1️⃣

001 - Algorithms Analysis

Note:

^ → Superscript

_ → Subscript

Q) **What is Algorithms Analysis?**

A) We should be concerned about the time and memory resources consumed by our program. We have to design our code depending on the application and our computing system. We have to see whether the code needs to work with less memory and / or less time.

The amount of resources consumed often depends on the amount of data you have. Intuitively, it makes sense that if you have more data you will need more space to store the data. It will also take more time for an algorithm to run.

> 💡 **What we really care about is the growth rate of resource consumption with respect to the data size.**

Analysis of algorithms is about measuring the growth of resource consumption as data size increases. Resources can be anything that has a limited supply. The top two are time and memory but these are not the only ones.

Space Complexity:

Space Complexity of an algorithm is the total space taken by the algorithm with respect to the input size. Space complexity includes both Auxiliary space and space used by input.

Auxiliary Space is the Extra Space or temporary space used by an algorithm.

Like with time complexity, space complexity denotes space growth in relation to the input size. It's also important to note that space complexity usually refers to any additional space that will be needed, and doesn't count the space of the input. So a function could have 10 arrays passed into it, but if all it does inside is print `'Hello World!'`, then it still takes `O(1)` space.

Time Complexity:

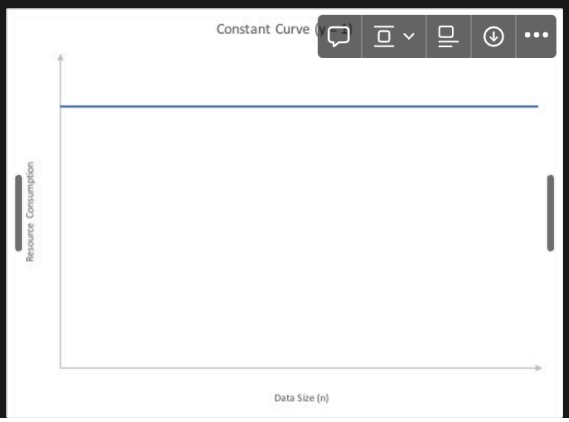
Time Complexity is not equal to time taken. Time Complexity is a mathematical function that gives us the relationship about how the time will grow as the size of the input grows.

Growth Rates:

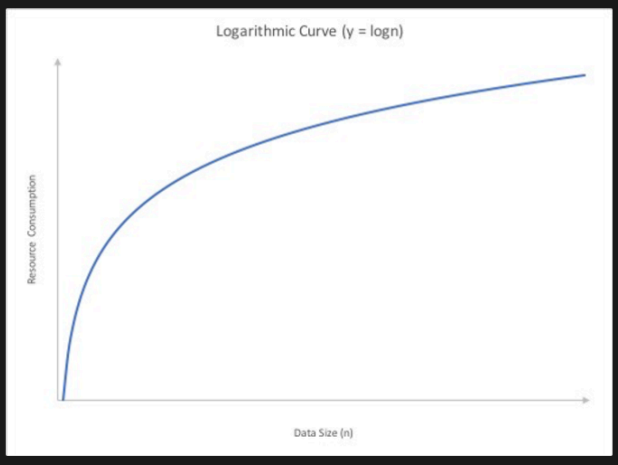
Growth rates can be typically defined as a curve of memory / time and the size of data.

We have the **size of the data on the X axis** and the **resource on the Y axis**.

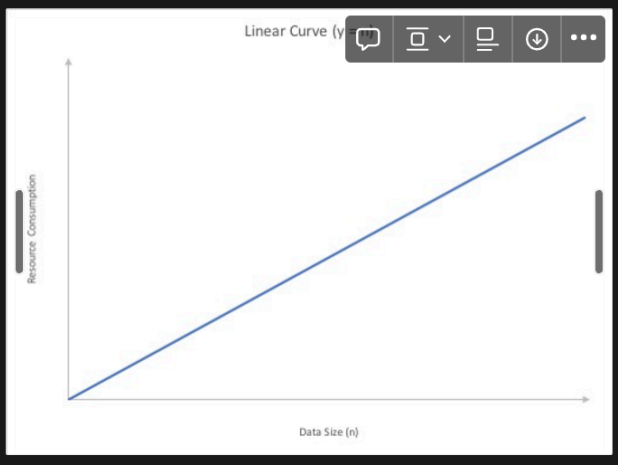
1. **Constant:** Resource does not grow with size.



