PROJECTO 3

PZYP: COMPRESSÃO LZSS (ßeta)

INTRODUÇÃO E OBJECTIVOS

A finalidade deste projecto passa por aplicar os conhecimentos adquiridos até agora (sobre programação em geral, e sobre Python 3 em particular) no desenvolvimento de aplicações úteis para manutenção e administração de sistemas. Em particular, pretende-se que desenvolva código modular, organizado em funções, utilizando os tipos de dados apropriados e, recorrendo a bibliotecas que implementem as funcionalidades de que precisa.

Pretende-se que desenvolva uma aplicação para **compressão** e **descompressão** de ficheiros, utilizando, para tal, o algoritmo LZSS, um sucessor do mais conhecido algoritmo LZ77. Estes algoritmos são a base do método de compressão empregue pelos programas zip e gzip, muito utilizados em sistemas Unix. ~~Como **extra**, pode desenvolver um comando para exibir o conteúdo de uma **directoria em árvore** ou um comando para **detectar semelhanças** entre ficheiros.~~ Todas as partes do projecto a desenvolver são mencionadas na secção **AVALIAÇÃO**. Nesta secção encontra também uma explicação sobre o que se entende por "extra" no contexto deste projecto.

PARTE I - PZYP: COMPRESSOR BASEADO EM LZSS

INTRODUÇÃO: DEFLATE, GZIP E ZIP

O **DEFLATE** é um algoritmo de compressão sem perdas (*lossless*), utilizado pelos populares programas de compressão **gzip** e **zip**, programas disponíveis na maioria dos sistemas operativos Unix. Foi concebido em 1993 por Phil Katz para a versão do 2 do programa **PKZIP**, este desenvolvido para o sistema operativo MS-DOS. Em 1996, foi criado o RFC 1951 com a especificação "oficial" do DEFLATE.

Faz parte do formato de ficheiros de aplicações como o MS Office, LibreOffice, OpenOffice, entre muitos outros. Além disso, é também o algoritmo de compressão implementado pela biblioteca **zlib**, a biblioteca utilizada pelos mais variados programas e sistemas para obter compressão (eg, kernel do Linux, macOS, PlayStation, servidor web Apache, ficheiros PNG, OpenSSL, etc.). Existem interfaces

para a zlib em quase todas as linguagens de programação, o que faz com que o DEFLATE esteja disponível para ser utilizado por qualquer ambiente ou linguagem. Por tudo isto, podemos argumentar que o DEFLATE é o algoritmo de compressão mais utilizado em todo o mundo.

O DEFLATE não deve ser confundido com o formato de ficheiro **ZIP**. Este formato é utilizado por programas, como os mencionados zip e PKZIP (e outros, como WinZip, WinRAR, 7Z, etc.), para arquivar uma colecção de outros ficheiros e directorias dentro de um só ficheiro, um pouco à semelhança do que se passa com os formatos TAR e RAR. Acontece que o zip e o PKZIP, além de arquivarem, recorrem ao DEFLATE para comprimir o conteúdo dos ficheiros dentro do arquivo ZIP. Apesar de também utilizar o DEFLATE, o gzip define o seu próprio formato de ficheiro, o formato **GZ**. Em teoria este formato permite comprimir e arquivar vários ficheiros num só arquivo. Na prática, o comando gzip cria um ficheiro GZ por cada ficheiro comprimido.

O DEFLATE resulta da combinação de dois algoritmos de compressão: **LZSS** e **Codificação de Huffman**, Por esta razão, também se utiliza a expressão "**método** de compressão" em vez de um "**algoritmo** de compressão".

LZ77

O algoritmo **LZ77** foi desenvolvido por *Abraham Lempel* e *Jacob Ziv* em 1977. O LZ77 é a base de outros algoritmos de compressão, nomeadamente, LZ78, LZSS, LZW, LZMA, entre outros, globalmente designados de algoritmos LZ.

Como os dois algoritmos, LZ77 e LZSS, são quase idênticos, vamos começar por explicar o funcionamento do LZ77.

