

# Семинар #3: git, продвинутый. Практика.

## Как сдавать задачи

Для сдачи ДЗ вам нужно создать репозиторий на GitLab (если он ещё не создан) под названием **devtools-homework**. Структура репозитория должна иметь вид:

```
├── seminar3_advanced_git/
│   ├── 01.sh
│   ├── 02.sh
│   ├── 03.sh
│   └── ...
└── ...
```

Для каждой задачи, если не сказано иное, нужно создать 1 файл решения с расширением **.sh**. Для каждой подзадачи нужно прописать все команды, которые исполняются в ходе выполнения этой подзадачи. Оформлять файл решения нужно в следующем формате:

```
# Subtask a
git merge feature
# Конфликт, разрешаю вручную
git add имена файлов
git merge --continue
# Открылся редактор, ввожу:
# This is my merge commit!

# Subtask b
git reset --hard HEAD~2
# Открываю reflog и ищу нужный коммит
git reset --hard 24e1f51
...
```

Если в задаче встречается вопрос или требование, то на это нужно ответить в комментариях (#) файла решения.

## Задача 1. Диапазоны

### Подготовка: генерация репозитория

Для решения этой задачи вам понадобится сгенерировать репозиторий с помощью скрипта **create\_two\_branch\_repo.sh**, который, как и другие скрипты этого задания, можно найти в папке: **seminar3\_advanced\_git/practice/scripts**. Для этого нужно сделать следующие шаги:

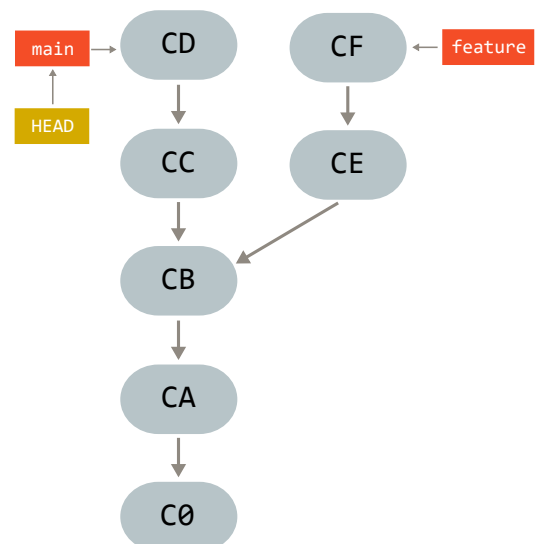
1. Скачайте скрипт к себе на компьютер
2. Сделайте скрипт исполняемым:  

```
$ chmod +x ./create_two_branch_repo.sh
```
3. Запустите его:  

```
$ ./create_two_branch_repo.sh test
```
4. В результате сгенерируется директория **test** с репозиторием. Зайдите в эту директорию и выполните:  

```
$ git log --oneline --all --graph
```

чтобы убедиться, что граф репозитория совпадает с тем, что изображено на рисунке.



## Подзадачи

**Важно!** В файле решения, в комментариях, выпишите результат выполнения всех команд этой задачи. Это важно, так как хэши коммитов в вашем репозитории могут отличаться (так как репозиторий генерируется скриптом), и я не смогу проверить задание, не зная хешей коммитов в вашем репозитории.

Также, все подзадачи этой задачи нужно выполнить без явного перечисления все коммитов, используя диапазоны. В реальных проектах в одной ветке могут быть тысячи коммитов и перечислить их всех не получится.

### a. Хэши всех коммитов

Выполните команду, которая будет печатать для каждого коммита в репозитории его сокращённый хэш и его сообщение в следующем формате.

```
Hash: 5b58166, message: "CF: add fox in animals.txt"
Hash: 5838367, message: "CD: add dog in animals.txt"
Hash: b25757f, message: "CE: add emu in animals.txt"
Hash: 7e58924, message: "CC: add cat in animals.txt"
Hash: 6f928eb, message: "CB: add bat in animals.txt"
Hash: a353350, message: "CA: add axolotl in animals.txt"
Hash: eca0dc3, message: "CO: initial commit"
```

### b. Диапазон коммитов, достижимых от одной ветки

Выполните команду, которая будет печатать хэши всех коммитов, достижимых от ветки `feature`.

