Sorting and Searching: Merge Sort

CMPS 1500

Prof. Ramgopal Mettu

Is Selection Sort Practical?

What is the running time of selection sort for lists that have thousands of items?

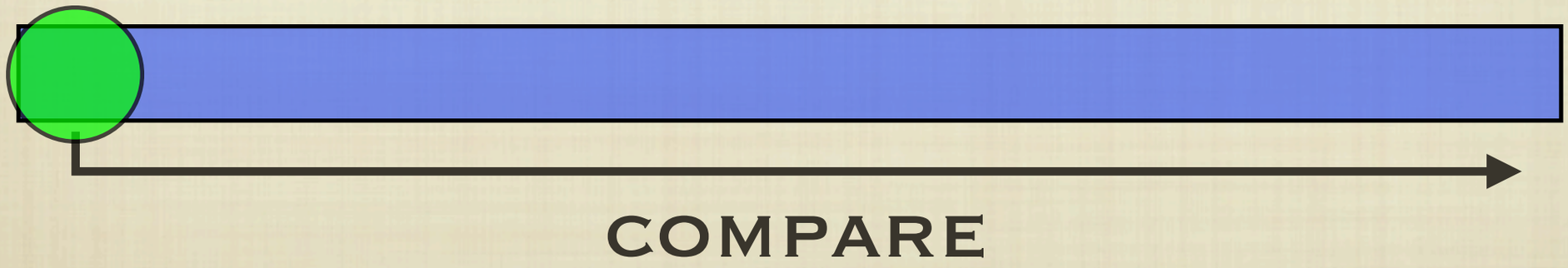
The theoretical performance is not particularly promising, neither is the practical performance:

Size	Selection Sort
10	0.00005
100	0.001681
1000	0.143361
10000	14.192897
100000	2638.207074

What is done in practice? How good is the Python library `sort()` function?

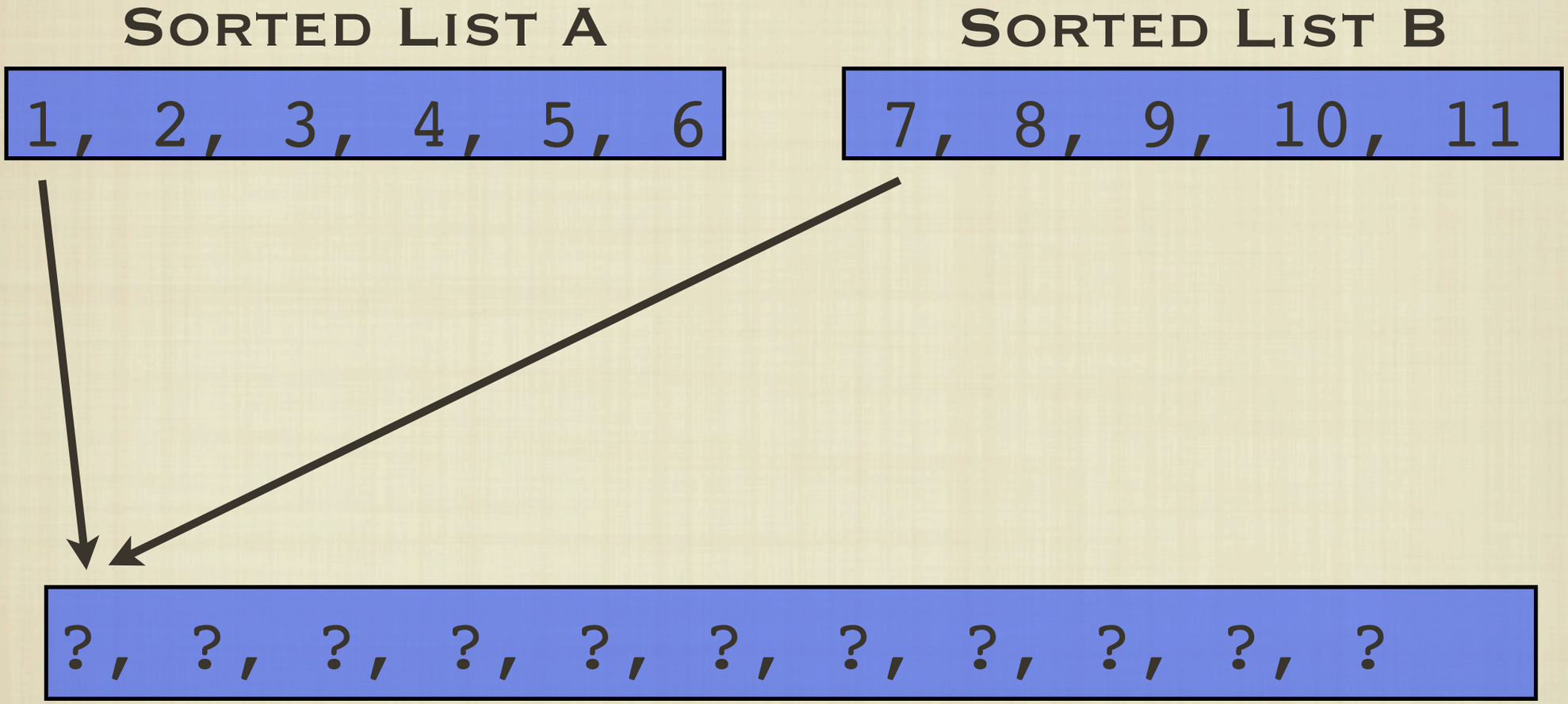
Revisiting Selection Sort

- What is the minimum amount of time required to sort a list?
- Selection sort takes linear time to place just a single element. Is this really necessary?



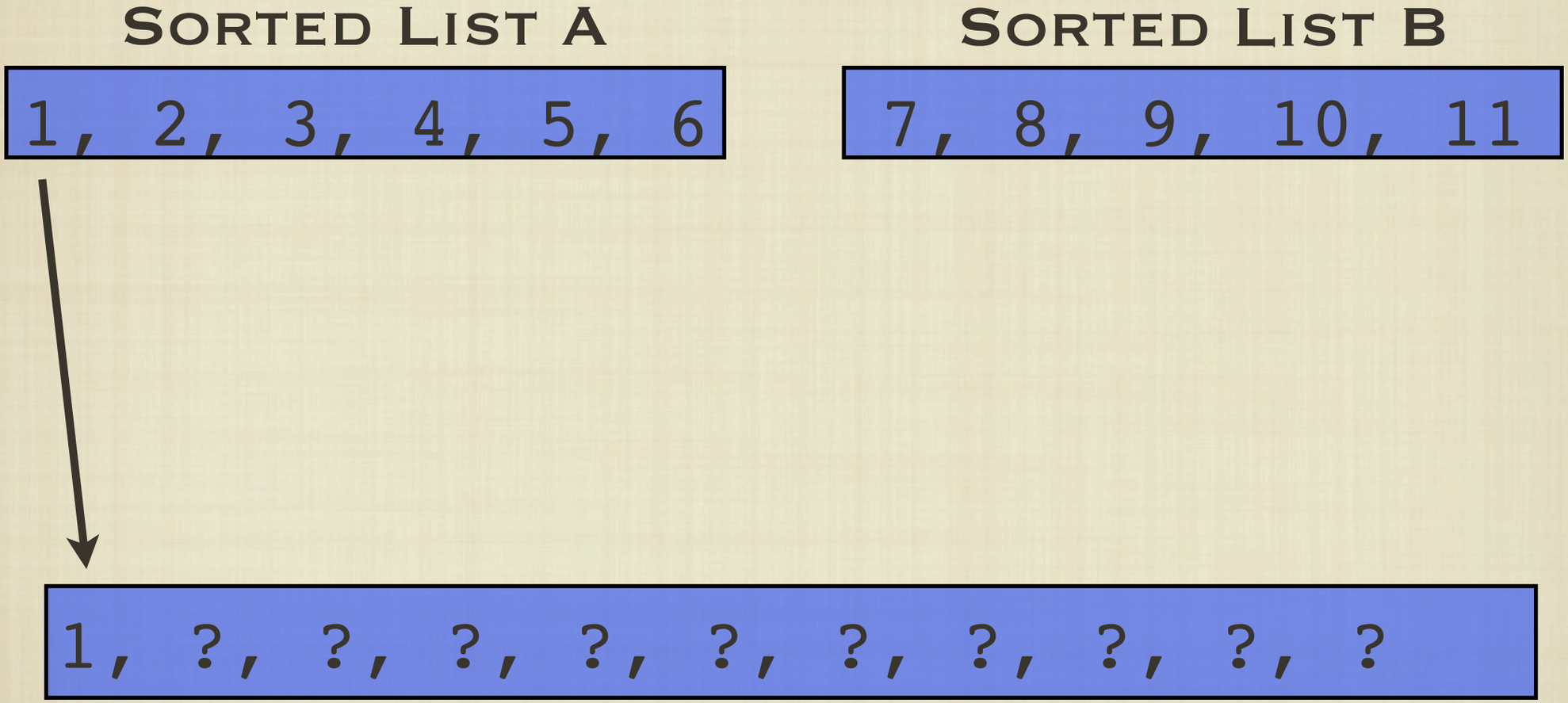
Merging Lists

Suppose that we instead had a list that had two sorted halves. Could we do better?



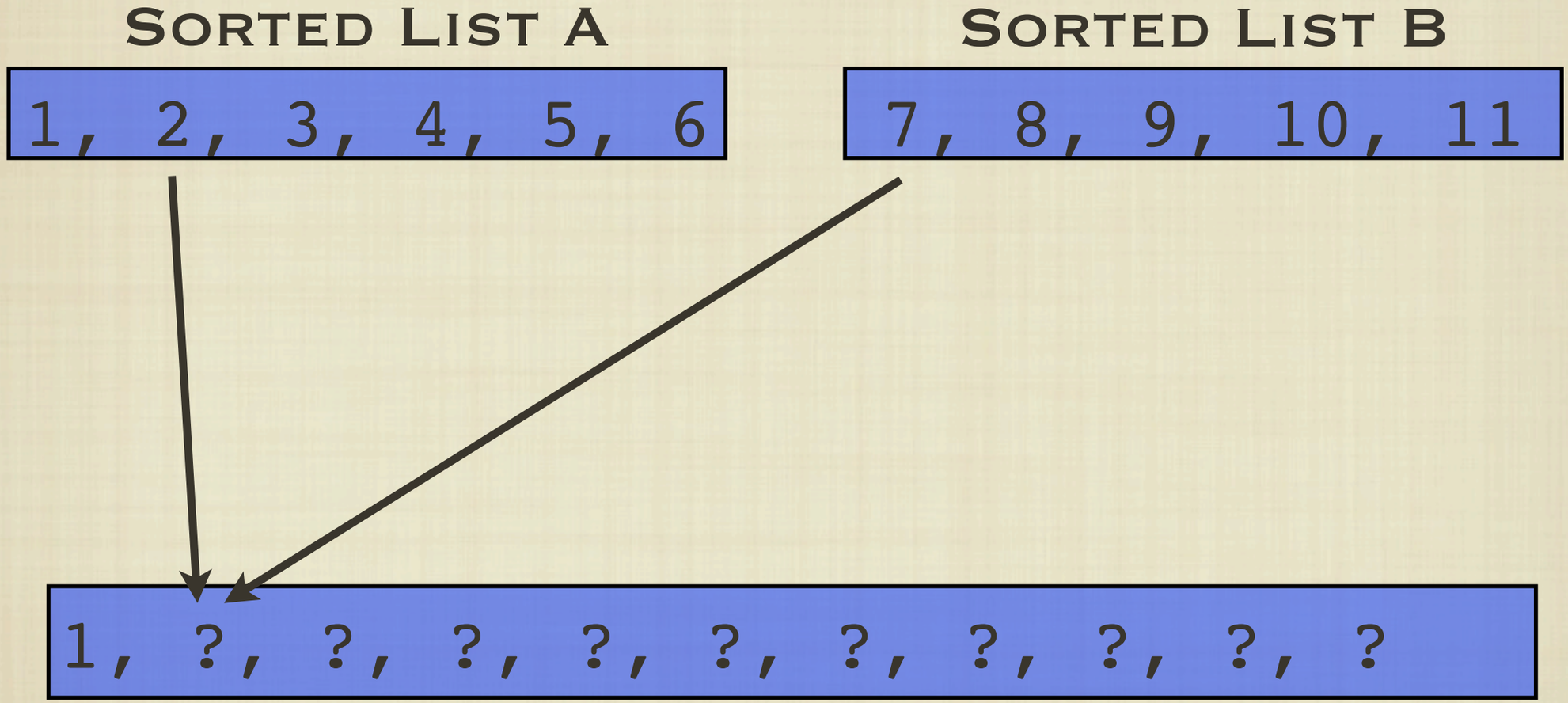
Merging Lists

Suppose that we instead had a list that had two sorted halves. Could we do better?



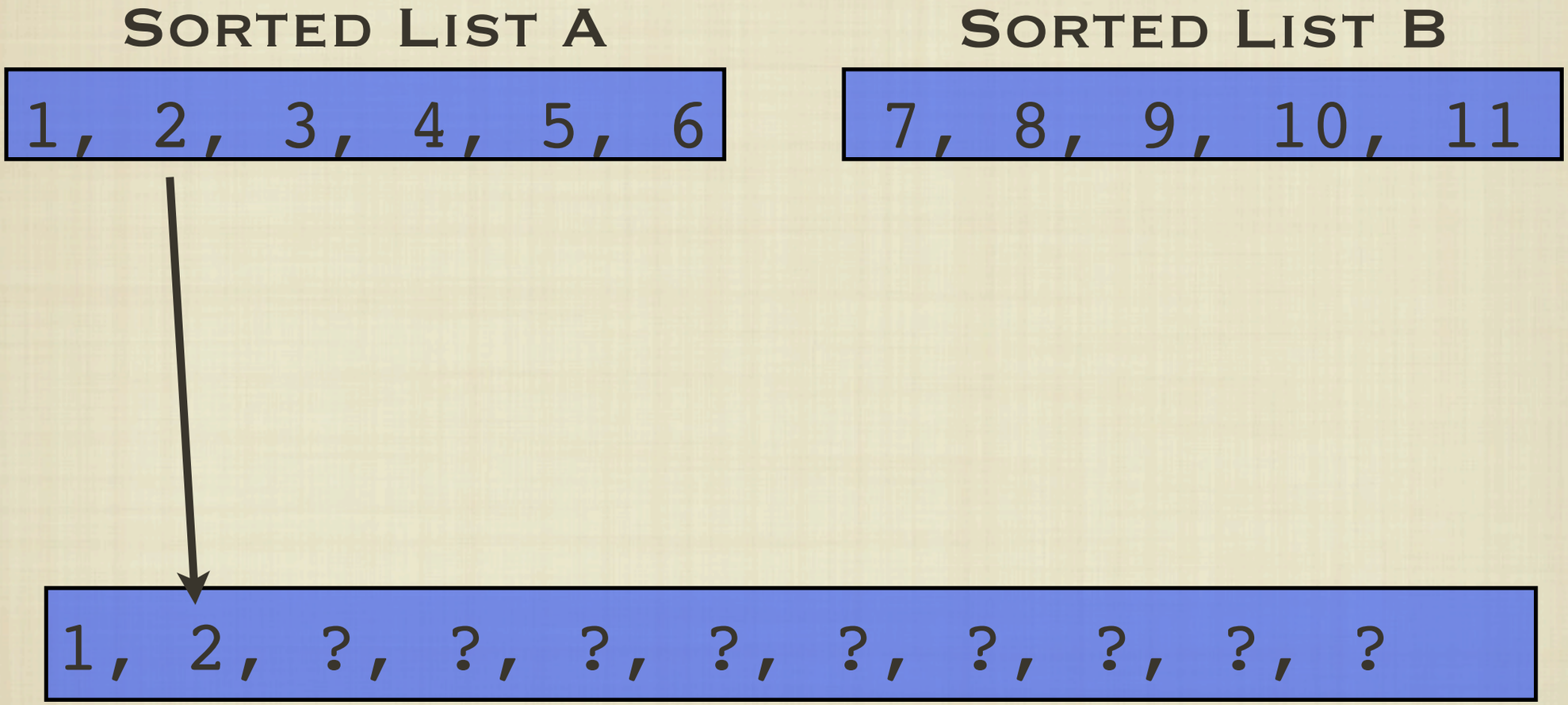
Merging Lists

Suppose that we instead had a list that had two sorted halves. Could we do better?



