©Nancy Lynch, 2000

Notes on Computational Processes, Installment 1

Nancy Lynch

February 28, 2000

1 Overview

Introduce some of the basic math ideas behind *computational processes*.

I'm using the term in a very general way, to denote any kind of system that begins in a special "starting state", and progresses from state to state in discrete steps, according to specified rules.

These include:

- Sequential algorithms = abstract versions of sequential programs.
  E.g., programs that sort data, or compute a numerical solution to a complicated equation.

- Distributed algorithms = abstract versions of distributed programs.
  E.g., programs that implement an application like a multi-branch banking system or an airline reservation system.

- Abstract descriptions of complicated computer systems like operating systems, computer architectures, or communication systems.

- Descriptions of games, with rules that constrain the moves.

- ...

Many computational processes are *nondeterministic*, which just means that the next state isn't always determined by the previous state:
There might be explicit choice points, like the next move in a game. Or just different possible orders in which things might happen, e.g., different orders in which two messages might arrive at a destination.

The idea of a computational process is very general.
But it turns out that some simple math can go a long way in helping us to understand and reason about them.

Today:

Introduce a basic math model for a computational process, called a *state machine*.
Also give basic definitions that will let us talk about how such a process behaves.

We'll show several examples, of several kinds. Based on sequential and parallel algorithms, objects, system descriptions, games...

Try to understand informally how these processes behave.
Also, since they are based on a real math model, it also makes sense to try to *prove* some things about how they behave.

Next week:
Continue with informal examples.

Get more systematic about how you do proofs about computational processes.
The most important idea is that of an "invariant", which describes a property that is always true of the state of a process, no matter how it behaves.
You will see how invariants can be used to prove basic correctness properties for algorithms.
Give systematic method for proving invariants, based on ordinary math induction.

We'll then look briefly at other kinds of properties one might want to prove about processes, e.g., that they terminate, or that they finish within some fixed amount of time.

Finally:
We'll take a look at some ideas that allow us to impose some structure on processes, specifically, to:

- Combine small processes to build bigger processes.

  For instance, build a distributed banking system out of pieces that run on different machines.

  Or, build a powerful application out of a bunch of simpler services. E.g., interactive computer game out of a numerical calculation process, a graphical user interface process and a database process.

- Consider systems at different levels of abstraction.

  For instance, view a parallel machine at the level of the processor and memory,
  or at the level of the application programming language, or at the level of the service provided by an application (e.g., a chess-playing program).

The focus here will be on providing the basic math defs and a little basic theory, and showing you some examples to see how these can be used.

These ideas should be useful to you when you work on (or take courses on) computer systems.
E.g., 6170 SWE includes a lot of object-oriented programming.
State machines can be used to describe objects, how some objects can be viewed at different levels of abstraction, and how objects can be combined.

2 Video interlude

Let's start with a movie clip showing a nice example of a computational process.

Die Hard

3 State machines

3.1 Basic definitions

Mathematically speaking, a state machine is just a binary relation on states, where the pairs in the relation correspond to the allowed steps of the machine.
In addition, a state machine has some states that are designated as start states—the ones in which it is allowed to begin executing.

Notation:
Q, the set of states (must be nonempty)
$Q_0 \subseteq Q$, the set of start states (must be nonempty)
$\delta \subseteq Q \times Q$, the set of allowed transitions between states.

Although this is really nothing more than a digraph with start states (nodes), we use different notation.
Instead of (A, R) as we used for relations, or $G = (V, E)$ as we used for graphs, we use (Q, Q_0, δ).
Reflects different customs in different areas.

Often write $q \to q'$ for the statement that $(q, q') \in \delta$.

This is similar to definition of finite-state machine that you might have seen in a hardware class (you will see this if you take 6045).
But these need not be finite-state.
Also, I haven't given them (yet) any notion of input and output.

Example:
State machine to model a bounded counter, which counts from 0 to 100

$Q = \{0, 1, 2, \ldots, 99, overflow\}$.
Special value for overflow, say.

