universidade de aveiro

Sistemas Operativos

GESTÃO DE ARMAZENAMENTOS: MONITORIZAÇÃO DO ESPAÇO OCUPADO

Bash

Professor

Nuno Lau (nunolau@ua.pt)

11/11/2023

Henrique Teixeira	Martim Souto
114588	114614
P6	P6

A distribuição de percentagem são iguais para os membros do grupo

Índice

1. Introdução 2

2. Estrutura do Código 3

2.1. “spacecheck.sh” 3

2.1.1. “Inicialização de variáveis e *arrays*” 3

2.1.2. “Tratamento de Opções” 5

2.1.3. “Chamar as funções” 6

2.2. “functions.sh” 7

2.2.1. Estrutura das funções com filtros 7

2.2.2. Especificações de cada função de filtragem 10

2.2.3. Impressão ordenada de dados 13

2.2.4. Funções de verificação auxiliares 16

2.3. “spacerate.sh” 17

2.3.1. Inicializar variáveis 17

2.3.2. Tratamento de Opções 17

2.3.3. Guardar linhas do ficheiro mais antigo num *array* 18

2.3.4. Comparar Ficheiros e Preparar output 19

2.3.5. Sistema de Prints 20

2.4.6. Problemas que surgiram na execução 21

3. Testes 22

3.1. Testes nossos 22

3.2. Testes fornecidos 22

4. Conclusão 23

5. Bibliografia 24

1. Introdução

No âmbito da cadeira Sistemas Operativos, do ano letivo 2023/2024, foi proposto um desafio prático que incide sobre a gestão de armazenamento em sistemas computacionais. Este desafio, estruturado em duas fases distintas, visa o desenvolvimento de scripts em *bash* que permitem a monitorização eficaz do espaço ocupado em disco, bem como a sua variação ao longo do tempo.

Na primeira fase do trabalho, foram desenvolvidos os scripts "spacecheck.sh" e “functions.sh”. Estes scripts são responsáveis pela visualização do espaço ocupado por ficheiros em determinadas diretorias, com a possibilidade de filtragem baseada em propriedades como nome, data de modificação, e tamanho mínimo dos ficheiros. Além disso, oferece funcionalidades como ordenação dos resultados e limitação do número de linhas exibidas. A sua relevância é notável no contexto da gestão de armazenamento, onde o conhecimento preciso do uso do espaço em disco é fundamental para a manutenção e otimização dos sistemas.

A segunda fase do trabalho englobou o desenvolvimento do script "spacerate.sh", que complementa o primeiro ao comparar a saída de duas execuções do "spacecheck.sh", permitindo assim analisar a evolução do uso do espaço em disco. Este script é particularmente útil para identificar tendências de crescimento ou diminuição do espaço ocupado, possibilitando uma gestão proativa do armazenamento.

2. Estrutura do Código

2.1. “spacecheck.sh”

```
1  #!/bin/bash
2
3  # Importa funções do ficheiro functions.sh
4  source functions.sh
```

Figura 1 – Definir *shebang* e importar o ficheiro com as funções a ser chamadas

O nosso código está organizado por funções, ou seja, temos como ficheiro principal, “spacecheck.sh” em que está programado todo o menu de opções, validação das mesmas, variáveis declaradas e a chamada de funções que estão presentes no ficheiro secundário, “functions.sh”.

2.1.1. “Inicialização de variáveis e *arrays*”

```
6  # Inicializa variáveis para controlo de opções
7  na=0
8  da=0
9  sa=0
10 ra=0
11 aa=0
12 nc=1
13 dc=1
14 sc=1
```

Figura 2 – Inicializar variáveis de controlo

Como estratégia de saber quais as *flags* que estariam ativas, nós definimos variáveis com os valores 0 e 1 (0-flag desativada, 1-flag ativada), então o nome de cada variável era a letra da *flag* seguida de um a de *active* (para a flag -n na=0).

```
16  #Arrays
17  dirs=()
18  declare -A associative
19  declare -A passed_filters
20  args_bons=()
21
22  max="Default"
23  lines_printed=1
24  folder_count=0
```

Figura 3 – Declarar *arrays* associativos, *arrays* não associativos e outras variáveis de controlo

- **dirs** – *array* que irá conter todos as diretorias presentes nos argumentos do comando de execução do *script*.

- **associative** – *array* associativo que contém informações sobre os tamanhos de cada arquivo, tem como *key* o arquivo e como *value* o tamanho associado ao arquivo

- **passed_filters** – *array* associativo onde estão guardados os ficheiros associados a cada arquivo, em que os arquivos são as *keys* e os ficheiros são os *values*.

- **args_bons** – *array* que irá conter todos os argumentos das opções já validados, ou seja, que passaram na validação feita no excerto de código da figura x.

- **max** – como podemos verificar na figura "max" está definida como "Default" pois para além de ser a variável de controlo da *flag* -l também é a variável que irá conter o número de linhas que queremos na nossa tabela.

- **lines_printed** – a variável "lines_printed" apenas é usada no ficheiro "functions.sh" é a variável que serve de *counter* no for para saber quando parar de imprimir linhas comparando-a á variável "max".

- **folder_count** – a variável "folder_count" é usada como counter para saber quantos ficheiros foram passados como argumentos para depois na função table_line_print() criarmos a condição que permita que não existam dois cabeçalhos de tabela no print ao invés de apenas um sendo que é apenas uma tabela.

