EECS 281: Lab 2

Week of September 17, 2018

Announcements

- Reminder: Project 1 is due Thursday, 9/27
  - To submit on 9/28, use ONE late day
  - To submit on 9/29, you must use your SECOND late day
- Lab 2 is due 9/28
  - REMEMBER: Computing CARES survey (worth 3 points for lab 2)
    - Credit is for completion only, your responses will be anonymous to the instructors and will have no bearing on your grade

Agenda

- Lab 1 Handwritten Problem Solution
- Complexity Analysis
- `perf` demo
- Recurrence Relations
- Amortization
- C-string and string
- Project 1 Tips
- Handwritten Problem

Lab 1 Handwritten Problem

```
bool isPalindrome(Node* start, Node* end) {

    // Breaks when they meet in the middle
    // or when they both traverse the whole list.
    while (start != end) {
        if (start->value != end->value)
            return false;

        // Needed for even-length words
        if (start->next == end)
            return true;

        start = start->next;
        end = end->prev;
    } // while

    return true;
} // isPalindrome()
```

Note: cannot use ‘less than’ as linked list is not contiguous in memory

Complexity Analysis

- Big-O, Big-Theta, Big-Omega: you should be familiar with this material from EECS 203, and this was briefly reviewed in lecture 2 for this course.
- Know the substitution method and Master Theorem (and when you are allowed to use it) for solving recurrence relations.
- Will go over a few trickier examples here, but if you still have questions about this material, come to office hours.

Practice Question: Big-O

Consider the complexity of the function `foo` that we saw earlier in discussion:

$f(n) = 3n^3 - 3n^2 + 4n + 2$.

Which of the following statements are true?

1. $f(n) = O(3n^2)$
2. $f(n) = O(3n^3 - 3n^2)$
3. $f(n) = O(2n^3)$
4. $f(n) = O(n^3)$
5. $f(n) = O(n^n)$

Practice Question

For which of the following pairs of f(n) and g(n) is f(n) = O(g(n))?

a) $f(n) = 1 + n/6$, $g(n) = \sqrt{n} + 5\log(n) + 7$

b) $f(n) = 3^{3^n}$, $g(n) = 3^{(3*n)}$

c) $f(n) = 4n^2$, $g(n) = 2n^2 - 10n$

d) $f(n) = \ln(n)$, $g(n) = \ln(n/10)$

Practice Question: Code Analysis

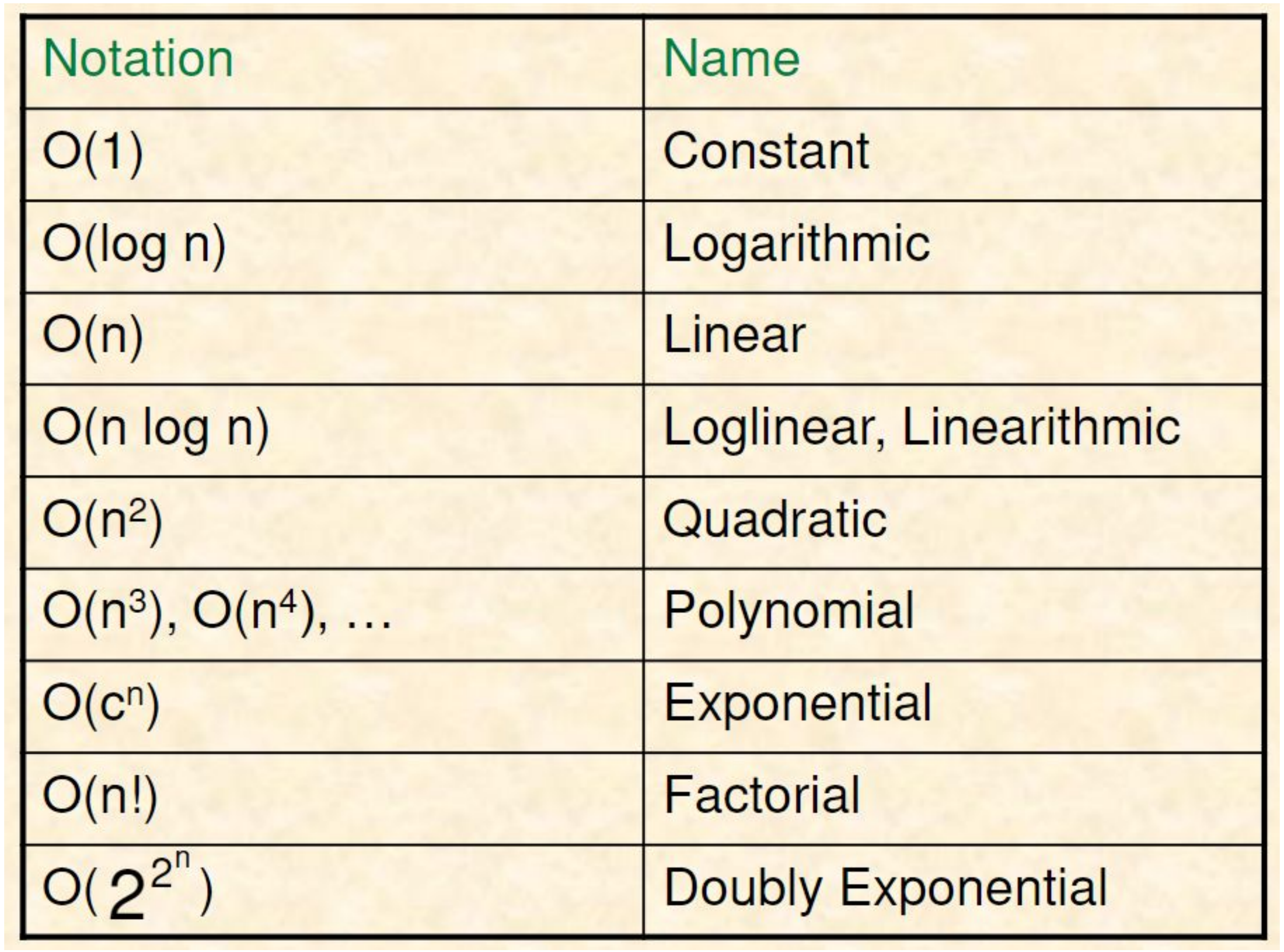
```
// Accepts an n x m array and an item to search for
// Assumes that each individual row is already sorted
void array2Dsearch(const vector<vector<int>> &array2D, int item) {
  for (size_t i = 0; i < array2D.size(); ++i) {
    if (binary_search(array2D[i].begin(), array2D[i].end(), item)) {
        cout << "found " << item << '\n';
        return;
    } // if
  } // for
  cout << "did not find " << item << '\n';
} // Total complexity (tightest bound) ???
```

perf demo

`perf` notes

- There is a reference guide on Canvas - check it out!
  - It’s in the Resources folder
  - Goes over the steps you take to use two different parts of `perf`
- You’ll be using `perf` for the lab assignment to help you learn more about it
- Feel free to use `perf` to help you out on projects, just keep in mind the tips listed on the reference guide before asking us for help at office hours!

Common Orders of Growth



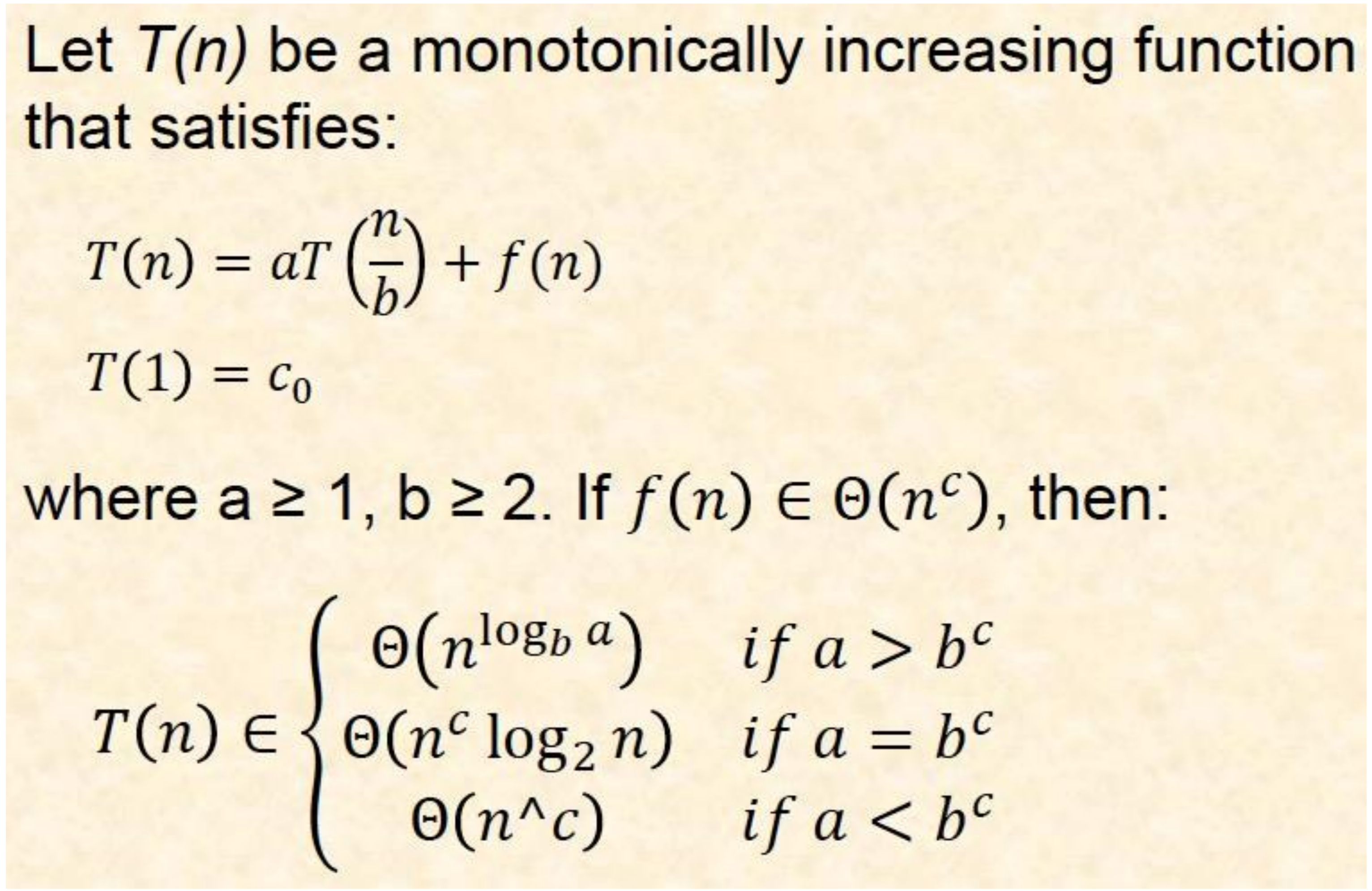
Notation	Name
$O(1)$	Constant
$O(\log n)$	Logarithmic
$O(n)$	Linear
$O(n \log n)$	Loglinear, Linearithmic
$O(n^2)$	Quadratic
$O(n^3)$, $O(n^4)$, …	Polynomial
$O(c^n)$	Exponential
$O(n!)$	Factorial
$O(2^{2^n})$	Doubly Exponential

