# **DS221**: Introduction to Scalable Systems

## Topic: Algorithms and Data Structures

# L2: Complexity Analysis & Performance Evaluation

# Algorithm Analysis

- Algorithms can be evaluated on two performance measures
- Time taken to run an algorithm
- Memory space required to run an algorithm
- Later, I/O and Communication complexity

  …for a given input size

- *Why are these important?*

# Space Complexity

- *Estimate of the amount of peak memory required for an algorithm to run to completion, <u>for a given input size</u>*
  - Core dumps/OOMEx: Memory required is larger than the memory available on a given system
  - Algorithm design problem OR "memory leaks" in implementation
- In some applications, we may want load all data in memory for performance

# Space Complexity

- **Fixed part**: The size required to store certain data/variables, that is independent of the size of the problem:
  - e.g., for all valid words, given a set of letters
  - e.g., etymology for each work in a dictionary
- **Variable part**: Space needed by variables, whose size is dependent on the size of the problem:
  - e.g., number of letters in a scrabble game
  - e.g., text of Shakespeare's plays

**Try yourself!**

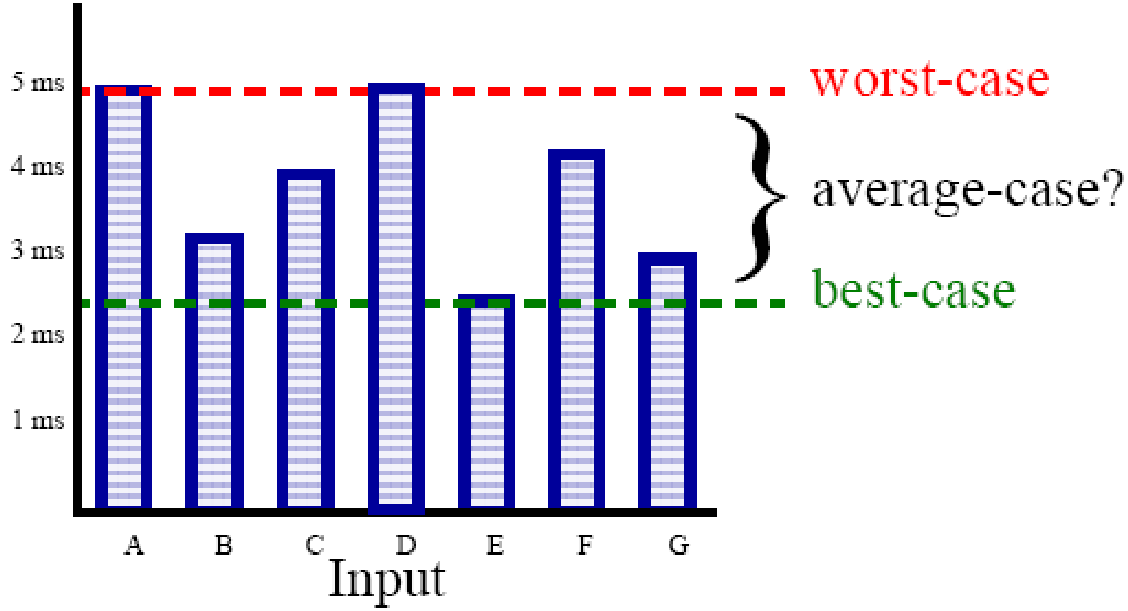
For some program with variable input sizes, find the space taken by the fixed part and the variable part, using **top** command

# Analyzing Running Time Empirically

- Write program
- Run Program
- Measure actual running time with some methods like *std::chrono::high_resolution_clock::now()* in C++

- *Is that good enough as a programmer?*

# Running Time




- Suppose the program includes an *if-then statement that may* execute or not → variable running time
- Typically algorithms are measured by their ***worst case***

**<u>Try yourself!</u>**
Run some program for the same input size (but different inputs), and see how the run time changes for each input

# General Methodology for Analysis

- Uses High Level Description (pseudo-code) instead of implementation
- Takes into account all variations of inputs of some size “n”
- Allows one to evaluate the efficiency independent of hardware/software environment

# Pseudo-Code

- Mix of natural language and high level programming concepts that describes the main idea behind algorithm
  - More detail than algo, less than implementation
- Control flow
  - If … then …else
  - While-loop
  - for-loop
- Simple data structures
  - Array : A[i]; A[I,j]
- Methods
  - Calls: methodName(args)
  - Returns: return value

# Pseudo-Code

- Mix of natural language and high level programming concepts that describes the main idea behind algorithm
  - More detail than algo, less than implementation
- Control flow
  - If … then …else
  - While-loop
  - for-loop
- Simple data structures
  - Array : A[i]; A[I,j]
- Methods
  - Calls: methodName(args)
  - Returns: return value

```
int arrayMax(int[] A, int n)
  Max=A[0]
  for i=1 to n-1 do
    if Max < A[i]
      then Max = A[i]
  return Max
```

