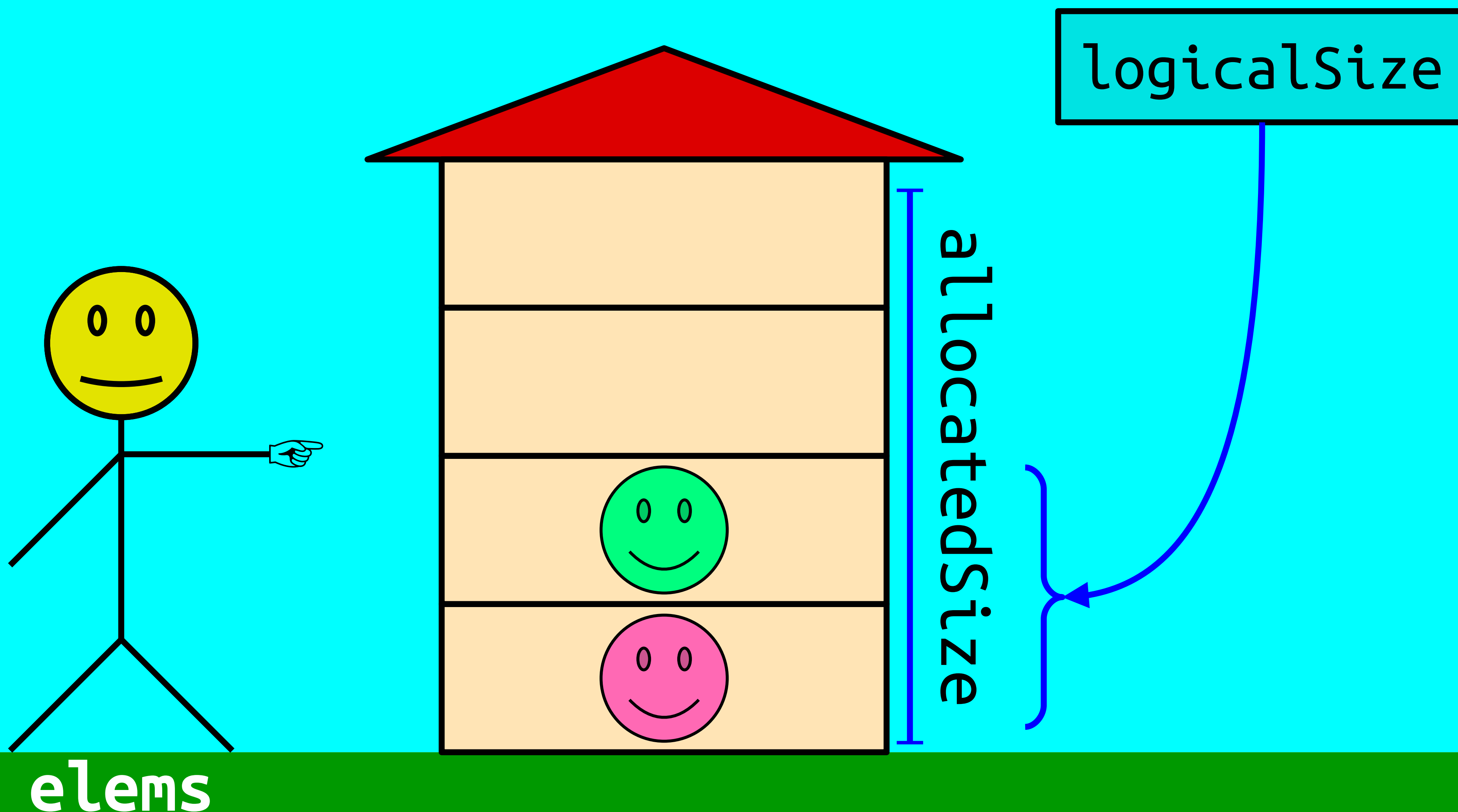
Implementing Abstractions

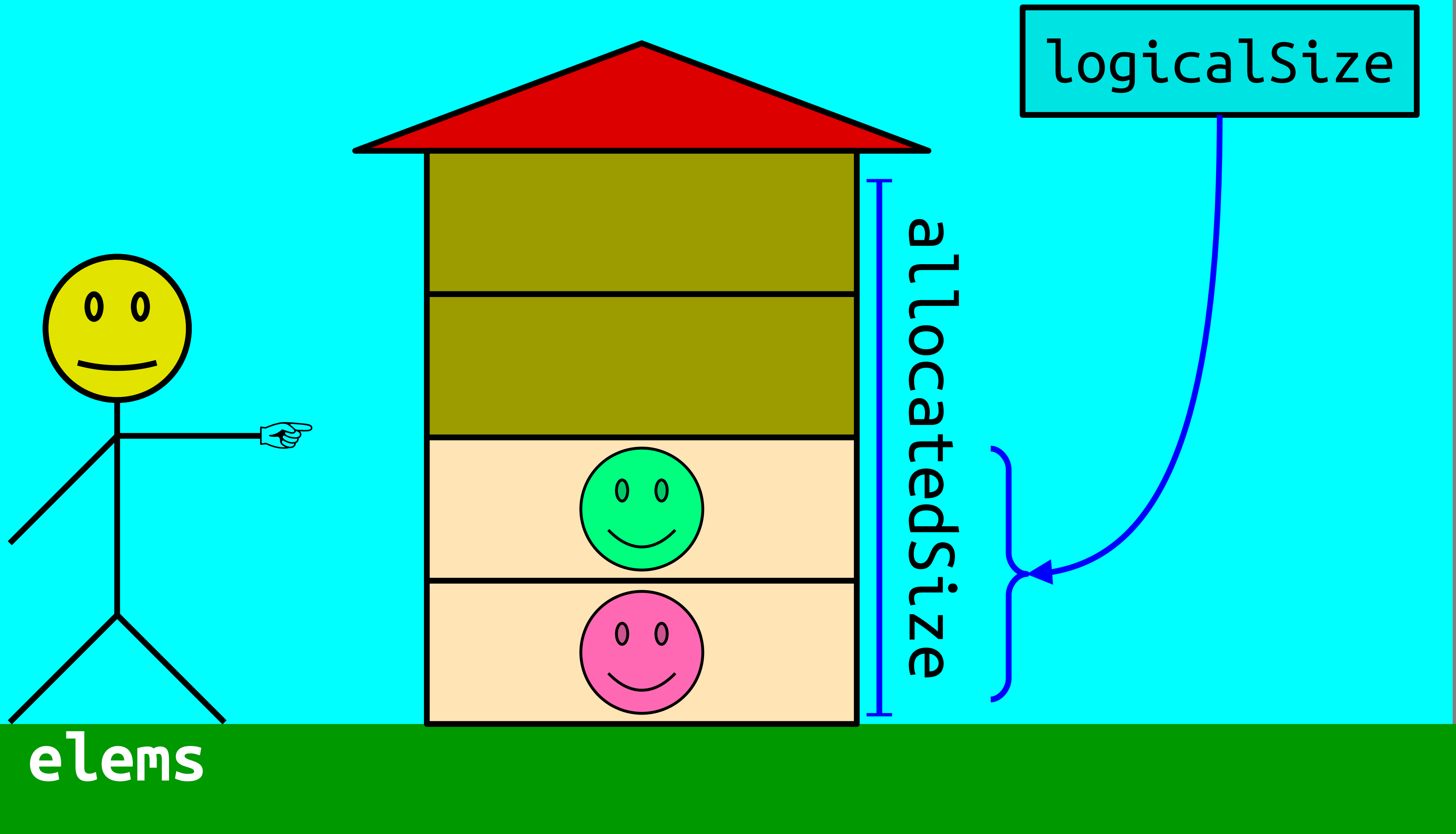
Part Two

Previously, on CS106B...

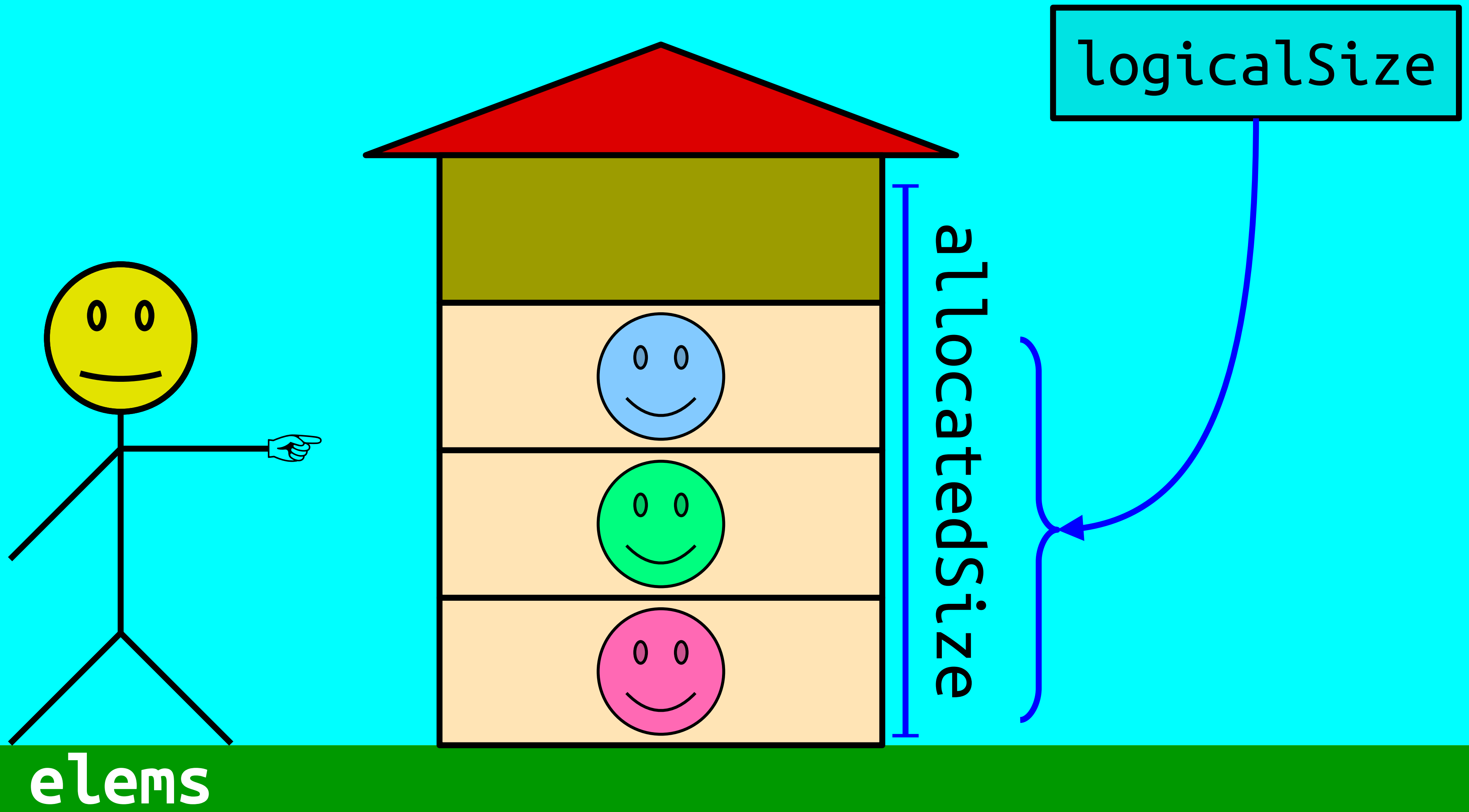
```
private:
    int* elems;
    int  allocatedSize;
    int  logicalSize;
```



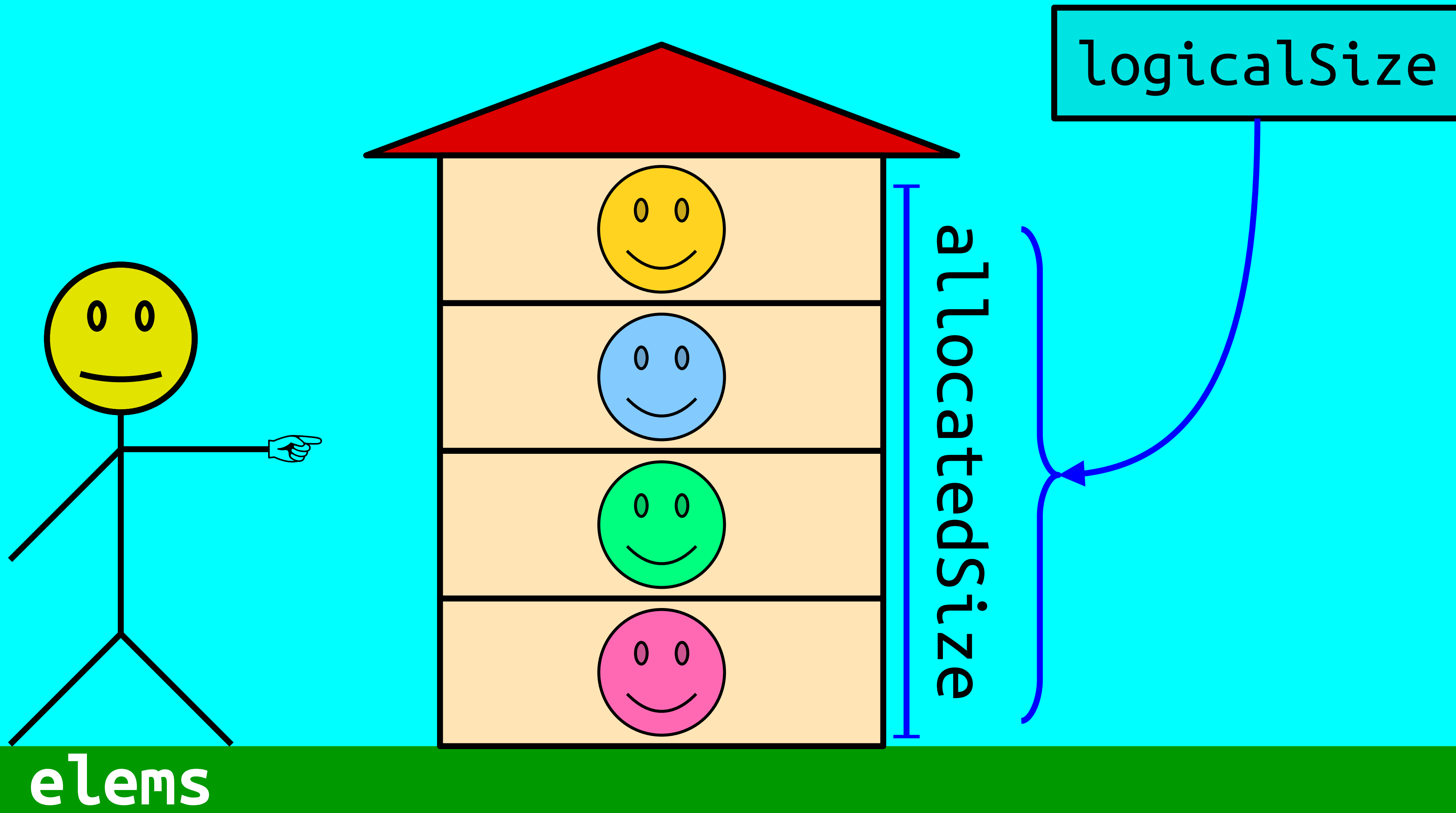
```
private:
    int* elems;
    int  allocatedSize;
    int  logicalSize;
```



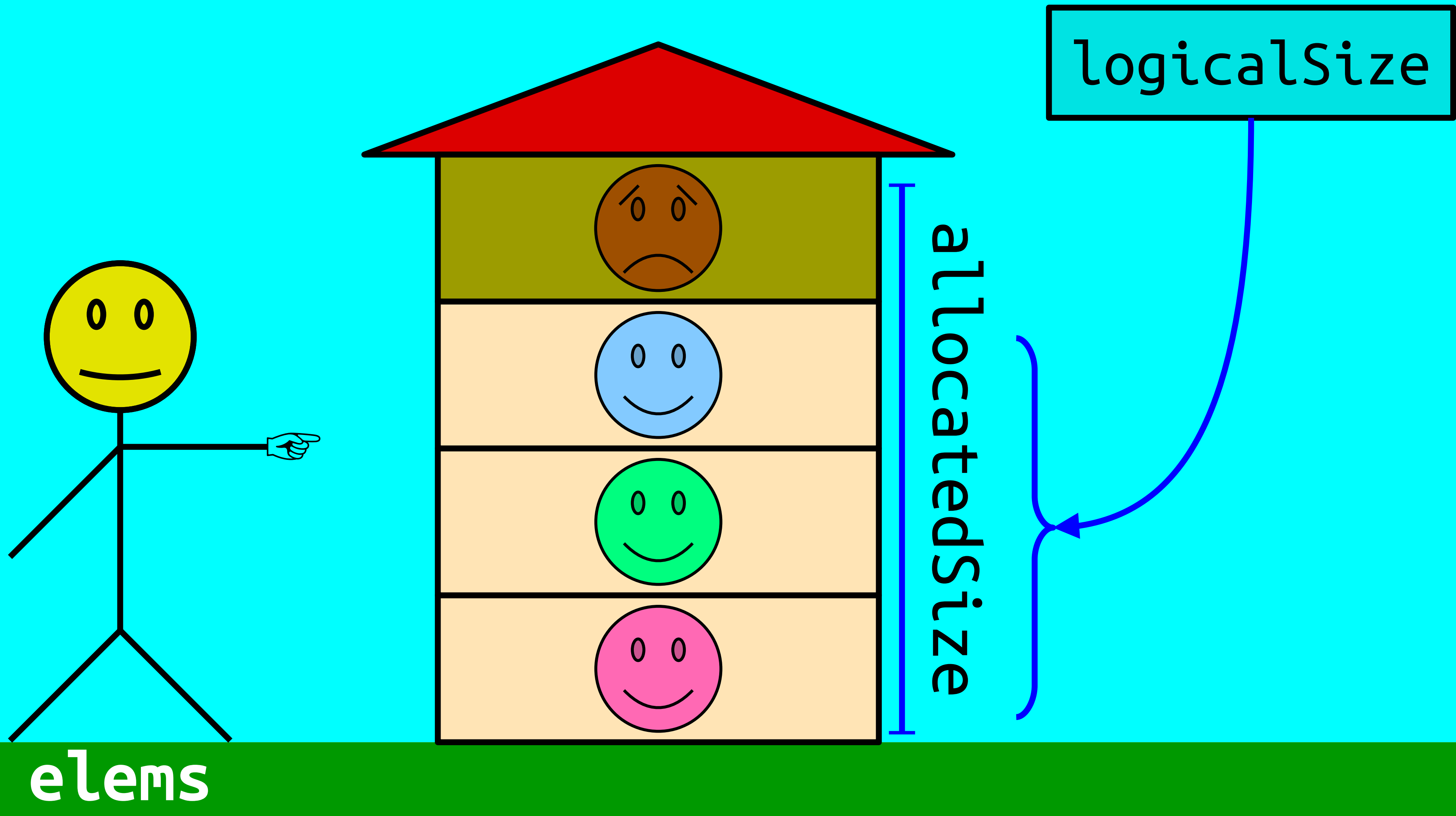
```
private:
    int* elems;
    int  allocatedSize;
    int  logicalSize;
```



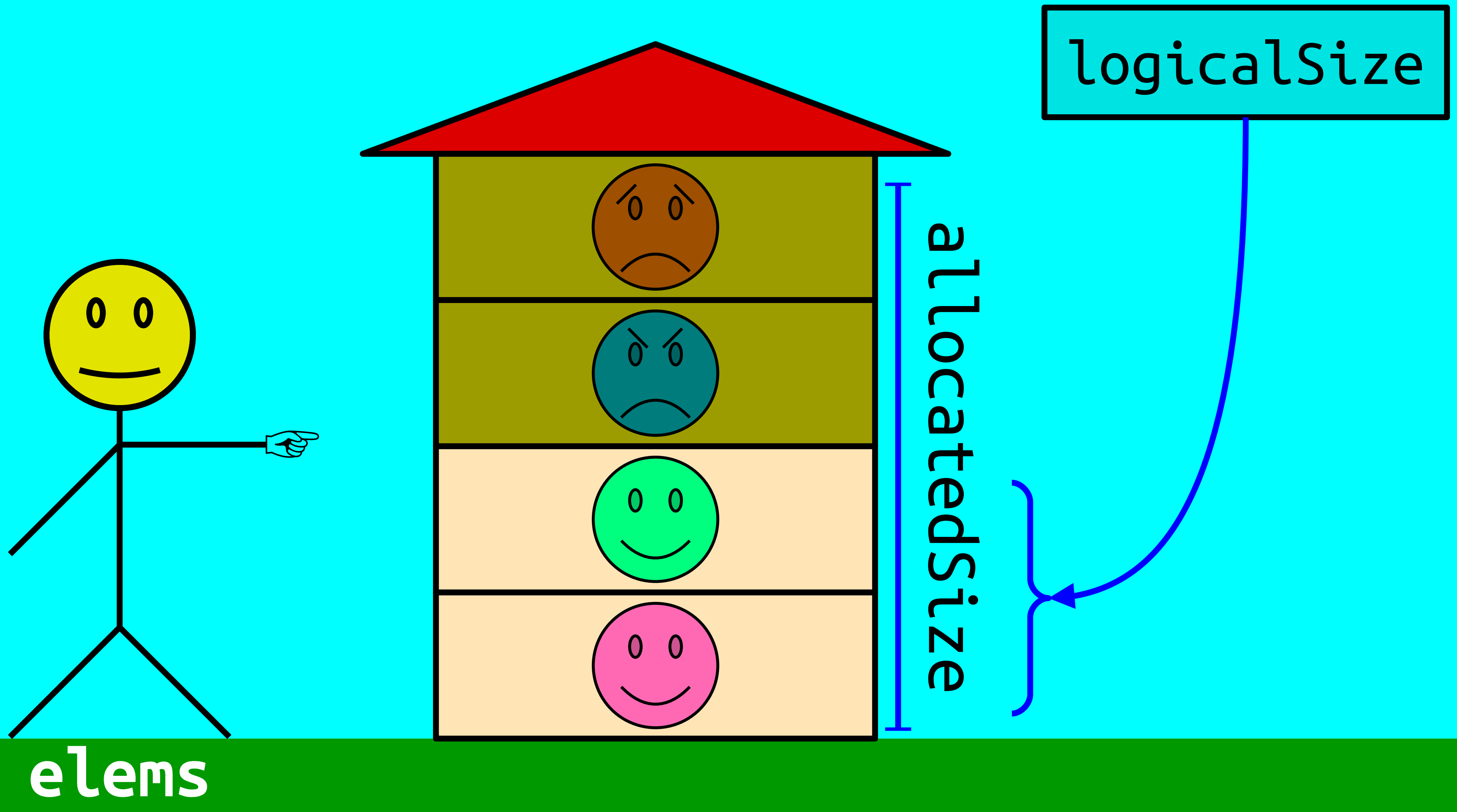
```
private:
    int* elems;
    int  allocatedSize;
    int  logicalSize;
```



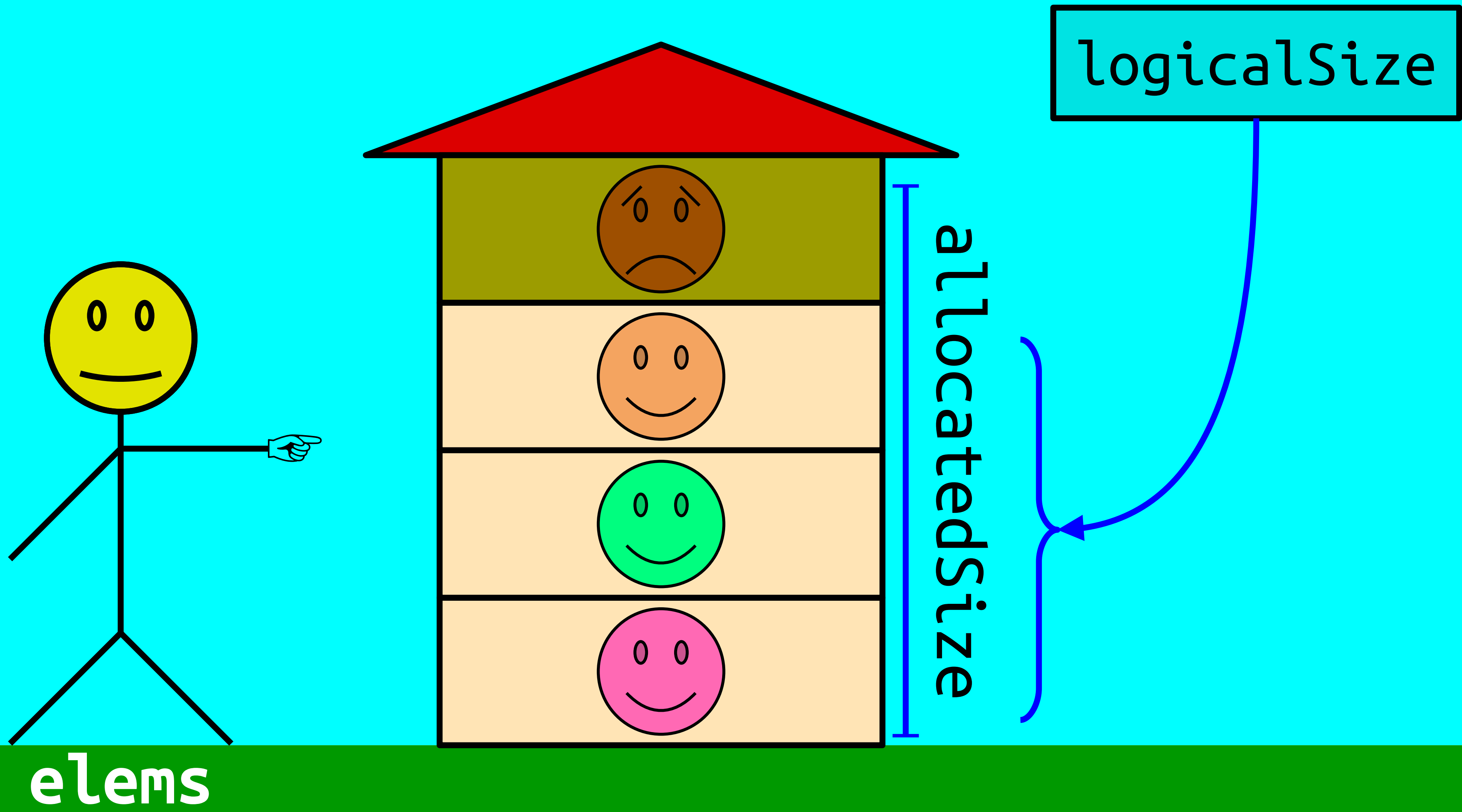
```
private:
    int* elems;
    int  allocatedSize;
    int  logicalSize;
```