### c. Разность множеств коммитов двух веток

Пусть  $S_{main}$  – это множество всех коммитов, достижимых от ветки `main`, а  $S_{feature}$  – это множество всех коммитов, достижимых от ветки `feature`. Выполните команду, которая будет печатать хэши всех коммитов, принадлежащих  $S_{feature} \setminus S_{main}$ . То есть нужно напечатать хэши всех коммитов, достижимых из ветки `feature`, но не достижимых из ветки `main`. Решите эту подзадачу двумя способами:

- (a) Используя синтаксис с двумя точками
- (b) Используя синтаксис исключения коммитов с символом  $\wedge$

### d. Симметрическая разность множеств коммитов двух веток

Выполните команду, которая будет печатать хэши всех коммитов, принадлежащих симметрической разности двух множеств:  $S_{main} \triangle S_{feature}$ .

### e. Последние коммиты

Используйте диапазон, чтобы напечатать хэши последних трёх коммитов в ветке `main`.

### f. Общий предок двух веток

Выполните команду, которая будет печатать хэш ближайшего общего предка веток `main` и `feature`.

### g. Разница между двумя коммитами

Выведите разницу между коммитами `main` и `main~`, используя команду:

```
$ git diff main main~
```

Объясните, что означает каждая строка данного вывода.

### h. Разница между ветками

Выведите разницу между ветками `main` и `feature`. Обратите внимание, что синтаксис перечисления коммитов через две/три точки имеет другое значение в команде `git diff` по сравнению со всеми другими командами `git`. Решите эту задачу двумя способами:

- (a) Передав `git diff` имена веток через аргументы.
- (b) Используя синтаксис `git diff A..B` (две точки).

### i. Разница между веткой и общим предком

Выведите разницу между веткой `feature` и общим предком этой ветки с веткой `main`. Решите эту задачу двумя способами:

- (a) Используя команду `git merge-base`.
- (b) Используя синтаксис `git diff A...B` (три точки).

## Задача 2. Типы слияния

### а. Трёхстороннее слияние (слияние по умолчанию)

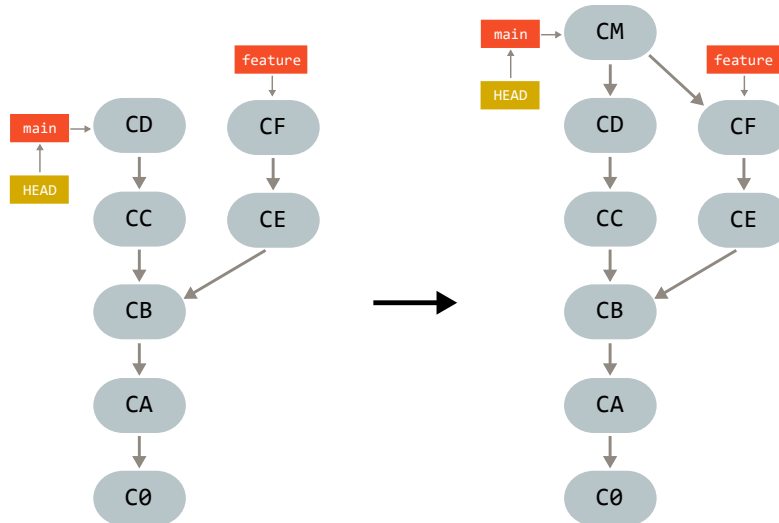
Самый распространённый тип слияния, при котором git ищет общего предка двух веток и смотрит на:

(A) Общий предок (`merge-base`)

(B) Текущая ветка (`HEAD`)

(C) Сливаемая ветка (`feature`)

Git попытается объединить изменения из B и C относительно общего предка A. Если строки изменены в обеих ветках, возникает конфликт, который нужно разрешить вручную.



