Insertion Sort, Proving Correctness, Run time analysis, Merge Sort, Quick Sort

1 Introduction to Sorting - Insertion Sort (CLRS §2.1, 2.2)

Input: array $A[1 \dots n]$ of n number

Goal: Sort in non decreasing order.

in case of repeated elements (doesn't strictly increase)

How do we write the algorithm?

1.1 The algorithm in words

Scan the array from left to right. For each position j, insert $A[j]$ into its correct position in $A[1 \dots j]$ by moving all elements larger than $A[j]$ one position to the right

1.2 Pseudocode

```
for j = 2 to n:
    Key ← A[j]
    i ← j-1
    while i>0 and A[i]>Key:
        A[i+1] ← A[i]     ← move element one position to the right
        i ← i-1
    A[i+1] ← Key
```

2 Proof by Induction, Loop Invariants, and Proving Correctness

- **Proof by induction**: We want to prove some statement P(n) for integers n ≥ 0. (or $n \in \mathbb{N}$ i.e, $n \geq 1$)
  1. ***Base case***: Prove P(0) or P(1)
  2. ***Inductive step***:

i.e. show P(K) ⇒ P(K+1)

- Inductive hypothesis: Assume P(n) holds for n = K   (n = K−1)   [n = 0, 1, .., K] ← strong induction
- Goal: Show P(K+1) holds   (P(K))

WARNING! DO NOT assume P(K+1) and "work backwards" to a true statement for the formal proof.

  3. ***Conclusion***: P(n) holds for all integers n ≥ 0. by principle of mathematical induction!

- **Loop invariant**: A property that holds throughout the execution of the algorithm
  1. ***Initialization***: loop invariant holds prior to the first iteration of the loop
  2. ***Maintenance***:

     If the loop invariant is true before an iteration of the loop, then it remains true before the next iteration.
  3. ***Termination***: When the loop terminates, the invariant gives useful property that helps show that the algorithm is correct.

Our loop invariant for insertion sort:

At the beginning of the jth iteration (or equivalently, the end of j-1st iteration) A[1 ... j-1] contains the elements that were originally there, but in sorted order.

At the end, j = n+1

"Termination" if loop invariant holds at the end of the algorithm, it tells us the output is correct.

```
for j=2 to n:
    Key ← A[j]
    i ← j-1
    while i>0 and A[i]>Key:
        A[i+1] ← A[i]
        i ← i-1
    A[i+1] ← Key
```

Claim: Loop invariant holds for j = 2, 3, 4, ..., n+1.

2.1 Proof of correctness of Insertion Sort

By induction.

First base case of induction ("initialization")
when $j=2$, $A[1]$ is just one number and is therefore sorted.

Induction Step ("maintenance") Assume the loop invariant holds for j. We will prove it holds for $j+1$.

inductive hypothesis.

By assumption $A[1 \ldots j-1]$ is sorted.

During the jth iteration, we insert $A[j]$ into its correct position. Therefore, after the jth iteration, $A[1 \ldots j]$ are sorted, so the loop invariant holds at the end of the jth iteration. ∎

$$S = 1 + 2 + 3 + \cdots + \overset{n-2}{98} + \overset{n-1}{99} + \overset{n}{100}$$

$$S = \underset{n}{100} + \underset{n-1}{99} + \underset{n-2}{98} + \cdots + 3 + 2 + 1$$

$$2S = 101 + 101 + 101 + \cdots 101 + 101 + 101$$

$$= 100 \times 101$$

$$S = \frac{100 \times 101}{2}$$

3 Introduction to analyzing running times

Example

for $i = 1$ to n
    $A[i] \leftarrow 5$

n steps
2n steps?
3n steps?

$\Theta(n)$

3.1 Running time of insertion sort

1. for $j = 2$ to n:
2. Key $\leftarrow A[j]$
3. $i \leftarrow j-1$
4. while $i>0$ and $A[i]>$Key:
5. $A[i+1] \leftarrow A[i]$
6. $i \leftarrow i-1$
7. $A[i+1] \leftarrow$ Key

(lines 2–3: c; lines 5–6: c')

Actually mean "worst case"

$$\sum_{j=2}^{n} (c + c' \underbrace{t_j}_{\text{number of iterations of the while loop}})$$

some constants (maybe $c \approx 3$, $c' \approx 2$)

worst case: $t_j \leq j$

$$\leq \sum_{j=2}^{n} (c + c' j) = c \overbrace{(n-1)}^{j=2,\dots,n \to n-1 \text{ iterations}} + c' \sum_{j=2}^{n} j$$

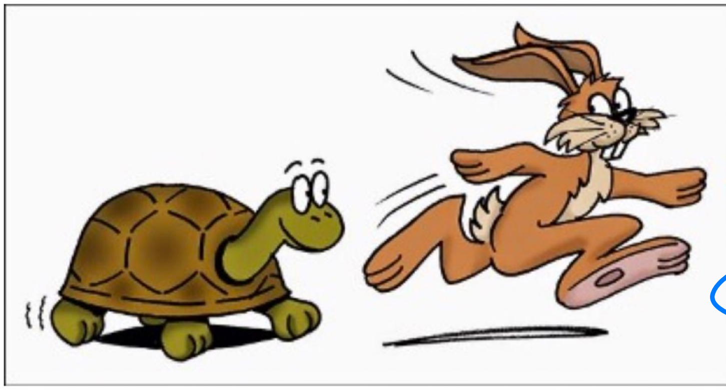
$$= c(n-1) + c'\left(\frac{n(n+1)}{2} - 1\right)$$

$$= \frac{c'}{2} n^2 + \left(c + \frac{c'}{2}\right)n - (c+c')$$

$$= \Theta(n^2) \leftarrow \text{running time of insertion sort.}$$

Motivation $5n$ "=" $7n$ $\lim_{n\to\infty} \frac{7n}{5n} = \frac{7}{5}$

Big-O (Asymptotic) Notation: Motivation



Motivation Why do we use it?

Which one below is easier to read?

- ◯ Algorithm A sorts a list containing n items in time at most $2.95n^2 + 10n + 5$
- ◯ Algorithm A sorts a list containing n items in time at most Cn^2 for some constant $C > 0$
- ◯ Algorithm A sorts a list containing n items in time $O(n^2)$

Prove that $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$ $n \in \mathbb{N}$

let $k \in \mathbb{N}$

inductive step

Inductive hypothesis: Assume $\sum_{i=1}^{k} i = \frac{k(k+1)}{2}$

want to show $\sum_{i=1}^{k+1} i = \frac{(k+1)(k+2)}{2}$

DON'T START with

$$\sum_{i=1}^{k+1} i = \frac{(k+1)(k+2)}{2}$$

Assumes what we are trying to show.

$$\sum_{i=1}^{k+1} i = \sum_{i=1}^{k} i + (k+1)$$

$$= \frac{k(k+1)}{2} + (k+1)$$

by the inductive hypothesis

$$= \ldots$$

$$= \frac{(k+1)(k+2)}{2}$$

large inputsizes

4 Growth of Functions and Asymptotic Behavior (CLRS §3.1)

Goal: Establish notation that enables us to compare the relative performance of different algorithms.

a "nice" function (one term, no constant factors)

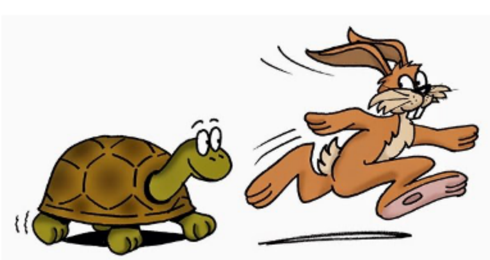
$\leq$ **Definition** • $\boxed{T(n) = O(g(n))}$ means there exists $c > 0$ such that $T(n) \leq cg(n)$ for sufficiently large n.

(Big-oh)

T is bounded above by g asymptotically

$\geq$ • $\boxed{T(n) = \Omega(g(n))}$ means there exists $c > 0$ such that $T(n) \geq cg(n)$ for sufficiently large n.

"=" • $\boxed{T(n) = \Theta(g(n))}$ means there exists c_1, c_2 such that $c_1 g(n) \leq T(n) \leq c_2 g(n)$ for sufficiently large n.



