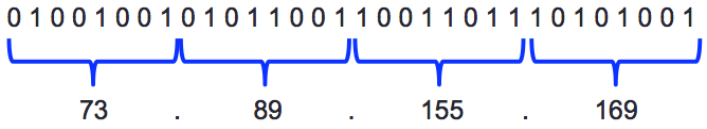
In this mini-project you will be simulating the logic of an Internet data router based on data structures you have encountered this semester.

What is an Internet Protocol Address?

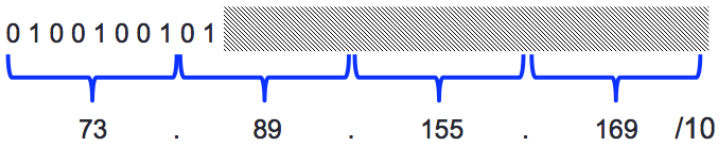
An Internet Protocol (IP) address (IP version 4, to be specific) is a 32-bit sequence of bits (0's and 1's) that uniquely identifies a destination on the *public* Internet. For example, Comcast rents me a modem to connect my household to the public Internet; that modem is a public destination with its own 32-bit IPv4 address. (On the other hand, all WiFi devices in my home are on a *private subnet* and don't have public IP addresses.)

There are a few special features of IP addresses that make them interesting.

- As numbers, they have no quantitative meaning. They are just sequences of 1's and 0's (binary digits, bits) that identify things. Like text strings, they are processed *left-to-right.*
- In the human world, IP addresses are written in *dotted-decimal* (aka "CIDR") notation instead of binary numbers: 4 decimal numbers with dots (periods) between them. The external IP address of my Comcast home router is `73.89.155.169`. Each of those four decimal numbers must lie between 0 and 255 inclusive. Why 255? Well, 255 is one less than 2^8, which means that 255 is the largest number that can be expressed using only 8 bits, if we assign a positive weight to each bit[1]. Concatenating four 8-bit numbers adds up to 32 bits. Here is my home IP address in both binary and dotted-decimal:



- To refer to only the first *n* bits of an IP address, *i.e.,* a prefix, append a length indicator to the dotted decimal notation. So `73.89.155.169`**`/10,`** for example, refers to the first 10 bits of my home IP address:



Prefix Routing With IP Addresses

[1] This means that IP addresses, unlike Java's `byte` data type, do not reserve one bit out of 8 to accommodate negative numbers.

When you ship a package using Federal Express, for example, packages from hundreds of planes arrive overnight in Memphis, Tennessee and each must be automatically *routed* to the correct departing plane. An automated *package router* reads the ZIP Code on each package, decides which plane that package should be sent to, and shoves it onto the correct conveyor belt.

Federal Express Router:
(ZIP_code) ➔ conveyor_number

Similarly, an Internet *packet router* is a piece of hardware that moves data around the Internet, similar to the way Federal Express moves packages between conveyor belts. A packet of data arrives at one of a router's *input ports*, tagged with the IP Address for its destination. Based on that IP Address, the router selects one of its *output ports* though which to send the data on toward that destination.

Internet Router:
(IP_Address) ➔ output_port_number

ZIP codes and IP addresses are not arbitrary. Just as ZIP codes are grouped *geographically* by prefix, so IP addresses are grouped *administratively*. Any ZIP code starting in `07` or `08` is in New Jersey, so Federal Express had better put packages with those Zip Codes on a plane bound for New Jersey. Similarly, some prefix of each IP address indicates to which *administrative domain* (*e.g.,* Comcast-USA) a data packet is sent for final delivery. Recall my home IP address: `73.89.155.169`. It so happens that any IP address with 8-bit prefix `01001001` (73 in decimal) belongs to Comcast, just like any ZIP code starting with `07` belongs to New Jersey. So an Internet router would associate any address matching CIDR prefix `73.0.0.0/8` with whichever output port points towards Comcast's domain.[2]

However, `73.0.0.0/8` is not the only prefix that belongs to Comcast. For example, any IP address with the prefix `66.176.0.0/15` also belongs to Comcast. Similarly with the prefix `67.229.0.0/16` In fact there are **70** different prefixes, *of many different lengths*, that all belong to Comcast's domain!! Any Internet router would have to recognize all of these. And Comcast is just one of thousands of providers.

How did Internet routing get to be so messy? Internet Service Providers sell and lease IP address ranges to each other like any other commodity. Also, owners of IP address ranges may hire another company (like Comcast) to manage their Internet. Consider the chaos if states decided to rent each other unused ZIP codes!!

---

[2] Note that `73.0.0.0/8` and `73.89.155.169/8` would both resolve to the same bit string value: `01001001`. When specifying a CIDR prefix, it is *customary* to set any bits that are not part of the prefix to zero, but it is not strictly required. Hence `73.0.0.0/8` is preferred over `73.89.155.169/8`, but they both just mean `01001001`.

Overlapping IP Address Ranges

Furthermore, it often happens that IP address ranges for two different customers can overlap. Suppose company A owns `129.151.0.0/16`. They decide they don't need all that space, so they lease `129.151.192.0/18` (the upper 25% of their address space) to B. Let's look at those in binary:

```
129.151.0.0/16:   1 0 0 0 0 0 0 1 1 0 0 1 0 1 1 1 ➔ A
129.151.192.0/18: 1 0 0 0 0 0 0 1 1 0 0 1 0 1 1 1 1 1 ➔ B
                  |←  first 16 bits identical  →|
```

Suppose a packet arrives addressed to `129.151.208.11`. This is in B's domain since `208 > 192`, but it actually matches the prefixes for both A and B. How would a router know this belongs to B and not A? Clearly, the *longer* of the two prefixes is more specific and restrictive, so we should route this packet to B. Whenever multiple routing rules match a packet, **longest prefix wins!!**

Building the Basic Router [60 points]

Carefully read all the skeleton classes provided with this assignment. You may find many of your questions answered in the JavaDoc or skeleton code provided. In the interests of simplicity, you are not being asked to write separate classes to implement interfaces. Do NOT alter the signatures of any public constructors or methods, or the automated tests will reject your code.

