0. Big O

Big O

Time Complexity

Big O describes an upper bound on the time. An algorithm that prints out all the values in array and can be labeled as O(N). This can also be described as the worst case scenario. If every possiblity has to be exhausted before reaching the solution. Thus why a **for** loop is always at least **N**.

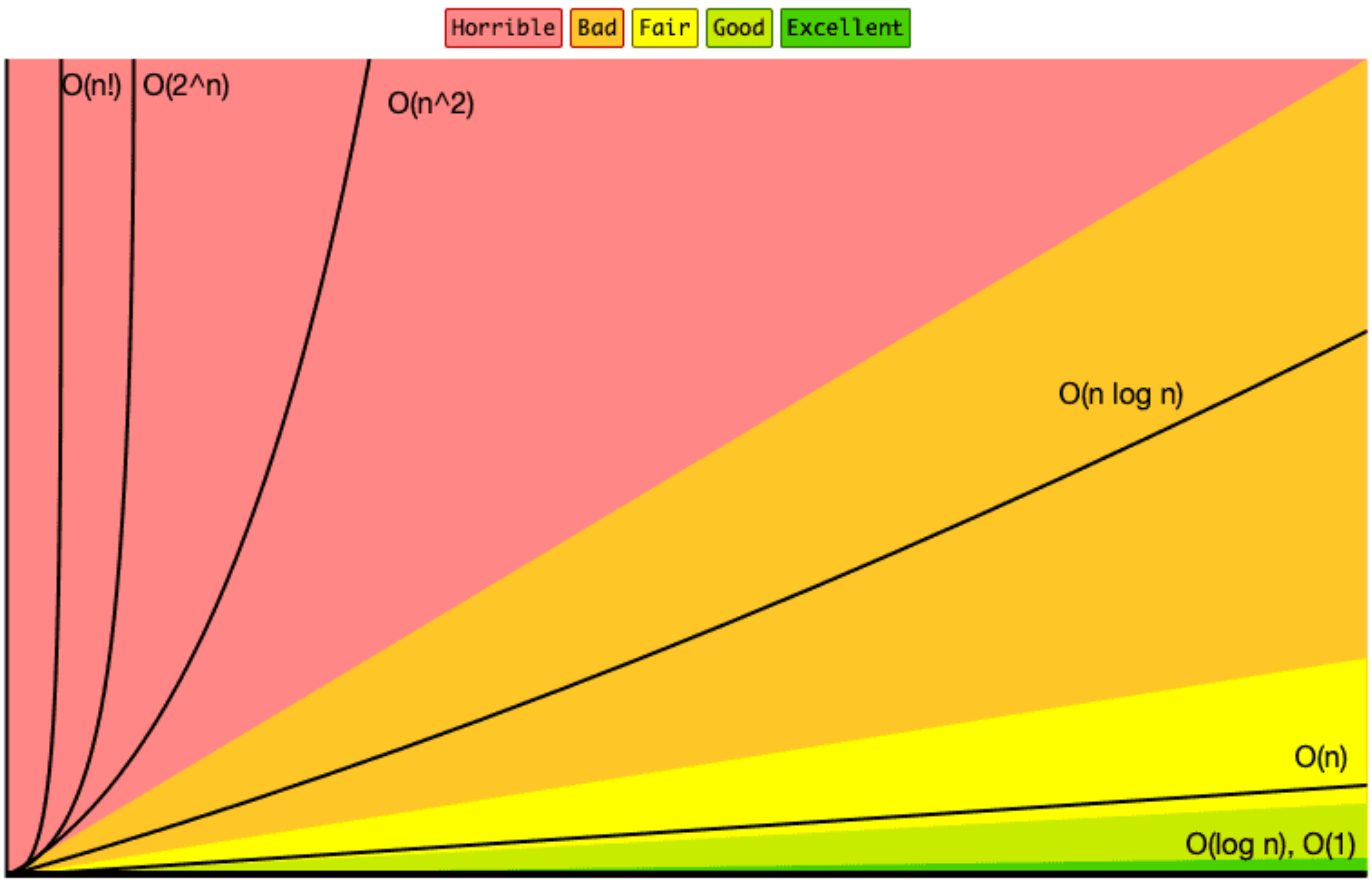
Space Complexity

This is the concept of how much memory is required to function. This is a parrallel concept to time. If we need to create an array of size **N** our space is now **O(N)**. If a 2D array is needed then our space complexity is now **O(N^2)**