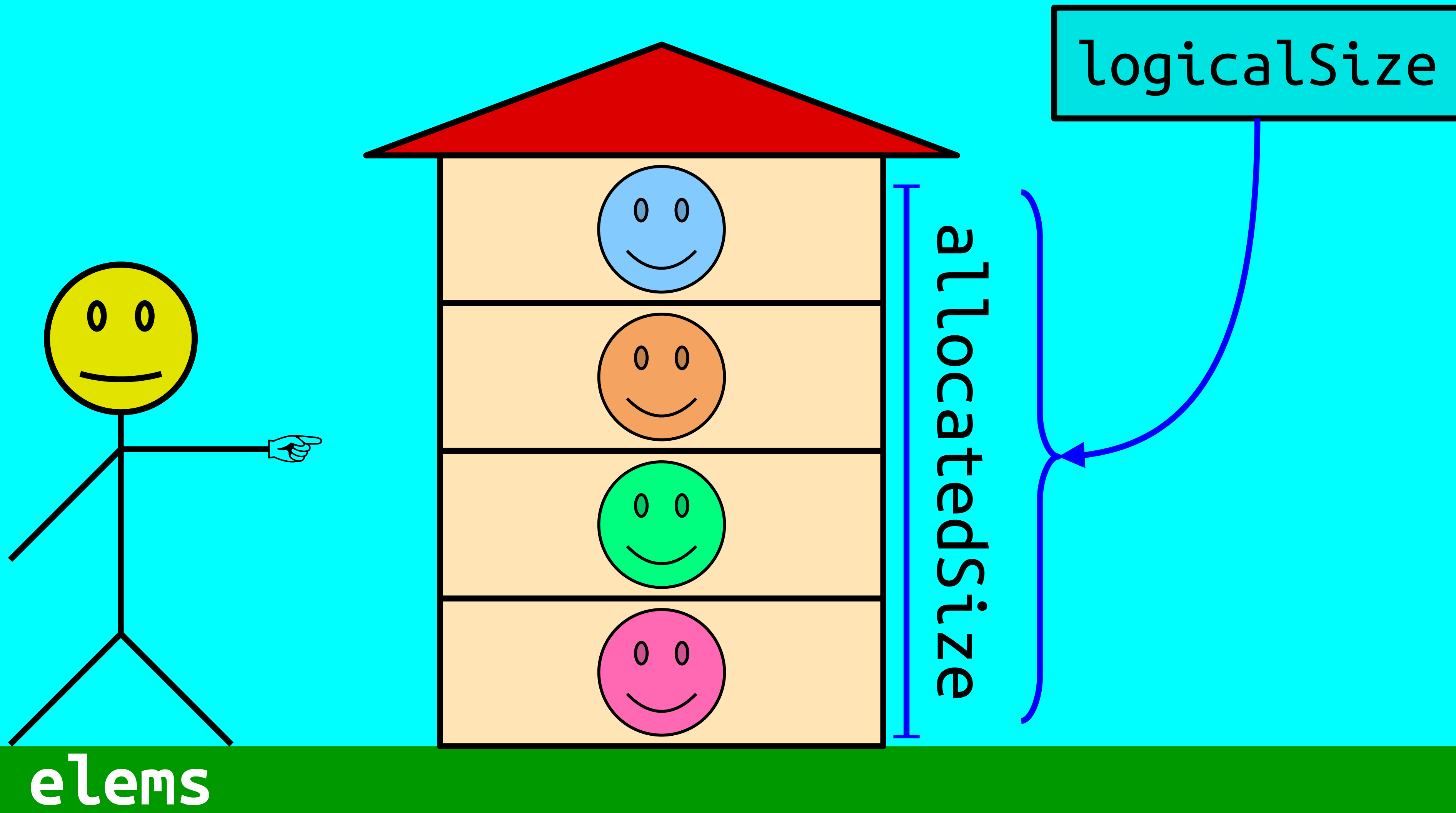
```
private:
    int* elems;
    int  allocatedSize;
    int  logicalSize;
```

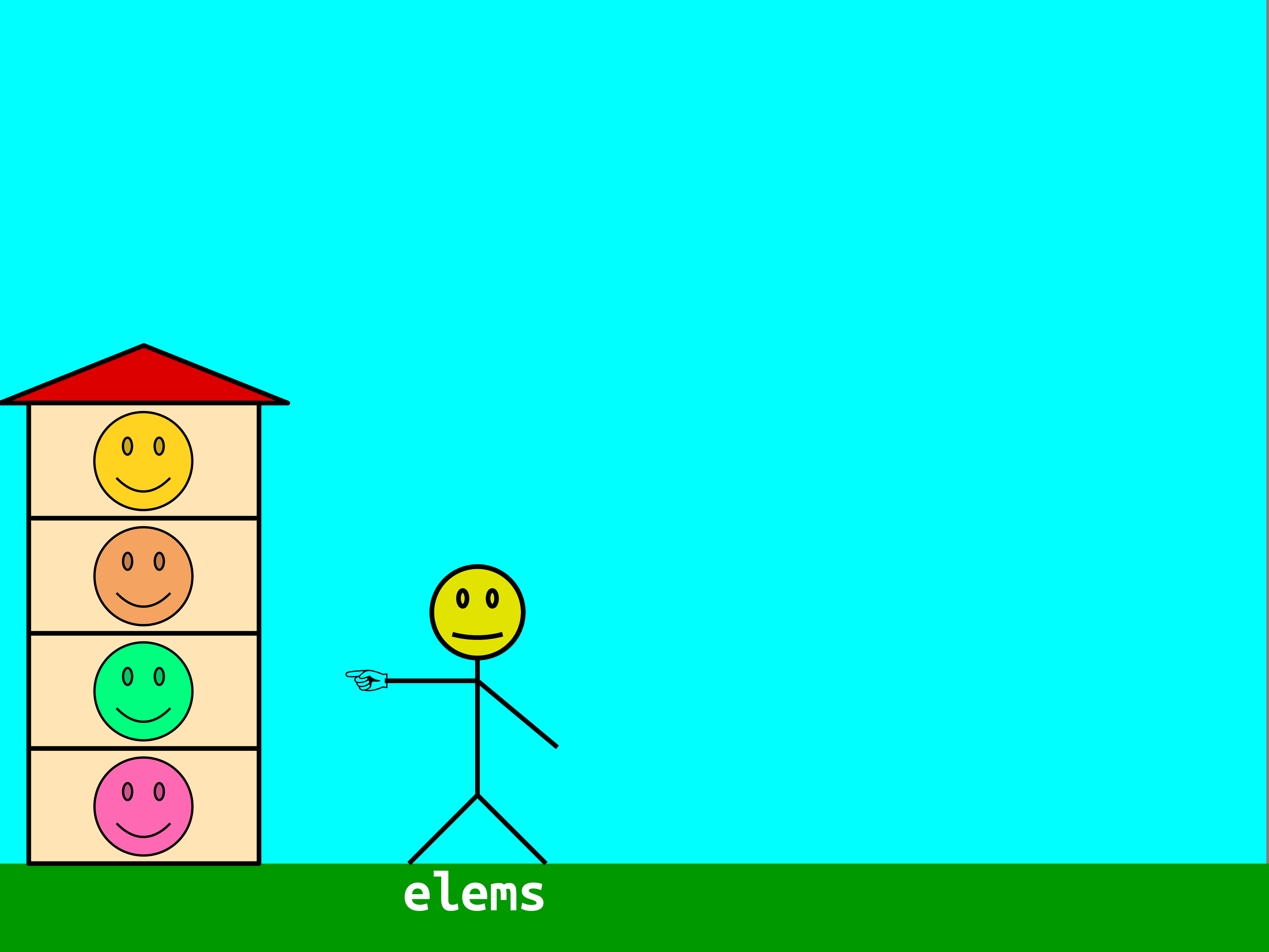


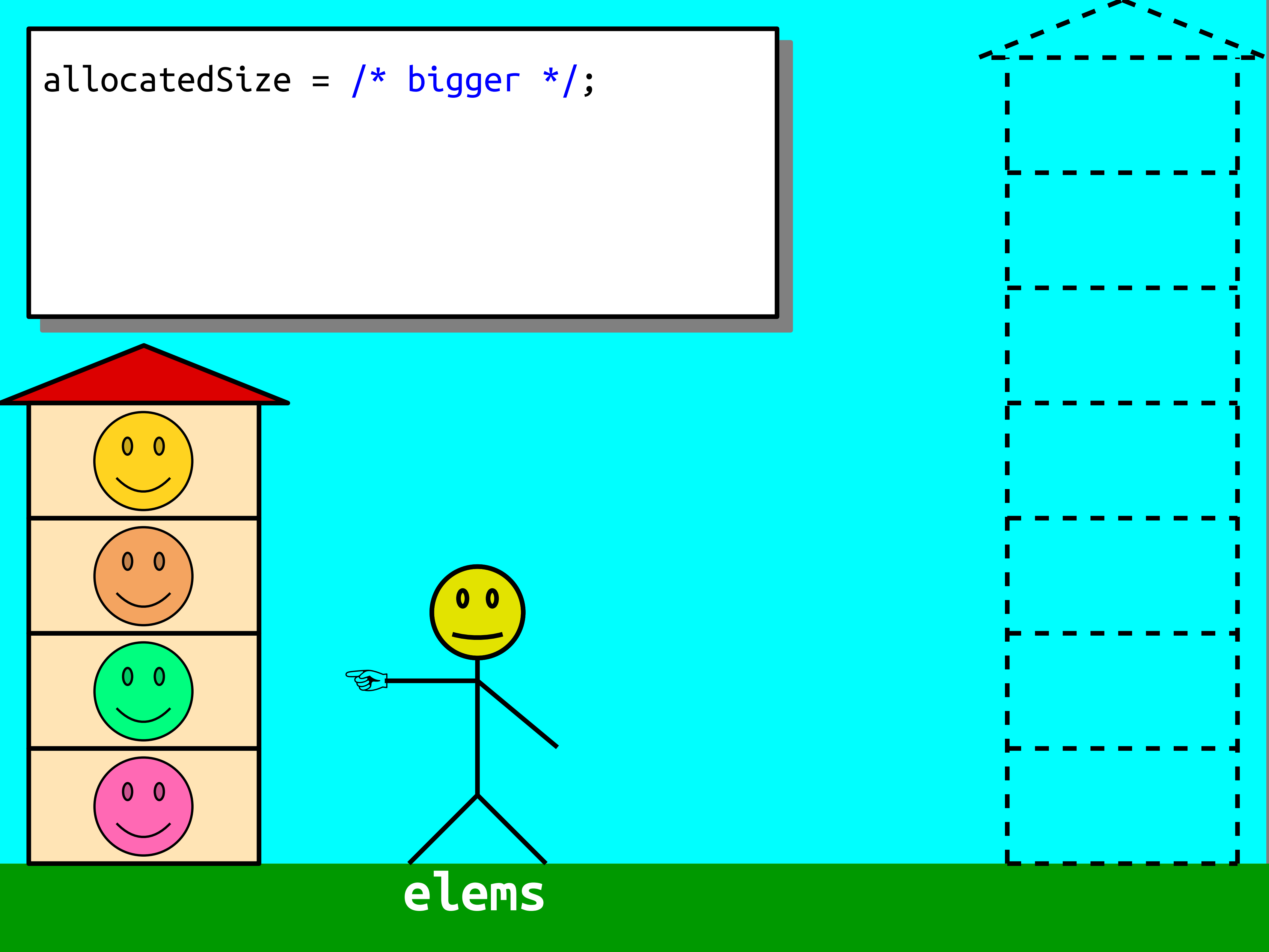
```
private:
    int* elems;
    int  allocatedSize;
    int  logicalSize;
```



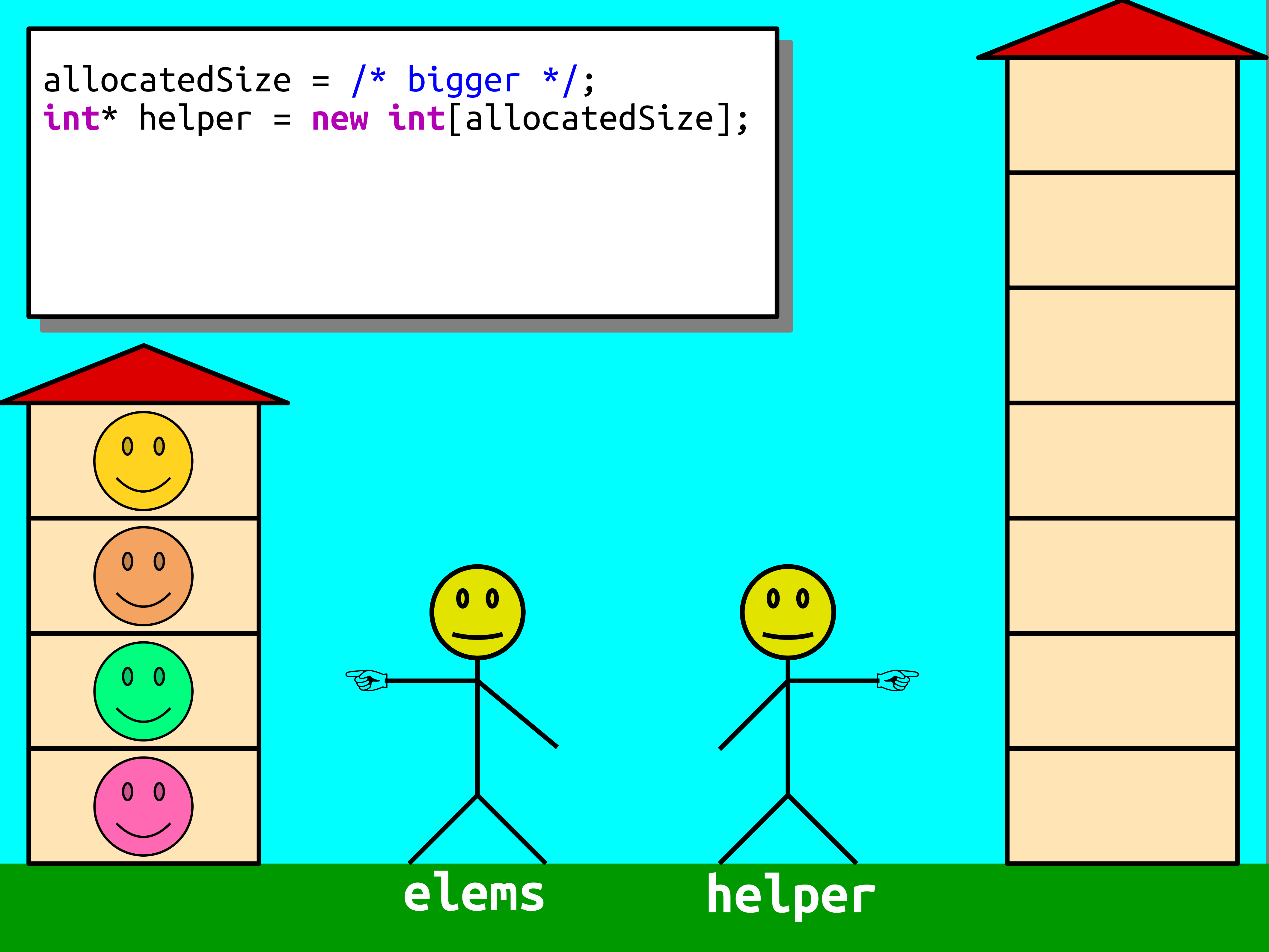
```
private:
    int* elems;
    int  allocatedSize;
    int  logicalSize;
```



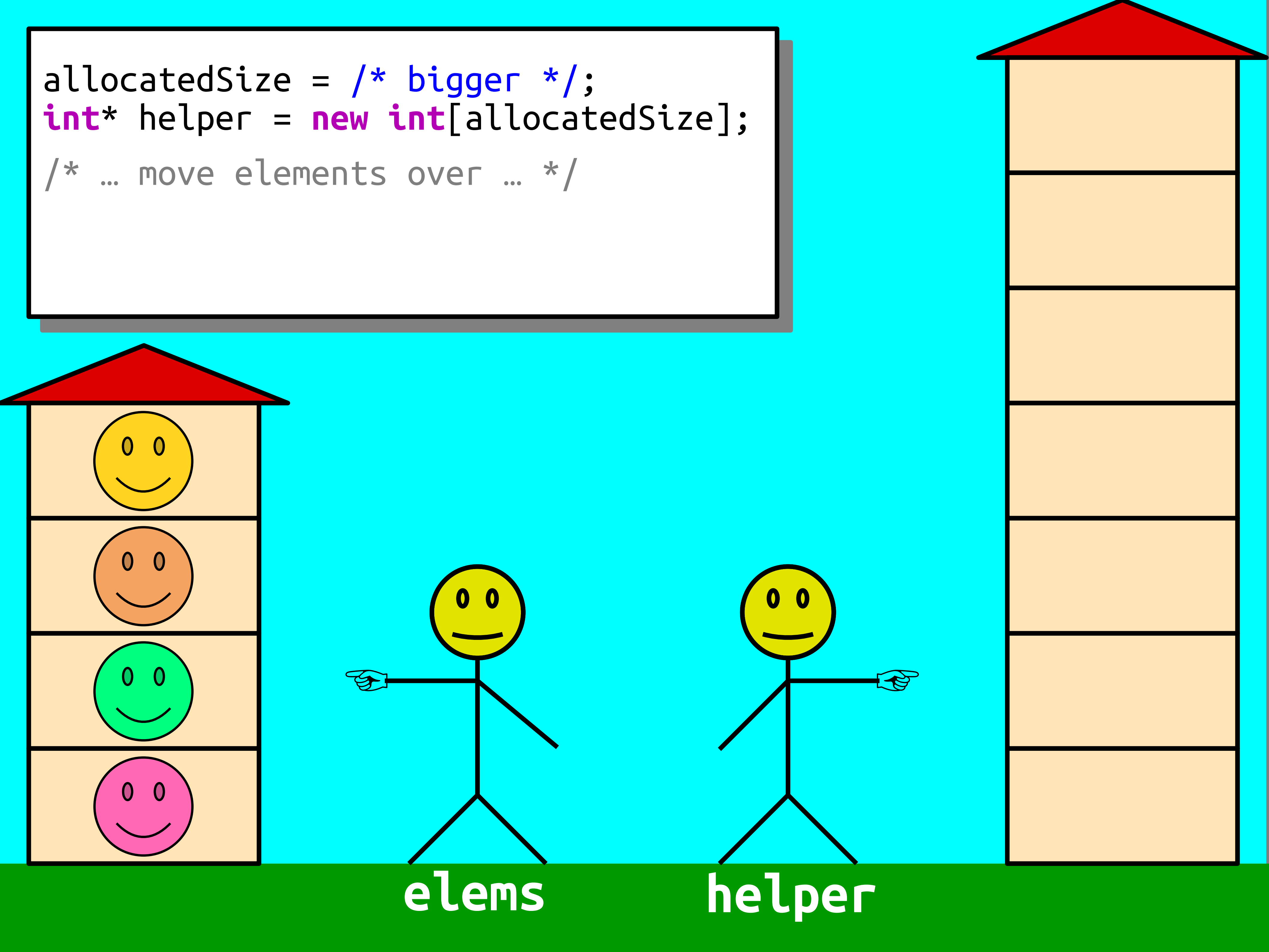

elems


allocatedSize = /* bigger */;
elems

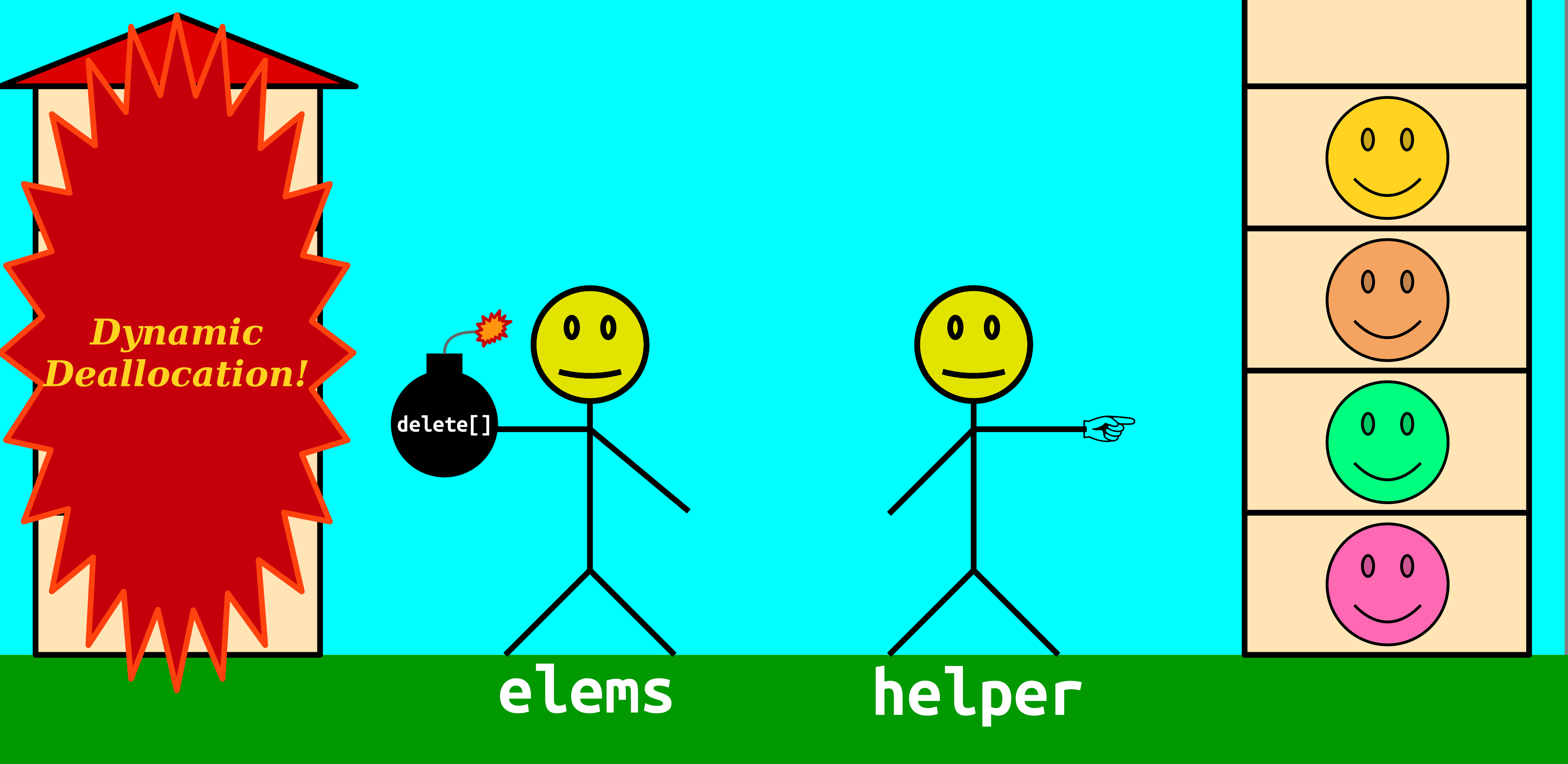
```
allocatedSize = /* bigger */;
int* helper = new int[allocatedSize];
```



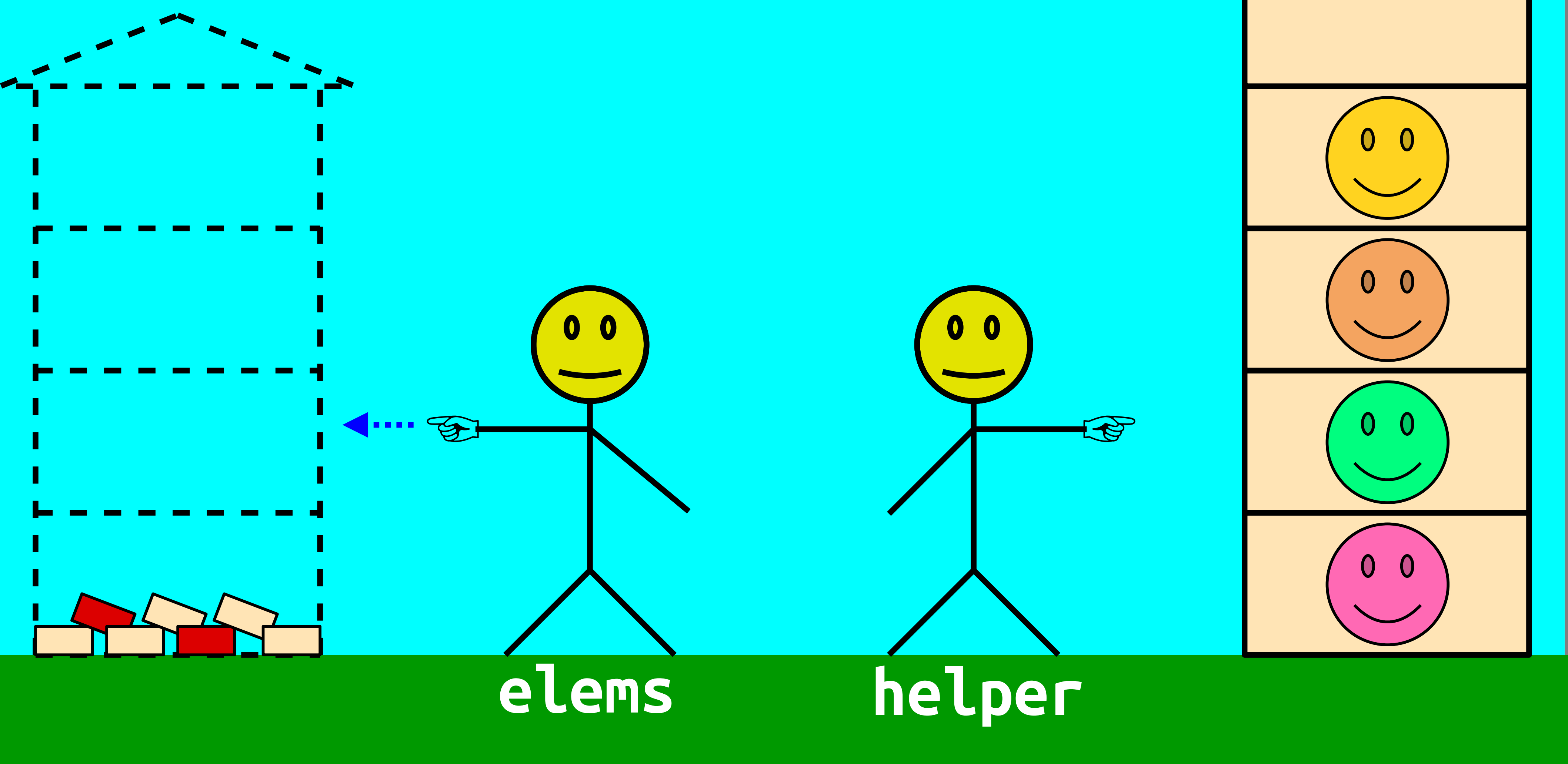
```
allocatedSize = /* bigger */;
int* helper = new int[allocatedSize];

/* … move elements over … */
```



