that handles this hash updating.

```
procedure RabinKarp(text, pattern)
  patternHash ← rolling hash of pattern
  textHash ← rolling hash of first pattern.length characters of text
  i ← 0
  while i <= text.length - pattern.length
    if patternHash = textHash then
      j ← 0
      while j < pattern.length and text[i + j] = pattern[j]
        j ← j + 1
      end while
      if j = length of pattern then
        return i
      end if
    end if i ← i + 1
    if i <= text.length - pattern.length then
      textHash ← new hash of text, with the hash window shifted over
    end if
  end while
  return -1
end procedure
```

P	**A**	**N**	**C**	**A**	**K**	**E**	**C**	**A**	**T**
C	A	T							
	C	A	T						
		C	A	T					
			C	A	T				
				C	A	T			
					C	A	T		
						C	A	T	
							C	A	T

Rabin-Karp will run in worst case **O(mn)** if our hash values end up being the same every shift. This would give us an **O(m)** comparison at most **O(n)** times.

Boyer Moore Example

chars	**A**	**B**	**C**	*****
Last Occ.	4	5	3	-1

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