Recurrence Relations

- A recurrence relation is an equation that is defined in terms of itself
- Recurrence relations define a sequence of values (think Fibonacci sequence)
- Many algorithms have time complexities which are naturally modelled by recurrence relations
    - i.e. algorithms that loop on a problem set, do something, and create smaller sub-problems
- Example:

$$f(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ f(n-1) + f(n-2) & \text{if } n \geq 2 \end{cases}$$

Solving RRs: Master Theorem



Let *T(n)* be a monotonically increasing function that satisfies:

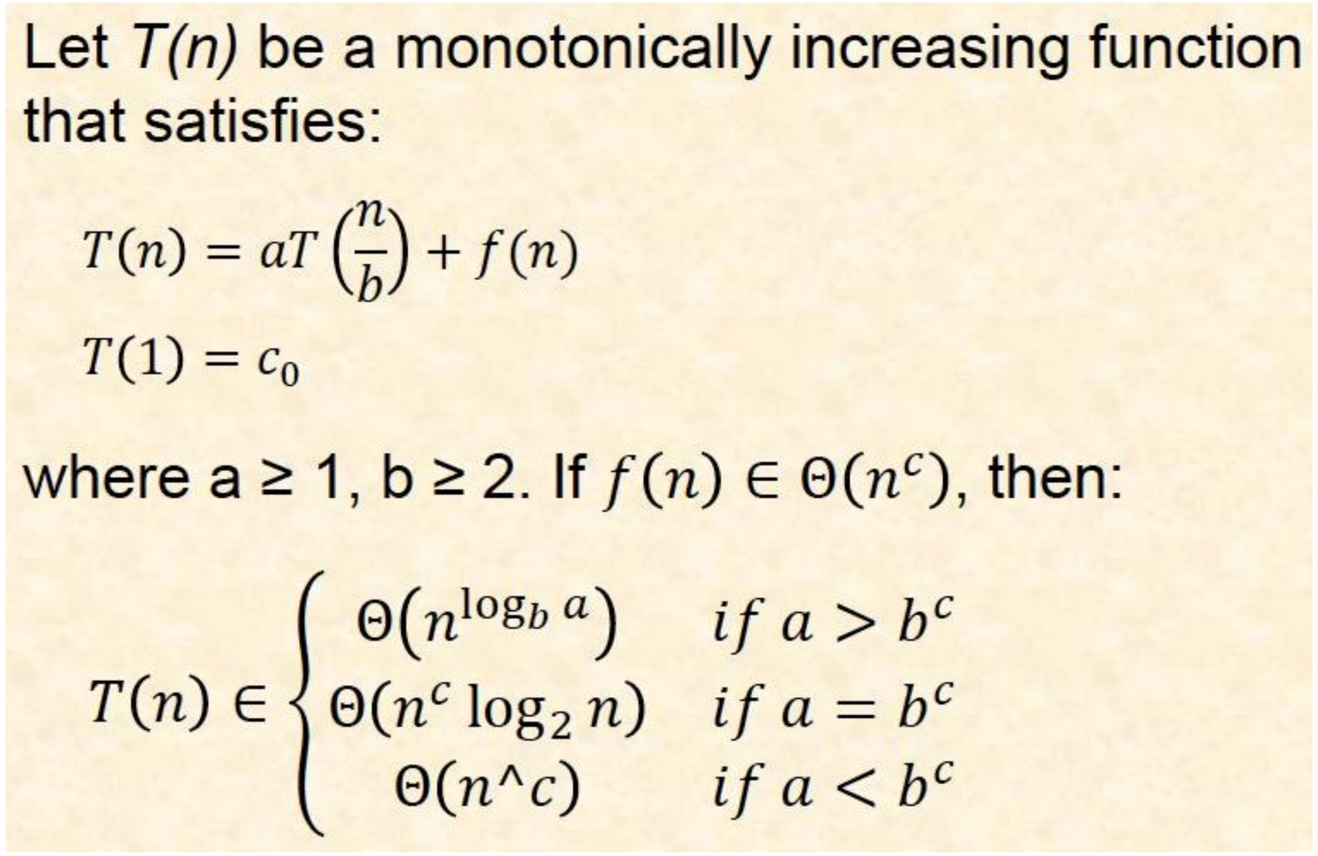
$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

$$T(1) = c_0$$

where a ≥ 1, b ≥ 2. If $f(n) \in \Theta(n^c)$, then:

$$T(n) \in \begin{cases} \Theta\left(n^{\log_b a}\right) & if\ a > b^c \\ \Theta(n^c \log_2 n) & if\ a = b^c \\ \Theta(n\text{^}c) & if\ a < b^c \end{cases}$$

Solving RRs: Master Theorem Practice



Let *T(n)* be a monotonically increasing function that satisfies:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

$$T(1) = c_0$$

where a ≥ 1, b ≥ 2. If $f(n) \in \Theta(n^c)$, then:

$$T(n) \in \begin{cases} \Theta\left(n^{\log_b a}\right) & if\ a > b^c \\ \Theta(n^c \log_2 n) & if\ a = b^c \\ \Theta(n^\wedge c) & if\ a < b^c \end{cases}$$

What is the complexity of the following recurrence relation?

$$T(n) = \begin{cases} c_0, & n = 1 \\ 8T(n/2) + 3n, & n > 1. \end{cases}$$

Solving RRs: Substitution

Sometimes recurrence relations won't satisfy the conditions required to use the master theorem!

Example: $T(n) = 2 * T(n - 1) + 1$, $T(1) = 1$

Problem: can't have numbers subtracted from n within the function

So instead, we can use substitution:

- Substitute the formula for T(n) into the recurrence terms on the RHS of the equation until a pattern is found
- Find a pattern that describes T(n) at the kth step
- Solve for k such that the term T(1) is present on the RHS. This makes the recurrence easy to solve for in closed form because T(1) is your base case!

Solving RRs: Substitution

Problem: $T(n) = 2 * T(n - 1) + 1$, $T(1) = 1$

Step 1: Substitute the formula for T(n) into the recurrence terms on the RHS of the equation until a pattern is found

Step #	Recurrence Equation	Subproblem Solution
1	$T(n) = 2 * T(n - 1) + 1$	$T(n - 1) = 2 * T(n - 2) + 1$
2		
3		
4		
…		
k	???????	

Solving RRs: Substitution

Problem: T(n) = 2 * T(n - 1) + 1, T(1) = 1

Step 1: Substitute the formula for T(n) into the recurrence terms on the RHS of the equation until a pattern is found

Tip: Don't simplify terms too much – leave them expanded so you can see a pattern!

Step #	Recurrence Equation	Subproblem Solution
1	T(n) = 2 * T(n - 1) + 1	T(n -1) = 2 * T(n - 2) + 1
2	T(n) = 2 * (2 * T(n - 2) + 1) + 1 = 2 * 2 * T(n - 2) + 2 + 1	T(n - 2) = 2 * T(n - 3) + 1
3		
4		
…		
k	???????	

Solving RRs: Substitution

Problem: T(n) = 2 * T(n - 1) + 1, T(1) = 1

Step 1: Substitute the formula for T(n) into the recurrence terms on the RHS of the equation until a pattern is found

Tip: Don't simplify terms too much – leave them expanded so you can see a pattern!

Step #	Recurrence Equation	Subproblem Solution
1	T(n) = 2 * T(n - 1) + 1	T(n -1) = 2 * T(n - 2) + 1
2	T(n) = 2 * (2 * T(n - 2) + 1) + 1 = 2 * 2 * T(n - 2) + 2 + 1	T(n - 2) = 2 * T(n - 3) + 1
3	T(n) = 2 * 2 * (2 * T(n - 3) + 1) + 2 + 1 = 2 * 2 * 2 * T (n - 3) + 2 * 2 + 2 + 1	T(n - 3) = 2 * T(n - 4) + 1
4		
…		
k	???????	

Solving RRs: Substitution

Problem: T(n) = 2 * T(n - 1) + 1, T(1) = 1

Step 1: Substitute the formula for T(n) into the recurrence terms on the RHS of the equation until a pattern is found

Tip: Don't simplify terms too much – leave them expanded so you can see a pattern!

Step #	Recurrence Equation	Subproblem Solution
1	T(n) = 2 * T(n - 1) + 1	T(n -1) = 2 * T(n - 2) + 1
2	T(n) = 2 * (2 * T(n - 2) + 1) + 1 = 2 * 2 * T(n - 2) + 2 + 1	T(n - 2) = 2 * T(n - 3) + 1
3	T(n) = 2 * 2 * (2 * T(n - 3) + 1) + 2 + 1 = 2 * 2 * 2 * T (n - 3) + 2 * 2 + 2 + 1	T(n - 3) = 2 * T(n - 4) + 1
4	T(n) = 2 * 2 * 2 * (2 * T(n - 4) + 1) + 2 * 2 + 2 + 1 = 2 * 2 * 2 * 2 * T(n - 4) + 2 * 2 * 2 + 2 * 2 + 2 + 1	…
…		
k	???????	