2. **Logarithmic:** Resource grows by one unit each time the data is doubled. The curve gets fllatter as the size increases but never reaches constant. For ex) If the data size goes from 1,000,000 to 2,000,000, the resource only grows by one unit, so therefore the curve gets flatter with increase in size.

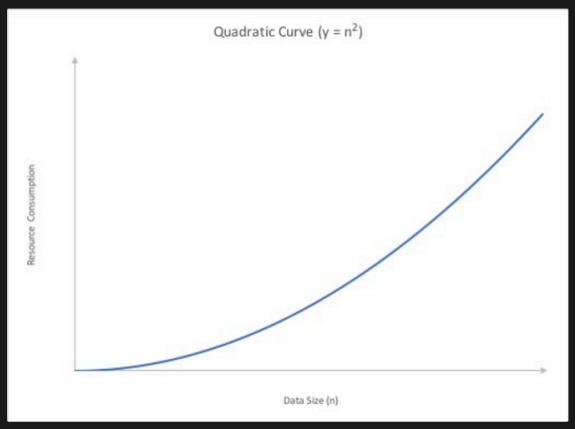


3. **Linear:** Resource and the size of the data are directly proportional to each other.

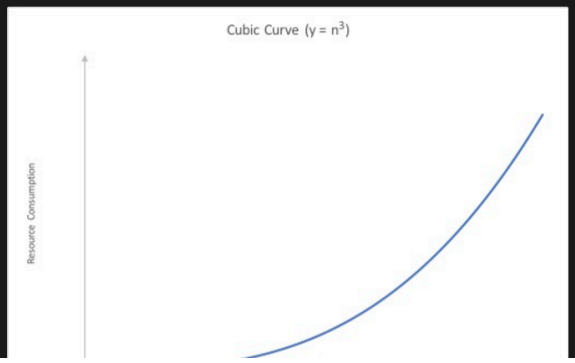


4. **Loglinear:** This is a slightly curved line. It is more pronounced for lower values than the higher ones.

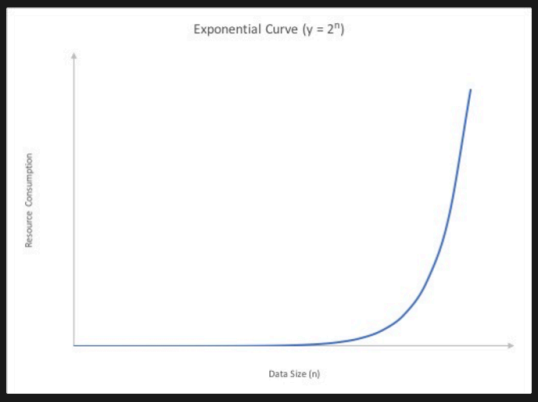
5. **Quadratic:** The curve is a parabola as everytime the value of the resource increases by one unit, the data size is squared.



6. **Cubic:** Similar to quadratic curve but grows a lot faster



7. **Exponential:** An exponential growth rate is one where each extra unit of data requires a doubling of resource. As you can see the growth rate starts off looking like it is flat but quickly shoots up to near vertical (note that it can't actually be vertical)



Asymptotic Notation:

Instead of timing a program, through asymptotic notation, we can calculate a program's runtime by looking at how many instructions the computer has to perform based on the size of the program's input: N.

> 💡 **Asymptotic means: As N approaches infinity**

These are notations used to describe

1. **Big O notation (O):**
    a. It represents the upper bound of the running time of an algorithm. It is used to describe the worst-case scenario or the maximum time taken by the algorithm.
    b. T(n) is O(f(n)) if and only if there exist constants c and n_0 such that $T(n) \leq c(f(n))$ for all $n \geq n\_0$.
    c. If T(n) is O(f(n)), it means that the running time of the algorithm T(n) is no worse than a constant multiple of f(n) for all n greater than or equal to some n_0.
2. **Omega notation (Ω):**

