Algorithm Analysis
Sorting II

Fall 2013
Carola Wenk

Is Selection Sort Practical?

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

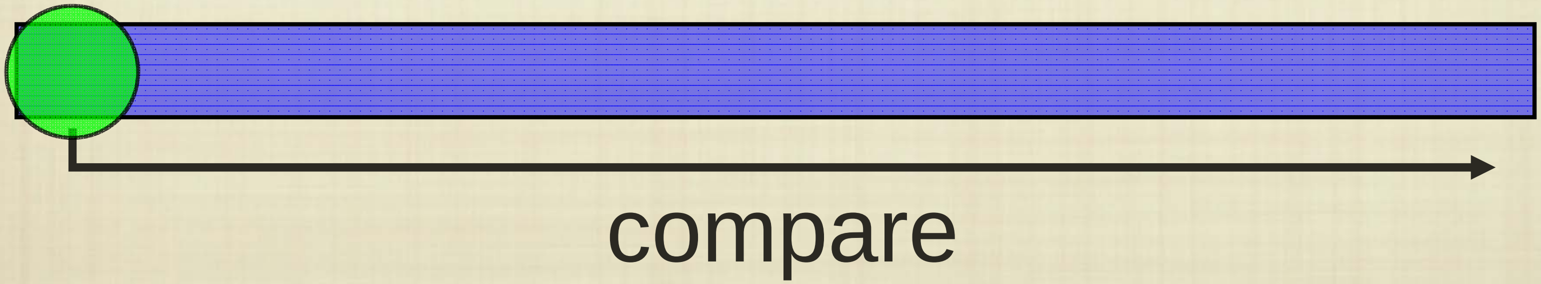
- The theoretical performance is not particularly promising, neither is the practical performance:

Size: n	n^2	Selection Sort: seconds
10	100	0.000505
100	10000	0.002175
1,000	1,000,000	0.178361
10,000	100,000,000	17.010634
100,000	10,000,000,000	2524.767636

Can we do better? What is done in practice? How good is the Python library sort function?

