CSCI 2170 **Lecture Notes (9)**
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

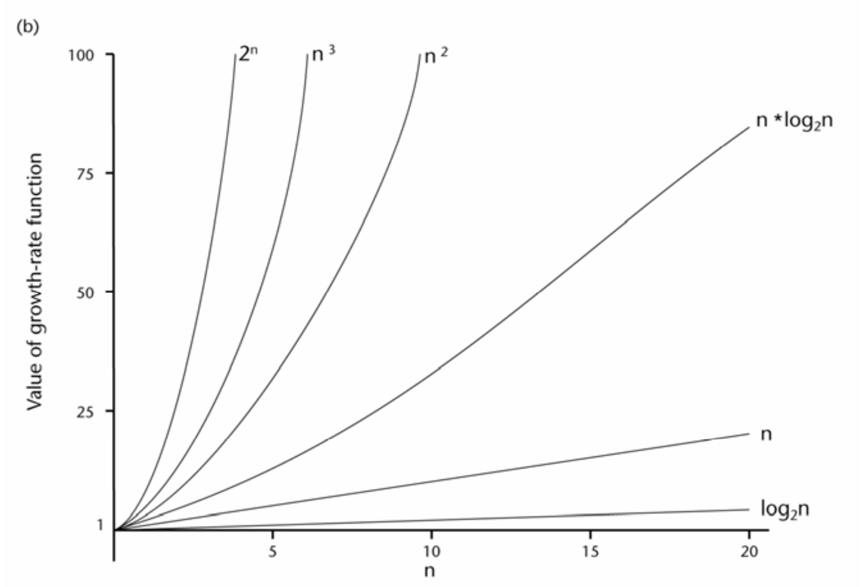
$f1(N) = 3N^{2-}N+5$ $f2(N) = N^2+6N$ $f3(N) = N+\log N$

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:

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

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.

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. $f3(n) = 1+2+3 \dots + n$
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: Try to analyze the following algorithm (binary search). 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;
}
```

A Formal Definition

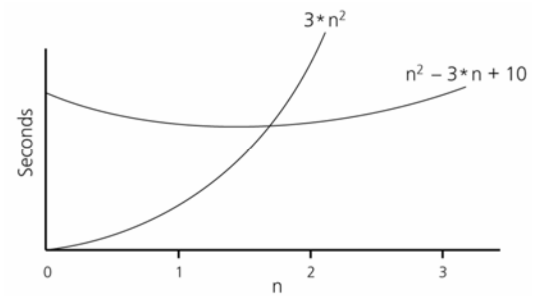
How to determine which Big O function does an algorithm's growth rate function falls into?
This is determined by using the big O notation:

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

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


3*n²
n² − 3*n + 10
Seconds
n