9

Solutions to System Design and Scalability

9.1 **Stock Data:** Imagine you are building some sort of service that will be called by up to 1,000 client applications to get simple end-of-day stock price information (open, close, high, low). You may assume that you already have the data, and you can store it in any format you wish. How would you design the client-facing service that provides the information to client applications? You are responsible for the development, rollout, and ongoing monitoring and maintenance of the feed. Describe the different methods you considered and why you would recommend your approach. Your service can use any technologies you wish, and can distribute the information to the client applications in any mechanism you choose.

pg 144

SOLUTION

From the statement of the problem, we want to focus on how we actually distribute the information to clients. We can assume that we have some scripts that magically collect the information.

We want to start off by thinking about what the different aspects we should consider in a given proposal are:

- *Client Ease of Use:* We want the service to be easy for the clients to implement and useful for them.
- *Ease for Ourselves:* This service should be as easy as possible for us to implement, as we shouldn't impose unnecessary work on ourselves. We need to consider in this not only the cost of implementing, but also the cost of maintenance.
- *Flexibility for Future Demands:* This problem is stated in a "what would you do in the real world" way, so we should think like we would in a real-world problem. Ideally, we do not want to overly constrain ourselves in the implementation, such that we can't be flexible if the requirements or demands change.
- *Scalability and Efficiency:* We should be mindful of the efficiency of our solution, so as not to overly burden our service.

With this framework in mind, we can consider various proposals.

Proposal #1

One option is that we could keep the data in simple text files and let clients download the data through some sort of FTP server. This would be easy to maintain in some sense, since files can be easily viewed and backed up, but it would require more complex parsing to do any sort of query. And, if additional data were added to our text file, it might break the clients' parsing mechanism.

Proposal #2

We could use a standard SQL database, and let the clients plug directly into that. This would provide the following benefits:

- Facilitates an easy way for the clients to do query processing over the data, in case there are additional features we need to support. For example, we could easily and efficiently perform a query such as "return all stocks having an open price greater than N and a closing price less than M."
- Rolling back, backing up data, and security could be provided using standard database features. We don't have to "reinvent the wheel," so it's easy for us to implement.
- Reasonably easy for the clients to integrate into existing applications. SQL integration is a standard feature in software development environments.

What are the disadvantages of using a SQL database?

- It's much heavier weight than we really need. We don't necessarily need all the complexity of a SQL backend to support a feed of a few bits of information.
- It's difficult for humans to be able to read it, so we'll likely need to implement an additional layer to view and maintain the data. This increases our implementation costs.
- Security: While a SQL database offers pretty well defined security levels, we would still need to be very careful to not give clients access that they shouldn't have. Additionally, even if clients aren't doing anything "malicious," they might perform expensive and inefficient queries, and our servers would bear the costs of that.

These disadvantages don't mean that we shouldn't provide SQL access. Rather, they mean that we should be aware of the disadvantages.

Proposal #3

XML is another great option for distributing the information. Our data has fixed format and fixed size: `company_name`, `open`, `high`, `low`, `closing price`. The XML could look like this:

```
<root>
  <date value="2008-10-12">
    <company name="foo">
      <open>126.23</open>
      <high>130.27</high>
      <low>122.83</low>
      <closingPrice>127.30</closingPrice>
    </company>
    <company name="bar">
      <open>52.73</open>
      <high>60.27</high>
      <low>50.29</low>
      <closingPrice>54.91</closingPrice>
    </company>
  </date>
  <date value="2008-10-11"> . . . </date>
</root>
```

The advantages of this approach include the following:

- It's very easy to distribute, and it can also be easily read by both machines and humans. This is one reason that XML is a standard data model to share and distribute data.
- Most languages have a library to perform XML parsing, so it's reasonably easy for clients to implement.

- We can add new data to the XML file by adding additional nodes. This would not break the client's parser (provided they have implemented their parser in a reasonable way).
- Since the data is being stored as XML files, we can use existing tools for backing up the data. We don't need to implement our own backup tool.

The disadvantages may include:

- This solution sends the clients all the information, even if they only want part of it. It is inefficient in that way.
- Performing any queries on the data requires parsing the entire file.

Regardless of which solution we use for data storage, we could provide a web service (e.g., SOAP) for client data access. This adds a layer to our work, but it can provide additional security, and it may even make it easier for clients to integrate the system.

However—and this is a pro and a con—clients will be limited to grabbing the data only how we expect or want them to. By contrast, in a pure SQL implementation, clients could query for the highest stock price, even if this wasn't a procedure we "expected" them to need.

So which one of these would we use? There's no clear answer. The pure text file solution is probably a bad choice, but you can make a compelling argument for the SQL or XML solution, with or without a web service.

The goal of a question like this is not to see if you get the "correct" answer (there is no single correct answer). Rather, it's to see how you design a system, and how you evaluate trade-offs.

9.2 **Social Network:** How would you design the data structures for a very large social network like Facebook or LinkedIn? Describe how you would design an algorithm to show the shortest path between two people (e.g., Me -> Bob -> Susan -> Jason -> You).

pg 145

SOLUTION

A good way to approach this problem is to remove some of the constraints and solve it for that situation first.

Step 1: Simplify the Problem—Forget About the Millions of Users

First, let's forget that we're dealing with millions of users. Design this for the simple case.

We can construct a graph by treating each person as a node and letting an edge between two nodes indicate that the two users are friends.

If I wanted to find the path between two people, I could start with one person and do a simple breadth-first search.

Why wouldn't a depth-first search work well? First, depth-first search would just find a path. It wouldn't necessarily find the shortest path. Second, even if we just needed any path, it would be very inefficient. Two users might be only one degree of separation apart, but I could search millions of nodes in their "subtrees" before finding this relatively immediate connection.

Alternatively, I could do what's called a bidirectional breadth-first search. This means doing two breadth-first searches, one from the source and one from the destination. When the searches collide, we know we've found a path.

In the implementation, we'll use two classes to help us. `BFSData` holds the data we need for a breadth-first search, such as the `isVisited` hash table and the `toVisit` queue. `PathNode` will represent the path as we're searching it, storing each `Person` and the `previousNode` we visited in this path.

```
1  LinkedList<Person> findPathBiBFS(HashMap<Integer, Person> people, int source,
2                                   int destination) {
3    BFSData sourceData = new BFSData(people.get(source));
4    BFSData destData = new BFSData(people.get(destination));
5
6    while (!sourceData.isFinished() && !destData.isFinished()) {
7      /* Search out from source. */
8      Person collision = searchLevel(people, sourceData, destData);
9      if (collision != null) {
10       return mergePaths(sourceData, destData, collision.getID());
11     }
12
13     /* Search out from destination. */
14     collision = searchLevel(people, destData, sourceData);
15     if (collision != null) {
16       return mergePaths(sourceData, destData, collision.getID());
17     }
18   }
19   return null;
20 }
21
22 /* Search one level and return collision, if any. */
23 Person searchLevel(HashMap<Integer, Person> people, BFSData primary,
24                    BFSData secondary) {
25   /* We only want to search one level at a time. Count how many nodes are
26    * currently in the primary's level and only do that many nodes. We'll continue
27    * to add nodes to the end. */
28   int count = primary.toVisit.size();
29   for (int i = 0; i < count; i++) {
30     /* Pull out first node. */
31     PathNode pathNode = primary.toVisit.poll();
32     int personId = pathNode.getPerson().getID();
33
34     /* Check if it's already been visited. */
35     if (secondary.visited.containsKey(personId)) {
36       return pathNode.getPerson();
37     }
38
39     /* Add friends to queue. */
40     Person person = pathNode.getPerson();
41     ArrayList<Integer> friends = person.getFriends();
42     for (int friendId : friends) {
43       if (!primary.visited.containsKey(friendId)) {
44         Person friend = people.get(friendId);
45         PathNode next = new PathNode(friend, pathNode);
46         primary.visited.put(friendId, next);
47         primary.toVisit.add(next);
48       }
49     }
50   }
51   return null;
52 }
53
```

```
/* Merge paths where searches met at connection. */
LinkedList<Person> mergePaths(BFSData bfs1, BFSData bfs2, int connection) {
  PathNode end1 = bfs1.visited.get(connection); // end1 -> source
  PathNode end2 = bfs2.visited.get(connection); // end2 -> dest
  LinkedList<Person> pathOne = end1.collapse(false);
  LinkedList<Person> pathTwo = end2.collapse(true); // reverse
  pathTwo.removeFirst(); // remove connection
  pathOne.addAll(pathTwo); // add second path
  return pathOne;
}

class PathNode {
  private Person person = null;
  private PathNode previousNode = null;
  public PathNode(Person p, PathNode previous) {
    person = p;
    previousNode = previous;
  }

  public Person getPerson() { return person; }

  public LinkedList<Person> collapse(boolean startsWithRoot) {
    LinkedList<Person> path = new LinkedList<Person>();
    PathNode node = this;
    while (node != null) {
      if (startsWithRoot) {
        path.addLast(node.person);
      } else {
        path.addFirst(node.person);
      }
      node = node.previousNode;
    }
    return path;
  }
}

class BFSData {
  public Queue<PathNode> toVisit = new LinkedList<PathNode>();
  public HashMap<Integer, PathNode> visited =
    new HashMap<Integer, PathNode>();

  public BFSData(Person root) {
    PathNode sourcePath = new PathNode(root, null);
    toVisit.add(sourcePath);
    visited.put(root.getID(), sourcePath);
  }

  public boolean isFinished() {
    return toVisit.isEmpty();
  }
}
```

Many people are surprised that this is faster. Some quick math can explain why.

Suppose every person has k friends, and node S and node D have a friend C in common.

- Traditional breadth-first search from S to D: We go through roughly k+k*k nodes: each of S's k friends, and then each of their k friends.

- Bidirectional breadth-first search: We go through 2k nodes: each of S's k friends and each of D's k friends.

Of course, 2k is much less than k+k*k.

Generalizing this to a path of length q, we have this:

- BFS: $O(k^q)$
- Bidirectional BFS: $O(k^{q/2} + k^{q/2})$, which is just $O(k^{q/2})$

If you imagine a path like A->B->C->D->E where each person has 100 friends, this is a big difference. BFS will require looking at 100 million (100^4) nodes. A bidirectional BFS will require looking at only 20,000 nodes (2×100^2).

A bidirectional BFS will generally be faster than the traditional BFS. However, it requires actually having access to both the source node and the destination nodes, which is not always the case.

