**SAVEETHA SCHOOL OF ENGINEERING**

SAVEETHA INSTITUTE OF MEDICAL AND TECHNICAL SCIENCES

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**

**LIST OF EXPERIMENTS**

COURSE CODE : CSA13

COURSE NAME : THEORY OF COMPUTATION

1.Write a C program to simulate a Deterministic Finite Automata (DFA) for the given language representing strings that start with a and end with a

AIM :

To write a C program to simulate a Deterministic Finite Automata.

ALGORTIHM :

Draw a DFA for the given language and construct the transition table.

Store the transition table in a two-dimensional array.

Initialize present\_state, next\_state and final\_state 4. Get the input string from the user.

Find the length of the input string. 6. Read the input string character by character.

7. Repeat step 8 for every character

8. Refer the transition table for the entry corresponding to the present state and the current input symbol and update the next state.

9. When we reach the end of the input, if the final state is reached, the input is accepted. Otherwise the input is not accepted.

PROGRAM:

#include<stdio.h>

#include<string.h>

#define max 20

int main()

{

int trans\_table[4][2]={{1,3},{1,2},{1,2},{3,3}};

int final\_state=2,i;

int present\_state=0;

int next\_state=0;

int invalid=0;

char input\_string[max];

printf("Enter a string:");

scanf("%s",input\_string);

int l=strlen(input\_string);

for(i=0;i<l;i++)

{

if(input\_string[i]=='a')

next\_state=trans\_table[present\_state][0];

else if(input\_string[i]=='b')

next\_state=trans\_table[present\_state][1];

else

invalid=l;

present\_state=next\_state;

}

if(invalid==l)

{

printf("Invalid input");

}

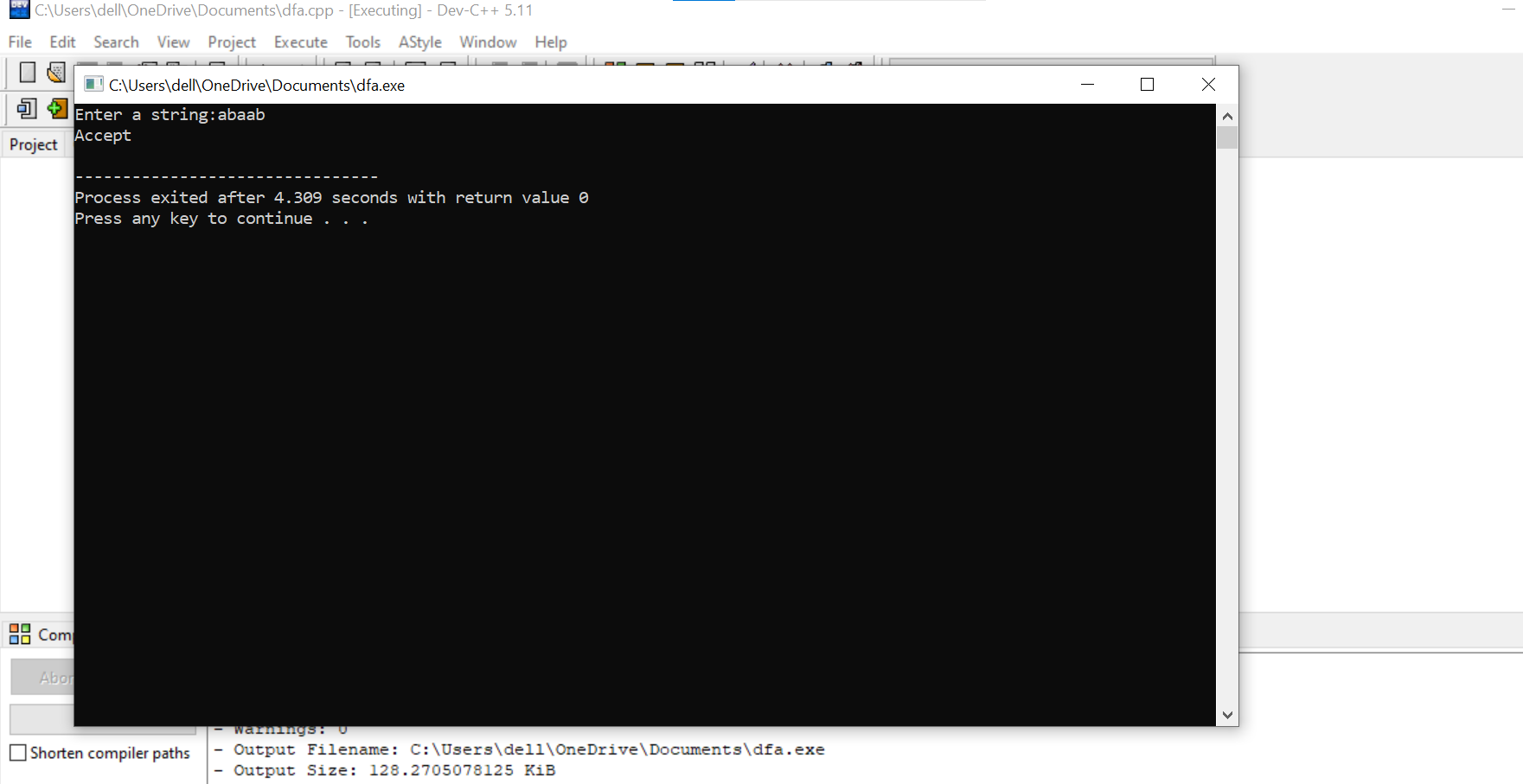
else if(present\_state==final\_state)

printf("Accept\n");

else

printf("Don't Accept\n");

}



RESULT:

THUS,executed successfully.

2. Write a C program to simulate a Deterministic Finite Automata (DFA) for the given language representing strings that start with 0 and end with 1

AIM :

To write a C program to simulate a Deterministic Finite Automata.

ALGORTIHM :

* Draw a DFA for the given language and construct the transition table.
* Store the transition table in a two-dimensional array.
* Initialize present\_state, next\_state and final\_state 4. Get the input string from the user.
* Find the length of the input string. 6. Read the input string character by character.
* Repeat step 8 for every character
* Refer the transition table for the entry corresponding to the present state and the current input symbol and update the next state.
* When we reach the end of the input, if the final state is reached, the input is accepted. Otherwise the input is not accepted.

PROGRAM:

#include<stdio.h>

#include<string.h>

#define max 20

int main()

{

int trans\_table[4][2]={{1,3},{1,2},{1,2},{3,3}};

int final\_state=2,i;

int present\_state=0;

int next\_state=0;

int invalid=0;

char input\_string[max];

printf("Enter a string:");

scanf("%s",input\_string);

int l=strlen(input\_string);

for(i=0;i<l;i++)

{

if(input\_string[i]=='a')

next\_state=trans\_table[present\_state][0];

else if(input\_string[i]=='b')

next\_state=trans\_table[present\_state][1];

else

invalid=l;

present\_state=next\_state;

}

if(invalid==l)

{

printf("Invalid input");

}

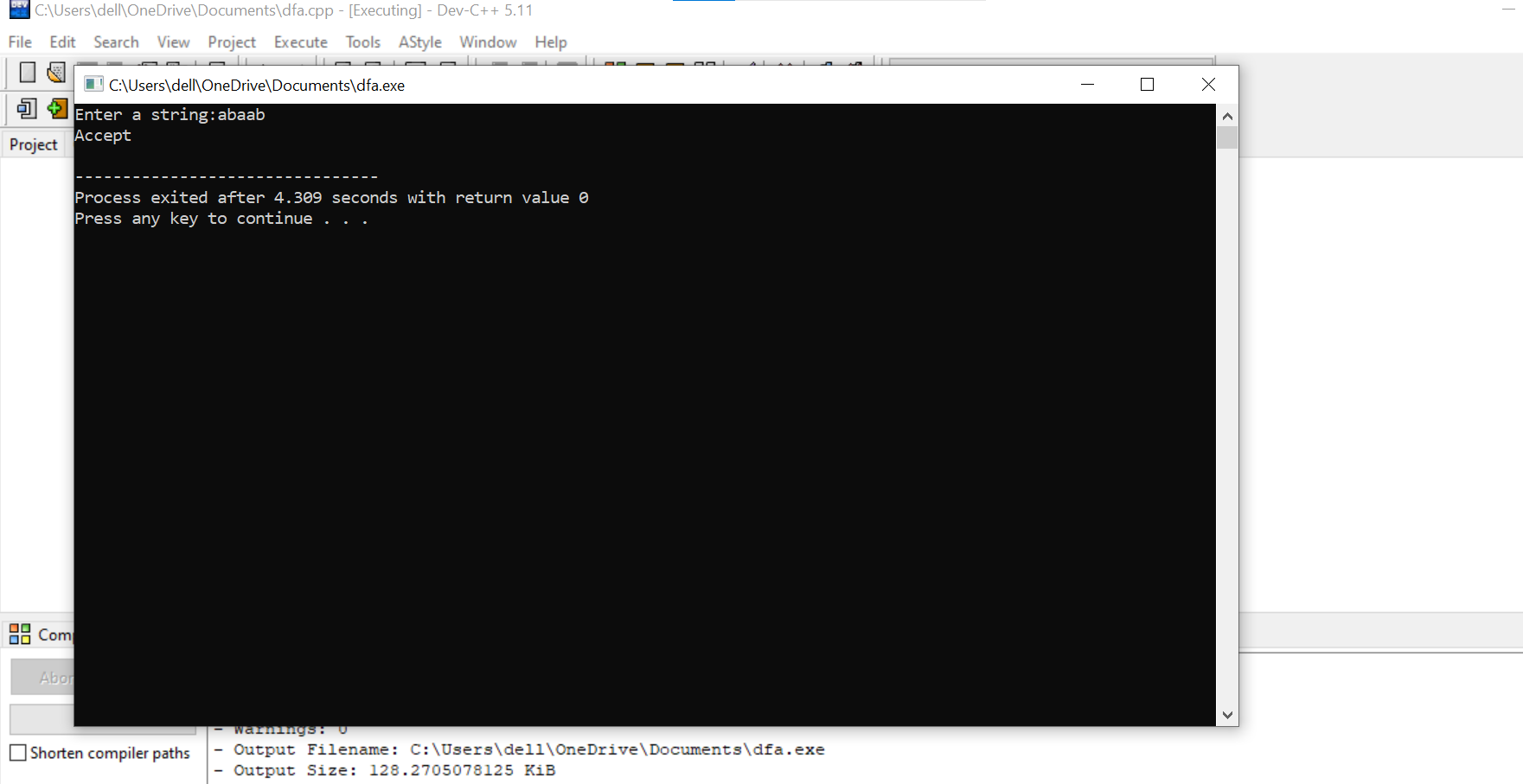
else if(present\_state==final\_state)

printf("Accept\n");

else

printf("Don't Accept\n");

}



RESULT:

THUS,executed successfully.

3.Write a C program to check whether a given string belongs to the language defined by a Context Free Grammar (CFG)

S → 0A1 A → 0A | 1A | ε

AIM :

To write a C program to find ε-closure of a Non-Deterministic Finite Automata with ε-moves

ALGORTIHM :

* Get the following as input from the user. i. Number of states in the NFA ii. Number of symbols in the input alphabet including ε iii. Input symbols iv. Number of final states and their names
* Declare a 3-dimensional matrix to store the transitions and initialize all the entries with -1
* Get the transitions from every state for every input symbol from the user and store it in the matrix. For example, consider the NFA shown below. There are 3 states 0, 1, and 2 There are three input symbols ε, 0 and 1. As the array index always starts with 0, we assume 0th symbol is ε, 1st symbol is 0 and 2nd symbol is 1. The transitions will be stored in the matrix as follows: From state 0, for input ε, there is one transition to state 1, which can be stored in the matrix as m[0][0][0]=1 From state 0, for input 0, there is no transition. From state 0, for input 1, there is one transition to state 1, whichcan be stored in the matrix as m[0][2][0]=1 Similarly, the other transitions can be stored as follows: m[1][0][0]=2 (From state 1, for input ε, the transition is to state 2) m[1][1][0]=1 (From state 1, for input 0, the transition is to state 1) All the other entries in the matrix will be -1 indicating no moves
* Initialize a two-dimensional matrix e\_closure with -1 in all the entries.
* ε-closure of a state q is defined as the set of all states that can be reached from state q using only ε-transitions.
* For every state i, find ε-closure as follows: If there is an ε-transition from state i to state j, add j to the matrix e\_closure[i]. Call the recursive function find\_e\_closure(j) and add the other states that are reachable from i using ε 7. For every state, print the ε-closure values

PROGRAM:

#include<stdio.h>

#include<string.h>

int trans\_table[10][5][3];

char symbol[5],a;

int e\_closure[10][10],ptr,state;

void find\_e\_closure(int x);

int main()

{

int i,j,k,n,num\_states,num\_symbols;

for(i=0;i<10;i++)

{

for(j=0;j<5;j++){

for(k=0;k<3;k++){

trans\_table[i][j][k]=-1;}}

printf("How may states in the NFA with e-moves:");

scanf("%d",&num\_states);

printf("How many symbols in the input alphabet including e :");

scanf("%d",&num\_symbols);

printf("Enter the symbols without space. Give 'e' first:");

scanf("%s",symbol);

for(i=0;i<num\_states;i++)

{

for(j=0;j<num\_symbols;j++)

{

printf("How many transitions from state %d for the input%c:",i,symbol[j]);

scanf("%d",&n);

for(k=0;k<n;k++)

{

printf("Enter the transitions %d from state %d for the input%c :", k+1,i,symbol[j]);

scanf("%d",&trans\_table[i][j][k]);

}}}

for(i=0;i<10;i++)

{

for(j=0;j<10;j++)

{

e\_closure[i][j]=-1;}}

for(i=0;i<num\_states;i++)

e\_closure[i][0]=i;

for(i=0;i<num\_states;i++){

if(trans\_table[i][0][0]==-1)

continue;

else

{

state=i;

ptr=1;

find\_e\_closure(i);

}

}

for(i=0;i<num\_states;i++)

{

printf("e-closure(%d)= {",i);

for(j=0;j<num\_states;j++)

{

if(e\_closure[i][j]!=-1)

{

printf("%d, ",e\_closure[i][j]);

}}

printf("}\n");}

}

void find\_e\_closure(int x)

{

int i,j,y[10],num\_trans;

i=0;

while(trans\_table[x][0][i]!=-1)

{

y[i]=trans\_table[x][0][i];

i=i+1;}

num\_trans=i;

for(j=0;j<num\_trans;j++)

