# Advanced Data Structure and Algorithm

Recurrence Relations and how to solve them!

Part-1

# Last two classes….

- Sorting: InsertionSort and MergeSort
- Analyzing correctness of iterative + recursive algs
  - Via "loop invariant" and induction
- Analyzing running time of recursive algorithms
  - By writing out a tree and adding up all the work done.
- How do we measure the runtime of an algorithm?
  - Worst-Case Analysis
  - Big-Oh Notation

# Today

- Recurrence Relations!
  - How do we calculate the runtime of a recursive algorithm?
- The Master Method
  - A useful theorem so we don’t have to answer this question from scratch each time.
- The Substitution Method
  - A different way to solve recurrence relations, more general than the Master Method.

# 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:

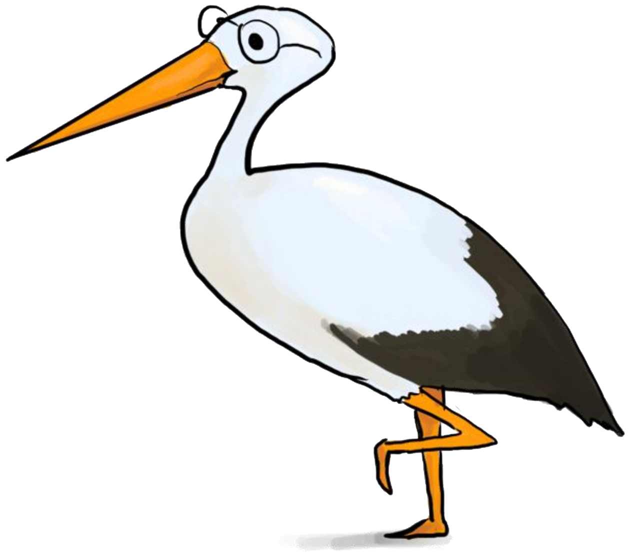
Given a recurrence relation for T(n), find a closed form expression for T(n).

- For example, T(n) = O(nlog(n))

