



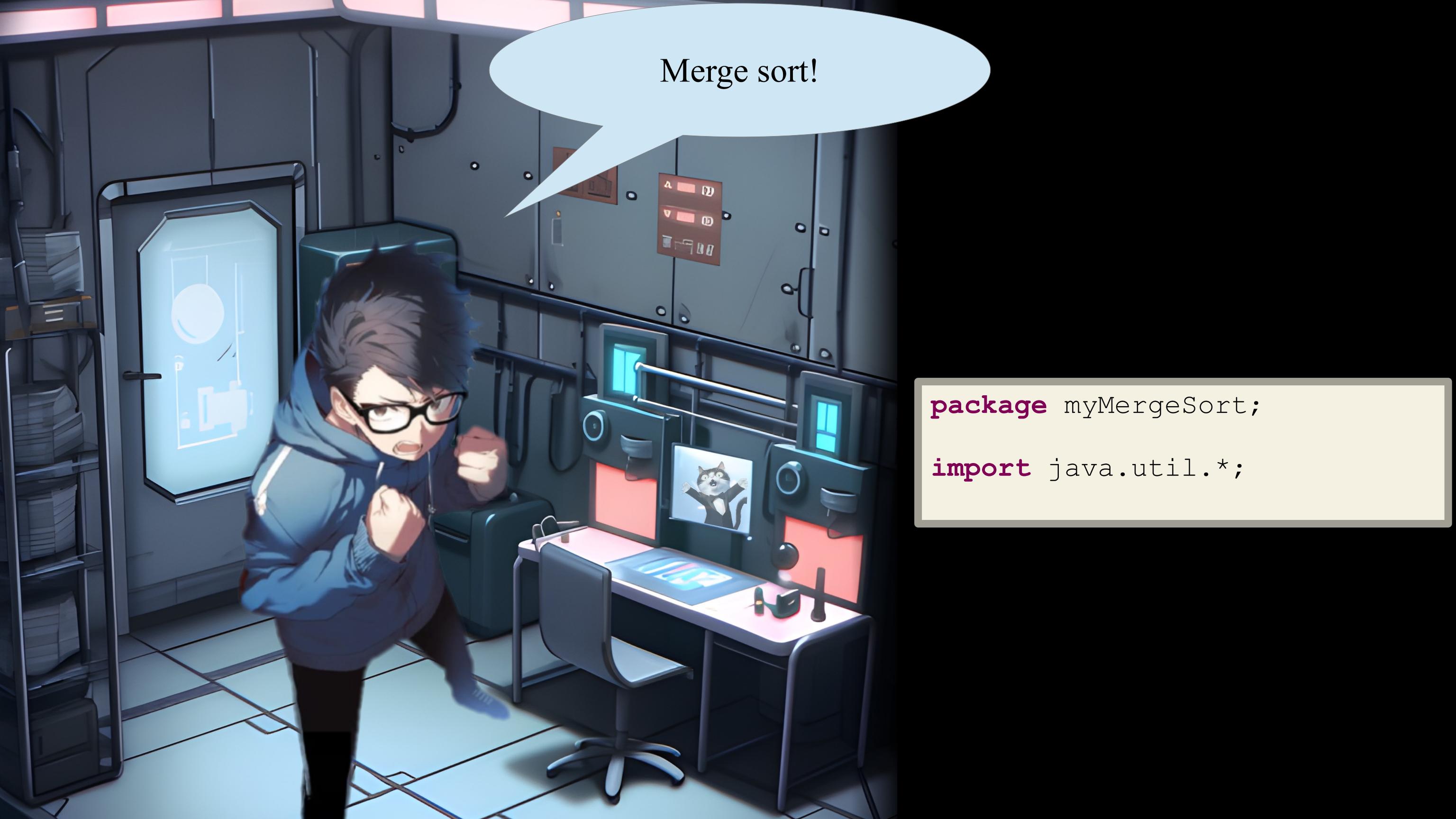


Anyway, lets start with
the first question!





Merge sort!



Merge sort!

```
package myMergeSort;  
import java.util.*;
```



Merge sort!

```
package myMergeSort;  
  
import java.util.*;
```

```
public class Sorter<T> {  
  
    public List<T> sort(List<T> list) { ... }  
}
```



Merge sort!

Clearly the generic type T must extend Comparable

```
package myMergeSort;  
  
import java.util.*;
```

```
public class Sorter<T extends Comparable<T>>{  
  
    public List<T> sort(List<T> list) { ... }  
  
}
```



Merge sort!

Clearly the generic type T must extend Comparable

I'm a little fuzzy on the use of '? super', but this is the way many methods requires comparable elements

```
package myMergeSort;  
  
import java.util.*;
```

```
public class Sorter<T extends Comparable<? super T>>{  
  
    public List<T> sort(List<T> list) { ... }  
  
}
```



```
public List<T> sort(List<T> list) {  
    ...  
}
```



Now, for the body of the sort method itself!

```
public List<T> sort(List<T> list) {  
    ...  
}
```



Now, for the body of the sort method itself!

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



Now, for the body of the sort method itself!

If I remember well,
we have two most important methods:
The recursive method ‘sort’ and
the auxiliary method ‘merge’



Now, for the body of the sort method itself!

If I remember well,
we have two most important methods:
The recursive method ‘sort’ and
the auxiliary method ‘merge’

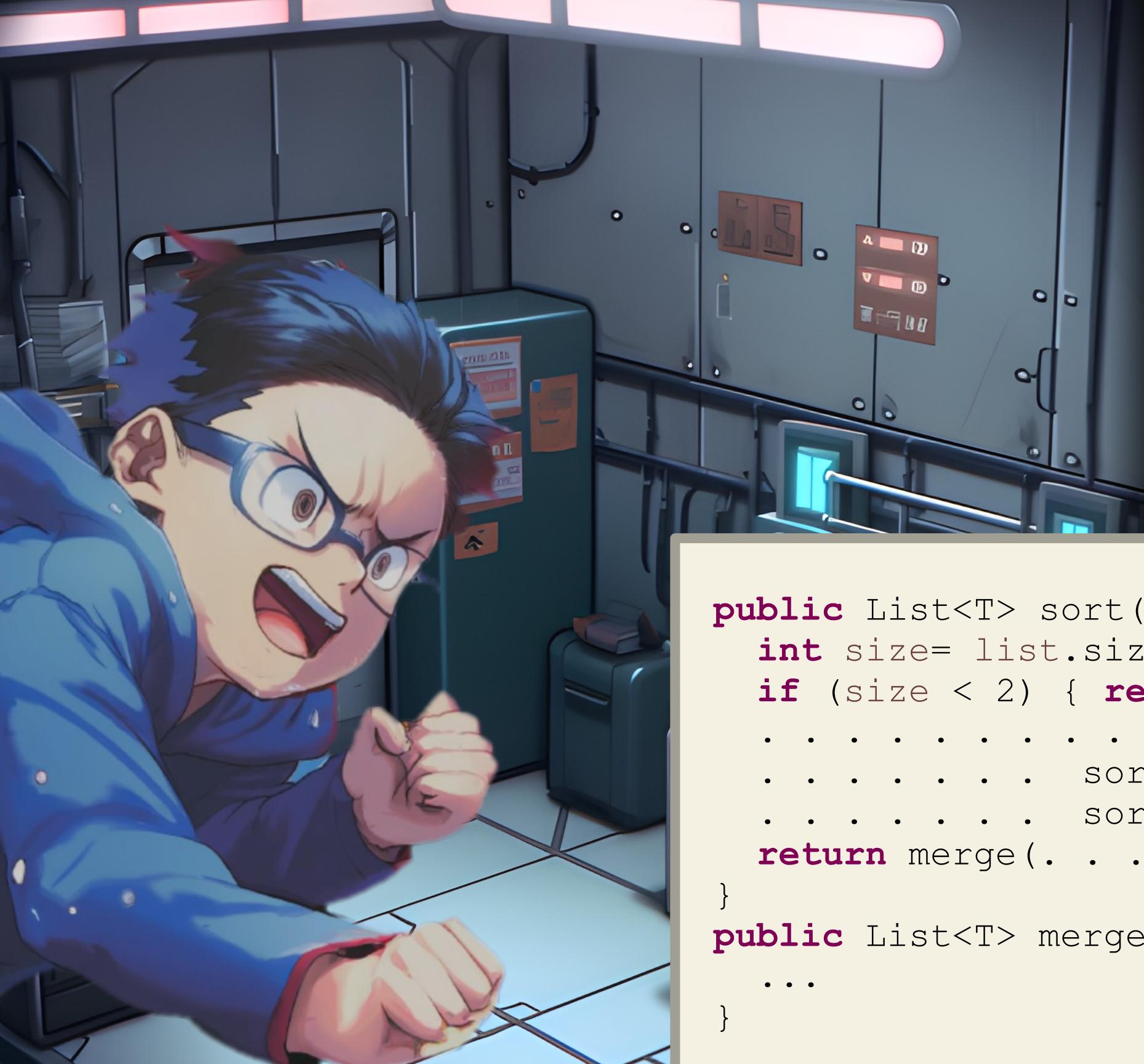
```
public List<T> sort(List<T> list) {  
    int size= list.size();  
    if (size < 2) { return list; }  
    . . . . . . . . .  
    . . . . . . . sort(. . . . . . . . .)  
    . . . . . . . sort(. . . . . . . . .)  
    return merge(. . . . . .);  
}  
public List<T> merge(List<T> a, List<T> b) {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



Now, for the body of the sort method itself!

If I remember well,
we have two most important methods:
The recursive method ‘sort’ and
the auxiliary method ‘merge’

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



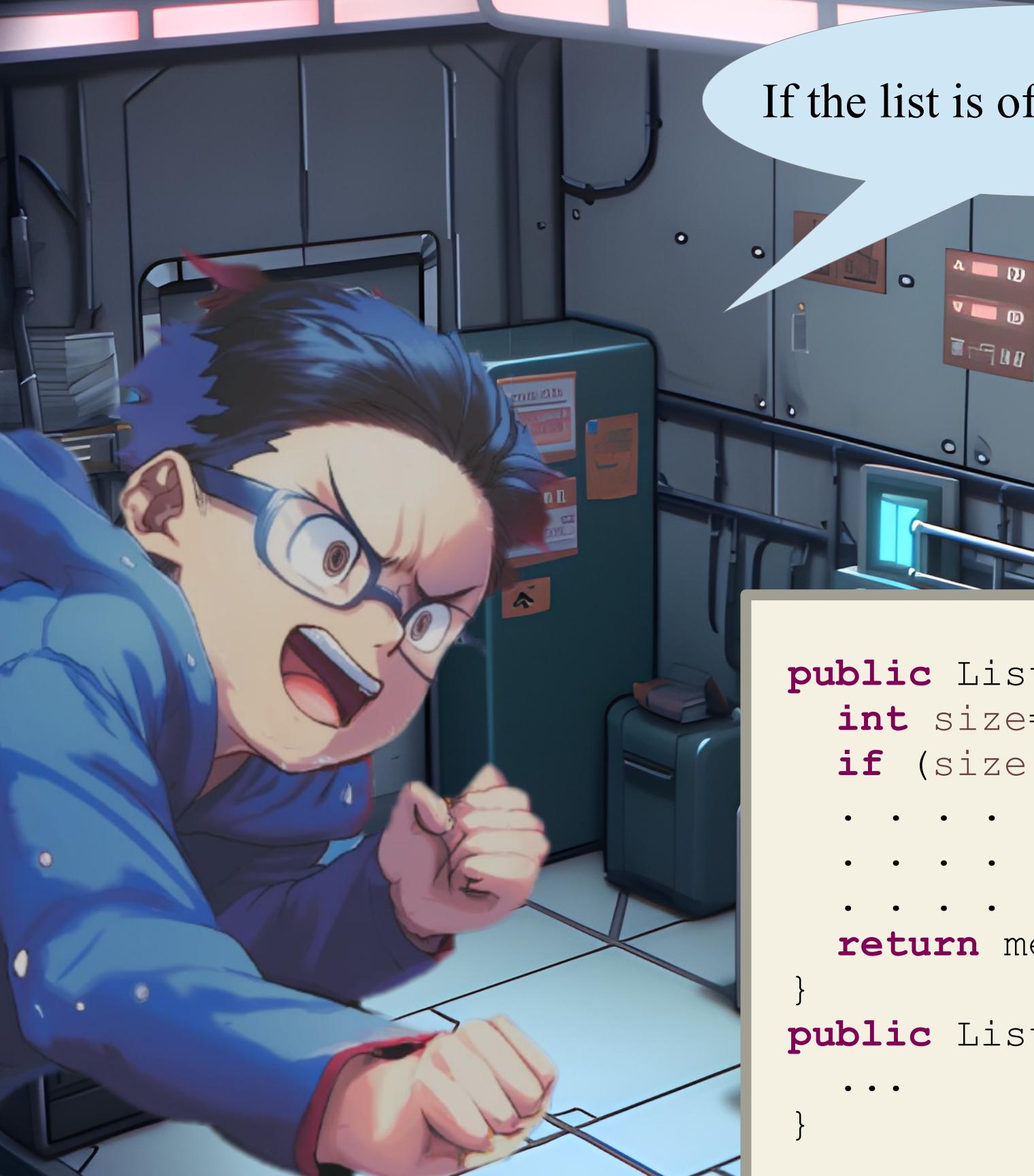
```
public List<T> sort(List<T> list) {  
    int size= list.size();  
    if (size < 2) { return list; }  
    . . . . . . . . . . . . . .  
    . . . . . . . . sort(. . . . . . . . . . . . . . . . . . . . . . . . . . );  
    . . . . . . . . sort(. . . . . . . . . . . . . . . . . . . . . . . . . .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



If the list is of size 1 or zero ...



If the list is of size 1 or zero ..



If the list is of size 1 or zero ..

We are done, we return the list,
since it is already sorted.

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



If the list is of size 1 or zero ..

We are done, we return the list,
since it is already sorted.

This is known as the base case of the recursion

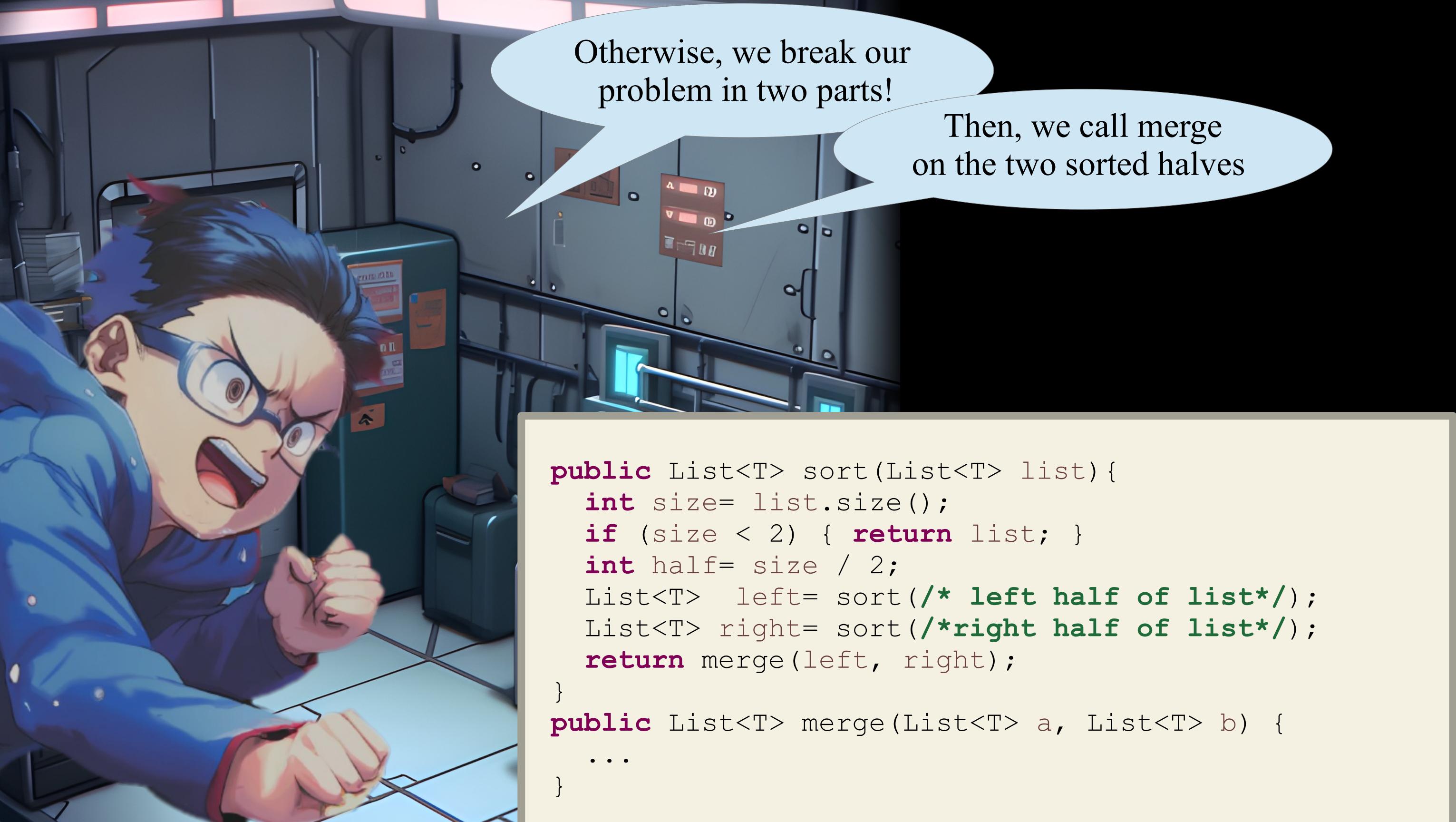


```
public List<T> sort(List<T> list) {  
    int size= list.size();  
    if (size < 2) { return list; }  
    int half= size / 2;  
    List<T> left= sort(/* left half of list */);  
    List<T> right= sort(/*right half of list */);  
    return merge(left, right);  
}  
  
public List<T> merge(List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



Otherwise, we break our problem in two parts!

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



Otherwise, we break our problem in two parts!

Then, we call merge on the two sorted halves

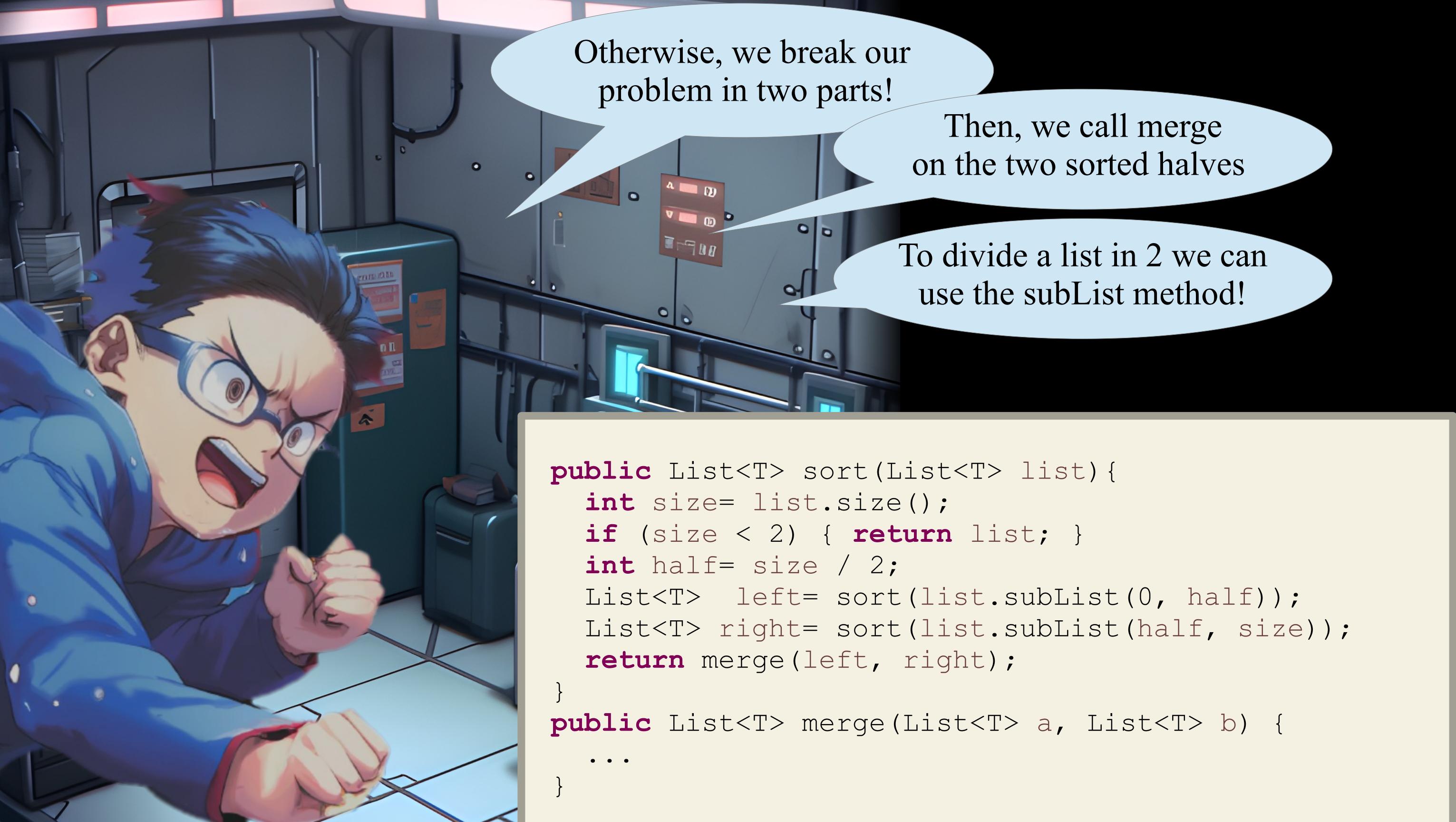
```
public List<T> sort(List<T> list) {  
    int size= list.size();  
    if (size < 2) { return list; }  
    int half= size / 2;  
    List<T> left= sort(/* left half of list */);  
    List<T> right= sort(/*right half of list */);  
    return merge(left, right);  
}  
  
public List<T> merge(List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



Otherwise, we break our problem in two parts!

