CS2010: Data Structures and Algorithms II

Substring Search – part 2

Ivana.Dusparic@scss.tcd.ie

Substring search algorithms – part 2

› Boyer-Moore

› Rabin-Karp

› Suffix arrays/suffix trees and LCPs

So far

- Brute force
  - M x N performance
  - Back up
- KMP
  - 2N - linear
  - No backup
  - Extra space – M x R

  (N length of a string, M length of a substring we're search for, R radix)
- can we do better?

Boyer-Moore

Boyer-Moore

- Big idea – when find a character not in the pattern, can skip up to M characters (so no need to loop through all N characters)
  - Mismatched character heuristic
  - Don't look through characters in order, start from the back and look at the last character in the pattern first and see if it's a match, or in the pattern at all
- Uses backup

Boyer-Moore

Intuition.

- Scan characters in pattern from right to left.
- Can skip as many as M text chars when finding one not in the pattern.

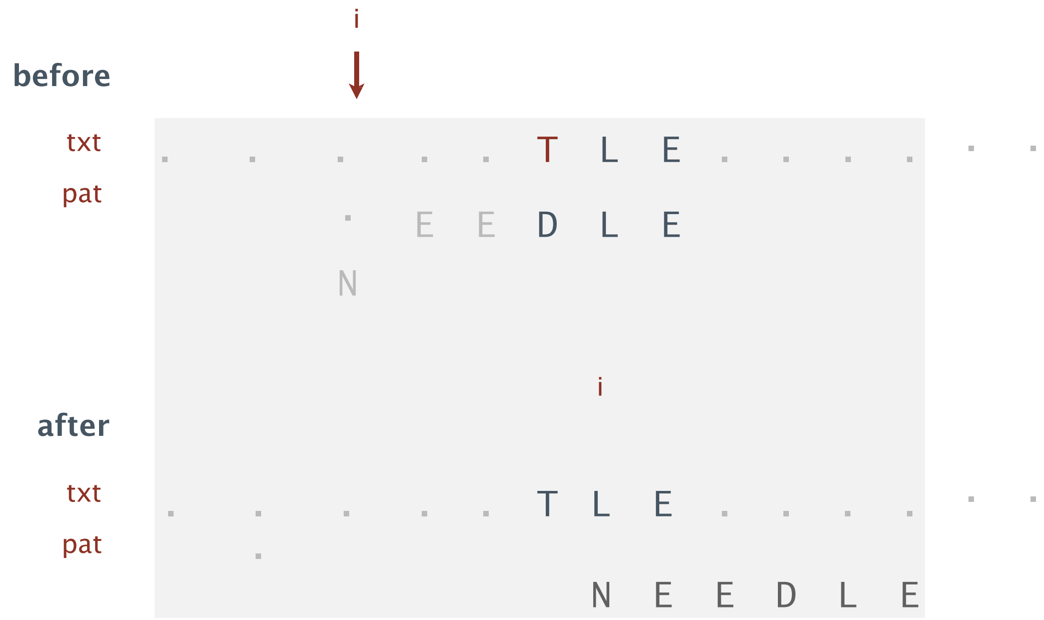
i	j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
	text →	F	I	N	D	I	N	A	H	A	Y	S	T	A	C	K	N	E	E	D	L	E	I	N	A
0	5	N	E	E	D	L	E ← *pattern*																		
5	5						N	E	E	D	L	E													
11	4												N	E	E	D	L	E							
15	0																N	E	E	D	L	E			

return i = 15

Boyer-Moore: mismatched character heuristic

Q. How much to skip?

Case 1. Mismatch character not in pattern.

