UEE1303(1009) S22
Winter Vacation Exercise

[Instruction]

1.This exercise is the prerequisites for this course. Please hand in before deadline. Otherwise, you can't take this course.

2. Please put your source code files into a folder named StudentID_hw0, compress the folder to a zip file (ex: 0810700_hw0.zip) , and upload the file to e3 before deadline.

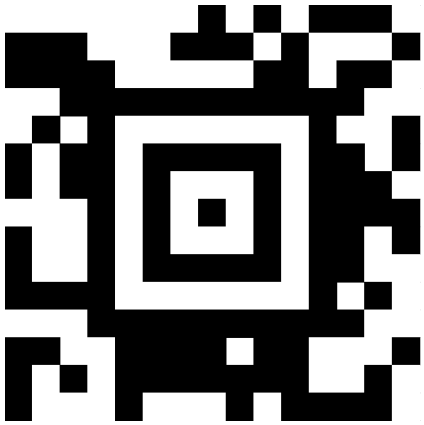
3. Note that the source code files should be named ex1.cpp, ex2.cpp and ex3.cpp.

4. We will provide you the testcases, you can download it from e3.

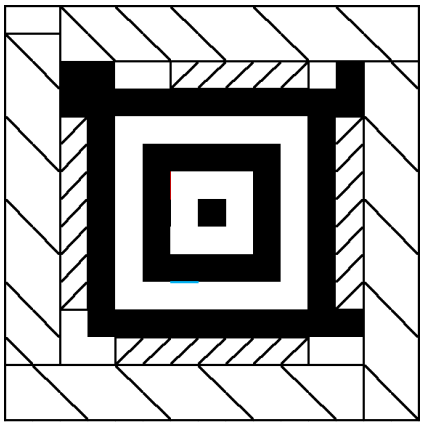
5. If you have any question, please send an email to TA.

PROBLEM 01 SIMPLIFIED LAYER-1 AZTEC CODE

There are numerous forms of 2D barcode such as QR-code, Data-Matrix, AR-code etc. Aztec is a sort of 2D barcode published in 1995 and is commonly used throughout Europe. It is invented by Andrew Longacre, Jr. and Robert Hussey. Layer-1 Aztec Code is 15x15 in size and can be divided into four components as illustrated below.



Aztec Code

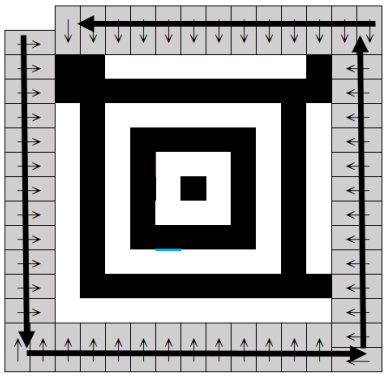


4 parts of Aztec Code

The first part is the position mark (black color) in the center. It is used to enable the scanner to recognize the direction of the Aztec Code 360 degrees, so that even if the scan is reversed, it is possible to determine which side is the beginning and which side is the end, avoiding reading errors. The second part is the mode message (covered by slash /), which is used to indicated which mode is being used and total amount of words in the barcode. The third part is the encode section (covered by backslash \), which contains the encoded and error correction message. Error correction message is generated by *Reed-Soloman error correction*. The final part is a blank area (white color), which is the unused part, always be blank.

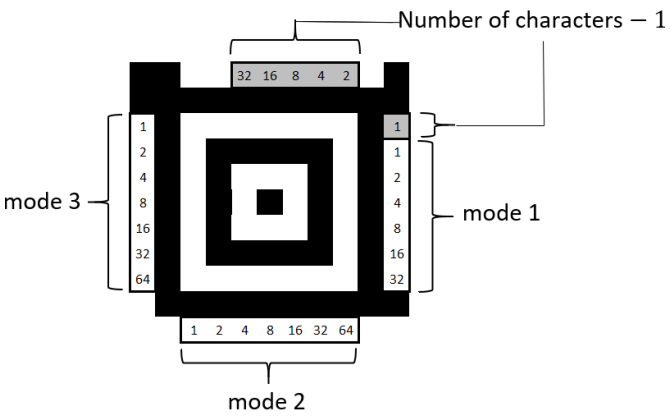
To generate the most basic Aztec Code, we must understand how scanners operate. To read the encoded sections (covered by backslash \) it will begin reading from the left-side and proceed counter-clockwise. The little arrow indicates the order for decoding each value of the encoded message. For example, if the encoded message is on the left,

the scanner reads each value from left to right; if it is on the bottom, the scanner reads from bottom to top.