{

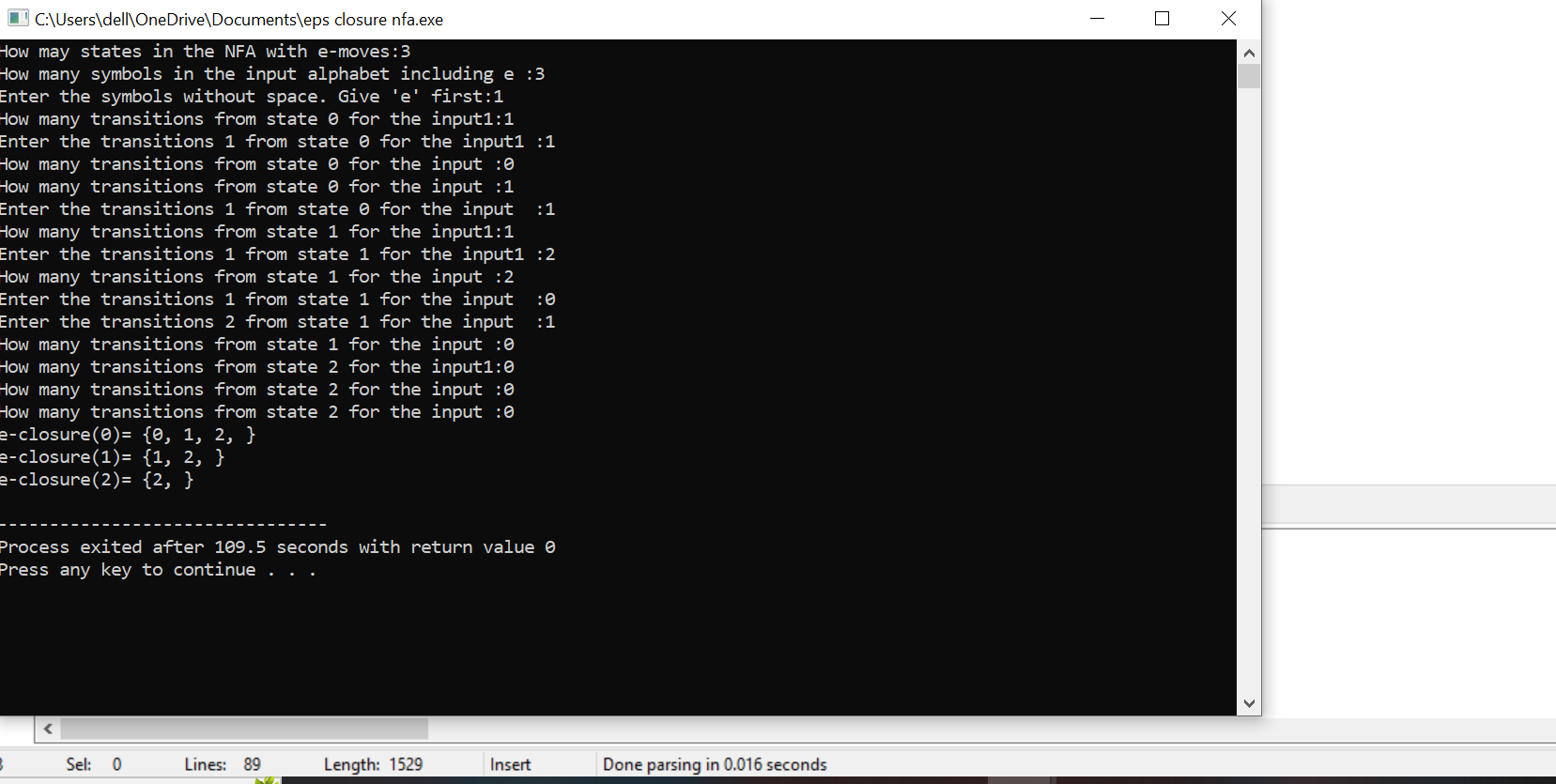
e\_closure[state][ptr]=y[j];

ptr++;

find\_e\_closure(y[j]);}

}

OUTPUT:



RESULT:

Thus ,executed successfully.

4.Write a C program to check whether a given string belongs to the language defined by a Context Free Grammar (CFG)

S → 0S0 | 1S1 | 0 | 1 | ε

**AIM :**

To write a C program to check whether a string belongs to the grammar

S -> 0 S 0 | 1 S 1 | 0 | 1 | ε

**Language defined by the Grammar**

Set of all strings over 𝚺={0,1} that are palindrome

**ALGORITHM :**

1. Get the input string from the user.

2. Find the length of the string. Let it be n.

3. Check whether all the symbols in the input are either 0 or 1. If so,

print “String is valid” and go to step 4. Otherwise print “String not

valid” and quit the program.

4. If the 1st symbol and nth symbol are the same, 2nd symbol and (n-1)th

symbol are the same and so on, then the given string is palindrome.

So, print “String accepted”. Otherwise, print “String not accepted”

**PROGRAM :**

#include<stdio.h>

#include<string.h>

void main()

{

char s[100];

int i,flag,flag1,a,b;

int l;

printf("enter a string to check:");

scanf("%s",s);

l=strlen(s);

flag=1;

for(i=0;i<l;i++)

{

if(s[i]!='0' && s[i]!='1')

{

flag=0;

}

}

if(flag!=1)

printf("string is Not Valid\n");

if(flag==1)

{

flag1=1;

a=0;b=l-1;

while(a!=(l/2))

{

if(s[a]!=s[b])

{

flag1=0;

}

a=a+1;

b=b-1;

}

if (flag1==1)

{

printf("The string is a palindrome\n");

printf("string is accepted\n");

}

else

{

printf("The string is not a palindrome\n");

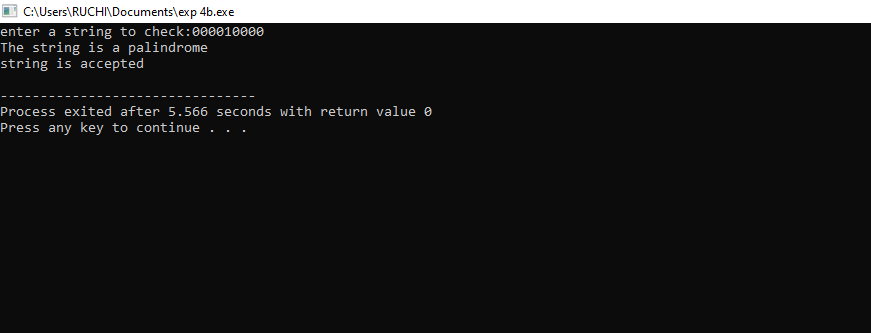
printf("string is Not accepted\n");

}

}

}

**OUTPUT :**



RESULT:

Thus executed successfully

5. Write a C program to check whether a given string belongs to the language defined by a Context Free Grammar (CFG)

S → 0S0 | A A → 1A | ε

**AIM :**

To write a C program to check whether a string belongs to the grammar

S -> 0 S 0 | A

A -> 1 A | ε

**Language defined by the Grammar**

Set of all strings over 𝚺={0,1} satisfying 0n1m0n

**ALGORITHM :**

1. Get the input string from the user.

2. Find the length of the string.

3. Check whether all the symbols in the input are either 0 or 1. If so,

print “String is valid” and go to step 4. Otherwise print “String not

valid” and quit the program.

4. Read the input string character by character

5. Count the number of 0’s in the front and store it in the variable

*count1*

6. Skip all 1’s

*7.* Count the number of 0’s in the end and store it in the variable *count2*

8. If *count1==count2*, print “String Accepted”. Otherwise print “String

Not Accepted”

**PROGRAM :**

#include<stdio.h>

#include<string.h>

void main()

{

char s[100];

int i,flag,flag1,a,b;

int l,count1,count2;

printf("enter a string to check:");

scanf("%s",s);

l=strlen(s);

flag=1;

for(i=0;i<l;i++)

{

if(s[i]!='0' && s[i]!='1')

{

flag=0;

}

}

if(flag!=1)

printf("string is Not Valid\n");

if(flag==1)

{

i=0;count1=0;

while(s[i]=='0') // Count the no of 0s in the front

{

count1++;

i++;

}

while(s[i]=='1')

{

i++; // Skip all 1s

}

flag1=1;

count2=0;

while(i<l)

{

if(s[i]=='0')// Count the no of 0s at the end

{

count2++;

}

else

{

flag1=0;

}

i++;

}

if(flag1==1)

{

if(count1==count2)

{

printf("The string satisfies the condition 0n1m0n\n");

printf ("String Accepted\n");

}

else

{

printf("The string does not satisfy the condition 0n1m0n\n");

printf("String Not Accepted\n");

}

}

else

{

printf("The string does not satisfy the condition 0n1m0n\n");

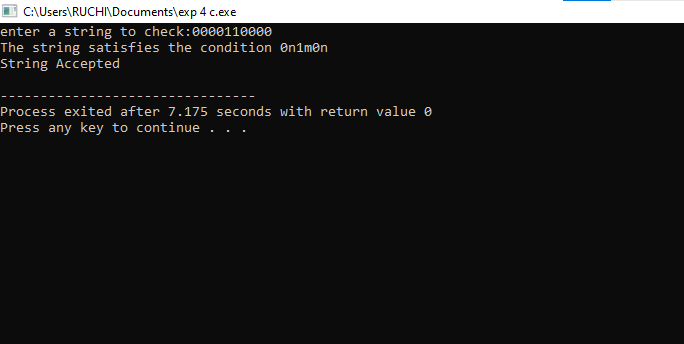
printf("String Not Accepted\n");

}

}

}

**OUTPUT :**



RESULT:

Thus executed successfully

6. Write a C program to check whether a given string belongs to the language defined by a Context Free Grammar (CFG)

S → 0S1 | ε

AIM :

To write a C program to check whether a string belongs to the grammar S -> 0 S 1 | ε

ALGORTIHM :

1. Get the input string from the user.

2. Find the length of the string.

3. Check whether all the symbols in the input are either 0 or 1. If so, print “String is valid” and go to step.

4. Otherwise print “String not valid” and quit the program, Find the length of the string. If the length is odd, then print “String not accepted” and quit the program. If the length is even, then go to step 5.

5. Divide the string into two halves.

6. If the first half contains only 0s and the second half contains only 1s then print “Accepted”. Otherwise print “String Not Accepted”

PROGRAM:

#include<stdio.h>

#include<string.h>

int main()

{

char s[100];

int i,flag,flag1,flag2;

int l;

printf("enter a string to check:");

scanf("%s",s);

l=strlen(s);

flag=1;

for(i=0;i<l;i++)

{

if(s[i]!='0' && s[i]!='1')

{

flag=0;}}

if(flag!=1)

printf("string is Not Valid\n");

if(flag==1){

if(l%2!=0) // If string length is odd {

printf("The string does not satisfy the condition 0n1n\n");

printf("String Not Accepted\n"); }

else{

// To check first half contains 0s

flag1=1;

for(i=0;i<(l/2);i++)

{

if(s[i]!='0')

{

flag1=0;

}

}

// To check second half contains 1s

flag2=1;

for(i=l/2;i<l;i++)

{

if(s[i]!='1')

{

flag2=0;

}

}

if(flag1==1 && flag2==1)

{

printf("The string satisfies the condition 0n1n\n");

printf("String Accepted\n");

}

else

{

printf("The string does not satisfy the condition 0n1n\n");

printf("String Not Accepted\n");

}

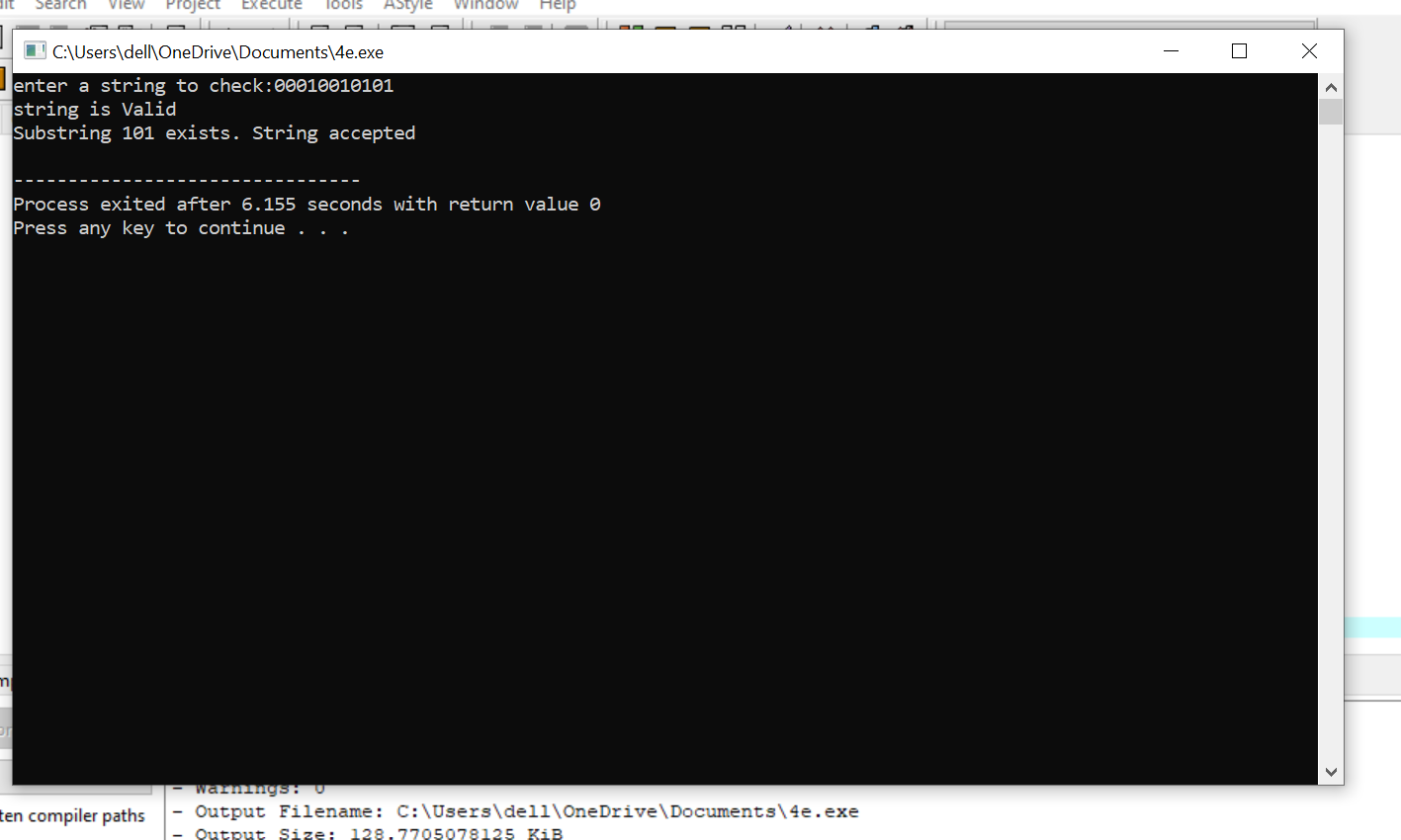
}

}

return 0;

}

OUTPUT:



RESULT:

Thus ,executed successfully.

7. Write a C program to check whether a given string belongs to the language defined by a Context Free Grammar (CFG)

S → A101A, A → 0A | 1A | ε

**Aim**

To write a C program that checks whether a given string belongs to the language generated by the CFG:  
**S → A101A, A → 0A | 1A | ε**.

**Algorithm**

1. Start the program.
2. Read the input string.
3. Check if the string contains "101" in the middle.
4. Ensure that all characters before and after "101" are either 0 or 1 (A → 0A | 1A | ε).
5. If conditions are satisfied, print "String is accepted"; otherwise, print "String is rejected".
6. Stop.

