# #3: Enumerative Search

**Sankha Narayan Guria**

EECS 700: Introduction to Program Synthesis

# Logistics

- **Paper reading #1 is out!**
- **Canvas Discussion Board**
  - To discuss paper
  - To find project teammate
- **Other questions?**

# Regular tree grammars (RTGs)

nonterminals rules (productions)

alphabet starting nonterminal

<Σ, N, R, S>

- Trees: $\tau \in T_\Sigma(N)$ = all trees made from $N \cup \Sigma$
- Rules are of the form: $A \to \sigma(A_1, \dots, A_n)$
- Derives in one step: $\mathcal{C}[A] \to \mathcal{C}[t]$ if $(A \to t) \in R$
  $A$ is the leftmost non-terminal in $\mathcal{C}[A]$
- Incomplete terms/programs: $\{\tau \in T_\Sigma(N) | A \to^* \tau\}$
- Complete terms/programs: $\{t \in T_\Sigma | A \to^* t\}$
  = programs without holes
- Whole programs: $\{t \in T_\Sigma | S \to^* t\}$
  = roughly, programs of the right type

```
concat(L,0)
```

$L \to concat(L, L)$

```
concat(L,L) -> concat(x,L)

find(concat(L,L),N)

find(concat(x,x),0)

sort(concat(L,L))
```

# SyGuS problems

- SyGuS problem = < theory, spec, grammar >

can inductive synthesis handle these?

A first-order logic formula over the theory

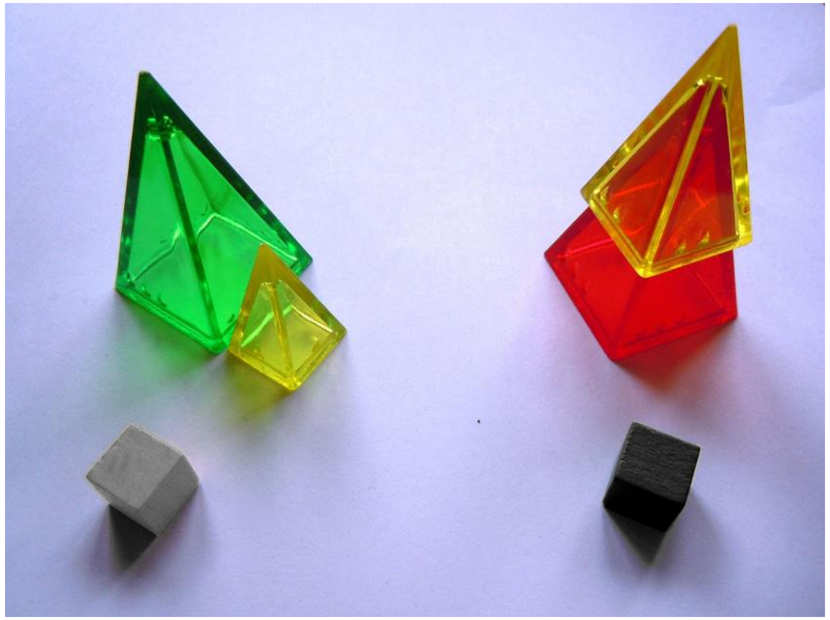
Examples:

```
f(0, 1) = 1 ∧
f(1, 0) = 1 ∧
f(1, 1) = 1 ∧
f(2, 0) = 2
```

Formula with free variables:

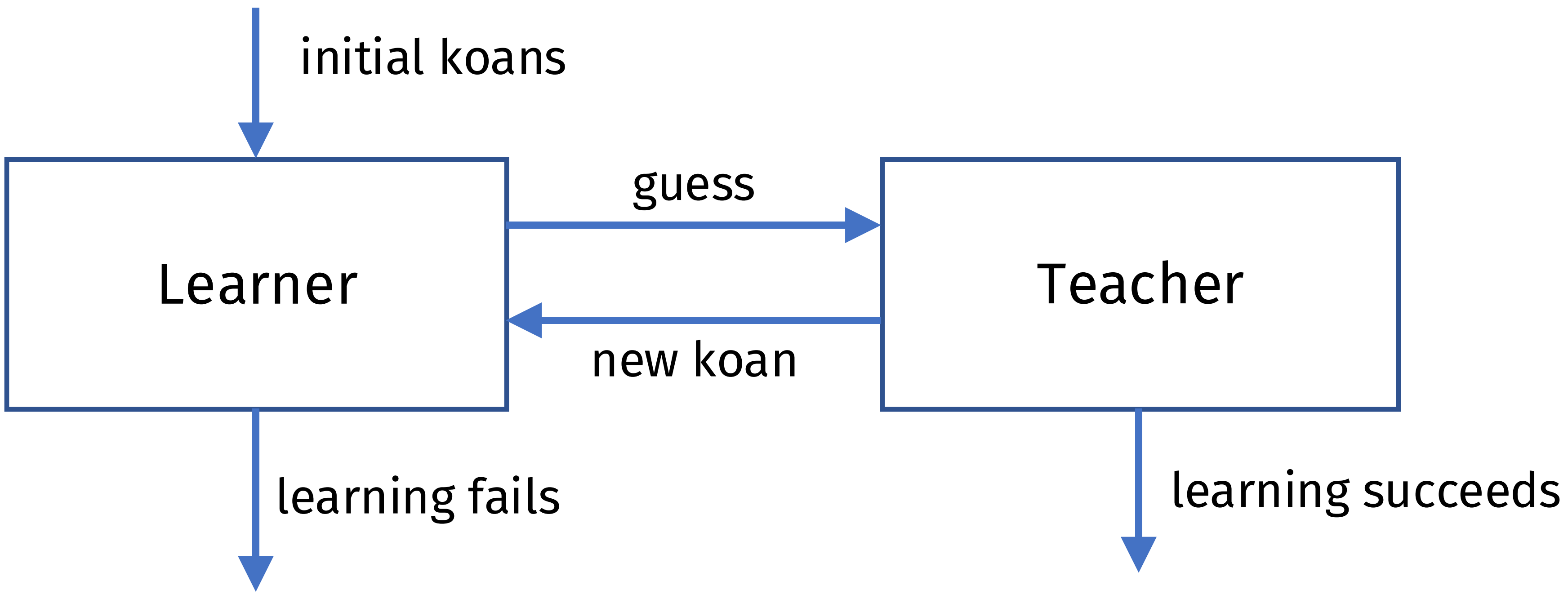
```
x ≤ f(x, y) ∧
y ≤ f(x, y) ∧
(f(x, y) = x ∨ f(x, y) = y)
```

# The Zendo game



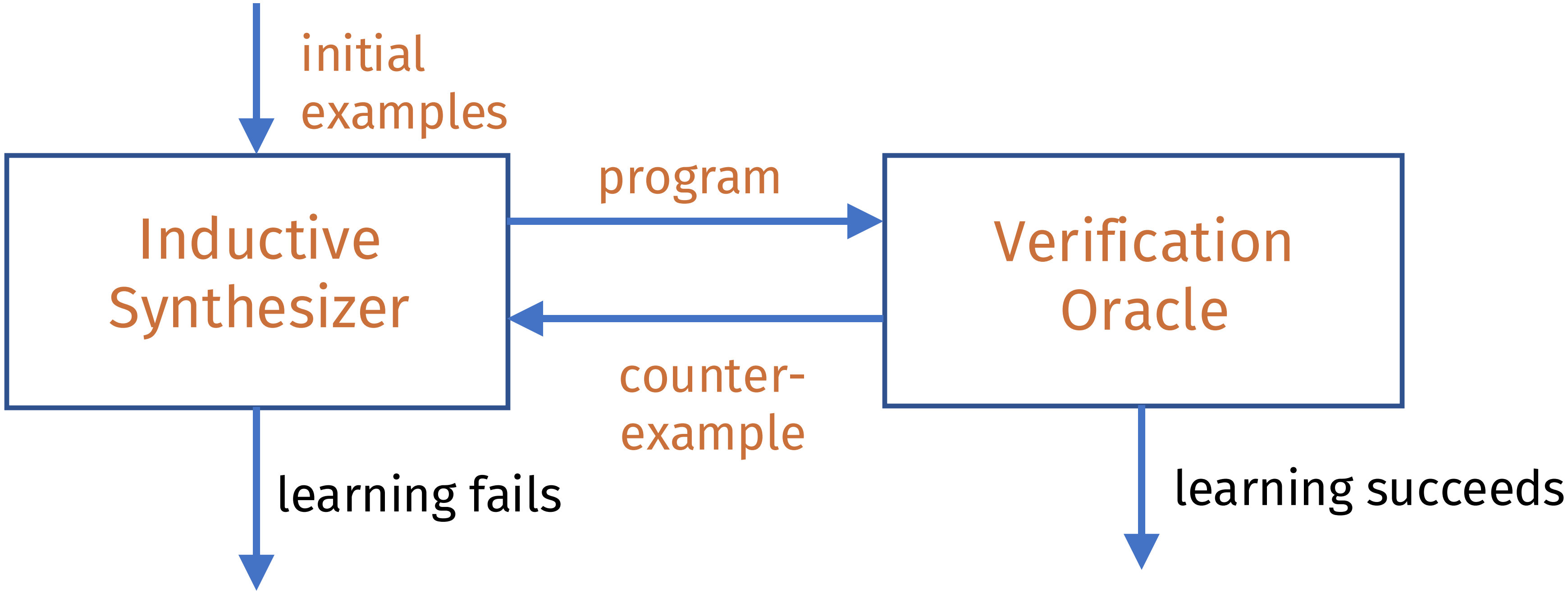
- The teacher makes up a secret rule
  - e.g. all pieces must be grounded
- The teacher builds two koans (a positive and a negative)
- A student can try to guess the rule
  - if they are right, they win
  - otherwise, the teacher builds a koan on which the two rules disagree

