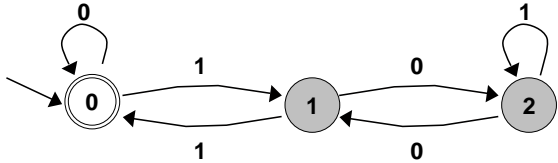


# Lecture T1: Pattern Matching



# Introduction to Theoretical CS

Two fundamental questions.

- What can a computer do?
- What can a computer do with limited resources?

General approach.

- Don't talk about specific machines or problems.
- Consider minimal abstract machines.
- Consider general classes of problems.

# Why Learn Theory

In theory . . .

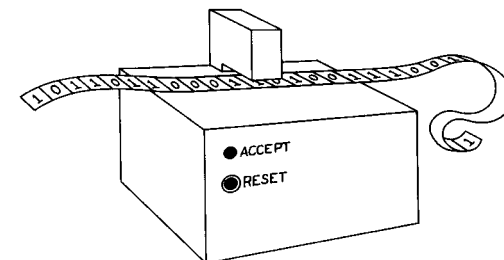
- Deeper understanding of what is a computer and computing.
- Foundation of all modern computers.
- Pure science.
- Philosophical implications.

In practice . . .

- Web search: theory of pattern matching.
- Sequential circuit: theory of finite state automata.
- Compilers: theory of context free grammar.
- Cryptography: theory of complexity.
- Data compression: theory of information.

# Finite State Automaton

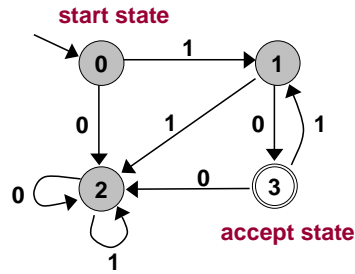
Simple machine with N states.



# Finite State Automata

Simple machine with N states.

- Start in state 0.
- Read an input bit.
- Move to new state
  - depends on input bit and current state
- Stop when last bit read.
  - 'yes' if end in accept state(s)
  - 'no' otherwise



State	0	1
0	2	1
1	3	2
2	2	2
3	2	1

'Yes' also called *accepted* or *recognized* inputs from a language.



# C Code for FSA

```

fsa3.c
#include <stdio.h>
#define STATES      4
#define START_STATE 0
#define ACCEPT_STATE 3

int main(void) {
    int i, state = START_STATE;
    int transition[STATES][2] =
        { {2, 1}, {3, 2}, {2, 2}, {2, 1} };

    while (scanf("%ld", &i) != EOF)
        state = transition[state][i];

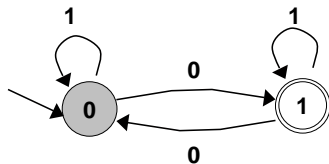
    if (state == ACCEPT_STATE)
        printf("Yes.\n");
    else
        printf("No.\n");
    return 0;
}
    
```

use 2D array

State	0	1
0	2	1
1	3	2
2	2	2
3	2	1

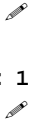
# A Second Example

Consider the following two state FSA.



What bit strings does it accept?

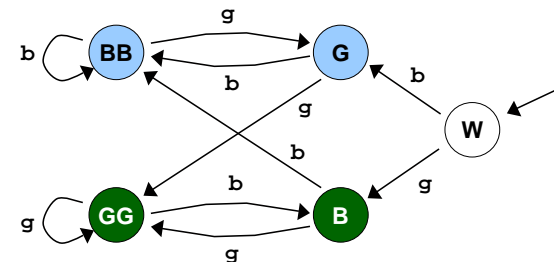
- Yes: 0, 11110, 00000, 100100111011
- No: 1, 1111, 00, 1011100111011



# An Application: Bounce Filter

Bounce filter: remove isolated b's and g's in input.

- Input: b b g b b b g g b g g g g b b b b
- Output: w b b b b b b g g g g g g g b b b b (one-step delay)



no accept state – instead output color of each state you visit

## An Application: Bounce Filter

**Bounce filter:** remove isolated b's and g's in input.

- **Input:**     b b **g** b b b g g **b** g g g g b b b b
- **Output:**   w b b **b** b b b g g g g g g b b b b  
(one-step delay)

**State interpretations.**

- **W:** start
- **BB:** at least two consecutive b's.
- **G:** sequence of b's followed by g.
- **GG:** at least two consecutive g's.
- **B:** sequence of g's followed by b.