2.1.2. “Tratamento de Opções”

```
33  regex_ar=()
34  options=("-n" "-d" "-s" "-l" "-r" "-a")
35
36  # Processa as opções da linha de comando com getopts
37  while getopts "n:d:s:l:ra" opt; do case $opt in
38          n)
39              regex=$OPTARG
40              if is_regex "$regex"; then
41                  regex_ar+=($regex)
42                  args_bons+=($regex)
43                  na=1
44                  nc=0
45              else
46                  echo "Error: Regex is either invalid or missing"
47                  exit 1
48              fi
49              ;;
50          d) ···
68          s) ···
79          r) ···
82          a) ···
85          l) ···
95          *)
96              echo "Error: Invalid option"
97              exit 1
98              ;;
99      esac
100 done
```

Figura 4 – Processar opções com o comando “getopts”

Como podemos verificar na figura 4, temos aqui a maneira que nós usámos para processar opções e validar as mesmas a lógica de validação é igual para todas, por exemplo na opção -n associamos o valor do argumento da opção á variável ”regex”, verificamos se realmente é uma expressão regular adicionamos ao array “regex_ar” para se houver mais do que uma expressão regular envolvida ativamos a *flag* -n, e prosseguimos para o resto do código.

Se não for uma opção válida, por exemplo -f é mostrada a utilizador a mensagem “Error: Invalid Option” e o programa acaba.

```
102 args=("$@")
103 for i in "${args[@]}"; do
104     if [[ " ${args_bons[@]} " =~ " $i " ]] || [[ " ${options[@]} " =~ " $i " ]] || [[ " ${dirs[@]} " =~ " $i " ]]; then
105         continue
106     else
107         echo "Error: Invalid argument \"$i\""
108         exit 1
109     fi
110 done
```

Figura 5 – Validar argumentos

Neste ciclo verificamos se os argumentos são usáveis de acordo o nosso código, ou seja, se o utilizador colocar um argumento qualquer o programa acaba, mesmo tendo a verificação para cada opção a mesma só verifica para os argumentos da opção então isto e necessário para qualquer argumento que não faça parte de uma opção ou não seja um diretório.

2.1.3. “Chamar as funções”

```
112  # Chama funções para processar os filtros e imprimir a tabela
113  for folder in ${dirs[@]}; do
114      name_counter=0
115      folder_count=$(($folder_count+1))
116      for key in "${!passed_filters[@]}"; do
117          unset passed_filters["$key"]
118      done
119      for key in "${!associative[@]}"; do
120          unset passed_filters["$key"]
121      done
122      table_header_print "${args[@]}"
123      no_argument "$folder"
124      for i in "${regex_ar[@]}"; do
125          name_counter=$(($name_counter+1))
126          name_filter "$folder" "$i"
127
128      done
129      date_filter "$folder" "$lDate"
130      size_filter "$folder" "$minsize"
131
132  done
```

Figura 6 – Chamada das funções do ficheiro “functions.sh” para todos os diretórios válidos introduzidos

Neste ciclo chamamos as funções para cada diretório introduzido pelo utilizador e atualizamos os *counters* para um bom funcionamento das funções provindas do ficheiro “functions.sh”.

2.2. “functions.sh”

2.2.1. Estrutura das funções com filtros

As funções são chamadas com dois argumentos, sendo o primeiro a diretoria que a função vai analisar e o segundo argumento o filtro que o utilizador introduz no comando.

No início destas funções tem de ser passado o valor da variável *checked* para *1* e tem de ser criado um *array* associativo *passed_x* que vai guardar como *keys* as pastas e como *values* os ficheiros que passam o filtro de dessa função, que irá no final da função dar *overwrite* ao *array* associativo global *passed_filters,* que é o array associativo que guarda como value todos os *files* que passaram nos filtros associados às suas pastas.

```
22  function name_filter() {
23      directory="$1"
24      padrao="$2"
25      nc=1 # declara que a função foi ativada
26      declare -A passed_name # declara um array associativo
```

Figura 7 – Inicializar variáveis de controlo

A primeira verificação da função é se ela foi ativada, se sim, vai fazer a segunda verificação, verificando se ela foi a primeira a ser corrida ou se já houve uma filtragem prévia à sua chamada.

```
28      if [ $na -eq 1 ]; then # verifica se a função
29          if [ ${#passed_filters[@]} -eq 0 ];then #
```

Figura 8 – Primeiras duas verificações

2.2.1.1. A primeira função a ser corrida

Se a função está de facto ativa entra a parte principal do script em ação, a função começa então à procura de pastas, *type –d,* na diretoria dada. Para executar isto implementamos um *while loop* , juntando o comando re*ad* com o *IFS (Internal Field Separator),* que é usado para definir o separador de campos, neste caso vai ler o nome da pasta até encontra um caracter nulo “\0”, o que é útil para ler uma pasta que tenha espaços no seu nome ou com caracteres especiais, guardando esta pasta na variável $k. Verificamos então se a pasta é *readable.* Se temos acesso à pasta é então criado a variável *size* com valor 0, que vai ser valor do *associative* da *key* $k, é também aqui criado o array *folder_files* que é o *array* onde vão ser guardados todos os ficheiros que passarem o filtro.

```
30          while IFS= read -r -d '' k; do
31              if [ -r "$k" ]; then # verificar se a pasta é readable
32                  size=0
33                  folder_files=()
```

Figura 9 – Inicio do loop da função

Caso isto não seja verdade, o valor do *size* vai ser igual a “NA”.

```
44              else
45                  size="NA" # se não, size=NA e passa para a next iteration
46              fi
```

Figura 10 – Pasta não é readable

De seguida vai da mesma maneira procurar por ficheiros, *-type f*, na pasta. O ficheiro encontrado vai ser guardado na variável $i, se este ficheiro passar o filtro da função, o seu tamanho irá ser somado à variável size e o seu *path* ao *array folder_files*. Caso o ficheiro não seja readable, o seu *size* vai se considerado “NA” e a função vai passar para a próxima pasta.