Then, we call merge on the two sorted halves

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



Otherwise, we break our problem in two parts!

Then, we call merge on the two sorted halves

To divide a list in 2 we can use the subList method!

```
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> 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) {  
    ...  
}
```

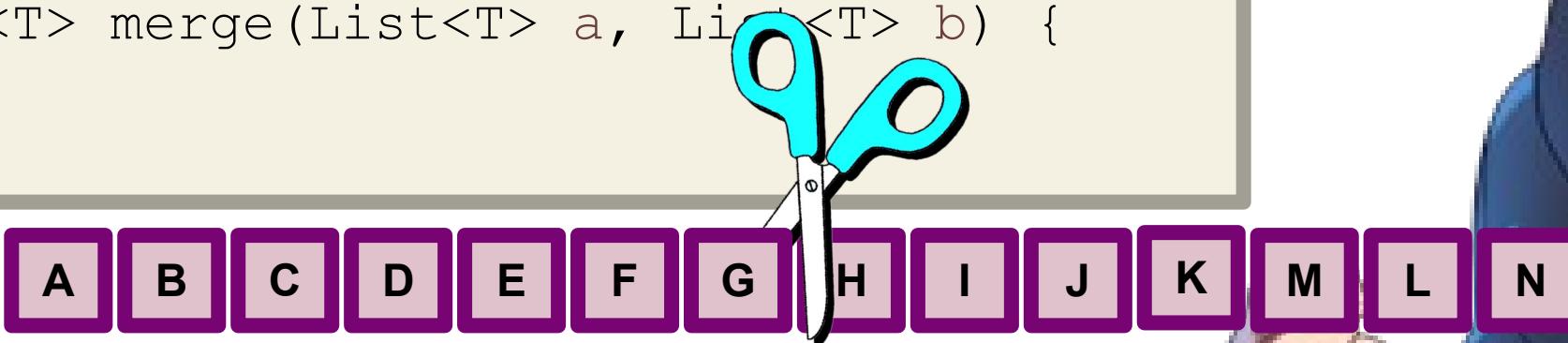


```
public List<T> sort(List<T> list) {  
    int size= list.size();  
    if (size < 2) { return list; }  
    int half= size / 2;  
    List<T> left= sort(list.subList(0, half));  
    List<T> right= sort(list.subList(half, size));  
    return merge(left, right);  
}  
  
public List<T> merge(List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

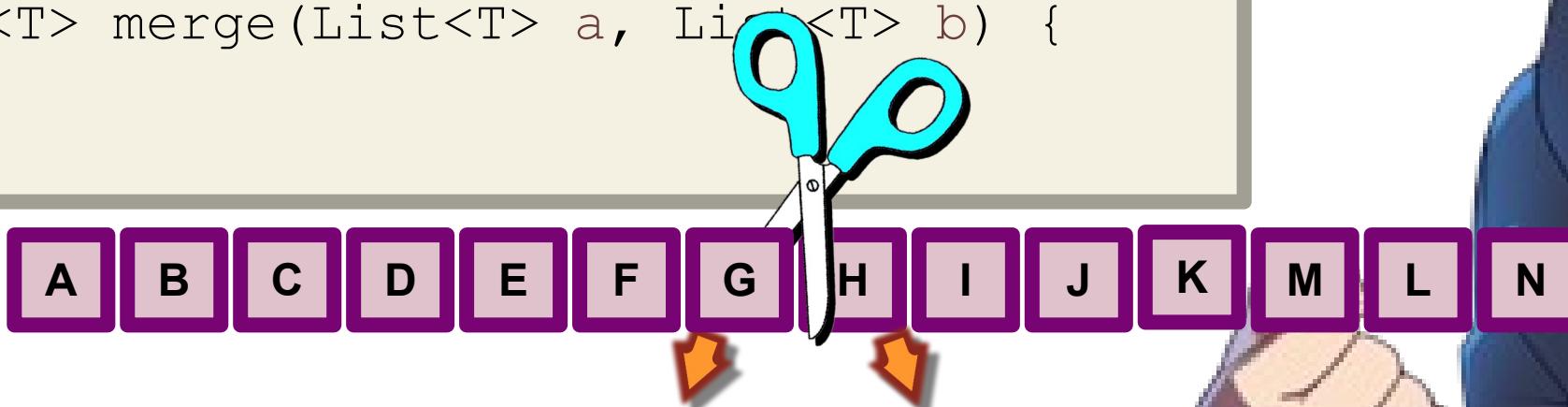
A B C D E F G H I J K M L N



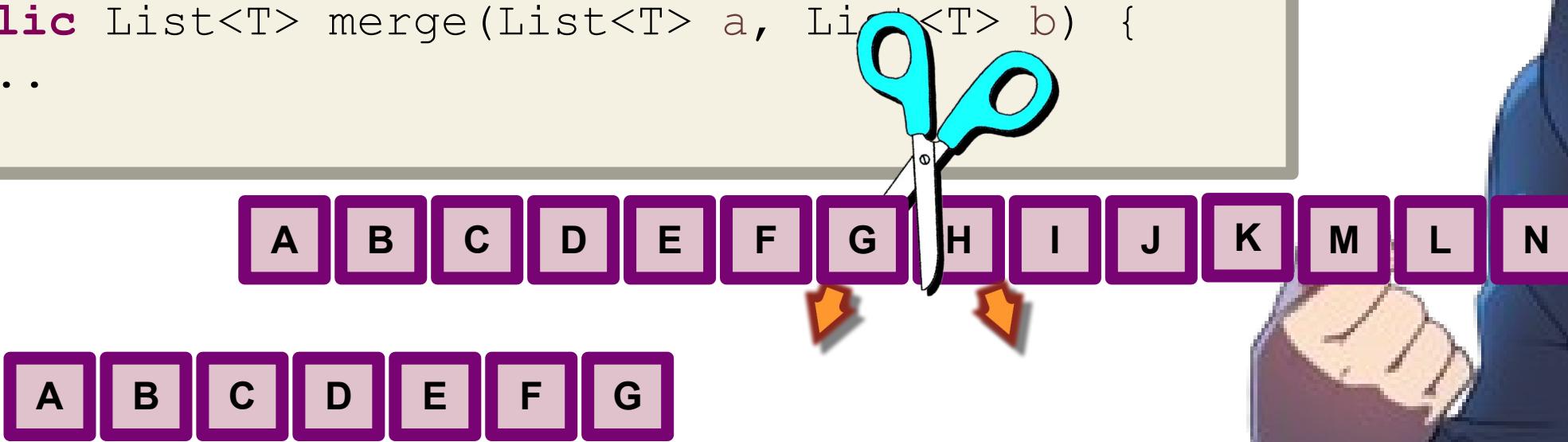
```
public List<T> sort(List<T> list) {  
    int size= list.size();  
    if (size < 2) { return list; }  
    int half= size / 2;  
    List<T> left= sort(list.subList(0, half));  
    List<T> right= sort(list.subList(half, size));  
    return merge(left, right);  
}  
  
public List<T> merge(List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



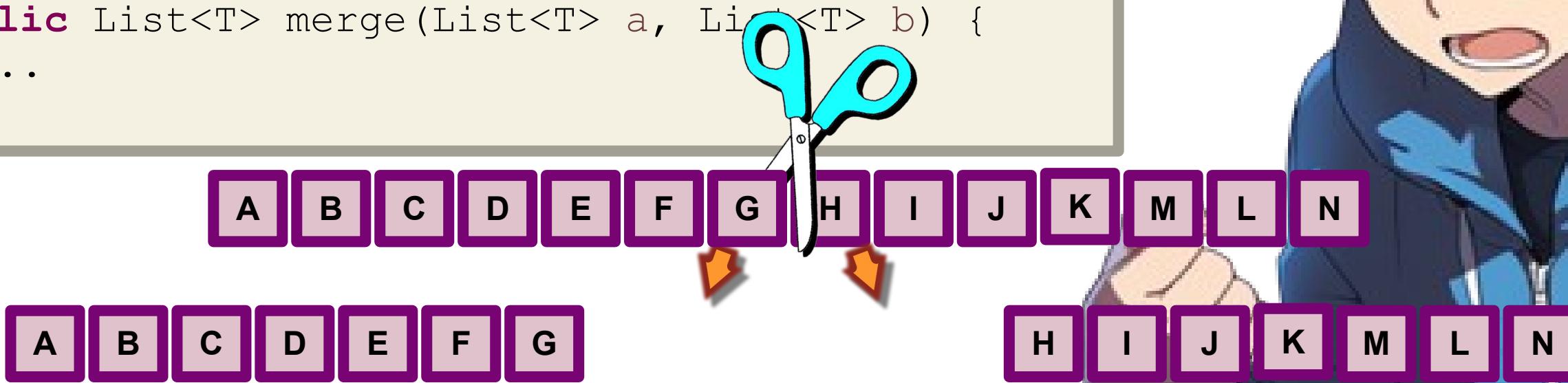
```
public List<T> sort(List<T> list) {  
    int size= list.size();  
    if (size < 2) { return list; }  
    int half= size / 2;  
    List<T> left= sort(list.subList(0, half));  
    List<T> right= sort(list.subList(half, size));  
    return merge(left, right);  
}  
  
public List<T> merge(List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



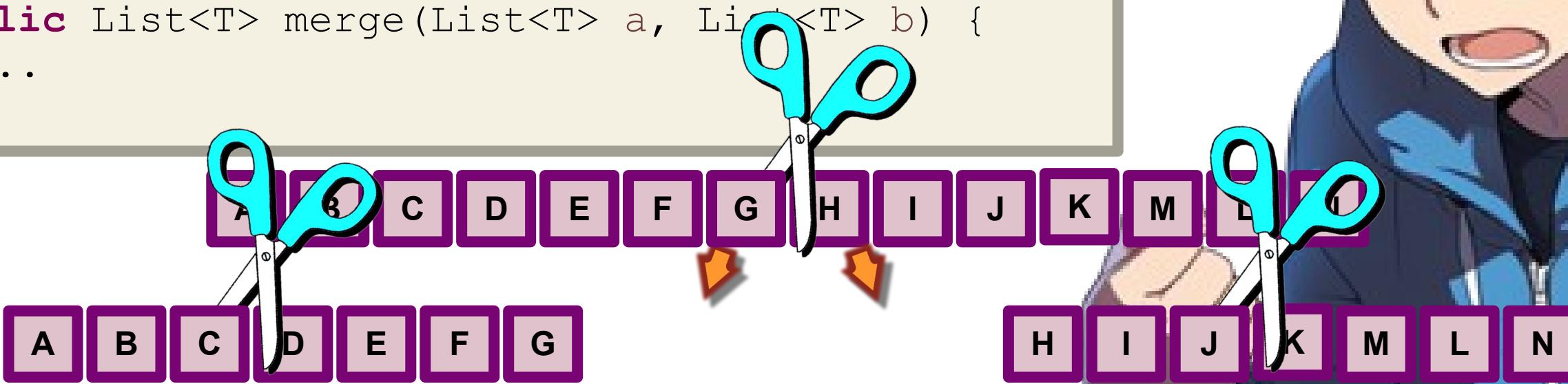
```
public List<T> sort(List<T> list) {  
    int size= list.size();  
    if (size < 2) { return list; }  
    int half= size / 2;  
    List<T> left= sort(list.subList(0, half));  
    List<T> right= sort(list.subList(half, size));  
    return merge(left, right);  
}  
  
public List<T> merge(List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



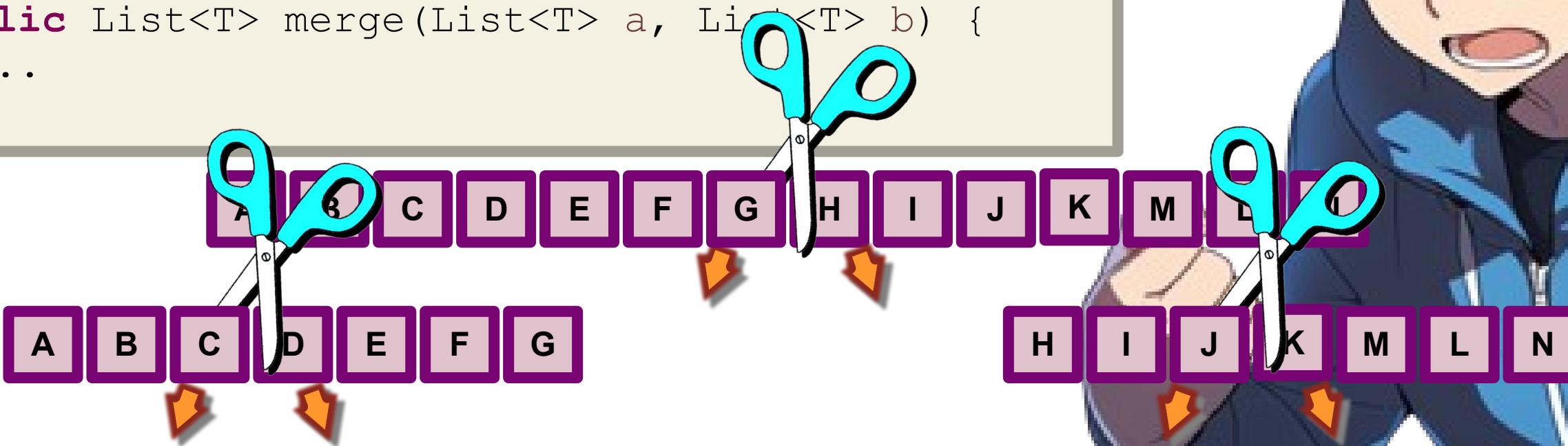
```
public List<T> sort(List<T> list) {  
    int size= list.size();  
    if (size < 2) { return list; }  
    int half= size / 2;  
    List<T> left= sort(list.subList(0, half));  
    List<T> right= sort(list.subList(half, size));  
    return merge(left, right);  
}  
  
public List<T> merge(List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



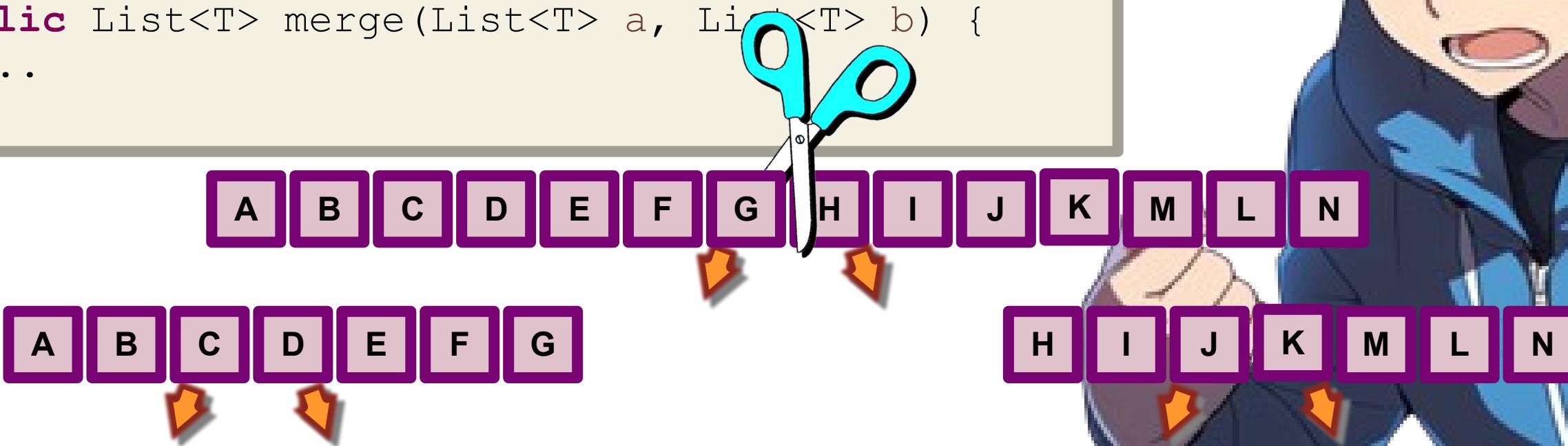
```
public List<T> sort(List<T> list) {  
    int size= list.size();  
    if (size < 2) { return list; }  
    int half= size / 2;  
    List<T> left= sort(list.subList(0, half));  
    List<T> right= sort(list.subList(half, size));  
    return merge(left, right);  
}  
  
public List<T> merge(List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



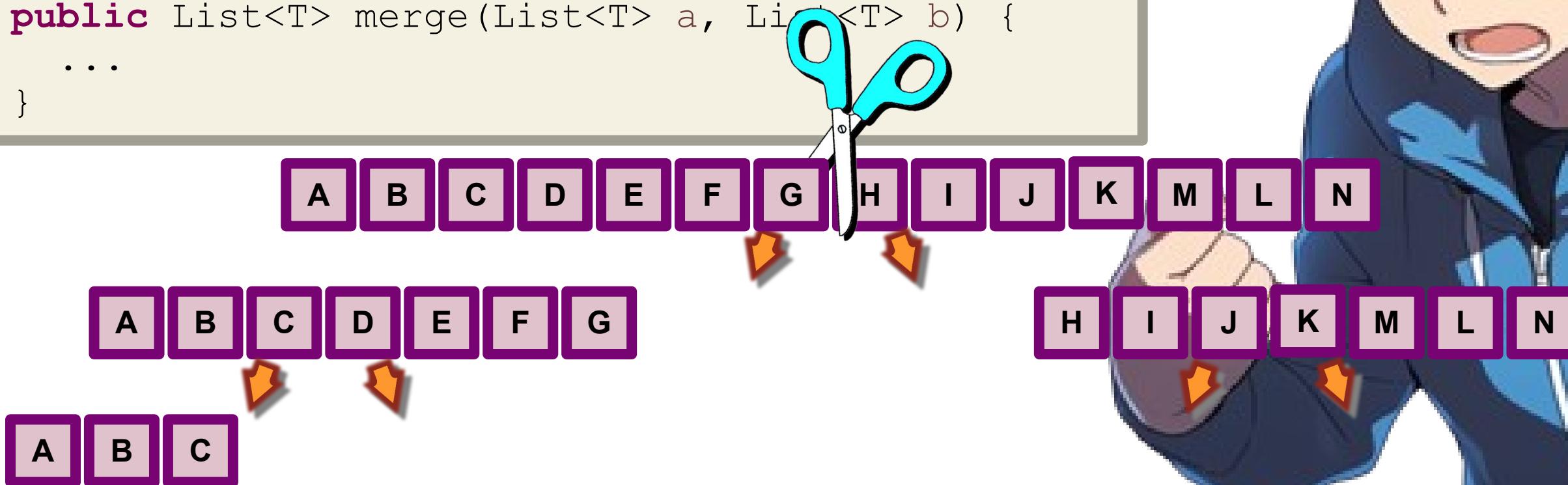
```
public List<T> sort(List<T> list) {  
    int size= list.size();  
    if (size < 2) { return list; }  
    int half= size / 2;  
    List<T> left= sort(list.subList(0, half));  
    List<T> right= sort(list.subList(half, size));  
    return merge(left, right);  
}  
  
public List<T> merge(List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



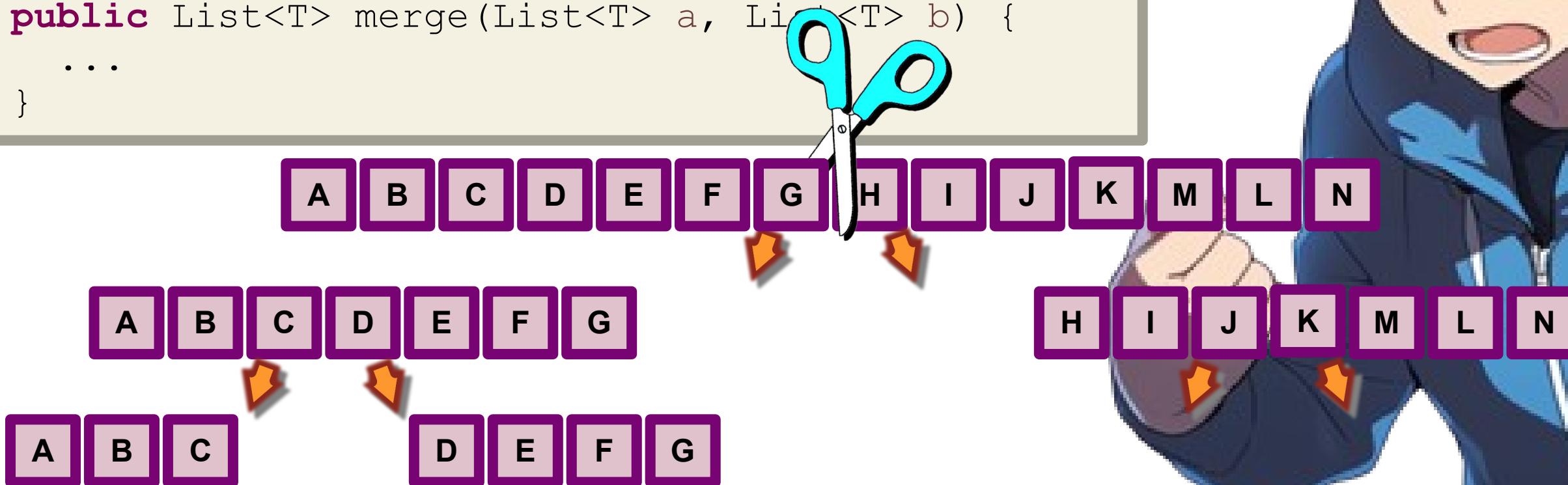
```
public List<T> sort(List<T> list) {  
    int size= list.size();  
    if (size < 2) { return list; }  
    int half= size / 2;  
    List<T> left= sort(list.subList(0, half));  
    List<T> right= sort(list.subList(half, size));  
    return merge(left, right);  
}  
  
