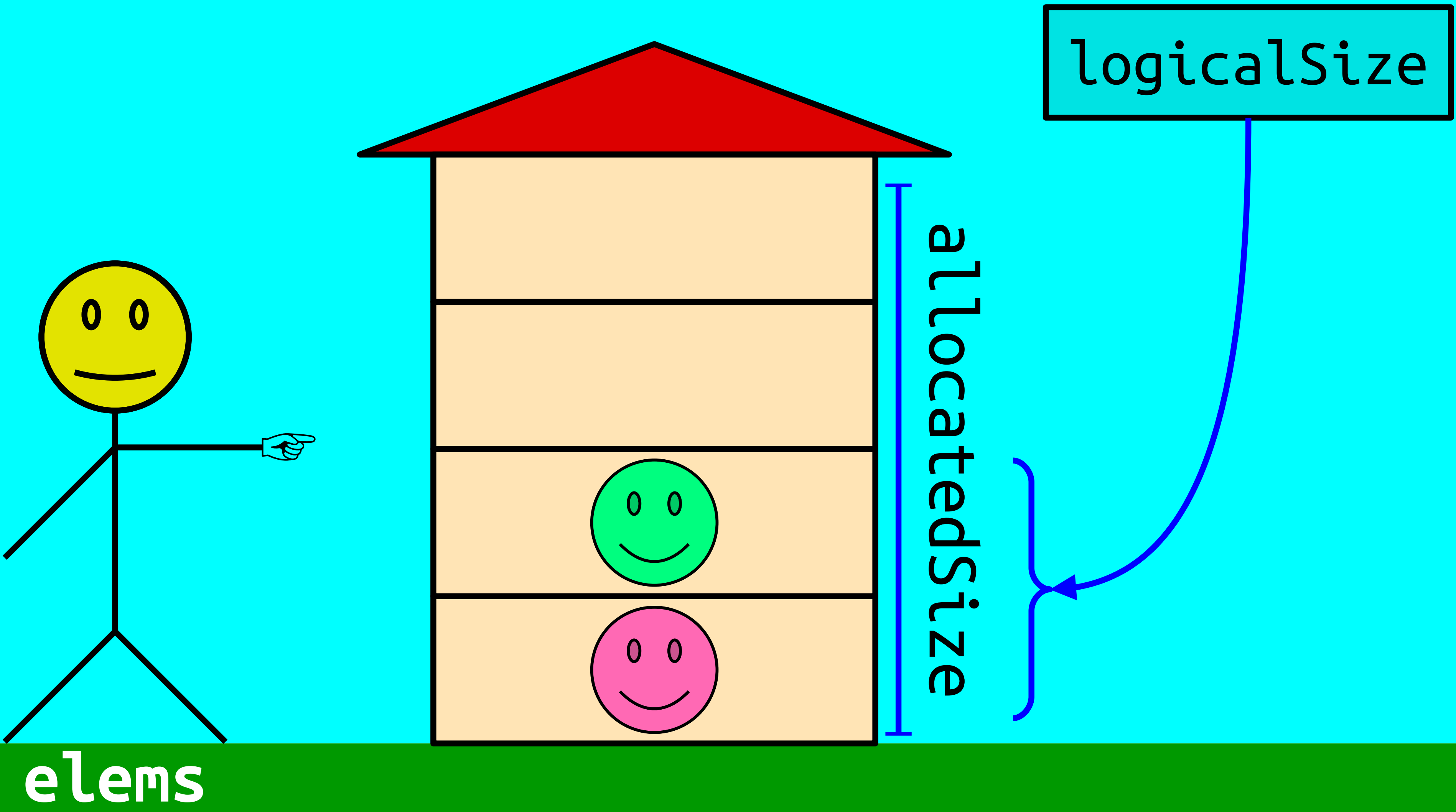
Implementing Abstractions

Part Two

Previously, on CS106B…

```
class OurStack {
public:
    OurStack();

    void push(int value);
    int  peek() const;
    int  pop();

    int  size() const;
    bool isEmpty() const;

private:
    int* elems;
    int  allocatedSize;
    int  logicalSize;
};
```

```
private:
    int* elems;
    int  allocatedSize;
    int  logicalSize;
```



