Starry Night, Subsequence and Minesweeper

Lab #4 (CS1010 AY2011/2 Semester 4)

Date of release: 16 July 2012, Monday, 14:00hr.

Submission deadline: 21 July 2012, Saturday, 23:59hr.

School of Computing, National University of Singapore

0 Introduction

Important: Please read Lab Guidelines before you continue.

This lab requires you to do 3 exercises. The main objective of this lab is on the use of arrays to solve problems.

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. 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.

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 p. 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 p.

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 **$6/p^2$**.

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:

$6/p^2$ = 6/10. Hence the estimated value of p 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:

$6/p^2$ = 24/45. Hence the estimated value of p 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* lines of unique positive integers (representing the random numbers). Your program then prints out an estimate value for p (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 Number of submission

For this exercise, the number of submission is **20**.

1.5 Important notes

- 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
  Refer to Week 6 lecture notes on how to compile your program with `gcd.o`
  Do **not** write your own **`gcd`** function. You must use the given `gcd.o`.
- The object file `gcd.o` has been compiled in sunfire. If you are testing your program outside sunfire, it will not work. You may use this C program gcd.c and compile it into `gcd.o` that works in your platform. However, do not submit this `gcd.c`.
- 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 p.
- 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: Maximum Subsequence Sum

2.1 Learning objectives

- Problem solving on array.
- Writing function.

2.2 Task

Given a list *A* with *n* values A_0, A_1, A_2, ..., A_{n-1}, we define a contiguous subsequence, or we shall simply call it *subsequence* in this exercise, to be A_j, A_{j+1}, ..., A_k, where $0 \le j$, $k < n$, and $j \le k$.

For example, if *A* is { 5, -12, 9, 10, -4, 7, -8, 6 }, then { -12, 9, 10, -4 }, { 10, -4, 7 }, { 5, -12, 9 } and { 10 } are some subsequences of *A*, while { 5, 9 } and { 9, 10, -8 } are not subsequences of *A*.

The value of a subsequence, or *subsequence value*, is the sum of the values in the subsequence. For example, { -12, 9, 10, -4 } has a value of 3. By convention, an empty subsequence { } has a value of 0.

Write a program **`sequence.c`** that reads in a positive integer *n* representing the length of the sequence, followed by *n* integers. Your program then computes the maximum subsequence sum.

For example, for our sequence *A* above, the maximum subsequence sum is **22** and the subsequence that gives this sum is { 9, 10, -4, 7 }.

You are to include a function **`int maxSubseqSum(int arr[], int size)`** that returns the maximum subsequence sum. You are also to explain the logic of the algorithm using comments.

You may assume that the sequence contains at most 15 integers.

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 sequence.c -o sequence
$ sequence
8
5 -12 9 10 -4 7 -8 6
Maximum subsequence sum = 22
```

Sample run #2:

