بِسْمِ ٱللَّهِ ٱلرَّحْمَٰنِ ٱلرَّحِيمِ

In the name of Allah, Most Gracious, Most Merciful

CSE 4303
Data Structure

Topic: Introduction to data structures, Complexity Time-Space Tradeoff

Asaduzzaman Herok
Lecturer | CSE | IUT
asaduzzaman34@iut-dhaka.edu

What is Data Structure?

Data: Simply values or sets of values, raw facts or figure without any specific meaning.

Data Structure: The logical or mathematical model of a particular organization of data.

- ➢ Can store data
  - Example: Integers, Strings, Floats, … …
- ➢ Can answer some questions about the stored data
  - Example: What is the smallest value not greater than x?
- ➢ Can add or remove data
  - Example: add the element x after y, remove values less than x.

Why Study Data Structure?

Applications of Data Structure:

- Computer file system (Data structure maps file names onto hard drive sectors)
- Google and other search engines (Data structure maps keywords on web pages containing those keywords)
- What is the longest common subsequence of two DNA can be found?
- Geographic systems (Data structure find data relevant to the current view/location)
- Finding large Prime Numbers
- Block chain (Linked list)
- Google Map (Finding shortest distances in terms of distance and time)
- Data Compression (Huffman’s encoding)
- Natural Language Processing (Strings)
- … … … … …

Many problems are solved efficiently just using the right data structure …

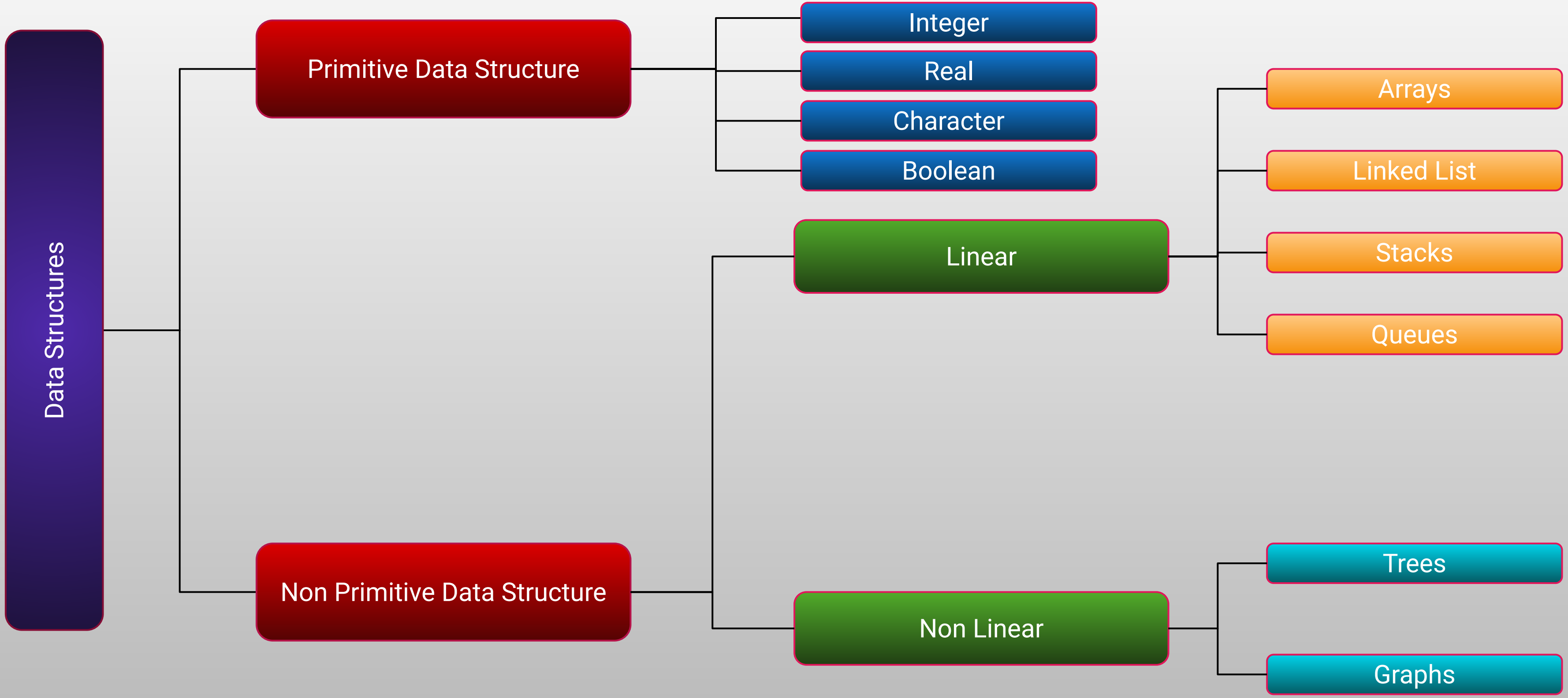
How do We Study Data Structures?

