Starry Night and Sudoku

Lab #4 (CS1010 AY2013/4 Semester 1)

Date of release: 4 October 2013, Friday, 9:00hr.

Submission deadline: 19 October 2013, Saturday, 9:00hr.

School of Computing, National University of Singapore

0 Introduction

Important: Please read Lab Guidelines before you continue.

This lab requires you to do 2 exercises. The main objective of this lab is on the use of arrays to solve problems.

The maximum number of submissions for each exercise is **15**.

If you have any questions on the task statements, you may post your queries **on the relevant IVLE discussion forum**. However, do **not** post your programs (partial or complete) on the forum before the deadline!

Important notes applicable to all exercises here:

- You should take the "Estimated Development Time" seriously and aim to complete your programming within that time. Use it to gauge whether your are within our expectation, so that you don't get surprised in your PE. We advise you to do the exercises here in a simulated test environment by timing yourself.

- Please do **not** use variable-length arrays. An example of a variable-length array is as follows:
  ```
  int i;
  int array[i];
  ```
  This is not allowed in ANSI C, as explained in Week 7 lecture. Declare an array with a known maximum size. We will tell you the maximum number of elements in an array.

- Note that you are **NOT allowed to use recursion** for the exercises here. Hold on your "recursive streak" till lab #5. Using recursion here would amount to violating the objective of this lab assignment.

- You are **NOT allowed to use global variables**. (A global variable is one that is not declared in any function.)

- You are free to introduce additional functions if you deem it necessary. This must be supported by well-thought-out reasons, not a haphazard decision. By now, you should know that you **cannot write a program haphazardly**.

- In writing functions, we would like you to include function prototypes before the main function, and the function definitions after the main function.

- As mentioned in Week 7 lecture "Section 8 Using UNIX input redirection", you may consider entering the input data in a file and then use UNIX input redirection to feed the data into your programs.

1 Exercise 1: Estimating Pi

1.1 Learning objectives

- Problem solving on array.
- Using external function.
- Writing function.

1.2 Task statement

This problem is adopted from National Software Competition 2006 for junior college students. (Copyright: NSC 2006.)

Professor Robert A.J. Matthews of the Applied Mathematics and Computer Science Department at the University of Aston in Birmingham, England, has recently described how the positions of stars across the night sky may be used to deduce a surprisingly accurate value of π. This result followed from the application of certain theorems in number theory.

Here, we don't have the night sky, but we can use the same theoretical basis to form an estimate for π.

Given any pair of positive integers chosen from a large set of random numbers, the probability that these two integers having no common factor other than one is $\mathbf{6/\pi^2}$.

For example, using this small set of five numbers {2, 3, 4, 5, 6}, there are 10 pairs that can be formed: (2,3), (2,4), (2,5), (2,6), (3,4), (3,5), (3,6), (4,5), (4,6) and (5,6). Six of these 10 pairs - (2,3), (2,5), (3,4), (3,5), (4,5), and (5,6) - have no common factor other than one. Using the ratio of the counts as the probability we have:

$6/\pi^2 = 6/10$. Hence the estimated value of π is 3.1623, correct to four decimal places.

As another example, given this set of 10 numbers {32391, 14604, 3902, 153, 292, 12382, 17421, 18716, 19718, 19895}, there are 24 pairs that have no common factor other than one, among a total of 45 pairs. We have:

$6/\pi^2 = 24/45$. Hence the estimated value of π is 3.3541, correct to four decimal places.

Write a program **`estimatePi.c`** that reads in a positive integer *n* representing the size of the list, followed by *n* unique positive integers (representing the random numbers). Your program then prints out an estimate value for π (using **`double`** type) accurate to 4 decimal places. The set contains at most 50 unique positive integers.

You will be given two files: **`gcd.h`** which you need to include in your program, and **`gcd.o`** which you need to compile together with your program. The **`gcd.o`** object file contains the GCD function which you are required to use in your program.

