Algorithms

http://algs4.cs.princeton.edu

1.4 Analysis of Algorithms

- *Asymptotic analysis*
- *Analyzing recursive algorithms*
- *The Master Theorem*

Algorithms

http://algs4.cs.princeton.edu

Slides adapted from
Robert Sedgewick | Kevin Wayne

http://algs4.cs.princeton.edu

1.4 Analysis of Algorithms

- *Asymptotic analysis*
- *Analyzing recursive algorithms*
- *The Master Theorem*

Big O Notation – Upper Bounds

Let $f(n)$ and $g(n)$ be running time functions. ($\geq 0\ \forall n \geq 0$)

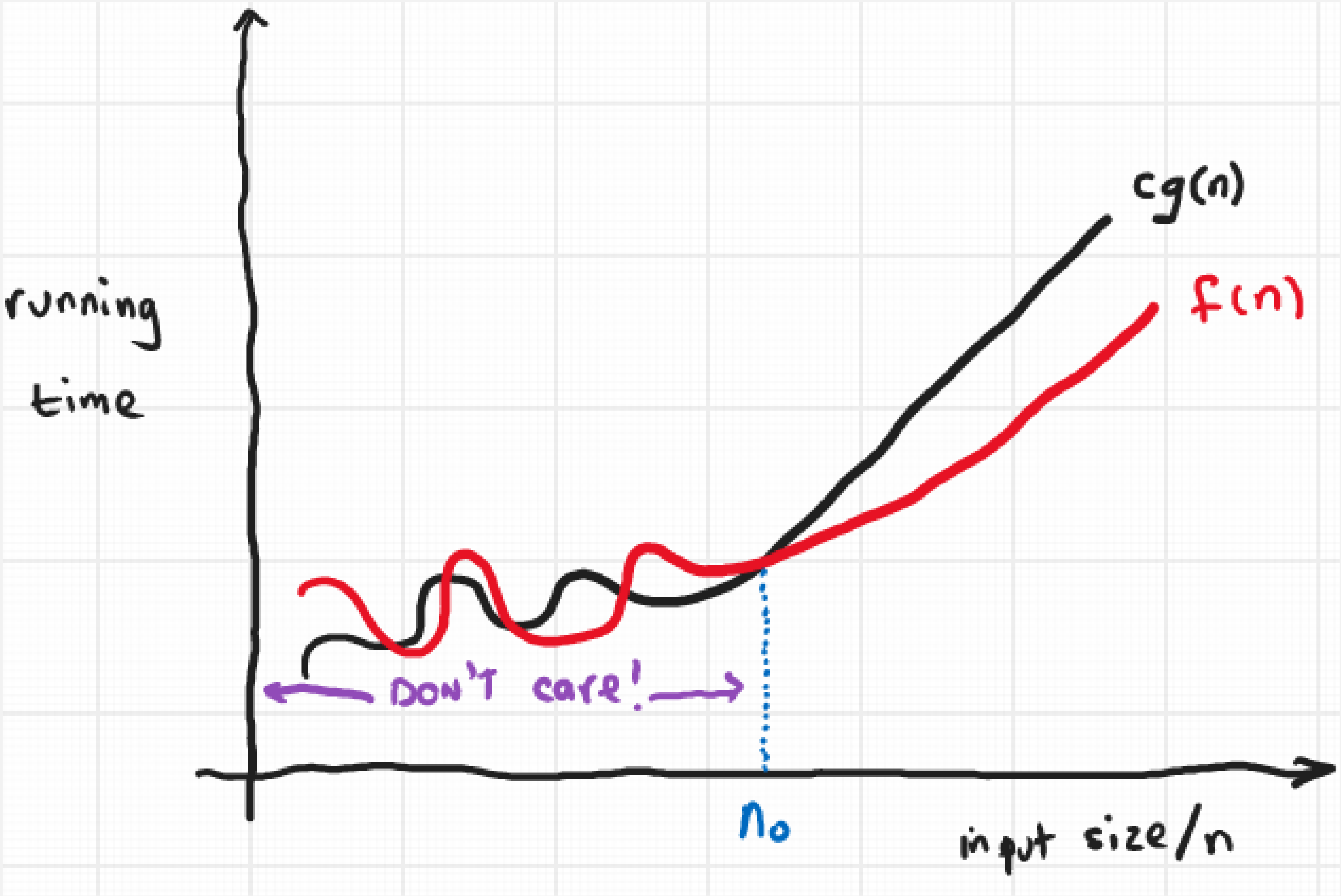
Definition.

