Learning objectives:

- ☐ develop recurrence relations to analyze the performance of recursive algorithms,
- ☐ solve recurrence relations using the *guess and check* method,
- ☐ solve recurrence relations using the *expand and pray* metod.

The goal for this lecture is to use some simple methods to solve recurrence relations. But first, we need to understand why recurrence relations are important. Where and how do they even show up? Well, they show up in a lot of places. Remember merge-sort? In an introductory class, you probably got a pretty hand-wavy explanation as to why the run-time of merge-sort is $O(n \log n)$. Actually, we can analyze this algorithm by developing a recurrence relation, and then solving it. But recurrence relations also show up in real life, like population dynamics and other areas of mathematical modeling. Consider the following example.

Example 1:
The Tower of Hanoi is a famous mathematical game invented in 1883 that consists of displacing n disks (of different diameters) from one rod to another, using only three rods. The game starts out with all n disks ordered from largest to smallest on one rod, as shown in Figure 1 with $n = 3$. The objective of the game is to displace all disks to another rod, such that they exhibit the same ordering on the new rod. The catch, however, is that larger disks can *never* be stacked on top of a smaller one. Here are the rules:

(1) Only one disk can be moved at a time.

(2) Every move consists of displacing a disk from the top of one stack, to the top of another stack (or an empty rod).

(3) No larger disk can ever be placed on top of a smaller one.

You can also try playing the game here. How many moves does it take to displace the entire stack of n disks?

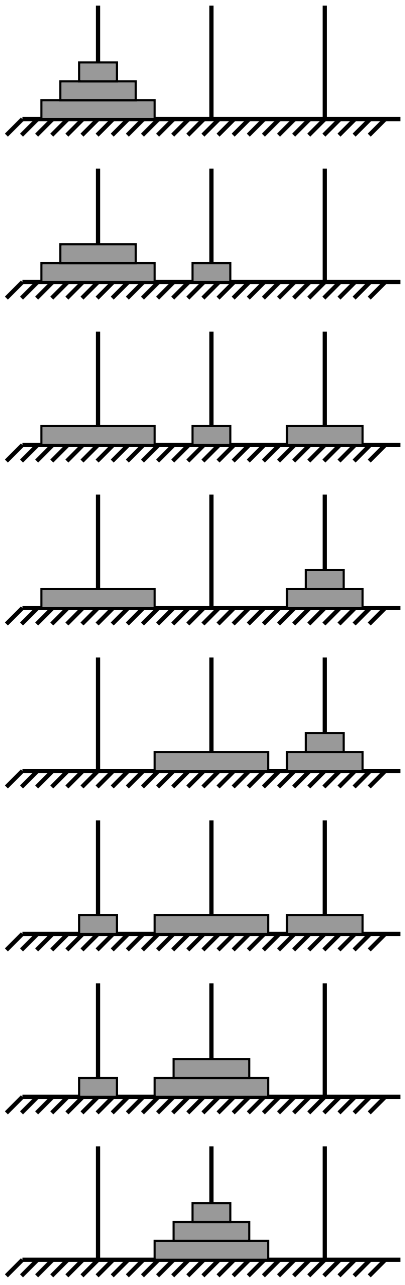


Figure 1: Solving the Towers of Hanoi puzzle (moves are ordered from top to bottom).

Let's now analyze the Towers of Hanoi puzzle. How many moves does it take to displace $n = 2$ disks? It takes three moves: one to displace the top (small) disk, another to displace the large one, and a third move to displace the small disk back onto the new position of the large disk. What about if $n = 3$? Well, we can extend our solution to the $n = 2$ case by just saying that it takes 3 moves to fully displace the first two disks (call this $T(2)$), then one to displace the bottom (largest) disk $(+1)$, and finally another 3 moves to displace the smallest two disks (which are now ordered) back onto the new position of the bottom disk (incurring another $T(2)$). Therefore,

$$T(3) = T(2) + T(2) + 1 = 2T(2) + 1.$$

The pattern stays the same as we increase n and we can write the minimum number of moves to displace n disks as

$$T(n) = 2T(n-1) + 1$$

Now we need to solve this! Ultimately, we want to solve for $T(n)$ such that there are no other $T(i)$'s on the right-hand-side. In general, you will also be given $T(\text{base})$, i.e. how many moves it takes if you have $n = 1$ disk?

Hmmm this sounds like recursion...

Yes! At the end of the lecture we will actually be able to analyze the performance of some recursive algorithms.

1 Method #1: guess and check

The *guess and check* method isn't all that fancy. It takes a little bit of intuition to guess a solution to the recurrence relation, so I don't really recommend this method. Once you guess a solution, you **must** verify it using a proof by induction.

Example 2:
Let's guess a solution to the recurrence relation for the Towers of Hanoi puzzle. Specifically, try $T(n) = 2^n - 1$.
Proof. We use a proof by induction on n. Let the induction hypothesis be the predicate $p(n)$: $T(n) = 2^n - 1$ *solves the recurrence relation* $T(n) = 2T(n-1) + 1$.

Base case: For $n = 1$, we have $T(1) = 2^1 - 1 = 1$, which verifies that we only need one move with a single disk.

Inductive step: Assume $p(n)$ is true. That is, it takes $2^n - 1$ moves to displace n disks. We need to show that it takes $T(n+1) = 2^{n+1} - 1$ moves to displace $n+1$ blocks. From the recurrence relation, we have

$$\begin{aligned} T(n+1) &= 2T(n+1-1) + 1 && \text{from recurrence relation,} \\ &= 2(2^n - 1) + 1 && \text{by } p(n), \\ &= 2^{n+1} - 2 + 1 && \text{manipulating,} \\ &= 2^{n+1} - 1 && \text{verifying } p(n+1). \end{aligned}$$

By induction on n, $p(n)$ is true. □

It's not always easy to guess a solution to a recurrence relation, so we need another method.

