deti universidade de aveiro
departamento de eletrónica, telecomunicações e informática

TRABALHO 1 - Gestão e Armazenamento: Monitorização do espaço ocupado

Índice

(Durante a leitura, pode retornar a esta página pelo ícone 🏠 no cabeçalho.)

Objetivo do trabalho

Existem 2 scripts a desenvolver ao longo deste projeto:

1. `spacecheck.sh` - Este script permite vizualizar o espaço ocupado por todos os ficheiros selecionados dentro da diretoria (e sub-diretorias) passada como argumento (vai mostrar uma lista de diretorias e o espaço ocupado pelo tipo de ficheiro selecionado, e não cada ficheiro separadamente)
2. `spacerate.sh` - Este script compara 2 outputs criados pelo comando spacecheck.sh e verfica a diferença no espaço ocupado entre estas duas verificações. Quando existe uma diretoria presente num dos ficheiros que não está presente no outro, será assinalado com mais texto (“NEW” ou “REMOVED”).

O objetivo é que ambos os scripts funcionem apropriadamente e com a devida validação de argumentos, para além de munir o código de medidas que assegurem o seu normal funcionamento. No fim deste projeto ambos os scripts devem funcionar corretamente e deve ser possível medir o espaço ocupado por ficheiros num diretório (com ou sem filtros) e também comparar este espaço ocupado em diferentes periodos de tempo.

Parte 1 - `spacecheck.sh`

Esta função requer explorar todos os ficheiros dentro de uma diretoria (que obedeçam ao filtro especificado) e também realizar o mesmo processo para qualquer sub-diretoria encontrada. Ignorando opções adicionais que teremos que incluir mais tarde, podemos separar este script em 3 problemas:

1. Encontrar os ficheiros filtrados dentro de uma diretoria e somar o espaço ocupado por estes.
2. Encontrar subdiretorias dentro da diretoria atual e realizar o mesmo processo para estas (e somar o valor ao espaço ocupado na própria diretoria no final).
3. Ordenar as diretorias e subdiretorias por espaço ocupado e realizar o print do output.

Este problema tem contornos recursivos. Podemos elaborar uma função para resolver o problema 1, que pode ser resolvido sem grandes problemas, e também para resolver o problema 2, com uso de métodos recursivos, e finalmente uma função printer que irá responsabilizar-se pela ordenação e print do output.

Problema 1 - Encontrar e filtrar ficheiros numa diretoria e somar o espaço ocupado

Imediatamente deparamo-nos com um problema que precisamos de resolver: a gama de valores de retorno de uma função é apenas 0-255. O problema surge quando uma diretoria tem um tamanho superior a 255 bits (o que vai acontecer regularmente). Assim não basta devolver um valor - teremos que introduzir uma variável global que será um valor correspondente ao espaço ocupado pelos ficheiros filtrados na diretoria (chamado total_var).

Para calcular o valor ocupado pelos ficheiros numa diretoria criamos uma função “espaco” que recebe como argumento um diretório. Nesta função podemos resolver o Problema 1:

```
files=($(find "$dir" -maxdepth 1 -type f))
   for j in "${files[@]}"; do
       if [[ ! -d "$j" ]]; then
           space=$(du "$j" | awk '{print $1}' | grep -oE '[0-9.]+')
       fi
       if [[ $space -ge $flag_s ]] ; then
           total_var=$(( $total_var + $space ))
       fi
   done
```

“files” é um array que vai conter todos os ficheiros incluidos dentro de um diretório, e depois o for loop vai iterar sobre os elementos desse array (fazendo nova verificação de que não estamos a ver um diretório) e atribuir a space o valor que esse ficheiro ocupa no disco. De seguida, somamos esse valor a total_var e continuamos a fazer isso até termos iterado todos os ficheiros.

