Instructions:

Please read this information carefully before you start working on your final exam, and read it again before turning it in to make sure you have followed the instructions. There will be no resubmission allowed.

- This exam is due on 5/14/2020 at Noon.
- The exam consists of a project section and a quiz.
- You are expected to clearly explain your answer to every quiz question and add concise explanatory comments to every function you use in the project.
- You may use any class materials to complete the project and reuse any parts of previous lab assignments as you may need, keeping in mind that you will have to explain all answers in detail, as well as add concise and informative comments to your program.
- Projects turned in with insufficient commenting or lack of clarity in answers will have points taken off (if I cannot follow your reasoning, you will receive partial or no credit even if you get the right answer).
- **You are expected to work on this final project on your own.**
- You may ask questions to the instructor or the TA, but we will not give answers or help with the programming section (i.e. do not send us your program, we will not look at it like we do with the labs).
- Here are instructions for answering your exam:
  1. For problems that start with a ”Q” (quiz), you need to enter your answers in the allotted space on Learn.
  2. For problems that start with a ”P” (project), you will find the Final Exam-Project assignment listed on Learn with a copy of this PDF. You will turn in your answers to these project problems as you do with regular assignments, through Learn. **Only a single attempt is allowed, so please make sure you are attaching the correct files.**
  3. Optional: You may also include in the attachments for the Final Exam-Project assignment any auxiliary notes that helped you find answers to any of the quiz (Q) or project (P) problems, making sure you follow the 4 points below:
     - The notes can be either typed or handwritten (scan or photo). If handwritten, make sure they are legible.
     - For each question, indicate on the top of the page your name and the question number that the note refers to.
     - Please combine all notes into a single PDF with filename <UNM Net ID>-FinalNotes.pdf (e.g. my file would be named bjacobson-FinalNotes.pdf).
     - Even if you choose to send me your notes, you are still required to answer all quiz questions (Q) in the allotted space on Learn as indicated above.

When you return this exam you acknowledge that all answers therein are your own.

Question:	Q1	Q2	Q3	Q4	Q5	Q6	Q7	P8	P9	Total	Extra Points
Points:	5	5	5	10	10	10	15	20	20	100	6
Score:											

Q1:

(a) What is the output of the program below? (1 pt)

(b) What is the algorithm in function foo commonly called? Explain the function foo in a few sentences. (2 pts)

(c) Describe one alternative to this function that we discussed in class. How does this alternative work? (2 pts)

```c
#include <stdio.h>

void foo(int array[], int n)
{
  int i, s = 1;

  while(s)
    {
      s = 0;
      for (i=0; i<n-1; i++)
        {
          if (array[i] > array[i+1])
            {
              int tmp = array[i];
              array[i] = array[i+1];
              array[i+1] = tmp;
              s = 1;
            }
        }
    }
}

void main(void)
{
  int numList[] = {12, 7, 20, 44, 1, 8, 13};
  int n = sizeof(numList)/sizeof(int);
  int i;

  foo(numList, n);
  for (i=0; i<7; i++)
    {
      printf("%d ", numList[i]);
    }
  printf("\n");
}
```

Q2: Explain the difference between the declarations: (5 pts)

```
char string[] = "Lux Hominum Vita";
char* unm = "Lux Hominum Vita";
char string[14] = "Lux Hominum Vita";
char string[17] = {'L','u','x','
','H','o','m','i','n','u','m',' ','V','i','t','a','\0'};
```

The structure shown below refers to questions Q3 and Q4:

```
struct TreeNode
{
  int data;
  struct TreeNode* left;
  struct TreeNode* right;
};
```

Explain in detail what each of the functions foo in Q1 and Q2 do:

Q3 (5 pts):

```
struct TreeNode* foo(struct TreeNode* root, int data)
{
  if(root == NULL)
  {
    root = createNode(data);
  }
  else if(data < root->data)
  {
    root->left = foo(root->left, data);
  }
  else
  {
    root->right = foo(root->right, data);
  }
  return root;
}
```

Q4 (10 pts):

```
int foo(struct TreeNode** rootRef, int data)
{
  struct TreeNode* node;
  if(*rootRef == NULL)
  {
    return 0;
  }
  else if(data < (*rootRef)->data)
  {
    return foo(&((*rootRef)->left), data);
  }
  else if(data > (*rootRef)->data)
  {
    return foo(&((*rootRef)->right), data);
  }
  else
  {
    if((*rootRef)->left == NULL)
    {
      node = *rootRef;
      *rootRef = (*rootRef)->right;
      free(node);
    }
    else if ((*rootRef)->right == NULL)
    {
      node = *rootRef;
      *rootRef = (*rootRef)->left;
      free(node);
    }
    else
    {
      (*rootRef)->data = minValueBST((*rootRef)->right);
      return foo(&((*rootRef)->right), (*rootRef)->data);
    }
    return 1;
  }
}
```

