# Asymptotic Analysis

Debarka Sengupta

# Slides are developed with help from

- Shreepriya Dogra
- Nupur Ahluwalia

# Background

# What is an Algorithm ?

- An algorithm is a step-by-step procedure or formula for solving a problem, based on conducting a sequence of specified actions.
- An informal definition could be "a set of rules that precisely define a sequence of operations", which would include all computer programs, including programs that do not perform numeric calculations.

# History

- Greek mathematicians used algorithms in, for example, the *sieve of Eratosthenes* for finding prime numbers and the *Euclidean algorithm* for finding the greatest common divisor of two numbers.
- The word **algorithm** itself derives from the 9th century mathematician Muḥammad ibn Mūsā al-Khwārizmī, Latinized *Algoritmi.*
- A partial formalization began with attempts to solve the Entscheidungs problem(decision problem) posed by David Hilbert in 1928.
- Later formalizations were framed as attempts to define "effective calculability"[9] or "effective method".

# Analysis of Algorithm

- To evaluate rigorously the resources (time and space) needed by an algorithm and represent the result of the evaluation with a formula
- In this course, we focus more on **time** requirement in our analysis
- The time requirement of an algorithm is also called the **time complexity** of the algorithm

# Why Algorithm Analysis?

- Predict performance
- Compare algorithms
- Provide guarantees
- Understand theoretical basis

**Primary practical reason:** avoid performance bugs.

# Why not measuring execution time?

- We can measure the actual running time of a program
  - Use **wall clock time** or insert timing code into program

- Simple time calculation gets impacted by
  - CPU speed
  - Different OS - different memory management
  - RAM size
  - Programming language
  - Algorithm implementation

# Some huge algorithmic successes

**Discrete Fourier transform:**

- Break down waveform of $N$ samples into periodic components.
- **Applications:** DVD, JPEG, MRI, astrophysics, ….
- **Brute force:** $N^2$ steps.
- **FFT algorithm:** $N \log N$ steps, **enables new technology.**

# Some huge algorithmic successes

**N-body simulation**:

- Simulate gravitational interactions among $N$ bodies.
- **Brute force:** $N^2$ steps.
- **Barnes-Hut algorithm:** $N \log N$ steps, **enables new research**.

# Asymptotic analysis - the route

Understanding algorithm

Expressing count of instructions wrt problem size

Understanding asymptotic behavior of the instruction counts

# Counting

# Counting Operations

- Instead of measuring the actual timing, we count the number of operations.
  - Operations: arithmetic, assignment, comparison, etc.
- Counting an algorithm’s operations is a way to assess its efficiency.
- An algorithm’s execution time is related to the number of operations it requires.

# Example: count number of operations required

```
for (int i = 1; i <= n; i++)
{
	perform 100 operations; // A

	for (int j = 1; j <= n; j++)

		{
			perform 2 operations; // B
		}
}
```

# Example: count number of operations required

```
for (int i = 1; i <= n; i++)
{
        perform 100 operations; // A

        for (int j = 1; j <= n; j++)

                {
                        perform 2 operations; // B
                }
}
```

Total Ops = A + B $= \sum_{i=1}^{n} 100 + \sum_{i=1}^{n} (\sum_{j=1}^{n} 2)$

$$= 100n + \sum_{i=1}^{n} 2n \quad = 100n + 2n^2 \quad = 2n^2 + 100n$$

# How does it help?

- Knowing the number of operations required by the algorithm, we can state that

  Algorithm X takes $2n^2 + 100n$ operations to solve problem of size $n$

- If the time t needed for one operation is known, then we can state

  Algorithm X takes $(2n^2 + 100n)t$ time units

# How does it help?

- However, time *t* is directly dependent on the factors mentioned earlier

    e.g. different languages, compilers and computers

- Instead of tying the analysis to actual time *t*, we can state

    Algorithm X takes time that is proportional to $2n^2 + 100n$ for solving problem of size $n$

# How does it help?

- Suppose the time complexity of

    Algorithm A is $3n^2 + 2n + logn + 1/(4n)$

    Algorithm B is $0.39n^3 + n$

- Intuitively, we know Algorithm A will outperform B

    When solving larger problem, i.e. larger n

- The dominating term $3n^2$ and $0.39n^3$ can tell us approximately how the algorithms perform
- The terms $n^2$ and $n^3$ are even simpler and preferred
- These terms can be obtained through asymptotic analysis