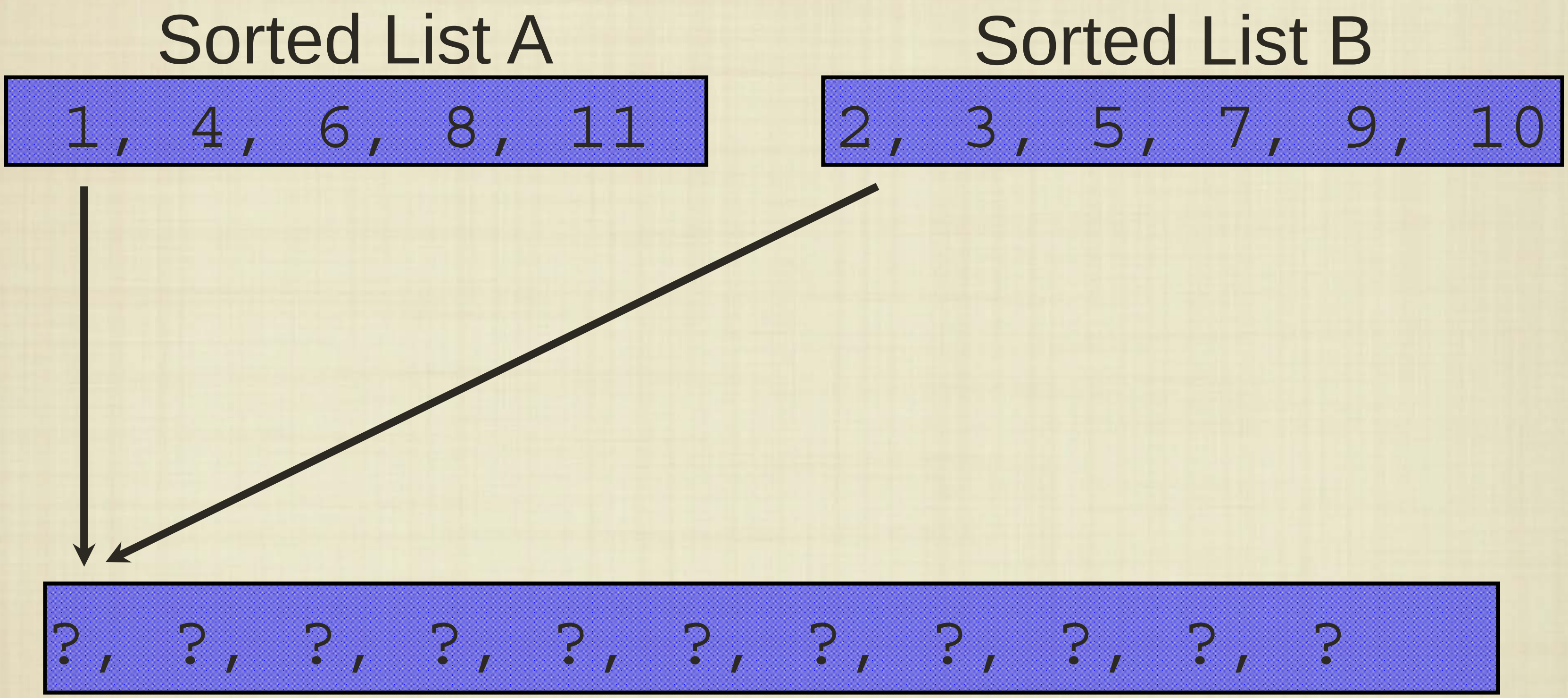
Revisiting Selection Sort

- 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?



