DS 320 Algorithms for Data Science **Lecture #10 Worksheet**
Spring 2024 Prof. Kira Goldner

Divide & Conquer I: MergeSort

The Problem

As with any sorting problem, our goal is to take in a list and return it sorted in non-decreasing order.

The main idea here is to break the list into two halves, and sort both halves. They we combine the two sorted halves together by continuing to take the smallest element. Each half is sorted recursively.

This is the general idea with all Divide & Conquer solutions. Typically, the "naive" algorithm is already a polynomial solution (like $\Theta(n^2)$ sorting algorithms). By identifying clever **subproblems** that we can divide our problem into, **solve recursively**, and **combine** our solutions together, we can give a *more efficient* algorithm.

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.

Algorithm 1 mergesort(A, lo, hi).

Input: Array of elements A, left and right indices lo and hi.
if `lo` = `hi` **then** // Base Case: $|A| = 1$
  return `A[lo]`
end if
`mid` = $\lfloor$(`lo` + `hi`)/2$\rfloor$
`L` = `mergesort`(`A`, `lo`, `mid`) // Subproblem: Recursive calls on halves
`R` = `mergesort`(`A`, `mid` + 1, `hi`)
while both `L` and `R` are non-empty **do** // Merge `L` and `R` into a single list `S` in $\Theta(n)$
  `frontL` = $L[1]$ and `frontR` = $R[1]$
  if `frontL` $\leq$ `frontR` **then**
    append `frontL` to `S` and remove it from `L`
  else
    append `frontR` to `S` and remove it from `R`
  end if
end while
if one of `L` or `R` is non-empty **then**
  append remaining list onto `S`
end if
return S

Divide & Conquer Outline

There are four main steps for a divide and conquer solution. Try to fill out each of these for the above MergeSort problem and algorithm to solve it.

Step 1: Define your recursive subproblem. Describe in English what your subproblem means, what its 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 subproblem in terms of "smaller" subproblems. 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.

Prove the following claim via induction:

Claim 1. `mergesort(L, lo, hi)` correctly sorts $\texttt{A}[\texttt{lo} \cdots \texttt{hi}]$ in non-decreasing order.

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.

Your running time should take into account: the number of recursive calls we make, the size of the recursive calls, how long it takes to merge, and the running time of the base case. See the next section for more details.

Solving Recurrences

Running Time Intuition: Trees

We can draw out trees of the recursive calls made by our algorithm and the work done at each stage to get intuition for the running time of our algorithm.

We write our runtime recurrences as

$$T(n) = a\,T(n/b) + f(n)$$

where a is the number of subproblems that we make a call to, n/b is the size of our subproblem, and $f(n)$ is the running time to divide and combine our subproblems.

MergeSort solves $a = 2$ subproblems. Each subproblem is of size $n/2$ (so $b = 2$). The running time to divide our solution is just the running time to compute `mid`, which is constant. The running time to combine requires looping through the first element of both subproblem solution, so is $\Theta(n) = f(n)$. Our base case is of size 1 and is constant runtime, $T(1) = O(1)$. Hence our recurrence for MergeSort is:

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

We construct recursion trees in order to understand the sum of the runtime from these recurrences. In constructing a recursion tree for the recurrence $T(n) = 2\,T(n/2) + O(n)$: Part (a) shows $T(n)$, which progressively expands in (b)–(d) to form the recursion tree. First we draw the two (a) recursive calls to subproblems of size $n/2$ that will run in time $T(n/2)$ (or $T(n/b)$). We then expand from $T(n/2)$ and its recursive calls, all the way down to the base cases. The fully expanded tree in part (d) has $\log_2(n) + 1$ levels (i.e., it has height $\log_2 n$, as indicated—this will depend on the parameter b), and we can see that each level contributes a total cost of cn. The total cost, therefore, is $cn \log_2 n + cn$, which is $\Theta(n \log_2 n)$.

Proving Runtime Recurrences with Substitution

How do you prove this? Like induction! Your base case here would be $k = 1$, where $T(1) = 1$ by our algorithm's code. Our "inductive hypothesis" here will be for $k = n/2$, that the running time is $n/2 \log_2 n/2$. (It's very important to be careful about boundary conditions when handling base cases.)

Then substituting into our recurrence we get:

$$T(n) \leq$$

The "Master Theorem" for Divide & Conquer Runtime

Recall that a is the number of subproblems, n/b is the size of our subproblem, and $f(n)$ is the running time to divide and combine our subproblems.

The master theorem tells us the running time for most recurrences as a function of the parameters a, b, and $f(n)$ mentioned above, although it does not actually cover every case, as there are polynomial gaps between the cases.

Theorem 1 (The Master Theorem)**.** *Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the non-negative integers by the recurrence*

$$T(n) = a\,T(n/b) + f(n),$$

where we interpret n/b as $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then

1. *If $f(n) = O(n^{\log_b a - \varepsilon})$ for a constant $\varepsilon > 0$, then*
$$T(n) = \Theta(n^{\log_b a}).$$
2. *If $f(n) = \Theta(n^{\log_b a})$, then*
$$T(n) = \Theta(n^{\log_b a} \log_2 n).$$
3. *If $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for a constant $\varepsilon > 0$ and $af(n/b) \leq cf(n)$ for a constant $c < 1$ and for all sufficiently large n, then*
$$T(n) = \Theta(f(n)).$$

Solve the following recurrences using the master theorem.

- $T(n) = 9T(n/3) + n.$
- $T(n) = T(2n/3) + 1.$
- $T(n) = 3T(n/4) + n \log n.$