Cleaning Up our Messes

Destructors

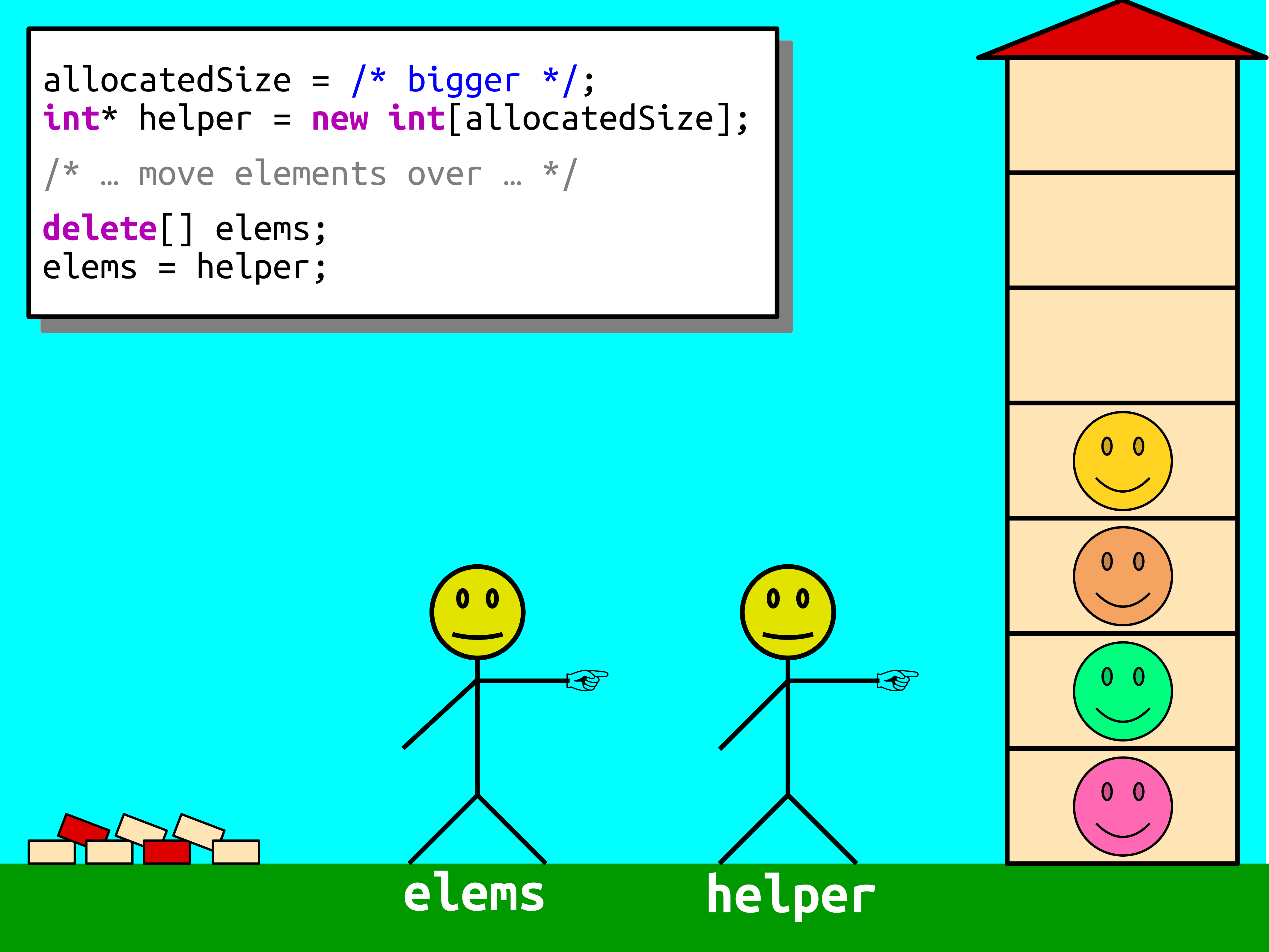
- A ***destructor*** is a special member function responsible for cleaning up an object's memory.
- It’s automatically called whenever an object’s lifetime ends (for example, if it’s a local variable that goes out of scope.)
- The destructor for a class named ***ClassName*** has signature

  ~***ClassName***();

```
class OurStack {
public:
    OurStack();
    ~OurStack();

    void push(int value);
    int  peek() const;
    int  pop();

    int  size() const;
    bool isEmpty() const;

private:
    int* elems;
    int  allocatedSize;
    int  logicalSize;
};
```

Getting More Space

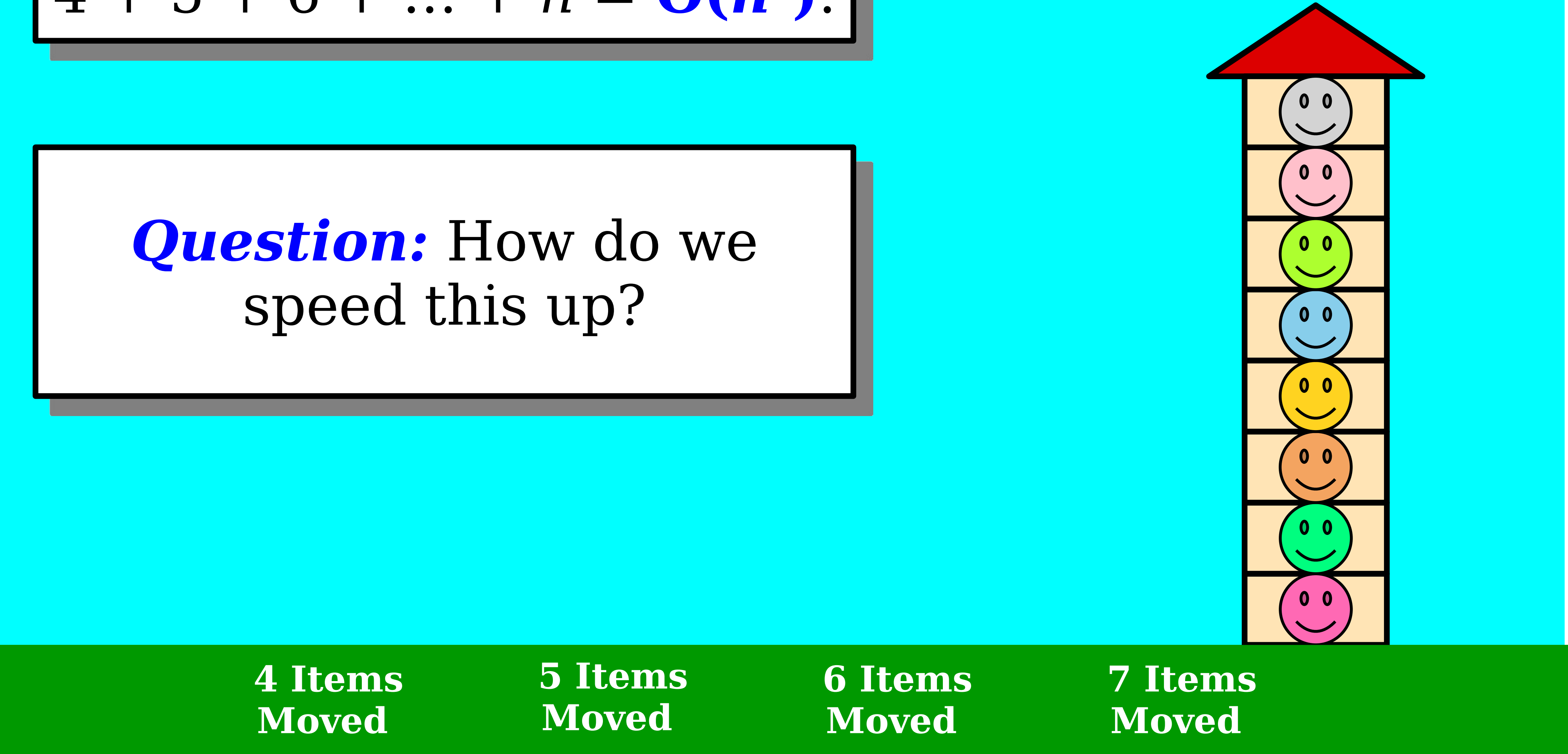
```
allocatedSize = /* bigger */;
int* helper = new int[allocatedSize];

/* … move elements over … */

delete[] elems;
elems = helper;
```

