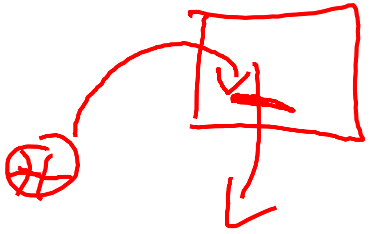
CS3102 Theory of Computation

$f : \{0,1\} \to \{0,1\}$



UVA wins: $\mathrm{NAND}(a, \neg a)$

NCAA men's basketball · Today, 9:00 PM

Notre Dame Fighting Irish (15 - 8)

at

Virginia Cavaliers (15 - 7)

ND wins: $\neg\mathrm{NAND}(a, \neg a)$

Warm up:

Is this function computable with a NAND-Circuit?

$$\boxed{f(a) = \begin{cases} 1 & \text{if UVA wins against ND tonight} \\ 0 & \text{otherwise} \end{cases}}$$

Logistics

- Quiz 3 out
  - Due Thursday
- Exercise 3 is out
  - Last exercise before midterm

Last Time

- Showing that NAND/AON can compute any finite function
- Started showing how we could have a function that simulates programs

program, input

How are programs run?

- Have a table of variables
- Execute code in sequence
- Update values in table
- Return a value from the table

variables	value
a	0
b	1
c	2
...	
...	
Return	0

output

Eval n, m, s

input

lines

NAND-Straightline

Programs as Bits

- To evaluate a program with another program, we need to convert the first program into bits

1. Number each variable (first $\boxed{n}$ go to input, last m to outputs)
2. Represent each line as 3 numbers (outvar, in1, in2)
3. Represent program as (n,m,[Lines])

0 1

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

2 3 4

(2,0,0)
(3,1,1)
(4,2,3)

(2,1,[(2,0,0),(3,1,1),(4,2,3)])

Variable	Number
a	0
b	1
temp1	2
temp2	3
return	4

XOR to bits $|\{0,1\}^x| = 2^x$

```
def XOR(a,b):
    u = NAND(a,b)
    v = NAND(a, u)
    w = NAND(b, u)
    return NAND(v,w)
```

[(2, 0, 1)
(3, 0, 2)
(4, 1, 2)
(5, 3, 4)]

return

Variable	Number
a	0
b	1
u	2
v	3
w	4
Return	5

$n = 2$

$m = 1$

$s = 4$

ess

Total bits = $\overset{n}{2} + \overset{m}{1} + 3 \cdot 5 \cdot \lceil \log_2(n+s) \rceil$

XOR to bits

```
def XOR(a,b):
    u = NAND(a,b)
    v = NAND(a, u)
    w = NAND(b, u)
    return NAND(v,w)
```

$n = 2$

$m = 1$

$s = 4$

Variable	Number
a	0
b	1
u	2
v	3
w	4
return	5

Total bits = 3 [numbers per line] $\cdot$ 3 [bits per number] $\cdot$ 4[lines] + 6 [length of n + m]

How big is this?

Number of var = n+s

1. Number each variable $\lceil \log_2 (n+s) \rceil$ ~~$\lceil \log_2 3s \rceil$ bits~~ each
2. Represent each line as 3 numbers (outvar, in1, in2)
3. Represent program as $(\underbrace{n,m}_{2\lceil \log_2 s \rceil \text{ bits}},[\text{Lines}])$

$3s \cdot \lceil \log_2 3s \rceil$ bits

$3 \cdot s \cdot \text{Length}(\text{var})$

$S(s)$ = size of the program of s lines in bits

$$S(s) \leq \underline{4s\lceil \log_2 3s \rceil}$$

$4 \cdot s \cdot \log s$

$$\ell = \lceil \log_2 3s \rceil$$

Defining EVAL

4 s log s

$$EVAL_{s,n,m}\colon \{0,1\}^{\overset{\text{program}}{\boxed{S(s)}}+\overset{\text{input}}{n}} \to \{0,1\}^m \leftarrow m \text{ outputs}$$

Input: bit string representing a program (first $S(s)$ bits) plus input values (remaining n bits)

Output: the result of running the represented program on the provided input, or m 0's if there's a "compile error"

Defining the EVAL function

$n = 2$
$m = 1$

Representation: XOR

(2, 0, 1),
(3, 0, 2),
(4, 1, 2),
(5, 3, 4)

Input:

0, 1

Variable	Value
0	0
1	1
2	1
3	1
4	0
5	1

Psuedocode for EVAL

- Table T:
  - holds variables and their values
- $GET(T, i)$
  - Returns the bit of T associated with variable i
- $UPDATE(T, i, b)$
  - Returns a new table such that variable i's value has been changed to b

T
1
0
1
1

Psuedocode for EVAL

- Input:
  - Numbers n, m, s, t representing the number of inputs, outputs, variables, and lines respectively
  - L, a list of triples representing the program
  - A string x to be given as input to the program
- Output:
  - Evaluation of the program represented by L when run on input x

load input bits into table

Read & write to the table

return the last m bits

```
Let T be table of size t
For i in range(n):
    T = UPDATE(T, i, x[i])
For (i,j,k) in L:
    a = GET(T, j)
    b = GET(T, k)
    T = UPDATE(T, i, NAND(a,b))
For i in range(m):
    Y[i] = GET(T, t − m + i)
Return Y
```

i gets assigned NAND(j,k)

EVAL in NAND

- Next we implement $EVAL_{s,n,m}$ using NAND

$$GET(T, \boxed{i})$$

ℓ = length of each variable

- Get the bit at "row" i of T

- Look familiar? $\quad lookup_\ell(T, i)$

- How many gates to implement? $\sim 2^\ell$

100 kupr

UPDATE

$$UPDATE_\ell : \{0,1\}^{2^\ell + \ell + 1} \rightarrow \{0,1\}^{2^\ell}$$

- To change index i of table T to bit $\boxed{b}$
- For every index except i, return the same value
- For index i, return b instead
- Define $EQUAL_j : \{0,1\}^\ell \rightarrow \{0,1\}$ which returns 1 if the input binary number is equal to j

Note: $EQUAL_j$ can be done in $c \cdot \ell$ gates

UPDATE pseudocode

For j in range(2^{ℓ}):

$$\left.\begin{array}{l} a = EQUALS_j(i) \\ newT[j] = IF(a, b, T[j]) \end{array}\right\} \text{Runs } 2^{\ell} \text{ times}$$

Return $newT$

GET = lookup

UPDATE =

Conclusion

- What we know:
  - We can compute any finite function with circuits
  - We can compute a function to evaluate programs of a certain size
- Big question:
  - How expensive are functions?
  - Some are more expensive than others, how big could they get?
  - If I wanted to be able to evaluate a program for any function $\{0,1\}^n \to \{0,1\}$, how big would the eval circuit need to be?

Complexity

- The "complexity" of a function:
  – Measure of the resources required to compute that function
- Complexity Class:
  – A set of functions defined by a complexity measure

Categorizing Functions by Circuit Size

- No functions require more than $cm2^n$ gates
  - Proved last class
- Some functions require much less
  - E.g. IF
- Observation: some functions are more "complicated" than others!
- Idea: categorize functions by resources required to implement them using a particular computing model

