# CHAPTER 6:
# VECTORS, LIST, AND ITERATORS


# VECTORS

# VECTORS

- A **vector** is a collection of elements, each associated with an index, just like an array**.**
- The main differences are that, unlike arrays:
  - vectors can **insert or remove an element at any index**
  - vectors can **grow in size**; this happens either when *inserting elements*, or when calling a method to *reserve memory space*.

# VECTORS: ADT

- Formally, a vector is an Abstract Data Type (ADT) that supports :
  - `at(i):` **Return the element** at index `i` (error if `i` is out of range)
  - `set(i,e):` **Replace the element** at index `i` with `e` (error if `i` is out of range)
  - `insert(i,e):` **insert** `e` at index `i` (error if `i` is out of range; $0 \leq i \leq n$)
  - `erase(i):` **Remove the element** at index `i` (error if `i` is out of range)
  - `size():` Return the **number of element**s in the vector.
  - `empty():` Return **`true` if the vector is empty** and `false` otherwise

# VECTORS: EXAMPLE

Now, let's fill this table!

P.S., under "**Output**" write:

- "error" if there is an error
- The value returned (if the function returns a value)
- "–" if there is **no error** and the function returns **no value**

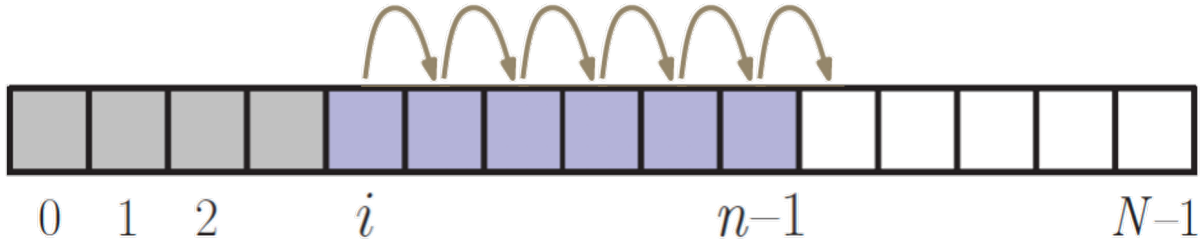
| *Operation* | *Output* | *V* |
|---|---|---|
| insert(0,7) | – | (7) |
| insert(0,4) | – | (4,7) |
| at(1) | 7 | (4,7) |
| insert(2,2) | – | (4,7,2) |
| at(3) | "error" | (4,7,2) |
| erase(1) | – | (4,2) |
| insert(1,5) | – | (4,5,2) |
| insert(1,3) | – | (4,3,5,2) |
| insert(4,9) | – | (4,3,5,2,9) |
| at(2) | 5 | (4,3,5,2,9) |
| set(3,8) | – | (4,3,5,8,9) |

# VECTOR IMPLEMENTATION

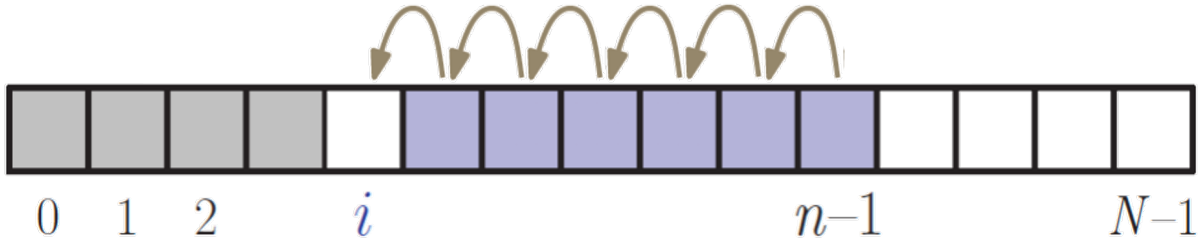
- As we said, the main differences between vectors and arrays are that:
  - vectors can **insert or remove an element at any index**
  - vectors can **grow in size**; this happens either when *inserting elements*, or when calling a method to *reserve memory space*.
- *How do we implement a vector using an array?*
  - To allow the vector to **insert or remove elements at any index**, we will have to keep shifting elements within the array.
  - To allow the vector to **grow in size**, we will simply create a very large array of size $N$. This way, whenever we add elements to the vector, the total number of elements, $n$, hopefully never exceeds $N$.

# ARRAY-BASED IMPLEMENTATION

- In an array-based implementation, where the array size is $N$, and vector contains $n$ elements:



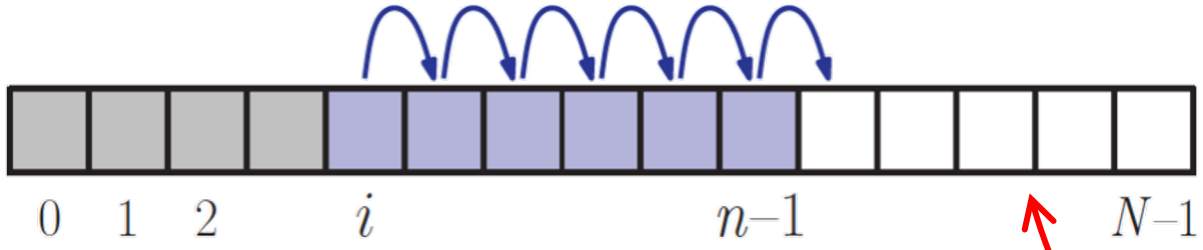

**Algorithm** insert($i$,$e$):
  **for** $j = n-1, n-2, \ldots, i$ **do**
    $A[j+1] \leftarrow A[j]$
  $A[i] \leftarrow e$
  $n \leftarrow n+1$



