COS 226 Lecture 12: String searching

TEXT: N characters

PATTERN: M characters

(C occurrences of pattern in text)

Existence

Any occurrence of pattern in text?

Enumerate

How many occurrences of pattern in text?

Search

Find an occurence of pattern in text

Search all

Find all occurences of pattern in text

Three parameters N, M, C

start with N >> M >> C

Ex: N = 100,000, M = 100, C = 5

Other factors

multiple patterns (preprocessing)

binary vs. ascii text

avoid backup in text

String searching examples

Text string: find gjkmxorzeoa in

kvjlixapejrbxeenpphkhthbkwyrwamnugzhppfx
iyjyanhapfwbghxmshwrlyujfjhrsovkvveylnbx
nawavgizyvmfohigeabgksfnbkmffxjffqbualey
tqrphyrbjqdjqavctgxjifqgfgydhoiwhrvwqbxg
rixydzbpajnhopvlamhhfavoctdfytvvggikngkw
zixgjtlxkozjlefilbrboignbzsudssvqymnapbp
qvlubdoyxkkwhcoudvtkmikansgsutdjythzlapa
wlvliygjkmxorzeoafeoffbfxuhkzukeftnrfmoc
ylculksedgrdivayjpgkrtedehwhrvvbbltdkctq

Binary: find 101001111010111011 in

1000100101011010001001101011010110110110
0111111110101001110101000000010111101001
1011100000011000110000100110000111101101
1100000011011110101111101111000111100001
1011111001110011001111011001000000011101
1111100001011001110011010010110101001001
0001111111111011100111101100101010011110
1111001110110010010010011000100010011100
0010000011100010101110101001101100100011

Long text and pattern

* fixed pattern, random text: return "not found"
* fixed text, random pattern: return "not found"

Brute-force string searching

Check for pattern at every text position

```
int brutesearch(char *p, char *a)
  {
    int i, j, cnt = 0;
    for (i = 0; i < strlen(a); i++)
      for (j = 0; j < strlen(p); j++)
        {
          if (a[i+j] != p[j]) break;
          if (j == strlen(p)-1) return i;
        }
    return strlen(a)+1;
  }
```

DON’T USE THIS PROGRAM! (Why?)

In C, strlen takes time proportional string length
- running time at least N^2
- even for simpler programs (count the a’s)

Performance matters in ADT design!

Exercise: implement string ADT with fast strlen
- need space to store length
- need to update length when changing string
- ...

Brute-force algorithm (continued)

Easy alternative (if bug is discovered!)

```
int brutesearch(char *p, char *a)
  { int i, j;
    int M = strlen(p), N = strlen(a), cnt = 0;
    for (i = 0; i < N; i++)
      for (j = 0; j < M; j++)
        {
          if (a[i+j] != p[j]) break;
          if (j == M-1) return i;
        }
    return N+1;
  }
```

Different implementation (same algorithm)

char match: increment both i and j

char mismatch: set j to 0, reset i

```
int brutesearch(char *p, char *a)
  {
    int i, j, M = strlen(p), N = strlen(a);
    for (i = 0, j = 0; j < M && i < N; i++, j++)
      while (a[i] != p[j]) { i -= j-1; j = 0; }
    if (j == M) return i-M; else return i;
  }
```

Analysis of brute-force algorithm

Running time depends on text and pattern

Worst case: search for 000001

```
.    000000000000000000000000000001
.    00000*
.     00000*
.      00000*
.       00000*
.        00000*
```

    M*N bit compares

Average case (fixed pattern, random text)

```
.    10011010010100010100111000111
.    *
.     00*
.      0*
.       *
.        *
.         0*
.          *
.           001
```

long pattern: 2*N compares

short pattern:

* precise cost depends on pattern
* (first 001 appears before first 000, on average)

Ref: Flajolet and Sedgewick

Rabin-Karp algorithm

Idea: Use hashing

* compute hash function for each text position
* no table! (just compare with pattern hash)