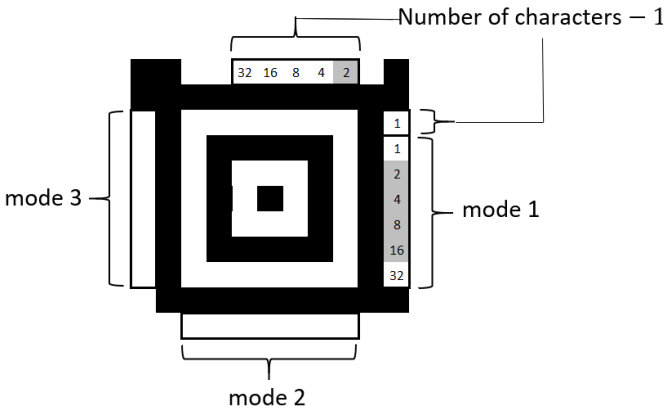


Reading encoded message

To simplify the problem, we divided the mode message into 4 parts. First part (covered by light grey) is used to record the number of characters in barcode. Furthermore, another 3 modes are utilized to record error checking and other information.
For example, if the message is "abc" with only mode 1 equal to 30, the number of characters will be the 3-1=2 and mode1 = 2 + 4 + 8 + 16 as shown as right figure below.



Reading mode message



Example of mode message

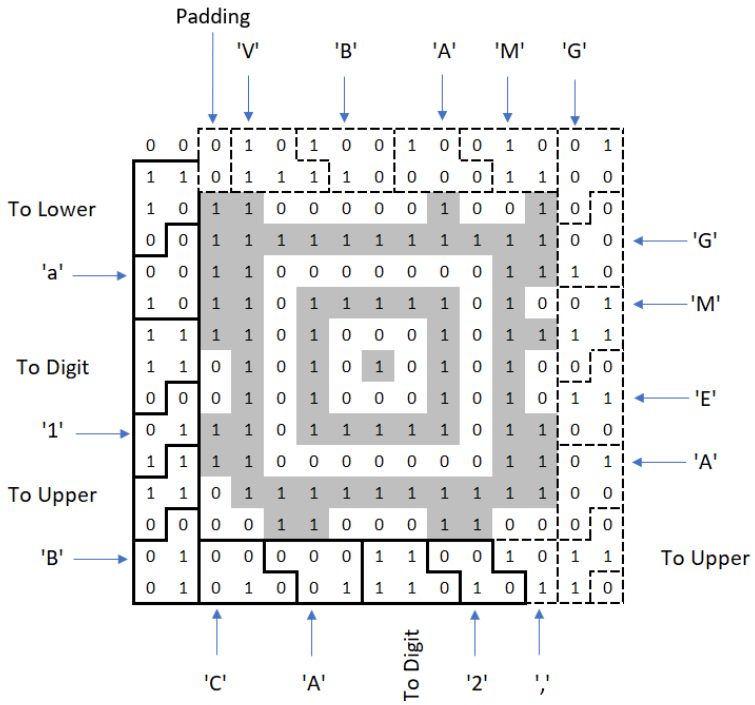
Because scanners only comprehend numbers, they will employ the Aztec Code character encoding table to convert words to Aztec Code (shown in below). This table is used in the same way as the ASCII table. There are 3 different modes (Upper, Lower, Digit) in table, which means that the mode must be changed using a specific value. In Upper and Lower mode, 5 bits are used to represent each character, but only 4 bits are used in digit mode. Remember that the initial mode is always be Upper mode. For example, encoding the word "I5", the step will be 01010 ('I') → 11110 (Upper to Digit mode) → 0111 ('5'), so the final output value will be 01010111100111. Several columns in this table have been filled in black to indicate that they are not relevant to this question.