# Pseudo-Code

- Mix of natural language and high level programming concepts that describes the main idea behind algorithm
  - More detail than algo, less than implementation
- Control flow
  - If … then …else
  - While-loop
  - for-loop
- Simple data structures
  - Array : A[i]; A[I,j]
- Methods
  - Calls: methodName(args)
  - Returns: return value

```
int arrayMax(int[] A, int n)
  Max=A.front();
  for i=1 to n-1 do
    if Max < A[i]
      then Max = A[i]
  return Max
```

# Pseudo-Code

- Mix of natural language and high level programming concepts that describes the main idea behind algorithm
  - More detail than algo, less than implementation
- Control flow
  - If … then …else
  - While-loop
  - for-loop
- Simple data structures
  - Array : A[i]; A[I,j]
- Methods
  - Calls: methodName(args)
  - Returns: return value

```
int arrayMax(int[] A, int n)
  Max=A[0]
  for i in range(0,n) do
   if Max < A[i]
     then Max = A[i]
  return Max
```

# Pseudo-Code

- Mix of natural language and high level programming concepts that describes the main idea behind algorithm
  - More detail than algo, less than implementation
- Control flow
  - If … then …else
  - While-loop
  - for-loop
- Simple data structures
  - Array : A[i]; A[I,j]
- Methods
  - Calls: methodName(args)
  - Returns: return value

```
int arrayMax(int[] A, int n)
  Max=A[0]
  for i=1 to n-1 do
    if Max < A[i]
      then Max = A[i]
  return Max
```

# Analysis of Algorithms

- Analyze time taken by Primitive Operations
- Low level operations independent of programming language
  - Data movement (assign..)
  - Control (branch, subroutine call, return…)
  - Arithmetic/logical operations (add, compare..)

- By inspecting the pseudo-code, we can count the number of primitive operations executed by an algorithm

# Example: Array Transpose

```
function Transpose(A[][], n)
  for i = 0 to n-1 do
    for j = i+1 to n-1 do
      tmp = A[i][j]
      A[i][j] = A[j][i]
      A[j][i] = tmp
    end
  end
end
```

| | j=0 | | | j=3 |
|---|---|---|---|---|
| i=0 | 0,0 | 0,1 | 0,2 | 0,3 |
| | 1,0 | 1,1 | 1,2 | 1,3 |
| | 2,0 | 2,1 | 2,2 | 2,3 |
| i=3 | 3,0 | 3,1 | 3,2 | 3,3 |

# Example: Array Transpose

```
function Transpose(A[][], n)
  for i = 0 to n-1 do
    for j = i+1 to n-1 do
      tmp = A[i][j]
      A[i][j] = A[j][i]
      A[j][i] = tmp
    end
  end
end
```

| | *j=0* | | | *j=3* |
|---|---|---|---|---|
| *i=0* | 0,0 | 0,1 | 0,2 | 0,3 |
| | 1,0 | 1,1 | 1,2 | 1,3 |
| | 2,0 | 2,1 | 2,2 | 2,3 |
| *i=3* | 3,0 | 3,1 | 3,2 | 3,3 |

Swap

Outer Loop

Estimated time for A[n][n] = (**n(n-1)/2**).(3+2) + 2.n

Inner Loop

*Is this constant for a given 'n'?*

# Asymptotic Analysis

- **Goal**: Simplify analysis of running time by getting rid of ‘details’ which may be affected by specific implementation and hardware
  - Like ‘rounding’: 1001 = 1000
  - $3n^2=n^2$
- How does the running time of an algorithm increase with the size of input in the limit?
  - Asymptotically more efficient algorithms are best for all but small inputs

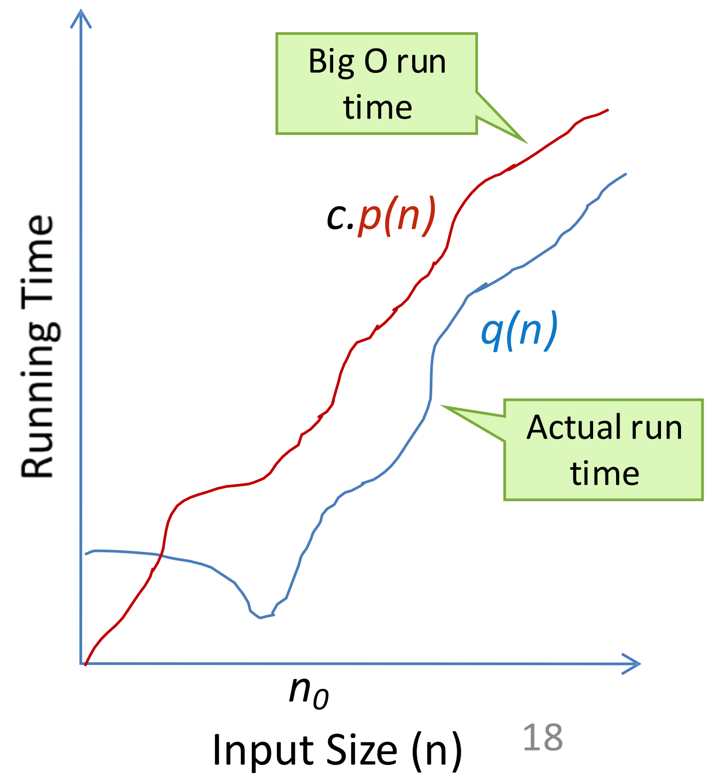
# Asymptotic Notation: "Big O"

