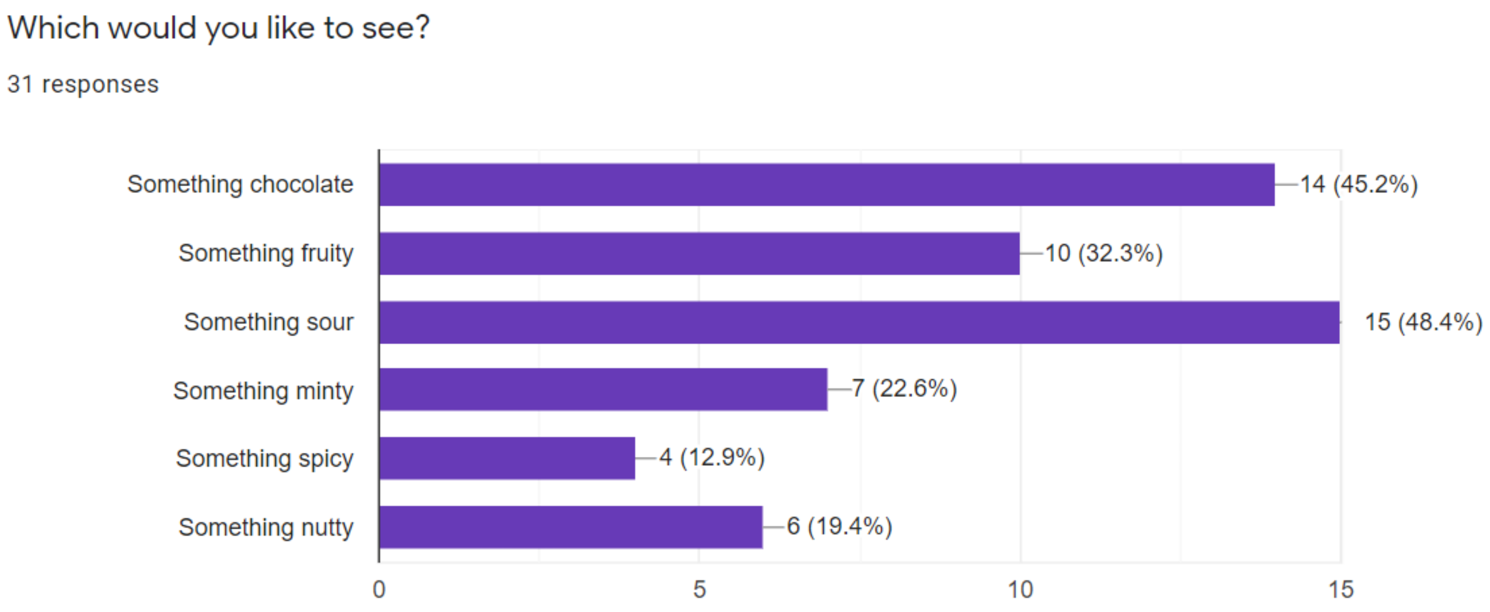
CS3102 Theory of Computation

www.cs.virginia.edu/~njb2b/cstheory/s2020

Warm up:

What features present in Java/Python are missing from straightline programs?



Missing Features

- goto
- loops next
- Arith ...
- types
- generic types
- concurrency
- pointers
- classes/objects

- IDE
- IF today
- exceptions
- Inheritance
- functions today
- error messages
- compiling
- graphics
- data structures today

