Licenciatura em Engenharia Informática

- Teoria da Informação -

Trabalho prático nº2 – Descompactação de Ficheiros “gzip”

Trabalho realizado por:

Miguel Curto Castela, uc202221272

Índice

1. Introdução

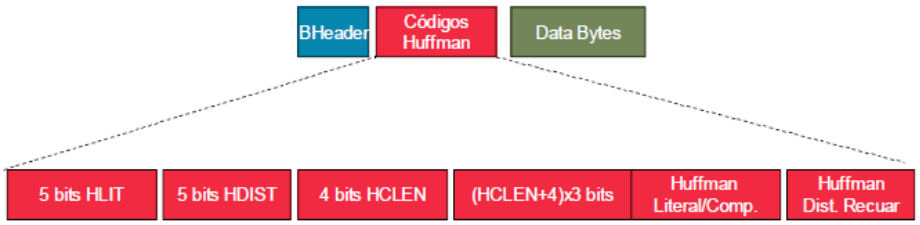
O principal objetivo deste trabalho é a criação de um código eficaz capaz de descomprimir ficheiros ***gzip***. Pretende-se, para isso, implementar um descodificador de blocos comprimidos com códigos de ***Huffman*** dinâmicos, usando o algoritmo ***deflate***.

Cada bloco lido durante a execução do programa requer a definição de três árvores de *Huffman*, cujos códigos serão posteriormente utilizados na fase de descompactação. Este processo combina a lógica das árvores de *Huffman* com o algoritmo ***LZ77***.

Neste relatório apresenta-se a estratégia adotada para a aplicação do algoritmo *LZ77* na descompactação, assim como as funções implementadas no código, dando destaque aquelas que exigem mais explicação.

2. O Algoritmo

2.1 Estrutura dos blocos de dados



Cada bloco de dados é constituído por 3 partes:

- Header: 3 *bits*, **o primeiro indica se o bloco atual é o último** e os **outros** dois **indicam o tipo de compressão usado** (este projeto apenas visa a descompressão de blocos usando arvores de *huffman*).
- Codigos *Huffman*: responsável pela definição dos códigos de huffman que serão utilizados para a descompactação:

    HLIT: 5 *bits* seguintes que definem o número de códigos no alfabeto de literais/comprimentos, este está no intervalo [257-286];

    HDIST: 5 *bits* seguintes que definem o número de códigos no alfabeto das distâncias, este está no intervalo [1-32];

    HCELN: 4 *bits* seguintes que definem o número de códigos no alfabeto dos comprimentos dos códigos;

    Após este conjunto de 14 *bits*, vêm mais ***(HCLEN +4) * 3 bits***, cada um destes subconjuntos de 3 *bits* define o comprimento do código na árvore dos

comprimentos de código do seu elemento **correspondente na tabela [16, 17, 18, 0, 8, 7, 9, 6, 10, 5, 11, 4, 12, 3, 13, 2, 14, 1, 15]**;

De seguida, há ***HLIT+257*** códigos de elementos da árvore de comprimentos de código, que correspondem aos **comprimentos de código dos elementos da árvore dos literais/comprimento**;

Por fim, há ***HDIST+1*** códigos de elementos da árvore de comprimentos de código, que correspondem aos **comprimentos de códigos dos elementos da arvore das distâncias.**

- Data Bytes: os dados comprimidos que serão descompactados com base nas arvores de Huffman definidas pela secção anterior.

3. Código e implementação dos exercícios

Nesta secção serão apresentadas com detalhe as resoluções e implementação dos exercícios, aprofundando a explicação dos **2,3,4 e 7**, tal como solicitado.

3.1 Exercício 1: método que leia o valor correspondente a HLIT, HDIST e HCLEN

O primeiro exercício pede para criar um método que leia o formato do bloco, ou seja, que devolva o valor correspondente a **HLIT, HDIST e HCLEN**.

