These notes serve as a tl;dr for using Big $\mathcal{O}$ Notation "in practice" to get the running time of programs. This is the one you should grapple with and study closely. But at the same time, don't worry about it too much---you'll naturally attain fluency with this topic as you use it more. This example spam should give you a ''feel'' for how analysis is done in practice. And believe me, it is very much by feel!

Part 3b: Asymptotic Analysis in Practice

Constant factors can typically be ignored.

- Computers are so fast that there is (usually) functionally no difference between performing 1 operation or 5 operations, when both can be done in a fraction of a millisecond anyway.

Example 1

```
# Sum of squares

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

print(ans)
```

Remember this problem from the first part? And how hassle it was to do operation counting for? Here's how we get its running time.

- We have a loop that runs n times.
- The loop only performs a constant number of operations per iteration.
- Thus, this runs in $\mathcal{O}(n)$.

It's that simple!

Example 2

```
n = int(input())
a = [int(x) for x in input().split()]  # size n

minnie = a[0]
for i in range(n):
    minnie = min(minnie, a[i])

maxi = a[0]
for i in range(n):
    maxi = min(maxi, a[i])

print(maxi - minnie)
```

- We have two loops that run n times, each performing a constant number of operations per iteration.
- Thus, this runs in $\mathcal{O}(2n)$, but that's just $\mathcal{O}(n)$.

Example 3

```
n = int(input())

ans = 0
for a in range(1, n+1):
    for b in range(1, n+1):
        if a % 2 == b % 2:
            ans += a * b

print(ans)
```

- We have a loop that runs n times
- In each of its iterations, we have a loop that runs n times.
- The inner loop only performs a constant number of operations.
- Thus, this runs in $\mathcal{O}(n \times n)$, or $\mathcal{O}(n^2)$.

Example 4

```
n = int(input())
a = [int(x) for x in input().split()]  # size n

ans = 0
for i in range(n):
    for j in range(n):
        for k in range(n):
            ans += a[i] * a[j] * a[k]

print(ans)
```

- We have three nested for loops, which go over all $n \times n \times n$ triples of indices.
- We do a constant amount of work per triple.
- The running time is $\mathcal{O}(n^3)$.
-

Example 5

```
n, m = int(input())
a = [int(x) for x in input().split()]  # size n
b = [int(x) for x in input().split()]  # size m

ans = 0
for i in range(n):
    for j in range(m):
        ans += a[i] * b[j]

print(ans)
```

- We have two nested for loops, which go over all $n \times m$ pairs of indices.

- We do a constant amount of work per pair.
- The running time is $\mathcal{O}(nm)$.

Example 6

```
n = int(input())
a = [int(x) for x in input().split()]  # size n

for i in range(n+1):
    left_half = 0
    for j in range(i):
        left_half += a[j]

    right_half = 0
    for j in range(i, n):
        right_half += a[j]

    print(abs(left_half - right_half))
```

- We have a loop that runs n times.
- In the i th iteration, we have a loop that goes from 0 to $i-1$, then another loop that goes from i to $n-1$
    - Both inner loops only do a constant number of operations
    - Thus, these two inner loops do $\mathcal{O}(n)$ work in total
- Since each of the n iterations of the outer loop performs $\mathcal{O}(n)$ work, the running time is $\mathcal{O}(n^2)$.

Example 7

```
n = int(input())
a = [int(x) for x in input().split()]  # size n

for i in range(n):
    less = 0
    for j in range(n):
```

```
        if a[j] < a[i]:
            less += 1

    greater = 0
    for j in range(n):
        if a[j] > a[i]:
            greater += 1

    print(f'{i}th element: There are {less} other elements less than it, and {greater} other elements greater than it')
```

- We have a loop that runs n times.
- In each iteration, we have two loops that each perform $\mathcal{O}(n)$ work, which is still $\mathcal{O}(n)$ when put together.
- Thus, the running time is $\mathcal{O}(n^2)$.

Example 8

```
n = int(input())
print(n*(n+1)*(2*n+1)//6)
```

- The amount of work done does not depend on n.
- The running time is $\mathcal{O}(1)$.

Example 9

```
n = int(input())

ans = 0

ans += n
ans -= n//2
ans -= n//3
ans -= n//5
```

```
ans -= n//7
ans += n//6
ans += n//10
ans += n//14
ans += n//15
ans += n//21
ans += n//35
ans -= n//30
ans -= n//42
ans -= n//70
ans -= n//105
ans += n//210

print(ans)
```

- The amount of work done does not depend on n.
- The running time is $\mathcal{O}(1)$.

