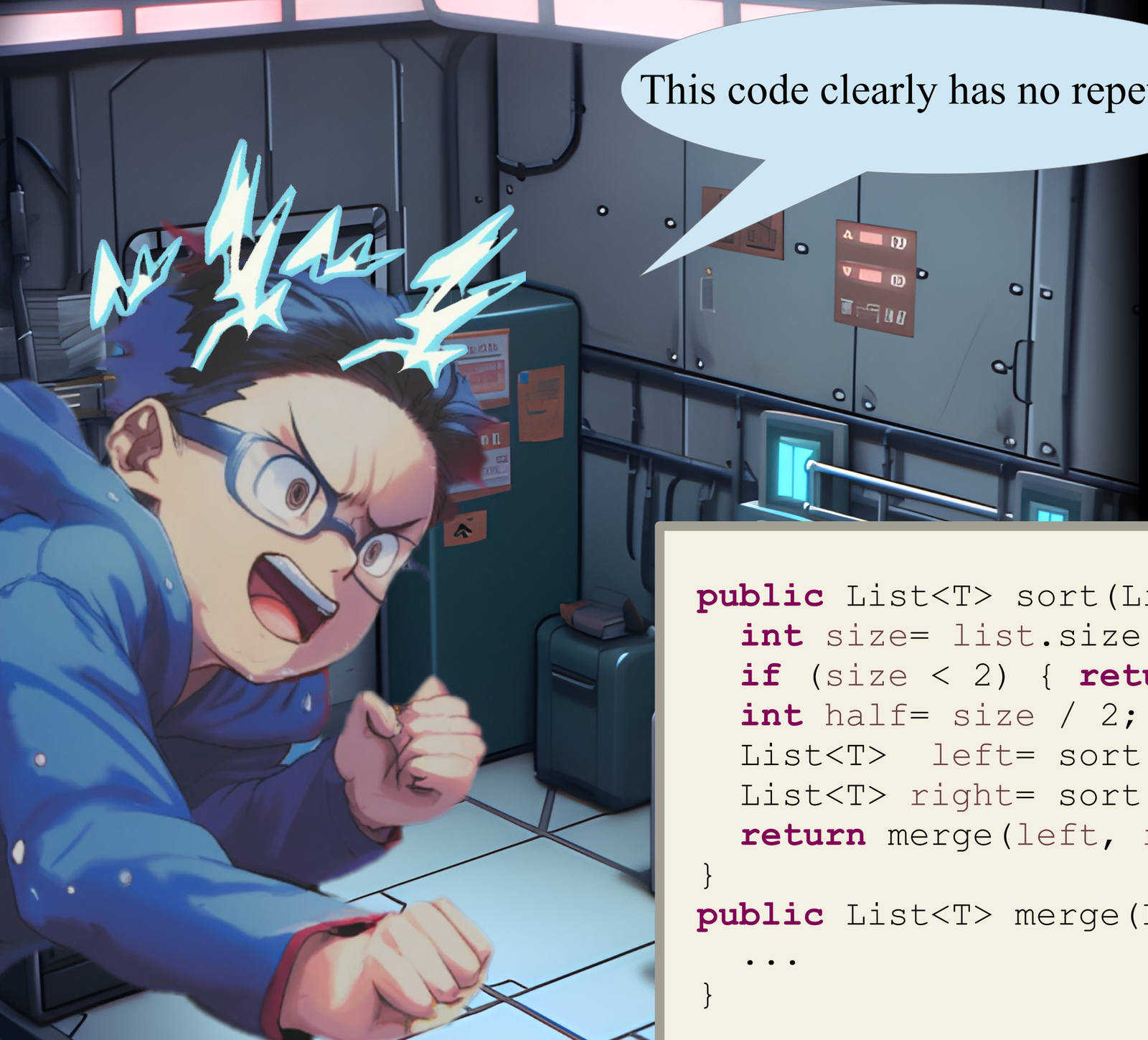

Power UP!

```java
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) {
  ...
}
```




```java
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) {
  ...
}
```


```java
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) {
  ...
}
```


```java
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) {
  ...
}
```

```java
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;
}
```


```java
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;
}
```


```java
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;
}
```


```java
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;
}
```

```java
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;
}
```


```java
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;
}
```


```java
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;
}
```

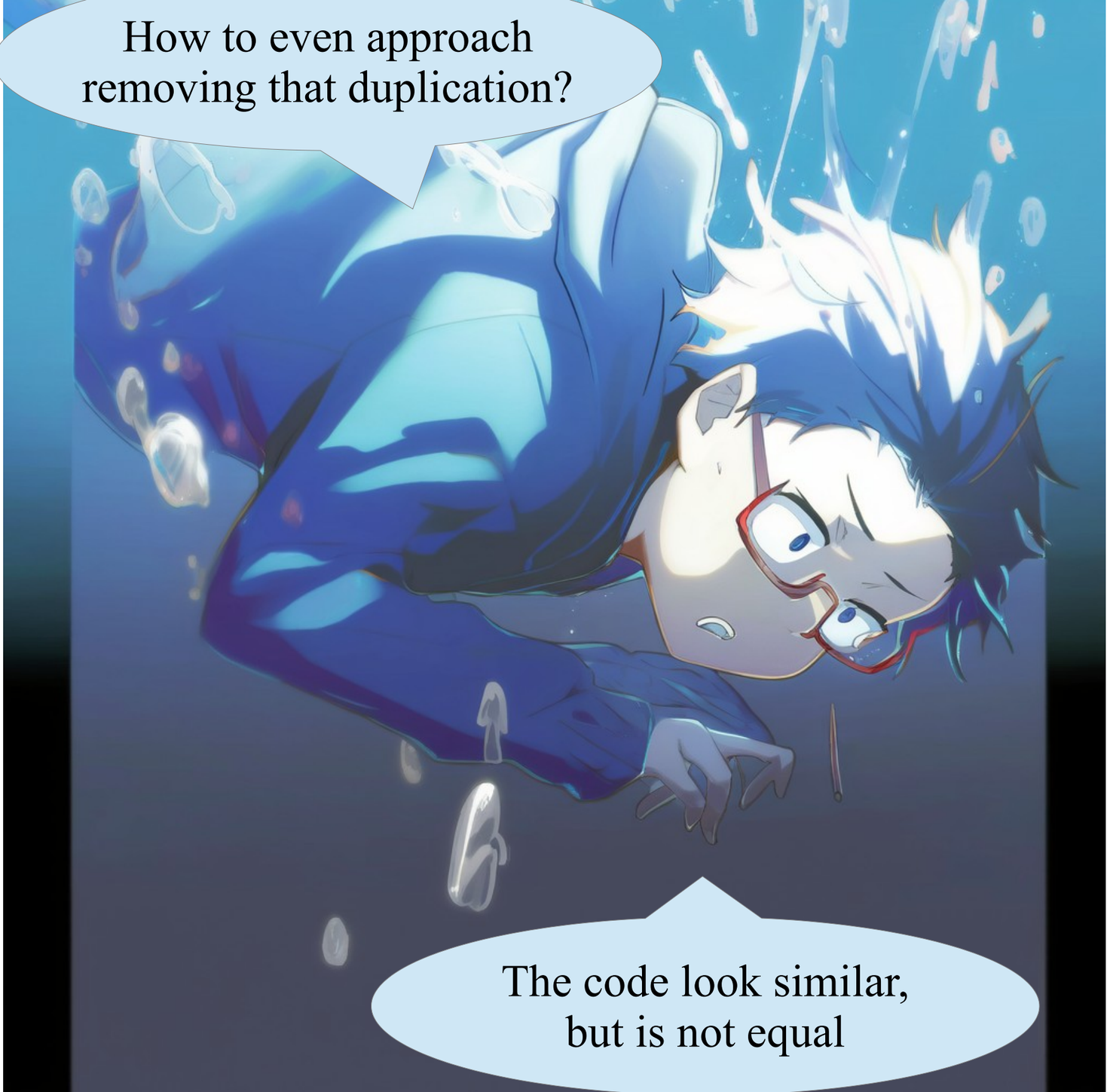

```java
public List<T> merge(List<T> a, List<T> b) {
  List<T> res= new ArrayList<>(a.size() + b.size());
  int indexA= 0;
  int indexB= 0;
  while (indexA < a.size() && indexB < b.size()){
    T currA= a.get(indexA);
    T currB= b.get(indexB);
    if (currA.compareTo(currB) <= 0){
      res.add(currA);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


```java
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;
}
```

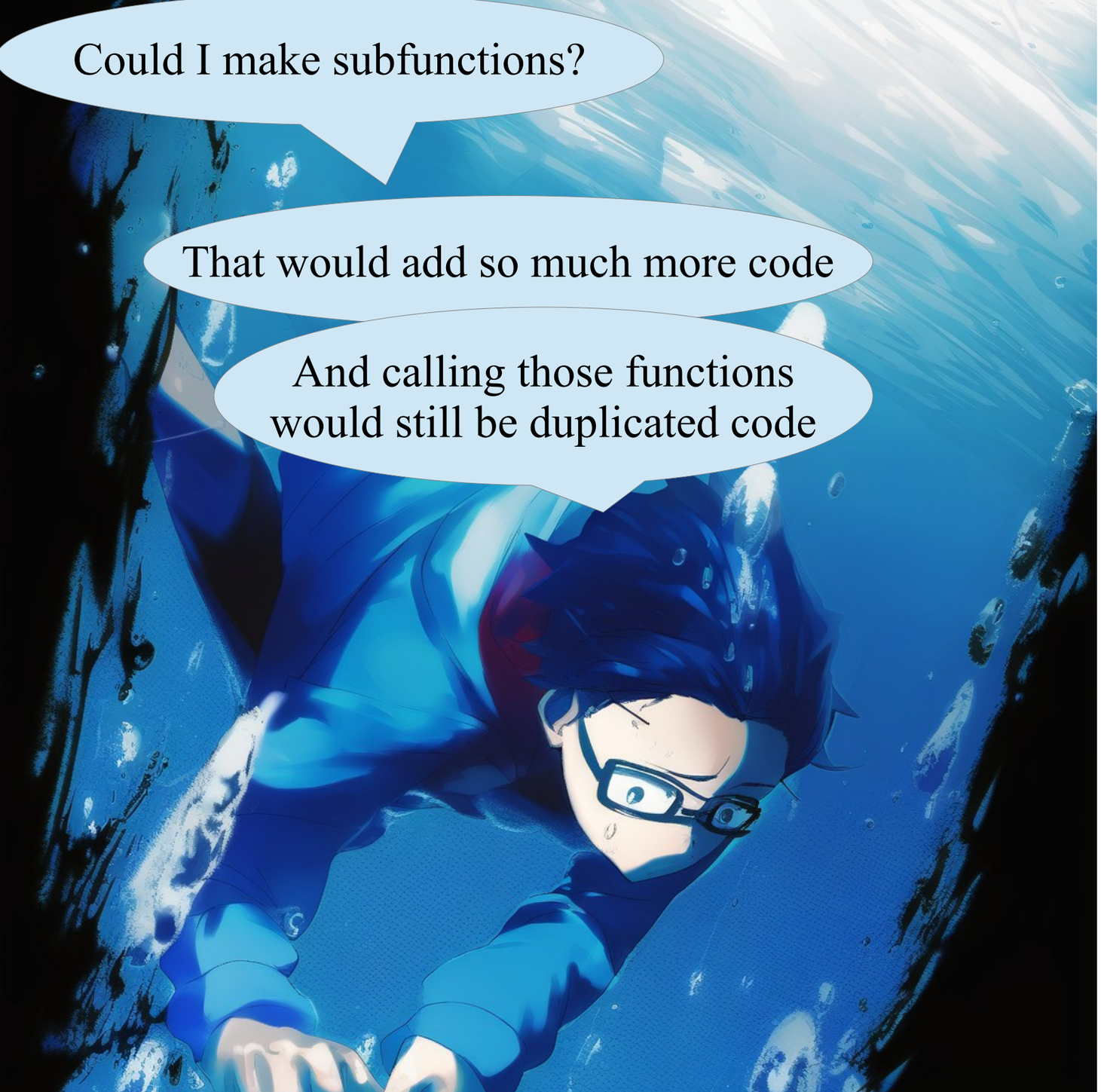

```java
public List<T> merge(List<T> a, List<T> b) {
  List<T> res= new ArrayList<>(a.size() + b.size());
  int indexA= 0;
  int indexB= 0;
  while (indexA < a.size() && indexB < b.size()){
    T currA= a.get(indexA);
    T currB= b.get(indexB);
    if (currA.compareTo(currB) <= 0){
      res.add(currA);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


```java
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;
}
```


```java
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;
}
```


How to even approach removing that duplication?
The code look similar, but is not equal


Diving deeper into the problem
One version is on list a,
one version is on list b.


