Worst-case Running Time

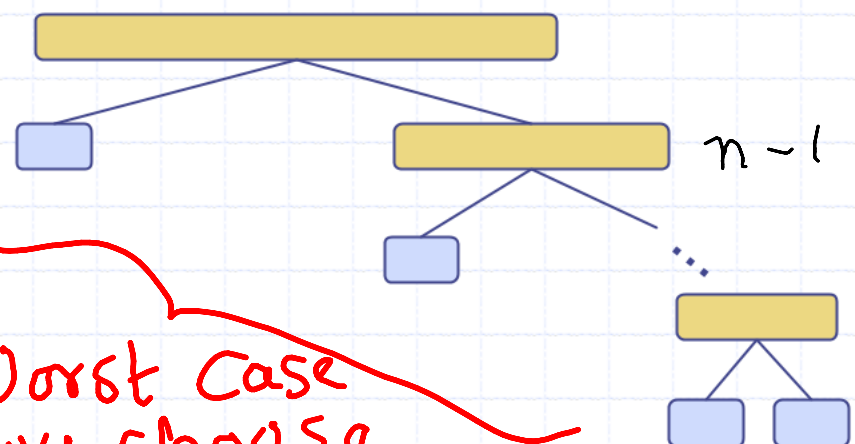
- The worst case for quick-sort occurs when the pivot is the unique minimum or maximum element
- One of L and G has size $n - 1$ and the other has size 0
- The running time is proportional to the sum