Para evitar que um erro seja emitido, caso haja algum erro no *find*, esse erro é redirecionado para */dev/null,* neste comando *find* é usado uma *flag* –print0 para a função ler corretamente ficheiros e pastas que contenham caracteres especiais ou espaços.

```
34              while IFS= read -r -d '' i; do
35                  if [ -r "$i" ]; then # verifica se o file é readable
36                      size_i=$(du -b "$i" | cut -f1)
37                      size=$(($size+$size_i))
38                      folder_files+=("$i")
39                  else
40                      size="NA" # se não, size=NA e passa para a next iteration
41                      break
42                  fi
43              done < <(find "$k" -type f -regex ".*$padrao.*" -print0 2>/dev/null)
```

Figura 11 – Leitura das pastas e seus ficheiros

Após todos os ficheiros da pasta terem sido corridos, somos deparados com um novo desafio, não é possível associar um *array* como valor de um *array* associativo em bash. Para resolver isto, criamos uma *string* para guardar todos elementos do *array,* que, com o IFS, tem todos os elementos separados por uma vírgula, assim pudemos mais tarde recriar facilmente o *array.* Associamos então essa *string* ao valor de $k no *passed_x* e a variável *size* ao valor $k no *associative.*

```
48              passed_name["$k"]=$(IFS=,; echo "${folder_files[*]}")
49              associative["$k"]="$size"
50
51          done < <(find "$directory" -type d -print0 2>/dev/null)
```

Figura 12 – Passagem de array para string e associação dos valores aos arrays associativos

2.2.1.2. Várias filtragens

Se já houve uma filtragem prévia a função vai iterar sobre as *keys* do *passed_filters* em vez da diretoria, usando agora um *for loop*, passando novamente para *array* o antigo *folder_files,* chamado de *folder_files_before.*

```
54      else
55          # Pega no array que contem os ficheiros que passaram os filtros e filtra novamente
56          for folder in "${!passed_filters[@]}"; do
57              if [ -r "$folder" ]; then
58                  size=0
59                  folder_files=()
60                  array_string="${passed_filters[$folder]}"
61
62                  # Reconverte a string que contem os ficheiros filtrados num array
63                  IFS=, read -ra folder_files_before <<< "$array_string"
64                  for j in "${folder_files_before[@]}"; do
65                      if [ -r "$j"]; then
66                          if [[ $j =~ $padrao ]]; then
67                              size_i=$(du -b "$j" | cut -f1)
68                              size=$(($size+$size_i))
69                              folder_files+=("$j")
70                          fi
71                      else
72                          size="NA"
73                          break
74                      fi
75                  done
76              else
77                  size="NA"
78              fi
79
80              passed_name["$folder"]=$(IFS=,; echo "${folder_files[*]}") # guarda num array associativo os files que passaram o filtro
81              associative["$folder"]="$size"
82          done
```

Figura 13 – Armazenamento de informação com mais de uma filtragem

Neste excerto de código:

- A *flag* -r: Indica que a os valores lidos devem ser lidos *raw*, sem interpretar barras invertidas como escape de caracteres.
- A *flag* -a: Indica que os valores lidos devem ser atribuídos a um *array*.
- "<<<" é um *HereString,* usado, como o nome sugere, para fornecer uma *string* como entrada de um comando.

Após isto, o *passed_filters* é esvaziado para que possa ser atualizado sem informação repetida nele, passando os valores do *passed_x* para ele, sendo estes os valores que passaram a todos os filtros chamados previamente. No caso do *name_filter* é também esvaziado o *passed_name* para caso a função seja chamada mais do que uma vez.

Após terminar uma função é também chamada a função table_line_print, que só será executada se todas as funções que foram chamadas no comando estão *checked*, mais sobre esta função à frente.

```
86          # esvazia o array que contem os ficheiros que passaram os filtros
87          for key in "${!passed_filters[@]}"; do
88              unset passed_filters["$key"]
89          done
90
91          # torna os valores do passed_filters nos valores do passed_name, que foram filtrados mais recentemente
92          for i in "${!passed_name[@]}"; do
93              if [ -z "${passed_filters[$i]}" ]; then
94                  passed_filters["$i"]=""
95              fi
96              passed_filters["$i"]+="${passed_name[$i]}"
97              # isto associa um folder a uma string de files filtrados
98          done
99
100         for key in "${!passed_name[@]}"; do
101             unset passed_name["$key"] # esvazia o array passed_name para caso esta função seja chamada novamente
102         done
103
104
105         table_line_print
106     fi
107 }
```

