Implementing Abstractions

Part Two

Previously, on CS106B...

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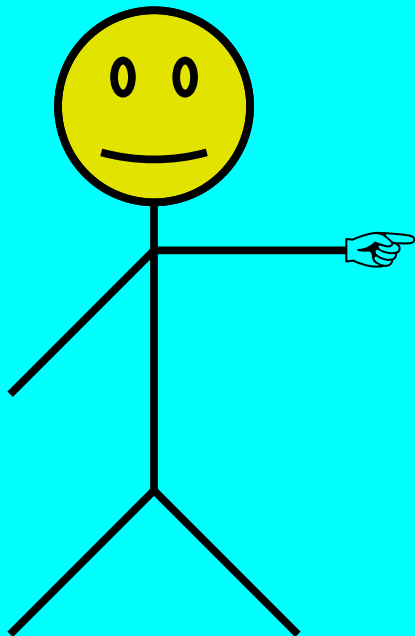
```cpp
class OurStack {
public:
    OurStack();

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

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

private:
    int* elems;
    int  allocatedSize;
    int  logicalSize;
};
```
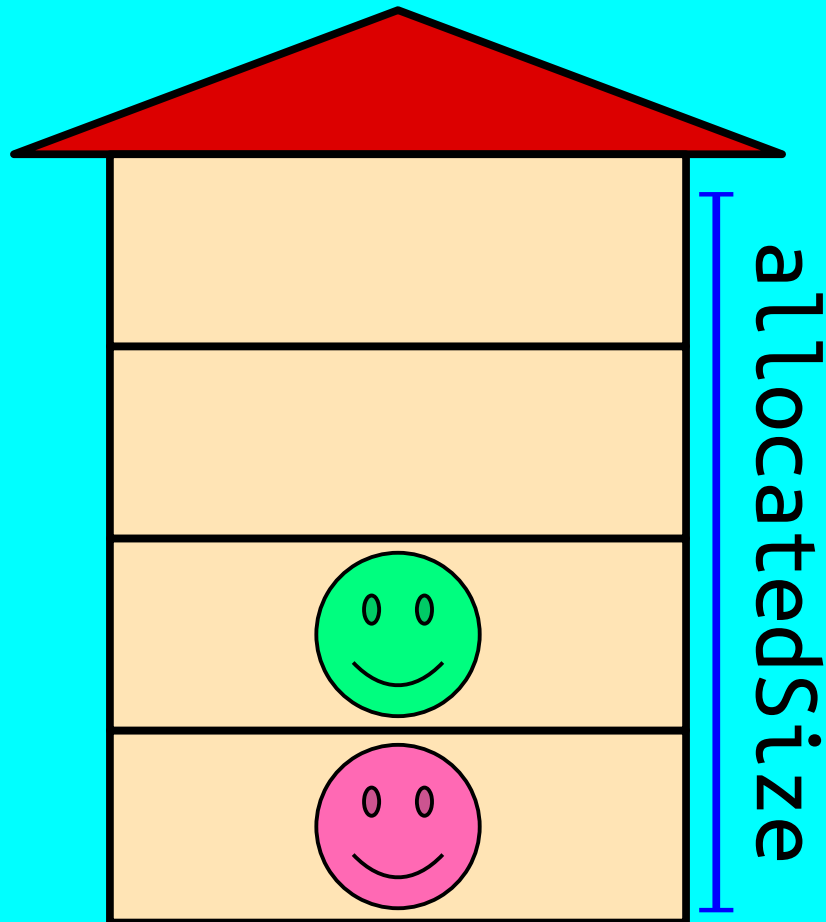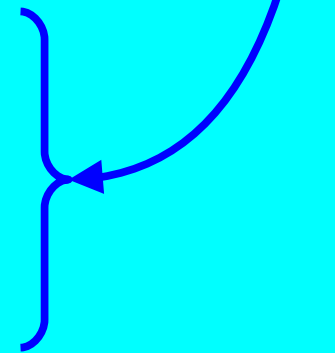
```cpp
class OurStack {
public:
    OurStack();

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

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

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

```cpp
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;
};
```

```cpp
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;
};
```

Cradle to Grave

```c
int main() {
    OurStack stack;

    /* The stack lives a rich, happy,
     * fulfilling life, the kind we
     * all aspire to.
     */

    return 0;
}
```

Cradle to Grave

```cpp
int main() {
    OurStack stack;

    /* The stack lives a rich, happy,
     * fulfilling life, the kind we
     * all aspire to.
     */

    return 0;
}
```

Cradle to Grave

element
array

allocated
size

???

logical
size

???

```c
int main() {
    OurStack stack;

    /* The stack lives a rich, happy,
     * fulfilling life, the kind we
     * all aspire to.
     */

    return 0;
}
```

Cradle to Grave

element
array

allocated
size

???

logical
size

???

```
int
    OurStack::OurStack() {
        logicalSize = 0;
        allocatedSize = kInitialSize;
        elems = new int[allocatedSize];
    }

}
```

Cradle to Grave

element array

allocated size `4`

logical size `0`

```
int
OurStack::OurStack() {
    logicalSize = 0;
    allocatedSize = kInitialSize;
    elems = new int[allocatedSize];
}

}
```

Cradle to Grave

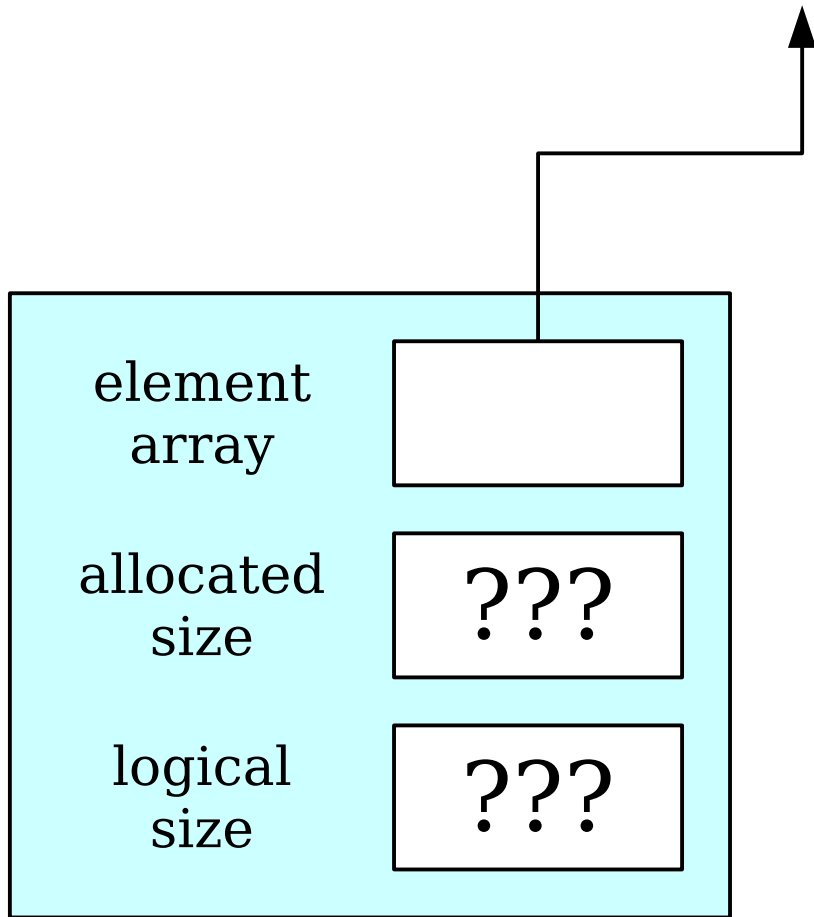element array

allocated size: 4
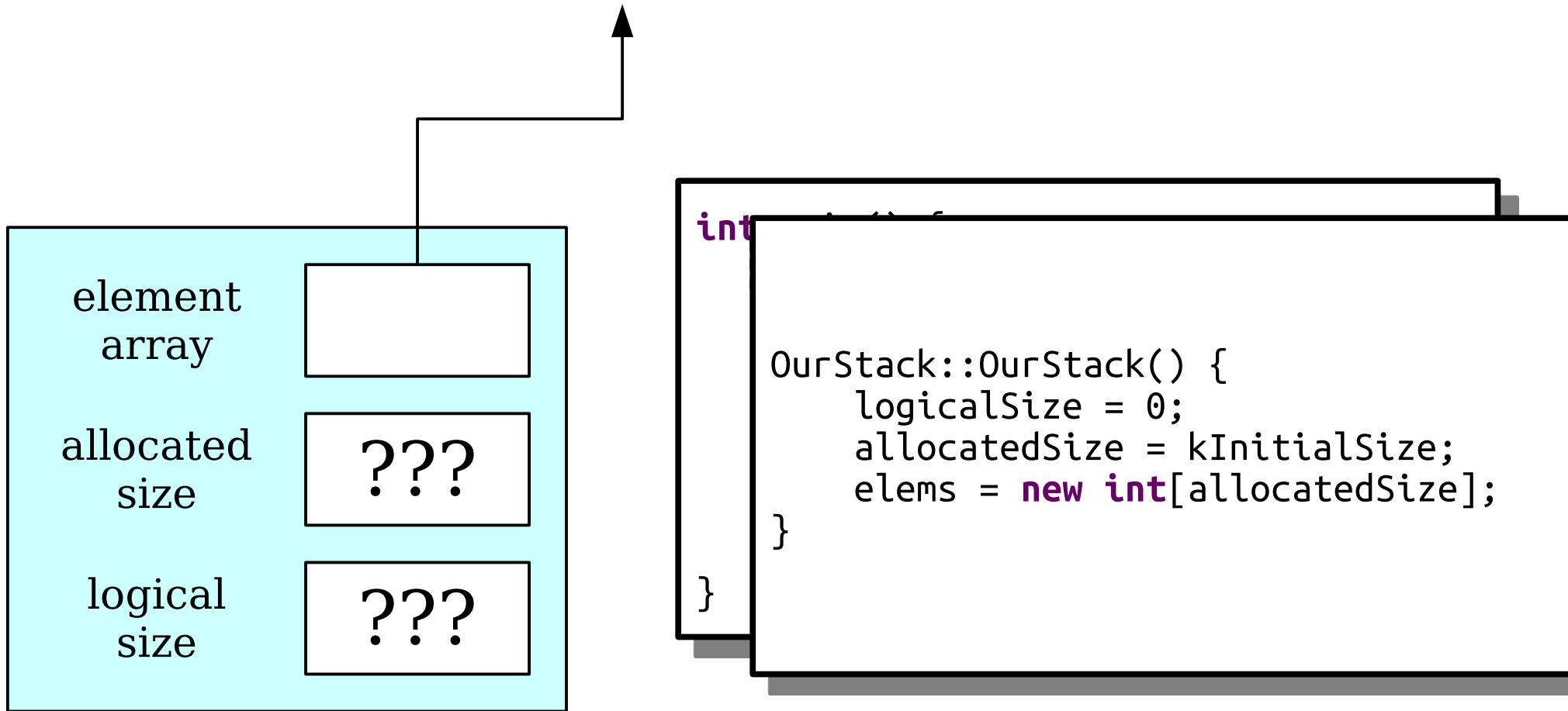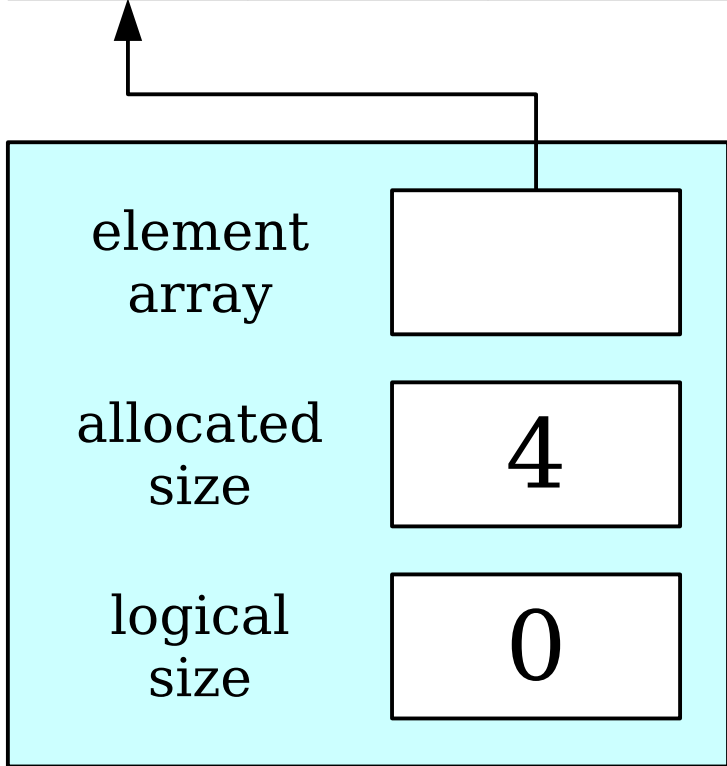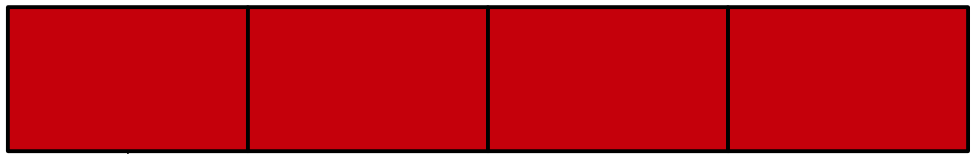
logical size: 0

```c
int main() {
    OurStack stack;

    /* The stack lives a rich, happy,
     * fulfilling life, the kind we
     * all aspire to.
     */

    return 0;
}
```

Cradle to Grave

element array

allocated size: 4

logical size: 0