O( ) notation exercises

$O(\cdot)$-Notation

- **Asymptotic notation:**
  - Functions $f(n)$ and $g(n)$ represent running times.
  - $f(n) = O(g(n))$ means that exist $c, n_0 > 0$ such that $f(n) \le c \cdot g(n)$ for every $n \ge n_0$.
  - **Intuitively.** $f(n)$ does not grow faster than $g(n)$ when n is large.

negation:
For all $c > 0$, $n_0 > 0$
there exists some $n \geq n_0$
such that $100n^2 > cn$

$100n^2 \neq O(n)$

Take $n \geq \max\{c, n_0\}$
$n \geq c$
$n \cdot n \geq cn$
$100n^2 \geq 100cn$
$> cn$

Which of the following are true and why? Discuss with your teammates!

For true statements, find valid values of c and n_0

(Select all that apply.)

- [ ] $1000 = O(n)$ True $\lim_{n\to\infty} \frac{1000}{n} = 0$ OR take $c=1$, $n_0 = 1000$
- [ ] $100n^2 = O(n)$ false
- [ ] $n + n^2 + n^3 + n^4 = O(n^4)$ True
- [ ] $n^2 \cdot n^3 = O(n^4)$ False
- [ ] $n^5 - 2n^4 = O(n^4)$ False
- [ ] $n^3 + n^4 = O(n^4)$ True
- [ ] $4^n = O(2^n)$ false

$\lim_{n\to\infty} \frac{2^{2n}}{2^n} = \lim_{n\to\infty} 2^n = \infty$

2^{n+1}

$4^n \overset{?}{\cancel{\le}} 2^1 \cdot 2^n$

$4^n = (2^2)^n = 2^{2n}$

$\lim_{n\to\infty} \frac{100n^2}{n} = \lim_{n\to\infty} 100n = \infty$

$4^n \neq O(2^n)$

$\lim_{n\to\infty} \frac{4^n}{2^n} = \lim_{n\to\infty} \left(\frac{4}{2}\right)^n$
$= \lim_{n\to\infty} 2^n = \infty$

$n^3 + n^4 = O(n^4)$
$n^3 + n^4 \le cn^4$ for $n \geq n_0$

$c = 2$

$n^3 + n^4 \le 2n^4$ $n_0 = 1$

$n^3 + n^4 \le n^4 + n^4$

$4^n \neq O(2^n)$ $\qquad\qquad$ $2^{2n} = \underbrace{2^n}_{} \cdot 2^n$

Let $c > 0$, $n_0 > 0$

want to show

$$\underbrace{4^n}_{} > c \cdot \underbrace{2^n}_{} \quad \text{for all } n \geq n_0$$

want $\quad 2^n \cdot 2^n > \underbrace{c}_{}\, \underbrace{2^n}_{}$

(scratch)

$$2^n \cdot 2^n \geq \underbrace{2^c}_{\geq c} \cdot 2^n \quad \text{if } n > c$$

---

take $\quad n = \max\{c, n_0\}$

$$2^n \geq 2^c$$

$$2^n \cdot 2^n \geq 2^c \cdot 2^n$$

$$\underline{4^n} \geq 2^c \cdot 2^n$$

$$> \underline{c \cdot 2^n}$$

4 Growth of Functions and Asymptotic Behavior (CLRS §3.1)

Goal: Establish notation that enables us to compare the relative performance of different algorithms.

$T(n) = 5.5n^2 + 7.789n \le cn^2$

$$\boxed{\lim_{n\to\infty} \frac{T(n)}{g(n)} = C < \infty}$$

complicated

a "nice" function

"≤"

Definition • T(n) = O(g(n)) means there exists c > 0 such that T(n) ≤ cg(n) for sufficiently large n.

(Big-oh)

Really $O(g(n))$ is a set

$\exists n_0 \ge 1$ such that $\forall n \ge n_0$

$$O(g(n)) = \{ f: \mathbb{N} \to \mathbb{R}^+ \mid \exists c > 0 \ \forall \text{ large enough } n,\ f(n) \le cg(n) \}$$

"there exists"

$T(n)$ "≤" $g(n)$

$g(n)$ is an asymptotic upper bound for $T(n)$, $T(n)$ does not grow faster than $g(n)$ when n is large.

"≥"

• T(n) = Ω(g(n)) means there exists c > 0 such that T(n) ≥ cg(n) for sufficiently large n.

(Omega)

$$\boxed{\lim_{n\to\infty} \frac{T(n)}{g(n)} > 0}$$

$T(n)$ "≥" $g(n)$

$g(n)$ is an asymptotic lower bound for $T(n)$

• T(n) = Θ(g(n)) means there exists c_1, c_2 such that $c_1g(n) \le T(n) \le c_2g(n)$ for sufficiently large n. (Theta)

"="

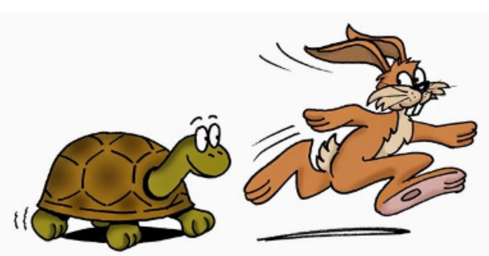
For large n, $T(n)$ and $g(n)$ are "equal" up to a constant factor

$$\boxed{\lim_{n\to\infty} \frac{T(n)}{g(n)} = C > 0}$$

"<"

• $T(n) = o(g(n))$ (Little-oh) means $\lim_{n\to\infty} \frac{T(n)}{g(n)} = 0$

$T(n)$ is negligible in comparison to $g(n)$.



Remark if $f \in \Omega(g(n))$ and $f \in O(g(n))$, then $f \in \Theta(g(n))$

Example

- polynomials – quadratic for any $a>0$, b, c $\underbrace{an^2+bn+c}_{\text{could be complicated}} \in \Theta(n^2)$ ← nice simple function
  – more generally $a_k n^k + a_{k-1} n^{k-1} + \cdots + a_0 \in \Theta(n^k)$
  $a_k > 0$
- $n \log_2 n = O(n^2)$
  in fact: $\log n = O(n^\varepsilon)$ for any $\varepsilon > 0$
  (for little-oh)
  $\lim_{n\to\infty} \frac{\ln(n)}{n} = 0$
- $2^n \log n \overset{?}{\cancel{=}} O(2^n)$
  not a constant factor
- $3^n \overset{?}{\cancel{=}} O(2^n)$
  but: $3^n = \Omega(2^n)$
- $3^n + 2^n \cancel{=} \Theta(2^n)$
  $= \Theta(3^n)$
  For all $n \geq 1$
  $\underbrace{1}_{c_1} \cdot 3^n \leq 3^n + 2^n \leq \underbrace{2}_{c_2} \cdot 3^n$
  $3^n + 2^n \leq 3^n + 3^n$

4.1 Proofs involving order of growth

added for posted notes (didn't go over in class)

Claim. $f \in O(g(n))$ if and only if $g \in \Omega(f(n))$

Proof.

We'll show if $f \in O(g(n))$ then $g = \Omega(f(n))$ ($\Rightarrow$)

Assume $f \in O(g(n))$. Then for all n sufficiently large,

$$f(n) \leq c\,g(n) \quad \text{for some } c > 0.$$

Then for all n sufficiently large,

$$g(n) \geq \frac{1}{c} f(n)$$

Thus there exists c_1 (namely $c_1 = \frac{1}{c}$) such that for n sufficiently large, $g(n) \geq c_1 f(n)$. Hence $g \in \Omega(f(n))$.
i.e. lower bound up to constant factor for sufficiently large n.

($\Leftarrow$) Similar exercise for you to try! □

Example $3n^3 + 5n^2 + 10643n \in \Theta(\ n^3\ )$

$4n^3$

$1 \cdot n^3 \le 3n^3 + 5n^2 + 10643n$ for $n \geq 0$

$3n^3 + 5n^2 + 10643n \le cn^3$

$n = 10^6$ $\quad 3 \cdot 10^{18} + 5 \cdot 10^{12} + 10643 \cdot 10^6 \le 4 \cdot 10^{18}$ $\quad c = 4$

Example Give an example of T(n) and g(n) such that $T(n) \neq o(g(n))$, but $T(n) = O(g(n))$.