Isto foi feito pelo método ***readDynamicBlock***, que usa o método *readBits* para ler os 5 *bits* do HLIT e do HDIST e os 4 *bits* do HCLEN:

```
def readDynamicBlock (self):
    '''Interprets Dinamic Huffman compressed blocks'''
    #redbits é dado
    HLIT = self.readBits(5)
    HDIST = self.readBits(5)
    HCLEN = self.readBits(4)

    return HLIT, HDIST, HCLEN
```

Quando chamo esta função no *decompress*:

```
#ex1 (semana1)
HLIT, HDIST, HCLEN = self.readDynamicBlock()
print("exercicio 1 ()",HLIT +257 + "Códigos Literais/Comprimento" , HDIST + 1 + "Códigos de Distância", HCLEN + 4 + "Códigos de Comprimentos de Código")
```

Adiciono 257 bits ao valor de *HLIT*, 1 *bit* ao valor de *HDIST*, e 4 *bits* a *HCLEN*, obtendo assim o resultado desejado!

3.2 Exercício 2: Criação de um método que armazena num *array* os comprimentos dos códigos do "alfabeto de comprimentos de códigos", com base em HCLEN

Neste exercício é solicitada a leitura dos comprimentos de código para o alfabeto de comprimentos.

Para isso é feita a leitura de *(HCLEN+3)*3 bits*, sendo que **cada subconjunto de 3 *bits* define o comprimento do código na árvore de comprimentos correspondente à posição em que é lido, de acordo com os índices da tabela [16, 17 ,18 ,0 ,8 ,7 ,9 ,6 ,10 ,5 ,11 ,4 ,12 ,3 ,13 ,2 ,14 ,1 ,15].**

Isto é feito através do método ***storeCLENLengths***:

```
def storeCLENLengths(self, HCLEN):
    '''Stores the code lengths for the code lengths alphabet in an array'''

    # Ordem de comprimentos em que os bits são lidos
    idxCLENcodeLens = [16, 17, 18, 0, 8, 7, 9, 6, 10, 5, 11, 4, 12, 3, 13, 2, 14, 1, 15]
    CLENcodeLens = [0 for i in range(19)]

    # CLENcodeLens[idx] = N traduz para: "o código para idx, no alfabeto de comprimentos de código, tem um comprimento de N"
    # se N == 0, o comprimento do código desse índice não é usado
    for i in range(0, HCLEN+4):
        temp = self.readBits(3)
        CLENcodeLens[idxCLENcodeLens[i]] = temp
    return CLENcodeLens
```

3.3 Exercício 3: Criação de um método que converte os comprimentos dos códigos em árvores de *huffman*

As **árvores** de *Huffman* **dos comprimentos dos códigos** vão ser basilares para o resto do algoritmo, para tal, criei o método ***createHuffmanFromLens***, este algoritmo baseia-se em 2 fundamentos:

- Todos os códigos com o mesmo comprimento vão ter valores lexicograficamente consecutivos.
  (Por exemplo, se 'D' e 'C' ambos tiverem comprimento 3 e forem os únicos no alfabeto com esse comprimento, o código de 'C' seria '110' e o código de 'D' seria '111');
- Os códigos mais curtos lexicograficamente vêm sempre antes dos mais longos

Explicação do método:

Primeiramente, guardei no índice *'i'* do *array bl_count* o número de códigos de comprimento *'i'*:

```
def createHuffmanFromLens(self, lenArray, verbose=False):
    '''Takes an array with symbols' Huffman codes' lengths and returns
    a formated Huffman tree with said codes
    If verbose==True, it prints the codes as they're added to the tree'''

    htr = HuffmanTree()
    # max_len é o codigo com o maior tamanho
    max_len = max(lenArray)
    # max_symbol é o maior símbolo a codificar, necessário para determinar o intervalo de comprimentos para a criação de códigos Huffman.
    max_symbol = len(lenArray)

    bl_count = [0 for i in range(max_len+1)]
    # Obter array com numero de codigos com comprimento N (bl_count)
    for N in range(1, max_len+1):
        bl_count[N] += lenArray.count(N)
```

A seguir, procedi à identificação do 'ponto de partida' correspondente a cada comprimento de código. Isto implica determinar o valor numérico do código de menor comprimento associado a cada comprimento. Precisamos de armazenar o código mais reduzido para o comprimento *'N'* no índice *'N'* do *array next_code*. Este processo será regido pelo seguinte algoritmo:

```
# obter o primeiro codigo de cada comprimento#
#code usado para gerar os codigos de huffman
code = 0
#next code guarda o codigo seguinte para o comprimento N
next_code = [0 for i in range(max_len+1)]
#Gera o próximo código Huffman para cada comprimento.Usa a contagem de códigos com o comprimento anterior para determinar o ponto inicial do comprimento atual.
codes_list = []
for bits in range(1, max_len+1):
    #<< move os bits para a esquerda, garantindo que os codigoes estao separados por pelo menos um bit
    code = (code + bl_count[bits-1]) << 1
    next_code[bits] = code
```

