Also: (See posted MergeSort induction example)

5 Using a Loop Invariant to Prove Stability

{ HW 3 Stability of MergeSort

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 $i_K < j_K$

```
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

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

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) → (use inductive hypothesis to say $i_{K+1} < j_{K+1}$)

either $i_{K+1} = i_K$ and $j_{K+1} = j_K$ — (neither moves)

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:

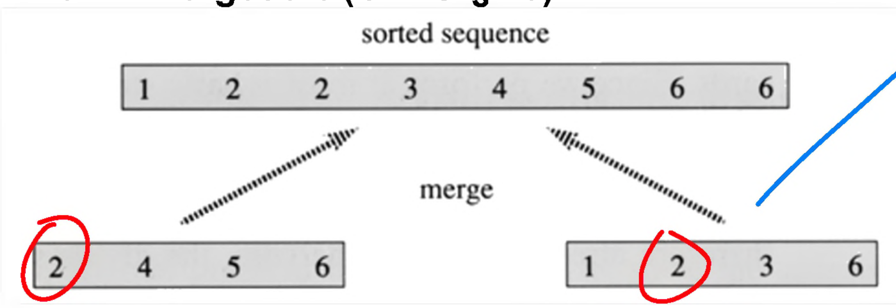
$i_{K+1} = \underbrace{\begin{cases} (i_K)+1 \text{ or} \\ i_K \end{cases}}_{\text{either shifted down by 1 or stays}}$ $\underbrace{j_{K+1} = j_K}_{\text{stays}}$

(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 $A[i_K]$.

6.1 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 $A[1 \ldots \frac{n}{2}]$ and $A[\frac{n}{2}+1 \ldots n]$

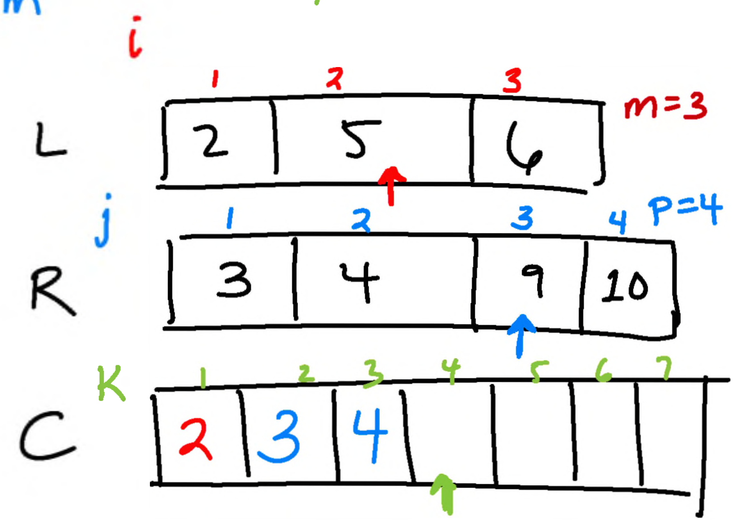
6.1.1 The Merge Subroutine

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        ← Check that in bounds of the array L
        C[k] ← L[i]
        i ← i + 1
    else if j ≤ p
        C[k] ← R[j]
        j ← j + 1
```



6.1.2 Proof of Correctness of Merge

Loop invariant:

