String Search

Brute force

Rabin-Karp

Knuth-Morris-Pratt

Right-Left scan

String Searching

Text with N characters

Pattern with M characters

- match existence: any occurence of pattern in text?
- enumerate: how many occurences?
- match: return index of any occurence  focus of this lecture
- all matches: return indices of all occurences
-

Sample problem: find `avoctdfytvv` in

```
kvjlixapejrbxeenpphkhthbkwyrwamnugzhppfxiyjyanhapfwbghx
mshrlyujfjhrsovkvveylnbxnawavgizyvmfohigeabgksfnbkmffxj
ffqbualeytqrphyrbjqdjqavctgxjifqgfgydhoiwhrvwqbxgrixydz
bpajnhopvlamhhf          ggikngkwzixgjtlxkozjlefilbrboi
gnbzsudssvqymnapbpqvlubdoyxkkwhcoudvtkmikansgsutdjythzl
apawlvliygjkmxorzeoafeoffbfxuhkzukeftnrfmocylculksedgrd
ivayjpgkrtedehwhrvvbbltdkctq
```

Assume that N >> M >> number of occurences

Ex: N = 100,000; M = 100; five occurences

String searching context

Find M-char pattern in N-char text

Applications

- word processors
- virus scanning
- text information retrieval (ex: Lexis/Nexis)
- digital libraries
- computational biology
- web search engines

Theoretical challenge: linear-time guarantee

- suffix-trie index costs ~NlgN

Practical challenge: avoid BACKUP

- often no room or time to save input chars

Fundamental algorithmic problem

```
Now is the time for all people to come to
the aid of their party.Now is the time for
all good people to come to the aid of
theirparty.Now is the time for manygood
people to come to the aid of their
party.Now is the time for all
good people to come to the aid of their
party.Now is the time for a lot of
good people to come to the aid of their
party.Now is the time for all of the
good people to come to the aid of their
party.Now is the time for all good people
to come to the aid of their
party. Now is the time for each good
person to come to the aid of their
party.Now is the time for all good people
to come to the aid of their
party. Now is the time for all
good Republicans to come to the aid of
their party.Now is the time for all
good people to come to the aid of their
party.  Now is the time for many or all
good people to come to the aid of their
party. Now is the time for all good people
to come to the aid of their
party.Now is the time for all good
Democrats to come to the aid of their
party. Now is the time for all people to
come to the aid of their party.Now is the
time for all good people to come to the aid
of theirparty.Now is the time for manygood
people to come to the aid of their
party.Now is the time for all
good people to come to the aid of their
party.Now is the time for a lot of
good people to come to the aid of their
party.Now is the time for all of the
good people to come to the aid of their
party.Now is the time for all good people
to come to the aid of theirattack at dawn
party. Now is the time for each
person to come to the aid of their
party.Now is the time for all good people
to come to the aid of their
party. Now is the time for all
good Republicans to come to the aid of
their party.Now is the time for all
good people to come to the aid of their
party.  Now is the time for many or all
good people to come to the aid of their
party. Now is the time for all good people
to come to the aid of their
party.Now is the time for all good
Democrats to come to the aid of their
party.
```

Modelling String Searching

Random pattern or text??

Sample problem 1: find `unwillingly` in

```
kvjlixapejrbxeenpphkhthbkwyrwamnugzhppfxiyjyanhapfwbghx
mshrlyujfjhrsovkvveylnbxnawavgizyvmfohigeabgksfnbkmffxj
ffqbualeytqrphyrbjqdjqavctgxjifqgfgydhoiwhrvwqbxgrixydz
bpajnhopvlamhhfavoctdfytvvggikngkwzixgjtlxkozjlefilbrboi
gnbzsudssvqymnapbpqvlubdoyxkkwhcoudvtkmikansgsutdjythzl
```

← random text

Sample problem 2: find `avoctdfytvv` in ← random pattern

```
all the world's a stage and all the men and women merely
players. They have their exits and their entrances, and
one man in his time plays many parts.  At first, the infant,
mewling and puking in the nurse's arms.  Then the whining
schoolboy, with his satchel and shining morning face, creeping
like snail unwillingly to school. And then the lover, sighing
```

Simple, effective algorithm: return "NOT FOUND"

.0000000000000084 for sample problems

- probability of match is less than $N/(\text{alphabet size})^M$

Better to model fixed pattern in fixed text at random position

- swap patterns in sample problems 1 and 2 makes both OK
- use random perturbations to test mismatch

Brute-force string searching

Check for pattern at every text position

using array to simplify alg descriptions
online apps need getchar()-based implementation

```
int brutesearch(char p[], char a[])
  {
    int i, j;
    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);
  }
```

text loop

pattern loop

match

mismatch

- returns `i` if leftmost pattern occurence starts at `a[i]`
- returns `N` if no match

DO NOT USE THIS PROGRAM!