$Q_0 = \{0\}$
$\delta = \{(0, 1), (1, 2), (2, 3), \ldots, (99, \mathit{overflow}), (\mathit{overflow}, \mathit{overflow})\}$.
A self-loop.

Draw as digraph.

Example:
Unbounded counter–similar, but infinite state set, infinite digraph.

Example:
Die Hard:
$Q = \{(b,l) | 0 \leq b \leq 5, 0 \leq l \leq 3\}$
(big, little); reals, not necessarily integers
$Q_0 = \{(0,0)\}$, both start empty.

δ has several kinds of steps:
1. Fill little jar: $(b,l) \rightarrow (b,3)$
2. Fill big jar: $(b,l) \rightarrow (5,l)$
3. Empty little jar: $(b,l) \rightarrow (b,0)$
4. Empty big jar: $(b,l) \rightarrow (0,l)$
5. Pour from little jar into big jar: $(b,l) \rightarrow (\min(b+l,5), \max(0,b+l-5))$.

The reason for these expressions:
The amount of water that winds up in the big jar is all the water than was there (b) plus as much of the water in the little jar as will fit in the big jar.
That's min(b+l,5).
–Can see in cases: if $b+l \leq 5$, get $b+l$, otherwise tops out at 5.
The second term represents what's left over.

6. Pour from big jar into little jar: $(b,l) \rightarrow (\max(b+l,3), \min(0,b+l-3))$.

This state machine is an example of a *nondeterministic* state machine:

Definition:
A state machine (Q, Q_0, δ) is *deterministic* if $|Q_0| = 1$ and for every $q \in Q$, $\exists$ at most one q' such that $q \rightarrow q'$.

Otherwise, *nondeterministic*.
Thus, the Die Hard state machine is nondeterministic.
In fact, it has as many as 6 different transitions out of some states (e.g., $(1,1)$).

The two counter machines are deterministic.

3.2 Executions of state machines

Can model the possible behaviors of the computational process by *executions* of the state machine.

Def: An *execution* is a (possibly infinite) sequence $q_0, q_1, \ldots$ such that:
$q_0 \in Q_0$, and
$\forall i \geq 0 (q_i, q_{i+1}) \in \delta$.

Like a path in the digraph, except:
–List the vertices rather than the edges.
–Must begin with a start state.

One thing we often want to understand about the behavior of a computational process is what kinds of states can be reached in its executions.

Def: A state is *reachable* if it's the final state in some finite-length execution.

Example:
In Die Hard, can reach state (4,0), by execution:
(0,0),
fill big jar, (5,0),
pour big into little, (2,3),
empty little, (2,0),
pour big into little, (0,2),
fill big jar, (5,2),
put big into little, (4,3),
empty little, (4,0).

So, state (4,0) is reachable.

Example: Die Harder

Look at the same problem, but with 9 and 3 gallon jars instead of 5 and 3.
Still want 4 (in the big jar).

δ:
1. Fill little jar: $(b, l) \rightarrow (b, 3)$
2. Fill big jar: $(b, l) \rightarrow (9, l)$
3. Empty little jar: $(b, l) \rightarrow (b, 0)$
4. Empty big jar: $(b, l) \rightarrow (0, l)$
5. Pour from little jar into big jar: $(b, l) \rightarrow (\min(b + l, 9), \max(0, b + l - 9))$.
6. Pour from big jar into little jar: $(b, l) \rightarrow (\max(b + l, 3), \min(0, b + l - 3))$.

Now impossible.
Why?
How to prove no reachable state has 4?

Some reachable states: (0,0), (0,3), (3,0), (3,3), (6,0),...

Prove:
At the end of any finite execution, the big jar doesn't contain 4.

Try induction?
Try $P(n)$: At the end of any n-step execution, the big jar doesn't contain 4.

Base: 0 steps. True.
But get stuck at inductive step–e.g., if previous state is (2,2)...

So try strengthening inductive hypothesis.

Define $P(n)$: At the end of any n-step execution, the contents of each jar are a multiple of 3 gallons.

