DS221

Data Structures, Algorithms & Data Science Platforms

Instructor: Chirag Jain
(slides from Prof. Simmhan)

CDS
The Department of Computational and Data Science

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
- …for a given input size
- Later, I/O and Communication complexity
- *Why are these important?*

Space Complexity

- *Estimate of the amount of peak memory required for an algorithm to run to completion, for a given input size*
    - Core dumps/OOMEx: Memory required is larger than the memory available on a given system
- Some algorithms may be more efficient if data completely loaded into memory
    - Need to look also at system limitations

Space Complexity

- **Fixed part**: The size required to store certain data/ variables, that is independent of the size of the problem:
  - e.g., simple variables and constants used
  - e.g., program/code size
- **Variable part**: Space needed by variables, whose size is dependent on the size of the problem:
  - e.g., dynamic memory allocation
  - e.g., recursion stack space

Try yourself!

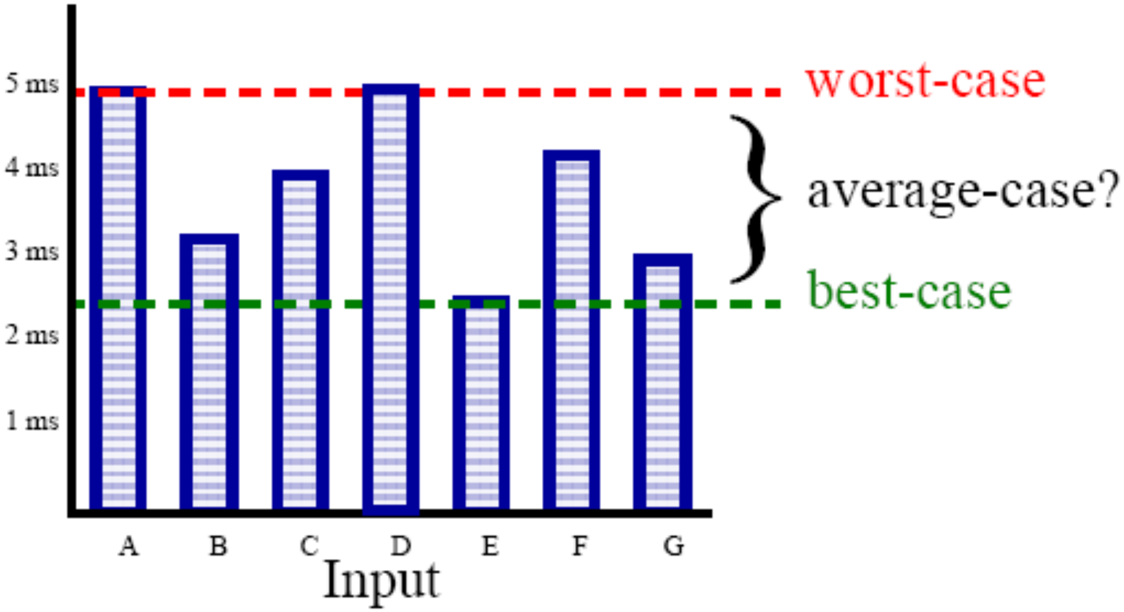
For some program with variable input sizes, find the space taken by the fixed part and the variable part, using **top** command

Analysing Running Time Empirically

- Write program
- Run Program
- Measure actual running time with some methods like *System.currentTimeMillis(), gettimeofday()*