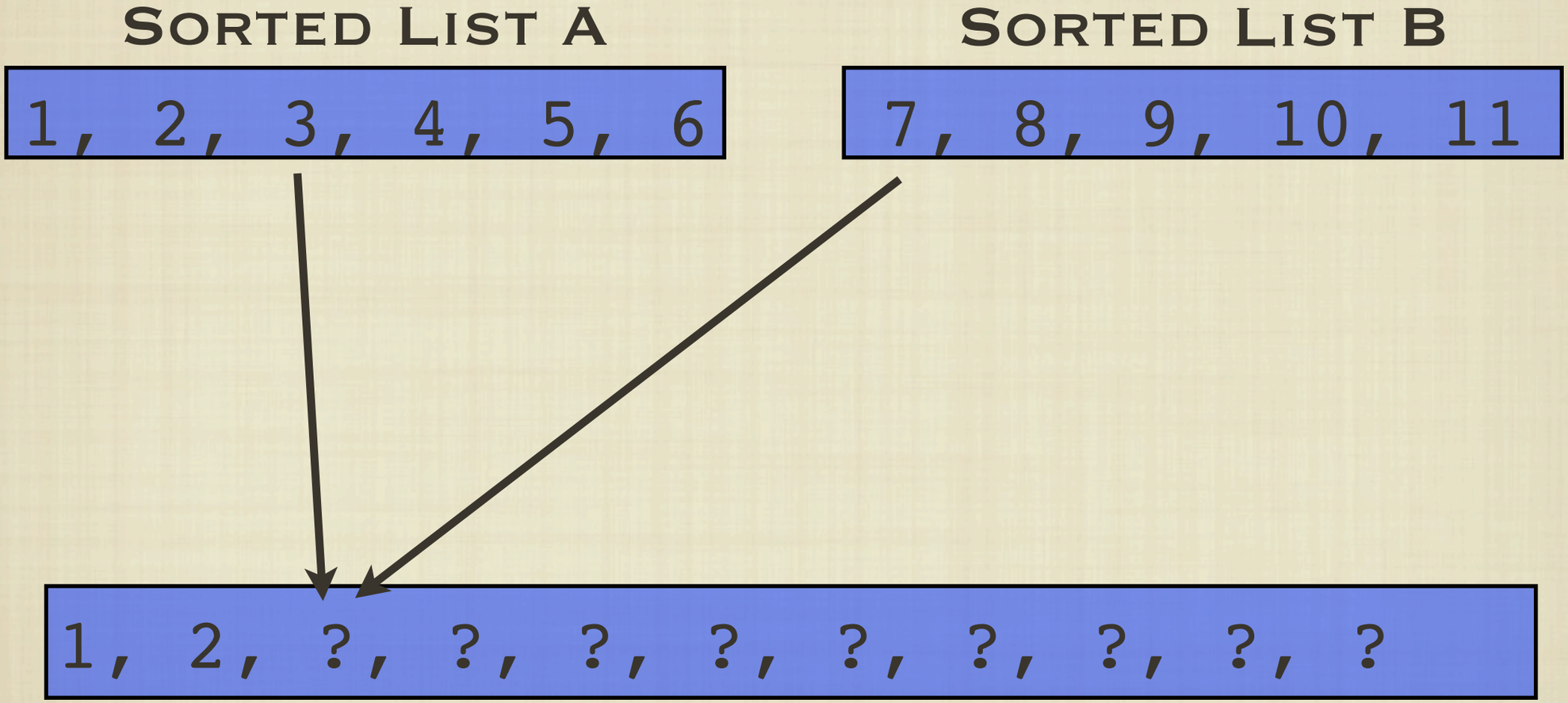
Merging Lists

Suppose that we instead had a list that had two sorted halves. Could we do better?



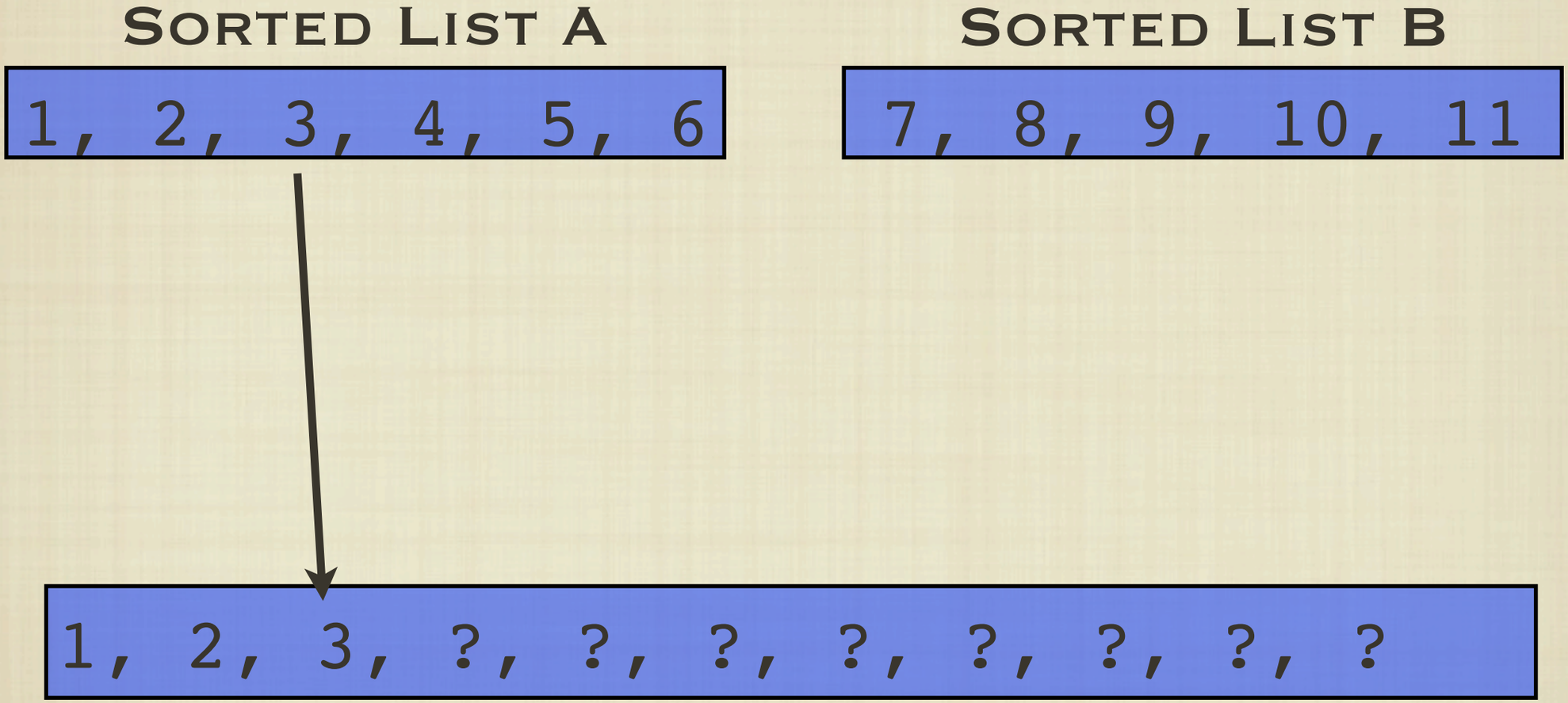
Merging Lists

Suppose that we instead had a list that had two sorted halves. Could we do better?



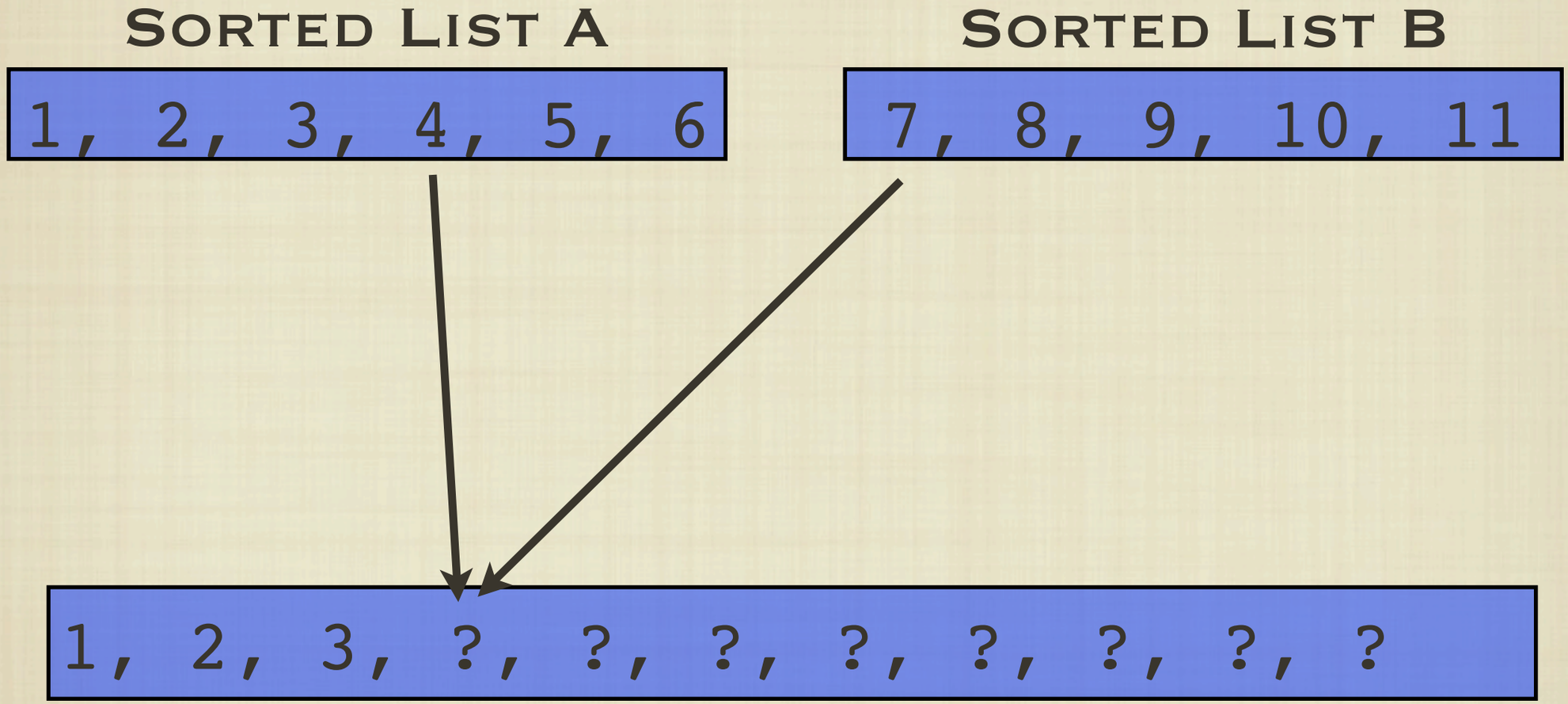
Merging Lists

Suppose that we instead had a list that had two sorted halves. Could we do better?



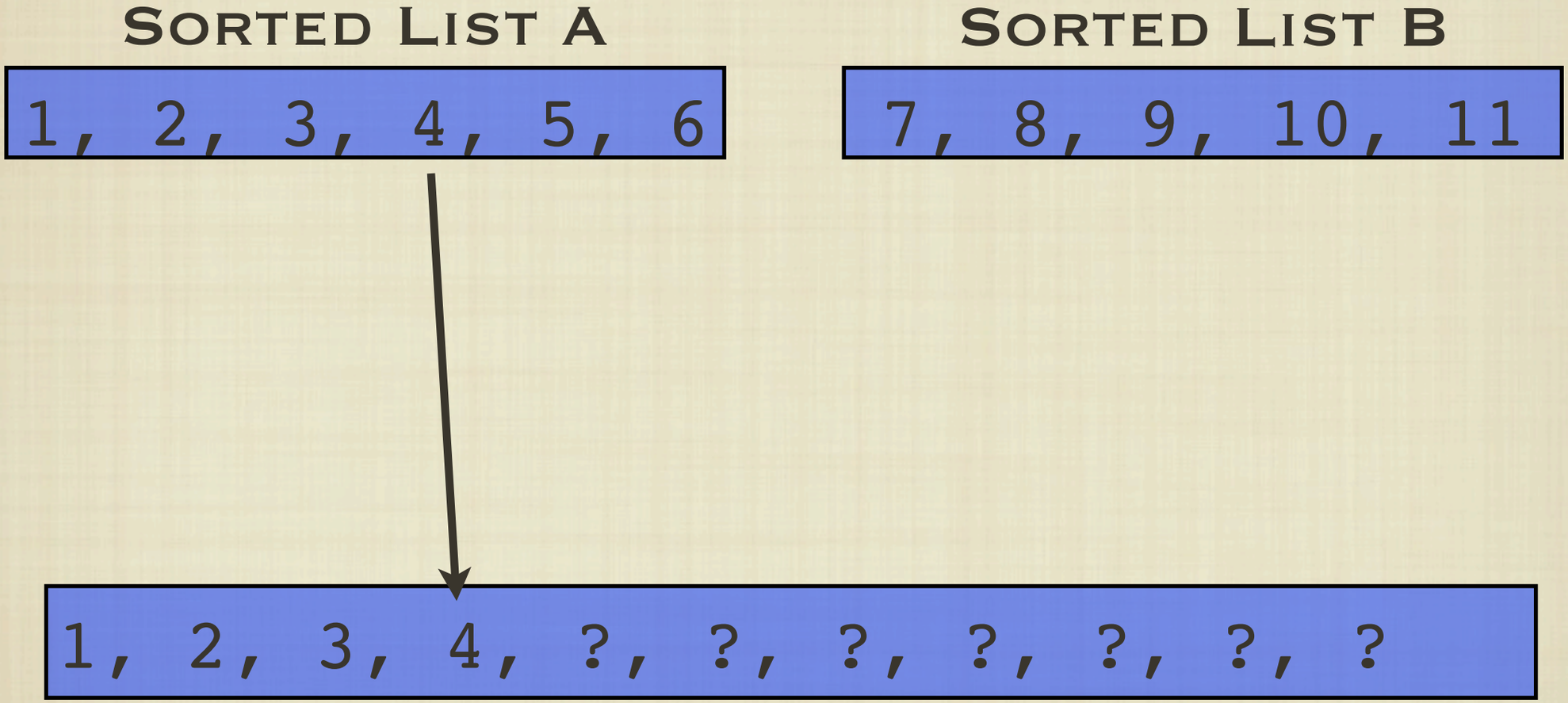
Merging Lists

Suppose that we instead had a list that had two sorted halves. Could we do better?