**Definition** Let $p(n)$ and $q(n)$ be two nonnegative functions.

$p(n)$ is **asymptotically bigger** ($p(n)$ asymptotically dominates $q(n)$) than the function $q(n)$ iff

$$\lim_{n \to \infty} \frac{q(n)}{p(n)} = 0$$

- ***O*** Notation
  - Asymptotic **upper** bound
  - *q(n)*=***O***(*p(n)*), if there exists constants *c* and $n_0$, s.t.
    - $q(n) \leq c.p(n)$ for $n \geq n_0$
  - *q(n) and p(n) are functions over non negative integers*
- Used for *worst-case* analysis
  - p(n) is the **asymptotic upper bound** of actual time taken



# Asymptotic Notation

- **Simple Rule:** Drop lower order terms and constant factors
  - `(n(n-1)/2).(3+2) + 2.n` is $O(n^2)$
  - `23.n.log(n)` is **O(n.log(n))**
  - `9n-6` is **O(n)**
  - $6n^2.\log(n) + 3n^2 + n$ is $O(n^2.\log(n))$

# Asymptotic Notation

- **Simple Rule:** Drop lower order terms and constant factors
  - $(n(n-1)/2).(3+2) + 2.n$ is $\mathbf{O(n^2)}$
  - $23.n.\log(n)$ is $\mathbf{O(n.\log(n))}$
  - $9n-6$ is $\mathbf{O(n)}$
  - $6n^2.\log(n) + 3n^2 + n$ is $\mathbf{O(n^2.\log(n))}$
- **Note:** It is expected that the approximation should be as tight an order as possible (e.g., avoid giving a loose $O(n^3)$ upper bound for function $3n^2 + 7n + 42$)

**<u>Try yourself!</u>**

Plot the observed and asymptotic expected curves for a program

# Asymptotic Analysis of Running Time

- Use *O* notation to express number of primitive operations executed as a function of input size.
- Hierarchy of functions

$$1 < \log n < n < n^2 < n^3 < 2^n$$

$$\xleftarrow{\hspace{6em}}$$

*Better*

- Warning! Beware of large constants (say 1M).
  - This may have lower performance than one running in time $2n^2$, which is $O(n^2)$, for even modest input sizes

# Example of Asymptotic Analysis

- Input: An array X[n] of numbers.
- Output: An array A[n] of numbers s.t A[k]=mean(X[0]+X[1]+…+X[k-1])

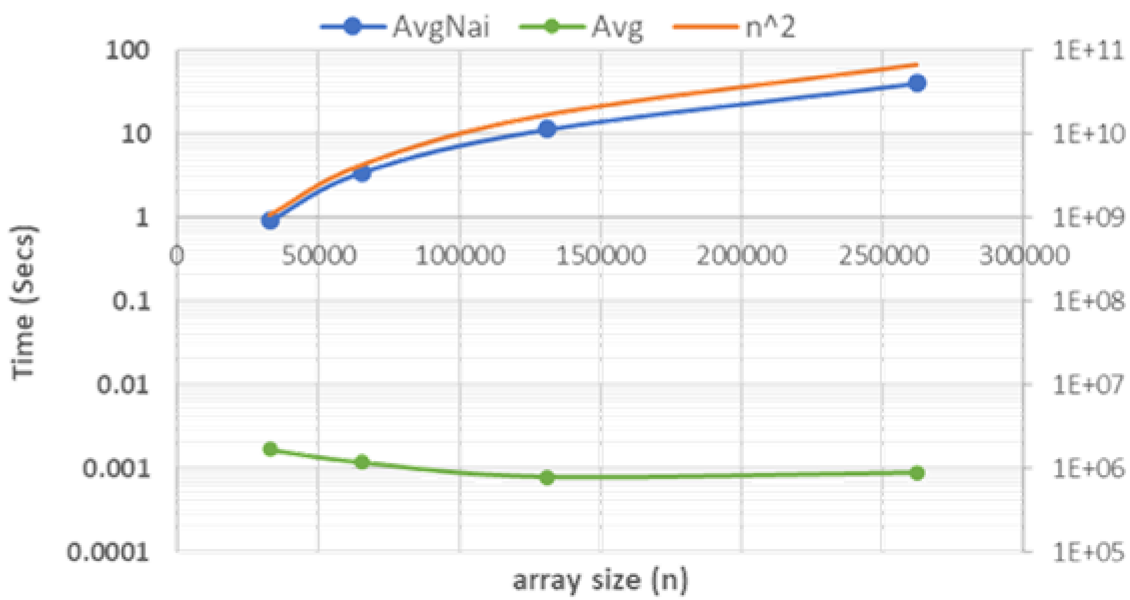
```
for i=0 to (n-1) do
  a=0
  for j=0 to i do
    a = a + X[j]
  end
  A[i] = a/(i+1)
end
return A
```

Analysis: running time is $O(n^2)$

- A naïve algorithm! *What's its complexity?*

# A Better Algorithm?

```
s=0
for i=0 to n do
   s = s + X[i]
   A[i] = s/(i+1)
end
return A
```




Time take by previous naive algorithm (O(n^2)) and current algorithm (O(n)). Orange line in secondary Y axis indicates n^2 line. The blue line matches the orange line. The green line should match O(n) but since time is small, there may be measurement errors.