```c
int main() {
    OurStack stack;
    /* The stack lives a rich, happy,
     * fulfilling life, the kind we
     * all aspire to.
     */

    return 0;
}
```
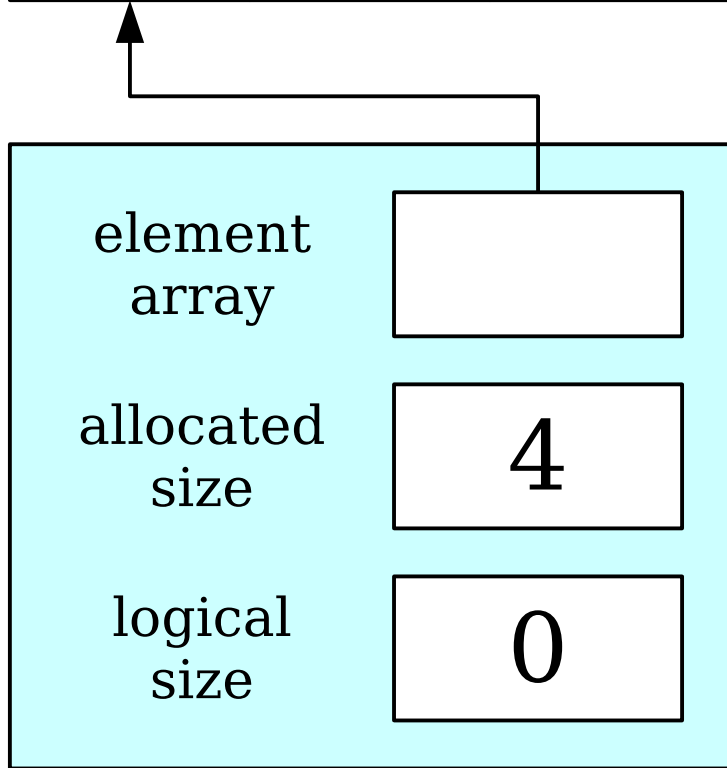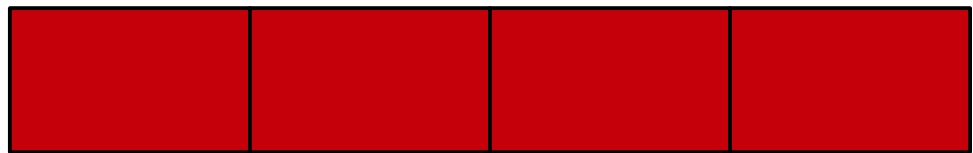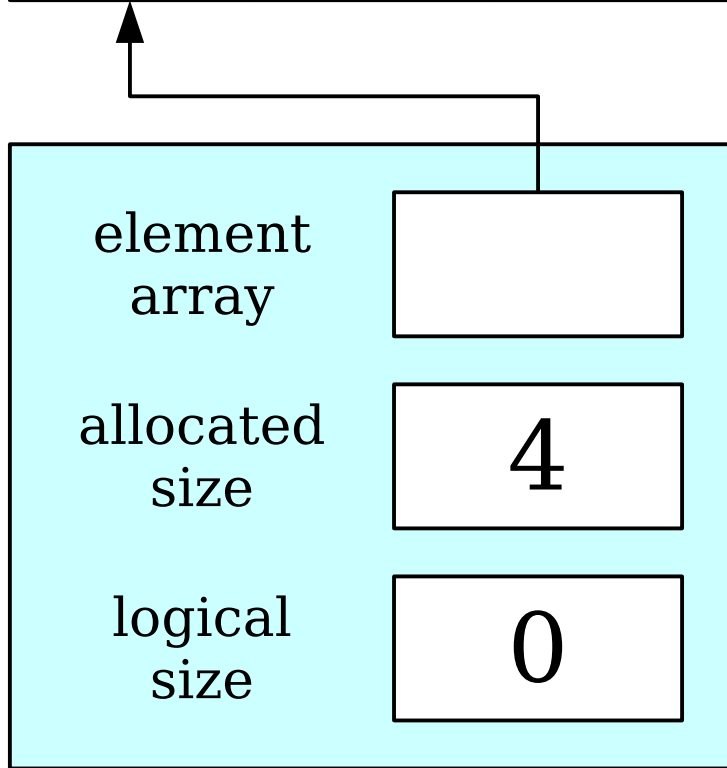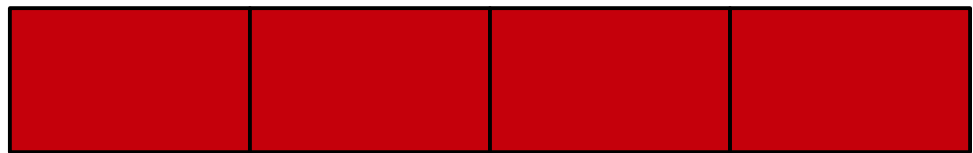
Cradle to Grave

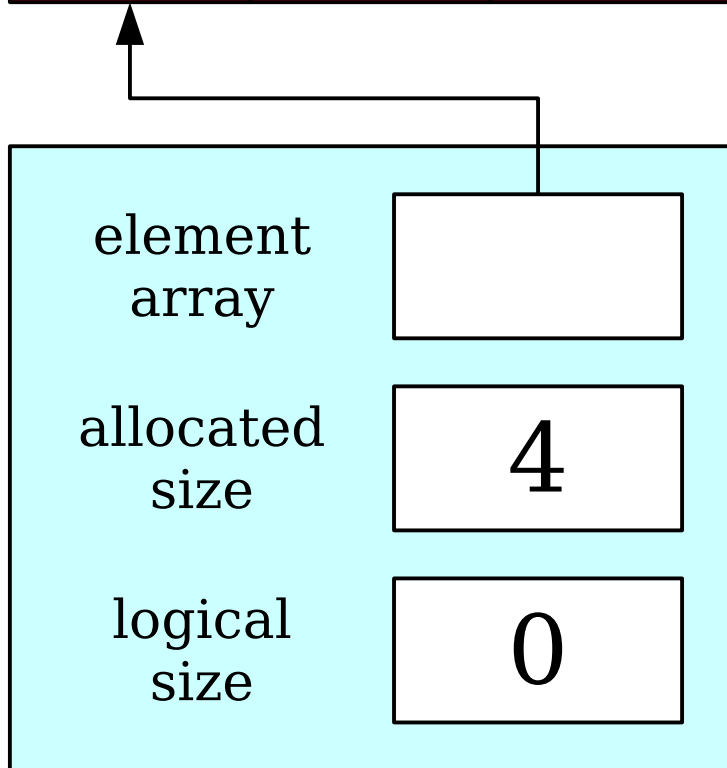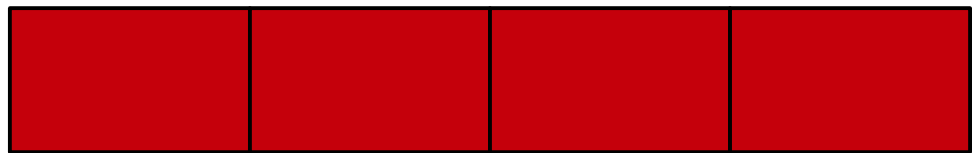```
int main() {
    OurStack stack;

    /* The stack lives a rich, happy,
     * fulfilling life, the kind we
     * all aspire to.
     */
    return 0;
}
```

element array

allocated size: 4

logical size: 0
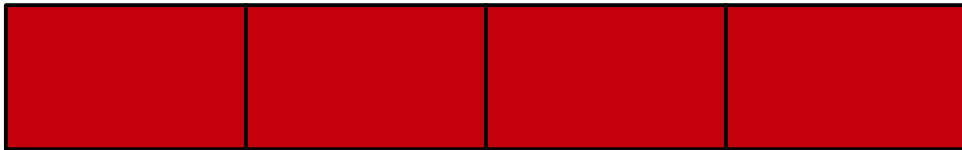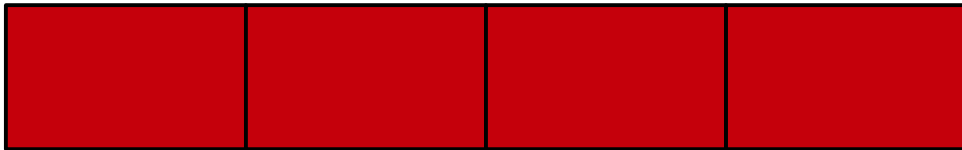
# Cradle to Grave

```
int main() {
    OurStack stack;

    /* The stack lives a rich, happy,
     * fulfilling life, the kind we
     * all aspire to.
     */
    return 0;
}
```

Cradle to Grave

Memory Leak!

```c
int main() {
    OurStack stack;

    /* The stack lives a rich, happy,
     * fulfilling life, the kind we
     * all aspire to.
     */
    return 0;
}
```

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();

```cpp
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;
};
```
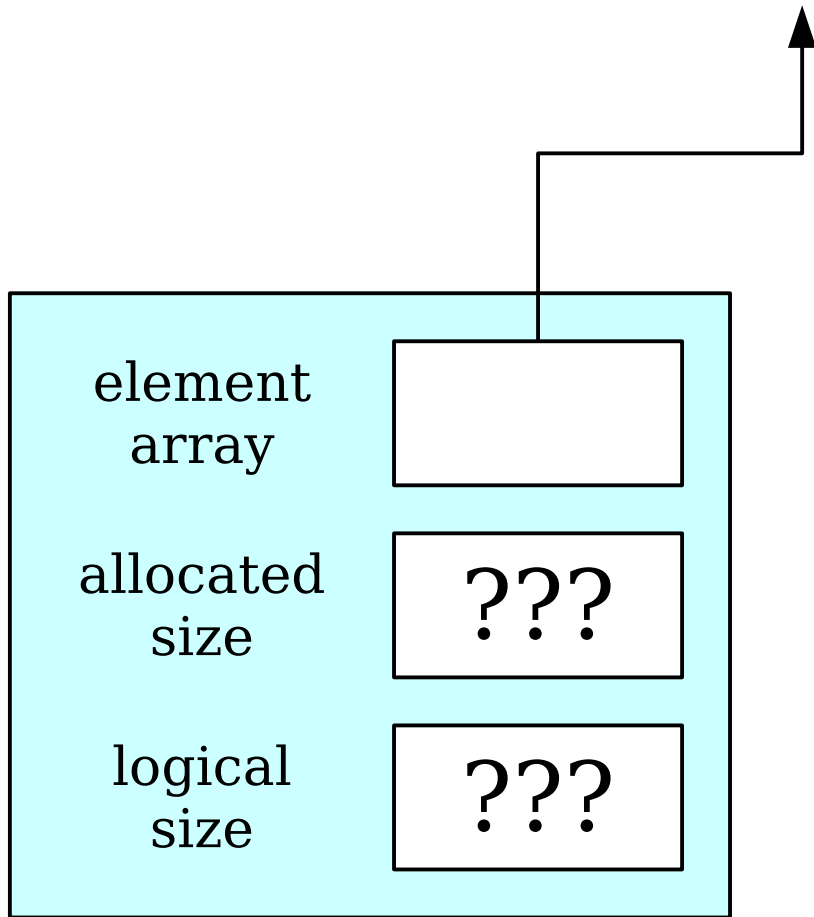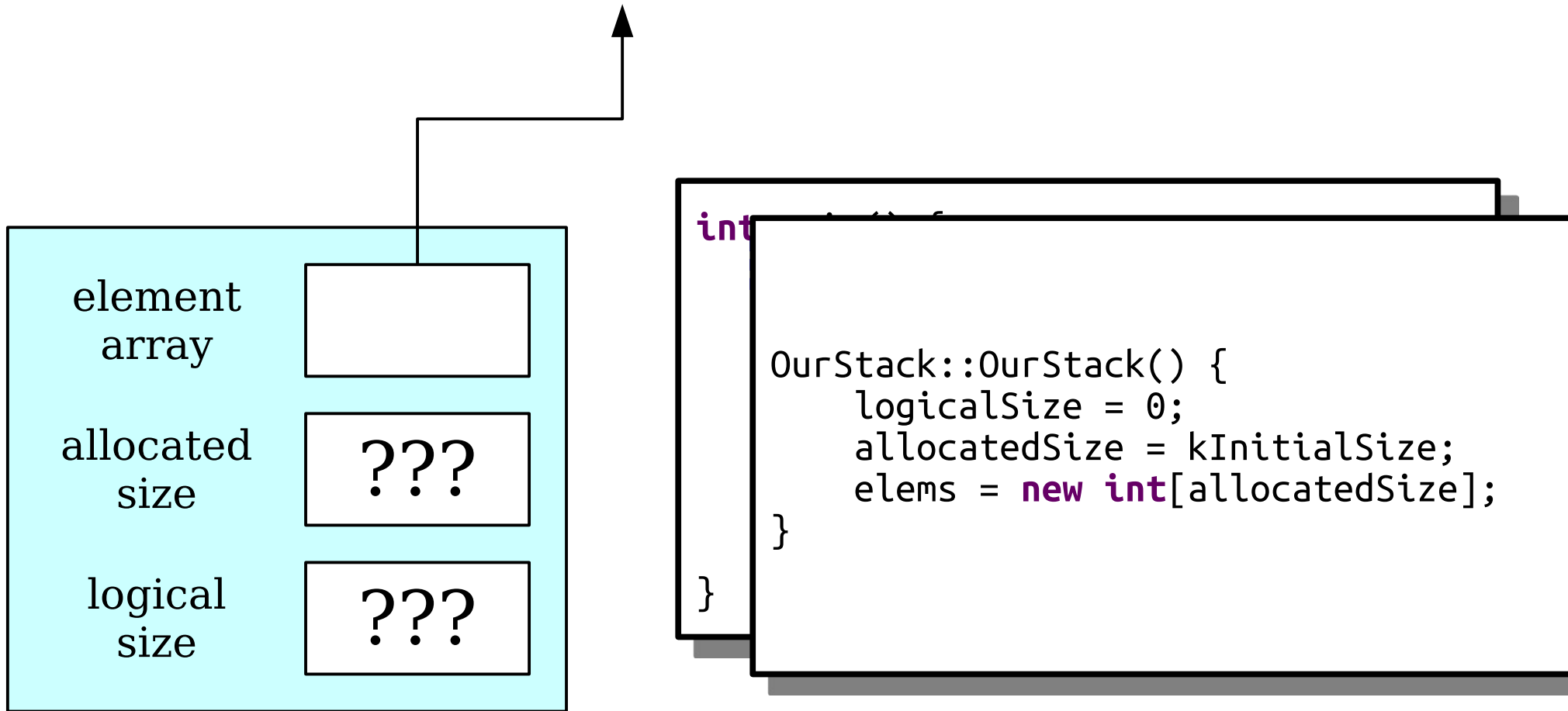
Cradle to Grave

```cpp
int main() {
    OurStack stack;

    /* The stack lives a rich, happy,
     * fulfilling life, the kind we
     * all aspire to.
     */

    return 0;
}
```