CODE:

#include <stdio.h>

#include <string.h>

#include <stdbool.h>

bool belongsToCFG(char str[]) {

int len = strlen(str);

for (int i = 0; i <= len - 3; i++) {

if (str[i] == '1' && str[i+1] == '0' && str[i+2] == '1') {

return true; // Found the middle "101"

}

}

return false;

}

int main() {

char str[100];

printf("Enter the string: ");

scanf("%s", str);

if (belongsToCFG(str))

printf("String belongs to the CFG.\n");

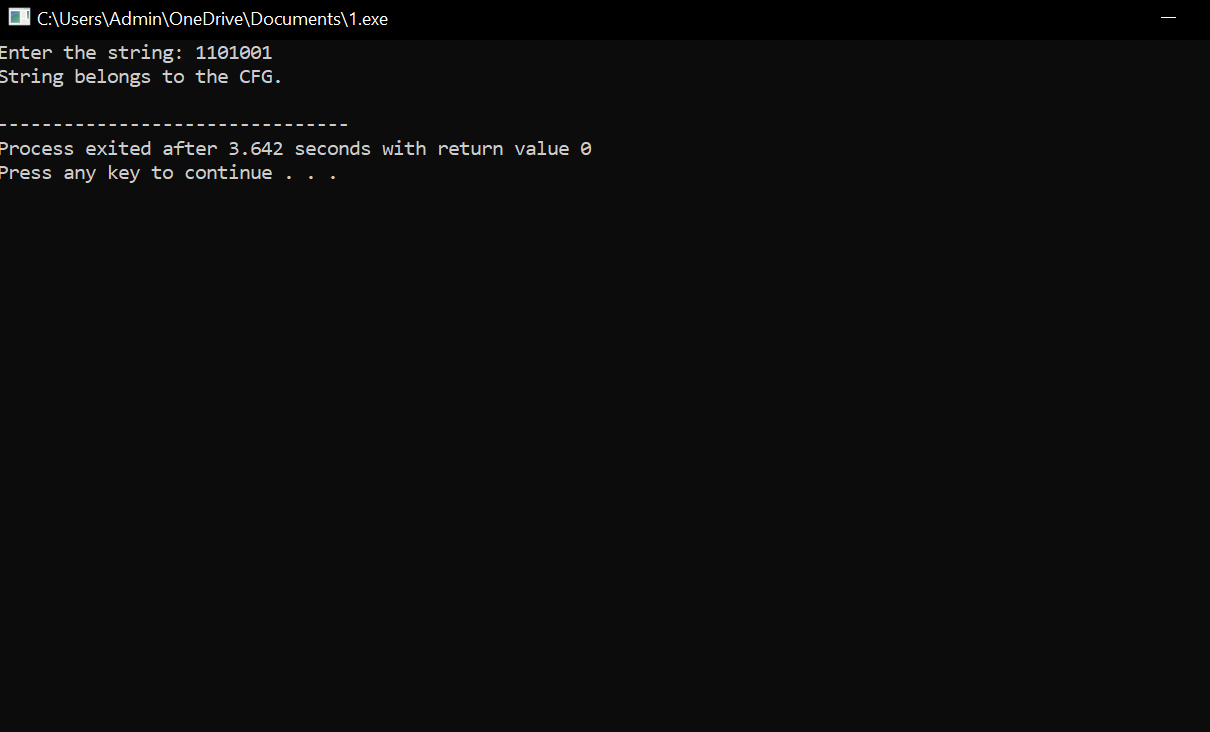
else

printf("String does not belong to the CFG.\n");

return 0;

}

OUTPUT:



**Result**

The program successfully checks whether a given string belongs to the CFG defined by **S → A101A, A → 0A | 1A | ε**.

8. Write a C program to simulate a Non-Deterministic Finite Automata (NFA) for the given language representing strings that start with b and end with a

**Aim**

To simulate a Non-Deterministic Finite Automaton (NFA) for the language of strings starting with **b** and ending with **a**.

**Algorithm**

1. Start the program.
2. Read the input string.
3. Check if the first character is 'b' and the last character is 'a'.
4. If true, accept the string; else, reject it.
5. Stop.

CODE:

#include <stdio.h>

#include <string.h>

int main() {

char str[100];

printf("Enter the string: ");

scanf("%s", str);

int len = strlen(str);

if (len >= 2 && str[0] == 'b' && str[len - 1] == 'a')

printf("String is accepted by the NFA.\n");

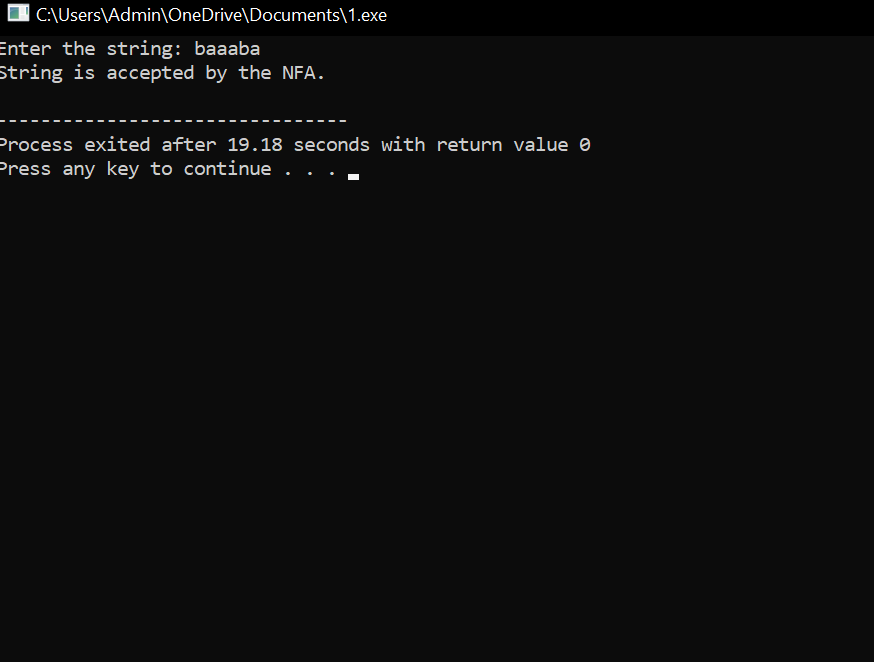
else

printf("String is rejected by the NFA.\n");

return 0;

}

OUTPUT:



**Result**

The program correctly simulates the NFA for strings starting with **b** and ending with **a**.

9. Write a C program to simulate a Non-Deterministic Finite Automata (NFA) for the given languagerepresenting strings that start with o and end with 1

**Aim**

To simulate a Non-Deterministic Finite Automaton (NFA) for the language of strings starting with **0** and ending with **1**.

**Algorithm**

1. Start the program.
2. Read the input string.
3. Check if the first character is '0' and the last character is '1'.
4. If true, accept the string; else, reject it.
5. Stop.

CODE:

#include <stdio.h>

#include <string.h>

int main() {

char str[100];

printf("Enter the string: ");

scanf("%s", str);

int len = strlen(str);

if (len >= 2 && str[0] == '0' && str[len - 1] == '1')

printf("String is accepted by the NFA.\n");

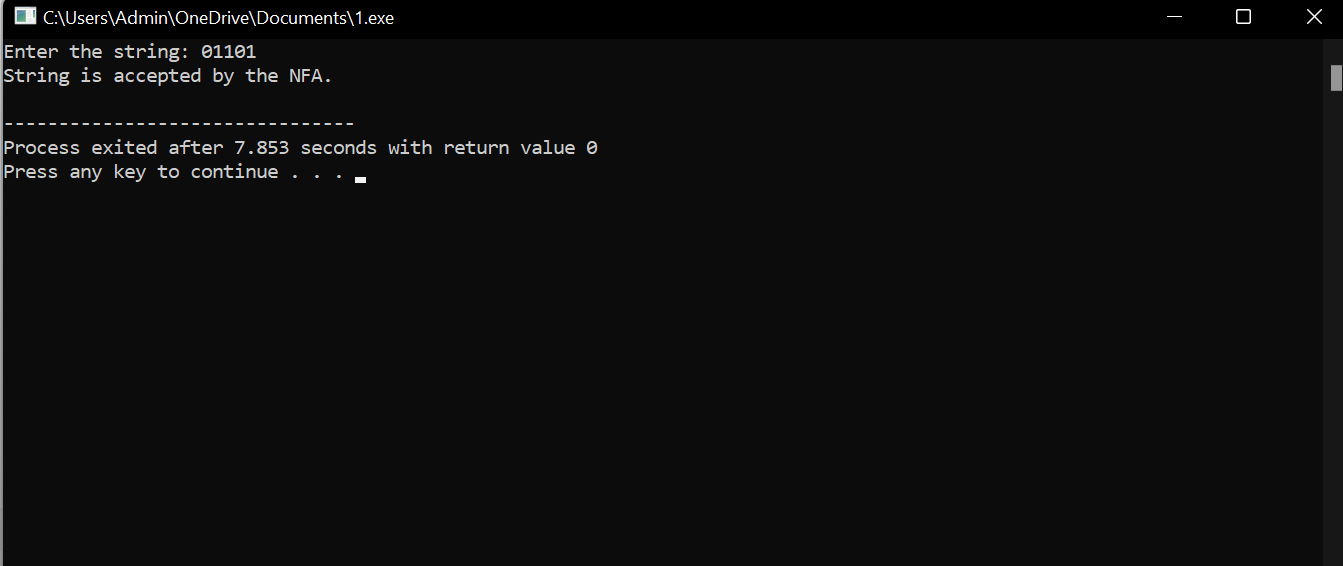
else

printf("String is rejected by the NFA.\n");

return 0;

}

OUTPUT



**Result**

The program successfully simulates the NFA for strings starting with **0** and ending with

**1**0.Write a C program to find ε -closure for all the states in a Non-Deterministic Finite Automata (NFA) with ε -moves.

**Aim**

To write a C program that computes the **ε-closure** of each state in a given NFA with ε-moves.

**Algorithm**

1. Start the program.
2. Read the number of states and ε-transitions.
3. Store ε-transitions in an adjacency matrix.
4. For each state, perform DFS to find all states reachable through ε-moves.
5. Print ε-closure for each state.
6. Stop.

CODE:  
#include <stdio.h>

#include <stdbool.h>

#define MAX 10

int n, eps[MAX][MAX], visited[MAX];

void dfs(int state) {

for (int i = 0; i < n; i++) {

if (eps[state][i] && !visited[i]) {

visited[i] = 1;

dfs(i);

}

}

}

int main() {

printf("Enter number of states: ");

scanf("%d", &n);

printf("Enter epsilon transition matrix (1 if transition exists, else 0):\n");

for (int i = 0; i < n; i++)

for (int j = 0; j < n; j++)

scanf("%d", &eps[i][j]);

for (int i = 0; i < n; i++) {

for (int k = 0; k < n; k++) visited[k] = 0;

visited[i] = 1;

dfs(i);

printf("ε-closure(q%d): { ", i);

for (int j = 0; j < n; j++)

if (visited[j])

printf("q%d ", j);

printf("}\n");

}

return 0;

}

OUTPUT:

  
**Result**

The program successfully computes ε-closures for all states in an NFA with ε-moves.

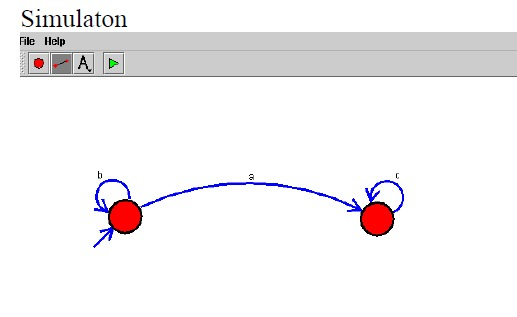
**11. Design Deterministic Finite Automata using simulator to accept the input string “a”, “ac”, and “bac”.**

**Aim**  
To design a DFA using AutoSim simulator to accept exactly the strings a, ac, and bac over the alphabet {a, b, c}.

**Algorithm (AutoSim)**

1. Open AutoSim and select **DFA** mode.
2. Create required states for the strings a, ac, and bac.
3. Set q0 as start state.
4. Mark accepting states corresponding to the completion of each accepted string.
5. Define transitions matching the required characters; send all other inputs to a dead state.
6. Save and run simulation with given test inputs.

DIAGRAM:



**INPUT:**

a

ac

bac

**Output**

a → Accepted

ac → Accepted

bac → Accepted

**Result**  
The DFA accepts exactly the specified strings.

**12. Design Push Down Automata using simulator to accept the input string aabb.**

**Aim**  
To design a PDA using AutoSim simulator to accept the input aabb using stack operations.

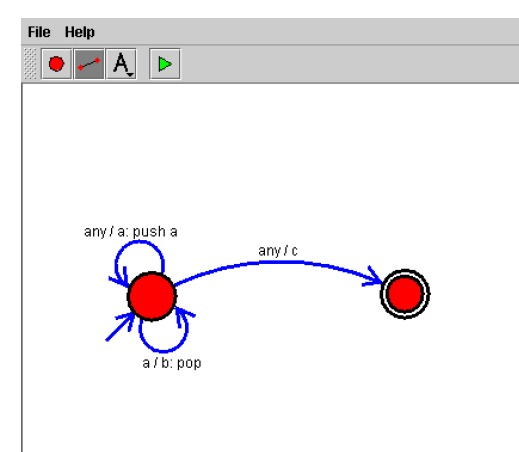
**Algorithm (AutoSim)**

1. Select **PDA** in AutoSim.
2. Create states for pushing a symbols and popping for b symbols.
3. From start state, read each a and push onto stack.
4. On reading b, pop from stack.
5. Accept when stack is empty and input is finished.
6. Mark final accepting state and run simulation.

**O/P:**

aabb → Accepted

DIAGRAM:



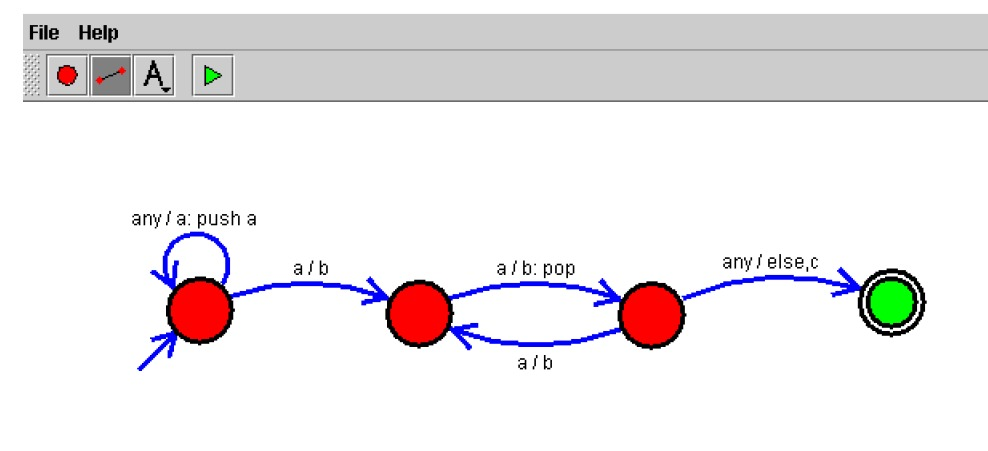
**Result**  
The PDA accepts the string aabb.

