Algorithim Analysis

- A way to say how efficient a computer algorithm is
- The speed of an algorithm can change significantly with the size of it's input
- for small datasets , a particular algorithm might be faster than than another, but as the dataset increases, the relative performance can shift.
- Comparing algorithms effectifely needs measures that are independent of machine specific charactersitcts,programing language

Execution Times are highly variable and depend on several factors unrelated to the algorithm's inherent efficiency such as : **Hardware**, Background processes so not a good measure.

Number of Statements executed is not a good measure since it varies with the programing language as well as the styple of the individual programmer

Ideal Solution: Function of Input Size (f(n))

The solution for comparing algorithms is to express their **running times** as functions of the input size. This offers several advantages:

Machine Independent, Language Independent, programing style , etc

The growth rate of an Algorithm

Refers to how the running time of an algorithm increases with the size of the input.

Higher Order Terms in the Context of Rate of Growth

In the context of the rate of growth, we are often interested in the term with the highest power.

For instance Let us assume that you go to a shop to buy a car and a bicycle. If your friend sees you there and asks what you are buying, then in general you say buying a car. This is because the cost of the car is high compared to the cost of the bicycle (approximating the cost of the bicycle to the cost of the car).

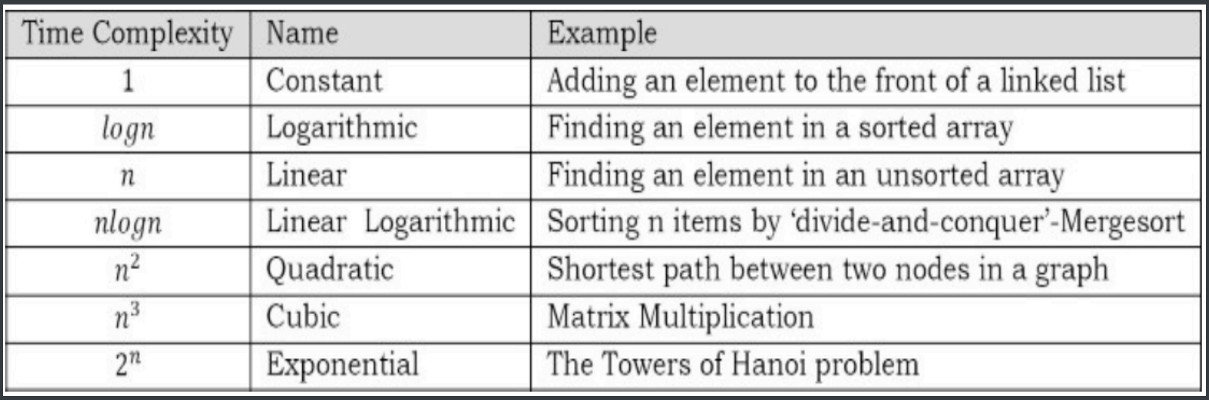
```
                       Total Cost = cost_of_car + cost_of_bicycle
                        Total Cost = cost_of_car(approximation)
```

- We can represent the cost of the car and bicycle in terms of function
- and for a given function ignore the lower oder terms that are relatively insignificant
- for example in the case n^4 , $2n^2$,$100m$ and 500 are the individual costs of some function and approximate to n^4 since is the highest growth rate