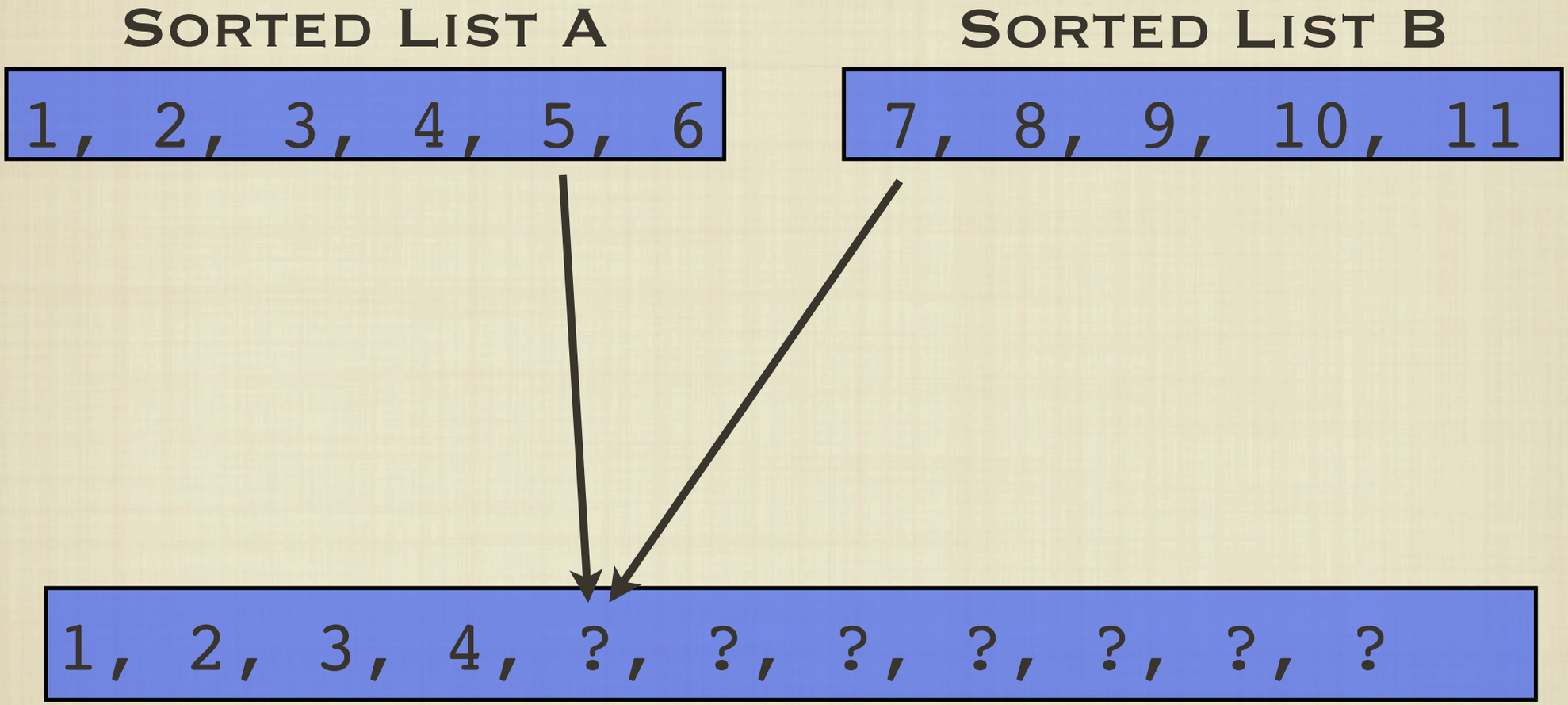
Merging Lists

Suppose that we instead had a list that had two sorted halves. Could we do better?



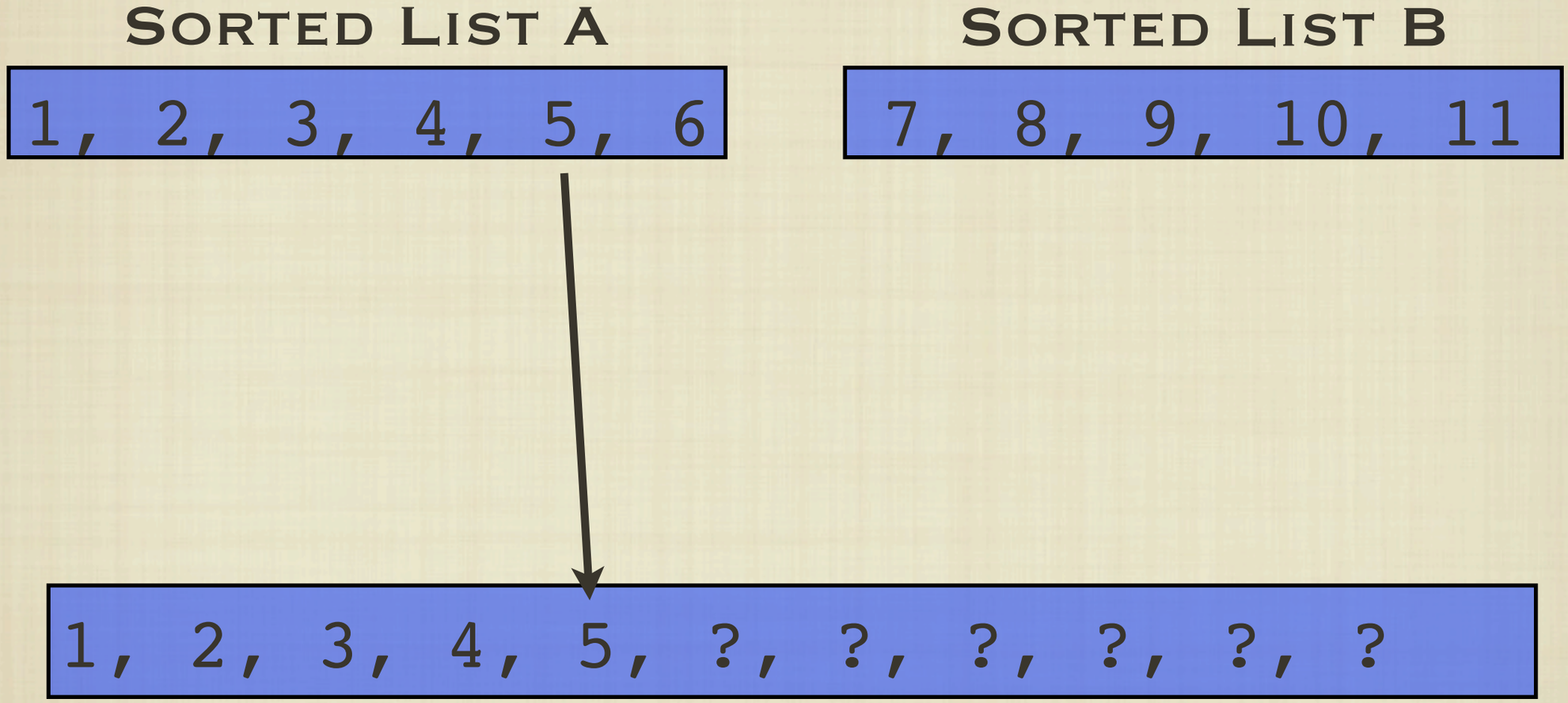
Merging Lists

Suppose that we instead had a list that had two sorted halves. Could we do better?



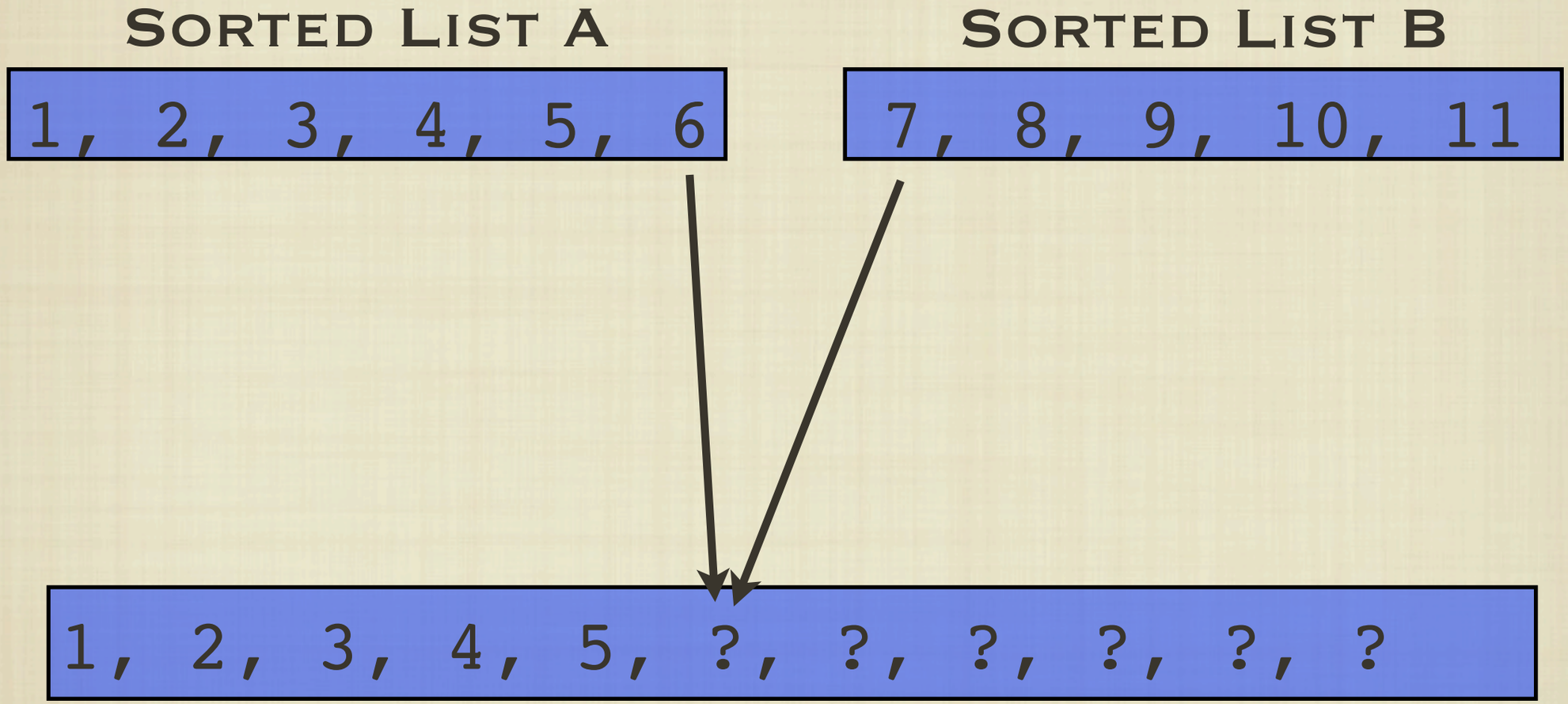
Merging Lists

Suppose that we instead had a list that had two sorted halves. Could we do better?



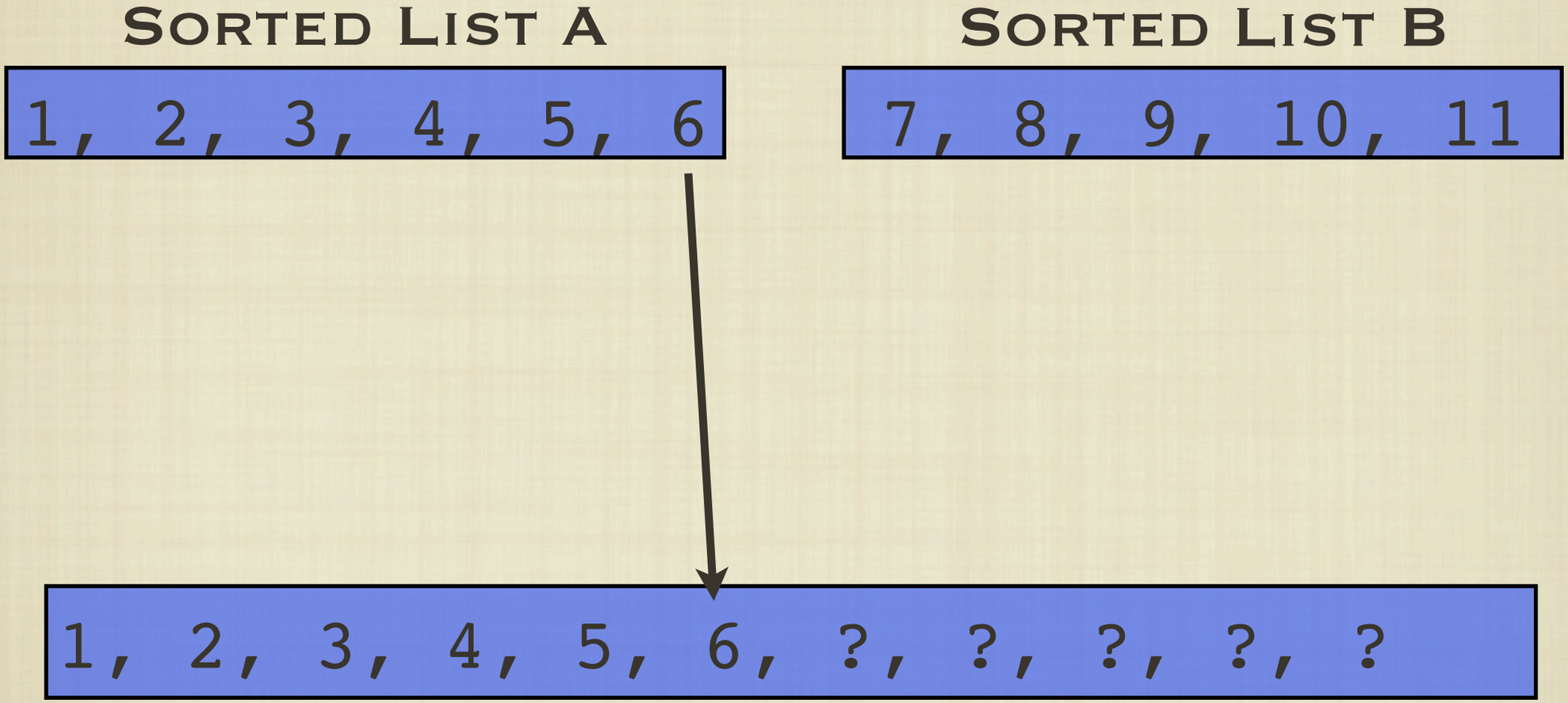
Merging Lists

Suppose that we instead had a list that had two sorted halves. Could we do better?



