Objectives:

- finish discussion of recurrences
- learn about heapsort, an $O(n \lg n)$, in-place sorting algorithm
- introduction to the heap as a data structure

Some miscellaneous examples of recurrence solution

Insertion sort

As noted earlier, insertion sort can be formulated as a recursive algorithm. The recurrence relation for the worst-case number of comparisons is $T(n) = T(n - 1) + n - 1$ if $n > 1$ and $T(1) = 0$.

Using the tree method (we could also have used the substitution method), we see only one problem instance at each level. The table description is

<table>
<thead>
<tr>
<th>Level</th>
<th># of instances</th>
<th>instance size</th>
<th>cost per instance</th>
<th>total cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>$n$</td>
<td>$n - 1$</td>
<td>$n - 1$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>$n - 1$</td>
<td>$n - 2$</td>
<td>$n - 2$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$i$</td>
<td>1</td>
<td>$n - i$</td>
<td>$n - i - 1$</td>
<td>$n - i - 1$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$k$</td>
<td>1</td>
<td>$n - k = 1$</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

So we can conclude that $k = n - 1$ and $T(n) = \sum_{i=0}^{k-1} (n - i - 1) = \sum_{j=1}^{n-1} j = n(n - 1)/2$.

Change of variables

Lest we turn the tree method into a religion . . .

Another way to deal with the recurrence

$$T(n) = aT(n/b) + f(n) \text{ if } n > s \text{ and } T(n) = c_0 \text{ otherwise}$$

is to begin with the assumption that $n = b^k$, as before, but then express the recurrence in terms of $k$. We then have (letting $k = \log_b n$)

$$T(k) = aT(k - 1) + f(b^k) \text{ if } k > \log_b s \text{ and } T(k) = c_0 \text{ otherwise}$$

This is especially simple when $a = 1$. Even in other cases, it’s easy to verify that $T(k) = c_0a^k + \sum_{i=1}^{k} a^{k-i} f(b^i)$, the same result as before.

Not a big deal, really.

But suppose we have something like $T(n) = 2T(\lg n) + 1$ (let’s not worry about $T(n)$ for small $n$ yet, nor about the fact that we probably mean $[\lg n]$ or $\lceil \lg n \rceil$).

How many levels will we have? (We want $k$ to be the number of levels above the bottom level).

Looks like if we take the log often enough, we’ll eventually reach something less than or equal to 1, at which point we should stop (since $\lg 1 = 0$ and $\lg x < 0$ when $x < 1$). The number of times we
can take the log before that happens has a special name in theoretical computer science — \( \lg^* n \), the iterated logarithm. By definition (look it up in the index), if we let \( k = \lg^* n \), then \( k - 1 = \lg^*(\lg n) \).

So, under change of variables, the above recurrence becomes

\[
T(k) = 2T(k - 1) + 1 \text{ for } k > 0 \text{ and } c_0 \text{ otherwise}
\]

Now we have something simpler on our hands, solvable by brute force, guess and check, the tree method, whatever . . .

\[
T(k) = c_0 2^k + \sum_{j=0}^{k-1} 2^j = 2^k(c_0 + 1) - 1
\]

We assumed \( k = \lg^* n \) so \( 2^k = 2^{\lg^* n} \). Is this faster growing than \( \lg^* n \)? Yes — for the same reason that \( 2^k \) grows faster than \( k \).

Prove to yourself:

1. If \( f(n) \in O(g(n)) \) then \( \lg f(n) \in O(\lg g(n)) \).
2. If \( f(n) \in \Omega(g(n)) \) then \( 2^{f(n)} \in \Omega(2^{g(n)}) \).