WORKSHEET, WEEK 12: ROLLING HASH

Objectives:

1. Sliding window technique with hashing.
2. Rolling hash implementations.
3. Naive string matching
4. Rabin-Karp string matching
5. Multi string matching.

12.1 Rolling Hash Functions

Rolling hash is an ADT: It is a data structure that accumulates some input fragment as a sort of list/queue and supports the following operations:

- RH `:=` $\textsc{RollingHash}([])$ – initialize a rolling hash to an empty list of symbols.
- $RH.\textsc{hash}()$ – return the hash value from the current list.
- $RH.\textsc{append}(val)$ – adds symbol `val` to the end of the list (like `enqueue(val)` for a queue)
- $RH.\textsc{skip}(val)$ – removes the front element from the list (like `dequeue(val)` for a queue). Parameter `val` is often implicit as it is stored at the front of the list stored in the rolling hash.

In the case of strings, the list is a list of characters. It can be a list of anything, but elements on that list are represented as integers in some encoding. For example if we interpret characters as ASCII codes, then character `'A'` is stored as 65 and `'B'` is stored as 66.

We want to treat a list of items as a multidigit number $u \in \mathcal{U}$ in base a (the list in the rolling hash is interpreted as a big number). For example, we can choose $a = 256$, the alphabet size for ASCII code.

12.1.1 Polynomial Rolling Hash

This is one of the most popular implementations for rolling hashes; it uses modular arithmetic. Pick some prime number q such that the number base a is not divisible by q. Define the three ADT functions as follows:

- $hash() = (u \bmod q)$
- $append(val) = ((u \cdot a) + val) \bmod q = ((u \bmod q) \cdot a + val) \bmod q$
- $skip(val) = (u - val \cdot (a^{|u|-1}\ mod\, q)) \bmod q = \big((u \bmod q) - val \cdot (a^{|u|-1} \bmod q)\big) \bmod q$

Example: Pick $a = 100$ and $q = 23$. Let RH be a rolling hash storing `[61, 8, 19, 91, 37]`. We can compute hash value:

$$hash([61, 8, 19, 91, 37]) = (6108199137 \bmod 23) = 12.$$

In general

$$hash([d_3, d_2, d_1, d_1]) = \left(d_3 \cdot a^3 + d_2 \cdot a^2 + d_1 \cdot a^1 + d_0 \cdot a^0\right) \bmod q$$

To make it easier to compute, consider computation with a Hamming code:

$$hash([d_3, d_2, d_1, d_1]) = (((d_3 \cdot a + d_2) \cdot a + d_1) \cdot a + d_0) \bmod q$$

Making this faster:

- Cache the result $(u \bmod p)$ (memorize it in the rolling hash data structure).
- Avoid exponentiation in skip: cache $a^{|u|-1} \bmod p$.
- To append: multiply the cashed $(u \bmod p)$ by base a.
- To skip: divide $(u \bmod p)$ by base (division is expensive, can use multicative inverse).

12.1.2 Cyclic Polynomial (Buzhash)

First introduce an arbitrary hash function `h(c)` mapping single characters to integers from the interval $\left[0, 2^L\right)$. Assume that there are not too many characters and the function can be defined by a lookup table. Define the cyclical shift (bit rotation) to the left. For example, $s(\texttt{1011}) = \texttt{0111}$. The rolling hash function for a list of characters $[c_1, \ldots, c_k]$ is defined by this equality:

$$H = s^{k-1}(h(c_1)) \oplus s^{k-2}(h(c_2)) \oplus \ldots \oplus s(h(c_{k-1})) \oplus h(c_k).$$

It looks like a polynomial, but the powers are replaced by rotating binary shifts. The result of this hash function is also a number in $\left[0, 2^L\right)$.

Implementing rotation:

Assume that we want to skip the first character: c_1 (and simultaneously add a new character c_{k+1}. Now the rolling hash covers the list $[c_2, \ldots, c_k, c_{k+1}]$ In this case do the following assignment to the new hash value H.

$$H := s(H) \oplus s^k(h(c_1)) \oplus h(c_{k+1}),$$

12.2 Naive String Matching Algorithm

String Matching Problem: We have a text T and a pattern P. We need to find all the offsets where the pattern P occurs in the text. Here is a straightforward solution that tries all the possible offsets:

```
NaiveStringMatching($T, P$):
    $n = len(T)$
    $m = len(P)$
    for $s = 0$ to $n - m$:
        if $P[0 : m-1] == T[s : (s+m-1)]$:
            output “Pattern found at offset”, $s$
```

Worst-case behavior: Pick some values for the length of pattern and the text itself. For example, $m = 4$, $n = 14$. Select the alphabet of just two letters: $\mathcal{A} = \{\texttt{a}, \texttt{b}\}$. Consider the following pattern and text:

$$\begin{cases} P = \texttt{aaab} \\ T = \texttt{aaaaaaaaaaaaaa} \end{cases}$$

You will end up comparing all four times for any offset $s = 0, \ldots, 10$. The total number of comparisons is $4 \cdot 11 = 44$. In general the worst-case complexity of the Naive String matching is $O(m \cdot (n - m + 1)) = O(m \cdot n)$.