Merging Lists

Suppose that we instead had a list that had two sorted halves. Could we do better?



Merging Lists

Suppose that we instead had a list that had two sorted halves. Could we do better?

Sorted List A

```
1, 2, 3, 4, 5, 6
```

Sorted List B

```
7, 8, 9, 10, 11
```

```
1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11
```

The key idea is to scan through both lists, while moving the smallest element to a new list. If we finish scanning either list, the rest of the other list is appended to the result.

Merging Lists

Suppose that we instead had a list that had two sorted halves. Could we do better?

Algorithm:

1. Start at the beginning of both lists.

2. Move the smaller element to the result list, and consider the next element.

3. Repeat until one list is exhausted.

4. Put the other list at the end of the result list.

Merging Lists

Algorithm:

1. Start at the beginning of both lists.

2. Move the smaller element to the result list, and consider the next element.

3. Repeat until one list is exhausted.

4. Put the other list at the end of the result list.

Does this always produce a sorted list?

Merging Lists

Algorithm:

1. Start at the beginning of both lists.

2. Move the smaller element to the result list, and consider the next element.

3. Repeat until one list is exhausted.

4. Put the other list at the end of the result list.

Does this always produce a sorted list? How long does it take?

Merging Lists

Algorithm:

1. Start at the beginning of both lists.

2. Move the smaller element to the result list, and consider the next element.

3. Repeat until one list is exhausted.

4. Put the other list at the end of the result list.

Does this always produce a sorted list? How long does it take? For two lists with a total of n items, cn time.

Merging Lists

Algorithm:

1. Start at the beginning of both lists.

2. Move the smaller element to the result list, and consider the next element.

3. Repeat until one list is exhausted.

4. Put the other list at the end of the result list.

What is the point of doing this? Aren't we trying to sort the list?

Merge Sort

Suppose that we know how to merge two sorted lists.
Then, we can sort recursively:

Merge Sort:

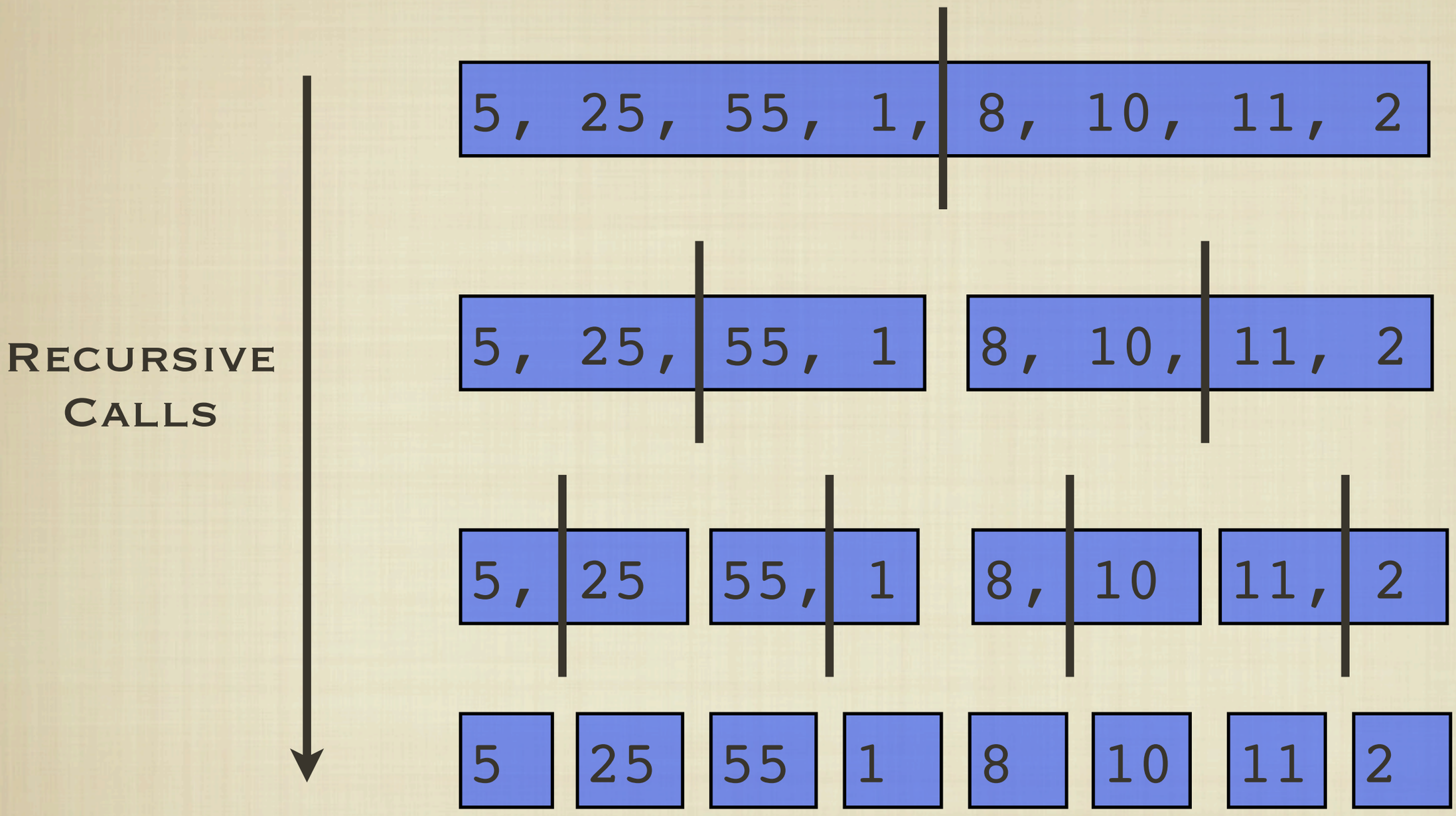
1. Split the given list into two equal parts.

2. Recursively sort each half.

3. Merge the sorted halves and return the result.

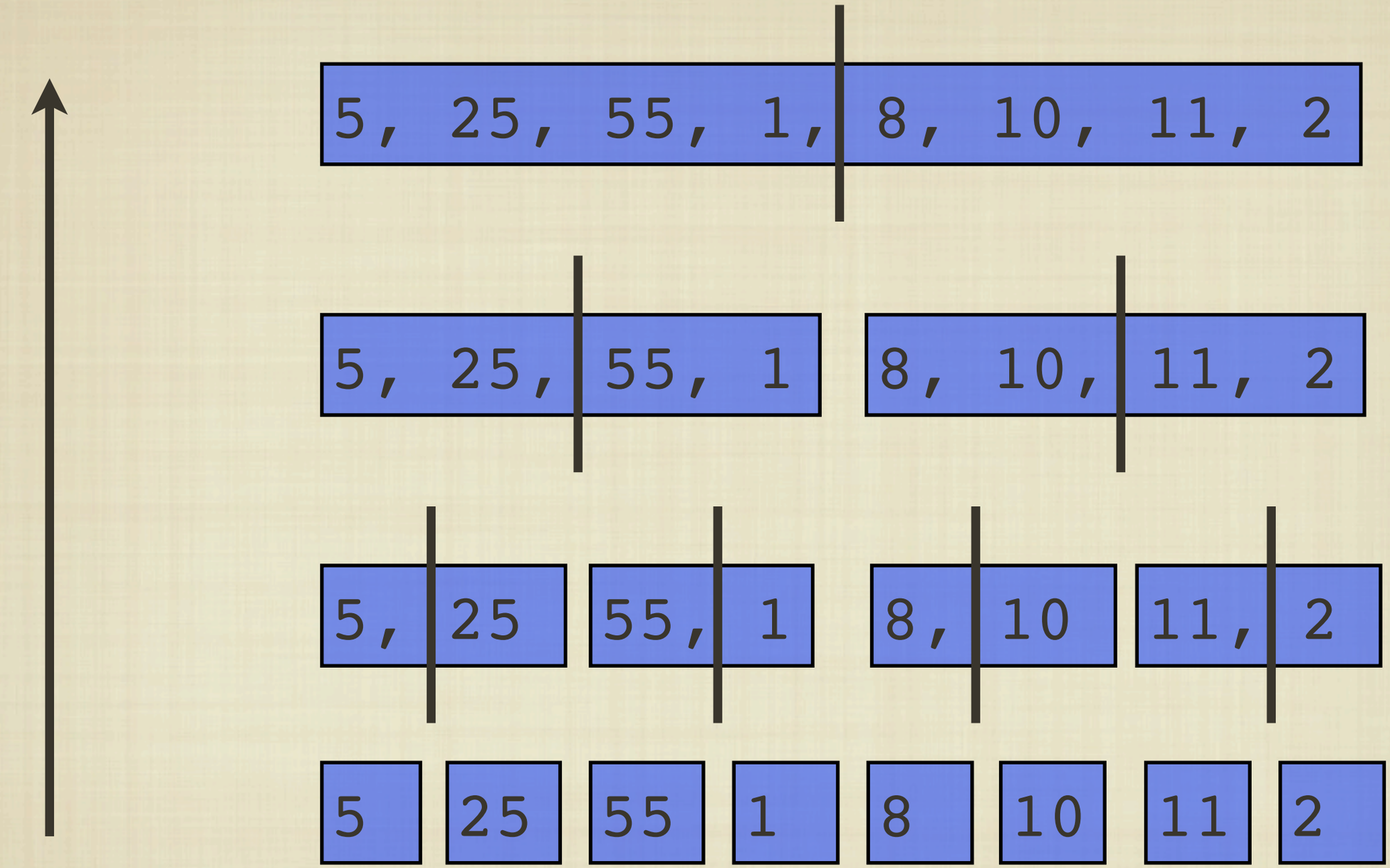
How could this possibly work?

Merge Sort



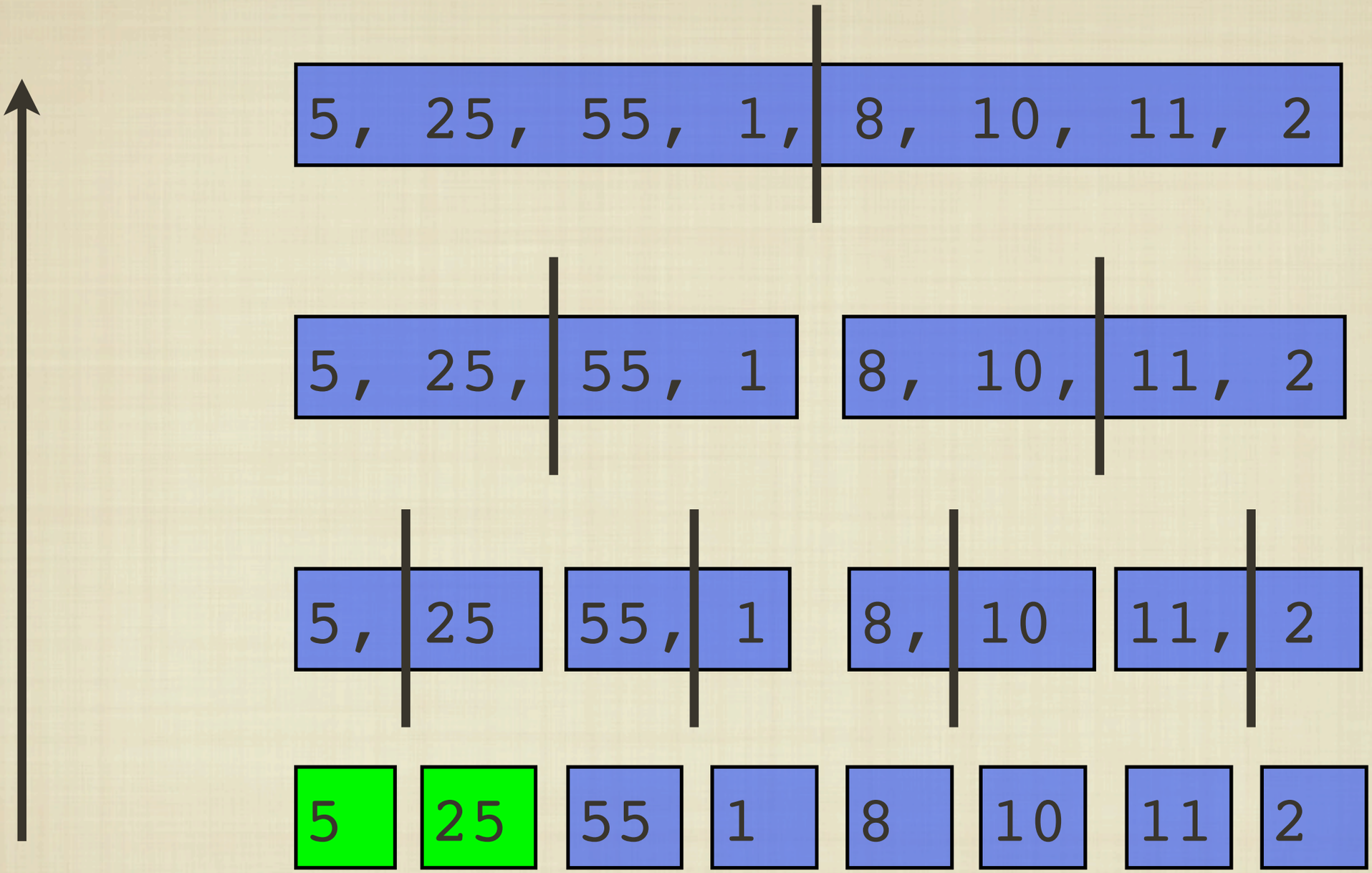
Actually, not a lot is happening in the recursive calls. So where is the sorting happening?