```
10
-1 -2 -3 -4 -5 -6 -7 -8 -9 -10
Maximum subsequence sum = 0
```

2.4 Number of submission

For this exercise, the number of submission is **20**.

2.5 Important notes

- The skeleton program is provided here: sequence.c
- The sequence contains at most 15 integers.
- This is a problem where a good algorithm (using a well-written single loop) is more superior than a straight-forward algorithm that uses doubly-nested loop. For this exercise, we have set aside some marks to be awarded only to good algorithms. Do not worry if you could only come out with an algorithm that uses doubly-nested loop. The marks difference is small and it should not affect the attempt mark, if you also make a reasonable attempt at the other two exercises.
- Further to above, you are required to **explain the logic of your algorithm** using comments. 10 marks has been allocated to this requirement.
- Do not discuss the algorithm in public before the deadline, in view of the above. You may request your DL to discuss it in class after the deadline.

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.

- Devising and writing the algorithm (pseudo-code): 40 minutes
- Translating pseudo-code into code: 10 minutes
- Typing in the code: 10 minutes
- Testing and debugging: 40 minutes
- **Total: 1 hour 40 minutes**

3 Exercise 3: Minesweeper

3.1 Learning objectives

- Problem solving on 2-dimensional arrays.
- Nested loops.
- Writing functions.

3.2 Task statement

Minesweeper is a computer game whose objective for the player is to clear a minefield without detonating a mine. The player is presented with a grid of squares (Figure 1) under which some contain mines, and some do not.

For a square that is safe (mine-free), it contains an integer value (0 to 8) indicating the number of mines surrounding it.

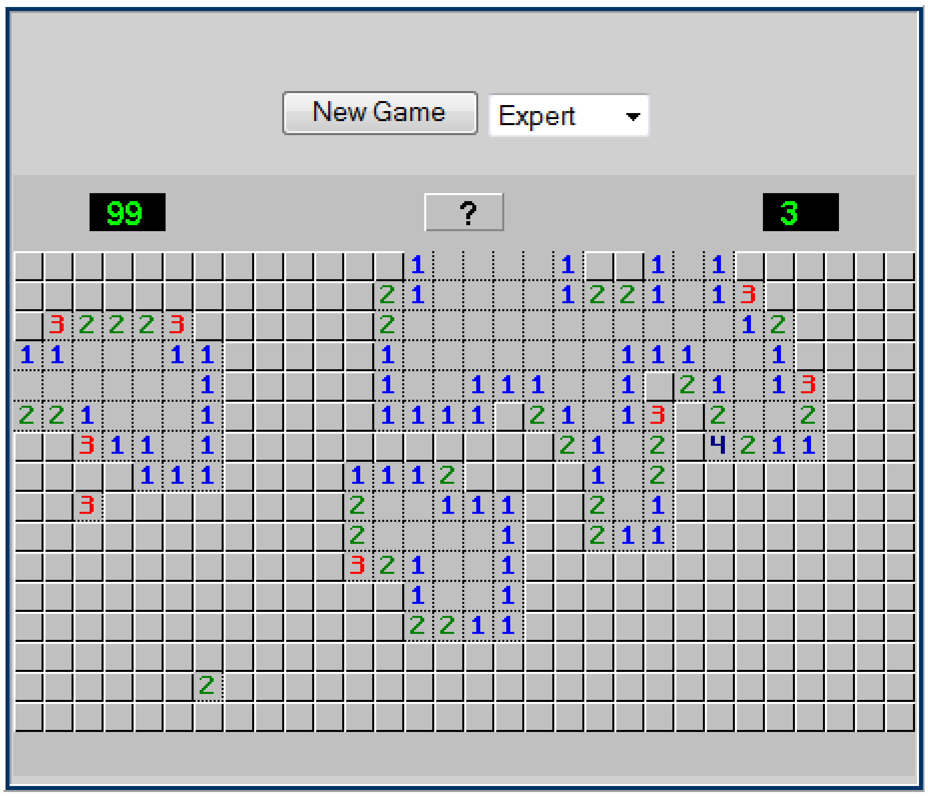


Figure 1. A Minesweeper game in progress.

