CSE 250
Data Structures

Dr. Eric Mikida
epmikida@buffalo.edu
Capen 208

Lecture 11
Amortized Runtime

Announcements

- PA1 Implementation AutoLab is now open!
  - Don’t rely on AutoLab to tell you what’s wrong – that’s what **your** tests are for
  - Add more tests as needed
- WA2 releases next week

`List` Summary (so far...)

	`Array`	`LinkedList` *(by index)*	`LinkedList` *(by reference)*
`get(i)`	$\Theta(1)$	$\Theta(i) \subset O(n)$	$\Theta(1)$
`set(i, v)`	$\Theta(1)$	$\Theta(i) \subset O(n)$	$\Theta(1)$
`size()`	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
`add(v)`	$O(n)$	$\Theta(1)$	$\Theta(1)$
`add(i, v)`	*???*	$\Theta(i) \subset O(n)$	$\Theta(1)$
`remove(idx)`	$O(n)$	$\Theta(i) \subset O(n)$	$\Theta(1)$

`ArrayList.add(elem)` - Analysis

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

What are the bounds of $T_{\text{add}}(n)$?

`ArrayList.add(elem)` - Analysis

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

What are the bounds of $T_{\text{add}}(n)$?

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

`ArrayList.add(elem)` - Analysis

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

What are the bounds of $T_{\text{add}}(n)$?

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

How often do our add calls require n steps?

`ArrayList.add(elem)` - Analysis

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

What are the bounds of $T_{\text{add}}(n)$?

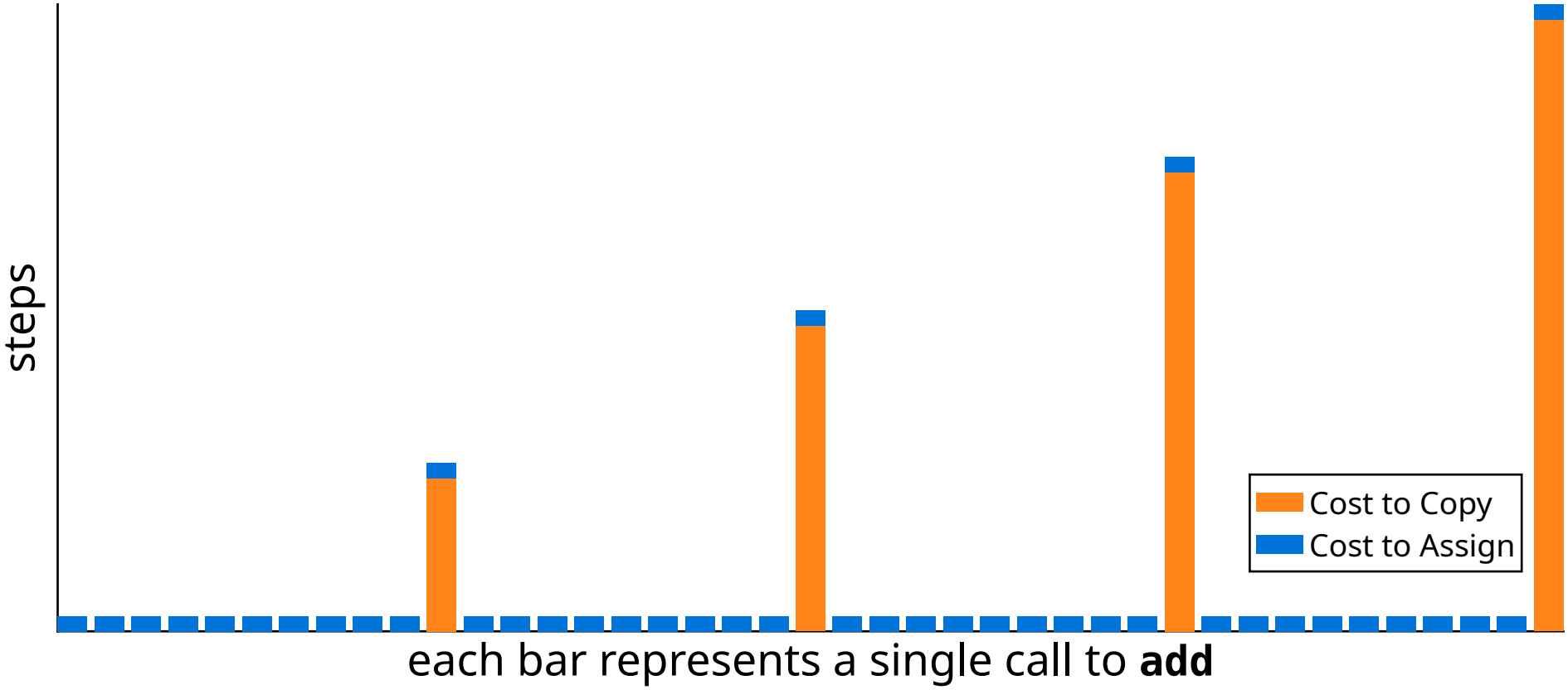
$$T_{\text{add}}(n) \in O(n), \Omega(1)$$

How often do our add calls require n steps?

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

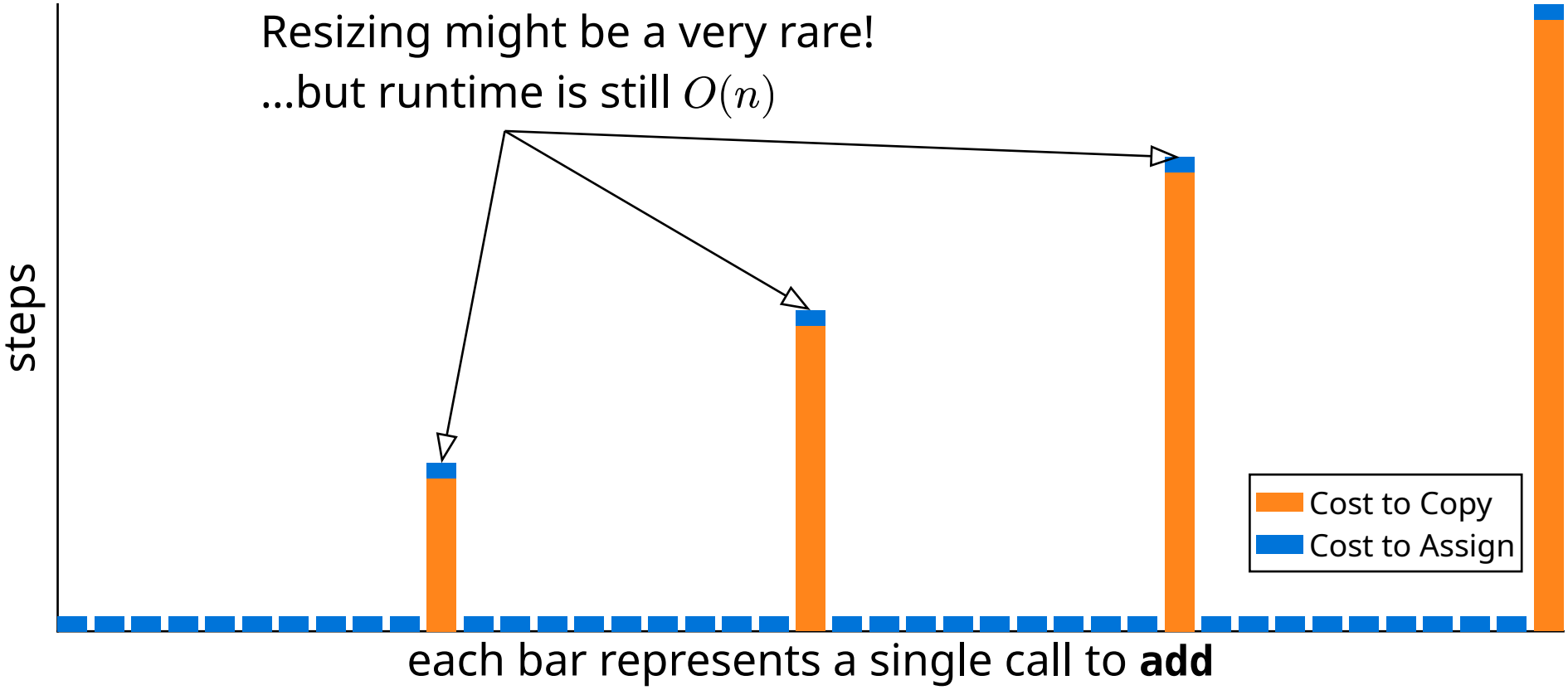
`ArrayList.add(elem)`: +10 Each Resize

Initial Size = 10, **`newLength = data.length + 10`**



`ArrayList.add(elem)`: +10 Each Resize

Initial Size = 10, **`newLength = data.length + 10`**



A Note on Runtime Complexity

So far, when discussing 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 $O(n)$

A Note on Runtime Complexity

So far, when discussing 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 $O(n)$

This type of analysis is referred to as unqualified analysis:

- Analyze a single run of the algorithm without any extra qualifications/context
- We would say the **unqualified runtime** of **`ArrayList.add`** is $O(n)$

A Note on Runtime Complexity

So far, when discussing 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 $O(n)$

This type of analysis is referred to as unqualified analysis:

- Analyze a single run of the algorithm without any extra qualifications/context
- We would say the **unqualified runtime** of **`ArrayList.add`** is $O(n)$

But sometimes extra context is relevant...how can we include it in our analysis?

Common Pattern: Repeated Calls

Oftentimes, we will 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?

Appending n Elements to a `LinkedList`

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

What is the unqualified runtime of this code snippet?

Appending n Elements to a `LinkedList`

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

What is the unqualified runtime of this code snippet?

$$\underbrace{\Theta(1) + \Theta(1) + ... + \Theta(1)}_{n \text{ iterations}} = n \cdot \Theta(1) = \Theta(n)$$

Appending n Elements to an `ArrayList`

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

What is the unqualified runtime of this code snippet?

Appending n Elements to an `ArrayList`

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

What is the unqualified runtime of this code snippet?

$$\underbrace{O(n) + O(n) + \ldots + O(n)}_{n \text{ iterations}} = n \cdot O(n) = O(n^2)$$

Appending n Elements to an `ArrayList`

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

What is the unqualified runtime of this code snippet?

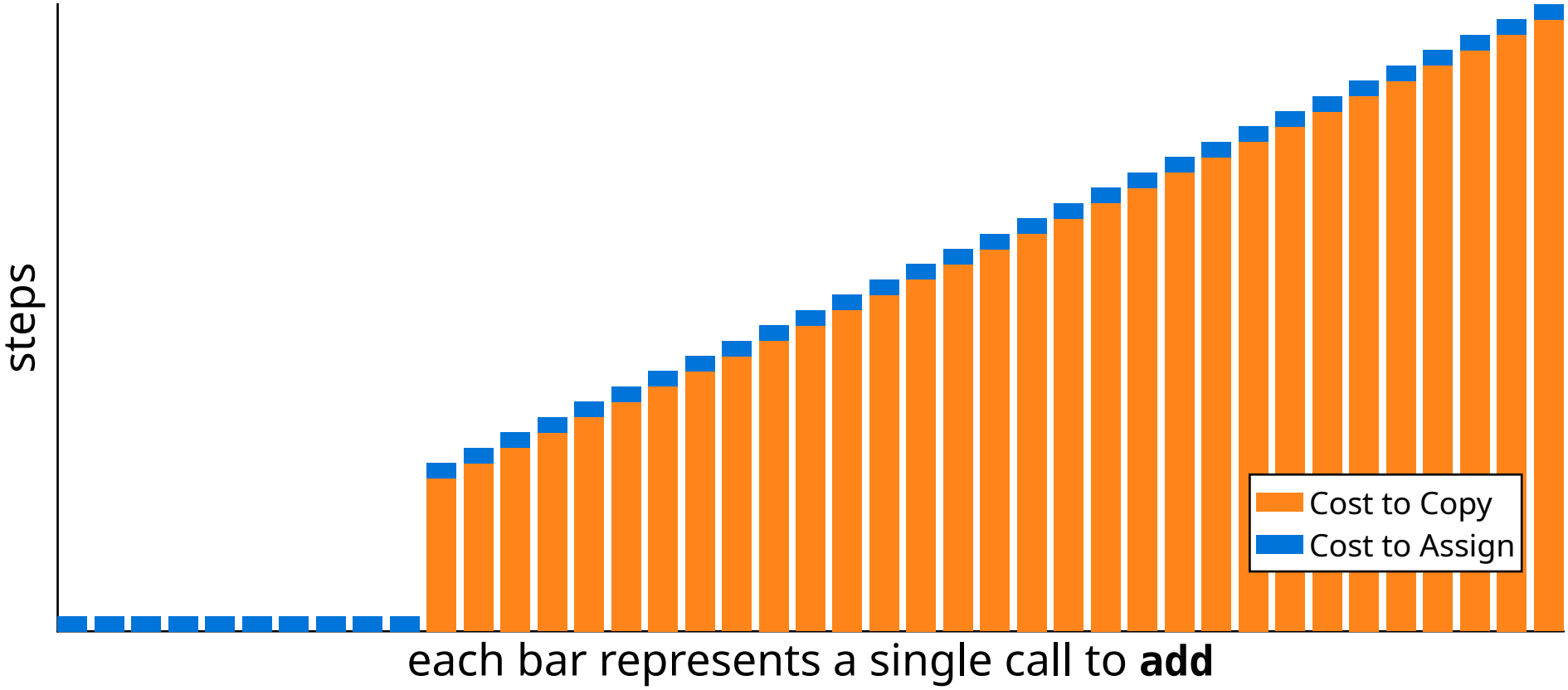
$$\underbrace{O(n) + O(n) + \ldots + O(n)}_{n \text{ iterations}} = n \cdot O(n) = O(n^2)$$

But is this a tight bound? Not every iteration takes n steps...

