# ALGORITHMS PART II

Sid Chi-Kin Chau

[Lecture 6]

# Recap from Last Lecture

- Divide-and-conquer
  - Example: Merge Sort, Karatsuba Integer Multiplication

- How did we measure the speed of an algorithm?
  - Count the number of operations
  - How the number scales with respect to the input size
  - Time complexity of computation
  - Karatsuba multiplication scales as $n^{1.6}$

# Goals of This Lecture

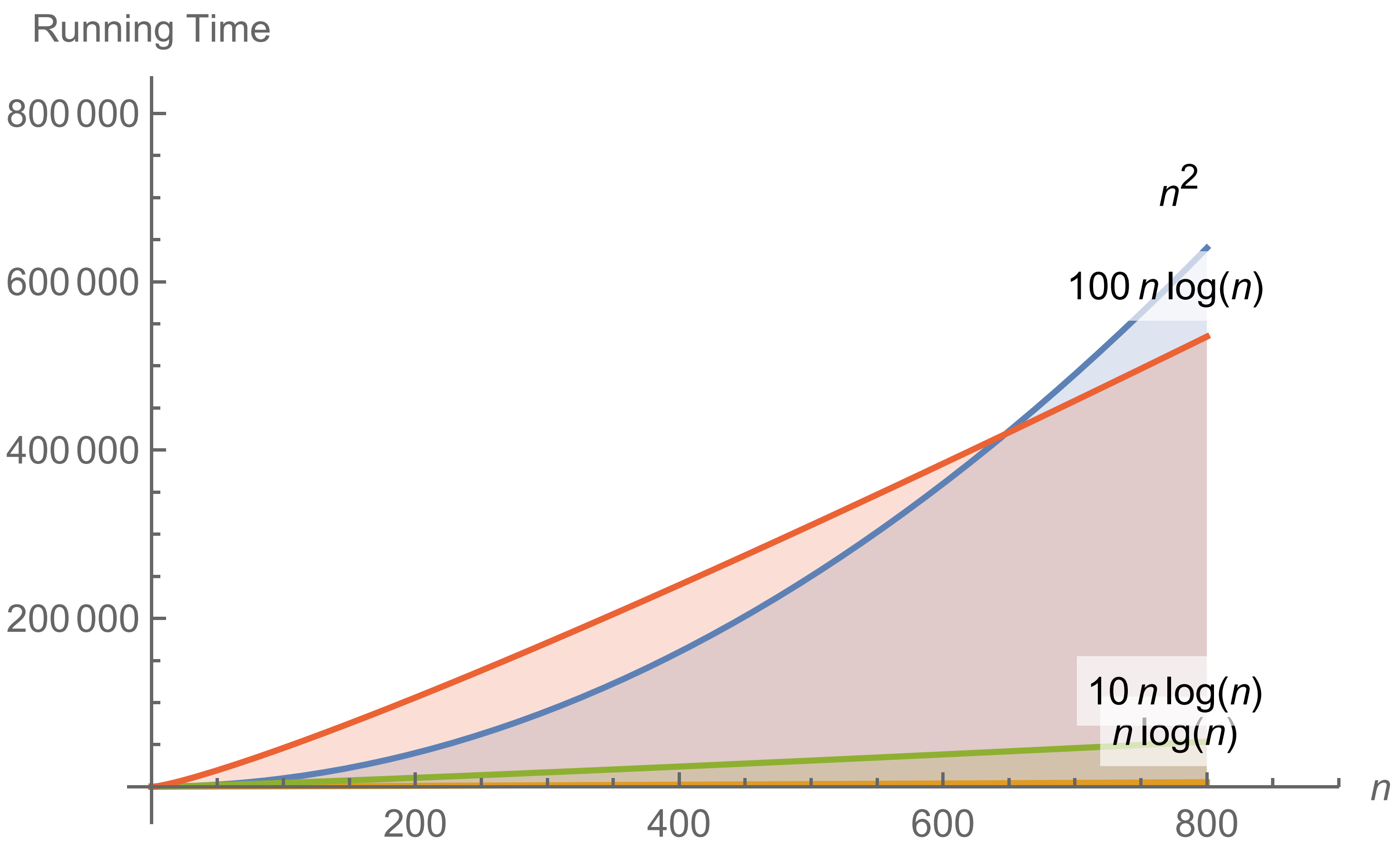
- Let us formally formulate the time complexity of computation
- How to formally measure the running time of an algorithm?
  - Big-O, Big-Ω, Big-Θ notations
- General approach of the running time with recursive algorithms
  - The Master theorem

# Big-O notation

- What do we mean when we measure runtime?
  - How long does it take to solve the problem, in seconds or minutes or hours?
- The exact runtime is heavily dependent on the programming language, system architecture, etc.
  - But the most factor is the input size (e.g. the number of bits that used in encoding input)
- We want a way to talk about the running time of an algorithm, with respect to the input size

# Main idea

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



# Asymptotic Analysis

- How does the running time scale as n gets large?
- 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)
  - $2^{100000000000000}$ n
  - is “better” than $n^2$ ?!?!