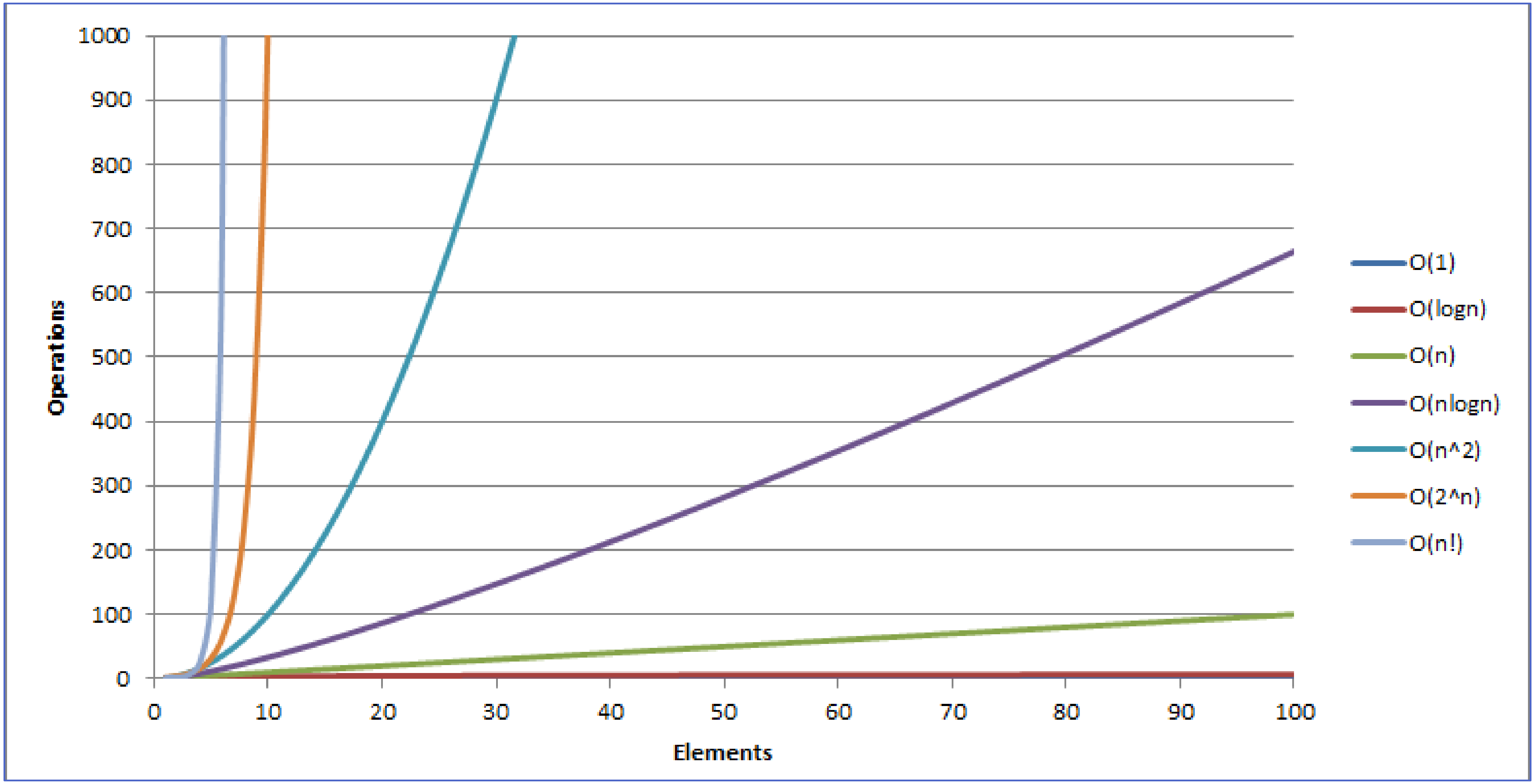
Analysis: running time is *O*(n)

# Comparison

| Logarithmic | Linear | Iterated logarithmic | Quadratic | Polynomial ($n^{\alpha}$) *$\alpha$ is a const. > 1* | Exponential ($\alpha^{n}$) *$\alpha$ is a const. > 1* |
|---|---|---|---|---|---|
| log n | n | n log n | $n^2$ | $n^3$ | $2^n$ |
| 0 | 1 | 0 | 1 | 1 | 2 |
| 1 | 2 | 2 | 4 | 8 | 4 |
| 2 | 4 | 8 | 16 | 64 | 16 |
| 3 | 8 | 24 | 64 | 512 | 256 |
| 4 | 16 | 64 | 256 | 4096 | 65536 |
| 5 | 32 | 160 | 1024 | 32768 | 4294967296 |

