Time Complexity

Some Simplified Rules:

- If $f(n) = O(g(n))$ and $g(n) = O(h(n))$, then $f(n) = O(h(n))$
- If $f(n) = O(k \cdot g(n))$ for any $k > 0$, then $f(n) = O(g(n))$
- If $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$,
  then $f_1(n) + f_2(n) = O(\max(g_1(n), g_2(n)))$
- If $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$,
  then $f_1(n) \cdot f_2(n) = O(g_1(n) \cdot g_2(n))$
- $O(1) = c$, where c is a constant
- $O(n) = c \cdot n = cn$, where c is a constant and n is a variable
- $c_1 \cdot O(1) = c_1 \cdot c = c_2 = O(1)$, where c, c_1, c_2 are constants
  $-O(1) + O(1) + O(1) = 3 \cdot O(1) = O(1)$
  $-5 \cdot O(1) = O(1)$
- $n \cdot O(1) = n \cdot c = cn = O(n)$, where c is a constant and n is a variable
- $O(m) + O(n) \neq O(m + n)$
- $O(m) \cdot O(n) = c_1 \cdot m \cdot c_2 \cdot n = (c_1 \cdot c_2)(mn) = O(mn)$
- $O(m) \cdot O(n) \cdot O(p) \cdot O(q) = O(m \cdot n \cdot p \cdot q) = O(mnpq)$
- $O(an^2 + bn + c) = O(n^2)$, where a, b, c are constants

Some Series Rules:

$$1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2}$$

$$2^0 + 2^1 + 2^2 + \cdots + 2^n = 2^{n+1} - 1$$

$$\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots + \frac{1}{2^n} = 1 - \frac{1}{2^n} = 1$$

Iterative Time Complexities:

```
for (j = 0; j < n; ++j) {
    // 3 atomics
    if (condition) break;
}
```

- Upper bound = O(4n) = O(n)
- Lower bound = $\Omega(4) = \Omega(1)$
- Complexity = O(n)

```
if(condition)
    i = 0;
else
    for ( j = 0; j < n; j++)
        a[j] = j;
```

- Complexity = ??

= O(1) + max (O(1), O(N))

= O(1) + O(N)

= O(N)

```
for (j = 0; j < n; ++j) {
    // 3 atomics
}
for (j = 0; j < n; ++j) {
    // 5 atomics
}
```

- Add the complexity of consecutive statements
- Complexity = O(3n + 5n) = O(n)

```
for (j = 0; j < n; ++j) {
    // 2 atomics
    for (k = 0; k < n; ++k) {
        // 3 atomics
    }
}
```

- Complexity = O((2 + 3n)n) = $O(n^2)$

Nested Independent Loop:

sum = 0;	1
for (i = 10; i <= n; i ++)	(n - 10) / 1
for (j = 0; j <= n; j += 2)	(n - 0) / 2
sum ++;	1
	1 + (((n - 10) / 1) * ((n - 0) / 2) * 1) = $O(n^2)$

Nested Dependent Loop:

sum = 0;	1
for (i = 1; i <= n; i ++)	n^2
for (j = 1; j <= i; j ++)	(Calculation shown below)
sum ++;	1
for (i = 1; i <= n; i ++)	n
sum ++;	1
	$1 + (n^2 * 1) + (n * 1)$ $= O(n^2)$

i	j	count
1	1	1
2	1, 2	2
3	1, 2, 3	3
…	…	…
n	1, 2, 3, … , n	n
		$1 + 2 + 3 + \ldots + n$ $= (n (n+1)) / 2$ $= O(n^2)$

Nested Independent Loop with Geometric Series:

sum = 0;	1
for (i = 1; i <= n; i *= 2)	$\log_2 n$ (Calculation shown below)
for (j = 1; j <= n; j ++)	n
sum++;	1
	1 + ($\log_2 n$ * n * 1) = O(n $\log_2 n$)

step	i
0	1 (2^0)
1	2 (2^1)
2	4 (2^2)
…	…
k	n (2^k)
	$n = 2^k$ $\log_2 n = \log_2 2^k$ $k = \log_2 n$

