CSE 250
Data Structures

Dr. Eric Mikida
epmikida@buffalo.edu
208 Capen Hall

Lec 10: ArrayList and Amortized Runtime

Announcements

- PA1 Implementation due Sunday, 2/18 @ 11:59PM
  - Continue with the same repo you've been using
- WA2 will be released after the PA1 deadline, due 2/25 @ 11:59PM

The List ADT

```
public interface List<E>
    extends Sequence<E> { // Everything a sequence has, and...
  /** Extend the sequence with a new element at the end */
  public void add(E value);

  /** Extend the sequence by inserting a new element */
  public void add(int idx, E value);

  /** Remove the element at a given index */
  public void remove(int idx);
}
```

List Runtimes (so far...)

	ArrayList	Linked List (by index)	Linked List (by reference)
`get(...)`	$\Theta(1)$	$\Theta(\text{idx})$ or $O(n)$	$\Theta(1)$
`set(...)`	$\Theta(1)$	$\Theta(\text{idx})$ or $O(n)$	$\Theta(1)$
`size()`	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
`add(...)`	TBD	$\Theta(\text{idx})$ or $O(n)$	$\Theta(1)$
`remove(...)`	TBD	$\Theta(\text{idx})$ or $O(n)$	$\Theta(1)$

ArrayList Changes

How can we implement `add`/`remove` for something like an Array?

Keep extra space and track how many elements you have

ArrayList Representation

```
class ArrayList<T> extends List<T> {
  private int used;
  private Optional<T>[] data;

  public int size() { return used; }

  /* ... */
}
```

ArrayList Representation

```java
public T get(int i) {
  if (i < 0 || i >= used) { throw new IndexOutOfBoundsException(i); }
  return data[i].get()
}

public void set(int i, T value) {
  if (i < 0 || i >= used) { throw new IndexOutOfBoundsException(i); }
  data[i] = Optional.of(value);
}
```

ArrayList Representation

The methods from the Sequence ADT are all still $\mathbf{\Theta(1)}$

What about **`remove`**?

ArrayList - remove(i)

```
public void remove(int i) {
  /* Sanity-check inputs */
  if (i < 0 || i >= used) { throw new IndexOutOfBoundsException(i); }

  /* Shift elements left */
  for (int j = i; j < used - 1; j++) {
    data[j] = data[j+1];
  }
  data[used-1] = Optional.empty();
  used--;
}
```

ArrayList - remove(i)

```
public void remove(int i) {
  /* Sanity-check inputs */
  if (i < 0 || i >= used) { throw new IndexOutOfBoundsException(i); }

  /* Shift elements left */
  for (int j = i; j < used - 1; j++) {
    data[j] = data[j+1];
  }
  data[used-1] = Optional.empty();
  used--;
}
```

Have to shift over right-most elements to fill the hole created by the removed element!

Analysis of `remove(i)`

$$T_{\text{remove}}(n) = \begin{cases} 1 & \text{if i = used - 1} \\ 2 & \text{if i = used - 2} \\ 3 & \text{if i = used - 3} \\ \dots & \dots \\ n-1 & \text{if i = 0} \end{cases}$$

Analysis of `remove(i)`

$$T_{\text{remove}}(n) = \begin{cases} 1 & \text{if i = used - 1} \\ 2 & \text{if i = used - 2} \\ 3 & \text{if i = used - 3} \\ \ldots & \ldots \\ n-1 & \text{if i = 0} \end{cases}$$

$$T_{\text{remove}}(n) \in O(n), \Omega(1)$$

ArrayList - `add(elem)`

Basic Idea:

1. If we are out of space, first create a new array of a larger size and copy the old elements over
2. Then, set the element at index `used` to `elem`

ArrayList - add(elem)

```
public void add(T elem) {
  if(used == data.length) { /* Sad case 🙁 */
    int newLength = ???
    Optional<T>[] newData = new Optional<T>[newLength];
    System.arrayCopy(data, 0, newData, 0, data.length);
    data = newData;
  }
  /* Happy case 😃 */
  data[used] = Optional.of(elem)
  used++;
}
```

ArrayList - add(elem)

```java
public void add(T elem) {
  if(used == data.length) { /* Sad case 🙁 */
    int newLength = ???
    Optional<T>[] newData = new Optional<T>[newLength];
    System.arrayCopy(data, 0, newData, 0, data.length);
    data = newData;
  }
  /* Happy case 😃 */
  data[used] = Optional.of(elem);
  used++;
}
```

How we choose the new length will be important!

ArrayList - `add(elem)`

```java
public void add(T elem) {
  if(used == data.length) { /* Sad case 🙁 */
    int newLength = ???
    Optional<T>[] newData = new Optional<T>[newLength];
    System.arrayCopy(data, 0, newData, 0, data.length);
    data = newData;
  }
  /* Happy case 😃 */
  data[used] = Optional.of(elem);
  used++;
}
```

This is the expensive part!

Analysis of `add(elem)`

$$T_{\text{add}}(n) = \begin{cases} 1 & \text{if used} < \text{data.length} \\ n & \text{if used} = \text{data.length} \end{cases}$$

$$T_{\text{add}}(n) \in O(n), \Omega(1)$$

Analysis of `add(elem)`

$$T_{\text{add}}(n) = \begin{cases} 1 & \text{if used} < \text{data.length} \\ n & \text{if used} = \text{data.length} \end{cases}$$

$$T_{\text{add}}(n) \in O(n), \Omega(1)$$

How often do our **add** calls require ***O*(*n*)** time?

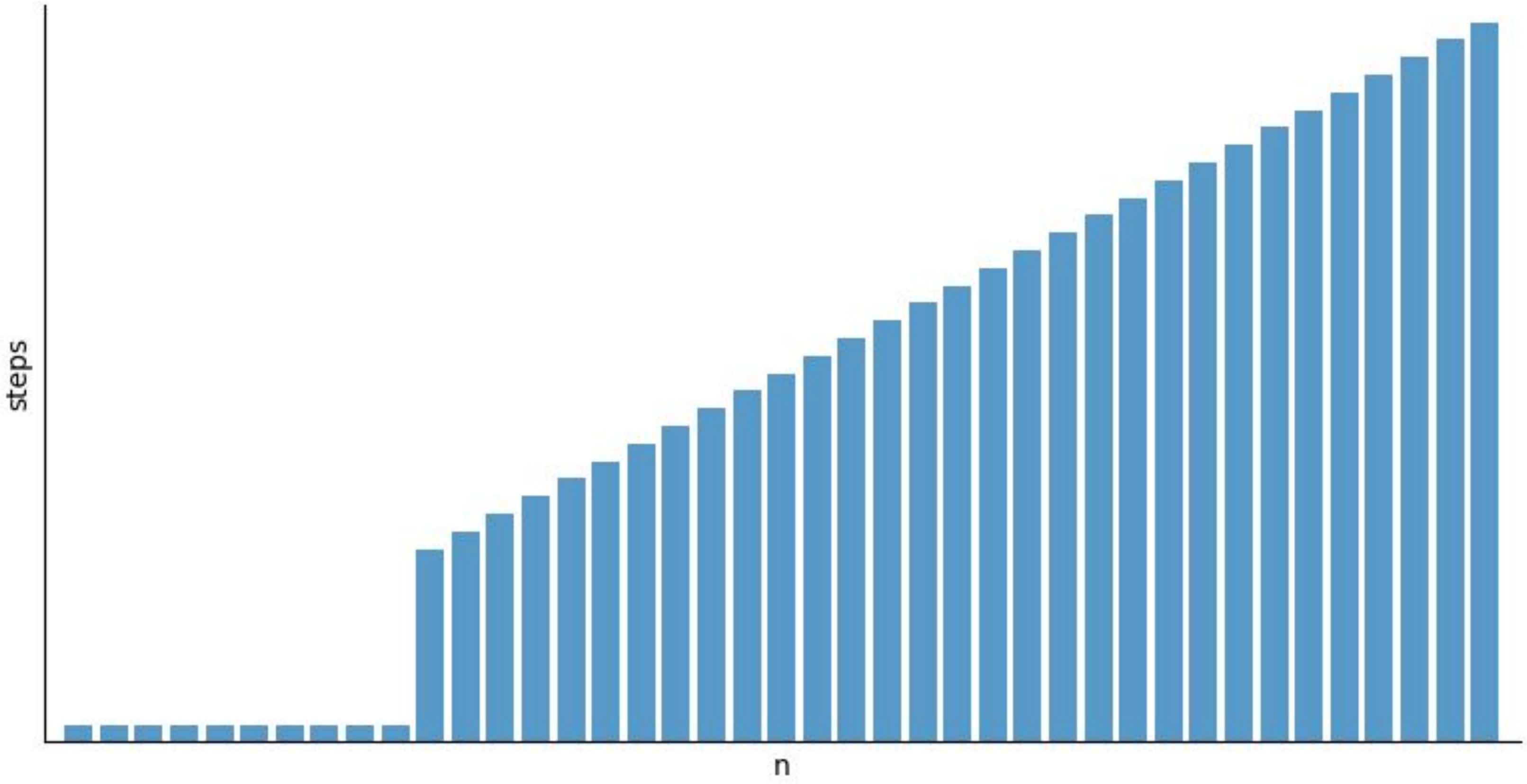
Analysis of `add(elem)`

$$T_{\text{add}}(n) = \begin{cases} 1 & \text{if used} < \text{data.length} \\ n & \text{if used} = \text{data.length} \end{cases}$$

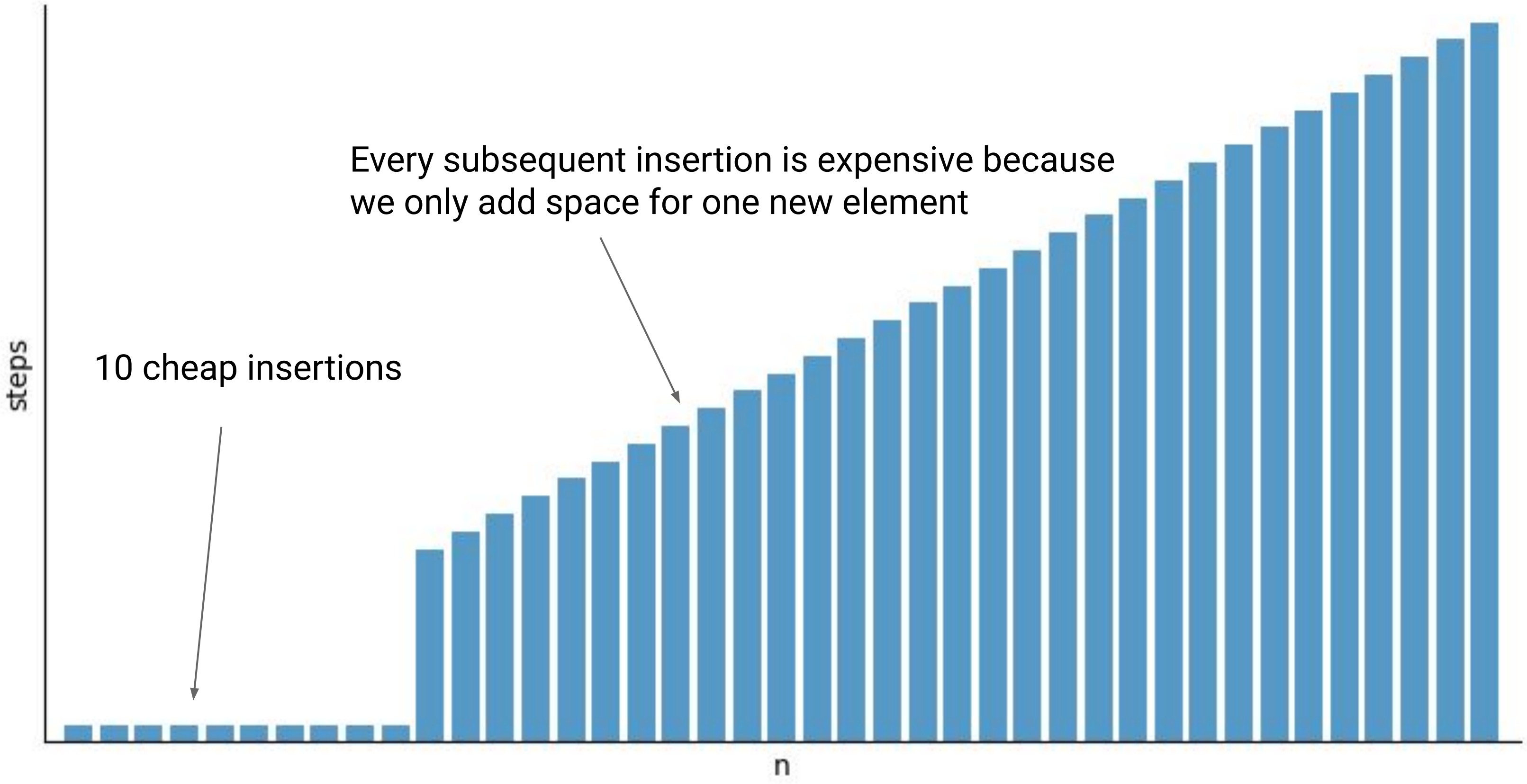
$$T_{\text{add}}(n) \in O(n), \Omega(1)$$

How often do our **`add`** calls require ***O*(*n*)** time?