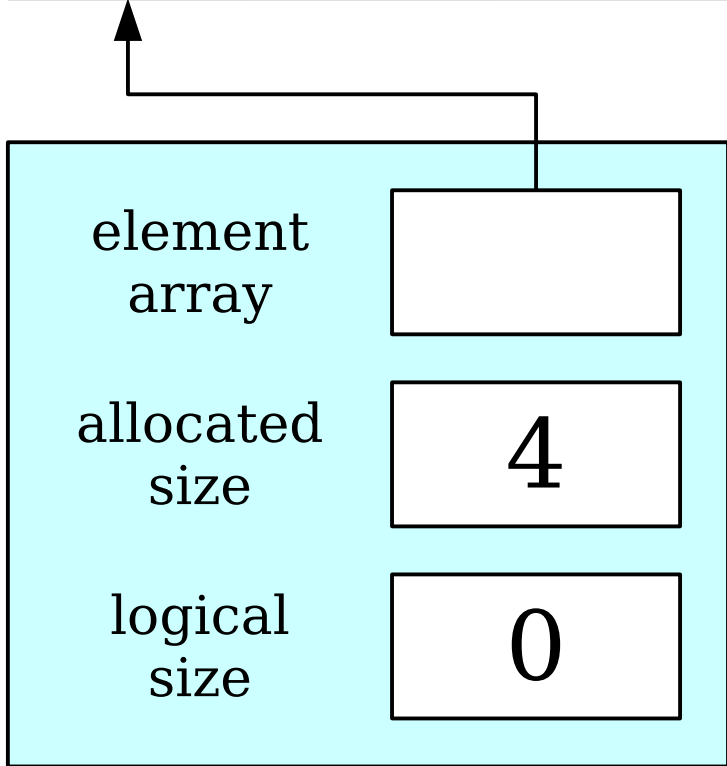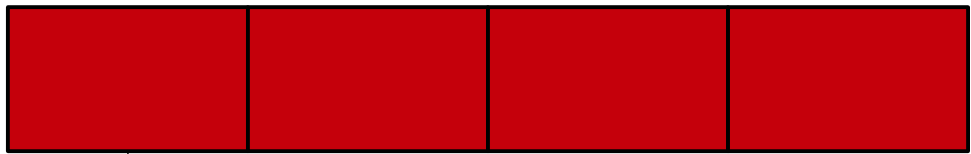# Cradle to Grave

```
int main() {
    OurStack stack;

    /* The stack lives a rich, happy,
     * fulfilling life, the kind we
     * all aspire to.
     */

    return 0;
}
```
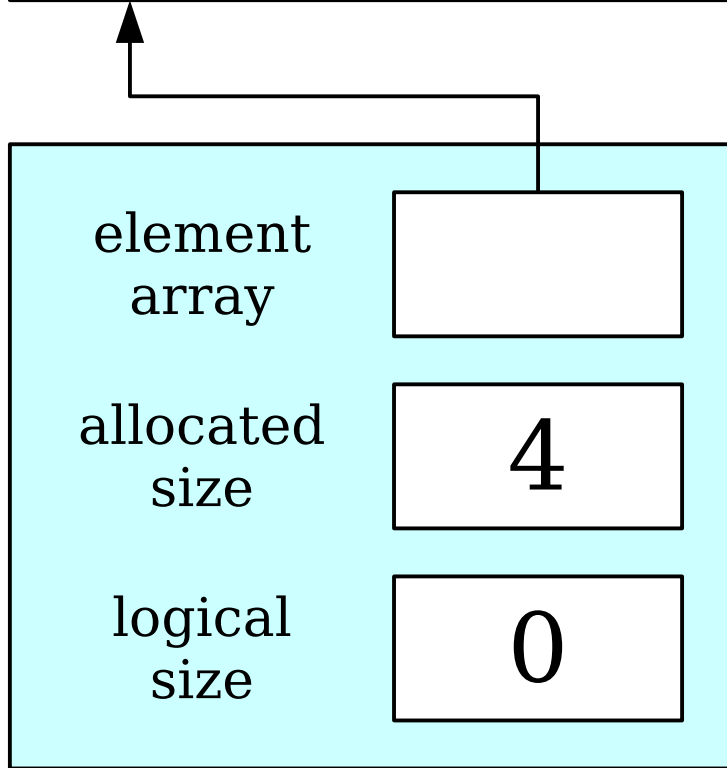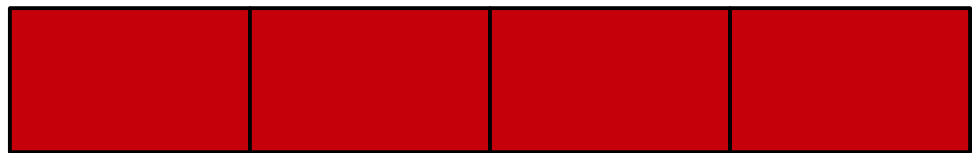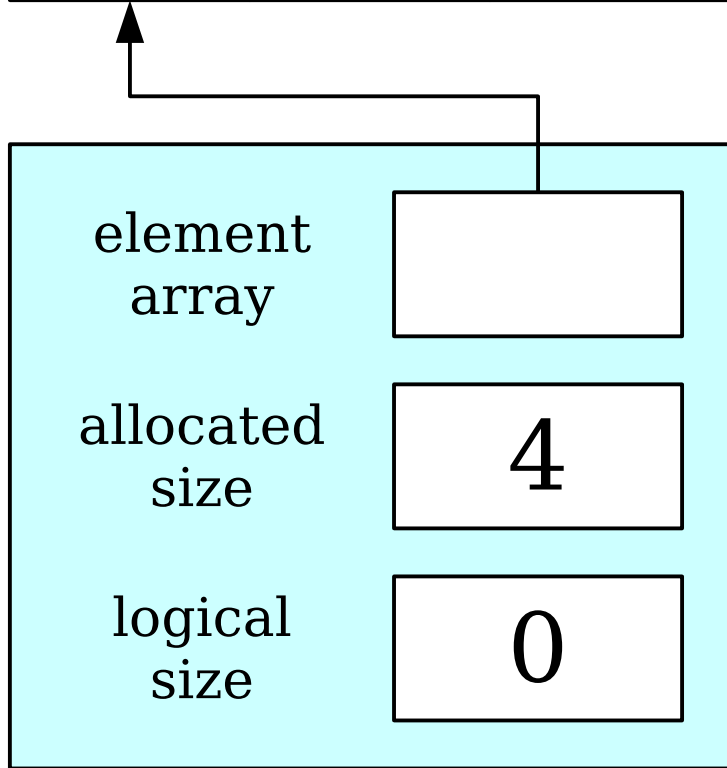
Cradle to Grave

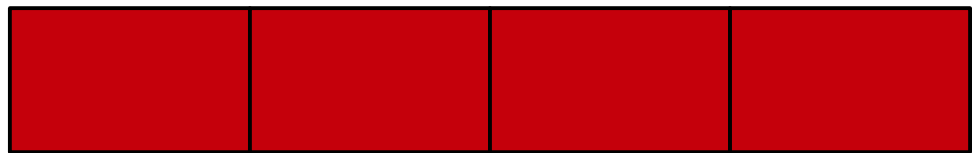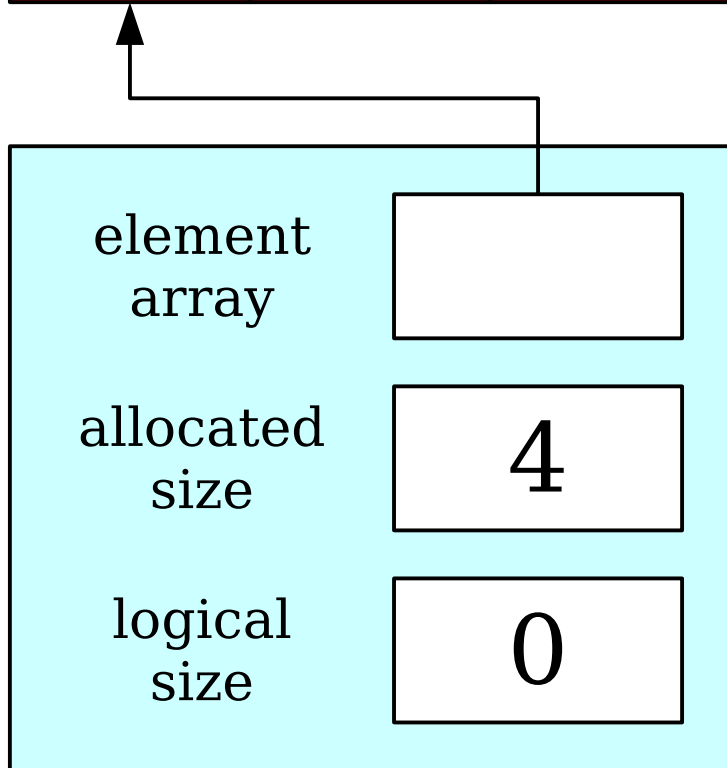element
array

allocated
size
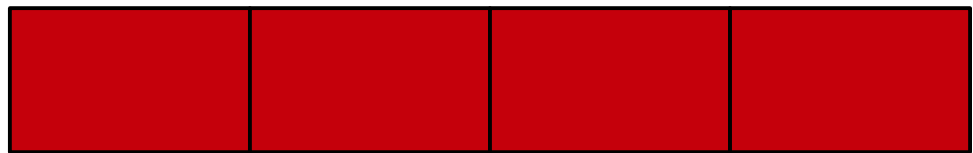
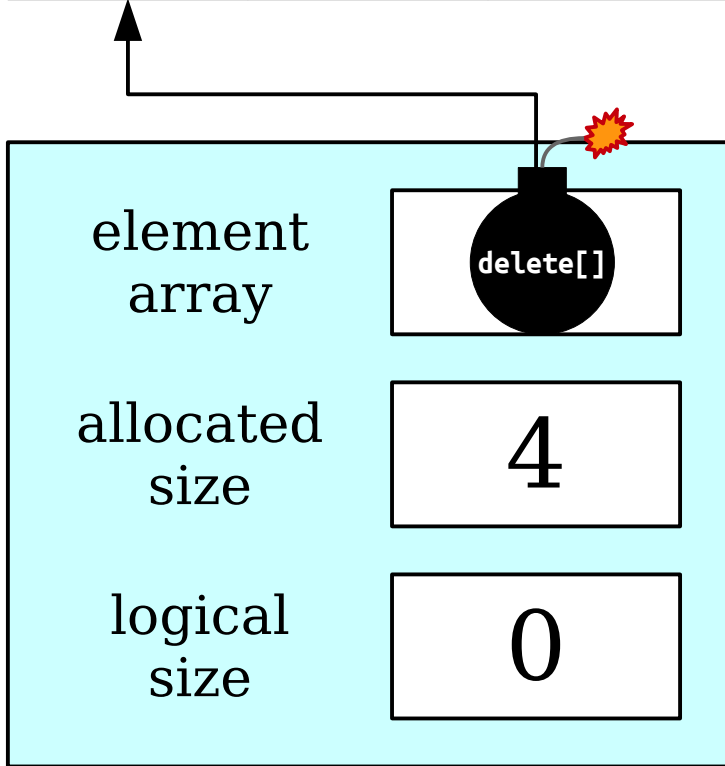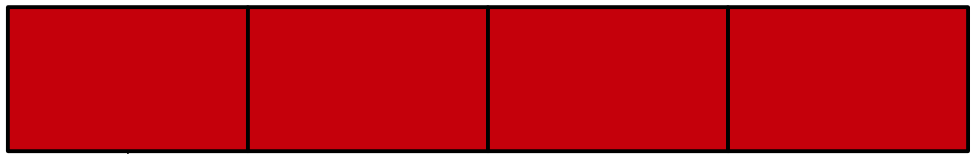???

logical
size

???

```
int main() {
    OurStack stack;

    /* The stack lives a rich, happy,
     * fulfilling life, the kind we
     * all aspire to.
     */

    return 0;
}
```

Cradle to Grave

element array

allocated size ???

logical size ???

```
int
OurStack::OurStack() {
    logicalSize = 0;
    allocatedSize = kInitialSize;
    elems = new int[allocatedSize];
}

}
```

Cradle to Grave

element array

allocated size: 4

logical size: 0

```cpp
int main() {

OurStack::OurStack() {
    logicalSize = 0;
    allocatedSize = kInitialSize;
    elems = new int[allocatedSize];
}

}
```

Cradle to Grave

| element array	
allocated size	4
logical size	0

```c
int main() {
    OurStack stack;

    /* The stack lives a rich, happy,
     * fulfilling life, the kind we
     * all aspire to.
     */

    return 0;
}
```
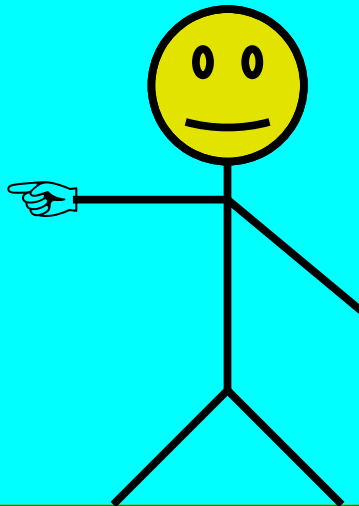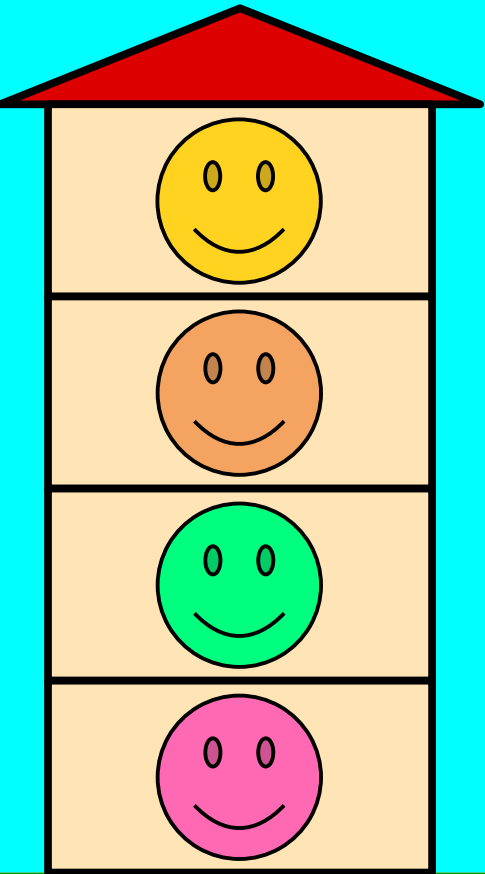
Cradle to Grave