We need to take a more detailed look to be sure.

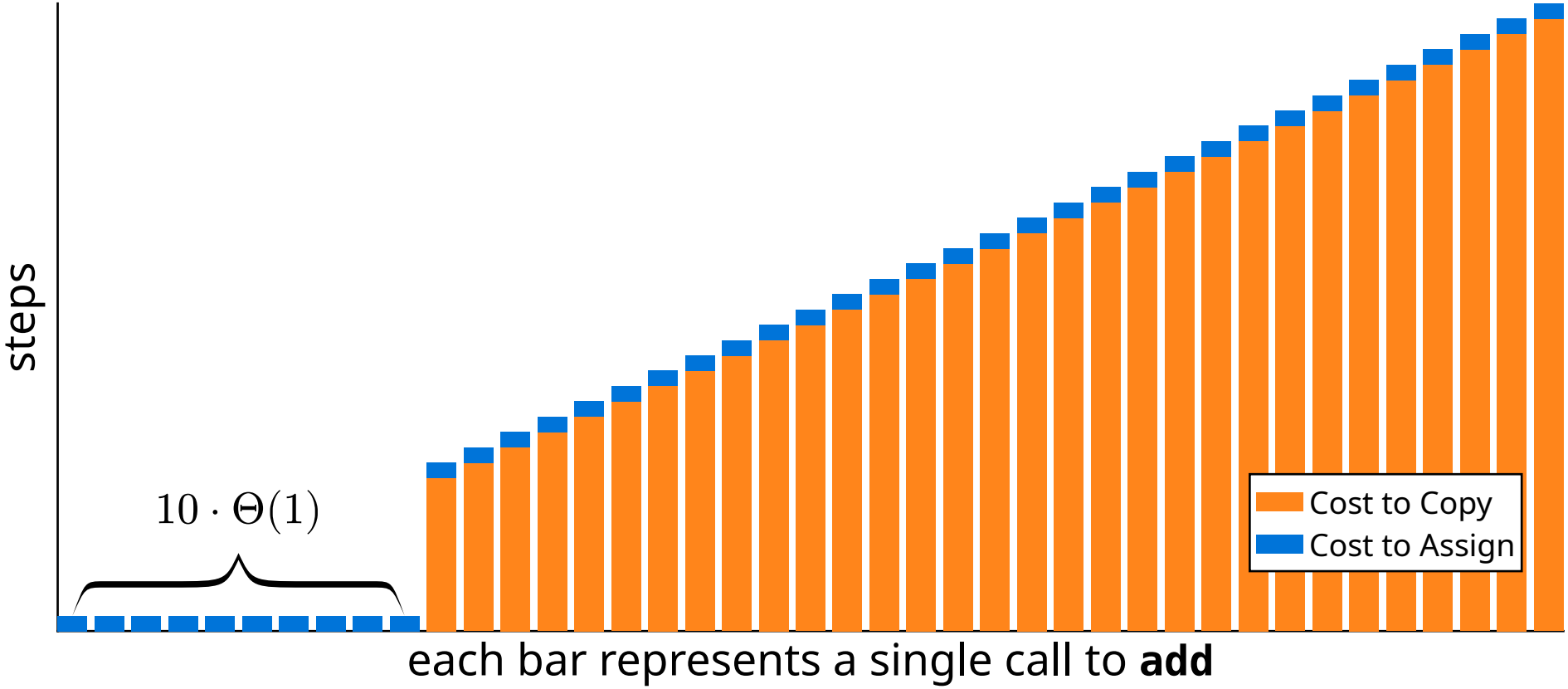
`ArrayList.add(elem)`: +1 Each Resize

Initial Size = 10, **`newLength = data.length + 1`**



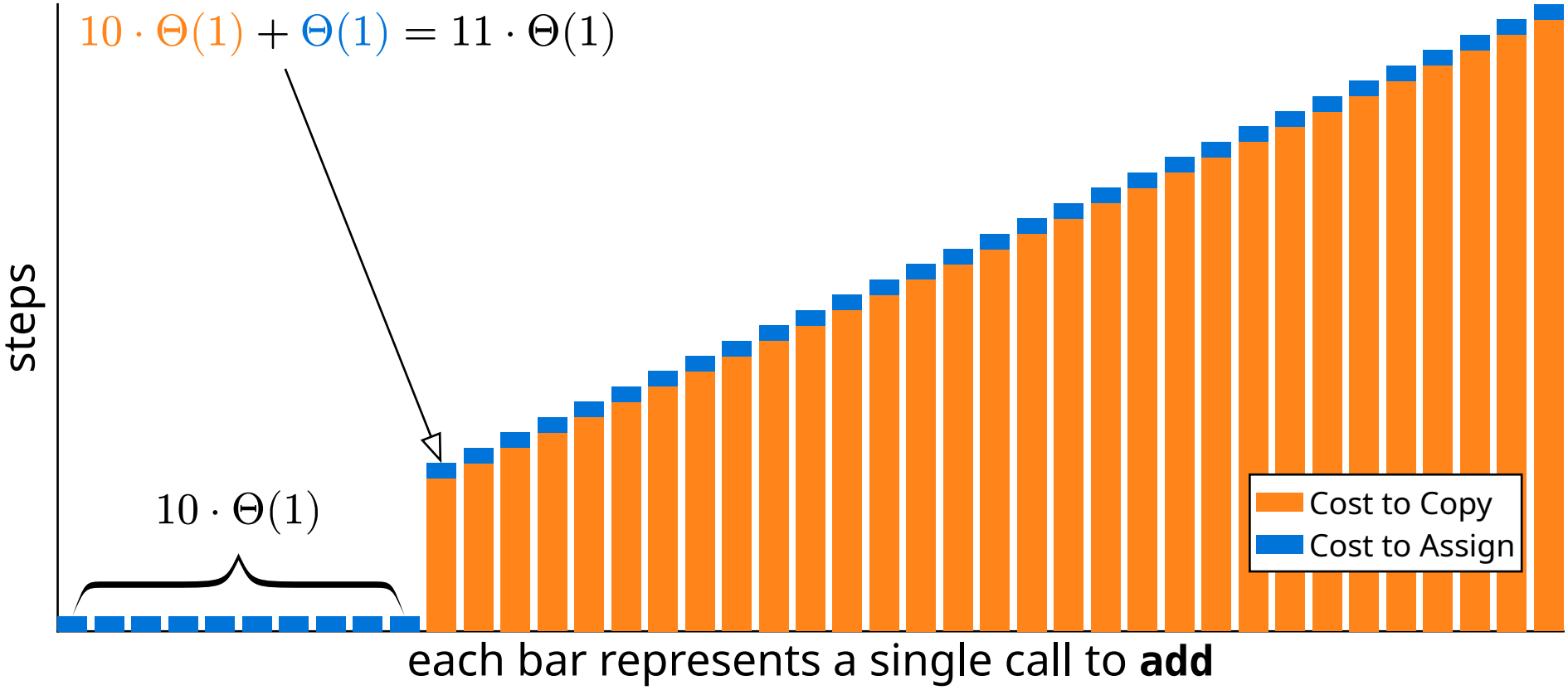
`ArrayList.add(elem)`: +1 Each Resize

Initial Size = 10, **`newLength = data.length + 1`**



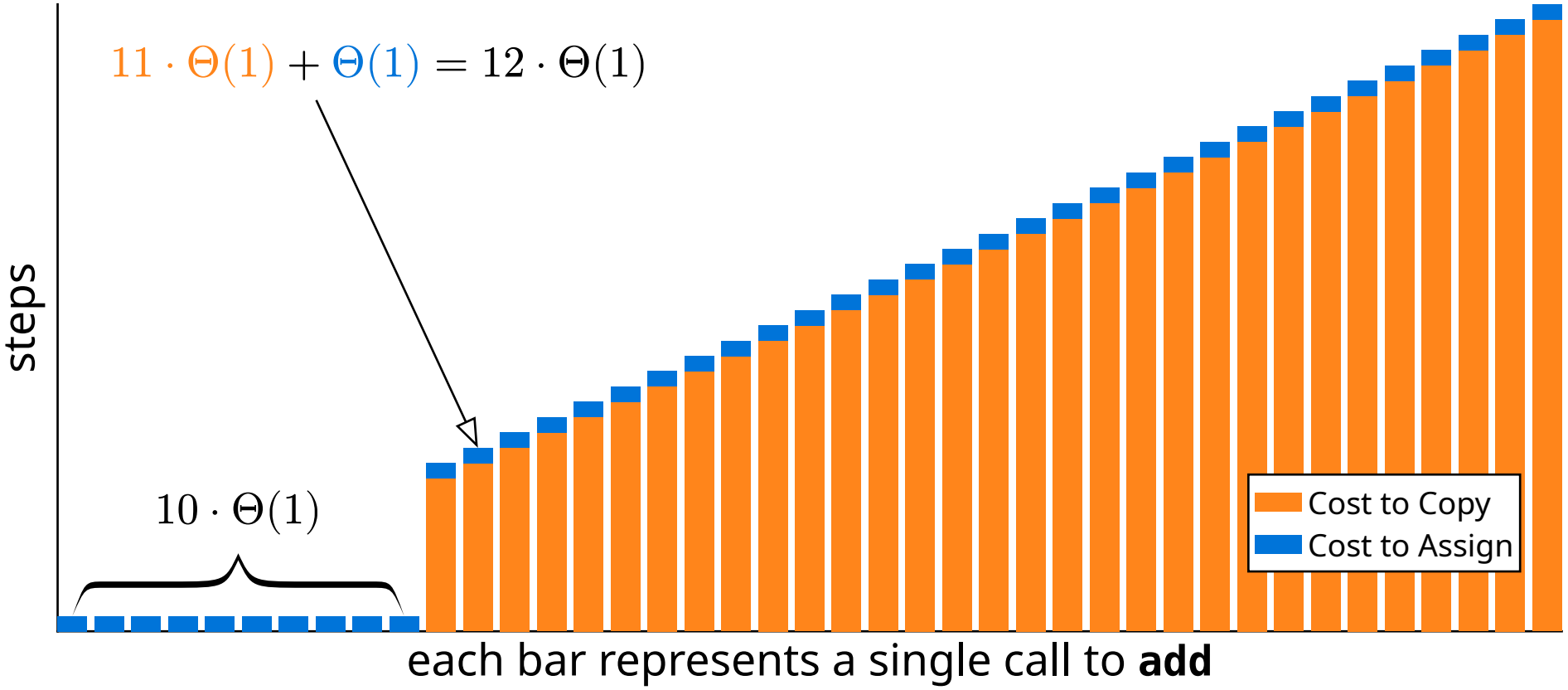
`ArrayList.add(elem)`: +1 Each Resize

Initial Size = 10, **`newLength = data.length + 1`**



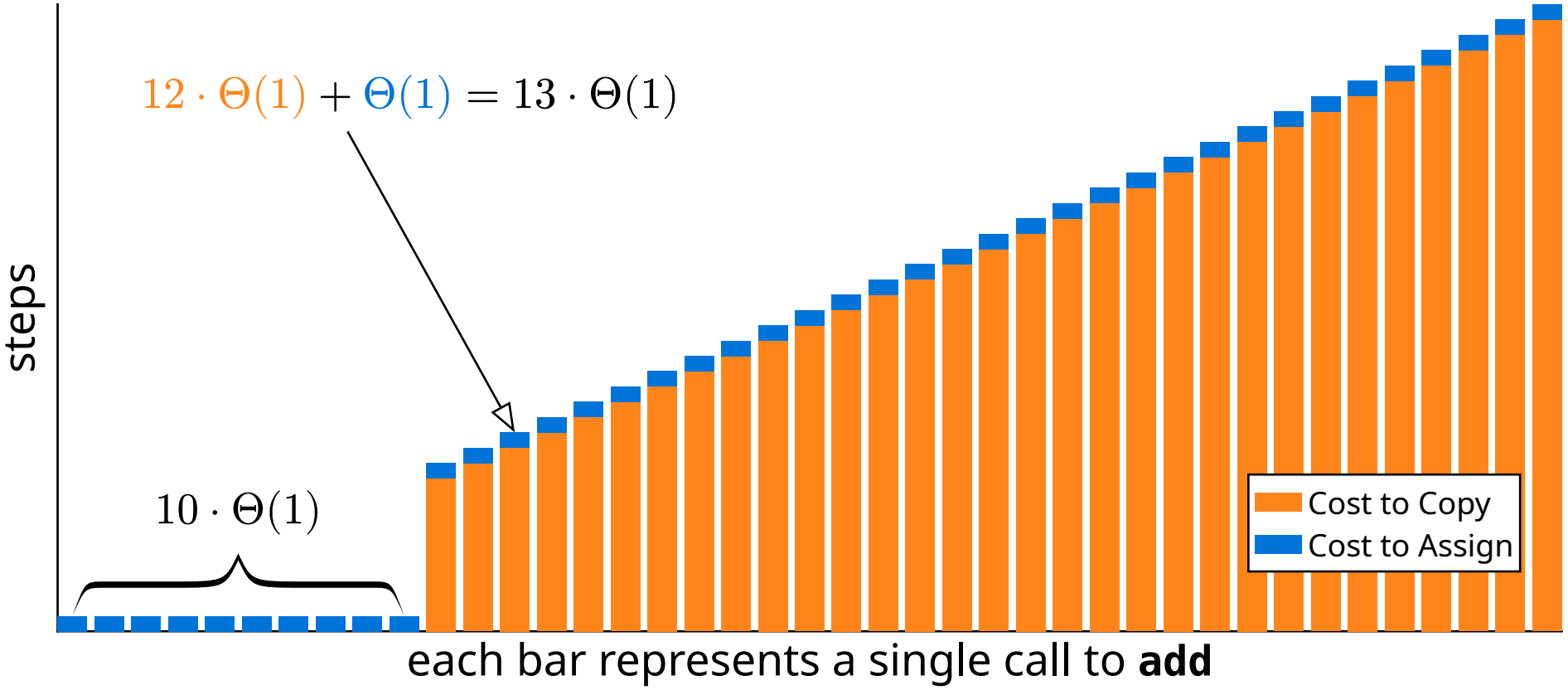
`ArrayList.add(elem)`: +1 Each Resize