# Technicalities I
## Base Case

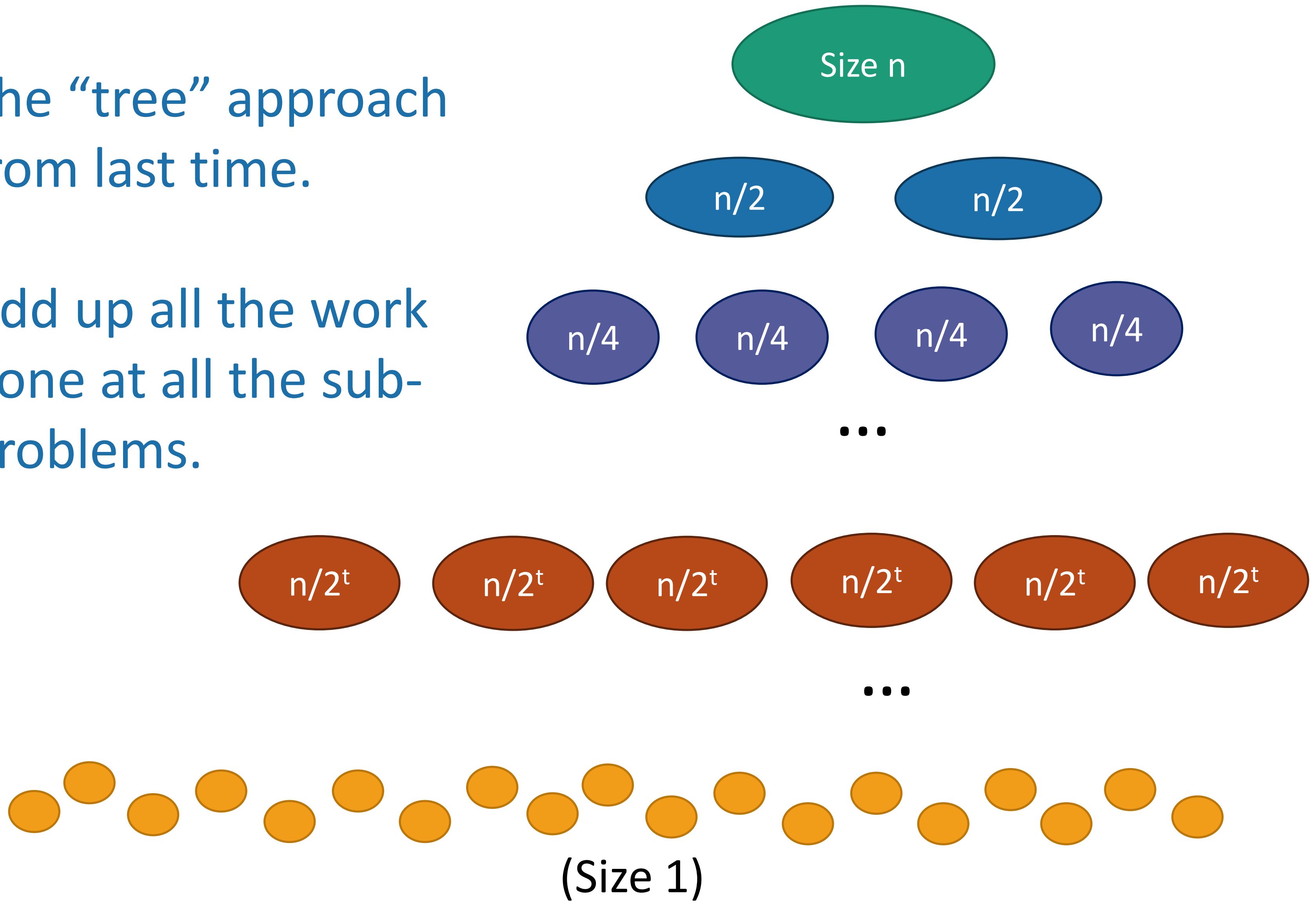
- Formally, we should always have base cases with recurrence relations.
- $T(n) = 2 \cdot T\left(\frac{n}{2}\right) + c \cdot n$ with $T(1) = O(1)$

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



# One approach

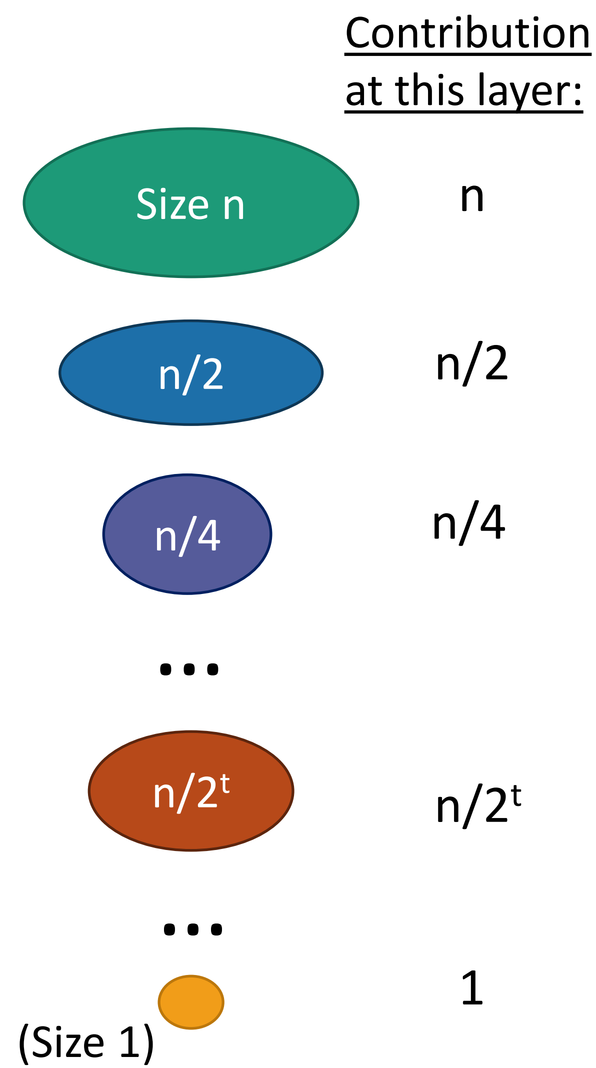
- The "tree" approach from last time.
- Add up all the work done at all the sub-problems.



