Why Analyze Algorithms?

- Procedure or formula for solving a problem
- Some are so useful they have names:
  - merge sort
  - bubble sort
  - insertion sort
  - ...
- How can we compare the algos to know which is better?

Exercise

Implement the function below, and measure how long time it takes to run it for different sizes of n (e.g. n = 1000, n = 50000, n = 1000000, and n = 1000000000).

(in java, you can e.g. use System.nanoTime() to obtain system time)

In [1]:
```java
public static int sum1( int n )
{
    // Take an input of n and return the sum of the numbers from 0 to n
    int partialSum;
    partialSum = 0;

    for (int i = 0; i <= n; i++){
        partialSum +=i;
    }
    return partialSum;
}
```

In [2]:
```java
// Try out the method sum1
sum1(50000)
```

Out[2]:
```
1250025000
```

In [3]:
```
// Let us try to time it

long nano_startTime = System.nanoTime();             // nano =  1 x 10^-9 of a second
long millis_startTime = System.currentTimeMillis();  // milli = 1 x 10^-3 of a second

sum1(1000000);

long nano_endTime = System.nanoTime();
long millis_endTime = System.currentTimeMillis();

System.out.println("Time taken in nano seconds: "
                           + (nano_endTime - nano_startTime));

System.out.println("Time taken in milli seconds: "
                           + (millis_endTime - millis_startTime));
```

```
Time taken in nano seconds: 46549100
Time taken in milli seconds: 48
```

The above is ok but let us try multiple values of n. For this we can use a for loop.

In [4]:
```
// We want to try out these values so we create an array
int [] numbers = {1000, 50000, 1000000, 1000000000};

for(int number: numbers) {      // this is called a "for each loop" or "enhanced for loop"

    long nano_startTime = System.nanoTime();
    long millis_startTime = System.currentTimeMillis();

    sum1(number);

    long nano_endTime = System.nanoTime();
    long millis_endTime = System.currentTimeMillis();

    System.out.println("Time taken in nano seconds: "
                           + (nano_endTime - nano_startTime));

    System.out.println("Time taken in milli seconds: "
                           + (millis_endTime - millis_startTime));

}
```

```
Time taken in nano seconds: 4900
Time taken in milli seconds: 0
Time taken in nano seconds: 83900
Time taken in milli seconds: 0
Time taken in nano seconds: 460200
Time taken in milli seconds: 0
Time taken in nano seconds: 426921800
Time taken in milli seconds: 427
```

The loop above is basically equal to the series:

$$\sum_{k=0}^{n} k = 1 + 2 + 3 + \ldots + n = \frac{n(n+1)}{2}$$

So let's try this one out:

In [5]:
```
public static int sum2( int n )
{
    // Take an input of n and return the sum of the numbers from 0 to n
    return (n*(n+1))/2;

}
```

In [6]:
```
// Let us try out our second function
sum2(100)
```

Out[6]:
```
5050
```

Let us "objectively" compare them

Function sum1 uses a loop to iteratively add across our range +1
Function sum2 uses a formula to solve a problem

- memory space
- time to run

sum1 vs. sum2

In [7]:
```
long nano_startTime = System.nanoTime();
long millis_startTime = System.currentTimeMillis();

sum1(1000000000);

long nano_endTime = System.nanoTime();
long millis_endTime = System.currentTimeMillis();

System.out.println("Time taken in nano seconds: "
                            + (nano_endTime - nano_startTime));

System.out.println("Time taken in milli seconds: "
                            + (millis_endTime - millis_startTime));
```

```
Time taken in nano seconds: 323273300
Time taken in milli seconds: 323
```

In [8]:
```
long nano_startTime = System.nanoTime();
long millis_startTime = System.currentTimeMillis();

sum2(1000000000);

long nano_endTime = System.nanoTime();
long millis_endTime = System.currentTimeMillis();

System.out.println("Time taken in nano seconds: "
                           + (nano_endTime - nano_startTime));

System.out.println("Time taken in milli seconds: "
                           + (millis_endTime - millis_startTime));
```

```
Time taken in nano seconds: 50200900
Time taken in milli seconds: 40
```

So we really don't get any smarter!

Obviously a the smaller the value of time is better

- connot simply rely on time to run because all computers are different and some faster than others
- to be hardware independent = Big 0

The original sum1 function will create an assignment $\mathbf{n + 1}$ times, we can see this from the range based function. This means it will assign the partialSum variable $n + 1$ times. We can then say that for a problem of n size (in this case just a number n) this function will take $1 + n$ steps.

This n notation allows us to compare solutions and algorithms relative to the size of the problem, since sum1(10) and sum1(100000) would take very different times to run but be using the same algorithm. We can also note that as n grows very large, the $+1$ won't have much effect. So let's begin discussing how to build a syntax for this notation.

Big-O notation describes **how quickly runtime will grow relative to the input as the input get arbitrarily large**.

Remember: $\mathcal{O}(\cdot)$ gives us the **worst case** scenario of the algorithm in question! Later we will also look at the best case and the average case scenarios.

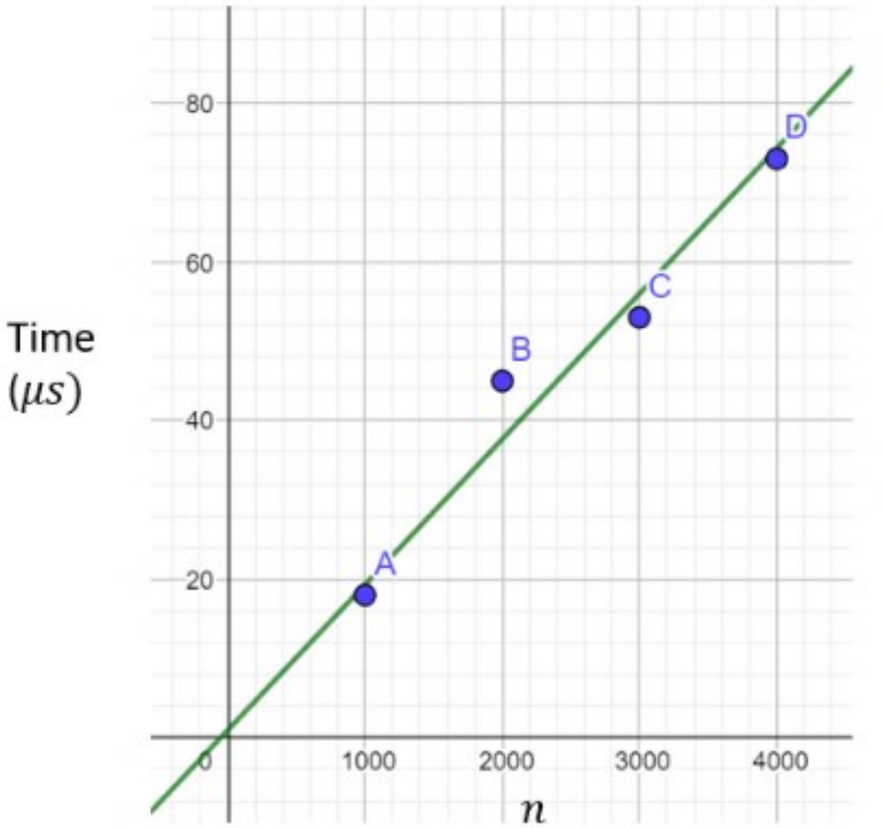
Big $\mathcal{O}$-notation

We have a special kind of notation to denote the time complexity of an algorithm: “Big O notation”.