Prove: $\forall n \geq 0(P(n))$

1. (Base) $P(0)$ — Start state is $(0,0)$, 0 is a multiple of 3.
2. (Inductive step) $\forall n \geq 0(P(n) \Rightarrow P(n+1))$
    1. Fix $n \geq 0$.
    2. Assume $P(n)$, that is, after n steps, both jars have multiples of 3.
    3. $P(n+1)$, that is, after $n+1$ steps, both jars have multiples of 3. — ???
    4. QED — Implication, UG
3. QED — Induction

The argument for the ??? involves fixing a particular $n+1$-step execution $q_0, q_1, \ldots, q_{n+1}$. The first part, except for the final state is an n-step execution. So use the inductive hypothesis to say that its final state, q_n, consists of two multiples of 3.
Then consider what happens at the final step.
6 cases.
Each one preserves the property.

3. $P(n+1)$, that is, after $n+1$ steps, both jars have multiples of 3.
    1. Fix $q_0, q_1, \ldots, q_{n+1}$, an execution with $n+1$ steps.
    2. In q_n, both jars have multiples of 3. — Inductive hypothesis.
    3. In q_{n+1}, both jars have multiples of 3.
        1. If the last move is to fill the little jar, then ...
        2. If the last move is to fill the big jar, then...
        3. If the last move is to empty the little jar...
        4. If the last move is to empty the big jar...
        5. Pour from little to big...
        6. Pour from big to little...
        7. QED — Cases
    4. QED — UG

Each of the six cases could be done in detail, but they're all easy.

Inductive proofs of properties of reachable states are very common in reasoning about program and system correctness.

4 A two-dimensional puzzle

Two-dimensional board, with integer grid.
Piece starts out at $(0,0)$, allowed to move in any of these 4 ways:
(+2,-1) Right 2, down 1
(-2,+1) Left 2, up 1
(+1,+3)
(-1,-3)

(draw in 4 points)

Q: Can the piece ever reach $(1, 1)$?

Draw in some points.
Seems not—how to prove?

Can represent as state machine, with:
$Q = \{(x, y) | x, y \in Z\}$
$Q_0 = \{(0, 0)\}$
δ consists of all $(x, y) \to (x', y')$, where (x', y') is one of the following:
$(x + 2, y - 1)$
$(x - 2, y + 1)$
$(x + 1, y + 3)$
$(x - 1, y - 3)$

Try to use induction to prove that this machine can't reach $(1, 1)$.
But can't let $P(n) =$ "the state after n steps isn't $(1, 1)$".
E.g., $(-1, 2) \to (1, 1)$.

So strengthen.
How?
Try to observe something about the pattern.
Draw more dots.
On a bunch of parallel lines:
$x + 2y = 0$
$x + 2y = 7$
$x + 2y = -7$
Etc.

So try: $P(n)$: The state after n steps satisfies the condition that $x + 2y$ is a multiple of 7.

Notation: $x + 2y \cong 0 mod 7$.
LHS and RHS have the same remainder on division by 7.

Prove: $\forall n \geq 0 (P(n))$
1. (Base) $P(0)$ Start state is $(0, 0)$, $0 + 2 \times 0$ is a multiple of 7.
2. (Inductive step) $\forall n \geq 0 (P(n) \Rightarrow P(n + 1))$
    1. Fix $n \geq 0$.
    2. Assume $P(n)$, that is, after n steps, $x + 2y$ is a multiple of 7.
    3. $P(n + 1)$, that is, after $n + 1$ steps, $x + 2y$ is a multiple of 7.
        1. Fix $q_0, q_1, \ldots, q_{n+1}$, an execution with $n + 1$ steps.
        2. In q_n, $x + 2y$ is a multiple of 7. Inductive hypothesis.
        3. In q_{n+1}, $x + 2y$ is a multiple of 7.
            1. If the last move is $(+2, -1)$ then in q_{n+1}, $x + 2y$ is a multiple of 7.
            2. If the last move is $(-2, +1)$ then in q_{n+1}, $x + 2y$ is a multiple of 7.
            3. If the last move is $(+1, +3)$ then in q_{n+1}, $x + 2y$ is a multiple of 7.
            4. If the last move is $(-1, -3)$ then in q_{n+1}, $x + 2y$ is a multiple of 7.

