CSC 344 – Algorithms and Complexity

Lecture #7 – Pattern Matching

String (or Pattern) Searching

- Pattern matching is a special case of sequential searching.
- It has many applications:
  - Applications in Computational Biology
  - Finding patterns in documents formed using a large alphabet
  - Matching strings of bytes containing a pattern
  - grep in unix

Pattern Searching in Computational Biology

- DNA sequence is a long word (or text) over a 4-letter alphabet
- GTTTGAGTGGTCAGTCTTTTCGTTTCGA CGGAGCCCCCAATTAATAAACTCATAAG CAGACCTCAGTTCGCTTAGAGCAGCCG AAA…..
- Find a Specific pattern W

Pattern Matching in Documents

- There are many places where such matching appears:
  - Word processing – important in trial preparation
  - Web searching
  - Desktop search (Google, MSN)

Not All Searches Are Text…

- Graphical data
- Machine code

String Matching Preliminaries

- **Pattern** – the string that we seek.
- **Text** – the longer string in which we are searching for the pattern.
- **Target** – an instance of the pattern within the text

Brute Force Searching

- A straight-forward example of the "sliding pattern" model.

1. Place the pattern at the start of the text and wee whether all the characters match.
2. If they do, the target is found. If not, then stop comparing after the first mismatch, shift the pattern one character to the right and try again.
3. Keep trying until the search succeeds or the end of the pattern extends past the end of the text.

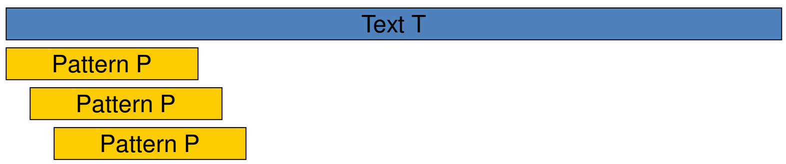
String Matching

```
abacaabaccabacabaabb
abacab
 abacab
  abacab
   abacab
    abacab
     abacab
      abacab
       abacab
        abacab
         abacab
          abacab
```

- **The brute force algorithm**
- **22+6=28** comparisons.

Brute Force

- Assume |T| = n and |P| = m



Compare until a match is found. If so return the index where match occurs

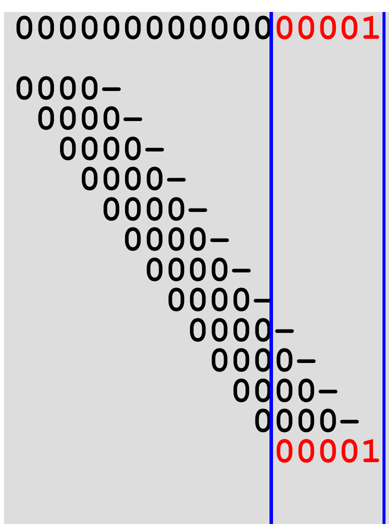
else return -1

bruteForceSearch()

```
int    bruteForceSearch
             (char text[], char pattern[]) {
       int   offset, //offset in text
             pat;    //pat is pattern's subscript
       int   n, m;

       m = strlen(pattern);
       n = strlen(text);

       pat = 0;
       offset = 0;
```

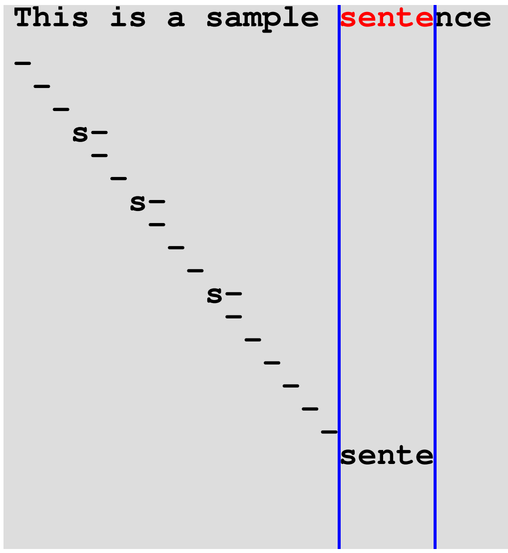
```
        while (pat < m && (pat + offset) < n)      {
              if (pattern[pat] == text[pat+offset])
                    pat++;
              else  {
                    offset++;
                    pat = 0;
              }
        }
        if (pat >= m)
              return offset;
        else
              return -1;


  }
```

A Bad Case



- 60+5 = **65** comparisons are needed
- **How many of them could be avoided?**

Typical Text Matching



- 20+5=**25** comparisons are needed

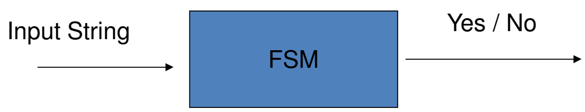
  (The match is near the same point in the target string as the previous example.)
- In practice, $0 \le j \le 2$

How Bad Is Brute Force?