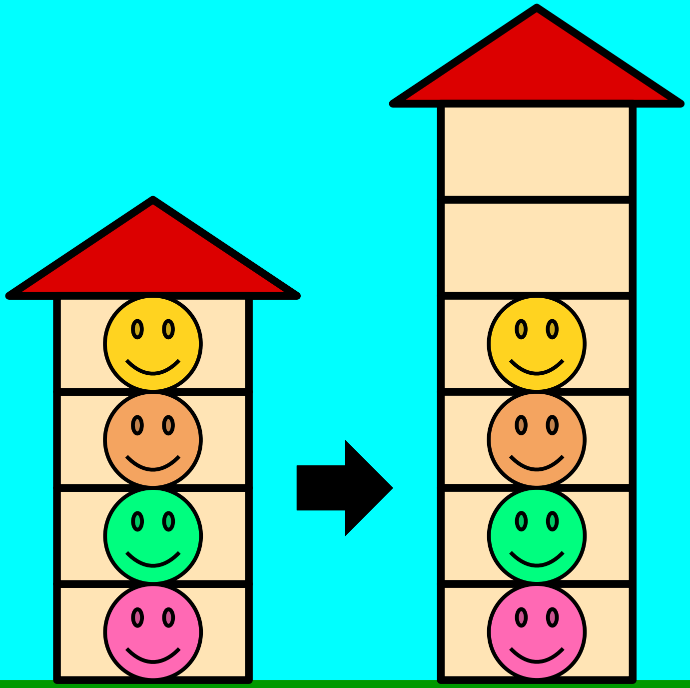


Every push beyond the first few requires moving all n elements from the old array to the new array.

Cost of doing n pushes:
$4 + 5 + 6 + \ldots + n = \mathbf{O(n^2)}$.

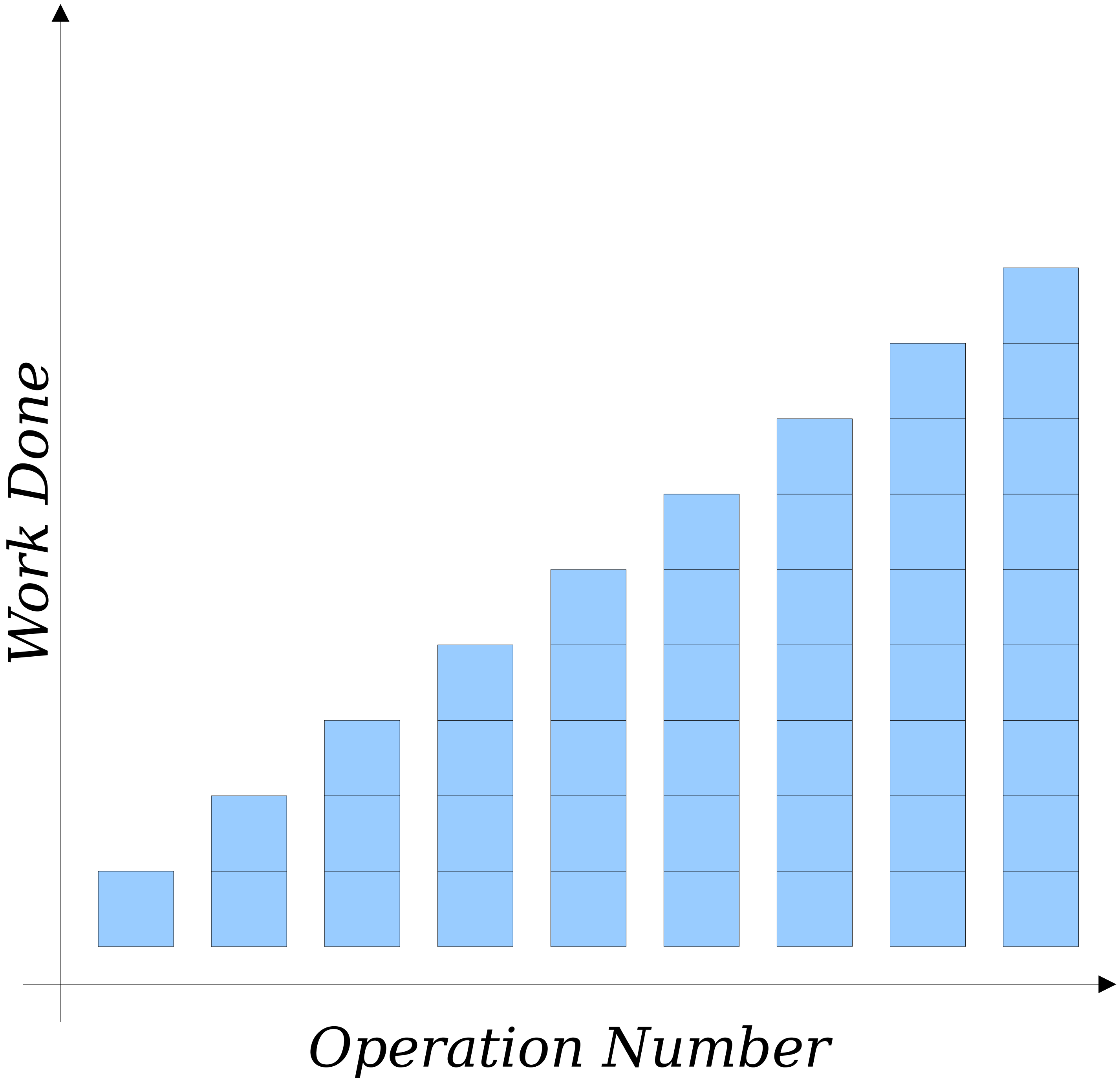
Question: How do we speed this up?





