1 Recurrence Relations

We can express the running time of iterative code (for loops, while loops, do-while loops, ect) with summations, but how can we express the running time of recursive code?

We will use recurrence relations:

Definition 1.1 (Book Definition)**.** *A* ***recurrence relation*** *is an equation or inequality that defines a function in terms of its value on smaller inputs.*

Definition 1.2 (Formal Definition)**.** *A* ***recurrence relation*** *is an equation or inequality that has the form*

$$a_n = \psi(n, a_{n-1})$$

where $\psi : \mathbb{N} \times X \to X$ *is a function and* X *is a set such that* $\forall n \in \mathbb{N}, a_n \in X$*.*
A ***recurrence relation of order*** k *has the form*

$$a_n = \psi(n, a_{n-1}, a_{n-2}, \ldots, a_{n-k})$$

where $\psi : \mathbb{N} \times X^k \to X$ *is a function*

You already know some recurrence relations!
For example, the factorial function:

$$\begin{aligned} 0! &= 1 \\ n! &= n(n-1)! && \forall n \geq 1 \end{aligned}$$

Or the Fibonacci sequence:

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_n &= F_{n-1} + F_{n-2} && \forall n \geq 2 \end{aligned}$$

For any given recurrence relation, the natural question is if there is a closed form - a way of writing the expression using a fixed number of "standard" operations?
To find a closed form, there are several techniques, such as

- **Recursion trees**
- **Substitution method**
- Characteristic polynomials
- Generating functions

and there are several theorems (for example Master theorem).
For this class, we will focus on recursion trees and the substitution method (I'll demonstrate substitution method, I don't recommend it for you).

Recursion trees

The recursion tree method works by using the recursive definition of the recurrence relation to expand it until a pattern emerges.
For example, with the recurrence relation

$$\begin{aligned} T(n) &= 2T(n-1)+1 && \forall n > 1 \\ T(1) &= 1 \end{aligned}$$

we can expand $T(n-1)$, then $T(n-2)$:

$$\begin{aligned} T(n) &= 2T(n-1)+1 \\ &= 2(2T(n-2)+1)+1 \\ &= 2(2(2T(n-3)+1)+1)+1 && \text{note 2 expansions were made} \\ &= 2^3T(n-3)+4+2+1 && \text{note 2 expansions were made} \end{aligned}$$

with a little imagination, I can see that after k expansions I will have that

$$T(n) = 2^{k+1}T(n-(k+1)) + \sum_{i=0}^{k} 2^i$$

How many expansions can I make?

I should stop when $n-(k+1)=1$, since I know that $T(1)=1$.

So I can make $k = n - 2$ expansions.

Therefore,

$$\begin{aligned} T(n) &= 2^{n-2+1}T(n-(n-2+1)) + \sum_{i=0}^{n-2} 2^i \\ &= 2^{n-1}T(1) + \sum_{i=0}^{n-2} 2^i \\ &= 2^{n-1} + \frac{1-2^{n-1}}{1-2} \\ &= 2^{n-1} + 2^{n-1} - 1 \\ &= 2 \cdot 2^{n-1} - 1 \\ &= 2^n - 1 \end{aligned}$$

Therefore, $T(n) = \Theta(2^n)$.

If we wanted to we could format this as a direct proof - instead I'll allow us to leave it in this more informal format.

Recursion trees + substitution method

Consider now this recurrence relation:

$$\begin{aligned} &T(n) = 2T(\lfloor n/2 \rfloor) + n && \forall n \geq 2 \\ &T(1) = 1 \end{aligned}$$

as we expand, we have to be careful:

$$\begin{aligned} T(n) &= 2T(\lfloor n/2 \rfloor) + n \\ &= 2\left[2T\left(\left\lfloor \frac{\lfloor n/2 \rfloor}{2} \right\rfloor\right) + \lfloor n/2 \rfloor\right] + n \end{aligned}$$

Yikes.
I'm too scared to try the next expansion.

This is where the recursion tree and substitution method become useful when used in tandem.

The substitution method works by making a guess and doing an induction proof.
The recursion method can be used in a less-rigorous ("handwavy") way to make a good guess.

