ASSEMBLY LANGUAGE PROJECTS 2016

Contents

Common Requirements 2
1. Sorting Algorithms 3
2. Text Encryption 4
3. Image \ Text Steganography 8
4. Minesweeper 13
5. Sudoku 15
6. Text Editor 18
7. Motif Finding Problem 20

Common Requirements

- Your program must be divided into PROCs, each of which is responsible for one and only one functionality. For example:
  - ReadArray: reads an array
  - MergeSort: sort a given array by using merge sort
  - ViewImage: views an image
- Your PROCs should be transparent for input parameters, do not forget the USES operator.
- Don't use any hardcoded values, instead use constants and operators
  - For example: Fetch array length by LENGTHOF operator … *etc*.
- Don't use the processor directives such as IF, IFE, *etc*.
- Bonus items will not be counted unless the original project is complete.
- If the GUI is necessary at your project; **here** you are a link of a simple tutorial for how to link Assembly code with high-level language.
- Keep your code clean, follow a specific convention, and choose meaningful identifiers (variables and procedures names).
- All projects must be submitted with a printed documentation. In your documentation, draw the flow chart of your programming logic and the procedures hierarchy (refer to chapter 5).
- [Code like a Pro]: Your code must be well documented; you should use Irvine documentation style.

```
;-----------------------------------------------
;Calculates: Sum of an integer array
;Receives: ESI Contains the offset of the Array
;   ECX Contains the length of the Array
;Returns:  EAX contains Array Sum
;----------------------------------------------
SumArr PROC
```

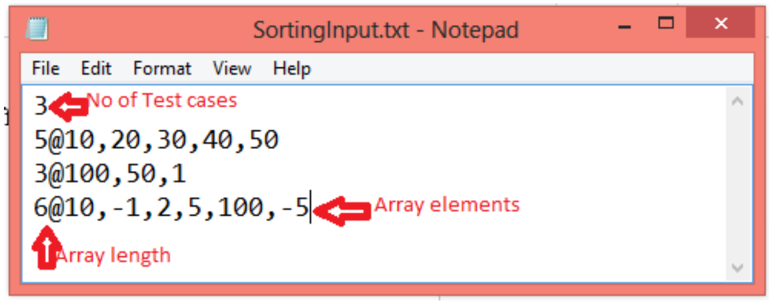
1. Sorting Algorithms

Write an assembly program that sorts an array of integers by three different algorithms (Merge, Insertion and Count sort).

- **Details**
  1. Algorithms required
     - Merge sort (Using Divide & Conquer algorithm).
     - Insertion sort.
     - Selection sort.
     - Count sort.
  2. Array maximum length is 100.
  3. Array will contain positive and negative values.

- **Input**

  You will be given a .txt file starts with the number of test cases, followed by several lines, each of which contains the array length followed the array elements.



```
3
5@10,20,30,40,50
3@100,50,1
6@10,-1,2,5,100,-5
```

- **Output**
  1. The array after sorting
  2. Time taken for each sorting algorithm,

- **Bonus**
  1. Bar chart graphical representation for time taken.
  2. Animation for the sorting algorithms

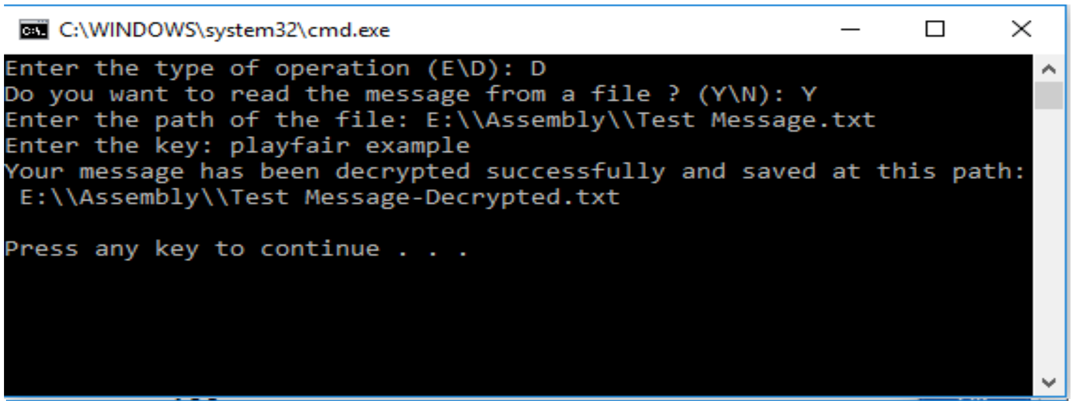
2. Text Encryption

Write an assembly program that allows the user to encrypt and decrypt a secret message using the **Playfair** cipher technique.

- **Details**

The original message consists of maximum 1000 characters.

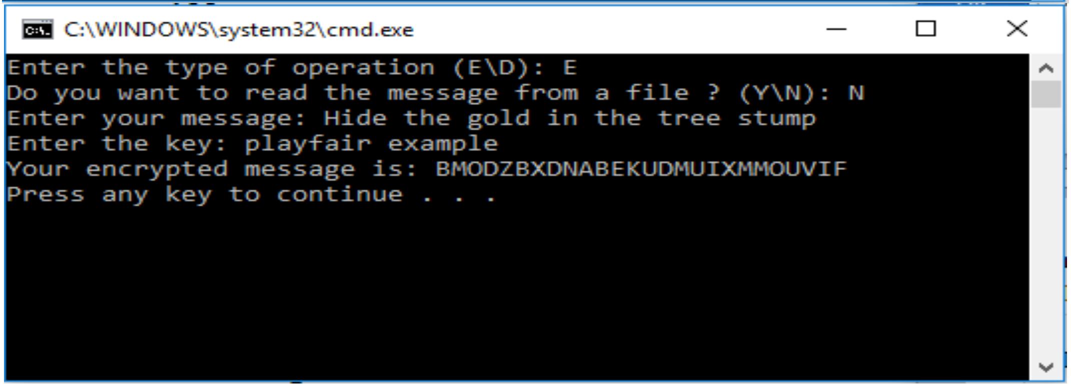
- **Input**
  1. You will take the message directly from the user or read it from a file
  2. The key will be taken from the user
  3. You should take from the user the type of operation he wants to do whether to encrypt or decrypt



