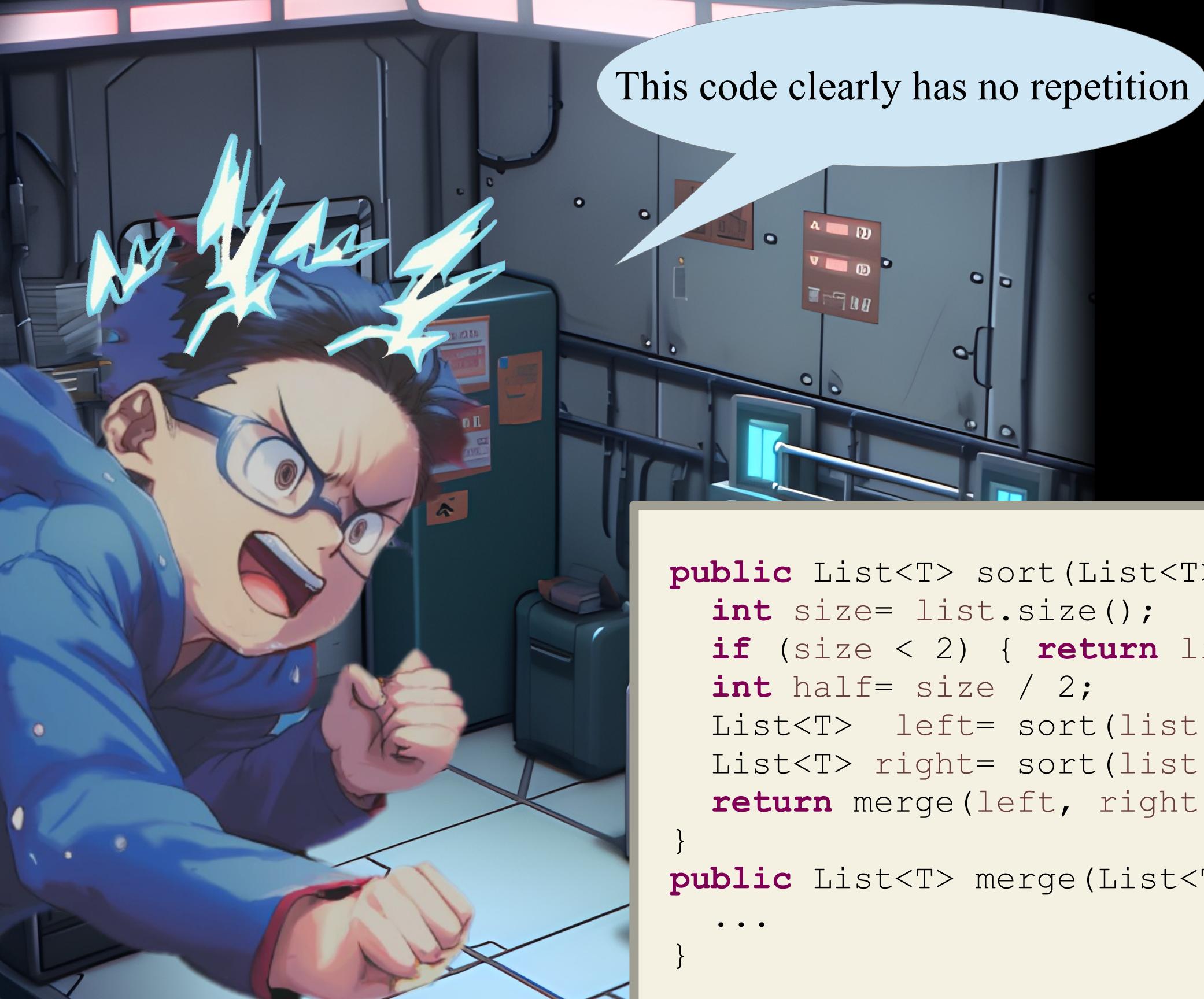


Power UP!



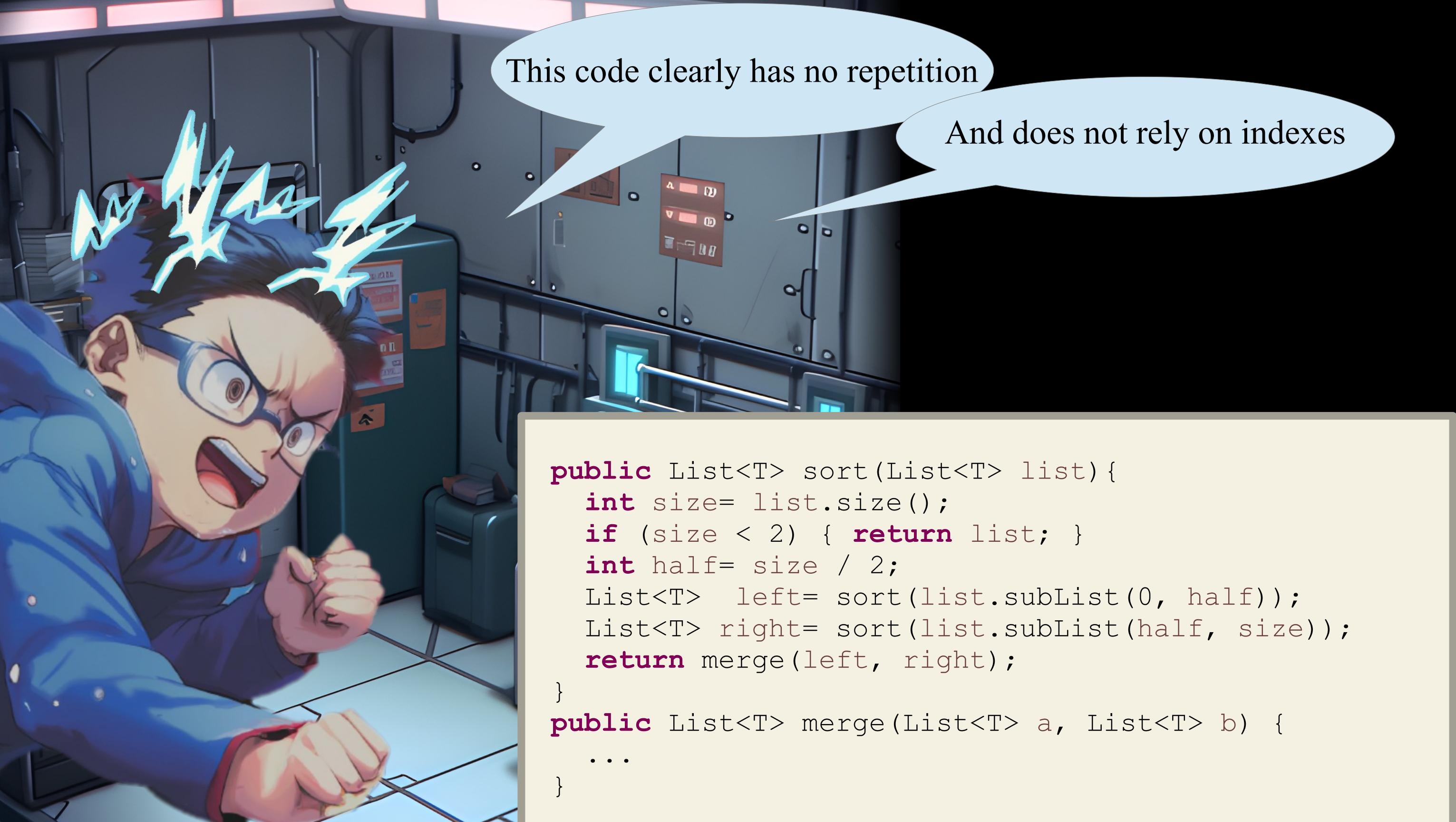


```
public List<T> sort(List<T> list) {
    int size= list.size();
    if (size < 2) { return list; }
    int half= size / 2;
    List<T> left= sort(list.subList(0, half));
    List<T> right= sort(list.subList(half, size));
    return merge(left, right);
}
public List<T> merge(List<T> a, List<T> b) {
    ...
}
```



This code clearly has no repetition

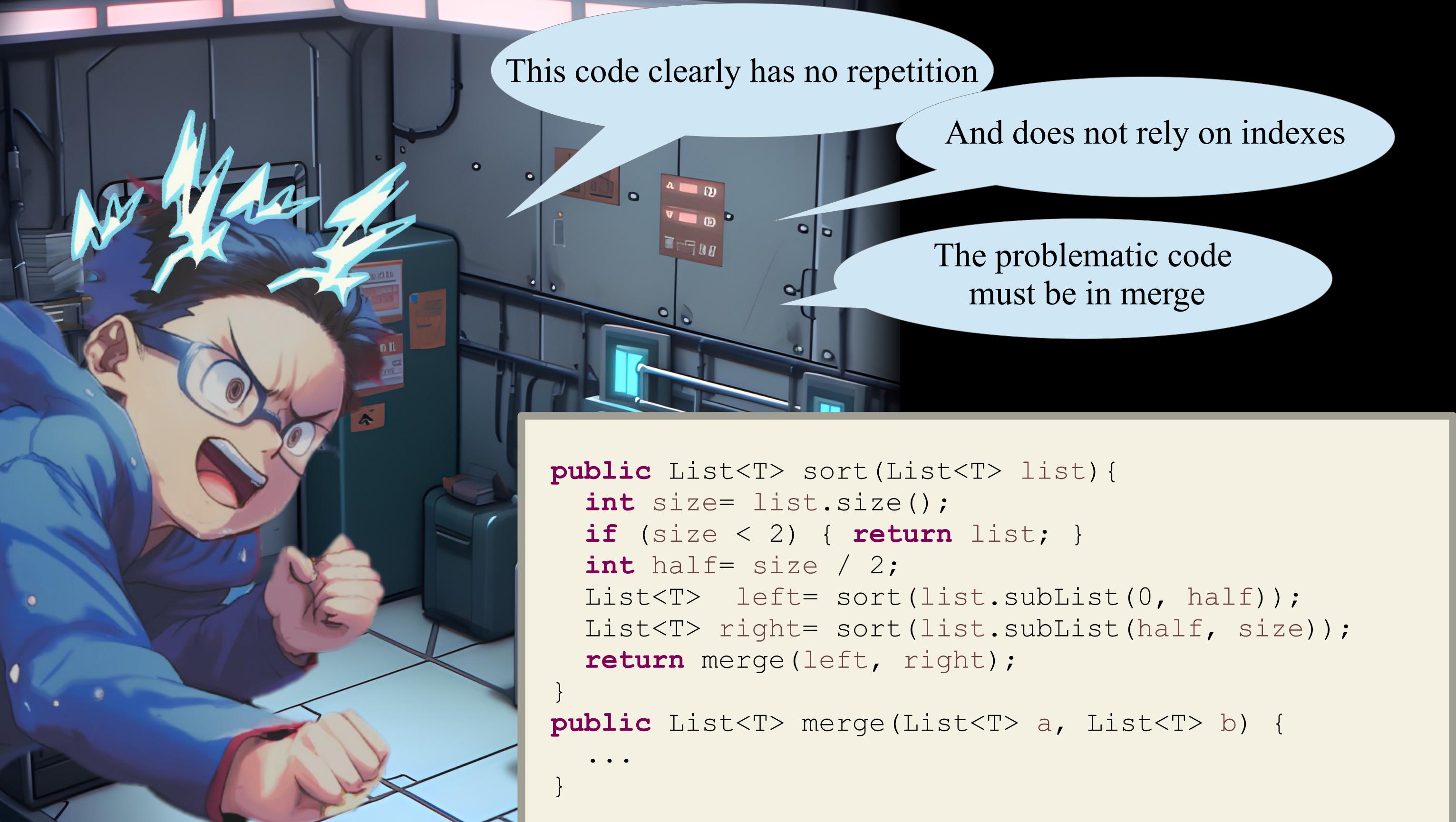
```
public List<T> sort(List<T> list) {  
    int size= list.size();  
    if (size < 2) { return list; }  
    int half= size / 2;  
    List<T> left= sort(list.subList(0, half));  
    List<T> right= sort(list.subList(half, size));  
    return merge(left, right);  
}  
public List<T> merge(List<T> a, List<T> b) {  
    ...  
}
```



This code clearly has no repetition

And does not rely on indexes

```
public List<T> sort(List<T> list) {  
    int size= list.size();  
    if (size < 2) { return list; }  
    int half= size / 2;  
    List<T> left= sort(list.subList(0, half));  
    List<T> right= sort(list.subList(half, size));  
    return merge(left, right);  
}  
public List<T> merge(List<T> a, List<T> b) {  
    ...  
}
```



This code clearly has no repetition

And does not rely on indexes

The problematic code
must be in merge

```
public List<T> sort(List<T> list) {  
    int size= list.size();  
    if (size < 2) { return list; }  
    int half= size / 2;  
    List<T> left= sort(list.subList(0, half));  
    List<T> right= sort(list.subList(half, size));  
    return merge(left, right);  
}  
public List<T> merge(List<T> a, List<T> b) {  
    ...  
}
```



```
public List<T> merge(List<T> a, List<T> b) {  
    List<T> res= new ArrayList<>(a.size() + b.size());  
    int indexA= 0;  
    int indexB= 0;  
    while (indexA < a.size() && indexB < b.size()) {  
        T currA= a.get(indexA);  
        T currB= b.get(indexB);  
        if (currA.compareTo(currB) <= 0) {  
            res.add(currA);  
            indexA++;  
        }  
        else {  
            res.add(currB);  
            indexB++;  
        }  
    }  
    while (indexA < a.size()) {  
        res.add(a.get(indexA));  
        indexA++;  
    }  
    while (indexB < b.size()) {  
        res.add(b.get(indexB));  
        indexB++;  
    }  
    return res;  
}
```

Tons of duplicated code here!



```
public List<T> merge(List<T> a, List<T> b) {  
    List<T> res= new ArrayList<>(a.size() + b.size());  
    int indexA= 0;  
    int indexB= 0;  
    while (indexA < a.size() && indexB < b.size()) {  
        T currA= a.get(indexA);  
        T currB= b.get(indexB);  
        if (currA.compareTo(currB) <= 0) {  
            res.add(currA);  
            indexA++;  
        }  
        else {  
            res.add(currB);  
            indexB++;  
        }  
    }  
    while (indexA < a.size()) {  
        res.add(a.get(indexA));  
        indexA++;  
    }  
    while (indexB < b.size()) {  
        res.add(b.get(indexB));  
        indexB++;  
    }  
    return res;  
}
```



Tons of duplicated code here!

The body of the if and the else are quite similar

```
public List<T> merge(List<T> a, List<T> b) {  
    List<T> res= new ArrayList<>(a.size() + b.size());  
    int indexA= 0;  
    int indexB= 0;  
    while (indexA < a.size() && indexB < b.size()) {  
        T currA= a.get(indexA);  
        T currB= b.get(indexB);  
        if (currA.compareTo(currB) <= 0) {  
            res.add(currA);  
            indexA++;  
        }  
        else {  
            res.add(currB);  
            indexB++;  
        }  
    }  
    while (indexA < a.size()) {  
        res.add(a.get(indexA));  
        indexA++;  
    }  
    while (indexB < b.size()) {  
        res.add(b.get(indexB));  
        indexB++;  
    }  
    return res;  
}
```



Same for the two conclusive whiles

```
public List<T> merge(List<T> a, List<T> b) {  
    List<T> res= new ArrayList<>(a.size() + b.size());  
    int indexA= 0;  
    int indexB= 0;  
    while (indexA < a.size() && indexB < b.size()) {  
        T currA= a.get(indexA);  
        T currB= b.get(indexB);  
        if (currA.compareTo(currB) <= 0) {  
            res.add(currA);  
            indexA++;  
        }  
        else {  
            res.add(currB);  
            indexB++;  
        }  
    }  
    while (indexA < a.size()) {  
        res.add(a.get(indexA));  
        indexA++;  
    }  
    while (indexB < b.size()) {  
        res.add(b.get(indexB));  
        indexB++;  
    }  
    return res;  
}
```



```
public List<T> merge(List<T> a, List<T> b) {  
    List<T> res= new ArrayList<>(a.size() + b.size());  
    int indexA= 0;  
    int indexB= 0;  
    while (indexA < a.size() && indexB < b.size()) {  
        T currA= a.get(indexA);  
        T currB= b.get(indexB);  
        if (currA.compareTo(currB) <= 0) {  
            res.add(currA);  
            indexA++;  
        }  
        else {  
            res.add(currB);  
            indexB++;  
        }  
    }  
    while (indexA < a.size()) {  
        res.add(a.get(indexA));  
        indexA++;  
    }  
    while (indexB < b.size()) {  
        res.add(b.get(indexB));  
        indexB++;  
    }  
    return res;  
}
```

And all the code is about indexes!



```
public List<T> merge(List<T> a, List<T> b) {  
    List<T> res= new ArrayList<>(a.size() + b.size());  
    int indexA= 0;  
    int indexB= 0;  
    while (indexA < a.size() && indexB < b.size()) {  
        T currA= a.get(indexA);  
        T currB= b.get(indexB);  
        if (currA.compareTo(currB) <= 0) {  
            res.add(currA);  
            indexA++;  
        }  
        else {  
            res.add(currB);  
            indexB++;  
        }  
    }  
    while (indexA < a.size()) {  
        res.add(a.get(indexA));  
        indexA++;  
    }  
    while (indexB < b.size()) {  
        res.add(b.get(indexB));  
        indexB++;  
    }  
    return res;  
}
```



And all the code is about indexes!

We should use iterators instead.

```
public List<T> merge(List<T> a, List<T> b) {  
    List<T> res= new ArrayList<>(a.size() + b.size());  
    int indexA= 0;  
    int indexB= 0;  
    while (indexA < a.size() && indexB < b.size()) {  
        T currA= a.get(indexA);  
        T currB= b.get(indexB);  
        if (currA.compareTo(currB) <= 0) {  
            res.add(currA);  
            indexA++;  
        }  
        else {  
            res.add(currB);  
            indexB++;  
        }  
    }  
    while (indexA < a.size()) {  
        res.add(a.get(indexA));  
        indexA++;  
    }  
    while (indexB < b.size()) {  
        res.add(b.get(indexB));  
        indexB++;  
    }  
    return res;  
}
```



And all the code is about indexes!

We should use iterators instead.

```
public List<T> merge(List<T> a, List<T> b) {  
    List<T> res= new ArrayList<>(a.size() + b.size());  
    int indexA= 0;  
    int indexB= 0;  
    while (indexA < a.size() && indexB < b.size()) {  
        T currA= a.get(indexA);  
        T currB= b.get(indexB);  
        if (currA.compareTo(currB) <= 0) {  
            res.add(currA);  
            indexA++;  
        }  
        else {  
            res.add(currB);  
            indexB++;  
        }  
    }  
    while (indexA < a.size()) {  
        res.add(a.get(indexA));  
        indexA++;  
    }  
    while (indexB < b.size()) {  
        res.add(b.get(indexB));  
        indexB++;  
    }  
    return res;  
}
```



A cartoon illustration of a man with glasses and a blue jacket, looking shocked with his mouth open and hands clenched. He has blue energy-like streaks above his head.

And all the code is about indexes!

We should use iterators instead.

```
public List<T> merge(List<T> a, List<T> b) {  
    List<T> res= new ArrayList<>(a.size() + b.size());  
    int indexA= 0;  
    int indexB= 0;  
    while (indexA < a.size() && indexB < b.size()) {  
        T currA= a.get(indexA);  
        T currB= b.get(indexB);  
        if (currA.compareTo(currB) <= 0) {  
            res.add(currA);  
            indexA++;  
        }  
        else {  
            res.add(currB);  
            indexB++;  
        }  
    }  
    while (indexA < a.size()) {  
        res.add(a.get(indexA));  
        indexA++;  
    }  
    while (indexB < b.size()) {  
        res.add(b.get(indexB));  
        indexB++;  
    }  
    return res;  
}
```

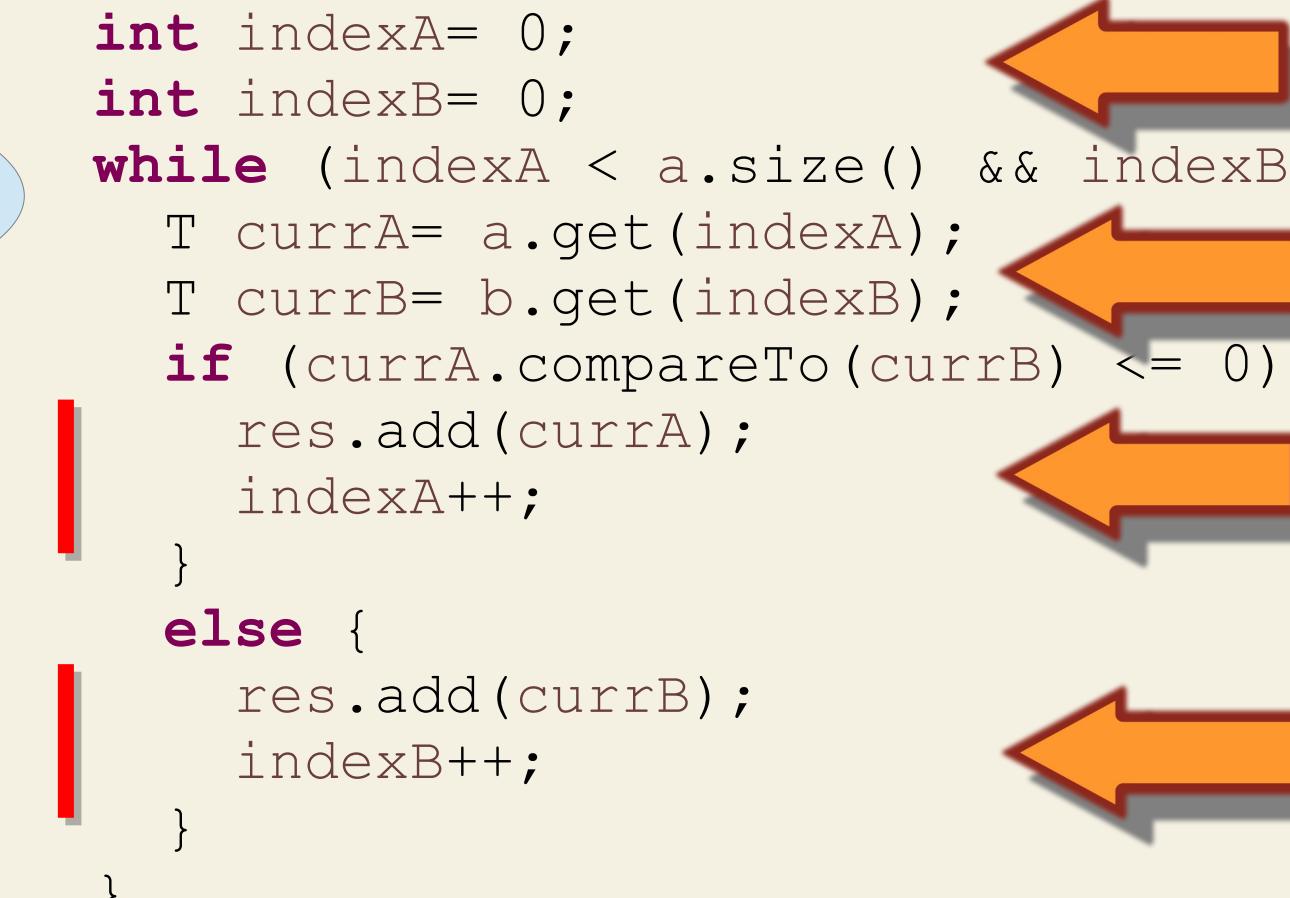
Two orange arrows point to the first two lines of the inner while loop, highlighting the use of indices.

A cartoon illustration of a man with glasses and a blue jacket, looking shocked with his mouth open and hands clenched. He has blue energy-like streaks above his head.

And all the code is about indexes!

We should use iterators instead.

```
public List<T> merge(List<T> a, List<T> b) {  
    List<T> res= new ArrayList<>(a.size() + b.size());  
    int indexA= 0;  
    int indexB= 0;  
    while (indexA < a.size() && indexB < b.size()) {  
        T currA= a.get(indexA);  
        T currB= b.get(indexB);  
        if (currA.compareTo(currB) <= 0) {  
            res.add(currA);  
            indexA++;  
        }  
        else {  
            res.add(currB);  
            indexB++;  
        }  
    }  
    while (indexA < a.size()) {  
        res.add(a.get(indexA));  
        indexA++;  
    }  
    while (indexB < b.size()) {  
        res.add(b.get(indexB));  
        indexB++;  
    }  
    return res;  
}
```

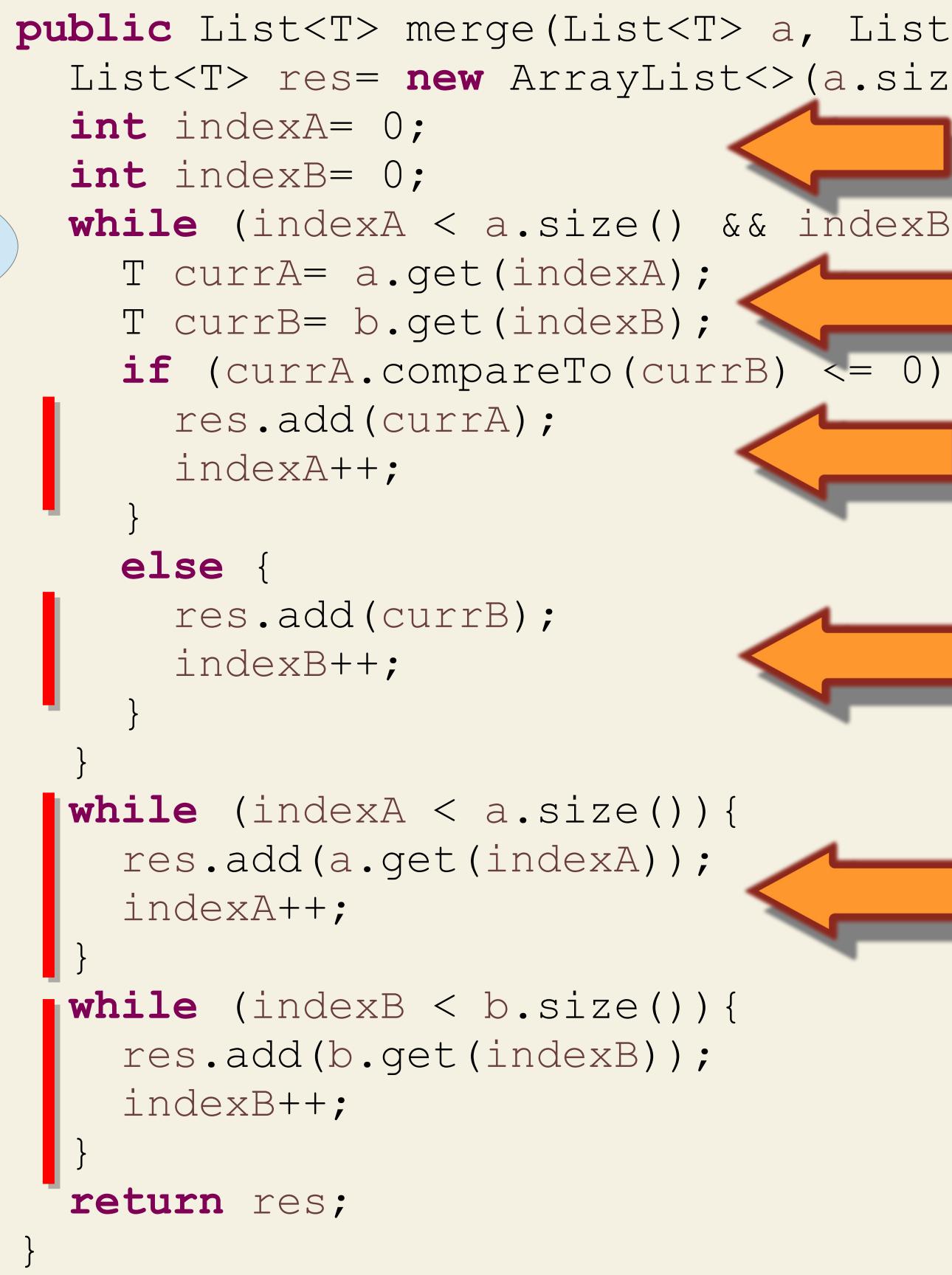
Four orange arrows point from the code blocks to the corresponding speech bubbles. The first arrow points to the first code block, the second to the second, the third to the third, and the fourth to the fourth.



