Midterm Practice

Review Session: Wednesday, April 26th 7-9pm Location TBA.
Midterm Exam: Friday, April 28th 2:15 - 4:15pm Location TBA
Alternate Exam: Thursday, April 27th 6-8pm Location TBA

The midterm will be a 2-hour exam, open-book /open-notes, but no computers allowed.

Time and place of the exam

The midterm exam is scheduled at two different times to accommodate those of you who have scheduling constraints. You may take the exam at either time and need not give advance notice of which exam you plan to take. If you are cannot take the exam at either of the two times, please send an e-mail message to **yves@cs** stating the following:

- The reason you cannot take the exam at either of the scheduled times.
- A two-hour period on Thursday or Friday at which you could take the exam. This time must be during the regular working day, and must therefore start between 8:30 and 3:00 (so that it ends by 5:00).

In order to schedule an alternate exam, we must receive an e-mail message from you by 5:00P.M. on Monday, April 24th. **Late requests will not be honored**. Instructions for taking the alternate midterm will be sent to you by e-mail.

Review session

Your regular section meeting next week will include time for discussing the problems on this sample exam as well as general questions you might have about the midterm material. Our wonderful head TA Yves will also hold a general review session Wednesday night. We'll let you know the location as soon we find it out ourselves!

Coverage

The exam covers the material presented in class through Monday, April 24, which spans through Chapter 11 of the text.

This handout is intended to give you practice solving problems that are comparable in format and difficulty to those which will appear on the midterm. All of these problems were taken from previous exam so they are good examples of what to expect. We'll hand a solution to these problems in Monday's lecture. You are highly encouraged to work through the problems in test-like conditions to prepare yourself for the actual exam. Some of our section problems have been taken from previous exams and chapter exercises from the text often make appearances in same or similar forms on exams, so both of those resources are a valuable source of study material as well.

Note: To conserve trees, I have cut back on answer space. The actual exam will have much more room for your answers and for any scratch work.

General instructions

Answer each of the questions included in the exam. Write all of your answers directly on the examination paper, including any work that you wish to be considered for partial credit.

When writing programs for the exam, you do not need to be concerned with **`#include`** lines, just assume any of the libraries that you need (both standard C or 106-specific) are already available to you. If you would like to use a function from a handout or textbook, you do not need to repeat the definition on the exam, just give the name of the function and the handout or chapter number in which its definition appears.

Unless otherwise indicated as part of the instructions for a specific problem, comments will not be required on the exam. Uncommented code that gets the job done will be sufficient for full credit on the problem. On the other hand, comments may help you to get partial credit if they help us determine what you were trying to do.

The examination is open-book, and you may make use of any texts, handouts, or course notes. You may not, however, use a computer of any kind.

Problem 1: Recursion and big-O

Assume that the functions **`Mystery`** and **`Enigma`** have been defined as follows:

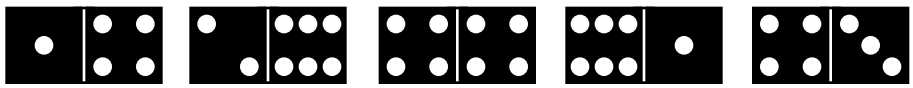
```
int Mystery(int n)
{
    if (n == 0) {
        return (1);
    } else {
        return (Enigma(2, Mystery(n - 1)));
    }
}

int Enigma(int n1, int n2)
{
    if (n1 == 0) {
        return (0);
    } else {
        return (n2 + Enigma(n1 - 1, n2));
    }
}
```

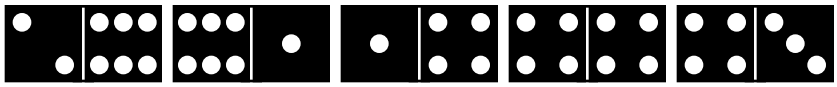
What is the value of **`Mystery(3)`**? What is the computational complexity of the **`Mystery`** function expressed in terms of big-O notation, where *N* is the value of the argument **`n`**, which you may assume is always a nonnegative integer.

Problem 2: Recursive procedures

The game of dominos is played with rectangular pieces composed of two connected squares, each of which is marked with a certain number of dots. For example, each of the following five rectangles represents a domino:



Dominos can be connected end-to-end to form chains, subject to the condition that two dominos can be linked together only if the numbers match. For example, you can form a chain consisting of all five of these dominos by connecting them in the following order:



Note: In traditional dominos, you can rotate a domino by 180˚ so that its numbers are reversed. As a simplification in this problem, this operation is not allowed, which means that the dominos must appear in their original orientation.

Dominos can, of course, be represented in C very easily as an abstract type whose underlying representation is a pair of integers. For this problem, assume you have access to a **domino.h** interface that exports the type **dominoADT** and the following operations:

```
dominoADT NewDomino(int leftDots, int rightDots);
int LeftDots(dominoADT domino);
int RightDots(dominoADT domino);
```

The first function creates a new domino with the indicated number of dots on its left and right sides. Given an existing domino, the other two functions return the number of dots on its left and right sides, respectively.

