More Repetition

Lab #3 (CS1010 AY2013/4 Semester 1)

Date of release: 18 September 2013, Wednesday, 9am.

Submission deadline: 5 October 2013, Saturday, 9am.

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, mainly on functions with address parameters, repetition control structure, and pseudo-random numbers. To receive the attempt mark for this lab assignment, you must submit all 3 programs and get a passing feedback mark for each of the program.

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!

Please be reminded that lab assignments must be done in your own effort.

Important notes applicable to all exercises here:

- As you have already gone through two rounds of lab assignment and know our expectation by now, we are going to be stricter in our marking schemes. That means mistakes will be more heavily penalised from now on. This is to give you a more accurate feedback and to better prepare you for PE2.
- Note that you are **NOT allowed to use array, string functions or recursion** for the exercises here. Using it 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, please put function prototypes before the main() function, and the function definitions after the main() function.

1 Exercise 1: Hourglass

1.1 Learning objectives

- Using selection and repetition statements.
- Writing function with address parameters.
- Problem solving.

1.2 Task statement

Write a program **`hourglass.c`** to read in 3 positive integers: **`a`** and **`b`** which are the durations of two hourglasses, and **`c`** which is the duration you want to measure. All values are in minutes. The program then determines if you can measure **`c`** exactly using the hourglasses, and if so, the **number of times** you need to flip the two hourglasses such that the total number of flips is the **minimum**.

You may assume that **`a`** < **`b`** < **`c`**, and that you can only use one hourglass at a time, as the desk is too small to accommodate two hourglasses at the same time.

For example, if you have 4-minute and 7-minute hourglasses, and you want to measure 28 minutes, you need only flip the 7-minute hourglass 4 times. If you want to measure 29 minutes, the solution is to flip the 4-minute hourglass twice and the 7-minute hourglass thrice, giving a total of 5 flips. If you want to measure 9 minutes, then it is impossible to solve.

As another example, if you have 6-minute and 9-minute hourglasses, and you want to measure 42 minutes. There are two ways: flip the 6-minute hourglass once and the 9-minute hourglass four times, hence a total of 5 flips; or flip the 6-minute hourglass four times and the 9-minute hourglass twice, hence a total 6 flips. The first solution is better as it gives the minimum total number of flips.

Your program should include a function **`int compute_flips(int, int, int, int *, int *)`**. It returns 1 (true) if it is possible to measure the given duration **`c`**, or 0 (false) if impossible. The first 3 parameters are **`a`**, **`b`** and **`c`**. The function passes back the number of flips for the two hourglasses through the last two parameters, if it is possible to measure. (If it is not possible to measure, then the values in these last two parameters are immaterial.)

(The above description uses symbols such as **`a`**, **`b`**, and **`c`**. In your program, you should use more descriptive variable names.)

1.3 Sample run

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.

Sample run #1:

```
$ gcc -Wall hourglass.c -o hourglass
$ hourglass
Enter 3 inputs: 4 7 29
Possible!
2 flip(s) for 4-minute hourglass and 3 flip(s) for 7-minute hourglass.
```

Sample run #2:

```
Enter 3 inputs: 4 7 9
Impossible!
```

Sample run #3:

```
Enter 3 inputs: 6 9 42
Possible!
1 flip(s) for 6-minute hourglass and 4 flip(s) for 9-minute hourglass.
```

1.4 Skeleton program and Test data

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

1.5 Important notes

- The mandatory function **`compute_flips()`** as described in section 1.2 above. You must not change the function prototype given in the skeleton program, or marks will be deducted.
- You may write other appropriate function(s) if necessary.
- Hint: You may find some similarity between this exercise and the Coin Change problem you have seen in class.

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.

This exercise might take you a lot of time testing and debugging.

- Devising and writing the algorithm (pseudo-code): 30 minutes
- Translating pseudo-code into code: 15 minutes
- Typing in the code: 15 minutes
- Testing and debugging: 1 hour
- **Total: 2 hours**

2 Exercise 2: Bisection Method

2.1 Learning objectives

- Using selection and repetition statements.
- Using constant.
- Writing function.

2.2 Task

Numerical analysis is an important area in computing. One simple numerical method we shall study here is the **Bisection method**, which computes the root of a continuous function. The **root**, *r*, of a function *f* is a value such that *f(r)* = 0.