C:\WINDOWS\system32\cmd.exe

```
Enter the type of operation (E\D): D
Do you want to read the message from a file ? (Y\N): Y
Enter the path of the file: E:\\Assembly\\Test Message.txt
Enter the key: playfair example
Your message has been decrypted successfully and saved at this path:
 E:\\Assembly\\Test Message-Decrypted.txt

Press any key to continue . . .
```



C:\WINDOWS\system32\cmd.exe

```
Enter the type of operation (E\D): E
Do you want to read the message from a file ? (Y\N): N
Enter your message: Hide the gold in the tree stump
Enter the key: playfair example
Your encrypted message is: BMODZBXDNABEKUDMUIXMMOUVIF
Press any key to continue . . .
```

- **Output**
  1. In case of taking the message from the user, the encrypted/decrypted message should be displayed on the console screen.
  2. In case of reading the message from a file you should create another file containing the encrypted/decrypted message and save it at the same path as the original one with adding the word "encrypted" or "decrypted" to its title.

- **Bonus**
  1. Implement a GUI for the project to take the input from the user or read it from a file and save the result to another file.

- **Description of the PlayFair Technique**
  The Playfair cipher uses a 5 by 5 table containing a key word or phrase. Memorization of the keyword and simple rules was all that was required to create the 5 by 5 table and use the cipher. To generate the key table, one would first fill in the spaces in the table with the letters of the keyword (dropping any duplicate letters), then fill the remaining spaces with the rest of the letters of the alphabet in order (put both "I" and "J" in the same space). The keyword will be written in the top rows of the table, from left to right. To encrypt a message, one would break the message into diagrams (groups of 2 letters) such that, for example, "HelloWorld" becomes "HE LL OW OR LD", and map them out on the key table. *If needed, append an uncommon monogram to complete the final diagram*. The two letters of the diagram are considered as the opposite corners of a rectangle in the key table. Note the relative position of the corners of this rectangle. Then apply the following rules, in order, to each pair of letters in the plaintext:
  - If both letters are the same (or only one letter is left), add an "X" after the first letter. Encrypt the new pair and continue. Some variants of Playfair use "Q" instead of "X", but any letter, itself uncommon as a repeated pair, will do.
  - If the letters appear on the same row of your table, replace them with the letters to their immediate right respectively (wrapping around to the left side of the row if a letter in the original pair was on the right side of the row).
  - If the letters appear on the same column of your table, replace them with the letters immediately below respectively (wrapping around to the top side of the column if a letter in the original pair was on the bottom side of the column).
  - If the letters are not on the same row or column, replace them with the letters on the same row respectively but at the other pair of corners of the rectangle defined by the original pair. The order is important – the first letter of the encrypted pair is the one that lies on the same **row** as the first letter of the plaintext pair.

Assume one wants to encrypt the diagram "***OR***". There are five general cases:

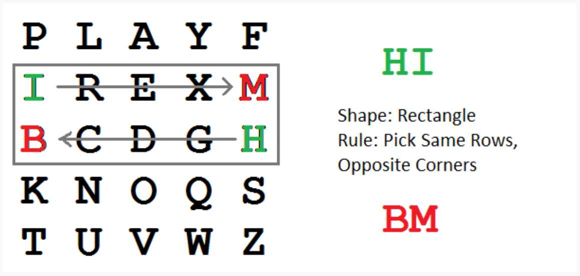
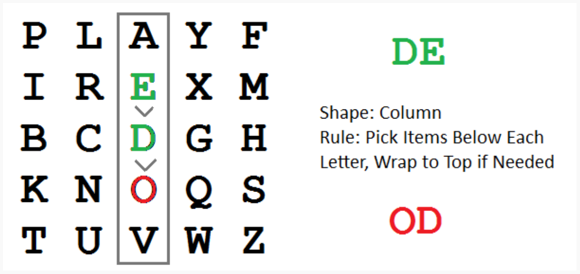
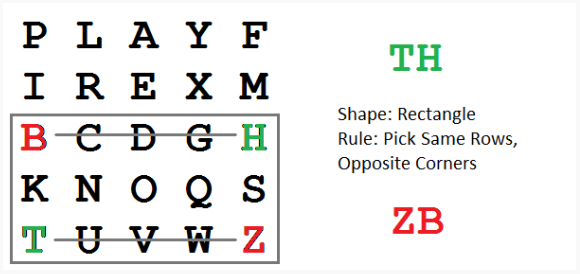
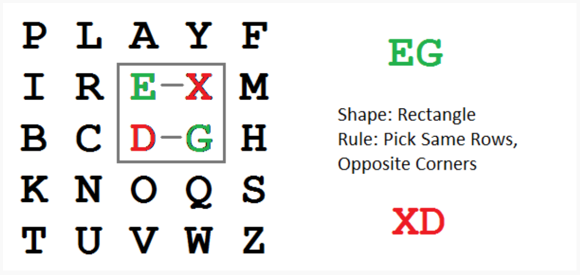
1)	2)	3)	4)	5)
* * * * *	* * O * *	Z * * O *	* * * * *	* * * * *
* O Y R Z	* * B * *	* * * * *	* * * * *	* * R * *
* * * * *	* * * * *	* * * * *	* O R C *	* * O * *
* * * * *	* * R * *	R * * X *	* * * * *	* * I * *
* * * * *	* * Y * *	* * * * *	* * * * *	* * * * *
Hence, OR → YZ	Hence, OR → BY	Hence, OR → ZX	Hence, OR → RC	Hence, OR → IO

