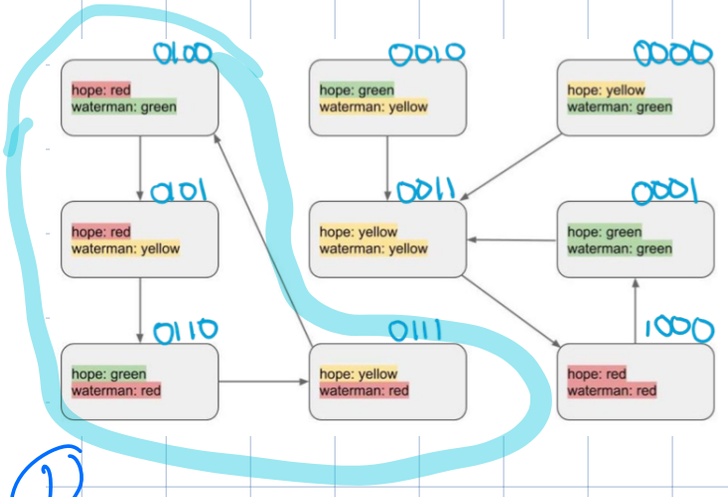
Continuing from last class: goal was to succinctly represent set of "safe" program states



1

Our goal: represent set of safe states

Could write as:
List[0100, 0101, 0110, 0111]

This is much smaller than objects, but still grows linearly!

2

Idea: give each state a name
- Smallest piece of information in a computer is a bit (0 or 1)
- Assign name to each state "bit string" (sequence of bits)'

For n states, how many bits? Log2(N)

REPRESENT AS BITS: 0, 1

2 STATES => => 1 BIT

4 POSSIBLE BITSTRINGS

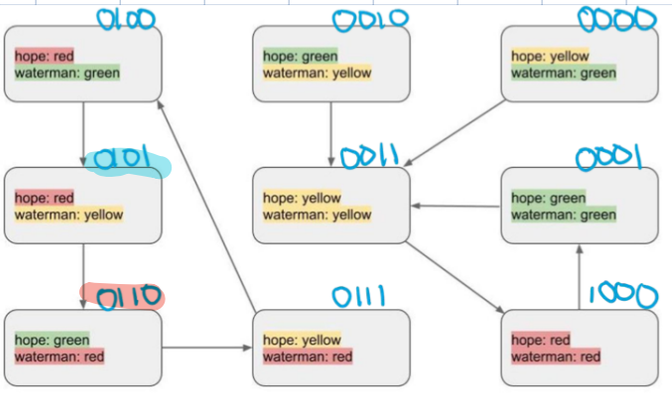
4 STATES => 00 01 10 11 => 2 BITS

8 POSSIBLE BITSTRINGS

8 " => 000 001 010 011 100 101 110 111

=> 3 BITS

N bits => 2^N possible "things" (in this case states)
OR
M states => log2(M) bits
log2(9) = 3.1 => 4 (need all strings to be same length)



=> Using this idea, we can represent each state as a bitstring of 0's and 1's

0 1 0 0

b_1 b_2 b_3 b_4

> Notation: we write state of individual bit as b0, b1, …, where b0 could be 0 or 1 (like a Boolean variable)

What can we do with this?

- Represent states VERY succinctly (lower space complexity)
- Can represent set of reachable states as a decision tree => Binary Decision Diagram (BDD)

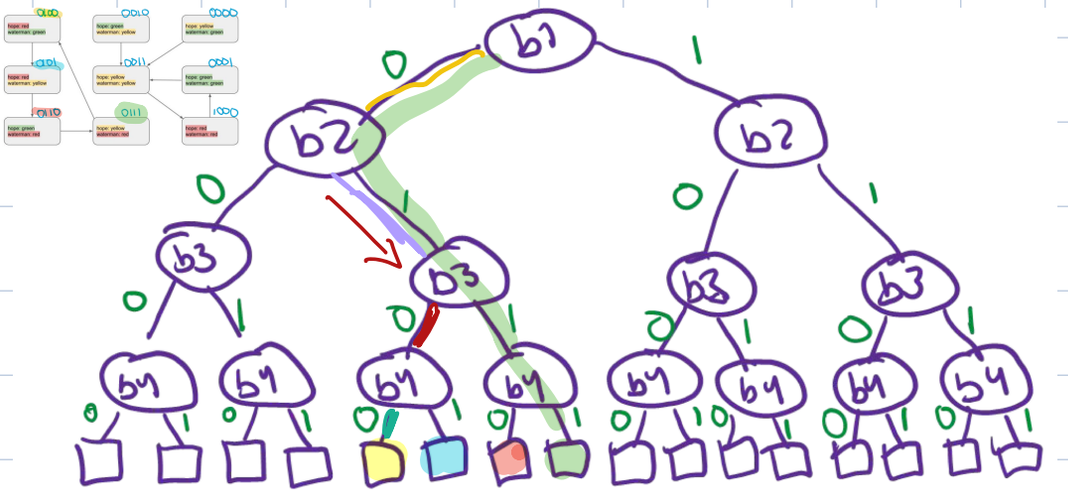
BDD: How it works (high level)