For example, if an algorithm runs in linear time, we say that it has a time complexity of $\mathcal{O}(n)$. If it runs in quadratic time, the time complexity is $\mathcal{O}(n^2)$.

In the next slides, we will see how to determine this if we are given the runtime as a function of input size for a given algorithm.

Big $\mathcal{O}$-notation



The graph shows the runtime of the "sum"-function on my computer. It is approximately

$$T = 1 + 0.02 \cdot n$$

To determine the time complexity of this function, go through the following steps:

- **Step 1: Find the fastest growing term**

  In this case, it is $0.02 \cdot n$

- **Step 2: Remove the coefficient**

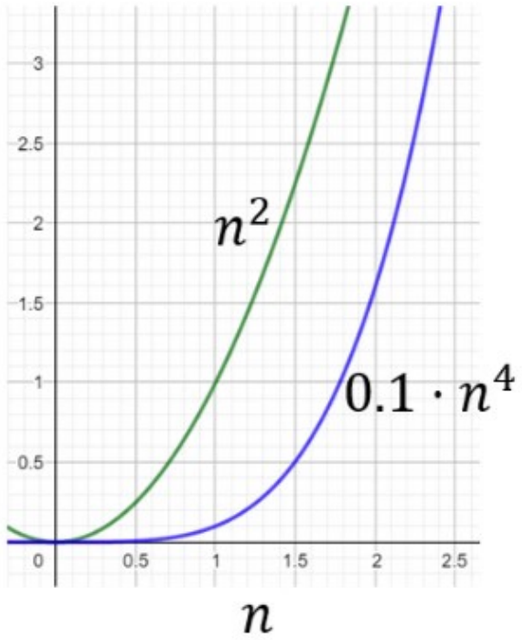
  In this case, we remove the coefficient 0.02, so only n is left.

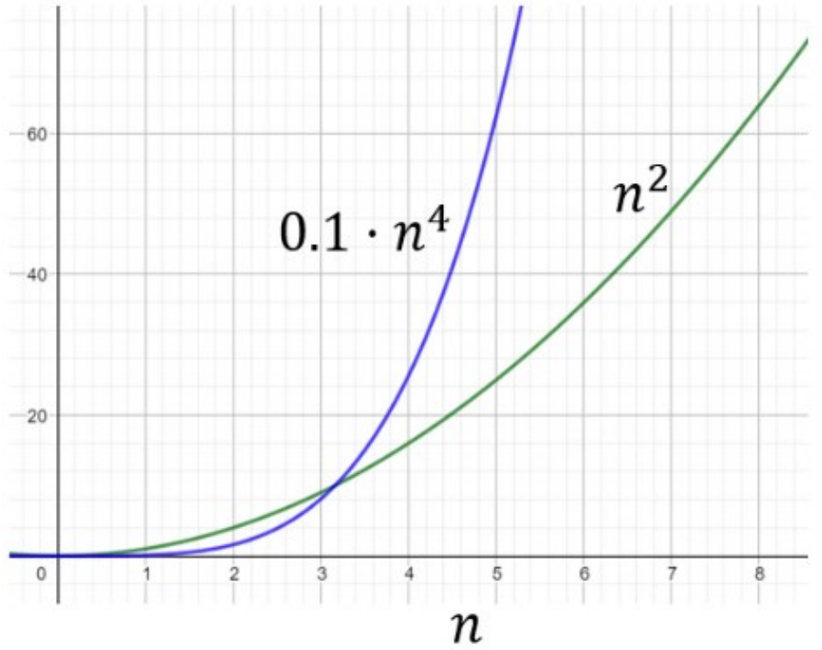
- **Whatever is left is the time complexity!**

  So in this case $\boldsymbol{T = \mathcal{O}(n)}$

… how do you find the fastest growing term?

Example: Which term grows faster, n^2 or $0.1 \cdot n^4$?





HOW WILL IT SCALE as data increases?

Try with larger numbers!

Remember, we want to compare how quickly runtime will grow, not compare exact runtimes, since those can vary depending on hardware.

Since we want to compare for a variety of input sizes, we are only concerned with how runtime grow relative to the input. This is why we use n for notation.

As n gets arbitrarily large we only worry about terms that will grow the fastest as n gets large, to this point, Big-O analysis is also known as asymptotic analysis.

In math, asymptotic analysis = **describing limiting behavior**

- which part of the algorithm has the GREATEST effect on final answer, which part of algo is the real bottleneck, which part is the limiting factor.

As for syntax, sum1() can be said to be $O(n)$ since its runtime grows linearly with the input size, where as sum2 does not need to loop through anything so the time complexity is constant, and thus the complexity is $O(1)$. So here changing to a more efficient algorithm (i.e. the closed form) gives us more fficient code.

When the runtime runs *linearly* as a function of the input size, we say that the algorithm has **linear time complexity**.

Asymptotic

Big O is what is known as an **asymptotic function**.

Another thing to note is that there are other asymptotic functions. Big Θ (theta) and Big Ω (omega) also both describe algorithms at the limit (remember, the limit just means for huge inputs).

- Big O describes the set of all algorithms that run no worse than a certain speed (it's an upper bound)
- Conversely, Big Ω describes the set of all algorithms that run no better than a certain speed (it's a lower bound)
- Finally. Big Θ describes the set of all algorithms that run at a certain speed (it's like equality or average)

Everything that is $\Theta(\cdot)$ is also $O(\cdot)$, but **not** the other way around.

Time complexity is said to be in $\Theta(\cdot)$ if it is both in $O(\cdot)$ and in $\Omega(\cdot)$.

In sets terminology, $\Theta(\cdot)$ is the intersection of $O(\cdot)$ and $\Omega(\cdot)$

Usually, you'll hear things described using Big O, but it doesn't hurt to know about Big Θ and Big Ω.

In [9]:
```
import java.lang.Math;

public double BigO(double n)
{
    return 45*Math.pow(n,3) + 20*Math.pow(n,2)+19;
}
```