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


ゾクッ

```java
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;
}
```


```java
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;
}
```


```java
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;
}
```


```java
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;
}
```

```java
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;
}
```


```java
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;
}
```


```java
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;
}
```

```java
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;
}
```


```java
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;
}
```


```java
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;
}
```


```java
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;
}
```

```java
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;
}
```


```java
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;
}
```


```java
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;
}
```


```java
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;
}
```

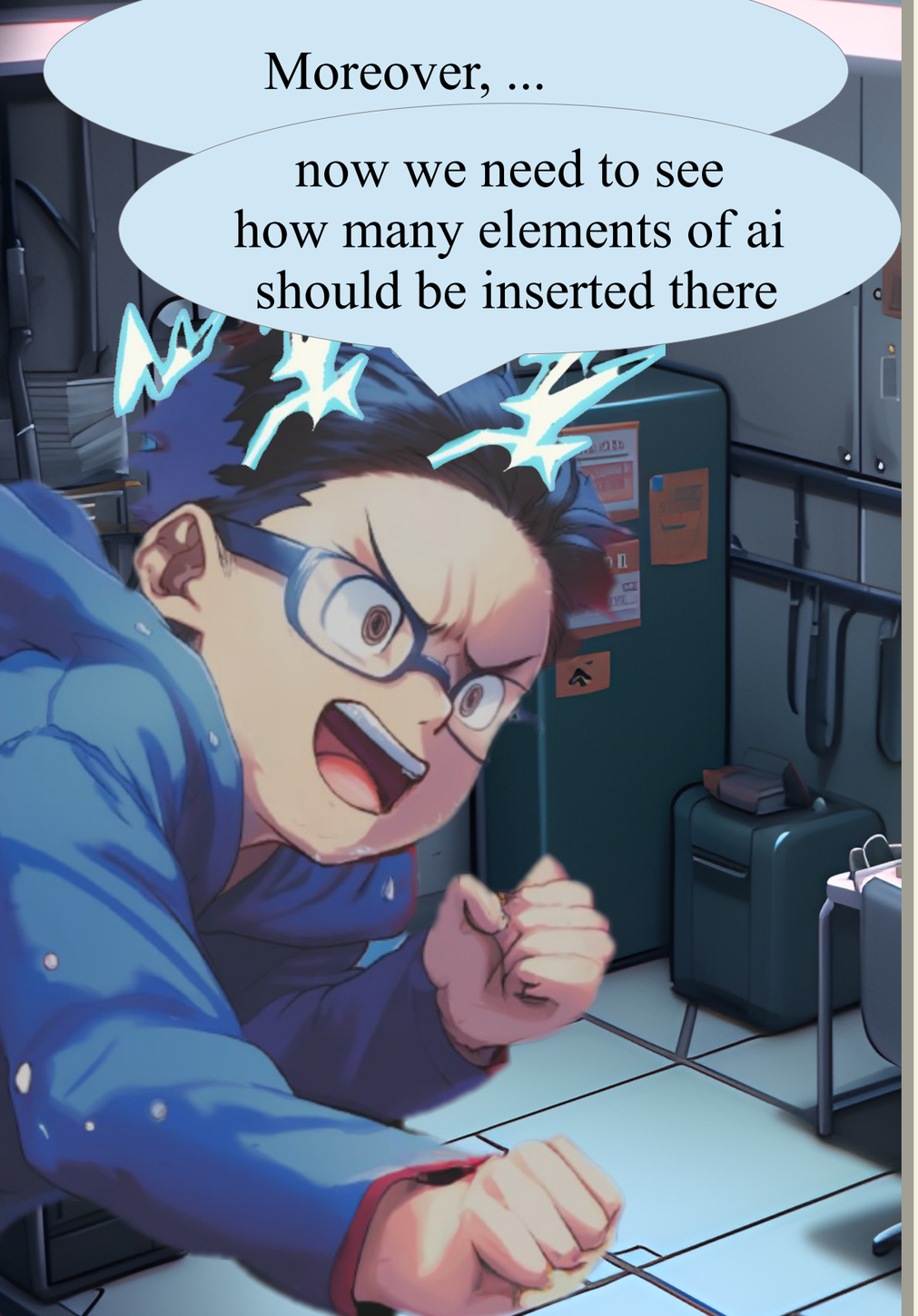
```java
public List<T> merge(List<T> a, List<T> b) {
  List<T>res= new ArrayList<>(a.size() + b.size());
  Iterator<T> ai= a.iterator();
  Iterator<T> bi= b.iterator();
  T currA= ai.next();//one element pending
  while(bi.hasNext()) {
    T currB= bi.next();//two elements pending
    if(currB.compareTo(currA) <= 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;
}
```


```java
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;
}
```

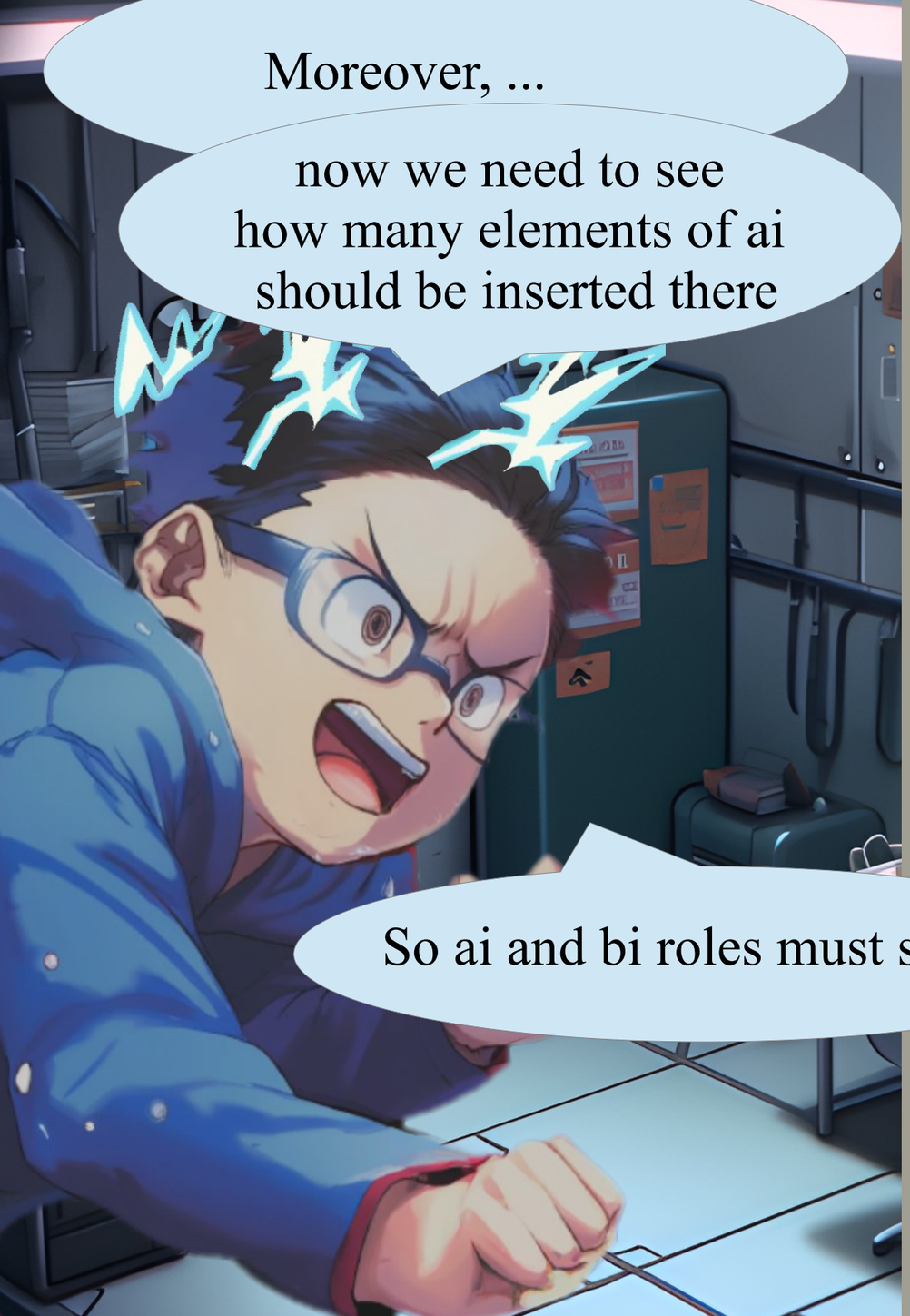

```java
public List<T> merge(List<T> a, List<T> b) {
  List<T>res= new ArrayList<>(a.size() + b.size());
  Iterator<T> ai= a.iterator();
  Iterator<T> bi= b.iterator();
  T currA= ai.next();//one element pending
  while(bi.hasNext()) {
    T currB= bi.next();//two elements pending
    if(currB.compareTo(currA) <= 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;
}
```

```java
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;
}
```

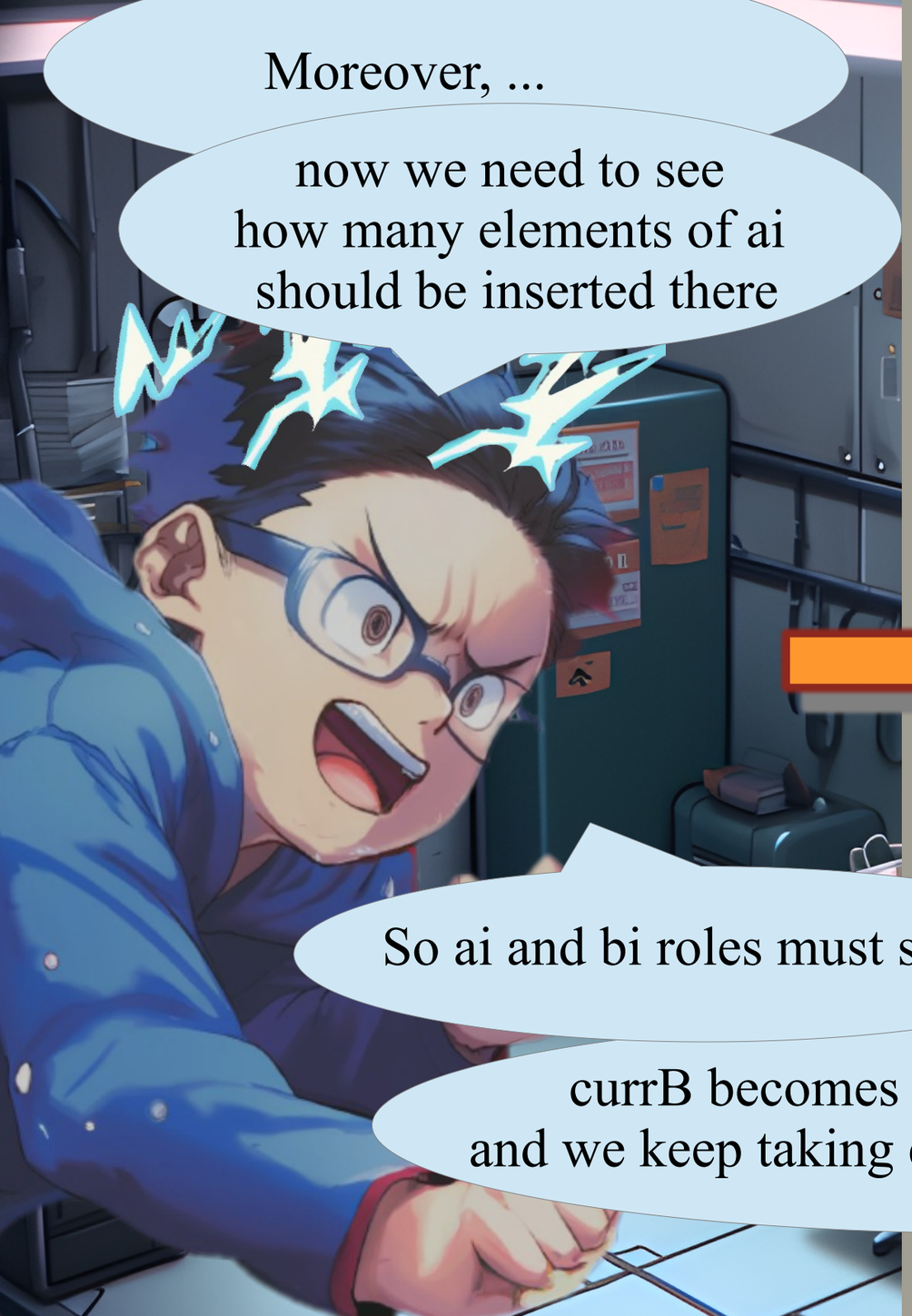

```java
public List<T> merge(List<T> a, List<T> b) {
  List<T>res= new ArrayList<>(a.size() + b.size());
  Iterator<T> ai= a.iterator();
  Iterator<T> bi= b.iterator();
  T currA= ai.next();//one element pending
  while(bi.hasNext()) {
    T currB= bi.next();//two elements pending
    if(currB.compareTo(currA) <= 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;
}
```




```java
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;
}
```




```java
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;
```




```java
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;
```

```java
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;
}
```