$T(n) = 2n^2$

$g(n) = n^2$

$\lim_{n \to \infty} \frac{2n^2}{n^2} = 2 \neq 0$

Note: little-oh is stronger condition than big-oh.

Example $5n^2 + 11 \in o(\ n^3\ )$ or $o(n^6)$ or $o(2^n)$

(little-oh)

4.2 Asymptotic notation in equations

A set in a formula represents an anonymous function in that set.

Remark.
- $f_1, f_2 \in O(g(n))$
$\Rightarrow f_1 + f_2 \in O(g(n))$

Example $f(n) = n^3 + O(n^2)$

means <u>there exists</u> $h(n) = O(n^2)$
such that $f(n) = n^3 + h(n)$

e.g. if $f(n) = n^3 + 2n$, since $2n \in O(n^2)$,
$f(n) = n^3 + O(n^2)$.

$2n = O(n^2)$
since $\lim_{n \to \infty} \frac{2n}{n^2} = 0$ or
for large enough n, $2n \le 2n^2$ $(c = 2, n = 1)$

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

convention: implicit "for all" on left hand side of equals and "there exists" on the right hand side.

$\rightarrow$ means for <u>any</u> $f(n) = O(n)$, there exists $h(n) = O(n^2)$ such that $n^2 + f(n) = h(n)$.

intuitively: adding on any linear term to a quadratic results in still bounded above by some quadratic.

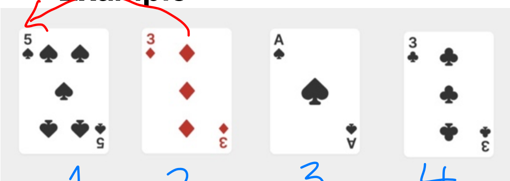
Also: (See posted MergeSort induction example)

5 Using a Loop Invariant to Prove Stability

{ HW 3 Stability of Merge Sort

Definition A sorting algorithm is **stable** if objects with equal keys appear in the same order in the sorted output as in the unsorted input.

Example



1 2 3 4

$i=2$ $j=4$

in the sorted output of a stable sorting algorithm 3 of diamonds would appear before 3 of clubs.

Note $i_1=2, j_1=4$
$i_2=1, j_2=4$
$i_3=2, j_3=4$

5.1 Stability Proof for Insertion Sort

Let i and j be initial indices with i < j such that A[i].key = A[j].key. We will show that their final positions i′ and j′ at the end of insertion sort satisfy i′ < j′.

Loop invariant: Let i_k and j_k be the positions at the start of the kth iteration of the for loop. Then $i_k < j_k$

Initialization: Before the first iteration, $i_1 = i$ $j_1 = j$, so $i_1 < j_1$

```
1. for j = 2 to n:
2.     Key ← A[j]
3.     i ← j-1
4.     while i>0 and A[i]>Key:
5.         A[i+1] ← A[i]
6.         i ← i-1
7.     A[i+1] ← Key
```

→ i & j here are different from i & j.

Maintenance:

Suppose for the inductive hypothesis that $i_k < j_k$. We will show that $i_{k+1} < j_{k+1}$

(1) If neither $A[i_k]$ or $A[j_k]$ are being inserted into the sorted portion in the kth iteration (either both already in sorted portion, or neither)

either $i_{k+1} = i_k$ and $j_{k+1} = j_k$ — (neither moves) (use inductive hypothesis to say $i_{k+1} < j_{k+1}$)

OR $i_{k+1} = (i_k) + 1$ and $j_{k+1} = (j_k) + 1$ (both shifted down 1 spot)

(2) If $A[i_k]$ is in the sorted portion and $A[j_k]$ is NOT being inserted in the kth iteration then:

$$\underbrace{i_{k+1} = \begin{cases} (i_k) + 1 \text{ or} \\ i_k \end{cases}}_{\text{either shifted down by 1 or stays}} \qquad \underbrace{j_{k+1} = j_k}_{\text{stays}}$$

(Termination:)

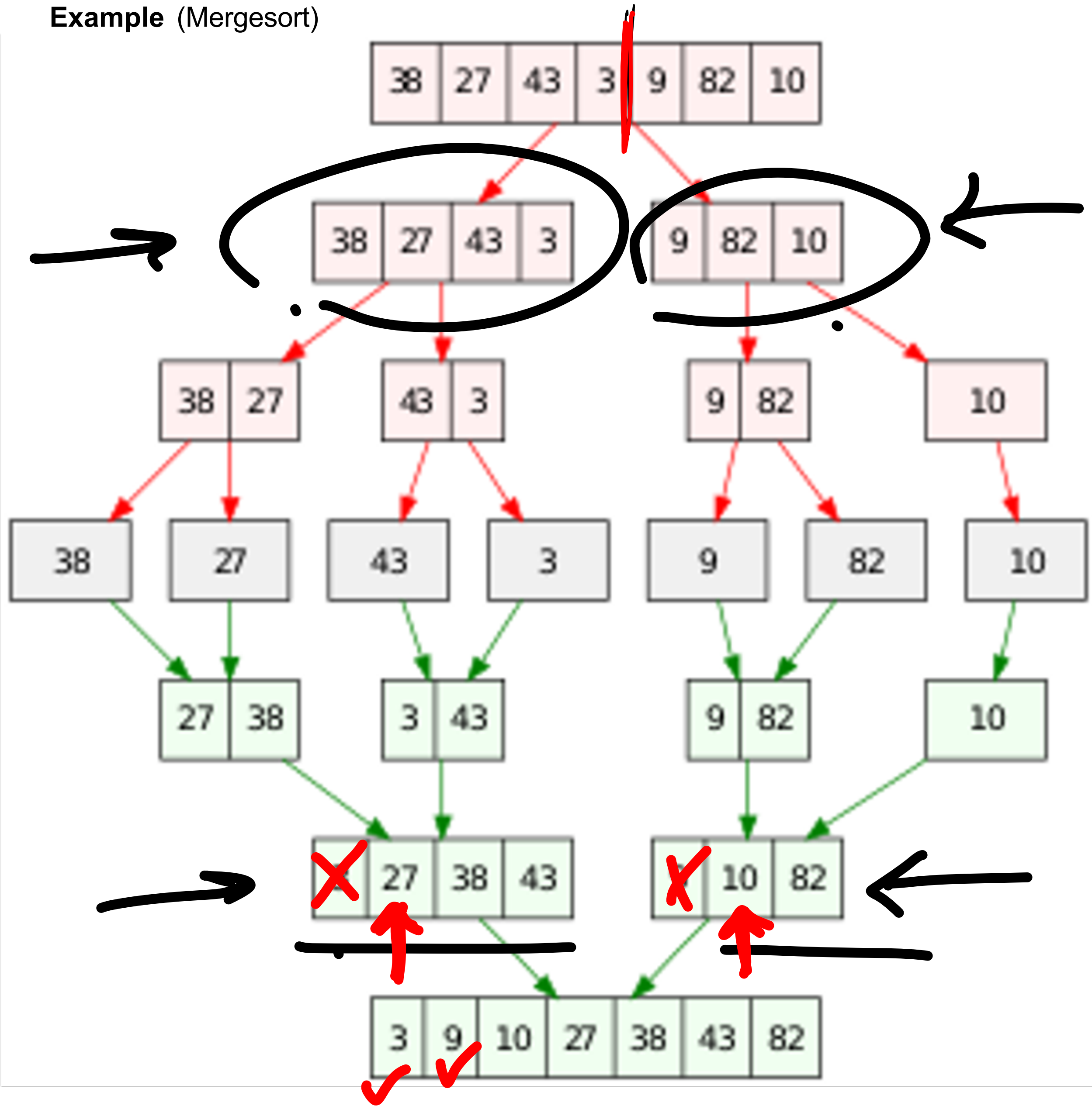
(3) If $A[j_k]$ is being inserted, then by the while loop condition on line 4, $j_{k+1} > i_{k+1}$

Note: by the inductive hypothesis, couldn't have $A[j_k]$ inserted and not $A[i_k]$.

