### Week 20: String Algorithms Advanced Topics

**Topics:** - Suffix Arrays and LCP Arrays - Suffix Automaton - Z-Algorithm and KMP Advanced Applications - Aho-Corasick Algorithm for Multiple Pattern Matching - String Hashing and Rabin-Karp

**Weekly Tips:** - Suffix arrays allow fast substring queries and comparisons. - LCP (Longest Common Prefix) arrays help in problems like longest repeated substring. - Z-Algorithm computes matching prefixes efficiently for pattern searching. - Aho-Corasick efficiently handles multiple pattern matches simultaneously. - Rolling hash (Rabin-Karp) allows constant-time substring comparisons.

**Problem 1: Longest Repeated Substring** **Link:** [SPOJ LRS](https://www.spoj.com/problems/LCS/) **Difficulty:** Advanced

**C++ Solution with Explanation Comments:**

#include <bits/stdc++.h>  
using namespace std;  
int main(){  
 string s; cin>>s;  
 int n=s.size();  
 vector<int> sa(n), rank(n), tmp(n);  
 for(int i=0;i<n;i++){ sa[i]=i; rank[i]=s[i]; }  
 for(int k=1;k<n;k\*=2){  
 auto cmp=[&](int i,int j){  
 if(rank[i]!=rank[j]) return rank[i]<rank[j];  
 int ri=i+k<n?rank[i+k]:-1;  
 int rj=j+k<n?rank[j+k]:-1;  
 return ri<rj;  
 };  
 sort(sa.begin(),sa.end(),cmp);  
 tmp[sa[0]]=0;  
 for(int i=1;i<n;i++) tmp[sa[i]]=tmp[sa[i-1]]+(cmp(sa[i-1],sa[i])?1:0);  
 rank=tmp;  
 }  
 vector<int> lcp(n);  
 for(int i=0,k=0;i<n;i++){  
 if(rank[i]==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) k--;  
 }  
 cout<<\*max\_element(lcp.begin(),lcp.end())<<endl;  
}

**Explanation Comments:** - Build suffix array to sort all suffixes lexicographically. - Compute LCP array to find longest common prefix between consecutive suffixes. - Maximum value in LCP array is length of longest repeated substring.

**Problem 2: Aho-Corasick Multiple Pattern Matching** **Link:** [HackerRank Aho-Corasick](https://www.hackerrank.com/topics/aho-corasick) **Difficulty:** Advanced

**C++ Solution with Explanation Comments:**

#include <bits/stdc++.h>  
using namespace std;  
struct Node{  
 Node\* nxt[26]={nullptr};  
 Node\* fail=nullptr;  
 vector<int> output;  
};  
Node\* buildTrie(vector<string>& patterns){  
 Node\* root=new Node();  
 for(int i=0;i<patterns.size();i++){  
 Node\* node=root;  
 for(char c:patterns[i]){  
 if(!node->nxt[c-'a']) node->nxt[c-'a']=new Node();  
 node=node->nxt[c-'a'];  
 }  
 node->output.push\_back(i);  
 }  
 queue<Node\*> q;  
 root->fail=root;  
 for(int i=0;i<26;i++) if(root->nxt[i]){ root->nxt[i]->fail=root; q.push(root->nxt[i]); }  
 while(!q.empty()){  
 Node\* cur=q.front(); q.pop();  
 for(int i=0;i<26;i++){  
 Node\* child=cur->nxt[i]; if(!child) continue;  
 Node\* f=cur->fail;  
 while(f!=root && !f->nxt[i]) f=f->fail;  
 if(f->nxt[i]) f=f->nxt[i];  
 child->fail=f;  
 child->output.insert(child->output.end(), f->output.begin(), f->output.end());  
 q.push(child);  
 }  
 }  
 return root;  
}  
void search(Node\* root,string& text){  
 Node\* node=root;  
 for(int i=0;i<text.size();i++){  
 while(node!=root && !node->nxt[text[i]-'a']) node=node->fail;  
 if(node->nxt[text[i]-'a']) node=node->nxt[text[i]-'a'];  
 for(int idx: node->output) cout<<"Pattern "<<idx<<" found at position "<<i<<endl;  
 }  
}  
int main(){  
 vector<string> patterns={"he","she","his","hers"};  
 string text="ahishers";  
 Node\* root=buildTrie(patterns);  
 search(root,text);  
}

**Explanation Comments:** - Build trie of all patterns. - Use failure links to jump when mismatch occurs. - Output vector stores indices of patterns found. - Efficient for multiple pattern matching in large texts.

**End of Week 20** - Practice advanced string algorithms for substring search, pattern matching, and string queries. - Suffix arrays, LCP, and Aho-Corasick are essential for ACM-ICPC string-intensive problems. - Understand trade-offs between different string matching approaches.