Ex: "table" size 97, M = 5

Search for 15926 = 18 (mod 97)

```
.      31415926535897932384626433
.      31415 = 84 (mod 97)
.       14159 = 94 (mod 97)
.        41592 = 76 (mod 97)
.         15926 = 18 (mod 97)
.          59265 = 95 (mod 97)
```

Problem: hash function depends on M characters

Solution: compute hash for i+1 from hash for i

(improves running time from N*M to N+M)

```
31415 = 84  (mod 97)
14159 = (31415 - 30000)*10 + 9
      = (84 - 3*9)*10 + 9 (mod 97)
      = 579 = 94 (mod 97)
41592 = (94 - 1*9)*10 + 2 = 76 (mod 97)
15926 = (76 - 4*9)*10 + 6 = 18 (mod 97)
59265 = (18 - 1*9)*10 + 5 = 95 (mod 97)
```

Problem: need full compare on collision

```
92653 = (95 - 5*9)*10 + 3 = 18 (mod 97)
```

Solution: use giant (virtual) table!

RK algorithm implementation

Take table size as big as possible
(not so big that overflow happens)

```
#define q 3355439
#define d 256
int rksearch(char *p, char *a)
  {
    int i, j, dM = 1, h1 = 0, h2 = 0;
    int M = strlen(p), N = strlen(a);
    for (j = 1; j < M; j++) dM = (d*dM) % q;
    for (j = 0; j < M; j++)
      {
        h1 = (h1*d + p[j]) % q;
        h2 = (h2*d + a[j]) % q;
      }
    for (i = M; i < N; i++)
      {
        if (h1 == h2) return i-M;
        h2 = (h2 + d*q - a[i-M]*dM) % q;
        h2 = (h2*d + a[i]) % q;
      }
    return N;
  }
```

Randomized algorithm:
take random (big) table size

Knuth-Morris-Pratt algorithm

Text characters that match are in the pattern!
Can precompute what to do on mismatch

Ex: searching for 000001
- mismatch 00000* implies
  - * text had 000000
  - * if next text char is 1, full match found
  - * if next text char is 0, back in same state
- mismatch 000* implies
  - * text had 0001
  - * restart search on next text char

Finite-state automaton for 000001

```
.      0   1   2   3   4   5
.   0  1   2   3   4   5   5
.   1  0   0   0   0   0   6
```

worst-case text

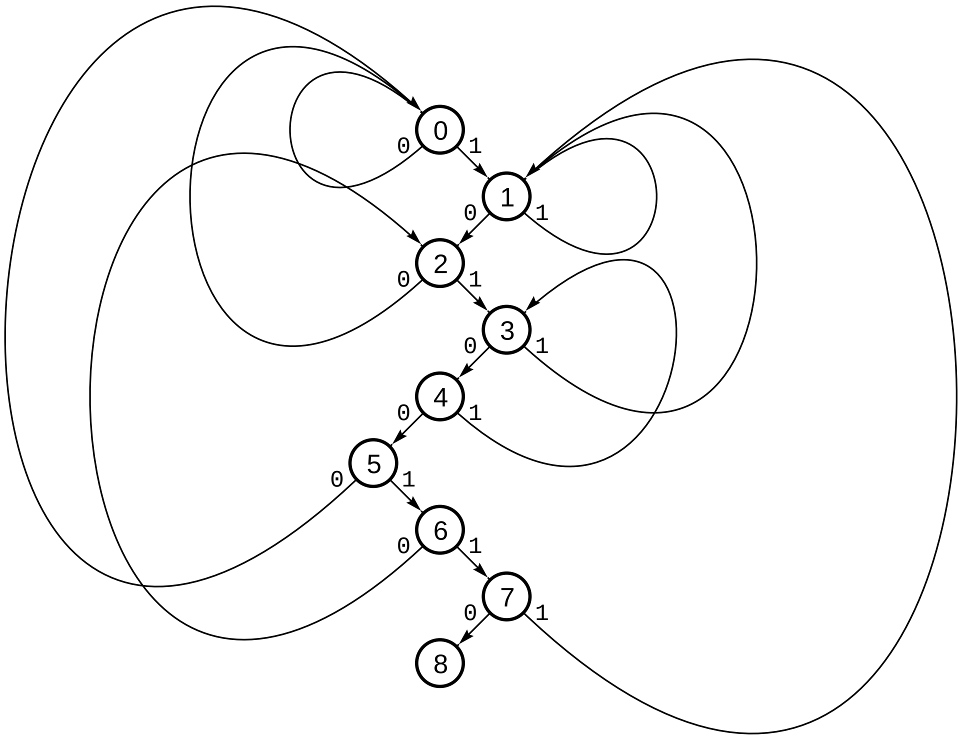
```
.   000000000000000000000000000001
.   012345555555555555555555555556
```