Initial Size = 10, **`newLength = data.length + 1`**



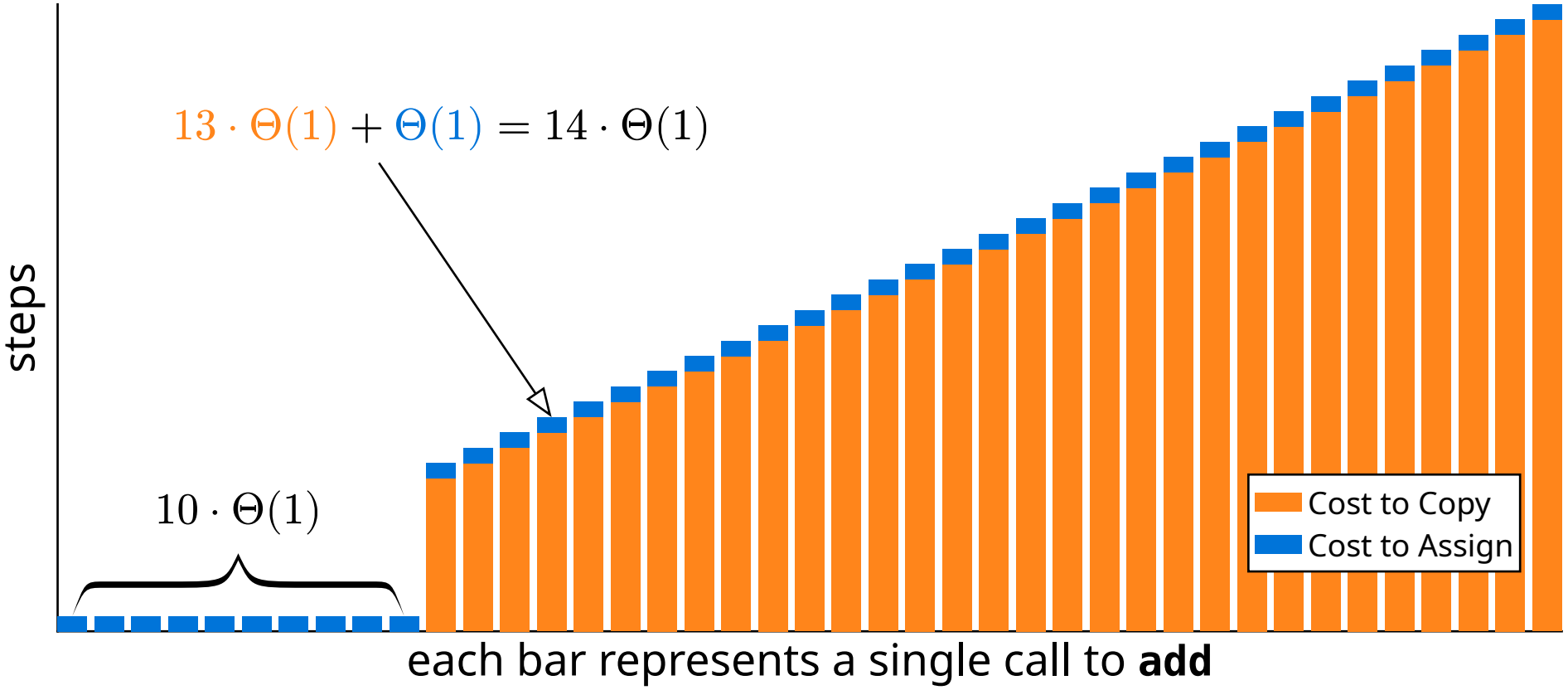
`ArrayList.add(elem)`: +1 Each Resize

Initial Size = 10, **`newLength = data.length + 1`**



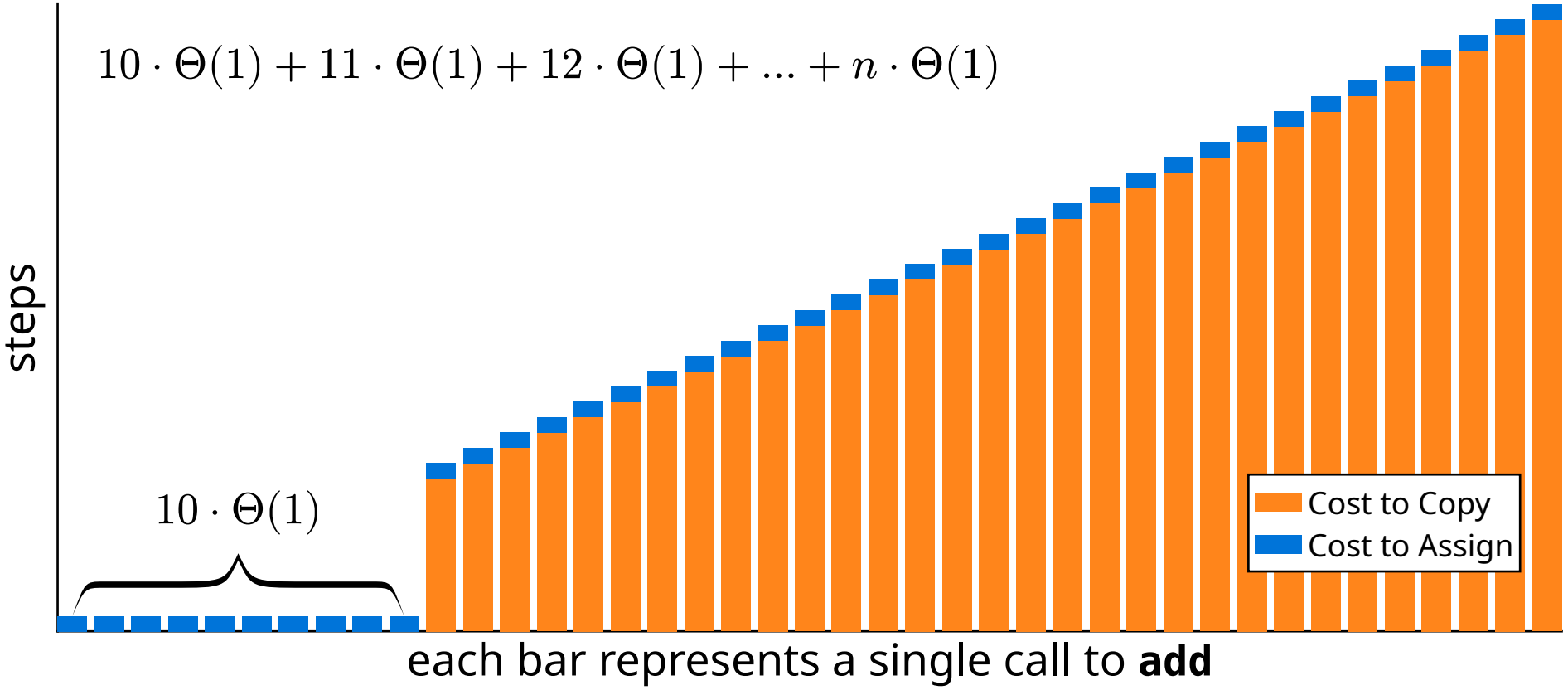
`ArrayList.add(elem)`: +1 Each Resize

Initial Size = 10, **`newLength = data.length + 1`**



`ArrayList.add(elem)`: +1 Each Resize

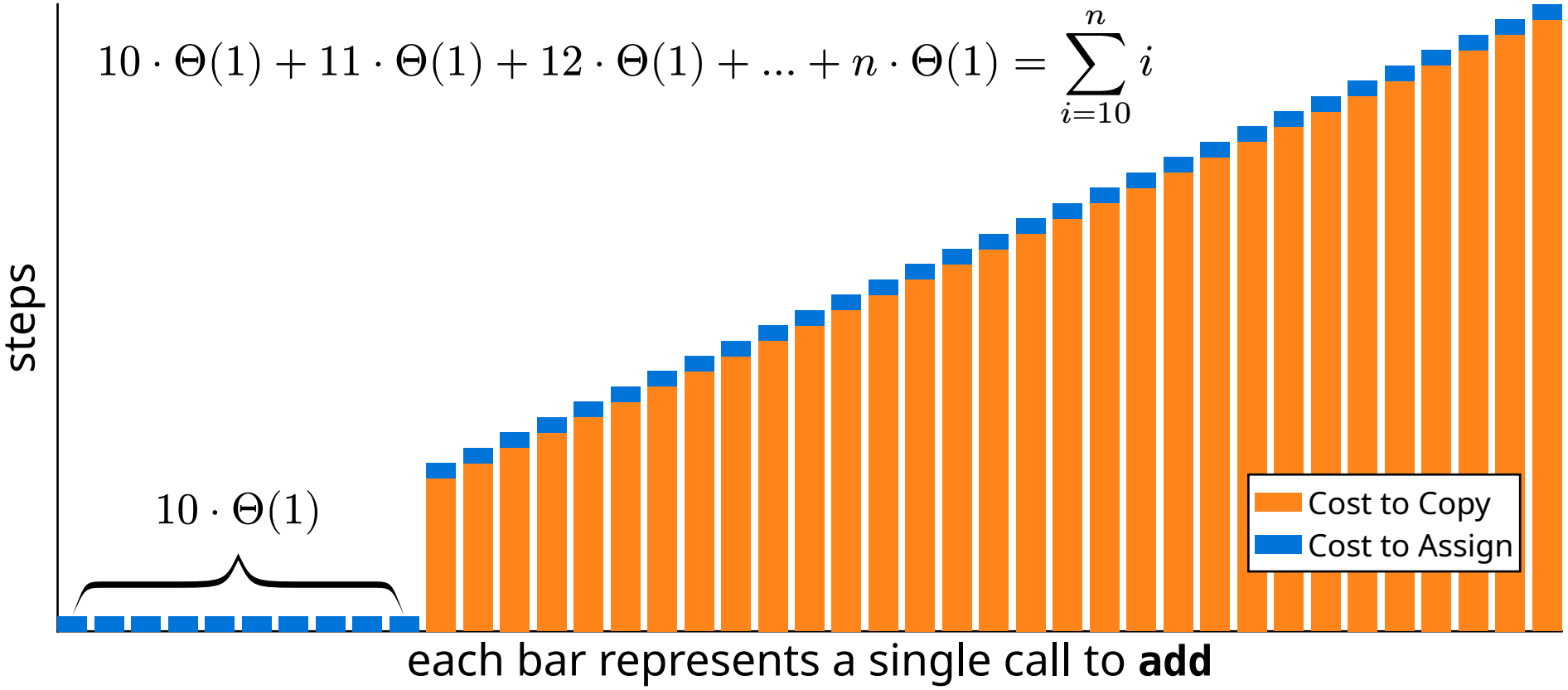
Initial Size = 10, **`newLength = data.length + 1`**



`ArrayList.add(elem)`: +1 Each Resize