- Brute force worst case
  - $O(m \cdot n)$, n = string length, m = pattern length
  - Expensive for long patterns in repetitive text
- How to improve on this?
- Intuition:
  - Remember what is learned from previous matches

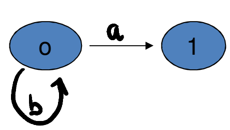
Finite State Machines (FSM)

- FSM is a computing machine that takes
  - A string as an input
  - Outputs YES/NO answer
    - That is, the machine "accepts" or "rejects" the string



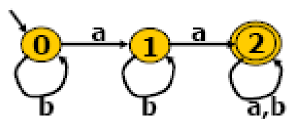
FSM Model

- Input to a FSM
  - Strings built from a fixed alphabet {a,b,c}
  - Possible inputs: aa, aabbcc, a etc..
- The Machine
  - A directed graph
    - Nodes = States of the machine
    - Edges = Transition from one state to another



FSM Model

- Special States
  - Start (q0) and Final (or Accepting) (q2)
- Assume the alphabet is {a,b}
  - Which strings are accepted by this FSM?



FSM Model

- **Exercise**: draw a finite automaton that accepts any string with “even” number of 1’s
- **Exercise**: draw a finite automaton that accepts any string with “even” number of consecutive 1’s followed by “odd” number of consecutive zeros

State Transitions

- Let Q be the set of states and ∑ be the alphabet. Then the transition function T is given by
  - T: Q x ∑ ➔ Q
- ∑ could be
  - {0,1} – binary
  - {C,G,T,A} – nucleotide base
  - {0,1,2,..,9,a,b,c,d,e,f} – hexadecimal
  - etc..

State Transitions (continued)

- Eg: Consider ∑ ={a,b,c} and P=aabc
  - set of states are all prefixes of P
  - Q = {   , a, aa, aab, aabc}   **or**
  - Q = {0   1   2   3   4 }
- State transitions T(0,'a') = 1; T(1, 'a') = 2, etc…
- What about failure transitions?

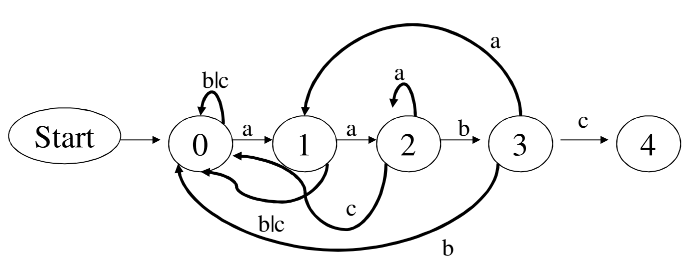
Failure Transitions

- Where do we go when a failure occurs?
- P="aabc"
- Q – current state
- Q' – next state
- initial state = 0
- end state = 4
- How to store state transition table?
  - as a matrix

Q	Σ	Q'
0	a	1
	{b,c}	0
1	a	2
	{b,c}	0
2	b	3
	a	2
	c	0
3	c	4
	a	1
	b	0

Using Finite State Machines in Pattern Matching

- Consider the alphabet {a,b,c}
- Suppose we are looking for pattern "aabc"
- Construct a finite automaton for "aabc" as follows



The Big Idea – The Knuth – Morris-Pratt (KMP) Algorithm

- Retain information from prior attempts.
- **Compute in advance how far to jump in P when a match fails.**
  - Suppose the match fails at `P[j]` ≠ `T[i+j]`.
  - Then we know that

    `P[0 .. j-1] = T[i .. i+j-1]`.

The Big Idea – The Knuth – Morris-Pratt Algorithm

- We must next try `P[0] ?= T[i+1]`.
  - But we know `T[i+1]=P[1]`
  - What if we compare: `P[1]?= P[0]`
    - If so, increment **j** by **1**. No need to look at **T**.
  - What if `P[1]=P[0]` and `P[2]=P[1]`?
    - Then increment `j` by `2`. Again, no need to look at `T`.
- In general, we can determine how far to jump without any knowledge of `T`!

Implementing KMP

- Never decrement **i**, ever.
  - Comparing
    T[i] with **P[j]**.
- Compute a table **f** of how far to jump **j** forward when a match fails.
  - The next match will compare
    T[i] with **P[f[j-1]]**
- Do this by matching **P** against itself in all positions.

