AQA Computer Science A-Level

4.3.5 Sorting algorithms

Advanced Notes

Specification:

4.3.5.1 Bubble sort

Know and be able to trace and analyse the time complexity of the bubble sort algorithm. This is included as an example of a particularly inefficient sorting algorithm, time-wise. Time complexity is $O(n^2)$.

4.3.5.2 Merge sort

Be able to trace and analyse the time complexity of the merge sort algorithm. The 'merge' sort is an example of 'Divide and Conquer' approach to problem solving. Time complexity is $O(n\log n)$.

Sorting Algorithms

In the case of a sorting algorithm, the task is to put the elements of an array into a specific order. A sorted list can often be more useful than an unsorted one. A binary search can only be used on a sorted list, whereas a linear search can be used on either. Binary searches (O(logN)) are a lot faster than linear searches (O(N)), so sorted lists can reduce the time it takes to locate an item. There are multiple different sorting algorithms of varying complexity. The two investigated below are the bubble sort and the merge sort.

Algorithm

An algorithm is a set of instructions which completes a task and always terminates.

Synoptic Link

Binary and **linear searches** are examples of **searching algorithms**. They are designed to **locate** a **named item** in a **list**.

Binary and linear searches are covered in **Searching Algorithms** under **Fundamentals of Algorithms.**

Synoptic Link

O(logn) and **O(n)** are examples of **Big O notation**. Big O is a way of **classifying algorithms** based on **time and space complexity**. In this case, O(n) is more complex than O(logn).

Big O is covered in **Classification of Algorithms** under **Fundamentals of Algorithms**.

Sorting Algorithms Overview:

To be sorted into ascending order: 12 3 8

Bubble Sort

Make passes through the data and swap adjacent items. Stops passing through data when no swaps are performed. BigO is $O(n^2)$

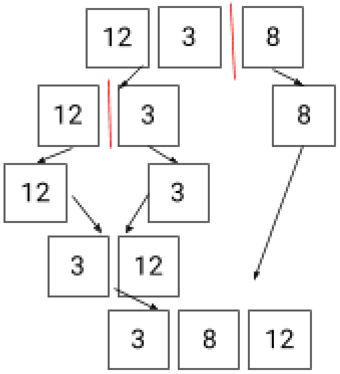
Pass 1: 12 3 8 → 3 12 8 → 3 8 12

Pass 2: 3 8 12 → 3 8 12 → 3 8 12

Sorted list: 3 8 12

Merge Sort

'Divide and Conquer' Method - split array up into individual sorted lists and then merge them together. BigO is O(nlogn)



Bubble Sort

The bubble sort algorithm uses the idea of swapping the position of adjacent items to order them. It has a time complexity of $O(n^2)$ so it is very inefficient.

Bubble Sort Example 1:

The following array needs to be sorted into ascending order.

Position	0	1	2	3	4
Data	Freddie	Brian	Roger	Adam	John

The first step is to compare the first two pieces of data.

Position	0	1	2	3	4
Data	Freddie	Brian	Roger	Adam	John

Freddie > Brian. Therefore Freddie and Brian should swap places.

Position	0	1	2	3	4
Data	Brian	Freddie	Roger	Adam	John

Position 1 is now checked against position 2.

Position	0	1	2	3	4
Data	Brian	Freddie	Roger	Adam	John

Freddie < Roger. Hence they are in the correct order, and do not need to be swapped.

Position	0	1	2	3	4
Data	Brian	Freddie	Roger	Adam	John

The data in position 2 of the array is checked against the data in position 3.

Position	0	1	2	3	4
Data	Brian	Freddie	Roger	Adam	John

Roger > Adam. They should swap positions.

Position	0	1	2	3	4
Data	Brian	Freddie	Adam	Roger	John

The third and fourth positions are checked.

Position	0	1	2	3	4
Data	Brian	Freddie	Adam	Roger	John

Roger > John, so they should swap positions in the array.

Position	0	1	2	3	4
Data	Brian	Freddie	Adam	John	Roger

There are no more positions to check. We can now say that we have made one pass through the data. We also know that the data in the last position, “Roger”, is in the correct position - we will highlight this in green to show that it does not need to be checked again; a good bubble sort algorithm will not have to check the last position. From looking, we can also see that “John” is in the correct position, however a computer will not know this until the second pass has been made.

The first two positions are checked again.

Position	0	1	2	3	4
Data	Brian	Freddie	Adam	John	Roger

Brian < Freddie. They are ordered, so should not be swapped.

Position	0	1	2	3	4
Data	Brian	Freddie	Adam	John	Roger

Position 1 is checked against position 2.

Position	0	1	2	3	4
Data	Brian	Freddie	Adam	John	Roger

Freddie > Adam. These pieces of data should swap positions.

Position	0	1	2	3	4
Data	Brian	Adam	Freddie	John	Roger

The second and third positions are checked.

Position	0	1	2	3	4
Data	Brian	Adam	Freddie	John	Roger

Freddie < John, so they do not have to be swapped.

Position	0	1	2	3	4
Data	Brian	Adam	Freddie	John	Roger

We can now say that we have made two passes through the data. Now John is definitely in the correct position, so it will be locked down, and there is no need to check it on the next pass.

The first two positions are checked again.

Position	0	1	2	3	4
Data	Brian	Adam	Freddie	John	Roger

Brian > Adam. They should swap positions.

Position	0	1	2	3	4
Data	Adam	Brian	Freddie	John	Roger

We can see that the data is correctly ordered, but a computer has no way of telling this. It can only determine that the list is in the correct order if it makes a pass through the data with no swaps.

The data in position 1 and 2 are checked.

Position	0	1	2	3	4
Data	Adam	Brian	Freddie	John	Roger

Brian < Freddie. They do not need to be swapped.

Position	0	1	2	3	4
Data	Adam	Brian	Freddie	John	Roger

We have now made a third pass through the data. The algorithm knows that the item in position 2, “Freddie” is in the correct place.

The data in position 0 and 1 are checked against each other.

Position	0	1	2	3	4
Data	Adam	Brian	Freddie	John	Roger

Adam < Brian. They do not need to be swapped. The data is now in the correct order.

Note

It is also possible to sort this list into ascending order and then reverse the end result - this could save time programming if you have already written the algorithm for an ascending sorting algorithm.

Bubble Sort Example 2:

The following data needs to be sorted into descending order.

Position	0	1	2	3	4	5	6
Data	Hannah	Jo	Jon	Bradley	Paul	Tina	Rachel

The first two pieces of data are checked.

Position	0	1	2	3	4	5	6
Data	Hannah	Jo	Jon	Bradley	Paul	Tina	Rachel

Hannah < Jo. Hence they need to be swapped.

Position	0	1	2	3	4	5	6
Data	Jo	Hannah	Jon	Bradley	Paul	Tina	Rachel

The data in position 1 and 2 are checked.

Position	0	1	2	3	4	5	6
Data	Jo	Hannah	Jon	Bradley	Paul	Tina	Rachel

Hannah < Jon so they swap positions.

Position	0	1	2	3	4	5	6
Data	Jo	Jon	Hannah	Bradley	Paul	Tina	Rachel

Positions 2 and 3 are checked.

Position	0	1	2	3	4	5	6
Data	Jo	Jon	Hannah	Bradley	Paul	Tina	Rachel

Hannah > Bradley, so they do not swap.

Position	0	1	2	3	4	5	6
Data	Jo	Jon	Hannah	Bradley	Paul	Tina	Rachel

The data in 3 and 4 are checked against one another.

Position	0	1	2	3	4	5	6
Data	Jo	Jon	Hannah	Bradley	Paul	Tina	Rachel

Bradley < Paul, so they swap positions.

Position	0	1	2	3	4	5	6
Data	Jo	Jon	Hannah	Paul	Bradley	Tina	Rachel

Next, positions 4 and 5 are examined.

Position	0	1	2	3	4	5	6
Data	Jo	Jon	Hannah	Paul	Bradley	Tina	Rachel

Bradley < Tina, so they should swap.

Position	0	1	2	3	4	5	6
Data	Jo	Jon	Hannah	Paul	Tina	Bradley	Rachel

Data in positions 5 and 6 are next to be checked.

Position	0	1	2	3	4	5	6
Data	Jo	Jon	Hannah	Paul	Tina	Bradley	Rachel

Bradley < Rachel, so they swap positions.

Position	0	1	2	3	4	5	6
Data	Jo	Jon	Hannah	Paul	Tina	Rachel	Bradley

We have made one pass through the data, so Bradley is locked in place.

Position	0	1	2	3	4	5	6
Data	Jo	Jon	Hannah	Paul	Tina	Rachel	Bradley

The second pass begins by checking positions 0 and 1.

Position	0	1	2	3	4	5	6
Data	Jo	Jon	Hannah	Paul	Tina	Rachel	Bradley

Jo < Jon, so they are swapped.

Position	0	1	2	3	4	5	6
Data	Jon	Jo	Hannah	Paul	Tina	Rachel	Bradley

Positions 1 and 2 are now observed.

Position	0	1	2	3	4	5	6
Data	Jon	Jo	Hannah	Paul	Tina	Rachel	Bradley

Jo > Hannah, so they do not swap positions.

Position	0	1	2	3	4	5	6
Data	Jon	Jo	Hannah	Paul	Tina	Rachel	Bradley

Positions 2 and 3 are checked.

Position	0	1	2	3	4	5	6
Data	Jon	Jo	Hannah	Paul	Tina	Rachel	Bradley

Hannah < Paul. They swap positions.

