Hash tables: Introduction

IP Access List

Analyze the access log and quickly answer queries: did anybody access the service from this IP during the last hour? How many times? How many IPs were used to access the service during the last hour?

```
Main Loop:
log - array of log lines (time, IP)
C - mapping from IPs to counters
i - first unprocessed log line
j - first line in current 1h window
i <- 0
j <- 0
C <- empty
Each second
    UpdateAccessList(log, i, j, C)
```

```
UpdateAccessList(log, i, j, C):
while log[i].time <= Now(): # Unprocessed log line
    C[log[i].IP] <- C[log[i].IP] + 1 # C[log[i].IP] is initialized with 
0, now it becomes 1
    i <- i + 1
while log[j].time <= Now() - 3600: # 1 hour ago
    C[log[j].IP] <- C[log[j].IP] - 1 # C[log[j].IP] now becomes 0
    j <- j + 1
```

```
AccessedLastHour(IP, C):
return C[IP] > 0
```

Direct addressing

- Convert IP to 32-bit integer
- Create an integer array A of size 2^{32}
- Use `A[int(IP)]` as `C[IP]`

```
int(IP): # Need 2^{32} memory even for few IP. It cannot be applied to 
IPv6: 2^{128} which exceeds memory limit. In general, O(N) memeory, N = 
|S|
return IP[1]*2^{24} + IP[2]*2^{16} + IP[3]*2^{8} + IP[4]
```

```
UpdateAccessList(log, i, j, A): # O(1) per log line
while log[i].time <= Now():
    A[int(log[i].IP)] <- A[int(log[i].IP)] + 1
    i <- i + 1
while log[j].time <= Now() - 3600:
    A[int(log[j].IP)] <- A[int(log[j].IP)] - 1
    j <- j + 1
```

```
AccessedLastHour(IP): # O(1)
return A[int(IP)] > 0
```

Asymptotic analysis of direct addressing

- For `IPv4`, it requires 2^{32} memory even for few IP. It has a huge memory requirement!
- `UpdateAccessList` process each log line at $O(1)$ cost
- `AccessedLastHour` return an element from an array which also takes $O(1)$

List-based Mapping

- Store only active IPs in a list
- Store only last occurrence of each IP
- Keep the order of occurrence

```
UpdateAccessList(log, i, L):
while log[i].time <= Now():
    log_line <- L.FindByIP(log[i].IP)
    if log_line != NULL:
        L.Erase(log_line) # Erase old record of log[i].IP
    L.Append(log[i]) # Update new record of log[i].IP
    i <- i + 1
while L.Top().time <= Now() - 3600:
    L.Pop()
```

```
AccessedLastHour(IP, L):
return L.FindByIP(IP) != NULL
```

Asymptotic analysis of list-based mapping

- Memory usage is $\Theta(n)$
- `L.Append, L.Top, L.Pop` are $\Theta(1)$
- `L.FindByIP, L.Erase` are $\Theta(n)$
- `UpdateAccessList` is $\Theta(n)$ per log line
- `AccessLastHour` is $\Theta(n)$

Hash function

For any set of objects S and any integer $m > 0$, a function $h : S \to \{0, 1, \ldots, m-1\}$ is called a hash function.

Cardinality

m is called the cardinality of hash function h

Collisions

When $h(o_1) = h(o_2)$ and $o_1 \neq o_2$, this is a collision.

Map

Map from S to V is a data structure with methods `HashKey(O)`, `Get(O)`, `Set(O, v)`, where $O \in S, v \in V$.

```
HasKey(O):
L <- A[h(O)]
for (O', v') in L:
    if O' == O:
        return true
return false
```

```
Get(O):
L <- A[h(O)]
for (O', v') in L:
    if O' == O:
        return v'
return n/a
```

```
Set(O, v):
L <- A[h(O)]
for p in L:
    if p.O == O:
        p.v <- v
        return
L.Append(O, v)
```

Lemma

Let c be the length of the longest chain in A. Then, the running time of `HasKey`, `Get`, `Set` is $\Theta(c + 1)$.

Proof:

If $L = A[h(O)]$, $\text{len}(L) = c, O \notin L$, need to scan all c items.

If $c = 0$, we still need $O(1)$ time

Lemma

Let n be the number of different keys O currently in the map and m be the cardinality of the hash function. Then, the memory consumption for chaining is $\Theta(n + m)$.

