Effective Bisection and Bibisection

Matthew Francis
Overview

- Introduction to bisection and bibisection
- How to bibisect effectively
  - Preparation
  - Execution
  - Results
- Questions
What is (bi)bisection?

A quick introduction
What is bisection?

- Bisection is a way to identify which commit introduced a bug without testing every single commit
- Using the “git bisect” command
- A number of commits must still be tested
- Each must be built from source
- Binary bisection (“bibisection”) is a way to do the same thing faster
- We provide you with pre-built binaries, making it unnecessary to build from source each time
What is bisection?

- How our git repository is structured

![Diagram showing git repository structure with master branch, release branches for versions x.y.0 and x.y.1, and branch point.](image-url)
What is bisection?

- How many commits are made in a typical release?
  - 4.1: 10064
  - 4.2: 12147
  - 4.3: 14208
  - 4.4: 10932
  - 5.0: 9732 (10010 including merged branches)
What is bisection?

How does bisection work?

Starting with a known good and known bad commit, we perform a binary search for the first bad commit.
What is bisection?

- How does bisection work?
- Each time, we test the commit in the middle of the latest known good and earliest known bad

![Diagram of bisection process]

Good   ?   Bad

Master
What is bisection?

- How does bisection work?
- Approximately half the commits in the remaining range can be eliminated on each test
What is bisection?

- How does bisection work?
  - This allows us to avoid testing most commits

This illustrates why git tells you “roughly N steps” are remaining – depending on whether this commit tests “good” or “bad”, there may be an extra step.
What is bisection?

- How does bisection work?
- After just a few tests, a single commit is identified
What is bibisection?

- Bibisection makes bisection faster
  - We provide you with a special git repository which contains pre-built binaries
  - No more need to spend all day compiling
  - A bibisect repository is at least several gigabytes in size
    - But you only have to download it once
    - Bugs can be bisected in minutes rather than hours
What is bibisection?

- The original kind of bibisect repository was sparse
  - Sparse bibisect repositories do not cover every source commit
    - Some contain 1 build for every N source commits, for instance 1 in 64 source commits
    - Some contain 1 build per day regardless of how many actual source commits occurred
  - Most sparse repositories are now obsolete, but a few are still useful
    - The “43all” repository covers some old periods of development which are not currently contained in any other repository
    - The “daily dbgutil” repository provides daily builds of current master
What is bibisection?

- Sparse bibisect → One bibisect every several source commits
What is bibisection?

- Most recently built bibisect repositories are fine-grained
  - Fine-grained bibisect attempts to cover every source commit
    - Some commits don't compile; these are still excluded
  - Fine-grained repositories for recent master epochs are built from clean each time
    - The Linux 44max and 50max repositories have been built this way
    - This gives the greatest confidence that the result of a bibisect identifies the right commit
What is bibisection?

- Fine-grained bibisect
  - One bibisect commit per source commit
What is bibisection?

- Some fine-grained repositories for older epochs have been built incrementally
- The Linux 41max, 42max and 43max repositories have been built this way
- Each epoch takes about 1/4 the time to build that it would do from clean (about one week of build time rather than one month for each epoch)
- This has allowed some remaining older regressions to be tracked down cheaply
- The results from these repositories should be treated with care, as build system dependency errors may cause them to be imprecise. Double-check that the commit pointed to is plausible
What is bibisection?

Bibisect coverage (as of September 2015)
What is bibisection?

- What doesn't bibisect cover?
  - Very old commits between LibreOffice 3.3 and 3.5
  - Source commits on the release branches (with the exception of the “releases” repository)
  - Almost all regressions are first introduced on master
  - Commits on release branches should consist only of bugfixes which have been backported from an existing commit on master
  - A few bugs have occurred on a release branch alone, but this is very rare
- Source commits on the release branches may be covered in future
What is bibisection?

- Which of these bibisect repositories should I use?
  - Use the fine-grained repositories if possible
    - Linux 41max, 42max, 43max, 44max, 50max
  - If a bug is before or after the range covered by the above, use a suitable sparse repository
    - Linux 43all, daily dbgutil
  - There are other historical repositories, but ignore them
  - Don't use the “releases” repository for bibisection
    - Except to show that a bug predates LibreOffice
How to bibisect effectively
How to bisect effectively

What does “effective” mean?

- Identifying either a single commit or a small range by a single person
- Forwarding the bug to that person
  - Ideally while it's still recent enough for them to know and care about
- If a single commit can't be identified, documenting what you find about when a bug was introduced will still save other QA and developers from wasting time repeating what you did
Before starting:

- Make sure you have solid reproduction steps, and can reproduce on some version
- If it's not clear enough from the bug report, comment the exact steps you've worked out
- If you can't reproduce at all, send the bug back for more information (NEEDINFO)
- Simplify the reproduction steps as much as possible – you will have to perform them several times
- If you have trouble reproducing, try near to the version the original reporter used
Before starting:

- Make sure the bug is still present on master
- Use the Linux daily dbgutil repo to check
  - Beware that occasionally the behaviour of dbgutil builds can be different
- If the bug was present before, but appears to have been fixed on master, consider performing a reverse bibisect to find the commit that fixed it
- Sometimes bugs are fixed tangentially. It's good to check that a fix has been applied to all currently supported releases
More on simplifying reproduction steps:

- If the original steps consist of doing several things to a document in sequence, try to prepare a test document at the point just before the bug occurs – and attach it to the bug if possible.

- If reproducing the bug requires running a macro, insert a form button which executes it so you don't have to hunt through the menus/dialogs for the macro each time.