A player clicks on a square to turn it over. If it contains a mine, the game ends and all the mines revealed (Figure 2). The game also ends when the player turned over all safe squares. Usually, a square that contains 0 (no mine around it) is simply displayed as a blank square.

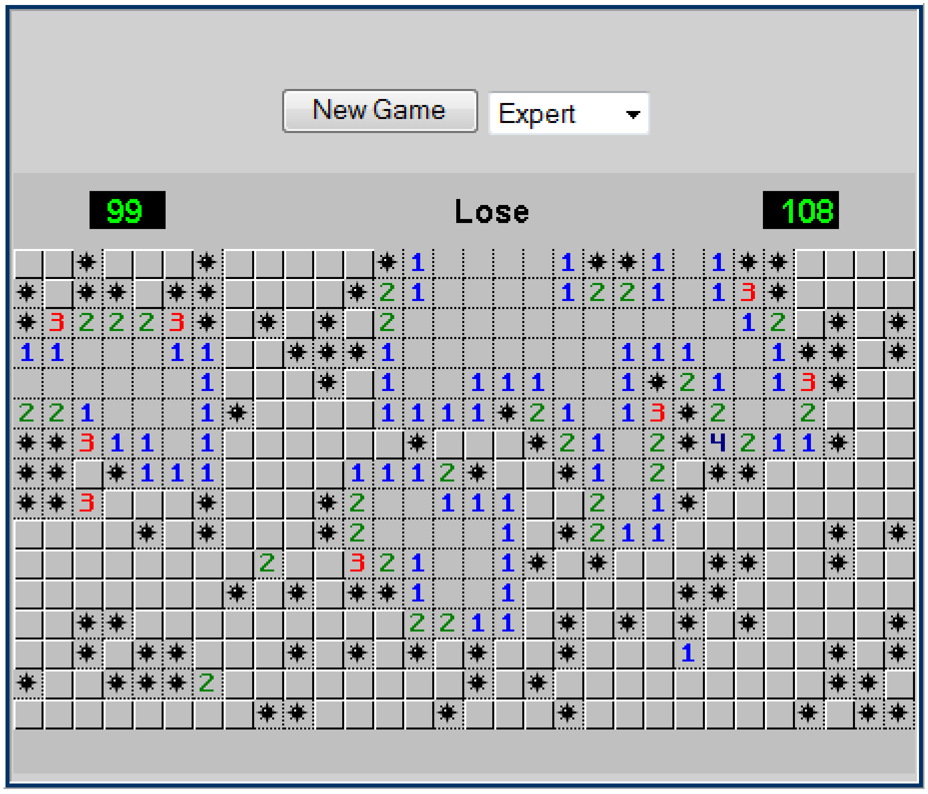


Figure 2. The Minesweeper game ends when player clicks on a square containing a mine.

You may try the game on GamesWizard.com and many other websites on the Internet.

For this exercise, you are only required to prepare the grid before the start of the game. Given the positions of all the mines in the grid, you are to fill in the numbers 0 to 9 in each of the safe squares.

For example, Figure 3 shows the positions of the mines in a minefield. You are to compute the values of the safe squares, as shown in Figure 4.

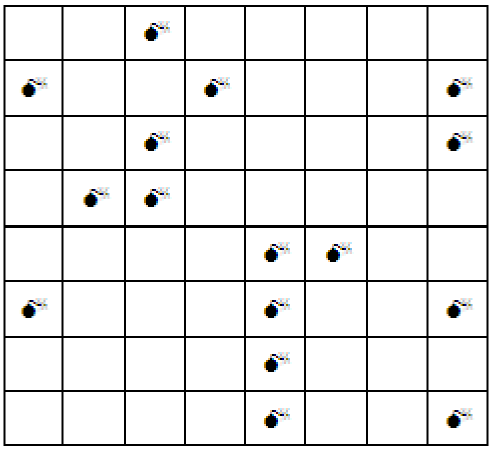
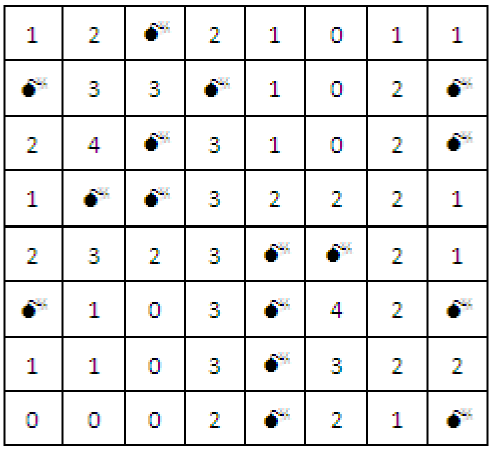


Figure 3. Positions of mines.



1	2	💣	2	1	0	1	1
💣	3	3	💣	1	0	2	💣
2	4	💣	3	1	0	2	💣
1	💣	💣	3	2	2	2	1
2	3	2	3	💣	💣	2	1
💣	1	0	3	💣	4	2	💣
1	1	0	3	💣	3	2	2
0	0	0	2	💣	2	1	💣

Figure 4. Values in safe squares.

Write a program **`minesweeper.c`** to read in a minefield containing the mines, and compute the values of the safe squares in a numeric 2-dimensional array.

There are 3 game levels, with each having the following grid dimension:

- Level 1: 8 × 8 grid
- Level 2: 12 × 16 grid
- Level 3: 16 × 30 grid

Your program is to read the game level, and then the grid containing characters comprising either **-** (mine-free) or ***** (a mine). Your program then outputs a 2-dimensional integer array showing the number of mines surrounding each square. If the square in the grid contains a mine, then its corresponding value in the integer array is 9.

3.3 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 minesweeper.c -o minesweeper
$ minesweeper
1
--*-----
*--*---*
--*----*
-**-----
----**--
*---*--*
----*---
----*--*
 1 2 9 2 1 0 1 1
 9 3 3 9 1 0 2 9
 2 4 9 3 1 0 2 9
 1 9 9 3 2 2 2 1
 2 3 2 3 9 9 2 1
 9 1 0 3 9 4 2 9
 1 1 0 3 9 3 2 2
 0 0 0 2 9 2 1 9
```