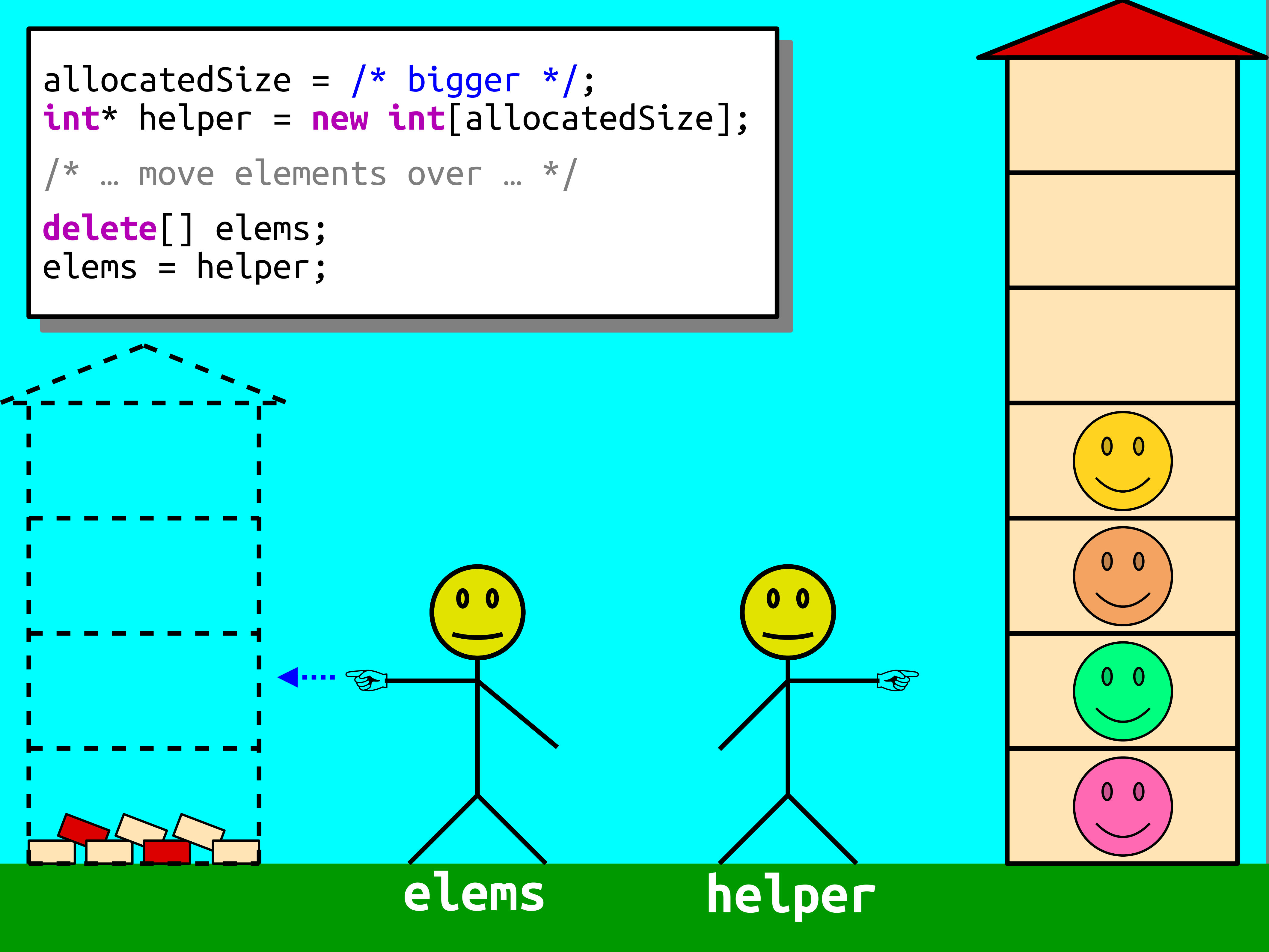
```
allocatedSize = /* bigger */;
int* helper = new int[allocatedSize];

/* … move elements over … */

delete[] elems;
```



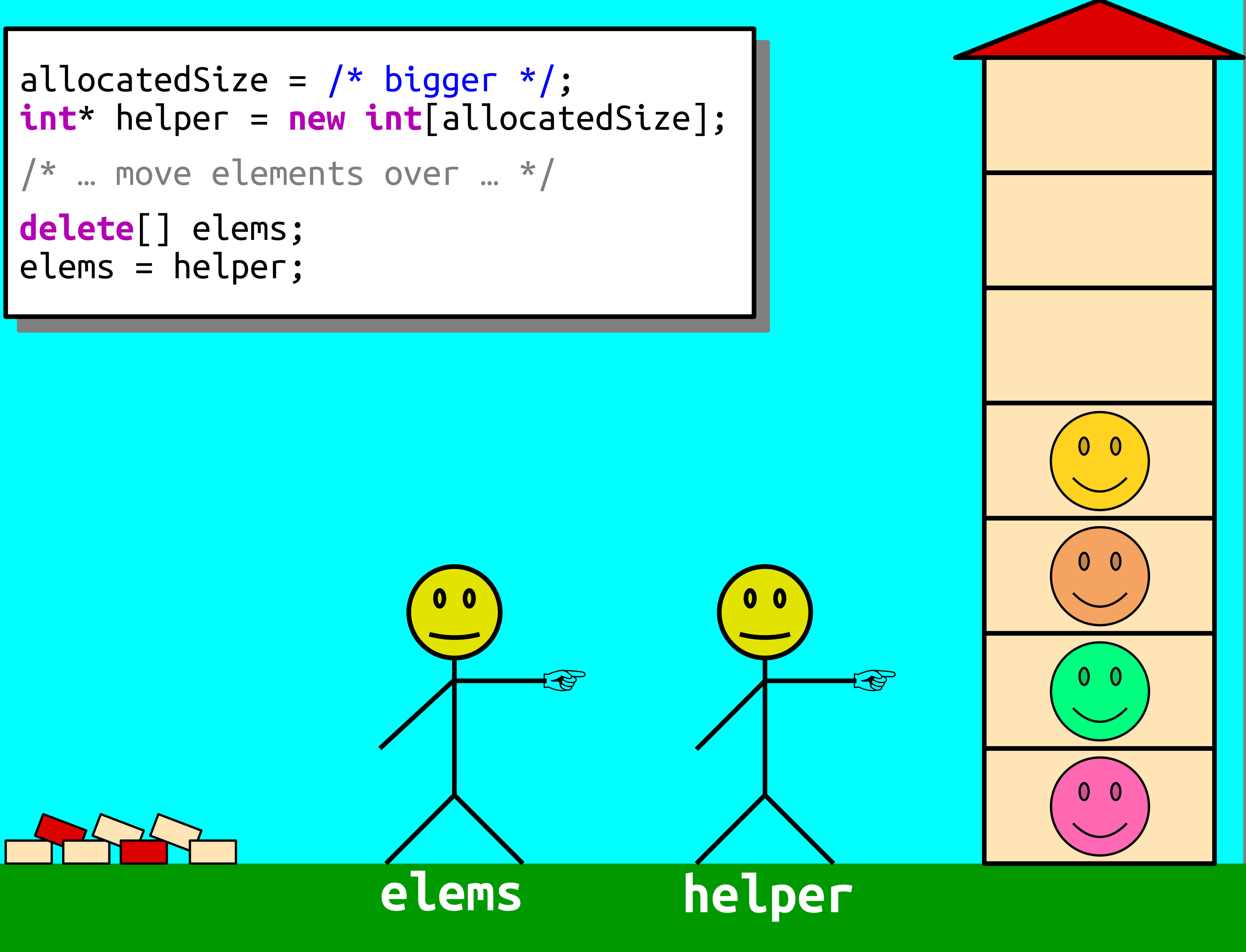
```
allocatedSize = /* bigger */;
int* helper = new int[allocatedSize];

/* … move elements over … */

delete[] elems;
```



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

/* … move elements over … */

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



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

/* … move elements over … */

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



What is the big-O cost of a push?
What is the big-O cost of *n* pushes?

Take thirty seconds to think this over. Look over the runtime plot and the code for the push operation.

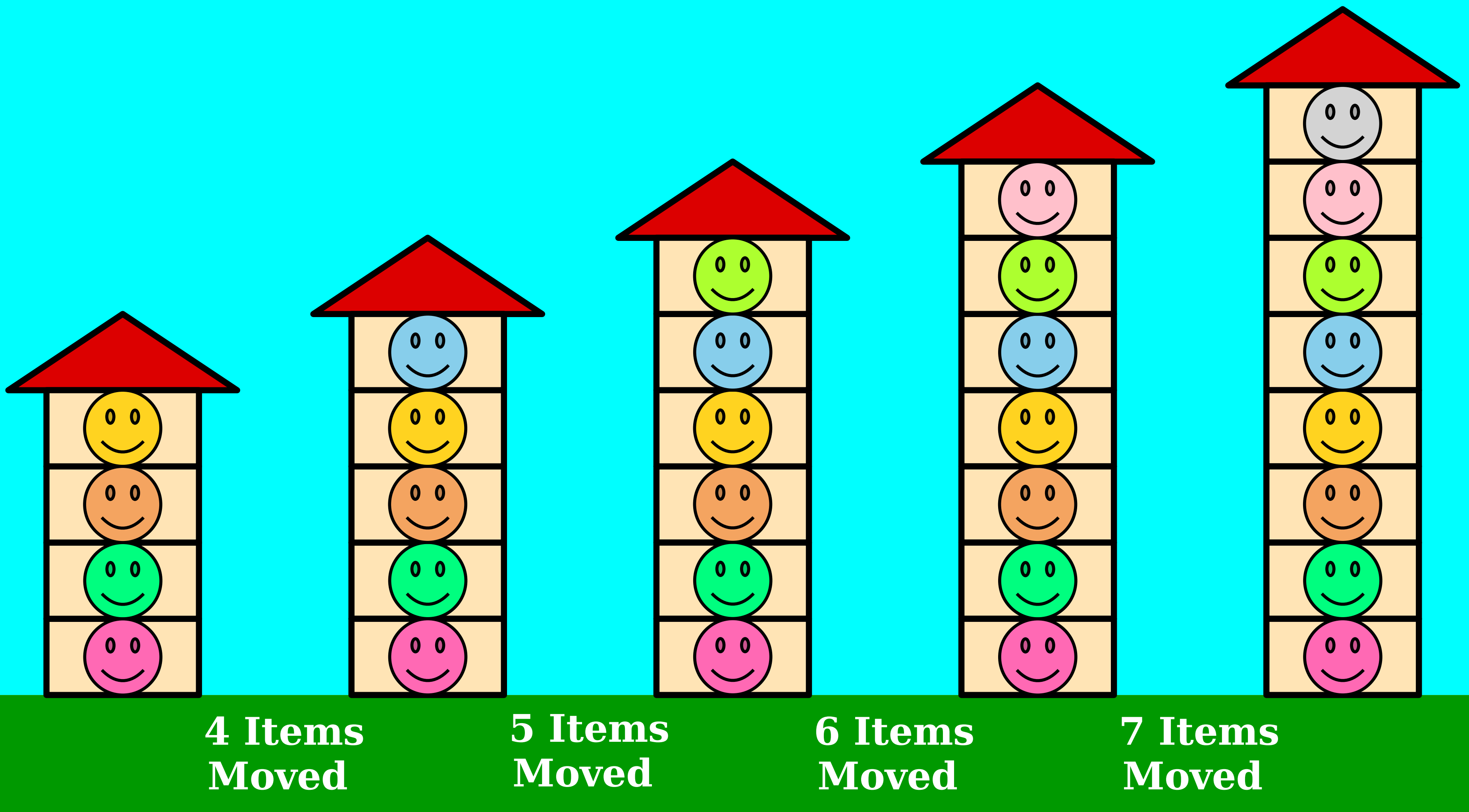
Formulate a hypothesis, but don't post it into chat just yet.

What is the big-O cost of a push?
What is the big-O cost of *n* pushes?

Now, post your hypothesis in chat.

Not sure? Just answer with question marks.

What is the big-O cost of a push?
What is the big-O cost of *n* pushes?



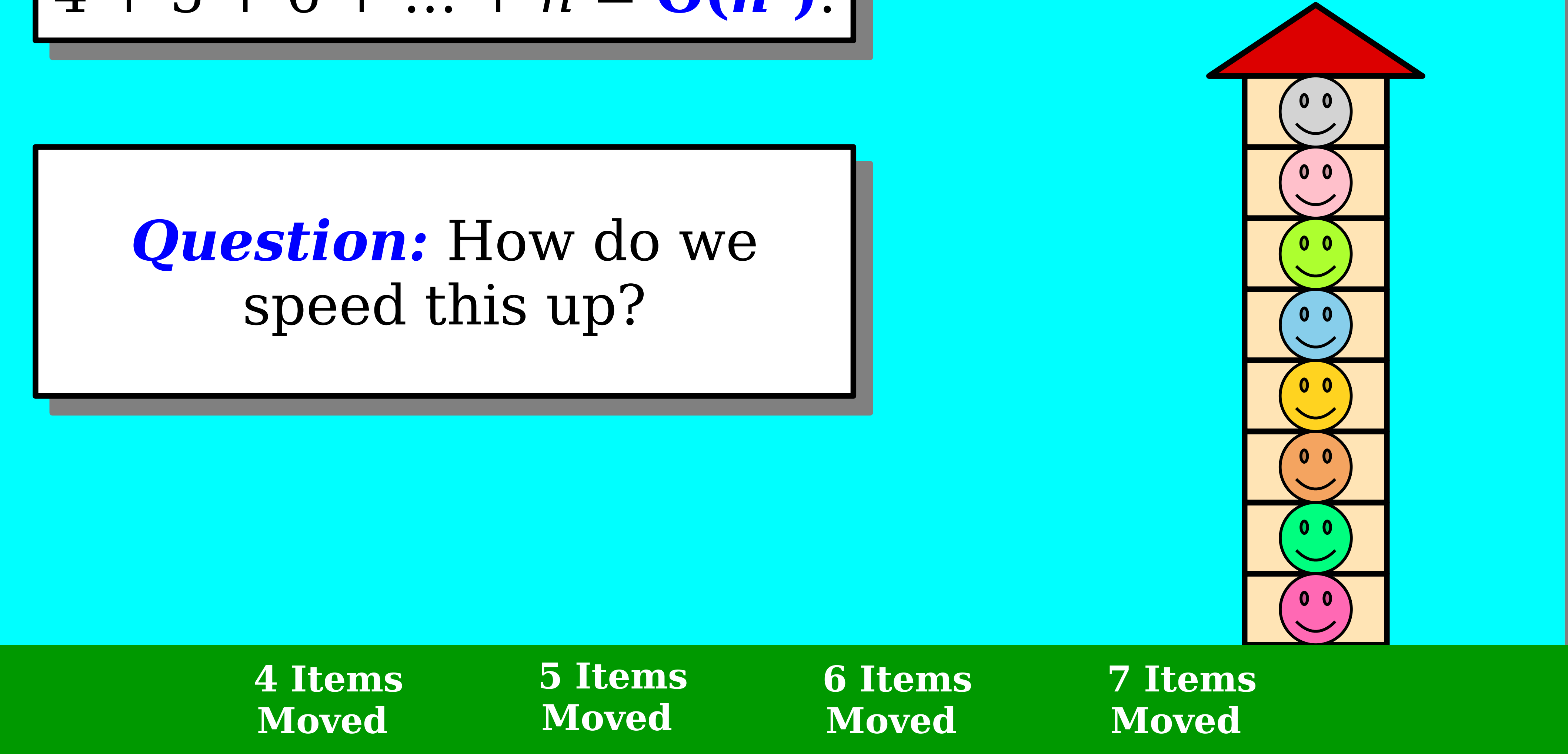
Every push beyond the first few requires moving all *n* elements from the old array to the new array.

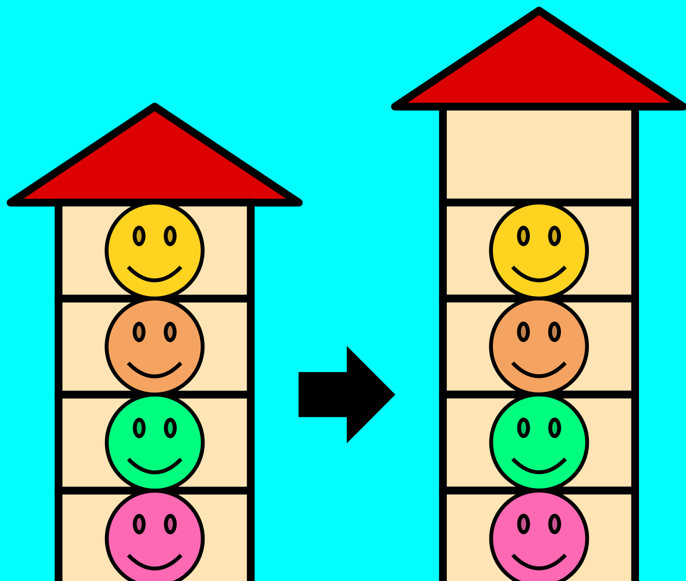
Cost of a single push: **O(*n*)**.

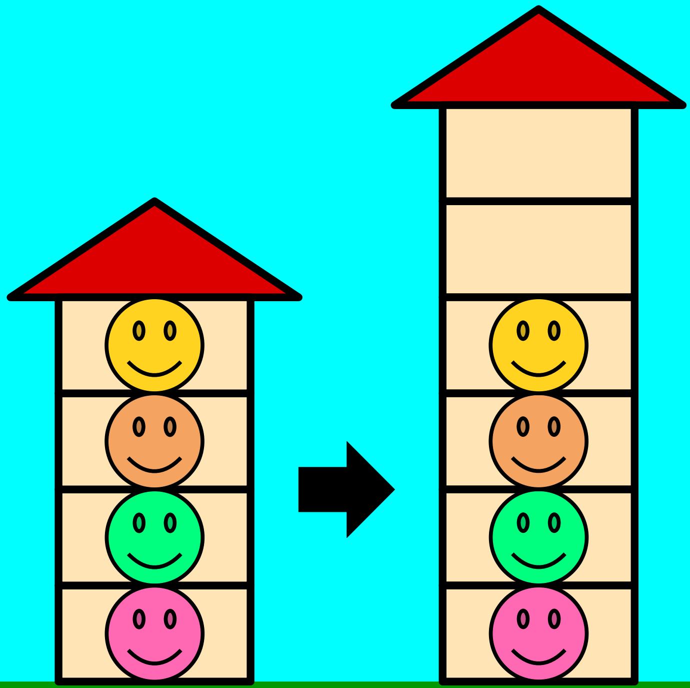
Every push beyond the first few requires moving all n elements from the old array to the new array.

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

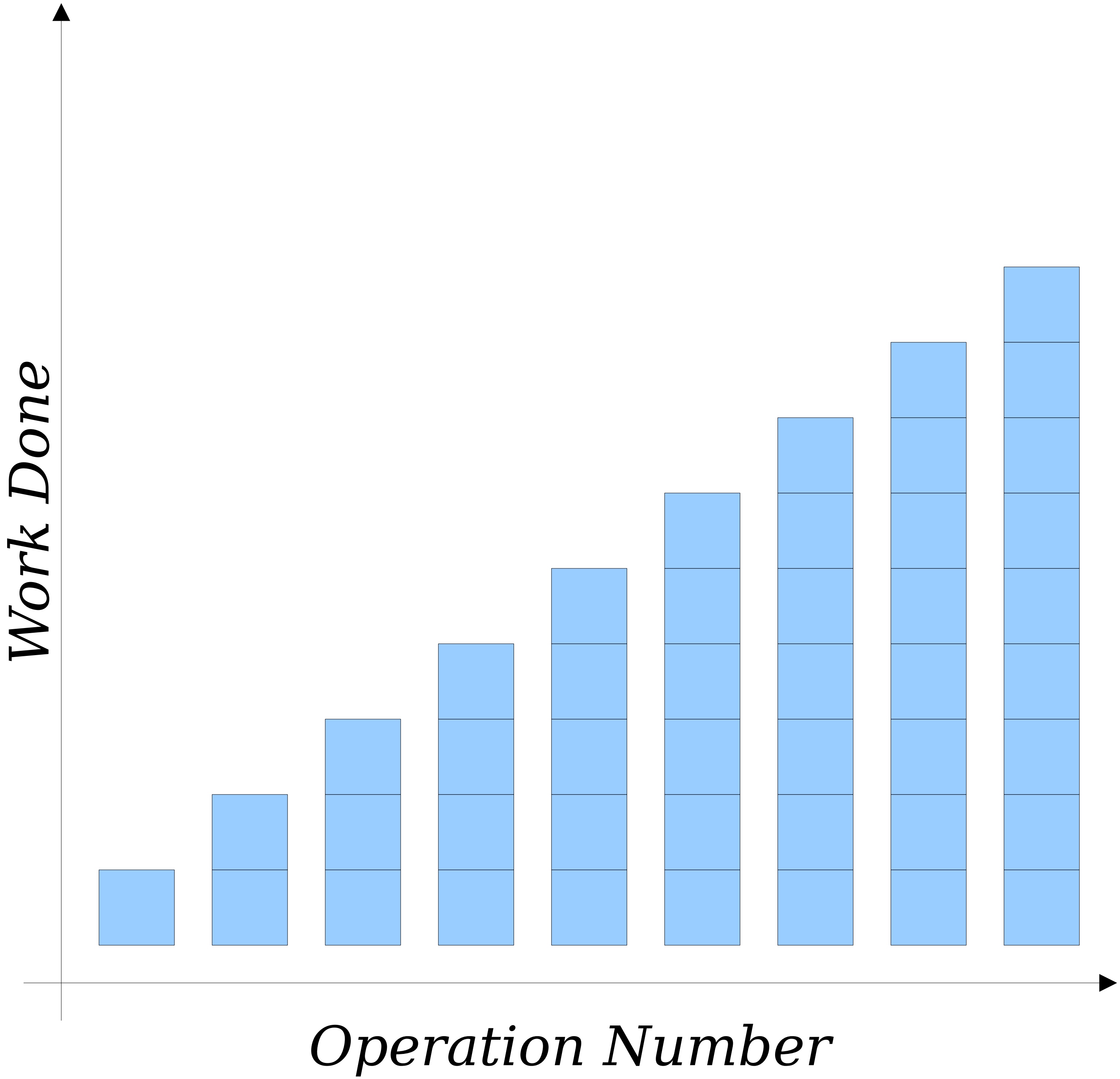
Question: How do we speed this up?





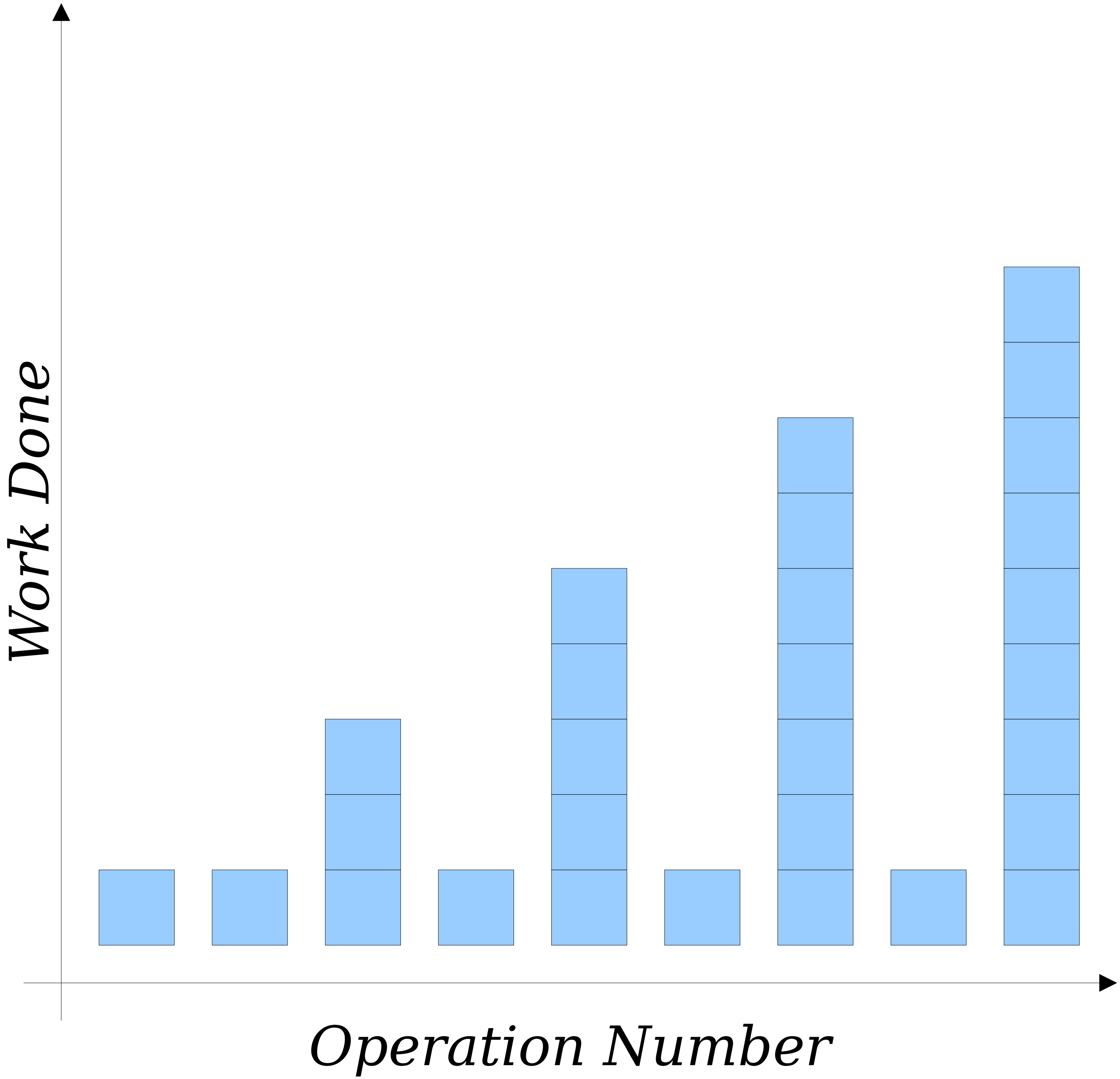


Now, only half the pushes we do will require moving everything to a new array.