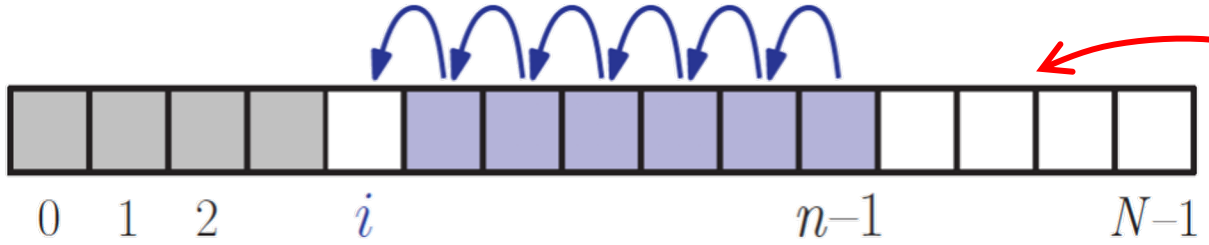

**Algorithm** erase($i$):
  **for** $j = i+1, i+2, \ldots, n-1$ **do**
    $A[j-1] \leftarrow A[j]$
  $n \leftarrow n-1$

# COMPLEXITY

- What is the complexity of these operations given an array-based implementation?




In the worst case, these would perform $n$ shifts (this happens when $i = 0$)

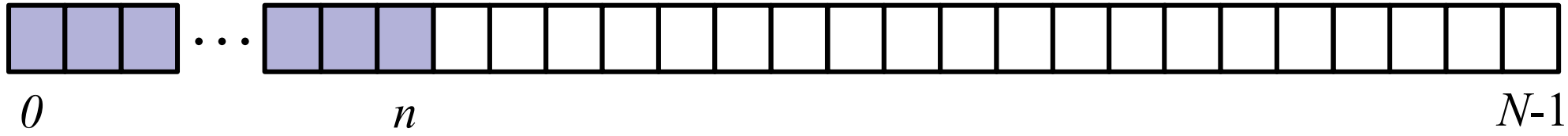



| *Operation* | *Time* |
|---|---|
| size() | |
| empty() | |
| at($i$) | |
| set($i, e$) | |
| insert($i, e$) | |
| erase($i$) | |

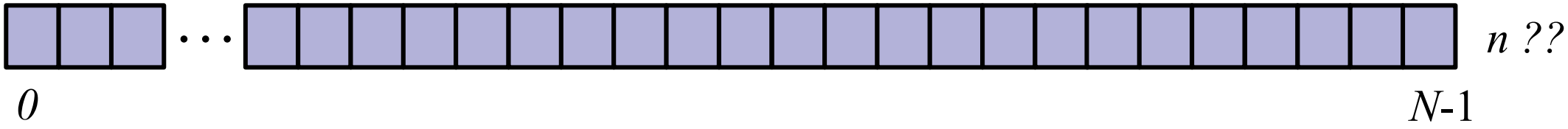
# PROBLEM WITH ARRAY'S CAPACITY

- One of the main disadvantages of an **array-based implementation of vectors** is that we have to *determine the maximum capacity*, $N$, in advance.

  - What if $N$ was **too large?** i.e., what if the number of actual elements, $n$, never reaches anywhere near $N$? We would waste memory space!




  - What if we set $N$ was **too small?** We would attempt to access an element at an index greater than N-1, and the implementation would crash!




We can improve this by using an "**extendable array**"!

## Basic idea of extendable arrays:

- Suppose the array is full, i.e., $n = N$ and we want to add 2 new elements sequentially. We can simply follow these steps.
- **Do you see a problem here?**

