Indian Institute of Information Technology Allahabad

Data Structures

Asymptotic Analysis

Dr. Shiv Ram Dubey
Assistant Professor
Department of Information Technology
Indian Institute of Information Technology, Allahabad

Email: srdubey@iiita.ac.in Web: https://profile.iiita.ac.in/srdubey/

DISCLAIMER

The content (text, image, and graphics) used in this slide are adopted from many sources for academic purposes. Broadly, the sources have been given due credit appropriately. However, there is a chance of missing out some original primary sources. The authors of this material do not claim any copyright of such material.

The plan

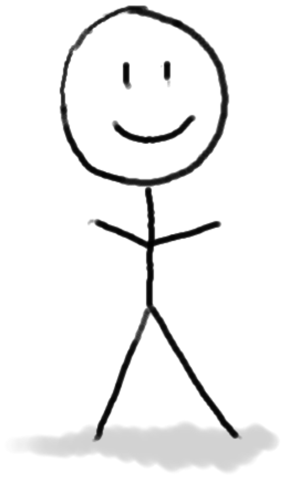
- Sorting Algorithms
  - InsertionSort: does it work and is it fast?
  - MergeSort: does it work and is it fast?
  - Skills:
    - Analyzing correctness of iterative and recursive algorithms.
    - Analyzing running time of recursive algorithms

- How do we measure the runtime of an algorithm?
  - Worst-case analysis
  - Asymptotic Analysis

Worst-case analysis

Sorting a sorted list should be fast!!

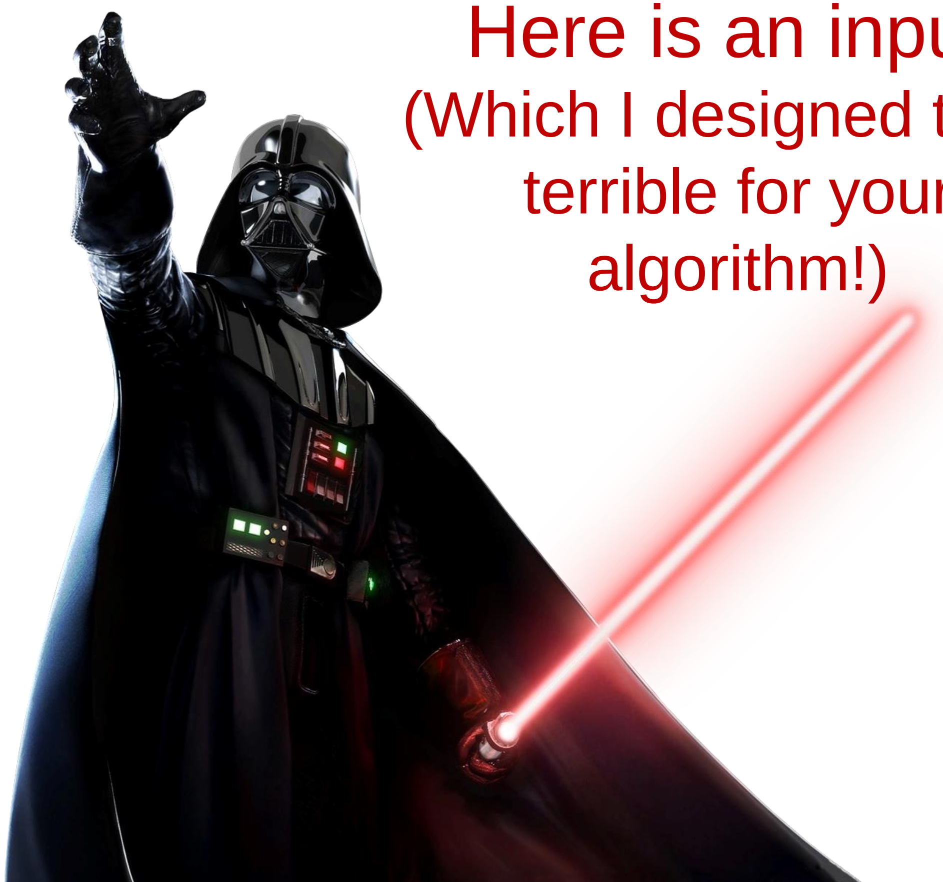
The "running time" for an algorithm is its running time on the worst possible input.



Algorithm designer

Here is your algorithm!

Algorithm:
- Do the thing
- Do the stuff
- Return the answer



Here is an input!
(Which I designed to be terrible for your algorithm!)

Big-O notation

- What do we mean when we measure runtime?
  - We probably care about wall time: how long does it take to solve the problem, in seconds or minutes or hours?
- This is heavily dependent on the programming language, architecture, etc.
- These things are very important, but are not the point of this class.
- We want a way to talk about the running time of an algorithm, independent of these considerations.

Main idea:

Focus on how the runtime **scales** with n (the input size).

Informally….

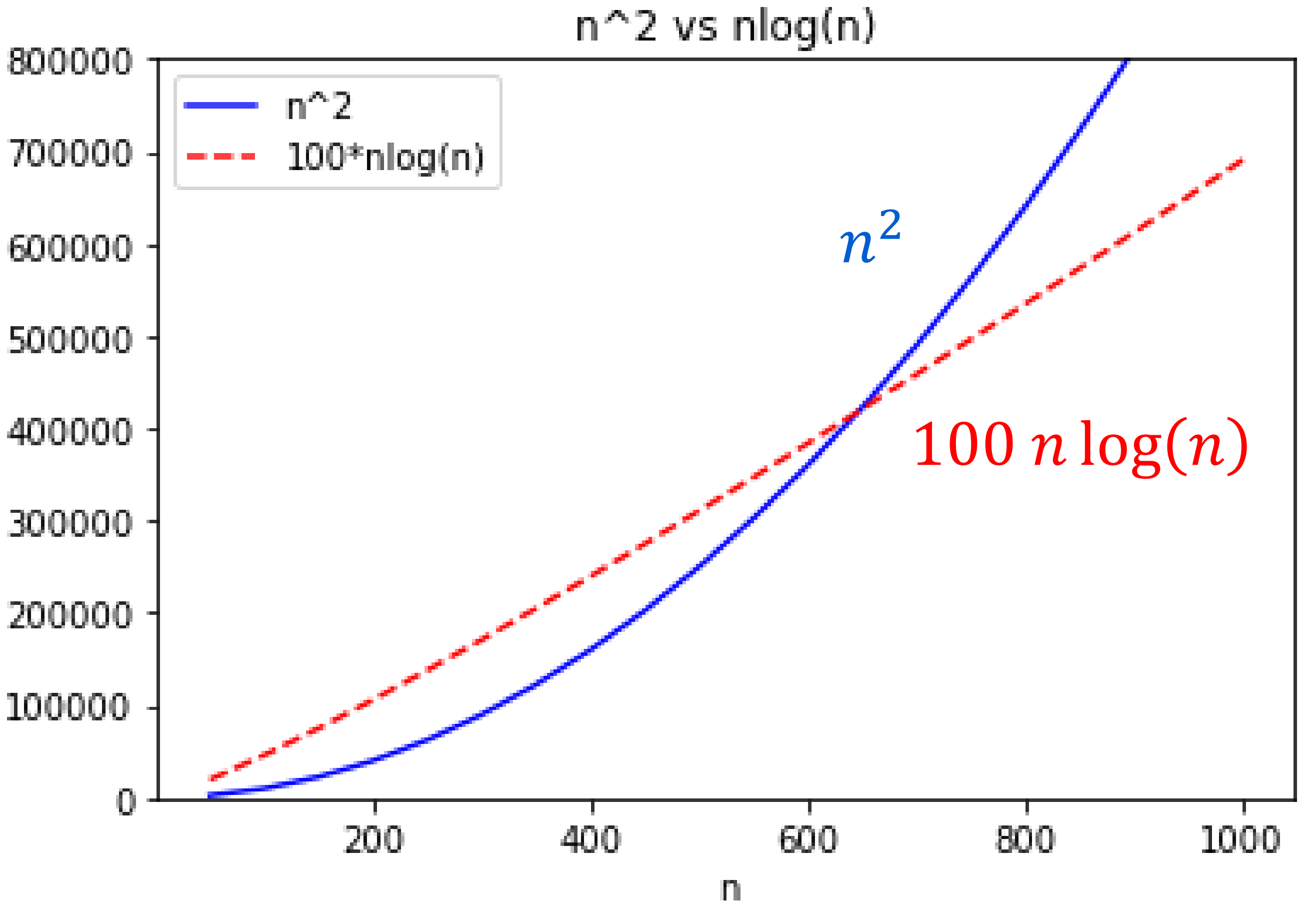
Number of operations	Asymptotic Running Time
$\frac{1}{10} \cdot n^2 + 100$	$O(n^2)$
$0.063 \cdot n^2 - .5\, n + 12.7$	$O(n^2)$
$100 \cdot n^{1.5} - 10^{10000}\sqrt{n}$	$O(n^{1.5})$
$11 \cdot n\log(n) + 1$	$O(n\log(n))$

(Only pay attention to the largest function of n that appears.)