# Big-O

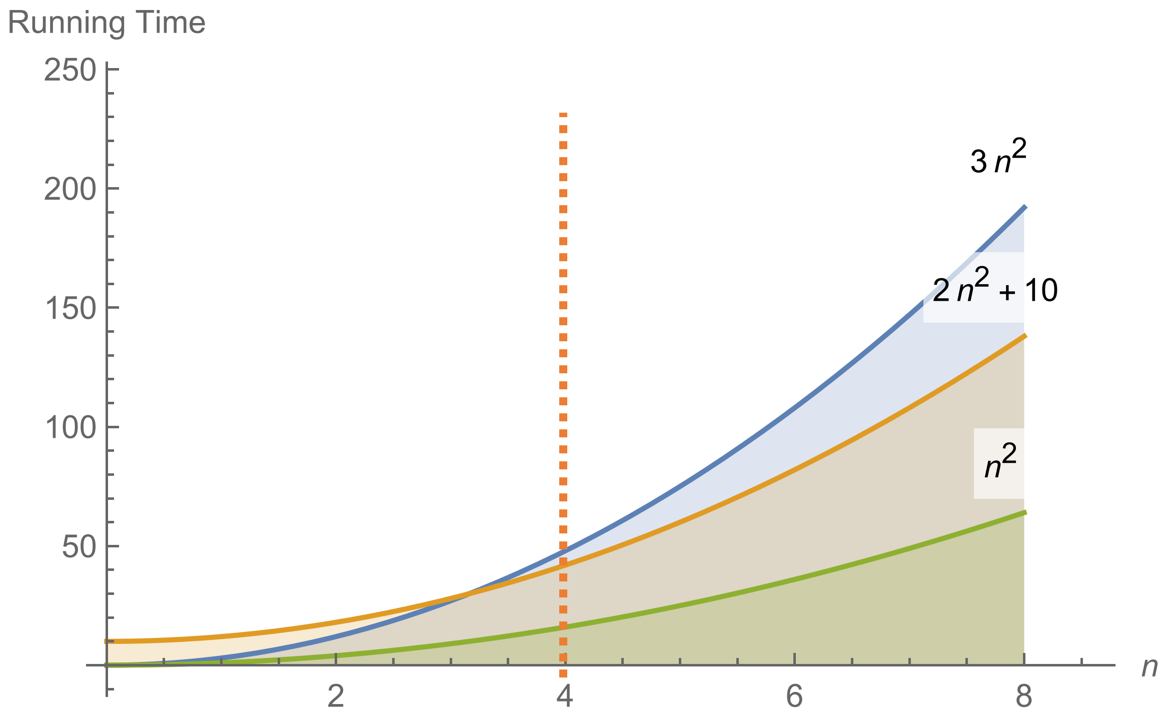
- Big-O means an **upper** bound
- Let $T(n)$, $g(n)$ be functions of positive integers
  - Think of $T(n)$ as the running time: positive and increasing in $n$
- We say that "$T(n) = O(g(n))$" if $g(n)$ grows at least as fast as $T(n)$, when $n$ gets large
- Formally,

$T(n) = O(g(n)) \iff$
There exist $c$, $n_0 > 0$, such that
$0 \leq T(n) \leq c \cdot g(n)$,
for all $n \geq n_0$

# Example

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

$T(n) = O(g(n)) \iff$
There exist $c$, $n_0 > 0$, such that
$0 \le T(n) \le c \cdot g(n)$,
for all $n \ge n_0$

