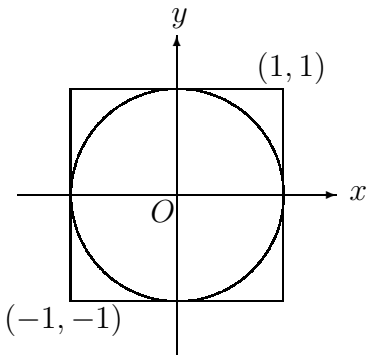
CS1010E Programming Methodology
Semester 1 2016/2017

Week of 24 October – 28 October 2016
Tutorial 9 Suggested Answers
Problem Solving with Arrays

1. A circle is inscribed in a square as shown in the following diagram. Let c be the area of the circle, s be the area of the square, and r be the ratio $\frac{c}{s}$. Suppose the value of π is unknown.



   (a) Express π in terms of r.
   $\pi = 4r$

   (b) You can determine the value of π if you can determine the value of r. You will use a computer simulation to approximate the value of r.
   Step 1. Generate a uniformly-distributed random point inside the square.
   Step 2. Determine if the point is inside the circle.
   Step 3. Repeat steps 1 and 2 for n trials.
   Step 4. Of these n trials, count the number of trials where the point occurred inside the circle. Let this number be m.
   Step 5. r is then approximated as $\frac{m}{n}$.

   Write a program to estimate the value of r and output an estimate for π.

   (a) Define a function `isInsideCircle` that takes a point as two floating point values and determines if the point is inside the circle. The function prototype is as follows:

   ```
   bool isInsideCircle(double x, double y);
   ```

   (b) Define a function `simulate` that takes in the number of simulation trials n, performs the simulation and returns the number of trials in which the point occurs within the circle.

   ```
   int simulate(int n);
   ```

   A sample run of the program is given below. User input is underlined.

```
Enter total number of random points, n: 1000000
The estimated value of pi is 3.142360.
```

```
#include <stdio.h>
#include <stdbool.h>
#include <stdlib.h>
#include <time.h>
bool isInsideCircle(double x, double y);
int simulate(int n);

int main() {
   int n, m;
   double r, pi;

   srand((unsigned int) time(0));
   printf("Enter total number of random points, n: ");
   scanf("%d", &n);
   m = simulate(n);
   r = 1.0 * m / n;
   pi = 4.0 * r;
   printf("The estimated value of pi is %f.\n", pi);
   return 0;
}

/*
   simulate runs the simulation for n trials.
   Precondition: none.
*/
int simulate(int n) {
   int k, count=0;
   double x, y;

   for (k = 1; k <= n; k++) {
      x = (2.0 * rand() / RAND_MAX) - 1.0;
      y = (2.0 * rand() / RAND_MAX) - 1.0;
      if (isInsideCircle(x,y))
         count++;
   }
   return count;
}

/*
   isInsideCircle takes in a point (x,y) and returns true if
   (x,y) is inside the circle of radius 1.0 and false otherwise.
   Precondition: none.
*/
bool isInsideCircle(double x, double y) {
   return (x*x + y*y) < 1.0;  /* no need to compute sqrt */
}
```

2. The birthday paradox is a classic elementary probability problem stated as follows:

> In a group of N randomly selected people, how large must N be so that there is more than 50% chance that at least two of them have the same birthday?

Since the chance of any two person having the same birthday is remote, many of us would expect N to be rather large. However, it turns out that this is not the case, and hence the paradox. *Try to make a rough guess of the value of N and see if you are anywhere near the answer.*

To make the simulation problem more interesting, the problem statement is generalized as follows:

> In a group of N randomly selected people, what is the probability P in which K of more person have the same birthday? Assume that nobody is born on the 29th of February.

We illustrate how a simulation is performed for $N = 5$ and $K = 2$. Pick any $N = 5$ people and check if there is at least one pair ($K = 2$) in the group having the same birthday. Then pick another group of five people and check again. By repeating this process for a number of times with different groups of five people, find the proportion of occurrences of any two persons having the same birthday.

(a) Declare an integer array `birthdays` in the `main` function to store the birthdays of a group of people. Think of how you can effectively use the array to find occurrences of same birthdays. You may assume that N will not exceed 1000.