$f(n)$ *is* $O\big(g(n)\big)$ if there exists a constant $c > 0$ and $n_0 \geq 0$ such that $f(n) \leq cg(n), \forall n \geq n_0$

Ex. $f(n) = 3n^3 + 2n^2 - n$

- $f(n)$ *is* $O(n^3)$

choose $c = 6, n_0 = 0$



We use it to provide an upper bound on the time Complexity of an algorithm.

We say, algorithm A takes $O(n^3)$ time to solve problem P.

POP Quiz

Q. If $f(n) = 32n^2 + 17n + 1$, which of the following is true?

A. $f(n)$ *is* $O(n)$

B. $f(n)$ *is* $O(n^2)$

C. $f(n)$ *is* $O(n^3)$

D. Both B and C

E. Both A and B

F. Both A and C

G. All of A, B, and C

Big O Properties

Let $f(n)$ and $g(n)$ be running time functions. ($\geq 0\ \forall n \geq 0$)

Reflexivity. $f(n)$ *is* $O(f(n))$

Constants. If $f(n)$ *is* $O(g(n))$ and $c > 0$, then $cf(n)$ *is* $O(g(n))$.

Products. If $f(n)$ *is* $O(g_1(n))$ and $h(n)$ *is* $O(g_2(n))$, then $f(n)h(n)$ *is* $O(g_1(n)g_2(n))$.

Sums. If $f(n)$ *is* $O(g_1(n))$ and $h(n)$ *is* $O(g_2(n))$, then $f(n) + h(n)$ *is* $O(\max\{g_1(n), g_2(n)\})$.

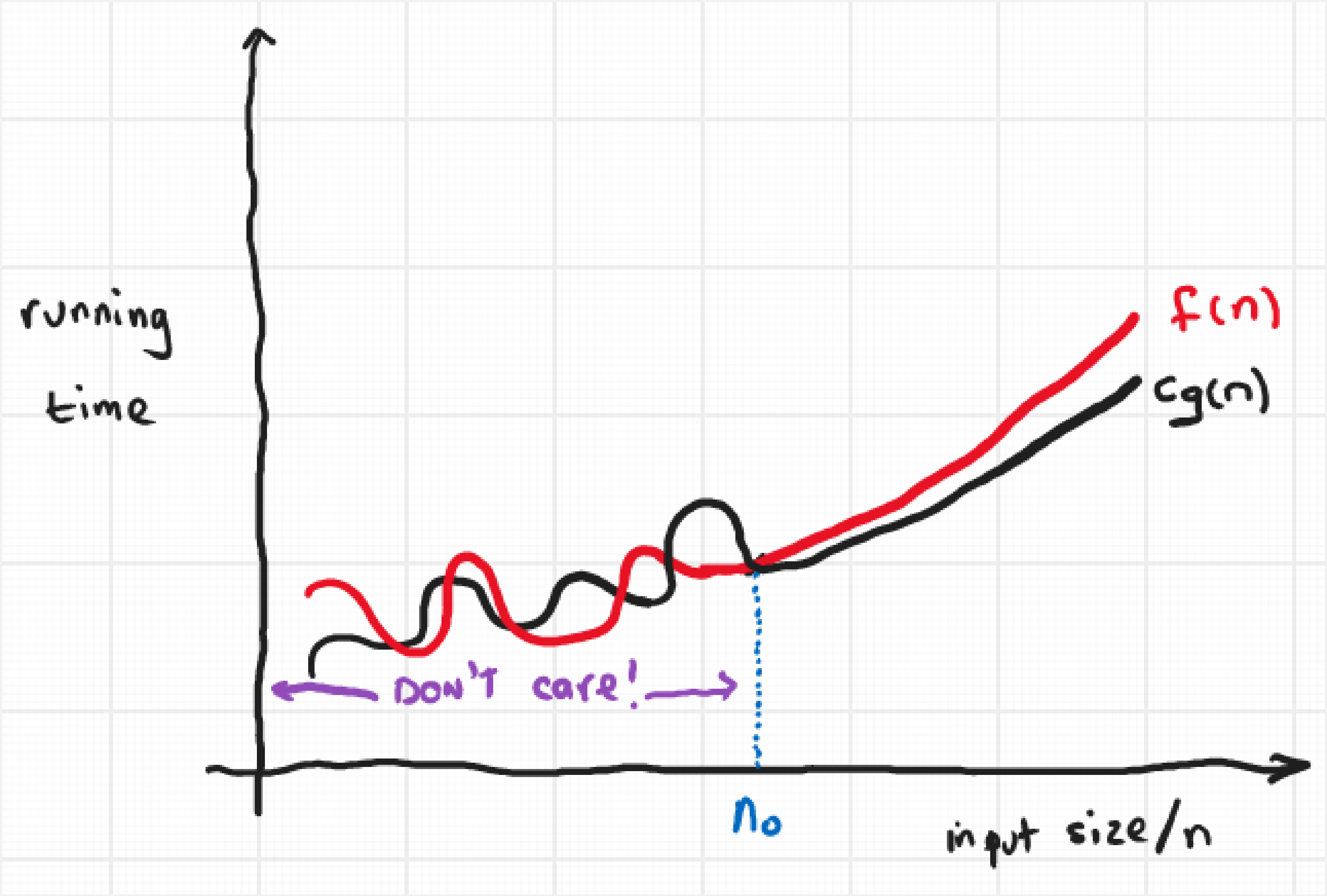
Transitivity. If $f(n)$ *is* $O(g_1(n))$ and $g_1(n)$ *is* $O(g_2(n))$, then $f(n)$ *is* $O(g_2(n))$.