O LZ77 é um algoritmo "**de dicionário**". Obtém compressão através da substituição de uma sequência de bytes repetida, por uma referência para a ocorrência original dessa mesma sequência. Se a referência para a sequência original ocupar menos bytes que a própria sequência, então a sequência repetida é efectivamente comprimida. O **compressor** LZ77 conserva um ***buffer* de dimensão fixa** (o dicionário) com os últimos bytes processados. Sempre que é lida uma cópia de uma sequência de bytes já existente no buffer, o LZ77 coloca na saída um **par** <distância, comprimento>. A **distância** (***offset***) diz onde encontrar dentro do buffer a primeira ocorrência da sequência repetida; o **comprimento** (***length***) guarda a dimensão da sequência ou, mais concretamente, da repetição (mais à frente sobre isto). À medida que vão sendo lidas, as sequências de bytes, repetidas ou não, são colocadas no buffer. A partir do momento que o buffer enche, sempre que uma sequências de bytes é adicionada ao final do buffer, a mesma quantidade de bytes é removida do início (estes são os bytes mais "antigos" aí presentes). Ou seja, o buffer como que "desliza" pelos dados de entrada. Devido a isto, este buffer é por vezes designado de **janela deslizante** (***sliding window***).

O **descompressor** recria o buffer e também "o faz deslizar" pelos dados de entrada, entretanto descomprimidos. O buffer é utilizado pelo descompressor para interpretar os pares distância e comprimento que vai encontrando no ficheiro comprimido. Em teoria, quanto maior o buffer, maior a taxa de compressão. Com um buffer maior, mais bytes podem ser armazenados e, como tal, mais cópias podem ser detectadas. Porém, isso também torna o compressor mais lento uma vez que tem que efectuar pesquisas mais demoradas. Além do mais, quanto maior o buffer, mais bits são necessários para representar a distância dentro do mesmo. Tamanhos comuns para este buffer são 4 KB, 16 KB e 32 KB (K = 1024).

A **dimensão do buffer** influencia o número de bits necessários para representar, quer a distância, quer o comprimento. Por exemplo, o RFC do DEFLATE especifica um buffer de 32 KB. Ou seja, este é o valor máximo da distância. Para representar esta distância precisamos de 15 bits ($\log_2 32K$). Também não é possível ter comprimentos superiores a este valor, mas, como não é expectável ter sequências tão longas, tipicamente utilizam-se menos bits para o comprimento. No caso do DEFLATE, o tamanho das sequências varia entra um mínimo de 3 bytes e um máximo de 258 bytes, o que corresponde a 256 comprimentos possíveis. Logo, são necessários 8 bits para representar o comprimento. Daqui resulta que uma referência consome 23 bits, 15 para a distância e 8 para o comprimento. Devido à Codificação de Huffman, o número médio de bits necessários é efectivamente menor.

Podemos ser levados a concluir que o comprimento deve ser sempre menor ou igual do que a distância. Um comprimento maior do que a distância significaria que a sequência repetida dependeria de bytes que ainda não foram colocados na janela. Na verdade, em determinadas situações é desejável obter comprimentos maiores do que a distância. Isto permite representar **repetições consecutivas** de uma mesma sequência, desde que esta sequência esteja "colada" ao final do buffer e continue a repetir-se "fora" da janela, até ao limite dado pela comprimento máximo permitido. Consulte as referências indicadas no final deste enunciado para aprofundar este aspecto. Alguns dos exemplos em baixo em baixo tiram partido desta possibilidade.

Para que se obtenha compressão, o par <distância, comprimento> deve ser menor do que a sequência que pretende substituir. Isto nem sempre ocorre com o LZ77. De facto, este algoritmo substitui cópias de qualquer dimensão por referências. Além do mais, como o descompressor precisa de distinguir informação comprimida de informação não comprimida, o compressor substitui cada byte não comprimido por uma referência de dimensão 0 e distância 0. Ora, isto pode levar a que um ficheiro com poucas repetições fique bastante maior após ter sido "comprimido".

