Problem 1-5: Greedy

This is CS50. Harvard University. Fall 2014.

Table of Contents

Questions? Feel free to head to CS50 on Reddit[1], CS50 on StackExchange[2], or the CS50 Facebook group[3].

Objectives

- Use algorithms to solve problems.
- Experiment with different designs in solving a problem.

Recommended Reading

- Pages 1 – 7, 9, and 10 of http://www.howstuffworks.com/c.htm.
- Chapters 1 – 5, 9, and 11 – 17 of *Absolute Beginner's Guide to C*.
- Chapters 1 – 6 of *Programming in C*.

Academic Honesty

This course's philosophy on academic honesty is best stated as "be reasonable." The course recognizes that interactions with classmates and others can facilitate mastery of

[1] https://www.reddit.com/r/cs50
[2] http://cs50.stackexchange.com
[3] https://www.facebook.com/groups/cs50

the course’s material. However, there remains a line between enlisting the help of another and submitting the work of another. This policy characterizes both sides of that line.

The essence of all work that you submit to this course must be your own. Collaboration on problems is not permitted (unless explicitly stated otherwise) except to the extent that you may ask classmates and others for help so long as that help does not reduce to another doing your work for you. Generally speaking, when asking for help, you may show your code or writing to others, but you may not view theirs, so long as you and they respect this policy’s other constraints. Collaboration on quizzes and tests is not permitted at all. Collaboration on the final project is permitted to the extent prescribed by its specification.

Below are rules of thumb that (inexhaustively) characterize acts that the course considers reasonable and not reasonable. If in doubt as to whether some act is reasonable, do not commit it until you solicit and receive approval in writing from your instructor. If a violation of this policy is suspected and confirmed, your instructor reserves the right to impose local sanctions on top of any disciplinary outcome that may include an unsatisfactory or failing grade for work submitted or for the course itself.

Reasonable

- Communicating with classmates about problems in English (or some other spoken language).
- Discussing the course’s material with others in order to understand it better.
- Helping a classmate identify a bug in his or her code, such as by viewing, compiling, or running his or her code, even on your own computer.
- Incorporating snippets of code that you find online or elsewhere into your own code, provided that those snippets are not themselves solutions to assigned problems and that you cite the snippets' origins.
- Reviewing past years' quizzes, tests, and solutions thereto.
- Sending or showing code that you’ve written to someone, possibly a classmate, so that he or she might help you identify and fix a bug.
- Sharing snippets of your own solutions to problems online so that others might help you identify and fix a bug or other issue.
- Turning to the web or elsewhere for instruction beyond the course’s own, for references, and for solutions to technical difficulties, but not for outright solutions to problems or your own final project.

- Whiteboarding solutions to problems with others using diagrams or pseudocode but not actual code.
- Working with (and even paying) a tutor to help you with the course, provided the tutor does not do your work for you.

Not Reasonable

- Accessing a solution to some problem prior to (re-)submitting your own.
- Asking a classmate to see his or her solution to a problem before (re-)submitting your own.
- Decompiling, deobfuscating, or disassembling the staff’s solutions to problems.
- Failing to cite (as with comments) the origins of code, writing, or techniques that you discover outside of the course’s own lessons and integrate into your own work, even while respecting this policy’s other constraints.
- Giving or showing to a classmate a solution to a problem when it is he or she, and not you, who is struggling to solve it.
- Looking at another individual’s work during a quiz or test.
- Paying or offering to pay an individual for work that you may submit as (part of) your own.
- Providing or making available solutions to problems to individuals who might take this course in the future.
- Searching for, soliciting, or viewing a quiz’s questions or answers prior to taking the quiz.
- Searching for or soliciting outright solutions to problems online or elsewhere.
- Splitting a problem’s workload with another individual and combining your work (unless explicitly authorized by the problem itself).
- Submitting (after possibly modifying) the work of another individual beyond allowed snippets.
- Submitting the same or similar work to this course that you have submitted or will submit to another.

- Using resources during a quiz beyond those explicitly allowed in the quiz's instructions.
- Viewing another's solution to a problem and basing your own solution on it.

Assessment

Your work on this problem set will be evaluated along four axes primarily.

Scope
To what extent does your code implement the features required by our specification?

Correctness
To what extent is your code consistent with our specifications and free of bugs?

Design
To what extent is your code written well (i.e., clearly, efficiently, elegantly, and/or logically)?

Style
To what extent is your code readable (i.e., commented and indented with variables aptly named)?

To obtain a passing grade in this course, all students must ordinarily submit all assigned problems unless granted an exception in writing by the instructor.

Time for Change

"Counting out change is a blast (even though it boosts mathematical skills) with this spring-loaded changer that you wear on your belt to dispense quarters, dimes, nickels, and pennies into your hand." Or so says the website[4] on which we found this here accessory (for ages 5 and up).

[4] http://hearthsong.com/

Of course, the novelty of this thing quickly wears off, especially when someone pays for a newspaper with a big bill. Fortunately, computer science has given cashiers everywhere ways to minimize numbers of coins due: greedy algorithms.

