

# Basic problem solving

How to go from problem description to code

Terence Parr

MSDS program

**University of San Francisco**

*BTW, This is an abbreviated version of what we normally talk about in data structures*

# Overall process

1. Understand
2. Solve
3. Code (notice this is a separate step from #2)
4. Verify/test



# 1. Understand

- Read the description (3x) then restate the problem, either on paper or out loud (yes, I talk to myself in my office)
- Describe and write out a minimal but *representative* example of
  - the intended input data or data structure *and*
  - the expected output *and*
  - ask if the example is correct
- Identify any edge cases you can think of by example, but don't focus on those cases initially

*act* → *cat*  
*its* → *sit*

# Reading problem descriptions

- Details matter, pay careful attention to the description
  - Pretend that someone is trying to trick you with the problem description!
  - Are the input data elements strings, ints, floats?
  - If data is numeric, are they always between 0 and 1? Can they be negative?
  - Is the input sorted?
  - Can you see all of the data at once or must you worry about streaming data?
  - Can you bound the maximum size of the input (e.g., to fit in memory)?
- When reading description, identify who is doing what to whom?
  - What are the nouns and verbs used in the description?
  - The nouns are usually data sources or data elements
  - The verbs are often operations you need to perform
  - Look for keywords like min, max, average, median, sort, argmax, sum, find, search, collect, filter out, select, compute, etc...



## 2. Solve



# Key ideas for solving problems

- Solving the problem has nothing to do with the computer
- You might not even be asked to code the solution
- If you can't walk through a correct sequence of operations **by hand** on paper, no amount of coding skill will help you!
- All the good programmers I know keep a notepad next to their computers, and it is full of boxes, bubbles, arrows, and notes
- It helps to use established **patterns**, templates, strategies, and common data transformation operations as a crutch

# Strategies for solving problems

1. *Start with the end result and work your way backwards*
  - Ask what the prerequisites are for each step
  - The processing step or steps preceding step  $i$  compute the data or values needed by step  $i$
  - E.g., median: to pick middle value, previous step must sort data
2. *Reduce or simplify a new problem to a variation of an existing problem with a known solution*
  - Ask what the difference is between the problem you're trying to solve and other problems for which you have a solution
  - E.g., Engineers building a new suspension bridge do not proceed as if such a thing has never been built before

# Requisite mathematician joke

- *“A physicist and a mathematician are sitting in a faculty lounge. Suddenly, the coffee machine catches on fire. The physicist grabs a bucket and leaps towards the sink, fills the bucket with water, and puts out the fire.*

*Second day, the same two sit in the same lounge. Again, the coffee machine catches on fire. This time, the mathematician stands up, gets a bucket, hands the bucket to the physicist, thus, reducing the problem to one with a known solution.”*



# Steps in “solve” phase

## A. Explore

- Look at the input-output example and imagine how you can manually operate on the input to get the output (using fingers on paper)
- Attempt any manual sequence of operations that appears to be in the right direction, even if you know it's not quite right
- Exploration helps you understand the problem and will trigger more questions, so ask questions
- Write down what you know; e.g., in a geometry problem, you might know which angles or arcs are the same length or that you can make two triangles out of the given rectangle by drawing a line

# Steps in “solve” phase continued

## **B. Reduce**

- Can you reduce the problem to known solution by preprocessing the input a bit?

## **C. Reuse**

- Look for and reuse familiar programming patterns like vector sum, min, sort, map, filter, and find
- E.g., to sort a list of numbers (slowly), repeatedly pull then delete the minimum value out of one array and add it to the end of another.

## **D. Systematize**

- Simplify and organize the steps in your process as pseudo-code
- This is your algorithm

# Steps in “solve” phase

## **E. Verify algorithm / process**

- Check that your algorithm solves the main problem and the edge cases.
- Later, when you have more experience, you will need to check your algorithm's complexity (performance as function of input size)

### 3. Translate your algorithm to code



- A. Write a function that takes your input as parameter(s)
  - Return value of function will typically be the expected problem result
- B. Write a main script that:
  - acquires the data
  - passes it to your function
  - sends the results to the appropriate file or standard output
- C. Translate the algorithm steps into statements in your function  
(It's okay if you create helper functions)

## 4. Verify/test

- Test your code on the representative examples you identified early on in this process
- Now, try some edge cases, which will likely break your code
- Go back to the algorithm and process design phase and alter it to handle the edge cases
- Translate the changes to code
- Verify that you did not break the representative examples and then test on the edge cases

# Unit tests

- In a job situation, you'd encode these tests as “unit tests”
- These tests are reproducible and should check edge cases, representative examples, and examples that should fail or cause exceptions
- All code changes over time, which can introduce bugs
- These tests are your primary line of defense against the introduction of bugs in working code (so-called "regressions")
- This is the difference between an amateur and a professional programmer; you cannot safely change code without tests that check the sanity of your system
- For machine learning scripts that just develop models, this might be less true, but it is very true for large or complex systems