We say this algorithm is "asymptotically faster" than the others.

So 100 $n\log(n)$ operations is "better" than n^2 operations?

But when n=200, that's not true at all!



Yeah, but it's true once n is at least 700 or so.

Asymptotic Analysis

One algorithm is “faster” than another if its runtime scales better with the size of the input.

Pros:

- Abstracts away from hardware- and language-specific issues.
- Makes algorithm analysis much more tractable.

Cons:

- Only makes sense if n is large (compared to the constant factors).

1000000000 n
is “better” than n^2 ?!?!

O(…) means an upper bound

pronounced “big-oh of …” or sometimes “oh of …”

- Let $T(n)$, $g(n)$ be functions of positive integers.
  - Think of $T(n)$ as a runtime: positive and increasing in n.
- We say “$T(n)$ is $O\big(g(n)\big)$” if $T(n)$ grows no faster than $g(n)$ as n gets large.
- Formally,

$$T(n) = O\big(g(n)\big)$$

$$\Longleftrightarrow$$

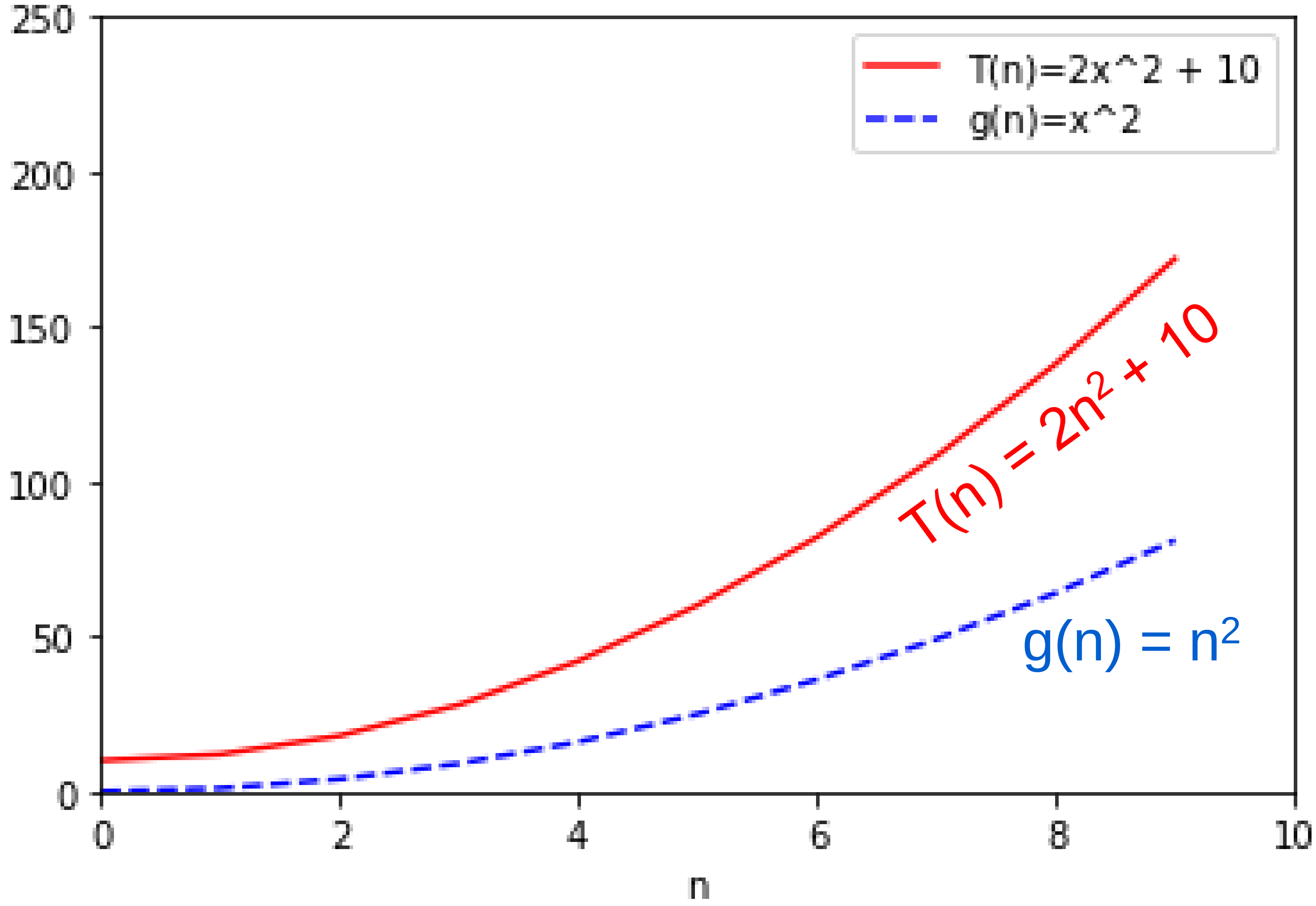
$$\exists c, n_0 > 0 \ \ s.t. \ \ \forall n \geq n_0,$$

$$0 \leq T(n) \leq c \cdot g(n)$$

Example

$$2n^2 + 10 = O(n^2)$$

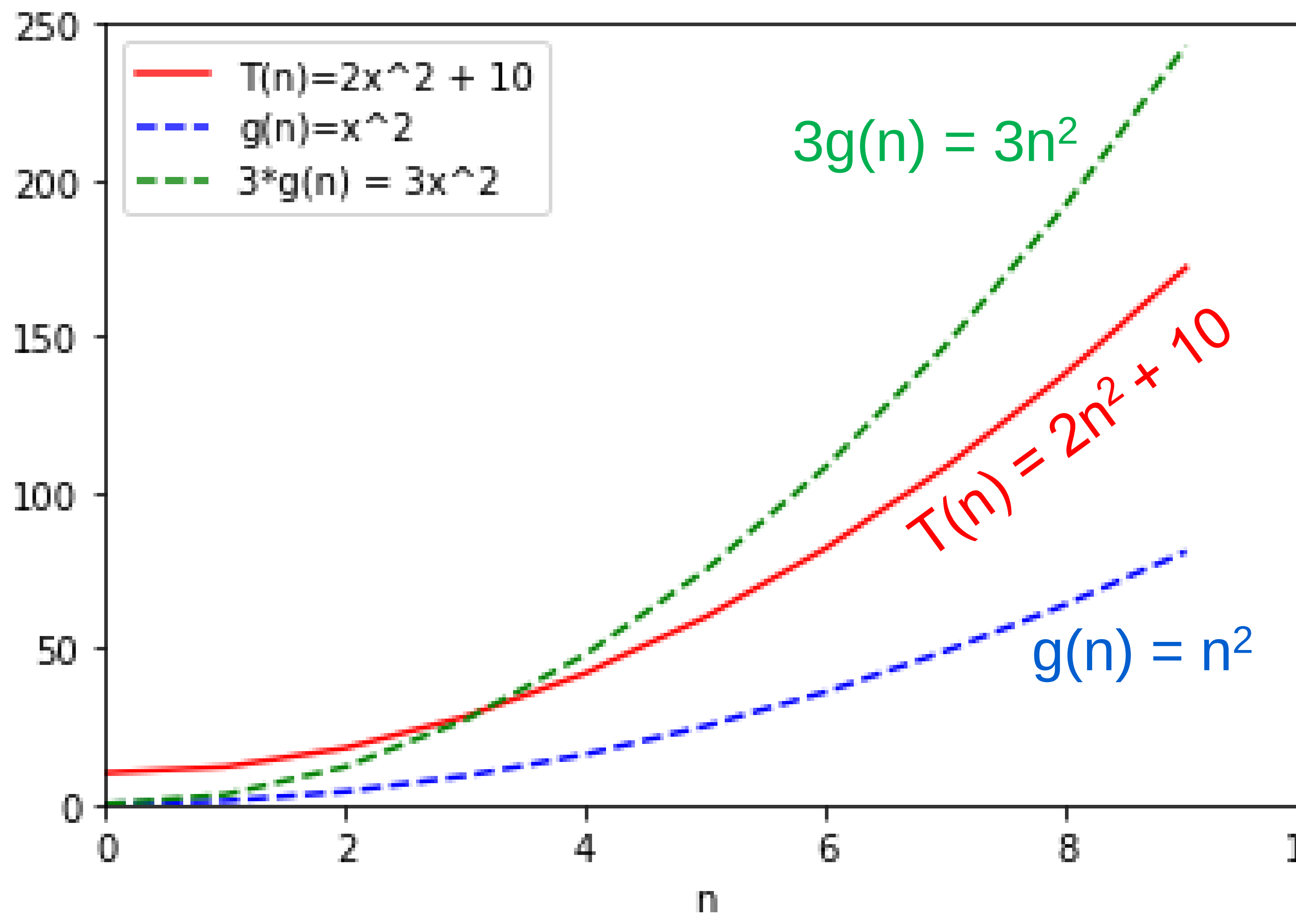
$$T(n) = O(g(n))$$
$$\Longleftrightarrow$$
$$\exists c, n_0 > 0 \;\; s.t. \;\; \forall n \geq n_0,$$
$$0 \leq T(n) \leq c \cdot g(n)$$