Solving RRs: Substitution

Problem: T(n) = 2 * T(n - 1) + 1, T(1) = 1

Step 2: Find a pattern that describes T(n) at the kth step

Step #	Recurrence Equation	Subproblem Solution
1	T(n) = 2 * T(n - 1) + 1	T(n -1) = 2 * T(n - 2) + 1
2	T(n) = 2 * (2 * T(n - 2) + 1) + 1 = 2 * 2 * T(n - 2) + 2 + 1	T(n - 2) = 2 * T(n - 3) + 1
3	T(n) = 2 * 2 * (2 * T(n - 3) + 1) + 2 + 1 = 2 * 2 * 2 * T (n - 3) + 2 * 2 + 2 + 1	T(n - 3) = 2 * T(n - 4) + 1
4	T(n) = 2 * 2 * 2 * (2 * T(n - 4) + 1) + 2 * 2 + 2 + 1 = 2 * 2 * 2 * 2 * T(n - 4) + 2 * 2 * 2 + 2 * 2 + 2 + 1	…
…		
k		

Solving RRs: Substitution

Problem: T(n) = 2 * T(n - 1) + 1, T(1) = 1

Step 2: Find a pattern that describes T(n) at the kth step

Step #	Recurrence Equation	Subproblem Solution
1	T(n) = 2 * T(n - 1) + 1	T(n -1) = 2 * T(n - 2) + 1
2	T(n) = 2 * (2 * T(n - 2) + 1) + 1 = 2 * 2 * T(n - 2) + 2 + 1	T(n - 2) = 2 * T(n - 3) + 1
3	T(n) = 2 * 2 * (2 * T(n - 3) + 1) + 2 + 1 = 2 * 2 * 2 * T (n - 3) + 2 * 2 + 2 + 1	T(n - 3) = 2 * T(n - 4) + 1
4	T(n) = 2 * 2 * 2 * (2 * T(n - 4) + 1) + 2 * 2 + 2 + 1 = 2 * 2 * 2 * 2 * T(n - 4) + 2 * 2 * 2 + 2 * 2 + 2 + 1	…
…		
k	$T(n) = 2^k \cdot T(n-k) + \Sigma_{m=0}^{k-1} 2^m$	

Solving RRs: Substitution

Problem: T(n) = 2 * T(n - 1) + 1, T(1) = 1

Step 3: Solve for k such that the term T(1) is present on the RHS. This makes the recurrence easy to solve for in closed form because T(1) is your base case!

- We know that at the kth step, T(n) = 2^k * T(n - k) + $\sum_{m=0}^{k-1} 2^m$
- T(n - k) = T(1) when k = n - 1
- So solve fore T(n) at the n - 1 step (i.e. substitute n - 1 for k):
  - T(n) = 2^{n-1} * T (1) + $\sum_{i=0}^{n-2} 2^i$ = $2^{n-1} + 2^{n-2} + 1$
  - T(n) = O(2^n)

Formula to simplify summation:

$$\sum_{i=m}^{n} a \cdot r^i = \frac{a(r^m - r^{n+1})}{1 - r}$$

a = 1, r = 2, m = 0, “n” = n – 2

1 (1 – 2^{n-2}) / (-1) = 2^{n-2} + 1

Amortization

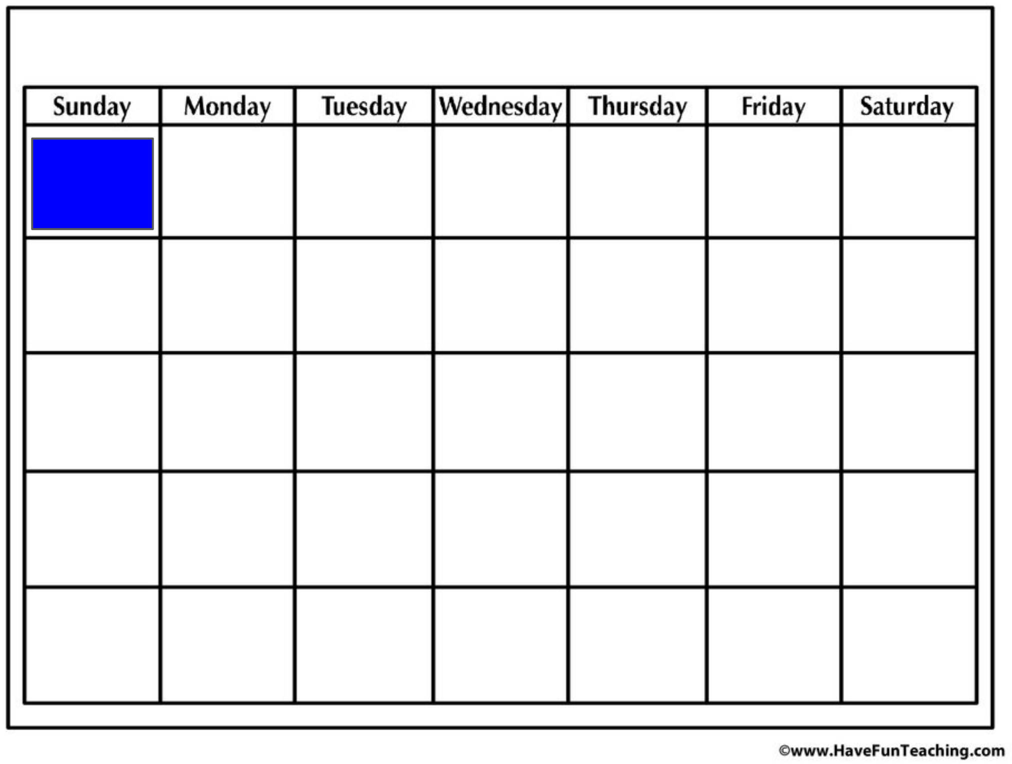
Amortized Complexity

- An alternative way to measure the cost of an operation:
  - We've heard of *worst case* and *average case*; these are both different
    - *Worst case*: what is the maximum amount of resources needed for *one* operation?
    - *Average case*: what is the *expected value* of resources needed?
      - Suggests that the operation or the input is random, otherwise this doesn't even make sense (think 203 or stats)

- **Amortized cost**:
  - How much does the operation contribute to the *total cost* of the program?
  - Used when the worst-case is too pessimistic

Example 1
Gym Membership

- On the first of each month you pay $30 for a gym membership

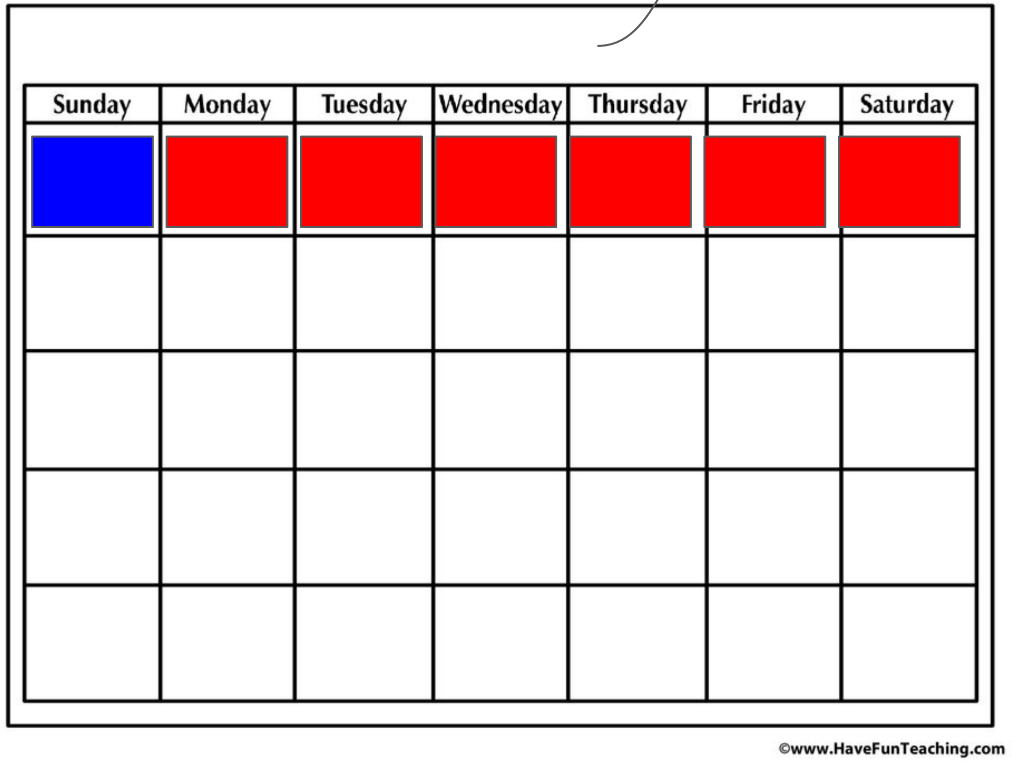


Total Cost: $30

Gym Membership

- On the first of each month you pay $30 for a gym membership
- In this case, there is only one payment every ~30 days
    - The rest of the days in the month, you don't have to pay anything

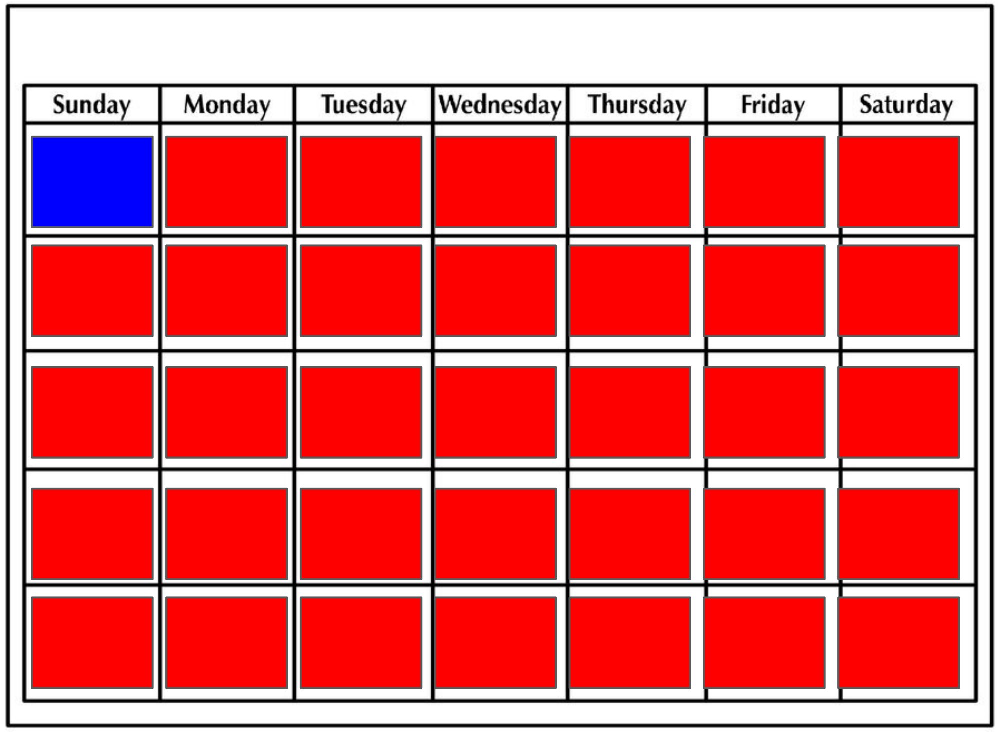
(35 day example not to scale)