Initial Size = 10, **`newLength = data.length + 1`**

$$10 \cdot \Theta(1) + 11 \cdot \Theta(1) + 12 \cdot \Theta(1) + ... + n \cdot \Theta(1) = \sum_{i=10}^{n} i$$

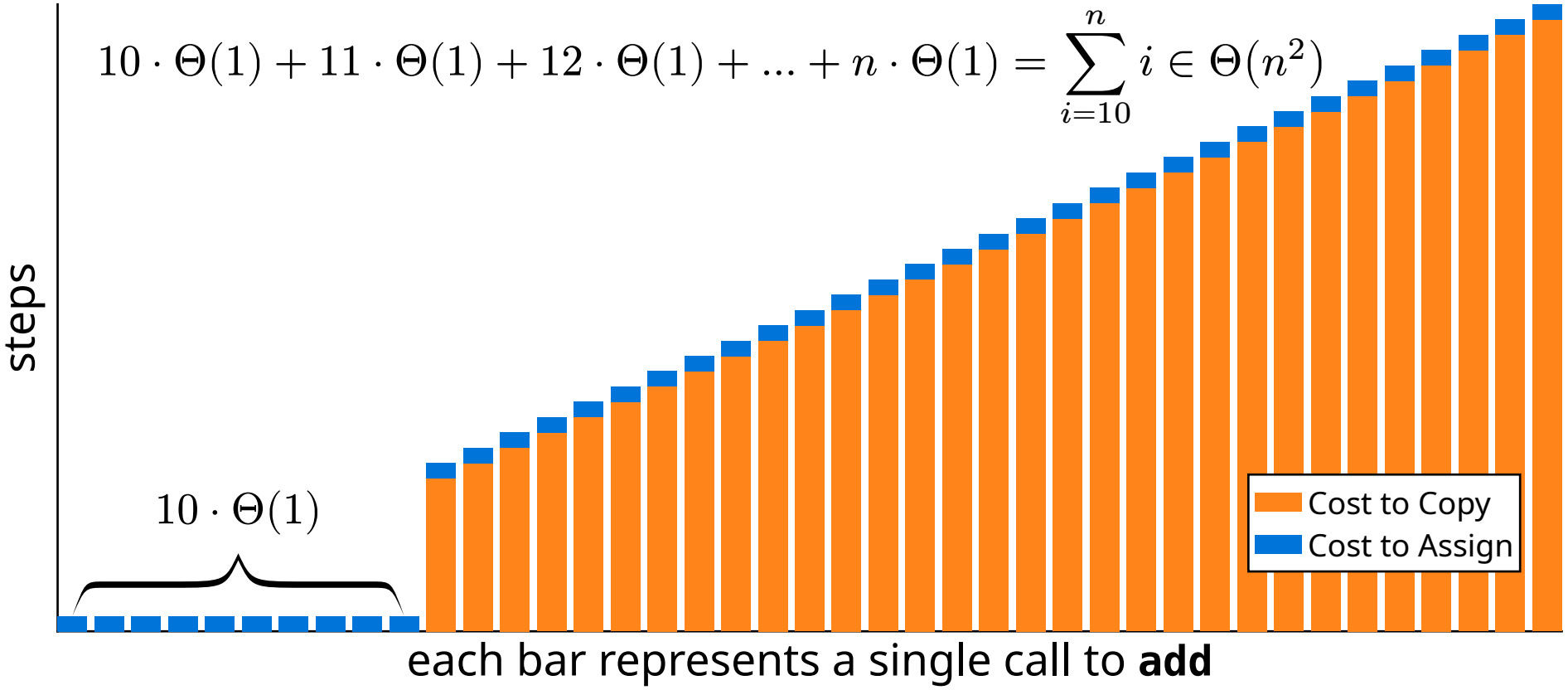


each bar represents a single call to **`add`**

`ArrayList.add(elem)`: +1 Each Resize

Initial Size = 10, **`newLength = data.length + 1`**

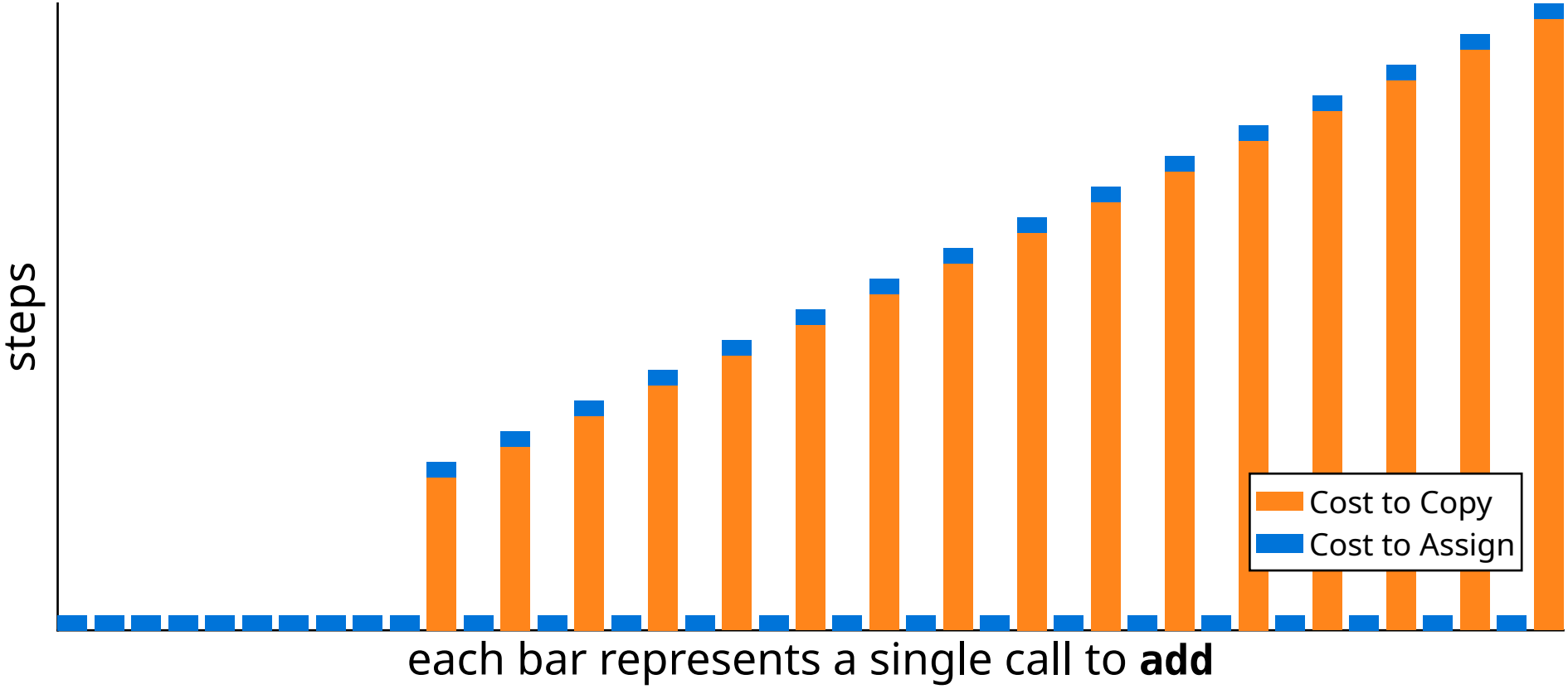
$$10 \cdot \Theta(1) + 11 \cdot \Theta(1) + 12 \cdot \Theta(1) + ... + n \cdot \Theta(1) = \sum_{i=10}^{n} i \in \Theta(n^2)$$