In [10]:
```
// n = 1
BigO(1)
```

Out[10]:
```
84.0
```

In [11]:
```
// n = 2
BigO(2)
```

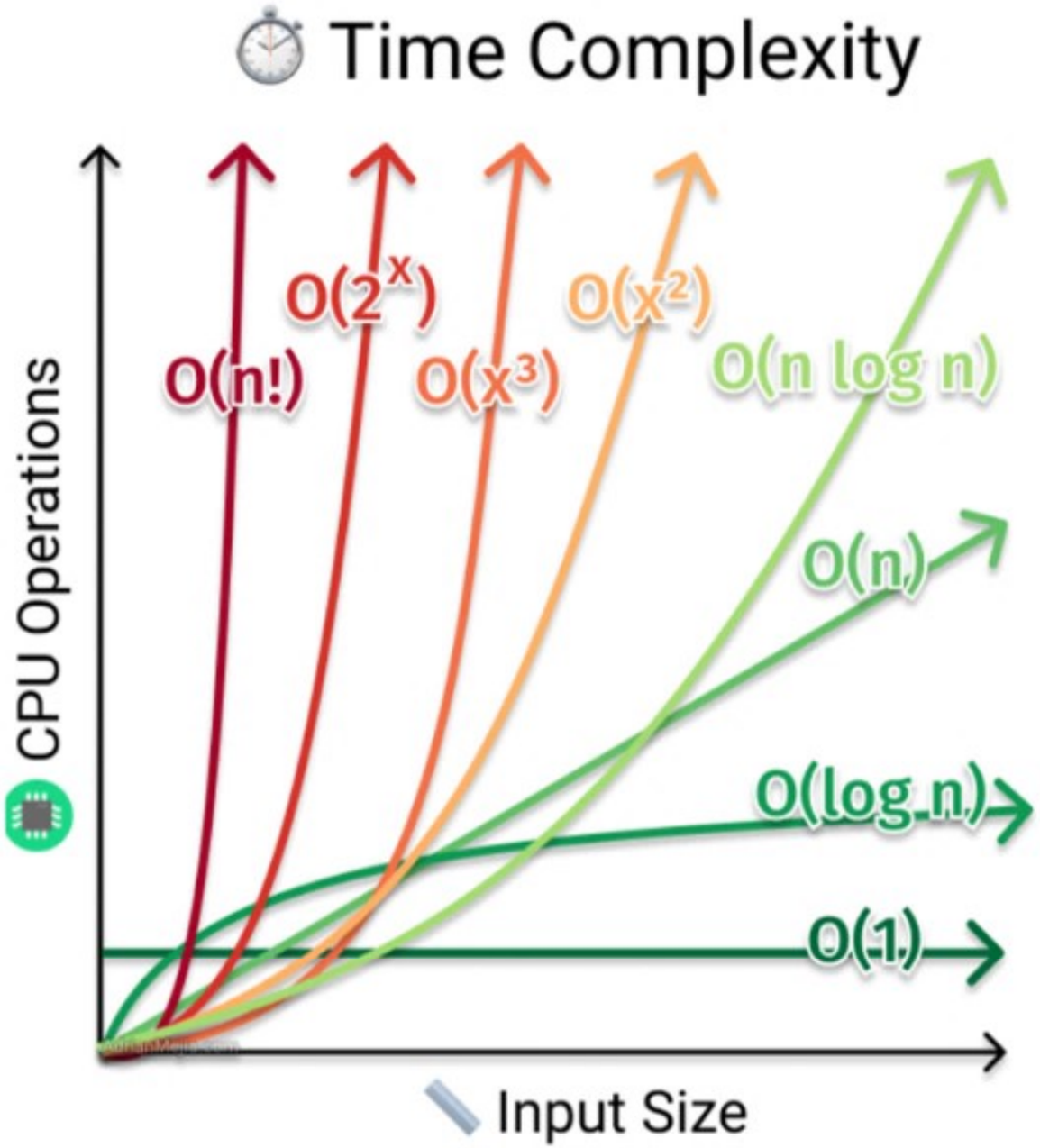
Out[11]:
```
459.0
```

In [12]:
```
// n = 100
BigO(10000000)
```

Out[12]:
```
4.5000002E22
```

It can be seen that the 19 does not hold much weight anymore.
The $20n^2$ in this case is 2000
The $45n^3$ in this case is 45,000

The part of this algo that really has a lot to do with final answer as data scales, will not be the 45 but the n^3. So this algo has an order of n^3.



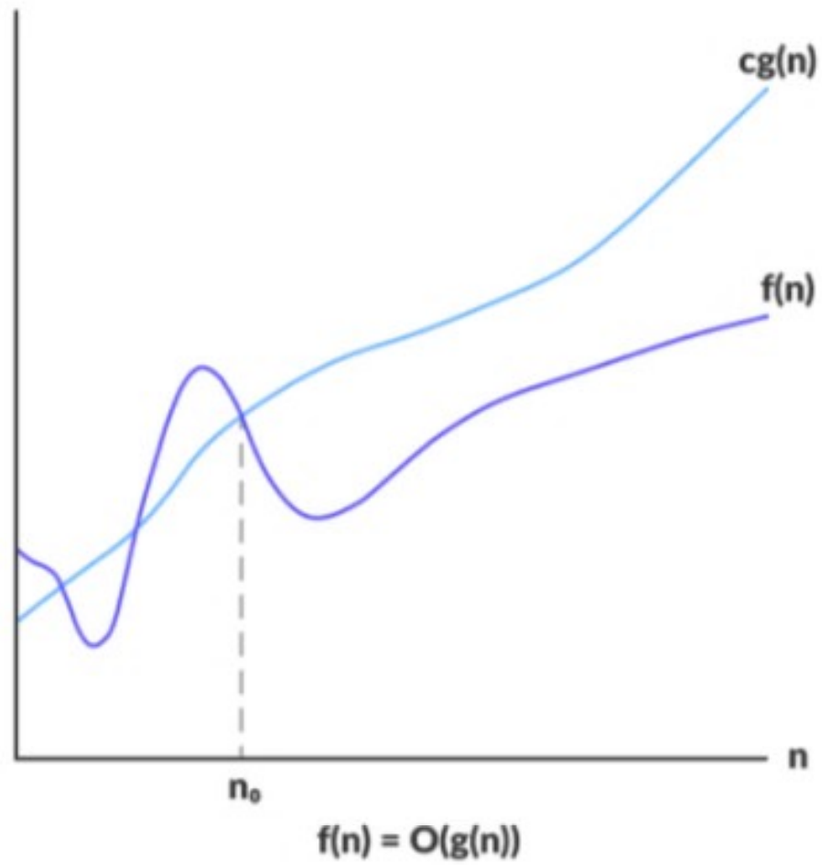
Notation	Common Name	Example Algorithm
O(1)	Constant time	Array index lookup
O(log n)	Logarithmic time	Binary search
O(n)	Linear time	Sequential search
O(n log n)	Log-linear time	Heapsort
O(n^2)	Quadratic	Bubble sort
O(n^c)	Polynomial	
O(c^n)	Exponential	Traveling salesman
O(n!)	Factorial	Brute force traveling salesman
O(n^n)	N to the N	

What does this actually mean?

When we say that $f(n) = \mathcal{O}(g(n))$ it means that $f(n)$ is smaller than or equal to some constant times $g(n)$ for large values of n.

It is equivalent to

$$f(n) = \mathcal{O}(g(n)) \Leftrightarrow \lim_{n \to \infty} \frac{f(n)}{g(n)} \neq \infty$$



Source: https://www.programiz.com/dsa/asymptotic-notations

An alternative measure: Big Θ

When we say that $f(n) = \Theta(g(n))$ it means that $f(n)$ is **smaller than** or equal to some constant times $g(n)$ and **larger than** or equal to another constant times $g(n)$ for large values of n.