Now, only half the pushes we do will require moving everything to a new array.

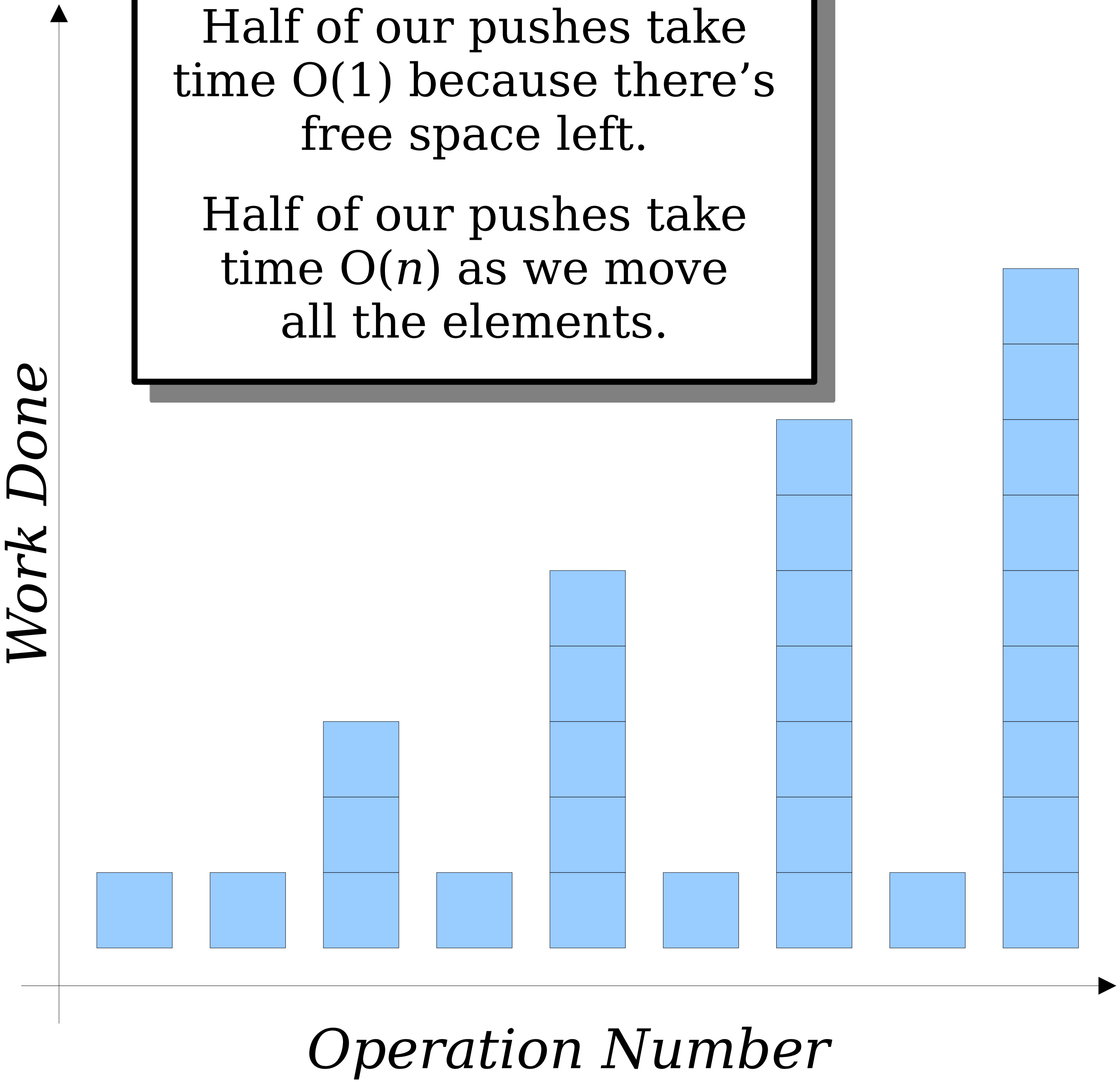
Increase array size by ***adding one***.