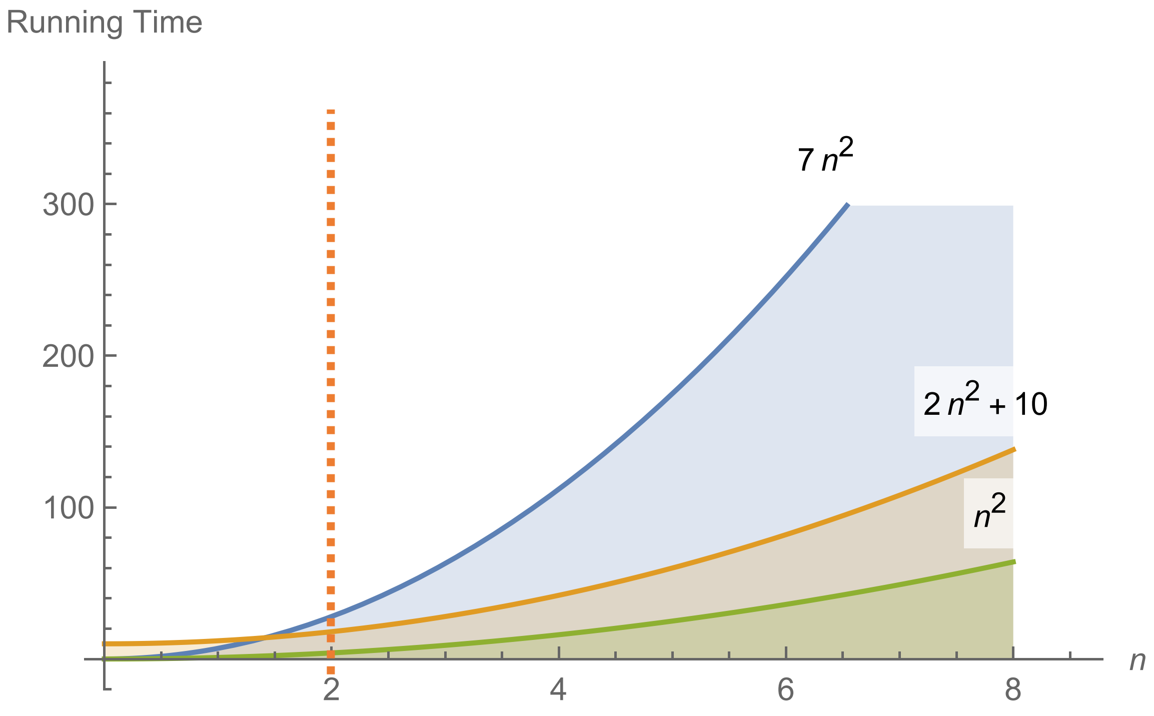



- Choose $c = 3$, $n_0 = 4$
- Then $0 \le 2n^2 + 10 \le 3n^2$, for all $n \ge 4$

# Example

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

$T(n) = O(g(n)) \iff$
There exist $c$, $n_0 > 0$, such that
$0 \leq T(n) \leq c \cdot g(n)$,
for all $n \geq n_0$



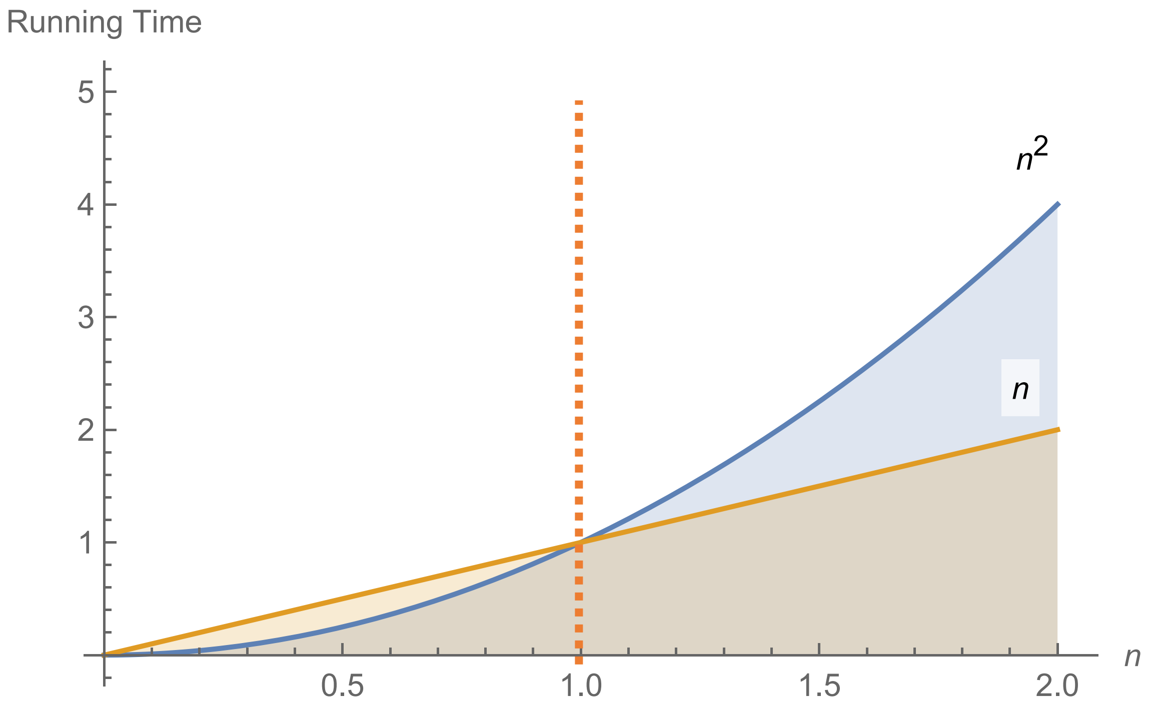

- Choose $c = 7$, $n_0 = 2$
- Then $0 \leq 2n^2 + 10 \leq 7n^2$, for all $n \geq 2$

