Rabin-Karp matching

Goldsmiths Computing

Motivation

- naïve string matching takes time in $\Theta(mn)$
- lots of wasted work

Naïve algorithm

```
function MATCH(T,P)
    m ← LENGTH(P)
    for 0 ≤ s ≤ LENGTH(T) - m do
        found ← true
        for 0 ≤ j < m do
            if T[s+j] ≠ P[j] then
                found ← false; break
            end if
        end for
        if found then
            return s
        end if
    end for
    return false
end function
```

Less work in the inner loop

- avoid $\Theta(m)$ comparisons where possible
- constant-time test:
    - hash value comparison

Rabin-Karp algorithm

```
function RKMATCH(T,P)
    m ← LENGTH(P); hm ← HASH(P)
    for 0 ≤ s ≤ LENGTH(T) - m do
        if HASH(T[s...s+m]) = hm then
            found ← true
            for 0 ≤ j < m do
                if T[s+j] ≠ P[j] then
                    found ← false; break
                end if
            end for
            if found then
                return s
            end if
        end if
    end for
    return false
end function
```

Hash function

Normally:

- HASH(T[s...s+m]) takes time in $\Theta(m)$
- no saved work in general

Rolling hash

Clever choice of hash function makes a difference!

- ROLLING-HASH(h,T[s-1],T[s+m])

Examples of suitable hash functions

modular add $\sum_i x_i \bmod k$

exclusive or $\oplus_i x_i$

modular polynomial $\sum_i x_i p^i \bmod k$

Modular add

$$\sum_i x_i \bmod k$$

- 21-bit characters: k might be 2^{24} or 2^{32}
  - (resist temptation to use 8-bit characters and k of 2^8)

function ROLLING-HASH(prev,remove,add)
    return (prev - remove + add) mod k
end function

- extremely limited bit mixing
- high chance of hash collisions in typical texts
  - *e.g.* HASH(ab) = HASH(ba)

Exclusive or

$$\oplus_i x_i$$

- no parameters
    - (still need to resist temptation to use 8-bit characters)

function ROLLING-HASH(prev,remove,add)
    return prev $\oplus$ remove $\oplus$ add
end function

- no bit mixing at all
- high chance of hash collisions in typical texts
    - *e.g.* HASH(`oboe`) = HASH(`bell`)

Modular polynomial

$$\sum_i x_i p^i \bmod k$$

- typically choose a small(ish) prime p
- use machine word (*e.g.* 2^{32}) for k

function ROLLING-HASH(prev,remove,add)
    return $\left((\text{prev} - \text{remove} \times p^{m-1}) \times p + \text{add}\right) \bmod k$
end function

- good mixing (*e.g.* for prime $p = 101$, character bits 0-7 affect hash bits 0-13)
- hash collisions in typical texts rarer

Complexity analysis

space

no need for extra space that scales with any parameter

$$\Longrightarrow \Theta(1)$$

time

- for good rolling hash:
  - new hash computation from old hash in $\Theta(1)$ time
  - hash collisions rare (still need to do at least two $\Theta(m)$ hash computations)

$$\Longrightarrow \Theta(n) + \Theta(m) \text{ (average case)}$$

- even for the best hash function...
  - ...suitably adversarial input will collide a lot

$$\Longrightarrow \Theta(nm) \text{ (worst case)}$$

Work

1. Reading
   - CLRS, section 32.2
2. Questions from CLRS
   - Exercise 32.2-2
3. Lab work
   - (week of 3rd December) implement Rabin-Karp string match for strings of characters. Use `OpCounter` to count how many character comparisons happen in the best and worst case. Construct a table and verify the theoretical results in this lecture.