Code	Mode		
	Upper	Lower	Digit
0			
1			
2	A	a	0
3	B	b	1
4	C	c	2
5	D	d	3
6	E	e	4
7	F	f	5
8	G	g	6
9	H	h	7
10	I	i	8
11	J	j	9
12	K	k	,
13	L	l	.
14	M	m	To Upper
15	N	n	

Code	Mode	
	Upper	Lower
16	O	o
17	P	p
18	Q	q
19	R	r
20	S	s
21	T	t
22	U	u
23	V	v
24	W	w
25	X	x
26	Y	y
27	Z	z
28	To Lower	
29		
30	To Digit	To Digit
31		

Aztec Code Character Encoding Table

In this problem, you must follow the rules and create a scannable Aztec Code. The total size of Aztec Code is 15x15, use ‘1’ to represent the grid is black, use ‘0’ to represent a white grid. If any remaining grids are not in use, they should be padded with white (value '0'). The figure below illustrates how to generate an Aztec Code.



Message: a1BCA2
Error checking Message: ,AEMGGMABV
Mode1: 50
Mode2: 99
Mode3: 103

Number of characters = 6 - 1 = 5
Mode1: 50 = 2 + 16 + 32
Mode2: 99 = 1 +2 +32 + 64
Mode3: 103 = 1 + 2 + 4 + 32 + 64

Example of generating Aztec Code

(testcase #1)

```
>vim message.txt
```

```
a1BCA2
,AEMGGMABV
50
99
103
```

```
>./ex1 message.txt
```

```
000101001001001
110111100001100
101100000100100
001111111111100
001100000001110
101101111101001
111101000101111
110101010101000
000101000101011
011101111101100
111100000001101
110111111111100
000011000110000
010000011001011
010100111010110
```

(testcase #2)

```
>vim message.txt
```

```
cialab
gcco74,505979,
40
123
37
```

```
>./ex1 message.txt
```

```
0000111011111010
1100110101011100
1011000001001111
0011111111111110
0111000000011100
0001011111010111
0111010001010011
0101010101010101
0001010001011100
0011011111010011
1001000000011011
0101111111111111
1000110111100101
0000100000010000
1010011001000011
```

(testcase #3)

```
>vim message.txt
```

```
aA
WQOMRgskzqmbza
55
20
99
```

```
>./ex1 message.txt
```

```
0000001110011100
1100101011001101
1011000000001101
0011111111111111
0011000000011101
1011011111011110
1101010001011101
1101010101011110
0101010001010000
1111011111011110
0011000000011110
0001111111111100
1000001010000010
1010000010011100
1000110011101111
```

