Network System Design
CS6100

Tutorial 01

R. Abinav
Roll No: ME23B1004

January 14, 2026

Question 1: What is the need for Longest Prefix Matching?

Introduction and Working Mechanism

Longest Prefix Matching (LPM) is a fundamental algorithm in IP routing that resolves ambiguity when a destination IP address matches multiple routing table entries. With CIDR (Classless Inter-Domain Routing), routers use variable-length network prefixes to represent entire subnets rather than storing billions of individual IP addresses. A single destination IP can simultaneously match multiple prefixes—for instance, 192.168.20.19 matches both 192.168.0.0/16 and 192.168.20.0/22. LPM operates through bit-by-bit comparison between the destination address and routing table prefixes, starting from the most significant bit and continuing for the length specified by each subnet mask. Among all matching entries, the router selects the one with the longest prefix length, representing the most specific network path.

Necessity and Consequences Without LPM

LPM is essential because it enables hierarchical address aggregation—ISPs can advertise a single /16 prefix representing thousands of smaller /24 networks, dramatically reducing global routing table size while maintaining fine-grained control over packet forwarding. Without LPM, routers would face an impossible trade-off: either maintain exact-match entries for every possible IP address (consuming terabytes of memory and making routing infeasible), or lose the ability to implement overlapping routes and specific subnet control. There would be no mechanism to resolve conflicts when multiple routes exist, leading to ambiguous and unpredictable forwarding behavior. Route aggregation would become impossible, forcing the global routing table to explode in size, and advanced traffic engineering—where operators steer traffic by advertising more specific prefixes to override broader routes—would be completely unavailable, eliminating critical network optimization capabilities.

Question 2: IP Lookup Implementation using Binary Search Tree

Implementation Overview

I have implemented the IP lookup algorithms in Rust as I am comfortable writing Rust code and prefer its memory safety guarantees and performance characteristics for systems programming tasks.

Code: ip_bst.rs

```rust
#[derive(Debug)]
pub struct BSTNode{
    prefix: u32,
    prefix_len: u8,
    next_hop: String,
    left: Option<Box<BSTNode>>,
    right: Option<Box<BSTNode>>,
}

impl BSTNode{
    pub fn new(prefix: u32, prefix_len: u8, next_hop: String) -> Self{
        BSTNode{
            prefix,
            prefix_len,
            next_hop,
            left: None,
            right: None,
        }
    }

    pub fn insert(&mut self, prefix: u32, prefix_len: u8, next_hop: String){
        if prefix < self.prefix{
            match &mut self.left {
                Some(node) => node.insert(prefix, prefix_len, next_hop),
                None => self.left = Some(Box::new(BSTNode::new(prefix, prefix_len, next_hop))),
            }
        }else{
            match &mut self.right{
                Some(node) => node.insert(prefix, prefix_len, next_hop),
                None => self.right = Some(Box::new(BSTNode::new(prefix, prefix_len, next_hop))),
            }
        }
    }
```

```
34
35      pub fn matches(&self, ip: u32) -> bool{
36          let mask = if self.prefix_len == 0{
37              0
38          }else{
39              !0u32 << (32 - self.prefix_len)
40          };
41
42          (ip & mask) == (self.prefix & mask)
43      }
44
45      pub fn lookup(&self, ip: u32, best: &mut Option<String>,
            best_len: &mut i32){
46          //check the curr node
47          if self.matches(ip) && (self.prefix_len as i32) > *
                best_len{
48              *best_len = self.prefix_len as i32;
49              *best = Some(self.next_hop.clone());
50          }
51
52          //search both the subtrees
53          if let Some(ref left) = self.left{
54              left.lookup(ip, best, best_len);
55          }
56          if let Some(ref right) = self.right{
57              right.lookup(ip, best, best_len);
58          }
59      }
60  }
```