- What does the data structure represents?
  Computer file system (data structure maps file names onto hard drive track and sectors)

- What are the operations does it supports?
  - Reading: looking something up at a particular spot within the data structure.
  - Searching: looking for a particular value within a data structure.
  - Inserting: adding a new value to the data structure.
  - Deleting: removing a value from the data structure.
  - Sorting: rearranging element in some logical order.
  - Merging: Combining records of two different sorted files into one sorted files.

- What kind of performance does it have?
  - How long does each operation take? (Time complexity)
  - How much space does it use? (Memory complexity)

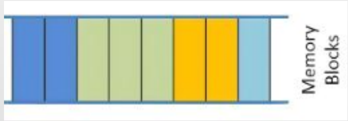
Classification of Data Structure



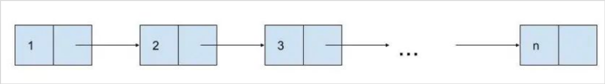
Memory Allocation

Memory allocation can be classified into followings:

- Contiguous
  - Example: arrays



- Linked
  - Example: linked lists

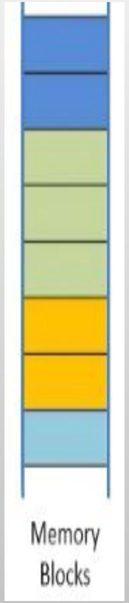


- Indexed
  - Example: array of pointers.

Contiguous Memory Allocation

An array stores n objects in a single contiguous space of memory.

- ➔ Can directly access any point randomly. Random access is possible.
- ➔ Unfortunately, if more memory is required, a request for new memory usually requires copying all information into the new memory.
- ➔ In general, you cannot request for the operating system to allocate to you the next n memory locations

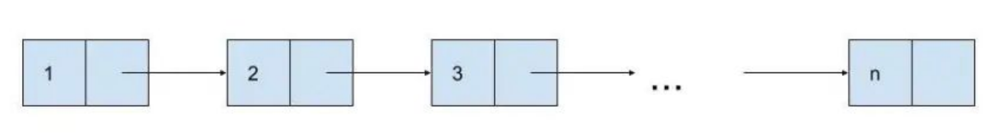


Linked Memory Allocation

Linked storage such as a linked list associates two pieces of data with each item being stored:

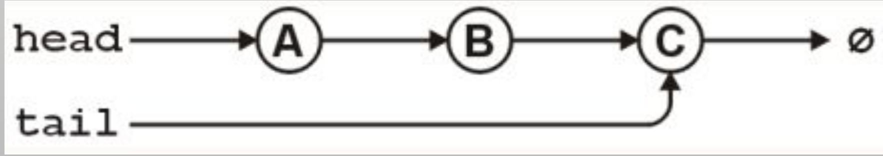
- ➔ The object itself, and
- ➔ A reference to the next item

- ➔ Random access to any data apart from the beginning is not possible since the address of a particular data is only stored to its previous data.



