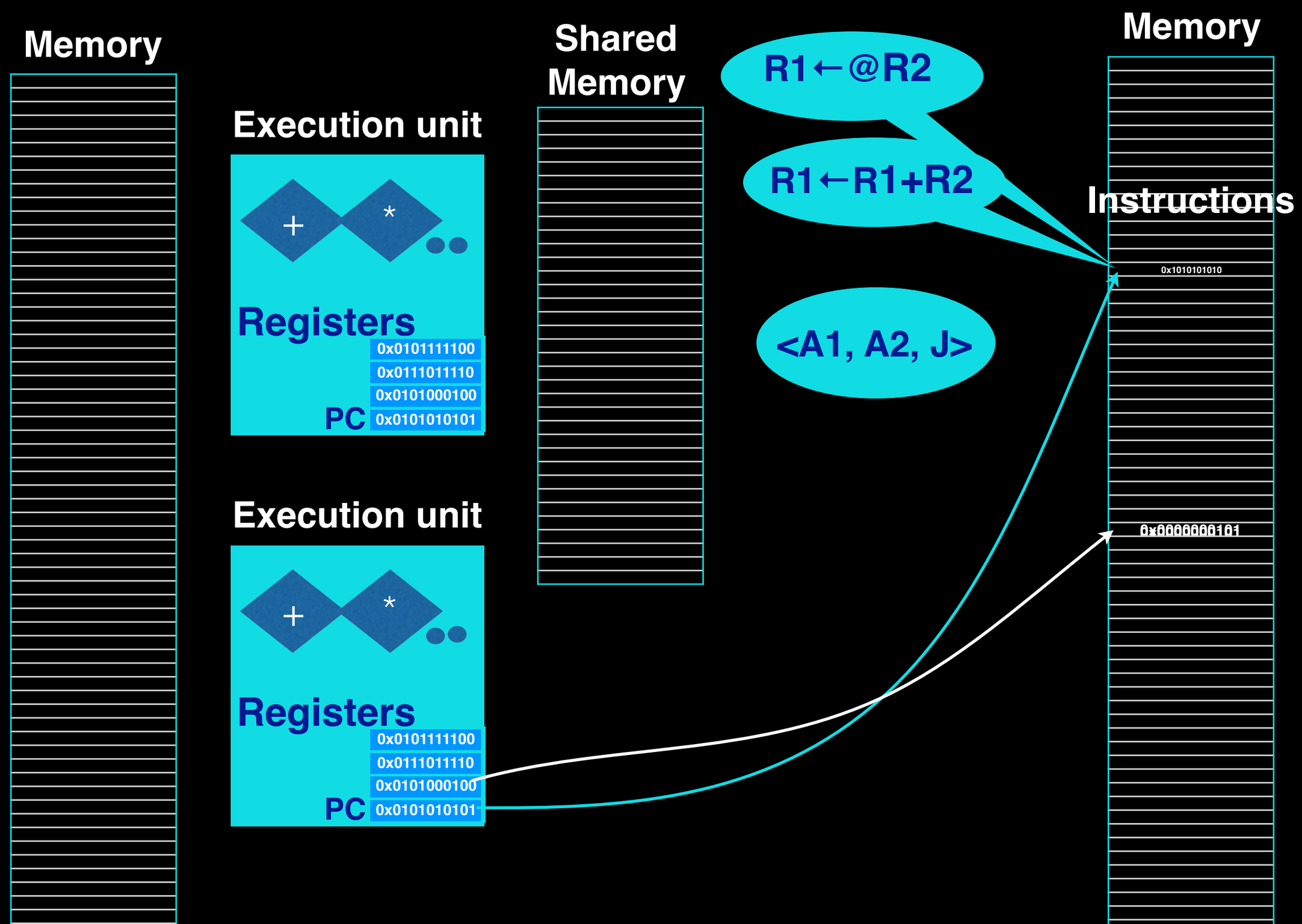
Introduction to CS

- Not a study of computers but computation
  - But we do study computers as well
- Computability
- How to formally specify and abstract problems
  - And design and evaluate solutions
- Algorithms and data management
  - Efficient computation

