06/12/25 - Video 1.3 Continued

Algorithm Swap(a, b)

```
{
  temp = a;   ——— 1
  a = b;      ——— 1
  b = temp;   ——— 1
}
```

P(n) = 3

Time: every simple statement in an algorithm takes one unit of time. Simple refers to a statement, not nested, just one simple line.

Space: to find space here, we count the number of variables that we have, and that becomes our answer.

* So if we find our time or space complexity to be a constant value such as 3, 30, 300, 9000, etc, we will always officially represent it as O(1).

* When dealing with time, in reality, we denote it using big-O notation, O(1), O(n)

temp, a, and b are variables in our example, thus S(n) = 3

And now b/c both of our complexities, time and space are constant: we can represent it officially as O(1).

06/14/25 - Video 1.4 - Frequency Count Method

```
Algorithm Sum(A,n)
{
  s=0
  for(i=0; i<n; i++)
    {
      s = s+A[i]
    }
  return s
}
```

This is an algorithm to compute sum of all elements in an array

A | 8 | 4 | 1 | 2 | 3 |

n = 5 (this refers to the length of the array)

Now we want to find the time complexity? But how do we do that? We use the frequency count method.

↳ If a statement is repeated, then we use the FCM. Finding the frequency finds the time

$s=0$ → 1

$i=0$ → 1

$i<n$ → $n+1$

$i++$ → n

$2n+2$

$2(n+1)$

just $(n+1)$ and we ignore constants, so this is

$s=s+A[i]$ → now b/c this is the statement inside of the for loop, it will repeat for n times b/c the loop is also repeating for n times. It's usually n too.

return s → 1

combining: $f(n) = 2n+3$ → $O(n)$ final answer

now we want to find the space complexity. we do this by first counting the variables we have: A, n, s, i.

A → n

n → 1

s → 1

i → 1

$n+3$

$O(n)$

06/14/25 - Video 1.4 Continued

```
Algorithm Add(A,B,n)
{
  For(i=0; i<n; i++)          → n+1
  {
    For(j=0; j<n; j++)        → n×(n+1)
    {
      C[i,j] = A[i,j] + B[i,j] → n×n
    }
  }
}
```

We now want to find the time complexity for this algorithm.

↳ we have one main for loop, we know that's n+1. inside it, is a statement, so that's n. BUT, that statement is a for loop, so it's n×(n+1). And inside that is a statement, so it's n. but remember everything inside a for loop is n, so we need to accumulate for everything

n+1 ✓ → $n+1$

n×(n+1) ✓ → n^2+n

n×n ✓ → n^2

$2n^2+2n+1$ → $O(n^2)$

We now want to find the space complexity.

↳ Collect all of the variables we're using so far

A → n^2

B → n^2

n → 1

i → 1

j → 1

C → n^2

$3n^2+3$

$O(n^2)$

6/15/25 - Video 1.4 Continued

```
Algorithm Multiply(A, B, n)
{
  for(i=0; i<n; i++) ──→ n+1
  {
    for (j=0; j<n; j++) ──→ n × (n+1)
    {
      c[i,j] = 0 ──→ n × n
      for (k=0; k<n; k++) ──→ n × n × (n+1)
      {
        c[i,j] = c[i,j] + A[i,k] * B[k,j] ──→ n × n × n
      }
    }
  }
}
```

Time Complexity

- we have one main for loop, so that's n+1 and everything inside is n
- we have a second for loop, which is n+1 and everything inside is n again
- we have a third for loop, so then we'll do the same things

- $n+1$
- n^2+n
- n^2
- n^3+n^2
- n^3

$2n^3 + 3n^2 + 2n + 1$

$O(n^3)$

Space Complexity

- we have A, B, n, i, j, C, k

A, B, C } n^2

i, j, k, n } 1

$n^2 + 4$

$O(n^2)$

6/15/25 - Video 1.5.1 - Time Complexity #1

```
for( i=0; i<n; i++)      → n+1
  code statement         → n
```

2n + 1 → $O(n)$

NOTE: we don't have to add them up, we just want to know about the statement, that's the most important thing there.