Big Omega Notation – Lower Bounds

Let $f(n)$ and $g(n)$ be running time functions. ($\geq 0\ \forall n \geq 0$)

Definition.

$f(n)$ *is* $\Omega(g(n))$ if there exists a constant $c > 0$ and $n_0 \geq 0$ such that $f(n) \geq cg(n), \forall n \geq n_0$



Ex. $f(n) = 3n^3 + 2n^2 - n$

- $f(n)$ *is* $\Omega(n^3)$

choose $c = 3, n_0 = 0$

We use it to provide a lower bound on the time Complexity for solving a problem

We say, any algorithm requires $\Omega(n^3)$ operations to solve problem P in the worst case. (on some computational model)

Big Theta Notation – Tight bounds

Let $f(n)$ and $g(n)$ be running time functions. ($\geq 0\ \forall n \geq 0$)

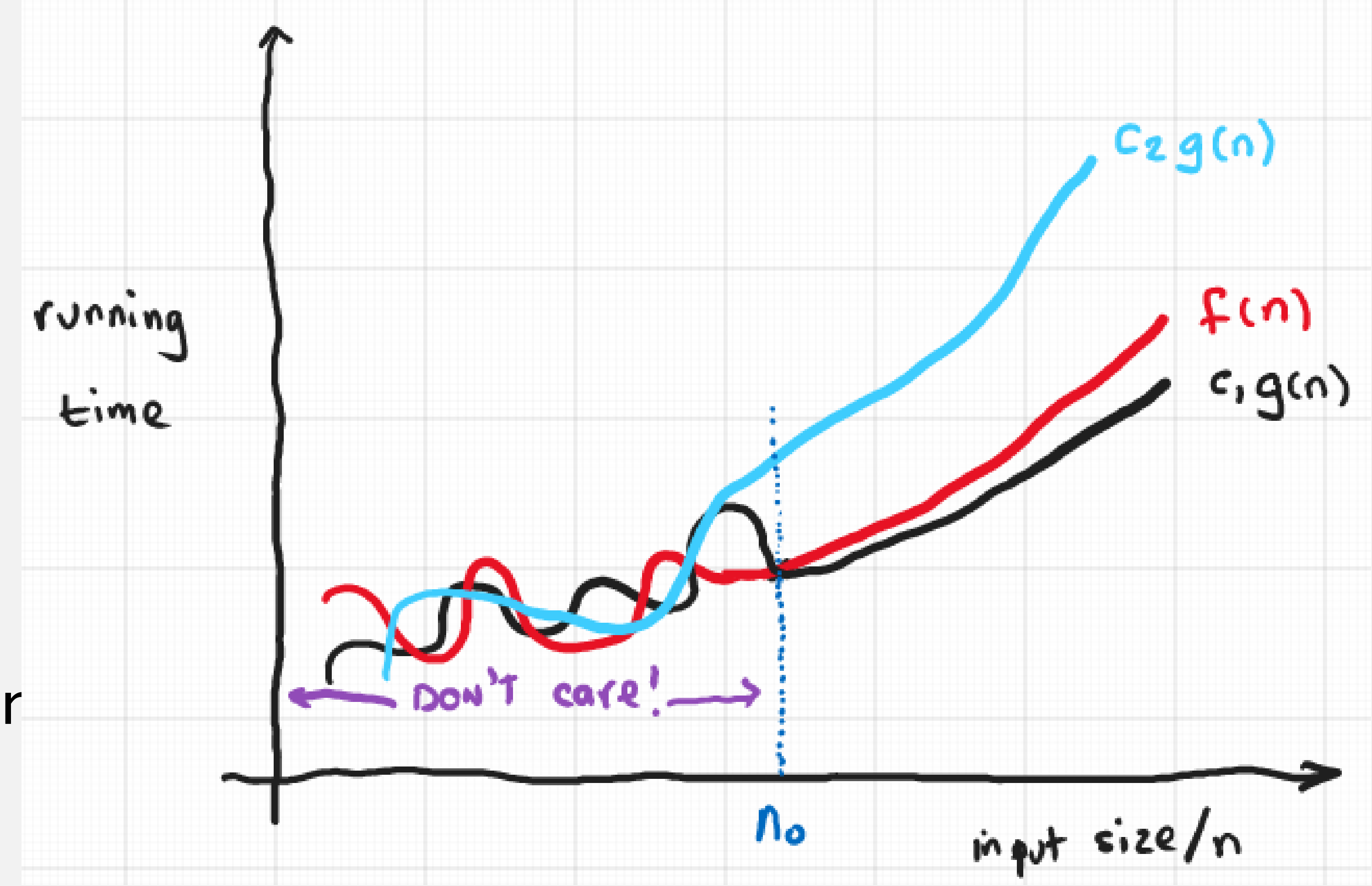
Definition.