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

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

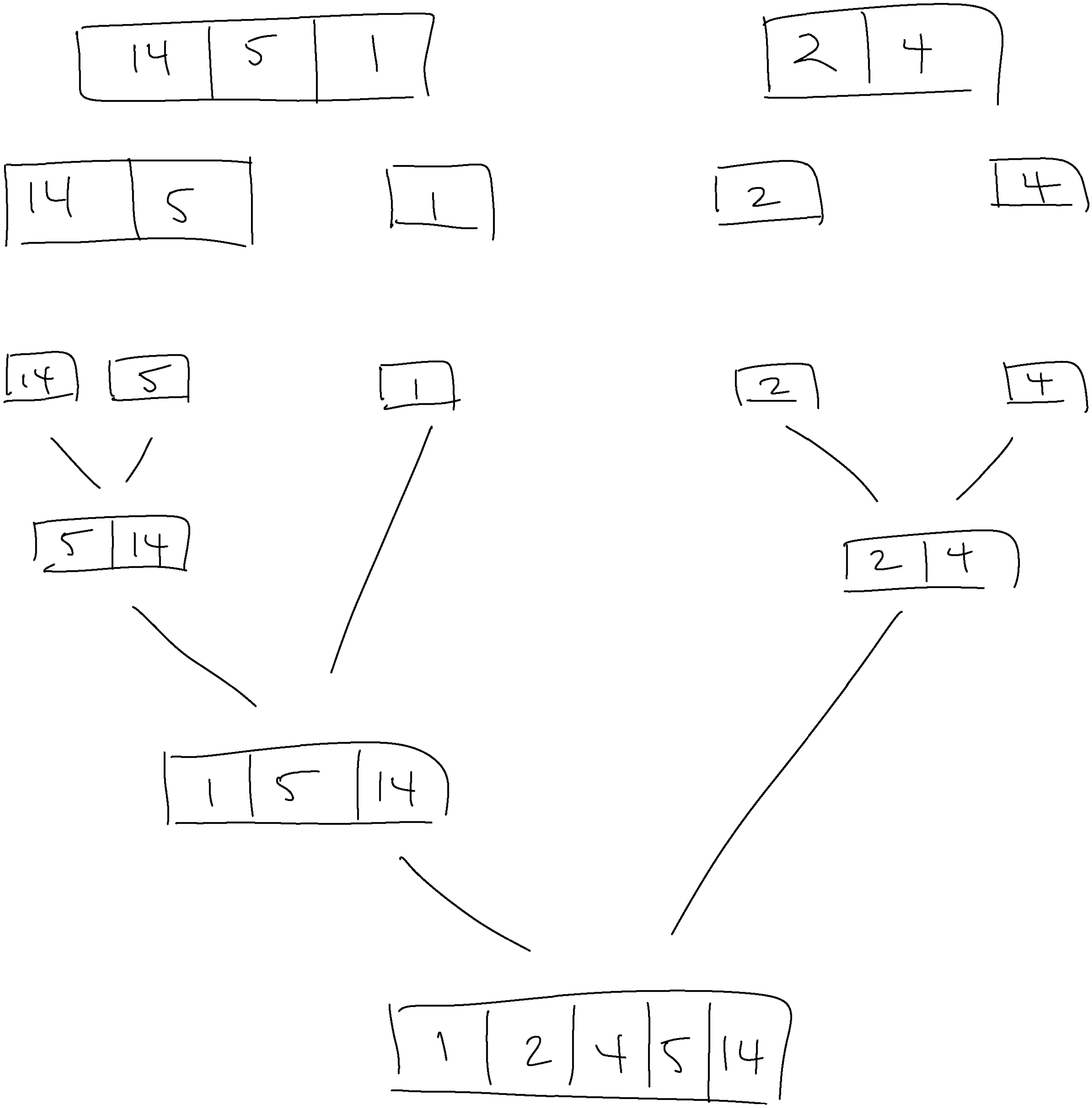
- Initialization: (Base case)

  $i=j=k=1 \Rightarrow$ 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



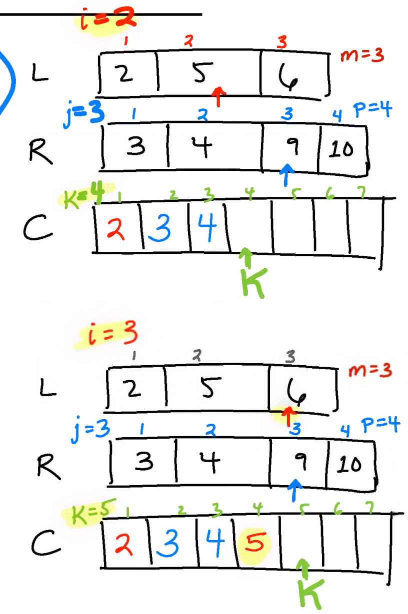
- Maintenance:

current iteration (or after previous iteration)