- Run BFS to find reachable states
- Mark each leaf of BDD as reachable for each state
- Can make a logical formula (a boolean expression) to represent reachable states

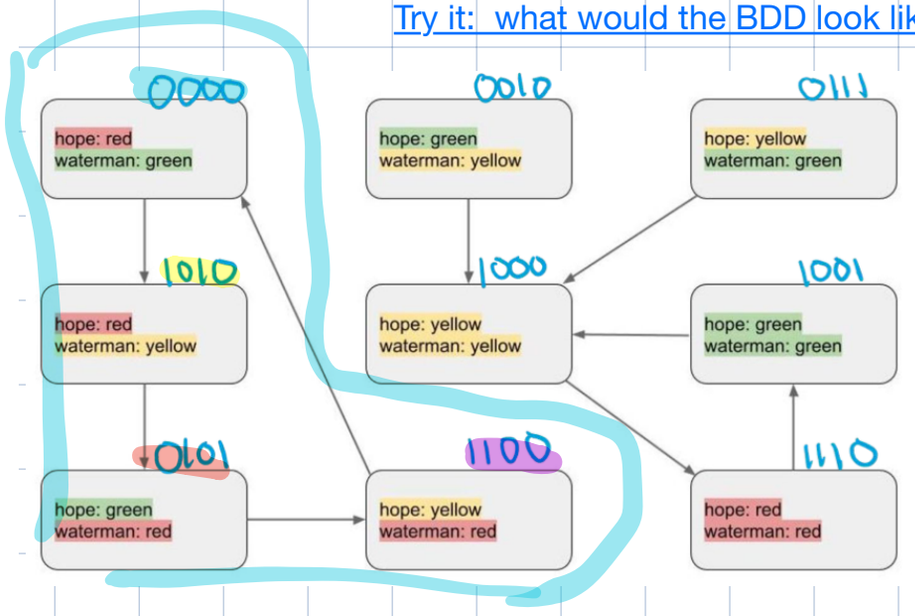
Reachable: (b1 == 0) AND (b2 == 1)

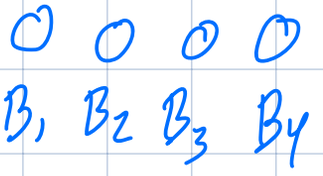
Could also write as: (not b1) and (b2)

==> To check if the state is safe, Don't even need to store a list of reachable states! Just need to check if the bitstring matches the formula!



Try it: what would the BDD look like if we picked these names instead?



