Due 2/9

Write a program to simulate the execution of a partial non deterministic finite automata (partial since we don't have epsilon transitions). Your program can be written in Java, C, or C++, and needs to be able to be compiled and executed on the computers in EB-G7 (or a Linux or Mac computer I have access to). If you do not know Java, C, or C++, you will need to talk to me to discuss options for you to complete this assignment.

Your program will read the definition of the machine from a file (the first command line argument), with the second command line argument being the string to simulate the machine running on (the input to the automata). The output of your program will be written to standard output. The output will consist of either: 1) the word accept followed by a blank space followed by the list of accept states (blank space delimited) that the automata can end up in after reading in the input string (if there is a way for the automata to end in an accept state after reading the input); or 2) the word reject followed by a blank space followed by the list of states (blank space delimited) that automata can end up in after reading the string (if there is no way for the automata to end in an accept state after reading the input). Your output needs to end with the newline character.

The input file will be tab delimited (should be easily parsed by Java, C, or C++).

For this program, the states will be numbered between 0 and 1,000 (not necessarily contiguous or entered in order). There are two types of special states – the start state (only one) and the accept states (0 or more).

There are two types of input lines: state lines and transition lines.

The state lines are of the form:
state x start
state x accept
state x acceptstart
state x start accept

where x is a number in [0, 1000]. States that are niether accept or start states will not have an input line. Note in the above, there is a tab between accept and start in “state x acceptstart”.

Here are some examples:

state 7 start accept
state 10 acceptstart
state 20 accept
state 27 start

There is no guarantee that the first state is the accept state or that the states are in order or they are contiguous.

The remainder of the file defines the transitions. For this machine, the transition format is “p,x->q” where p is the current state that the machine is in, x is the symbol that the machine reads, and q is the state that the machine transitions to. There will be at most 100,000 transitions.

The format of the transitions in the file will be:

Due 2/9

transition p x q

where p and q are states in [0, 1000], and x is the symbol to read. For this program you can assume that x will be a digit {0, 1, …, 9} or a lower case letter {a, b, …, z}.

Since the machine is non deterministic, there may be multiple states that the automata can transition to for a single state and input symbol combination.

The input will be a string, consisting of digits and lower case letters. Initially the machine will be looking at the left most symbol of the input string and in the start state (just like the finite automata that we are currently discussing in class).

If the machine can end in an accept state after reading the input string, then your program should output accept followed by a blank space followed by the list accept states that the automata can end in after reading the entire input. The states can be output in any order, but each state is to be only listed once.

If the machine can never end in an accept state after reading the input string, then your program should output reject followed by a blank space followed by the list of all states that the automata can end in after reading the input. The states can be output in any order, but each state is to be only listed once. If there is no transition for the given state and input symbol, then you should assume that the current branch of the computation ends in a non accept state (although, you will not have a state for this branch to add to the list of states reached after reading the input).

For java, standard input is System.in, standard output is System.out, and standard error is System.err.
For C, standard input is stdin, standard output is stdout, and standard error is stderr.
For C++, standard input is cin, standard output is cout, and standard error is cerr.

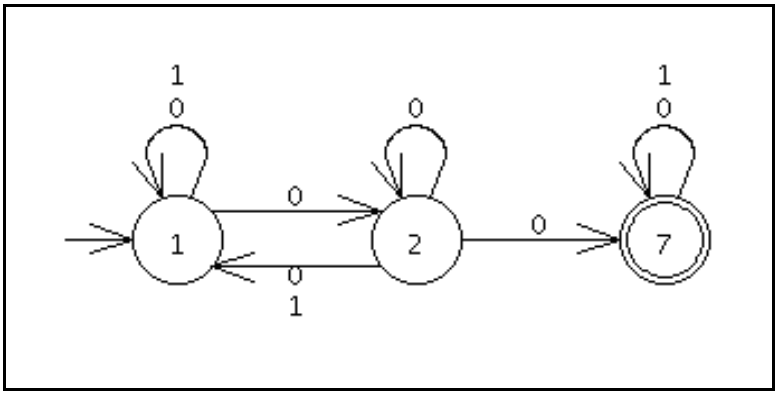
E-mail your program (source file(s) and makefile if required) to me (david.garrison@binghamton.edu) by 11:59:59pm on the date due. Your main filename must be your last name (lower case) followed by "_p1" (for example, my filename would be "garrison_p1.java") and the subject of your e-mail is "CS 373 program 1".

Your attachments to the submission e-mail are only to be your source code and makefile (if required). They should be attached directly into the e-mail (not put in a zip, tar, jar, rar, or any other archive file).

The grading will be based on the percentage of correct results your program gets for my collection of test finite automata and test strings and following the directions.

In particular, 20% of the grade will be following the instructions (filename, compilation, attachments, subject, etc) and 80% will be correct results (check my output format). If you are using java, I should be able to execute "javac "your last name in lower case_p1".java" to compile your program and java "your last name in lower case_p1" to execute. For C or C++ you should include a makefile so that I can simply execute "make "your last name in lower case_p1"" to compile to "your last name in lower case_p1" and ./"your last name in lower case_p1" to execute.

Here's is sample_1 state transition diagram.