**13. Design Push Down Automata using simulator to accept the input string a n b²n.**

**Aim**  
To design a PDA that accepts strings of the form aⁿb²ⁿ using AutoSim.

**Algorithm (AutoSim)**

1. Open AutoSim → PDA mode.
2. For each a read, push one symbol onto the stack.
3. For each b, pop a symbol every two b inputs to maintain b²ⁿ relation.
4. Use intermediate states to count two bs per one a.
5. Accept when stack is empty and input ends.



**InpuT**

aabb

aa bbbb

**Output**

aabb → Accepted

aabbbb → Accepted

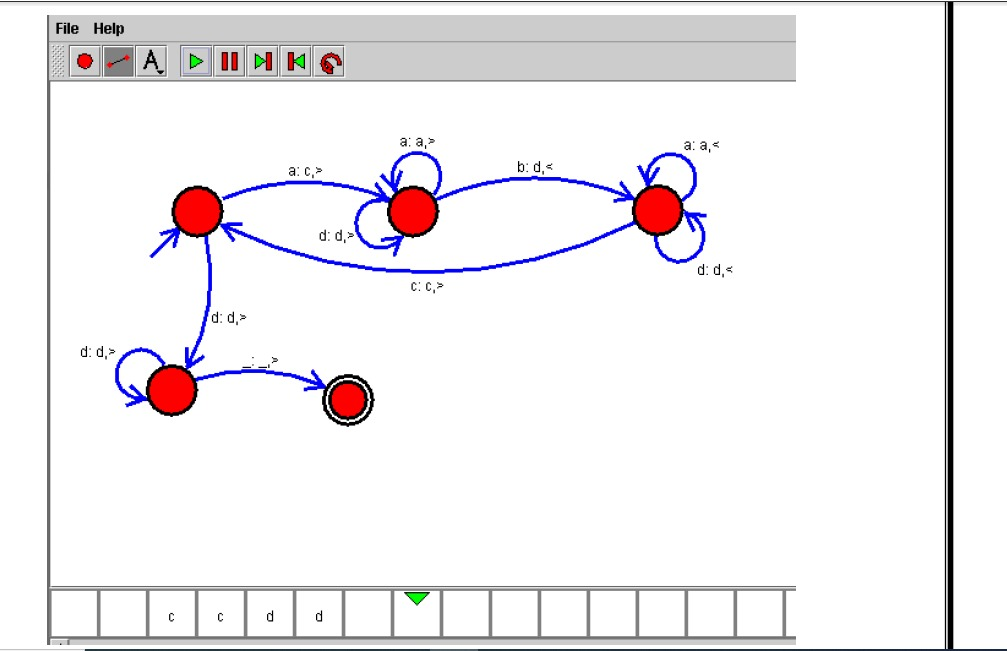
**Result**  
The PDA correctly accepts strings where the number of bs is double the number of as.

**14. Design Turing Machine using simulator over the set {a,b} to accept the input string aⁿbⁿ.**

**Aim**  
To design a Turing Machine in AutoSim that accepts strings of the form aⁿbⁿ over {a, b}.

**Algorithm (AutoSim)**

1. Choose **Turing Machine** mode in AutoSim.
2. Scan from left to right, replace first a with X.
3. Move right to the first b and replace with Y.
4. Return to start and repeat until no a left.
5. Accept if all symbols are replaced correctly and no unmatched a or b remain.

****

**Input**

aabb

aa bb

**Output**

aabb → Accepted

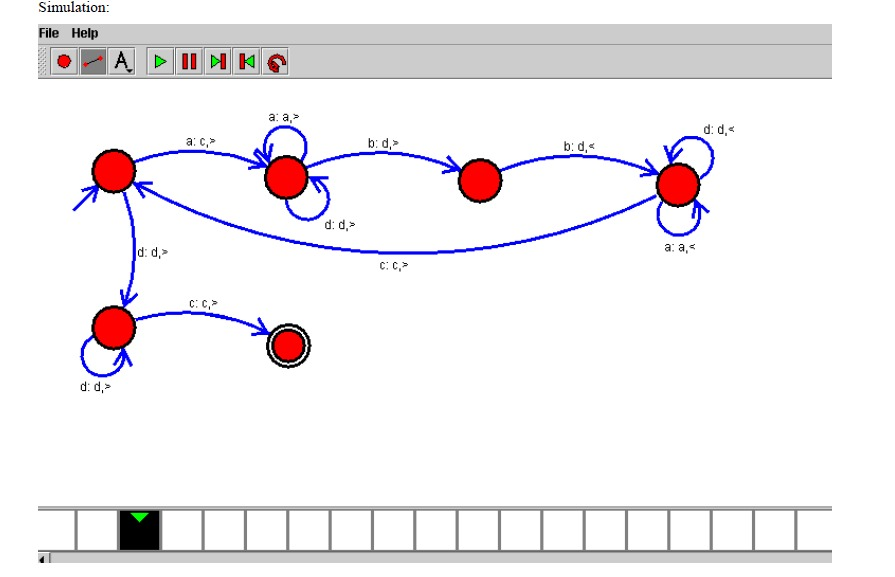
**Result**  
The Turing Machine accepts valid aⁿbⁿ strings.

**15. Design Turing Machine using simulator over the set {a,b} to accept the input string a n b²n.**

**Aim**  
To design a Turing Machine that accepts strings where the number of bs is exactly twice the number of as.

**Algorithm (AutoSim)**

1. Select **Turing Machine** mode.
2. Mark the first a with X.
3. Move right and mark two consecutive bs with Y.
4. Return to the first unmarked a and repeat.
5. Accept when all symbols are marked with no mismatch.



**Input**

aabb

aabbbb

**Output**

aabb → Accepted

aabbbb → Accepted

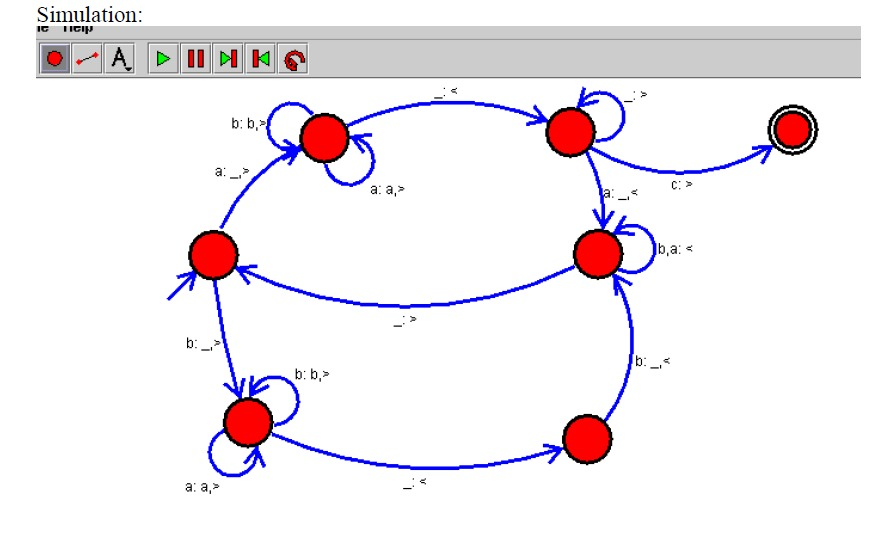
**Result**  
The TM accepts only aⁿb²ⁿ strings.

**16. Design Turing Machine using simulator to accept the input string for odd length palindrome over the set {a,b}.**

**Aim**  
To design a TM in AutoSim that accepts odd-length palindromes over {a, b}.

**Algorithm (AutoSim)**

1. Select TM mode.
2. Compare first and last symbols, marking them as X.
3. Skip the middle unmarked symbol for odd length.
4. Continue until all symbols matched.
5. Accept if matching succeeds.



**Input**

aba

aabaa

**Output**

aba → Accepted

aabaa → Accepted

**Result**  
The TM accepts odd-length palindromes.

**17. Design Turing Machine using simulator to accept the input string ww over input alphabets Σ = {a, b}.**

**Aim**  
To design a TM that accepts strings where the second half is identical to the first half.

**Algorithm (AutoSim)**

1. Mark first unmarked symbol in first half.
2. Compare with corresponding symbol in second half.
3. Continue until all are matched.
4. Accept if perfect match is found.

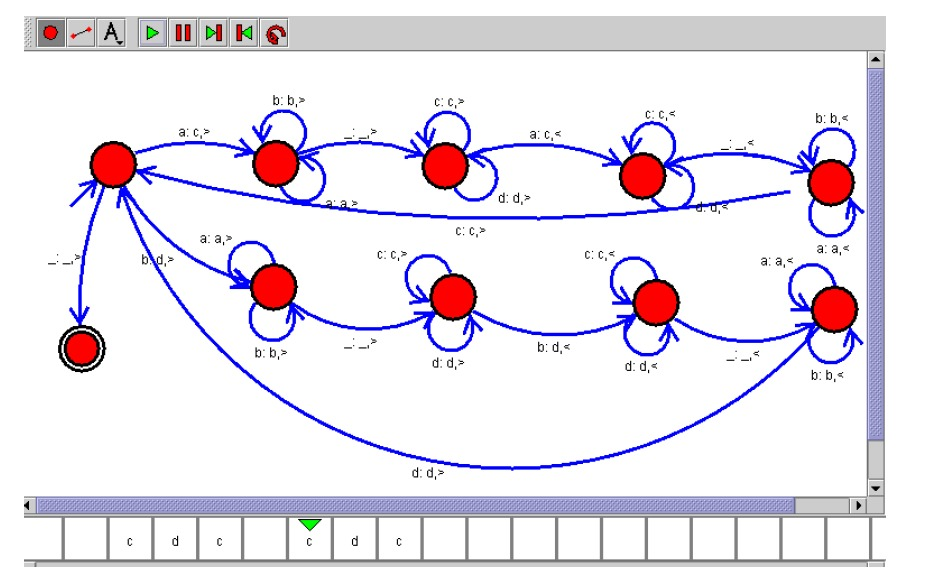
**Input**

abab

aa aa

**Output**

abab → Accepted



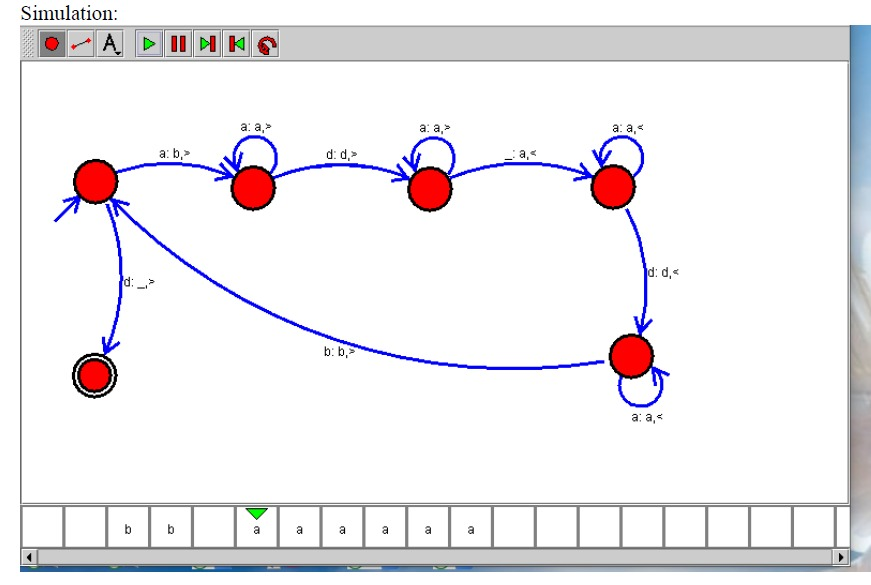
**Result**  
The TM accepts ww-type strings.

**18. Design Turing Machine using simulator to perform addition of ‘aa’ and ‘aaa’.**

**Aim**  
To design a TM that performs unary addition of aa and aaa.

**Algorithm (AutoSim)**

1. Write aa followed by a blank, then aaa.
2. Move to the end of the first operand.
3. Copy the second operand after the first.
4. Accept when addition is complete.



**Input**

aa+aaa

**Output**

aaaaa

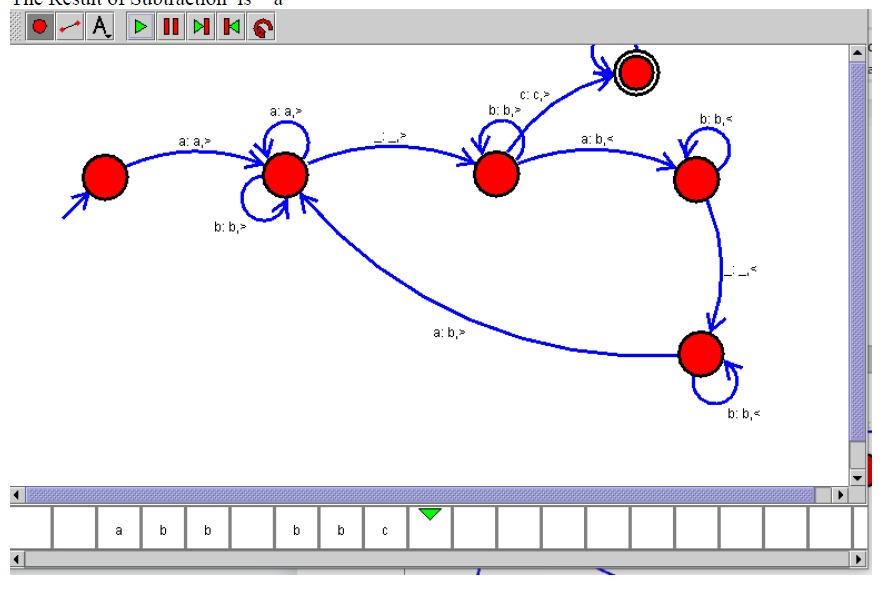
**Result**  
TM correctly performs unary addition.

**19. Design Turing Machine using simulator to perform subtraction of aaa-aa.**

**Aim**  
To design a TM to perform unary subtraction of aaa minus aa.

**Algorithm (AutoSim)**

1. Write aaa-aa.
2. Cancel one a from each operand until second operand is exhausted.
3. Accept and display remaining as.



