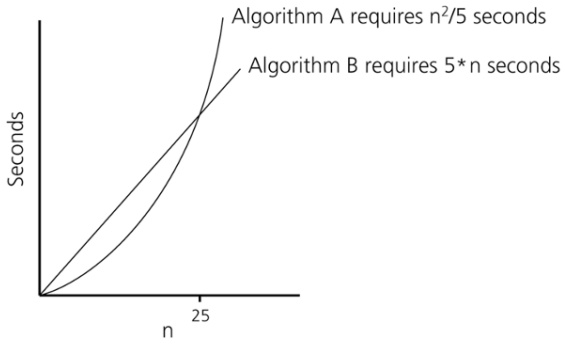
CSCI 3110 Algorithm Analysis

- **Motivation**
  - Given two solutions (algorithms) for a given problem, which one is better? Some criteria we use to evaluate algorithms are:
    - Correctness – does the algorithm do what it is supposed to do?
    - Generality – does it solve only a specific case or will it work for the general case too?
    - Robustness – will the algorithm recover following an error
    - Portability
    - ….
    - **Efficiency** – Does it accomplish its goals with a minimal usage of computer resources (time and space)?
      - **Memory efficiency**: which algorithm requires less memory space during run time?
      - **Time efficiency**: which algorithm requires less computational time to complete?

- **When comparing the algorithms in terms of efficiency, we consider the amount of memory and time required as a function of the size of the data.**

  - **What is the size of the data?**
    - Sorting a set of numbers → the number of items to be sorted
    - Search for an item in a set of items → the number of items to search from
    - Computation involved with a n x n matrix → size of the matrix
    - A list of strings (numStrings * string Length)
    - Use n (N) to represent the size of data



- **Time Complexity:**
  - **Best time**
    The minimum amount of time required by the algorithm for any data of size n.
    → Seldom of interest.

  - **Worst time → our focus**
    The maximum amount of time required by the algorithm for any data of size n

- **Average time → will discuss**
  - The average amount of time required by the algorithm over all data of size n
  - Average complexity analysis is a lot harder to determine than worst and best case complexity. It requires making an assumption regarding the distribution of data to be processed. In many cases, the average time complexity is equal to ½ of the worst case time complexity. But this is not always true.

- Similar definitions can be given for space complexity analysis (**Best case, worst case, average case**)

- **Empirical vs. theoretical analysis:**
  - Empirical approach: Wall Clock Time: problem → difference in computer platforms, loads, compiler specific, language specific issues
  - Machine instructions
  - Theoretical approach: Count number of C++ statements executed during run time or obtain an acceptable approximation. → approach typically used
    - The focus of this type of analysis is performing frequency counts: a count of how many times a statement is executed in an algorithm.

  - **How to count the number of statements executed during run time?** (worst case)

Example1:

	# statements
for (int i=1; i<=N; i++)	**N+1**
x = x+1;	**N**
total:	**2N+1**

Example 2: **worst case**

Smallest = array[0];	**1**
for (int i=1; i<N; i++)	**N // why not N+1 here?**
if (smallest > array[i])	**N-1**
smallest = array[i];	**N-1**
total:	**3N-1**

Example 3:

for (int i=1; i<=N; i++)	**N+1**
for (int j=1; j<=N; j++)	**N*(N+1)= N^{2+}N**
x=x+1;	**N*N=N^2**
total:	**$2N^2$ +2 N+1**

Arithmetic sequence is a sequence of numbers such that the difference from any succeeding term to its preceding term remains constant throughout the sequence.
For example: 1, 2, 3, 4, …100 or 3, 6, 9, 12, …99
To compute the sum of the arithmetic sequence of numbers, use :

Arithmetic Series Formula

$$S_n = n\left(\frac{a_1 + a_n}{2}\right)$$

n = the number of terms being added

a_1 = the first term

a_n = the nth term (last term)