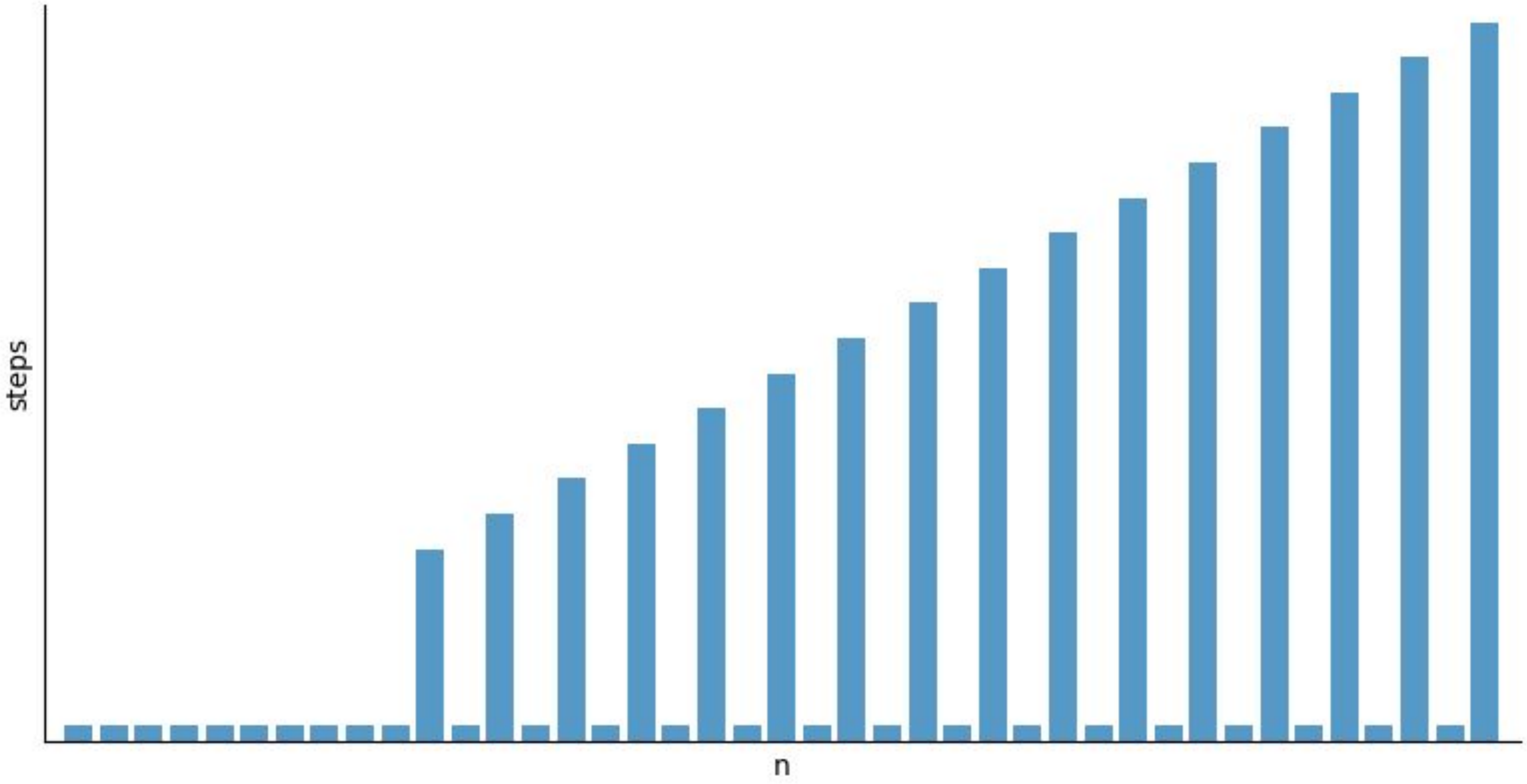
It depends on how we calculate **`newLength`**



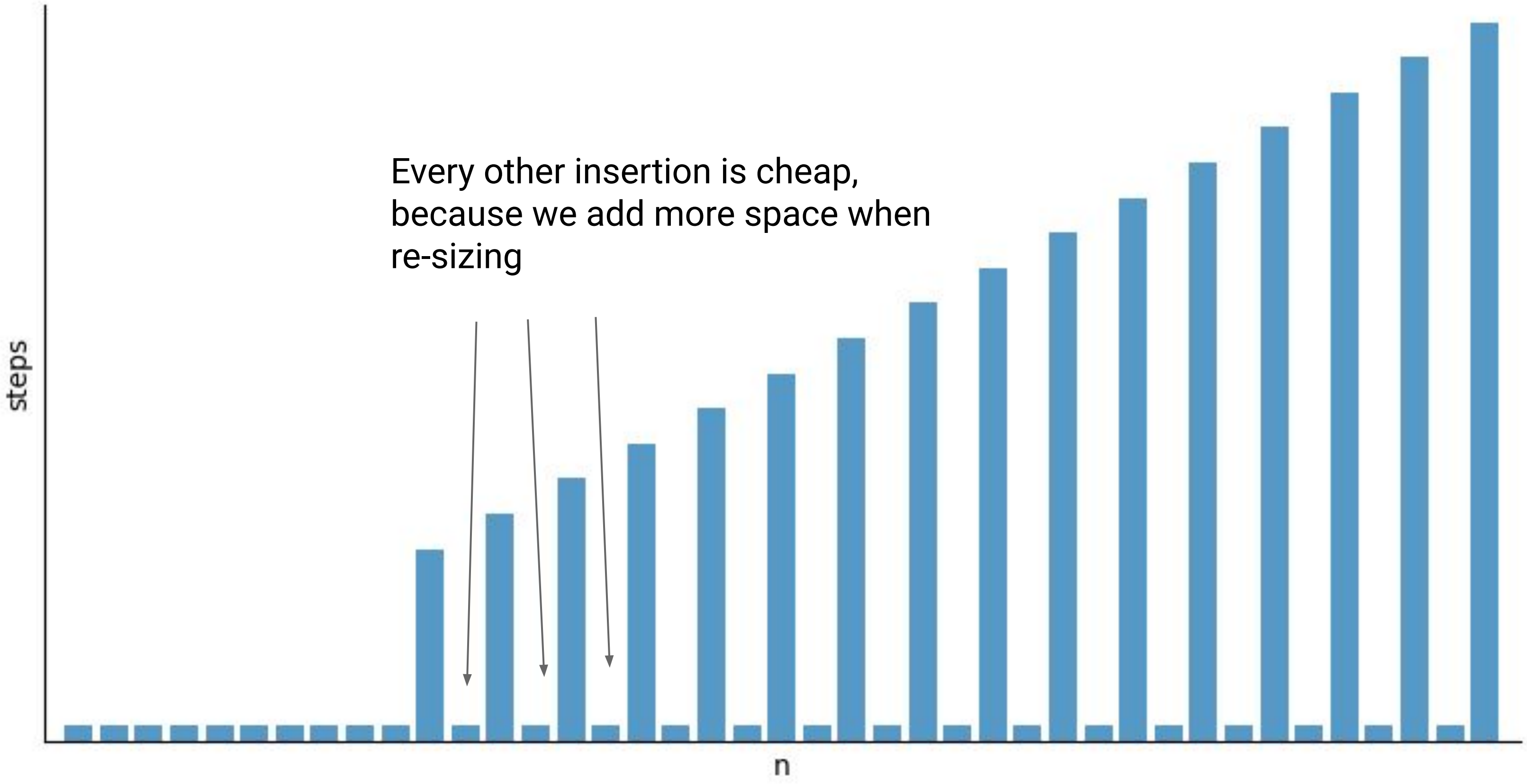
Initial size = 10, newLength = data.length + 1



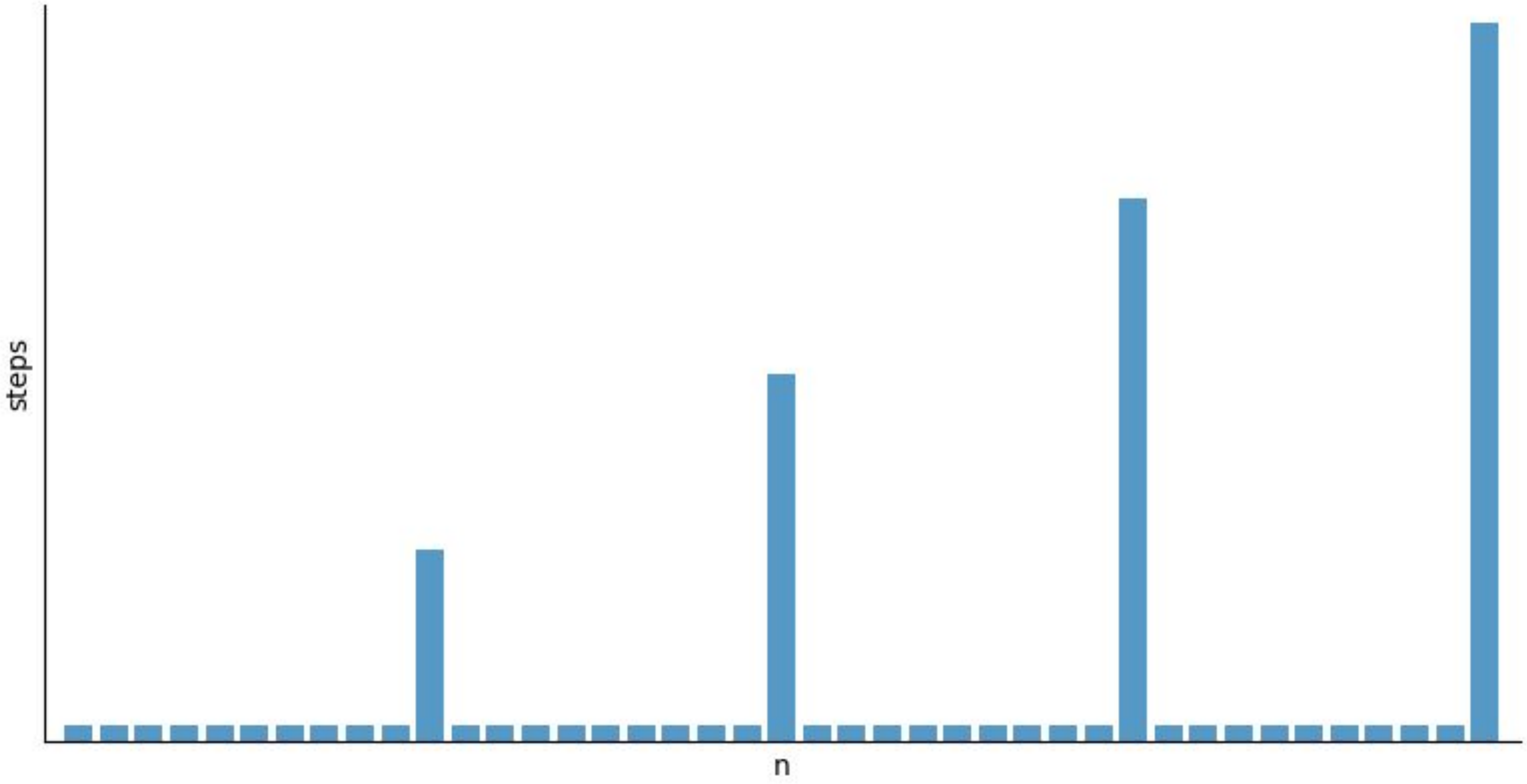
Initial size = 10, newLength = data.length + 1