element array

allocated size **4**

logical size **0**

```
int main() {
    OurStack stack;
    /* The stack lives a rich, happy,
     * fulfilling life, the kind we
     * all aspire to.
     */

    return 0;
}
```
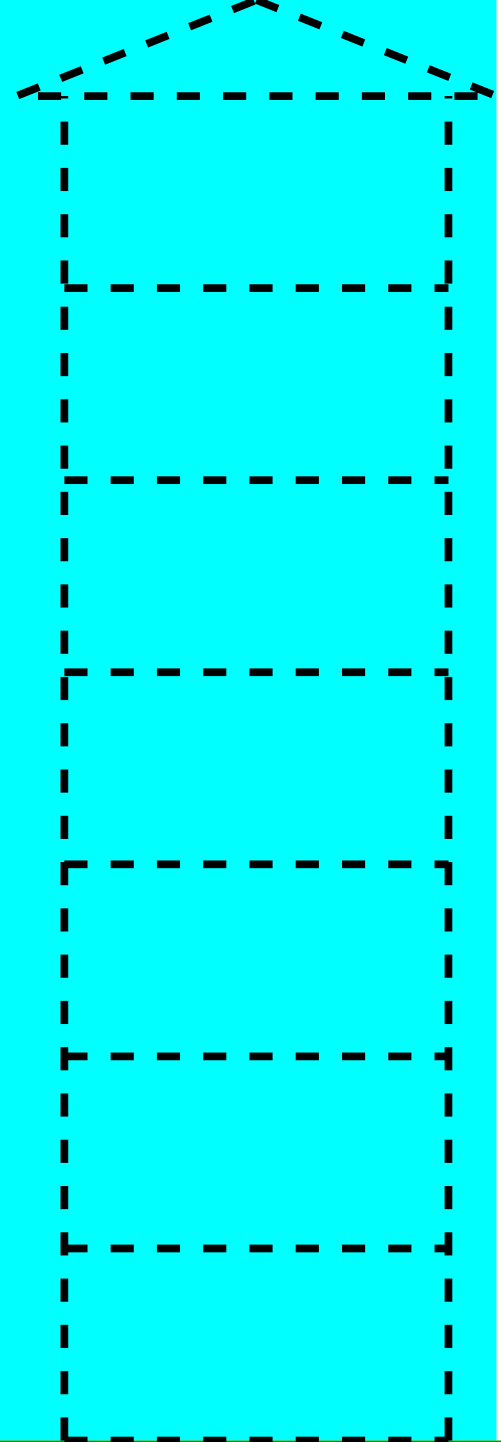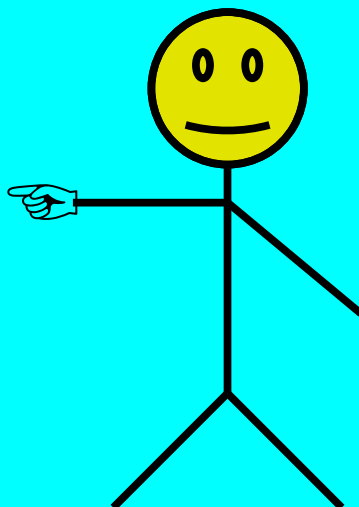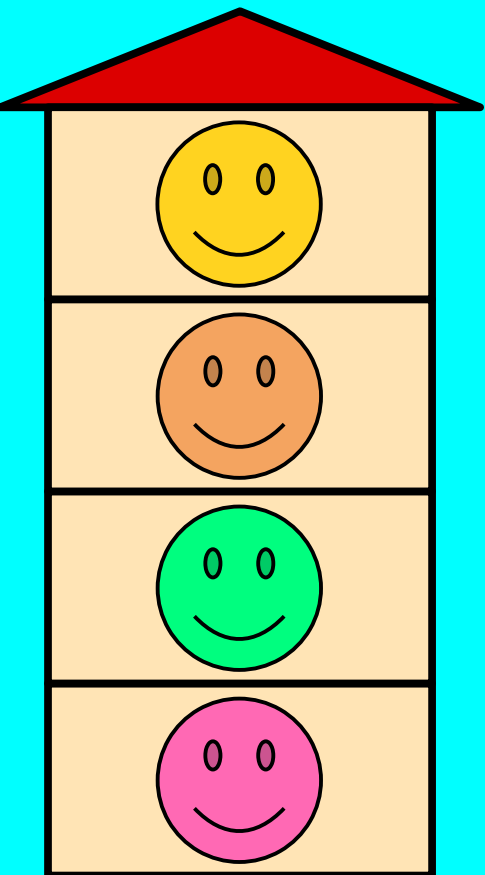
Cradle to Grave

element
array

allocated
size

4

logical
size

0

```c
int main() {
    OurStack stack;

    /* The stack lives a rich, happy,
     * fulfilling life, the kind we
     * all aspire to.
     */
    return 0;
}
```

Cradle to Grave

element
array

delete[]

allocated
size

4

logical
size

0

```
int main() {

OurStack::~OurStack() {
    delete[] elems;
}

}
```

Cradle to Grave

Dynamic Deallocation!

element array	
allocated size	4
logical size	0

```cpp
int main() {

OurStack::~OurStack() {
    delete[] elems;
}

}
```

Cradle to Grave

element array

allocated size 4

logical size 0

```
int main() {


OurStack::~OurStack() {
    delete[] elems;
}


}
```

Cradle to Grave

element
array

allocated
size

4

logical
size

0

```
int main() {
    OurStack stack;

    /* The stack lives a rich, happy,
     * fulfilling life, the kind we
     * all aspire to.
     */
    return 0;
}
```
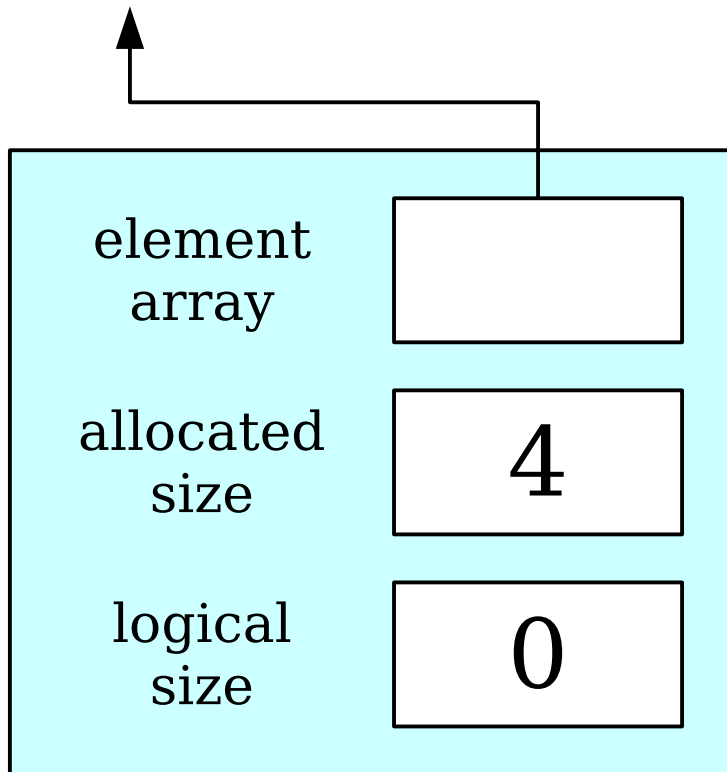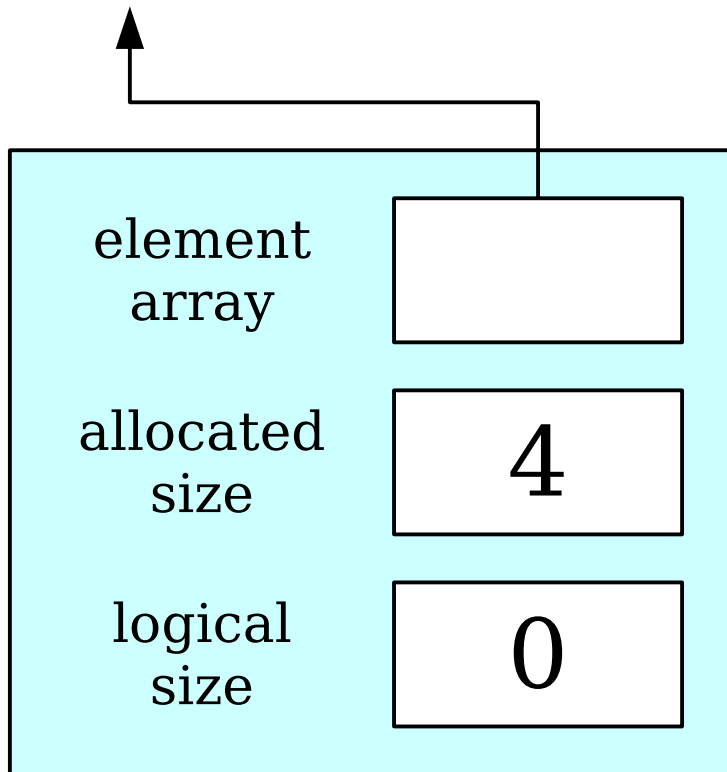
Cradle to Grave

```
int main() {
    OurStack stack;

    /* The stack lives a rich, happy,
     * fulfilling life, the kind we
     * all aspire to.
     */
    return 0;
}
```
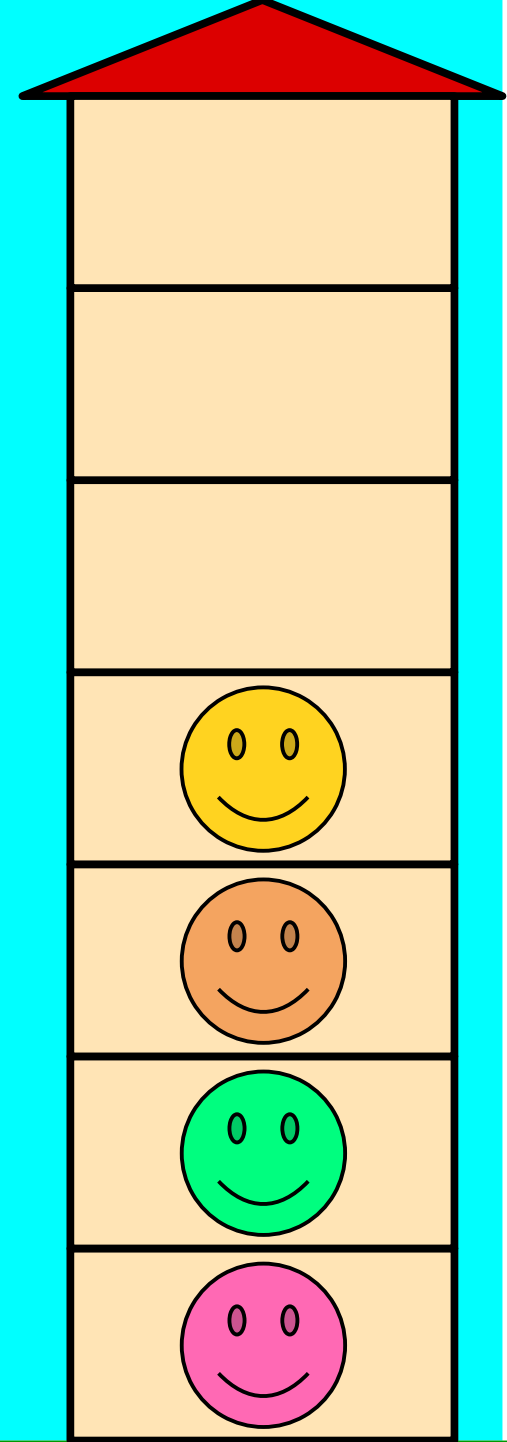
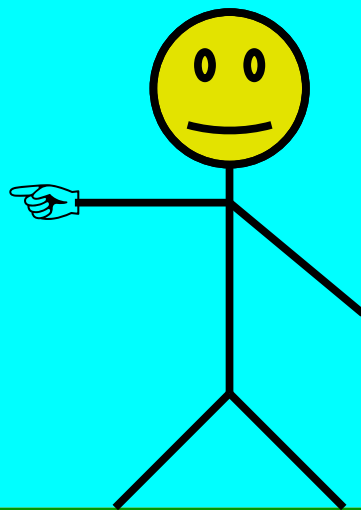# Getting More Space
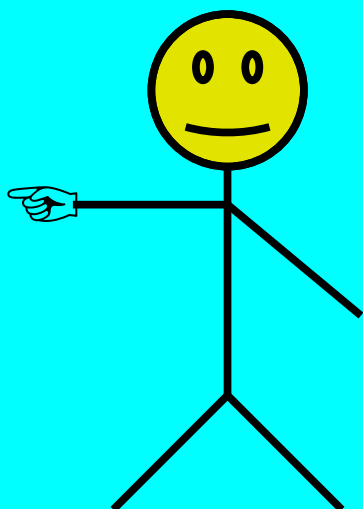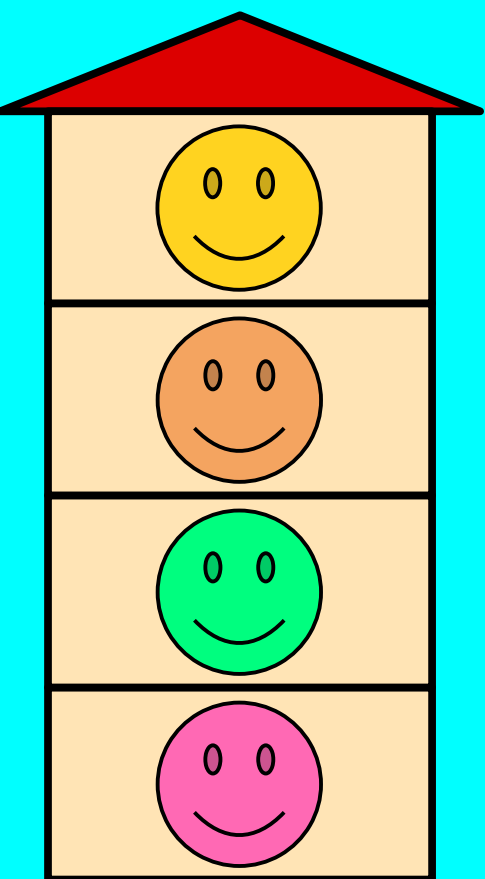
elems

```
allocatedSize = /* bigger */;
```