Considering again our recurrence relation

$$\begin{aligned} T(n) &= 2T(\lfloor n/2 \rfloor) + n \\ T(1) &= 1 \end{aligned}$$

we start over our attempt at the recursion tree method.
Consider instead this similar-looking recurrence relation:

$$\begin{aligned} T'(n) &= 2T'(n/2) + n && \forall n > 1 \\ T'(n) &= 1 && \forall n \leq 1 \end{aligned}$$

Solving $T'(n)$ should (hopefully) give me a good guess about $T(n)$.

Performing the expansions:

$$\begin{aligned} T'(n) &= 2T'(n/2) + n \\ &= 2(2T'(n/4) + n/2) + n \\ &= 2(2(2T'(n/8) + n/4) + n/2) + n \\ &= 2(2(2(2T'(n/16) + n/8) + n/4) + n/2) + n \\ &= 2^4 T'(n/2^4) + 4n \end{aligned}$$

Once we see the pattern here, we know that after k expansions, we will have that

$$T'(n) = 2^{k+1} T'(n/2^{k+1}) + (k+1)n$$

We should stop our expansions when $n/2^{k+1} \leq 1$, or in other words when $k = \log_2(n) - 1$.

Plugging in this value of k into our expression above, we get that

$$\begin{aligned} T'(n) &= 2^{\log_2(n)} T'(n/2^{\log_2(n)}) + \log_2(n)n \\ &= 2^{\log_2(n)} T'(1) + n\log_2(n) \\ &= n + n\log_2(n) \end{aligned}$$

So our guess about $T(n)$ should be $T(n) = \Theta(n\log(n))$.
To actually prove this, we need to use an induction proof, which is where the substitution method comes in:

Proof by Induction (Strong).

Want to prove there exists constants $c, n_0 > 0$ such that $T(n) \leq cn\log_2(n)$ for all $n \geq n_0$.

Induction Step:
Let $n \geq 4$.

Assume there exists $c > 0$ such that for all m such that $1 < m < n$ we have that

$$T(m) \leq cm\log_2(m).$$

Then

$$\begin{aligned} T(n) &= 2T(\lfloor n/2 \rfloor) + n \\ &\leq 2(c\lfloor n/2 \rfloor \log_2(\lfloor n/2 \rfloor)) + n && \text{since } n \geq 4 \text{ we have } \lfloor n/2 \rfloor \geq 2 \\ &\leq 2(cn/2\log_2(n/2)) + n \\ &= cn\log_2(n/2) + n \\ &= cn(\log_2(n) - \log_2(2)) + n \\ &= cn\log_2(n) - cn\log_2(2) + n \\ &\leq cn\log_2(n) && \text{as long as } c > 1 \end{aligned}$$

Base Case:
Sometimes when doing strong induction, you have to consider many base cases.

Consider the case with $n = 2$.

$$T(n) = T(2) = 2 \cdot T(1) + 2 = 4 \leq 2 \cdot 2 = 2 \cdot 2\log_2(2) = 2n\log_2(n)$$

Consider the case with $n = 3$.

$$T(n) = T(3) = 2 \cdot T(1) + 3 = 5 \leq 2 \cdot 3 \leq 2 \cdot 3\log_2(3) = 2n\log_2(n)$$

Conclusion:
For all $n \geq 2$, we have that $T(n) \leq 2n\log_2(n)$.

Therefore, $T(n) = O(n\log_2(n))$. □

For future recurrence relations, we will skip the induction proof.
Instead we will use a combination of recursion trees and upper/lower bound techniques.

Binary Search Algorithm

Consider this possible implementation of the binary search algorithm:

```
int BinarySearch(A, i, j, k)
    if i>j
        index = -1
    else
        midpt = (i+j)/2
        if k = A[midpt]
            index = midpt
        else if k < A[midpt]
            index = BinarySearch(A, i, midpt - 1, k)
        else
            index = BinarySearch(A, midpt + 1, j, k)
    return index
```

Here A is an array index from 1 to n, i and j define the "search area", and k is the value we are searching for.
Whenever we encounter a new algorithm, our first priority should be convincing ourselves that it is a correct algorithm; no point in thinking about the running time if it is a wrong algorithm.

