DS 320 Algorithms for Data Science **Divide and Conquer**
Spring 2024 Prof. Kira Goldner

Main Steps

There are four main steps for a divide and conquer solution.

Step 1: Define your recursive sub-problem. Describe in English what your sub-problem means, what it's parameters are, and anything else necessary to understand it.

Step 2: Define your base cases. Your recursive algorithm has base cases, and you should state what they are.

Step 3: Present your recursive cases. Give a mathematical definition of your sub-problem in terms of "smaller" sub-problems. Make sure your recursive call uses the same number and type of parameters as in your original definition.

Step 4: Prove correctness. This will be an inductive proof that your algorithm does what it is supposed to. You should start by arguing that your base cases return the correct result, then for the inductive step, argue why your recursive cases combine to solve the overall problem.

Step 5: Prove running time. This is especially important for divide and conquer solutions, as there is usually an efficient brute-force solution, and the point of the question is to find something more efficient than brute-force.

Example: Mergesort

We define a recursive algorithm that, given a list A of elements as well as left and right indices `lo` and `hi`, returns the elements `A`[`lo`], …, `A`[`hi`] in non-decreasing sorted order.

```
mergesort(Elements[] A, lo, hi )
  if lo = hi, i.e. |A| = 1  then
    return the list of one element, i.e. A[lo]
  Find midpoint of current list of elements, i.e. mid = ⌊(lo + hi)/2⌋
  Recursively run algorithm on left half, i.e. L = mergesort(A, lo, mid)
  Recursively run algorithm on right half, i.e. R = mergesort(A, mid + 1, hi)
  Merge L and R into a single list S in linear time:
  while both L and R are non-empty do
    let frontL and frontR denote the front elements of L and R, respectively.
    if first element of L is smaller than first element of R (i.e. frontL ≤ frontR) then
      append frontL to S and remove it from L
    else
      append frontR to S and remove it from R
  if one of L or R is non-empty then
    append remaining list onto S
  return S
```

Running Time. *Let $T(n)$ denote the running time of* `mergesort(A, lo, hi)` *where* $n = \texttt{hi} - \texttt{lo} + 1$. *Then since we make 2 recursive calls of half the size and merge in linear time we have*

$$T(n) = 2T(n/2) + O(n), \qquad\qquad T(1) = O(1)$$

And therefore the running time is $O(n \log n)$.

Claim. `mergesort(L, lo, hi)` *correctly sorts* `A`[`lo` ⋯ `hi`] *in non-decreasing order.*

Proof. For a list `A`[`lo` ⋯ `hi`], let P(`A`[`lo` ⋯ `hi`]) be the statement that `mergsort(A, lo, hi)` correctly sorts `A`[`lo` ⋯ `hi`] into non-decreasing order. We will prove P(`A`[`lo` ⋯ `hi`]) is true for any list `A`[`lo` ⋯ `hi`] by strong induction on $|A| = \texttt{hi} - \texttt{lo} + 1$.

As a base case, consider when $|A| = 1$, i.e. when `hi` = `lo`. This one-element list is already sorted, and our algorithm correctly returns `A[lo]` as the sorted list.

For the induction hypothesis, suppose that `P`(`A`[`lo` ⋯ `hi`]) is true for *all* lists of length $< n$; that is, for any list `A` and $\texttt{hi} - \texttt{lo} + 1 < n$, `mergesort(A, lo, hi)` correctly sorts `A`[`lo` ⋯ `hi`].

Now consider a list A of length n. Our algorithm divides A into two halves of size $< n$; therefore, L and R are in sorted order by our induction hypothesis. The minimum element of A is therefore either the minimum element of L (which is at the front of L) or the minimum element of R (which is at the front of R). We correctly take whichever is smallest as the minimum of A. We do this repeatedly, always selecting the next minimum element from the front of L or R, until we've produced the sorted A.

Thus we have by induction that `mergesort(A, lo, hi)` correctly sorts any list `A`[`lo` ⋯ `hi`]. □