String Algorithms

Tawhid Bin Omar

String Processing Fundamentals

Strings are sequences of characters. Efficient string algorithms are crucial for text processing, pattern matching, and DNA analysis.

1 String Basics

Common Operations

```
string s = "hello";
s.length();       // or s.size()
s[i];             // Access character
s.substr(pos, len); // Substring
s.find("ll");     // Find substring
s.rfind("l");     // Find from right
s + s2;           // Concatenation
s.append(s2);     // Append
s.insert(pos, s2); // Insert
s.erase(pos, len); // Erase
s.replace(pos, len, s2); // Replace

// Character operations
tolower(c);       // Lowercase
toupper(c);       // Uppercase
isalpha(c);       // Is letter
isdigit(c);       // Is digit
isalnum(c);       // Is alphanumeric

// String to number
int x = stoi(s); // String to int
long long x = stoll(s); // String to long long
double x = stod(s); // String to double

// Number to string
string s = to_string(123);
```

String Comparison

```
// Lexicographic comparison
s1 < s2;          // Compare
s1 == s2;         // Equal
s1.compare(s2);   // Returns <0, 0, >0

// Sorting
vector<string> words;
sort(words.begin(), words.end());

// Custom comparator
sort(words.begin(), words.end(),
   [](string& a, string& b) {
      return a.length() < b.length();
   });
```

2 Pattern Matching

Naive Pattern Matching

Complexity

Time: $O(nm)$ where n = text length, m = pattern length

```
vector<int> naivePatternMatch(string text,
   string pattern) {
   vector<int> positions;
   int n = text.length();
   int m = pattern.length();

   for (int i = 0; i <= n - m; i++) {
      bool match = true;
      for (int j = 0; j < m; j++) {
         if (text[i + j] != pattern[j]) {
            match = false;
            break;
         }
      }
      if (match) positions.push_back(i);
   }
   return positions;
}
```

KMP (Knuth-Morris-Pratt)

Complexity

Time: $O(n + m)$ | **Space:** $O(m)$

Key Insight

KMP avoids re-checking matched characters by using a prefix function (LPS array).

```
// Compute LPS (Longest Proper Prefix-Suffix)
vector<int> computeLPS(string pattern) {
   int m = pattern.length();
   vector<int> lps(m, 0);
   int len = 0; // length of previous longest prefix

   for (int i = 1; i < m; i++) {
      while (len > 0 &&
         pattern[i] != pattern[len]) {
         len = lps[len - 1];
```

```
        }

        if (pattern[i] == pattern[len]) {
            len++;
        }
        lps[i] = len;
    }
    return lps;
}

// KMP pattern matching
vector<int> KMP(string text, string pattern) {
    int n = text.length();
    int m = pattern.length();
    vector<int> lps = computeLPS(pattern);
    vector<int> positions;

    int i = 0; // index for text
    int j = 0; // index for pattern

    while (i < n) {
        if (text[i] == pattern[j]) {
            i++;
            j++;
        }

        if (j == m) {
            positions.push_back(i - j);
            j = lps[j - 1];
        } else if (i < n && text[i] != pattern[j]) {
            if (j != 0) {
                j = lps[j - 1];
            } else {
                i++;
            }
        }
    }
    return positions;
}
```

Rabin-Karp (Rolling Hash)

Complexity

Time: $O(n+m)$ average | **Space:** $O(1)$

```
const long long MOD = 1e9 + 7;
const long long BASE = 31;

// Compute hash of string
long long computeHash(string s) {
    long long hash = 0;
    long long pow = 1;

    for (char c : s) {
        hash = (hash + (c - 'a' + 1) * pow) % MOD;
        pow = (pow * BASE) % MOD;
    }
    return hash;
}

// Rabin-Karp pattern matching
vector<int> rabinKarp(string text,
    string pattern) {
    int n = text.length();
    int m = pattern.length();

    if (m > n) return {};

    long long patternHash = computeHash(pattern);
    long long textHash = computeHash(
        text.substr(0, m));

    long long pow = 1;
    for (int i = 0; i < m - 1; i++)
        pow = (pow * BASE) % MOD;

    vector<int> positions;

    for (int i = 0; i <= n - m; i++) {
        if (textHash == patternHash) {
            // Verify match (avoid hash collision)
            if (text.substr(i, m) == pattern)
                positions.push_back(i);
        }

        if (i < n - m) {
            // Rolling hash: remove first, add next
            textHash = (textHash -
                (text[i] - 'a' + 1) + MOD) % MOD;
            textHash = (textHash *
                modInv(BASE, MOD)) % MOD;
            textHash = (textHash +
                (text[i + m] - 'a' + 1) * pow) % MOD;
        }
    }
    return positions;
}

// Polynomial rolling hash (simpler)
vector<int> rabinKarpSimple(string text,
    string pattern) {
    int n = text.length();
    int m = pattern.length();
    long long patHash = 0, txtHash = 0;
    long long h = 1;

    // h = BASE^(m-1)
    for (int i = 0; i < m - 1; i++)
        h = (h * BASE) % MOD;

    // Initial hash
    for (int i = 0; i < m; i++) {
        patHash = (patHash * BASE + pattern[i]) % MOD;
        txtHash = (txtHash * BASE + text[i]) % MOD;
    }

    vector<int> positions;

    for (int i = 0; i <= n - m; i++) {
        if (patHash == txtHash) {
            if (text.substr(i, m) == pattern)
                positions.push_back(i);
        }

        if (i < n - m) {
            txtHash = (txtHash - text[i] * h % MOD
                + MOD) % MOD;
            txtHash = (txtHash * BASE +
                text[i + m]) % MOD;
        }
    }
    return positions;
}
```

