Homework # 1

Due Tuesday April 17, 2018, at 2:30 PM
Collaboration and discussions are not allowed on Problem 1 and are allowed and encouraged on Problem 2

1. **Mutations and Duplications**(*Collaboration is not allowed on this problem*)
   Motivated by DNA mutations, we consider strings of symbols and study *point mutations* and *tandem duplication* processes. A point mutation is a change in a single symbol of a given string. For example, for the binary string 0011101, a *point mutation* in the fourth location results in 0010101. A *tandem duplication* of length k, is a duplication of a substring of length k next to its original position. For example, a *tandem duplication* of length 3 starting in the second location of 0011101 results in 0011011101.

   In class we proved that every binary string can be generated by tandem duplication steps from one of the following six strings $\{0, 1, 01, 10, 010, 101\}$ (that we call *seeds*). The generation process from the seeds is not unique. For example, consider the string 001001. It can be generated from 01 in two ways: $01 \to 0101 \to 00101 \to 001001$, requiring 3 tandem duplication steps, and $01 \to 001 \to 001001$, requiring 2 tandem duplication steps.

   The *duplication distance* is defined as the minimum number of steps required to generate a given string from a seed by a tandem duplication process. In the example above, the duplication distance of 001001 is 2.

   (a) Find the duplication distance of the following strings. Show the steps leading from the seed to the string.

      i. 00010101
      ii. 0001100001
      iii. 0001010111
      iv. 0100000101011
      v. 0010001100100111
      vi. 0110100110010110

(b) Here we allow point mutations and define the *mutation distance* as the minimum number of the sum of tandem duplications steps and point mutations steps required to generate a string from a given seed. For example, the duplication distance of 111 is 2 (from the seed 1), while the mutation distance is 1 (from the seed 101).

Find the mutation distance of the following strings from the seed 01. Show the steps leading from the seed to the string.

i. 0100111
ii. 00010101
iii. 0001100001
iv. 0001010111
v. 1010111000

(c) A duplication of a substring of length k next to its original position creates a *tandem repeat*. Here we consider the reverse process. Namely, given a string with *tandem repeats*, in each step we remove one copy of a repeat (in some order) until the string does not have any tandem repeats. We call a string with no tandem repeats a *squarefree string*. For example:

$$0100\underline{00}101 \to 010\underline{0}101 \to 0101\underline{01} \to 0\underline{1}01 \to 01$$

The removal of a copy of a tandem repeat is called *deduplication* and the final squarefree string obtained is the *seed* of x. Hence, the seed of 010000101 is 01. In class we proved that every binary string has a *unique seed* and it belongs to the set $\{0, 1, 01, 10, 010, 101\}$. However, for strings over ternary (or higher) alphabet, there can be multiple seeds. For example the ternary string $y = 012101212$, has two seeds:

$$0121\underline{01}2\underline{1}2 \to 012\underline{1}2 \to 012 = Seed\ 1$$

$$012101\underline{21}2 \to 0121012 = Seed\ 2$$

Find all possible deduplication seed(s) of the following strings. Show your work.

i. 30121013012101212
ii. 123032123032303
iii. 120312031230321230323031212

2. **Eternal Memory with DNA** (*Collaboration is allowed on this problem*)
We are at an interesting time in the history of the human race; our life is dominated by information production and exchange that is global, practically instantaneous and has an infinite volume. A challenging aspect in this new reality is the retention of information for a very long time - practically eternity. A simplistic implementation for eternal memory includes, for example, storage (programming) of information as part of a DNA of bacteria, and the survival of the information based on the survival of the genomic material in the decedents of the programmed bacteria. However, the natural evolution of the programmed bacteria might distort the genomic material and affect the correctness of the stored information. In this problem, you will learn about simple schemes for error correction. For the sake of simplicity, information will be represented by a binary alphabet. A useful operation for implementing error correction is the parity function on binary variables (bits), also known as the eXclusive OR (XOR) function, defined as follows:

Consider the n binary variables, $x_i \in \{0,1\}$, $1 \leq i \leq n$, let $X = (x_1, x_2, \cdots, x_n)$ then

$$XOR(X) = \begin{cases} 1 & \text{if the number of 1's in } X \text{ is odd} \\ 0 & \text{otherwise} \end{cases}$$

We use the following notation, $XOR(x_1, x_2) = x_1 \oplus x_2$. For example, $1 \oplus 1 \oplus 0 = 0$ and $0 \oplus 1 \oplus 0 = 1$.