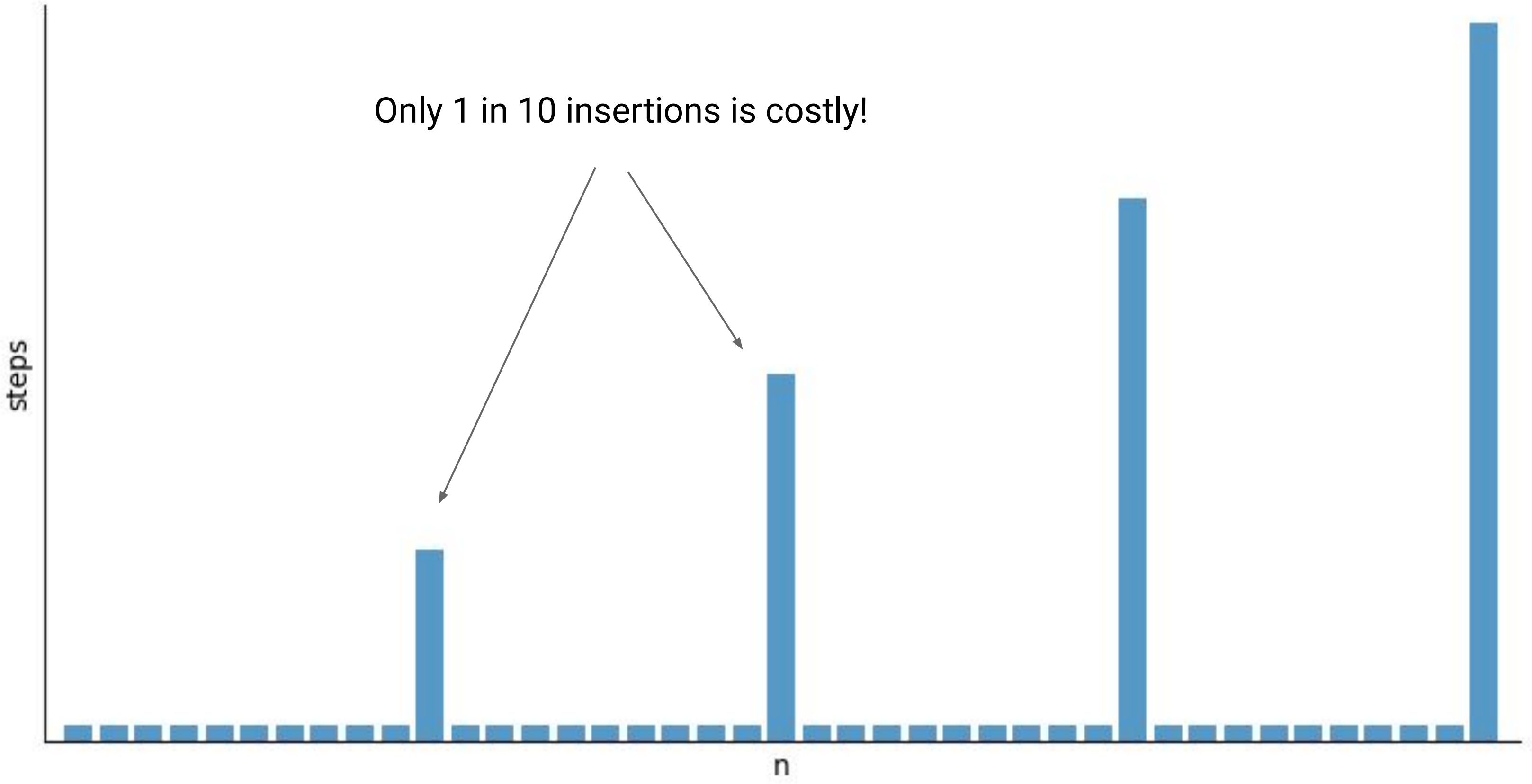
Initial size = 10, newLength = data.length + 2



Initial size = 10, newLength = data.length + 2



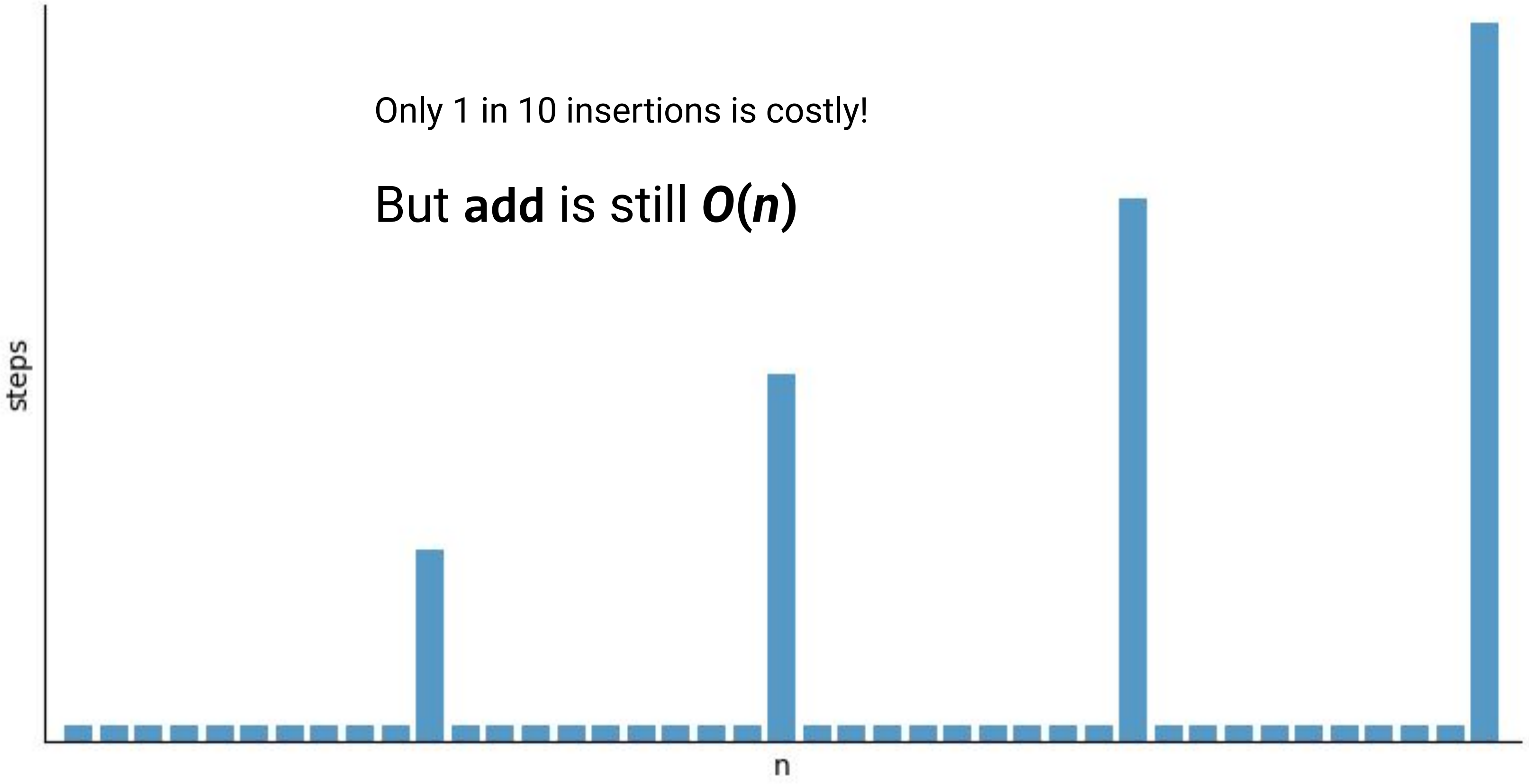
Initial size = 10, newLength = data.length + 10



Initial size = 10, newLength = data.length + 10

Only 1 in 10 insertions is costly!

But **add** is still ***O*(*n*)**



Initial size = 10, newLength = data.length + 10

A Note on Runtime Complexity

So far, when we've discussed runtime bounds we have done so without taking any extra information/context into account.

For example, the worst-case runtime of **`ArrayList.add`** is $\boldsymbol{O(n)}$

Our analysis doesn't capture the fact that oftentimes it is faster than $\boldsymbol{O(n)}$

A Note on Runtime Complexity

So far, when we've discussed runtime bounds we have done so without taking any extra information/context into account.

For example, the worst-case runtime of **`ArrayList.add`** is $O(n)$

Our analysis doesn't capture the fact that oftentimes it is faster than $O(n)$

We refer to this as the <u>unqualified runtime</u>…it is the runtime without any extra qualifications, caveats, etc

A Note on Runtime Complexity

So far, when we've discussed runtime bounds we have done so without taking any extra information/context into account.

For example, the worst-case runtime of `ArrayList.add` is $\boldsymbol{O(n)}$

Our analysis doesn't capture the fact that oftentimes it is faster than $\boldsymbol{O(n)}$

We refer to this as the <u>unqualified runtime</u>…it is the runtime without any extra qualifications, caveats, etc

But, sometimes the extra context can be relevant…
how can we include this in our analysis?

Common Pattern: Repeated Calls

Oftentimes we will want to call a function many times in a row

- ie read through a CSV file and add all records to a List

What can we say about the runtime in this case?

Adding *n* Elements to a LinkedList

```java
List<Integer> list = new LinkedList<Integer>();
for (int i = 0; i < n; i++) {
  list.add(i);
}
```

Adding *n* Elements to a LinkedList

```
Θ(1)
for (int i = 0; i < n; i++) {
  Θ(1)
}
```

Adding *n* Elements to a LinkedList

```
Θ(1)
Θ(n)
```

Total Complexity: Θ(n)

Adding *n* Elements to an ArrayList

```
List<Integer> list = new ArrayList<Integer>();
for (int i = 0; i < n; i++) {
  list.add(i);
}
```

Adding *n* Elements to an ArrayList

```
O(1)
for (int i = 0; i < n; i++) {
  O(n)
}
```

Adding *n* Elements to an ArrayList

```
O(1)
O(n²)
```

Upper Bound: $O(n^2)$

But is this a tight upper bound?