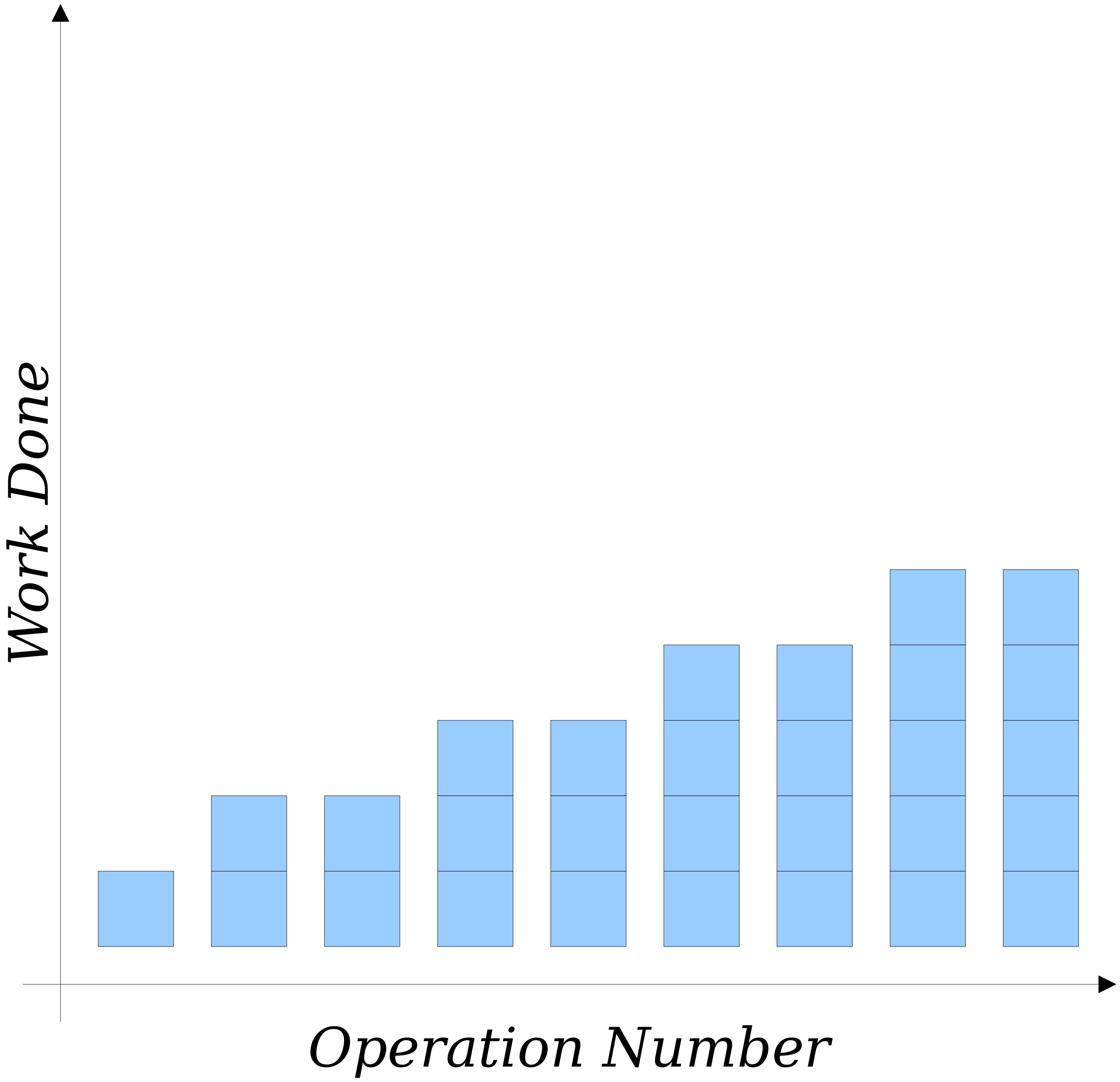


Half of our pushes take time O(1) because there’s free space left.

