# CS101
# Data Structures and Algorithms

Lecture 03

Need for Algorithm Analysis

# Introduction

- Computer can be used to solve only computational problem
- Real world problems need Computational / Mathematical modeling for both data and operations
- Computer Algorithm is a series of computable steps to achieve a desired computational objective
- Data Structure models Data, and Algorithm sequences Operations
- Program = Algorithm + Data Structures (Horowitz, Sahni Book Cover)

# Lets Count Operations for 64 bit word

- How many additions and multiplications are needed to compute 1+2+3+...+n?
  - Approaches: Naive: (n-1) Add / Formula: 2 Mult, 1 Add, 1 Div
- For sum of squares of n consecutive integers?
  - Approaches: Naive: (n-1) Mult,(n-1) Add / Formula: 3 Mult, 2 Add, 1 Div
- For sum of cubes of consecutive integers?
  - Approaches: Naive: 2(n-1) Mult,(n-1) Add
  - Direct Formula: 3 Mult, 1 Add, 1 Div / Optimized: 2 Mult, 1 Add, 1 Div
- Calculate the operations for all the above three cases for 128 bit numbers.

# Lets Count Constant Operations

- Note: The formula approach gives constant operation count, though not always.
- Constant operation algorithms are all acceptable (?)
- What if the constant is $2^{100}$?
- Say 1 multiplication takes 1 nano second = $10^{-9.}$ > $2^{-27}$ seconds
- Then $2^{100}$ multiplications will take at least $2^{73}$ seconds
- Age of earth in seconds = $1.433 \times 10^{17}$ seconds << $2^{68}$

# Lets keep Counting

- Sum of n terms of a geometric series
  - Naive: $n(n-1)/2$ Mult, $n-1$ Add
  - Single Increment Exponentiation: $(n-1)$ Mult, $n-1$ Add
  - Direct Geometric sum Formula: n Mult, 1 Add, 1 Div
  - Progressive Exponentiation: $\lceil \log_2(n) \rceil$ Mult, 1 Add, 1 Div
  - The last one is optimum
- Moral of the story: Algorithm should be designed for optimal performance

# Performance of an Algorithm

- The term Performance is synonymous to Complexity
- There are two Complexity measures for an algorithm
  - Time complexity and – Space complexity
- Counting the number of operations gives time measure. Justify.
- Increasing input size leads to increasing running time but complexity measure remains same.
- Amount of memory/disk space used gives space measure.
- Typically Time complexity is inversely proportional to Space complexity

# Measuring wall clock time

```
from time import time
start_time = time()  #Start time stamp
run algorithm
end_time = time() # Completion time stamp
Elapsed = end_time - start_time
```

Time may be in secs / epocs. For the later, epoc calculator may be used to decode the wall clock date time.

# Note about Python

- `from` is python keyword
- `time` is the python package
- `import` means to read that package into this source script
- `time`, (after import) is the method that is needed from the package
- Note that the package may contain many other methods which are not included in the source script.

# Primitive Operations

- Assignment operation
- Determining the object associated with an identifier
- Performing an arithmetic operation
- Comparison
- Accessing a single element of a Python list by index
- Function Call
- Function return

# A note about Comparison Operation

- Without Comparison Op only trivial programs that has Linear Flow can be written
- Example: A calculator that does only addition of two numbers. Take two numbers input and output the sum, and nothing more can be done with this calculator.
- Incorporating Comparison Op in the programming language produces Branching Flow
- Example: A calculator that take two numbers and a arithmetic operator as input.
- Based on the operator (Here comparison is used for operation selection) result is computed

# An example for Array Access Abuse

- Problem Instance: Sort an array A of 1024 numbers
  1) Break A into two equal halves A1 and A2
  2) Sort A1, A2 separately   # This is a recursive invocation
  3) Merge the sorted A1, A2 and store it back in A
- If Step 1 creates new subarrays A1, A2, then compute the total memory requirement.
- The entire array of 1024 number is read and copied to A1, A2 with no additional result.
- Proceeding recursively we get the following tree of memory requirement.

# Compute the total memory requirement