Given this abstract domino type, write a recursive function

```
bool FormsDominoChain(dominoADT dominos[], int n);
```

that returns **TRUE** if it possible to build a chain consisting of all **n** dominos in the array **dominos**.

As an example, suppose that the **dominos** array has been initialized as follows:

```
dominoADT dominos[5];

dominos[0] = NewDomino(1, 4);
dominos[1] = NewDomino(2, 6);
dominos[2] = NewDomino(4, 4);
dominos[3] = NewDomino(6, 1);
dominos[4] = NewDomino(4, 3);
```

Given these dominos, calling **FormsDominoChain(dominos, 5)** would return **TRUE** because it is possible to form the chain shown in the second diagram on the page. On the other hand, calling **FormsDominoChain(dominos, 4)** would return **FALSE** because there is no way to form a four-domino chain using only the first four dominos in the array.

Problem 3: Editor buffers

Besides simple insertion and deletion, most editors (and particularly mouse-based editors like the Macintosh) offer some facility for cutting and pasting text. The "cut" operation deletes text from the buffer but saves that text in a "clipboard" from which it can later be inserted by a "paste" operation.

In this problem, your job is to add a cut-and-paste facility to the stack-based editor from Chapter 9 of the text. Like the editor buffer itself, we will assume that the storage used for the clipboard is stored as a stack, which is declared globally as follows

```
stackADT clipboard;
```

and which you may assume has been initialized to an empty stack. Given this structure, you are to write implementations for the two functions whose interface descriptions are given below:

```
/*
 * Function: Cut
 * Usage: Cut(buffer, n);
 * ----------------------
 * Moves the n (or some smaller number if the buffer
 * has fewer than n characters left) characters
 * following the cursor from the buffer into the
 * clipboard and deletes those characters from the
 * buffer.  Any previous contents of the clipboard
 * are lost.
 */

void Cut(bufferADT buffer, int n);

/*
 * Function: Paste
 * Usage: Paste(buffer);
 * ---------------------
 * Inserts the contents of the clipboard at the
 * current cursor position.  The cursor ends up
 * at the end of the inserted text and the
 * clipboard contents do not change.
 */

void Paste(bufferADT buffer);
```

In this problem, you are not allowed to change the stack interface or cross the abstraction boundary to look inside the stack implementation.

Problem 4: Data Structure Design and ADTs

You are going to implement an ADT that can represent a matrix of floating point values. A matrix is a form of a two-dimensional array. Each row and col has a value which, in this case, will be of type `double`. Typical operations on such an ADT would probably include:

```
matrixADT NewMatrix(int numRows, int numCols);
void FreeMatrix(matrixADT m);
double GetElementAt(matrixADT m, int row, int col);
void SetElementAt(matrixADT m, double value, int row, int col);
void PrintMatrix(matrixADT m);
```

The hitch is that you know this ADT will be used most often to represent very large and very sparse matrices. A *sparse* matrix is one in which most of the elements are zero. A matrix of 1000 rows and 10000 columns might only have 100 non-zero values. The overhead of actually using a 2-D array would be prohibitive[1] and so you will need to develop an alternate representation.

In implementing the matrixADT, you must only store those elements that are non-zero. For each row, you must keep a singly-linked list of the non-zero elements, and the elements in each row should be stored in sorted order (sorted by column). The decisions of what data is stored in the CDT itself, what is kept in each list cell, whether to have a dummy header cell, etc. are left up to you. Your choices must result in an implementation that at least meets the following running time specifications (where R is the number of rows, C the number of columns):

NewMatrix	O(R)
FreeMatrix	O(R*C)
GetElementAt	O(C)
SetElementAt	O(C)
PrintMatrix	O(C*R)

The three functions in bold are ones that you have to write. You will not have to implement the others. As always, you are encouraged to write private helper functions to assist you in implementing the public operations. The interface file is given beginning on the next page. Be sure to carefully read the descriptions provided in the header file and make sure that your implementation is true to the published interface.

This space intentionally left blank. Please go on to next page.