For this assignment, you are provided with a *finished* Java class `IPAddress` (along with a superclass `BitVector`) that represents IP addresses and their prefixes, handling all the details of conversion between CIDR and binary representation. Read the internal JavaDoc for these classes to figure out how to use them.

The auxiliary class `BitVector` inherits from `java.util.BitSet`, and `IPAddress` in turn inherits from `BitVector`. (This is a benign use of Java's "extends" inheritance between concrete classes.) The main difference is that `BitSet` considers bits to represent `boolean` values, while we want them to represent numeric {0,1} values. `BitVector` encapsulates the primitive implementation of a bit sequence while still maintaining the compactness.

Your router is encapsulated by a single class, `IPRouter`, although it will make use of the `IPAddress` class defined earlier. Since it is doing prefix matching, it will also need to make use of a Trie class. A preliminary skeleton for the router class is found in the file `IPRouter.java` and a skeleton for the Trie class is in `BitVectorTrie.java`. *Please read the JavaDoc in those files carefully before you code the method bodies.* You will need to construct a router with N output ports

numbered 0 thru N-1, add rules to the router, and then inject IP addresses to observe the routing decisions being made.

The `TrieST` class in Sedgewick's book can be adapted for `BitVectorTrie` by changing only a couple dozen lines at most and deleting a lot of stuff you don't need. You are no longer matching strings against character prefixes, but rather binary sequences (as one `int` per bit) against binary sequence prefixes. You are also not looking up values by exact match, but just by prefixes, and the longest prefix wins. Note that instead of 256 different values for each prefix element, there are now only two -- 0 and 1 (and not '0' and '1'). This makes a Trie structure about as space-efficient as it could ever be.

Testing your work [20 points]

Your router will need to be tested, obviously. Several useful fragments of Java testing code are provided for you in testing.txt. You should incorporate these into the JUnit test class `TestRouter.java` with proper setup and teardown methods. Each address to be routed should be in its own test case. A few sanity rules are already done for you. You should add any additional rules and test cases you deem necessary to fully exercise all the logic of the router (*e.g.,* testing 3 or more overlapping rules for one address…).

Your testing grade will depend on thoroughly you cover the usual sorts of corner cases as well as the Sunny Day scenarios. Two text files of example routes to use are included in the zip file.

Adding a cache [20 points]

Note: this part is at least as much work as the previous parts, even though it only counts for 1/4 as many points. This is intentional, so that even those who don't complete this part can get a not-totally-horrible grade. Get the previous parts solid first!

A real router is designed around the anticipation that recently seen IP Addresses will shortly be seen again ("temporal locality"). For this reason, you are to add a cache of recently seen addresses and their output ports to your router. A skeleton of this cache is provided in `RouteCache.java`. The cache is to have a maximum capacity of *M distinct* IP addresses, set by the router's constructor; only that number of distinct IP addresses may be cached. Whenever the router tries to route an uncached IP address and the cache is at capacity, the *least recently used* (LRU) IP address must be purged from the cache to make room for the new one. LRU and first-in-first-out (FIFO) are not the same at all, although they are related (see below). *If your cache does not properly implement LRU logic, you are eligible for at most half the points in this section.*

The cache must coordinate two different data structures:

- a hash table for looking up a route in O(1) from an IP address, and
- a bounded-length queue of the *M* most recently accessed distinct IP addresses, where addresses are inserted at the *tail* and removed from the *head* when they age out.

Use `java.util.HashMap` for your hash table.

As long as all addresses coming in are *distinct*, LRU and FIFO work the same. Each new address gets added to the hash table and to the tail of the queue; if the queue was already at capacity, then the address at the head of the queue is removed from both the queue and the hash table.

The LRU part comes if you are asked to route an address that is already cached. In that case, that address must be removed from its current position in the queue and re-inserted at the back of the queue. Furthermore, this operation must be done in constant time, so all the following operations must be O(1):

- finding the location of an IP address in the queue (no linear search allowed!!)
- removing it from its current position
- re-inserting it at the tail of the queue

To find an IP address in a queue without search, the hash table must have some kind of direct reference to it. To be able to remove it from an arbitrary position and re-insert it at the tail O(1), the queue structure must be doubly linked. Standard `java.util.LinkedList` is a doubly linked list, but it unfortunately lacks an O(1) method for removing an object by direct reference. Even if you tell it which object to remove, it still searches linearly for the first object that is `.equals` to the one you specified!! So you will have to implement your own doubly linked queue from the skeleton provided.

Pedantic Note: IPAddress inherits an O(1) `.equals` method in case you were worried about that detail.

Important: Update your unit tests to make sure the cache is fully tested. `IPRouter` and `RouteCache` classes have some package-private methods for testing the cache. You must finish these. Test with an artificially small cache capacity (3-10) to make verification easier. You may add other package-private testing methods to `IPRouter` and `RouteCache` if you feel you need to, but don't make them public.