# 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

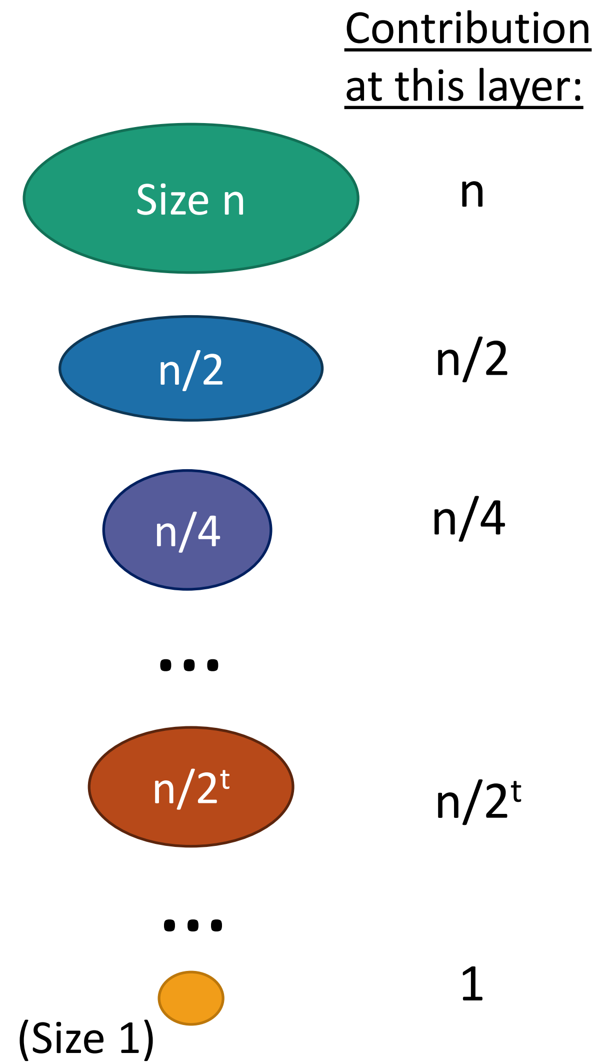
To find the sum of a finite geometric series, use the formula,
$S_n = \frac{a_1(1-r^n)}{1-r}, r \neq 1$ ,
where $n$ is the number of terms, $a_1$ is the first term and $r$ is the common ratio .