Figura 14 – Final da função

2.2.2. Especificações de cada função de filtragem

2.2.2.1. Função *name_filter()*

A função *name_filte*r tem como segundo argumento uma expressão regular, que é guardada na variável *padrao*, o array associativo que guarda os ficheiros que passam o filtro associado a cada pasta chama-se *passed_name*.

```
34          while IFS= read -r -d '' i; do
35              if [ -r "$i" ]; then # verifica se o file é readable
36                  size_i=$(du -b "$i" | cut -f1)
37                  size=$(($size+$size_i))
38                  folder_files+=("$i")
39              else
40                  size="NA" # se não, size=NA e passa para a next iteration
41                  break
42              fi
43          done < <(find "$k" -type f -regex ".*$padrao.*" -print0 2>/dev/null)
```

Figura 15 – Filtragem dos ficheiros por nome

Nesta função a filtragem dos ficheiros é feita no próprio comando *find,* devido a este ter uma *built-in flag* de regex.

No entanto, quando a função não é a primeira a ser corrida, quando está a iterar sobre o *passed_filters* a filtragem vai ser feita com uma comparação ao *padrao*, usando a mesma lógica de seguida.

```
64          for j in "${folder_files_before[@]}"; do
65              if [ -r "$j"]; then
66                  if [[ $j =~ $padrao ]]; then
67                      size_i=$(du -b "$j" | cut -f1)
68                      size=$(($size+$size_i))
69                      folder_files+=("$j")
70                  fi
71              else
72                  size="NA"
73                  break
74              fi
75          done
```

Figura 16 – Primeiras dos ficheiros com mais de um filtro

2.2.2.2. Função *size_filter()*

O segundo argumento da função *size_filter* é um número inteiro, sendo este guardado na variável *minsize*, o array associativo desta função chama-se *passed_size*.
A filtragem desta função é feita dentro do loop que lê os ficheiros, comparando o tamanho dos ficheiros ao *minsize*.

```
122                 while IFS= read -r -d '' i; do
123                     if [ -r "$i" ]; then
124                         size_i=$(du -b "$i" | cut -f1)
125                         if [ $size_i -ge $minsize ]; then
126                             size=$(($size+$size_i))
127                             folder_files+=("$i")
128                         fi
129                     else
130                         size="NA"
131                         break
132                     fi
133                 done < <(find "$k" -type f -print0 2>/dev/null)
```

Figura 17 – Filtragem dos ficheiros por tamanho

No caso desta função a filtragem dela é sempre feita da mesma maneira, seguindo também a estrutura previamente apresentada para guardar a informação.

Figura 18 – Filtragem por tamanho com mais de um filtro

2.2.2.3. Função *date_filter()*

Esta função recebe como segundo argumento uma data máxima, guardado na variável *user_date_seconds*, sendo o *array* associativo desta função *passed_date*.
Nesta função a data de cada ficheiro é passada para segundos, para ficar no mesmo formato em que o segundo argumento está, pois que foi assim transformada a data no *spacecheck.sh*, onde esta função foi chamada. A data do ficheiro é depois comparada à data máxima passada.

```
203     if [ ${#passed_filters[@]} -eq 0 ]; then
204         while IFS= read -r -d '' k; do
205             if [ -r "$k" ]; then
206                 size=0
207                 folder_files=()
208                 while IFS= read -r -d '' i; do
209                     if [ -r "$i" ]; then
210                         file_date=$(date -r "$i" "+%Y-%m-%d")
211                         file_date_seconds=$(date -r "$i" +%s) # converte a data para segundos
212 
213                         if [[ "$file_date_seconds" -le "$user_date_seconds" ]]; then
214                             size_i=$(du -b "$i" | cut -f1)
215                             size=$(($size+$size_i))
216                             folder_files+=("$i")
217                         fi
218                     else
219                         size="NA"
220                         break
221                     fi
222 
223 
224                 done < <(find "$k" -type f -print0 2>/dev/null)
225             else
226                 size="NA"
227             fi
228 
229             passed_date["$k"]=$(IFS=,; echo "${folder_files[*]}")
230             associative["$k"]="$size"
231 
232         done < <(find "$directory" -type d -print0 2>/dev/null)
233 
```

Figura 19 – Primeiras dos ficheiros por data

Não apresentamos aqui o código do *else statement* devido a ester ser a mesma lógica previamente apresentada.

2.2.2.4. Função *no_argument()*

Esta função apenas corre quando o comando passado não inclui nenhum filtro, sendo assim a função *default*, que simplesmente guarda todos os ficheiros de todas as pastas.

Esta função não precisa de verificar se o *passed_filters* está vazio ou não porque se ela está a ser usada então o *passed_filters* nunca chegou a ter valores.

```
function no_argument() {
    # a função executa apenas quando o unico argumento é o diretorio
    directory="$1"

    if [ $na -eq 0 ] && [ $sa -eq 0 ] && [ $da -eq 0 ]; then # verifica se nehnuma função foi ativada

        while IFS= read -r -d '' k; do # guarda o directorio na variable k
            size=0
            while IFS= read -r -d '' i; do # guarda o file na variable i
                size_i=$(du -b "$i" | cut -f1) # encontra o tamanho do file
                size=$(($size+$size_i))
            done < <(find "$k" -type f -print0 2>/dev/null) # encontra todos os files dentro do diretorio k

            associative["$k"]="$size" # guarda o tamanho como value da key folder

        done < <(find "$directory" -type d -print0 2>/dev/null) # encontra todos os subdiretorios dentro do diretorio dado

        table_line_print # imprime a tabela
    fi
}
```