expressive
efficiency
human

Logistics

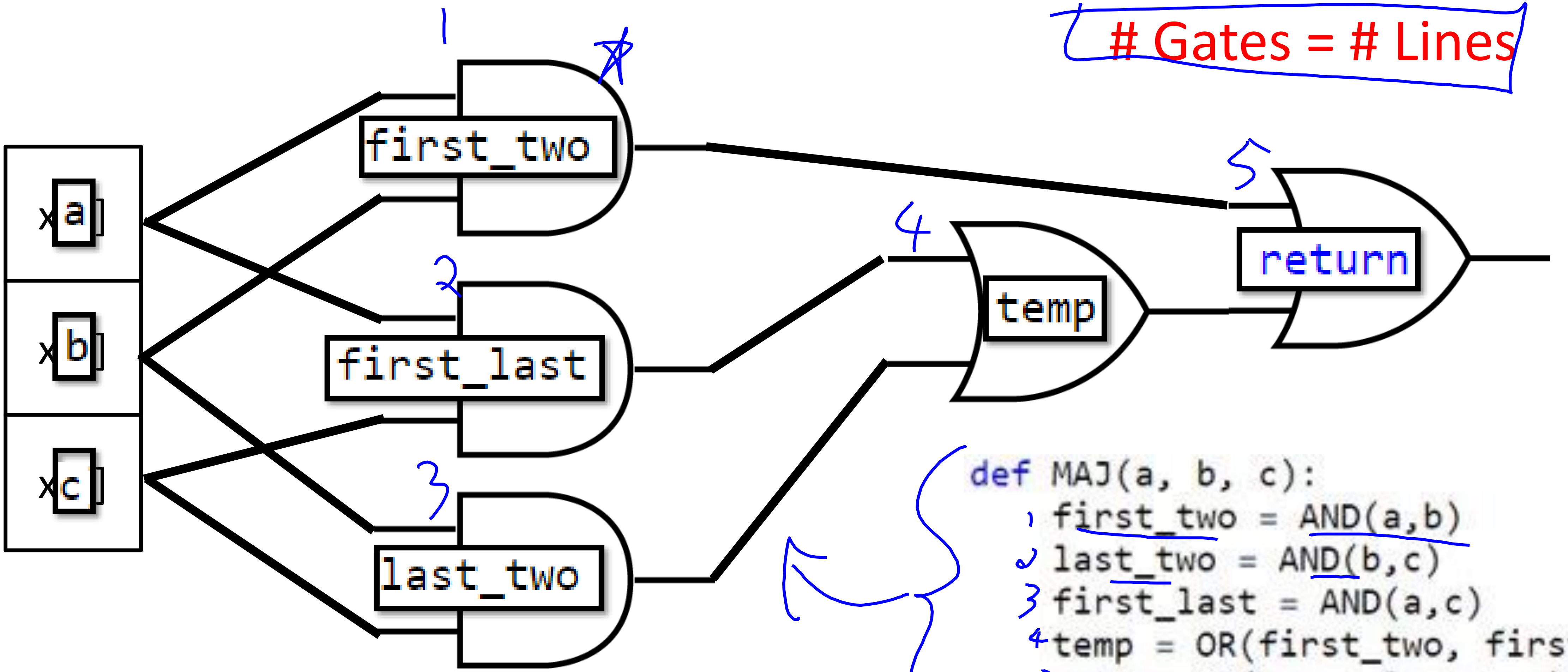
- Exercise 1 due this afternoon
  - Didn't submit? You have 48 hours to do so with a 25% penalty
- Quiz ~~2 due today~~
- Exercise 2 is out.
  - Some stuff due Thursday, the rest due Tuesday

Last Time

- Boolean Circuits as a model of computing
- Straightline Programs as a model of computing
- Proved NAND-Straightline = NAND-Circ = AON-Circ = AON-straightline

Majority with Boolean Circuits

Gates = # Lines




```
def MAJ(a, b, c):
  1 first_two = AND(a,b)
  2 last_two = AND(b,c)
  3 first_last = AND(a,c)
  4 temp = OR(first_two, first_last)
  5 return OR(temp, last_two)
```

NAND Straightline = AON Straightline

NAND -> AON

x = NAND(a,b)

Becomes

temp = AND(a,b)

x = NOT(temp)

AON -> NAND

x = NOT(a)

Becomes

x= NAND(a,a)

x = AND(a,b)

Becomes

temp= NAND(a,b)

x=NAND(temp,temp)

x = OR(a,b)

Becomes

t1 = NAND(a,a)

t2 = NAND(b,b)

x= NAND(t1,t2)

same functions

Syntactic Sugar

- "Full-featured" programming languages are identical to simple ones
- We can add new features without changing the underlying computing model
- These features can make programs easier to reason about and more readable