Example

$$2n^2 + 10 = O(n^2)$$

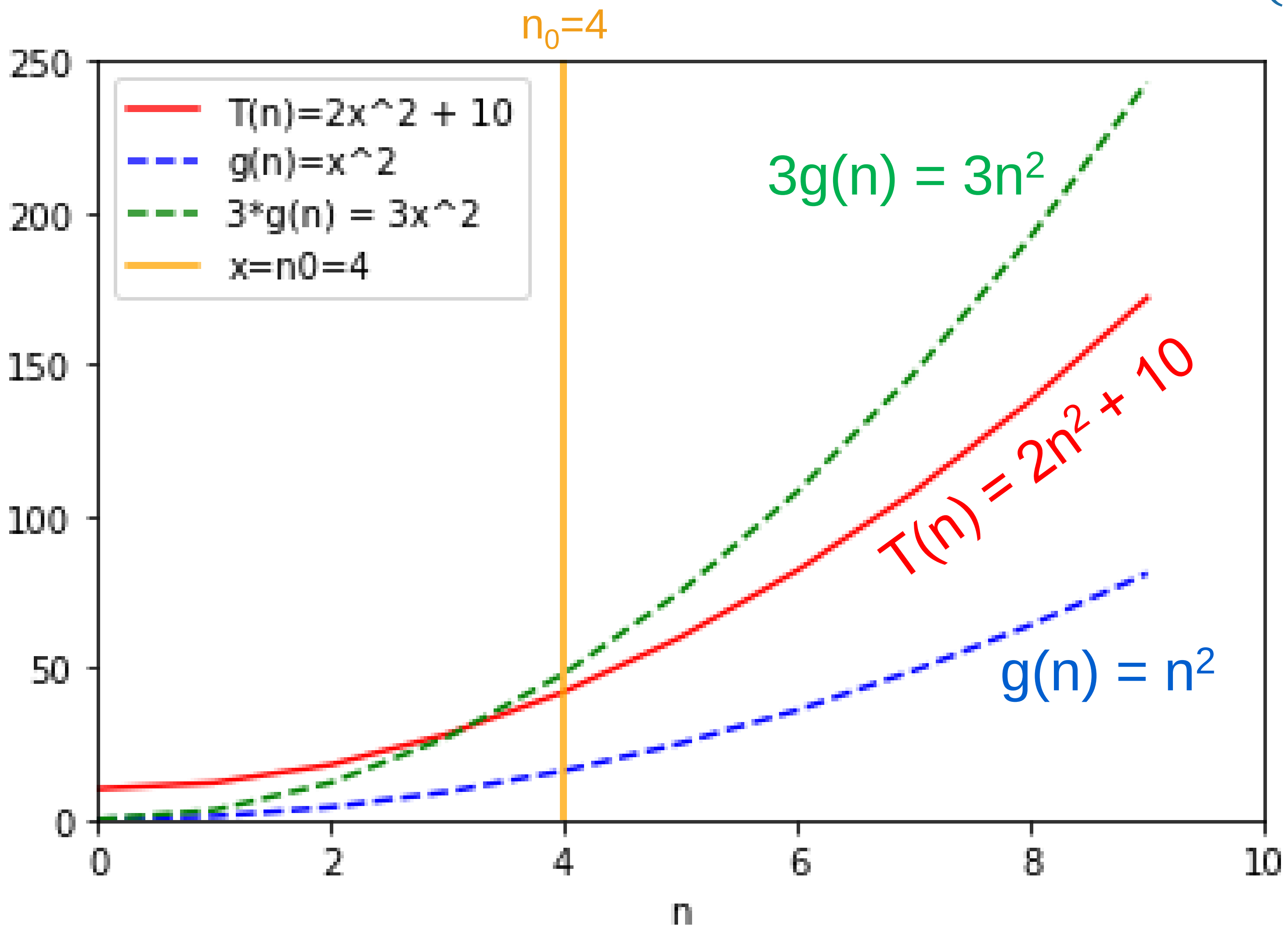
$$T(n) = O(g(n))$$
$$\Leftrightarrow$$
$$\exists c, n_0 > 0 \;\; s.t. \;\; \forall n \geq n_0,$$
$$0 \leq T(n) \leq c \cdot g(n)$$



Example

$$2n^2 + 10 = O(n^2)$$

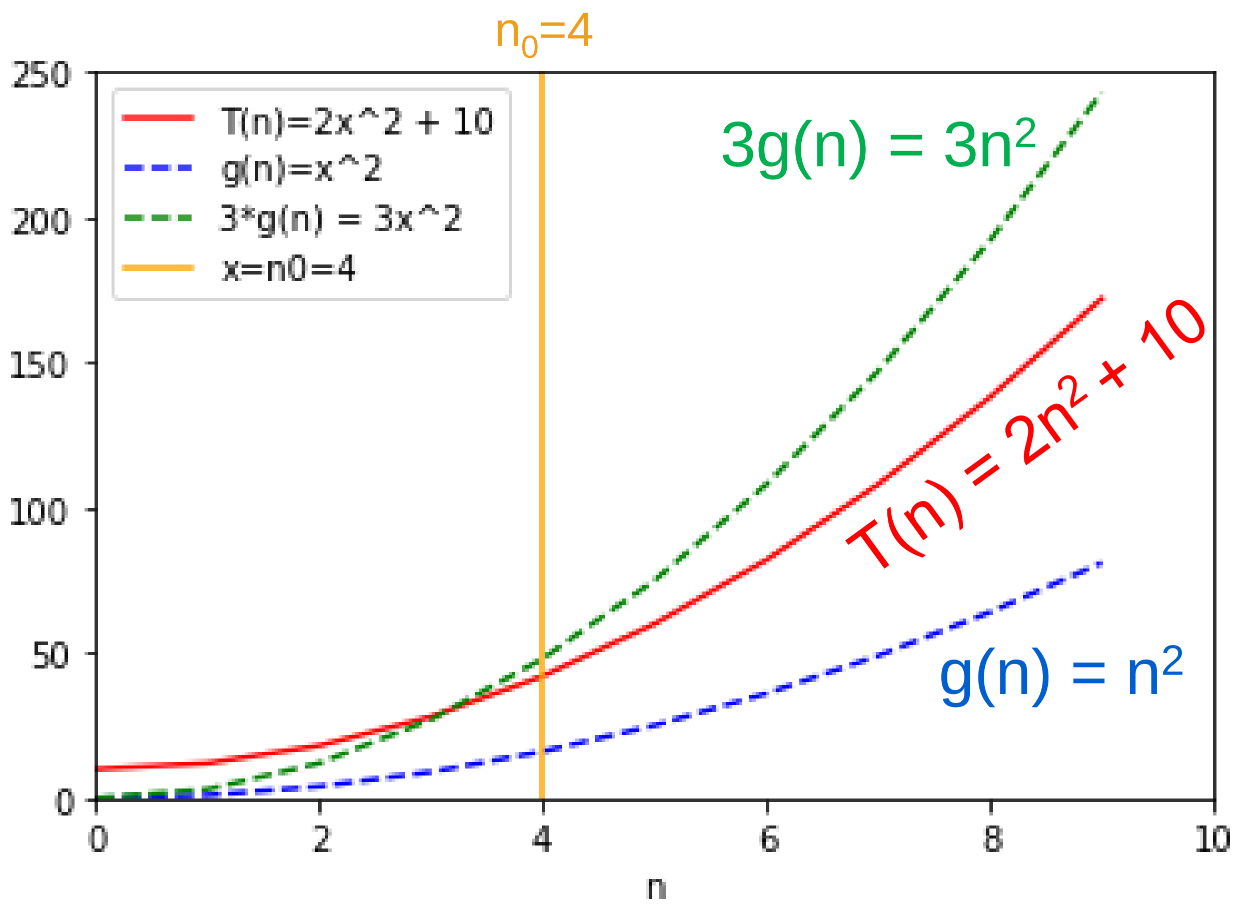
$$T(n) = O(g(n))$$
$$\Longleftrightarrow$$
$$\exists c, n_0 > 0 \;\; s.t. \;\; \forall n \geq n_0,$$
$$0 \leq T(n) \leq c \cdot g(n)$$