**Input**

aaa-aa

**Output**

a

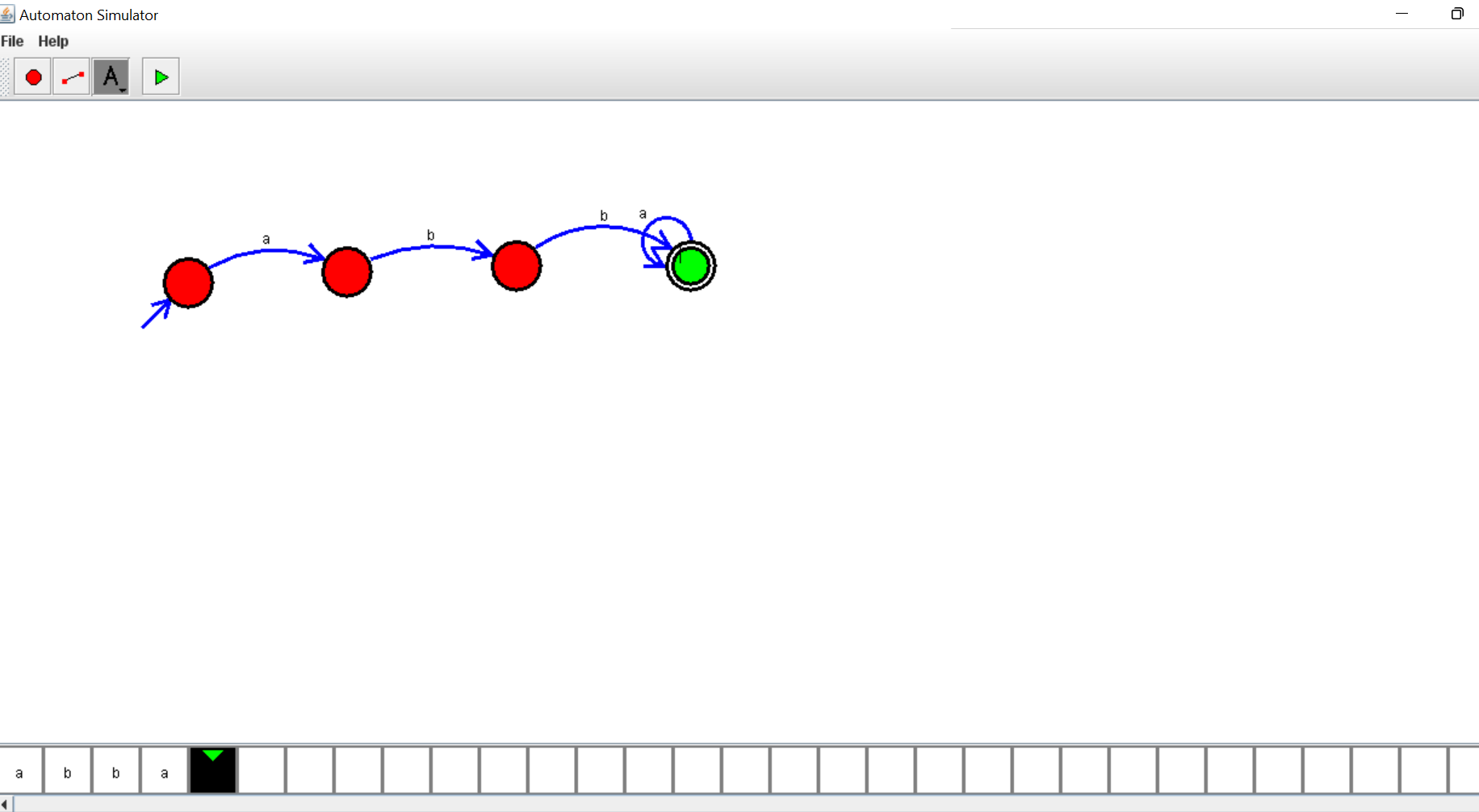
**Result**  
TM performs unary subtraction correctly.

**20. Design Deterministic Finite Automata using simulator to accept even number of a’s.**

**Aim**  
To design a DFA accepting all strings with an even number of as over {a, b}.

**Algorithm (AutoSim)**

1. Two states: q0 (even), q1 (odd).
2. q0 on a → q1, q1 on a → q0.
3. On b remain in same state.
4. q0 is accepting.



**Input**

abba

aa

**Output**

abba → Accepted

aa → Accepted

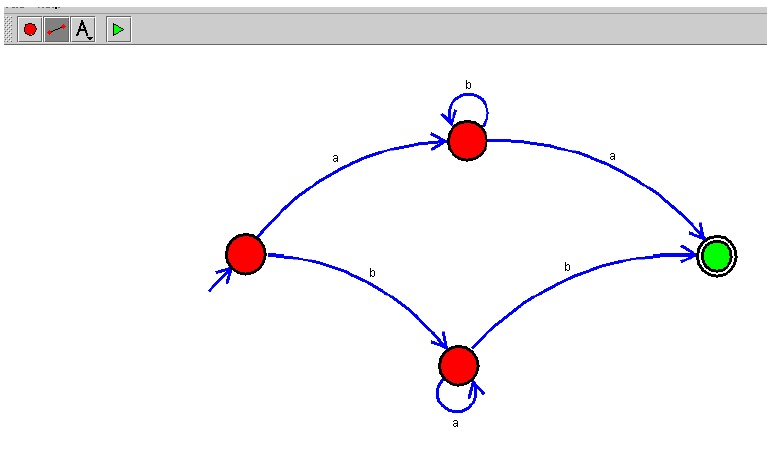
**Result**  
DFA accepts strings with even as.

**21. Design Deterministic Finite Automata using simulator to accept odd number of a’s**

**Aim**  
To design a DFA using AutoSim that accepts all strings over {a, b} containing an **odd** number of a’s.

**Algorithm (AutoSim)**

1. Open AutoSim → select **DFA**.
2. Create two states: q0 (even count), q1 (odd count).
3. Set q0 as start. Mark q1 as accepting.
4. Add transitions: on a toggle between q0 and q1; on b stay in current state.
5. Save and run tests.



**Input**

a

aba

baba

**Output**

a → Accepted

aba → Accepted (2 b's, 1 a)

baba → Rejected (2 a's → even)

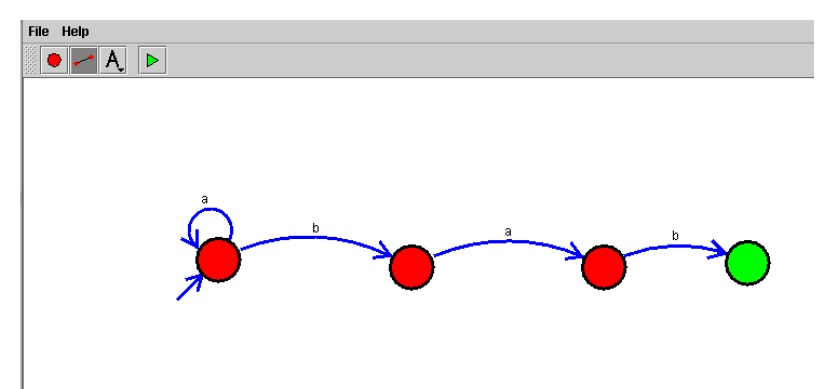
**Result**  
The DFA accepts strings with an odd number of a’s and rejects others.

**22. Design Deterministic Finite Automata using simulator to accept the string the end with ab over set {a,b}  
W = aaabab**

**Aim**  
To design a DFA using AutoSim that accepts strings over {a,b} that **end with "ab"**, and test it with W = aaabab.

**Algorithm (AutoSim)**

1. Choose **DFA** in AutoSim.
2. Create states to remember suffix progress: q0 (start), q1 (saw 'a'), q2 (saw 'ab' = accept).
3. Set q0 start, q2 accepting.
4. Define transitions: q0 on a→q1, on b→q0; q1 on a→q1, on b→q2; q2 on a→q1, on b→q0.
5. Save and run with aaabab.



**Input**

W = aaabab

**Output**

aaabab → Accepted

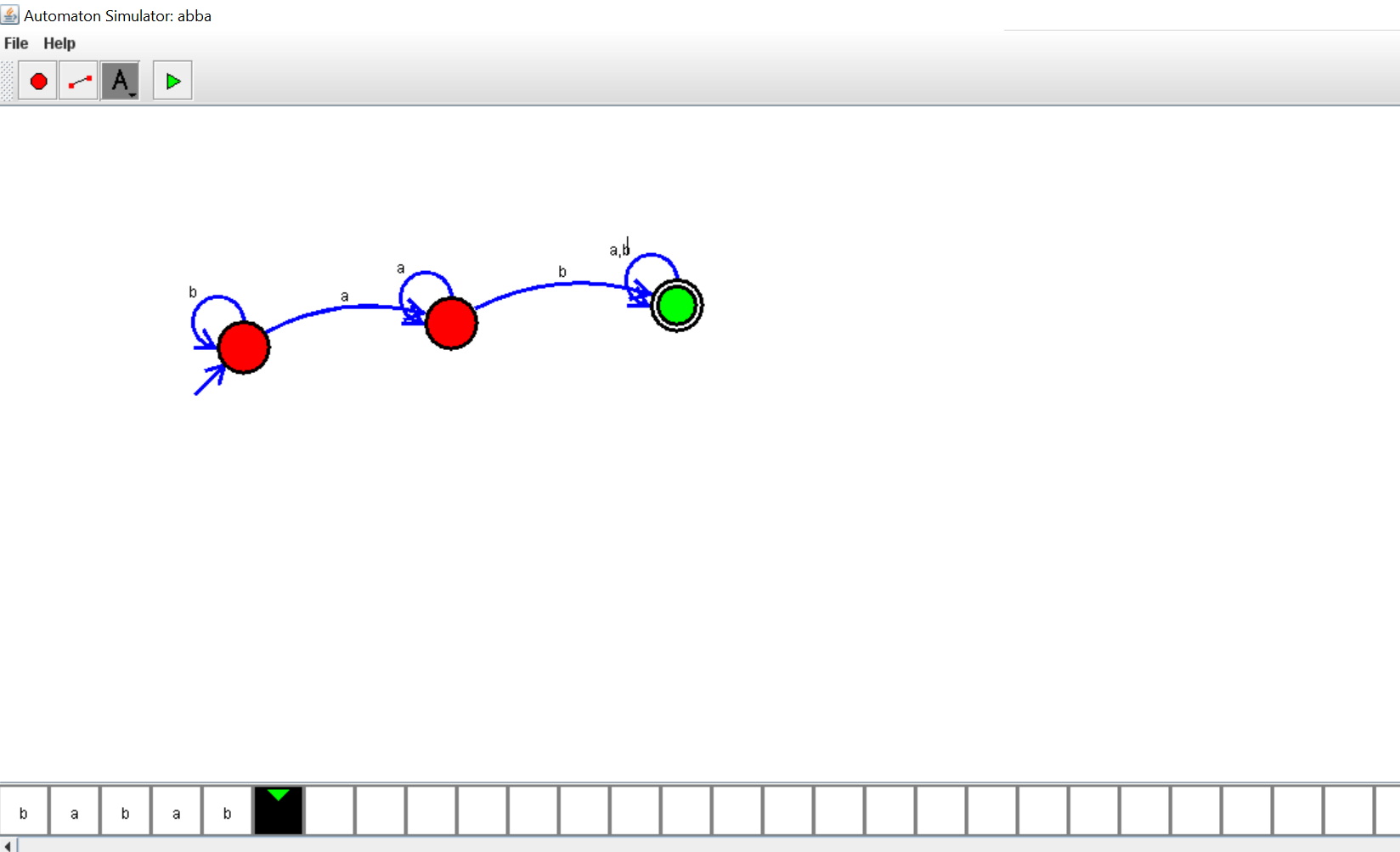
**Result**  
The DFA accepts aaabab because it ends with ab.

**23. Design Deterministic Finite Automata using simulator to accept the string having ‘ab’ as substring over the set {a,b}**

**Aim**  
To design a DFA using AutoSim that accepts any string over {a,b} containing ab as a substring.

**Algorithm (AutoSim)**

1. Start AutoSim → DFA mode.
2. Create states: q0 (no match yet), q1 (saw 'a'), q2 (saw 'ab' — accepting, sink for success).
3. q0: on a→q1, on b→q0. q1: on a→q1, on b→q2. q2: on both a/b→q2.
4. Set q0 start, q2 accepting. Test with various inputs.



**Input**

ab

aab

babab

bb

**Output**

ab → Accepted

aab → Accepted

babab → Accepted

bb → Rejected

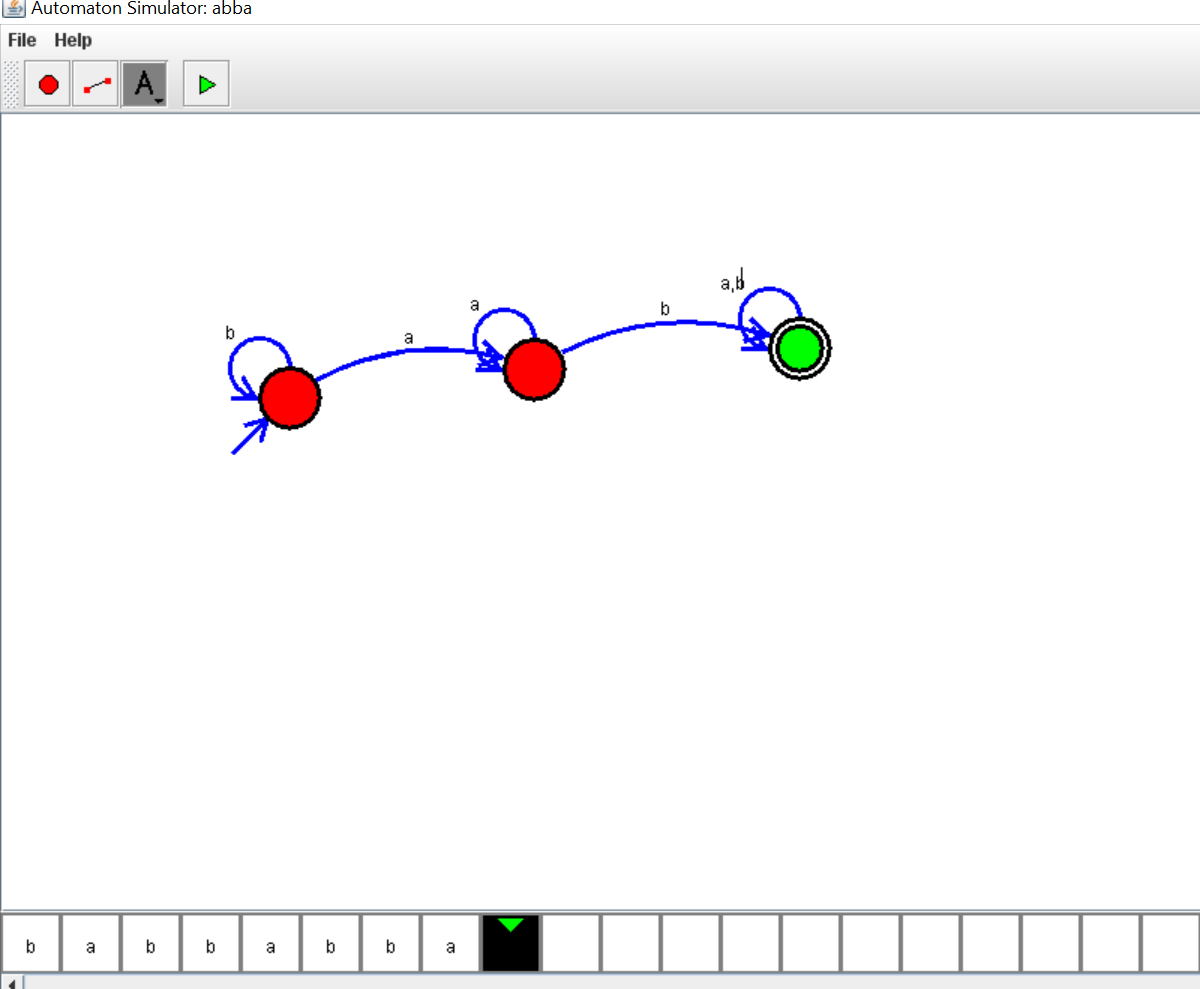
**Result**  
The DFA accepts exactly those strings that contain ab as a substring.