Merge Sort



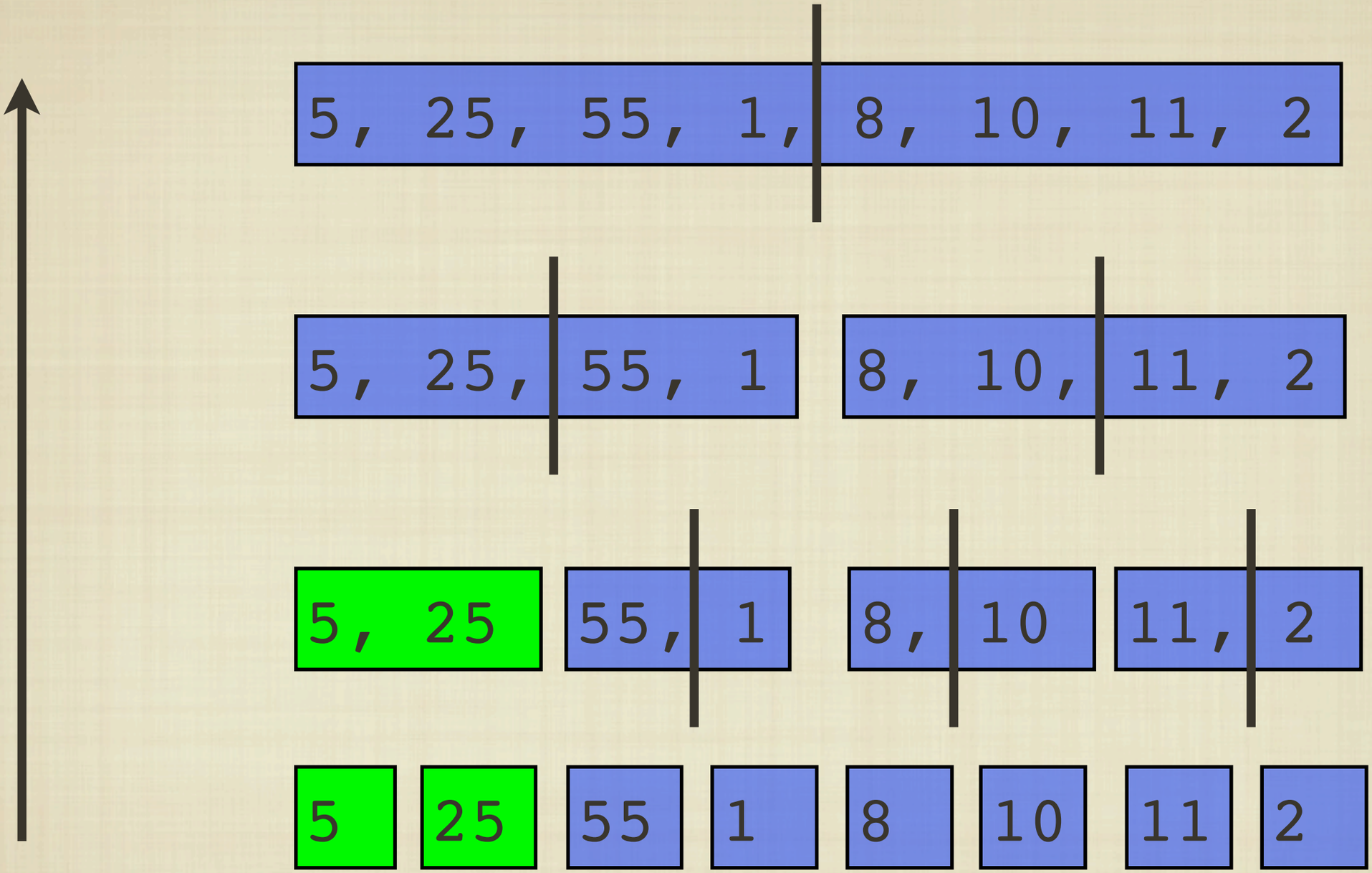
The merge step is actually doing all of the work!

Merge Sort



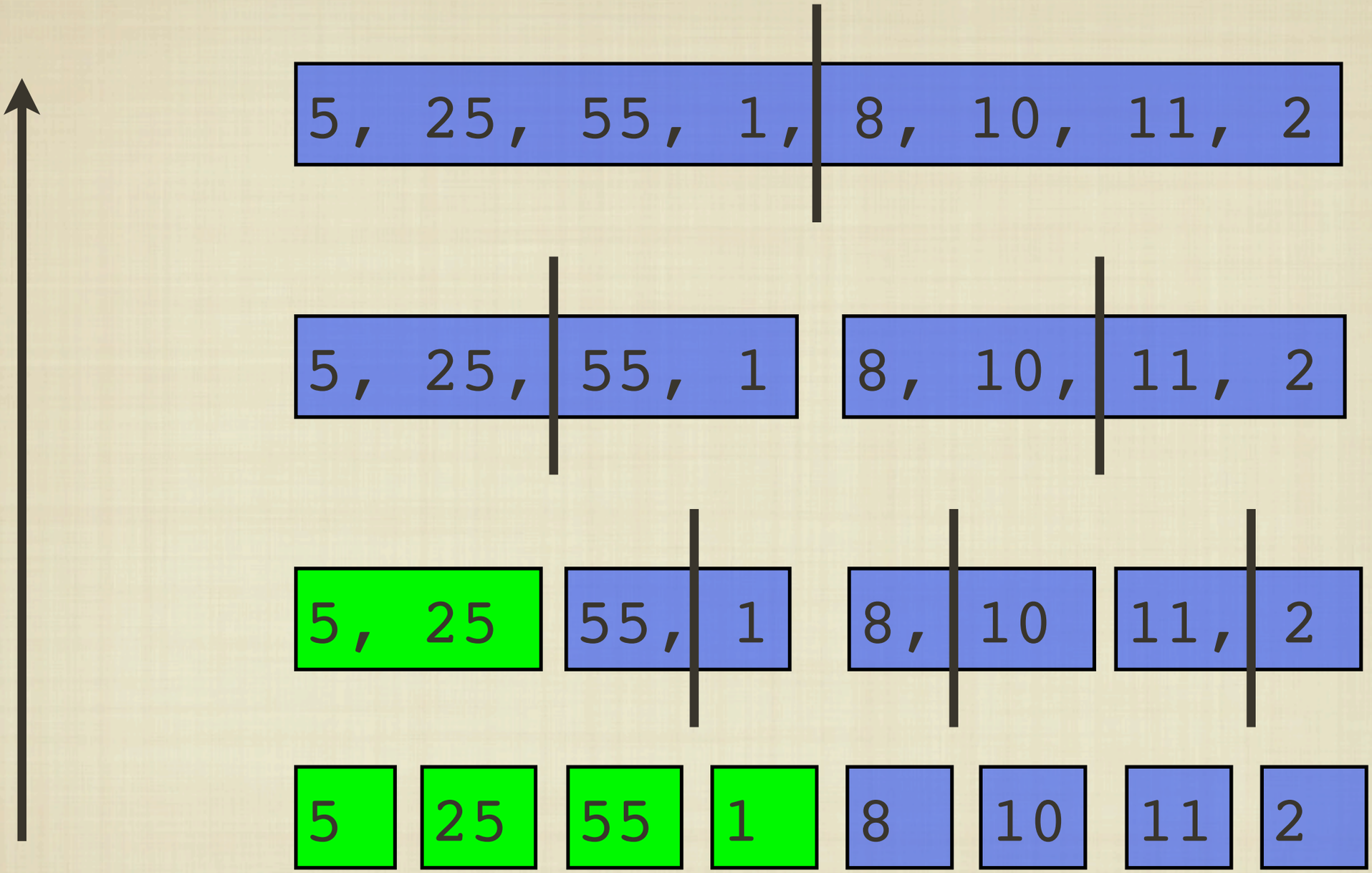
The merge step is actually doing all of the work!

Merge Sort



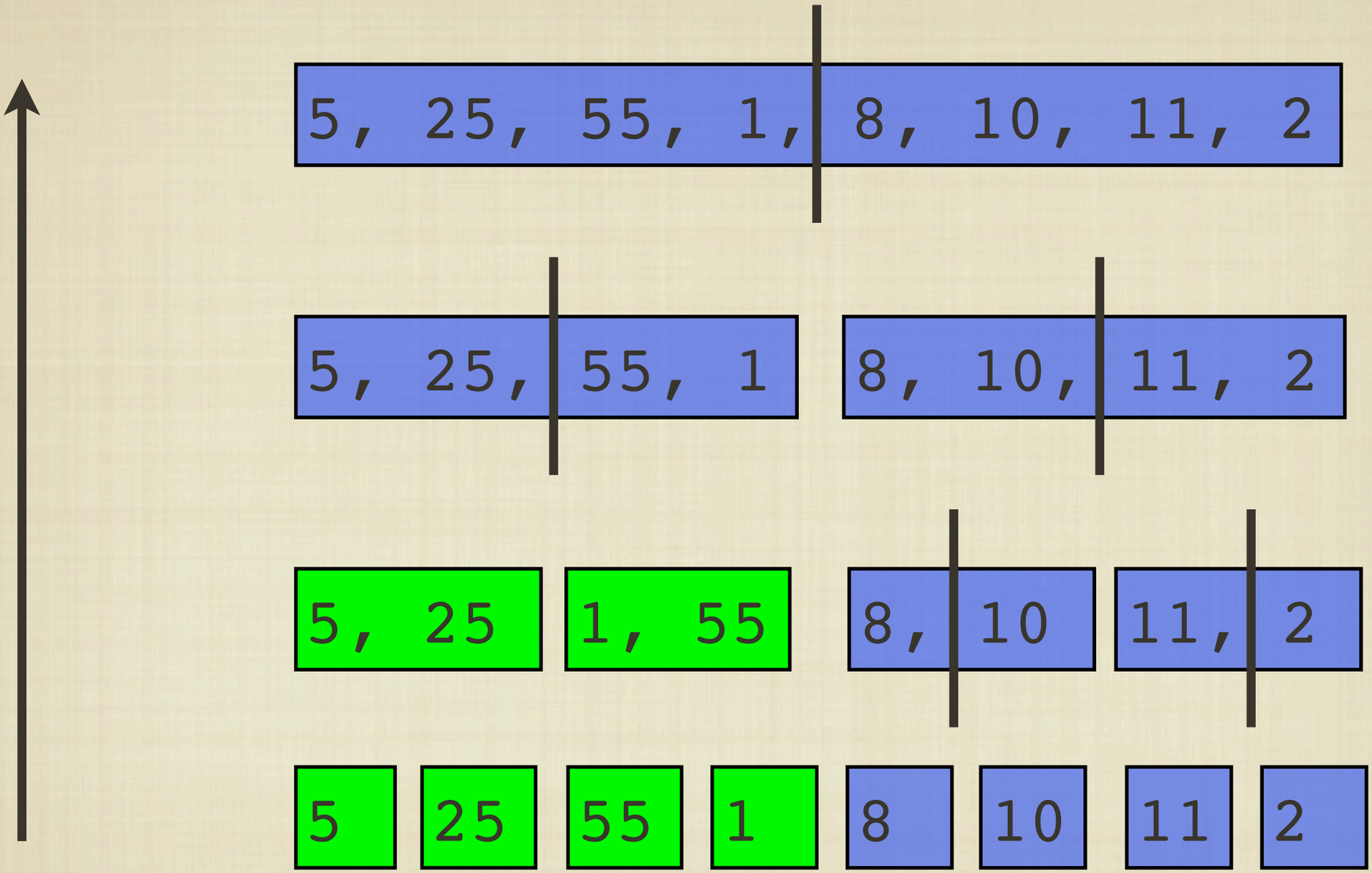
The merge step is actually doing all of the work!

Merge Sort



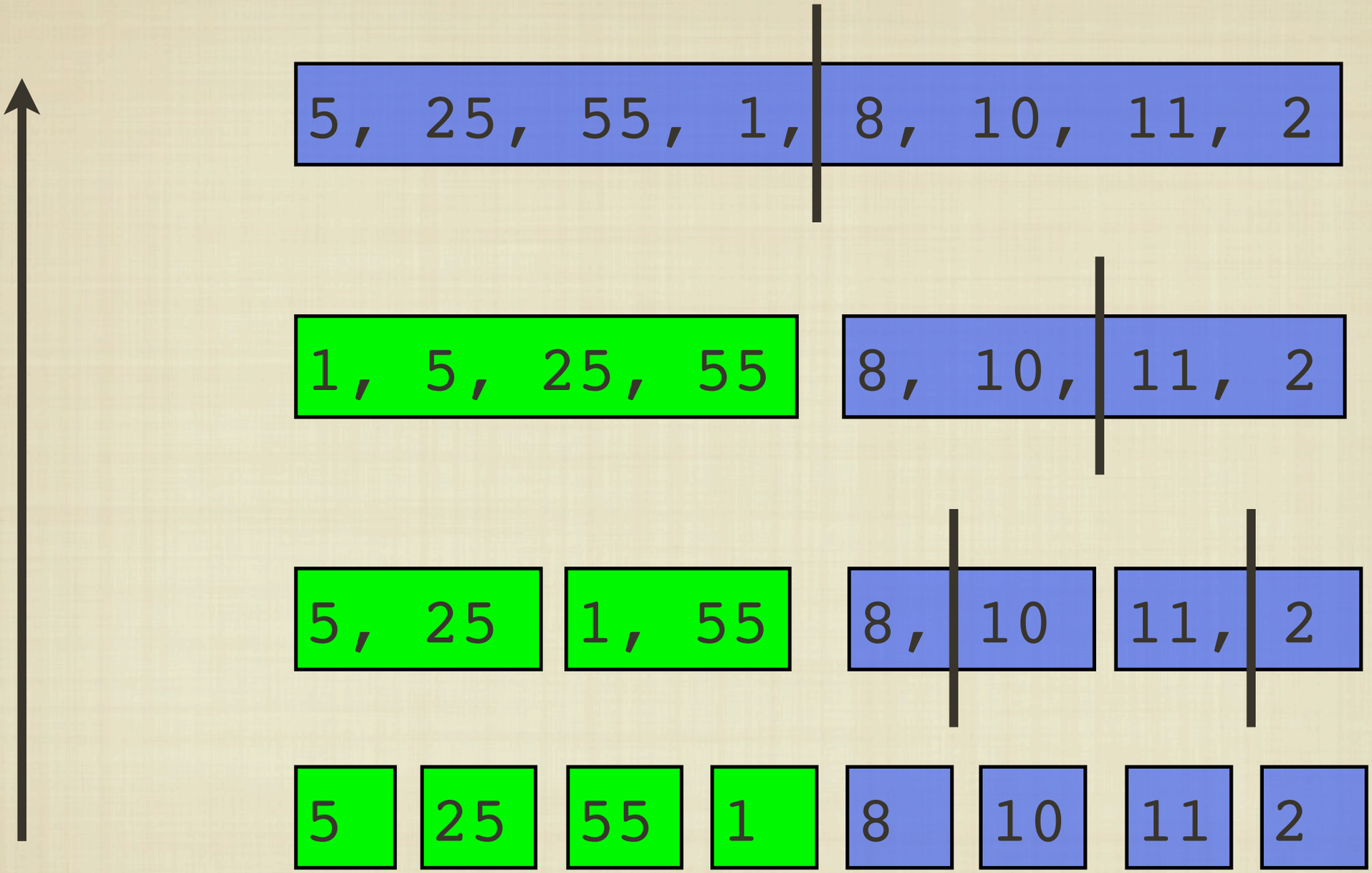
The merge step is actually doing all of the work!