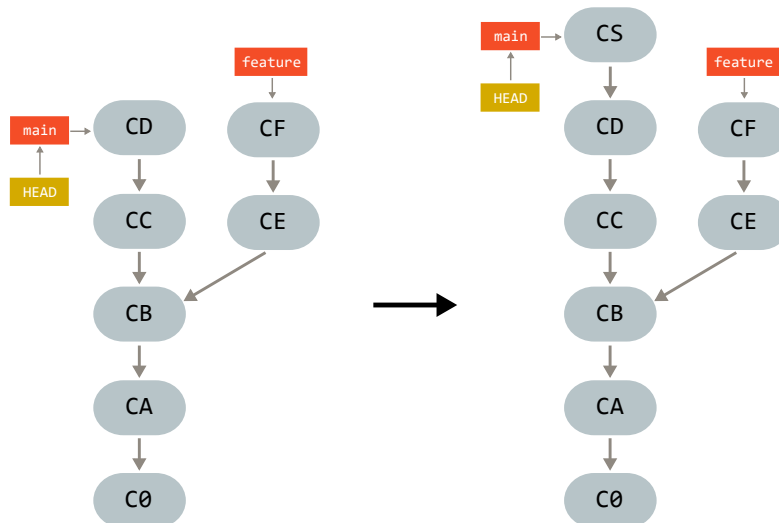
Сгенерируйте новый репозиторий с помощью скрипта `create_two_branch_repo.sh` и сделайте слияние ветки `feature` в ветку `main`, используя трёхстороннее слияние. Что будет содержаться в файле `animals.txt` до разрешения конфликта? Разрешите конфликт и создайте коммит слияния `CM`.

### б. Трёхстороннее слияние с опциями `ours` и `theirs`

Снова создайте репозиторий через `create_two_branch_repo.sh`. Сделайте слияние `feature` в `main`, используя трёхстороннее слияние с опцией `ours`. Снова сгенерируйте новый репозиторий и сделайте слияние, используя опцию `theirs`. Что будет содержаться в `animals.txt` после каждого из слияний?

### в. Squash-слияние (хотя технически это не слияние)

Squash-слияние – это способ объединить все изменения другой ветки в один коммит текущей ветки.



Сгенерируйте новый репозиторий с помощью скрипта `create_two_branch_repo.sh` и сделайте squash-слияние ветки `feature` в ветку `main`.

d. Сливаемая ветка позади текущей

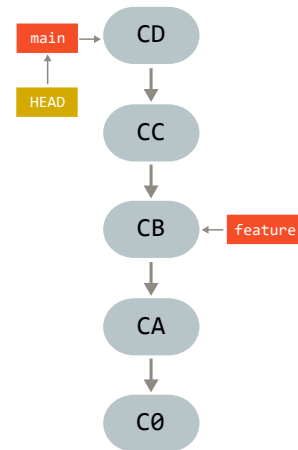
Предположим, что ветка `feature` находится позади ветки `main` и мы пытаемся сделать слияние `feature` в `main`:

```
$ git switch main
$ git merge feature
```

Ответ на следующие вопросы:

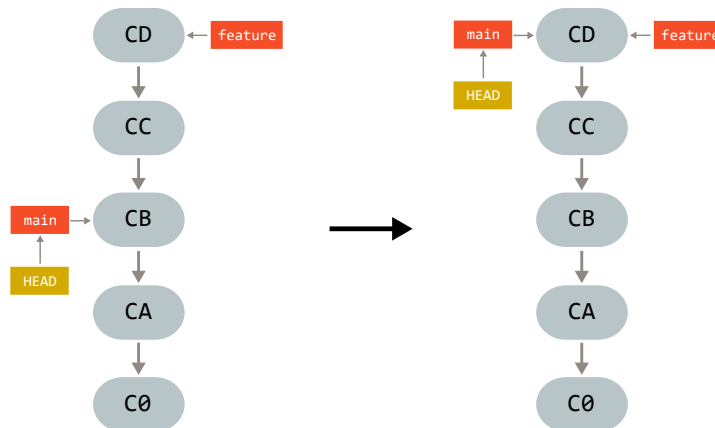
- (a) Будет ли создан новый коммит (коммит слияния) при такой операции?
- (b) Куда будет указывать ветка `main`?
- (c) Куда будет указывать ветка `feature`?

Сгенерировать репозиторий можно с помощью скрипта `create_feature_behind_main.sh`, который можно найти в `seminar3_advanced_git/practice/scripts`.



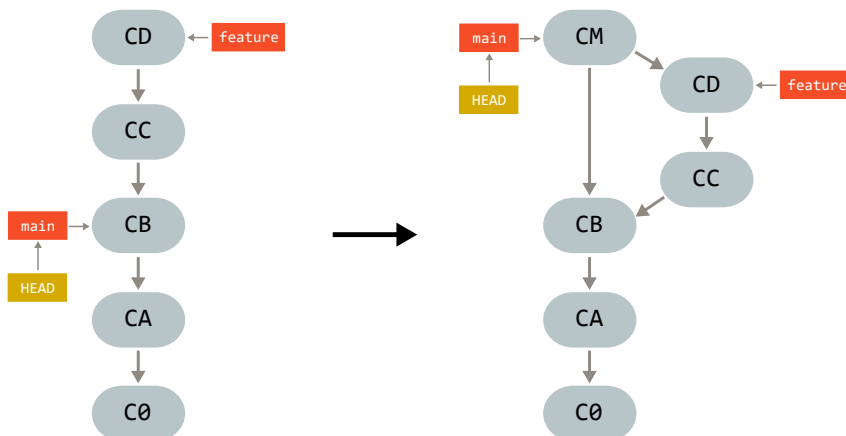
e. Сливаемая ветка впереди текущей: слияние перемоткой

