Amortized and Basic Runtimes

1 Amortized Runtime

Though we frequently want the worst case runtime, the more realistic case is that we want the average runtime, this is called *amortized runtime.* Amortized runtime tends to choose a worst case input and then calculating how long a series of operations would take on an input of that size For example, say we want to calculate the amortized runtime when adding to an array, we can do it as follows:

$$\frac{(1+1+1...)+N}{N} \rightarrow \frac{N+N}{N} \rightarrow \frac{2N}{N} \rightarrow 2 \text{ amortized runtime is } \Theta(1) \quad \text{N is the size of the array}$$

The above equation comes from the fact that N adding operations done in constant time and there is 1 call to resize the array which takes N time. We divide this by the total amount of operations which is N -1 and as a result we get $\Theta(1)$ as the amortized runtime.

It is imperative that amortized runtime not be confused with average case runtime. Average-case implies that the input is "average" whereas amortized runtime refers to the runtime of a worst case input in the long run. Amortized runtime will become increasingly important when we start to discuss data structures.

2 Calculating Basic Runtimes

We have gone over what runtime is and the basic notations for it; now we are going to learn how we can calculate the runtimes of some basic functions. Before we get started with actually calculating the runtimes, we must define some formulas that will be helpful in the future.

$$1+2+3+4+5+....+n = \frac{n(n+1)}{2} \rightarrow N^2$$

$$1+2+4+8+16+...+N = N$$

When attempting to figure out runtimes, it is important that we realize what we are trying to find the runtime in terms of. When there are multiple variables in play, we want to be explicit on what the running time is scaling to. Usually, we scale in reference to an input array (or other data structure) size, some number, or the length of some string. Now that we know some basic formulas, let's attempt to figure out some runtimes. Let's look at the following function:

```
public void hello(int n){
     for(int i = n; i > 0 ; i = i/2){
          System.out.println("Hello?"}
}
```

The running time for the above method would be $log(n)$. This is because the problem is divided by 2 each time. The i in the for loop will start off as $n \rightarrow \frac{n}{2} \rightarrow \frac{n}{4} ... \rightarrow 1$ leading to $log_2(n)$ amount of iterations which simplifies to $\Theta(log(n))$

Let's look at another problem, this time with while loops:

```
public void ptTwo(int[] arr){
      int i = 0;
      while(i < arr.length){
           i++;
          }
 }
```

The running time for the above method is n. This is because you are doing 1 iteration for each element of the array. Since there are N elements, you do N iterations.