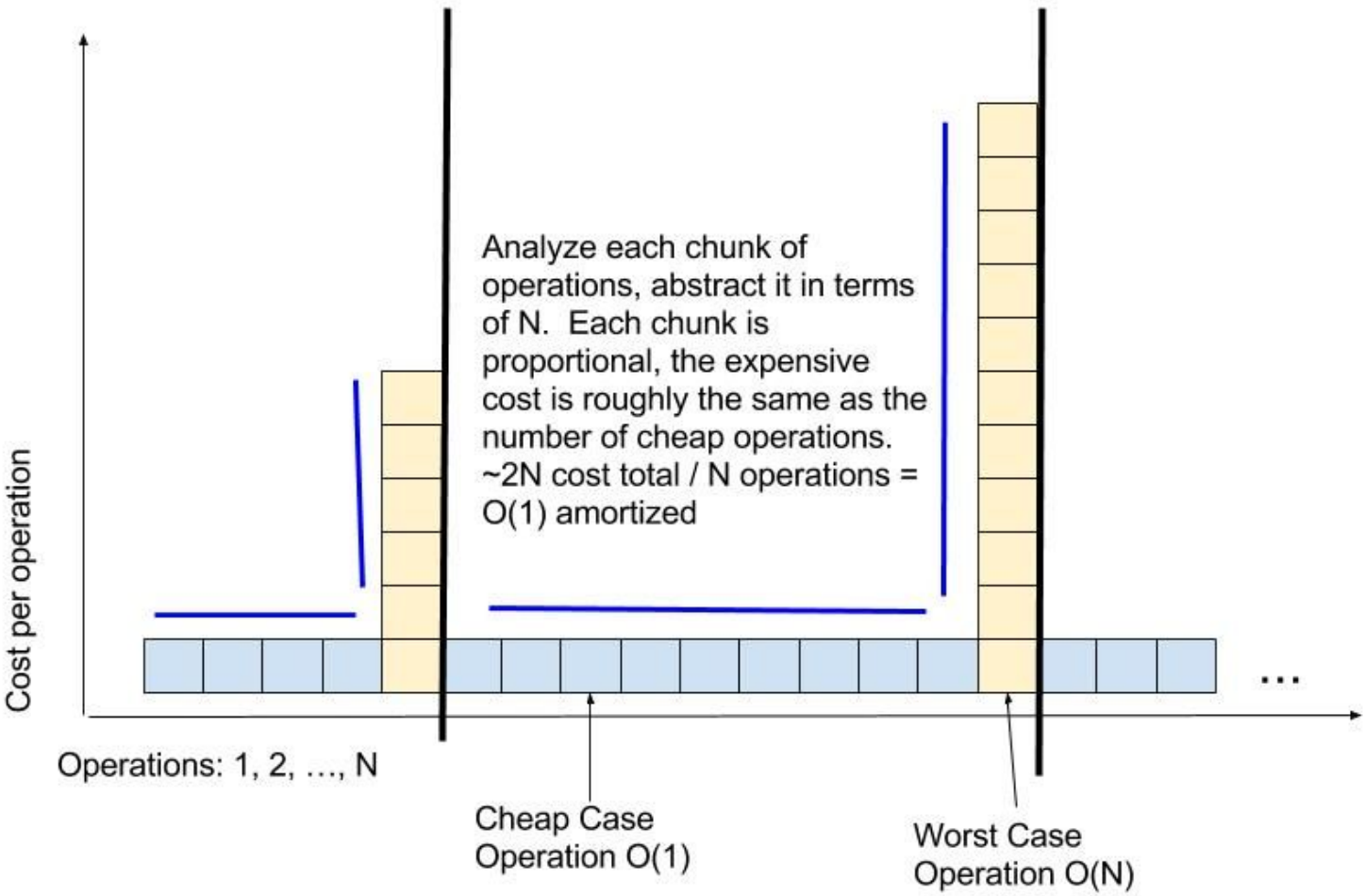
Amortized Runtime

When calculating amortized runtime, your goal is to compare the **total cost** of a series of operations with **how many** of those operations happened. The intuition here is that you want to "build up enough credit" with a series of cheap operations, so that when you have one (or more) expensive operations, you can average out the cost of the expensive one.