Предположим, что ветка `feature` находится впереди ветки `main` и мы хотим сделать слияние перемоткой ветки `feature` в ветку `main`:



Сгенерируйте репозиторий с помощью скрипта `create_feature_ahead_main.sh` и произведите слияние перемоткой ветки `feature` в ветку `main`.

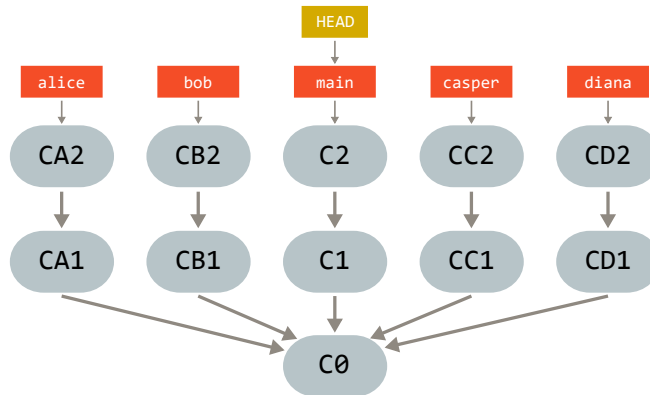
f. Сливаемая ветка впереди текущей: слияние без перемотки



Снова сгенерируйте репозиторий с помощью скрипта `create_feature_ahead_main.sh` и произведите обычное слияние без перемотки ветки `feature` в ветку `main`.

### g. Octopus-слияние

Git поддерживает слияние сразу нескольких веток одной операцией, но только с условием, что при этом не произойдёт ни одного конфликта. Это так называемое octopus-слияние.



Сгенерируйте репозиторий с помощью скрипта `create_octopus.sh` и произведите слияние веток `alice`, `bob`, `casper` и `diana` в ветку `main`. Распечатайте хэши всех родителей получившегося коммита слияния, используя  $\wedge$ -нотацию и команду `git rev-parse`.

## Задача 3. Работа с репозиторием

### Подготовка. Генерация репозитория, который будет использоваться в подзадачах

Репозиторий для этой задачи можно найти по адресу: `mipt-hsse.gitlab.yandexcloud.net/v.biryukov/utils`. Клонировать этот репозиторий себе:

```
git clone git@mipt-hsse.gitlab.yandexcloud.net:v.biryukov/utils.git
cd utils
```

### Подзадачи

#### a. Просмотр содержимого репозитория

Перейдите на все ветки и посмотрите, какие файлы содержатся в их последних коммитах:

```
$ git switch alice
$ git switch bob
$ git switch main
```

Это важно сделать ещё и потому, что при клонировании репозитория создаются только `remote-tracking` ветки (например, `origin/alice`), тогда как обычные локальные ветки (например, `alice`) отсутствуют. Их можно создать вручную или же они будут созданы автоматически при переключении на них – при условии, что соответствующая `remote-tracking` ветка существует.

#### b. Просмотрите граф коммитов

Это можно сделать на GitLab во вкладке `Code -> Repository Graph` или у вас на машине, с помощью:

```
$ git log --oneline --all --graph
```

Для вывода на экран, если вывод не помещается на экран полностью, команда `git log` используется программа `less`. Чтобы показать следующую строчку используйте `Enter`. Чтобы выйти из программы `less` просто нажмите клавишу `Q`.