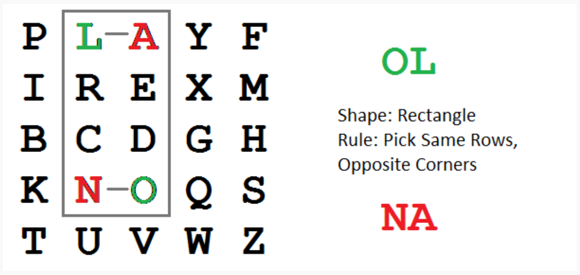
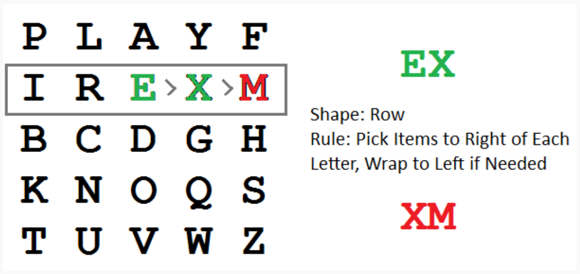
Example:
Using "**playfair example**" as the key, the key table becomes (omitted letters in red):

P L A Y F A
I R E X A M PLE A
B C D EF G H I=J
K LM N O P Q R S
T U V W XY Z

Encrypting the message "Hide the gold in the tree stump" **(note the null "X" used to separate the repeated "E"s):**

HI DE TH EG OL DI NT HE TR EX ES TU MP

1. The pair HI forms a rectangle, replace it with BM	 HI Shape: Rectangle Rule: Pick Same Rows, Opposite Corners BM
2. The pair DE is in a column, replace it with OD	 DE Shape: Column Rule: Pick Items Below Each Letter, Wrap to Top if Needed OD
3. The pair TH forms a rectangle, replace it with ZB	 TH Shape: Rectangle Rule: Pick Same Rows, Opposite Corners ZB
4. The pair EG forms a rectangle, replace it with XD	 EG Shape: Rectangle Rule: Pick Same Rows, Opposite Corners XD

5. The pair OL forms a rectangle, replace it with NA	 P L A Y F I R E X M B C D G H K N O Q S T U V W Z OL Shape: Rectangle Rule: Pick Same Rows, Opposite Corners NA
6. The pair DI forms a rectangle, replace it with BE	
7. The pair NT forms a rectangle, replace it with KU	
8. The pair HE forms a rectangle, replace it with DM	
9. The pair TR forms a rectangle, replace it with UI	
10. The pair EX (X inserted to split EE) is in a row, replace it with XM	 P L A Y F I R E X M B C D G H K N O Q S T U V W Z EX Shape: Row Rule: Pick Items to Right of Each Letter, Wrap to Left if Needed XM
11. The pair ES forms a rectangle, replace it with MO	
12. The pair TU is in a row, replace it with UV	
13. The pair MP forms a rectangle, replace it with IF	

Encrypted message is:

```
BM OD ZB XD NA BE KU DM UI XM MO UV IF
```

Thus the message "Hide the gold in the tree stump" becomes "BMODZBXDNABEKUDMUIXMMOUVIF".

- **To decrypt, use the INVERSE (opposite) of the last 3 rules, and the 1st as-is.**

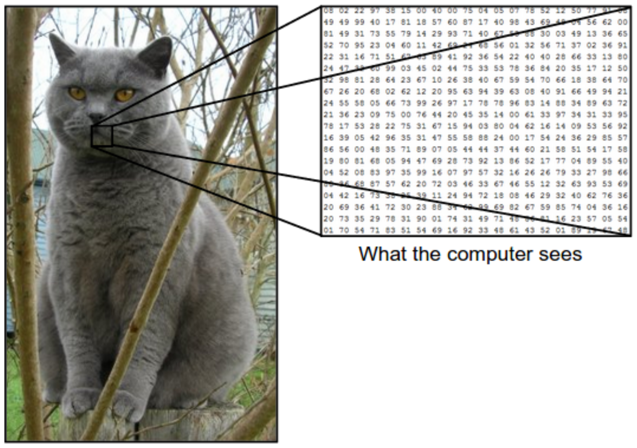
3. Image \ Text Steganography

- **Introduction**
  Steganography is the science of hiding a file, message, image or video within another file, message, image or video. So, for example Alice can send a secret message to Bob that no one else can view:

Alice
Message
Bob

Steganography works by replacing bits of useless or unused data in regular computer files with bits of different, invisible information. This hidden information can be plain text, cipher text, or even images. In our case, our data will be the plain text that we need to hide, and the unused data is the least significant bits (LSBs) in the image pixels.
As mentioned before given an image and a text message, we want to hide this message in the given image by manipulating LSBs of the pixels.

- **What is a computer image?**
  A computer image is a digitized version of a picture taken with a capturing device like camera or it is the electronic visual representation of pictures which is stored in computer. As we all know that the computer understands 1's and 0's only, so everything you see on the computer screen is mapped to binary number(s) and of course a computer image is mapped to some binary numbers. A computer image stored as a 2D array of pixels, each pixel consists of three color channels Red, Green and Blue (RGB) each color channel takes a value between 0 – 255 inclusive. Mixing decimal values of these color channels gives us the different color that we see on the screen.



A portion of an image showing what the computer sees

- **How to?**
  Given a text message as an input and a **NON-Transparent** image, you should output an image exact the same as the input image (visually) after hiding the secret message within it. You should also be able to retrieve the secret message after hiding it then display it.
  Let's begin with hiding a single character. A character is represented in 8 binary bits as we all know, so we are going to decompose this character into its binary bits, mix them with some pixels then we got a character hidden in the image!

To hide a group of characters here is the algorithm:

1. Starting from pixel (0, 0), Loop through the pixels of the image. In each iteration, get the RGB values separated.
2. For some groups of R, G, and B of the pixels, we are going to modify their LSB. These bits will be used in hiding characters.
3. Get the current character and convert it to integer. Then hide its 8 bits in R1, G1, B1, R2, G2, B2, R3, G3, where the numbers refer to the numbers of the pixels. In each LSB of these elements (from R1 to G3), hide the bits of the character consecutively.
4. When the 8 bits of the character are processed, jump to the next character, and repeat the process until the whole text is processed.
5. The text can be hidden in a small part of the image according to the length of that text. So, there must be something to indicate that here we reached the end of the text. The indicator is simply 8 consecutive zeros or you can write the size of the message at the beginning of the image. **This will be needed when extracting the text from the image.**

- **Example**

If we want to hide 'A' character into an image here are the steps in action for the first three iterations starting from Pixel (0, 0):