Adding *n* Elements to an ArrayList

$O(1)$

$O(n^2)$

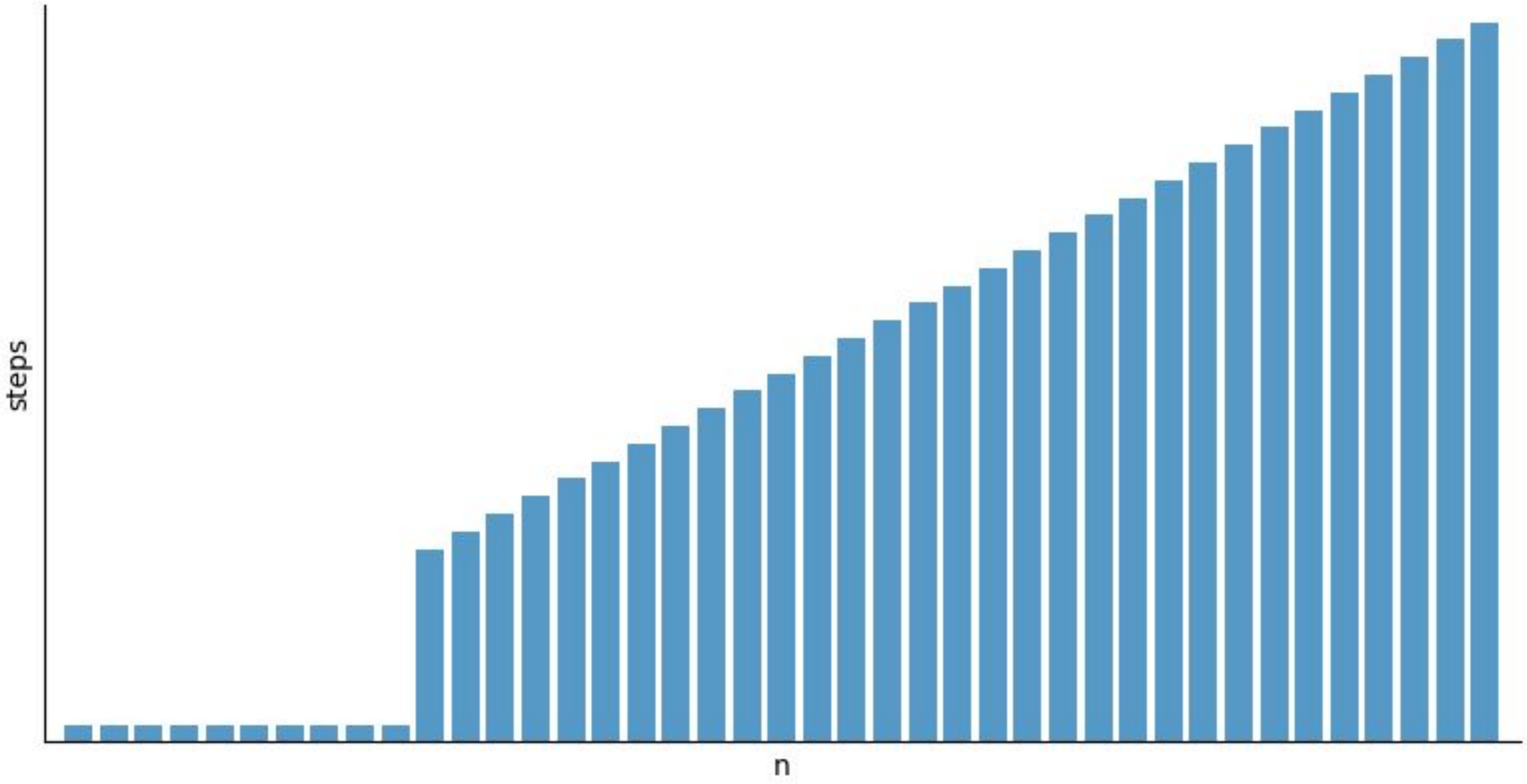
Upper Bound: $O(n^2)$

But is this a tight upper bound? Let's do a more detailed analysis

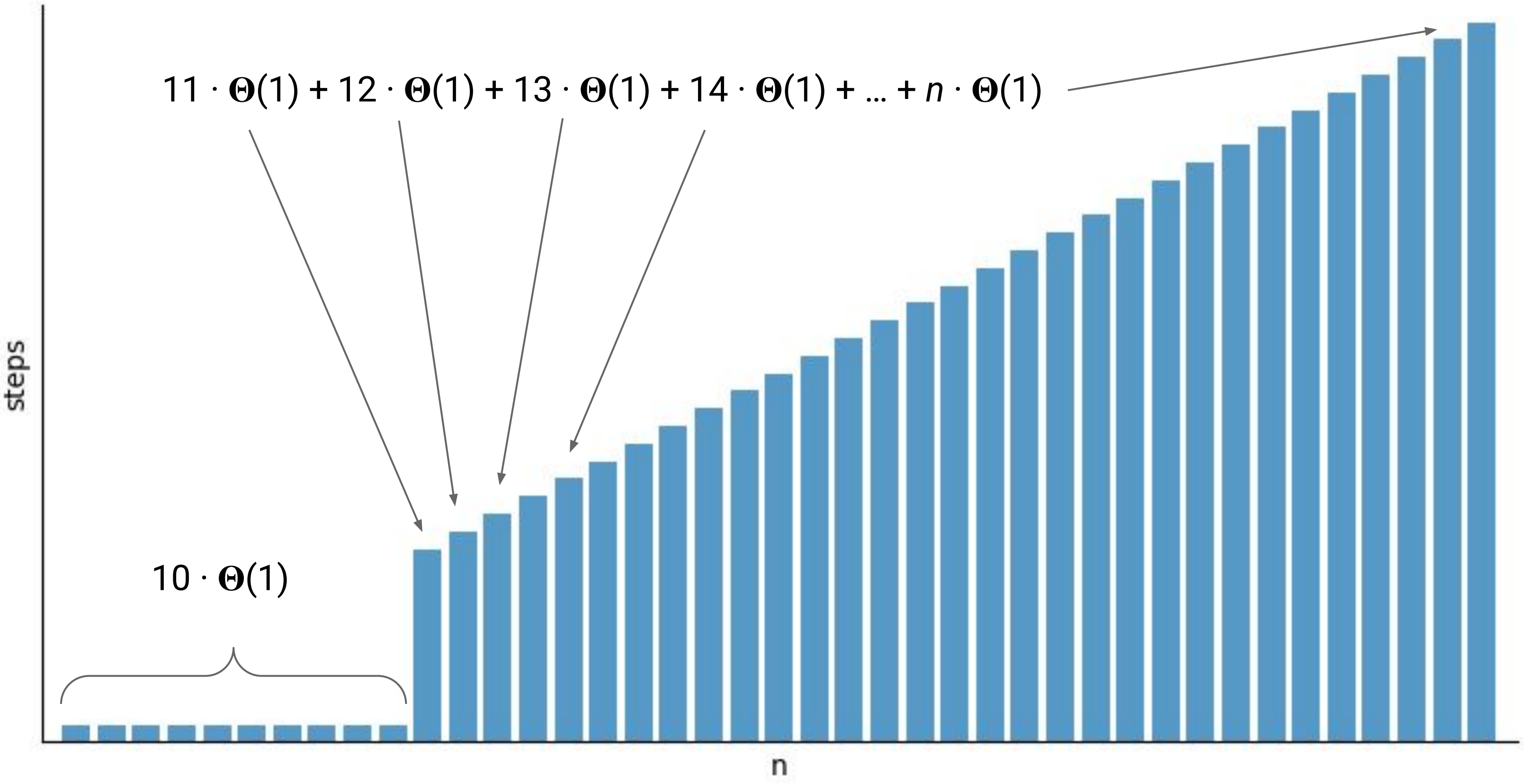
Adding *n* Elements to an ArrayList

```
List<Integer> list = new ArrayList<Integer>();
for (int i = 0; i < n; i++) {
  list.add(i);
}
```

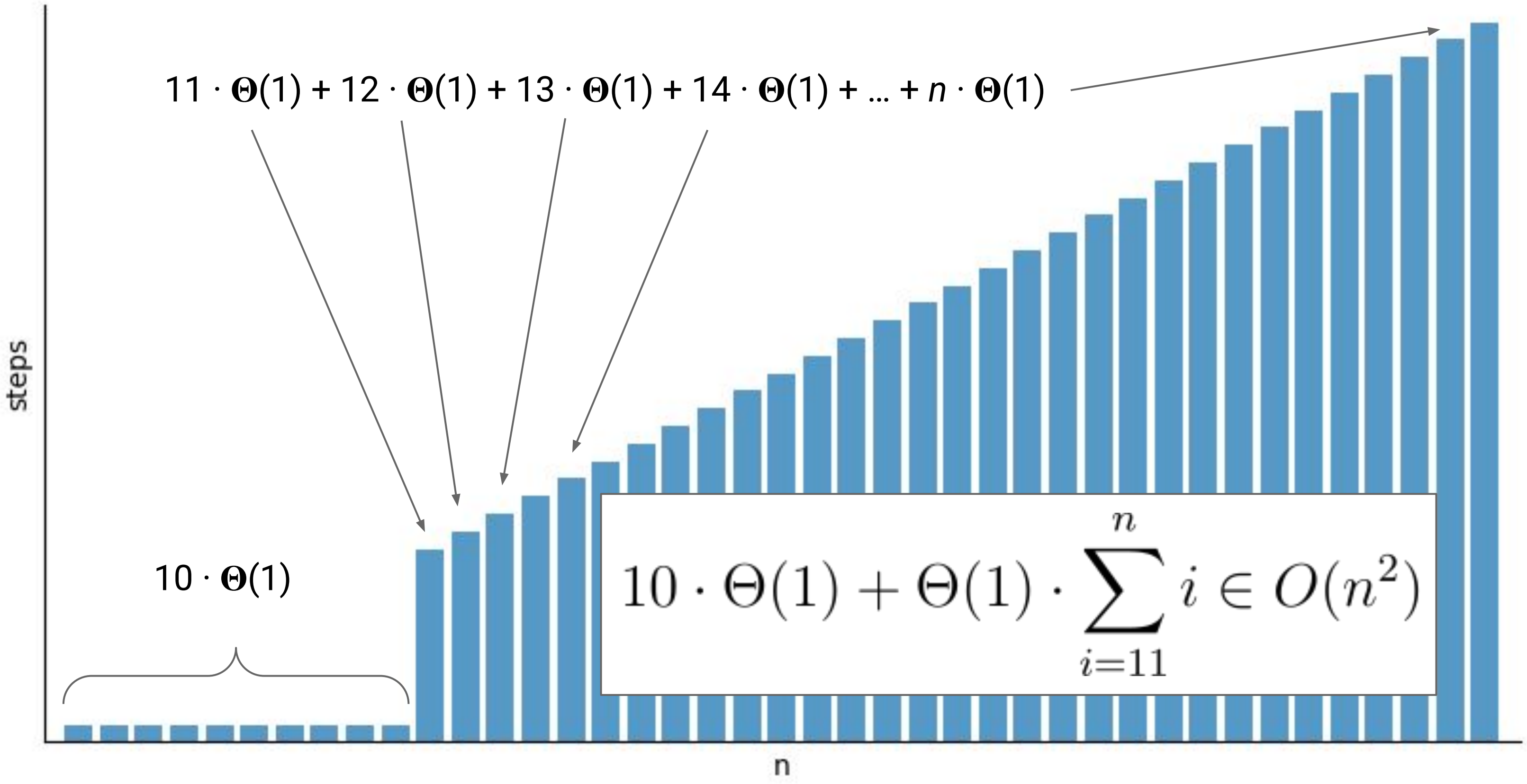
How many steps does this loop do?



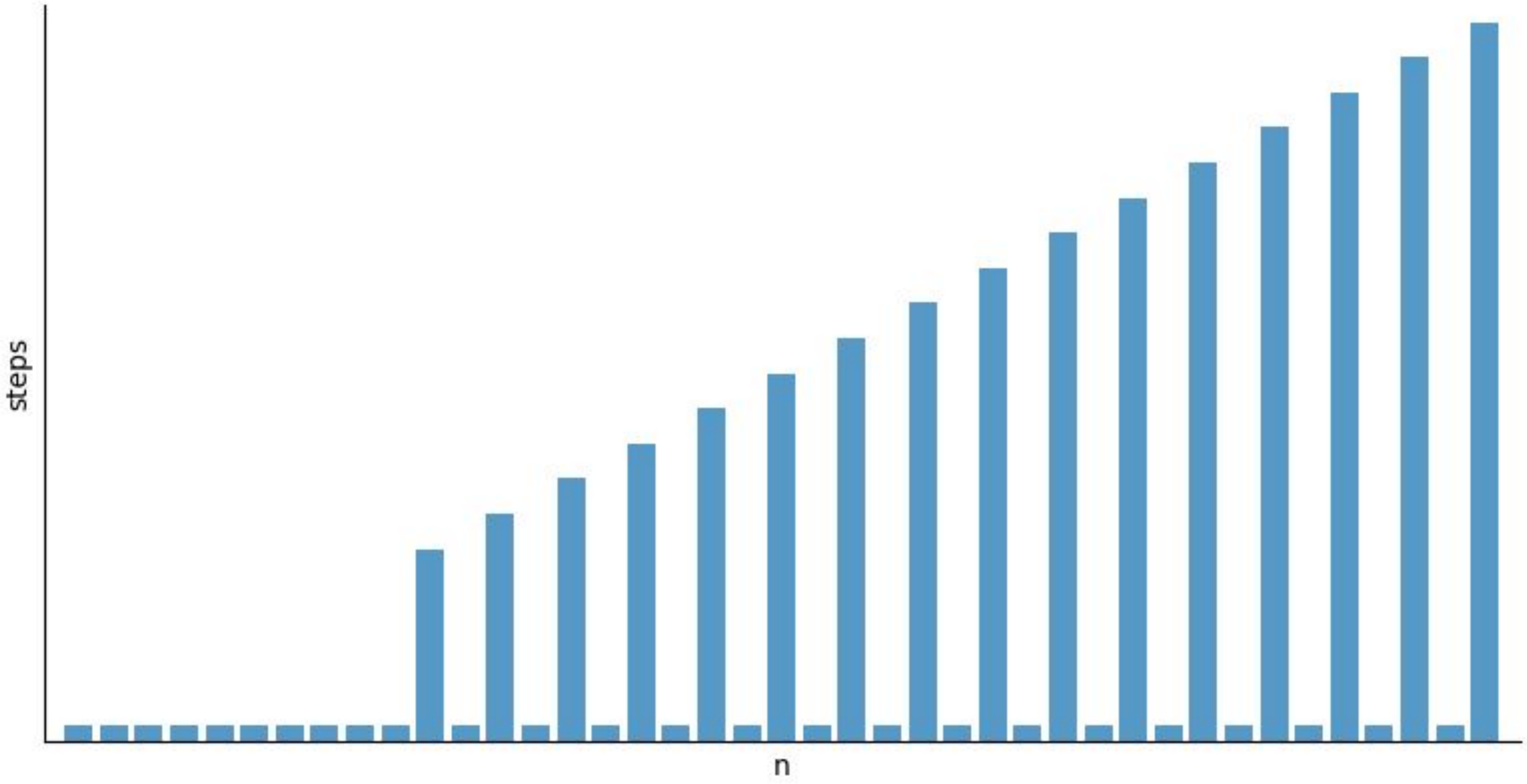
Initial size = 10, newLength = data.length + 1