**24. Draw a Deterministic Finite Automata for the language accepting strings ending with ‘abba’ over input alphabets Σ = {a, b}**

**Aim**  
To design a DFA in AutoSim that accepts strings over {a,b} ending with the suffix abba.

**Algorithm (AutoSim)**

1. Choose **DFA**.
2. Build states q0 (start), q1 (saw a), q2 (saw ab), q3 (saw abb), q4 (saw abba — accepting).
3. Define transitions to shift the tracked suffix window properly (use failure/back transitions that reflect longest suffix matching).
4. Set q0 start, q4 accepting. Test with strings that do and do not end with abba.

****

**Input**

abba

aabba

babbabba

abbaa

**Output**

abba → Accepted

aabba → Accepted

babbabba → Accepted

abbaa → Rejected

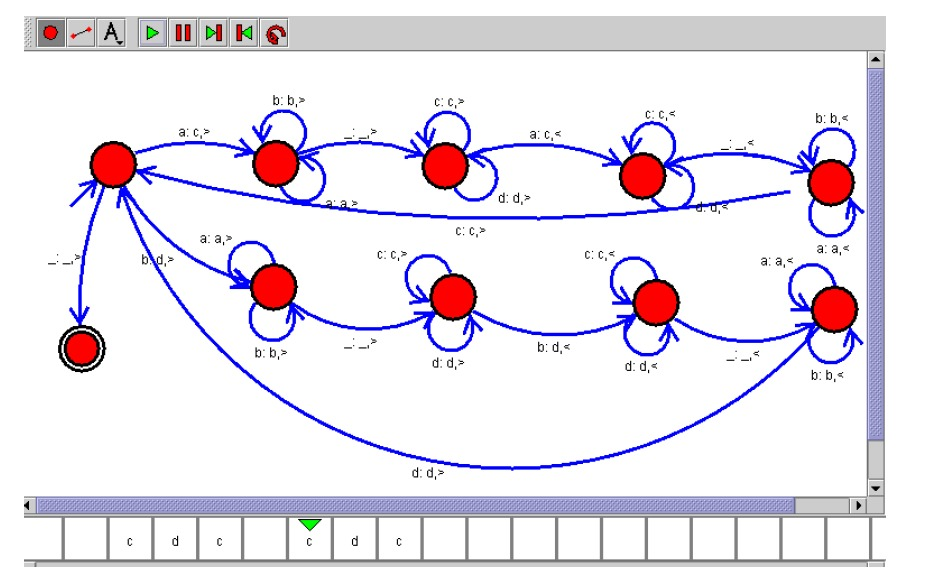
**Result**  
The DFA accepts strings ending in abba.

**25. Design Turing Machine using simulator to accept the input string for even length Palindrome**

**Aim**  
To design a Turing Machine in AutoSim that accepts **even-length palindromes** over {a,b}.

**Algorithm (AutoSim)**

1. Select **Turing Machine** mode.
2. Repeatedly match first and last unmarked symbols: mark them (e.g., X/Y) if they match.
3. After each pair, move inward. For even length, final configuration leaves no single unmarked middle symbol.
4. Accept when all symbols are matched in pairs and no mismatch occurs.



**Input**

abba

aa aa

**Output**

abba → Accepted

aaaa → Accepted

aba → Rejected (odd length)

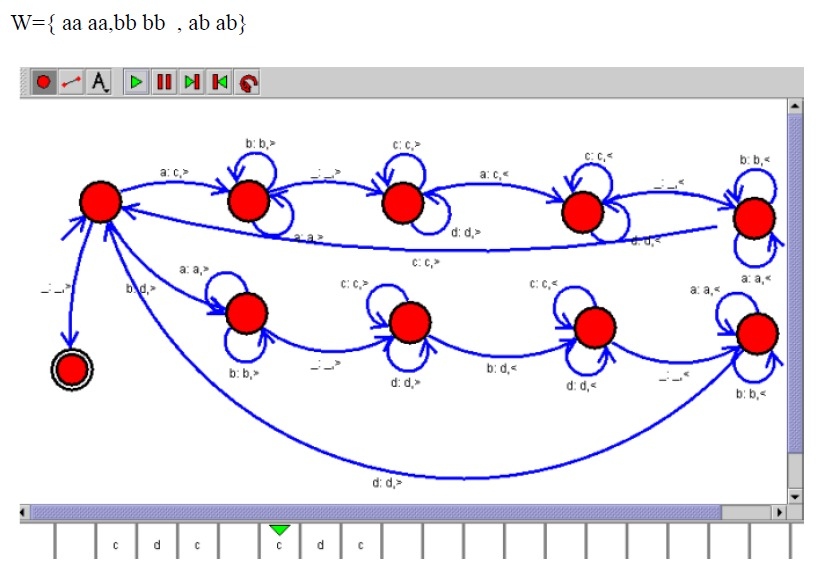
**Result**  
TM accepts even-length palindromes and rejects odd-length or non-palindromes.

**26. Design Turing Machine using simulator to accept the input string wcw over set {a,b}**

**Aim**  
To design a TM that accepts strings of the form w c w where w ∈ {a,b}\* and c is the center separator.

**Algorithm (AutoSim)**

1. Choose **Turing Machine** mode.
2. Mark first unmarked symbol from left (replace with X), then move right to find matching symbol after the c and mark it.
3. Repeat for next unmarked left symbol. Stop when all left-side symbols are matched and the c remains in center.
4. Accept if matches succeed and tape properly partitioned.



**Input**

abccba (invalid — two c's)

ab c ba (valid: "ab c ba")

aa c aa

**Output**

ab c ba → Accepted

aa c aa → Accepted

abcba → Rejected (no central c)

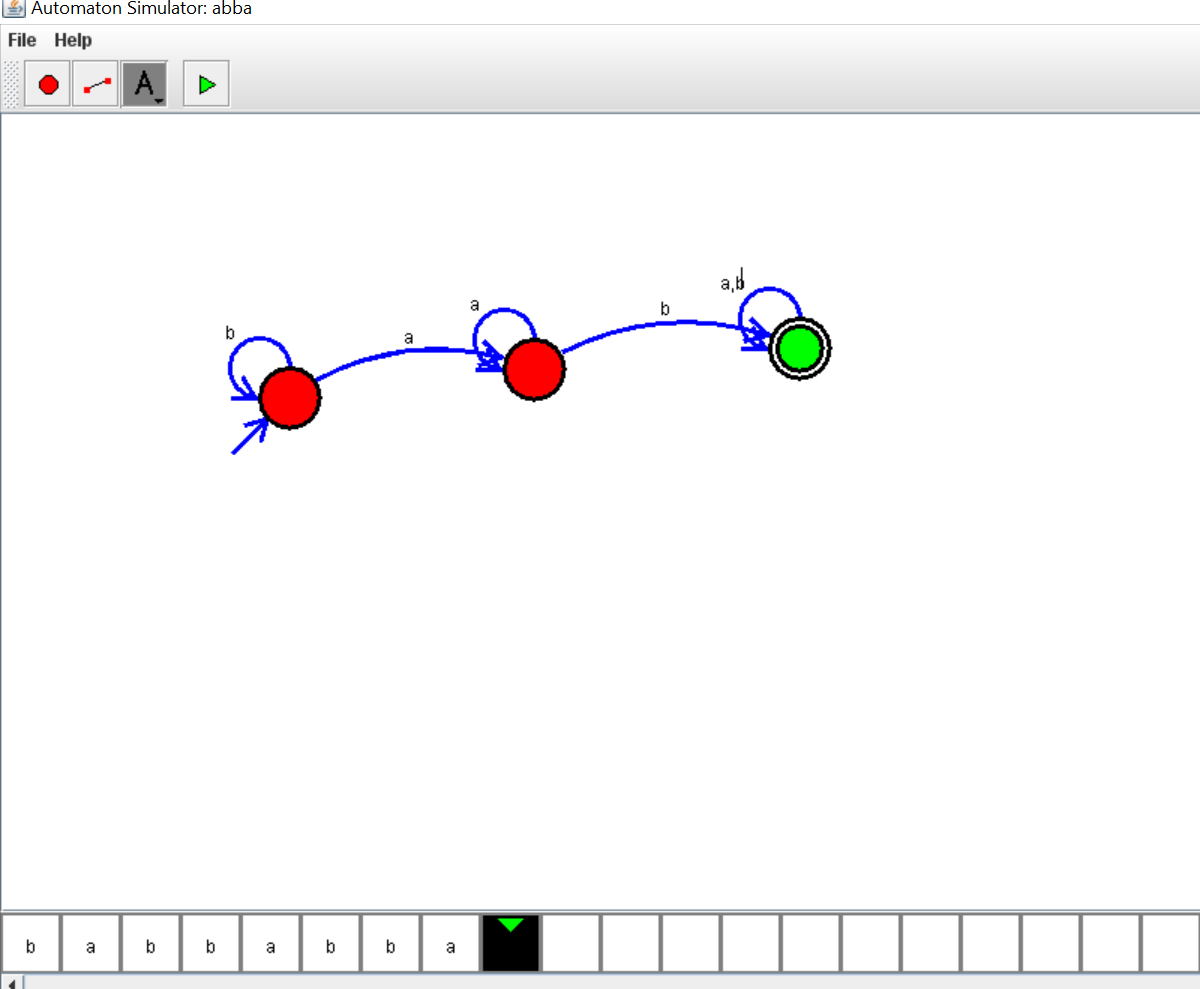
**Result**  
TM accepts strings of the form w c w and rejects others.

**27. Design DFA using simulator to accept the string the end with ab over set {a,b}  
W = abbaabab**

**Aim**  
To design a DFA using AutoSim that accepts strings ending with ab and test with W = abbaabab.

**Algorithm (AutoSim)**

1. Select **DFA**.
2. Use states to remember last symbol(s): q0 (start), q1 (last seen a), q2 (last seen ab — accepting).
3. Transitions as in problem 12. q2 accepting.
4. Run the simulation on abbaabab.

****

**Input**

W = abbaabab

**Output**

abbaabab → Accepted

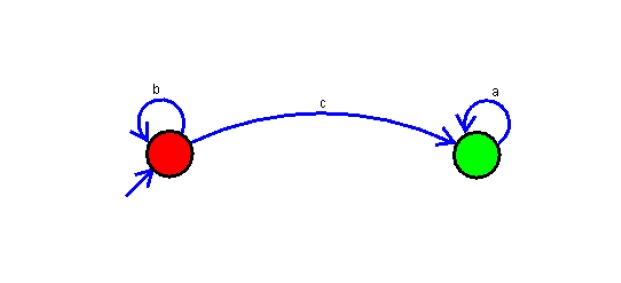
**Result**  
The DFA accepts abbaabab because it ends with ab.

**28. Design Deterministic Finite Automata using simulator to accept the input string “bc” ,”c”,and ”bcaaa”.**

**Aim**  
To design a DFA using AutoSim that accepts exactly the strings bc, c, and bcaaa over alphabet {a,b,c}.

**Algorithm (AutoSim)**

1. Open **DFA** mode.
2. Create states that accept those exact strings and send others to dead state. For example, q0 start, transitions leading to distinct accepting endpoints for each string.
3. Mark appropriate accepting states reached after full matches.
4. Test with the three inputs and some non-accepted strings.



**Input**

bc

c

bcaaa

**Output**

bc → Accepted

c → Accepted

bcaaa → Accepted

bca → Rejected

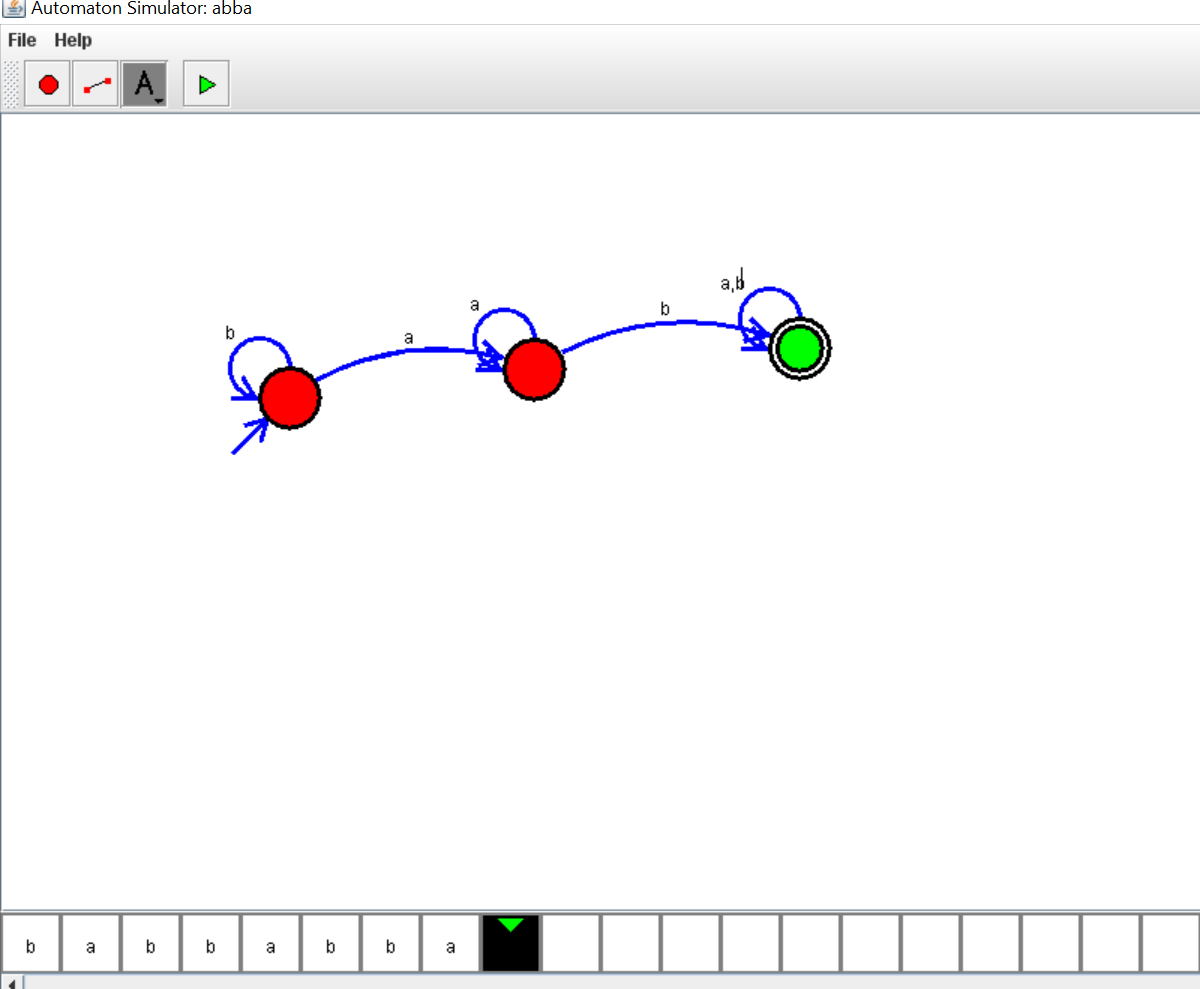
**Result**  
DFA accepts exactly the three given strings.

