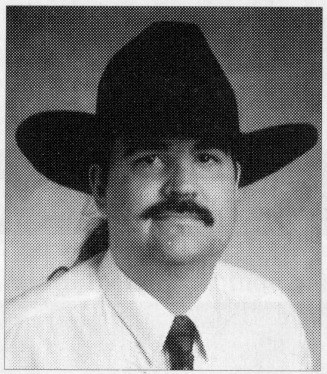
CS 3510 Algorithms: This Class + Big-O Notation

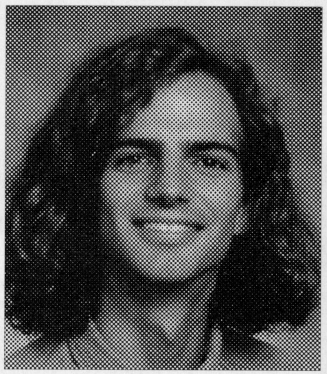
Aaron Hillegass
Georgia Tech

I Was In Over My Head

Eiffel

Eiffel asserts itself





Rock Howard Aaron Hillegass

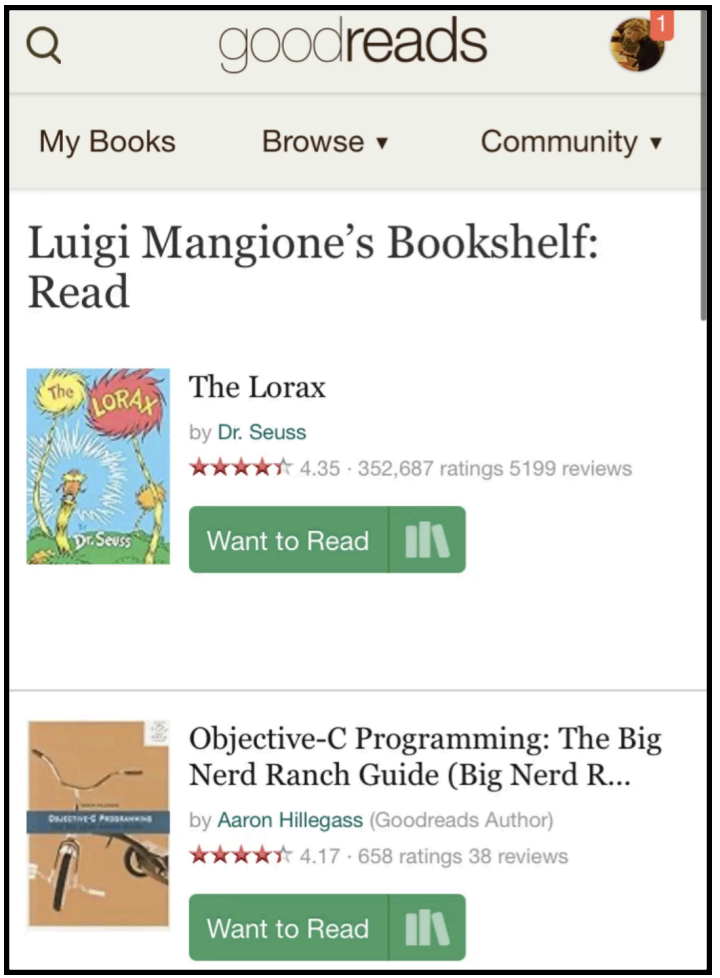
FEEDBACK FROM REAL-WORLD projects seems to indicate that the most immediate benefit of object technology (OT) is the opportunity for improvements in code quality. Other benefits, such as reduced maintenance costs

clients and suppliers within an object-oriented design. This approach is called "design by contract." In "design by contract," each client-supplier interaction has a clear and verifiable contract. As this concept is integrated into more O-O design method

For any routine, the preconditions are listed after the keyword require. The postconditions are listed after the word ensure. For example, in the class STRING, the procedure append takes another string as a supplied argument and appends it to the re

What Came Next

- Worked on Wall Street
- Worked with Steve Jobs
- Part of the NeXT/Apple merger
- Founded Big Nerd Ranch
- Published four books
- Sold Big Nerd Ranch
- Went back to school at Georgia Tech



This Class

Study theoretical properties of algorithms:

- Correctness
- Computational complexity
- Memory requirements

Psuedocode. Mathematical Proofs. No actual coding.

4 exams, 18% each = 72%

Homeworks = 20%

In-class worksheets: 8%

Office Hours

Me: Room 3 in Institut Lafayette. My hours will be Tuesday 1:30-3pm and Wednesday 9-11am.