Example

$$2n^2 + 10 = O(n^2)$$

$$T(n) = O(g(n))$$
$$\Longleftrightarrow$$
$$\exists c, n_0 > 0 \;\; s.t. \;\; \forall n \geq n_0,$$
$$0 \leq T(n) \leq c \cdot g(n)$$



Formally:

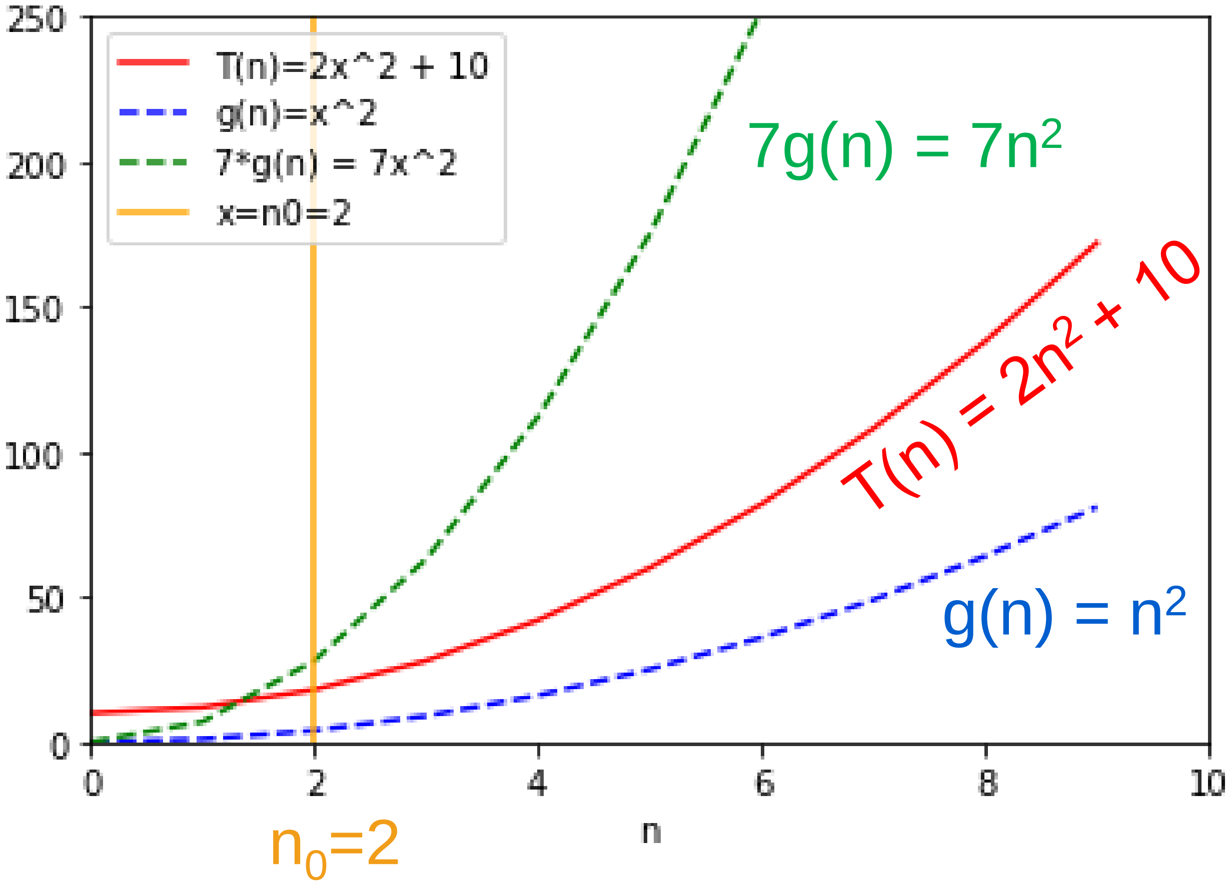
- Choose c = 3
- Choose $n_0 = 4$
- Then:

$$\forall n \geq 4,$$
$$0 \leq 2n^2 + 10 \leq 3 \cdot n^2$$

Example

$$2n^2 + 10 = O(n^2)$$

$$T(n) = O(g(n))$$
$$\Leftrightarrow$$
$$\exists c, n_0 > 0 \ \ s.t. \ \ \forall n \geq n_0,$$
$$0 \leq T(n) \leq c \cdot g(n)$$



Formally:

- Choose c = 7
- Choose n_0 = 2
- Then:

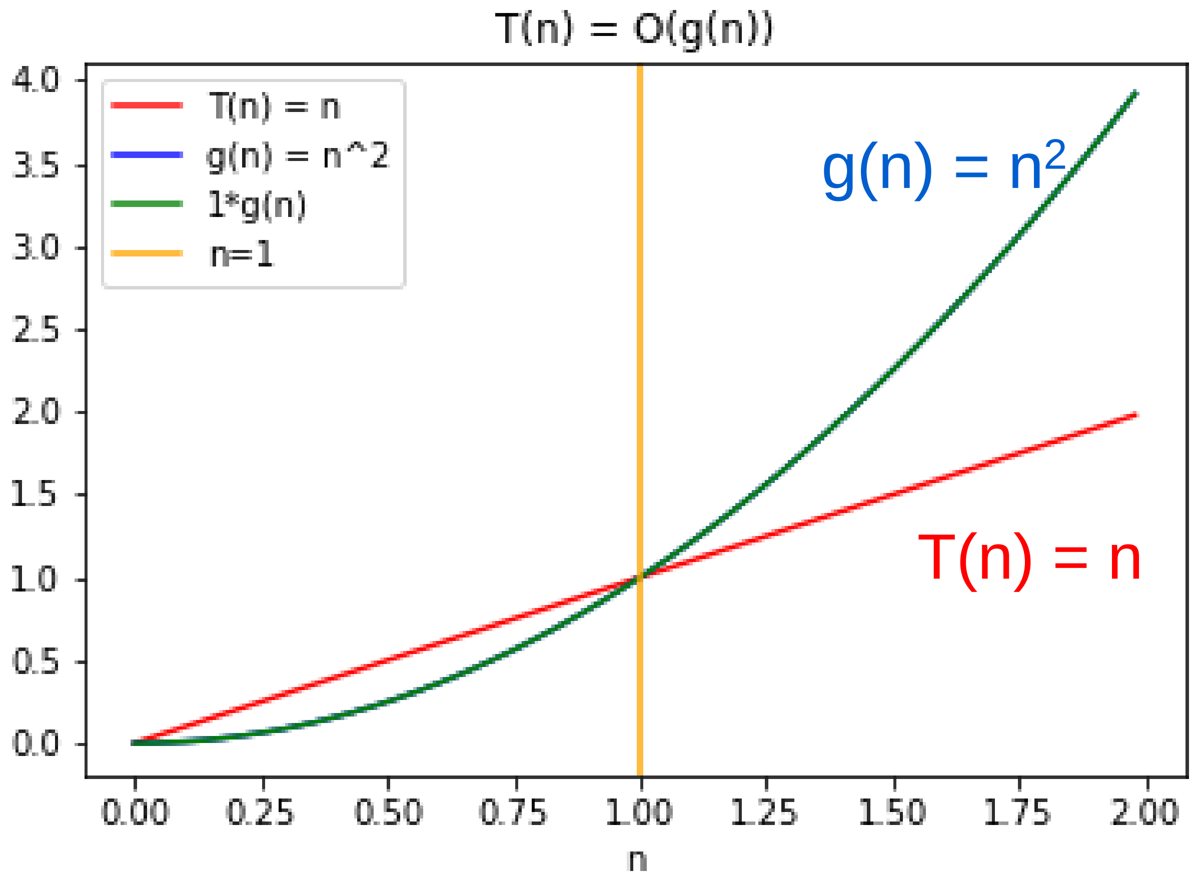
$$\forall n \geq 2,$$
$$0 \leq 2n^2 + 10 \leq 7 \cdot n^2$$