$$n + (n - 1) + \ldots + 2 + 1 = O(n^2)$$

- Thus, the worst-case running time of quick-sort is $O(n^2)$

For worst case, functions like selection sort

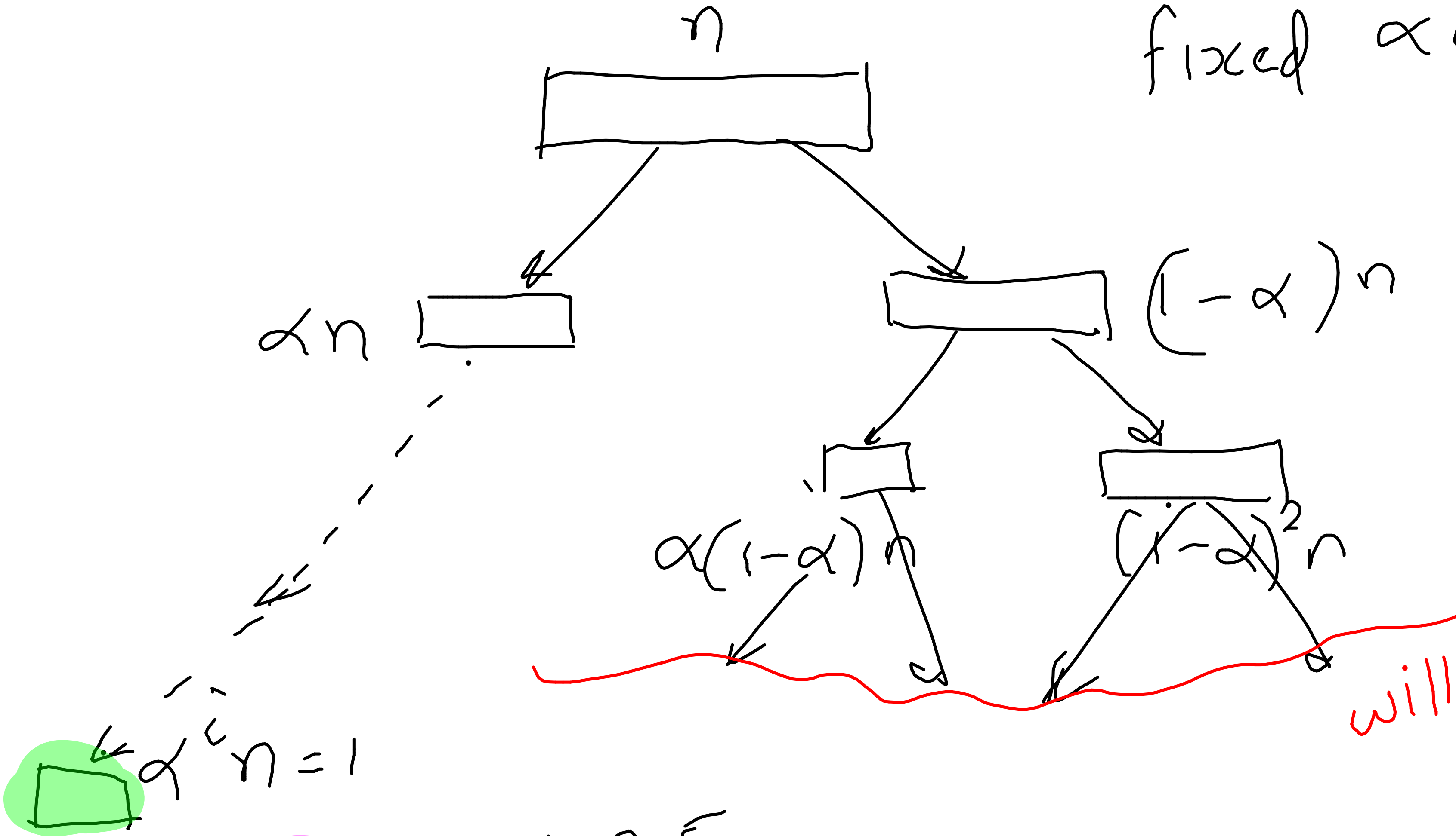
depth	time
0	n
1	$n - 1$
…	…
$n - 1$	1



$n-1$

$n-i$

Worst case you choose smallest element as pivot at each step fixed $\alpha \in (0,1)$



If wlog $\alpha > 0.5$

without loss of generality

then longest branch will have

$$\alpha^h n = 1 \Rightarrow h = O(\log_{1/\alpha} n)$$
$$= O(\log_2 n)$$

You can prove that Qsort will be $O(n \log n)$ in this case

Thus: Most Qsort partitions are not bad.

Let us consider amortized analysis of Qsort

Assume: Fixed array of length n we will average over all choices of pivot

Let $T(n)$ = average time for quicksort, average over all choices (n in number) of pivot

Recurrence relation:

$$T(n) = \frac{1}{n}\Big[\underbrace{T(n-1)+T(0)+n}$$
$$+T(n-2)+T(1)+n$$
$$\vdots$$
$$+T(n-k-1)+\underbrace{T(k)+n}$$
$$\vdots$$
$$+T(0)+T(n-1)+n\Big]$$

Note:
1. n is reqd for swaps
2. The pivot element need not be further analysed

$(n-k-1)+k$
$= n-1$

$$T(n) = \left(\frac{2}{n}\sum_{i=0}^{n-1} T(i)\right) + n \rightarrow A$$

$$T(n-1) = \left(\frac{2}{n-1}\sum_{i=0}^{n-2} T(i)\right) + n-1 \rightarrow B$$

Rescaling B and substituting in A

$$T(n) = \frac{2}{n}T(n-1) + \frac{n-1}{n}T(n-1) - \frac{(n-1)^2}{n} + n$$

$$= \frac{n+1}{n}T(n-1) + \left(2 - \frac{1}{n}\right) \rightarrow C$$

Replacing n with n-1

$$T(n-1) = \frac{n}{n-1}T(n-2) + \left(2 - \frac{1}{n-1}\right) \rightarrow D$$

Substituting for T(n-1) from D into C

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

$$= \frac{n+1}{n-1}T(n-2) + \frac{2n}{n} + \frac{2}{n} - \frac{n+1}{n(n-1)} + 2 - \frac{1}{n}$$

$$= \frac{n+1}{n-1}T(n-2) + \frac{2n}{n} + \frac{1}{n} - \frac{n+1}{n^2-n} + 2$$

$$\vdots$$

$$= \frac{n+1}{n-i+1} \cdot T(n-i) + 2(n+1)\left(\frac{1}{n} + \frac{1}{n-1} + \dots + \frac{1}{n-i+1}\right) + 2$$

$$\vdots$$

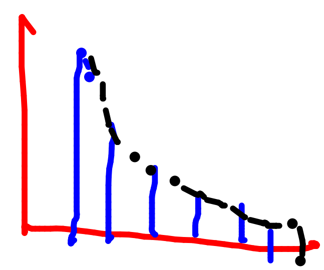
$$= \frac{n+1}{1} T(0) + 2(n+1)\left(\frac{1}{n} + \dots + \frac{1}{2} + 1\right) + 2$$