# Expectation

Informally, a probability distribution defines the relative frequency of outcomes of a random variable - the expected value can be thought of as a weighted average of those outcomes (weighted by the relative frequency). Similarly, the expected value can be thought of as the arithmetic mean of a set of numbers generated in exact proportion to their probability of occurring (in the case of a continuous random variable this isn't exactly true since specific values have probability 0).

$$E(x) = \sum xP(X=x)$$

# Example: Linear Search Analysis

**Best case:** O(1)= Target is in first position of the array

**Worst Case:** O(n)= Target at Last position or not present

**Average Case:**

**First situation -** The searched item is present in the array. It can be present at any location of the array with equal probabilities.

Expected number of operations for first situation

$= 1 \times 1/n + 2 \times 1/n + 3 \times 1/n \ldots\ldots + n \times 1/n$

$= 1/n\ (1+2+3+\ldots\ldots+n)$

$= 1/n \times n(n+1)/2$

$= (n+1)/2$

# Example: Linear Search Analysis

**Second situation -** The searched element is not in array. Futile scan of array amounts to *n* comparisons.

**Let's reconcile ...**

In absence of any prior understanding of the system, it if fair to assume both situations have equal chances to occur.

Expected number of operations

$= 0.5*n + 0.5*(n+1)/2$

$= n/2 + (n+1)/4$

$= (2n+n+1)/4$

$= (3n+1)/4$

$= 3n/4 + 1/4$

# Find the complexity for the below function:

```c
void function(int n)
{
    int i = 1, s =1;
    while (s <= n)
    {
        i++;
        s += i;
        printf("*");
    }
}
```

# Solution

- We can define the terms 's' according to relation $s_i = s_{i-1} + i$. The value of 'i' increases by one for each iteration.
- The value contained in 's' at the $i^{th}$ iteration is the sum of the first 'i' positive integers.
- If k is total number of iterations taken by the program, then while loop terminates

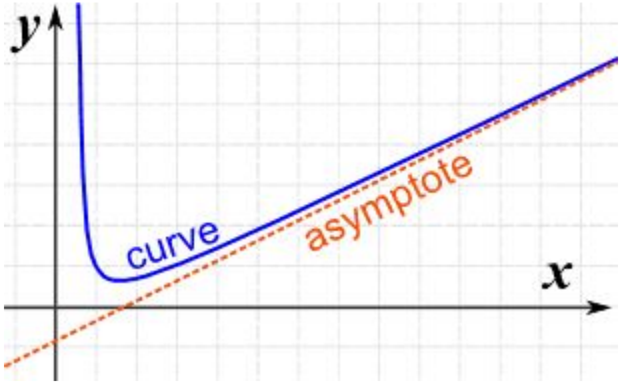
**if: $1 + 2 + 3 \ldots + k = [k(k+1)/2] > n$, So $k = O(\sqrt{n})$.**

- Time Complexity of the above function $O(\sqrt{n})$.

# Asymptotic analysis

# Asymptote

Put simply, an asymptote is a line (or a curve) that the function keeps getting close to but never actually touches(though we symbolically say it touches it at x = infinity).




# Asymptotic Analysis

Asymptotic analysis is an analysis of algorithms that focuses on:

- Analyzing problems of large input size
- Consider only the leading term of the formula
- Ignore the coefficient of the leading term

# Leading Term

- Lower order terms contribute lesser to the overall cost as the input grows larger
- Example
  - $f(n) = 2n^2 + 100n$
  - $f(1000) = 2(1000)^2 + 100(1000)$

    $= 2{,}000{,}000 + 100{,}000$
  - $f(100000) = 2(100000)^2 + 100(100000)$

    $= 20{,}000{,}000{,}000 + 10{,}000{,}000$
- Asymptotic behavior of a function is not dependent on the lower order terms.