5. QED — Cases

4. QED — Implication, UG

3. QED — Induction

Show two of the four cases; the others are similar.

1. If the last move is $(+2, -1)$ then in q_{n+1}, $x + 2y$ is a multiple of 7.
   1. Assume the last move is $(+2, -1)$.
   2. Let x and y denote the values in state q_n, x' and y' the values in state q_{n+1}.
   3. $x' = x + 2$, $y' = y - 1$. — Definition of the move
   4. $x + 2y \cong 0 mod 7$ — Inductive hypothesis
   5. $x' + 2y' = (x + 2) + 2(y - 1) = x + 2y \cong 0 mod 7$ — Algebra
   6. In q_{n+1}, $x + 2y$ is a multiple of 7.
   7. QED — Implication

3. If the last move is $(+1, +3)$ then in q_{n+1}, $x + 2y$ is a multiple of 7.
   1. Assume the last move is $(+1, +3)$.
   2. Let x and y denote the values in state q_n, x' and y' the values in state q_{n+1}.
   3. $x' = x + 1$, $y' = y + 3$. — Definition of the move
   4. $x + 2y \cong 0 mod 7$ — Inductive hypothesis
   5. $x' + 2y' = (x + 1) + 2(y + 3) = x + 2y + 7 \cong 0 mod 7$ — Algebra
   6. In q_{n+1}, $x + 2y$ is a multiple of 7.
   7. QED — Implication

$(1, 1)$ not reachable.
Because $x + 2y = 3$, which is not congruent to 0 mod 7.

5 Sequential algorithm examples

Now move from puzzles to sequential algorithms.
Describe a few typical algorithms involving the data types we've studied so far.

Prove a few simple properties.

5.1 GCD

Given 2 positive natural numbers, $a, b \in N^+$, find GCD (define).

Algorithm: (Not especially efficient—a more efficient version of this appears as the Euclidean GCD algorithm, p. 129 of Rosen.)

Keep working values x and y, initialized at a and b.
Then:
If $x = y$, that's the answer.

If $x > y$, then let $r = x - y$.

Claim: GCD(x,y) = GCD(y,r).
Because a positive integer divides 2 positive numbers if and only if it divides their sum, if and only if it divides their difference.

So, now the problem reduces recursively to finding GCD(y,r).

If $y > x$, then let $r = y - x$, and problem reduces recursively to finding GCD(x,r).

Example:
Find $GCD(112, 84) = ?$
Step 1: $112 - 84 = 28$, reduce to $(28, 84)$.
Step 2: $84 - 28 = 56$, reduce to $(28, 56)$.
Step 3: $56 - 28 = 28$, reduce to $(28, 28)$.
Now done, answer is 28.

As a state machine:
$Q = \{(a, b, x, y) | a, b, x, y, \in N^+\}$
$Q_0 = \{(a, b, x, y) | x = a \text{ and } y = b\}$
δ: $(a, b, x, y) \rightarrow (a', b', x', y')$, where:
$a' = a$
$b' = b$
Either $x > y$ and $y' = y, x' = x - y$,
or $y > x$ and $x' = x, y' = y - x$

Prove that this correctly computes $GCD(a, b)$.
It is convenient to break this claim down into two parts.

First, if we ever finish, then we have the right answer:
1. If we ever reach a point where $x = y$, then in fact $x = GCD(a, b)$.

Second, we actually finish:
2. Eventually, $x = y$.

The meaning of 2. should not be clear to you yet.
We'll come back to this in a minute.
First let's prove 1.

We want to prove: In any reachable state, if $x = y$ then $x = GCD(a, b)$.

To prove this by induction, we have to strengthen the statement to say something about intermediate states.

Define $P(n)$:
After any n-step execution, $GCD(x, y) = GCD(a, b)$.