random text

```
.   100111010010100010100111000111
.   001200010120101230101200012300
```

KMP algorithm (continued)

Finite-state automaton for 10100110



.		**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
.	**0**	**0**	**2**	**0**	**4**	**5**	**0**	**2**	**8**
.	**1**	**1**	**1**	**3**	**1**	**3**	**6**	**7**	**1**

Search example:

.   100111010010101010100110000111

.   012011123456234343434345678

KMP algorithm:

- build FSA (or C program) from pattern
- run FSA on text
- N+M running time
- no backup in text

KMP automaton construction

FSA builds itself (!)

Successors to state i
* match: i+1
* mismatch:
  FSA state corresponding to
  * next putative text match
  * take mismatch into account

Ex: FSA for 10100110

State 6

    1010011: go to state 7
    1010010: go to state for 010010 (state 2)
        remember state for 010011 (X = state 1)

State 7

    10100110: go to state 8
    10100111: go to state for 0100111 (state 1)
        remember state for 0100110 (X = state 2)

main point: these are successors of old X

Algorithm:

```
    X: mismatch state
    match: FSA[match][i] = i+1
    mismatch:
        FSA[mismatch][i] = FSA[mismatch][X]
        X = FSA[match][X]
```

KMP automaton construction example

Pattern: 10100110

```
.       0   1
.   0   0   2     1
.   1   1   1     01

.       0   1   2
.   0   0   2   0     00
.   1   1   1   3     000

.       0   1   2   3
.   0   0   2   0   4     011
.   1   1   1   3   1     0011

.       0   1   2   3   4
.   0   0   2   0   4   5    0101
.   1   1   1   3   0   3    00123

.       0   1   2   3   4   5
.   0   0   2   0   4   5   0    01000
.   1   1   1   3   0   0   6    001200

.       0   1   2   3   4   5   6
.   0   0   2   0   4   5   0   2    010010
.   1   1   1   3   0   0   6   7    0012012

.       0   1   2   3   4   5   6   7
.   0   0   2   0   4   5   0   2   8    0100111
.   1   1   1   3   0   0   6   7   1    00120111
```

KMP implementation

Build FSA from pattern, run FSA on text string

Improvement:

- match state in FSA unnecessary (always i+1)
- use mismatch state only ("next" array)
- small hack: next[0] = -1

```
int kmpsearch(char *p, char *a)
{
  int i, j, M = strlen(p), N = strlen(a);
  initnext(p);
  for (i = 0, j = 0; j < M && i < N; i++, j++)
    while ((j>=0) && (a[i]!=p[j])) j = next[j];
  if (j == M) return i-M; else return i;
}
```

Ultimate search program for pattern:

machine language version of FSA

```
int kmpsearch(char *a)
{ int i = 0;
s0: if (a[i] != '1') goto s0; i++;
s1: if (a[i] != '0') goto s1; i++;
s2: if (a[i] != '1') goto s0; i++;
s3: if (a[i] != '0') goto s1; i++;
s4: if (a[i] != '0') goto s3; i++;
s5: if (a[i] != '1') goto s0; i++;
s6: if (a[i] != '1') goto s2; i++;
s7: if (a[i] != '0') goto s1; i++;
    return i-8; }
```

Right-left pattern scan

Can get sublinear algorithms by moving
  right to left in pattern
  (left to right in text)

Ex: Find string of 9 consecutive 0s

```
.    100111010010100010100111000111
.            0
.           1
.                    1
.                             0
.                            0
.                           1
```

Same idea effective in large alphabet
  for every character in alphabet
  compute skip when mismatch on that char
  (skip at least to next text char!)