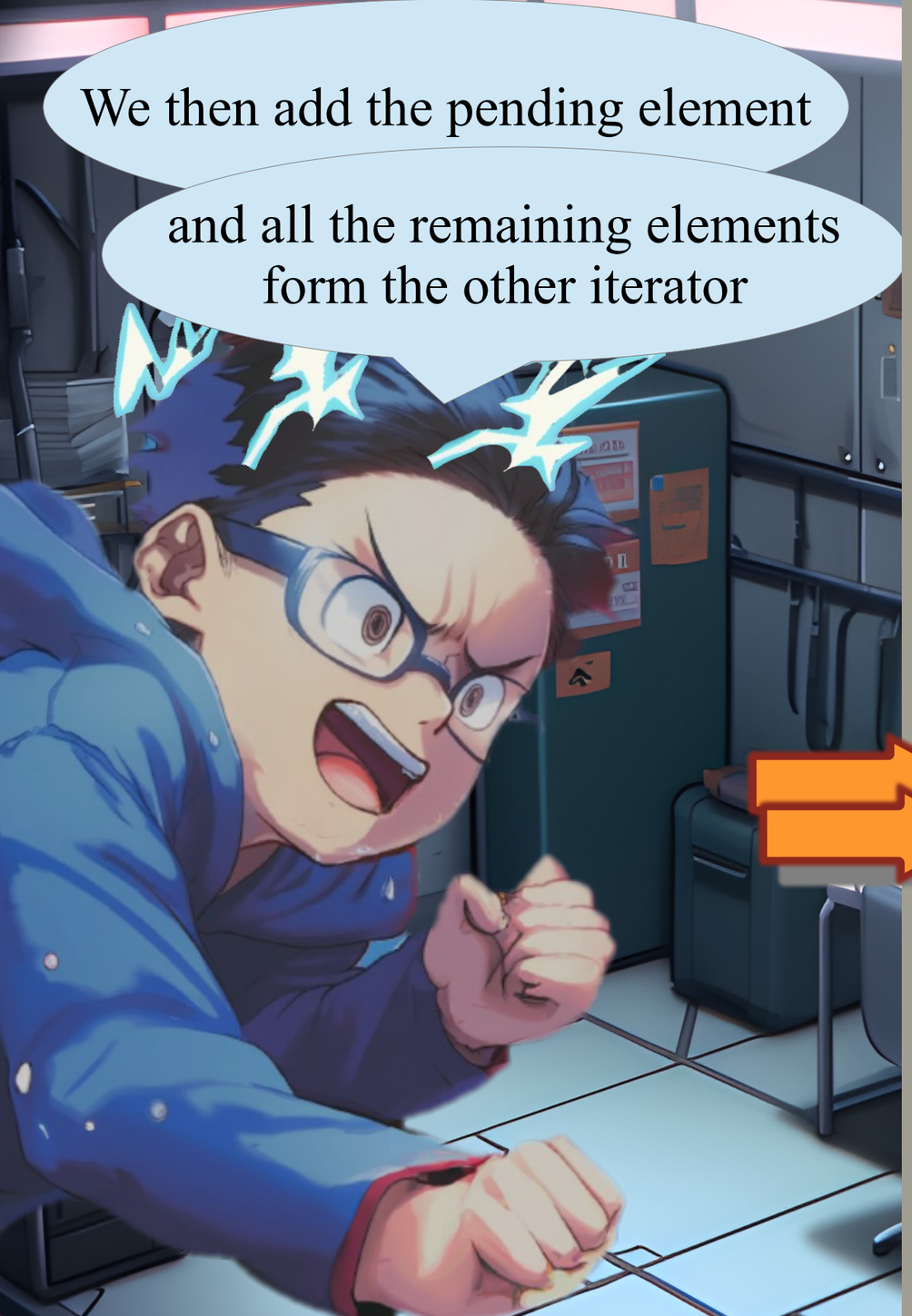

```java
public List<T> merge(List<T> a, List<T> b) {
  List<T>res= new ArrayList<>(a.size() + b.size());
  Iterator<T> ai= a.iterator();
  Iterator<T> bi= b.iterator();
  T currA= ai.next();//one element pending
  while(bi.hasNext()) {
    T currB= bi.next();//two elements pending
    if(currB.compareTo(currA) <= 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;
}
```


```java
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;
}
```


```java
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;
}
```

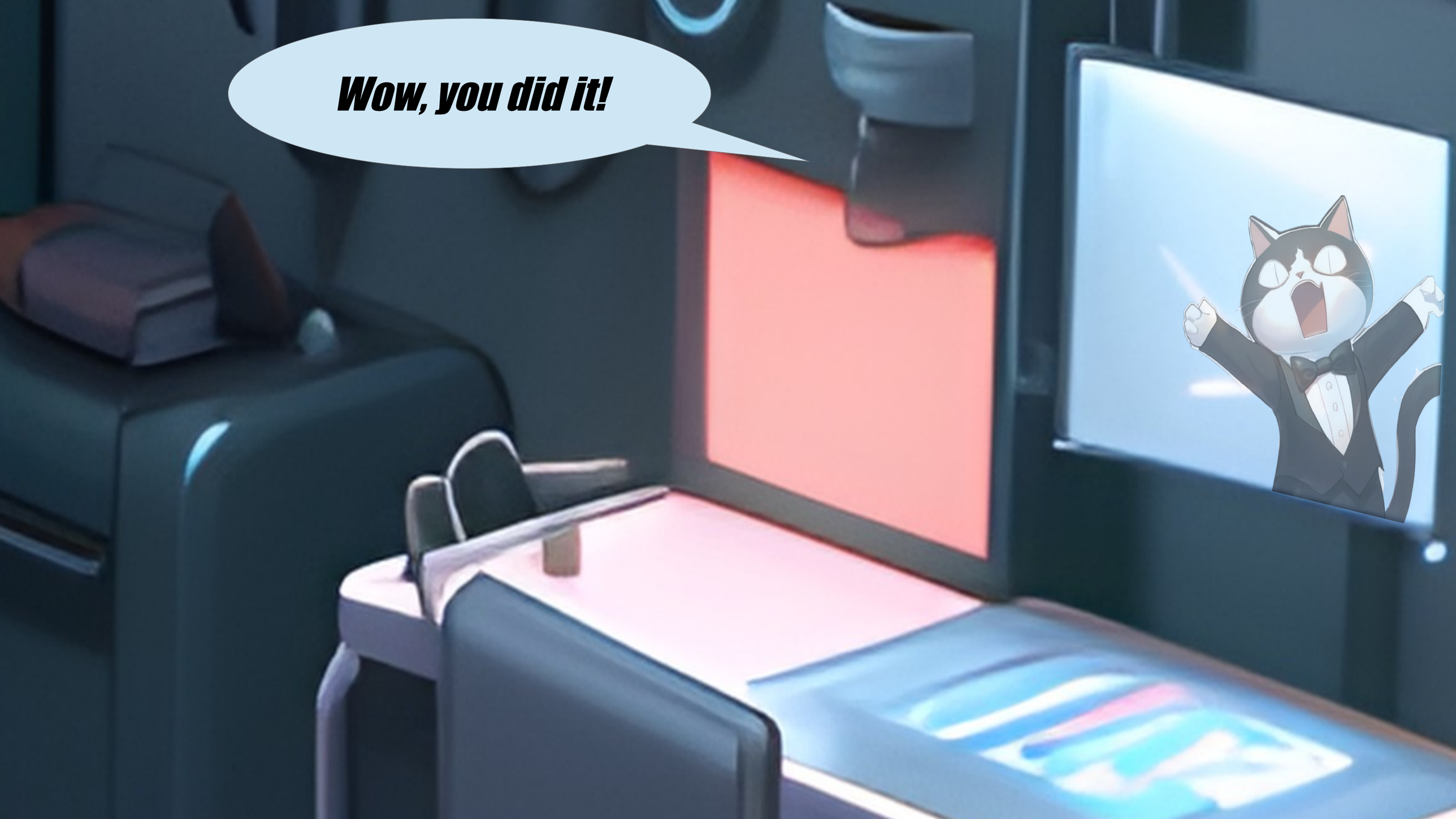
```java
public List<T> merge(List<T> a, List<T> b) {
  List<T>res= new ArrayList<>(a.size() + b.size());
  Iterator<T> ai= a.iterator();
  Iterator<T> bi= b.iterator();
  T currA= ai.next();//one element pending
  while(bi.hasNext()) {
    T currB= bi.next();//two elements pending
    if(currB.compareTo(currA) <= 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;
}
```

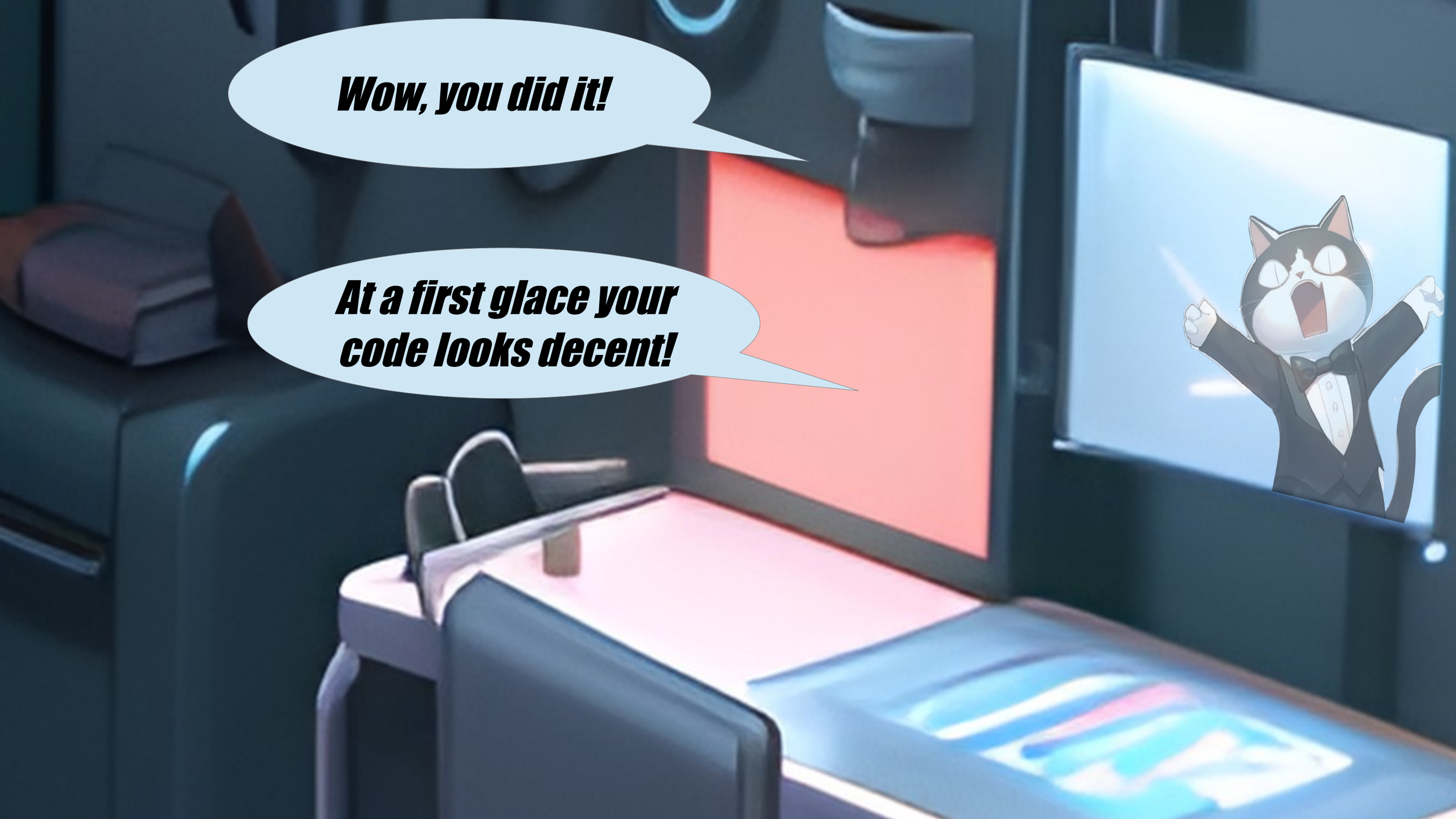

```java
public List<T> merge(List<T> a, List<T> b) {
  List<T>res= new ArrayList<>(a.size() + b.size());
  Iterator<T> ai= a.iterator();
  Iterator<T> bi= b.iterator();
  T currA= ai.next();//one element pending
  while(bi.hasNext()) {
    T currB= bi.next();//two elements pending
    if(currB.compareTo(currA) <= 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;
}
```