Listing 1: Binary Search Tree Implementation for IP Lookup

Question 3: IP Lookup Implementation using Binary Trie

Code: ip_bin_trie.rs

```rust
#[derive(Debug)]
pub struct TrieNode{
    left: Option<Box<TrieNode>>,
    right: Option<Box<TrieNode>>,
    next_hop: Option<String>
}

impl TrieNode{
    pub fn new() -> Self {
        TrieNode {
            left: None,
            right: None,
            next_hop: None,
        }
    }

    pub fn insert(&mut self, prefix: u32, prefix_len: u8,
       next_hop: String){
        let mut curr = self;

        for i in (32 - prefix_len..32).rev(){
            let bit = (prefix >> i) & 1;

            if bit == 0{
                curr = curr.left.get_or_insert_with(|| Box::new(
                   TrieNode::new()));
            }else{
                curr = curr.right.get_or_insert_with(|| Box::new(
                   TrieNode::new()));
            }
        }

        curr.next_hop = Some(next_hop);
    }

    pub fn lookup(&self, ip: u32) -> Option<String>{
        let mut curr = self;
        let mut result = None;

        for i in (0..32).rev(){
            //update result if curr node is valid prefix
            if let Some(ref hop) = curr.next_hop {
                result = Some(hop.clone());
            }

```

```
            let bit = (ip >> i) & 1;

            curr = if bit == 0{
                match &curr.left{
                    Some(node) => node.as_ref(),
                    None => break,
                }
            }else{
                match &curr.right{
                    Some(node) => node.as_ref(),
                    None => break,
                }
            };
        }

        if let Some(ref hop) = curr.next_hop{
            result = Some(hop.clone());
        }

        return result;
    }
}
```

Listing 2: Binary Trie Implementation for IP Lookup

Testing and Performance Comparison

Utility Functions (utils.rs)

```rust
pub fn ip_to_u32(ip: &str) -> u32{
    let parts: Vec<u32> = ip.split('.').map(|s| s.parse().unwrap
       ()).collect();
    (parts[0] << 24) | (parts[1] << 16) | (parts[2] << 8) | parts
       [3]
}

pub fn u32_to_ip(ip: u32) -> String {
    format!(
        "{}.{}.{}.{}",
        (ip >> 24) & 0xFF,
        (ip >> 16) & 0xFF,
        (ip >> 8) & 0xFF,
        ip & 0xFF
    )
}
```