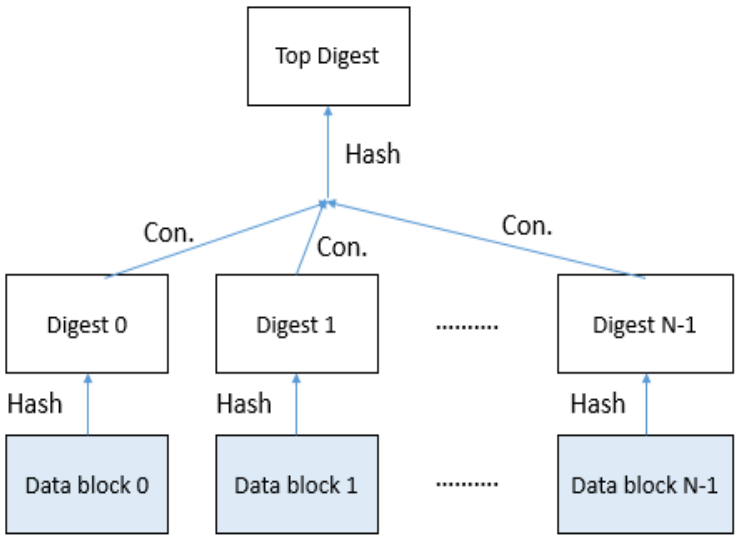
PROBLEM 02 FILE VERIFICATION CODE

In this problem, you need to generate the hash list of a text file that are composed of visible characters. You will read a text file and output the hash list to the output file. The output file and the input file name will be indicated in the command line. The command line format shows as below.

```
>./ex2 [input_file_name] [output_file_name]
```

Introduction

When transmitting a file, we cannot guarantee that the file will not be damaged. To ensure the file's integrity, the sender will divide it into numerous data blocks and calculate message digests for each data block. The receiver can obtain the message digests, which are valid. After transmitting the final bit of the final data block, the sender will generate a hash list as the following picture. As illustrated in the picture, the receiver will calculate the message digest for each received data block and then the top digest by hashing all message digests. The receiver will compare these digests to the sender's digests after computing the hash list. The recipient will begin by comparing the top digest. If the top digest differs, the file is corrupted, and the receiver will check each digest to determine which data blocks are corrupted. After determining which data blocks are faulty, the receiver will request that the sender retransmit them.

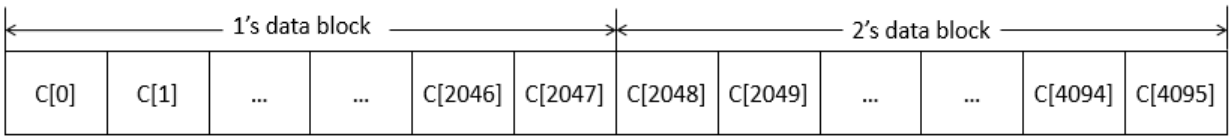


This program is consisted of two parts: divide file into data blocks and construct hash function.

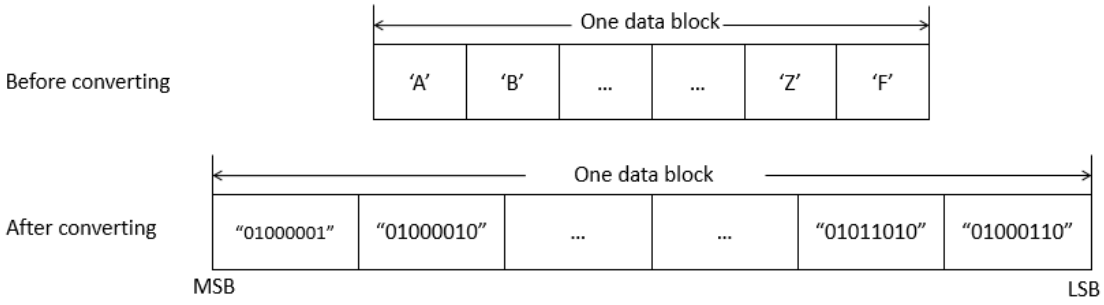
Procedure

1. Divide file into data blocks

The size of each data block is 2K bytes, which means that each data block contains 2048 characters. To begin, determine whether the amount of characters in the file is a multiple of 2048 characters or not. If not, you must pad underline (‘_’) to make the file a multiple of 2048 in size.



After padding, each character must be converted to the binary string. For example, the 'A' character is converted to "01100001" binary string that length is 8 according ASCII table.

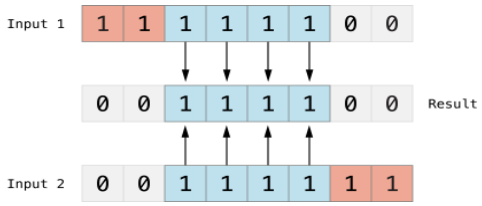


2. Construct hash function

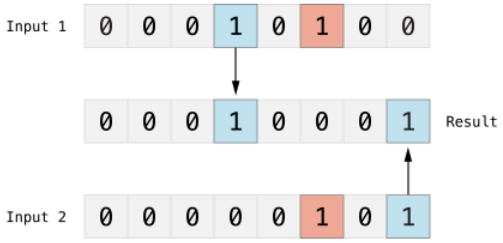
Before constructing hash function, you first need to implement the 7 functions.

