Big-O Analysis of Algorithms:

Types of data structures:

- ArrayLists
- Lists
- Stacks
- Queues
- Trees
- Maps
- Sets
- Heaps
- Graphs

These data structures are not specific to Java and can be implemented in many programming languages.

The efficiency of each data structure varies based on the problem you are solving, so it is good to know how to compare those efficiencies.

Abstract Data Type: a general abstract description of a structure for sorting data and operating that can be performed on that data.
Ex: *ADT for List: a list is a sequence of values that we can add to, search through, and remove from.*

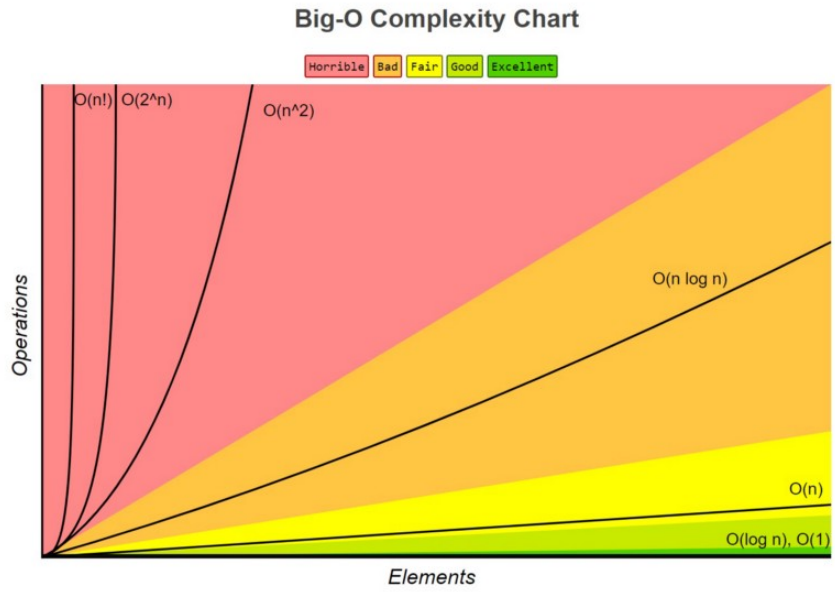
- *Can be implemented in many ways, like Array Lists or LinkedList*

How do we compare efficiencies and speeds?

- The naive way is to consider timing - starting and stopping a stopwatch for the program. The problem is that this will vary based on hardware.
    - An older computer might run a program in 1 second, while a newer one might take 1 millisecond.
- A better way is to think about how many steps a computer must take, and how that step count changes based on the size of the inputs.
    - Ex: 10 steps for 1 input, but 100 steps for 2 inputs.
- Examples of what counts as 1 “step”:
    - Adding numbers
    - Comparing numbers
    - Other simple operations

Big-O notation is based on large values of N (the number of inputs) only. It looks like this O (function of N).

- Constant time algorithm: order 1: O(1)
- Linear time algorithm: order N: O(N)
- Quadratic time algorithm: Order N squared: $O(N^2)$
- We can ignore constants and lower order terms.
    - If for N inputs the program takes $3N^2 + 2N + 1$ steps, we only take the N^2 into consideration and call it an $O(N^2)$ algorithm.



Examples of Calculating Big-O

Statements:

```
Statement 1;
Statement 2;
…
Statement k;
```

If each statement is simple (only basic operations), then no matter the value of N, there will always be k constant-time steps done, leading to O(1).

If-Then Else

```
If (cond):
      Block 1 (runs O(n^4))
else:
      Block 2 (runs O(log(N)))
```

The Big-O time of this program is the slower of the big-O time of Block 1 and big-O time of block 2. Since block 1 is slower in the example, we say the overall program has a big-O of $O(N^4)$.

Loops:

```
for(int i = 0; i < n; i++) {
      Block (that takes constant time)
}
```

This kind of loop has O(N) time because for every increase in the value of N, the for loop needs to increase the number of iterations (because it takes longer for i to get to n). Notice how the step to initialize i is constant time, as is the block, but since O(N) is slower we use that for the overall code.

Sequential Search

```
public int search(int target, int[] a) {
int i = 0;
while(true) {
      if(i == a.length) {
            return -1;
            }
      if(a[i] == target) {
            return i;
            }
      i++;

}
}
```

On average, the search function will find the value in the middle of the area, so it takes N/2 iterations to find. We ignore the ½ coefficient and say the program is O(N).

- You can also think about it like the while-loop goes from 0 to n (where n = a.length), so it's O(N).

Binary Search

Since binary search cuts out half of each array, it takes $O(\log_2(N))$. However, we can express $\log_2(N)$ as log(N) / log(2), and we take out the 1/log(2) constant to get O(log(N)).

Multiple Loops

```
int total = 0;
int[] arr = new int[N];
for(int a: arr) {
      for(int b = 1; b < N/2; b *= 3) {
            total = a+b;
      }
}
```

This is O(N*log(N)). The first loop is O(N), and the second loop is O(log(N)) (because the value of b goes up exponentially fast to reach N/2) and since they are nested, they multiply!

Example Program 1:

```
public void mystery(int size) {
      int x = 5;
      for (int i = 0; i < 15+size; i++)
      {
            for (int i = size; i < size; i++) {
                  for (int i = 1; i < size; i--)
                  {
                        x++;
```

```
                    }
                }
            }
}
```

This is O(N). Creating and assigning x is constant, the first for-loop is O(n), and the second loop never runs, which prevents the third loop from running. Therefore, it is O(N).