# Example

- $n = O(n^2)$

$T(n) = O(g(n)) \iff$
There exist $c$, $n_0 > 0$, such that
$0 \leq T(n) \leq c \cdot g(n)$,
for all $n \geq n_0$

- Choose $c = 1$, $n_0 = 1$
- Then $0 \leq n \leq n^2$, for all $n \geq 1$



# Big-Ω

- Big-Ω means an **lower** bound
- Let T(*n*), g(*n*) be functions of positive integers
  - Think of T(*n*) as the running time: positive and increasing in *n*
- We say that "T(*n*) = Ω(g(*n*))" if g(*n*) grows at most as fast as T(*n*), when *n* gets large
- Formally,

$T(n) = \Omega(g(n)) \iff$
There exist $c$, $n_0 > 0$, such that
$c \cdot g(n) \leq T(n)$,
for all $n \geq n_0$

# Big-Θ

- Big-Θ means a **tight** bound
- Let T($n$), g($n$) be functions of positive integers
  - Think of T($n$) as the running time: positive and increasing in $n$
- We say that "T($n$) = Θ(g($n$))" if g($n$) grows tightly as fast as T($n$), when $n$ gets large
- Formally,

$T(n) = \Theta(g(n)) \iff$
There exist $c$, $c'$, $n_0 > 0$, such that
$c \cdot g(n) \leq T(n) \leq c' \cdot g(n)$,
for all $n \geq n_0$

# Example

- $2n^2 + 10 = O(n^2)$, $2n^2 + 10 = \Omega(n^2)$, $2n^2 + 10 = \Theta(n^2)$
- $2n + 10 = O(n^2)$, $2n + 10 \neq \Omega(n^2)$, $2n + 10 \neq \Theta(n^2)$
- $2n^2 + 10 \neq O(n)$, $2n^2 + 10 = \Omega(n)$, $2n^2 + 10 \neq \Theta(n)$
- Note that if $T(n) = O(g(n))$ and $T(n) = \Omega(g(n))$, then $T(n) = \Theta(g(n))$
  - Why?

# Example 1

- What is the running time T(*n*) of the following procedure?
- Assume `c()` requires constant running time

```
public void method(int n) {
   for (int i = 0; i < n; i++) {
      for (int j = 0; j < n; j++) {
          for (int k = 0; k < n; k++) {
              for (int l = 0; l < n; l++) {
                c();
              }
          }
      }
    }
```

- $T(n) = O(n^4)$

# Example 2

- What is the running time T($n$) of the following procedure?
- Assume `c()` requires constant running time

```
public void method(int n) {
        h=1;
        while (h <= n)
        {
                c();
                h = 2*h;
        }
}
```

- $h = 1, 2, 4, \ldots, 2^{\log(n)}$
- $T(n) = O(\log n)$

# Example 3

- What is the running time T(*n*) of the following procedure?
- Assume `c()` requires constant running time

```
public void method(int n) {
   for (int j = 0; j < n; j++) {
       for (int i = 0; i < j; i++) {
           c();
       }
   }
}
```

- Each inner for-loop (*i*) gets *j* times
- $T(n) = 1+2+\ldots+n = O(n^2)$

# Summing Up

- Big-O
  - $T(n) = O(g(n)) \Leftrightarrow$ there exist $c$, $n_0 > 0$, such that $0 \leq T(n) \leq c \cdot g(n)$, for all $n \geq n_0$
- Big-Ω
  - $T(n) = \Omega(g(n)) \Leftrightarrow$ there exist $c$, $n_0 > 0$, such that $c \cdot g(n) \leq T(n)$, for all $n \geq n_0$
- Big-Θ
  - $T(n) = \Theta(g(n)) \Leftrightarrow$ there exist $c$, $c'$, $n_0 > 0$, such that $c \cdot g(n) \leq T(n) \leq c' \cdot g(n)$, for all $n \geq n_0$

- But we should always be careful not to abuse it
  - $c = 2^{1000000}$
  - $n \geq n_0 = 2^{1000000}$

# Exercise

- Suppose the $n$ is the size of input. Which of the following algorithms are the fastest and slowest?

  A. Algorithm A with runtime modelled as $T(n) = 1.5n + n$
  B. Algorithm B with runtime modelled as $T(n) = 2n + 200000$
  C. Algorithm C with runtime modelled as $T(n) = n^{1.1}$

- Which one of the following is INCORRECT?