public List<T> merge(List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



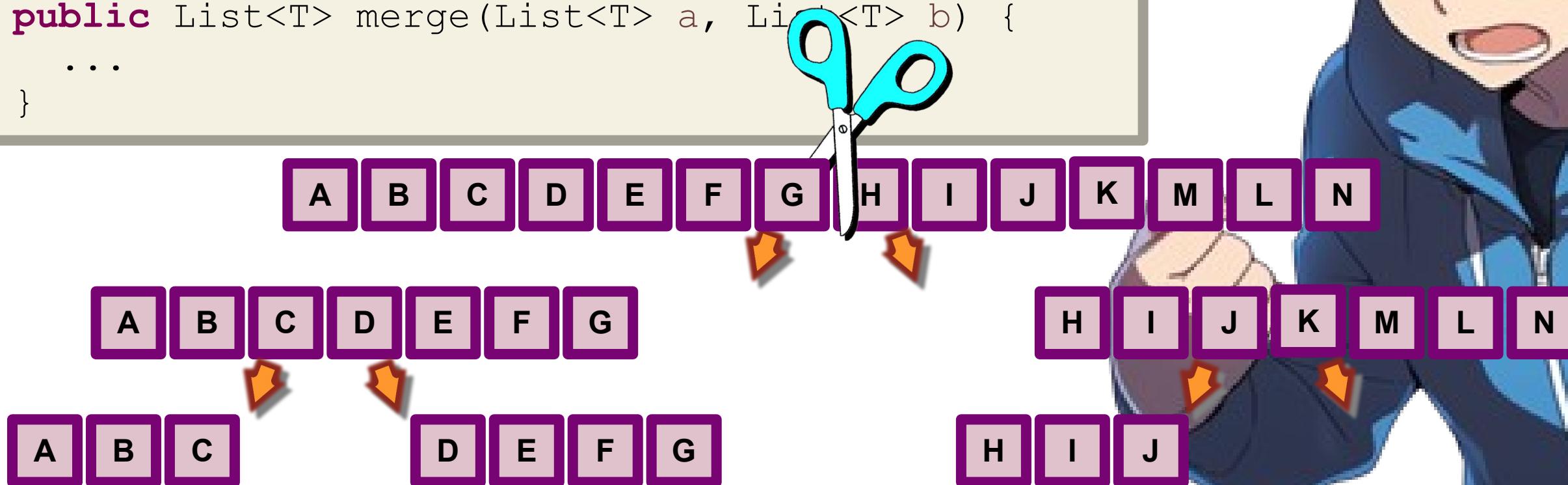
```
public List<T> sort(List<T> list) {  
    int size= list.size();  
    if (size < 2) { return list; }  
    int half= size / 2;  
    List<T> left= sort(list.subList(0, half));  
    List<T> right= sort(list.subList(half, size));  
    return merge(left, right);  
}  
  
public List<T> merge(List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



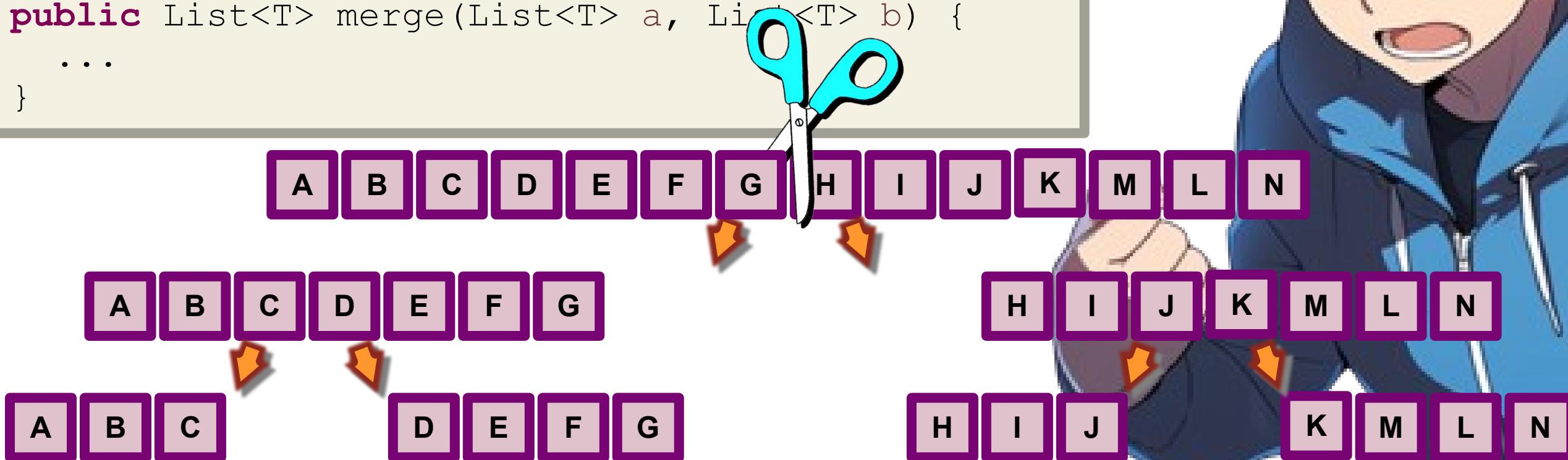
```
public List<T> sort(List<T> list) {  
    int size= list.size();  
    if (size < 2) { return list; }  
    int half= size / 2;  
    List<T> left= sort(list.subList(0, half));  
    List<T> right= sort(list.subList(half, size));  
    return merge(left, right);  
}  
  
public List<T> merge(List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



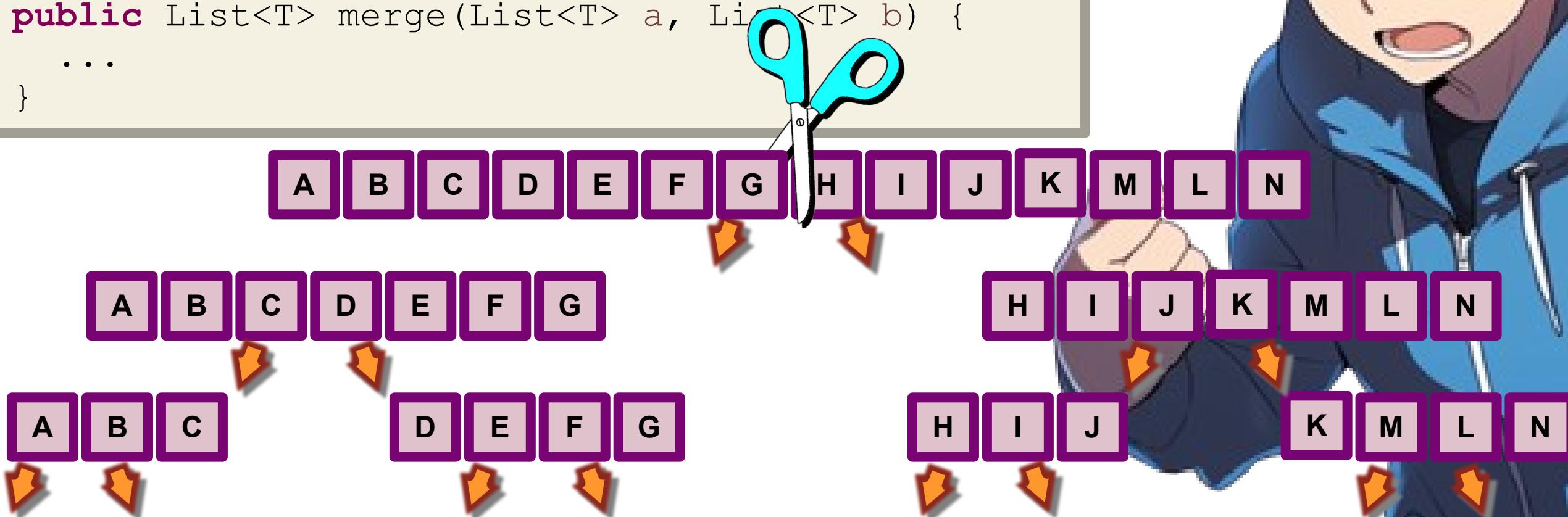
```
public List<T> sort(List<T> list) {  
    int size= list.size();  
    if (size < 2) { return list; }  
    int half= size / 2;  
    List<T> left= sort(list.subList(0, half));  
    List<T> right= sort(list.subList(half, size));  
    return merge(left, right);  
}  
  
public List<T> merge(List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



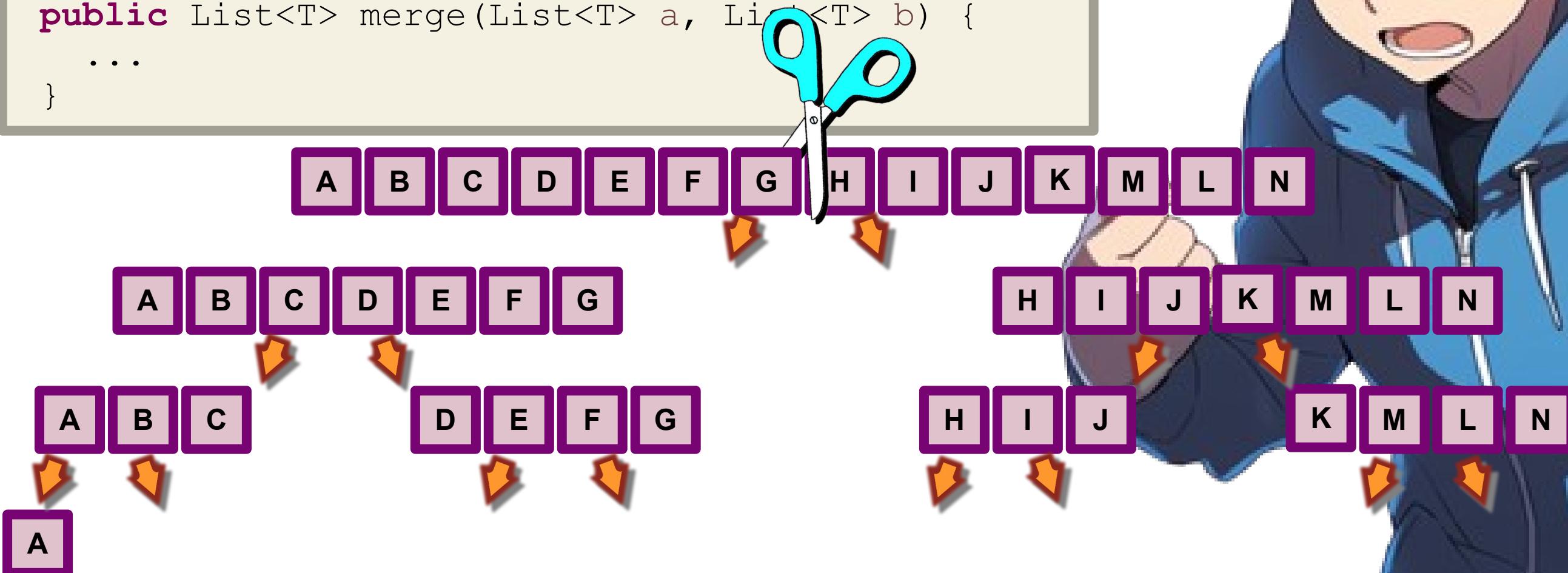
```
public List<T> sort(List<T> list) {  
    int size= list.size();  
    if (size < 2) { return list; }  
    int half= size / 2;  
    List<T> left= sort(list.subList(0, half));  
    List<T> right= sort(list.subList(half, size));  
    return merge(left, right);  
}  
  
public List<T> merge(List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



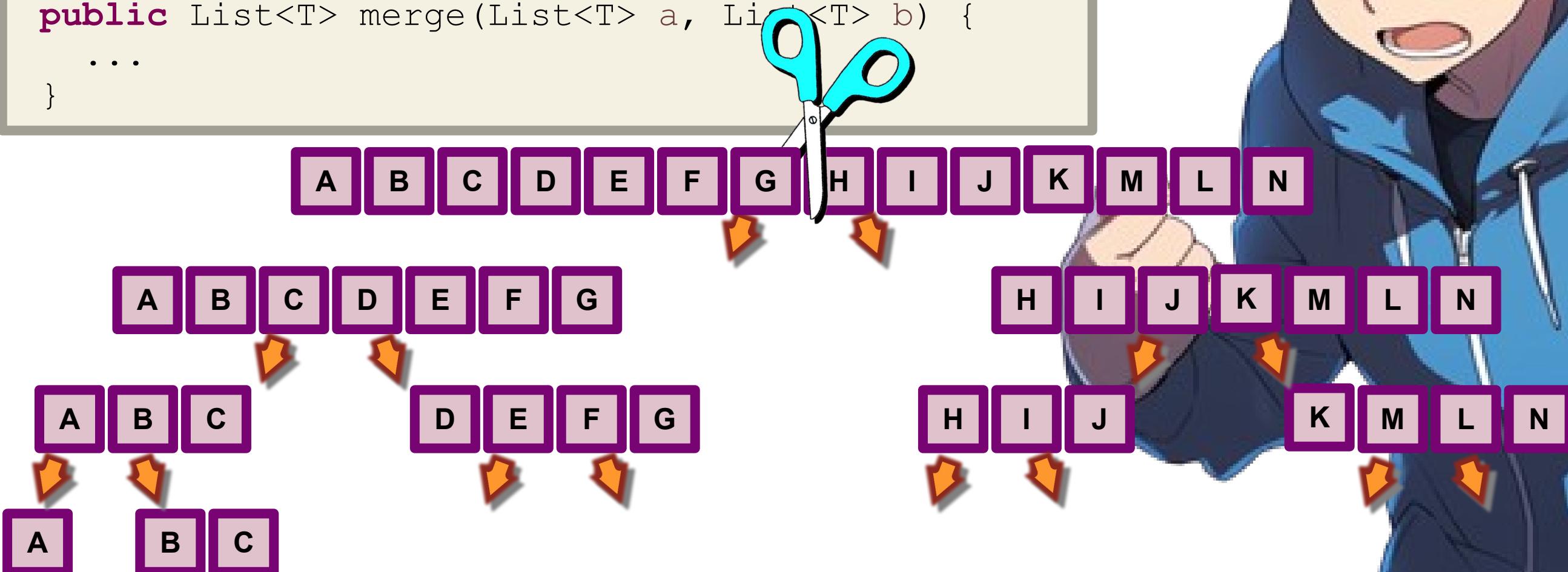
```
public List<T> sort(List<T> list) {  
    int size= list.size();  
    if (size < 2) { return list; }  
    int half= size / 2;  
    List<T> left= sort(list.subList(0, half));  
    List<T> right= sort(list.subList(half, size));  
    return merge(left, right);  
}  
  
public List<T> merge(List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



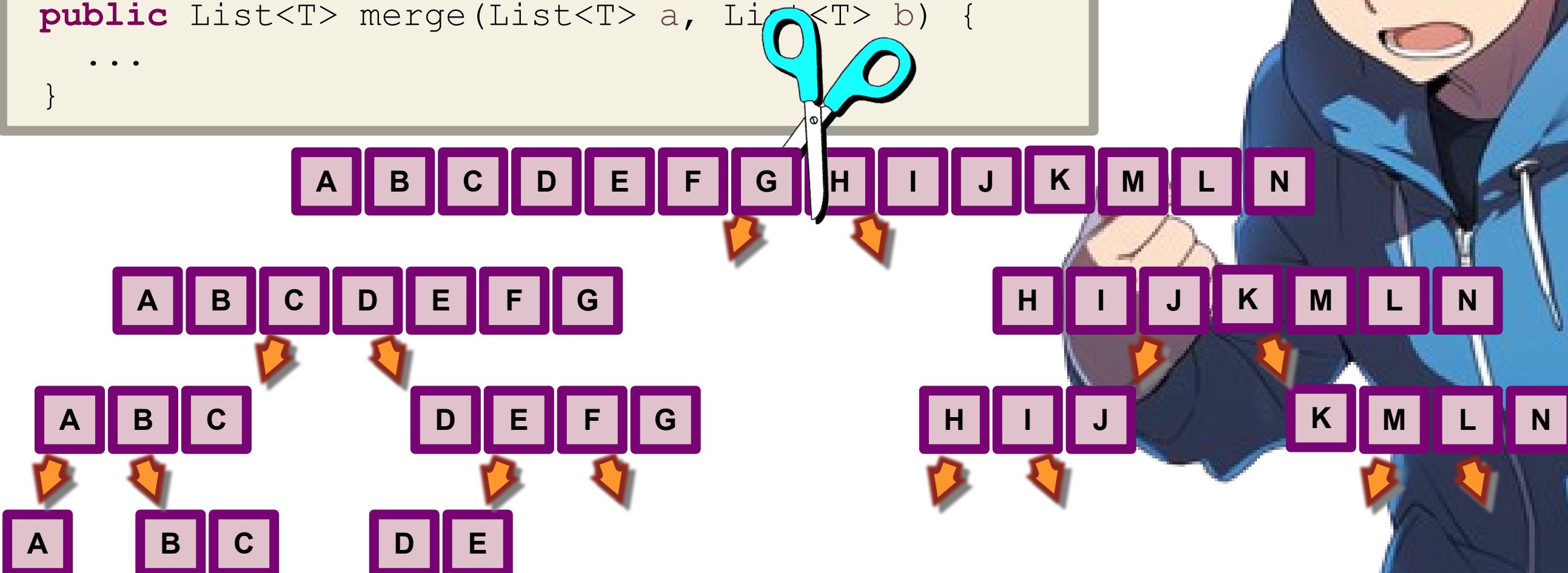
```
public List<T> sort(List<T> list) {  
    int size= list.size();  
    if (size < 2) { return list; }  
    int half= size / 2;  
    List<T> left= sort(list.subList(0, half));  
    List<T> right= sort(list.subList(half, size));  
    return merge(left, right);  
}  
  
