CS1010E: Programming Methodology

Assessed Lab 4A: Array [11%]

29 Mar 2017

Instructions

Please read all the instructions very carefully!

1. This is an **Open Book** assessment:
   - You are allowed to bring any printed materials and calculator
   - You are NOT allowed to use other electronic devices besides the lab's computer
   - You are NOT allowed to talk with your friends, to talk with invigilators please raise your hand
   - You are NOT allowed to access the internet except to the `plab` server via `SSH terminal`
2. This lab assessment consists of **one (1)** problems with several tasks:
   - The tasks are intended to guide you in solving the problem
   - Each task should have **its own separate file** where the task number is written at the back: `task3.c` is used for task 3
   - To proceed to the next level (*e.g., from task 2 to task 3*), copy your program using the command `cp task2.c task3.c`
   - Fill in your **Name**, **Matric** (*starts with* `A`), and **NUSNET ID** (*starts with either* `A` *or* `E`)
3. Numerical and precision guides:
   - **Two (2)** types of *input* numbers: `real` (*may have decimal point*) and `integer` (*no decimal point*)
   - `integer` may contain leading *zeroes*: always use `scanf("%d")` to ensure *decimal* representation
   - `integer` has a range of -2^{31} to $+2^{31}-1$, `unsigned integer` has a range of 0 to $+2^{32}-1$
   - Always use `double` for `real` number input for high precision, but numbers that differs by less than `0.001` are considered *equal*
4. Starting the tests:
   - Use the program `SSH Secure Shell Client`
   - Login to `plab` server using the given `username` and `password`
5. Testing and debugging guides:
   - You may open **two (2)** or more `SSH Terminal`: 1 for *coding* and 1 for *compilation + testing*
   - Assumption stated in the task is considered to always hold and no checking is necessary
   - Assumption NOT stated in the task will be tested in hidden input: *always think of worst case*
   - Test case outputs are organized by task number and test case number:
     - Task number `T` on test case number `C` have output file `testT_C.out`
     - *For example*: task number 2 with test case number 3 have output file `test2_3.out`
   - Test case inputs are the same for all tasks: *e.g.,* `test2.in`
6. Marking:
   - Grading is done *automatically* using CodeCrunch: only the largest correct task is considered
   - For instance: Task 1 is *empty (i.e., not done at all)*, Task 2 is *correct*, Task 3 is *incorrect*
     ↦ mark for Task 2 is taken
   - The mark for each task is given on the right side, it is a *cumulative* mark
7. Time management suggestion: [Total Time: **1 hour 30 minutes**]:
   - *Coding*: approx. **1 hour** (±**30 minutes** for debugging)
   - *Ending*: approx. last **5 minutes** ensures that you save the filename correctly

Coin Change [100 %]

Problem Description

"The change-making problem addresses the following question: how can a given amount of money be made with the least number of coins of given denominations? It is a knapsack type problem, and has applications wider than just currency." – Wikipedia

The formulation of this problem is as follows. Given a coin *denomination* in an array as $[c_1, c_2, c_3, ..., c_n]$ and a change to be made as C, count the number of ways you can mix and match coins in the *denomination* to be equal to C such that the ordering of the coin does not matter under the assumption that you have an infinite number of coins of each *denomination*. For instance, given the denomination $[1, 3, 5]$ and a change 8, there are **five (5)** ways to make the change. The combinations are listed below:

```
1+1+1+1+1+1+1+1 = 8
1+1+1+1+1+3 = 8
1+1+3+3 = 8
1+1+1+5 = 8
3+5 = 8
```

The simple way to think about the problem is as follows:

- Consider the current change C and the *denomination* $[c_1, c_2, ..., c_n]$
  - If you choose to pay with c_1, how many ways can you make change to $C - c_1$
  - If you choose NOT to pay with c_1, then you have to pay C with $[c_2, ..., c_n]$
- The number of ways is the combination of both methods

Now, although *efficiency* is **not tested** in this module, the problem can quickly become unbearably slow to solve if you decide to *copy* the *denomination* array in a recursion. As such, your solution should involve ways to not copy the *denomination* array via the use of `index` to the *denomination* array indicating the start point of the *denomination*. Lastly, while the problem can be solved **without recursion**, the coding is extremely tedious and you should try to solve it using recursion. That being the case, you should find out what are the base cases for the recursion.