#### c. Новый alias

Добавьте новый `alias` для этой команды, назовите его `lg`. То есть теперь, при вызове:

```
$ git lg
```

Должно печататься то же, что и при вызове

```
$ git log --oneline --all --graph
```

d. **Повторное использование записанного разрешения конфликта**

В процессе работы с большими репозиториями могут возникнуть ситуации, когда один и тот же конфликт приходится разрешать множество раз. Git может самостоятельно разрешать повторяющиеся конфликты, если включить специальную опцию **rerere** (*reuse recorded resolution*). Включите эту опцию локально для данного репозитория.

e. **Слияние с конфликтом**

Слейте ветку **bob** в ветку **main**. Для этого вам нужно перейти в ветку **main** и вызвать:

```
$ git merge bob
```

При этом должен возникнуть конфликт. Разрешите его. Если у вас включена настройка **rerere**, то Git запомнит то, как вы разрешили этот конфликт. В этой и последующих задачах необязательно, чтобы код после слияния был корректным, так как это задание на git, а не на язык программирования. Посмотрите как выглядит граф коммитов после слияния с помощью **git lg**.

f. **Отмена слияния и повторное разрешение конфликта**

Отмените только что выполненное слияние. Затем произведите слияние заново. На этот раз конфликт должен разрешиться автоматически. Вам нужно будет только проверить результат и создать коммит слияния.

g. **Отмена слияния**

Отмените только что выполненное слияние, используя **ORIG\_HEAD** или **HEAD@{1}**.

h. **Перебазирование с конфликтом**

Перебазировать ветку **bob** на ветку **main**. Для этого вам нужно перейти в ветку **bob** и вызвать:

```
$ git rebase main
```

При этом могут возникнуть конфликты. Разрешите их. Посмотрите как выглядит граф коммитов после перебазирования с помощью **git lg**.

i. **Отмена перебазирования**

Отмените только что выполненное перебазирование, используя **ORIG\_HEAD** или **HEAD@{1}**.

j. **Копия диапазона коммитов с конфликтом**

Скопируйте из ветки **alice** на ветку **bob** диапазон коммитов от **4d6662c** до **fe2e890**, используя одну команду **git cherry-pick**. При этом могут возникнуть несколько конфликтов. Разрешите их. Посмотрите как выглядит граф коммитов с помощью **git lg**.

k. **Просмотр файлов**

Напишите команды, которые печатают файлы на экран:

- файл **arithmetic.py** из ветки **main**
- файл **arithmetic.py** из коммита **alice~5**

l. **Разница**

Напишите команды, которые печатают на экран разницу (**diff**) между следующими коммитами или отдельными файлами:

- между коммитами **alice** и **alice~5**
- между файлом **integration.py** в ветке **main** и файлом **integration.py** ветки **alice**.
- между файлом **misc.py** коммита **fe2e890** и файлом **arithmetic.py** коммита **765df07**.

m. **Поиск первого коммита, содержащего ошибку**

Перейдите в ветку **alice** и запустите скрипт **sorting.py**

```
python ./sorting.py
```

Вы увидите, что одна из сортировок работает неправильно, хотя в других ветках эта сортировка работала правильно. Значит в одном из коммитов ветки **alice** была допущена ошибка. Найдите коммит, в котором была допущена ошибка с помощью **git bisect**. Укажите в файле решения хэш коммита, файл и строку в которой впервые возникает ошибка.

#### n. Фрагменты

Перейдите на ветку `alice` и добавьте изменения. В рабочей папке добавьте комментарии к функциям `add`, `factorial`, `is_prime` и `is_perfect_number` из файла `arithmetic.py`. Комментарии должны выглядеть примерно так:

```
# This function adds two numbers
def add(a: float, b: float) -> float:
    return a + b
```

Добавьте в индекс только изменения, связанные с функциями `add` и `is_prime`. Остальные изменения добавлять не нужно. Создавать новый коммит не нужно. Используйте команду `git add` с опцией `-p`.

#### o. Хранилище

В реальной работе с репозиторием часто возникает следующая ситуация: вы работаете в какой-либо ветке, изменяете файлы, добавляете их в индекс, но пока не готовы делать коммит. В этот момент может возникнуть необходимость переключиться на другую ветку, например, чтобы срочно исправить баг. Если в рабочей директории или в индексе есть изменения, которые могут быть перезаписаны при переключении на другую ветку, то Git не позволит выполнить `git switch`. Для решения этой проблемы можно использовать команду `git stash`, которая временно сохраняет изменения. После этого можно переключиться на другую ветку, выполнить нужную работу, а затем вернуться и восстановить сохранённые изменения.