Assume the invariant holds for the kth iteration and (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:

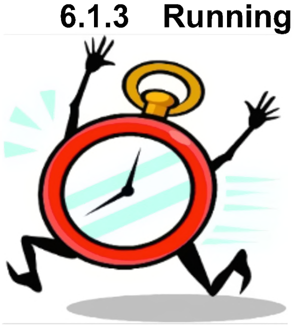
After the last iteration $k = m+p+1$ and $i = m+1$, $j = p+1$, so C contains <u>all</u> ($k-1 = m+p$) elements of L and R in sorted order.

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 $\left\lfloor \frac{n}{2} \right\rfloor$ and $\left\lceil \frac{n}{2} \right\rceil$) floor ceiling

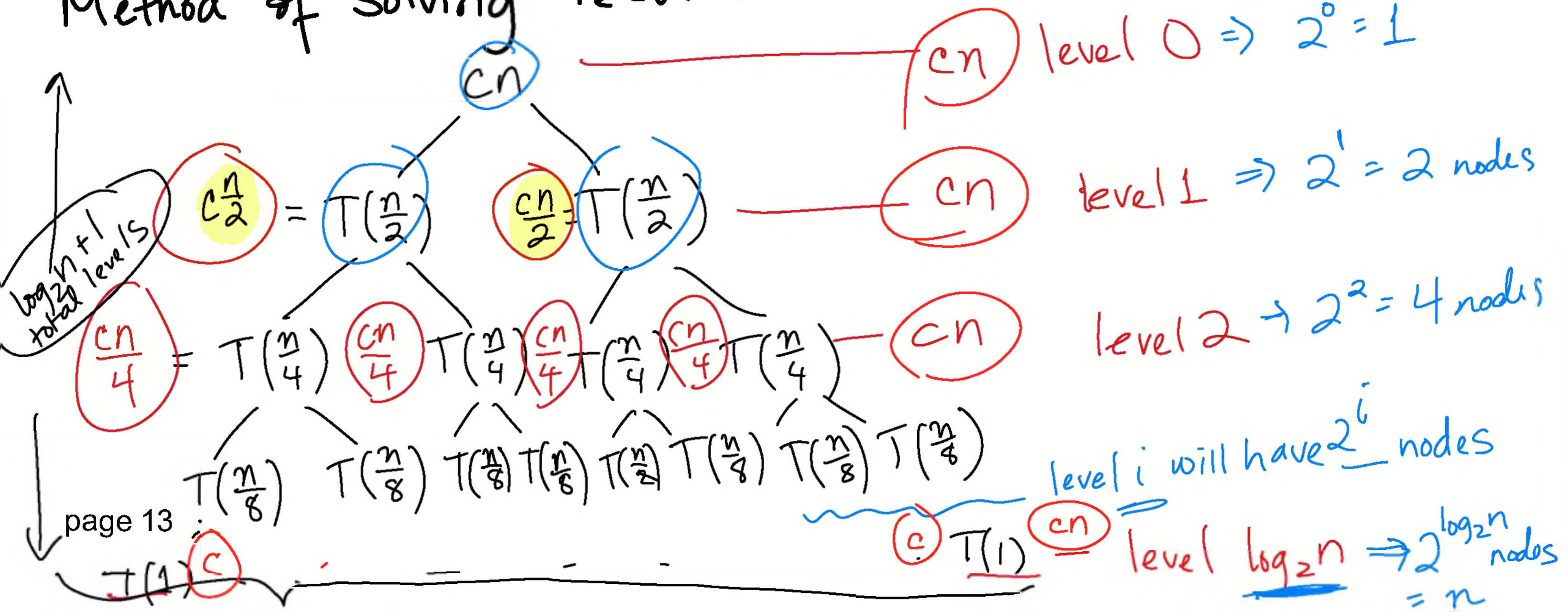
OR

for simplicity

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

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

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 $n \ge 2$ $\boxed{T(n) \le c_1 n \log n}$ (*) (worst case and average case)