public List<T> merge(List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



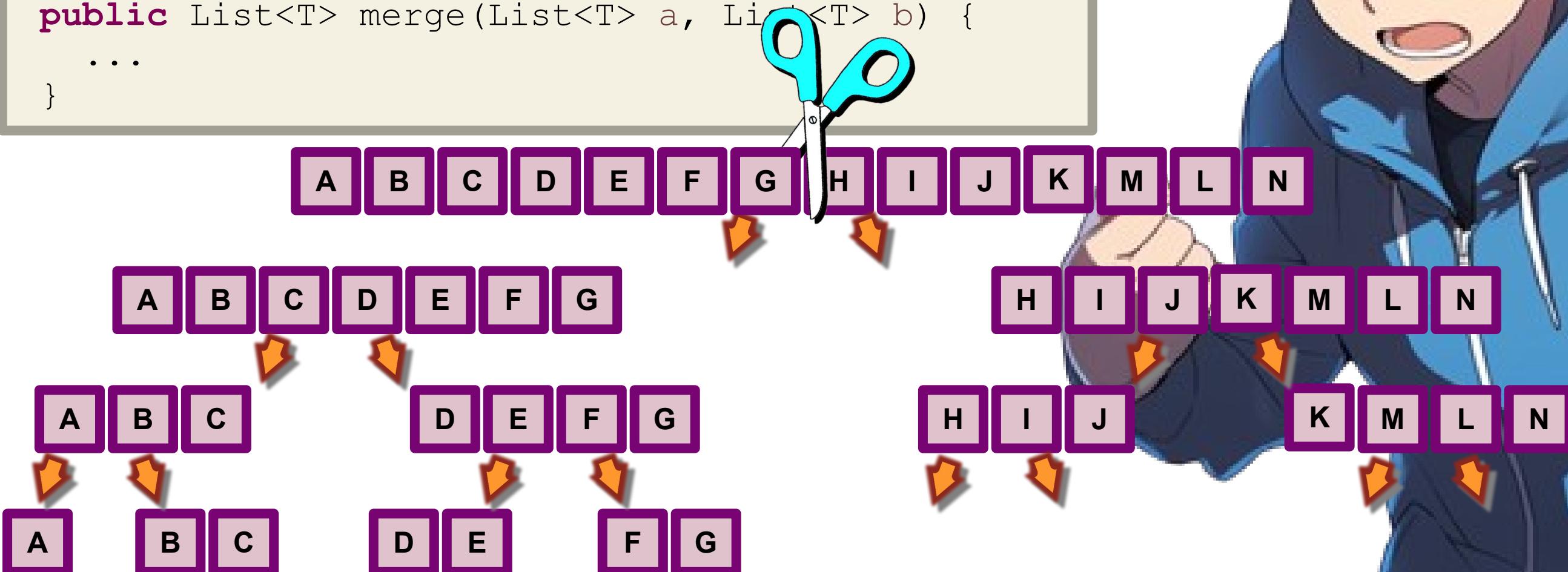
```
public List<T> sort(List<T> list) {  
    int size= list.size();  
    if (size < 2) { return list; }  
    int half= size / 2;  
    List<T> left= sort(list.subList(0, half));  
    List<T> right= sort(list.subList(half, size));  
    return merge(left, right);  
}  
  
public List<T> merge(List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



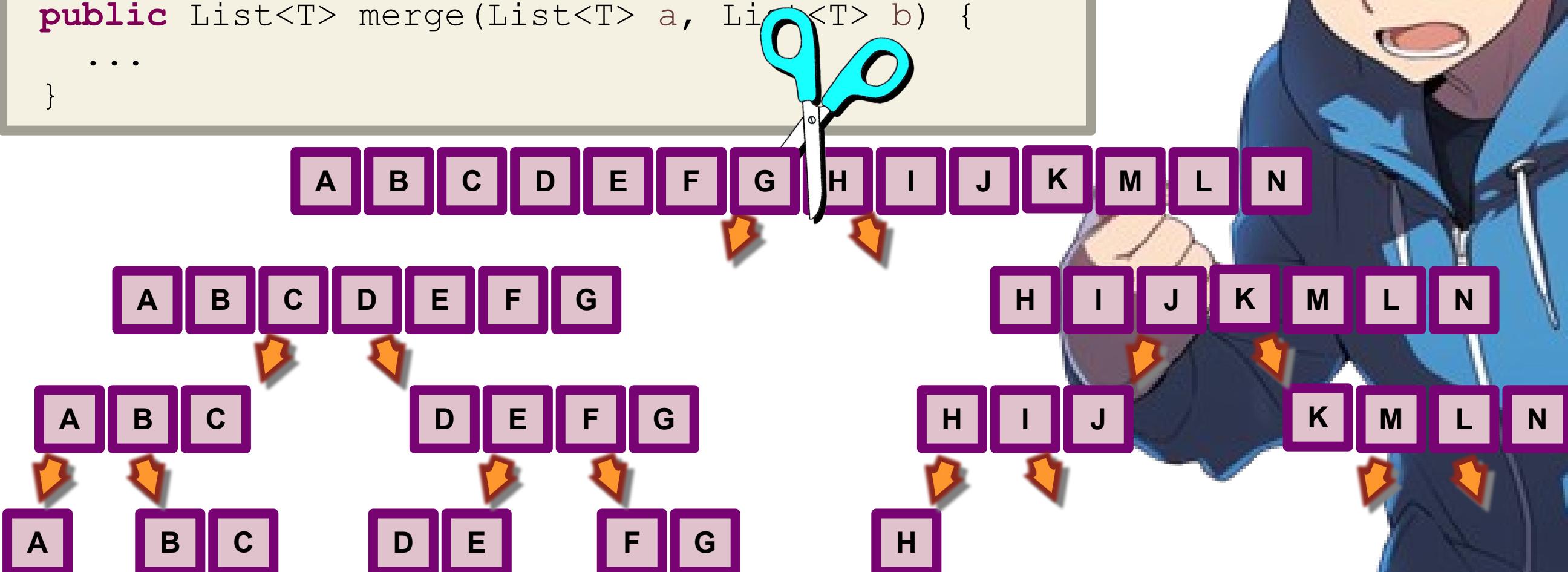
```
public List<T> sort(List<T> list) {  
    int size= list.size();  
    if (size < 2) { return list; }  
    int half= size / 2;  
    List<T> left= sort(list.subList(0, half));  
    List<T> right= sort(list.subList(half, size));  
    return merge(left, right);  
}  
  
public List<T> merge(List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



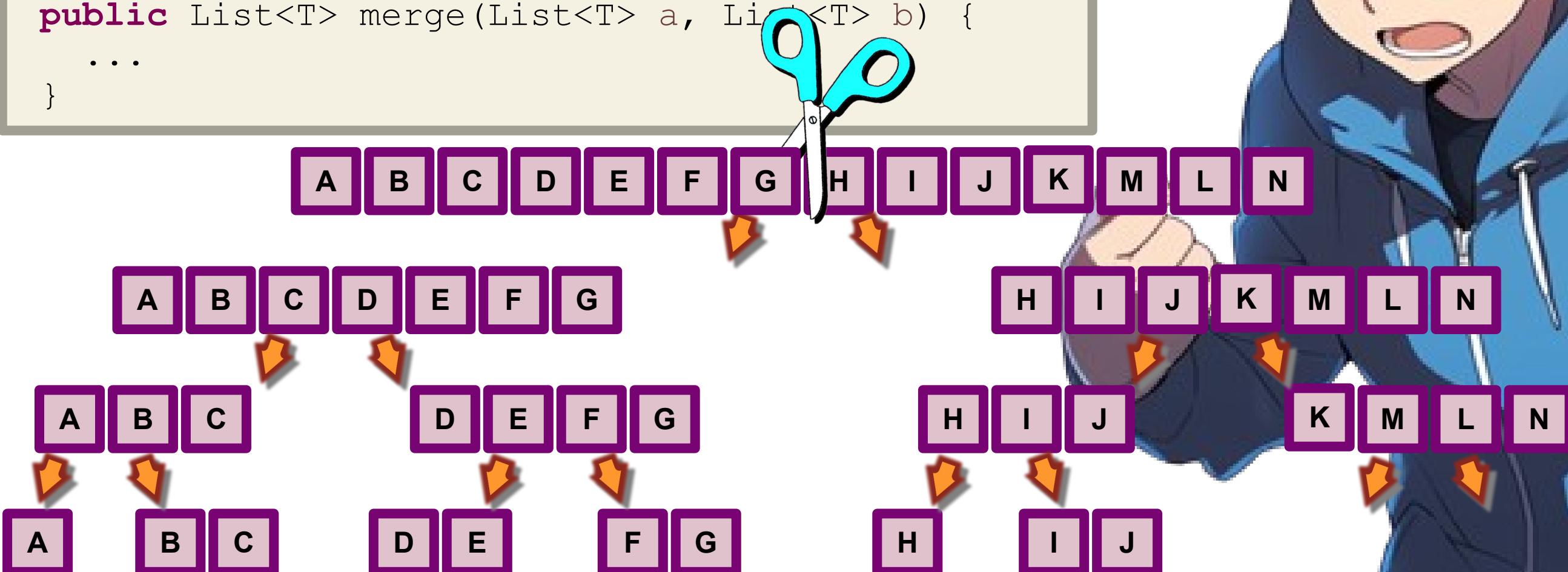
```
public List<T> sort(List<T> list) {  
    int size= list.size();  
    if (size < 2) { return list; }  
    int half= size / 2;  
    List<T> left= sort(list.subList(0, half));  
    List<T> right= sort(list.subList(half, size));  
    return merge(left, right);  
}  
  
public List<T> merge(List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



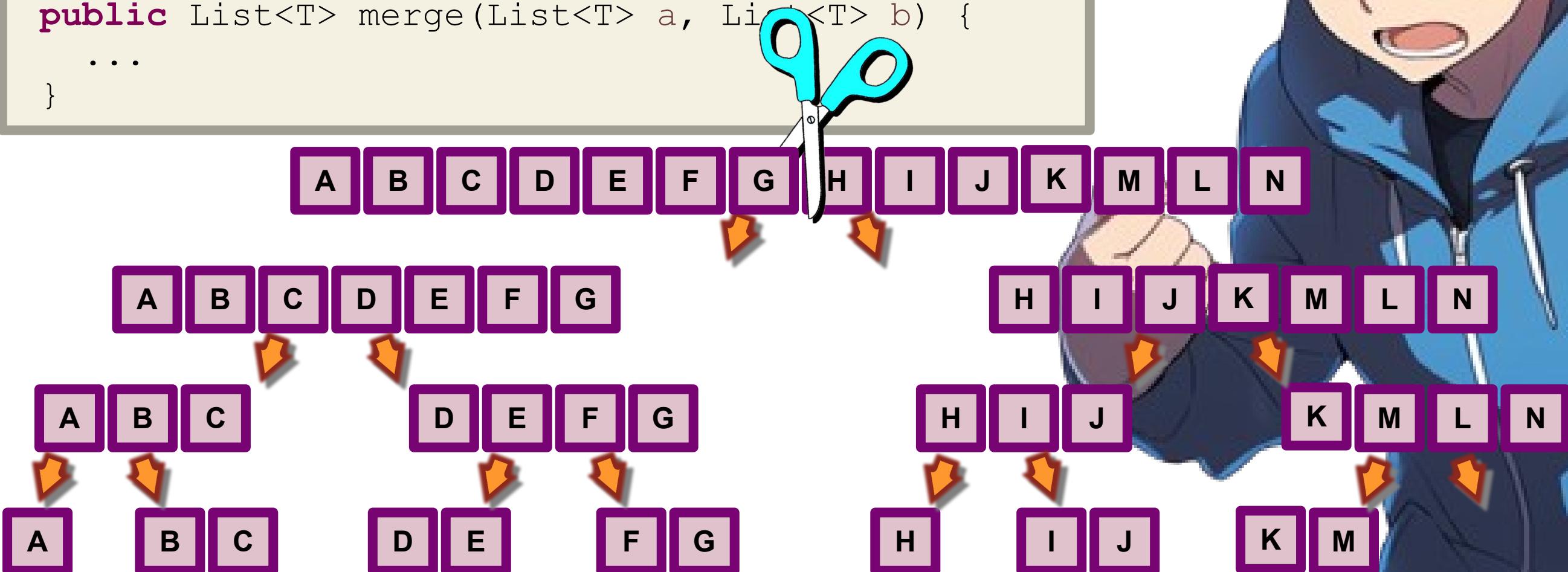
```
public List<T> sort(List<T> list) {  
    int size= list.size();  
    if (size < 2) { return list; }  
    int half= size / 2;  
    List<T> left= sort(list.subList(0, half));  
    List<T> right= sort(list.subList(half, size));  
    return merge(left, right);  
}  
  
public List<T> merge(List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



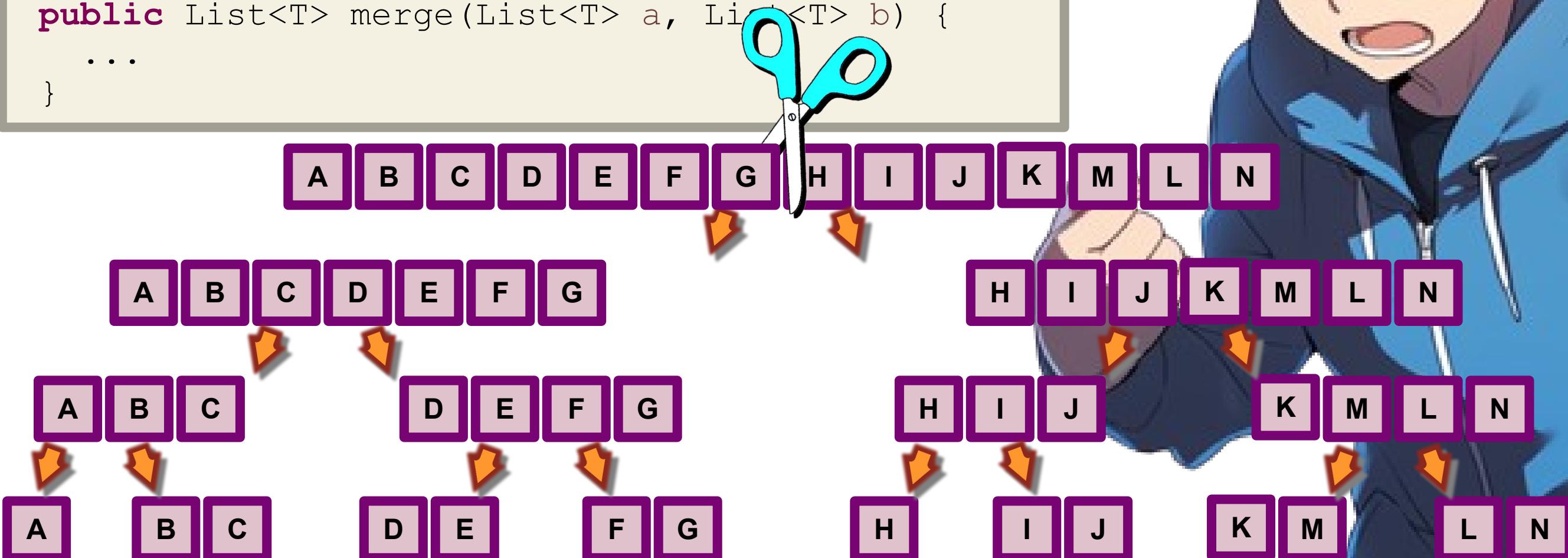
```
public List<T> sort(List<T> list) {  
    int size= list.size();  
    if (size < 2) { return list; }  
    int half= size / 2;  
    List<T> left= sort(list.subList(0, half));  
    List<T> right= sort(list.subList(half, size));  
    return merge(left, right);  
}  
  
public List<T> merge(List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



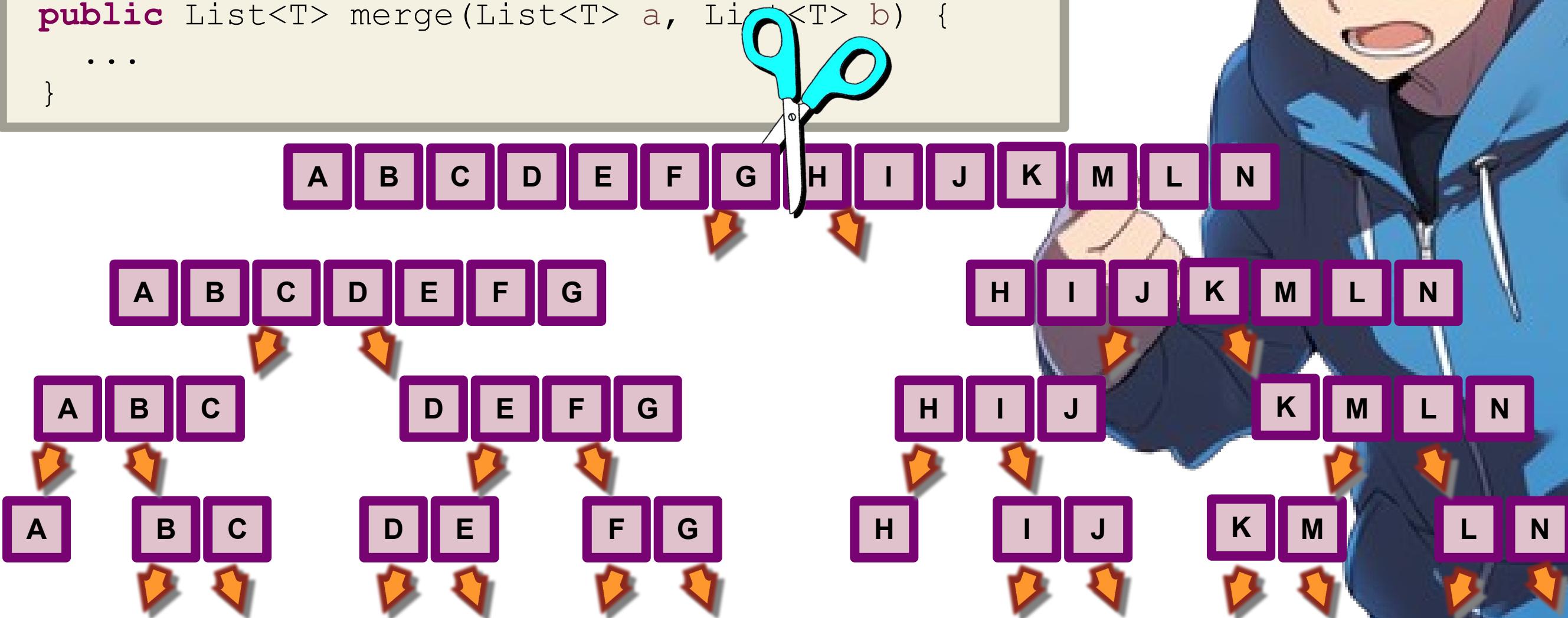
```
public List<T> sort(List<T> list) {  
    int size= list.size();  
    if (size < 2) { return list; }  
    int half= size / 2;  
    List<T> left= sort(list.subList(0, half));  
    List<T> right= sort(list.subList(half, size));  
    return merge(left, right);  
}  
  
public List<T> merge(List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



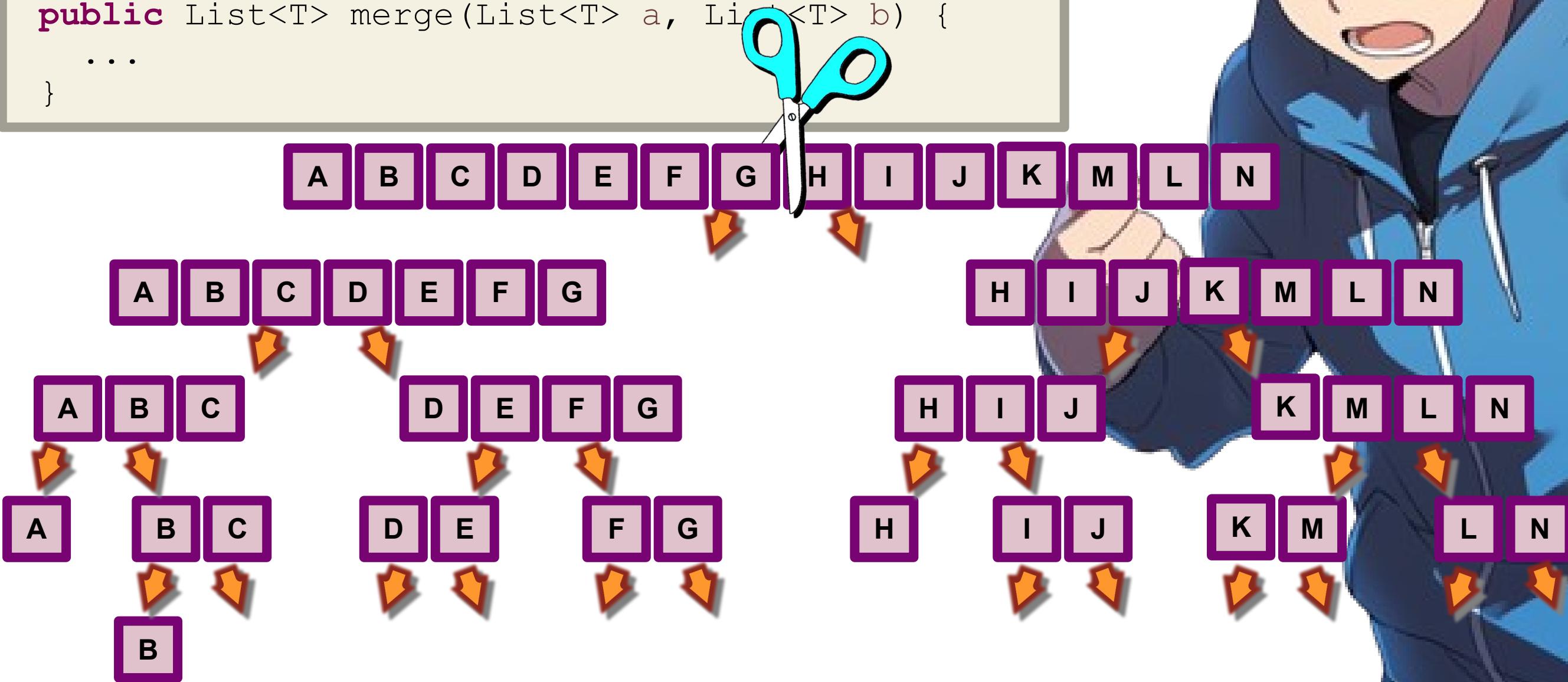
```
public List<T> sort(List<T> list) {  
    int size= list.size();  
    if (size < 2) { return list; }  
    int half= size / 2;  
    List<T> left= sort(list.subList(0, half));  
    List<T> right= sort(list.subList(half, size));  
    return merge(left, right);  
}  
  
public List<T> merge(List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



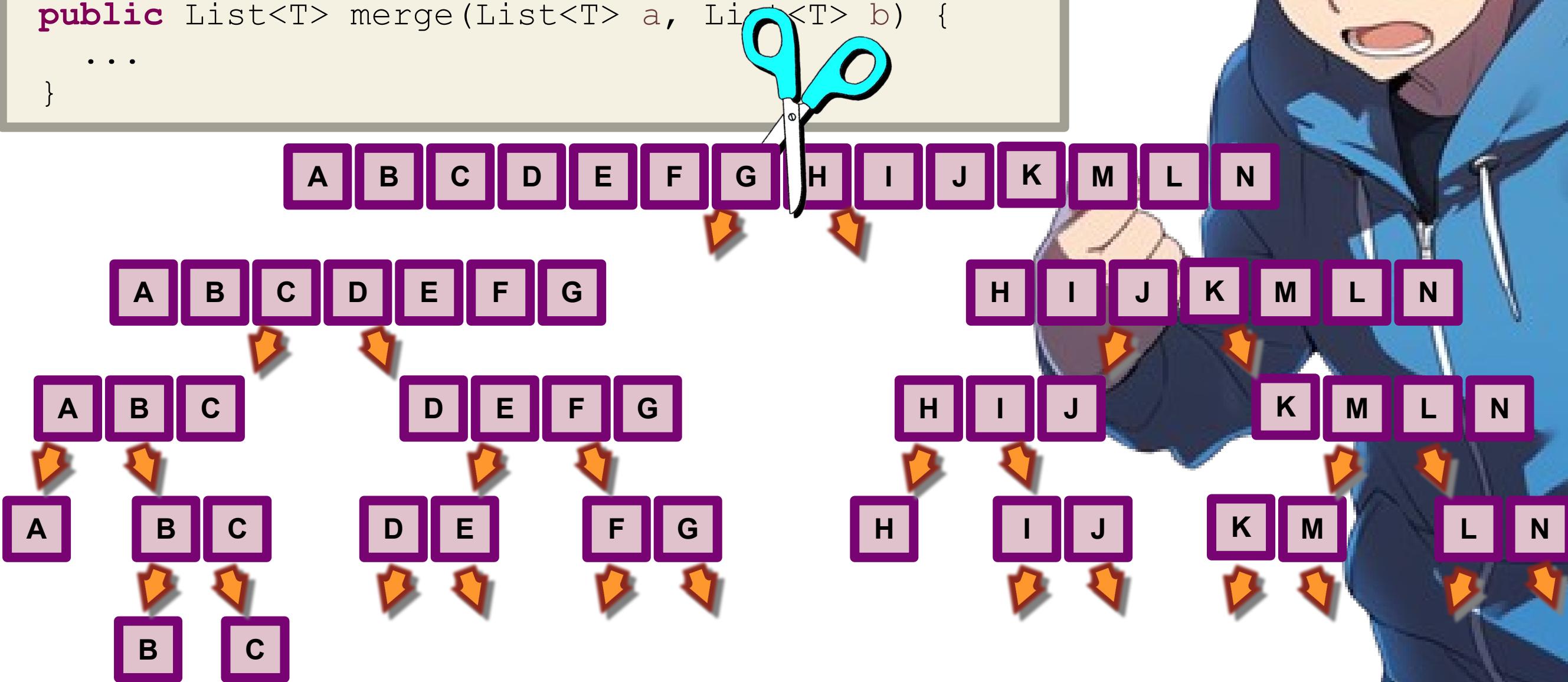
```
public List<T> sort(List<T> list) {  
    int size= list.size();  
    if (size < 2) { return list; }  
    int half= size / 2;  
    List<T> left= sort(list.subList(0, half));  
    List<T> right= sort(list.subList(half, size));  
    return merge(left, right);  
}  
  