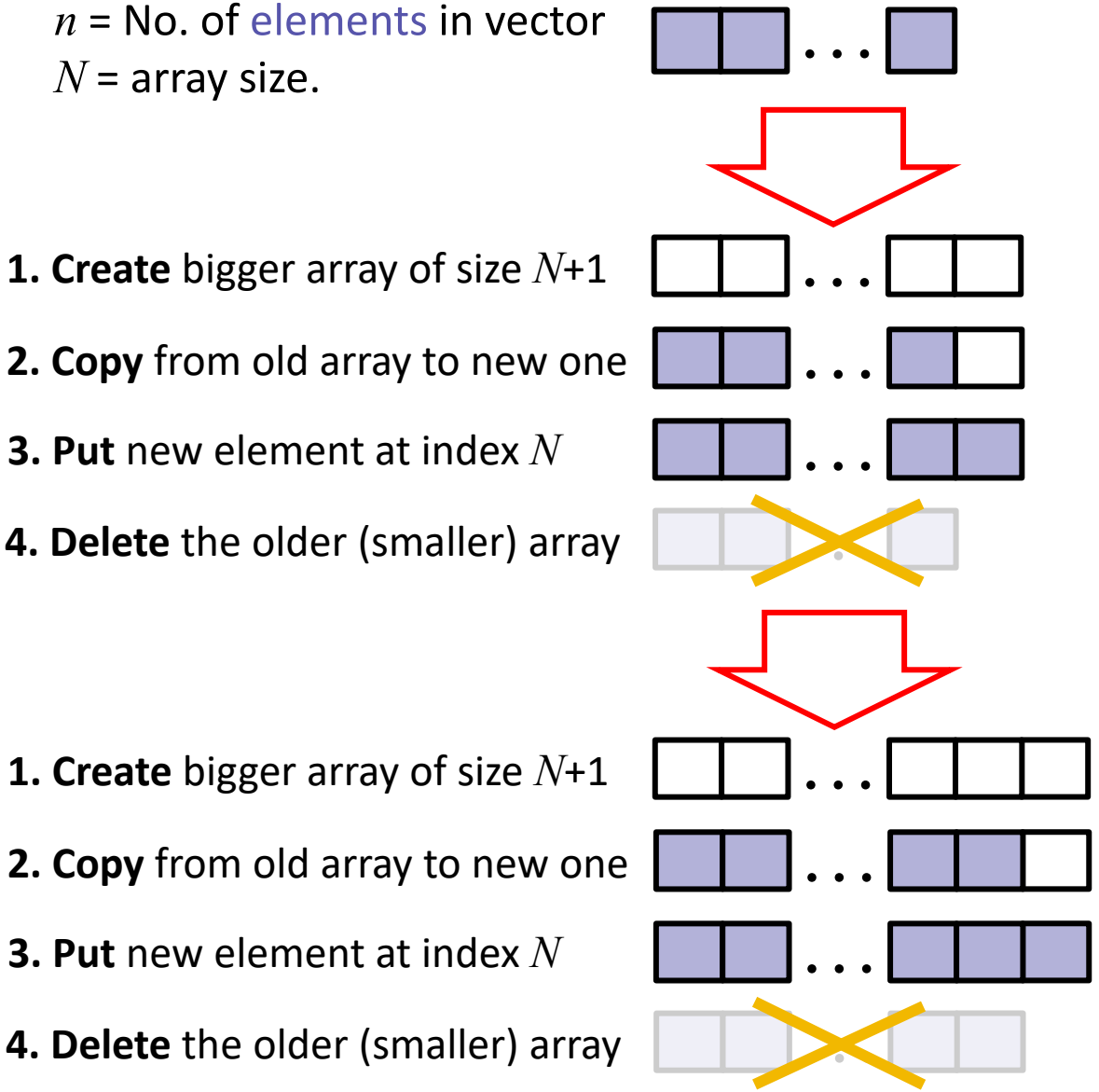
  This requires copying all elements every time a new element is added
- **What is the running time?**

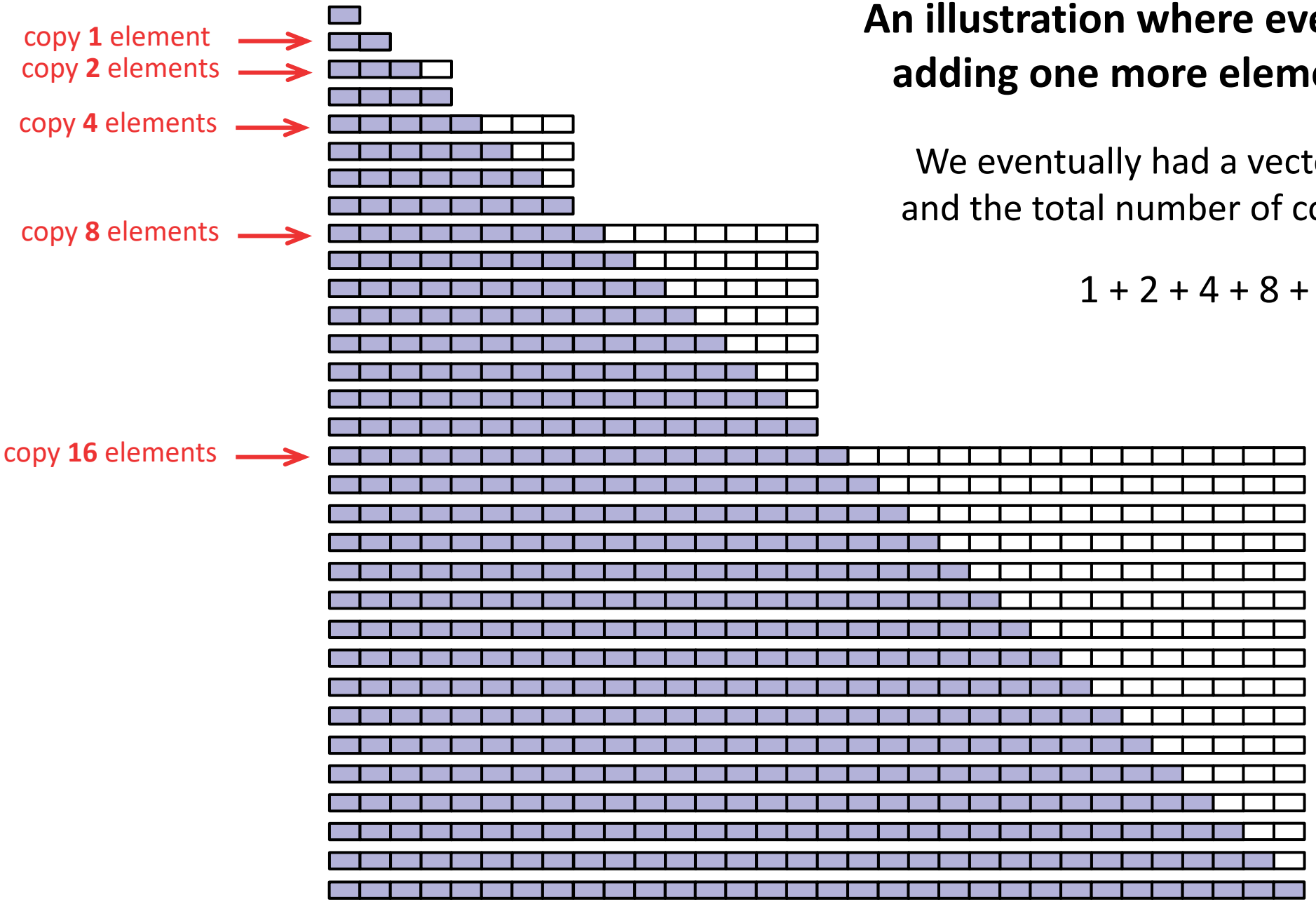
  If we start with an array that fits of size 1, and keep copying, the total number of copy-operations needed to put $n$ elements is:

  $1 + 2 + 3 + \ldots + n = \frac{n(n+1)}{2}$ → $O(n^2)$
- **Is there a more efficient design?**

  What if, whenever the array is full, we **double** the array size?