It is equivalent to

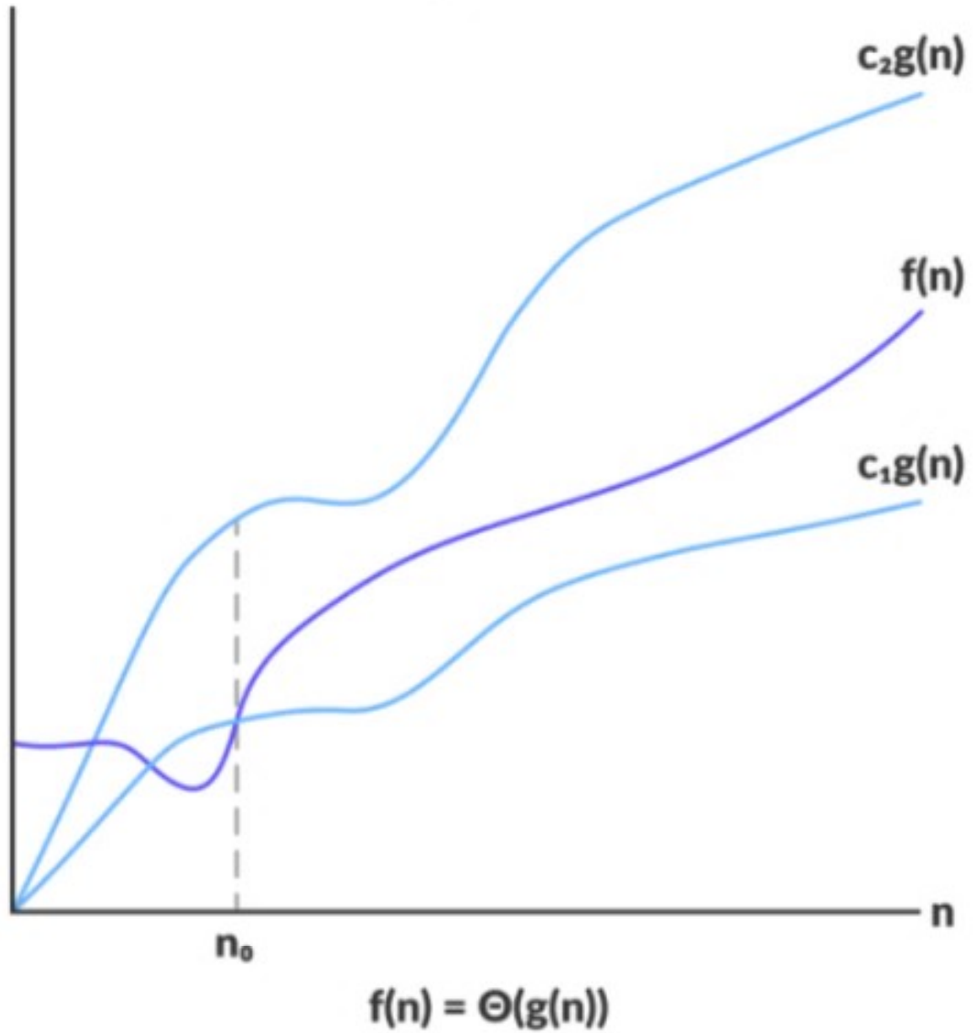
$$f(n) = \Theta\big(g(n)\big)$$

$$\Updownarrow$$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \neq 0 \text{ and } \lim_{n\to\infty} \frac{f(n)}{g(n)} \neq \infty$$

$$\Updownarrow$$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c, c \neq 0$$



A simple comparison

Big O is like $\leq$ .... or maybe more like $\lesssim$

$f(n) = \mathcal{O}(g(n))$ means " $f(n) \lesssim g(n)$" for large n.

Big Θ is like $=$ ... or maybe more like $\approx$

$f(n) = \Theta(g(n))$ means " $f(n) \approx g(n)$" for large n.

How to Determine Complexities

1. Sequence of statements

   statement 1;

   statement 2;

   ...

   statement k;

The total time is found by adding the times for all statements: total time = time(statement 1) + time(statement 2) + ... + time(statement k)

If each statement is "simple" (only involves basic operations) then the time for each statement is constant and the total time is also constant: O(1). In the following examples, assume the statements are simple unless noted otherwise.

If statement

1. if-then-else statements

```
if (condition) {
sequence of statements 1
}
else {
sequence of statements 2
}
```

   Here, either sequence 1 will execute, or sequence 2 will execute. Therefore, the worst-case time is the slowest of the two possibilities:

max(time(sequence 1), time(sequence 2)).

For example, if sequence 1 is O(N) and sequence 2 is O(1) the worst-case time for the whole if-then-else statement would be O(N).

Loops

1. for loops

```
for (i = 0; i < N; i++) {
 sequence of statements
}
```

The loop executes N times, so the sequence of statements also executes N times. Since we assume the statements are O(1), the total time for the for loop is N * O(1), which is O(N) overall.

Nested loops

1. Nested Loops

First we'll consider loops where the number of iterations of the inner loop is **independent** of the value of the outer loop's index. For example:

```
for (i = 0; i < N; i++) {
    for (j = 0; j < M; j++) {
        sequence of statements
    }
}
```

The outer loop executes N times. Every time the outer loop executes, the inner loop executes M times. As a result, the statements in the inner loop execute a total of N *M times. Thus, the complexity is O(N* M). In a common special case where the stopping condition of the inner loop is j < N instead of j < M (i.e., the inner loop also executes N times), the total complexity for the two loops is O(N^2).

Nested loops

Now let's consider nested loops where the number of iterations of the inner loop depends on the value of the outer loop's index. For example:

```
for (i = 0; i < N; i++) {
    for (j = i; j < N; j++) {
        sequence of statements
    }
}
```

Now we can't just multiply the number of iterations of the outer loop times the number of iterations of the inner loop, because the inner loop has a different number of iterations each time. So let's think about how many iterations that inner loop has. That information is given in the following table:

i	inner loop iterations
0	N
1	N-1
2	N-2
...	...
N-2	2
N-1	1

So we can see that the total number of times the sequence of statements executes is: N + N-1 + N-2 + ... + 3 + 2 + 1. We've seen that formula before: the total is O(N^2).

Example

In [13]:
```
public static void ifElseStatement(int n)
{

    if (n == 0) {
        System.out.println("n is zero!");
    }
    else {
        for (int i = 0; i < n; i++) {
            System.out.println("Hi!");
        }

    }

}
```

Best vs. worst case?

In [14]:
```
ifElseStatement(8)
```

```
Hi!
Hi!
Hi!
Hi!
Hi!
Hi!
Hi!
Hi!
```

Big O Examples

O(1) Constant

Example: Finding the time complexity of “verySimpleFunction”.

```
public static int verySimpleFunction(int n){
    int total = 0;
    return total;
}
```

1 time unit

1 time unit

1. Assume each operation takes 1 time unit.
2. Figure out how many time units it takes to run each line.
3. Add them all: $T = 1 + 1 = 2 = \mathcal{O}(1)$

Example

In [15]:
```
int n = 1000;
System.out.println("Hey - your input is: " + n);
```

```
Hey - your input is: 1000
```

Clearly, it doesn't matter what n is, above. This piece of code takes a constant amount of time to run. It's not dependent on the size of n.

In [16]:
```
int n = 10000000;
System.out.println("Hey - your input is: " + n);
System.out.println("Hmm.. I'm doing more stuff with: " + n);
System.out.println("And more: " + n);
```

```
Hey - your input is: 10000000
Hmm.. I'm doing more stuff with: 10000000
And more: 10000000
```

The above example is also constant time. Even if it takes 3 times as long to run, it doesn't depend on the size of the input, n. We denote constant time algorithms as follows: O(1). Note that O(2), O(3) or even O(1000) would mean the same thing.

We don't care about exactly how long it takes to run, only that it takes constant time.

The three lines above is an example of **consecutive statements**. It follows that when we have consequtive statements we only really care about the statement with the largest complexity since the other will have no effect on complexity as n grows bigger.

Logarithmic Time Algorithms – $O(\log n)$

Constant time algorithms are (asymptotically) the quickest. Logarithmic time is the next quickest. Unfortunately, they're a bit trickier to imagine.

One common example of a logarithmic time algorithm is the **binary search** algorithm. We will get back to this in a couple of weeks.