Position	0	1	2	3	4	5	6
Data	Jon	Jo	Paul	Hannah	Tina	Rachel	Bradley

The data in positions 3 and 4 are observed.

Position	0	1	2	3	4	5	6
Data	Jon	Jo	Paul	Hannah	Tina	Rachel	Bradley

Hannah < Tina, so they swap.

Position	0	1	2	3	4	5	6
Data	Jon	Jo	Paul	Tina	Hannah	Rachel	Bradley

4 and 5 are examined.

Position	0	1	2	3	4	5	6
Data	Jon	Jo	Paul	Tina	Hannah	Rachel	Bradley

Hannah < Rachel. They have to swap.

Position	0	1	2	3	4	5	6
Data	Jon	Jo	Paul	Tina	Rachel	Hannah	Bradley

We have made a second pass through the data, so Hannah is locked down.

Position	0	1	2	3	4	5	6
Data	Jon	Jo	Paul	Tina	Rachel	Hannah	Bradley

The third pass starts by checking the data in the first two positions.

Position	0	1	2	3	4	5	6
Data	Jon	Jo	Paul	Tina	Rachel	Hannah	Bradley

Jon > Jo. They are in order, so do not need to swap.

Position	0	1	2	3	4	5	6
Data	Jon	Jo	Paul	Tina	Rachel	Hannah	Bradley

The data in positions 1 and 2 need to be checked.

Position	0	1	2	3	4	5	6
Data	Jon	Jo	Paul	Tina	Rachel	Hannah	Bradley

Jo < Paul, so they swap positions.

Position	0	1	2	3	4	5	6
Data	Jon	Paul	Jo	Tina	Rachel	Hannah	Bradley

Positions 2 and 3 are examined.

Position	0	1	2	3	4	5	6
Data	Jon	Paul	Jo	Tina	Rachel	Hannah	Bradley

Jo < Tina. They trade positions.

Position	0	1	2	3	4	5	6
Data	Jon	Paul	Tina	Jo	Rachel	Hannah	Bradley

The items in positions 3 and 4 are tested.

Position	0	1	2	3	4	5	6
Data	Jon	Paul	Tina	Jo	Rachel	Hannah	Bradley

Jo < Rachel, so they swap places.

Position	0	1	2	3	4	5	6
Data	Jon	Paul	Tina	Rachel	Jo	Hannah	Bradley

The third pass has been completed; Jo is locked in place.

Position	0	1	2	3	4	5	6
Data	Jon	Paul	Tina	Rachel	Jo	Hannah	Bradley

The fourth pass starts with the first two positions.

Position	0	1	2	3	4	5	6
Data	Jon	Paul	Tina	Rachel	Jo	Hannah	Bradley

Jon < Paul. They swap.

Position	0	1	2	3	4	5	6
Data	Paul	Jon	Tina	Rachel	Jo	Hannah	Bradley

The data in positions 1 and 2 are inspected.

Position	0	1	2	3	4	5	6
Data	Paul	Jon	Tina	Rachel	Jo	Hannah	Bradley

Jon < Tina. They trade positions.

Position	0	1	2	3	4	5	6
Data	Paul	Tina	Jon	Rachel	Jo	Hannah	Bradley

Positions 2 and 3 are examined next.

Position	0	1	2	3	4	5	6
Data	Paul	Tina	Jon	Rachel	Jo	Hannah	Bradley

Jon < Rachel. They swap places.

Position	0	1	2	3	4	5	6
Data	Paul	Tina	Rachel	Jon	Jo	Hannah	Bradley

A fourth pass has been made through the data so Jon can be locked in place.

Position	0	1	2	3	4	5	6
Data	Paul	Tina	Rachel	Jon	Jo	Hannah	Bradley

The fifth pass begins by checking positions 0 and 1.

Position	0	1	2	3	4	5	6
Data	Paul	Tina	Rachel	Jon	Jo	Hannah	Bradley

Paul < Tina, so they swap positions.

Position	0	1	2	3	4	5	6
Data	Tina	Paul	Rachel	Jon	Jo	Hannah	Bradley

Next, items 1 and 2 are inspected.

Position	0	1	2	3	4	5	6
Data	Tina	Paul	Rachel	Jon	Jo	Hannah	Bradley

Paul < Rachel. Hence, they trade places.

Position	0	1	2	3	4	5	6
Data	Tina	Rachel	Paul	Jon	Jo	Hannah	Bradley

The fifth pass has been made through the data, so Paul is locked in place.

Position	0	1	2	3	4	5	6
Data	Tina	Rachel	Paul	Jon	Jo	Hannah	Bradley

We can see that the data is sorted, but the computer must make a pass through the data with no swaps to determine this.

The sixth pass checks positions 0 and 1.

Position	0	1	2	3	4	5	6
Data	Tina	Rachel	Paul	Jon	Jo	Hannah	Bradley

Tina > Rachel, so they do not swap. The list is sorted.

Bubble Sort Example 3

A question may ask you how the data looks after a stated number of passes, or how many passes are required to sort the array. The below example only shows how the list would look after each pass.

Here is our unsorted list:

Position	0	1	2	3	4	5	6
Data	Mon	Tues	Weds	Thurs	Fri	Sat	Sun

First Pass:

Position	0	1	2	3	4	5	6
Data	Mon	Tues	Thurs	Fri	Sat	Sun	Weds

Second Pass:

Position	0	1	2	3	4	5	6
Data	Mon	Thurs	Fri	Sat	Sun	Tues	Weds

Third Pass:

Position	0	1	2	3	4	5	6
Data	Mon	Fri	Sat	Sun	Thurs	Tues	Weds

Fourth Pass:

Position	0	1	2	3	4	5	6
Data	Fri	Mon	Sat	Sun	Thurs	Tues	Weds

Although the data is sorted, the computer needs to make a pass through the data where there are no swaps.

Fifth Pass:

Position	0	1	2	3	4	5	6
Data	Fri	Mon	Sat	Sun	Thurs	Tues	Weds

This list took 5 passes through the data to be sorted.

Pseudocode (Ascending)

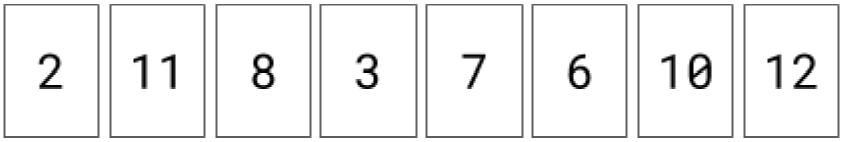
```
Integer Max
String Temp
Boolean Swapped
Integer Passes

Max ← List.Count - 1
Swapped ← TRUE
Passes ← 0

Do until Swapped == FALSE or Max == 0
     Swapped ← FALSE
     Max ← Max - 1
     Passes ← Passes + 1
     For a = 0 to max
          If List(a) > List(a + 1)
               Temp ← List(a)
               List(a) ← List(a + 1)
               List(a + 1) ← Temp
               Swapped ← TRUE
          End If
     Next
Loop

OUTPUT Passes
OUTPUT List
```

Merge Sort

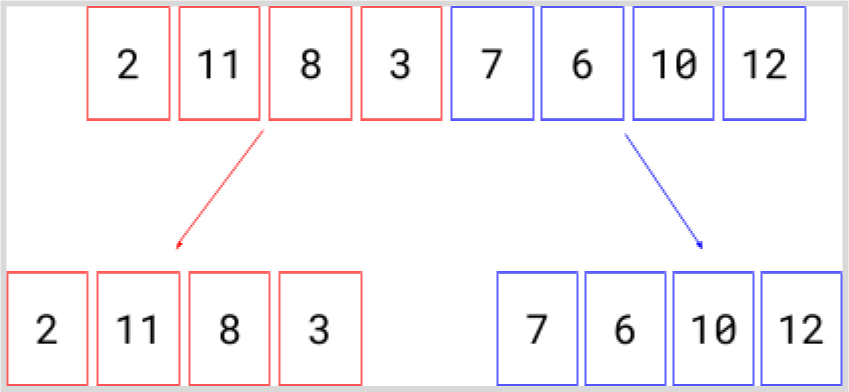
A merge sort orders arrays by splitting them into smaller lists, and then reforming them - the ‘divide and conquer’ method. It is quicker than a bubble sort; it has a time complexity of O(nlogn).

Merge Sort Example 1:

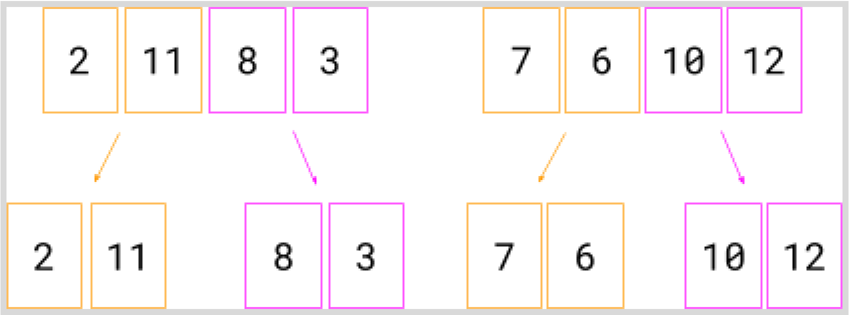
Here is an unsorted list.