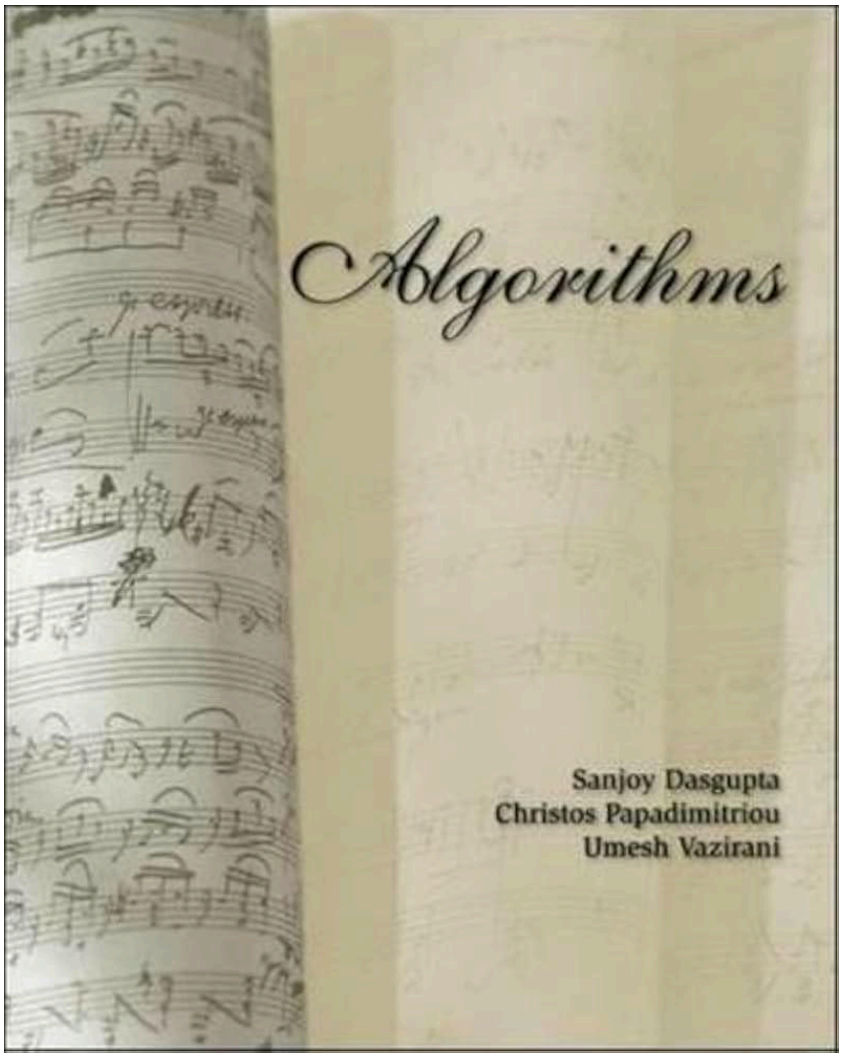
TA: Ameen Agbaria, Thursday from 3:00-4:30. Location TBD

Why?

- The best need to know if their solutions will scale.
- This class will get you through a technical interview.
- The algorithms are...fun!

Parts

- Divide and Conquer
- Graph Algorithms
- Dynamic Programming
- NP-Completeness



Correctness

Always the first: *Does the algorithm give correct answers?*

- **Sound**: If it returns an answer, the answer is correct.
- **Complete**:
  - If a correct answer exists, the algorithm finds one in finite time.
  - If a correct answer doesn't exist the algorithm reports this in finite time.

Then we talk about speed/space.

What we want to know:

How long will this algorithm take?

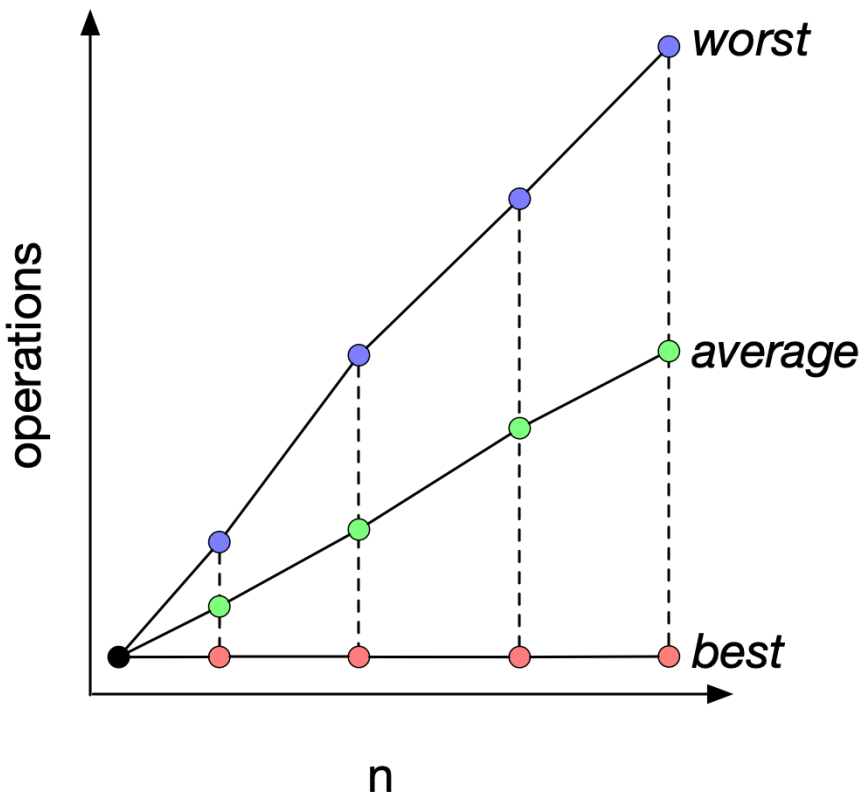
Linear scan of an array


```
bool contains(uint16_t *buf, usize n, uint16_t x) {
    for (usize i = 0; i < n; i++) {
        if buf[i] == x {
            return true;
        }
    }
    return false;
}
```

- From RAM to cache: 12 clock cycles (brings 64 bytes)
- Cache to register: 1 clock cycle
- Comparing two registers: 1 clock cycle

Linear scan of array

- Best case?
- Average case?
- Worst case?



Problems with counting clock cycles

- Hardware-dependent
- Hassle
- The differences we are looking for are *BIG*

What we are satified knowing:

How much longer will this algorithm take if we double the size of the data?

Find in an unsorted array

If n doubles, the time doubles.

$O(n)$ means “In the worst case, time scales linearly with n”

```
bool contains(uint16_t *buf, usize n, uint16_t x) {
    for (usize i = 0; i < n; i++) {
        if buf[i] == x {
            return true;
        }
    }
    return false;
}
```

Intuition

$O(1)$: Double n? Time stays the same.

$O(\log n)$: Double n? Time increases by a constant amount.

$O(n)$: Double n? Time doubles.

$O(n^2)$: Double n? Time $\times 4$

$O(n^3)$: Double n? Time $\times 8$

$O(2^n)$: Increase n by 1? Time doubles.

$O(n^n)$: Increase n by 1? Weep.