1.3 Sample runs

Sample runs using interactive input (user's input shown in blue; output shown in **bold purple**). Note that the first two lines (in green below) are commands issued to compile and run your program on UNIX.

```
$ gcc -Wall -lm estimatePi.c gcd.o -o estimatePi
$ estimatePi
3
7 4 10
Estimated pi = 3.0000
```

Sample run #2:

```
5
2 3 4 5 6
Estimated pi = 3.1623
```

Sample run #3:

```
10
32391 14604 3902 153 292 12382 17421 18716 19718 19895
Estimated pi = 3.3541
```

1.4 Skeleton program and Test data

- The skeleton program is provided here: estimatePi.c
- The header file and the object file for the GCD function are provided here: gcd.h and gcd.o
  You should put these files in the same directory as `estimatePi.c`.
- Refer to Week 6 lecture notes (Section 8 Separate Compilation) on how to compile your program with `gcd.o`. The command is also given in the first sample run above.
- Do **not** write your own **`gcd`** function. You must use the given `gcd.o`. We want you to understand how separate compilation is done.
- Note that the object file `gcd.o` has been compiled in sunfire, so it only works in sunfire.
- Test data: Input files | Output files

1.5 Important notes

- The set contains at most 50 unique positive integers.
- You should write a function **`pi(int arr[], int size)`** to take in an integer array **`arr`** that contains the values, and the number of values, **`size`**, in that array. The function should return the estimated value of π.
- Do **not use any additional array** besides the one that holds the given values.

1.6 Estimated development time

The time here is an estimate of how much time we expect you to spend on this exercise. If you need to spend way more time than this, it is an indication that some help might be needed.

- Devising and writing the algorithm (pseudo-code): 15 minutes
- Translating pseudo-code into code: 10 minutes
- Typing in the code: 10 minutes
- Testing and debugging: 10 minutes
- **Total: 45 minutes**

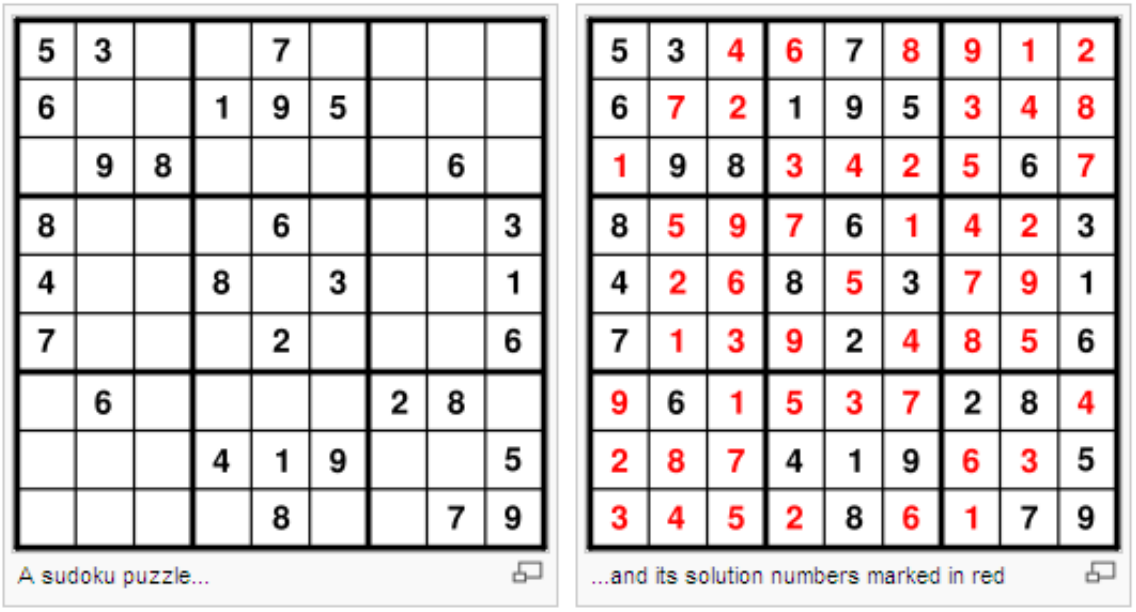
2 Exercise 2: Mini-Sudoku

2.1 Learning objectives

- 2-dimensional array
- Writing a popular game solver (albeit using a very simple algorithm on a mini version of the game)