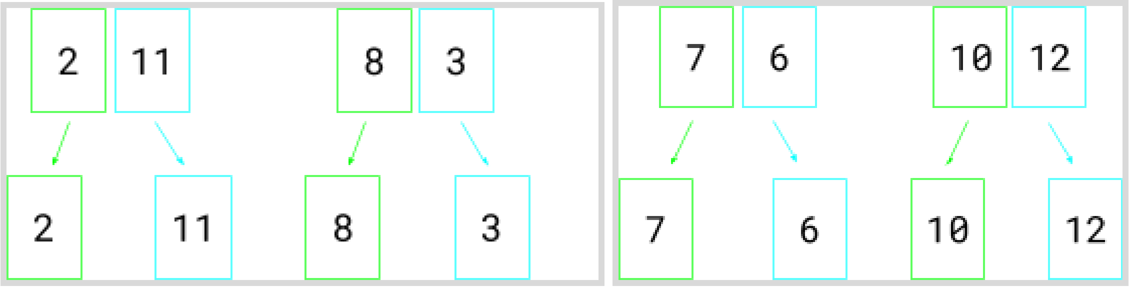
The first stage in a merge sort is to split the list into two smaller lists.



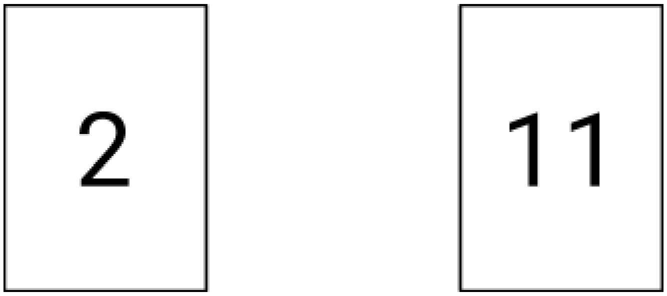
These lists are still unsorted, so they need to be split further.



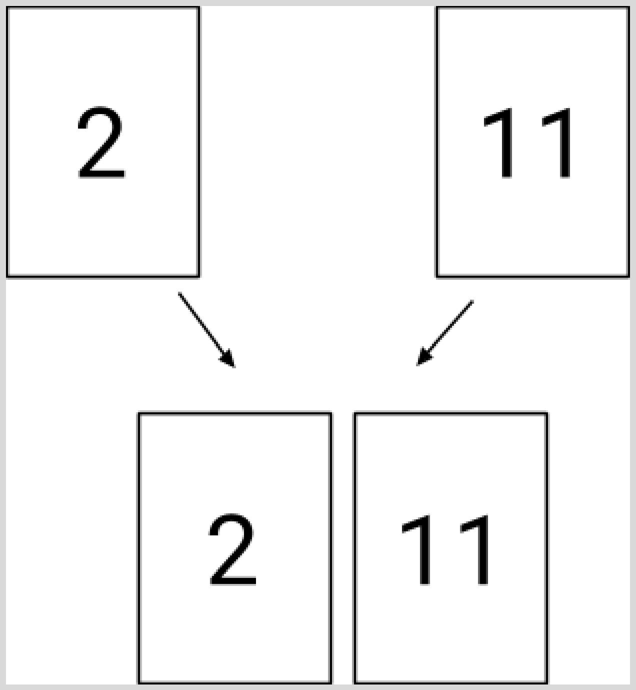
These lists are unsorted, with two elements; they need to be split further.



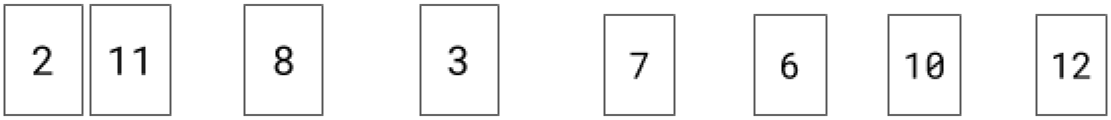
Now there are eight lists each with one element. Since there is only one element in each, they are all ordered lists. Now they can be put back together by comparison. We start with the first two lists.



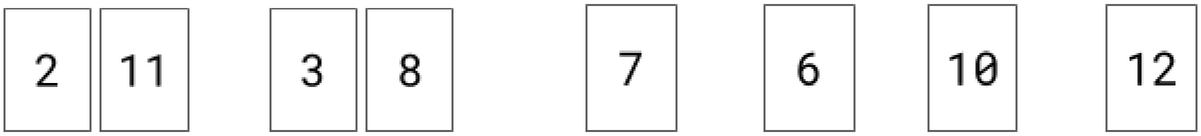
2 < 11, so the ordered list is 2, 11.



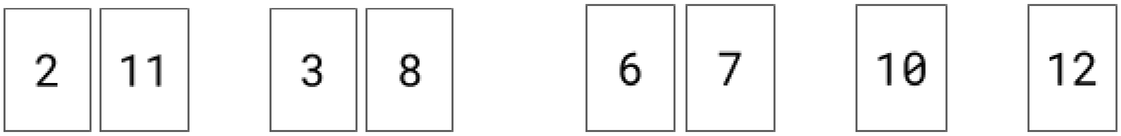
Our collection of lists looks like this:



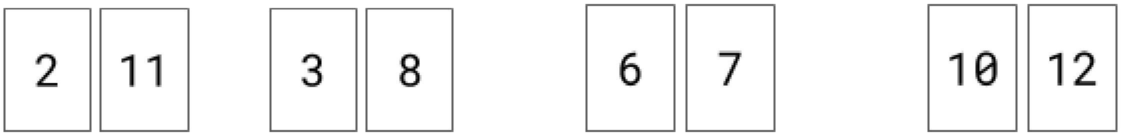
The next pair is 8 and 3. 8 > 3 so they are paired up like this.



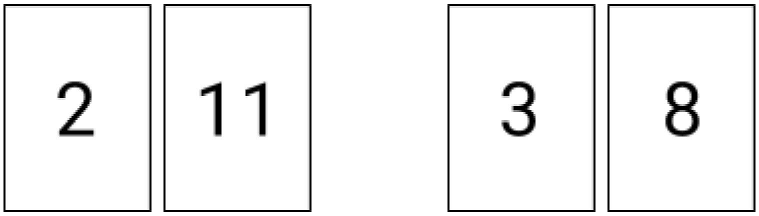
The next pair is 7 and 6. 7 > 6, so they are combined with 6 as the first element in the list.



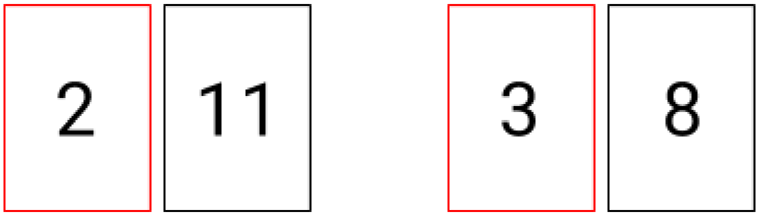
The last pair is 10 and 12. 10 < 12, so they are paired up as follows.



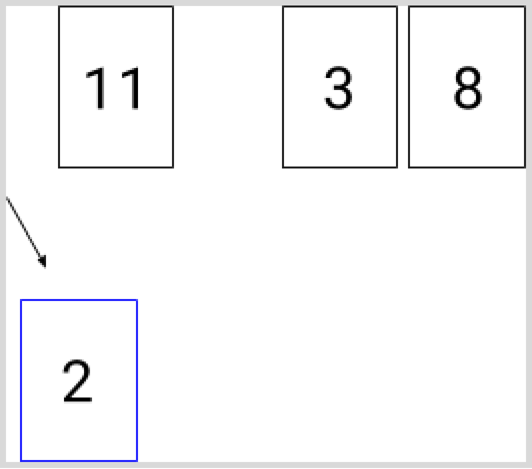
We now have four sorted lists, each containing two elements. The next stage is to once again consider adjacent lists. We start with the first two lists, (2,11) and (3,8).



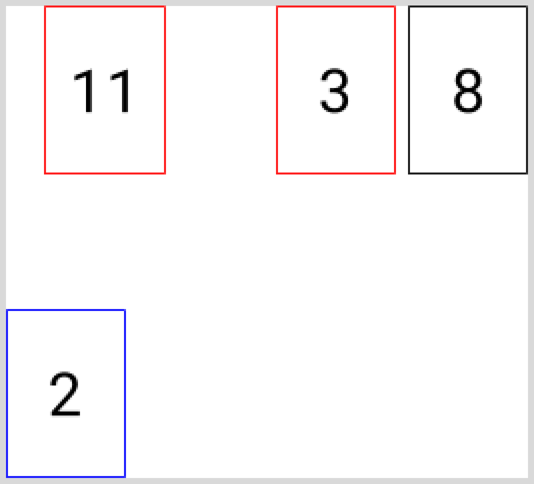
The smallest element in the first list is 2, and the smallest element in the second list is 3.



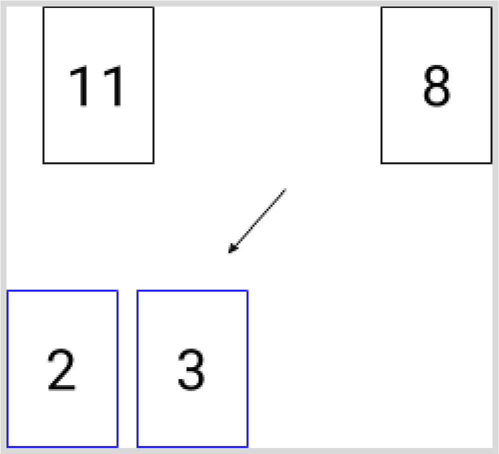
2 < 3, so it is added to the new list first.