The actual linked list class must store two pointers

- ➔ A head and tail:
  - ◆ Node *head;
  - ◆ Node *tail;

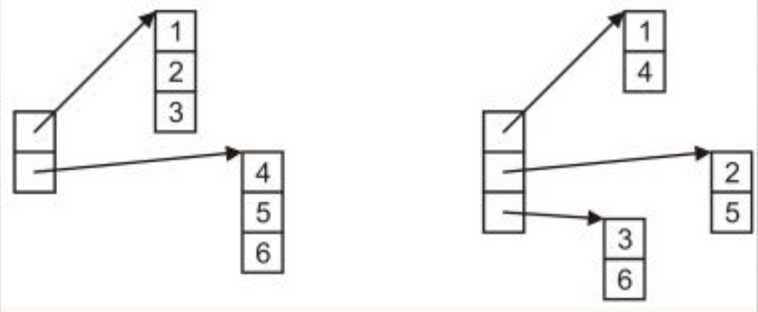


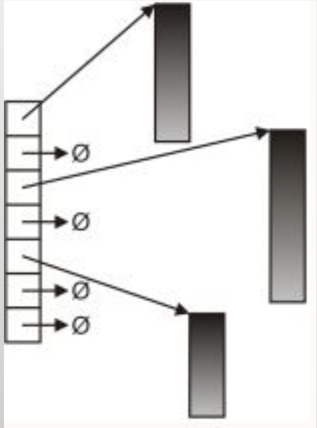
Indexed Memory Allocation

With indexed allocation, an array of pointers (possibly NULL) link to a sequence of allocated memory locations.

Matrices can be implemented using indexed allocation:

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$



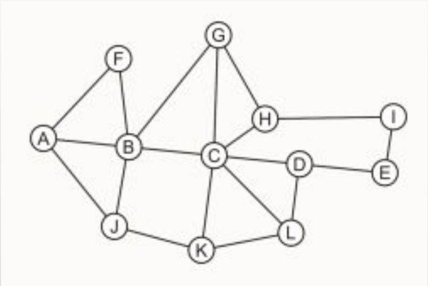


Other Memory Allocations

We will look at some varieties or hybrids of these memory allocations including:

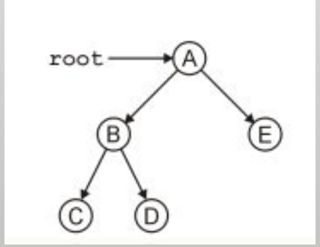
- ➔ Trees
- ➔ Graphs

Arbitrary relations among the objects in a container

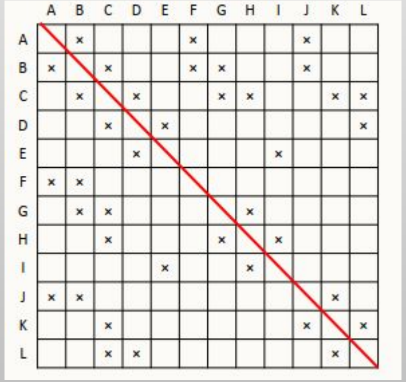


Graph

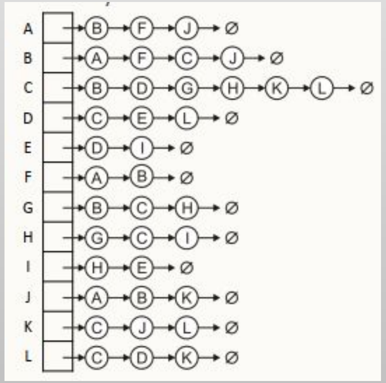
A rooted tree is similar to a linked list but with multiple next pointers



Tree



adjacency matrix



adjacency list

Complexity, Time-space tradeoff

- A function that estimates the running time/space with respect to the input size.
- Less time and space requirement is a blessing!
- Deals with large input size.
- Tradeoff: Increased amount of space to store data can sometimes reduce time requirement (or vice-versa).

