COS 226 Lecture 12: String searching

TEXT: N characters
PATTERN: M characters

Existence: Any occurrence of pattern in text?
Enumerate: How many occurrences of pattern in text?
Search: Find an occurrence of pattern in text
Search all: Find all occurrences of pattern in

Three parameters N, M, C (number of occurrences)
- 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
kvjlixaepjrzxeenpkkhthhkwywamnumugzhppfx
iyjyjanhapfwbghxmskswlyujjfhhrsovkvevylbxx
nawavgizyvmfohigeabgksfnbkmffxjffqbualey
tgqphyrbjdjqaqcgtgjijifqgfygydhoiwhrvwqbxg
rxydzbajnhopvlamhffavocdfytvvgqikngkw
zikgtlxkojflilbrboignbsudssvqymnabpp
qvlubdoyskkkwhcordvktmkikansgtsudjythzlapa
wlvligyjkmxorzeoafoffbfuxhkukeftnrfmcq
ylculksedgrdivayjgpkrtedehhrvbbldtkctq

Binary: find gjkmxorzeoa in
100100100101101011100100111101101101110110
01111111111011001100010011011011011011011
01101100001110110011001100110011011011011
110000001101111011111101111111011111110110
1011110110111011011011011011111111111111111
111110000101101100110011011001100110011001
110011111111011110110111011110111111011111
111100111110110011001100110011001100110011
00100000110011010111110101101101110010001

String search analysis

Idea to test algorithms:
- use random pattern or random text

Ex: binary text
- ~N possible different patterns
- Probability of search success < N/2^M
- < .00000000000000000000001 for N = 100,000, M = 100
Probabilities are much lower for bigger alphabets

NOT FOUND is a simple algorithm that works

Better idea to test algorithms:
- search for fixed patterns in fixed text
- use random position for successful search
- use random perturbations for unsuccessful search

Brute-force string searching

Check for pattern at every text position

```c
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)) return i;
    return strlen(a)+1;
}
```

DON'T USE THIS PROGRAM!
Problem with brute-force implementation

for (i = 0; i < strlen(a); i++)

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

PERFORMANCE BUG

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 (bug fixed)

```c
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;
}
```

Brute-force algorithm (alternative)

Different implementation (same algorithm)
- char match: increment both \( i \) and \( j \)
- char mismatch: set \( j \) to 0, reset \( i \)

```c
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*
   00000*
```

\( M \times N \) bit compares

[Too slow when \( N \) and \( M \) are both large]
Average-case analysis

Fixed pattern, random text
100110100100010100111000111
  *  
  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 = 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: hash function depends on M characters
  - (running time N*M)

Rabin-Karp algorithm (continued)

Solution: use previous hash to compute next hash

92653 = 95 - 5*9 * 10 + 3 = 18 (mod 97)
Solution: use giant (virtual) table!
  - limit on table size: overflow on arithmetic ops

RK algorithm implementation

#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 + a[i]) % q;
    }
    return N;
}
Randomized algorithm: take random (huge) table size
Knuth-Morris-Pratt algorithm

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

Ex: when searching for 000001, mismatch 00000* implies
  • text had 00000
  • if next text char is 1, full match found
  • if next text char is 0, back in same state

Ex: when searching for 000001, mismatch 000* implies
  • text had 0001
  • restart search on next text char

More complicated in general, but
  • can always move to next text character
  • next action uniquely determined

Nonobvious algorithm
  • solved open theoretical and practical problems

KMP finite state automaton

“next action uniquely determined”
  • Build FSA from pattern
  • Run FSA on text (inner loop: $S = \text{FSA}[S][\text{a}[i++]]$)

$N+M$ running time
NO BACKUP in text

Ex: FSA for 000001

  0 1 2 3 4 5
  0 1 2 3 4 5
  1 0 0 0 0 0 6

worst-case text
  0000000000000000000000000000000001
  0123455555555555555555555555555556

random text
  1001110100101010101010011000111
  012011123456234343434345678

KMP automaton construction

FSA builds itself (!)
  • mismatch state is defined earlier in FSA

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: states needed are successors of old X
  • X: mismatch state
  • match: $\text{FSA[match]}[i] = i+1$
  • mismatch: $\text{FSA[mismatch]}[i] = \text{FSA[mismatch]}[X]$
  • $X = \text{FSA[match]}[X]$
KMP automaton construction example

Pattern: 10100110

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

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

```c
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;
}
```

initnext function constructs automaton (see text)

KMP implementation (continued)

Easy to create specialized C program for pattern

```c
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; }
```

Ultimate search program for pattern:
machine language version of FSA

Right-left pattern scan

SUBLINEAR ALGORITHMS
- Move right to left in pattern
- Move left to right in text

Ex: Find string of 9 consecutive 0s

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

Same idea effective in large alphabet

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

```c
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;
    }
```

```c
#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: KMP-like computation to improve skip, if possible

**Multiple patterns**

**DICTIONARY** (symbol table ADT)
- 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 lg N steps

**generalization of KMP**
- build trie from set of patterns
- convert to FSA with KMP-like computation
- find patterns in given text in N steps

**Multiple patterns example**

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
- 11100100100101110100000
- 0222534534534568

```
Ex: FSA for 000, 011, and 1010
```