Assume that you would like to 'eternally' remember the following binary matrix.

$$A = \begin{array}{|c|c|c|c|} \hline 0 & 1 & 1 & 1 \\ \hline 0 & 1 & 1 & 0 \\ \hline 1 & 1 & 1 & 1 \\ \hline 1 & 0 & 1 & 1 \\ \hline \end{array}$$

However, when you perform your experiment using your DNA storage in bacteria, you discover that some of the information is not readable, and while you stored A you read $\hat{A}$. The Es represent the unreadable (erased) spaces, called erasures.

$$\hat{A} = \begin{array}{|c|c|c|c|} \hline 0 & 1 & 1 & E \\ \hline 0 & E & 1 & E \\ \hline 1 & 1 & 1 & E \\ \hline 1 & E & 1 & 1 \\ \hline \end{array}$$

To combat those erasures you decide to add a fifth column (called a parity column) that contains the row parity, as follows:

a_1	b_1	c_1	d_1	$a_1 \oplus b_1 \oplus c_1 \oplus d_1$
a_2	b_2	c_2	d_2	$a_2 \oplus b_2 \oplus c_2 \oplus d_2$
a_3	b_3	c_3	d_3	$a_3 \oplus b_3 \oplus c_3 \oplus d_3$
a_4	b_4	c_4	d_4	$a_4 \oplus b_4 \oplus c_4 \oplus d_4$

(a) Show the matrix A with the additional parity column. Explain how to recover the erased bits in $\hat{A}$ using the parity column. Can you recover all of them?

(b) Assume that you have an $n \times n$ matrix with an additional parity column. We define an *erasure pattern* as a subset of entries in the $n \times (n+1)$ matrix. What are the erasure patterns that can be corrected in an $n \times n$ matrix with an additional parity column?

(c) To improve your erasure correcting scheme you decide to add both a parity column and a parity row, as follows:

a_1	b_1	c_1	d_1	$a_1 \oplus b_1 \oplus c_1 \oplus d_1$
a_2	b_2	c_2	d_2	$a_2 \oplus b_2 \oplus c_2 \oplus d_2$
a_3	b_3	c_3	d_3	$a_3 \oplus b_3 \oplus c_3 \oplus d_3$
a_4	b_4	c_4	d_4	$a_4 \oplus b_4 \oplus c_4 \oplus d_4$
$a_1 \oplus a_2 \oplus a_3 \oplus a_4$	$b_1 \oplus b_2 \oplus b_3 \oplus b_4$	$c_1 \oplus c_2 \oplus c_3 \oplus c_4$	$d_1 \oplus d_2 \oplus d_3 \oplus d_4$	

Show the matrix A with the additional parity column and parity row. Explain how to recover the erased bits in $\hat{A}$. Can you recover all of them?

(d) Assume that you have an $n \times n$ matrix with additional parity column and parity row. We define an *erasure pattern* as a subset of entries in the $(n+1) \times (n+1)$ matrix. What are the erasure patterns that can be corrected in an $n \times n$ matrix with additional parity column and parity row.

(e) **Errors.** So far we studied erasures, where an erasure results from a failure to read from a specific location. In contrast, an error is a result of a successful read of an erroneous information. For example, if you store the matrix A and have an error in the last bit in the first row you will read:

$$\tilde{A} = \begin{array}{|c|c|c|c|} \hline 0 & 1 & 1 & 0 \\ \hline 0 & 1 & 1 & 0 \\ \hline 1 & 1 & 1 & 1 \\ \hline 1 & 0 & 1 & 1 \\ \hline \end{array}$$

Assume that you stored the matrix A with an additional parity column and parity row. Can you correct the error in $\tilde{A}$? What are the correctable error patterns that can be corrected in an $n \times n$ matrix with additional parity column and parity row?

(f) **Deletions.** Another kind of mutation error is a *deletion* error. For example, the binary string 0101 can mutate to 101. Namely, the first bit was deleted. A single-deletion-correcting code is a set of strings (codewords) that when used to store information can recover the information even if there is a single deletion. For example $\{00, 01\}$ is not a single-deletion-correcting code, because when 0 is read, it is not clear which one of the codewords was written. However, $\{00, 11\}$

is a single-deletion-correcting code. The *length of a codeword* is the number of bits in the codeword. Here we consider codes that have codewords of identical length, we call it the *length of the code.* For example, the code $\{101, 111\}$ has length 3.

i. Is $\{101, 111\}$ a single-deletion-correcting code? Prove your claim.
Is $\{000, 101\}$ a single-deletion-correcting code? Prove your claim.
Is there a length 3 code with more than 2 codewords? Prove your claim.

ii. Is $\{0000, 0110, 1010\}$ a single-deletion-correcting code? Prove your claim.
Prove or disprove, there is a single-deletion-correcting code of length 4 with 4 codewords.

iii. Is $\{00000, 10001, 01010, 11100, 00111\}$ a single-deletion-correcting code? Prove your claim.
Prove or disprove, there is a single-deletion-correcting code of length 5 with 6 codewords.

(g) **Insertions.** Another kind of mutation error is an *insertion* error. For example, the binary string 0101 can mutate to 01011. Namely, the last bit was inserted. A single-insertion-correcting code is a set of strings (codewords) that when used to store information can recover the information even if there is a single insertion. For example $\{00, 01\}$ is not a single-insertion-correcting code, because when 010 is read, it is not clear which one of the codewords was written, it can be either 00, with a 1 inserted in the middle, or 01 with a 0 inserted at the end. However, $\{00, 11\}$ is a single-insertion-correcting code.

i. Is $\{101, 111\}$ a single-insertion-correcting code? Prove your claim.
Is $\{000, 101\}$ a single-insertion-correcting code? Prove your claim.

ii. Prove that a code is a single-insertion-correcting code *if and only if* it is a single-deletion-correcting code.