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



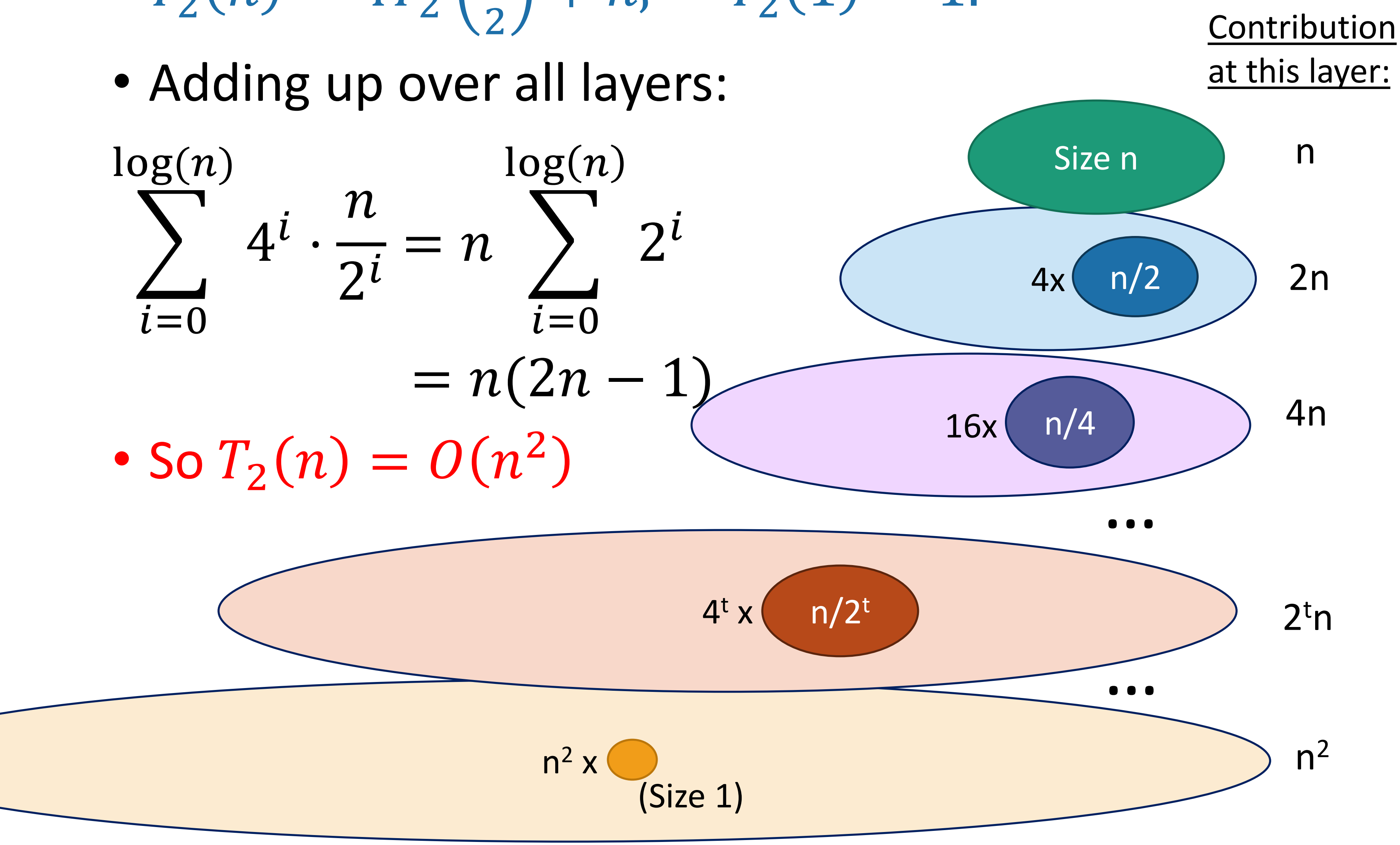
# Another Example

- $T_2(n) = 4T_2\left(\frac{n}{2}\right) + n, \quad T_2(1) = 1.$
- 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

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

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

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

# The master theorem

- A formula for many recurrence relations.
- Proof: “Generalized” tree method.

A useful
formula it is.
You should know,
why it works.

Jedi master Yoda

# The master theorem

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 $a \geq 1, b > 1,$ 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