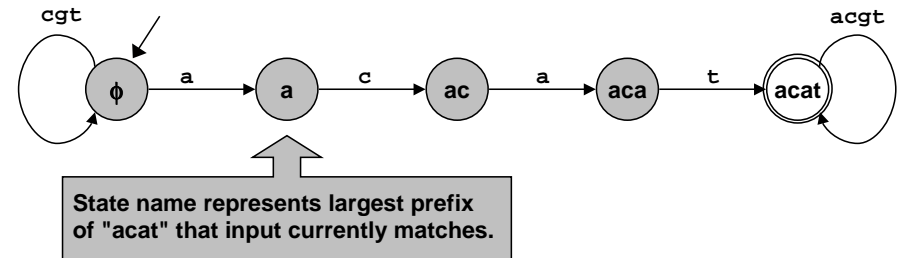
9

## Text Searching

**Build an FSA that accepts all strings that contain 'acat' as a substring.**

- tatg**ac**atg
- ac**a**catg

**Start building:**



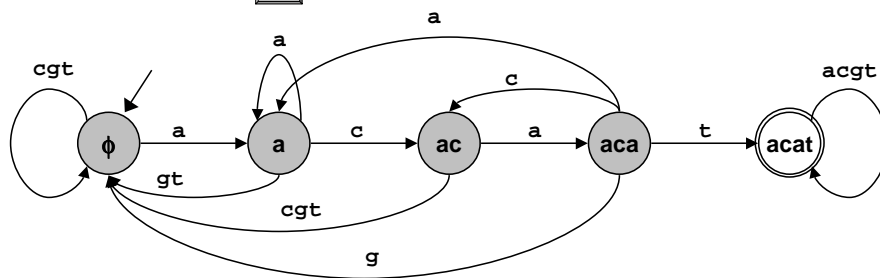
11

## Text Searching

**Build an FSA that accepts all strings that contain 'acat' as a substring.**

- tatg**ac**atg
- ac**a**catg

**Finish building:**



14

## Web Search Application

**Web search engines build FSAs.**

**Standard Web search for:** `cos 126 pattern matching`



**Search engines have different methods for specifying patterns.**

- Which one is most powerful?
- Theory of computation helps us address such issues.

15

## Unix Pattern Matching Tool: egrep

### General regular expressions pattern matching.

- Acts as filter.
- Sends lines from stdin to stdout that "match" argument string.

#### Elementary Examples

```
%egrep 'beth' classlist
03/Smythe/Elizabeth/6/esmythe
03/Bethke/Kristen/3/kbethke
```

Find all lines in file classlist with substring 'beth'

```
% egrep '/3/' classlist
03/Marin/Anthony/3/amarin
03/Arellano/Belen/3/arellano
. . .
03/Weiss/Jacob/3/weiss
```

List all people in precept 3.

```
%egrep 'zeuglodon' mobydick.txt
rechristened the monster zeuglodon and in his
```

```
%egrep 'acat' human.data
gcaacgcacacaacatgcatttt
```

16

## Crossword Puzzle or Scrabble Too Hard?

/usr/dict/words is a list of (25,143) words in dictionary.

/u/cs126/files/textfiles/wordlist.txt is a list of 234,936 words.

#### More Examples

```
% egrep 'hh' /usr/dict/words
beachhead
highhanded
withheld
withhold
```

Two consecutive h's.

```
% egrep 'u.u.u' /usr/dict/words
cumulus
```

A dot matches any single character

```
% egrep '..oo..oo' /usr/dict/words
bloodroot
nincompoophood
schoolbook
schoolroom
```

but not "cookbook"

17

## Egrep Pattern Conventions

### Conventions for egrep:

c	any non-special character matches itself
.	any single character
r*	zero or more occurrences of r
r+	one or more occurrences of r
r?	zero or one occurrences of r
(r)	grouping
r1 r2	logical OR
[aeiou]	any vowel
[^ aeiou]	any non-vowel
^	beginning of line
\$	end of line

### Flags for egrep:

egrep -v match all lines except those specified by pattern

18

## Still More Examples

#### Unix

```
% egrep 'n(ie|ei)ther' /usr/dict/words
neither
```

Do spell checking by specifying what you know.

```
% egrep 'actg(atac)*gcta' human.data
ggtactggctaggac
```

```
% egrep 'actg(atac)*gcta' student.data
tatactgatacatatacatacgctattac
```

Starts and ends with y, odd number of characters.

```
% egrep '^y(...)*y$' /usr/dict/words
yesterday
```

```
% egrep -v '[aeiou]' /usr/dict/words |
egrep '.....'
rhythm
syzygy
```

Find all words with no vowels and 6 or more letters.