  A. $2n^2 + 10000^{10000}$ is in $O(2n^2)$
  B. $2n^2 + n + 100$ is in $O(n^3)$
  C. $0.1n^2$ is in $O(n \log(n))$
  D. $2n^2 + 10$ is in $O(n^2)$

# The Master Theorem

- Recursive integer multiplication
  - $T(n) = 4\,T(n/2) + O(n)$
  - $T(n) = O(n^2)$
- Karatsuba integer multiplication
  - $T(n) = 3\,T(n/2) + O(n)$
  - $T(n) = O(n^{\log(3)}) \approx O(n^{1.6})$
- What's the pattern?
  - A formula that solves recurrences when all of sub-problems are the same size
  - "Generalized" tree method

# The Master Theorem

- The **master theorem** applies to recurrence form:

$$T(n) = a \cdot T(n/b) + O(n^d),$$

  where $a \geq 1$, $b > 1$
  - $a$: 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
- Case 1: If $a = b^d$, then $T(n) = O(n^d \log(n))$
- Case 2: If $a < b^d$, then $T(n) = O(n^d)$
- Case 3: If $a > b^d$, then $T(n) = O(n^{\log_b(a)})$

# Example

- $T(n) = T(n/2) + O(1)$
  - $a = 1, b = 2, d = 0 \Rightarrow a = b^d$
  - Hence, $T(n) = O(\log(n))$
- $T(n) = T(n/2) + O(n)$
  - $a = 1, b = 2, d = 1 \Rightarrow a < b^d$
  - Hence, $T(n) = O(n)$
- $T(n) = 2 \cdot T(n/2) + O(1)$
  - $a = 2, b = 2, d = 0 \Rightarrow a > b^d$
  - Hence, $T(n) = O(n)$
- $T(n) = 2 \cdot T(n/2) + O(n)$
  - $a = 2, b = 2, d = 1 \Rightarrow a = b^d$
  - Hence, $T(n) = O(n \log(n))$

# Example

- $T(n) = 4 \cdot T(n/2) + O(1)$
  - $a = 4, b = 2, d = 0 \Rightarrow a > b^d$
  - Hence, $T(n) = O(n^2)$
- $T(n) = 4 \cdot T(n/2) + O(n)$
  - $a = 4, b = 2, d = 1 \Rightarrow a > b^d$
  - Hence, $T(n) = O(n^2)$
- $T(n) = 3 \cdot T(n/2) + O(1)$
  - $a = 3, b = 2, d = 0 \Rightarrow a > b^d$
  - Hence, $T(n) = O(n^{\log(3)}) \approx O(n^{1.6})$
- $T(n) = 3 \cdot T(n/2) + O(n)$
  - $a = 3, b = 2, d = 1 \Rightarrow a > b^d$
  - Hence, $T(n) = O(n^{\log(3)}) \approx O(n^{1.6})$
  - (Karatsuba Multiplication)

# Example 4

- What is the running time T($n$) of the following procedure?

```
public void method(int n) {
        c();
        if (n > 0) method(n-1);
}
```

- If $n > 0$,
  - $T(n) = T(n - 1) + 1$
- Else
  - $T(n) = 1$
- Hence, $T(n) = O(n)$

# Example 5

- What is the running time T($n$) of the following procedure?

```
public void method(int n) {
        c();
        if (n > 0) method(n/2);
}
```

- If $n > 0$,
  - $T(n) = T(n/2) + 1$
- Else
  - $T(n) = 1$
- Hence, $T(n) = O(\log(n))$

# Example 6

- What is the running time T($n$) of the following procedure?

