Week 11 - Efficiency

Updated November 20, 3:30

Last week, we studied two famous recursive sorting algorithms, mergesort and quicksort. We said that these were much faster than the iterative sorting algorithms you've seen before like selection sort and insertion sort, but didn't explain why. This is the main topic of this lecture!

Efficiency of Recursive Algorithms

Iterative algorithms are typically quite straightforward to analyse, as long as you can determine precisely how many iterations each loop will run. Recursive code can be trickier, because not only do you need to analyse the running time of the non-recursive part of the code, you must also factor in the cost of each of the recursive calls you make.

You'll learn how to do this formally in CSC236, but for this course we're going to see a nice visual intuition for analysing simple recursive functions.

Mergesort

Recall the mergesort algorithm:

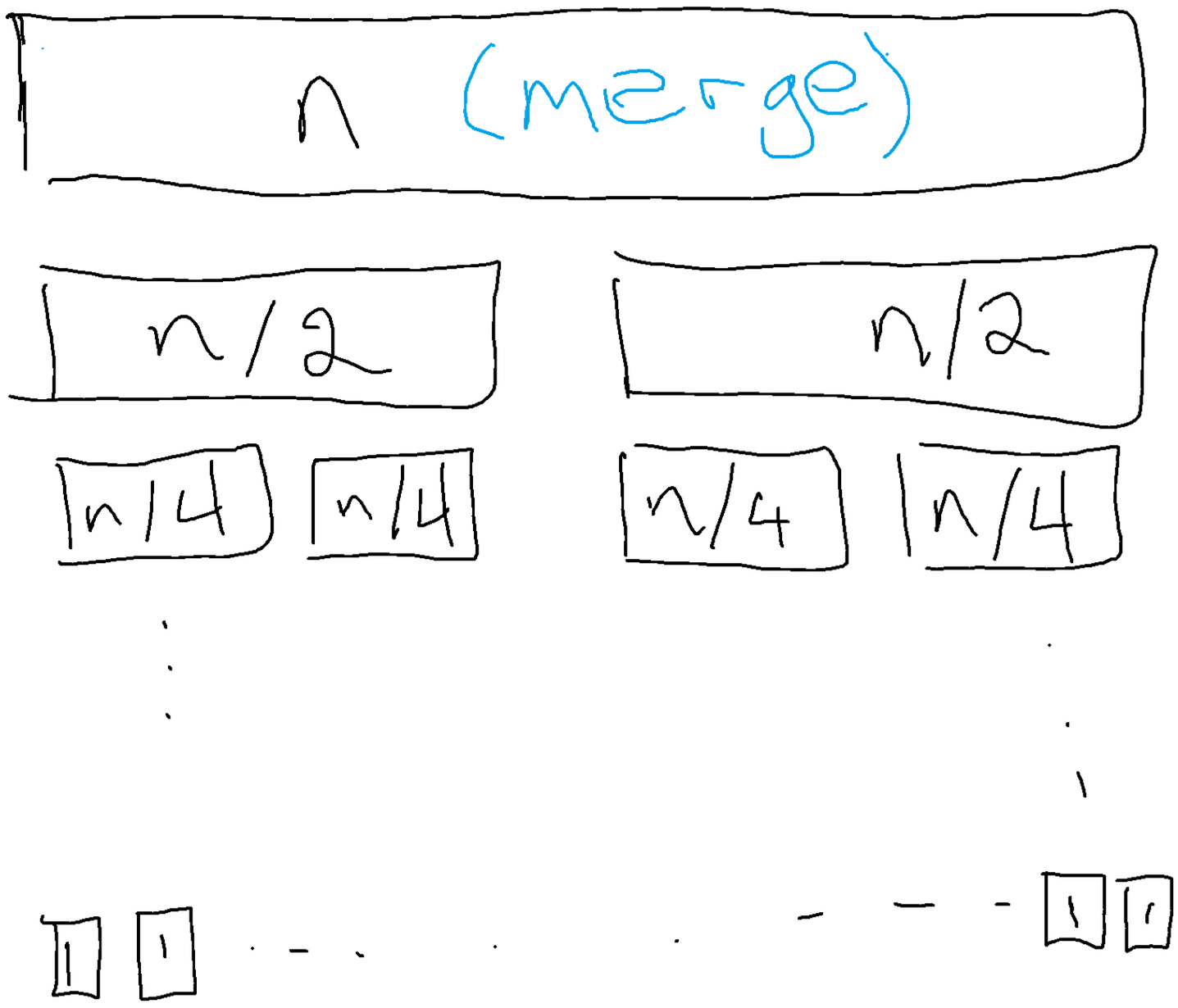
```
def mergesort(lst):
    """ (list) -> list
    Return a sorted list with the same elements as lst.
    Do *not* mutate lst.
    """
    if len(lst) <= 1:
        # return a copy of lst
        return lst[:]
    else:
        m = len(lst) // 2 # This is the midpoint of lst
        left_sorted = mergesort(lst[:m])
        right_sorted = mergesort(lst[m:])
        # "merge" the two sorted halves (how?)
        return merge(left_sorted, right_sorted)
```

