Input Processing

PROBLEM: LUHN CHECKSUM VALIDATION

The Luhn formula is a widely used system for validating identification numbers. Using the original number, double the value of every other digit. Then add the values of the individual digits together (if a doubled value now has two digits, add the digits individually). The identification number is valid if the sum is divisible by 10.

Write a program that takes an identification number of arbitrary length and determines whether the number is valid under the Luhn formula. The program must process each character before reading the next one.

Breaking Down the Problem

Program:

- Input = ID # of arbitrary length
- Function = Determine whether # is valid under Luhn Formula
- Constraints
  - process each char before reading the next one. (Not reading and storing whole number at once)

Luhn Formula:

- Given a number

1) Double value of every other digit (Ex: 1 7 6 2 4 8)
2) Add value of all individual digits together
   - If a doubled value has 2 digits. Add digits individually
3) Valid if sum is divisible by 10

Developing the Plan

List of Issues (Subproblems )

- Double every other digit starting from right end of ID #
- Treating doubled #s >= 10 according to individual digits
- Process user input (of arbitrary length) 1 char at a time
    * input will be L→R, we need L←R
- Know when we've reached the end of the number

Prioritizing

- Because we'll solve problems by working on individual pieces before writing a final solution there is no need to work in a particular order
    - You can choose to start w/ the easiest, most challenging or most interesting piece.

Solving The Problems

Treating doubled #s >= 10 according to individual digits

Constraints

- Range of possible values for digits 0-9 ⇒ 10-18
- 1st digit is always 1

Process

- Code experiment to determine formula
- Create function from formula

Process user input (of arbitrary length) 1 digit at a time

*

Constraints:

- input will be L→R, we need L←R
- arbitrary length
- input must be read 1 digit at a time

Questions

- What form should we receive it in to minimize the amount of code needed to work with it.

Q: What form should we receive it in to minimize the amount of code needed to work with it.

- reading an int would just give one long number and there is a limit to how big of an int can be read
- We can read chars 2 & 1 but would have to convert it to an int to use mathematically

A: We will use chars but we need a mechanism to convert char "7" to int=7

Issues

- Read a character from user
- Convert character to eq integer
- Output integer to demonstrate results

Convert character to eq integer

Character	Character Code	Desired Integer	Difference
0	48	0	48
1	49	1	48
2	50	2	48
3	51	3	48
4	52	4	48
5	53	5	48
6	54	6	48
7	55	7	48
8	56	8	48
9	57	9	48

Double every other digit starting from right end of ID #

- Problem Reduction (Make Getting started as easy as possible)
  - Because the input can be of arbitrary length
  - Reducing it to a fixed length can help us more easily confirm a general formula

Reduction 1: Input has fixed length of 6 chars (Including check digit)

Problem/Issues:
- Recieve 6 digit ID # as input
- Determine if # is valid under luhn
- Process input char by char

Reduction 2: None of the digits has to be doubled

Problem/Issues:
- Recieve 6 digit ID # as input
- Determine if the value of the digits added is divisible by 10
- Process input char by char

Luhn Formula:
- Given a number
1) ~~Double value of every other digit (Ex: 1 7 6 2 4 8)~~
2) Add value of all individual digits together
   ~~If a doubled value has 2 digits. Add digits individually~~
3) Valid if sum is divisible by 10

# of Arbitrary length
- Divide and Conquer:
  - Handling the doubling of numbers is different for even and odd #s
  - It helps to divide these and solve each through experimentation initially

* Use listing issues (subproblems) as checkpoints & exit criteria & place finders

Issues:
- Determine when you've reached end of number
- Know the length of the number (# digit chars)

Know when we've reached the end of the number

- Each OS has a different value for its end character

At this point we know how to handle an even and odd number of any length

- What we need to know is the length to decide what logic to use.

Know the length of the number (# digit chars)

Solve By Analogy

- If you haven't seen an analogy already, try making one yourself

- Lets make a problem that is explicitely about this situation - "Staring the problem in the face"

PROBLEM: POSITIVE OR NEGATIVE

Write a program that reads 10 integers from the user. After all the numbers have been entered, the user may ask to display the count of positive numbers or the count of negative numbers.

//I don't think I would come up with this as an analogy

Issues:

- Recieve 10 integers from the user
- After all #s entered if requested display the count of positive or negative numbers

List of Issues (Subproblems) (original)

- ~~Double every other digit starting from right end of ID #~~
- ~~Treating doubled #s >= 10 according to individual digits~~
- ~~Process user input (of arbitrary length) 1 char at a time~~

  ~~* input will be L->R, we need L<-R~~

- ~~Know when we've reached the end of the number~~

Putting it all together

- At this point we've solved our original subproblems and can put everything together.
- The final solution will be composed of previously implemented parts

Closing Notes

- "My plan is not necessarily your plan."
- My list of initial issues will likely differ from yours as well as the steps taken to solve the problems
- "There is no one right solution for a problem as any program that meets all the constraints counts as a solution. And for any solution there is no one right way of reaching it."
- It's always better to take more steps than to try to do too much at once

- Avoid Frustration!!!

  - The more you try to do in each step the more you invite frustration

Also remember the last of my general rules for problem solving: *Avoid frustration.* The more work you try to do in each step, the more you invite potential frustration. Even if you back off a difficult step and break it up into substeps, the damage will have been done because psychologically you'll feel like you're going backward instead of making progress. When I coach beginning programmers in a step-by-step approach, I sometimes have a student complain, "Hey, that step was too easy." To which I reply, "What are you complaining about?" If you've taken a problem that initially looked tough and broken it down into pieces so small that every piece is trivial to accomplish, I say: Congratulations! That's just what you should hope for.