Concepts Tested:

1. Input/Output: `scanf` and `printf`
2. Modulo & Boolean Arithmetic: `%`, `||`, `&&`, `==`, etc
3. Selection Statement: `if` and/or `if-else`
4. Repetition Statement: `while` and/or `for` as well as *nested repetition*
5. Function: including *simple recursion*
6. Arrays: including *2D arrays*

Final Objective

Given a *denomination* array and a change C to be made, calculate the number of ways the *combination* of coins from the denomination can be used to make a valid change C.

Assumptions

The following assumptions are considered to be true, they limit the inputs to the following restrictions:

- ▷ $3 \leq$ `N` ≤ 8 (*The number of coins in the denomination*)
- ▷ $1 \leq$ `coin` ≤ 100 (*The value of coins in the denomination*)
- ▷ $1 \leq$ `C` ≤ 100 (*The change to be made*)
- ▷ The `coin` *denomination* are *unique*

Tasks

The problem is split into 4 tasks with 2 number of testcases given. In the sample run, please note the following:

- ↩ is the *invisible* `[newline]` character.
- User input in blue and program output in purple color.
- Comments are in green color and are not part of the input and/or output.
- If the test(s) give(s) **NO** message(s), it means your program is correct.

Task 1/4: ***[Input/Output]*** **[10%]**

Write a program to read the coin *denomination* and the change C, and print the list of denomination. The input is given as **two (2)** lines. The first line consists of **two (2)** `integer` numbers N and C corresponding to the number of coins in the *denomination* and the value of the change to be made. The second line consists of N `integer` numbers $c_1, c_2, ..., c_N$ corresponding to the value of coins in the *denomination*. Note that there is **NO** additional `[space]` at the end.

Sample Run:

Inputs:	Outputs:
3 8 3 5 1	3 5 1↩

Save your program in the file named `coin1.c`. No submission is necessary.

Test your program using the following command(s):

```
./a.out < test1.in | diff - test1_1.out
./a.out < test2.in | diff - test1_2.out
```

To proceed to the next task (*e.g., task 2*), copy your program using the following command:

```
cp coin1.c coin2.c
```

Task 2/4: ***[Sorting]*** **[30%]**

Write a program to read the coin *denomination* and the change C, and print the list of denomination in a *sorted* order from smallest to largest. The input is given as **two (2)** lines. The first line consists of **two (2)** `integer` numbers N and C corresponding to the number of coins in the *denomination* and the value of the change to be made. The second line consists of N `integer` numbers $c_1, c_2, ..., c_N$ corresponding to the value of coins in the *denomination*. Note that there is **NO** additional `[space]` at the end.

Sample Run:

Inputs:	Outputs:
3 8 3 5 1	1 3 5↩

Save your program in the file named `coin2.c`. No submission is necessary.

Test your program using the following command(s):

```
./a.out < test1.in | diff - test2_1.out
./a.out < test2.in | diff - test2_2.out
```

To proceed to the next task (*e.g., task 3*), copy your program using the following command:

```
cp coin2.c coin3.c
```

Task 3/4: ***[Greed]*** **[60%]**

Write a program to read the coin *denomination* and the change C, and check if the change C can be made using a *greedy* method. The *greedy* method try to make changes from the largest possible denomination whenever possible. The input is given as **two (2)** lines. The first line consists of **two (2)** `integer` numbers N and C corresponding to the number of coins in the *denomination* and the value of the change to be made. The second line consists of N `integer` numbers $c_1, c_2, ..., c_N$ corresponding to the value of coins in the *denomination*. Note that there is **NO** additional [`space`] at the end.

Sample Run:

Inputs:

```
3 8
3 5 1
```

Outputs:

```
Yes↩ | Change: 5 + 3
```

Sample Run:

Inputs:

```
3 9
3 5 2
```

Outputs:

```
No↩ | Greedily take 5 and 3: 9-8 = 1 > 2
| Note that non-greedy method can make the change
```

Save your program in the file named `coin3.c`. No submission is necessary.