I. ADD operator for a 32-bit binary string

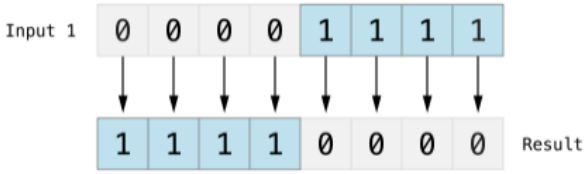
II. Bitwise AND operator for a 32-bit binary string



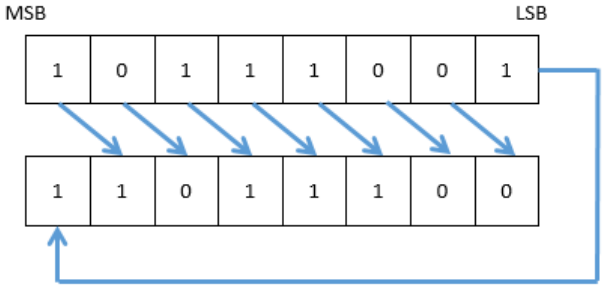
III. Bitwise XOR operator for a 32-bit binary string



IV. Bitwise NOT operator for a 32-bit binary string



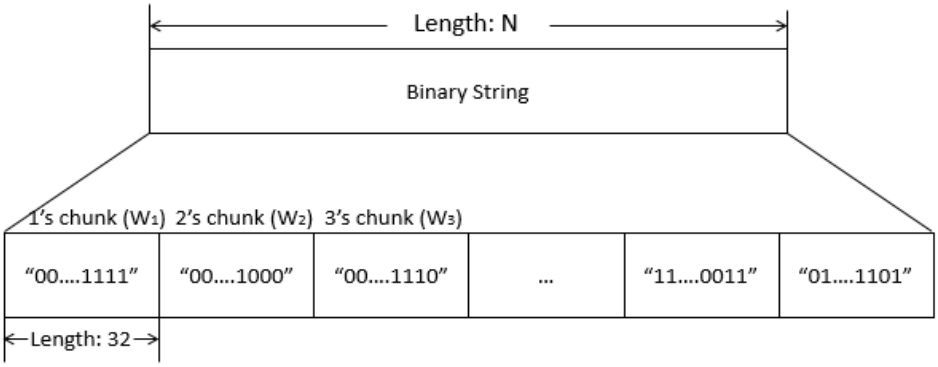
V. Circular Right Shift operator (RotR) for a 32-bit binary string
Right bit shifting means moving the position of the bits in the right direction. For example, if we shift 10111001 right with one position, we'll get 11011100.



VI. Choose operator for a 32-bit binary string
This operator is composed of AND, NOT and XOR operators. It has three input and one output. The boolean function is that
Ch(X, Y, Z) = (X AND Y) XOR (~X AND Z)

VII. Majority operator for a 32-bit binary string
This operator is composed of AND and XOR operators. It has three input and one output. The boolean function is that
Maj(X, Y, Z) = (X AND Y) XOR (X AND Z) XOR (Y AND Z)

What is a hash function? A hash function is any function that can be used to map data of arbitrary size to fixed-size values. In this problem, we'll use a custom hash function. This function will split the input binary string to lots of chunks. Each chunk contains a 32-bit binary string.



