String Matching

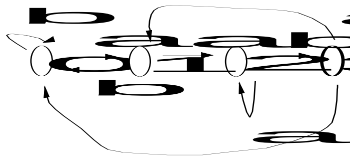
String Matching

- Problem is to find if a pattern P[1..m] occurs within text T[1..n]
- Simple solution: Naïve String Matching
  - Match each position in the pattern to each position in the text
    - `T    =     AAAAAAAAAAAAAA`
    - `P    =     AAAAAB`

```
               AAAAAB
```
                etc.
  - O(mn)

String Matching Automaton

- Create a DFA to match the string, just like we did in the automata portion of the class
- Example for string “aab” with ∑ = {a,b}:



- Runs in O(n) time but requires O(m|∑|) time to construct the DFA, where ∑ is the alphabet

Rabin Karp

- Idea: Before spending a lot of time comparing chars for a match, do some pre-processing to eliminate locations that could not possibly match
- If we could quickly eliminate most of the positions then we can run the naïve algorithm on whats left
- Eliminate enough to hopefully get O(n) runtime overall

Rabin Karp Idea

- To get a feel for the idea say that our text and pattern is a sequence of bits.
  - For example,
    - `P=010111`
    - `T=0010110101001010011`
  - The parity of a binary value is to count the number of one’s. If odd, the parity is 1. If even, the parity is 0. Since our pattern is six bits long, let’s compute the parity for each position in T, counting six bits ahead. Call this f[i] where f[i] is the parity of the string T[i..i+5].

Parity

T=0010110101001010011

P=010111

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
t[i]	0	0	1	0	1	1	0	1	0	1	0	0	1	0	1	0	0	1	1
f[i]	1	1	0	1	0	1	0	1	0	1	0	0	1	1					

Since the parity of our pattern is 0, we only need to check positions 2, 4, 6, 8, 10, and 11 in the text

Rabin Karp

- On average we expect the parity check to reject half the inputs.
- To get a better speed-up, by a factor of q, we need a fingerprint function that maps m-bit strings to q different fingerprint values.
- Rabin and Karp proposed to use a hash function that considers the next m bits in the text as the binary expansion of an unsigned integer and then take the remainder after division by q.
- A good value of q is a prime number greater than m.

Rabin Karp

- More precisely, if the m bits are $s_0 s_1 s_2$ .. s_m-1 then we compute the fingerprint value: $\left(\sum_{j=0}^{m-1} s_j 2^{m-1-j}\right) \bmod q$
- For the previous example, f[i] = $\left(\sum_{j=0}^{5} t[i+j] 2^{5-j}\right) \bmod 7$

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
t[i]	0	0	1	0	1	1	0	1	0	1	0	0	1	0	1	0	0	1	1
$\sum$	11	22	45	26	53	42	20	41	18	37	10	20	41	19					
f[i]	4	1	3	5	4	0	6	6	4	2	3	6	6	5					

For our pattern 010111, its hash value is 23 mod 7 or 2. This means that we would only use the naïve algorithm for positions where f[i] = 2

Rabin Karp Wrapup

- But we want to compare text, not bits!
  - Text is represented using bits
  - For a textual pattern and text, we simply convert the pattern into a sequence of bits that corresponds to its ASCII sequence, and the same for the text.
- Skipping the details of the actual implemention, we can compute f[i] in O(m) time giving us the expected runtime of O(m+n) given a good hashing.

KMP : Knuth Morris Pratt

- This is a famous linear-time running string matching algorithm that achieves a O(m+n) running time.
- Uses an auxiliary function pi[1..m] precomputed from P in time O(m).
- We'll give an overview of it here but not go into details of how to implement it.

Pi Function

- This function contains knowledge about how the pattern matches shifts against itself.
- If we know how the pattern matches against itself, we can slide the pattern more characters ahead than just one character as in the naïve algorithm.

Pi Function Example

Naive

P: **pappar**
T: **pappap**papparrassanuaragh

P: **pappar**
T: **pappap**papparrassanuaragh

Smarter technique:

We can slide the pattern ahead so that the longest PREFIX of P that we have already processed matches the longest SUFFIX of T that we have already matched.

P: pa**ppar**
T: pappa**ppap**parrassanuaragh

KMP Example

P: pa**ppar**
T: pappa**ppap**parrassanuaragh

P: pa**ppar**
T: pappappa**ppar**rassanuaragh

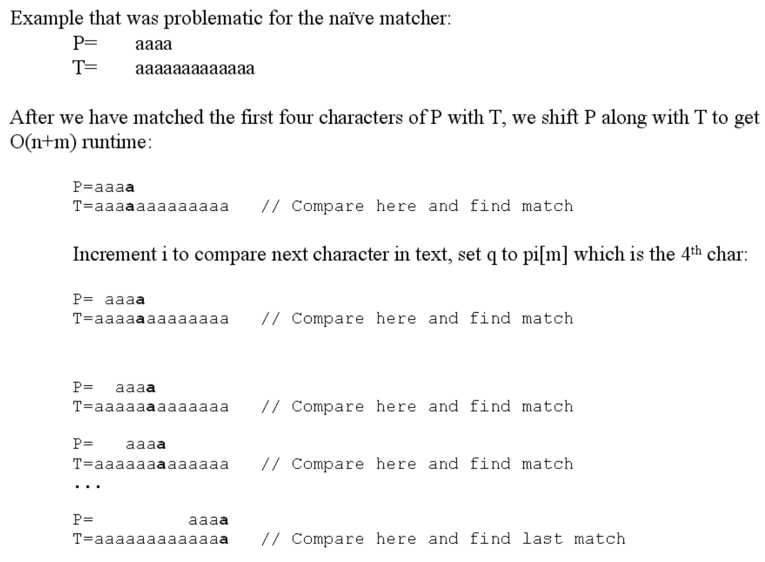
P: **p**appar
T: pappappapparr**r**assanuaragh

The characters mismatch so we shift over one character for both the text and the pattern:

P: **p**appar
T: pappappapparr**a**ssanuaragh

We continue in this fashion until we reach the end of the text.