Finalmente, atribuí valores numéricos a todos os códigos, incrementado para cada código com um determinado comprimento. Iniciei com os códigos definidos na etapa anterior e adicionei cada código gerado à árvore de *Huffman htr*, associando-o ao símbolo correspondente:

```
# Define codigos para cada simbolo em ordem lexicográfica
for n in range(max_symbol):
    # Tamanho associado ao simbolo n
    length = lenArray[n]
    #se nao for zero
    if(length != 0):
        #[2:] remove o prefixo "0b"
        code = bin(next_code[length])[2:]
        # No caso de haver 0 no inicio do codigo bin,temos de os adicionar manualmente
        # length-len(code) 0s têm se ser adicionados (int nao os adiciona)
        #Calcula o número de zeros a serem adicionados ao início da representação binária para torná-la o comprimento correto
        extension = "0"*(length-len(code))
        huffman_code = extension + code
        codes_list.append((n, huffman_code))
        #Adiciona um nó à árvore Huffman com o código Huffman gerado, o índice de símbolos e imprime o código se verbose for True.
        htr.addNode(extension+code, n, verbose=True)              #incrementa o próximo código para o comprimento atual, preparando-o para a próxima iteração.
        next_code[length] += 1

print(f"lista de (símbolos, codigos): {codes_list}")
return htr;
```

O *"print(f"lista de (simbolos, codigos): {codes_list}")"* dá print a lista ordenada de codigos, como está presente nos resultados dos exercicios 3 e 6 da professora.

A arvore é depois criada no ***decompress ,*** utilizando o array criado no método anterior:

```
#ex2 (semana1)
# Armazena os comprimentos de código da árvore CLEN em uma ordem predefinida
CLENcodeLens = self.storeCLENLengths(HCLEN)
print("exercício 2 - Os comprimentos do CLEN são:", CLENcodeLens)
#print("Comprimentos de códigos dos índices i da árvore de comprimentos de código:", CLENcodeLens)
#ex3 (semana2)
# Com base nos comprimentos de código da árvore CLEN, define uma árvore Huffman para CLEN
print("exercício 3 : HuffmanTreeCLENs")
HuffmanTreeCLENs = self.createHuffmanFromLens(CLENcodeLens, verbose=False)
```

3.5 Exercício 4 e 5: criação de um método que leia e armazena num *array* os *HLIT+257* e *HDIST +1* comprimentos dos códigos referentes ao alfabeto de literais/comprimentos, codificados segundo o código de *Huffman* de comprimentos de códigos

As árvores de Huffman para o alfabeto dos **literais/comprimentos** e para as **distâncias** serão construídas de maneira semelhante à árvore ***HuffmanTreeCLEN***. No entanto, diferencia-se desta última no sentido de que, em vez de os comprimentos de código serem lidos diretamente em conjuntos de 3 bits, **os comprimentos dos códigos nas árvores dos literais/comprimentos e das distâncias serão expressos na forma de códigos provenientes da árvore *HuffmanTreeCLEN***.

O método que eu criei por esta leitura é o ***storeTreeCodeLens*** e este recebe 2 parâmetros, o **tamanho da árvore cujos comprimentos de código serão guardados** e a **árvore dos comprimentos de código** (*Huffman TreeCLEN*). Isto porque quando queremos guardar as **árvores**

de códigos literais/comprimentos e **as das distâncias baseadas nos códigos de arvore de *CLEN*** a leitura será feita até os *arrays* de comprimento de código terem o tamanho adequado: ***HLIT + 257*** (exercício 4) e **HDIST +1** (exercício 5), correspondentemente:

```
#ex4 (semana3)
# Armazena os comprimentos de código da árvore literal e de comprimento com base nos códigos da árvore CLEN
LITLENcodeLens = self.storeTreeCodeLens(HLIT + 257, HuffmanTreeCLENs)
print("exercício 4 LEN:", LITLENcodeLens)
#ex5 (semana 4)
# Armazena os comprimentos de código da árvore de distância com base nos códigos da árvore CLEN
DISTcodeLens = self.storeTreeCodeLens(HDIST + 1, HuffmanTreeCLENs)
print("exercício 5 LEN:", DISTcodeLens)
```