A = 65d	0100 0001
Pixel (0, 0) Channel (R) = 119d	0111 0111
Result	0111 0110

Iteration #0: store 8th bit of 'A' in LSB of Pixel (0) channel (R)

A = 65d	0100 0001
Pixel (0, 0) Channel (G) = 125d	0111 1101
Result	0111 1101

Iteration #1: store 7th bit of 'A' in LSB of Pixel (0, 0) channel (G)

A = 65d	0100 0001
Pixel (0, 0) Channel (G) = 255d	1111 1111
Result	1111 1110

Iteration #2: store 6th bit of 'A' in LSB of Pixel (0, 0) channel (B)

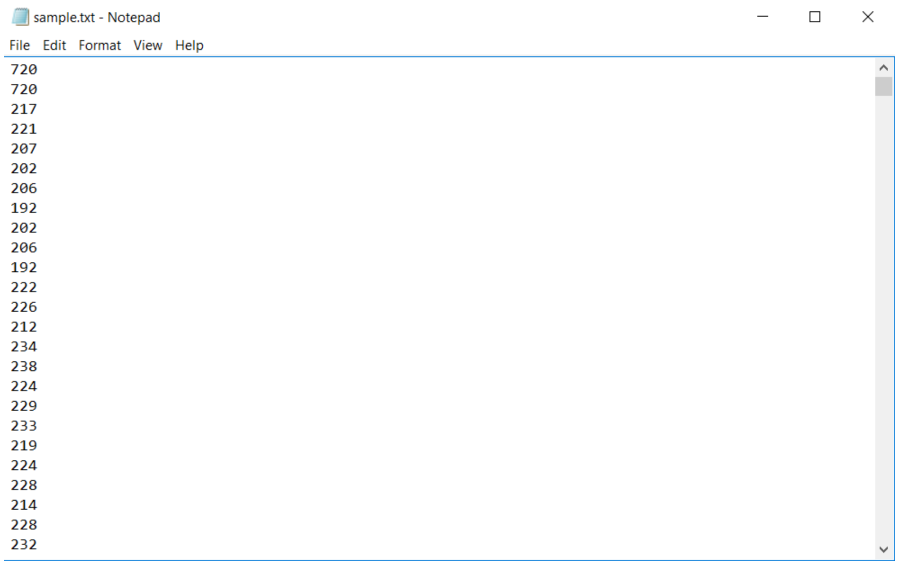
And so on until all the character's bits are hidden in pixels. As we have 8 bits for each character, we need three pixels to hide a single character (3 pixels * 3 color channels = 9).

- **Why LSB?**

Because by changing the least significant bit we will keep the color of the original pixel from changing visually, then no one can spot the difference between the modified pixel and the original one by the naked eyes. So, by changing the least significant bit we are only changing the least effective bit, so the pixel value after modification will either incremented or decremented by one. Very little change isn't it?

- **What to do?**

Unfortunately, Irvine library doesn't support images, so we have to find a method to read pixels' values from the image. We will deal with image's pixels as decimal values dumped in a text file, then we will deal with the pixels using file operations (read & write). Here is a sample for an image's pixels dumped into a text file



```
720
720
217
221
207
202
206
192
202
206
192
222
226
212
234
238
224
229
233
219
224
228
214
228
232
```

First two numbers represent the width and height of the image

You are required to deliver two modules:

1. **Image Viewer**

The first module to deliver an image viewer which is a GUI desktop application written in any high level language that you feel comfortable with, which has the following functionalities:

   a. Open bitmap image from the disk.
   b. Read a text file containing pixels' values of an image and convert it to bitmap then view it.
   c. Save loaded image as a bitmap.
   d. Save loaded image as a text file.

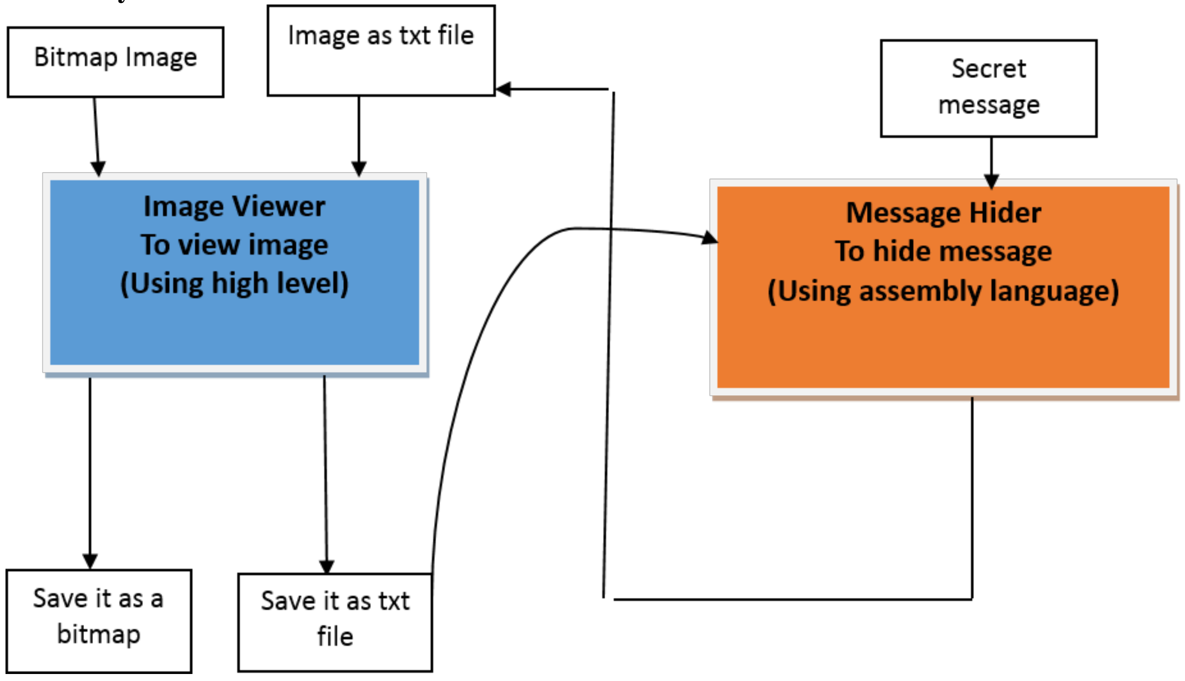
You are completely free to change the structure of the output text file according to your needs (pixels' separator, image's width and height position, etc.).
Here is a link to the sample desktop application. You can use it as your reference when you are building your own image viewer, it contains all the required functionalities.