*(solvable in polynomial time)*

# Comparison




https://stackoverflow.com/questions/4317414/polynomial-time-and-exponential-time

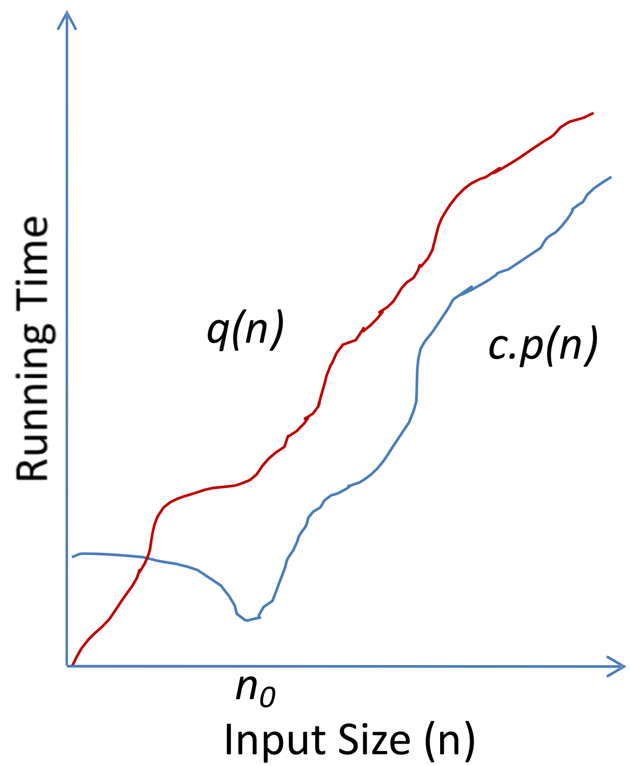
# Tasks

- Self study (Sahni Textbook)
  - Chapter 3 & 4 “Asymptotic Notation” & “Performance Measurement”
- Try the mean calculation code in C++ yourselves

# Asymptotic Notation: Lower Bound

- The "big-Omega" Ω notation
  - asymptotic *lower* bound
  - q(n) = **Ω**(p(n)) if there exists const. c and $n_0$ s.t.
    - **c.p(n) ≤ q(n)** for $n \geq n_0$
- Tells you: the algorithm will take at least this much time for large n (up to constant factors)
- Can also be useful to give a lower bound for the entire class of algorithms

(e.g., searching in an unsorted array is Ω(n), that is, no algorithm can offer faster worst-case time complexity)



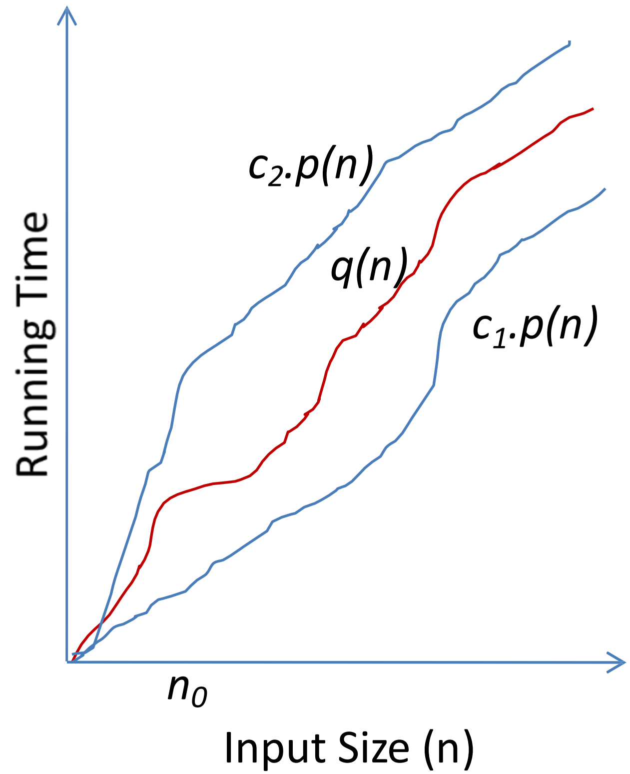
# Asymptotic Notation: Tight Bound

- The "big-Theta" θ-Notation
  - Asymptotically tight bound
  - q(n) = **θ**(p(n)) if there exists constants $c_1$, $c_2$ and $n_0$ s.t.