Definition of Big-O

$$f(n) \in O(g(n))$$

means:

There exists a $C > 0$ and an n_0 such that

$$f(n) \leq Cg(n) \text{ for all } n \geq n_0$$

Properties

Multiplying by a constant doesn't matter.

In sum, only fastest growing ("dominating") term matters.

Note: the base of the log doesn't matter: $\log_a n = \left(\frac{1}{\log_b a}\right)(\log_b n)$

Exponentials always dominate polynomials

Polynomials always dominate logs

$O(1) < O(\log n) < O(n) < O(n^2) < O(2^n) < O(3^n) < O(n!) < O(n^n)$

Big O vs Big Θ vs Big Ω

$$f(n) \in O(g(n))$$

There $\exists C, n_0$ such that

$$f(n) \leq Cg(n)$$

for all $n > n_0$

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

There $\exists C_1, C_2, n_0$ such that

$$C_1 g(n) \leq f(n) \leq C_2 g(n)$$

for all $n > n_0$

Or: $g(n) \in O(f(n))$ and $f(n) \in O(g(n))$

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

There $\exists C, n_0$ such that

$$f(n) \geq Cg(n)$$

for all $n > n_0$

Or: $g(n) \in O(f(n))$

Fibonacci Numbers

0,1,1,2,3,5,8,13,...

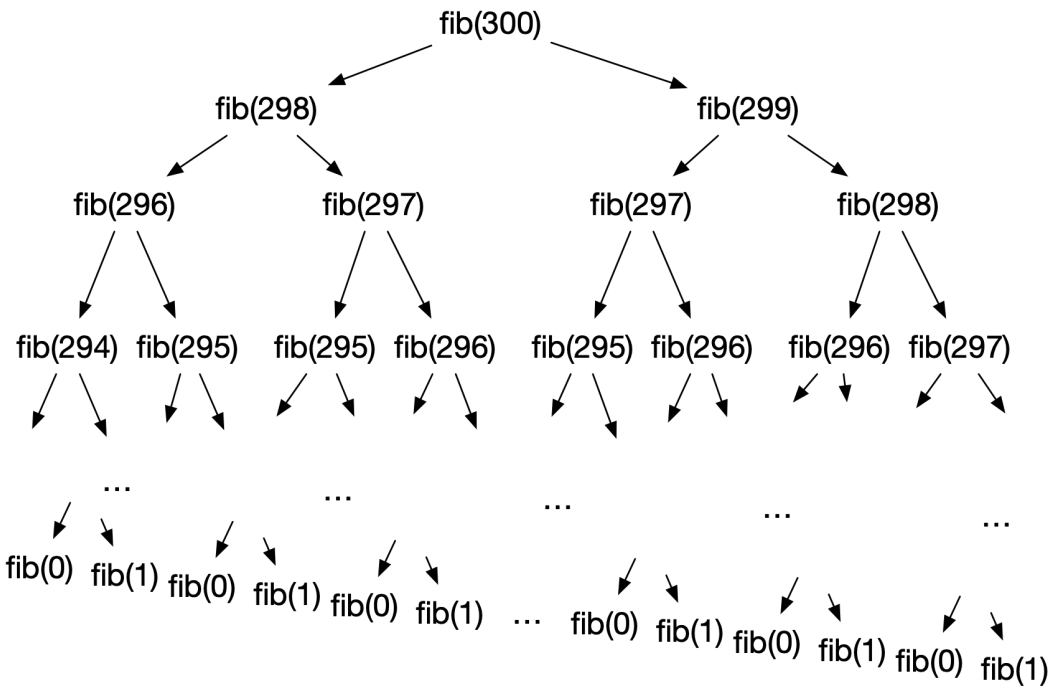
Inductive definition:

- $F_n = F_{n-1} + F_{n-2}$
- $F_0 = 0, F_1 = 1$

Fibonacci Numbers Recursively


```
1 define fib(n):
2     if n == 0 or n == 1:
3         return n
4     return fib(n-1) + fib(n-2)
```

What can we say about $T(n)$, the number of steps to compute F_n?



Fibonacci Numbers Recursively

$$T(n) = T(n-1) + T(n-2) + C_1$$

$$T(0) = T(1) = C_0$$

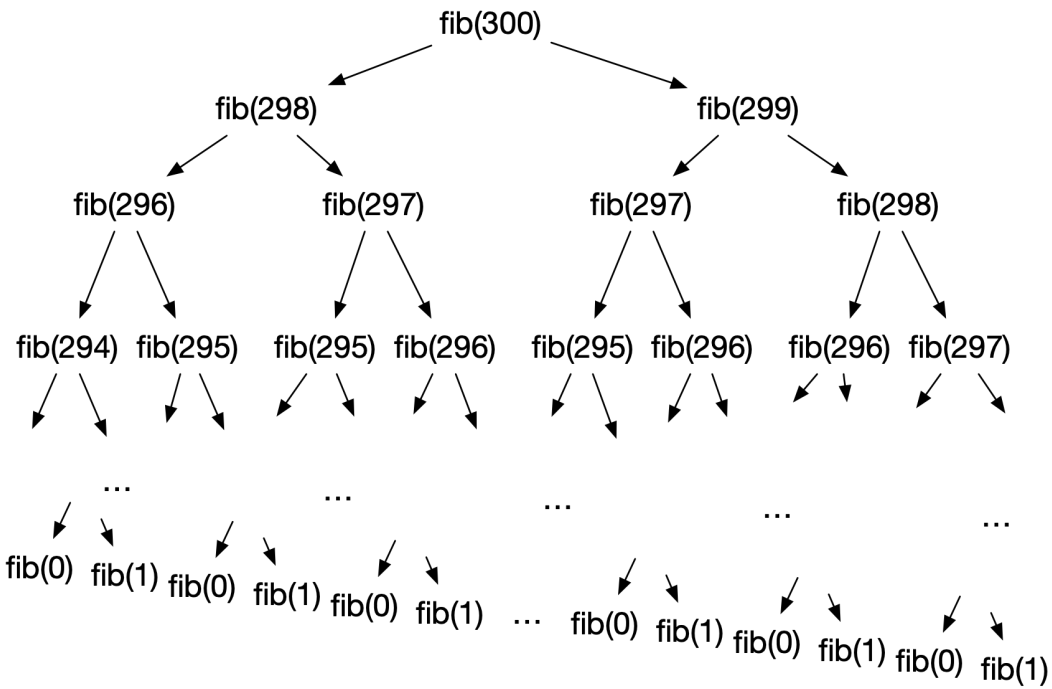
For big enough n:

$T(n-1) > T(n-2) + C_1 > T(n-2) - C_1$

$2T(n-1) > T(n) > 2T(n-2)$

$2^n > T(n) > 2^{\frac{n}{2}}$

$$2^n > T(n) > \left(\sqrt{2}\right)^n$$



Fibonacci Numbers Recursively

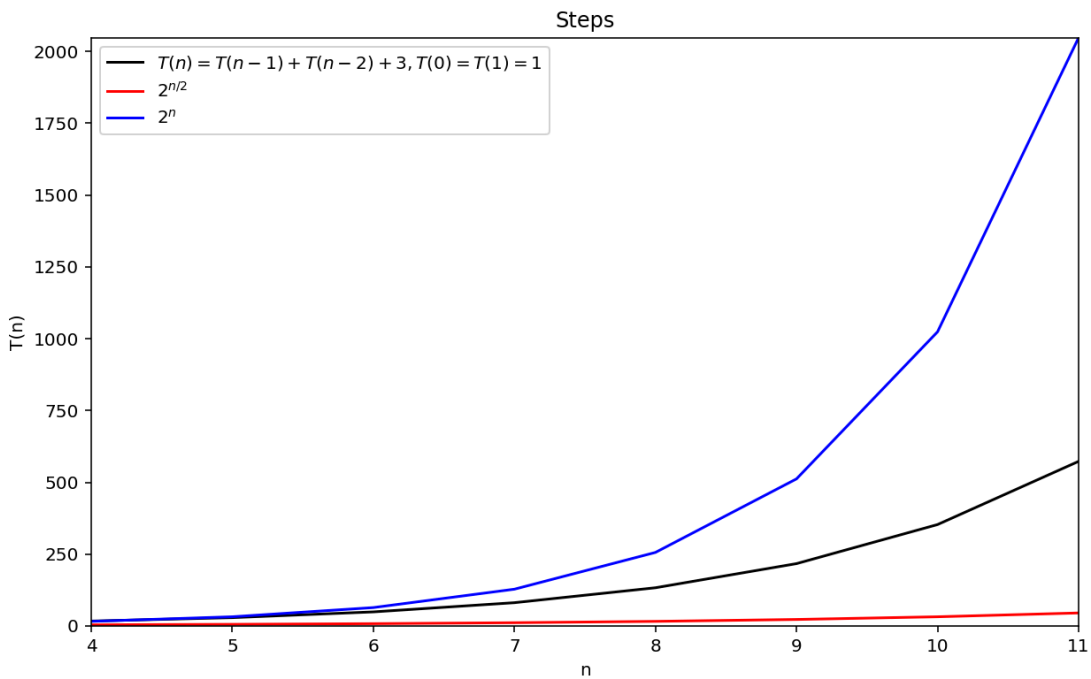
$$T(n) = T(n-1) + T(n-2) + C_1$$
$$T(0) = T(1) = C_0$$

For big enough n:

$$2^n > T(n) > \left(\sqrt{2}\right)^n$$

$$\text{fib}(n) \in O(2^n)$$

$$\text{fib}(n) \in \Omega\left(\left(\sqrt{2}\right)^n\right)$$



Fibonacci Numbers Recursively

$$T(n) = T(n-1) + T(n-2) + C_1$$

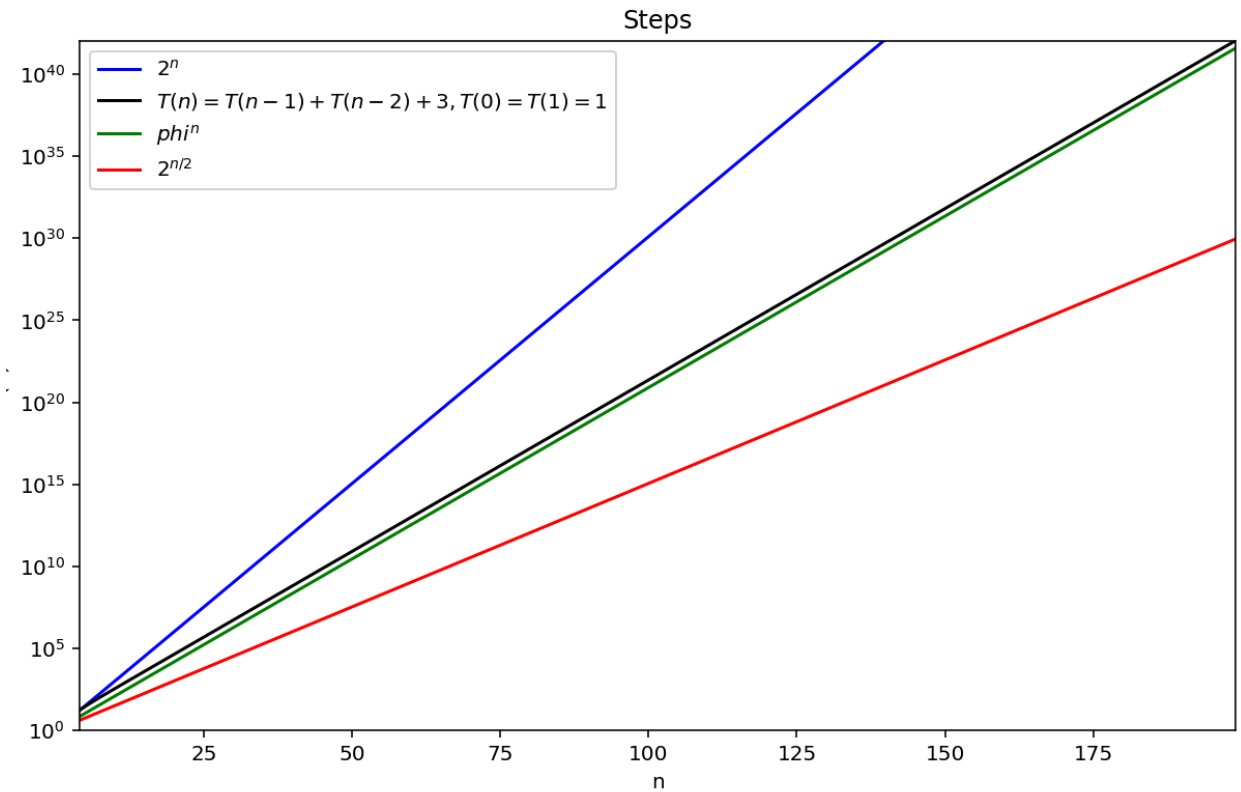
$$T(n) = a^n$$

$$a^n = a^{n-1} + a^{n-2} + C_1$$

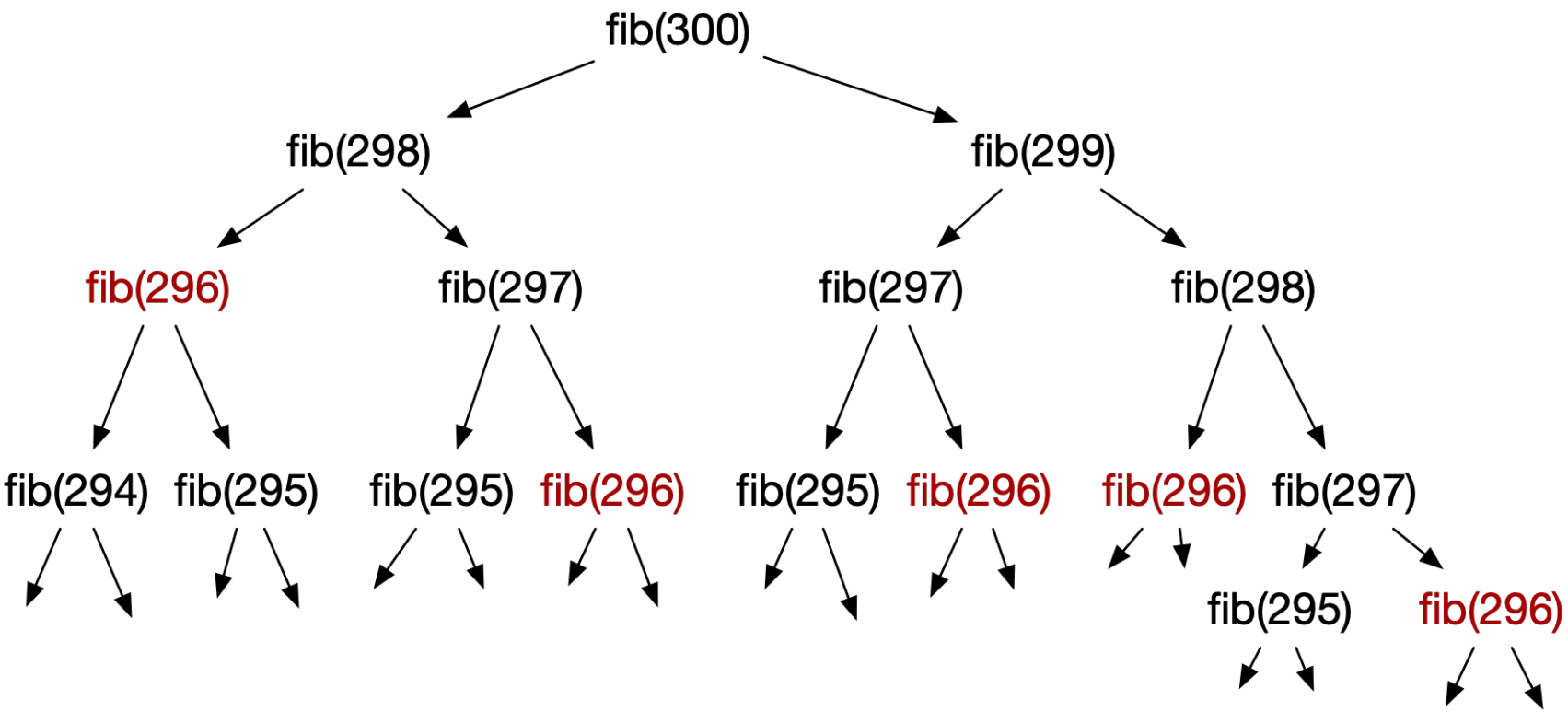
$$a^2 = a + 1 + \frac{C_1}{a^{n-2}}$$

$$a = \frac{1+\sqrt{5}}{2} = \varphi \approx 1.618$$

$$\text{fib(n)} \in \Theta(\varphi^n)$$

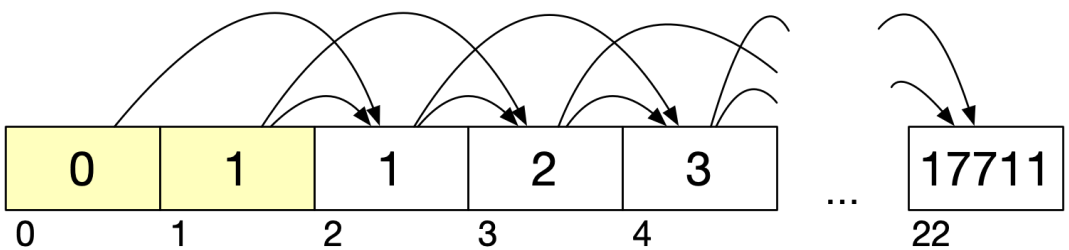


Dumb Recursion



Fibonacci Numbers: Dynamic Programming


```
1 define fib2(n):
2     a := array[0.. = n]
3     a[0] := 0, a[1] := 1
4     for i from 2 to n:
5         a[i] := a[i − 1] + a[i − 2]
6     return a[n]
```