Merge Sort



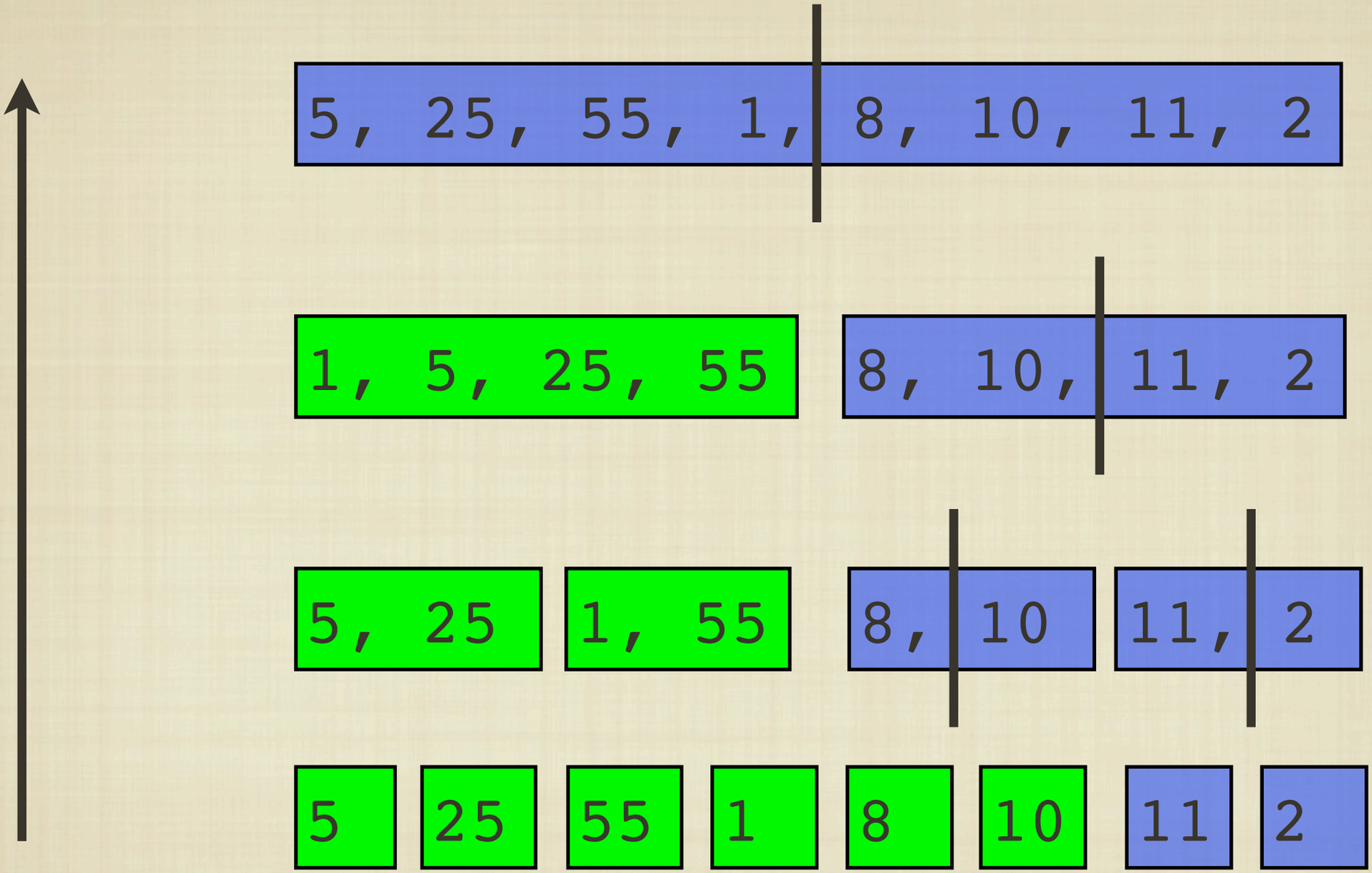
The merge step is actually doing all of the work!

Merge Sort



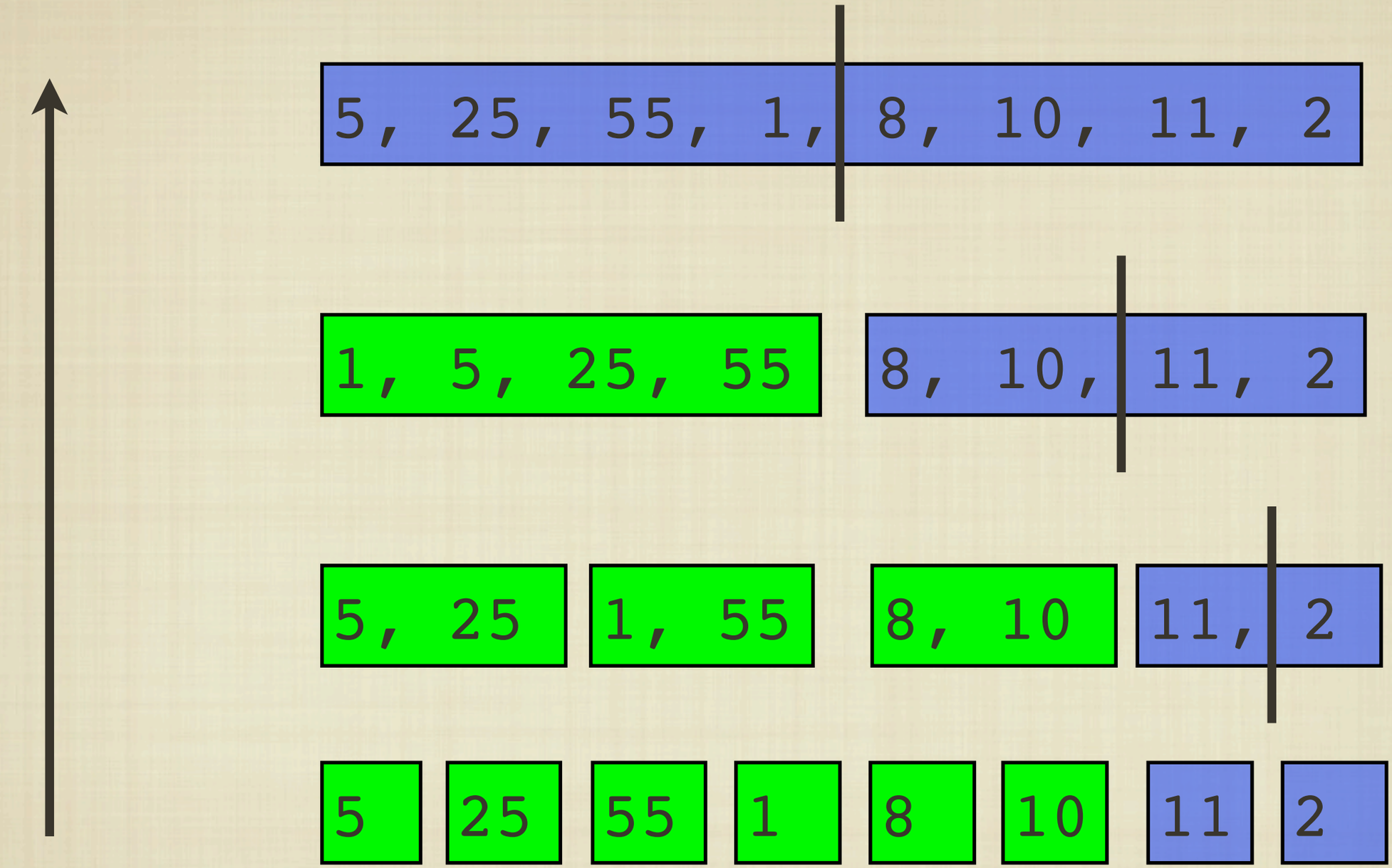
The merge step is actually doing all of the work!

Merge Sort



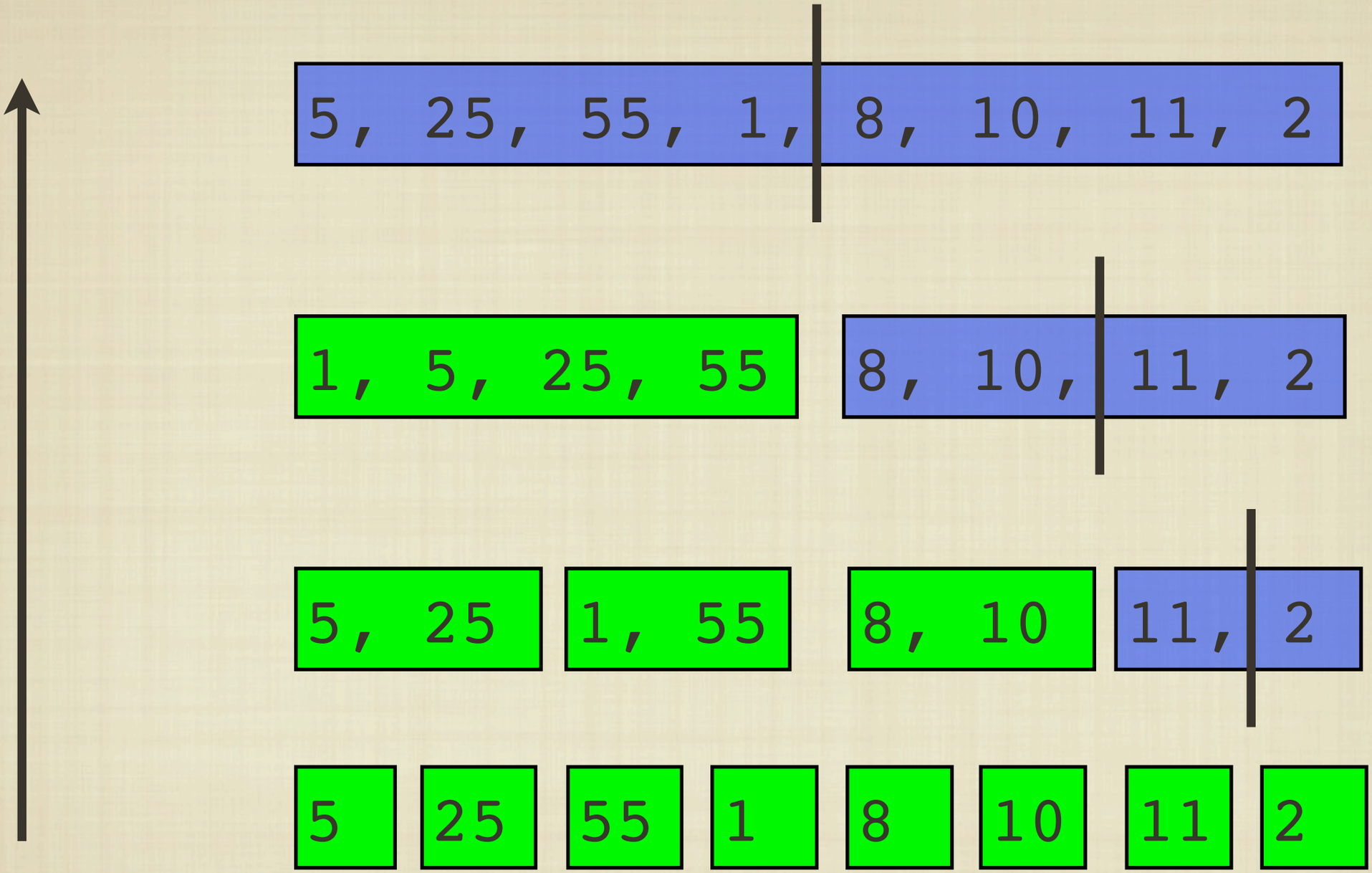
The merge step is actually doing all of the work!

Merge Sort



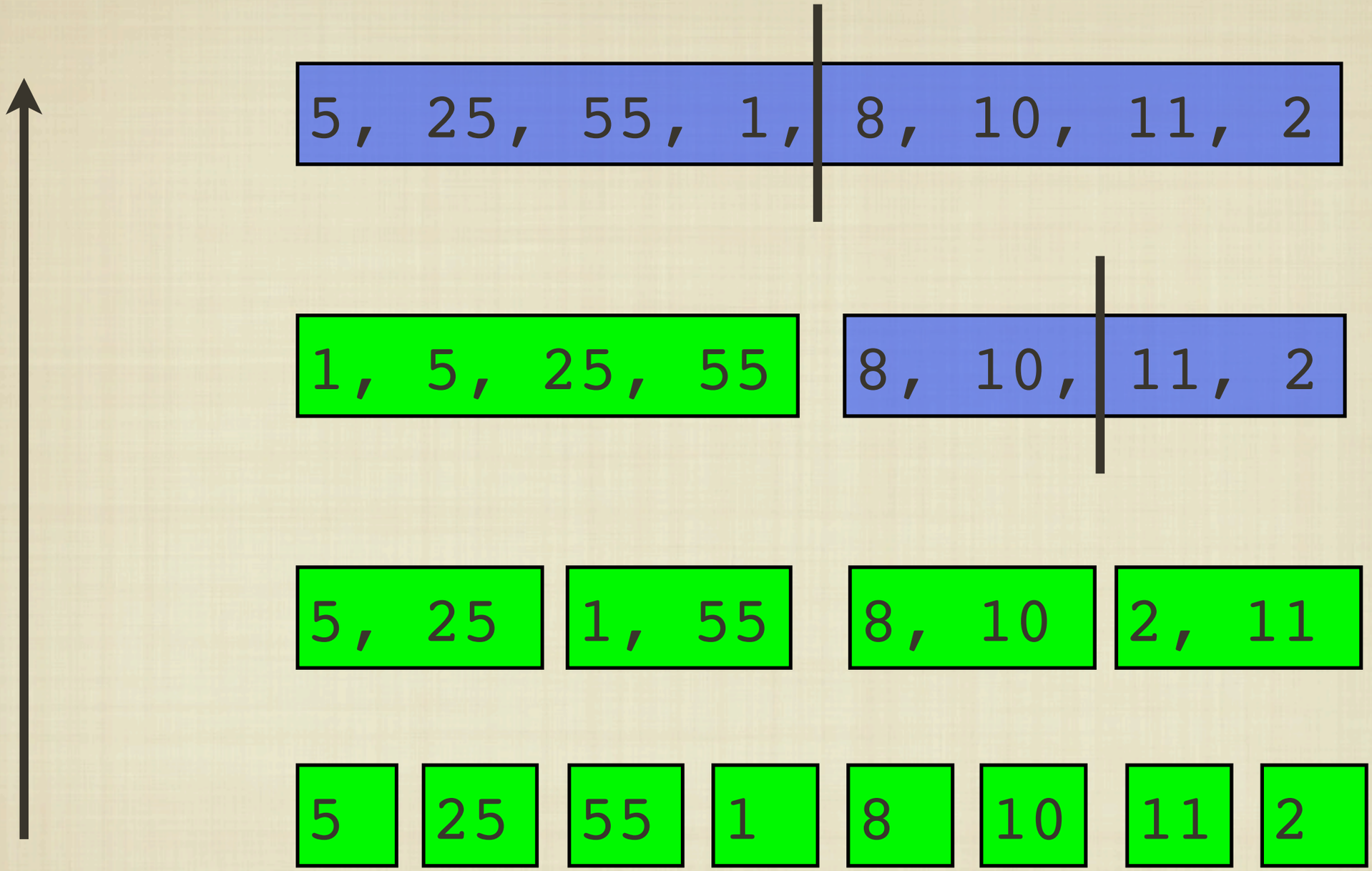
The merge step is actually doing all of the work!