# Examples: Leading Terms

- **$a(n) = \frac{1}{2}n + 4$**
  - Leading term: $\frac{1}{2} n$
- **$b(n) = 240n + 0.001n^2$**
  - Leading term: $0.001n^2$
- **$c(n) = n \lg(n) + \lg(n) + n \lg(\lg(n))$**
  - Leading term: $n\lg(n)$
  - Note that $\lg(n) = \log_2(n)$

# Why Ignore Coefficient of Leading Term?

- Suppose two algorithms have $2n^2$ and $30n^2$ as the leading terms, respectively
- Although actual time will be different due to the different constants, the **growth rates** of the running time are the same
- Compare with another algorithm with leading term of $n^3$, the difference in growth rate is a much more dominating factor
- Hence, we can drop the coefficient of leading term when studying algorithm complexity

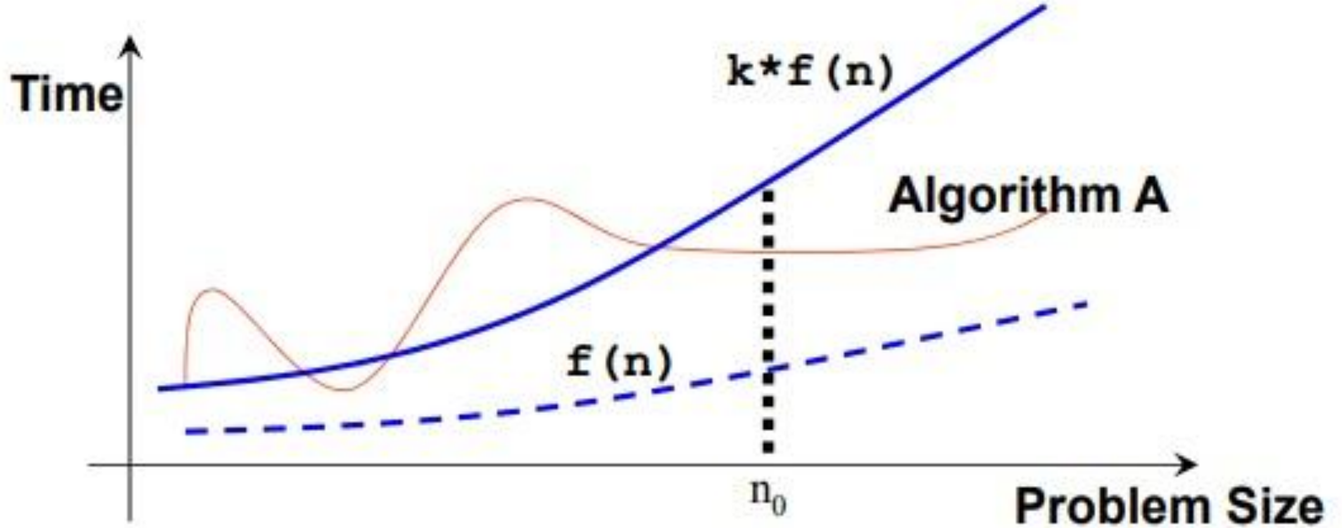
# The Big-O Notation

If algorithm *A* requires time proportional to *f(n)*

- Algorithm *A* **is of the order of** *f(n)*
- Denoted as Algorithm A is **O(*f(n)*)**
- *f(n)* is the **growth rate function** for Algorithm *A*