**$c_1$ p(n) ≤ q(n) ≤ $c_2$ p(n)** for $n \geq n_0$

- *q(n) = θ (p(n))* **if and only if**

*q(n)=O(p(n))* and *q(n)=Ω(p(n))*



# Asymptotic Notation

- Analogy with real numbers
  - $q(n) = O(p(n))$ → $q \leq p$
  - $q(n) = \Omega(p(n))$ → $q \geq p$
  - $q(n) = \theta(p(n))$ → $q \approx p$

# Analyze the runtime (In-class exercise)

```
void mystery(int n) {
    count = 0;
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= i*i; j++) {
            if (j % i == 0) {
                count++;
            }
        }
    }
}
```

Say $q(n)$ denotes the worst-case runtime of this algorithm

$q(n) = O(p_1(n))$. Specify $p_1(n)$

$q(n) = \Omega(p_2(n))$. Specify $p_2(n)$

$q(n) = \theta(p_3(n))$. Specify $p_3(n)$

# Polynomial and Intractable Algorithms

- **Polynomial Time complexity**
  - An algorithm is said to be polynomial if it is $O(n^{\alpha})$ for some integer $\alpha$
  - Polynomial algorithms are said to be efficient
    - They solve problems in "reasonable" times
- ***Intractable Algorithms***
  - Algorithms for which there is no *known* polynomial time algorithm

# Complexity: List using Arrays

- **Storage Complexity**: Amount of storage required by the data structure, relative to items stored
- List using Array: …
- **Computational Complexity**: Number of CPU cycles required to perform each data structure operation
- size(), set(), get(), indexOf()

# Complexity: List using Linked List

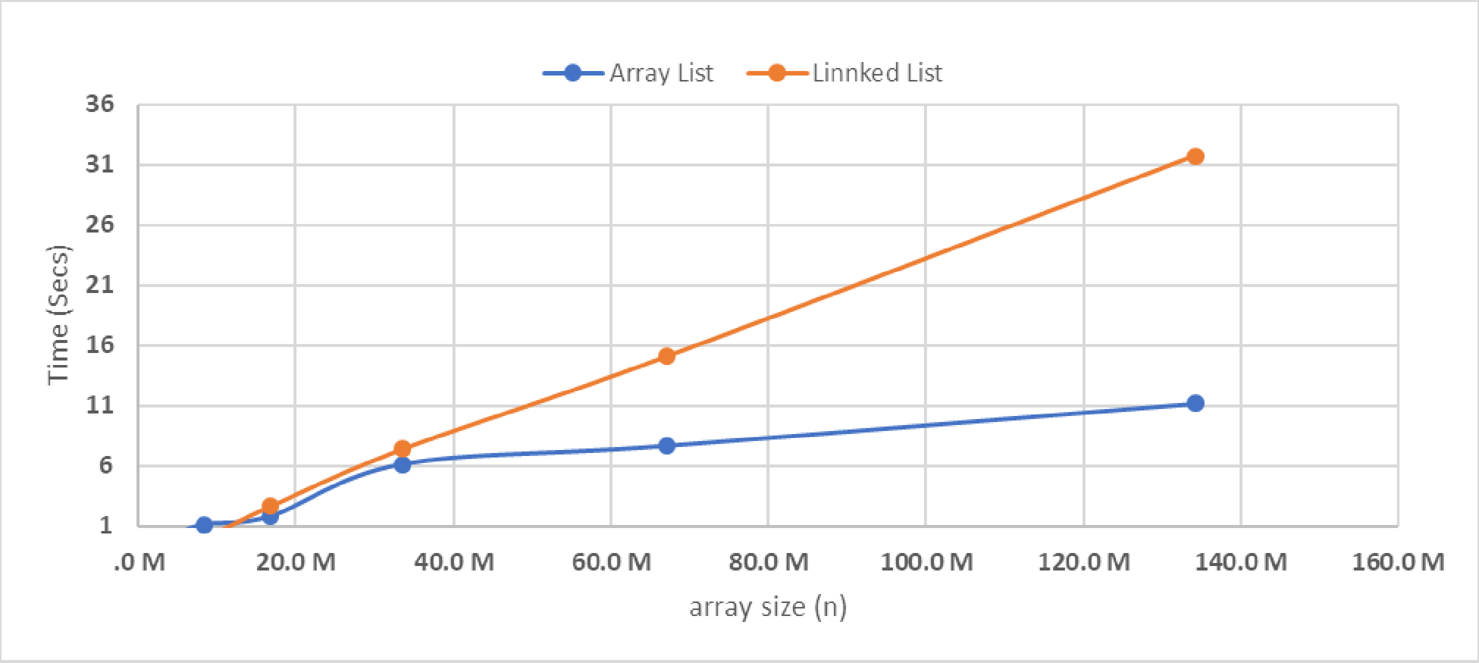
- Storage Complexity
  - Only store as many items as you need
  - But…
- Computational Complexity
  - set(), get(), remove()
  - indexOf()
- Other Pros & Cons?
  - Memory management, mixed item types

# Empirical Validation

- How do I check if complexity analysis matches reality?
  - Timing
    - Static overheads
    - Asymptotic behaviour
    - Locality effects
    - Disk Thrashing
  - Memory used
    - Adding up variable sizes, reference pointer sizes, “sizeof”
    - Deep vs. shallow size
    - Effect of padding for struct to align with word length/cache line
  - Profiling: CPU used, top, cache hits/misses, iops, context switching