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 $n = K$:

$$T(K) = 2\boxed{T\left(\tfrac{K}{2}\right)} + c\cdot K$$

$$\le 2\boxed{c_1 \frac{K}{2} \log_2\left(\frac{K}{2}\right)} + cK$$

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

$$= c_1 K \log_2\left(\frac{K}{2}\right) + cK$$

$$= c_1 K\left(\log_2 K - \underbrace{\log_2 2}_{1}\right) + cK$$

$$= c_1 K (\log_2(K) - 1) + cK$$

$$= c_1 K \log_2(K) - c_1 K + cK$$

$$= c_1 K\log_2(K) - \underbrace{K(c_1 - c)}_{\ge 0}$$

$$\le c_1 K \log(K)$$

$c_1 \geq 2c$

$\Rightarrow c_1 - 2c \geq 0$

$c_1 - c - c \geq 0$

$\therefore T(K) = \Theta(K\log K)$ (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.

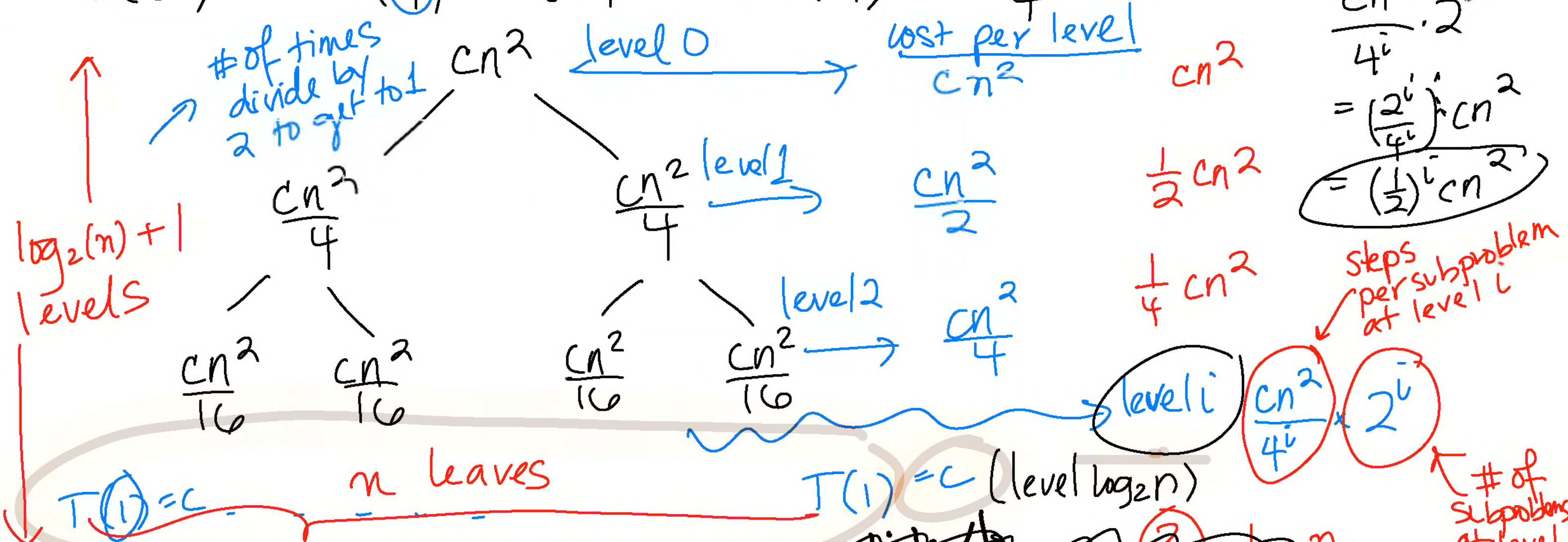
Running time analysis of such algorithms naturally involves *recurrences* since we may state the running time in terms of the running time on smaller inputs. Let's think more about solving recurrences to help us determine the running time of such algorithms!