KMP Example


Example that was problematic for the naïve matcher:

P=   aaaa
T=   aaaaaaaaaaaaa

After we have matched the first four characters of P with T, we shift P along with T to get O(n+m) runtime:

```
P=aaaa
T=aaaaaaaaaaaaa   // Compare here and find match
```

Increment i to compare next character in text, set q to pi[m] which is the 4th char:

```
P= aaaa
T=aaaaaaaaaaaaa   // Compare here and find match


P=  aaaa
T=aaaaaaaaaaaaa   // Compare here and find match

P=   aaaa
T=aaaaaaaaaaaaa   // Compare here and find match
...

P=         aaaa
T=aaaaaaaaaaaaa   // Compare here and find last match
```

KMP

- More details in the book how to implement KMP, skipping here.
  - Build a special type of DFA
- Runtime
  - O(m) to compute the Pi values
  - O(n) to compare the pattern to the text
  - Total O(n+m) runtime

Horspool’s Algorithm

- It is possible in some cases to search text of length n in less than n comparisons!
- Horspool’s algorithm is a relatively simple technique that achieves this distinction for many (but not all) input patterns. The idea is to perform the comparison from right to left instead of left to right.

Horspool’s Algorithm

- Consider searching:

```
T=BARBUGABOOTOOMOOBARBERONI
P=BARBER
```

- There are four cases to consider

  1. There is no occurrence of the character in T in P. In this case there is no use shifting over by one, since we’ll eventually compare with this character in T that is not in P. Consequently, we can shift the pattern all the way over by the entire length of the pattern (m):

```
T=   BARBUGABOOTOOMOOBARBERONI
P=   BARBER
P=         BARBER
```

Horspool’s Algorithm

2. There is an occurrence of the character from T in P. Horspool’s algorithm then shifts the pattern so the rightmost occurrence of the character from P lines up with the current character in T:

```
T=   BARBUBABOOTOOMOOBARBERONI
P=   BARBER
P=     BARBER
```

Horspool’s Algorithm

We’ve done some matching until we hit a character in T that is not in P. Then we look at the last character in P and shift:

3. By the entire pattern if the rightmost character in P does not appear again to the left

```
T=    BARTERRBERBOOTOOMOOBARBERONI
P=    BAZBER
P=          BAZBER
```

4. By an amount to line up the next to last character in P (among the m-1 characters) to the last character, like case 2

```
T=    BARAERABOOTOOMOOBARBERONI
P=    BARBER
P=       BARBER
```

Horspool’s Algorithm

- More on case 4

Here is another one where the shift shows how it would result in a match:

```
T=    BABBARBEROTOOMOOBARBERONI
P=    BARBER
P=       BARBER
```

Here is a case where the algorithm doesn’t perform as well as it could:

```
T=    AABABAAAAAAA
P=    BABAAAAAA
P=     BABAAAAAA
```

Horspool Implementation

- We first precompute the shifts and store them in a table. The table will be indexed by all possible characters that can appear in a text. To compute the shift T(c) for some character c we use the formula:
  - T(c) = the pattern's length m, if c is not among the first m-1 characters of P, else the distance from the rightmost occurrence of c in P to the end of P

Pseudocode for Horspool

```
Horspool(P[0..m-1], T[0..n-1])
    T ← ComputeShifts(P)                 ' Generate table of shifts
    i ← m – 1                            ' Position of pattern's right end
    while i ≤ n – 1
        k ← 0                            ' Number of matched characters
        while k ≤ m -1 and P[m-1-k] = T[i-k]
            k++
        if k = m
            return "Match at " + (i – m + 1)
        else
            i ← i + T[i]
    return -1                            ' No match found
```

Horspool Example

Here is an example for BARBER:

Character c	A	B	C	D	E	F	…	R	…
Shift T(c)	4	2	6	6	1	6	6	3	6

```
T: JIM SAW ME IN A BARBERSHOP
P: BARBER
       BARBER
        BARBER
              BARBER
                BARBER
                   BARBER
```

In running only make 12 comparisons, less than the length of the text! (24 chars)

Worst case scenario?

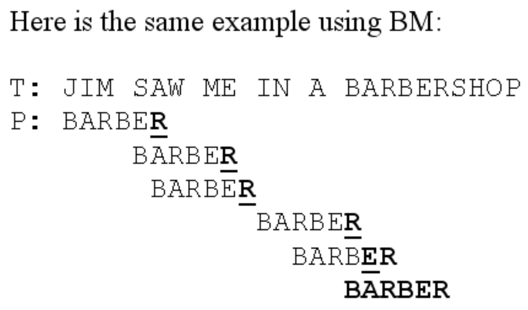
Boyer Moore

- Similar idea to Horspool's algorithm in that comparisons are made right to left, but is more sophisticated in how to shift the pattern.
- Using the "bad symbol" heuristic, we jump to the next rightmost character in P matching the char in T:

Example:

```
T=    BUBAERBARBERBARB
P=    BARBER
P=      BARBER
```

Boyer Moore Example



Here is the same example using BM:

```
T: JIM SAW ME IN A BARBERSHOP
P: BARBER
       BARBER
        BARBER
              BARBER
                BARBER
                   BARBER
```

Also uses 12 comparisons.

However, the worst case is O(nm+|∑|): requires O(m+| ∑ |) to compute the last-bad character, and we could run into same worst case as the naïve brute force algorithm (consider P=aaaa, T=aaaaaaaa…).