Homework: Git Bisect

Most software projects reside in a Git repository. One of the more useful Git commands is `bisect`. The following is an example from its manual page:

```
$ git bisect start
$ git bisect bad                 # Current version is bad
$ git bisect good abc1234        # abc1234 is known to be good
```

The current version has some bug (is `bad`); version `abc1234` does not (it is `good`). The goal is to find the commit that introduced the bug. Git will checkout some version between `abc1234` and the current one and then let you figure out if the bug is present. If the bug is present, you should run the following command:

```
git bisect bad
```

If the bug is not present, you should run the following command:

```
git bisect good
```

After this, Git checks out another version, you inform Git if the version is bad or good, and this process repeats several times. Eventually, Git reports what is the commit that introduced the bug. Now you have a good hint about where in the code you should look. Or, at least, you know who to blame.

Git tries to figure out where the bug was introduced by asking few questions. After all, figuring out if the bug is present or not may be time consuming: at the very least, it involves compiling the whole project. Git-bisect uses a heuristic that works on arbitrary histories, and degenerates to binary search when used on linear histories. Your task is to try to do better.

A Small Example

You will write a program that communicates with a server: your program plays the role of `git-bisect`, the server plays the role of a human. The communication works over WebSockets, with messages in JSON format. We use WebSockets because, unlike sockets, they have a notion of message; that is, communication is not a stream of bytes, but a stream of messages. We use JSON because it is a widely used standard for representing structured information in a human-readable format, which makes it easy to debug.

Let us consider an example. Immediately after connecting, the client authenticates by sending the following message:

```
{"User": ["rg399", "TOKEN"] }
```

Above, `TOKEN` should be replaced by the access token you received by email.

Next, the server replies with a repository description (Repo), followed by a problem instance (Instance):

```
{"Repo":{"name":"pb0","instance_count":10,"dag":[["a",[]],["b",["a"]],["c",["b"]]]}}
{"Instance":{"good":"a","bad":"c"}}
```

The dag (directed acyclic graph) lists the parents of each commit:

- `a` has no parent
- `b` has parent `a`
- `c` has parent `b`

This is a linear history, because each commit but one has exactly one parent (i.e., there is no branching). In general, one can have *merge* commits, which have multiple parents. Cycles, however, will never be present.

The Repo message is followed by an Instance message, which tells us that `a` is good and `c` is bad. So, the bug could have been introduced by `b` or `c`; but, which one?

The client now asks, and the server responds:

```
{"Question":"b"}
{"Answer":"Good"}
```

And again:

```
{"Question":"c"}
{"Answer":"Bad"}
```

At this point, the client figured out that the bug was introduced in version `c`, and communicates its solution to the server:

```
{"Solution":"c"}
```

The server now continues in one of three ways:

- With a new Repo message.
- With a new Instance message. (Each Instance message refers to the last Repo message. The Repo message gives the dag; the Instance message gives the known good/bad commits.)
- With a Scores message, if the submission was successful.

In this example, there is no further problem instance to solve, so the server replies with:

```
{"Score":{"pb0":{"Correct":2}}}
```

This message says that the solution was correct, and it was found with 2 questions. The other possible outcomes are the following:

```
{"Score":{"pb0":"Wrong"}}
{"Score":{"pb0":"GaveUp"}}
```

The last outcome is achieved by sending a `"GiveUp"` message instead of a question or a solution.

Observe that, from the problem instance, we already know that `c` is bad, so the second question asked by the client is redundant: this problem instance can be solved with just one question!

Miscellany

Limits.

- At deadline, the server will hang up.
- The client may ask *at most 30 questions*; otherwise, the server hangs up and the submission does not count. To avoid this, the client should send a `"GiveUp"` message.
- The size of the dag (the number of strings occurring at the right of "dag") will be at most 1 million.

Deadlines, and Simple Solution. There is more information on Moodle. In particular, you can find there what are the deadlines. And you can download a simple Java solution and you can download a simple Java solution. The simple solution will ask a question for each commit, but give up if the problem instance has more than 30 commits.

Clarifications. For clarifications, we recommend using Piazza. You can also just ask us.

Marking

This programming project is worth 30% of your final mark. Because of this, the scoring rules here describe how 30 points are awarded. (However, because of how SDS works, marks will appear there out of 100 points, then they will be weighted to give a coursework mark, they weighted again for the final mark, and after all that weighting the score gets back to something out of 30, except that small rounding errors are introduced in the process. Sigh.)

Rules:

- 21 points are given for correctness; 9 points are given for efficiency.
- You may assume that your submission was successful only if you receive the **Scores** message from the server. In particular, if you go over the question limit, or if the websocket connection times out, then the submission was *not* successful and does not count.
- If a solution is wrong, then not only the score for that problem is zero, but efficiency points are also zero. To avoid this situation, you should `"GiveUp"` when you cannot determine the right answer.
- The number of points you get for correctness will be proportional to the number of problem instances you solve correctly; for example, if there will be 1000 problem instances and you solve correctly 537 of them, then you get $537/1000 \times 21 = 11.277$ points.
- You may get efficiency points only if you do not give any wrong answer. Let r_i be your raw score for problem instance i; we set $r_i = 30$ if you gave up on problem instance i. The estimated raw score is $\hat{r} = \frac{1}{n}\sum_{i=1}^{n} r_i$; that is, the arithmetic mean of the raw scores. A raw score of 30 gives you 0 efficiency points; if you have the lowest raw score of all (last) submissions you get 9 efficiency points. Anything in-between is linearly interpolated.

Test Data

There are two servers: the test server and the submission server. Only what you submit to the submission server counts towards your mark.

On each connection, you receive *the same repositories* but *different problem instances*. Instances are generated randomly, as described below. The test server has 100 instances, which means that scores may fluctuate. The submission server has 10000 instances, which means that scores will be quite stable, but it also means that a real submission will take 100 times the time of a test submission. You must remember that you need to be able to finish your real submission by the deadline.

The test server has a scoreboard visible by everyone. The submission server does not.

The test data is generated as follows:

1. For each size range out of $(0, 30]$, $(30, 10^4)$, $[10^4, 10^5)$, $[10^5, 10^6)$, several git repositories are chosen. For the small range $(0, 30]$, most repositories are synthetic; for the other ranges, all repositories are repositories of real software projects.
2. From each repository, problem instances are generated, as follows:
   - A non-root commit b is chosen uniformly at random as *the (hidden) bug*.
   - A descendant of b is chosen uniformly at random as *the known bad commit*.
   - A non-descendant of b is chosen uniformly at random as *the known good commit*.

In total, the submission server has

- 5000 tests with the number of commits in the range $(0, 30]$,
- 2200 tests with the number of commits in the range $(30, 10^4)$,
- 1400 tests with the number of commits in the range $[10^4, 10^5)$, and
- 1400 tests with the number of commits in the range $[10^5, 10^6)$.

The test server has 100 times fewer tests, in each size category.

References

1. Git Bisect Internals
2. Git Objects (commits, trees).

Changelog

- **2020-02-13**
  - Drop the requirement that each server message is answered quickly.
  - Added the possibility to give up on a problem.
  - Updated score split to 21 for correctness plus 9 for efficiency.
  - Described how test data is generated.
- **2020-02-14**: The sample solution *is* correct because it does give up.
- **2020-02-26**: Added access tokens.
- **2020-03-02**: Split Problem messages into Repo/Instance messages. Reduced maximum questions to 30. Updated scoring rules to depend on average number of questions. Tests are different on each connection.