# Perf of Array, LinkedList

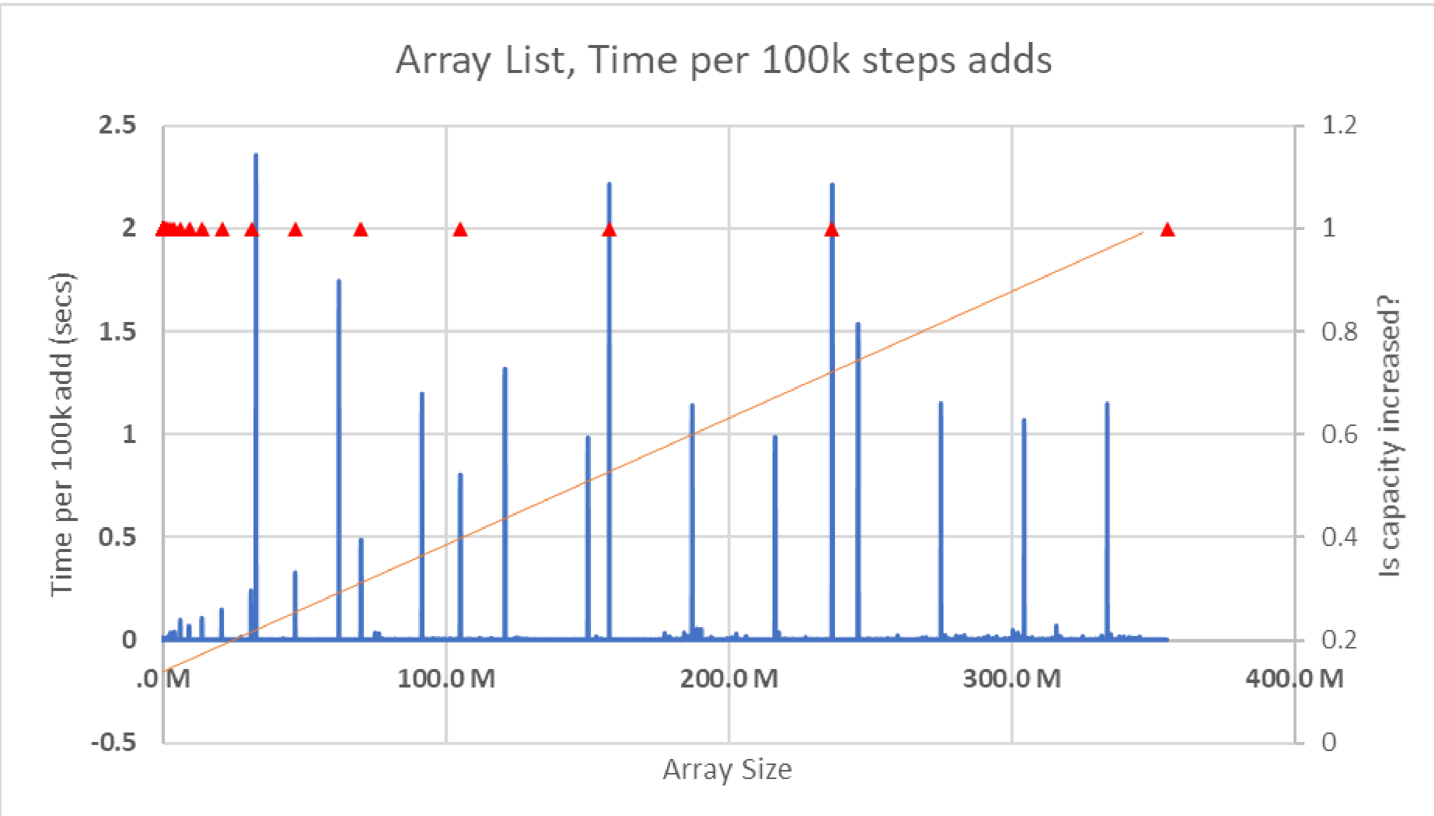
- Time to insert into array
  - Time to copy from one array to larger, on resize?
- Time to insert into linked list




Time to insert *n* items into a list. X axis is "n".
We are doing an append to the end. So linked list and array implementations have O(n) time.
But time to allocate memory for an item is higher in LL than time to set allocated array location.

# Perf of Array List

- Array Copy Times on growth




Time to insert *100k items* incrementally into a single array list. X axis is the number of inserted so far.
We expect each 100k insertion to take constant time.
Some spikes indicate array capacity being full and reallocation/ moving of prior data.
The red dots show where the capacity may have been increased based on implementation logic.