Step 2: Handle the Millions of Users

When we deal with a service the size of LinkedIn or Facebook, we cannot possibly keep all of our data on one machine. That means that our simple `Person` data structure from above doesn't quite work—our friends may not live on the same machine as we do. Instead, we can replace our list of friends with a list of their IDs, and traverse as follows:

1. For each friend ID: `int machine_index = getMachineIDForUser(personID);`
2. Go to machine `#machine_index`
3. On that machine, do: `Person friend = getPersonWithID(person_id);`

The code below outlines this process. We've defined a class `Server`, which holds a list of all the machines, and a class `Machine`, which represents a single machine. Both classes have hash tables to efficiently lookup data.

```
1  class Server {
2    HashMap<Integer, Machine> machines = new HashMap<Integer, Machine>();
3    HashMap<Integer, Integer> personToMachineMap = new HashMap<Integer, Integer>();
4
5    public Machine getMachineWithId(int machineID) {
6      return machines.get(machineID);
7    }
8
9    public int getMachineIDForUser(int personID) {
10     Integer machineID = personToMachineMap.get(personID);
11     return machineID == null ? -1 : machineID;
12   }
13
14   public Person getPersonWithID(int personID) {
15     Integer machineID = personToMachineMap.get(personID);
16     if (machineID == null) return null;
17
18     Machine machine = getMachineWithId(machineID);
19     if (machine == null) return null;
20
21     return machine.getPersonWithID(personID);
22   }
23 }
24
25 class Person {
```

```
26    private ArrayList<Integer> friends = new ArrayList<Integer>();
27    private int personID;
28    private String info;
29
30    public Person(int id) { this.personID = id; }
31    public String getInfo() { return info; }
32    public void setInfo(String info) { this.info = info; }
33    public ArrayList<Integer> getFriends() { return friends; }
34    public int getID() { return personID; }
35    public void addFriend(int id) { friends.add(id); }
36 }
```

There are more optimizations and follow-up questions here than we could possibly discuss, but here are just a few possibilities.

Optimization: Reduce machine jumps

Jumping from one machine to another is expensive. Instead of randomly jumping from machine to machine with each friend, try to batch these jumps—e.g., if five of my friends live on one machine, I should look them up all at once.

Optimization: Smart division of people and machines

People are much more likely to be friends with people who live in the same country as they do. Rather than randomly dividing people across machines, try to divide them by country, city, state, and so on. This will reduce the number of jumps.

Question: Breadth-first search usually requires "marking" a node as visited. How do you do that in this case?

Usually, in BFS, we mark a node as visited by setting a `visited` flag in its node class. Here, we don't want to do that. There could be multiple searches going on at the same time, so it's a bad idea to just edit our data.

Instead, we could mimic the marking of nodes with a hash table to look up a node id and determine whether it's been visited.

Other Follow-Up Questions:

- In the real world, servers fail. How does this affect you?
- How could you take advantage of caching?
- Do you search until the end of the graph (infinite)? How do you decide when to give up?
- In real life, some people have more friends of friends than others, and are therefore more likely to make a path between you and someone else. How could you use this data to pick where to start traversing?

These are just a few of the follow-up questions you or the interviewer could raise. There are many others.

9.3 Web Crawler: If you were designing a web crawler, how would you avoid getting into infinite loops?

pg 145

SOLUTION

The first thing to ask ourselves in this problem is how an infinite loop might occur. The simplest answer is that, if we picture the web as a graph of links, an infinite loop will occur when a cycle occurs.

To prevent infinite loops, we just need to detect cycles. One way to do this is to create a hash table where we set `hash[v]` to `true` after we visit page v.

We can crawl the web using breadth-first search. Each time we visit a page, we gather all its links and insert them at the end of a queue. If we've already visited a page, we ignore it.

This is great—but what does it mean to visit page v? Is page v defined based on its content or its URL?

If it's defined based on its URL, we must recognize that URL parameters might indicate a completely different page. For example, the page `www.careercup.com/page?pid=microsoft-interview-questions` is totally different from the page `www.careercup.com/page?pid=google-interview-questions`. But, we can also append URL parameters arbitrarily to any URL without truly changing the page, provided it's not a parameter that the web application recognizes and handles. The page `www.careercup.com?foobar=hello` is the same as `www.careercup.com`.

"Okay, then," you might say, "let's define it based on its content." That sounds good too, at first, but it also doesn't quite work. Suppose I have some randomly generated content on the careercup.com home page. Is it a different page each time you visit it? Not really.

The reality is that there is probably no perfect way to define a "different" page, and this is where this problem gets tricky.

One way to tackle this is to have some sort of estimation for degree of similarity. If, based on the content and the URL, a page is deemed to be sufficiently similar to other pages, we *deprioritize* crawling its children. For each page, we would come up with some sort of signature based on snippets of the content and the page's URL.

Let's see how this would work.

We have a database which stores a list of items we need to crawl. On each iteration, we select the highest priority page to crawl. We then do the following:

1. Open up the page and create a signature of the page based on specific subsections of the page and its URL.
2. Query the database to see whether anything with this signature has been crawled recently.
3. If something with this signature has been recently crawled, insert this page back into the database at a low priority.
4. If not, crawl the page and insert its links into the database.

Under the above implementation, we never "complete" crawling the web, but we will avoid getting stuck in a loop of pages. If we want to allow for the possibility of "finishing" crawling the web (which would clearly happen only if the "web" were actually a smaller system, like an intranet), then we can set a minimum priority that a page must have to be crawled.

This is just one, simplistic solution, and there are many others that are equally valid. A problem like this will more likely resemble a conversation with your interviewer which could take any number of paths. In fact, the discussion of this problem could have taken the path of the very next problem.

9.4 Duplicate URLs: You have 10 billion URLs. How do you detect the duplicate documents? In this case, assume "duplicate" means that the URLs are identical.

pg 145

SOLUTION

Just how much space do 10 billion URLs take up? If each URL is an average of 100 characters, and each character is 4 bytes, then this list of 10 billion URLs will take up about 4 terabytes. We are probably not going to hold that much data in memory.

But, let's just pretend for a moment that we were miraculously holding this data in memory, since it's useful to first construct a solution for the simple version. Under this version of the problem, we would just create a hash table where each URL maps to `true` if it's already been found elsewhere in the list. (As an alternative solution, we could sort the list and look for the duplicate values that way. That will take a bunch of extra time and offers few advantages.)

Now that we have a solution for the simple version, what happens when we have all 4000 gigabytes of data and we can't store it all in memory? We could solve this either by storing some of the data on disk or by splitting up the data across machines.

Solution #1: Disk Storage

If we stored all the data on one machine, we would do two passes of the document. The first pass would split the list of URLs into 4000 chunks of 1 GB each. An easy way to do that might be to store each URL u in a file named `<x>.txt` where `x = hash(u) % 4000`. That is, we divide up the URLs based on their hash value (modulo the number of chunks). This way, all URLs with the same hash value would be in the same file.

In the second pass, we would essentially implement the simple solution we came up with earlier: load each file into memory, create a hash table of the URLs, and look for duplicates.

Solution #2: Multiple Machines

The other solution is to perform essentially the same procedure, but to use multiple machines. In this solution, rather than storing the data in file `<x>.txt`, we would send the URL to machine `x`.

Using multiple machines has pros and cons.

The main pro is that we can parallelize the operation, such that all 4000 chunks are processed simultaneously. For large amounts of data, this might result in a faster solution.

The disadvantage though is that we are now relying on 4000 different machines to operate perfectly. That may not be realistic (particularly with more data and more machines), and we'll need to start considering how to handle failure. Additionally, we have increased the complexity of the system simply by involving so many machines.

Both are good solutions, though, and both should be discussed with your interviewer.

9.5 **Cache:** Imagine a web server for a simplified search engine. This system has 100 machines to respond to search queries, which may then call out using `processSearch(string query)` to another cluster of machines to actually get the result. The machine which responds to a given query is chosen at random, so you cannot guarantee that the same machine will always respond to the same request. The method `processSearch` is very expensive. Design a caching mechanism to cache the results of the most recent queries. Be sure to explain how you would update the cache when data changes.

pg 145

SOLUTION

Before getting into the design of this system, we first have to understand what the question means. Many of the details are somewhat ambiguous, as is expected in questions like this. We will make reasonable assumptions for the purposes of this solution, but you should discuss these details—in depth—with your interviewer.

Assumptions

Here are a few of the assumptions we make for this solution. Depending on the design of your system and how you approach the problem, you may make other assumptions. Remember that while some approaches are better than others, there is no one "correct" approach.

- Other than calling out to `processSearch` as necessary, all query processing happens on the initial machine that was called.
- The number of queries we wish to cache is large (millions).
- Calling between machines is relatively quick.
- The result for a given query is an ordered list of URLs, each of which has an associated 50 character title and 200 character summary.
- The most popular queries are extremely popular, such that they would always appear in the cache.

Again, these aren't the *only* valid assumptions. This is just one reasonable set of assumptions.

System Requirements

When designing the cache, we know we'll need to support two primary functions:

- Efficient lookups given a key.
- Expiration of old data so that it can be replaced with new data.

In addition, we must also handle updating or clearing the cache when the results for a query change. Because some queries are very common and may permanently reside in the cache, we cannot just wait for the cache to naturally expire.

Step 1: Design a Cache for a Single System

A good way to approach this problem is to start by designing it for a single machine. So, how would you create a data structure that enables you to easily purge old data and also efficiently look up a value based on a key?

- A linked list would allow easy purging of old data, by moving "fresh" items to the front. We could implement it to remove the last element of the linked list when the list exceeds a certain size.

- A hash table allows efficient lookups of data, but it wouldn't ordinarily allow easy data purging.

How can we get the best of both worlds? By merging the two data structures. Here's how this works:

Just as before, we create a linked list where a node is moved to the front every time it's accessed. This way, the end of the linked list will always contain the stalest information.

In addition, we have a hash table that maps from a query to the corresponding node in the linked list. This allows us to not only efficiently return the cached results, but also to move the appropriate node to the front of the list, thereby updating its "freshness."

For illustrative purposes, abbreviated code for the cache is below. The code attachment provides the full code for this part. Note that in your interview, it is unlikely that you would be asked to write the full code for this as well as perform the design for the larger system.

```
1  public class Cache {
2    public static int MAX_SIZE = 10;
3    public Node head, tail;
4    public HashMap<String, Node> map;
5    public int size = 0;
6
7    public Cache() {
8      map = new HashMap<String, Node>();
9    }
10
11   /* Moves node to front of linked list */
12   public void moveToFront(Node node) { ... }
13   public void moveToFront(String query) { ... }
14
15   /* Removes node from linked list */
16   public void removeFromLinkedList(Node node) { ... }
17
18   /* Gets results from cache, and updates linked list */
19   public String[] getResults(String query) {
20     if (!map.containsKey(query)) return null;
21
22     Node node = map.get(query);
23     moveToFront(node); // update freshness
24     return node.results;
25   }
26
27   /* Inserts results into linked list and hash */
28   public void insertResults(String query, String[] results) {
29     if (map.containsKey(query)) { // update values
30       Node node = map.get(query);
31       node.results = results;
32       moveToFront(node); // update freshness
33       return;
34     }
35
36     Node node = new Node(query, results);
37     moveToFront(node);
38     map.put(query, node);
39
40     if (size > MAX_SIZE) {
41       map.remove(tail.query);
42       removeFromLinkedList(tail);
43     }
```

```
44      }
45  }
```

Step 2: Expand to Many Machines

Now that we understand how to design this for a single machine, we need to understand how we would design this when queries could be sent to many different machines. Recall from the problem statement that there's no guarantee that a particular query will be consistently sent to the same machine.

The first thing we need to decide is to what extent the cache is shared across machines. We have several options to consider.

Option 1: Each machine has its own cache.

A simple option is to give each machine its own cache. This means that if "foo" is sent to machine 1 twice in a short amount of time, the result would be recalled from the cache on the second time. But, if "foo" is sent first to machine 1 and then to machine 2, it would be treated as a totally fresh query both times.