Why Do We Care?

solution#1

```
for i=2 to n-1
   if i divides n
        n is not a prime
```

$(n-2)$ divisions in worst case

solution#2

for i=2 to $\sqrt{n}$
   if i divides n
        n is not a prime

$(\sqrt{n}-1)$ divisions in worst case

Complexity, Time-space tradeoff

Assuming 1 ms to perform a division		
	<u>Solution #1</u>	<u>Solution#2</u>
n=11	9 ms	~2 ms
n=101	99 ms	~9 ms
n=1000003 =10^6+3	~10^6 ms =1000 sec =16.66min	~10^3 ms = 1sec
n=10^10	10^10 ms =10^7sec =115 days	~10^5 ms = 100sec = 1.66 mins

Complexity, Time-space tradeoff

Two functions plotted in this graph:

$\boldsymbol{f(x) = x}$ (red)

$\boldsymbol{f(x) = \sqrt{x}}$ (blue)

Blue function is a bit costly in the beginning, but cheaper as x increases.

Time Complexity Analysis

Measures how fast the time requirement of a program grows when the input size increases.

Running time of program may depend on:

- ➔ Single vs multi processor
- ➔ Read/write speed of memory
- ➔ 32-bit or 64-bit
- ➔ Size of input

For time complexity analysis, we are only interested in (size of input)

- Takes same amount of time regardless of input size
- Constant time algorithm
- Time Complexity O(1)

```
Sum(a,b) {
  return a+b
}
```

Let's think about this function

Time requirement: ~ 2 time-units (1 unit for addition, 1 unit for return statement)

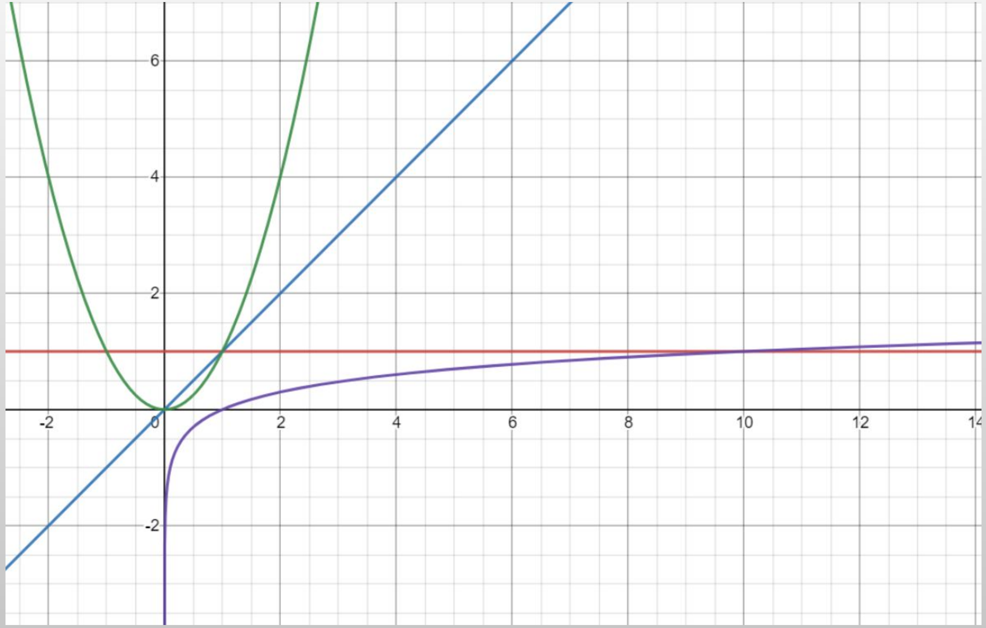
Time Complexity Analysis

	# times	Cost unit
1. sumOfList (A, n) {		
2. total=0	1	1 (c1)
3. for i=0 to n-1	n+1	2 (c2)
4. total = total + A[i]	n	2 (c3)
5. return total	1	1 (c4)
6. }		

Comments
<u>In line 3:</u> - Executes n+1 times. One extra checking for breaking condition. - c2: 1 unit for increment, 1 unit for assignment. <u>In line 4:</u> - 1 unit for addition, 1 for assignment.

- $T(n) = 1 + 2(n+1) + 2n + 1 = 4n + 4$
- In other words, $\mathrm{T(n)} = c_1 + c_2(n+1) + c_3 n + c_4 = \boldsymbol{cn + c'}$
  - here $(c = c_2 + c_3\,, \& \; c' = c_1 + c_3 + c_4)$
- Don't care much about value of $c \; or \; c'$, focus on the rate of growth.
- Here the growth is linear. Termed as ***O(n)***, AKA 'Big-oh of n' AKA 'Order of n' .