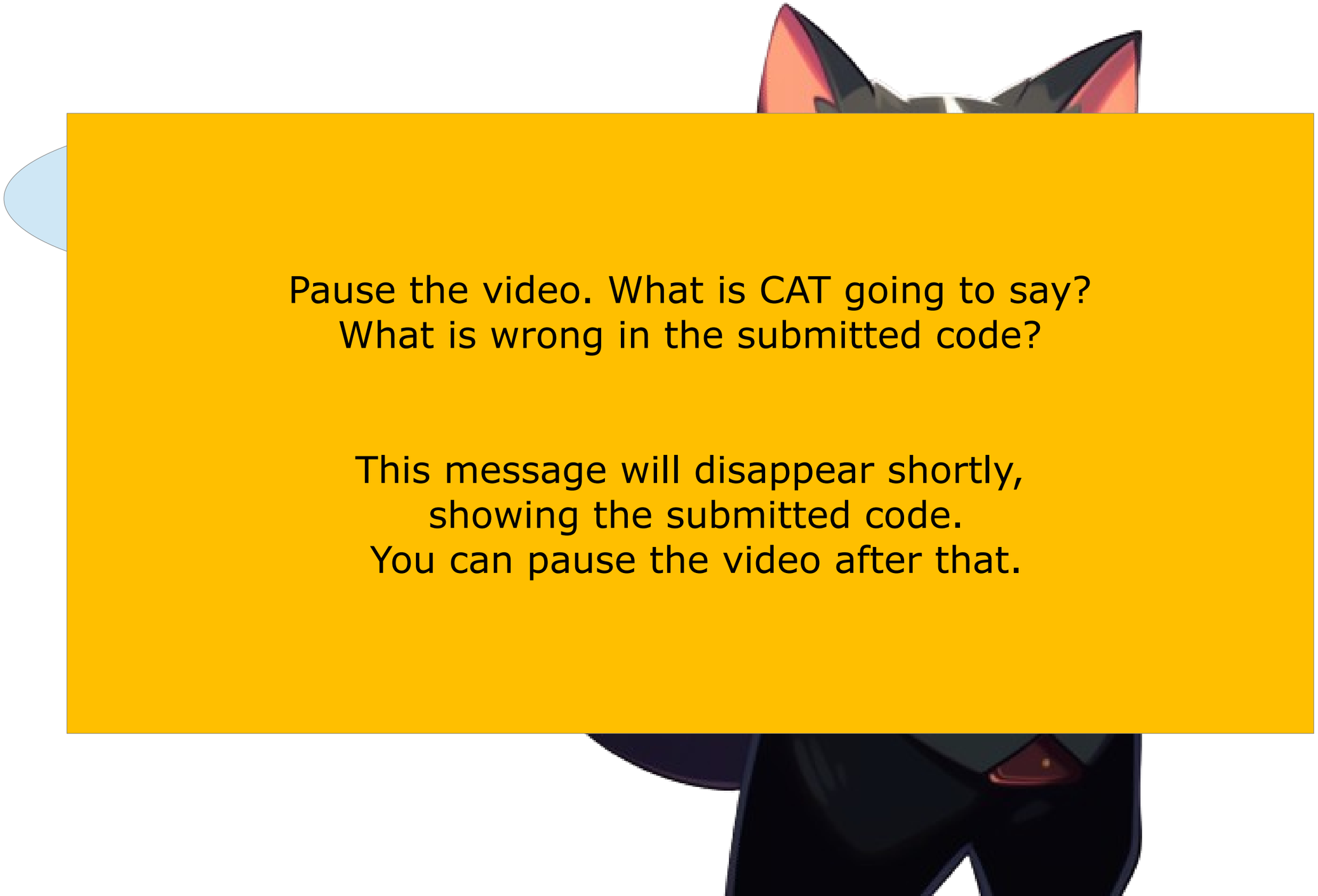

```java
public List<T> merge(List<T> a, List<T> b) {
  List<T>res= new ArrayList<>(a.size() + b.size());
  Iterator<T> ai= a.iterator();
  Iterator<T> bi= b.iterator();
  T currA= ai.next();//one element pending
  while(bi.hasNext()) {
    T currB= bi.next();//two elements pending
    if(currB.compareTo(currA) <= 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;
}
```


```java
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!


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.

```java
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!

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


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?

```java
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;
}
```


```java
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;
}
```


```java
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;
}
```

```java
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;
}
```


```java
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;
```

```java
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!

```java
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;
}
```


```java
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;
}
```

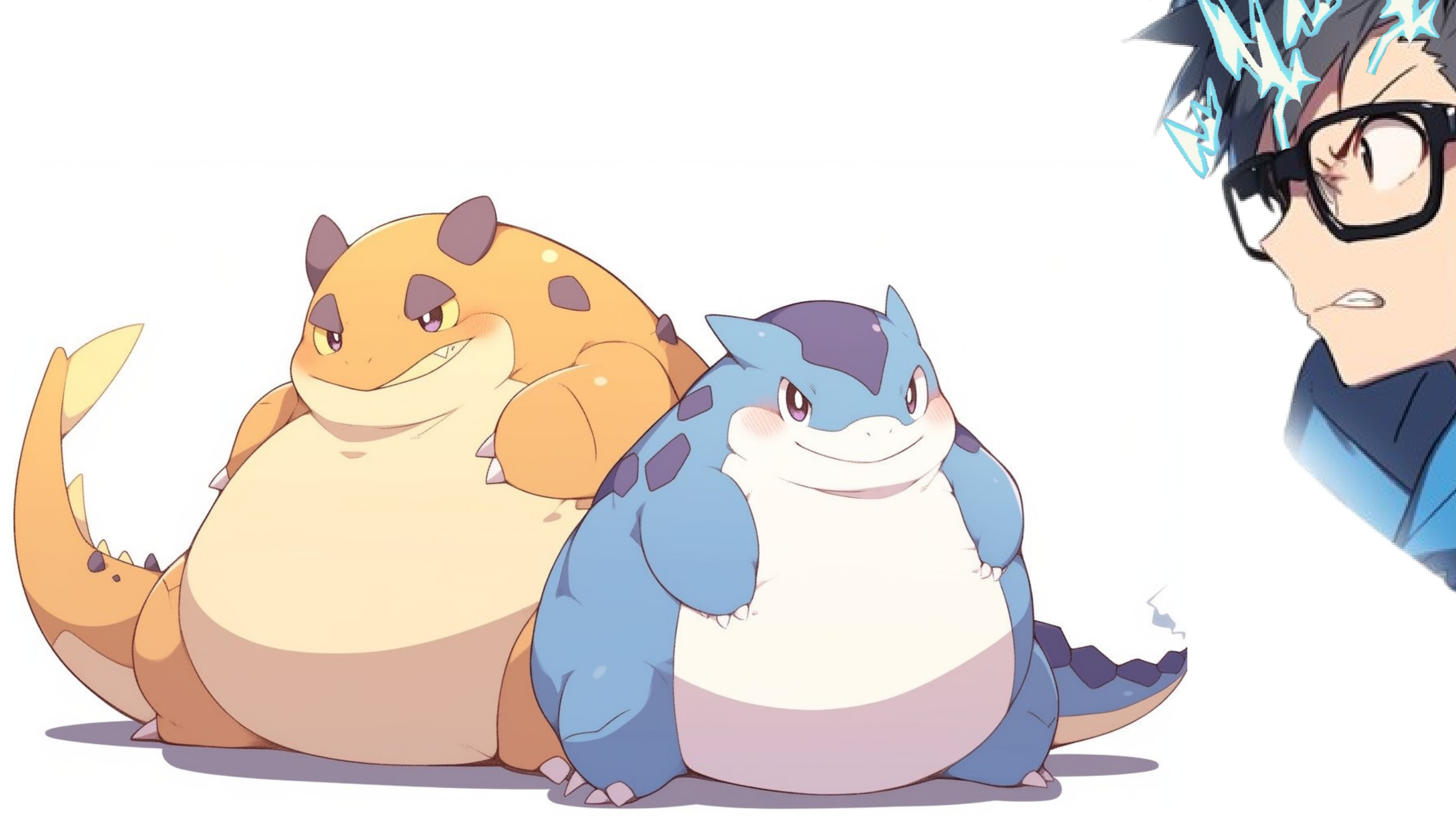
```
public List<T> merge(List<T> a, List<T> b) {
  List<T>res= new ArrayList<>(a.size() + b.size());
  Iterator<T> ai= a.iterator();
  Iterator<T> bi= b.iterator();
  T currA= ai.next();//one element pending
  while(bi.hasNext()) {
    T currB= bi.next();//two elements pending
    if(currB.compareTo(currA)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

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!


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.

```java
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){
  ...
}
```

```java
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

```java
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

```java
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

```java
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){
  ...
}
```

```java
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

```java
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.

```java
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

```java
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);
}
```


```java
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);
}
```

```java
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);
}
```


```java
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);
}
```

```java
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);
}
```

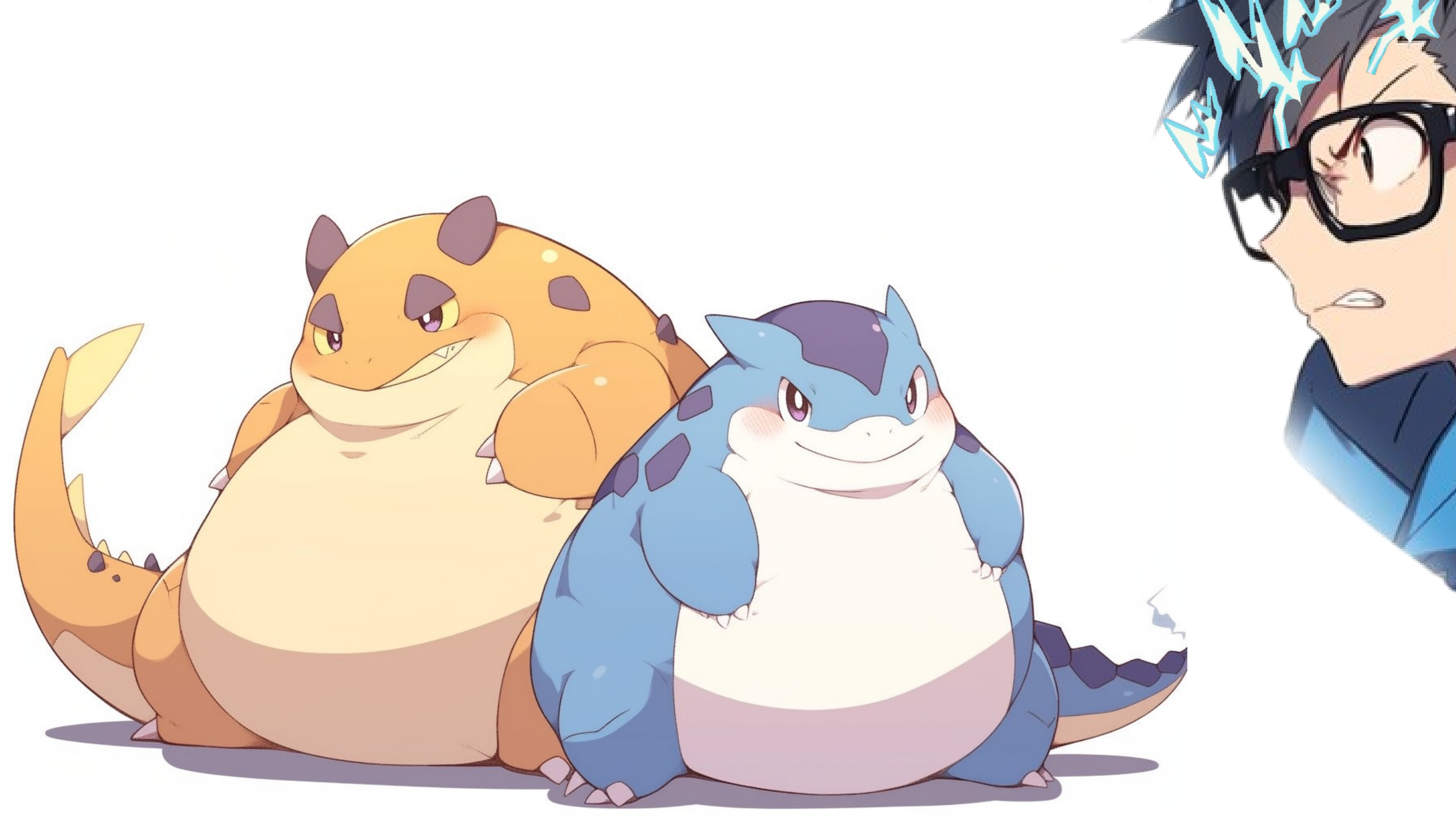
Then we split both the input and the result in two parts using subList

```java
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);
}
```

```java
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);
}
```


```java
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);
    erator<T> tmp = ai; ai = bi; bi = tmp;
    urrentA = currentB;
  }
  res.accept(currentA);
  ai.forEachRemaining(res);
}
```


IT IS
SWAP-RECURSION
TIME!



tmp


tmp
result


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


We receive ‘tmp’ and ‘result’
and we need to write in ‘result’
tmp
result
At the start, ‘result’
contains the data to sort


We receive ‘tmp’ and ‘result’
and we need to write in ‘result’
tmp
result
At the start, ‘result’
contains the data to sort
At the end, ‘result’
must contains the sorted data


We receive ‘tmp’ and ‘result’
and we need to write in ‘result’
tmp
result
At the start, ‘result’
contains the data to sort
At the end, ‘result’
must contains the sorted data
Crucially, we have the permission to use
both tmp and result as scribble space


tmp
result


tmp
result


tmp
result
sortHelper(,) sortHelper(,)