$\text{fib2}(n) \in O(n)$

Fibonacci Numbers: Using Matrices

If you know F_k and F_{k+1} you can compute F_{k+2} using a matrix:

$$\begin{pmatrix} F_{k+2} \\ F_{k+1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} F_{k+1} \\ F_k \end{pmatrix}$$

Starting with $F_0 = 0$ and $F_1 = 1$, we can compute F_{n+1} and F_n for any n:

$$\begin{pmatrix} F_{n+1} \\ F_n \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

How does matrix exponentation scale?

Exponentiation by Squaring

You want to know a^{20}?

What is 20 in binary? 10100

So $a^{20} = a^{16}a^4$

Exponentiation by n is $O(\log n)$.

Exponentiation Diagonalizable Matrices

Matrix B is *diagonalizable* if it can be written as $B = PDP^{\{-1\}}$ where D is a diagonal matrix.

Assume B is diagonalizable.

$B^3 = (PDP^{-1})(PDP^{-1})(PDP^{-1}) = PD^3P^{-1}$

Fibonacci: Binet's Formula

$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$ is diagonalizable:

$$P = \begin{pmatrix} \varphi & -\varphi^{-1} \\ 1 & 1 \end{pmatrix}$$

$$D = \begin{pmatrix} \varphi & 0 \\ 0 & -\varphi^{-1} \end{pmatrix}$$

$$P^{-1} = \left(\frac{1}{\sqrt{5}}\right)\begin{pmatrix} 1 & \varphi^{-1} \\ -1 & \varphi \end{pmatrix}$$

(Reminder: $\varphi = \frac{1+\sqrt{5}}{2} \approx 1.6180339887$)

$$\begin{pmatrix} F_{n+1} \\ F_n \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n \begin{pmatrix} 1 \\ 0 \end{pmatrix} = PD^nP^{-1}\begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

$$F_n = \left(\frac{1}{\sqrt{5}}\right)\left(\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right)$$

Comparison

- Recursive: $O(\varphi^n)$
- Dynamic Programming: $O(n)$
- Matrix Exponentiation: $O(\log n)$
- Binet's Formula: $O(1)$

Questions we won't ask in this class


```
bool contains(uint16_t *buf, usize n, uint16_t x) {
    for (usize i = 0; i < n; i++) {
        if buf[i] == x {
            return true;
        }
    }
    return false;
}
```

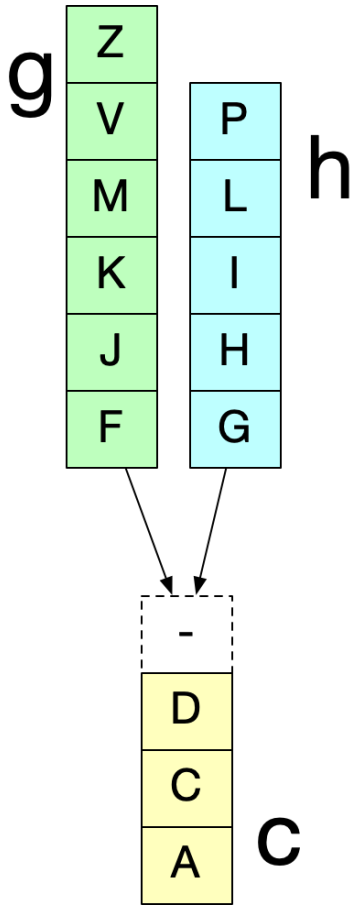
- Can it be parallelized? Vectorized?
- Locality of reference? Cache hits?
- Could we speed it up via prefetching?
- How hard would it be to make the data persistent?

Why are computer science professors so obsessed with ***sorting***!?

Comparison Sorting Algorithms

- Serial? $O(n \log n)$
- *Stable*: equal values stay in the same order
- $O(1)$ memory usage for in place, $O(n)$ otherwise
- Most good algorithms are $O(n)$ for best case

```
1  define Merge(g, h) -> sorted list:
2      c := new List
3      while g and h are non-empty:
4          if head(g) ≤ head(h):
5              c.append(g.pop_head())
6          else
7              c.append(h.pop_head())
8      if g is not empty:
9          Append everying in g to c
10     if h is not empty:
11         Append everything in h to c
12     return c
```