Isto quer dizer que o código do método terá de estar contido no seguinte ciclo *while*:

```
def storeTreeCodeLens(self, size, CLENTree):
    '''Takes the code lengths huffmantree and stores the code lengths accordingly'''

    # Array onde o comprimento dos codigos ira ser guaradado
    treeCodeLens = []
    prevCode=-1

    while (len(treeCodeLens) < size):
```

A cada iteração, o processo inicia-se ao posicionar o nó atual na raiz ***HuffmanTreeCLEN***. Em seguida, são lidos *bits* sequencialmente até que uma folha da árvore seja encontrada utilizando o método *nextNode* da classe *HuffmanTree*. A verificação ocorre com base na lógica de que o método nextNode retorna -1 quando o código atual é **inválido** e -2 quando o código atual não **corresponde a uma folha da árvore:**

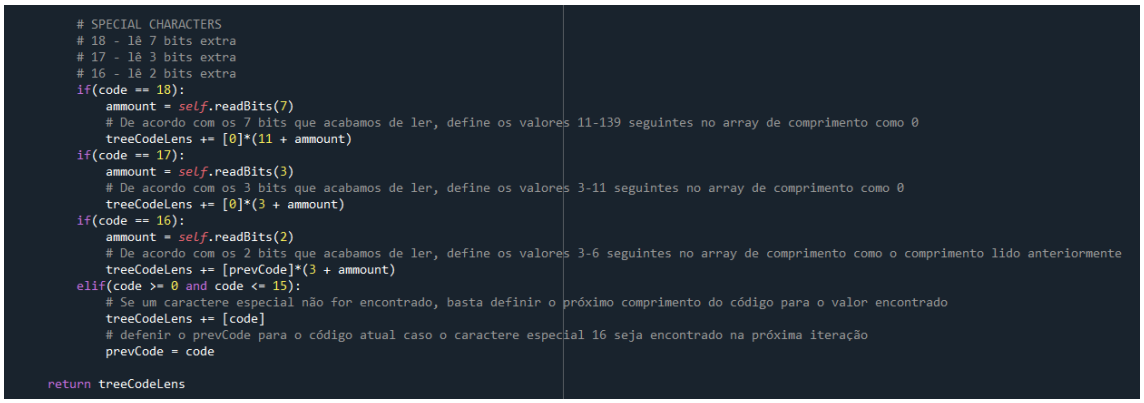
```
while (len(treeCodeLens) < size):
    # mete o nó atual na raiz da arvore de huffman
    CLENTree.resetCurNode()
    found = False

    # Durante a leitura, se uma folha não foi encontrada, continue procurando bit a bit
    while(not found):
        curBit = self.readBits(1)
        # atualiza o nó atual de acordo com o bit lido
        code = CLENTree.nextNode(str(curBit))
        if(code != -1 and code != -2):
            # se foi encontrada uma foha, dar break do loop
            found = True
```

Após identificar uma folha, simplesmente adicionar o símbolo correspondente ao array de comprimentos de código **não é suficiente**. Antes desse passo, é necessário lidar com caracteres especiais que seguem regras específicas:

- 16: Lê mais 2 *bits* que **definem quantas vezes será copiado o caracter lido no ciclo anterior**, o número de cópias pode variar entre 3 e 6.
- 17: Lê mais 3 *bits* que **definem quantas vezes será inserido '0'**, o número de inserções pode variar entre 3 e 10.
- 18: Lê mais 7 *bits* que **definem quantas vezes será inserido '0'**, o número de inserções pode variar entre 11 e 138.