How does bisection method work? It is best explained with an example. Given this polynomial function $p(x) = x^3 + 2x^2 + 5$, we need to first provide two endpoints *a* and *b* such that the signs of *p*(*a*) and *p*(*b*) are **different**. For example, let *a*=-3 (hence *p*(*a*)=-4) and *b*=0 (hence *p*(*b*)=5).

The principle is that, the root of the polynomial (that is, the value *r* where *p*(*r*) = 0) must lie somewhere between *a* and *b*. So for the above polynomial, the root *r* must lie somewhere between -3 and 0, because *p*(-3) and *p*(0) have opposite signs. (NOT because -3 and 0 have opposite signs!)

This is achieved as follows. The bisection method finds the midpoint *m* of the two endpoints *a* and *b*, and depending on the sign of *p*(*m*) (the function value at *m*), it replaces either *a* or *b* with *m* (so *m* now becomes one of the two endpoints). It repeats this process and stops when one of the following two events happens:

1. when the midpoint *m* is the root, or
2. when the difference between the two endpoints *a* and *b* falls within a threshold, that is, when they become very close to each other. We shall set the threshold to **0.0001** for this exercise. Then the midpoint *m* is calculated as (*a*+*b*)/2, and this is the approximated root.

Figure 2 below shows the two endpoints *a* (-3) and *b* (0), their midpoint *m* (-1.5), and the function values at these 3 points: *p(a)* = -4, *p(b)* = 5, *p(m)* = 6.125.

Since *p(m)* has the same sign as *p(b)* (both values are positive), this means that *m* will replace *b* in the next iteration.

