CS1010E: Programming Methodology

Tutorial 08: Matrices

27 Mar 2017 - 31 Mar 2017

1. Discussion Questions

(a) [2D Traversal] Given a 2D array, there are 8 simple ways to *traverse* the array based on four direction available: 1) left-to-right (LR), 2) right-to-left (RL), 3) top-to-bottom (TB), and 4) bottom-to-top (BT). Since it is a 2D array, the traversal is a combination of the four directions.

A traversal that starts from top-left going to top-right before moving down one level until it ends up at bottom-right is called LRTB (*since it goes from left-to-right followed top-to-bottom*). The code below is a generic code for 2D array traversal. Assume that the size of the array is $N \times M$.

Fill in the blank for each of the traversal mechanism.

```
for(i=0; i<N; i++)
  for(j=0; j<M; j++)
    printf("%d ", matrix[i][j]); // This one is for LRTB
```

(b) [Matrix Modification; In-Place] In-place operations are operations that do not require additional array storage. Fill in the blank below for the intended in-place operations on $N \times N$ 2D array. You are given the following function:

```
void swap(int matrix, int i1, int j1, int i2, int j2) {
  int temp = matrix[i1][j1];
  matrix[i1][j1] = matrix[i2][j2];
  matrix[i2][j2] = temp;
}
```

i. Matrix Transpose (*reflection along main diagonal*)

```
for(i=0; i<N; i++)
  for(j=i+1; j<N; j++)
    swap(matrix, ____, ____, ____, ____);
```

ii. 90° Clockwise Matrix Rotation

```
for(i=0; i<N/2; i++) {
  for(j=i; j<N-i-1; j++) {
    swap(matrix,  T ,  R ,  T ,  L ); // top-right with top-left
    swap(matrix,  T ,  L ,  B ,  L ); // top-left with bottom-left
    swap(matrix,  B ,  L ,  B ,  R ); // bottom-left with bottom-right
  }
}
```

2. Program Analysis

(a) [Matrix Reasoning] What is/are the output of code fragments below?

i.
```
int matrix[5][5] = {
  {1, 2, 3, 4, 5},
  {1, 2, 3, 4, 5},
  {1, 2, 3, 4, 5},
  {1, 2, 3, 4, 5},
  {1, 2, 3, 4, 5},
}, i, j, N = 5, ans = 0;
for(i=0; i<N; i++)
  for(j=0; j<N; j++)
    if(i == j) break;
    else       ans += matrix[i][j];
printf("%d", ans);
```

i. 20 (lower triangle)

ii.
```
int matrix[5][5] = {
  {1, 1, 1, 1, 9},
  {1, 1, 1, 9, 1},
  {1, 1, 9, 1, 1},
  {1, 9, 1, 1, 1},
  {9, 1, 1, 1, 1},
}, i, j, N = 5, ans = 0;
for(i=0; i<N; i++)
  for(j=0; j<N; j++)
    if(j+i == N-1) continue;
    else           ans += matrix[i][j];
printf("%d", ans);
```

ii. 20 (exclude diagonal)

iii.
```
int matrix[5][5] = {
  {1, 9, 1, 9, 1},
  {9, 1, 9, 1, 9},
  {1, 9, 1, 9, 1},
  {9, 1, 9, 1, 9},
  {1, 9, 1, 9, 1},
}, N = 5, ans = 0, i = N/2, j = N/2, k = 0, x = -1, y = 1;
while(k<N) {
  if(i == j) ans += matrix[k][(i+j)/2];
  else       ans += matrix[k][i] + matrix[k][j];
  if(i==0) { x = 1; y = -1; }
  k++; i+=x; j+=y;
} printf("%d", ans);
```

iii. 8 (diamond shape)

(b) [Sort] What is/are the output of code fragments below?

i.
```
int S(int A[], int n, int k) {
  int l = 0, r = n-1, m, i = -1;
  while(i == -1 && l <= r) {
    m = (l + r) / 2;
    printf("%d ", A[m]);
    if(A[m] == k) i = m;
    if(A[m] <  k) l = m+1;
    if(A[m] >  k) r = m-1;
  } return i;
}
int main() {
  int arr[] = {1, 3, 5, 7, 9, 11, 13, 15, 17, 19};
  printf("%d", S(arr, 10, 17));
  return 0;
}
```

i. 9 15 17 8 (binary search)

(c) [Matrix Reasoning]What is/are the output of code fragments below?