This has the advantage of being relatively quick, since no machine-to-machine calls are used. The cache, unfortunately, is somewhat less effective as an optimization tool as many repeat queries would be treated as fresh queries.

Option 2: Each machine has a copy of the cache.

On the other extreme, we could give each machine a complete copy of the cache. When new items are added to the cache, they are sent to all machines. The entire data structure—linked list and hash table—would be duplicated.

This design means that common queries would nearly always be in the cache, as the cache is the same everywhere. The major drawback however is that updating the cache means firing off data to N different machines, where N is the size of the response cluster. Additionally, because each item effectively takes up N times as much space, our cache would hold much less data.

Option 3: Each machine stores a segment of the cache.

A third option is to divide up the cache, such that each machine holds a different part of it. Then, when machine `i` needs to look up the results for a query, machine `i` would figure out which machine holds this value, and then ask this other machine (machine `j`) to look up the query in `j`'s cache.

But how would machine `i` know which machine holds this part of the hash table?

One option is to assign queries based on the formula `hash(query) % N`. Then, machine `i` only needs to apply this formula to know that machine `j` should store the results for this query.

So, when a new query comes in to machine `i`, this machine would apply the formula and call out to machine `j`. Machine `j` would then return the value from its cache or call `processSearch(query)` to get the results. Machine `j` would update its cache and return the results back to `i`.

Alternatively, you could design the system such that machine `j` just returns `null` if it doesn't have the `query` in its current cache. This would require machine `i` to call `processSearch` and then forward the results to machine `j` for storage. This implementation actually increases the number of machine-to-machine calls, with few advantages.

Step 3: Updating results when contents change

Recall that some queries may be so popular that, with a sufficiently large cache, they would permanently be cached. We need some sort of mechanism to allow cached results to be refreshed, either periodically or "on-demand" when certain content changes.

To answer this question, we need to consider when results would change (and you need to discuss this with your interviewer). The primary times would be when:

1. The content at a URL changes (or the page at that URL is removed).
2. The ordering of results change in response to the rank of a page changing.
3. New pages appear related to a particular query.

To handle situations #1 and #2, we could create a separate hash table that would tell us which cached queries are tied to a specific URL. This could be handled completely separately from the other caches, and reside on different machines. However, this solution may require a lot of data.

Alternatively, if the data doesn't require instant refreshing (which it probably doesn't), we could periodically crawl through the cache stored on each machine to purge queries tied to the updated URLs.

Situation #3 is substantially more difficult to handle. We could update single word queries by parsing the content at the new URL and purging these one-word queries from the caches. But, this will only handle the one-word queries.

A good way to handle Situation #3 (and likely something we'd want to do anyway) is to implement an "automatic time-out" on the cache. That is, we'd impose a time out where *no* query, regardless of how popular it is, can sit in the cache for more than X minutes. This will ensure that all data is periodically refreshed.

Step 4: Further Enhancements

There are a number of improvements and tweaks you could make to this design depending on the assumptions you make and the situations you optimize for.

One such optimization is to better support the situation where some queries are very popular. For example, suppose (as an extreme example) a particular string constitutes 1% of all queries. Rather than machine `i` forwarding the request to machine `j` every time, machine `i` could forward the request just once to `j`, and then `i` could store the results in its own cache as well.

Alternatively, there may also be some possibility of doing some sort of re-architecture of the system to assign queries to machines based on their hash value (and therefore the location of the cache), rather than randomly. However, this decision may come with its own set of trade-offs.

Another optimization we could make is to the "automatic time out" mechanism. As initially described, this mechanism purges any data after X minutes. However, we may want to update some data (like current news) much more frequently than other data (like historical stock prices). We could implement timeouts based on topic or based on URLs. In the latter situation, each URL would have a time out value based on how frequently the page has been updated in the past. The time out for the query would be the minimum of the time outs for each URL.

These are just a few of the enhancements we can make. Remember that in questions like this, there is no single correct way to solve the problem. These questions are about having a discussion with your interviewer about design criteria and demonstrating your general approach and methodology.

9.6 **Sales Rank**: A large eCommerce company wishes to list the best-selling products, overall and by category. For example, one product might be the #1056th best-selling product overall but the #13th best-selling product under "Sports Equipment" and the #24th best-selling product under "Safety." Describe how you would design this system.

pg 145

SOLUTION

Let's first start off by making some assumptions to define the problem.

Step 1: Scope the Problem

First, we need to define what exactly we're building.

- We'll assume that we're only being asked to design the components relevant to this question, and not the entire eCommerce system. In this case, we might touch the design of the frontend and purchase components, but only as it impacts the sales rank.
- We should also define what the sales rank means. Is it total sales over all time? Sales in the last month? Last week? Or some more complicated function (such as one involving some sort of exponential decay of sales data)? This would be something to discuss with your interviewer. We will assume that it is simply the total sales over the past week.
- We will assume that each product can be in multiple categories, and that there is no concept of "subcategories."

This part just gives us a good idea of what the problem, or scope of features, is.

Step 2: Make Reasonable Assumptions

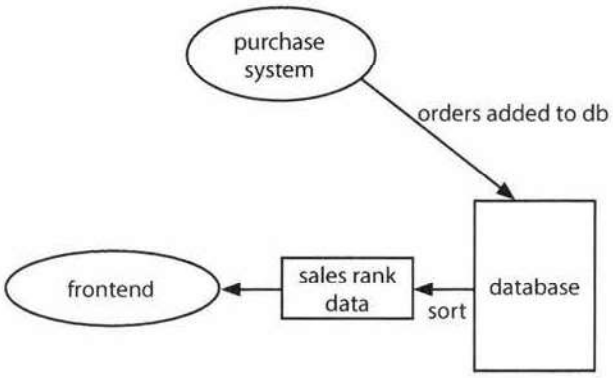
These are the sorts of things you'd want to discuss with your interviewer. Because we don't have an interviewer in front of us, we'll have to make some assumptions.

- We will assume that the stats do not need to be 100% up-to-date. Data can be up to an hour old for the most popular items (for example, top 100 in each category), and up to one day old for the less popular items. That is, few people would care if the #2,809,132th best-selling item should have actually been listed as #2,789,158th instead.
- Precision is important for the most popular items, but a small degree of error is okay for the less popular items.
- We will assume that the data should be updated every hour (for the most popular items), but the time range for this data does not need to be precisely the last seven days (168 hours). If it's sometimes more like 150 hours, that's okay.
- We will assume that the categorizations are based strictly on the origin of the transaction (i.e., the seller's name), not the price or date.

The important thing is not so much which decision you made at each possible issue, but whether it occurred to you that these are assumptions. We should get out as many of these assumptions as possible in the beginning. It's possible you will need to make other assumptions along the way.

Step 3: Draw the Major Components

We should now design just a basic, naive system that describes the major components. This is where you would go up to a whiteboard.



In this simple design, we store every order as soon as it comes into the database. Every hour or so, we pull sales data from the database by category, compute the total sales, sort it, and store it in some sort of sales rank data cache (which is probably held in memory). The frontend just pulls the sales rank from this table, rather than hitting the standard database and doing its own analytics.

Step 4: Identify the Key Issues

Analytics are Expensive

In the naive system, we periodically query the database for the number of sales in the past week for each product. This will be fairly expensive. That's running a query over all sales for all time.

Our database just needs to track the total sales. We'll assume (as noted in the beginning of the solution) that the general storage for purchase history is taken care of in other parts of the system, and we just need to focus on the sales data analytics.

Instead of listing every purchase in our database, we'll store just the total sales from the last week. Each purchase will just update the total weekly sales.

Tracking the total sales takes a bit of thought. If we just use a single column to track the total sales over the past week, then we'll need to re-compute the total sales every day (since the specific days covered in the last seven days change with each day). That is unnecessarily expensive.

Instead, we'll just use a table like this.

Prod ID	Total	Sun	Mon	Tues	Wed	Thurs	Fri	Sat

This is essentially like a circular array. Each day, we clear out the corresponding day of the week. On each purchase, we update the total sales count for that product on that day of the week, as well as the total count.

We will also need a separate table to store the associations of product IDs and categories.

Prod ID	Category ID

To get the sales rank per category, we'll need to join these tables.

Database Writes are Very Frequent

Even with this change, we'll still be hitting the database very frequently. With the amount of purchases that could come in every second, we'll probably want to batch up the database writes.

Instead of immediately committing each purchase to the database, we could store purchases in some sort of in-memory cache (as well as to a log file as a backup). Periodically, we'll process the log / cache data, gather the totals, and update the database.

> We should quickly think about whether or not it's feasible to hold this in memory. If there are 10 million products in the system, can we store each (along with a count) in a hash table? Yes. If each product ID is four bytes (which is big enough to hold up to 4 billion unique IDs) and each count is four bytes (more than enough), then such a hash table would only take about 40 megabytes. Even with some additional overhead and substantial system growth, we would still be able to fit this all in memory.

After updating the database, we can re-run the sales rank data.

We need to be a bit careful here, though. If we process one product's logs before another's, and re-run the stats in between, we could create a bias in the data (since we're including a larger timespan for one product than its "competing" product).

We can resolve this by either ensuring that the sales rank doesn't run until all the stored data is processed (difficult to do when more and more purchases are coming in), or by dividing up the in-memory cache by some time period. If we update the database for all the stored data up to a particular moment in time, this ensures that the database will not have biases.

Joins are Expensive

We have potentially tens of thousands of product categories. For each category, we'll need to first pull the data for its items (possibly through an expensive join) and then sort those.

Alternatively, we could just do one join of products and categories, such that each product will be listed once per category. Then, if we sorted that on category and then product ID, we could just walk the results to get the sales rank for each category.

Prod ID	Category	Total	Sun	Mon	Tues	Wed	Thurs	Fri	Sat
1423	sportseq	13	4	1	4	19	322	32	232
1423	safety	13	4	1	4	19	322	32	232

Rather than running thousands of queries (one for each category), we could sort the data on the category first and then the sales volume. Then, if we walked those results, we would get the sales rank for each category. We would also need to do one sort of the entire table on just sales number, to get the overall rank.

We could also just keep the data in a table like this from the beginning, rather than doing joins. This would require us to update multiple rows for each product.

Database Queries Might Still Be Expensive

Alternatively, if the queries and writes get very expensive, we could consider forgoing a database entirely and just using log files. This would allow us to take advantage of something like MapReduce.

Under this system, we would write a purchase to a simple text file with the product ID and time stamp. Each category has its own directory, and each purchase gets written to all the categories associated with that product.

We would run frequent jobs to merge files together by product ID and time ranges, so that eventually all purchases in a given day (or possibly hour) were grouped together.

```
/sportsequipment
    1423,Dec 13 08:23-Dec 13 08:23,1
    4221,Dec 13 15:22-Dec 15 15:45,5
    ...
/safety
    1423,Dec 13 08:23-Dec 13 08:23,1
    5221,Dec 12 03:19-Dec 12 03:28,19
...
```

To get the best-selling products within each category, we just need to sort each directory.

How do we get the overall ranking? There are two good approaches:

- We could treat the general category as just another directory, and write every purchase to that directory. That would mean a lot of files in this directory.
- Or, since we'll already have the products sorted by sales volume order for each category, we can also do an N-way merge to get the overall rank.