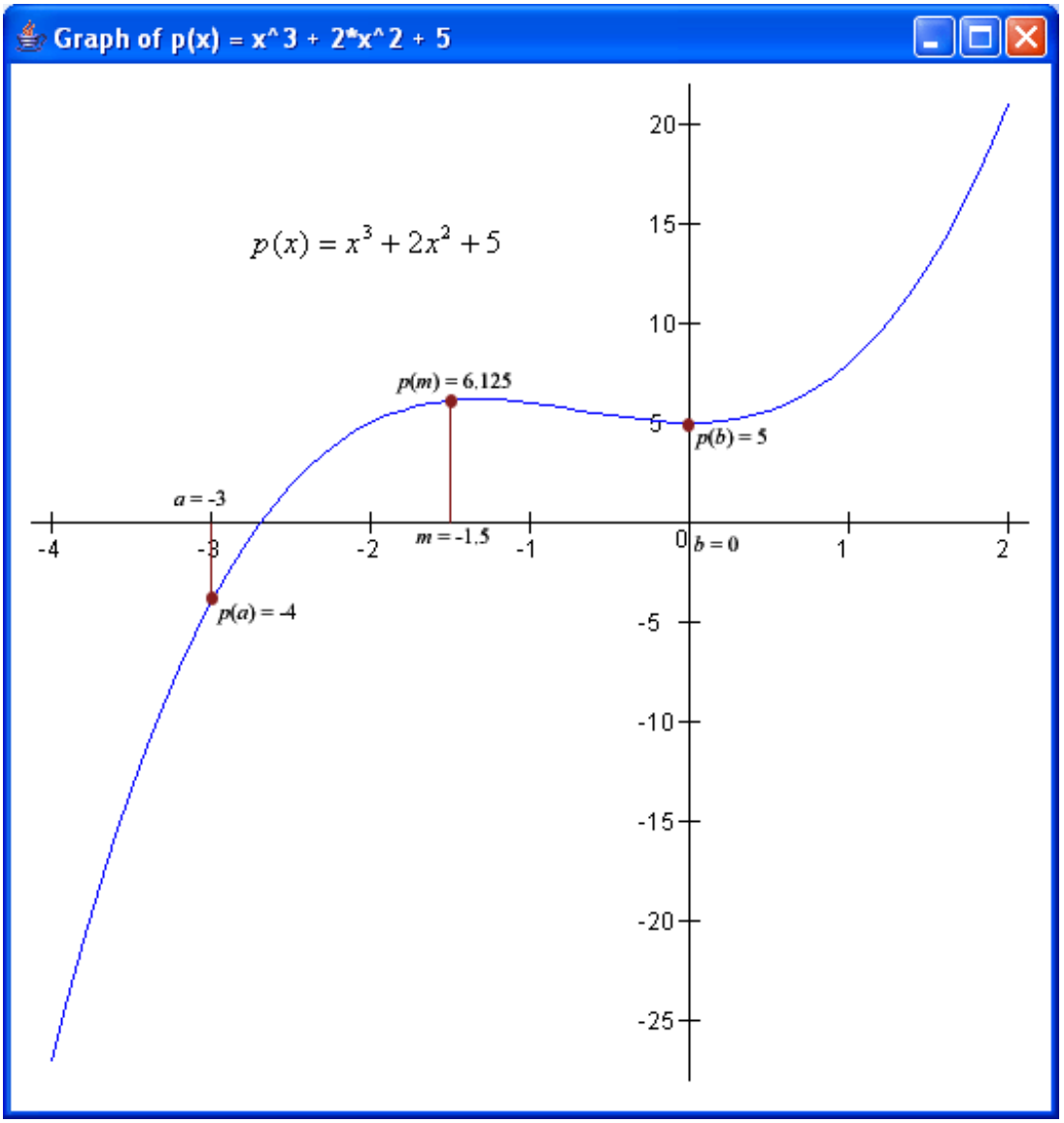

Figure 2. Graph of $p(x) = x^3 + 2x^2 + 5$

The following table illustrates the iterations in the process. The end-point that is replaced by the mid-point value computed in the previous iteration is highlighted in pink background.

iteration	endpoint *a*	endpoint *b*	midpoint *m*	function value *p*(*a*)	function value *p*(*b*)	function value *p*(*m*)
1	-3.000000	0.000000	-1.500000	-4.000000	5.000000	6.125000
2	-3.000000	-1.500000	-2.250000	-4.000000	6.125000	3.734375
3	-3.000000	-2.250000	-2.625000	-4.000000	3.734375	0.693359
4	-3.000000	-2.625000	-2.812500	-4.000000	0.693359	-1.427002
5	-2.812500	-2.625000	-2.718750	-1.427002	0.693359	-0.312714
6	-2.718750	-2.625000	-2.671875	-0.312714	0.693359	0.203541
7	-2.718750	-2.671875	-2.695312	-0.312714	0.203541	-0.051243
8	-2.695312	-2.671875	-2.683594	-0.051243	0.203541	0.076980
9	-2.695312	-2.683594	-2.689453	-0.051243	0.076980	0.013077
10	-2.695312	-2.689453	-2.692383	-0.051243	0.013077	-0.019031
11	-2.692383	-2.689453	-2.690918	-0.019031	0.013077	-0.002964

12	-2.690918	-2.689453	-2.690186	-0.002964	0.013077	0.005059
13	-2.690918	-2.690186	-2.690552	-0.002964	0.005059	0.001048
14	-2.690918	-2.690552	-2.690735	-0.002964	0.001048	-0.000958
15	-2.690735	-2.690552	-2.690643	-0.000958	0.001048	0.000045
16	-2.690735	-2.690643	-2.690689	-0.000958	0.000045	-0.000456

Hence the root of the above polynomial is **-2.690689** (because the difference between *a* and *b* in the last iteration is smaller than the threshold 0.0001), and the function value at that point is **-0.000456**, close enough to zero.

(Some animations on the bisection method can be found on this website: Bisection method. Look at the first one.)

Write a program **`bisection.c`** that asks the user to enter the integer coefficients (c_3, c_2, c_1, c_0) for a polynomial of degree 3: $c_3*x^3 + c_2*x^2 + c_1*x + c_0$. It then asks for the two endpoints, which are real numbers. You may assume that the user enters a continuous function that has a real root. You may use the **`double`** data types for real numbers. To simplify matters, you may also assume that the two endpoints the user entered have function values that are opposite in signs.

Your program should have a function **`double polynomial(double, int, int, int, int)`** to compute the polynomial function value. This is mandatory.

In the output, real numbers are to be displayed accurate to 6 decimal digits (see output in sample runs below).

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.

Only the last 2 lines shown in bold purple are what your program needs to produce. The iterations are shown here for your own checking only.