$$n^4 + 2n^2 + 100n + 500 \approx n^4 \tag{1}$$

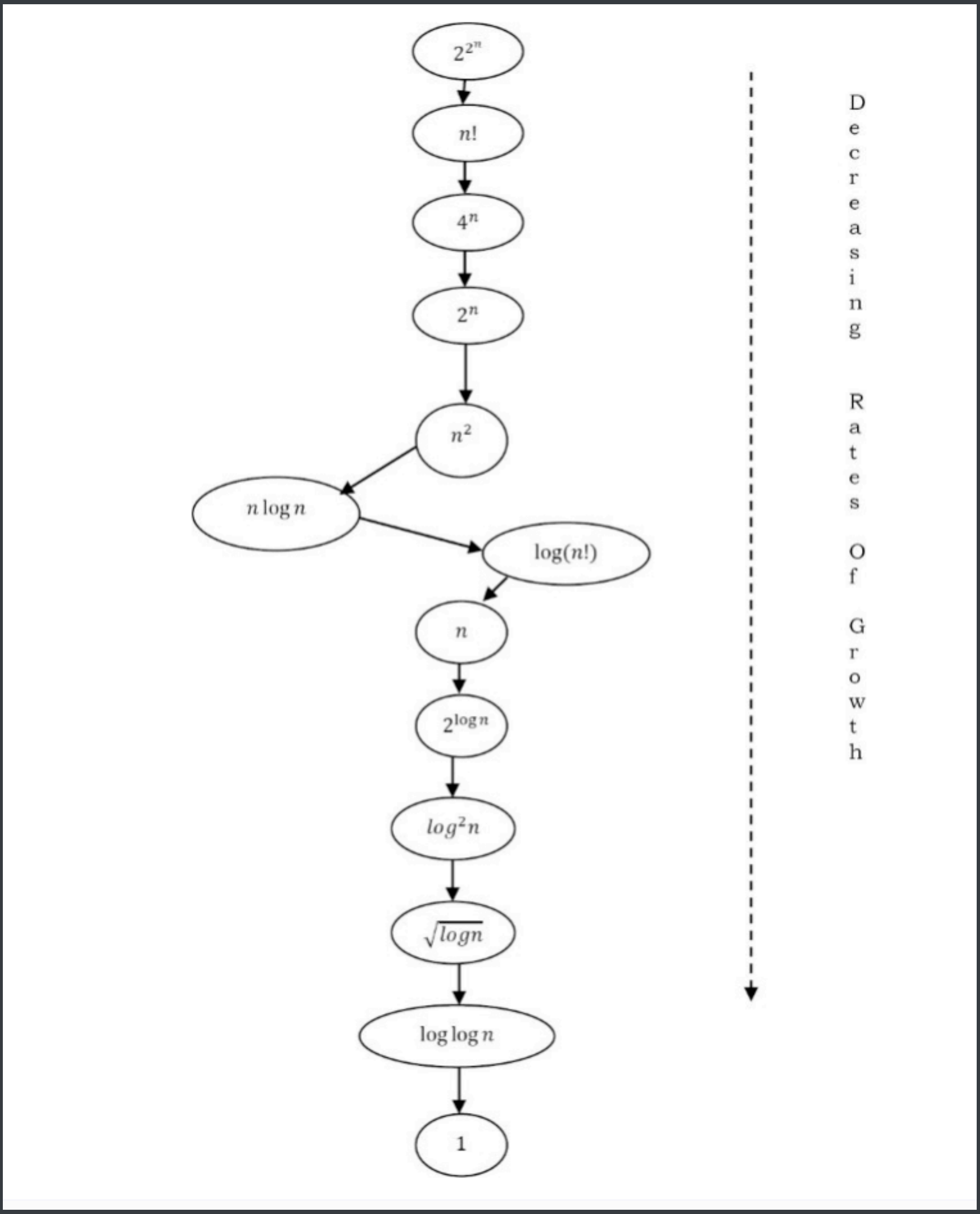
Commonly Used Rates of Growth

Below is the list of growth rates you will come across in the following chapters.



Time Complexity	Name	Example
1	Constant	Adding an element to the front of a linked list
$logn$	Logarithmic	Finding an element in a sorted array
n	Linear	Finding an element in an unsorted array
$nlogn$	Linear Logarithmic	Sorting n items by 'divide-and-conquer'-Mergesort
n^2	Quadratic	Shortest path between two nodes in a graph
n^3	Cubic	Matrix Multiplication
2^n	Exponential	The Towers of Hanoi problem

The diagram below represents different functions ordered by rate of growth



From top to bottom , the functions are ordered from the highest growth rate to the lowest:

- 2^{2^n}(Double exponential growth – extremely fast-growing function)
- $n!$(Factorial growth – very fast-growing for large n)
- 4^n (Exponential growth with base 4)
- 2^n (Exponential growth with base 2)
- n^2 (Polynomial growth – specifically, quadratic growth)
- $nlogn$(Linearithmic growth – faster than linear but slower than quadratic)
- $log(n!)$(Logarithmic factorial growth)
- n (Linear growth)
- 2^{logn}(This simplifies to n due to properties of logarithms, so it's linear growth)
- log^2n(Iterated logarithmic growth)
- $\sqrt{l}ogn$ (Logarithmic growth under a square root)
- $loglogn$ (Double logarithmic growth – very slow-growing)
- 1 (Constant time – does not grow with n)