19

## Fundamental Questions: Theoretical Minimum

Specifying "pattern" for egrep can be complex.

```
^[^aeiou]*a[^aeiou]*e[^aeiou]*i[^aeiou]*o[^aeiou]*u[^aeiou]*
```



Which aspects are essential?

- Unix egrep regular expressions are useful.
- But more complex than theoretical minimum.

20

## Fundamental Questions: Theoretical Minimum

Regular expressions.

- Match WHOLE string.  
(to convert to equivalent egrep pattern, surround by ^ and \$)
  - regular expression: `0(0|1)*1`
  - egrep pattern: `^0(0|1)*1$`
- `c` any non-special character matches itself
- `r*` zero or more occurrence of `r`
- `(r)` grouping
- `r1|r2` logical OR
- `r1.r2` concatenate (usually suppress `.` symbol)
- `.` any single character
- `[aeiou]` any vowel
- `[^aeiou]` any non-vowel
- `r+` one or more occurrences of `r`
- `r?` zero or one occurrence of `r`
- `-v` match all patterns except...

← not needed

21

## Fundamental Questions: What Kinds of Patterns

What kinds of patterns can be specified by regular expressions?  
(all but one of following)

All bit strings that:

- Begin with 0 and end with 1.
- Equal number of 0's and 1's.
- Have no consecutive 1's.
- Has an odd number of 0's.
- Has 011010 as a substring.

Example

```
0001011011
0111100010
0100101001
0100101011
0001101000
```

22

## Fundamental Questions: What Kinds of Patterns

What kinds of patterns can be specified by regular expressions?  
(all but one of following)

All bit strings that:

- Begin with 0 and end with 1.
- Equal number of 0's and 1's.
- Have no consecutive 1's.
- Has an odd number of 0's.
- Has 011010 as a substring.

Regular Expression

```
0(0|1)*1
not possible
(0|10)*(1|0*)
(1*01*01*)*(1*01*)
(0|1)*011010(0|1)*
```

23

# Formal Languages

An ALPHABET is a finite set of symbols.

- Binary alphabet = {0, 1}
- Lower-case alphabet = {a, b, c, d, ..., y, z}
- Genetic alphabet = {a, c, t, g}

A STRING is a finite sequence of symbols in the alphabet.

- '0111011011' is a string in the binary alphabet.
- 'tigers' is a string in the lower-case alphabet.
- 'acctgaacta' is a string in the genetic alphabet.

A FORMAL LANGUAGE is an (unordered) set of strings in an alphabet.

- Can have infinitely many strings.
- Examples:
  - {0, 010, 0110, 01110, 011110, 0111110, ...}
  - {11, 1111, 111111, 11111111, 1111111111, ...}

# Formal Languages

Can cast any computation as a language recognition problem.

- Is  $x = 23,536,481,273$  a prime number?

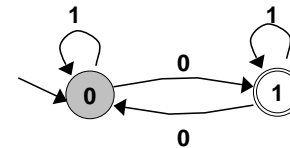


FSA.

- Machine determines whether a string is in language.

Regular expression.

- Shorthand method for specifying a language.



$(1^*01^*01^*)^*(1^*01^*)$   
even # of 0's    exactly one 0

# Duality Between FSA's and RE's

Observation: for each FSA we create, we can find a regular expression that matches the same strings that the FSA accepts.

Is this always the case?



What about the OTHER way around?



Stay tuned: see Lecture T2.

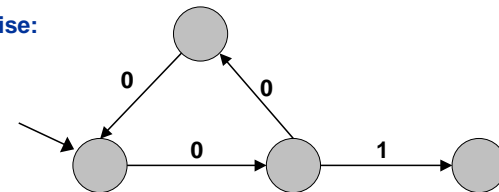
# Limitations of FSA

FSA are simple machines.

- $N$  states  $\Rightarrow$  can't "remember" more than  $N$  things.
- Some languages require "remembering" more than  $N$  things.

No FSA can recognize the language of all bit strings with an equal number of 0's and 1's.

