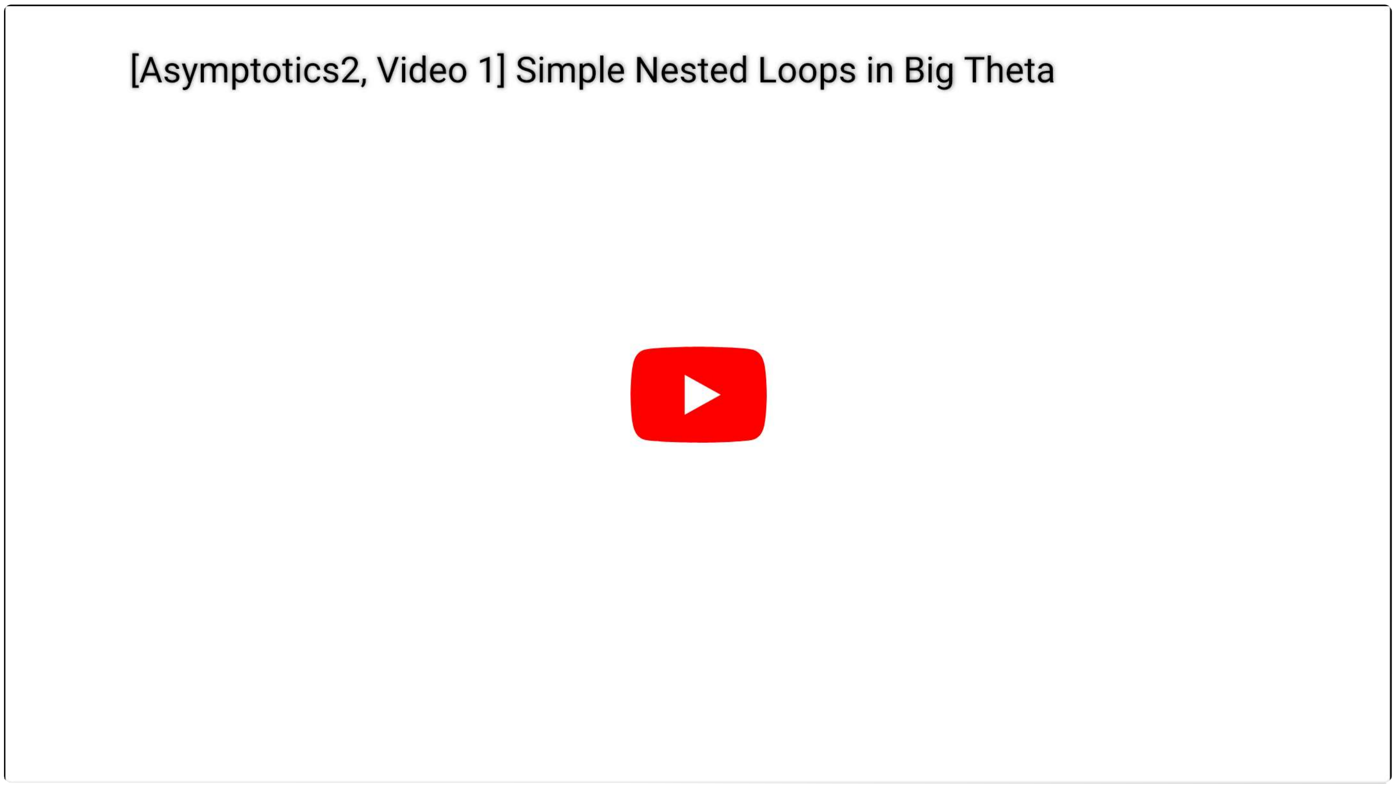
15.1 For Loops

Count, count, count...

Now that we've seen some runtime analysis, let's work through some more difficult examples. Our goal is to get some practice with the patterns and methods involved in runtime analysis. This can be a tricky idea to get a handle on, so the more practice the better.