public List<T> merge(List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



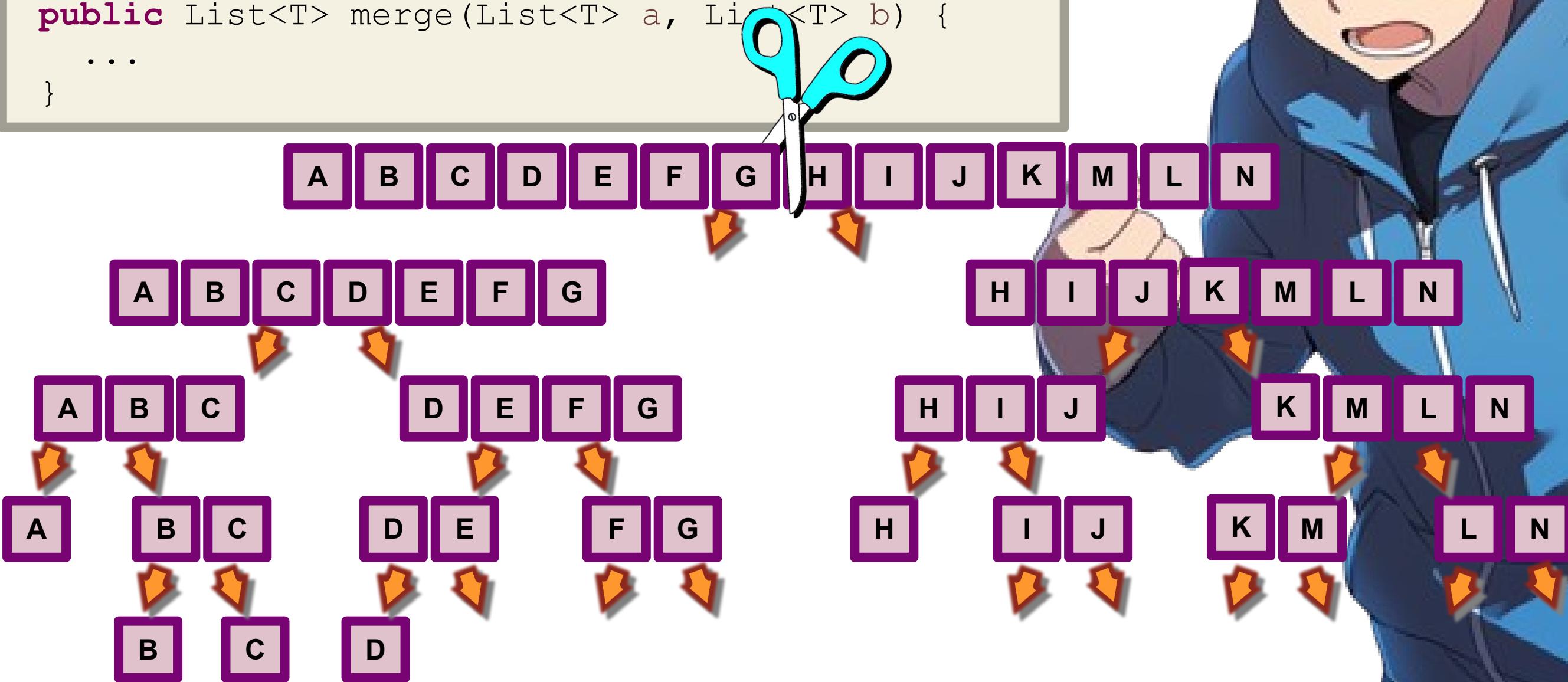
```
public List<T> sort(List<T> list) {  
    int size= list.size();  
    if (size < 2) { return list; }  
    int half= size / 2;  
    List<T> left= sort(list.subList(0, half));  
    List<T> right= sort(list.subList(half, size));  
    return merge(left, right);  
}  
  
public List<T> merge(List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



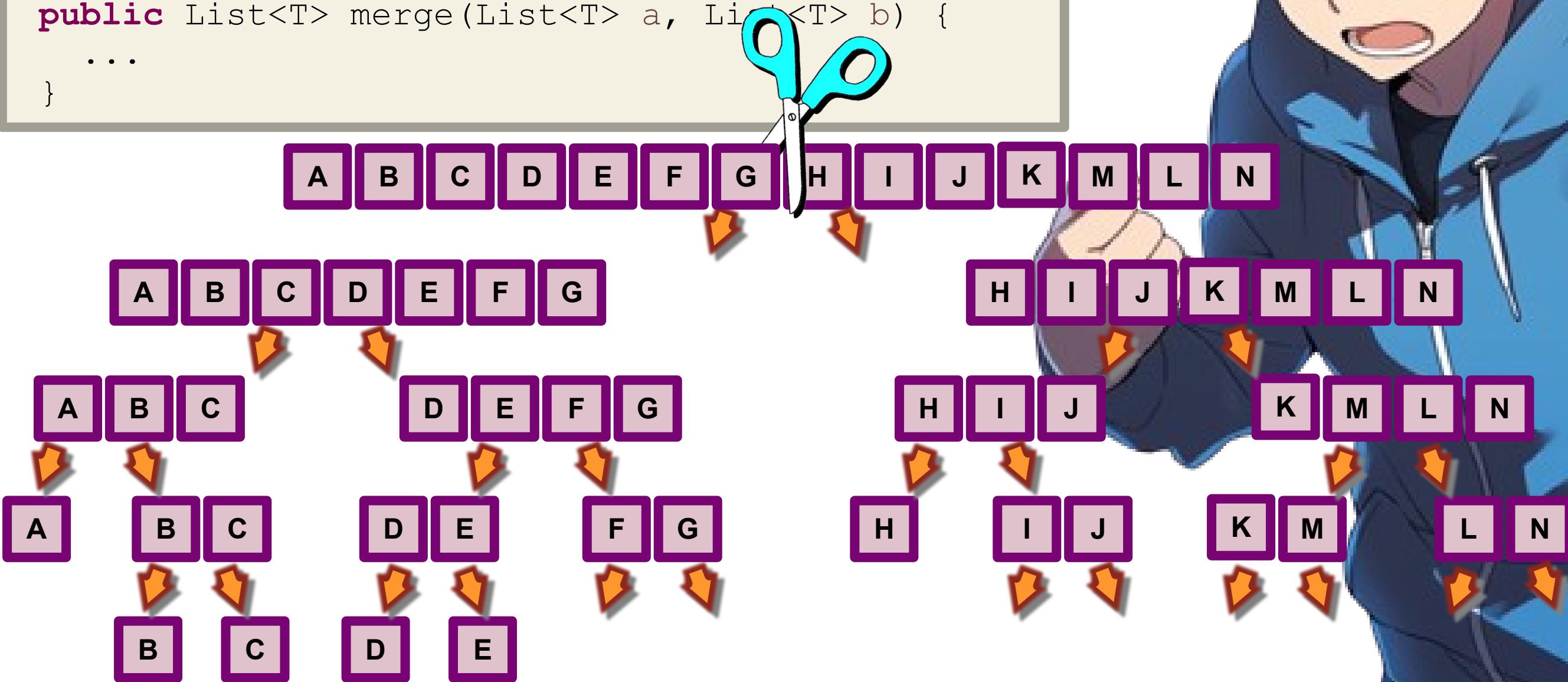
```
public List<T> sort(List<T> list) {  
    int size= list.size();  
    if (size < 2) { return list; }  
    int half= size / 2;  
    List<T> left= sort(list.subList(0, half));  
    List<T> right= sort(list.subList(half, size));  
    return merge(left, right);  
}  
  
public List<T> merge(List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



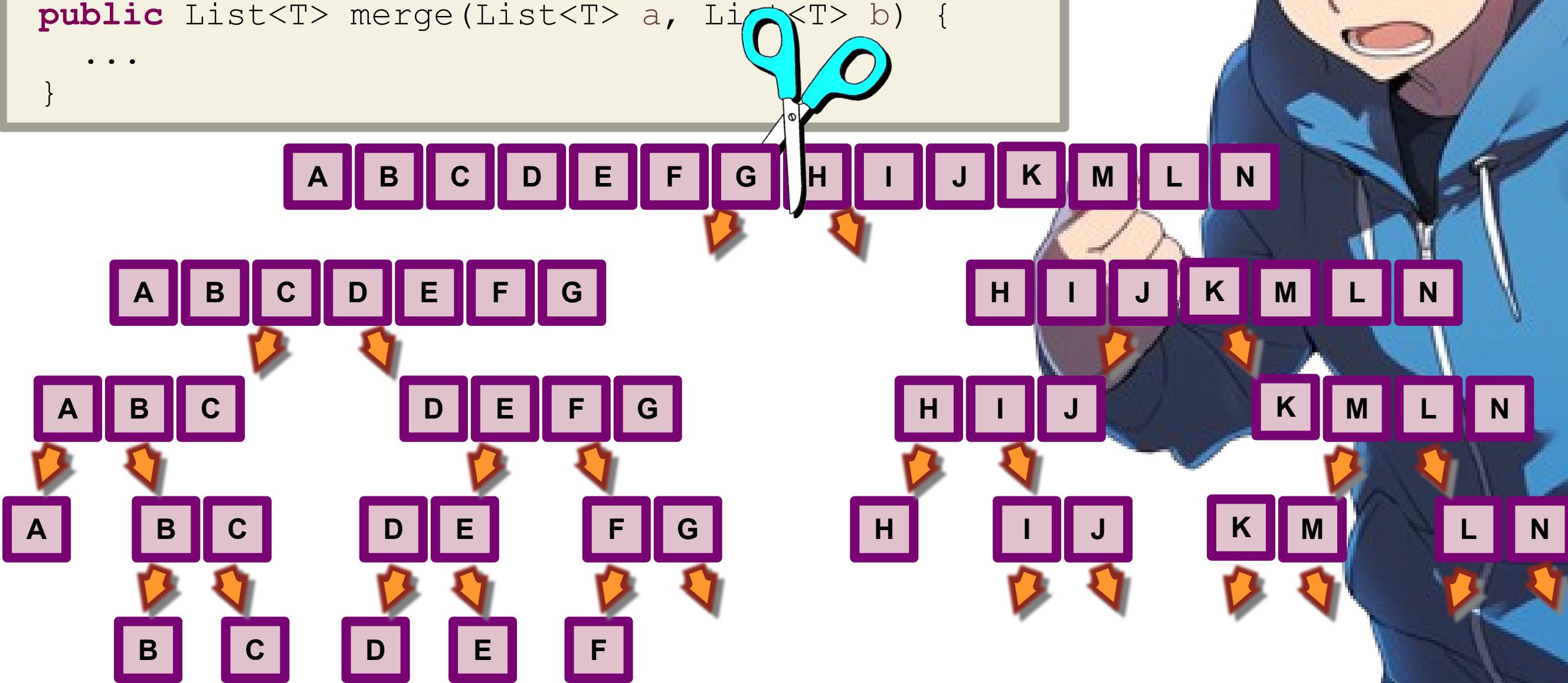
```
public List<T> sort(List<T> list) {  
    int size= list.size();  
    if (size < 2) { return list; }  
    int half= size / 2;  
    List<T> left= sort(list.subList(0, half));  
    List<T> right= sort(list.subList(half, size));  
    return merge(left, right);  
}  
  
public List<T> merge(List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



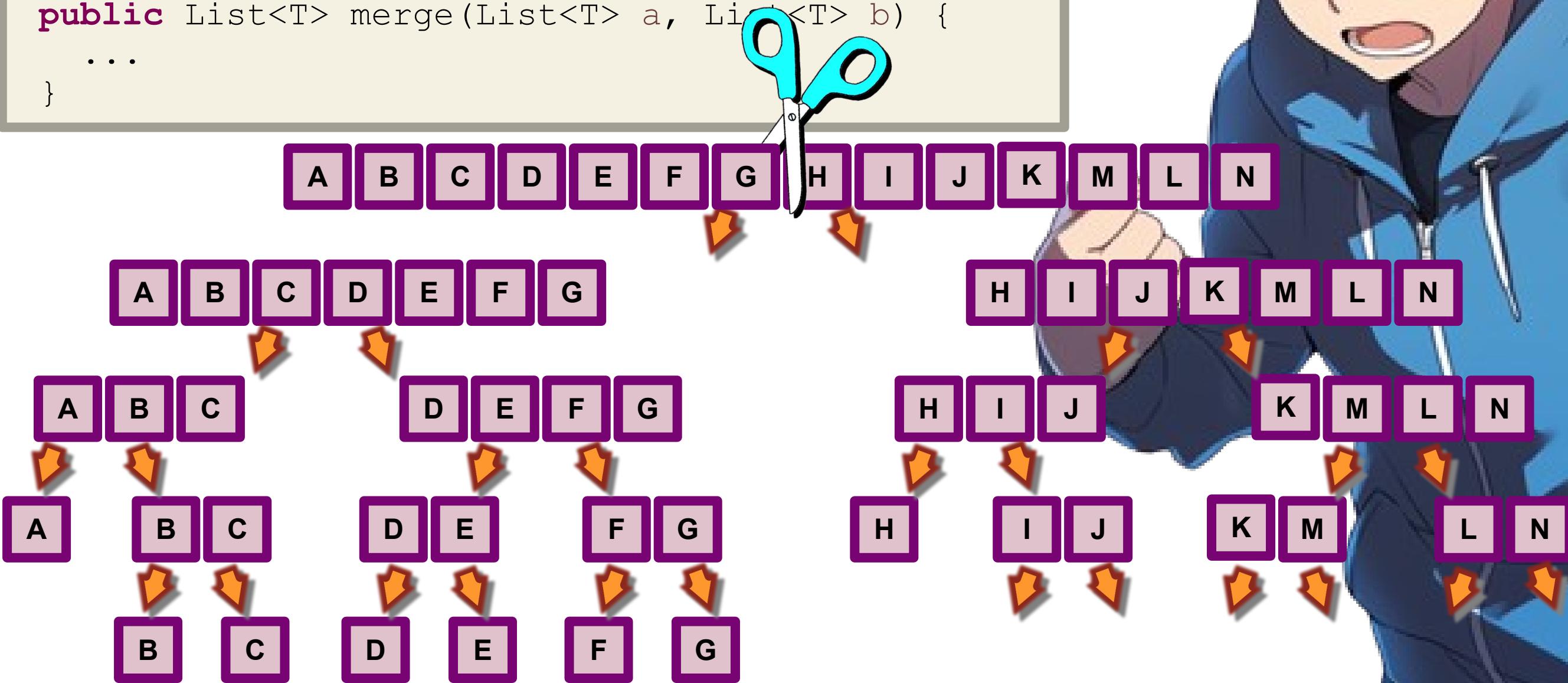
```
public List<T> sort(List<T> list) {  
    int size= list.size();  
    if (size < 2) { return list; }  
    int half= size / 2;  
    List<T> left= sort(list.subList(0, half));  
    List<T> right= sort(list.subList(half, size));  
    return merge(left, right);  
}  
  
public List<T> merge(List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



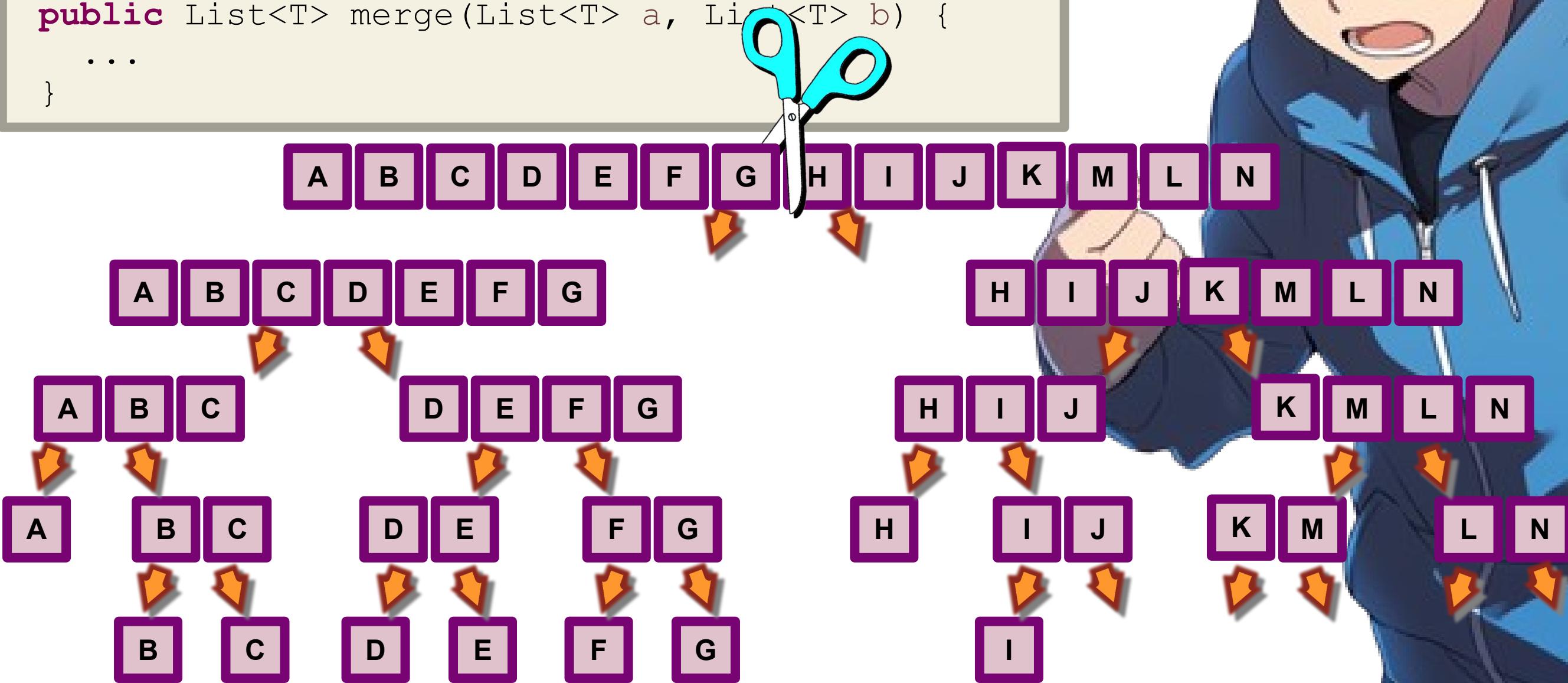
```
public List<T> sort(List<T> list) {  
    int size= list.size();  
    if (size < 2) { return list; }  
    int half= size / 2;  
    List<T> left= sort(list.subList(0, half));  
    List<T> right= sort(list.subList(half, size));  
    return merge(left, right);  
}  
  
public List<T> merge(List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



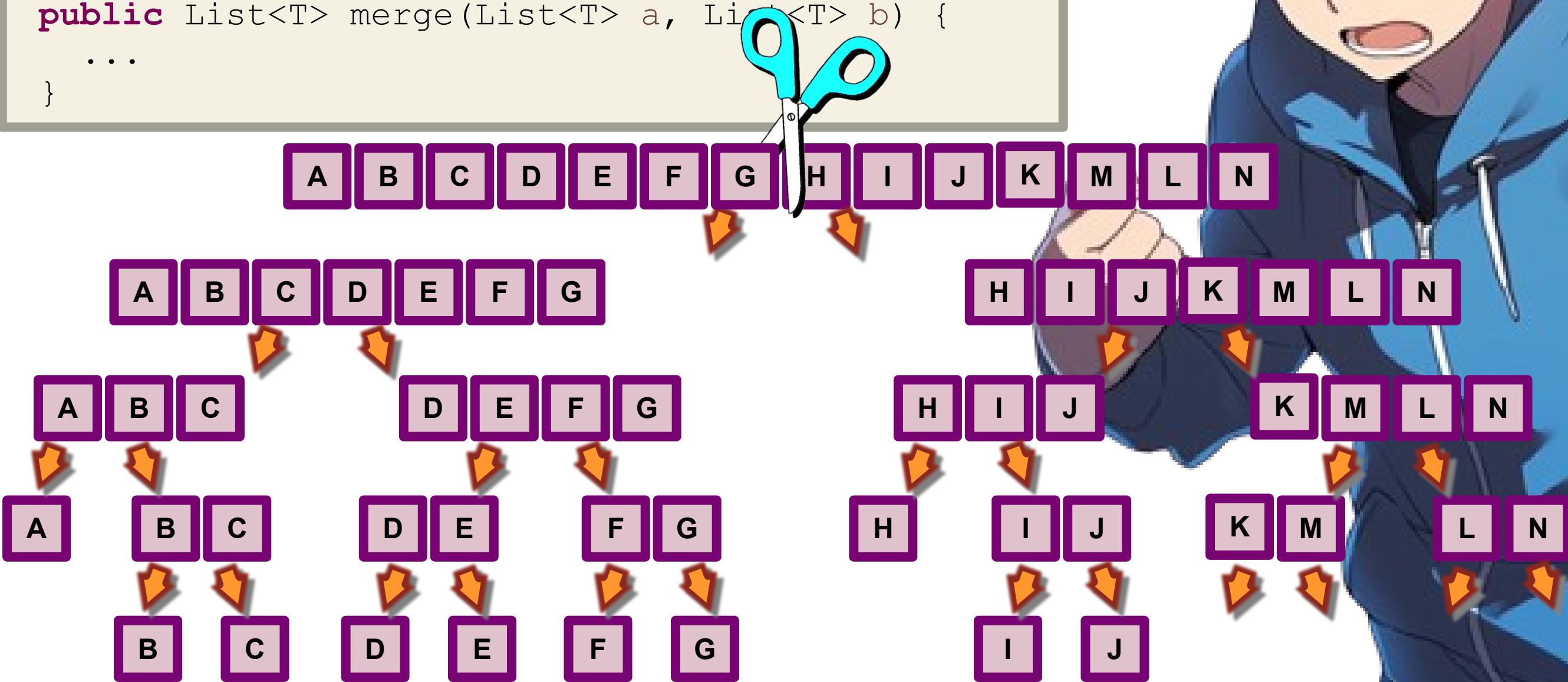
```
public List<T> sort(List<T> list) {  
    int size= list.size();  
    if (size < 2) { return list; }  
    int half= size / 2;  
    List<T> left= sort(list.subList(0, half));  
    List<T> right= sort(list.subList(half, size));  
    return merge(left, right);  
}  
  