Types of Analysis

- This involves understanding with which input the algorithm takes less time and with which inputs it takes long time
- Measured in terms of **time complexity** (how long it takes to run) and **space complexity** (how much memory it requires)
- Helps to predict how an algorithm performs in various senarios
- Relevant for the selection of the most efficent algorithm for a given problem

  The three types of analysis are:

Worst Case Analysis (Big 0 Notation)

- Defines the input for which the algorithm takes a **long time**(slowest time to complete)
- It is a measure of the **maximum amount of time** an algorithm is allowed to run

  Importance: it gives an **upper bound** on the running time, ensuring the algoritm will not perform worse than

  this under any circumstance

  Example: In the case of linear search algorithm, the **worst case occurs** when the element being search is the **last element** of the array or **is not present at all**.

  - if the array has n elements, the worst case time complexity is $O(n)$

Best Case Analysis(Big Omega Notation)

- identifies the input for which the algorithm takes the least amount of time to complete
- Is a measure of the minimum time an algorithm is required to run

  Importance: It provides a lower bound on the running time of an algorithm, representing the most optimistic scenario

  Example: In the case of linear search, the best case occurs when the element being search is the first element of the array.

  - The best case time complexity is $0(1)$

Average Case Analysis (Big Theta Notation)

- This tries to provide a prediction about the running time of an algorithm under **average** case
- It involves running the algorithm multiple times with various inputs and calculating the average running time
- **Importance :** It offers a more realistic measure of an algorithm's performance, assuming all inputs are equally likely to occur
- Example : For linear search search, if we assume that the search element is equally likely to be at any position in the array, the average case becomes $O(n/2)$
- For simplification in Big O notation this is considered $O(n)$

Asymptotic Notation

- Asymptotic notation is a language we use to define the upper and lower bounds of an algorithm's running time
- It simplifies how we express the rate of growth of an algorithms runtime represented by `n`

Big O Notation

- Big O notation is a mathematical concept used in computer science to describe the upper bound of an algorithm's performance
- It gives us an upper limit on the time an algorithm will take to run in terms of the input data.
- $f(n) = O(g(n))$ means that the function $f(n)$ grows at a rate that is at constant multiple of $g(n)$ for large enough values of n.
- $f(n)$ represents the complexity of the algorithm for an input size of n
- O stands for oder of , indicating the growth rate in terms of the upper limit
- $g(n)$ represents the upper bound on the growth rate of $f(n)$. it helps to classify the complexity of the algorithm.
- To say that $f(n) = O(g(n))$ is to assert that there exist a positive constant C and a value n_0

  such that for all $n \geq n_0$, the ineqality $f(n) \leq C.g(n)$ holds true.
- Generally lower values of n are discarded

  for instance give two functions that represent the time complexity of two different algorithms for processing a an array of n elements:

  1. $f(n) = 3n + 2$ we say that this function is $O(n)$ because as n grows, the. dorminant term in the function is $3n$
     - The constant factors like 3 and + 2 don't affect the growth rate for large n
  2. $g(n) = n^2$ the dorminant term is n^2 so the growth rate is quadratic so it is classified as $O(n^2)$

Example

1. Find upper bound for $f(n) = 3n + 8$

   To find the upper bound for the function using Big O . we aim to identify a function $g(n)$ such that $f(n)$

   does not grow faster than $C.g(n)$ for some constant C and for all n greater than some n_0

   we show $3n + 8 \leq C.n$ **for all n** $\geq n_0$

   Choosing C to be 4 we can see that:

   $3n + 8 \leq 4n$ for all $n \geq 8$

   Therefore, $O(n)$ = 3n + 8 with c=4 and n_0 = 8

2. Find the upper bound for $f(n) = n^2 + 1$

   The term that grows the fastest as n increases is n^2 so we are considering a quadratic growth rate as the upper bound for $f(n)$

   we want to show $n^2 + 1 \leq C.n^2$ for all $n \geq n_0$

   Choosing $C = 2$ we can see that $n^2 + 1 \leq 2n^2$ for all $n \geq 1$

   Therefore $O(n) = n^2$

Exercise

1. Find the upper bound for $f(n) = n^4 + 100n^2 + 50$
2. Find the upper bound for $2n^3 - 2n^2$

There is no unique set of values for n0 and c in proving the asymptotic bounds. Let us consider,