# The Zendo game

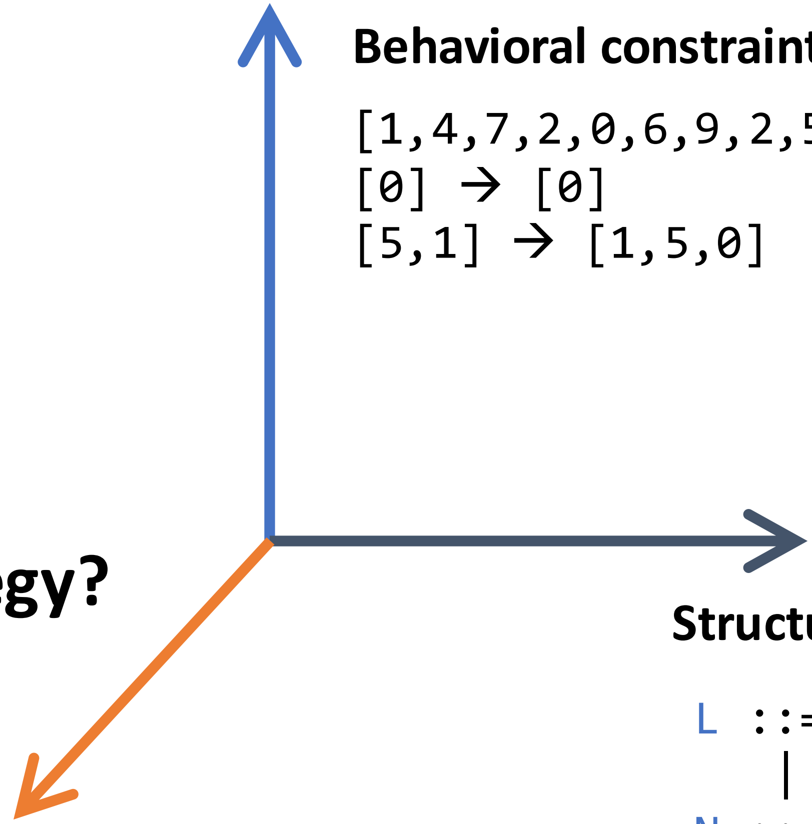


# Counter-example guided inductive synthesis (CEGIS)

*The Zendo of program synthesis*



# The problem statement



**Behavioral constraints = examples**

```
[1,4,7,2,0,6,9,2,5]  →  [1,2,4,7,0]
[0] → [0]
[5,1] → [1,5,0]
```

**Search strategy?**

**Structural constraints = grammar**

```
L ::= sort(L)  |  L[N..N]
    |  L + L  |  [N]  |  x
N ::= find(L,N)  |  0
```

# Enumerative search

# Enumerative search

=

Explicit / Exhaustive Search

Idea: Enumerate programs from the grammar one by one and test them on the examples