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

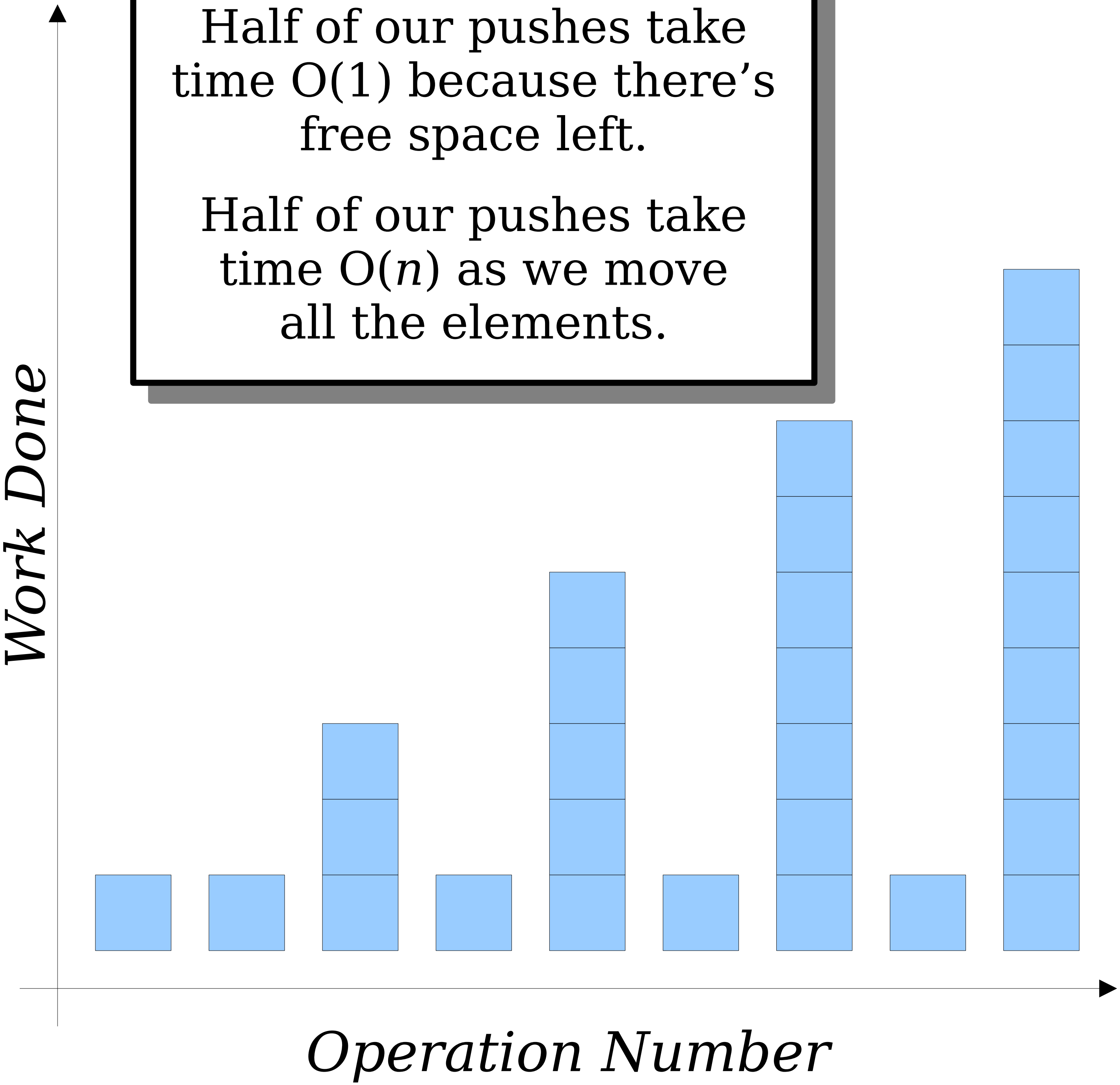
Increase array size by ***adding two***.



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.

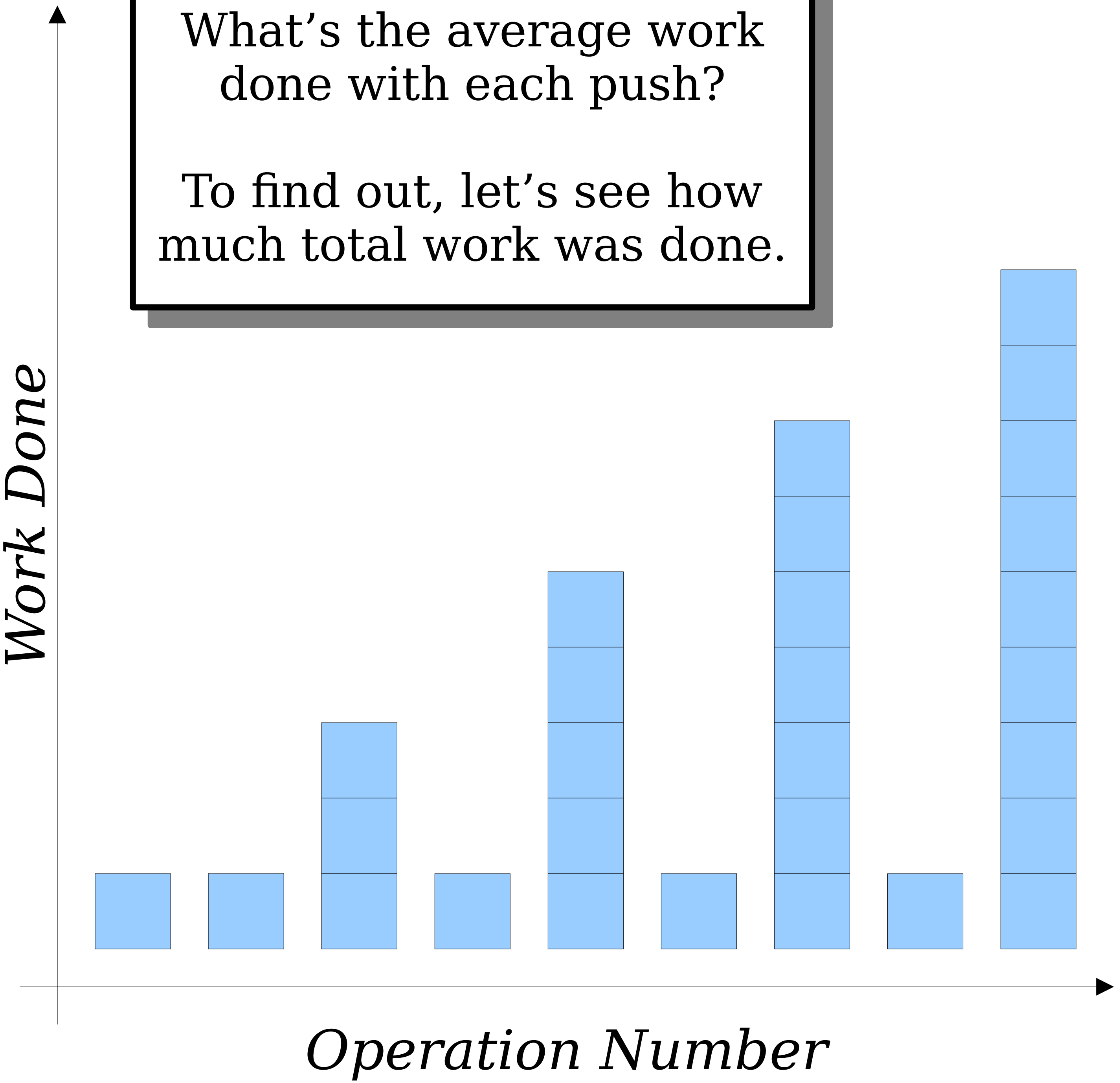
Increase array size by ***adding two***.



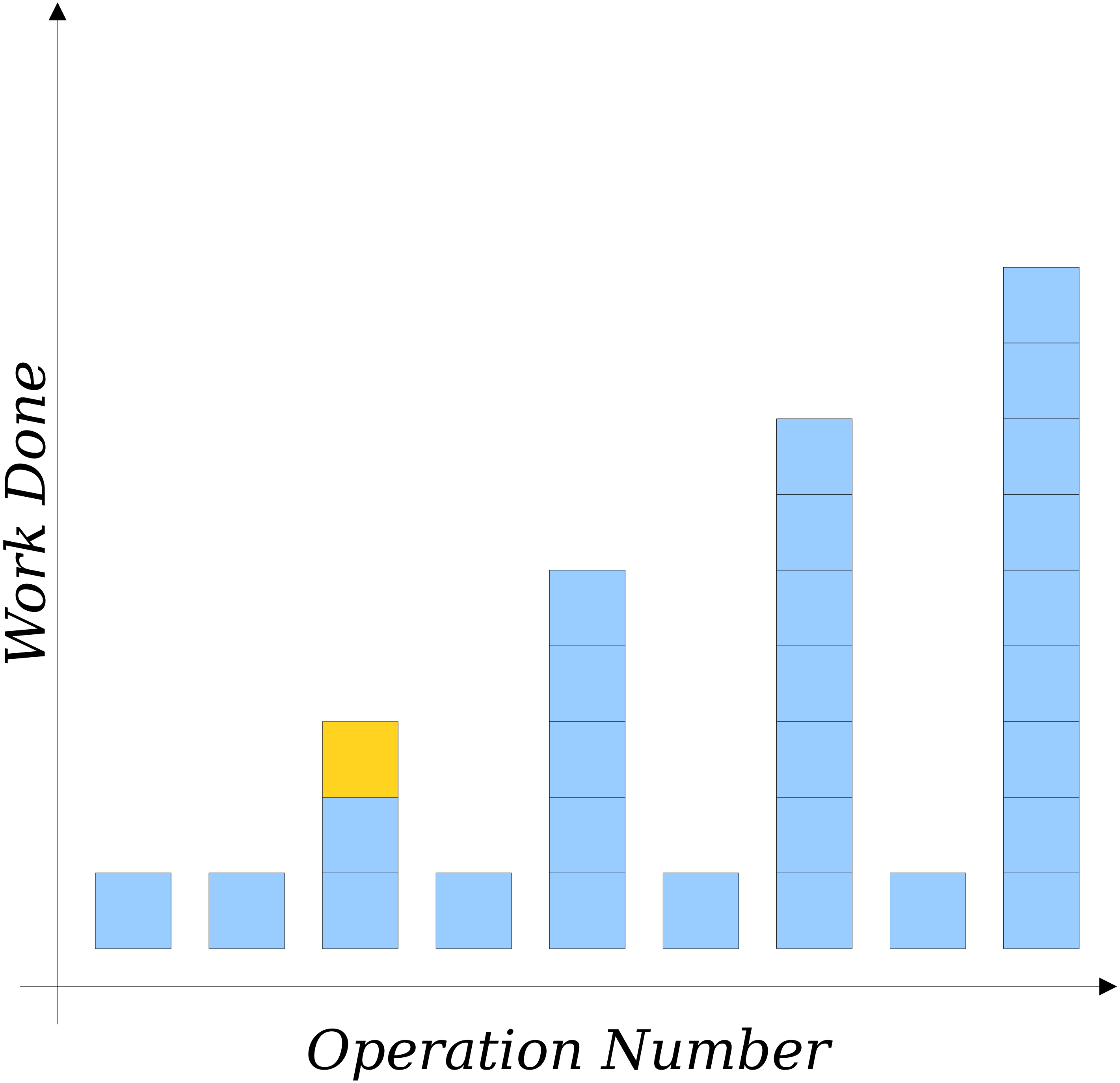
What’s the average work done with each push?

To find out, let’s see how much total work was done.

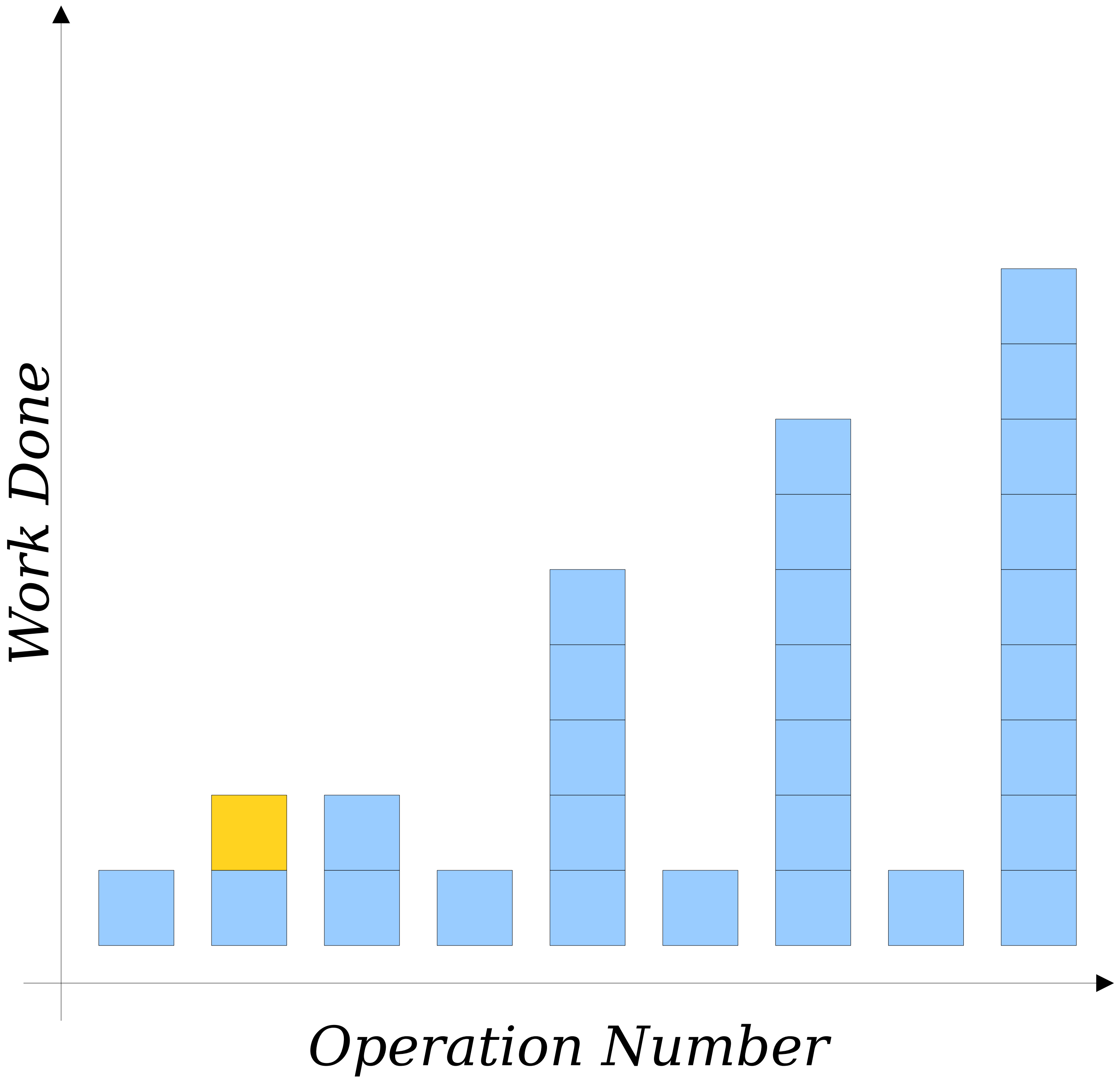
Increase array size by ***adding two***.



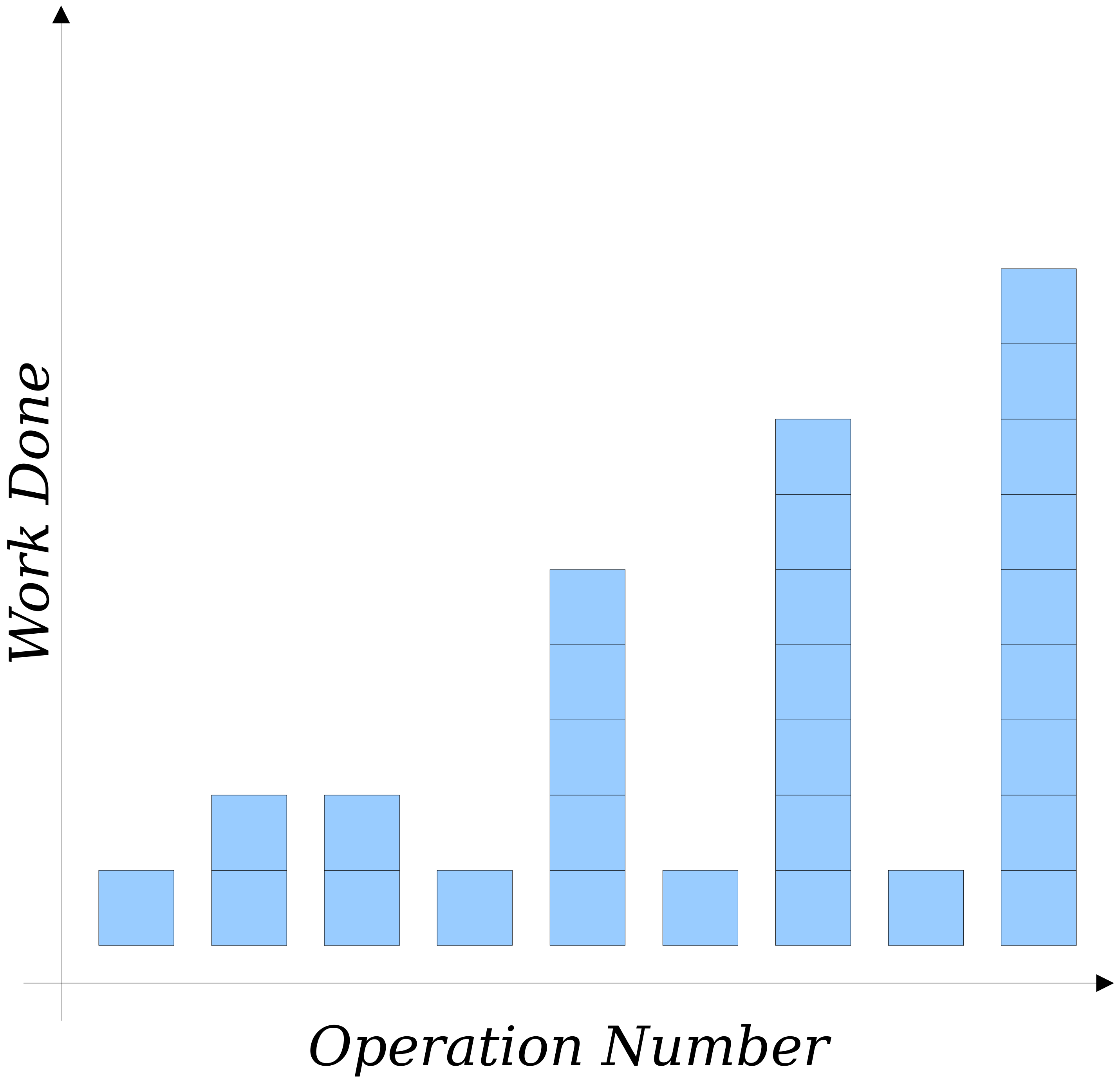
Increase array size by ***adding two***.



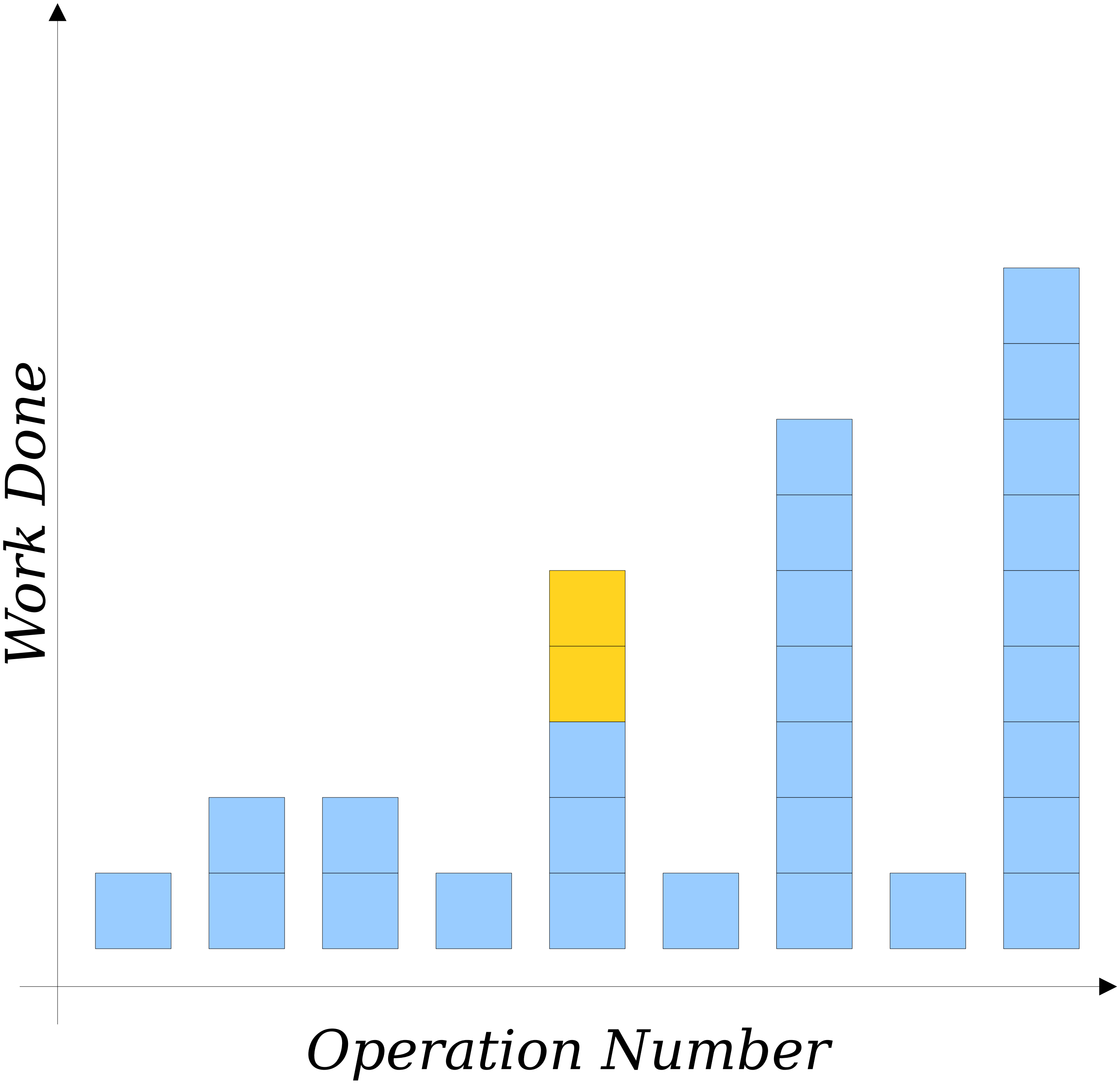
Increase array size by ***adding two***.



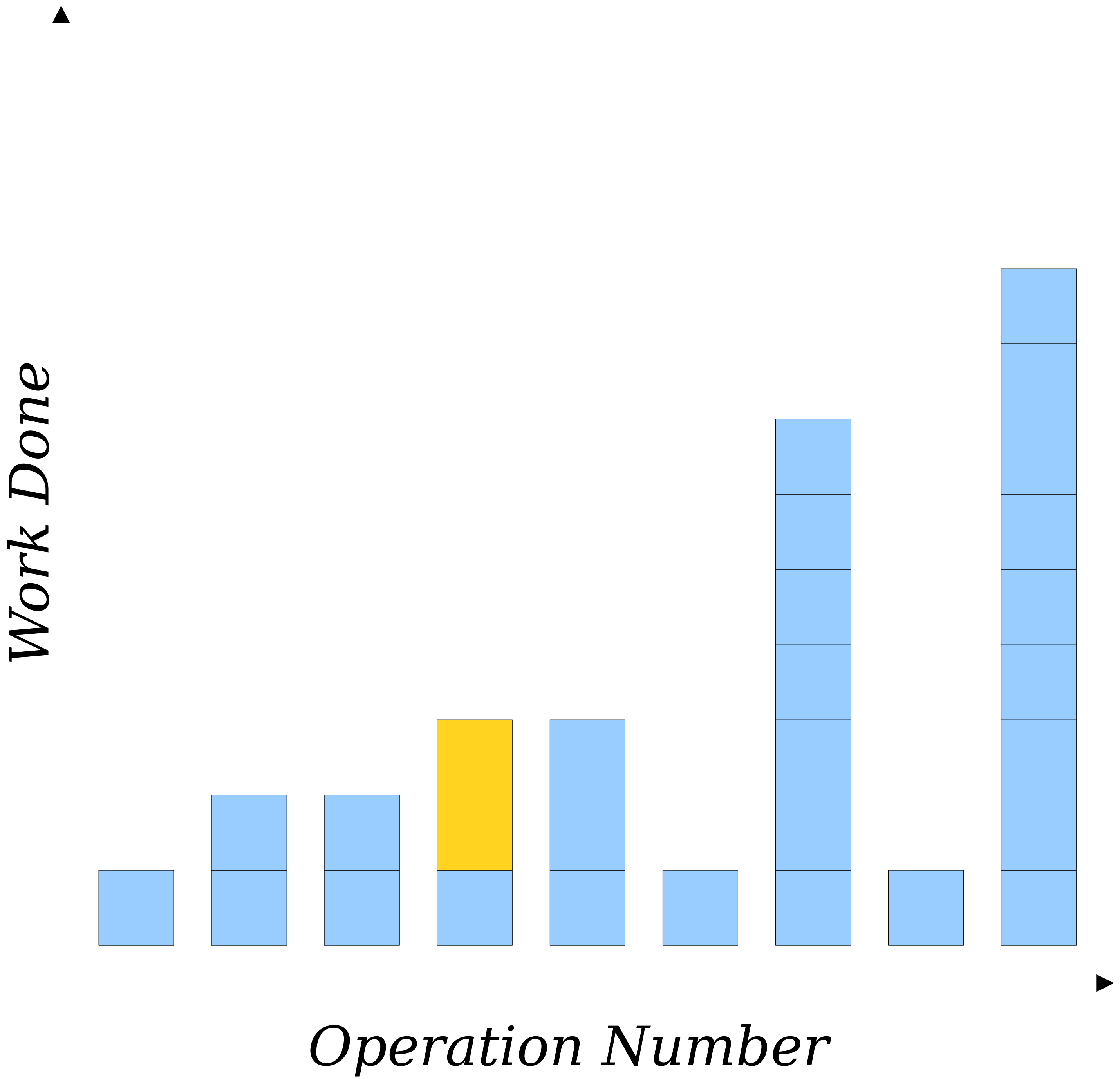
Increase array size by ***adding two***.