- *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 detailed than algorithm, but less than actual 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
```

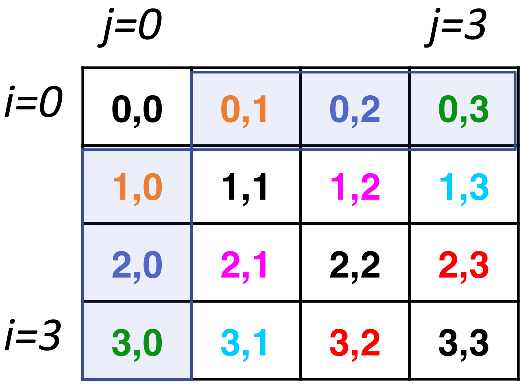
Analysis of Algorithms

- Analyse 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

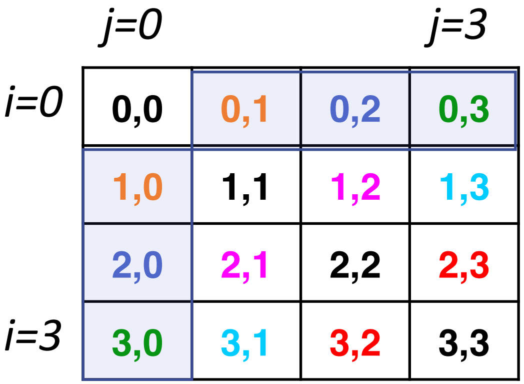
```
function **Transpose**(A[][], n)
    for i = 0 to n-1 do
        for j = i+1 to n-1 do
            tmp = A[i][j]
            A[i][j] = A[j][i]
            A[j][i] = tmp
        end
    end
end
```



	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

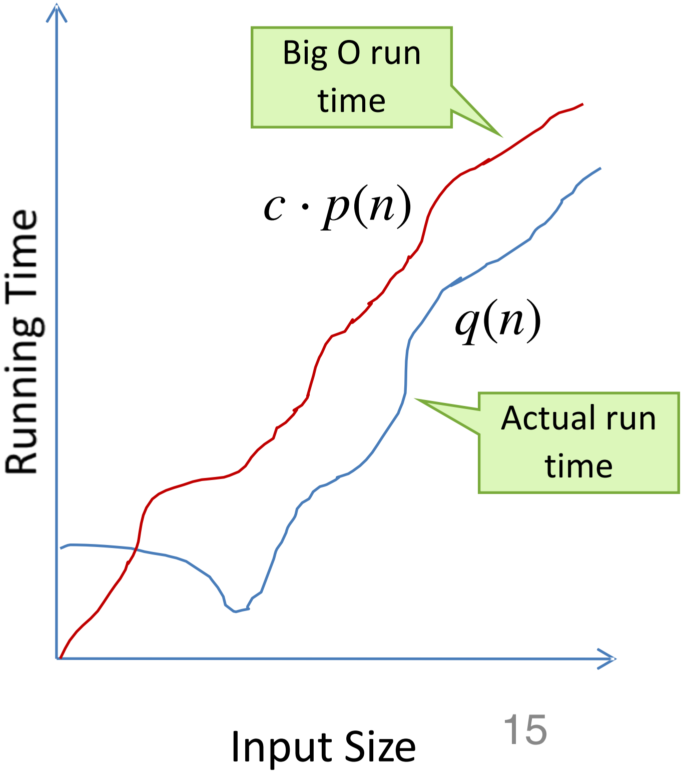
- **Goal**: Simplify analysis of running time by getting rid of ‘details’ which may be affected by specific implementation and hardware
  - Like ‘rounding’: 1001 = 1000
  - $3n^2=n^2$
- How does the running time of an algorithm increase with the size of input in the limit?
  - Asymptotically more efficient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

Question

- For a given problem with input size n, I’ve designed an algorithm whose worst-case time complexity is O(n)
- What does this mean?

Asymptotic Notation: “Big O”

- O Notation
  - $q(n) = O(p(n))$, if there exists constants c and n_0, s.t.
    - $q(n) \leq c \cdot p(n)$ for all $n \geq n_0$
  - $q(n)$ and $p(n)$ are functions over non negative integers
  - Asymptotic upper bound: Function q grows at most as fast as function p asymptotically (up to a constant factor)



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 $\mathbf{O(n^2.\log(n))}$

<u>Try yourself!</u>

Plot the observed and asymptotic expected curves for a program. Is the upper bound rule met?

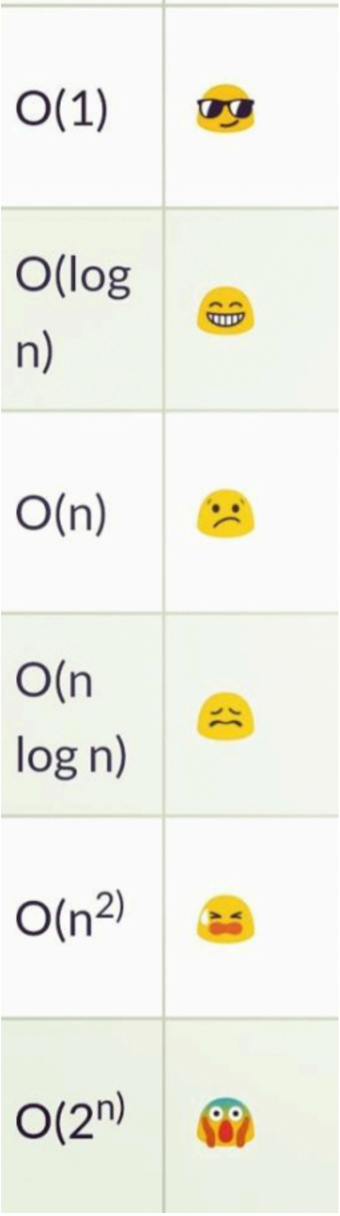
Asymptotic Analysis of Running Time

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

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

← *Better*

- Warning! Beware of large constants (say 1M).
  - *O*(n) algo with very large constant may have lower performance than *O*(n^2) algo for modest input sizes



O(1)	😎
O(log n)	😁
O(n)	🙁
O(n log n)	😣
O(n^2)	😫
O(2^n)	😱

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 $A$
```