elems

```
allocatedSize = /* bigger */;
int* helper = new int[allocatedSize];
/* … move elements over … */
```
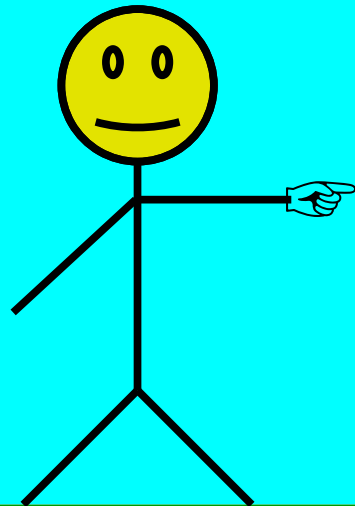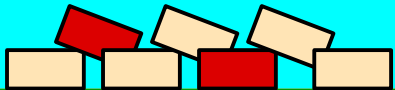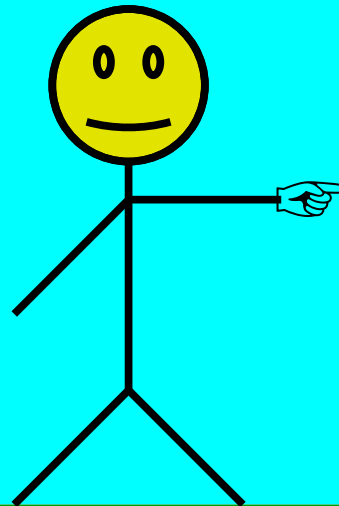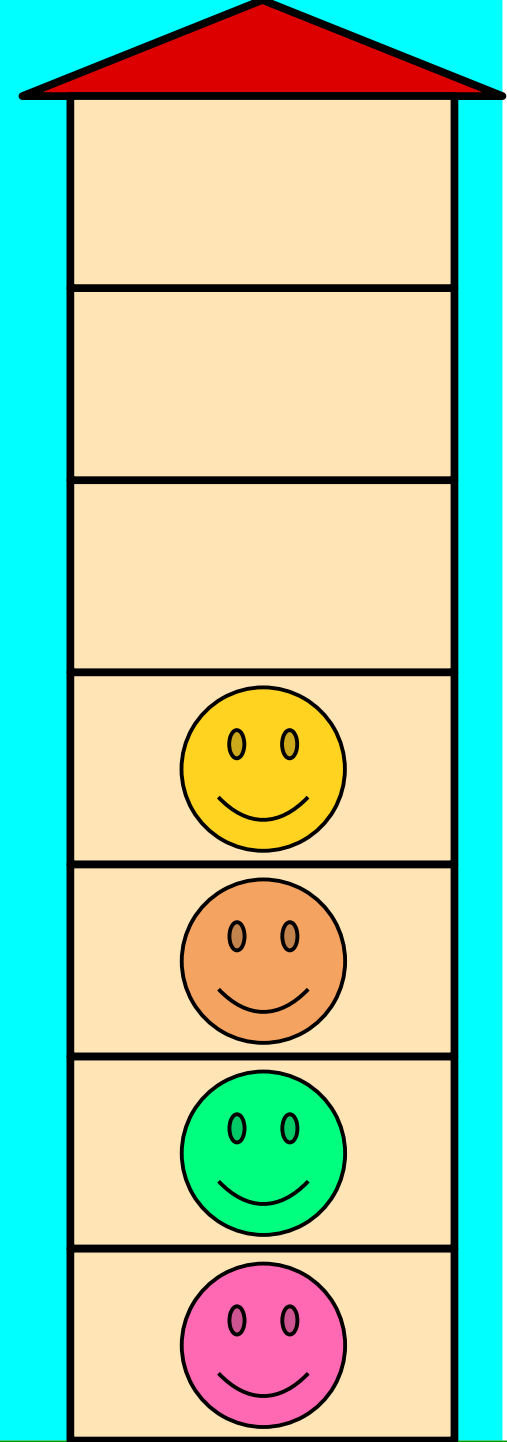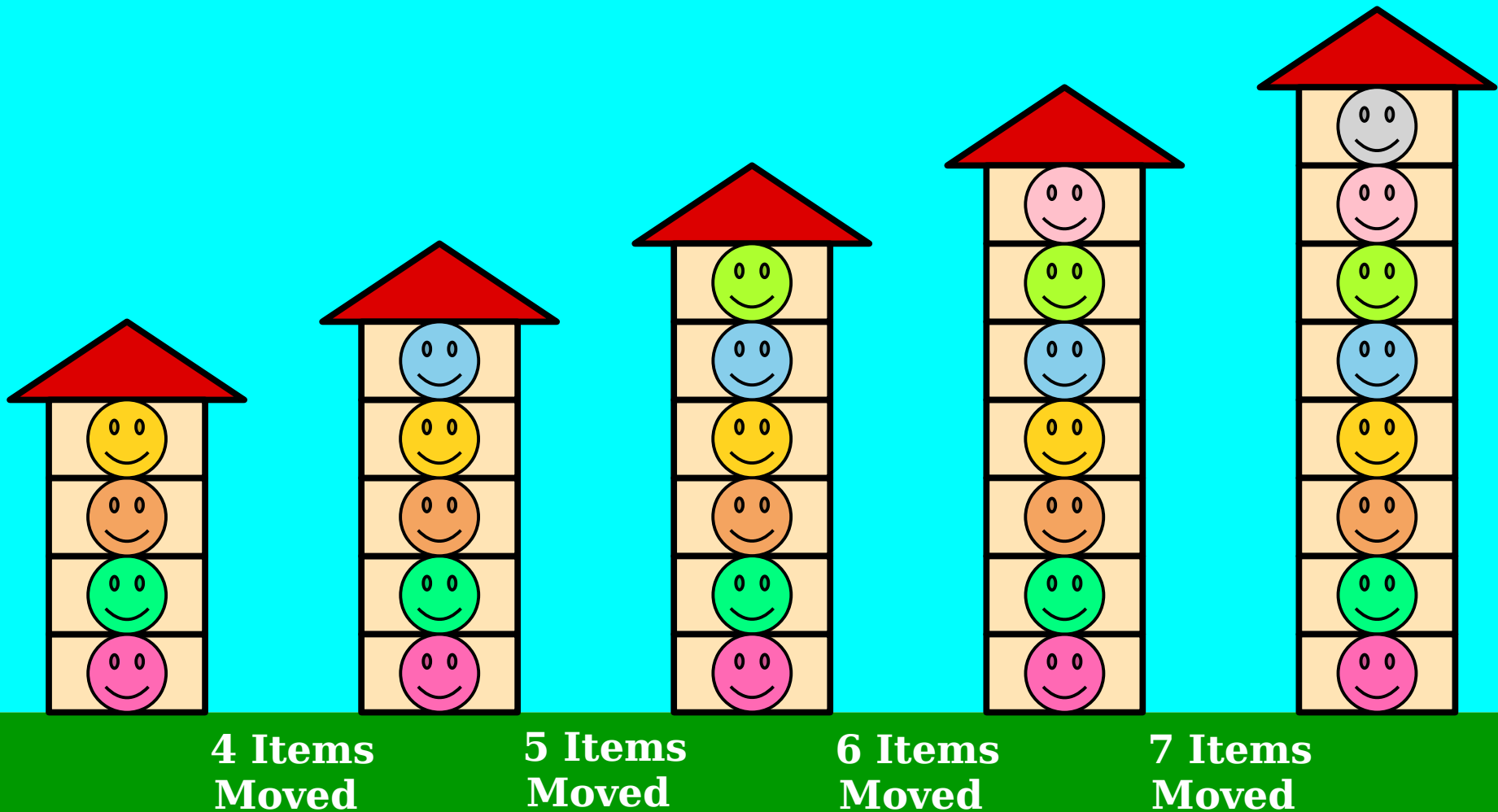
Every push beyond the first few requires moving all n elements from the old array to the new array.

Cost of a single push: $O(n)$.
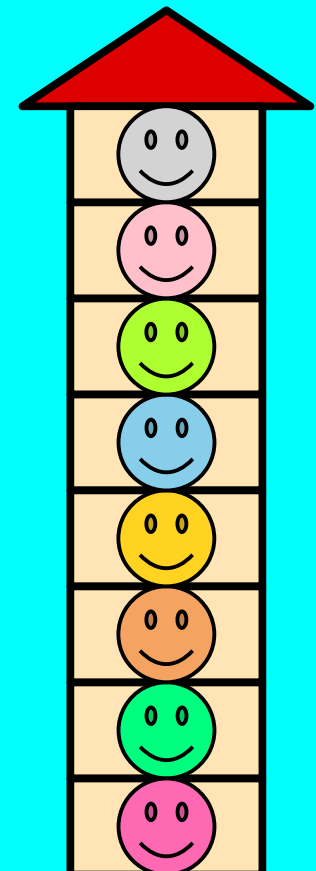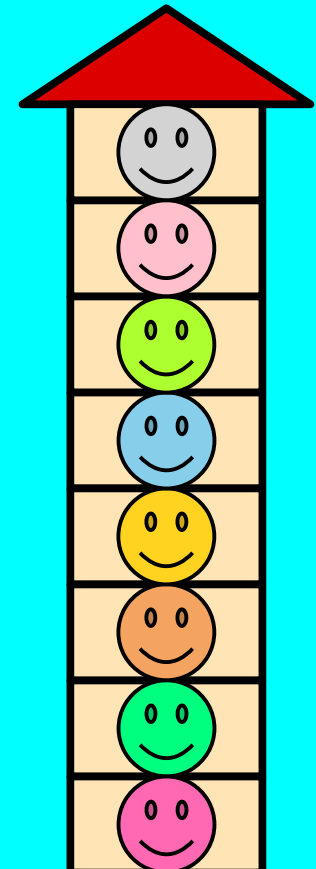
4 Items Moved

5 Items Moved

6 Items Moved

7 Items Moved

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

Cost of doing n pushes:
$4 + 5 + 6 + \ldots + n = \mathbf{O(n^2)}$.
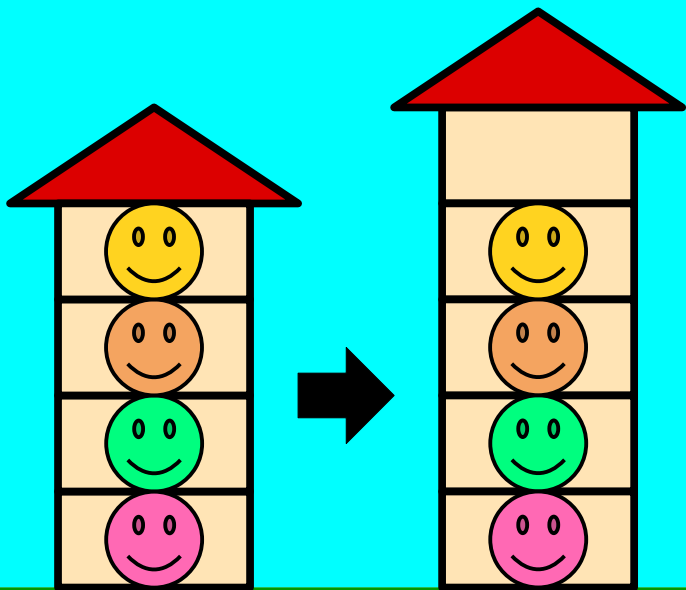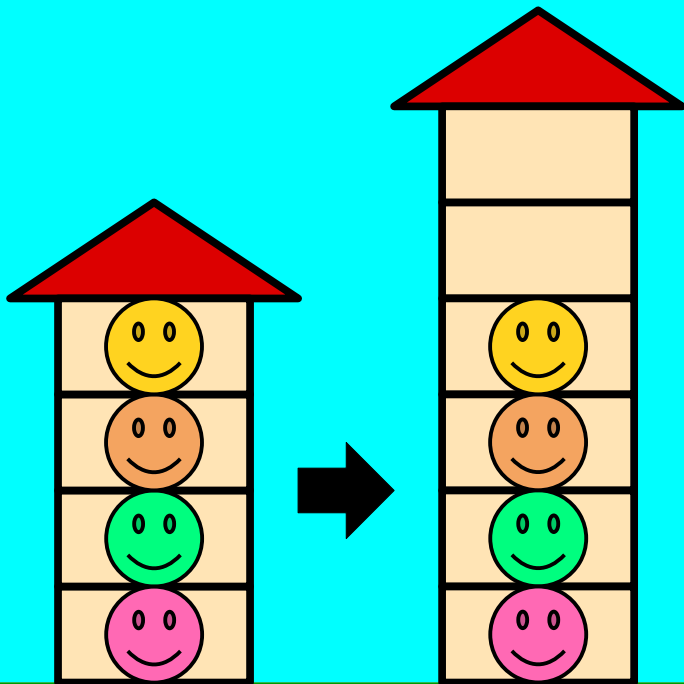
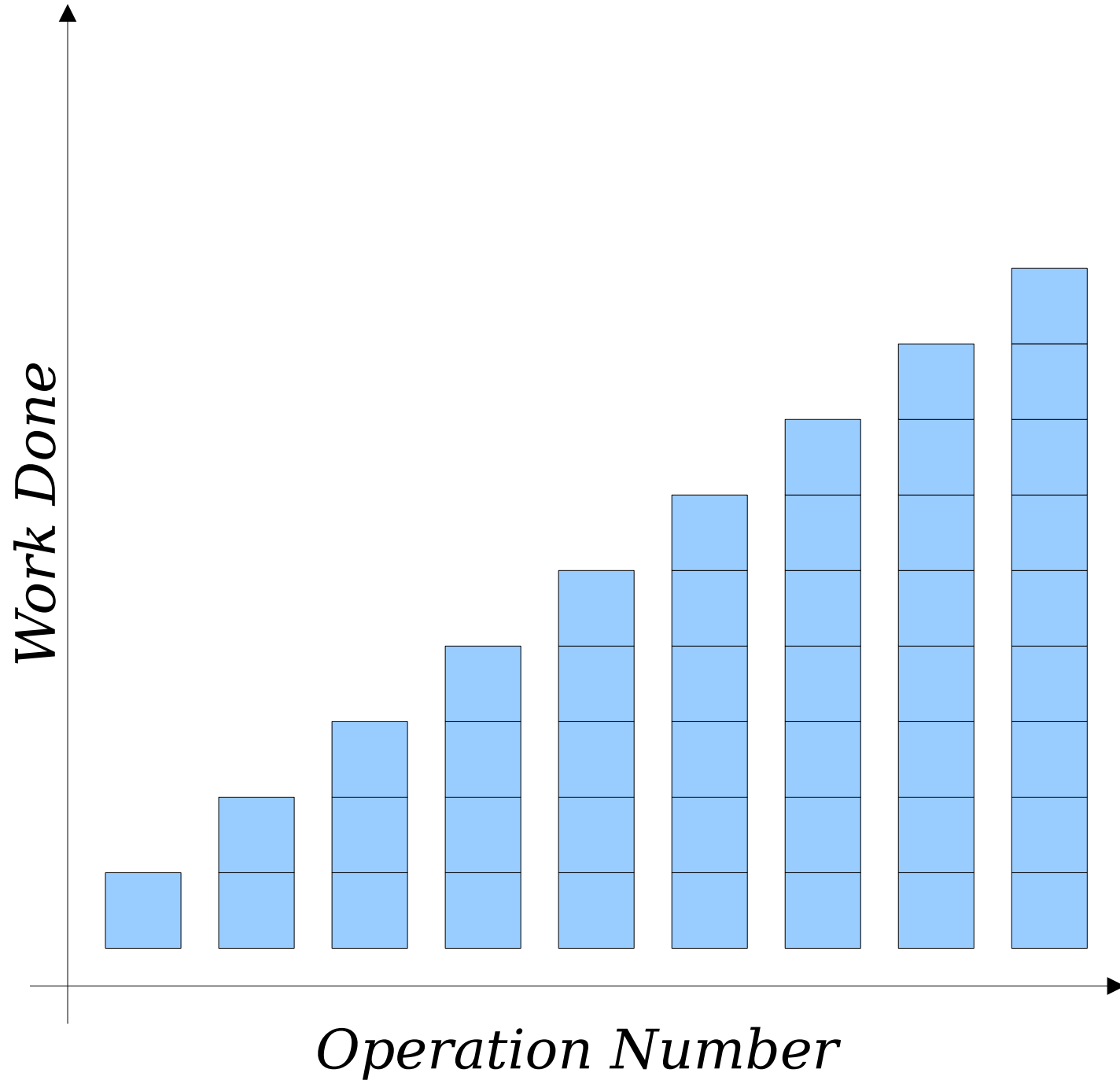***Question:*** How do we speed this up?
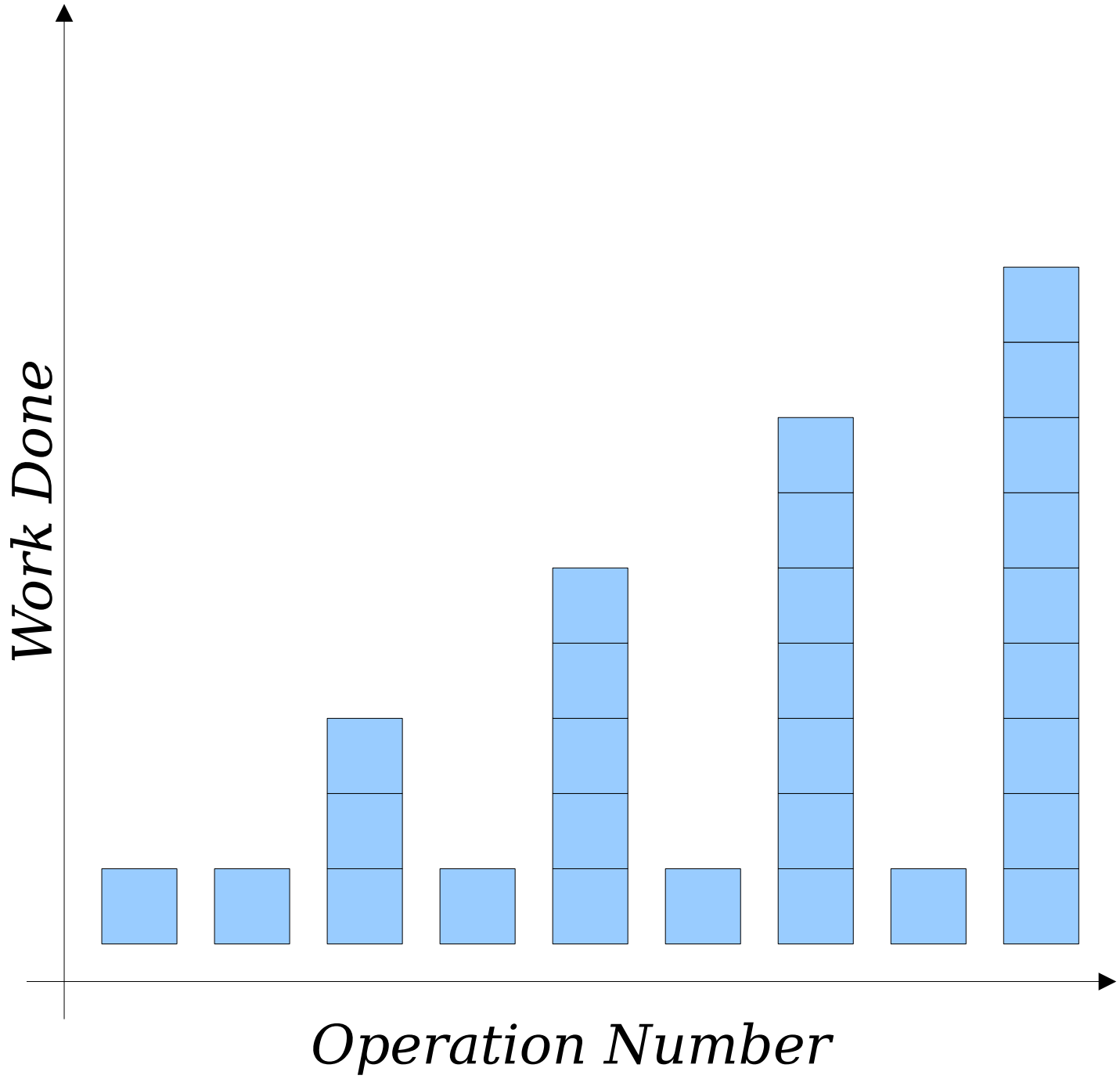
4 Items Moved

