Ling 5801: Lecture Notes 6
Correctness, Complexity, and Generalization

Contents

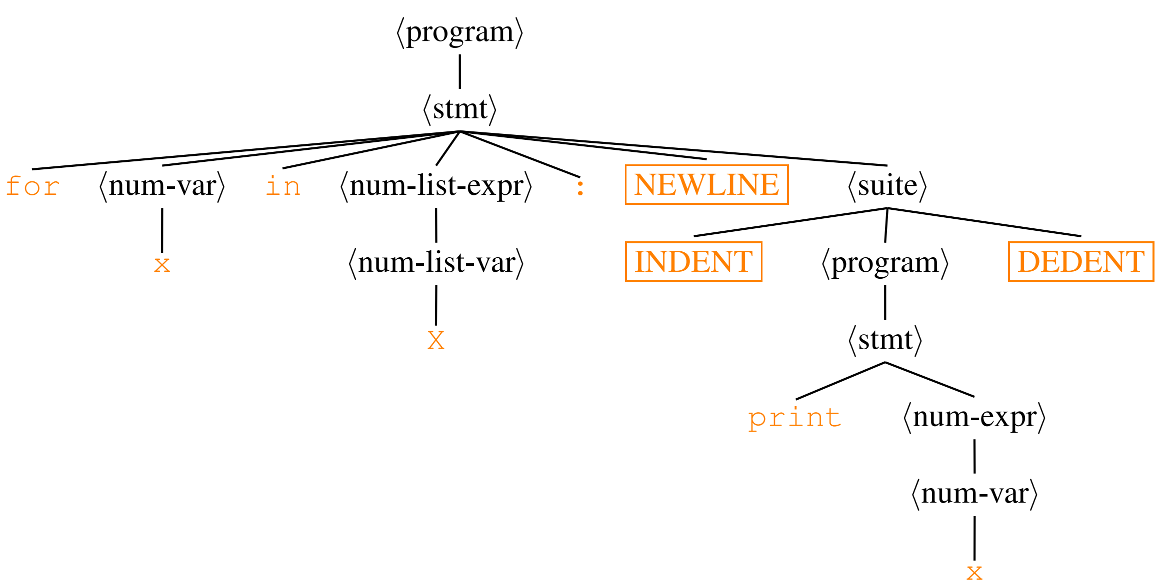
6.1 Operations in an algorithm

The syntax rules used in every program defines a tree.

For example:

```
for x in X :
    print x
```

has the following tree:



In this tree, each *non-unary lexicalized* rule counts as an operation:

- 'non-unary' rules have more than one child
- 'lexicalized' rules contain at least one terminal symbol
  (other than NEWLINE, INDENT, or DEDENT)

(or count first keyword of each rule: '`if`', '`for`', '`=`', '`+`', '`[`', ...)

Each operation takes some number of clock cycles to execute

Loops execute all operations under loop on *each iteration*!

(so time complexity of loops within loops grows exponentially with each loop)

6.2 Complexity: how efficient is a program/algorithm?

Time taken by an algorithm A can be measured in terms of *complexity classes*:

linear :	$A \in \mathcal{O}(n)$
quadratic :	$A \in \mathcal{O}(n^2)$
cubic :	$A \in \mathcal{O}(n^3)$
... :	$A \in \mathcal{O}(g(n))$

Definition of (worst-case) complexity classes:

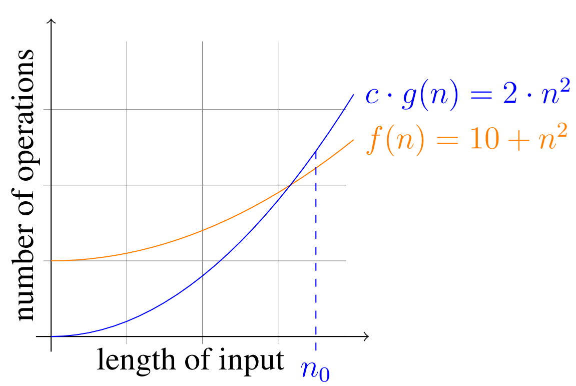
$$A \in \mathcal{O}(g(n)) \text{ if and only if } \overbrace{\exists_{n_0,c}}^{\text{there are some parameters}} \underbrace{\forall_{x_1..x_n}}_{\text{for all inputs}} \overbrace{n < n_0}^{\text{before threshold length}} \vee \underbrace{\tau(A(x_1..x_n)) \leq c \cdot g(n)}_{\text{number of operations is bounded}}$$

where:

- n_0 is a point at which higher-order terms overtake lower-order terms in $g(n)$
- c is a constant time cost for the group of most deeply nested statements
- $x_1..x_n$ is an input sequence of observations of length n
- $\tau(A(x_1..x_n))$ is the time (in number of operations) required to execute A on $x_1..x_n$

In other words, an algorithm A is in class $\mathcal{O}(g(n))$ if there is a length n_0 beyond which all input $x_1..x_n$ takes time within a constant c multiple of $g(n)$.

For example:



What counts as input? Our `FSArec` has input `X` (n is the number of characters defining `X`)

Other terms? if algo is flexible, they count too (separately): q chars defining `S`, `F`, `M`

For loops, complexity (in statements executed) exponential on number of nested loops.

For example, our FSA recognizer:

```
# initialize table of possible states at time step 0 using start states
V = {}
for q in Q:
   V[0,q] = S.get(q,False)

# for each possible state qP in V at time t-1, for each qP,x,q in M, add q
for t in range(1,len(Input)):
   for qP in Q:
      for q in Q:
         V[t,q] = V.get((t,q),False) or (V[t-1,qP] and M.get((qP,Input[t],q),False))
```

requires $A_{\text{FSA}} \in \mathcal{O}(n \cdot q^2)$ because a statement is nested in one loop over `X`, two loops over `Q`

6.3 Correctness: does a program do what it should?

Correctness of an algorithm (abstraction of a program) depends on correctness of statements.

Most statements are straightforward.

But loops are more complex; usually proven by induction:

- define a loop invariant
- base case: demonstrate invariant satisfied at beginning of loop
- induction step: demonstrate invariant satisfied after each iteration if satisfied before
- demonstrate if invariant is satisfied at end, program is correct

For example, using our FSA implementation (prior to final state checking):

```
# initialize table of possible states at time step 0 using start states
V = {}
for q in Q:
   V[0,q] = S.get(q,False)

# for each possible state qP in V at time t-1, for each qP,x,q in M, add q
for t in range(1,len(Input)):
   for qP in Q:
      for q in Q:
         V[t,q] = V.get((t,q),False) or (V[t-1,qP] and M.get((qP,Input[t],q),False))
```

We can prove correctness of the inner loop over `q` in the last nesting group, given `t` and `qP`:

- loop invariant:

  After each iteration, `V` shows states at or before `q` reachable from states at or before `qP` on input up to time `t`.

- base case:

  Before loop begins, `V` shows states reachable from sources before `qP` on input up to time `t`.

- induction step:

  After each iteration, `V` shows states at or before `q` reachable from states at or before `qP` on input up to time `t` if:

  1. `V` shows states before `q` reachable from states at or before `qP` at time `t` before iter,
  2. `V` shows `qP` was reachable on input up to `t-1`, and
  3. `M` contains a transition from `qP` to `q` on the input at `t`.

- correctness:

  After loop ends, because it looped over all states, `V` shows all reachable states from `qP` on input up to time `t`.

We can now prove correctness of the next inner loop over `qP`, given `t`:

- loop invariant:

  After each iteration, `V` shows states reachable from states at or before `qP` on input up to time `t`.

- base case:

  Before loop begins, `V` shows states reachable on input up to the previous time `t-1`.

- induction step:

  After each iteration, `V` shows states reachable from states at or before `qP` on input up to time `t` if

  1. `V` shows states reachable from states before `qP` on input up to time `t`, and
  2. the inner loop leaves `V` showing reachable states from `qP` on input up to time `t`.

- correctness:

  After loop ends, because it looped over all states, `V` shows reachable states at or before time `t`.

We can now prove correctness of the outer loop over `t`:

- loop invariant:

  After each iteration, `V` shows reachable states at time `t`.

- base case:

  Before loop begins, `V` contains only initial states.

- induction step:

  After each iteration, `V` shows states reachable on input up to `t` if

  1. `V` shows states reachable on input up to time `t-1`, and
  2. the inner two loops leave `V` showing reachable states on input at time `t`.

- correctness:

  After loop ends, `V` shows reachable states at end of input.

Then do same for other loops, proving correctness of assumptions in induction step.