Nested Dependent Loop with Geometric Series:

sum = 0;	1
for (i = 1; i <= n; i *= 2)	n
for (j = 1; j <= i; j ++)	(Calculation shown below)
sum++;	1
	1 + (n * 1) = O(n)

step	i	j	count
0	1 (2^0)	1	1
1	2 (2^1)	1, 2	2
2	4 (2^2)	1, 2, 3, 4	4
…	…	…	…
k	n (2^k)	1, 2, 3, 4, … , n	n
	$n = 2^k$		$1 + 2 + 4 + \dots + n$ $= 2^0 + 2^1 + 2^2 + \dots + 2^k$ $= 2^{k+1} - 1$ $= 2^k.2 - 1$ $= n.2 - 1$ $= 2n - 1$ $= O(n)$

Recursive Time Complexities:

With an iterative algorithm, we can analyze loops to get an accurate estimate of time complexity in most cases. However, this becomes much trickier if recursion is involved.

Recursive Function - A recursive function has two parts: the base case, and the recursive case. The recursive case is when the function calls itself. The base case is when the function doesn't call itself again so it doesn't go into an infinite loop.

```
def position(n):
    if (n==0):
        return 1
    else:
        return 1+ position(n-1)
```

Recursion is used when it makes the solution clearer. There's no performance benefit to using recursion; loops are sometimes better for performance.

Recurrences:

Previously, we defined the runtime of an algorithm as a function T(n) of its input size n. We will still do this when dealing with recursive functions. However, because a recursive algorithm calls itself with a smaller input size, we will write T(n) as a recurrence relation, or an equation that defines the runtime of a problem in terms of the runtime of a recursive call on smaller input.

A recurrence is an equation that describes a function in terms of its value on other, typically smaller, arguments. Recurrences go hand in hand with the divide-and-conquer method because they give us a natural way to characterize the running times of recursive algorithms mathematically.

An "algorithmic recurrence" is just a fancy way of saying there's a formula that tells you how much time your divide-and-conquer algorithm takes to solve a problem of size 'n'.

Linear Recurrence Relations:

$$T(n) = \begin{cases} \Theta(1), & \text{if } n = 0 \\ T(n-1) + \Theta(1), & \text{if } n > 0 \end{cases}$$

Here, T(n) represents the runtime of factorial(n), T(n− 1) represents the runtime of the recursive call to factorial(n - 1), and the additional Θ(1) term represents the constant work we do outside the recursive call. Since the recurrence relation will be used to calculate time complexities, having an exact number for this constant term does not matter.

```
int32_t foo(int32_t n) {
  if (n == 1) {
    return 1;
  } // if
  return foo(n - 1) + foo(n - 1) + 1;
} // foo()
```

To solve this problem, first convert the function into a recurrence relation. This recurrence relation is:

$$T(n) = \begin{cases} \Theta(1), & \text{if } n = 1 \\ 2T(n-1) + \Theta(1), & \text{if } n > 1 \end{cases}$$

From now on, we will substitute Θ(1) with the constant 1. This simplifies our math without changing the result of our analysis.

$$T(n) = \begin{cases} 1, & \text{if } n = 1 \\ 2T(n-1) + 1, & \text{if } n > 1 \end{cases}$$

To solve recurrences, there are 3 ways:

- Master Theorem Method
- Substitution Method
- Recurrence Tree Method

Master Theorem Method:

If the recurrence relation of an algorithm is of the form:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

where a ≥ 1, b > 1, and f(n) is a positive function that is $\Theta(n^c)$, then the following is true:
[Note: O() and Θ() are interchangeable]

$$T(n) = \begin{cases} \Theta\left(n^{\log_b(a)}\right), & \text{if } a > b^c \\ \Theta(n^c \log_b(n)), & \text{if } a = b^c \\ \Theta(n^c), & \text{if } a < b^c \end{cases}$$

The steps for using the Master Theorem are as follows:

1. Determine the values of a, b, and c.

2. Make sure the Master Theorem can be used on the recurrence relation.

- The coefficient of the recursive call, a, must be at least one.
- The argument of the recursive call must be divided by some number, b, that is larger than one.
- The function f(n) must be an asymptotically positive function with complexity $\Theta(n^c)$.
- A base case exists that can be solved in a constant amount of time.