Total Cost: $30

Gym Membership

- On the first of each month you pay $30 for a gym membership
- In this case, there is only one payment every ~30 days
    - The rest of the days in the month, you don't have to pay anything
- After amortization, you find you are spending only around 1 dollar a day to go to the gym, despite it seeming like much more earlier in the month



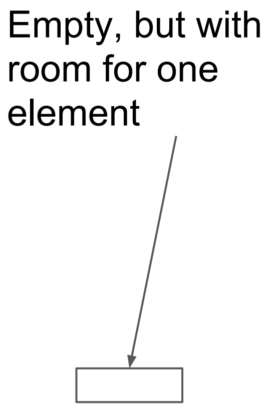
©www.HaveFunTeaching.com

Total Cost: STILL $30

Amortized Cost: ~$1

Example 2 - std::vector Amortized Analysis: “Proof”

Let’s fill up a vector by calling `push_back` over and over:



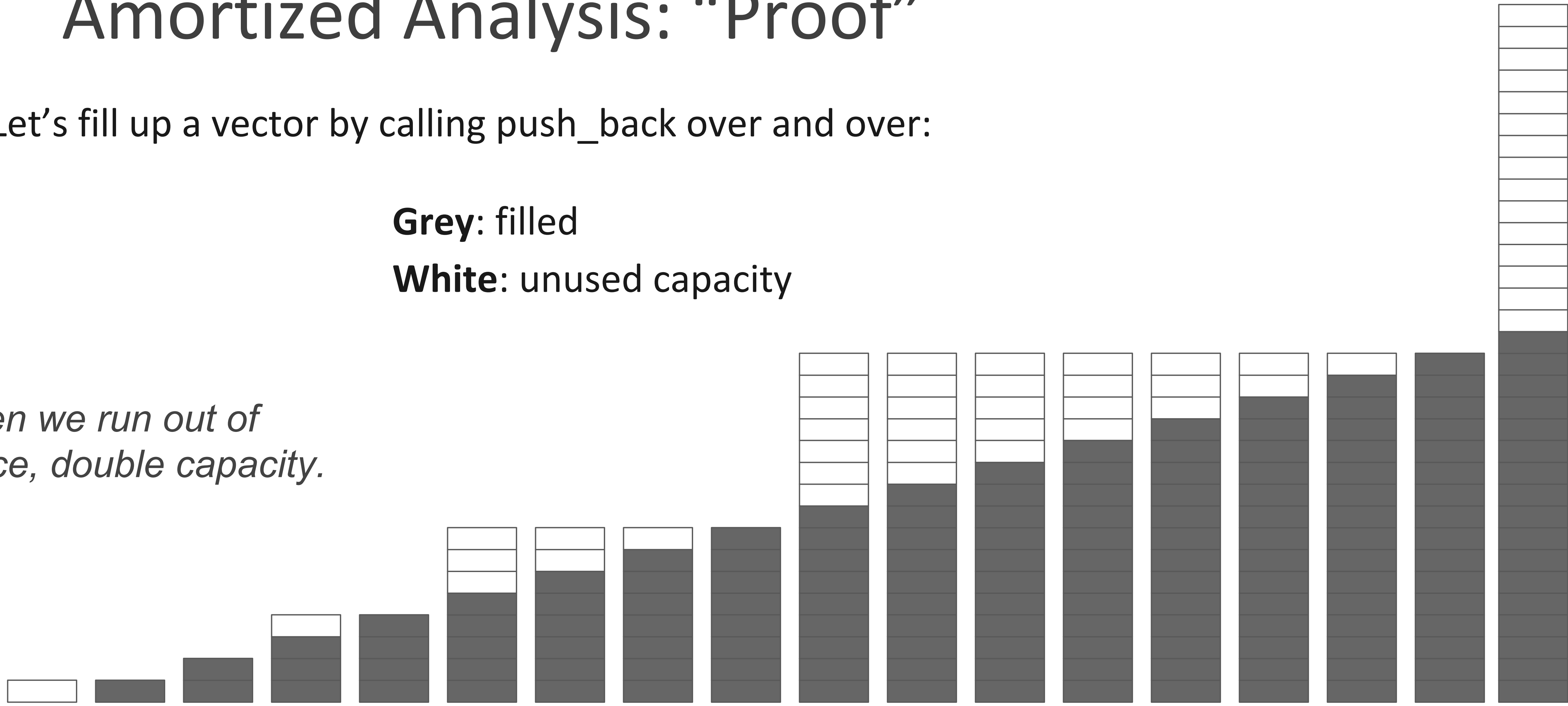
Amortized Analysis: “Proof”

Let’s fill up a vector by calling push_back over and over:

Grey: filled

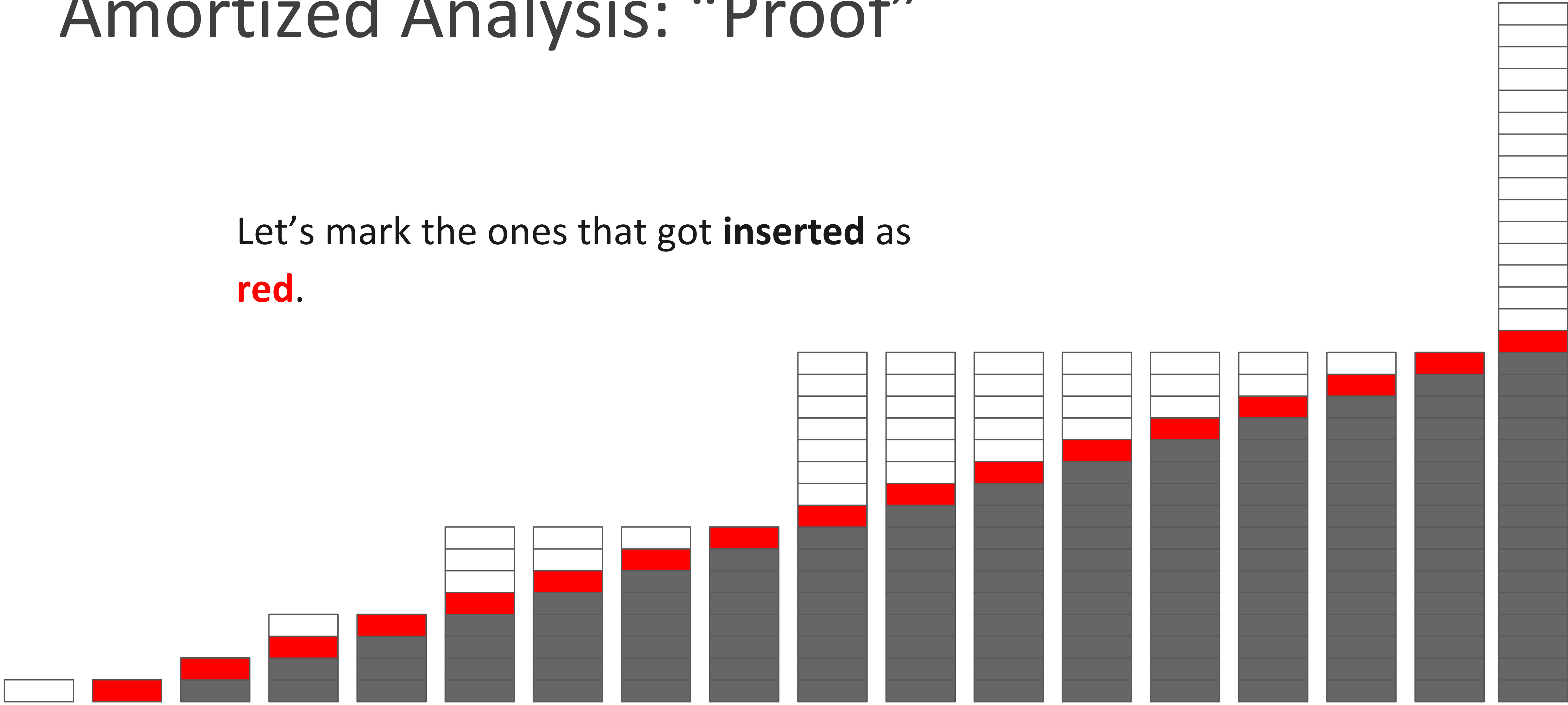
White: unused capacity

When we run out of space, double capacity.



Amortized Analysis: “Proof”

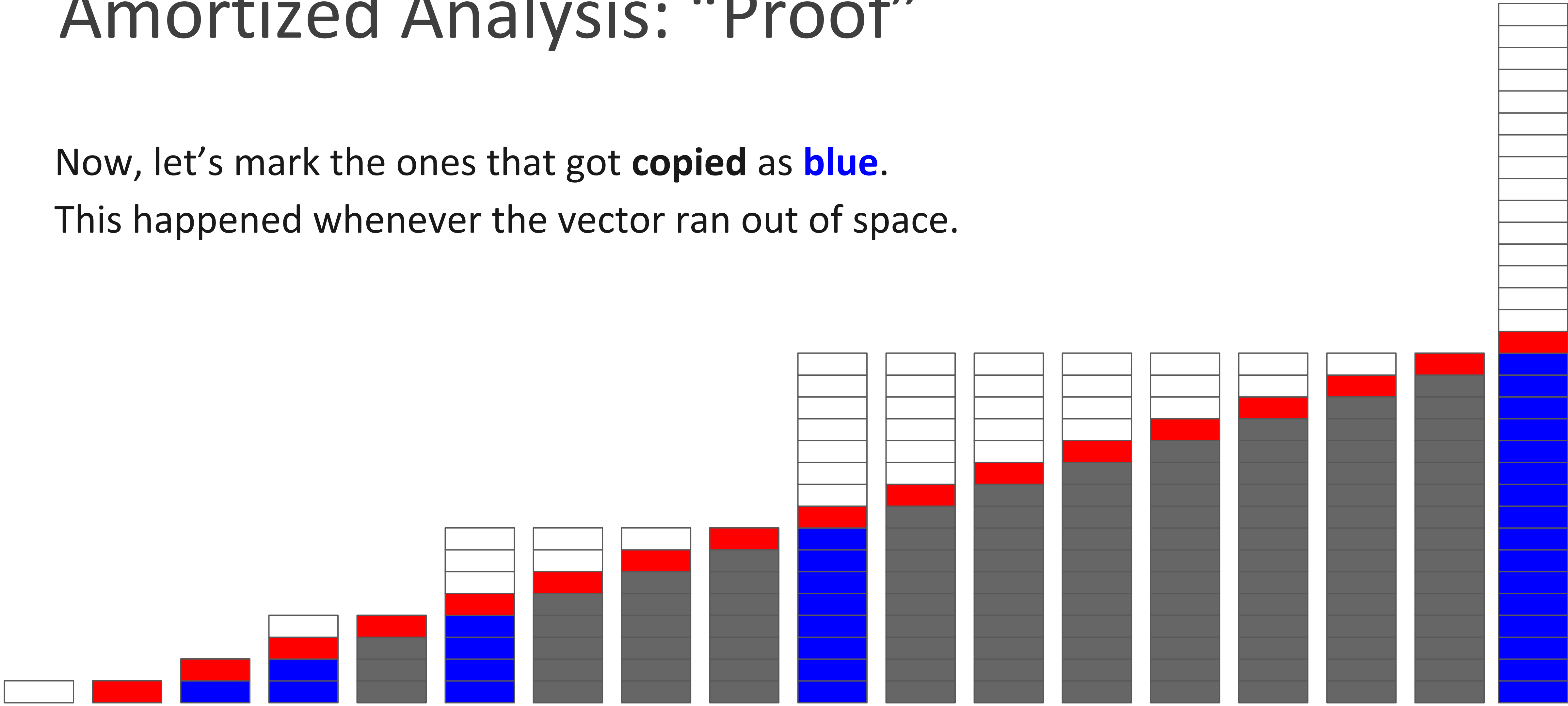
Let’s mark the ones that got **inserted** as **red**.