Estes caracteres são verificados e tratados no algoritmo em:



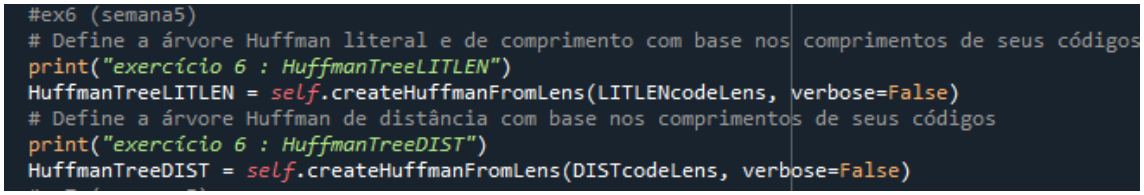
```
        # SPECIAL CHARACTERS
        # 18 - lê 7 bits extra
        # 17 - lê 3 bits extra
        # 16 - lê 2 bits extra
        if(code == 18):
            ammount = self.readBits(7)
            # De acordo com os 7 bits que acabamos de ler, define os valores 11-139 seguintes no array de comprimento como 0
            treeCodeLens += [0]*(11 + ammount)
        if(code == 17):
            ammount = self.readBits(3)
            # De acordo com os 3 bits que acabamos de ler, define os valores 3-11 seguintes no array de comprimento como 0
            treeCodeLens += [0]*(3 + ammount)
        if(code == 16):
            ammount = self.readBits(2)
            # De acordo com os 2 bits que acabamos de ler, define os valores 3-6 seguintes no array de comprimento como o comprimento lido anteriormente
            treeCodeLens += [prevCode]*(3 + ammount)
        elif(code >= 0 and code <= 15):
            # Se um caractere especial não for encontrado, basta definir o próximo comprimento do código para o valor encontrado
            treeCodeLens += [code]
            # defenir o prevCode para o código atual caso o caractere especial 16 seja encontrado na próxima iteração
            prevCode = code

    return treeCodeLens
```

Por fim, o método retorna ***TreeCodeLens*** e o valor de retorno é chamado no respetivo *array* de comprimentos de código.

3.6 Exercício 6: determine os códigos de Huffman referentes aos dois alfabetos (literais/comprimentos e distâncias) e armazene-os num *array*

Para isso usei o método do ponto3, usando as variáveis ***LITLENcodeLens*** e ***DistcodeLens*** criadas no exercício anterior par criar as **árvores de *Huffman*** respetivas:



```
#ex6 (semana5)
# Define a árvore Huffman literal e de comprimento com base nos comprimentos de seus códigos
print("exercício 6 : HuffmanTreeLITLEN")
HuffmanTreeLITLEN = self.createHuffmanFromLens(LITLENcodeLens, verbose=False)
# Define a árvore Huffman de distância com base nos comprimentos de seus códigos
print("exercício 6 : HuffmanTreeDIST")
HuffmanTreeDIST = self.createHuffmanFromLens(DISTcodeLens, verbose=False)
```

3.7 Exercicio 7: Criação da função necessária à descompactação dos dados comprimidos com base nos códigos de Huffman e no algoritmo LZ77

Depois de terem sido definidas as árvores de *Huffman* para os literais/comprimentos e para as distâncias, nomeadamente a ***Huffman*TreeLITLEN** e a ***HuffmanTreeDIST***, o próximo passo envolve a leitura da parte final do bloco que contém os dados comprimidos. A descompressão será realizada de acordo com o algoritmo ***LZ77***, utilizando o método ***decompressLZ77***.

O código do caractere **256** na árvore de *Huffman TreeLITLEN* representa um **caráter especial** que indica o final do bloco. Com isso em mente, a leitura desta parte do bloco será realizada dentro de um ciclo *while*. Em cada iteração do ciclo, o nó atual é colocado na raiz da árvore ***HuffmanTreeLITLEN***, e *bits* são lidos sequencialmente até encontrar uma folha dessa árvore.

```
# le da stream do input ate 256 ser encontrado
while(codeLITLEN != 256):
    # Dá reset ao nó atual para a base da arvore
    HuffmanTreeLITLEN.resetCurNode()

    foundLITLEN = False
    distFound = True

    # enquanto um literal ou comprimento não seja encontrado na árvore LITLEN, continua a pesquisar bit a bit

    while(not foundLITLEN):
        curBit = str(self.readBits(1))
        #Dá update ao nó atual de acordo ao bit lido
        codeLITLEN = HuffmanTreeLITLEN.nextNode(curBit)

        #Caso seja atingida uma folha na árvore LITLEN, segue as instruções de acordo com o valor encontrado
        if (codeLITLEN != -1 and codeLITLEN != -2):
            foundLITLEN = True
```