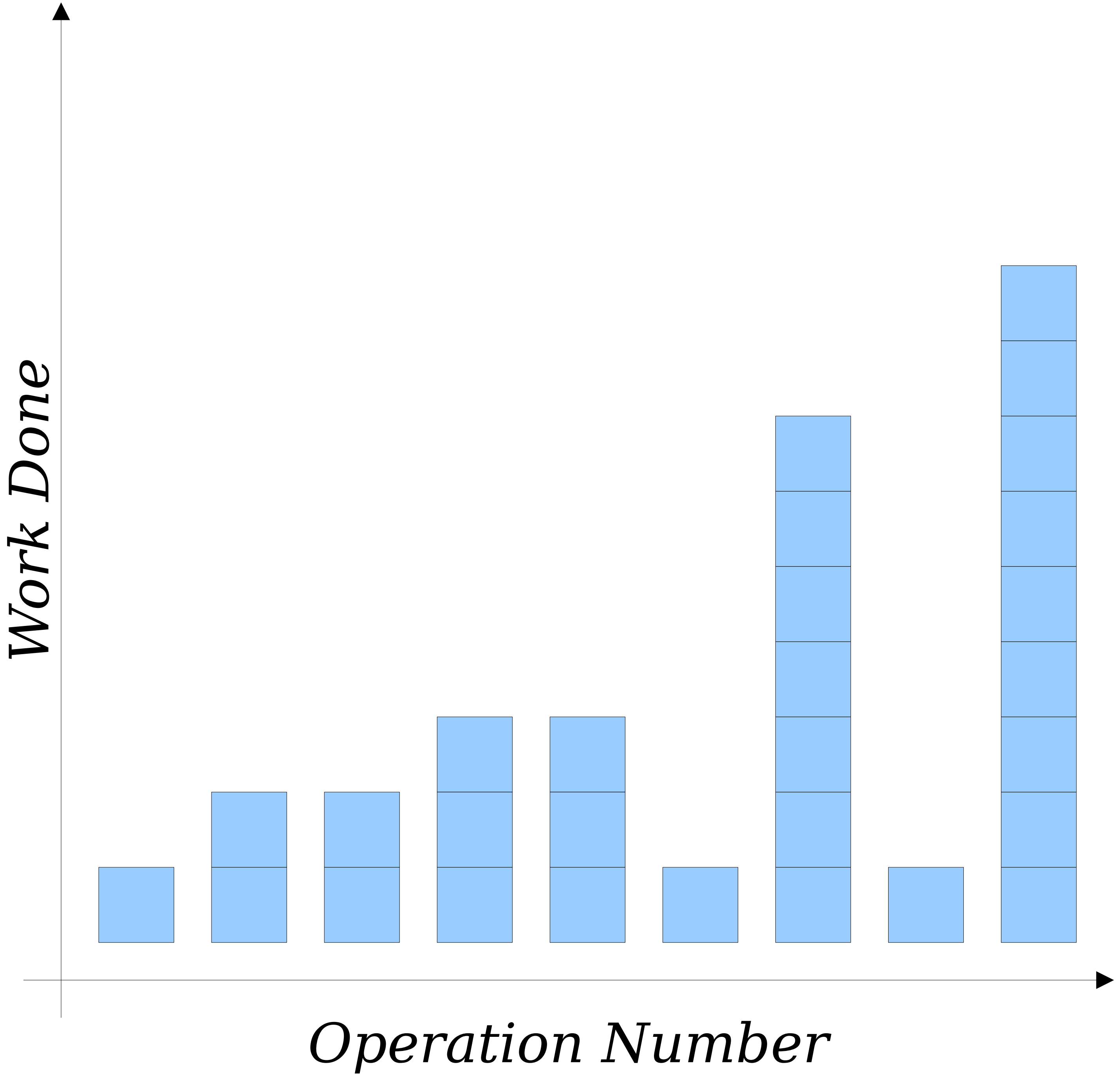
Increase array size by ***adding two***.



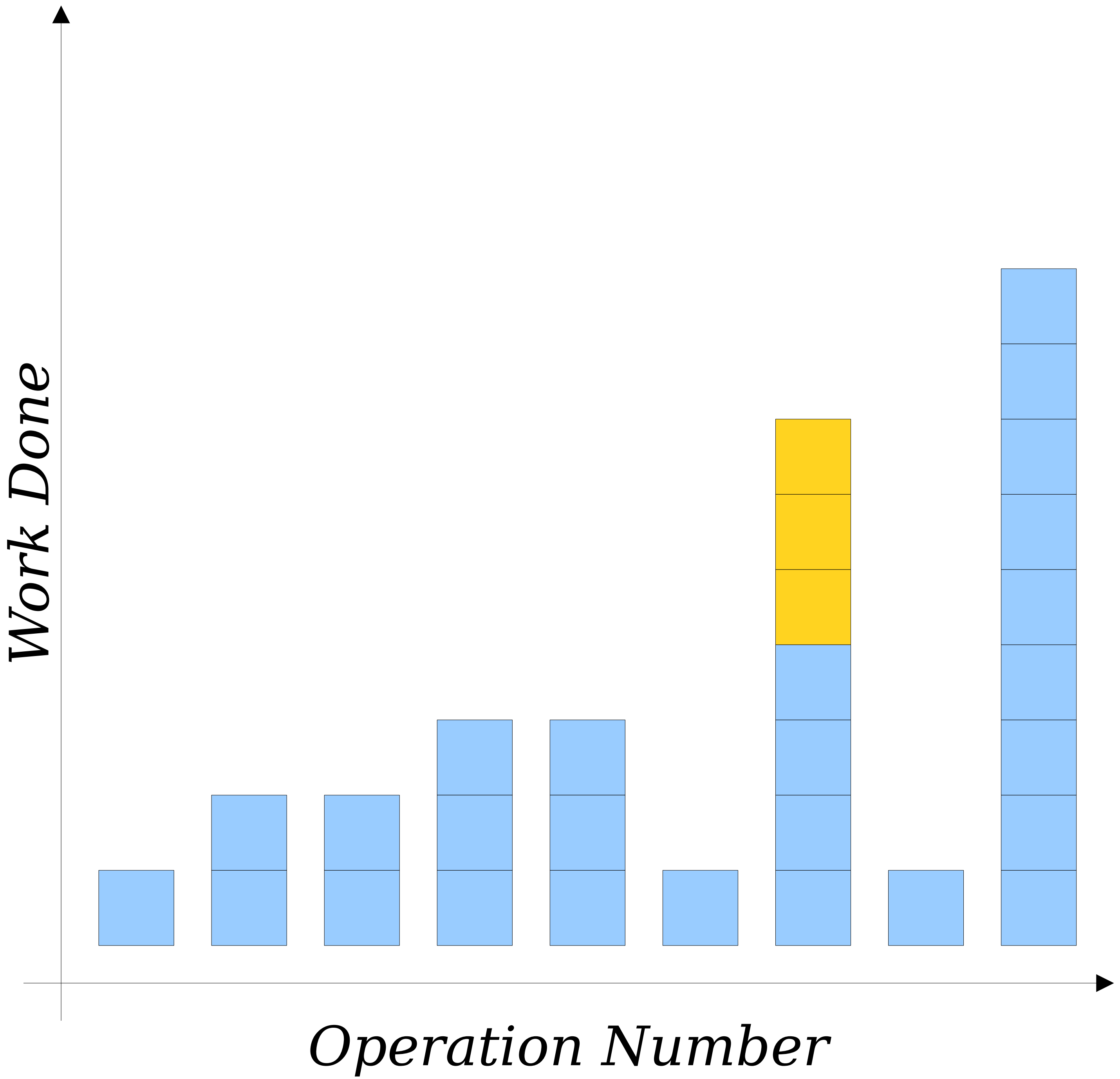
Increase array size by ***adding two***.



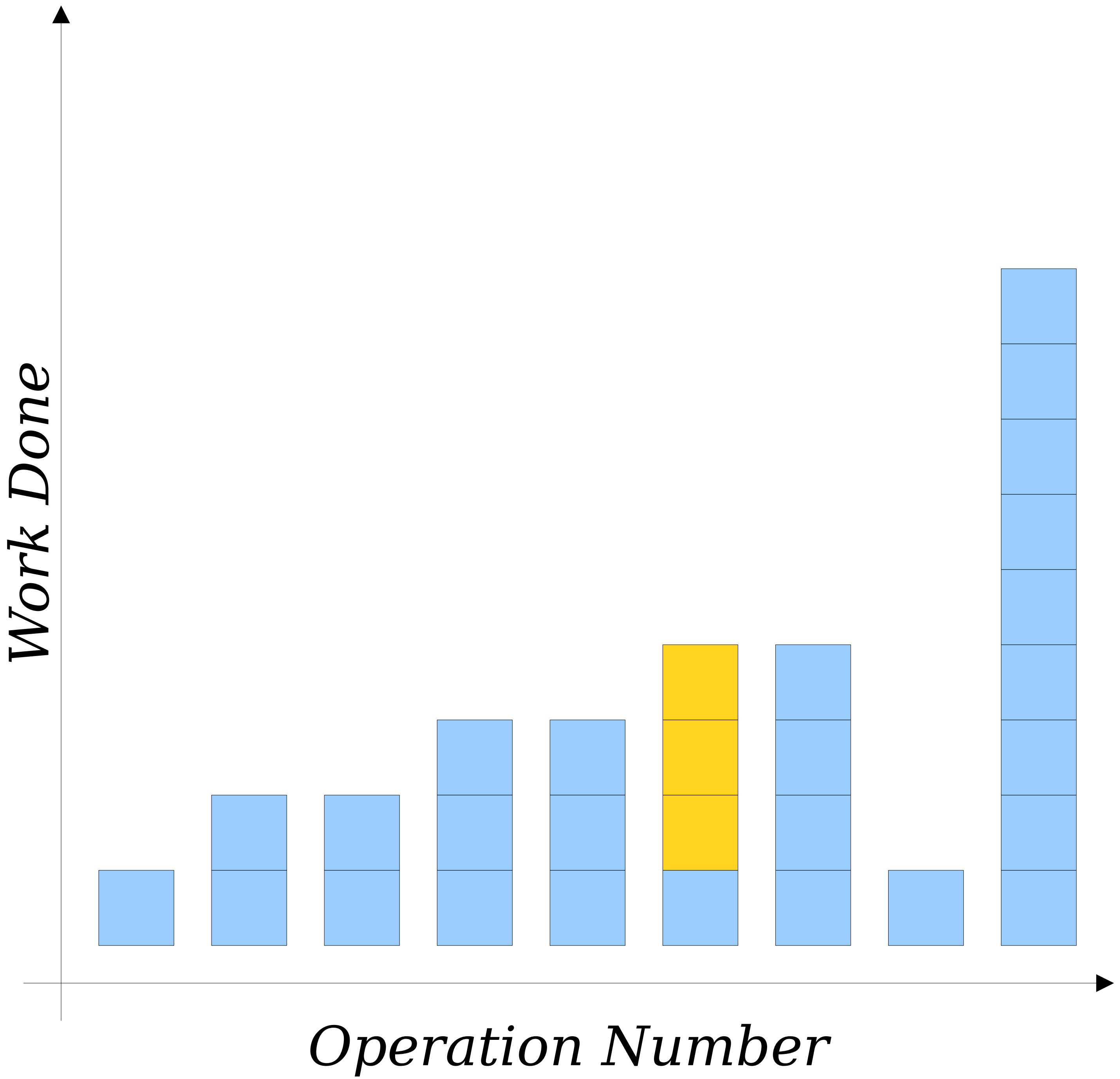
Increase array size by ***adding two***.



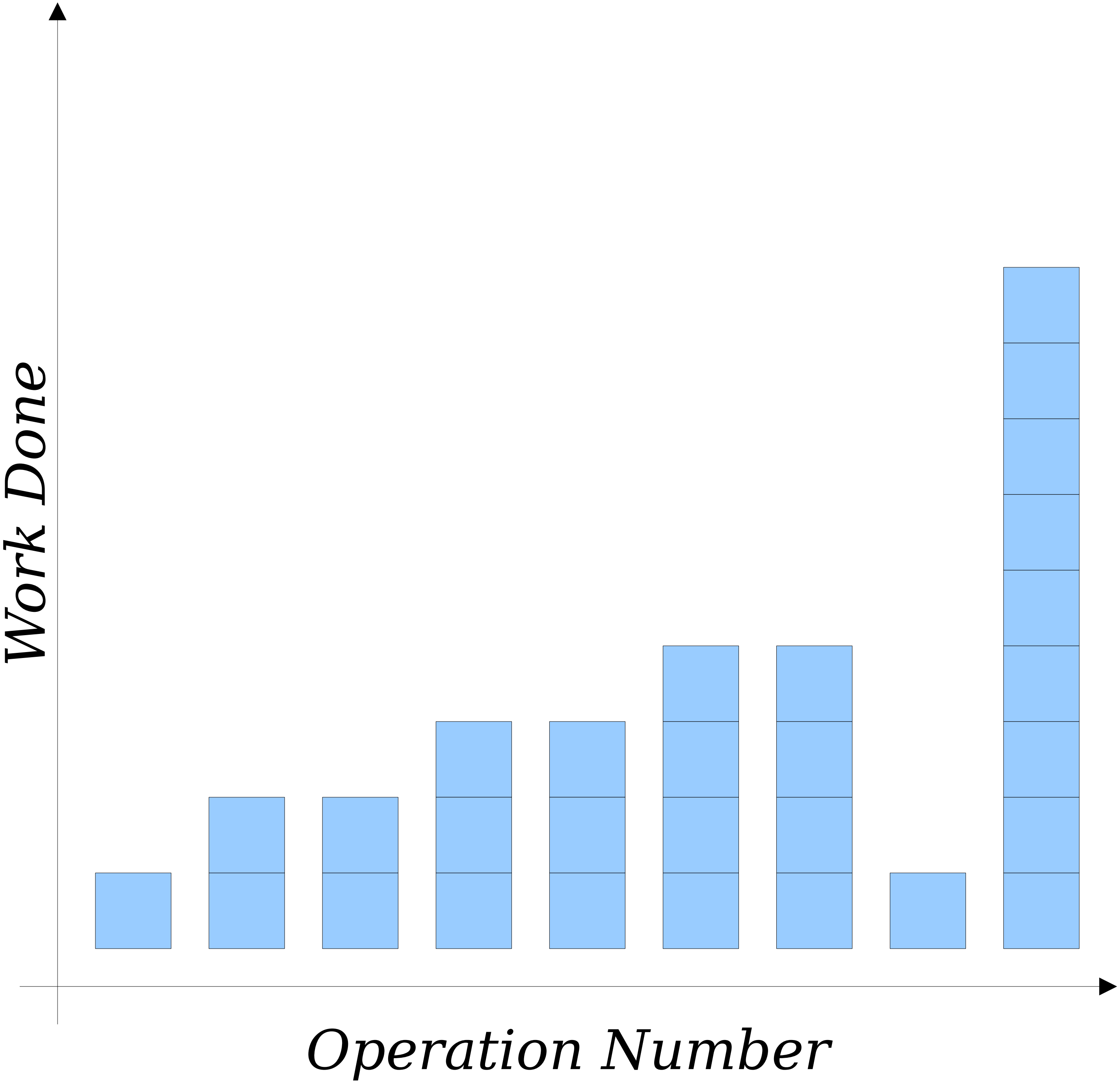
Increase array size by ***adding two***.



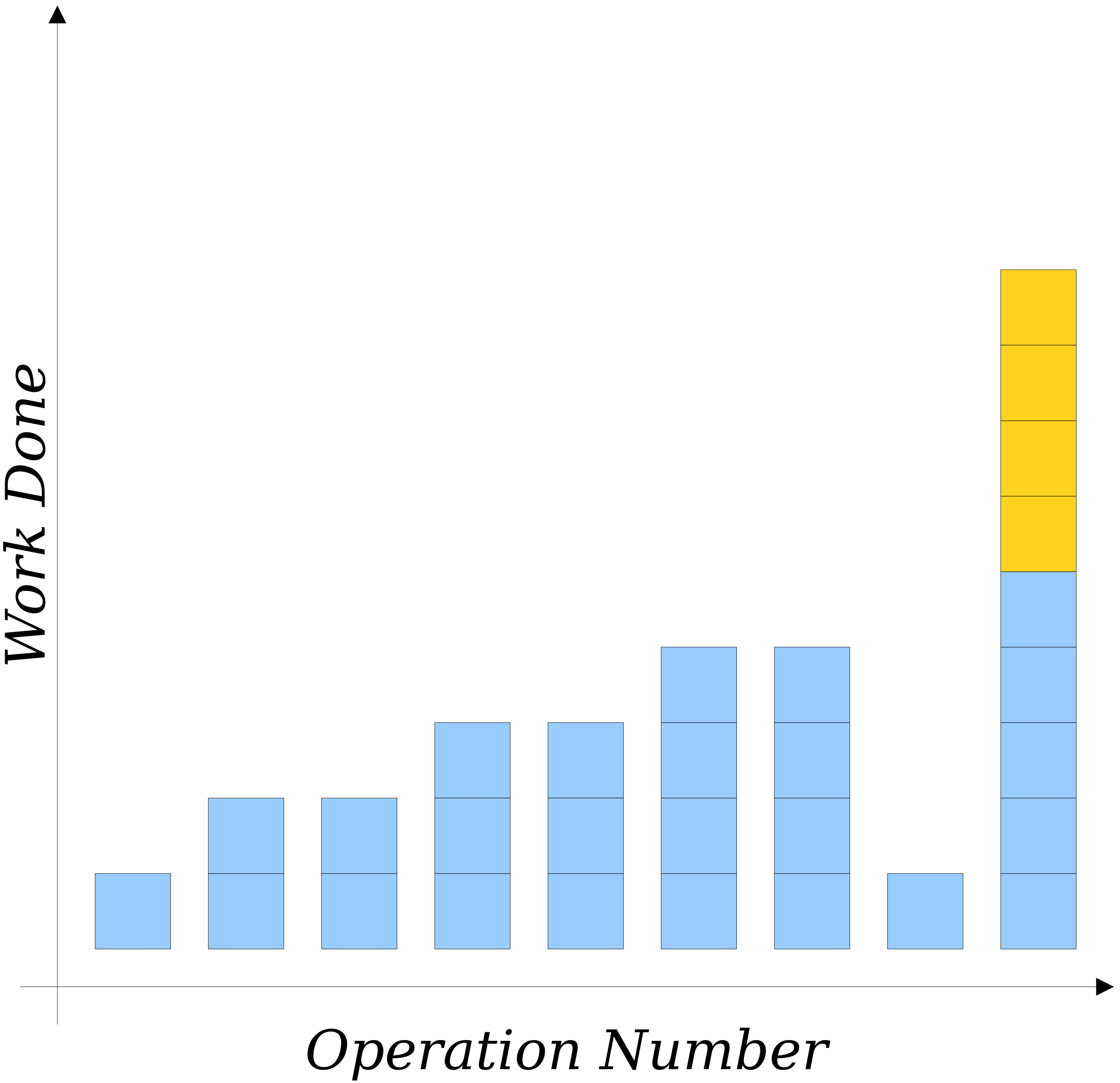
Increase array size by ***adding two***.



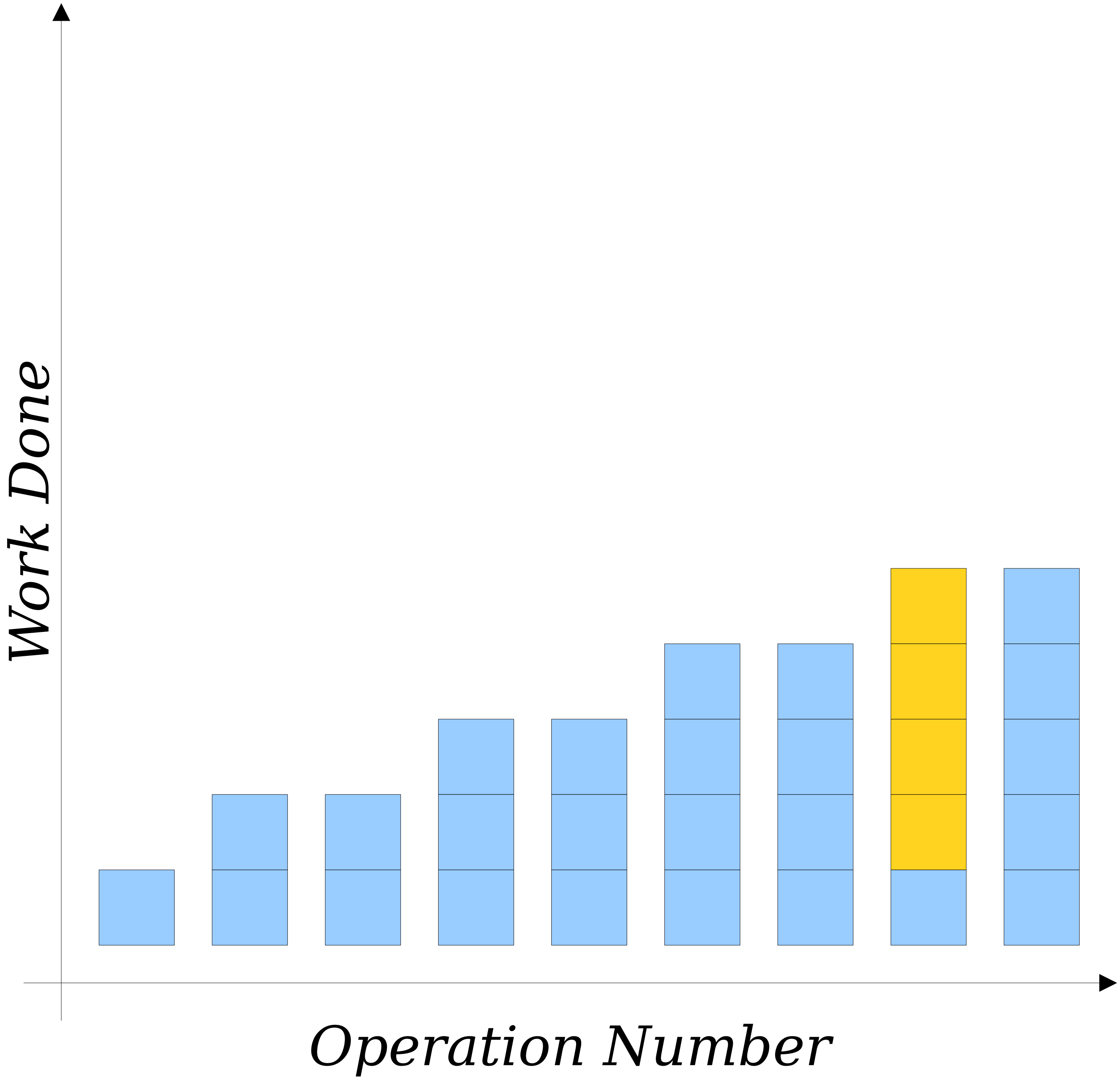
Increase array size by ***adding two***.