Building the Table for `f`

- P = `1010011`
- Find self-overlaps

Table for the Failure Function **f**

Prefix	Overlap	j	f
1	.	1	0
10	.	2	0
10**1**	**1**	3	1
10**10**	**10**	4	2
10100	.	5	0
10100**1**	**1**	6	1
101001**1**	**1**	7	1

What **f** Means

- If **f** is zero, there is no self-match.
  - Set **j=0**
  - Do not change **i**.
    - The next match is **T[i] ?= P[0]**

What **`f`** Means

- **`f`** being non-zero implies there is a self-match.
  E.g., **`f=2`** means
  `P[0..1] = P[j-2..j-1]`
- Hence must start new comparison at **`j-2`**, since we know **`T[i-2..i-1] = P[0..1]`**

What **`f`** Means

In general:

- Set **`j=f[j-1]`**
- Do not change **i**.
  - The next match is
    `T[i]  ?=  P[f[j-1]]`

Favorable Conditions

- P = **1234567**
- Find self-overlaps

Prefix	Overlap	j	f
1	.	1	0
12	.	2	0
123	.	3	0
1234	.	4	0
12345	.	5	0
123456	.	6	0
1234567	.	7	0

Mixed Conditions

- P = **1231234**
- Find self-overlaps

Prefix	Overlap	j	f
1	.	1	0
12	.	2	0
123	.	3	0
123**1**	**1**	4	1
123**12**	**12**	5	2
123**123**	**123**	6	3
1231234	.	7	0

Mixed Conditions

- P = **1111110**
- Find self-overlaps

Prefix	Overlap	j	f
1	.	1	0
1**1**	**1**	2	1
1**11**	**11**	3	2
1**111**	**111**	4	3
1**1111**	**1111**	5	4
1**11111**	**11111**	6	5
1111110	.	7	0

kmpSearch()

```
// kmpSearch() - the Knuth-Moore-Pratt Algorithm
int StringSearch::kmpSearch(char text[],
                              char pattern[]) {
  int n, m;
  int i, j;
  int f[patternStringLength];

  n = strlen(text);
  m = strlen(pattern);

  // Compute Table F for Pattern P
  kmpMakeTable(f, m, n, pattern);
  i = j = 0;
```

```
  // As long as you're still in the string
  while(i < n) {
    if(pattern[j] == text[i]) {
            // You've reached the end of the
            // pattern
            if (j == m-1)
                  return (i - m + 1);
            i++; j++;
      }
    // No match - check the failure table to see
    // where to go
    else if (j > 0)
            j=f[j-1];
    else
            i++;
  }
}
```

kmpMakeTable()

```
// kmpMakeTable() - Building the Failure Table
void  StringSearch::kmpMakeTable(int f[], int m,
                        int n, char pattern[])  {
    int i, j;
    // Define a table f of size m
    f[0] = 0;
    i = 1; j = 0;
    while(i<m) {
      // compare P[i] and P[j];
      if(pattern[j] == pattern[i]) {
          // They match; continue
          f[i] = j+1;
          i++; j++;
      }
```

```
            else if (j>0)
                  j = f[j-1];
            else  {
                  f[i] = 0;
                  i++;
            }
      }
}
```

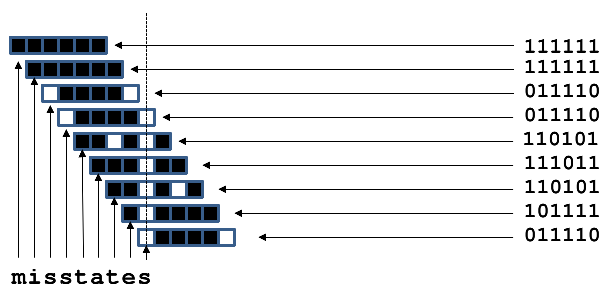
Baeza-Yates – Gonnet (BYG) Algorithm

- Published by Ricardo Baeza-Yates and Gaston Gonnet in 1992.
- The BYG (or Bitap) algorithm uses an array of bit vectors (one for each character0 to serve as bit masks.
- Each position in the bit map corresponds to a character in "alphabet"; the vectors are as long as the pattern

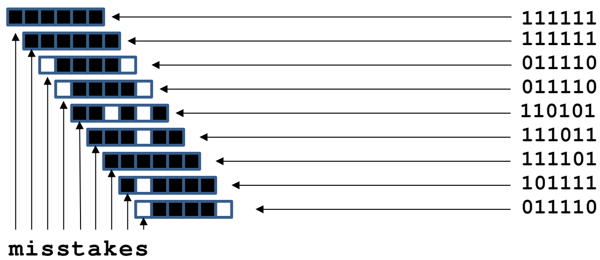
Example – Searching for **`states`**

	654321
a	111011
b	111111
c	111111
d	111111
e	101111
f	111111
…	…
r	111111
s	011110
t	110101
u	111111
…	