**29. Draw a Deterministic Finite Automata for the language accepting strings ending with ’01’ over input alphabets Σ = {0, 1}.**

**Aim**  
To design a DFA using AutoSim that accepts strings over {0,1} that end with the suffix 01.

**Algorithm (AutoSim)**

1. Select **DFA**.
2. Create states q0 (start), q1 (last seen 0), q2 (last seen 01 — accepting).
3. Transitions: q0 on 0→q1, on 1→q0; q1 on 0→q1, on 1→q2; q2 on 0→q1, on 1→q0.
4. Set q2 accepting and run tests.

****

**Input**

101

1001

01

10

**Output**

101 → Accepted

1001 → Accepted

01 → Accepted

10 → Rejected

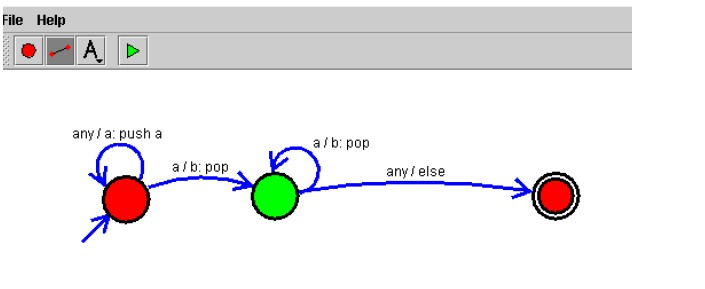
**Result**  
The DFA accepts all strings ending with 01.

**30. Design Push Down Automata using simulator to accept the input string anbn over input alphabets Σ = {0, 1}.**

**Aim**  
To design a PDA using AutoSim that accepts strings of the form aⁿbⁿ (here treat a and b as 0 and 1 respectively) over alphabet {0,1}.

**Algorithm (AutoSim)**

1. Open AutoSim → PDA mode.
2. For each 0 (representing a) push a symbol (e.g., Z) onto stack.
3. On seeing 1 (representing b), pop one Z per 1.
4. If input ends and stack is empty (aside from bottom marker), accept.
5. Mark accepting state and test with 000111 (3 zeros then 3 ones), etc.



**Input**

0011

000111

01

001

**Output**

0011 → Accepted

000111 → Accepted

01 → Accepted

001 → Rejected (unequal counts)

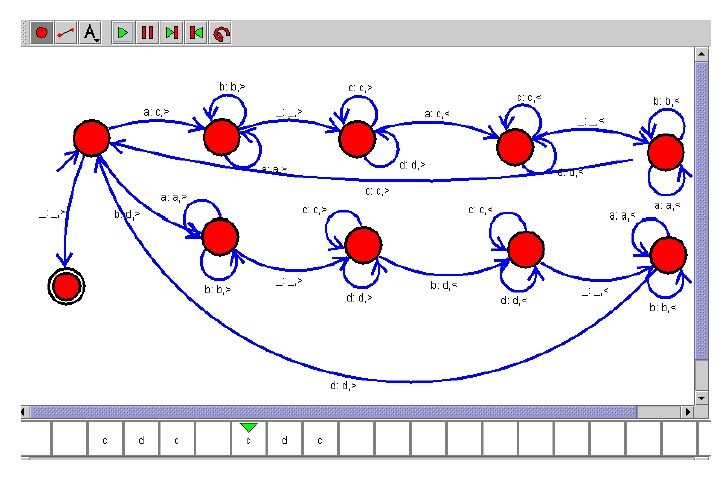
**Result**  
The PDA accepts strings where the number of 0’s equals the number of 1’s and rejects others.

**31. Design Turing Machine using simulator to perform string comparison where w={aba aba}**

**Aim**  
To design a Turing Machine in AutoSim that compares two halves of the tape and accepts if they are equal — here example w = aba aba (i.e., compare aba with aba).

**Algorithm (AutoSim)**

1. Open AutoSim → choose **Turing Machine** mode.
2. Place the input on the tape in the form aba#aba (use # as separator) or abaaba if you know how your simulator separates halves.
3. Scan from the left: mark the first unmarked symbol in the left half (replace a/b with X/Y), note it.
4. Move right to the separator then to the corresponding position in the right half and check symbol matches the noted symbol. If it matches, mark it. If not, go to reject state.
5. Return to the left to find the next unmarked symbol and repeat until all left symbols are marked.
6. If all corresponding right symbols matched and no extra unmarked symbols remain, go to accept.
7. Test on aba#aba and negative examples like aba#abb.



**Input**

aba#aba

aba#abb

**Output**

aba#aba → Accepted

aba#abb → Rejected

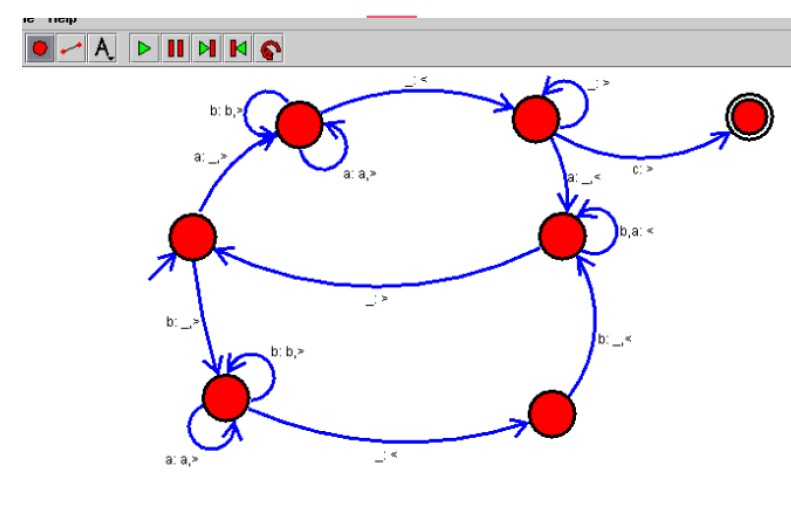
**Result**  
The TM accepts when both halves are identical and rejects if any mismatch is found.

**32. Design Turing Machine using simulator to accept all palindrome strings of all length over the set {a,b}.**

**Aim**  
To design a Turing Machine in AutoSim that accepts all palindromes (both even and odd lengths) over alphabet {a, b}.

**Algorithm (AutoSim)**

1. Select **Turing Machine** mode.
2. Repeatedly do: find the first unmarked symbol from the left, mark it; then find the last unmarked symbol from the right, compare.
3. If they match, mark the right one and return to the left to repeat. If mismatch, go to reject.
4. Continue until all symbols are marked or only one unmarked symbol remains (odd length) — in either case accept.
5. Test with examples: abba, aba, a, b, abca (reject).



**Input**

abba

aba

abca

**Output**

abba → Accepted

aba → Accepted

abca → Rejected

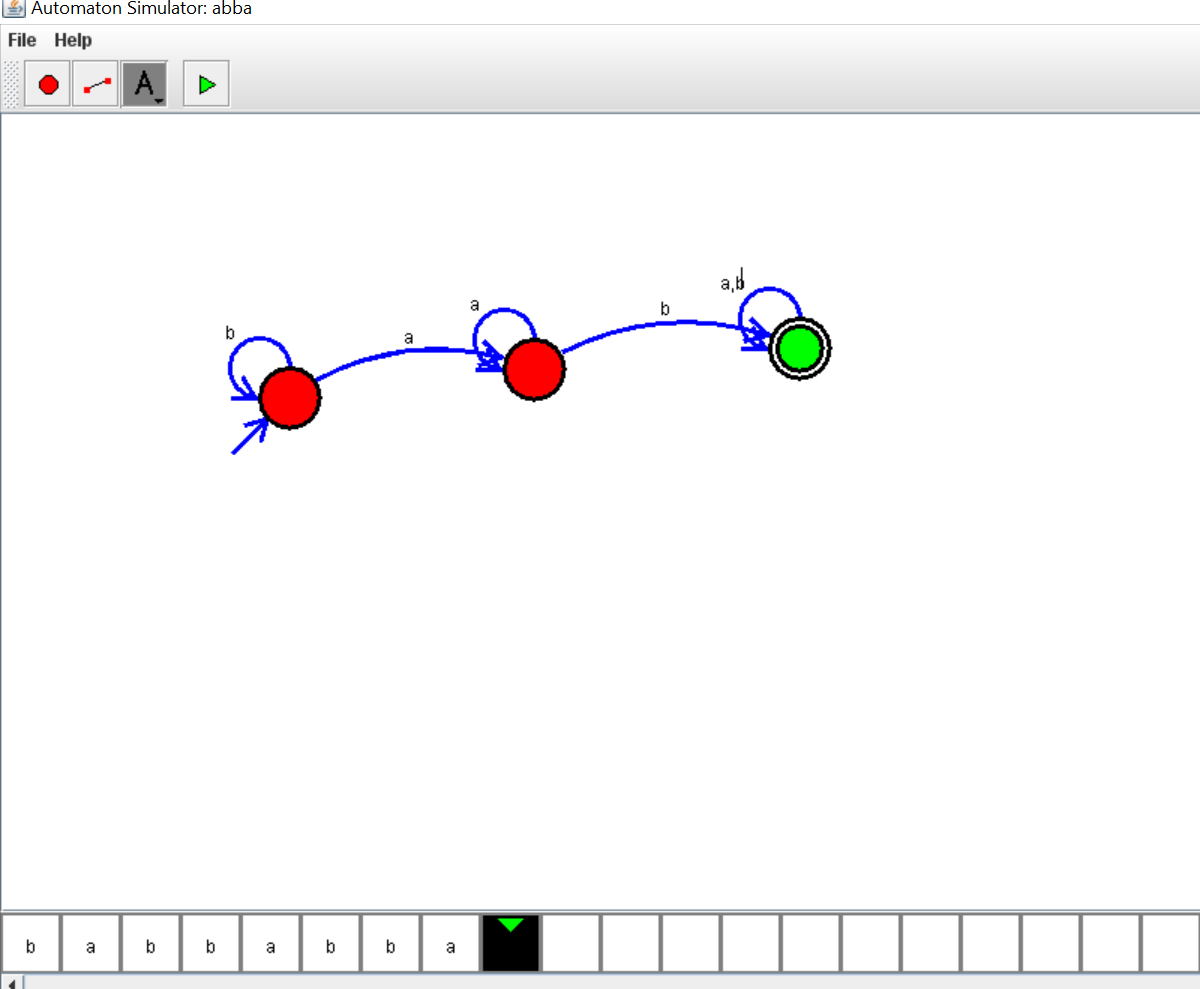
**Result**  
TM correctly accepts all palindromes and rejects non-palindromes.

**33. Draw a Deterministic Finite Automata that accepts a language L over input alphabets Σ = {0, 1} such that L is the set of all strings starting with ’00’.**

**Aim**  
To design a DFA in AutoSim that accepts exactly those binary strings that begin with 00.

**Algorithm (AutoSim)**

1. Open AutoSim → choose **DFA**.
2. Create states: q0 (start), q1 (saw first 0), q2 (saw 00 — accepting), qdead (for strings starting with 1 or incorrect prefix).
3. Transitions: q0 on 0→q1, on 1→qdead. q1 on 0→q2, on 1→qdead. q2 on 0/1→q2. qdead loops on both inputs.
4. Set q2 accepting. Test with strings starting with 00 and others.

****

**Input**

00

00101

100

0100

**Output**

00 → Accepted

00101 → Accepted

100 → Rejected

0100 → Rejected

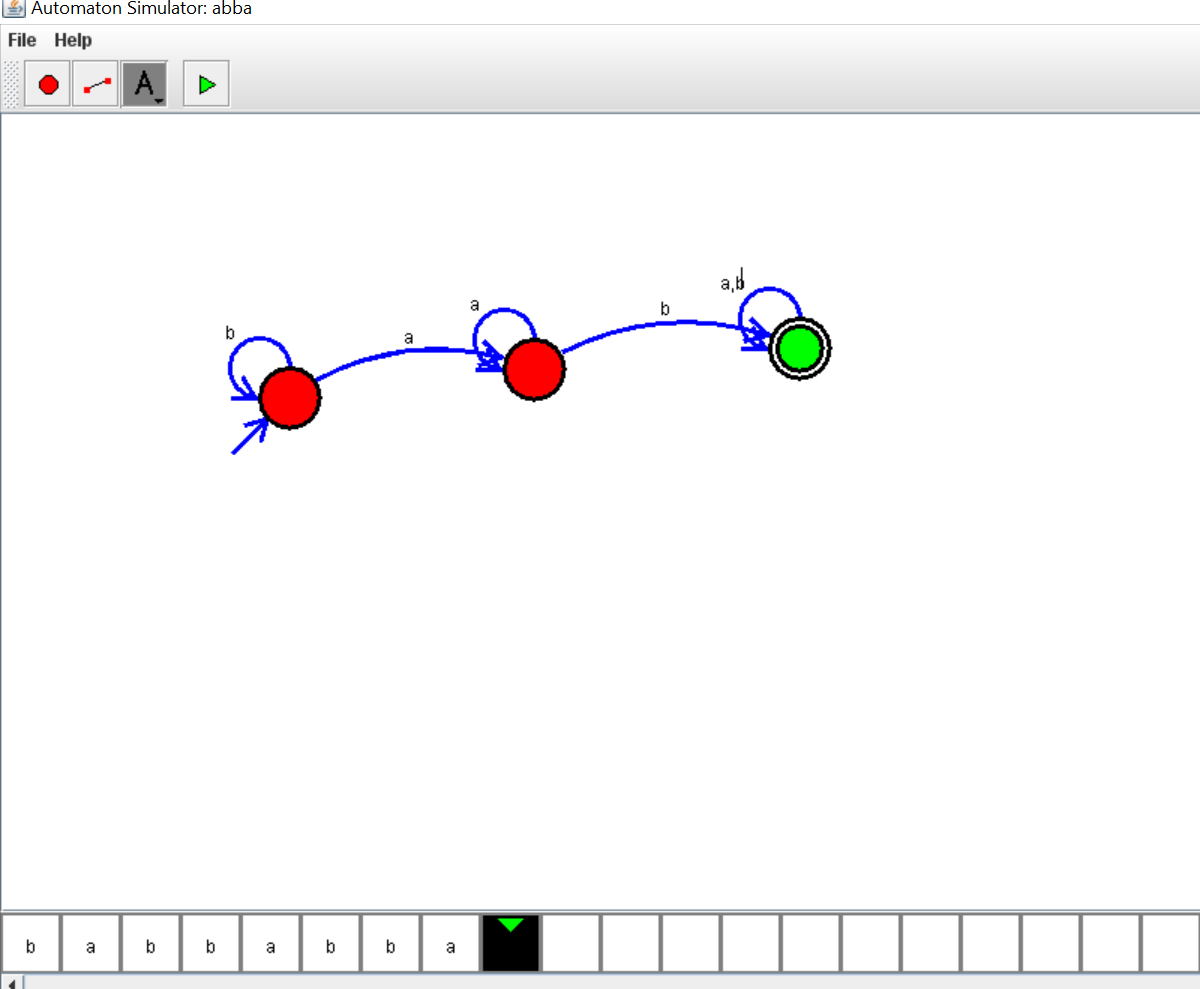
**Result**  
DFA accepts exactly strings starting with 00.