$T(0) = 0$

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

$$\le \left(2(n+1)\int_1^n \frac{1}{x}\,dx\right) + 2$$

(Area below curve is $\geq$ sum of blue bars)



$$= 2(n+1)\log n + 2$$

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

You could also substitute $T(n) = O(n\log n)$ into ⓒ & prove that they are consistent by using recurrence

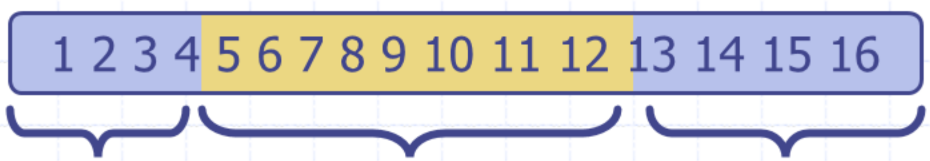
(Optional Reading: Another loose way of amortized analysis)

Expected Running Time

- Consider a recursive call of quick-sort on a sequence of size s
  - **Good call:** the sizes of L and G are each less than $3s/4$
  - **Bad call:** one of L and G has size greater than $3s/4$

Good call **Bad call**

- A call is good with probability $1/2$
  - 1/2 of the possible pivots cause good calls:

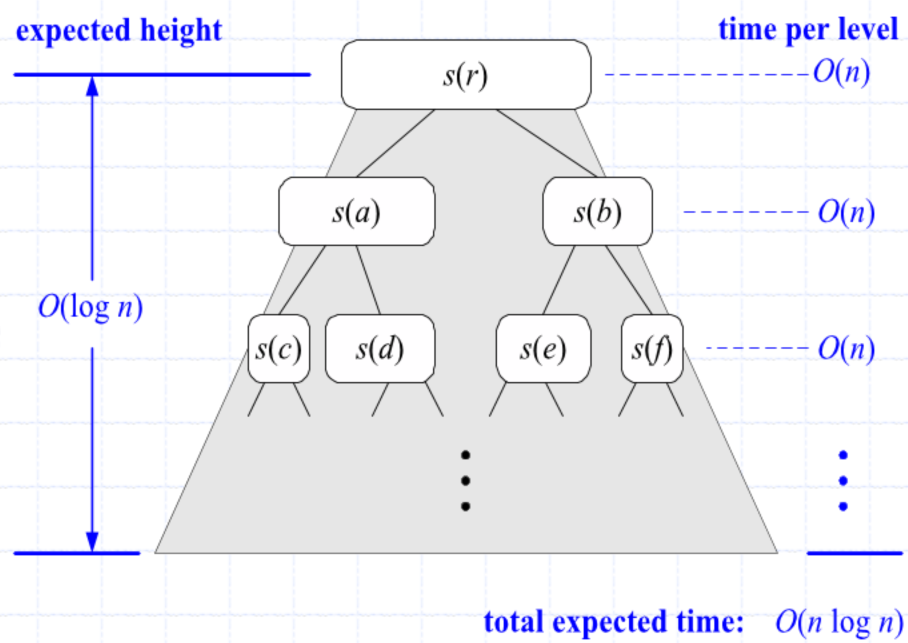


Bad pivots **Good pivots** **Bad pivots**

(Optional reading

Expected Running Time, Part 2

- Probabilistic Fact: The expected number of coin tosses required in order to get k heads is $2k$
- For a node of depth i, we expect
  - $i/2$ ancestors are good calls
  - The size of the input sequence for the current call is at most $(3/4)^{i/2}n$
- Therefore, we have
  - For a node of depth $2\log_{4/3}n$, the expected input size is one
  - The expected height of the quick-sort tree is $O(\log n)$
- The amount or work done at the nodes of the same depth is $O(n)$
- Thus, the expected running time of quick-sort is $O(n \log n)$