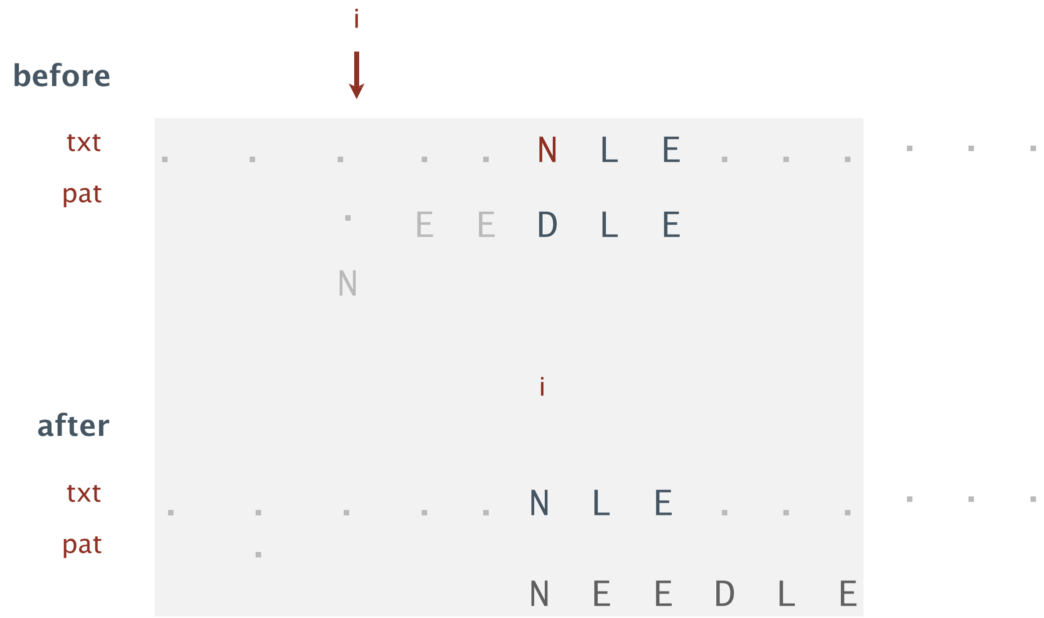


mismatch character 'T' not in pattern: increment i one character beyond 'T'

Boyer-Moore: mismatched character heuristic

Q. How much to skip?

Case 2a. Mismatch character in pattern.



mismatch character 'N' in pattern: align text 'N' with rightmost pattern 'N'

Boyer-Moore: mismatched character heuristic

Q. How much to skip?

Case 2b. Mismatch character in pattern (but heuristic no help).

i

before

txt . . . . . . E L E . . . . . .

pat N E E D L E

.

i

aligned with rightmost E?

txt . . . . . . E L E . . . . . .

pat

.