LZSS

O algoritmo **LZSS** (**Lempel–Ziv–Storer–Szymanski**) foi desenvolvido por *James Storer* e *Thomas Szymanski* em 1982, com o objectivo de resolver, ou minorar, os dois problemas em cima identificados,

e com isto, tornar a compressão do LZ77 mais eficiente. Primeiro, nem todas as cópias são substituídas por referências. Por exemplo, se a referência necessitar de 2 bytes, então apenas compensa substituir cópias de sequências de 3 ou mais bytes. O LZSS define um ***break even point*** a partir do qual vale a pena tentar a compressão. Segundo, para distinguir, entre informação comprimida de não comprimida, **bytes não comprimidos**, também designados de (**bytes**) **literais**, são **prefixados com um bit a 0**, ao passo que **referências** são **prefixadas com um bit a 1**. Ou seja, cada byte não comprimido passa a necessitar de 9 bits no ficheiro de saída (trata-se, na verdade, de uma expansão de 12,5%), ao passo que cada referência de 2 bytes - ie, 16 bits - necessita de 17 bits.

O descompressor processa o ficheiro de entrada em blocos de 9 ou 17 bits, dependendo do valor do primeiro bit. Se este estiver a `0`, então lê o byte seguinte e coloca-o directamente na saída e no fim do buffer. Se estiver a `1`, então lê os dois bytes seguintes, que devem conter uma referência para uma sequência já lida, utiliza-a para localizar a sequência original no buffer e, a partir desta, recriar a cópia que é depois colocada na saída e no fim do buffer. O bit de compressão que precede cada byte literal e cada referência é designado de ***flag* de compressão**.

O LZSS acrescenta ainda outra optimização. Como vimos, não compensa tentar comprimir sequências cuja dimensão é inferior ou igual ao *break even point*. Vamos assumir que, à semelhança do que acontece com os exemplos em baixo, utilizamos referências de 2 bytes, onde 12 bits representam a distância, e 4 bits são o comprimento. O break even para estes parâmetros é 2, o que faz com que o LZSS apenas procure repetições de sequências com pelo menos 3 bytes. Uma vez que nunca vão ser geradas referências com os comprimentos 0, 1 e 2, e como o tamanho máximo de uma sequência repetida é 2^{16} - 1 = 15 bytes, o compressor pode utilizar estes valores para representar comprimentos até 18. Basta apenas subtrair 3 ao comprimento da repetição. Assim, os comprimentos 18, 17, ..., 5, 4 e 3 são **mapeados** em 15, 14, ..., 2, 1 e 0, respectivamente. Por seu turno, o descompressor necessita apenas de somar três ao comprimento de cada referência, antes de reconstituir a sequência original. Isto permite "resumir" sequências maiores, aumentando assim ligeiramente a taxa de compressão.

LZSS: EXEMPLOS

Os exemplos que se seguem ajudam a perceber o funcionamento do LZSS. Por "simplicidade pedagógica", utilizamos apenas texto nos primeiros exemplos e omitimos a codificação das flags. Como estes exemplos utilizam caracteres da gama ASCII, cada caractere equivale a 1 byte. Para efeitos de demonstração, vamos assumir que o algoritmo utiliza as tais referências de 2 bytes, com 12 bits para a distância e 4 bits para o comprimento. Isto permite localizar uma sequência repetida com dimensão entre 3 e 18 bytes. Além do mais, ficamos a saber que o tamanho da janela é de 2^{12} = 4.096 B, apesar de não ser preciso um buffer maior do que 150 B para qualquer um dos exemplos.

Para tornar os primeiros exemplos mais claros e fáceis de seguir, nestes não aplicámos qualquer

mapeamento ao comprimento das sequências.

Exemplo 1

```
 0: I meant what I said\n
20: and I said what I meant\n
44: \n
45: From there to here\n
64: from here to there\n
83: I said what I meant
```

==> LZSS ==>

```
 0: I meant what I said\n
20: and<11,7><23,8><36,5>\n
30: \n
31: From there to <8,4>\n
47: f<19,4><18,8><27,5>\n
55: <59,18>t
```

O texto original ocupa 102 bytes. Sem contar com as flags de compressão, o LZSS reduz o texto para para 43 + 2 x 8 = 59 bytes. Com as flags, temos que acrescentar um bit por cada caractere ou referência. Ou seja, necessitamos de mais 43 + 8 bits = 6,25 bytes que, na prática, são 7 bytes. A taxa de compressão obtida é assim de (102-66)/102 ≈ 35%.