This says we are correctly reducing the problem to equivalent problems.
The property we want is a corollary of $\forall n(P(n))$:

Prove: $\forall q$, reachable state ($x = y$ in q implies $x = GCD(a, b)$ in q).
1. Fix reachable state q.
2. Assume $x = y$ in q
3. $GCD(x, x) = GCD(a, b)$ in q. — By the theorem just above
4. $GCD(x, x) = x$ in q — Number theory
5. $x = GCD(a, b)$ in q. — Algebra
6. QED — Implication, UG

Prove P by induction:

Prove: $\forall n(P(n))$
1. (Base) $P(0)$ — After 0 steps, $x = a$ and $y = b$.
2. (Inductive step) $\forall n \geq 0(P(n) \Rightarrow P(n+1))$
    1. Fix $n \geq 0$.
    2. Assume $P(n)$, that is, after n steps, $GCD(x, y) = GCD(a, b)$.
    3. $P(n+1)$, that is, after $n+1$ steps, $GCD(x, y) = GCD(a, b)$.
       ???
    4. QED — Implication, UG
3. QED — Induction

To show ???, we can consider cases based on whether x is less than or greater than y. We don't have to consider the case where they're equal, because nothing (no transition) happens in this case.
The two cases are analogous, so we just show one:

3. $P(n+1)$, that is, after $n+1$ steps, $GCD(x, y) = GCD(a, b)$.
    1. Fix $q_0, q_1, \ldots, q_{n+1}$, an execution with $n+1$ steps.
    2. In q_n, $GCD(x, y) = GCD(a, b)$. — Inductive hypothesis.
    3. In q_{n+1}, $GCD(x, y) = GCD(a, b)$.
        1. If $x > y$ in q_n then in q_{n+1}, $GCD(x, y) = GCD(a, b)$.
            1. Assume $x > y$ in q_n.
            2. Let x and y denote the values in state q_n, x' and y' the values in state q_{n+1}.
            3. $x' = x - y$, $y' = y$ — Definition of the step
            4. $GCD(x, y) = GCD(a, b)$. — Inductive hypothesis
            5. $GCD(x', y') = GCD(x - y, y)$ — Plug in
            6. $GCD(x - y, y) = GCD(x, y)$ — Number theory fact
            7. $GCD(x', y') = GCD(a, b)$ — Algebra
            8. In q_{n+1}, $GCD(x, y) = GCD(a, b)$
            9. QED — Implication
        2. If $x < y$ in q_n then in q_{n+1}, $GCD(x, y) = GCD(a, b)$.
           Analogous to the previous case
        3. QED — Cases

So, this shows that the answer, if it's ever produced, is correct.

Now turn to the second property, that the answer is actually produced at some point, i.e., eventually $x = y$.

It is not obvious what this means.
To make sense of this, we need to make explicit the assumption that says that steps keep happening—an execution doesn't just stop for no reason.
The only situation in which the execution is allowed to stop is when it reaches a state where no steps are "enabled" (defined starting from that state).

E.g.: In Euclidean GCD, the execution can stop when $x = y$, because no steps are defined.
But we want to assume that the execution continues as long as $x \neq y$.

Leads to a definition:
An execution is *live* provided it satisfies:
If it's finite, then no transitions are defined from the last state.

Unwinding the definition using propositional logic, we see that an execution is live in two situations:
It's infinite.
It's finite, but no step is enabled in the last state.

So, the way to state the "eventuality" requirement of the GCD algorithm is:

Prove: Any live execution of GCD contains some state in which $x = y$.

How to prove this?

One way is to notice that something about the size of the problem being considered is decreasing at every step.
What? E.g., the sum $x + y$.

This sum can't decrease forever, because it would eventually have to reach its minimum possible value $1 + 1$.
But if it ever reaches this value, the goal is reached.

Also, if it hasn't reached the goal, another step is enabled.

One way to express all this in a proof by contradiction, assuming that the execution goes on forever and showing that we must eventually reach a nonsense state.

