**Solutions for Chapter 2**

[Solutions for Section 2.2](http://infolab.stanford.edu/~ullman/ialcsols/sol2.html#sol22)

[Solutions for Section 2.3](http://infolab.stanford.edu/~ullman/ialcsols/sol2.html#sol23)

[Solutions for Section 2.4](http://infolab.stanford.edu/~ullman/ialcsols/sol2.html#sol24)

[Solutions for Section 2.5](http://infolab.stanford.edu/~ullman/ialcsols/sol2.html#sol25)

Revised 9/6/01.

**Solutions for Section 2.2**

**Exercise 2.2.1(a)**

States correspond to the eight combinations of switch positions, and also must indicate whether the previous roll came out at *D*, i.e., whether the previous input was accepted. Let 0 represent a position to the left (as in the diagram) and 1 a position to the right. Each state can be represented by a sequence of three 0's or 1's, representing the directions of the three switches, in order from left to right. We follow these three bits by either *a*indicating it is an accepting state or *r*, indicating rejection. Of the 16 possible states, it turns out that only 13 are accessible from the initial state, 000r. Here is the transition table:

|  |  |  |
| --- | --- | --- |
|  | **A** | **B** |
| ->000r | 100r | 011r |
| \*000a | 100r | 011r |
| \*001a | 101r | 000a |
| 010r | 110r | 001a |
| \*010a | 110r | 001a |
| 011r | 111r | 010a |
| 100r | 010r | 111r |
| \*100a | 010r | 111r |
| 101r | 011r | 100a |
| \*101a | 011r | 100a |
| 110r | 000a | 101a |
| \*110a | 000a | 101a |
| 111r | 001a | 110a |

**Exercise 2.2.2**

The statement to be proved is *δ-hat(q,xy) = δ-hat(δ-hat(q,x),y)*, and we proceed by induction on the length of *y*.

Basis: If *y = ε*, then the statement is *δ-hat(q,x) = δ-hat(δ-hat(q,x),ε)*. This statement follows from the basis in the definition of *δ-hat*. Note that in applying this definition, we must treat *δ-hat(q,x)* as if it were just a state, say*p*. Then, the statement to be proved is *p = δ-hat(p,ε)*, which is easy to recognize as the basis in the definition of *δ-hat*.

Induction: Assume the statement for strings shorter than *y*, and break *y = za*, where *a* is the last symbol of *y*. The steps converting *δ-hat(δ-hat(q,x),y)* to *δ-hat(q,xy)* are summarized in the following table:

|  |  |
| --- | --- |
| **Expression** | **Reason** |
| *δ-hat(δ-hat(q,x),y)* | Start |
| *δ-hat(δ-hat(q,x),za)* | *y=za* by assumption |
| *δ(δ-hat(δ-hat(q,x),z),a)* | Definition of *δ-hat*, treating *δ-hat(q,x)* as a state |
| *δ(δ-hat(q,xz),a)* | Inductive hypothesis |
| *δ-hat(q,xza)* | Definition of *δ-hat* |
| *δ-hat(q,xy)* | *y=za* |

**Exercise 2.2.4(a)**

The intuitive meanings of states *A, B*, and *C* are that the string seen so far ends in 0, 1, or at least 2 zeros.

|  |  |  |
| --- | --- | --- |
|  | **0** | **1** |
| ->A | B | A |
| B | C | A |
| \*C | C | A |

**Exercise 2.2.6(a)**

The trick is to realize that reading another bit either multiplies the number seen so far by 2 (if it is a 0), or multiplies by 2 and then adds 1 (if it is a 1). We don't need to remember the entire number seen --- just its remainder when divided by 5. That is, if we have any number of the form *5a+b*, where *b* is the remainder, between 0 and 4, then *2(5a+b) = 10a+2b*. Since 10*a* is surely divisible by 5, the remainder of *10a+2b* is the same as the remainder of 2b when divided by 5. Since *b*, is 0, 1, 2, 3, or 4, we can tabulate the answers easily. The same idea holds if we want to consider what happens to *5a+b* if we multiply by 2 and add 1.