Exemplo 2

```
 0: I am Sam\n
 9: \n
10: Sam I am\n
19: \n
20: That Sam-I-am!\n
35: That Sam-I-am!\n
50: I do not like\n
64: that Sam-I-am!\n
79: \n
80: Do you like green eggs and ham?\n
112: \n
113: I do not like them, Sam-I-am.\n
143: I do not like green eggs and ham.
```

==> LZSS ==>

```
 0: I am Sam\n
 9: \n
10: <5,3> <14,4>\n
16: \n
17: That<20,4>-I-am!<15,16>I do not like\n
45: t<19,14>\n
49: Do you<28,5> green eggs and ham?\n
78: <63,14> them,<64,9>.<30,15><65,18>.
```

Neste caso, os 176 bytes do texto original reduzem-se a 74 + 2 x 10 = 94 bytes. As flags de compressão acrescentam 84 bits → 11 bytes. A taxa de compressão é assim (176 - 105)/176 ≈ 40%.

Exemplo 3

Neste exemplo ilustramos duas possíveis compressões com o LZSS.

```
0: Blah blah blah blah blah!
```

==> LZSS ==>

```
0: Blah b<5,18>!
```

```
0: Blah blah blah blah blah!        ==> LZSS ==>      0: Blah b<5,5><5,5><5,5><5,3>!
```

Tecnicamente, a primeira é a mais correcta. O LZSS permite que o comprimento seja maior do que a distância sempre que a sequência à entrada seja a cópia de uma sequência "encostada", e desde que esta sequência (ou parte dela; ver o exemplo com atenção) se repita consecutivamente antes de se esgotar o comprimento máximo permitido. A segunda conversão não considera a hipótese de mais repetições consecutivas "fora" da janela.

Exemplo 4

Neste exemplo mostramos uma conversão binária. Isto é, temos em consideração que cada byte ou referência tem que ser prefixado com 1 bit a indicar se houve ou não compressão. Depois o resultado final tem quer completado com `0`'s, de modo a termos um número inteiro de bytes. Neste exemplo, já aplicamos mapeamento(s) ao(s) comprimento(s) (ver em baixo).

```
0: ABCABC                    ==> LZSS texto ==>    0: ABC<3,0>

0: ABCABC                    ==> LZSS hexa ==>     0: \x20\x90\x88\x70\x03\x00
```

A representação numérica de `A`, `B` e `C` em ASCII/Unicode é 65, 66 e 67. Em binário isto corresponde a `01000001`, `01000010` e `01000011` [pode obter o valor de `'A'` em binário, introduzindo no REPL do Python a expressão `bin(ord('A'))`]. A referência `<3, 3>`, é transformada pelo LZSS em `<3, 0>`, e quando convertida para 12 + 4 bits dá origem a `000000000011|0000`. Como o total de bits dá 44, temos que acrescentar mais 4 bits a `0` de modo a termos uma quantidade inteira de bytes que, neste caso, é 6 bytes.

Obtemos assim:

0	0	1	0	0	0	0	0	1	0	0	1	0	0	0	0	1	0	0	0	1	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
\x20								\x90								\x88								\x70								\x03								\x00							

PZYP: TRABALHO A DESENVOLVER

O objectivo final desta parte do projecto consiste em desenvolver o comando **pzyp**. O processo de desenvolvimento abrange duas ou três fases, que serão descritas já de seguida. Deve também consultar as secções **AVALIAÇÃO** e **ENTREGAS** para saber o que deve entregar, quando e como.

Primeira fase

Nesta fase deve implementar uma versão do **pzyp** para "comprimir" e descomprimir **ficheiros de texto**. Não necessita de se preocupar com manipulação binária, com flags de compressão ou com acrescento de bits a `0`. As **referências** devem ser colocadas à saída, em texto, no formato **<distancia,comprimento>**, tal como demostrado nos exemplos em cima indicados. O descompressor deve identificar uma referência através deste padrão de texto, e, caso detecte uma referência inválida, deve gerar um erro apropriado (assinalando uma excepção do tipo `PZYPError`). Para evitar ambiguidades, deve fazer o **escaping** do caractere `'<'`. Sempre que este caractere surgir na entrada, o compressor duplica-o à saída. Deste modo, o descompressor fica a saber, sem quaisquer ambiguidades, que um `'<'` representa o início de uma referência, ao passo que dois `'<'` representam uma ocorrência literal do caractere `'<'`. Este mecanismo de *escaping* é considerado uma funcionalidade **extra** do projecto.

