Big-Os, the Rationale.

Consider a little function like this. In fact, exactly this function:

```
1     double compute(double A, double N)
2     { double answer=1.0;
3       for (int i=1; i<=N; i+=1)
4       { double part=1.0;
5         for (int j=1; j<=N; j+=1)
6           part=part*A;
7         answer=answer*part; }
8       return answer; }
```

It computes ((A to-the-power-of N) to-the-power-of N) very inefficiently.

How long does the function take to run? Obviously we can't say. The answer clearly depends on the value of N, and will vary significantly from one computer to another. But we can say something about the answer.

Look at the simple statement on line 6. Although we can't say how much time it will take, we do know that it doesn't depend on anything. Each time it is executed it will take the same amount of time. Call that amount of time T_1.

Now consider the loop on lines 5 to 6. It will certainly execute N times, each time it will execute statement 6, but it will also have to do `j+=1` and check `j<=N` each time too. Again we can't say how long those two statements will take, but we do know that time will not change. Let's define T_2 to be the total time taken for `j<=N`, `part=part*A`, and `j+=1`.

So the time required to execute the whole inner loop, lines 5 to 6 is $T_2 \times N$.

Now look at the outer loop. Each time it goes round, it must check `i<=N`, perform `part=1.0`, `j=1`, and `answer=answer*part`, as well as executing the whole inner loop. Let's define T_3 to be the total time taken for `i<=N`, `part=1.0`, `j=1`, and `answer=answer*part`.

So the time taken for each time round the outer loop is going to be $T_3+T_2 \times N$.

This outer loop clearly goes round N times, so its total execution time will be $T_2 \times N^2+T_3 \times N$.

On top of executing the inner loop, lines 3 to 7, the whole function must also perform `answer=1.0`, `i=1`, and `return answer`. Of course we don't know how long those three operations will take, but they clearly don't depend on anything: they will always take the same unknown amount of time. Let's call it T_4.

So the run-time for the whole function is $T_2 \times N^2 + T_3 \times N + T_4$, whatever T_2, T_3, and T_4 turn out to be.

The important thing to note is that the only variable here is N. The T values are unknown, but they won't change each time the function is called. Contrarily, each time the function is called, N will be known, but it can be different each time.

To make the formula easier to look at, I'll write it as: $time = aN^2 + bN + c$. A very familiar quadratic.

Although we don't know what a, b, and c are, we do know that they are quite small. They all represent the time it takes a modern computer to perform a couple of very simple operations. If the value N isn't very big, the total time will be too short to measure.

But nobody cares about that. If a function is so fast that it only takes a micro-second to run, nobody is going to be worried. It is only when things start to take a long time that time really matters.

A, b, and c are very small and quite similar to each other, so the only way that the time could get large is when N gets large.

And what happens when N gets large? N=1,000 is hardly large at all by modern standards, but still, time = 1,000,000a + 1,000b + c. A's significance is 1,000 times that of b.

Getting larger, N=1,000,000: time = 1,000,000,000,000a + 1,000,000b + c. The influence of A is a million times greater than that of B.

Whenever N is large, the influence of B and C diminishes well below the limits of practical measurement. They are *useless*.

You'll remember from calculus or analysis, that in the limit as N increases:

$$1 << \log_2 N << N << N^2 << N^3 << N^4 << .... << 2^N << N^N$$

For even a slightly large N, the larger terms will always make all lower terms insignificant.
(N^N is very similar to N!)

Getting back to our function, it is perfectly safe and reasonable to say the running time is aN^2.
And even that is saying too much. "a" is a totally unknown constant. All it really does is express the power of your computer hardware. You can make a as small as you like by getting faster computers (within reasonable limits anyway), and you can make a as large as you like by slowing down your computer. The constant a could be any value you care to make it, except zero or negative of course. Particularly as we haven't said anything about what units we are going to measure time in. "a" is irrelevant and meaningless.
The only meaningful fact is that the time taken by our function is proportional to N squared. Meaning that if you double N, you'll quadruple the time; if you triple N, you'll nine-ple the time.

That is what Big-O is all about.

Our function is $O(N^2)$

And that's it. With that one piece of knowledge, all it takes is a couple of experiments on any computer (really measuring the time for a chosen large N) and you'll be able to work out the time for that particular computer for any value of N.

O(1)=constant, O(logN)=logarithmic, O(N)=linear, $O(N^2)$=quadratic,
$O(N^3)$=cubic, $O(N^4)$=quartic, $O(2^N)$=exponential.