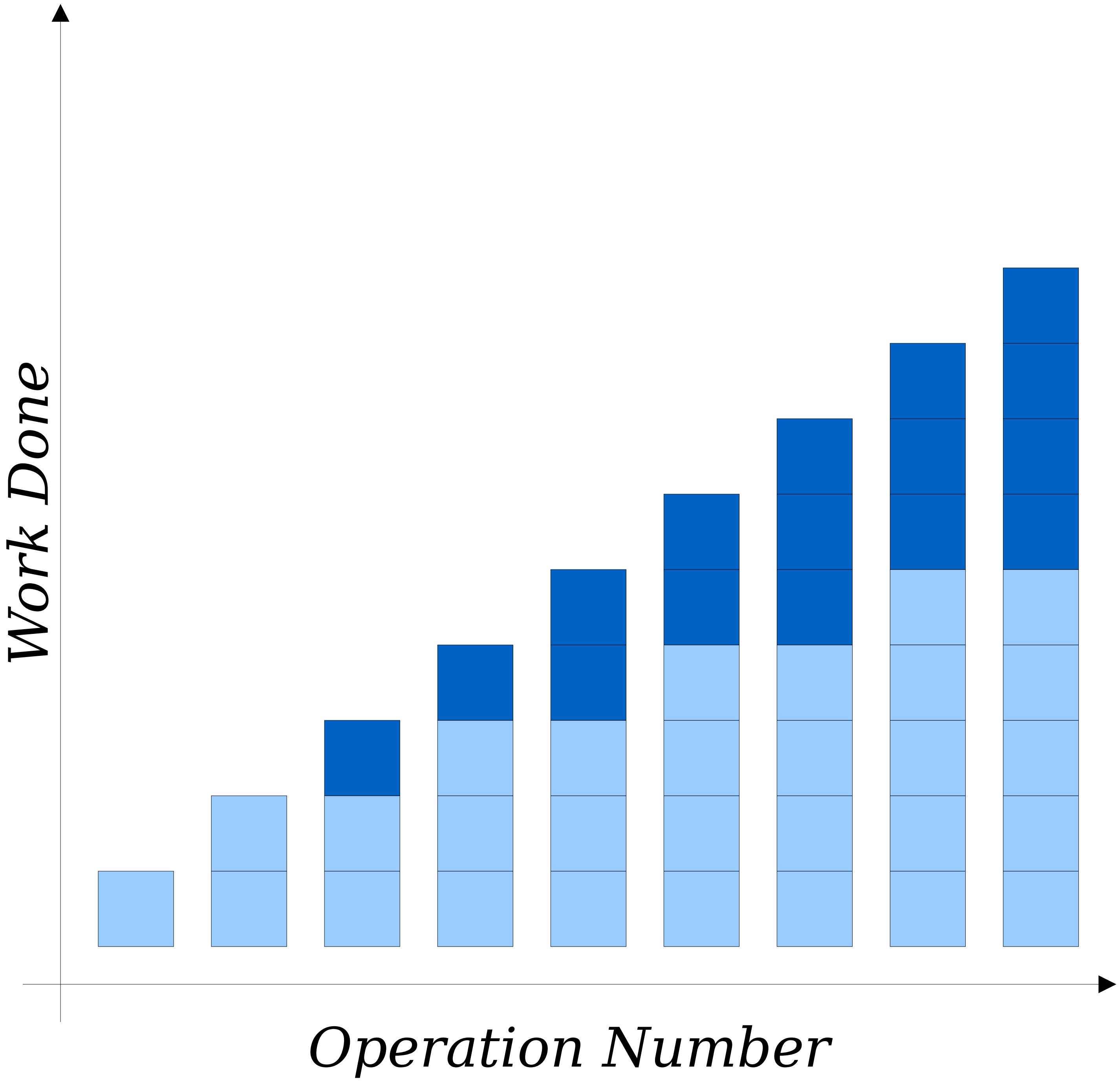
Half of our pushes take time O(*n*) as we move all the elements.

Increase array size by ***adding two***.



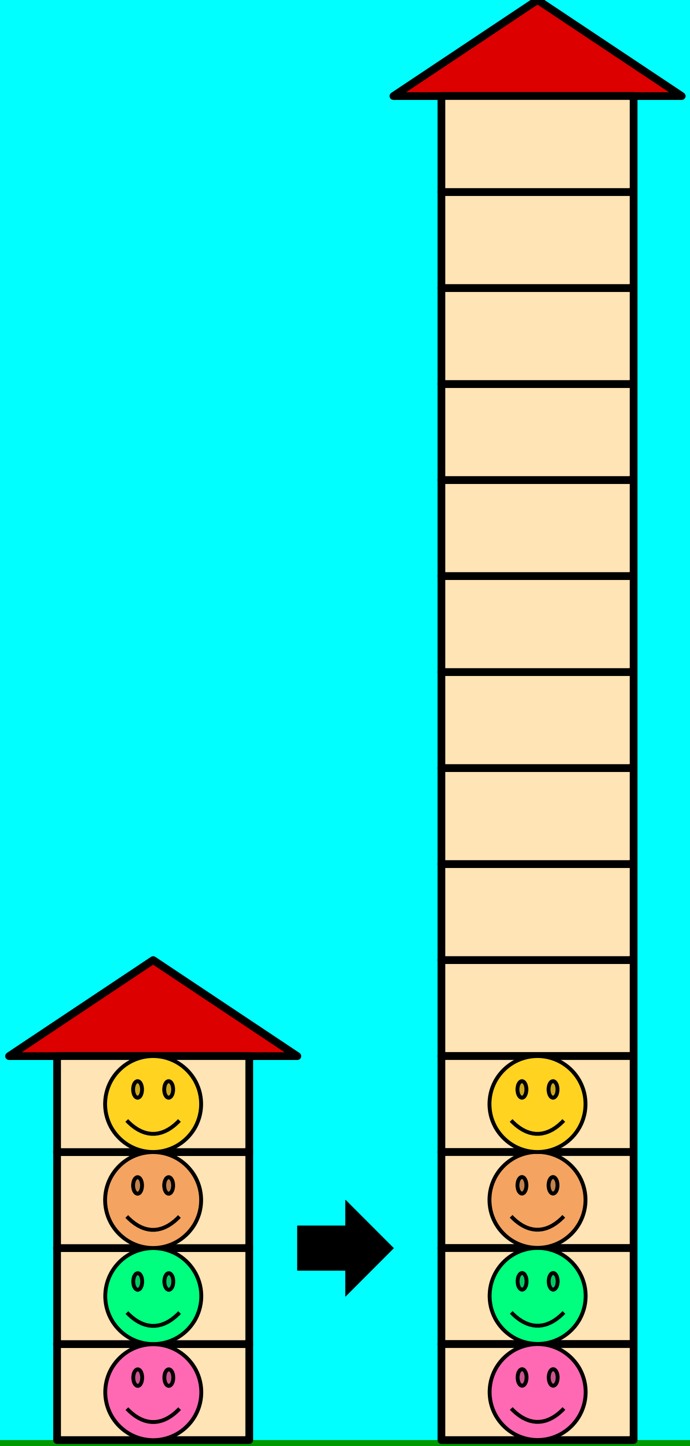


Increase array size by ***adding two***.