There is not a "unique" choice of c and n_0

Another example:

$$n = O(n^2)$$

$$T(n) = O(g(n))$$
$$\Leftrightarrow$$
$$\exists c, n_0 > 0 \;\; s.t. \;\; \forall n \geq n_0,$$
$$0 \leq T(n) \leq c \cdot g(n)$$



- Choose c = 1
- Choose $n_0 = 1$
- Then

$$\forall n \geq 1,$$
$$0 \leq n \leq n^2$$

This is not tight bound as $n = O(n)$

$\Omega(\ldots)$ means a lower bound

- We say "$T(n)$ is $\Omega(g(n))$" if $T(n)$ grows at least as fast as $g(n)$ as n gets large.
- Formally,

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

$$\Leftrightarrow$$

$$\exists c, n_0 > 0 \;\; s.t. \;\; \forall n \geq n_0,$$

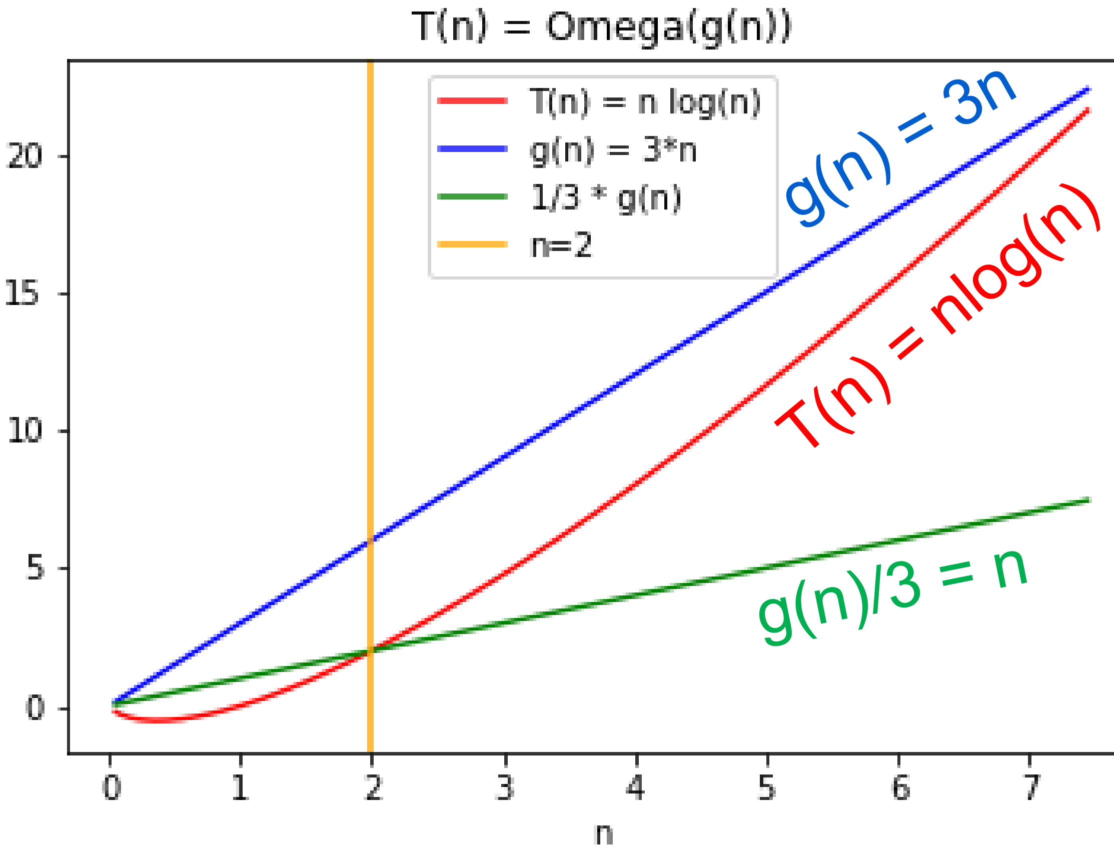
$$0 \leq c \cdot g(n) \leq T(n)$$

Switched these!!

Example

$$T(n) = \Omega(g(n))$$
$$\Leftrightarrow$$
$$\exists c, n_0 > 0 \;\; s.t. \;\; \forall n \geq n_0,$$
$$0 \leq c \cdot g(n) \leq T(n)$$

$$n \log_2(n) = \Omega(3n)$$



- Choose c = 1/3
- Choose $n_0 = 2$
- Then

$$\forall n \geq 2,$$
$$0 \leq \frac{3n}{3} \leq n \log_2(n)$$

Θ(…) means both!

- We say "$T(n)$ is $\Theta(g(n))$" iff both:

$$T(n) = O\big(g(n)\big)$$

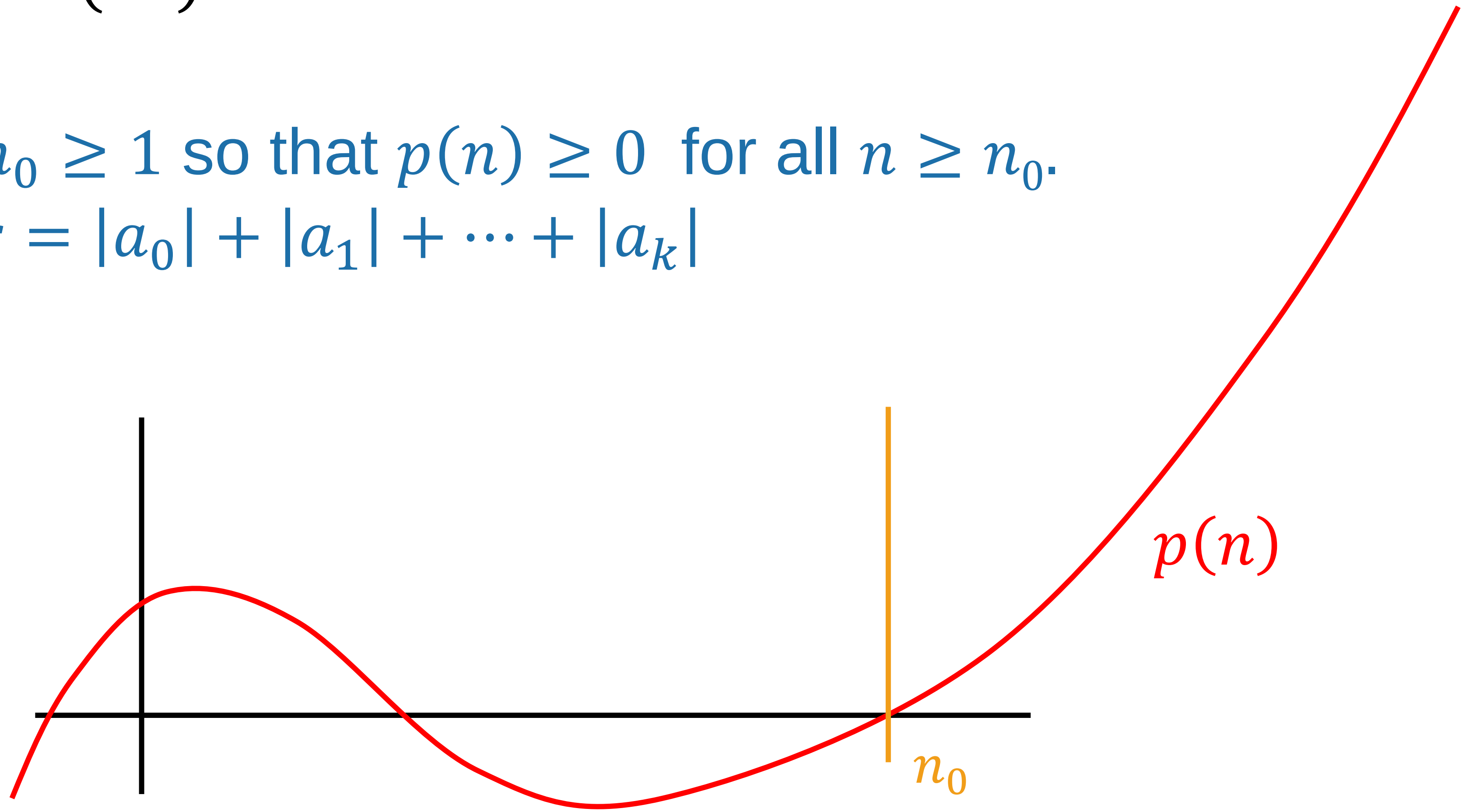
and

$$T(n) = \Omega\big(g(n)\big)$$

Example: polynomials

- Suppose the p(n) is a polynomial of degree k:

  $p(n) = a_0 + a_1 n + a_2 n^2 + \cdots + a_k n^k$ where $a_k > 0$.
- Then $p(n) = O\left(n^k\right)$
- Proof:
  - Choose $n_0 \geq 1$ so that $p(n) \geq 0$ for all $n \geq n_0$.
  - Choose $c = |a_0| + |a_1| + \cdots + |a_k|$



Example: polynomials

- Suppose the p(n) is a polynomial of degree k:

  $p(n) = a_0 + a_1 n + a_2 n^2 + \cdots + a_k n^k$ where $a_k > 0$.
- Then $p(n) = O\left(n^k\right)$
- Proof:
  - Choose $n_0 \geq 1$ so that $p(n) \geq 0$ for all $n \geq n_0$.
  - Choose $c = |a_0| + |a_1| + \cdots + |a_k|$
  - Then for all $n \geq n_0$:
  - $0 \leq p(n) = |p(n)| \leq |a_0| + |a_1| n + \cdots + |a_k| n^k$
  - $\leq |a_0| n^k + |a_1| n^k + \cdots + |a_k| n^k$
  - $= c \cdot n^k$

Definition of c

Because $n \leq n^k$ for $n \geq n_0 \geq 1$.

Example: more polynomials

- For any $k \geq 1, n^k$ is NOT $O(n^{k-1})$.
- Proof:
  - Suppose that it were.
    - Then there is some c, n_0 so that $n^k \leq c \cdot n^{k-1}$ for all $n \geq n_0$
  - Aka, $n \leq c$ for all $n \geq n_0$
  - But that's not true!
  - We have a contradiction!
    - It *can't* be that $n^k = O(n^{k-1})$.

Take-away from examples

- To prove T(n) = O(g(n)), you have to come up with c and n_0 so that the definition is satisfied.

- To prove T(n) is NOT O(g(n)), one way is **proof by contradiction**:
  - Suppose (to get a contradiction) that someone gives you a c and an n_0 so that the definition ***is*** satisfied.
  - Show that this someone must by lying to you by deriving a contradiction.

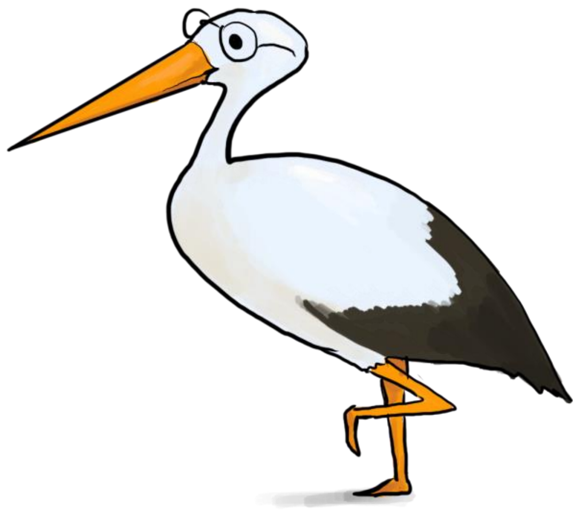
Yet more examples

- $n^3 + 3n = O(n^3 - n^2)$
- $n^3 + 3n = \Omega(n^3 - n^2)$
- $n^3 + 3n = \Theta(n^3 - n^2)$

- 3^n is **NOT** $O(2^n)$
- $\log(n) = \Omega(\ln(n))$
- $\log(n) = \Theta(\ 2^{\log\log(n)}\ )$

remember that $\log = \log_2$ in this class.

Work through these on your own!



Some brainteasers

- Are there functions f, g so that NEITHER $f = O(g)$ nor $f = \Omega(g)$?
- Are there non-decreasing functions f, g so that the above is true?
- Define the n'th fibonacci number by $F(0) = 1$, $F(1) = 1$, $F(n) = F(n-1) + F(n-2)$ for $n > 1$.
  - 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, …

True or false:

- $F(n) = O(2^n)$
- $F(n) = \Omega(2^n)$

Recurrence Relations!

Recurrence Relations!

- How do we calculate the runtime of a recursive algorithm?

Running time of MergeSort

- Let's call this running time T(n), when the input has length n.

- We know that T(n) = O(nlog(n)).

- We also know that T(n) satisfies:

$$T(n) \leq 2 \cdot T\left(\frac{n}{2}\right) + c \cdot n$$

Last time we showed that the time to run MERGE on a problem of size n is at most c*n operations.

```
MERGESORT(A):
    n = length(A)
    if n ≤ 1:
        return A
    L = MERGESORT(A[1:n/2-1])
    R = MERGESORT(A[n/2:n])
    return MERGE(L,R)
```

Recurrence Relations

- $T(n) = 2 \cdot T\left(\frac{n}{2}\right) + c \cdot n$ is a **recurrence relation.**
- It gives us a formula for T(n) in terms of T(less than n)

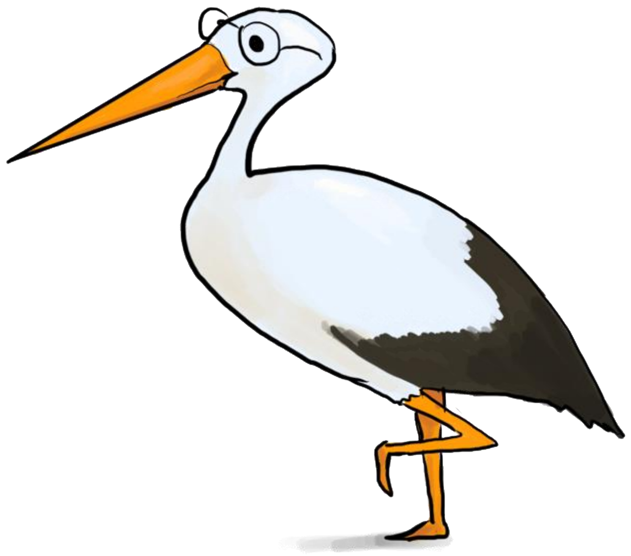
- The challenge:
  Given a recurrence relation for T(n), find a closed form expression for T(n).

