The analysis was inspired by a problem involving Santa Claus, 1024 wrapped gifts (with one slightly heavier), and a balance.

The question: What is the least number of weighs to find out which gift out of the 1024 is the slightly heavier one?

The video answer: A simple binary divide and conquer algorithm computes the solution in 10 comparisons every time.

The Binary Divide and Conquer Algorithm:

```
def findBox(int[] weights):
    mid = len(weights)/2
    if totalWeight(weights[0:mid]) < totalWeight(weights[mid:]):
        return findBox(weights[mid:])
    return findBox(weights[0:mid])
```

Our recurrence relation is then $T(n) = T(n/2) + O(1)$, giving us a runtime of $log_2 n$ comparisons. This is obviously a much better approach than a nave approach, which makes $O(n)$ comparisons to see when two boxes weigh different.

However, my intuition was that instead of dividing the boxes into TWO groups every time, we could instead divide it into THREE. Then, wed compare two of these thirds, and then if they were equal, wed know the heavier box MUST be in the third we didnt weigh, by simple parity. If they werent equal, wed know the heavier box is in the third that was heavier. Thus, our problem size goes down by a factor of 3 each time. There was one problem that I thought about (that the original problem managed to circumvent since 1024 is a power of 2): what if the sample size cant be divided into 3 groups? Well, I reasoned that at most you could have 2 leftover; wed just put these in the third we arent putting on the balance. Then, if the two were weighing are equal, wed have the other third (with the extras) be our new set. This still cuts the amount of boxes we look at asymptotically by 3 each iteration.

The Ternary Divide and Conquer Algorithm:

```
def heavierBoxByThirds(int[] weights):
    third1, third2 = len(weights)/3, 2*len(weights)/3
    if totalWeight(weights[0:third1]) > totalWeight(weights[third1:third2]):
        return heavierBoxByThirds(weights[0:third1])
    elif totalWeight(weights[0:third1]) < totalWeight(weights[third1:third2]):
        return heavierBoxByThirds(weights[third1:third2])
    return heavierBoxByThirds(weights[third2:])
```

The recurrence relation is thus $T(n) = T(n/3) + O(1)$, leading to a runtime of $log_3 n$, asymptotically faster than the binary divide and conquer algorithm.

One potential issue that came up: What if at each splitting into thirds, we find that the first two thirds are always equal in weight (so the heavier box is in the third with extras)? This would mean that our problem size wouldnt be decreasing from n to n/3 at each iteration, but rather n to $n/3 + 2$ (at worst case). Could this ever be worse than the binary divide and conquer, which is a

decrease from n to $n/2$? Well, $n/3 + 2 < n/2$ for all $n > 12$. And the problems we are looking at are all over a size of 12! So we are golden. Fun note: even for sizes under 12, it turns out binary and ternary splits are the SAME number of comparisons. Try it out!

So this ternary divide and conquer algorithm should be the most optimal way of finding the heavier box. I confirmed my intuition with a simpler version of this problem.

Bonus:

I thought of a way to optimize the Binary Divide and Conquer Algorithm. It turns out a similar quarternary divide and conquer algorithm would have the same worst case run-time, but a MUCH better expected runtime! The thing about our binary divide and conquer algorithm is that is MUST go through $log_2 n$ comparisons. Theres no way to skip steps. If we instead to a quarternary divide, we could get lucky sometimes and cut the sample size from n to n/4 in just one comparison instead of two! Even if the two quarters are even, we will still have eliminated half of the boxes, and we still get a decrease from n to n/2 in one comparison, just as we did with the binary divide and conquer.

So half of the time, well get lucky and get a size reduction of 4 and half the time well just have to settle four a size reduction of 2. This averages out to a size reduction of 3! Which means the quarternary divide and conquer actually has an AVERAGE running time that ties with ternary divide and conquer. Cool stuff! Unfortunately, it seems like further splits wouldnt be optimal. After you compare less than half of the original boxes each time

$$(1/5 + 1/5 = 2/5 < 1/2)$$

you cant really take advantage of divide and conquer anymore. To put my solution in context with the binary divide and conquer: $log_2 1024 = 10$ comparisons. $log_3 1024 = 6.3$ comparisons! Thats 37% faster.

Anyways, it was fun thinking about the problem It was fun using some of my Algorithms class knowledge, but most importantly, its cool that my solution works.