Keep only the dominating term.

- We only care about order of magnitude, and pretty much only the dominating term matters in determining this.

Example 10

```
n = int(input())
a = [int(x) for x in input().split()]  # size n

ans = 0
for i in range(n):
    for j in range(n):
        for k in range(n):
            ans += max(a[i], a[j], a[k])

for i in range(n):
```

```
        for j in range(n):
            ans -= min(a[i], a[j])

print(ans)
```

- We have three nested for loops doing $\mathcal{O}(n^3)$ work, followed by two nested for loops doing $\mathcal{O}(n^2)$ work.
- Together, the total running time is $\mathcal{O}(n^3 + n^2)$, but that's just $\mathcal{O}(n^3)$.

Less Straightforward Analysis

Let's revisit **Example 6**

```
n = int(input())
a = [int(x) for x in input().split()]  # size n

for i in range(n):
    left_half = 0
    for j in range(i):
        left_half += a[j]

    right_half = 0
    for j in range(i, n):
        right_half += a[j]

    print(abs(left_half - right_half))
```

Now, let's suppose we tried to optimize this code as follows.

Example 11

```
n = int(input())
```

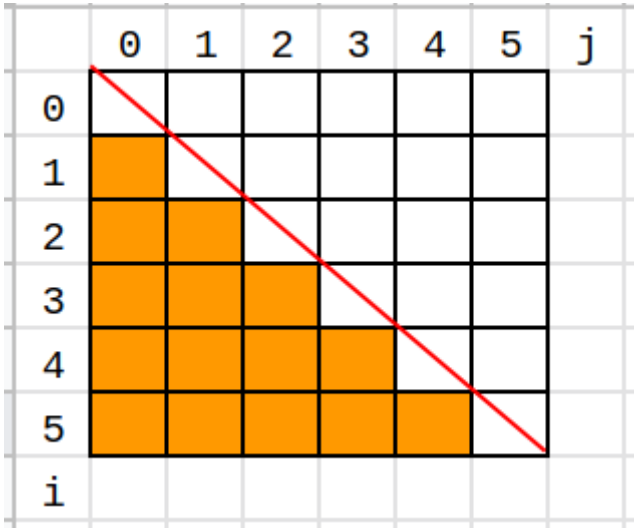
```
a = [int(x) for x in input().split()]  # size n

total = 0
for i in range(n):
    total += a[i]

for i in range(n+1):
    left_half = 0
    for j in range(i):
        left_half += a[j]
    right_half = total - left_half
    print(abs(left_half - right_half))
```

- The first loop has a running time of $\mathcal{O}(n)$.
- Let's now try to analyze the complexity of the two nested for loops that compute `left_half` and `right_half`.
    - There are n iterations of this loop.
    - The i th iteration has an inner loop that performs i additions.
    - Thus, the total amount of work is $0 + 1 + 2 + \cdots + n - 1$.
    - A well-known math formula tells us that the above sum evaluates to $\frac{n(n-1)}{2}$.
    - Thus, the running time is $\frac{n(n-1)}{2} = \frac{1}{2}n^2 - \frac{1}{2}n \in \mathcal{O}(n^2)$.
- Therefore, the runnning time for this algorithm is still $\mathcal{O}(n^2)$.
- We cut the amount of work done *by half*... but when dealing with enormous numbers, that $0.5\times$ is relatively insignificant.

By the way, here's a very nice visual argument that should intuitively show why $0 + 1 + 2 + \cdots + n - 1$ is approximately $n/2$.



- This helps us determine the **bottlenecks** in our code.
- If we want to most significantly improve our running time, we need to focus those fixes on the "heaviest" parts of the code.
    - Optimizations should aim to eliminate the dominating term. Optimizing anywhere else, while technically helpful, will likely make no meaningful difference.
- In this case, we see that the issue comes from the nested for loops. If we want a sub-quadratic solution, we need to somehow get rid of the inner for loop.
- Here is a faster solution. We will analyze its running time, but we leave proving its correctness to you.

Example 12

```
n = int(input())
a = [int(x) for x in input().split()]  # size n

total = 0
for i in range(n):
    total += a[i]

left_half = 0
for i in range(n+1):
```

```
    right_half = total - left_half
    print(abs(left_half - right_half))
    if i < n:
        left_half += a[i]
```

- We have two loops, each doing $\mathcal{O}(n)$ work.
- Thus, this solution runs in $\mathcal{O}(n)$.