Amortized Analysis

In this lesson, we'll cover the amortized analysis of algorithms.

This is used for algorithms where only a few of the operations are slow. We need to prove that the worst case of the entire algorithm is lower than the worst case of the particular slow operation times the number of operations.

Consider this algorithm: We start with an array of size 2 and each operation adds one element to the array, we do this operation N times. If the array is full, we see the current size of array say `sz`. Allocate the `2*sz` memory and copy the array to its location so we have space for the new `sz` elements.

- Adding to the array if it's not empty: $O(1)$
- Copying array of size `sz` to a new location: $O(sz)$

The second operation is obviously slower. We do this for N elements so the worst case would be $O(N^2)$. Yes! But it's easy to see that the second operation is not that frequent and will happen only $logN$ times. That means we can arrive at a more strict upper bound than $O(N^2)$.

Let's list the steps:

Step	Operations	Array Size / Max size
insert `e1`	1	$1/2$
insert `e2`	1	$2/2$
copy array of size 2 to new location of size 4 then insert `e3`	2 + 1	$3/4$
insert `e4`	1	$4/4$
copy array of size 4 to	4 + 1	$5/8$

new location of size 8 then insert `e5`		
insert `e6`	1	$6/8$
insert `e7`	1	$7/8$
insert `e8`	1	$8/8$
copy array of size 8 to new location of size 16 then insert `e9`	8 + 1	$9/16$

Total number of operations:

$1 + 1 + (1 + 2) + 1 + (1 + 4) + 1 + 1 + 1 + (1 + 8) + 1 + 1 \ldots$

=> $(1 + 1 + \ldots + 1)$ N times $+(2 + 4 + 8 + \ldots) < N + 2N <= 3N$

So, the complete algorithm runs in $O(N)$ time

In the next lesson, we will compare different runtimes and visualize growth.