$n$ = No. of elements in vector
$N$ = array size.



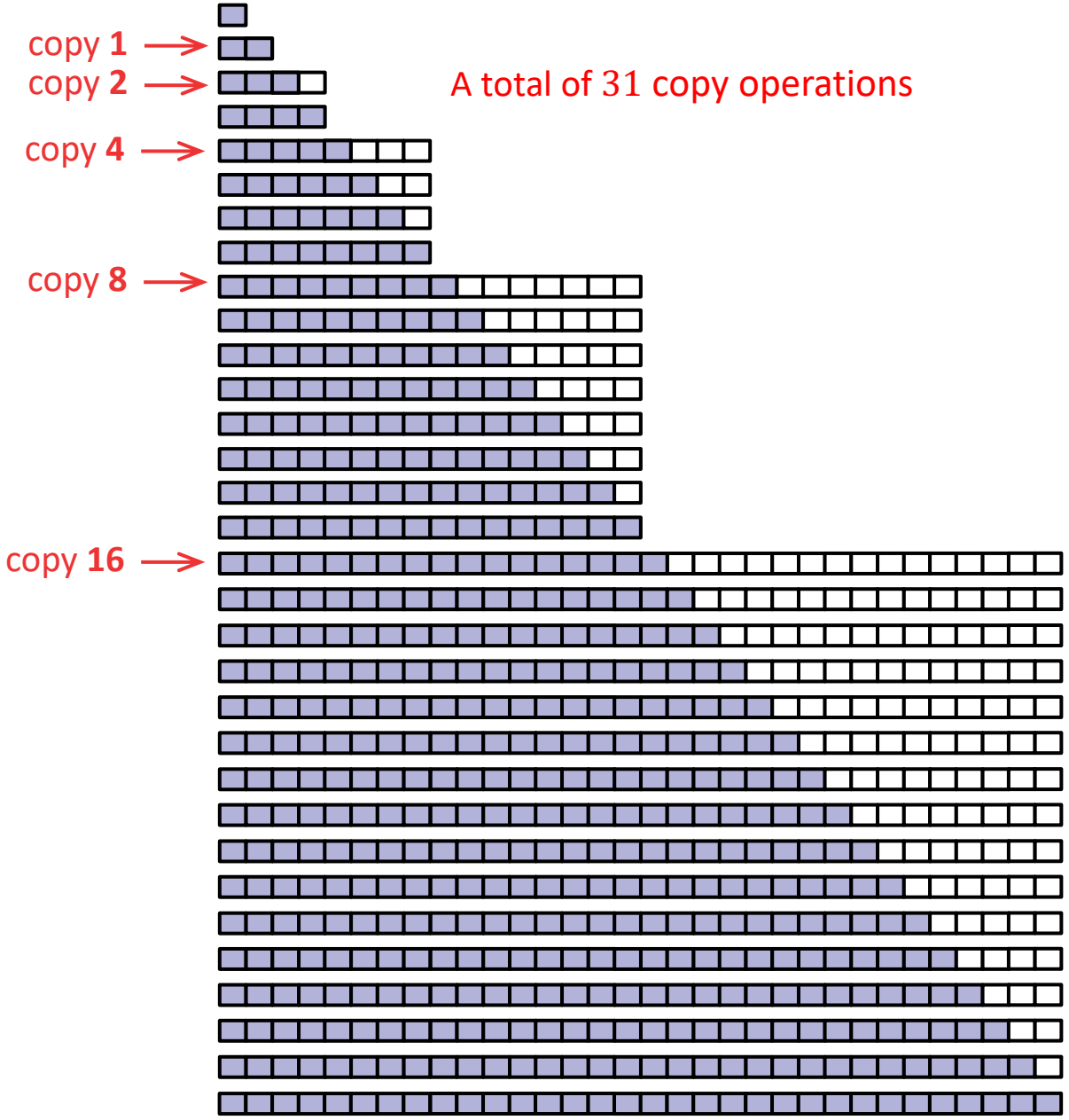



**An illustration where every row represents adding one more element to the vector.**

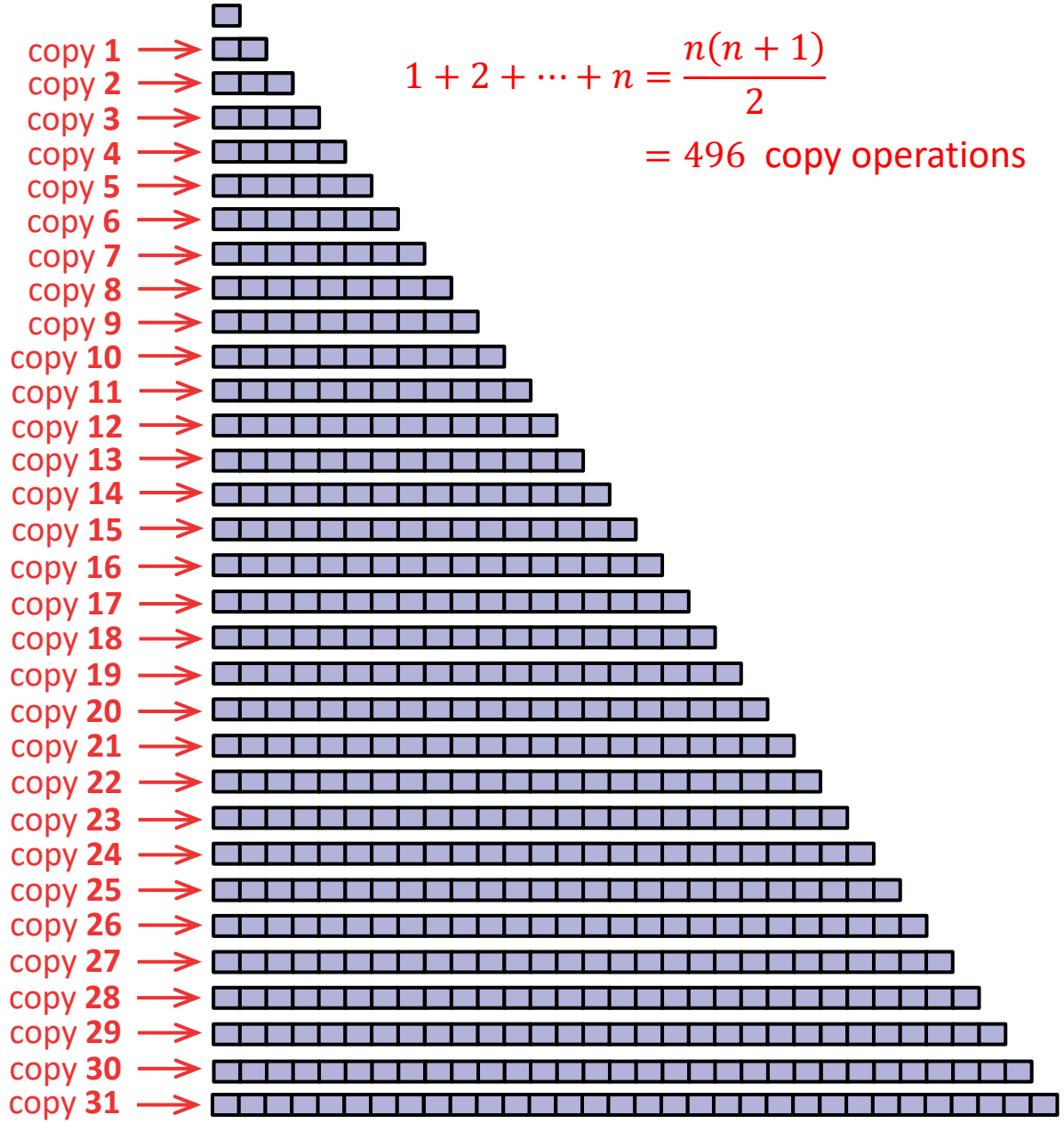
We eventually had a vector of 32 elements, and the total number of copies we made was

$1 + 2 + 4 + 8 + 16 = \mathbf{31}$

# Doubling the array size




# Increasing the array size by 1




$$1 + 2 + \cdots + n = \frac{n(n+1)}{2}$$

$= 496$ copy operations

# C++ IMPLEMENTATION

```
typedef int Elem; // A user-defined element type
class ArrayVector {
public:
   ArrayVector(); // constructor
   int size() const; // returns the number of elements "n"
   bool empty() const; // returns "true" if the vector is empty
   Elem& operator[ ](int i); // overloading the "[ ]" operator
   Elem& at(int i) throw(IndexOutOfBounds); // returns element at index "i"
   void erase(int i); // remove element at index "i"
   void insert(int i, const Elem& e) throw(IndexOutOfBounds); // insert "e" at index "i"
   void reserve(int N); // extend array size to N (unless it's already bigger than N)
private:
   int capacity; // current array size, i.e., "N"
   int n; // number of elements in the vector
   Elem* A; // Array of elements of type "Elem"
};
```

# C++ IMPLEMENTATION

```
typedef int Elem; // A user-defined element type
class ArrayVector {
public:
   ArrayVector(); // constructor
   int size() const; // returns the number of elements "n"
   bool empty() const; // returns "true" if the vector is empty
   Elem& operator[ ](int i);
   Elem& at(int i) throw(IndexOutOfBounds); // returns element at index "i"
   void erase(int i); // remove element at index "i"
   void insert(int i, const Elem& e) throw(IndexOutOfBounds);
   void reserve(int N); // extend array size to N (unless it's already bigger than N)
private:
   int capacity; // current array size, i.e., "N"
   int n; // number of elements in the vector
   Elem* A; // Array of elements of type "Elem"
};
```

```
// overloading the "[ ]" operator
Elem& ArrayVector::operator[ ](int i)
{ return A[i]; }
```