And all the code is about indexes!

We should use iterators instead.

```
public List<T> merge(List<T> a, List<T> b) {  
    List<T> res= new ArrayList<>(a.size() + b.size());  
    int indexA= 0;  
    int indexB= 0;  
    while (indexA < a.size() && indexB < b.size()) {  
        T currA= a.get(indexA);  
        T currB= b.get(indexB);  
        if (currA.compareTo(currB) <= 0) {  
            res.add(currA);  
            indexA++;  
        }  
        else {  
            res.add(currB);  
            indexB++;  
        }  
    }  
    while (indexA < a.size()) {  
        res.add(a.get(indexA));  
        indexA++;  
    }  
    while (indexB < b.size()) {  
        res.add(b.get(indexB));  
        indexB++;  
    }  
    return res;  
}
```

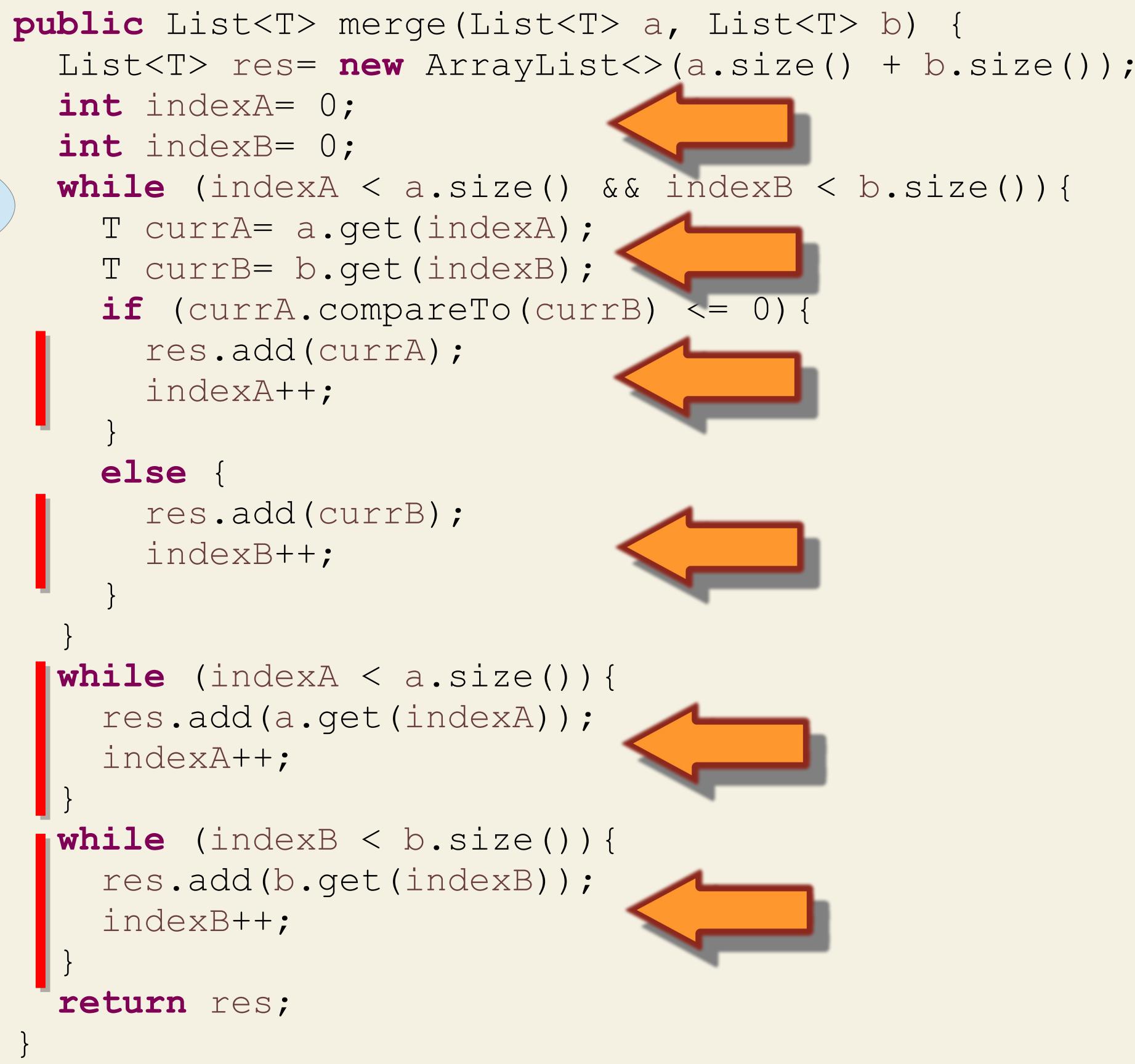


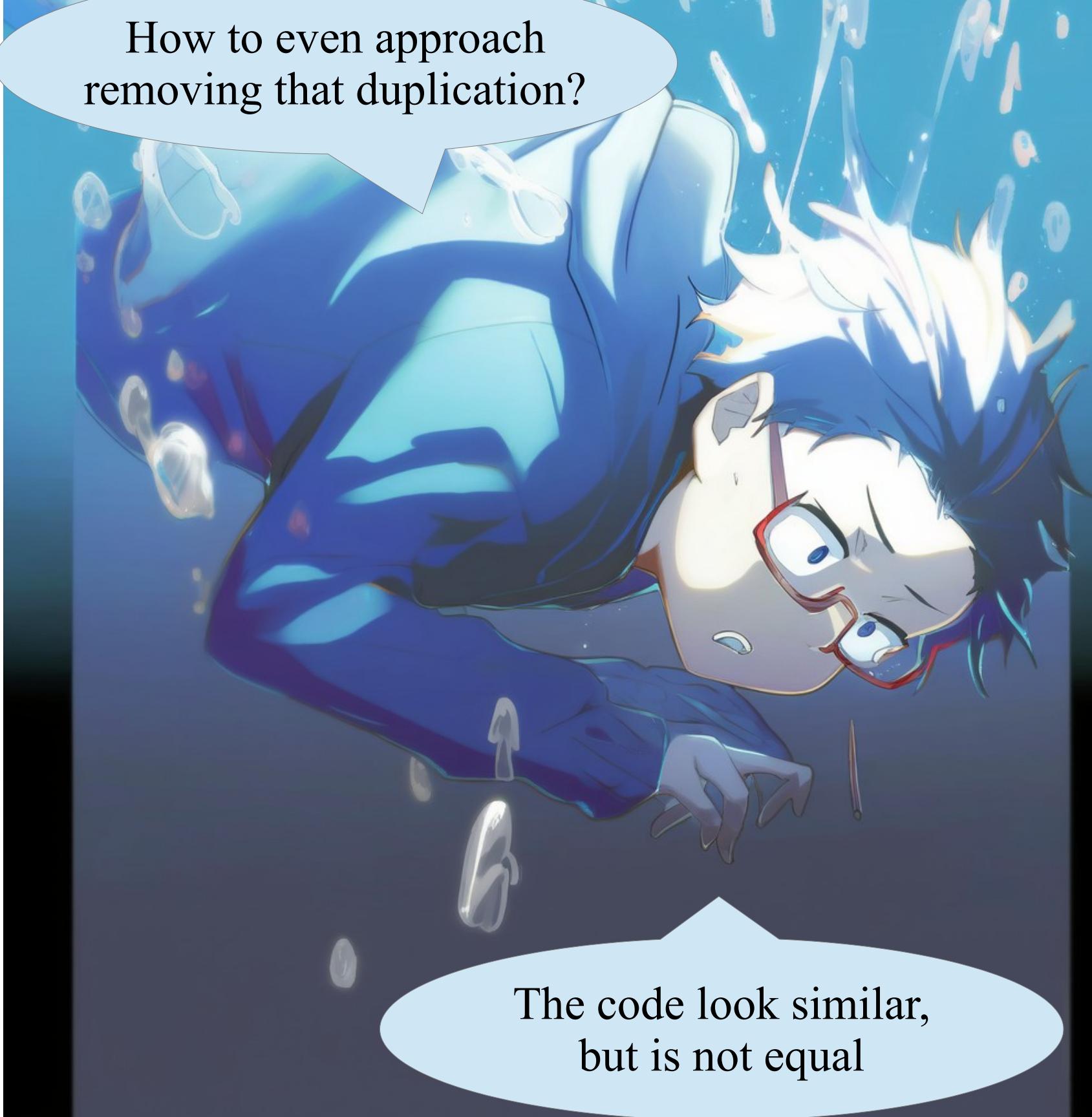
A cartoon illustration of a man with glasses and a blue jacket, looking shocked with sweat drops on his forehead. He is leaning forward, holding a piece of paper or a small object in his hand.

And all the code is about indexes!

We should use iterators instead.

```
public List<T> merge(List<T> a, List<T> b) {  
    List<T> res= new ArrayList<>(a.size() + b.size());  
    int indexA= 0;  
    int indexB= 0;  
    while (indexA < a.size() && indexB < b.size()) {  
        T currA= a.get(indexA);  
        T currB= b.get(indexB);  
        if (currA.compareTo(currB) <= 0) {  
            res.add(currA);  
            indexA++;  
        }  
        else {  
            res.add(currB);  
            indexB++;  
        }  
    }  
    while (indexA < a.size()) {  
        res.add(a.get(indexA));  
        indexA++;  
    }  
    while (indexB < b.size()) {  
        res.add(b.get(indexB));  
        indexB++;  
    }  
    return res;  
}
```

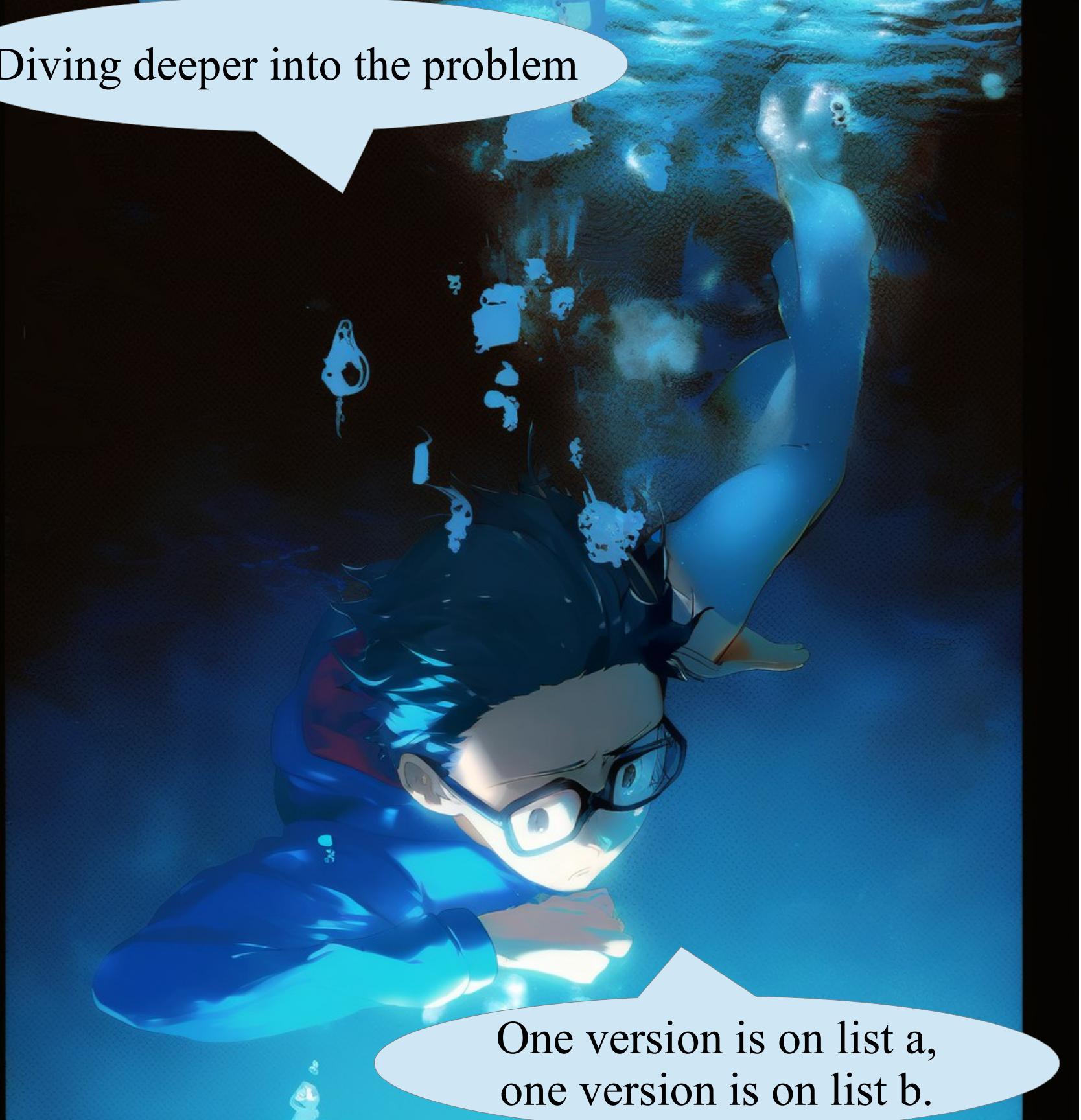
The diagram shows the original code structure on the left and the refactored code structure on the right. Orange arrows point from the original code's opening brace to the corresponding opening brace in the refactored code. They also point from the original code's closing brace to the corresponding closing brace in the refactored code. This visualizes how the code has been transformed into a more iterator-based form.

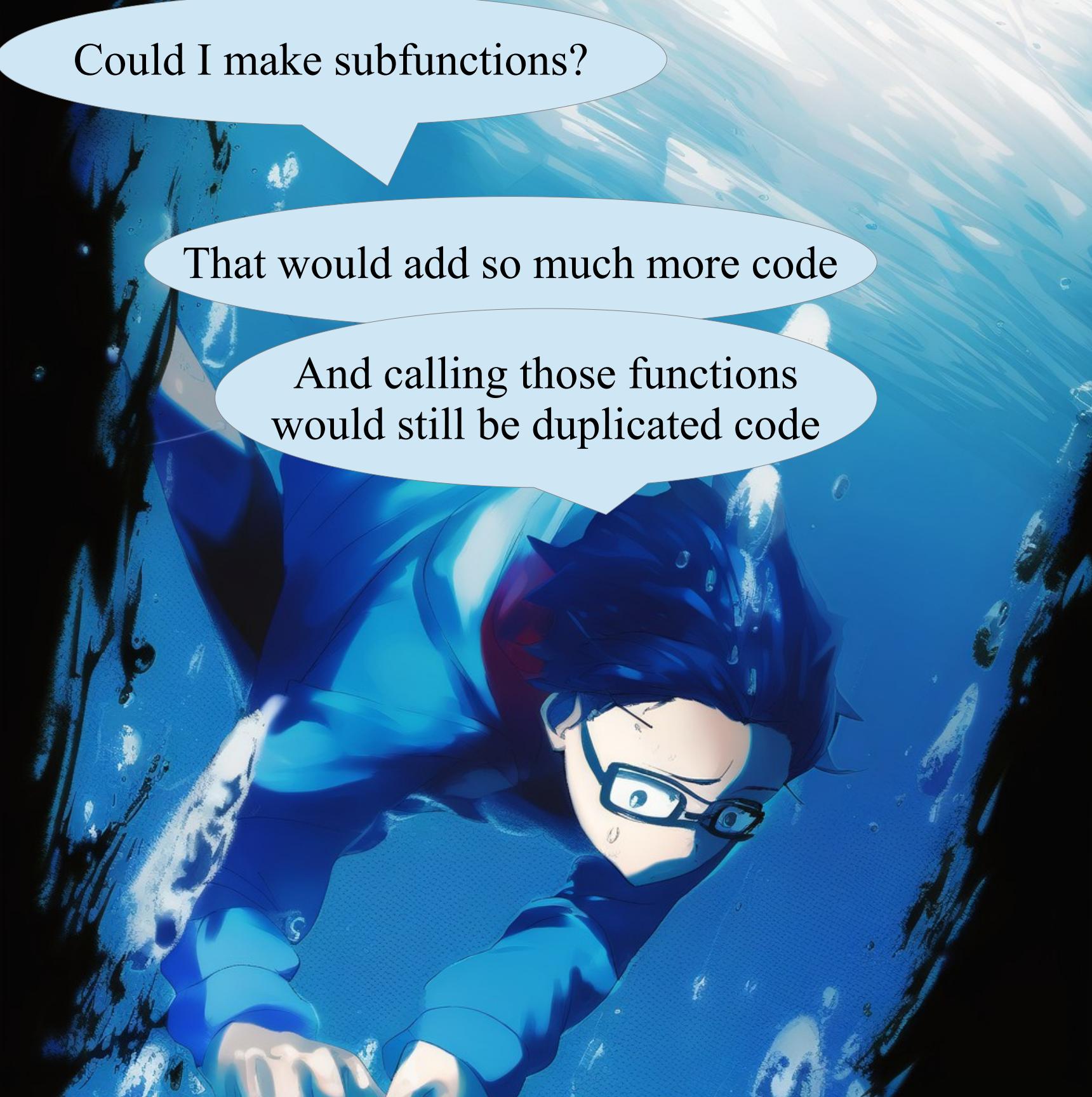


How to even approach
removing that duplication?

The code look similar,
but is not equal

Diving deeper into the problem





Could I make subfunctions?

That would add so much more code

And calling those functions
would still be duplicated code





I'm nearly out of air



I'm nearly out of air

And yet, this problem
will not budge





I need to change my perspective



I need to change my perspective

Can I swap it with
another point of view?



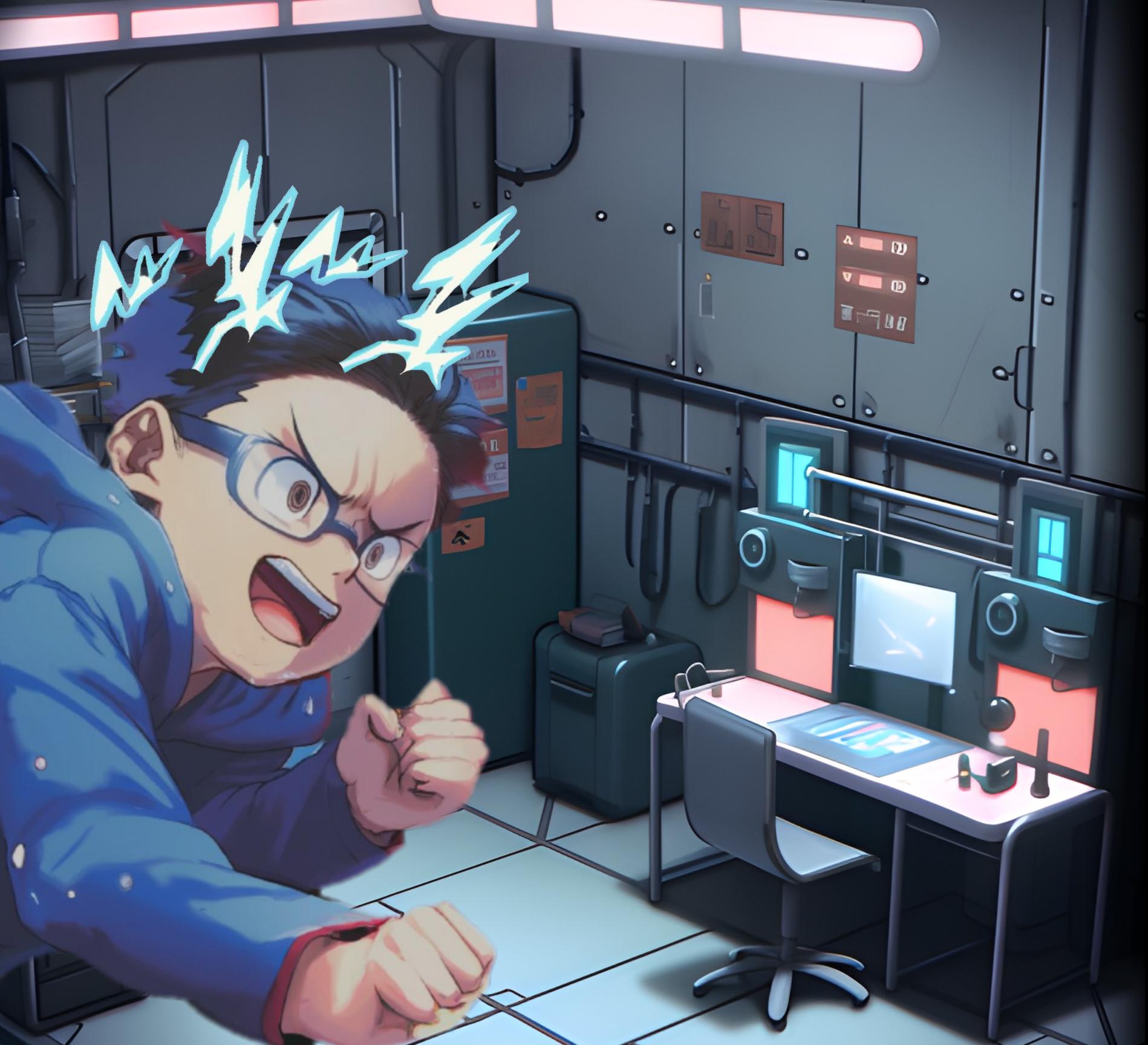


Swap! Here it is again!



Swap! Here it is again!

I can swap the iterators
for a and b instead of
duplicating the code





```
public List<T> merge(List<T> a, List<T> b) {  
    List<T>res= new ArrayList<>(a.size() + b.size());  
    Iterator<T> ai= a.iterator();  
    Iterator<T> bi= b.iterator();  
    T currA= ai.next(); //one element pending  
    while(bi.hasNext()) {  
        T currB= bi.next(); //two elements pending  
        if(currB.compareTo(currA) <= 0) {  
            res.add(currB); //only one element pending  
            continue;  
        }  
        res.add(currA); //only one element pending  
        currA = currB; //that is now called currA  
        var tmp= ai; ai = bi; bi = tmp; //swap ai/bi  
    }  
    res.add(currA); //zero elements pending  
    ai.forEachRemaining(res::add);  
    return res;  
}
```



```
public List<T> merge(List<T> a, List<T> b) {  
    List<T>res= new ArrayList<>(a.size() + b.size());  
    Iterator<T> ai= a.iterator();  
    Iterator<T> bi= b.iterator();  
    T currA= ai.next(); //one element pending  
    while(bi.hasNext()) {  
        T currB= bi.next(); //two elements pending  
        if(currB.compareTo(currA) <= 0) {  
            res.add(currB); //only one element pending  
            continue;  
        }  
        res.add(currA); //only one element pending  
        currA = currB; //that is now called currA  
        var tmp= ai; ai = bi; bi = tmp; //swap ai/bi  
    }  
    res.add(currA); //zero elements pending  
    ai.forEachRemaining(res::add);  
    return res;  
}
```



```
public List<T> merge(List<T> a, List<T> b) {  
    List<T> res = new ArrayList<>(a.size() + b.size());  
    Iterator<T> ai = a.iterator();  
    Iterator<T> bi = b.iterator();  
    T currA = ai.next(); //one element pending  
    while (bi.hasNext()) {  
        T currB = bi.next(); //two elements pending  
        if (currB.compareTo(currA) <= 0) {  
            res.add(currB); //only one element pending  
            continue;  
        }  
        res.add(currA); //only one element pending  
        currA = currB; //that is now called currA  
        var tmp = ai; ai = bi; bi = tmp; //swap ai/bi  
    }  
    res.add(currA); //zero elements pending  
    ai.forEachRemaining(res::add);  
    return res;  
}
```



```
public List<T> merge(List<T> a, List<T> b) {  
    List<T>res= new ArrayList<>(a.size() + b.size());  
    Iterator<T> ai= a.iterator();  
    Iterator<T> bi= b.iterator();  
    T currA= ai.next(); //one element pending  
    while(bi.hasNext()) {  
        T currB= bi.next(); //two elements pending  
        if(currB.compareTo(currA) <= 0) {  
            res.add(currB); //only one element pending  
            continue;  
        }  
        res.add(currA); //only one element pending  
        currA = currB; //that is now called currA  
        var tmp= ai; ai = bi; bi = tmp; //swap ai/bi  
    }  
    res.add(currA); //zero elements pending  
    ai.forEachRemaining(res::add);  
    return res;  
}
```



```
public List<T> merge(List<T> a, List<T> b) {  
    List<T>res= new ArrayList<>(a.size() + b.size());  
    Iterator<T> ai= a.iterator();  
    Iterator<T> bi= b.iterator();  
    T currA= ai.next(); //one element pending  
    while(bi.hasNext()) {  
        T currB= bi.next(); //two elements pending  
        if(currB.compareTo(currA) <= 0) {  
            res.add(currB); //only one element pending  
            continue;  
        }  
        res.add(currA); //only one element pending  
        currA = currB; //that is now called currA  
        var tmp= ai; ai = bi; bi = tmp; //swap ai/bi  
    }  
    res.add(currA); //zero elements pending  
    ai.forEachRemaining(res::add);  
    return res;  
}
```

The outer structure is the same.



```
public List<T> merge(List<T> a, List<T> b) {  
    List<T>res= new ArrayList<>(a.size() + b.size());  
    Iterator<T> ai= a.iterator();  
    Iterator<T> bi= b.iterator();  
    T currA= ai.next();//one element pending  
    while(bi.hasNext()) {  
        T currB= bi.next();//two elements pending  
        if(currB.compareTo(currA) <= 0) {  
            res.add(currB);//only one element pending  
            continue;  
        }  
        res.add(currA);//only one element pending  
        currA = currB; //that is now called currA  
        var tmp= ai; ai = bi; bi = tmp;//swap ai/bi  
    }  
    res.add(currA);//zero elements pending  
    ai.forEachRemaining(res::add);  
    return res;  
}
```



The outer structure is the same.