Alternatively, we can take advantage of the fact that the data doesn't need (as we assumed earlier) to be 100% up-to-date. We just need the most popular items to be up-to-date.

We can merge the most popular items from each category in a pairwise fashion. So, two categories get paired together and we merge the most popular items (the first 100 or so). After we have 100 items in this sorted order, we stop merging this pair and move onto the next pair.

To get the ranking for all products, we can be much lazier and only run this work once a day.

One of the advantages of this is that it scales nicely. We can easily divide up the files across multiple servers, as they aren't dependent on each other.

Follow Up Questions

The interviewer could push this design in any number of directions.

- Where do you think you'd hit the next bottlenecks? What would you do about that?
- What if there were subcategories as well? So items could be listed under "Sports" and "Sports Equipment" (or even "Sports" > "Sports Equipment" > "Tennis" > "Rackets")?
- What if data needed to be more accurate? What if it needed to be accurate within 30 minutes for all products?

Think through your design carefully and analyze it for the tradeoffs. You might also be asked to go into more detail on any specific aspect of the product.

9.7 **Personal Financial Manager**: Explain how you would design a personal financial manager (like Mint.com). This system would connect to your bank accounts, analyze your spending habits, and make recommendations.

pg 145

SOLUTION

The first thing we need to do is define what it is exactly that we are building.

Step 1: Scope the Problem

Ordinarily, you would clarify this system with your interviewer. We'll scope the problem as follows:

- You create an account and add your bank accounts. You can add multiple bank accounts. You can also add them at a later point in time.
- It pulls in all your financial history, or as much of it as your bank will allow.
- This financial history includes outgoing money (things you bought or paid for), incoming money (salary and other payments), and your current money (what's in your bank account and investments).
- Each payment transaction has a "category" associated with it (food, travel, clothing, etc.).
- There is some sort of data source provided that tells the system, with some reliability, which category a transaction is associated with. The user might, in some cases, override the category when it's improperly assigned (e.g., eating at the cafe of a department store getting assigned to "clothing" rather than "food").
- Users will use the system to get recommendations on their spending. These recommendations will come from a mix of "typical" users ("people generally shouldn't spend more than X% of their income on clothing"), but can be overridden with custom budgets. This will not be a primary focus right now.
- We assume this is just a website for now, although we could potentially talk about a mobile app as well.
- We probably want email notifications either on a regular basis, or on certain conditions (spending over a certain threshold, hitting a budget max, etc.).
- We'll assume that there's no concept of user-specified rules for assigning categories to transactions.

This gives us a basic goal for what we want to build.

Step 2: Make Reasonable Assumptions

Now that we have the basic goal for the system, we should define some further assumptions about the characteristics of the system.

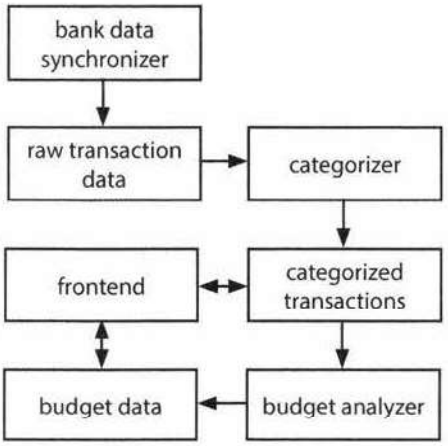
- Adding or removing bank accounts is relatively unusual.
- The system is write-heavy. A typical user may make several new transactions daily, although few users would access the website more than once a week. In fact, for many users, their primary interaction might be through email alerts.
- Once a transaction is assigned to a category, it will only be changed if the user asks to change it. The system will never reassign a transaction to a different category "behind the scenes", even if the rules change. This means that two otherwise identical transactions could be assigned to different categories if the rules changed in between each transaction's date. We do this because it may confuse users if their spending per category changes with no action on their part.
- The banks probably won't push data to our system. Instead, we will need to pull data from the banks.
- Alerts on users exceeding budgets probably do not need to be sent instantaneously. (That wouldn't be realistic anyway, since we won't get the transaction data instantaneously.) It's probably pretty safe for them to be up to 24 hours delayed.

It's okay to make different assumptions here, but you should explicitly state them to your interviewer.

Step 3: Draw the Major Components

The most naive system would be one that pulls bank data on each login, categorizes all the data, and then analyzes the user's budget. This wouldn't quite fit the requirements, though, as we want email notifications on particular events.

We can do a bit better.



With this basic architecture, the bank data is pulled at periodic times (hourly or daily). The frequency may depend on the behavior of the users. Less active users may have their accounts checked less frequently.

Once new data arrives, it is stored in some list of raw, unprocessed transactions. This data is then pushed to the categorizer, which assigns each transaction to a category and stores these categorized transactions in another datastore.

The budget analyzer pulls in the categorized transactions, updates each user's budget per category, and stores the user's budget.

The frontend pulls data from both the categorized transactions datastore as well as from the budget datastore. Additionally, a user could also interact with the frontend by changing the budget or the categorization of their transactions.

Step 4: Identify the Key Issues

We should now reflect on what the major issues here might be.

This will be a very data-heavy system. We want it to feel snappy and responsive, though, so we'll want as much processing as possible to be asynchronous.

We will almost certainly want at least one task queue, where we can queue up work that needs to be done. This work will include tasks such as pulling in new bank data, re-analyzing budgets, and categorizing new bank data. It would also include re-trying tasks that failed.

These tasks will likely have some sort of priority associated with them, as some need to be performed more often than others. We want to build a task queue system that can prioritize some task types over others, while still ensuring that all tasks will be performed eventually. That is, we wouldn't want a low priority task to essentially "starve" because there are always higher priority tasks.

One important part of the system that we haven't yet addressed will be the email system. We could use a task to regularly crawl user's data to check if they're exceeding their budget, but that means checking every

single user daily. Instead, we'll want to queue a task whenever a transaction occurs that potentially exceeds a budget. We can store the current budget totals by category to make it easy to understand if a new transaction exceeds the budget.

We should also consider incorporating the knowledge (or assumption) that a system like this will probably have a large number of inactive users—users who signed up once and then haven't touched the system since. We may want to either remove them from the system entirely or deprioritize their accounts. We'll want some system to track their account activity and associate priority with their accounts.

The biggest bottleneck in our system will likely be the massive amount of data that needs to be pulled and analyzed. We should be able to fetch the bank data asynchronously and run these tasks across many servers. We should drill a bit deeper into how the categorizer and budget analyzer work.

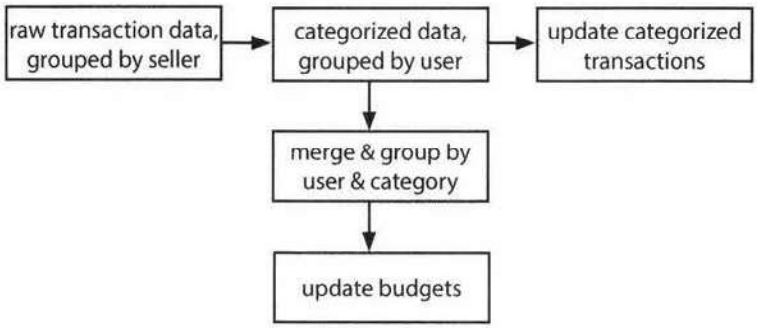
Categorizer and Budget Analyzer

One thing to note is that transactions are not dependent on each other. As soon as we get a transaction for a user, we can categorize it and integrate this data. It might be inefficient to do so, but it won't cause any inaccuracies.

Should we use a standard database for this? With lots of transactions coming in at once, that might not be very efficient. We certainly don't want to do a bunch of joins.

It may be better instead to just store the transactions to a set of flat text files. We assumed earlier that the categorizations are based on the seller's name alone. If we're assuming a lot of users, then there will be a lot of duplicates across the sellers. If we group the transaction files by seller's name, we can take advantage of these duplicates.

The categorizer can do something like this:



It first gets the raw transaction data, grouped by seller. It picks the appropriate category for the seller (which might be stored in a cache for the most common sellers), and then applies that category to all those transactions.

After applying the category, it re-groups all the transactions by user. Then, those transactions are inserted into the datastore for this user.

before categorizer	after categorizer

amazon/
  user121,$5.43,Aug 13
  user922,$15.39,Aug 27
  ...
comcast/
  user922,$9.29,Aug 24
  user248,$40.13,Aug 18
  ...
	user121/
	  amazon,shopping,$5.43,Aug 13
	  ...
	user922/
	  amazon,shopping,$15.39,Aug 27
	  comcast,utilities,$9.29,Aug 24
	  ...
	user248/
	  comcast,utilities,$40.13,Aug 18
	  ...

Then, the budget analyzer comes in. It takes the data grouped by user, merges it across categories (so all Shopping tasks for this user in this timespan are merged), and then updates the budget.

Most of these tasks will be handled in simple log files. Only the final data (the categorized transactions and the budget analysis) will be stored in a database. This minimizes writing and reading from the database.

User Changing Categories

The user might selectively override particular transactions to assign them to a different category. In this case, we would update the datastore for the categorized transactions. It would also signal a quick recomputation of the budget to decrement the item from the old category and increment the item in the other category.

We could also just recompute the budget from scratch. The budget analyzer is fairly quick as it just needs to look over the past few weeks of transactions for a single user.

Follow Up Questions

- How would this change if you also needed to support a mobile app?
- How would you design the component which assigns items to each category?
- How would you design the recommended budgets feature?
- How would you change this if the user could develop rules to categorize all transactions from a particular seller differently than the default?

9.8 **Pastebin**: Design a system like Pastebin, where a user can enter a piece of text and get a randomly generated URL for public access.

pg 145

SOLUTION

We can start with clarifying the specifics of this system.

Step 1: Scope the Problem

- The system does not support user accounts or editing documents.
- The system tracks analytics of how many times each page is accessed.
- Old documents get deleted after not being accessed for a sufficiently long period of time.
- While there isn't true authentication on accessing documents, users should not be able to "guess" docu-

ment URLs easily.

- The system has a frontend as well as an API.
- The analytics for each URL can be accessed through a "stats" link on each page. It is not shown by default, though.

Step 2: Make Reasonable Assumptions

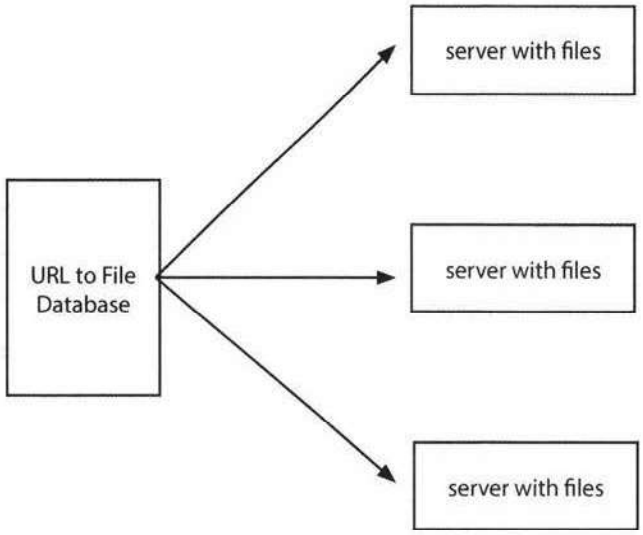
- The system gets heavy traffic and contains many millions of documents.
- Traffic is not equally distributed across documents. Some documents get much more access than others.