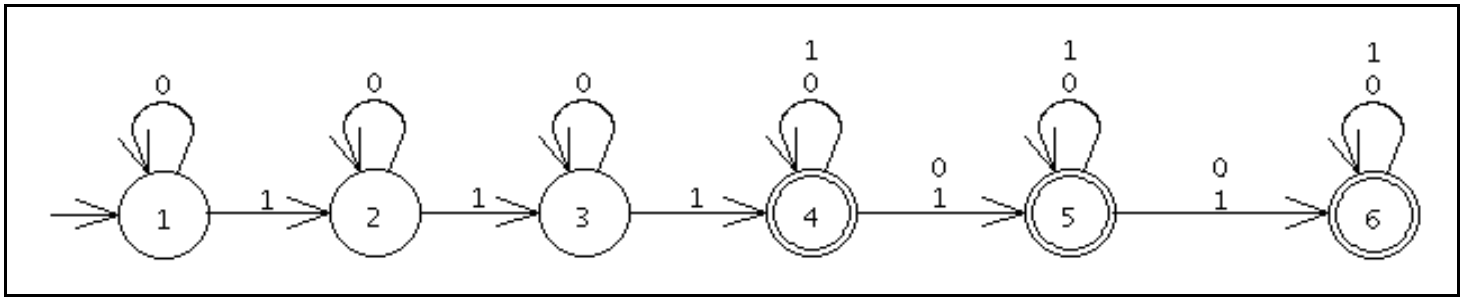


In the above 1 is the start state (has an arrow going into it from nowhere) and 7 is an accept state.

Here is the text file associated with sample_1.

```
state   1       start
state   7       accept
transition      1       0       1
transition      1       1       1
transition      1       0       2
transition      2       0       2
transition      2       0       1
transition      2       1       1
transition      2       0       7
transition      7       0       7
transition      7       1       7
```

Here's is sample_2 state transition diagram.



Here is the text file associated with sample_1.

```
state   1       start
state   4       accept
state   5       accept
state   6       accept
transition      1       0       1
transition      2       0       2
transition      3       0       3
```

```
transition    4    0    4
transition    4    1    4
transition    5    0    5
transition    5    1    5
transition    6    0    6
transition    6    1    6
transition    1    1    2
transition    2    1    3
transition    3    1    4
transition    4    1    5
transition    5    1    6
transition    4    0    5
transition    5    0    6
```

For my C++ version of the program, I only have a single file, garrison_p1.cpp. So, I can simply execute "make garrison_p1" to compile and create the executable "garrison_p1".

Below are some sample runs that ideally are correct.

```
./garrison_p1 sample_1.txt 0          ← command line
reject 1 2                            ← output written to standard output

./garrison_p1 sample_1.txt 000
accept 7

./garrison_p1 sample_1.txt 10
reject 1 2

./garrison_p1 sample_2.txt 0
reject 1

./garrison_p1 sample_2.txt 0101
reject 3

./garrison_p1 sample_2.txt 010111
accept 4 5

./garrison_p1 sample_2.txt 0101110000
accept 4 5 6
```

To simulate the execution of a finite automata we simply keep track of the current state. Initially the current state is the start state. The input string is read and processed from left to right, one symbol at a time. To process a symbol we simply find each transition that matches the current state and input symbol and "execute" the transition. A matching transition is a transition in which the current state of the automata matches the current state of the transition and the next symbol of the input string matches the

symbol associated with the transition. For each matching transition, we read the next input symbol (each transition reduces the length of the input string by one) and update the current state to the state that the machine transitions to from the transition. For each transition that matches the current state and next input symbol, we have a new configuration of the machine. This process continues until we are left with only a collection of configurations (state and remaining input string) with the remaining input string being empty (no symbols left). The automata accepts the input string if any of the configurations with empty input string have an accept state as the current state. Otherwise the string is rejected.

Your program needs to be able to handle at least 1,000,000 configurations.
I recommend you use the heap (versus the stack) to allocate large amounts of memory. For this program though, you may be able to use the stack for allocating the required memory.

For C or C++ the following functions allocate and deallocate an int array or an int matrix in which the elements can be referenced using the standard array and matrix indexing (A[i] or A[i][j]). The functions use the heap for memory allocation.

```
int *makeIntArray(int length)
{
    int *p = NULL;

    if( length > 0 )
    {
        p = (int *) malloc(length*sizeof(int));
    }
    return p;
}

void deleteIntArray(int *p)
{
    if( p != NULL )
    {
        free(p);
    }
}

int **makeIntMatrix(int width, int height)
{
    int **p = (int **) NULL;
    int *pData = (int *) NULL;

    if( (width > 0) && (height > 0) )
    {
        p = (int **) malloc(width*sizeof(int*));
        pData = (int *) malloc(width*height*sizeof(int));

        int i = 0;
```

```
        while( i < width )
        {
                p[i] = pData+(i*height);
                i = i+1;
        }
    }
    return p;
}

void deleteIntMatrix(int **p)
{
    if( p != NULL )
    {
        if( p[0] != NULL )
        {
                free(p[0]);
        }
        free(p);
    }
}
```

For Java you can use “new int[100]” or “new int[100][200]” with 100 and 200 replaced with the appropriate values.

Although the program seems somewhat complex, it is in fact relatively easy and straight forward, once you have come up with a way to keep track of the configurations.

For what is it worth, I used vectors to keep track of the collection of accept states and collection of transitions. And I used an int array (as shown earlier) to keep track of the current state of each configuration, a char matrix to keep track of the remaining input string association with each configuration, and a bool array to keep track of wether the configuration had been processed (matched with transitions and all matching transitions executed). Once all of the configuration has been processed (all executed or have empty remaining input string), I simply checked the configurations that had empty input strings to see if the string was accepted or not.

You don't need to have overly complex data structures to handle this problem. I used as linear list for the configurations. Each time a new configuration was created, I simply added to the end of the list (actually what I'm calling a list was simply an array or matrix, and adding to the end of the list is adding it to the next unused index). Your program needs to be able to handle up to 1,000,000 configurations.

My preliminary implementation gives 786,429 configurations for sample_1.txt with input 10101010101010101010101010101010.