Learning objectives:

- ☐ identify divide-and-conquer recurrence relations,
- ☐ apply the tree method to solve divide-and-conquer recurrence relations,
- ☐ analyze the complexity of merge-sort and binary search algorithms.

In the last lecture, we looked at recurrences of the form $f(n) = \sum_{i=1}^{d} a_i f(n-i)$. Today, we'll still consider linear recurrences, but instead of having a rate of 1 (in front of the n term), we'll consider recurrences that result from breaking the problem into smaller chunks upon every recursive function call. Let's motivate this by analyzing merge-sort.

Linear?

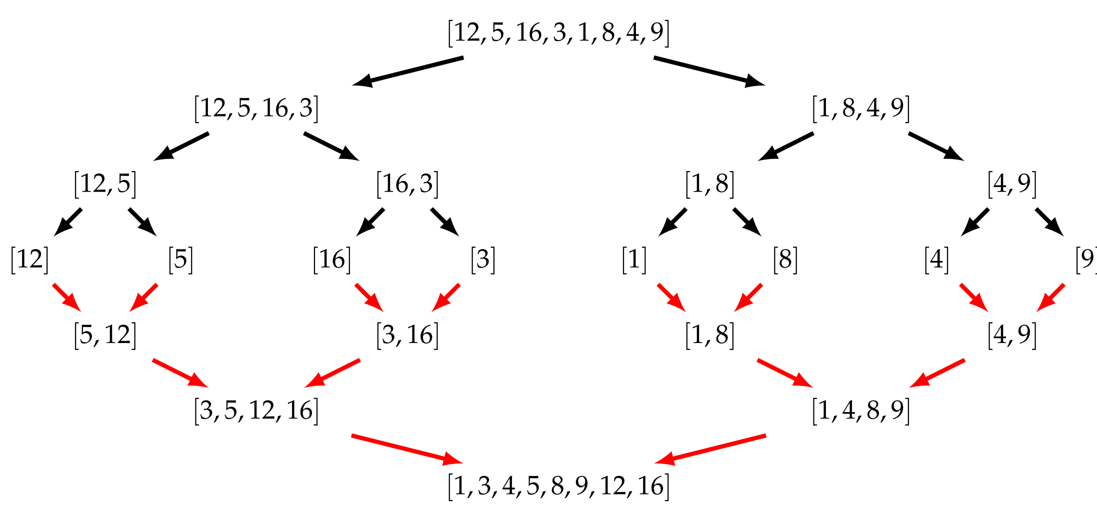
We called these *linear* because the $(n-i)$-part is linear in n.

Given an array of n items, merge sort consists of the following steps:

- If $n = 1$, then return the single item because this is automatically sorted.
- Otherwise, break up the array into two pieces, each of length $n/2$ and call merge-sort on each sub-array. Then, *merge* the two sorted sub-arrays.

Example 1:
Apply merge-sort to sort the array of integers $[12, 5, 16, 3, 1, 8, 4, 9]$.

Solution:
Let's visualize the sorting procedure with a graph. The black edges represent recursive calls to merge-sort and, equivalently, when the input arrays are broken into two sub-arrays. The red edges represent the merging procedure.



How many operations are performed in merge-sort? We can determine the number of operations by developing a recurrence relation. The number of operations during any particular call to merge-sort on an input array of length n requires $n-1$ operations to merge the two sub-arrays since we need to compare the leading (lowest) remaining values of each sub-array, comparing them with the current lowest value in the merged array, incurring at most $n-1$ comparisons - we don't need to do a comparison for the last remaining value. Note that no comparisons are needed when we have a single value ($n=1$). Since we need to call merge-sort on both sub-arrays, then the number of operations needed to sort an array of length n is

$$T(n) = 2T(n/2) + n - 1. \tag{1}$$

Why is the upper bound of the sum $\log n - 1$?

It takes $\log n$ times to break up the array of length n in half into subarrays of length 1 (assuming the base of the log is 2). Our sum starts at $i = 0$, so will get $\log n$ terms in the summation with an upper bound of $\log n - 1$.

