

Laboratórios de Informática I

2022/2023

Licenciatura em Engenharia Informática

Ficha 2

Sistemas de Controlo de Versões

O desenvolvimento de *software* é cada vez mais complexo, e obriga a que uma equipa de programadores possa desenvolver uma mesma aplicação ao mesmo tempo, sem se preocuparem com os detalhes do que outros membros dessa mesma equipa estejam a fazer. Alterações concorrentes (realizadas por diferentes pessoas ao mesmo tempo) podem provocar conflitos quando várias pessoas editam o mesmo bocado de código.

Além disso, não nos devemos esquecer que algumas alterações a um programa, no sentido de corrigir ou introduzir alguma funcionalidade, podem elas mesmas conter erros, e pode por isso ser necessário repor uma versão prévia da aplicação, anterior a essa alteração.

Para colmatar estes problemas são usados sistemas de controlo de versões.

1 Panorama nos Sistemas de Controlo de Versões

Existe um grande conjunto de sistemas que permitem o desenvolvimento cooperativo de *software*. Todos eles apresentam diferentes funcionalidades mas os seus principais objetivos são exatamente os mesmos.

Habitualmente divide-se este conjunto em dois, um conjunto de sistemas denominados de *centralizados*, e um outro de sistemas *distribuídos*:

- Sistemas de controlo de versões centralizados:
 - Concurrent Versions System (CVS): <http://www.nongnu.org/cvs/>;
 - Subversion (SVN): <https://subversion.apache.org/>;
- Sistemas de controlo de versões distribuídos:
 - Git: <http://git-scm.com/>;
 - Mercurial (hg): <http://mercurial.selenic.com/>;
 - Bazaar (bzzr): <http://bazaar.canonical.com/en/>;

Estes são apenas alguns exemplos dos mais usados. A grande diferença entre os centralizados e os distribuídos é que, nos centralizados existe um repositório, denominado de servidor, que armazena, a todo o momento, a versão mais recente do código fonte. Por sua vez, nos distribuídos, cada utilizador tem a sua própria cópia do repositório, que podem divergir, havendo posteriormente métodos para juntar repositórios distintos.

2 Instalação do Git

Na disciplina de Laboratórios de Informática I será utilizado o sistema *Git*. Para o instalar siga as instruções em <https://git-scm.com/downloads>.

3 Configurar o git

Para configurar o Git ao nível do sistema deve indicar o nome e o email que ficarão associados às actividades realizadas no repositório.

```
$ git config --global user.name "a999999"
```

Configura o nome que irá ficar associado aos git commits, **a999999** neste exemplo.

```
$ git config --global user.email "a999999@alunos.uminho.pt"
```

Configura o email que irá ficar associado aos git commits, **a999999@alunos.uminho.pt** neste exemplo.

4 Uso do Git

4.1 Init

Vamos criar um repositório local chamado **AulasLI1** para experimentar alguns comandos do *Git*.

```
$ git init AulasLI1
```

Verifique que foi criada na directoria actual a subdirectoria **AulasLI1/**.

4.2 Adição de novas directorias e ficheiros

O passo seguinte corresponde a adicionar novos ficheiros ou pastas que queiramos armazenar no repositório. Como exemplo, vamos adicionar um ficheiro **README.md** ao repositório. Crie na directoria **AulasLI1** um ficheiro chamado **README.md**, com o seu nome. Para adicionar o ficheiro **README.md** ao repositório terá de começar por executar o comando:

```
$ git add README.md
```

O ficheiro foi adicionado à área de preparação para que possa ser incluído no repositório, mas ainda não faz parte do repositório controlado pelo Git. Como veremos, tal só acontecerá quando for executado o comando *commit*.

4.3 Status

Um comando extremamente simples, mas bastante útil, designado por *status*, permite ver o estado atual do repositório local (sem realizar qualquer ligação ao servidor)

```
$ git status
On branch master
No commits yet
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   README.md
```

Indica que há um novo ficheiro pronto a ser inserido no repositório.

Se criar agora um ficheiro de texto, denominado `exemplo.txt` com um qualquer conteúdo, mas não o adicionar com o comando `add`, ao executar o comando `status` obtém-se (além do que já tínhamos):

```
$ git status
...
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    exemplo.txt
```

Isto indica que o Git não sabe nada sobre aquele ficheiro, e que portanto o irá ignorar em qualquer comando executado. Para que este ficheiro seja adicionado ao repositório terá que usar o comando `git add exemplo.txt`, tal como usado anteriormente.

Uma boa prática de desenvolvimento é adquirir o hábito de gerir todo o código que programar para o trabalho prático através do sistema de versões.