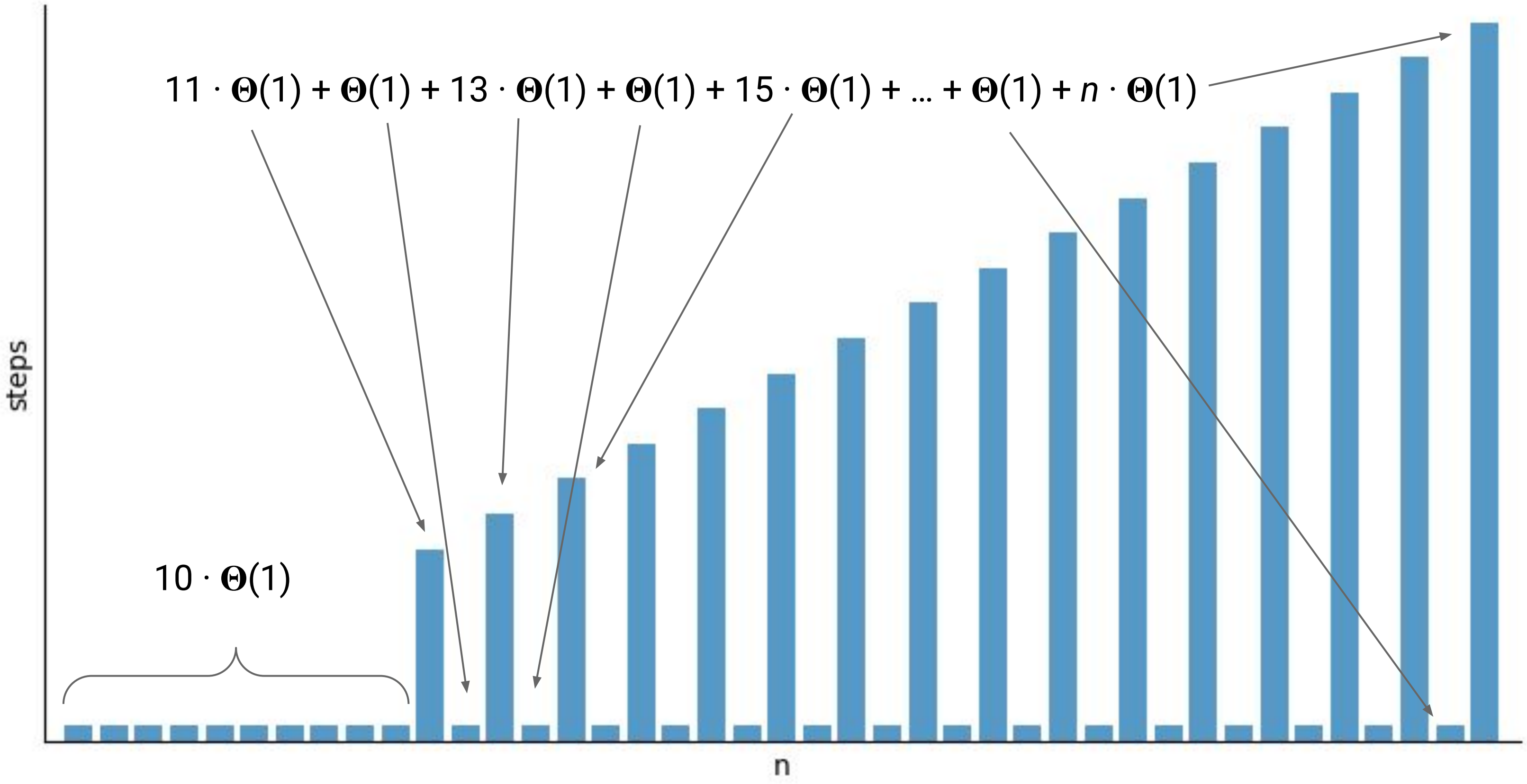
Initial size = 10, newLength = data.length + 1



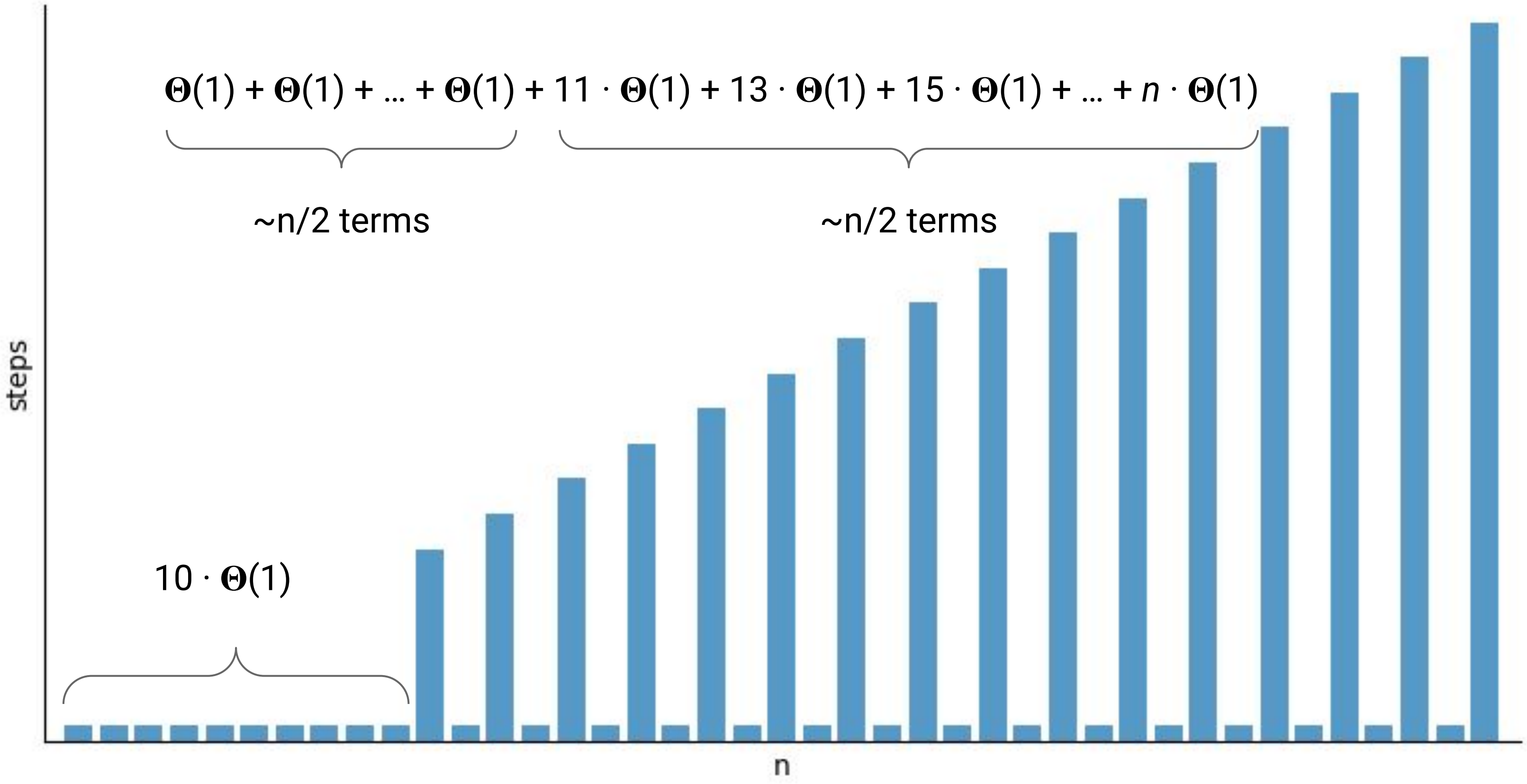
Initial size = 10, newLength = data.length + 1



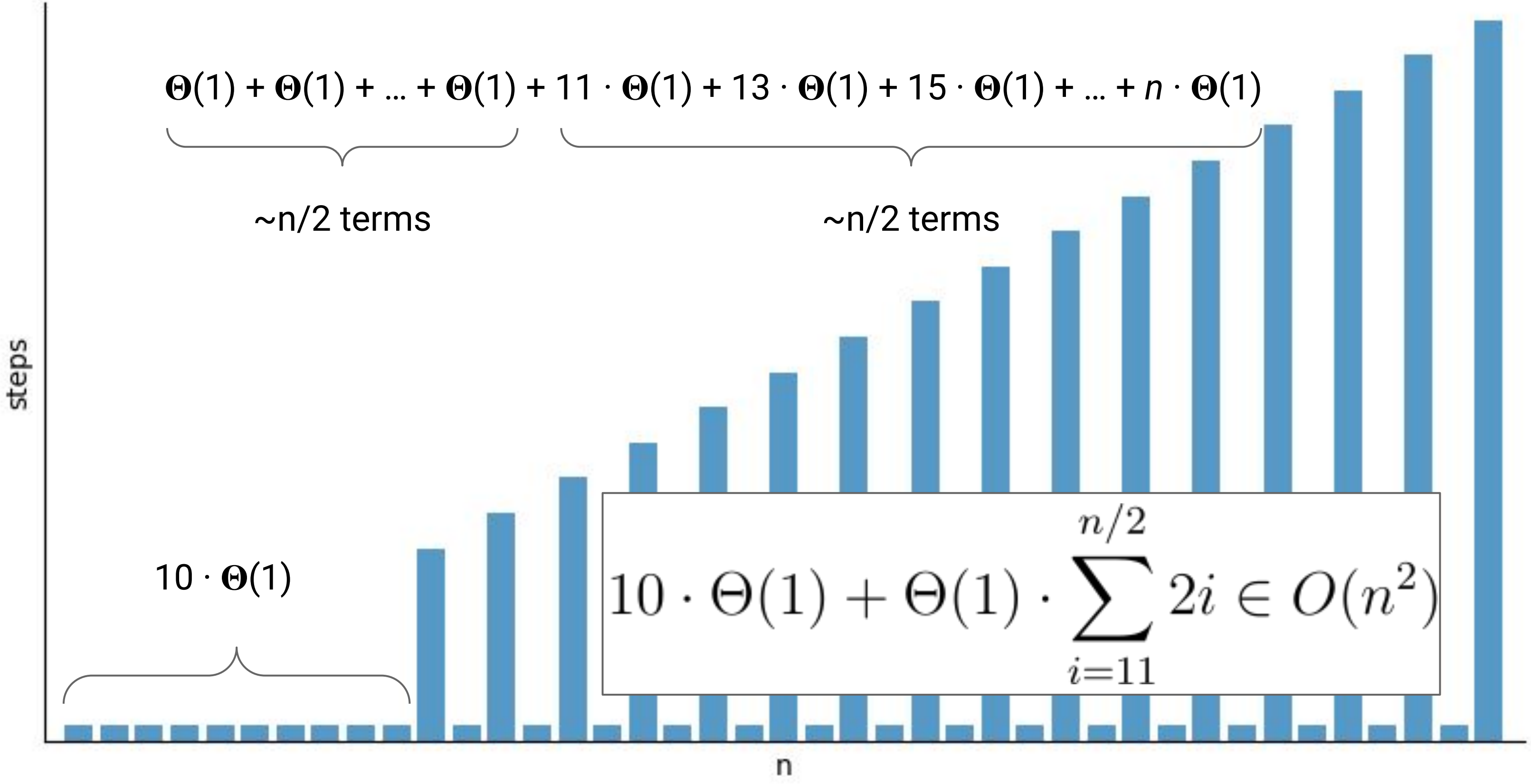
Initial size = 10, newLength = data.length + 2



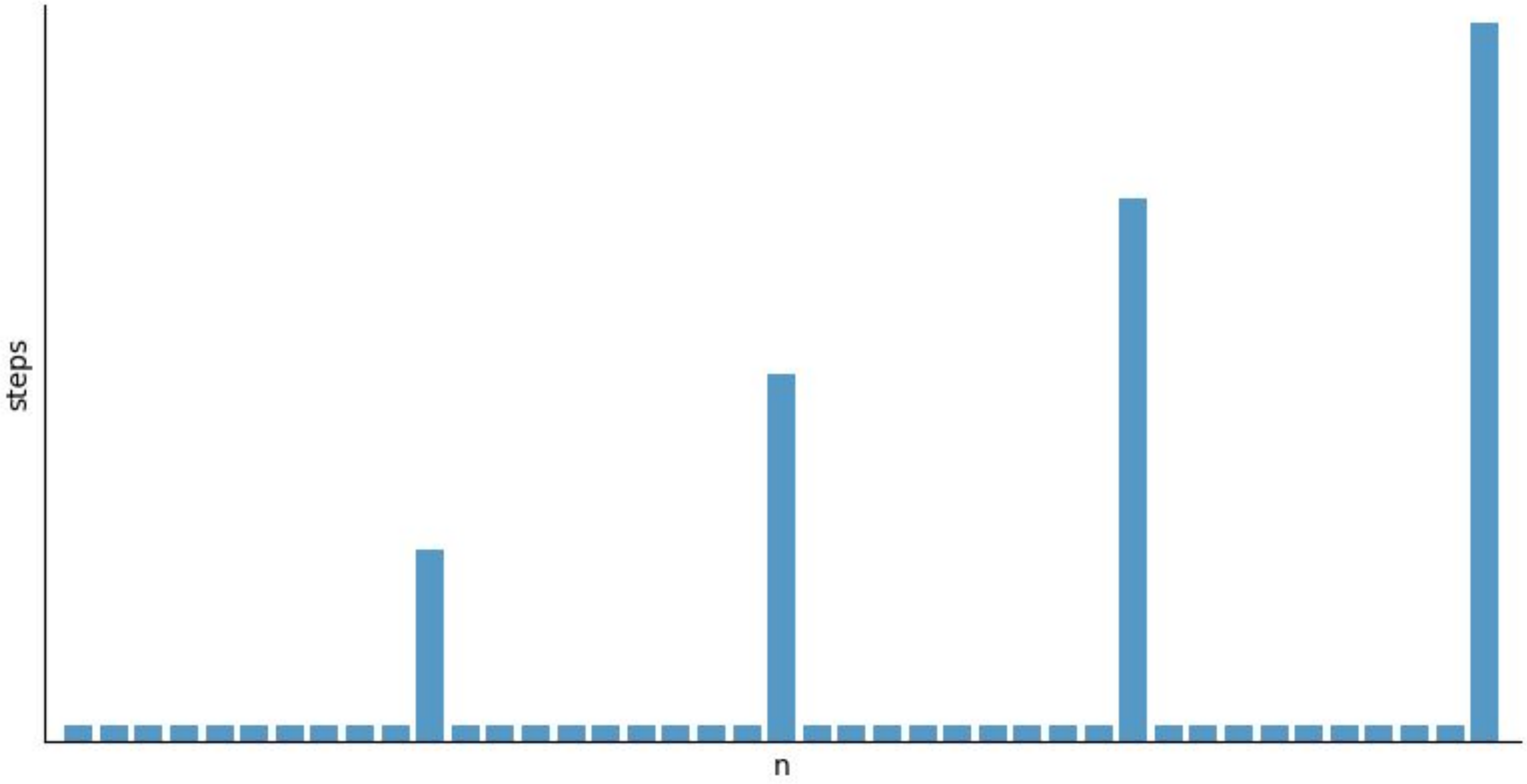
Initial size = 10, newLength = data.length + 2