i.
```
int C(int x, int y) {
  int mtrx[101][101] = {1}, i, j;
  for(i=1; i<=x+1; i++)
    for(j=1; j<=y+1; j++)
      mtrx[i][j] = mtrx[i-1][j-1] + mtrx[i-1][j];
  return mtrx[x+1][y+1];
}
int main() {
  printf("%d %d %d", C(3, 2), C(5, 3), C(7, 5));
  return 0;
}
```

i. 3 10 21 ($^3C_2$, $^5C_3$, $^7C_5$ via Pascal's triangle)

ii.
```c
void T(int arr[][4], int n) {
  int t, b, l, r, k; t = l = 0; b = r = n-1;
  while(n>0) {
    for(k=l; k<=r; k++) { printf("%d ", arr[t][k]); } t++;
    for(k=t; k<=b; k++) { printf("%d ", arr[k][r]); } r--;
    for(k=r; k>=l; k--) { printf("%d ", arr[b][k]); } b--;
    for(k=b; k>=t; k--) { printf("%d ", arr[k][l]); } l++;
    n -= 2;
  }
}
int main() {
  int matrix[4][4] = {
    { 1,  2,  3,  4},
    { 5,  6,  7,  8},
    { 9, 10, 11, 12},
    {13, 14, 15, 16},
  };
  T(matrix, 4); return 0;
}
```

ii. 1 2 3 4 8 12 16 15 14 13 9 5 6 7 11 10

iii.
```c
void T(int arr[][4], int si, int ei, int sj, int ej) {
  if(si == ei) { printf("%d", arr[si][sj]); return; }
  T(arr, si,          (si+ei)/2, sj,          (sj+ej)/2);
  T(arr, si,          (si+ei)/2, (sj+ej+1)/2, ej);
  T(arr, (si+ei+1)/2, ei,        sj,          (sj+ej)/2);
  T(arr, (si+ei+1)/2, ei,        (sj+ej+1)/2, ej);
}
int main() {
  int matrix[4][4] = {
    { 1,  2,  3,  4},
    { 5,  6,  7,  8},
    { 9, 10, 11, 12},
    {13, 14, 15, 16},
  };
  T(matrix, 0, 3, 0, 3); return 0;
}
```

iii. 1 2 5 6 3 4 7 8 9 10 13 14 11 12 15 16

3. Designing a Solution

(a) [Modularity] Magic Square is a special square of size $N \times N$ such that:

- It is filled with numbers in the range $[1, N^2]$ exactly **once**
- The sum of numbers in any row, any column, and any longest diagonals is exactly the same

This example shows how modular programming can help in developing a program to check for the magic square property. Assume that the maximum size of the magic square is $M \times M$, where M is defined separately.

i. Write a code to get the sum of row R. Use the template below:

```
int sumRow(int square[][M], int R, int n) {
  int sum = 0, i;
  /* square: square to be checked, R: row number, n: size of square */
  for(i=0; i<n; i++)
    sum += square[R][i];
  return sum;
}
```

ii. Write a code to get the sum of column C. Use the template below:

```
int sumCol(int square[][M], int C, int n) {
  int sum = 0, i;
  /* square: square to be checked, C: column number, n: size of square */
  for(i=0; i<n; i++)
    sum += square[i][C];
  return sum;
}
```

iii. Write a code to check for the first property of a Magic Square. Use the template below:

```
bool checkRange(int square[][M], int n) {
  /* square: square to be checked, n: size of square */
  int i, j, k, found = 0;
  for(k=1; k<=n*n; k++) { // Check all possible numbers: 1 to n^2
    found = 0;
    for(i=0; i<n; i++)    // in  all possible rows
      for(j=0; j<n; j++)  // and all possible columns
        if(square[i][j] == k) found++; // note how many times it is found
    if(found != 1) return false;       // it has to be found exactly once
  }
  return true;
}
```

iv. Given a 1D array of numbers, check if all the numbers within it are all equal? Use the template below:

```
bool isEqual(int arr[], int n) {
  /* arr: 1D array to be checked, n: size of array */
  int i;
  for(i=0; i<n-1; i++) // Check with neighbor only
    if(arr[i] != arr[i+1]) return false;
  return true;




}
```