But, we use iterators
instead of indexes

```
public List<T> merge(List<T> a, List<T> b) {  
    List<T> res = new ArrayList<>(a.size() + b.size());  
    Iterator<T> ai = a.iterator();  
    Iterator<T> bi = b.iterator();  
    T currA = ai.next(); //one element pending  
    while (bi.hasNext()) {  
        T currB = bi.next(); //two elements pending  
        if (currB.compareTo(currA) <= 0) {  
            res.add(currB); //only one element pending  
            continue;  
        }  
        res.add(currA); //only one element pending  
        currA = currB; //that is now called currA  
        var tmp = ai; ai = bi; bi = tmp; //swap ai/bi  
    }  
    res.add(currA); //zero elements pending  
    ai.forEachRemaining(res::add);  
    return res;  
}
```





```
public List<T> merge(List<T> a, List<T> b) {  
    List<T>res= new ArrayList<>(a.size() + b.size());  
    Iterator<T> ai= a.iterator();  
    Iterator<T> bi= b.iterator();  
    T currA= ai.next(); //one element pending  
    while(bi.hasNext()) {  
        T currB= bi.next(); //two elements pending  
        if(currB.compareTo(currA) <= 0) {  
            res.add(currB); //only one element pending  
            continue;  
        }  
        res.add(currA); //only one element pending  
        currA = currB; //that is now called currA  
        var tmp= ai; ai = bi; bi = tmp; //swap ai/bi  
    }  
    res.add(currA); //zero elements pending  
    ai.forEachRemaining(res::add);  
    return res;  
}
```

We then extract an element from ai



```
public List<T> merge(List<T> a, List<T> b) {  
    List<T> res = new ArrayList<>(a.size() + b.size());  
    Iterator<T> ai = a.iterator();  
    Iterator<T> bi = b.iterator();  
    T currA = ai.next(); //one element pending  
    while (bi.hasNext()) {  
        T currB = bi.next(); //two elements pending  
        if (currB.compareTo(currA) <= 0) {  
            res.add(currB); //only one element pending  
            continue;  
        }  
        res.add(currA); //only one element pending  
        currA = currB; //that is now called currA  
        var tmp = ai; ai = bi; bi = tmp; //swap ai/bi  
    }  
    res.add(currA); //zero elements pending  
    ai.forEachRemaining(res::add);  
    return res;  
}
```



We then extract an element from ai

We can do it because we know that a is not an empty list.



```
public List<T> merge(List<T> a, List<T> b) {  
    List<T> res = new ArrayList<>(a.size() + b.size());  
    Iterator<T> ai = a.iterator();  
    Iterator<T> bi = b.iterator();  
    T currA = ai.next(); //one element pending  
    while (bi.hasNext()) {  
        T currB = bi.next(); //two elements pending  
        if (currB.compareTo(currA) <= 0) {  
            res.add(currB); //only one element pending  
            continue;  
        }  
        res.add(currA); //only one element pending  
        currA = currB; //that is now called currA  
        var tmp = ai; ai = bi; bi = tmp; //swap ai/bi  
    }  
    res.add(currA); //zero elements pending  
    ai.forEachRemaining(res::add);  
    return res;  
}
```





We then extract an element from ai

We can do it because we know that a is not an empty list.



```
public List<T> merge(List<T> a, List<T> b) {  
    List<T> res = new ArrayList<>(a.size() + b.size());  
    Iterator<T> ai = a.iterator();  
    Iterator<T> bi = b.iterator();  
    T currA = ai.next(); //one element pending  
    while (bi.hasNext()) {  
        T currB = bi.next(); //two elements pending  
        if (currB.compareTo(currA) <= 0) {  
            res.add(currB); //only one element pending  
            continue;  
        }  
        res.add(currA); //only one element pending  
        currA = currB; //that is now called currA  
        var tmp = ai; ai = bi; bi = tmp; //swap ai/bi  
    }  
    res.add(currA); //zero elements pending  
    ai.forEachRemaining(res::add);  
    return res;  
}
```





```
public List<T> merge(List<T> a, List<T> b) {  
    List<T>res= new ArrayList<>(a.size() + b.size());  
    Iterator<T> ai= a.iterator();  
    Iterator<T> bi= b.iterator();  
    T currA= ai.next(); //one element pending  
    while(bi.hasNext()) {  
        T currB= bi.next(); //two elements pending  
        if(currB.compareTo(currA) <= 0) {  
            res.add(currB); //only one element pending  
            continue;  
        }  
        res.add(currA); //only one element pending  
        currA = currB; //that is now called currA  
        var tmp= ai; ai = bi; bi = tmp; //swap ai/bi  
    }  
    res.add(currA); //zero elements pending  
    ai.forEachRemaining(res::add);  
    return res;  
}
```

Until the bi elements are smaller...



```
public List<T> merge(List<T> a, List<T> b) {  
    List<T> res= new ArrayList<>(a.size() + b.size());  
    Iterator<T> ai= a.iterator();  
    Iterator<T> bi= b.iterator();  
    T currA= ai.next(); //one element pending  
    while(bi.hasNext()) {  
        T currB= bi.next(); //two elements pending  
        if(currB.compareTo(currA) <= 0) {  
            res.add(currB); //only one element pending  
            continue;  
        }  
        res.add(currA); //only one element pending  
        currA = currB; //that is now called currA  
        var tmp= ai; ai = bi; bi = tmp; //swap ai/bi  
    }  
    res.add(currA); //zero elements pending  
    ai.forEachRemaining(res::add);  
    return res;  
}
```

Until the bi elements are smaller...

of currA, we take
those elements and...



```
public List<T> merge(List<T> a, List<T> b) {  
    List<T> res = new ArrayList<>(a.size() + b.size());  
    Iterator<T> ai = a.iterator();  
    Iterator<T> bi = b.iterator();  
    T currA = ai.next(); //one element pending  
    while (bi.hasNext()) {  
        T currB = bi.next(); //two elements pending  
        if (currB.compareTo(currA) <= 0) {  
            res.add(currB); //only one element pending  
            continue;  
        }  
        res.add(currA); //only one element pending  
        currA = currB; //that is now called currA  
        var tmp = ai; ai = bi; bi = tmp; //swap ai/bi  
    }  
    res.add(currA); //zero elements pending  
    ai.forEachRemaining(res::add);  
    return res;  
}
```





Until the bi elements are smaller...

of currA, we take those elements and...



we save them in the result.

```
public List<T> merge(List<T> a, List<T> b) {  
    List<T>res= new ArrayList<>(a.size() + b.size());  
    Iterator<T> ai= a.iterator();  
    Iterator<T> bi= b.iterator();  
    T currA= ai.next(); //one element pending  
    while(bi.hasNext()) {  
        T currB= bi.next(); //two elements pending  
        if(currB.compareTo(currA) <= 0) {  
            res.add(currB); //only one element pending  
            continue;  
        }  
        res.add(currA); //only one element pending  
        currA = currB; //that is now called currA  
        var tmp= ai; ai = bi; bi = tmp; //swap ai/bi  
    }  
    res.add(currA); //zero elements pending  
    ai.forEachRemaining(res::add);  
    return res;  
}
```



```
public List<T> merge(List<T> a, List<T> b) {  
    List<T>res= new ArrayList<>(a.size() + b.size());  
    Iterator<T> ai= a.iterator();  
    Iterator<T> bi= b.iterator();  
    T currA= ai.next(); //one element pending  
    while(bi.hasNext()) {  
        T currB= bi.next(); //two elements pending  
        if(currB.compareTo(currA) <= 0) {  
            res.add(currB); //only one element pending  
            continue;  
        }  
        res.add(currA); //only one element pending  
        currA = currB; //that is now called currA  
        var tmp= ai; ai = bi; bi = tmp; //swap ai/bi  
    }  
    res.add(currA); //zero elements pending  
    ai.forEachRemaining(res::add);  
    return res;  
}
```

If currA is the smaller



```
public List<T> merge(List<T> a, List<T> b) {  
    List<T>res= new ArrayList<>(a.size() + b.size());  
    Iterator<T> ai= a.iterator();  
    Iterator<T> bi= b.iterator();  
    T currA= ai.next(); //one element pending  
    while(bi.hasNext()) {  
        T currB= bi.next(); //two elements pending  
        if(currB.compareTo(currA) <= 0) {  
            res.add(currB); //only one element pending  
            continue;  
        }  
        res.add(currA); //only one element pending  
        currA = currB; //that is now called currA  
        var tmp= ai; ai = bi; bi = tmp; //swap ai/bi  
    }  
    res.add(currA); //zero elements pending  
    ai.forEachRemaining(res::add);  
    return res;  
}
```

If currA is the smaller

Then it is time to
add currA to the result.



```
public List<T> merge(List<T> a, List<T> b) {  
    List<T> res = new ArrayList<>(a.size() + b.size());  
    Iterator<T> ai = a.iterator();  
    Iterator<T> bi = b.iterator();  
    T currA = ai.next(); //one element pending  
    while (bi.hasNext()) {  
        T currB = bi.next(); //two elements pending  
        if (currB.compareTo(currA) <= 0) {  
            res.add(currB); //only one element pending  
            continue;  
        }  
        res.add(currA); //only one element pending  
        currA = currB; //that is now called currA  
        var tmp = ai; ai = bi; bi = tmp; //swap ai/bi  
    }  
    res.add(currA); //zero elements pending  
    ai.forEachRemaining(res::add);  
    return res;  
}
```



```
public List<T> merge(List<T> a, List<T> b) {  
    List<T>res= new ArrayList<>(a.size() + b.size());  
    Iterator<T> ai= a.iterator();  
    Iterator<T> bi= b.iterator();  
    T currA= ai.next(); //one element pending  
    while(bi.hasNext()) {  
        T currB= bi.next(); //two elements pending  
        if(currB.compareTo(currA) <= 0) {  
            res.add(currB); //only one element pending  
            continue;  
        }  
        res.add(currA); //only one element pending  
        currA = currB; //that is now called currA  
        var tmp= ai; ai = bi; bi = tmp; //swap ai/bi  
    }  
    res.add(currA); //zero elements pending  
    ai.forEachRemaining(res::add);  
    return res;  
}
```

Moreover, ...



```
public List<T> merge(List<T> a, List<T> b) {  
    List<T>res= new ArrayList<>(a.size() + b.size());  
    Iterator<T> ai= a.iterator();  
    Iterator<T> bi= b.iterator();  
    T currA= ai.next(); //one element pending  
    while(bi.hasNext()) {  
        T currB= bi.next(); //two elements pending  
        if(currB.compareTo(currA) <= 0) {  
            res.add(currB); //only one element pending  
            continue;  
        }  
        res.add(currA); //only one element pending  
        currA = currB; //that is now called currA  
        var tmp= ai; ai = bi; bi = tmp; //swap ai/bi  
    }  
    res.add(currA); //zero elements pending  
    ai.forEachRemaining(res::add);  
    return res;  
}
```

Moreover, ...

now we need to see
how many elements of ai
should be inserted there



```
public List<T> merge(List<T> a, List<T> b) {  
    List<T> res = new ArrayList<>(a.size() + b.size());  
    Iterator<T> ai = a.iterator();  
    Iterator<T> bi = b.iterator();  
    T currA = ai.next(); //one element pending  
    while (bi.hasNext()) {  
        T currB = bi.next(); //two elements pending  
        if (currB.compareTo(currA) <= 0) {  
            res.add(currB); //only one element pending  
            continue;  
        }  
        res.add(currA); //only one element pending  
        currA = currB; //that is now called currA  
        var tmp = ai; ai = bi; bi = tmp; //swap ai/bi  
    }  
    res.add(currA); //zero elements pending  
    ai.forEachRemaining(res::add);  
    return res;  
}
```



Moreover, ...

now we need to see
how many elements of ai
should be inserted there

So ai and bi roles must swap!

```
public List<T> merge(List<T> a, List<T> b) {  
    List<T> res = new ArrayList<>(a.size() + b.size());  
    Iterator<T> ai = a.iterator();  
    Iterator<T> bi = b.iterator();  
    T currA = ai.next(); //one element pending  
    while (bi.hasNext()) {  
        T currB = bi.next(); //two elements pending  
        if (currB.compareTo(currA) <= 0) {  
            res.add(currB); //only one element pending  
            continue;  
        }  
        res.add(currA); //only one element pending  
        currA = currB; //that is now called currA  
        var tmp = ai; ai = bi; bi = tmp; //swap ai/bi  
    }  
    res.add(currA); //zero elements pending  
    ai.forEachRemaining(res::add);  
    return res;
```



Moreover, ...

now we need to see
how many elements of ai
should be inserted there

So ai and bi roles must swap!

currB becomes the new reference point,
and we keep taking elements until we beat currB!

```
public List<T> merge(List<T> a, List<T> b) {  
    List<T> res = new ArrayList<>(a.size() + b.size());  
    Iterator<T> ai = a.iterator();  
    Iterator<T> bi = b.iterator();  
    T currA = ai.next(); //one element pending  
    while (bi.hasNext()) {  
        T currB = bi.next(); //two elements pending  
        if (currB.compareTo(currA) <= 0) {  
            res.add(currB); //only one element pending  
            continue;  
        }  
        res.add(currA); //only one element pending  
        currA = currB; //that is now called currA  
        var tmp = ai; ai = bi; bi = tmp; //swap ai/bi  
    }  
    res.add(currA); //zero elements pending  
    ai.forEachRemaining(res::add);  
    return res;
```



```
public List<T> merge(List<T> a, List<T> b) {  
    List<T>res= new ArrayList<>(a.size() + b.size());  
    Iterator<T> ai= a.iterator();  
    Iterator<T> bi= b.iterator();  
    T currA= ai.next(); //one element pending  
    while(bi.hasNext()) {  
        T currB= bi.next(); //two elements pending  
        if(currB.compareTo(currA) <= 0) {  
            res.add(currB); //only one element pending  
            continue;  
        }  
        res.add(currA); //only one element pending  
        currA = currB; //that is now called currA  
        var tmp= ai; ai = bi; bi = tmp; //swap ai/bi  
    }  
    res.add(currA); //zero elements pending  
    ai.forEachRemaining(res::add);  
    return res;  
}
```

We stop the while when ...



```
public List<T> merge(List<T> a, List<T> b) {  
    List<T> res = new ArrayList<>(a.size() + b.size());  
    Iterator<T> ai = a.iterator();  
    Iterator<T> bi = b.iterator();  
    T currA = ai.next(); //one element pending  
    while (bi.hasNext()) {  
        T currB = bi.next(); //two elements pending  
        if (currB.compareTo(currA) <= 0) {  
            res.add(currB); //only one element pending  
            continue;  
        }  
        res.add(currA); //only one element pending  
        currA = currB; //that is now called currA  
        var tmp = ai; ai = bi; bi = tmp; //swap ai/bi  
    }  
    res.add(currA); //zero elements pending  
    ai.forEachRemaining(res::add);  
    return res;  
}
```



We stop the while when ...

the iterator we are exploring is finished

```
public List<T> merge(List<T> a, List<T> b) {  
    List<T> res = new ArrayList<>(a.size() + b.size());  
    Iterator<T> ai = a.iterator();  
    Iterator<T> bi = b.iterator();  
    T currA = ai.next(); //one element pending  
    while (bi.hasNext()) {  
        T currB = bi.next(); //two elements pending  
        if (currB.compareTo(currA) <= 0) {  
            res.add(currB); //only one element pending  
            continue;  
        }  
        res.add(currA); //only one element pending  
        currA = currB; //that is now called currA  
        var tmp = ai; ai = bi; bi = tmp; //swap ai/bi  
    }  
    res.add(currA); //zero elements pending  
    ai.forEachRemaining(res::add);  
    return res;  
}
```



We stop the while when ...

the iterator we are exploring is finished

```
public List<T> merge(List<T> a, List<T> b) {  
    List<T> res = new ArrayList<>(a.size() + b.size());  
    Iterator<T> ai = a.iterator();  
    Iterator<T> bi = b.iterator();  
    T currA = ai.next(); //one element pending  
    while (bi.hasNext()) {  
        T currB = bi.next(); //two elements pending  
        if (currB.compareTo(currA) <= 0) {  
            res.add(currB); //only one element pending  
            continue;  
        }  
        res.add(currA); //only one element pending  
        currA = currB; //that is now called currA  
        var tmp = ai; ai = bi; bi = tmp; //swap ai/bi  
    }  
    res.add(currA); //zero elements pending  
    ai.forEachRemaining(res::add);  
    return res;  
}
```





```
public List<T> merge(List<T> a, List<T> b) {  
    List<T>res= new ArrayList<>(a.size() + b.size());  
    Iterator<T> ai= a.iterator();  
    Iterator<T> bi= b.iterator();  
    T currA= ai.next(); //one element pending  
    while(bi.hasNext()) {  
        T currB= bi.next(); //two elements pending  
        if(currB.compareTo(currA) <= 0) {  
            res.add(currB); //only one element pending  
            continue;  
        }  
        res.add(currA); //only one element pending  
        currA = currB; //that is now called currA  
        var tmp= ai; ai = bi; bi = tmp; //swap ai/bi  
    }  
    res.add(currA); //zero elements pending  
    ai.forEachRemaining(res::add);  
    return res;  
}
```

We then add the pending element



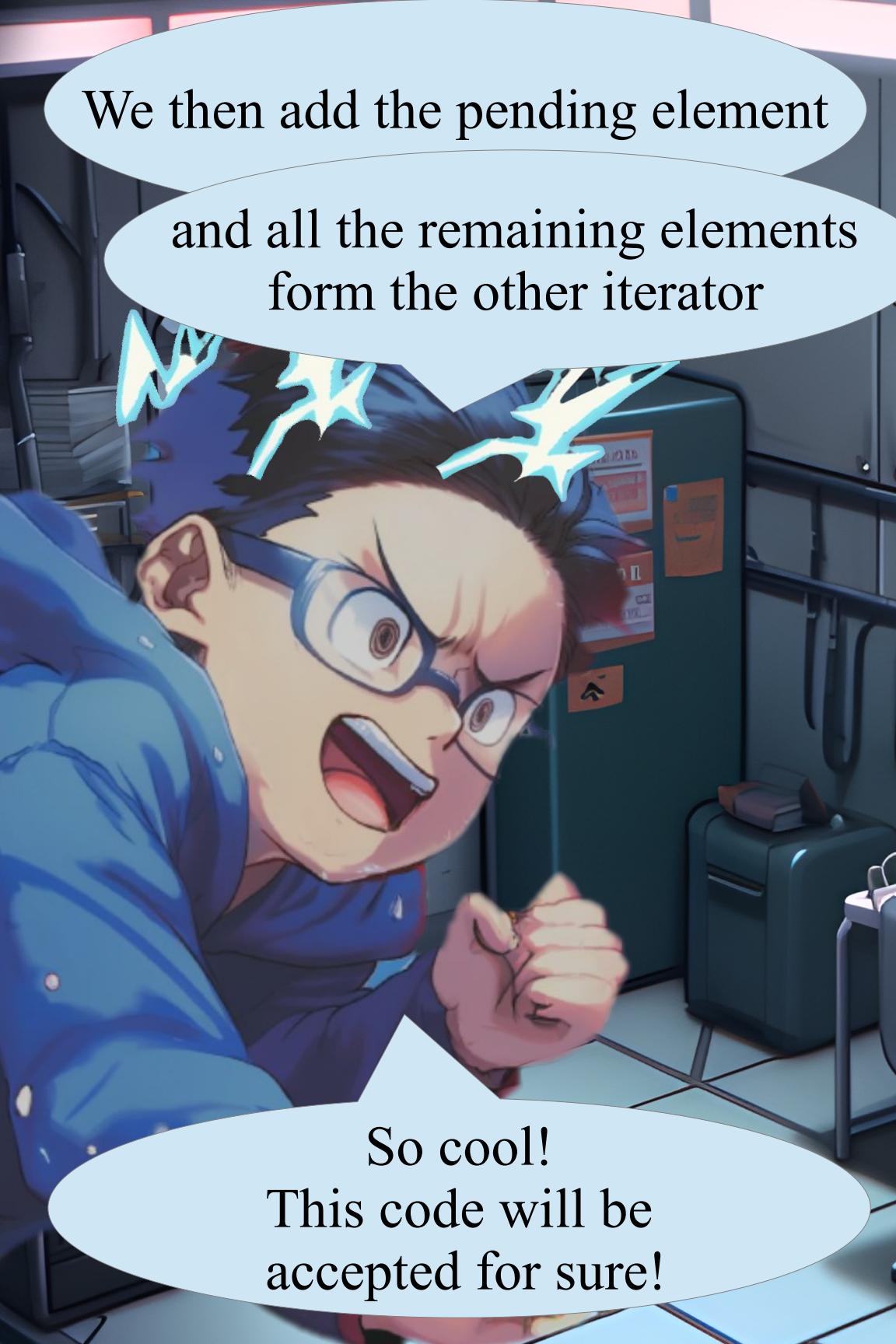
```
public List<T> merge(List<T> a, List<T> b) {  
    List<T>res= new ArrayList<>(a.size() + b.size());  
    Iterator<T> ai= a.iterator();  
    Iterator<T> bi= b.iterator();  
    T currA= ai.next(); //one element pending  
    while(bi.hasNext()) {  
        T currB= bi.next(); //two elements pending  
        if(currB.compareTo(currA) <= 0) {  
            res.add(currB); //only one element pending  
            continue;  
        }  
        res.add(currA); //only one element pending  
        currA = currB; //that is now called currA  
        var tmp= ai; ai = bi; bi = tmp; //swap ai/bi  
    }  
    res.add(currA); //zero elements pending  
    ai.forEachRemaining(res::add);  
    return res;  
}
```

We then add the pending element

and all the remaining elements
form the other iterator



```
public List<T> merge(List<T> a, List<T> b) {  
    List<T> res = new ArrayList<>(a.size() + b.size());  
    Iterator<T> ai = a.iterator();  
    Iterator<T> bi = b.iterator();  
    T currA = ai.next(); //one element pending  
    while (bi.hasNext()) {  
        T currB = bi.next(); //two elements pending  
        if (currB.compareTo(currA) <= 0) {  
            res.add(currB); //only one element pending  
            continue;  
        }  
        res.add(currA); //only one element pending  
        currA = currB; //that is now called currA  
        var tmp = ai; ai = bi; bi = tmp; //swap ai/bi  
    }  
    res.add(currA); //zero elements pending  
    ai.forEachRemaining(res::add);  
    return res;  
}
```



We then add the pending element

and all the remaining elements
form the other iterator

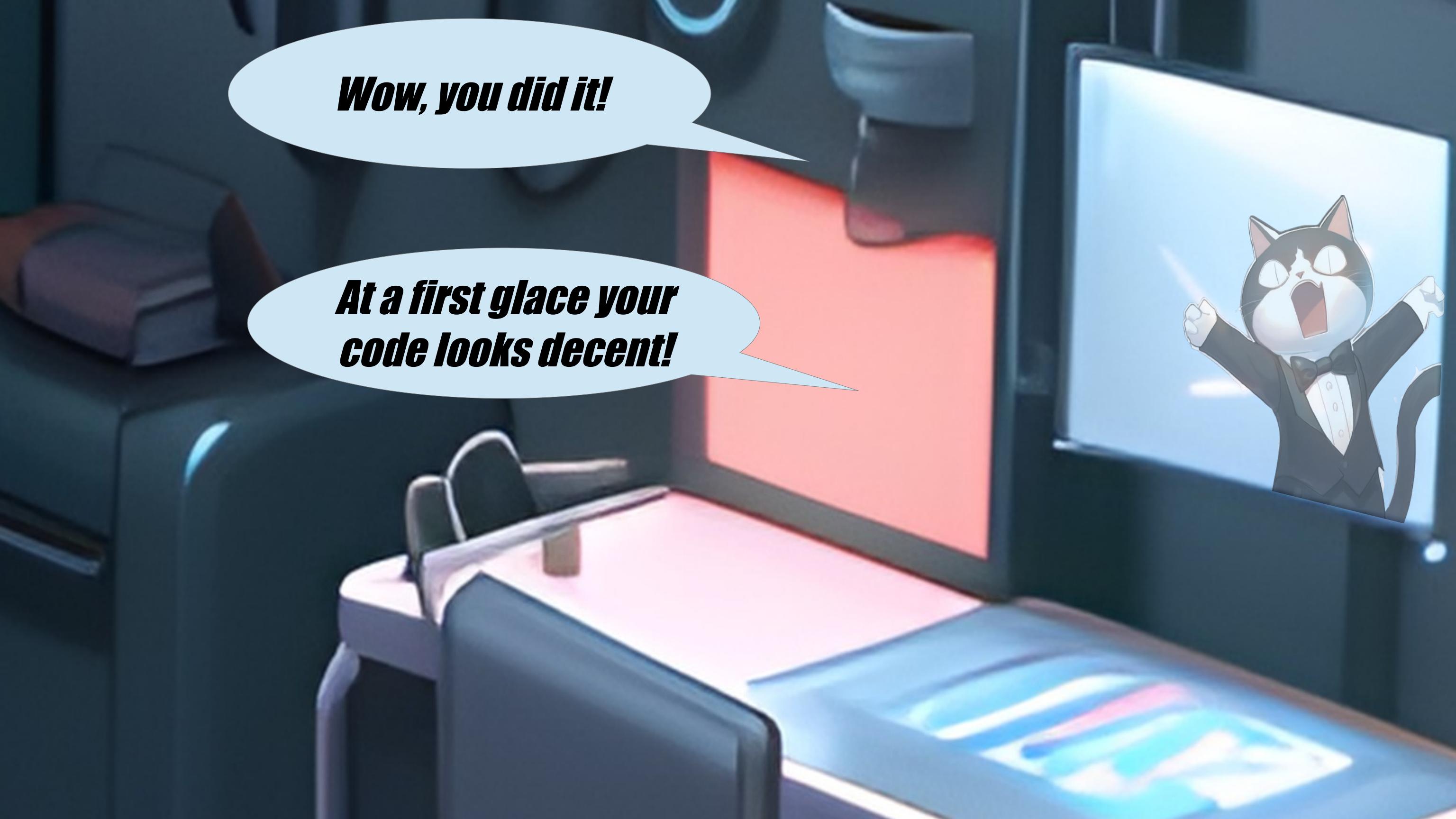
So cool!
This code will be
accepted for sure!

```
public List<T> merge(List<T> a, List<T> b) {  
    List<T>res= new ArrayList<>(a.size() + b.size());  
    Iterator<T> ai= a.iterator();  
    Iterator<T> bi= b.iterator();  
    T currA= ai.next(); //one element pending  
    while(bi.hasNext()) {  
        T currB= bi.next(); //two elements pending  
        if(currB.compareTo(currA) <= 0) {  
            res.add(currB); //only one element pending  
            continue;  
        }  
        res.add(currA); //only one element pending  
        currA = currB; //that is now called currA  
        var tmp= ai; ai = bi; bi = tmp; //swap ai/bi  
    }  
    res.add(currA); //zero elements pending  
    ai.forEachRemaining(res::add);  
    return res;  
}
```



Wow, you did it!



A stack of books with a cat on top, looking at a computer screen.

Wow, you did it!

*At a first glace your
code looks decent!*







However...



Pause the video. What is CAT going to say?
What is wrong in the submitted code?