Omega-Ω Notation

- Similar to the O discussion, this notation gives the tighter lower bound of the given algorithm
- it is represented as $f(n) = \Omega(g(n))$
- There exist a positive constant C and n_o such that $0 \leq c.g(n) \leq f(n)$ for all $n \geq n_o$
- $g(n)$ is a lower bound of $f(n)$

Example: Find the lower bound of $5n^2$

The primary component of the functions growth rate is n^2

Therefore we can say $5n^2 = \Omega(n^2)$. for some constant $C > 0$ and for all $n \geq n_o, 5n^2 \geq c.n^2$

Theta θ Notation

- The Θ-Notation basically decide whether the upper and the lower bound of a given algorithmic function are the same.
- The average running time is between the lower and the upper bounds.
- When the lower and upper bounds are the same then the Θ-Notation will also have the same growth rate.Similarly,
- when the best and the worst cases are the same then the average case will also be the same.

Definition:

$$\Theta(g(n)) = \{f(n) : \text{there exist positive constants } c_1, c_2 \text{ and } n_0 \text{ such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n$$

Example:

Find θ bound for $f(n) = \frac{n^2}{2} - \frac{n}{2}$

Bounding $f(n)$ below:

$\frac{n^2}{5} \leq f(n)$

Bounding $f(n)$ above:

$f(n) \leq n^2$

Combining the Inequalities:

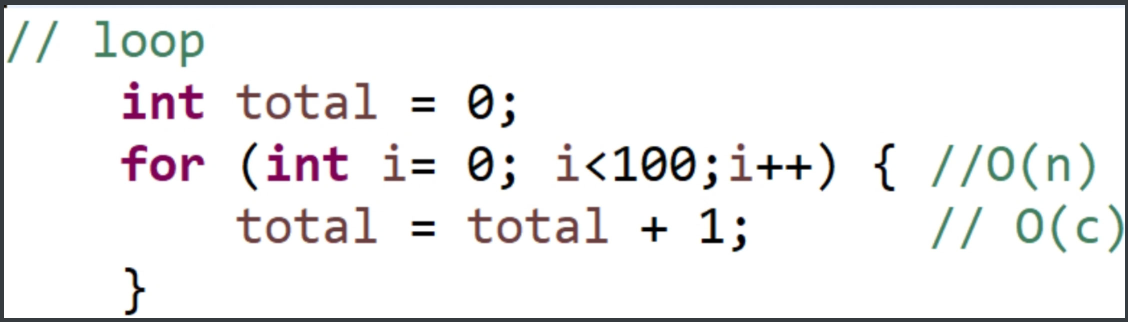
$\frac{n^2}{5} \leq f(n) \leq n^2$

This is stating that $f(n)$ is squeezed between $\frac{n^2}{5}$ and n^2 for all $n \geq 2$

Therefore $f(n) = \Theta(n^2)$ with constants $c_1 = \frac{1}{5}$, $c_2 = 1$ and $n_0 = 2$ which satisfies the defination of Big Theta:

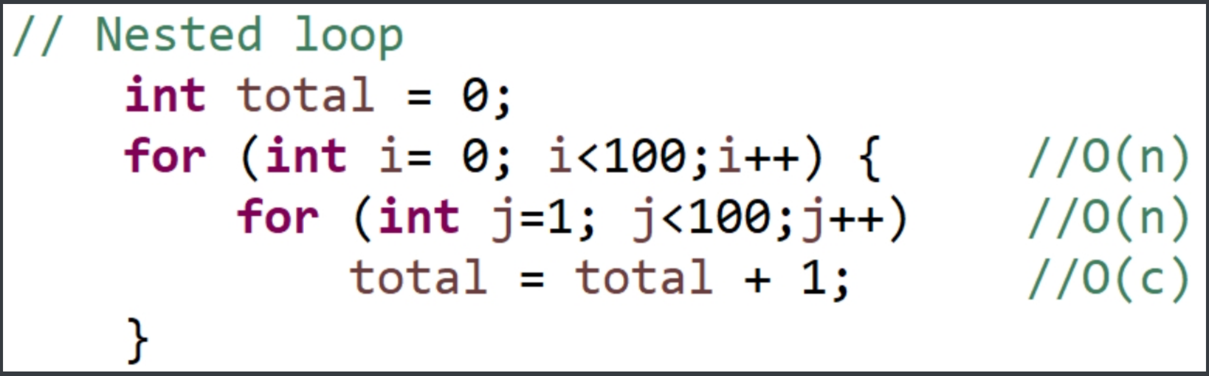
Guidelines for Asymptotic Analysis

- There are some general rules to help us determine the running time of an algorithm.
  1. **Loops**: The running time of a loop is, at most, the running time of the statements inside the loop (including tests) multiplied by the number of iterations.



