# CS 241 – Week 6 Tutorial Solutions

Regular Languages: DFAs and Regular Expressions

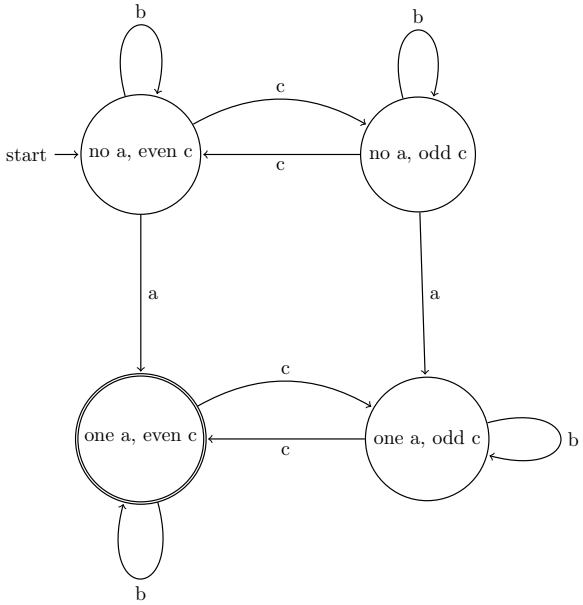
Spring 2015

## 1 DFA Solutions

1. We want to find a DFA for the language of strings over a,b,c with an an $a$ and an even number of $c$s. Our DFA should accept strings such as $ab$, $abcc$ and $abbbcccbbccc$ while rejecting strings like $baabcc$, $babccc$ and the empty string.

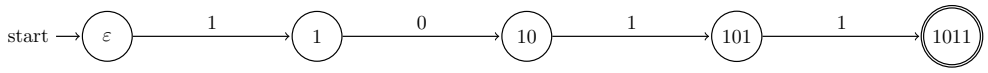
   Generally, a good way to approach DFA problems is to think about what states you will need to differentiate strings that are not in the language from strings that are in the language. Once you figure out the set of states, filling in the transitions is often straightforward.

   For example, we might have a state labelled "one a, even c", which we enter whenever we have consumed an $a$ and an even number of $c$s. It should be an accepting state, since these are precisely the strings we want to accept. To differentiate between these strings and ones not in the language, we might need to create states representing other possibilities. Namely, "one a, odd c", "no a, even c", "no a, odd c", "> one a, odd c", "> one a, even c". None of these states should be accepting. Note that we do not need to explicitly create the states "> one a, odd c" and "> one a, even c", because once we read more than one 'a' there is no way that the string can possibly be accepted: we are essentially in an error state.




2. In this problem, we want a DFA that recognizes binary strings ending in 1011. For example, 1011, 0001011, and 1011011101001011 should be accepted, and strings such as 1010110 and anything else not ending in 1011 should be rejected.

We proceed once again by figuring out what states are necessary to recognize this language. Sometimes when you are not sure what states you will need, it is helpful to pick a simple example string from the language and figure out what states you will need to recognize just that particular string. For example, it is clear that the states and transitions shown below are required to recognize 1011:



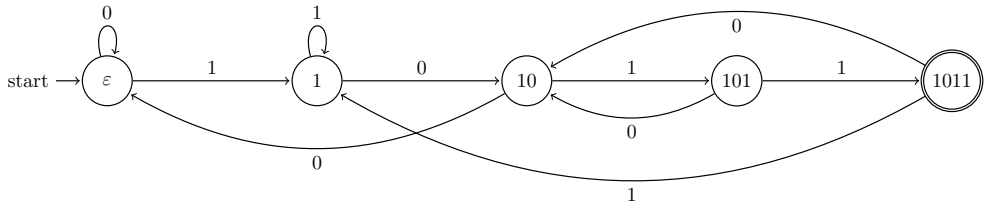

Think about what these states correspond to. If you are in the state 1 that means the string you have read ends in 1, and you must read 011 to obtain a string in the language. If you are in the state 101, the string you have read ends in 101 and you only have to read 1 to get a string in the language.

The states track how much of the terminating suffix 1011 we have seen, and how much we still need to see to complete the suffix. If the entire suffix is read, we will be in the accepting state 1011. Otherwise, we will be in one of the intermediate states leading up to it.

To complete the DFA, we just need to fill in the missing transitions one by one.

