

Comunicação Digital

Trabalho Prático - Módulo 2

2º Módulo

Grupo 03

49470 Ana Carolina Pereira
49465 Carolina Tavares
49988 Danilo Vieira

Licenciatura em Engenharia Informática e de Computadores
Semestre de Verão 2022/2023

19/6/2023

Índice

1. INTRODUÇÃO	3
2. EXERCÍCIO 1	4
2.1. ALÍNEA A)	4
2.1.1. Sub-alínea i)	4
2.1.2. Sub-alínea ii)	5
2.1.3. Sub-alínea iii)	6
2.2. ALÍNEA B)	7
3. EXERCÍCIO 2	7
3.1. Alínea a)	7
3.2. Alínea b)	9
3.3. Alínea c)	10
4. Conclusão	11

1. Introdução

Este trabalho consistiu na realização de vários exercícios em Linguagem C e Python, com foco no estudo e aplicação de conceitos sobre SCD e códigos de controlo de erros. No âmbito da realização destes exercícios, foi necessário aprofundar vários conhecimentos lecionados ao longo do semestre. Um aspecto importante abordado neste projeto foi a implementação de códigos de controlo de erros. Neste sentido serão exploradas técnicas como checksum, paridade, códigos de Hamming entre outras, para mitigar erros e assegurar a precisão das informações.

2. Exercício 1

2.1. Alínea a)

2.1.1. Sub-alínea i)

Para a resolução desta alínea foram realizadas 3 funções:

A função ***symbolToBinaryConversion*** é usada para converter uma sequência de caracteres em binário. Desta forma a função recebe uma sequência de caracteres como parâmetro e retorna uma string que contém a representação binária dos caracteres passados.

Para converter os caracteres em binário, é utilizada a codificação **UTF-8** para converter a sequência de caracteres numa matriz de bytes. Posto isto, itera sobre os bytes e converte cada byte numa representação binária de 8 bits usando a função **format()**. Assim, todas as representações binárias são concatenadas numa única string de forma a ser retornada.

A função ***binaryToSymbolConversion*** recebe uma sequência de bits como parâmetro. Inicialmente, a sequência é convertida num número inteiro, depois, o código verifica o número mínimo de bytes necessários para representar o inteiro convertido usando a função **bit_length**. Com base nesse número de bytes, a sequência de bits é convertida de volta numa matriz de bytes usando a função **to_bytes**. Por fim, a matriz de bytes é decodificada numa sequência de caracteres usando a função **decode**. O resultado final é retornado com a representação ASCII dos símbolos.

Para o exemplo representado na Figura 1 o resultado obtido foi o representado na Figura 2.

```
t1 = symbolToBinaryConversion("hello")
print(t1)
t1a = binaryToSymbolConversion(t1)
print(t1a)
```

Figura 1 - Exemplo de teste

```
0110100001100101011011000110110001101111
hello
```

Figura 2 - Resultado do teste 1

```
PS C:\isel\CD\AULAPRATICA1> & .\AULAPRATICA1/testBerPara1ai.py
BER quando p é 10^-1 =
0.504
BER quando p é 10^-2 =
0.0792
BER quando p é 10^-3 =
0.00992
```

Figura 3 - Resultados do BER para 10^{-1} , 10^{-2} , 10^{-3}

2.1.2. Sub-alínea ii)

Para a realização desta alínea, foram realizadas as seguintes funções:

openfile(filename): Recebe o nome de um ficheiro como parâmetro e retorna o conteúdo desse ficheiro como uma string.

turnBinary(filewords): Recebe uma string de palavras e converte cada caractere na sua representação binária de 8 bits. Os caracteres são concatenados para formar uma sequência binária única.

fillUpto8bits(sequence): A função acrescenta uma sequência binária com zeros à esquerda para garantir que tenha um comprimento de 8 bits.

convertRep3_1(binaryseq): Esta função recebe uma sequência binária e converte-a para uma sequência codificada pelo código de repetição (3,1). Cada bit '1' é substituído por '111' e cada bit '0' é substituído por '000'.

simulatedBSC(seqOfBits, p): Simula um BSC onde cada bit na sequência de bits tem uma probabilidade 'p' de ser invertido. A função itera sobre cada bit na sequência e com base numa probabilidade aleatória, inverte o bit ou mantém-o intacto.

decodeRep3_1(binaryseq): Recebe uma sequência binária codificada pelo código de repetição (3,1) e realiza a descodificação, revertendo a codificação e restaurando a sequência original. Itera sobre a sequência binária, identifica os padrões de repetição e reconstrói a sequência original. Sendo que está a ser operado este código em modo correção, para cada padrão de repetição, caso seja detectado um erro, o mesmo será o corrigido, no entanto, se existirem dois erros os mesmos não poderão ser corrigidos, por exemplo, o bit '1' gera uma sequência original "111", se for recebida como "110", o bit com erro, neste caso o último, será corrigido e na descodificação irá originar o bit '1', contudo, se for recebida a sequência "010", por exemplo, a sequência tem presentes 2 erros, mas o código não tem capacidade de correção para 2 erros, o que na descodificação originará o bit '0'.

turnCharacter(binarySeq): Esta função recebe uma sequência binária e converte-a numa sequência de caracteres. Itera sobre a sequência binária, lendo conjuntos de 8 bits e converte-os de volta para os seus caracteres correspondentes usando a função chr().