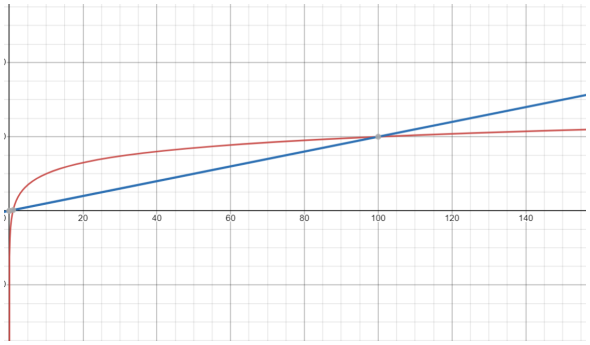
Example Program 2:

```
public boolean mystery(int size) {
int a = 1;
int b = a+2;
int c = size + a;
for(int d = 0; d < c; d++) { //O(N)
      b += d;
}
for(int i = 0; i < size+17; i++) { //O(N)
      for(int j = 0; j < 10; j++) { //O(1)
            for(int k = size-5; k < size+2; k += 3) { //O(1)
                  b -= k;
}
      }
}
return b > -50 ? true: false;
}
```

O(N).

Large Values of N

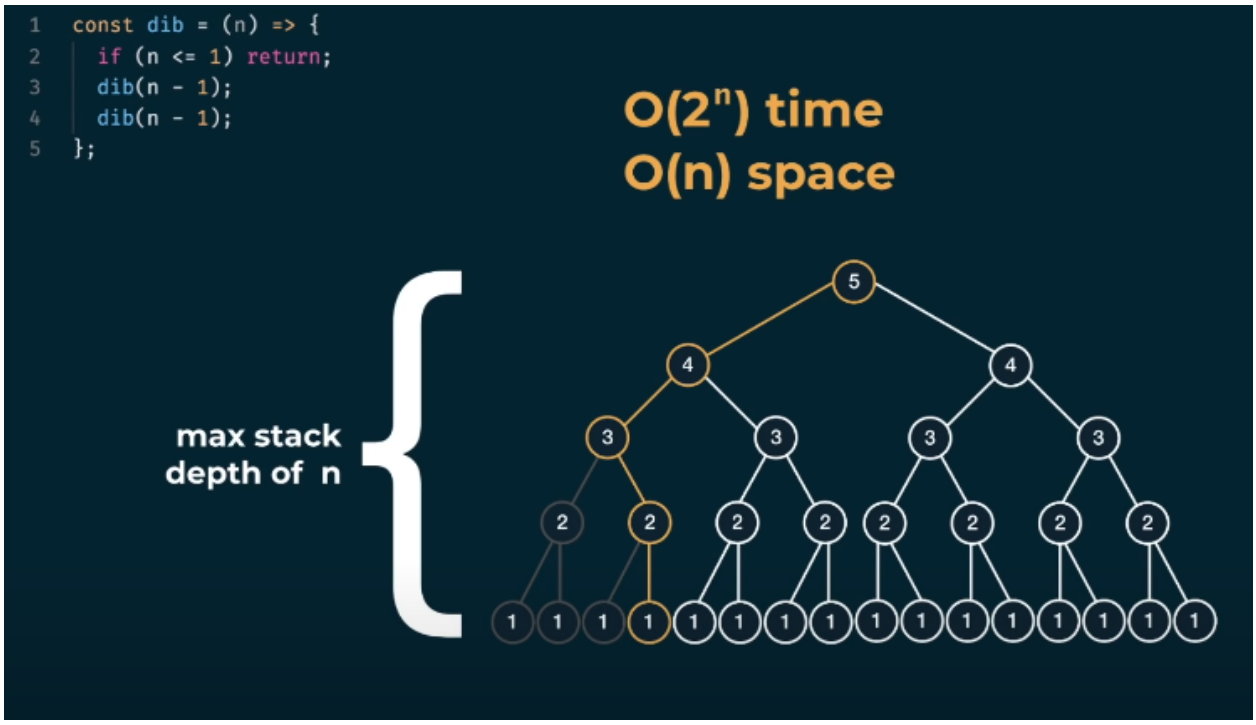
For smaller values, some "slower" functions take less steps. However, we want to think about the limit as N goes to infinity, and there, the "faster" function will take fewer steps.



In the example, for values of N under 100, the linear function runs faster, but for bigger values, the logarithmic function takes less steps!

Function	Order of Growth
$Ne^N + 2N$	$O(Ne^N)$
$-.6N + 7.8N^2 + 7$	$O(N^2)$
$5N^4 - \sin(N)$	$O(N^4)$
$500 - N + 3N!$	$O(N!)$
$2^N + N^{19}$	$O(2^N)$

Recursive Example



Common Mistakes

*****Make sure you take arbitrary HIGH values of n**

Multiplying O(n) bounds when the code segments do not nest:

```
int a = 0;
for(int i = 0; i < n; i++) { //segment 1
```

```
      a++;
}
for(int j = 0; j < 3*n; j += 4) { //segment 2
      a--;
}
for(int i = 0; i < n+7; i++) { //segment 3
  a += 1;
}
```

This code segment is O(n), not O(n^3). The for-loops are not nested, so you do not multiply. You can think about this as O(segment 1) + O(segment 2) + O(segment 3)= O(n) + O(n) + O(n) = O(n)

Not considering extremely (infinitely) large values of N:

```
int i = 0;
while (i < n ) {
      if(n < 10000) {
      for(int k = 0; k < 2*n; k++){
            System.out.println("hi");
      }
      }
      i++;
}
```

This looks like O(n^2), but for large values of N, the if statement is false, skipping over the for-loop. Therefore, only the outer loop runs, leading to O(n).