Dynamic Array Resizing: The Hidden Cost of Fixed Growth

Understanding why constant-size array resizing leads to quadratic time complexity—and why most programming languages avoid this for good reason.

The Problem: Fixed-Size Growth Strategy

Traditional Approach

When a dynamic array runs out of space, we must create a new, larger array and copy all existing elements. The question is: how much larger should the new array be?

One seemingly reasonable approach is to add a fixed number of slots (k) each time we resize. However, this innocent-looking strategy hides a performance trap.

The Mathematical Reality

If we store n elements and add k slots per resize, the mathematics become clear:

- After t resizes, capacity grows by $t \cdot k$
- We need at least $t \cdot k \geq n$
- Therefore: $t \geq n/k$ resizes required

This relationship forms the foundation of our complexity analysis.

When a dynamic array is resized by a fixed amount k, the total number of elements copied over time forms an arithmetic series. Let's trace the work done:

- 1st resize: k elements copied.
- 2nd resize: $2k$ elements copied.
- 3rd resize: $3k$ elements copied.
- ...
- t-th resize: tk elements copied.

The total number of copy operations after t resizes is the sum of an arithmetic series:

$$\sum_{i=1}^{t} ik = k \sum_{i=1}^{t} i = k \frac{t(t+1)}{2}$$

Since we established that $t \approx n/k$ (for n elements, and k added slots per resize), we can substitute t in the formula:

$$\text{Total Copies} \approx k \frac{(n/k)(n/k+1)}{2} = k \frac{n^2/k^2 + n/k}{2} = \frac{n^2}{2k} + \frac{n}{2}$$

Therefore, the total work done (copy operations) is proportional to n^2. This leads to a time complexity of $\Theta(n^2)$ for n insertions, making the fixed-size growth strategy highly inefficient for large datasets.

Amortised Analysis: Cost Per Operation

Amortised analysis examines the average cost per operation across a sequence of operations:

$$\text{Amortised cost} = \frac{\text{Total cost}}{\text{Number of operations}}$$

From our previous calculation:

- Total cost: $\Theta(n^2/k)$
- Number of operations: n
- Amortised cost per operation: $\Theta(n/k)$

If k is constant, this becomes $\Theta(n)$ per operation—linear amortised time, which is far worse than the $O(1)$ we'd prefer.

$$O(n^2)$$

Total Time

For n insertions

$$O(n)$$

Per Operation

Amortised cost

Key Takeaways from Fixed-Size Growth

Inefficient Strategy

Fixed-size increments lead to poor performance for dynamic array resizing.

Quadratic Complexity

The total work for n insertions is $\Theta(n^2)$ due to extensive copying.

Linear Amortised Cost

Each operation, on average, takes $\Theta(n)$ time, making it highly impractical for large datasets.