12.3 Rabin-Karp Algorithm

```
RabinKarpStringMatching($T, P, a, q$)
    $rP$ = RollingHash([])
    $rT$ = RollingHash([])
    for $i$ in range($0, len(P)$):
        $rP$.append($P[i]$)
        $rT$.append($T[i]$)
    for $i$ in range($len(P), len(T)$):
        if $rP$.hash() == $rT$.hash():
            (Here we need to double-check as collisions are possible)
            if $P == T[i - len(P) + 1 : i + 1]$
                output "Pattern found at offset", $i - len(P) + 1$
        $rT$.skip($T[i - len(P)]$)
        $rT$.append($T[i]$)
```

Worst-case behavior: Can we ensure that false matches (hash collisions) do not happen more frequently than with the probability $1/len(P)$?

12.4 Rabin-Karp Multi-String Matching Algorithm

Problem: We have a text T of length n as before. But now we have not just one pattern to search, but a set of k patterns $\mathcal{P} = \{P_0, P_1, \ldots, P_{k-1}\}$; each pattern has the same length m.

```
RabinKarpMultiString($T, \mathcal{P}, m$):
    $hashes$ := $Set$.Empty()
    foreach $P_i \in \mathcal{P}$:
        $rP_i$ = RollingHash([])
        for $j$ in range($0, m$):
            $rP_i$.append($P[j]$)
        $hashes$.insert($rP_i$.hash())
    $rT$ = RollingHash([])
    for $j$ in range($0, m$):
        $rT$.append($T[i]$)
    for $j$ in range($1, n - m + 1$)
        if $rT$.hash() $\in hashes$ and $T[j : j + m - 1] \in \mathcal{P}$
            output "Pattern found at offset", $j$
        $rT.\textsc{skip}(T[j])$
        $rT.\textsc{append}(T[j+m])$
```

This algorithm would take $O(n+km)$ running time. Naive string matching could take $O(nmk)$ running time, if we probe all the k patterns one by one.

12.5 Bloom Filters

Often we have to find patterns in a very large collection of documents. Optimal hashtables that use linear probing to resolve collisions typically have load factors (number of keys stored divided by the size of the hashtable) around $\ln 2 \approx 0.7$. Namely, if the expected load for a hashtable is expected to fluctuate around 1000 items, then the optimal hashtable would have about 1400 slots. Consequently, very large text documents (multiple megabytes) would require many millions of slots in the hashtable. This is not practical, so there is a useful optimization called *Bloom Filter*.

Bloom Filter creates a single set-like data structure that supports two operations:

- Add a new item to a set: `BF.add(item)`.
- Tests, if an item belongs to a set: `BF.contains(item)` (return Boolean true/false).

Note: Unlike regular hashtables Bloom Filters often do not support remove operations. They are useful in contexts where we need to check membership of a comparatively large set, but we have limited space to store that set.

It can be implemented by creating k hash functions – each one randomly and independently from other hash functions maps an item to one of m bits. Assume that we have inserted n elements, the probability that a certain bit is still 0 is:

$$p = \left(1 - \frac{1}{m}\right)^{kn} \approx e^{-kn/m};$$

Now test membership of an element that is not in the set. Each of the k array positions computed by the hash functions is 1 with a probability $(1-p)$. The probability of all of them being 1, which would cause the Bloom filter to erroneously claim that the element is in the set, is the following:

$$\varepsilon = \left(1 - \left[1 - \frac{1}{m}\right]^{kn}\right)^{k} \approx \left(1 - e^{-kn/m}\right)^{k}.$$

Given the size of the bit-array m and the number of items to be inserted n, we can find the optimal number k:

$$k = \frac{m}{n} \ln 2$$

If we know the maximum tolerance of false positives ε, then we can adjust the value m accordingly so that we have limited number of false positives.

12.6 Questions

Question 1 (Computing Rolling Hash – Polynomial Method): Assume that we have an alphabet of 100 symbols. They are denoted by pairs of digits: $\{00, 01, \ldots, 99\}$. Select the sliding window size $m = 5$ and the prime number for modulo $q = 23$.

Let the input be `[3, 14, 15, 92, 65, 35, 89, 79, 31]`.

- Initialize an empty rolling hash `rH` and add the first five numbers using `append()` function defined as follows:

$$append(val) = ((u \cdot a) + val) \bmod q = ((u \bmod q) \cdot a + val) \bmod q.$$

- Show how the rolling hash can "roll" from the list `[3, 14, 15, 92, 65]` to `[14, 15, 92, 65, 35]` (first skip value `3`, then append value `35`.

Question 2 (Computing Rolling Hash – Cyclic Polynomial): Consider 16 Latin letters with randomly assigned 4-bit codes (i.e. $L = 4$):

Letter	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
h(x)	1100	0100	0010	0110	0111	0000	1001	1111	0101	1101	1110	1011	0011	1010	1000	0001

Also assume that the sliding window has size $k = 5$. Find the hash value for $ABIDE$ and then rotate it over to $BIDEN$. Please recall that the rolling hash uses the following formula:

$$H = s^{k-1}(h(c_1)) \oplus s^{k-2}(h(c_2)) \oplus \ldots \oplus s(h(c_{k-1})) \oplus h(c_k),$$

Question 3 (Rolling Hash ADT with Cyclic Polynomial): Write formulas for the Rolling Hash functions (when using Cyclic Polynomial or Buzhash):

- Formula to initialize `RH` to an empty list.
- Formula to append a new character c_i to the existing list (without skipping anything).
- Formula to skip the head of the existing list c_j (without appending anything).

Question 4 (Average Complexity of Naive String Matching) Suppose that pattern P and text T are randomly chosen strings of length m and n, respectively, from an alaphabet with a letters ($a \geq 2$). What is the expected character to character comparisons in the naive algorithm, if any single comparison has a chance $1/a$ to succeed?