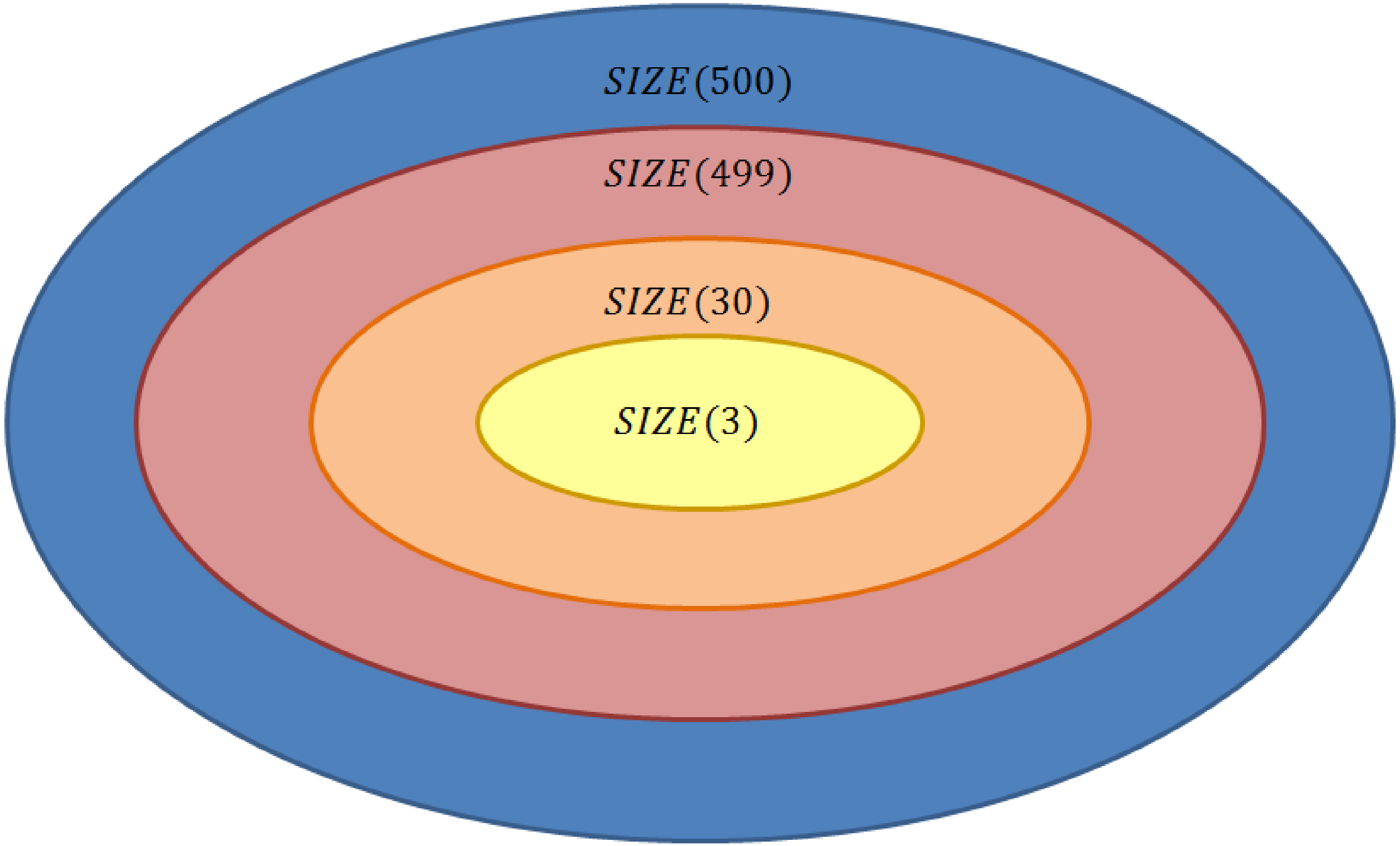
SIZE

- $SIZE(s)$: The set of all functions that can be implemented by a circuit of at most s NAND gates

$SIZE(1000m2^n)$ Contains all functions $f: \{0,1\}^n \rightarrow \{0,1\}^m$

- TCS also uses:
  - $SIZE_{n,m}(s)$:The set of all n-input, m-output functions that can be implemented with at most s NAND gates
  - $SIZE_n(s)$: The set of all n-input, 1-output functions that can be implemented with at most s NAND gates

Comparing Classes



If $x \leq y$, then $SIZE(s) \subseteq SIZE(y)$

Theorem

- Let $SIZE^{AON}(s)$ represent the set of all functions that can be computed using at most s AND/OR/NOT gates