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

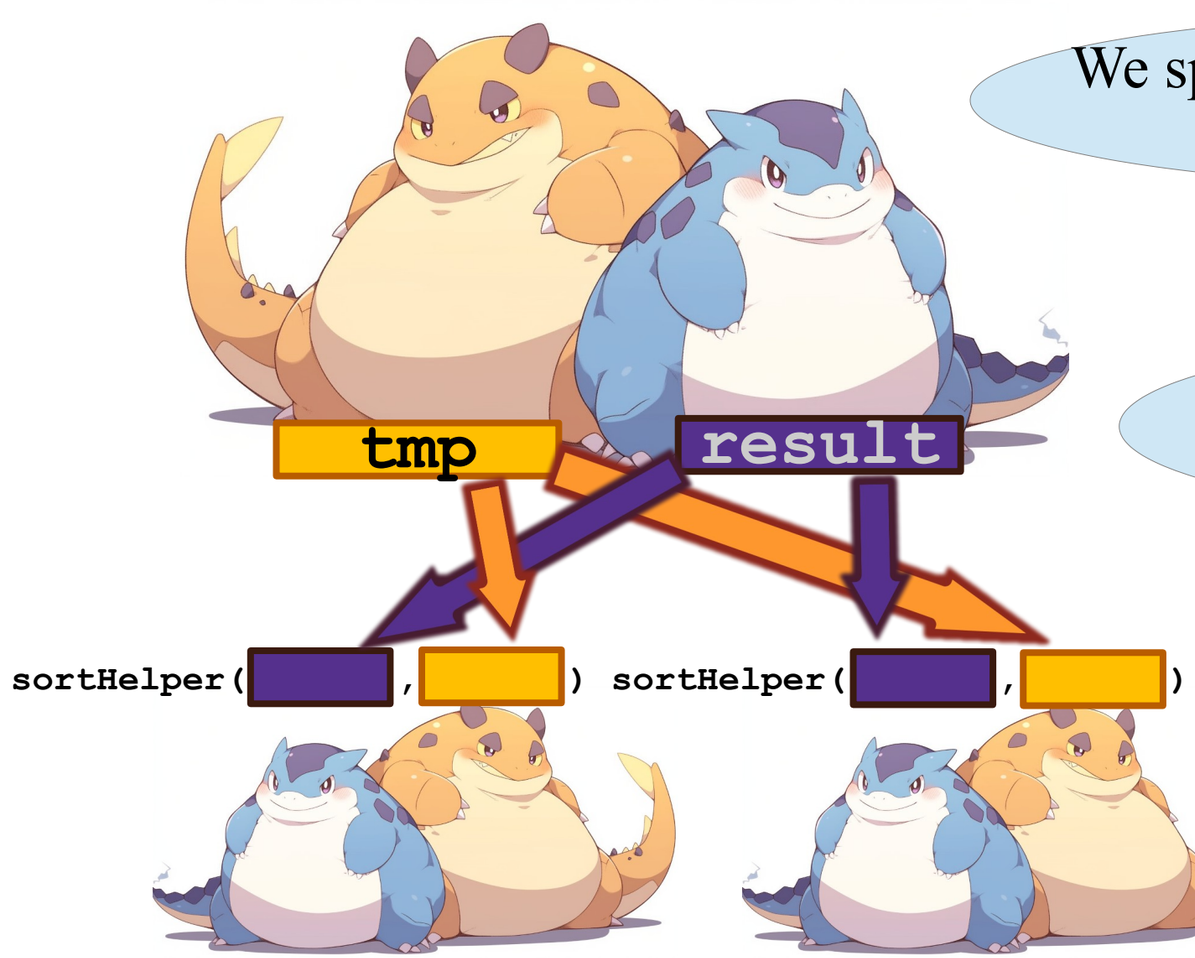
We split ‘tmp’ and ‘result’ in two, so we get 4 sublists

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.

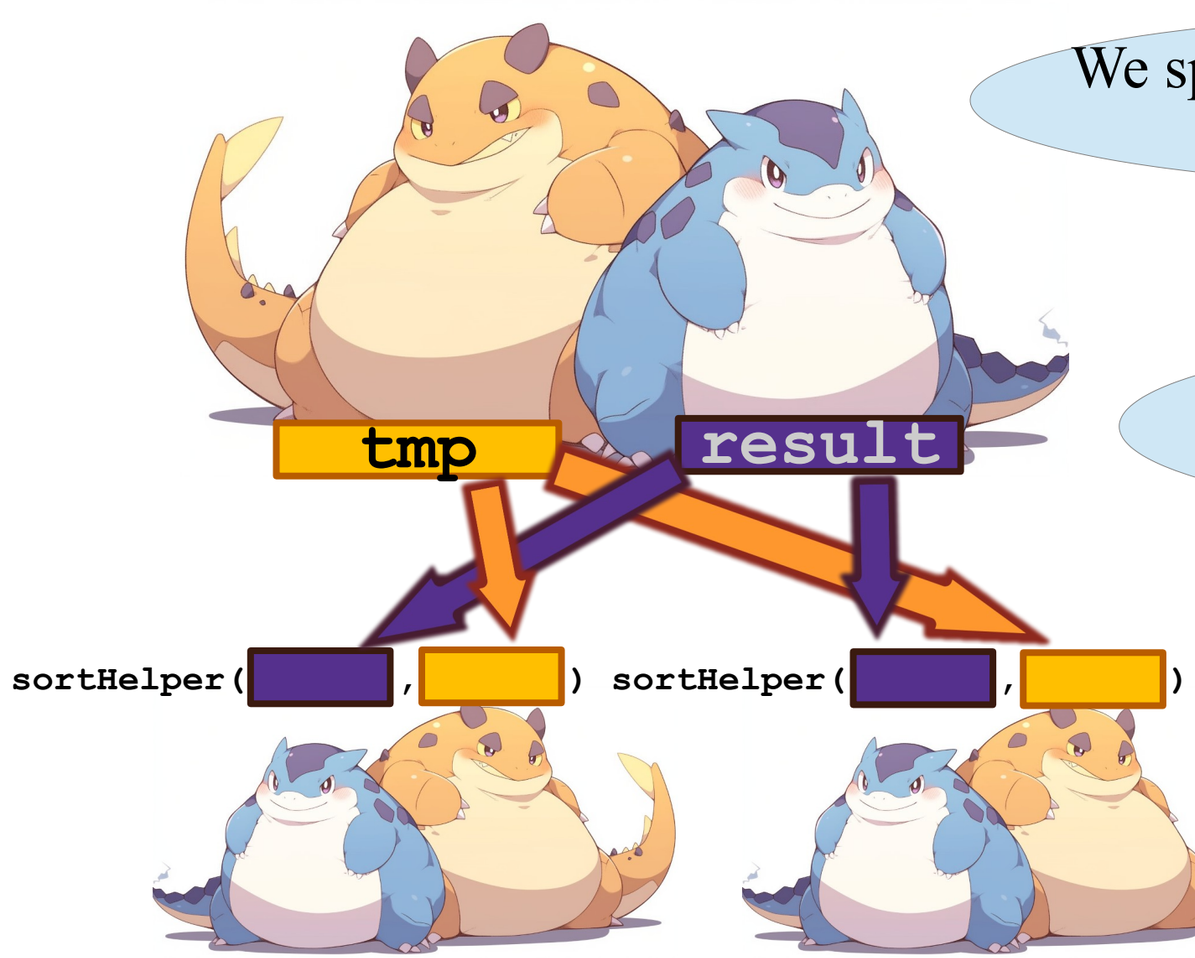


We split ‘tmp’ and ‘result’ in two,
so we get 4 sublists

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’



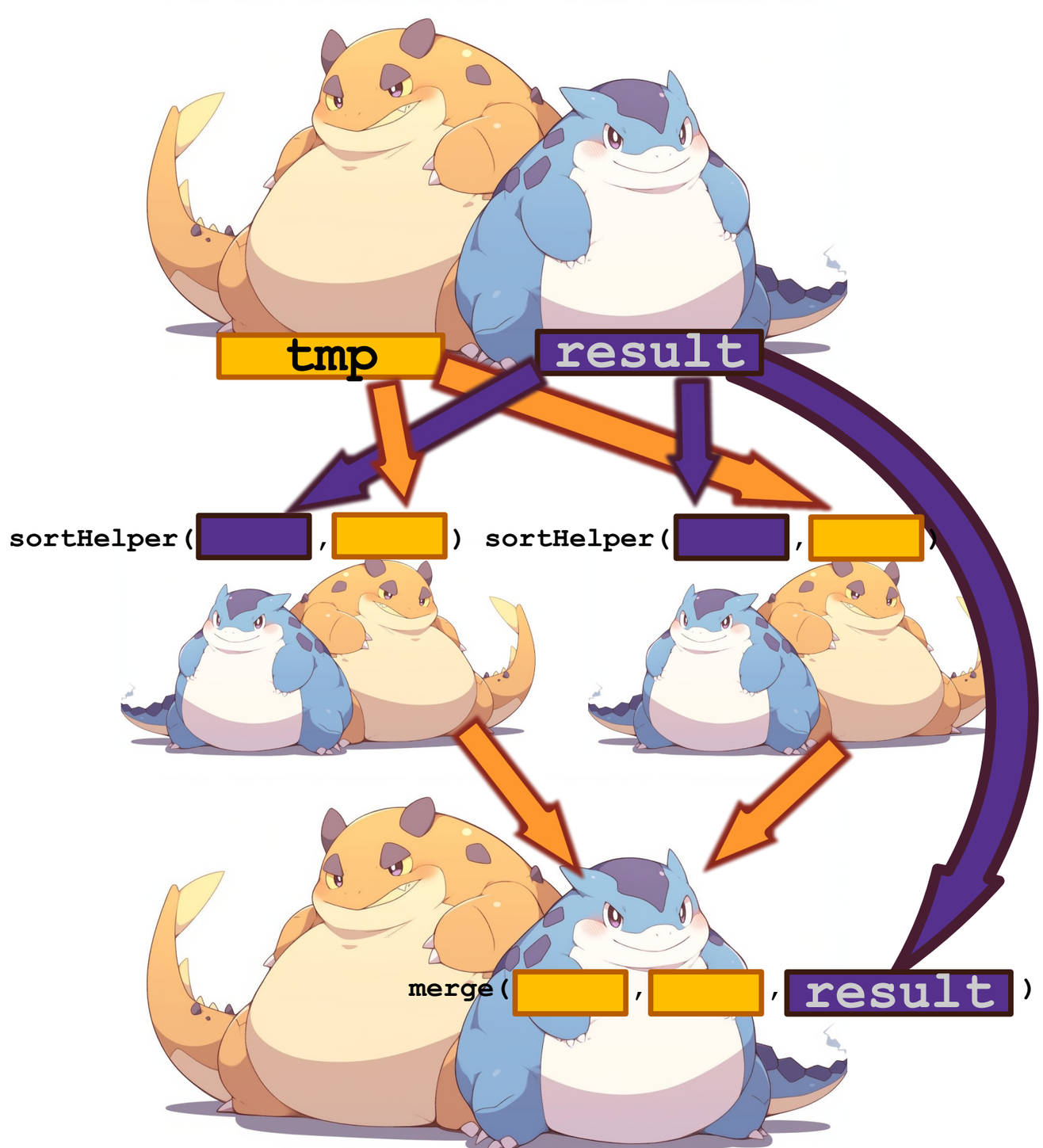
We split ‘tmp’ and ‘result’ in two,
so we get 4 sublists

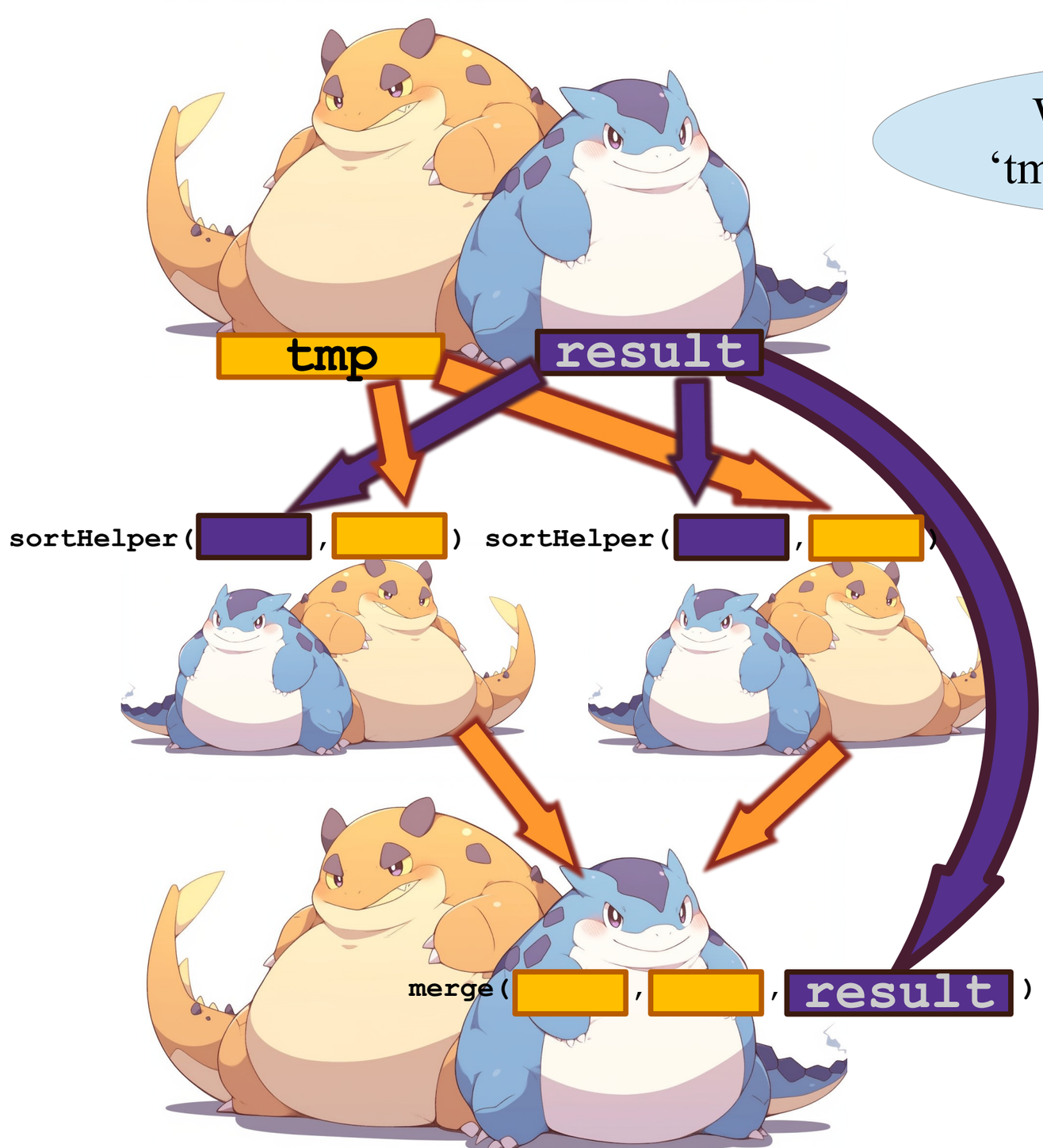
We pass ‘tmp’ and ‘result’ to
the recursion, but swapped!