WE CARE ABOUT THE DEGREE!!

how whether we are incrementing or decrementing within our for loop, it'll still take the same number of steps, thus $O(n)$ still.

```
for (i=0; i<n; i++)          → n+1
{
  for (j=0; j<n; j++)        → n × (n+1)
  {
    code statement           → n × n
  }
}
```

remember, we only care about the statements inside, not the actual loop declaration, so it's n×n, or n^2

$O(n^2)$

```
                  n+1
for (i=0; i<n; i++)
{                 n×(n+1)
  for (j=0; j<i; j++)
  {
    statement     → n × n
  }
}
```

This is a special type of problem b/c, the inside loop has execution j<i. So we need to be careful

The statement will execute for n number of times. Assume it's going till n so

$1 + 2 + 3 + \dots n = \frac{n(n+1)}{2} \rightarrow \frac{n^2+n}{2}$

$O(n^2)$

*if we did it the wrong way we also still get $O(n^2)$

i	j	# of times executed
0	0 x	0
1	0 ✓	1
	1 x	
2	0 ✓	2
	1 ✓	
	2 x	
⋮	⋮	⋮
		n

6/15/25- Video 1.5.1 Continued

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

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

repeats for k times

$1+2+3+4+\ldots+k$ → same as $\frac{k(k+1)}{2}$

k^2

But we can't just say $O(k^2)$ b/c we deal with n and, we have a condition.

Assume $p > n$. we do this b/c that's the stopping condition.

$p = \frac{k(k+1)}{2} \rightarrow k^2 > n$

$k = \sqrt{n}$

so

$O(\sqrt{n})$

each time, we're adding bigger and bigger numbers to p. And p also becomes a lot bigger.

★★★★★

And we care about when the loop stops b/c when dealing with time complexity, we always ask ourselves "As n gets huge, how long will this code run?" Time complexity measures how many times the code runs as the input gets larger. Stopping condition is key.

06/15/25 - Video 1.5.2 - Time Complexity #2

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

This isn't our normal for loop b/c we aren't regularly incr/decr b/c we have $i = i * 2$.

Let's trace it

$i = 1$ then $i = 2$ then $i = 4$ then $i = 8$ then.....

↳ b/c everything is related to 2, we should simplify this and write it as 2^k times. but remember, we write our TC's in terms of n. So what's our stopping condition here? $i > n$ is when we stop.

$2^k >= n$

$k = \log_2 n$

so time complexity is $O(\log_2 n)$

NOTE: if the counter is multiplied, the time complexity will deal with log here.

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

here, i starts at n and then it becomes $n/2$ and then it becomes $n/2^2$ and then $n/2^3$ ... the value of the exponent keeps rising and we don't know what it is. So $n/2^k$.

the stopping condition here is when $i < 1$.

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

$k = \log_2 n$

Time complexity is again $O(\log_2 n)$

6/15/25- Video 1.5.2 Continued

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

So we don't actually need to do the gruesome long work here because of our condition. $i * i < n$. Our stopping condition will be

$i \times i >= n$

$i^2 >= n$

$i^2 = n$

$i = \sqrt{n}$

Time Complexity is $O(\sqrt{n})$

If we have two regular for loops one after the other, then we take the complexity of each, which will just be: n, and then combine so its 2n. Thus $O(n)$

```
p=0
for(i=1; i<n; i=i*2)
{
  p++
}
for(j=1; j<p; j=j*2)
{
  statement
}
```

this is also log b/c of the multiplication. so it's log n here.

we've seen this before, this is a common log one b/c we're multiplying this is $\log p$ → need to represent in n.

$p = \log n$ so

Time Complexity is $O(\log \log n)$

06/15/25– Video 1.3.2 Continued

Cheatsheet: for loops only

- `for(i=0; i<n; i++)` → $O(n)$
- `for(i=0; i<n; i=i+2)` → $O(n)$
- `for(i=0; i>n; i--)` → $O(n)$
- `for(i=1; i<n; i=i*2)` → $O(\log_2 n)$
- `for(i=n; i>1; i=i/2)` → $O(\log_2 n)$
- `for(i=1; i<n; i=i*3)` → $O(\log_3 n)$

- regular incr/decr is $O(n)$
- if mult/division, is $O(\log ? n)$