Now, we can write, e.g.,

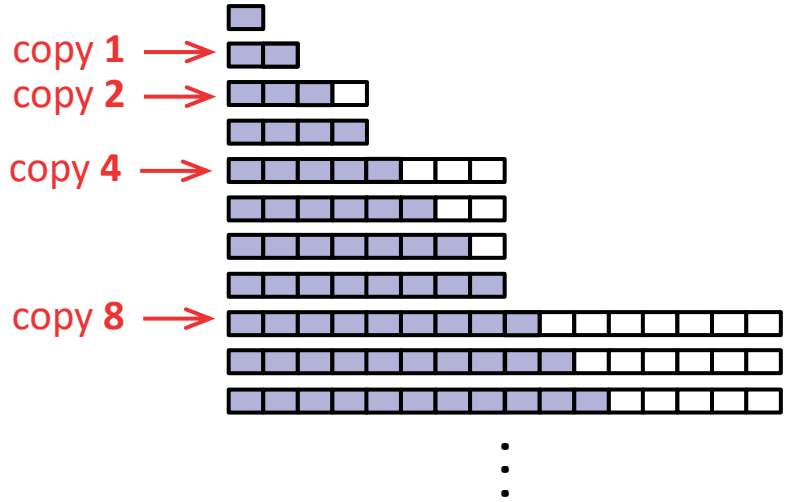
```
ArrayVector x = new ArrayVector();
...
x[0] = x[1];
cout << x[10];
```

# C++ IMPLEMENTATION

```
typedef int Elem; // A user-defined element type
class ArrayVector {
public:
   ArrayVector(); // constructor
   int size() const; // returns the number of elements "n"
   bool empty() const; // returns "true" if the vector is empty
   Elem& operator[ ](int i);
   Elem& at(int i) throw(IndexOutOfBounds); // returns element at index "i"
   void erase(int i); // remove element at index "i"
   void insert(int i, const Elem& e) throw(IndexOutOfBounds);
   void reserve(int N);
private:
   int capacity; // current array size, i.e., "N"
   int n; // number of elements in the vector
   Elem* A; // Array of elements of type "Elem"
};
```

```
// extend array size to N (unless it's already bigger than N)
void ArrayVector::reserve(int N) {
    if (capacity >= N) return;
    Elem* B = new Elem[N]; // resize the array
    for (int j = 0; j < n; j++) // copy contents to new array
        B[j] = A[j];
    if (A != NULL) delete [ ] A; // discard old array
    A = B; // make B the new array
    capacity = N; // set new capacity
```

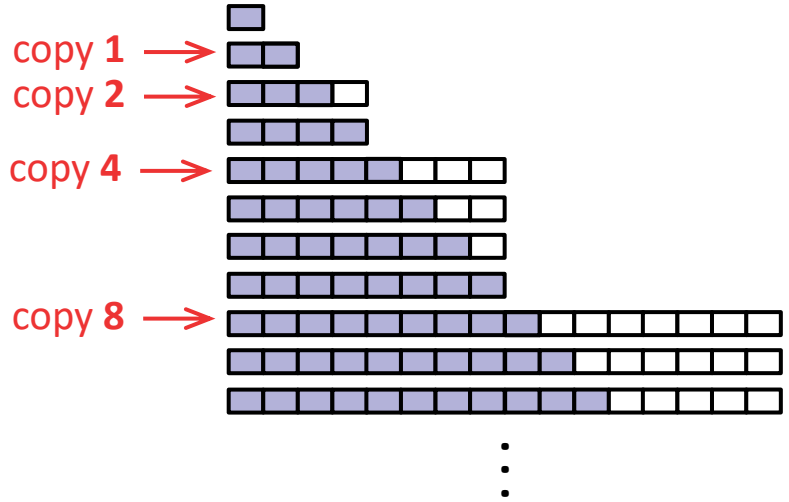
# C++ IMPLEMENTATION




```
typedef int Elem; // A user-defined element type
class ArrayVector {
public:
   ArrayVector(); // constructor
   int size() const; // returns the number of elements "n"
   bool empty() const; // returns "true" if the vector is empty
   Elem& operator[ ](int i);
   Elem& at(int i) throw(IndexOutOfBounds); // returns element at index "i"
   void erase(int i); // remove element at index "i"
   void insert(int i, const Elem& e) throw(IndexOutOfBounds);
   void reserve(int N);
private:
   int capacity; // current array size, i.e., "N"
   int n; // number of elements in the vector
   Elem* A; // Array of elements of type "Elem"
};
```

// insert "e" at index "$i$"
How would you implement the method "insert"?
P.S., you can use "reserve" in your implementation.

# C++ IMPLEMENTATION




```
typedef int Elem; // A user-defined element type
class ArrayVector {
public:
   ArrayVector(); // constructor
   int size() const; // returns the number of elements "n"
   bool empty() const; // returns "true" if the vector is empty
   Elem& operator[ ](int i);
   Elem& at(int i) throw(IndexOutOfBounds); // returns element at index "i"
   void erase(int i); // remove element at index "i"
   void insert(int i, const Elem& e) throw(IndexOutOfBounds);
   void reserve(int N);
private:
   int capacity; // current array size, i.e., "N"
   int n; // number of elements in the vector
   Elem* A; // Array of elements of type "Elem"
};
```

The capacity is multiplied by 2, unless the capacity was 0, in which case multiplying it by 2 won't help. In this case, we set the capacity to 1

```
void ArrayVector::insert(int i, const Elem& e)
throw(IndexOutOfBounds) {
   if (i<0||i>n)
      throw IndexOutOfBounds("illegal index)");
   if (n == capacity) // overflow?
      reserve( max(1, 2 * capacity) ); // double the size
   for (int j = n − 1; j >= i; j−−) // shift elements up
      A[j+1] = A[j];
   A[i] = e; // put "e" at index "i"
   n++; }
```