Amortized Example: Resizing an array when full

Resize Rule: When trying to insert when the array is full, make a new array that is double the size and copy over elements. Not exactly the same as, but similar to this picture:



You're ***always*** analyzing the following equation:

$$\frac{total\ cost\ of\ operations}{total\ number\ of\ operations}$$

Where an "operation" is the operation a client is doing through your public interface, like insert(5) or pop() or add(3).

Let's say you have an array with the 5 elements [1, 2, 3, 4, 5]

1	2	3	4	5

When you try to add an element 6, you need to resize your array:

1	2	3	4	5	6				

This operation was expensive! You had to copy over all of the elements into the new array before adding the 6. The analysis for the runtime of each of these operations is:

add(1) ----> O(1)
add(2) ----> O(1)
add(3) ----> O(1)
add(4) ----> O(1)
add(5) ----> O(1)
add(6) ----> O(N)

So you got 5 cheap operations before you got 1 expensive one. This continues, if you keep adding elements 7, 8, 9, 10, 11, when you add 11 you will have to do the same copy operation. So to continue the analysis for the runtime:

add(1) ----> O(1)
add(2) ----> O(1)
add(3) ----> O(1)
add(4) ----> O(1)
add(5) ----> O(1)
add(6) ----> O(N)
add(7) ----> O(1)
add(8) ----> O(1)
add(9) ----> O(1)
add(10) ----> O(1)
add(11) ----> O(N)

And this continues. So for after the first resize operation, you get 4 cheap operations before an expensive one. After the second resize operation, you get 9 cheap operations before an expensive one. And so on. The number of cheap operations you get before 1 expensive operation keeps increasing, since you double the array each time you resize:
5 initial cheap, 1 expensive, 4 cheap, 1 expensive, 9 cheap, 1 expensive, 19 cheap, …

This means that the number of cheap operations you get is increasing linearly as the size of your array increases. A.K.A. you get **N cheap operations, before 1 expensive operation**.

So, again, your job in analyzing this is to compare the **total cost** of a series of operations with **how many** of those operations happened.

$$= \frac{\frac{total\ cost\ of\ operations}{total\ number\ of\ operations}}{N\ cheap\ operations * O(1)\ cost + 1\ expensive\ operation * O(N)\ cost}{N + 1\ operations\ total}$$