5 Items Moved

6 Items Moved

7 Items Moved

Work Done

Operation Number

Increase array size by **adding one**.

Increase array size by **adding two**.

Work Done

Operation Number

Increase array size by *adding two*.

Work Done

Operation Number

Work Done

Operation Number

Increase array size by *adding two*.

Increase array size by **adding two**.

Work Done

Operation Number

Increase array size by *adding two*.

Work Done

Operation Number

Increase array size by **adding two**.

Work Done

Operation Number

Work Done (vertical axis)

Operation Number (horizontal axis)

Increase array size by *adding two*.

Increase array size by **adog two**.
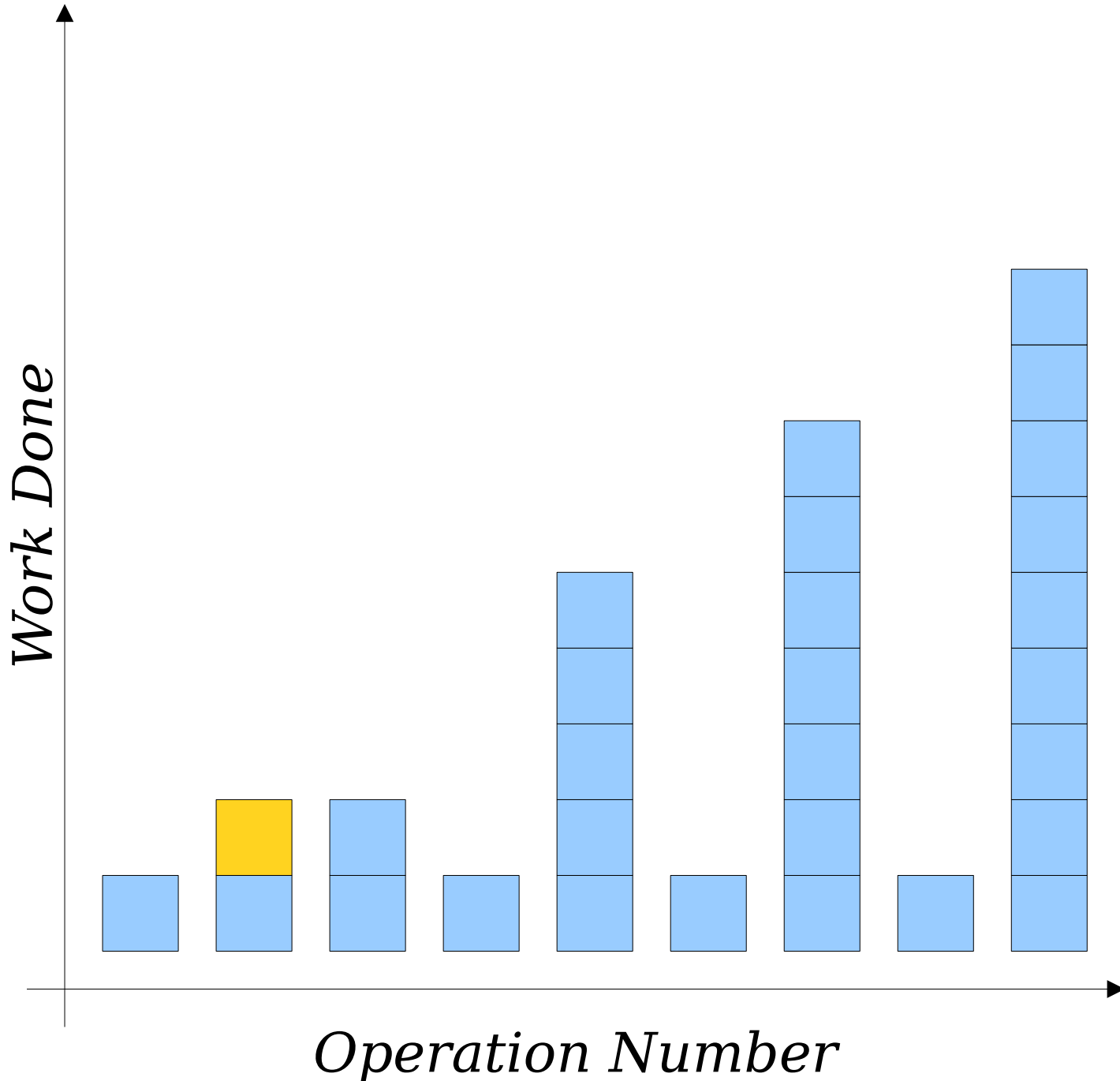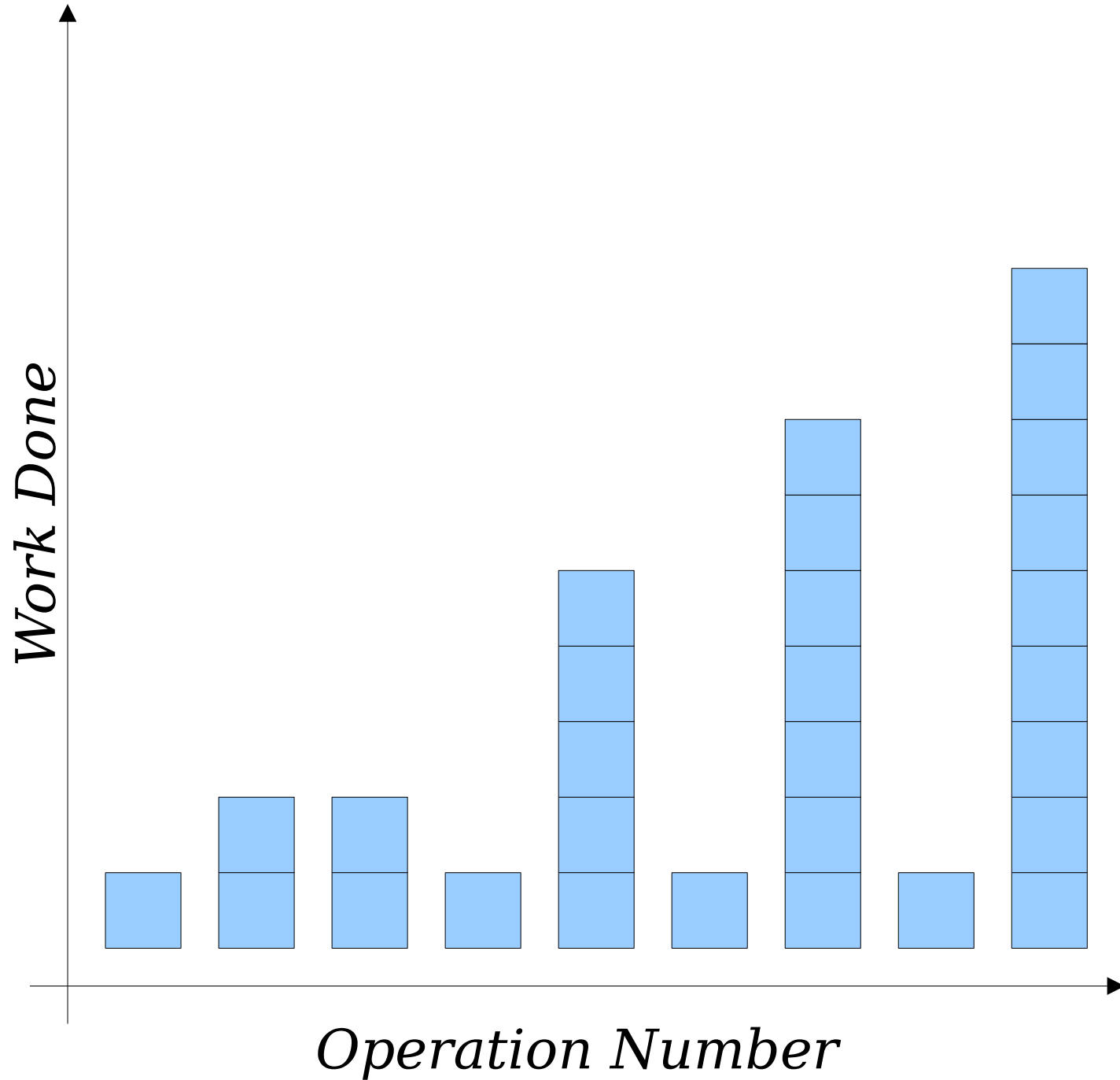
Work Done