A warmup exercise:



If 01xyz accepted then so is 00001xyz

## Limitations of FSA

No FSA can recognize the language of all bit strings with an equal number of 0's and 1's.

- Suppose an N-state FSA can recognize this language.
- Consider following input: 0000000011111111

N+1 0's    N+1 1's

- FSA must accept this string.
- Some state x is revisited during first N+1 0's since only N states.



0000000011111111  
x    x



- Machine would accept same string without intervening 0's.  
000011111111

- This string doesn't have an equal number of 0's and 1's. 

28

## Looking Ahead

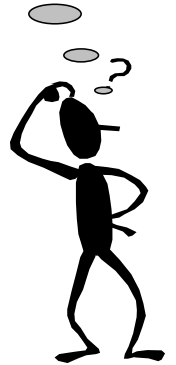
Today.

- Defined a simple abstract machine = FSA.
- Capable of pattern matching.
- Incapable of "counting."
- Need to consider more powerful machines.

Hmm. Which will we run out of first?

Future lectures.

- Define an abstract machine.
- Understand how it works and what it can do.
- Find things it can't do.
- Define a more powerful machine.
- Repeat until we run out of problems or machines.



29

## Lecture T1: Supplemental Notes



## C Code for FSA

fsa1.c

```
#include <stdio.h>
int main(void) {
    int c, state = 0;
    while ((c = getchar()) != EOF) {
        if (state == 0 && c == '0') state = 2;
        else if (state == 0 && c == '1') state = 1;
        else if (state == 1 && c == '0') state = 3;
        else if (state == 1 && c == '1') state = 2;
        else if (state == 2 && c == '0') state = 2;
        else if (state == 2 && c == '1') state = 2;
        else if (state == 3 && c == '0') state = 2;
        else if (state == 3 && c == '1') state = 1;
    }

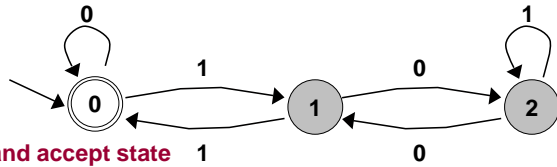
    if (state == 3)
        printf("Yes.\n");
    else
        printf("No.\n");
    return 0;
}
```

straightforward to convert FSA's into C program or to build with hardware

31

## A Fourth Example

FSA to decide if integer (represented in binary) is divisible by 3?



0 is start and accept state

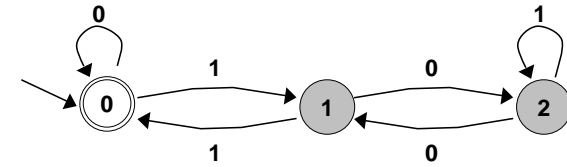
What bit strings does it accept?

- Yes: 11 ( $3_{10}$ ), 110 ( $6_{10}$ ), 1001 ( $9_{10}$ ), 1100 ( $12_{10}$ ), 1111 ( $15_{10}$ ), 10011 ( $18_{10}$ ), integers divisible by 3.
- No: 1 ( $1_{10}$ ), 10 ( $2_{10}$ ), 100 ( $4_{10}$ ), 101 ( $5_{10}$ ), 111 ( $7_{10}$ ), integers not divisible by 3.

32

## A Fourth Example

FSA to decide if input (convert binary to decimal) is divisible by 3?



How does it work?

- State 0: input so far is divisible by 3.
- State 1: input has remainder 1 upon division by 3.
- State 2: input has remainder 2 upon division by 3.
- Transition example.
  - Input 1100 ( $12_{10}$ ) ends in state 0.
  - If next bit is 0 then stay in state 0: 11000 ( $24_{10}$ ).
  - Adding 0 to last bit is same as multiplying number by 2. Remains divisible by 3.

33

## Regular Expressions

Rules for creating regular expressions (RE's):

- 0 or 1 or  $\epsilon$  symbols
- (a) grouping
- ab concatenation
- a + b logical OR
- a\* closure (0 or more replications)

use + instead of |

where a and b are regular expressions.

Examples:

$\epsilon$  = empty string

- (10)\*  $\epsilon$ , 10, 1010, 101010, ...
- 0(0 + 1)\*0 00, 000, 010, 0000, 0110, ...
- (1\*01\*01\*01\*)\*  $\epsilon$ , 000, 000000, 11101110101111, ...

37