- When we are in the initial state, we still need to see the characters 1011 in order to reach the accepting state. If we next read a 0, we haven't seen the first character of that suffix, so we loop on 0.
- When we are in the 1 state, our string ends in 1 and we still want to see 011. If we read a 1, our string ends in 11 and we still want to see 011. This means we must loop on 1.
- When we are in the 10 state, our string ends in 10 and we still want to see 11. If we read a 0, our string ends in 100, which means we must see the entire suffix 1011 next to reach the accepting state. So we return to the initial state on 0.
- When we are in the 101 state, out string ends in 101 and we still want to see 1. If we read a 0, our string ends in 1010, which means we need to see 11 to reach the accepting state. So we should go to the state 10 on 0.
- When we are in the accepting state 1011, our string ends in the desired suffix. If we read a 1, it ends in 10111, which means we must see 011 next to reach the accepting state, so we transition to state 1. If we read a 0, it ends in 10110, so we transition to state 10.

We obtain this DFA, which accepts the desired language:



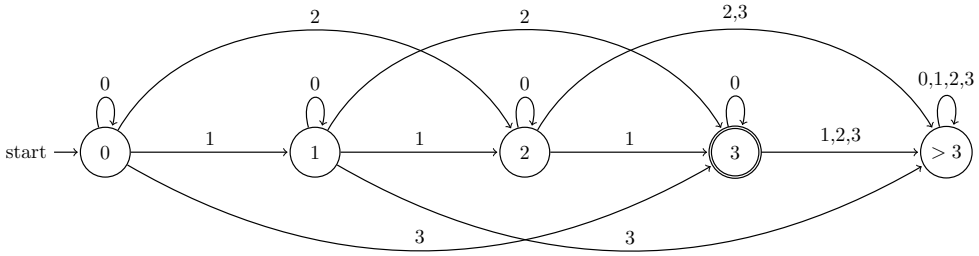

3. We want to find a DFA that recognizes strings over the alphabet 0,1,2,3 which are integers with a digit sum of 3, and may have leading zeros. For example, 120 should be in the language since $1 + 2 + 0 = 3$, and 0003 should be in the language since $0 + 0 + 0 + 3 = 3$, but 20 and 231 should not be in the language since their digit sums are 2 and 6, respectively.

   Once again, the best way to think about this problem is to think about what states we could use to distinguish strings in the language from those not in the language. Since we want to accept strings with a digit sum of 3, it make sense to have our states track the digit sum. We can have an accepting state called 3, which represents that the sum of the digits we have read so far is 3. Then we can have non-accepting states called 0, 1 and 2 corresponding to each of those digit sums. We also want to reject

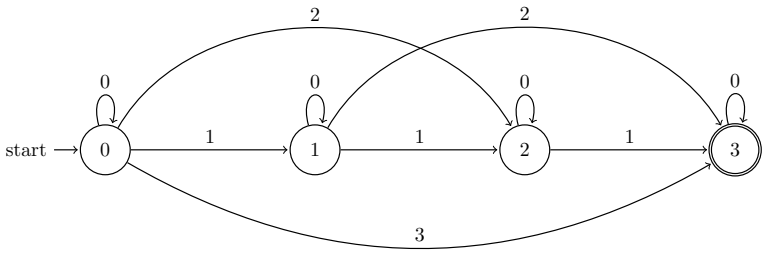
all strings with a digit sum larger than 3; we could add another non-accepting state called something like >3 to represent this.

The transition function can then be defined very easily. If we are in state $x$ and the symbol $i$ is next in the input, then the digit sum of the integer we have seen so far is $x$ and the new digit sum after reading $i$ will be $x+i$. Thus we should have transitions from state $x$ on symbol $i$ to state $x+i$ if $x+i$ is less than or equal to 3, or to state >3 if $x+i$ is greater than 3. For example, we would have a transition from state 1 on the symbol 2 to state 3, since $1 + 2 = 3$, and a transition from state 3 on symbol 1 to state >3 since $3 + 1 > 3$.

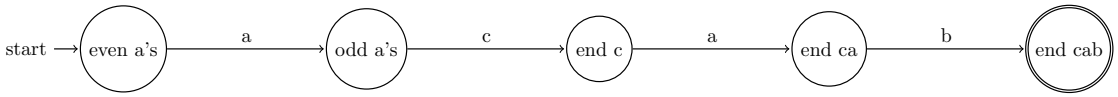
Filling in all the transitions in this way, we get the following DFA:




Notice that the >3 state is not only non-accepting, but it loops back to itself on every symbol. Instead of drawing this state, we could use the implicit error state convention discussed in class. If a transition is not drawn on the DFA diagram, we can assume it implicitly goes to a non-accepting error state that loops back to itself on every symbol, and thus if we encounter an undefined transition when trying to recognize a string, the input will be rejected. Using this convention lets us remove the >3 state from the diagram and make it simpler (though we would still have to list the error state in a formal description of this DFA).