public List<T> merge(List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



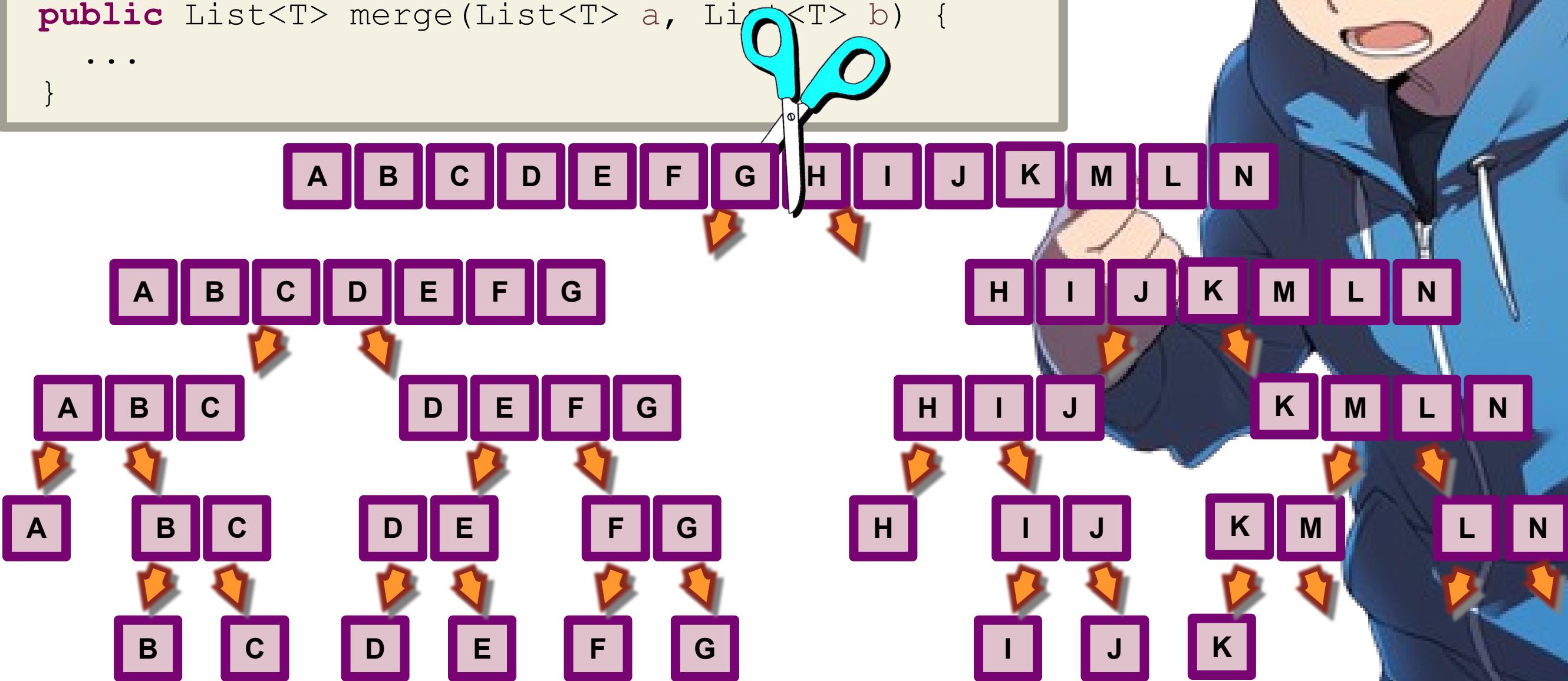
```
public List<T> sort(List<T> list) {  
    int size= list.size();  
    if (size < 2) { return list; }  
    int half= size / 2;  
    List<T> left= sort(list.subList(0, half));  
    List<T> right= sort(list.subList(half, size));  
    return merge(left, right);  
}  
  
public List<T> merge(List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



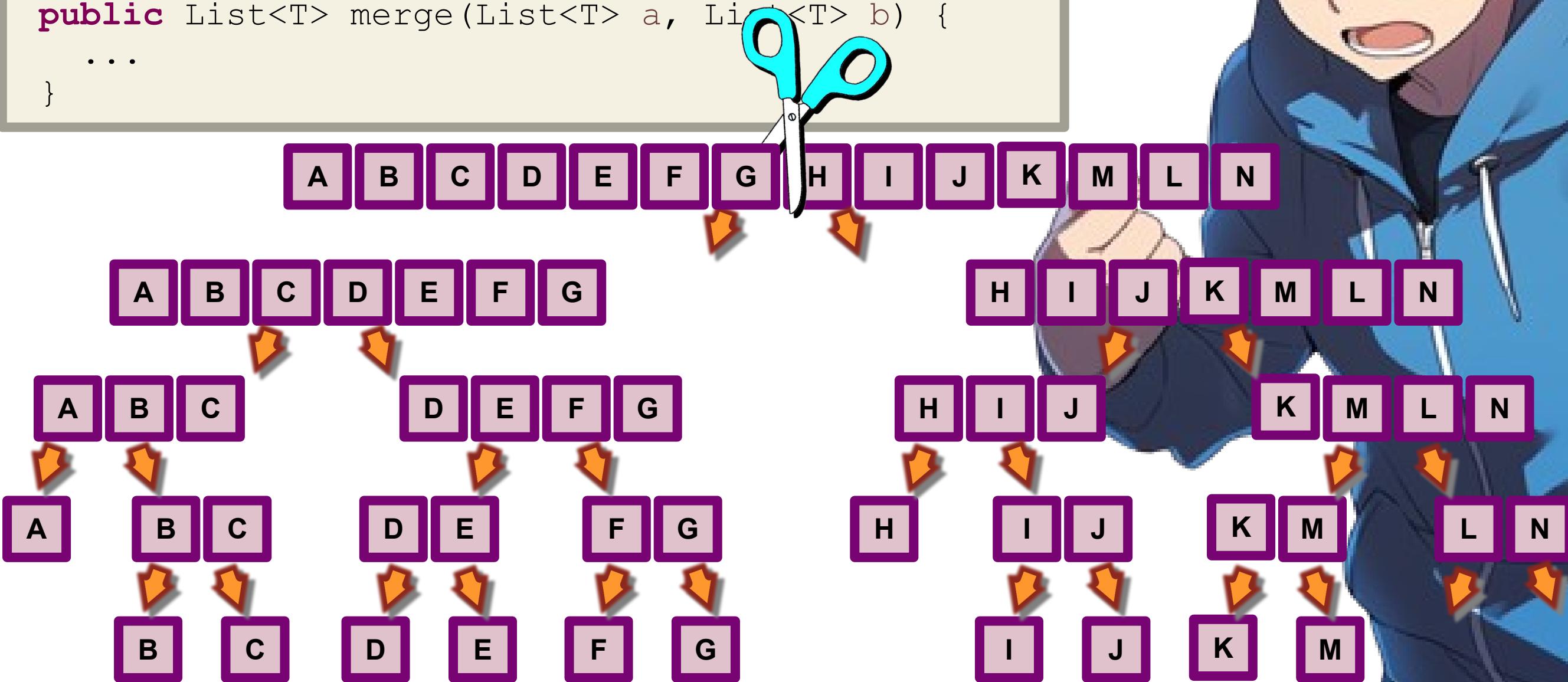
```
public List<T> sort(List<T> list) {  
    int size= list.size();  
    if (size < 2) { return list; }  
    int half= size / 2;  
    List<T> left= sort(list.subList(0, half));  
    List<T> right= sort(list.subList(half, size));  
    return merge(left, right);  
}  
  
public List<T> merge(List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



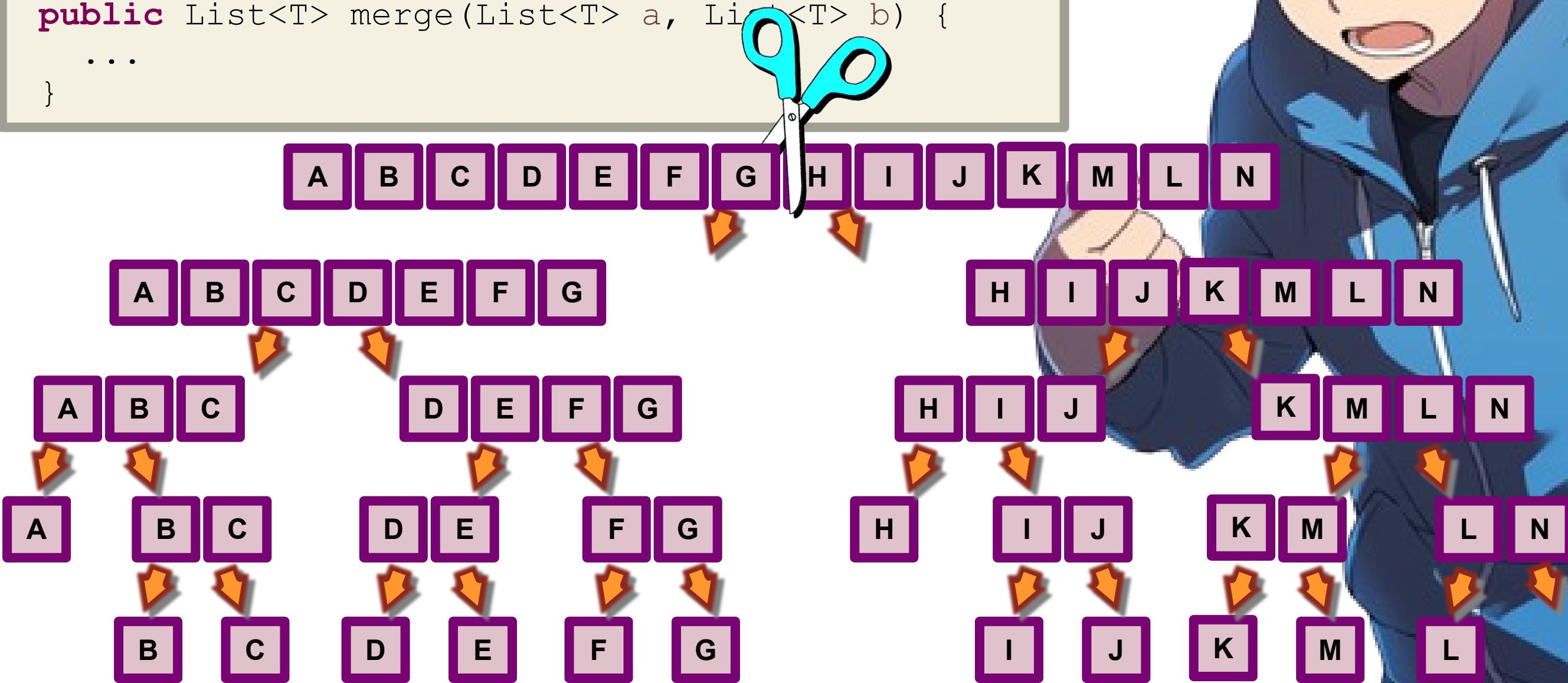
```
public List<T> sort(List<T> list) {  
    int size= list.size();  
    if (size < 2) { return list; }  
    int half= size / 2;  
    List<T> left= sort(list.subList(0, half));  
    List<T> right= sort(list.subList(half, size));  
    return merge(left, right);  
}  
  
public List<T> merge(List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



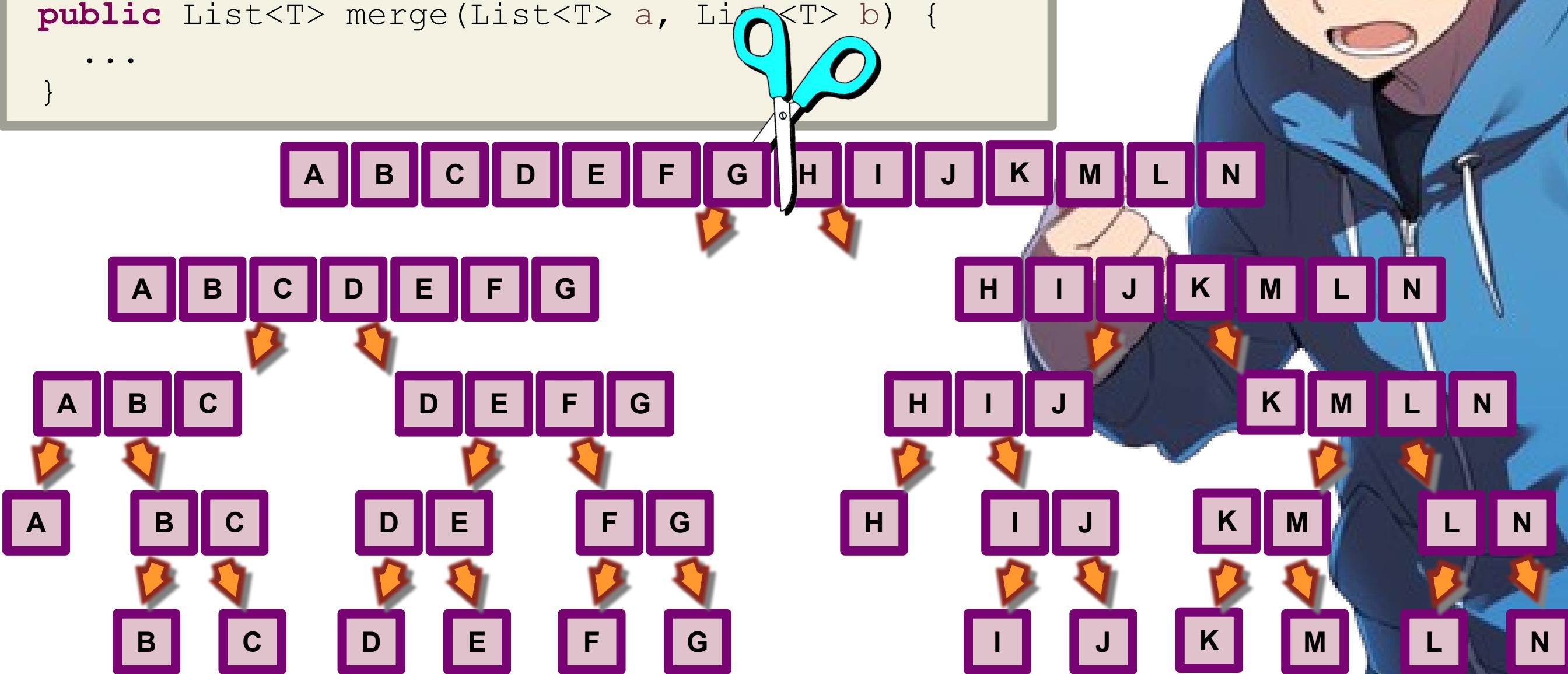
```
public List<T> sort(List<T> list) {  
    int size= list.size();  
    if (size < 2) { return list; }  
    int half= size / 2;  
    List<T> left= sort(list.subList(0, half));  
    List<T> right= sort(list.subList(half, size));  
    return merge(left, right);  
}  
  
public List<T> merge(List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



```
public List<T> 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) {  
    ...  
}
```

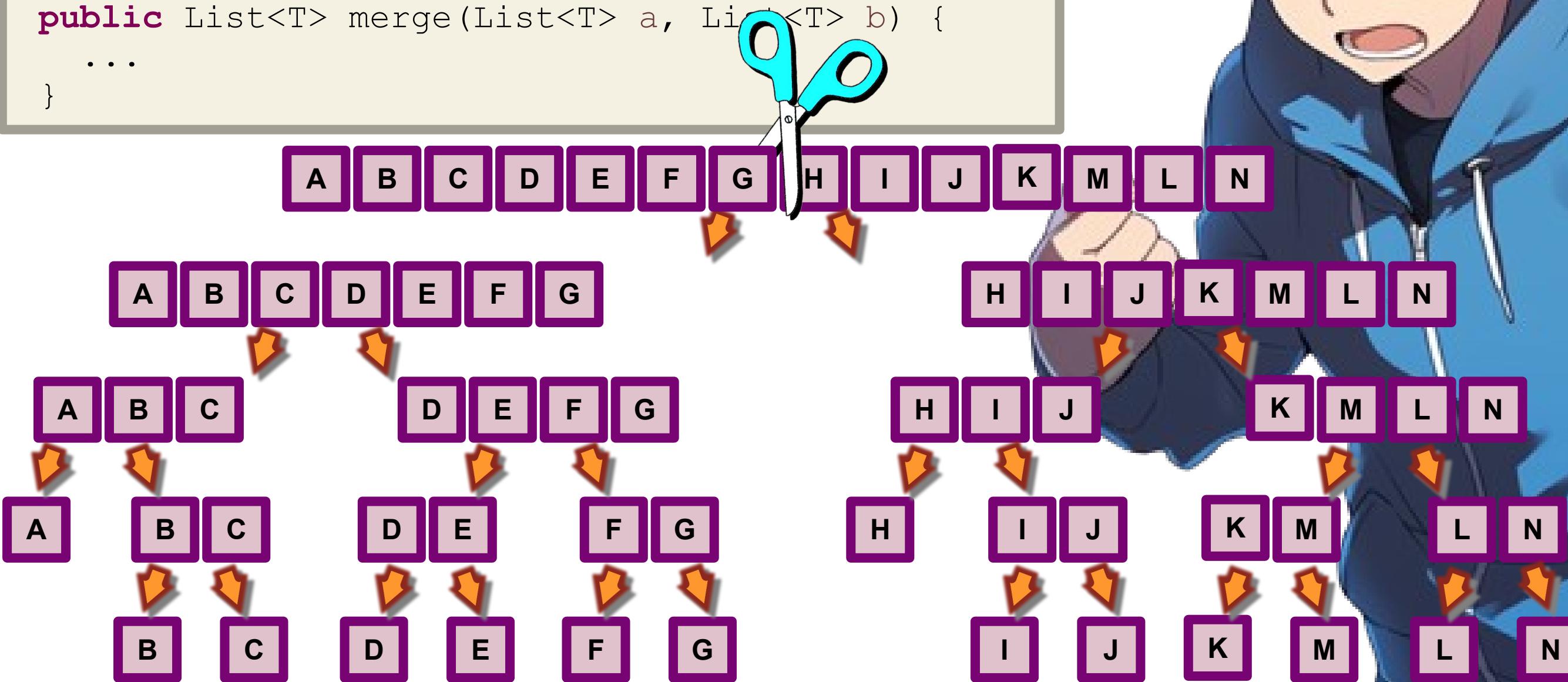


```
public List<T> sort(List<T> list) {  
    int size= list.size();  
    if (size < 2) { return list; }  
    int half= size / 2;  
    List<T> left= sort(list.subList(0, half));  
    List<T> right= sort(list.subList(half, size));  
    return merge(left, right);  
}  
  
public List<T> merge(List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