The table below shows this automaton. State *qi* means that the input seen so far has remainder *i* when divided by 5.

|  |  |  |
| --- | --- | --- |
|  | **0** | **1** |
| ->\*q0 | q0 | q1 |
| q1 | q2 | q3 |
| q2 | q4 | q0 |
| q3 | q1 | q2 |
| q4 | q3 | q4 |

There is a small matter, however, that this automaton accepts strings with leading 0's. Since the problem calls for accepting only those strings that begin with 1, we need an additional state *s*, the start state, and an additional ``dead state'' *d*. If, in state *s*, we see a 1 first, we act like *q0*; i.e., we go to state *q1*. However, if the first input is 0, we should never accept, so we go to state *d*, which we never leave. The complete automaton is:

|  |  |  |
| --- | --- | --- |
|  | **0** | **1** |
| ->s | d | q1 |
| \*q0 | q0 | q1 |
| q1 | q2 | q3 |
| q2 | q4 | q0 |
| q3 | q1 | q2 |
| q4 | q3 | q4 |
| d | d | d |

**Exercise 2.2.9**

Part (a) is an easy induction on the length of *w*, starting at length 1.

Basis: *|w|* = 1. Then *δ-hat(q0,w) = δ-hat(qf,w)*, because *w* is a single symbol, and *δ-hat* agrees with *δ* on single symbols.

Induction: Let *w = za*, so the inductive hypothesis applies to *z*. Then *δ-hat(q0,w) = δ-hat(q0,za) = δ(δ-hat(q0,z),a) = δ(δ-hat(qf,z),a)* [by the inductive hypothesis] = *δ-hat(qf,za) = δ-hat(qf,w)*.

For part (b), we know that *δ-hat(q0,x) = qf*. Since *xε*, we know by part (a) that *δ-hat(qf,x) = qf*. It is then a simple induction on *k* to show that *δ-hat(q0,xk) = qf*.

Basis: For *k=1* the statement is given.

Induction: Assume the statement for *k-1*; i.e., *δ-hat(q0,xSUP>k-1) = qf*. Using Exercise 2.2.2, *δ-hat(q0,xk) = δ-hat(δ-hat(q0,xk-1),x) = δ-hat(qf,x)* [by the inductive hypothesis] = *qf* [by (a)].

**Exercise 2.2.10**

The automaton tells whether the number of 1's seen is even (state *A*) or odd (state *B*), accepting in the latter case. It is an easy induction on *|w|* to show that *dh(A,w) = A* if and only if *w* has an even number of 1's.

Basis: *|w|* = 0. Then *w*, the empty string surely has an even number of 1's, namely zero 1's, and *δ-hat(A,w) = A*.

Induction: Assume the statement for strings shorter than *w*. Then *w = za*, where *a* is either 0 or 1.

Case 1: *a* = 0. If *w* has an even number of 1's, so does *z*. By the inductive hypothesis, *δ-hat(A,z) = A*. The transitions of the DFA tell us *δ-hat(A,w) = A*. If *w* has an odd number of 1's, then so does *z*. By the inductive hypothesis, *δ-hat(A,z) = B*, and the transitions of the DFA tell us *δ-hat(A,w) = B*. Thus, in this case, *δ-hat(A,w) = A* if and only if *w* has an even number of 1's.

Case 2: *a* = 1. If *w* has an even number of 1's, then *z* has an odd number of 1's. By the inductive hypothesis, *δ-hat(A,z) = B*. The transitions of the DFA tell us *δ-hat(A,w) = A*. If *w* has an odd number of 1's, then *z* has an even number of 1's. By the inductive hypothesis, *δ-hat(A,z) = A*, and the transitions of the DFA tell us *δ-hat(A,w) = B*. Thus, in this case as well, *δ-hat(A,w) = A* if and only if *w* has an even number of 1's.

[Return to Top](http://infolab.stanford.edu/~ullman/ialcsols/sol2.html#top)

**Solutions for Section 2.3**

**Exercise 2.3.1**

Here are the sets of NFA states represented by each of the DFA states A through H: A = {p}; B = {p,q}; C = {p,r}; D = {p,q,r}; E = {p,q,s}; F = {p,q,r,s}; G = {p,r,s}; H = {p,s}.

|  |  |  |
| --- | --- | --- |
|  | **0** | **1** |
| ->A | B | A |
| B | D | C |
| C | E | A |
| D | F | C |
| \*E | F | G |
| \*F | F | G |
| \*G | E | H |
| \*H | E | H |

