CSCI 2170
Algorithm Analysis

Given two solutions (algorithms) for a given problem, which one is better?
In computer science, we measure the quality of algorithms in terms of its memory and time efficiency.

- **Memory efficiency**: which algorithm requires less memory space during run time?
- **Time efficiency**: which algorithm requires less computational time to complete?
    - Wall Clock Time : problem → difference in computer platforms (MIP)
    - Machine instructions:
    - Number of C++ statements executed during run time

How to count the number of C++ statements executed during run time?
Example1:

```
                                        # statements
int sum=0;
for (int i=0; i<N; i++)
    sum += array[i];
cout << sum;

                                    _____________
                                    total:
```

Example 2:

```
                                              Worst case          best case
int largest = 0;
for (int=0; i<N; i++)
     for (int j=0; j<N; j++)
          if (largest < matrix [i][j])
               largest  = matrix[i][j];
cout << largest << endl;

                                    _____________       _____________
                                    total:
```

The growth rate function: $f(N) = 3N^2+2N+2$, indicates the number of statement executed as a function of the size of the data, N.

Best case analysis, Average case analysis, Worst case analysis

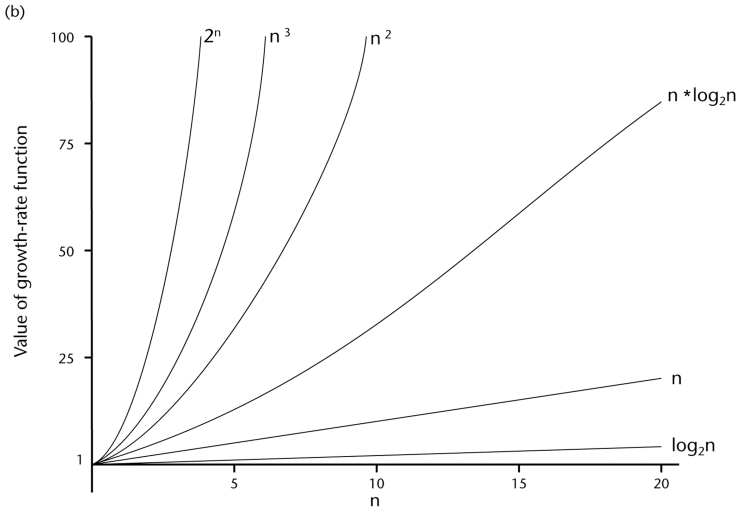
When does best case and worst case situations occur in example 2?
If best case analysis result is the same as the worst case analysis result → average case analysis should have the same result.

Given time complexity of two algorithms, for example
$f1(N) = 3N^2-N+5$ $f2(N) = N^2+6N$ $f3(N) = N+\log N$
Which one is more efficient?

The efficiency of an algorithm is determined in terms of the order of growth of the function. The algorithms are grouped into groups according to the order of growth of their time complexity functions, which include:

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

look at the graph with all these functions plotted against each other>



Arranged in order of their growth rate function→ the order of time efficiency
Algorithms that have growth rate function of the same Big O category are considered the same efficient.

If we know the time complexity of our algorithm, it can determine the feasibility of solving problems of various sizes. Suppose we have a computer that can do 1,000,000 operations per second for simplicity

Growth rate F(N)	**N=20**	**N=50**	**N=100**	**N=500**	**N=1000**
1000N	0.02 s	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 m	2.7 h
$N^{\log N}$	.4 s	1.1 hr	220 d	$5*10^8$ centuries	
2^N	1m	35 y	$3*10^4$ centuries		
3^N	58 m	$2*10^9$ centuries			

For any f(n), there can be many 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).

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 are analyzed separately first, and we want to know the time complexity of the total of the two segments

(Try the rules on example $f(n) = 2n^4-3n^2 +4$)

Practice:

1. $f1(n) = 8n^2 – 9n +3$
2. $f2(n) = 7 \log_2 n + 4n$
3. $f3(n) = 1+2+3 \ldots + n$
4. $f5(n) = f2(n) + f3(n)$

Practice : find the big O function of the following algorithm

```
sum = 1;
powerTwo = 2;
while (powerTwo < N)
{
        sum += powerTwo;
        powerTwo = 2*powerTwo;
}
print sum;
```

total: _____

Exercise 1: What is the big O function for linear search?

```
int  LinearSearch(int data[], int size, int itemToSearch)
{
      for (int i=0; i<size; i++)
         if (data[i] == itemToSearch)
               return i;

      return -1;
}
```

Exercise 2: What is the big O function for this algorithm?

```
// binary search is an efficient search algorithm for SORTED array
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;
}
```

Exercise 3: What is the big O function for bubble sort?

```
void BubbleSort(dataType A[], int N)
{
  bool Sorted = false;  // false when swaps occur

  for (int Pass = 1; (Pass < N) && !Sorted; ++Pass)
  {

    Sorted = true;  // assume sorted
    for (int Index = 0; Index < N-Pass; ++Index)
    {
      int NextIndex = Index + 1;
      if (A[Index] > A[NextIndex])
      {
        Swap(A[Index], A[NextIndex]);
        Sorted = false;  // signal exchange
      }
    }

  }  // end for
}  // end BubbleSort

void Swap(dataType& X, dataType& Y)
{
  dataType Temp = X;
  X = Y;
  Y = Temp;
}  // end Swap
```

Exercise 4:What is the big O function for quick sort?

```
void Quicksort(dataType A[], int F, int L)
// ----------------------------------------------------------
// Sorts the items in an array into ascending order.
// Precondition: A[F..L] is an array.
// Postcondition: A[F..L] is sorted.
// ----------------------------------------------------------
{
  int PivotIndex;

  if (F < L)
  {  // create the partition: S1, Pivot, S2
    Partition(A, F, L, PivotIndex);

    // sort regions S1 and S2
    Quicksort(A, F, PivotIndex-1);
    Quicksort(A, PivotIndex+1, L);
  }  // end if
}  // end Quicksort

void ChoosePivot(dataType A[], int F, int L);
// ----------------------------------------------------------
// Chooses a pivot for quicksortÕs partition algorithm and swaps it with the first item in an array.
// Precondition: A[F..L] is an array; F <= L.
// Postcondition: A[F] is the pivot.
// ----------------------------------------------------------
void Partition(dataType A[], int F, int L, int& PivotIndex)
{
  ChoosePivot(A, F, L);      // place pivot in A[F]
  dataType Pivot = A[F];     // copy pivot

  // initially, everything but pivot is in unknown
  int LastS1 = F;            // index of last item in S1
  int FirstUnknown = F + 1;  // index of first item in unknown

  // move one item at a time until unknown region is empty
  for (; FirstUnknown <= L; ++FirstUnknown)
  {  // move item from unknown to proper region
    if (A[FirstUnknown] < Pivot)
    {  // item from unknown belongs in S1
      ++LastS1;
      Swap(A[FirstUnknown], A[LastS1]);
    }  // end if

    // else item from unknown belongs in S2
  }  // end for

  // place pivot in proper position and mark its location
  Swap(A[F], A[LastS1]);
  PivotIndex = LastS1;
}  // end Partition
```