Example One:



Last time, we saw the function dup1, that checks for the first time any entry is duplicated in a list:

```java
int N = A.length;
for (int i = 0; i < N; i += 1)
   for (int j = i + 1; j < N; j += 1)
      if (A[i] == A[j])
         return true;
return false;
```

We have two ways of approaching our runtime analysis:

1. Counting number of operations
2. Geometric visualization

Method 1: Count Number of Operations

Since the main repeating operation is the comparator, we will count the number of **"=="** operations that must occur.

The first time through the outer loop, the inner loop will run $N-1$times. The second time, it will run $N-2$ times. Then $N-3$, $N-4$, ... all the way till running the inner loop exactly 1 time when i = $N-1$. In the worst case, we have to go through every entry, and the outer loop runs N times.

Then, let C = total number of "==" operations that have occurred. The number of comparisons is:

$$C = 1 + 2 + 3 + ... + (N-3) + (N-2) + (N-1) = N(N-1)/2$$

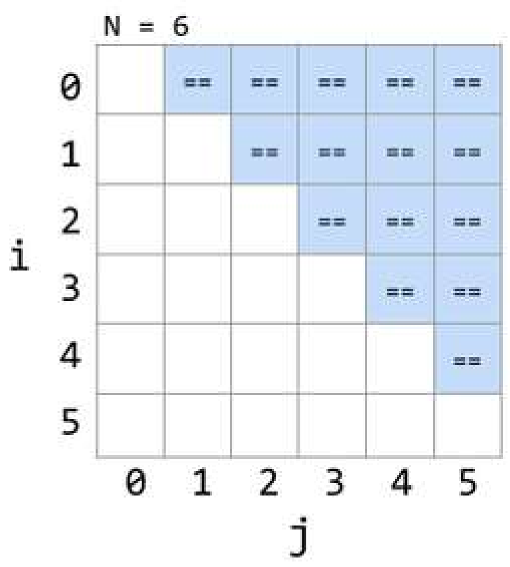
where $N(N-1)/2$ is part of the N^2 family.

Since "==" is a constant time operation, the overall runtime in the worst case is $\theta(N^2)$.

Method 2: Geometric Visualization

We can also approach this from a geometric view.

Let's draw out when we use == operations in the grid of i, jcombinations:



We see that the number of == operations is the same as the *area* of a right triangle with a side length of $N-1$. Since area is in the N^2 family, we see again that the overall runtime is $\theta(N^2)$.

Example 2:

[Asymptotics2, Video 2] Nested For Loops with Geometric Outer Loop

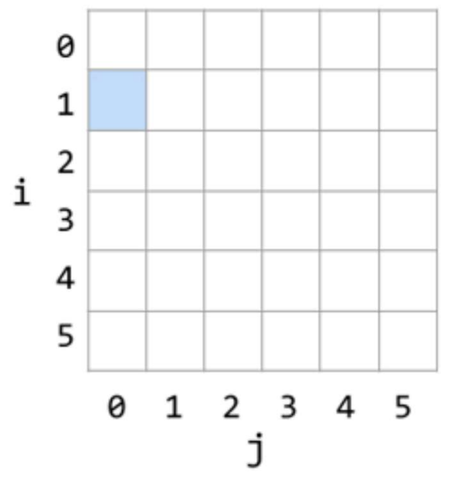
Let's look at a more involved example next. Consider the following function, with similar nested for loops:

```java
public static void printParty(int N) {
   for (int i = 1; i <= N; i = i * 2) {
      for (int j = 0; j < i; j += 1) {
         System.out.println("hello");
         int ZUG = 1 + 1;
      }
   }
}
```

The outer loop advances by *multiplying* `i` by 2 each time. The inner loop runs from 0 to the current value of `i`. The two operations inside the loop are both constant time, so let's approach this by asking **"how many times does this print out "hello" for a given value of N?"**

Our visualization tool from above helped us see dup1's runtime, so let's use a similar approach here. We'll lay out the grid for the nested for loops, and then track the total number of print statements needed for a given N below.