Prove: Any live execution of GCD contains some state in which $x = y$.
1. Assume not, that is, $\exists$ a live execution in which we never have $x = y$.
2. Fix $q_0, q_1, q_2, \ldots$, a live execution in which we never have $x = y$.
EI
3. $q_0, q_1, q_2, \ldots$ is either finite with $x = y$ in the final state, or is infinite.
Definition of "live execution"
4. $q_0, q_1, q_2, \ldots$ is infinite — By 3 and the fact that we don't have $x = y$.
5. $\exists$ a smallest value of $x + y$ that occurs in the states of the execution
Well-ordering

6. Fix q_k to be a state in which this smallest value is reached.
EI
7. The value of $x + y$ in q_{k+1} is smaller than the value of $x + y$ in q_k.
By the way the steps are defined
8. q_k does not have the smallest value of $x + y$ in the execution.
By 7
9. QED Contradiction, 6 and 8.

5.2 Reflexive, transitive closure of relations

In the puzzles and algorithm so far, all data have been numbers.
Computational processes can involve other data types too.
Including high level, "abstract" data types like relations.

Example:
Consider a simple algorithm to compute the reflexive, transitive closure of a binary relation R on a finite set.
Recall that this is equal to $I \cup R \cup R^2 \cup R^3 \cup \ldots R^n$, where n is the number of elements in the set.
Here, I is the identity relation, which just relates (x, x) for every x.
Also write as $I + R + R^2 + R^3 + \ldots$ (infinite sum), but nothing new gets added after the R^n term.

One way to compute this is to first calculate $I + R$.
This relates x to y provides that $x = y$ or $(x, y) \in R$, that is, if x is related to y by 0 or 1 "steps".

Now square the result: gives $I + R + R^2$, which represents all paths of length at most 2.

If we square again?
All paths of length at most 4.

Each time we square, we double the length of the path.
So it doesn't take very long for this to stabilize.
Only $\lceil \log n \rceil$ steps.

Example:

R is the relation on set $S = \{a, b, c, d, e\}$ consisting of the pairs (a, b), (b, c), (c, d), (d, e).
Initially, T contains R augmented with the pairs (a, a), (b, b), (c, c), (d, d), (e, e).
First step squares T, which adds in the two-step paths (a, c), (b, d) and (c, e).
Next step squares T again, which yields the 3-step and 4-step paths (a, d), (b, e), (a, e).

As a state machine:
Fix an underlying finite set S, n elements.

$Q = \{(R,T)\}$, both binary relations
$Q_0 = \{(R,T) | T = R \cup I\}$
δ: Steps of the form $(R,T) \rightarrow (R',T')$, where $R' = R$, $T' = T^2$.

Thus, the state machine keeps squaring forever.
The only executions are infinite.
However, after a while, the state stops changing (self-loops).

What might we prove about this?

After $\lceil \log n \rceil$ steps, T is the reflexive, transitive closure of R.

Follows from more general:
For any $k \geq 0$, after k steps, $T = I \cup R \cup \ldots \cup R^{2^k}$.

Which can be proved by induction on k, as usual.
Relational algebra used in the inductive step.

Also, notice that all the live executions are infinite—another step can always be done.
In any live execution, eventually get $T =$ the reflexive, transitive closure.

Note that I described the previous algorithm at a high "level of abstraction", where the data is described as being binary relations.
I didn't say anything about how these binary relations are implemented in terms of more primitive data types like integers, strings, Booleans, etc.
(E.g., I could have talked about Boolean matrices, or adjacency lists.)

That was on purpose.
It is very often a good idea to think about algorithms at a high level of abstraction.
Because some things you want to say about the algorithms are independent of the representation and the implementation details.

5.3 Graph coloring

Consider (undirected) graph $G = (V,E)$ in which no node has more than d neighbors.
(degree $\leq d$)

It is possible to color the vertices of G with at most $d+1$ colors, in such a way that no two neighboring nodes are colored the same color.

Show this by producing an algorithm that actually does the coloring.
The idea:
Start out with all the vertices uncolored.
Repeatedly choose a vertex and color it, assigning it any color that has not already been used for a neighbor.

Represented as a state machine:
Assuming $G = (V,E)$ is fixed.

Q = the set of functions from V to $colors \cup \{null\}$.
(Null a special value meaning "not yet colored".)

Q_0 = the function mapping all elements of V to the value $null$.