What is important here is that the **running time grows in proportion to the logarithm of the input (in this case, log to the base 2):**

In [17]:
```
// The array of numbers we want to use
int [] numbers = {1, 2, 4, 8, 16};

// We create a loop:
for(int number: numbers) {
    System.out.println("Hey - I'm busy looking at: " + number);
    number *=2;
}
```

```
Hey - I'm busy looking at: 1
Hey - I'm busy looking at: 2
Hey - I'm busy looking at: 4
Hey - I'm busy looking at: 8
Hey - I'm busy looking at: 16
```

Our simple loop ran $\log(32) = 5$ times.

We could create the above as a method that takes an input using a "normal" for loop:

In [18]:
```
public static void pow2(int n)
{
    // here is the 'normal' for loop
    for (int i = 1; i < n; i = i * 2 ){
        System.out.println("Hey - I'm busy looking at: " + i);
    }
}
```

In [19]:
```
// Try out the method with n = 32. How many times will it print?
pow2(32)
```

```
Hey - I'm busy looking at: 1
Hey - I'm busy looking at: 2
Hey - I'm busy looking at: 4
Hey - I'm busy looking at: 8
Hey - I'm busy looking at: 16
```

We see that if n is 32 the algorithm ran $\log(32) = 5$ times

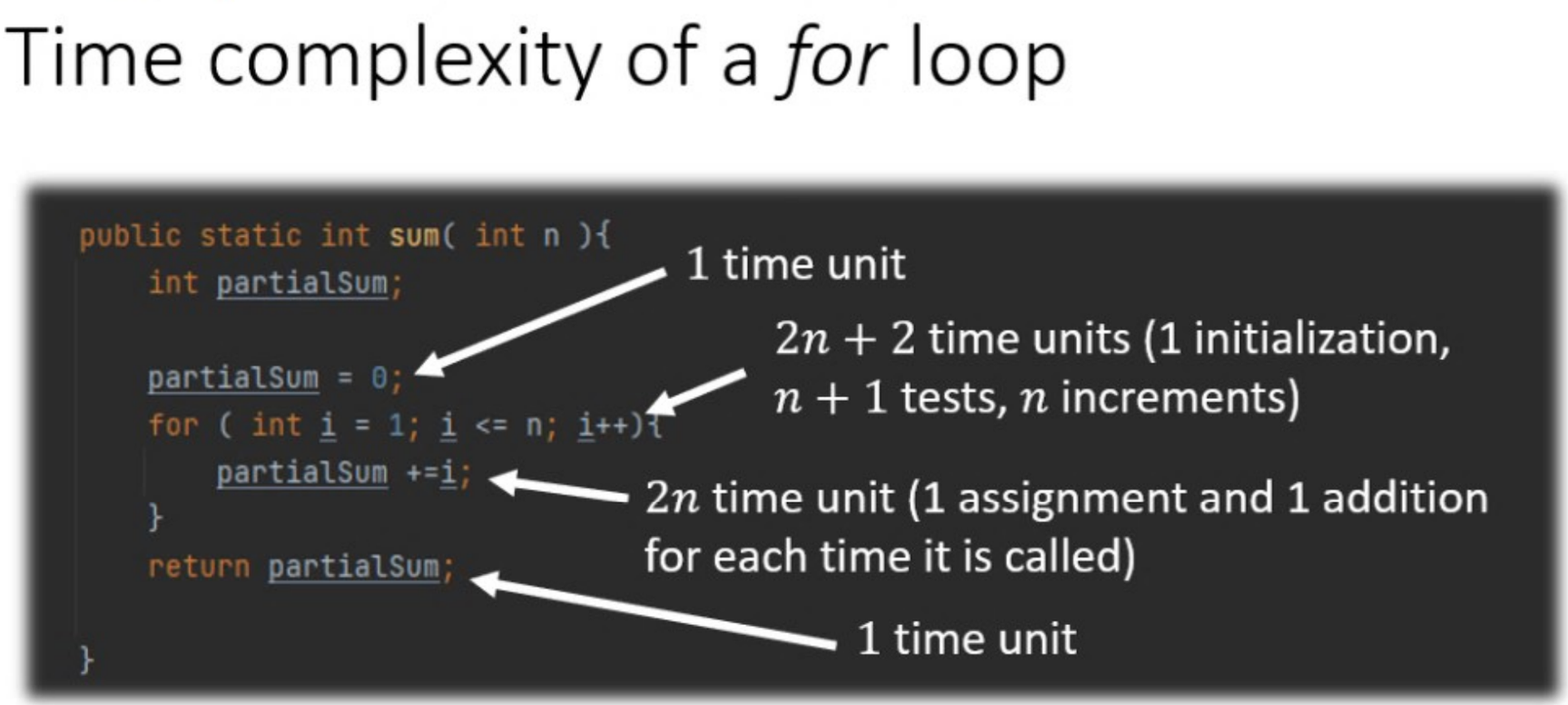
Linear Time Algorithms - $O(n)$

After logarithmic time algorithms, we get the next fastest class: linear time algorithms.

If we say something grows linearly, we mean that it grows directly proportional to the size of its inputs.

We saw an example of a linear time algorithm previously (the sum).

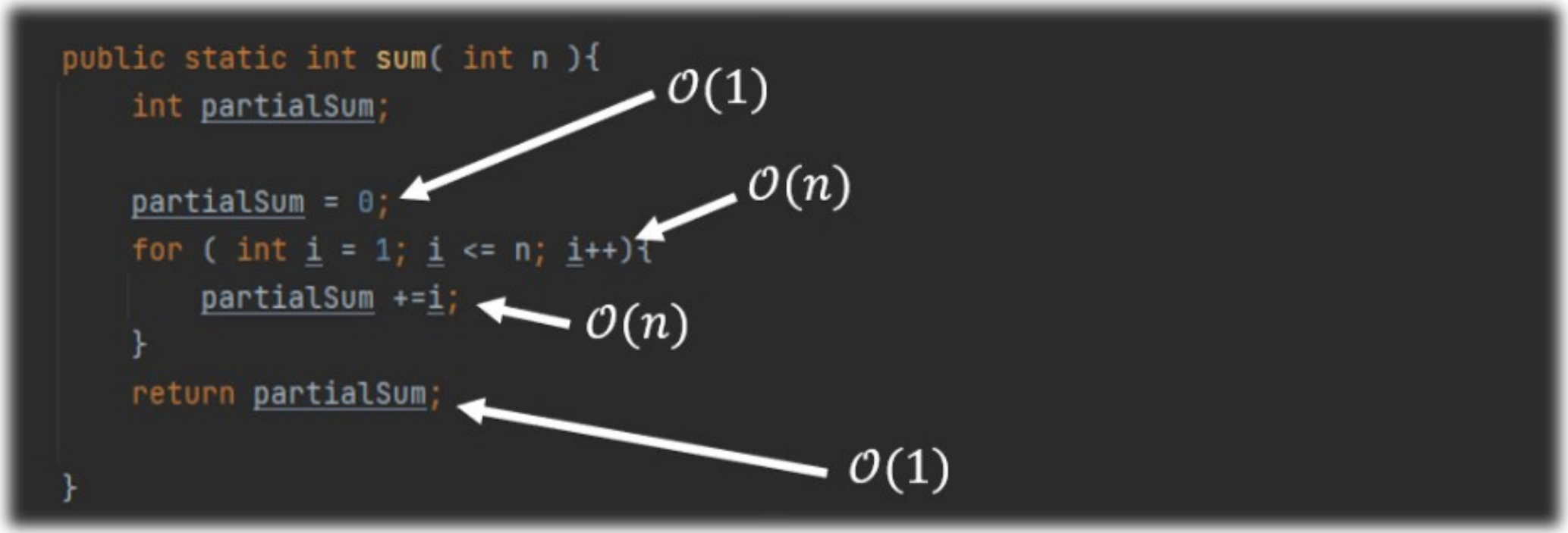
Think of a simple for loop:



