Suffix Array & Suffix Tree in DSA: Beginner to Advanced Guide

Subtitle: Learn how to efficiently handle substring queries, pattern matching, and text algorithms using suffix structures.

Website Name: haas.dev
Website Link: https://dev-roast-app.vercel.app

Introduction

Suffix Arrays and Suffix Trees are powerful **string data structures** for solving **substring, pattern matching, and text analysis problems** efficiently. They help in **fast search, longest common substring, and string frequency queries**.

Step 1: Core Concepts

- **Suffix Array:** Sorted array of all suffixes of a string
- **Suffix Tree:** Compressed trie of all suffixes of a string
- Used for **pattern matching, substring search, and LCP (Longest Common Prefix)**
- **Time Complexity:**
  - Suffix Array: O(n log n)
  - Suffix Tree: O(n) for construction (Ukkonen's algorithm)

Step 2: Suffix Array Construction (Naive)

```
function buildSuffixArray(s){

 let suffixes = [];

 for(let i=0;i<s.length;i++){

  suffixes.push({index:i, suffix:s.slice(i)});

 }

 suffixes.sort((a,b)=>a.suffix.localeCompare(b.suffix));

 return suffixes.map(x=>x.index);

}


// Example usage
```

```
let str = "banana";

let sa = buildSuffixArray(str);

console.log("Suffix Array:", sa); // [5, 3, 1, 0, 4, 2]
```

- The array **[5,3,1,0,4,2]** corresponds to sorted suffixes:
  "a","ana","anana","banana","na","nana"

Step 3: LCP Array (Longest Common Prefix)

- LCP[i] = length of longest common prefix between **suffixes sa[i] and sa[i-1]**

```
function buildLCP(s, sa){
  let n = s.length, rank = Array(n).fill(0), lcp = Array(n).fill(0);
  for(let i=0;i<n;i++) rank[sa[i]]=i;
  let k=0;
  for(let i=0;i<n;i++){
    if(rank[i]===0){ k=0; continue; }
    let 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;
}


let lcp = buildLCP(str, sa);

console.log("LCP Array:", lcp); // [0,1,3,0,0,2]
```

Step 4: Suffix Tree Overview

- Compressed trie storing all suffixes
- Each **edge stores substring**
- Efficient for:
  - Pattern matching in O(m)
  - Counting distinct substrings

- Longest repeated substring

Note: Suffix Tree construction is advanced (Ukkonen's algorithm) and usually implemented in C++/Java for efficiency

Step 5: Applications

1. **Substring search and pattern matching**
2. **Counting distinct substrings**
3. **Longest repeated substring / palindrome detection**
4. **String compression and text analysis**

Step 6: Mini Exercises

1. Build **suffix array and LCP** for a sample string
2. Count **number of distinct substrings** using suffix array
3. Find **longest repeated substring**
4. Solve **pattern matching** problems using binary search on suffix array

Step 7: Common Mistakes

- Forgetting **0-based vs 1-based indexing** in arrays
- Miscalculating LCP between wrong suffix pairs
- Using naive substring comparison → slow for large strings
- Confusing suffix array vs suffix tree applications

Step 8: Practice Plan for Suffix Structures

Day 1:

- Build suffix array and LCP → 3–5 problems

Day 2:

- Pattern matching and substring queries → 3–5 problems

Day 3:

- Advanced string problems using suffix tree/array → 5–7 problems

Key Takeaways

- Suffix Arrays/Trees allow **efficient substring and pattern analysis**
- Essential for **text algorithms, string matching, and competitive programming**
- Practice **construction, LCP, and query operations**
- Foundation for **advanced string algorithm mastery**

Visit haas.dev for more Suffix Array & Suffix Tree guides, problem sets, and interview preparation resources.

Website Name: haas.dev
Website Link: https://dev-roast-app.vercel.app