The sample run below shows the output for the above example. Note that the iterations end when the difference of the two endpoints is less than 0.0001, and the result (root) is the midpoint of these two endpoints.

```
$ gcc -Wall bisection.c -o bisection
$ bisection
Enter coefficients (c3,c2,c1,c0) of polynomial: 1 2 0 5
Enter endpoints a and b: -3 0
#1: a = -3.000000; b = 0.000000; m = -1.500000
     p(a) = -4.000000; p(b) = 5.000000; p(m) = 6.125000
#2: a = -3.000000; b = -1.500000; m = -2.250000
     p(a) = -4.000000; p(b) = 6.125000; p(m) = 3.734375
#3: a = -3.000000; b = -2.250000; m = -2.625000
     p(a) = -4.000000; p(b) = 3.734375; p(m) = 0.693359
#4: a = -3.000000; b = -2.625000; m = -2.812500
     p(a) = -4.000000; p(b) = 0.693359; p(m) = -1.427002
#5: a = -2.812500; b = -2.625000; m = -2.718750
     p(a) = -1.427002; p(b) = 0.693359; p(m) = -0.312714
(... omitted for brevity ...)
#15: a = -2.690735; b = -2.690552; m = -2.690643
     p(a) = -0.000958; p(b) = 0.001048; p(m) = 0.000045
#16: a = -2.690735; b = -2.690643; m = -2.690689
     p(a) = -0.000958; p(b) = 0.000045; p(m) = -0.000456
root = -2.690689
p(root) = -0.000456
```

The second sample run below shows how to find the square root of 5. For polynomial where there are more than one real root, only one root needs to be reported.

```
$ bisection
Enter coefficients (c3,c2,c1,c0) of polynomial: 0 1 0 -5
Enter endpoints a and b: 1.0 3.0
#1: a = 1.000000; b = 3.000000; m = 2.000000
     p(a) = -4.000000; p(b) = 4.000000; p(m) = -1.000000
#2: a = 2.000000; b = 3.000000; m = 2.500000
     p(a) = -1.000000; p(b) = 4.000000; p(m) = 1.250000
(... omitted for brevity ...)
#16: a = 2.236023; b = 2.236084; m = 2.236053
     p(a) = -0.000201; p(b) = 0.000072; p(m) = -0.000065
root = 2.236053
p(root) = -0.000065
```

The third sample run below shows how to find the root of the function $2x^2 - 3x$. Since the midpoint of the given endpoints *a* and *b* is 1.5 which is the root of the function, the loop ends after the first iteration.

```
$ bisection
Enter coefficients (c3,c2,c1,c0) of polynomial: 0 2 -3 0
Enter endpoints a and b: 0.5 2.5
#1: a = 0.500000; b = 2.500000; m = 1.500000
     p(a) = -1.000000; p(b) = 5.000000; p(m) = 0.000000
root = 1.500000
p(root) = 0.000000
```

2.4 Skeleton program and Test data

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

2.5 Important notes

- You are reminded that only two lines of output (those shown in **bold purple** in the sample runs above) are the required output. Your program is **not** to display the iterations in your output.
- The loop should terminate when the midpoint is the root, or when the two endpoints *a* and *b* are very close to each other, **not** when the two function values *p(a)* and *p(b)* are very close to each other. This is a common mistake.
- You should define a constant for the threshold (0.0001) in your program.
- The mandatory function is **`polynomial()`** as described in section 2.2 above. You must not change the function prototype given in the skeleton program, or marks will be deducted.
- Hint: You may use the **`fabs()`** function in **`math.h`**.

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 might take you a lot of time testing and debugging.

- Devising and writing the algorithm (pseudo-code): 30 minutes
- Translating pseudo-code into code: 15 minutes

- Typing in the code: 15 minutes
- Testing and debugging: 1 hour
- **Total: 2 hours**

3 Exercise 3: Monte Carlo

3.1 Learning objectives

- Using the selection and repetition statements.
- Generating pseudo-random numbers.
- Writing function.

3.2 Task statement

(**Note:** You should have read through Week 6 Discussion Sheet "Section III Exploration -- Question 6" on the functions **`rand()`** and **`srand()`** before you attempt this exercise.)

π, or `pi`, is a well-known constant 3.14159265..., which is the ratio of a circle's circumference to its diameter.

Consider the *unit circle* which, in cartesian coordinates, is defined by the equation **x*x + y*y = 1**. Its area is `pi`, and hence the area of the part of the unit circle that lies in the first quadrant is `pi`/4.