Some Growth Functions



$f(x) = 1$ (red),

$f(x) = x$ (blue),

$f(x) = x^2$ (green),

$f(x) = log x$ (purple)

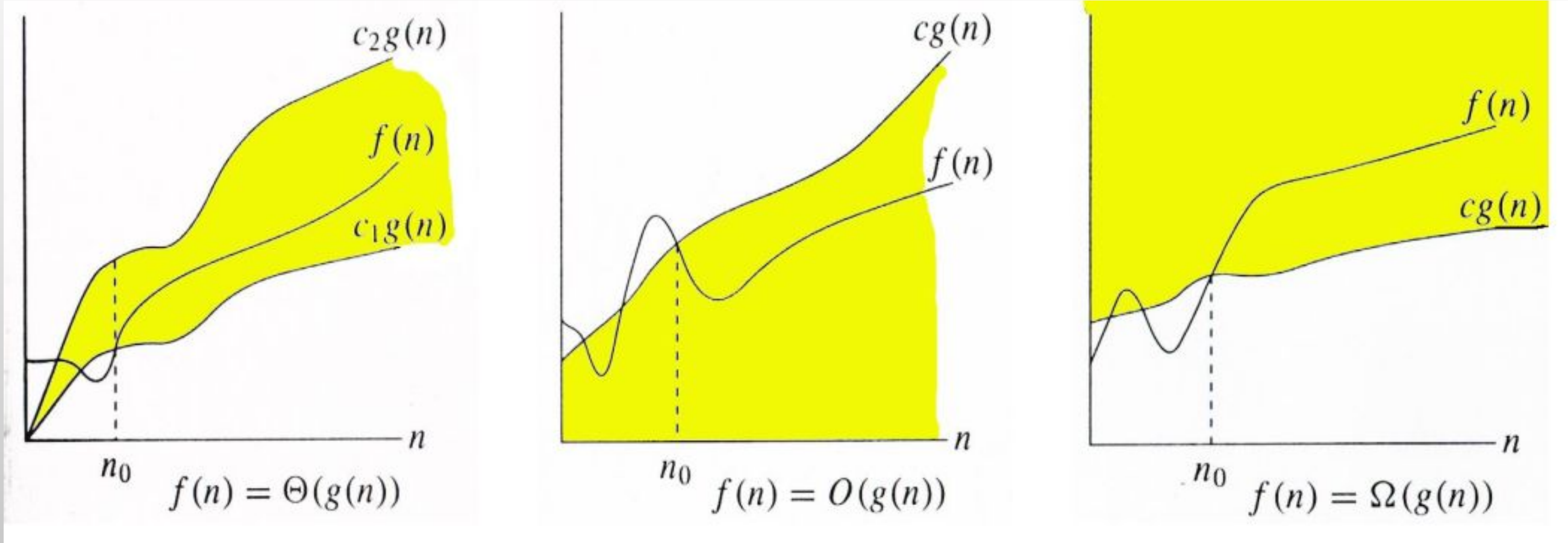
Check the growth of function as values in x axis grows!

Asymptotic Analysis

- Asymptotic Analysis is the big idea that helps to analyze algorithms.
- In Asymptotic Analysis, we evaluate the performance of an algorithm in terms of input size (we don’t measure the actual running time).
- Define mathematical bound of how the time (or space) taken by an algorithm increases with the input size.
- An algorithm that is asymptotically more efficient will be the best choice for all but very small inputs.

[Generally, the term ‘asymptotic’ means approaching but never connecting with a line or curve.]