Figura 20 – Funcionamento da função default

2.2.3. Impressão ordenada de dados

2.2.3.1. Função *table_header_print()*

Esta é a função que lê toda a linha de comando passada na chamada do script *spacecheck.sh* e ordena o *header* da tabela á qual se vai dar *print* de seguida.
A primeira verificação que esta função faz é se já foram lidas todas as diretorias passadas no comando, apenas atuando se tal se confirmar para poder dar *print* com todas as informações recolhidas. A função começa então por dar print de “SIZE NAME” e da data em que o comando foi executado.

```
255  function table_header_print() {
256      if [ $folder_count -eq "${#dirs[@]}" ]; then
257
258          header="SIZE NAME $(date +'%Y%m%d') "
259          printf "% s % s % s" $header
260          diretorios=()
261
```

Figura 21 – Cabeçalho estático da tabela

Após isso a função vai examinar o *array* args, identificando qual é a melhor maneira de dar *print* a cada elemento, se for uma expressão regular ou uma data será entre aspas e se for um diretório no final do *header*, o resto dará print sem aspas.

```
320         for i in "${args[@]}"; do
321             if is_number "$i"; then
322                 printf " %s" "$i" # se for um numero, printa o numero
323             elif is_regex "$i" || [[ "$i" =~ ^[A-Z][a-z]{2}\ [0-9]{2}\ [0-9]{2}:[0-9]{2}$ ]]; then
324                 printf " \"%s\"" "$i" # se o argumento for uma expressão regular ou uma data, printa o argumento entre aspas
325             elif [ -d "$i" ]; then
326                 diretorios+=("$i") # se for um directory, adiciona a um array para dar print no final do header
327                 continue
328             else
329                 printf " %s" "$i"
330             fi
331         done
332         for i in "${diretorios[@]}"; do
333             printf " %s" $(basename "$i")
334         done
335         printf "\n" ""
336     fi
337 
338 }
```

Figura 22 – Parte dinâmica do cabeçalho da tabela

2.2.3.2. Função table_line_print()

A função responsável por dar *print* ao tamanho total de cada ficheiro de forma ordenada é esta.
A função verifica várias condições usando *if statements* para determinar se deve prosseguir com a impressão da tabela ou não. As condições incluem verificar se todas as funções estão *checked*(*dc*, *nc*, *sc =1*), se as informações de tamanho foram coletadas, se o número de diretórios lidos é igual ao número passado (*folder_count* é igual ao número de diretórios armazenados em *${dirs[@]}* e se a função *name_filter* foi corrida tantas vezes quantas aquelas que foi chamada.

```
# verifica se tudo o que foi pedido foi corrido
if [ $dc -eq 1 ] && [ $nc -eq 1 ] && [ $sc -eq 1 ] && [ $folder_count -eq "${#dirs[@]}" ] && [ $name_counter -eq "${#regex_ar[@]}" ]; then
```

Figura 23 – Condições necessárias para execução da função

Se todas essas condições forem atendidas, o código prossegue a ordenar os diretórios antes de imprimi-los. A ordem de classificação dos diretórios depende das variáveis *aa* e *ra*, sendo estas a maneira que o utilizador quer ver ordenada as pastas, *aa* indicando que a ordem alfabética está ativa, e *ra* indicando que a ordem reversa está ativa. Por default, a função vai imprimir do maior tamanho para o menor.
Por *default*, o comando *sort* ordena por ordem alfabética, logo, se quiser ordenar por ordem alfabética basta usar *sort*, e se quiser o inverso, *sort –r*, que é a *flag* de *reverse.*
A linha *sort -n -r*, usa o comando *sort* para ordenar a lista de pares em ordem numérica (-n) e em ordem reversa (-r). Isso significa que os pares serão ordenados do maior para o menor valor numérico. O comando *tac* inverte a ordem das linhas.

```
function table_line_print() {

    if [ $dc -eq 1 ] && [ $nc -eq 1 ] && [ $sc -eq 1 ] && [ $folder_count -eq "${#dirs[@]}" ] && [ $name_counter -eq "${#regex_ar[@]}" ]; then
        if [ $aa -eq 1 ] && [ $ra -eq 1 ]; then
            folders=($(echo "${!associative[@]}" | tr ' ' '\n' | sort -r ))
        elif [ $aa -eq 1 ]; then
            folders=($(echo "${!associative[@]}" | tr ' ' '\n' | sort ))
        elif [ $ra -eq 1 ]; then
            folders=($(for i in "${!associative[@]}"; do echo "${associative[$i]} $i"; done | sort -n -r | awk '{print $2}' | tac ))
        else
            # POR DEFAUT IMPRIME POR SIZE
            folders=($(for i in "${!associative[@]}"; do echo "${associative[$i]} $i"; done | sort -n -r | awk '{print $2}' ))
        fi
```