2. **Text Hiding Application**

Write an assembly program to parse the text file generated by your image viewer and use it to hide secret messages inside the image exactly like the presented algorithm above.
Maximum message size is 1000 characters.
You should prompt the user to select from two choices:

   a. Retrieve hidden message
      - You should ask the user to enter the path of image txt file.
      - Retrieve the secret message using LSB method.
   b. Hide given message
      - You should ask the user to enter the path of image txt file.
      - Then ask him to enter the secret message.
      - Hide it using LSB method and update the image txt file.
      - Open the image txt file with your image viewer to see the image after hiding the message.

- **System Architecture**



- **Bonus**
  1- Encrypt the message before hiding it using a simple encryption algorithm with a key such as XORing, *etc*. but not the playfair algorithm.
  2- Make the whole system using **ASSEMBLY LANGUAGE ONLY** (search for other libraries).

- **References and Important Chapters**
  1- MS-Windows Programming Chapter 11: File I/O, Reading and Writing from files.
  2- Bit Masking.

4. Minesweeper

Minesweeper is a single-player game. The objective of the game is to clear a square board containing hidden "mines" without detonating any of them, with help from clues about the number of neighboring mines in each field.

The player is initially presented with a grid of undifferentiated squares. Some **randomly** selected squares, unknown to the player, are designated to contain mines you have to save their locations in file. The size of the grid is 9 X 9 and the number of mines is 15.

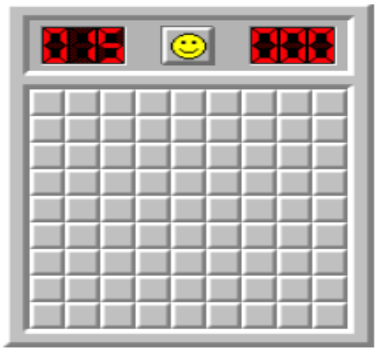
The game is played by revealing squares of the grid by letting user choose location of Chosen Square. If a square containing a mine is revealed, the player loses the game and board should reveal all mines. If no mine is revealed, a digit is instead displayed in the square, indicating how many adjacent squares (8 neighboring cells) contain mines; if no mines are adjacent, the square becomes blank, and all adjacent squares will be revealed. The player uses this information to deduce the contents of other squares, and may either safely reveal each square or mark the square as containing a mine.

The game is won when all mine-free squares are revealed, because all mines have been located.

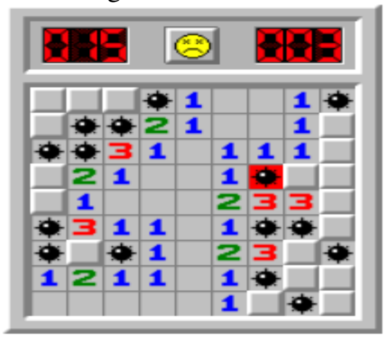
- **Input**
  The user will click on a chosen square in the GUI and sends the (x, y) point to the assembly code, where x is the row and y is the column of the chosen square.

- **Output**
  Update the board according the rules of the game.

- **Design and Regularities**
  1. All requirements should be implemented in assembly except the GUI part.
  2. You should draw your board similar to the following:



  3. You should reveal all mines when game is over like following:



4. You should have a button for starting a new game (new generation of board).
5. You should use struct for saving chosen position coordinates.
6. You should display time user took for playing.
7. You should display number of generated mines.
8. You should save the mine indices in file.
9. You should play sound (PC sound) when game is over.

- **Bonus**

1. Apply square revealing recursively
   - Initialize a list.
   - If Current Square is non-mine uncover it and add to list, otherwise game-over.
   - Count mines adjacent to it.
   - If adjacent mine count is zero, add any adjacent covered squares to list and uncover them.
   - Go to step 2 if it's not the last element on list, otherwise finish.

2. Levels (let user choose level beginner uses board 9X9, intermediate uses board 16X16 and number of mines are 40 and Expert uses board 16X30 and number of mines are 99)

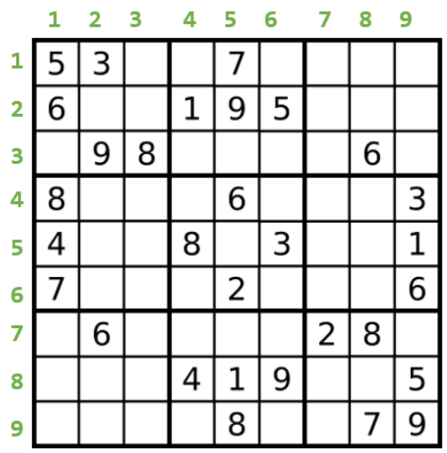
- **References**

1. Check the game online link or link.

5. Sudoku

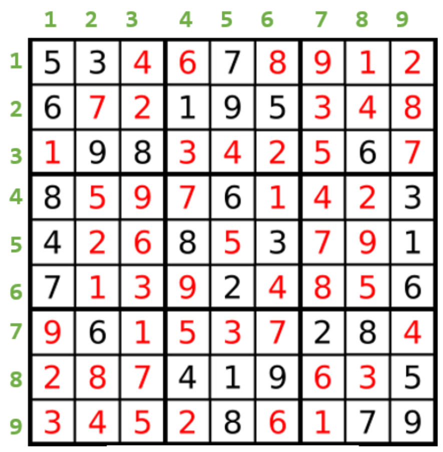
Sudoku is a popular Japanese puzzle game based on the logical placement of numbers. Being a game of logic, Sudoku doesn’t require any calculation nor special math skills; all that is needed are brains and concentration.

The objective is to fill a 9×9 grid with digits so that each column, each row, and each of the nine 3×3 sub-grids that compose the grid containing all of the digits from 1 to 9. Below is an example of a typical starting puzzle grid and its solution grid.