**Exercise 2.3.4(a)**

The idea is to use a state *qi*, for *i* = 0,1,...,9 to represent the idea that we have seen an input *i* and guessed that this is the repeated digit at the end. We also have state *qs*, the initial state, and *qf*, the final state. We stay in state *qs* all the time; it represents no guess having been made. The transition table:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **0** | **1** | **...** | **9** |
| ->qs | {qs,q0} | {qs,q1} | ... | {qs,q9} |
| q0 | {qf} | {q0} | ... | {q0} |
| q1 | {q1} | {qf} | ... | {q1} |
| ... | ... | ... | ... | ... |
| q9 | {q9} | {q9} | ... | {qf} |
| \*qf | {} | {} | ... | {} |

[Return to Top](http://infolab.stanford.edu/~ullman/ialcsols/sol2.html#top)

**Solutions for Section 2.4**

**Exercise 2.4.1(a)**

We'll use q0 as the start state. q1, q2, and q3 will recognize *abc*; q4, q5, and q6 will recognize *abd*, and q7 through q10 will recognize *aacd*. The transition table is:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **a** | **b** | **c** | **d** |
| ->q0 | {q0,q1,q4,q7} | {q0} | {q0} | {q0} |
| q1 | {} | {q2} | {} | {} |
| q2 | {} | {} | {q3} | {} |
| \*q3 | {} | {} | {} | {} |
| q4 | {} | {q5} | {} | {} |
| q5 | {} | {} | {} | {q6} |
| \*q6 | {} | {} | {} | {} |
| q7 | {q8} | {} | {} | {} |
| q8 | {} | {} | {q9} | {} |
| q9 | {} | {} | {} | {q10} |
| \*q10 | {} | {} | {} | {} |

**Exercise 2.4.2(a)**

The subset construction gives us the following states, each representing the subset of the NFA states indicated: *A = {q0}; B = {q0,q1,q4,q7}; C = {q0,q1,q4,q7,q8}; D = {q0,q2,q5}; E = {q0,q9}; F = {q0,q3}; G = {q0,q6}; H = {q0,q10}*. Note that *F, G* and *H* can be combined into one accepting state, or we can use these three state to signal the recognition of *abc, abd*, and *aacd*, respectively.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **a** | **b** | **c** | **d** |
| ->A | B | A | A | A |
| B | C | D | A | A |
| C | C | D | E | A |
| D | B | A | F | G |
| E | B | A | A | H |
| \*F | B | A | A | A |
| \*G | B | A | A | A |
| \*H | B | A | A | A |

[Return to Top](http://infolab.stanford.edu/~ullman/ialcsols/sol2.html#top)

**Solutions for Section 2.5**

**Exercise 2.5.1**

For part (a): the closure of *p* is just *{p}*; for *q* it is *{p,q}*, and for *r* it is *{p,q,r}*.

For (b), begin by noticing that *a* always leaves the state unchanged. Thus, we can think of the effect of strings of *b*'s and *c*'s only. To begin, notice that the only ways to get from *p* to *r* for the first time, using only *b*, *c*, and ε-transitions are *bb*, *bc*, and *c*. After getting to *r*, we can return to *r* reading either *b* or *c*. Thus, every string of length 3 or less, consisting of *b*'s and *c*'s only, is accepted, with the exception of the string *b*. However, we have to allow *a*'s as well. When we try to insert *a*'s in these strings, yet keeping the length to 3 or less, we find that every string of *a*'s *b*'s, and *c*'s with at most one *a* is accepted. Also, the strings consisting of one *c* and up to 2 *a*'s are accepted; other strings are rejected.

There are three DFA states accessible from the initial state, which is the ε closure of *p*, or *{p}*. Let *A = {p}, B = {p,q}*, and *C = {p,q,r}*. Then the transition table is:

|  |  |  |  |
| --- | --- | --- | --- |
|  | **a** | **b** | **c** |
| ->A | A | B | C |
| B | B | C | C |
| \*C | C | C | C |

[Return to Top](http://infolab.stanford.edu/~ullman/ialcsols/sol2.html#top)

**Solutions for Section 3.1**

**Exercise 3.1.1(a)**

The simplest approach is to consider those strings in which the first *a* precedes the first *b* separately from those where the opposite occurs. The expression: **c\*a(a+c)\*b(a+b+c)\* + c\*b(b+c)\*a(a+b+c)\***

**Exercise 3.1.2(a)**

(Revised 9/5/05) The trick is to start by writing an expression for the set of strings that have no two adjacent 1's. Here is one such expression: **(10+0)\*(**ε**+1)**

To see why this expression works, the first part consists of all strings in which every 1 is followed by a 0. To that, we have only to add the possibility that there is a 1 at the end, which will not be followed by a 0. That is the job of (ε+1).