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

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

- 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

$a < b^d$ ✓

# Proof of the master theorem

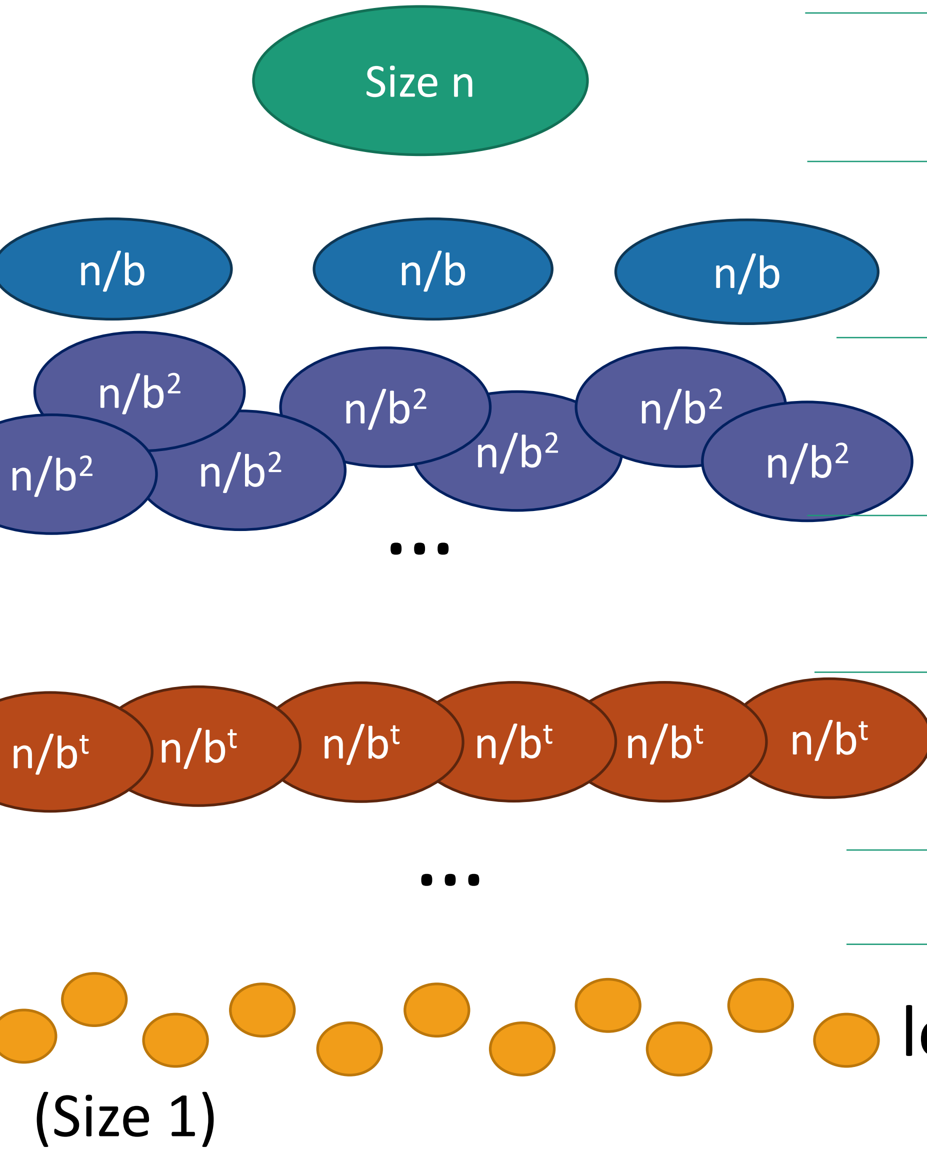
- We'll do the same recursion tree thing we did for MergeSort, but be more careful.
- Suppose that $T(n) = a \cdot T\left(\frac{n}{b}\right) + c \cdot n^d$.

Hang on! The hypothesis of the Master Theorem was that the extra work at each level was O($n^d$). That's NOT the same as work <= $cn^d$ for some constant c.

That's true … we'll actually prove a weaker statement that uses this hypothesis instead of the hypothesis that $T(n) = a \cdot T\left(\frac{n}{b}\right) + O(n^d)$. It's a good exercise to make this proof work rigorously with the O() notation.

# Recursion tree

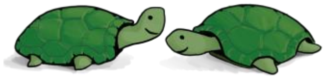
$$T(n) = a \cdot T\left(\frac{n}{b}\right) + c \cdot n^d$$