(b) Write a function `fillBirthdays` that takes as argument the `birthdays` array and the value of N, and proceeds to fill the array with N randomized birthdays.

```
void fillBirthdays(int birthdays[], int N)
```

(c) Write a function `findSameBirthday` that takes as argument the `birthdays` array and the value K. The function returns true if there is an occurrence of K similar birthdays within the N elements.

```
bool findSameBirthday(int birthdays[], int k)
```

(d) Write the `main` function to read in the values of N and K from the user, followed by T, the number of simulation trials. Within each trial, fill the `birthdays` array with N random birthdays and test for the occurrence of K similar birthdays. Track the number of such occurrences over all T trials, and find the desired probability.

A sample run of the program is given below. User input is underlined.

```
Enter number of people, N: 23
Enter number of similar birthdays, K: 2
Enter number of simulation trials, T: 1000

The probability of 23 people having 2 similar birthdays is 0.509000.
```

```
#include <stdio.h>
#include <stdbool.h>
#include <stdlib.h>
#include <time.h>

void fillBirthdays(int birthdays[], int N);
bool findSameBirthday(int birthdays[], int k);

int main(void) {
   double prob;
   int N, K, trials, count=0, birthdays[365], i;

   srand((unsigned int) time(0));
   printf("Enter number of people, N: ");
   scanf("%d", &N);
   printf("Enter number of similar birthdays, K: ");
   scanf("%d", &K);
   printf("Enter number of simulation trials, T: ");
   scanf("%d", &trials);

   for (i = 0; i < trials; i++) {
      fillBirthdays(birthdays,N);
      if (findSameBirthday(birthdays,K)) {
         count++;
      }
   }
   prob = 1.0 * count / trials;
   printf("The probability of %d people having %d ", N, K);
   printf("similar birthdays is %f.\n", prob);

   return 0;
}
```

```
/*
    fillBirthdays fills N birthdays into the array birthdays
    by using a table-lookup method.  The indices 0 to 364
    represent the different birthdays, and the value of
    birthdays[k] is the number of people within the group of N
    having the same birthday k.

    Precondition: none.
*/
void fillBirthdays(int birthdays[], int N) {
   int i;

   /* initialize all to zero. */
   for (i = 0; i < 365; i++) {
      birthdays[i] = 0;
   }

   for (i = 0; i < N; i++) {
      (birthdays[rand()%365])++;
   }

   return;
}

/*
    findSameBirthday returns true if there are k (or more)
    same birthdays, and false otherwise.

    Precondition: none.
*/
bool findSameBirthday(int birthdays[], int k) {
   int i;

   for (i = 0; i < 365; i++) {
      if (birthdays[i] >= k) {
         return true;
      }
   }
   return false;
}
```

3. The *bubble sort* algorithm makes a series of pairwise checks on the array elements, and performs swaps if necessary. During each pass of bubble sort, the first and second elements are checked to see if the former element is larger; if so, the elements are swapped. Then the second and third elements are checked and swapped, if necessary. Pairwise checking is performed all the way until the second last and last elements.

   - Original order:
     5 3 12 8 1 9
   - The first and second element are out of order:
     3 5 12 8 1 9
   - The second and third element are in order:
     3 5 12 8 1 9
   - The third and fourth element are out of order:
     3 5 8 12 1 9
   - The fourth and fifth element are out of order:
     3 5 8 1 12 9
   - The fifth and sixth element are out of order:
     3 5 8 1 9 12

   The above completes the first pass of bubble sort. With subsequent passes,

   - At the end of the first pass:
     3 5 8 1 9 12
   - At the end of the second pass:
     3 5 1 8 9 12
   - At the end of the third pass:
     3 1 5 8 9 12
   - At the end of the fourth pass:
     1 3 5 8 9 12
   - At the end of the fifth pass:
     1 3 5 8 9 12

   the array becomes sorted. Write a function `bubbleSort` that takes in an integer array x and sorts the first n elements of x in ascending order using bubble sort.

