Algorithm Analysis
REVIEW

Key Terms:

Algorithm – step by step procedure for solving a problem (like a recipe)
Program – the code that embodies an algorithm (like a chef that cooks a recipe)

What is algorithm analysis and why do we use it?
Many problems in computer science (and in the real world) have multiple correct solutions. Correct solutions are good, but in this class, we are looking for more than just correct. It is our job to figure out what the *BEST* correct solution/algorithm is. This is the one that is most efficient in terms of time and space. Algorithm analysis allows us to compare performances of different algorithms to determine the *BEST* one for a problem.

Algorithm Analysis Approaches:

Approach 1: **Simulation Timing** – Physically timing your algorithm using a clock

```
auto t1 = clock::now()

// code for algorithm here …

auto t2 = clock::now()

print("Time it takes algorithm to execute is…");
print(t2 – t1);
```

Easy to measure and interpret

Results vary across machines
Not predictable for small inputs

Approach 2: **Modeling/Counting** – Physically adding up how many times each expression will execute

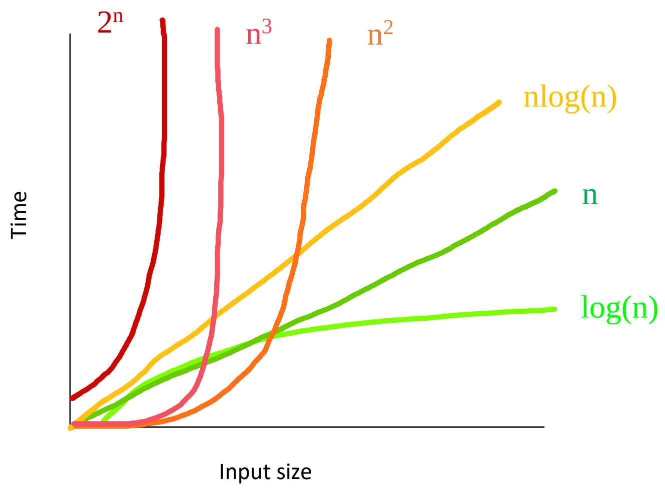
```
int sum = 0; +1

    +1      +(n+1) +n
for(int i = 0; i < n; i++)
{
        sum += i; +n
}

print(sum); +1
```

Independent of computer

Very tedious to compute

Total: 3n + 4

Approach 3: **Asymptotic Analysis**



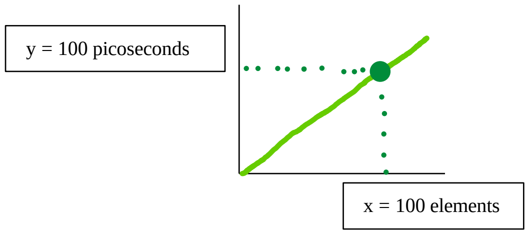
Growth rates: **MEMORIZE THIS**

$\log(n) < n < n\log(n) < n^2 < n^3 < 2^n < n!$

Asymptotic Analysis: Some things to know

- All algorithms have a **growth rate** function **T(n)** that represents the relationship between their input size (x) and their execution time (y).

  Ex] Linear search algorithm: **T(n) = n**

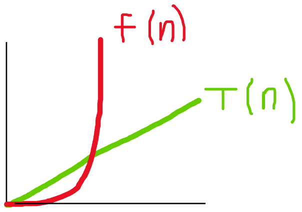


- We say that **T(n)** $\in$ **O** (**f(n)**) (an algorithm with growth rate T(n) is Big-O f(n)) IF…
  f(n) is an UPPER BOUND on the function T(n). In other words, T(n) grows *SLOWER THAN or at the SAME RATE as* f(n)

  T(n) <= **f(n)**

  O

  Ex]



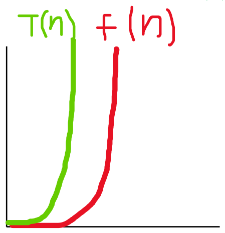
n is O(**n**^{2}) because n grows SLOWER THAN n^2
T(n) <= f(n)