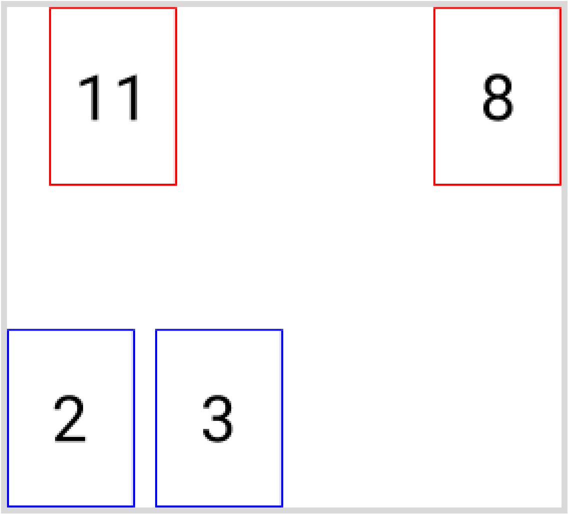
Now the smallest element in the first list is 11.



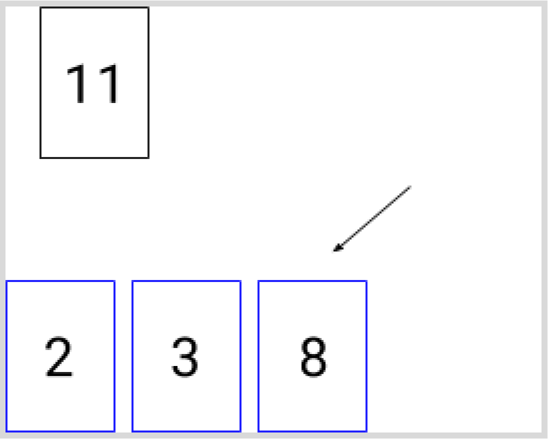
3 < 11, so it is added to the list next.



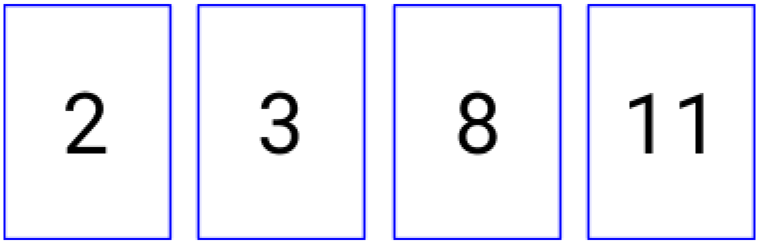
Now the smallest element in the second list is 8.



8 < 11, so it is added to the list first.



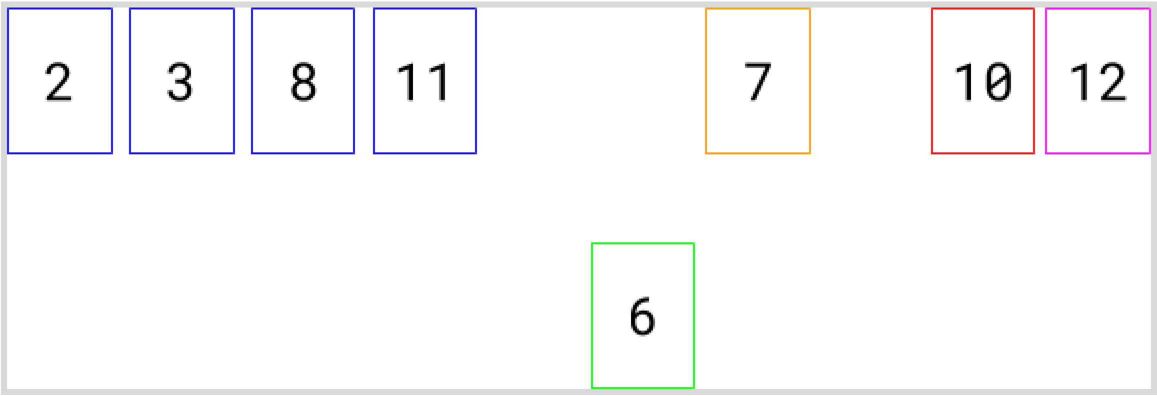
The final element is 11, so it goes on the end of the sorted list.



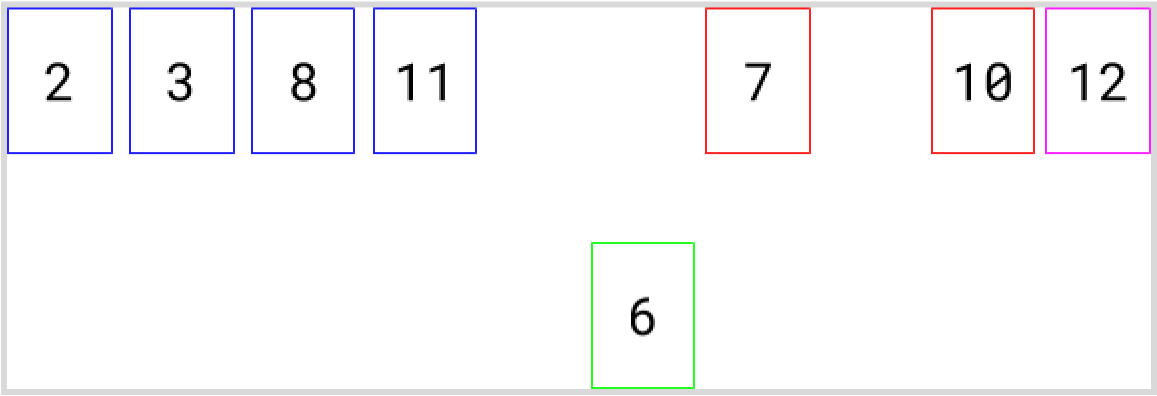
All of our lists together look like this.

We now need to put together the other two lists. The smallest element in the orange list is 6, and the smallest element in the pink list is 10.

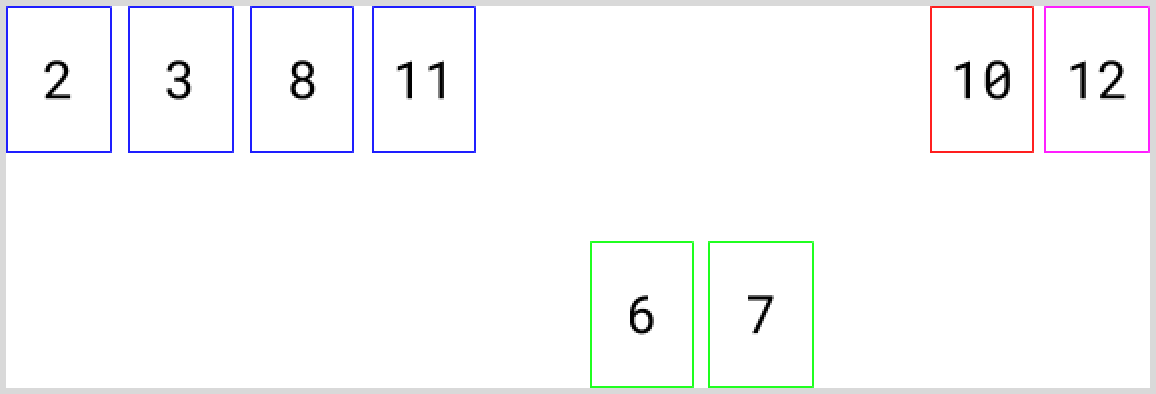
6 < 10 so it is added to the new sorted list first.



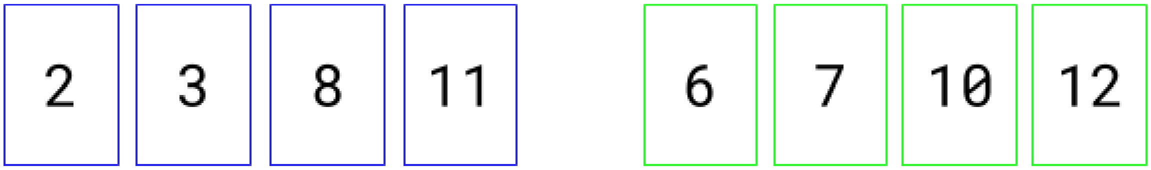
7 is now the smallest element in the orange list.



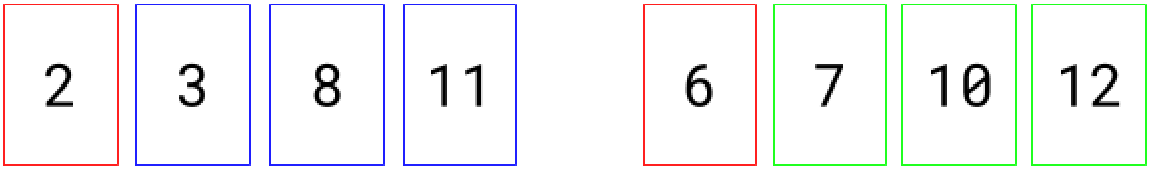
7 < 10 so 7 is added first.