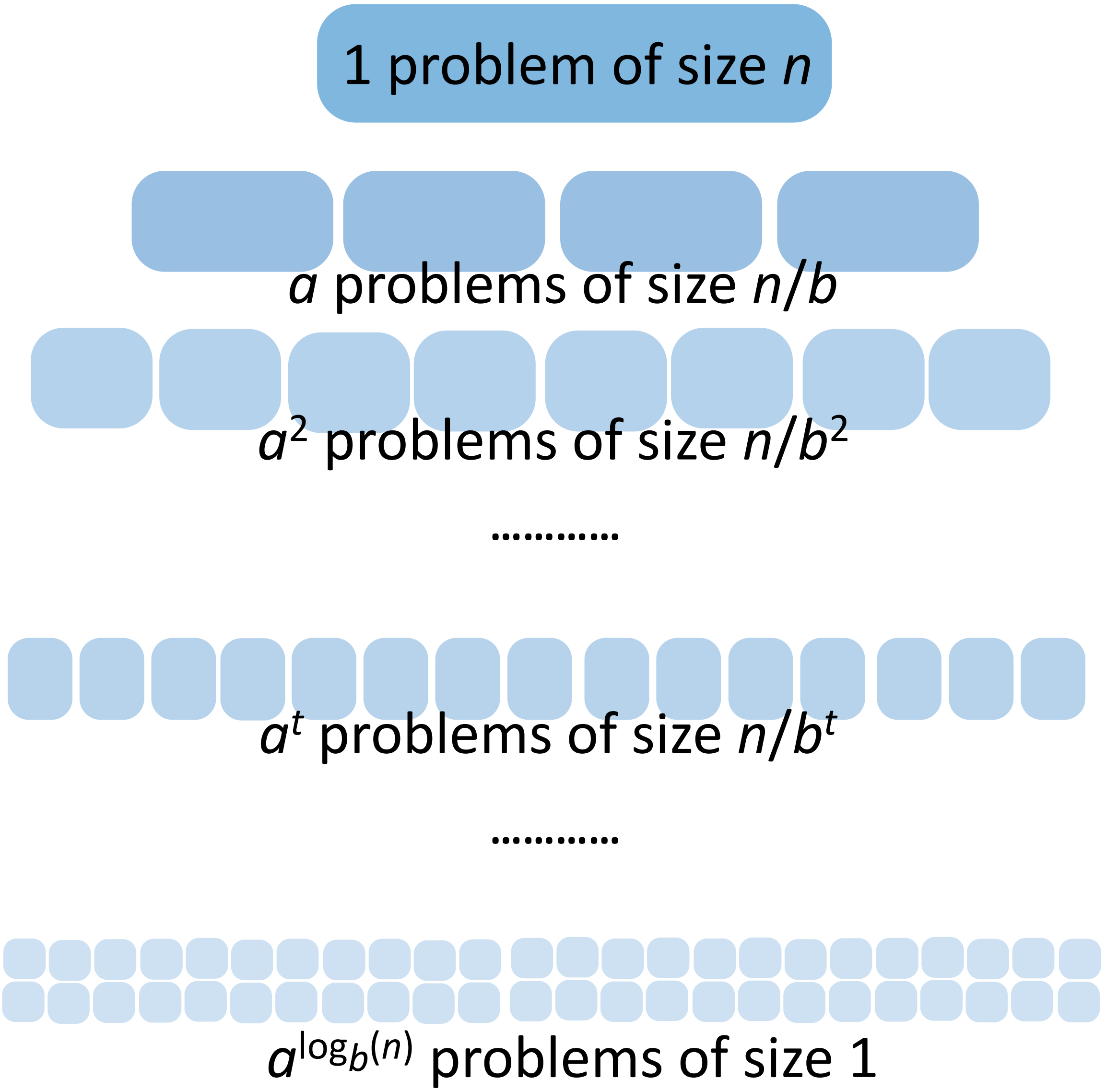
```
public void method(int n) {
        c();
        if (n > 0) { method(n/2); method(n/2);}
}
```

- If $n > 0$,
  - $T(n) = 2 \cdot T(n/2) + 1$
- Else
  - $T(n) = 1$
- Hence, $T(n) = O(n)$

# Proof of the Master Theorem

- We'll do the same recursion tree thing we did for multiplication, but be more careful.
- Suppose that $T(n) = a \cdot T(n/b) + c \cdot n^d$
- The hypothesis of the Master Theorem was the extra work at each level was $O(n^d)$
  - That's NOT the same as work $\leq c \cdot n^d$ for some constant $c$
- That's true … we'll actually prove a weaker statement that uses this hypothesis instead of the hypothesis that $T(n) = a \cdot T(n/b) + O(n^d)$

$$T(n) = a \cdot T(n/b) + c \cdot n^d$$




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

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

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

# The Master Theorem

- The **master theorem** applies to recurrence form:

$$T(n) = a \cdot T(n/b) + O(n^d),$$

  where $a \geq 1$, $b > 1$
  - $a$: 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 solution
- Case 1: If $a = b^d$, then $T(n) = O(n^d \log(n))$
- Case 2: If $a < b^d$, then $T(n) = O(n^d)$
- Case 3: If $a > b^d$, then $T(n) = O(n^{\log_b(a)})$

# Closer Look at All the Cases

- Total Work = $c \cdot n^d \cdot \sum_{t=0}^{\log_b(n)} \left(\frac{a}{b^d}\right)^t$,
  - where $\sum_{t=0}^{\log_b(n)} \left(\frac{a}{b^d}\right)^t$ is a sum of geometric sequence
- If $\frac{a}{b^d} = 1$, then T($n$) = $c \cdot n^d \cdot \sum_{t=0}^{\log_b(n)} \left(\frac{a}{b^d}\right)^t$ = O($n^d \log(n)$)
- If $\frac{a}{b^d} < 1$, then T($n$) = $c \cdot n^d \cdot \sum_{t=0}^{\log_b(n)} \left(\frac{a}{b^d}\right)^t$ = O($n^d$)
- If $\frac{a}{b^d} > 1$, then T($n$) = $c \cdot n^d \cdot \sum_{t=0}^{\log_b(n)} \left(\frac{a}{b^d}\right)^t$ = O($n^{\log_b(a)}$)

# The Eternal Struggle

- Branching causes the number of problems to explode!
  - The most work is at the bottom of the tree!
- The problems lower in the tree are smaller!
  - The most work is at the top of the tree!