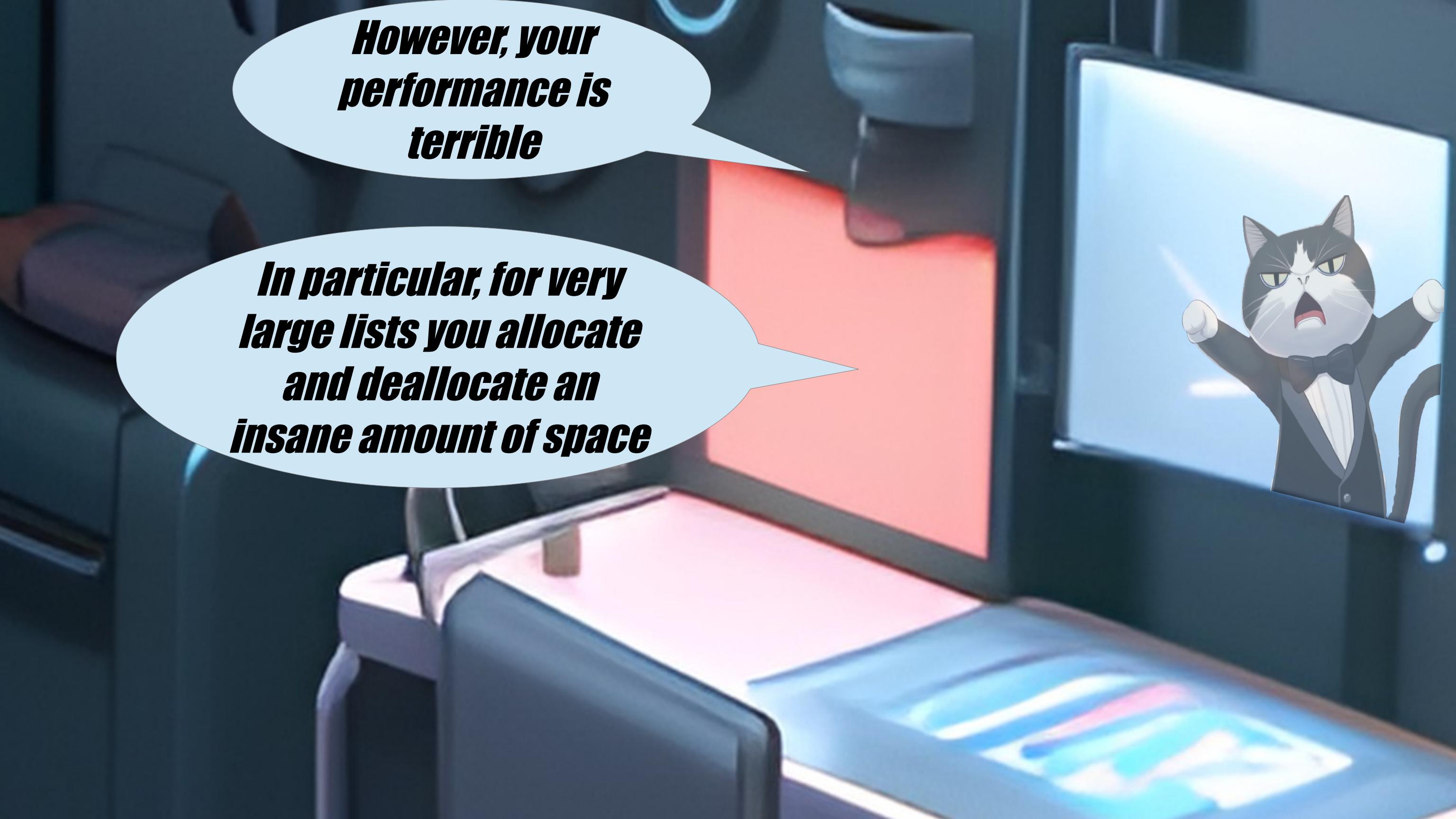
This message will disappear shortly,
showing the submitted code.
You can pause the video after that.

```
public List<T> sort(List<T> list) {
    int size= list.size();
    if (size < 2) { return list; }
    int half= size / 2;
    List<T> left= sort(list.subList(0, half));
    List<T> right= sort(list.subList(half, size));
    return merge(left, right);
}
public List<T> merge(List<T> a, List<T> b) {
    List<T>res= new ArrayList<>(a.size() + b.size());
    Iterator<T> ai= a.iterator();
    Iterator<T> bi= b.iterator();
    T currA= ai.next(); //one element pending
    while(bi.hasNext()) {
        T currB= bi.next(); //two elements pending
        if(currB.compareTo(currA) <= 0) {
            res.add(currB); //only one element pending
            continue;
        }
        res.add(currA); //only one element pending
        currA = currB; //that is now called currA
        var tmp= ai; ai = bi; bi = tmp; //swap ai/bi
    }
    res.add(currA); //zero elements pending
    ai.forEachRemaining(res::add);
    return res;
}
```





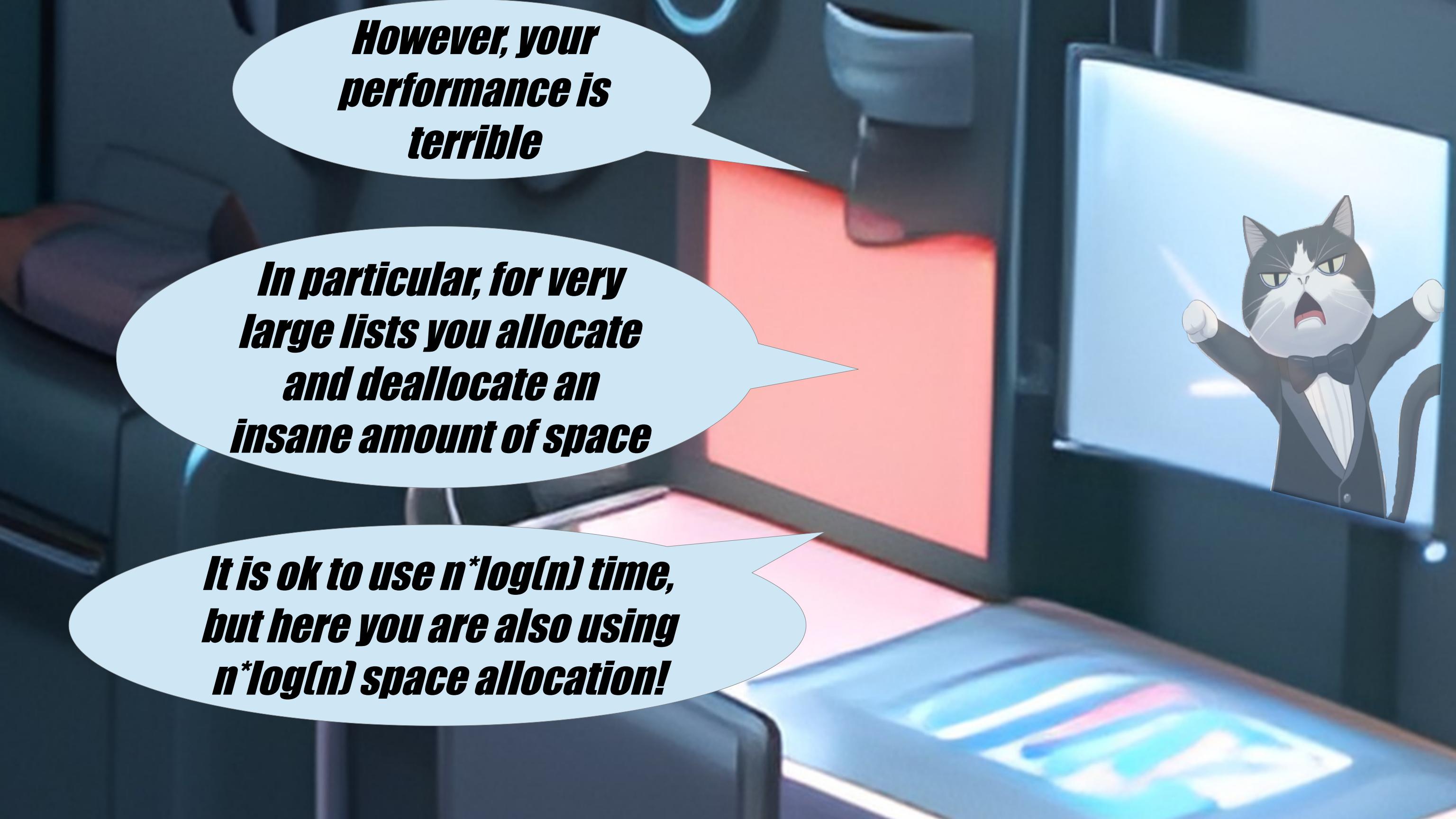
*However, your
performance is
terrible*



*However, your
performance is
terrible*

*In particular, for very
large lists you allocate
and deallocate an
insane amount of space*

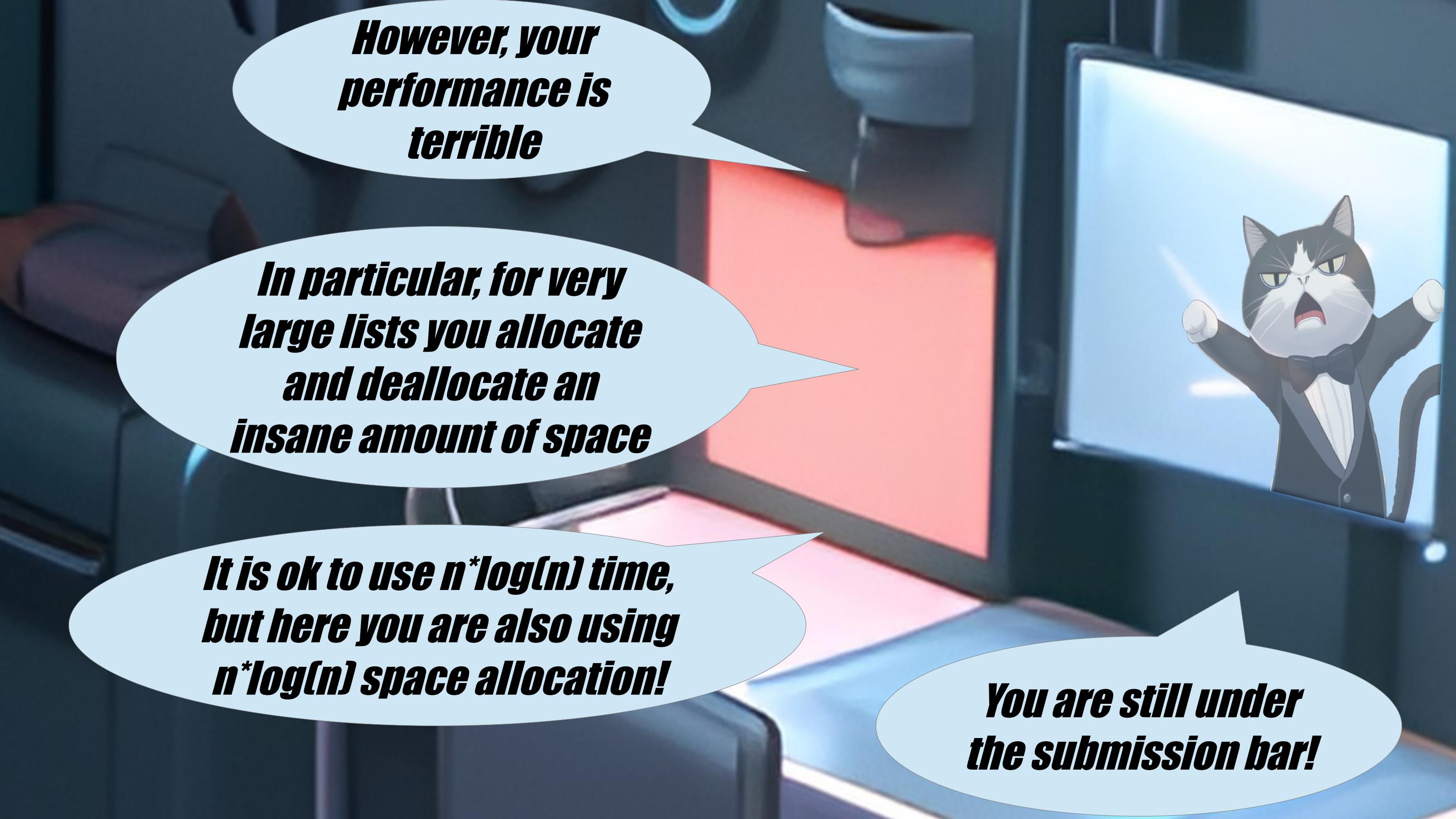




*However, your
performance is
terrible*

*In particular, for very
large lists you allocate
and deallocate an
insane amount of space*

*It is ok to use $n \log(n)$ time,
but here you are also using
 $n \log(n)$ space allocation!*

A stack of books with a cat in a tuxedo pointing at it.

*However, your
performance is
terrible*

*In particular, for very
large lists you allocate
and deallocate an
insane amount of space*

*It is ok to use $n \log(n)$ time,
but here you are also using
 $n \log(n)$ space allocation!*

*You are still under
the submission bar!*





No!
Another 10% is gone

A dynamic illustration of a man in a blue suit and glasses running towards the viewer. He has a determined expression, with his right fist clenched and forward. The background consists of blurred blue and white streaks, suggesting speed. Two speech bubbles are positioned to the right of the character.

No!
Another 10% is gone

Keep calm Dany,
keep calm





My power is still active,
I can go on!

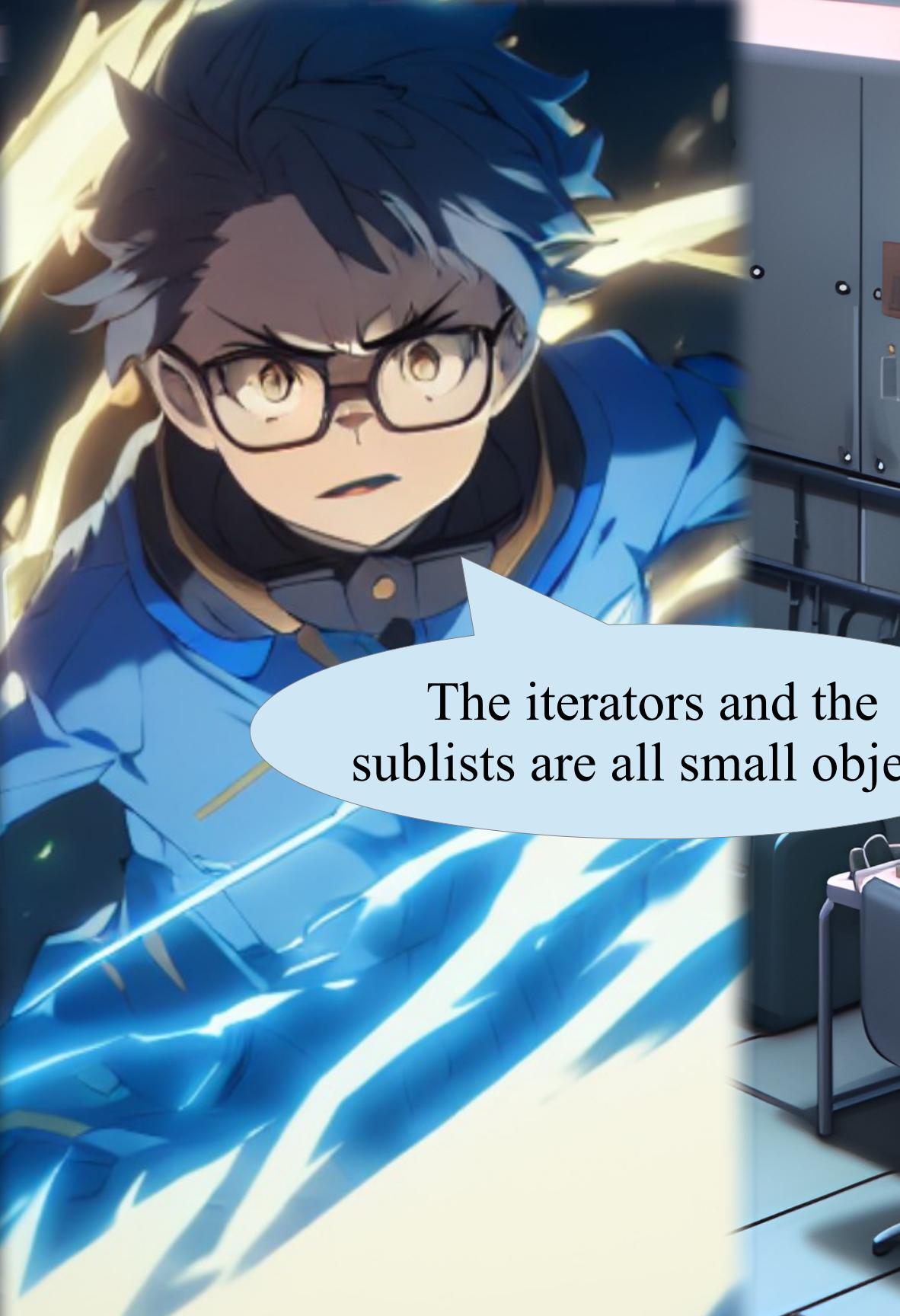


My power is still active,
I can go on!

What is wrong in my code?
Where do I allocate too much?



```
public List<T> merge(List<T> a, List<T> b) {  
    List<T> res= new ArrayList<>(a.size() + b.size());  
    Iterator<T> ai= a.iterator();  
    Iterator<T> bi= b.iterator();  
    T currA= ai.next(); //one element pending  
    while(bi.hasNext()) {  
        T currB= bi.next(); //two elements pending  
        if(currB.compareTo(currA) <= 0) {  
            res.add(currB); //only one element pending  
            continue;  
        }  
        res.add(currA); //only one element pending  
        currA = currB; //that is now called currA  
        var tmp= ai; ai = bi; bi = tmp; //swap ai/bi  
    }  
    res.add(currA); //zero elements pending  
    ai.forEachRemaining(res::add);  
    return res;  
}
```



The iterators and the
sublists are all small objects

```
public List<T> merge(List<T> a, List<T> b) {  
    List<T> res = new ArrayList<>(a.size() + b.size());  
    Iterator<T> ai = a.iterator();  
    Iterator<T> bi = b.iterator();  
    T currA = ai.next(); //one element pending  
    while (bi.hasNext()) {  
        T currB = bi.next(); //two elements pending  
        if (currB.compareTo(currA) <= 0) {  
            res.add(currB); //only one element pending  
            continue;  
        }  
        res.add(currA); //only one element pending  
        currA = currB; //that is now called currA  
        var tmp = ai; ai = bi; bi = tmp; //swap ai/bi  
        res.add(currA); //zero elements pending  
        ai.forEachRemaining(res::add);  
    }  
    return res;  
}
```



The iterators and the
sublists are all small objects

They are well optimized in Java and
are not the reason we allocate $n * \log(n)$ space

```
public List<T> merge(List<T> a, List<T> b) {  
    List<T> res = new ArrayList<>(a.size() + b.size());  
    Iterator<T> ai = a.iterator();  
    Iterator<T> bi = b.iterator();  
    T currA = ai.next(); //one element pending  
    while (bi.hasNext()) {  
        T currB = bi.next(); //two elements pending  
        if (currB.compareTo(currA) <= 0) {  
            res.add(currB); //only one element pending  
            continue;  
        }  
        res.add(currA); //only one element pending  
        currA = currB; //that is now called currA  
        var tmp = ai; ai = bi; bi = tmp; //swap ai/bi  
        res.add(currA); //zero elements pending  
        ai.forEachRemaining(res::add);  
    }  
    return res;  
}
```



```
public List<T> merge(List<T> a, List<T> b) {  
    List<T> res= new ArrayList<>(a.size() + b.size());  
    Iterator<T> ai= a.iterator();  
    Iterator<T> bi= b.iterator();  
    T currA= ai.next(); //one element pending  
    while(bi.hasNext()) {  
        T currB= bi.next(); //two elements pending  
        if(currB.compareTo(currA) <= 0) {  
            res.add(currB); //only one element pending  
            continue;  
        }  
        res.add(currA); //only one element pending  
        currA = currB; //that is now called currA  
        var tmp= ai; ai = bi; bi = tmp; //swap ai/bi  
    }  
    res.add(currA); //zero elements pending  
    ai.forEachRemaining(res::add);  
    return res;  
}
```



```
public List<T> merge(List<T> a, List<T> b) {  
    List<T> res = new ArrayList<>(a.size() + b.size());  
    Iterator<T> ai = a.iterator();  
    Iterator<T> bi = b.iterator();  
    T currA = ai.next(); //one element pending  
    while (bi.hasNext()) {  
        T currB = bi.next(); //two elements pending  
        if (currB.compareTo(currA) <= 0) {  
            res.add(currB); //only one element pending  
            continue;  
        }  
        res.add(currA); //only one element pending  
        currA = currB; //that is now called currA  
        var tmp = ai; ai = bi; bi = tmp; //swap ai/bi  
    }  
    res.add(currA); //zero elements pending  
    ai.forEachRemaining(res::add);  
    return res;
```

It must be those new ArrayLists that
I'm creating over and over again



```
public List<T> merge(List<T> a, List<T> b) {  
    List<T> res = new ArrayList<>(a.size() + b.size());  
    Iterator<T> ai = a.iterator();  
    Iterator<T> bi = b.iterator();  
    T currA = ai.next(); //one element pending  
    while (bi.hasNext()) {  
        T currB = bi.next(); //two elements pending  
        if (currB.compareTo(currA) <= 0) {  
            res.add(currB); //only one element pending  
            continue;  
        }  
        res.add(currA); //only one element pending  
        currA = currB; //that is now called currA  
        var tmp = ai; ai = bi; bi = tmp; //swap ai/bi  
    }  
    res.add(currA); //zero elements pending  
    ai.forEachRemaining(res::add);  
    return res;
```

It must be those new ArrayLists that I'm creating over and over again

down the recursive rabbit hole of unbounded depth





Let see how far this sword of residual power brings me





I must cut through the problem
in a single strike!

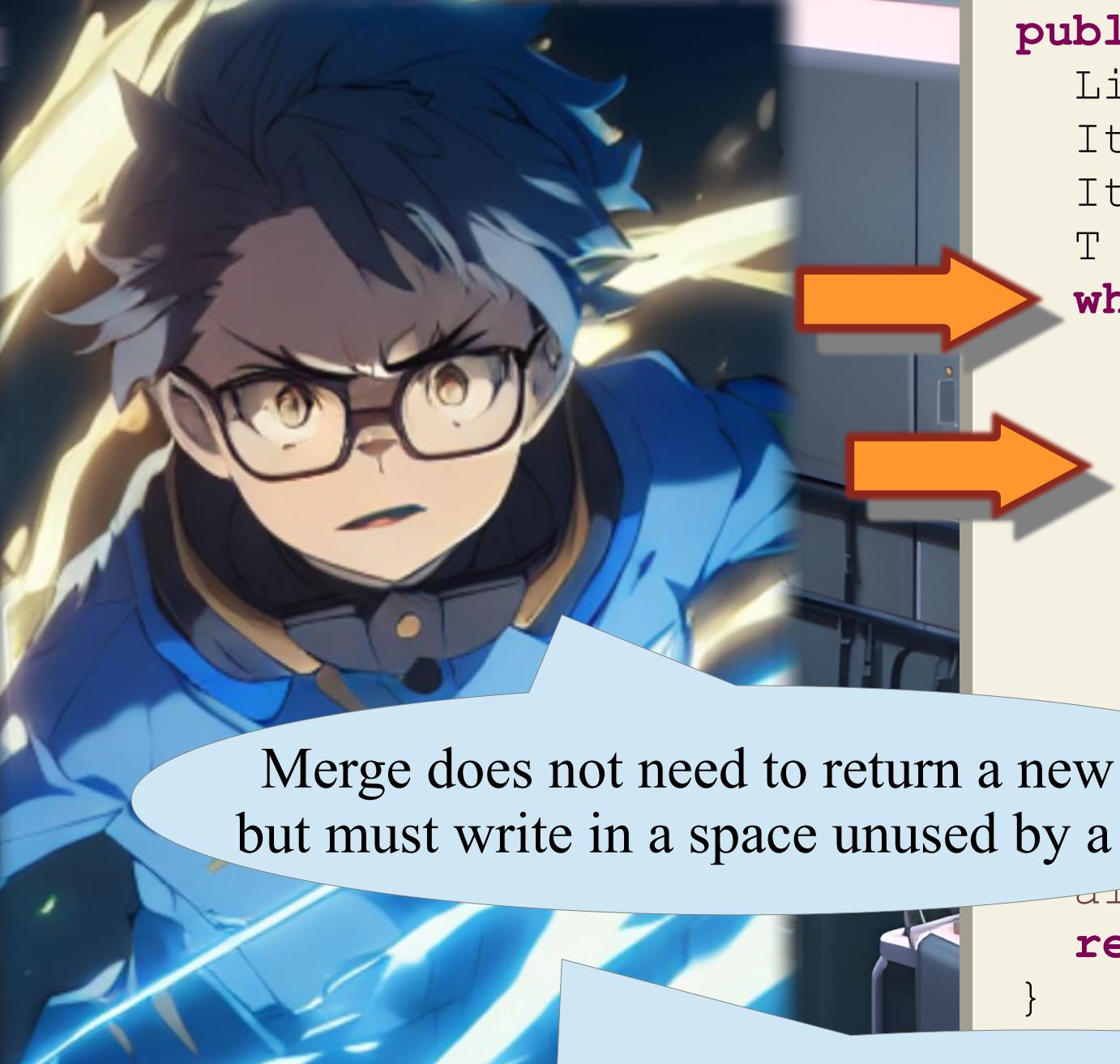


```
public List<T> merge(List<T> a, List<T> b) {  
    List<T> res= new ArrayList<>(a.size() + b.size());  
    Iterator<T> ai= a.iterator();  
    Iterator<T> bi= b.iterator();  
    T currA= ai.next(); //one element pending  
    while(bi.hasNext()) {  
        T currB= bi.next(); //two elements pending  
        if(currB.compareTo(currA) <= 0) {  
            res.add(currB); //only one element pending  
            continue;  
        }  
        res.add(currA); //only one element pending  
        currA = currB; //that is now called currA  
        var tmp= ai; ai = bi; bi = tmp; //swap ai/bi  
    }  
    res.add(currA); //zero elements pending  
    ai.forEachRemaining(res::add);  
    return res;  
}
```



Merge does not need to return a new list,
but must write in a space unused by a and b

```
public List<T> merge(List<T> a, List<T> b) {  
    List<T> res = new ArrayList<>(a.size() + b.size());  
    Iterator<T> ai = a.iterator();  
    Iterator<T> bi = b.iterator();  
    T currA = ai.next(); //one element pending  
    while (bi.hasNext()) {  
        T currB = bi.next(); //two elements pending  
        if (currB.compareTo(currA) <= 0) {  
            res.add(currB); //only one element pending  
            continue;  
        }  
        res.add(currA); //only one element pending  
        currA = currB; //that is now called currA  
        tmp = ai; ai = bi; bi = tmp; //swap ai/bi  
        currA = ai.next(); //zero elements pending  
        ai.forEachRemaining(res::add);  
    }  
    return res;  
}
```



Merge does not need to return a new list,
but must write in a space unused by a and b

a and b come from sublists, so they are connected to the main input.
We can not write back onto a and b, nor on the main input.
We need the information to stay stable while we read it!

```
public List<T> merge(List<T> a, List<T> b) {  
    List<T> res = new ArrayList<>(a.size() + b.size());  
    Iterator<T> ai = a.iterator();  
    Iterator<T> bi = b.iterator();  
    T currA = ai.next(); //one element pending  
    while (bi.hasNext()) {  
        T currB = bi.next(); //two elements pending  
        if (currB.compareTo(currA) <= 0) {  
            res.add(currB); //only one element pending  
            continue;  
        }  
        res.add(currA); //only one element pending  
        currA = currB; //that is now called currA  
        tmp = ai; ai = bi; bi = tmp; //swap ai/bi  
        ai.forEachRemaining(res::add);  
    }  
    return res;  
}
```





There is no fully in-place algorithm!





So, I will allocate a space for the result a single time at the start!