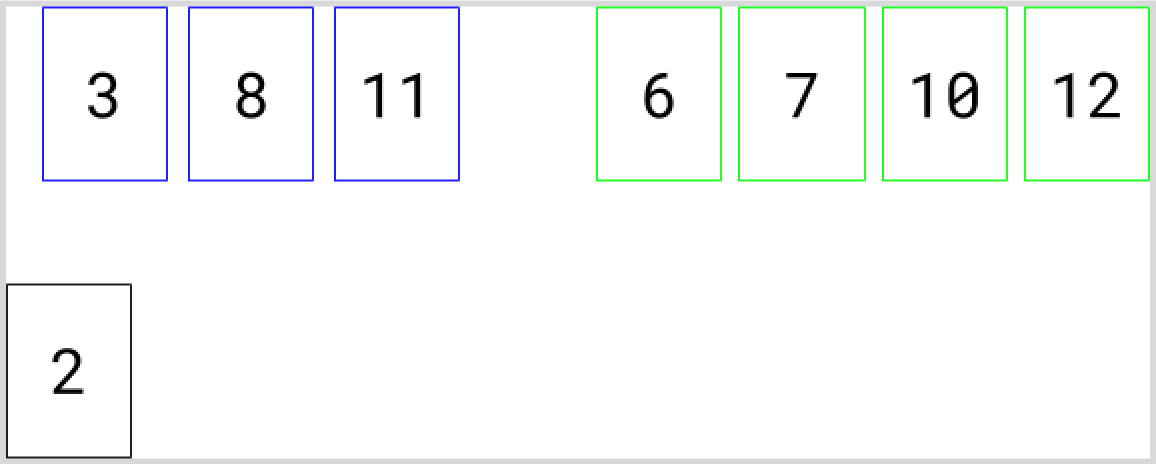
The pink list is already sorted, so it can be added onto the end of the 6 and 7.



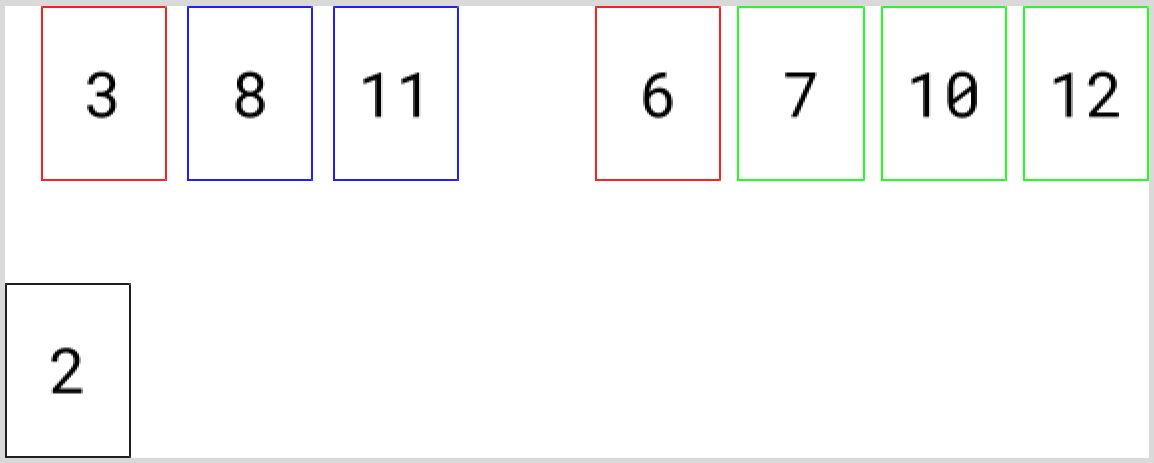
We have two sorted lists each of four elements. The process of combining lists is repeated. The smallest element in the blue list is 2, and the smallest element in the green list is 6.



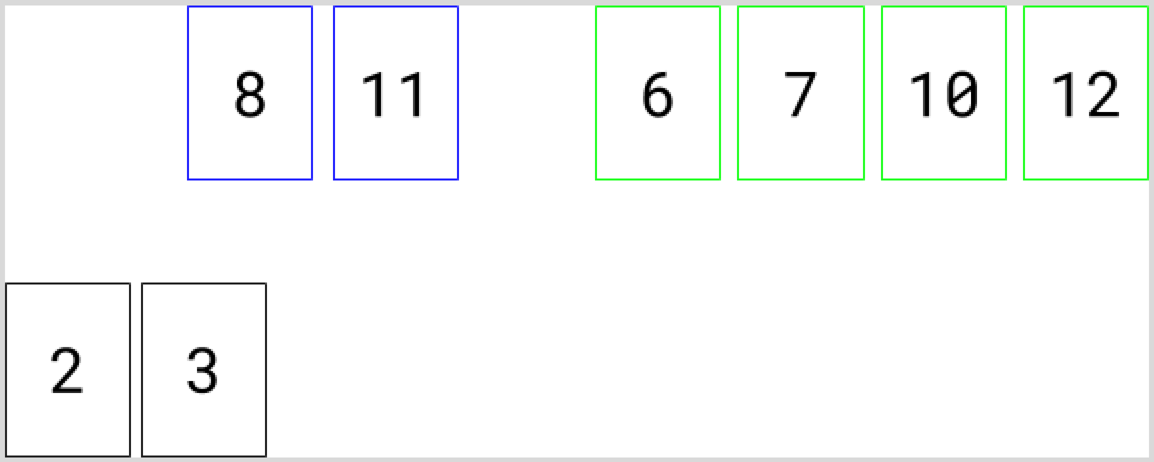
2 < 6. 2 is first on the sorted list.



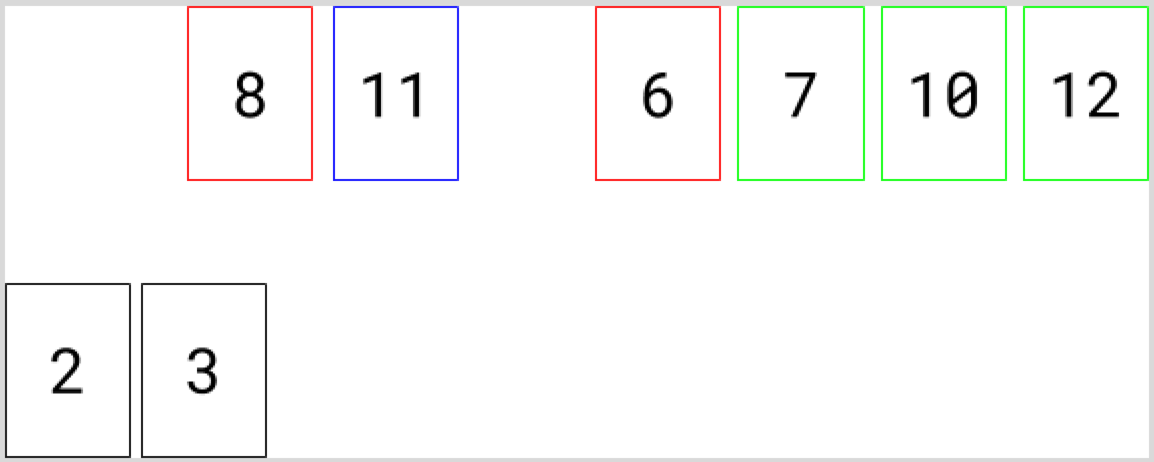
The smallest element in the blue list is 3.



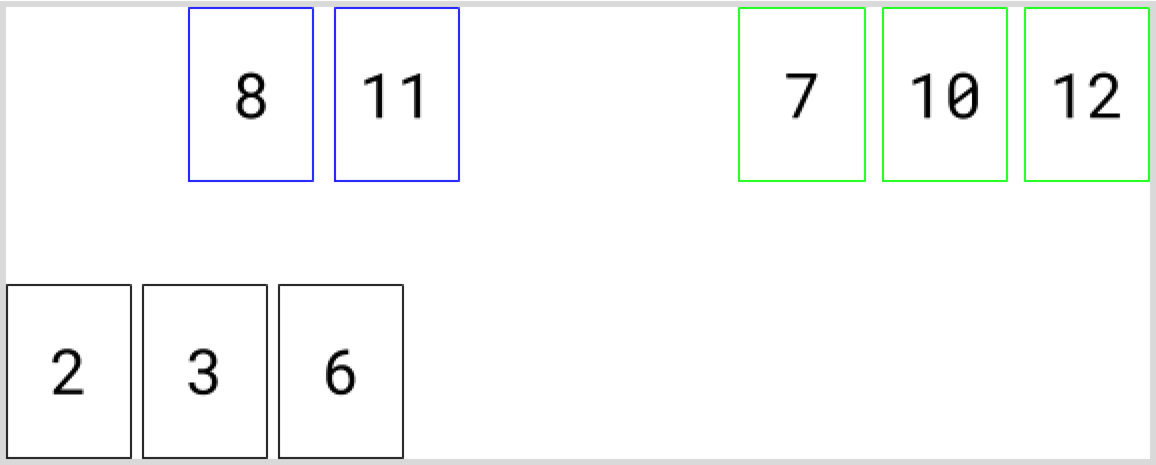
3 < 6, so 3 the next element on the sorted list.



