# Competitive Training Camp

## Lecture 7

**Christian Yongwhan Lim**

Monday, July 21, 2025

# Overview: Strings

- String Hashing
- Rabin-Karp
- Longest Common Prefix (LCP)

# String Hashing: Main Idea

$$\begin{aligned}\text{hash}(s) &= s[0] + s[1] \cdot p + s[2] \cdot p^2 + \ldots + s[n-1] \cdot p^{n-1} \mod m \\ &= \sum_{i=0}^{n-1} s[i] \cdot p^i \mod m,\end{aligned}$$

# String Hashing: Implementation

```
long long compute_hash(string const& s) {
    const int p = 31;
    const int m = 1e9 + 9;
    long long hash_value = 0;
    long long p_pow = 1;
    for (char c : s) {
        hash_value = (hash_value + (c-'a'+1)*p_pow)%m;
        p_pow = (p_pow * p) % m;
    }
    return hash_value;
}
```

# Example Problem

- Given a list of `n` strings $s_i$, each no longer than `m` characters, find all the duplicate strings and divide them into groups.

# Solution

```
vector<vector<int>>
group_identical_strings(vector<string> const& s) {
    int n = s.size();
    vector<pair<long long, int>> hashes(n);
    for (int i = 0; i < n; i++)
        hashes[i] = {compute_hash(s[i]), i};
    sort(hashes.begin(), hashes.end());
```

# Solution

```
        vector<vector<int>> groups;
        for (int i = 0; i < n; i++) {
            if (i == 0 ||
                hashes[i].first != hashes[i-1].first)
                groups.emplace_back();
            groups.back().push_back(hashes[i].second);
        }
        return groups;
    }
```

# Fast hash calculation of substrings of given string

- Given a string `s` and indices `i` and `j`, find the hash of the substring `s[i...j]`.

# Solution

$$\text{hash}(s[i \dots j]) = \sum_{k=i}^{j} s[k] \cdot p^{k-i} \mod m$$

$$\begin{aligned} \text{hash}(s[i \dots j]) \cdot p^i &= \sum_{k=i}^{j} s[k] \cdot p^k \mod m \\ &= \text{hash}(s[0 \dots j]) - \text{hash}(s[0 \dots i-1]) \mod m \end{aligned}$$

# Applications

- **Rabin-Karp** algorithm for pattern matching in a string in $O(n)$ time.

- Calculating the *number of different substrings* of a string in $O(n^2 \log n)$

# Determine the number of different substrings in a string

- Given a string s of length n, consisting only of lowercase English letters, find the *number of different substrings* in this string.

# Solution

```
int count_unique_substrings(string const& s) {
    int n = s.size();
    const int p = 31;
    const int m = 1e9 + 9;
    vector<long long> p_pow(n);
    p_pow[0] = 1;
    for (int i = 1; i < n; i++)
        p_pow[i] = (p_pow[i-1] * p) % m;
    vector<long long> h(n + 1, 0);
    for (int i = 0; i < n; i++)
        h[i+1] = (h[i] + (s[i]-'a'+1) * p_pow[i]) % m;
```

# Solution (con't)

```
    int cnt = 0;
    for (int l = 1; l <= n; l++) {
        set<long long> hs;
        for (int i = 0; i <= n - l; i++) {
            long long cur_h = (h[i + l]+m-h[i]) % m;
            cur_h = (cur_h * p_pow[n-i-1]) % m;
            hs.insert(cur_h);
        }
        cnt += hs.size();
    }
    return cnt;
}
```

# Rabin-Karp (1987): Problem

- Given two strings - a pattern `s` and a text `t`, determine if the pattern appears in the text and if it does, enumerate all its occurrences in $O(|s| + |t|)$ time.

# Rabin-Karp (1987): Main Idea

- Calculate the hash for the pattern `s`.
- Calculate hash values for all the prefixes of the text `t`.
- Now, we can compare a substring of length `|s|` with `s` in constant time using the calculated hashes.
- So, compare each substring of length `|s|` with the pattern.
- This will take a total of `O(|t|)` time.
- Hence the final complexity of the algorithm is `O(|t|+|s|)`
  - `O(|s|)` is required for calculating the hash of the pattern and;
  - `O(|t|)` for comparing each substring of length `|s|` with the pattern.

# Rabin-Karp: Implementation

```
vector<int> rabin_karp(string const& s,
                       string const& t) {
    const int p = 31;
    const int m = 1e9 + 9;
    int S = s.size(), T = t.size();
    vector<long long> p_pow(max(S, T));
    p_pow[0] = 1;
    for (int i = 1; i < (int)p_pow.size(); i++)
        p_pow[i] = (p_pow[i-1] * p) % m;
```

# Rabin-Karp: Implementation

```
vector<long long> h(T + 1, 0);
for (int i = 0; i < T; i++)
    h[i+1] = (h[i] + (t[i]-'a'+1) * p_pow[i]) % m;
long long h_s = 0;
for (int i = 0; i < S; i++)
    h_s = (h_s + (s[i]-'a'+1) * p_pow[i]) % m;
```

# Rabin-Karp: Implementation

```
    vector<int> occurences;
    for (int i = 0; i + S - 1 < T; i++) {
        long long cur_h = (h[i+S] + m - h[i]) % m;
        if (cur_h == h_s * p_pow[i] % m)
            occurences.push_back(i);
    }
    return occurences;
}
```

# Longest Common Prefix (LCP)

- For a given string s, we want to compute the longest common prefix (**LCP**) of two arbitrary suffixes with position i and j.

- We can use **suffix array** and **Kasai's algorithm** to compute this in **O(n)**.

# Example Problems

- Finding a substring in a string.
- Comparing two substrings of a string.
- Number of different substrings.