each bar represents a single call to **`add`**

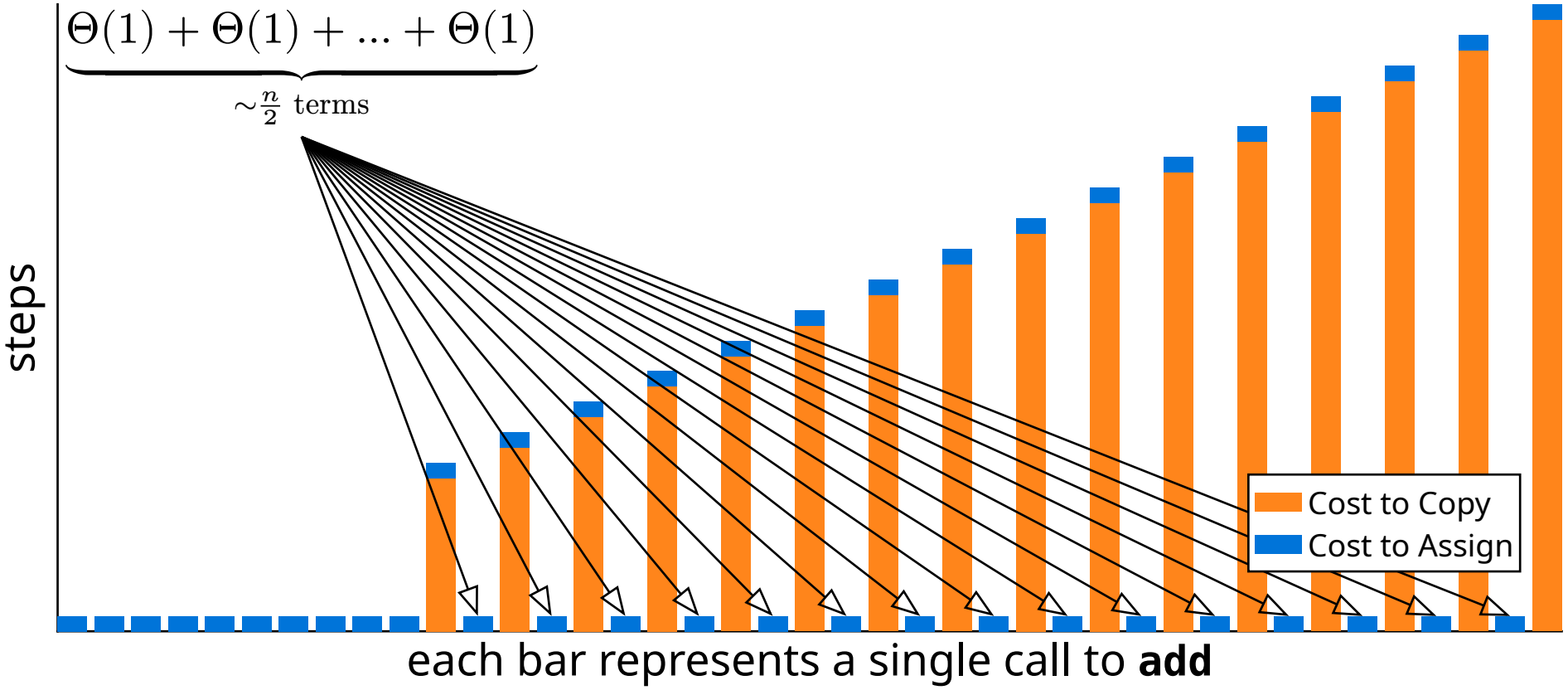
`ArrayList.add(elem)`: +2 Each Resize

Initial Size = 10, **`newLength = data.length + 2`**



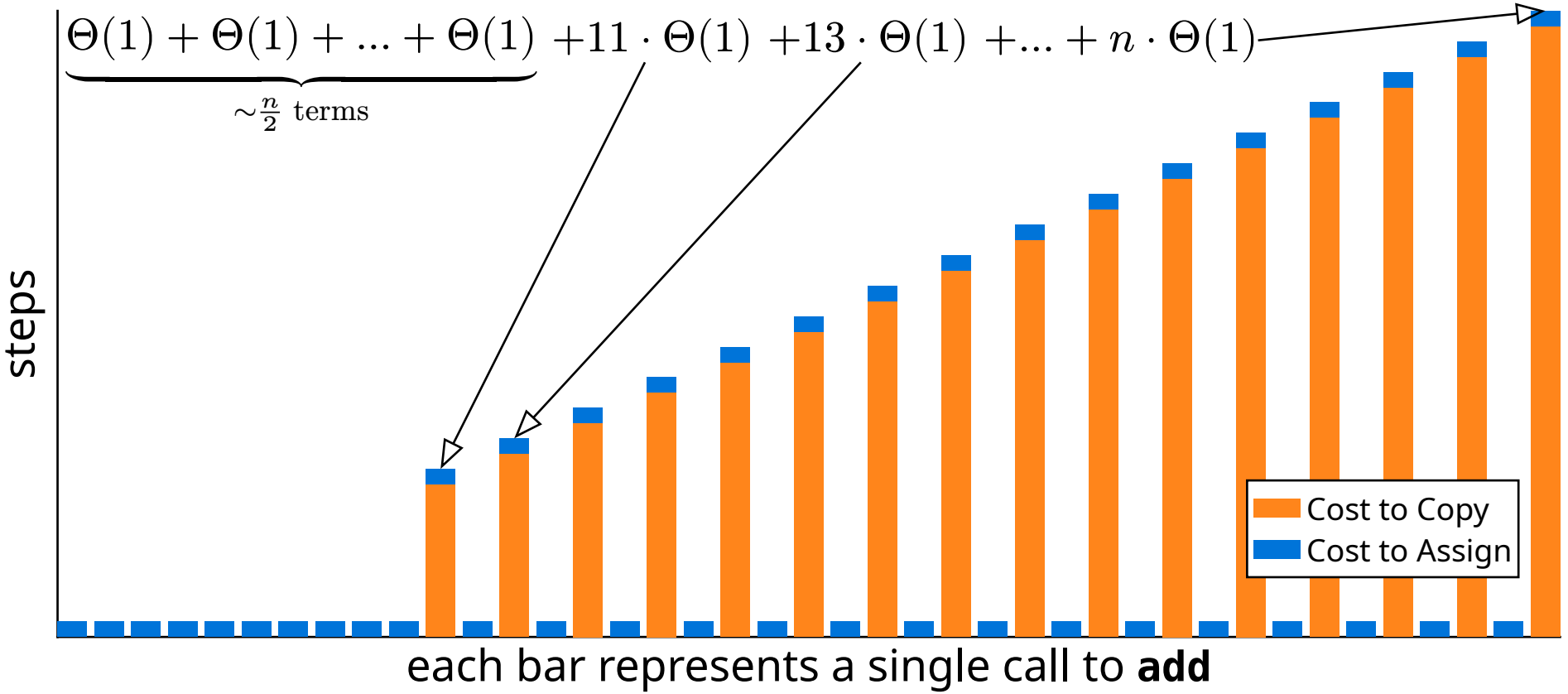
`ArrayList.add(elem)`: +2 Each Resize

Initial Size = 10, **`newLength = data.length + 2`**



`ArrayList.add(elem)`: +2 Each Resize

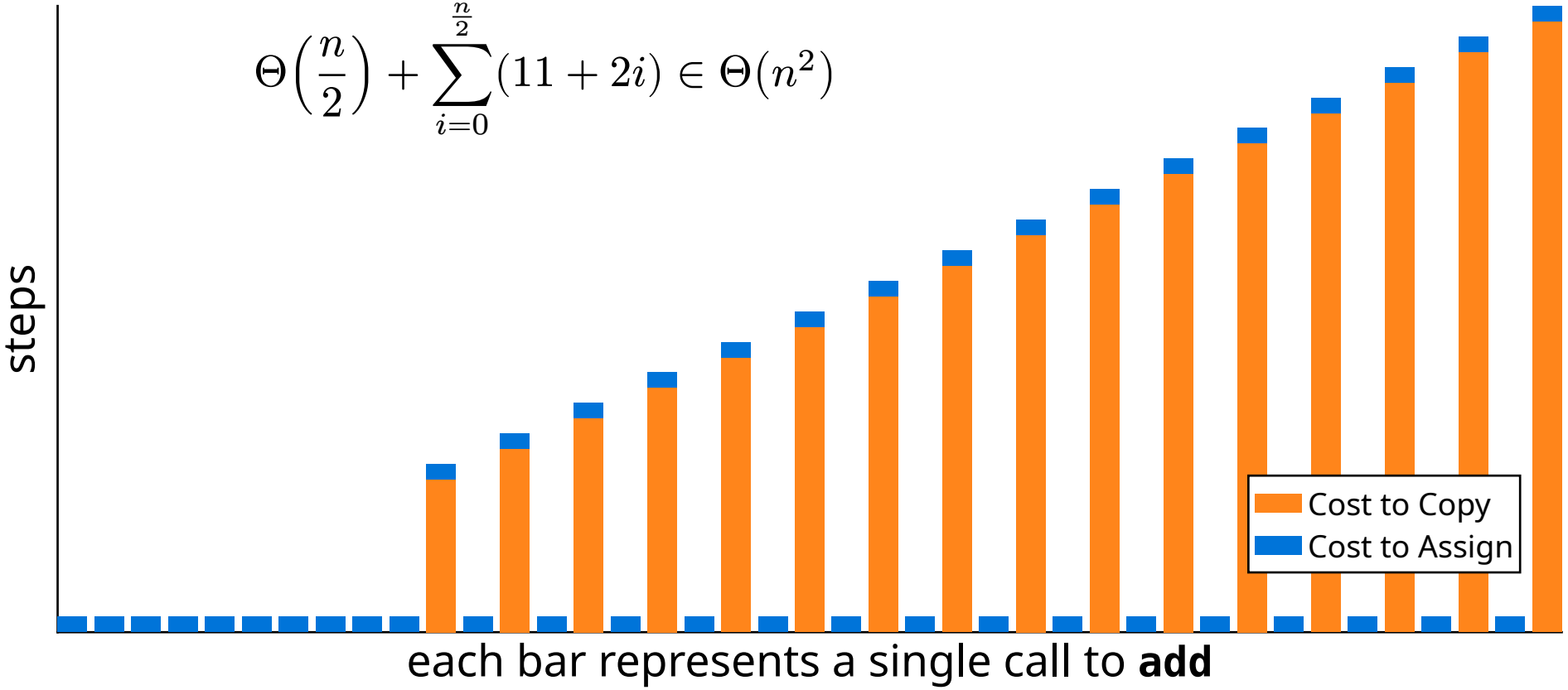
Initial Size = 10, **`newLength = data.length + 2`**



`ArrayList.add(elem)`: +2 Each Resize

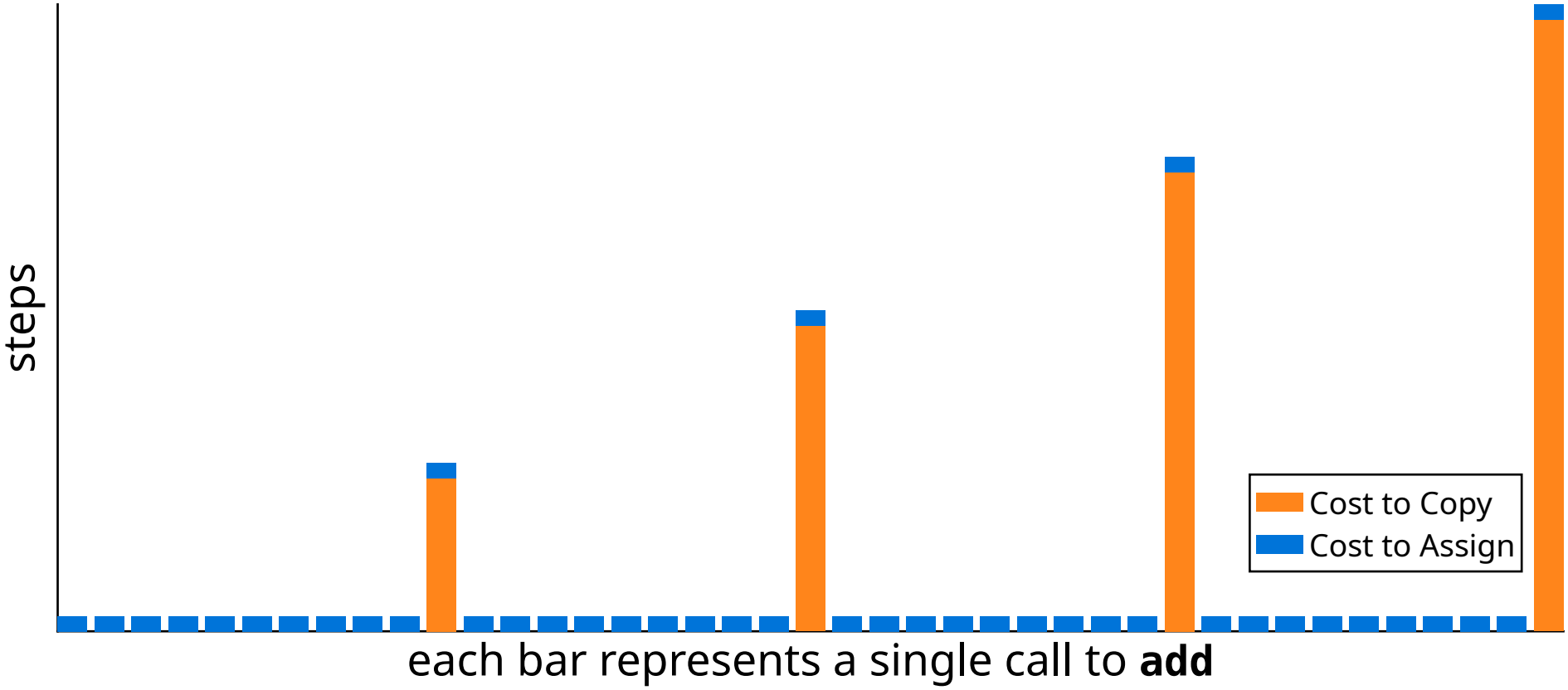
Initial Size = 10, **`newLength = data.length + 2`**

$$\Theta\left(\frac{n}{2}\right) + \sum_{i=0}^{\frac{n}{2}} (11 + 2i) \in \Theta(n^2)$$