3. Compare the values of a and b^c to determine which case of the Master Theorem should be used.

$$T(n) = \begin{cases} 1, & \text{if } n = 1 \\ 3T(n/2) + 5n + 13, & \text{if } n > 1 \end{cases}$$

$a = 3, b = 2, c = 1$ as $f(n) = 5n + 13 = O(n)$
$a = 3, b^c = 2$
$\Rightarrow a > b^c$
$\Rightarrow O\left(n^{\log_b a}\right)$
$\Rightarrow O\left(n^{\log_2 3}\right)$
$\therefore O\left(n^{1.58}\right)$

$$T(n) = \begin{cases} 1, & \text{if } n = 1 \\ 6T(n/2) + 5n + 2^n, & \text{if } n > 1 \end{cases}$$

$a = 6, b = 2$ as $f(n) = 5n + 2^n = O(2^n)$
Not in the form n^c, hence cannot be solved by Master Theorem

$$T(n) = \begin{cases} 1, & \text{if } n = 1 \\ 7T(10n/11) + 9, & \text{if } n > 1 \end{cases}$$

$a = 7, b = 11/10, c = 0$ as $f(n) = 9 = 9n^0 = O(n^0)$
$a = 7, b^c = 1$
$\Rightarrow a > b^c$
$\Rightarrow O(n^{\log_b a})$
$\Rightarrow O\left(n^{\log_{\frac{11}{10}} 7}\right)$
$\therefore O(n^{20.4})$

Substitution Method:

$$T(n) = 2T(n-1) + 1, T(1) = 1$$

$$\begin{aligned} T(n) &= 2\{2T(n-2) + 1\} + 1 \\ &= 4T(n-2) + 2 + 1 \\ &= 4\{2T(n-3) + 1\} + 2 + 1 \\ &= 8T(n-3) + 4 + 2 + 1 \\ &= 2^{n-1}T(n-(n-1)) + 2^{n-2} + 2^{n-3} \ldots + 4 + 2 + 1 \\ &= 2^{n-1} + 2^{n-2} + 2^{n-3} + \ldots + 4 + 2 + 1 \\ &= 2^{n+1} - 1 \\ &= O(2^n) \end{aligned}$$

$$T(n) = T(n-1) + n, T(1) = 1$$

$$\begin{aligned}
T(n) &= T(n-2) + (n-1) + n \\
&= T(n-3) + (n-2) + (n-1) + n \\
&= T(n-(n-1)) + (n-(n-1-1)) + (n-(n-1-1-1)) + \ldots + n \\
&= 1 + 2 + 3 + \cdots + n \\
&= \frac{n(n+1)}{2} \\
&= O(n^2)
\end{aligned}$$

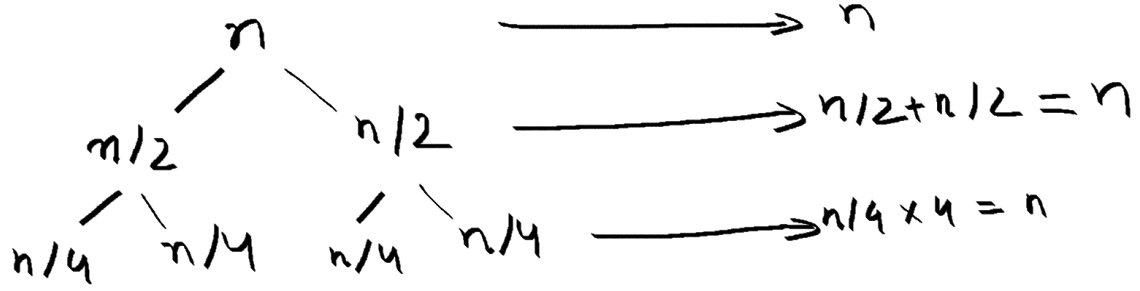
Recurrence Tree Method: [OPTIONAL]

Recursion trees give us a way to visualize the total work done by a recursive algorithm. Each node of a recurrence tree represents a single recursive call, and it stores the work that is done at that recursive call.

To get the total cost or work done, we can sum the work done by each node in the tree.

- If the cost of nodes in each level is the same, we can get the total cost by multiplying the height of the tree and the cost of each level.
- If the cost of nodes in each level is different, we generally get the total sum by using an infinite geometric sequence and adding the cost of all levels.