# COMPLEXITY OF "PUSHING" AN ELEMENT

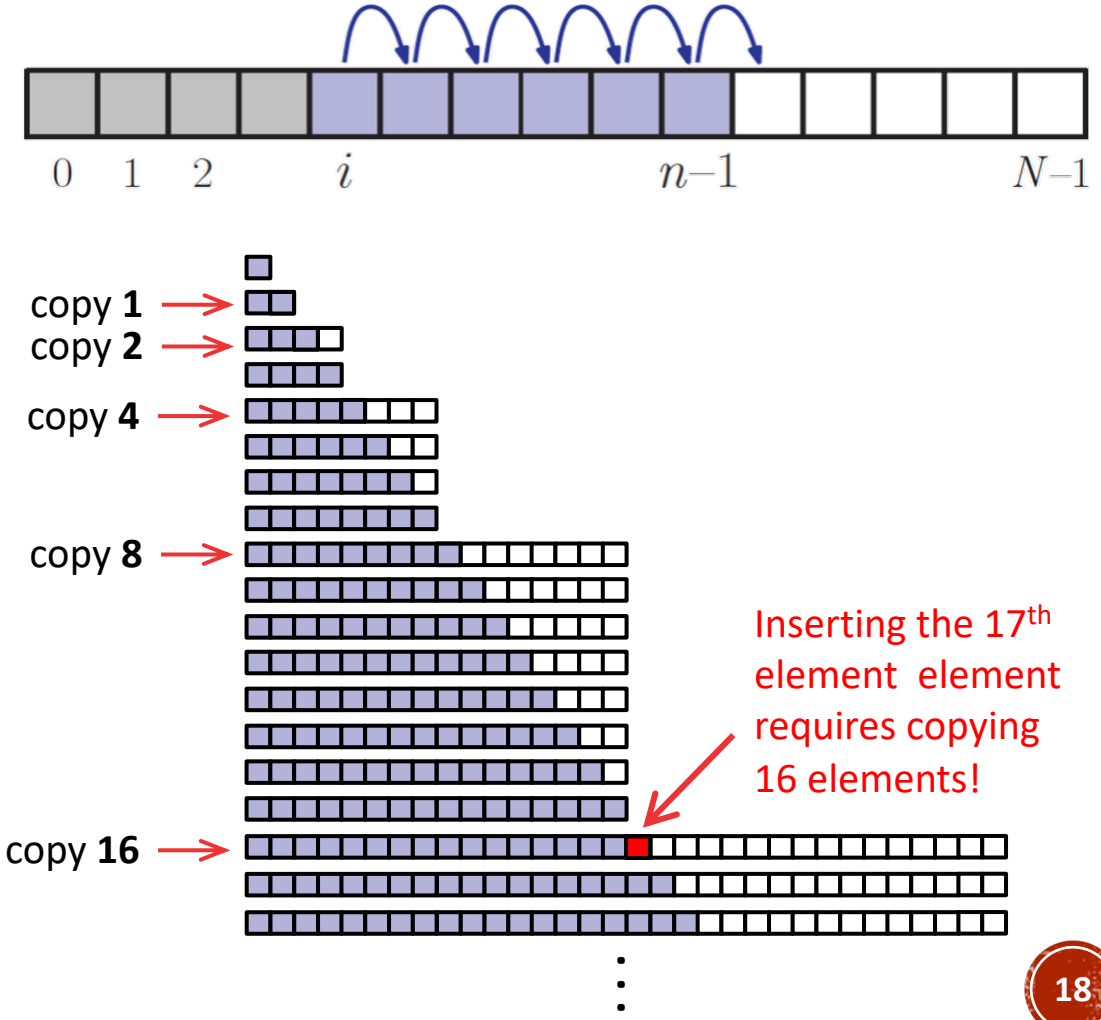
- Let us use the term "push" when inserting an element at index $n$ in the vector
- **What is the complexity** of push given a normal array (i.e., not extendable)?

  O(1), because no shifting is needed! But of course, this will crash if $n$ reaches $N$
- **What is the complexity** of push given an extendable array?

  O(n), the worst case requires copying $n$ values!

  But is this a fair assessment? Only very few "pushes" would involve copying $n$ values!

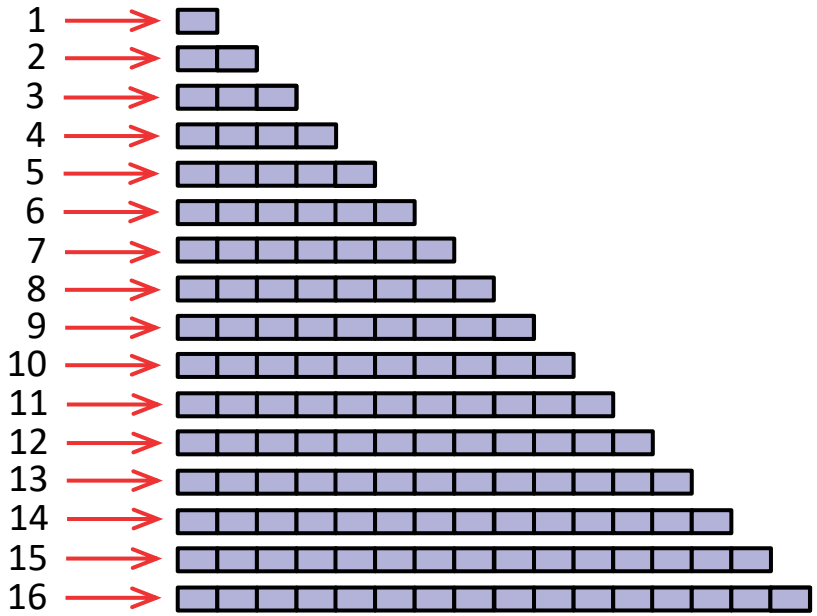
  In such cases, we need an **amortized analysis**!