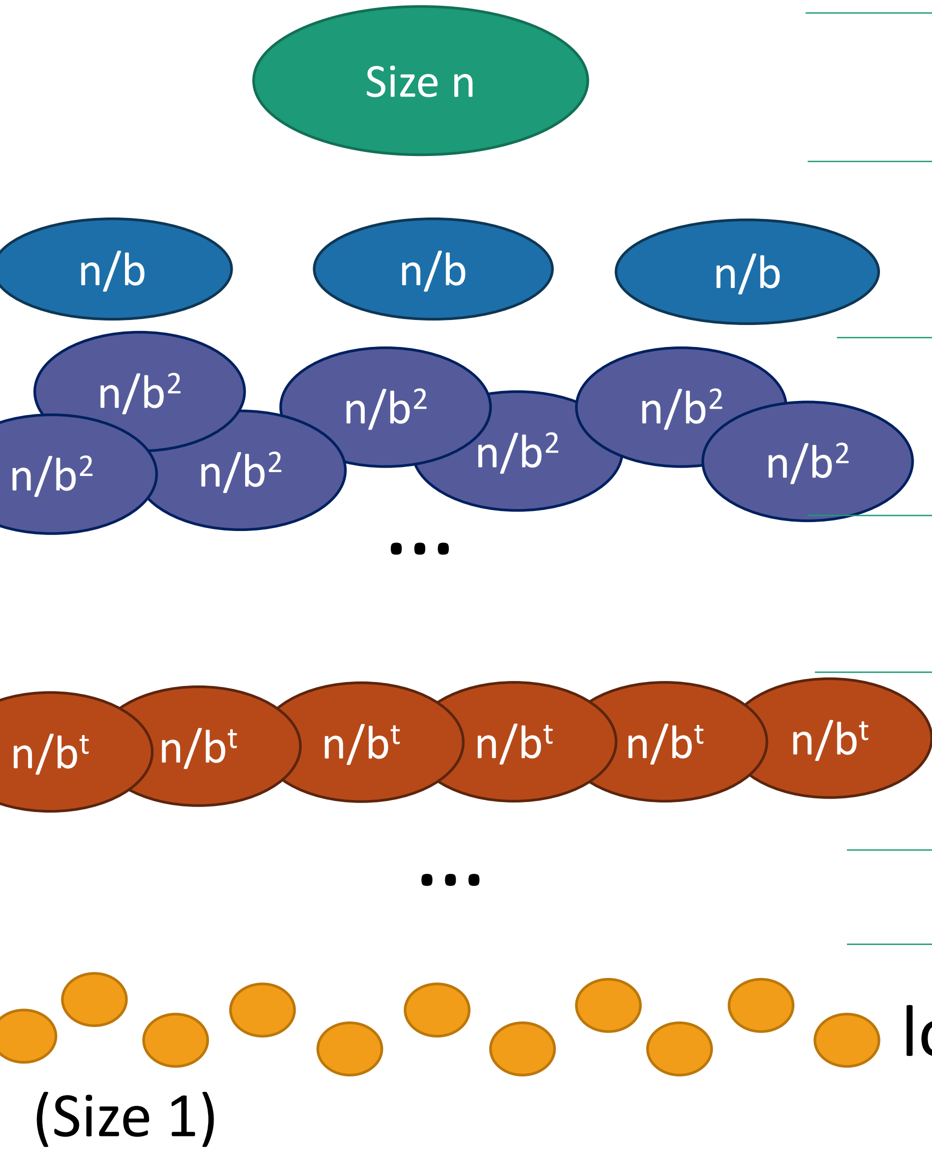

| Level | # problems | Size of each problem | Amount of work at this level |
|---|---|---|---|
| 0 | 1 | n | |
| 1 | a | n/b | |
| 2 | $a^2$ | $n/b^2$ | |
| | | | |
| t | $a^t$ | $n/b^t$ | |
| … | | | |
| $\log_b(n)$ | $a^{\log_b(n)}$ | 1 | |

# Recursion tree

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



Help me fill this in!