Asymptotic Analysis



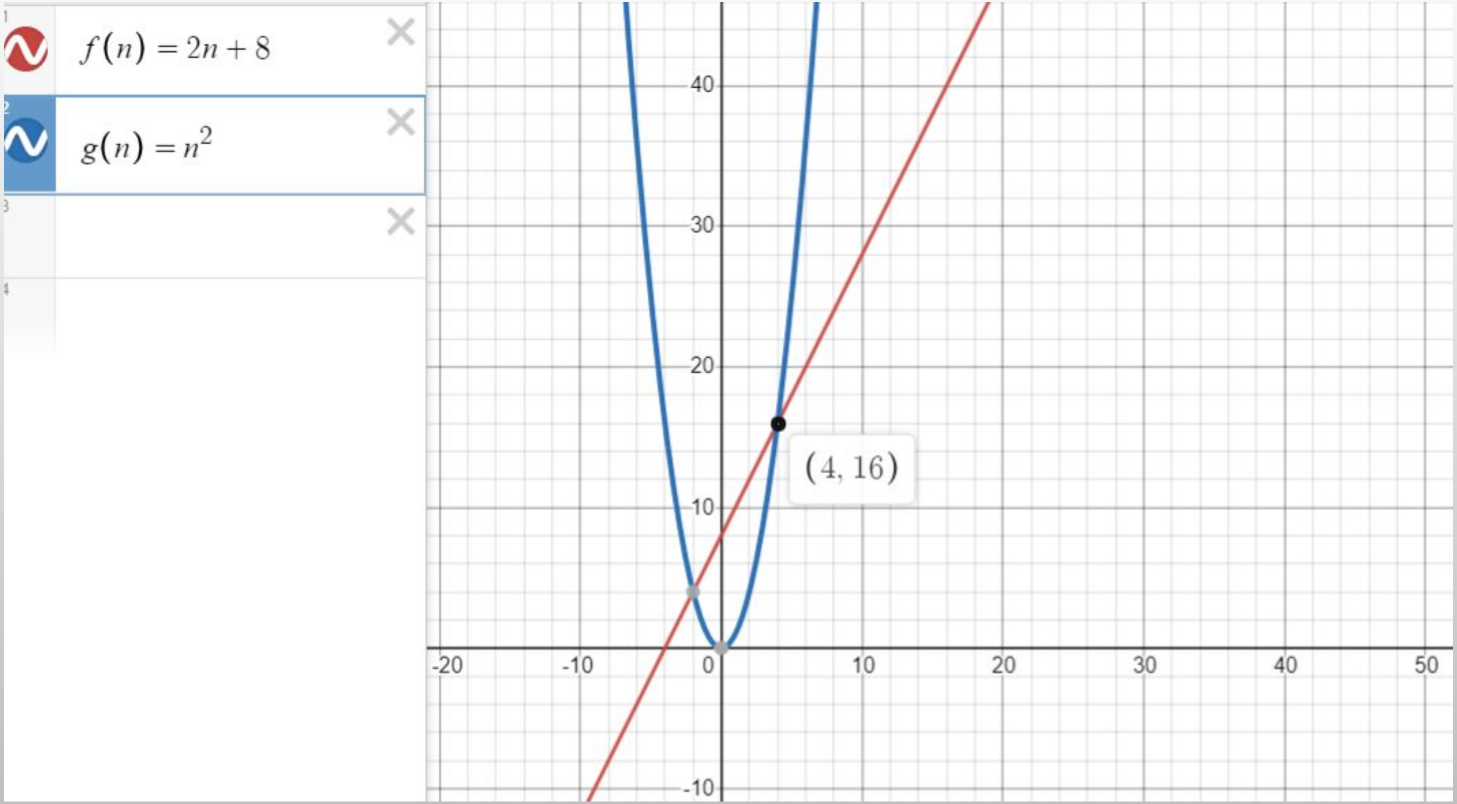
The Big 'O'

- The $'O'$ Notation
  - A function $f(n) = O\big(g(n)\big)$ if there exists n_0 and c such that $f(n) < cg(n)$
  - Whenever, $n > n_0$
    - O (pronounced big-oh) is the formal method of expressing *upper bound* of an algorithm's running time.
    - Measures the *longest amount of time it could possibly take*.
    - $g(n)$ is an *asymptotic upper bound* for $f(n)$.