2.2 Task

Sudoku is a popular logic-based number placement puzzle. The 9×9 board has 9 rows, 9 columns and 9 sections of 3×3 cells. The objective is to fill the board so that each row, each column and each section contains the digits from 1 to 9. There is only one unique solution. Figure 1 below, taken from Wikipedia: Sudoku, shows a Sudoku puzzle and its solution. (There are numerous websites on Sudoku, surf the Internet to explore!)



5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

A sudoku puzzle...

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

...and its solution numbers marked in red

Figure 1. A Sudoku puzzle and its solution.

In this exercise, we shall solve mini-Sudoku puzzles on a **4×4 board**. The board consists of digits from 0 to 4 where 0 represents a blank cell. The solver needs to replace all the 0s with the correct values (1 - 4).

A skeleton program is given and you must complete all the given functions. You may write additional function(s) if necessary. (You would probably write additional functions as the code is complex and making it modular would help you plan and debug.)

We will only give you the simplest Sudoku puzzles to solve: these puzzles are such that at any time, there is at most one blank cell (0) in a certain row, a column, or a 2×2 section. Hence, you are able to fill in the blank cells progressively until the whole board is filled.

One very simple algorithm for `solve()` is described below. We shall call it **algorithm *A***.

> ***Repeat the following until no more blank cells can be filled:***
>
> > ***For each row, check whether there is a single 0. If so, replace that 0 with the obvious value.***
> >
> > ***For each column, check whether there is a single 0. If so, replace that 0 with the obvious value.***
> >
> > ***For each 2×2 section, check whether there is a single 0. If so, replace that 0 with the obvious value.***

We will use an example to illustrate **algorithm *A***. Figure 2a below shows a puzzle with seven blank cells, labelled from A to G for easy reference.

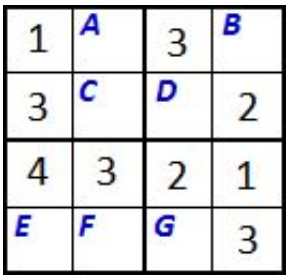


Figure 2a. A Sudoku puzzle.

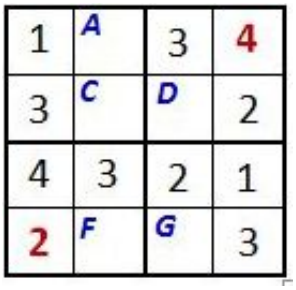


Figure 2b.

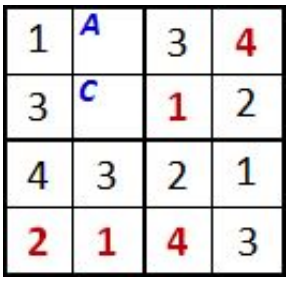


Figure 2c.

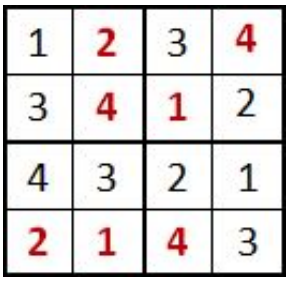


Figure 2d. The solution.

After examining each row, we cannot fill in any blank cell as none of the rows has a single blank cell.

We proceed to examine every column. In the first column, we find that we can fill E with 2, and in the fourth column, B with 4, as shown in Figure 2b.

We proceed to examine every 2×2 section. We find that we can fill D with 1, F with 1, and G with 4. See Figure 2c.