Nesta fase o seu programa apenas necessita de ser invocado da seguinte forma:

```
$ pzyp [-d] FILE
```

Após compressão deve gerar um ficheiro com extensão `.LZS`. A opção `-d` indica que `FILE` deve ser descomprimido. Nesta fase, pode implementar a **interface da linha de comandos** por **inspecção directa de** `sys.argv`. Sugere-se, no entanto, que implemente já a interface final da linha de comandos, conforme descrito na secção seguinte.

É nesta fase que deve criar um **Worspace** apropriado no **VS Code**, bem como o repositório respectivo no **GitHub**.

Além das tarefas em cima indicados, deve também entregar os fluxogramas mencionados na secção de **ENTREGAS**.

Segunda fase

Nesta fase pretende-se que o seu comando comprima **ficheiros de todo o tipo**. Ser-lhe á fornecido o código para fazer a gestão da janela e para escrever/ler bits. Ou seja, nesta fase a sua "missão" passa por adaptar e integrar o código que desenvolveu na 1a fase, com o código que lhe será fornecido.

Agora o seu programa é invocado da seguinte forma:

```
$ pzyp [-c [-l LEVEL] | -d | -s | -h] [-p PASSWORD] FILE...
```

***NOTA**: Atenção que, ao contrário do que esta sintaxe dá a entender, deve ser possível utilizar as opções -s e -d em simultâneo.*

Eis um resumo de cada uma das opções:

- As opções `-c` e `-d`, ou `--compress` e `--decompress`, indicam se vai comprimir ou descomprimir

os ficheiros dados por `FILE...` .

- A opção `-l`/`--compression-level` com o argumento `LEVEL` indica o nível de compressão. `LEVEL` é um número a começar em 1 (ver à frente).
- A opção `-s`/`--summary` permite obter um resumo sobre o ficheiro comprimido. Ou seja, permite visualizar a meta-informação guardada no cabeçalho do ficheiro LZS (ver à frente).
- A opção `-p`/`--password` permite introduzir uma palavra-passe que será utilizada para para encriptar, através de criptografia simétrica, o ficheiro resultante da compressão LZSS. Pode utilizar um módulo da biblioteca do Python ou uma biblioteca externa instalável com pip (eg, criptography).
- A opção `-h`/`--help` mostra a ajuda do comando.

Utilize o módulo **docopt** para ler as opções da linha de comandos.

À semelhança do comando gzip, após codificação (opção `-c`) cada ficheiro `FILE` passa a possuir uma extensão específica que, neste caso, é a extensão `.LZS`. Esta extensão é removida após descodificação, tal como sucede com o comando gunzip.

Cada ficheiro LZS deve possuir um **cabeçalho** ou **rodapé** de 7 a 262 bytes. Os campos são os seguintes:

- Byte 1: número de bits para **distância**. Por exemplo, se o valor deste primeiro byte for 11, significa que são utilizados 11 bits para distância e, logo, a dimensão da janela é $2^{11} = 2$ KB.
- Byte 2: número de bits para o **comprimento**. O valor 7, a título de exemplo, significa que o comprimento máximo, sem mapeamento, é 2^7 - 1 = 127. Com mapeamento, e assumindo que são utilizados os tais 11 bits para a distância, o comprimento pode variar entre 0 + M e 127 + M, onde M = Ceiling((11+7)/8) = 3, ou seja, entre 3 e 130.
- Bytes 3-6: a **data/hora** (**timestamp**) da compressão do arquivo em segundos. Este é um número inteiro de 32 bits que pode ser obtido a partir de `time.time`. Deve ser gravado em formato *big endian* (veja `int.to_bytes` ou o módulo `struct`).
- Bytes 7-262: o **nome original** do ficheiro. Este campo é opcional e tem dimensão variável. O byte 7 diz o comprimento (N) em bytes do nome. Deste modo, o tamanho máximo para este campo é 255. O nome, propriamente dito, abrange os bytes de 8 até 7 + N. O nome deve ser cortado caso seja superior a 255 caracteres. Se N = 0, então este campo está vazio, isto é, não foi utilizado pelo compressor, e o conteúdo comprimido começa logo no byte 8.

Pode utilizar o módulo **struct** para gerar e ler este cabeçalho.