Using The Mask for **`misstates`**



Using The Mask for **misstakes**



bygSearch()

```
int StringSearch::bygSearch(char text[],
                               char pattern[])  {
     int m = strlen(pattern);
     unsigned long R;
     unsigned long pattern_mask[CHAR_MAX+1];
     int i;

     if (pattern[0] == '\0') return 0;
     if (m > 31) return -2;

     /* Initialize the bit array R */
     R = ~1;
```

```
    /* Initialize the pattern bitmasks */
    for (i = 0; i <= CHAR_MAX; ++i)
      pattern_mask[i] = ~0;
    for (i = 0; i < m; ++i)
      pattern_mask[pattern[i]] &= ~(1UL << i);

    for (i = 0; text[i] != '\0'; ++i) {
        /* Update the bit array */
        R |= pattern_mask[text[i]];
        R <<= 1;

        if (0 == (R & (1UL << m)))
          return (i - m) + 1;
    }
     return -1;
}
```

Efficiency of BYG Search

- The efficiency is $\Theta(n)$, which is the length of the text string. Each pass through the target is a constant multiplier.

Boyer-Moore Algorithm

- 3 main ideas
  - Right to left scan
  - Bad character rule
  - Good suffix rule

Substring Search Right to Left

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
text→		F	I	N	D	I	N	A	H	A	Y	S	T	A	C	K	N	E	E	D	L	E	I	N
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**		

Skip Table

		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	-1	**0**	0	0	0	0	0
…								-1

The Bad Character Rule (BCR)

- On a mismatch between the pattern and the text, we can shift the pattern by more than one place.

Sublinearity!

ddbbacdcbaabcddcdaddaaabcbcb

acabc

Summarizing the Bad Character Rule

- On a mismatch, shift the pattern to the right until the first occurrence of the mismatched char in P.
- Still O(n m) worst case running time:

```
T: aaaaaaaaaaaaaaaaaaaaaaaaa
P: abaaaa
```

The Good Suffix Rule (GSR)

- We want to use the knowledge of the matched characters in the pattern's suffix.
- If we matched S characters in T, what is (if exists) the smallest shift in P that will align a sub-string of P of the same S characters ?

GSR - Example

- Example 1 – how much to move:

```
 ↓
T: bbacdcbaabcddcdaddaaabcbcb
P:  cabbabdbab
        cabbabdbab
```

GSR - Example

- Example 2 – what if there is no alignment:

```
                      ↓
T: bbacdcbaabcbbabdbabcaabcbcb
P: bcbbabdbabc
            bcbbabdbabc
```

GSR – Detailed

- We mark the matched sub-string in T with t and the mismatched char with x
- In case of a mismatch: shift right until the first occurrence of t in P such that the next char y in P holds $y \neq x$
- Otherwise, shift right to the largest prefix of P that aligns with a suffix of t.

Boyer Moore Algorithm

Preprocess(P)

k := n

while ($k \leq m$) do

- Match P and T from right to left starting at k
- If a mismatch occurs: shift P right (advance k) by max(good suffix rule, bad char rule).
- else, print the occurrence and shift P right (advance k) by the good suffix rule.

Algorithm Correctness

- The bad character rule shift never misses a match
- The good suffix rule shift never misses a match

`boyerMooreSearch()`

```
int boyerMooreSearch(char text[], char pattern[])  {
      const int   R = 256;     // Size of the
                               // character set
      int         right[R];
      int         m, n;  // size of the pattern and
                         // text respectively
      int         skip;

      m = strlen(pattern);
      n = strlen(text);

      // Create the skip table
      // -1 means the character is not in the
      // pattern
```

```
        for (int c = 0;  c < R;  c++)
                right[c] = -1;

        // Rightmost position for characters in the
        // pattern
        for (int j = 0;  j < m; j++)
            right[pattern[j]] = j;

        for (int i = 0;  i <= n - m; i +=skip)    {
            // Does the pattern match the text at
            // position i?
            skip = 0;
            for (int j = m - 1; j >= 0;  --j)
                if (pattern[j] != text[i+j])  {
                    skip = j - right[text[i+j]];
```

```
                    if (skip < 1)
                        // Found it
                        skip = 1;
                    break;
                }
                if (skip  == 0 )
                    return i;
        }
        // Didn't find it
        return -1;
}
```