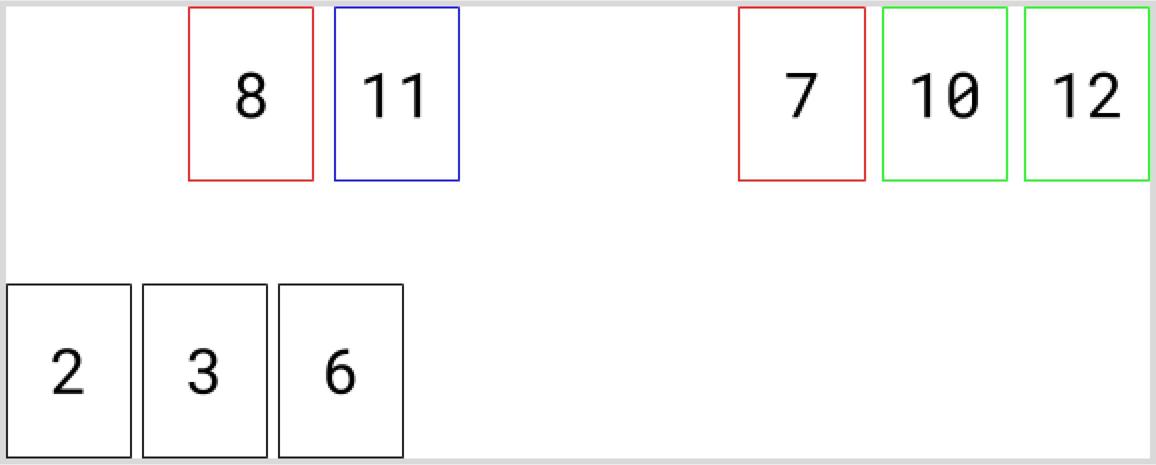
The smallest element in the blue list is 8.



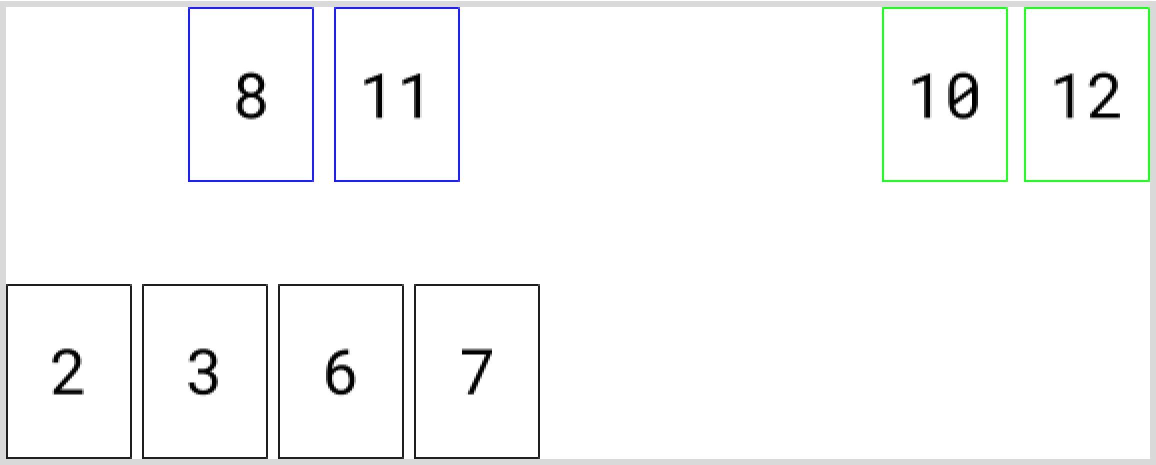
6 < 8, so 6 is added next.



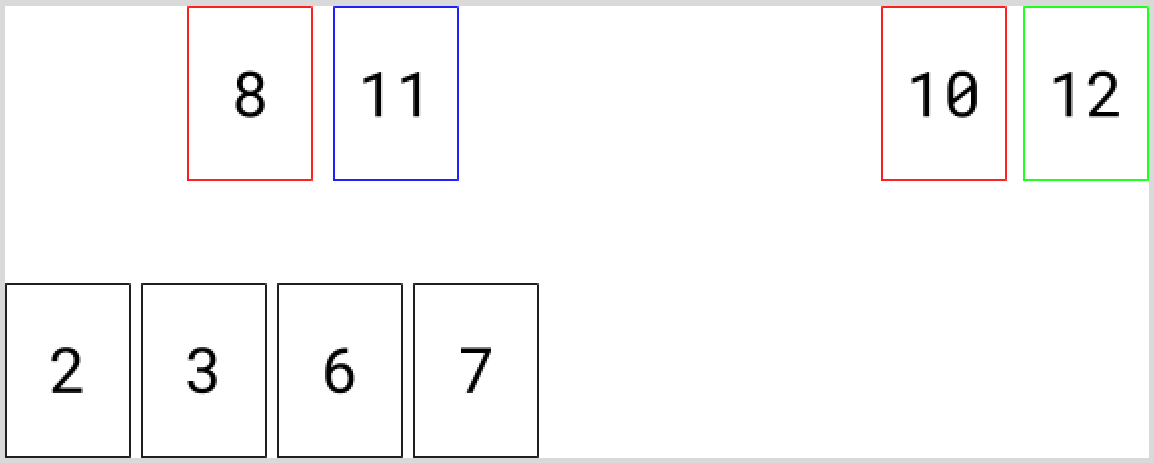
The smallest item in the green list is 7.



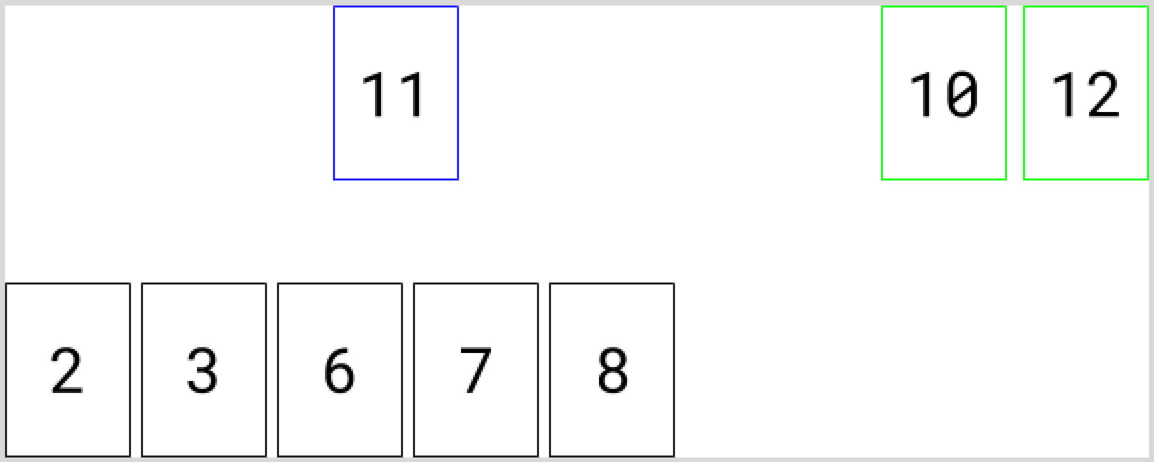
8 > 7. Hence, 7 is next on the list.



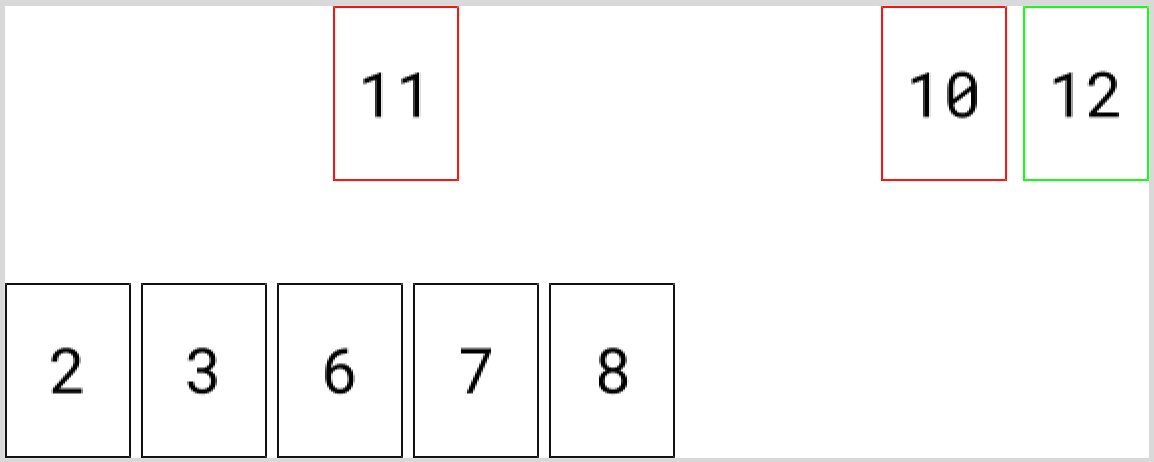
The smallest element in the green list is 10.