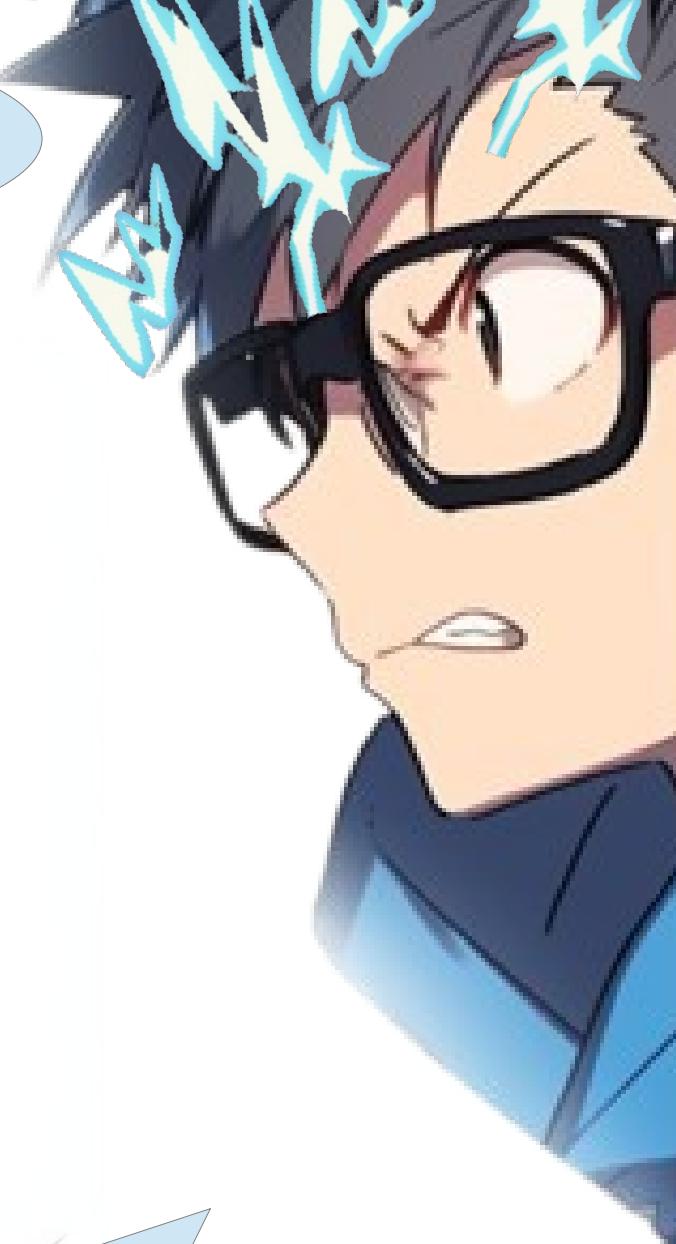


With two slots of space,
I can read from one and write on the other

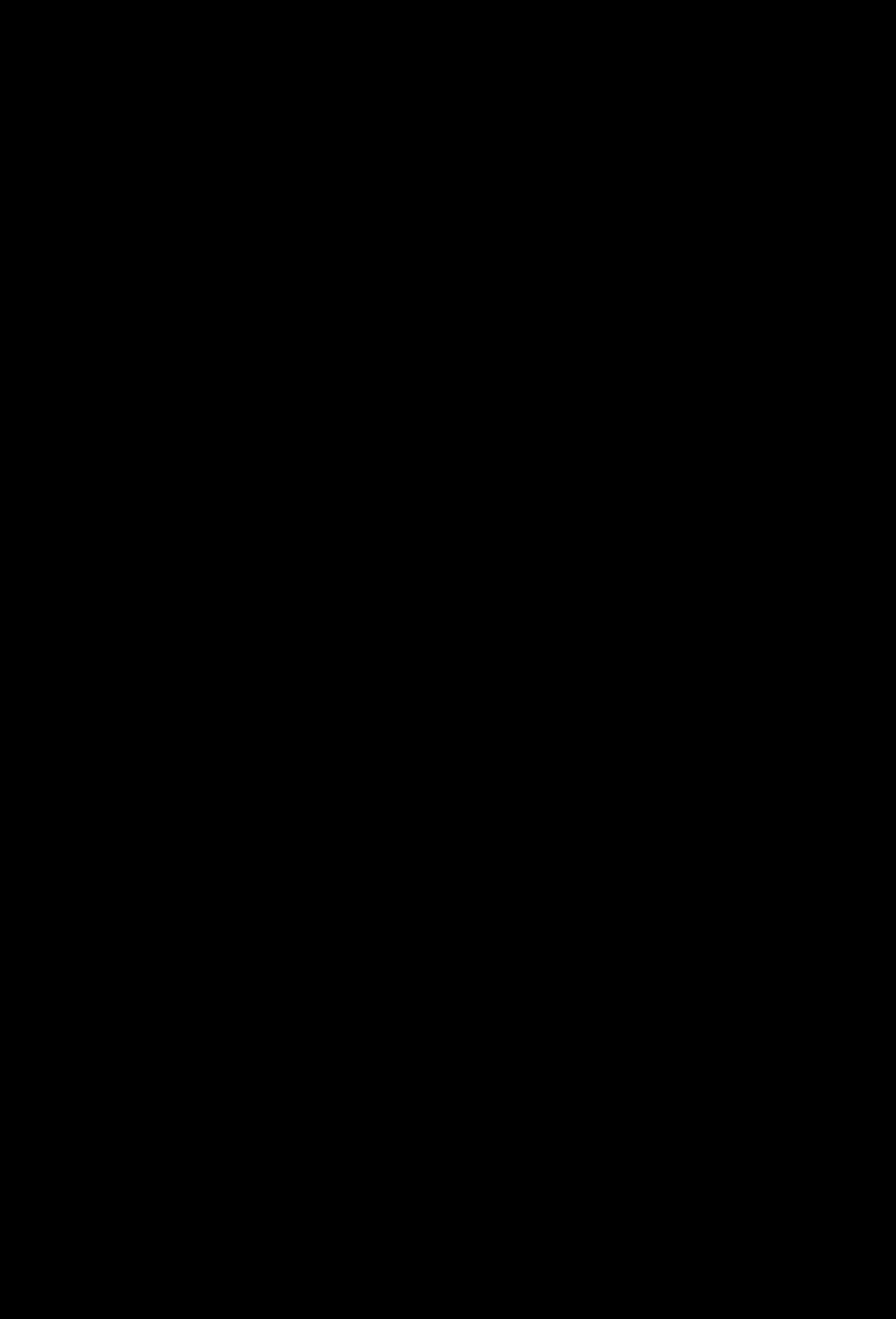




With two slots of space,
I can read from one and write on the other



I will swap them at every recursive level,
so that the fresh result is written
into the data of the former step.





```
public List<T> sort(List<T> list) {  
  
    List<T> result= new ArrayList<>(list);  
    sortHelper(list, result);  
    return result;  
}  
private void sortHelper(List<T> tmp, List<T> result) {  
    ...//recursive  
}  
private void merge(Iterator<T> ai, Iterator<T> bi, ??? res) {  
    ...  
}
```



```
public List<T> sort(List<T> list) {  
  
    List<T> result= new ArrayList<>(list);  
    sortHelper(list, result);  
    return result;  
}  
private void sortHelper(List<T> tmp, List<T> result) {  
    ...//recursive  
}  
private void merge(Iterator<T> ai, Iterator<T> bi, ??? res) {  
    ...  
}
```

This could be the new code structure



```
public List<T> sort(List<T> list) {  
  
    List<T> result= new ArrayList<>(list);  
    sortHelper(list, result);  
    return result;  
}  
  
private void sortHelper(List<T> tmp, List<T> result) {  
    ...//recursive  
}  
  
private void merge(Iterator<T> ai, Iterator<T> bi, ??? res) {  
    ...  
}
```

This could be the new code structure

sort allocates all the needed
space and calls sortHelper



```
public List<T> sort(List<T> list) {  
  
    List<T> result= new ArrayList<>(list);  
    sortHelper(list, result);  
    return result;  
}  
  
private void sortHelper(List<T> tmp, List<T> result) {  
    ...//recursive  
}  
  
private void merge(Iterator<T> ai, Iterator<T> bi, ??? res) {  
    ...  
}
```

This could be the new code structure

sort allocates all the needed
space and calls sortHelper

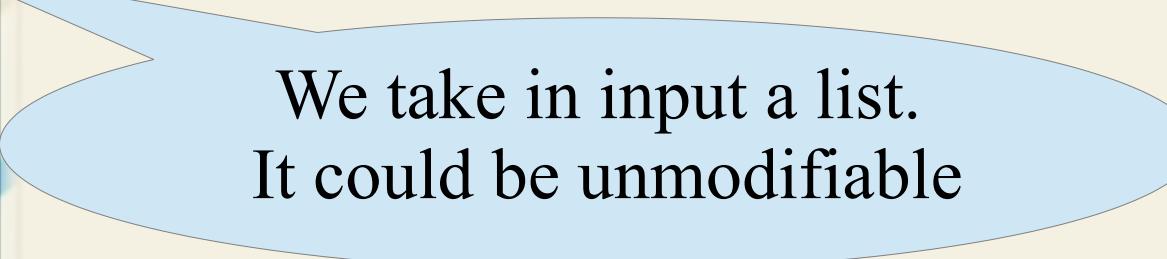
sortHelper calls sortHelper and merge



```
public List<T> sort(List<T> list) {  
    List<T> tmp = new ArrayList<>(list);  
    List<T> result= new ArrayList<>(list);  
    sortHelper(tmp, result);  
    return Collections.unmodifiableList(result);  
}  
private void sortHelper(List<T> tmp, List<T> result) {  
    ...  
}  
private void merge(Iterator<T> ai, Iterator<T> bi, ??? res) {  
    ...  
}
```



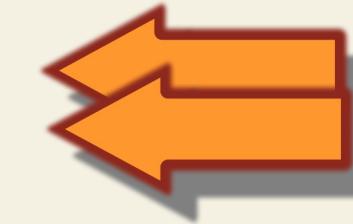
```
public List<T> sort(List<T> list) {  
    List<T> tmp = new ArrayList<>(list);  
    List<T> result= new ArrayList<>(list);  
    sortHelper(tmp, result);  
    return Collections.unmodifiableList(result);  
}  
private void sortHelper(List<T> tmp, List<T> result) {  
    ...  
}  
private void merge(Iterator<T> ai, Iterator<T> bi, ??? res) {  
    ...  
}
```



We take in input a list.
It could be unmodifiable



```
public List<T> sort(List<T> list) {  
    List<T> tmp = new ArrayList<>(list);  
    List<T> result= new ArrayList<>(list);  
    sortHelper(tmp, result);  
    return Collections.unmodifiableList(result);  
}  
  
private void sortHelper(List<T> tmp, List<T> result) {  
    ...  
}  
  
private void merge(Iterator<T> ai, Iterator<T> bi, ??? res) {  
    ...  
}
```



We take in input a list.
It could be unmodifiable

So we make two copies of the input,
both modifiable ArrayLists.



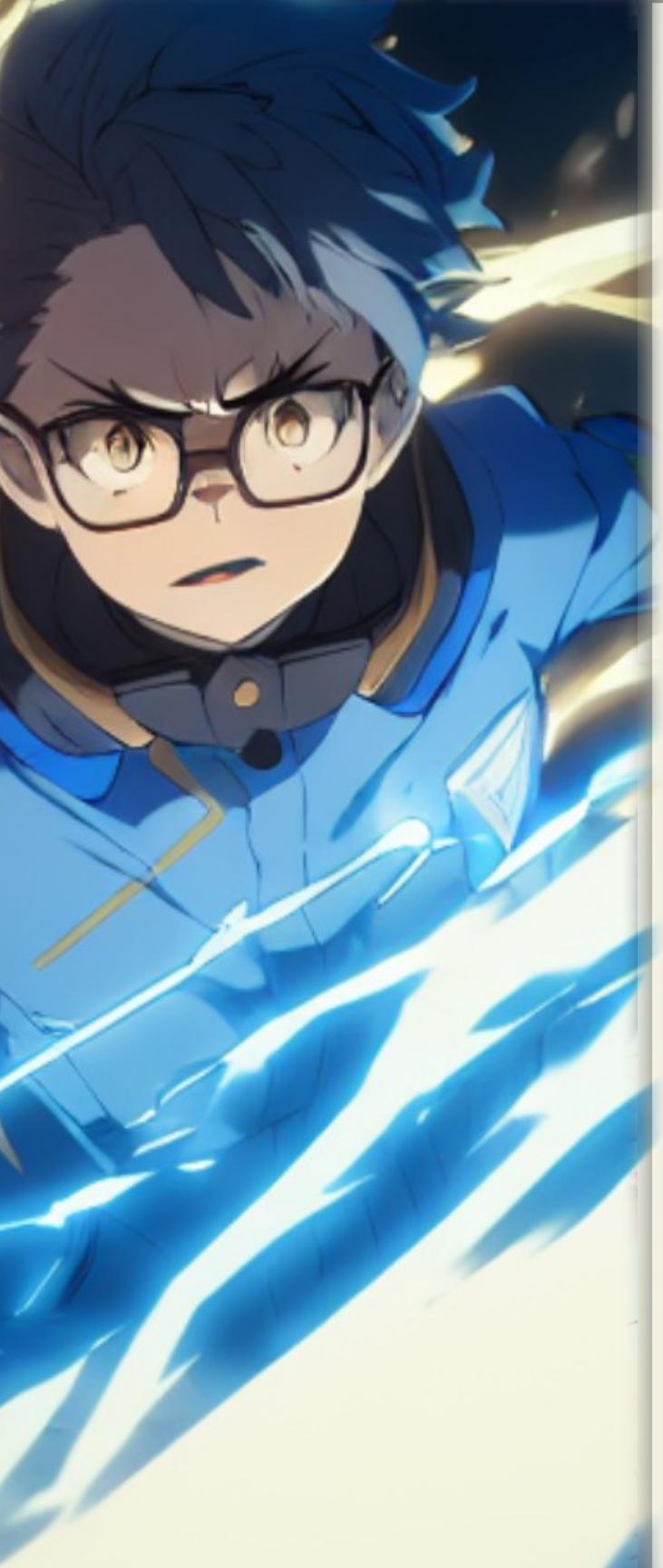
```
public List<T> sort(List<T> list) {  
    List<T> tmp = new ArrayList<>(list);  
    List<T> result= new ArrayList<>(list);  
    sortHelper(tmp, result);  
    return Collections.unmodifiableList(result);  
}  
  
private void sortHelper(List<T> tmp, List<T> result) {  
    ...  
}  
  
private void merge(Iterator<T> ai, Iterator<T> bi, ??? res) {  
    ...  
}
```



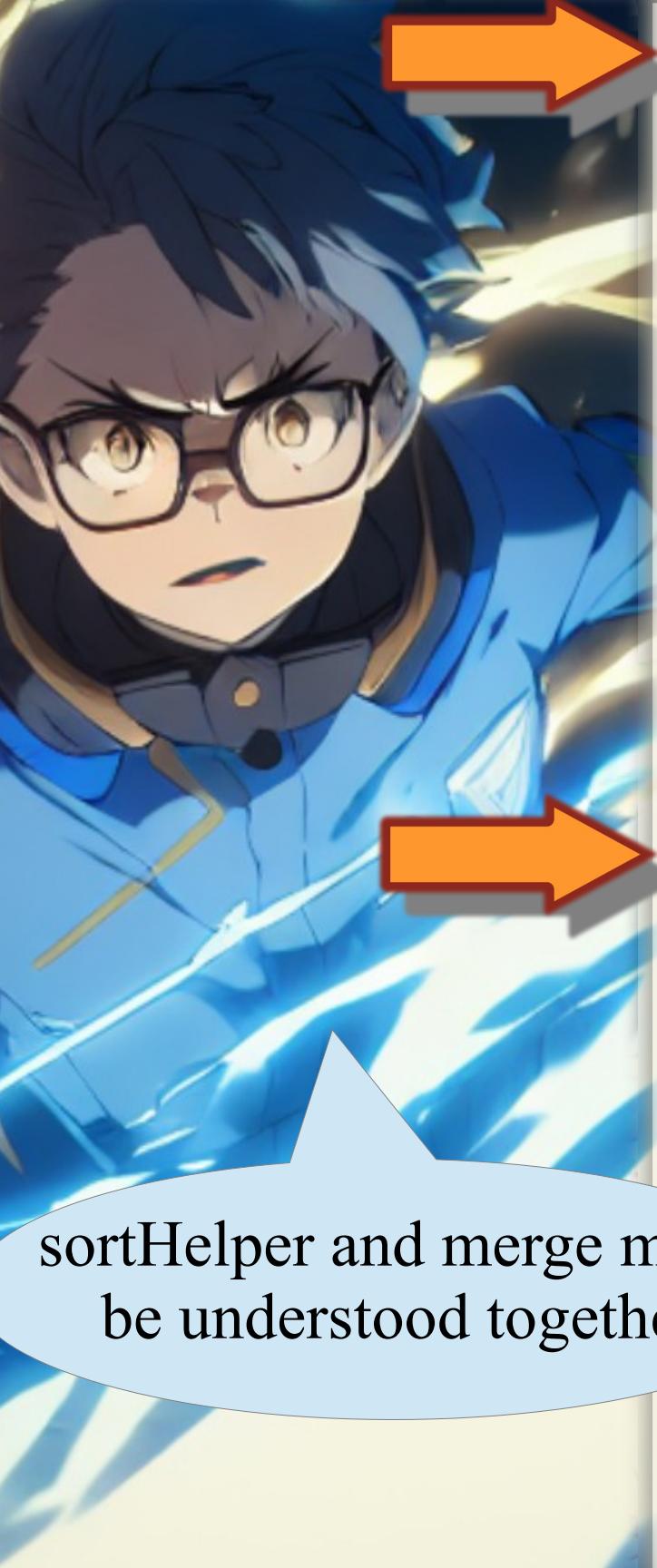
We take in input a list.
It could be unmodifiable

So we make two copies of the input,
both modifiable ArrayLists.

Finally, it looks cleaner to
return an unmodifiable result



```
private void sortHelper(List<T> tmp, List<T> result) {  
    int size= tmp.size();  
    if (size < 2) { return; }  
    int half = size / 2;  
    List<T> leftTmp      = tmp.subList(0, half);  
    List<T> rightTmp     = tmp.subList(half, size);  
    List<T> leftResult = result.subList(0, half);  
    List<T> rightResult= result.subList(half, size);  
    sortHelper(leftResult, leftTmp);//swap result/tmp  
    sortHelper(rightResult, rightTmp);//swap result/tmp  
    var resi= result.listIterator();  
    Consumer<T> c= e->{resi.next(); resi.set(e); };  
    merge(leftTmp.iterator(), rightTmp.iterator(), c);  
}  
  
private void merge(Iterator<T> ai, Iterator<T> bi, Consumer<T> res) {  
    T currentA= ai.next();  
    while (bi.hasNext()) {  
        T currentB= bi.next();  
        var bSmaller= currentB.compareTo(currentA) <= 0;  
        if (bSmaller) { res.accept(currentB); continue; }  
        res.accept(currentA);  
        Iterator<T> tmp = ai; ai = bi; bi = tmp;  
        currentA = currentB;  
    }  
    res.accept(currentA);  
    ai.forEachRemaining(res);  
}
```

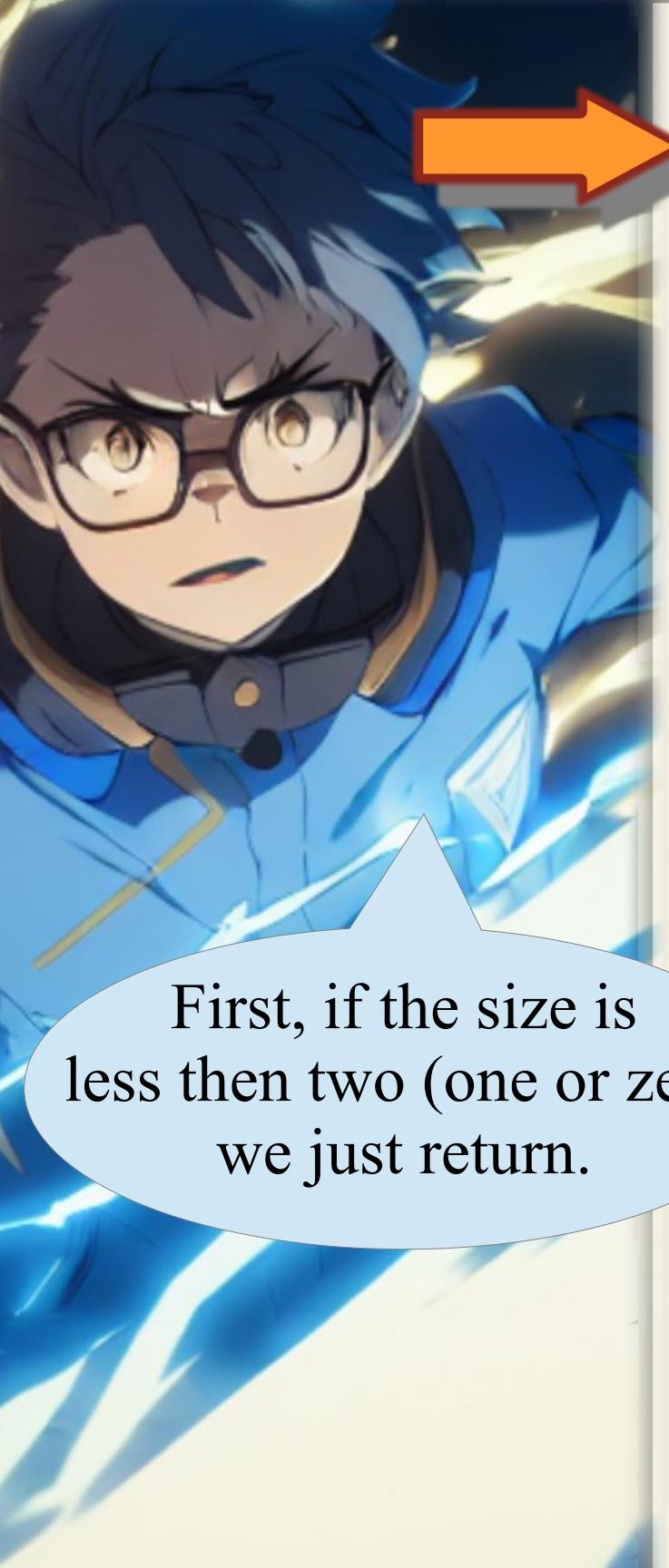


```
private void sortHelper(List<T> tmp, List<T> result) {  
    int size= tmp.size();  
    if (size < 2) { return; }  
    int half = size / 2;  
    List<T> leftTmp      = tmp.subList(0, half);  
    List<T> rightTmp     = tmp.subList(half, size);  
    List<T> leftResult = result.subList(0, half);  
    List<T> rightResult= result.subList(half, size);  
    sortHelper(leftResult, leftTmp);//swap result/tmp  
    sortHelper(rightResult, rightTmp);//swap result/tmp  
    var resi= result.listIterator();  
    Consumer<T> c= e->{resi.next(); resi.set(e); };  
    merge(leftTmp.iterator(), rightTmp.iterator(), c);  
}  
  
private void merge(Iterator<T> ai, Iterator<T> bi, Consumer<T> res) {  
    T currentA= ai.next();  
    while (bi.hasNext()) {  
        T currentB= bi.next();  
        var bSmaller= currentB.compareTo(currentA) <= 0;  
        if (bSmaller) { res.accept(currentB); continue; }  
        res.accept(currentA);  
        Iterator<T> tmp = ai; ai = bi; bi = tmp;  
        currentA = currentB;  
    }  
    res.accept(currentA);  
    ai.forEachRemaining(res);  
}
```

sortHelper and merge must
be understood together



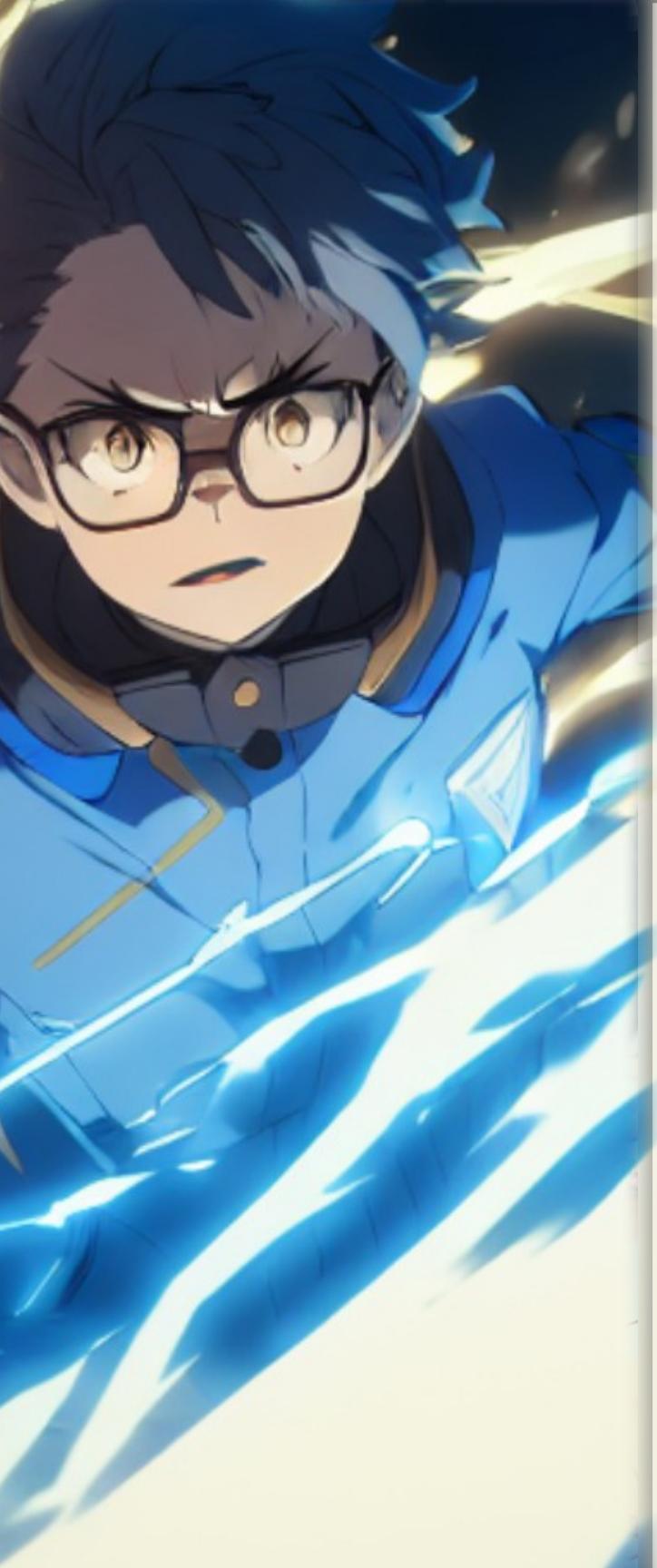
```
private void sortHelper(List<T> tmp, List<T> result) {  
    int size= tmp.size();  
    if (size < 2) { return; }  
    int half = size / 2;  
    List<T> leftTmp      = tmp.subList(0, half);  
    List<T> rightTmp     = tmp.subList(half, size);  
    List<T> leftResult = result.subList(0, half);  
    List<T> rightResult= result.subList(half, size);  
    sortHelper(leftResult, leftTmp);//swap result/tmp  
    sortHelper(rightResult, rightTmp);//swap result/tmp  
    var resi= result.listIterator();  
    Consumer<T> c= e->{resi.next(); resi.set(e); };  
    merge(leftTmp.iterator(), rightTmp.iterator(), c);  
}  
  
private void merge(Iterator<T> ai, Iterator<T> bi, Consumer<T> res) {  
    T currentA= ai.next();  
    while (bi.hasNext()) {  
        T currentB= bi.next();  
        var bSmaller= currentB.compareTo(currentA) <= 0;  
        if (bSmaller) { res.accept(currentB); continue; }  
        res.accept(currentA);  
        Iterator<T> tmp = ai; ai = bi; bi = tmp;  
        currentA = currentB;  
    }  
    res.accept(currentA);  
    ai.forEachRemaining(res);  
}
```



```
private void sortHelper(List<T> tmp, List<T> result) {
    int size= tmp.size();
    if (size < 2) { return; }
    int half = size / 2;
    List<T> leftTmp      = tmp.subList(0, half);
    List<T> rightTmp     = tmp.subList(half, size);
    List<T> leftResult = result.subList(0, half);
    List<T> rightResult= result.subList(half, size);
    sortHelper(leftResult, leftTmp); //swap result/tmp
    sortHelper(rightResult, rightTmp); //swap result/tmp
    var resi= result.listIterator();
    Consumer<T> c= e->{resi.next(); resi.set(e); };
    merge(leftTmp.iterator(), rightTmp.iterator(), c);
}

private void merge(Iterator<T> ai, Iterator<T> bi, Consumer<T> res) {
    T currentA= ai.next();
    while (bi.hasNext()) {
        T currentB= bi.next();
        var bSmaller= currentB.compareTo(currentA) <= 0;
        if (bSmaller) { res.accept(currentB); continue; }
        res.accept(currentA);
        Iterator<T> tmp = ai; ai = bi; bi = tmp;
        currentA = currentB;
    }
    res.accept(currentA);
    ai.forEachRemaining(res);
}
```

First, if the size is less then two (one or zero) we just return.