Step 3: Draw the Major Components

We can sketch out a simple design. We'll need to keep track of URLs and the files associated with them, as well as analytics for how often the files have been accessed.

How should we store the documents? We have two options: we can store them in a database or we can store them on a file. Since the documents can be large and it's unlikely we need searching capabilities, storing them on a file is probably the better choice.

A simple design like this might work well:



Here, we have a simple database that looks up the location (server and path) of each file. When we have a request for a URL, we look up the location of the URL within the datastore and then access the file.

Additionally, we will need a database that tracks analytics. We can do this with a simple datastore that adds each visit (including timestamp, IP address, and location) as a row in a database. When we need to access the stats of each visit, we pull the relevant data in from this database.

Step 4: Identify the Key Issues

The first issue that comes to mind is that some documents will be accessed much more frequently than others. Reading data from the filesystem is relatively slow compared with reading from data in memory. Therefore, we probably want to use a cache to store the most recently accessed documents. This will ensure

that items accessed very frequently (or very recently) will be quickly accessible. Since documents cannot be edited, we will not need to worry about invalidating this cache.

We should also potentially consider sharding the database. We can shard it using some mapping from the URL (for example, the URL's hash code modulo some integer), which will allow us to quickly locate the database which contains this file.

In fact, we could even take this a step further. We could skip the database entirely and just let a hash of the URL indicate which server contains the document. The URL itself could reflect the location of the document. One potential issue from this is that if we need to add servers, it could be difficult to redistribute the documents.

Generating URLs

We have not yet discussed how to actually generate the URLs. We probably do not want a monotonically increasing integer value, as this would be easy for a user to "guess." We want URLs to be difficult to access without being provided the link.

One simple path is to generate a random GUID (e.g., 5d50e8ac-57cb-4a0d-8661-bcdee2548979). This is a 128-bit value that, while not strictly guaranteed to be unique, has low enough odds of a collision that we can treat it as unique. The drawback of this plan is that such a URL is not very "pretty" to the user. We could hash it to a smaller value, but then that increases the odds of collision.

We could do something very similar, though. We could just generate a 10-character sequence of letters and numbers, which gives us 36^{10} possible strings. Even with a billion URLs, the odds of a collision on any specific URL are very low.

> This is not to say that the odds of a collision over the whole system are low. They are not. Any one specific URL is unlikely to collide. However, after storing a billion URLs, we are very likely to have a collision at some point.

Assuming that we aren't okay with periodic (even if unusual) data loss, we'll need to handle these collisions. We can either check the datastore to see if the URL exists yet or, if the URL maps to a specific server, just detect whether a file already exists at the destination.

When a collision occurs, we can just generate a new URL. With 36^{10} possible URLs, collisions would be rare enough that the lazy approach here (detect collisions and retry) is sufficient.

Analytics

The final component to discuss is the analytics piece. We probably want to display the number of visits, and possibly break this down by location or time.

We have two options here:

- Store the raw data from each visit.
- Store just the data we know we'll use (number of visits, etc.).

You can discuss this with your interviewer, but it probably makes sense to store the raw data. We never know what features we'll add to the analytics down the road. The raw data allows us flexibility.

This does not mean that the raw data needs to be easily searchable or even accessible. We can just store a log of each visit in a file, and back this up to other servers.

One issue here is that this amount of data could be substantial. We could potentially reduce the space usage considerably by storing data only probabilistically. Each URL would have a `storage_probability` associated with it. As the popularity of a site goes up, the `storage_probability` goes down. For example, a popular document might have data logged only one out of every ten times, at random. When we look up the number of visits for the site, we'll need to adjust the value based on the probability (for example, by multiplying it by 10). This will of course lead to a small inaccuracy, but that may be acceptable.

The log files are not designed to be used frequently. We will want to also store this precomputed data in a datastore. If the analytics just displays the number of visits plus a graph over time, this could be kept in a separate database.

URL	Month and Year	Visits
12ab31b92p	December 2013	242119
12ab31b92p	January 2014	429918
...	...	...

Every time a URL is visited, we can increment the appropriate row and column. This datastore can also be sharded by the URL.

As the stats are not listed on the regular pages and would generally be of less interest, it should not face as heavy of a load. We could still cache the generated HTML on the frontend servers, so that we don't continuously reaccess the data for the most popular URLs.

Follow-Up Questions

- How would you support user accounts?
- How would you add a new piece of analytics (e.g., referral source) to the stats page?
- How would your design change if the stats were shown with each document?

10

Solutions to Sorting and Searching

10.1 Sorted Merge: You are given two sorted arrays, A and B, where A has a large enough buffer at the end to hold B. Write a method to merge B into A in sorted order.

pg 149

SOLUTION

Since we know that A has enough buffer at the end, we won't need to allocate additional space. Our logic should involve simply comparing elements of A and B and inserting them in order, until we've exhausted all elements in A and in B.

The only issue with this is that if we insert an element into the front of A, then we'll have to shift the existing elements backwards to make room for it. It's better to insert elements into the back of the array, where there's empty space.

The code below does just that. It works from the back of A and B, moving the largest elements to the back of A.

```
1  void merge(int[] a, int[] b, int lastA, int lastB) {
2     int indexA = lastA - 1; /* Index of last element in array a */
3     int indexB = lastB - 1; /* Index of last element in array b */
4     int indexMerged = lastB + lastA - 1; /* end of merged array */
5
6     /* Merge a and b, starting from the last element in each */
7     while (indexB >= 0) {
8        /* end of a is > than end of b */
9        if (indexA >= 0 && a[indexA] > b[indexB]) {
10          a[indexMerged] = a[indexA]; // copy element
11          indexA--;
12       } else {
13          a[indexMerged] = b[indexB]; // copy element
14          indexB--;
15       }
16       indexMerged--; // move indices
17    }
18 }
```

Note that you don't need to copy the contents of A after running out of elements in B. They are already in place.

10.2 Group Anagrams: Write a method to sort an array of strings so that all the anagrams are next to each other.

pg 150

SOLUTION

This problem asks us to group the strings in an array such that the anagrams appear next to each other. Note that no specific ordering of the words is required, other than this.

We need a quick and easy way of determining if two strings are anagrams of each other. What defines if two words are anagrams of each other? Well, anagrams are words that have the same characters but in different orders. It follows then that if we can put the characters in the same order, we can easily check if the new words are identical.

One way to do this is to just apply any standard sorting algorithm, like merge sort or quick sort, and modify the comparator. This comparator will be used to indicate that two strings which are anagrams of each other are equivalent.

What's the easiest way of checking if two words are anagrams? We could count the occurrences of the distinct characters in each string and return `true` if they match. Or, we could just sort the string. After all, two words which are anagrams will look the same once they're sorted.

The code below implements the comparator.

```
1  class AnagramComparator implements Comparator<String> {
2    public String sortChars(String s) {
3      char[] content = s.toCharArray();
4      Arrays.sort(content);
5      return new String(content);
6    }
7
8    public int compare(String s1, String s2) {
9      return sortChars(s1).compareTo(sortChars(s2));
10   }
11 }
```

Now, just sort the arrays using this `compareTo` method instead of the usual one.

```
12 Arrays.sort(array, new AnagramComparator());
```

This algorithm will take `O(n log(n))` time.

This may be the best we can do for a general sorting algorithm, but we don't actually need to fully sort the array. We only need to *group* the strings in the array by anagram.

We can do this by using a hash table which maps from the sorted version of a word to a list of its anagrams. So, for example, `acre` will map to the list `{acre, race, care}`. Once we've grouped all the words into these lists by anagram, we can then put them back into the array.

The code below implements this algorithm.

```
1  void sort(String[] array) {
2    HashMapList<String, String> mapList = new HashMapList<String, String>();
3
4    /* Group words by anagram */
5    for (String s : array) {
6      String key = sortChars(s);
7      mapList.put(key, s);
8    }
```

```
9
10    /* Convert hash table to array */
11    int index = 0;
12    for (String key : mapList.keySet()) {
13       ArrayList<String> list = mapList.get(key);
14       for (String t : list) {
15          array[index] = t;
16          index++;
17       }
18    }
19 }
20
21 String sortChars(String s) {
22    char[] content = s.toCharArray();
23    Arrays.sort(content);
24    return new String(content);
25 }
26
27 /* HashMapList<String, Integer> is a HashMap that maps from Strings to
28  * ArrayList<Integer>. See appendix for implementation. */
```

You may notice that the algorithm above is a modification of bucket sort.

10.3 Search in Rotated Array: Given a sorted array of n integers that has been rotated an unknown number of times, write code to find an element in the array. You may assume that the array was originally sorted in increasing order.

EXAMPLE

Input: find 5 in {15, 16, 19, 20, 25, 1, 3, 4, 5, 7, 10, 14}

Output: 8 (the index of 5 in the array)

pg 150

SOLUTION

If this problem smells like binary search to you, you're right!

In classic binary search, we compare x with the midpoint to figure out if x belongs on the left or the right side. The complication here is that the array is rotated and may have an inflection point. Consider, for example, the following two arrays:

```
Array1: {10, 15, 20,  0,  5}
Array2: {50,  5, 20, 30, 40}
```

Note that both arrays have a midpoint of 20, but 5 appears on the left side of one and on the right side of the other. Therefore, comparing x with the midpoint is insufficient.

However, if we look a bit deeper, we can see that one half of the array must be ordered normally (in increasing order). We can therefore look at the normally ordered half to determine whether we should search the left or right half.

For example, if we are searching for 5 in `Array1`, we can look at the left element (10) and middle element (20). Since 10 < 20, the left half must be ordered normally. And, since 5 is not between those, we know that we must search the right half.

In Array2, we can see that since 50 > 20, the right half must be ordered normally. We turn to the middle (20) and right (40) element to check if 5 would fall between them. The value 5 would not; therefore, we search the left half.

The tricky condition is if the left and the middle are identical, as in the example array {2, 2, 2, 3, 4, 2}. In this case, we can check if the rightmost element is different. If it is, we can search just the right side. Otherwise, we have no choice but to search both halves.

```
1  int search(int a[], int left, int right, int x) {
2    int mid = (left + right) / 2;
3    if (x == a[mid]) { // Found element
4      return mid;
5    }
6    if (right < left) {
7      return -1;
8    }
9
10   /* Either the left or right half must be normally ordered. Find out which side
11    * is normally ordered, and then use the normally ordered half to figure out
12    * which side to search to find x. */
13   if (a[left] < a[mid]) { // Left is normally ordered.
14     if (x >= a[left] && x < a[mid]) {
15       return search(a, left, mid - 1, x); // Search left
16     } else {
17       return search(a, mid + 1, right, x); // Search right
18     }
19   } else if (a[mid] < a[left]) { // Right is normally ordered.
20     if (x > a[mid] && x <= a[right]) {
21       return search(a, mid + 1, right, x); // Search right
22     } else {
23       return search(a, left, mid - 1, x); // Search left
24     }
25   } else if (a[left] == a[mid]) { // Left or right half is all repeats
26     if (a[mid] != a[right]) { // If right is different, search it
27       return search(a, mid + 1, right, x); // search right
28     } else { // Else, we have to search both halves
29       int result = search(a, left, mid - 1, x); // Search left
30       if (result == -1) {
31         return search(a, mid + 1, right, x); // Search right
32       } else {
33         return result;
34       }
35     }
36   }
37   return -1;
38 }
```

This code will run in O(log n) if all the elements are unique. However, with many duplicates, the algorithm is actually O(n). This is because with many duplicates, we will often have to search both the left and right sides of the array (or subarrays).