   ```
   void bubbleSort(int x[], int n);
   ```

   Start with the most rudimentary bubble sort algorithm and improve on it by considering efficiency issues. Test the different implementations by devising a suitable `main` function.

```
#include <stdio.h>
#include <stdbool.h>

#define SIZE 6

void printList(int x[], int n);
void bubbleSort1(int x[], int n);
void bubbleSort2(int x[], int n);
void bubbleSort3(int x[], int n);

int main(void) {
   int list[SIZE] = {5,3,12,8,1,9};

   printList(list,SIZE);
   bubbleSort3(list,SIZE); /* use bubbleSort1, bubbleSort2 or bubbleSort3 */
   printList(list,SIZE);
   return 0;
}

/*
   bubbleSort1 performs bubble sort on first n elements of array x
   using n-1 passes, with n-1 pairwise comparisons in each pass.

   Precondition: x contains at least n valid elements.
*/
void bubbleSort1(int x[], int n) {
   int i, j, temp;
   for (i = 0; i < n-1; i++) {
      for (j = 1; j < n; j++) {
         if (x[j-1] > x[j]) {
            temp = x[j];
            x[j] = x[j-1];
            x[j-1] = temp;
         }
      }
   }
   return;
}

/*
   bubbleSort2 performs bubble sort on first n elements of array x
   using n-1 passes, with progressively lesser pairwise comparisons
   in each pass.

   Precondition: x contains at least n valid elements.
*/
void bubbleSort2(int x[], int n) {
   int i, j, temp;
```

```
   for (i = 0; i < n-1; i++) {
      for (j = 1; j < n-i; j++) {
         if (x[j-1] > x[j]) {
            temp = x[j];
            x[j] = x[j-1];
            x[j-1] = temp;
         }
      }
   }
   return;
}

/*
   bubbleSort3 performs bubble sort on first n elements of array x.
   Sorting terminates when the most recent pass does not incur any
   swaps, i.e. the array is already sorted.

   Precondition: x contains at least n valid elements.
*/
void bubbleSort3(int x[], int n) {
   int j, temp;
   bool swapped;

   do {
      swapped = false;
      for (j = 1; j < n; j++) {
         if (x[j-1] > x[j]) {
            temp = x[j];
            x[j] = x[j-1];
            x[j-1] = temp;
            swapped = true;
         }
      }
      n--;
   } while (swapped);
   return;
}

void printList(int x[], int n) {
   int i;

   for (i = 0; i < n; i++)
      printf("%d ", x[i]);
   printf("\n");
   return;
}
```

4. We continue the explorations of Conway's Game of Life in two-dimensional form. A population is represented by a two-dimensional grid of 24 rows and 60 columns where each cell represents an organism. Each cell may be occupied (alive) or unoccupied (dead). For each generation, the occupancy of a cell depends on its own previous state and the previous states of eight of it's neighbouring cells. Conway's original rule is very simple.

   - If a cell is unoccupied in one generation, it will become occupied only if it has three neighbours; otherwise it remains dead.
   - If a cell is occupied in one generation, it will stay alive only if it has two or three neighbours; otherwise it dies due to loneliness (one neighbour or less) or overcrowding (four or more neighbours).

   Write a program to simulate the *Game-of-Life*. You will first need to set the window size to 25 rows (the default in cygwin). Use the following as a guide to implement your program. The following constants are defined.

```
#define ROWS 24
#define COLS 60
#define ALIVE 1
#define DEAD  0
```

   (a) Write a `main` function to implement the following:

      i. Declare a two-dimensional array `grid` of size `ROWS` $\times$ `COLS` and initialize all elements to `DEAD`.

      ii. Read the number of generations, `numGen`. followed by a sequence of integer pairs representing the (row,column) location of an occupied cell and initialize `grid` accordingly by setting the corresponding cells to `ALIVE`.
      A sample input file `first.txt` is given with the following contents.

```
50
0 1
1 0
1 1
3 3
3 4
3 5
```

      In order to redirect this file to the program using

```
./a.out < first.txt
```

      the method of reading until the end-of-file marker is used. Use the following program fragment and adapt it to suit your needs.

```
scanf("%d", &numGen);
while(scanf("%d%d", &i, &j) != EOF) {
   printf("%d %d\n", i, j);
}
```

(b) Write a function `countNeighbours` to count the number of neighbours of a cell centered at `i` and `j`.

```
int countNeighbours(int grid[][COLS], int i, int j);
```

To check that a neighbour is within the bounds, write a function `isValid` to determine if indices `i` and `j` are within the `grid` array.

```
int isValid(int i, int j);
```

(c) Write a function `generate` to run the game of life for one generation according to Conway's rules.

```
void generate(int grid[][COLS], int rows, int cols);
```

You may write other helper methods to further modularize your function.