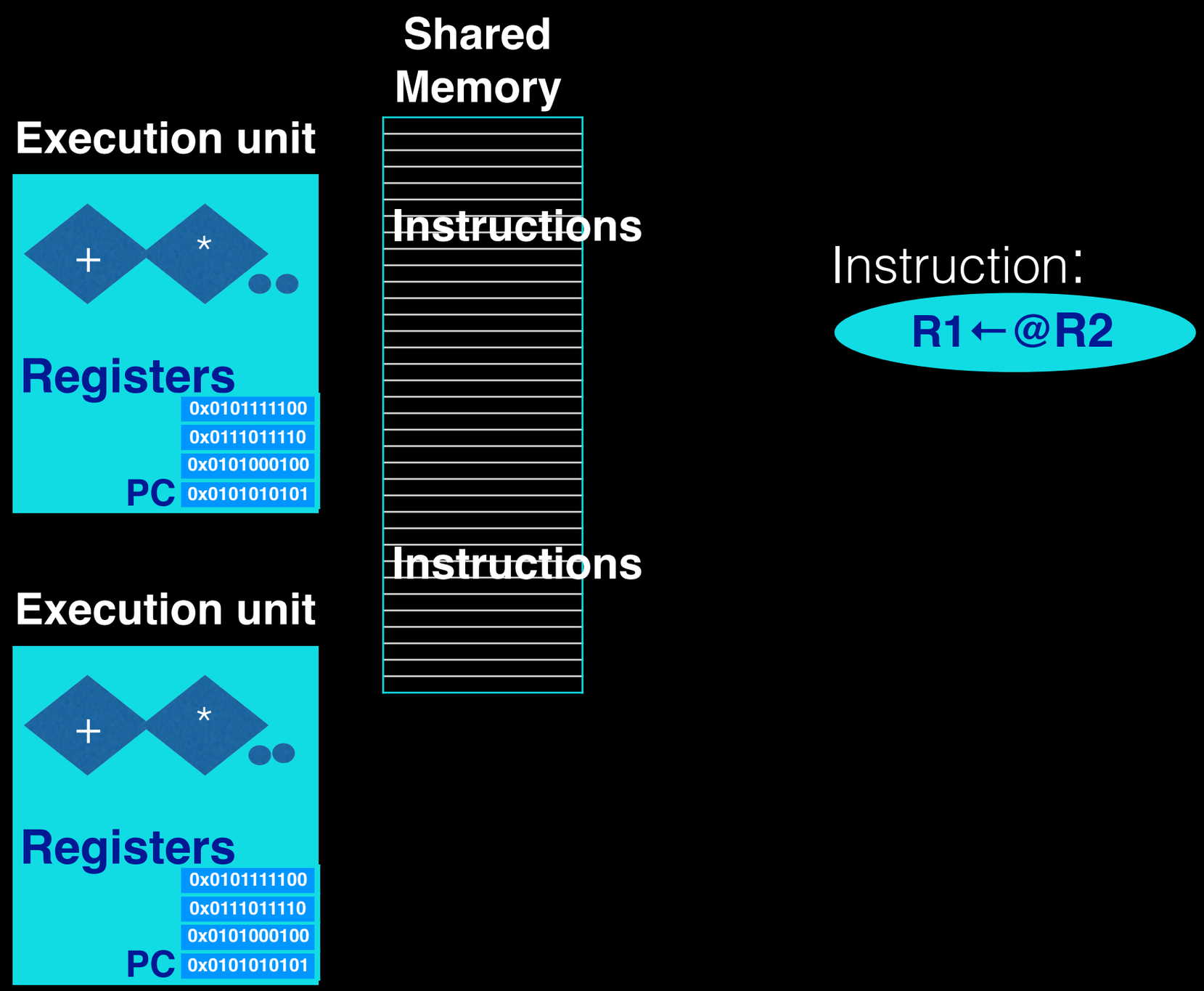
What is an Algorithm

- Self-contained *set of actionable steps* that lead to a solution
  - with order among (some) steps clearly spelled out
- Incomplete without
  - an understanding of steps
  - an understanding of input requirements
  - an understanding of output requirements

Von-Neumann Model



Von-Neumann Model



What is Programming?