The puzzle is still not solved, so we repeat the process. As we examine the rows this time, we find that we are able to fill A with 2, and C with 4. This gives the solution as shown in Figure 2d.

There are many possible algorithms to solve Sudoku puzzles. Let me show you another one below (let's call it **algorithm *B***).

> ***Repeat the following until no more blank cells is found:***
>
> > ***Scan the board from top to bottom, and for each row from left to right.***
> > ***For each blank cell encountered:***
> >
> > > ***Examine the row, column and section the blank cell is in,***
> > > ***if only one value for that blank cell fits, fill it with that value.***

As **algorithm *B*** is more general than **algorithm *A***, it is able to solve puzzles that the latter could not. For example, Figure 3 is one puzzle that can be solved by **algorithm *B*** but not by **algorithm *A***. The solution is the same as Figure 2d. Of course, as a simple algorithm, **algorithm *B*** still does not solve all Sudoku puzzles (which is beyond the scope of CS1010.)

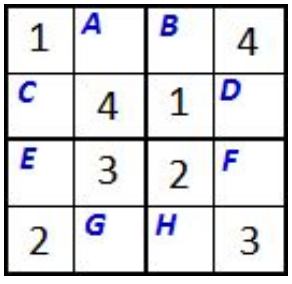


Figure 3. Another Sudoku puzzle.

However, as stated above, all puzzles given for this exercise are such that at any time you are able to find a row, column, or 2×2 section with only at most one blank cell, so puzzles like Figure 3 will not be given. Hence, you need only implement **algorithm *A***, and not **algorithm *B*** or any other more complex algorithm.

(Please submit only **algorithm *A*** so that your DL can focus on checking its correctness. You may implement **algorithm *B*** or other more complex algorithm on your own for practice purpose.)

2.3 Sample runs

Sample run using interactive input (user's input shown in blue; output shown in **bold purple**). Note that the first two lines (in green below) are commands issued to compile and run your program on UNIX.

```
$ gcc -Wall sudoku.c -o sudoku
$ sudoku
Enter board (0 for blank cell):
1 0 3 0
3 0 0 2
4 3 2 1
0 0 0 3
Sudoku puzzle solved:
1 2 3 4
3 4 1 2
4 3 2 1
2 1 4 3
```

Second sample run:

```
$ sudoku
Enter board (0 for blank cell):
0 1 3 2
2 0 1 0
1 0 0 3
3 4 2 1
Sudoku puzzle solved:
4 1 3 2
2 3 1 4
1 2 4 3
3 4 2 1
```

2.4 Skeleton program and Test data

- The skeleton program is provided here: sudoku.c
- Test data: Input files | Output files

2.5 Important notes

- Do **NOT** use recursion.
- Test your program thoroughly with different inputs.
- Ensure that the output of your program comforms to the output format. Note that in the output of the board, every digit is followed by a space character. Hence the first line of the completed board for the first sample run is "**1 2 3 4** ".

2.6 Estimated development time

The time here is an estimate of how much time we expect you to spend on this exercise. If you need to spend way more time than this, it is an indication that some help might be needed.

This exercise is tedious to code. Please brace yourself for long hours of preparation, coding and testing!

- Devising and writing the algorithm (pseudo-code): 0 minute (given)

- Translating pseudo-code into code: 1.5 hours
- Typing in the code: 30 minutes
- Testing and debugging: 2 to 4 hours (or even more!)
- Total: 4 to 6 hours

2.7 Exploration

This is for your own practice and is not to be submitted.

Write a more general solver to solve more complex 4×4 Sudoku puzzles. Then extend your algorithm to the original 9×9 Sudoku board. Have fun!

3 Deadline

The deadline for submitting all programs is **19 October 2013, Saturday, 9:00hr**. Late submission will NOT be accepted.

Aaron
Sunday, September 29, 2013 02:23:32 PM SGT