

HoP101: Session 1

Computational Thinking and Introduction to Programming

Mrigank Pawagi

Indian Institute of Science

ACM-W Student Chapter, Summer 2023



The problem-solving process

- Decomposition
- Pattern recognition
- Abstraction
- Algorithm design



The problem-solving process

Decomposition

- Breaking down a complex problem into smaller, more manageable subproblems
- Each subproblem is solved individually
- The solutions to the subproblems are combined to solve the original problem



The problem-solving process

Pattern recognition

- Identifying similarities among and within problems and solutions
- Using these similarities to solve new problems



The problem-solving process

Abstraction

- Identifying useful details in the pattern
- Using the similarities to create a general solution to a problem



The problem-solving process

Algorithm design

- Developing step-by-step instructions for reaching the solution for any input
- The instructions are unambiguous and terminate



Some examples

Problem

Given a list of numbers, find the maximum number in the list.

Problem

Join two sorted lists of numbers such that the resulting list is also sorted.
Can this help us sort any given list of numbers?



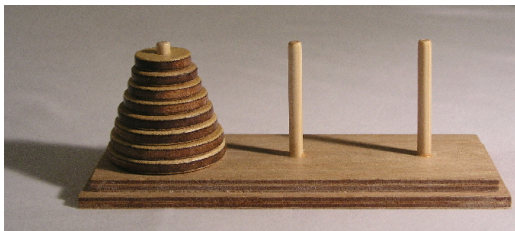
Advanced example (homework): Tower of Brahma/Hanoi

Problem

Given a stack of n disks, find the minimum number of moves required to move the stack from one peg to another, given the following constraints:

- Only one disk can be moved at a time
- A disk can only be moved if it is the uppermost disk on a stack
- No disk may be placed on top of a smaller disk

Describe the sequence of moves as well.



Source: Evanherk on Wikimedia Commons



All of this is important not just for problem solving,
but also for "solving problems", i.e.,
writing computer programs!



What is a programming language?

- It is essentially just a translation interface.
- This in turn enforces a certain structure called **syntax**.



"Turing" completeness for programming languages

Disclaimer

This is not an accurate definition of turing completeness.

What is completeness?

A programming language is said to be (turing) complete if it can be used to solve any problem that can be solved by a "turing machine".

In our context, this means almost all problems that we can think of and will come across in everyday life.



"Turing" completeness for programming languages

Baby criteria for completeness

Ability to perform conditional branching. This immediately implies several other things.

- Ability to perform certain operations (arithmetic, logical, etc.)
- Ability to read and write from something like a memory (think variables!)
- Ability to jump to any point in the program (think loops!)
 - It is necessary to allow unbounded loops for completeness



"Turing" completeness for programming languages

The complete characterization also involves arbitrary amounts of memory, but this is impossible in practice and thus true "turing" completeness can never be achieved. What we really mean is that some systems have the ability to approximate Turing-completeness up to the limits of their available memory.



If we could break down any problem into basic steps consisting of arithmetic and logical operations, reading and writing to memory, and jumping to different parts of the program, we could solve any problem using a computer program.



Why do we have so many programming languages?

In real world, we optimize

- for performance
- for ease and speed of production

Just like in research, an important idea in programming is to build over things made before us, i.e. "standing on the shoulders of giants".



Why isn't programming very simple?

It really is!

We often abstract away several intricate combinations of basic constructs to make life easy, and work with higher order objects. It is often these which make things tricky for beginners.

- Objects
- Classes
- Functions
- Iterators
- Structures



Let us see how Python is complete!



Let me know if you have any issues with

- Installation
- Git
- Anything else



This will seem a bit too loaded at the moment, but hang on! We have the problem solving process to help us.

How can we simulate the spread of a new disease and determine its long-term effects, like the number of people left infected?



The task

- Decompose this question into smaller sub-problems, and then those sub-problems into sub-sub-problems, and so on. Try to go as far as you can.
- Try to come up with narrower questions that can be dealt with individually.
- (Less important) Try to come up with an implementation strategy based on these sub-problems.

This will be discussed on teams before the next session.