- Express the problem formally
  - Then say “Solve(Problem)”
- Formal vs Natural language
- High level language
  - Preferably Turing complete
  - Declarative style
  - Imperative Style
  - Object-oriented
  - Syntax & Semantics

Language Styles

Declarative

```
x, where x*x = n
```

(functional)

Imperative

```
x = 0
while( x < n):
   if(x*x == n):
      return x
   x = x+1
```

Object-oriented

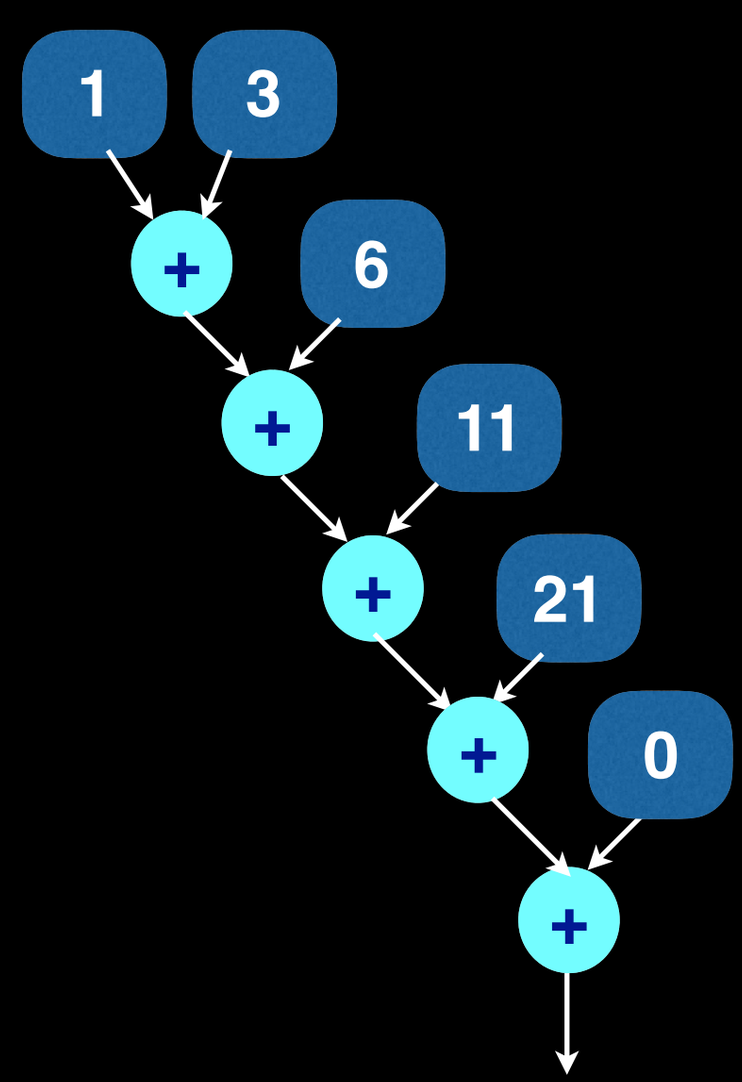
```
x = n.sqrt()
```

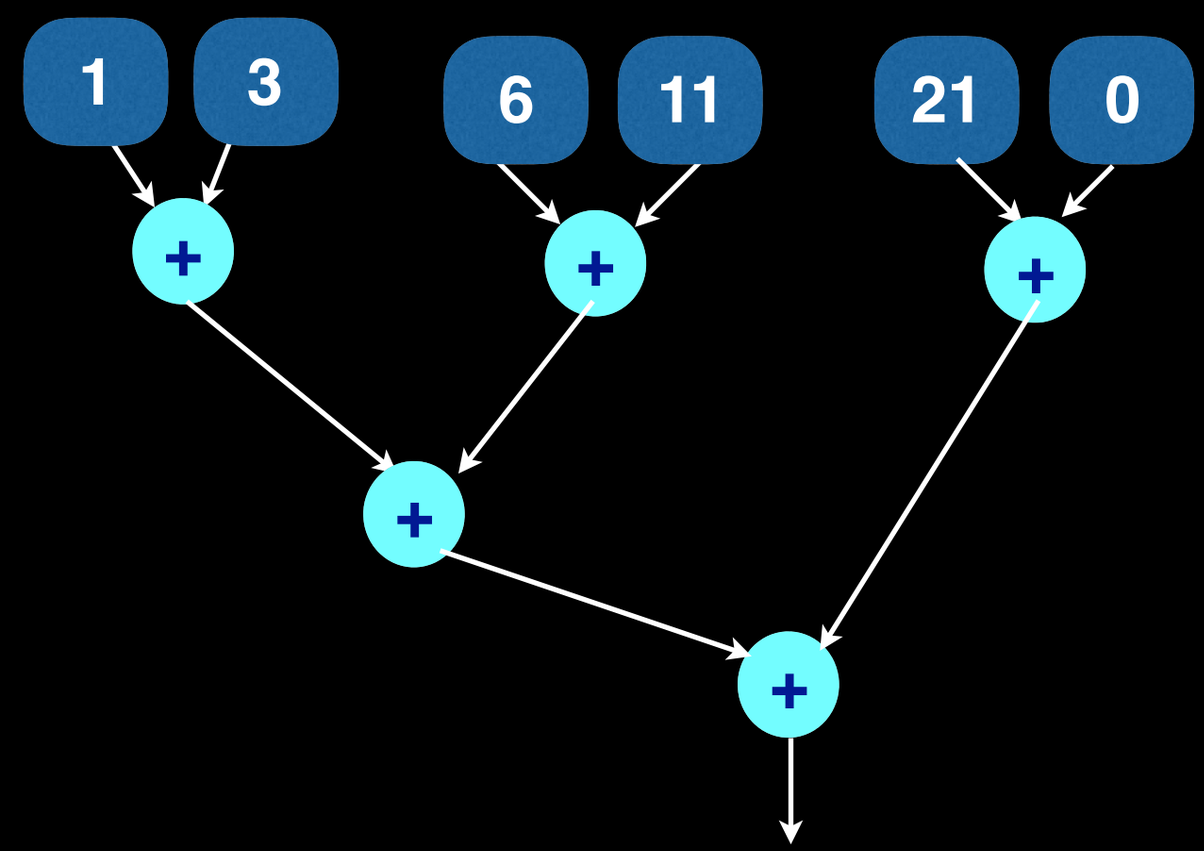
Object method sqrt:

```
y = 0
while( y < me):
   if(y*y == me):
      return y
   x = x+1
```