Problema 2 - Recursividade nas subdiretorias

Também precisamos de usar a função *espaco* nas subdiretorias do nosso argumento - não só para somar o espaço ocupado por ficheiros nestas ao valor total da diretoria principal, mas também para as armazenar no array.

```
dirs=($(find "$dir" -mindepth 1 -maxdepth 1 -type d))
for k in "${dirs[@]}"; do
      temp_var=$total_var
      total_var=0
      espaco "$k"
      total_var=$(( $temp_var + $total_var ))
done
dict["$dir"]=$total_var
```

Chamamos apenas as subdiretorias da própria diretoria (ou seja, excluimos qualquer subdiretoria de subdiretorias) para que o resultado final seja mais acertado. Asseguramos assim que cada ficheiro dentro de qualquer diretoria seja apenas lido e somado ao total uma vez. total_var é uma variável global que vai ser utilizada pela função quando a usarmos recursivamente, portanto precisamos de a guardar num espaço seguro (usando a variável temp_var) e só depois restituir o seu valor. Quando a função estiver de novo nesta função e todas as subdiretorias tiverem sido percorridas, podemos finalmente introduzir este diretório num dicionário que vai ser utilizado para imprimir o resultado final.

Problema 3 - Imprimir o output

```
ordered=($(for i in "${!dict[@]}"; do
      done | sort -k1,1nr | cut -d' ' -f1))
for i in "${!ordered[@]}" ; do
      echo "${ordered[$i]} $i"
done
```

Para imprimir uma lista ordenada (em relação ao espaço ocupado e em ordem decrescente) podemos usar a seguinte estrutura de pipes. *ordered* vai ser um array que itera sobre dict e ordena tendo em conta a 1ª coluna, usando como desempate todas as colunas até à 1ª (significado de 1,1) de forma numérica e revertida. Usando espaços como delimitadores podemos isolar a primeira coluna como sendo o espaço ocupado (vão existir problemas com isto no futuro, que são irrelevantes para resolver o problema atual).

Com isto, temos um output aceitável sem quaisquer flags ou preocupação por possíveis erros que possam surgir, algo que ainda temos que corrigir.

Problema 4 - Flags

Existem 6 flags possíveis: -n, -d, -s, -r, -a e -l. Para implementar estas flags utilizamos o comando getopts:

```
#valores por predefinição das flags
flag_d=$(date +%s) #HOJE
flag_n="*" #TODOS OS FICHEIROS
flag_s=0 #>=0 kbs
flag_r=0 #Ordem decrescente
flag_a="1,1n" #Sem ordenação por nome
flag_l=0 #Sem limite de linhas

while getopts "d:n:ras:l:" opt; do
   case $opt in
       d)
           flag_d=$(date -d "$OPTARG" +%s 2>/dev/null) #Data
especificada (No formato M d HH:MM)
           if [[ -z $flag_d ]]; then
               echo "Formato de data inválido"
               exit 1
           fi
           ;;
       n)
           flag_n="$OPTARG" #ficheiros com o padrão especificado
           if [[ "${flag_n:0:1}" == "-" ]]; then
               echo "-n requere uma string (pattern)"
               exit 1
           fi
           ;;
       r)
           flag_r=1 #Ordem crescente
           ;;
       a)
           flag_a="2,1000" #Com ordenação por nome
           ;;
       s)
           flag_s="$OPTARG" #>=flag_s kbs
           if [[ ! $flag_s =~ ^[0-9]+$ ]]; then
               echo "-s requer um número."
               exit 1
           fi
```

```
            ;;
        l)
            flag_l="$OPTARG" #Com limite de linhas flag_l
            if [[ ! $flag_l =~ ^[0-9]+$ ]]; then
                echo "-l requer um número."
                exit 1
            fi
            ;;
        \?)
            echo "A flag -$OPTARG é inválida (flags válidas são -d, -n,
-r, -a, -s e -l)"
            ;;
    esac
done
```

Existem valores default para todas as flags (caso estas não sejam usadas, estes valores default irão contribuir para um print “normal” do output). Explorando cada flag individualmente:

A **flag -d** restringe o cálculo de espaço ocupado por ficheiros que não tenham sido modificados depois de uma data oferecida como argumento. A data default é a data (e hora) atual, ou seja, todos os ficheiros passam nesse teste. Quando a flag -d é chamada, o programa vai tentar converter o próximo argumento numa data, e se não conseguir vai descartar a mensagem de erro que o comando *date* normalmente forneceria e em vez disso vai verificar se a variável ficou com um valor null (sinal de que algo não correu bem). Caso esse seja o caso, o script emite um aviso e termina o programa sem imprimir mais nada. O valor desta flag irá ser inserido dentro do comando *find* que irá agora incluir apenas ficheiros criados:

```
files=($(find "$dir" -maxdepth 1 -type f ! -newermt "@$flag_d"))
```

A **flag -n** restringe o cálculo de espaço ocupado por ficheiros que tenham um nome que satisfaça o padrão introduzido como argumento para esta flag. Por defeito, esta flag é “*” permitindo a qualquer ficheiro que seja contabilizado. Aquando a utilização da flag -n existe uma verificação para certificar que existe de facto um argumento e não outra flag logo a seguir, e caso esse seja o caso, utiliza-se essa flag para filtrar os ficheiros por nome. Esta flag também vai ser utilizada no comando *find* como já aconteceu com a flag -d:

```
files=($(find $dir" -maxdepth 1 -type f -name "$flag_n" ! -newermt
"@$flag_d"))
```

A **flag -r** reverte a ordem pela qual as linhas são imprimidas. Por defeito o valor desta flag é 0, e quando ativada muda para 1. Quando o valor da flag_r é 0, então uma nova variável *r* vai ser atualizada para “r”, caso contrário é atualizada para “”. No entanto, caso a flag -a seja ativada este não será o caso. Esta flag vai ser utilizada no comando sort:

```
    ordered=($(for i in "${!dict[@]}"; do
```

```
            done | sort -k1,1n"$r" | cut -d' ' -f1))
      for i in "${!ordered[@]}" ; do
            echo "${ordered[$i]} $i"
      done
```

A **flag -a** alterna entre ordenação por espaço ocupado (default) e por nome das diretorias. Caso não esteja presente a flag nos argumentos, o valor de *flag_a* vai ser “1,1n” que vai ser incluido no comando sort (significa fazer sort da primeira coluna à primeira coluna, numericamente), e caso contrário, esse valor irá mudar para “2,1000” (significa fazer sort da segunda coluna à milésima), o que permite gerir ordenação de nomes de diretórios com espaços. Esta flag também vai ser utilizada no comando sort:

```
      ordered=($(for i in "${!dict[@]}"; do
            done | sort -k"$flag_a$r" | cut -d' ' -f1))
      for i in "${!ordered[@]}" ; do
            echo "${ordered[$i]} $i"
      done
```

A **flag -s** apenas contabiliza os ficheiros que ocupem mais que o espaço indicado como argumento. Para isso existe uma verificação inicial de que o argumento é um número, e este número irá ser inserido num if para somar apenas se o espaço ocupado pelo ficheiro for igual ou superior ao valor na flag (fizemos isto aqui e não dentro do comando du porque estava a resultar em alguns erros em que a filtração não era efetuada corretamente):

```
       if [[ $space -ge $flag_s ]] ; then #verifica se o tamanho do
ficheiro encontra os requisitos de tamanho
            total_var=$(( $total_var + $space )) #soma o espaço do
ficheiro analisado ao total até agora
       fi
```

A **flag -l** restringe o número de linhas que o output contém. Requer um argumento, que é um número, e é aplicado aquando o print do output:

```
       counter=1
       if [[ $counter -gt $flag_l ]] && [[ $flag_l -ne 0 ]]; then
           exit 0
       fi
```

O valor default da flag_l é 0, pelo que esta verificação só é ativada se o valor for diferente de 0. Quando o while loop é executado, o contador irá incrementar em 1, e irá existir esta verificação. Isto não impede que todos os espaços tenham que ser calculados, tanto porque o espaço ocupado por uma diretoria depende de todas as subdiretorias, mas também porque no final tem que ocorrer uma ordenação e só depois podemos restringir o número de linhas

.