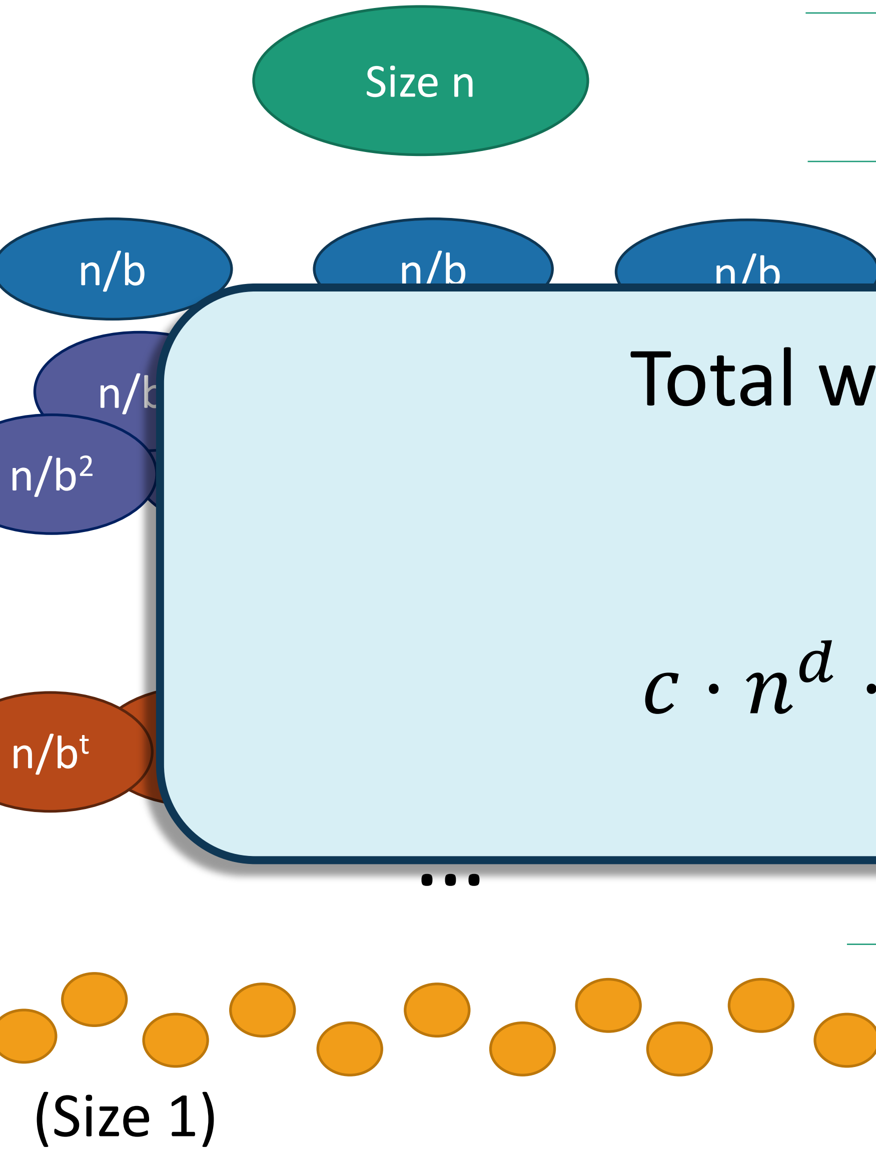



| Level | # problems | Size of each problem | Amount of work at this level |
|---|---|---|---|
| 0 | 1 | n | $c \cdot n^d$ |
| 1 | a | n/b | $ac\left(\frac{n}{b}\right)^d$ |
| 2 | $a^2$ | $n/b^2$ | $a^2c\left(\frac{n}{b^2}\right)^d$ |
| | | | |
| t | $a^t$ | $n/b^t$ | $a^tc\left(\frac{n}{b^t}\right)^d$ |
| … | | | |
| $\log_b(n)$ | $a^{\log_b(n)}$ | 1 | $a^{\log_b(n)}c$ |

(Let's pretend that the base case isT(1) = c for convenience).

# Recursion tree

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




| Level | # problems | Size of each problem | Amount of work at this level |
|---|---|---|---|
| 0 | 1 | n | $c \cdot n^d$ |
| 1 | a | n/b | $ac\left(\frac{n}{b}\right)^d$ |
| ... | | | |
| $\log_b(n)$ | $a^{\log_b(n)}$ | 1 | $a^{\log_b(n)}c$ |

(Let's pretend that the base case isT(1) = c for convenience).