```
public List<T> 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) {  
    ...  
}
```

We can visualize like this!

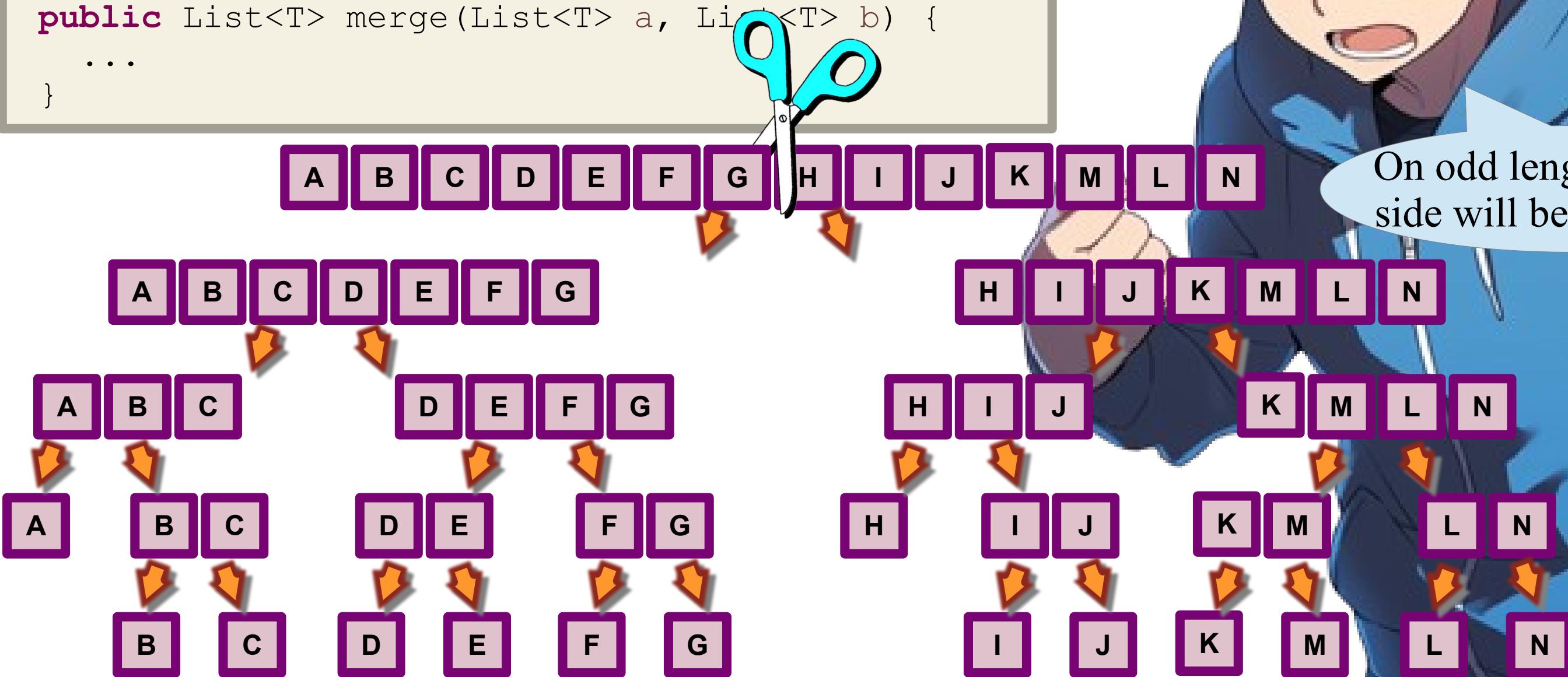


```

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) {
    ...
}

```

We can visualize like this!



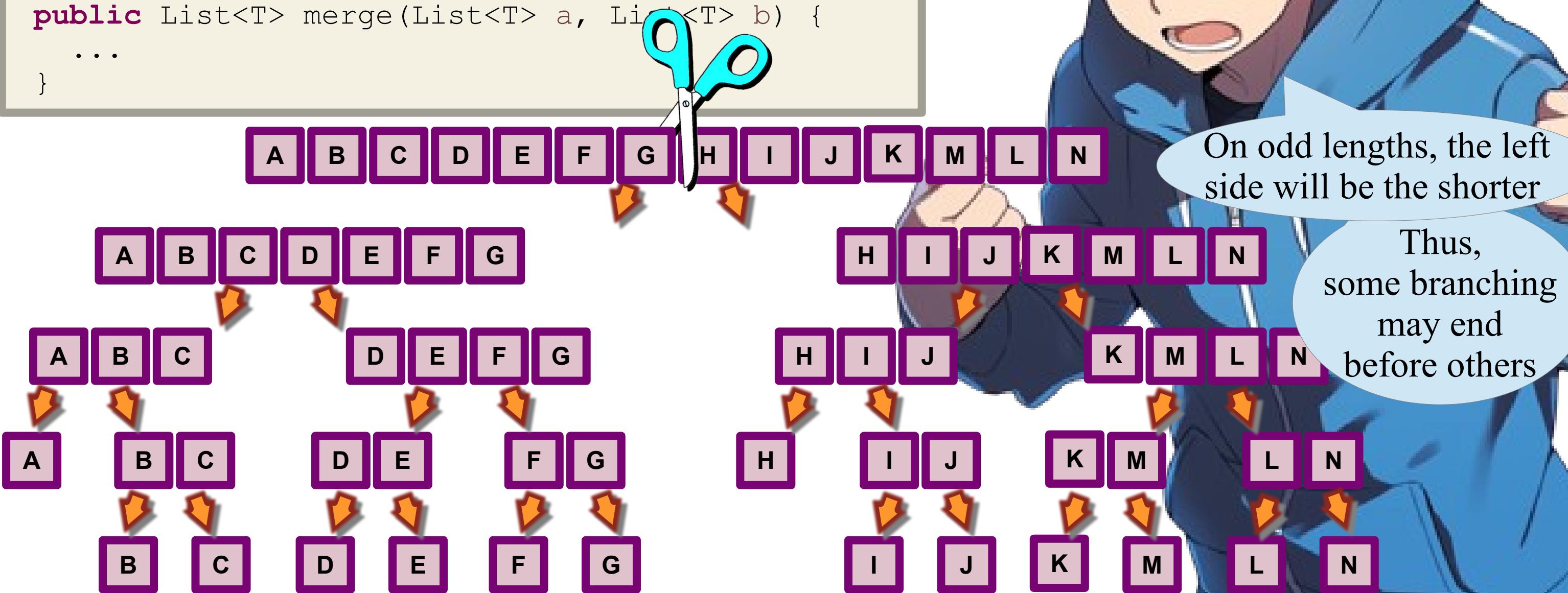
On odd lengths, the left side will be the shorter

```

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) {
    ...
}

```

We can visualize like this!







Sort cuts into pieces



Sort cuts into pieces

Merge glues two
pieces together



```
public List<T> merge(List<T> a, List<T> b) {
```

```
}
```



How can we write merge?

```
public List<T> merge(List<T> a, List<T> b) {
```

```
}
```

```
public List<T> merge(List<T> a, List<T> b) {
```



How can we write merge?

Merge works on two
non empty sorted lists

```
}
```



How can we write merge?

Merge works on two non empty sorted lists

```
public List<T> merge(List<T> a, List<T> b) {  
    List<T> res= new ArrayList<>(a.size() + b.size());  
  
    return res;  
}
```



How can we write merge?

Merge works on two non empty sorted lists

Merge returns a sorted list with all the elements of a and b

```
public List<T> merge(List<T> a, List<T> b) {  
    List<T> res= new ArrayList<>(a.size() + b.size());  
  
    return res;  
}
```



How can we write merge?

Merge works on two non empty sorted lists

Merge returns a sorted list with all the elements of a and b

So we can allocate the result list with space for the elements of the two lists

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

Merge works on two
non empty sorted lists.
We can rely on this property!



```
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 (                                ??? ) {  
        T currA= a.get(indexA);  
        T currB= b.get(indexB);  
        if (currA < currB) {  
            res.add(currA);  
            indexA++;  
        } else {  
            res.add(currB);  
            indexB++;  
        }  
    }  
    return res;  
}
```



Merge works on two
non empty sorted lists.
We can rely on this property!

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 () {
 T currA= a.get(indexA);
 T currB= b.get(indexB);
 ???
 return res;
 }
}

A cartoon illustration of a character with blue hair and glasses, looking shocked or surprised. A white speech bubble originates from their mouth.

Merge works on two
non empty sorted lists.
We can rely on this property!

```
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 (                                ??? ) {  
        T currA= a.get(indexA);  
        T currB= b.get(indexB);  
  
        if (currA < currB) {  
            res.add(currA);  
            indexA++;  
        } else {  
            res.add(currB);  
            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 (                                ??? ) {  
        T currA= a.get(indexA);  
        T currB= b.get(indexB);  
        if (currA.compareTo(currB) <= 0) {  
            res.add(currA);  
            indexA++;  
        }  
        else {  
            }  
    }  
  
    return res;  
}
```



We compare currA and currB

```
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 (                                ??? ) {  
        T currA= a.get(indexA);  
        T currB= b.get(indexB);  
        if (currA.compareTo(currB) <= 0) {  
            res.add(currA);  
            indexA++;  
        }  
        else {  
            }  
    }  
  
    return res;  
}
```



We compare currA and currB

```
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 (                                ??? ) {  
        T currA= a.get(indexA);  
        T currB= b.get(indexB);  
        if (currA.compareTo(currB) <= 0) {  
            res.add(currA);  
            indexA++;  
        }  
        else {  
            return res;  
        }  
    }  
}
```



We compare currA and currB

```
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 (                                ??? ) {  
        T currA= a.get(indexA);  
        T currB= b.get(indexB);  
        if (currA.compareTo(currB) <= 0) {  
            res.add(currA);  
            indexA++;  
        }  
        else {  
            res.add(currB);  
            indexB++;  
        }  
    }  
    return res;  
}
```



If currA is the smaller,

we add it to the result
and we move indexA forward

We compare currA and currB

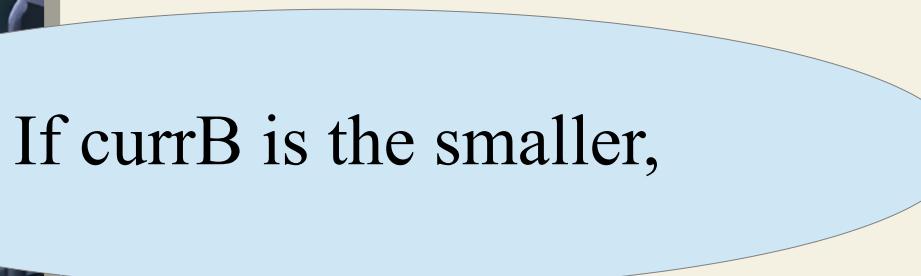


```
public List<T> merge(List<T> a, List<T> b) {  
    List<T> res= new ArrayList<>(a.size() + b.size());  
    int indexA= 0;  
    int indexB= 0;  
    while (                                ??? ) {  
        T currA= a.get(indexA);  
        T currB= b.get(indexB);  
        if (currA.compareTo(currB)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



We compare currA and currB

```
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 (                                ??? ) {  
        T currA= a.get(indexA);  
        T currB= b.get(indexB);  
        if (currA.compareTo(currB) <= 0) {  
            res.add(currA);  
            indexA++;  
        }  
        else {  
            res.add(currB);  
            indexB++;  
        }  
    }  
    return res;  
}
```

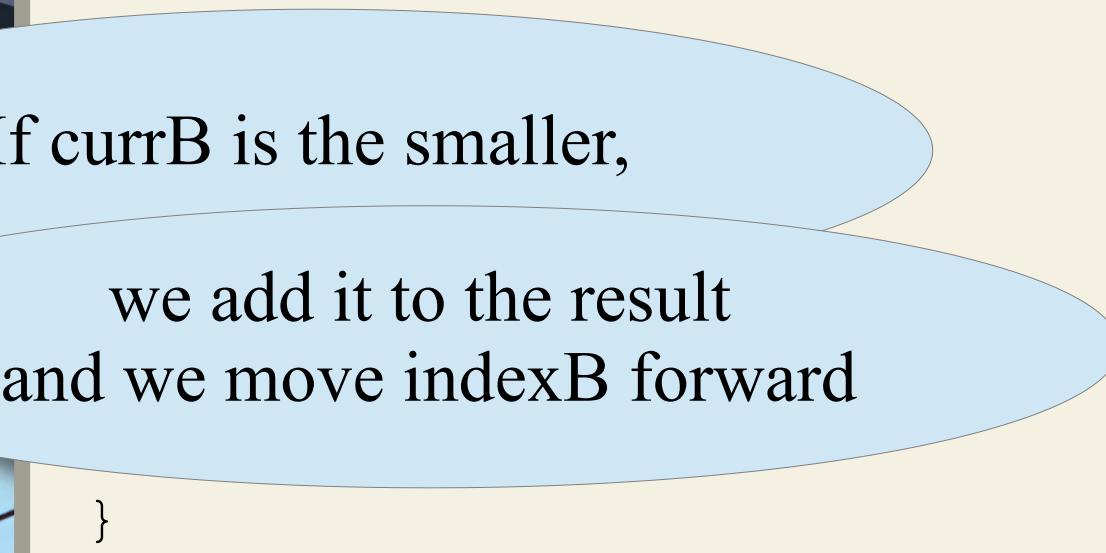


If currB is the smaller,



We compare currA and currB

```
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 (                                ??? ) {  
        T currA= a.get(indexA);  
        T currB= b.get(indexB);  
        if (currA.compareTo(currB) <= 0) {  
            res.add(currA);  
            indexA++;  
        }  
        else {  
            res.add(currB);  
            indexB++;  
        }  
    }  
}
```



If currB is the smaller,
we add it to the result
and we move indexB forward



```
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 (                                ??? ) {  
        T currA= a.get(indexA);  
        T currB= b.get(indexB);  
        if (currA.compareTo(currB) <= 0) {  
            res.add(currA);  
            indexA++;  
        }  
        else {  
            res.add(currB);  
            indexB++;  
        }  
    }  
  
    return res;  
}
```



Now, where we stop our while?

```
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 (                                ??? ) {  
        T currA= a.get(indexA);  
        T currB= b.get(indexB);  
        if (currA.compareTo(currB) <= 0) {  
            res.add(currA);  
            indexA++;  
        }  
        else {  
            res.add(currB);  
            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++;  
        }  
    }  
  
    return res;  
}
```

We need to access a and b correctly, so ...



```
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++;  
        }  
    }  
  
    return res;  
}
```



We need to access a and b correctly, so ...

Loop until either one goes out of bounds

```
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++;  
        }  
    }  
  
    return res;  
}
```

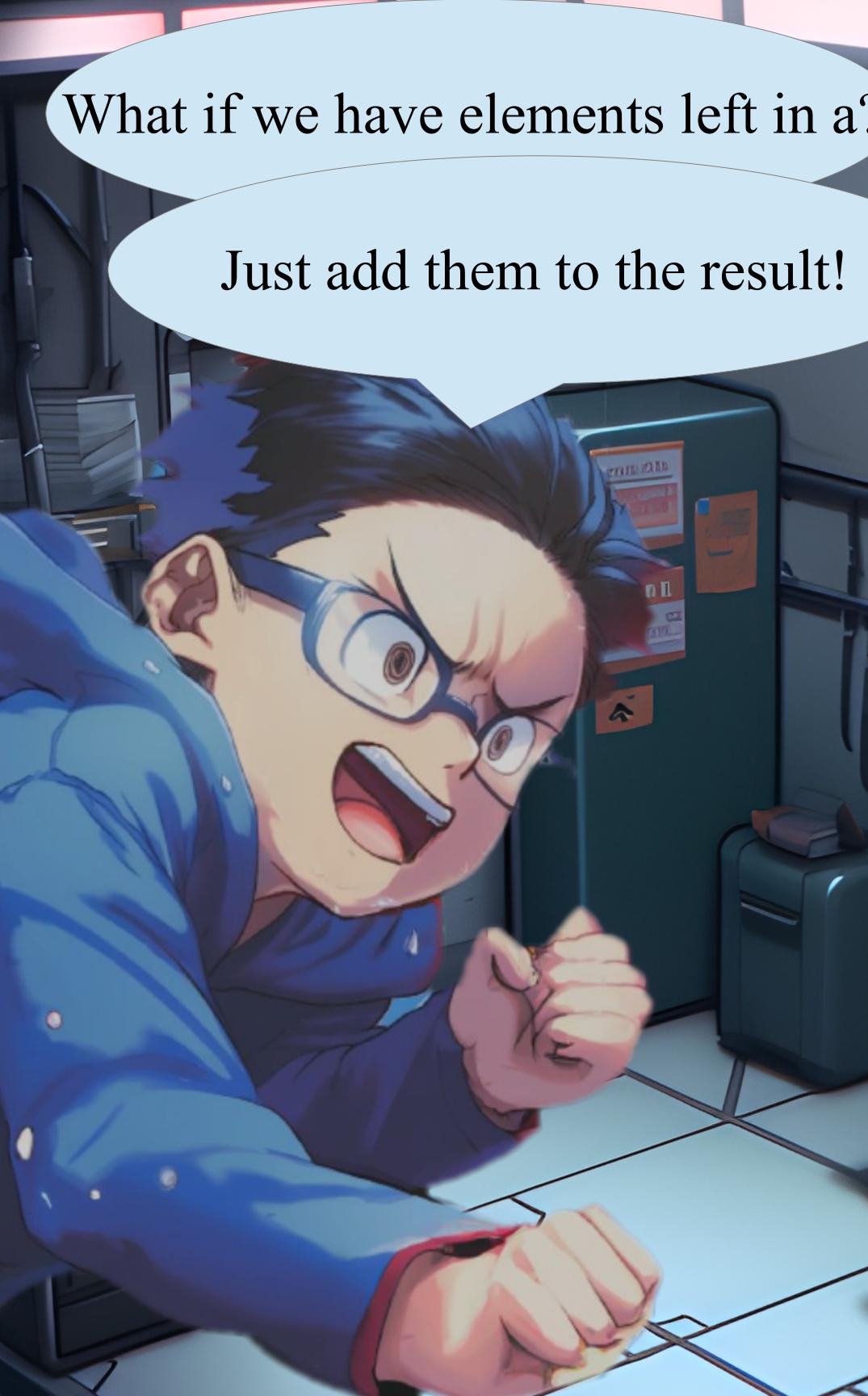


```
public List<T> merge(List<T> a, List<T> b) {  
    List<T> res= new ArrayList<>(a.size() + b.size());  
    int indexA= 0;  
    int indexB= 0;  
    while (indexA < a.size() && indexB < b.size()) {  
        T currA= a.get(indexA);  
        T currB= b.get(indexB);  
        if (currA.compareTo(currB)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

What if we have elements left in a?



```
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++;  
    }  
  
    return res;  
}
```

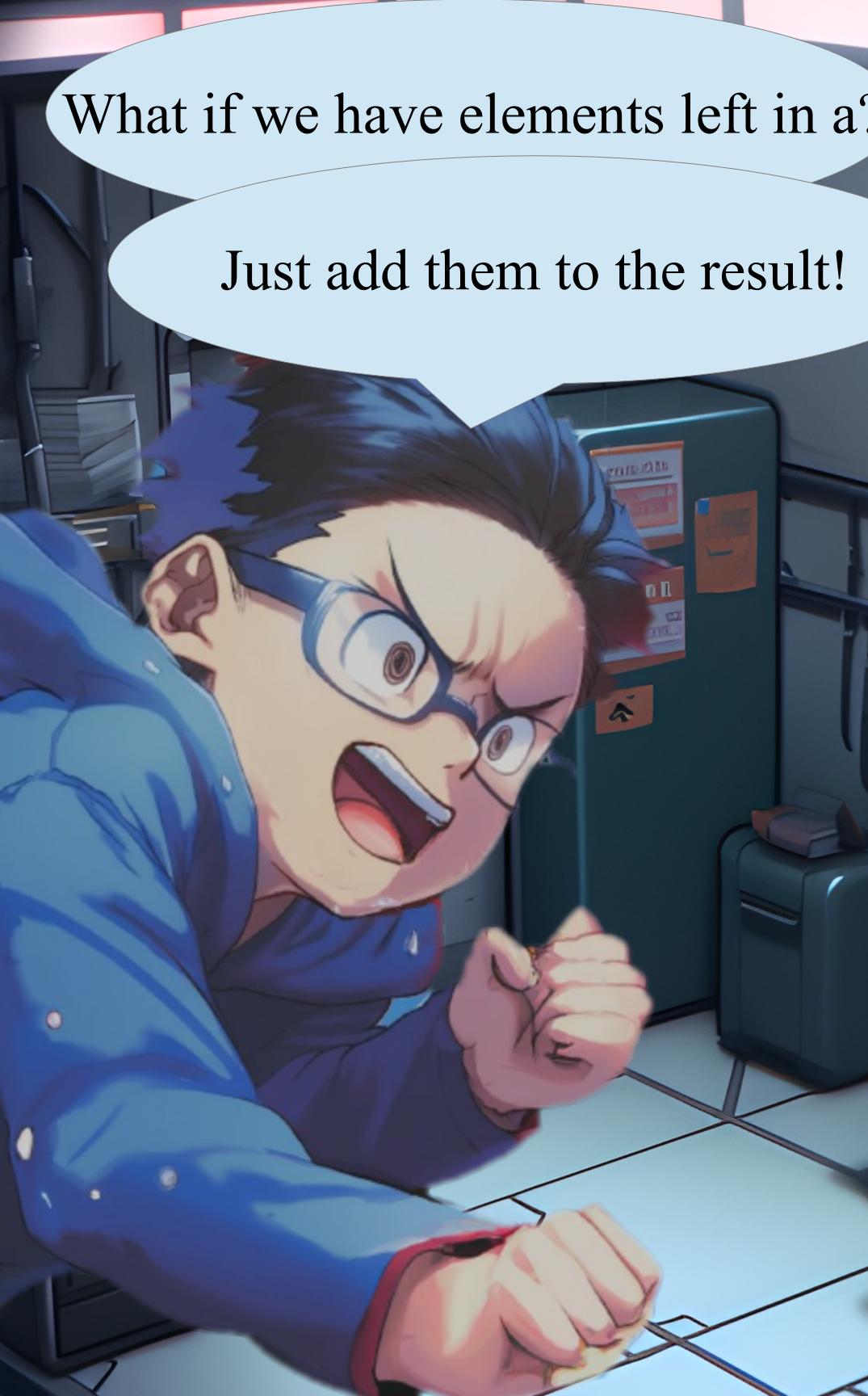


What if we have elements left in a?

Just add them to the result!