Summary of Sorting Algorithms

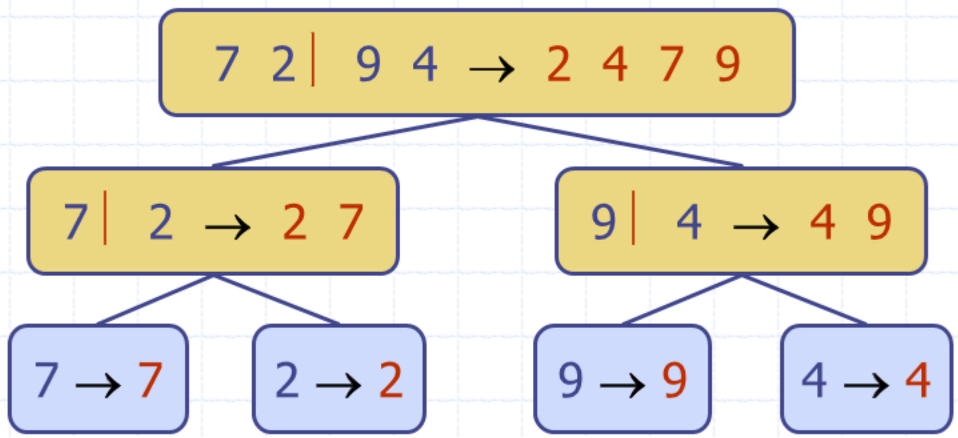
Algorithm	Time	Notes
selection-sort	$O(n^2)$	◆ in-place ◆ slow (good for small inputs)
insertion-sort	$O(n^2)$	◆ in-place ◆ slow (good for small inputs)
quick-sort	$O(n \log n)$ expected	◆ in-place, randomized ◆ fastest (good for large inputs)
heap-sort	$O(n \log n)$	◆ in-place ◆ fast (good for large inputs)
merge-sort	$O(n \log n)$	◆ sequential data access ◆ fast (good for huge inputs)

Java Implementation

only works for distinct elements

```
public static void quickSort (Object[] S, Comparator c) {
  if (S.length < 2) return; // the array is already sorted in this case
  quickSortStep(S, c, 0, S.length-1); // recursive sort method
}
private static void quickSortStep (Object[] S, Comparator c,
                                   int leftBound, int rightBound ) {
  if (leftBound >= rightBound) return; // the indices have crossed
  Object temp;  // temp object used for swapping
  Object pivot = S[rightBound];
  int leftIndex = leftBound;      // will scan rightward
  int rightIndex = rightBound-1; // will scan leftward
  while (leftIndex <= rightIndex) { // scan right until larger than the pivot
    while ( (leftIndex <= rightIndex) && (c.compare(S[leftIndex], pivot)<=0) )
      leftIndex++;
    // scan leftward to find an element smaller than the pivot
    while ( (rightIndex >= leftIndex) && (c.compare(S[rightIndex], pivot)>=0))
      rightIndex--;
    if (leftIndex < rightIndex) { // both elements were found
      temp = S[rightIndex];
      S[rightIndex] = S[leftIndex]; // swap these elements
      S[leftIndex] = temp;
    }
  } // the loop continues until the indices cross
  temp = S[rightBound]; // swap pivot with the element at leftIndex
  S[rightBound] = S[leftIndex];
  S[leftIndex] = temp; // the pivot is now at leftIndex, so recurse
  quickSortStep(S, c, leftBound, leftIndex-1);
  quickSortStep(S, c, leftIndex+1, rightBound);
}
```

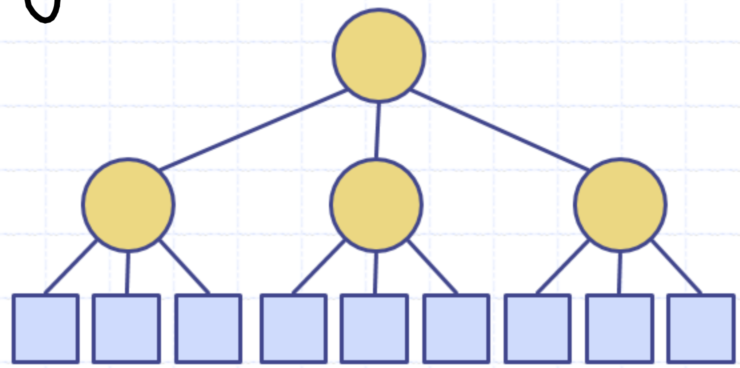
Divide-and-Conquer



Divide-and-Conquer

- Divide-and conquer is a general algorithm design paradigm:
  - Divide: divide the input data S in two or more disjoint subsets S_1, S_2, …
  - Recur: solve the subproblems recursively
  - Conquer: combine the solutions for S_1, S_2, …, into a solution for S
- The base case for the recursion are subproblems of constant size
- Analysis can be done using **recurrence equations**