Suppose the functions below are part of a sudoku solver program. Questions Q5, Q6, Q7 refer to these functions.

```c

#include <stdio.h>
#include <math.h>

unsigned int row[9],col[9],grd[9],t[81];
int sol[81];
int idx[3],nidx[2];

void initializeArrays(void)
{
  int i;
  for (i=0;i<9;i++)
  {
    row[i] = 0;
    col[i] = 0;
    grd[i] = 0;
  }
  for (i=0;i<3;i++)
  {
    idx[i] = 0;
  }
  for (i=0;i<81;i++)
  {
    t[i] = 0;
  }
}

void getIndexes(int i)
{
  idx[0] = i/9; /*row*/
  idx[1] = i%9; /*column*/
  idx[2] = idx[1]/3+3*(idx[0]/3); /*grid*/
}


int foo(int count,int n)
{
  int ir,ic,ig;
  int i;
  unsigned int number,tr,tc,tg;

  i = count;

  getIndexes(i);

  ir = idx[0];
  ic = idx[1];
  ig = idx[2];

  number = (unsigned int) pow(2,(double)n);

  tr = number & row[ir];
  tc = number & col[ic];
  tg = number & grd[ig];

  if(tg == 0 && tc == 0 && tr == 0)
  {
    row[ir] += number;
    col[ic] += number;
    grd[ig] += number;
  }
  else
  {
    return 1;
  }
  return 0;
}

int main(void)
{
  int number = 0;
  int err = 0;
  int count;
  char c;
  initializeArrays();
  while ((c=getchar()) != EOF)
  {
    putchar(c);
    if (c != '\n')
    {
        if (c == '.')
        {
            number = 0;
        }
        else if(c >='1' && c <= '9')
        {
            number = c - '0';
        }
        else
        {
```

```
91              err = 1;
92          }
93          if (count >= 81)
94          {
95              err = 1;
96          }
97          else
98          {
99              if (err == 0)
100             {
101                 if(number != 0) err = foo(count,number);
102                 sol[count] = number;
103             }
104         }
105         count++;
106     }
107     else
108     {
109         if (count < 81) err = 1;
110         if (err == 1)
111         {
112             printf("Error!\n");
113         }
114         else
115         {
116             solvePuzzle();
117         }
118         err = 0;
119         count = 0;
120         initializeArrays();
121     }
122   }
123   return 0;
124 }
```

Q5: Explain what is happening in lines (10 pts):

(a) 78-92 (4 pts)

(b) 93-96 and 109-117 (4 pts)

(c) 118-120 (2 pt)

Q6: Explain lines 99-103 and the function foo that starts in line 36. (10 pts)

The following function foo2 is called by solvePuzzle (line 116) of the sudoku solver if a square at position i is empty:

```c
void foo2(int i, int ir, int ic, int ig)
{
    int nt;
    unsigned int temp;

    nt = ((grd[ig] | col[ic]) | row[ir]);
    nt = cleanBits(nt);
    t[i] = ~nt-1;
    temp = t[i];

    if (t[i] == 0)
    {
      printf("No solution!");
      return;
    }

    printf("temp=%u\n",temp);
    sol[i]=testSolution(temp);
}
```

The functions cleanBits() and testSolution() are:

```c
unsigned int cleanBits(unsigned int b)
{
  int i;
  for (i = 10; i< 32; i++)
  {
    b = b ^ 1 << i;
  }
  return b;
}

int testSolution(unsigned int sol)
{
  switch(sol)
  {
  case 2:
    return 1;
  case 4:
    return 2;
  case 8:
    return 3;
  case 16:
    return 4;
  case 32:
      return 5;
    case 64:
      return 6;
    case 128:
      return 7;
    case 256:
      return 8;
    case 512:
      return 9;
    default:
      return 0;
  }
}
```

Q7:

1. What does cleanBits() in line 7 of foo2 output? (2 pts)
2. What do the values of `nt` and `t[i]` represent in foo2? (3 pts)
3. Based on what you have learned about this sudoku solver program, consider an empty square i in row 2, column 1 of the game (columns, rows, and grids are numbered from 0 to 8). Explain whether the expression below is true or false:
   `if(a = (row[0] & row[1] & col[0] & col[1] & t[i]) != 0) sol[i] = a;`
   with t[i] as found in line 9 of foo2(). If it is true, explain why. If it is false, fix it to find sol[i] and explain. (10 pts)

P8: Finding Primes. Write 3 programs that find all primes from 2 to $N_{max} = 200$ via 3 different methods:

(a) Program primeArray.c: Initialize an array of size N_{max} that includes numbers 2 through N_{max}. Then exclude all primes from the array. (5 pts)

(b) Program primeSieve.c: Use the sieve of Eratosthenes method to remove non-primes from an array of numbers from two to N_{max}. The sieve of Eratosthenes works by starting with an array with all numbers from 2 to N_{max}. For the first prime, 2, remove every two numbers until N_{max} (removing all multiples of two). The first number left in the array after 2 is also prime. Then read that number i and remove every i numbers until N_{max}. This method does not check for division with primes. (5 pts)

(c) Program primeList.c: Build a singly linked list of all prime numbers in {2,..., N_{max}}, starting from 2. Do not use arrays. (8 pts)

(d) For $N_{max} = 100$, if you need to store all primes from 2 to 100, would a linked list or an array (not dynamically allocated) use less memory? How about for $N_{max} = 200$? At which point one does the size of one becomes smaller than the other? (2 pts)

(e) **Extra 1 point** if you write a table (or make a plot) comparing the size of the array and the size of the linked list as a function of N_{max} for values of N_{max} between 10 and 200. (Hint: You may combine two programs above into one to create a table)

- 1 point off (-1) from each program that does not have your name at the top of the file.
- 1 point off (-1) from each program that does not follow CS 241 coding standards.
- 1 point off (-1) from each program that does **not** print all primes in a line with a space separating them.

P9: Thue-Morse Sequence and the Koch Snowflake.

Goal: Write a program tmSnow.c that generates the Thue-Morse sequence, and use it to draw a Koch snowflake.

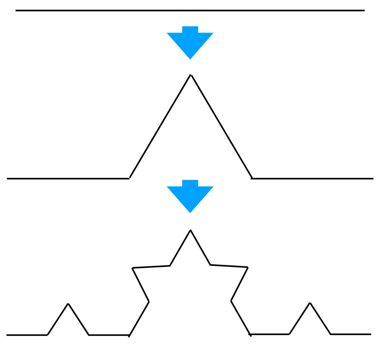
The Thue-Morse sequence starts with a single bit (0), and grows according to the following rules:

$$\begin{aligned} 0 &\rightarrow 01 \\ 1 &\rightarrow 10 \end{aligned}$$

So by generation, the sequence is:

0
01
0110
01101001
0110100110010110
...

Your program tmSnow.c will generate this sequence and use it to draw a bitmap file containing a Koch snowflake with a bitmap.c file.
The Koch snowflake is a fractal that grows like:

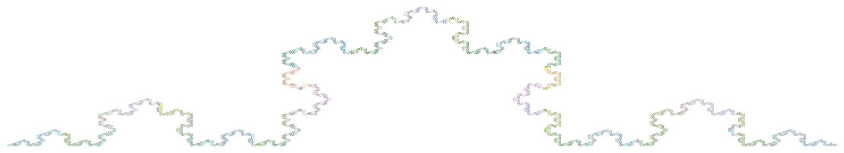


The Thue-Morse sequence generates the snowflake by following the two rules:

- For every 0 in the sequence, draw a line (in our case, a pair of adjacent pixels. See bitmap.c how the first pixel is drawn)
- for every 1 in the sequence, rotate by 60 degrees counter-clockwise (do not draw). The rotation guides where the next pixel will be drawn when you encounter a 0.

You are provided with a Makefile, a header file bitmap.h, and a file to write a bitmap, bitmap.c. You do not need to change bitmap.c much (you may leave all bitmap header information intact). You only need to add to the bitmap function in bitmap.c the algorithm to read in the Thue-Morse sequence that you generated with tmSnow.c, and use the function setRGB to draw the pixels at a position (x,y) that you will determine from the Thue-Morse sequence. You will need to initialize new variables in the bitmap function. Feel free to play with the colors of your snowflake. See the comments in bitmap.c to know where to make changes.

This is what mine looks like:



The bmp file generated can be quite large (it is not compressed). You can convert it to a jpeg from command line with:

```
convert snow.bmp snow.jpg
```

- 10 points: You can generate the 15th generation of the Thue-Morse sequence with tmSnow.c
- 10 points: You can draw the snow flake.
- Extra 5 points if you write a program tmList.c that saves each 0 and 1 of the Thue-Morse sequence into a linked list.

Attach your files bitmap.c, tmSnow.c, and snow.jpg.

- 1 point off (-1) from each program that does not have your name at the top of the file.
- 1 point off (-1) from each program that does not follow CS 241 coding standards.
- 1 point off (-1) from each program that does **not** print your Thue-Morse sequence in a line with a space separating each bit.

Hint: You can also play with the size of your bmp file (variables pixelWidth and pixelHeight) and also with the initial position where your first pixel is drawn at xpix and ypix.