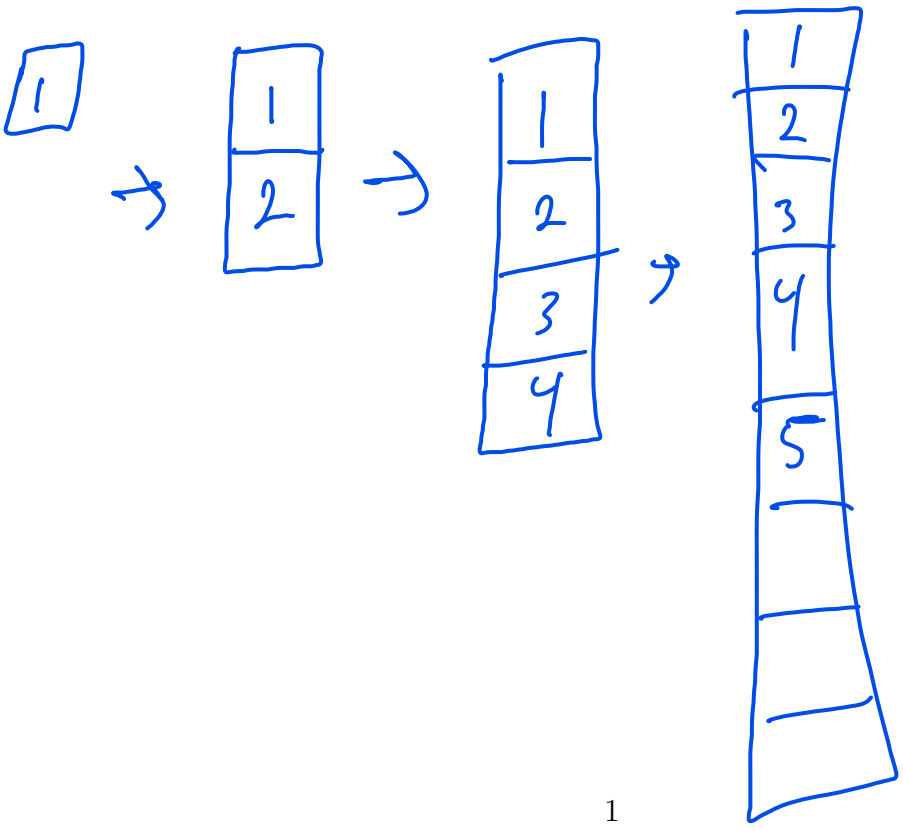
Suppose we are working with an array. Elements can be inserted into the array, and each inserted element is "appended" to the rest of the elements (if we have made i insertions then the elements should be in the first i positions of the array).

We may not know how many elements will need to be inserted into our data structure. We will begin by maintaining a small array, and then increase the size of the array if there is not enough space to insert a new element.

- Like vectors in C++ and lists in Python.



How many steps do we take to insert n elements into our dynamic array? In the worst case, a single insertion will require $O(n)$ steps (double the size of the array and copy the existing elements into the new array).

Therefore we have an upper bound on the number of steps of $O(n^2)$.

The upper bound is not tight. The worst case is $O(n)$ steps for an insertion, but there will be *many* insertions which just use 1 step.

Rather than just upper bounding each insertion by the worst case, can we use a more careful analysis to achieve a smaller upper bound?

Let c_i denote the cost of the ith insertion.

- $c_i = 1+$(steps to double array in insertion i)

i	1	2	3	4	5	6	7	8	9	10
$size_i$	1	2	4	4	8	8	8	8	16	16
c_i	1+0	1+1	1+2	1+0	1+4	1+0	1+0	1+0	1+8	1+0

$$\sum_{i=1}^{n} c_i = n + \sum_{j=0}^{\log_2 n} 2^j \leq n + 2n = 3n \in O(n)$$

This type of analysis is known as an **amortized analysis** where we collectively analyze a sequence of operations. There are two main ways amortized analysis can be carried out:

- **Aggregate analysis**: show that the entire sequence of n operations takes $T(n)$ time (the analysis we just did was an aggregate analysis).
- Upper bound the **amortized cost** of a single operation. That is, provide an upper bound on the average cost of a single operation.

Analyzing the amortized cost of an operation is useful when you have many cheap operations and a few operations which will be much more expensive.

Note that amortized analysis deals with averages, there is no probability involved. We use this analysis to show the average cost is small in the *worst case* rather than in the *average case*.

We just saw an example of aggregate analysis. It is simple, but lacks in precision (it may be difficult to determine the exact amortized cost of a single operation).

