Basic Algorithm's lec 21

Notation:
- \( U \): "universe" of keys to hash
- \( K \): "index set" of hash family
- \( V \): "value" set to map the hash function

Family:
- \( \mathcal{H} = \{ h_k \}_{k \in K} \), \( h_k : U \rightarrow V \)

Recall universal hashing

\[ \left| \left\{ k \mid h_k(a) = h_k(b) \right\} \right| \leq \frac{|K|}{|V|} \]

Not always possible to get such a bound,

Definition of \( \varepsilon \)-universality

\[ \left| \left\{ k \mid h_k(a) = h_k(b) \right\} \right| \leq \varepsilon \cdot |K| \]

For universal hashing \( \varepsilon = \frac{1}{m=|V|} \)
Probabilistically
\[ P_r [ \text{hash}(a) = \text{hash}(b) ] \leq \varepsilon \]

---
Previous proof for expected block size
\[ \rightarrow 1 + \varepsilon \cdot n \]

---

E.g., Rolling Hash
\[ U = Z_p^{t+1}, K = Z_p, V = Z_p \]
\[ \text{hash}(x_0, \ldots, x_t) = \sum_{i=0}^{t} x_i \cdot K^i \]

Also called the polynomial hash function

Claim: This family is \( t/p \) universal

Proof: Let \[ \text{hash}(a) = \text{hash}(b) \]
\[ \rightarrow \sum a_i \cdot K^i = \sum b_i \cdot K^i \]
\[ \rightarrow \sum c_i \cdot K^i = 0 \]

\( c_i \) are the indices causing collision are the roots of this polynomial.
the polynomial is of degree \( \leq t \)

\[ \Rightarrow \# \text{roots} \leq t \]

\[ \Rightarrow \left\{ k \mid h_k(a) = h_k(b) \right\} \leq t = \frac{t}{p} \cdot p \]

\[ \Rightarrow \text{the family is a } \frac{t}{p} \text{ universal family} \]

\[ \text{E.g. another rolling hash} \]

\[ U = \mathbb{Z}_p^t, \quad K = \mathbb{Z}_p, \quad Y = \mathbb{Z}_p \]

\[ h_k(x_0, \ldots, x_t) = \frac{t}{11} (x_i + k) \]

\[ i = 0 \]

By a similar proof, this is also \( \frac{t}{p} \) universal.

These families are also called "weak" universal families.

for strong families (for given \( c \))

\[ \Pr_k \left[ h_k(a) = h_k(b) \right] \leq \frac{1}{mc} \]

This is a \( c \)-universal hash family.
Variable length input

\[ U = \bigcup_{i=0}^{t} \mathbb{Z}_p^+ \quad K = \mathbb{Z}_p \quad V = \mathbb{Z}_p \]

\[ h_k(x_0, \ldots, x_l) = k^{t+1} + \sum_{i=0}^{l} x_i \cdot k^i \]

* Polynomial with padding.

* is also \( \mathbb{T}_p \) universal.

Applications.

String finding in expected \( O(n) \).

Given a database of text of size \( n \) and a pattern of length \( m \), find the pattern in \( O(n \cdot m) \).

Naive method.

- Go to position \( i = 1, \ldots, n - m \).
- Check if next \( m \) characters form the pattern.

\( \rightarrow O(n \cdot m) \).
**Randomized - Las-Vegas.**

Find hash of each substring of length \( m \) and only check for matches.

* Still need to hash \( n-m \) strings of size \( m \)

\[ \text{Rolling hash.} \]

\[
\begin{align*}
  h_1 &= \sum_{i=0}^{m-1} b_i \cdot k^i \\
  h_2 &= \sum_{i=1}^{m} b_i \cdot k^{i-1}
\end{align*}
\]

\[ \Rightarrow h_2 = \frac{h_1 - b_0 + b_m \cdot k^{m-1}}{k} \]

\( O(1) \) operation

We can calculate hash of next substring in \( O(1) \).
Find all hashes of $n-m$ substrings in $O(n)$. Let the number of collisions to the hash of the pattern $= \alpha$.

For each of the $\alpha$ collisions, check if they are the pattern

$\Rightarrow \quad T(n,m) = n + \alpha m$.

$\mathbb{E}[T(n,m)] = n + \mathbb{E}[\alpha] m$.

If we are doing the hashing mod $p$, $\mathbb{E}[\alpha] = \frac{m-1}{p}$

$\Rightarrow \quad \mathbb{E}[T(n,m)] = n + \frac{m(m-1)}{p}$

so if we take a prime $p \geq m^2$, we are still linear.

[If $p \geq m^2$, $\log p = 2 \log m$]

Bit level complexity is not affected as the factor is multiplicative.
Longest common Substring

given two strings of length \( n \), \( a, \ldots, a_n, b, \ldots, b_n \)
find the length of longest substring.

Can use DP, \( D[i..n][j..n] \)

\[
D[i][j] = \begin{cases} 
\text{length of longest substring ending at} & \text{if } a_i = b_j \\
D[i-1][j-1] + 1 & 0 \\
0 & 0 \text{ otherwise}
\end{cases}
\]

\( \mathcal{O}(n^2) \) time & space

Can do better with a Las Vegas algorithm
Assume the following:

Given an array, bi,..., bL, a length L, we can find the LCS of length L in expected O(n).

If there is a common subsequence of length L, we can binary search over length L.

We can bouncy search over length L in expected L-sizes.

If low = high = u + 1, we only check for O(\log n) L-sizes.

while (high - low > 1)

if l-substring of length = wid

dow = low = wid

else bi = wid.
The expected time complexity

\[ = O(n \log n) \]

as we need expected time = \( O(n) \) to check for wid.

**Proof of assumption.**

Do calculations of Rabin Karp string matching mod \( p \)

where \( \log p = O(\log n) \). [actually \( p \approx 2^{\log n} \)]

1) Build a hash table of all \( n-l+1 \) substrings of \( b \).

\( \text{using rolling hash in } O(n) \).

2) For each substring of \( a \) of size \( l \), check if it is in the Table \( b \) (if there is a collision check the substring.

\# substrings of \( a = O(n) \).

probability of collision for "one" substring

\[ = \frac{l-1}{n} = O(1) \]

\( \therefore \) expected time per substring = \( O(l) \)
\[ \Rightarrow \text{total time} = O(n^2) \]

we haven't found any improvement.

\* we need a smaller probability of collision

if \( P = 2^{\log n} \), doubling the size of \( P \).

\[ \Rightarrow \text{collision probability} \approx \frac{e^n}{n^2} = O\left(\frac{1}{n}\right) \]

\[ \Rightarrow \text{time per substring} = O\left(\frac{e}{n}\right) \]

\[ \Rightarrow \text{total time} = O(n) \]

\[ \text{But: we have to make a table of size} = n^2 = P \]

\[ O(n^2) \]

\[ \therefore \]

\underline{Solution}

In the table \([ P = 2^{\log n} ]\) store a second

dash of each substring, \( g \) where the second

dash is calculated using \( P_a = 2^{\log n} \)
and now before comparing two strings first check if their second hash values are same or not.