This proof is not very straightforward but we will try to make it as simple as possible. This is the solution we discussed today in class for the problem https://www.hackerrank.com/contests/nyc-abcs19-recursion/challenges/store-the-candies

First lets discuss how to evaluate the time complexity of recursive algorithms. For iterative algorithms, we look at how many times a loop executes and compute the time complexity accordingly. For recursive algorithms though we don't have a clear number of steps like loops but instead we need to see how many times the function gets called. In general, the time complexity will be O(Number of time a function gets called * The work you do in each function call).

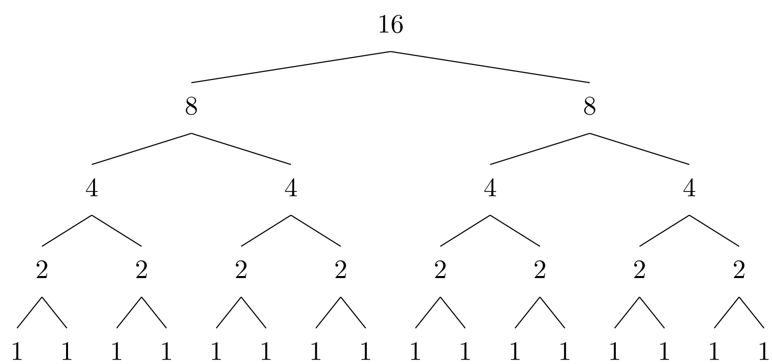
Now each recursive function execution can be represented as a tree (remember our discussion in the session today) so we need to estimate how many nodes we have in this tree.

Lets look at the naive solution first

```
def findNumJars(N, capacity):
    if N <= capacity:
        return 1
    if N % 2 == 1:
        return findNumJars(N // 2, capacity) + findNumJars(N // 2 + 1,
            capacity)
    return findNumJars(N // 2, capacity) + findNumJars(N // 2, capacity)
```

I claim the time complexity of this solution is O(N) because the total number of nodes in the tree will be O(N) and the work we do in each node is O(1)

Proof: First lets visualize the tree for the following input, count = 16, capacity = 1, our tree will look like this.



Before anything, lets define to important things for this tree.

1. Branching factor B, this is basically how many children a node can have, in this case it's 2 because this is a binary tree (the function calls itself twice)

2. Depth D, this is the levels the tree has, since we start with N and divide by 2 all the way to 1, the depth here will be Log(N)

So how do we count the number of nodes in this tree? Lets count the number of nodes in each level and add them together.

The first level we have 1 node, second level we have 2, 3rd level we have 4, 4th level e have 8. Do you see a pattern? At any level X, we will have B^X nodes.

The total number of nodes is $2^0 + 2^1 + 2^2 + 2^3 ...... + 2^D$ but we already know what D is here which is Log(N) so our summation is $2^0 + 2^1 + 2^2 + 2^3 ...... + 2^{Log(N)}$. This summation is bounded by $2^{Log(N)+1} = 2^{Log(N)} * 2$. and since $2^{Log(N)} = N$ then $2^{Log(N)+1} = N * 2 = O(N)$

So that's for the naive solution above. How about the optimized version we discussed in class?

```python
def findNumJars(N, capacity):
    if N <= capacity:
        return 1
    if N % 2 == 1:
        return findNumJars(N // 2, capacity) + findNumJars(N // 2 + 1,
            capacity)
    return findNumJars(N // 2, capacity) * 2
```

One would expect this one to have a better time complexity since we have less branching, but how do we calculate it exactly? In reality, this function has a **best case** and a **worst case**.

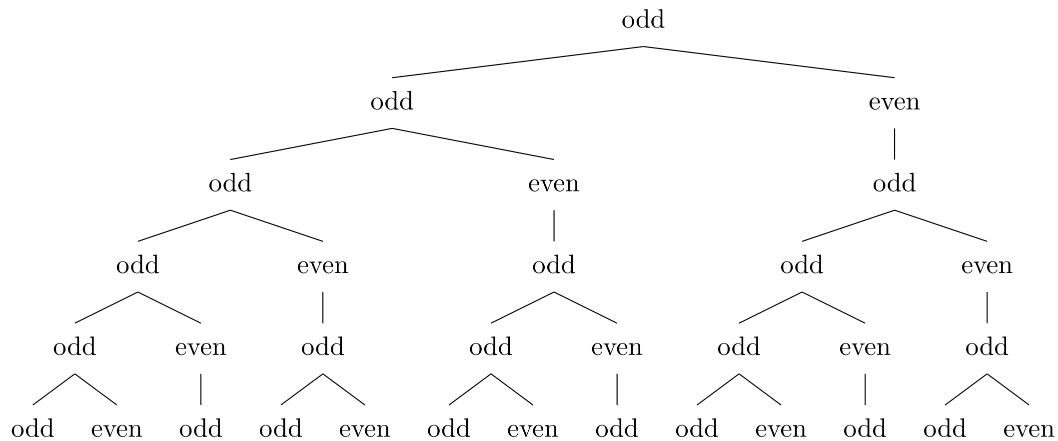
What's the best case? The best case is when we never see any odd numbers in our recursion, the numbers that satisfy this property are powers of 2, so 2, 4, 8, 16, 32, 64, 128. Lets look at a recursive tree

```
16
|
8
|
4
|
2
|
1
```

As you can see, we only have Log(N) nodes in this tree because each function call makes only one single other call and the depth is Log(N) so this is the **best case**.

How about the worst case? The worst case is when we have as many odds as possible.

Lets look at an example of the worst tree possible.



This is the worst shape your tree can ever look like, an odd number goes to one even number and one odd number but even numbers can either go to an odd number or even number, in the worst case they go to an odd number.

So how do we estimate the number of nodes in this tree? Lets do as we did before, try to estimate a branching factor, and a depth.

1. Depth D, this is similar to the naive solution, since we start with N and divide by 2 all the way to 1, the depth here will be Log(N)

2. Branching factor B, this one is not very straightforward, but lets see what we can do. You'd notice that if we look at the branching factor of each level it's not really helpful since some nodes have branching of two (odds) and some other nodes have a branching of one (even). The key point is instead of looking at the branching at separate levels, **we will look at the branching of two levels together** .

   If you look at any **two levels** together under any node, you will realize that the branching factor is **at most 3**. In our previous naive solution, the branching factor for any two levels was 4, here it's 3 which is better because we have less nodes.

So we can say that our branching factor B = 3 but our depth is now halved so it's Log(N) / 2 because our branching factor is per two levels not per each level.

So how many nodes do we have? we can do the summation we did before, the first two levels we have 3^1 at most, the second two levels we have 3^2 at most, the third two levels we have 3^3 nodes at most all the way to Log(N) / 2 levels.

So our summation is: $3^1 + 3^2 + 3^3 .... + 3^{Log(N)/2}$ which is bounded by $3^{(Log(n)/2)+1} = 3^{(Log(n)/2)} * 3$.

We can simplify this $3^{Log(n)/2} = 3^{1/2^{Log(n)}} = sqrt(3)^{Log(N)} = 1.732^{Log(N)}$.

So, $3^{(Log(n)/2)} * 3 = 1.732^{Log(N)} * 3 = O(1.732^{Log(N)})$

So the worst case complexity is $O(1.732^{Log(N)})$ which is better than the naive solution that was $O(2^{Log(N)})$