# Consider Three Different Cases

**(Case 1)** $T(n) = 2 \cdot T(n/2) + O(n)$

- $a = 2,\ b = 2,\ d = 1 \Rightarrow a = b^d$

**(Case 2)** $T(n) = T(n/2) + O(n)$

- $a = 1,\ b = 2,\ d = 1 \Rightarrow a < b^d$

**(Case 3)** $T(n) = 4 \cdot T(n/2) + O(n)$

- $a = 4,\ b = 2,\ d = 1 \Rightarrow a > b^d$

# Consider Three Different Cases

**(Case 1)** $T(n) = 2 \cdot T(n/2) + O(n)$

- $a = 2,\ b = 2,\ d = 1 \Rightarrow a = b^d$

- The branching just balances out the amount of work
- The same amount of work is done at every level
- $T(n) = (\text{number of levels}) \cdot (\text{work per level})$

  $= \log(n) \cdot O(n) = O(n \log(n))$

1 problem of size $n$

2 problems of size $n/2$

4 problems of size $n/4$

…………

$a^t$ problems of size $n/a^t$

…………

$n$ problems of size 1

# Consider Three Different Cases

**(Case 2)** T($n$) = T($n$/2) + O($n$)

- $a = 1$, $b = 2$, $d = 1 \Rightarrow a < b^d$

- The amount of work done at the top (the biggest problem) swamps the amount of work done anywhere else
- T($n$) = T(work at top) = O($n$)