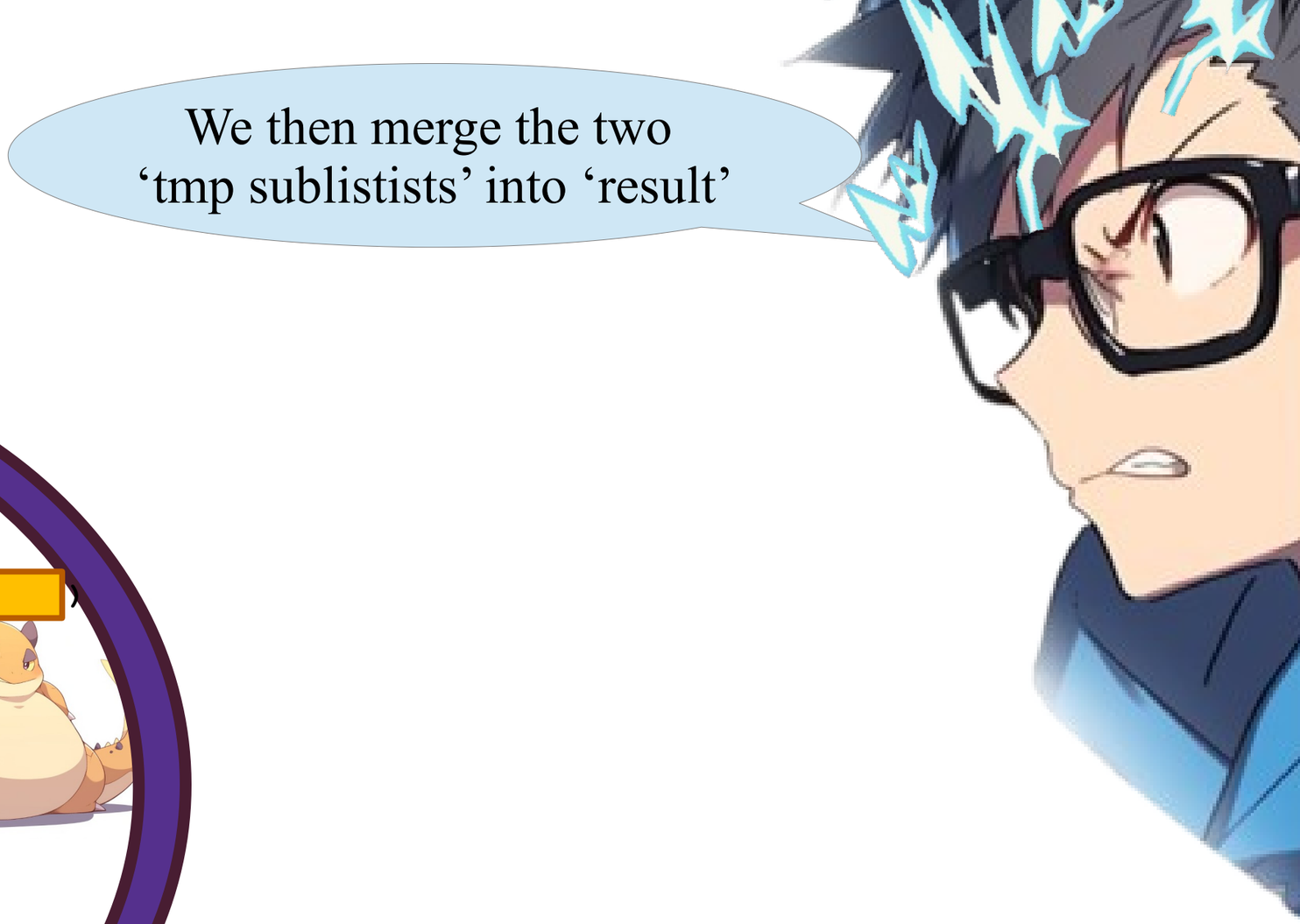
After recursion, ‘result’ is scribbled over.

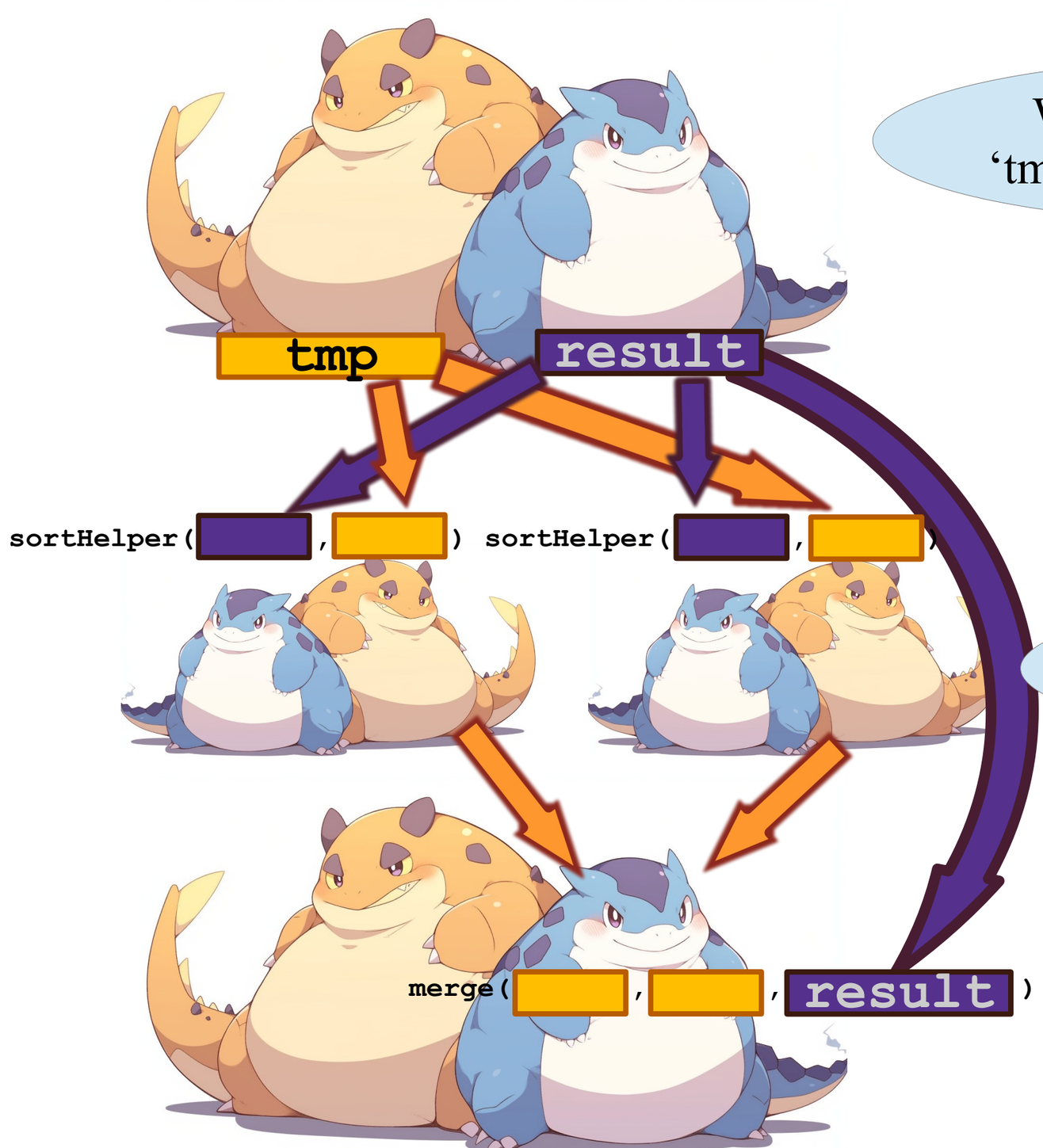
After recursion, the yellow dragon ‘tmp’
is holding semi sorted data
from the blue dragon ‘result’

That is, the two sublists
composing ‘tmp’ are now sorted


tmp
result
sortHelper(,) sortHelper(,)
merge(, , result)

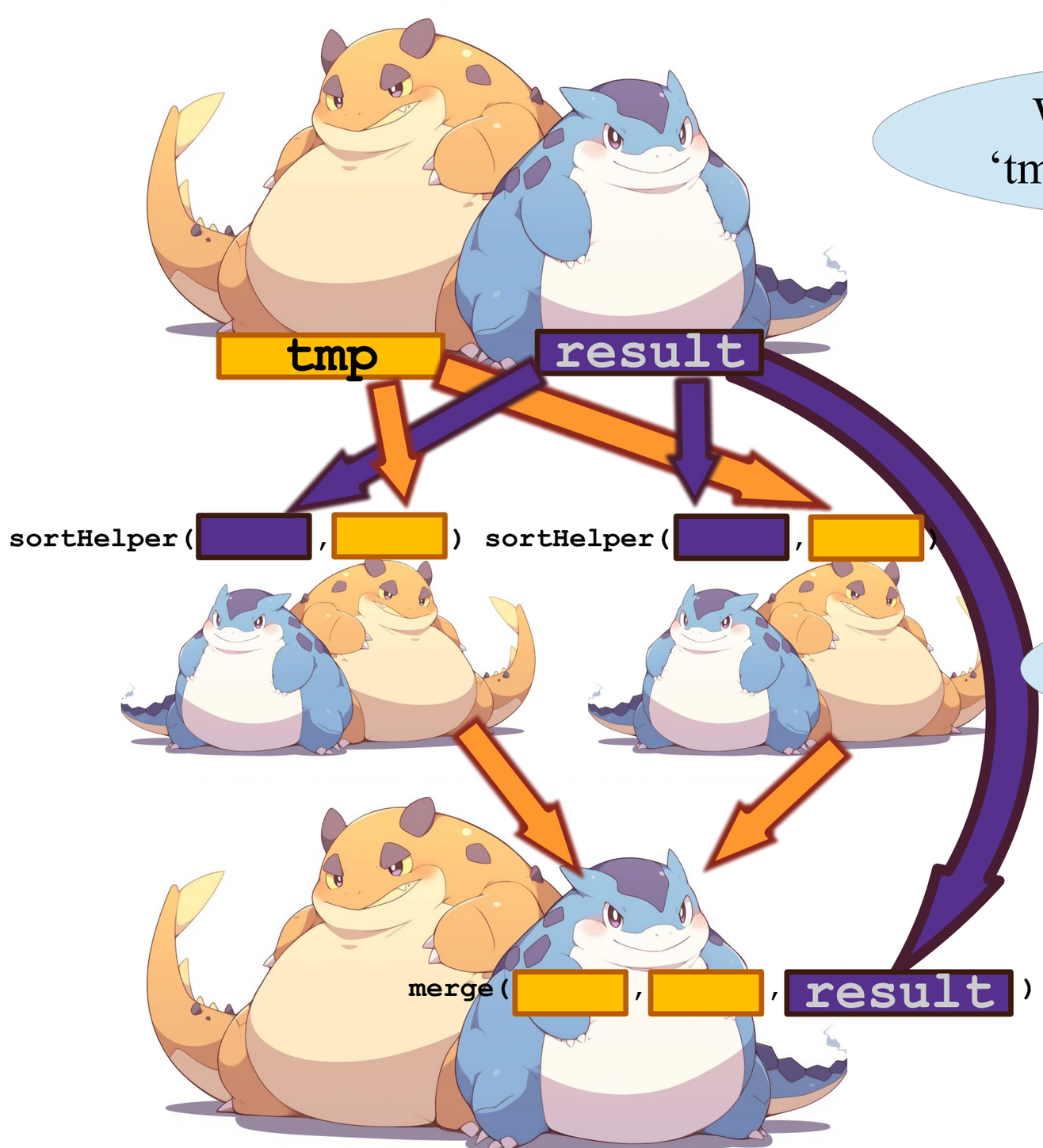

tmp
result
sortHelper(,) sortHelper(,)
merge(, , result)


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



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

At the end ‘tmp’ is split sorted ...