Example 2:
Use the expand-and-pray method to verify the number of operations of merge-sort is $O(n \log n)$.

Solution:
We can write out the first few terms of the recurrence relation in Equation 1 and try to see a pattern:

$$\begin{aligned}
T(n) &= (n-1) + 2T\left(\frac{n}{2}\right) \\
&= (n-1) + 2\left(\left(\frac{n}{2} - 1\right) + 2T\left(\frac{n}{4}\right)\right) \\
&= (n-1) + (n-2) + 4T\left(\frac{n}{4}\right) \\
&= (n-1) + (n-2) + 4\left(\left(\frac{n}{4} - 1\right) + 2T\left(\frac{n}{8}\right)\right) \\
&= (n-1) + (n-2) + (n-4) + 8T\left(\frac{n}{8}\right) \\
&= (n-1) + (n-2) + (n-4) + \cdots + \left(n - 2^{i-1}\right) + 2^i T\left(\frac{n}{2^i}\right) \\
&= (n-1) + (n-2) + (n-4) + \cdots + \left(n - 2^{\log n - 1}\right) + 2^{\log n} \underbrace{T(1)}_{0} \\
&= \sum_{i=0}^{\log n - 1} \left(n - 2^i\right) \\
&= \sum_{i=0}^{\log n - 1} n - \sum_{i=0}^{\log n - 1} 2^i \\
&= n \log n - \left(2^{\log n} - 1\right) \\
&= n \log n - n + 1
\end{aligned}$$

The detailed number of operations is $T(n) = n \log n - n + 1$, which is $O(n \log n)$.

The expand-and-pray method worked fine, but it's a bit tricky when the recurrence relation is more complicated. Luckily, there is a more formal method for solving recurrences of this type. We call these types of recurrences *divide-and-conquer* recurrences, since we are "dividing" the problem into a few subproblems, "conquering" those subproblems, and then solving the current problem using the solution to the subproblems.

1 Tree method for divide-and-conquer recurrences

Consider a recurrence relation of the form

$$f(n) = a \cdot f\left(\frac{n}{b}\right) + c \cdot n^d \tag{2}$$

where $n = b^k$ for some $k \in \mathbb{Z}^+$, $a \geq 1$, $b > 1$, $b \in \mathbb{Z}$, $c > 0$, $d \geq 0$ and $a, c, d \in \mathbb{R}$. We can characterize the complexity of $f(n)$ for various cases:

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

Before we can apply the theorem, it's important to be able to say in words what all the terms in Equation 2 mean:

- a: number of subproblems created during the recursive step,
- b: factor by which problem shrinks in the recursive steps,
- c, d: characterizes extra work performed outside of recursive function call.

Master Theorem versus Tree Method?
In textbooks, this is often known as the Master Theorem for solving divide-and-conquer recurrence relations, which can be proved by induction. However, the use of the word "master" can have negative connotations, so we will call this method the *Tree Method*. In fact, there exists a more general method, proved by Akra & Bazzi in 1996.

In other words, divide-and-conquer algorithms divide a problem of size n into a subproblems, each of which has a size of n/b. Let's do a few examples to practice applying this method.

Example 3:
Use the Tree Method to determine the complexity of merge-sort.

Solution:
Matching coefficients in Equation 1 with Equation 2 gives $a = 2$, $b = 2$, $d = 1$. Since $a = b^d$, then $T(n) = O(n \log n)$.

Example 4:
Develop a recurrence relation for the binary search algorithm, described in Algorithm 1 and apply the Tree Method to determine the number of operations executed in binary search.

Solution:
On each entry to the function, the binary search algorithm makes a single recursive function call, depending on the value of the middle index m, x and a (either Lines 9 or 11 are executed, but never both). Each of these divides the problem into half. There are four operations: one to retrieve the length of the array, another to compute the middle index, a comparison with the requested value, and a final one to determine whether we need to search the lower/upper half of the array. We'll just use a constant c to represent these operations.

$$T(n) = T(n/2) + c.$$

Since only a single operation is performed in the $n = 0$ case, then $T(0) = 1$. Again, the actual value of the constant doesn't really matter (i.e. if you have 0, 1 or even 2 operations), as long as it is independent of n. Using the Tree method, we identify the constants as $a = 1$ $b = 2$ and $d = 0$, so $T(n) = O(\log n)$.