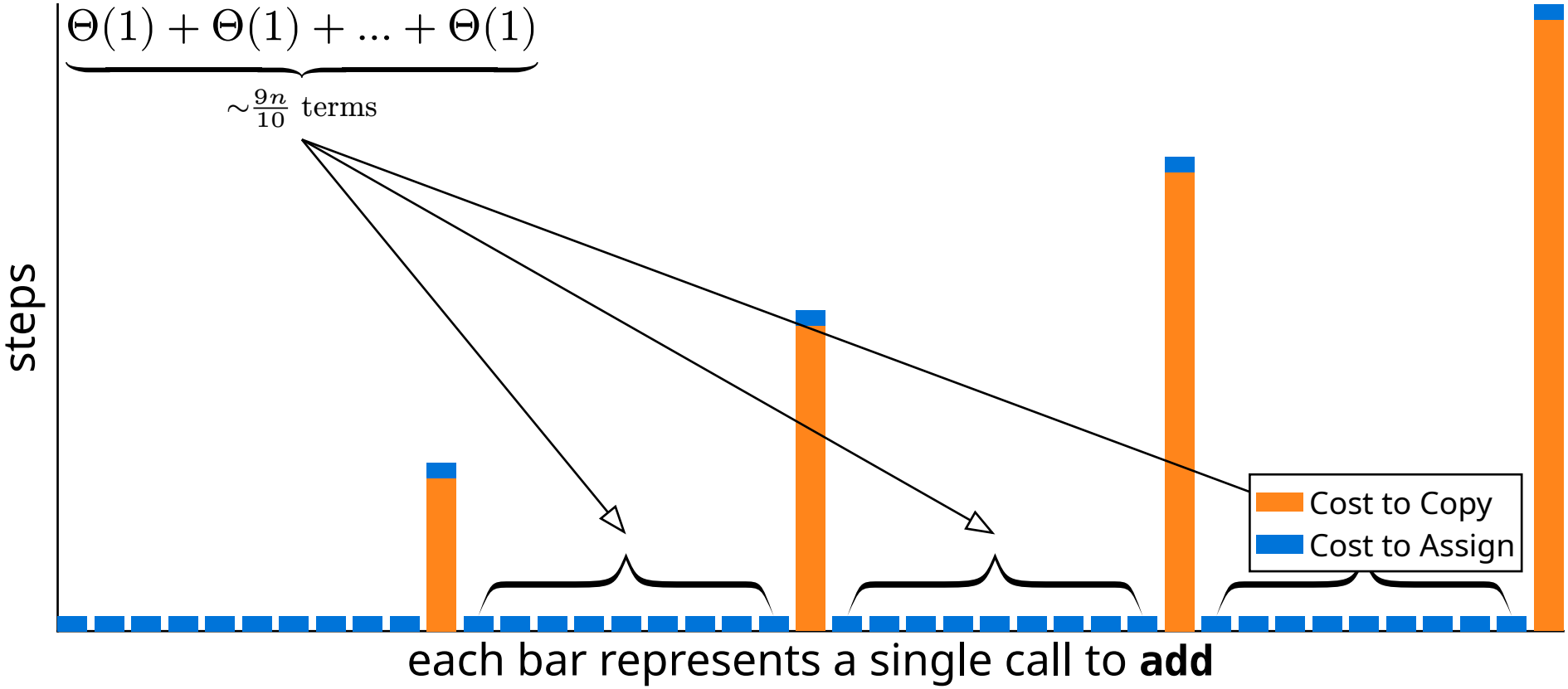
`ArrayList.add(elem)`: +10 Each Resize

Initial Size = 10, **`newLength = data.length + 10`**



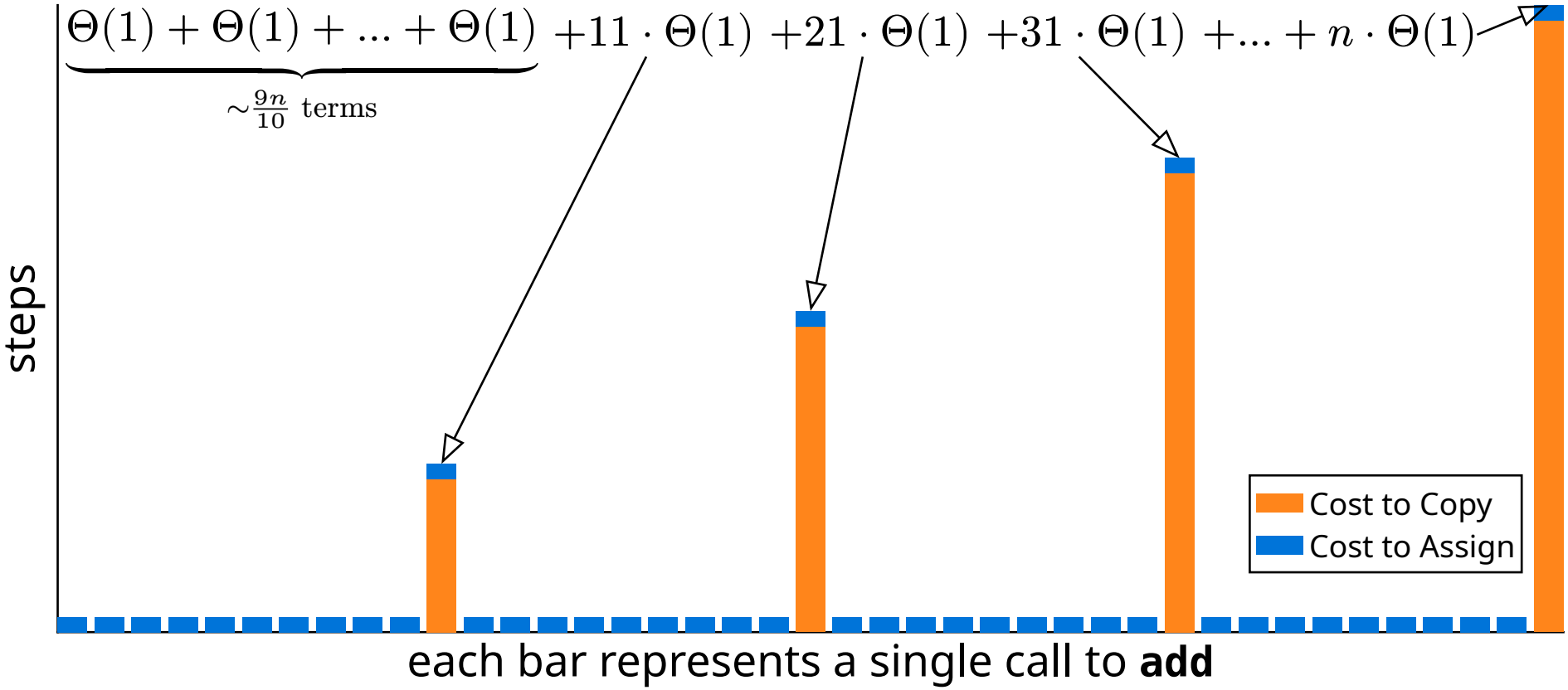
`ArrayList.add(elem)`: +10 Each Resize

Initial Size = 10, **`newLength = data.length + 10`**



`ArrayList.add(elem)`: +10 Each Resize

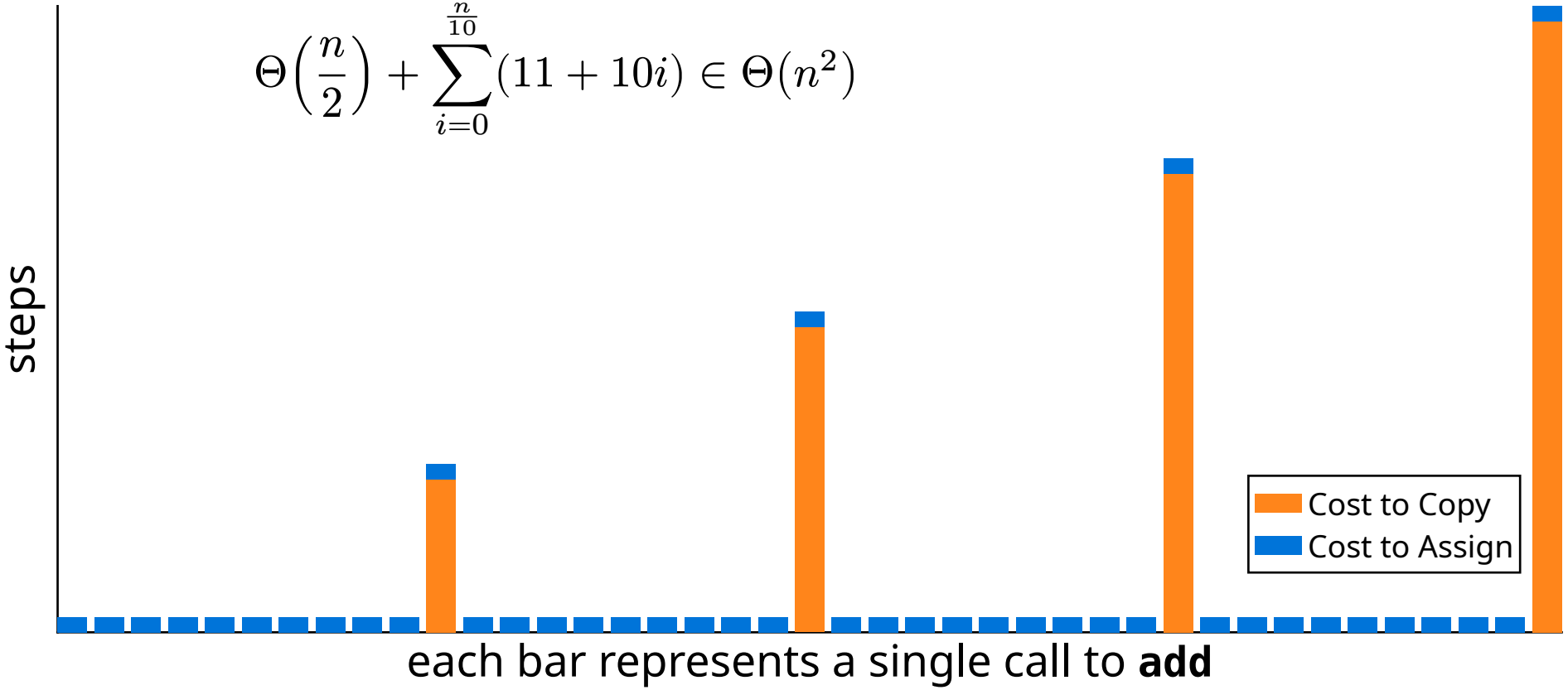
Initial Size = 10, **`newLength = data.length + 10`**



`ArrayList.add(elem)`: +10 Each Resize

Initial Size = 10, **`newLength = data.length + 10`**

$$\Theta\left(\frac{n}{2}\right) + \sum_{i=0}^{\frac{n}{10}} (11 + 10i) \in \Theta(n^2)$$



A Different Approach

Problem: If we increase **`data.length`** by a constant, n adds costs n^2 steps...

How else could we increase the size?