Suppose we call `mergesort` on a list of length *n*. The main operations are the recursive calls to the left and right halves (each of length *n*/2), and the `merge` operation. The `merge` operation takes linear time, that is, approximately *n* steps (why?), but what about the recursive calls?

Well, since we split the list in half, each new list has length *n*/2, and each of those merge operations will take approximately *n*/2 steps, and then these two recursive calls will make four recursive calls, each on a list of length *n*/4, etc.

We can represent the total cost as a big tree, where at the top level we write the cost of the merge operation for the original recursive call, at the second level are the two recursive

calls on lists of size $n/2$, and so on until we reach the base case (lists of length 1).



Note that even though it looks like the tree shows the *size* of the lists at each recursive call, what it's actually showing is the *running time of the non-recursive part of each recursive call*, which just happens to be (approximately) equal to the size of the list!

The height of this tree is the **recursion depth**: the number of recursive calls that are made before the base case is reached. Since for mergesort we start with a list of length n and divide the length by 2 until we reach a list of length 1, the recursion depth of mergesort is the number of times you need to divide n by 2 to get 1. Put another way, it's the number k such that $2^k \approx n$. Remember logarithms? This is precisely the definition: $k \approx \log_2 n$, and so there are approximately $log_2 n$ levels.

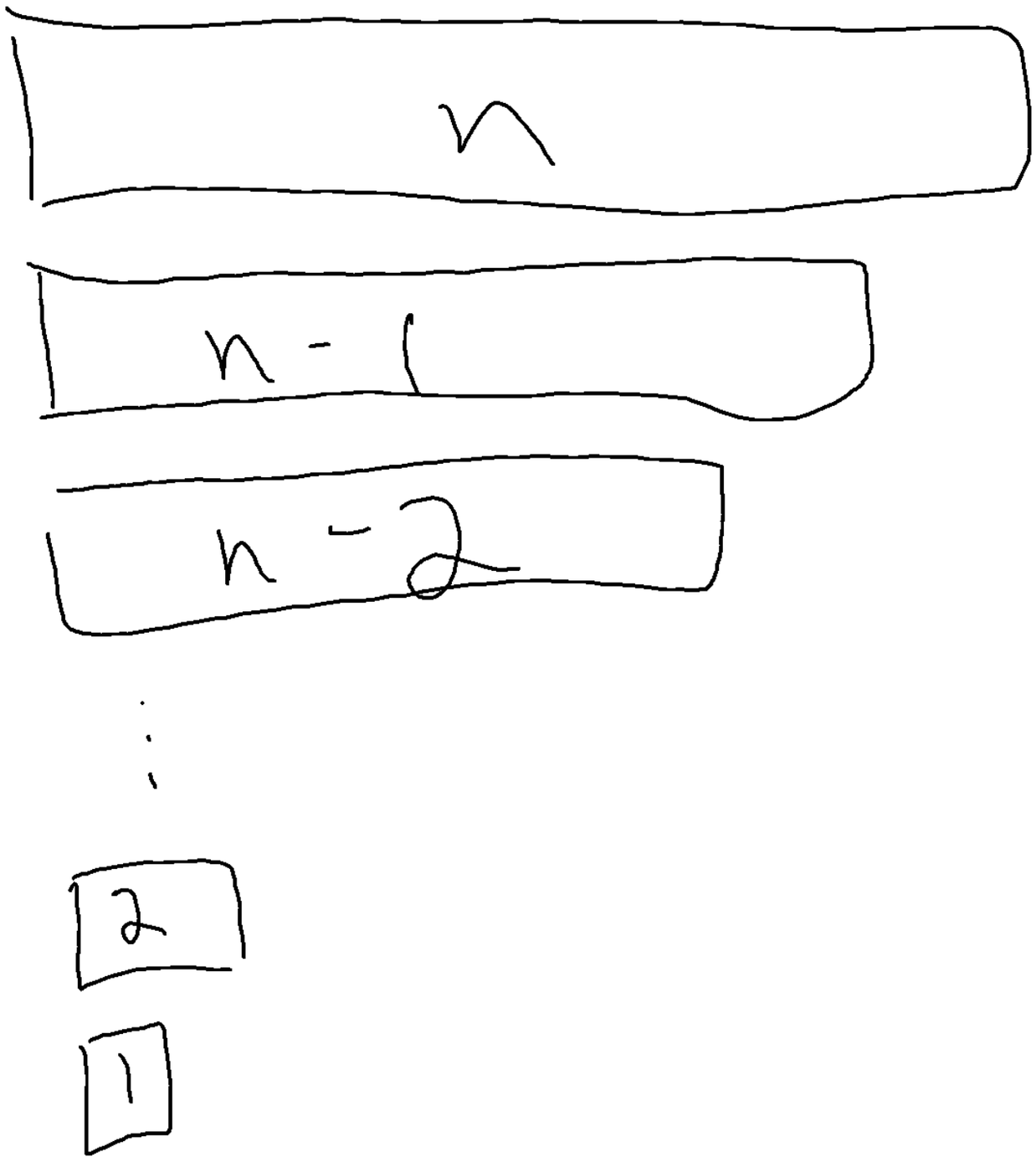
Finally, notice that at each level, the total cost is n. This makes the total cost of mergesort $n\log_2 n$, which is *much* better than the quadratic n^2 runtime of insertion and selection sort when n gets very large!

The Perils of Quicksort