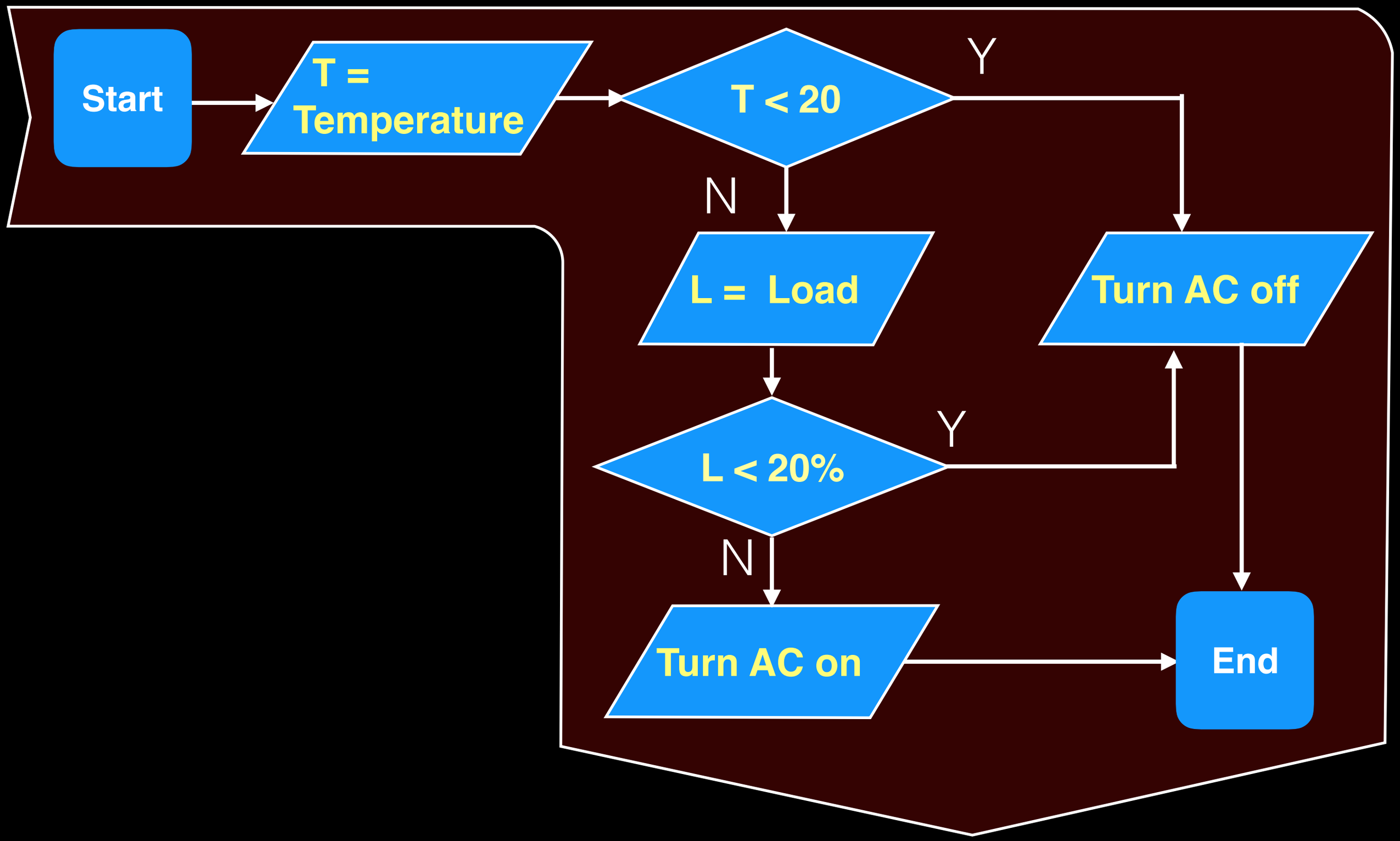
Program as
A Graph of Simpler Tasks

SUM: [1, 3, 6, 11, 21, 0]