2. **Omega notation (Ω):**
    a. It represents the lower bound of the running time of an algorithm. It is used to describe the best-case scenario or the minimum time taken by the algorithm.
    b. T(n) is Ω(f(n)) if and only if there exist constants c and n_0 such that T(n) ≥ c(f(n)) for all n ≥ n_0.
    c. If T(n) is Ω(f(n)), it means that the running time of the algorithm T(n) is no better than a constant multiple of f(n) for all n greater than or equal to some n_0.
3. **Theta notation (Θ):**
    a. It represents the tight bound of the running time of an algorithm. It is used to describe the average-case scenario or the exact time taken by the algorithm. We use big Theta when a program has only one case in term of runtime.
    b. T(n) is Θ(f(n)) if and only if T(n) is O(f(n)) and T(n) is Ω(f(n)).
    c. If T(n) is Θ(f(n)), it means that the running time of the algorithm T(n) is both O(f(n)) and Ω(f(n)), which means that it has the same order of growth as f(n).
4. **Little o notation (o):**
    a. It is similar to big O notation, but it provides a more precise upper bound on the running time of an algorithm.
    b. T(n) is o(f(n)) if and only if T(n) is O(f(n)) and T(n) is not Θ(f(n)).
    c. T(n) is strictly less than a constant multiple of f(n) for all n greater than or equal to some n_0. In other words, if T(n) is o(f(n)), it means that the algorithm T(n) is faster than f(n) but not as fast as Θ(f(n)).

> **These notations provide a standardized way to compare the efficiency of different algorithms or functions and help in selecting the most appropriate one for a particular use case. In practice, most of the time we deal with either worst case or average case. We rarely ever consider best case. Each case can still have an upper bound(big-O) or lower bound (Omega).**

💡 **The rankings from lowest to highest resource growth are:**
O(1) > O(logn) > O(n) > O(nlogn) > O(n^2) > O(n^3) > O(2^n).

Mathematical Breakdown:

T(n) is O(f(n)) iff there exist constants c and n_0 such that T(n) ≤ c(f(n)) for all n ≥ n_0.

- **n:** Size of the dataset
- **T(n):** A function that describes the usage of a resource with respect to n.
- **f(n):** A function that describes a curve.
- **O(f(n)):** The curve described by *f(n)* is an upper bound for the resource needs of a function *T(n)*
- **c and n_0:** Constants are values that do not change with respect to n, and for a definition of Big O notation to be true, two values, c and n_0, both constants, must exist such that the rest of the statement is true.
- **T(n) ≤ cf(n):** This constant factor, denoted by 'c', can take any value, as long as it is a constant. So, if T(n) is O(f(n)), it doesn't necessarily have to follow the exact shape of f(n), it can be under any constantly scaled version of f(n). For example, if f(n) is n^2, T(n) can be under 10n^2 or 200n^2, or any other constant value times n^2.
- **n ≥ n_0:** T(n) can exceed the expected growth rate given by cf(n) for small values of n, but once n becomes large enough (greater than n_0), T(n) must fall below cf(n) and stay there. This is often useful in analyzing algorithms or functions where there are some initial costs or overhead that contribute to the growth rate of T(n) but are eventually overcome as n gets larger. n_0 is the point of intersection of the curves T(n) and cf(n).

💡 **In summary, when we are looking at big-O, we are in essence looking for a description of the growth rate of the resource increase. The exact numbers do not actually matter in the end.**

Steps of Analysis:

```python
def factorial(num):
    fact = 1
    for i in range(1, num + 1):
        fact *= i
    return fact

print(factorial(4)) # Output: 24
```

STEP 1: ESTABLISH VARIABLES AND FUNCTIONS (MATHEMATICAL ONES)

- Let n represent the value we are finding the factorial for.
- Let T(n) represent the number of operations necessary to find n!.

STEP 2: COUNT YOUR OPERATIONS

```python
def factorial(num):
    fact = 1 # 1
    for i in range(1, num + 1): # (n - 1) + 1 + 1
                               # (n-1) loop iterations
                               # 1 more for + operator
                               # 1 more for call to range function
        fact *= i   # 2(n-1)
                    # 2 as there are two operators
                    # (n-1) as loop runs n-1 times
    return fact     # 1

print(factorial(4)) # Output: 24
```

> 💡 **In python3, range() and len() are both constant. Further, range() calls are done only once to determine the range() of values used for looping. This is not the same as testing the continuation condition in C/C++ which happens at top of every loop. Here, you loop through the entire range of values.**

- You change the value of i each time you go through loop. Thus we have (n - 1).
- As range is constant we only count it once. range() is only called once to get all the values loop must iterate through.
- Whatever takes some time to execute is basically counted as an operation
- Additional Note: You can count [] as an op if you want. If you do it, do it consistently or not at all.