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

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?*

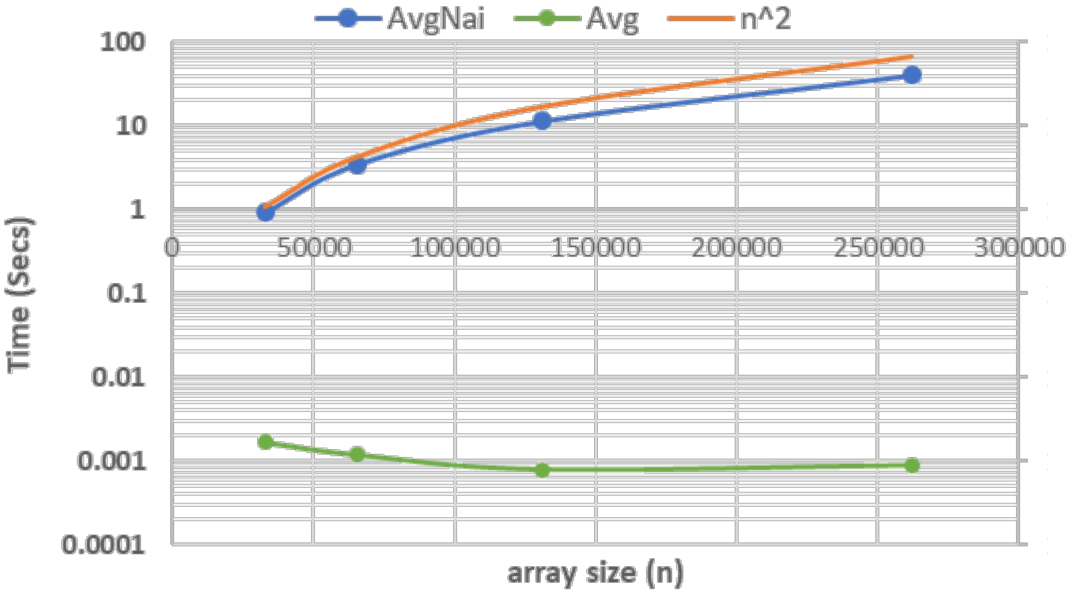
A Better Algorithm?

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

A Better Algorithm?