$f(n)\ is\ \Theta(g(n))$ if there exists constants $c_1 > 0, c_2 > 0$ and $n_0 \geq 0$ such that $c_1 g(n) \leq f(n) \leq c_2 g(n), \forall n \geq n_0$

Ex. $f(n) = 3n^3 + 2n^2 - n$

- $f(n)\ is\ \Theta(n^3)$

choose $c_1 = 3, c_2 = 6, n_0 = 0$



We use it to provide tight bounds on Complexity of an algorithm solving problem P.

We say, merge-sort makes $\Theta(n \lg n)$ comparisons to sort n elements.

Big O Properties

Let $f(n)$ and $g(n)$ be running time functions. ($\geq 0\ \forall n \geq 0$)

Proposition. $f(n)\ is\ \Theta(g(n))$ if and only if $f(n)\ is\ \Omega\big(g(n)\big)$ and $f(n)\ is\ O(g(n))$.

Proposition. If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$ for some constant $c > 0$, then $f(n)\ is\ \Theta(g(n))$.

Proposition. If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$ then $f(n)\ is\ \mathrm{O}(g(n))$.

Proposition. If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$ then $f(n)\ is\ \Omega(g(n))$.

Proposition. If $f(n) = a_d n^d + a_{d-1} n^{d-1} + \cdots + a_1 n + a_o$, with $a_d > 0$, then $f(n)\ is\ \Theta(n^d)$.

Proposition. $\log_a n\ is\ \mathrm{O}(n^d)$ for every $a > 1$ and every $d > 0$.

Proposition. $n^d\ is\ \mathrm{O}(r^n)$ for every $d > 0$ and every r > 1.

POP Quiz

Q. In each of the following, indicate whether $f(n) is\ O(g(n))$, or $f(n) is\ \Omega(g(n))$, or both.

A. $f(n) = n - 100$, $g(n) = n - 200$

B. $f(n) = 2^n$, $g(n) = 2^{n+1}$

C. $f(n) = 10 \log n$, $g(n) = \log n^2$

POP Quiz

Q. For each of the following algorithms, give an analysis of their running time. Big O will do.

```
QUAD(a, b, c)
d = a^2 − 4bc
if d ≥ 0
    return true
return false
```

```
SUM1(n)
x = 0
for i = 1 to n
    x = x + 1
return x
```

POP Quiz

Q. For each of the following algorithms, give an analysis of their running time. Big O will do.

```
SUM2(n)
x = 0
for i = 1 to n
  for j = 1 to i
    x = x + 1
return x
```

```
SUM3(n)
x = 0
for i = 1 to n
   x = x + SUM1(n)
return x
```