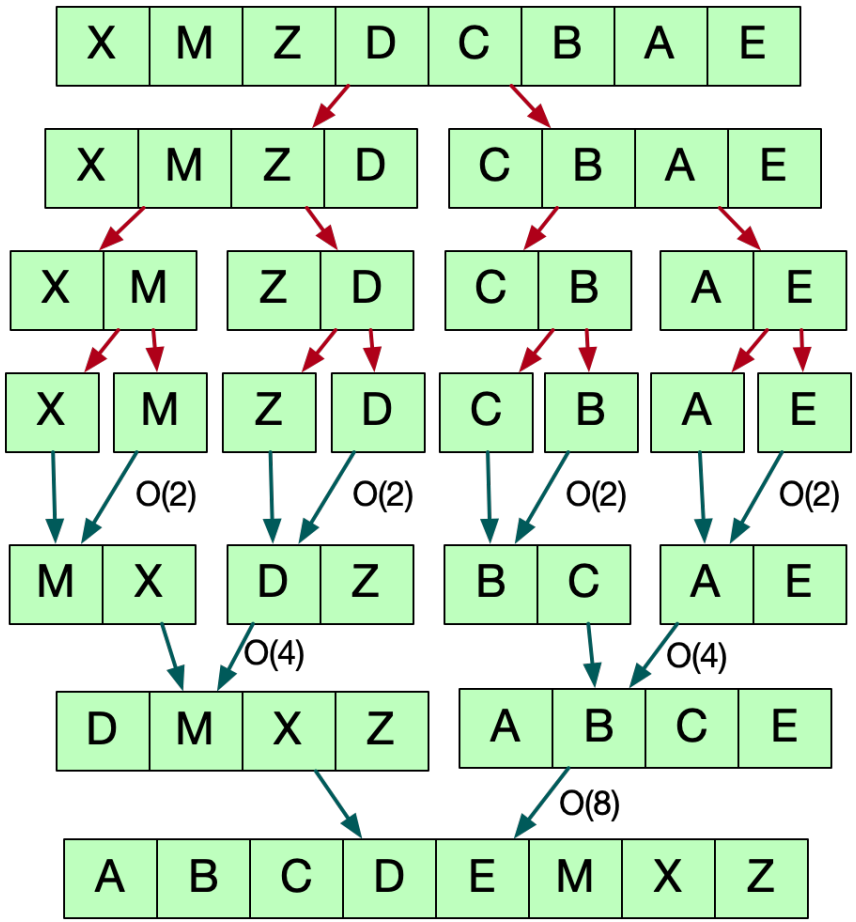


$O(n)$ where n is $\text{len}(g) + \text{len}(h)$

MergeSort

```
1 define MergeSort(g) → sorted list:
2     n := length of g
3     if n == 0 or n == 1, return g
4     s = ⌊n/2⌋
5     left := MergeSort(prefix of g of length s)
6     right := MergeSort(rest of  g)
7     return Merge(left, right)
```

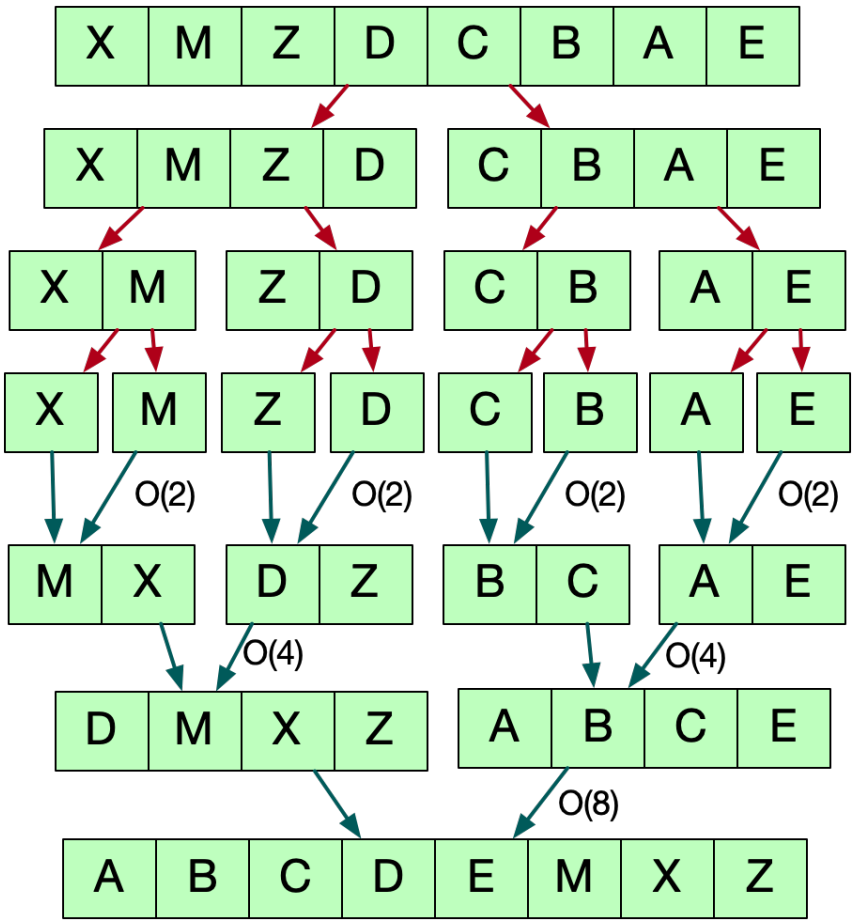
Time complexity?



MergeSort Time Complexity

- How many layers of merges? $\log_2(n)$
- Complexity of each layer? $O(n)$

Total time complexity: $O(n \log n)$



Can we come up with a general rule for the time complexity of divide-and-conquer algorithms?

MergeSort Revisited

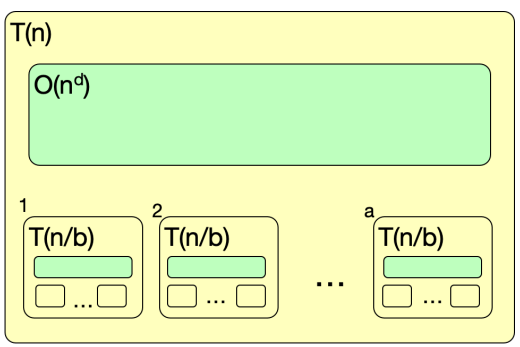
- MergeSort divides into two MergeSorts on half n
- The cost of this is the $O(n)$ merge.