Z-Algorithm

Complexity

Time: $O(n)$ | Computes longest prefix match at each position

```
// Z[i] = length of longest substring starting
// from i which is also prefix of s
vector<int> zAlgorithm(string s) {
    int n = s.length();
    vector<int> z(n);
    z[0] = n;

    int l = 0, r = 0;
    for (int i = 1; i < n; i++) {
        if (i > r) {
```

```
            l = r = i;
            while (r < n && s[r - l] == s[r])
               r++;
            z[i] = r - l;
            r--;
         } else {
            int k = i - l;
            if (z[k] < r - i + 1) {
               z[i] = z[k];
            } else {
               l = i;
               while (r < n && s[r - l] == s[r])
                  r++;
               z[i] = r - l;
               r--;
            }
         }
      }
      return z;
}

// Pattern matching using Z-algorithm
vector<int> zPatternMatch(string text,
      string pattern) {
      string s = pattern + "$" + text;
      vector<int> z = zAlgorithm(s);

      vector<int> positions;
      int m = pattern.length();

      for (int i = m + 1; i < z.size(); i++) {
         if (z[i] == m) {
            positions.push_back(i - m - 1);
         }
      }
      return positions;
}
```

3 String Hashing

Polynomial Hash

```
class StringHash {
   vector<long long> hash, pow;
   const long long MOD = 1e9 + 7;
   const long long BASE = 31;

public:
   StringHash(string s) {
      int n = s.length();
      hash.resize(n + 1, 0);
      pow.resize(n + 1);
      pow[0] = 1;

      for (int i = 0; i < n; i++) {
         hash[i + 1] = (hash[i] +
            (s[i] - 'a' + 1) * pow[i]) % MOD;
         pow[i + 1] = (pow[i] * BASE) % MOD;
      }
   }

   // Hash of substring [l, r]
   long long getHash(int l, int r) {
      long long result = (hash[r + 1] - hash[l]
         + MOD) % MOD;
      result = (result * modInv(pow[l], MOD)) % MOD;
      return result;
   }

   // Check if s[l1..r1] == s[l2..r2]
   bool equal(int l1, int r1, int l2, int r2) {
      return getHash(l1, r1) == getHash(l2, r2);
   }
};

// Double hashing (reduces collision)
class DoubleHash {
   StringHash h1, h2;

public:
   DoubleHash(string s) : h1(s), h2(s) {
      // h2 uses different MOD and BASE
   }

   pair<long long, long long> getHash(int l,
      int r) {
      return {h1.getHash(l, r), h2.getHash(l, r)};
   }
};
```

4 Trie (Prefix Tree)

```
struct TrieNode {
   map<char, TrieNode*> children;
   bool isEnd;
   int count; // number of strings ending here

   TrieNode() : isEnd(false), count(0) {}
};

class Trie {
   TrieNode* root;

public:
   Trie() { root = new TrieNode(); }

   void insert(string word) {
      TrieNode* node = root;
      for (char c : word) {
         if (!node->children[c])
            node->children[c] = new TrieNode();
         node = node->children[c];
      }
```

```
        node->isEnd = true;
        node->count++;
    }

    bool search(string word) {
        TrieNode* node = root;
        for (char c : word) {
            if (!node->children[c])
                return false;
            node = node->children[c];
        }
        return node->isEnd;
    }

    bool startsWith(string prefix) {
        TrieNode* node = root;
        for (char c : prefix) {
            if (!node->children[c])
                return false;
            node = node->children[c];
        }
        return true;
    }

    // Auto-complete
    void dfsWords(TrieNode* node, string current,
        vector<string>& words) {
        if (node->isEnd) {
            for (int i = 0; i < node->count; i++)
                words.push_back(current);
        }

        for (auto& [c, child] : node->children) {
            dfsWords(child, current + c, words);
        }
    }

    vector<string> autocomplete(string prefix) {
        TrieNode* node = root;
        for (char c : prefix) {
            if (!node->children[c])
                return {};
            node = node->children[c];
        }

        vector<string> words;
        dfsWords(node, prefix, words);
        return words;
    }
};
```