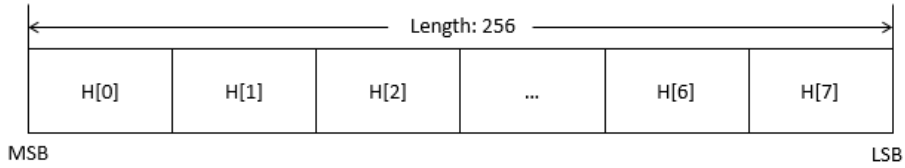
After splitting, the hash function will create 8 variables that are binary strings. Each variable is 32 characters in length and has the following initialized value:

```
H[0] = "01101010000010011110011001100111"
H[1] = "10111011011001111010111010000101"
H[2] = "00111100011011101111001101110010"
H[3] = "10100101010011111111010100111010"
H[4] = "01010001000011100101001001111111"
H[5] = "10011011000001010110100010001100"
H[6] = "00011111100000111011001101010111"
H[7] = "01011011111000001100110100011001"
```

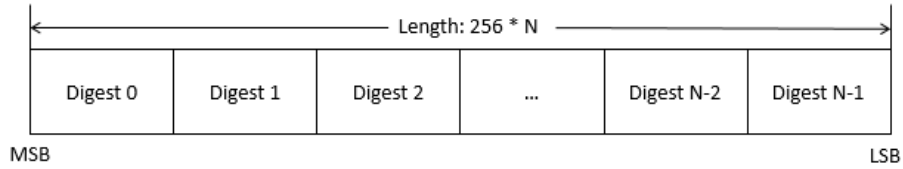
The function will iterate each chunk in the ascending order to execute the following pseudocode.

```
In each iteration, the chunkᵢ does the following operations
    T1 = H[7] + [RotR(H[4], 6) XOR RotR(H[4, 11) XOR RotR(H[4],
    25)] + Ch(H[4], H[5], H[6]) + Wᵢ
    T2 = [RotR(H[0], 2) XOR RotR(H[0], 13) XOR RotR(H[0], 22)]
    + Maj(H[0], H[1], H[2])
    H[7]  = H[6]
    H[6]  = H[5]
    H[5]  = H[4]
    H[4]  = H[3]  + T1
    H[3]  = H[2]
    H[2]  = H[1]
    H[1]  = H[0]
    H[0]  = T1 + T2
```

After iterating through all chucks, the hash function concatenates eight variables to form a binary string containing 256 characters, as illustrated below. The output of the custom hash function is this binary string.



You must concatenate all digests after hashing the data blocks to generate a result that contains 256 times N bits. The top digest is then obtained by hashing the concatenating value.



INPUT

The input file is composed of visible characters.

OUTPUT

You must print the entire hash list. Each line contains a digest which is a 256-bit binary string. The output format is as following

```
[Top Digest]
[Digest 0]
[Digest 1]
…
[Digest N]
```

(testcase #1)

>vim txtFile.txt

```
_
```

>./ex2 txtFile.txt output

>cat output

```
000110010101…111100
011111000001…101101
```

(testcase #2)

>vim txtFile.txt

```
Fc5sBX,udq
```

>./ex2 txtFile.txt output

>cat output

```
101101110001…100010
011001011001…001011
```

(testcase #3)

>vim txtFile.txt

```
l_6*/5Kbo0*nx3^Kc>k=W/|#ad3iIF%6'8A3Kls;y}…|BDz}@[hV>[;
```

>./ex2 txtFile.txt output

>cat output

```
11101001010…011000
11000101001…111011
```

(testcase #4)

>vim txtFile.txt

```
B;$n2Rsuisc[rVOip?J8\.y:W{uI@(xb@|NRMBE4... :ANUf]rh]@4+
```

>./ex2 txtFile.txt output

>cat output

```
01010011110…101111
00001011001…100101
00101001101…100011
11110001110…110010
```