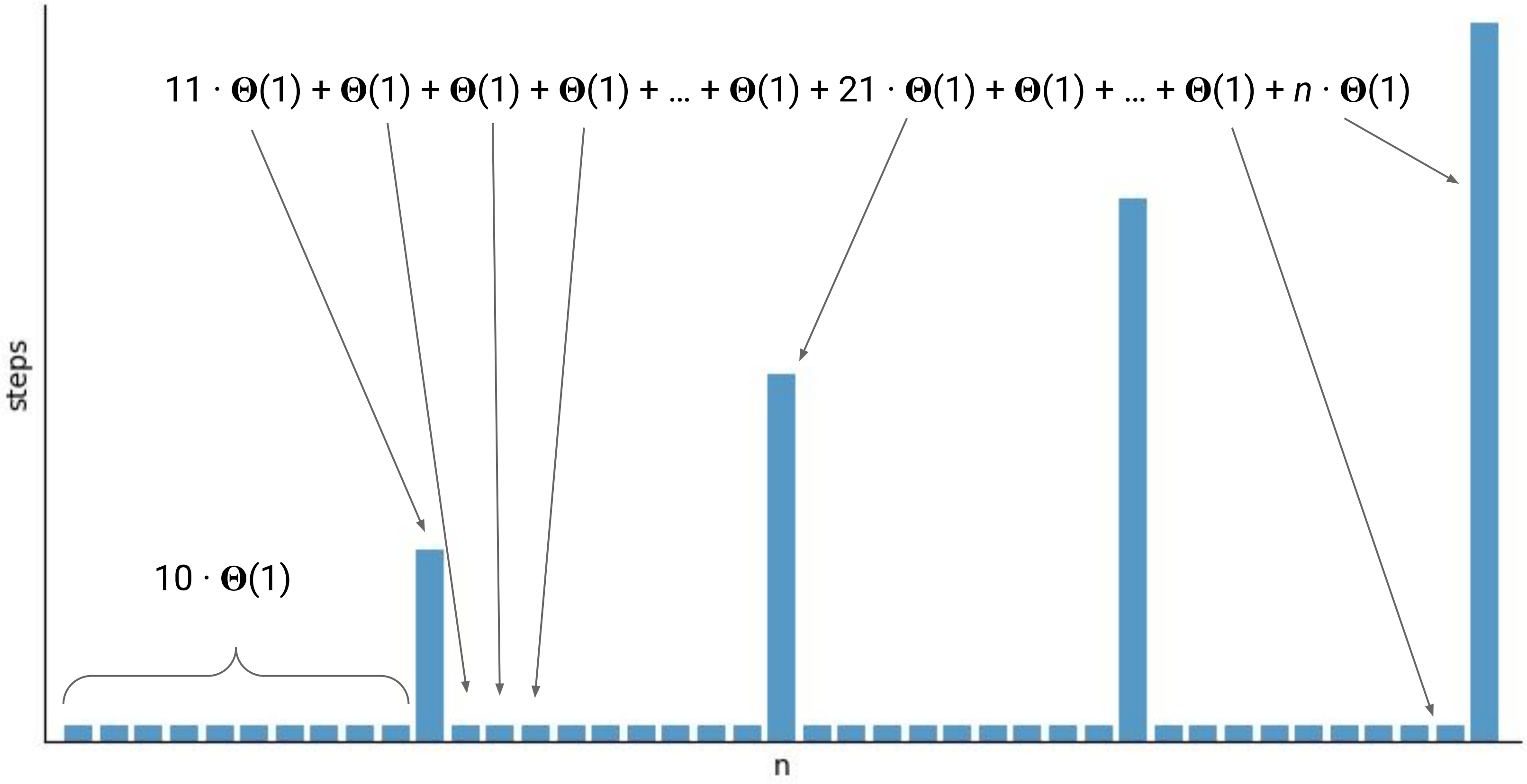
Initial size = 10, newLength = data.length + 2



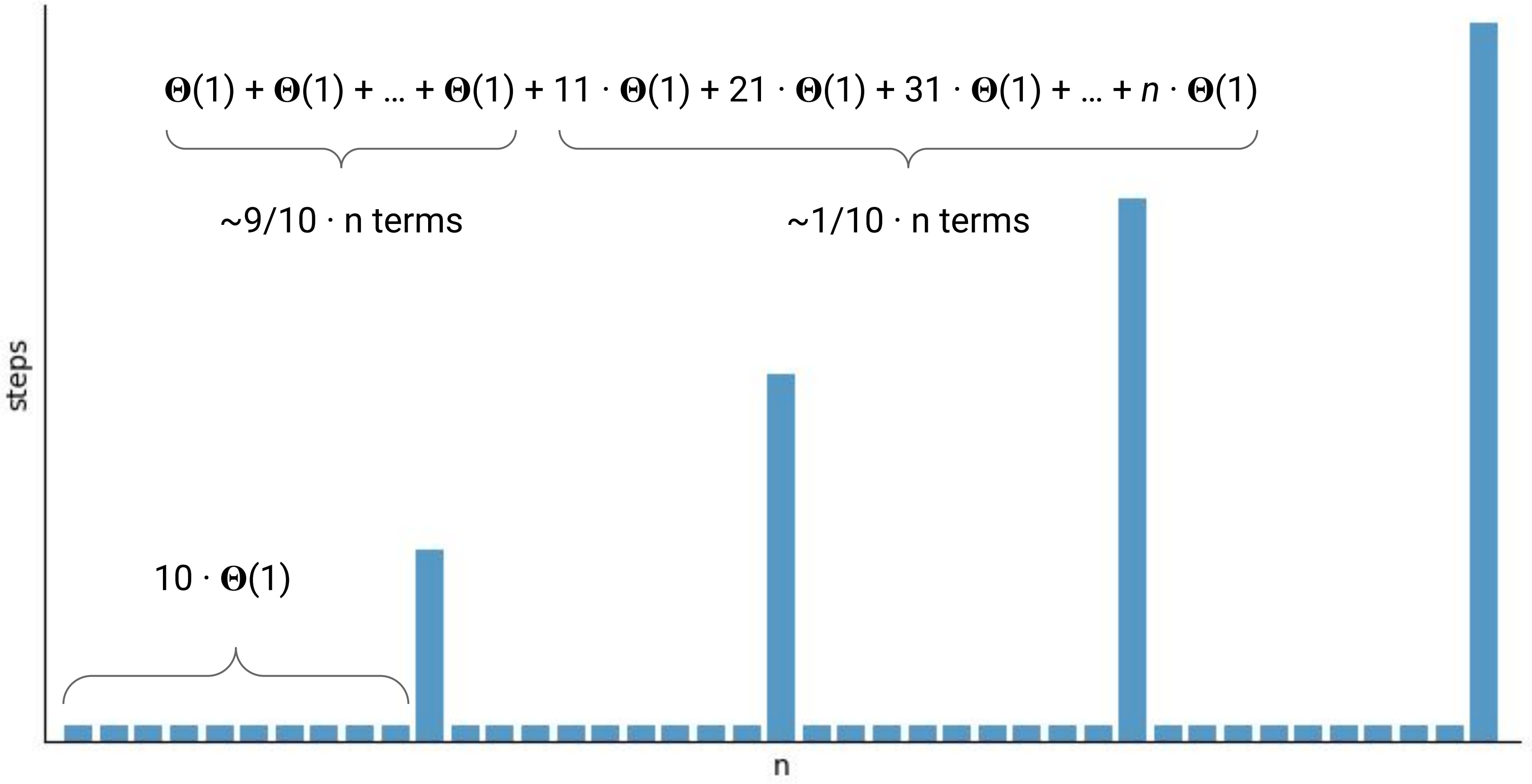
Initial size = 10, newLength = data.length + 2



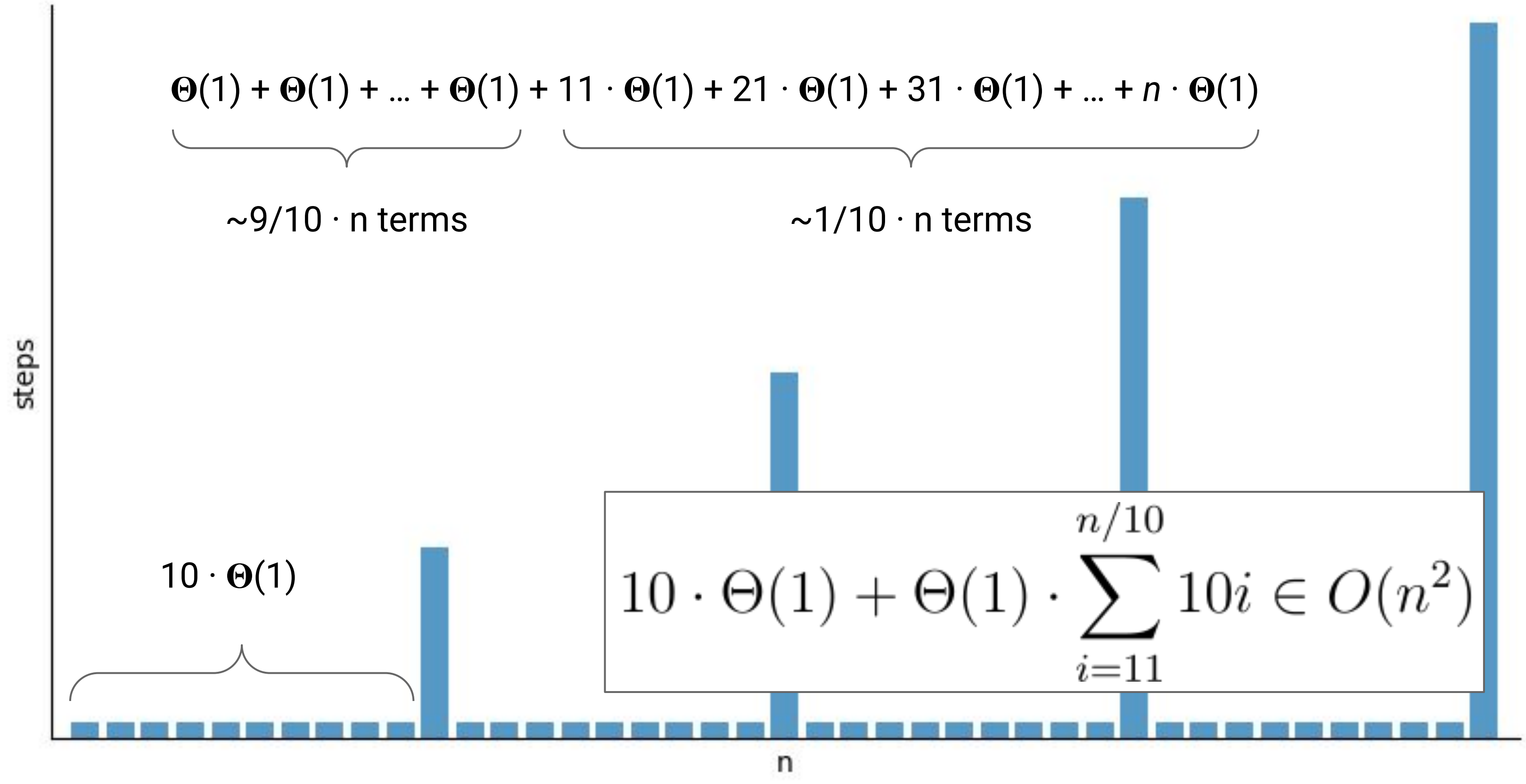
Initial size = 10, newLength = data.length + 10