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

Analysis: running time is *O*(n)

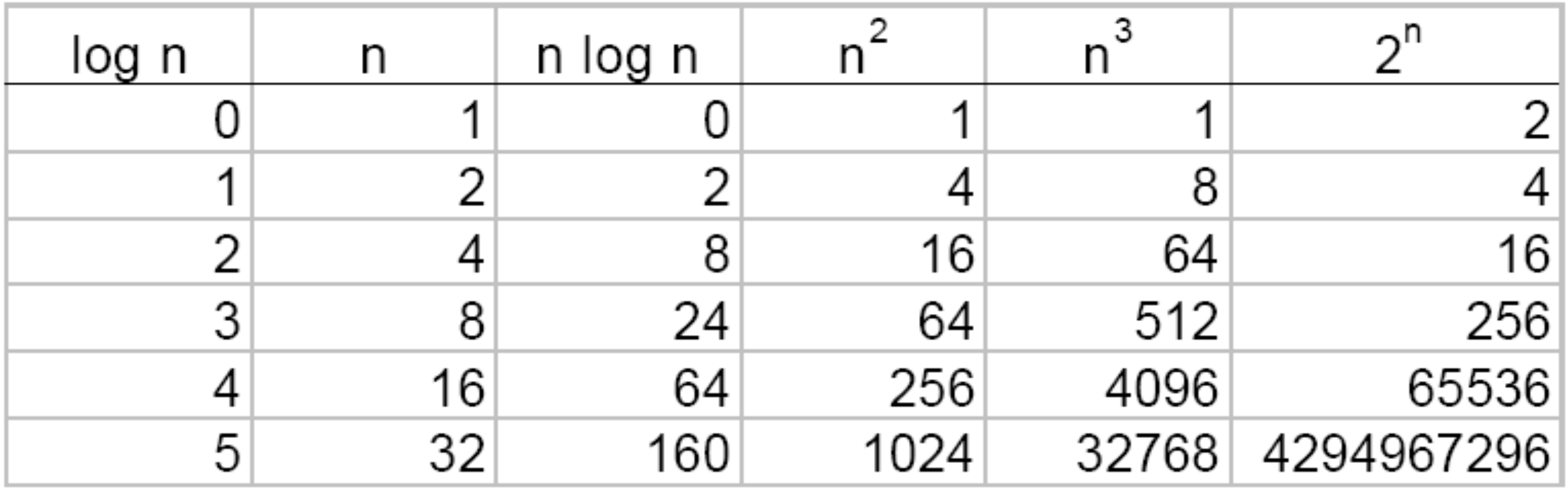


Time take by previous naive algorithm (O(n^2)) and current algorithm (O(n)). Orange line in secondary Y axis indicates n^2 line. The blue line matches the orange line. The green line should match O(n) but since time is small, there may be measurement errors.

Asymptotic Notation

- Terminology
    - Logarithmic $O(\log n)$
    - Linear $O(n)$
    - Quadratic $O(n^2)$
    - Polynomial $O(n^k)$, $k > 1$
    - Exponential $O(a^n)$, $a>1$