POP Quiz

Q. How many times, asymptotically (Big O), is the statement $x = x + 1$ executed?

```
SUM4(n);
j = n
x = 0
while j ≥ 1
    x = x + 1
    j = j/2
return x
```

```
SUM5(n);
j = n
x = 0
while j ≥ 1
    for i = 1 to j
        x = x + 1
    j = j/2
return x
```

http://algs4.cs.princeton.edu

1.4 Analysis of Algorithms

- *Asymptotic analysis*
- *Analyzing recursive algorithms*
- *The Master Theorem*

POP Quiz

Q. What is the running time of the following algorithms?

ALGO1(A[1:n], n) 1. if n == 1 2. return 3. for i = 1 to n 4. A[i] = n 5. ALGO1(A, n/2)	ALGO2(A[0:n−1], n) 1. if n == 0 2. return A[0] 3. return A[n] + ALGO2(A, n−1)

Recurrences

A **recurrence** is a recursive definition of a function. It has a **base case(s)** and one or more **recursive cases**.

Example.

$$f(n) = \begin{cases} 0, & n = 0 \\ f(n-1) + 1, & n > 0 \end{cases}$$

$$f(n) = \begin{cases} 1, & n = 1 \\ f\left(\frac{n}{3}\right) + f\left(\frac{2n}{3}\right) + n, & n > 1 \end{cases}$$

These recurrences often arise when analyzing recursive algorithms. We seek closed form solutions to these kinds of relations.

General Recurrence Formula

$$T(n) = \begin{cases} c, if\ n = n_0 \\ aT\left(\frac{n}{b}\right) + D(n) + C(n), otherwise \end{cases}$$

- $D(n)$ is the time for dividing the input
- $C(n)$ is the time to combine solutions
- a is the number of subproblems, each of size $\frac{1}{b}$
- n_0 is the base case(s) and c is a constant

Example #1

```
ALGO2(A[0:n−1], n)
1. if n == 0
2.   return A[0]
3. return A[n] + ALGO2(A, n−1)
```

- $D(n) = 1$ is the time for dividing the input --- at line 3, subtract 1 from n
- $C(n) = 1$ is the time to combine solutions --- at line 3, addition
- $n_0 = 0$ is the base case and $c = 3$
- General formula is $T(n) = T(n - 1) + 2, T(0) = 3$

Solving recurrences by iteration/subtitution

$$T(0) = 3$$

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

Solution by <u>iteration</u>

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

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

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

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

$$= T(n-3) + 2 + 2 + 2$$

$T(n-3) = T(n-3-1) + 2$

$$= T(n-4) + 2 + 2 + 2 + 2$$

$$\vdots$$

$$= T(0) + \overbrace{2 + 2 + 2 + \cdots + 2}^{n \text{ twos! why?}}$$

counted from $n-1$ to 1 + the two already there!

$$= 3 + 2n$$

$\therefore T(n)$ is $O(n)$.

Pause…

Solve the recurrences.

1. $T(n) = T(n-1) + n, T(0) = 0$
2. $T(n) = 2T(n-1) + 1, T(1) = 1$

Example #2

```
ALGO1(A[1:n], n)
1. if n == 1
2.     return
3. for i = 1 to n
4.   A[i] = n
5. ALGO1(A, n/2)
```

- $D(n) = 1$ is the time for dividing the input --- at line 3, division by 2
- $C(n) = n$ we don't combine solutions, however, when $n > 1$, we execute lines 3 and 4 which takes $O(n)$ time.
- $n_0 = 1$ is the base case and $c = 2$
- General formula is $T(n) = T(n/2) + n, T(1) = 2$

The Master Theorem

Let $T(n) = aT\left(\frac{n}{b}\right) + O(n^d)$ for some constants $a > 0, b > 1,$ and $d \geq 0,$ then

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

Solving recurrences by iteration/subtitution

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

$$T(1) = 2$$

Note that we could have used iteration → then have to deal with summations!!

By Master Theorem.

$a = 1$, $b = 2$, $d = 1$

$$d \boxed{>} \log_b a$$

$$1 > \log_2 1 = 0$$

$$\therefore T(n) = O(n)$$

Pause…

Solve the recurrences.

1. $T(n) = 2T(n/2) + n$
2. $T(n) = 4T(n/2) + n^2$
3. $T(n) = 4T\left(\frac{n}{3}\right) + n$

http://algs4.cs.princeton.edu

1.4 Analysis of Algorithms

- *Asymptotic analysis*
- *Analyzing recursive algorithms*
- *The Master Theorem*