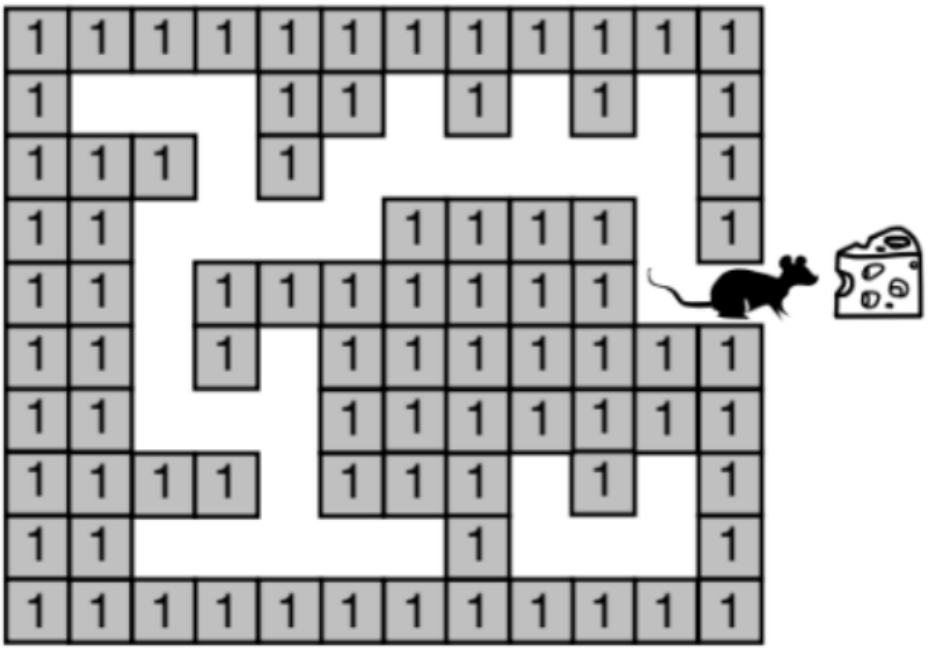
PROBLEM 03 HELP MOUSE FIND CHEESE

Write a program that simulates the movement of a mouse through a maze. The program must **print the path** travelled by the mouse from the start point to the end point, or **print merely the start point** if the mouse cannot exit the maze. Note that the output must be written to the file specified by the second parameter in the command line.

```
>./ex3 [input_file_name] [output_file_name]
```

You'd have two inputs, both of which come from the same input file as the first argument from the command line. There are a binary map of a maze and a point list that stands for candidates to designate where the mouse begins its journey. Multiple cases are contained in an input file; each case is denoted by a symbol.

The following image depicts an example of a maze. Each pixel stands for an element that can be either black or white. A black element is a square that the mouse cannot enter; on the other hand, a white element is a square that the mouse can utilize. A black element would be represented by a 1, while a white element would be represented by a 0.



The maze's points can be represented by a **structure made up of two integer fields**. The row and column coordinates of the maze are the first and second fields, respectively. The exit in the illustration, for example, is at (5, 12), which is row 5, column 12.

The program starts by reading the map and starting points of the maze from the input file. Continue reading the files until you reach the conclusion of each one.

You'll be given a list of candidates of start point. You should check the point's veracity. If the chosen point is in the black square (number 1), you must continue selecting points

from the list until the chosen point is valid.

Two linked lists are required to solve this problem. The first linked list is the visited list, stores the mouse's current path. It checks first to see if the mouse's present location is an exit. If it isn't, the location is added to the list. If the mouse comes to a halt and must retrace, the list is used (pop back one by one).

When the mouse enters an intersection, all options are added to a second list. If the mouse strikes a decision token while retracing, the token is deleted and the next choice in the alternatives list is selected.

If the mouse reaches a dead end with both stacks empty, it is stuck in that area of the maze. In this case, you must return to the starting position and then end the search. Note that there may be another path out of the maze; you only need to print one of them.

(testcase)
>vim testcase.in

```
.
111
101
100
[(1, 1), (2, 2)]

.
111
101
111
[(2, 1), (2, 2)]
.
11110
10011
11101
[(2, 2), (3, 4), (3, 1)]

.
11110
10011
11101
[(3, 4), (2, 2), (3, 1)]
```

>./ex3 testcase.in testcase.out
>vim testcase.out

```
.
(2, 2)
[(2, 2), (3, 2)]

.
(2, 2)
[(2, 2)]

.
(2, 2)
[(2, 2)]

.
(3, 4)
[(3, 4)]
```