CS3102 Theory of Computation

Warm up:

Why might we consider computing infinite functions?

Logistics

- Exercise 3 is out
  - Last exercise before midterm

Last Time

- Using a circuit to evaluate a function

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 Thursday
- 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

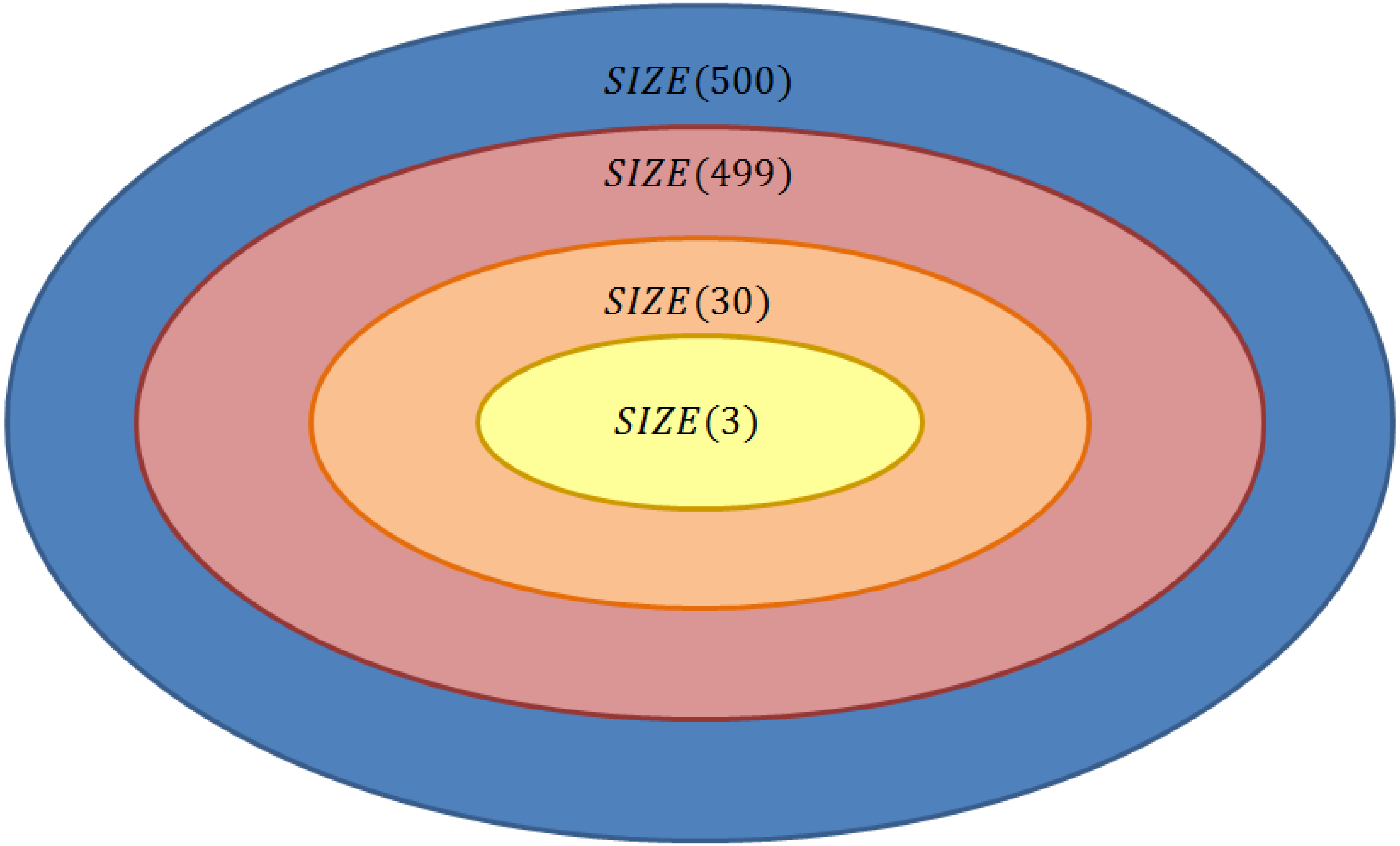
SIZE

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

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

- 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

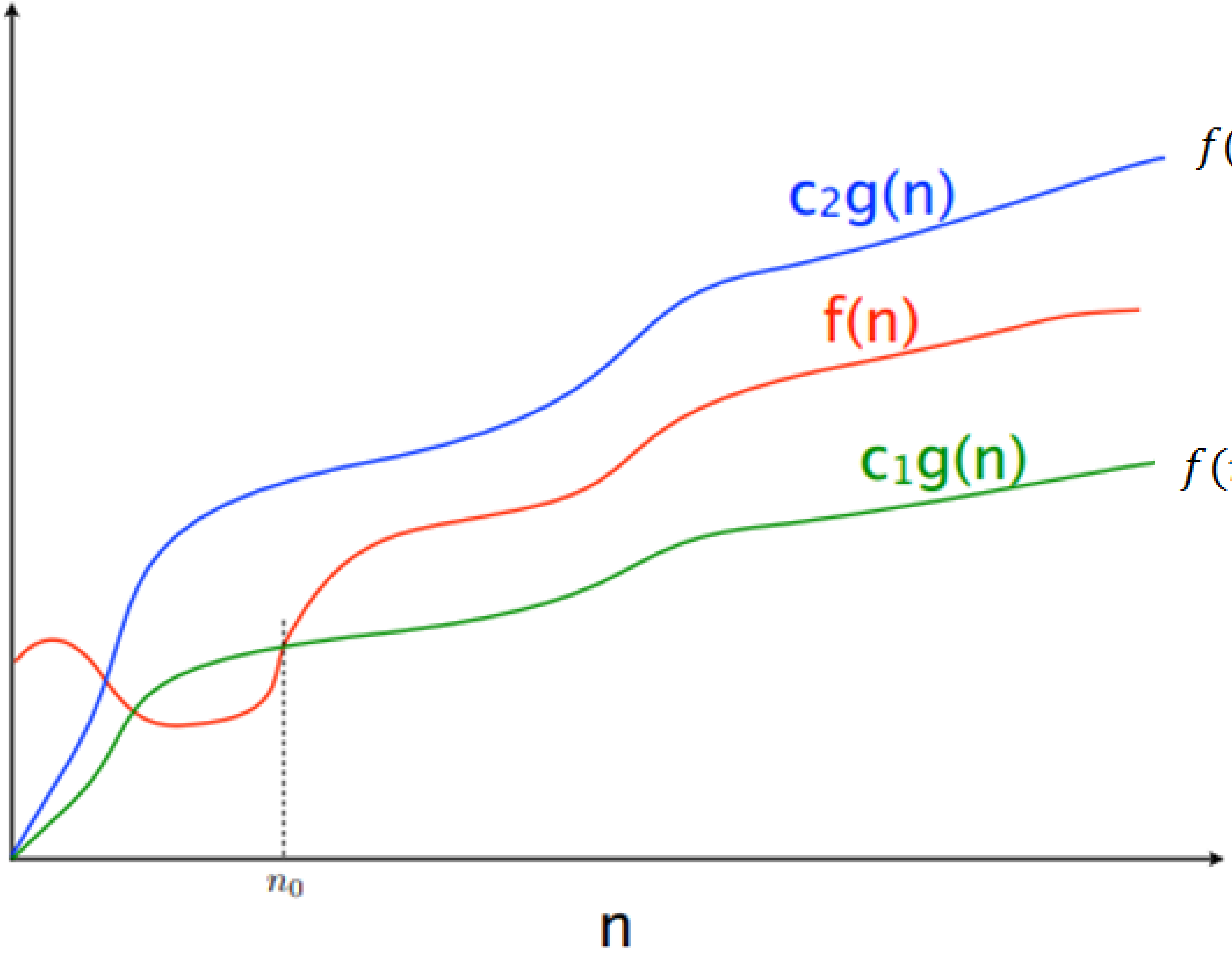
$$SIZE\left(\frac{s}{2}\right) \subseteq SIZE^{AON}(s) \subseteq SIZE(3s)$$