- We say that **T(n)** $\in$ Ω (**f(n)**) (an algorithm with growth rate T(n) is Big- Ω f(n)) IF…
  f(n) is a LOWER BOUND on the function T(n). In other words, T(n) grows *FASTER THAN or at the SAME RATE as* f(n)

  T(n) >= **f(n)**

  Ω

  Ex]



n^3 is Ω (**n**^{2}) because n^3 grows FASTER THAN n^2
T(n) >= f(n)

CHECK FOR UNDERSTANDING:

Q: Which of these functions is $\Omega(n^5\log_2(n))$?

(a) $2n^6$
(b) n^5
(c) $n^5\log_4(n)$

A: (a) and (c) because (a) grows faster and (c) grows at the same rate

Q: So how do we determine an algorithms time complexity in terms of Big-O?

3 RULES TO REMEMBER

Non nested loops? ADD TERMS

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

}

for(int j = 0; j < m; j++)
{

}
```

$O(n + m)$

***Notice**: this CANNOT be simplified to O(n) or O(m) because we don't know if n amd m grow at the same rate or not*

Drop constant multipliers

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

}

for(int j = 0; j < n; j++)
{

}
```

$O(n + n)$
$O(2n)$
$O(n)$

Drop lower order terms

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

}

for(int j = 0; j < n; j*= 2)
{

}
```

$O(n + \log_2 n)$
$O(n)$

Try out some problems!

```
for(int i = 0; i < n; i++)
{
        for(int j = 0; j < m; j++)
        {
                print("Hi");
        }
}
```

Explanation:
Start from inside and work your way out. The print line only executes 1 time so it is O(1). In the inner loop, j starts at 0 and then keeps incrementing by a value of 1 (j++) until it reaches m. This means the whole inner loop will run a total of m times. The inner loop is thus O(m). Similarly, the outer loop will run n times so it is O(n). The loops are nested, so we multiply and get our final result.

O (n * m)

```
for(int i = 0; i < n; i++)
{
        for(int j = n; j > 0; j /= 2)
        {
                print("I like pie");
        }
}
```

Explanation:
Start inside and work your way out. Print statement is O(1). In the inner loop j starts at n, and then it *halves itself* at every iteration (j /= 2) until it reaches 0. So, first j = n, then j = n/2, then j = n/4 … etc. This pattern is characteristic of logarithmic growth, so the inner loop is $O(\log_2 n)$. The outer loop is again O(n). So, we multiply (b/c of nesting) and get

O(n*log(n))

```
for(int i = 100; i > -1; i--)
{
        for(int j = i; j > 1; j /= 2)
        {
                print("Apple pie");
        }
}
```

Explanation:
Start by examining the innermost loop. Notice that the "step" expression is **j /=2.** This indicates that we are dealing with logs. But log of what exactly…? Notice that the start expression in the inner loop is j = i. What is i? Well, we have to look at the outer loop. i = 100 initially so the inner most loop will initially run $\log_2(100)$ times. $\log_2(100) \sim 6.64$ = constant = O(1). What about in the second iteration of the outer loop when i decrements to 99? Well then again, in the inner loop j starts at 99 then goes to 99/2 then to (99/2)/2 = 99/4 … etc. This pattern tells us that on the second iteration of the outer loop, the inner loop will run $\log_2(99)$ times. $\log_2(99) \sim 6.62$ = constant = O(1). So basically, for each iteration of the outer loop, the inner loop runs a constant number of times. The outer loop runs 100 times (also O(1)). So overall we end with the result **O(1)**

IMPORTANT NOTE: So far, we have been looking only at WORST CASE time complexities. What about best and average?
Best case: minimum time required for algorithm execution
Average case: average time required for algorithm execution
Worst case: maximum time required for algorithm execution

When looking at best vs average vs worst case, input size needs to stay CONSTANT. Meaning, you **shouldn't** say "Oh best case is when the input is really small, and worst case is when the input is really big"