Outros problemas

Ainda existem alguns problemas para resolver. Nomeadamente:

- Nomes de diretorias / ficheiros com espaços (ex: “Meu Diretório”)
- Ficheiros ou diretorias inacessíveis (a diretoria onde estes ficheiros estão deve ter o valor NA)

Para tratar do primeiro problema, utilizámos um while loop com o comando *read -r -d ‘’ directory* que irá iterar sobre o comando *find [...]* e usar como delimitador o caracter nulo (\0), de maneira a evitar que diretórios com espaços sejam interpretados como diretórios diferentes.

```
   dirs=()
   while read -r -d '' directory; do
       dirs+=("$directory")
   done < <(find "$dir" -mindepth 1 -maxdepth 1 -type d ! -name "*.*"
-print0 2>/dev/null)
```

O comando < <(processo) tem duas componentes: O comando <(processo) é também chamado de substituição de processo (process substitution), que corre o processo e manda o output para um ficheiro especial (`/dev/fd/63`), e o comando < pega no ficheiro usado como argumento (ou seja o ficheiro originado de <(processo)) e utiliza-o como input dentro do loop, sem criar uma subshell. Um problema que existiria se utilizássemos um pipe normalmente.

```
   find "$dir" -mindepth 1 -maxdepth 1 -type d ! -name "*.*" | while
read -r -d '' directory; do
       dirs+=("$directory")
   done
```

A diferença no output é notória:

```
E/UNIVERSIDADE/2_ano/projetos/SO$ ./spacecheck.sh test_a1
SIZE NAME 20231111 test_a1
66508 test_a1
40368 test_a1/aaa
9544 test_a1/rrr
7240 test_a1/aaa/zzzz
```

Função spacecheck.sh executada usando process substitution

```
E/UNIVERSIDADE/2_ano/projetos/SO$ ./spacecheck.sh test_a1
SIZE NAME 20231111 test_a1
16596 test_a1
```

Função spacecheck.sh executada usando um pipe que cria uma subshell

A diferença vem do facto que quando usamos process substitution, não é criada uma subshell, o que modifica permanentemente os valores das variáveis, enquanto que usar um pipe implica criar uma subshell que iria modificar o array dentro desse subprocesso mas essas alterações não se iriam refletir no processo principal.

Quanto a ficheiros ou diretorias inacessíveis, quando isto acontece o comando *find* que normalmente retornaria uma lista de ficheiros, irá retornar um erro. Assim, este comando irá devolver 1 em vez de 0 na variável $?. Para resolver este problema basta verificar se o valor de $? é diferente de 0 (que seria o valor de retorno para um teste bem sucedido) quando iteramos sobre os ficheiros. Se este for o caso, o espaço ocupado por ficheiros no diretório será -1 (um código de erro usado quando fazemos o print) e a função devolve 1. Quando a função é chamada recursivamente não queremos somar o valor -1 ao valor respetivo à diretoria que a chamou. Assim, caso o valor de retorno da função *espaco* não seja 0, o valor de total_var é posto de novo em 0 e depois a soma é efetuada.

```
   for k in "${dirs[@]}"; do
       temp_var=$total_var
       total_var=0
       espaco "$k"
       if [[ $? -ne 0 ]];then
           total_var=0
       fi
       total_var=$(( $temp_var + $total_var ))
   done
```

