## Thought Puzzle

- Sometimes we need to find the opposites of conditions
- An example is error checking to prompt the user for the correct value
  - We know what kind of value we ***want***
  - But we have to loop on a condition for the values we ***don't want***…
- Other examples, too

## Thought Puzzle

- What is the opposite of:

  if small<middle and middle<large :

  A: if small<middle or middle<large :

  B: if small>=middle and middle>=large :

  C: if small>=middle or middle>=large :

## Thought Puzzle

- What is the opposite of:

  if small<middle and middle<large :

- For that whole statement to be ***true***, both
  - (small < middle) == true
  - **and**
  - (middle < large) == true

## Thought Puzzle

- What is the opposite of:

  if small<middle and middle<large :

- For that whole statement to be ***false***, at least
  - (first statement) == false
  - **or**
  - (second statement) == false

## Thought Puzzle

- What is the opposite of:

  if small<middle and middle<large :

- For that whole statement to be ***false***, at least
  - not(small<middle)
  - **or**
  - not(middle<large)

## Thought Puzzle

- What is the opposite of:

  if small<middle and middle<large :

- For that whole statement to be ***false***, at least
  - (small>=middle)
  - **or**
  - (middle>=large)

## DeMorgan’s Laws

The opposite of:

A and B

Is

opposite (A and B) =

opposite(A) or opposite(B)

## So how do you go the other way?

The opposite of:

A or B

Is

opposite (A or B) =

opposite(A) and opposite(B)

## Reverse these

- num > 0 and num < 10
- status==“student” and amountOwed<500.00
- 0 < inputNum < 100

## What is an Algorithm?

- Process or a set of rules to be followed in calculations or other problem-solving operations

Or, more informally…

- A recipe for solving a problem

## Algorithm vs. Program

- An **algorithm** is a description of how to solve a problem
- A **program** is an implementation of an algorithm in a particular language to run on a computer (usually a particular kind of computer)
- Difference between **description of what we want to do** and **what we actually did**

## Algorithm vs. Program

- So which do we write first?
  - Algorithm?
  - Program?

## What's the Difference, Really?

- We can analyze the algorithm independent of its implementation.
  - Is this the most efficient way to solve a problem?
- We can examine how easily, or with what difficulty, a language allows us to realize an algorithm
  - Some languages allow us to more easily solve certain types of problems than others

## Current Programming Assignment (Partial)

- Add the digits in a number together.
  - 1550 is 1+5+5+0 = 11

# So what was your algorithm?

- Set a sum to zero
- Look at each digit in the number, one at a time
- Add number to a sum

- Is that sufficient?

## A Stronger Algorithm

- Set a sum to zero
- For each digit in the number:
  - Look at the last digit in the number (using %)
  - Add that last digit to a sum
  - Shrink the number by one digit (using //)
  - Repeat until there are no more digits to look at

## Aspects of an Algorithm

- <u>Detailed</u>: Provide enough detail to be implementable. Can be tricky to define completely, relies on “common sense”
- <u>Effective</u>: the algorithm should eventually halt, and halt in a “reasonable” amount of time. “reasonable” might change under different circumstances (faster computer, more computers, etc.)

## Aspects of an Algorithm (2)

- Specify Behavior: the algorithm should be specific about the information that goes in (quantity, type, etc.) and the information that comes out.
- General Purpose: algorithms should be idealized and therefore general purpose. A sorting algorithm should be able to sort anything (numbers, letters, patient records, etc.)

## A Lot to Do!

- That is a lot to do for the burgeoning programmer.
- Get better as we go along, but good to know what the standards are!

## Aspects of a Program

- Readability
- Robustness
- Correctness

## Aspects of a Program: Readability

- We will emphasize, over and over, that a program is an essay on problem solving intended to be read by other people, even if “other people” is you in the future!
- Write a program so that you can read it, because it is likely that sometime in the future **you will** have to read it!
- So what makes a program readable…

## Readability(2): Naming

- The easiest thing to do that affects readability is good naming
  - use names for the items you create that reflect their purpose

## What Does this Do?

```
a = int(input("give a number: "))
b=1
c = 0
while b <= a :
    c = c + b
    b = b + 1
print (a,b,c)
print( "Result: ", c/(b – 1))
```

## What Does this Do?

```
# average of a sum of integers in a given range
limit = int(input("range is 1 to input: "))
count = 1
total = 0
while count <= limit:
    total = total + count
    count = count + 1
average = total / (count - 1)
print("Average of sum of integers from 1 to", \
    limit,"is", average)
```

## What Does this Do?

```
# a simpler solution…
limit = int(input(“range is 1 to input: ”))
total = 0
for i in range(1, limit + 1):
    total = total + i
average = total / (limit)
print(“Average of sum of integers from 1 to”, \
    limit,”is”, average)
```

## Readability(3): Comments

- info at the top, the goal of the code
- purpose of variables (if not obvious by the name)
- purpose of other functions being used
- anything “tricky”. If it took you time to write, it probably is hard to read and needs a comment

## Readability(4): Indenting

- indenting is a visual cue to say what code is “part of” other code.
- This is not always required as it is in Python, but Python forces you to indent.
- This aids readability greatly.

## Aspects of Programming (2)

- Robust: As much as possible, the program should account for inputs that are not what is expected.

## Aspects of Programming (2)

- Correct: Our programs should produce correct results. Much harder to ensure than it looks!

## The Problem is “Problem-Solving”

- Remember, two parts to our goal:
  - Understand the problems to be solved
  - Encode the solution in a programming language, e.g. Python

## Mix of Both

- The goal in this class is to do a little of both: problem solving and Python
- Very important: *try and understand how to solve the problem* ***<u>first</u>*** *before you try and code it.*
    - **Develop the algorithm first!**
    - **Then develop the program.**

## So which is harder?

- Writing the algorithm?
- Writing the program?

- It’s a lot like studying French poetry when you don’t know much about French or poetry…

## Steps to Problem Solving

- Engage/Commit
- Visualize/See
- Try it/Experiment
- Simplify
- Analyze/Think
- Relax

## Engage

You need to commit yourself to addressing the problem.

- Don’t give up easily
- Try different approaches
- Set the “mood”

Just putting in time does not mean you put in a real effort!!!

# Visualize/See the Problem

Find a way that works for you,
some way to make the problem tangible.

- draw pictures
- layout tables
- literally "see" the problem somehow

Everyone has a different way, find yours!

## Try It/Experiment

Sometimes you may be afraid to just “try” some solution. Don’t fear failure: experiments done by yourself are the best way to figure out problems.

Be willing to try, and fail, to solve a problem. Get started, don’t wait for enlightenment!

## Simplify

Simplifying the problem so you can get a handle on it is one of the **<u>most powerful</u>** problem solving tools.

Given a hard problem, make is simpler (smaller, clearer, easier), figure that out, then ramp up to the harder problem.

*If you can't solve a problem, then there is an easier problem you can solve: find it. --George Polya (1945)*

## Think it Over/Analyze

If your solution isn’t working:

- stop
- evaluate how you are doing
- analyze and keep going, or start over.

## One More Thing: Relax

Take your time. Not getting an answer right away is not the end of the world. Put it away and come back to it.

You'd be surprised how easy it is to solve if you let it go for awhile. That's why **<u>starting early</u>** is a luxury you should afford yourself.