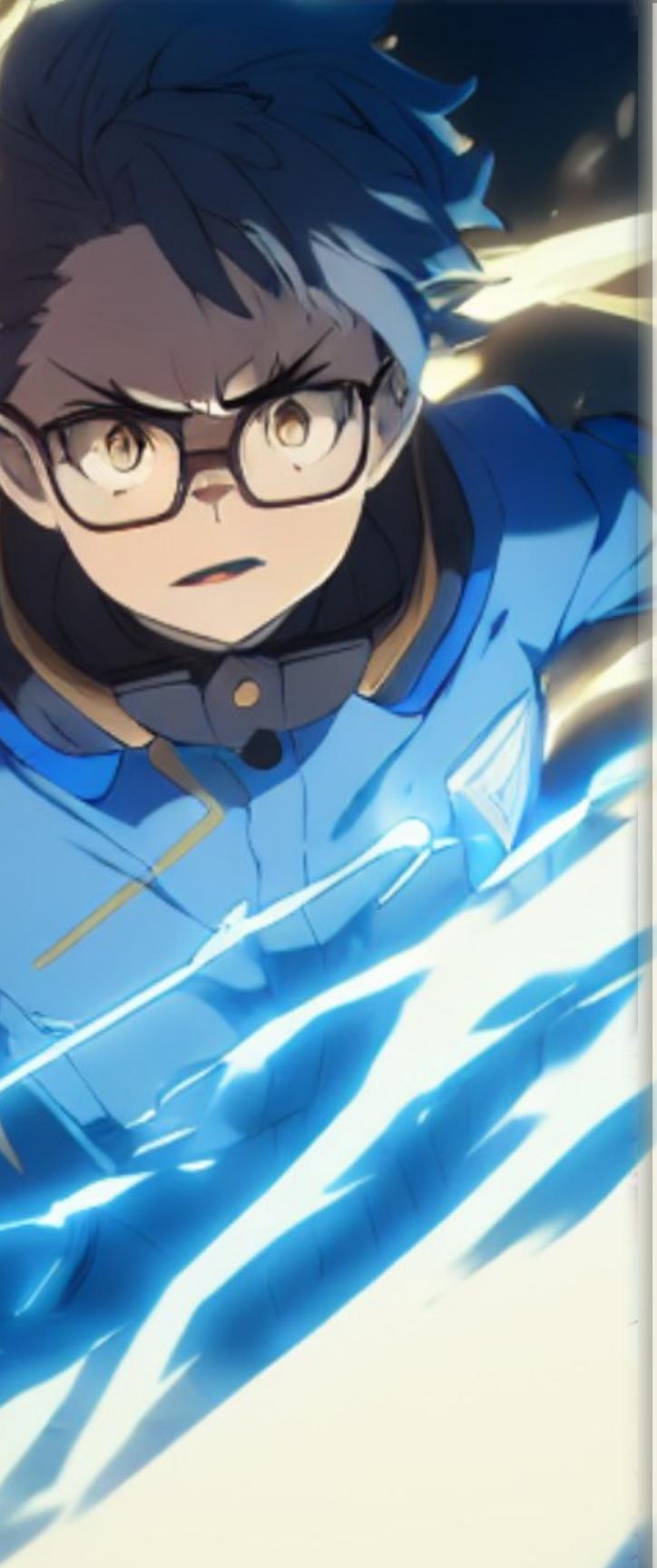


```
private void sortHelper(List<T> tmp, List<T> result) {  
    int size= tmp.size();  
    if (size < 2) { return; }  
    int half = size / 2;  
    List<T> leftTmp      = tmp.subList(0, half);  
    List<T> rightTmp     = tmp.subList(half, size);  
    List<T> leftResult  = result.subList(0, half);  
    List<T> rightResult = result.subList(half, size);  
    sortHelper(leftResult, leftTmp); //swap result/tmp  
    sortHelper(rightResult, rightTmp); //swap result/tmp  
    var resi= result.listIterator();  
    Consumer<T> c= e->{resi.next(); resi.set(e); };  
    merge(leftTmp.iterator(), rightTmp.iterator(), c);  
}  
  
private void merge(Iterator<T> ai, Iterator<T> bi, Consumer<T> res) {  
    T currentA= ai.next();  
    while (bi.hasNext()) {  
        T currentB= bi.next();  
        var bSmaller= currentB.compareTo(currentA) <= 0;  
        if (bSmaller) { res.accept(currentB); continue; }  
        res.accept(currentA);  
        Iterator<T> tmp = ai; ai = bi; bi = tmp;  
        currentA = currentB;  
    }  
    res.accept(currentA);  
    ai.forEachRemaining(res);  
}
```



Then we split both the input and the result in two parts using subList

```
private void sortHelper(List<T> tmp, List<T> result) {  
    int size= tmp.size();  
    if (size < 2) { return; }  
    int half = size / 2;  
    List<T> leftTmp      = tmp.subList(0, half);  
    List<T> rightTmp     = tmp.subList(half, size);  
    List<T> leftResult  = result.subList(0, half);  
    List<T> rightResult = result.subList(half, size);  
    sortHelper(leftResult, leftTmp); //swap result/tmp  
    sortHelper(rightResult, rightTmp); //swap result/tmp  
    var resi= result.listIterator();  
    Consumer<T> c= e->{resi.next(); resi.set(e); };  
    merge(leftTmp.iterator(), rightTmp.iterator(), c);  
}  
  
private void merge(Iterator<T> ai, Iterator<T> bi, Consumer<T> res) {  
    T currentA= ai.next();  
    while (bi.hasNext()) {  
        T currentB= bi.next();  
        var bSmaller= currentB.compareTo(currentA) <= 0;  
        if (bSmaller) { res.accept(currentB); continue; }  
        res.accept(currentA);  
        Iterator<T> tmp = ai; ai = bi; bi = tmp;  
        currentA = currentB;  
    }  
    res.accept(currentA);  
    ai.forEachRemaining(res);  
}
```



```
private void sortHelper(List<T> tmp, List<T> result) {  
    int size= tmp.size();  
    if (size < 2) { return; }  
    int half = size / 2;  
    List<T> leftTmp      = tmp.subList(0, half);  
    List<T> rightTmp     = tmp.subList(half, size);  
    List<T> leftResult  = result.subList(0, half);  
    List<T> rightResult= result.subList(half, size);  
    sortHelper(leftResult, leftTmp); //swap result/tmp  
    sortHelper(rightResult, rightTmp); //swap result/tmp  
    var resi= result.listIterator();  
    Consumer<T> c= e->{resi.next(); resi.set(e); };  
    merge(leftTmp.iterator(), rightTmp.iterator(), c);  
}  
  
private void merge(Iterator<T> ai, Iterator<T> bi, Consumer<T> res) {  
    T currentA= ai.next();  
    while (bi.hasNext()) {  
        T currentB= bi.next();  
        var bSmaller= currentB.compareTo(currentA) <= 0;  
        if (bSmaller) { res.accept(currentB); continue; }  
        res.accept(currentA);  
        Iterator<T> tmp = ai; ai = bi; bi = tmp;  
        currentA = currentB;  
    }  
    res.accept(currentA);  
    ai.forEachRemaining(res);  
}
```



```
private void sortHelper(List<T> tmp, List<T> result) {  
    int size= tmp.size();  
    if (size < 2) { return; }  
    int half = size / 2;  
    List<T> leftTmp      = tmp.subList(0, half);  
    List<T> rightTmp     = tmp.subList(half, size);  
    List<T> leftResult  = result.subList(0, half);  
    List<T> rightResult= result.subList(half, size);  
    sortHelper(leftResult, leftTmp); //swap result/tmp  
    sortHelper(rightResult, rightTmp); //swap result/tmp  
    var resi= result.listIterator();  
    Consumer<T> c= e->{resi.next(); resi.set(e); };  
    merge(leftTmp.iterator(), rightTmp.iterator(), c);  
}  
  
private void merge(Iterator<T> ai, Iterator<T> bi, Consumer<T> res) {  
    T currentA= ai.next();  
    while (bi.hasNext()) {  
        T currentB= bi.next();  
        var bSmaller= currentB.compareTo(currentA) <= 0;  
        if (bSmaller) { res.accept(currentB); continue; }  
        res.accept(currentA);  
        Iterator<T> tmp = ai; ai = bi; bi = tmp;  
        currentA = currentB;  
    }  
    res.accept(currentA);  
    ai.forEachRemaining(res);  
}
```

Then it is swap-recusion time!



IT IS
SWAP-RECURSION
TIME!



tmp



tmp

result





We receive ‘tmp’ and ‘result’
and we need to write in ‘result’

tmp

result





tmp

We receive ‘tmp’ and ‘result’
and we need to write in ‘result’

result

At the start, ‘result’
contains the data to sort

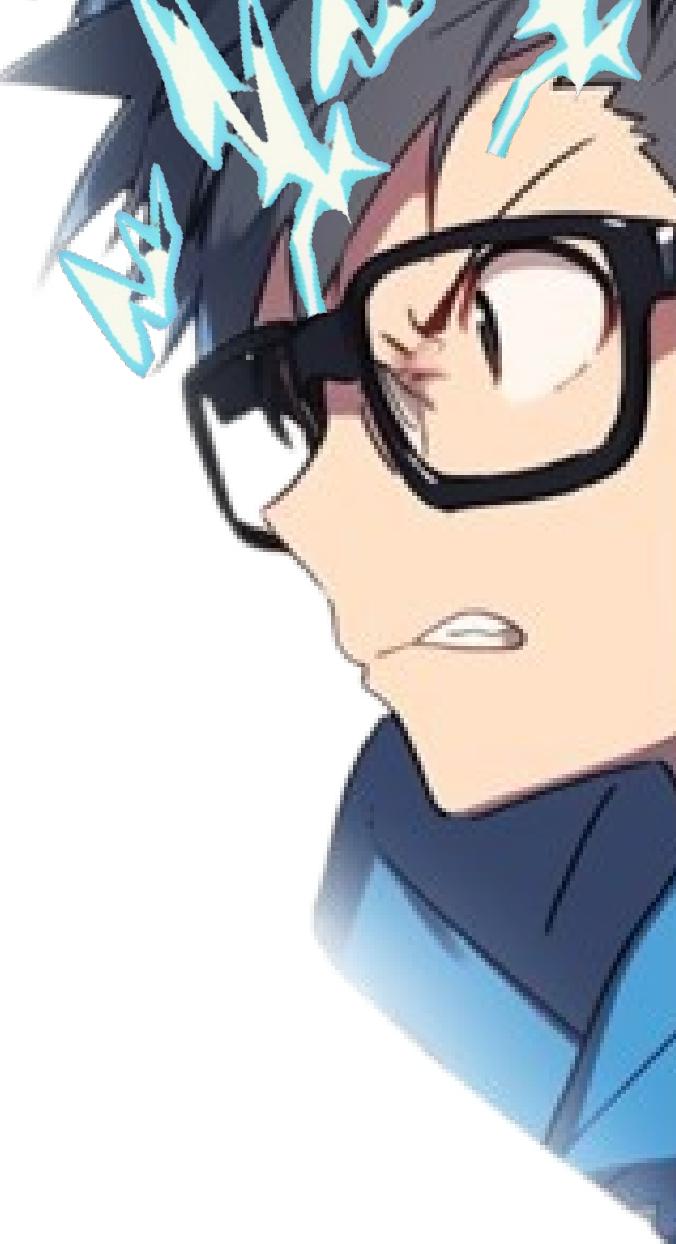
At the end, ‘result’
must contains the sorted data





tmp

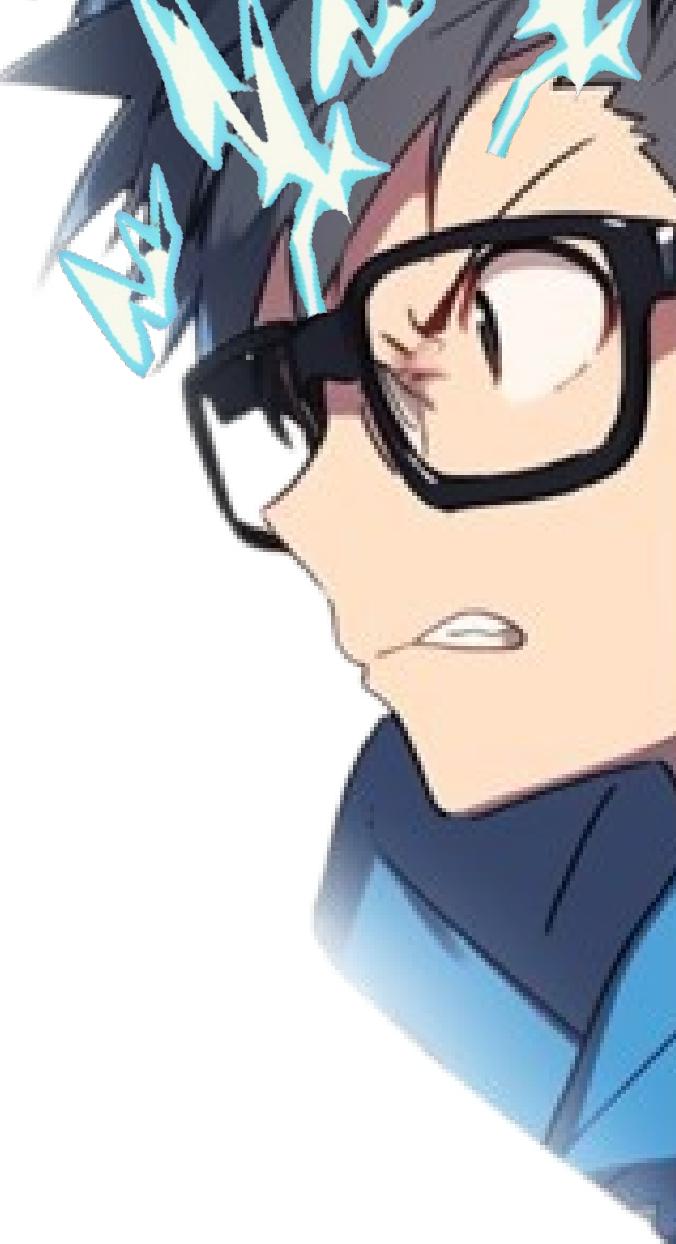
result

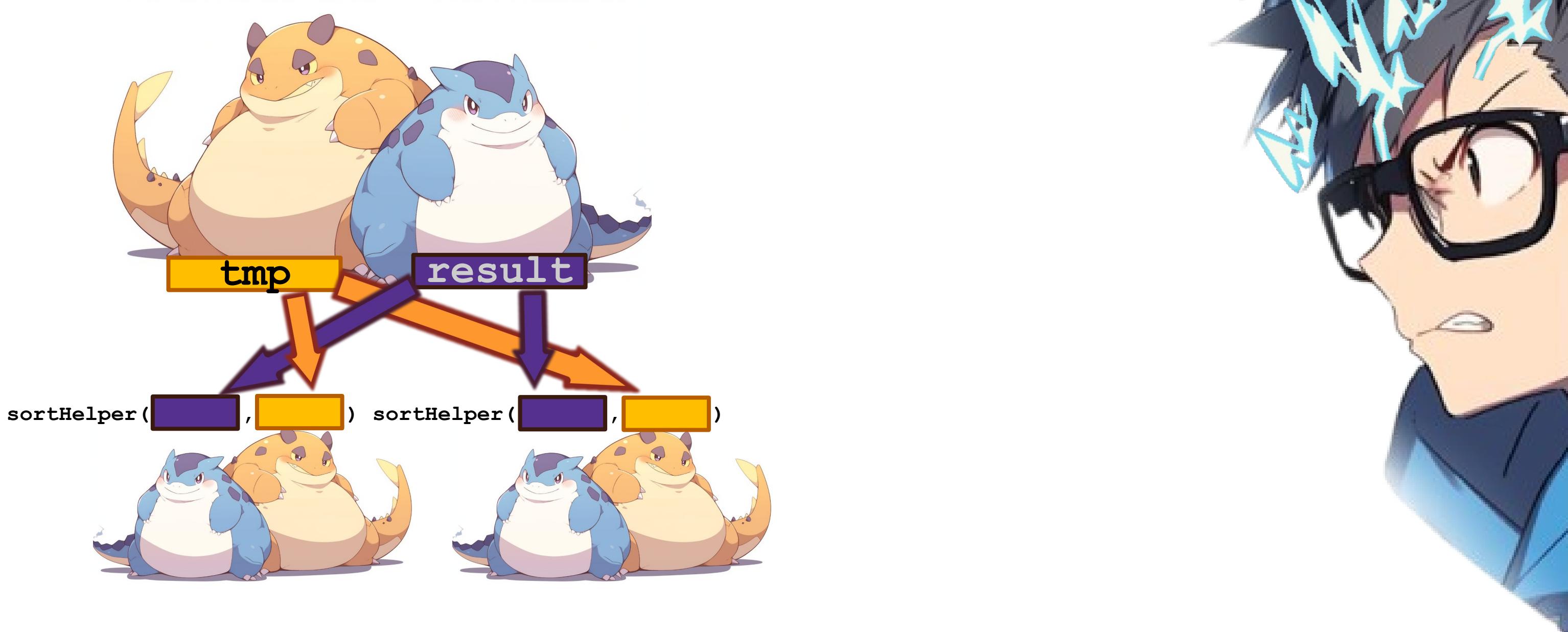


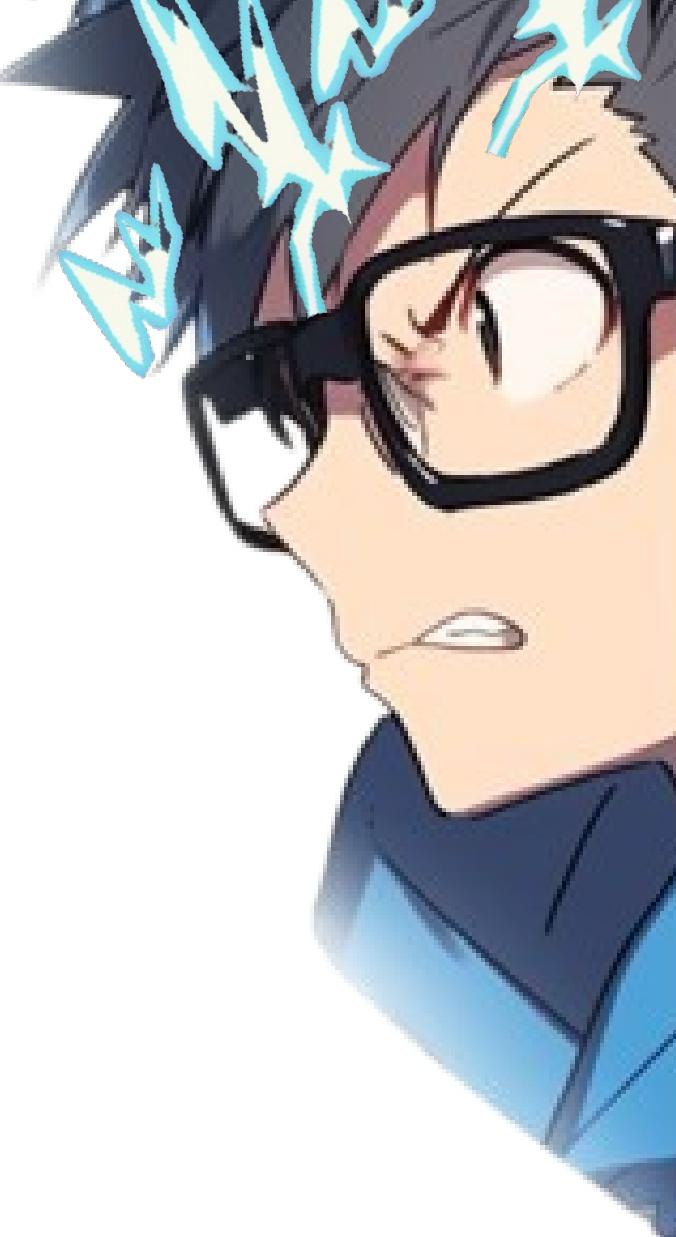


tmp

result







We split 'tmp' and 'result' in two,
so we get 4 sublists





We split ‘tmp’ and ‘result’ in two,
so we get 4 sublists

We pass ‘tmp’ and ‘result’ to
the recursion, but swapped!



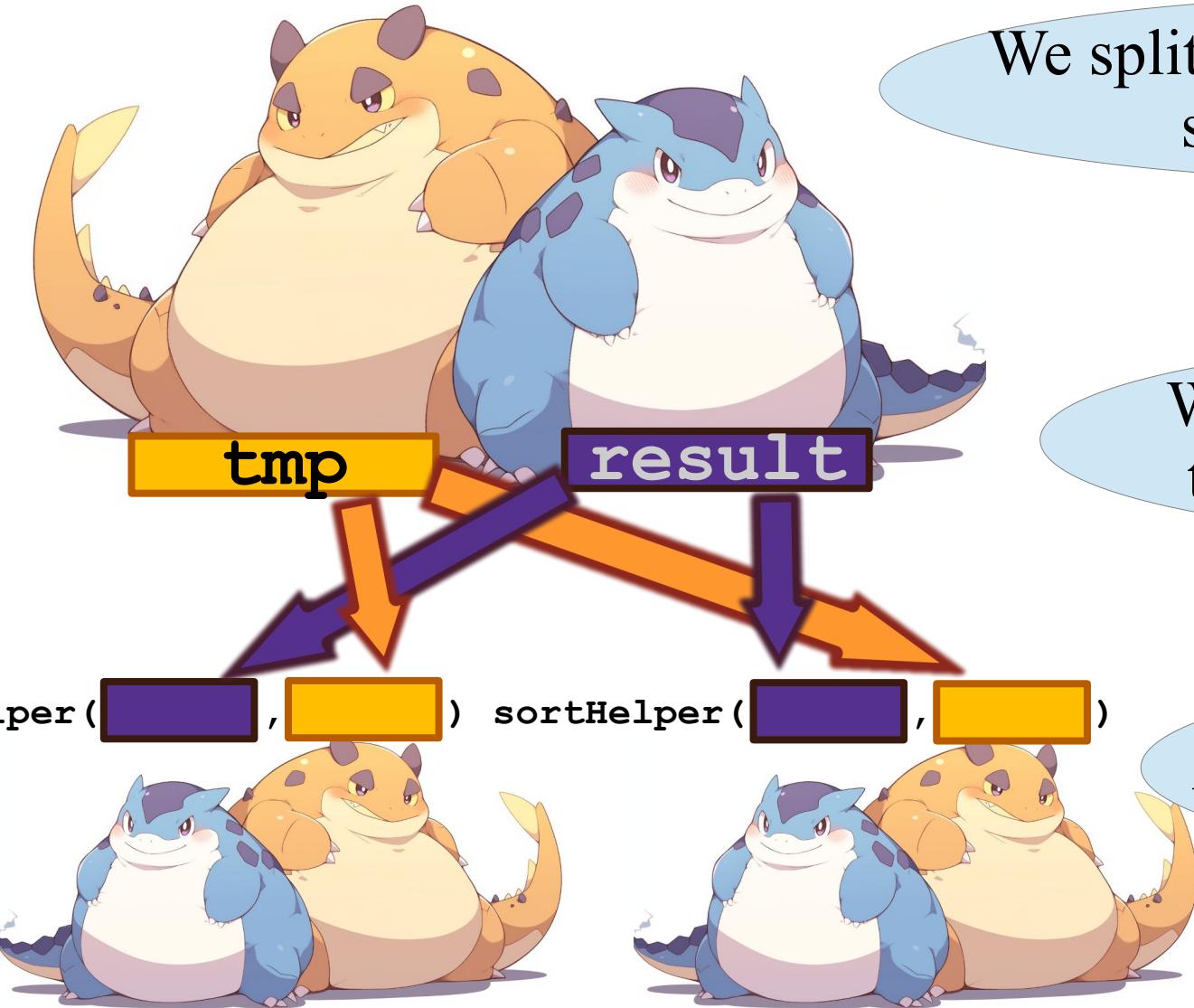


We split ‘tmp’ and ‘result’ in two,
so we get 4 sublists

We pass ‘tmp’ and ‘result’ to
the recursion, but swapped!

After recursion, ‘result’ is scribbled over.





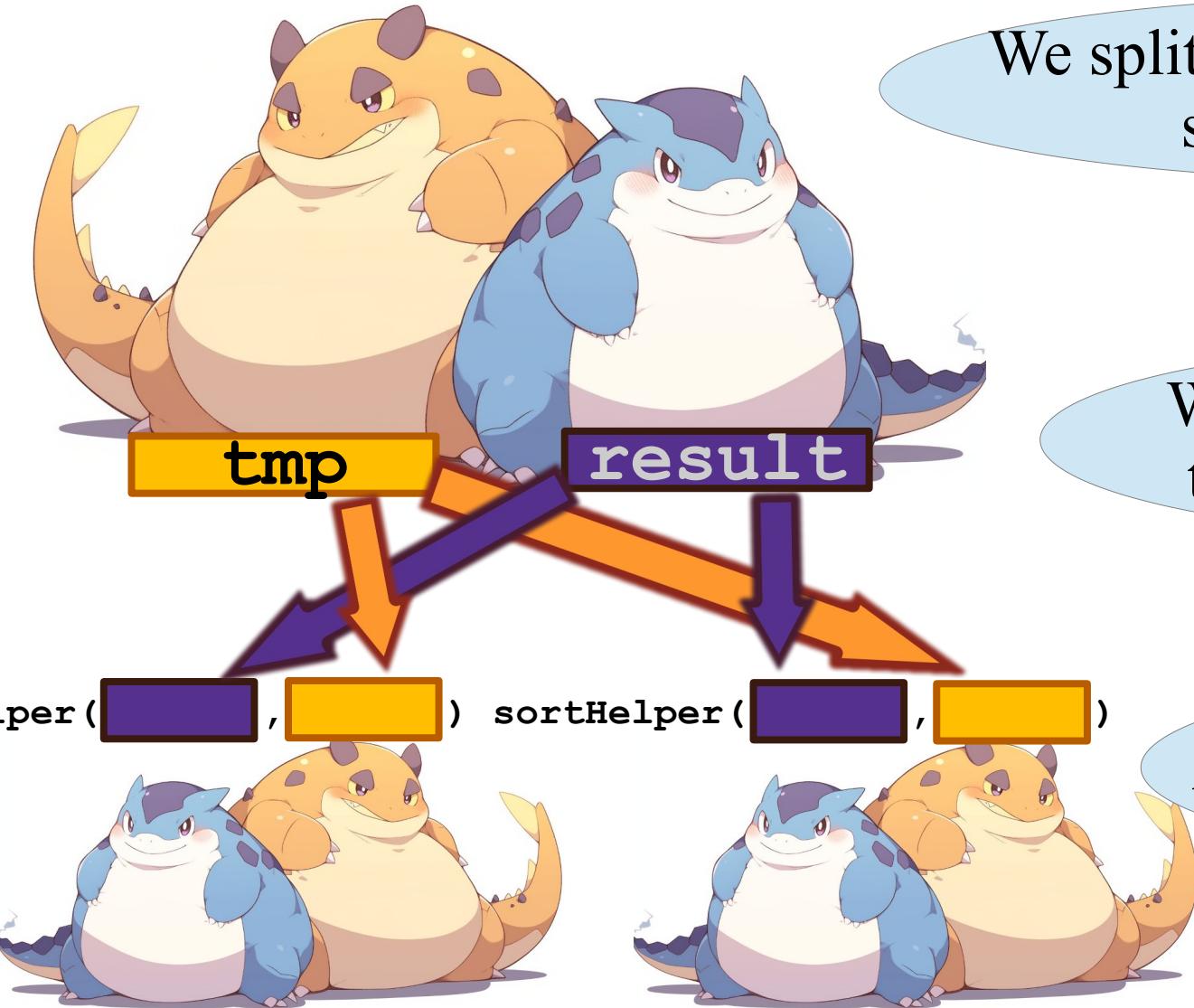
We split 'tmp' and 'result' in two,
so we get 4 sublists

We pass 'tmp' and 'result' to
the recursion, but swapped!

After recursion, 'result' is scribbled over.