Note that while this problem is not conceptually very complex, it is actually very difficult to implement flawlessly. Don't feel bad if you had trouble implementing it without a few bugs. Because of the ease of making off-by-one and other minor errors, you should make sure to test your code very thoroughly.

10.4 Sorted Search, No Size: You are given an array-like data structure `Listy` which lacks a size method. It does, however, have an `elementAt(i)` method that returns the element at index `i` in `O(1)` time. If `i` is beyond the bounds of the data structure, it returns `-1`. (For this reason, the data structure only supports positive integers.) Given a `Listy` which contains sorted, positive integers, find the index at which an element x occurs. If x occurs multiple times, you may return any index.

pg 150

SOLUTION

Our first thought here should be binary search. The problem is that binary search requires us knowing the length of the list, so that we can compare it to the midpoint. We don't have that here.

Could we compute the length? Yes!

We know that `elementAt` will return `-1` when `i` is too large. We can therefore just try bigger and bigger values until we exceed the size of the list.

But how much bigger? If we just went through the list linearly—1, then 2, then 3, then 4, and so on—we'd wind up with a linear time algorithm. We probably want something faster than this. Otherwise, why would the interviewer have specified the list is sorted?

It's better to back off exponentially. Try 1, then 2, then 4, then 8, then 16, and so on. This ensures that, if the list has length n, we'll find the length in at most `O(log n)` time.

> Why `O(log n)`? Imagine we start with pointer q at `q = 1`. At each iteration, this pointer q doubles, until q is bigger than the length n. How many times can q double in size before it's bigger than n? Or, in other words, for what value of k does $2^k = n$? This expression is equal when `k = log n`, as this is precisely what `log` means. Therefore, it will take `O(log n)` steps to find the length.

Once we find the length, we just perform a (mostly) normal binary search. I say "mostly" because we need to make one small tweak. If the mid point is -1, we need to treat this as a "too big" value and search left. This is on line 16 below.

There's one more little tweak. Recall that the way we figure out the length is by calling `elementAt` and comparing it to -1. If, in the process, the element is bigger than the value x (the one we're searching for), we'll jump over to the binary search part early.

```
1  int search(Listy list, int value) {
2    int index = 1;
3    while (list.elementAt(index) != -1 && list.elementAt(index) < value) {
4      index *= 2;
5    }
6    return binarySearch(list, value, index / 2, index);
7  }
8
9  int binarySearch(Listy list, int value, int low, int high) {
10   int mid;
11
12   while (low <= high) {
13     mid = (low + high) / 2;
14     int middle = list.elementAt(mid);
15     if (middle > value || middle == -1) {
16       high = mid - 1;
17     } else if (middle < value) {
```

```
18          low = mid + 1;
19        } else {
20          return mid;
21        }
22      }
23      return -1;
24  }
```

It turns out that not knowing the length didn't impact the runtime of the search algorithm. We find the length in `O(log n)` time and then do the search in `O(log n)` time. Our overall runtime is `O(log n)`, just as it would be in a normal array.

10.5 Sparse Search: Given a sorted array of strings that is interspersed with empty strings, write a method to find the location of a given string.

EXAMPLE

Input: `ball, {"at", "", "", "", "ball", "", "", "car", "", "", "dad", "", ""}`

Output: 4

pg 150

SOLUTION

If it weren't for the empty strings, we could simply use binary search. We would compare the string to be found, `str`, with the midpoint of the array, and go from there.

With empty strings interspersed, we can implement a simple modification of binary search. All we need to do is fix the comparison against `mid`, in case `mid` is an empty string. We simply move `mid` to the closest non-empty string.

The recursive code below to solve this problem can easily be modified to be iterative. We provide such an implementation in the code attachment.

```
1   int search(String[] strings, String str, int first, int last) {
2     if (first > last) return -1;
3     /* Move mid to the middle */
4     int mid = (last + first) / 2;
5
6     /* If mid is empty, find closest non-empty string. */
7     if (strings[mid].isEmpty()) {
8       int left = mid - 1;
9       int right = mid + 1;
10      while (true) {
11        if (left < first && right > last) {
12          return -1;
13        } else if (right <= last && !strings[right].isEmpty()) {
14          mid = right;
15          break;
16        } else if (left >= first && !strings[left].isEmpty()) {
17          mid = left;
18          break;
19        }
20        right++;
21        left--;
22      }
23    }
```

```
24
25    /* Check for string, and recurse if necessary */
26    if (str.equals(strings[mid])) { // Found it!
27       return mid;
28    } else if (strings[mid].compareTo(str) < 0) { // Search right
29       return search(strings, str, mid + 1, last);
30    } else { // Search left
31       return search(strings, str, first, mid - 1);
32    }
33 }
34
35 int search(String[] strings, String str) {
36    if (strings == null || str == null || str == "") {
37       return -1;
38    }
39    return search(strings, str, 0, strings.length - 1);
40 }
```

The worst-case runtime for this algorithm is O(n). In fact, it's impossible to have an algorithm for this problem that is better than O(n) in the worst case. After all, you could have an array of all empty strings except for one non-empty string. There is no "smart" way to find this non-empty string. In the worst case, you will need to look at every element in the array.

Careful consideration should be given to the situation when someone searches for the empty string. Should we find the location (which is an O(n) operation)? Or should we handle this as an error?

There's no correct answer here. This is an issue you should raise with your interviewer. Simply asking this question will demonstrate that you are a careful coder.

10.6 **Sort Big File:** Imagine you have a 20 GB file with one string per line. Explain how you would sort the file.

pg 150

SOLUTION

When an interviewer gives a size limit of 20 gigabytes, it should tell you something. In this case, it suggests that they don't want you to bring all the data into memory.

So what do we do? We only bring part of the data into memory.

We'll divide the file into chunks, which are x megabytes each, where x is the amount of memory we have available. Each chunk is sorted separately and then saved back to the file system.

Once all the chunks are sorted, we merge the chunks, one by one. At the end, we have a fully sorted file.

This algorithm is known as external sort.

10.7 Missing Int: Given an input file with four billion non-negative integers, provide an algorithm to generate an integer that is not contained in the file. Assume you have 1 GB of memory available for this task.

FOLLOW UP

What if you have only 10 MB of memory? Assume that all the values are distinct and we now have no more than one billion non-negative integers.

pg 150

SOLUTION

There are a total of 2^{32}, or 4 billion, distinct integers possible and 2^{31} non-negative integers. Therefore, we know the input file (assuming it is `int`s rather than `long`s) contains some duplicates.

We have 1 GB of memory, or 8 billion bits. Thus, with 8 billion bits, we can map all possible integers to a distinct bit with the available memory. The logic is as follows:

1. Create a bit vector (BV) with 4 billion bits. Recall that a bit vector is an array that compactly stores boolean values by using an array of ints (or another data type). Each int represents 32 boolean values.
2. Initialize BV with all 0s.
3. Scan all numbers (num) from the file and call `BV.set(num, 1)`.
4. Now scan again BV from the 0th index.
5. Return the first index which has a value of 0.

The following code demonstrates our algorithm.

```
1  long numberOfInts = ((long) Integer.MAX_VALUE) + 1;
2  byte[] bitfield = new byte [(int) (numberOfInts / 8)];
3  String filename = ...
4
5  void findOpenNumber() throws FileNotFoundException {
6    Scanner in = new Scanner(new FileReader(filename));
7    while (in.hasNextInt()) {
8      int n = in.nextInt ();
9      /* Finds the corresponding number in the bitfield by using the OR operator to
10      * set the nth bit of a byte (e.g., 10 would correspond to the 2nd bit of
11      * index 2 in the byte array). */
12     bitfield [n / 8] |= 1 << (n % 8);
13   }
14
15   for (int i = 0; i < bitfield.length; i++) {
16     for (int j = 0; j < 8; j++) {
17       /* Retrieves the individual bits of each byte. When 0 bit is found, print
18        * the corresponding value. */
19       if ((bitfield[i] & (1 << j)) == 0) {
20         System.out.println (i * 8 + j);
21         return;
22       }
23     }
24   }
25 }
```

Follow Up: What if we have only 10 MB memory?

It's possible to find a missing integer with two passes of the data set. We can divide up the integers into blocks of some size (we'll discuss how to decide on a size later). Let's just assume that we divide up the integers into blocks of 1000. So, block 0 represents the numbers 0 through 999, block 1 represents numbers 1000 - 1999, and so on.

Since all the values are distinct, we know how many values we *should* find in each block. So, we search through the file and count how many values are between 0 and 999, how many are between 1000 and 1999, and so on. If we count only 999 values in a particular range, then we know that a missing int must be in that range.

In the second pass, we'll actually look for which number in that range is missing. We use the bit vector approach from the first part of this problem. We can ignore any number outside of this specific range.

The question, now, is what is the appropriate block size? Let's define some variables as follows:

- Let `rangeSize` be the size of the ranges that each block in the first pass represents.
- Let `arraySize` represent the number of blocks in the first pass. Note that `arraySize` = $2^{31}/\texttt{rangeSize}$ since there are 2^{31} non-negative integers.

We need to select a value for `rangeSize` such that the memory from the first pass (the array) and the second pass (the bit vector) fit.

First Pass: The Array

The array in the first pass can fit in 10 megabytes, or roughly 2^{23} bytes, of memory. Since each element in the array is an int, and an int is 4 bytes, we can hold an array of at most about 2^{21} elements. So, we can deduce the following:

$$\texttt{arraySize} = \frac{2^{31}}{\texttt{rangeSize}} \leq 2^{21}$$
$$\texttt{rangeSize} \geq \frac{2^{31}}{2^{21}}$$
$$\texttt{rangeSize} \geq 2^{10}$$

Second Pass: The Bit Vector

We need to have enough space to store `rangeSize` bits. Since we can fit 2^{23} bytes in memory, we can fit 2^{26} bits in memory. Therefore, we can conclude the following:

$$2^{11} \texttt{ <= rangeSize <= } 2^{26}$$

These conditions give us a good amount of "wiggle room," but the nearer to the middle that we pick, the less memory will be used at any given time.

