Lecture 19: April 23, 2002

Pattern matching:

- Randomized (Karp-Rabin)
- Deterministic (Knuth-Morris-Pratt)

Pattern Matching

Input: Two strings

- $T[1 \ldots n]$ (the text)
- $P[1 \ldots m]$ (the pattern)

Assume all characters are just bits, for simplicity.

Task: Check if pattern appears in text and if so, report such an occurrence.

E.g., this is a match:

0	1	0	1	1	1	1	0	0	0
				1	1	1	0		

Natural algorithm

- Let $T^s = T[s+1 \ldots s+m]$, $s = 0, 1 \ldots$
- For every s check if $T^s = P$ (say, left to right)
- Running time: $\Theta(nm)$

More clever algorithm

- Stop checking $T^s = P$ when the *first* difference is detected
- Much better in practice
- $O(n+m)$ for *random* text and pattern

- Still $\Theta(nm)$ in the worst case

T:

0	0	0	0	0	1	1	1	1	1

P:

0	0	0	0	0	0

Our goal

- $O(n+m)$ worst case
- By modifying the natural algorithm
  - Check $T^s = P$ more efficiently using hashing
  - Check $T^s = P$ only for *important* symbols

First algorithm

- Consider the m-bit strings T^s and P
- Can think of them as of *long* integers; we just need to check their equality
- Problem: the integers are large (m-bit long)
  $\Rightarrow$ cannot compare in constant time
- Use hashing to reduce their length ! (to $O(\log n)$)

Hashing

Want a function h such that

1. For all s we have $h(P) = h(T^s)$ if and only if $P = T^s$
2. We can compute all values $h(T^s)$ in linear time

How to find h ? E.g.,

$$h(x) = ax + b \bmod q$$

(a, b random) satisfies (1) but not (2) !

Karp-Rabin approach

- For $R = R[0 \ldots m-1]$ define
  $h(R) = R[0]2^0 + R[1]2^1 + \ldots R[m-1]2^{m-1} (\text{mod } q)$
  where q is a random prime
- Can easily compute $h(T^s)$ given $h(T^{s-1})$, since
  $h(T^s) = 1/2(h(T^{s-1}) - T[s]) + 2^{m-1}T[m+s] (\text{mod } q)$
  i.e., shift the number to the left and add new most significant bit

  Note: since the computation is done modulo q, we use $2^{m-1} \bmod q$ instead of 2^{m-1}.

- Runs in linear time if q is $O(\log n)$ bits long
  $\Rightarrow$ property (2) satisfied

Collisions

- What kind of primes are bad? Those s.t. $T^s \neq P$ but $h(T^s) = h(P)$
- If $h(T^s) - h(P) = 0$, then q must divide $x = T^s - P$
- There are at most $\log_2(|x|) \leq m$ such primes
  - product of any k primes has value $> 2^k$
  - $|x| \leq 2^m$
- Random prime from a set of $2nm$ primes is bad for fixed s with probability $< \frac{1}{2n}$
- ... is bad for *any* s with probability $< \frac{n}{2n} = 1/2$

Analysis ctd.

Still need to show:

- How to pick a random prime from a set of $2nm$ primes ?
- Is the prime a short integer ?

Questions are left to the careful reader ...

Conclusion:

- Running time $O(n + m)$
- Probability of correctness $1/2$

Second algorithm

T:

0	0	0	0	0	1	1	1	1	1

P:

0	0	0	0	0	0

Problems:

- We check $T^s = P$ even when we read enough symbols to *know* it is false
- Should "recycle" the information
- Two ideas:
  - Increment s by a lot if you can
  - Do not redo the comparisons already made

“Little birdie” approach

- The algorithm performs a check if $T^s = P$
- If the comparison $T^s = P$ fails (i.e., $T^s[k] \neq P[k]$), the birdie tells us the smallest $s' > s$ so that based on the symbols read so far, it *could* be the case that $T^{s'} = P$
- Formally, the birdie returns the largest *suffix* of $T^s[1 \ldots k]$ which is a *prefix* of P
- We proceed to checking if $T^{s'} = P$, starting from the first *unread* position

0	1	0	1	1	1	1	0	0	0

Analysis

- Running time: $O(n)$ comparisons and birdie queries
- Implementation of the little birdie (i.e., how to find quickly the largest suffix of $T^s[1 \ldots k]$ which is a prefix of P):
  - We prepare the birdie for a specific pattern P
  - Naively, our birdie could have exponential size
  - However, $T^s[1 \ldots k] = P^s[1 \ldots k-1]T^s[k]$ $\Rightarrow$ the birdie only needs to know $k, T^s[k]$
  - Thus, the birdie can be implemented as an array $B_P[1 \ldots m, 0 \ldots 1]$
  - *Given P, the birdie B_P can be computed in time $O(m)$ (for binary alphabet)
- Overall running time: $O(n+m)$

For fans of pattern matching

- Boyer-Moore algorithm
  - $O(n+m)$ worst case
  - $O(\frac{n}{m}\log n)$ average case
- Galil-Seiferas
  - in place (i.e., no birdies)