Total number of time units:

$$T = 1 + 2n + 2 + 2n + 1 = 4 + 4n = \mathcal{O}(n)$$

We can simplify this analysis by considering the time complexity of each line instead



Total number of time units:

$$T = \mathcal{O}(1) + \mathcal{O}(n) + \mathcal{O}(n) + \mathcal{O}(1) = \mathcal{O}(n)$$

In [20]:
```
int n = 25;
for (int i = 0; i < n; i++) {
    System.out.println("Hey - I'm busy looking at: " + i);
}
```

```
Hey - I'm busy looking at: 0
Hey - I'm busy looking at: 1
Hey - I'm busy looking at: 2
Hey - I'm busy looking at: 3
Hey - I'm busy looking at: 4
Hey - I'm busy looking at: 5
Hey - I'm busy looking at: 6
Hey - I'm busy looking at: 7
Hey - I'm busy looking at: 8
Hey - I'm busy looking at: 9
Hey - I'm busy looking at: 10
Hey - I'm busy looking at: 11
Hey - I'm busy looking at: 12
Hey - I'm busy looking at: 13
Hey - I'm busy looking at: 14
Hey - I'm busy looking at: 15
Hey - I'm busy looking at: 16
Hey - I'm busy looking at: 17
Hey - I'm busy looking at: 18
Hey - I'm busy looking at: 19
Hey - I'm busy looking at: 20
Hey - I'm busy looking at: 21
Hey - I'm busy looking at: 22
Hey - I'm busy looking at: 23
Hey - I'm busy looking at: 24
```

How many times does this for loop run? n times, of course! We don't know exactly how long it will take for this to run – and we don't worry about that.

What we do know is that the simple algorithm presented above will grow linearly with the size of its input.

We'd prefer a run time of 0.1n than (1000n + 1000), but both are still linear algorithms; they both grow directly in proportion to the size of their inputs.

Again, if the algorithm was changed to the following:

In [21]:

```
int n = 20;
for (int i = 0; i < n; i++) {
    System.out.println("Hey - I'm busy looking at: " + i);
    System.out.println("Hmm.. Let's have another look at: " + i);
    System.out.println("And another: " + i);
}
```

```
Hey - I'm busy looking at: 0
Hmm.. Let's have another look at: 0
And another: 0
Hey - I'm busy looking at: 1
Hmm.. Let's have another look at: 1
And another: 1
```

```
Hey - I'm busy looking at: 19
Hmm.. Let's have another look at: 19
And another: 19
```

N Log N Time Algorithms – O(n log n)

$n \log n$ is the next class of algorithms. The running time grows in proportion to $n \log n$ of the input:

In [22]:
```
int n = 9;
for (int i = 1; i <= n; i++) {          //n times
    for(int j = 1; j < n; j = j * 2) {  // log n times
        System.out.println("Hey - I'm busy looking at: " + i + " and " + j);
    }
}
```

```
Hey - I'm busy looking at: 1 and 1
Hey - I'm busy looking at: 1 and 2
Hey - I'm busy looking at: 1 and 4
Hey - I'm busy looking at: 1 and 8
Hey - I'm busy looking at: 2 and 1
Hey - I'm busy looking at: 2 and 2
Hey - I'm busy looking at: 2 and 4
Hey - I'm busy looking at: 2 and 8
```

```
Hey - I'm busy looking at: 9 and 1
Hey - I'm busy looking at: 9 and 2
Hey - I'm busy looking at: 9 and 4
Hey - I'm busy looking at: 9 and 8
```

For example. if n is 8 , then this algorithm will run $8 \log(8) = 8 \times 3 = 24$ times. Whether we have strict inequality or not in the for loop is irrelevant for the sake of a Big O Notation.

Many "divide and conquer" algorithms run on $n \log n$, e.g. many sorting algorithms like merge sort. We look at "divide and conquer next week.

Polynomial Time Algorithms – O(n^p)

Next up we've got polynomial time algorithms. These algorithms are even slower than $n \log n$ algorithms.

The term polynomial is a general term which contains quadratic $\left(n^2\right)$, cubic $\left(n^3\right)$ quartic $\left(n^4\right)$, etc. functions. What's important to know is that $O\left(n^2\right)$ is faster than $O\left(n^3\right)$ which is faster than $O\left(n^4\right)$, etc.

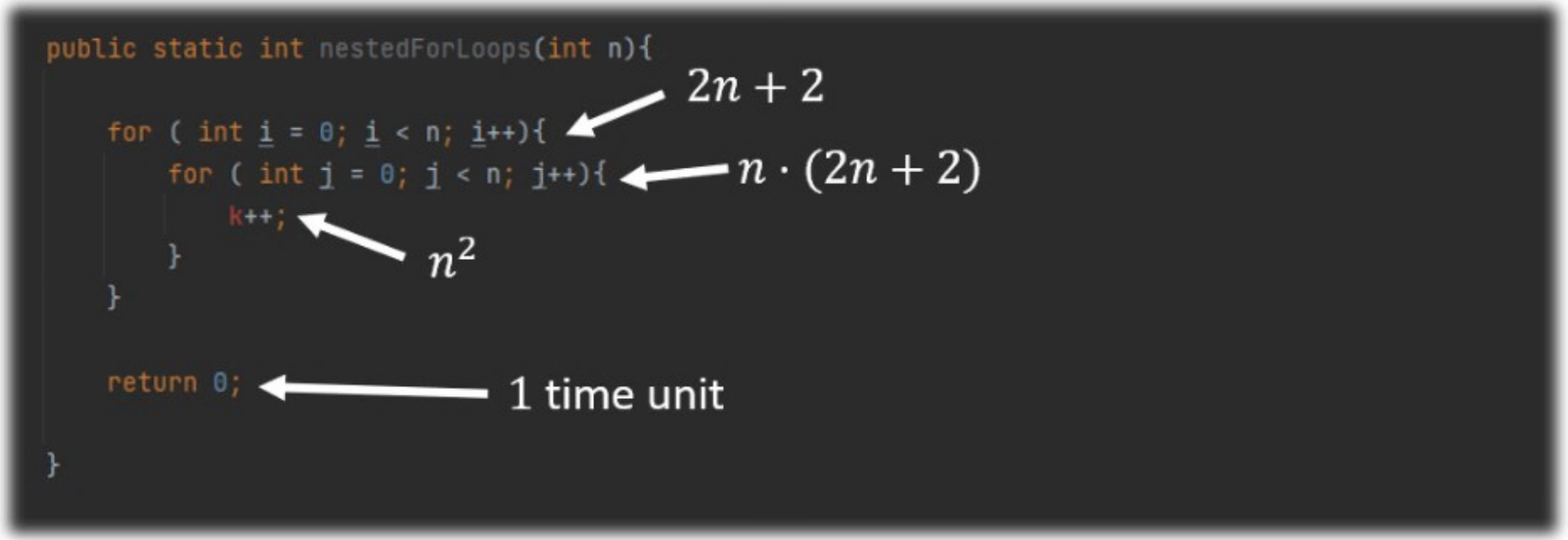
In [23]:
```
int n = 8;
for (int i = 1; i <= n; i++) {    // n times
    for(int j = 1; j <= n; j++) { // n times
        System.out.println("Hey - I'm busy looking at: " + i + " and " + j);
    }
}
```

```
Hey - I'm busy looking at: 1 and 1
Hey - I'm busy looking at: 1 and 2
Hey - I'm busy looking at: 1 and 3
Hey - I'm busy looking at: 1 and 4
Hey - I'm busy looking at: 1 and 5
Hey - I'm busy looking at: 1 and 6
Hey - I'm busy looking at: 1 and 7
Hey - I'm busy looking at: 1 and 8
Hey - I'm busy looking at: 2 and 1
Hey - I'm busy looking at: 2 and 2
Hey - I'm busy looking at: 2 and 3
Hey - I'm busy looking at: 2 and 4
Hey - I'm busy looking at: 2 and 5
Hey - I'm busy looking at: 2 and 6
Hey - I'm busy looking at: 2 and 7
Hey - I'm busy looking at: 2 and 8
```

```
Hey - I'm busy looking at: 8 and 1
Hey - I'm busy looking at: 8 and 2
Hey - I'm busy looking at: 8 and 3
Hey - I'm busy looking at: 8 and 4
Hey - I'm busy looking at: 8 and 5
Hey - I'm busy looking at: 8 and 6
Hey - I'm busy looking at: 8 and 7
Hey - I'm busy looking at: 8 and 8
```