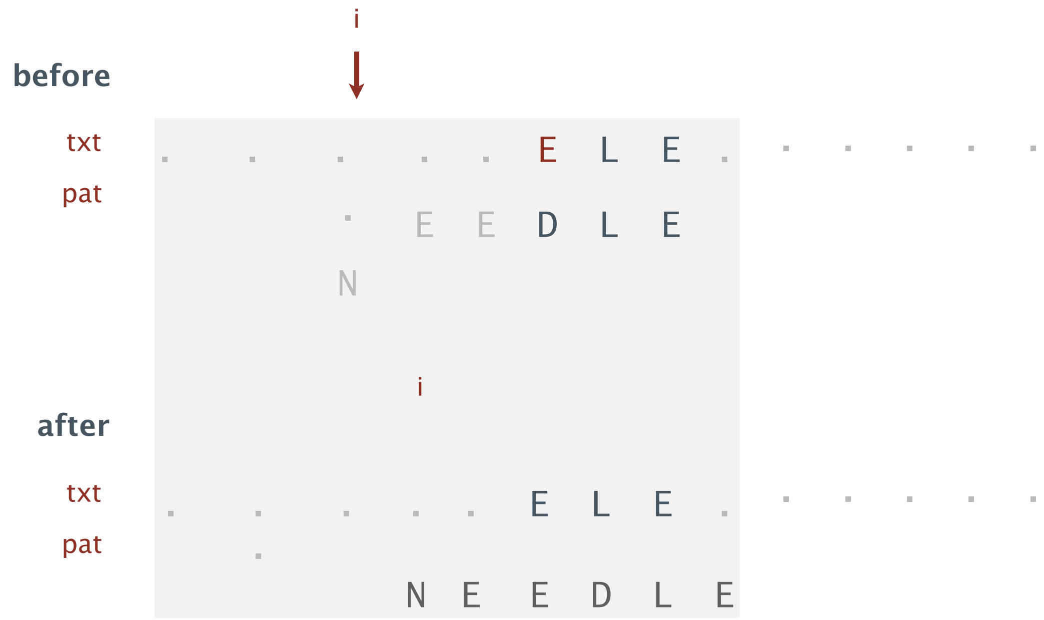
N E E D L E

mismatch character 'E' in pattern: align text 'E' with rightmost pattern 'E' ?

Boyer-Moore: mismatched character heuristic

Q. How much to skip?

Case 2b. Mismatch character in pattern (but heuristic no help).



mismatch character 'E' in pattern: increment i by 1

Boyer-Moore: mismatched character heuristic

Q. How much to skip?

A. Precompute index of rightmost occurrence of character c in pattern.
(-1 if character not in pattern)

```
right = new int[R];
for (int c = 0; c < R; c++)
   right[c] = -1;
for (int j = 0; j < M; j++)
   right[pat.charAt(j)] = j;
```

		N	E	E	D	L	E	
c		0	1	2	3	4	5	right[c]
A	-1	-1	-1	-1	-1	-1	-1	-1
B	-1	-1	-1	-1	-1	-1	-1	-1
C	-1	-1	-1	-1	-1	-1	-1	-1
D	-1	-1	-1	-1	3	3	3	3
E	-1	-1	1	2	2	2	5	5
...								-1
L	-1	-1	-1	-1	-1	4	4	4
M	-1	-1	-1	-1	-1	-1	-1	-1
N	-1	0	0	0	0	0	0	0
...								-1

Boyer-Moore skip table computation

Precomputing the index of right-most occurrence

```
// position of rightmost occurrence of c in the pattern
right = new int[R];
for (int c = 0; c < R; c++)
    right[c] = -1;
for (int j = 0; j < pattern.length; j++)
    right[pattern[j]] = j;
```

Boyer-Moore: Java implementation

```
public int search(String txt)
{
   int N = txt.length();
   int M = pat.length();
   int skip;
   for (int i = 0; i <= N-M; i += skip)
   {
      skip = 0;
      for (int j = M-1; j >= 0; j--)
      {
         if (pat.charAt(j) != txt.charAt(i+j))
         {
            skip = Math.max(1, j - right[txt.charAt(i+j)]);
            break;
         }
      }
      if (skip == 0) return i;
   }
   return N;
}
```

compute skip value

in case other term is nonpositive

match

Boyer-Moore: analysis

Property. Substring search with the Boyer-Moore mismatched character heuristic takes about $\sim N / M$ character compares to search for a pattern of length M in a text of length N. sublinear!

What's the worst case input/performance of Boyle-Moore?

Turning point.

Boyer-Moore: analysis

Worst-case. Can be as bad as $\sim M N$.

i	skip	0	1	2	3	4	5	6	7	8	9
	txt	B	B	B	B	B	B	B	B	B	B
0	0	A	B	B	B	B		pat			
1	1		A	B	B	B	B				
2	1			A	B	B	B	B			
3	1				A	B	B	B	B		
4	1					A	B	B	B	B	
5	1						A	B	B	B	B

Rabin-Karp

Rabin-Karp

Basic idea = modular hashing.