The below code provides one implementation for this algorithm.

```
1  int findOpenNumber(String filename) throws FileNotFoundException {
2    int rangeSize = (1 << 20); // 2^20 bits (2^17 bytes)
3
4    /* Get count of number of values within each block. */
5    int[] blocks = getCountPerBlock(filename, rangeSize);
6
7    /* Find a block with a missing value. */
8    int blockIndex = findBlockWithMissing(blocks, rangeSize);
9    if (blockIndex < 0) return -1;
```

```
10
11    /* Create bit vector for items within this range. */
12    byte[] bitVector = getBitVectorForRange(filename, blockIndex, rangeSize);
13
14    /* Find a zero in the bit vector */
15    int offset = findZero(bitVector);
16    if (offset < 0) return -1;
17
18    /* Compute missing value. */
19    return blockIndex * rangeSize + offset;
20  }
21
22  /* Get count of items within each range. */
23  int[] getCountPerBlock(String filename, int rangeSize)
24        throws FileNotFoundException {
25    int arraySize = Integer.MAX_VALUE / rangeSize + 1;
26    int[] blocks = new int[arraySize];
27
28    Scanner in = new Scanner (new FileReader(filename));
29    while (in.hasNextInt()) {
30        int value = in.nextInt();
31        blocks[value / rangeSize]++;
32    }
33    in.close();
34    return blocks;
35  }
36
37  /* Find a block whose count is low. */
38  int findBlockWithMissing(int[] blocks, int rangeSize) {
39    for (int i = 0; i < blocks.length; i++) {
40      if (blocks[i] < rangeSize){
41        return i;
42      }
43    }
44    return -1;
45  }
46
47  /* Create a bit vector for the values within a specific range. */
48  byte[] getBitVectorForRange(String filename, int blockIndex, int rangeSize)
49        throws FileNotFoundException {
50    int startRange = blockIndex * rangeSize;
51    int endRange = startRange + rangeSize;
52    byte[] bitVector = new byte[rangeSize/Byte.SIZE];
53
54    Scanner in = new Scanner(new FileReader(filename));
55    while (in.hasNextInt()) {
56      int value = in.nextInt();
57      /* If the number is inside the block that's missing numbers, we record it */
58      if (startRange <= value && value < endRange) {
59        int offset = value - startRange;
60        int mask = (1 << (offset % Byte.SIZE));
61        bitVector[offset / Byte.SIZE] |= mask;
62      }
63    }
64    in.close();
65    return bitVector;
```

```
66 }
67
68 /* Find bit index that is 0 within byte. */
69 int findZero(byte b) {
70   for (int i = 0; i < Byte.SIZE; i++) {
71     int mask = 1 << i;
72     if ((b & mask) == 0) {
73       return i;
74     }
75   }
76   return -1;
77 }
78
79 /* Find a zero within the bit vector and return the index. */
80 int findZero(byte[] bitVector) {
81   for (int i = 0; i < bitVector.length; i++) {
82     if (bitVector[i] != ~0) { // If not all 1s
83       int bitIndex = findZero(bitVector[i]);
84       return i * Byte.SIZE + bitIndex;
85     }
86   }
87   return -1;
88 }
```

What if, as a follow up question, you are asked to solve the problem with even less memory? In this case, we can do repeated passes using the approach from the first step. We'd first check to see how many integers are found within each sequence of a million elements. Then, in the second pass, we'd check how many integers are found in each sequence of a thousand elements. Finally, in the third pass, we'd apply the bit vector.

10.8 Find Duplicates: You have an array with all the numbers from 1 to N, where N is at most 32,000. The array may have duplicate entries and you do not know what N is. With only 4 kilobytes of memory available, how would you print all duplicate elements in the array?

pg 151

SOLUTION

We have 4 kilobytes of memory which means we can address up to $8 * 4 * 2^{10}$ bits. Note that $32 * 2^{10}$ bits is greater than 32000. We can create a bit vector with 32000 bits, where each bit represents one integer.

Using this bit vector, we can then iterate through the array, flagging each element v by setting bit v to 1. When we come across a duplicate element, we print it.

```
1  void checkDuplicates(int[] array) {
2    BitSet bs = new BitSet(32000);
3    for (int i = 0; i < array.length; i++) {
4      int num = array[i];
5      int num0 = num - 1; // bitset starts at 0, numbers start at 1
6      if (bs.get(num0)) {
7        System.out.println(num);
8      } else {
9        bs.set(num0);
10     }
11   }
12 }
13
14 class BitSet {
```

```
15    int[] bitset;
16
17    public BitSet(int size) {
18       bitset = new int[(size >> 5) + 1]; // divide by 32
19    }
20
21    boolean get(int pos) {
22       int wordNumber = (pos >> 5); // divide by 32
23       int bitNumber = (pos & 0x1F); // mod 32
24       return (bitset[wordNumber] & (1 << bitNumber)) != 0;
25    }
26
27    void set(int pos) {
28       int wordNumber = (pos >> 5); // divide by 32
29       int bitNumber = (pos & 0x1F); // mod 32
30       bitset[wordNumber] |= 1 << bitNumber;
31    }
32 }
```

Note that while this isn't an especially difficult problem, it's important to implement this cleanly. This is why we defined our own bit vector class to hold a large bit vector. If our interviewer lets us (she may or may not), we could have of course used Java's built in `BitSet` class.

10.9 Sorted Matrix Search: Given an `M x N` matrix in which each row and each column is sorted in ascending order, write a method to find an element.

pg 151

SOLUTION

We can approach this in two ways: a more naive solution that only takes advantage of part of the sorting, and a more optimal way that takes advantage of both parts of the sorting.

Solution #1: Naive Solution

As a first approach, we can do binary search on every row to find the element. This algorithm will be `O(M log(N))`, since there are `M` rows and it takes `O(log(N))` time to search each one. This is a good approach to mention to your interviewer before you proceed with generating a better algorithm.

To develop an algorithm, let's start with a simple example.

15	20	40	85
20	35	80	95
30	55	95	105
40	80	100	120

Suppose we are searching for the element 55. How can we identify where it is?

If we look at the start of a row or the start of a column, we can start to deduce the location. If the start of a column is greater than 55, we know that 55 can't be in that column, since the start of the column is always the minimum element. Additionally, we know that 55 can't be in any columns on the right, since the first element of each column must increase in size from left to right. Therefore, if the start of the column is greater than the element `x` that we are searching for, we know that we need to move further to the left.

For rows, we use identical logic. If the start of a row is bigger than `x`, we know we need to move upwards.

Observe that we can also make a similar conclusion by looking at the ends of columns or rows. If the end of a column or row is less than x, then we know that we must move down (for rows) or to the right (for columns) to find x. This is because the end is always the maximum element.

We can bring these observations together into a solution. The observations are the following:

- If the start of a column is greater than x, then x is to the left of the column.
- If the end of a column is less than x, then x is to the right of the column.
- If the start of a row is greater than x, then x is above that row.
- If the end of a row is less than x, then x is below that row.

We can begin in any number of places, but let's begin with looking at the starts of columns.

We need to start with the greatest column and work our way to the left. This means that our first element for comparison is `array[0][c-1]`, where c is the number of columns. By comparing the start of columns to x (which is 55), we'll find that x must be in columns 0, 1, or 2. We will have stopped at `array[0][2]`.

This element may not be the end of a row in the full matrix, but it is an end of a row of a submatrix. The same conditions apply. The value at `array[0][2]`, which is 40, is less than 55, so we know we can move downwards.

We now have a submatrix to consider that looks like the following (the gray squares have been eliminated).

15	20	40	85
20	35	80	95
30	55	95	105
40	80	100	120

We can repeatedly apply these conditions to search for 55. Note that the only conditions we actually use are conditions 1 and 4.

The code below implements this elimination algorithm.

```
1  boolean findElement(int[][] matrix, int elem) {
2    int row = 0;
3    int col = matrix[0].length - 1;
4    while (row < matrix.length && col >= 0) {
5      if (matrix[row][col] == elem) {
6        return true;
7      } else if (matrix[row][col] > elem) {
8        col--;
9      } else {
10       row++;
11     }
12   }
13   return false;
14 }
```

Alternatively, we can apply a solution that more directly looks like binary search. The code is considerably more complicated, but it applies many of the same learnings.

Solution #2: Binary Search

Let's again look at a simple example.

15	20	70	85
20	35	80	95
30	55	95	105
40	80	100	120

We want to be able to leverage the sorting property to more efficiently find an element. So, we might ask ourselves, what does the unique ordering property of this matrix imply about where an element might be located?

We are told that every row and column is sorted. This means that element `a[i][j]` will be greater than the elements in row `i` between columns 0 and `j - 1` and the elements in column `j` between rows 0 and `i - 1`.

Or, in other words:

```
a[i][0] <= a[i][1] <= ... <= a[i][j-1] <= a[i][j]
a[0][j] <= a[1][j] <= ... <= a[i-1][j] <= a[i][j]
```

Looking at this visually, the dark gray element below is bigger than all the light gray elements.

15	20	70	85
20	35	80	95
30	55	95	105
40	80	100	120

The light gray elements also have an ordering to them: each is bigger than the elements to the left of it, as well as the elements above it. So, by transitivity, the dark gray element is bigger than the entire square.

15	20	70	85
20	35	80	95
30	55	95	105
40	80	100	120

This means that for any rectangle we draw in the matrix, the bottom right hand corner will always be the biggest.

Likewise, the top left hand corner will always be the smallest. The colors below indicate what we know about the ordering of elements (light gray < dark gray < black):

15	20	70	85
20	35	80	95
30	55	95	105
40	80	120	120

Let's return to the original problem: suppose we were searching for the value 85. If we look along the diagonal, we'll find the elements 35 and 95. What does this tell us about the location of 85?

15	20	70	85
25	35	80	95
30	55	95	105
40	80	120	120

85 can't be in the black area, since 95 is in the upper left hand corner and is therefore the smallest element in that square.

85 can't be in the light gray area either, since 35 is in the lower right hand corner of that square.

85 must be in one of the two white areas.

So, we partition our grid into four quadrants and recursively search the lower left quadrant and the upper right quadrant. These, too, will get divided into quadrants and searched.

Observe that since the diagonal is sorted, we can efficiently search it using binary search.