The Big 'O'

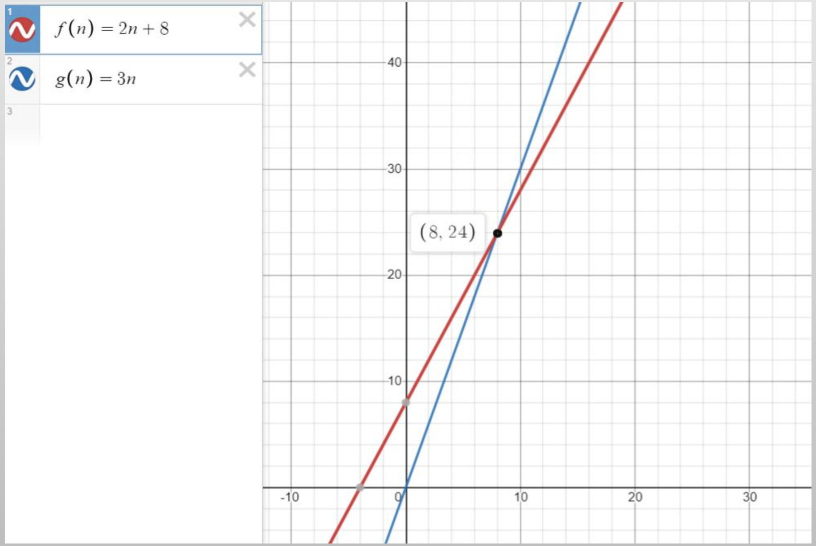
- Example of $'O'$ notation:
  - Suppose, $f(n) = 2n + 8$ and $g(n) = n^2$
  - Can we find a constant n_0, so that $2n + 8 \leq n^2$?
  - $n_0 = 4$ works here!
  - For any number n greater than 4, this will still work. Since we are trying to generalize this for large values of n
    - $f(n)$ is bounded by g(n) and will always be less. (here $c = 1$ is good enough.)
    - Conclusion, $f(n) = O\big(g(n)\big),\ for\ all\ n > 4$
    - Thus here, $f(n) = O(n^2)$

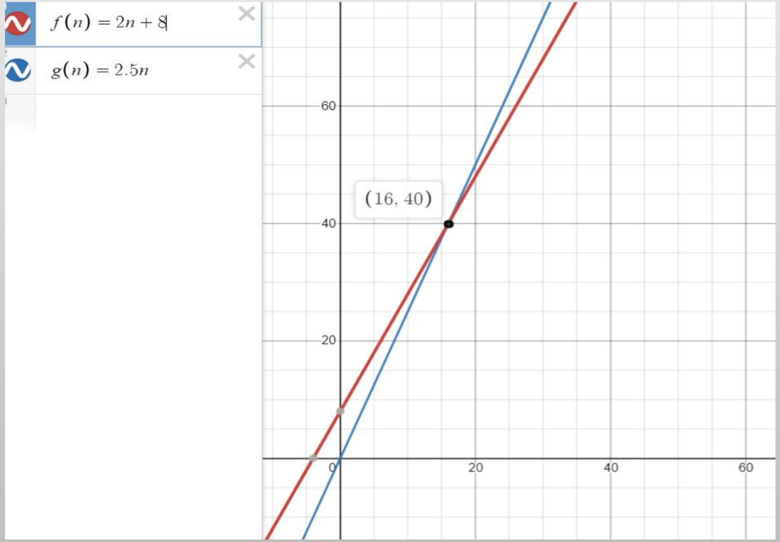
The Big ‘O’



The Big 'O'

- Can we bound $f(n) = 2n + 8$ using $g(n) = n$? (meaning, can $f(n) = O(n)$ be true?)
  - Yes! Pick the value of $'c'$ carefully!
  - $if\ c = 3,\ f(n) = O(n)\ for\ all\ n \geq 8$
  - We can also define, $if\ c = 2.5,\ f(n) = O(n)\ for\ all\ n \geq 16$