Agora temos também que ajustar a função relativa ao print final. Com este elemento de NA em conta, podemos mudar todos os valores que resultem em -1 para “NA”, depois de ordenados. Isto torna bastante conveniente a leitura do output já que todas as diretorias que tenham o valor “NA” associado estarão agrupadas no final (ou inicio se a ordem for revertida) caso o utilizador não ordene por nome.

```
for key in "${!dict[@]}"; do
       printf "%s %s\n" "${dict["$key"]}" "$key"
   done | sort -k"$flag_a""$r" | while read -r line; do
       if [[ $counter -gt $flag_l ]] && [[ $flag_l -ne 0 ]]; then
           exit 0
       fi
       space=$(echo "$line" | awk '{print $1}')
       if [[ $space == -1 ]]; then
           space="NA"
       fi
       dir=$(echo "$line" | cut -d" " -f2-)
       echo "$space $dir"
       counter=$(( $counter + 1 ))
   done #ordena a dict por ordem decrescente de tamanho e guarda os
nomes dos diretórios ordenados
```

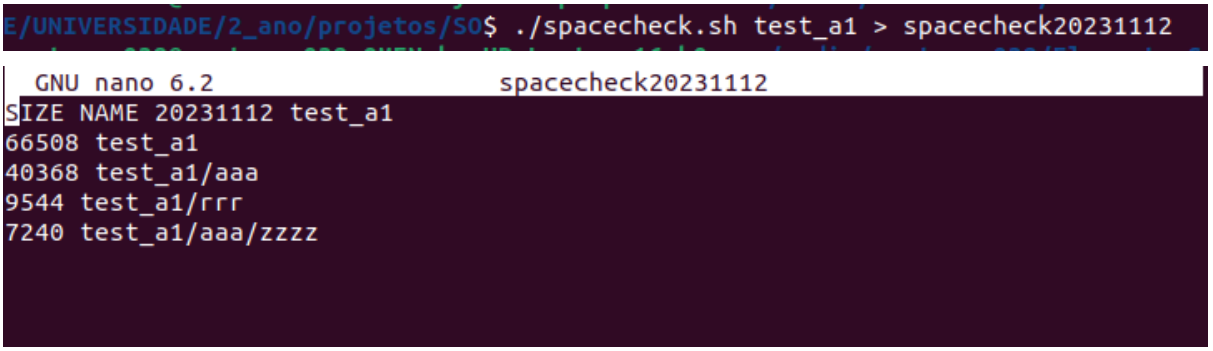
Neste excerto de código, o programa vai começar por fazer print de todas as linhas sem qualquer ordenação, que vai ser absorvido pelo pipe. É feito o sort deste output seguindo as flags a e r, e de seguida todas as linhas do output são lidas (o delimitador sendo \n por default) e aí é feita a verificação do número de linhas a aplicar e também se o valor “space” é -1, e nesse caso será convertido para “NA”.

Para verificar que o spacecheck estava a funcionar corretamente, foi feito um teste final num diretório que percorreu centenas de subdiretórios com um espaço ocupado a superar 1TB, e este conseguiu com sucesso gerir todas as possiveis dificuldades relacionadas com diretórios com espaço no nome ou existência de ficheiros / diretórios cujo espaço era impossível determinar.

Parte 2 - `spacerate.sh`

Nesta função temos que ter como argumento 2 ficheiros com output do spacecheck.sh.

Estes ficheiros são bastante fáceis de obter:



Assumimos desde já que o primeiro argumento contém o “novo” espaço ocupado por ficheiros, enquanto que o segundo argumento contém o “antigo” espaço ocupado por ficheiros.

Para chegarmos ao produto final, temos de executar os seguintes passos:

1. Ler os dados de ambos os ficheiros e colocar em 2 dicionários distintos
2. Fazer a subtração do espaço ocupado pelas diretorias e caso essa diretoria apenas apareça num ficheiro, colocar a tag “NEW” ou “REMOVED” a seguir ao nome
3. Fazer print do resultado