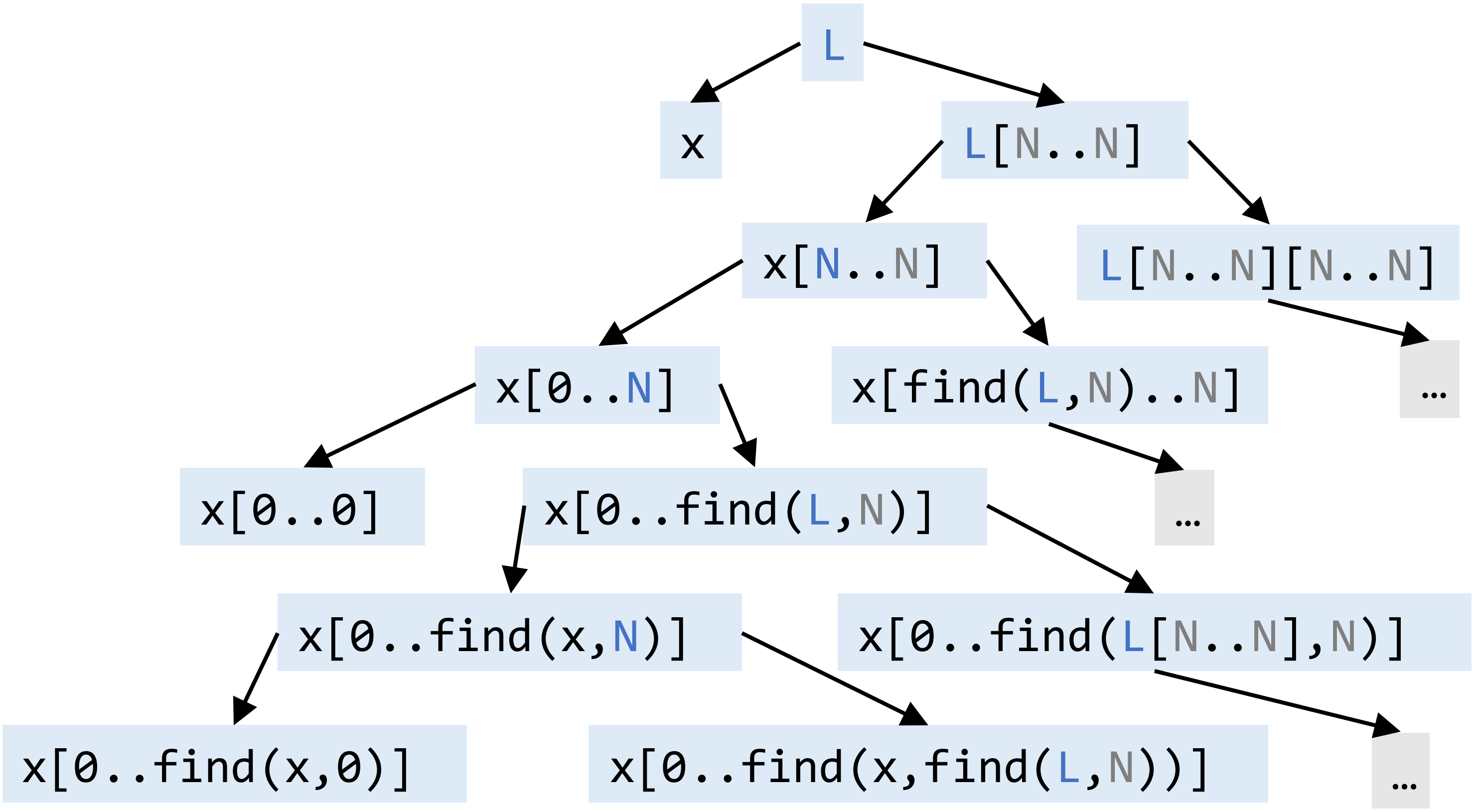
Challenge: How do we systematically enumerate all programs?

top-down vs bottom-up

# Top-down enumeration: search space

Search space is a tree where

- nodes are whole incomplete programs
- edges are "derives in one step"




```
L ::= L[N..N]    |
      x
N ::= find(L,N)  |
      0
```

[[1,4,0,6] → [1,4]]

# Top-down enumeration = tree traversal

Search tree can be traversed:

- depth-first (for fixed max depth)
- breadth-first
- later in class: best-first

General algorithm:

- Maintain a worklist of incomplete programs
- Initialize with the start non-terminal
- Expand left-most non-terminal using all productions

```
L ::= L[N..N]    |
      x
N ::= find(L,N)  |
      0

[[1,4,0,6] → [1,4]]
```

# Top-down: algorithm

nonterminals rules (productions)

alphabet starting nonterminal

```
top-down(<Σ, N, R, S>, [i → o]):
  wl := [S]
  while (wl != []):
    τ:= wl.dequeue()
    if (complete(τ) ∧ τ([i]) = [o]):
      return τ
    wl.enqueue(unroll(τ))

unroll(τ):
  wl’ := []
  A := left-most non-term in τ
  forall (A → rhs) in R:
    τ’ = τ[A -> rhs]
    if !exceeds_bound(τ’): wl’ += τ’
  return wl’
```

depth- or breadth-first
depending on where you enqueue

can impose bounds on depth/size

```
L ::= L[N..N] |
      x
N ::= find(L,N) |
      0

[[1,4,0,6] → [1,4]]
```

# Top-down: example (depth-first)

Worklist `wl`

iter 0: `L`

iter 1: `x` `L[N..N]`

iter 2: `L[N..N]`

iter 3: `x[N..N]` `L[N..N][N..N]`

iter 4: `x[0..N]` `x[find(L,N)..N]` `L[N..N][N..N]`

iter 5: `x[0..0]` `x[0.. find(L,N)]` `x[find(L,N)..N]` …

iter 6: `x[0.. find(L,N)]` `x[find(L,N)..N]` …

iter 7: `x[0.. find(x,N)]` `x[0.. find(L[N..N],N)]` …

iter 8: `x[0.. find(x,0)]` `x[0.. find(x,find(L,N))]` …

iter 9:

```
L ::= L[N..N] |
      x
N ::= find(L,N) |
      0
```

`[[1,4,0,6] → [1,4]]`

# Bottom-up enumeration

- The dynamic programming approach
- Maintain a bank of complete programs
- Combine programs in the bank into larger programs using productions

```
L ::= sort(L)    |
      L[N..N]    |
      L + L      |
      [N]        |
      x
N ::= find(L,N)  |
      0
```

```
[[1,4,0,6]  →  [1,4]]
```

# Bottom-up: algorithm (take 1)

nonterminals rules (productions)

alphabet starting nonterminal

```
bottom-up (<Σ, N, R, S>, [i → o]):
  bank := {}
  for d in [0..]:
    forall (A → rhs) in R:
      forall t in new-terms(A→rhs, d, bank):
        if (A = S ∧ t([i]) = [o]):
          return t
        bank += t;

new-terms(A → σ(A_1…A_k), d, bank):
 if (d = 0 ∧ k = 0) yield σ
 else forall <t_1,…,t_k> in bank^k:
        if A_i ->* t_i: yield σ(t_1,…,t_k)
```

```
L ::= sort(L)    |
      L[N..N]    |
      L + L      |
      [N]        |
      x
N ::= find(L,N)  |
      0
```

[[1,4,0,6] → [1,4]]

inefficient, better index bank by non-terminal!

# Bottom-up: algorithm (take 2)

```
bottom-up (<Σ, N, R, S>, [i → o]):
  bank[A] := {} forall A
  for d in [0..]:
    forall (A → rhs) in R:
      forall t in new-terms(A→rhs, d, bank):
        if (A = S ∧ t([i]) = [o]):
          return t
        bank[A] += t;

new-terms(A → σ(A₁…Aₖ), d, bank):
 if (d = 0 ∧ k = 0) yield σ
 else forall <t₁,…,tₖ> in bank[A₁] × … × bank[Aₖ]:
            yield σ(t₁,…,tₖ)
```

```
L ::= sort(L)    |
      L[N..N]    |
      L + L      |
      [N]        |
      x
N ::= find(L,N)  |
      0
```