Now, we can rethink the question as asking for strings that have a prefix with no adjacent 1's followed by a suffix with no adjacent 0's. The former is the expression we developed, and the latter is the same expression, with 0 and 1 interchanged. Thus, a solution to this problem is **(10+0)\*(**ε**+1)(01+1)\*(**ε**+0)**. Note that the ε+1 term in the middle is actually unnecessary, as a 1 matching that factor can be obtained from the **(01+1)\*** factor instead.

**Exercise 3.1.4(a)**

This expression is another way to write ``no adjacent 1's.'' You should compare it with the different-looking expression we developed in the solution to Exercise 3.1.2(a). The argument for why it works is similar. **(00\*1)\***says every 1 is preceded by at least one 0. **0\*** at the end allows 0's after the final 1, and (ε+**1**) at the beginning allows an initial 1, which must be either the only symbol of the string or followed by a 0.

**Exercise 3.1.5**

The language of the regular expression ε. Note that ε\* denotes the language of strings consisting of any number of empty strings, concatenated, but that is just the set containing the empty string.

[Return to Top](http://infolab.stanford.edu/~ullman/ialcsols/sol3.html#top)

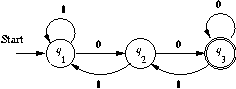
**Solutions for Section 3.2**

**Exercise 3.2.1**

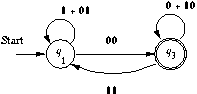
Part (a): The following are all R0 expressions; we list only the subscripts. R11 = ε+**1**; R12 = **0**; R13 = phi; R21 = **1**; R22 = ε; R23 = **0**; R31 = phi; R32 = **1**; R33 = ε+**0**.

Part (b): Here all expression names are R(1); we again list only the subscripts. R11 = **1**\*; R12 = **1\*0**; R13 = phi; R21 = **11\***; R22 = ε+**11\*0**; R23 = **0**; R31 = phi; R32 = **1**; R33 = ε+**0**.

Part (e): Here is the transition diagram:

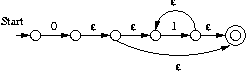


If we eliminate state *q2* we get:



Applying the formula in the text, the expression for the ways to get from *q1* to *q3* is: **[1 + 01 + 00(0+10)\*11]\*00(0+10)\***

**Exercise 3.2.4(a)**



**Exercise 3.2.6(a)**

(Revised 1/16/02) *LL*\* or *L+*.

**Exercise 3.2.6(b)**

The set of suffixes of strings in *L*.

**Exercise 3.2.8**

Let *R(k)ijm* be the number of paths from state *i* to state *j* of length *m* that go through no state numbered higher than *k*. We can compute these numbers, for all states *i* and *j*, and for *m* no greater than *n*, by induction on *k*.

Basis: *R0ij1* is the number of arcs (or more precisely, arc labels) from state *i* to state *j*. *R0ii0* = 1, and all other *R0ijm*'s are 0.

Induction: *R(k)ijm* is the sum of *R(k-1)ijm* and the sum over all lists *(p1,p2,...,pr)* of positive integers that sum to *m*, of *R(k-1)ikp1 \* R(k-1)kkp2 \*R(k-1)kkp3 \*...\* R(k-1)kkp(r-1) \* R(k-1)kjpr*. Note *r* must be at least 2.

The answer is the sum of *R(k)1jn*, where *k* is the number of states, 1 is the start state, and *j* is any accepting state.

[Return to Top](http://infolab.stanford.edu/~ullman/ialcsols/sol3.html#top)

**Solutions for Section 3.4**

**Exercise 3.4.1(a)**

Replace *R* by *{a}* and *S* by *{b}*. Then the left and right sides become *{a} union {b} = {b} union {a}*. That is, *{a,b} = {b,a}*. Since order is irrelevant in sets, both languages are the same: the language consisting of the strings*a* and *b*.