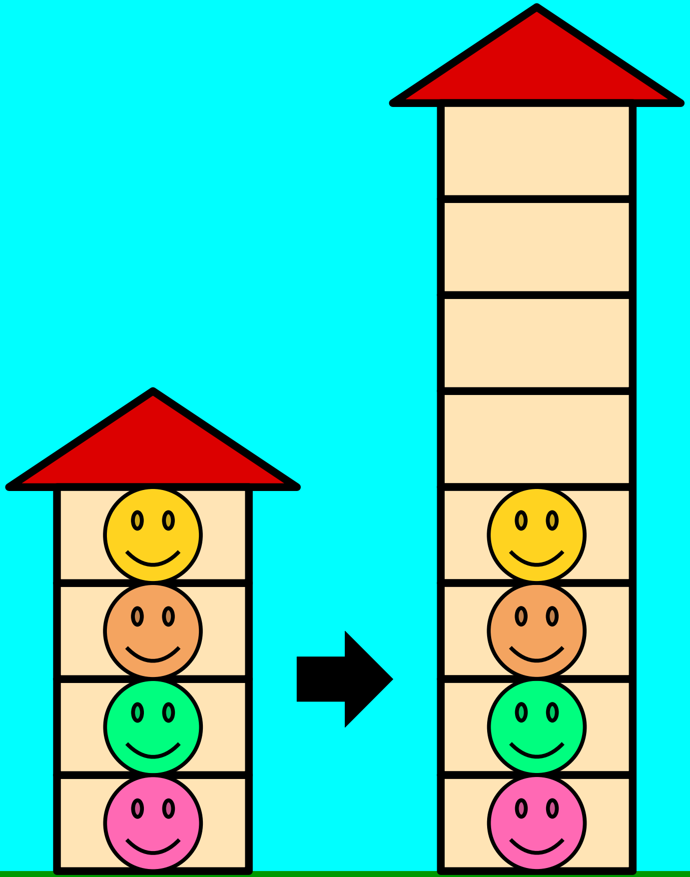
Increase array size by ***adding two***.

This roughly halves the work done.



If we make the new array too big, we’re might not make use of all the new space.

What’s a good compromise?

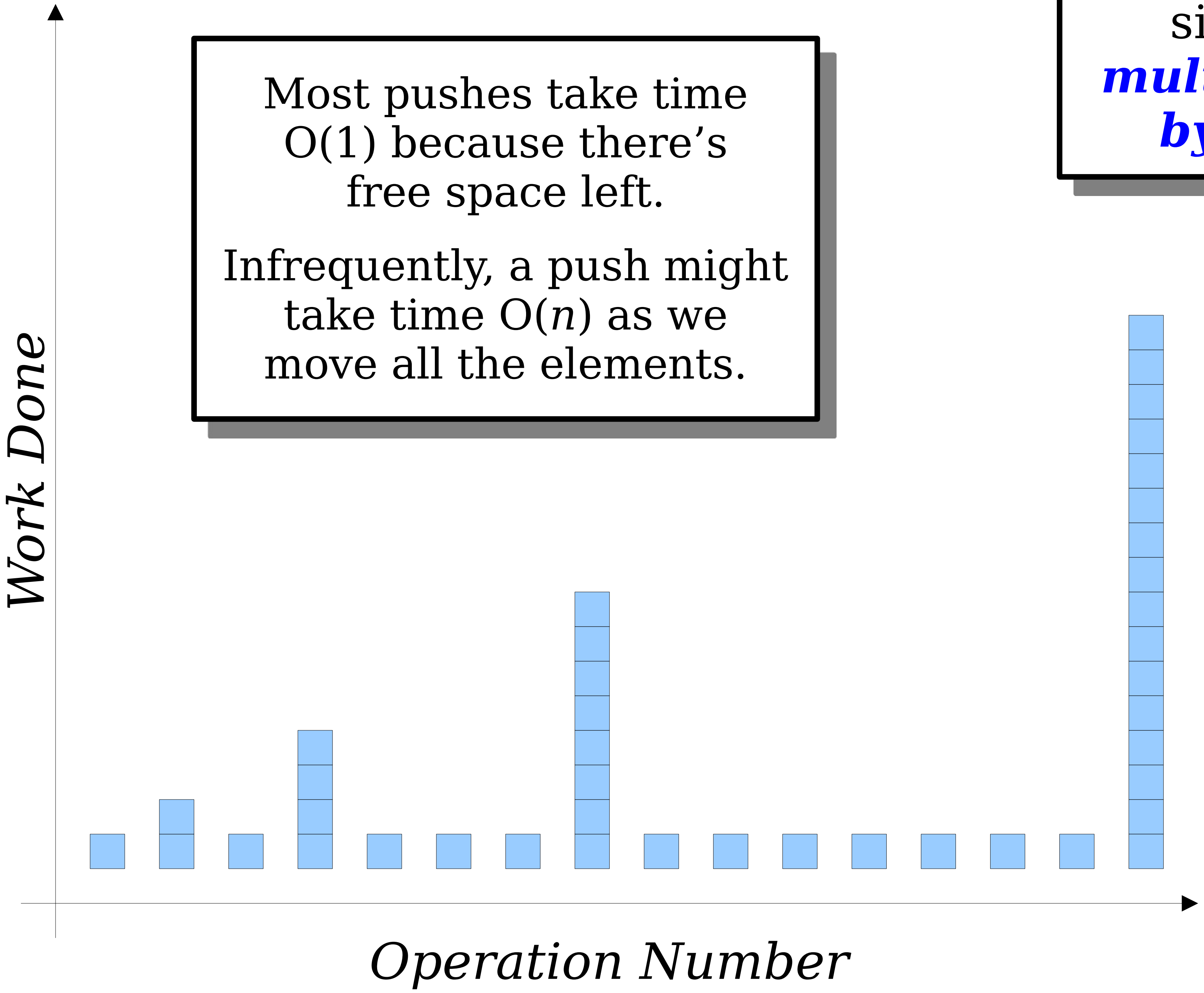


Idea: Make the new array twice as big as the old one.