Despois da folha ser encontrada, é necessário apurar se o caracter associado se trata de um literal ou de um comprimento (**literais valores menores do que 256 e comprimentos são maiores que 256),** caso se trate de um literal, basta adicionar ao *output* o respetivo caractere:

```
    # Se o código atingido estiver no intervalo [0, 256[, apenas adiciona o valor lido correspondente a um literal ao array de saída
    if(codeLITLEN < 256):
        output += [codeLITLEN]
```

Caso se trate de um caractere ***C*** de comprimento, há algumas regras especiais para saber o seu valor:

- Se ***C*** estiver no intervalo [257, 265], o comprimento é definido por ***C***-257+3
- Se ***C***>265 será necessário ler bits extra. A **quantidade** de bits extra a ler é definida pelo índice ***C***-265 do *array* (definido no inicio do método):

```
def decompressLZ77(self, HuffmanTreeLITLEN, HuffmanTreeDIST):

    #Quantos bits são necessários ler se o comprimento da leitura do código for maior que 265
    ExtraLITLENBits = [1, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 4, 4, 4, 4, 5, 5, 5, 5]
```

O valor do **comprimento** é definido pela soma do mesmo índice do *array:*

```
#Comprimento necessário adicionar se o código de comprimento lido for maior que 265
ExtraLITLENLens = [11, 13, 15, 17, 19, 23, 27, 31, 35, 43, 51, 59, 67, 83, 99, 115, 131, 163, 195, 227]
```

e dos bits acabados de ler, ou seja: *ExtraLITLENBits*[***C*** – 265]+*ExtraLITLens*[***C***-265]:

```
# Se o código estiver no intervalo [257, 265[, define o comprimento da string a ser copiada como o código lido - 257 + 3
if(codeLITLEN < 265):
    length = codeLITLEN - 257 + 3

# Os códigos no intervalo [265, 285] são especiais e requerem mais bits para serem lidos
else:
    # dif define os índices nos "Arrays Extras" a serem usados
    dif = codeLITLEN - 265
    # Quantos bits extras precisarão ser lidos
    readExtra = ExtraLITLENBits[dif]
    # Quanto comprimento extra adicionar
    lenExtra = ExtraLITLENLens[dif]
    length = lenExtra + self.readBits(readExtra)
```

Depois de obter um valor de comprimento, a próxima etapa é determinar a distância para retroceder. Para isso, será realizada uma busca na árvore *HuffmanTreeDist* até se encontrar uma folha:

```
while(not distFound):
    distBit = str(self.readBits(1))
    # Atualiza o nó atual de acordo com o bit acabado de ler
    codeDIST = HuffmanTreeDIST.nextNode(distBit)

    # Se uma folha for atingida na árvore LITLEN, siga as instruções de acordo com o valor encontrado
    if(codeDIST != -1 and codeDIST != -2):
        distFound = True
```

Tal como na leitura dos comprimentos, há algumas regras especiais a aplicar após a leitura do caracter distância ***d***

- Se ***d*** pertencer ao intervalo [0,3] o valor da distância é dado por ***d***+1
- Se ***d***>3 é necessário ler bits extra, a **quantidade** de *bits* extra a ler é definido pelo índice ***d***-4 do *array ExtrDISTBits*:

```
#Quantos bits necessários ler se o código de distância lido for maior que 4
ExtraDISTBits = [1, 1, 2, 2, 3, 3, 4, 4, 5, 5, 6, 6, 7, 7, 8, 8, 9, 9, 10, 10, 11, 11, 12, 12, 13, 13]
```

O valor **distancia** é dado pela soma do mesmo índice do *array ExtraDISTLens*:

```
#Distância necessária adicionar se o caractere especial lido for maior que 4
ExtraDISTLens = [5, 7, 9, 13, 17, 25, 33, 49, 65, 97, 129, 193, 257, 385, 513, 769, 1025, 1537, 2049, 3073, 4097, 6145, 8193, 12289, 16385, 24577]
```