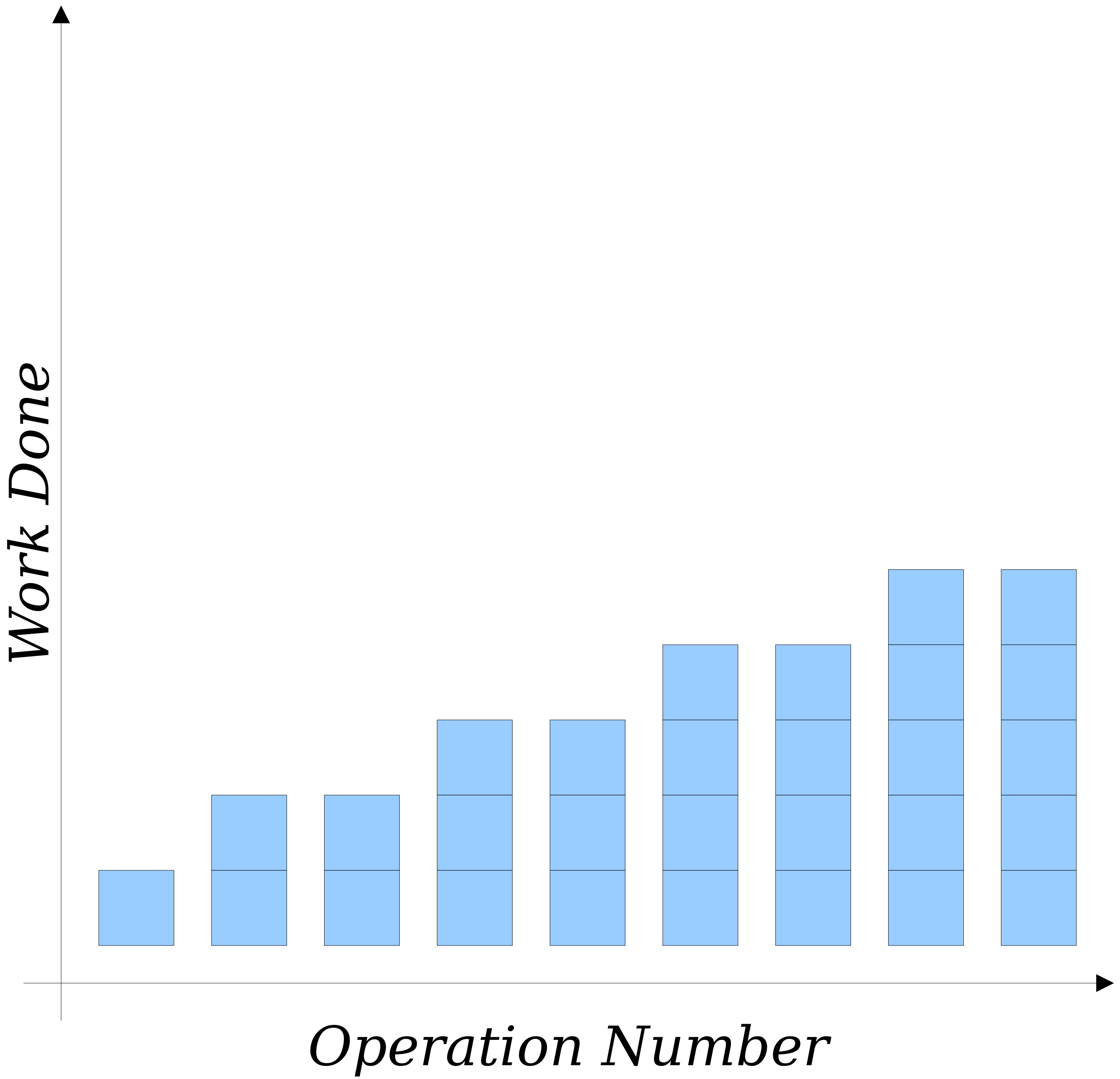
Increase array size by ***adding two***.

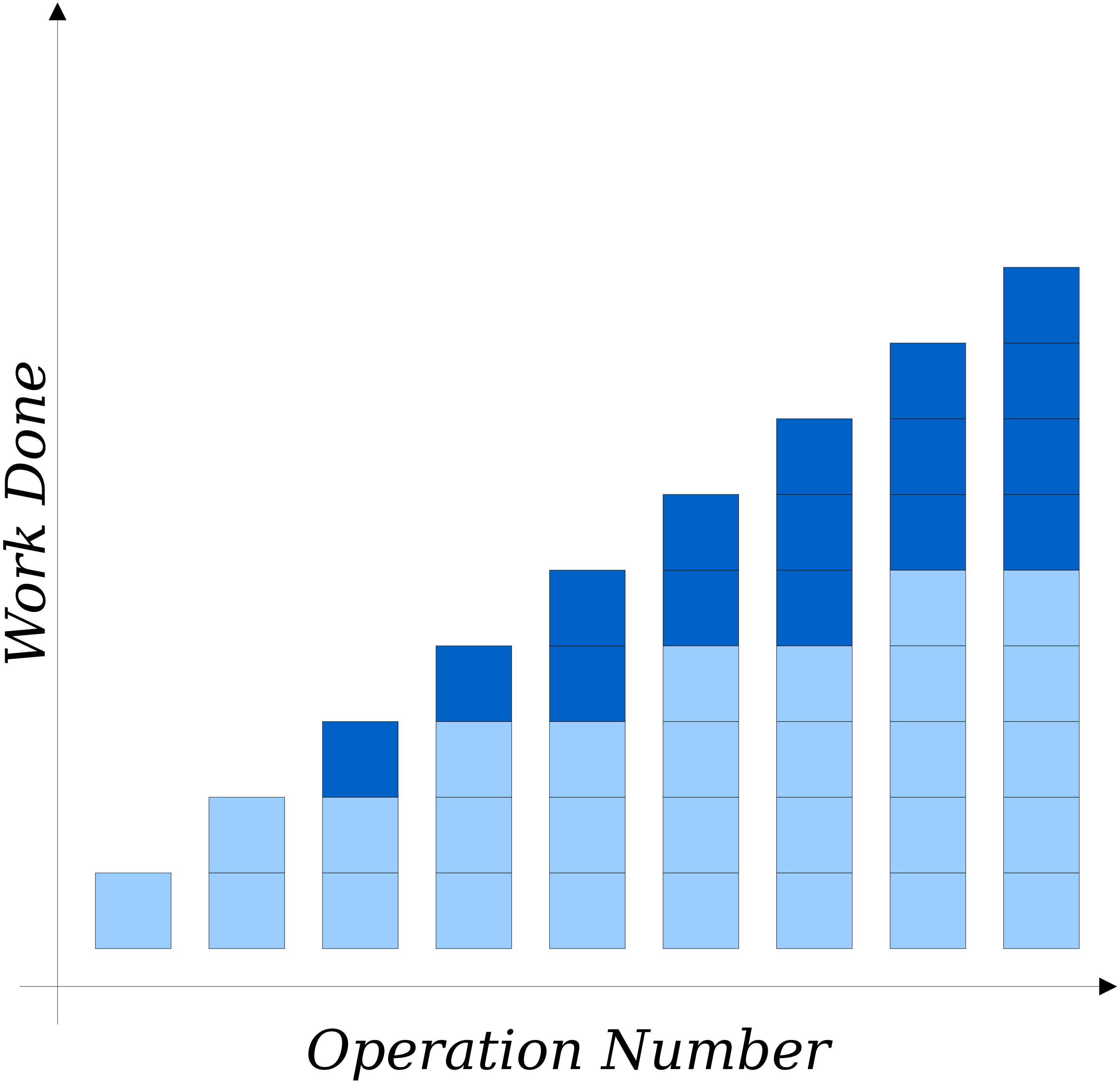


Increase array size by ***adding two***.



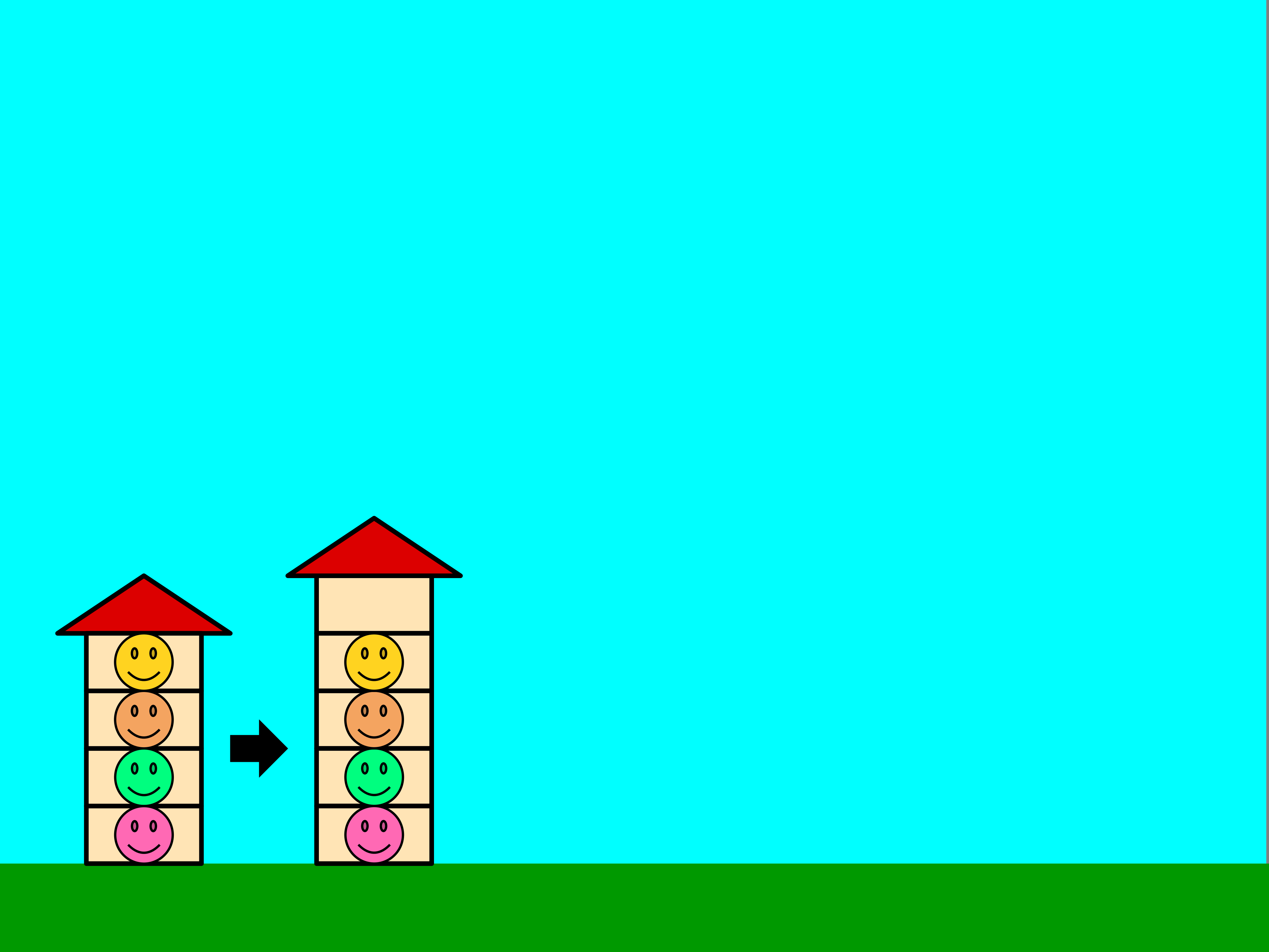
Increase array size by ***adding two***.

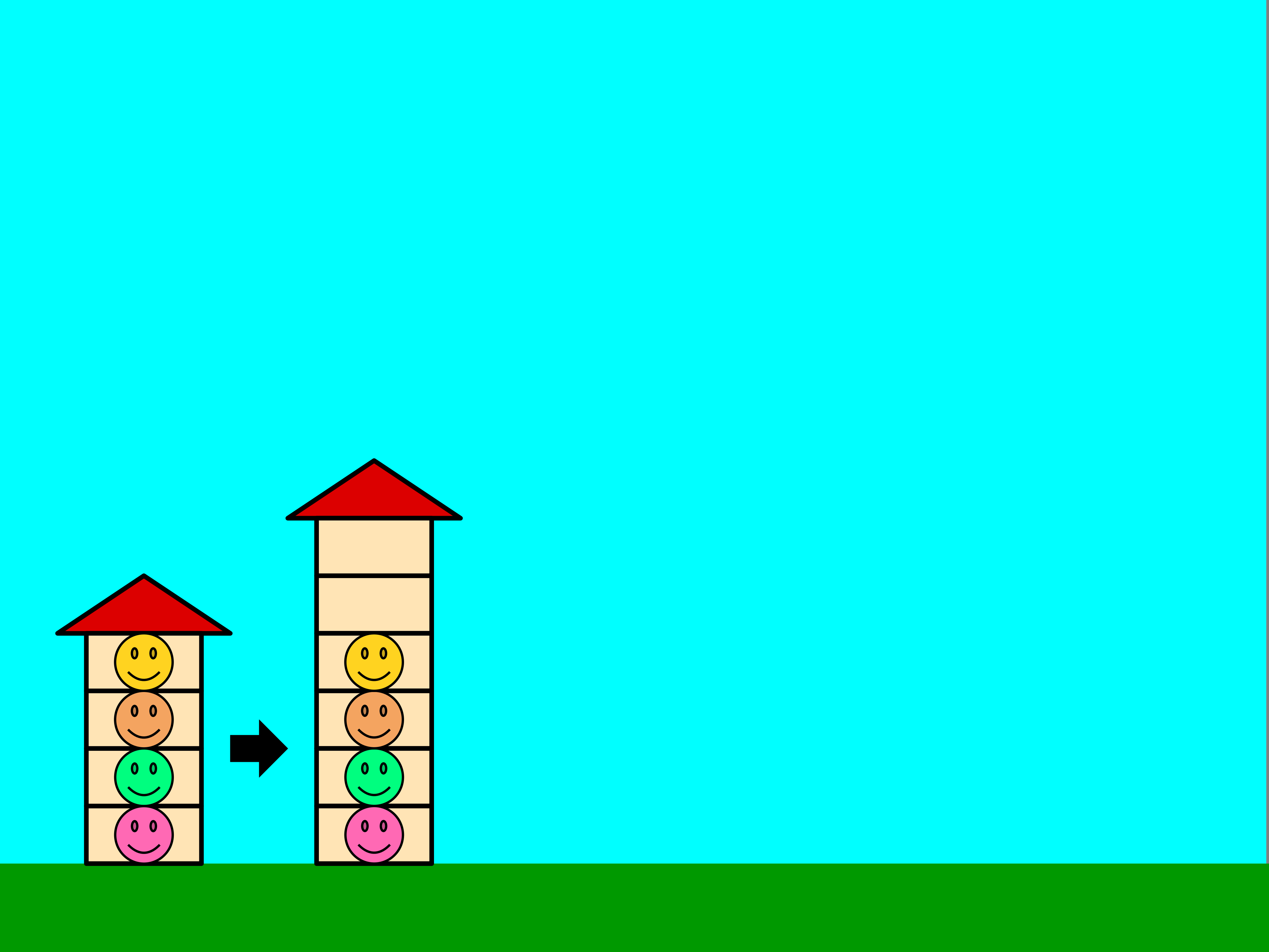


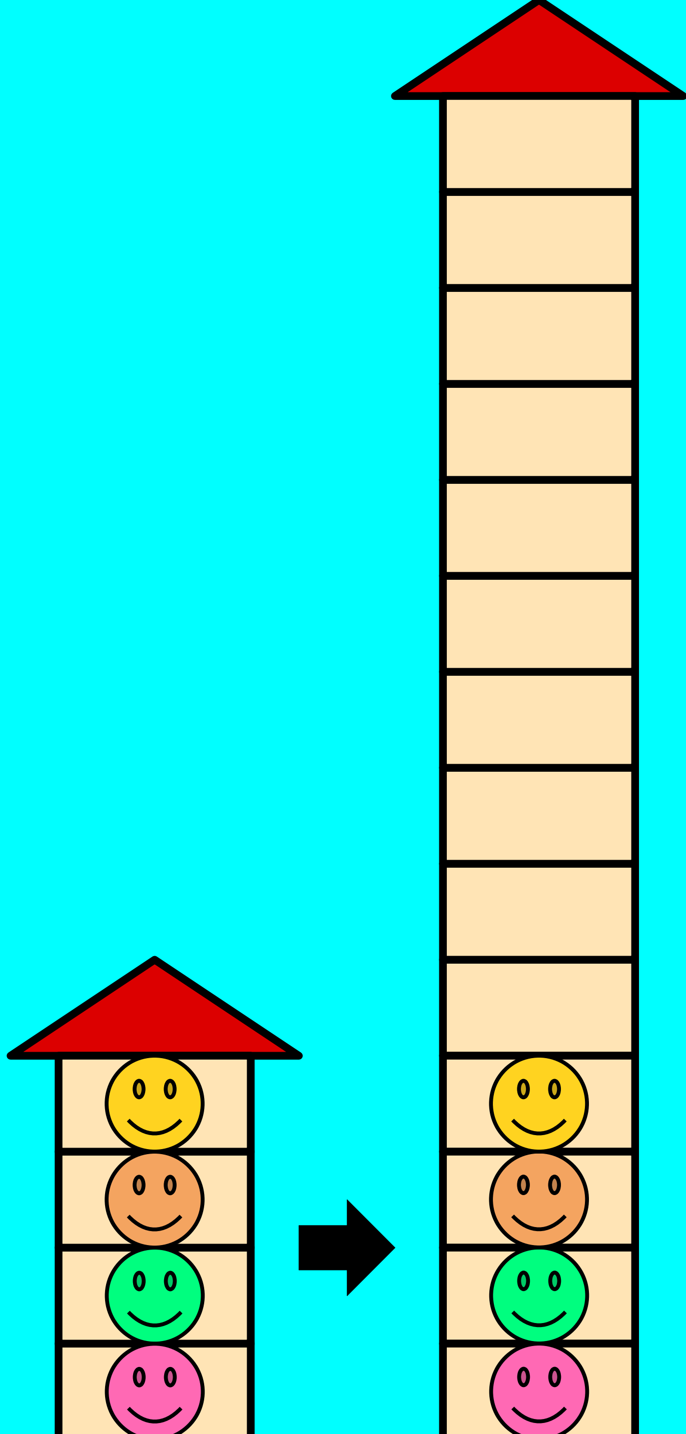


Increase array size by ***adding two***.

This roughly halves the work done.

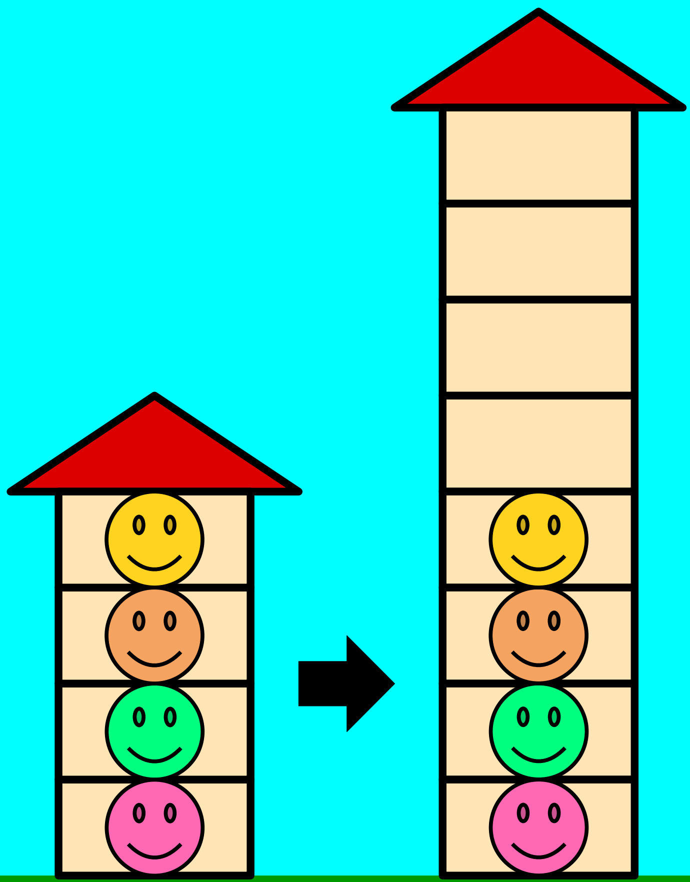






If we make the new array too big, we're might not make use of all the new space.

What's a good compromise?



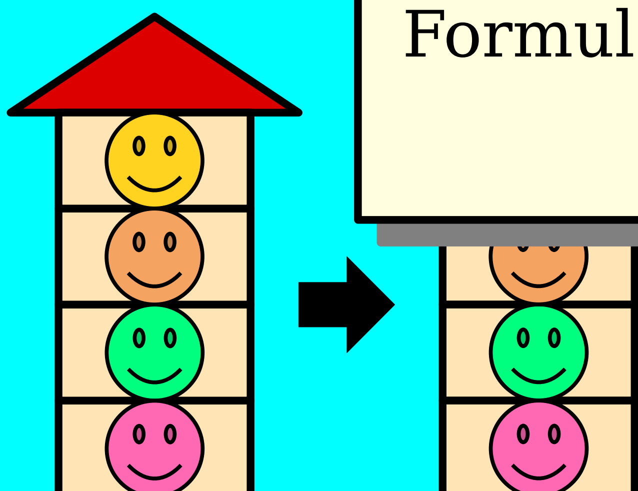
Idea: Make the new array twice as big as the old one.

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

What is the big-O cost of a push?
What is the big-O cost of *n* pushes?

Take thirty seconds to think this over.
Look over the runtime plot and the code for the push operation.

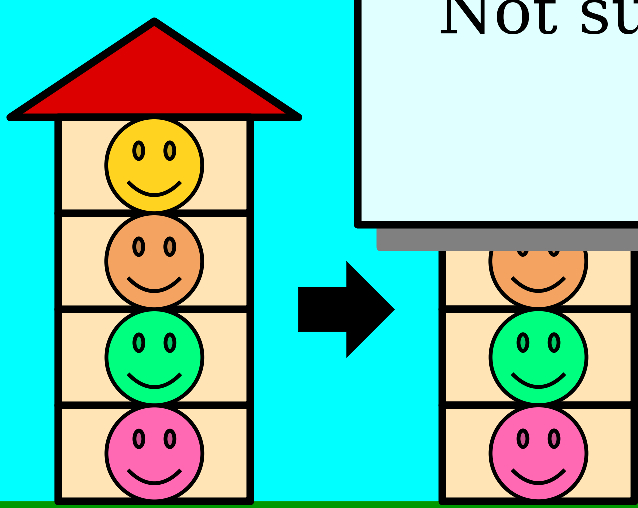
Formulate a hypothesis, but don't post it into chat just yet.



What is the big-O cost of a push?
What is the big-O cost of *n* pushes?

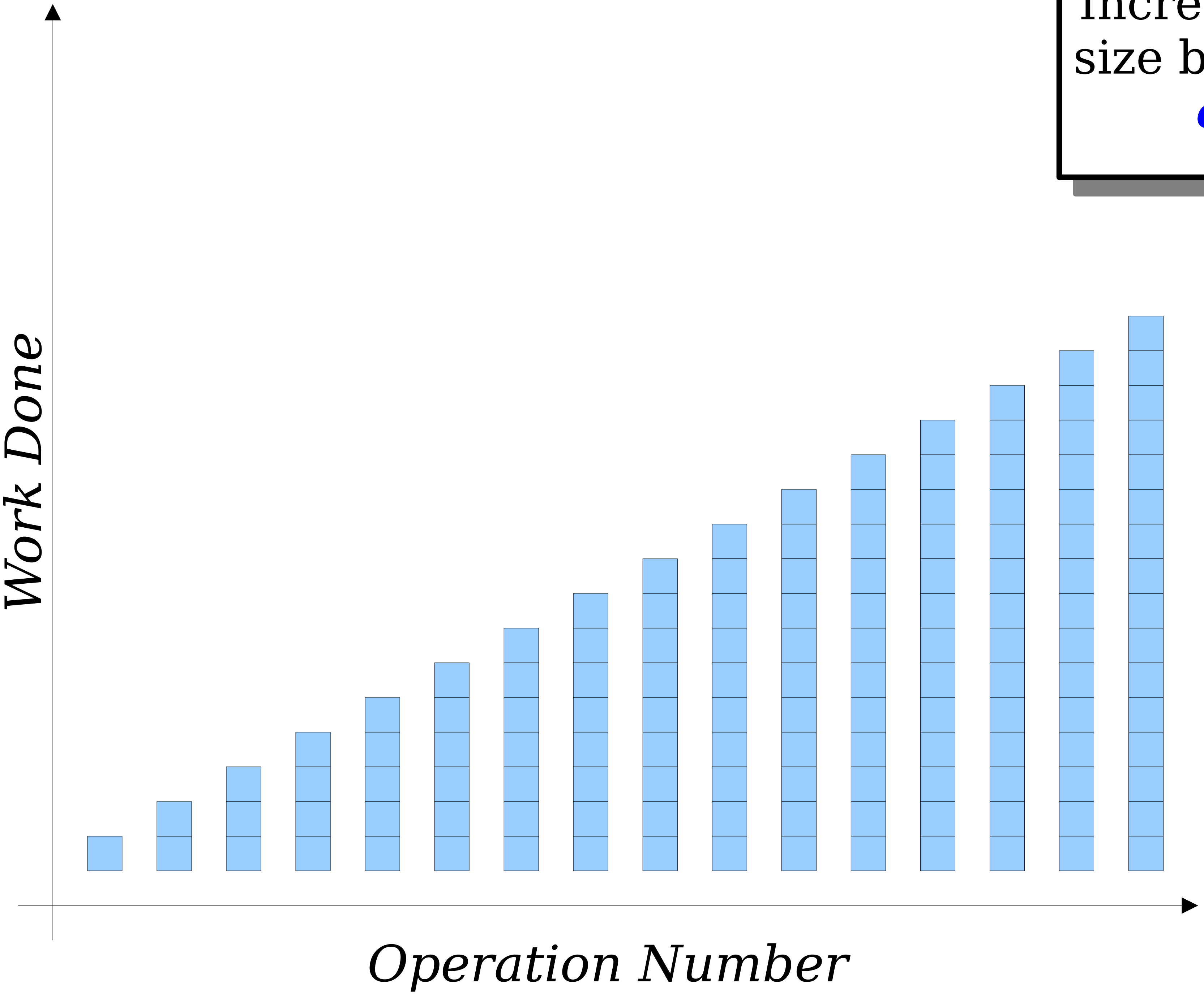
Now, post your best guess in chat.

Not sure? Just respond with “??”.