This gives us a lot of free space, and we never use more than twice the space we need.

How do we analyze this?

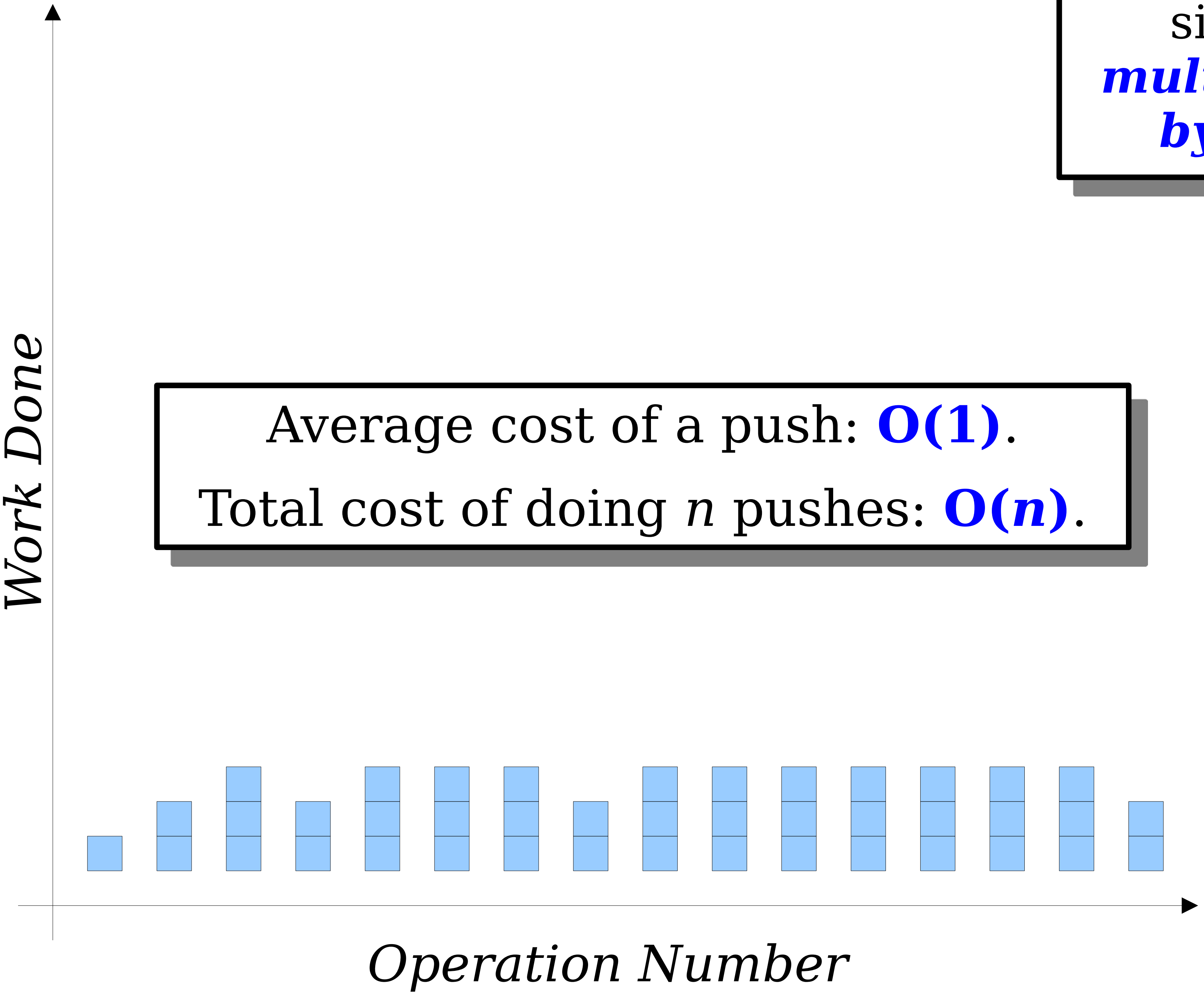
Increase array size by ***multiplying by two***.



Most pushes take time O(1) because there's free space left.

Infrequently, a push might take time O(n) as we move all the elements.

Increase array size by ***multiplying by two***.



Average cost of a push: **O(1)**.

Total cost of doing *n* pushes: **O(*n*)**.

Amortized Analysis

- The analysis we have just done is called an ***amortized analysis***.
- We reason about the total work done by allowing ourselves to backcharge work to previous operations, then look at the "average" amount of work done per operation.
- In an amortized sense, our implementation of the stack is extremely fast!
- This is one of the most common approaches to implementing `Stack` (and `Vector`, for that matter).

Summary for Today

- We can make our stack grow by creating new arrays any time we run out of space.
- Growing that array by one extra slot or two extra slots uses little memory, but makes pushes expensive (average cost O(n)).
- Doubling the size of the array when we run out of space uses more memory, but makes pushes cheap (amortized cost O(1)).
- In practice, it’s worth paying this slight space cost for a marked improvement in runtime.

Your Action Items

- ***Read Chapter 11 and Chapter 12.1***
    - There’s a lot of useful information there about dynamic memory allocation and class design.
- ***Start Assignment 5.***
    - Aim to complete Debugging Warmups tonight and String Simulation by Monday at the start of lecture.
    - Ask for help if you need it! That’s what we’re here for.

Next Time

- ***No Class Monday***
- ***Then, When We Get Back…***
  - ***Hash Functions***
    - A magical and wonderful gift from the world of mathematics.
  - ***Hash Tables***
    - How do we implement `Map` and `Set`?