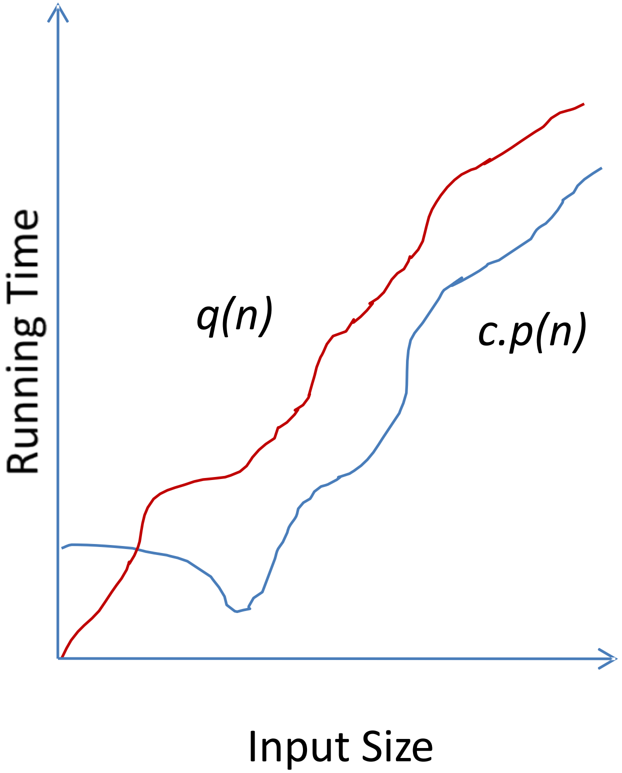
Comparison



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

Asymptotic Notation: Lower Bound

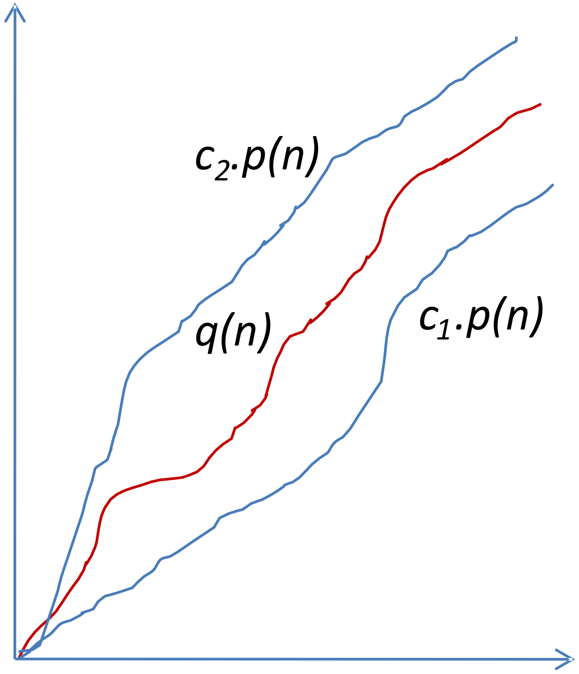
- The “big-Omega” Ω notation
  - $q(n) = \Omega(p(n))$ if there exists const. c and n_0 s.t.
    - $q(n) \geq c \cdot p(n)$ for all $n \geq n_0$
  - Asymptotic lower bound: Function q grows at least as fast as function p asymptotically (up to a constant factor)



Asymptotic Notation: Tight Bound

- The "big-Theta" θ-Notation
  - q(n) = θ(p(n)) if there exists consts. c_1, c_2 and n_0 s.t. **c_1 p(n) ≤ q(n) ≤ c_2 p(n)** for n ≥ n_0
  - Function *q* grows about as fast as function *p* asymptotically
- *q(n)* = θ *(p(n))* **iff**

q(n)=O(p(n)) and *q(n)=Ω(p(n))*



Input Size

Asymptotic Notation

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

Polynomial and Intractable Algorithms

- **Polynomial-time complexity**
  - An algorithm is said to be polynomial if it is $O(n^d)$ for some integer d
  - Polynomial algorithms are said to be efficient
    - They typically solve problems in reasonable times!
- **Intractable problems**
  - Problems 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(), get(), indexOf()