- For example, T(n) = O(nlog(n)) in this case

Technicalities I: Base Case

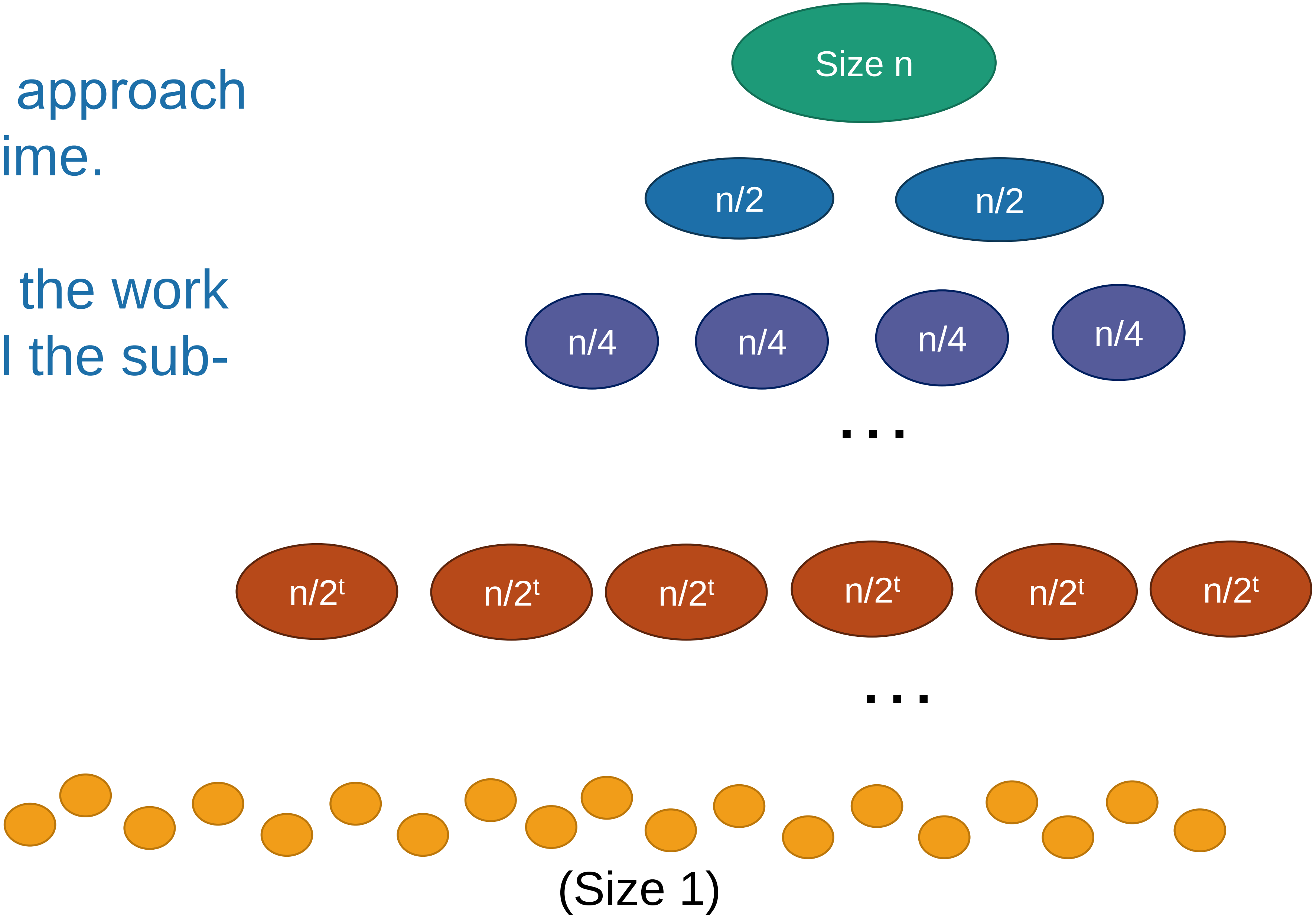
- Formally, we should always have base cases with recurrence relations.
- $T(n) = 2 \cdot T\left(\frac{n}{2}\right) + c \cdot n$ with $T(1) = O(1)$

Why does T(1) = O(1)?



One approach

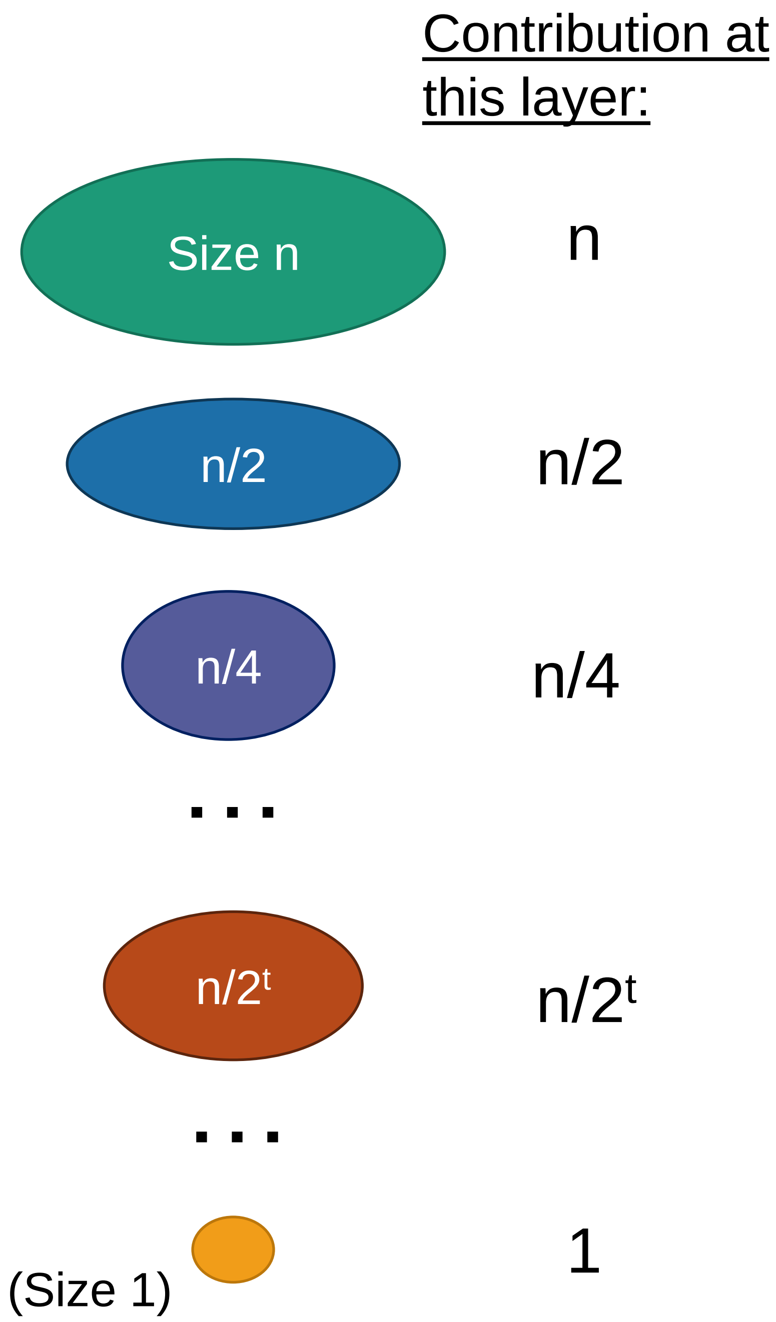
- The “tree” approach from last time.
- Add up all the work done at all the sub-problems.



Another Example

- $T_1(n) = T_1\left(\frac{n}{2}\right) + n, \quad T_1(1) = 1.$
- Adding up over all layers:

$$\sum_{i=0}^{\log(n)} \frac{n}{2^i} = 2n - 1$$



Aside

Finite Geometric Series

To find the sum of a finite geometric series, use the formula,

$$S_n = \frac{a_1(1-r^n)}{1-r}, r \neq 1,$$

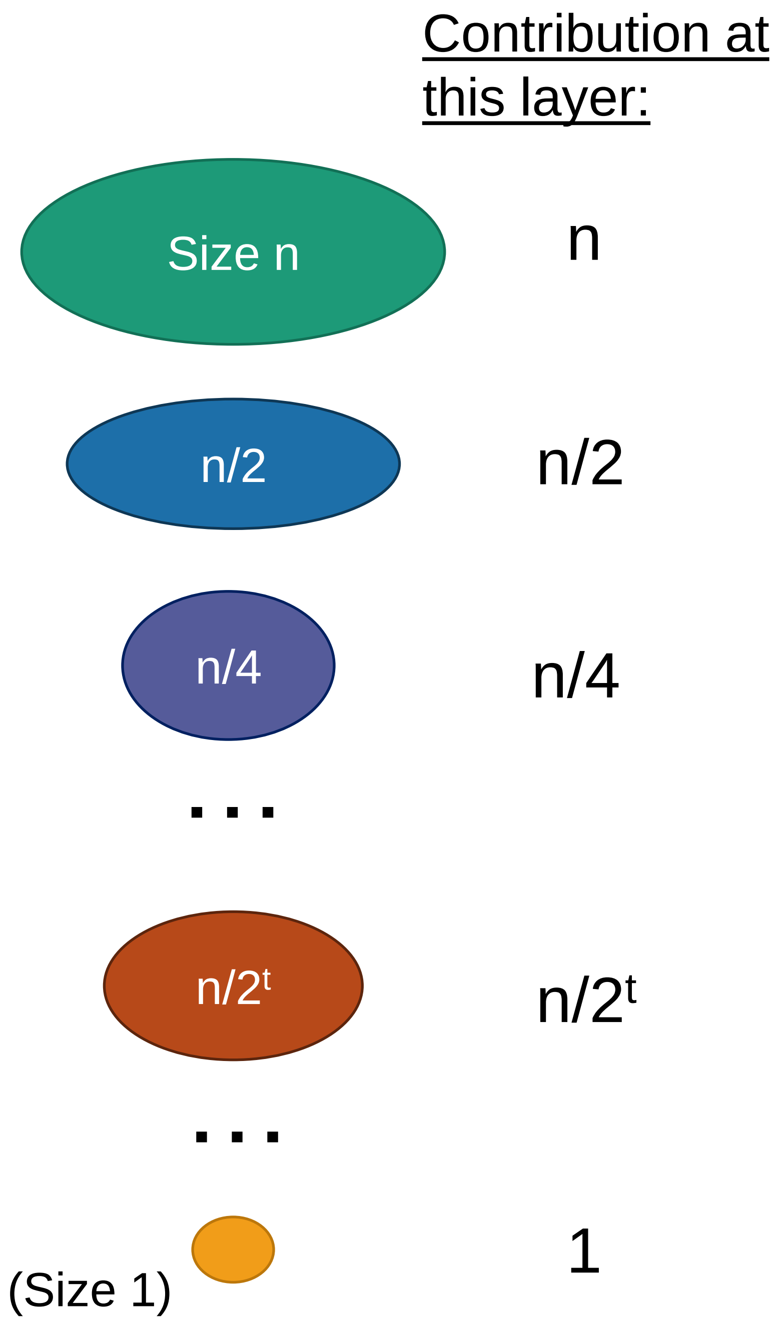
where n is the number of terms, a_1 is the first term and r is the common ratio .