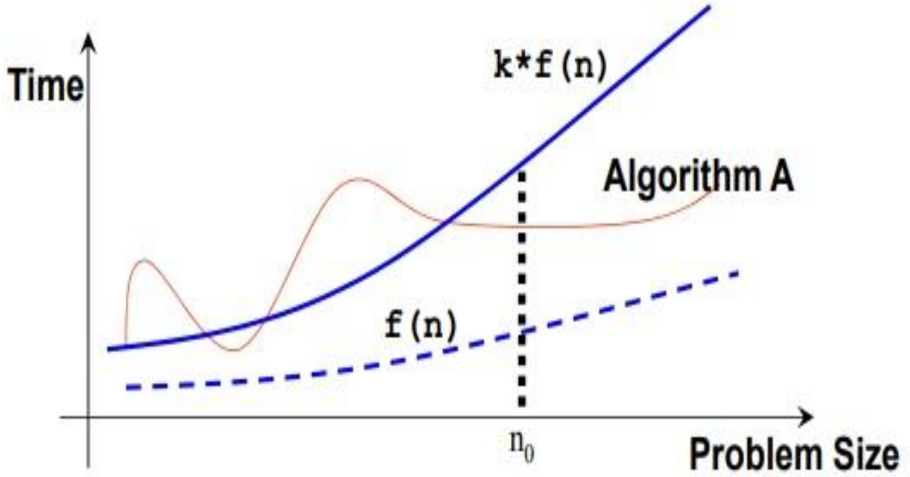
# Formal Definition

- Algorithm *A* is of O(*f( n)*) if there exist a constant **k**, and a positive integer $\mathbf{n_0}$ such that Algorithm *A* requires no more than **k*f(n)** time units to solve a problem of size $\mathbf{n >= n_0}$



# The Big-O Notation

- When problem size is larger than $\mathbf{n_0}$, Algorithm *A* is **bounded from above** by **k * f(n)**
- **Observations :**
  - $\mathbf{n_0}$ and **k** are not unique
  - There are many possible ***f(n)***



# Prove

$$F(n) = n^2 + 42n + 7 = O(n^2)$$

# Solution sketch

**$F(n) = n^2 + 42n + 7 \leq n^2 + 42n^2 + 7n^2 = 50n^2 \ \forall \ n \geq 1$**

**So $F(n) \leq k.G(n) \ \forall \ n \geq n_0$ where $k = 50$ and $n_0 = 1$**

**Now we can say $F(n) = O(G(n)) = O(n^2)$**

# Prove

$$F(n) = 5n.\log_2 n + 8n - 200 = O(n.\log_2 n)$$

# Solution hint

$$F(n) = 5n.\log_2 n + 8n - 200 \leq 5n.\log_2 n + 8n$$

$$\leq 5n.\log_2 n + 8n.\log_2 n \ (\forall\ n \geq 2)$$

$$\leq 13n.\log_2 n \ (\forall\ n \geq 2)$$

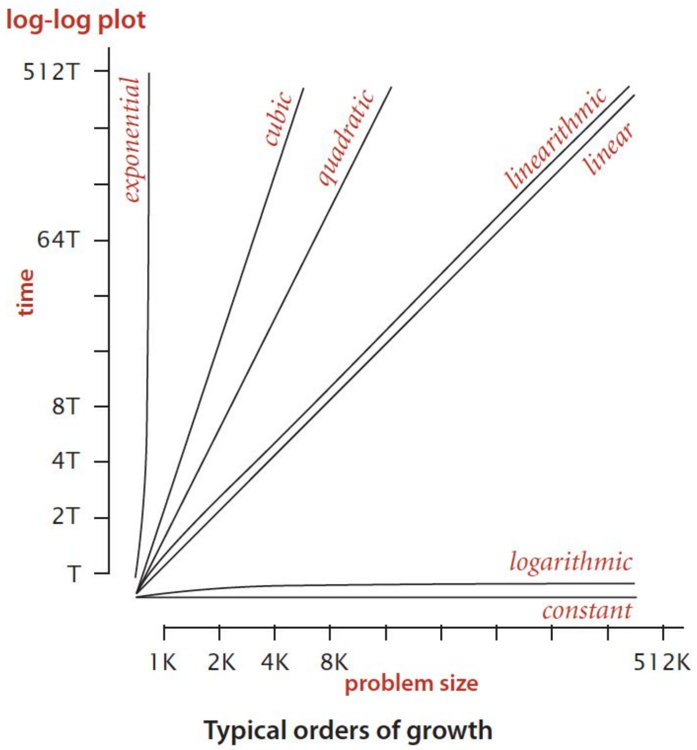
# Prove

$$O(\ln n) = O(\log_2 n)$$

# Solution hint

**$\log_a n = \log_c n / \log_c a = \text{constant} \times \log_c n$**

# Order of growth functions




- **O(1) — constant time**
  - Independent of n
- **O(n) — linear time**
  - Grows as the same rate of n
  - E.g. double input size -> double execution time
- **O($n^2$) — quadratic time**
  - Increases rapidly w.r.t. n
  - E.g. double input size -> quadruple execution time
- **O($n^3$) — cubic time**
  - Increases even more rapidly w.r.t. n
  - E.g. double input size -> 8 * execution time
- **O($2^n$) — exponential time**
  - Increases very very rapidly w.r.t. n