5 Suffix Array

Complexity

Time: $O(n \log n)$ to build | Enables binary search on suffixes

```
// Build suffix array in O(n log n)
vector<int> buildSuffixArray(string s) {
    s += "$"; // sentinel
    int n = s.length();
    vector<int> p(n), c(n);

    // k = 0
    vector<pair<char, int>> a(n);
    for (int i = 0; i < n; i++)
        a[i] = {s[i], i};
    sort(a.begin(), a.end());

    for (int i = 0; i < n; i++)
        p[i] = a[i].second;

    c[p[0]] = 0;
    for (int i = 1; i < n; i++) {
        if (a[i].first == a[i-1].first)
            c[p[i]] = c[p[i-1]];
        else
            c[p[i]] = c[p[i-1]] + 1;
    }

    // k > 0
    int k = 0;
    while ((1 << k) < n) {
        vector<pair<pair<int,int>, int>> a(n);
        for (int i = 0; i < n; i++) {
            a[i] = {{c[i], c[(i + (1 << k)) % n]}, i};
        }
        sort(a.begin(), a.end());

        for (int i = 0; i < n; i++)
            p[i] = a[i].second;

        c[p[0]] = 0;
        for (int i = 1; i < n; i++) {
            if (a[i].first == a[i-1].first)
                c[p[i]] = c[p[i-1]];
            else
                c[p[i]] = c[p[i-1]] + 1;
        }
        k++;
    }

    return vector<int>(p.begin() + 1, p.end());
}

// LCP array (Longest Common Prefix)
vector<int> buildLCP(string s, vector<int>& sa) {
    int n = s.length();
    vector<int> rank(n), lcp(n - 1);

    for (int i = 0; i < n; i++)
        rank[sa[i]] = i;

    int k = 0;
    for (int i = 0; i < n; i++) {
        if (rank[i] == n - 1) {
            k = 0;
            continue;
        }

        int j = sa[rank[i] + 1];
        while (i + k < n && j + k < n &&
            s[i + k] == s[j + k])
            k++;

        lcp[rank[i]] = k;
        if (k > 0) k--;
    }

    return lcp;
}
```

6 Manacher's Algorithm

Find longest palindromic substring in $O(n)$.

```
// Manacher's algorithm
string longestPalindrome(string s) {
    // Transform to handle even-length palindromes
    string t = "#";
    for (char c : s) {
        t += c;
        t += "#";
    }

    int n = t.length();
    vector<int> p(n, 0);
    int center = 0, right = 0;

    for (int i = 0; i < n; i++) {
        int mirror = 2 * center - i;
```

```
        if (i < right)
            p[i] = min(right - i, p[mirror]);

        // Expand around i
        while (i + p[i] + 1 < n && i - p[i] - 1 >= 0
            &&
            t[i + p[i] + 1] == t[i - p[i] - 1])
            p[i]++;

        // Update center and right boundary
        if (i + p[i] > right) {
            center = i;
            right = i + p[i];
        }
    }

    // Find longest palindrome
    int maxLen = 0, centerIndex = 0;
    for (int i = 0; i < n; i++) {
        if (p[i] > maxLen) {
            maxLen = p[i];
            centerIndex = i;
        }
    }

    int start = (centerIndex - maxLen) / 2;
    return s.substr(start, maxLen);
}
```

7 String Problems

Anagram Detection

```
bool isAnagram(string s1, string s2) {
    if (s1.length() != s2.length())
        return false;

    vector<int> count(26, 0);
    for (int i = 0; i < s1.length(); i++) {
        count[s1[i] - 'a']++;
        count[s2[i] - 'a']--;
    }

    for (int c : count)
        if (c != 0) return false;

    return true;
}

// Group anagrams
vector<vector<string>> groupAnagrams(
    vector<string>& strs) {
    unordered_map<string, vector<string>> groups;

    for (string& s : strs) {
        string key = s;
        sort(key.begin(), key.end());
        groups[key].push_back(s);
    }

    vector<vector<string>> result;
    for (auto& [key, group] : groups) {
        result.push_back(group);
    }
    return result;
}
```