δ: In words: The mapping in the new state is the same as that in the old state, except that one node that was previously uncolored (value null) gets assigned a color different from any that has already been assigned to any of its neighbors.

What to prove about this?

1. No 2 neighbors are ever colored the same color.
2. Eventually finish coloring all the vertices.

For 1:

Can prove by induction on the number of steps.
Clearly true in the initial state, because nothing is colored.
Inductive step:
If no two neighbors are colored the same color, and we color one more node as allowed by δ, then again no two neighbors are colored the same color.
The new step colors a node a different color from any of its neighbors, so doesn't introduce any counterexamples.
Doesn't change any of the previous colorings, so they still yield no counterexamples.

For 2:

Remember, only consider live executions.
Either infinite, or finite and nothing enabled in final state.
Look for something that decreases at every step:
Number of uncolored nodes.

Starts at n, and any step reduces by 1.
Number can't decrease forever.
If haven't colored all the nodes, another step is always enabled.
Try a direct proof this time:

Prove: In any live execution, there exists some state in which all the nodes are colored.
1. Fix $q_0, q_1, q_2, \ldots$, a live execution.
2. $q_0, q_1, q_2, \ldots$ is either finite with no step enabled in the final state, or is infinite.
Definition of "live execution".
3. $q_0, q_1, q_2, \ldots$ isn't infinite — Number of uncolored nodes decrease at every step, can't go below 0.
4. $q_0, q_1, q_2, \ldots$ is finite with no step enabled in the final state.
By 2. and 3.
5. All nodes are colored in the final state.
    1. If not all nodes are colored in the final state, then some step is enabled.

	Any uncolored node can be colored; there are enough colors.
2. QED	Contradiction (5.1. and 4.)
6. QED	EG, UG

EG: We found a state with the property we want, so one exists.
UG: Showed the property for a generic execution, so holds for all executions.

6 Programming-language-style notation for sequential algorithms

6.1 Overview

Sequential algorithms are usually presented (in textbooks, papers), not by describing their states and transitions, but using a programming-language-style notation.

Not programs in a real programming language like C or Java, but "pseudo-programs" in a simplified programming language.

Rosen, Appendix B, gives a typical such program notation.

The basic constructs are assignment statements, conditionals, loops.
Also statement grouping constructs like blocks and procedures.
Procedures may have side-effects.
Procedures can call other procedures, possibly recursively.

Style of these programs is like PASCAL, Algol, C.

These programs denote step-by-step computational processes, i.e., state machines.

6.2 Reflesive, transitive closure

Notation:
Assume positive integer n and set S are fixed.
Let I be a constant denoting the identity relation on S.

procedure reflexiveTransitiveClosure(R: binary relation on S)
declare T: binary relation on S
$T := R \cup I$
while true do $T := T^2$

This program is written in an *iterative* style, executing a loop.
Here, the loop executes forever, so the procedure never returns.
To modify it, put a stopping condition in the loop.
And some indication of where the answer is.

procedure transitiveClosure(R: binary relation on S)
declare T: binary relation on S
$T := R \cup I$

for $i := 1$ to $\lceil \log n \rceil$ do $T := T^2$
return T

This program can be translated into a state machine, states and transitions.
What's in the state?

T, R,...

A little more info, for bookkeeping:
An indication of what the processing is up to—the beginning, at the loop, at the end.
And if at the loop, which pass.

E.g., we can include: – A *program counter* saying where we're up to, in the statements of the program, and – A *loop index counter*, keeping track of which pass of the loop we're up to, when we're doing the loop.

With this extra info, we can describe the allowable steps as state transitions $(T, R, pc, i) \to (T', R', pc', i')$.
These can be of several different kinds, e.g.:

- Initialization of T:
  $pc =$ "$begin''$, $T' = R \cup I$, $R' = R$, $pc' =$ "$at - loop''$, and $i' = 1$.
- The ith pass through the loop:
  $pc =$ "$at - loop''$, $i \leq \lceil \log n \rceil$, $T' = T^2$, $R' = R$, $pc' =$ "$at - loop''$, and $i' = i + 1$.
- Moving to after the loop:
  $pc =$ "$at - loop''$, $i > \lceil \log n \rceil$, $T' = T$, $R' = R$, $pc' =$ "$at - end''$, and $i' = 0$.