4.4 Add

Sempre que realizar alterações a um ficheiro e as quiser registar, terá de dar essa informação ao Git. Tal como descrito anteriormente, o mesmo comando também é usado para adicionar novos ficheiros ao repositório. Assim, depois de terminar as alterações a um ficheiro (novo ou não), deve executar o comando `git add`.

Tarefas

1. Altere o ficheiro `README.md` adicionando o número de aluno.
2. Execute o comando `git status` e verifique que obtém:

```
$ git status
On branch master
No commits yet
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   README.md
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)
    modified:   README.md
```

Untracked files:

(use "git add <file>..." to include in what will be committed)
exemplo.txt

Para além da informação anterior, é assinalado que o ficheiro `README.md` foi modificado e que é necessário fazer *add* para actualizar a versão pronta a submeter ao repositório.

3. Faça agora:

```
$ git add README.md
$ git add exemplo.txt
```

4. Volte a executar o comando `status` devendo obter:

```
$ git status
On branch master
No commits yet
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   README.md
    new file:   exemplo.txt
```

indicando que ambos os ficheiros estão agora sob controlo do Git.

4.5 Commit

Para registar as alterações e os novos ficheiros ou pastas no repositório local é necessário realizar um processo designado por *commit* (após o *add*). Isto poderá ser feito através do comando `git commit`.

```
$ git commit -m "Adicionados os ficheiros README.md e exemplo.txt"
[master (root-commit) 0ddb6d3] Adicionados os ficheiros README.md, exemplo.txt
2 files changed, 3 insertions(+)
create mode 100644 README.md
create mode 100644 exemplo.txt
```

No comando *commit* executado foi adicionada uma opção (`-m`) que é usada para incluir uma mensagem explicativa das alterações que foram introduzidas ao repositório. É boa prática adicionar uma mensagem clara em cada *commit*. Deve realizar um *commit* sempre que faça alterações ao seu código que em conjunto formem uma modificação coerente. Os seguintes exemplos podem originar novos commits:

- adicionar uma nova função;
- adicionar um nova funcionalidade;
- corrigir um bug;

Note que depois do *commit* o código está no repositório, mas apenas na sua cópia local.

4.6 Diff

O comando `git diff` mostra as diferenças linha a linha dos ficheiros alterados (antes do add).

Altere o ficheiro `README.md` acrescentando o turno de que faz parte e alterando o número de aluno (troque o "a" por "A" ou o inverso).

Execute:

```
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)
        modified:   README.md
no changes added to commit (use "git add" and/or "git commit -a")
```

e pode comprovar que há alterações identificadas.

Execute agora:

```
$ git diff
diff --git a/README.md b/README.md
index 352a4e4..7159b7a 100644
--- a/README.md
+++ b/README.md
@@ -1,2 +1,3 @@
   Aluno: Olga Pacheco
 -Número: a99999
 +Número: A99999
 +Turno: PL10
```

para identificar as diferenças entre as versões do ficheiro `README.md` (repare nas cores, no que foi substituído e acrescentado).

Se executar novamente `git add README.md` e de seguida `git diff`, verificará que não há agora diferenças identificadas. Se fizer `git status`, verificará que o ficheiro actualizado `README.md` está pronto a ser submetido ao repositório, o que acontecerá logo que faça `git commit -m "alteração README.md"`.

4.7 Log

O comando `git log` lista o histórico de versões. Depois das alterações acima mencionadas, obtemos:

```
$ git log
commit c5277f4112d6663cc8903f79fac8c540d72b705e (HEAD -> master)
Author: [omp] <omp@di.uminho.pt>
Date:   Sun Oct 24 17:37:37 2021 +0100
```

alteração de README.md

```
commit 0ddb6d30a08045670abdc5f83f6490f0003cec3
Author: [omp] <omp@di.uminho.pt>
Date:   Sun Oct 24 17:22:02 2021 +0100
```

Adicionados os ficheiros README.md e exemplo.txt

4.8 Remove

Vamos agora ver como remover ficheiros do repositório. Isto poderá ser feito através do comando `git rm`, seguido do nome do ficheiro. Para remover o ficheiro `exemplo.txt` faz-se:

```
$ git rm exemplo.txt
rm 'exemplo.txt'
```

Tal como na operação *add*, temos que fazer *commit* para que um ficheiro marcado para ser apagado seja efetivamente apagado no repositório local.