Complexity: List using Linked List

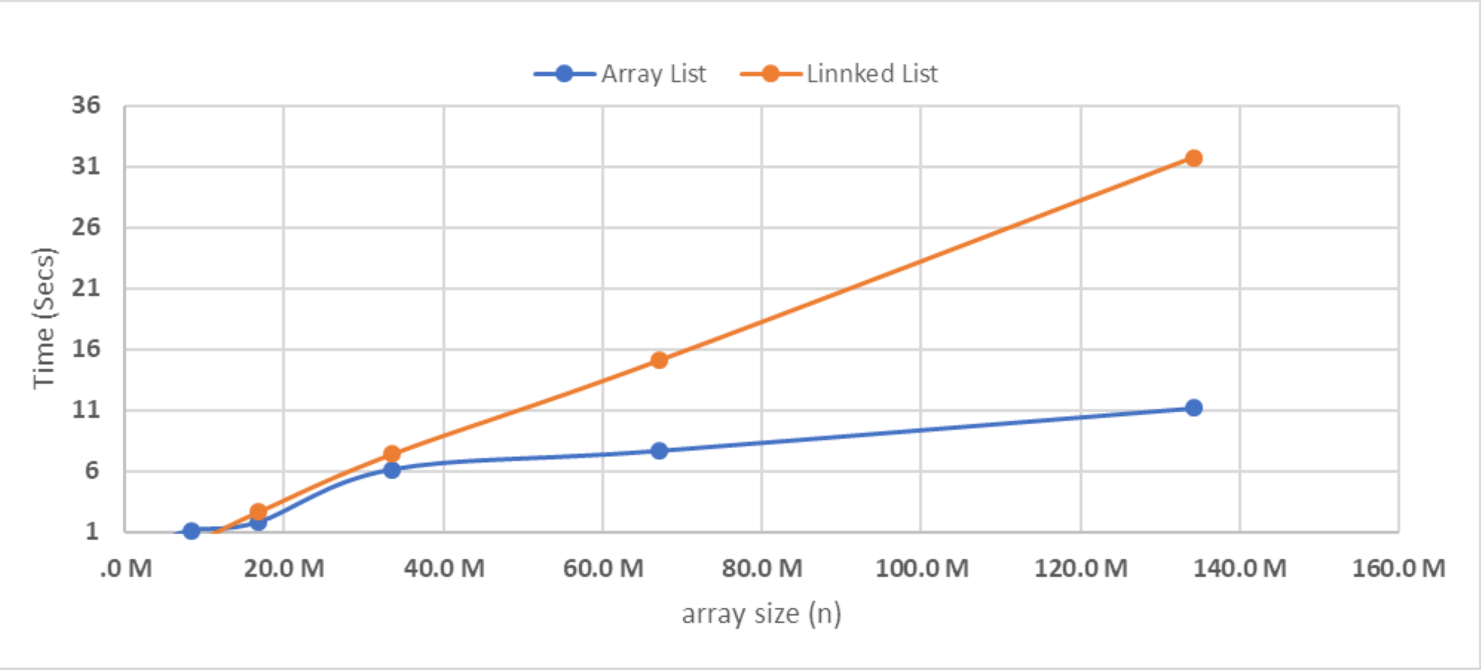
- Storage Complexity
  - Only store as many items as you need
- Computational Complexity
  - get(), remove()
  - indexOf()
- Other Pros & Cons?
  - Memory management, mixed item types