The code below implements this algorithm.

```
1  Coordinate findElement(int[][] matrix, Coordinate origin, Coordinate dest, int x){
2    if (!origin.inbounds(matrix) || !dest.inbounds(matrix)) {
3      return null;
4    }
5    if (matrix[origin.row][origin.column] == x) {
6      return origin;
7    } else if (!origin.isBefore(dest)) {
8      return null;
9    }
10
11   /* Set start to start of diagonal and end to the end of the diagonal. Since the
12    * grid may not be square, the end of the diagonal may not equal dest. */
13   Coordinate start = (Coordinate) origin.clone();
14   int diagDist = Math.min(dest.row - origin.row, dest.column - origin.column);
15   Coordinate end = new Coordinate(start.row + diagDist, start.column + diagDist);
16   Coordinate p = new Coordinate(0, 0);
17
18   /* Do binary search on the diagonal, looking for the first element > x */
19   while (start.isBefore(end)) {
20     p.setToAverage(start, end);
21     if (x > matrix[p.row][p.column]) {
22       start.row = p.row + 1;
23       start.column = p.column + 1;
24     } else {
25       end.row = p.row - 1;
26       end.column = p.column - 1;
27     }
28   }
29
30   /* Split the grid into quadrants. Search the bottom left and the top right. */
31   return partitionAndSearch(matrix, origin, dest, start, x);
32 }
33
34 Coordinate partitionAndSearch(int[][] matrix, Coordinate origin, Coordinate dest,
35                               Coordinate pivot, int x) {
36   Coordinate lowerLeftOrigin = new Coordinate(pivot.row, origin.column);
37   Coordinate lowerLeftDest = new Coordinate(dest.row, pivot.column - 1);
38   Coordinate upperRightOrigin = new Coordinate(origin.row, pivot.column);
39   Coordinate upperRightDest = new Coordinate(pivot.row - 1, dest.column);
40
41   Coordinate lowerLeft = findElement(matrix, lowerLeftOrigin, lowerLeftDest, x);
42   if (lowerLeft == null) {
43     return findElement(matrix, upperRightOrigin, upperRightDest, x);
44   }
```

```
45    return lowerLeft;
46  }
47
48  Coordinate findElement(int[][] matrix, int x) {
49    Coordinate origin = new Coordinate(0, 0);
50    Coordinate dest = new Coordinate(matrix.length - 1, matrix[0].length - 1);
51    return findElement(matrix, origin, dest, x);
52  }
53
54  public class Coordinate implements Cloneable {
55    public int row, column;
56    public Coordinate(int r, int c) {
57      row = r;
58      column = c;
59    }
60
61    public boolean inbounds(int[][] matrix) {
62      return  row >= 0 && column >= 0 &&
63              row < matrix.length && column < matrix[0].length;
64    }
65
66    public boolean isBefore(Coordinate p) {
67      return row <= p.row && column <= p.column;
68    }
69
70    public Object clone() {
71      return new Coordinate(row, column);
72    }
73
74    public void setToAverage(Coordinate min, Coordinate max) {
75      row = (min.row + max.row) / 2;
76      column = (min.column + max.column) / 2;
77    }
78  }
```

If you read all this code and thought, "there's no way I could do all this in an interview!" you're probably right. You couldn't. But, your performance on any problem is evaluated compared to other candidates on the same problem. So while you couldn't implement all this, neither could they. You are at no disadvantage when you get a tricky problem like this.

You help yourself out a bit by separating code out into other methods. For example, by pulling `partitionAndSearch` out into its own method, you will have an easier time outlining key aspects of the code. You can then come back to fill in the body for `partitionAndSearch` if you have time.

10.10 Rank from Stream: Imagine you are reading in a stream of integers. Periodically, you wish to be able to look up the rank of a number x (the number of values less than or equal to x). Implement the data structures and algorithms to support these operations. That is, implement the method `track(int x)`, which is called when each number is generated, and the method `getRankOfNumber(int x)`, which returns the number of values less than or equal to x (not including x itself).

EXAMPLE

Stream (in order of appearance): `5, 1, 4, 4, 5, 9, 7, 13, 3`

`getRankOfNumber(1) = 0`

`getRankOfNumber(3) = 1`

`getRankOfNumber(4) = 3`

pg 151

SOLUTION

A relatively easy way to implement this would be to have an array that holds all the elements in sorted order. When a new element comes in, we would need to shift the other elements to make room. Implementing `getRankOfNumber` would be quite efficient, though. We would simply perform a binary search for n, and return the index.

However, this is very inefficient for inserting elements (that is, the `track(int x)` function). We need a data structure which is good at keeping relative ordering, as well as updating when we insert new elements. A binary search tree can do just that.

Instead of inserting elements into an array, we insert elements into a binary search tree. The method `track(int x)` will run in `O(log n)` time, where n is the size of the tree (provided, of course, that the tree is balanced).

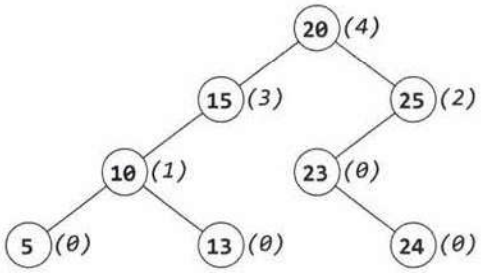
To find the rank of a number, we could do an in-order traversal, keeping a counter as we traverse. The goal is that, by the time we find x, `counter` will equal the number of elements less than x.

As long as we're moving left during searching for x, the counter won't change. Why? Because all the values we're skipping on the right side are greater than x. After all, the very smallest element (with rank of 1) is the leftmost node.

When we move to the right though, we skip over a bunch of elements on the left. All of these elements are less than x, so we'll need to increment `counter` by the number of elements in the left subtree.

Rather than counting the size of the left subtree (which would be inefficient), we can track this information as we add new elements to the tree.

Let's walk through an example on the following tree. In the below example, the value in parentheses indicates the number of nodes in the left subtree (or, in other words, the rank of the node *relative* to its subtree).



Suppose we want to find the rank of 24 in the tree above. We would compare 24 with the root, 20, and find that 24 must reside on the right. The root has 4 nodes in its left subtree, and when we include the root itself, this gives us five total nodes smaller than 24. We set `counter` to 5.

Then, we compare 24 with node 25 and find that 24 must be on the left. The value of `counter` does not update, since we're not "passing over" any smaller nodes. The value of `counter` is still 5.

Next, we compare 24 with node 23, and find that 24 must be on the right. `Counter` gets incremented by just 1 (to 6), since 23 has no left nodes.

Finally, we find 24 and we return `counter`: 6.

Recursively, the algorithm is the following:

```
int getRank(Node node, int x) {
  if x is node.data, return node.leftSize()
  if x is on left of node, return getRank(node.left, x)
  if x is on right of node, return node.leftSize() + 1 + getRank(node.right, x)
}
```

The full code for this is below.

```
RankNode root = null;

void track(int number) {
  if (root == null) {
    root = new RankNode(number);
  } else {
    root.insert(number);
  }
}

int getRankOfNumber(int number) {
  return root.getRank(number);
}


public class RankNode {
  public int left_size = 0;
  public RankNode left, right;
  public int data = 0;
  public RankNode(int d) {
    data = d;
  }

  public void insert(int d) {
    if (d <= data) {
```

```
26          if (left != null) left.insert(d);
27          else left = new RankNode(d);
28          left_size++;
29        } else {
30          if (right != null) right.insert(d);
31          else right = new RankNode(d);
32        }
33      }
34
35      public int getRank(int d) {
36        if (d == data) {
37          return left_size;
38        } else if (d < data) {
39          if (left == null) return -1;
40          else return left.getRank(d);
41        } else {
42          int right_rank = right == null ? -1 : right.getRank(d);
43          if (right_rank == -1) return -1;
44          else return left_size + 1 + right_rank;
45        }
46      }
47  }
```

The `track` method and the `getRankOfNumber` method will both operate in `O(log N)` on a balanced tree and `O(N)` on an unbalanced tree.

Note how we've handled the case in which d is not found in the tree. We check for the -1 return value, and, when we find it, return -1 up the tree. It is important that you handle cases like this.

10.11 Peaks and Valleys: In an array of integers, a "peak" is an element which is greater than or equal to the adjacent integers and a "valley" is an element which is less than or equal to the adjacent integers. For example, in the array {5, 8, 6, 2, 3, 4, 6}, {8, 6} are peaks and {5, 2} are valleys. Given an array of integers, sort the array into an alternating sequence of peaks and valleys.

EXAMPLE

Input: {5, 3, 1, 2, 3}

Output: {5, 1, 3, 2, 3}

pg 151

SOLUTION

Since this problem asks us to sort the array in a particular way, one thing we can try is doing a normal sort and then "fixing" the array into an alternating sequence of peaks and valleys.

Suboptimal Solution

Imagine we were given an unsorted array and then sort it to become the following:

```
0  1  4  7  8  9
```

We now have an ascending list of integers.

How can we rearrange this into a proper alternating sequence of peaks and valleys? Let's walk through it and try to do that.

- The `0` is okay.

- The 1 is in the wrong place. We can swap it with either the 0 or 4. Let's swap it with the 0.
  ```
  1  0  4  7  8  9
  ```
- The 4 is okay.
- The 7 is in the wrong place. We can swap it with either the 4 or the 8. Let's swap it with the 4.
  ```
  1  0  7  4  8  9
  ```
- The 9 is in the wrong place. Let's swap it with the 8.
  ```
  1  0  7  4  9  8
  ```

Observe that there's nothing special about the array having these values. The relative order of the elements matters, but all sorted arrays will have the same relative order. Therefore, we can take this same approach on any sorted array.

Before coding, we should clarify the exact algorithm, though.

1. Sort the array in ascending order.
2. Iterate through the elements, starting from index 1 (not 0) and jumping two elements at a time.
3. At each element, swap it with the previous element. Since every three elements appear in the order `small <= medium <= large`, swapping these elements will always put medium as a peak: `medium <= small <= large`.

This approach will ensure that the peaks are in the right place: indexes 1, 3, 5, and so on. As long as the odd-numbered elements (the peaks) are bigger than the adjacent elements, then the even-numbered elements (the valleys) must be smaller than the adjacent elements.

The code to implement this is below.

```
1  void sortValleyPeak(int[] array) {
2    Arrays.sort(array);
3    for (int i = 1; i < array.length; i += 2) {
4      swap(array, i - 1, i);
5    }
6  }
7
8  void swap(int[] array, int left, int right) {
9    int temp = array[left];
10   array[left] = array[right];
11   array[right] = temp;
12 }
```

This algorithm runs in `O(n log n)` time.

Optimal Solution

To optimize past the prior solution, we need to cut out the sorting step. The algorithm must operate on an unsorted array.

Let's revisit an example.

```
9  1  0  4  8  7
```

For each element, we'll look at the adjacent elements. Let's imagine some sequences. We'll just use the numbers 0, 1 and 2. The specific values don't matter.

```
0  1  2
0  2  1       // peak
1  0  2
1  2  0       // peak
2  1  0
```

```
2   0   1
```

If the center element needs to be a peak, then two of those sequences work. Can we fix the other ones to make the center element a peak?

Yes. We can fix this sequence by swapping the center element with the largest adjacent element.

```
0   1   2   -> 0   2   1
0   2   1       // peak
1   0   2   -> 1   2   0
1   2   0   // peak
2   1   0   -> 1   2   0
2   0   1   -> 0   2   1
```

As we noted before, if we make sure the peaks are in the right place then we know the valleys are in the right place.

> We should be a little cautious here. Is it possible that one of these swaps could "break" an earlier part of the sequence that we'd already processed? This is a good thing to worry about, but it's not an issue here. If we're swapping `middle` with `left`, then `left` is currently a valley. `Middle` is smaller than `left`, so we're putting an even smaller element as a valley. Nothing will break. All is good!

The code to implement this is below.

```
1  void sortValleyPeak(int[] array) {
2    for (int i = 1; i < array.length; i += 2) {
3      int biggestIndex = maxIndex(array, i - 1, i, i + 1);
4      if (i != biggestIndex) {
5        swap(array, i, biggestIndex);
6      }
7    }
8  }
9
10 int maxIndex(int[] array, int a, int b, int c) {
11   int len = array.length;
12   int aValue = a >= 0 && a < len ? array[a] : Integer.MIN_VALUE;
13   int bValue = b >= 0 && b < len ? array[b] : Integer.MIN_VALUE;
14   int cValue = c >= 0 && c < len ? array[c] : Integer.MIN_VALUE;
15
16   int max = Math.max(aValue, Math.max(bValue, cValue));
17   if (aValue == max) return a;
18   else if (bValue == max) return b;
19   else return c;
20 }
```

This algorithm takes O(n) time.