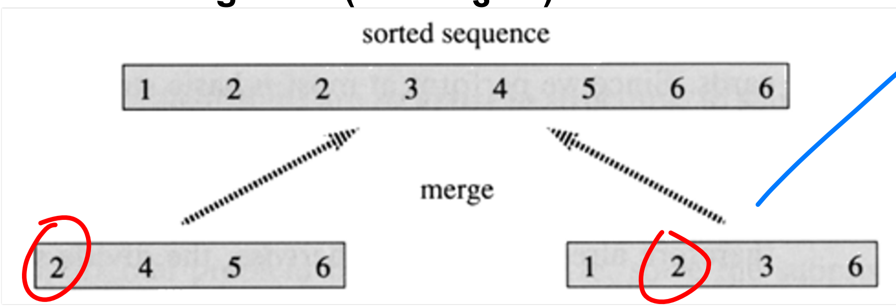
6 Divide and Conquer

- Divide the problem into subproblems similar to original but smaller in size
- Conquer each subproblem recursively
- Combine solutions to subproblems

Example (Mergesort)



6.1 Mergesort (CLRS §2.3)



repeated elements & stability - you'll prove in HW 2.

MERGESORT($A[1 \ldots n]$)
    MERGESORT($A[1 \ldots \frac{n}{2}]$)
    MERGESORT($A[\frac{n}{2}+1 \ldots n]$)
    MERGE $A[1 .. \frac{n}{2}]$ and $A[\frac{n}{2}+1 \ldots n]$

6.1.1 The Merge Subroutine

Description of the algorithm, or more English version of pseudocode:

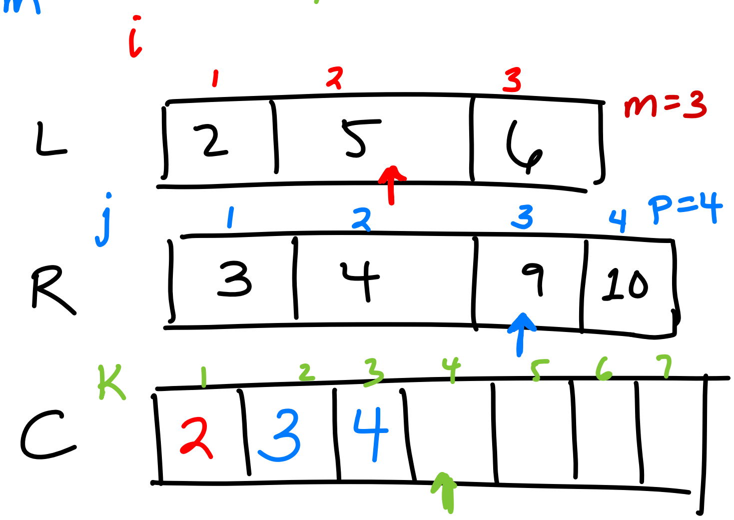
To merge sorted arrays L[1 ... m] and R[1 ... p] into array C[1 ... m+p]
    Maintain a current index for each list, each initialized to 1
    While both lists have not been completely traversed:
        Let L[i] and R[j] be the current elements
        Copy the smaller of L[i] and R[j] to C
        Advance the current index for the array from which the smaller element was selected
    EndWhile
    Once one array has been completely traversed, copy the remainder of the other array to C

m = L.length

p = R.length

```
MERGE(L, m, R, p, C)   // L[1...m] and R[1..p] are sorted arrays
i ← 1
j ← 1
for k = 1 to m+p
    if L[i] ≤ R[j] and i ≤ m
        C[k] ← L[i]
        i ← i+1
    else if j ≤ p
        C[k] ← R[j]
        j ← j+1
```

Check that in bounds of the array L



6.1.2 Proof of Correctness of Merge

Loop invariant:

At the start of the k^{th} iteration $C[1 \dots k-1]$ contains the $k-1$ smallest elements of L and R in sorted order

Those elements are from $L[1 \dots i-1]$ and $R[1 \dots j-1]$

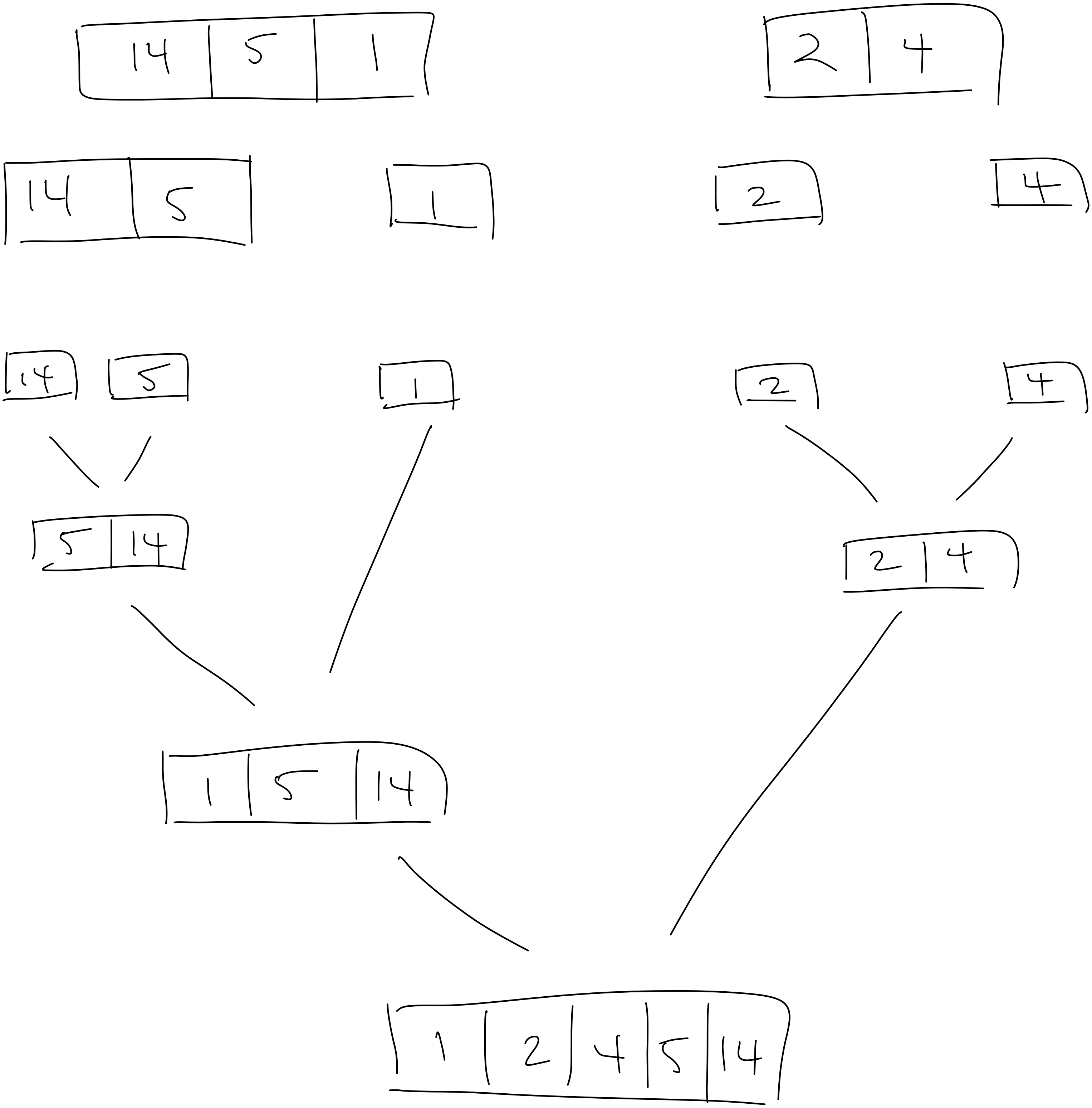
- Initialization: (Base case)

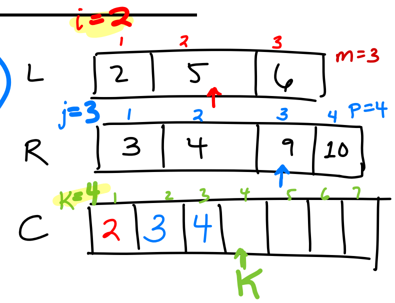
  $i = j = k = 1 \Rightarrow$ C is empty so C contains the 0 smallest elements of L and R in sorted order.