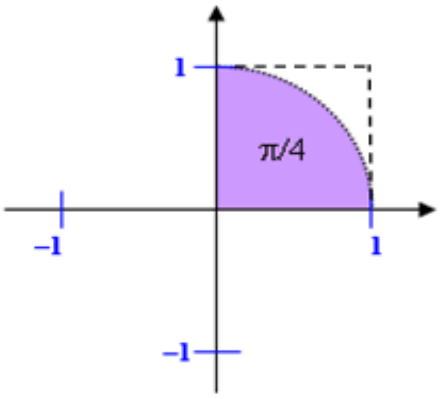

Figure 3.

Imagine that you throw darts at random at the unit square (in the first quadrant) so that the darts land randomly on (x, y) coordinates satisfying 0 ≤ x ≤ 1 and 0 ≤ y ≤ 1. Some of these darts will land inside the unit circle's quadrant. As shown in Figure 3 above, the unit square is the square box which contains the unit circle's quadrant (the purple-shaded region).

Since the unit circle's quadrant has area `pi`/4 and the unit square has area 1, if you program it such that the darts always land within the unit square, then the proportion of darts you randonly throw that land in the unit circle's quadrant will be approximately `pi`/4.

(How do you determine if a dart lands inside the unit circle's quadrant?)

Write a program **`montecarlo.c`** that asks the user to enter the number of darts to throw. Pass this value to a function **`throwDarts(int)`**. This is a mandatory function.

In the function, generate the appropriate random numbers to simulate the throwing of darts. Calculate the number of darts that land inside the unit circle's quadrant. (Consider the dart to have landed inside if $\mathbf{x^2 + y^2 \leq 1}$, i.e. including the unlikely event that it lands exactly on the boundary of the unit circle's quadrant.) Return this number back to the caller, which then computes the approximate value of `pi`.

It is reasonable to expect that the more darts we throw, the more accurately the computed value of `pi` approximates the real value of π.

Your program should output the number of darts that landed inside the unit circle's quadrant, and the value of `pi` computed correct to 4 decimal places.

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 montecarlo.c -o montecarlo
$ montecarlo
How many darts? 5000
Darts landed inside unit circle's quadrant = 3934
Approximated pi = 3.1472
```

Second sample run:

```
$ montecarlo
How many darts? 90000000
Darts landed inside unit circle's quadrant = 70686491
Approximated pi = 3.1416
```

3.4 Skeleton program and Test data

- The skeleton program is provided here: montecarlo.c
- No test data since your program generates random values.

3.5 Important notes

- Write a function **`int throwDarts(int)`** to take in the number of darts, and compute and return the number of darts that landed inside the unit circle. The caller than uses this result to compute the value of pi. This is a mandatory function. You must not change the function prototype given in the skeleton program, or marks will be deducted.
- Ensure that the random values generated are within the desired range.
- The **`rand()`** function returns an integer in the range [0, RAND_MAX]. The constant RAND_MAX is defined in <stdlib.h>. You may use `printf("%d\n", RAND_MAX);` to see its value. You may need to use RAND_MAX in your program to ensure that the generated random values are real numbers in the range [0.0, 1.0].
- You should test your program with several values of the numbers of darts to be thrown, some of which should be very large (but within the range of **`int`**).
- Since your program generates random values, there will not be any test data to test your program. Hence, CodeCrunch will not be able to perform tests on your program. It will give you a grade of "E", which you may ignore. Your grader will go through your program manually to check.

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): 20 minutes
- Translating pseudo-code into code: 5 minutes
- Typing in the code: 10 minutes
- Testing and debugging: 15 minutes
- **Total: 50 minutes**

4 Deadline

The deadline for submitting all programs is **5 October 2013, Saturday, 9am**. Late submission will NOT be accepted.

- 0 Introduction
- 1 Exercise 1: Hourglass
    - 1.1 Learning objectives
    - 1.2 Task statement
    - 1.3 Sample run
    - 1.4 Skeleton program and Test data
    - 1.5 Important notes
    - 1.6 Estimated development time
- 2 Exercise 2: Bisection Method
    - 2.1 Learning objectives
    - 2.2 Task statement
    - 2.3 Sample runs
    - 2.4 Skeleton program and Test data
    - 2.5 Important notes
    - 2.6 Estimated development time
- 3 Exercise 3: Monte Carlo
    - 3.1 Learning objectives
    - 3.2 Task statement
    - 3.3 Sample runs
    - 3.4 Skeleton program and Test data
    - 3.5 Important notes
    - 3.6 Estimated development time
- 4 Deadline

Aaron Tan

Sunday, September 15, 2013 02:59:20 PM SGT