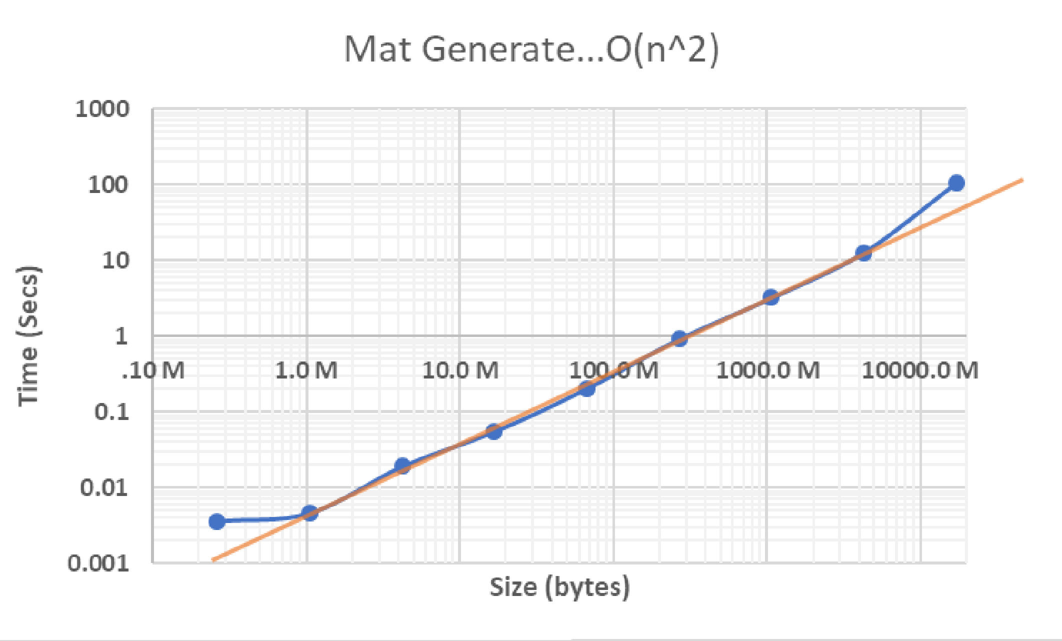
# Complexity of Matrix Ops

- Generating a 2D matrix with n x n elements

$O(n^2)$

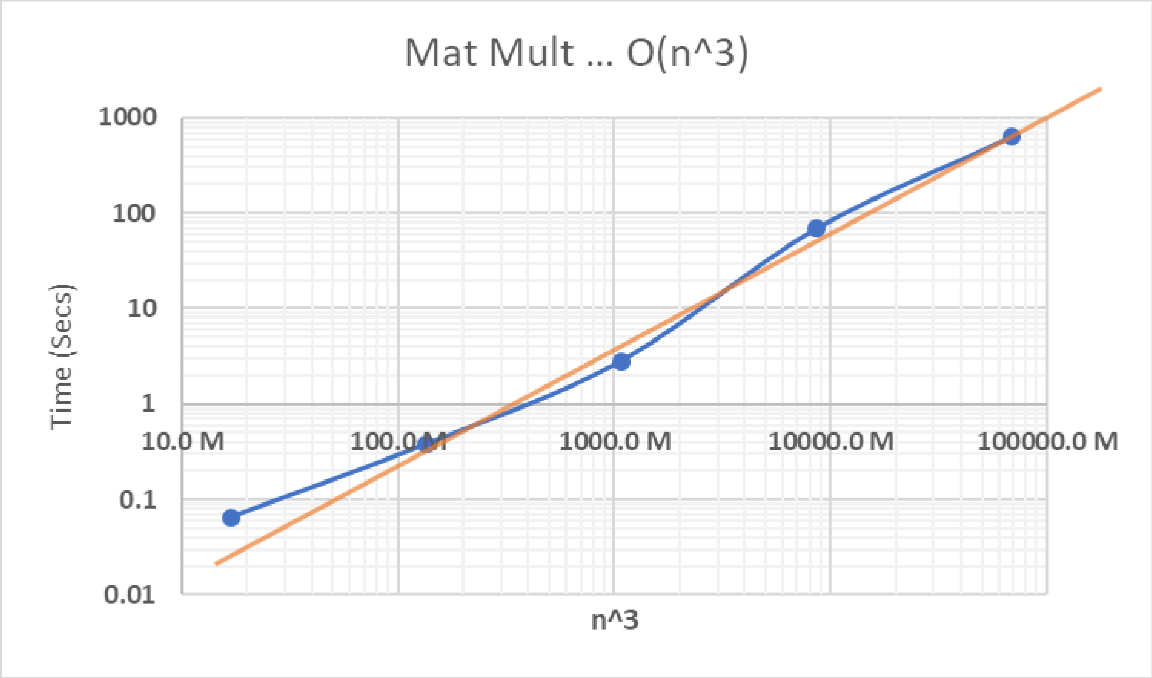
```
function MatMult(A[][], B[][], n)
  for i = 0 to n-1 {
    for j = 0 to n-1 {
      sum=0
      for k = 0 to n-1 {
        sum = sum + A[i][k]*B[k][j]
      }
      c[i][j] = sum
    }
  }
```

$O(n^3)$




Time to generate and set values in matrix of size nxn. X axis is number of elements in matrix. Orange line is a linear trend line.

Time to multiply two matrices of size nxn. X axis is n^3. Orange line is a linear trend line.



# Tasks

- Self study (Sahni Textbook)
  - Chapter 3 & 4 “Asymptotic Notation” & “Performance Measurement”
- Try the code in C++ yourselves
- Assignment 1
  - Due on **20 September**
  - No extension will be granted
  - 20% weightage
  - Get started early to avoid any last day hiccups
  - Access to teaching cluster- soon
- 1st tutorial on Aug 29 (Friday)