These notes serve as a tl;dr for the motivation and theoretical concepts necessary for algorithm analysis. Your main takeaway should be conceptual understanding---no need to take detailed notes or attempt to memorize key details. Just read it, and if you feel like you generally understand what's going on, you're good. You can always return to a particular section again later.

Part 1: Operation Counting

An algorithm is more efficient than another if it solves **the same problem in less time**. But how do we measure the efficiency of our code? Is there a better solution than just literally implementing it and timing it?

- The running time of an algorithm is directly related to how much **"stuff"** it does
    - This is **not necessarily** related to code length.
    - There exists short code which does a lot of stuff (because of loops, functions, recursion, etc.)
- Let the smallest unit of "stuff" be **1 operation**. The actions that cost "1 operation" are called **primitive**.
    - All complex code can be eventually broken down into its primitive operations.
    - The *defining feature* of a primitive operation is that it's so basic that it *can't* be broken down further.
        - Variable assignment
        - Arithmetic operations
        - Boolean operations
        - If statement checks
    - Examples of primitive operations:
        - `x = 5`
        - `1 + 2`
        - `a or b`
        - `if condition:`
    - Examples of **non-primitive** operations:
        - `sum(arr)`
        - `max(arr)`
        - `range(n)`

- The code's efficiency is found by **counting** the number of primitive operations it performs.
    - The operation count is typically a *function* of the input.
    - In other words, programs typically do a different amount of stuff depending on what the input is.

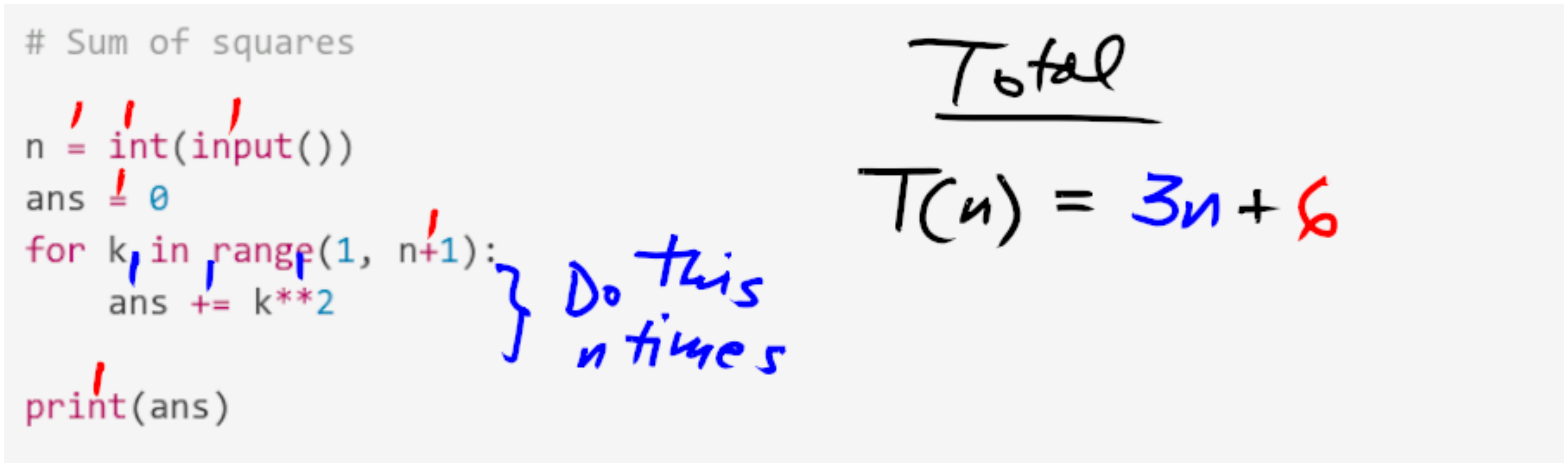
Example: Let's count the amount of stuff done in the following code snippet.

```python
# Sum of squares

n = int(input())
ans = 0
for k in range(1, n+1):
    ans += k**2

print(ans)
```

Here's my attempt at counting the amount of stuff this code does, as a function of n. This isn't a precise science, so don't be surprised if we get slightly different answers. It actually doesn't matter.



```python
# Sum of squares

n = int(input())
ans = 0
for k in range(1, n+1):
    ans += k**2

print(ans)
```

It would be sensible to say that its running time is $T(n) = 3n + 6$.

Part 2: Converting operations $\to$ seconds

But I don't want an operation count, I want to know how much literal actual time it takes for my program to run!

- Operation counting is nice because it relies only on the design of the algorithm.
    - The actual running time of an algorithm is dependent on many factors, such as:
        - The programming language used
            - C++ is faster than Python
        - The quality of the hardware
            - Supercomputer vs gaming PC vs potato PC vs Nintendo DS
- In 2021: Assume that computers can do $\approx 10^9$ operations per second (that's why you hear **giga**hertz with modern processors).
    - Because of the other factors mentioned previously, again, this isn't precise. It's just a rough estimate.
    - These operations are at very low-level machine code.
        - There is an additional "tax" when converting from a high-level programming language.
- The following **orders of magnitude** are your magic numbers.
    - 10^9 low-level operations can be done per second.
    - If in **Python**, if your code does order-of-magnitude $\approx 10^7$ ish operations per second, it will probably terminate in less than a second.
        - If sagad, submit to PyPy (which is faster than Python for magic reasons).
    - If in **C++**, if your code does order-of-magnitude $\approx 10^8$ ish operations per second, it will probably terminate in less than a second.

Will My Code TLE?

1. Get the operation count, as a function of the input parameters
2. Plug the worst case values into the function
3. Compare this to your language's magic number.

Example: Consider the sum of squares function from earlier, with a running time of $T(n) = 3n + 6$.