This algorithm will run $8^2 = 64$ times. Note, if we were to nest another for loop. this would become an $O\left(n^3\right)$ algorithm.

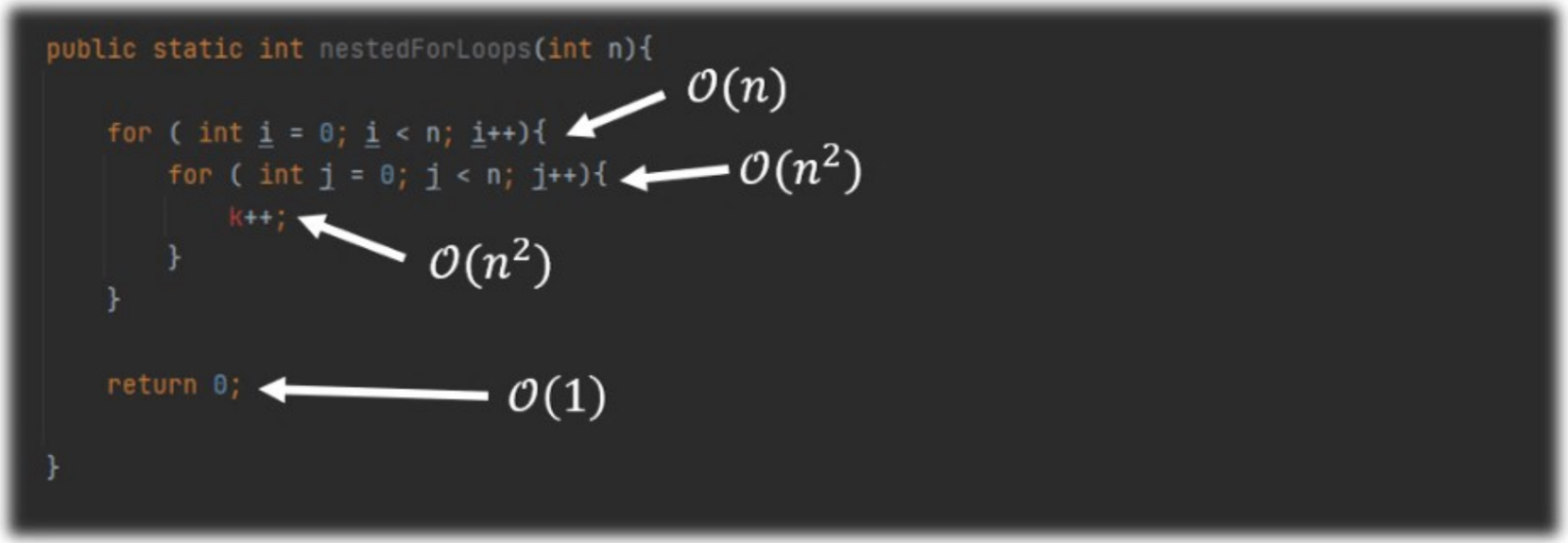
Time complexity of *nested for* loops



Total number of time units:

$$T = 2n + 2 + n \cdot (2n + 2) + n^2 + 1 = 3n^2 + 4n + 3 = \mathcal{O}(n^2)$$

… in terms of time complexities of each line:



Total:

$$T = \mathcal{O}(n) + \mathcal{O}(n^2) + \mathcal{O}(n^2) + \mathcal{O}(1) = \mathcal{O}(n^2)$$

Exponential Time Algorithms $-O\left(k^n\right)$

Now we are getting into dangerous territory; these algorithms grow in proportion to some factor exponentiated by the input size.

For example, $O\left(2^n\right)$ algorithms double with every additional input.

- So, if $n = 2$, these algorithms will run four times
- if $n = 3$, they will run eight times (kind of like the opposite of logarithmic time algorithms).

$O\left(3^n\right)$ algorithms triple with every additional input

$O\left(k^n\right)$ algorithms will get k times bigger with every additional input.

Let's have a look at a simple example of an $O\left(2^n\right)$ time algorithm:

In [24]:

```
int n = 8;
for (int i = 1; i <= Math.pow(2, n); i++){
    System.out.println("Hey - I'm busy looking at: " + i);
}

// What if n = 8?
```

```
Hey - I'm busy looking at: 1
Hey - I'm busy looking at: 2
Hey - I'm busy looking at: 3
Hey - I'm busy looking at: 4
Hey - I'm busy looking at: 5
Hey - I'm busy looking at: 6
Hey - I'm busy looking at: 7
Hey - I'm busy looking at: 8
Hey - I'm busy looking at: 9
Hey - I'm busy looking at: 10
```

```
Hey - I'm busy looking at: 250
Hey - I'm busy looking at: 251
Hey - I'm busy looking at: 252
Hey - I'm busy looking at: 253
Hey - I'm busy looking at: 254
Hey - I'm busy looking at: 255
Hey - I'm busy looking at: 256
```

Factorial Time Algorithms - O(n!)

In most cases, this is pretty much as bad as it'll get. This class of algorithms has a run time proportional to the factorial of the input size.

A classic example of this is solving the traveling salesman problem using a brute-force approach to solve it.

An explanation of the solution to the traveling salesman problem is beyond this course but you will deal with it on your 6th semester in ADS.

