TIMING

We need to estimate how long a program will run. Every time we run the program we are going to have different input values and the running time will vary. Since the running time will vary, we need to calculate the worst case running time. The worst case running time represents the maximum running time possible for all input values. We call the worst case timing **"big Oh"** written **O(n)**. The **n** represents the worst case execution time units. How do we calculate "**Big Oh**" ???

We first must know how many time units each kind of programming statement will take:

1. simple programming statement: O(1)

```
k++;
```

Simple programming statements are considered 1 time unit

2. linear for loops: O(n)

```
k=0;
for(i=0; i<n; i++)
        k++
```

For loops are considered **n** time units because they will repeat a programming statement **n** times. The term linear means the **for** loop increments or decrements by 1

3. non linear loops: O(log n)

```
k=0;
for(i=n; i>0; i=i/2)
        k++;
```

```
k=0;
for(i=0; i<n; i=i*2)
        k++;
```

For every iteration of the loop counter **i** will divide by 2. If **i** starts is at 16 then then successive **i's** would be 16, 8, 4, 2, 1. The final value of **k** would be 4. Non linear loops are **logarithmic**. The timing here is definitely $\log_2 n$ because $2^4 = 16$. Can also works for multiplication.

4. nested for loops $O(n^2)$: $O(n) * O(n) = O(n^2)$

```
k=0;
for(i=0; i<n; i++)
        for(j=0; j<n; j++)
                k++
```

Nested for loops are considered n^2 time units because they represent a loop executing inside another loop. The outer loop will execute **n** times. The inner loop will execute **n** times for each iteration of the outer loop. The number of programming statements executed will be **n * n.**

5. sequential for loops: O(n)

```
k=0;
for(i=0; i<n; i++)
    k++;

k=0;
for(j=0; j<n; j++)
    k++;
```

Sequential for loops are not related and loop independently of each other. The first loop will execute n times. The second loop will execute n times after the first loop finished executing. The worst case timing will be:

O(n) + O(n) = 2 * O(n) = O(n)

We drop the constant because constants represent 1 time unit. The worst case timing is **O(n)**.

6. loops with non-linear inner loop: O(n log n)

```
k=0;
for(i=0;i<n;i++)
    for(j=i; j>0; j=j/2)
        k++;
```

The outer loop is **O(n)** since it increments linear. The inner loop is **O(log n)** and is non-linear because decrements by dividing by 2. The final worst case timing is:

O(n) * O(log n) = O(n log n)

7. inner loop incrementer initialized to outer loop incrementer: $O(n^2)$

```
k=0;
for(i=0;i<n;i++)
    for(j=i;j<n;j++)
        k++;
```

In this situation we calculate the worst case timing using both loops. For every **i** loop and for start of the inner loop **j** will be n-1 , n-2, n-3…

O(1) + O(2) + O(3) + O(4) + ...

which is the binomial series:

$$\frac{n(n-1)}{2} = \frac{n^2 - n}{2} = \frac{n^2}{2} = \mathbf{O(n^2)}$$

i	j
0	n-0
1	n-1
2	n-2
3	n-3
4	n-4

8. power loops: $O(2^n)$

```
k = 0;
for(i=1;  i<=n; i=i*2)
    for(j=1; j<=i; j++)
        k++;
```

To calculate worst case timing we need to combine the results of both loops. For every iteration of the loop counter **i** will multiply by 2. The values for j will be 1, 2, 4, 8, 16 and k will be the sum of these numbers 31 which is 2^n- 1.

9. if-else statements

With an **if else** statement the worst case running time is determined by the branch with the largest running time.

```
/* O(n) */
if (x == 5)
    {
    k=0;
    for(i=0; i<n; i++)
       k++;
    }

/* O(n^2) */
else
   {
   k=0;
   for(i=0;i<n;i++)
       for(j=i; j>0; j=j/2)
           k++;
   }
```

choose branch that has largest delay

The largest branch has worst case timing of **$O(n^2)$**

10. recursive

From our recursive function let T(n) be the running time.

```
int f(int n)
        {
        if(n == 0)
                return 0;
        else
                return f(n-1) + n
        }
```

recursion behaves like a loop.
The base case is the termination for recursion.

For the line: **if(n == 0) return 0;** this is definitely: **T(1)**
For the line: **else return f(n-1) + n** the time would be : **T(n-1) + T(1)**

The total time will be: **T(1) + T(n-1) + T(1) = T(n-1) + 2** which is **O(n)**

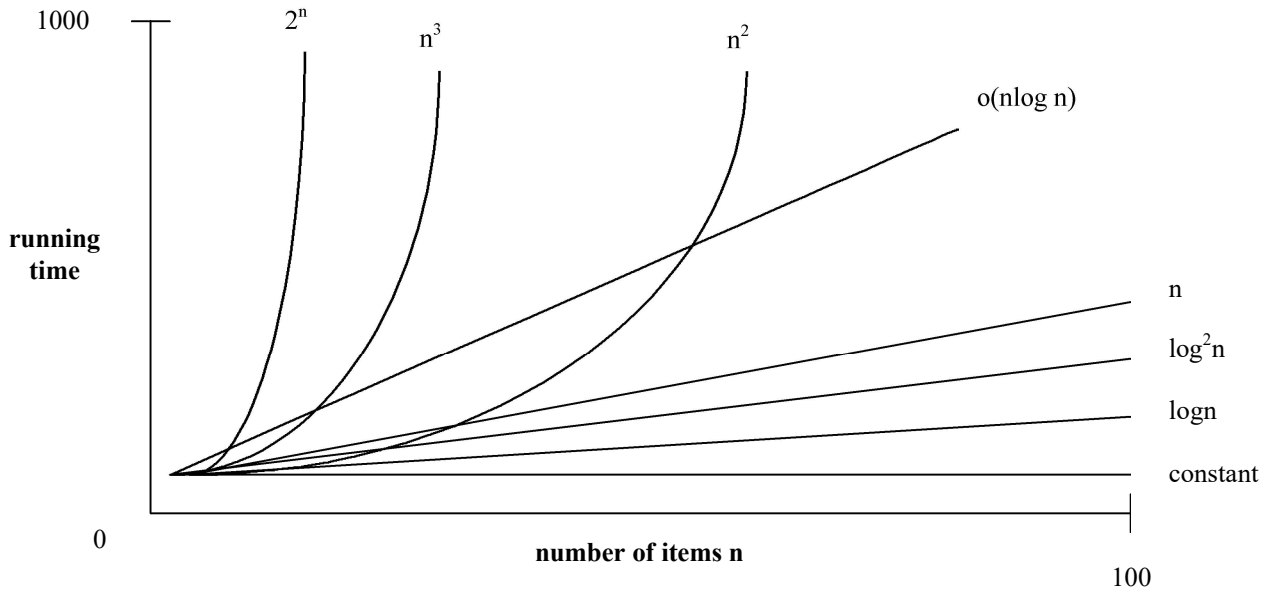
growth rates summary

The following chart lists all the possible growth rates:

c	constant
log n	logarithmic
log_2 n	log squared
n	linear
n log n	linear log squared
n^2	quadratic
n^3	cubic
2^n	exponential

growth rate graph

The growth rates are easily visualized in the following chart. The horizontal axis **x** represent **n** and the vertical **y** axis represent running time:



As n increases the running depends on the growth rates. For any n constant is the fastest and 2^n is the slowest. Worst case timing is also dependent on n For small values of n we can see n^2 is mush faster than n log n. For larger values of n (n log n) is much faster than n^2. You must be careful in choosing your algorithms for values of n.

$$n^2 \sim n \log n$$