Example 4: **worst case**

for (int i=1; i<=N; i++)	**N+1**
for (int j=1; j<=i; j++)	**(2+3 … + N+(N+1)) = (N)*(2+(N+1))/2**
	= $N^2/2 + 3N/2$
x = x+1;	**(1+2+3 … + N) = N*(N+1)/2**
	= $N^2/2 + N/2$
total:	**$N^2 + 3N+1$**

Example 5: (Assume values in *matrix* are all positive)

	Worst case	**best case**
int largest = 0;	**1**	**1**
for (int=0; i<N; i++)	**N+1**	**N+1**
for (int j=0; j<N; j++)	**(N+1)*N**	**(N+1)*N**
if (largest < matrix [i][j])	**N*N**	**N*N**
largest = matrix[i][j];	**N*N**	**1**
cout << largest << endl;	**1**	**1**
total:	**$3N^2+2N+3$**	**$2N^2+2N+4$**

Discussion :

- When do the best case and worst-case situations occur in example 2 and example 5?
- If best case analysis result is the same as the worst-case analysis result → average case analysis should have the same result.

- Time complexity of the code/algorithm as a function of the size of data N
  F(N) = **$3N^2+2N+3$**
  This function is referred to as the growth rate function.

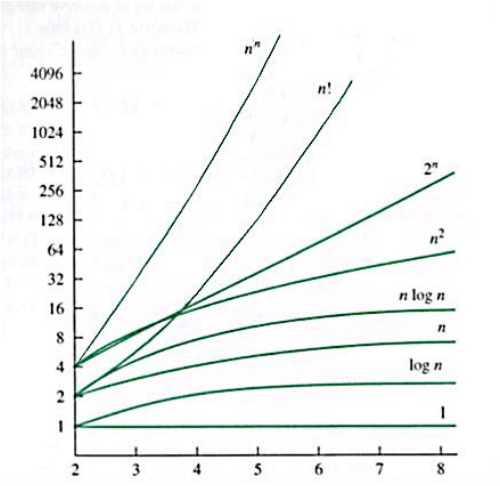
Question:

Given growth rate function of 3 algorithms for a problem, which one is more efficient? $f1(N) = 3N^2-N+5$ $f2(N) = N^2+6N$ $f3(N) = N+\log N$

- The growth rate functions can help us estimate the time needed to solve problems of various sizes. Suppose we have a computer which can do 1,000,000 operations per second, this table shows the amount of time required to solve each application case.

Growth rate Function F(N)	**N=20**	**N=50**	**N=100**	**N=500**	**N=1000**
1000N	0.02 seconds	0.05 s	0.1 s	0.5 s	1 s
500NlogN	0.045 s	0.15 s	0.3 s	2.25 s	5 s
$50N^2$	0.02 s	0.125 s	0.5 s	12.5 s	60 s
$10\ N^3$	0.02 s	1 s	10 s	21 minutes	2.7 hour
N^{logN}	.4 s	1.1 hour	220 day	$5*10^8$ centuries	
2^N	1	35 year	$3*10^4$ centuries		
3^N	58 minutes	$2*10^9$ centuries			

Notice the order of magnitude differences observed among different functions.
When comparing the efficiency of different algorithms, we focus on the significant differences, order of growth, among different algorithms.