Test your program using the following command(s):

```
./a.out < test1.in | diff - test3_1.out
```

```
./a.out < test2.in | diff - test3_2.out
```

To proceed to the next task (*e.g., task 4*), copy your program using the following command:

```
cp coin3.c coin4.c
```

Task 4/4: ***[Coin Change]*** **[100%]**

Write a program to read the coin *denomination* and the change C, and check if the change C can be made using a *greedy* method. The *greedy* method try to make changes from the largest possible denomination whenever possible. The input is given as **two (2)** lines. The first line consists of **two (2)** `integer` numbers N and C corresponding to the number of coins in the *denomination* and the value of the change to be made. The second line consists of N `integer` numbers $c_1, c_2, ..., c_N$ corresponding to the value of coins in the *denomination*. Note that there is **NO** additional [`space`] at the end.

Sample Run:

Inputs:

```
3 8
3 5 1
```

Outputs:

```
5↩
```

Sample Run:

Inputs:

```
3 9
3 5 2
```

Outputs:

```
3↩ | 5+2+2 & 3+2+2+2 & 3+3+3
```

Save your program in the file named `coin4.c`. No submission is necessary.

Test your program using the following command:

```
./a.out < test1.in | diff - test4_1.out
```

```
./a.out < test2.in | diff - test4_2.out
```

Useful VIM and SSH Terminal Commands

- **VIM Mode Switch:**
  - **i** i nsert (*from* Command)
  - **esc** esc ape to Command
- **Basic VIM Commands:** [mode=Command]
  - **:w** w rite file
  - **:q** q uit file
  - **:q!** q uit file (*forced*: *without saving*)
  - **:wq** w rite and q uit
- **Advanced VIM Commands:** [mode=Command]
  - **/text** find text
  - **n** find n ext text
  - **shift + n** find previous text
  - **gg=G** auto-indentation all lines
- **VIM Text Edit Commands:** [mode=Command]
  - **dd** d elete line at cursor (*cut*)
  - **yy** y ank line at cursor (*copy*)
  - **p** p aste after current cursor
  - **u** u ndo one change
  - **x** cut one character at cursor
  - **:red** red o undone changes
  - *N* **dd** d elete N lines down (N is number)
  - *N* **yy** y ank N lines down (N is number)
- **VIM Auto-Completion:** [mode=Insert]
  - **ctrl + n** complete word
  - **ctrl + x** complete line
- **Basic SSH Terminal Commands:**
  - **cd dir** open folder dir
  - **cd ..** open *parent* folder
  - **rm file** remove file file
  - **rm -r dir** remove folder dir
  - **vim file** open file in VIM
  - **ls** list files in folder
  - **ls -all** list ALL files in folder
  - **cat file** open *small* text file
  - **less -e file** open *large* text file
  - **cp f1 f2** copy f1 to f2
  - **mv f1 f2** move f1 to f2

    (*in effect, rename if in same folder*)
- **Execute Your Program in SSH Terminal :**
  - **gcc -Wall file** compile file
  - **gcc -Wall -lm file**

    compile file with math library (i.e. #define <math.h>) included
  - **./a.out** run program
  - **gcc -Wall file -o f1**

    compile file and rename executable into f1 (run using ./f1)
- **Advanced Program Execution Commands in SSH Terminal :**
  - **./a.out < f_in**

    run program with input redirection from file located at f_in

    (e.g. **./a.out < test1.in**)
  - **./a.out < f_in > f_out**

    run program with input redirection from file located at f_in and redirect the output to write into (*non-existing*) file called f_out

    (e.g. **./a.out < test1.in > output1**)
  - **diff f1 f2**

    compares the two files (f1 compared with f2) line by line (*note: no news is good news*)

    (e.g. **diff output1 test1_1.out**)
  - **./a.out < f_in | diff - f_out**

    run program with input from f_in immediately compare output with f_out

    (e.g. **./a.out < test1.in | diff - test3_1.out**)
- **SSH Terminal Emergency Commands:**
  - *Infinite loop* press **ctrl + c**
  - *End input* press **ctrl + d**

  (*better way is to use input redirection*)
- **VIM DO NOT DO LIST**
  - **ctrl + z** move to background

    (if done, type **fg** into **SSH Terminal**)
  - **ctrl + s** suspend

    (if done, press ctrl+q)
  - *Close without using* **:q**
    - on reopen, .swp file created
    - open file, choose Recover & exit VIM
    - open file again & choose Delete
- **GCC DO NOT DO LIST**
  - **gcc file -o file**

    compile file and rename into file (now, file is no longer a C program file)
    - ***pray hard...***
    - look for .file.history by typing **ls -all**
    - copy to windows using **SSH File Transfer**
    - ***hope*** latest code is at *end of file*