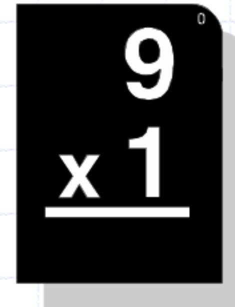
Quicksort 4 mergesort used only 2 disjoint subsets



Multiway Trees

eg: Red black 4 splay

Integer Multiplication



- Algorithm: Multiply two n-bit integers I and J.
  - Divide step: Split I and J into high-order and low-order bits

$$\left.\begin{aligned} I &= I_h 2^{n/2} + I_l \end{aligned}\right\} \; I = I_h \, I_l$$

$$\left.\begin{aligned} J &= J_h 2^{n/2} + J_l \end{aligned}\right\} \; J = J_h \, J_l$$

shift by n/2 bits

Each has length n/2

  - We can then define I*J by multiplying the parts and adding:

$$I * J = (I_h 2^{n/2} + I_l) * (J_h 2^{n/2} + J_l)$$

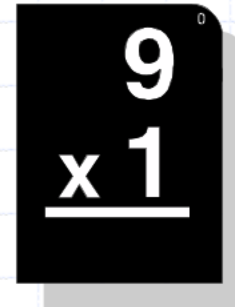
$$= I_h J_h 2^n + I_h J_l 2^{n/2} + I_l J_h 2^{n/2} + I_l J_l$$

  - So, $T(n) = \underbrace{4T(n/2)}_{\text{4 subproblems of size } n/2 \text{ each}} + n$

$5000 = 5 \times 10^3$

$I_h * 2^n$: $I_h << n$
in C/C++/Java

An Improved Integer Multiplication Algorithm



- Algorithm: Multiply two n-bit integers I and J.
  - Divide step: Split I and J into high-order and low-order bits

$$I = I_h 2^{n/2} + I_l$$

$$J = J_h 2^{n/2} + J_l$$

  - Observe that there is a different way to multiply parts:

$$I * J = I_h J_h 2^n + [(I_h - I_l)(J_l - J_h) + I_h J_h + I_l J_l] 2^{n/2} + I_l J_l$$

$$= I_h J_h 2^n + [(I_h J_l - I_l J_l - I_h J_h + I_l J_h) + I_h J_h + I_l J_l] 2^{n/2} + I_l J_l$$

$$= I_h J_h 2^n + (I_h J_l + I_l J_h) 2^{n/2} + I_l J_l$$

  - So, T(n) = 3T(n/2) + n

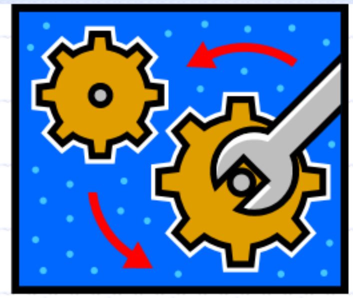
Recurrence Equation Analysis FOR MERGE-SORT

- The conquer step of merge-sort consists of merging two sorted sequences, each with $n/2$ elements and implemented by means of a doubly linked list, takes at most bn steps, for some constant b.
- Likewise, the basis case ($n < 2$) will take at b most steps.
- Therefore, if we let $T(n)$ denote the running time of merge-sort:

$$T(n) = \begin{cases} b & \text{if } n < 2 \\ 2T(n/2) + bn & \text{if } n \geq 2 \end{cases}$$

- We can therefore analyze the running time of merge-sort by finding a **closed form solution** to the above equation.
  - That is, a solution that has $T(n)$ only on the left-hand side.