Figura 24 – Condições das funções prints e ordenação arrays

Em seguida, um *for loop* é usado para iterar sobre as pastas e imprimir informações sobre eles. A variável *folder_pretty* é usada para formatar o nome das pastas removendo partes indesejadas, como os pontos e barras do *path*. A variável *size* armazena o tamanho das pastas, que é recuperado do *array associativo* ${associative[@]}. Se ele for inexistente, ele é definido como "NA". Dependendo da variável *max*, obtida no caso em que o utilizador usa “-l” no comando, o código controla quantas linhas da tabela são impressas. Se *max* for igual a "*Default*", todos os diretórios são impressos. Caso contrário, apenas um número limitado de linhas (determinado por *max*) será impresso.

```
        for i in "${folders[@]}"; do
            folder_pretty=$(echo "${i}" | sed -e 's/\.\.\///g' -e 's/\.\///g')
            size="${associative[$i]}"
            if [ -z $size ]; then
                size="NA"
            fi

            if [ "$max" == "Default" ]; then
                printf "% s % s \n" "$size" "$folder_pretty"
            else
                if [ $lines_printed -le $max ]; then
                    printf "% s % s \n" "$size" "$folder_pretty"
                    lines_printed=$(($lines_printed+1))
                fi
            fi
        done
        exit 0
    fi
}
```

Figura 25 – Formatação das pastas e prints das funções

2.2.4. Funções de verificação auxiliares

2.2.4.1. Função *is_regex()*

Esta função foi criada para auxiliar a validação da expressão regular passada como argumento da opção "-n". Simplesmente verifica se a *string* passada é uma expressão *regex*, se sim, é retornado o valor 0, caso contrário é retornado o valor 1.

```
236  function is_regex() {
237      pattern="$1"
238
239      if [[ "$pattern" =~ ^[a-zA-Z0-9.*?]+$ ]]; then
240          return 0
241      else
242          return 1
243      fi
244  }
```

Figura 26 – Validação das expressões regulares

2.2.4.2. Função *is_number()*

A função *is_number* auxilia na verificação do argumento das opções *-s* e *–l*, é uma simples verificação se o argumento que lhe é passado é um número inteiro superior ou igual a 0.

```
244  function is_number() {
245      local re='^[0-9]+$'
246      if [[ $1 =~ $re ]]; then
247          return 0
248      else
249          return 1
250      fi
251  }
```

Figura 27 – Validação dos argumentos dados como inteiros

2.3. “spacerate.sh”

2.3.1. Inicializar variáveis

```
#!/bin/bash

# Inicialize as variáveis de opção para desativado
reverse=0
alphabetical=0

files=() # array para armazenar os nomes dos arquivos de entrada

```

Figura 28 – Declarar a *shebang*, definir variáveis de controlo e *arrays*

Neste ficheiro novamente optamos por usar variáveis de controlo para saber quais as flags que estão ativas e as que não estão, para conseguirmos comparar os dois ficheiros dados como argumentos, usámos uma array para guardar o nome dos mesmo e depois conseguir aceder a ambos através de indexação.

2.3.2. Tratamento de Opções

```
while getopts "ra" opt; do
    case $opt in
        r)
        reverse=1               # reverse ativo
        ;;
        a)
        alphabetical=1          # alphabetical ativo
        ;;
    esac
done
shift $((OPTIND -1))

```

Figura 29 – Uso do comando getopts para processar as opções

Neste caso apenas temos duas opções que não contêm argumentos pois são opções de modificação de ordem do output, usamos o comando case para verificar qual das opções foi escolhida pelo

utilizador ou até mesmo as duas e atualizamos a variável de controlo para o valor um querendo dizer que está ativa.

```
21  # Verificar se os argumentos fornecidos são ficheiros
22  for file in "$@"; do
23      if [ -f "$file" ]; then
24          files+=("$file")
25      else
26          echo "File $file does not exist, try again"
27      fi
28  done
29
30  # Verifique se o número de arquivos fornecidos é exatamente 2
31  if [ ${#files[@]} -ne 2 ]; then
32      echo "This script requires exactly two input files to compare."
33      exit 1
34  fi
```

Figura 30 – Uso do comando getopts para processar as opções

Com este ficheiro ao invés de estarmos a trabalhar com diretorias, estamos a trabalhar com ficheiros o que nos traz uma verificação de argumentos diferente. Para além de estarmos a trabalhar com ficheiros também estamos a trabalhar com um número de argumentos fixo, então precisamos de garantir que são apenas dois, então usamos uma verificação simples que compara o número de elementos da lista de argumentos com o número dois e se falhar o programa termina.

2.3.3. Guardar linhas do ficheiro mais antigo num *array*

```
36  # Crie um array associativo para armazenar as informações do arquivo mais antigo
37  declare -A folders_older
38
39  while IFS=' ' read -r line; do
40      if [[ "$line" != *"SIZE"* ]]; then
41          folder_older=$(echo "$line" | awk '{print $2}')
42          size_older=$(echo "$line" | awk '{print $1}')
43
44          folders_older["$folder_older"]=$size_older
45      fi
46  done < "${files[1]}"
```

Figura 31 – Criação de array associativo e guardar as informações do ficheiro mais antigo

Ao analisarmos o problema percebemos que estaríamos a trabalhar com um ficheiro mais antigo que o outro (não necessariamente), então para começarmos decidimos guardar num array associativo o tamanho e o nome do diretório antigo, onde a *key* seria o nome do diretório e o *value* o seu tamanho.

Usámos o comando awk que divide a linha por campos baseando-se nos espaços brancos. Considerando uma linha típica como "1050 SO/project/test", `awk '{print $2}'` captura "SO/project/test" como o nome do diretório, e `awk '{print $1}'` captura "1050" como o tamanho. Estes dados são então armazenados no array associativo com o nome do diretório como *key* e o tamanho como *value*.