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

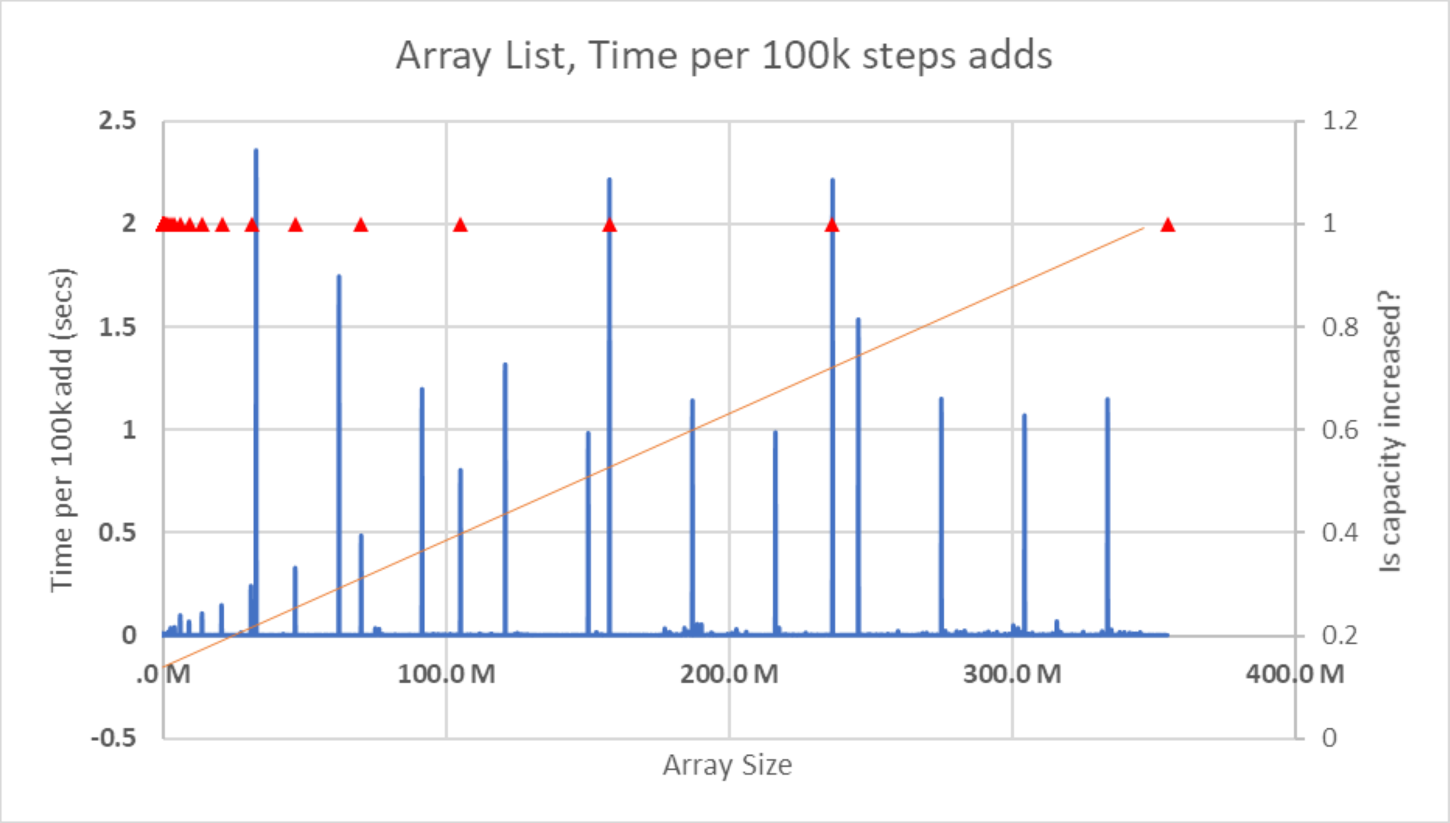
- Time to insert into array
- Time to insert into linked list



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, LinkedList

- 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.