Search time proportional to N/M
  for practical problems
Time decreases as pattern length increases (!)

Yet another parameter: alphabet size
  binary: 2 (group bits for this alg)
  text string: 256 (probably less)

Right-left pattern scan examples

Text character not in pattern

skip forward M chars

```
. now is the time for all good people to come
.       *      *      *      *      e
.                                  l
.                                 p
.                                o
.                               e
.                              p
```

Text character in pattern

skip forward to putative pattern end

```
. you can fool some of the people some of the time
.       *      *      *      o
.                               e
.                              l
.                             p
.                            o
.                           e
.                          p
```

Right-left pattern scan implementation

```
initskip(char *p)
  { int j, M = strlen(p);
    for (j = 0; j < 256; j++) skip[j] = M;
    for (j = 0; j < M; j++) skip[p[j]] = M-j-1;
  }

#define max(A, B) (A > B) ? A : B;
int mischarsearch(char *p, char *a)
  { int i, j;
    int M = strlen(p), N = strlen(a);
    initskip(p);
    for (i = M-1, j = M-1; j >= 0; i--, j--)
      while (a[i] != p[j])
        {
          i += max(M-j, skip[a[i]]);
          if (i >= N) return N;
          j = M-1;
        }
    return i+1;
  }
```

Boyer–Moore algorithm

mismatched character heuristic

KMP–like computation for skip

(pick the largest)

Multiple patterns and text

Related to symbol table ADT

DICTIONARY

build trie from text (preprocess)

pattern lookups in O(lgN) steps

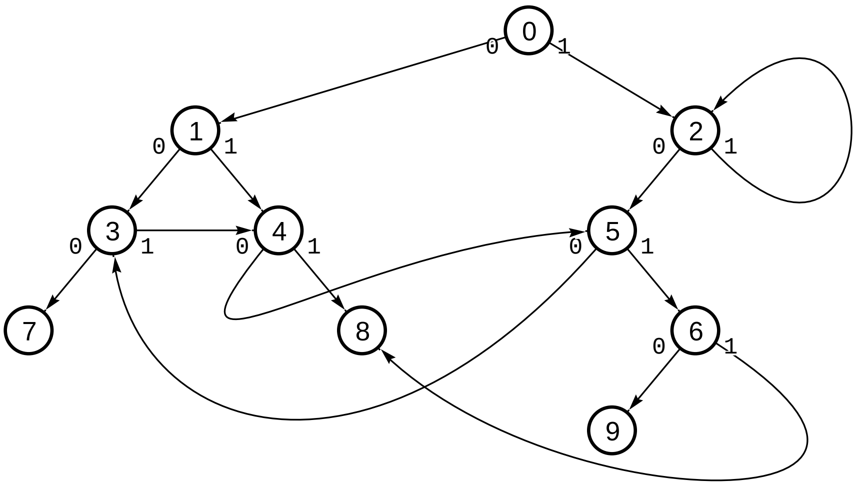
EXCEPTION DICTIONARY

build trie from set of patterns (preprocess)

find patterns in given text in N steps

generalization of KMP

Ex: FSA for 000, 011, and 1010



.		**0**	**1**	**2**	**3**	**4**	**5**	**6**
.	**0**	**1**	**3**	**5**	**7**	**5**	**3**	**9**
.	**1**	**2**	**4**	**2**	**4**	**8**	**6**	**8**

Simultaneous search for all patterns in text

. **111100100100101110100000**

. **0222253453453 4568**