Operation Number

Work Done

Operation Number
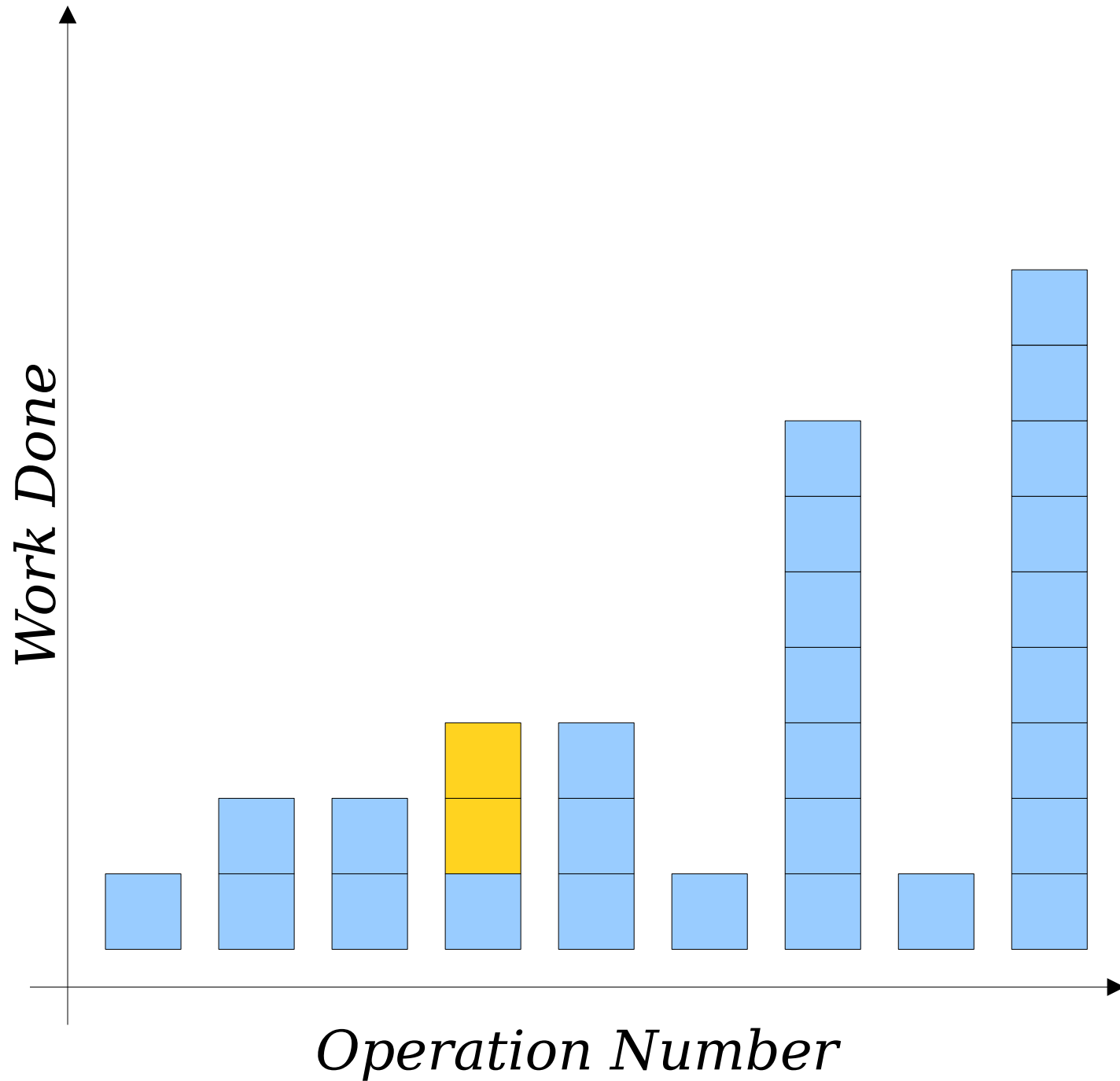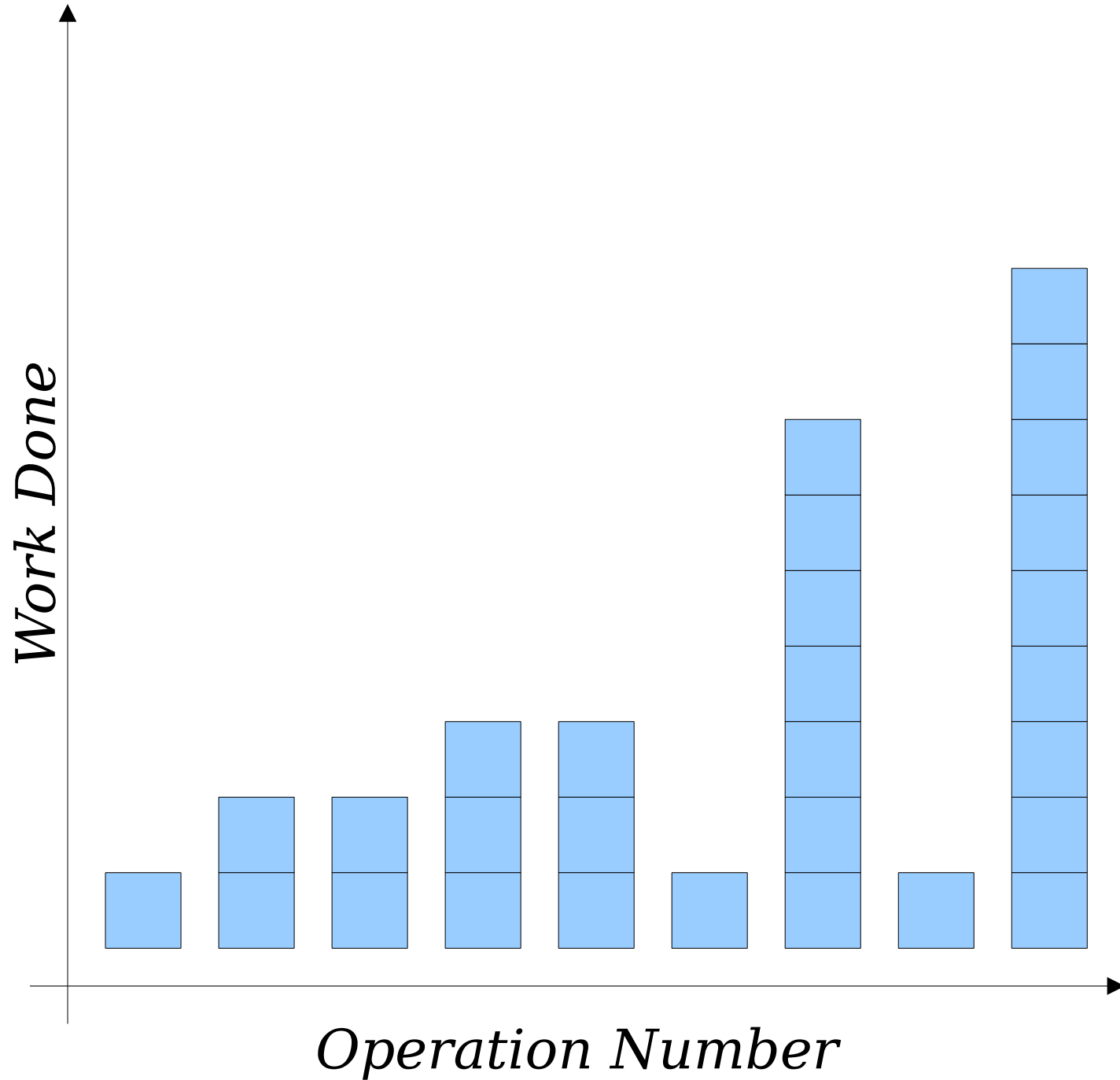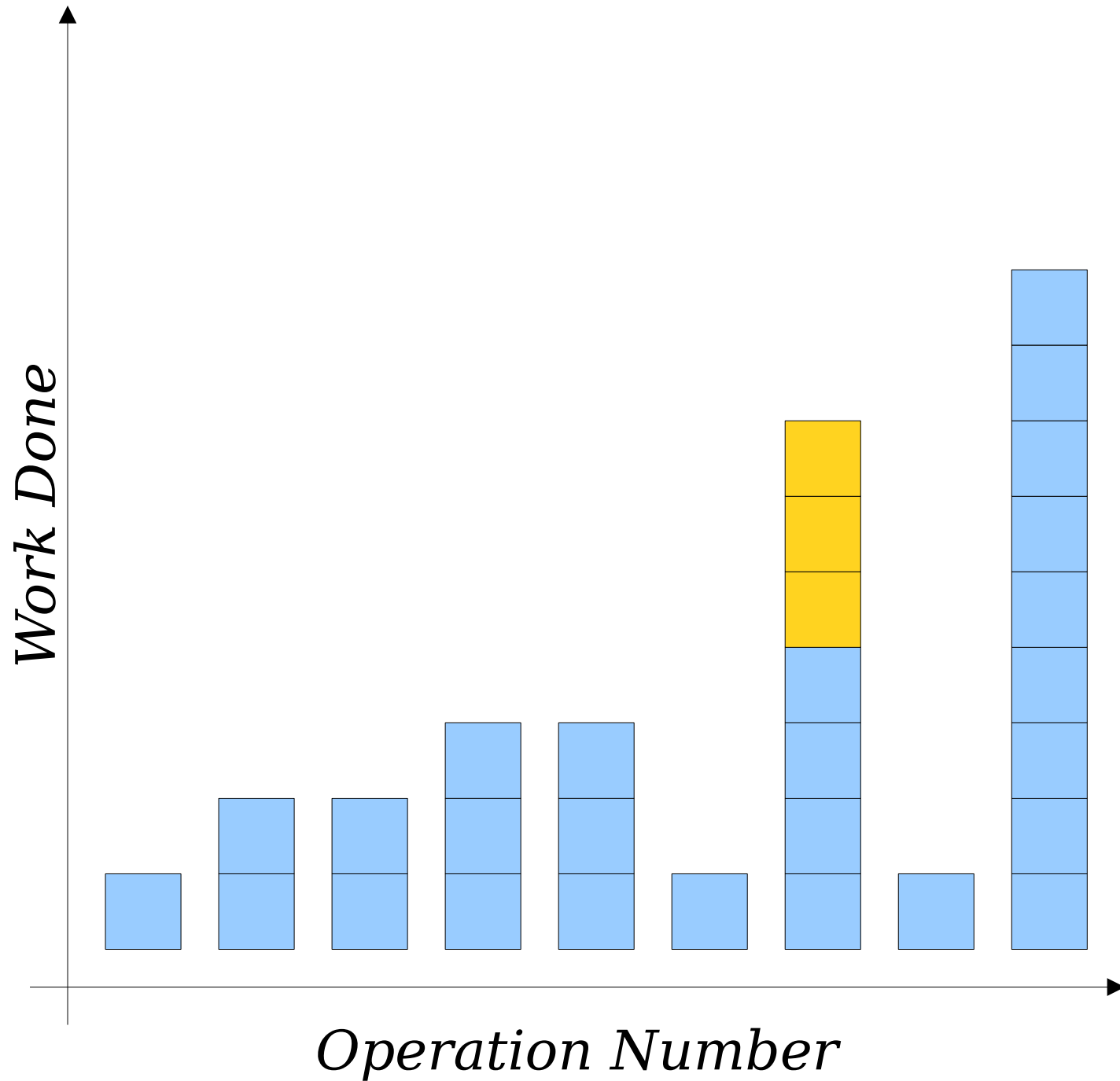
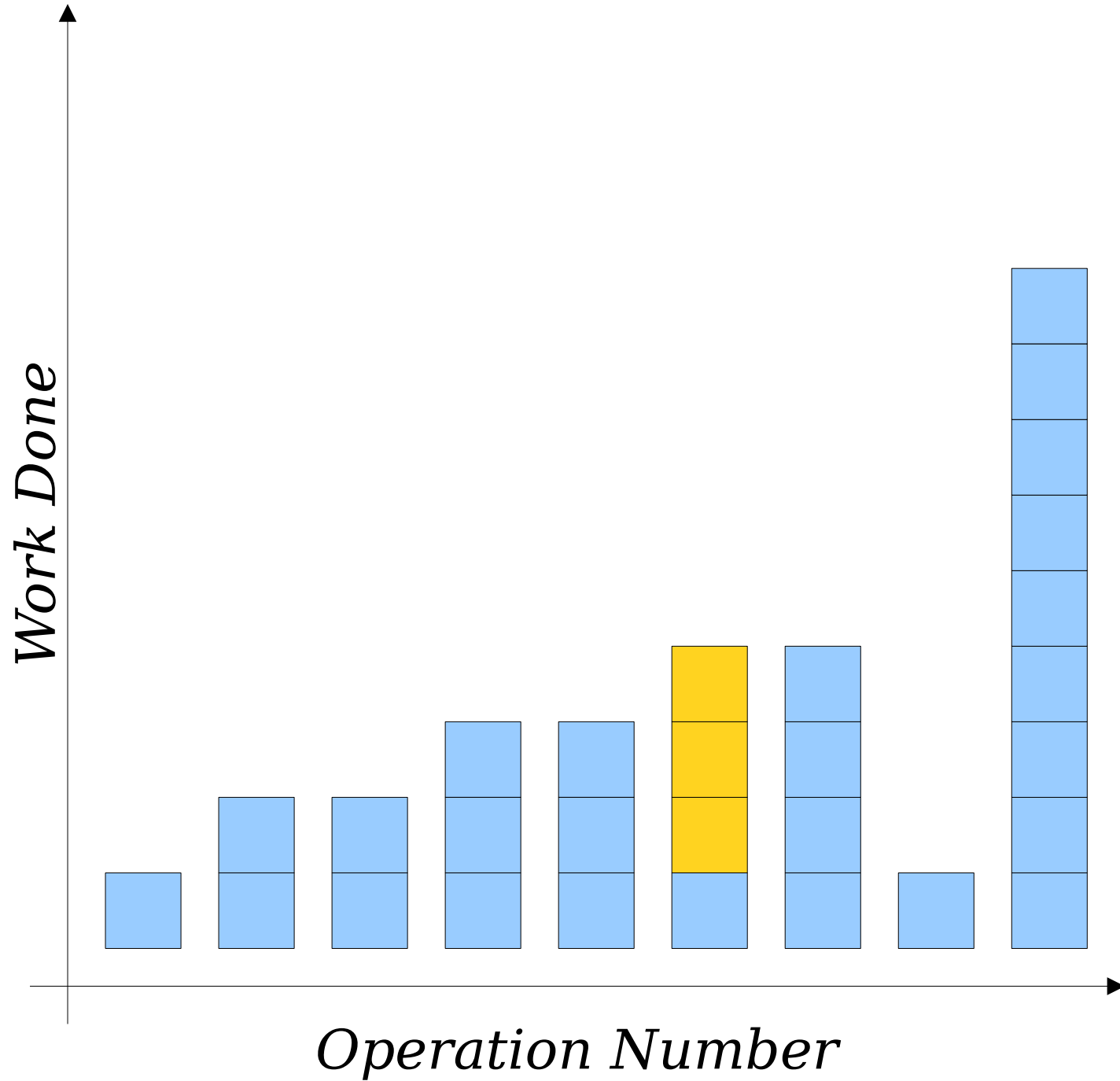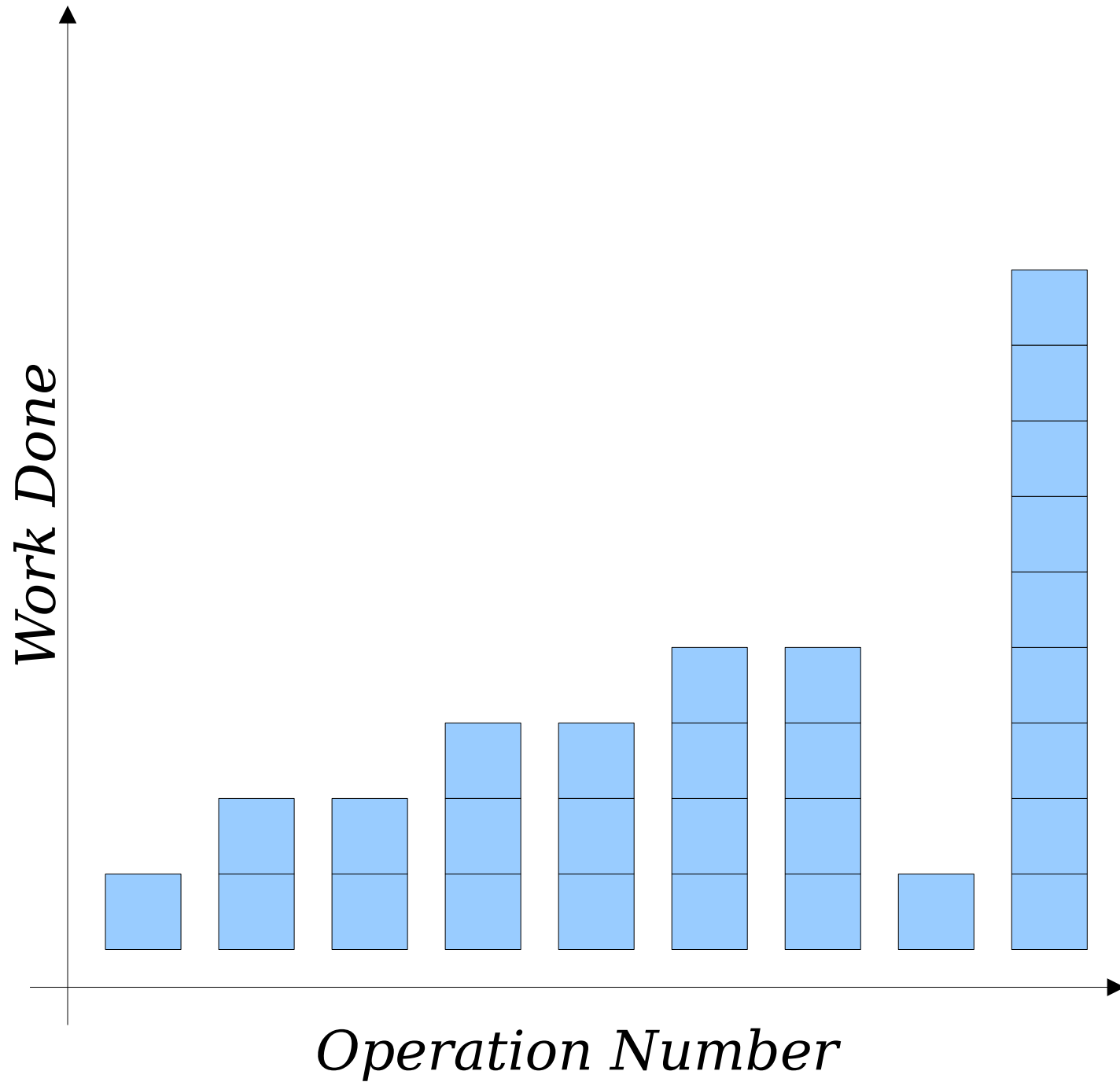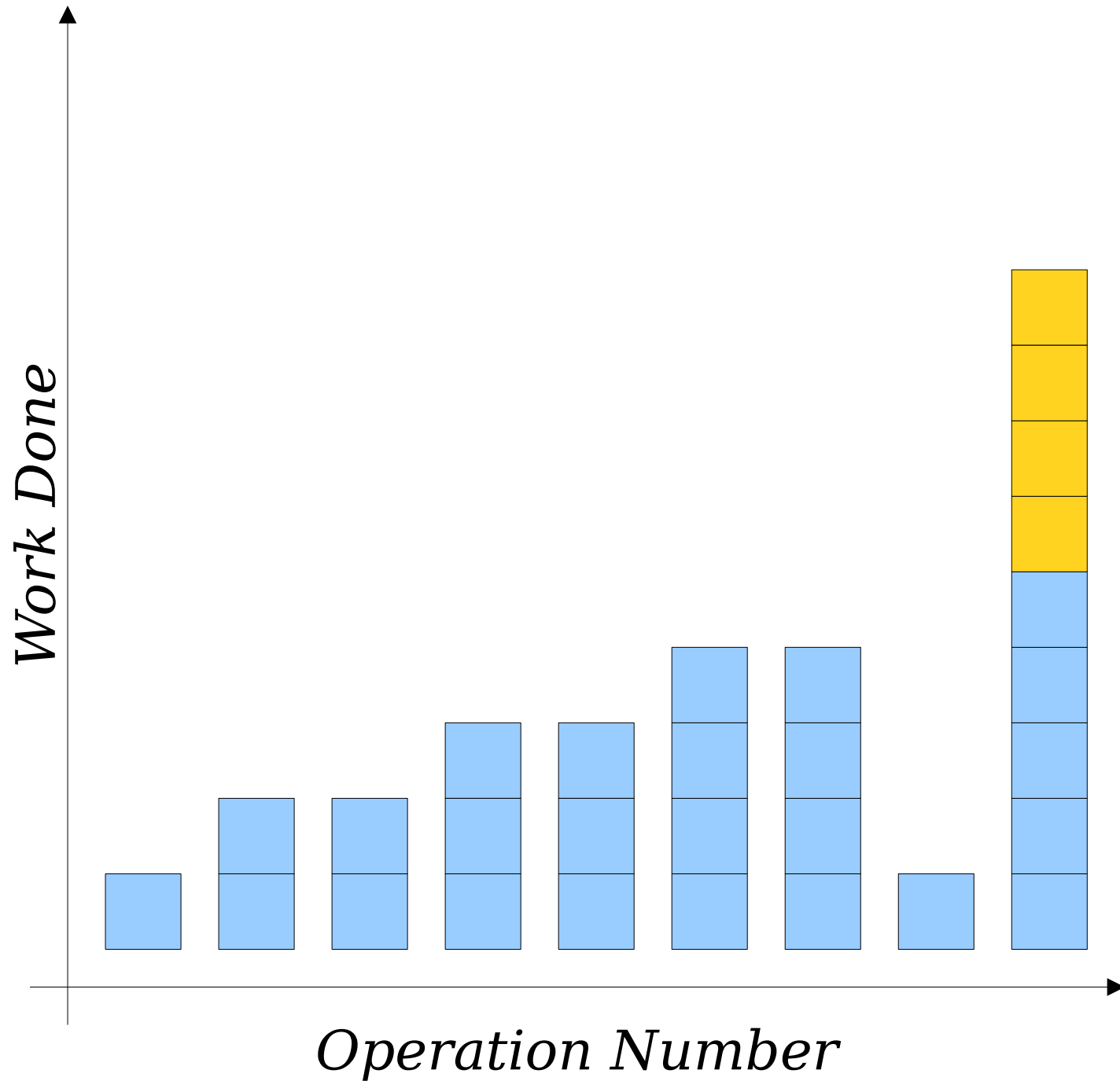Increase array size by *adding two*.