Simple Rules

- Drop the constants of a calculated runtime. ie: O(2N) === O(N)
- Drop non-dominant terms. ie: O(N^2 + N) === O(N^2)

Runtime Visual



Multi-Part Algorithms: Add vs. Multiply

Consider the following when do you add or multiply the runtimes?

Add the Runtimes: O(A + B)

```
1  for (int a : arrA) {
2    print(a);
3  }
4
5  for (int b : arrB) {
6    print(b);
7  }
```

Multiply the Runtimes: O(A*B)

```
1  for (int a : arrA) {
2    for (int b : arrB) {
3      print(a + ",” + b);
4    }
5  }
```

On the left we ADD the runtimes, since the loops are independent. Which then comes out to be **O(A + B)**

On the right we MULT the runtimes since the loops are nested. Which then comes out to be **O(AB)**

Good rule of thumb: If the algrorithm is "Do this, then do that", you add. If its more along the lines of "do this each time you do that" you multiply.

Log N Runtimes

Seeing **O(log N)** is not uncommon to see, but where does this come from?
If we look at a binary search algorithm:

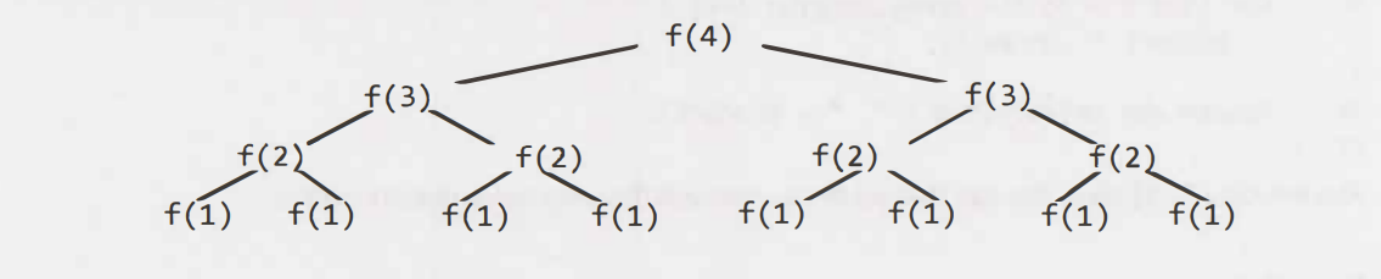
```
int binarySearch(int arr[], int left, int right int target) {
    if (right >= left)
        int mid = left + (right - 1) / 2;

        if(arr[mid] == target)
            return target;

        if(arr[mid] > target)
            return binarySearch(arr, left, mid - 1, target);

        return binarySearch(arr, mid + 1, right, target);
}
```

If you can understand what a binary search is doing you can see what is going on. Here we are searching for a target within a (sorted) array. We check our bounds of array (left-mid/ mid-right) then check which of those bounds the target falls in. We take it and search that again until we get the target.
So we are HALVING our search space every iteration thus we get **O(log N)**.
See a problem that halves the search space **O(log N)** is in the runtime atleast .

Recursive Runtimes

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

Whast the runtime here? Think of the call stack -> how many times are we calling f (4)?



So we can see we get $\mathbf{O(2^N)}$. The tree will be of depth N, and each node has two children. There fore each new level will have twice as many nodes as the level above it.