Another Example

- $T_1(n) = T_1\left(\frac{n}{2}\right) + n, \quad T_1(1) = 1.$
- Adding up over all layers:

$$\sum_{i=0}^{\log(n)} \frac{n}{2^i} = 2n - 1$$

So $T_1(n) = O(n)$.

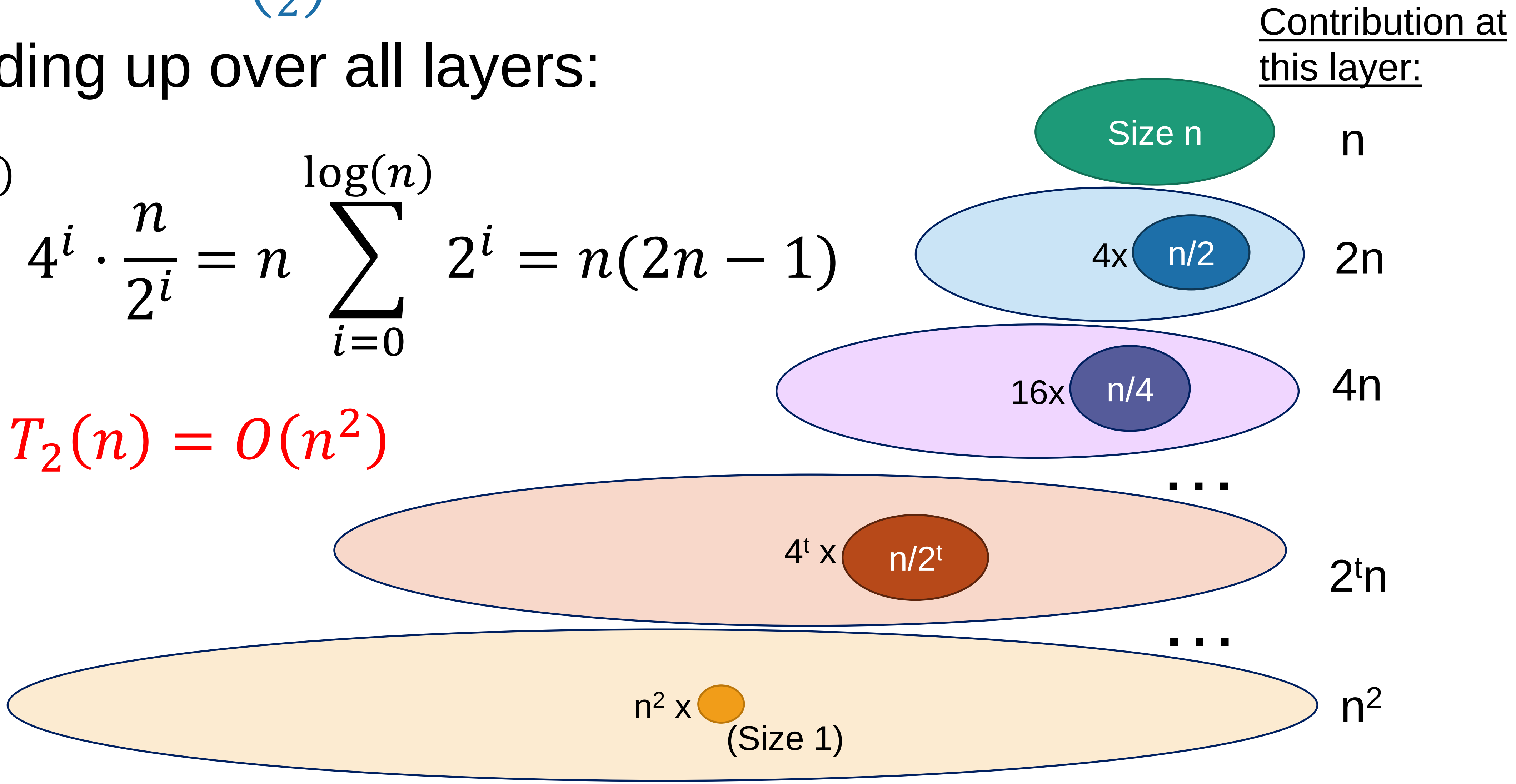


Another Example

- $T_2(n) = 4T_2\left(\frac{n}{2}\right) + n, \quad T_2(1) = 1.$
- Adding up over all layers:

$$\sum_{i=0}^{\log(n)} 4^i \cdot \frac{n}{2^i} = n \sum_{i=0}^{\log(n)} 2^i = n(2n-1)$$

- So $T_2(n) = O(n^2)$



More examples

Recursion 1

- $T(n) = 4\,T(n/2) + O(n)$
- $T(n) = O(\,n^2\,)$

Recursion 2

- $T(n) = 3\,T(n/2) + O(n)$
- $T(n) = O(\,n^{\log_2(3)} \approx n^{1.6}\,)$

Recursion 3

- $T(n) = 2T(n/2) + O(n)$
- $T(n) = O(\,n\log(n)\,)$

Recursion 4

- $T(n) = T(n/2) + O(n)$
- $T(n) = O(\,n\,)$

T(n) = time to solve a problem of size n.

What's the pattern?!?!?!?!

The master theorem

- A formula for many recurrence relations.

Jedi master Yoda

The master theorem (Optional)

We can also take n/b to mean either $\left\lfloor \frac{n}{b} \right\rfloor$ or $\left\lceil \frac{n}{b} \right\rceil$ and the theorem is still true.

- Suppose that $a \geq 1, b > 1, \text{and } d$ are constants (independent of n).
- Suppose $T(n) = a \cdot T\left(\frac{n}{b}\right) + O\left(n^d\right)$. Then

$$T(n) = \begin{cases} O\left(n^d \log(n)\right) & \text{if } a = b^d \\ O\left(n^d\right) & \text{if } a < b^d \\ O\left(n^{\log_b(a)}\right) & \text{if } a > b^d \end{cases}$$

Three parameters:
a : number of subproblems
b : factor by which input size shrinks
d : need to do n^d work to create all the subproblems and combine their solutions.

Many symbols those are….

Examples

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + O(n^d).$$

$$T(n) = \begin{cases} O(n^d \log(n)) & \text{if } a = b^d \\ O(n^d) & \text{if } a < b^d \\ O(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

- Recursion 1
  - $T(n) = 4\,T(n/2) + O(n)$
  - $T(n) = O(n^2)$

a = 4
b = 2
d = 1

$a > b^d$ ✓

- Recursion 2
  - $T(n) = 3\,T(n/2) + O(n)$
  - $T(n) = O(n^{\log_2(3)} \approx n^{1.6})$

a = 3
b = 2
d = 1

$a > b^d$ ✓

- Recursion 3
  - $T(n) = 2T(n/2) + O(n)$
  - $T(n) = O(n\log(n))$

a = 2
b = 2
d = 1

$a = b^d$ ✓

- Recursion 4
  - $T(n) = T(n/2) + O(n)$
  - $T(n) = O(n)$

a = 1
b = 2
d = 1

$a < b^d$ ✓

Acknowledgement

- Stanford University

Thank You