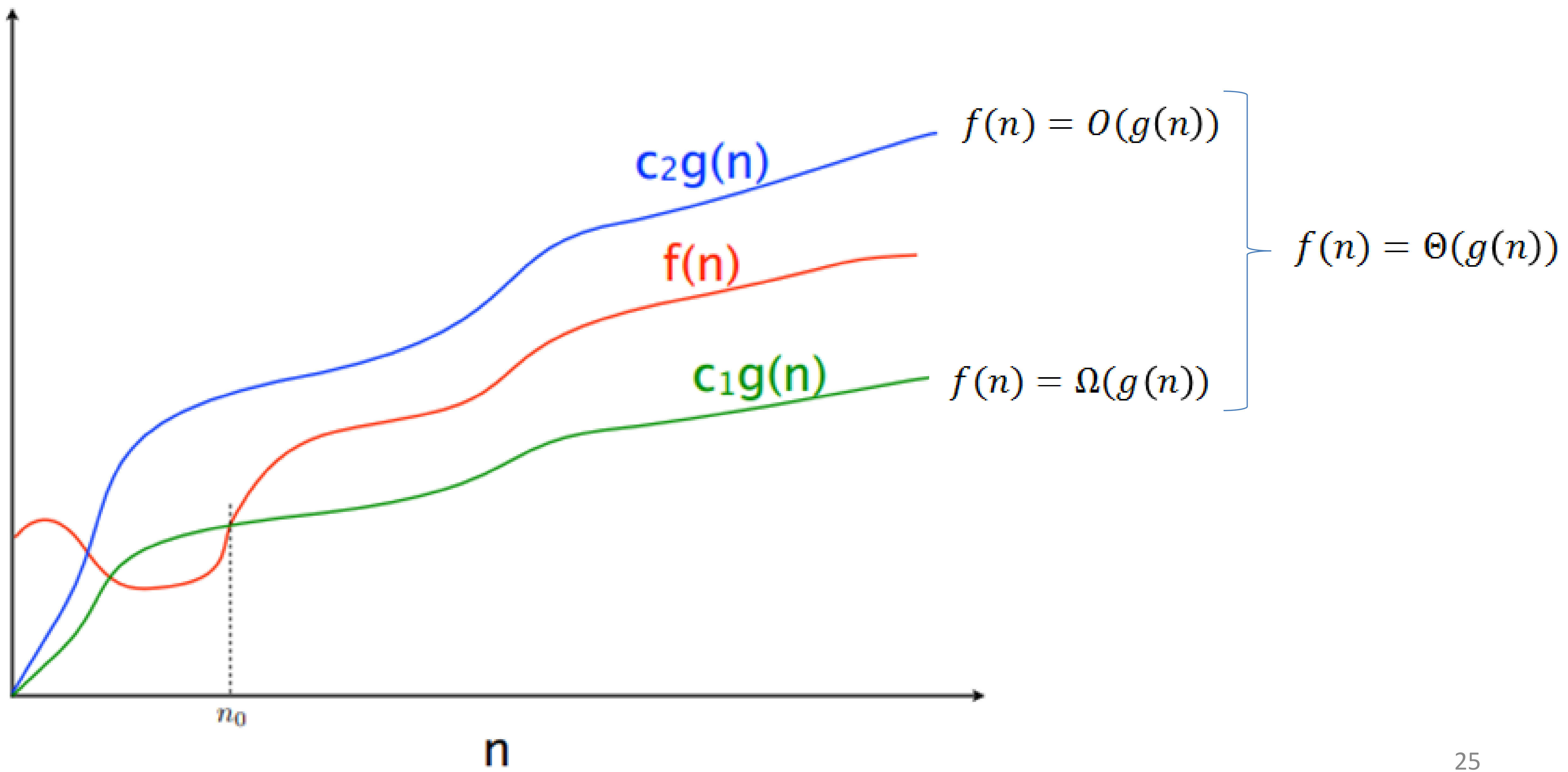
$$SIZE\left(\frac{s}{2}\right) \subseteq SIZE^{AON}(s) \subseteq SIZE(3s)$$

Proof

$$SIZE\left(\frac{s}{2}\right) \subseteq SIZE^{AON}(s) \subseteq SIZE(3s)$$

O, Ω, Θ

- Groups functions together
- Each uses a function as a bound for other functions
- O (Big-Oh):
  - O(f(n)) = the set of all functions "asymptotically upper-bounded" by f
- Ω (Big-Omega):
  - Ω(f(n)) = the set of all functions "asymptotically lower-bounded" by f
- Θ (Big-Theta):
  - Θ(f(n)) = the set of all functions "asymptotically tight-bounded" by f



$c_2g(n)$
$f(n)$
$c_1g(n)$
n_0
n
$f(n) = O(g(n))$
$f(n) = \Theta(g(n))$
$f(n) = \Omega(g(n))$

Definitions

- $O(g(n))$
  - At most within constant of g for large n
  - $\{f: \mathbb{R} \to \mathbb{R} | \exists$ constants $c, n_0 > 0$ s.t. $\forall n > n_0, f(n) \leq c \cdot g(n)\}$

- $\Omega(g(n))$
  - At least within constant of g for large n
  - $\{f: \mathbb{R} \to \mathbb{R} | \exists$ constants $c, n_0 > 0$ s.t. $\forall n > n_0, f(n) \geq c \cdot g(n)\}$

- $\Theta(g(n))$
  - "Tightly" within constant of g for large n
  - $\Omega(g(n)) \cap O(g(n))$

Showing Big-Oh

- To show: $n \log n \in O(n^2)$

Showing Big-Omega

- To Show: $2^n \in \Omega(n^2)$

Showing Big-Theta

- To Show: $\log_x n = \Theta(\log_y n)$