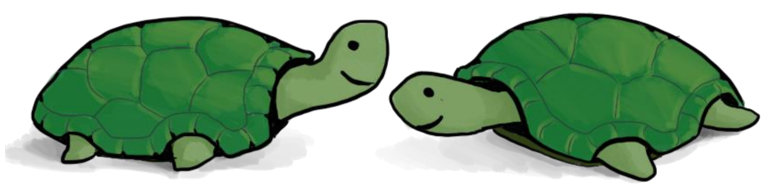
**Total work is at most:**

$$c \cdot n^d \cdot \sum_{t=0}^{\log_b(n)} \left(\frac{a}{b^d}\right)^t$$

# Now let's check all the cases

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

Do the first one!



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

# Case 1: $a = b^d$

- $T(n) = c \cdot n^d \cdot \sum_{t=0}^{log_b(n)} \left(\frac{a}{b^d}\right)^t$

$$= c \cdot n^d \cdot \sum_{t=0}^{log_b(n)} 1$$

$$= c \cdot n^d \cdot (\log_b(n) + 1)$$

$$= c \cdot n^d \cdot \left(\frac{\log(n)}{\log(b)} + 1\right)$$

$$= O\left(n^d \log(n)\right)$$

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

# Case 2: $a < b^d$

- $T(n) = c \cdot n^d \cdot \sum_{t=0}^{log_b(n)} \left(\frac{a}{b^d}\right)^t$ ← Less than 1!

# Aside: Geometric sums

- What is $\sum_{t=0}^{N} x^t$?
- You may remember that $\sum_{t=0}^{N} x^t = \frac{x^{N+1}-1}{x-1}$ for $x \neq 1$.
- Morally:

$$x^0 + x^1 + x^2 + x^3 + \cdots + x^N$$

If $0 < x < 1$, this term dominates.

(If $x = 1$, all terms the same)

If $x > 1$, this term dominates.

$$1 \leq \frac{1 - x^{N+1}}{1 - x} \leq \frac{1}{1 - x}$$

(Aka, doesn't depend on N).

$$x^N \leq \frac{x^{N+1}-1}{x - 1} \leq x^N \cdot \left(\frac{x}{x - 1}\right)$$

(Aka, $\Theta(x^N)$ if x is constant and N is growing).

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

# Case 2: $a < b^d$

- $T(n) = c \cdot n^d \cdot \sum_{t=0}^{log_b(n)} \left(\frac{a}{b^d}\right)^t$ ← Less than 1!

  $= c \cdot n^d \cdot [\text{some constant}]$

  $= O\left(n^d\right)$

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

# Case 3: $a > b^d$

- $T(n) = c \cdot n^d \cdot \sum_{t=0}^{\log_b(n)} \left(\frac{a}{b^d}\right)^t$

$$= O\left(n^d \left(\frac{a}{b^d}\right)^{\log_b(n)}\right)$$

$$= O\left(n^{\log_b(a)}\right)$$

Larger than 1!

Convice yourself that this step is legit!

# Now let's check all the cases

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

# Even more generally, for T(n) = aT(n/b) + f(n)...

**Theorem 3.2** (Master Theorem). *Let* $T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$ *be a recurrence where* $a \geq 1$, $b > 1$. *Then,*

- *If* $f(n) = O\left(n^{\log_b a - \epsilon}\right)$ *for some constant* $\epsilon > 0$, $T(n) = \Theta\left(n^{\log_b a}\right)$.
- *If* $f(n) = \Theta\left(n^{\log_b a}\right)$, $T(n) = \Theta\left(n^{\log_b a} \log n\right)$.
- *If* $f(n) = \Omega\left(n^{\log_b a + \epsilon}\right)$ *for some constant* $\epsilon > 0$ *and if* $af(n/b) \leq cf(n)$ *for* $c < 1$ *and all sufficiently large* $n$, *then* $T(n) = \Theta(f(n))$.

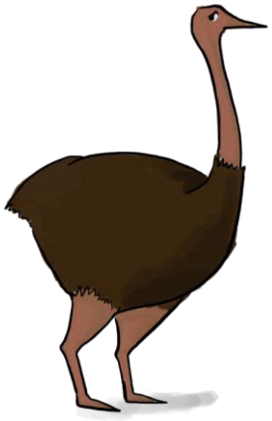


Figure out how to adapt the proof we gave to prove this more general version!

# Acknowledgement

- Stanford University