Palindrome Check

```
bool isPalindrome(string s) {
    int l = 0, r = s.length() - 1;
    while (l < r) {
        if (s[l] != s[r])
            return false;
        l++;
        r--;
    }
    return true;
}

// Longest palindromic substring (DP)
string longestPalindromeDP(string s) {
    int n = s.length();
    vector<vector<bool>> dp(n, vector<bool>(n,
        false));
    int start = 0, maxLen = 1;

    // Single characters
    for (int i = 0; i < n; i++)
        dp[i][i] = true;

    // Two characters
    for (int i = 0; i < n - 1; i++) {
        if (s[i] == s[i + 1]) {
            dp[i][i + 1] = true;
            start = i;
            maxLen = 2;
        }
    }

    // Longer substrings
    for (int len = 3; len <= n; len++) {
        for (int i = 0; i <= n - len; i++) {
            int j = i + len - 1;
            if (s[i] == s[j] && dp[i + 1][j - 1]) {
                dp[i][j] = true;
                start = i;
                maxLen = len;
            }
        }
    }

    return s.substr(start, maxLen);
}
```

Longest Common Substring

```
// Using DP
string longestCommonSubstring(string s1,
    string s2) {
    int n = s1.length(), m = s2.length();
    vector<vector<int>> dp(n + 1,
        vector<int>(m + 1, 0));

    int maxLen = 0, endPos = 0;

    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            if (s1[i - 1] == s2[j - 1]) {
                dp[i][j] = dp[i - 1][j - 1] + 1;
                if (dp[i][j] > maxLen) {
                    maxLen = dp[i][j];
                    endPos = i - 1;
                }
            }
        }
    }

    return s1.substr(endPos - maxLen + 1, maxLen);
}
```

String Rotation

```
// Check if s2 is rotation of s1
bool isRotation(string s1, string s2) {
    if (s1.length() != s2.length())
        return false;
```

```
    string s1s1 = s1 + s1;
    return s1s1.find(s2) != string::npos;
}
```

Run-Length Encoding

```
string encode(string s) {
    string result = "";
    int count = 1;

    for (int i = 1; i <= s.length(); i++) {
        if (i < s.length() && s[i] == s[i - 1]) {
            count++;
        } else {
            result += s[i - 1];
            result += to_string(count);
            count = 1;
        }
    }
    return result;
}

string decode(string s) {
    string result = "";
    for (int i = 0; i < s.length(); i += 2) {
        char c = s[i];
        int count = s[i + 1] - '0';
        result += string(count, c);
    }
    return result;
}
```

Minimum Window Substring

Problem

Find minimum window in s containing all characters of t.

Solution

```
string minWindow(string s, string t) {
    if (s.length() < t.length()) return "";

    unordered_map<char, int> need, window;
    for (char c : t) need[c]++;

    int left = 0, right = 0;
    int valid = 0;
    int start = 0, minLen = INT_MAX;

    while (right < s.length()) {
        char c = s[right];
        right++;

        if (need.count(c)) {
            window[c]++;
            if (window[c] == need[c])
                valid++;
        }

        while (valid == need.size()) {
            if (right - left < minLen) {
                start = left;
                minLen = right - left;
            }

            char d = s[left];
            left++;

            if (need.count(d)) {
                if (window[d] == need[d])
                    valid--;
                window[d]--;
            }
        }
    }

    return minLen == INT_MAX ? "" :
        s.substr(start, minLen);
}
```

Regular Expression Matching

```
// '.' matches any character
// '*' matches zero or more of preceding element
bool isMatch(string s, string p) {
    int n = s.length(), m = p.length();
    vector<vector<bool>> dp(n + 1,
        vector<bool>(m + 1, false));

    dp[0][0] = true;

    // Handle patterns like a*, a*b*, etc.
    for (int j = 2; j <= m; j++) {
        if (p[j - 1] == '*')
            dp[0][j] = dp[0][j - 2];
    }

    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            if (p[j - 1] == '*') {
                dp[i][j] = dp[i][j - 2]; // zero
                    occurrence

                if (p[j - 2] == '.' ||
                    p[j - 2] == s[i - 1]) {
                    dp[i][j] = dp[i][j] || dp[i - 1][j];
                }
            } else if (p[j - 1] == '.' ||
                p[j - 1] == s[i - 1]) {
                dp[i][j] = dp[i - 1][j - 1];
            }
        }
```

```
    }

    return dp[n][m];
}
```

Practice Problems

Pattern Matching:

- CSES - String Matching
- LeetCode 28 - Implement strStr()
- SPOJ - NHAY (KMP)

Palindromes:

- CSES - Palindrome Queries
- LeetCode 5 - Longest Palindromic Substring
- LeetCode 647 - Palindromic Substrings

Hashing:

- CSES - String Hashing
- Codeforces 7D - Palindrome Degree
- LeetCode 1044 - Longest Duplicate Substring

Trie:

- LeetCode 208 - Implement Trie
- LeetCode 212 - Word Search II
- CSES - Word Combinations

Advanced:

- CSES - Finding Periods
- LeetCode 76 - Minimum Window Substring
- SPOJ - LPS (Longest Palindrome)

Master string algorithms for text processing, DNA analysis, and pattern recognition.