According to the National Institute of Standards and Technology (NIST), a greedy algorithm[5] is one "that always takes the best immediate, or local, solution while finding an answer. Greedy algorithms find the overall, or globally, optimal solution for some optimization problems, but may find less-than-optimal solutions for some instances of other problems."

What’s all that mean? Well, suppose that a cashier owes a customer some change and on that cashier’s belt are levers that dispense quarters, dimes, nickels, and pennies. Solving this "problem" requires one or more presses of one or more levers. Think of a "greedy" cashier as one who wants to take, with each press, the biggest bite out of this problem as possible. For instance, if some customer is owed 41¢, the biggest first (i.e., best immediate, or local) bite that can be taken is 25¢. (That bite is "best" inasmuch as it gets us closer to 0¢ faster than any other coin would.) Note that a bite of this size would whittle what was a 41¢ problem down to a 16¢ problem, since 41 - 25 = 16. That is, the remainder is a similar but smaller problem. Needless to say, another 25¢ bite would be too big (assuming the cashier prefers not to lose money), and so our greedy cashier would move on to a bite of size 10¢, leaving him or her with a 6¢ problem. At that point, greed calls for one 5¢ bite followed by one 1¢ bite, at which point the problem is solved. The customer receives one quarter, one dime, one nickel, and one penny: four coins in total.

[5] http://www.nist.gov/dads/HTML/greedyalgo.html

It turns out that this greedy approach (i.e., algorithm) is not only locally optimal but also globally so for the United States dollar (and also the Euro). That is, so long as a cashier has enough of each coin, this largest-to-smallest approach will yield the fewest coins possible.

How few? Well, you tell us. Write, in a file called `greedy.c` in your `~/workspace/unit1` directory, a program that first asks the user how much change is owed and then spits out the minimum number of coins with which said change can be made. Use `GetFloat` from the CS50 Library to get the user's input and `printf` from the Standard I/O library to output your answer. Assume that the only coins available are quarters (25¢), dimes (10¢), nickels (5¢), and pennies (1¢).

We ask that you use `GetFloat` so that you can handle dollars and cents, albeit sans dollar sign. In other words, if some customer is owed $9.75 (as in the case where a newspaper costs 25¢ but the customer pays with a $10 bill), assume that your program's input will be `9.75` and not `$9.75` or `975`. However, if some customer is owed $9 exactly, assume that your program's input will be `9.00` or just `9` but, again, not `$9` or `900`. Of course, by nature of floating-point values, your program will likely work with inputs like `9.0` and `9.000` as well; you need not worry about checking whether the user's input is "formatted" like money should be. And you need not try to check whether a user's input is too large to fit in a `float`. But you should check that the user's input makes cents! Er, sense. Using `GetFloat` alone will ensure that the user's input is indeed a floating-point (or integral) value but not that it is non-negative. If the user fails to provide a non-negative value, your program should re-prompt the user for a valid amount again and again until the user complies.

Incidentally, do beware the inherent imprecision of floating-point values. For instance, `0.01` cannot be represented exactly as a float. Try printing its value to, say, `50` decimal places, with code like the below:

```c
float f = 0.01;
printf("%.50f\n", f);
```

Before doing any math, then, you'll probably want to convert the user's input entirely to cents (i.e., from a `float` to an `int`) to avoid tiny errors that might otherwise add up!

[6] https://cs50.harvard.edu/resources/cppreference.com/stdmath/round.html

Of course, don't just cast the user's input from a `float` to an `int`! After all, how many cents does one dollar equal? And be careful to round[6] and not truncate your pennies!

Not sure where to begin? Not to worry, start with a walkthrough:

https://www.youtube.com/watch?v=9dZzyl7dCuw
Incidentally, so that we can automate some tests of your code, we ask that your program's last line of output be only the minimum number of coins possible: an integer followed by `\n`. Consider the below representative of how your own program should behave, wherein underlined text is some user's input.

```
username@ide50:~/workspace/unit1 $ ./greedy
O hai! How much change is owed?
0.41
4
```

By nature of floating-point values, that user could also have inputted just `.41`. (Were they to input `41`, though, they'd get many more coins!)

Of course, more difficult users might experience something more like the below.

```
username@ide50:~/workspace/unit1 $ ./greedy
O hai! How much change is owed?
-0.41
How much change is owed?
-0.41
How much change is owed?
foo
Retry: 0.41
4
```

Per these requirements (and the sample above), your code will likely have some sort of loop. If, while testing your program, you find yourself looping forever, know that you can kill your program (i.e., short-circuit its execution) by hitting ctrl-c (sometimes a lot).

We leave it to you to determine how to compile and run this particular program!

If you'd like to check the correctness of your program with `check50`, you may execute the below.

```
check50 1516.unit1.greedy greedy.c
```

And if you'd like to play with the staff's own implementation of `greedy`, you may execute the below.

```
~cs50/pset1/greedy
```

Incidentally, you should be aware that there are *many* ways to solve this particular problem. After you solve it one way, if you find yourself with some more time, attempt to obtain the same results using a different approach, perhaps trying to **optimize** your program by coming up with a more efficient solution. It's good programming practice, for starters, but also gets you thinking about what alternative designs![7]

This was Problem 1-5.

[7] Hint: There's a way to solve this problem that doesn't involve the use of loops at all!