Amortized Analysis: “Proof”

Now, let’s mark the ones that got **copied** as **blue**.

This happened whenever the vector ran out of space.



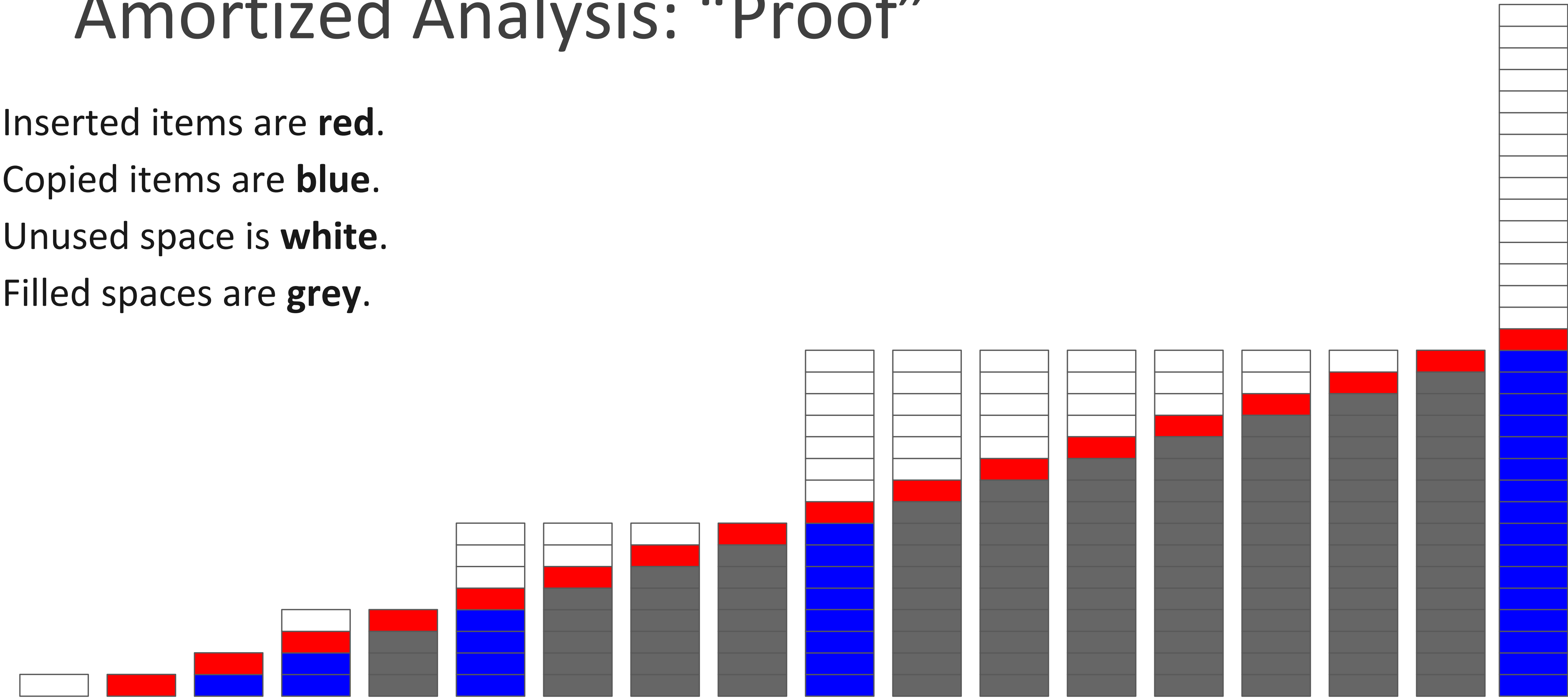
Amortized Analysis: "Proof"

Inserted items are **red**.

Copied items are **blue**.

Unused space is **white**.

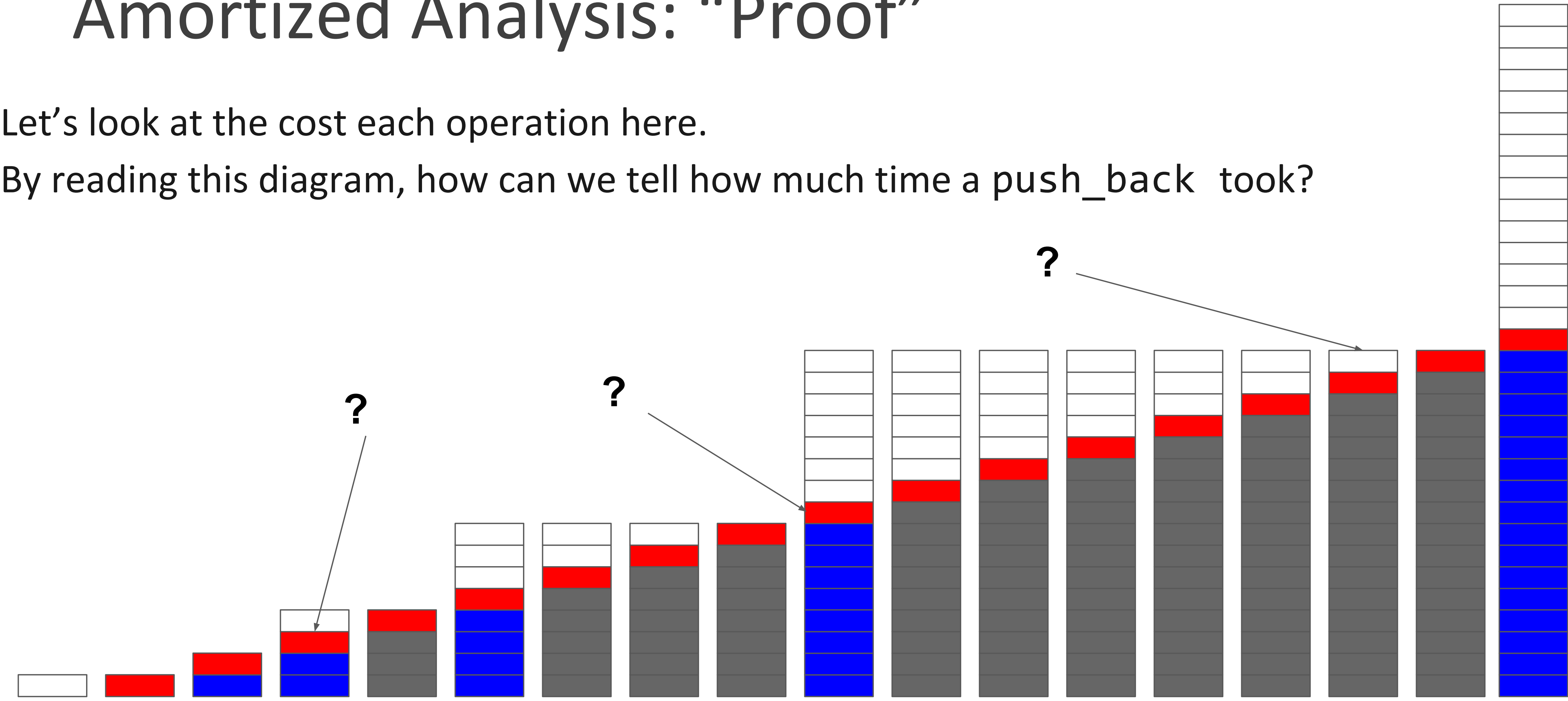
Filled spaces are **grey**.



Amortized Analysis: “Proof”

Let’s look at the cost each operation here.

By reading this diagram, how can we tell how much time a push_back took?