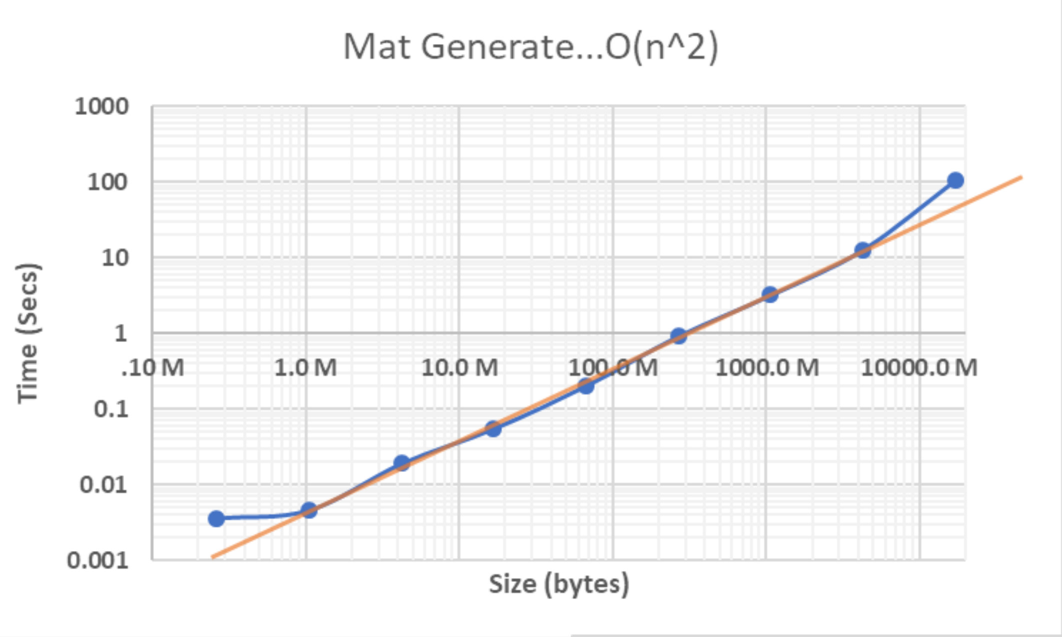
Complexity of Matrix Ops

- Initialise a 2D matrix with n x n elements

$O(n^2)$

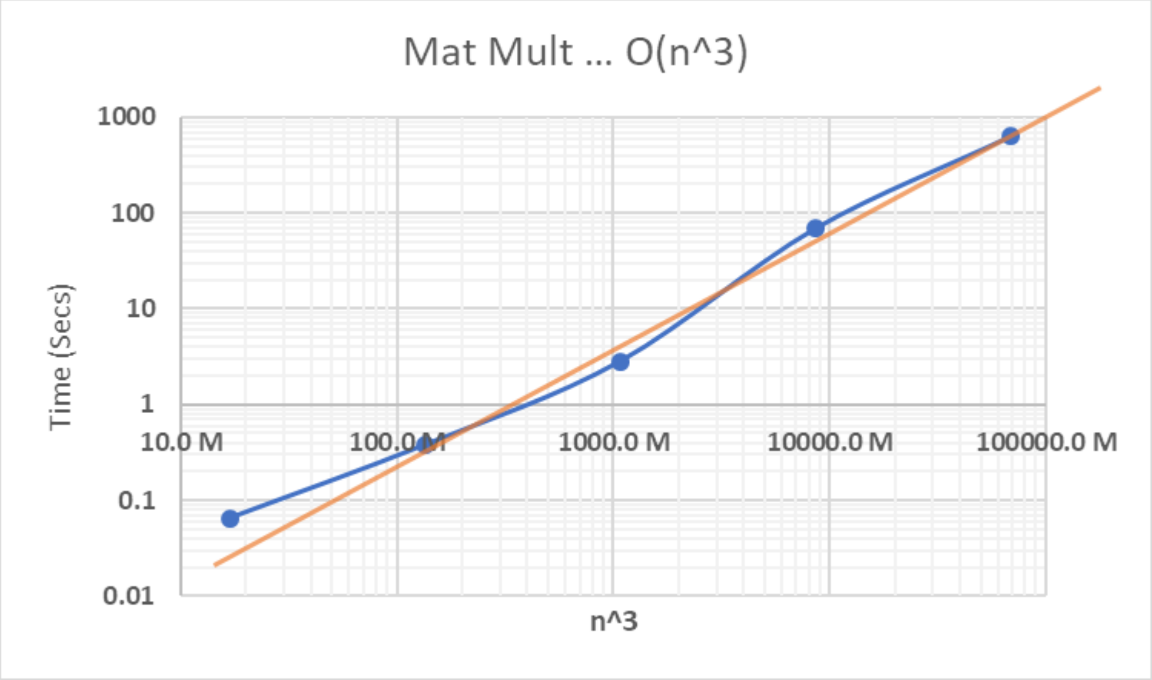
```
function MatMult(A[][], B[][], n)
    for i = 0 to n-1 {
        for j = 0 to n-1 {
            sum=0
            for k = 0 to n-1 {
                sum = sum + A[i][k]*B[k][j]
            }
            c[i][j] = sum
        }
    }
```

$O(n^3)$



Time to generate and set values in matrix of size nxn. X axis is number of elements in matrix (n^2). 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

Tutorials/Special Class

- Monday 5-6 pm OR 10 am - 11pm?
  Tutorial on C++ STL, Debugging using GDB, Makefile
- **Next to next week**: Tutorial on Git

Thank you