Initial size = 10, newLength = data.length + 10



Initial size = 10, newLength = data.length + 10



Initial size = 10, newLength = data.length + 10

A New Type of Bounds

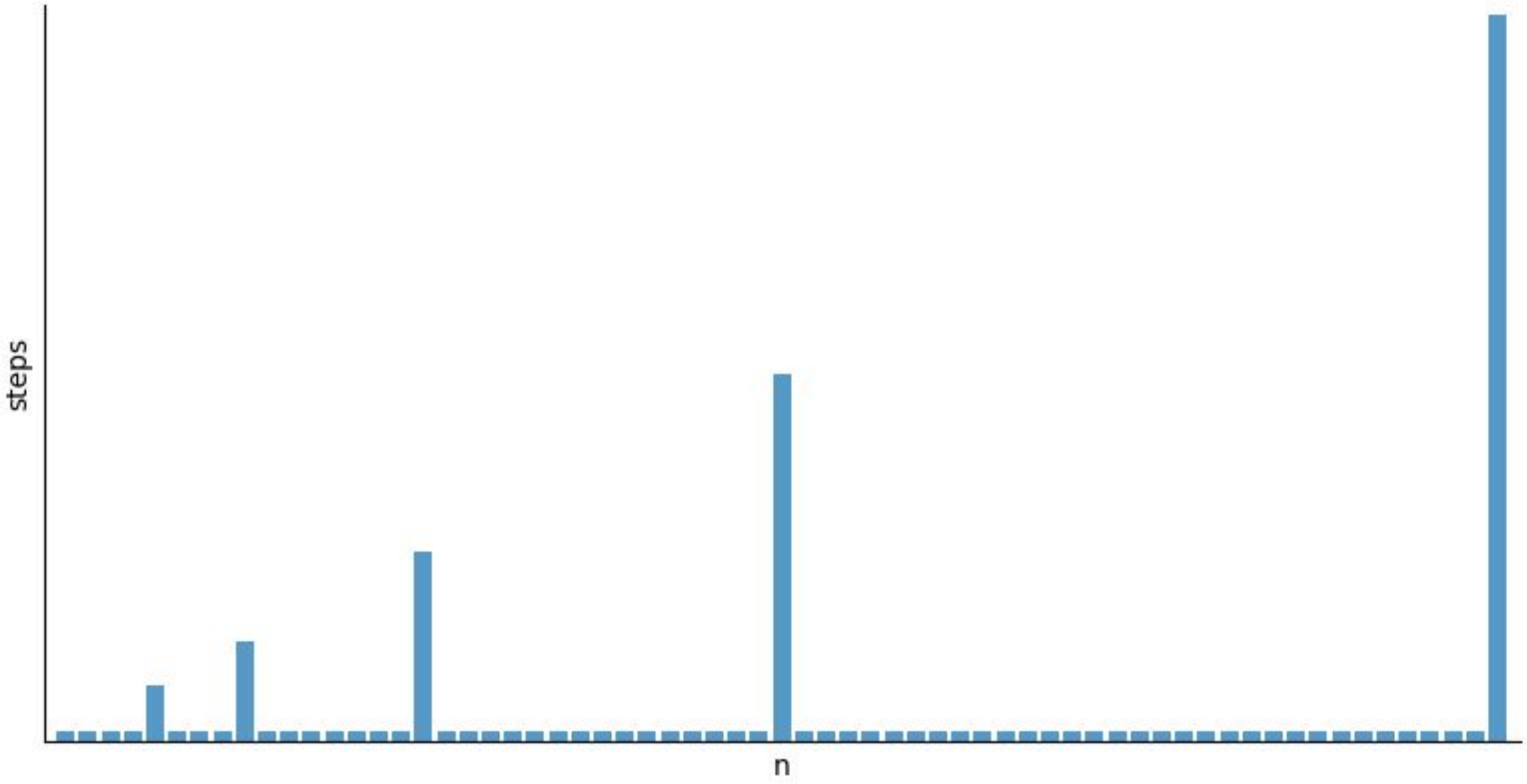
Problem: If we increase the size by a constant amount, doing n adds still costs a total of n^2

How else could we increase the size?

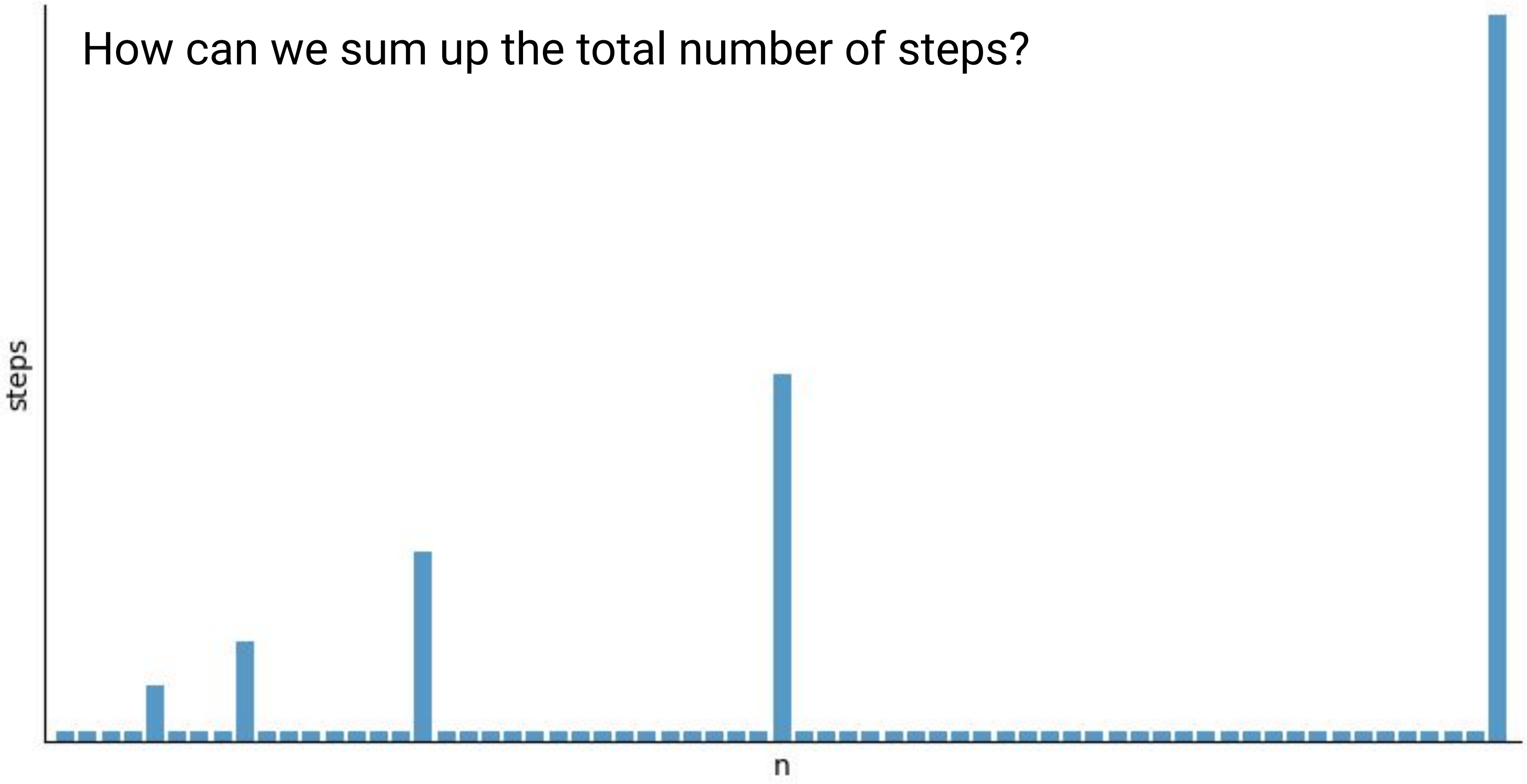
A New Type of Bounds

Problem: If we increase the size by a constant amount, doing ***n*** adds still costs a total of n^2