- Compute a hash of `pat[0..M-1]`.
- For each `i`, compute a hash of `txt[i..M+i-1]`.
- If pattern hash = text substring hash, check for a match.

pat.charAt(i)

i	0	1	2	3	4	
	2	6	5	3	5	% 997 = 613

txt.charAt(i)

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
	3	1	4	1	5	9	2	6	5	3	5	8	9	7	9	3	
0	3	1	4	1	5												% 997 = 508
1		1	4	1	5	9											% 997 = 201
2			4	1	5	9	2										% 997 = 715
3				1	5	9	2	6									% 997 = 971
4					5	9	2	6	5								% 997 = 442
5						9	2	6	5	3							% 997 = 929
6 ← *return* i = 6							2	6	5	3	5						% 997 = 613 *match*

modular hashing with R = 10 and hash(s) = s (mod 997)

Modular hashing

- Remember hash tables
- *hash function* - map data of arbitrary size to data of fixed size
- Eg map any value to an index in the array
- With **modular hashing**, the hash function is simply $h(k) = k \bmod m$ for some m. The value k is an integer hash code generated from the key (generally used with positive integers)

```
int h(int k, int M) {
    return k% M;
}
```

Rabin-Karp – modular hashing

› Won't it overflow, for large search substrings?

Math trick. To keep numbers small, take intermediate results modulo Q.

Ex.
$$
\begin{aligned}
(10000 + 535) * 1000 &\pmod{997} \\
= (30 + 535) * 3 &\pmod{997} \\
= 1695 &\pmod{997} \\
= 698 &\pmod{997}
\end{aligned}
$$

$$(a + b) \bmod Q = ((a \bmod Q) + (b \bmod Q)) \bmod Q$$

$$(a * b) \bmod Q = ((a \bmod Q) * (b \bmod Q)) \bmod Q$$

two useful modular arithmetic identities

Efficiently computing the hash function

Modular hash function. Using the notation t_i for `txt.charAt(i)`, we wish to compute

$$x_i = t_i R^{M-1} + t_{i+1} R^{M-2} + \ldots + t_{i+M-1} R^0 \pmod{Q}$$

Intuition. M-digit, base-R integer, modulo Q.

Horner's method. Linear-time method to evaluate degree-M polynomial.

```
         pat.charAt()
 i    0    1    2    3    4
      2    6    5    3    5
 0    2    % 997 = 2           R                Q
 1    2    6    % 997 = (2*10 + 6) % 997 = 26
 2    2    6    5    % 997 = (26*10 + 5) % 997 = 265
 3    2    6    5    3    % 997 = (265*10 + 3) % 997 = 659
 4    2    6    5    3    5    % 997 = (659*10 + 5) % 997 = 613
```

```
// Compute hash for M-digit key
private long hash(String key, int M)
{
   long h = 0;
   for (int j = 0; j < M; j++)
      h = (h * R + key.charAt(j)) % Q;
   return h;
}
```

26535 = 2*10000 + 6*1000 + 5*100 + 3*10 + 5
= ((((2) *10 + 6) * 10 + 5) * 10 + 3) * 10 + 5

Efficiently computing the hash function

Challenge. How to efficiently compute x_{i+1} given that we know x_i.

$$x_i \ = t_i R^{M-1} + t_{i+1} R^{M-2} + \ldots + t_{i+M-1} R^0$$

$$x_{i+1} = t_{i+1} R^{M-1} + t_{i+2} R^{M-2} + \ldots + t_{i+M} R^0$$

Key property. Can update "rolling" hash function in constant time!

$$x_{i+1} \ = \ (\underbrace{x_i}_{\text{current value}} \ - \ \underbrace{t_i R^{M-1}}_{\text{subtract leading digit}}) \ \underbrace{R}_{\text{multiply by radix}} \ + \ \underbrace{t_{i+M}}_{\text{add new trailing digit}}$$

(can precompute R^{M-1})

	i	...	2	3	4	5	6	7	...	
current value		1	4	1	5	9	2	6	5	*text*
new value			4	1	5	9	2	6	5	

	4	1	5	9	2	*current value*
-	4	0	0	0	0	
		1	5	9	2	*subtract leading digit*
			*	1	0	*multiply by radix*
	1	5	9	2	0	
				+	6	*add new trailing digit*
	1	5	9	2	6	*new value*

Rabin-Karp substring search example

First R entries: Use Horner's rule.