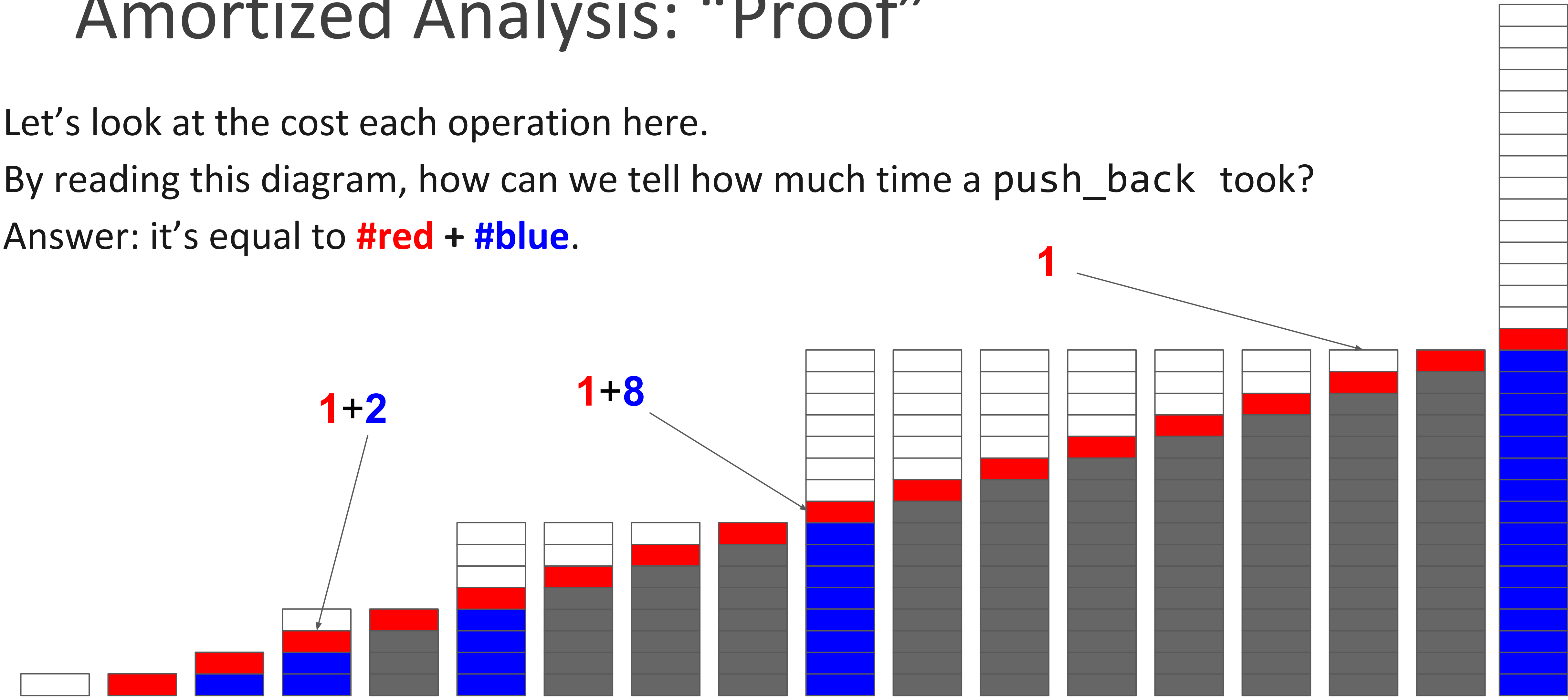


Amortized Analysis: “Proof”

Let’s look at the cost each operation here.

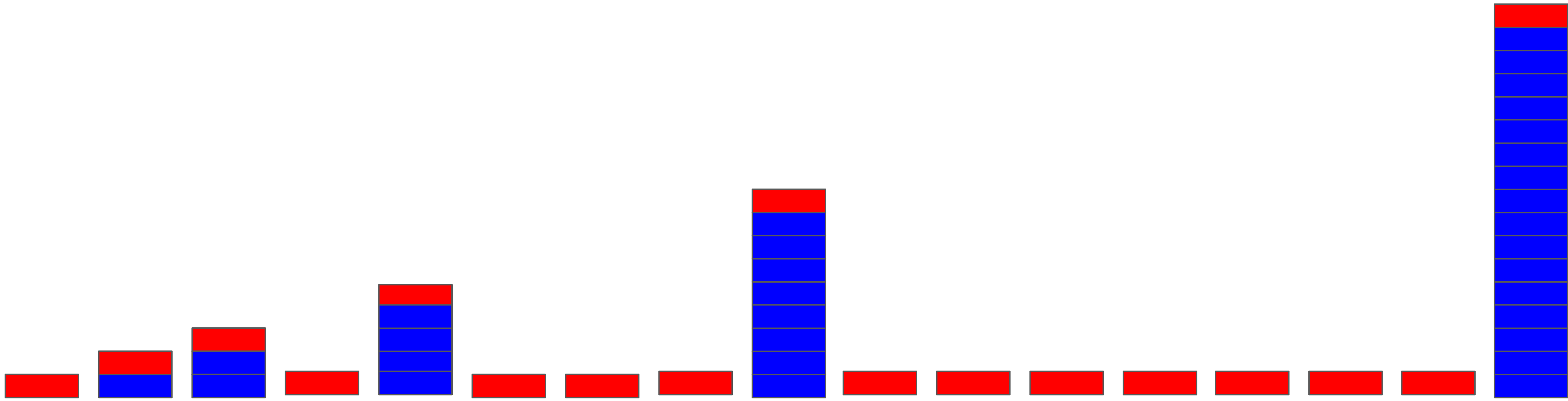
By reading this diagram, how can we tell how much time a `push_back` took?

Answer: it’s equal to **#red** + **#blue**.



Amortized Analysis: “Proof”

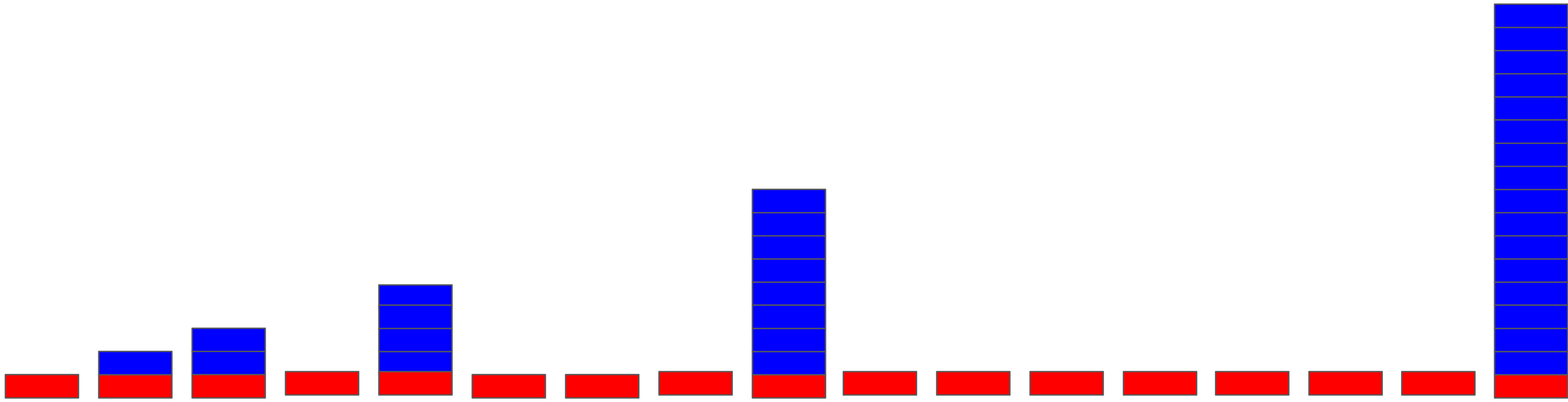
The ***height*** of a tower is now the ***cost*** of the corresponding `push_back`!



Amortized Analysis: “Proof”

To make things nicer, move the reds to the bottom.

- This doesn’t change their height, so the total cost is the same.

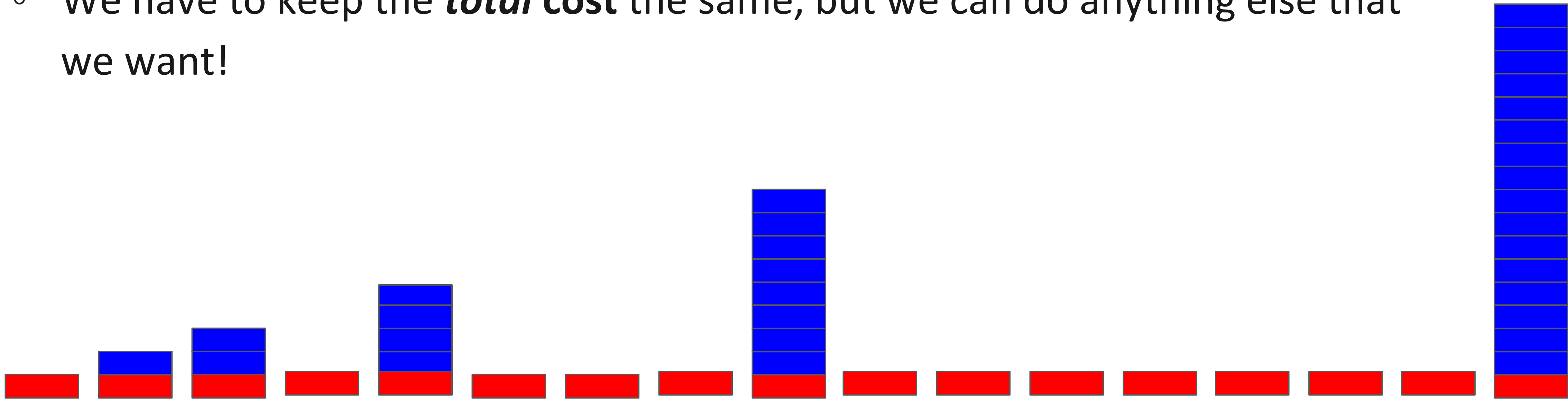


Amortized Analysis: “Proof”

Now, we’re going to **amortize the costs**!

This means we can move them around.

- We have to keep the ***total* cost** the same, but we can do anything else that we want!

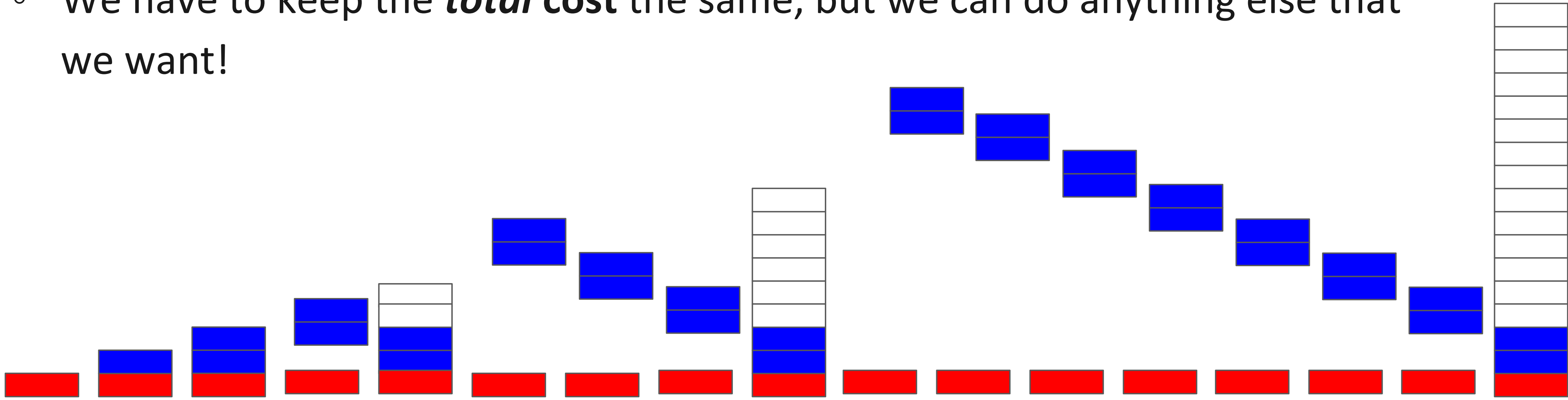


Amortized Analysis: “Proof”

Now, we’re going to **amortize the costs**!