Tarefas

Analise a sequência de comandos seguinte:

```
$ git rm exemplo.txt
rm 'exemplo.txt'
```

```
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
        deleted:    exemplo.txt
```

```
$ git commit -m "remoção de exemplo.txt"
[master c359369] remoção de exemplo.txt
 1 file changed, 1 deletion(-)
 delete mode 100644 exemplo.txt
```

```
$ git status
On branch master
nothing to commit, working tree clean
```

```
$ git log
commit c3593696879c7ec6afe0b72480507452081a5475 (HEAD -> master)
Author: [omp] <omp@di.uminho.pt>
Date:   Sun Oct 24 17:42:46 2021 +0100
```

remoção de exemplo.txt

```
commit c5277f4112d6663cc8903f79fac8c540d72b705e
Author: [omp] <omp@di.uminho.pt>
Date:   Sun Oct 24 17:37:37 2021 +0100
```

alteração de README.md

```
commit 0ddb6d30a08045670abdc5f83f6490f0003cec3
Author: [omp] <omp@di.uminho.pt>
Date:   Sun Oct 24 17:22:02 2021 +0100
```

Adicionados os ficheiros README.md e exemplo.txt

NOTA: Para esclarecer dúvidas sobre algum comando poderá escrever: `git help COMMAND`.

4.9 Push and Pull

Até ao momento utilizamos o Git para gerir um repositório local. Contudo, o objectivo principal da utilização de sistemas de controlo de versões é suportar o desenvolvimento cooperativo de *software*. Nos sistemas distribuídos como o Git, cada utilizador tem a sua própria cópia do repositório, que podem divergir, havendo posteriormente métodos para juntar repositórios distintos.

Comandos como `git clone`, `git push` e `git pull` serão discutidos nas próximas fichas.

Referências

Para informação mais detalhada sugere-se a consulta de documentação do *Git*, nomeadamente:

<https://git-scm.com/doc>

<https://docs.gitlab.com/>

5 Funções recursivas sobre listas ou sobre inteiros, em Haskell

Tarefas: actualizar repositório

Vamos agora actualizar o repositório `AulasLI1` com o trabalho realizado na semana passada. Mova a directoria `Aula1/` para `AulasLI1/` (i.e. `Aula1/` fica como subdirectoria de `AulasLI1/`). Actualize o repositório.

Pretende-se de seguida programar funções recursivas simples em *Haskell* e usar o interpretador `ghci` para as testar. Para o efeito, crie a directoria `Aula2/` como subdirectoria de `AulasLI1/`, crie o ficheiro `Aula2.hs` e grave-o na directoria `AulasLI1/Aula2/`. Escreva nesse ficheiro as diversas funções que definir na próxima tarefa. Actualize o repositório `AulasLI1` sempre que considere conveniente.

Tarefas: funções recursivas

1. Defina uma função recursiva que recebe uma lista de inteiros e adiciona um valor dado a cada elemento da lista.

2. Defina uma função que recebe uma lista de **Strings** e remove todas as **Strings** iniciadas por um dado caractere.
3. Defina uma função recursiva que recebe uma lista de pares de inteiros e adiciona um valor dado à primeira componente de cada par.
4. Defina uma função recursiva que recebe uma lista, não vazia, de pares de inteiros e calcula qual o maior valor da segunda componente.
5. Defina uma função que recebe um dígito e calcula o próximo dígito (considere que o sucessor de '9' é '0'). Sugestão: use as funções `chr` e `ord` (depois de importar o módulo `Data.Char`).
6. Defina uma função que recebe uma lista de dígitos e substitui cada um deles pelo seu sucessor.
7. Defina uma função que recebe uma lista de vogais e substitui cada uma delas pela vogal seguinte (considere que a vogal seguinte a 'u' é 'a').
8. Considere o seguinte tipo de dados para representar posições de uma pessoa num sistema de coordenadas cartesiano (considere (0,0) no canto inferior esquerdo):

```

type Nome = String
type Coordenada = (Int, Int)
data Movimento= N | S | E | W   deriving (Show,Eq) -- norte, sul, este, oeste
type Movimentos = [Movimento]
data PosicaoPessoa = Pos Nome Coordenada   deriving (Show,Eq)

```

- (a) Defina uma função que calcule a posição de uma pessoa depois de executar uma sequência de movimentos:

```
posicao:: PosicaoPessoa -> Movimentos -> PosicaoPessoa
```

- (b) Dada uma lista de posições de pessoas, actualize essa lista depois de todas executarem um movimento dado:

```
posicoesM:: [PosicaoPessoa] -> Movimento -> [PosicaoPessoa]
```

- (c) Dada uma lista de posições de pessoas, actualize essa lista depois de todas as pessoas executarem uma mesma sequência de movimentos. Use as funções anteriormente definidas.

```
posicoesMs:: [PosicaoPessoa] -> Movimentos -> [PosicaoPessoa]
```

- (d) Defina uma função que calcule quais as pessoas posicionadas mais a norte:

```
pessoasNorte:: [PosicaoPessoa] -> [Nome]
```

9. Resolva outros exercícios propostos em Programação Funcional.