[HW2 Q4]

7.1 Using the Recursion Tree Method (CLRS §4.4)

Example T(n) = 2T(n/2) + cn², T(1) = c Desmos slide 3 ? : $n^2 \log n$

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



$$\frac{cn^2}{4^i}\cdot 2^i = \left(\frac{2^i}{4^i}\right)cn^2 = \left(\frac{1}{2}\right)^i cn^2$$

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

# of subproblems at level

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

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

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

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

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

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

work at the leaves:

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

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

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)

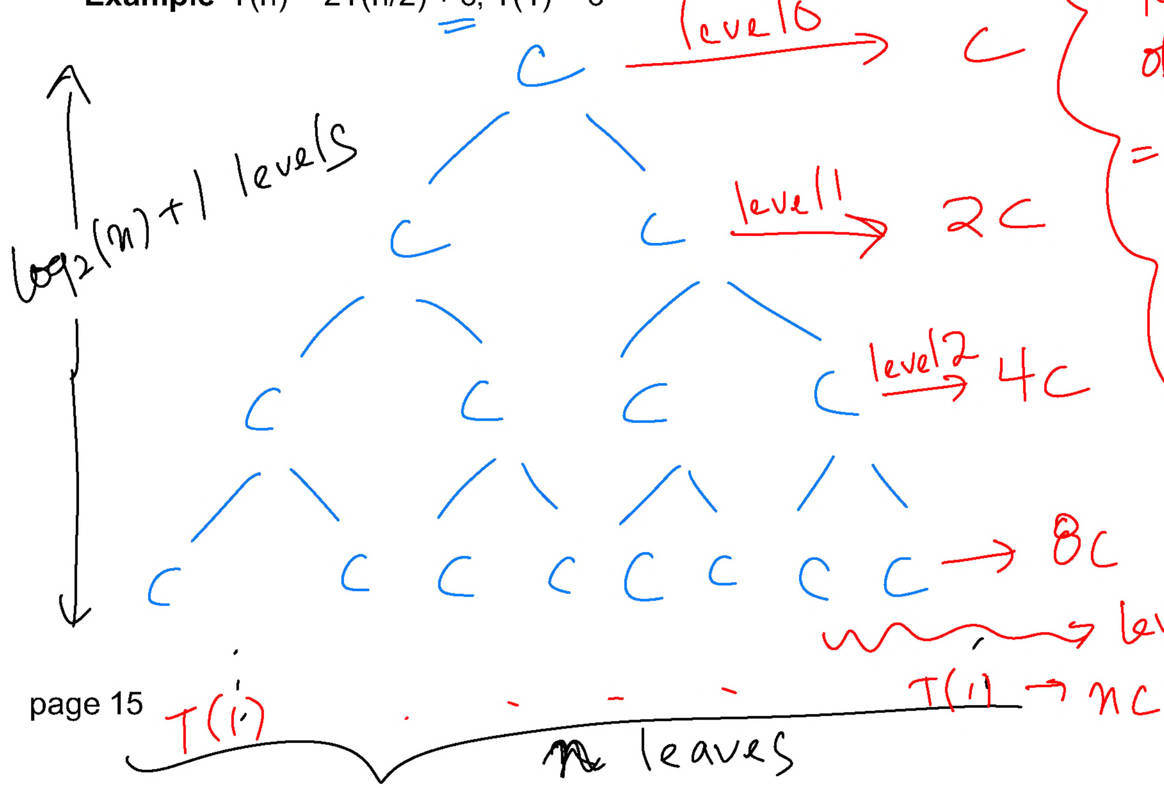
Example T(n) = 2T(n/2) + cn², T(1) = c → $T(n) = \Theta(n^2)$