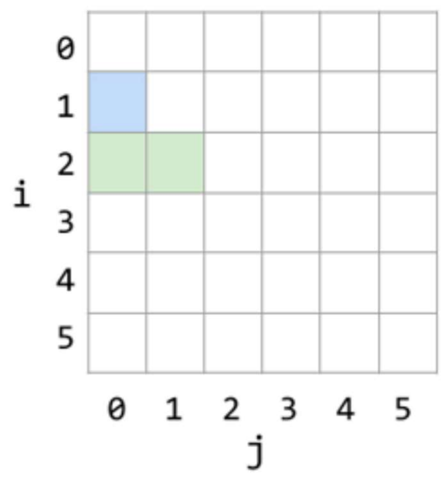
If N is 1, then `i` only reaches 1, and `j` is only 0, since 0 < 1. So there is only one print statement:

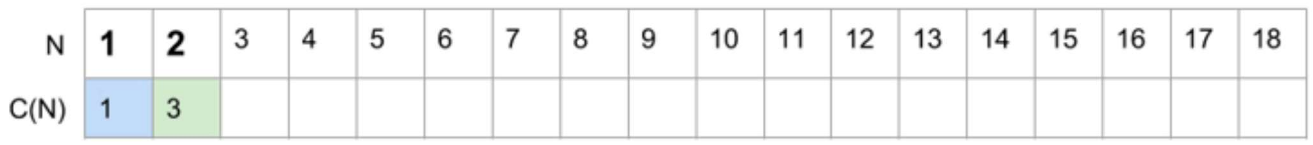


N	**1**	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
C(N)	1																	

Visualizations when N = 1

If N is 2, the next time through the loop `i` will be $1 * 2 = 2$, and `j` can reach 1. The total number of print statements will be 3 = 1 (from first loop) + 2 (from second loop).



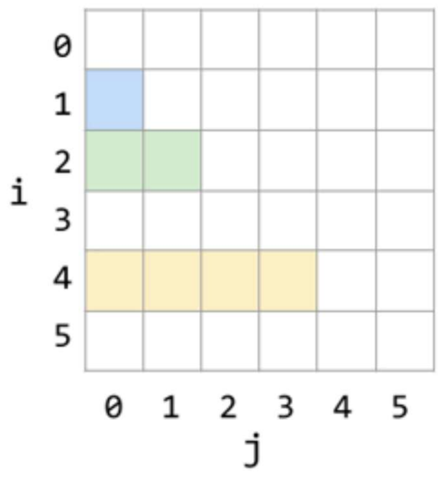


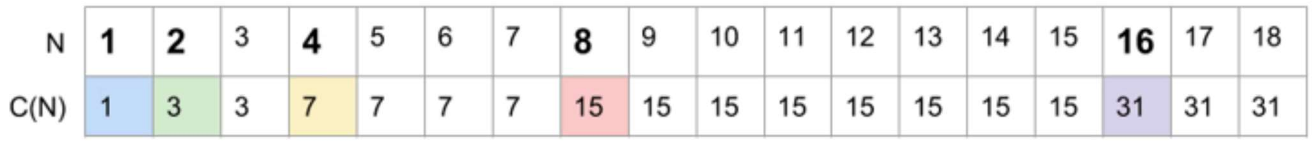
N	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
C(N)	1	3																

Visualizations when N = 2

› Conceptual Check: What happens when N = 3?

We can keep filling out our diagram to get a fuller picture. Here it is up to N = 18:





N	**1**	**2**	3	**4**	5	6	7	**8**	9	10	11	12	13	14	15	**16**	17	18
C(N)	1	3	3	7	7	7	7	15	15	15	15	15	15	15	15	31	31	31

Visualizations for N = 18

What we see, if we add up all the counts at each stage of the loops, is that the number of print statements is:

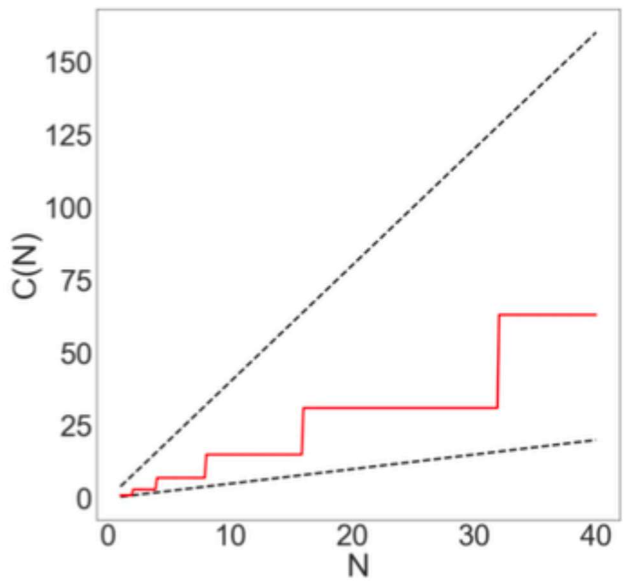
$$C(N) = 1 + 2 + 4 + ... + N$$

(if N is a power of 2).

Again, we can think of this in two ways. Since we're already on a graphical roll, let's start there.

Method 1: Finding the Bound Visually

If we graph the trajectory of 0.5 N (lower dashed line), and 4N (upper dashed line), and $C(N)$ itself (the red staircase line), we see that C(N) is fully bounded between those two dashed lines.



Therefore, the runtime (by definition) must also be linear: $\theta(N)$.

Method 2: Finding the Bound Mathematically

We can obtain the same result by solving our equation from above with the power of mathematics:

$$C(N) = 1 + 2 + 4 + ... + N = 2N - 1$$

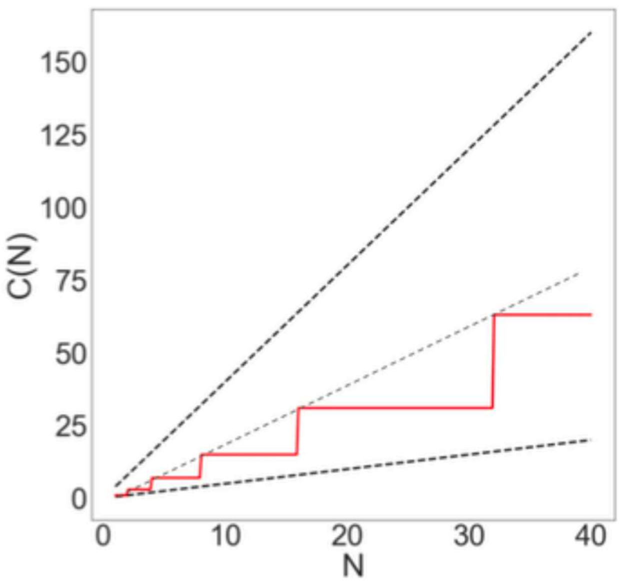
Again if N is a power of 2.

For example, if $N = 8$:

$$C(N) = 1 + 2 + 4 + 8 = 15 = 2 * 8 - 1$$

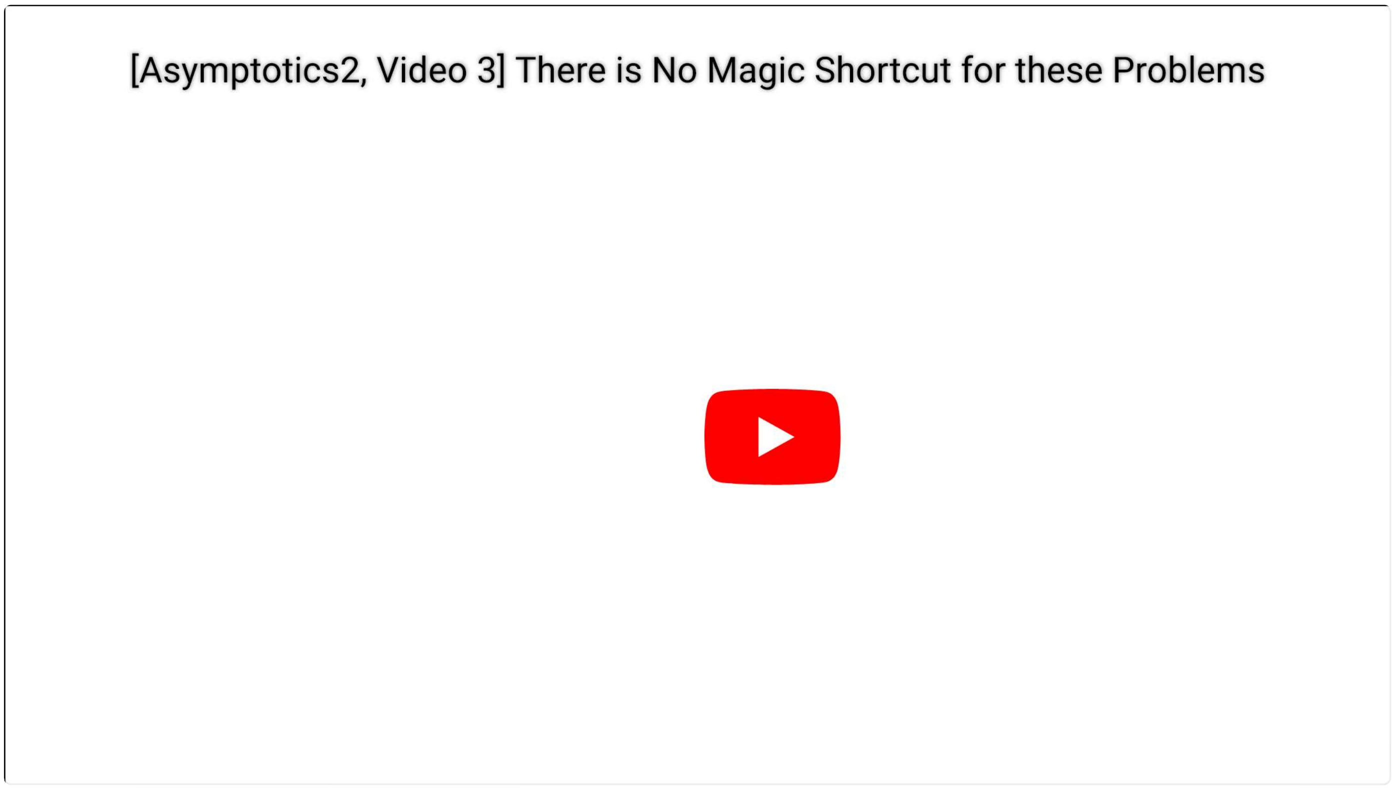
And by removing lesser terms and multiplicative constants, we know that $2N - 1$ is in the linear family, so the runtime is $\theta(N)$.

We now get a more exact value to the red-staircase line plotted below, which is $2N$.



Graph of 2N, bounded by 0.5N and 4N

Techniques: No Magic Shortcuts



[Asymptotics2, Video 3] There is No Magic Shortcut for these Problems

It would be really nice if there were some magic way to look at an algorithm and just *know* its runtime. And it would be even nicer if all nested for loops have a runtime of N^2 .

Unfortunately, they're not. And we know this because we just did two nested for loop examples above, each with *different* runtimes.

In the end, there is **no shortcut** to doing runtime analysis. It requires careful thought. But there are a few useful techniques and things to know:

- **Find exact sum**
- **Write out examples**
- **Draw pictures**

We used each of these techniques above.

Also used in the examples above are two important sums you'll see very often:

- **Sum of First Natural Numbers**: $1 + 2 + 3 + ... + Q = Q(Q + 1)/2 = \Theta(Q2)$
- **Sum of First Powers of 2**: $1 + 2 + 4 + 8 + ... + Q = 2Q - 1 = \Theta(Q)$

You saw both of these above, and they'll return again and again in runtime analysis.

Previous
15. Asymptotics II

Next
15.2 Recursion

Last updated 1 year ago