Proof:

$\Theta(n)$ to store n pairs (O, v) and $\Theta(m)$ to store array A of size m

Set

Set is a data structure with methods `Add(O), Remove(O), Find(O)`.

```
Find(O):
L <- A[h(O)]
for O' in L:
    if O' == O:
        return true
return false
```

```
Add(O):
L <- A[h(O)]
for O' in L:
    if O' == O:
        return
L.Append(O)
```

```
Remove(O):
if not Find(O):
    return
L <- A[h(O)]
L.Erase(O)
```

Hash tables: Hash functions

Good hash functions

- Deterministic (not random value)
- Fast to compute
- Distributes keys well into different cells
- Few collisions (Worse case will be $O(n)$)

Lemma

If number of possible keys is big ($|U| \gg m$), for any hash function h, there is a bad input resulting in many collisions

Universal family

Let U be the universe - the set of all possible keys. A set of hash functions

$$\mathcal{H} = \{h : U \to \{0, 1, 2..., m-1\}\}$$

is called a universal family if for any two keys $x, y \in U$, $x \neq y$ the probability of collision

$$\Pr(h(x) = h(y)) \leq \frac{1}{m}$$

Lemma

If h is chosen randomly from a universal family, the average length of the longest chain c is $O(1 + \alpha)$ where $\alpha = \frac{n}{m}$ is the load factor of the hash table.

Corollary

If h is from universal family, operations with hash table run on average in time $O(1 + \alpha)$.

How randomization works?

- Select a random function h from $\mathcal{H}$
- Fixed h is used throughout the algorithm

Choosing hash table size

- Control amount of memory used with m
- Ideally, load factor $0.5 < \alpha < 1$
- Use $O(m) = O(\frac{n}{\alpha}) = O(n)$ memory to store n keys
- Operations run in time $O(1 + \alpha) = O(1)$ on average

Dynamic hash tables

- If the number of keys n is unknown in advance, starting with a big hash table would cost a lot of memory
- Resize the hash table when α becomes too large (keep load factor below 0.9)
- Choose new hash function and rehash all the objects

```
Rehash(T):
loadFactor <- T.numberOfKeys / T.size
if loadFactor > 0.9:
    Create T_new of size 2 * T.size
    Choose h_new with cardinality T_new.size
    For each object O in T:
        Insert O in T_new using h_new
    T <- T_new
    h <- h_new
```

Hashing integers

Lemma

$\mathcal{H}_p = \{h_p^{a,b}(x) = ((ax + b) \mod p) \mod m\}$ for all $a, b : 1 \leq a \leq p - 1, 0 \leq b \leq p - 1$ is a universal family.

Example with hashing phone number

Select $a = 34, b = 2$, so $h = h_p^{34,2}$ and consider $x = 1482567$ corresponding to phone number 148-25-67. $p = 10000019$.

$$
\begin{aligned}
(34 \times 1482567 + 2) \mod 10000019 = 407185 \\
407185 \mod 1000 = 185 \\
h(x) = 185
\end{aligned}
$$

General case

- Define maximum length L of a phone number
- Convert phone numbers to integers from 0 to $10^L - 1$
- Choose prime number $p > 10^L$
- Choose hash table size m
- Choose random hash function from universal family $\mathcal{H}_p$ (choose random $a \in [1, p-1]$ and $b \in [0, p-1]$)

Hashing strings

String length

Denote by $|S|$ the length of string S.

Polynomial hashing

Family of hash functions

$$\mathcal{P}_p = \{h_p^X(S) = \sum_{i=0}^{|S|-1} S[i]x^i \mod p\}$$

with a fixed prime p and all $1 \leq x \leq p-1$ is called polynomial.

```
PolyHash(S, p, x): # O(|S|)
hash <- 0
for i from |S| - 1 down to 0:
    hash <- (hash * x + S[i]) mod p
return hash
```

Cardinality Fix

- For use in a hash table of size m, we need a hash function of cardinality m.
- First apply random h from $\mathcal{P}_p$, and then hash the resulting value again using integer hashing. Denote the resulting function by h_m.

Lemma

For any two different strings s_1 and s_2 of length at most $L + 1$, if you choose h from $\mathcal{P}_p$ at random (by selecting a random $x \in [1, p-1]$), the probability of collision $\Pr(h(s_1) = h(s_2))$ is at most $\frac{L}{p}$.

Proof idea

This follows from the fact that the equation $a_0 + a_1x + a_2x^2 + \ldots + a_Lx^L = 0 \mod p$ for prime p has at most L different solutions x.

Lemma

For any two different strings s_1 and s_2 of length at most $L + 1$ and cardinality m, the probability of collision $\Pr(h_m(s_1) = h_m(s_2))$ is at most $\frac{1}{m} + \frac{L}{p}$.

Corollary

If $p > mL$, for any two different strings s_1 and s_2 of length at most $L + 1$, the probability of collision $\Pr(h_m(s_1) = h_m(s_2))$ is $O(\frac{1}{m})$.

Proof:

$$\frac{1}{m} + \frac{L}{p} < \frac{1}{m} + \frac{L}{mL} = \frac{1}{m} + \frac{1}{m} = \frac{2}{m} = O(\frac{1}{m})$$

The second inequality comes from $p > mL$.

Hash tables: String search

Searching for Patterns

Given a text T (book, website, facebook profile) and a pattern P (word, phrase, sentence), find all occurrences of P in T.

Substring

Denote by $S[i \ldots j]$ the substring of string S starting in position i and ending in position j.

Find Pattern in Text

Input: Strings T and P

Output: All such positions i in T, $0 \le i \le |T| - |P|$ that $T[i \ldots i + |P| - 1] = P$.