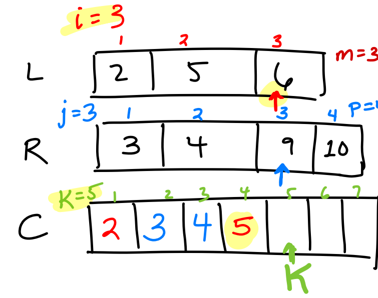
  So the invariant holds.

Desmos slide 3 : Mergesort on

14 5 1 2 4







- Maintenance:

  Assume the invariant holds for the kth iteration and current iteration (or after previous iteration) (without loss of generality) $L[i] \le R[j]$

  Then $L[i]$ is the smallest among elements from L and R not copied yet into C.

  Therefore since i and k are incremented, the invariant holds in the next iteration.

- Termination:

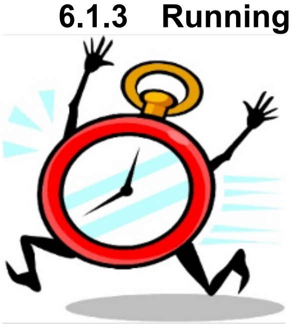
  After the last iteration $k = m+p+1$ and $i = m+1$, $j = p+1$, so C contains all ($k-1 = m+p$) elements of L and R in sorted order.

At the start of the kth iteration $C[1 \ldots k-1]$ contains the $k-1$ smallest elements of L and R in sorted order

Those elements are from $L[1 \ldots i-1]$ and $R[1 \ldots j-1]$

```
MERGE(L, m, R, p, C)
i ← 1
j ← 1
for k = 1 to m+p
    if L[i] ≤ R[j] and i ≤ m
        C[k] ← L[i]
        i ← i+1
    else if j ≤ p
        C[k] ← R[j]
        j ← j+1
```

6.1.3 Running Time of Merge (worst case)



$\Theta(m+p)$ m, p – sizes of subarrays of $A[1 \ldots n]$

$\Theta(n)$ complexity for $A[1 \ldots n]$

so far just did merge

6.1.4 Back to Mergesort: Correctness, Running Time, Recursion Tree

correctness follows from correctness of Merge and using induction on the size of the array.
(in general use induction to prove correctness of a recursive algorithm)

Running time: use recurrence - describe running time in terms of running time on smaller inputs.

let $T(n)$ be the running time on a problem of size n.

$$T(n) = \underbrace{2T\left(\tfrac{n}{2}\right)}_{\text{recursion}} + \underbrace{cn}_{\text{merge}} + \underbrace{c_1}_{\text{divide}}$$

(really $\lfloor \frac{n}{2} \rfloor$ and $\lceil \frac{n}{2} \rceil$)
floor ceiling

OR

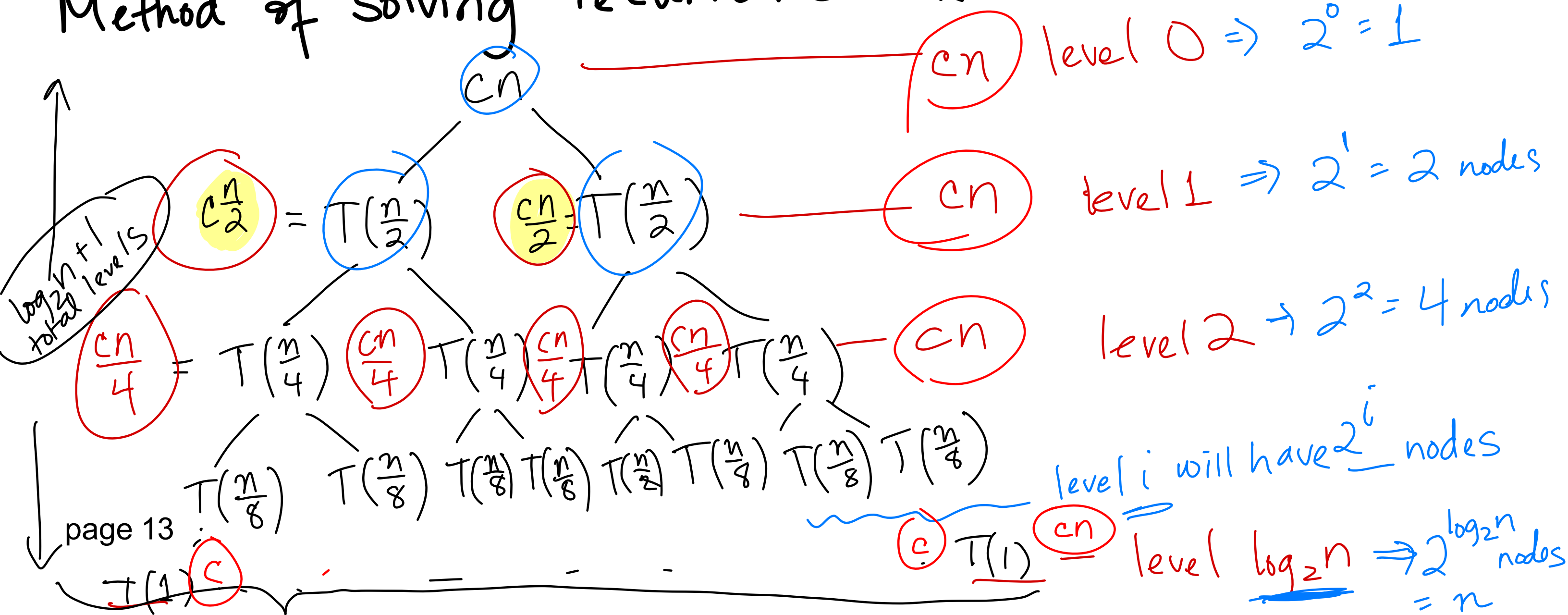
for simplicity

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ \underbrace{2T\left(\tfrac{n}{2}\right)}_{\text{size of subproblem}} + \underbrace{cn}_{\Theta(n)} & \text{if } n > 1 \end{cases}$$

number of subproblems

$T\left(\frac{n}{2}\right) = 2T\left(\frac{n}{4}\right) + c \cdot \frac{n}{2}$

Method of solving recurrence: Recursion tree expansion.
(informal but helpful)



n leaves $\left(= 2^{\log_2(n)}\right)$

Note the depth is $\boxed{\log_2 n + 1}$ since we start at level 0 and go up to level $\log_2(n)$

[For $\log_2(n)$ we also use $\lg(n)$]

Merge Sort is better than insertion Sort.

$\log(n) \le n$ for large enough

$n\log(n) \le n^2$

So overall runtime is: $T(n) = (\log_2(n) + 1) * cn = cn\log_2 n + cn$
$= \Theta(n\log_2 n)$

Proof of time complexity

Claim. For large enough $c_1 > 0$, and for all $n \geq 2$ $\boxed{T(n) \le c_1 n \log n}$ (*) (worst case and average case)

c_1 to be determined

Proof sketch.

Base case: $n=2$

$$T(2) = 2T\left(\frac{2}{2}\right) + c\cdot 2 = 2T(1) + c\cdot 2$$
$$= 2c + c\cdot 2$$
$$= 4c.$$

$n=2$ in the right hand side of (*)

$$c_1 n\log n = c_1 \cdot 2\log_2 2$$
$$= c_1 \cdot 2 = 2c_1$$

$4c \le 2c_1$ (Take $c_1 \geq 2c$)

Assume by induction that (*) holds for $n < K$.
Then for $n = K$:

$$T(K) = 2\boxed{T\left(\frac{K}{2}\right)} + c\cdot K$$
$$\le \cancel{2}\boxed{c_1 \frac{K}{\cancel{2}} \log_2\left(\frac{K}{2}\right)} + cK$$

using the inductive hypothesis on $T\left(\frac{K}{2}\right) \le c_1 \frac{K}{2}\log_2\left(\frac{K}{2}\right)$

$$= c_1 K \log_2\left(\frac{K}{2}\right) + cK$$
$$= c_1 K\left(\log_2 K - \log_2 2\right) + cK \quad (\log_2 2 \to 1)$$
$$= c_1 K\left(\log_2(K) - 1\right) + cK$$
$$= c_1 K \log_2(K) - c_1 K + cK$$
$$= c_1 K\log_2(K) - \underbrace{K(c_1 - c)}_{\geq 0}$$
$$\le c_1 K\log(K)$$

$c_1 \geq 2c$
$\Rightarrow c_1 - 2c \geq 0$
$c_1 - c - c \geq 0$

$\therefore T(K) = O(K\log K)$ (similarly prove lower bound to show Θ)

Show

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

i.e. $\underline{T(n) \geq d\, n \log n}$ for n suff large

$d > 0$

$$T(K) = 2T\left(\frac{K}{2}\right) + cn$$

$$\geq \underline{\qquad\qquad}$$

7 Solving Recurrences

We are exploring the algorithm design technique known as **Divide and Conquer**. We'll see various algorithms that use this technique.

