DS 320 Algorithms for Data Science **Divide and Conquer**
Spring 2025 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 $\texttt{A}[\texttt{lo}], \ldots, \texttt{A}[\texttt{hi}]$ 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. $\texttt{mid} = \lfloor(\texttt{lo}+\texttt{hi})/2\rfloor$
  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 $\leq$ 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 $S$ =0
```

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

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

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

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

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

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

For the induction hypothesis, suppose that $\texttt{P}(\texttt{A}[\texttt{lo}\cdots\texttt{hi}])$ 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 $\texttt{A}[\texttt{lo}\cdots\texttt{hi}]$.

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 $\texttt{A}[\texttt{lo}\cdots\texttt{hi}]$. □

Solving Divide & Conquer Recurrences

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.

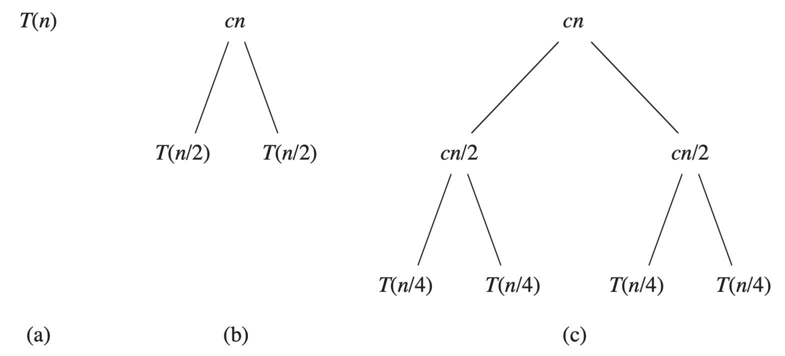
Example: MergeSort solves $a = 2$ subproblems. Each subproblem is of size $n/2$ (so $b = 2$). The running time to divide our solution is just the running time to compute `mid`, which is constant. The running time to combine requires looping through the first element of both subproblem solution, so is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

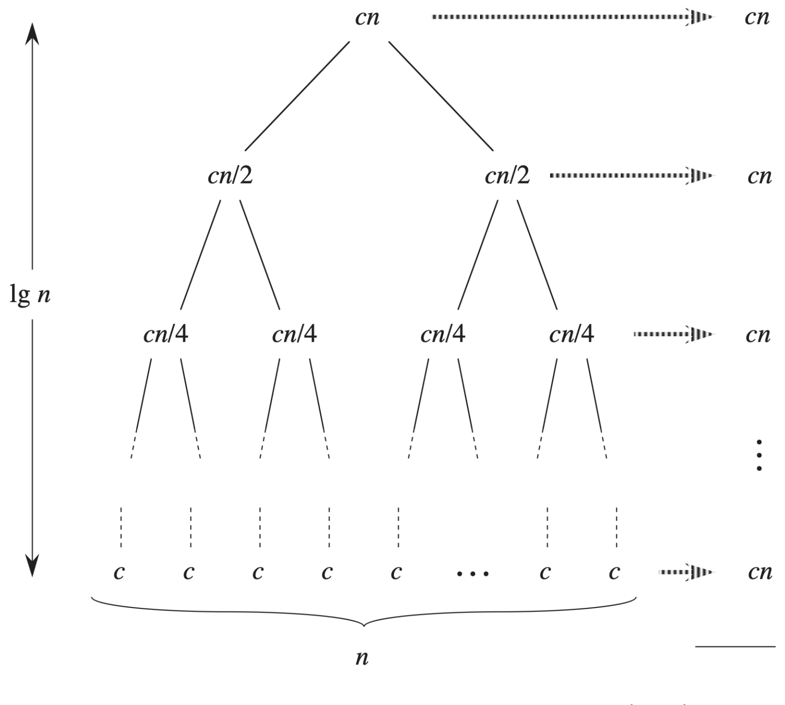
$$T(n) = 2T(n/2) + O(n), \qquad\qquad T(1) = O(1).$$

Recursion 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. **See the last section of this guide for the general recurrence and tree.**

Example: 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 $\Theta(n\log_2 n)$.





(d)

Aside: Proving Runtime Recurrences with Substitution

We will not require this in class, but it is good to know how you might prove this.

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:

$$\begin{aligned} T(n) &\leq 2\left(c\lceil\frac{n}{2}\rceil \log_2 \lceil\frac{n}{2}\rceil\right) + n \\ &= cn\log_2 n - cn\log_2 2 + n \\ &= cn\log_2 n - cn + n \\ &\leq cn\log_2 n \\ &= O(n\log n). \end{aligned}$$

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.** Thus it is **essential** to understand recurrences in an alternative way, such as via trees.

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*
$$T(n) = \Theta(f(n)).$$

Here is the master theorem with the cases slightly reworded.