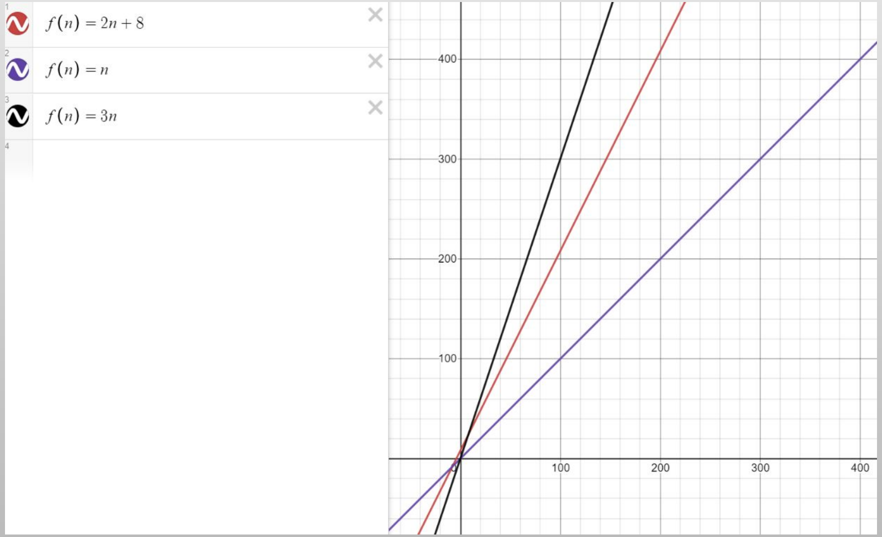


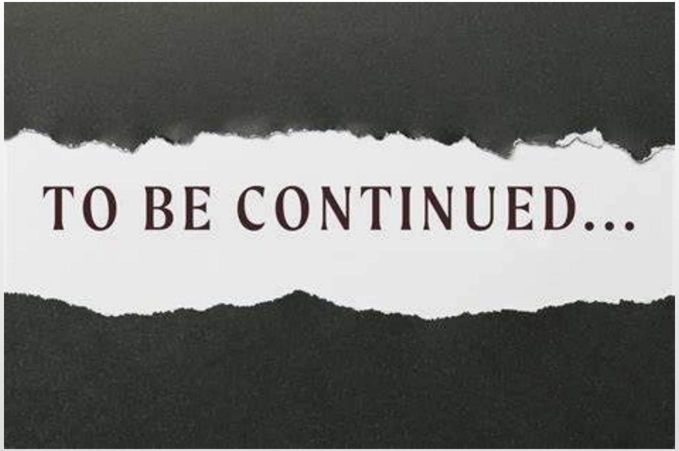
The Big 'Ω'

- Big-Omega Notation:
  - A function $f(n) = \Omega(g(n))$ if there exists n_0 and c such that $f(n) > cg(n)$
  - Whenever $n > n_0$:
    - Almost same definition as Big-Omega, except that $'f(n) > cg(n)'$
    - This makes $g(n)$ a *__lower bound__* function, instead of a upper bound function.
    - $g(n)$ is an *__asymptotic lower bound__* for $f(n)$
    - Describes the *__best that can happen__* for a given data size.

The Big 'θ'

- <u>Big-Theta Notation</u>:
  - A function $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$
  - $f(n)$ is *<u>bounded both from the top and bottom</u>* ***<u>by the same function</u>*** $\boldsymbol{g(n)}$.
  - Thus, $g(n)$ is an *<u>asymptotic tight bound</u>* for $f(n)$
  - Tight bounds are obtained from asymptotic upper and lower bounds.
  - $3n + 3$ is:
    - $O(n)$ (let's say for $c = 4$)
    - $\Omega(n)$ (let's say for $c = 1$)
    - So it can be written as $\Theta(n)$
  - $3n + 3$ is
    - $O(n^2)$ $(for\ all\ n \geq 4)$
    - $\cancel{\Omega(n^2)}$ (only true for $n = 1,2,3$)
    - So it **can not** be written as $\cancel{\Theta(n^2)}$





Acknowledgement

Rafsanjany Kushol
PhD Student, Dept. of Computing Science,
University of Alberta

Sabbir Ahmed
Assistant Professor
Department of CSE, IUT