После изменений, сделанных в прошлой подзадаче, попробуйте переключиться на ветку `main`. Произойдёт ошибка, так как есть несохранённые изменения в рабочей директории и в индексе. Используйте `git stash`, чтобы спрятать изменения. Перейдите на ветку `main` и сделайте там любой коммит. Вернитесь на ветку `alice` и восстановите изменения из `stash`. Восстановить изменения нужно таким образом, чтобы те изменения, которые были в индексе, остались в индексе, а те изменения, которые были в рабочей папке, остались в рабочей папке.

#### p. Перенос хранилища

Снова добавьте эти же изменения в `stash`. Перейдите на ветку `bob` и вытащите изменения, сделанные на ветке `alice`, в ветку `bob`. Возможен конфликт. Исправьте конфликт и сделайте коммит с этими изменениями в ветке `bob`.

#### q. Новая рабочая директория

Ещё один способ решения проблемы переключения на другую ветку при незакоммиченных изменениях – это использование нескольких рабочих директорий, которые можно создать с помощью `git worktree`.

Перейдите в ветку `alice` и внесите изменения в рабочую директорию и индекс. После этого вы не сможете переключиться на ветку `bob`, используя `git switch`. Создайте новую рабочую директорию (`worktree`), соответствующую ветке `bob`. Лучше располагать её вне текущей рабочей директории, чтобы она не мешала работе над текущей веткой. Перейдите в новую рабочую директорию. Обратите внимание, что `.git` в этой директории является обычным файлом, а не директорией. Что содержится в этом файле? Сделайте любой коммит в ветке `bob` и вернитесь обратно в первоначальную рабочую директорию.

#### r. Сборка мусора

В этой задаче нужно полностью удалить коммит из локального репозитория. Git обычно пытается обереечь пользователя от удаления коммитов. Даже если вы выполните `git reset --hard` и некоторые коммиты станут недостижимы через ветки, они на самом деле не удалятся и будут доступны ещё какое-то время (по умолчанию 1–2 месяца). На них можно будет перейти, если вы знаете их хэши. Удалить же коммит полностью можно, используя команду для сборки мусора `git gc`.

Перейдите на ветку `bob` и создайте там любой коммит. Запомните хэш этого коммита. После этого полностью удалите этот коммит из локального репозитория. Убедиться, что коммит на самом деле удалён, можно если попытаться переключиться на него: `git reset --hard` хэш. Если коммита не существует, то переключиться не получится.

#### s. Фильтрация репозитория

`git filter-repo` это не команда самого git, а сторонняя программа. Для решения этой подзадачи необходимо установить её. Перед выполнением этой подзадачи на всякий случай скопируйте весь репозиторий.

Напишите команды, используя `git filter-repo`, которые делают следующее:

- переименовывают файл `arithmetic.py` в файл `ar.py` во всех коммитах репозитория
- удаляют файл `sorting.py` во всех коммитах репозитория
- добавляют строку `"#COPYRIGHT"` в начало каждого `.py` файла каждого коммита репозитория

## Задача 4. Проблема CRLF и настройка autocrlf

### Краткая теория

При работе с текстовыми файлами в разных операционных системах может возникнуть проблема, связанная с использованием различных символов перевода строки.

- В операционных системах семейства Unix (Linux, macOS и другие) для перехода на новую строку используется один байт со значением 10 ( $A$  в шестнадцатеричной системе). Этот символ исторически называется LF (*Line Feed*).
- В операционных системах семейства Windows для перехода на новую строку используется последовательность из двух байт со значениями 13 ( $D_{16}$ ) и 10 ( $A_{16}$ ). Исторически байт со значением 13 носит название CR (*Carriage Return*).

Таким образом, если вы, например, откроете текстовый редактор и запишите туда:

```
aaa  
bbb  
ccc
```

А затем просмотрите байты этого файла, то, если вы делали это в Linux, вы увидите:

```
$ xxd a.txt  
00000000: 6161 610a 6262 620a 6363 630a          aaa.bbb.ccc.
```

Если же вы делали это в Windows, то вы увидите:

```
$ xxd a.txt  
00000000: 6161 610d 0a62 6262 0d0a 6363 630d 0a    aaa..bbb..ccc..
```

Эта проблема может проявиться, если разработчики работают с одним репозиторием на разных операционных системах. Например, если в репозитории весь код использует LF окончания строк, а один из разработчиков использует Windows и клонировал репозиторий, сделал одно маленькое изменение в файле, то на самом деле в файле изменится каждая строка, так как в конце каждой строки добавится дополнительный символ CR. Чтобы бороться с этой проблемой в Git можно использовать настройку `core.autocrlf`.