Merge Sort



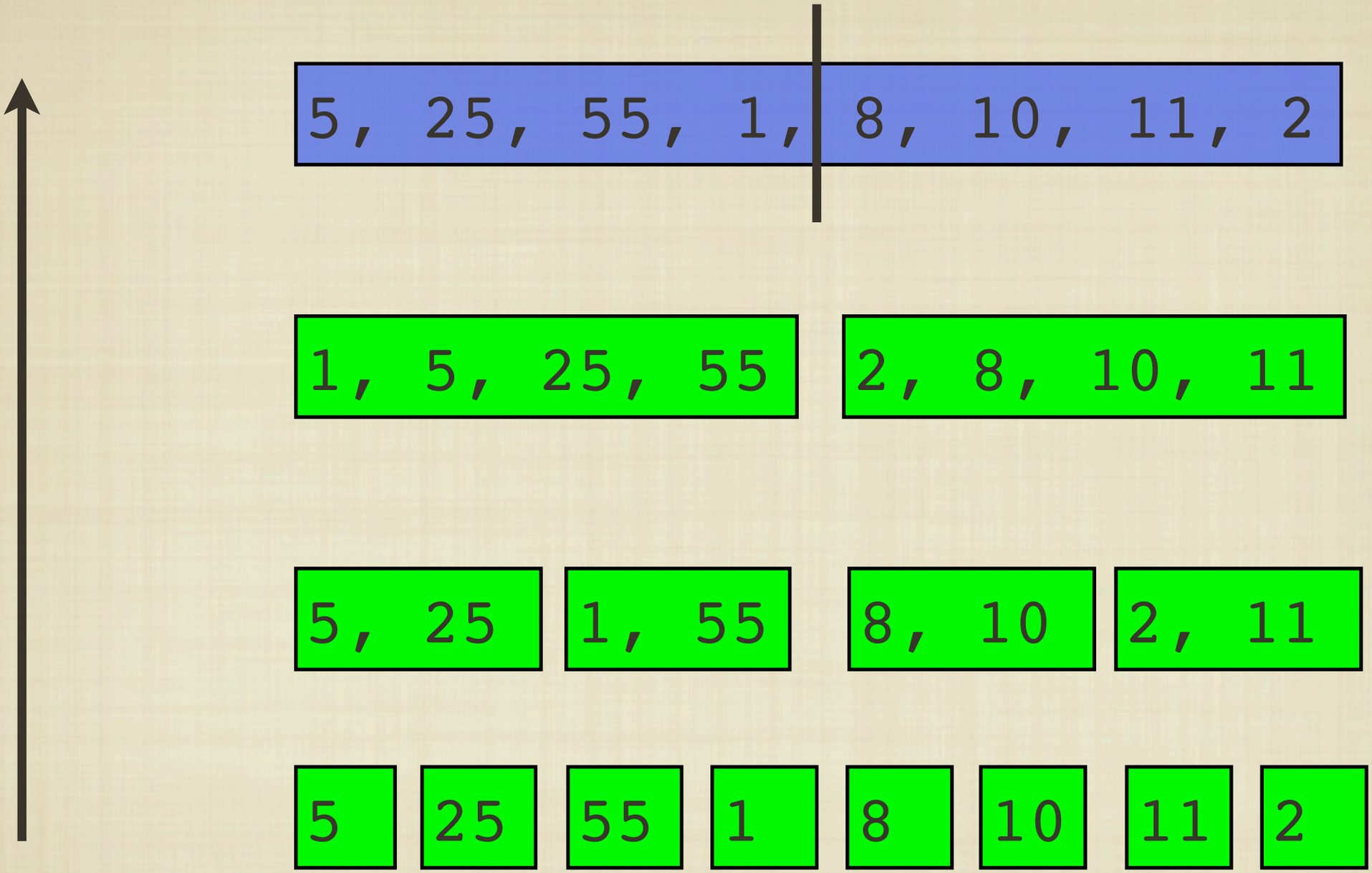
The merge step is actually doing all of the work!

Merge Sort



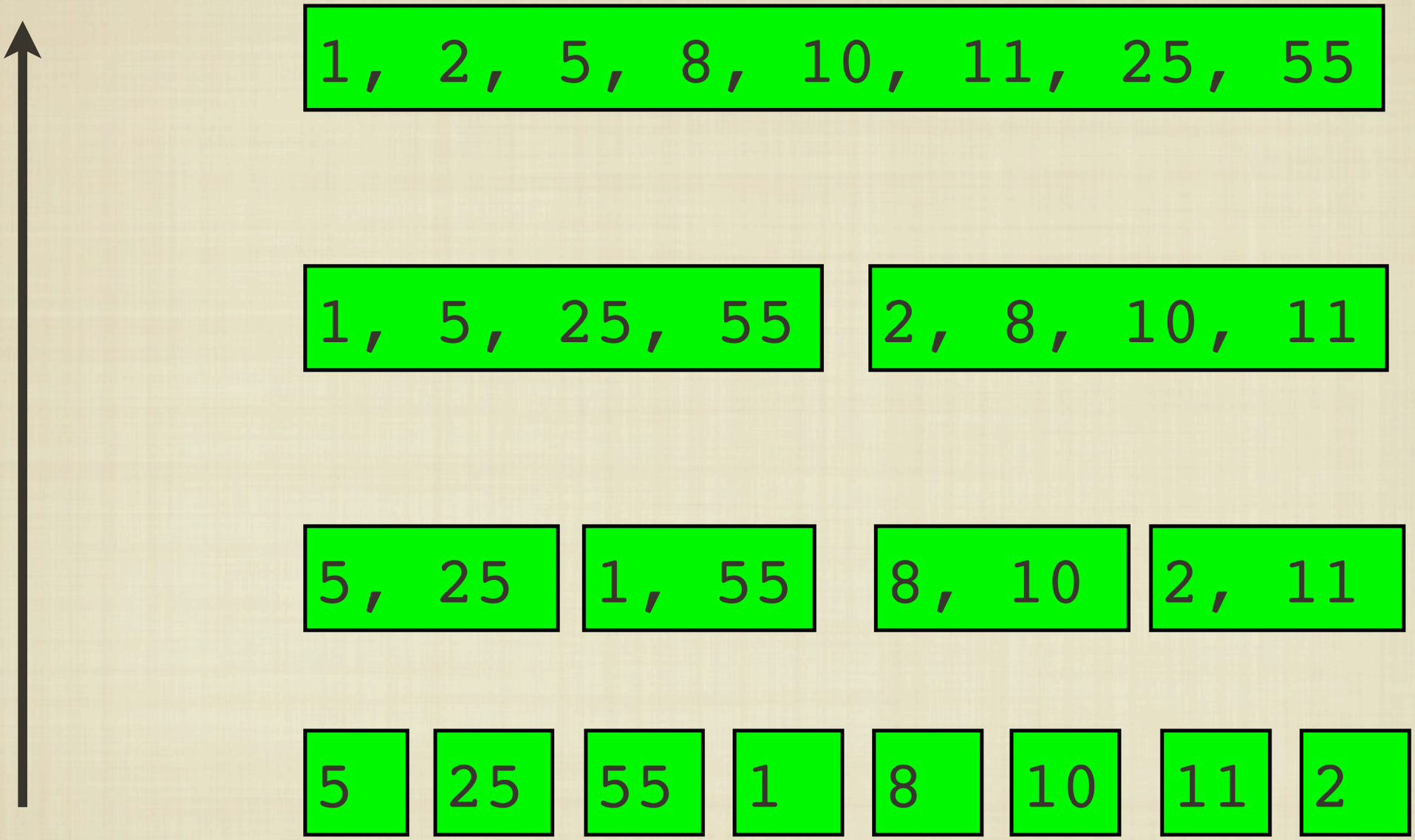
The merge step is actually doing all of the work!

Merge Sort



The merge step is actually doing all of the work!

Merge Sort



The merge step is actually doing all of the work!

Merge Sort Analysis

It is easiest to analyze the performance of this algorithm using a recurrence relation. Suppose that merging two lists of size $n/2$ requires cn operations. Then the recurrence that defines the performance of MergeSort is:

$$T(1) = c$$

$$T(n) = cn + 2T(n/2)$$

Merge Sort Analysis

It is easiest to analyze the performance of this algorithm using a recurrence relation. Suppose that merging two lists of size $n/2$ requires cn operations. Then the recurrence that defines the performance of Merge Sort is:

$$T(1) = c$$

$$T(n) = cn + 2T(n/2)$$

$$T(n) = cn + 2(cn/2 + 2T(n/4))$$

Merge Sort Analysis

It is easiest to analyze the performance of this algorithm using a recurrence relation. Suppose that merging two lists of size $n/2$ requires cn operations. Then the recurrence that defines the performance of Merge Sort is:

$$T(1) = c$$

$$T(n) = cn + 2T(n/2)$$

$$T(n) = cn + cn + 4T(n/4)$$

Merge Sort Analysis

It is easiest to analyze the performance of this algorithm using a recurrence relation. Suppose that merging two lists of size $n/2$ requires cn operations. Then the recurrence that defines the performance of Merge Sort is:

$$T(1) = c$$

$$T(n) = cn + 2T(n/2)$$

$$T(n) = cn + cn + 4T(n/4)$$

$$T(n) = cn + cn + 4(cn/4 + 2T(n/8))$$

Merge Sort Analysis

It is easiest to analyze the performance of this algorithm using a recurrence relation. Suppose that merging two lists of size $n/2$ requires cn operations. Then the recurrence that defines the performance of Merge Sort is:

$$T(1) = c$$

$$T(n) = cn + 2T(n/2)$$

$$T(n) = cn + cn + 4T(n/4)$$

$$T(n) = cn + cn + cn + 8T(n/8))$$

Merge Sort Analysis

It is easiest to analyze the performance of this algorithm using a recurrence relation. Suppose that merging two lists of size $n/2$ requires cn operations. Then the recurrence that defines the performance of Merge Sort is:

$$T(1) = c$$

$$T(n) = cn + 2T(n/2)$$

$$T(n) = cn + cn + 4T(n/4)$$

$$T(n) = cn + cn + cn + 8T(n/8))$$

$$T(n) = cn + cn + cn + \cdots + 2^k T(1)$$

Merge Sort Analysis

It is easiest to analyze the performance of this algorithm using a recurrence relation. Suppose that merging two lists of size $n/2$ requires cn operations. Then the recurrence that defines the performance of Merge Sort is:

$$T(1) = c$$

$$T(n) = cn + 2T(n/2)$$

$$T(n) = cn + cn + 4T(n/4)$$

$$T(n) = cn + cn + cn + 8T(n/8))$$

$$T(n) = cn + cn + cn + \cdots + c \cdot 2^k$$

Merge Sort Analysis

It is easiest to analyze the performance of this algorithm using a recurrence relation. Suppose that merging two lists of size $n/2$ requires cn operations. Then the recurrence that defines the performance of Merge Sort is:

$$T(1) = c$$

$$T(n) = cn + 2T(n/2)$$

$$T(n) = cn + cn + 4T(n/4)$$

$$T(n) = cn + cn + cn + 8T(n/8))$$

$$T(n) = \underbrace{cn + cn + cn + \cdots +}_{\text{how many terms?}} c \cdot 2^k$$

what is k?

Merge Sort Analysis

It is easiest to analyze the performance of this algorithm using a recurrence relation. Suppose that merging two lists of size $n/2$ requires cn operations. Then the recurrence that defines the performance of Merge Sort is:

$$T(1) = c$$

$$T(n) = cn + 2T(n/2)$$

$$T(n) = cn + cn + 4T(n/4)$$

$$T(n) = cn + cn + cn + 8T(n/8))$$

$$T(n) = \underbrace{cn + cn + cn + \cdots +}_{\text{how many terms?}} c \cdot 2^k = cn(1 + \log_2 n)$$

what is k?

Merge Sort Analysis

It is easiest to analyze the performance of this algorithm using a recurrence relation. Suppose that merging two lists of size $n/2$ requires cn operations. Then the recurrence that defines the performance of Merge Sort is:

$$T(1) = c$$

$$T(n) = cn + 2T(n/2)$$

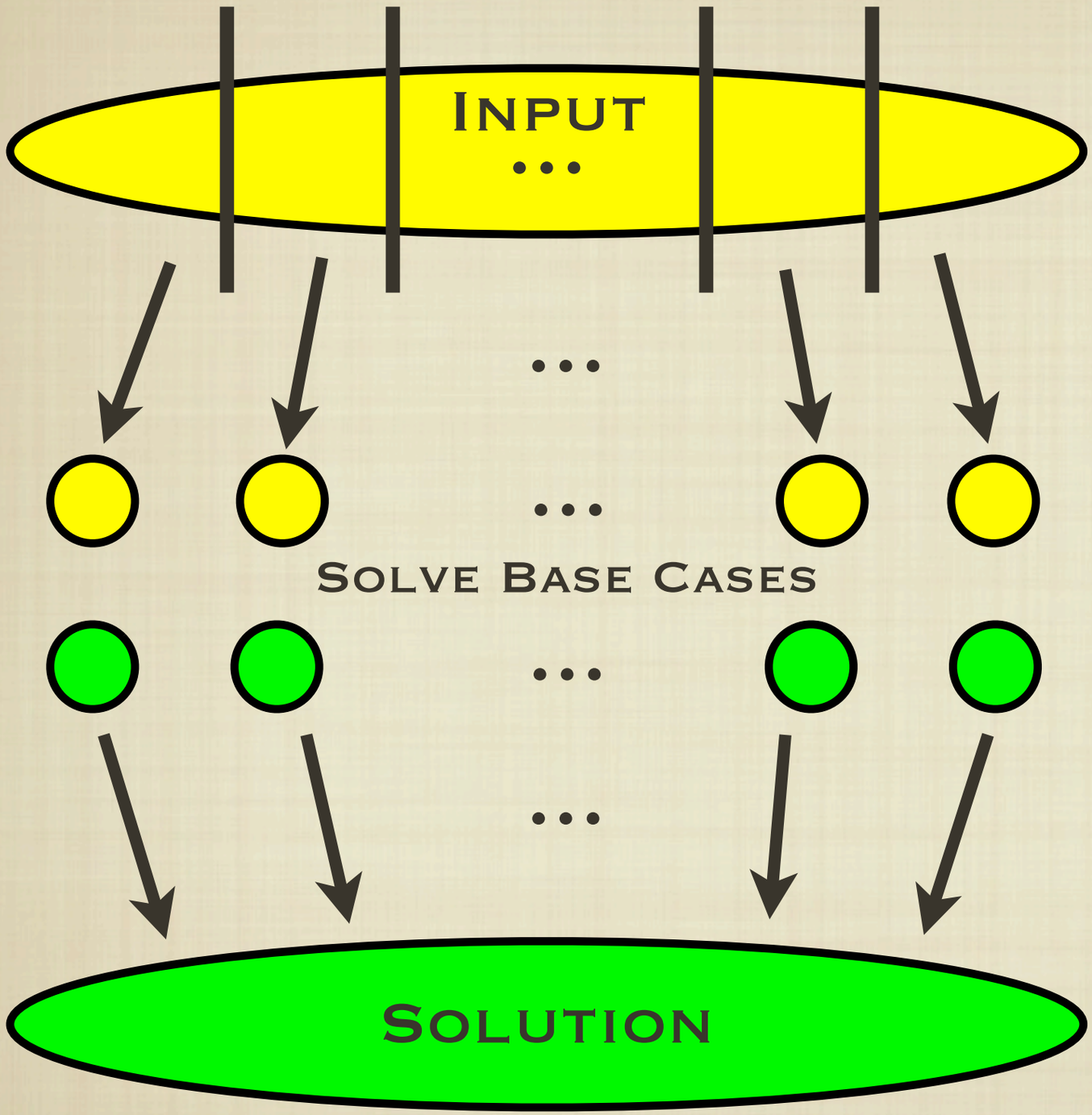
$$T(n) = cn + cn + 4T(n/4)$$

$$T(n) = cn + cn + cn + 8T(n/8))$$

$$T(n) = cn(1 + \log_2 n) = O(n \log_2 n)$$

Is this faster than selection sort? By how much?