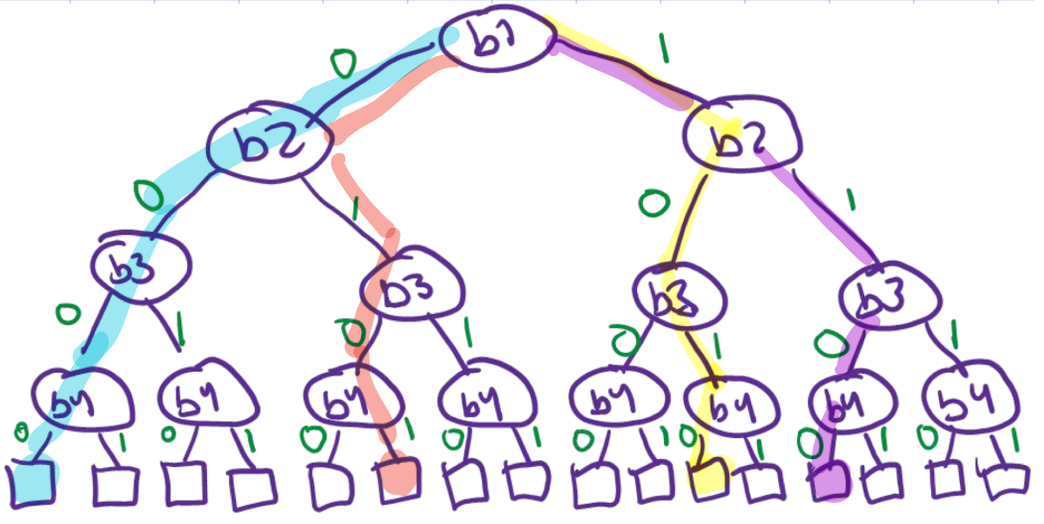


Formula would be a lot more complicated!
((not b1) and (not b2) and (not b3) and (not b4)) OR
((b1) and (not b2) and (b3) and (not b4) OR
…

Takeaways:
- Coming up with a good naming of states is nontrivial
  => Problems are provably computationally hard, people work on this for specific settings

- Depending on what the BDD looks like, logical formula may get complex

Want to learn more?
Consider taking: Logic for Systems
=> Learn about algorithms people have made to represent these in more concise ways, make tools to help, software to use



Compression: how to store text efficiently?

"HATS"

↳ 8 BITS/CHAR (ASCII)

32 BITS/CHAR (UNICODE)

ASCII: Use 8 bit bitstring for each character
=> 4 * 8 bits = 32 bits total

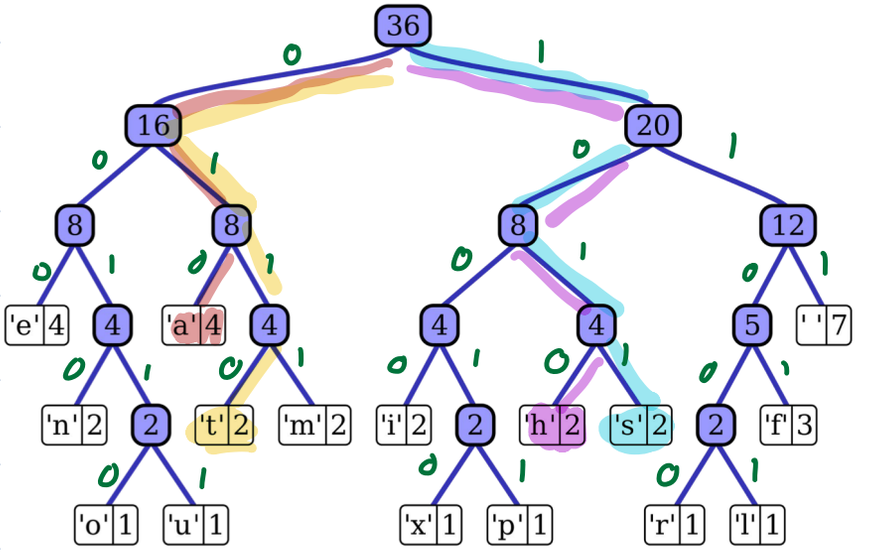
Storage: increases linearly with number of characters
BigWiki (from HW6):
141 million characters * 8 bits ~= 136MB

Idea: language (and most data) isn't random
Compression: use fewer bits to represent data that appears more frequently
"e" appears more often than "x"—so maybe we can use fewer bits?

Huffman coding:
- for some input data, find unique bitstring to represent characters, use fewer bits for more common letters

Takeaways:
=> Leveraging patterns in data for efficiency
=> Encoding pattern in BDD structure (use to convert)

"this is an example of a huffman tree"



https://en.wikipedia.org/wiki/Huffman_coding

Char	Freq	Code
space	7	111
a	4	010
e	4	000
f	3	1101
h	2	1010
i	2	1000
m	2	0111
n	2	0010
s	2	1011
t	2	0110
l	1	11001
o	1	00110
p	1	10011
r	1	11000
u	1	00111
x	1	10010

"HATS"
h: 1010 4
a: 010 3
t: 0110 4
s: 1011 4

ENCODED AS => 15 BITS

2x SMALLER!

To decompress the data, we'd start with the string of all these bits, then follow the tree to find each letter:

1010 | 010 | 0110 | 1011

h | a | t | s