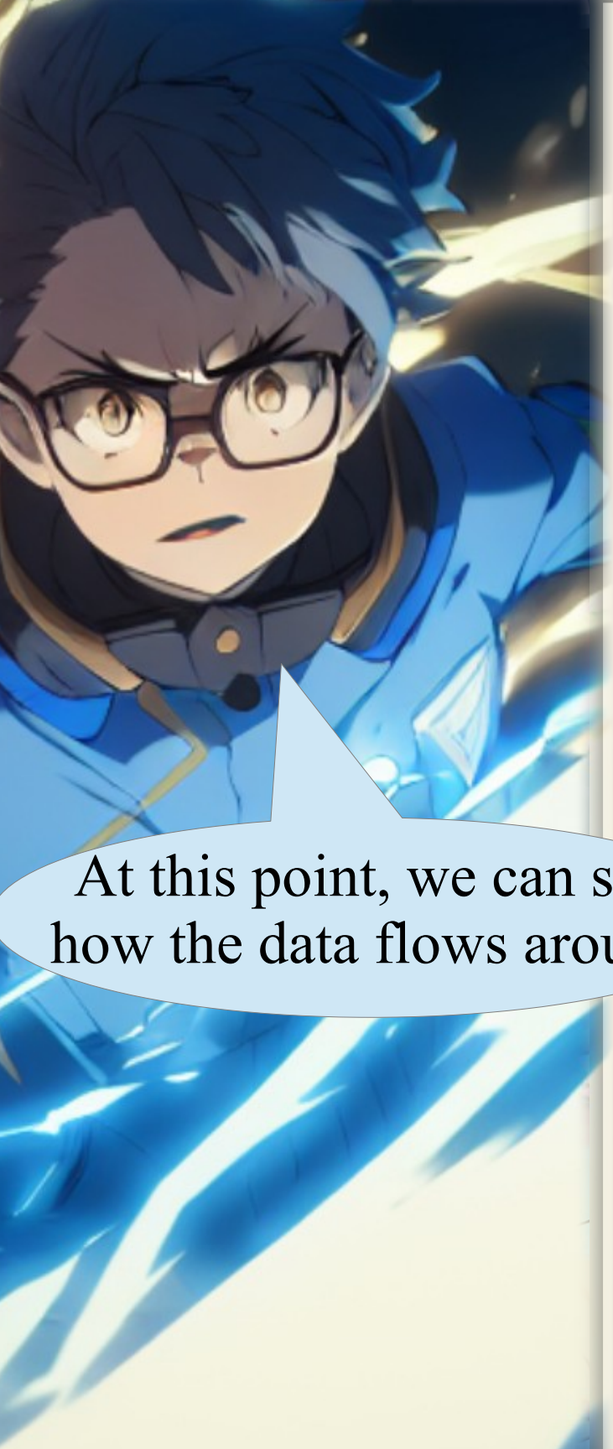


We then merge the two ‘tmp sublistists’ into ‘result’

At the end ‘tmp’ is split sorted ...

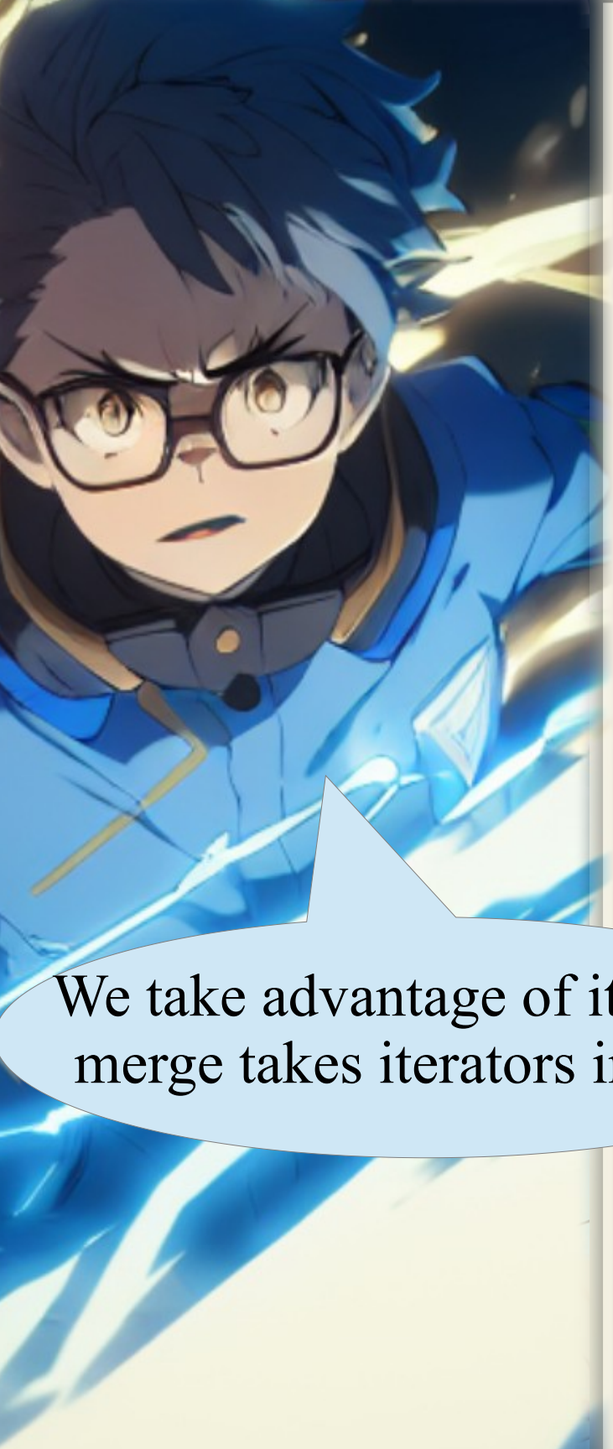
and ‘result’ is fully sorted.

```java
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);
}
```



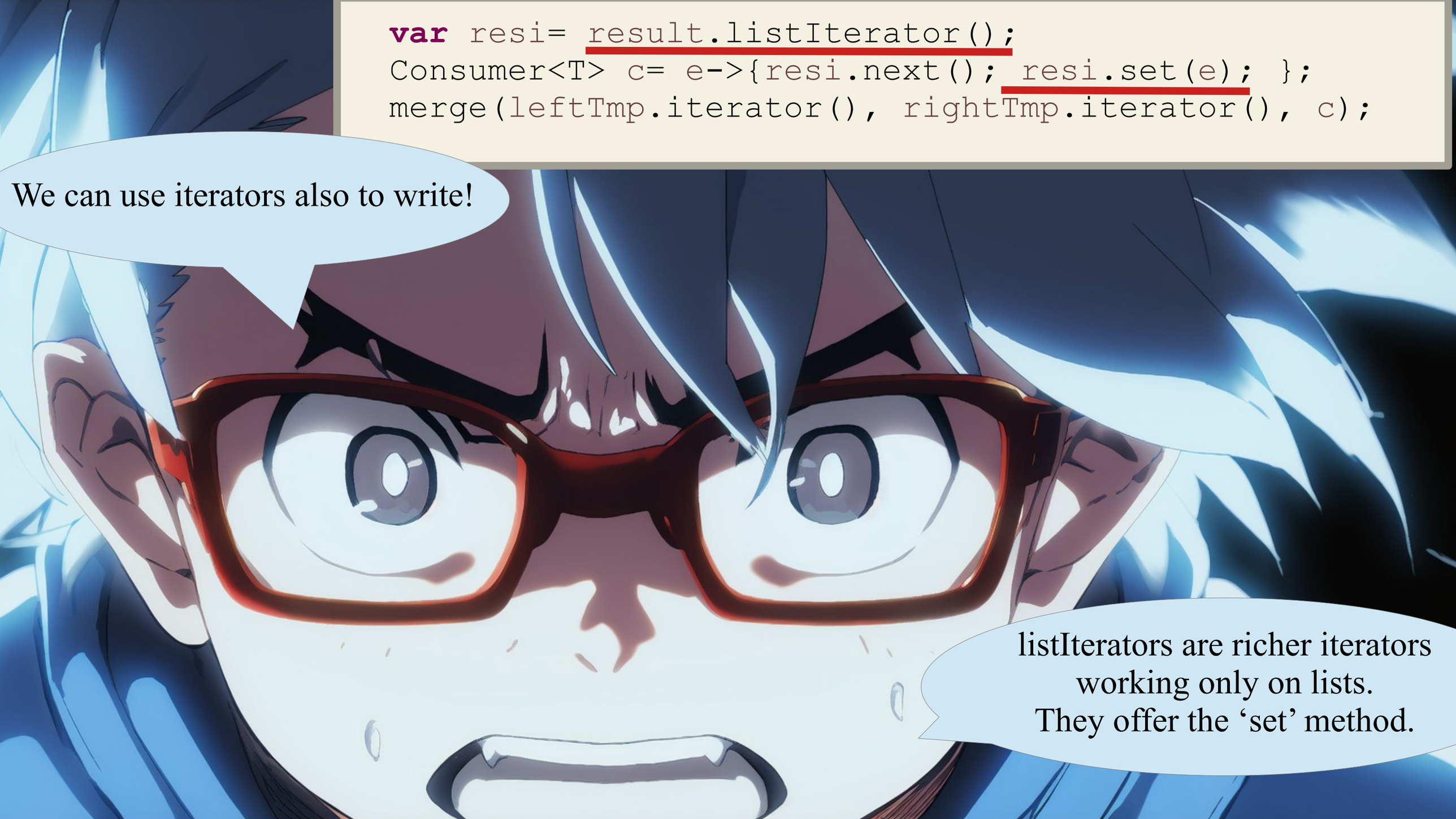

```java
private void sortHelper(List<T> tmp, List<T> result){
  int size= tmp.size();
  if (size < 2) { return; }
  int half = size / 2;
  List<T> leftTmp    = tmp.subList(0, half);
  List<T> rightTmp   = tmp.subList(half, size);
  List<T> leftResult = result.subList(0, half);
  List<T> rightResult= 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();
  ile (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);
}
```

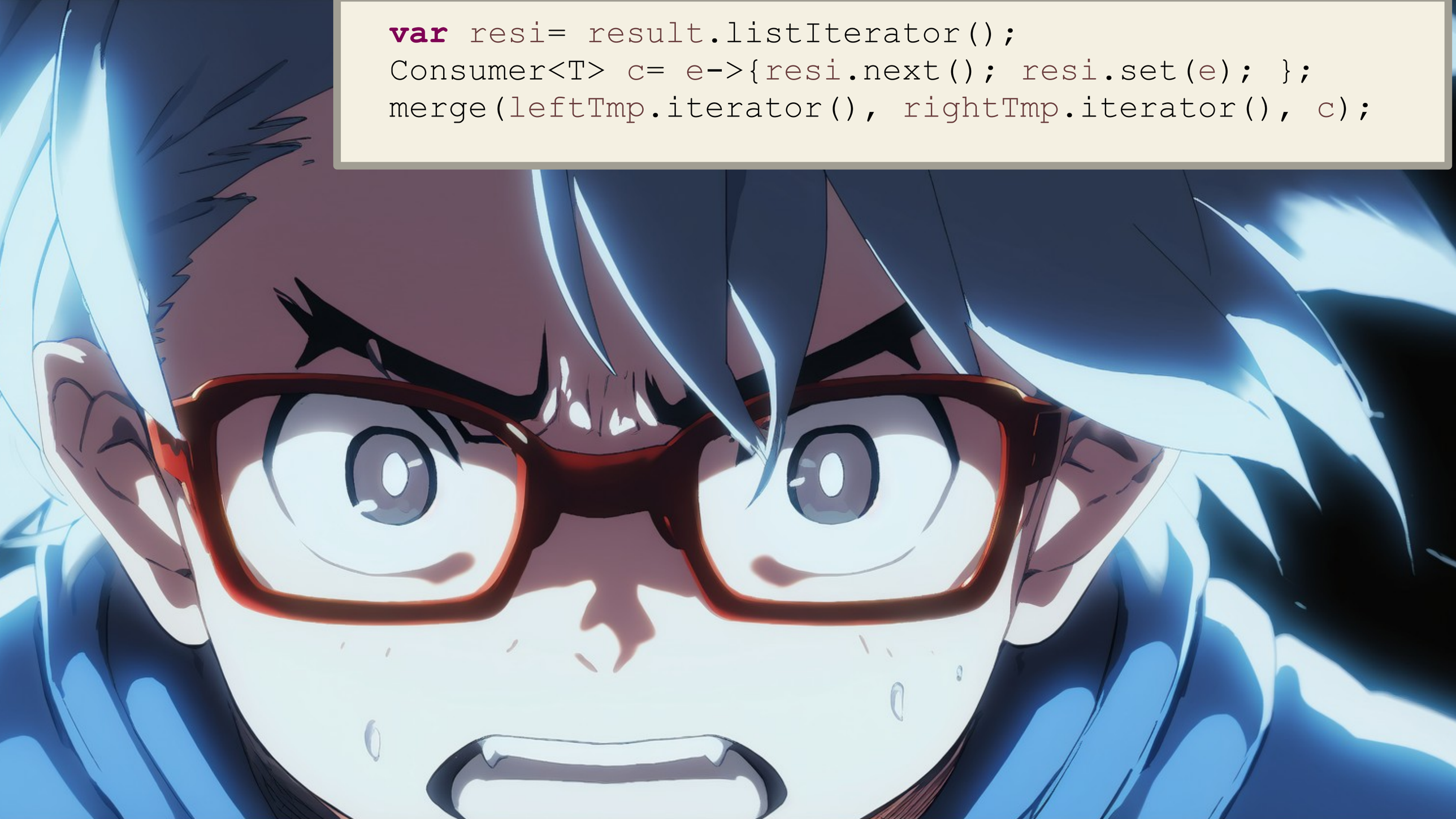
```java
private void sortHelper(List<T> tmp, List<T> result){
  int size= tmp.size();
  if (size < 2) { return; }
  int half = size / 2;
  List<T> leftTmp    = tmp.subList(0, half);
  List<T> rightTmp   = tmp.subList(half, size);
  List<T> leftResult = result.subList(0, half);
  List<T> rightResult= 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);
}
```