$c \rightarrow T(n) = \Theta(n)$

$cn \rightarrow T(n) = \Theta(n \log(n))$

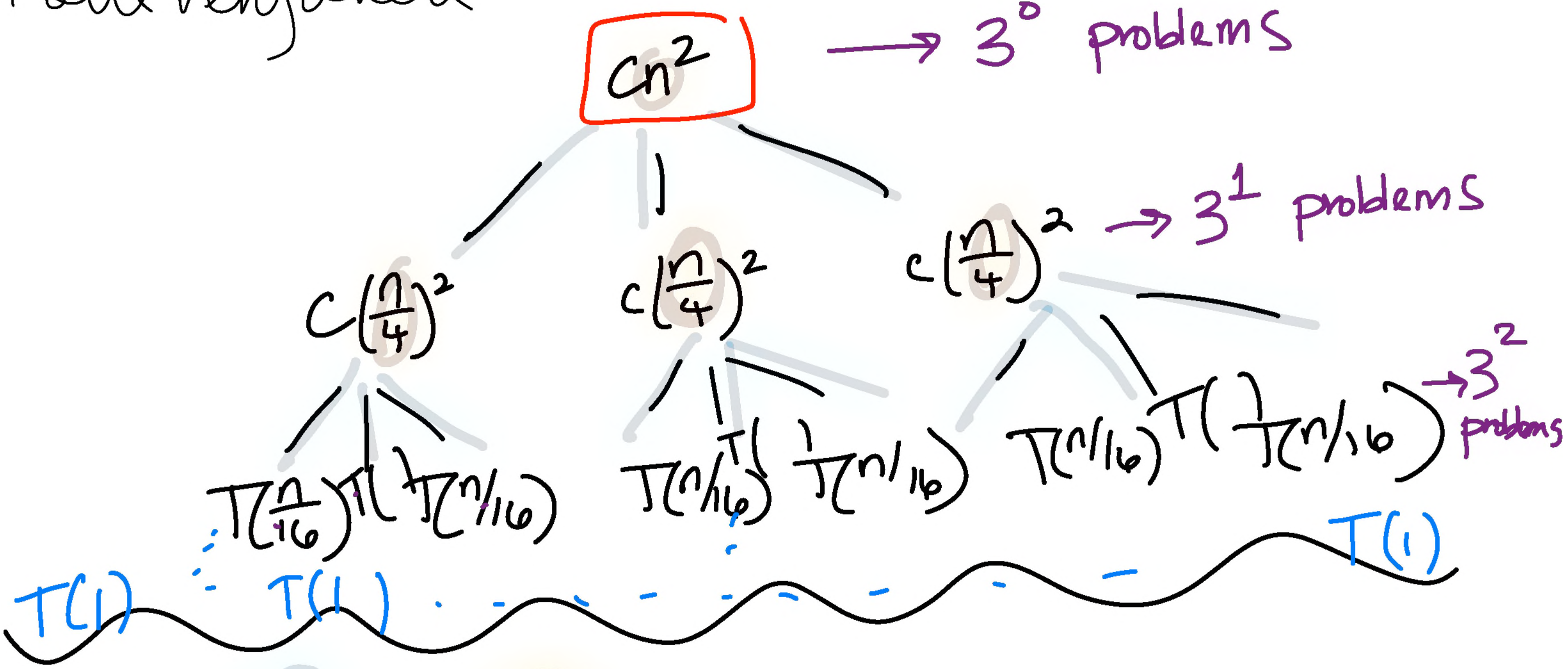
$r \neq 1$: $\sum_{k=0}^{n} r^k = \left(\frac{1-r^{n+1}}{1-r}\right)$

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



Total # of steps $= \sum_{i=0}^{\log_2 n} 2^i c = c \sum_{i=0}^{\log_2(n)} 2^i = \Theta(n)$ (HW1 Q1 part 4)

Pollevenywhere



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

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

equal to $n^{\boxed{\log_4 3}}$ ← $3^{\log_4 n}$ problems (how many leaves we have)

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

helpful for analyzing Θ bounds (big-oh)