- Suppose $1 \leq n \leq 10^5$. Then, the amount of operations performed is at most $T(10^5) < 10^7$, so it probably passes.
- Suppose $1 \leq n \leq 10^{18}$. Then, the amount of operations performed is at most $T(10^{18}) >>> 10^7$, so it definitely will not pass.

Part 3a: Asymptotic Analysis in Theory

No one does operation counting

1. It's boring.
2. It's tedious.
3. It's needlessly precise, when we lose most of that precision anyway because of hardware and implementation details and many other things outside our control.

Solution: We don't need the exact running time function. It's okay if we just *approximate* it. And this makes our lives way easier.

Big Θ Notation

- Let $\Theta(f(n))$ be a **family** of functions that all **grow at the same rate** as $f(n)$.
  - **Examples of Families:**
    - Linear functions $\Theta(n)$
      - n
      - $n - 7$
      - $3n - 2$
    - Quadratic functions $\Theta(n^2)$
      - n^2
      - $n^2 + 2n + 1$
      - $2n^2 - 7$
    - Square root functions $\Theta(\sqrt{n})$
      - $\sqrt{n}$
      - $\sqrt{2n+1}$
      - $3\sqrt{n} + 4$

        - Constant functions $\Theta(1)$
            - 1
            - 5
            - 30
    - How to characterize different families?
        - If you plug in a very large value of n, functions in the same family will have the same order of magnitude.
            - If you plug in $n = 10^6$...
                - All the functions in $\Theta(n)$ are $\approx 10^6$ ish.
                - All the functions in $\Theta(n^2)$ are $\approx 10^{12}$ ish.
                - All the functions in $\Theta(\sqrt{n})$ are $\approx 10^3$ ish.
            - Again, I emphasize that these statements are made about general orders of magnitude.
        - Notice that if n is quite large, then constant factors typically don't affect the order of magnitude that much.

- Any member of the family can be the representative $f(n)$, so typically we describe families by their ''simplest'' element

- **How to determine which family a function belongs to**

    - Keep only the dominating term. Or, drop all lower-order terms.
        - $n^2 + n + 7 \in \Theta(n^2)$
        - $n + \sqrt{n} \in \Theta(n)$
    - Drop constant-factor coefficients.
        - $2n^3 \in \Theta(n^3)$
        - $\frac{1}{2}n^2 \in \Theta(n^2)$
    - Most functions you encounter in practice can be ''simplified'' with these two rules
        - $5n^2 - 10n + 7 \in \Theta(n^2)$
        - $2(n-1) + \sqrt{n+1} \in \Theta(n)$
        - $\frac{n(n+1)(2n+1)}{6} \in \Theta(n^3)$
    - *Essentially*, we are saying that lower-order terms and constant-factor coefficients can be safely ignored without changing the order of magnitude of the answer of our function (for large inputs).
    - Because we only care about the general behavior of the function for very large inputs, we call this **asymptotic analysis**

- **Insight:** We only need to figure out *what family* the running time function is in. Then, we can use that simplified version of the function when comparing against the magic number of operations.
    - **Examples**
        - In the sum of square example, the running time was $T(n) = 3n + 6 \in \Theta(n)$.
            - If $n = 10^5$, then this will probably terminate within a second, because $10^5 < 10^7$.
            - If $n = 10^{18}$, then this will probably **not** terminate within a second, because $10^{18} >>> 10^7$.
        - Consider the following code.

```
ans = 0
for i in range(1, n+1):
    for j in range(1, n+1):
        ans += i * j

print(ans)
```

- We claim that it would make sense for its running time to be $3 + n(1 + 3n)$. But, we can massively simplify this by saying that $3 + n(1 + 3n) \in \Theta(n^2)$.
    - If $n = 10^3$, then this will probably terminate within a second, because $(10^3)^2 < 10^7$.
    - If $n = 10^5$, then this will probably **not** terminate within a second, because $(10^5)^2 > 10^7$.
- Here are some common families of time complexities, and values of n for which an algorithm with this running time would safely finish within 1 second (**these values are not the tightest they can be**).

Family	Safe Upper Bound	Special Name?
$\Theta(1)$	Any	Constant
$\Theta(n)$	10^7	Linear
$\Theta(n^2)$	2000	Quadratic
$\Theta(n^3)$	200	Cubic

Family	Safe Upper Bound	Special Name?
$\Theta(\sqrt{n})$	10^{12}	
$\Theta(n \log_2 n)$	2×10^5	
$\Theta(\log_2 n)$	10^{18}	Logarithmic
$\Theta(2^n)$	20	Exponential

Big $\mathcal{O}$ Notation

- We have another convenient family of sets. Let $\mathcal{O}(f(n))$ be a family of functions that all grow at the same rate as $f(n)$ **or slower**.
    - Θ is to $=$, as $\mathcal{O}$ is to $\leq$
    - Uses the same rules as Θ
    - **Examples**
        - $2n^2 \in \mathcal{O}(n^2)$
        - $n + 5 \in \mathcal{O}(n^2)$
        - $n^3 \notin \mathcal{O}(n^2)$
- Used in cases when $\leq$ is easier to show than $=$
    - Or, because $\leq$ the magic number is sufficient, some people just use $\mathcal{O}$ all the time
    - We'll see some concrete examples when we see how this is used in theory!

To summarize: Will My Code TLE? (Easier Version)

Here is our current process.

1. **Estimate the family** of the running time, picking a very simple function to make our lives easier.
2. Plug the worst case values into the function
3. Compare this to your language's magic number.

But this leaves us with one final question---how the heck do we find the *family* of the running time, without going through the tedious process of getting the actual running time? It turns out that it's quite easy! But it's best shown through examples and demonstration, rather than through words.