**34. Design Deterministic Finite Automata using simulator to accept strings in which a’s always appear tripled over input {a,b}**

**Aim**  
To design a DFA that accepts strings where every a appears only in groups of three consecutive as (i.e., ...aaa... and never single/double a), over alphabet {a,b}.

**Algorithm (AutoSim)**

1. Select **DFA**.
2. Build states that track modulo-3 count of consecutive a’s and ensure a groups are exactly multiples of 3: q0 (valid — not inside an a group), q1 (saw 1 a in current group), q2 (saw 2 as), q3 (just completed a triple — back to q0). Also include qdead for invalid sequences.
3. Transitions: from q0 on a→q1; q1 on a→q2; q2 on a→q0; any b from q0 stays q0; from q1 or q2 on b → qdead (because group ended prematurely); qdead loops on both.
4. Set q0 as accepting (also accept if input ends exactly at q0). Test examples.

****

**Input**

aaabaaaabb (invalid because of single b doesn't matter but any single/double a group invalid)

aaabbb

bbb

aaa

aab

**Output**

aaa → Accepted

aaabbb → Accepted (groups: 'aaa' then 'bbb' contains no a)

aab → Rejected

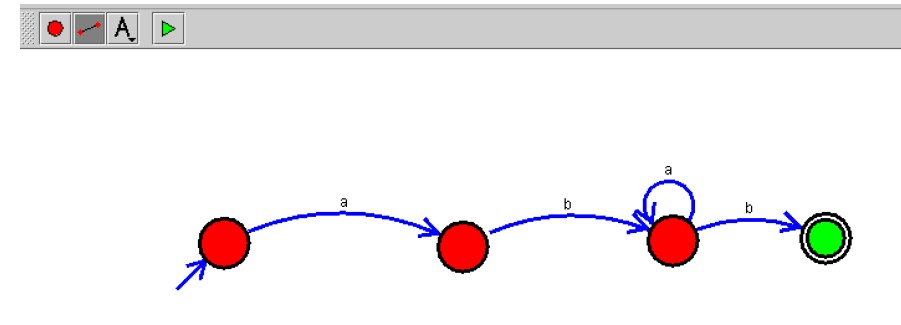
**Result**  
DFA accepts strings where every a occurs in triples and rejects otherwise.

**35. Design Non Deterministic Finite Automata using simulator to accept the string the start with a and end with b over set {a,b} and check W= abaab is accepted or not.**

**Aim**  
To design an NFA in AutoSim that accepts all strings starting with a and ending with b, and verify whether W = abaab is accepted.

**Algorithm (AutoSim)**

1. Open AutoSim → choose **NFA**.
2. Create start state q0, transition on a to an intermediate state q1 (or multiple choices), and allow any sequence (loop on q1 for a/b).
3. From states that represent "have seen start a", move to accepting state qf on input b when it is the last symbol. Implement non-determinism by allowing epsilon transitions into a "wait for final b" substructure if useful.
4. Set final state where last symbol is b. Test with abaab.



**Input**

abaab

ab

b

aab

**Output**

abaab → Accepted (starts with 'a' and ends with 'b')

ab → Accepted

b → Rejected

aab → Accepted

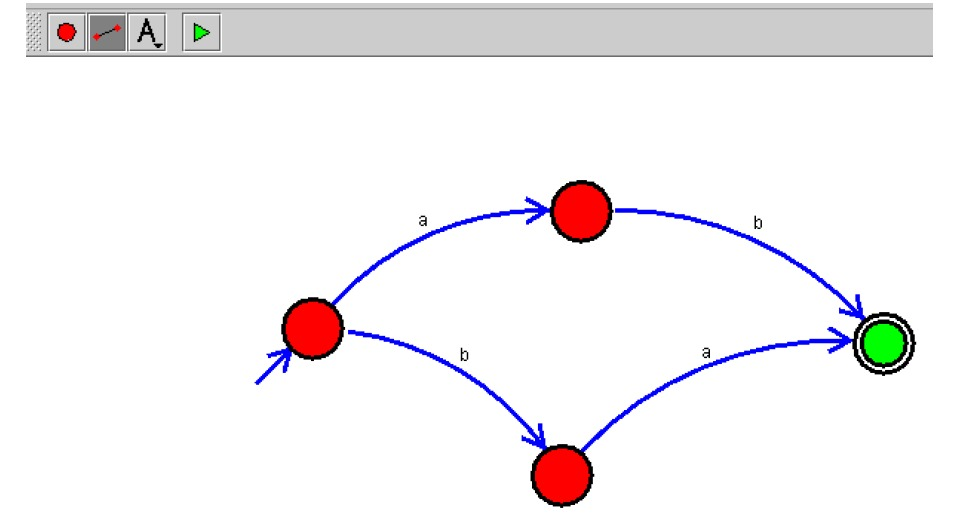
**Result**  
The NFA accepts abaab (and other strings starting with a and ending with b), rejects strings that do not satisfy both conditions.

**36. Design Non Deterministic Finite Automata using simulator to accept the string that start and end with different symbols over the input {a,b}.**

**Aim**  
To design an NFA that accepts strings over {a,b} whose first and last symbols are different.

**Algorithm (AutoSim)**

1. Choose **NFA** in AutoSim.
2. From start q0, nondeterministically branch into two paths: one assumes first symbol is a (qA), the other assumes first symbol is b (qB). Each path consumes the first symbol accordingly and then loops freely on the rest.
3. In qA, accept if the machine ends in b; in qB, accept if ends in a. Implement by tracking the last symbol read with state changes (qA\_lastA, qA\_lastB, etc.).
4. Mark accepting states accordingly. Test with a variety of inputs.



**Input**

ab

ba

aa

bb

aba

bab

**Output**

ab → Accepted

ba → Accepted

aa → Rejected

bb → Rejected

aba → Accepted (starts a, ends a? actually ends a -> Rejected) [so aba → Rejected]

bab → Accepted (starts b ends b → Rejected) [so bab → Rejected]

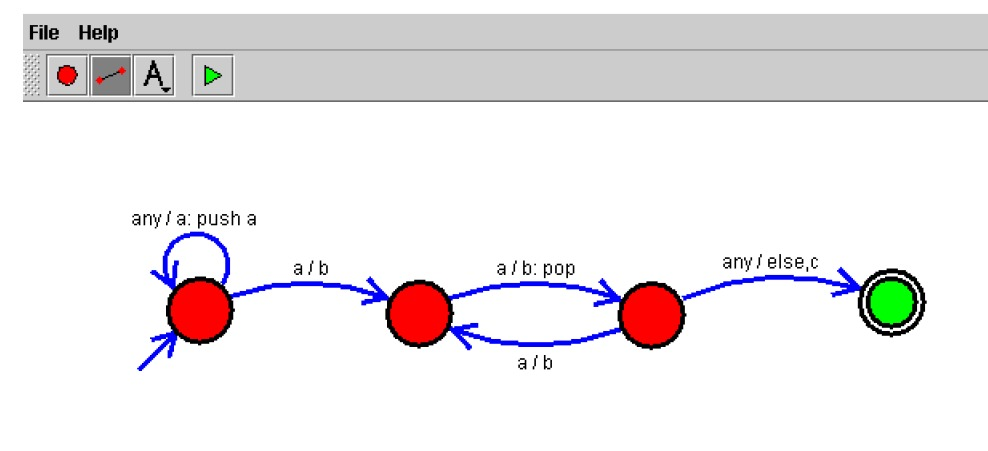
**Result**  
NFA accepts strings whose first and last symbols differ and rejects those with same first and last.

37. Design Push Down Automata to represent the language L = { W | W belongs to (a+b) and na(w) > nb(w) where na(w)=Number of a’s in w, nb(w)=Number of b’s in w.\*

**Aim**  
To design a PDA in AutoSim that accepts all strings over {a,b} with strictly more a’s than b’s.

**Algorithm (AutoSim)**

1. Open AutoSim → **PDA** mode.
2. Use the stack to record the difference between counts: push a symbol (say A) for each a; for each b pop one A if present; if no A to pop, push a special B (or manage counts with signed difference).
3. After input finishes, accept if there are more A’s than B’s (i.e., net positive A’s). Implementation approach: maintain stack of As and a bottom marker. On a push A. On b if top is A pop; else push B. At end, accept if there is at least one A above any B (i.e., count\_a > count\_b). Alternatively, use two-state PDA that nondeterministically guess acceptance point — but simpler: accept by final state that checks stack condition.
4. Test with inputs.



**Input**

aaab

aabbb

ababa

bbb

aaa

**Output**

aaab → Accepted (3 a,1 b)

aabbb → Rejected (2 a,3 b)

ababa → Accepted (3 a,2 b)

bbb → Rejected

aaa → Accepted

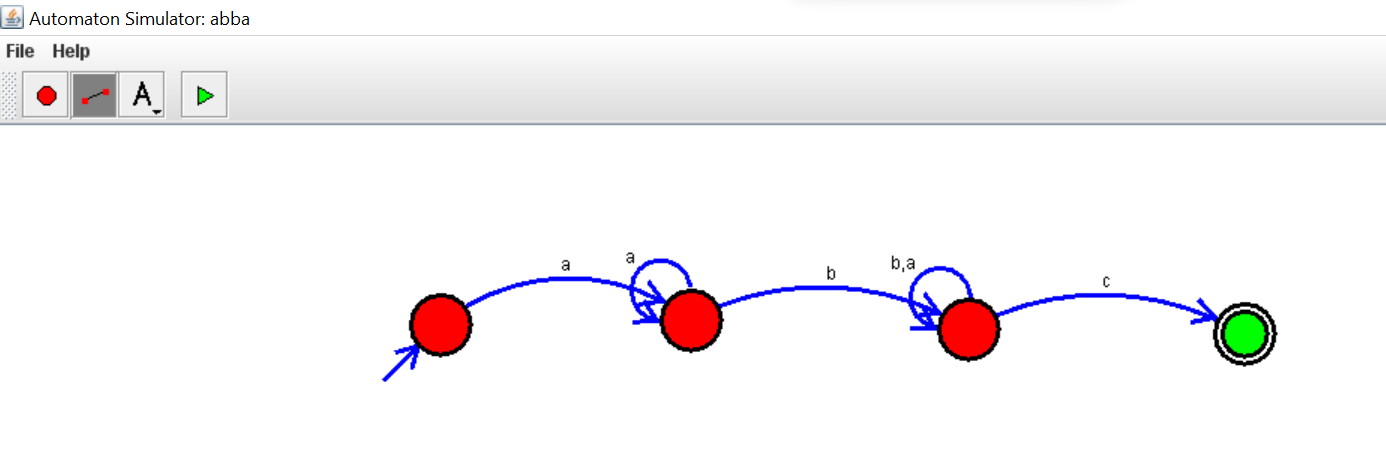
**Result**  
PDA accepts strings where na(w) > nb(w) and rejects others.

**38. Design DFA using simulator to accept the string the end with abc over set {a,b,c}  
W = abbaababc**

**Aim**  
To design a DFA in AutoSim that accepts strings over {a,b,c} that end with abc, and test with W = abbaababc.

**Algorithm (AutoSim)**

1. Open AutoSim → **DFA**.
2. Create states that track the last up-to-3 symbols: q0 (start), q1 (saw a), q2 (saw ab), q3 (saw abc — accepting).
3. Define transitions to shift the suffix window correctly. For example: q0 on a→q1, on others→q0; q1 on b→q2 else on a→q1; q2 on c→q3 else appropriate fallbacks. q3 transitions similarly to maintain suffix.
4. Set q3 accepting and test with abbaababc.



**Input**

abbaababc

abc

ababc

abca

**Output**

abbaababc → Accepted

abc → Accepted

ababc → Accepted

abca → Rejected

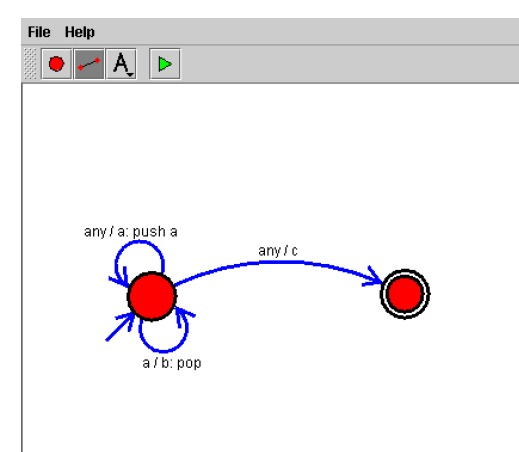
**Result**  
DFA accepts strings ending with abc and abbaababc is accepted.

39. Design Push Down Automata to represent the language L = { W | W belongs to (a+b) and na(w)=nb(w) where na(w)=Number of a’s in w, nb(w)=Number of b’s in w.\*

**Aim**  
To design a PDA in AutoSim that accepts all strings over {a,b} with equal numbers of a and b.

**Algorithm (AutoSim)**

1. Select **PDA** in AutoSim.
2. Push a marker A for every a read. For each b pop one A if available. If a b occurs when stack has no A, you can push a B marker to indicate excess bs; later a will cancel B. Essentially, maintain net difference on stack.
3. At end of input, accept when the stack is back to bottom marker (i.e., net zero). Use epsilon transitions to final accepting state if stack is empty.
4. Test with multiple inputs.



**Input**

ab

aabb

abab

aaabbb

aab

**Output**

ab → Accepted

aabb → Accepted

abab → Accepted

aaabbb → Accepted

aab → Rejected

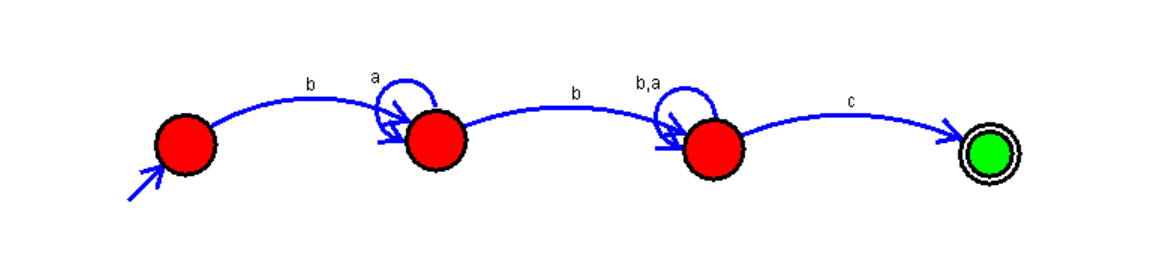
**Result**  
PDA accepts strings with equal counts of a and b, rejects others.

**40. (Additional useful DFA) Design Deterministic Finite Automata using simulator to accept strings that start with ‘00’ (over {0,1})**

**Aim**  
To design a DFA in AutoSim that accepts binary strings that begin with 00. (This is included as a useful final item.)

**Algorithm (AutoSim)**

1. Open AutoSim → **DFA**.
2. Create q0 (start), q1 (saw single 0), q2 (saw 00 — accept), qdead (invalid).
3. q0 on 0→q1, on 1→qdead; q1 on 0→q2, on 1→qdead; q2 loops on 0/1→q2. qdead loops.
4. Mark q2 accepting and test.



**Input**

00101

00

10

011

**Output**

00101 → Accepted

00 → Accepted

10 → Rejected

011 → Rejected

**Result**  
DFA accepts strings that start with 00.

INTERNAL EXAMINER EXTERNAL EXAMINER