AON

User-Defined Procedures

```
def NOT(a):
    return NAND(a,a)
def AND(a,b):
    temp = NAND(a,b)
    return NOT(temp)
def OR(a,b):
    temp1 = NOT(a)
    temp2 = NOT(b)
    return NAND(temp1,temp2)

def MAJ(a,b,c):
    and1 = AND(a,b)
    and2 = AND(b,c)
    and3 = AND(a,c)
    or1 = OR(and1,and2)
    return OR(or1,and3)
```

start : NAND

def AND(a,b) :

temp = NAND(a,b)

~~return NOT(temp)~~

~~temp~~

return NAND(temp,temp)

"Translating" Procedures

- Adding procedures does not change computing model
- We can convert a program with procedures into a program without them

```
def NOT(a):
    return NAND(a,a)

def AND(a,b):
    temp = NAND(a,b)
    return NOT(temp)
```

```
def AND(a,b):
    temp = NAND(a,b)
    return NAND(temp,temp)
```

Procedure for translating procedures

- Paste code from procedure
- Use arguments in place of parameters
- Rename variables from the procedure to be "fresh"

```
def NOT(a):
    return NAND(a,a)

def AND(a,b):
    temp = NAND(a,b)
    return NAND(temp,temp)


def OR(a,b):
    temp1 = NAND(a, a)
    temp2 = NAND(b,b)
    return NAND(temp1, temp2)
```

```
def MAJ(a,b,c):
    and1 = AND(a,b)
    and2 = AND(b,c)
    and3 = AND(a,c)
    or1 = OR(and1,and2)
    return OR(or1,and3)
```

Before

After

How many gates?

- How many NAND gates does this use to compute MAJ?

```
def NOT(a):
    return NAND(a,a)

def AND(a,b):
    temp = NAND(a,b)            2
    return NAND(temp,temp)         3


def OR(a,b):
    temp1 = NAND(a, a)      4
    temp2 = NAND(b,b)      5
    return NAND(temp1, temp2)    6
```

```
def MAJ(a,b,c):
    and1 = AND(a,b)  >  2
    and2 = AND(b,c)  8  2
    and3 = AND(a,c)  9  2
    or1 = OR(and1,and2) 10  3
    return OR(or1,and3) 11  3
```

12 gates

Conditionals

- Values of some variables might depend on a condition
- [Code]
- [Translated]

```
def example(a,b):
    w = AND(a,b)
    if w:
        x = OR(a,b)
        y = NOT(a)
        z = NOT(b)
    else:
        x = AND(a,b)
        y = OR(a,b)
        z = NOT(a)
    return x,y,z
```

Translating Conditionals

- Pre-compute each of the possible values
- Use a procedure to determine which to assign

```
def IF(cond,a,b):
    not_cond = NAND(cond,cond)
    temp1 = NAND(b,not_cond)
    temp2 = NAND(a,cond)
    return NAND(temp1,temp2)


def IF(cond,a,b):
    not_cond = NOT(cond)
    if_true = AND(cond,a)
    if_false = AND(not_cond,b)
    return OR(if_true,if_false)
```

```
def example(a,b):
    w = AND(a,b)

    x_ct = OR(a,b)
    y_ct = NOT(a)
    z_ct = NOT(b)

    x_cf = AND(a,b)
    y_cf = OR(a,b)
    z_cf = NOT(a)

    x = IF(w,x_ct,x_cf)
    y = IF(w,y_ct,y_cf)
    z = IF(w,z_ct,z_cf)
```

Lookup

- Indexing into a bitstring
- The $Lookup$ function of order k:

$$LOOKUP_k : \{0,1\}^{\boxed{2^k}+k} \rightarrow \{0,1\}$$

Defined such that for $x \in \{0,1\}^{2^k}, i \in \{0,1\}^k$:

$$LOOKUP_k(x, i) = x_i$$

$LOOKUP_k$

$k = 3$

8

x:

0	1	2	3	4	5	6	7
0	1	0	0	1	1	1	0

First 2^k bits of input
Considered as a bitstring

i:

0	1	0

Last k bits of input
Considered as an index

Theorem

There is a NAND-Cricuit that computes $LOOKUP_k: \{0,1\}^{2^k+k} \to \{0,1\}$

Moreover, the number of gates required is at most $4 \cdot 2^k$

Proof idea

$k = 3$

$k-1$

- Consider index i
- If the first bit of i is 0, then the bit we're looking for is in the first half of x
- Do lookup for $k - 1$

$\text{lookup}(x_0, x_1, i) = \text{IF}(i, x_1, x_0)$

x_0 x_1 i

x:

0	1	2	3	4	5	6	7
0	1	0	0	1	1	1	0

i:

0	1	0

x:

0	1	2	3
0	1	0	0

i:

1	0

Defining $LOOKUP_k$

For $k \geq 2, LOOKUP_k(x_0, \ldots, x_{2^k-1}, i_0, \ldots, i_{k-1})$ is equal to:

$$IF(i_0, LOOKUP_{k-1}\left(x_{2^{k-1}}, \ldots, x_{2^k-1}, i_1, \ldots, i_{k-1}\right), LOOKUP_{k-1}(x_0, \ldots, x_{2^{k-1}-1}, i_1, \ldots, i_{k-1})$$

Base Case

```
def LOOKUP1(x0, x1, i0):
    return IF(i0,x1,x0)
```

Next Step

LOOKUP2

```
def LOOKUP2(x0,x1,x2,x3,i0,i1):
    first_half = LOOKUP1(x0,x1,i1)
    second_half = LOOKUP1(x2,x3,i1)
    return IF(i0,second_half,first_half)
```

1
2

[LOOKUP3 and 4](#)

Counting Gates

Show this uses at most $3 \cdot 2^k$ gates (lines of code)

Base case: $k = 1$

3 gates

Inductive Hyp: #gates $\text{100Gup}_{k-1} \le 4 \cdot 2^{k-1}$

$\text{100G}_k \le 4 \cdot 2^k$

$\text{100G}_k = 2 \cdot \text{100G}_{k-1} + 3 \le 2 \cdot (4 \cdot 2^{k-1}) + 3$

$= 4 \cdot 2^k + 3$

Counting Gates

Show this uses at most $4 \cdot 2^k$ gates (lines of code)

$4 \cdot 2^{\tau}$

Computing Every Finite Function

- Next we'll show that NAND is "universal"
- Any finite function can be computed by some NAND-straightline program (equivalently, a NAND-circuit)

Idea

Consider the function $f:\{0,1\}^3 \to \{0,1\}$

Input	Output
000	0
001	0
010	1
011	0
100	1
101	1
110	0
111	0

We will have one variable to represent each possible input. We'll do a lookup with the actual input to select the proper output

Straightline Code for F

```
def F(x0,x1,x2):
    F000=0
    F001=0
    F010=1
    F011=0
    F100=1
    F101=1
    F110=0
    F111=1
    return LOOKUP3(F000,F001,F010,F011,F100,F101,F110,F111,x0,x1,x2)
```

Input	Output
000	0
001	0
010	1
011	0
100	1
101	1
110	0
111	0

Getting 0 and 1

Computing any function

- Make a variable to represent each possible input
- Assign its value to match the correct output
- Use LOOKUP to select the proper output for the given input

How many gates?

- How many gates does this construction take?

You can compute any finite function $f: \{0,1\}^n \rightarrow \{0,1\}^m$ with a NAND Circuit using no more than $c \cdot m \cdot 2^n$ gates

Note: This can be imporved to $c \cdot m \cdot \frac{2^n}{n}$ (theorem 4.16 in TCS)

Counting gates

1. Create variables for each input

2. Assign 0,1 to each input

3. Do the LOOKUP

What does this mean?

- Your laptop is a 64-bit machine. Given enough transistors, it can compute any function $f: \{0,1\}^{64} \rightarrow \{0,1\}^{64}$