2.3.4. Comparar Ficheiros e Preparar output

2.3.4.1. Adicionar status *NEW*

```
55  output=()   # array para armazenar as informações de saída
56  while IFS=' ' read -r line; do
57      if [[ "$line" == *"SIZE"* ]]; then
58          continue                            # saltar a linha de cabeçalho
59      fi
60
61      size_new=$(echo "$line" | awk '{print $1}')  # armazenar o tamanho da pasta
62      folder_new=$(echo "$line" | awk '{print $2}')   # armazenar o nome da pasta
63
64      if [[ ! "${folders_older[$folder_new]+_}" ]]; then         # verificar se o array com os folders antigos tem a pasta,
65          output+=("$size_new $folder_new NEW")                  # se tiver adiciona ao output com o status NEW
```

Figura 32 – Ciclo onde comparamos linha a linha o ficheiro antigo com o mais recente

Começamos por definir um array para guardar as informações para depois facilitar no *print* e no uso das *flags* -r e -a. Na linha 57 temos um *if statement* que nos permite saltar a linha do cabeçalho.

De seguida guardamos os valores das linhas usando na mesma o comando awk. Como havia um status para colocar a frente das linhas que tinham sido removidas ou no caso adicionadas ao ficheiro, o uso de um *array* associativo com as informações do ficheiro antigo facilitou as coisas e tivemos de usar um *if statement* (linha 64) que verifica se o array "folders_older" contém o "folder_new" nas suas keys.

```
66      else
67          size_older=${folders_older[$folder_new]}
68          unset "folders_older[$folder_new]"
69
70          if [ "$size_new" -gt "$size_older" ]; then      # verificar se o tamanho da pasta nova é maior que a pasta antiga
71              size_diff=$((size_new - size_older))
72              output+=("$size_diff $folder_new")
73          elif [ "$size_new" -lt "$size_older" ]; then        # verificar se o tamanho da pasta nova é menor que a pasta antiga
74              size_diff=$((size_older - size_new))
75              output+=("-$size_diff $folder_new")
76          else                                                # verificar se o tamanho da pasta nova é igual que a pasta antiga
77              output+=("0 $folder_new")
78          fi
79      fi
80  done < "${files[0]}"
```

Figura 33 – Ciclo onde comparamos linha a linha o ficheiro antigo com o mais recente

Se o *if* falhar significa que não é uma linha nova e então seguimos para a parte onde comparamos os tamanhos e definimos assim o output, se "size_new" > "size_older", significa que o tamanho aumentou então damos print ao nome do diretório com a diferença de tamanho entre o novo e o antigo. Se isso não acontecer ele segue para a próxima condição onde verifica exatamente o contrário, a diferença aqui é que se ele for menor no print irá aparecer um menos pois ele perdeu tamanho. Se nenhuma delas aconteceu quer dizer que ficou igual.

2.3.4.2. Adicionar Status *REMOVED*

```
106   # Verifique se há pastas removidas no arquivo mais antigo
107 for folder_older in "${!folders_older[@]}"; do
108       size_older=${folders_older[$folder_older]}
109       output+=("-$size_older $folder_older REMOVED")      # adiciona ao output com o status REMOVED as pastas removidas
110   done
111
112   printf "%s %s\n" "SIZE" "NAME"
```

Figura 34 – Ciclo que adiciona ao output os as linhas que foram removidas

Neste ciclo ele analisa os "folder_older" que ainda estão noa array e adiciona os mesmo ao output com o status *REMOVED*. E depois do ciclo temos o print da linha do cabeçalho

2.3.5. Sistema de Prints

```
114   # Imprimir o array de saída de acordo com as opções
115 if [ $alphabetical -eq 1 ] && [ $reverse -eq 1 ]; then
116       printf "%s\n" "${output[@]}" | awk '{print substr($0, index($0,$2)) "\t" $0}' | sort -r | awk 'BEGIN{FS="\t"}{print $2}'
117 elif [ $alphabetical -eq 1 ]; then
118       printf "%s\n" "${output[@]}" | awk '{print substr($0, index($0,$2)) "\t" $0}' | sort | awk 'BEGIN{FS="\t"}{print $2}'
119
120 elif [ $reverse -eq 1 ]; then
121       printf "%s\n" "${output[@]}" | sort -n -r | awk '{printf "%s %s %s\n", $1, $2, $3}' | tac
122 else
123       # POR DEFAUT IMPRIME POR SIZE
124       printf "%s\n" "${output[@]}" | sort -n -r | awk '{printf "%s %s %s\n", $1, $2, $3}'
125   fi
```

Figura 35 – Verificar quais opções estão ativas e dar print do array output

Neste excerto de código temos os sistemas de prints que verifica quais opções estão ativas e faz o print.

- Na primeira condição é verificado se a opção -a e a opção -r estão ambas ativas e se estiverem é executado ' printf "%s\n" " ${output[@]}" ' que imprime todos os elementos do *array* linha a linha. De seguida vem um comando 'awk' que adiciona um separador "\t" (tab) para preparar a linha para ser ordenada alfabeticamente pelo nome do diretório, 'sort -r' ordena as linhas de forma reversa e o segundo 'awk' define o separador de campo para tab e imprime a segunda coluna que é a linha original de forma reversa.