```
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++;  
    }  
  
    return res;  
}
```

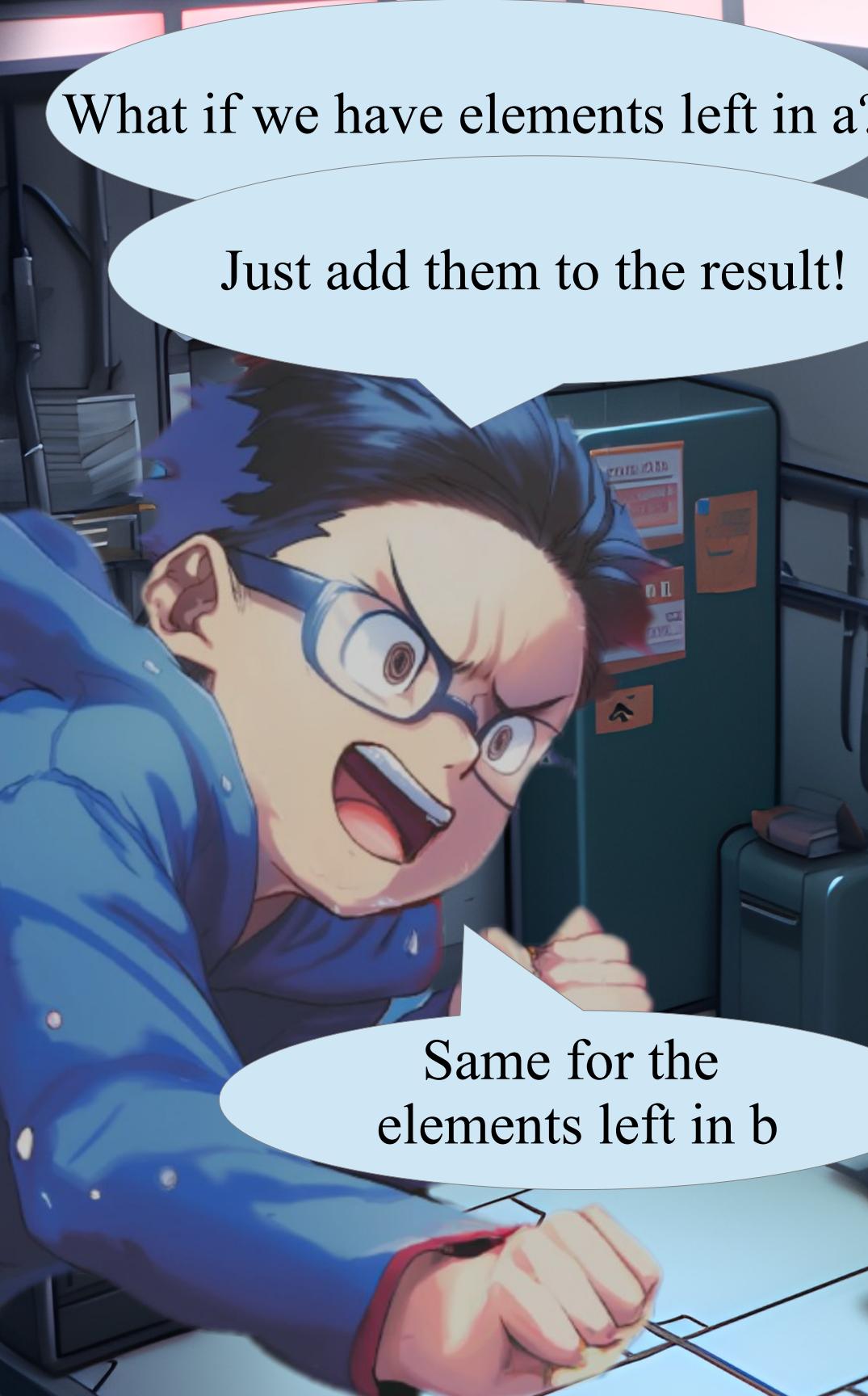




What if we have elements left in a?

Just add them to the result!

```
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;  
}
```



What if we have elements left in a?

Just add them to the result!

Same for the
elements left in 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;  
}
```



What if we have elements left in a?

Just add them to the result!

Same for the
elements left in b

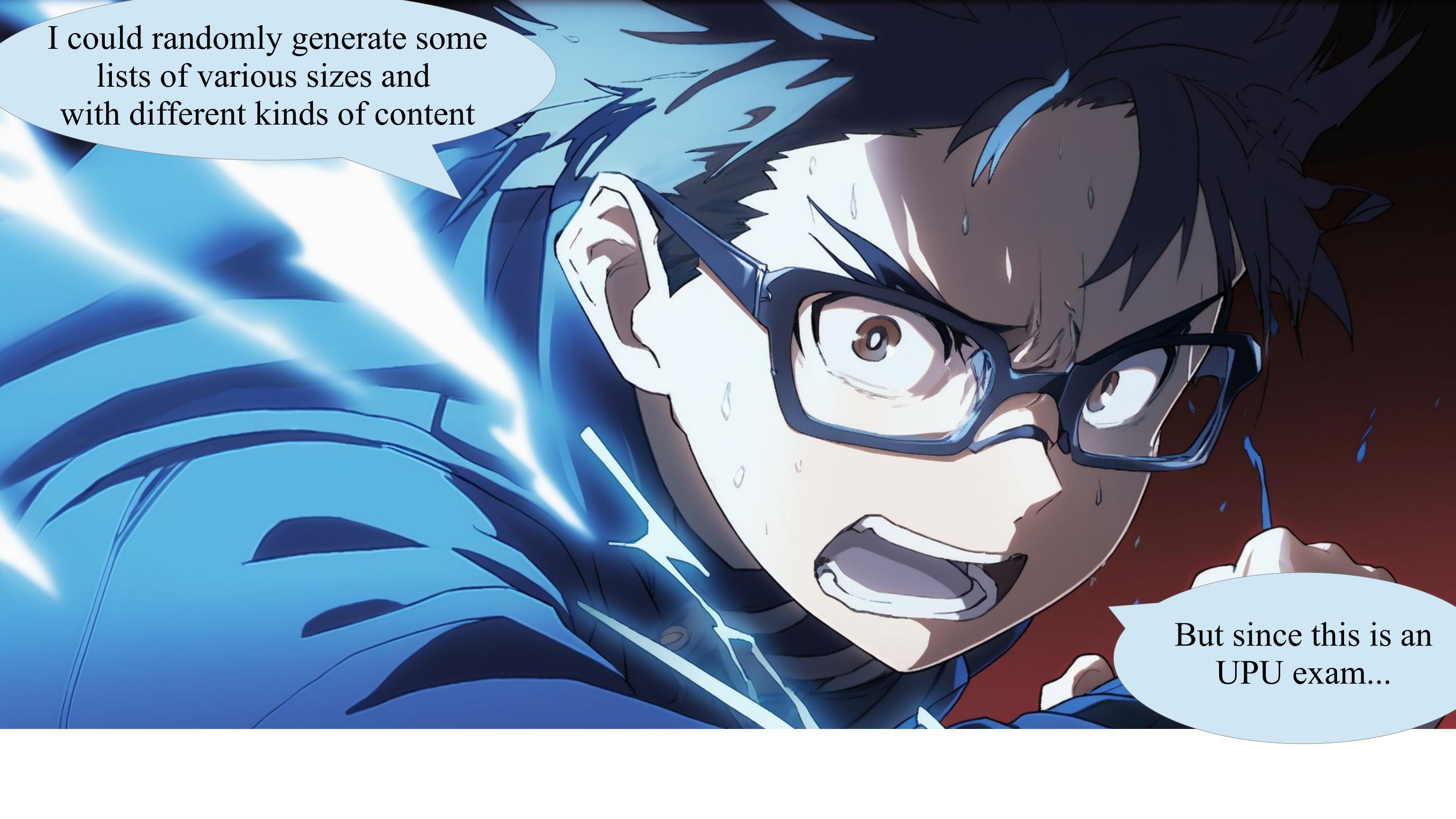
Only one of the last
two whiles will run

```
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 code looks complete

Ideally, I should compile it
and test it extensively.



I could randomly generate some lists of various sizes and with different kinds of content

But since this is an UPU exam...



I can not compile it,
let alone run tests.

I can only submit
it as is!



***Submission in
progress!***





*Submission in
progress!*

*Oh, no!
Your submission*



*Submission in
progress!*

*Oh, no!
Your submission*

*It does not even
met the minimum
requirements
to be marked!*

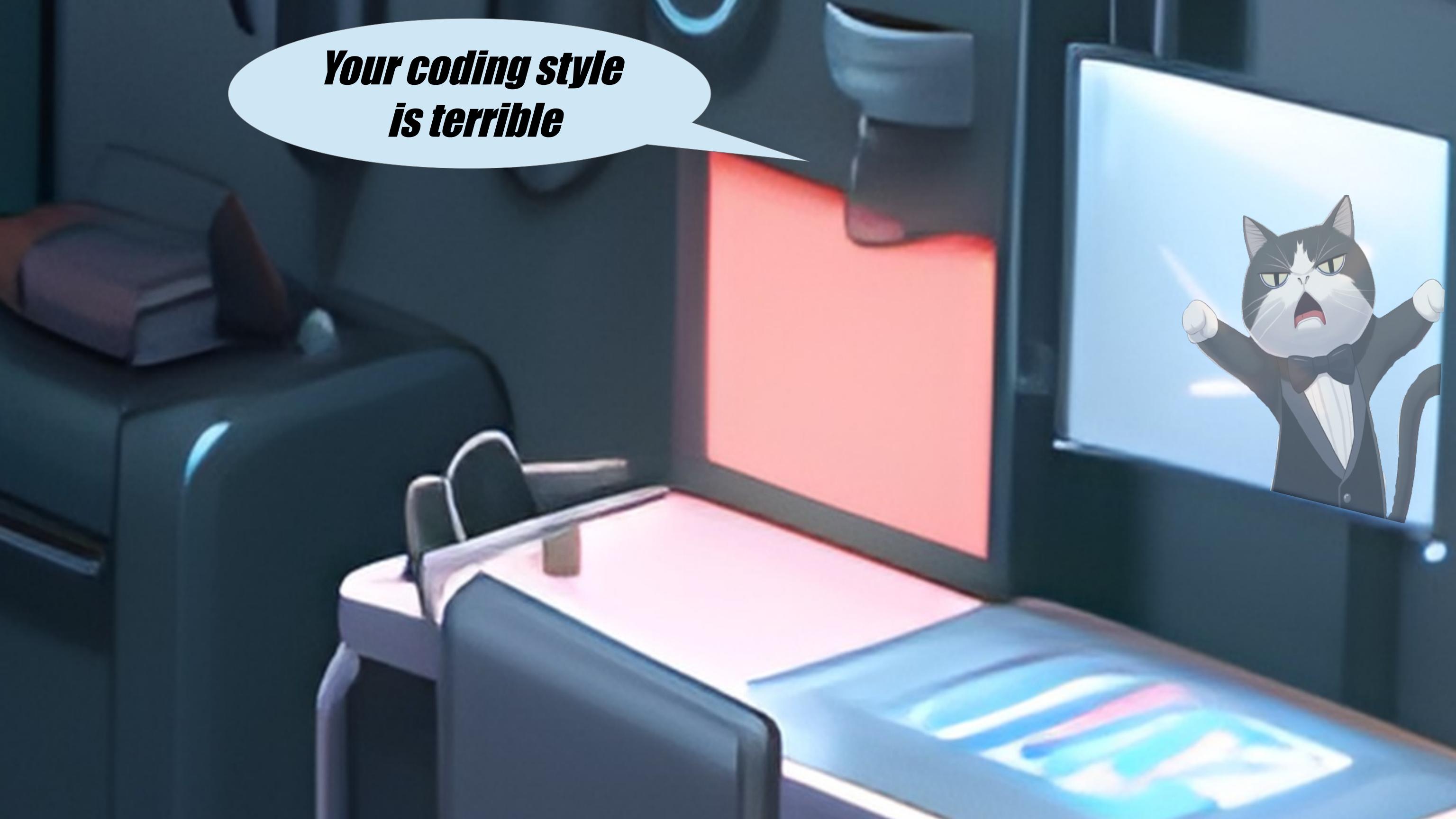




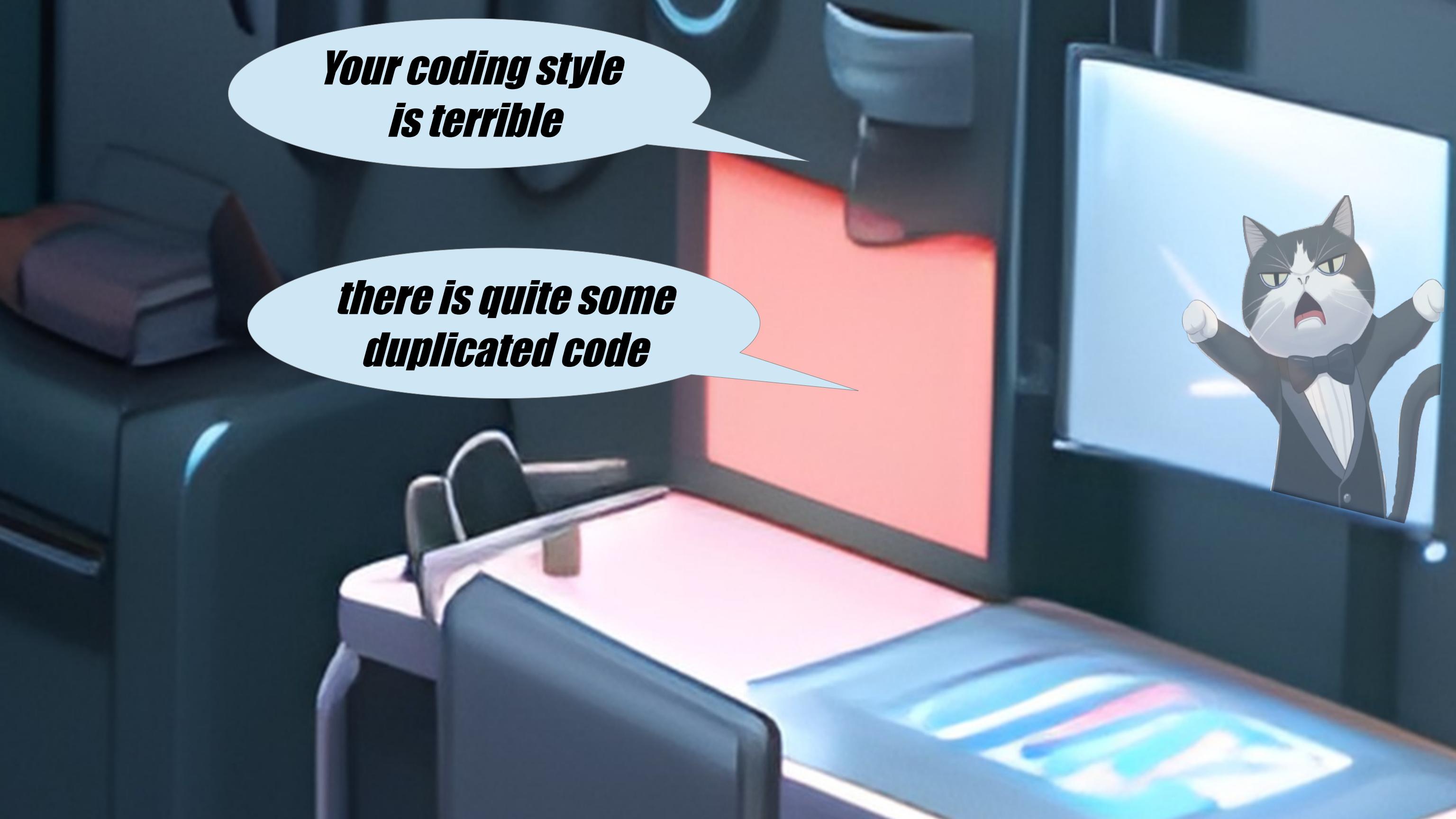
*To be clear:
it does compile, and
passes all our tests.*

So,
why is it
refused then?



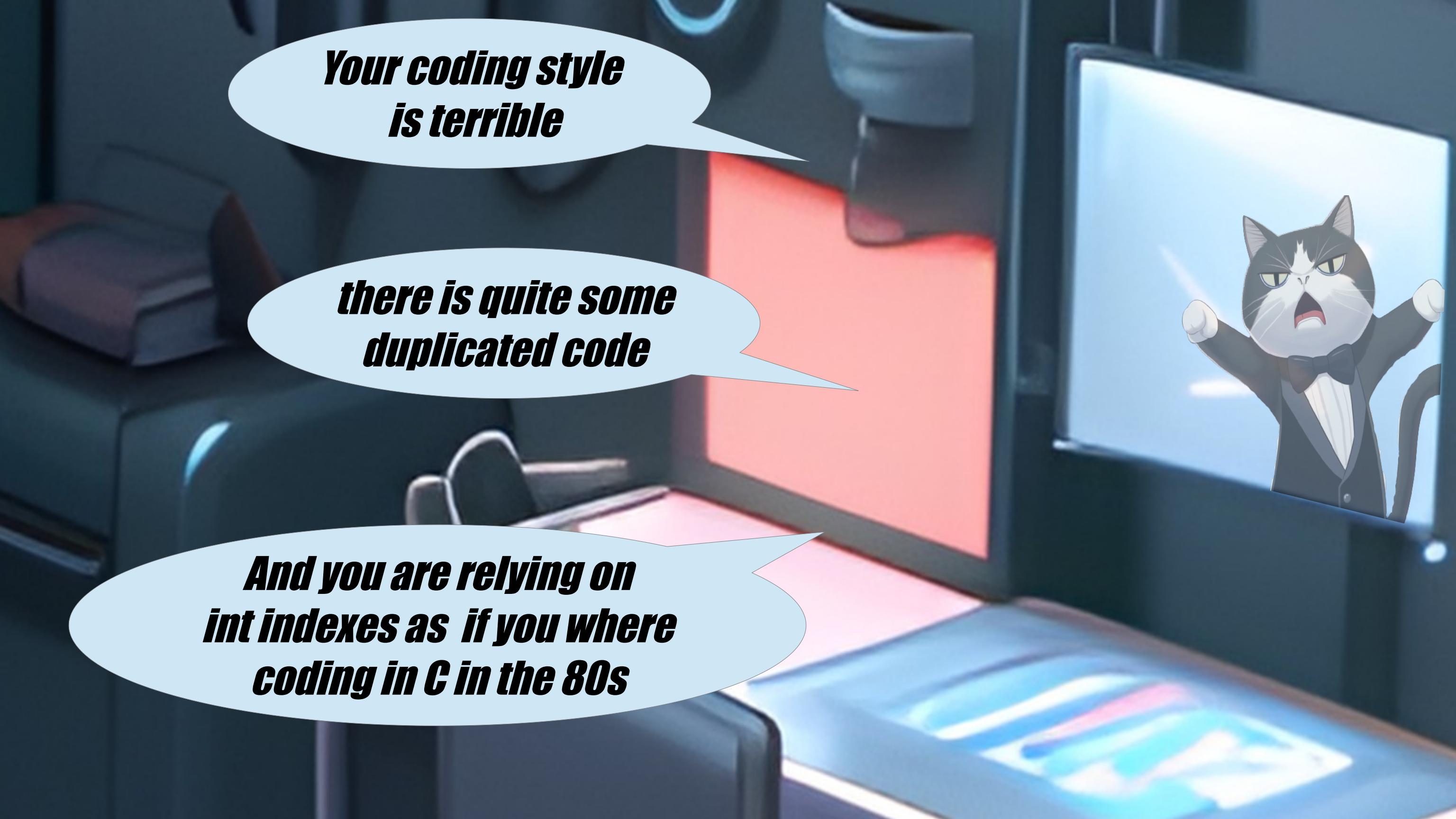


*Your coding style
is terrible*



*Your coding style
is terrible*

*there is quite some
duplicated code*



*Your coding style
is terrible*

*there is quite some
duplicated code*

*And you are relying on
int indexes as if you where
coding in C in the 80s*



But not all is lost!





But not all is lost!

***You can simply
resubmit when you
are ready***





But not all is lost!

***You can simply
resubmit when you
are ready***

***Every re submission caps
your mark 10% down***





***At your first re submission
any mark over 90% will be
capped to 90%.***





***At your first re submission
any mark over 90% will be
capped to 90%.***

***At your second
re submission
It will be 80%***



***Pass grade is 50%, so
between the two questions
you can submit 5 more times***



***Pass grade is 50%, so
between the two questions
you can submit 5 more times***

And still go to second year!







Just what I would
expect from UPU



Just what I would
expect from UPU

A first year student writing
perfectly working code...



Just what I would
expect from UPU

A first year student writing
perfectly working code...

at the first attempt, does not
even meet the minimum standards!



A close-up, front-facing shot of a young man with dark blue hair and glasses. He has a determined expression, with his mouth slightly open and a sweat drop on his forehead. He is wearing a blue zip-up hoodie. The background is a blurred cityscape at night with lights and streaks of motion.

It is time to activate my power!