STEP 3: ESTABLISH THE MATHEMATICAL EXPRESSION FOR T(n)

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

STEP 4: SIMPLIFY YOUR EXPRESSION

$T(n) = n + 3 + 2n - 2$

$T(n) = 3n + 1$

STEP 5: STATE YOUR FINAL RESULT

T(n) is O(n)

> **Once you have a polynomial, you can find the dominating term. This is the term that, as n becomes bigger and bigger, the other terms become so insignificant that they no longer matter. In this case the term is 3n. Remove the constant 3. We don't need it because the definition of big-O allows us to stretch the curve. Putting it in will make it look like you don't understand the definition so typically you remove it.**

Analysis of Linear Search:

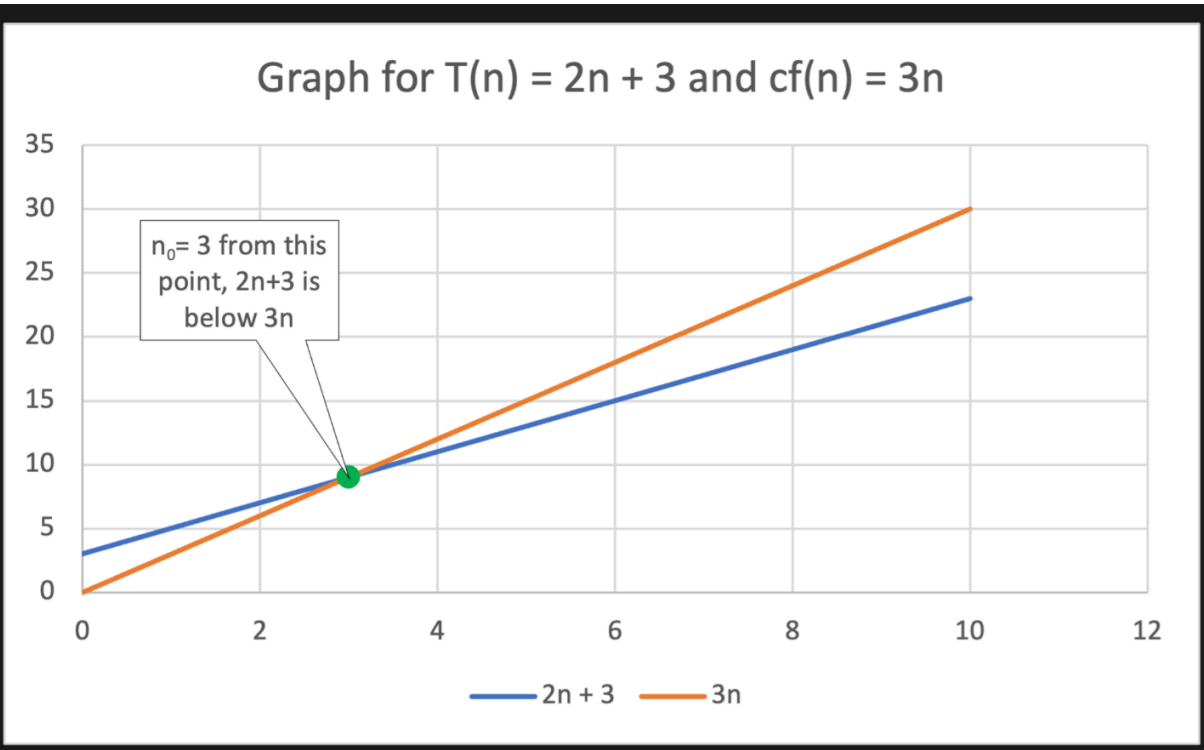
```
def linear_search(my_list, key):
    for i in range(0, len(my_list)): # n + 1 + 1
        if my_list[i] == key: # n
            return i
    return -1 #1 -> Worst Case

print(linear_search([32, 454, 7, 21, 754, 1234], 454)) # Output: 1
print(linear_search([32, 454, 7, 21, 754, 1234], 1)) # Output: -1
```

- Let n represent the size of the array.
- Let T(n) represent the number of operations necessary to perform linear search on an array of items.
- T(n) = n + 1 + 1 + n + 1 = 2n + 3

> Now, imagine n becoming a very very large number. As n gets bigger and bigger, the 3 matters less and less. Thus, in the end we only care about 2n. 2n is the ***dominating term*** of the polynomial. This is similar to this idea. If you had 2 dollars, then adding 3 dollars is a significant increase to the amount of money you have. However, if you had 2 million dollars, then 3 dollars matter very little. Similarly, in order to find something in an array with 2 million elements by comparing against each of them, doing just 3 ops we need to do to find the range, length of list, and return matters very little.