**Exercise 3.4.1(f)**

Replace *R* by *{a}*. The right side becomes *{a}\**, that is, all strings of *a*'s, including the empty string. The left side is *({a}\*)\**, that is, all strings consisting of the concatenation of strings of *a*'s. But that is just the set of strings of*a*'s, and is therefore equal to the right side.

**Exercise 3.4.2(a)**

Not the same. Replace *R* by *{a}* and *S* by *{b}*. The left side becomes all strings of *a*'s and *b*'s (mixed), while the right side consists only of strings of *a*'s (alone) and strings of *b*'s (alone). A string like *ab* is in the language of the left side but not the right.

**Exercise 3.4.2(c)**

Also not the same. Replace *R* by *{a}* and *S* by *{b}*. The right side consists of all strings composed of zero or more occurrences of strings of the form *a...ab*, that is, one or more *a*'s ended by one *b*. However, every string in the language of the left side has to end in *ab*. Thus, for instance, *ε* is in the language on the right, but not on the left.

[Return to Top](http://infolab.stanford.edu/~ullman/ialcsols/sol3.html#top)

## Solutions for Section 4.1

### Exercise 4.1.1(c)

Let *n* be the pumping-lemma constant (note this *n* is unrelated to the *n* that is a local variable in the definition of the language *L*). Pick *w = 0n10n*. Then when we write *w = xyz*, we know that *|xy| <= n*, and therefore *y*consists of only 0's. Thus, *xz*, which must be in *L* if *L* is regular, consists of fewer than *n* 0's, followed by a 1 and exactly *n* 0's. That string is not in *L*, so we contradict the assumption that *L* is regular.

### Exercise 4.1.2(a)

Let *n* be the pumping-lemma constant and pick *w = 0n2*, that is, *n2* 0's. When we write *w = xyz*, we know that *y* consists of between 1 and *n* 0's. Thus, *xyyz* has length between *n2 + 1* and *n2 + n*. Since the next perfect square after *n2* is *(n+1)2 = n2 + 2n + 1*, we know that the length of *xyyz* lies strictly between the consecutive perfect squares *n2* and *(n+1)2*. Thus, the length of *xyyz* cannot be a perfect square. But if the language were regular, then *xyyz* would be in the language, which contradicts the assumption that the language of strings of 0's whose length is a perfect square is a regular language.

### Exercise 4.1.4(a)

We cannot pick *w* from the empty language.

### Exercise 4.1.4(b)

If the adversary picks *n = 3*, then we cannot pick a *w* of length at least *n*.

### Exercise 4.1.4(c)

The adversary can pick an *n > 0*, so we have to pick a nonempty *w*. Since *w* must consist of pairs 00 and 11, the adversary can pick *y* to be one of those pairs. Then whatever *i* we pick, *xyiz* will consist of pairs 00 and 11, and so belongs in the language.

[Return to Top](http://infolab.stanford.edu/~ullman/ialcsols/sol4.html#top)

## Solutions for Section 4.2

### Exercise 4.2.1(a)

*aabbaa*.

### Exercise 4.2.1(c)

The language of regular expression **a(ab)\*ba**.

### Exercise 4.2.1(e)

Each *b* must come from either 1 or 2. However, if the first *b* comes from 2 and the second comes from 1, then they will both need the *a* between them as part of *h(2)* and *h(1)*, respectively. Thus, the inverse homomorphism consists of the strings *{110, 102, 022}*.

### Exercise 4.2.2

Start with a DFA *A* for *L*. Construct a new DFA *B*, that is exactly the same as *A*, except that state *q* is an accepting state of *B* if and only if *δ(q,a)* is an accepting state of *A*. Then *B* accepts input string *w* if and only if *A*accepts *wa*; that is, *L(B) = L/a*.

### Exercise 4.2.5(b)

We shall use *Da* for ``the derivative with respect to *a*.'' The key observation is that if *epsilon* is not in *L(R)*, then the derivative of *RS* will always remove an *a* from the portion of a string that comes from *R*. However, if*epsilon* is in *L(R)*, then the string might have nothing from *R* and will remove *a* from the beginning of a string in *L(S)* (which is also a string in *L(RS)*. Thus, the rule we want is:

If *epsilon* is not in *L(R)*, then *Da(RS) = (Da(R))S*. Otherwise, *Da(RS) = Da(R)S + Da(S)*.