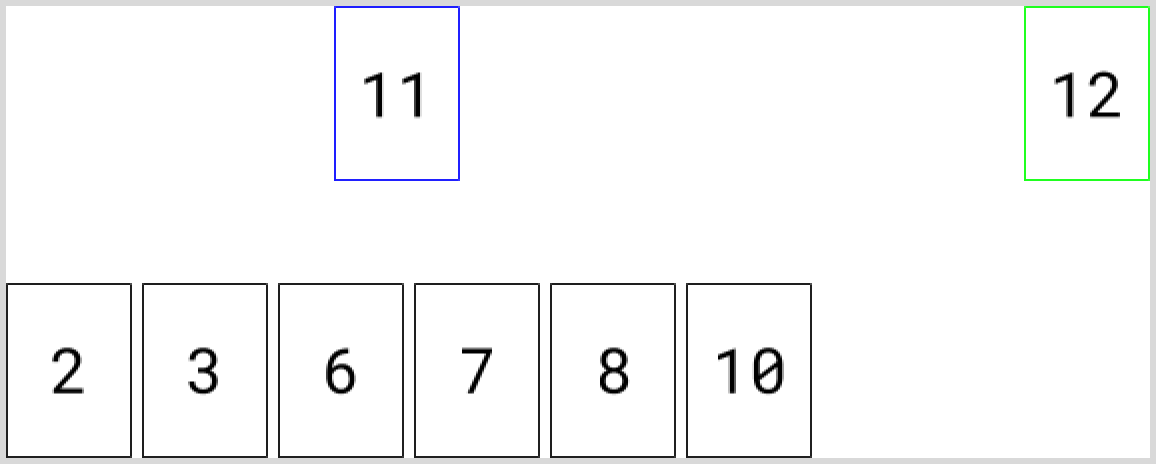
8 < 10. 8 is added next.



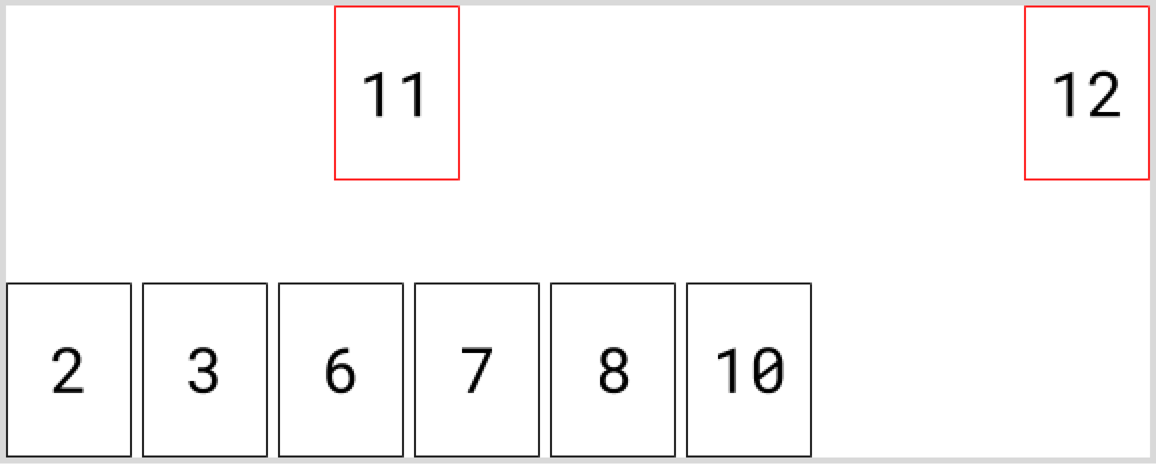
The smallest item in the blue list is 11.



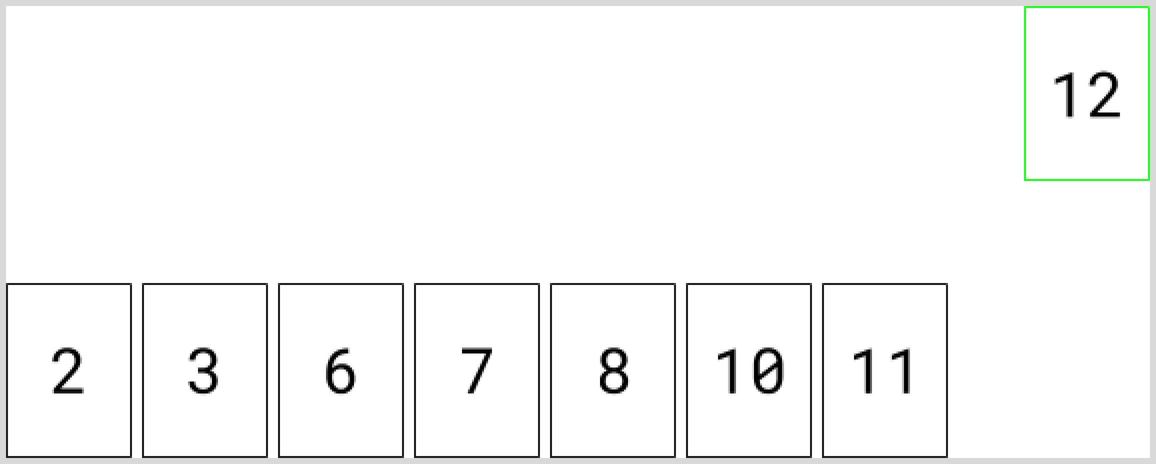
11 > 10. Thus, 10 is added next.



12 is the smallest element in the green list.



11 < 12. 11 is added to the list.



The blue list is empty, so the contents of the ordered green list can be added to the end of the black list. This is our final ordered list.

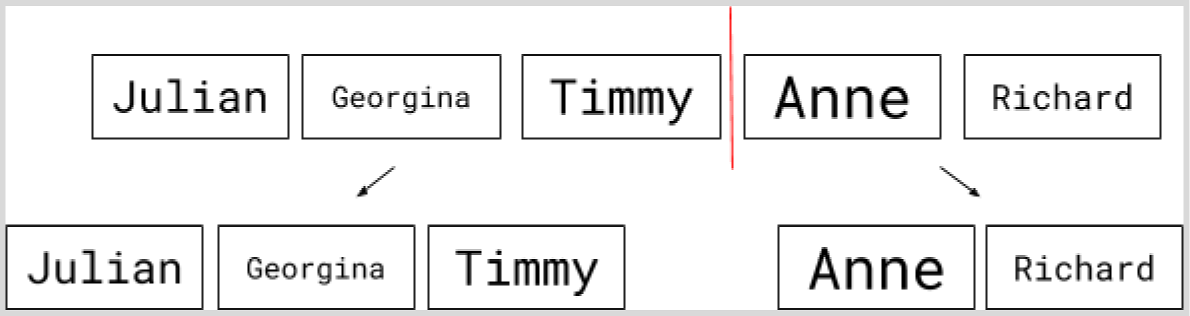
2	3	6	7	8	10	11	12

Merge Sort Example 2:

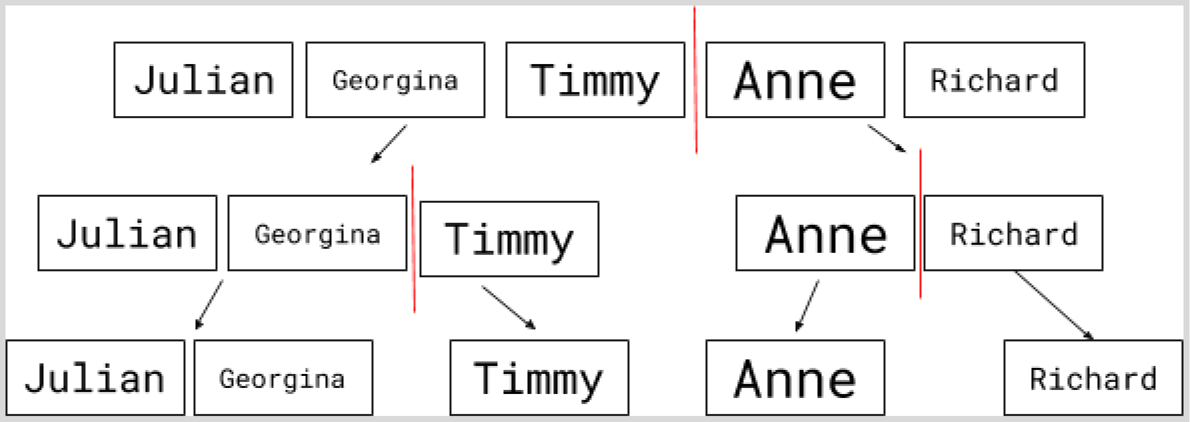
Here is an unsorted array:

Julian	Georgina	Timmy	Anne	Richard

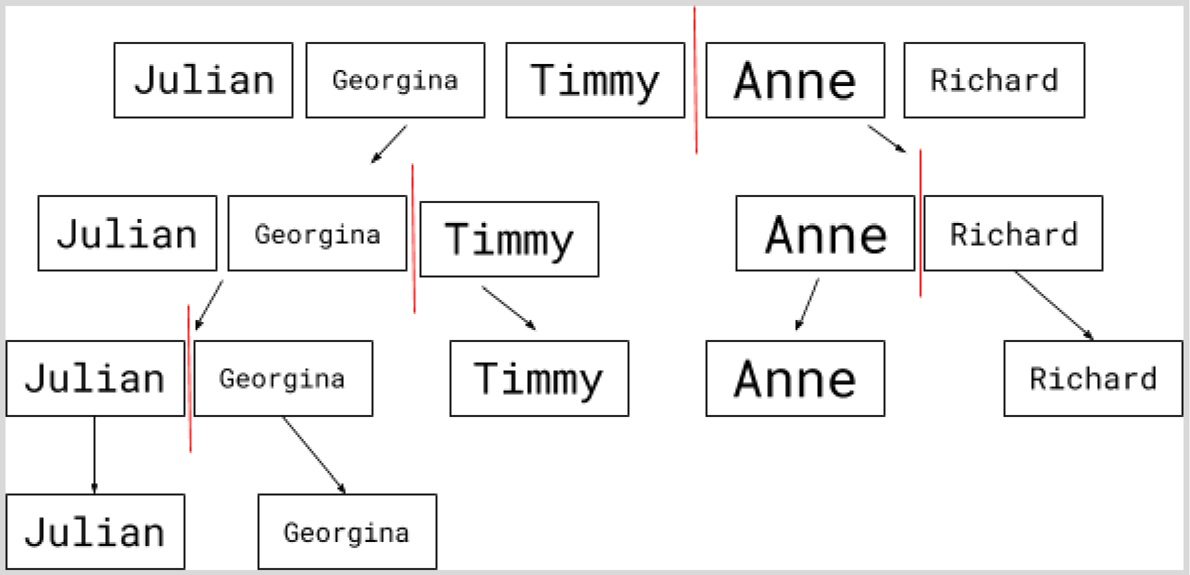
The first step is to split the array into two.