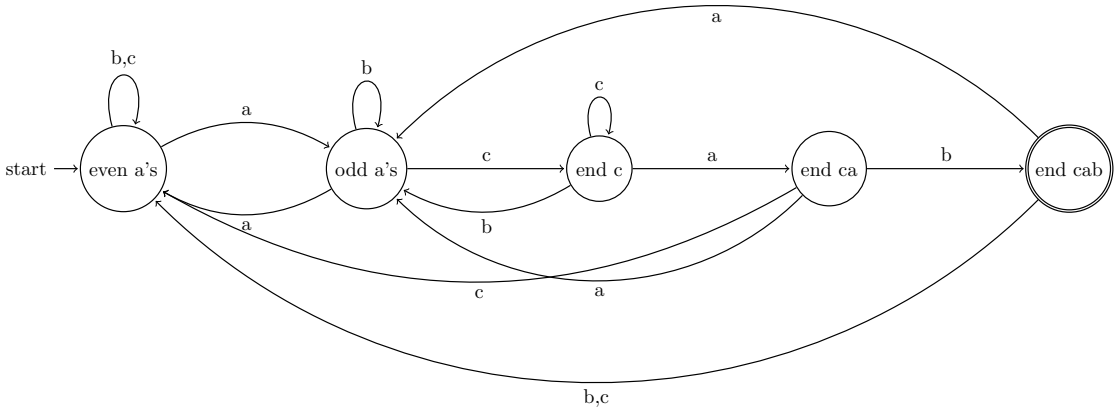

4. We want to find a DFA that recognizes strings over the alphabet a,b,c which end in cab and have an even number of a's. Since cab itself has one a, which is an odd number of a's, there must be at least one a before cab. So the smallest string in our language is acab. We can build a DFA to accept this string:




Now, like for problem 2, we simply need to fill in the missing transitions:

- From the initial state if we see *b* or *c* it doesn't change the number of a's in the string so we loop.
- From the *odd a's* state if we see an *a* we go back to the *even a's* state, if we see a *b* the number of a's don't change so we loop.

- From the *end c* state if we see a $c$ we loop because we still could be reading the first letter of *cab*, if we see a $b$ we cannot be in a *cab* so we return to the *odd a's* state.
- From the *end ca* state if we see a $c$ we return to the *even a's* state, if we see an $a$ we return to the *odd a's* state.
- From the *end cab* state if we see a $b$ or $c$ we return to the even a's state, and if we see an $a$ we return to the *odd a's* state.




## 2 Regular Expression Solutions

Regular languages:

1. $\{0,1\}\{0\}\{0,1\}\{0,1\}\{1\}\{0,1\}^*$
2. $\{0,1\}^*\{1\}\{1\}\{0\}\{1\}\{0\}\{1\}\{0,1\}^*$

Regular expressions:

1. Naive solution:$(xx|xy|yx|yy)$. Better solution: $(x|y)(x|y)$
2. $(G|C|A|T)^*GACAT(G|C|A|T)^*$
3. (a) $(0|1)0(0|1)(0|1)1(0|1)^*$

   (b) $(0|1)^*110101(0|1)^*$

   Doesn't that feel much better?
4. The regular expression for a term is: $(a|b)$

   The regular expression for an operator is: $(+|-|*|/)$

   The structure of an arithmetic expression is term operator term operator term ... operator term. An expression must start with a term: $(a|b)$

   Then we have "operator term" repeated 0 or more times after the initial term:
   $(a|b)((+|-|*|/)(a|b))^*$
5. The first thing we realize is that we must have at least one 1. This will be the basis for our regular expression: 1

What can come before the first 1? As many even counts of 0's as we like mixed with any number of 2's: $2^*(02^*02^*)^*$. Note that we need the leading $2^*$ since otherwise we could never have leading 2's.

What can proceed the first 1? The same as can precede it plus any number of 1's as well: $(1|2)^*(0(1|2)^*0(1|2)^*)^*$

So we have: $2^*(02^*02^*)^*1(1|2)^*(0(1|2)^*0(1|2)^*)^*$. Does this solve our problem?

No. Because we could have an odd number of 0's on the left and right of the first 1 and still have an even number of 0's. How do we fix this? By either have 1 or having 1 with a 0 on the left and a zero on the right. Note that we will also have to allow for additional 2's on the left and additional 1's and 2's on the right. The new core becomes $(1|(02^*1(1|2)^*0)$

Combining this with our original left and right pieces we get:
$2^* \left(02^*02^*\right)^* \left(1| \left(02^*1 \left(1|2\right)^* 0\right)\right) \left(1|2\right)^* \left(0 \left(1|2\right)^* 0 \left(1|2\right)^*\right)^*$