Increase array size by **adding two**.

Work Done

Operation Number

Work Done

Operation Number

Increase array size by *adding two*.

Work Done (vertical axis)

Operation Number (horizontal axis)

Increase array size by *adding two*.

This roughly halves the work done.
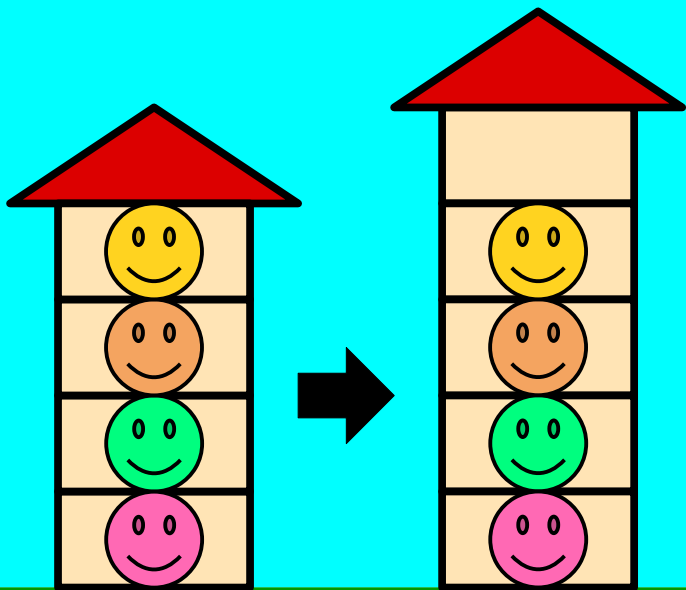
If we make the new array too big, we're might not make use of all the new space.

What's a good compromise?
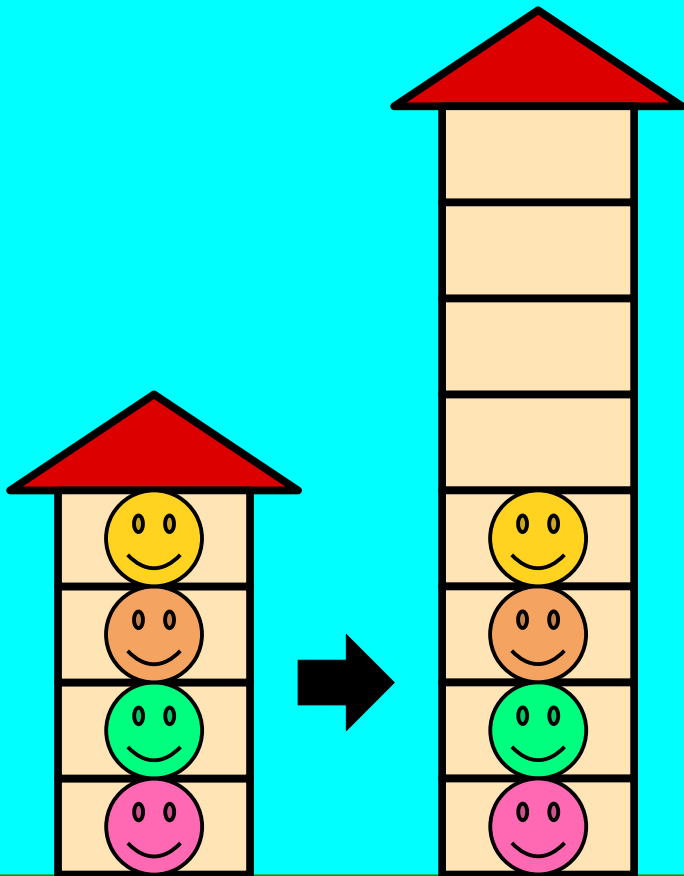
Idea: Make the new array twice as big as the old one.

This gives us a lot of free space, and we never use more than twice the space we need.
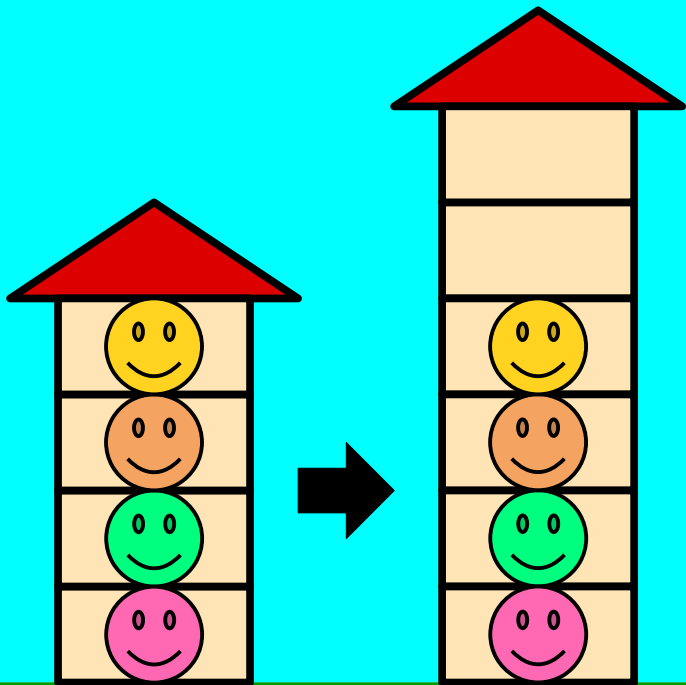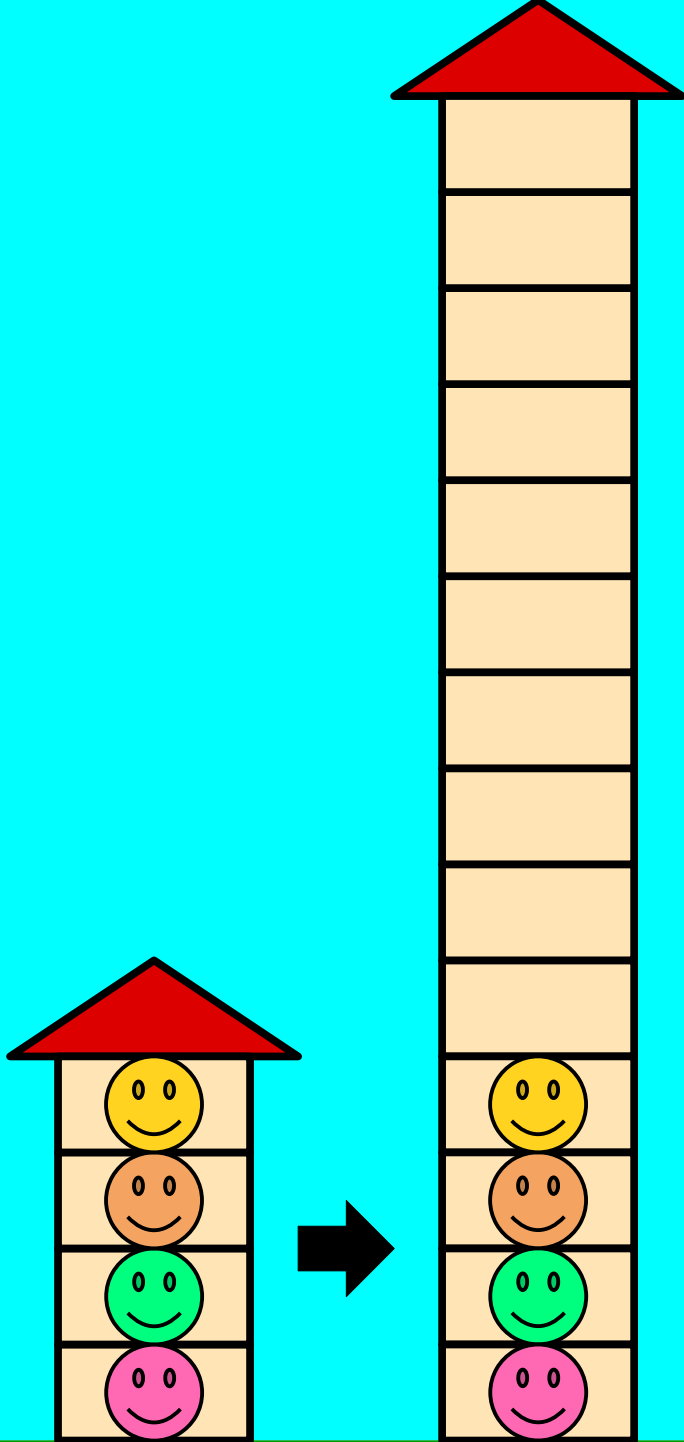
How do we analyze this?

Increase array size by **_multiplying by two_**.

Work Done

Operation Number

Work Done

Operation Number

Increase array size by **_multiplying by two_**.

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

Work Done

Operation Number

Work Done

Operation Number

Increase array size by **_multiplying by two_**.

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

Work Done

Operation Number

Work Done (vertical axis)

Operation Number (horizontal axis)

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

Work Done

Operation Number

Increase array size by **_multiplying by two_**.

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`?