Problem with brute-force implementation

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

In C, `strlen` takes time proportional to string length

- evaluated every time through loop
- running time is at least N^2
- same problem for simpler programs (ex: count the blanks)

PERFORMANCE BUG

Textbook example: Performance matters in ADT design

Exercise: implement string ADT with fast `strlen`

- need space to store length
- need to update length when changing string
- might slow down some other simple programs

Brute-force string searching (bug fixed)

Check for pattern at every text position

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

lengths won't change; precompute them

text loop

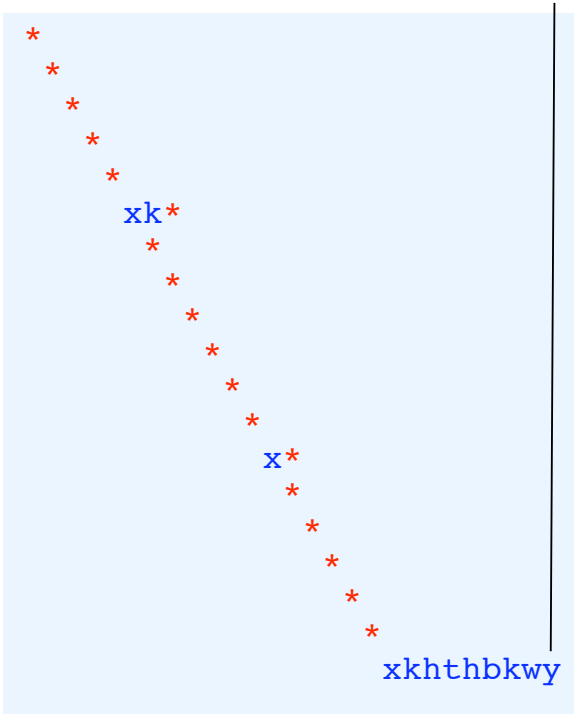
pattern loop

- returns `i` if leftmost pattern occurence starts at `a[i]`
- returns `N` if no match

Brute-force typical case

pattern: xkhthbkwy

text: kvjlixkpejrbxeenppxkhthbkwy



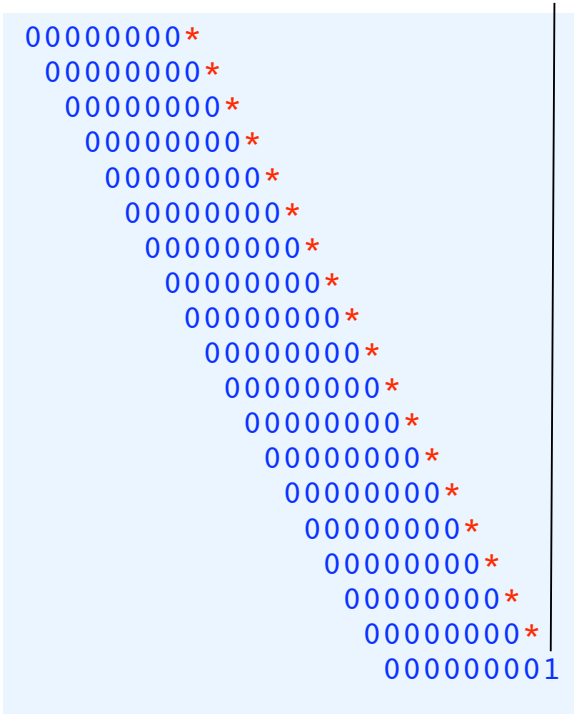
character compares: N+3

Can we guarantee performance?