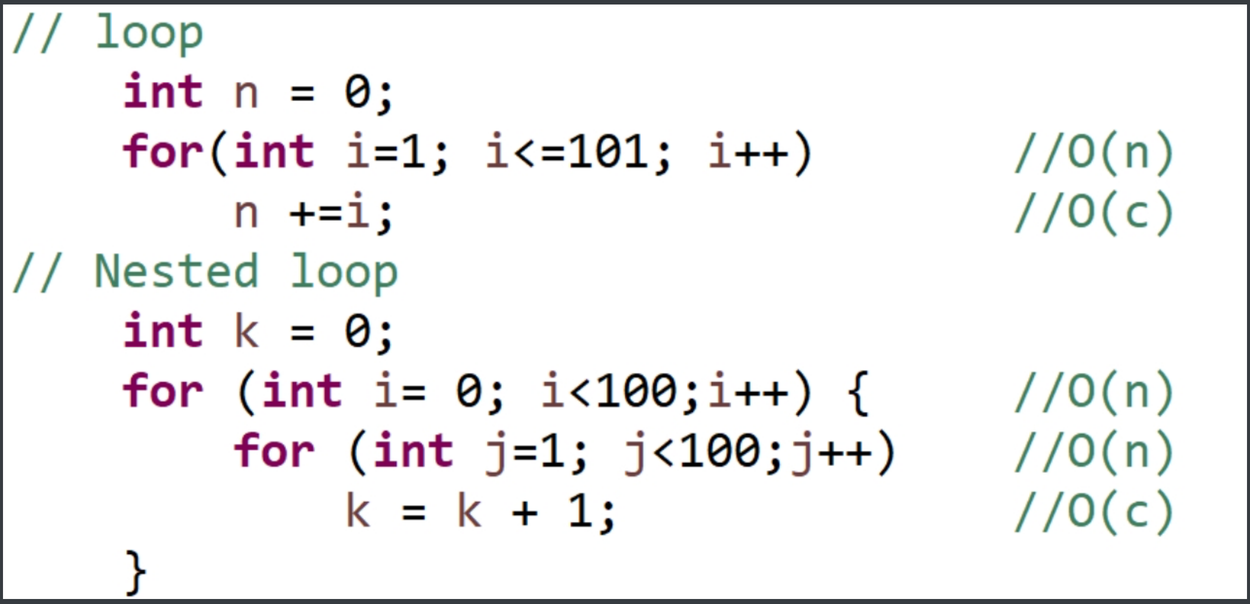
```
// loop
    int total = 0;
    for (int i= 0; i<100;i++) { //O(n)
        total = total + 1;      // O(c)
    }
```

Total time = c×n=cn=O(n)



```
// Nested loop
     int total = 0;
     for (int i= 0; i<100;i++) {      //O(n)
          for (int j=1; j<100;j++)    //O(n)
               total = total + 1;     //O(c)
     }
```

Total time = $c \times n \times n = n^2 = O(n^2)$



```
// loop
     int n = 0;
     for(int i=1; i<=101; i++)         //O(n)
          n +=i;                       //O(c)
// Nested loop
     int k = 0;
     for (int i= 0; i<100;i++) {       //O(n)
          for (int j=1; j<100;j++)     //O(n)
               k = k + 1;              //O(c)
     }
```

Total time = $c \times n \times n + c \times n = c_1 n^2 + c_2 n = O(n^2)$

Sorting

- When you have a significant database, you might think of way to sort it
- you need to arrange names in alphabetical order , students by gradem customers by zip code, cities in order of increasing population, countries by GDP
- sorting helps in structuring data in a way that makes it more accessible and easier to understand.
- Sorting isn't just about organization for the sake of clarity; it also serves a functional purpose in the context of searching through data.
- Sorting data may also be a preliminary step to searching it. for exampe in binary search
- Because sorting is so important and potentially so timeconsuming, it has been the subject of extensive research in computer science.
- some very sophisticated methods have been developed.
- simpler sorting algorithms include selection sort , bubble sort and insertion sort
- Advance sorting algorithms include **Shellsort** and **quicksort sort**