(d) Modify the `main` function in part 4a so as to simulate the game of life for a number of generations using the initial configuration of the input file. In order to improve the aesthetics of the output, the following function `printGrid` is given to print the cells. The function prints a `*` when a cell is `ALIVE`. Otherwise, a blank is printed.

```
void printGrid( int grid[][COLS], int rows, int cols ) {
   int i, j;

   if(system("clear")) {
      system("cls");
   }
   for (i = 0; i < rows; i++) {
      for (j = 0; j < cols; j++) {
         if (grid[i][j] == ALIVE)
            printf("*");
         else
            printf(" ");
      }
      printf("\n");
   }
   return;
}
```

With the file `first.txt`, you will see the screen flicker between the two screenshots below.

```
**
**
    *
    *
    *
```

```
**
**

   ***
```

Test your program with the two files `glider.txt` and `gliderGun.txt` and observe the animation. Have fun!

```
#include <stdio.h>
#define ROWS 24
#define COLS 60
#define ALIVE 1
#define DEAD 0

void copyGrid(int dst[][COLS], int src[][COLS], int rows, int cols);
void generate(int grid[][COLS], int rows, int cols);
int isValid(int i, int j);
int countNeighbors( int grid[][COLS], int i, int j);
void printGrid( int grid[][COLS], int rows, int cols);

int main(void) {
   int numGen, i, j, grid[ROWS][COLS]={{0}};

   scanf("%d", &numGen);
   while (scanf("%d%d", &i, &j) != EOF) {
      grid[i][j] = ALIVE;
   }

   printGrid(grid, ROWS, COLS);
   for (i = 0; i < numGen; i++) {
      generate(grid, ROWS, COLS);
      printGrid(grid, ROWS, COLS);
   }

   return 0;
}

void printGrid( int grid[][COLS], int rows, int cols ) {
   int i, j;

   if(system("clear")) {
      system("cls");
   }
   for (i = 0; i < rows; i++) {
      for (j = 0; j < cols; j++) {
         if (grid[i][j] == ALIVE)
            printf("*");
         else
            printf(" ");
      }
      printf("\n");
   }
   return;
}
```

```
void generate(int grid[][COLS], int rows, int cols) {
   int nextGen[ROWS][COLS] = {{0}}; /* next generation */
   int i, j, numNeighbours;

   for (i = 0; i < rows; i++) {
      for (j = 0; j < cols; j++)
      {
         numNeighbours = countNeighbors(grid, i, j);
         if (grid[i][j] == ALIVE) {
            if (numNeighbours < 2 || numNeighbours > 3)
               nextGen[i][j] = DEAD;
            else
               nextGen[i][j] = ALIVE;
         } else {
            if (numNeighbours == 3)     /* reproduction */
               nextGen[i][j] = ALIVE;
            else                        /* stay dead */
               nextGen[i][j] = DEAD;
         }
      }
   }
   copyGrid(grid, nextGen, rows, cols);
   return;
}

int countNeighbors( int grid[][COLS], int i, int j ) {
   int x, y, count = 0;

   for ( x = i-1; x <= i+1; x++) {
      for ( y = j-1; y <= j+1; y++) {
            if (isValid(x,y) && (x != i || y != j)) {
               if (grid[x][y] == ALIVE) {
                  count++;
               }
            }
      }
   }
   return count;
}

int isValid(int i, int j) {
    return i >= 0 && i < ROWS && j >= 0 && j < COLS;
}
```

```
void copyGrid(int dst[][COLS], int src[][COLS], int rows, int cols) {
   int i, j;

   for ( i = 0; i < rows; i++) {
      for ( j = 0; j < cols; j++) {
         dst[i][j] = src[i][j];
      }
   }
   return;
}
```

A useful trick to ensure that array access will always be within bounds is to include "buffer" rows at the top, bottom, left and right of the grid, i.e. the grid is now declared as

```
int grid[ROWS+2][COLS+2] = {DEAD};
```

Checking individual cells can span from the top left corner `grid[1][1]` to the bottom right corner `grid[ROW][ROW]`. This way when checking for neighbours of boundary locations, there is no need to worry about out-of-bounds array access. Modify the program to include the "buffer" rows and dispense away with the `isValid` function. You would also need to tweak the function headers slightly.