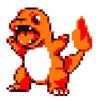
2 Method #2: expand and pray

The *expand and pray* method is a little fancier than the *guess and check* method, but still has its drawbacks. With this method, you keep substituting values for $T(n)$, $T(n-1)$, $T(n-2), \ldots$ until you notice a pattern. This trick is noticing the pattern, and then applying some knowledge about sequences and series to find a solution.

Example 3:
Expanding a few terms in the recurrence relation for the Towers of Hanoi puzzle gives

$$\begin{aligned}
T(n) &= 1 + 2T(n-1) \\
&= 1 + 2(1 + 2T(n-2)) \\
&= 1 + 2 + 4T(n-2) \\
&= 1 + 2 + 4(1 + 2T(n-3)) \\
&= 1 + 2 + 4 + 2^3 T(n-3) \\
&= \sum_{i=1}^{n} 2^{i-1} \\
&= \frac{1-2^n}{1-2} && \text{formula for a geometric series} \\
&= 2^n - 1.
\end{aligned}$$

Remember a geometric series!



Recall that the formula for the sum of a geomtric series is

$$\sum_{i=0}^{n} r^i = \frac{1-r^{n+1}}{1-r}.$$

3 Analyzing the complexity of recursive algorithms

Recurrence relations can be developed when analyzing the number of operations performed in recursive algorithms. In general, the recursive algorithms we will study (and that you will see in the future) will be in one of two forms.

3.1 Linear form

The first form is described in Algorithm 1 and will be referred to as the *linear* form. Here, the recursive case returns a linear combination of other recursive function calls, where the input variable linearly decreases upon each call.

The Towers of Hanoi is an example of this type of recursive algorithm. Can you think of another?

```
recursive($n$)

    input: $n$ (size of problem), some other data
    output: some output
1   if $n$ at base case
2       return $c_0$ (some value for the base case)
3   else
4       return $c_{n-1} \cdot$ recursive$(n-1) + c_{n-2} \cdot$ recursive$(n-2) + \dots$
                $+c_2 \cdot$ recursive$(2) + c_1 \cdot$ recursive$(1) + d$
```

Algorithm 1: General outline of a **linear** recursive algorithm with a base case and recursive function call. Note that $g(n)$ is some function of n. The goal is to analyze how many operations are performed in these types of algorithms.

Example 4:
Consider the pseudocode to compute Fibonacci numbers, listed in Algorithm 2. See if you can determine the coefficients c_i $0 \leq i \leq n-1$.

Solution:
Matching coefficients with the general form in Algorithm 1 gives $c_{n-1} = 1$ and $c_{n-2} = 1$. All other c_i's are zero except $c_0 = 1$.

```
fibonacci($n$)

    input: $n$ (integer))
    output: some output
1   if $n \leq 1$
2       return $0$
3   else
4       return fibonacci$(n-1)$ + fibonacci$(n-2)$
```

Algorithm 2: Recursive program to compute Fibonacci numbers. Compare this to Algorithm 1 and notice that $c_{n-1} = 1$ and $c_{n-2} = 1$. All other c_i's are zero except $c_0 = 1$. Also, $f(n) = 0$.

```
recursive($n$)

    input: $n$ (size of problem), some other data
    output: some output
1   if $n$ at base case
2       return $c_0$ (some value for the base case)
3   else
4       return $c_k \cdot$ recursive$(b_k n) + c_{k-1} \cdot$ recursive$(b_{k-1} n) + \ldots$
                $+ c_2 \cdot$ recursive$(b_2 n) + c_1 \cdot$ recursive$(b_1 n) + d$
```

Algorithm 3: General outline of a **divide-and-conquer** recursive algorithm with a base case and recursive function call. Note that $g(n)$ is some function of n. The goal is to analyze how many operations are performed in these types of algorithms.

3.2 Divide-and-conquer (DC) form

The other form we will study is described in Algorithm 3.

Merge-sort is an example of a divide-and-conquer algorithm. Remember that the general idea of merge-sort is to break an array into two arrays (each half the size of the original) and then sort those two sub-arrays by recursively breaking them into sub-arrays, sorting them recursively, etc., until we get to arrays of length 1, which can be trivially sorted. Once we've sorted the sub-arrays, we then merge them back together, which incurs $n-1$ comparisons (where n is the size of the full array). The number of operations required in merge-sort is then

$$\begin{aligned} T(n) &= \text{\# ops to merge} + \text{\# ops to sort two chunks (recursively)} \\ &= (n-1) + 2T(n/2) \qquad (1) \end{aligned}$$

So merge sort has $k = 1$, with $c_1 = 2$ and $b_1 = \frac{1}{2}$. There is an extra amount of work that is done, separate from the recursive function calls: the $(n-1)$ operations that are needed to merge the two sorted sub-arrays. We will explore the details of how to solve divide-and-conquer recurrence relations in a few lectures.

3.3 Combinations of both forms

You can certainly have algorithms that combine both the *linear* and *divide-and-conquer* forms, however, the methods to solve these is beyond the scope of this course. A more general method to solve these types of recurrences was developed by Akra & Bazzi in 1996.

Akra & Bazzi

In fact, Akra & Bazzi were students at the time they discovered a general method for solving recurrences, which is pretty amazing!