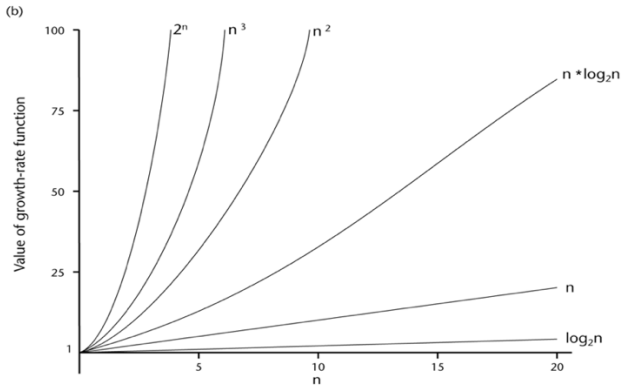
n

Function	10	100	1,000	10,000	100,000	1,000,000
1	1	1	1	1	1	1
$\log_2 n$	3	6	9	13	16	19
n	10	10^2	10^3	10^4	10^5	10^6
$n * \log_2 n$	30	664	9,965	10^5	10^6	10^7
n^2	10^2	10^4	10^6	10^8	10^{10}	10^{12}
n^3	10^3	10^6	10^9	10^{12}	10^{15}	10^{18}
2^n	10^3	10^{30}	10^{301}	$10^{3,010}$	$10^{30,103}$	$10^{301,030}$

- **The efficiency of an algorithm is determined in terms of the order of growth (Big O) of the function.**
  - "How fast does the algorithm grow as the size of data N grows?" ➔ compare the rate of growth (order of magnitude) of the functions and compare the **highest orders**.
  - Normally, the growth function will approach some simpler function asymptotically.
  - For growth functions having different orders of magnitude, the exact frequency counts and the constants become insignificant.
  - The algorithms are grouped into groups according to the order of growth (Big O) of their growth rate functions:

    $O(1)<O(\log N)<O(N)<O(N\log N)<O(N^2)<O(N^3)<O(2^N)< O(3^N)<O(N!)<O(N^N)$

    *(important to know the **order** of these Big O functions)*



Arranged in order of their growth rate function (GRF)➔ the order of time efficiency
Algorithms that have growth rate function of the same Big O category are considered of the same efficiency.

- How to determine which Big O function does an algorithm's growth rate function falls into? ➔Determined by using **the big O notation**
  - Rules of Big O notations that help to simplify the analysis of an algorithm
    - **Ignore the low order terms in an algorithm's growth rate function**
    - **Ignore the multiplicative constants in the high order terms of GRF**
    - **When combining GRF, O(f(N))+O(g(N)) = O(f(N)+g(N))**

    (e.g., when two segments of codes of a program are first analyzed separately, and we want to know the total time complexity of the two code segments)

- **Practice Questions:**
  Try the rules on these growth functions to derive the corresponding big O functions:
  1. $f1(n) = 8n^2 – 9n +3$
  2. $f2(n) = 7 \log_2 N + 4n$
  3. $f3(n) = 2+4+6 \ldots + 2n$
  4. $f4(n) = f2(n) + f3(n)$
  5. $f5(n) = 2n^4 - 3n^2 + 4$
  6. $f6(n) = 5N\log N + 5n^2 + 100$
  7. $f7(n) = 30n^2 + 2^N + 1000N - 40\log N$

- **Formal Definition of Big O function:**

  f(N) = O(g(N)) if there is a constant value C, such that |f(N)| <= C*|g(N)| for all N>= N_0, where N_0 is non-negative integer

Example 1: Given f(n) = 5*n+10, show that f(n)=O(n)

|f(n)| <=C*|n|
C*(- n) <= 5*n+10 <= C*n

(1) 5*n+10 <=C*n
if C=6, then
5*n + 10 <= 6*n
n-10 >= 0
n>=10
so, for n_0=10, n>=n_0, 5*n+10 <=C*n holds

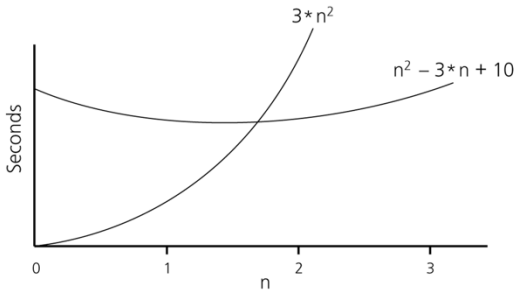
(2) for C=6, n_0=10,
-C*n <= 5*n+10 also holds
This means, for C=6, n_0=10,
|f(n)| <=C*|n|
holds for all n>=n_0.