Ou seja, *ExtraDISTBits*[***d***-4] + *ExtraDISTLens*[***d***-4]:

```
# Se o código lido estiver no intervalo [0, 4[, define a distância para retroceder como o código lido + 1
if(codeDIST < 4):
    distance = codeDIST + 1

# dif define os índices nos "Arrays Extras" a serem usados
else:
    # dif define os índices nos "Arrays Extras" a serem usados
    dif = codeDIST - 4
    readExtra = ExtraDISTBits[dif]
    # Quantos bits extras precisam ser lidos
    distExtra = ExtraDISTLens[dif]
    # Quanto distância extra adicionar
    distance = distExtra + self.readBits(readExtra)
```

No final deste procedimento, tanto o **comprimento** (*length*) quanto a **distância** (*distance*) serão armazenados na memória. Portanto, basta copiar o valor de *length* a partir do índice *-distance* para o início do *output*:

```
# Para cada uma das iterações no intervalo(length), copie o caractere no índice len(output)-distance para o final do array de saída
for i in range(length):
    output.append(output[-distance])
```

O método é chamado no ***decompress,*** e assim os dados são descomprimidos de acordo com o algoritmo ***LZ77:***

```
#ex7 (semana 5)
# Com base nas árvores definidas até agora, descomprime os dados de acordo com o algoritmo Lempel-Ziv77
output = self.decompressLZ77(HuffmanTreeLITLEN, HuffmanTreeDIST, output)
print("exercício 7 :", output)
```

3.8 Exercício 8: Gravação dos dados descompactados num ficheiro com o nome original

Possível leitura de mais blocos:

Neste algoritmo, é crucial manter em memória os últimos 32768 caracteres lidos, garantindo a capacidade de retroceder uma distância de até 32768, mesmo que algumas dessas distâncias estejam em blocos anteriores ao bloco que está sendo atualmente lido. Para alcançar isso, ao final de cada bloco, os caracteres excedentes (aqueles além de *len(output)-32769*) são escritos no arquivo *.txt* e removidos da memória do programa. Assim, apenas os últimos 32768 caracteres são mantidos em armazenamento.

Esse método bloco a bloco foi **escolhido como uma alternativa mais eficiente** do que (a partir do momento em que a memoria chega ao comprimento de 32768 e um caractere é lido) salvar

caractere a caractere no arquivo .txt e adicionar o último caractere lido ao final do *array*, pois essa abordagem é **ineficiente**.

Para a implementação deste algoritmo a seguinte operação acontece sempre após a leitura de um bloco (presente no ***decompress***):

```
if(len(output) > 32768):
    # Escreve cada caractere que excede a faixa de 32768 no arquivo
    f.write(bytes(output[0 : len(output) - 32768]))
    # Mantém o restante no array de saída
    output = output[len(output) - 32768 :]

# Atualiza o número de blocos lidos
numBlocks += 1
```

Por fim, quando todos os blocos tiverem sido percorridos, resta guardar no ficheiro *.txt* (com o nome original do ficheiro) os bytes atualmente no *array* de *output*:

```
if __name__ == '__main__':

    # gets filename from command line if provided
    fileName = "FAQ.txt.gz"
    if len(sys.argv) > 1:
        fileName = sys.argv[1]

    # decompress file
    gz = GZIP(fileName)
    gz.decompress()
```

```
f = open(self.gzh.fName, 'wb')
output = []
```

```
    #ex8 (semana5)

    # Write the bytes corresponding to the output array elements
    f.write(bytes(output))
    # Close the file
    f.close
```

3.9 Testes adicionais

Para alem do teste original "**FAQ.txt**" que o algoritmo descomprimiu corretamente usando **1 bloco** (de 32768 caracteres), o professor disponibilizou uma imagem comprimida **"sample_image.jpeg"**, um ficheiro de áudio **"sample_audio.mp3"** e um texto maior que o original **"sample_large_text.txt".** Todos estes descomprimiram corretamente usando o algoritmo, precisando o texto grande de **4 blocos**, a imagem de **1** e o ficheiro áudio de **9**.

O número de blocos é mostrado como mensagem na consola, após os dados de cada função serem impressos.

4. Conclusão

Este projeto viabilizou o desenvolvimento de um decodificador para blocos comprimidos que empregam códigos de *Huffman* dinâmicos por meio do algoritmo *deflate*. Isso proporcionou-me uma compreensão mais aprofundada sobre o funcionamento de algoritmos de compressão de dados, especialmente no contexto da utilização de árvores de *Huffman* para esse fim.