Two methods which allow us to determine the amortized cost of an operation:

- Accounting method
- Potential method

We will now discuss the **accounting method**. The idea is that each operation will pay some amount of money to a bank ($1 pays for 1 "step" of an algorithm). When the algorithm needs to perform a task, it "pays" for it from the money in the bank.

The idea is that for each cheap operation (inserting into dynamic array when we have space) pays more than what it takes to cover it's own operation, so money accumulates when a cheap operation is performed.

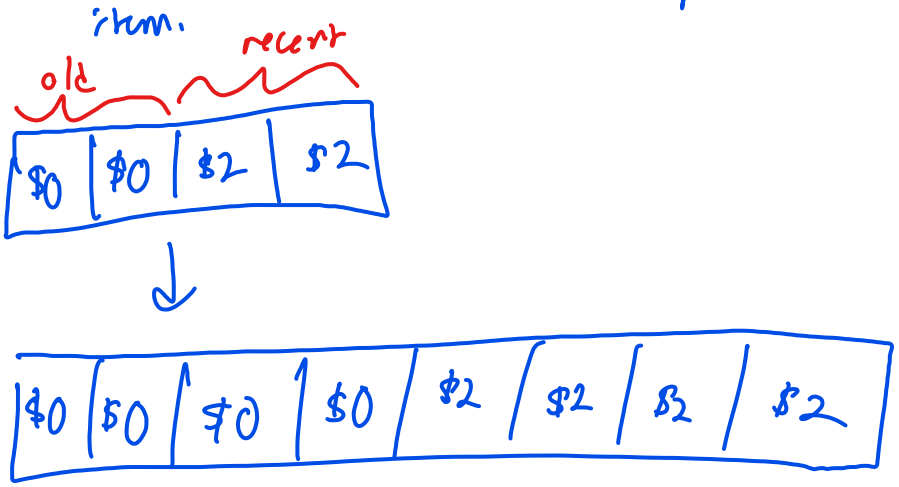
When an expensive operation is performed (doubling the array), there will be enough money in the bank to cover the cost of that operation.

If we show that there will always be enough money in the bank to perform any operation, then we have that the amortized cost of an operation is the amount it paid to the bank.

In our dynamic array example, suppose we each insertion operation pays \$3 to the bank. We will show that there will always be enough money in the the bank to pay for any operation.

\$1 pays for immediate insertion
\$2 stored in bank for later doubling.

When the array doubles, \$1 pays to move a "recent" item and \$1 pays to move an "old" item.



Now we will discuss the **potential method**. This is a different idea that is similar to the accounting method. Intuitively, we have a potential function Φ which is equal to the amount of money saved in the bank at a particular time.

Framework:

- Start with initial data structure D_0.
- Operation i transforms D_{i-1} into D_i.
- Cost of operation i is c_i
- Potential function $\Phi : \{D_i\} \to \mathbb{R}$ such that $\Phi(D_0) = 0$ and $\Phi(D_i) \geq 0$.
- Amortized cost $\hat{c}_i$ with respect to Φ is $\hat{c}_i := c_i + \Phi(D_i) - \Phi(D_{i-1})$

Total Amortized cost:

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} \left(c_i + \Phi(D_i) - \Phi(D_{i-1}) \right)$$

$$= \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_{n-1}) + \Phi(D_{n-1}) - \Phi(D_{n-2}) + \Phi(D_{n-2}) - \Phi(D_{n-3})$$

$$\cdots - \Phi(D_1) - \Phi(D_0)$$

$$= \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0)$$

$\Phi(D_n) \geq 0$, $\Phi(D_0) = 0$

$$\Rightarrow \sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i$$

Potential Function Φ:

$$\Phi(D_i) = 2i - size_i$$

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$$

$$= \begin{cases} 1 + (2i - size_i) - (2(i-1) - size_{i-1}) & \text{if } i-1 \text{ is not a power of 2} \\ i + (2i - size_i) - (2(i-1) - size_{i-1}) & \text{o/w} \end{cases}$$

$i-1$ is a power of 2

$$\hat{C}_i = i + (2i - \text{size}_i) - (2(i-1) - \text{size}_{i-1})$$

$2(i-1)$ $\qquad$ $(i-1)$

$$= i + 2i - 2i + 2 - i + 2 + 1 - 1$$

$$= 3$$

$i-1$ is not a power of 2

$$\hat{c}_i = 1 + (2i - \text{size}_i) - (2(i-1) - \text{size}_{i+1})$$

Same

$$= 1 + (2i - \text{size}_i) - 2i + 2 + \text{size}_{i-1}$$

$= 3$