- **T(n) is O(n)**
- T(n) is O(f(n)) if and only if there exist constants c and n_0 such that $T(n) \leq c(f(n))$ for all $n \geq n\_0$. Now c can be any constant that we want as long as it fulfills the criteria. For c, we can pick any constant greater than 2 as $2n + 3 \leq c(f(n))$. So if we pick 2, it would be $2n+3 \leq 2n$ which is not true.
- If we pick 3 however, it would be $2n + 3 \leq 3n$, which is true.



- Here, At n == 3 which is our n_0 now, the line for 2n+3 falls under 3n and it never gets above it again as n gets bigger and bigger. So, c = 3, n_0 = 3 as well.
- This was the analysis for the worst case scenario. However, even if we do it for the best case, it would end up the same.

Analysis of Binary Search:

```python
def binary_search(arr, key):
    start = 0 # 1
    end = len(arr) - 1 # 3

    # loop runs (log_2 n + 1) times
    while (start <= end): # 1
        mid = start + int(((end - start) / 2)) # 1 + 1 + 1 + 1 + 1 = 5
        if arr[mid] == key: # 1
            return mid
        elif arr[mid] > key: # 1
            end = mid - 1
        elif arr[mid] < key: # 1
            start = mid + 1 # 1
    return -1 # 1

print(binary_search([1, 2, 3, 4, 5, 6, 7, 8, 9, 10], 10)) # Output: 9
print(binary_search([1, 2, 3, 4, 5, 6, 7, 8, 9, 10], 11)) # Output: -1
```

- Now we will analiyze the run time for the worst case.

Loop iteration	Number of elements between low and high inclusive (approx)	Denominator
1	n	1 = 2^0
2	n / 2	2 = 2^1
3	n / 4	4 = 2^2
4	n / 8	8 = 2^3
5	n / 16	16 = 2^4
...	...	...
??	1 = *n* / *n*	n = 2^n

- As we know: **b^x = a → x = log_b a**
- If we can find what we need to raise 2 to in order to get n we will be able to find the number of iterations the loop will run.
- so: **2^x = n → x = log_2 n**
- Our loop runs one more time than the exponent, thus our loop runs **(log_2 n + 1)** times
- T(n) = 4 + 9(logn + 1) = 4 + 9logn + 9

  T(n) = 9logn + 13
- **T(n) is O(logn)**

Analysis of Recursive Algorithms:

At any particulart point in time no two calls at the same level of recursion will be in the stack at the same time. Only calls that are interlinked wil be in the stack at the same time.

Space Complexity = Height of the recursion tree

Analysis of a Recursive Function: Factorial

Code:

C++ Copy Caption

```cpp
unsigned int factorial(unsigned int n){
    unsigned int rc = 1;
    if(n > 1) {
        rc = n * factorial(n-1);
    }
    return rc;
}
```

Analysis Steps:

1. Declare mathematical variables and functions:
    - Let `n` represent the number we are finding the factorial of.
    - Let `T(n)` represent the number of operations required by the recursive function.
2. Count the number of operations:
    - Initialize `rc` with 1 (1 operation).
    - Check if `n` is greater than 1 (1 operation).
    - If true, assign `rc` as `n` multiplied by the result of `factorial(n-1)`.
        - This step requires 3 operations and the number of operations performed by `factorial(n-1)`.
3. Rewrite the count summary:

C++ Copy Caption

```cpp
unsigned int factorial(unsigned int n){
    unsigned int rc = 1;
    if(n > 1) {
        rc = n * factorial(n-1);
    }
    return rc;
}
```

1. Define `T(n)` in terms of `T(n-1)`:

```
T(n) = 3 + T(n-1)
```

1. Determine base cases:
    - `T(0) = 3`
    - `T(1) = 3`
2. Determine the general case for `T(n)` when `n >= 2`:

```
T(n) = 6 + T(n-1)
```

1. Observe the pattern and derive a formula for `T(n)`:
   - `T(n-1) = 6 + T(n-2)`
   - `T(n-2) = 6 + T(n-3)`
   - ...
   - `T(1) = 3`
   - `T(0) = 3`

   Therefore,

```
T(n) = 6 + 6 + ... + 6 + 3  (n-1 times)
     = 6(n-1) + 3
     = 6n - 3
```

2. Complexity analysis:
   - The number of operations, `T(n)`, is proportional to `n`.
   - The time complexity of the factorial function is `O(n)`.

Condensed summary:

The recursive factorial function has a time complexity of `O(n)` since the number of operations required to calculate the factorial of `n` is directly proportional to `n`.