Session 1

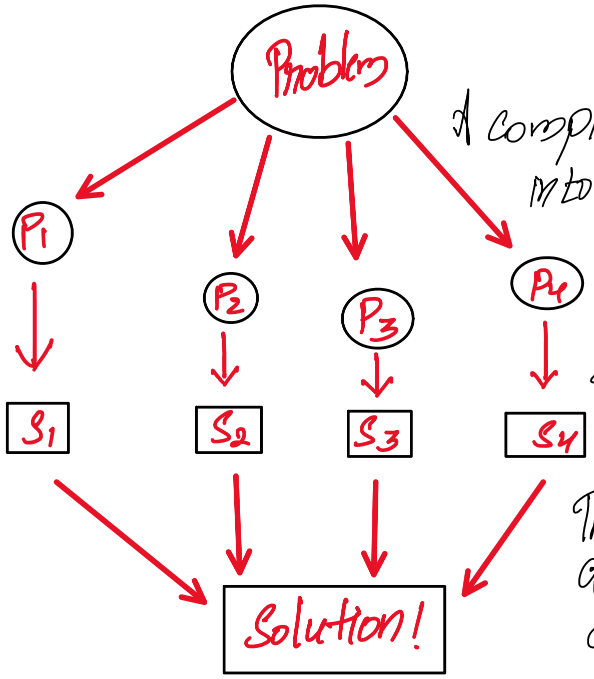
Monday, 26 June 2023 5:59 PM

Computational Thinking and Introduction to Programming

The Problem Solving Process

1. Decomposition
2. Pattern Recognition
3. Abstraction
4. Algorithm Design

① Decomposition



A complex problem is broken down into smaller, more manageable

→ Subproblems. [P1, P2, P3 . . . . .]

Each subproblem is solved individually.

→ (individual solutions) (S1, S2 . . .)

The solutions to the subproblems are combined to solve the original problem.

By identifying similarities among and within problems and solutions in this system, we can use these similarities to solve new problems → ② Pattern Recognition.

③

By identifying useful details in

③ Abstraction

By identifying useful details in the pattern we can use the similarities to create a general solution to a problem.

④ Algorithm Design

- Developing step-by-step instructions for reaching the solution for any input.

(instructions → unambiguous and terminate.)

Some Examples

Problem 1: Given a list of numbers, find the maximum number in the list.

Soln:

Lets try to decompose this problem into subproblems.

That would be to find the maximum of first two numbers, then the first three numbers and so on.....

Do we see a pattern here?

Certainly, we do. When we find the max of first 2 numbers we compare it with the third number which is again narrowed down to comparison of two numbers as we did in the beginning! This pattern keeps repeating till the end where we compare the last number with the max of all the previous numbers.

To make this more general, lets find the max of one number - which is that number itself. Store this and compare it to the next number and so on.... We have found the algorithm as well.

Problem 2: Join two sorted lists of numbers such that the resulting list is also sorted. Can this help us sort any given list of numbers?

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

PROGRAMMING LANGUAGES

What is a programming language?
→ essentially just a translation interface.
→ This in turn enforces a certain structure called **SYNTAX**.

CODE —SYNTAX→ BINARY

helps in conversion without any ambiguity.

An idea of 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.

Baby Criteria for Completeness: Ability to perform:

- ability to perform certain operations (arithmetic, logical, etc)
- CONDITIONAL BRANCHING
- 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.

Let's dive into each one of these conditions one by one.

Assume two variables x and y
How do we make an algorithm to perform certain actions?
For example, if $x < y$
- Yes → execute code of branch 1
- No → execute code of branch 2.

For example, if $x<y$ ⟨ ... execute code of branch 1
No → execute code of branch 2.

condition — branching

①
Arithmetic operations: code should be able to perform +, −, ×, ÷ etc...
logical operations: AND, OR etc

② Ability to read and write from memory.

When we ask our computer to perform an operation (1+2)
How do we feed 1 and 2 in the computer?

The Process ⟹ We store 1 in one part of the memory
and 2 in some other part of the memory.
Ask the computer to perform the operation (+) on these two parts of the memory and store the output in some other part of the memory from where we can eventually read the output (3).

For example: Input: Print (Hello) → written in the memory for a short period of time.
read by screen / hardware ↵

③ Jump to any point in the program.

In our example when we say
if $x<y$, when this condition is fulfilled the code jumps to the line which says $max = y$
otherwise it jumps to the line which says $max = x$.

This gives us one more power: **we can loop in the program!**

1. Print (Hello)
2. Go back to (1)

} → we have created a loop.

3. if user has clicked spacebar, go to 4.
4.

} we have exited the loop.

The complete characterization also involves arbitrary amounts of

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 limits of their available memory.

Why do we have so many Programming Languages?

In real world, we optimize

- for performance
- for ease of production.

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

Advanced example (HW): Tower of Brahma/Hanoi