1 problem of size $n$

1 problems of size $n/2$

1 problems of size $n/4$

…………

1 problems of size $n/2^t$

…………

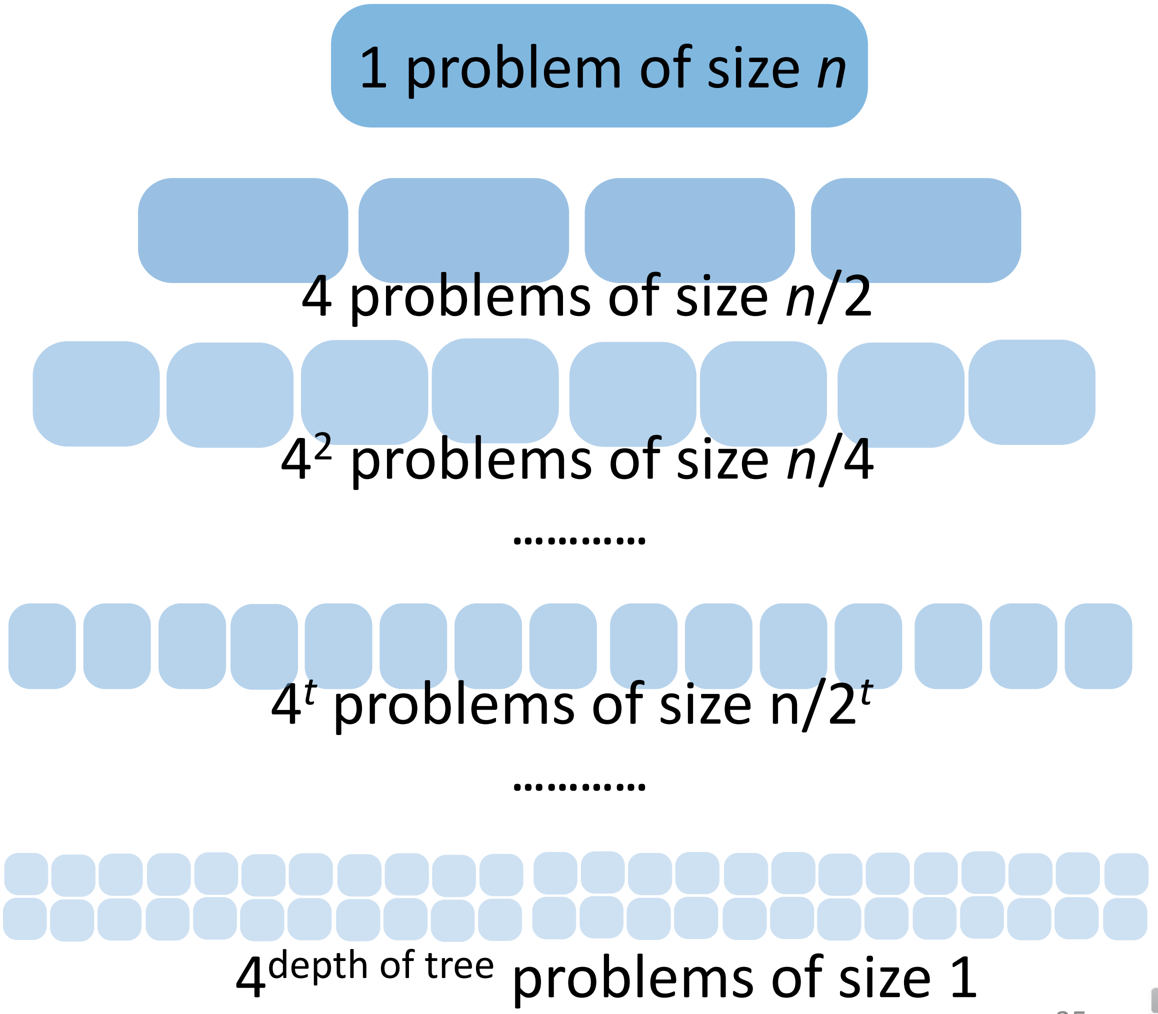
1 problems of size 1

# Consider Three Different Cases

**(Case 3)** T($n$) = 4·T($n$/2) + O($n$)

- $a = 4$, $b = 2$, $d = 1 \Rightarrow a > b^d$

- There are a HUGE number of leaves, and the total work is dominated by the time to do work at these leaves
- T($n$) = O(work at bottom) = O( $4^{\text{depth of tree}}$ ) = O($n^2$)



# Exercise

- Which of the following are the fastest and the slowest?
  A. $T(n) = T(n/2) + O(1)$
  B. $T(n) = 5\ T(n/6) + O(n)$
  C. $T(n) = 2\ T(n/3) + O(1)$
  D. $T(n) = 10\ T(n/30) + O(n^{1.1})$
  E. $T(n) = T(0.5n) + O(n^2)$
  F. $T(n) = T(n\text{-}1) + T(n\text{-}2) + O(1)$

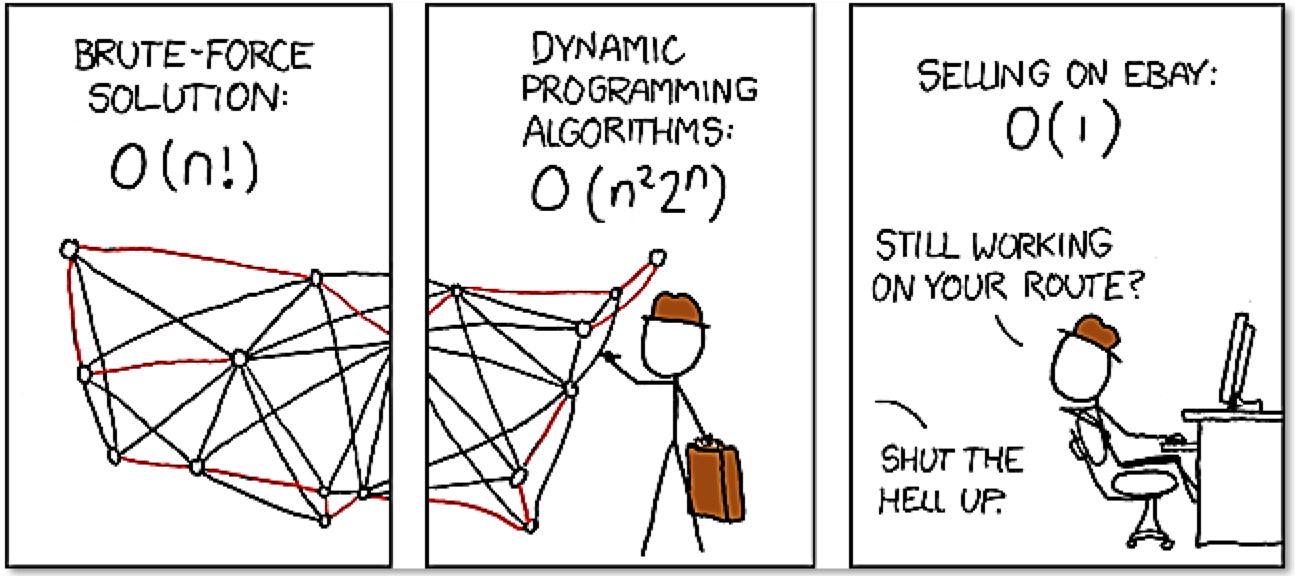
# Summary

- Asymptotic Analysis: Big-O, Big-Ω, Big-Θ
- The "Master Theorem" is a powerful tool
  - It is a systematic approach to calculate general recurrence relations from scratch

- The **master theorem** applies to recurrence form:

$$T(n) = a \cdot T(n/b) + O(n^d),$$

  where $a \geq 1$, $b > 1$
  - $a$: 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 solution
- Case 1: If $a = b^d$, then $T(n) = O(n^d \log(n))$
- Case 2: If $a < b^d$, then $T(n) = O(n^d)$
- Case 3: If $a > b^d$, then $T(n) = O(n^{\log_b(a)})$


BRUTE-FORCE SOLUTION:
O(n!)
DYNAMIC PROGRAMMING ALGORITHMS:
O(n²2ⁿ)
SELLING ON EBAY:
O(1)
STILL WORKING ON YOUR ROUTE?
SHUT THE HELL UP.