Listing 3: IP Conversion Utilities

Main Testing Program (main.rs)

```rust
mod utils;
mod ip_bst;
mod ip_bin_trie;

use std::time::Instant;
use utils::{ip_to_u32, u32_to_ip};
use ip_bst::BSTNode;
use ip_bin_trie::TrieNode;

fn main() {
    println!("ip lookup\n");

    //base table
    let mut routes = vec![
        ("192.168.0.0", 16, "Router_A"),
        ("192.168.1.0", 24, "Router_B"),
        ("192.168.1.128", 25, "Router_C"),
        ("10.0.0.0", 8, "Router_D"),
        ("172.16.0.0", 12, "Router_E"),
    ];

    let mut generated_routes: Vec<(String, u8, String)> = vec![];
    for i in 0..100 {
        generated_routes.push((
            format!("10.{}.0.0", i),
            16,
            format!("Router_{}", i + 100),
        ));
        generated_routes.push((
```

```
            format!("172.{}.0.0", (i % 240) + 16),
            20,
            format!("Router_{}", i + 200),
        ));
        generated_routes.push((
            format!("192.168.{}.0", i % 256),
            24,
            format!("Router_{}", i + 300),
        ));
    }

    println!("Total routes: {} (base) + {} (generated) = {}",
        routes.len(), generated_routes.len(),
        routes.len() + generated_routes.len());

    let mut bst_root = BSTNode::new(
        ip_to_u32(routes[0].0),
        routes[0].1,
        routes[0].2.to_string(),
    );

    let mut trie_root = TrieNode::new();

    for (i, (prefix, len, hop)) in routes.iter().enumerate() {
        let prefix_ip = ip_to_u32(prefix);
        if i < 5 {
            println!("{}/{} -> {}", prefix, len, hop);
        }

        if i == 0 {
            //first route init bst root
        } else {
            bst_root.insert(prefix_ip, *len, hop.to_string());
        }
        trie_root.insert(prefix_ip, *len, hop.to_string());
    }

    for (prefix, len, hop) in &generated_routes {
        let prefix_ip = ip_to_u32(prefix);
        bst_root.insert(prefix_ip, *len, hop.to_string());
        trie_root.insert(prefix_ip, *len, hop.to_string());
    }

    //test lookups
    println!("\nLookup Tests:");

    let test_ips = vec![
        "192.168.1.5",
        "192.168.1.200",
        "10.5.10.1",
        "172.16.5.5",
        "8.8.8.8",
    ];

    for ip_str in &test_ips {
        let ip = ip_to_u32(ip_str);

        //bst
```

```
        let mut bst_result = None;
        let mut best_len = -1;
        bst_root.lookup(ip, &mut bst_result, &mut best_len);

        //trie
        let trie_result = trie_root.lookup(ip);

        println!("\nLooking up: {}", ip_str);
        println!("BST  Result: {}",
            bst_result.unwrap_or("No route".to_string()));
        println!("Trie Result: {}",
            trie_result.unwrap_or("No route".to_string()));
    }

    println!("\nPerformance test - bst vs binary trie");
    println!("Performing 100,000 lookups on {} routes...\n",
        routes.len() + generated_routes.len());

    let lookup_ip = ip_to_u32("192.168.1.5");

    //bst
    let start = Instant::now();
    for _ in 0..100_000 {
        let mut result = None;
        let mut best_len = -1;
        bst_root.lookup(lookup_ip, &mut result, &mut best_len);
    }
    let bst_time = start.elapsed();

    //trie
    let start = Instant::now();
    for _ in 0..100_000 {
        let _ = trie_root.lookup(lookup_ip);
    }
    let trie_time = start.elapsed();

    println!("BST  Time: {:.3} ms", bst_time.as_secs_f64() * 1000.0);
    println!("Trie Time: {:.3} ms", trie_time.as_secs_f64() * 1000.0);

    let speedup = bst_time.as_secs_f64() / trie_time.as_secs_f64();
    println!(
        "\nSpeedup: {:.2}x {}",
        speedup,
        if trie_time < bst_time {
            "(Trie is faster)"
        } else {
            "(BST is faster)"
        }
    );
}
```

Listing 4: Performance Comparison Implementation

Program Output

```
warning: `tutorial-01` (bin "tutorial-01") generated 3 warnings (run `cargo fix --bin "tutorial-01"` to apply 2 suggestions)
    Finished `release` profile [optimized] target(s) in 1.09s
     Running `target/release/tutorial-01`
ip lookup

Total routes: 5 (base) + 300 (generated) = 305
192.168.0.0/16 -> Router_A
192.168.1.0/24 -> Router_B
192.168.1.128/25 -> Router_C
10.0.0.0/8 -> Router_D
172.16.0.0/12 -> Router_E

Lookup Tests:

Looking up: 192.168.1.5
BST  Result: Router_B
Trie Result: Router_301

Looking up: 192.168.1.200
BST  Result: Router_C
Trie Result: Router_C

Looking up: 10.5.10.1
BST  Result: Router_105
Trie Result: Router_105

Looking up: 172.16.5.5
BST  Result: Router_200
Trie Result: Router_200

Looking up: 8.8.8.8
BST  Result: No route
Trie Result: No route

Performance test - bst vs binary trie
Performing 100,000 lookups on 305 routes...

BST  Time: 220.410 ms
Trie Time: 8.805 ms

Speedup: 25.03x (Trie is faster)
```