Algorithmic Efficiency

We can measure an algorithm based on how many computer instructions it takes to solve a problem of a given size as a function n of the size of the input data.
When we measure this way, we get two benefits:

1. We can compare two algorithms for a given sized input.
2. We can predict the performance of those algorithms when they are applied to less or more data.

This is the idea between the **Big-O** concept used in computer science.

The Big-O approach measures an algorithm by the gross number of steps that it requires to process an input of **size N** in the **WORST CASE SCENARIO**.
Example:
Algorithm X requries 5N^2 + 3N + 20 steps to process N items.
With Big-O, we ignore the **coefficients** and **lower-order** terms of the expression, so the Big-O of algorithm X is N^2.

A function f(n) is O(g(N)) if there exists N0 and k such that for all N >= N0, f(N) <= k * g(N)

To compute the Big-O of a function, first we need to determine the number of operations an algorithm performs.
Operations:

- Accessing an item (e.g. in an array)
- Evaluating a mathematical expression
- Traversing a single link in a linked list, etc.

```
int arr[i][n];

for(int i = 0; i < n; i++) <=initializes i once, performs n comparisons between i and n, increme

nts the variable i n times

{

    for(int j = 0; j < n; j++) <=initializes j n different times, performs n^2 comparisons betwe

en j and n, increments the variable j n^2 times

    {

        arr[i][j] = 0; <=sets arr[i][j]'s value n^2 times

    }

}
```

f(n) = 1 + n + n + n + n^2 + n^2 + n^2 = 3n^2 + 3n + 1
==> f(n) becomes O(n^2)
If I say, "this algorithm is O(n^2), to process n items", this algorithm requires roughly n^2 operations.

Big-O: The Complete Approach

- Determine how many steps f(n) an algorithm requries to solve a problem, in terms of the number of items **n**
- Keep the most significant term of that function and throw away the rest
  - f(n) = 3n^2 + 3n + 1 becomes f(n) = 3n^2
  - f(n) = 2nlog(n) + 3n becomes f(n) = 2nlog(n)
- Remove any constant multiplier from the function
  - f(n) = 3n^2 becomes f(n) = n^2
  - f(n) = 2nlog(n) becomes f(n) = nlog(n)
- This gives you your big O
  - O(n^2)
  - O(n log(n))

To simplify, all you need to do is focus on the most frequently occurring operations to save time.

Find the Big-O Challenge

```
for(int i = 0; i < n; i += 2)
    sum++;
```

O(n / 2) => O(n)

```
for(int i = 0; i < q; i++) //the outer loop runs q times
    for(int j = 0; j < q; j++) //every time the outer loop runs once, the inner loop runs q times
        sum++;
```

O(q * q) = O(n^2)

```
for(int i = 0; i < n; i++) //outer loop runs n times
    for(int j = 0; j < n * n; j++) //every time outer loop runs once, inner loop runs n * n times
        sum++;
```

O(n * n^2) = O(n^3)

These examples are important!!!

```
k = n;
while(k > 1)
{
    sum++;
```

```
    k = k / 2;

}
```

k goes from n to n / 2 to n /4 all the way down to 1. It takes log2(n) steps to finish, since we divide by 2 each time.
O(log2(n))

```
for(int i = 0; i < n; i++) //loop runs n times

{

    int k = n;

    while(k > 1) //k goes from n/3 to n/9 all the way down to 1

    {

        sum++;

        k = k / 3;

    } //the while loop takes log3(n) steps to finish

}
```

O(n * log3(n))

Other examples:

```
for(int j = 0; j < n; j++) //runs n times

    for(int k = 0; k < j; k++) //substitute j with n

        sum++; ===> O(n^2)

for(int i = 0; i < q * q; i++)

    for(int j = 0; j < i; j++) //substitute i with q * q

        sum++; ===> O(n^2 * n^2) = O(n^4);

for(int i = 0; i < n; i++)

    for(int j = 0; j < i * i;) //substitute i * i for n * n

        for(int k = 0; k < j; k++) //substitute j for n * n

            sum++; ===> O(n * n^2 * n^2) = O(n^5)

for(int i = 0; i < p; i++)

    for(int j = 0; j < i * i; j++) //substitute i * i for p * p

        for(int k = 0; k < i; k++) //substitute i for p

            sum++; ===> O(p * p^2 * p) = O(p^4) = O(n^4)
```

```
for(int i = 0; i < n; i++) //loop runs n times

{

    Circ arr[n]; //a Circ object is constructed n times when the loop runs once

    arr[i].setRadius(i);

} ===> O(n * n) = O(n^2)
```

```
for(int i = 0; i < n; i++) //outer loop runs n times

{

    int k = i; //replace i with n

    while(k > 1)

    {

        sum++;

        k = k / 2;

    } //take log2(n) steps to finish

}
```

O(n * log2(n))

Big-O for Multi-input Algorithms
Often, an algorithm will operate on two (or more) independent data sets, each of a difference size.
In these cases, when we compute the algorithm's Big-O, we must take into account both independent sizes.

```
//the number of people, p, and the number of foods, f, are completely independent

void buffet(string people[], int p, string foods[], int f)

{

    int i, j;

    for(int i = 0; i < p; i++) //outer loop runs p times

        for(int j = 0; j < f; j++) //inner loop runs f times

            cout << people[i] << " ate " << foods[j] << endl;

} ===> O(p * f)
```

```
void tinder(string csmajors[], int c, string eemajors[], int e)

{

    for(int i = 0; i < c; i++) //outer loop runs c times

        for(int j = 0; j < c; j++) //inner loop runs c times

            cout << csmajors[i] << " dates " << csmajors[j] << endl; ===> O(c^2)

    for(int k = 0; k < e; k++) //loop runs e times

        cout << eemajors[k] << " sits at home."; ===> O(e)

} ===> O(c^2 + e)
```