De forma a calcular cada valor de BER nas diferentes probabilidades de erro, para cada caso foram realizados 3 testes e calculados os BERs de cada 1, e realizado uma média para saber qual o valor de BER para cada probabilidade de erro.

```
PS C:\isel\CD\AULAPRATICA1>
python .\testBerPara1aii.py
BER quando p é 10-1 =
0.168
BER quando p é 10-2 =
0.0016
BER quando p é 10-3 =
0.0
```

Figura 4 - Resultados do BER para 10^{-1} , 10^{-2} , 10^{-3}

2.1.3. Sub-alínea iii)

Para a realização desta alínea, foram realizadas as seguintes funções:

symbolToBinaryConversionHamming(string): Esta função recebe uma string de símbolos como parâmetro e converte cada símbolo na sua codificação binária. De seguida, aplica a codificação Hamming à representação binária, adicionando bits de paridade para a correção de erros. A função retorna a representação binária.

BinaryToSymbolConversionHamming(seqOfBits): Esta função recebe uma sequência de bits como parâmetro, que é considerada a saída da sequência binária proveniente do BSC. Executa detecção e correção de erros usando códigos Hamming e converte os bits corrigidos em símbolos. Para funcionar em modo correção, os bits de paridade/redundância são recalculados e submetidos a uma operação xor com os bits de paridade das sequências recebidas, caso o resultado da operação seja zero, então essa sequência é mantida, caso contrário, é usada a função **ErrorCorrection**, que irá calcular o padrão de erro através do síndrome dado como parâmetro, e corrigirá o erro. Esta função apenas corrige 1 bit uma vez que é essa capacidade de correção deste código. A função retorna os símbolos originais.

XorBitsToOne(BitsStr): Esta função executa um XOR bit a bit nos bits da string fornecida e retorna o resultado como um único bit.

XorBitsString(BitsStr1, BitsStr2): Esta função executa um XOR bit a bit entre duas sequências de bits com comprimento igual e retorna o resultado como uma nova sequência.

symbolToBinaryConversion(seqOfChars): Converte uma sequência de caracteres numa codificação binária usando o esquema de codificação UTF-8.

binaryToSymbolConversion(seqOfBits): Converte uma sequência de bits em caracteres decodificando a representação binária usando a codificação UTF-8.

ErrorCorrection(síndrome, SevenBitsStr): Esta função executa a correção de erros numa representação binária de 7 bits e retorna os bits corrigidos.

Além disso, foram ainda criadas algumas funções auxiliares para simular um BSC com uma probabilidade de erro especificada - **simulatedBSC()** e contar o número de 0 ou 1 numa determinada sequência - **countzerosOrOnes()**.

```
PS C:\isel\CD\AULAPRATICA1> python .\testBerPara1a1iii.py
BER quando p é 10^-1 =
0.288
BER quando p é 10^-2 =
0.0056
BER quando p é 10^-3 =
8e-05
```

Figura 5 - Resultados do BER para 10^{-1} , 10^{-2} , 10^{-3}

2.2. Alínea b)

Nesta alínea, para todas cada sub-alínea da alínea a), antes da sequência ser convertida para código binário, é utilizada a função **interleaving()**, função que é responsável pela minimização de erros consecutivos numa sequência, para tal, esta função irá receber uma sequência binária e organizá-la numa matriz, linha por linha, fazendo com que os bits sejam então retornados coluna a coluna dessa matriz apresentando uma disposição completamente distinta da original, minimizando assim a probabilidade de erros consecutivos.

Após a codificação da sequência, passagem pelo **canal BSC**, e decodificação da sequência, a mesma é novamente submetida ao método de interleaving de forma a reverter a sequência anteriormente “baralhada”, para a sequência original.

Para teste desta implementação, encontra-se aqui apenas um exemplo das alíneas anteriores, sendo ele o 1. a) ii), que corresponde ao Código de Repetição (3,1). Neste teste, apenas é utilizado o ficheiro “test.txt” com a sequência “aaabbc” para demonstração do funcionamento com interleaving.