A Different Approach

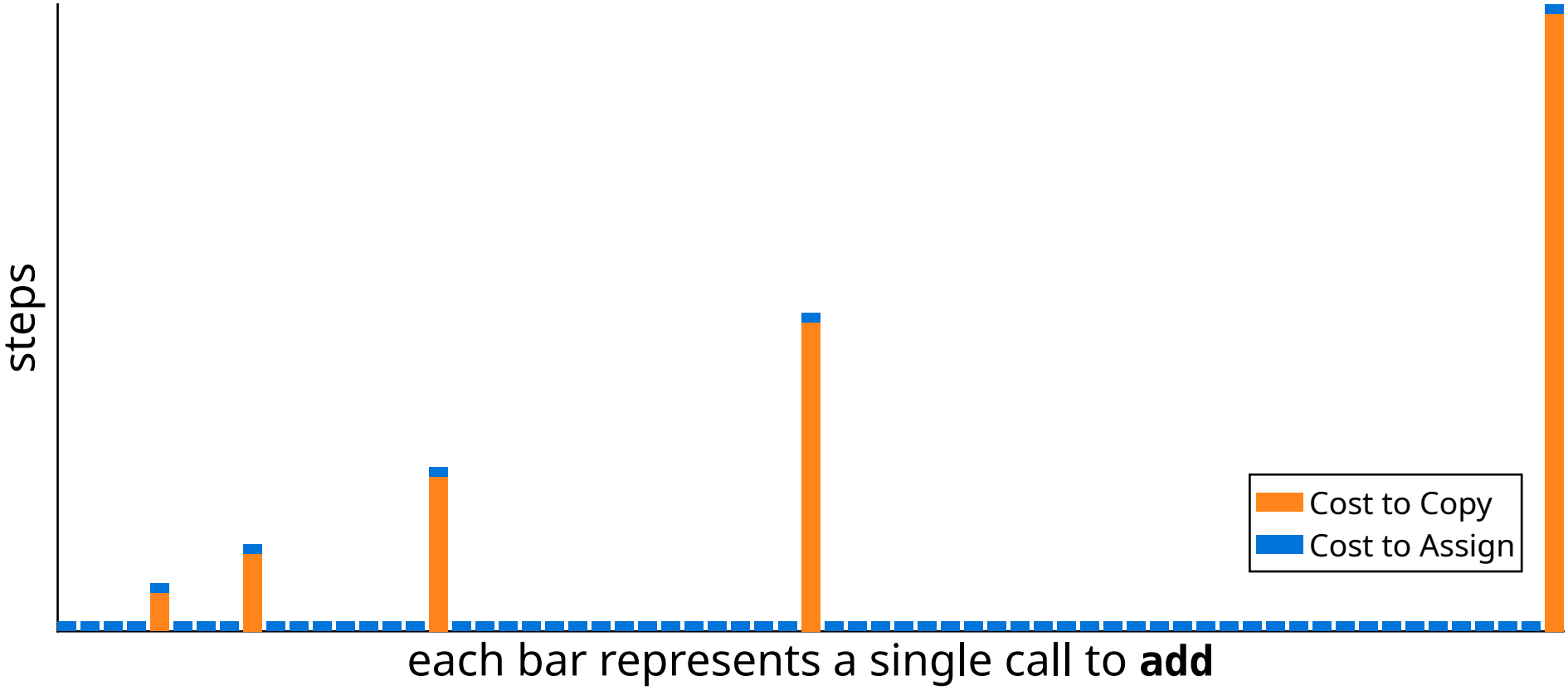
Problem: If we increase **`data.length`** by a constant, n adds costs n^2 steps...

How else could we increase the size?

Double it!

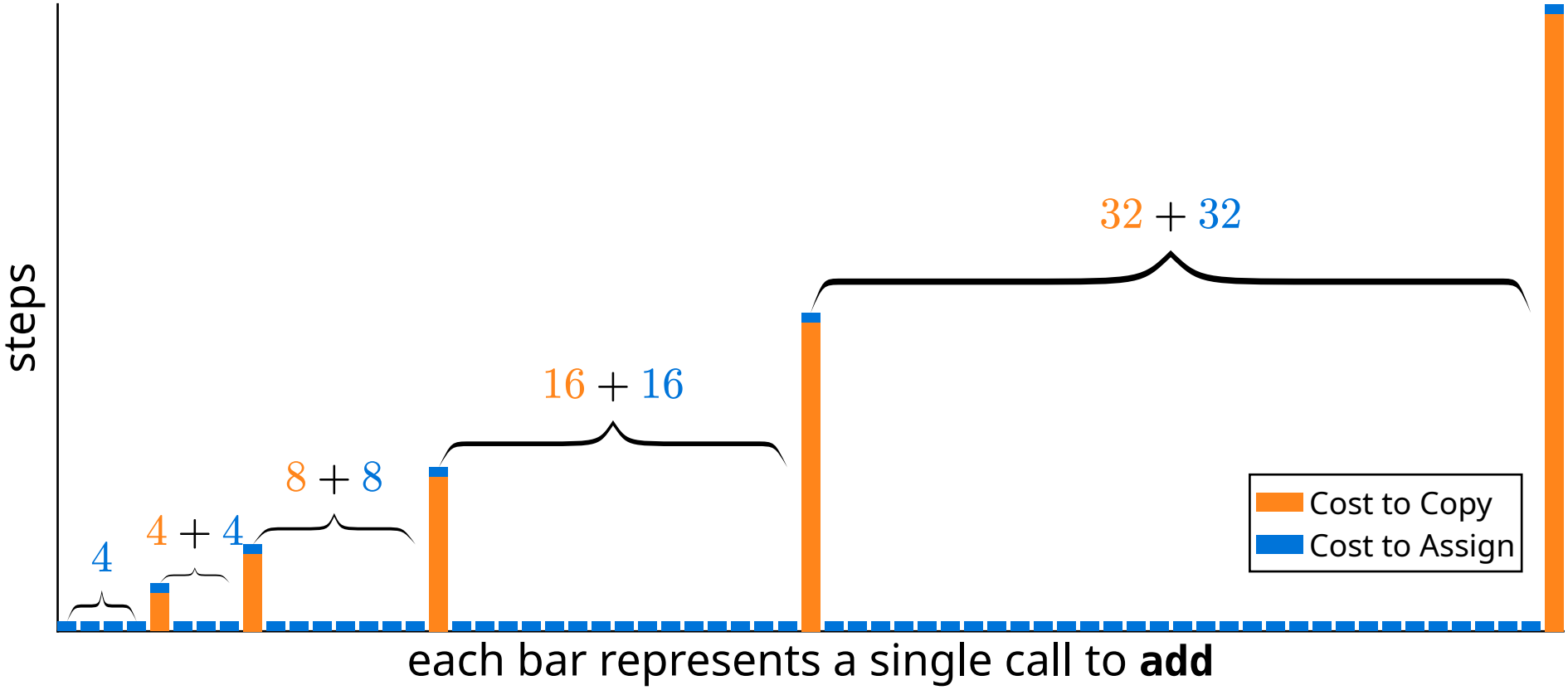
Doubling the Length Each Time

Initial Size = 4, **`newLength = data.length * 2`**



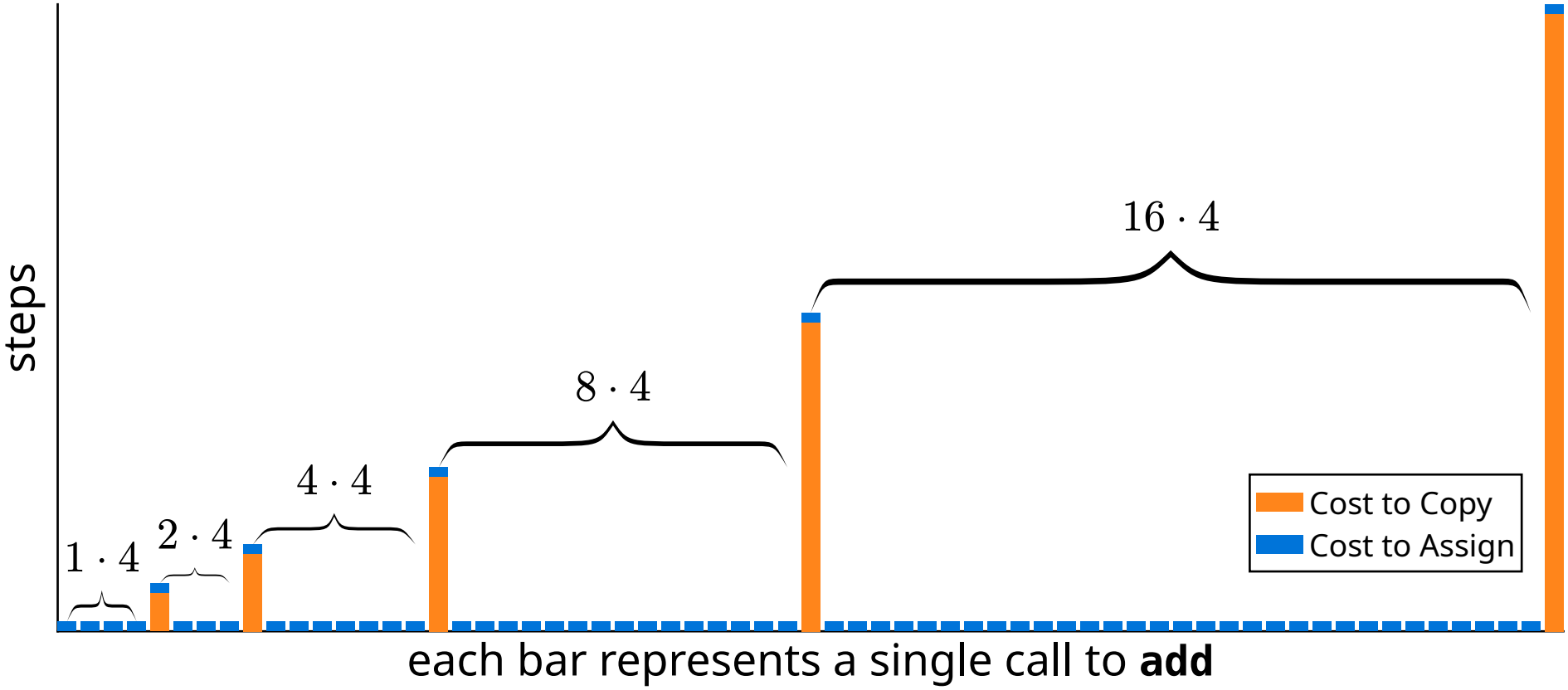
Doubling the Length Each Time

Initial Size = 4, **`newLength = data.length * 2`**



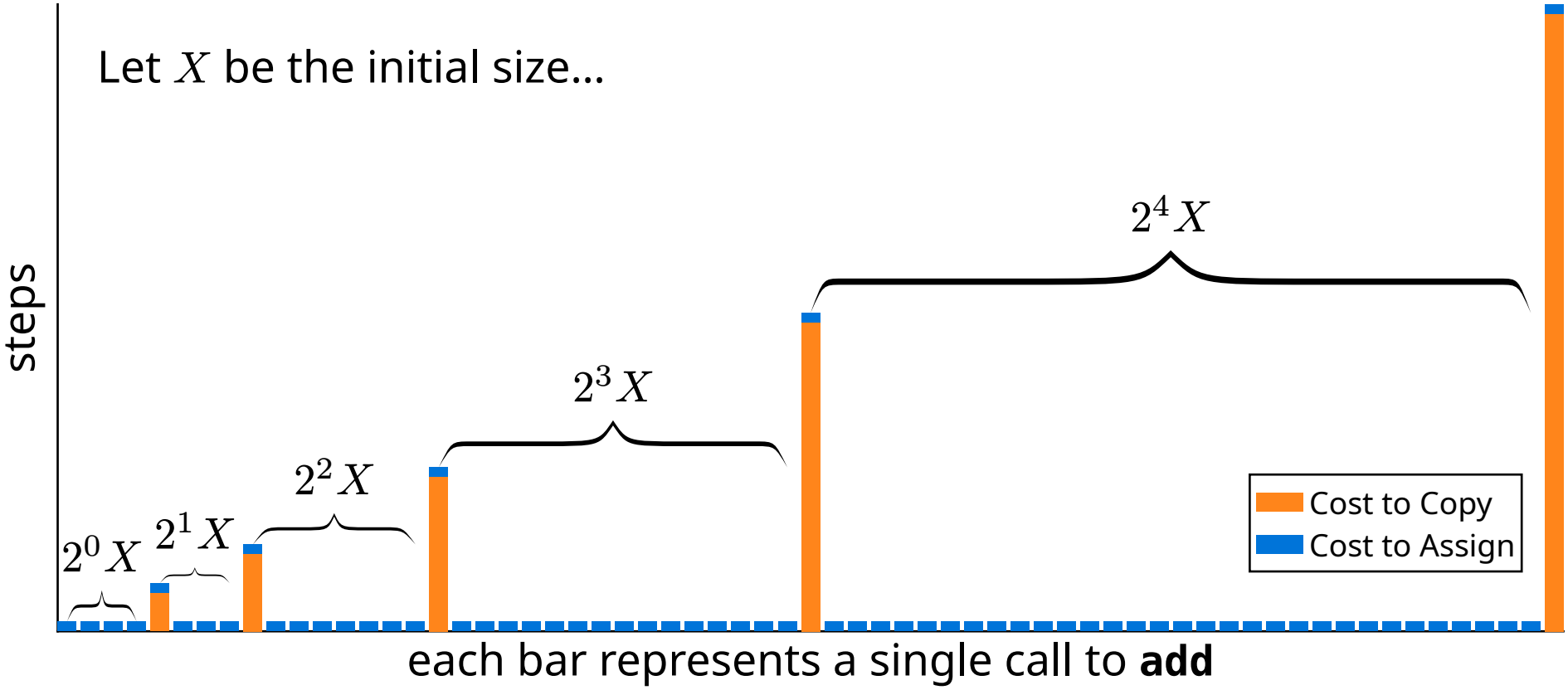
Doubling the Length Each Time

Initial Size = 4, **`newLength = data.length * 2`**



Doubling the Length Each Time

Initial Size = 4, **`newLength = data.length * 2`**



Analysis

Cost of the i^{th} region of **add** calls: $2^i X + 2^i X \in \Theta(2^i)$

How many regions of **add** calls are there for n total calls?