### Exercise 4.2.5(e)

*L* may have no string that begins with 0.

### Exercise 4.2.5(f)

This condition says that whenever *0w* is in *L*, then *w* is in *L*, and vice-versa. Thus, *L* must be of the form *L(0\*)M* for some language *M* (not necessarily a regular language) that has no string beginning with 0.

In proof, notice first that *D0(L(0\*)M = D0(L(0\*))M union D0(M) = L(0\*)M*. There are two reasons for the last step. First, observe that *D0* applied to the language of all strings of 0's gives all strings of 0's, that is, *L(0\*)*. Second, observe that because *M* has no string that begins with 0, *D0(M)* is the empty set [that's part (e)].

We also need to show that every language *N* that is unchanged by *D0* is of this form. Let *M* be the set of strings in *N* that do not begin with 0. If *N* is unchanged by *D0*, it follows that for every string *w* in *M*, *00...0w* is in *N*; thus, *N* includes all the strings of *L(0\*)M*. However, *N* cannot include a string that is not in *L(0\*)M*. If *x* were such a string, then we can remove all the 0's at the beginning of *x* and get some string *y* that is also in *N*. But *y*must also be in *M*.

### Exercise 4.2.8

Let *A* be a DFA for *L*. We construct DFA *B* for *half(L)*. The state of *B* is of the form *[q,S]*, where:

* *q* is the state *A* would be in after reading whatever input *B* has read so far.
* *S* is the set of states of *A* such that *A* can get from exactly these states to an accepting state by reading any input string whose length is the same as the length of the string *B* has read so far.

It is important to realize that it is not necessary for *B* to know how many inputs it has read so far; it keeps this information up-to-date each time it reads a new symbol. The rule that keeps things up to date is: *δB([q,S],a) = [δA(q,a),T]*, where *T* is the set of states *p* of *A* such that there is a transition from *p* to any state of *S* on any input symbol. In this manner, the first component continues to simulate *A*, while the second component now represents states that can reach an accepting state following a path that is one longer than the paths represented by *S*.

To complete the construction of *B*, we have only to specify:

* The initial state is *[q0,F]*, that is, the initial state of *A* and the accepting states of *A*. This choice reflects the situation when *A* has read 0 inputs: it is still in its initial state, and the accepting states are exactly the ones that can reach an accepting state on a path of length 0.
* The accepting states of *B* are those states *[q,S]* such that *q* is in *S*. The justification is that it is exactly these states that are reached by some string of length *n*, and there is some other string of length *n* that will take state *q* to an accepting state.

### Exercise 4.2.13(a)

Start out by complementing this language. The result is the language consisting of all strings of 0's and 1's that are *not* in 0\*1\*, plus the strings in *L0n1n*. If we intersect with 0\*1\*, the result is exactly *L0n1n*. Since complementation and intersection with a regular set preserve regularity, if the given language were regular then so would be *L0n1n*. Since we know the latter is false, we conclude the given language is not regular.

### Exercise 4.2.14(c)

Change the accepting states to be those for which the first component is an accepting state of *AL* and the second is a nonaccepting state of *AM*. Then the resulting DFA accepts if and only if the input is in *L - M*.

[Return to Top](http://infolab.stanford.edu/~ullman/ialcsols/sol4.html#top)

## Solutions for Section 4.3

### Exercise 4.3.1

Let *n* be the pumping-lemma constant. Test all strings of length between *n* and 2*n*-1 for membership in *L*. If we find even one such string, then *L* is infinite. The reason is that the pumping lemma applies to such a string, and it can be ``pumped'' to show an infinite sequence of strings are in *L*.

Suppose, however, that there are no strings in *L* whose length is in the range *n* to 2*n*-1. We claim there are no strings in *L* of length 2*n* or more, and thus there are only a finite number of strings in *L*. In proof, suppose *w* is a string in *L* of length at least 2*n*, and *w* is as short as any string in *L* that has length at least 2*n*. Then the pumping lemma applies to *w*, and we can write *w = xyz*, where *xz* is also in *L*. How long could *xz* be? It can't be as long as 2*n*, because it is shorter than *w*, and *w* is as short as any string in *L* of length 2*n* or more.