O(c^2 + e)
Not! just O(c^2) because we must include both independent variables in our Big-O

We must include both variables in the Big-O even if one is higher-order than the other because either variable could dominate the other! Don't forget - you must still eliminate lower-order terms for each independent variable.

```
void barf(int n, int q)

{

    for(int i = 0; i < n; i++) //outer loop runs n times

    {

        if(i == n / 2) //the 1 time i is equal to n/2, we run this inner loop q times

        {

            for(int k = 0; k < q; k++)

                cout << "Muahahaha";

        }

        else

            cout << "Burp!";

    }

}
```

O(n + q)
Accessing an element of an array can be done in constant time, so it wouldn't matter how big the array is.

If it's true for O(N - 1), it's true for O(N) as well.

Common Big O functions

- O(1) - constant time

- O(log n) - logarithmic time
- O(n log n) - polylogarithmic time
- O(n) - linear time
- O(n^2) - quadratic time
- O(n^c) - polynomial time
- O(c^n) - exponential time

Exponential time is very inefficient for big algorithms.

```
for(int i = 0; i < N; i++) <======O(N)

    c[i] = a[i] + b[i]; <======O(1)

//overall: O(N)
```

```
for(int i = 0; i < N; i++) <=======O(N^2)

{ <==========================O(N)

    a[i] *= 2;  <==========O(1)

    for(int j = 0; j < N; j++) <=========O(N)

    {

        d[i][j] = a[i] * c[j]; <=========O(1)

    }

}

//overall: O(N^2)
```

Just because there's a nested loop does not mean it efficiency is going to be O(N^2)

```
for(int i = 0; i < N; i++) <=========O(N)

{ <==================================O(1)

    a[i] *= 2; <=====================O(1)

    for(int j = 0; j < 100; j++) <=========O(1)

        d[i][j] = a[i] * c[j]; <============O(1)

}

//overall: O(N)
```

```
for(int i = 0; i < N; i++) <==========O(N^2)

{ <==================================O(i)

    a[i] *= 2; <=====================O(1)

    for(int j = 0; j < i; j++) <==========O(i) = O(N)

        d[i][j] = a[i] * c[j]; <=============O(1)

}

//overall: O(N^2)
```

```
for(int i = 0; i < N; i++) <========O(N^2)

{

    if(find(a, a + N, 10 * i) != a + N) <=========O(N^2 log N)

        count++; <==========================O(1)

}
```

```
for(int i = 0; i < N; i++) <=============O(N^2 log N)

{ <=========================O(N log N)

    a[i] *= 2; <=============O(1)

    for(int j = 0; j < N; j++) <=========O(N log N)

        d[i][j] = f(a, N); <==========O(log N) //f(a, N) is O(log N)

}
```

STL and Big-O
STL (stacks, queues, sets, vectors, lists, and maps) use algorithms to get things done...and these algorithms have Big-Os too!

For example, if we want to search for a word in a set that contains n words, it requires O(log2(n)) steps!

```
void inDict(set<string> &d, string w)

{

    if(d.find(w) == d.end())

        cout << w << " isn't in dictionary!";

```

```
}
```

If we want to add a value to the end of a vector holding n items, it takes just one step, so it's O(1)!

```
void otherFunc(vector<int> &vec)

{

    vec.push_back(42);

}
```

And if we want to delete the 1st value from a vector containing n items, it takes n steps, making it O(n).

```
void otherFunc1(vector<int> &vec)

{

    vec.erase(vec.begin());

}
```

```
void spellCheck(set<string> &dict, string doc[], int D)

{

    for(int i = 0; i < D; i++) //loop runs D times

        inDict(dict, doc[i]); //searching a set of N items requires log2(N) steps

}
```

O(D * log2(n))

```
void printNums(vector<int> &v)

{

    int q = v.size();

    for(int i = 0; i < q; i++) //runs q times

    {

        int a = v[0]; //O(1)

        cout << a;

        v.erase(v.begin()); //O(q)
```

```
            v.push_back(a); //O(1);
        }
    } ===> q * (1 + q + 1) = O(q^2)
```

O(q^2)

Cheat Sheet

List

- Inserting an item: O(1)
- Deleting an item: O(1)
- Accessing an item (top or bottom): O(1)
- Accessing an item (middle): O(n)
- Finding an item: O(n)

Set

- Inserting a new item: O(log2(n))
- Finding an item: O(log2(n))
- Deleting an item: O(log2(n))

Queue and Stack

- Inserting a new item: O(1)
- Popping an item: O(1)
- Examining the top: O(1)

Vector

- Inserting an item (top or middle): O(n)
- Inserting an item (bottom): O(1)
- Deleting an item (top or middle): O(n)
- Deleting an item (bottom): O(1)
- Accessing an item: O(1)
- Finding an item: O(n)

Map

- Inserting a new item: O(log2(n))
- Finding an item: O(log2(n))
- Deleting an item: O(log2(n))