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):
  - $\Omega(f(n))$ = the set of all functions "asymptotically lower-bounded" by f
- Θ (Big-Theta):
  - $\Theta(f(n))$ = the set of all functions "asymptotically tight-bounded" by f



$f(n) = O(g(n))$

$f(n) = \Omega(g(n))$

$f(n) = \Theta(g(n))$

Definitions

- $O(g(n))$
  - At most within constant of g for large n
  - $\{f : \mathbb{R} \to \mathbb{R} \mid \exists$ 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
  - $\{f : \mathbb{R} \to \mathbb{R} \mid \exists$ 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: $\log_x n = \Theta\left(\log_y n\right)$

How does this help us?

- We often want to know the “trend” of efficiency
- Constants don’t matter as much (often change among models of computing)
- Makes it easier to measure complexity

Using O to measure *EVAL*

- Input:
  - Numbers n, m, s, t representing the number of inputs, outputs, slines, and variables 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

```
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
```

UPDATE pseudocode

For j in range(2^{ℓ}):

$\quad a = EQUALS_j(i)$

$\quad newT[j] = IF(a, b, T[j])$

Runs 2^{ℓ} times

Return $newT$

$\ell = \log_2 3s =$ bits required per variable

How many gates are required?

- TCS Theorem 5.3: There is a constant $\delta > 0$, such that for every sufficiently large n there is a function $f: \{0,1\}^n \rightarrow \{0,1\}$ such that $f \notin SIZE\left(\frac{\delta 2^n}{n}\right)$. That is, the shortest NAND program to compute f requires at least $\delta \cdot \frac{2^n}{n}$ gates.

How to show this

1. Count the number of n- input functions
2. Count the number of programs of size $\delta \cdot \frac{2^n}{n}$
3. Show there are more functions than programs

How many functions?

- How many functions are there of form $\{0,1\}^n \to \{0,1\}$?
- How can we count this?

How many programs?

- Bits required for an s-line program:
  - At most $3s$ variables (3 variables mentioned for each of the s lines)
    - $\log_2 3s$ bits per variable
  - 3 variables per line
    - $3 \cdot \log_2 3s$ bits per line
  - s lines total
    - $3s \log_2 3s$ bits total
- Upper bound on the number of s-line programs:
  - $2^{3s\log_2 3s}$
  - $2^{O(s \log s)}$

Fixing the Length

- If we fix the length of the programs to be $\delta \cdot \frac{2^n}{n}$ lines, how many programs are there?
- $2^{c \cdot s \log s}$ programs of length s
- $2^{\frac{c\delta 2^n}{n} \log s}$programs
- Let $\delta = \frac{1}{c}$
- $2^{\frac{2^n}{n} \log s} < 2^{2^n}$
- Some programs require more than $\delta \cdot \frac{2^n}{n}$ lines

64 bit machine

- I want to make $EVAL$ to evaluate any program for a function $f: \{0,1\}^{64} \rightarrow \{0,1\}$. How many gates do I need?
- Some functions will require at least $\delta \cdot \frac{2^n}{n}$ gates.
  - Assume $\delta = \frac{1}{10}$
- We must evaluate programs longer than: $\frac{2^{64}}{640}$ lines
- We need at least $\left(\frac{2^{64}}{640}\right)^2 \log_2\left(\frac{2^{64}}{640}\right)$ gates
  - 4.5×10^{34} gates
  - Your computer would need to be the area of the solar system

Conclusion

- A domain of 2^{64} is large enough that perhaps it's not useful to think of the function as finite
- Let's think of that as an infinite function instead
- We need a model of computing for infinite functions

After the exam

- A model of computing for infinite functions
- How to do simple operations over and over again to compute
  - Real computers update memory by computing "simple" functions in hardware over and over again