### Задача

Предположим, у нас есть три разработчика: Алиса, Боб и Чарли. Мы хотим, чтобы в текстовых файлах использовались следующие окончания:

- В репозиториях, как удалённом, так и локальных, должны использоваться LF-окончания.
- Алиса использует Linux. Когда она создаёт файлы, они всегда имеют окончания LF. Алиса хочет, чтобы при извлечении файлов из репозитория все файлы имели LF-окончания.
- Боб использует Windows. Когда он создаёт файлы, они имеют окончания CRLF. Боб хочет, чтобы при извлечении файлов из репозитория их окончания автоматически конвертировались из LF в CRLF. При добавлении файлов в индекс/репозиторий нужно чтобы окончания всех файлов конвертировались из CRLF в LF.
- Чарли использует Windows. Но он использует текстовый редактор, который создаёт файлы с окончаниями LF. Однако иногда он может использовать другой редактор и создать файлы с окончаниями CRLF. Чарли хочет, чтобы при извлечении файлов из репозитория их окончания не конвертировались. Но при добавлении файлов в индекс/репозиторий нужно, чтобы окончания всех файлов при необходимости конвертировались из CRLF в LF.

Какие настройки `core.autocrlf` должен использовать каждый из разработчиков, чтобы Git производил преобразования окончаний строк подобным образом? Напишите команды, которые устанавливают эти настройки.



## Задача 5. Файл .gitattributes

### Краткая теория

Использование `core.autocrlf` для решения проблемы CRLF имеет ряд недостатков:

- **Некоторые файлы должны всегда иметь CRLF-окончания строк**  
В частности, файлы с расширениями `.bat`, `.cmd` и `.ps1` являются скриптами оболочек Windows и должны всегда иметь CRLF-окончания, иначе эти скрипты могут выдать ошибку при запуске. Даже если эти файлы находятся в Linux, они в теории могут быть скопированы на Windows в обход Git.
- **Бинарные файлы не должны изменяться при добавлении или извлечении из репозитория**  
При включённой настройке `autocrlf` Git при добавлении файла сканирует его и заменяет все пары байт CRLF на один байт LF. Такую операцию Git должен производить только с текстовыми файлами, но не с бинарными. Если Git изменит байты бинарном файле, то он повредит этот файл.  
Как определить, какой файл является текстовым, а какой бинарным? В общем случае по расширению это сделать нельзя, так как любое расширение может использоваться как для текстового, так и для бинарного файла. Поэтому Git анализирует содержимое файла и применяет эвристики, например ищет байты, которые обычно не встречаются в текстовых файлах. Обычно Git достаточно точно определяет бинарность файла, но не абсолютно точно. На самом деле на 100% определить, является ли файл текстовым или бинарным, невозможно.
- **Необходимо настраивать настройку `autocrlf` отдельно для каждого разработчика**  
Если кто-то из разработчиков забудет настроить `autocrlf`, то он может случайно добавить в множество файлов с неправильными окончаниями.

Более надёжный способ решения проблемы — использовать файл `.gitattributes`, где можно явно указать, какие файлы должны иметь CRLF, а какие LF. В файле `.gitattributes` можно указать не только, какие окончания строк использовать для конкретных файлов, но и задать ряд других атрибутов. В частности можно указать, являются ли те или иные файлы текстовыми или бинарными, что используется не только при преобразовании окончаний строк, но и при нахождении разницы между файлами (текстовые файлы сравниваются построчно, для бинарных файлов просто отображается факт различия) или при слиянии (текстовые сливаются построчно, бинарные ни сливаются).

### Задача

Предположим, что вы разрабатываете большой проект на языке Java, в котором в дополнение к языку Java используются язык Python, а также скрипты оболочек разных операционных систем. Создайте файл `.gitattributes`, который бы устанавливал следующие атрибуты:

- Файлы с расширениями `.java`, `.gradle`, `.py`, `.sh`, `.xml`, `.json`, `.yaml` и `.txt` должны распознаваться как текстовые. При добавлении таких файлов в репозиторий, если в них обнаружатся окончания CRLF, они должны преобразовываться в окончания LF. При извлечении таких файлов из репозитория, окончания строк должны оставаться в формате LF.
- Файлы с расширениями `.bat`, `.cmd` и `.ps1` должны распознаваться как текстовые. При добавлении таких файлов в репозиторий, если в них обнаружатся окончания CRLF, они должны преобразовываться в окончания LF. При извлечении таких файлов из репозитория, окончания строк должны преобразовываться в формат CRLF.
- Файлы с расширениями `.class`, `.jar`, `.pyc`, `.png`, `.jpg` должны распознаваться как бинарные.
- Все остальные файлы должны распознаваться автоматически с помощью эвристик Git.