[1] If you used the declaration

```
double matrix[1000][10000];
```

The variable `matrix` would take up sizeof(double)*1000*10000 = 80 million bytes or ~80 MB, about the size of your entire zip disk!

```
/* File: matrix.h
 * -------------
 * This interface exports the type and operations for a matrix of
 * floating points values.
 */
#include "genlib.h"

typedef struct matrixCDT *matrixADT;


/* Function: NewMatrix
 * Usage: m = NewMatrix(1000,5000)
 * -------------------------------
 * Creates a new matrixADT object and returns it.  The new matrix has the
 * specified number of rows & cols.  There is no upper limit on the size of
 * a matrix that can be created.  Each element is assumed to be zero at start.
 */
matrixADT NewMatrix(int numRows, int numCols);

/* Function: FreeMatrix
 * Usage:  FreeMatrix(m);
 * ----------------------
 * Frees the storages allocated with matrix m.
 */
void FreeMatrix(matrixADT m);


/* Function: SetElementAt
 * Usage:  SetElementAt(m, 3.14159, 12, 4300);
 * -------------------------------------------
 * Sets (or replaces!) the value at (row,col) position in matrix with new
 * value.  Raises an error when asked to set a position that is not a legal
 * row and col position. Row and col both start with 0 index.
 */
void SetElementAt(matrixADT m, double value, int row, int col);


/* Function: GetElementAt
 * Usage:  GetElementAt(m, 945, 4999);
 * -----------------------------------
 * Returns the value at (row,col) position of the matrix.  Any position
 * which has not been set is assumed to be zero.  Raises an error when asked
 * to retrieve a position that is not legal element in the matrix.
 */
double GetElementAt(matrixADT m, int row, int col);


/* Function: PrintMatrix
 * Usage: PrintMatrix(m);
 * ----------------------
 * Prints matrix in row col format. Do not be concerned about decimal
 * places, lining up decimals points and other printf formatting details.
    3.4 0.0 5.12           /* Row 0 */
    0.0 0.0 0.0            /* Row 1 */
    0.0 0.1 0.0            /* Row 2 */
 */
void PrintMatrix(matrixADT m);
```

Fill in the implementation of matrix.c. Define helper functions as needed. You can assume the FreeMatrix and GetElementAt functions are already written for you.

/* put type definitions, including struct matrixCDT, here */

```
matrixADT NewMatrix(int numRows, int numCols)
```

```
void SetElementAt(matrixADT m, double value, int row, int col)
```

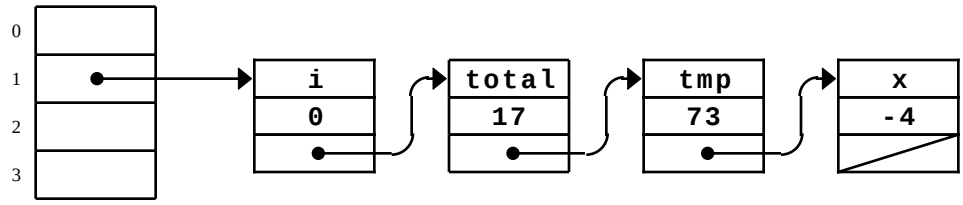
```
void PrintMatrix(matrixADT m)
```

Problem 5: Symbol tables

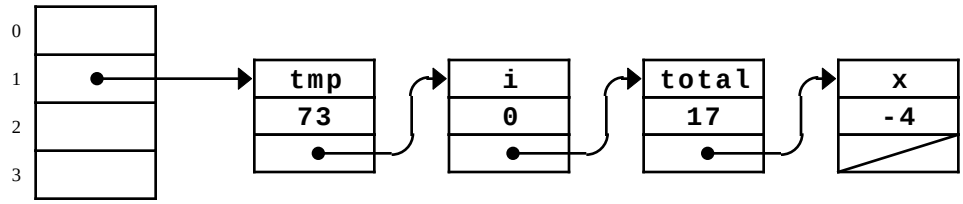
If you are using a hash table that contains more keys than there are buckets, the linked-list chains that hold the key/value pairs will get longer, which in turn decreases the efficiency of the **Enter** and **Lookup** operations. The rehashing strategy discussed in the text provides one solution to this, but is relatively costly in terms of execution time.

Given the pattern of references in a typical symbol table, you can often improve the performance of a hash table using a much simpler approach. In many applications, references to particular symbols are clustered in time, in the sense that several **Lookup** and **Enter** operations are likely to occur for the same key one after another. If such clustering occurs, you can reduce the cost of the symbol table references by moving the cell for that key to the beginning of its linked-list chain whenever it is referenced in either a **Lookup** or **Enter** call.

For example, suppose that you have a hash table in which bucket #1 has grown to include four key/value pairs, as follows:



Suppose the client now decides to call **Lookup** to find the key **"tmp"**. The hash-table implementation given in the text would leave the cell containing **"tmp"** as the third element in the linked-list chain. The essence of the proposed strategy is that you can probably improve performance by moving this cell to the front of the chain, like this:



After making this change to the list structure, subsequent calls to **Lookup** and **Enter** that refer to the key **"tmp"** will run much faster because that key is now at the beginning of the chain.

The easiest way to implement this strategy is to replace the calls to **FindCell** in the **Lookup** and **Enter** functions with the following call:

```
cp = FindAndReposition(table, bucket, key);
```

where the function **FindAndReposition** looks through the specified bucket chain for the entry matching **key**. If it finds it, the code should modify the link pointers in the chain so that the matching element becomes the first entry in the list. The return value is the same as for the old **FindCell** function: if a matching key is found, **FindAndReposition** returns a pointer to the corresponding cell; if not, the function returns **NULL**.

Starting with the code in Figure 11-2 on page 465, implement the **FindAndReposition** function as described.