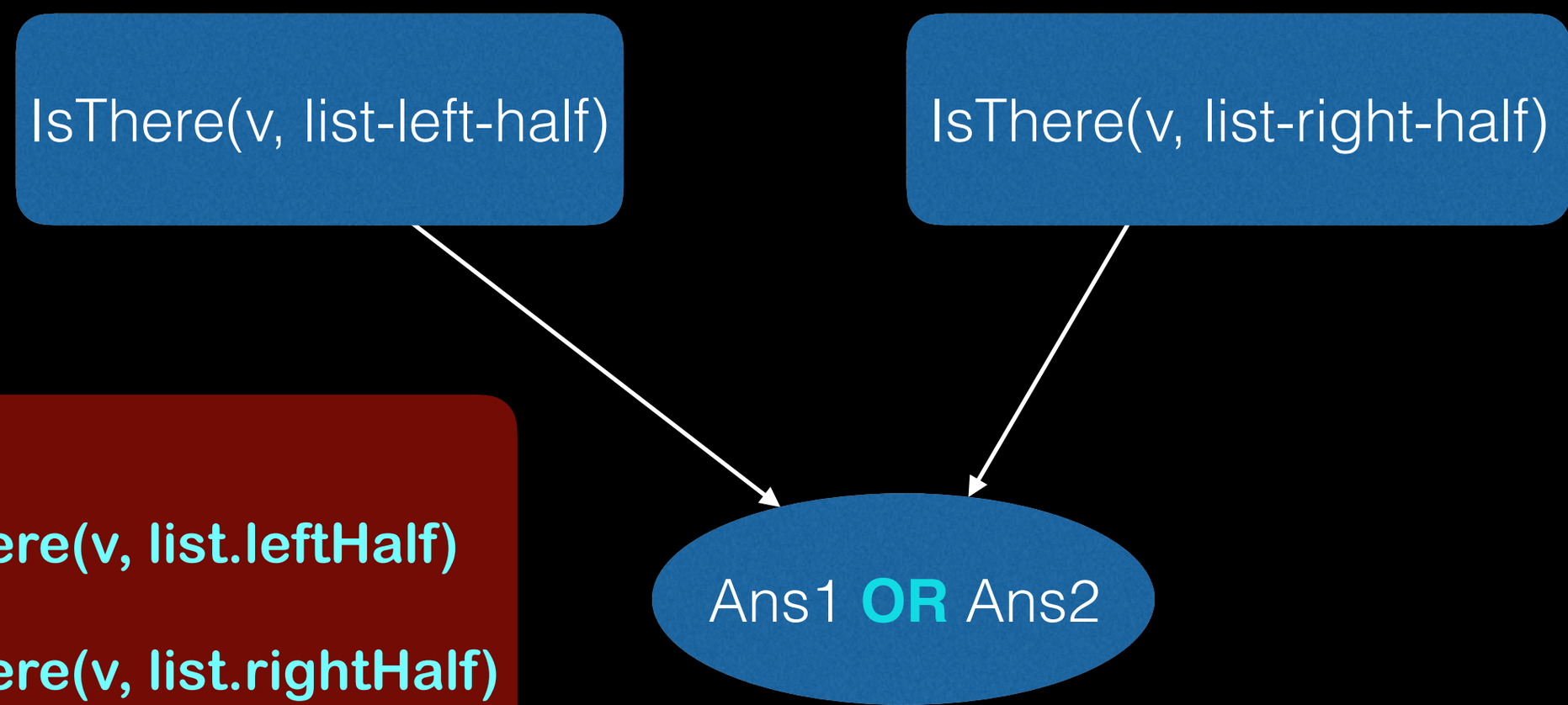
Flow Chart


Start
T = Temperature
T < 20
Y
N
L = Load
Turn AC off
L < 20%
Y
N
Turn AC on
End

Programming is Problem-Solving

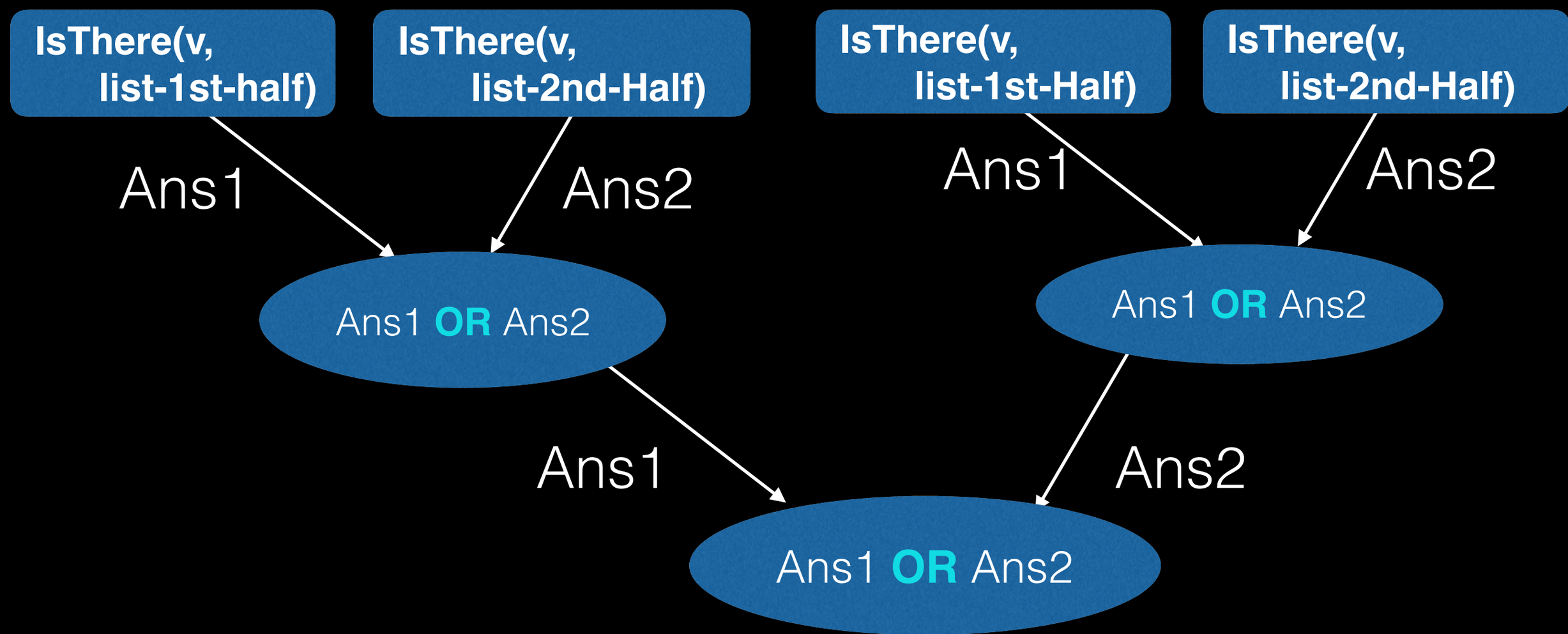
- Understand the problem: output for each input
  - Formalize problem specification
- Formulate the over-all structure of the algorithm
  - Coarse steps first
  - Refine each step into simpler steps
  - Until you know how to implement those steps
- Implement, Test & Maintain