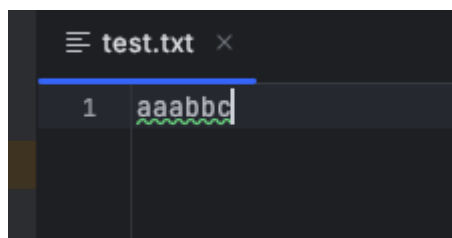


Figura 6 - Conteúdo do ficheiro “test.txt”

```
In order to write the file's name, use "_filename_.txt"
File name: test.txt
['aaabbc']
Interleaving: aaabbc
Binary Sequence: 110000111000011100001011000101100011
Encoded Sequence: 1111110000000000001111111100000000000011111111000000000011100011111000000000111000111110000000000111111
BSC: 11111100000100111111100011110000000011111001110010000010011111110000010001011111110000100000011111111100000000000011110
Decoded Sequence: 110001101000110100011100001110000111000011
De-Interleaving: 1
```

Figura 7 - Output da aplicação da técnica de interleaving

3. Exercício 2

3.1. Alínea a)

Para a resolução desta alínea foi necessária a realização de dois ficheiros: um ficheiro em Arduino (utilizando o Arduino IDE) e um ficheiro em Python.

- Ficheiro Arduino

A função `Nth_of_GP` recebe três parâmetros: a , que representa o primeiro termo da PG; r , que representa a razão da PG e n , que indica o número do termo que desejamos calcular.

Dentro da função, é feito o cálculo do N-ésimo termo da progressão geométrica utilizando a fórmula: $pg = a * (int)(pow(r, n - 1))$, onde `pow` é uma função da biblioteca `math.h`. Posto isto, o valor calculado é escrito no ecrã usando a função `Serial.println(pg)`.

Na secção `setup`, a função `Serial.begin(9600)` é chamada para iniciar a comunicação serial com a velocidade de transmissão de 9600 bits por segundo. Essa configuração é necessária para que seja possível visualizar a saída do programa no Monitor Serial.

Na secção `loop`, são definidos os valores dos parâmetros a , r e n para 1, 2 e 8 respectivamente. De seguida, é declarado um `for` que por cada iteração, irá chamar a função `Nth_of_GP` passando como o valor de a e r anteriormente declarados, para o parâmetro n é passado i que corresponde ao n da progressão geométrica. Este i será incrementado em cada iteração de forma a construir a progressão. Por fim, depois do cálculo e escrita do N-ésimo termo da progressão geométrica, é feita uma pausa de 1 segundo (1000 milissegundos) usando a função `delay(1000)`, antes do loop ser repetido.

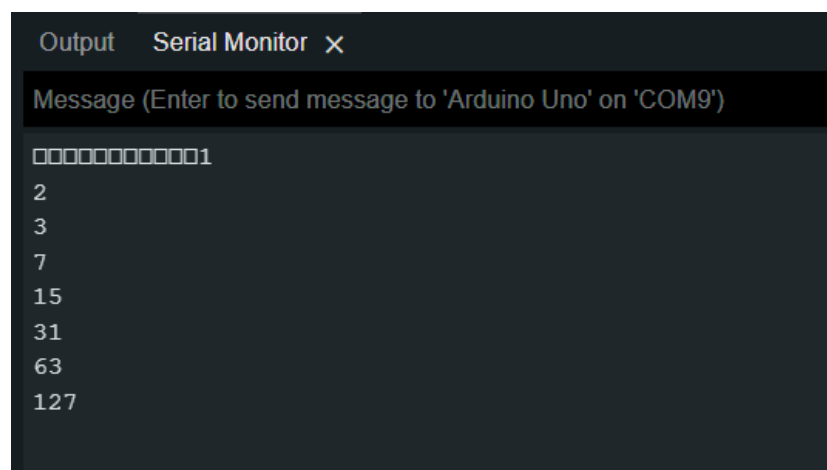


Figura 8 - Resultados da progressão geométrica no Arduino IDE

- Ficheiro Python

Inicia-se pela definição de duas variáveis, `port` e `baudrate`, que correspondem respetivamente à porta da entrada USB que estará conectada ao Arduino e à taxa de transmissão segundo a qual os dados serão enviados.

De seguida, é utilizada uma função da biblioteca `pyserial` que será responsável pela criação de um objeto de comunicação em modo serial, que irá comunicar com a placa Arduino.