v. Write the code to check if a square is a magic square. Use the template below:

```
bool isMagicSquare(int square[][M], int n) {
  /* square: square to be checked, n: size of square */
  int sums[2*M+2] = 0; // Collect the sums of rows, columns, and diagonals
  // Question: Why is the size 2*M+2?
  int i, j, k = 0, diag1sum = 0, diag2sum = 0;
  if(!checkRange(square, n)) return false; // Check first property
  for(i=0; i<n; i++) {// Get sums of rows, columns, and diagonals
    sums[k++] = sumRow(square, i, n); // Sum of rows
    sums[k++] = sumCol(square, i, n); // Sum of columns
    diag1sum += square[i][i];         // Sum of top-left to bottom-right
    diag2sum += square[i][n-i-1];     // Sum of top-right to bottom-left
  }
  sums[k++] = diag1sum; sums[k++] = diag2sum;
  return isEqual(sums, k); // Check second property




}
```

(b) [Matrix Modification] Given that we have a code to check if a square is a Magic Square, we go to a simpler problem: creating a magic square. There is a quick way to create an odd-size Magic Square using the algorithm described below:

Algorithm 1: Magic Square Construction for Odd Size Square

```
1  Let (x, y) be the index of the middle column of top row;
2  Let num be 1;
3  repeat
4      Insert num at cell[x][y];
5      Increment num by 1;
6      Let next index (xN, yN) be the cell diagonal upwards and to the right;
7      if goes beyond top row then
8          (xN, yN) overflows to the bottom row;
9      if goes beyond the right column then
10         (xN, yN) overflows to the left column;
11     if cell at the next index is already filled then
12         (xN, yN) is directly below index (x, y)
13     Assign (xN, yN) to (x, y);
14 until all cells are filled;
```

The illustration below shows the creation of a 3×3 Magic Square. Note that the value _ means that the value is currently not set.

```
_ 1 _    _ 1 _    _ 1 _    _ 1 _    _ 1 _    _ 1 6    _ 1 6    8 1 6    8 1 6
_ _ _ => _ _ _ => 3 _ _ => 3 _ _ => 3 5 _ => 3 5 _ => 3 5 7 => 3 5 7 => 3 5 7
_ _ _    _ _ 2    _ _ 2    4 _ 2    4 _ 2    4 _ 2    4 _ 2    4 _ 2    4 9 2
```

Write your program below:

```
void createMagicSquare(int square[][M], int n) {
  /* square: square to be checked, n: size of square to be created (always odd) */
  int i = 0, j = n/2, num = 0;
  for(i=0; i<n; i++)
    for(j=0; j<n; j++)
      square[i][j] = 0; // reset: just to make sure...
  while(num++<n*n) {
    square[i][j] = num; // Algorithm: line 4
    if(square[(n+i-1)%n][(j+1)%n] != 0)
      i = (i+1)%n;   // Go down
    else {
      i = (n+i-1)%n; // Go up
      j = (j+1)%n;   // Go right --> diagonal overall
    }
  }
}
```

4. Challenge

(a) [Simulation] A maze consists of either Wall or Road. The only rule is that you cannot pass through Wall. We will use the value `0` for Road and `1` for Wall. We say that two Road are connected if the two Road are adjacent to each other in the direction of up, down, left, or right. We say that there is a path from one point to another in the maze if there is a consecutive connected Road.

Write a program to check if there is a path from the top-left end of the maze to bottom-right end of the maze. Use the template below:

```
#define WALL 0    #define ROAD 1    #define MARK 2    #define DONE 3
bool isConnected(int maze[][M], int n) {
```

```
  /* maze: the maze in the description, n: size of maze */
  int i = 0, j = 0; bool hasPath = true;
  maze[0][0] = MARK; // initiate the marking process
  while(hasPath) {   // continue while continuation is found
    hasPath = false; // set to false, unless continuation is found
    for(i=0; i<n; i++)
      for(j=0; j<n; j++)
        if(maze[i][j] == MARK) { // if marked: mark the 4 directions
          if(i != 0   && maze[i-1][j] == ROAD) maze[i-1][j] = MARK; // up
          if(i != n-1 && maze[i+1][j] == ROAD) maze[i+1][j] = MARK; // down
          if(j != 0   && maze[i][j-1] == ROAD) maze[i][j-1] = MARK; // left
          if(j != n-1 && maze[i][j+1] == ROAD) maze[i][j+1] = MARK; // right
          maze[i][j] = DONE; hasPath = true; // mark as done, found a continuation
        }
    return maze[n-1][n-1] == DONE; // if this is marked, means connected
  }
```

```
}
```

Hint: At the starting point (0,0), you mark the neighbors. Now, consider all the points marked, this is the point connected to the starting point. How would you expand this set of point? Simple way is to expand from the marked points.