IsThere(v, list)?

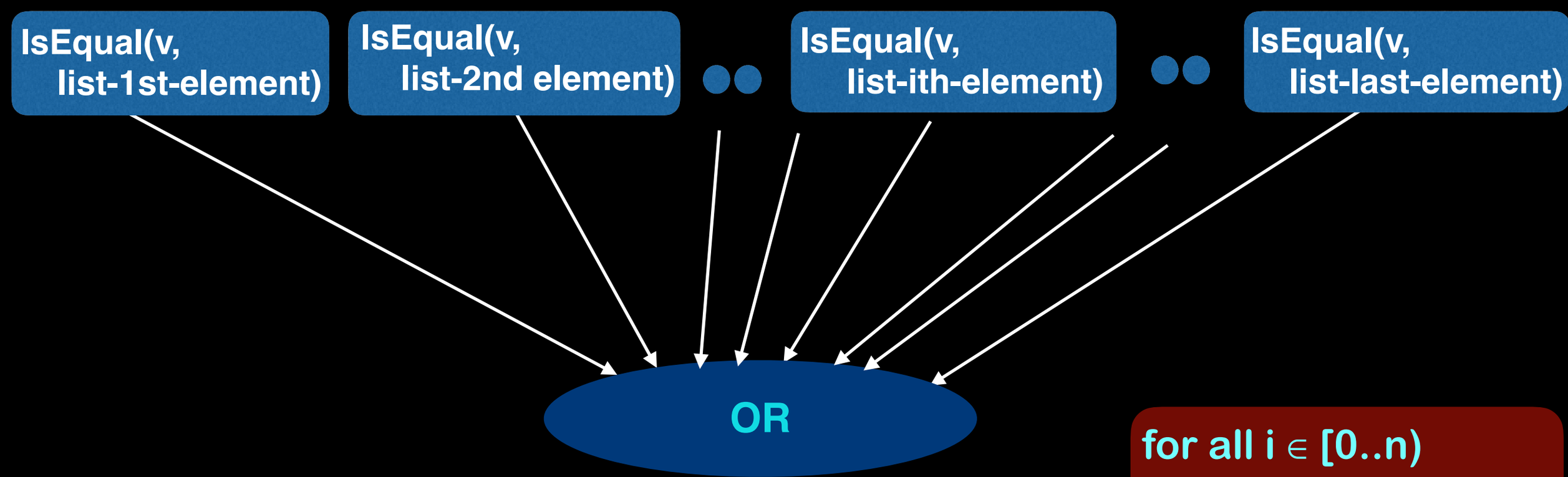



```
return
  isThere(v, list.leftHalf)
  OR
  isThere(v, list.rightHalf)
```

IsThere(v, list)?



IsThere(v, list)?