	1	2	3	4	5	6	7	8	9
1	5	3			7				
2	6			1	9	5			
3		9	8					6	
4	8				6				3
5	4			8		3			1
6	7				2				6
7		6					2	8	
8				4	1	9			5
9					8			7	9

Initial Sudoku Board

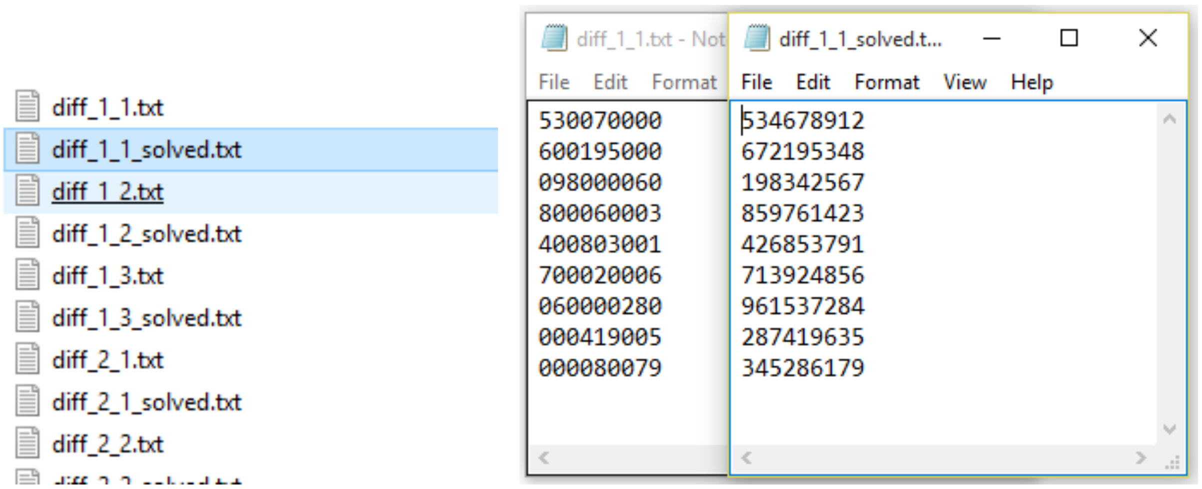


	1	2	3	4	5	6	7	8	9
1	5	3	4	6	7	8	9	1	2
2	6	7	2	1	9	5	3	4	8
3	1	9	8	3	4	2	5	6	7
4	8	5	9	7	6	1	4	2	3
5	4	2	6	8	5	3	7	9	1
6	7	1	3	9	2	4	8	5	6
7	9	6	1	5	3	7	2	8	4
8	2	8	7	4	1	9	6	3	5
9	3	4	5	2	8	6	1	7	9

Solved Sudoku Board

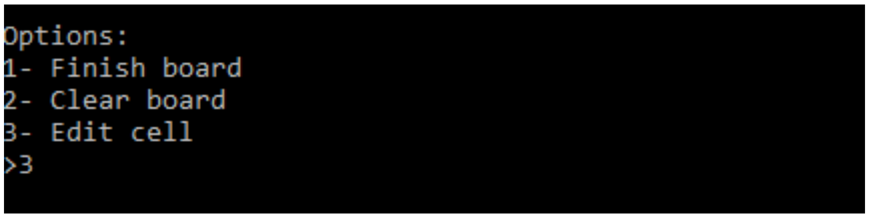
A well-constructed Sudoku puzzle has a *unique solution* and can be solved by logic, although it may be necessary to employ "guess and test" methods in order to eliminate options.

- **Details**
  1. Your game should provide an option to choose a difficulty level. There should be 3 difficulty levels.
  2. You should have 9 *unsolved* and *solved* Sudoku boards (3 Sudoku boards for each difficulty level), saved in a text file(s). When a player selects a difficulty level, load the *unsolved* Sudoku board of the selected difficulty from the file and display it on the screen. *(See an example screenshot for the files below. Zeros in the unsolved files mean these will be empty cells that the player must fill correctly in the game.)*
  3. Take care that your saved Sudoku boards have unique solutions.



```
530070000    534678912
600195000    672195348
098000060    198342567
800060003    859761423
400803001    426853791
700020006    713924856
060000280    961537284
000419005    287419635
000080079    345286179
```

- **Input**
  At each step in the game, there should be 3 options available:
  1. An option to print the finished board.
  2. An option to clear the board to the initial Sudoku board.
  3. An option to edit a single cell in the board. *(ie. put a value)*



```
Options:
1- Finish board
2- Clear board
3- Edit cell
>3
```

If the player chooses option #3 (to edit the board), then the next input should be 3 numbers:
1. Row number of cell to assign a number
2. Column number of cell to assign a number
3. Value to assign to the cell

Example (to edit the board):

```
4        ; <- Row number
3        ; <- Column Number
5        ; <- Put 5 in cell (4, 3)
```

- **Output**
  1. If the player decides to finish the game, then display the entire solved board and end the game.
  2. Should the player decide to edit a cell, then once they provide the input and press enter, you should check whether that solution is correct by comparing the player's input by the value of the corresponding cell that is saved in the solution file.
     a. If the solution is incorrect, print a message in **red** saying that the solution was incorrect. No changes should be made to the board.
     b. If the solution is correct, print a message in **green** saying that the solution was correct and print an updated Sudoku board.
  3. At the end of the game, print a report stating:
     a. The number of times the user entered an incorrect solution.

b. The number of times the user entered a correct solution.
c. The amount of time the user took to correctly complete the entire board.