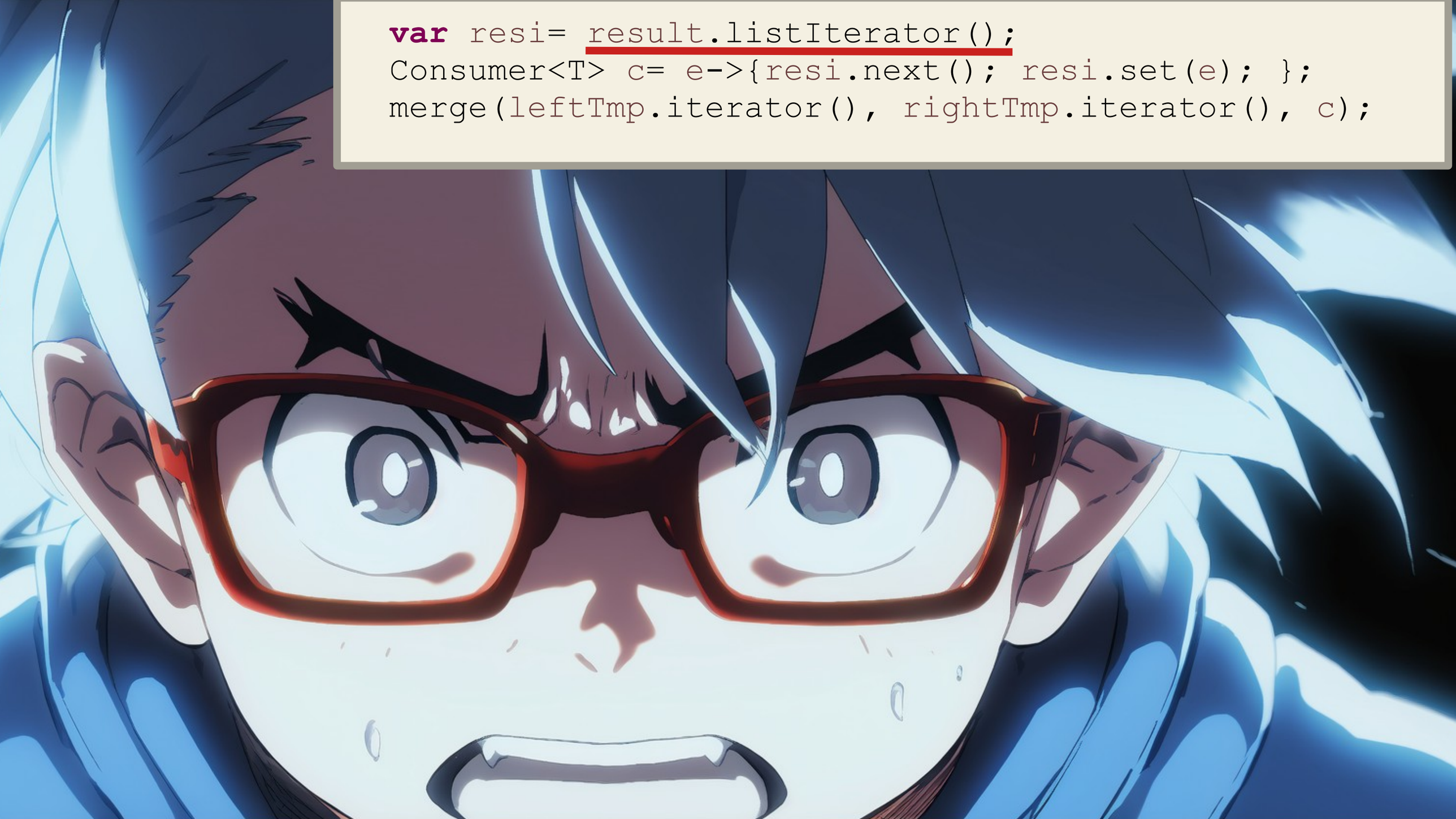

```java
private void sortHelper(List<T> tmp, List<T> result){
  int size= tmp.size();
  if (size < 2) { return; }
  int half = size / 2;
  List<T> leftTmp    = tmp.subList(0, half);
  List<T> rightTmp   = tmp.subList(half, size);
  List<T> leftResult = result.subList(0, half);
  List<T> rightResult= 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();
    bSmaller= currentB.compareTo(currentA) <= 0;
    bSmaller){ res.accept(currentB); continue; }
    es.accept(currentA);
    Iterator<T> tmp = ai; ai = bi; bi = tmp;
    currentA = currentB;
  }
  res.accept(currentA);
  ai.forEachRemaining(res);
}
```

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

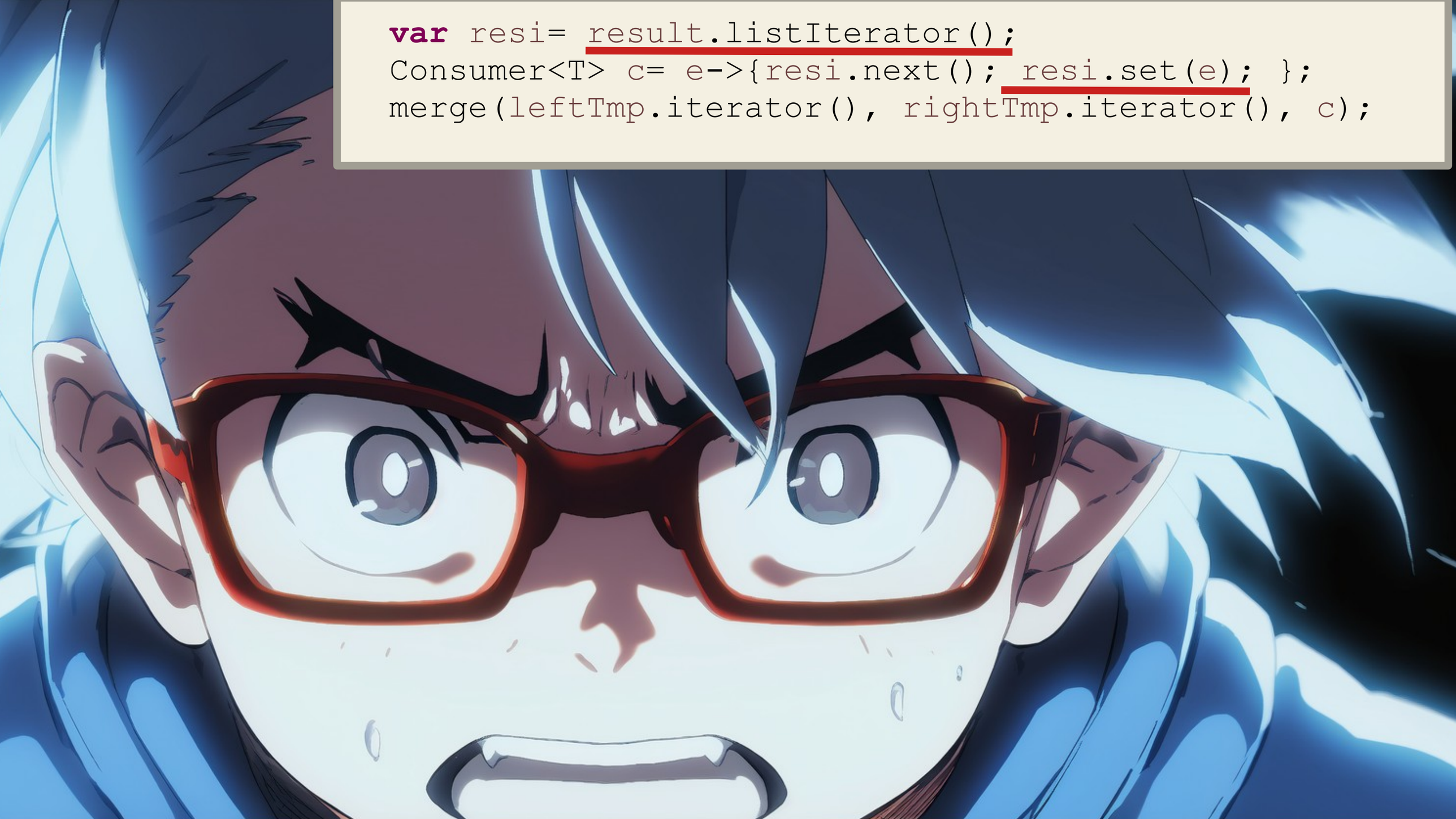




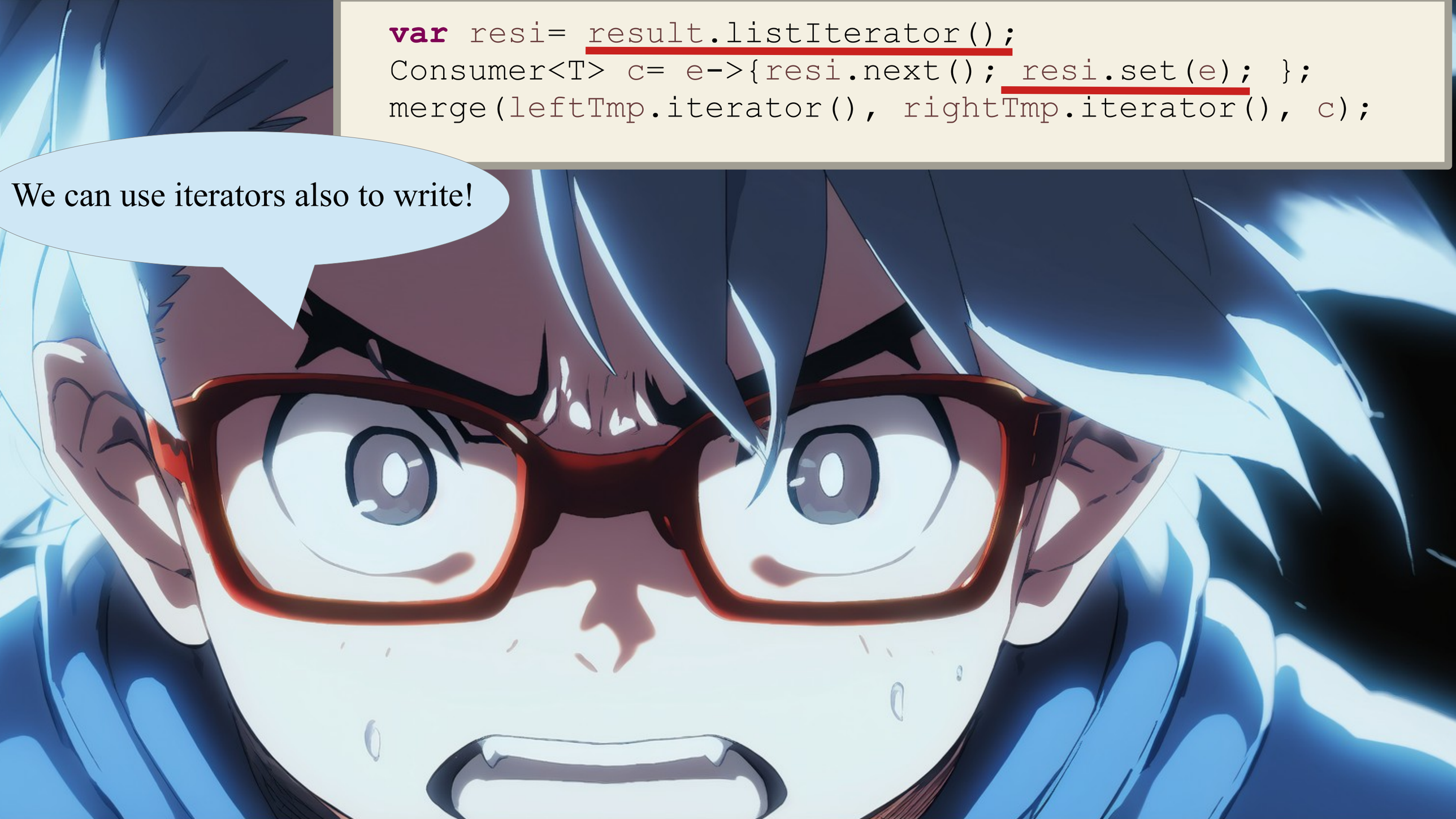
```
var resi= result.listIterator();
Consumer<T> c= e->{resi.next(); resi.set(e); };
merge(leftTmp.iterator(), rightTmp.iterator(), c);
```



```
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);
```


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);
```