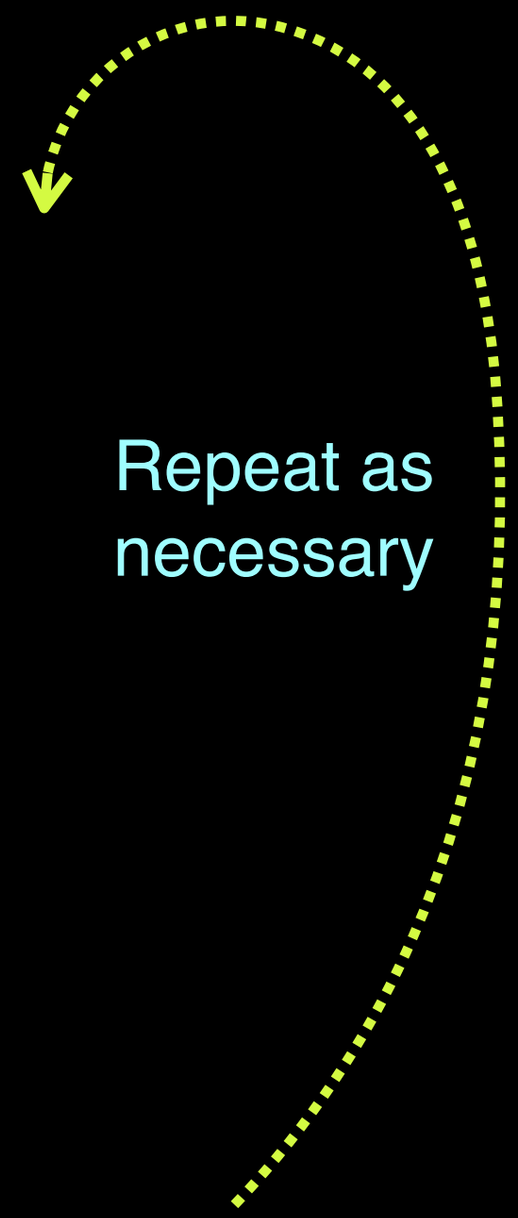

```
for all i ∈ [0..n)
    if(list[i] == v)
        return TRUE

return FALSE
```

How does a Program Look?

- A set of instructions in a “programming language”
- May maintains state
  - Notion of Variables
  - Or, name and binding
  - Collections
- May take action based on (some part of) current state
- May repeatedly take action
- May interact with other programs, people, or devices

Programming Steps

- Understand specification
  - Formulate as formally as you can
- Devise the test plan
- Algorithmic design
  - Analyse the performance
- Refine the test plan
- Implement incrementally
  - Test each time
  - Error debugging + performance debugging



Programming Errors

- Syntax/ Semantic errors
- Crash
  - Exception, Illegal access, Resource unavailability, System fault
- Hang

Termination

- Any program without loops or recursion terminates
- For loop
  - Find an integer function of some program variables
  - Integer value is non-negative at the start of the loop
  - Integer value is zero at the end of the loop
  - Value of guaranteed to decrease progressively
- For a recursive function:
  - every recursive call will eventually reach a basis case

Every execution

Test Termination

```
def isThere(v, list):
    for l in list:
        if(l == v):
            return TRUE
return FALSE
```

v = length of list - position of l in list - 1

```
def factorial(n):
    fact = 1
    for i in xrange(1, n+1):
        fact = fact * i
    return fact
```

v = n - i if n > 0

0 otherwise

Programming Errors

- Syntax/ Semantic errors
- Crash
  - Exception, Illegal access, Resource unavailability, System fault
- Hang
- Wrong answer
  - Occasionally wrong

Program Correctness

- Starts with a correct specification of the requirements
  - Correctness can only be with respect to the specification
- Correct design
  - Algorithmically correct
- Correct implementation
  - Re-use of already correct code helps

Safe Coding Style

- Indicative names
  - Comment
- Always check for error value returned by functions
  - Handle exception
- Validate user input
- Assert known state

Prove Correctness

```
def isThere(v, list):
    for l in list:
        if(l == v):
            return TRUE
return FALSE
```

Check all termination conditions

```
def factorial(n):
    fact = 1
    for i in xrange(1, n+1):
        fact = fact * i
    return fact
```

fact = (i-1)!

fact = i!

Induction on n

Python: Understand Error Report

- SyntaxError
- IndentationError
- TypeError
- NameError
- IndexError
- UnboundLocalError
- AssertionError