- **Design and Regulations**

1. The Sudoku board must employ at least 2 different colors: 1 color representing the initial Sudoku board and another color representing all the cells solved by the user. *(i.e. as shown in the example above).*

- **Bonus**

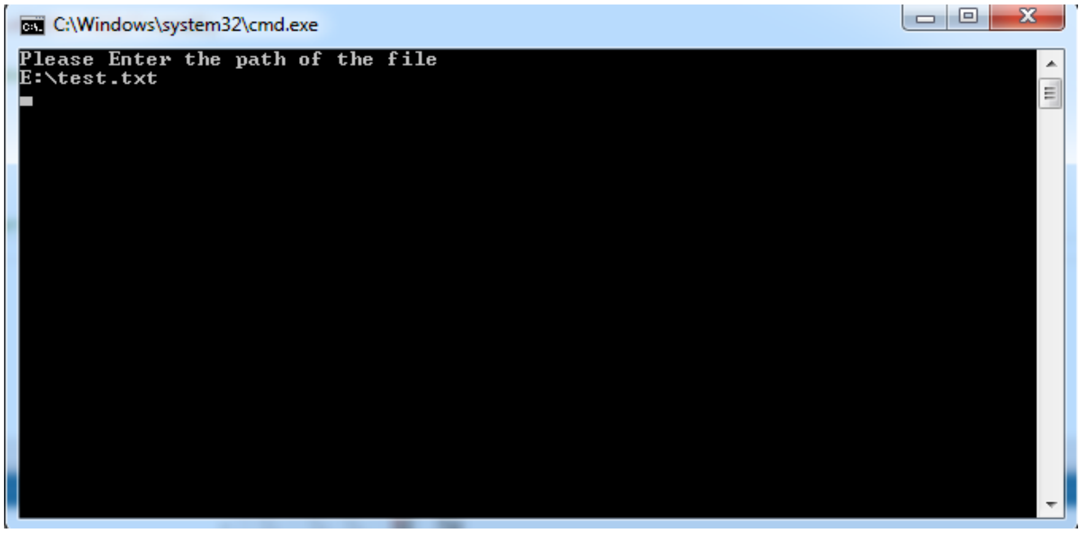
1. Implement a GUI for the game.
2. Score Ranking:
   Each time a player plays a game, save their name and stats *(stats: number of times number of times the user entered an incorrect solution, number of times the user entered a correct solution, amount of time the user took to correctly complete the entire board)* in an external file. At the end of each game, print the 5 top-ranking players and their stats.

6. Text Editor

Write an assembly program to make a text editor which supports a list of different functions:

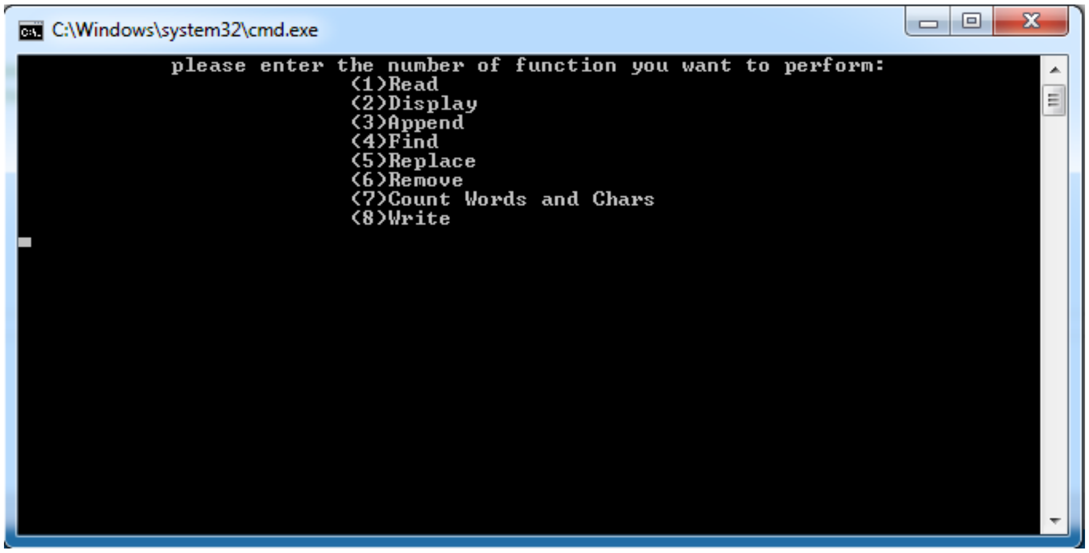
- Read text from specific file (which the file path taken as a user input).
- Displays the first page of this file (as much as can be displayed on the screen).
- Append sentence at the end of file
- Finds a specific string in file and check whether found or not:
  a. Search in file by the specific string if the string found, display a message box with message string is found and how many time this string repeated in whole file.
- Replaces a given string with new one:
  a. User should enter the new string and times of replacement (r) program should find all matched string and replace each old string with the new one according the value of (r).
- Remove specific string from the file:
  a. User should enter a string and times of replacement (r) program should find all matched string and remove them according the value of (r).
- Count number of words and number of characters in the file.
- Write the updated content (string) into the file.

- **Description**
  1. Input: At the beginning the user should enter the path of file and read the file (figure 1).
  2. Use the heap in assembly language to create a string that holds the content of the file.
  3. The program should be able to let the user chooses which function to perform on the created string (figure 2).
  4. Output: At the end write any changes caused by any of these functions into the file (i.e. any change must be reflected on the file).



```
Please Enter the path of the file
E:\test.txt
```

Figure 1: file path



```
          please enter the number of function you want to perform:
                         (1)Read
                         (2)Display
                         (3)Append
                         (4)Find
                         (5)Replace
                         (6)Remove
                         (7)Count Words and Chars
                         (8)Write
```

Figure 2: list of options

- **Bonus**
  1. Implement a GUI for the game.

- **References**
  1. Chapter 11: MS-Windows Programming

7. Motif Finding Problem

Pattern discovery in DNA sequences is one of the most challenging problems in molecular biology and computer science. In its simplest form, the problem can be formulated by giving a set of sequences, find an unknown pattern that occurs frequently with an allowed number ***n*** of mutations. A mutation means that the sequence differs from the search pattern in ***n*** characters. A DNA motif is defined as a nucleic acid sequence pattern that has some biological significance such as being DNA binding sites for a regulatory protein.