Sample run #2:

```
2
--*-----*--*---*
--*----*-**-----
```

```
____**__*___*__*
____*_______*__*
**_____*___***__
*____*____*___*_
*__**_________**
__*__*_*______**
______***_______
___*_*_____***__
*___*__________*
___*_____*_____*
 0 2 9 2 0 0 1 2 9 3 3 9 1 0 1 9
 0 2 9 3 2 2 2 9 4 9 9 3 2 1 2 2
 0 1 1 3 9 9 2 2 9 3 2 3 9 2 2 9
 2 2 1 2 9 3 2 2 2 1 1 4 9 4 3 9
 9 9 1 1 2 2 2 9 1 1 2 9 9 9 3 2
 9 4 2 2 3 9 2 1 1 1 9 3 3 4 9 3
 9 3 2 9 9 3 3 1 1 1 1 1 0 3 9 9
 1 2 9 3 3 9 4 9 3 1 0 0 0 2 9 9
 0 1 2 2 3 3 9 9 9 1 1 2 3 3 3 2
 1 1 1 9 3 9 3 3 2 1 1 9 9 9 2 1
 9 1 2 3 9 2 1 0 1 1 2 2 3 2 3 9
 1 1 1 9 2 1 0 0 1 9 1 0 0 0 2 9
```

3.4 Number of submission

For this exercise, the number of submission is **20**.

3.5 Important notes

- The skeleton program is provided here: minesweeper.c. It contains the scan_mines() function to read the input.
- Constants are useful for this problem. In the above skeleton program, we have added some constants for you. We have also added a few lines of lines we thought could be useful. You may choose not to use it.
- A more efficient way than processing the minefield character by character is to process it row by row, where each row is a string. However, before you do do this you are strongly advised to complete your Week 8 Discussion questions on this issue first.
- This is a problem where incremental coding will help you to prevent a lot of headache. Type in your program incrementally, adding in one function at a time and thoroughly testing it before you type in the next function.

3.6 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): 40 minutes
- Translating pseudo-code into code: 20 minutes
- Typing in the code: 20 minutes
- Testing and debugging: 40 minutes
- **Total: 2 hours**

4 Deadline

The deadline for submitting all programs is **21 July 2012, Saturday, 23:59hr**. Late submission will NOT be accepted.

Go back to CS1010 Labs page.

Aaron Tan
Thursday, October 6, 2011 10:19:00 PM SGT