$T(n) = 2T(n/2) + n,\ T(1) = 1$



$T(n/2^k) = T(1)$

$\Rightarrow n/2^k = 1$

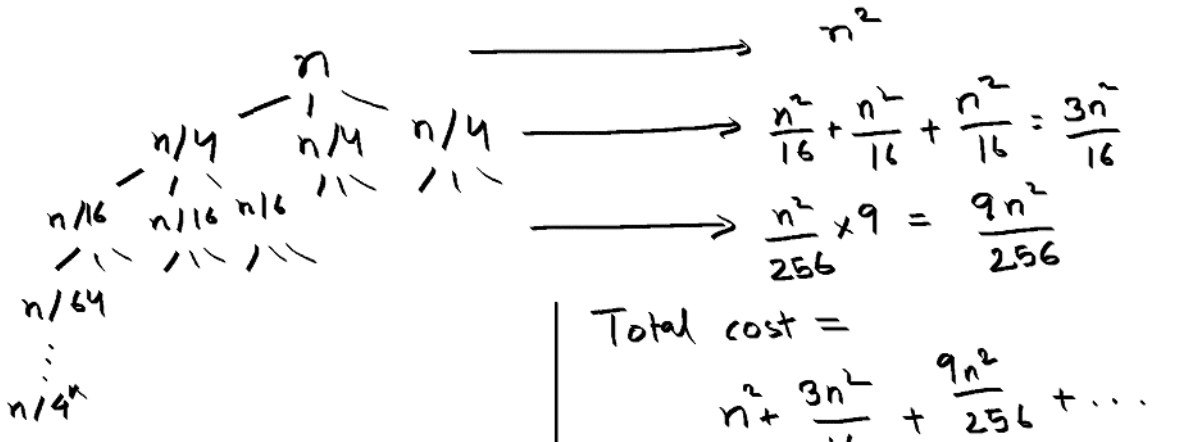
$\therefore k = \log_2 n$

Total cost $= k \times n$

$= \log_2 n \times n$

$\therefore O(n \lg n)$

$T(n) = 3T(n/4) + n^2,\ T(1) = 1$



Total cost $=$

$n^2 + \frac{3n^2}{16} + \frac{9n^2}{256} + \dots$

$= n^2\left(1 + \frac{3}{16} + \left(\frac{3}{16}\right)^2 + \dots\right)$

$= n^2\left(\frac{1}{1 - 3/16}\right)$

$= \frac{16}{13} n^2$

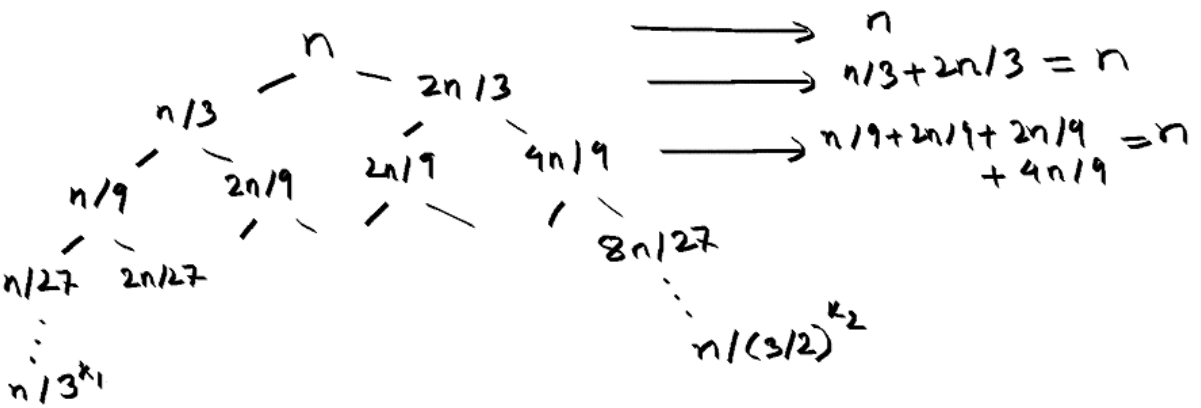
$\therefore O(n^2)$

Sum of infinite geometric sequence:

$a + ar^1 + ar^2 + ar^3 + \dots$

$= \frac{a}{1-r}$, when $r < 1$

$T(n) = T(n/3) + T(2n/3) + n, \; T(1) = 1$



$k_1 = \log_3 n$

$k_2 = \log_{3/2} n$

$k = \max(k_1, k_2)$

$k = \log_{3/2} n$

Total cost $= k \times n$

$= \log_{3/2} n \times n$

$\therefore O(n \log_{3/2} n)$

Why max?
Know the definition of height of a tree