[[1,4,0,6] → [1,4]]

inefficient, generating same terms again and again!
better index bank by depth

# Bottom-up enumeration

```
bottom-up (<Σ, N, R, S>, [i → o]):
  bank[A,d] := {} forall A, d
  for d in [0..]:
    forall (A → rhs) in R:
      forall t in new-terms(A→rhs, d, bank):
        if (A = S ∧ t([i]) = [o]):
          return t
        bank[A,d] += t;

new-terms(A → σ(A₁…Aₖ), d, bank):
 if (d = 0 ∧ k = 0) yield σ
 else forall <d₁,…,dₖ> in [0..d-1]ᵏ s.t. max(d₁,…,dₖ) = d-1:
      forall <t₁,…,tₖ> in bank[A₁,d₁] × … × bank[Aₖ,dₖ]:
        yield σ(t₁,…,tₖ)
```

```
L ::= sort(L)    |
      L[N..N]    |
      L + L      |
      [N]        |
      x
N ::= find(L,N)  |
      0
```

[[1,4,0,6] → [1,4]]

# Bottom-up: example

Program bank

d= 0: `x` `0`

d =1: `sort(x)` `x + x` `x[0..0]` `[0]` `find(x,0)`

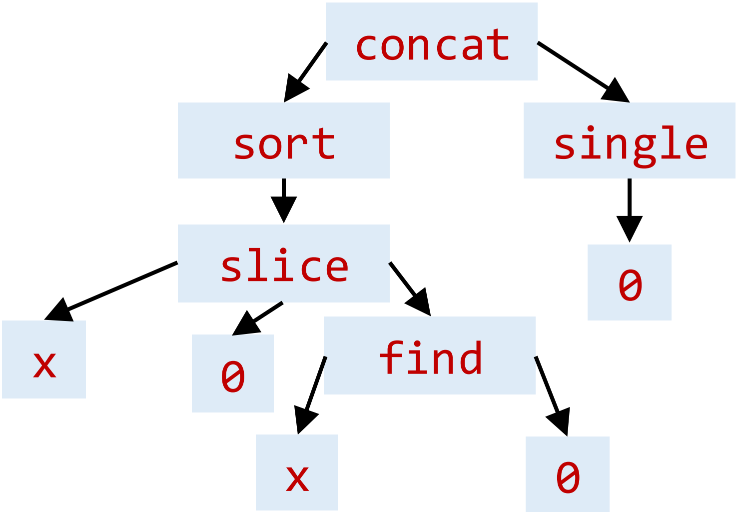
d = 2: `sort(sort(x))` `sort(x[0..0])` `sort(x + x)` `sort([0])` `x + (x + x)` `x + [0]` `sort(x) + x` `x[0..0] + x` `(x + x) + x` `[0] + x` `x + x[0..0]` `x + sort(x)` `x[0..find(x,0)]` ✓

```
L ::= sort(L)    |
      L + L      |
      L[N..N]    |
      [N]        |
      x
N ::= find(L,N)  |
      0
```

[[1,4,0,6] → [1,4]]

# Bottom-up: discussion

- What are some optimizations that come to mind?
- Instead of by depth, we can enumerate by size
  - Why would we want that?




depth = 4, size = 10

programs of size <= 10: 8667
programs of depth <= 4: >1M

  - Which parts of the algo would we need to change?

# Bottom-up vs top-down

- **Top-down**
- **Bottom-up**

Smaller to larger depth

- Has to explore between $3*10^9$ and $10^{23}$ programs to find `sort(x[0..find(x, 0)]) + [0]` (depth 6)

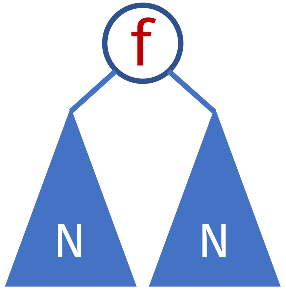
- Candidates are whole but might not be complete
  - Cannot always run on inputs
  - Can always relate to outputs (?)

- Candidates are complete but might not be whole
  - Can always run on inputs
  - Cannot always relate to outputs

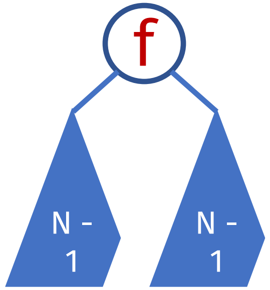
# How to make it scale

**Prune**

- Discard useless subprograms




$m * N^2$




$m * (N - 1)^2$

**Prioritize**

- Explore more promising candidates first

```
P = {[0][N..N] ,
      x[N..N] ,
       ... }
```

dequeue this first