Brute-force worst case

pattern: 000000001

text: 00000000000000000000000000001



Backs up in text:
need to retain M-char buffer

character compares: M*N

Too slow when M and N are large

Rabin-Karp algorithm

Idea 1: Use hashing

- compute hash function for each text position
- NO TABLE needed: just compare with pattern hash

Example: search for `59265` in `3141592653589793238462643`3

pattern hash: `59265 = 95 (mod 97)`

text hashes: `3141592653589793238462643`3

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

Problem: Hash uses M characters, so running time is N*M

Rabin-Karp algorithm (continued)

Idea 2: Use hash for previous position to compute hash

```
   14159      = (   31415      -  30000       )*10 + 9

14159 mod 97 = (31415 mod 97 - 30000 mod 97)*10 + 9 (mod 97)
                      known from
                      previous position
                                  precompute 9 = 10000 (mod 97)
               = (    84         -    3*9       )*10 + 9 (mod 97)

               = 579 mod 97 = 94
```

Key point: all ops involve small numbers
No restriction on N and M

Example: search for 59265 in 3141592653589793238462643

pattern hash: 59265 = 95 (mod 97)

text hashes for 3141592653589793238462643:

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

Slight problem: Still need full compare on collisions

Easy fix: use giant (virtual) hash table. ← need table size >> N^2 (birthday paradox)

Implementation of Rabin-Karp algorithm

```
#define q 3355439
#define d 256
```

random q much larger than N^2
q much smaller than (alphabet size)^M
d*q smaller than max integer

radix hash

pattern hash

initial text hash

main search loop

add d*q to keep value positive

Example for
q = 97, d = 10

```
10000 (mod 97) = 9
  10*10*10*10
mod after each op

59265 (mod 97) = 95
  Horner's method
mod after each op

31415 (mod 97) = 84
  Horner's method
mod after each op
```

```
31415 (mod 97) = 84

95 != 84

84 - 3*9 (mod 97) = 57

57*10 + 9 (mod 97) = 94

14159 (mod 97) = 94
```

Randomized algorithms

A randomized algorithm uses random numbers to gain efficiency

- quicksort with random partitioning element
- randomized BSTs
- Rabin-Karp

Las Vegas algorithm

- expected to be fast
- guaranteed to be correct

Examples: quicksort, randomized BSTs, Rabin-Karp with match check

Monte Carlo algorithm

- guaranteed to be fast
- expected to be correct

Example: Rabin-Karp without match check

String search implementations cost summary

Search for an M-character pattern in an N-character text

	typical	worst
brute-force	$N^{\dagger}$	N*M
Rabin-Karp	$N^{\dagger}$	$N^{\dagger}$

† assumes appropriate model

† assumes system can produce "random" numbers

Do we need the assumptions?

Knuth-Morris-Pratt algorithm

Observation: On mismatch at pattern char `j` we know the previous `j-1` chars in the text (they are also in the pattern)

Idea: precompute what to do on mismatch

Example 1: mismatch `00000*` when searching for `000001` in binary text

- text had `000000`
- compare next text char with last pattern char

Example 2: mismatch `000*` when searching for `000001` in binary text

- text had `0001`
- compare next text char with first pattern char

char to check is completely deduced from pattern

KMP algorithm

- precompute table of pattern char to check on mismatch, indexed by pattern position
- set pattern index from table in inner loop on mismatch instead of always resetting to 0

Surprising solution to open theoretical and practical problem (1970s)

KMP examples

pattern: 10100110

text: 10011101001010101010011000111

```
10*                                 |
   1*                               |
    1*                              |
     101001*                        |
          1010*                     |
            1010*                   |
              1010*                 |
                10100110
01201112345623434345678
```

mismatch table

0	1	2	3	4	5	6	7
0	1	0	1	3	0	2	1

pattern: 00000001

text: 00100000100000000000001

```
00*                                 |
   00000*                           |
         0000000*                   |
          0000000*                  |
           0000000*                 |
            0000000*                |
             0000000*               |
              0000000*              |
               00000001
01201234501234567777777778
```

mismatch table

0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	7

blue: match
red: mismatch
black: implicit

N character compares in every case

KMP implementation

Check for pattern at every text position

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

lengths won't change; precompute them

text loop

pattern loop

- returns `i` if leftmost pattern occurence starts at `a[i]`
- returns `N` if no match

KMP implementation

Check for pattern at every text position

- char match: increment both `i` and `j`
- char mismatch: set `j` to `mismatch[j]`
  (special case: `j = 0`, just increment `i`)

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

Differs from brute-force in two very significant ways

- need to compute `next` table (stay tuned)
- text pointer never backs up

KMP mismatch table construction

Table builds itself (!!)