What about quicksort? Is it possible to do the same analysis? Not quite. The key difference is that with mergesort we know that we're splitting up the list into two equal halves (each of size $n/2$); this isn't necessarily the case with quicksort!

Suppose we get lucky, and at each recursive call we choose the pivot to be the median of the list, so that the two partitions both have size $n/2$. Then problem solved, the analysis is the same as mergesort, and we get the $n\log n$ runtime. (Something to think about: what do we need to know about `partition` for the analysis to be the same? Why is this true?)

But what if we're always extremely unlucky with the choice of pivot: say, we always choose the smallest element? Then the partitions are as uneven as possible, with one having no elements, and the other having size $n - 1$. We get the following tree:



Here, the recursion depth is n (the size decreases by 1 at each recursive call), so adding the cost of each level gives us the expression $n + (n - 1) + (n - 2) + ... + 1 = n(n + 1)/2$, making the runtime be quadratic!

This means that for quicksort, the choice of pivot is *extremely* important, because if we repeatedly choose bad pivots, the runtime gets much worse!

Note about practicality

You might wonder if quicksort is truly used in practice, if its worst-case performance is so much worse than mergesort's. Keep in mind that we've swept a lot of details under the rug by saying that both `merge` and `partition` take n steps; in practice, the non-recursive parts of quicksort can be significantly faster than the non-recursive parts of mergesort. This means that for an "average" input, quicksort actually does better than mergesort, even though its worst-case performance is so poor!

In fact, with some more background in probability theory, we can even talk about the performance of quicksort on a *random* list of length n; it turns out that the average performance is essentially $n\log n$, indicating that the actual "bad" inputs for quicksort are quite rare.

Computer Science
UNIVERSITY OF TORONTO

David Liu (liudavid at cs dot toronto dot edu)
Come find me in BA4260