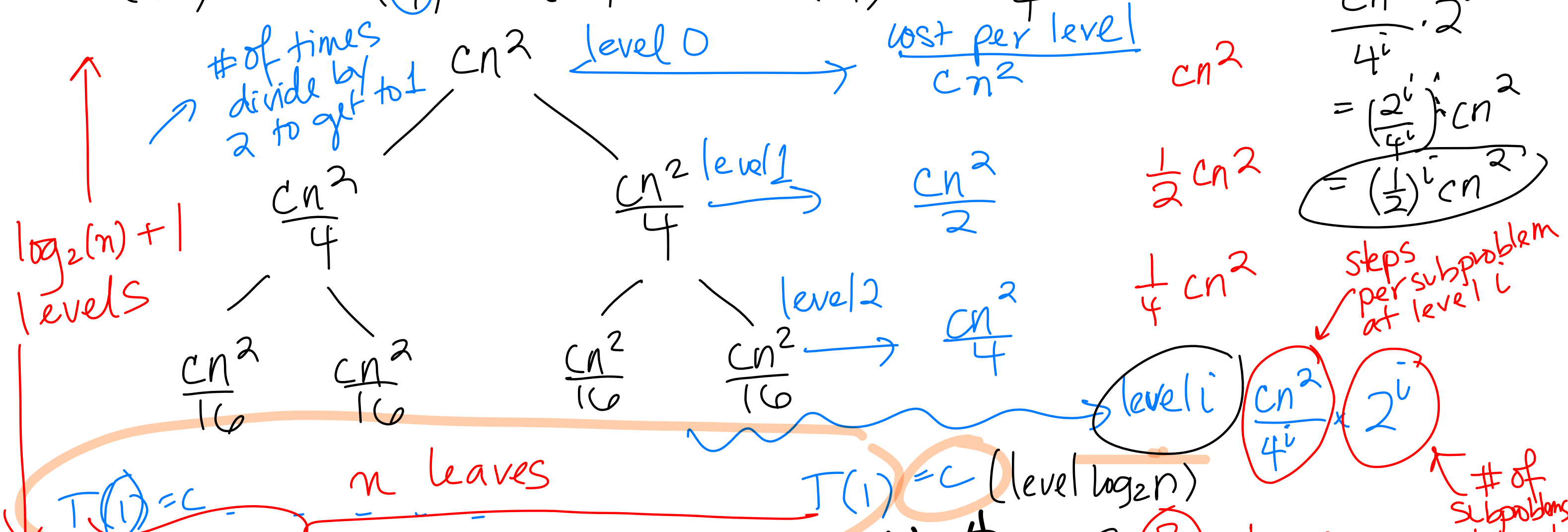
Running time analysis of such algorithms naturally involves *recurrences* since we may state the running time in terms of the running time on smaller inputs. Let's think more about solving recurrences to help us determine the running time of such algorithms!

[HW2 Q4]

7.1 Using the Recursion Tree Method (CLRS §4.4)

Example $T(n) = 2T(n/2) + cn^2$, $T(1) = c$ Desmos slide 3 ? : $n^2 \log n$

$$T\left(\frac{n}{2}\right) = 2T\left(\frac{n}{4}\right) + c\left(\frac{n}{2}\right)^2 = 2T\left(\frac{n}{4}\right) + c\frac{n^2}{4}$$



~~**Example** $T(n) = 2T(n/2) + c$, $T(1) = c$ Desmos slide 4 ? $\log n$~~

cn

$$\underbrace{\frac{cn^2}{4^i}}_{\text{Steps per subproblem at level } i} \times \overbrace{2^i}^{\text{\# of subproblems at level}} = \left(\frac{1}{2}\right)^i cn^2 \Rightarrow \text{Total steps across all levels}$$

$$= \sum_{i=0}^{\log_2(n)-1} \left(\frac{1}{2}\right)^i cn^2 + cn$$

$$= cn^2 \sum_{i=0}^{\log_2(n)-1} \left(\frac{1}{2}\right)^i + cn$$

sum of Geometric sequence $\frac{1}{1-r}$, $r = 1/2$

$$\le cn^2\underbrace{\left(1 + \frac{1}{2} + \frac{1}{4} + \cdots\right)}_{} + cn$$

$$= cn^2 \frac{1}{1-1/2} = 2cn^2 + cn = \Theta(n^2)$$

work at the leaves:

$\left(\frac{1}{2}\right)^i cn^2$ when $i = \log_2(n)$

$= \left(\frac{1}{2}\right)^{\log_2 n} cn^2 = n^{-1} \cdot cn^2 = cn$

7 Solving Recurrences

We are exploring the algorithm design technique known as **Divide and Conquer**. We'll see various algorithms that use this technique.

Running time analysis of such algorithms naturally involves *recurrences* since we may state the running time in terms of the running time on smaller inputs. Let's think more about solving recurrences to help us determine the running time of such algorithms!

7.1 Using the Recursion Tree Method (CLRS §4.4)

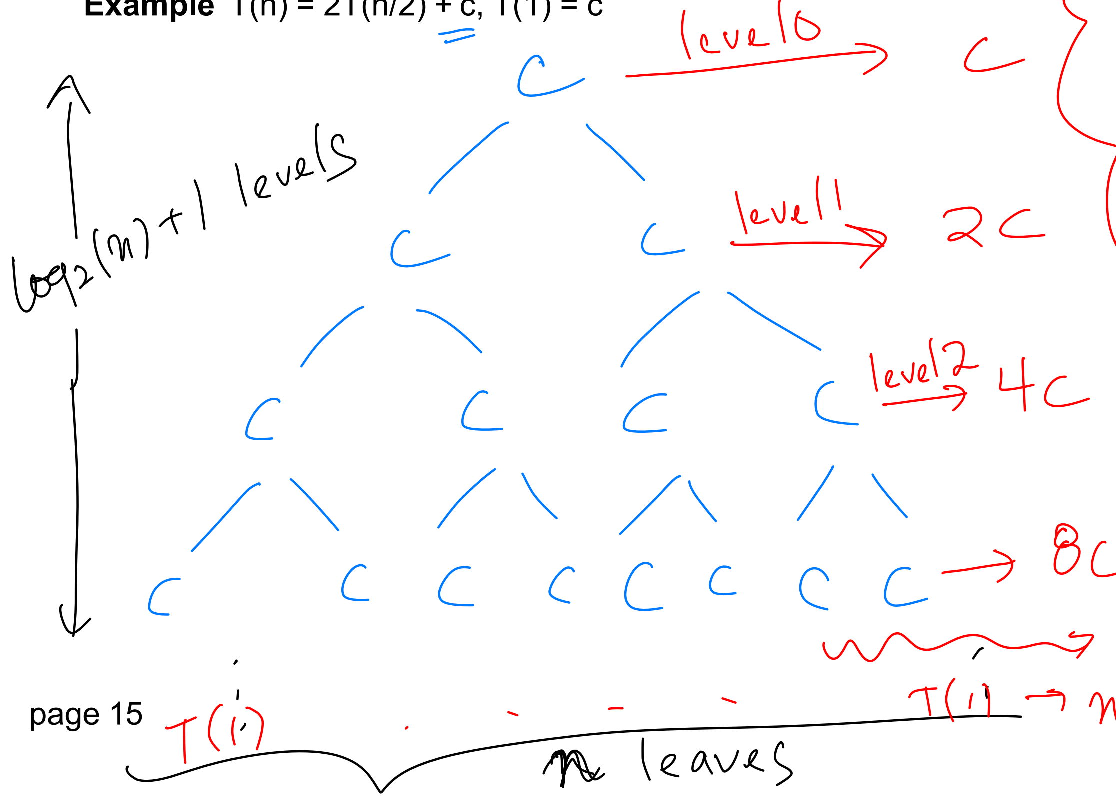
Example $T(n) = 2T(n/2) + cn^2, T(1) = c$ $\longrightarrow T(n) = \Theta(n^2)$