After recursion, the yellow dragon 'tmp'
is holding semi sorted data
from the blue dragon 'result'





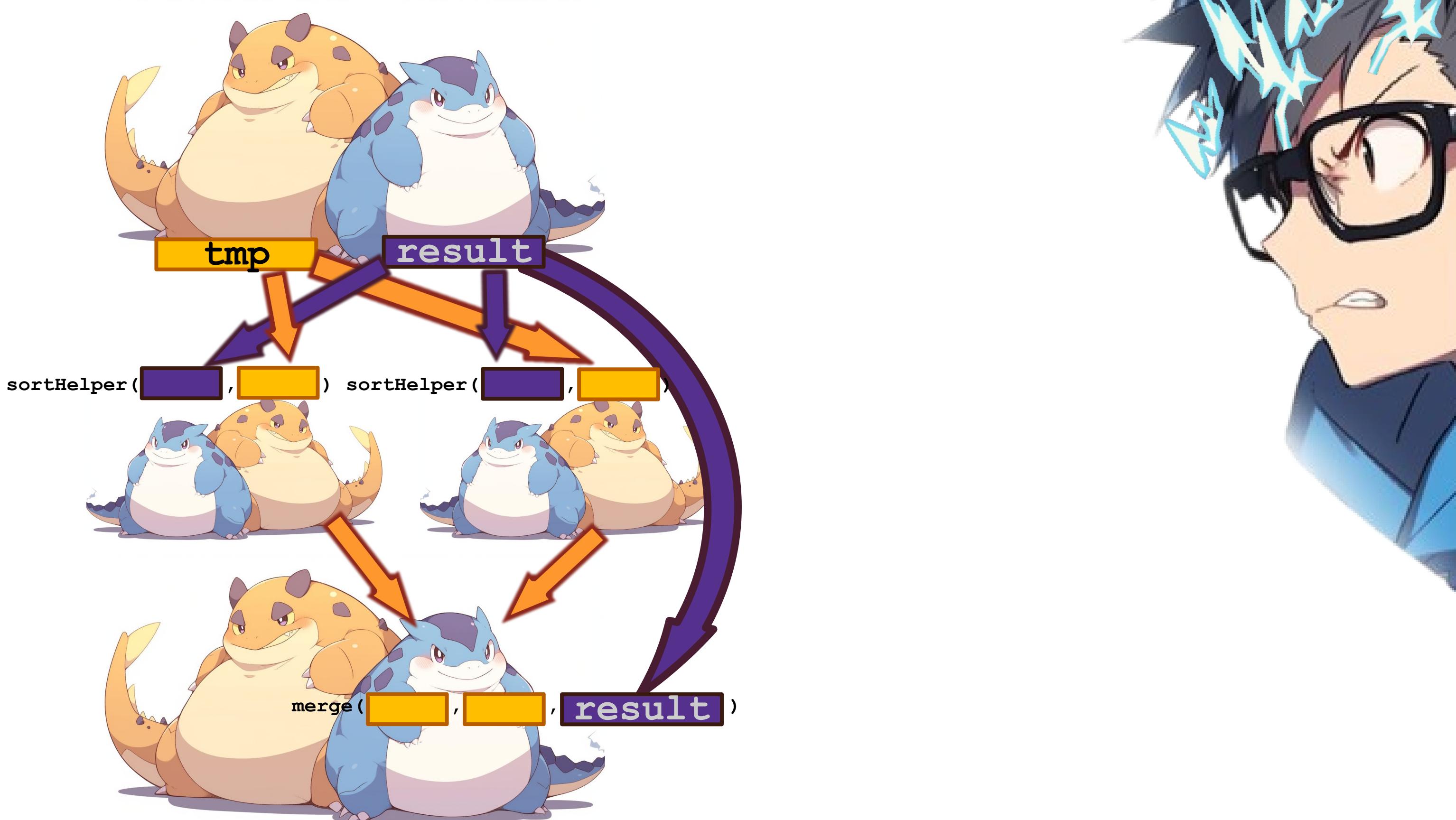
We split 'tmp' and 'result' in two,
so we get 4 sublists

We pass 'tmp' and 'result' to
the recursion, but swapped!

After recursion, the yellow dragon 'tmp'
is holding semi sorted data
from the blue dragon 'result'

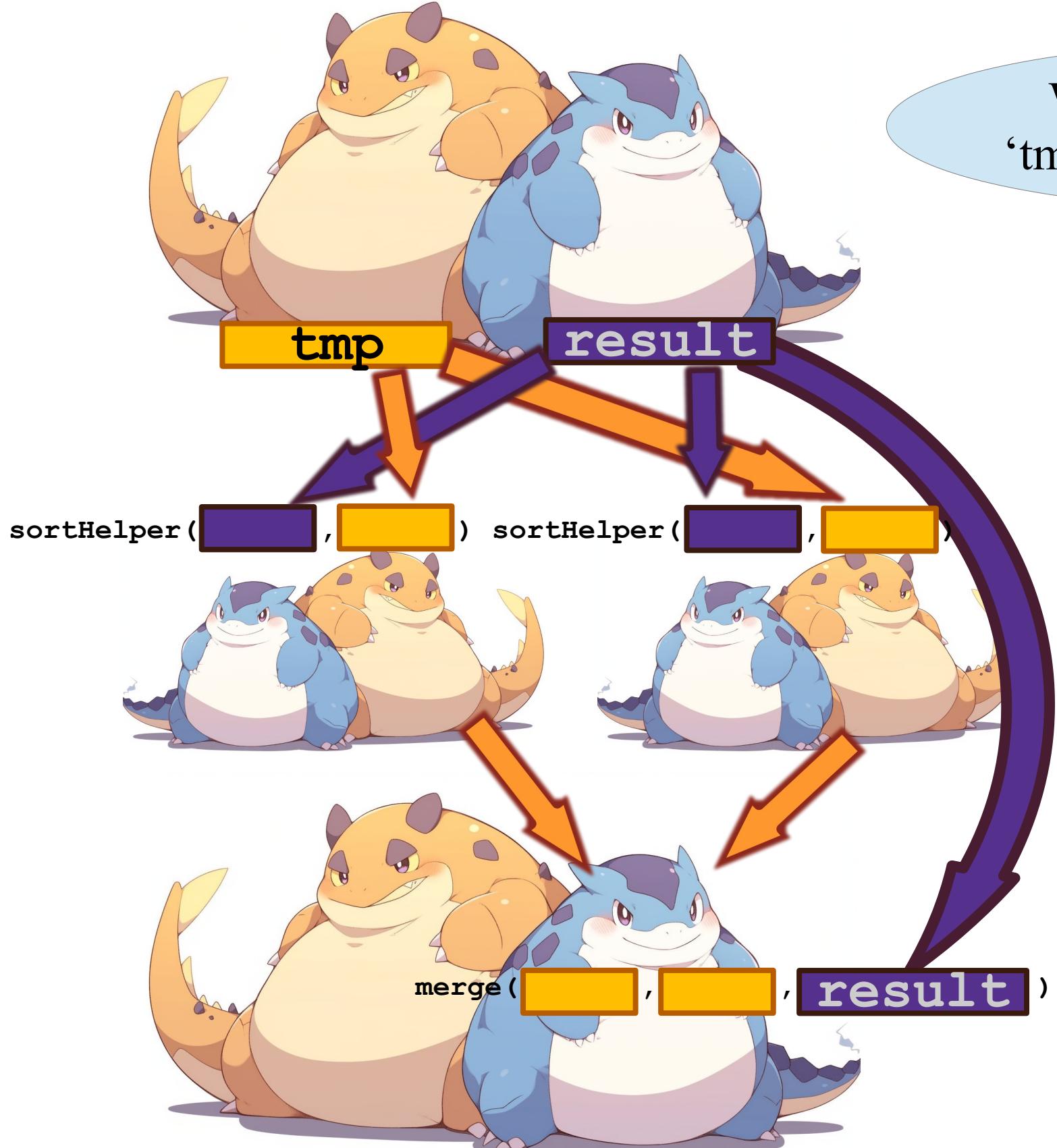
That is, the two sublists
composing 'tmp' are now sorted

After recursion, 'result' is scribbled over.



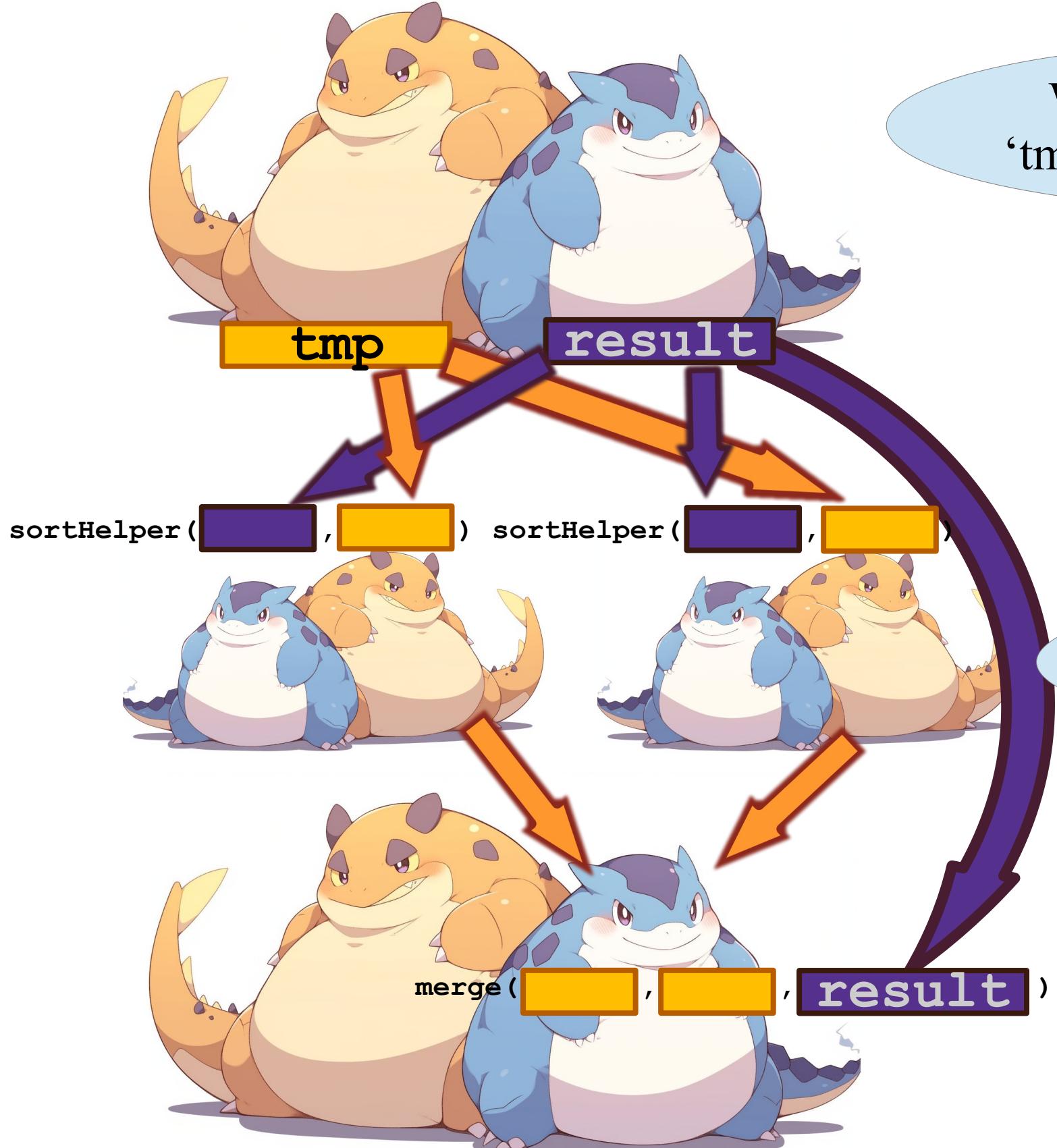


We then merge the two
‘tmp sublist’ into ‘result’





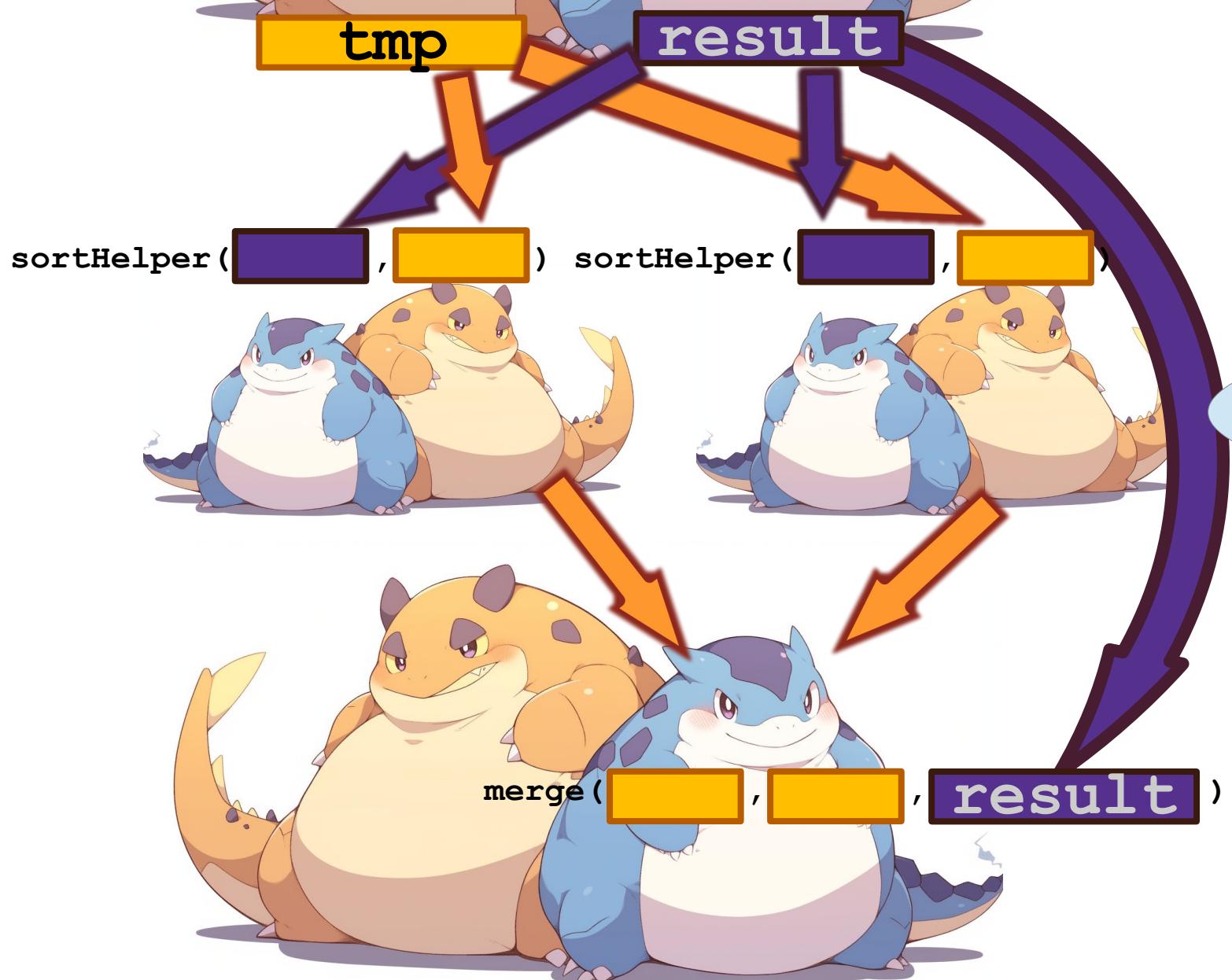
We then merge the two
‘tmp sublist’ into ‘result’



At the end ‘tmp’ is split sorted ...

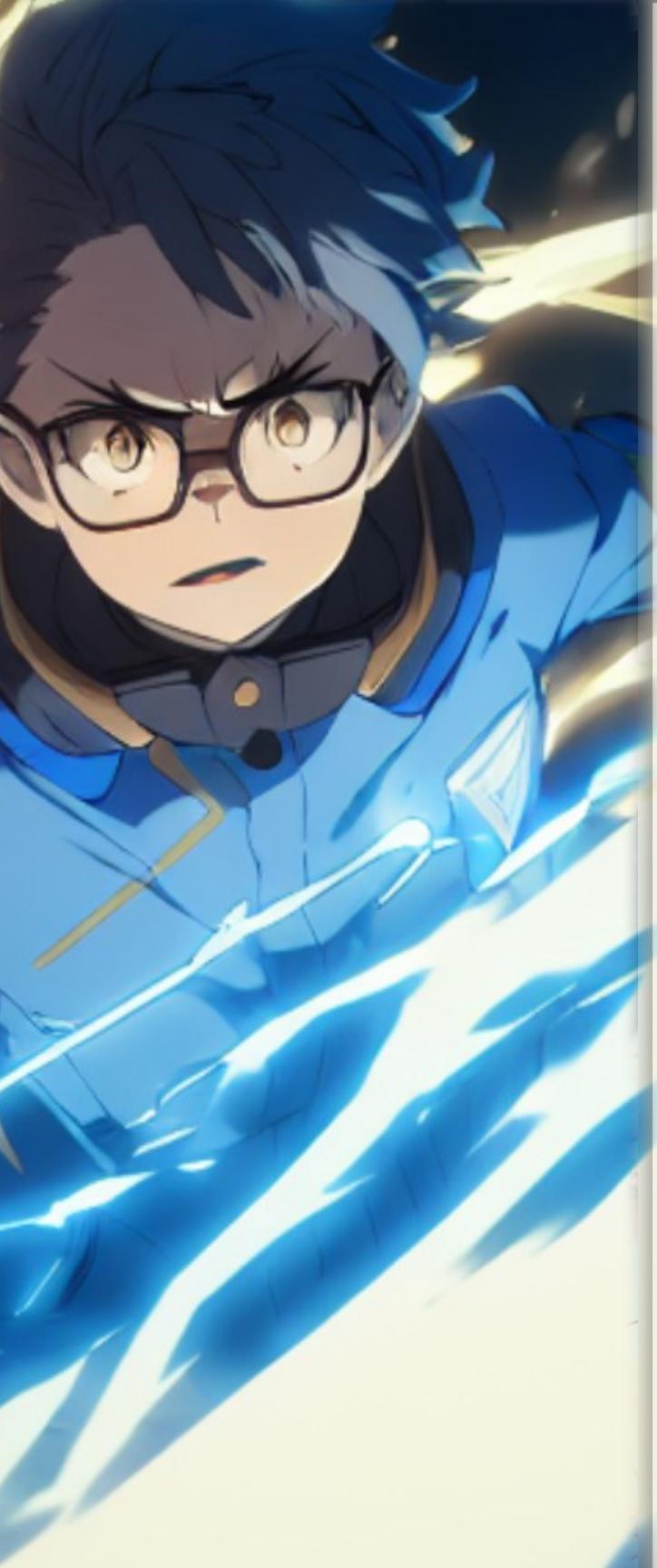


We then merge the two
'tmp sublist' into 'result'

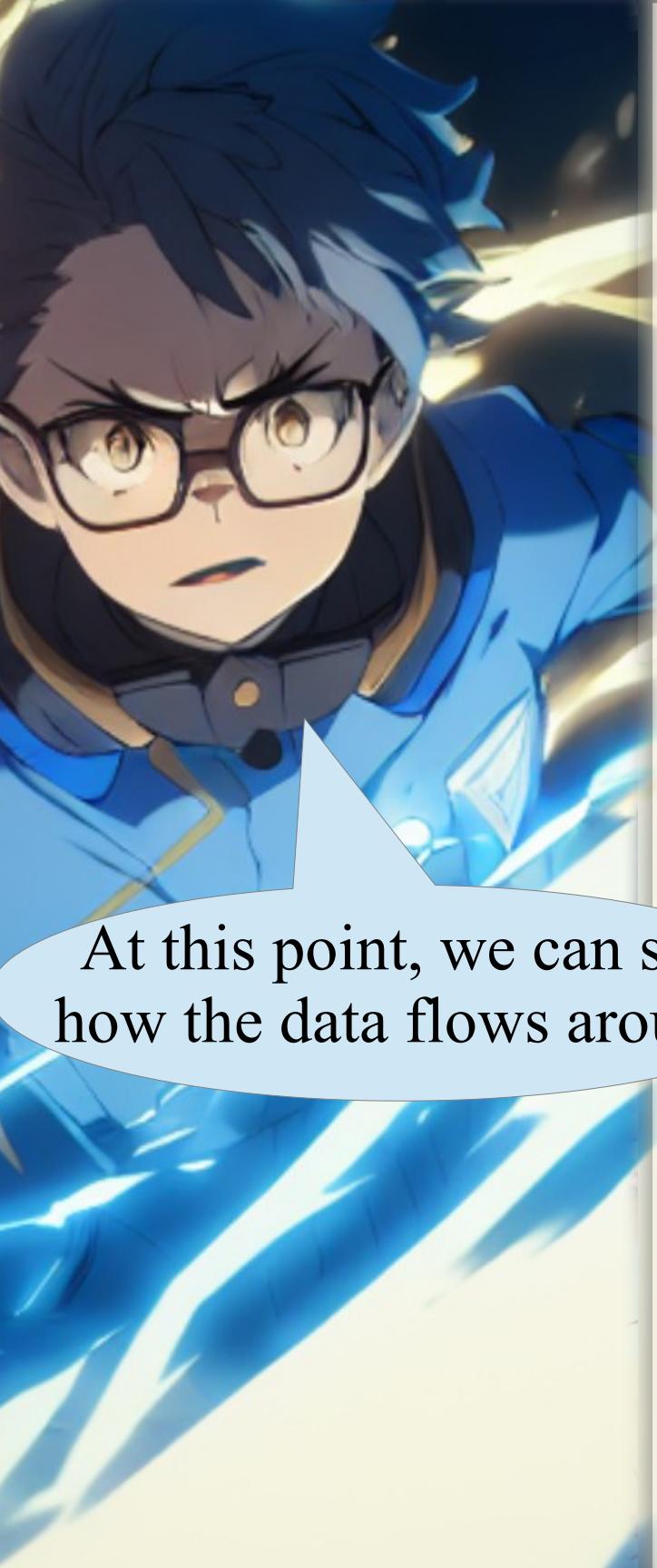


At the end 'tmp' is split sorted ...

and 'result' is fully sorted.

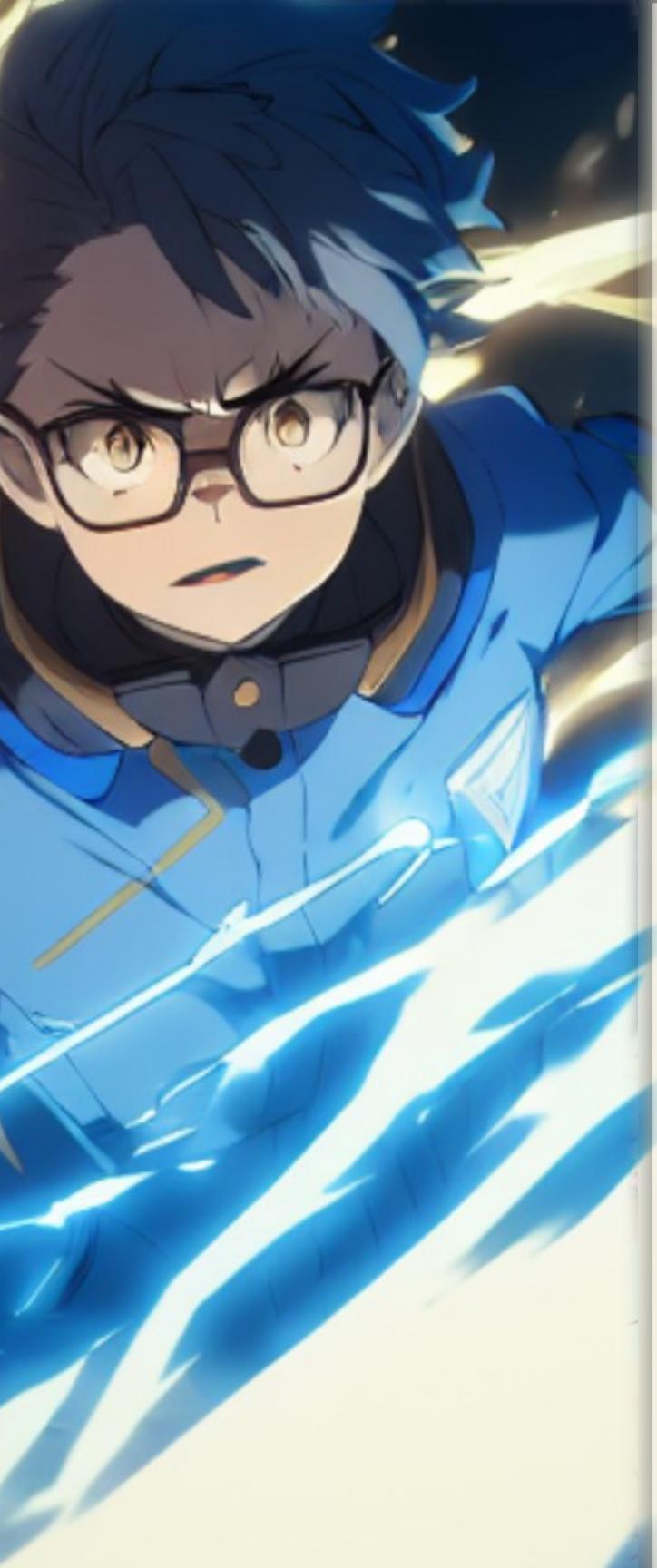


```
private void sortHelper(List<T> tmp, List<T> result) {  
    int size= tmp.size();  
    if (size < 2) { return; }  
    int half = size / 2;  
    List<T> leftTmp      = tmp.subList(0, half);  
    List<T> rightTmp     = tmp.subList(half, size);  
    List<T> leftResult  = result.subList(0, half);  
    List<T> rightResult= result.subList(half, size);  
    sortHelper(leftResult, leftTmp); //swap result/tmp  
    sortHelper(rightResult, rightTmp); //swap result/tmp  
    var resi= result.listIterator();  
    Consumer<T> c= e->{resi.next(); resi.set(e); };  
    merge(leftTmp.iterator(), rightTmp.iterator(), ~result~);  
}  
  
private void merge(Iterator<T> ai, Iterator<T> bi, Consumer<T> res) {  
    T currentA= ai.next();  
    while (bi.hasNext()) {  
        T currentB= bi.next();  
        var bSmaller= currentB.compareTo(currentA) <= 0;  
        if (bSmaller) { res.accept(currentB); continue; }  
        res.accept(currentA);  
        Iterator<T> tmp = ai; ai = bi; bi = tmp;  
        currentA = currentB;  
    }  
    res.accept(currentA);  
    ai.forEachRemaining(res);  
}
```

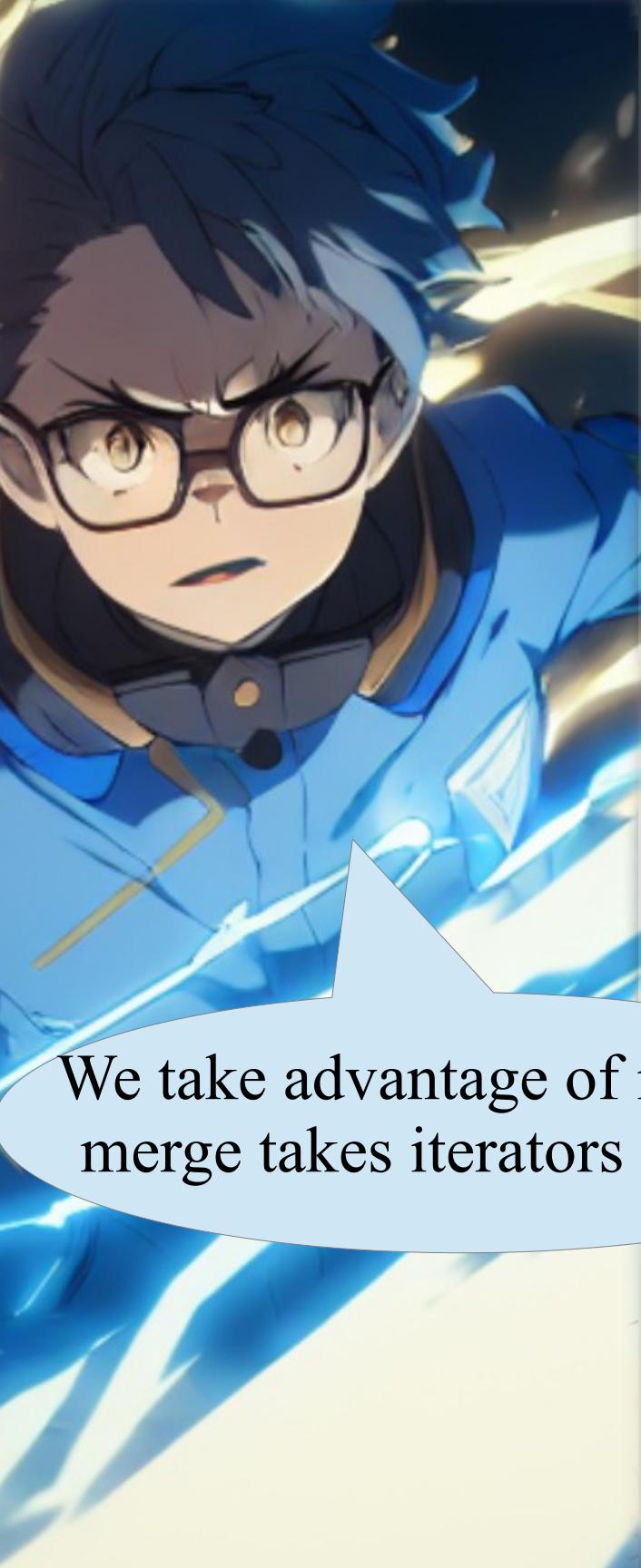


At this point, we can see
how the data flows around

```
private void sortHelper(List<T> tmp, List<T> result) {  
    int size= tmp.size();  
    if (size < 2) { return; }  
    int half = size / 2;  
    List<T> leftTmp      = tmp.subList(0, half);  
    List<T> rightTmp     = tmp.subList(half, size);  
    List<T> leftResult  = result.subList(0, half);  
    List<T> rightResult= result.subList(half, size);  
    sortHelper(leftResult, leftTmp); //swap result/tmp  
    sortHelper(rightResult, rightTmp); //swap result/tmp  
    var resi= result.listIterator();  
    Consumer<T> c= e->{resi.next(); resi.set(e); };  
    merge(leftTmp.iterator(), rightTmp.iterator(), ~result~);  
}  
  
private void merge(Iterator<T> ai, Iterator<T> bi, Consumer<T> res) {  
    T currentA= ai.next();  
    while (bi.hasNext()) {  
        T currentB= bi.next();  
        var bSmaller= currentB.compareTo(currentA) <= 0;  
        if (bSmaller) { res.accept(currentB); continue; }  
        res.accept(currentA);  
        Iterator<T> tmp = ai; ai = bi; bi = tmp;  
        currentA = currentB;  
    }  
    res.accept(currentA);  
    ai.forEachRemaining(res);  
}
```