Os bytes 1 e 2 são indispensáveis para o descompressor dimensionar a janela e interpretar as referências. Os bytes de 3 a 262 são utilizados pela opção `-s` para exibir um sumário sobre o ficheiro, sendo que este sumário também inclui a informação sobre a janela. Vejamos um exemplo:

```
Name of compressed file   : dados.pdf
Compression date/time     : 2018-07-23 13:51
Compression parameters    : Buffer -> 4096 KB (12 bits), Max Seq. Len. -> 18 (4 bits)
```

Utilize `time.strftime` e `time.localtime` para exibir a data/hora no formato utilizado no exemplo. A opção `-s` é uma funcionalidade **extra** do projecto.

A opção `-l` também é uma funcionalidade **extra** deste projecto. Esta permite especificar o nível de compressão. `LEVEL` é um valor inteiro de `1` a `4`, sendo `2` o valor por omissão. O nível de compressão afecta a dimensão da janela (W) e o tamanho da sequência máxima (M) a procurar na janela.

- Nível 1: W = 1 KB ⇒ 10 bits M = 15 + 2 ⇒ 4 bits
- Nível 2: W = 4 KB ⇒ 12 bits M = 15 + 3 ⇒ 4 bits
- Nível 3: W = 16 KB ⇒ 14 bits M = 32 + 3 ⇒ 5 bits
- Nível 4: W = 32 KB ⇒ 15 bits M = 32 + 3 ⇒ 5 bits 9

A informação sobre o nível de compressão é guardada nos bytes 1 e 2 do cabeçalho. Caso não implemente a opção `-l`, então compressor e descompressor devem utilizar referências de 16 bits (12+4).

Note que o cabeçalho deve ser sempre gerado, mesmo que não implemente as opções `-s` e `-l`. A presença de um cabeçalho correctamente gerado é levada em linha de conta na avaliação.

Consultar: https://en.wikipedia.org/wiki/Lempel%E2%80%93Ziv%E2%80%93Storer%E2%80%93Szymanski
http://michael.dipperstein.com/lzss/
https://en.wikipedia.org/wiki/LZ77_and_LZ78#LZ77
https://en.wikipedia.org/wiki/DEFLATE
https://www.w3.org/Graphics/PNG/RFC-1951#spec
https://zlib.net/feldspar.html
http://www.codersnotes.com/notes/elegance-of-deflate/
http://docopt.org/

Terceira fase

Nesta fase pretende-se que desenvolva a interface gráfica, utilizando para tal a *framework* PySide6 (ou PyQt em alternativa). Pode basear-se no projecto anterior (PyCoder) para desenvolver esta interface.

Esta terceira fase pode ser desenvolvida em paralelo com a duas anteriores.

AVALIAÇÃO

No contexto deste projecto, um elemento **extra** é um componente ou funcionalidade cuja cotação (ver tabela em baixo) é substancialmente inferior à de outros componentes ou funcionalidades de dificuldade semelhante. É possível ter uma boa classificação neste projecto não realizando nenhum dos elementos extras.

ELEMENTO	**COTAÇÃO MÁXIMA (0..20)**
LZSS (Compressor)	9
LZSS (Descompressor)	8
Opção `-l`	1
Opção `-s`	1
Executável	1

Deve também elaborar um relatório (ver secção **ENTREGAS**) que valerá 20% da cotação do projecto. Aconselha-se que a primeira parte do relatório, **INTRODUÇÃO E OBJECTIVOS** e **DESENHO E ESTRUTURA**, seja realizada em primeiro lugar, antes mesmo de avançar para uma implementação.

O projecto deve ser resolvido em grupos de dois formandos. Excepcionalmente, poderá ser realizado por grupos com outras dimensões. Como incentivo, a classificação das funcionalidades **extra** pode contribuir até 2 valor(es) para a nota do próximo teste escrito e até 1 valor(es) para a nota do próximo projecto.

A classificação final será confirmada via uma avaliação oral a realizar em data a combinar.

ENTREGAS

O projecto deve ser entregue até às **23h59m** do dia **28/02/2022**. Um atraso de N dias na entrega levará a uma penalização dada pela fórmula $0.5\times2^{(N-1)}$ ($N > 0$).