This means we can move them around.

- We have to keep the ***total* cost** the same, but we can do anything else that we want!



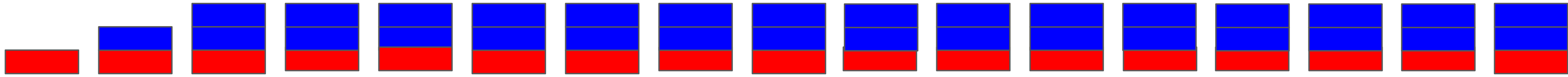
Amortized Analysis: “Proof”

Hey, look, it’s constant!

All we did is shuffle around the costs.

- The **total** is the same. *Very important*.
- The **number** of operations is the same.
  - (Also, we didn’t move anything into the future - but this is a more subtle restriction)

Thus 1 + 2 = 3 steps = O(1) is the amortized cost of the operation!



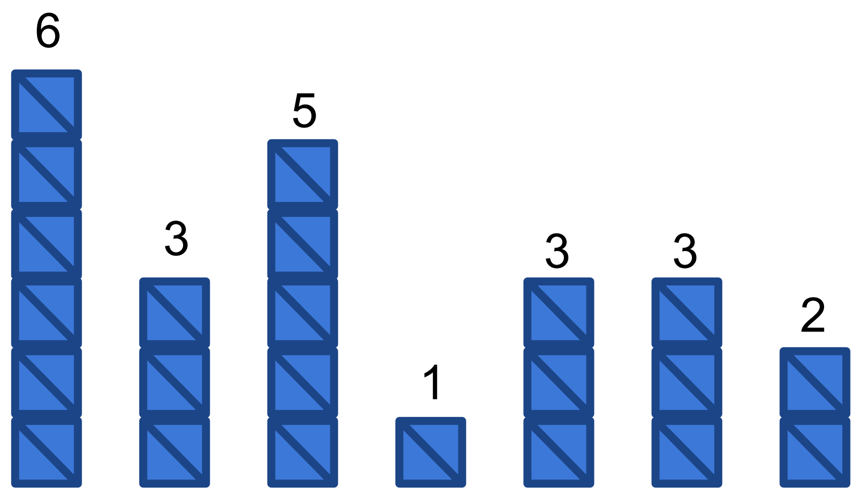
Array Resizing: Final Comment

- Strings, Vectors, and other “auto-resizing” containers may reallocate their memory and move their data
    - When this happens, pointers to their data (such as those from .c_str()) are invalidated!
    - I.e., if we store a pointer or an iterator to vec[10] and the vector resizes, the pointer is invalidated
- However, if you do not change a vector’s size or capacity throughout its lifetime, pointers to its elements will not be invalidated
    - So, if you know the vector’s size in advance, set it using resize or reserve and then don’t change it!

Example #3 - Box Stacking

Your warehouse gets into the *box stacking* business.

You have various towers of boxes; each has a *height* (how many boxes it contains).

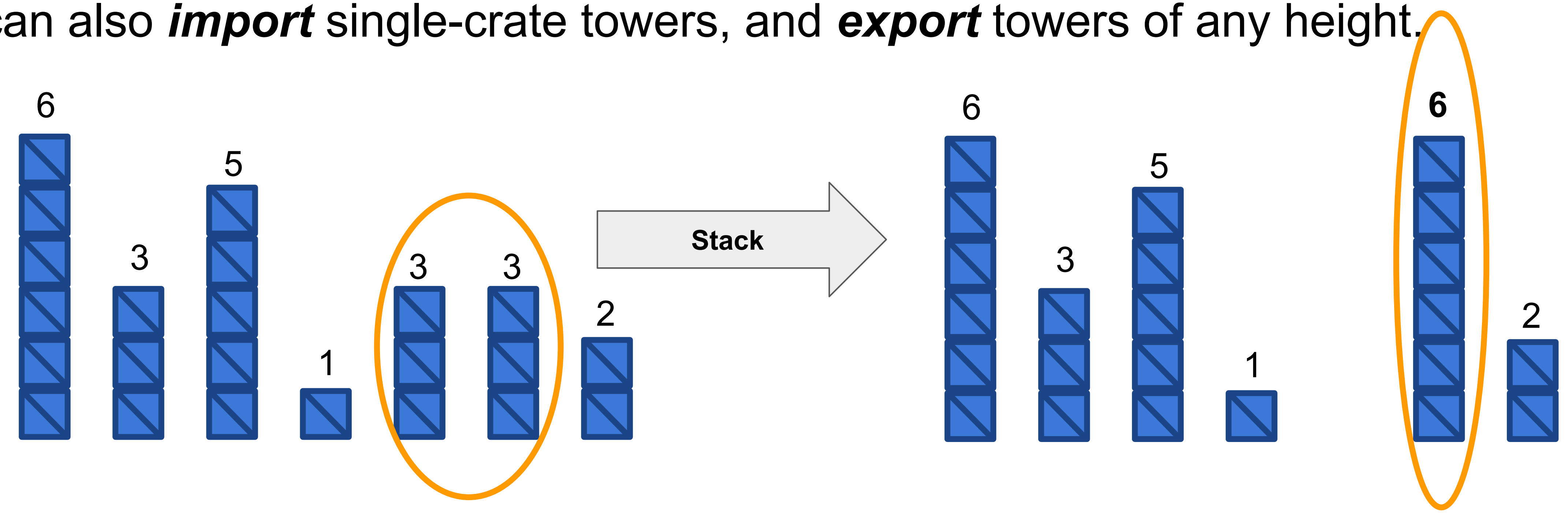


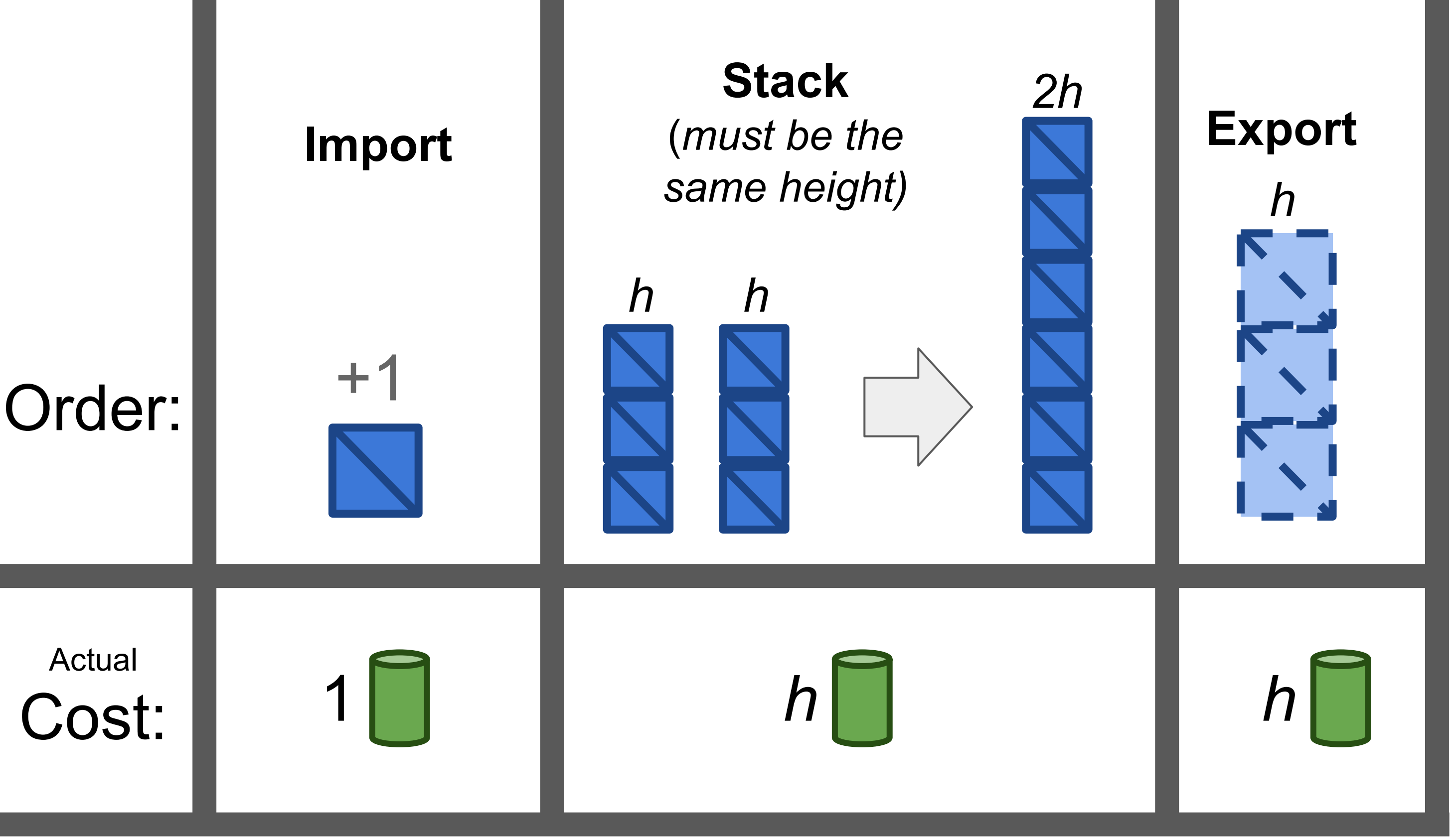
Your warehouse gets into the *box stacking* business.

You have various towers of boxes; each has a *height* (how many boxes it contains).

Two towers can be ***stacked*** if they have the *same height*, producing a new tower that is twice as tall:

We can also ***import*** single-crate towers, and ***export*** towers of any height.





Every morning, the warehouse begins empty. You then receive ***k*** orders. Each order will either be to import one crate, export one tower, or combine two towers of the same height together.