- If the bug is about loading / saving in different file formats (e.g. loss of formatting), use command line conversion:
  ```
  opt/program/soffice --headless --convert-to docx something.odt
  ```
Setting up the bisect:
- Select a commit that works and one that doesn't
  - Do actually check these
- A single master epoch is usually a good range
  - i.e. a single fine-grained repo such as 44max or 50max
- The same or a similar bug can be introduced several times, but bisection will only reveal one instance
- Picking a good range helps ensure you find the latest instance of the bug, which will be the most useful to know
While you're bisectioning:

- Take care to follow the reproduction steps you worked out.
- Sometimes you may find commits which can't be used at all, or behave in some other way than before the bug or on current master.
- Start by assuming that there are only a few of these, and use “git bisect skip” to skip over them.
- If many commits must be skipped, split the work into two tasks and start again.
- Pick any commit with the third behaviour (“skip”) between the “good” and “bad” commits you started with.
- First bisect between “good” and “skip”.
- Then bisect between “skip” and “bad”.
- Comment both results on the bug.
Dealing with the results

- After finishing a binary search:
  - Re-check the result!
  - It's very easy to bisect "good" when you meant "bad" and vice versa
  - Occasionally you may find that a bug wasn't as reliably reproducible as you thought

  → Assuming a single commit has been identified in a fine-grained repository, checkout the "first bad" commit and retest that, then from that position checkout HEAD~1 and retest that
Dealing with the results

What sorts of result are possible?

- Fine-grained bibisect, single commit
  → The best sort! We can send it straight to the author

- Fine-grained bibisect, range of skipped commits
  Or when a source range couldn't be built
  → Maybe still OK – if all by one person. Otherwise equivalent to...

- Sparse bibisect, single commit
  Still a range of source commits really
  → Someone will still need to look at this more later

- Sparse bibisect, range of skipped commits
  → Oh dear – extended build failure, the hardest to diagnose
Dealing with the results

If the result points at a single commit or several by one person:

- Adjust the bug fields
  - Remove Whiteboard: bibisectRequest
  - Check whether the bug is a true regression, or was introduced together with the feature it relates to
    - If it's a true regression, set the bug fields to include:
      - Whiteboard: bibisected
      - Keywords: bisected, regression
    - If the bug arrived together with its feature:
      - Whiteboard: bibisected implementationError
      - Keywords: bisected
Dealing with the results

- If the result points at a single commit or several by one person:
  - Add a comment which identifies the source commit(s) in question
  - Cc: the original author on the bug
    - Note that “author” and “committer” may not be the same
    - Don't Cc: the bibisect repo builder – listed as the “author” of the commit in the bibisect repo
      → Look at the “author” and “committer” in the body of the commit message
- Some commits are ported from OpenOffice. Don't try to contact the author in this case (typically addresses @apache.org)
- Just for this case, Cc: the committer instead
- Not everybody uses the same email address in Bugzilla as they do to commit with. If you type the author's name in the Cc: box, Bugzilla will show you a list which should contain the correct address
- A few occasional contributors don't have accounts on Bugzilla. If you can't find a suitable email address, just note this in your comment
Dealing with the results

- If the result points at a single commit or several by one person:
  - Provided that the result has been double-checked, and the commit(s) identified could plausibly have introduced the bug, there's no reason to include the output of “git bisect log”
  - It adds visual noise to the bug, and doesn't add much useful information to the result
What if the results are incomplete?

- If a single commit couldn't be identified:
  - I.e. the result is a set of skipped commits that aren't by a single person, or
  - There was an extended build breakage, or
  - A sparse bibisect repository was used
- Replace Whiteboard: bibisectRequest with Whiteboard: bibisected
- Don't add Keywords: bisected
- In this case, it's reasonable to append the “git bisect log” output in a comment
- Don't Cc: anybody
  - Unless you can read the corresponding source commits / commit logs and identify a very likely commit
Reverse bisection

- Sometimes it's useful to know the commit that fixed a bug
- E.g. to check that it's been applied to the branches of all currently supported releases
- To find this, you have to bisect in reverse
  - “git bisect” doesn't like to search in reverse
    - Make the initial “good” commit of the search one that contains the bug
    - Make the initial “bad” commit one in which it has been fixed
- As you bisect, remember that “git bisect good” means “this commit contains the bug”, and “git bisect bad” means “the bug was fixed in this commit”
- Repeat to yourself, “Fair is foul and foul is fair”
Git hints and tips

- Using the 43all repository
  - Some commits contain files they shouldn't (parts of the user profile)
  - “git bisect good / bad” may refuse to continue because there are files in the way
  - To fix this, from within the 43all directory, either:
    - Manually delete the files it complains about (slow, safe)
    - Or, run “git clean -dffx” (fast, hazardous)
  - Caution: this will delete all untracked files in the directory and reset all others to their state in the repository
Git hints and tips

- Use “git bisect visualize”
  - Shows information about the range of commits still to be tested
  - Useful to find the bounds of a range of skipped commits if bisection ended that way
  - “git bisect log” unhelpfully lists the skipped commits in random order
  - Needs to have “gitk” installed for graphical display
Git hints and tips
Moving forward in history

There's no simple way to move forward in history from a specific commit in git, but you can use the following:

For the source repository and the daily dbgutil bibisect repo:

```
git checkout `git rev-list --topo-order HEAD..master | tail -1`
```

For other regular bibisect repositories:

```
git checkout `git rev-list --topo-order HEAD..latest | tail -1`
```

Useful when you want to quickly scrub backward / forward to re-check a result

Make them into scripts or shell aliases

I call mine “gitforward” and “bbforward”
Further information

- [https://wiki.documentfoundation.org/QA/Bibisect](https://wiki.documentfoundation.org/QA/Bibisect)
- Freenode IRC channel #libreoffice-qa
Questions?
Thank you