$\underline{c} \rightarrow T(n) = \Theta(n)$

$\underline{cn} \rightarrow T(n) = \Theta(n \log(n))$

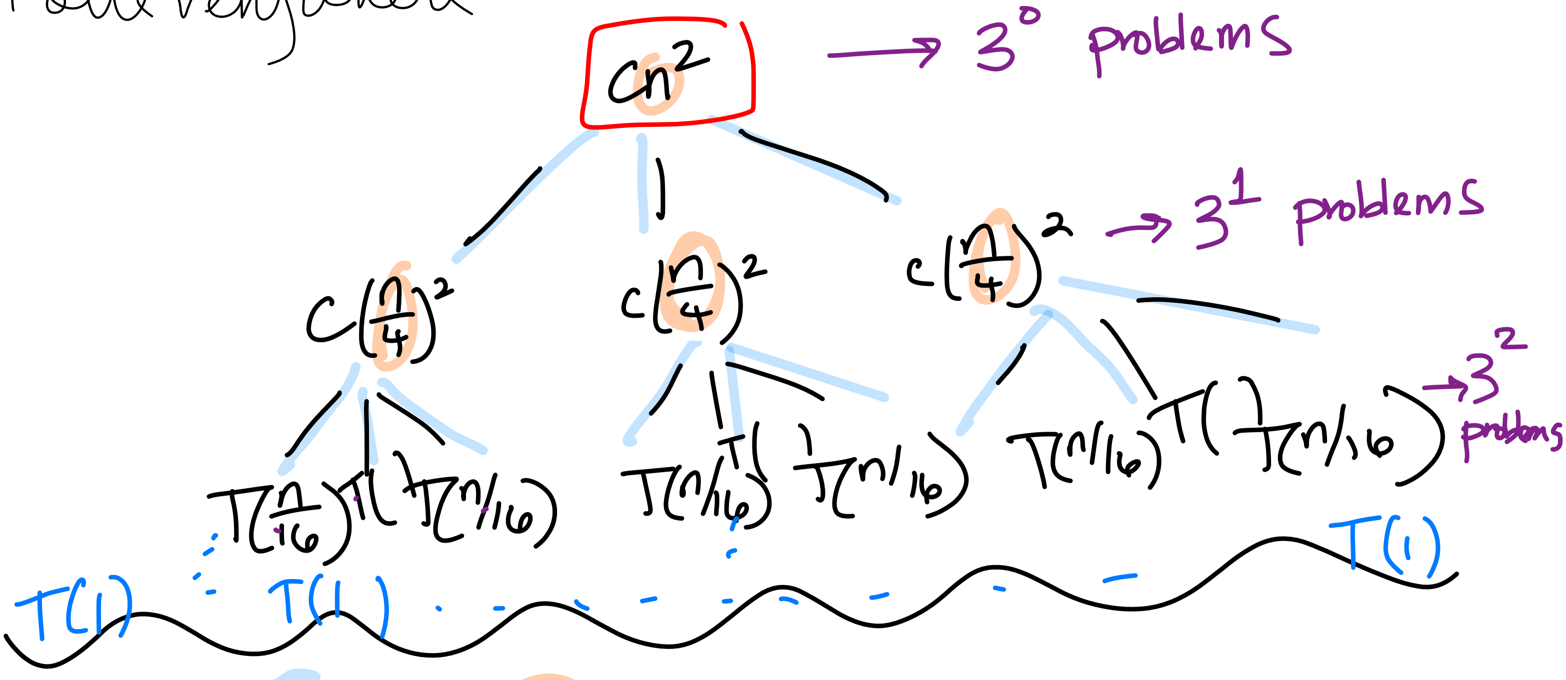
$r \neq 1$: $\sum_{k=0}^{n} r^k = \left(\frac{1-r^{n+1}}{1-r}\right)$

Example $T(n) = 2T(n/2) + c, T(1) = c$



Total # of steps $= \sum_{i=0}^{\log_2 n} 2^i c = c \sum_{i=0}^{\log_2(n)} 2^i = \Theta(n)$ (HW) Q1 part 4

Pollev everywhere



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

How many leaves do we end up with?

(1) What is the last level?

$\log_4 n$

$\frac{n}{4^i}$ is the size of the subproblem at level i

$\frac{n}{4^d} = 1$ ← d = last level

$n = 4^d \Rightarrow d = \log_4 n$

(2) At level i, we have 3^i problems so at level $\log_4 n$, we have $3^{\log_4 n}$ problems (how many leaves we have)

equal to $n^{\log_4 3}$ ←

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

helpful for analyzing Θ bounds (big-oh)