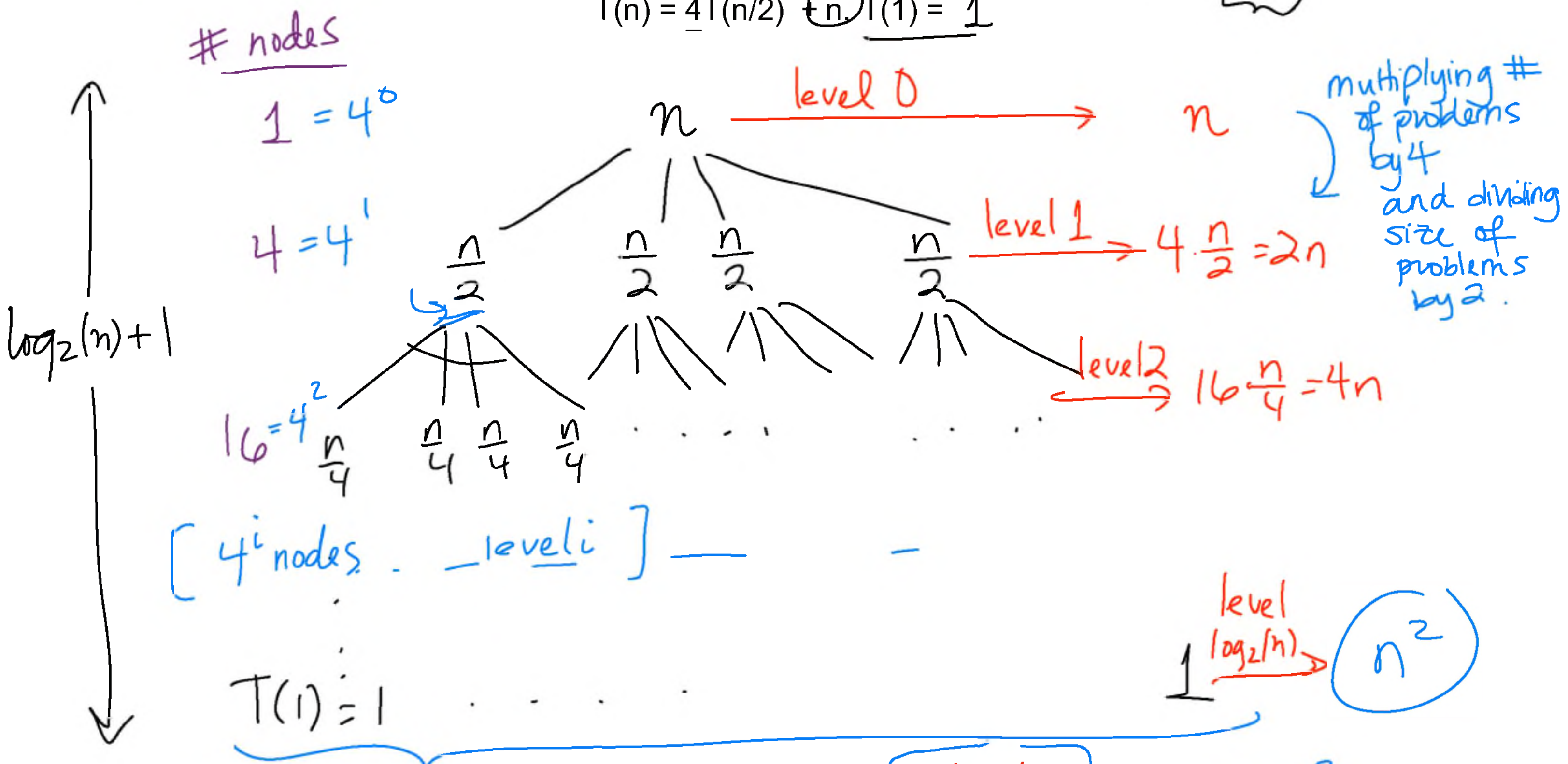
HW2 Q5

7.2 The substitution method (CLRS §4.3)

Example

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

$$T(n) = 4T(n/2) + n, \quad T(1) = 1$$



$4^{\log_2(n)}$ leaves. $\left(2^2\right)^{\log_2(n)} = 2^{\log_2 n^2} = n^2$

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, $T(2) = 4T(1) + 2 = 4 + 2 = 6$

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

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

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

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

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