Example

1. Finding a motif 'AAAAAAAGGGGGG' in the following random sample:

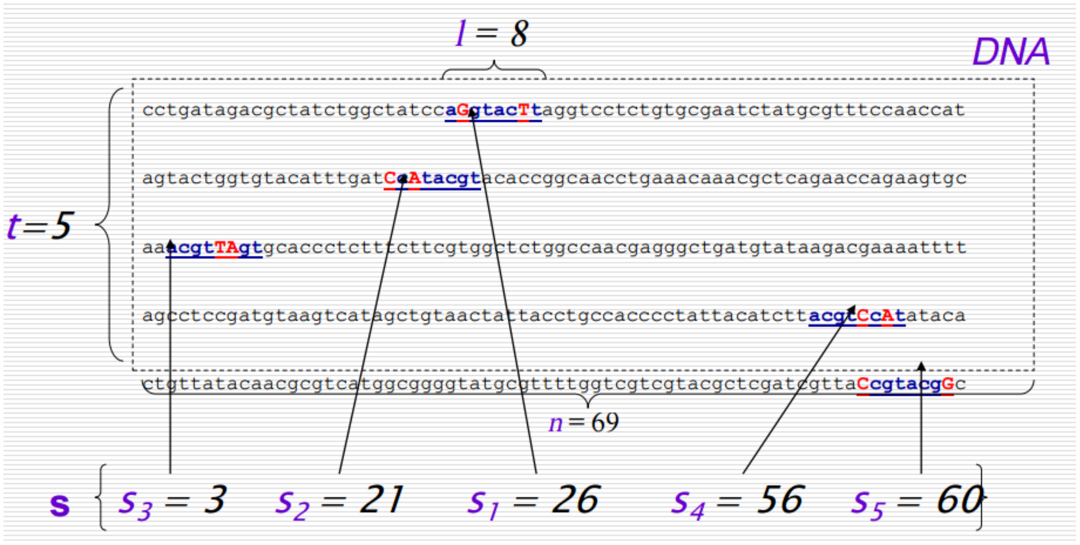
```
atgaccgggatactgatAAAAAAAAGGGGGGGggcgtacacattagataaacgtatgaagtacgttagactcggcgccgccg
acccctatttttgagcagatttagtgacctggaaaaaaaatttgagtacaaacttttccgaataAAAAAAAAGGGGGGGa
tgagtatccctgggatgacttAAAAAAAAGGGGGGGtgctctcccgattttgaatatgtaggatcattcgccagggtccga
gctgagaattggatgAAAAAAAAGGGGGGGtccacgcaatcgcgaaccaacgcggacccaaaggcaagaccgataaaggaga
tccttttgcggtaatgtgccgggaggctggttacgtagggaagccctaacggacttaatAAAAAAAAGGGGGGGcttatag
gtcaatcatgttcttgtgaatggatttAAAAAAAAGGGGGGGgaccgcttggcgcacccaaattcagtgtgggcgagcgcaa
cggttttggcccttgttagaggcccccgtAAAAAAAAGGGGGGGcaattatgagagagctaatctatcgcgtgcgtgttcat
aacttgagttAAAAAAAAGGGGGGGctggggcacatacaagaggagtcttccttatcagttaatgctgtatgacactatgta
ttggcccattggctaaaagcccaacttgacaaatggaagatagaatccttgcatAAAAAAAAGGGGGGGaccgaaagggaag
ctggtgagcaacgacagattcttacgtgcattagctcgcttccggggatctaatagcacgaagcttAAAAAAAAGGGGGGGa
```

2. Finding a motif ‘AAAAAAAAGGGGGG’ with 4 allowed mutations

```
atgaccgggatactgatAgAAgAAAGGttGGGggcgtacacattagataaacgtatgaagtacgttagactcggcgccgccg
acccctatttttgagcagatttagtgacctggaaaaaaaatttgagtacaaacttttccgaatacAAtAAAAcGGcGGGa
tgagtatccctgggatgacttAAAAtAAtGGaGtGGtgctctcccgattttgaatatgtaggatcattcgccagggtccga
gctgagaattggatgcAAAAAAAGGGattGtccacgcaatcgcgaaccaacgcggacccaaaggcaagaccgataaaggaga
tccttttgcggtaatgtgccgggaggctggttacgtagggaagccctaacggacttaatAtAAtAAAGGaaGGGcttatag
gtcaatcatgttcttgtgaatggatttAAcAAtAAGGGctGGgaccgcttggcgcacccaaattcagtgtgggcgagcgcaa
cggttttggcccttgttagaggcccccgtAtAAAcAAGGaGGGccaattatgagagagctaatctatcgcgtgcgtgttcat
aacttgagttAAAAAAtAGGGaGccctggggcacatacaagaggagtcttccttatcagttaatgctgtatgacactatgta
ttggcccattggctaaaagcccaacttgacaaatggaagatagaatccttgcatActAAAAAGGaGcGGaccgaaagggaag
ctggtgagcaacgacagattcttacgtgcattagctcgcttccggggatctaatagcacgaagcttActAAAAAGGaGcGGa
```

- **Input**
  A file contains t x n matrix of DNA, l, and m
  - t: number of sample DNA sequences
  - n: length of each DNA sequence
  - l: the length of pattern to find
  - m: the number of allowed mutations

- **Output**
  1. Display the file on the screen with highlighting the search pattern in the file.
  2. A file contains an array of *t* starting positions *s* = (s1, s2, s3, … st)



l = 8
DNA
t=5
cctgatagacgctatctggctatccaGgtacTtaggtcctctgtgcgaatctatgcgtttccaaccat
agtactggtgtacatttgatCcAtacgtacaccggcaacctgaaacaaacgctcagaaccagaagtgc
aaacgtTAgtgcaccctctttcttcgtggctctggccaacgagggctgatgtataagacgaaaatttt
agcctccgatgtaagtcatagctgtaactattacctgccaccccctattacatcttacgtCcAtataca
ctgttatacaacgcgtcatggcggggtatgcgttttggtcgtcgtacgctcgatcgttaCcgtacgGc
n = 69
s
s3 = 3
s2 = 21
s1 = 26
s4 = 56
s5 = 60

- **Bonus**
  1. Implement a GUI for the project to read search pattern from user, load the DNA sequences file, and display the file with the search pattern highlighted. The starting positions should be displayed to the user and be able to save it in a file.