# AMORTIZATION

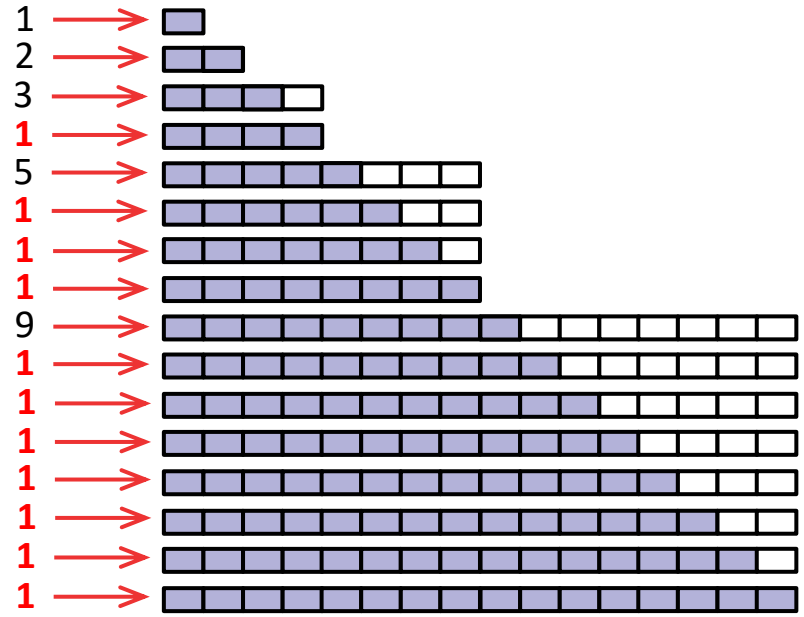
Given a naïve implementation, every push operation requires **copying the entire vector** to a new one of size $N$+1

No. of operations:




But if we "**invest**" by allocating memory in advance, then our investment pays off since many subsequent "push" operations would take O(1) time.

No. of operations:

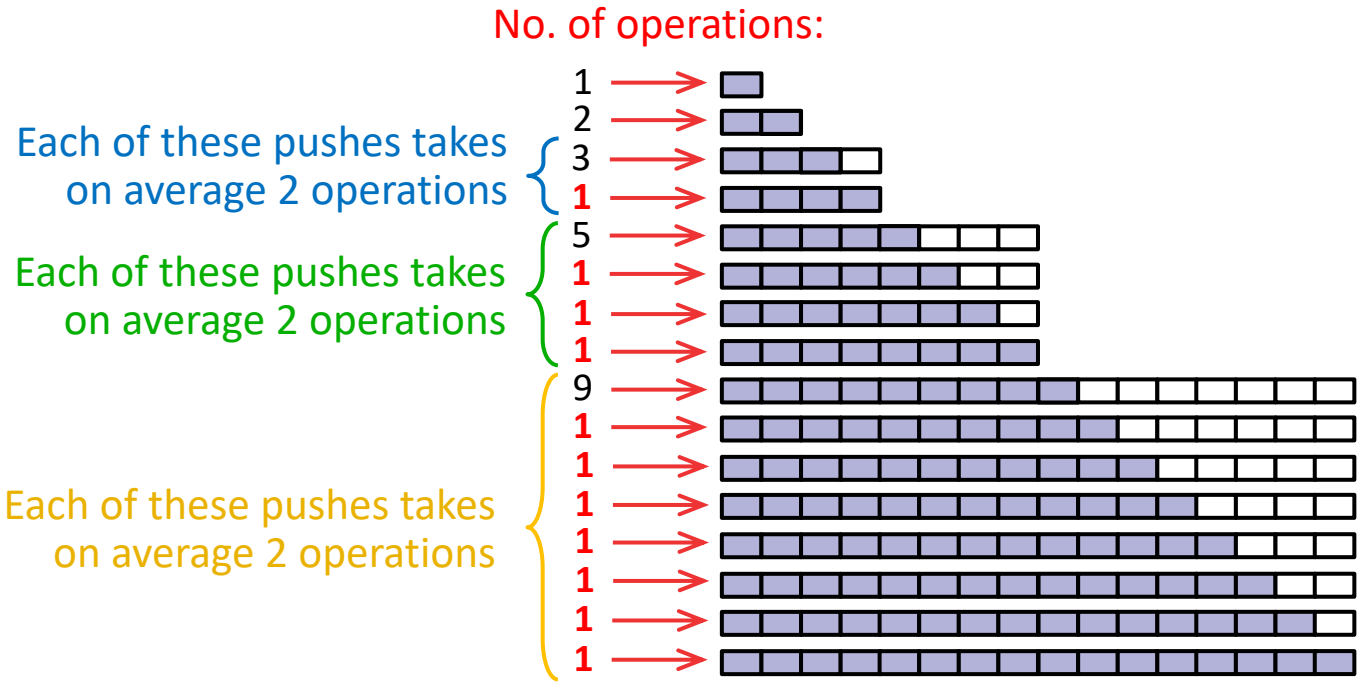



This way of solving problems (where we execute an expensive operation, to make future operations easier) is the main idea behind the "amortization" design pattern.

# AMORTIZATION

The idea behind amortization is to not focus on the worst case when analyzing a given operation, because it could be the case that a single expensive operation will make many subsequent operations much easier.

Based on this, our complexity analysis should **take the average over the expensive operation and all the subsequent, cheap operations**.




Thus, although a push takes O($n$) in the worst case, it only takes O(1) time on average.
Based on this, **the total time to perform a series of $n$ pushes is just O($n$) .**