## Задача 6. Хуки

### а. Проверка расширений

Напишите `pre-commit` хук, который будет проверять, что все файлы коммита имеют расширение `.py`, используя язык Bash. Поместите скрипт в `.git/hooks/pre-commit` и проверьте работу этого хука, попытавшись закоммитить файл с другим расширением.

b. **Проверка размера файлов**

Напишите pre-commit хук, который будет проверять, что все файлы коммита имеют размер меньше, чем 1 мегабайт. Проверьте работу этого хука, создав коммит, попытавшись создать коммит, содержащий файлы большого размера.

c. **Проверка форматирования сообщения коммита**

Напишите commit-msg, который бы проверял, что сообщение коммита соответствует стандарту оформления Conventional Commits.

## Задача 7. Низкоуровневые команды Git

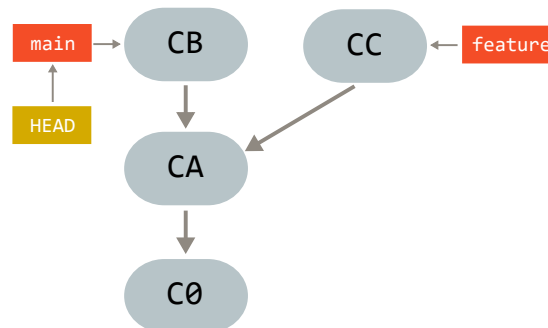
В этом задании можно использовать только низкоуровневые команды Git, такие как:

- `git init`
- `git hash-object`
- `git cat-file`
- `git ls-files`
- `git mktree`
- `git commit-tree`
- `git update-ref`
- `git ls-tree`
- `git rev-list`

А также команды для проверки результатов:

- `git status`
- `git log`
- `git show`
- `git diff`

Создайте репозиторий, граф которого изображён на рисунке, используя только эти команды.



При этом каждый коммит должен добавлять в репозиторий хотя бы один непустой, отличный от других файл. А коммит CA должен добавлять новую директорию и новый непустой файл в этой директории.

## Задача 8. Внутреннее устройство Git

Ответьте на следующие вопросы о внутреннем устройстве Git репозитория.

- a. Что такое объект в Git? Какие типы объектов существуют в Git?
- b. Что хранится в директории `.git`? Для чего нужны следующие файлы этой директории:

- `config`
- `HEAD`
- `ORIG_HEAD`
- `index`

Что хранится в следующих директориях внутри папки `.git`:

- `objects`
- `refs`
- `hooks`
- `info`

- c. При клонировании репозитория копируется весь репозиторий или только последнее состояние?

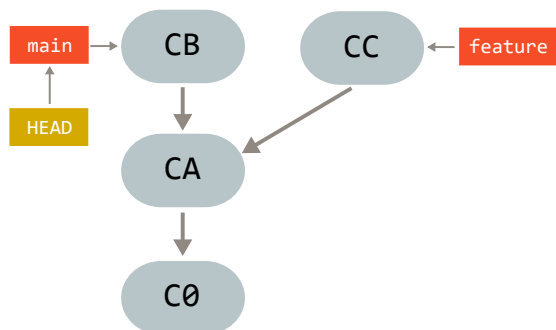
- d. Если в двух разных коммитах содержится одинаковый файл, то содержимое этого файла хранится дважды в репозитории?
- e. Если в проекте хранятся две копии одного и того же файла, то в репозитории этого проекта содержимое этого файла хранится дважды или в единственном экземпляре?
- f. Представьте, что в одном коммите мы добавили большой файл, а в следующем коммите изменили в нём всего один байт. Будет ли Git хранить обе версии файла целиком и дважды занимать место, или он умеет сохранять только различия между версиями, чтобы избежать дублирования?



## Необязательные задачи (не будут учитываться при оценивании)

### Задача 9. Ещё более низкоуровневый git

Создайте репозиторий, граф которого изображён на рисунке, вообще без использования команд Git.



Можно только использовать команды для проверки результатов:

- `git status`
- `git log`
- `git show`
- `git diff`