- Na segunda condição acontece exatamente o que acontece na primeira, mas agora sem usar a flag -r no comando sort pois aqui o objetivo e apenas ordenar alfabeticamente.

- Na terceira condição é usada outra *flag* do comando sort, -n que nos permite ordenar algo numericamente, como também tem a *flag* -r ele vai ordenar de forma reversa, aqui o comando awk é usado apenas para garantir uma boa formatação dos campos.

- Na quarta opção acontece o print *default* que no nosso caso é os elementos do array ordenados numericamente.

2.4.6. Problemas que surgiram na execução

```
48  # captura a ultima linha caso esta nao acabe com \n
49  if [ -n "$line" ]; then
50      folder_older=$(echo "$line" | awk '{print $2}')
51      size_older=$(echo "$line" | awk '{print $1}')
52      folders_older["$folder_older"]=$size_older
53  fi
```

```
83  if [ -n "$line" ]; then
84      size_new=$(echo "$line" | awk '{print $1}')
85      folder_new=$(echo "$line" | awk '{print $2}')
86
87      if [[ ! "${folders_older[$folder_new]+_}" ]]; then
88          output+=("$size_new $folder_new NEW")
89      else
90          size_older=${folders_older[$folder_new]}
91          unset "folders_older[$folder_new]"
92
93          if [ "$size_new" -gt "$size_older" ]; then
94              size_diff=$((size_new - size_older))
95              output+=("$size_diff $folder_new")
96          elif [ "$size_new" -lt "$size_older" ]; then
97              size_diff=$((size_older - size_new))
98              output+=("-$size_diff $folder_new")
99          else
100             output+=("0 $folder_new")
101         fi
102     fi
103 fi
```

Figura 36 e 37 – Problema com as linhas que acabam em \n

Enquanto fazíamos o nosso código na fase de testes deparamos nos com um problema, o nosso código só adicionava ao array output as linhas que acabavam em \n então para resolvermos isso e os ficheiros não terem de seguir uma estrutura para o código dar certo, nós fizemos a adição destes dois *if statements* que verificam exatamente isso e adicionam ao array as linhas que não tiverem \n também.

3. Testes

3.1. Testes nossos

Para saber se o nosso *script* estava realmente a funcionar como intencionado fomos fazendo vários testes à medida que íamos construindo o programa, para isto criamos um diretório "Teste", que continha vários subdiretórios e ficheiros de diferentes tipos. Verificamos no terminal o tamanho real de cada diretório com os filtros e comparamos com o que obtivemos na execução do nosso *script.*

```
henriqueft_04@ric:~/SO-Project$ bash spacecheck.sh -n ".*c" -s 100 -d "Nov 20 10:10" -a -r -l 4 Teste/
SIZE NAME 20231113 -n ".*c" -s 100 -d "Nov 20 10:10" -a -r -l 4 Teste
17152 Teste/aula05
0 Teste/aula02
370622 Teste/AED_Guiao_02
387774 Teste/
```

Figura 38 – Testes implementados por nós

Também testamos com o diretório da cadeira para ter um pool de dados maior.

```
henriqueft_04@ric:~/SO-Project$ bash spacecheck.sh -n ".*sh" -s 100 -d "Nov 01 10:10" -a -r -l 15 ../so
SIZE NAME 20231113 -n ".*sh" -s "100" -d "Nov 01 10:10" -a -r -l "15" so
133 so/teste_xpto
0 so/aula06/output
0 so/aula06
0 so/aula05/output
0 so/aula05
517 so/aula04
4003 so/aula03
1250 so/aula02
0 so/aula01
5903 so
```

Figura 39 – Testes com pasta da cadeira

3.2. Testes fornecidos

Corremos também o programa de testes simples fornecidos no *e-learning* pelo docente da disciplina.

```
henriqueft_04@ric:~/SO-Project/test_a1$ ./test_a1.sh
OK
```

Figura 40 – Testes simples passados

4. Conclusão

Neste projeto, a nossa meta era desenvolver dois scripts para gerenciar o armazenamento em diretórios específicos, utilizando vários filtros, e exibir as informações filtradas para o usuário. Optamos por uma abordagem baseada em funções chamadas a partir de um arquivo principal, utilizando a linguagem bash. Este projeto permitiu aprofundar nosso conhecimento em arrays associativos, loops e processamento de argumentos múltiplos.

Enfrentamos vários desafios, por vezes duvidando da possibilidade de acabar certas implementações, mas conseguimos superálos. Este processo não só fortaleceu nossa compreensão técnica, mas também aprimorou nossas habilidades de resolução de problemas.

A validade da nossa solução foi confirmada por testes elaborados por nós além dos testes propostos pelo docente. O sucesso nestes testes reafirmou a eficácia da nossa abordagem.

Um aspeto crucial do projeto foi o trabalho em equipa. A colaboração efetiva e a divisão de tarefas desempenharam um papel fundamental no cumprimento dos nossos objetivos e no enriquecimento da nossa experiência de aprendizagem.

Em resumo, este projeto não só reforçou os nossos conhecimentos técnicos, mas também ressaltou a importância do trabalho em equipa, preparando-nos para desafios futuros na nossa carreira.

5. Bibliografia

-Para a realização deste trabalho foram consultados os seguintes sites:

https://stackoverflow.com

https://linuxhint.com

https://www.gnu.org/software/bash/manual/html_node/index.html#SEC_Contents

https://man.cx/bash

https://www.educative.io