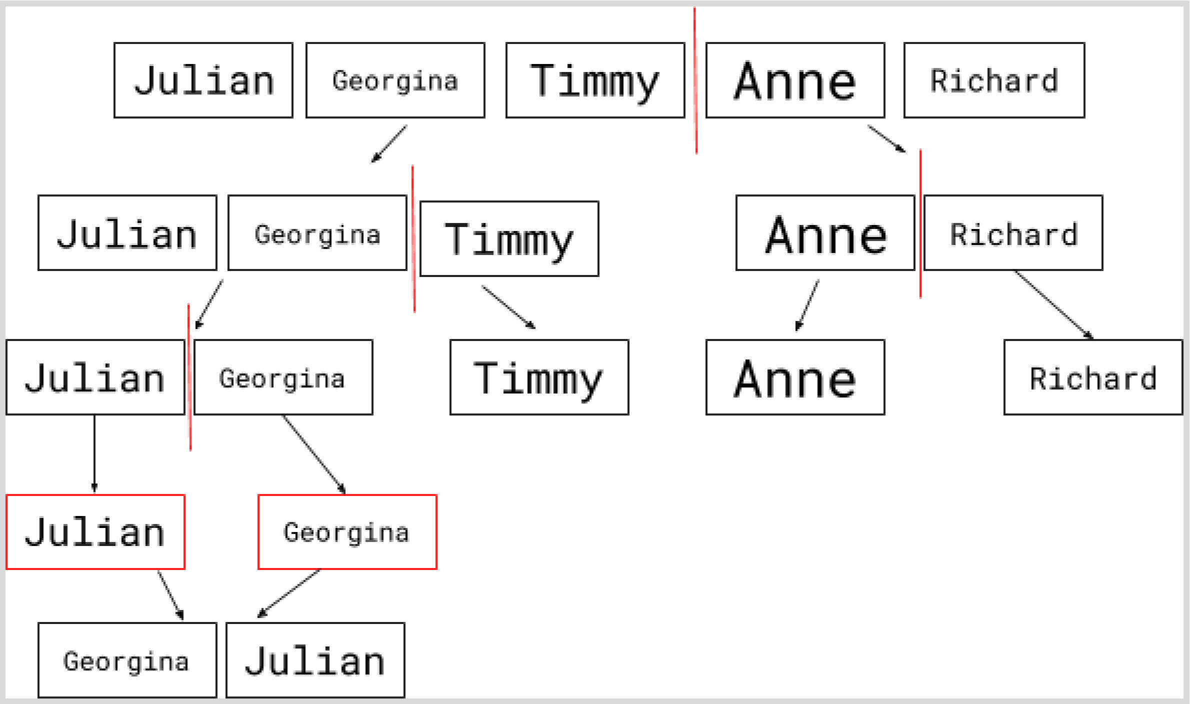
Split them again.



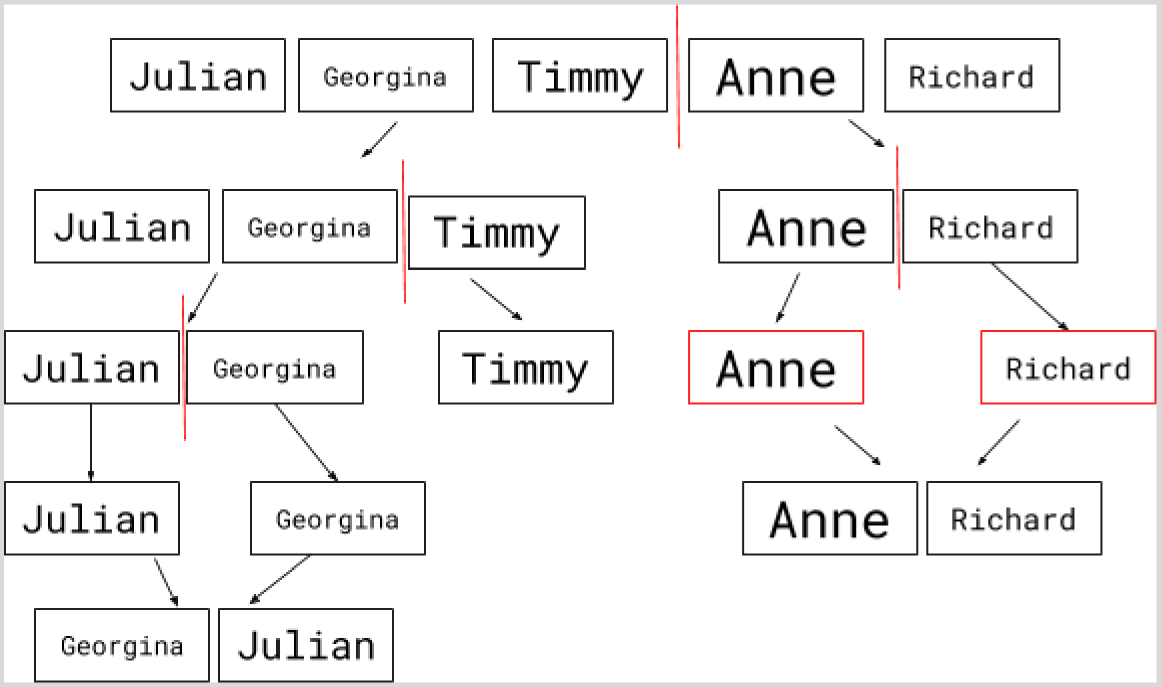
Julian and Georgina are still a pair, so they need to be split again.



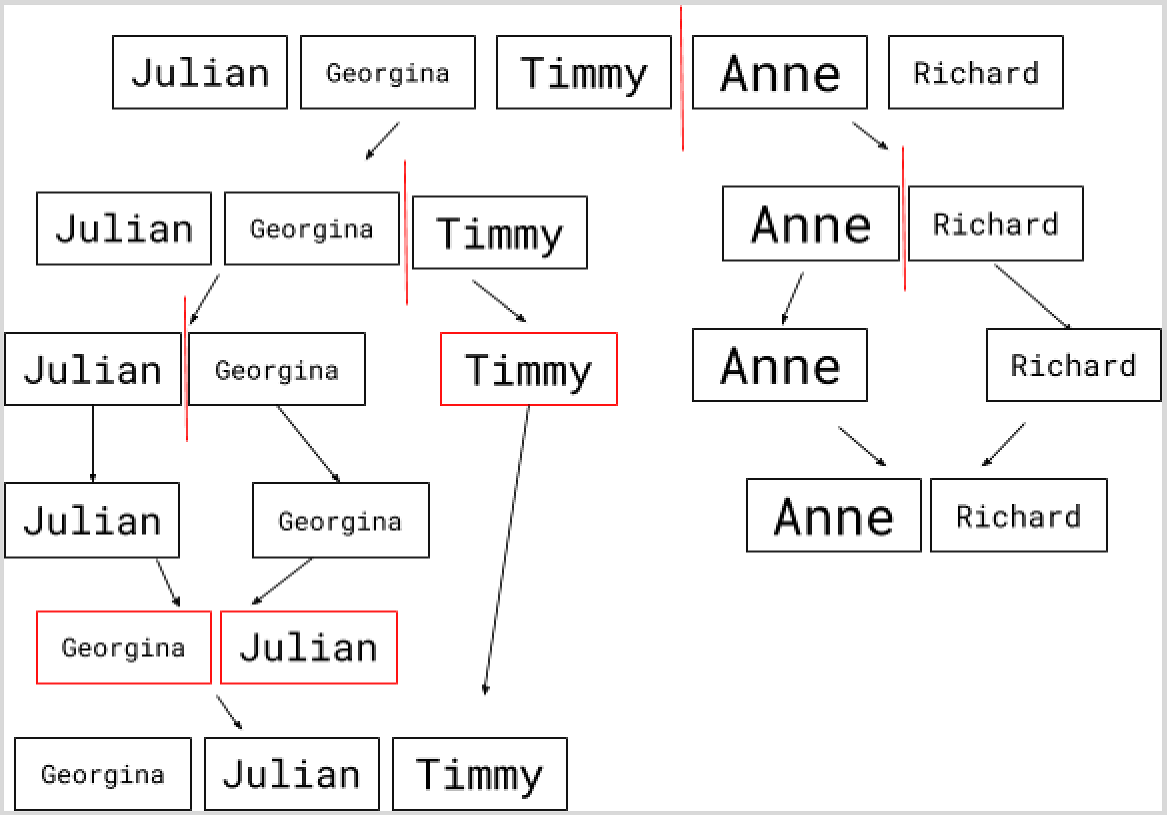
Reform the lists by merging ordered single items. Julian & Georgina:



Reform the lists by creating ordered pairs. Anne & Richard:



Reform lists with further comparisons.



Finally, merge both sorted lists together.