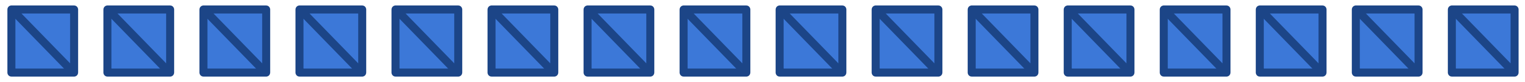
In terms of k, what is the maximum amount of fuel that you can spend in one day?

hk ?   How does *h* relate to *k*?

How does *h* relate to *k*?

h can get pretty big:

perform *k*/2 **import** operations



How does *h* relate to *k*?

h can get pretty big:

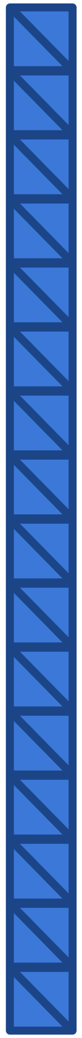
perform *k/2* **import** operations
perform *k/4* **stack(1)** operations
perform *k/8* **stack(2)** operations
perform *k/16* **stack(4)** operations
…
perform *one* **stack(k/2)** operation

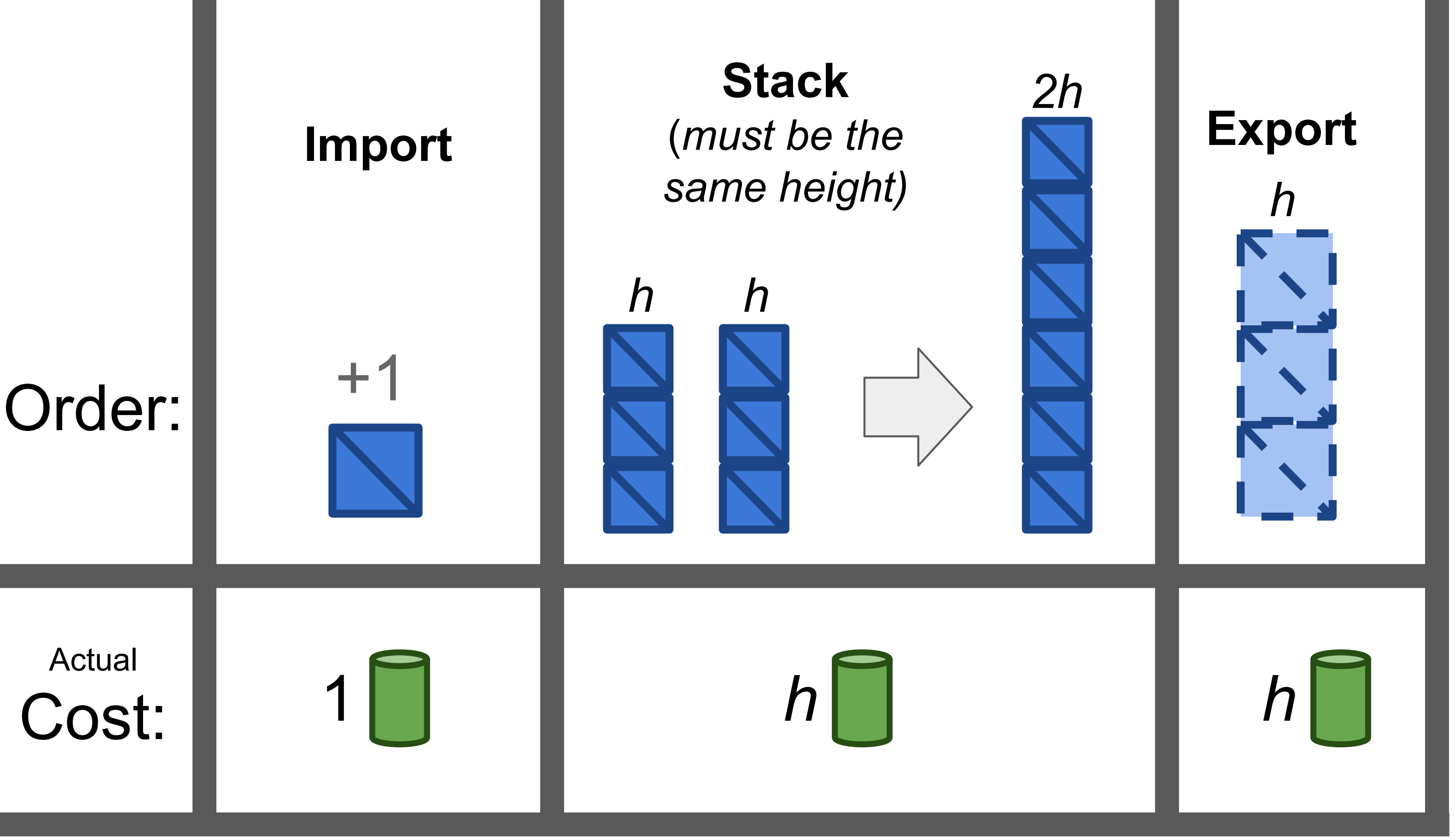
Height: *k/2*
Total Operation Count:

$k/2 + k/4 + k/8 + \ldots + 1 = k(\frac{1}{2} + \frac{1}{4} + \ldots)$

$= k$

Thus, ***h* ≤ *k/2***.





$h \le k/2$

Your turn!

Every morning, the warehouse begins empty. You then receive ***k*** orders. Each order will either be to buy one crate, sell one tower, or combine two towers of the same height together.

In terms of k, what is the maximum amount of fuel that you can spend in one day?

if we spend the maximum cost on every box: $k^2/2$ **Can we do better???**

Hint:

Follow one box: how much cost can you attribute to *one crate* over its lifetime?

Amortization: A wrap up

Amortization:

- I’m measuring the total cost of a sequence of operations.
- Some of my operations are expensive, the majority of my operations are cheap. So multiplying the largest cost by the number of operations gives a cost that is **too high**.
- When the excess from the expensive (the difference between the expensive and cheap operations) is averaged out over the cheap operations, I find that the total cost is essentially constant.

C-Strings vs C++ Strings

C-Strings

- `char str[] = “HELLO\n”;`
- What is the length of this C-string?

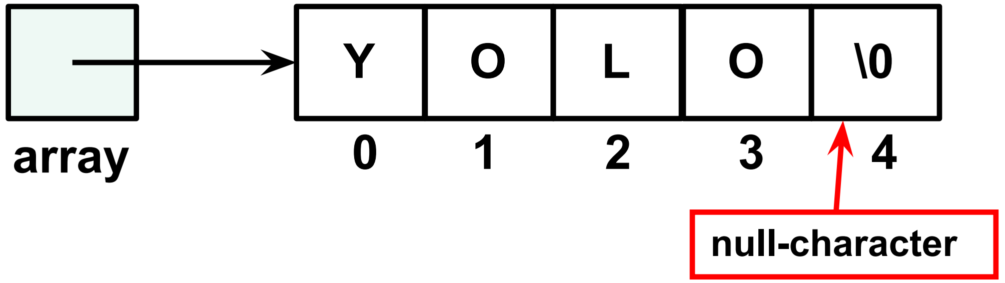
C-Strings

- `char str[] = “HELLO\n”;`
- What is the length of this C-string? 6 (h, e, l, l, o, newline)
- A C-string is as long as the number of characters between the beginning of the string and the terminating null character, *without* including the terminating null character itself

C-Strings

- Array of characters terminated by null-character

```
const char* array = "YOLO";
```
Or
```
char array[] = "YOLO";
```
Or
```
char* array = new char[5];
array[0] = 'y';
array[1] = 'o';
array[2] = 'l';
array[3] = 'o';
array[4] = '\0';
```



- C's Standard Library has many functions for C-Strings (<cstring> in C++)

▪Copying:	`strcpy, strncpy`	Searching:	`strchr, strcspn, etc.`
▪Concatenation:	`strcat, strncat`	Comparison:	`strcmp, strncmp, etc.`

C++ String

- Full-fledged object, defined in <string> of STL

```
string str1 = "YOLO";       // str1 contains "YOLO"
string str2("YOLO");        // str2 contains "YOLO"
string str3("YOLO", 2, 1); // str3 contains "L"
```

- STL has many member functions for C++ strings

Iterators:	`begin, end, etc.`
Capacity:	`size, length, resize, reserve, clear, etc.`
Element Access:	`operator[], at, back, front`
Modifiers:	`operator+=, append, push_back, etc.`
Operations:	`c_str, get_allocator, find, etc.`

C-String vs. C++ String

- Use of functions on strings
  - C-String (function call with string passed as function argument)

```
char str[] = "YOLO";
cout << strlen(str); // prints 4
```

  - C++ String (dot operator for member function)

```
string str("YOLO");
cout << str.length(); // prints 4
```

C-String vs. C++ String

- **In general, use C++ Strings over C-Strings**
  - Encapsulation
    - Size is stored - no need to keep track of null-terminator.
  - More fully featured: iterators, easy growth syntax, safety with getline, to_string
  - Some C++ string functions are faster than C-String counterpart
    - Example: strlen() vs. .length()

Project 1 Tips

- Write modular code (i.e. separate you code into functions!)
- Think hard about data structures
  - This project is strict on memory!

- Look at Piazza posts
- Submit ASAP
  - Students often take 10+ submits to get their desired score
- Pass anything other than basic types (int, char, bool) by reference

Project 1 Tips cont.

- Break up the work into parts, so it’s not so overwhelming
  - Makefile
  - Getopt
  - Storing the input dictionary
  - Routing
  - Backtracking
  - First output mode
  - Second output mode

Steps for Solving Coding Problems

1. **Read carefully and pay attention to problem specifics**
   - Typically every detail of the problem description is needed to come up with the optimal solution
2. **Come up with a good example and use it to inform your algorithm design**
   - Make sure this example is fairly large and generic
   - Walk through your algorithm and run it on this example BEFORE you start coding
3. **Code your algorithm**
   - Make sure to write neatly with clear indentations and appropriate variable names
   - Modularize your code where appropriate
4. **Test your solution with multiple small examples**
   - Start with a generic example and then test edge cases
   - Make sure to be thorough – go line by line and actually test your code, not just your algorithm!

Handwritten Problem - Anagrams

Write a function that takes in two strings and returns if they are anagrams (contain the same letters) of each other. The only characters will be spaces and lowercase letters. This can be done in O(n) time.

Example 1: Given s1 = "anagram" and s2 = "nagaram".

Return true.

Example 2: Given s1 = "i love eecs" and s2 = "i scole ve e".

Return true.

Example 3: Given s1 = “anagrams” and s2 = “anagrams anagrams”.

Return false.

```
bool isAnagram(string s1, string s2);
```