mismatch table

0	1	2	3	4	5	6	7
0	1	0	1	3	0	2	1

Idea 1: Simulate restart ala brute-force

Ex: `mismatch[6]` for `1010011`0

- if mismatch at `101001*` then text was `1010010`
- for `...1010010x`, `x` compares to `p[2]`
- note also: for `...1010011x`, `x` compares to `p[1]`

```
10011101001010101010011000111
     101001*
      1010010110
```

pattern: 10100110
text: 010010x

```
  *
   10*
      10
0012012
```

Idea 2: Remember simulation for previous entry

Ex: `mismatch[7]` for `1010011`0

- if mismatch at `1010011*` then text was `10100111`
- just noted: for `...1010011x`, `x` compares to `p[1]`
- for `...10100111x`, `x` compares to `p[1]` mismatch at `p[1]`
- for `...10100110x`, `x` compares to `p[2]` match at `p[1]`

pattern: 10100110
text: 010011x

```
  *
   10*
      1*
0012011
```

KMP mismatch table construction implementation

`mis[j]`: index of pattern char to compare against next text char on mismatch on jth pattern character

`t`: index of pattern char that brute-force algorithm would compare against next text char on iteration after mismatch

To compute `mis[j]`, compare `p[j]` with `p[t]`

match:

- `mismatch[j] = mismatch[t]` since mismatch action same as for `t`
- `t = t+1` since we know that brute-force algorithm will find match

mismatch: opposite assignment

```
t = 0; mismatch[0] = 0;
for (int j = 1; j < M; j++)
  if (p[j] == p[t])
    { mismatch[j] = mismatch[t]; t = t+1; }
  else
    { mismatch[j] = t+1; t = mismatch[t]; }
```

Computation more complicated for nonbinary alphabet

		0	1	2	3	4	5	6	7
		1	1	1	0	0	1	1	0
j	t								
	0	0							
1	0	0	1						
2	1	0	1	0					
3	2	0	1	0	1				
4	0	0	1	0	1	3			
5	1	0	1	0	1	3	0		
6	1	0	1	0	1	3	0	2	
7	2	0	1	0	1	3	0	2	1

Optimized KMP implementation

Easy to create specialized program for given pattern
(build in mismatch table)

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

pattern 10100110

mismatch table

Ultimate search program for specific pattern:
compile directly to machine code

String search implementations cost summary

Search for an M-character pattern in an N-character text

	typical	worst	
brute-force	$N^{\dagger}$	N*M	
Rabin-Karp	$N^{\dagger}$	$N^{\dagger}$	← inner loop with several arithmetic instructions
Knuth-Morris-Pratt	N	N	← tiny inner loop

† assumes appropriate model

† assumes system can produce "random" numbers

KMP is optimal. Can we do better?

Right-left pattern scan

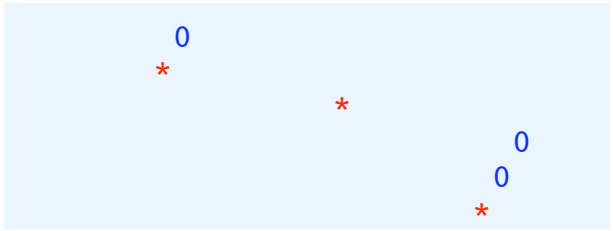
Sublinear algorithms

- move right to left in pattern
- move left to right in text

Q: Does binary string have 9 consecutive 0s?

pattern: 000000000

text: 1001110100101000101001110001111



A: No. (Needed to look at only 6 of 30 chars.)

Idea effective for general patterns, larger alphabet

Search time proportional to N/M for practical problems

Time decreases as pattern length increases (!)

Right-left scan examples

Text char not in pattern: skip forward M chars

pattern: `people`

text: `now is the time for all good people`

Text char in pattern: skip to end of pattern

pattern: `people`

text: `you can fool some of the people some of`

Boyer-Moore algorithm:
figure out best skip ala KMP

Implementation of right-left pattern scan

```
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 M = strlen(p), N = strlen(a);
    int i, j;
    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;
  }
```

build skip table

right-to-left scan

restart at right end of pattern

main search loop

String search implementations cost summary

Search for an M-character pattern in an N-character text

	typical	worst
brute-force	$N^{\dagger}$	N*M
Rabin-Karp	$N^{\dagger}$	$N^{\dagger}$
Knuth-Morris-Pratt	N	N
Right-left scan	$N/M^{\dagger}$	N

beats optimal by a factor of 100 for M = 100

† assumes appropriate model

† assumes system can produce "random" numbers

String search summary

Ingenious algorithms for a fundamental problem

Rabin-Karp

- easy to implement
- extends to more general settings (ex: 2D search)

Knuth-Morris-Pratt

- quintessential solution to theoretical problem
- works well in practice, too (no backup, tight inner loop)

Right-left scan

- simple idea leads to dramatic speedup for long patterns

Tip of the iceberg (stay tuned)

- multiple patterns?
- wild-card characters?