(b) [Mathematics] One practical use for matrix is an image. An image is simply a collection of pixels (*picture elements*). This makes an image to be a 2D array of pixel values. Image operations such as *blurring*, *sharpening*, or *egde detection* are simply operations on this image matrix.

The main operations on image is called *image convolution*. Image convolution uses two matrices: 1) image matrix and 2) kernel matrix. The kernel matrix is a linear operation matrix. Let's take a look at the example below for kernel operation.

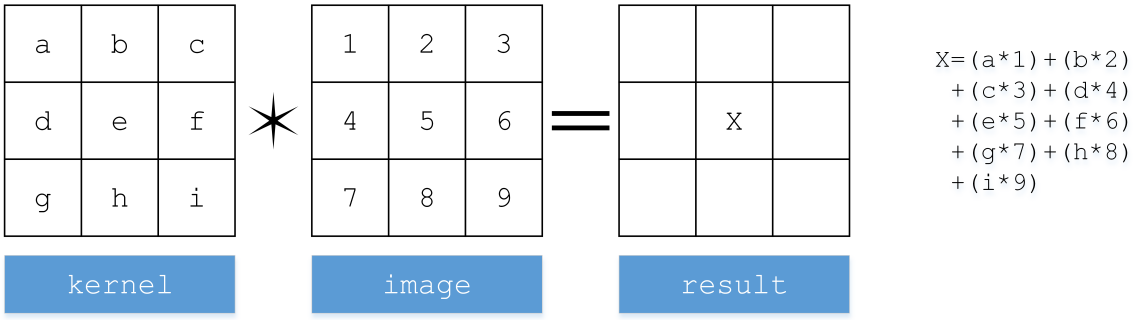

Diagram 1: Convolution of 3×3 kernel with 3×3 image. The result only affects the center pixel while the rest of the pixels are unchanged. The value of the middle pixel is shown on the right.

Images and kernel may be more than 3×3. In which case, the kernel is fitted into every part of the image such that all part of the kernel must be fully within the image. For instance, for a 5×5 image with a 3×3 kernel, the affected pixels are shown below in Red and Orange. The pixel in Orange is affected by the 3×3 square in green border. Note how the white pixel are not affected because the kernel will go out of the image.

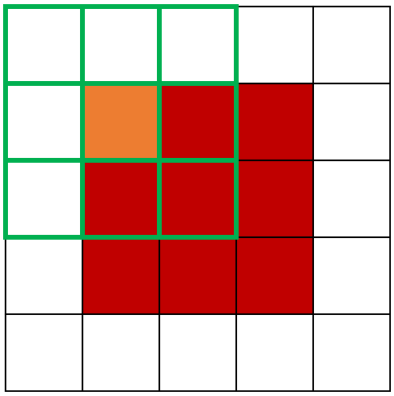

Diagram 2: Affected areas for convolution of 5×5 image by a 3×3 kernel.

An example of convolution process is shown below:

```
Kernel =  0  1  0     Image = 1 1 1 1 1   Result = 1  1  1  1  1
          1 -4  1             1 2 2 2 1            1 -2  0 -2  1
          0  1  0             1 2 3 2 1            1  0 -4  0  1
                              1 2 2 2 1            1 -2  0 -2  1
                              1 1 1 1 1            1  1  1  1  1
| edge detection kernel (4 directional egde detection)
| negative values are corner edges (only require sharp contrast in values)
```

Your job is to write a program to process an image with a kernel using image convolution. Assume that the maximum size of image is $N \times N$ and the maximum size of the kernel is $M \times M$, where $M < N$. Write your program below:

```
void convolution(int image[][N], int kernel[][M], int n, int m) {
  /* image: image, kernel: kernel, n: size of image, m: size of kernel */
  int i, j, x, y, temp[N][N] = 0, val;
  for(i=m/2; i<n-m/2; i++)
    for(j=m/2; j<n-m/2; j++)     // for every part of affected image
      for(x=-m/2; x<=m/2; x++)
        for(y=-m/2; y<=m/2; y++) // for every part of kernel
          temp[i][j] += kernel[x+m/2][y+m/2] * image[i+x][j+y];
  for(i=m/2; i<n-m/2; i++)
    for(j=m/2; j<n-m/2; j++)     // for every part of affected image
      image[i][j] = temp[i][j]
}
```