Deve enviar até ao dia **13/02/2022** todos os **fluxogramas** mencionados na descrição do relatório, bem como os elementos pedidos para a **primeira fase**. Os fluxogramas devem ser todos colocados num único PDF. O código pedido deve ser colocado num ZIP que deverá ser a imagem do repositório à data de entrega. Neste deverá constar o ficheiro `pzyp.py` com a interface pedida para esta fase.

A data para entregar o pedido para a **segunda fase** é **24/02/2022**. Envie novamente um ZIP com a imagem do repositório até esta data, isto caso tenha feito o que foi pedido para esta segunda fase.

Por cada data falhada, a não-recepção do material solicitado para essa data leva a uma penalização de 10% da nota final. Ou seja, a penalização será de 20% se falhar ambas as datas.

A entrega final deverá constar de um ZIP com o repositório de GitHub que criou anteriormente. A estrutura deste repositório deve ser similar à utilizada no projecto anterior (PyCoder).

1. Cada ficheiro de código deve conter no início uma *docstring* apropriada. A *docstring* deve incluir a data de entrega e o nome dos elementos do grupo. Deve existir pelo menos um ficheiro: `pzyp.py` .

   Finalmente, este ZIP deve também incluir, na pasta apropriada, o ficheiro `relatorio.`PDF (ver a seguir).

2. Relatório em PDF que deve seguir o modelo fornecido em anexo. Em termos de formatação, adapte apenas a designação da acção e o nome dos módulos. Siga as recomendações relacionadas com a elaboração de um relatório, dadas pelo formador Fernando Ruela. O relatório deve incluir uma capa simples com o símbolo do IEFP, referência ao Centro de Formação de Alcântara, data, indicação do curso e da acção (eg, *Técnico de Informática - Sistemas 07*) e dos elementos que elaboraram o trabalho.

   Em termos de conteúdo o seu relatório deve possuir as seguintes secções e anexos:

   2.1 Introdução e Objectivos: Por palavras suas, descreva o propósito do projecto e quais os principais objectivos a atingir. Não plagie a introdução deste enunciado.

   2.2 Desenho e Estrutura: Para este projecto, é suficiente elaborar um fluxograma a descrever o algoritmo/processo principal de cada um dos programas realizados.

   No caso do **pzyp**, deve elaborar dois fluxogramas: um para o compressor, outro para o descompressor.

   Os fluxogramas devem incidir nos algoritmos e nos processos, e na lógica que lhes é subjacente. Não devem incluir instruções de programação, nem incluir detalhes de visualização (eg, "apaga ecrã" ou "formata a 20 colunas").

   2.3 Implementação: Deve descrever brevemente a solução implementada para cada um dos programas. Em particular, os aspectos mais importantes a mencionar para cada programa

são:

- As funções principais que definiu.

- As estruturas de dados principais utilizadas e sua finalidade

- Os principais módulos utilizados, qual a finalidade de cada um e, dentro destes, quais os mecanismos efectivamente utilizados.

Sempre que achar necessário pode incluir pedaços de código ilustrativos.

2.4 Conclusão: Além de seguir as recomendações indicadas no modelo a respeito da elaboração da conclusão de um relatório, deve também listar o que foi implementado e o que ficou por implementar, indicando, neste último caso, o porquê de não ter sido implementado.

NOTAS

Aqui ficam algumas noções relacionados com esta temática, em inglês, por falta de tempo para traduzir, mas também para fortalecer o nosso inglês técnico.

Encoding	*Provide a 'physical' representation for data. This can be motivated by the fact that there is no such representation available yet. But it can also be done in order to obtain different representations for the same data with the purpose of gaining efficiency in processing the data, or reducing the size (compression), or "shield it" from interference when sending it through a communication channel, or obscuring the data (encryption), etc.*
Encryption	*The process of encoding (obscuring) data to make it unreadable.*
Decryption	*The process of decoding data to make it readable again.*
Cipher	*Encoding algorithm for performing encryption and decryption.*
Plaintext	*The original data.*
Ciphertext	*The encrypted data.*
Compression	*An encoding process which aims to eliminate redundant information allowing for the original data to be compressed.*

Observe que todos os codificadores podem possuir uma interface comum assente em dois métodos: `encode` para codificar os dados, e `decode` para descodificar os dados. Uma das vantagens de definir uma interface passa pela possibilidade de substituir um codificador por outro sem que o código cliente tenha que ser alterado.