Primeiro que tudo temos que encontrar os ficheiros dentro dos argumentos. Quando estivemos a desenvolver as nossas funções optámos por adotar uma filosofia “dummy proof” - o que implica que em vez de restringir a maneira como os argumentos devem ser apresentados e não executar o programa ou mostrar um erro de cada vez que os argumentos são mal apresentados, tentamos aproveitar ao máximo o input que o utilizador fornece. Neste caso, um utilizador poderia, por exemplo, colocar as flags depois dos nomes dos ficheiros e o programa funcionaria na mesma. Temos um processo semelhante no script *spacecheck.sh*:

```
for l in "$@"; do
   if [[ -d "$l" ]]; then
        l=$(realpath --relative-to="$maindir" "$l")
        total_var=0
        espaco "$l"
   fi
done
```

Em que todos os argumentos são vistos e tudo o que for uma diretoria é inserido como argumento. Assim o utilizador pode inserir como argumento qualquer diretoria que queira, e o número de diretorias que queira. Inserir várias diretorias e subdiretorias não afeta o código já que utilizámos um dicionário e forçamos todos os paths a serem relativos ao diretório atual.

Voltando ao script *spacerate.sh*, temos uma estrutura semelhante à utilizada no *spacecheck.sh* para distinguir entre ficheiros e flags - damos ao utilizador a liberdade de colocar todos os ficheiros que queira, mas apenas contabilizamos os 2 primeiros ficheiros que aparecem, e deixamos de procurar após isso.

```
for arg in "$@"; do
   if [ -f "$arg" ]; then
       if [ -z "$input_novo" ]; then
           input_novo="$arg"
       elif [ -z "$input_antigo" ]; then
           input_antigo="$arg"
           break  # Encontrámos ambos os ficheiros nos argumentos
       fi
   fi
done

if [[ -z "$input_novo" ]] || [[ -z "$input_antigo" ]]; then
   echo "Usage: $0 [-a] [-r] <input_novo> <input_antigo>"
   exit 1
fi
```

Também verificamos se os ficheiros têm permissão de leitura:

```
if [ ! -r $input_antigo ]; then
   echo "File $input_antigo nao tem permissao de leitura."
   exit 1
fi

if [ ! -r $input_novo ]; then
   echo "File $input_novo nao tem permissao de leitura."
   exit 1
fi
```

E só depois deste processo de verificação e validação é que começamos a analisar os dados dentro dos ficheiros. Temos que realizar o mesmo processo de leitura para ambos os ficheiros:

```
while read -r line
do
 if [ "$first_line" = true ]; then
   first_line=false
   continue
 fi

 size=$(echo "$line" | cut -d\ -f1) #separa o tamanho do nome do
diretório
 name=$(echo "$line" | cut -d\ -f2-)

 dictantigo["$name"]=$size #guarda o tamanho e o nome do diretório numa
dict

done < "$input_antigo"

first_line=true

while read -r line
do
 if [ "$first_line" = true ]; then
   first_line=false
   continue
 fi

 size=$(echo "$line" | cut -d\ -f1) #separa o tamanho do nome do
diretório
 name=$(echo "$line" | cut -d\ -f2-)

 dictnovo["$name"]=$size #guarda o tamanho e o nome do diretório numa
dict

done < "$input_novo"
```

Utilizando o método que aplicamos anteriormente (< para usar um ficheiro como input do comando read) lemos cada linha de read (exceto a primeira linha que descartamos) e damos a *size* o valor da primeira coluna da linha, e ao resto da linha atribuímos à variável *name*, colocando-os num dicionário. Repetimos o mesmo processo para todas as linhas em ambos os ficheiros.