Even though the algorithm is represented a little differently, we would still like to prove the same kinds of properties as before.

Properties would come out looking slightly different.
E.g., might want to prove:
If $pc = at - loop$ then $T = I \cup R \cup R^2 \cup \cdots \cup R^{2^{i-1}}$..

Typical in state machine representing programs to have properties include mention of program counters and loop index variables.
Say things like "If control of processing is at to a particular place in the program code, then predicate P (of the program variables, possible bookkeeping variables) is true.

In fact, sometimes when properties are of this form, people *annotate* their programs, writing down the properties at the appropriate points in the code.

E.g., can write the property:
If $pc = at - loop$ then $T = I \cup R \cup R^2 \cup \cdots \cup R^{2^{i-1}}$..

by just writing the statement $T = R^{2^{i-1}}$ someplace in the code, corresponding to the beginning of the execution of the body of the loop.

6.3 GCD

procedure $GCD(a, b \in N^+)$
declare $x, y \in N^+$
$x := a$
$y := b$
while $x \neq y$ do
  if $x > y$ then $x := x - y$
  else $y := y - x$
return x

Again, can transform this into (view this as) a state machine, where the state consists of a, b, x, y plus bookkeeping information, here, a program counter.
(No loop index here because we don't need to count the number of times through this kind of loop.)

Can prove the "right answer" and termination properties for this state machine.

6.4 Some complications

Warning:
The translations from pseudo-code to state machine were very simple for these two examples.
They get more complicated when the program includes procedure calls.
For instance, consider GCD rewritten using (recursive) procedure calls:

procedure $GCD(a, b \in N^+)$
declare $ans \in N^+$
if $a = b$ then $ans := a$
else
  if $a > b$ then $ans := GCD(a - b, b)$
  else $ans := GCD(a, b - a)$
return ans

This recursive programming style is typical of languages like LISP, Scheme.

Example execution: $GCD(112, 84)$

Starts with $a = 112$, $b = 84$.
Test says $a > b$, so call $GCD(28, 84)$.
Remember that when this returns, you want to set your ans to that value and return it to the caller.

Now we start over, with a new a and b: This a gets set to 28, this b to 84.
Now test says $b > a$, so call $GCD(28, 56)$

Another a, b, a set to 28, b to 56.
Calls $GCD(28, 28)$.

This one sets its $ans := 28$, returns that value to its caller.
That caller puts the result in ITS ans, returns to ITS caller,

Etc., top-level procedure returns value of its ans to the original caller.

How to translate this into a state machine?
Why would you want to?

In order for a recursive program to execute, it must be translated into something machine-like.
That's what a compiler does.

Anyway, we can't construct a state machine as we did before, letting the state consist of just the program variables plus some bookkeeping info.
Why?

Nested calls of the recursive procedure are actually active at the same time.
They use their own copies of arguments, program variables, etc.
E.g., in the example above, we use a separate a, b and ans for each call.
A state machine describing how this algorithm actually executes would maintain separate copies of these variables.
Arranged in a "stack", where we "push" variables on the stack when we make the calls and "pop" them from the stack when we return.

For proving properties of such a recursive algorithm, you could use the techniques we've been using so far:
– Induction on the number of steps in the state machine.
– Defining a measure that decreases with each step.
But this can get pretty complicated, because of the stack manipulations.
Not usually done—such state-machine-style proofs are usually done for state machines that are derived from "iterative" (loop-style) programs.

6.5 Recursive programs

Another way of reasoning about (purely functional) recursive programs is to treat them as recursive function definitions, and use induction on some measure of the arguments.

Example: GCD

Can write like a function definition, as follows:
$GCD(x, y) = x$ if $x = y$, $GCD(x - y, y)$ if $x > y$, and
$GCD(x, y - x)$ if $x < y$.

Can prove by induction on $x + y$ that $GCD(x, y)$ is defined and that it equals the correct GCD value.