How else could we increase the size? **Double It!**

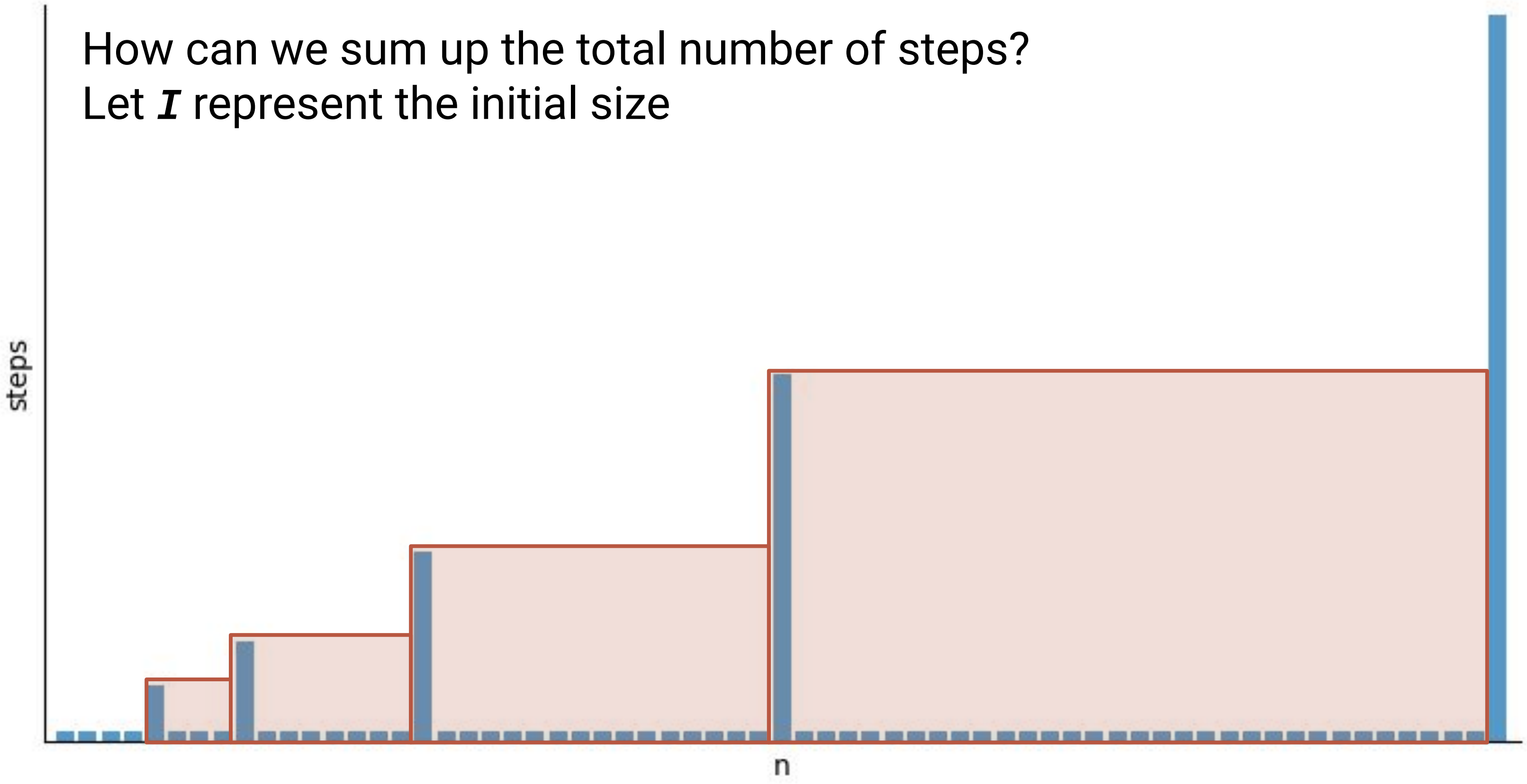


Initial size = 4, newLength = data.length × 2



Initial size = 4, newLength = data.length × 2

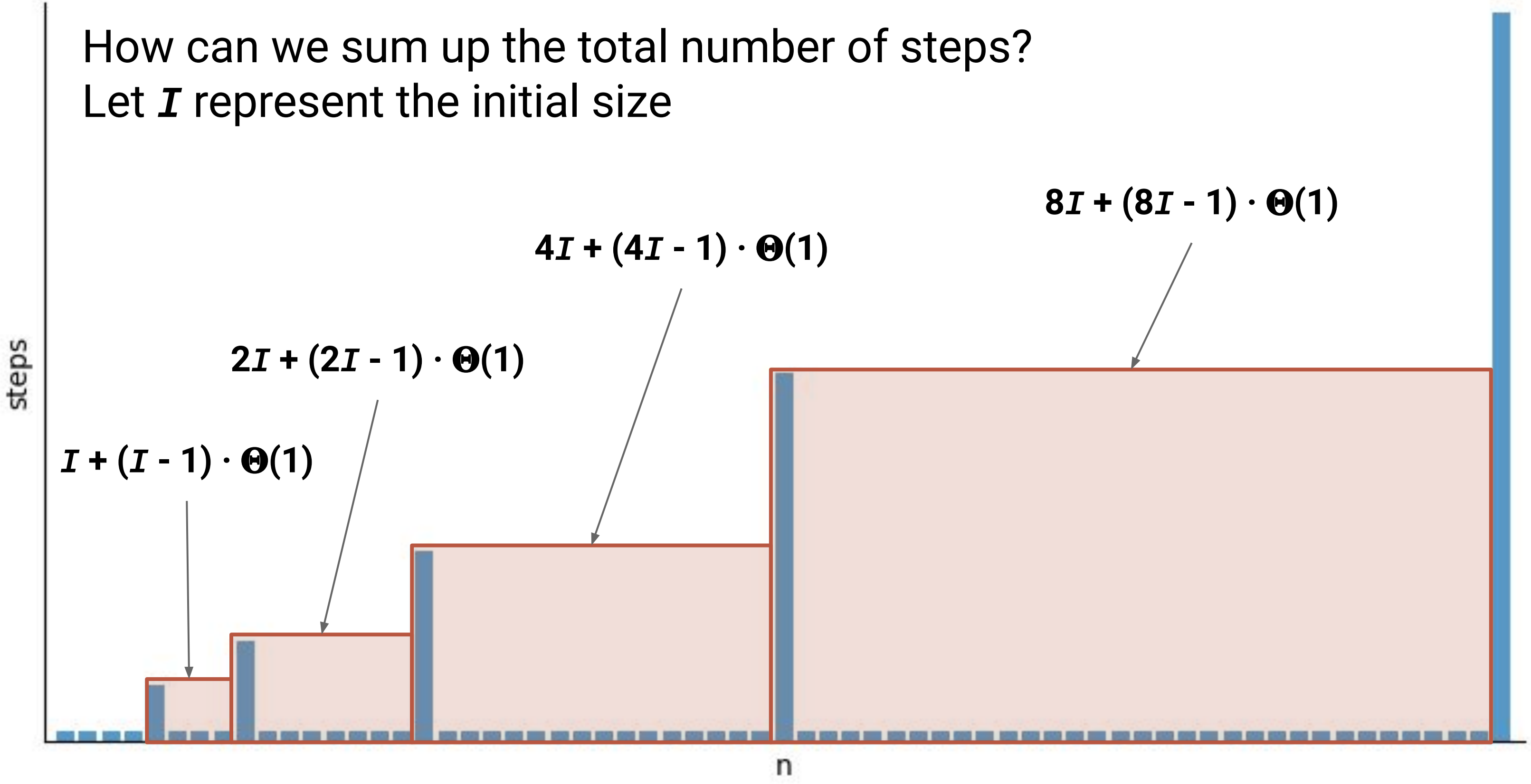
How can we sum up the total number of steps?
Let I represent the initial size



Initial size = 4, newLength = data.length × 2

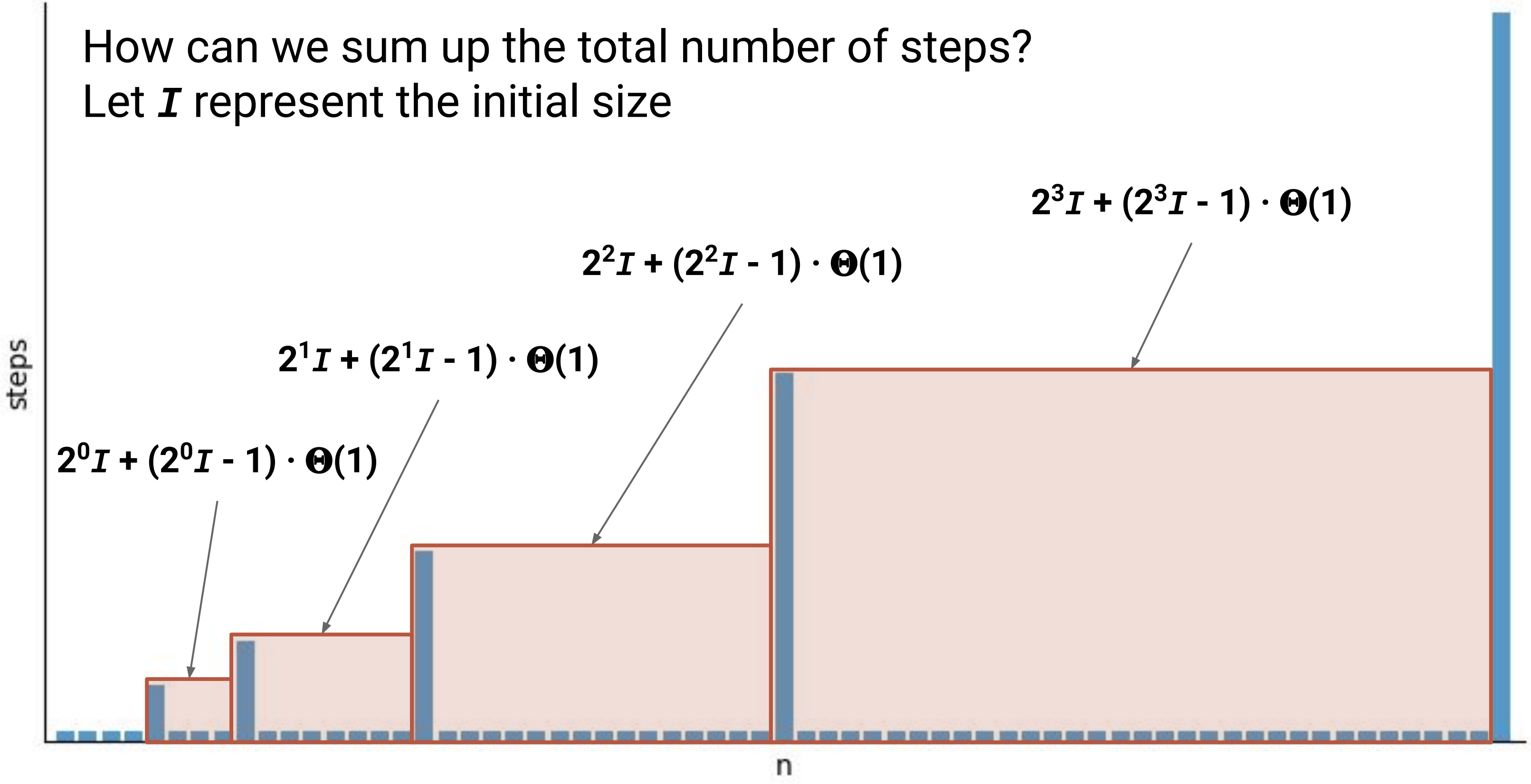
How can we sum up the total number of steps?

Let I represent the initial size



Initial size = 4, newLength = data.length × 2

How can we sum up the total number of steps?
Let I represent the initial size



Initial size = 4, newLength = data.length × 2

Doubling the Array Size

So the cost of the ***i*th** red box is: $2^i I + (2^i I - 1) \cdot \Theta(1) \in \Theta(2^i)$

Doubling the Array Size

So the cost of the i^{th} red box is: $2^i I + (2^i I - 1) \cdot \Theta(1) \in \Theta(2^i)$

How many red boxes are there for ***n*** inserts?

Doubling the Array Size

So the cost of the ***ith*** red box is: $2^i I + (2^i I - 1) \cdot \Theta(1) \in \Theta(2^i)$

How many red boxes are there for ***n*** inserts? $\Theta(\log(n))$

Doubling the Array Size

So the cost of the ***i**^th^* red box is: $2^i I + (2^i I - 1) \cdot \Theta(1) \in \Theta(2^i)$

How many red boxes are there for ***n*** inserts? $\Theta(\log(n))$

How many steps in total?

Doubling the Array Size

So the cost of the ***i*th** red box is: $2^i I + (2^i I - 1) \cdot \Theta(1) \in \Theta(2^i)$

How many red boxes are there for ***n*** inserts? $\Theta(\log(n))$

How many steps in total? $\sum_{i=0}^{\log(n)} 2^i$

Doubling the Array Size

$$\sum_{i=0}^{\log(n)} 2^i = 2^{\log(n)+1} - 1$$

Doubling the Array Size

$$\sum_{i=0}^{\log(n)} 2^i = 2^{\log(n)+1} - 1$$

$$= 2n - 1$$

Doubling the Array Size

$$\sum_{i=0}^{\log(n)} 2^i = 2^{\log(n)+1} - 1$$

$$= 2n - 1$$

$$= O(n)$$

Another Perspective

Let's assume we have a FULL array with 16 elements…What do the next 16 adds look like?

Another Perspective

Let's assume we have a FULL array with 16 elements…What do the next 16 adds look like?

- Create a new array of size 32
- Copy over the 16 elements
- Assign the new element to the next free spot 16 times

Total: Copy 16 elements, assign 16 elements = 2 * 16 operations

Another Perspective

Let's assume we have a FULL array with 32 elements…What do the next 32 adds look like?

- Create a new array of size 64
- Copy over the 32 elements
- Assign the new element to the next free spot 32 times

Total: Copy 32 elements, assign 32 elements = 2 * 32 operations

Another Perspective

Let's assume we have a FULL array with 64 elements…What do the next 64 adds look like?

- Create a new array of size 128
- Copy over the 64 elements
- Assign the new element to the next free spot 64 times

Total: Copy 64 elements, assign 64 elements = 2 * 64 operations

Another Perspective

Let's assume we have a FULL array with ***n*** elements…What do the next ***n*** adds look like?

- Create a new array of size 2****n***
- Copy over the ***n*** elements
- Assign the new element to the next free spot ***n*** times

Total: Copy ***n*** elements, assign ***n*** elements = 2 * ***n*** operations = Θ(***n***)

Another Perspective

Let's assume we have a FULL array with ***n*** elements…What do the next ***n*** adds look like

- Create a r
- Copy over
- Assign the new element to the next free spot ***n*** times

Each chunk starts with a big copy, but that big copy also "pays" for the rest of the insertions in that chunk!

Total: Copy ***n*** elements, assign ***n*** elements = 2 * ***n*** operations = $\Theta(n)$

Doubling the Array Size

Wait…so one call to add is ***O*(*n*)**...

But ***n*** calls to add are ***O*(*n*)** as well?

MOST calls only require constant time…

The total cost of ***n*** calls is **<u>guaranteed</u>** ***O*(*n*)** steps

Amortized Runtime

If n calls to a function take $O(f(n))$...

We say the **Amortized Runtime** is $O(f(n) / n)$

The **amortized runtime** of `add` on an `ArrayList` is: $O(n/n) = O(1)$
The **unqualified runtime** of `add` on an `ArrayList` is: $O(n)$

List Runtimes (so far...)

	ArrayList	Linked List (by index)	Linked List (by reference)
`get(...)`	$\Theta(1)$	$\Theta(\text{idx})$ or $O(n)$	$\Theta(1)$
`set(...)`	$\Theta(1)$	$\Theta(\text{idx})$ or $O(n)$	$\Theta(1)$
`size()`	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
`add(...)`	TBD	$\Theta(\text{idx})$ or $O(n)$	$\Theta(1)$
`remove(...)`	TBD	$\Theta(\text{idx})$ or $O(n)$	$\Theta(1)$

List Runtimes (so far...)

	ArrayList	Linked List (by index)	Linked List (by reference)
`get(...)`	$\Theta(1)$	$\Theta(\text{idx})$ or $O(n)$	$\Theta(1)$
`set(...)`	$\Theta(1)$	$\Theta(\text{idx})$ or $O(n)$	$\Theta(1)$
`size()`	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
`add(...)`	$O(n)$, Amortized $\Theta(1)$	$\Theta(\text{idx})$ or $O(n)$	$\Theta(1)$
`remove(...)`	$O(n)$	$\Theta(\text{idx})$ or $O(n)$	$\Theta(1)$

Follow-Up Questions

What is the amortized runtime of `add` for a `LinkedList`?

What is the runtime of `add(int idx, E elem)` for an `ArrayList`?

Follow-Up Questions

What is the amortized runtime of **`add`** for a **`LinkedList`**?

Each **`add`** is $O(1)$. Total for n calls is $O(n)$. Amortized is $O(n/n) = O(1)$

What is the runtime of **`add(int idx, E elem)`** for an **`ArrayList`**?

To **`add`** between two elements requires the rest of the elements to be shifted to the right (opposite of **`remove`**), so runtime is always $O(n)$.

What Data Structure is Best?

Scenario #1: You need to read in the lines of a CSV file, store them in a List, and later be able to access individual records based on index.

What Data Structure is Best?

Scenario #1: You need to read in the lines of a CSV file, store them in a List, and later be able to access individual records based on index.

`ArrayList`

Since the amortized runtime of add for **`ArrayList`** and **`LinkedList`**, adding the $\boldsymbol{n}$ lines of the CSV file will take $\boldsymbol{O(n)}$ time for both…

But **`ArrayLists`** will then have an advantage because looking up records by index will be $\boldsymbol{O(1)}$ whereas **`LinkedLists`** will be $\boldsymbol{O(n)}$

What Data Structure is Best?

Scenario #2: Users logging onto an online game need to be efficiently added to a List in the order they log on. From time to time you must be able to iterate through the list from beginning to end.

What Data Structure is Best?

Scenario #2: Users logging onto an online game need to be efficiently added to a List in the order they log on. From time to time you must be able to iterate through the list from beginning to end.

`LinkedList`

The enumeration will cost a total of $O(n)$ for both types of List

But some users will experience longer waits being added to the List if implemented as an **`ArrayList`** due to the need for it to occasionally resize