Take a look at this instead:

In [25]:
```
//There is no built in factorial method, so we write one
public long factorial(int n) {
    long fact = 1;
    for (int i = 2; i <= n; i++) {
        fact = fact * i;
    }
    return fact;
}

int n = 5;
for (int i = 1; i <= factorial(n); i++){
    System.out.println("Hey - I'm busy looking at: " + i);
}
```

```
Hey - I'm busy looking at: 1
Hey - I'm busy looking at: 2
Hey - I'm busy looking at: 3
Hey - I'm busy looking at: 4
Hey - I'm busy looking at: 5
Hey - I'm busy looking at: 6
Hey - I'm busy looking at: 7
Hey - I'm busy looking at: 8
Hey - I'm busy looking at: 9
Hey - I'm busy looking at: 10
```

```
Hey - I'm busy looking at: 115
Hey - I'm busy looking at: 116
Hey - I'm busy looking at: 117
Hey - I'm busy looking at: 118
Hey - I'm busy looking at: 119
Hey - I'm busy looking at: 120
```

If $n = 8$, this algorithm will run $8! = 40320$ times.

Exercise 3

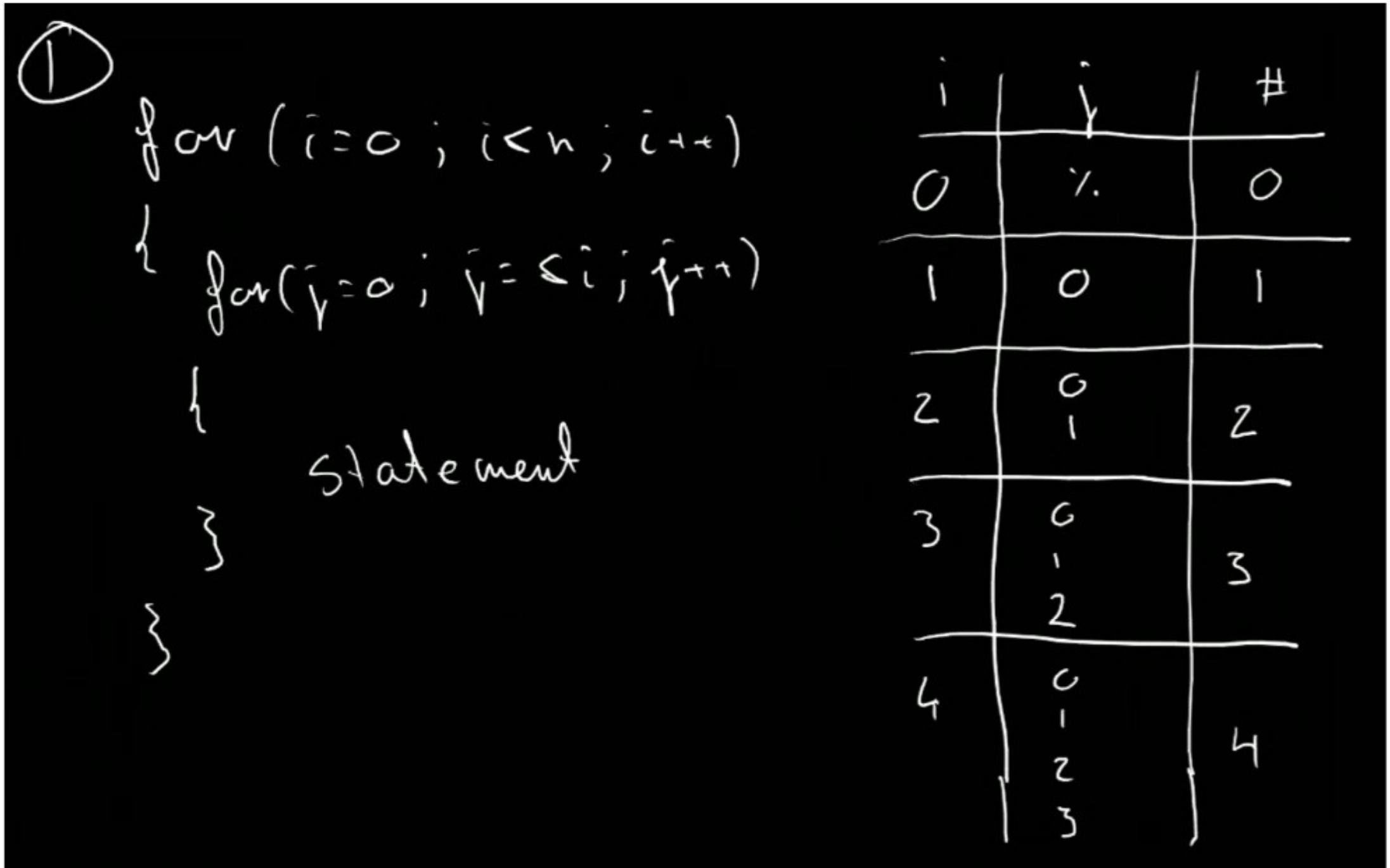
In [26]:
```java
// Algorithm1 - Iterative
public static int power1(int x, int n)
{
    // initialize result by 1
    int pow = 1;

    // multiply x exactly n times
    for (int i = 0; i < n; i++) {
        pow = pow * x;
        System.out.println("Hey - I'm busy looking at: " + (i+1));
    }
    return pow;
}
// System.out.println("Hey - I'm busy looking at: " + (i+1));
```

In [27]:
```java
power1(2,8)
```

```
Hey - I'm busy looking at: 1
Hey - I'm busy looking at: 2
Hey - I'm busy looking at: 3
Hey - I'm busy looking at: 4
Hey - I'm busy looking at: 5
Hey - I'm busy looking at: 6
Hey - I'm busy looking at: 7
Hey - I'm busy looking at: 8
```

Out[27]: 256

Analysing Algorithms



(2)

```
p = 0;
for (i = 1 ; p <= n ; i++)
{
    p = p + i;
}
```

i	p
1	0+1 = 1
2	1+2 = 3
3	1+2+3 = 6
4	1+2+3+4 = 10
⋮	⋮
K	1+2+...+K

Assume $p > n \rightarrow$ stops

$$p = \frac{K(K+1)}{2}$$

$$\frac{K(K+1)}{2} > n$$

$$K^2 > n \Rightarrow K > \sqrt{n} \rightarrow O(\sqrt{n})$$

(3)

```
for (i=1; i<n; i= i·2)
{
    statement;
}
```

i
1
$1 \times 2 = 2$
$2 \times 2 = 2^2$
$2^2 \times 2 = 2^3$
$2^3 \times 2 = 2^4$
⋮
2^k

$i >= n$ → steps

$i = 2^k$

$2^k >= n$

$k = \log_2 n \rightarrow O(\log_2 n)$

OR:

$i = 1 \times 2 \times 2 \times 2 \cdots = n$

$2^k = n$

$k = \log_2 n$

(4)

```
for(i=n; i>=1, i=i/2)
{
    statement
}
```

Stops $i<1$

$\frac{n}{2^k} < 1$

$\frac{n}{2^k} = 1$

$k = \log_2 n \rightarrow O(\log_2 n)$

i
n
$\frac{n}{2}$
$\frac{n}{2^2}$
$\frac{n}{2^3}$
$\frac{n}{2^k}$

(5)

```
for (i=0; i*i<n; i++)
{
    statement
}
```

Runs while $i*i<n$

terminates $i*i>=n$

$i^2 >= n$

$i = \sqrt{n}$

$O(\sqrt{n})$

for (i = 0; i < n; i++) — $O(n)$

for (i = 0; i < n; i+2) — $\frac{n}{2}\, O(n)$

for (i = n; i > 1 i--) — $O(n)$

for (i = 1; i < n; i = i*2) — $O(\log_2 n)$

for (i = 1; i < n; i = i*3) — $O(\log_3 n)$

for (i = n; i > 1; i = i/2) — $O((\log_2 n)$