Remaining entries: Use rolling hash (and % to avoid overflow).

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	3	1	4	1	5	9	2	6	5	3	5	8	9	7	9	3

Q

0 3 % 997 = 3

1 3 1 % 997 = (3*10 + 1) % 997 = 31

2 3 1 4 % 997 = (31*10 + 4) % 997 = 314

3 3 1 4 1 % 997 = (314*10 + 1) % 997 = 150

4 3 1 4 1 5 % 997 = (150*10 + 5) % 997 = 508

Horner's rule

RM R

5 1 4 1 5 9 % 997 = ((508 + 3*(997 - 30))*10 + 9) % 997 = 201

6 4 1 5 9 2 % 997 = ((201 + 1*(997 - 30))*10 + 2) % 997 = 715

7 1 5 9 2 6 % 997 = ((715 + 4*(997 - 30))*10 + 6) % 997 = 971

8 5 9 2 6 5 % 997 = ((971 + 1*(997 - 30))*10 + 5) % 997 = 442

9 9 2 6 5 3 % 997 = ((442 + 5*(997 - 30))*10 + 3) % 997 = 929

match

10 ← *return* i-M+1 = 6 2 6 5 3 5 % 997 = ((929 + 9*(997 - 30))*10 + 5) % 997 = 613

rolling hash

-30 (mod 997) = 997 – 30

10000 (mod 997) = 30

Rabin-Karp: Java implementation

```
public class RabinKarp
{
   private long patHash;    // pattern hash
                                value
   private int M;           // pattern length
   private long Q;          // modulus
   private int R;            / radix
   private long RM1;         / R^(M-1) % Q
   public RabinKarp(String pat) {
                             /
      M = pat.length();      /
      R = 256;
      Q = longRandomPrime();

      RM1 = 1;
      for (int i = 1; i <= M-1; i++)
         RM1 = (R * RM1) % Q;
      patHash = hash(pat, M);
   }

   private long hash(String key, int M)
   {  /* as before */  }

   public int search(String txt)
   {  /* see next slide */  }
}
```

a large prime
(but avoid overflow)

precompute R^{M-1} (mod Q)

Rabin-Karp: Java implementation (continued)

Monte Carlo version. Return match if hash match.

```
public int search(String txt)
{
    int N = txt.length();
    int txtHash = hash(txt, M);
    if (patHash == txtHash) return 0;
    for (int i = M; i < N; i++)
    {
        txtHash = (txtHash + Q - RM*txt.charAt(i-M) % Q) % Q;
        txtHash = (txtHash*R + txt.charAt(i)) % Q;
        if (patHash == txtHash) return i - M + 1;
    }
    return N;
}
```

check for hash collision
using rolling hash function

Las Vegas version. Check for substring match if hash match; continue search if false collision.

Rabin-Karp analysis

Monte Carlo version.

- Always runs in linear time.
- Extremely likely to return correct answer (but not always!).

Las Vegas version.

- Always returns correct answer.
- Extremely likely to run in linear time (but worst case is $M N$).

Rabin-Karp

Advantages.

- Extends to 2d patterns.
- Extends to finding multiple patterns.

Disadvantages.

- Arithmetic ops slower than char compares.
- Las Vegas version requires backup.
- Poor worst-case guarantee.