De seguida, temos apenas que calcular a diferença entre os valores para todas as chaves nos dois dicionários - verificando ao mesmo tempo se alguma chave estava num dos dicionários sem estar no outro, e caso esse seja o caso simplesmente adicionar ao nome a palavra respetiva.

```
for key in "${!dictnovo[@]}"; do #percorre a dict do diretório novo
   if [[ ${dictantigo["$key"]} ]]; then #se o diretório já existia no
diretório antigo
       dictfinal["$key"]=$(( ${dictnovo["$key"]} -
${dictantigo["$key"]} )) #guarda a diferença de tamanho entre o
diretório antigo e o novo

   else #se o diretório é novo
       dictfinal["$key NEW"]=${dictnovo["$key"]} #guarda o tamanho do
diretório novo
   fi
done

for key in "${!dictantigo[@]}"; do #percorre a dict do diretório antigo
   if [[ ! ${dictnovo["$key"]} ]]; then #se o diretório foi removido
       dictfinal["$key REMOVED"]="-${dictantigo["$key"]}" #guarda o
simétrico do tamanho do diretório antigo
   fi
done
```

E no final de colocarmos todos estes dados num dicionário final, falta apenas fazer o print:

```
echo "SIZE NAME"
for key in "${!dictfinal[@]}"; do #percorre a dict final
   printf "%s %s\n" "${dictfinal["$key"]}" "$key" #imprime o tamanho e
o nome do diretório
done | sort -k"$flag_a$r" | while read -r line; do #
   space=$(echo "$line" | awk '{print $1}') #
   dir=$(echo "$line" | cut -d" " -f2-) #
   printf "%s %s\n" "$space" "$dir" #
done #ordena a dict por ordem decrescente de tamanho e guarda os nomes
dos diretórios ordenados
```

De notar que para as flags a e r usamos o mesmo método que já utilizámos no script *spacecheck.sh*.

Este script trata corretamente de eventualidades como nomes com espaços e NA, como mostrado nas imagens seguintes:

```
E/UNIVERSIDADE/2_ano/projetos/SO$ ./spacerate.sh spacecheck2 spacecheck1
SIZE NAME
300 teste 01/nova pasta NEW
100 teste 01/Sub01
0 teste 01
0 teste 01/2iredor
0 teste 01/b
NA teste 01/a
NA teste 01/outra pasta NEW
NA teste 01/Sub01/SubSub01 REMOVED
-100 teste 01/antiga REMOVED
-182 teste 01/c
```

spacecheck1

```
1   SIZE NAME 20231031 teste 01
2   1376 teste 01
3   812 teste 01/Sub01
4   700 teste 01/c
5   NA teste 01/Sub01/SubSub01
6   NA teste 01/a
7   4 teste 01/b
8   0 teste 01/2iredor
9   100 teste 01/antiga
10
```

spacecheck2

```
1   SIZE NAME 20231031 teste 01
2   1376 teste 01
3   912 teste 01/Sub01
4   518 teste 01/c
5   NA teste 01/a
6   4 teste 01/b
7   0 teste 01/2iredor
8   300 teste 01/nova pasta
9   NA teste 01/outra pasta
10
```

Resumo do projeto, bibliografia e ideias finais

Com os dois scripts desenvolvidos e a funcionar corretamente, demos o trabalho por terminado. Neste projeto foi necessário fazer bastante pesquisa para ajudar ao desenvolvimento dos scripts, particularmente do *spacecheck.sh*. Consultámos diversos links associados ao Stack Overflow (https://stackoverflow.com/), e explorámos novos comandos como o process substitution, até mesmo consultando a documentação do bash para o fazer (https://www.gnu.org/software/bash/manual/html_node/Process-Substitution.html). Também utilizámos a plataforma ChatGPT (https://chat.openai.com) para debugging e para fornecer um ponto de partida para resolver alguns problemas.

Tivemos que utilizar várias ferramentas de programação, tais como programação recursiva (cuja eficiência comparámos a programação não recursiva utilizando o comando date), para além de ter de aprofundar o nosso conhecimento sobre como o sistema operativo trata o output e erros e os manipular a nosso favor, e usámos também bastante do código e técnicas utilizadas nas aulas práticas. Sentimos agora que temos um conhecimento de bash um pouco mais aprofundado, para além de ter tido algumas lições sobre como o sistema operativo trabalha (não só em relação ao output, mas por exemplo também em relação aos subshells e a diferença em relação a process substitution).