Por fim, é implementado um loop `while 1`, que corresponde a um `while(true)`, onde realizará a leitura dos dados provenientes da comunicação serial e realiza a impressão desses resultados, caso os dados

recebidos correspondam a `b'\x00'`, que indica que estão a ser transmitidos dados nulos, o loop irá ser então interrompido e a comunicação em série será fechada.

```
[Done] exited with code=1 in 0.243 seconds

[Running] python -u "c:\Users\carol\OneDrive\Ambiente de Trabalho\Faculdade\uni-projects-repo\CD\M2\exc2_a.py"
1
2
3
7
15
31
63
127
```

Figura 9 - Resultados da progressão geométrica recebido após correr o ficheiro Python

3.2. Alínea b)

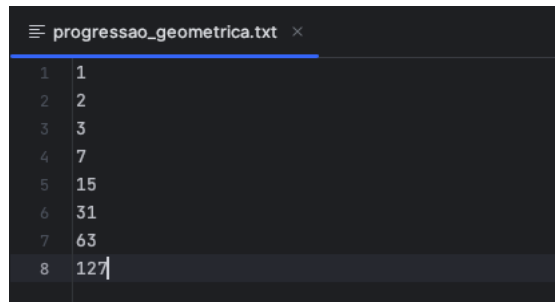
Na implementação da presente alínea, foi utilizado um ficheiro com a progressão geométrica obtida na alínea anterior. Do conteúdo presente neste ficheiro, é submetido à função `detect_error`, que recebe como parâmetro dados (*data*) e a posição (*error_index*) onde se pretende forçar um bit errado.

Antes de o erro ser forçado, é calculado o checksum esperado, ou seja, é uma soma de verificação que pode ser usada para detectar erros de transmissão ou corrupção de dados. De forma a realizar o cálculo, é utilizada a função `fletcher_checksum`, que recebe como parâmetro os dados dos quais se pretende calcular o checksum. Nesta função o somador *sum1* é incrementado com o valor atual e realiza a operação de módulo 255 para garantir que o resultado esteja dentro do intervalo válido (de 0 a 255).

Em seguida, o somador *sum2* é incrementado com o novo valor de *sum1*, sobre o qual também se realiza uma operação de módulo 255. Depois que todos os valores em *data* forem processados, o checksum final é calculado combinando os valores de *sum2* e *sum1*. O valor de *sum2* é deslocado 8 bits para a esquerda (`<< 8`) e o valor de *sum1* é combinado utilizando a operação lógica **OR**. O resultado de checksum, por fim, é retornado.

Após o cálculo, é então forçado o bit de erro, novamente é calculado o valor de checksum de forma a ser possível a comparação entre o valor esperado e o recebido. Essa comparação é realizada através da operação lógica **xor** entre o valor esperado e o recebido, caso o valor do resultado seja diferente de 0, então é discriminada a posição onde o erro está presente, caso contrário, indica que não existem erros.

De forma a testar a implementação, o ficheiro gerado da conexão com o Arduino da alínea a) é utilizado, e é forçado um bit a erro na posição 2.



	progressao_geometrica.txt
1	1
2	2
3	3
4	7
5	15
6	31
7	63
8	127

Figura 10 - Conteúdo do ficheiro “progressao_geometrica.txt”
(dados gerados pelo Arduino)

```
Expected checksum = 58105
Received checksum = 59642
Erro detectado na posição 2.
```

Figura 11 - Output com valor de checksum esperado, valor de checksum recebido e a posição onde se encontra o bit de erro

3.3. Alínea c)

Na implementação desta alínea utilizou-se as funcionalidades da alínea anterior, tanto a função `detect_error` como a função `fletcher_checksum`, no entanto esta última sofreu algumas alterações.

Nesta alínea, a função `fletcher_checksum` passa a receber como parâmetro, além dos dados e o index a partir do qual se pretende inserir erro, o número de bits consecutivos se pretende inserir erros. A função mantém todas as funcionalidades que tinha anteriormente, alterando apenas que em vez de forçar erro apenas numa posição irá forçar no número que é passado como parâmetro. Para tal, criou-se um loop while, no qual a variável *i*, que é responsável por iterar sobre os dados, irá forçar erro na posição apresentada no valor de *i*.

Para testar esta implementação, o ficheiro da alínea anterior foi também aqui utilizado, e colocou-se o erro a começar na posição 2 e a acabar na posição 4, ou seja, 3 bits de erro.

```
Expected checksum = 58105
Received checksum = 61948
Erro detectado da posição 2 até à posição 4.
```

Figura 12 - Output com valor de checksum esperado, valor de checksum recebido e a posição de início e fim do erro inserido

4. Conclusão

Com este trabalho foi possível aplicar conhecimentos teóricos lecionados nas aulas de forma a perceber o seu contexto na resolução de problemas. Com isto, foi também necessário aplicar conhecimentos de linguagem C e Python uma vez que ainda não tinham sido trabalhados ao longo do curso.