Iterative Substitution



- In the iterative substitution, or "plug-and-chug," technique, we iteratively apply the recurrence equation to itself and see if we can find a pattern:

$$
\begin{aligned}
T(n) &= 2T(n/2)+bn \\
&= 2(2T(n/2^2))+b(n/2))+bn \\
&= 2^2T(n/2^2)+2bn \\
&= 2^3T(n/2^3)+3bn \\
&= 2^4T(n/2^4)+4bn \\
&= \ldots \\
&= 2^iT(n/2^i)+ibn
\end{aligned}
$$

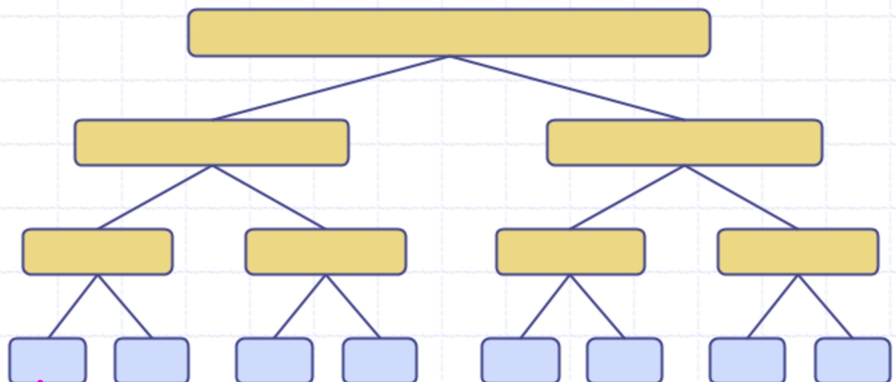
- Note that base, T(n)=b, case occurs when $2^i=n$. That is, i = log n.
- So, $T(n) = bn + bn\log n$ dominates
- Thus, T(n) is O(n log n).

The Recursion Tree

- Draw the recursion tree for the recurrence relation and look for a pattern:

$$T(n) = \begin{cases} b & \text{if } n < 2 \\ 2T(n/2) + bn & \text{if } n \geq 2 \end{cases}$$

depth	T's	size	time
0	1	n	bn
1	2	$n/2$	bn
i	2^i	$n/2^i$	bn
…	…	…	…

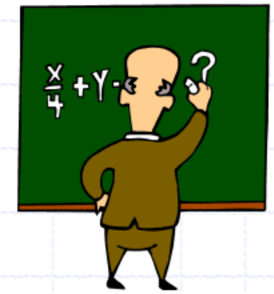


b for each leaf node

Total time = $bn + bn \log n$

(last level plus all previous levels)

Guess-and-Test Method



- In the guess-and-test method, we guess a closed form solution and then try to prove it is true by induction:

$$T(n) = \begin{cases} b & \text{if } n < 2 \\ 2T(n/2) + bn\log n & \text{if } n \geq 2 \end{cases}$$

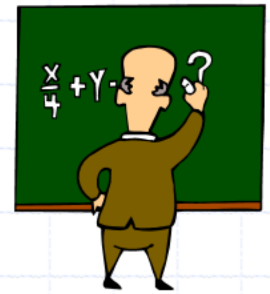
Note that this recurrence is different from that of mergesort

- Guess: T(n) < cn log n.

$$\begin{aligned} T(n) &= 2T(n/2) + bn\log n \\ &< 2(c(n/2)\log(n/2)) + bn\log n \\ &= cn(\log n - \log 2) + bn\log n \\ &= cn\log n - cn + bn\log n \end{aligned}$$

- Wrong: we cannot make this last line be less than cn log n

Guess-and-Test Method, Part 2



- Recall the recurrence equation:

$$T(n) = \begin{cases} b & \text{if } n < 2 \\ 2T(n/2) + bn\log n & \text{if } n \ge 2 \end{cases}$$

- Guess #2: $T(n) < cn \log^2 n$.

$$\begin{aligned} T(n) &= 2T(n/2) + bn\log n \\ &= 2(c(n/2)\log^2(n/2)) + bn\log n \\ &= cn(\log n - \log 2)^2 + bn\log n \\ &= cn\log^2 n - \underbrace{2cn\log n + cn + bn\log n}_{<0 \text{ for any } c>b} \\ &\le cn\log^2 n \end{aligned}$$

  - if c > b.
- So, $T(n)$ is $O(n \log^2 n)$.
- In general, to use this method, you need to have a good guess and you need to be good at induction proofs.

Even $T(n) < n^4$ would have worked!