"Divide-And-Conquer"



Divide-and-Conquer:

1. If the input is small enough, solve.

2. Otherwise, split input into parts.

3. Recursively solve each part.

4. Merge solutions.

Implementing these algorithms is easy because they are recursive.

Analysis of Divide-and-Conquer

The divide-and-conquer paradigm for algorithms is easy to implement because we can use recursion, but the trick to is have an efficient merge step.

How can we analyze these kinds of algorithms?

Generalized Divide-and-Conquer Recurrence

$$T(1) = c \qquad T(n) = f(n) + a \cdot T(n/b)$$

- $f(n)$: work to merge
- a: number of recursive calls
- b: size of each split

Because of the “divide” step, these algorithms will often have a logarithmic term in the running time.

Real-World Sorting

Size	Selection Sort	Merge Sort	Tim Sort
10	0.000050	0.000079	0.000011
100	0.001681	0.001019	0.000045
1000	0.143361	0.014503	0.000623
10000	14.192897	0.172488	0.008769
100000	2638.207074	2.186937	0.206392

For randomly-ordered lists, Selection Sort cannot scale, but Merge Sort can easily handle lists with hundreds of thousands of items. The built-in `sort` is cleverly optimized to run even faster, although its theoretical worst-case performance is identical to Merge Sort.

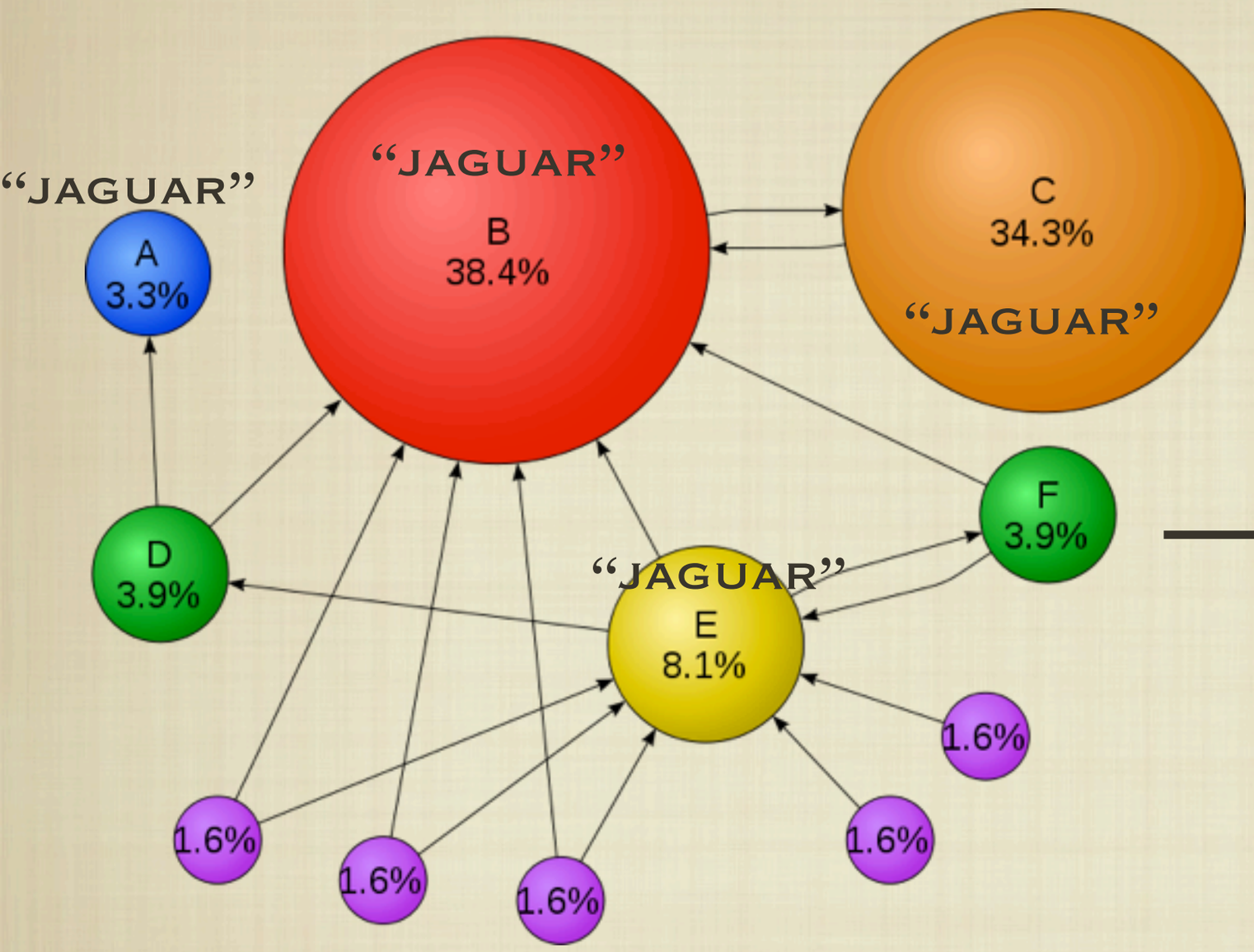
Real-World Sorting

Size	Selection Sort	Merge Sort	Tim Sort
10	0.000050	0.000079	0.000011
100	0.001681	0.001019	0.000045
1000	0.143361	0.014503	0.000623
10000	14.192897	0.172488	0.008769
100000	2638.207074	2.186937	0.206392

So, what is the point of sorting? Is it really so important to do quickly?

Yes! Sorting is probably the most commonly used “subroutine” in software, and the savings in work can add up drastically.

Google in a Nutshell



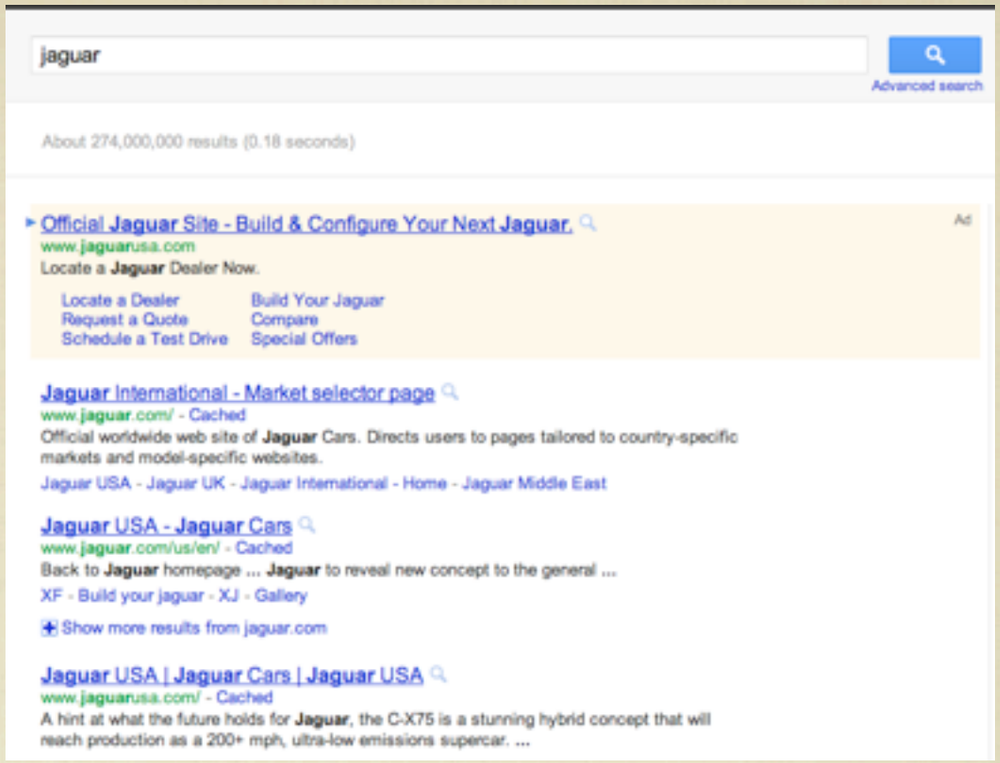
Results

Page B
Page C
Page E
Page A
.
.
.

Google processes the entire web and computes "PageRank" to determine which pages are most authoritative. The PageRank is essentially the chance that a random web-surfer would end up on a particular page.

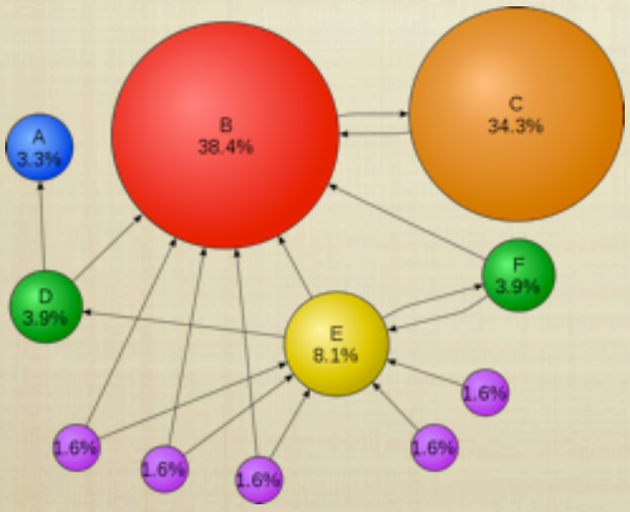
Google in a Nutshell

Query



Result

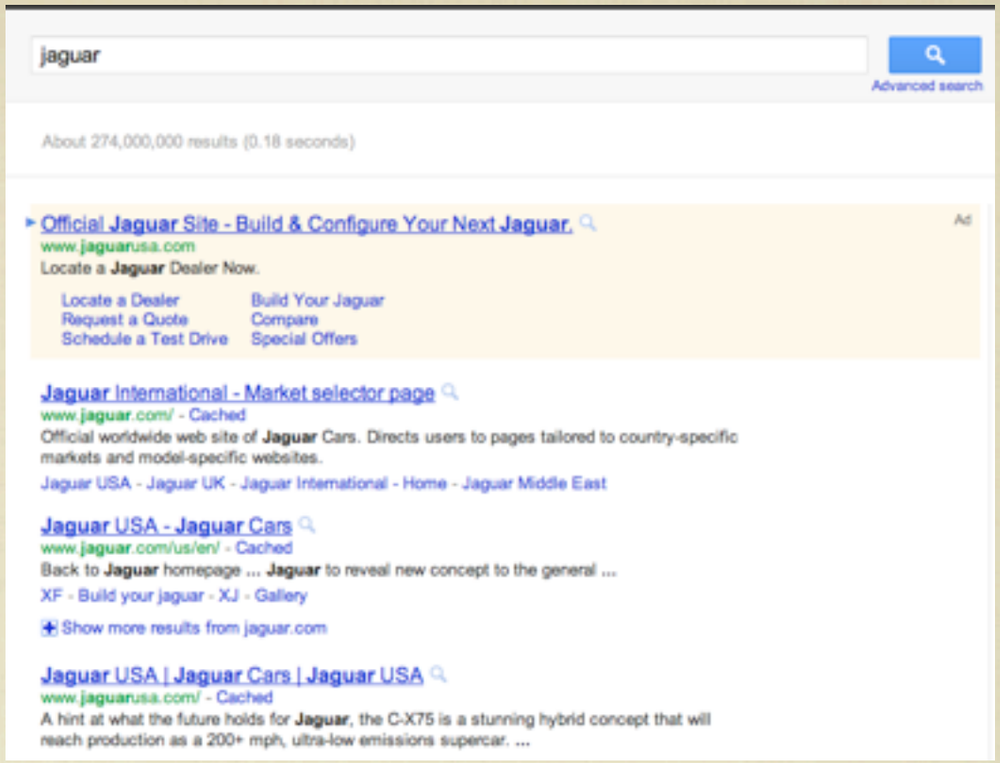
Google Data Center



1. Search for query keywords in mined pages.
2. Select a set of “matching” pages and ads.
3. Sort pages by PageRank and return results.

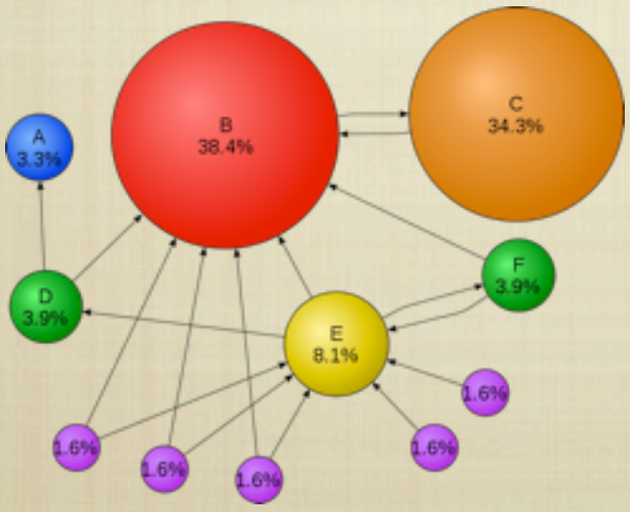
Google in a Nutshell

Query



Result

Google Data Center



1. Search for query keywords in mined pages.
2. Select a set of “matching” pages and ads.
3. Sort pages by PageRank and return results.

This is done 3,000,000,000 times a day.

Google Data Center on the Columbia River in Oregon.

An average Google query takes .2s. Suppose that 50% of the time was due to sorting, and that we are sorting about 10,000 items. What would happen if we substituted selection sort?

Recall that computation is work, and requires electricity. This is a major recurring cost for Google (2 billion kWh in 2010); they attempt to maximize the “revenue-per-query.”

Google is investing in “green” infrastructure, why?