Analysis

Cost of the i^{th} region of **add** calls: $2^i X + 2^i X \in \Theta(2^i)$

How many regions of **add** calls are there for n total calls? $\Theta(\log(n))$

How many total steps?

Analysis

Cost of the i^{th} region of **add** calls: $2^i X + 2^i X \in \Theta(2^i)$

How many regions of **add** calls are there for n total calls? $\Theta(\log(n))$

How many total steps?

$$\text{cost of region } 0 + \text{cost of region } 1 + \ldots + \text{cost of region } \log(n) = \sum_{i=0}^{\log(n)} 2^i$$

Analysis

Cost of the i^{th} region of **add** calls: $2^i X + 2^i X \in \Theta(2^i)$

How many regions of **add** calls are there for n total calls? $\Theta(\log(n))$

How many total steps?

$$\text{cost of region } 0 + \text{cost of region } 1 + \ldots + \text{cost of region } \log(n) = \sum_{i=0}^{\log(n)} 2^i$$

$$\begin{aligned}\sum_{i=0}^{\log(n)} 2^i &= 2^{\log(n)+1} - 1 \\ &= 2 \cdot 2^{\log(n)} - 1 \\ &= 2n - 1 \in \Theta(n)\end{aligned}$$

Another Perspective

Imagine a completely full **`ArrayList`** with 16 elements

What do the next 16 calls to `add(e)` look like?

1. Allocate a single new array of size 32	1 step
2. Copy 16 elements to the new array	16 steps
3. Insert 16 elements into the open spaces	16 steps

Total: $2 \cdot 16 + 1$ steps

Another Perspective

Imagine a completely full **`ArrayList`** with 32 elements

What do the next 32 calls to `add(e)` look like?

1. Allocate a single new array of size 64	1 step
2. Copy 32 elements to the new array	32 steps
3. Insert 32 elements into the open spaces	32 steps

Total: $2 \cdot 32 + 1$ steps

Another Perspective

Imagine a completely full **`ArrayList`** with 64 elements

What do the next 64 calls to `add(e)` look like?

1. Allocate a single new array of size 128	1 step
2. Copy 64 elements to the new array	64 steps
3. Insert 64 elements into the open spaces	64 steps

Total: $2 \cdot 64 + 1$ steps

Another Perspective

Imagine a completely full **`ArrayList`** with n elements

What do the next n calls to `add(e)` look like?

1. Allocate a single new array of size $2n$	1 step
2. Copy n elements to the new array	n steps
3. Insert n elements into the open spaces	n steps

Total: $2 \cdot n + 1$ steps

Amortized Runtime

A single call to **`add(e)`** takes $O(n)$ steps (and $\Omega(1)$) steps...

But n calls to **`add(e)`** in a row take a total of $\Theta(n)$ steps!

Amortized Runtime

A single call to **`add(e)`** takes $O(n)$ steps (and $\Omega(1)$) steps...

But n calls to **`add(e)`** in a row take a total of $\Theta(n)$ steps!

If the total runtime of n calls to a function take $O(f(n))$ steps...

We say the **Amortized Runtime** of that function is $O\left(\frac{f(n)}{n}\right)$

Amortized Runtime

A single call to **`add(e)`** takes $O(n)$ steps (and $\Omega(1)$) steps...

But n calls to **`add(e)`** in a row take a total of $\Theta(n)$ steps!

If the total runtime of n calls to a function take $\Omega(f(n))$ steps...

We say the **Amortized Runtime** of that function is $\Omega\left(\frac{f(n)}{n}\right)$

Amortized Runtime

A single call to **`add(e)`** takes $O(n)$ steps (and $\Omega(1)$) steps...

But n calls to **`add(e)`** in a row take a total of $\Theta(n)$ steps!

If the total runtime of n calls to a function take $\Theta(f(n))$ steps...

We say the **Amortized Runtime** of that function is $\Theta\left(\frac{f(n)}{n}\right)$

Comparing List Implementations

If **`list`** is a **`LinkedList`**:

If **`list`** is an **`ArrayList`**:

Comparing List Implementations

If **`list`** is a **`LinkedList`**:

```
1 list.add(i);
```

Runtime of 1 call to **add**: $\Theta(1)$

Unqualified Runtime of add: $\Theta(1)$

If **`list`** is an **`ArrayList`**:

Comparing List Implementations

If **`list`** is a **`LinkedList`**:

```
1 list.add(i);
```

Runtime of 1 call to **add**: $\Theta(1)$

Unqualified Runtime of add: $\Theta(1)$

```
1 for (int i = 0; i < n; i++) {
2     list.add(i);
3 }
```

Runtime of n calls to **add**: $\Theta(n)$

Amortized Runtime of add: $\Theta(1)$

If **`list`** is an **`ArrayList`**:

Comparing List Implementations

If **`list`** is a **`LinkedList`**:

```
1 list.add(i);
```

Runtime of 1 call to **add**: $\Theta(1)$

Unqualified Runtime of add: $\Theta(1)$

```
1 for (int i = 0; i < n; i++) {
2     list.add(i);
3 }
```

Runtime of n calls to **add**: $\Theta(n)$

Amortized Runtime of add: $\Theta(1)$

If **`list`** is an **`ArrayList`**:

```
1 list.add(i);
```

Runtime of 1 call to **add**: $O(n)$

Unqualified Runtime of add: $O(n)$

Comparing List Implementations

If **`list`** is a **`LinkedList`**:

```
1 list.add(i);
```

Runtime of 1 call to **add**: $\Theta(1)$

Unqualified Runtime of add: $\Theta(1)$

```
1 for (int i = 0; i < n; i++) {
2     list.add(i);
3 }
```

Runtime of n calls to **add**: $\Theta(n)$

Amortized Runtime of add: $\Theta(1)$

If **`list`** is an **`ArrayList`**:

```
1 list.add(i);
```

Runtime of 1 call to **add**: $O(n)$

Unqualified Runtime of add: $O(n)$

```
1 for (int i = 0; i < n; i++) {
2     list.add(i);
3 }
```

Runtime of n calls to **add**: $\Theta(n)$

Amortized Runtime of add: $\Theta(1)$

`List` Summary (so far...)

	`Array`	`LinkedList` *(by index)*	`LinkedList` *(by reference)*
`get(i)`	$\Theta(1)$	$\Theta(i) \subset O(n)$	$\Theta(1)$
`set(i, v)`	$\Theta(1)$	$\Theta(i) \subset O(n)$	$\Theta(1)$
`size()`	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
`add(v)`	$O(n)$, Amortized $\Theta(1)$	$\Theta(1)$	$\Theta(1)$
`add(i, v)`	***???***	$\Theta(i) \subset O(n)$	$\Theta(1)$
`remove(idx)`	$O(n)$	$\Theta(i) \subset O(n)$	$\Theta(1)$

When is Amortized Relevant?

Imagine a coffee shop that sells coffee for **$1**

They also sell a reusable cup for **$3.50**, and filling it only costs **$0.50**

When is Amortized Relevant?

Imagine a coffee shop that sells coffee for **$1**

They also sell a reusable cup for **$3.50**, and filling it only costs **$0.50**

How much does it cost to buy coffee for a week?

- Without the reusable cup?
- With the reusable cup?

When is Amortized Relevant?

Imagine a coffee shop that sells coffee for **$1**

They also sell a reusable cup for **$3.50**, and filling it only costs **$0.50**

How much does it cost to buy coffee for a week?

- Without the reusable cup? **$7**
- With the reusable cup?

When is Amortized Relevant?

Imagine a coffee shop that sells coffee for **$1**

They also sell a reusable cup for **$3.50**, and filling it only costs **$0.50**

How much does it cost to buy coffee for a week?

- Without the reusable cup? **$7**
- With the reusable cup? **$7**

When is Amortized Relevant?

Imagine a coffee shop that sells coffee for **$1**

They also sell a reusable cup for **$3.50**, and filling it only costs **$0.50**

How much does it cost to buy coffee for a week?

- Without the reusable cup? **$7**
- With the reusable cup? **$7**

*The **amortized cost** of buying coffee with the reusable up still ends up being $1 per cup, even though you spend a lot more on the first day ($4).*

When is Amortized Relevant?

Imagine a coffee shop that sells coffee for **$1**

They also sell a reusable cup for **$3.50**, and filling it only costs **$0.50**

Should we buy the reusable cup?

When is Amortized Relevant?

Imagine a coffee shop that sells coffee for **$1**

They also sell a reusable cup for **$3.50**, and filling it only costs **$0.50**

Should we buy the reusable cup?

- If we plan to buy the coffee all week: buying the cup would be fine
- If we only plan to buy one or two cups: don’t buy the cup
- If we have a limit to how much we can spend each day: don’t buy the cup

When is Amortized Relevant?

Imagine a coffee shop that sells coffee for **$1**

They also sell a reusable cup for **$3.50**, and filling it only costs **$0.50**

Should we buy the reusable cup?

- If we plan to buy the coffee all week: buying the cup would be fine
- If we only plan to buy one or two cups: don't buy the cup
- If we have a limit to how much we can spend each day: don't buy the cup

These ideas can also be applied to choosing between ***`LinkedList`*** *and* ***`ArrayList`*** *for appending*

Why is Amortized Runtime Helfpul?

```
1 for (i = 0; i < n; i++) {
2     foo(i);
3 }
```

Why is Amortized Runtime Helfpul?

```
1 for (i = 0; i < n; i++) {
2     foo(i);
3 }
```

Imagine documentation tells you the **unqualified runtime** of **`foo`** is $O(n^3)$

What can you say about the runtime of the above code?

Why is Amortized Runtime Helfpul?

```
1 for (i = 0; i < n; i++) {
2     foo(i);
3 }
```

Imagine documentation tells you the **unqualified runtime** of **`foo`** is $O(n^3)$

What can you say about the runtime of the above code? $O(n^4)$

Why is Amortized Runtime Helfpul?

```
1 for (i = 0; i < n; i++) {
2     foo(i);
3 }
```

Imagine documentation tells you the **unqualified runtime** of **`foo`** is $O(n^3)$

What can you say about the runtime of the above code? $O(n^4)$

...but is that bound tight? We don't know!

When all you have is a tight upper bound on unqualified runtime, the shortcut of $(\#\ \text{iterations} \times \text{cost})$ is not guaranteed to give a tight bound!

You would need to do further analysis

Why is Amortized Runtime Helfpul?

```
1 for (i = 0; i < n; i++) {
2     foo(i);
3 }
```

Imagine documentation also tells you the **amortized runtime** of **`foo`** is $\Theta(n)$

What can you say about the runtime of the above code?

Why is Amortized Runtime Helfpul?

```
1 for (i = 0; i < n; i++) {
2     foo(i);
3 }
```

Imagine documentation also tells you the **amortized runtime** of **`foo`** is $\Theta(n)$

What can you say about the runtime of the above code? $\Theta(n^2)$

Why is Amortized Runtime Helfpul?

```
1 for (i = 0; i < n; i++) {
2     foo(i);
3 }
```

Imagine documentation also tells you the **amortized runtime** of **`foo`** is $\Theta(n)$

What can you say about the runtime of the above code? $\Theta(n^2)$

...and by definition this is a tight bound!

Knowing the amortized runtime makes analyzing loops easy!

Just use the shortcut $(\#\text{iterations} \times \text{amortized cost})$