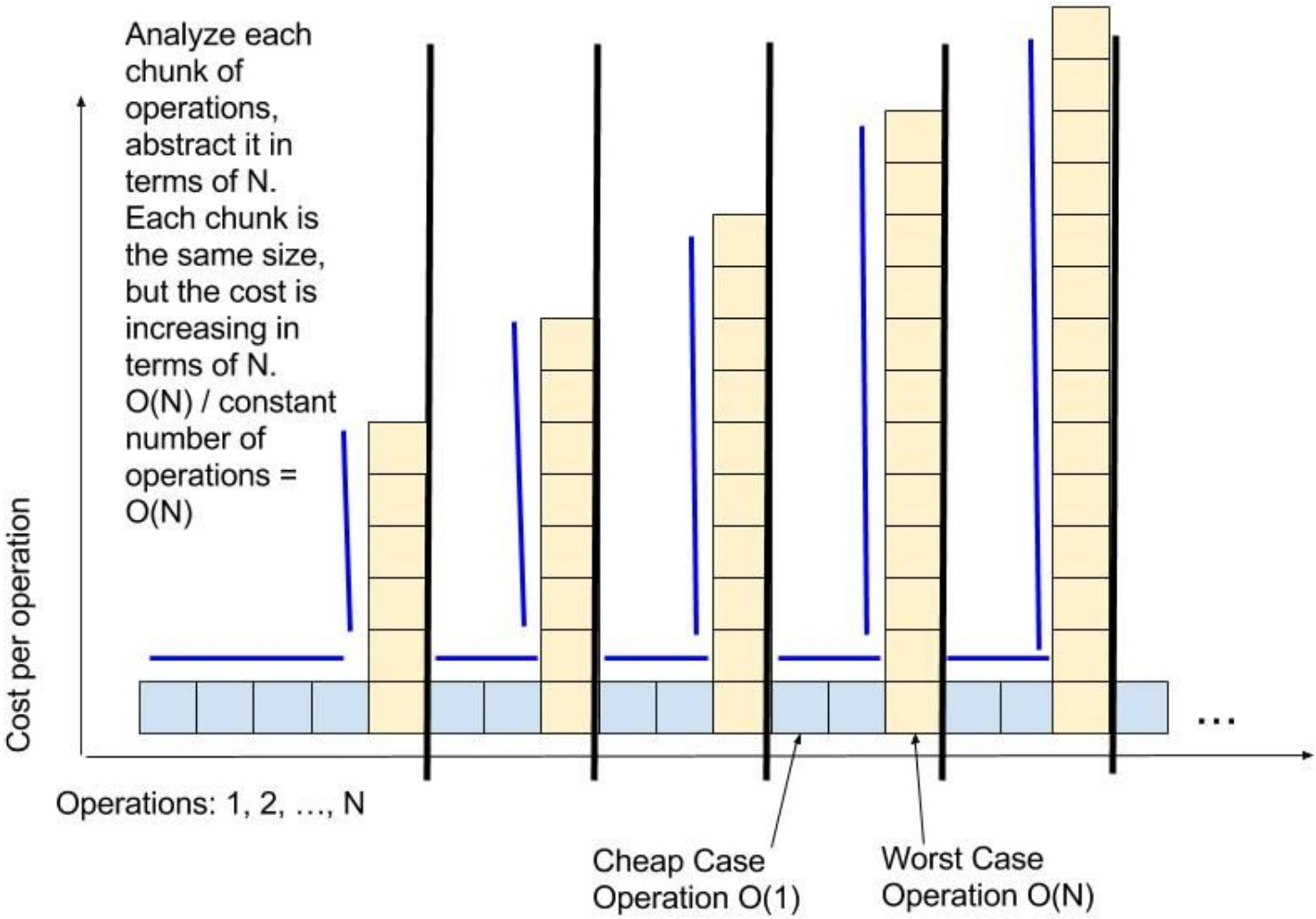
$$= \frac{O(N)}{N}$$

$$= O(1)$$

What matters here is that you had N cheap operations, and then 1 operation that cost O(N).

Not Amortized Example: Resizing an array when full

Resize Rule: When trying to insert when the array is full, make a new array that has 3 extra slots. Not exactly the same as, but similar to this picture:



Let's say you have an array with the 5 elements [1, 2, 3, 4, 5]

1	2	3	4	5

When you try to add an element 6, you need to resize your array:

1	2	3	4	5	6		

This operation was expensive! You had to copy over all of the elements into the new array before adding the 6. The analysis for the runtime of each of these operations is:

add(1) ----> O(1)
add(2) ----> O(1)
add(3) ----> O(1)
add(4) ----> O(1)
add(5) ----> O(1)
add(6) ----> O(N)

So you got 5 cheap operations before you got 1 expensive one. This continues, except now the copy operation happens for every 3 adds. So to continue the analysis for the runtime:

add(1) ----> O(1)
add(2) ----> O(1)
add(3) ----> O(1)
add(4) ----> O(1)
add(5) ----> O(1)
add(6) ----> O(N)
add(7) ----> O(1)
add(8) ----> O(1)
add(9) ----> O(N)
add(10) ----> O(1)
add(11) ----> O(1)
add(12) ----> O(N)
add(13) ----> O(1)
add(14) ----> O(1)
add(15) ----> O(N)

And this continues. So for after the first resize operation, you get 2 cheap operations before an expensive one. After the second resize operation, you get 2 cheap operations before an expensive one. And so on. The number of cheap operations you get before 1 expensive operation **stays constant**, since you double the array each time you resize:
5 initial cheap, 1 expensive, 2 cheap, 1 expensive, 2 cheap, 1 expensive, 2 cheap, …

So, again, your job in analyzing this is to compare the **total cost** of a series of operations with **how many** of those operations happened.

$$= \frac{\frac{\textit{total cost of operations}}{\textit{total number of operations}}}{\frac{2\ \textit{cheap operations} * O(1)\ \textit{cost} + 1\ \textit{expensive operation} * O(N)\ \textit{cost}}{2 + 1\ \textit{operations total}}}$$

$$= \frac{O(N)}{3}$$

$$= O(N)$$

What matters here is that you only had 2 cheap operations, and then 1 operation that cost O(N). You can't amortize the O(N) expensive operation over the 2 cheap operations you get from adding only 3 extra slots to your array.

Summary:

You are always comparing the **total cost** of a series of operations with **how many** of those operations happened. So you are trying to compare the **number of cheap operations** you get with **how expensive your expensive operation** is.

You're ***always*** analyzing the following equation:

$$\frac{\textit{total cost of operations}}{\textit{total number of operations}}$$

Where an "operation" is the operation a client is doing through your public interface, like insert(5) or pop() or add(3).