Substring search cost summary

Cost of searching for an M-character pattern in an N-character text.

algorithm	version	operation count		backup in input?	correct?	extra space
		guarantee	typical			
brute force	—	MN	$1.1\,N$	*yes*	*yes*	1
Knuth-Morris-Pratt	*full DFA* (*Algorithm 5.6*)	$2N$	$1.1\,N$	*no*	*yes*	MR
	mismatch transitions only	$3N$	$1.1\,N$	*no*	*yes*	M
Boyer-Moore	*full algorithm*	$3N$	N/M	*yes*	*yes*	R
	mismatched char heuristic only (*Algorithm 5.7*)	MN	N/M	*yes*	*yes*	R
Rabin-Karp†	*Monte Carlo* (*Algorithm 5.8*)	$7N$	$7N$	*no*	*yes*†	1
	Las Vegas	$7N$ †	$7N$	*yes*	*yes*	1

† *probabilisitic guarantee, with uniform hash function*

So which algorithm should I use?

- Java String.contains() method – brute force
  - Very compact, very little operations inside the loop, so loop runs fast. As there is no overhead it performs better when searching short strings.

- Boyer-Moore
  - grep
  - Works well for long search patterns
  - The more distinct letters in the strings, the better the positive impact
  - backup

- KMP
  - Small alphabet, repeated subpatterns
  - No backup

Suffix Arrays and LCP Arrays

Suffix Tree

- A tree containing all the suffixes of the given text as their keys and positions in the text as their values
- Space and time linear to length of the string
- Speeds up operations such as: substring search, matches for regular expression patterns, longest common substring etc
- Cost - storing a string's suffix tree typically requires significantly more space than storing the string itself.

Suffix Array

- slower but takes up less space and better cache performance

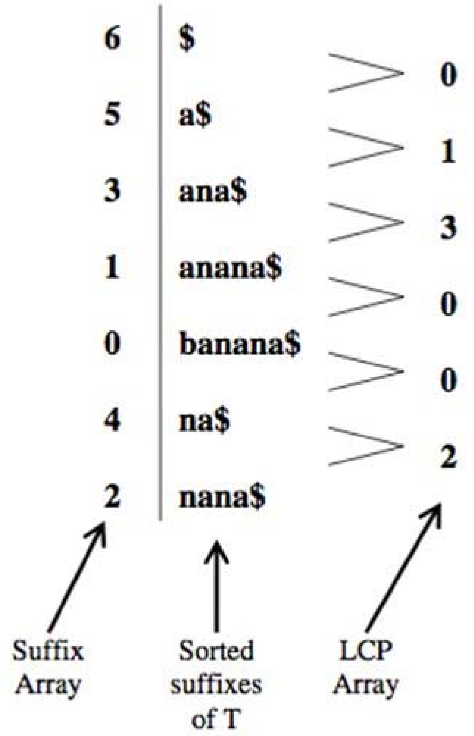
Suffix											Index
a	b	r	a	c	a	d	a	b	r	a	0
	b	r	a	c	a	d	a	b	r	a	1
		r	a	c	a	d	a	b	r	a	2
			a	c	a	d	a	b	r	a	3
				c	a	d	a	b	r	a	4
					a	d	a	b	r	a	5
						d	a	b	r	a	6
							a	b	r	a	7
								b	r	a	8
									r	a	9
										a	10

Sorted Suffix Array

Sorted Suffix											Index
a											10
a	b	r	a								7
a	b	r	a	c	a	d	a	b	r	a	0
a	c	a	d	a	b	r	a				3
a	d	a	b	r	a						5
b	r	a									8
b	r	a	c	a	d	a	b	r	a		1
c	a	d	a	b	r	a					4
d	a	b	r	a							6
r	a										9
r	a	c	a	d	a	b	r	a			2

Longest Common Prefix Array

- stores the lengths of the longest common prefixes (LCPs) between all pairs of consecutive suffixes in a sorted suffix array.



More on this and practice problems

› Robert Sedgewick and Kevin Wayne
https://algs4.cs.princeton.edu/63suffix/

› Suffix arrays – a programming contest approach
http://web.stanford.edu/class/cs97si/suffix-array.pdf