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n^1)$$

Generally:

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

```
1 define MergeSort(g) -> sorted list:
2 |  n := length of g
3 |  if n == 0 or n == 1:
4 |  |  return g
5 |  s = ⌊n/2⌋
6 |  left := MergeSort(g[1..s])
7 |  right := MergeSort(g[s + 1..n])
8 |  return Merge(left, right)
```



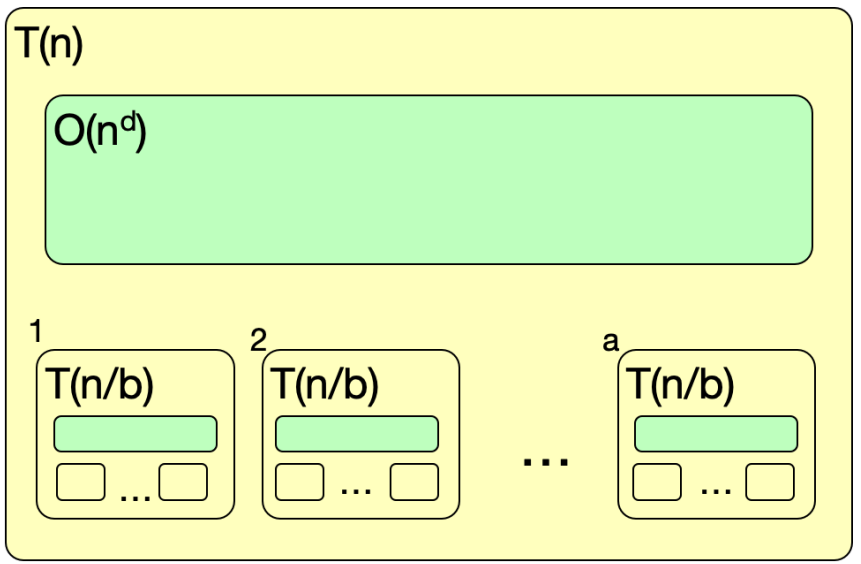
Master Theorem Stated

Given:

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

Then:

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



Applied to MergeSort

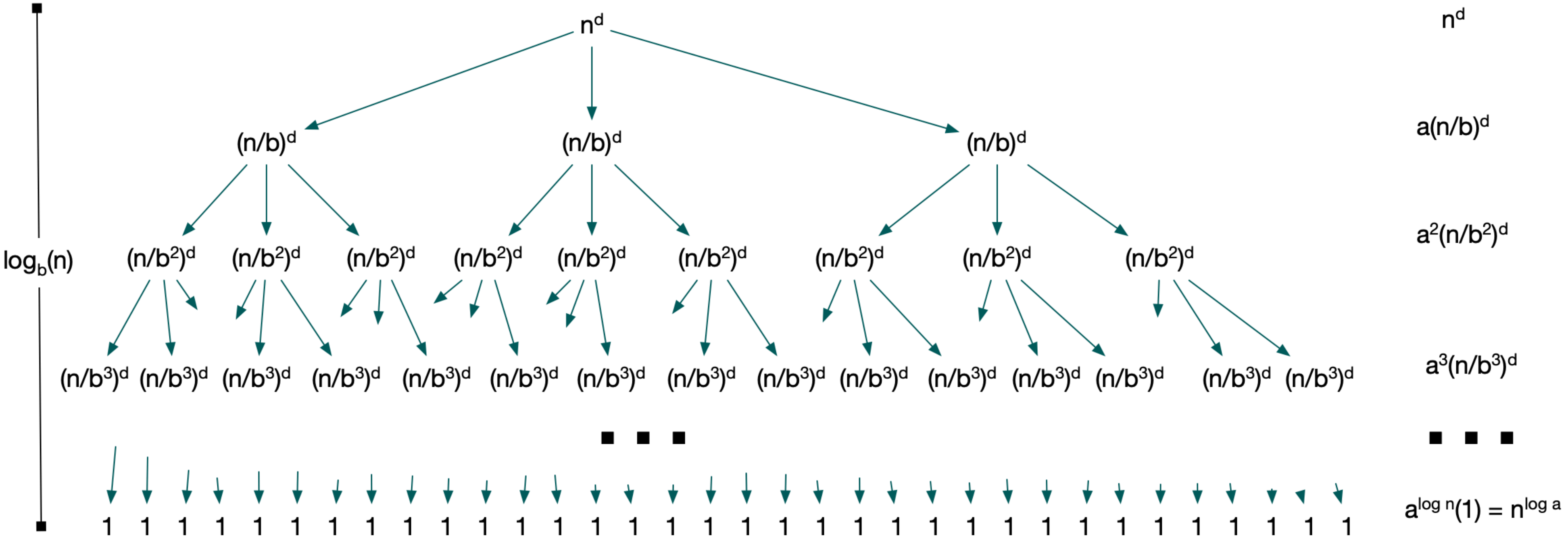
$T(n) = aT\left(\frac{n}{b}\right) + O\left(n^d\right)$

For MergeSort: $a = 2$, $b = 2$, $d = 1$

Case: $d = \log_b a$

MergeSort has $O(n^1 \log n)$ or just $O(n \log n)$

Proof: Consider $T(n) = aT\left(\frac{n}{b}\right) + O\left(n^d\right)$



$$T(n) = \sum_{i=0}^{\log_b n} a^i \left(\frac{n}{b^i}\right)^d + O\left(n^{\log_b a}\right)$$

Why $a^{\log_b n} = n^{\log_b a}$

Remember change of base formula:

$$\log_x y = \frac{\log_c y}{\log_c x}$$

Thus

$$\log_b a = \frac{\log_a a}{\log_a b} = \frac{1}{\log_a b}$$

Thus

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

Simplify

$$T(n) = \sum_{i=0}^{\log_b n} a^i \left(\frac{n}{b^i}\right)^d + O\left(n^{\log_b a}\right)$$

$$T(n) = n^d \sum_{i=0}^{\log_b n} \left(\frac{a}{b^d}\right)^i + O\left(n^{\log_b a}\right)$$

- $d > \log_b a$, $\frac{a}{b^d} < 1$, first term of geometric series dominates: $O\left(n^d\right)$
- $d < \log_b a$: leaf term dominates: $O\left(n^{\log_b a}\right)$
- $d = \log_b a$: $\frac{a}{b^d} = 1$, every term has the same value: $O\left(n^d \log n\right)$

MergeSort: What if instead of dividing the list in half, we divide it into k parts?

Apply It

Apply Master Theorem to Binary Sort

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

Next

Read chapter 0 and chapter 1

Next lecture: FFT and Arithmetic

Questions?

Slides by Aaron Hillegass