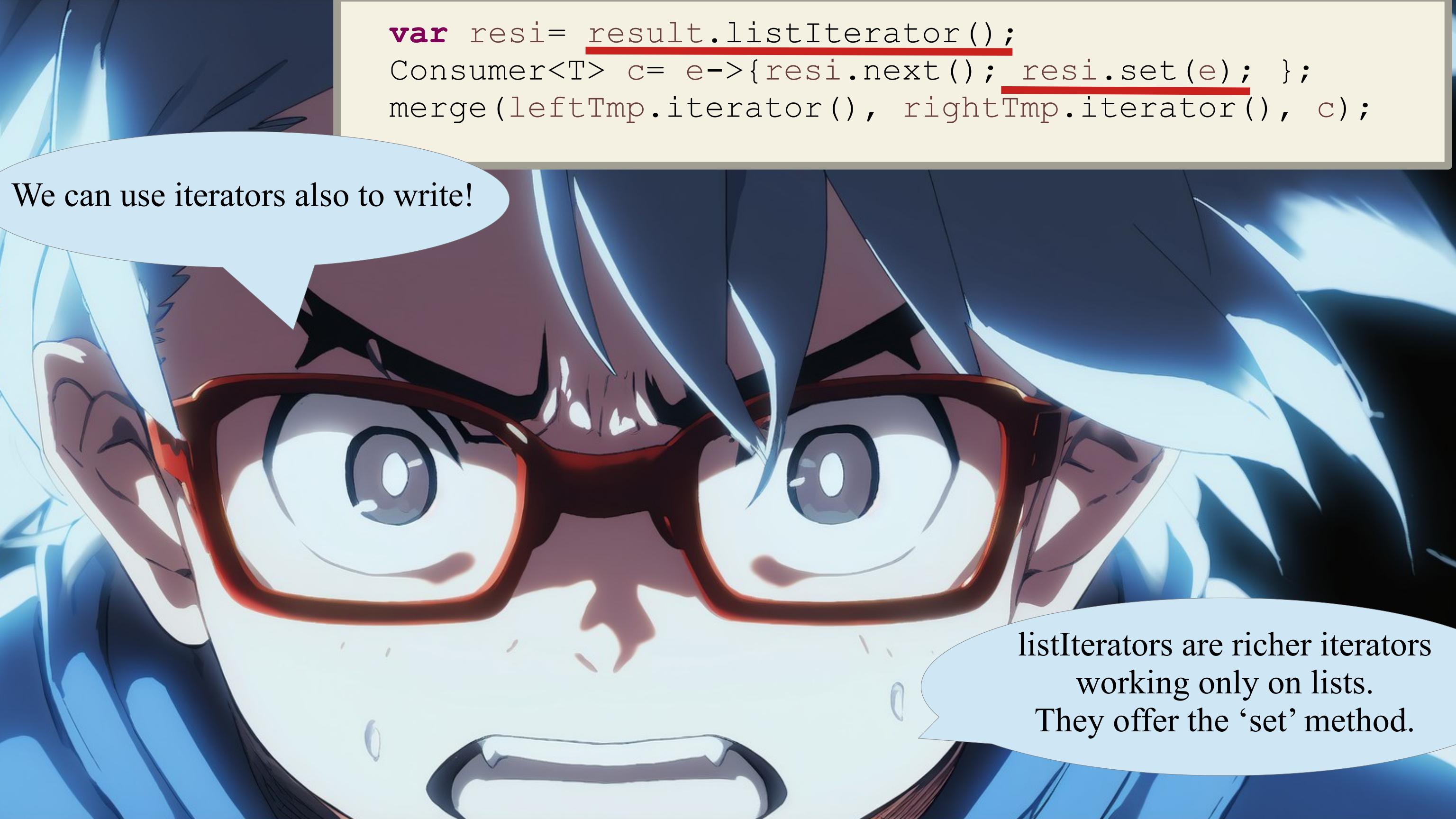


```
private void sortHelper(List<T> tmp, List<T> result) {  
    int size= tmp.size();  
    if (size < 2) { return; }  
    int half = size / 2;  
    List<T> leftTmp      = tmp.subList(0, half);  
    List<T> rightTmp     = tmp.subList(half, size);  
    List<T> leftResult  = result.subList(0, half);  
    List<T> rightResult= result.subList(half, size);  
    sortHelper(leftResult, leftTmp); //swap result/tmp  
    sortHelper(rightResult, rightTmp); //swap result/tmp  
    var resi= result.listIterator();  
    Consumer<T> c= e->{resi.next(); resi.set(e); };  
    merge(leftTmp.iterator(), rightTmp.iterator(), c);  
}  
  
private void merge(Iterator<T> ai, Iterator<T> bi, Consumer<T> res) {  
    T currentA= ai.next();  
    while (bi.hasNext()) {  
        T currentB= bi.next();  
        var bSmaller= currentB.compareTo(currentA) <= 0;  
        if (bSmaller) { res.accept(currentB); continue; }  
        res.accept(currentA);  
        Iterator<T> tmp = ai; ai = bi; bi = tmp;  
        currentA = currentB;  
    }  
    res.accept(currentA);  
    ai.forEachRemaining(res);  
}
```



```
private void sortHelper(List<T> tmp, List<T> result) {  
    int size= tmp.size();  
    if (size < 2) { return; }  
    int half = size / 2;  
    List<T> leftTmp      = tmp.subList(0, half);  
    List<T> rightTmp     = tmp.subList(half, size);  
    List<T> leftResult  = result.subList(0, half);  
    List<T> rightResult= result.subList(half, size);  
    sortHelper(leftResult, leftTmp); //swap result/tmp  
    sortHelper(rightResult, rightTmp); //swap result/tmp  
    var resi= result.listIterator();  
    Consumer<T> c= e->{resi.next(); resi.set(e); };  
    merge(leftTmp.iterator(), rightTmp.iterator(), c);  
}  
  
private void merge(Iterator<T> ai, Iterator<T> bi, Consumer<T> res) {  
    T currentA= ai.next();  
    while (bi.hasNext()) {  
        T currentB= bi.next();  
        if (currentB.compareTo(currentA) <= 0;)  
            res.accept(currentB); continue;  
        res.accept(currentA);  
        Iterator<T> tmp = ai; ai = bi; bi = tmp;  
        currentA = currentB;  
    }  
    res.accept(currentA);  
    ai.forEachRemaining(res);  
}
```

We take advantage of iterators,
merge takes iterators in input



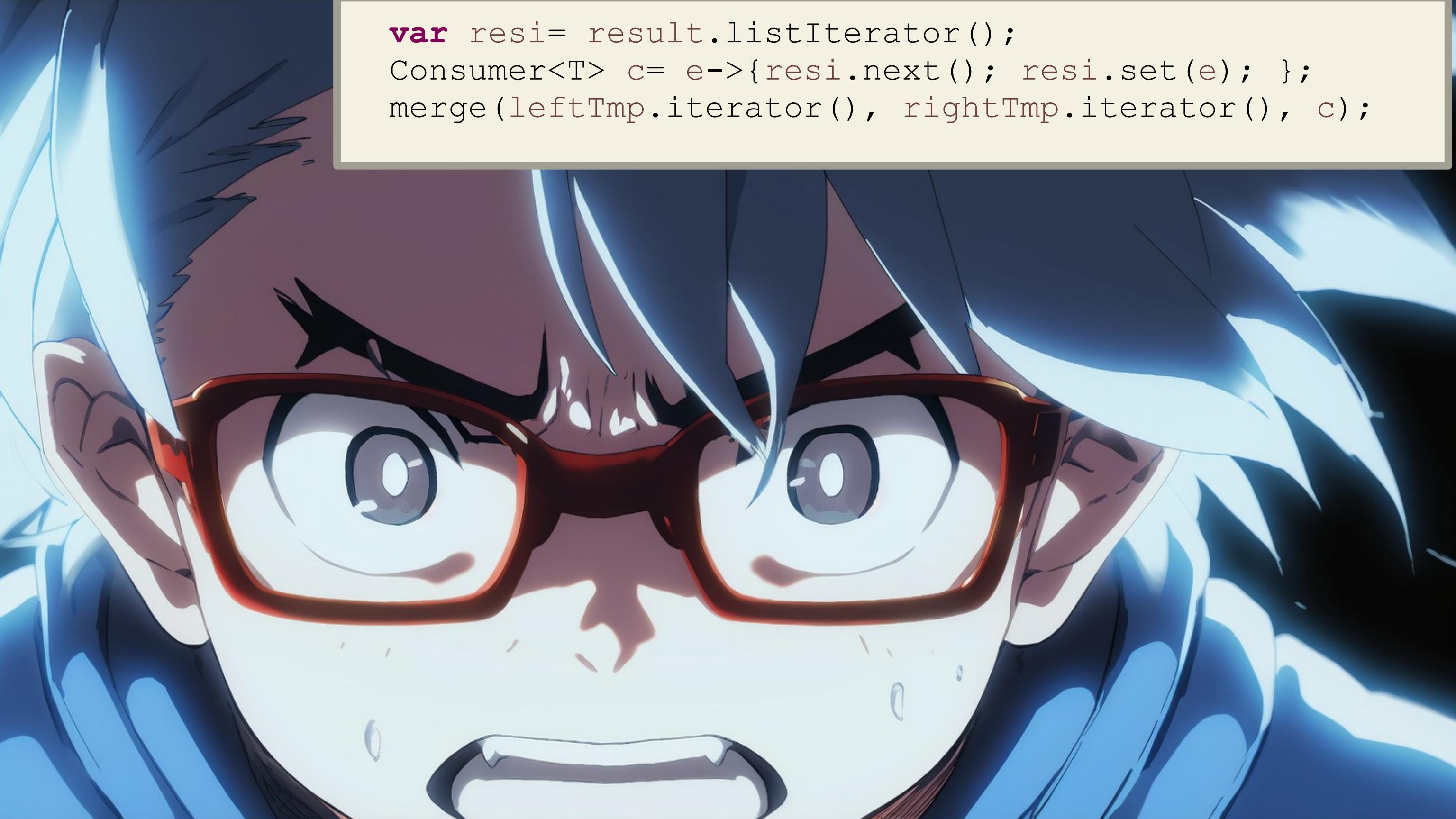
```
var resi= result.listIterator();  
Consumer<T> c= e->{resi.next(); resi.set(e); };  
merge(leftTmp.iterator(), rightTmp.iterator(), c);
```

We can use iterators also to write!

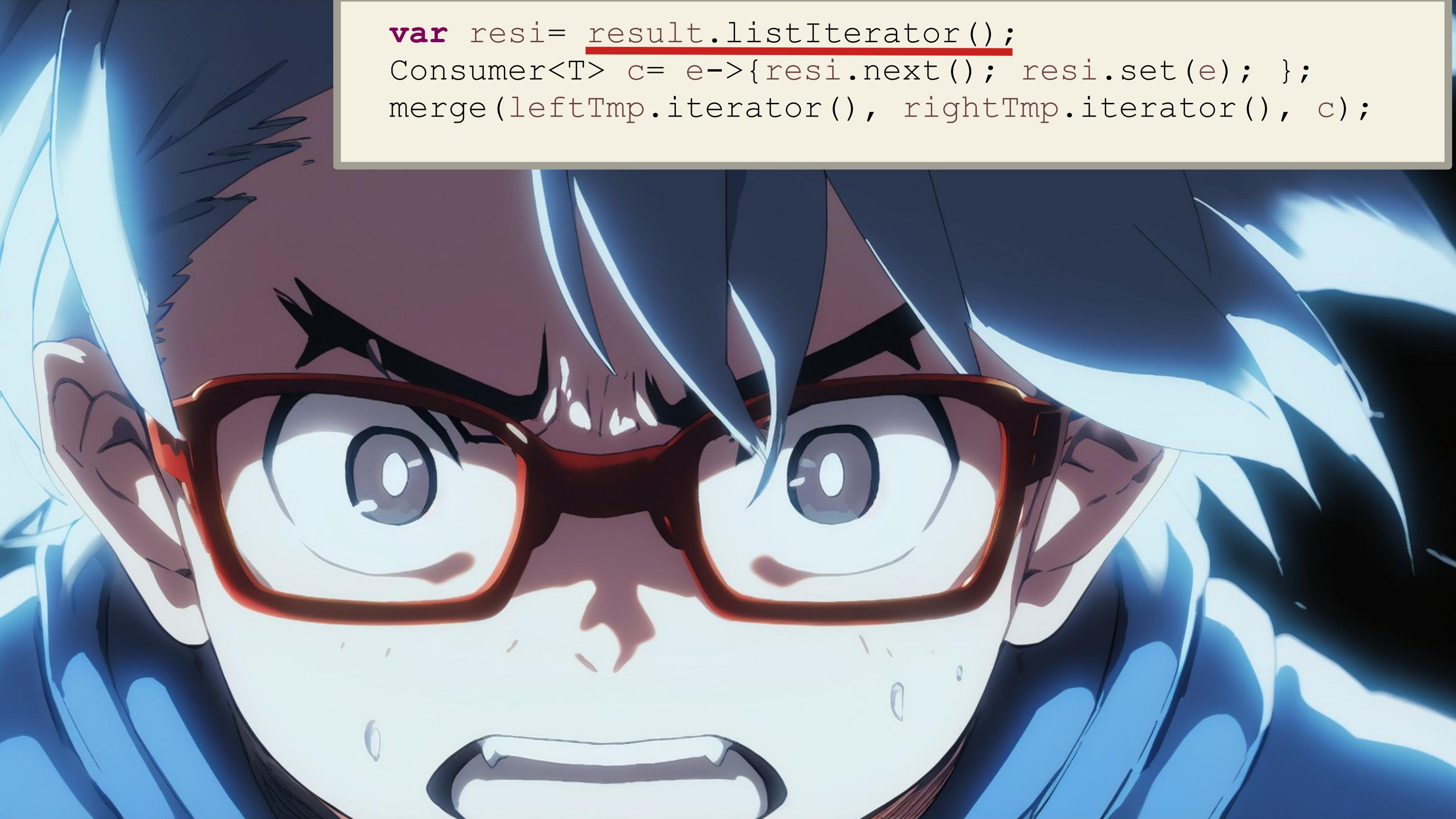
listIterators are richer iterators
working only on lists.
They offer the ‘set’ method.



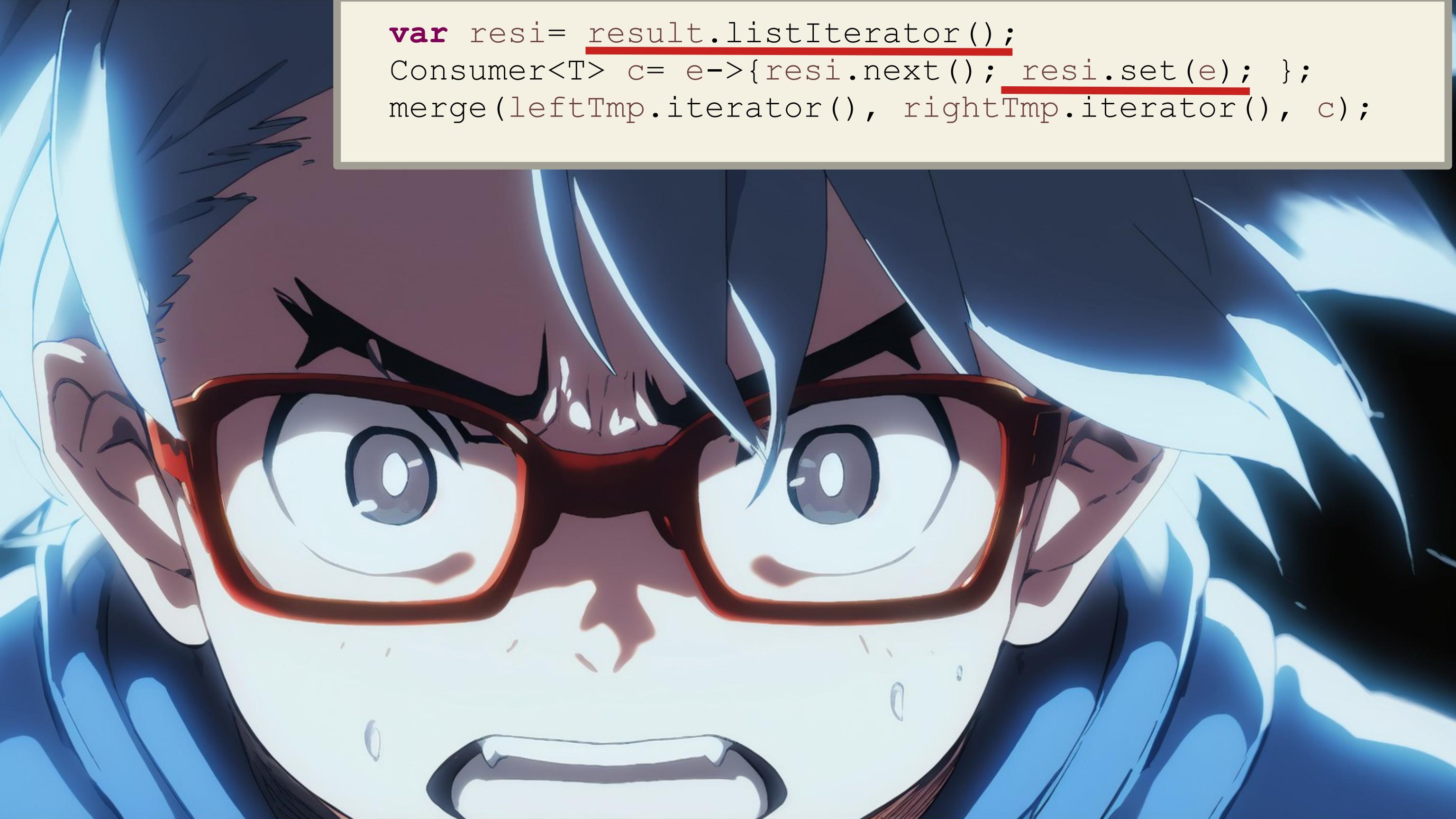
```
var resi= result.listIterator();
Consumer<T> c= e->{resi.next(); resi.set(e); };
merge(leftTmp.iterator(), rightTmp.iterator(), c);
```

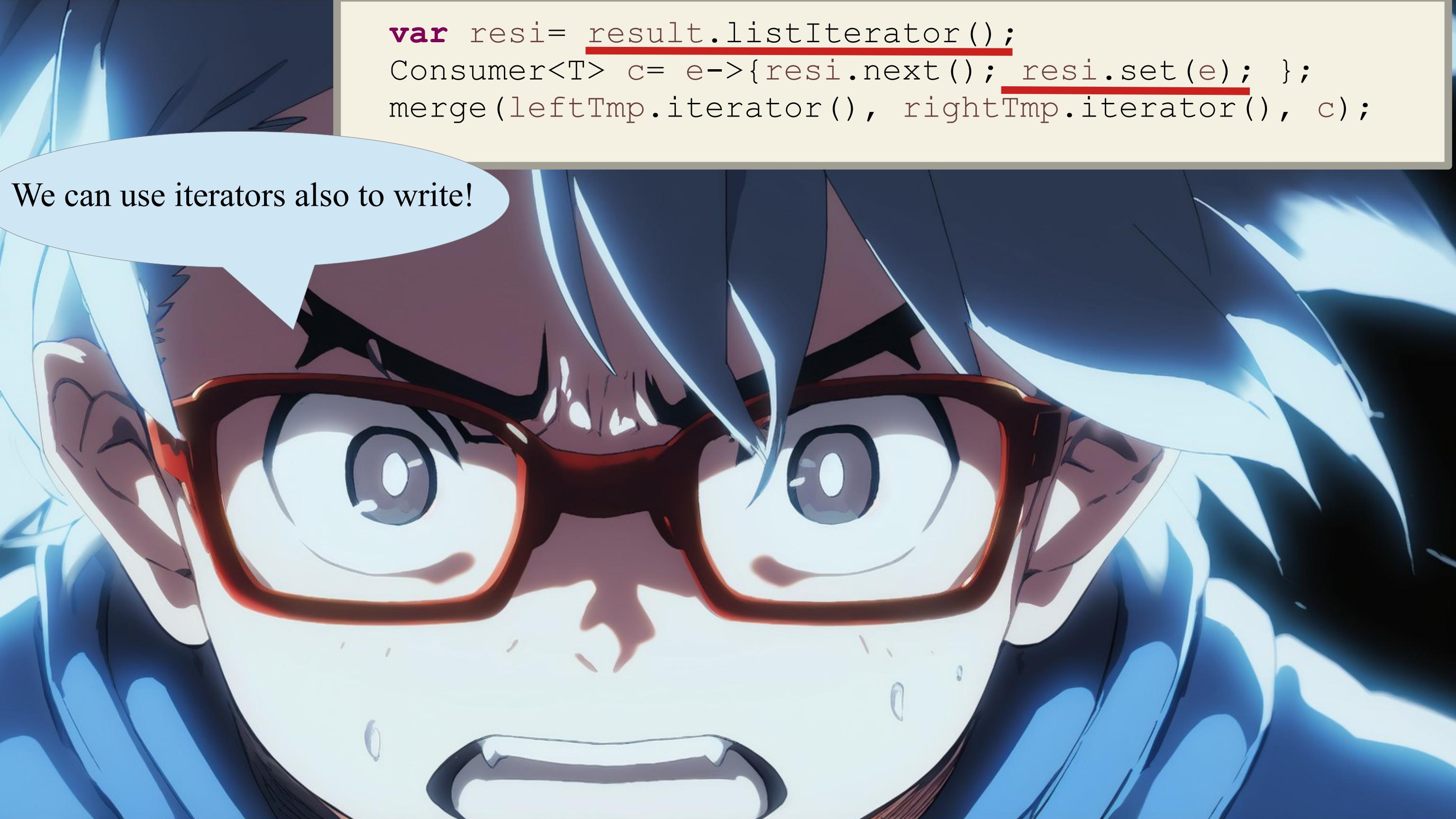


```
var resi= result.listIterator();  
Consumer<T> c= e->{resi.next(); resi.set(e); };  
merge(leftTmp.iterator(), rightTmp.iterator(), c);
```



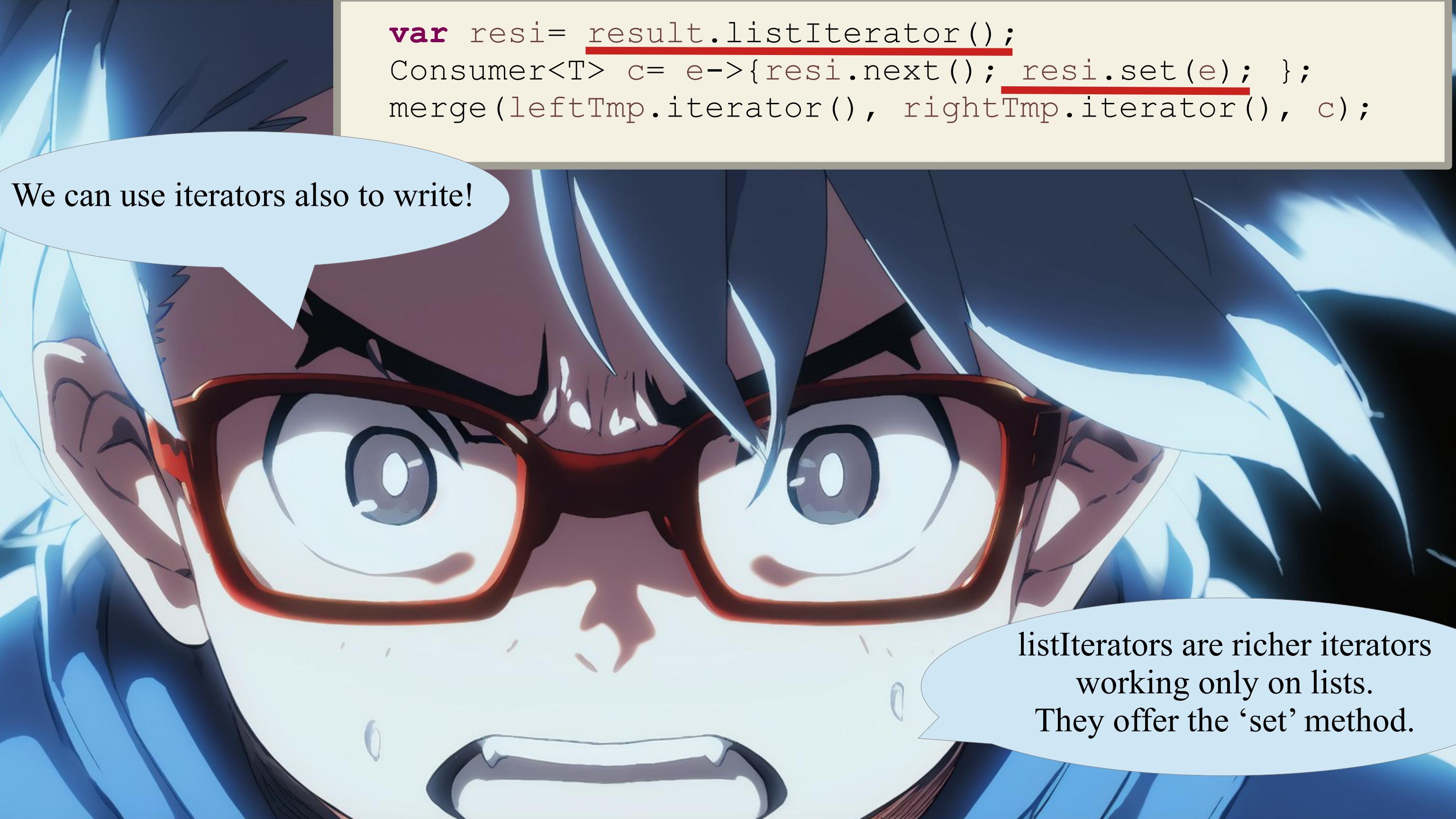
```
var resi= result.listIterator();  
Consumer<T> c= e->{resi.next(); resi.set(e); };  
merge(leftTmp.iterator(), rightTmp.iterator(), c);
```





```
var resi= result.listIterator();  
Consumer<T> c= e->{resi.next(); resi.set(e); };  
merge(leftTmp.iterator(), rightTmp.iterator(), c);
```

We can use iterators also to write!



```
var resi= result.listIterator();  
Consumer<T> c= e->{resi.next(); resi.set(e); };  
merge(leftTmp.iterator(), rightTmp.iterator(), c);
```

We can use iterators also to write!

listIterators are richer iterators
working only on lists.
They offer the ‘set’ method.