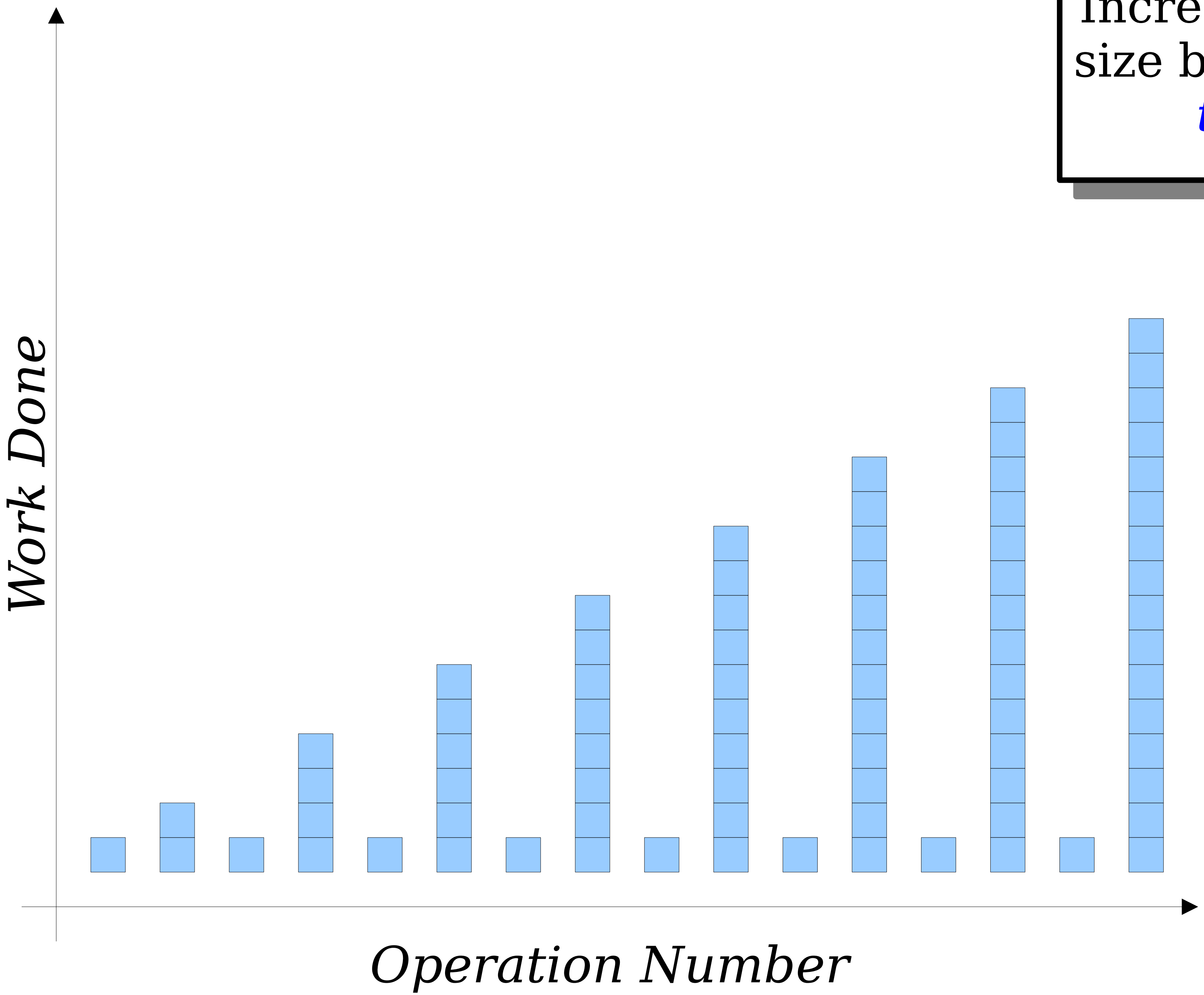


How do we analyze this?

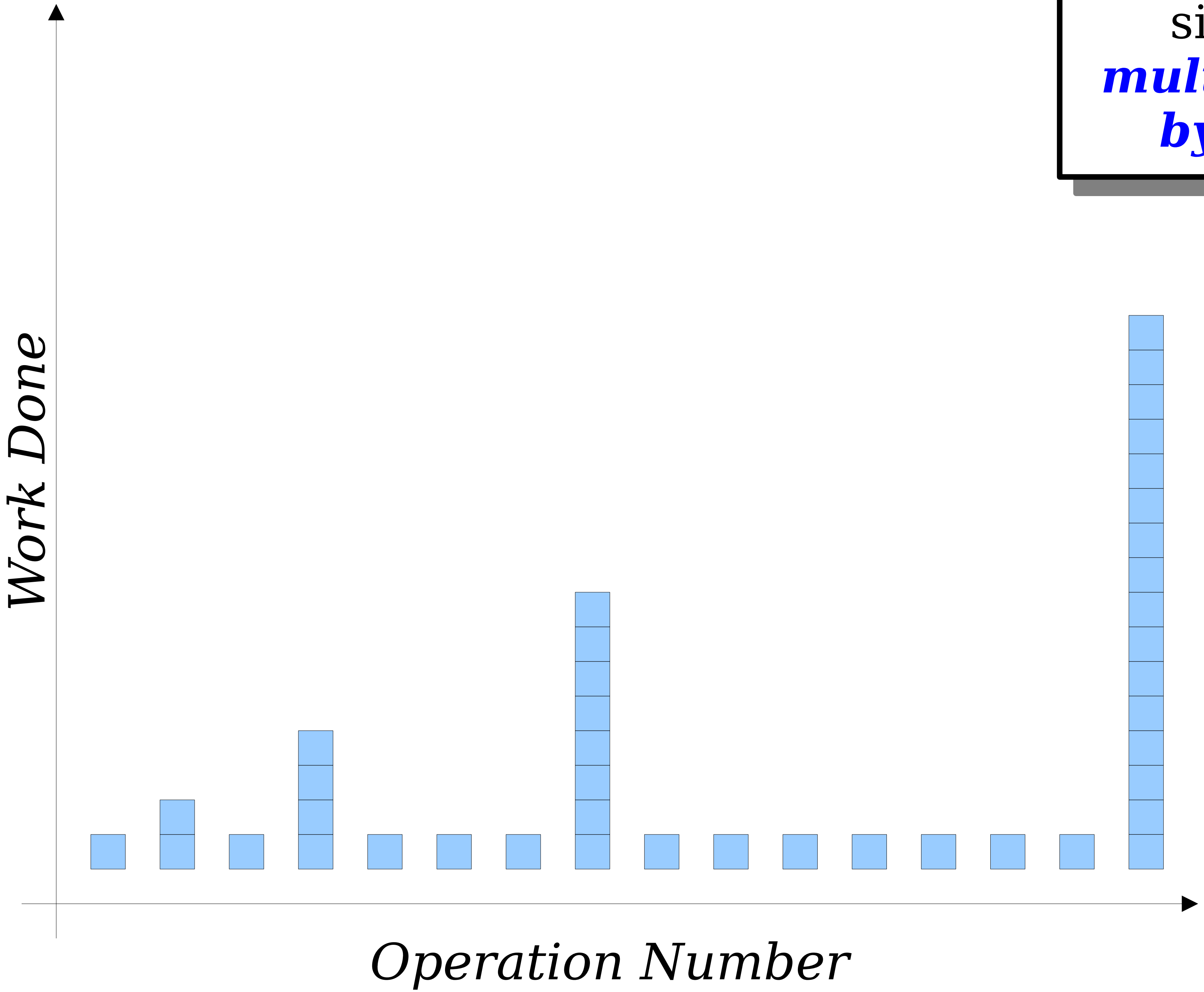
Increase array size by ***adding one***.



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



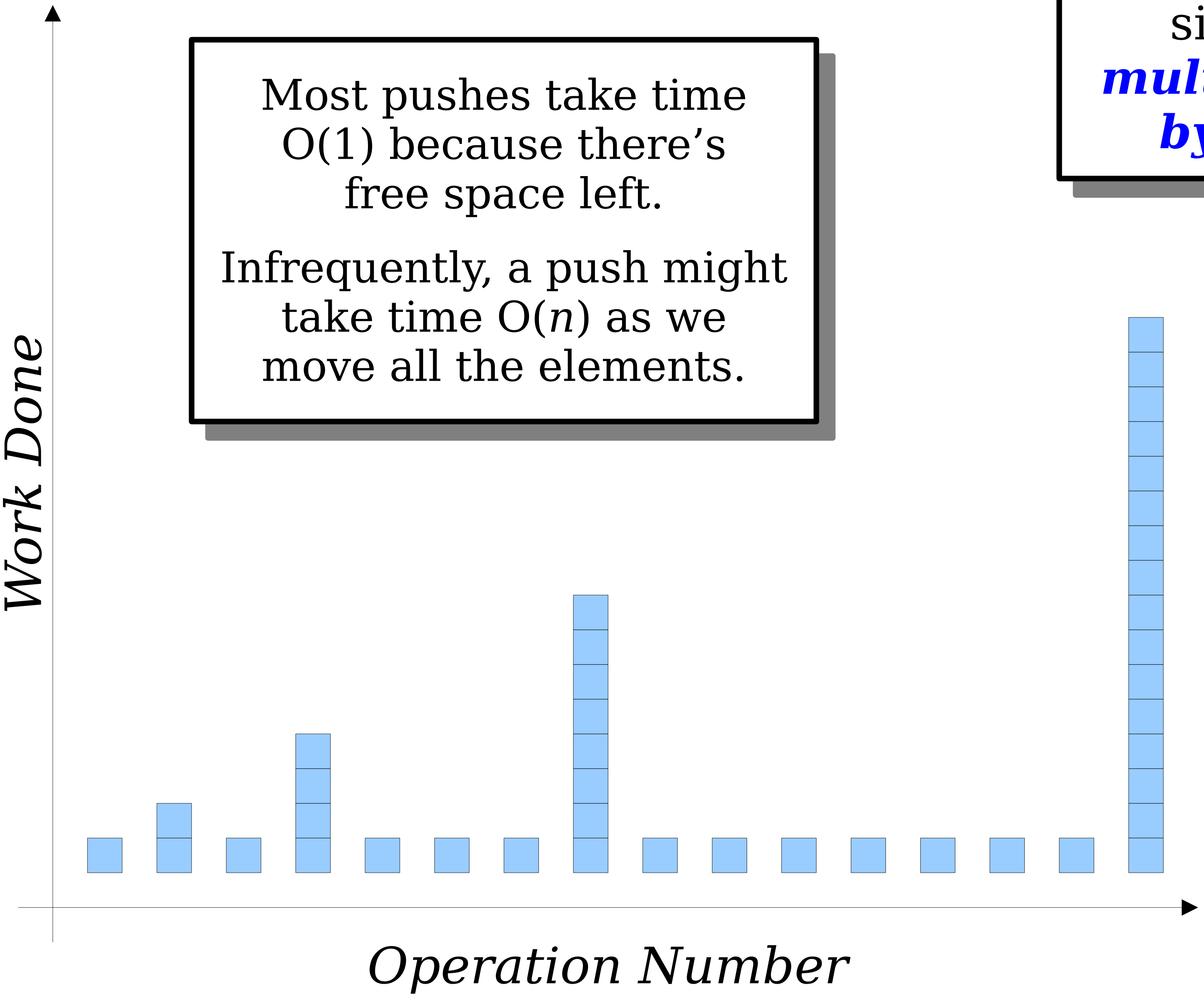
Increase array size by ***multiplying by two***.



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

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

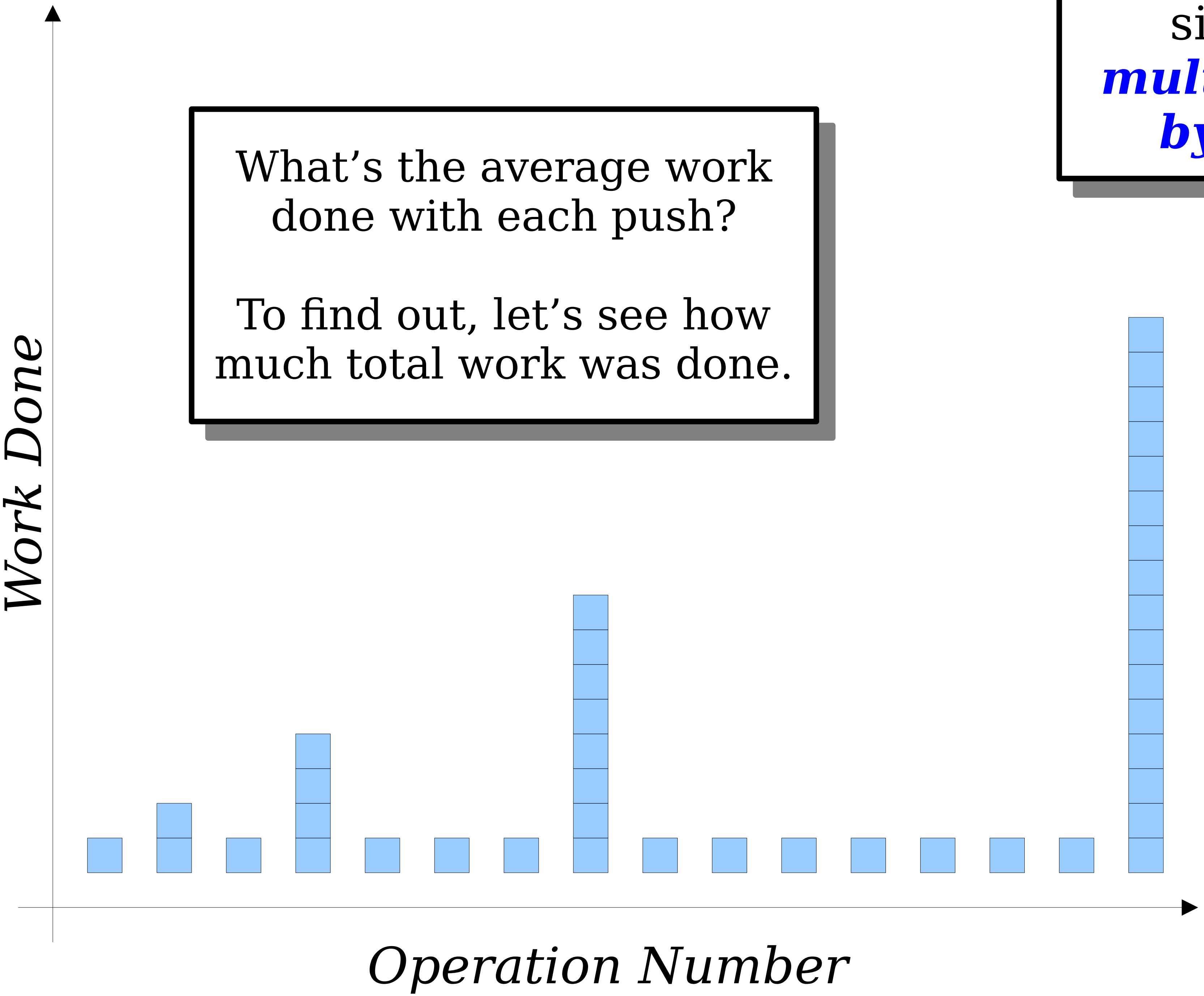
Infrequently, a push might take time O(*n*) as we move all the elements.



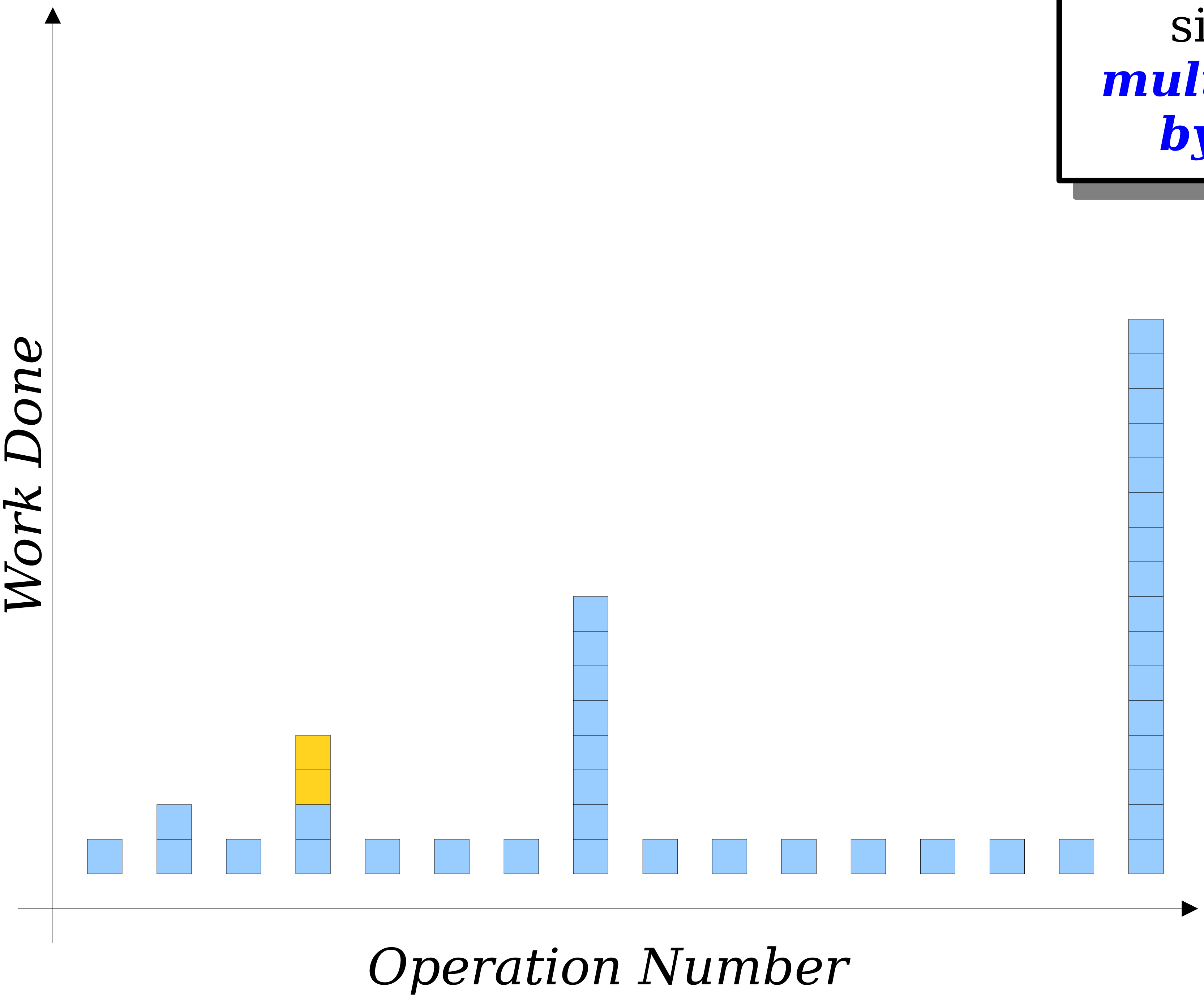
Increase array size by ***multiplying by two***.

What's the average work done with each push?

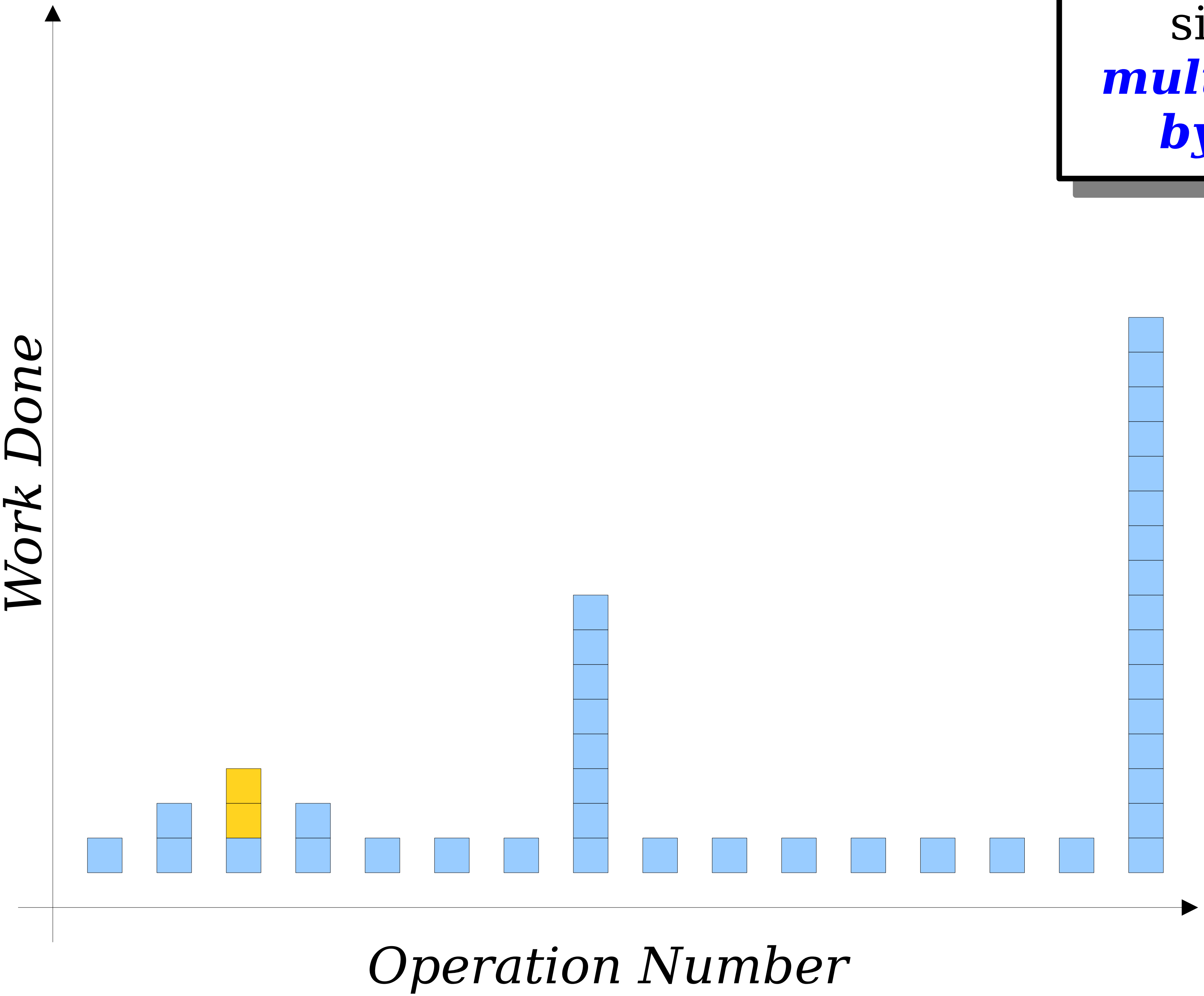
To find out, let's see how much total work was done.



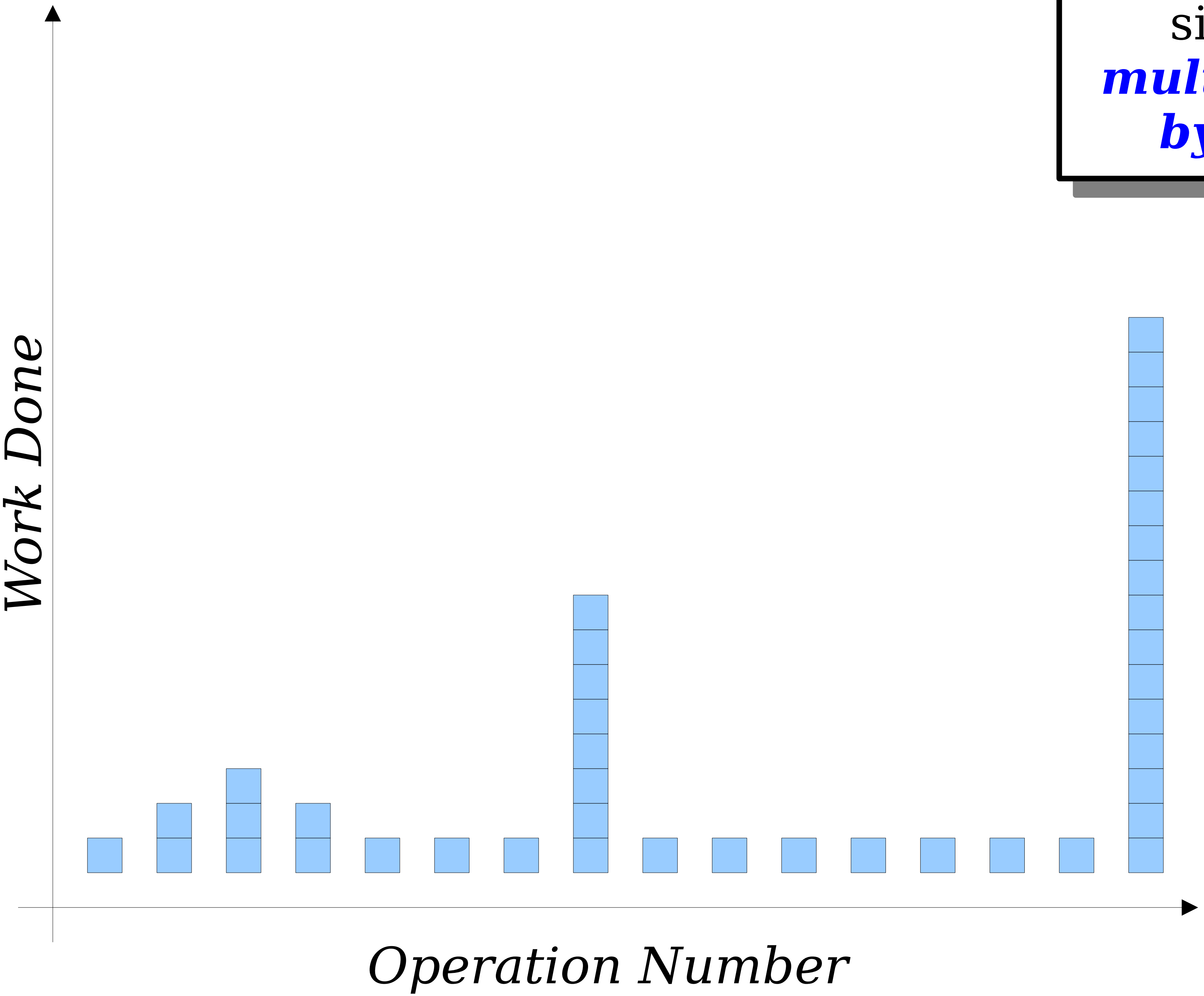
Increase array size by ***multiplying by two***.