Naive algorithm

```
AreEqual(S_1, S_2):
if |S_1| != |S_2|:
    return False
for i from 0 to |S_1| - 1:
    if S_1[i] != S_2[i]:
        return False
return True
```

```
FindPatternNaive(T, P):
result <- empty list
for i from 0 to |T| - |P|:
    if AreEqual(T[i...i + |P| - 1], P):
        result.Append(i)
return result
```

Lemma

Running time of `FindPatternNaive(T, P)` is $O(|T||P|)$.

Proof:

- Each `AreEqual` call is $O(|P|)$
- $|T| - |P| + 1$ calls of `AreEqual` total to $O((|T| - |P| + 1)|P|) = O(|T||P|)$

Rabin-Karp algorithm

- If $h(P) \neq h(S)$, then definitely $P \neq S$
- If $h(P) = h(S)$, call `AreEqual(P, S)`
- Use polynomial hash family $\mathcal{P}_p$ with prime p
- If $P \neq S$, the probability $\Pr(h(P) = h(S))$ is at most $\frac{|P|}{p}$ for polynomial hashing

```
RabinKarp(T, P):
p <- big prime
x <- random(1, p - 1)
result <- empty list
pHash <- PolyHash(P, p, x)
for i from 0 to |T| - |P|:
    tHash <- PolyHash(T[i...i+|P|-1], p, x)
    if pHash != tHash:
        continue
    if AreEqual(T[i...i+|P|-1], P):
        result.Append(i)
return result
```

False Alarms

- It is the event when P is compared with $T[i \ldots i + |P| - 1]$, but $P \neq T[i \ldots i + |P| - 1]$.
- The probability of false alarm is at most $\frac{|P|}{p}$
- On average, the total number of false alarms will be $(|T| - |P| + 1)\frac{|P|}{p}$, which can be made small by selecting $p \gg |T||P|$

Running time without `AreEqual`

- $h(P)$ is computed in $O(|P|)$

- $h(T[i \dots i+|P|-1])$ is computed in $O(|P|)$, $|T|-|P|+1$ times
- $O(|P|)+O((|T|-|P|+1)|P|)=O(|T||P|)$

`AreEqual` running time

- `AreEqual` is computed in $O(|P|)$
- `AreEqual` is called only when $h(P)=h(T[i \dots i+|P|-1])$, meaning either an occurrence of P is found or a false alarm happened
- By selecting $p \gg |T||P|$, we make the number of false alarms negligible

Total running time

- If P is found q times in T, then total time spent in `AreEqual` is $O((q+\frac{(|T|-|P|+1)|P|}{p})|P|)=O(q|P|)$ for $p \gg |T||P|$
- Total running time is $O(|T||P|)+O(q|P|)=O(|T||P|)$ as $q \leq |T|$
- It has the same running time as naive algorithm, but it can be improved

Recurrence of hashes

$$
\begin{aligned}
H[i+1] &= \sum_{j=i+1}^{i+|P|} T[j]x^{j-i-1} \mod p \\
H[i] &= \sum_{j=i}^{i+|P|-1} T[j]x^{j-i} \mod p \\
&= \sum_{j=i+1}^{j+|P|} T[j]x^{j-i} + T[i] - T[i+|P|]x^{|P|} \mod p \\
&= x\sum_{j=i+1}^{i+|P|} T[j]x^{j-i-1} + (T[i] - T[i+|P|]x^{|P|}) \mod p \\
H[i] &= xH[i+1] + (T[i] - T[i+|P|]x^{|P|}) \mod p
\end{aligned}
$$

```
PrecomputeHashes(T, |P|, p, x):
H <- array of length |T| - |P| + 1
S <- T[|T|-|P|...|T|-1]
H[|T|-|P|] <- PolyHash(S, p, x)
y <- 1
for i from 1 to |P|:
    y <- (y * x) mod p
for i from |T|-|P|-1 down to 0:
    H[i] <- (x * H[i + 1] + T[i] - y * T[i + |P|]) mod p
return H
```

`PrecomputeHashes` running time

- `PolyHash` is called once, $O(|P|)$
- First for loop runs in $O(|P|)$
- Second for loop runs in $O(|T|-|P|)$

- Total precomputation time $O(|T| + |P|)$

```
RabinKarp(T, P):
p <- big prime
x <- random(1, p - 1)
result <- empty list
pHash <- PolyHash(P, p, x)
H <- PrecomputeHashes(T, |P|, p, x)
for i from 0 to |T|-|P|:
    if pHash != H[i]:
        continue
    if AreEqual(T[i...i+|P|-1], P):
        result.Append(i)
return result
```

- $h(P)$ is computed in $O(|P|)$
- `PrecomputeHashes` runs in $O(|T| + |P|)$
- Total time spent in `AreEqual` is $O(q|P|)$ on average where q is the number of occurrences of P in T
- Average running time $O(|T| + (q+1)|P|)$