Once we are convinced of correctness, we move on to analyzing the running time.

After some quick reasoning about the algorithm, we see that there isn't a nice expression we can say the running time is equal too - if the value k we are searching for is right in the middle of the array the function returns immediately, and if the value k is not in the array we still have to search the entire array.

There are three "types" of running time we talk about: Best Case, Worst Case, and Average

- The best case running time is a lower bounding function, corresponding to the "fastest" of all possible inputs as $n \to \infty$.
- The worst case running time is an upper bounding function, corresponding to the "slowest" of all possible inputs as $n \to \infty$.
- The average case running time is a function describing the average running time over all possible inputs as $n \to \infty$.

In this class we will focus on the best case and the worst case.
In Foundations 2 you will also talk about the average case.

Best case for BinarySearch

For BinarySearch, the best case is easy to see and give the running time for. Whenever our input comes to us with the value k at the midpoint, the algorithm just does this:

```
int BinarySearch(A, i, j, k)
    if i>j
        ...
    else
        midpt = (i+j)/2
        if k = A[midpt]
            index = midpt
    return index
```

Since we are using the "practical" model of computation, we can consider this to all be happening in constant time.
Any increase to the size of the array has no affect on this best case.
Therefore, the best case running time for BinarySearch is $\Theta(1)$.

Worst case for BinarySearch

Now we need to think about what would give us the worst running time for BinarySearch.

BinarySearch keeps making recursive calls until the value is found, or the search area has shrunk to nothing.

So the maximum amount of work is done when the value k is not in the array at all.

We need to create a recurrence relation that matches this situation.

In general, we can follow this rule to create a recurrence relation for any code:

$$T(n) = f(n)T(g(n)) + h(n)$$

where

1. $f(n)$ is the number of recursive calls that will happen ($f(n) = 1$ or 2 for most algorithms you will see)
2. $g(n)$ describes how the size of the problem changes from one call to the next
3. $h(n)$ describes the amount of work happening before and after the recursive calls

Examining BinarySearch, we see that while there are two recursive calls written in the algorithm, at most only one of them will actually execute in any particular call to the function.

So $f(n) = 1$.

Last see the from one call to the next, the size of the list being examined is cut (roughly) in half.

so $g(n) = n/2$.

We also see that before and after the recursive call, a constant amount of work is being done.

So $h(n) = c$ for some constant c.

Therefore, the recurrence relation which describes the worst case running time of BinarySearch is

$$\begin{aligned} T(n) &= T(n/2) + c \\ T(n) &= 1 \qquad\qquad\qquad\qquad\qquad\qquad n < 1 \end{aligned}$$

Analyzing this, we have that

$$\begin{aligned} T(n) &= T(n/2) + c \\ &= [T(n/4) + c] + c \\ &= [[T(n/8) + c] + c] + c \end{aligned}$$

Therefore, after k expansions, we will have

$$T(n) = T(n/2^{k+1}) + c(k+1)$$

Since we keep making recursive calls while the list has at least one element, we need to know how many substitutions we can make until $n/2^{k+1} = 1$.

Solving for k gives $k = \log_2(n) - 1$, and so we substitute this value of k back into out expression:

$$T(n) = T(n/2^{\log_2(n)-1+1}) + c(\log_2(n) - 1 + 1) = 1 + c\log_2(n)$$

So the worst case running time of BinarySearch is $\Theta(\log(n))$.

Phrasing!

Carefully consider each of the following statements. Some of these statements are true, and some are false:

1. The best case running time of BinarySearch is $\Theta(1)$
2. The worst case running time of BinarySearch is $\Theta(\log(n))$
3. The best case running time of BinarySearch is $\Omega(1)$
4. The worst case running time of BinarySearch is $O(\log(n))$
5. The running time of BinarySearch is $\Theta(1)$ and $\Theta(\log(n))$
6. The running time of BinarySearch is $\Theta(1)$ or $\Theta(\log(n))$
7. The running time of BinarySearch is $\Omega(1)$ and $O(\log(n))$
8. The running time of BinarySearch is $\Omega(1)$ or $O(\log(n))$

Which of these statements are true and which statements are false?
For the statements that are true, are some more accurate or precise than others?