Four operations?

Depending on how you write the code, you might have a few more or a few less operations, so we'll just say that we take c (where c is some constant) operations on each recursive call, before dividing the problem and making subsequent recursive calls.

```
binary_search(a, x)

    input: sorted array a, value x
    output: boolean as to whether a contains the value x
1   n ← length(a)
2   if n == 0
3       return False
4   else
5       m ← n//2 # use integer division to get middle index
6       if a[m] == x
7           return True
8       else if a[m] < x
9           return binary_search(a[m : n], x)
10      else
11          return binary_search(a[0 : m], x)
```

Algorithm 1: Binary search algorithm to determine if an array a contains a value x (returns True or False)

.

2 Derivation of the Tree method

Let's derive the Tree method! You'll also see why it's called the Tree method. Recall we're working with divide-and-conquer recurrences of the form:

$$f(n) = a \cdot f\left(\frac{n}{b}\right) + c \cdot n^d$$

where $n = b^k$ for some $k \in \mathbb{Z}^+$, $a \geq 1$, $b > 1$, $b \in \mathbb{Z}$, $c > 0$, $d \geq 0$ and $a, c, d \in \mathbb{R}$. Recall that a refers to the number of subproblems created on each recursive step, b is the fraction by which the problem size decreases on each recursive function call, and n^d characterizes the amount of work done (in the recursive step) separate from the recursive function call.

Consider drawing a stack diagram for the recursive function calls. This will look like a tree, in which each internal vertex has a children. Let k represent the number of levels we have traversed down the tree.

Example 5:

(a) How many vertices are there at level k?

(b) What is the size of the problem at level k?

(c) How much work is done within a single vertex of the tree at level k?

(d) How much work is done at level k?

(e) At what level will the original array of length n have been broken up into arrays of length 1?

(f) How much total work is done?

(g) Simplify your expression for part (f) for the case when n gets really really big. Do this for three cases: (i) $a < b^d$, (ii) $a = b^d$ and (iii) $a > b^d$.

Solution:

(a) a^k

(b) n/b^k

(c) $(n/b^k)^d$

(d) $a^k(n/b^k)^d = n^d(a/b^d)^k$

(e) $\log_b n$

(f) Adding up the work done in every leaf gives:

$$\sum_{k=0}^{\log_b n} (a/b^d)^k n^d = n^d \sum_{k=0}^{\log_b n} (a/b^d)^k$$

(g) This looks like a geometric series with $r = a/b^d$. Recall our general formula for a geometric series (for $r \neq 1$), is

$$\sum_{k=0}^{N} r^k = \frac{1 - r^{N+1}}{1 - r}$$

Here, $N = \log_b n$. When $r < 1$ and n gets really really big, the summation is equal to $1/(1-r)$. Therefore, for $a < b^d$, we have $1/(1 - a/b^d) = b^d/1 - b^d$. But don't forget the n^d in front, which is actually the dominant term in our final expression, so for $a < b^d$, (i) $f(n) = O(n^d)$. When $r = 1$ (ii), then the summation is equal to $\log_b n + 1$, and remembering the n^d in front, we have $f(n) = O(n^d \log_b n)$. For the last case, $a > b^d$, we have

$$n^d \left(\frac{1 - \left(\frac{a}{b^d}\right)^{\log_b n}}{1 - a/b^d} \right) = \mathcal{O}\left(n^d \left(\frac{a}{b^d}\right)^{\log_b n} \right) = \mathcal{O}(n^{\log_b a}).$$

To prove that last step, recall that $x^y = e^{y \log x}$. Letting $x = (a/b^d)$, we then have $n^d e^{\log x \log n / \log b} = e^{(\log a - d \log b) \log n / \log b + d \log n} = e^{\log a \log n / \log b} = e^{\log n (\log_b a)} = n^{\log_b a}$.