Theorem 2 (The Master Theorem (reworded))**.** *The solution to the recurrence relation*

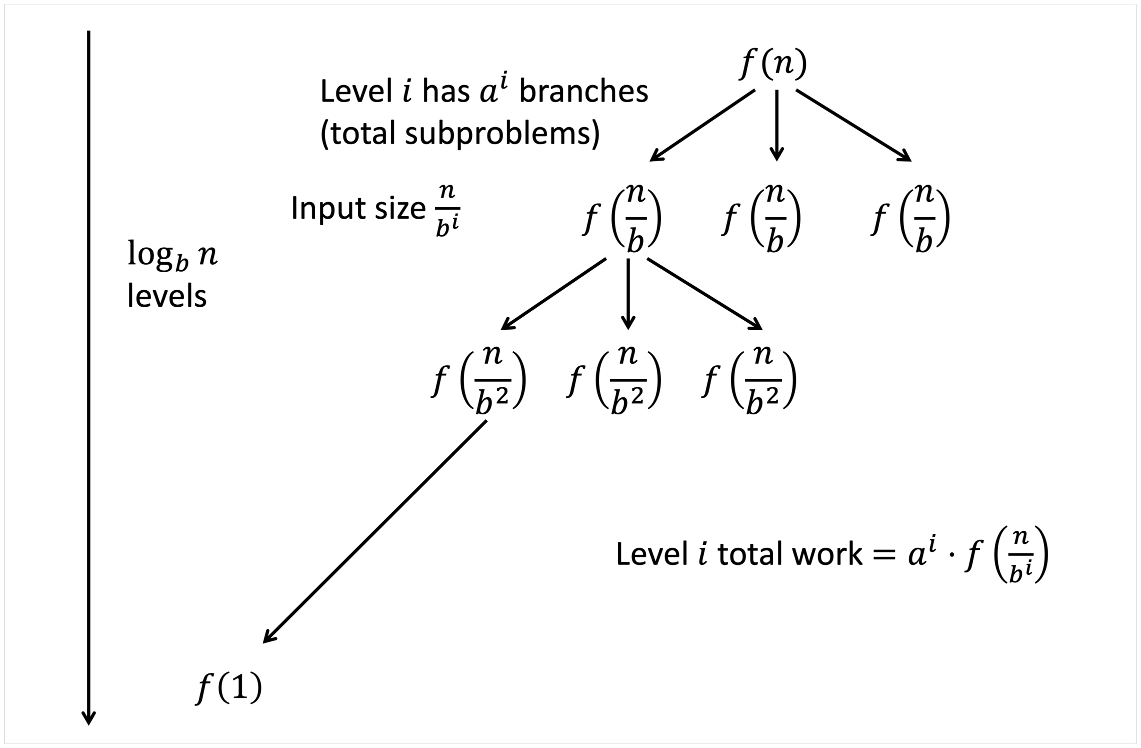
$$T(n) = a\,T(n/b) + cn^k,$$

where $a \geq 1$ and $b \geq 2$ are integers and c and k are positive constants satisfies:

$$T(n) = \begin{cases} O(n) & \text{if } a > b^k \\ O(n^k \log n) & \text{if } a = b^k \\ O(n^k) & \text{if } a < b^k. \end{cases}$$

Understanding The Master Theorem via Recurrence Trees

Below is the recursion tree for the general recurrence $T(n) = aT(n/b) + f(n)$ with a base case of 1. We will use it to try to generally calculate the running time, and thus the three cases of the Master Theorem.


Level i has a^i branches (total subproblems)
$f(n)$
Input size $\frac{n}{b^i}$
$f\left(\frac{n}{b}\right)$
$f\left(\frac{n}{b}\right)$
$f\left(\frac{n}{b}\right)$
$\log_b n$ levels
$f\left(\frac{n}{b^2}\right)$
$f\left(\frac{n}{b^2}\right)$
$f\left(\frac{n}{b^2}\right)$
Level i total work = $a^i \cdot f\left(\frac{n}{b^i}\right)$
$f(1)$

The levels of the tree range from $i = 0$ to $\log_b n$ with input size $\frac{n}{b^i}$ and number of subproblems per layer a^i. Then the runtime per layer is $a^i \cdot f\left(\frac{n}{b^{i-1}}\right)$, or in total,

$$\sum_{i=0}^{\log_b n} a^i \cdot f\left(\frac{n}{b^{i-1}}\right).$$

Notice that the cost of the leaves is just $a^{\log_b n} \cdot \Theta(1)$, and since $a^{\log_b n} = n^{\log_b a}$, we pull this term out separately:

$$T(n) = n^{\log_b a} + \sum_{i=0}^{\log_b n - 1} a^i \cdot f\left(\frac{n}{b^{i-1}}\right).$$

Then the master theorem is just discussing where the dominating terms are: the leaves (case 1), evenly distributed (case 2), or the root (case 3).