Therefore, f(n)=O(n)

Example 2:

What is the big O function for $f(n)=n^2-3n+10$?

$|f(n)| <= |3\ n^2|$, for n>=2. In this case, C=3, N_0=2
therefore, $f(n) = O(n^2)$



The following illustrates the steps involved in showing that f(n) = O(n2):
We need to show that |f(n)| <= C*|g(n)|
is satisfied with certain constant C value and n0 value:

$|n^2-3n+10| <= C*n^2$
$C*(-n^2) <= n^2-3n+10 <= C*n^2$
(1) $n^2-3n+10 <= C*n^2$
if C=3, then
$n^2-3n+10 <= 3*n^2$
$2n^2 + 3n -10 >=0$
if n_0= 2, then the above inequality holds.
This means, when C=2, n_0=2, $n^2-3n+10 <= C*n^2$ for all n>= n_0

(2) In addition, when C=2, n_0=2,
$-C*n^2 <= n^2-3n+10$, for all n >= n_0
Therefore, we derived that, for C=2, n_0=2,
$|n^2-3n+10| <= C*n^2$
holds for all n>n_0.

This shows that $f(n) = O(n^2)$

Questions:

1. **(based on the definition of Big O function), If a growth rate function f(n) is O(n), is this function also $O(n^2)$?**

2. **Which Big O function should be used to represent the actual order of growth of a GRF f(n)?**
   Answer: For any f(n), there can be many functions g(n) that satisfy the requirement in Big O notation. In algorithm analysis, we want to find **the g(n) that is the tightest bound around f(n).**

- **Practice problems:**

Find the time complexity of the following algorithm in terms of its growth rate function, show the big O function.

(a) Power of 2 computation

```
sum = 1;                                1
powerTwo = 2;                           1
while (powerTwo < N)                    K+1   (K=number of times
                                                body of loop executes
{
        sum += powerTwo;                K
        powerTwo = 2*powerTwo;          K
}
print sum;                              1

                                  total:        3K+4
```

since : $K=\log_2 N$ ➔ $F(n) = 3 \log_2 N + 4$

$f(n) = O(\log_2 N)$

(b) Analyze the following algorithm (linear search). What is the growth function for this algorithm? What is the big O of this algorithm?

```
int linearSearch(int arr[], int size, int value)
{
        int index = 0;                  // Used as a subscript to search the array
        int position = -1;              // To record the position of search value
        bool found = false;             // Flag to indicate if value was found

        while (index < size && !found)
        {
                if (arr[index] == value)  // If the value is found
                {
                        found = true;           // Set the flag
                        position = index;       // Record the value's subscript
                }
                index++;                // Go to the next element
        }
        return position;            // Return the position, or -1
}
```

(c) Analyze the following algorithm (binary search). What is the growth function for this algorithm? What is the big O of this algorithm?

```
void  BinarySearch(int    item, bool & found, int & position, int data[], int length)
{
        int first = 0;
        int last = length;
        int  middle;

        found = false;
        while  (last >= first && !found)
        {
                middle = (first+last)/2;
                if (item > data[middle])
                        first = middle+1;
                else if (item < data[middle])
                        last = middle -1;
                else
                        found = true;
        }
        if (found)
                position = middle;
}
```

(d) Derive the growth rate function for the following algorithm, show the corresponding big O function.

```
for (int j=1; j<N; j++)                 N
{
        key=a[j];                       N-1
        i=j-1;                          N-1
        while (i>=0 && a[i] > key)      2+3+ … N=(N-1)*(2+N)/2
        {
                a[i+1] = a[i];          1+2+ …+(N-1) = (N-1)*(1+N-1)/2
                i--;                    1+2+ …+ (N-1) = (N-1)*(1+N-1)/2
        }
        a[i+1] = key;                   N-1
}
min = a[0];                             1
max = a[n-1];                           1

                              total:    3/2N^2 + 7/2N - 2
```