NAME

   git-merge-base - Find as good common ancestors as possible for a merge

SYNOPSIS

   git merge-base [-a | --all] <commit> <commit>...
   git merge-base [-a | --all] --octopus <commit>...
   git merge-base --is-ancestor <commit> <commit>
   git merge-base --independent <commit>...
   git merge-base --fork-point <ref> [<commit>]

DESCRIPTION

   git merge-base finds best common ancestor(s) between two commits to use in a three-way merge. One
   common ancestor is *better* than another common ancestor if the latter is an ancestor of the former. A
   common ancestor that does not have any better common ancestor is a *best common ancestor*, i.e. a
   merge base. Note that there can be more than one merge base for a pair of commits.

OPERATION MODES

   As the most common special case, specifying only two commits on the command line means
   computing the merge base between the given two commits.

   More generally, among the two commits to compute the merge base from, one is specified by the first
   commit argument on the command line; the other commit is a (possibly hypothetical) commit that is a
   merge across all the remaining commits on the command line.

   As a consequence, the *merge base* is not necessarily contained in each of the commit arguments if more
   than two commits are specified. This is different from **git-show-branch**(1) when used with the
   --merge-base option.

   --octopus
       Compute the best common ancestors of all supplied commits, in preparation for an n-way merge.
       This mimics the behavior of *git show-branch --merge-base*.

   --independent
       Instead of printing merge bases, print a minimal subset of the supplied commits with the same
       ancestors. In other words, among the commits given, list those which cannot be reached from any
       other. This mimics the behavior of *git show-branch --independent*.

   --is-ancestor
       Check if the first <commit> is an ancestor of the second <commit>, and exit with status 0 if true,

or with status 1 if not. Errors are signaled by a non-zero status that is not 1.

--fork-point
    Find the point at which a branch (or any history that leads to <commit>) forked from another branch (or any reference) <ref>. This does not just look for the common ancestor of the two commits, but also takes into account the reflog of <ref> to see if the history leading to <commit> forked from an earlier incarnation of the branch <ref> (see discussion on this mode below).

OPTIONS
-a, --all
    Output all merge bases for the commits, instead of just one.

DISCUSSION
Given two commits *A* and *B*, **git merge-base A B** will output a commit which is reachable from both *A* and *B* through the parent relationship.

For example, with this topology:

```
    o---o---o---B
   /
---o---1---o---o---o---A
```

the merge base between *A* and *B* is *1*.

Given three commits *A*, *B* and *C*, **git merge-base A B C** will compute the merge base between *A* and a hypothetical commit *M*, which is a merge between *B* and *C*. For example, with this topology:

```
    o---o---o---o---C
   /
  /   o---o---o---B
 /   /
---2---1---o---o---o---A
```

the result of **git merge-base A B C** is *1*. This is because the equivalent topology with a merge commit *M* between *B* and *C* is:

```
    o---o---o---o---o
   /           \
  /   o---o---o---o---M
 /   /
```

```
---2---1---o---o---o---A
```

and the result of **git merge-base A M** is *1*. Commit *2* is also a common ancestor between *A* and *M*, but *1* is a better common ancestor, because *2* is an ancestor of *1*. Hence, *2* is not a merge base.

The result of **git merge-base --octopus A B C** is *2*, because *2* is the best common ancestor of all commits.

When the history involves criss-cross merges, there can be more than one *best* common ancestor for two commits. For example, with this topology:

```
---1---o---A
   \ /
    X
   / \
---2---o---o---B
```

both *1* and *2* are merge-bases of A and B. Neither one is better than the other (both are *best* merge bases). When the **--all** option is not given, it is unspecified which best one is output.

A common idiom to check "fast-forward-ness" between two commits A and B is (or at least used to be) to compute the merge base between A and B, and check if it is the same as A, in which case, A is an ancestor of B. You will see this idiom used often in older scripts.

```
A=$(git rev-parse --verify A)
if test "$A" = "$(git merge-base A B)"
then
        ... A is an ancestor of B ...
fi
```

In modern git, you can say this in a more direct way:

```
if git merge-base --is-ancestor A B
then
        ... A is an ancestor of B ...
fi
```

instead.

DISCUSSION ON FORK-POINT MODE

After working on the **topic** branch created with **git switch -c topic origin/master**, the history of remote-tracking branch **origin/master** may have been rewound and rebuilt, leading to a history of this shape:

```
          o---B2
         /
   ---o---o---B1--o---o---o---B (origin/master)
        \
         B0
          \
           D0---D1---D (topic)
```
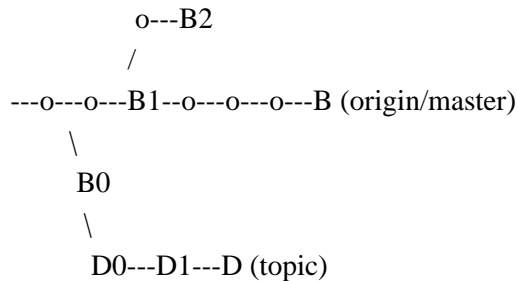
where **origin/master** used to point at commits B0, B1, B2 and now it points at B, and your **topic** branch was started on top of it back when **origin/master** was at B0, and you built three commits, D0, D1, and D, on top of it. Imagine that you now want to rebase the work you did on the topic on top of the updated origin/master.

In such a case, **git merge-base origin/master topic** would return the parent of B0 in the above picture, but B0^..D is **not** the range of commits you would want to replay on top of B (it includes B0, which is not what you wrote; it is a commit the other side discarded when it moved its tip from B0 to B1).

git merge-base --fork-point origin/master topic is designed to help in such a case. It takes not only B but also B0, B1, and B2 (i.e. old tips of the remote-tracking branches your repository's reflog knows about) into account to see on which commit your topic branch was built and finds B0, allowing you to replay only the commits on your topic, excluding the commits the other side later discarded.
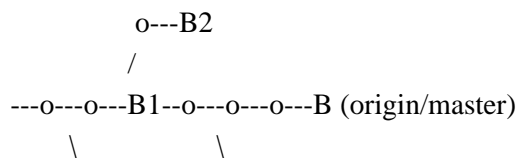
Hence

```
    $ fork_point=$(git merge-base --fork-point origin/master topic)
```

will find B0, and

```
    $ git rebase --onto origin/master $fork_point topic
```

will replay D0, D1 and D on top of B to create a new history of this shape:

```
          o---B2
         /
   ---o---o---B1--o---o---o---B (origin/master)
        \           \
```

```
    B0              D0'--D1'--D' (topic - updated)
     \
      D0---D1---D (topic - old)
```

A caveat is that older reflog entries in your repository may be expired by **git gc**. If B0 no longer appears in the reflog of the remote-tracking branch **origin/master**, the **--fork-point** mode obviously cannot find it and fails, avoiding to give a random and useless result (such as the parent of B0, like the same command without the **--fork-point** option gives).

Also, the remote-tracking branch you use the **--fork-point** mode with must be the one your topic forked from its tip. If you forked from an older commit than the tip, this mode would not find the fork point (imagine in the above sample history B0 did not exist, origin/master started at B1, moved to B2 and then B, and you forked your topic at origin/master^ when origin/master was B1; the shape of the history would be the same as above, without B0, and the parent of B1 is what **git merge-base origin/master topic** correctly finds, but the **--fork-point** mode will not, because it is not one of the commits that used to be at the tip of origin/master).

SEE ALSO
git-rev-list(1), **git-show-branch**(1), **git-merge**(1)

GIT
Part of the **git**(1) suite