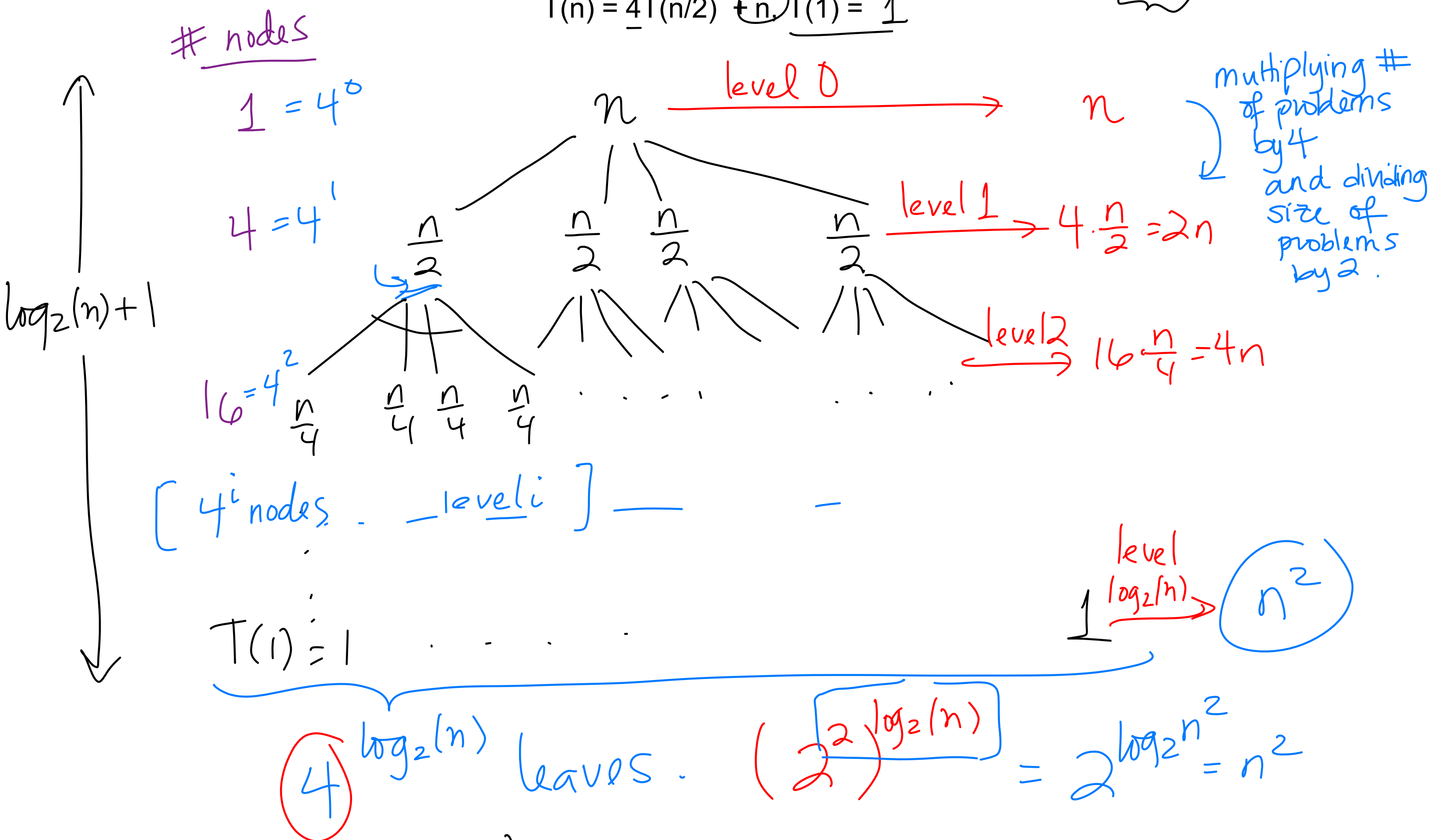
HW2 Q5

$$T(\tfrac{n}{2}) = 4T(\tfrac{n}{4}) + \tfrac{n}{2}$$
$$T(\tfrac{n}{4}) = 4T(\tfrac{n}{16}) + \tfrac{n}{4}$$

7.2 The substitution method (CLRS §4.3)

Example

T(n) = 4T(n/2) + n, T(1) = 1



Guess $T(n) = \Theta(n^2)$.

Attempt 1: $T(n) \le cn^2$ for $c > 0$, $n \ge 2$

Base case: $n = 2$. using the recurrence, $T(2) = 4T(1) + 2 = 4 + 2 = 6$

$T(n) \le cn^2 \rightarrow$ for $n = 2$ $\qquad 6 \le 4c$, so pick $c \ge 2/3$

Inductive step: Suppose $T(n) \le cn^2$ for $2 \le n < k$

Consider $T(k) = 4\boxed{T(\frac{k}{2})} + k$

$\le 4\boxed{c(\frac{k}{2})^2} + k$ by the inductive hypothesis

$\le ck^2 + k$ $\quad = \Theta(k^2)$ WRONG!

We needed to show $T(k) \le ck^2$

Note: we need the exact form of the inequality to complete the induction, because c cannot depend on K.

7.2.1 Careful with asymptotic notation and bogus proofs! [HW 3]

① $T(n) = 2T(\frac{n}{2}) + n$ ← Mergesort Runtime $T(n) = \Theta(n \log n)$

"Claim" $T(n) = O(n)$

"proof" (inductive step) $T(K) = 2T(\frac{K}{2}) + K$ (Assume true for $n < K$)

$= 2\,O(K) + K = O(K)$ WRONG!!

7.2.2 Ceilings and floors [HW3]

Example

$$T(n) = T(\lceil n/2 \rceil) + 1, \; T(1) = 1$$

Substitution method for solving recurrences

Slide 3

Try using substitution to show $T(n) \leq cn^2$ for the recurrence $T(n) = 4T\left(\frac{n}{2}\right)+n^2$.

(it doesn't work out!)

Bogus proofs

Slide 4

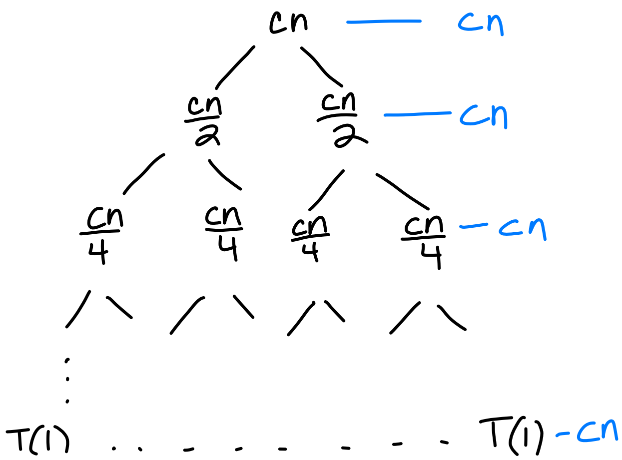
$T(n) = 2T\left(\frac{n}{2}\right)+n$

claim: $T(n) = O(n)$

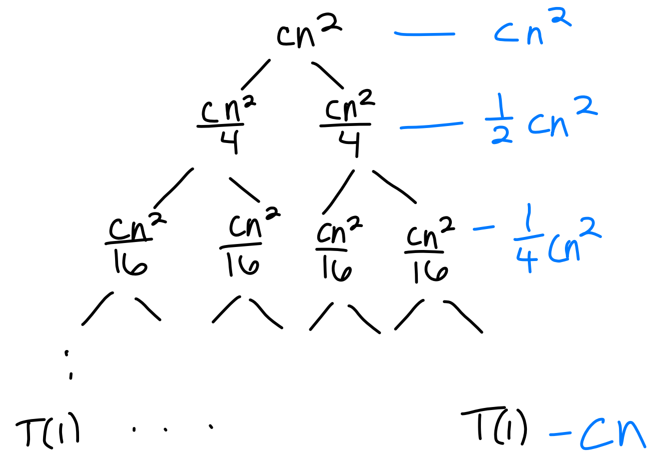
proof: $T(k) = 2T\left(\frac{k}{2}\right)+k = 2O(k) + k = O(k)$

Hm... that doesn't seem right, especially since we said $T(n) = \Theta(n \log n)$. What's wrong with the proof above?

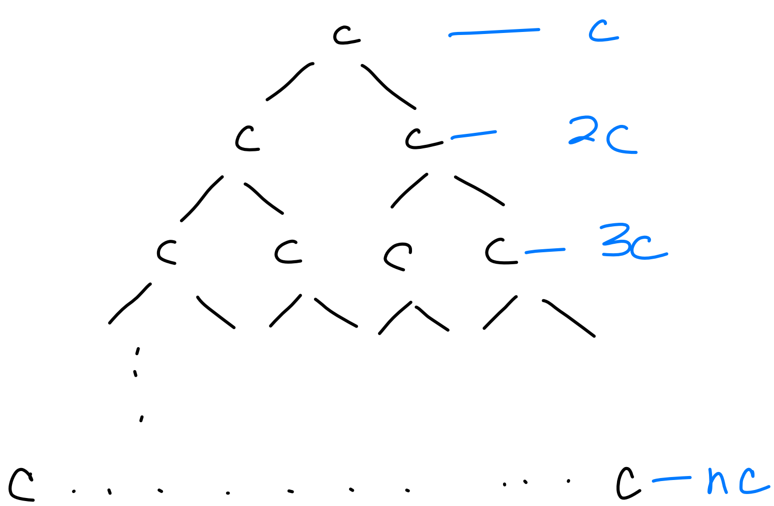
$$T(n) = 2T(n/2) + cn$$



$$T(n) = 2T(\tfrac{n}{2}) + cn^2$$



$$T(n) = 2T(\tfrac{n}{2}) + c$$



7.3 Master Theorem (CLRS §4.5)

Let $a \geq 1$ and $b > 1$ be constants, $f(n)$, $T(n)$ a function defined on natural numbers by the recurrence:

$$T(n) = aT(n/b) + f(n)$$

We interpret n/b to mean either $\lceil n/b \rceil$ or $\lfloor n/b \rfloor$. Then, $T(n)$ has the following asymptotic bounds:

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

Desmos slides 6-9

Example Use the Master theorem to solve the following:

1. $T(n) = 9T(\lceil n/3 \rceil) + n$
2. $T(n) = T(\lceil 2n/3 \rceil) + 1$
3. $T(n) = T(\lceil n/2 \rceil) + n^2$

What is an example of a recurrence that does not fit the form of the Master theorem?

8 Quicksort (CLRS §7.1, 7.2)

Idea:

Example

```
k=PARTITION
QUICKSORT
QUICKSORT
```

8.1 Partition

```
pivot
i
for j = 1 to n-1
      if A[j]

            i

RETURN i
```

8.2 Correctness of Partition (and Quicksort)

Loop invariant:

- Initialization:

- Maintenance:

- Termination:

8.3 Running Time of Partition and Quicksort