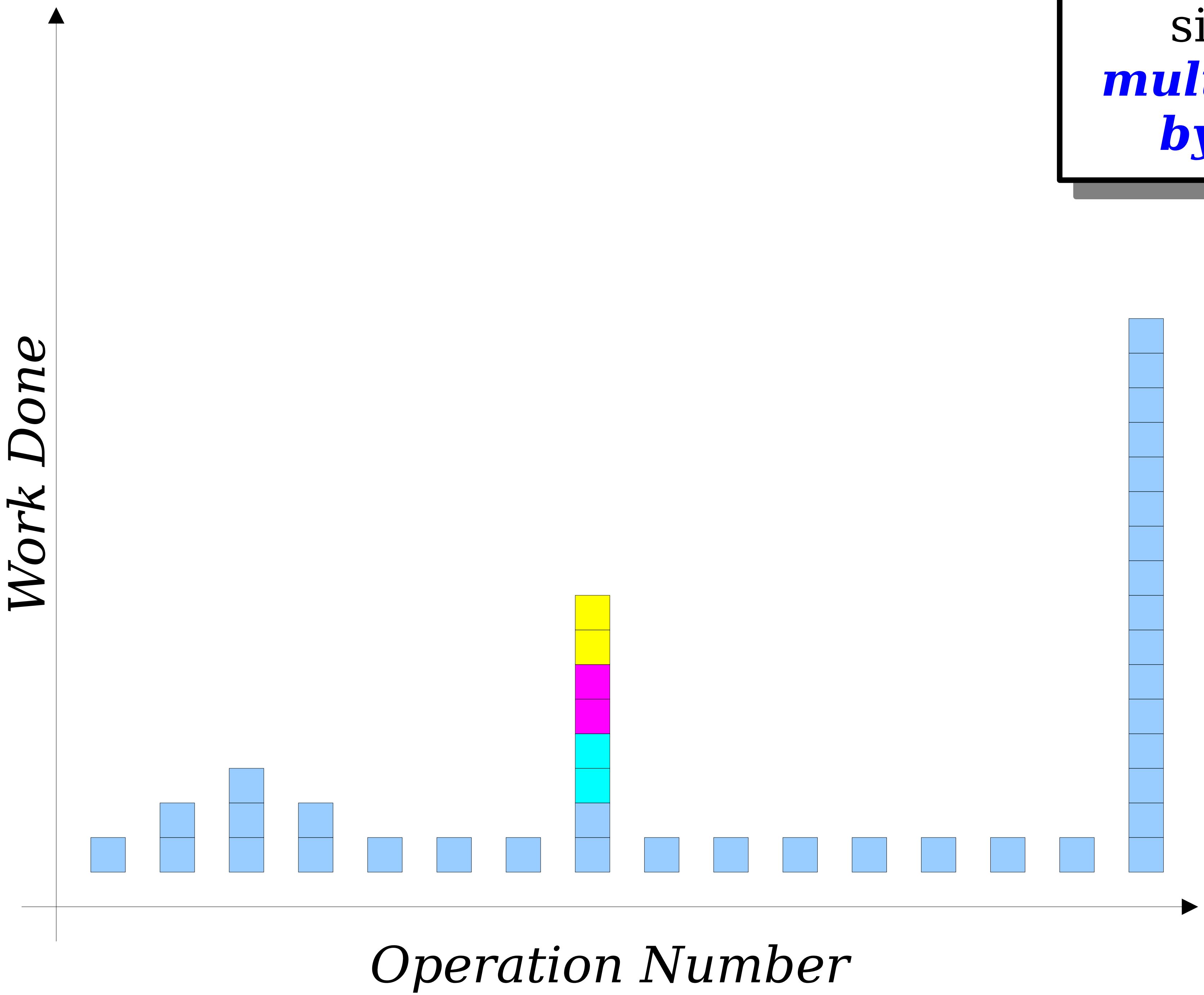
Increase array size by ***multiplying by two***.



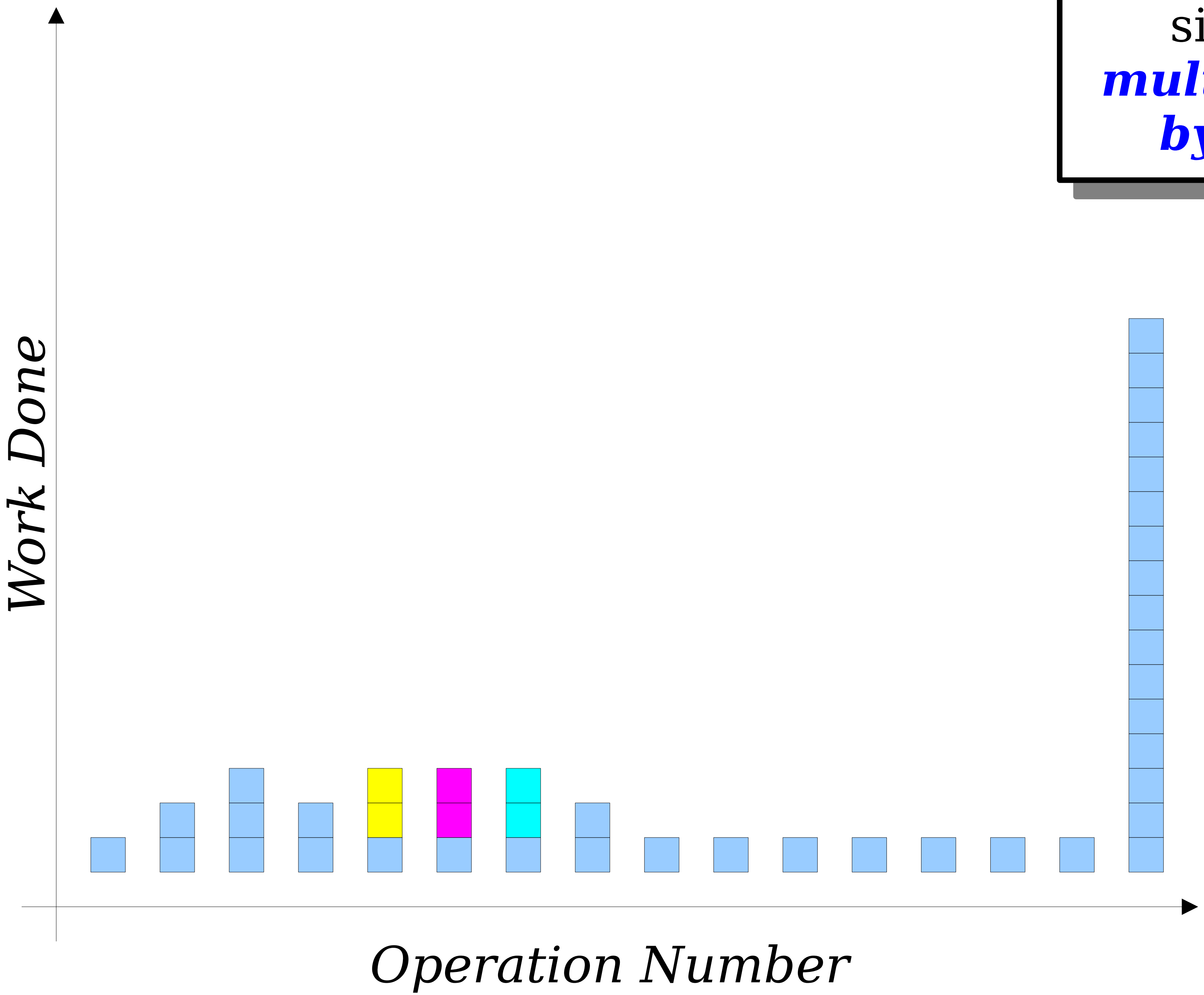
Increase array size by ***multiplying by two***.



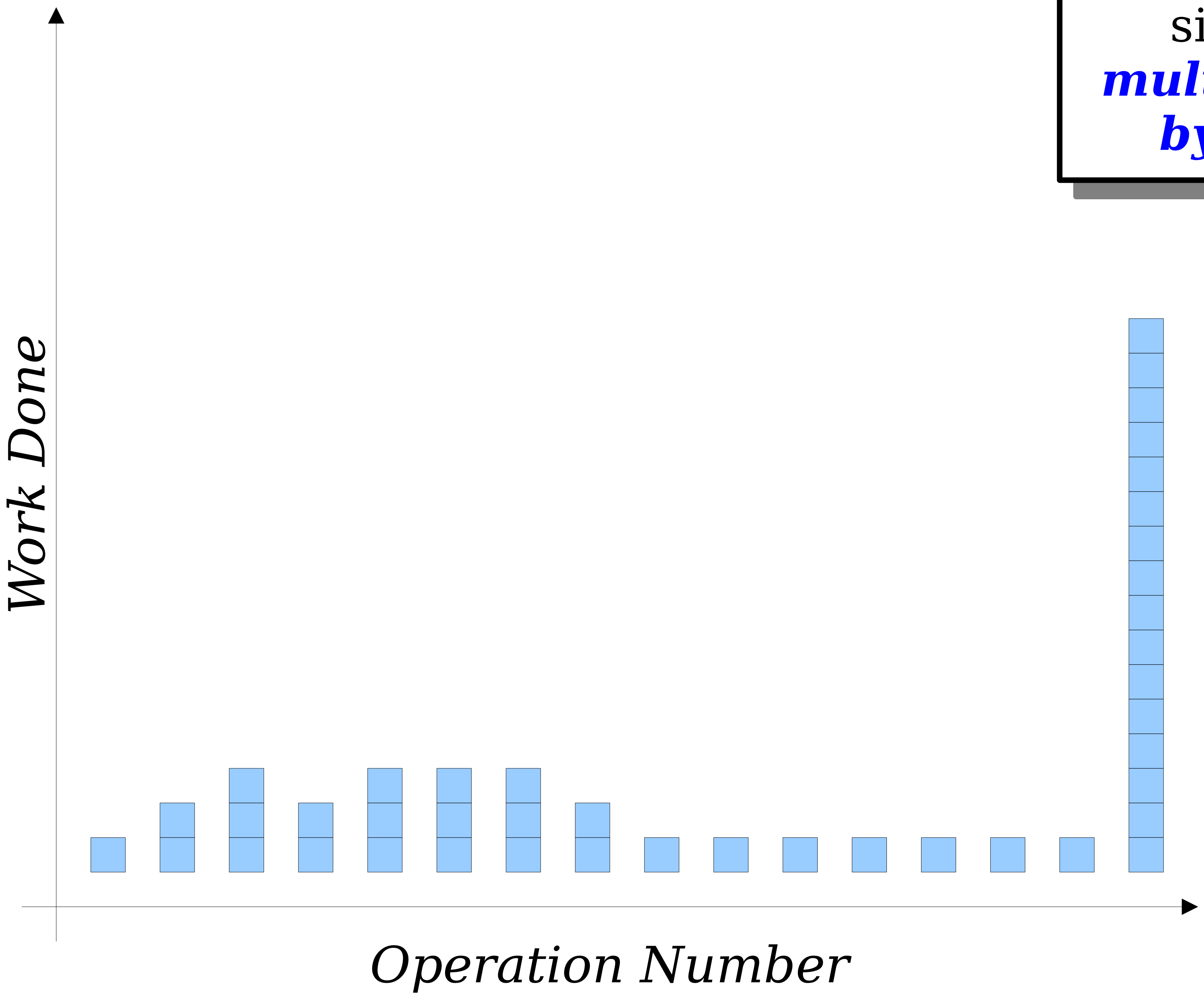
Increase array size by ***multiplying by two***.

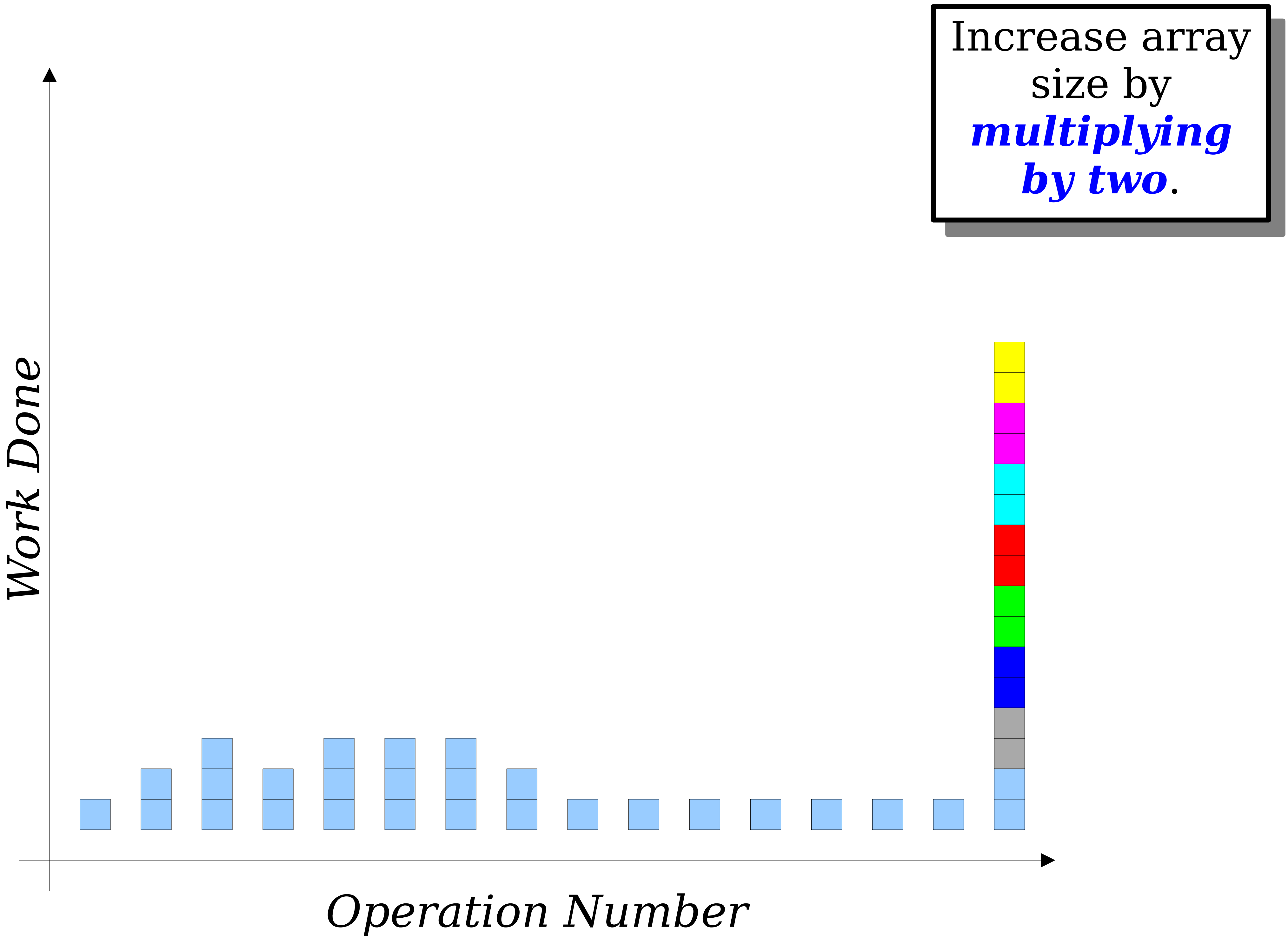


Increase array size by ***multiplying by two***.

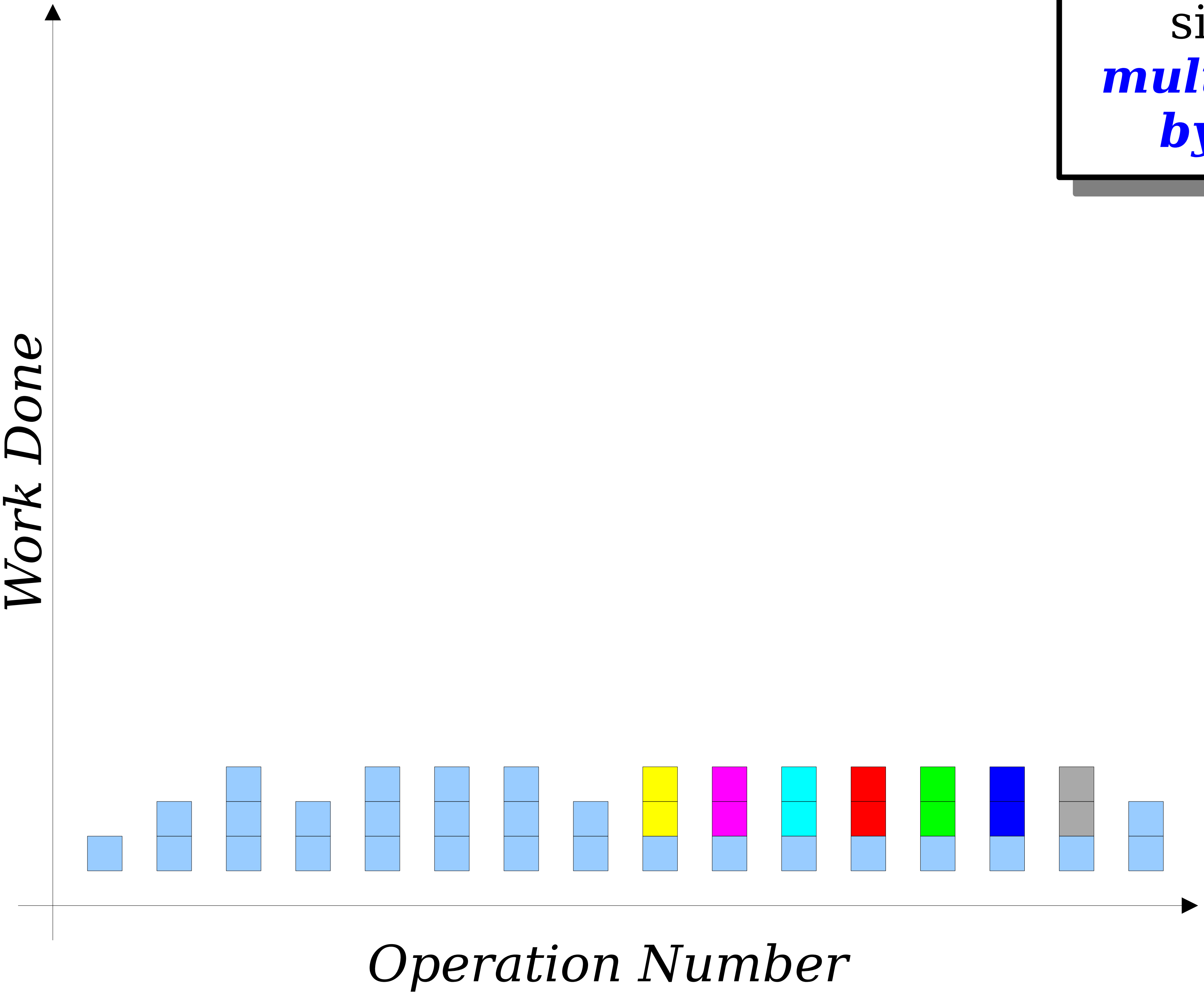


Increase array size by ***multiplying by two***.

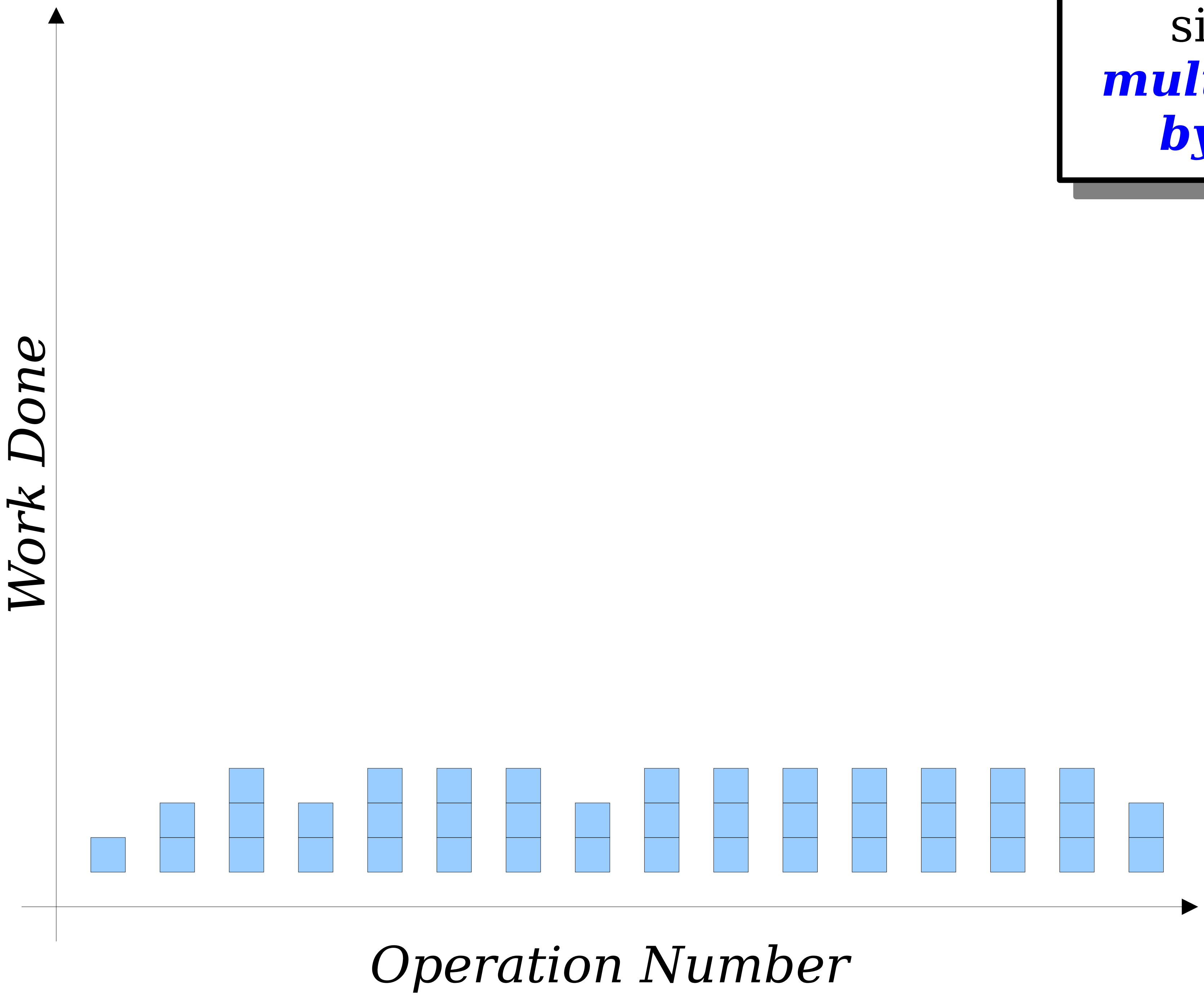



Increase array size by ***multiplying by two***.
Work Done
Operation Number

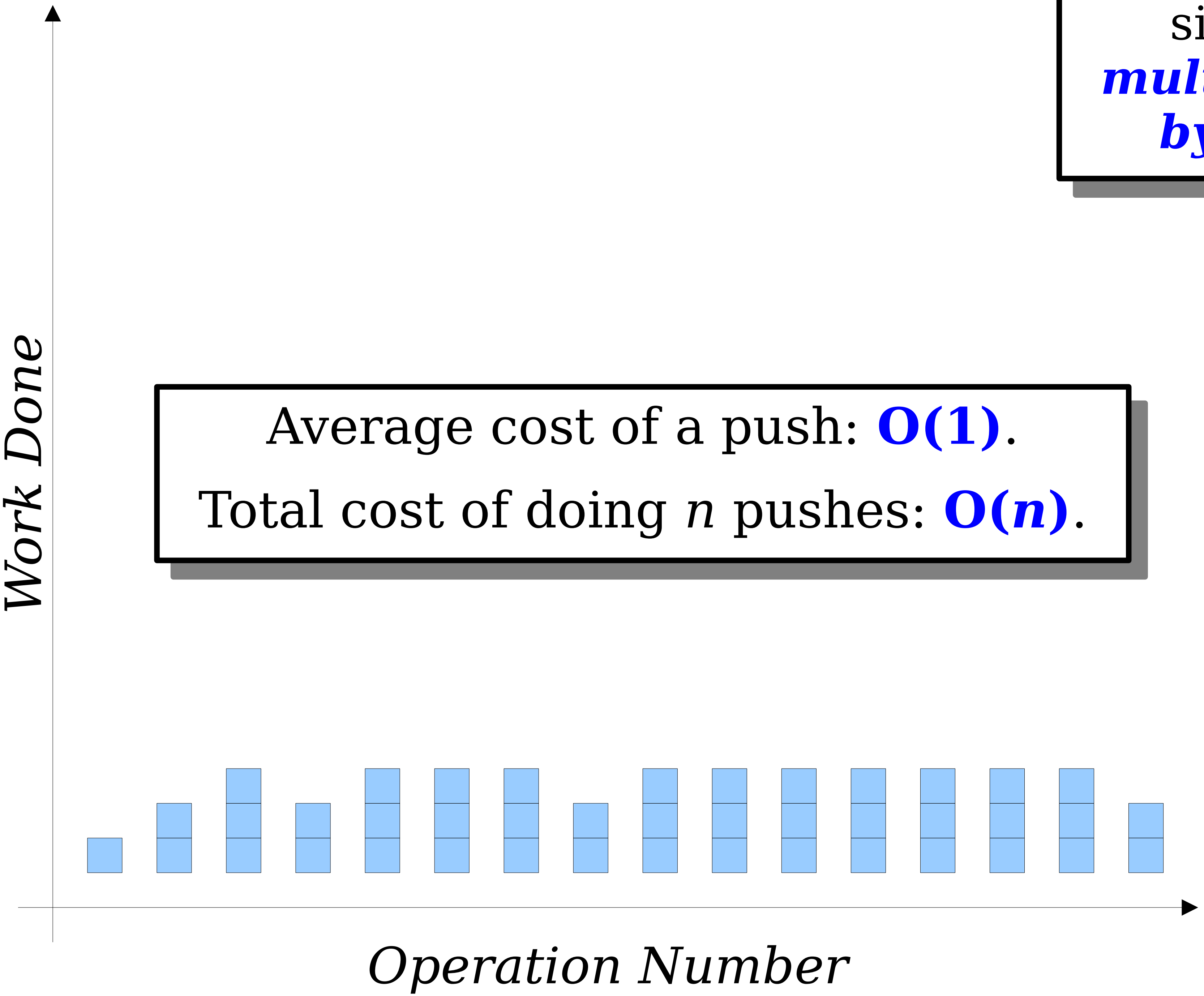
Increase array size by ***multiplying by two***.



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



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

- ***Start Assignment 6.***
  - Slow and steady progress is the name of the game here.
  - Ask for help if you need it! That’s what we’re here for.
- ***Review your midterm feedback.***
  - Regrade requests are due on Monday, so make sure you’ve read over the graders’ comments by then.

Next Time

- ***Hash Functions***
  - A magical and wonderful gift from the world of mathematics.
- ***Hash Tables***
  - How do we implement `Map` and `Set`?