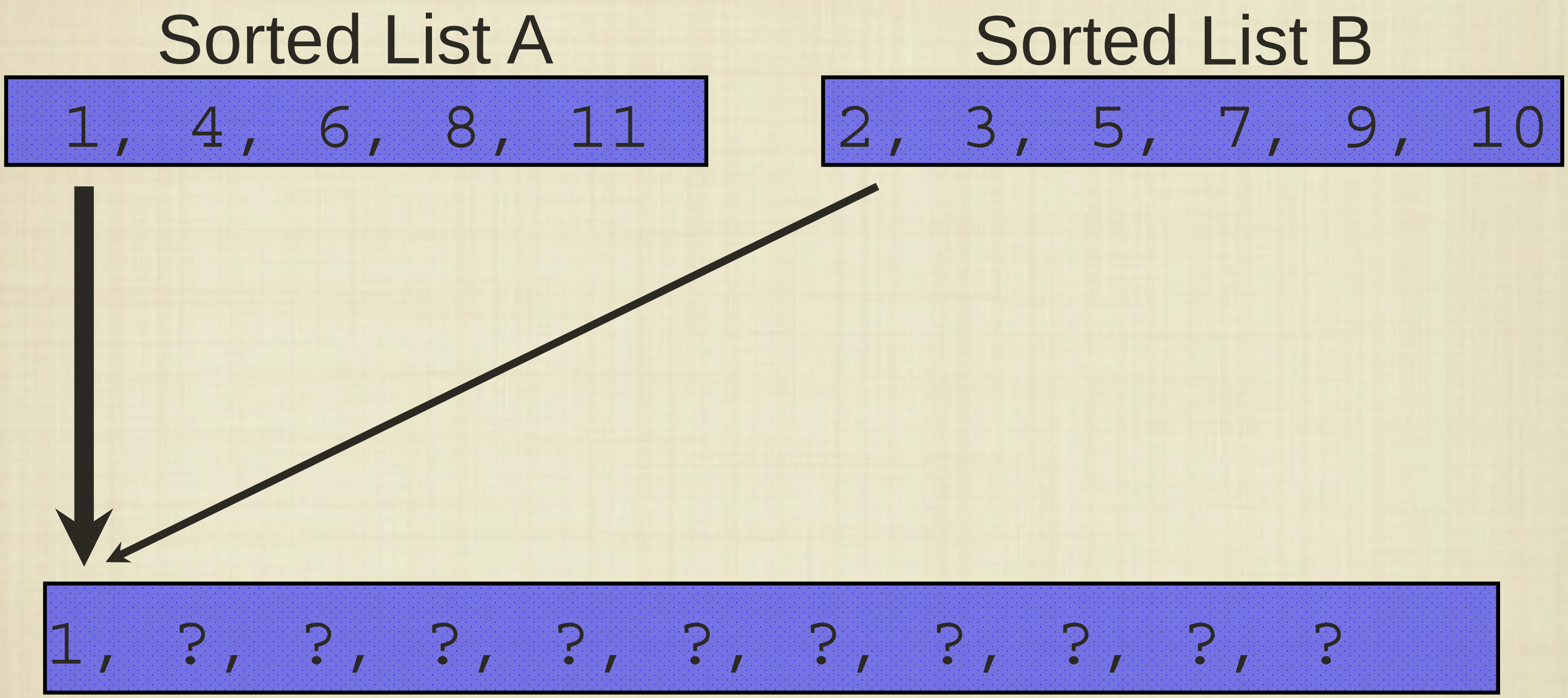
Merging Lists

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



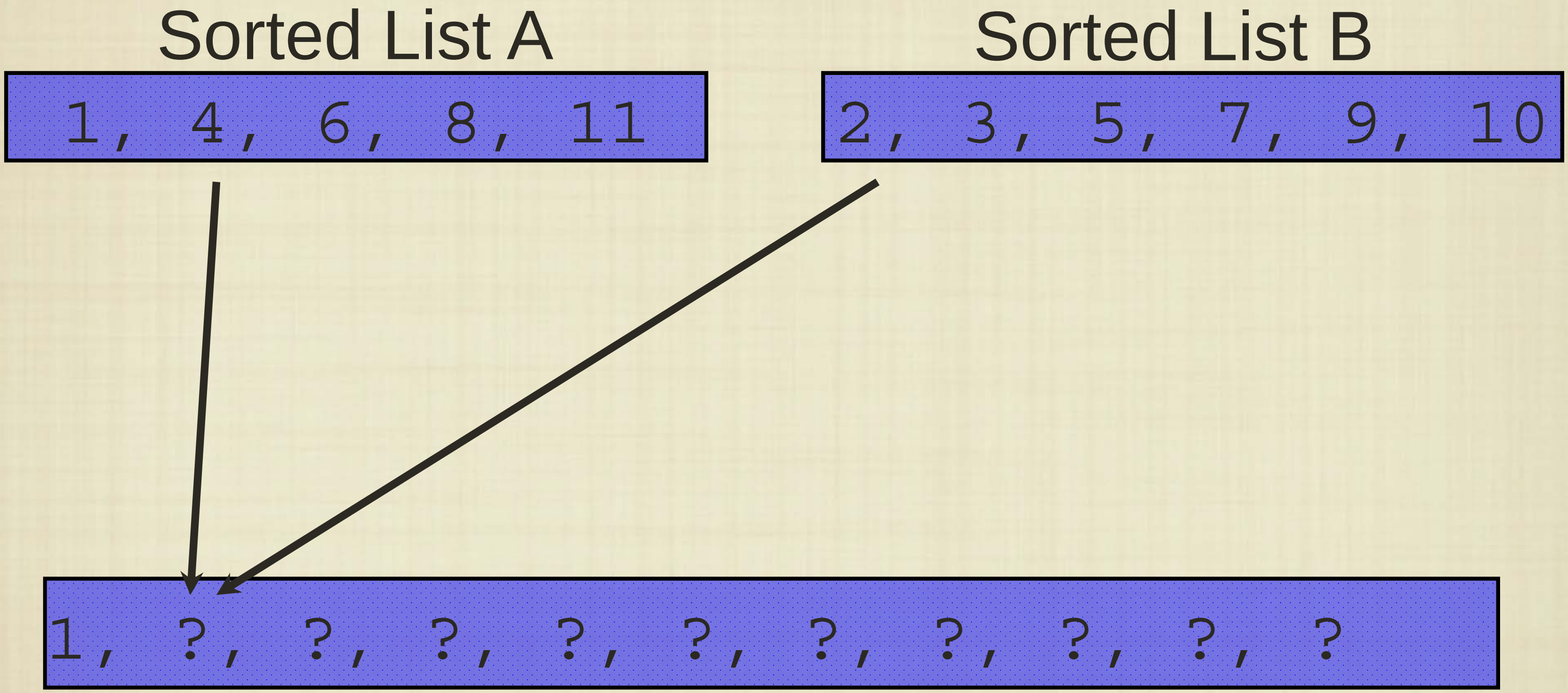
Merging Lists

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



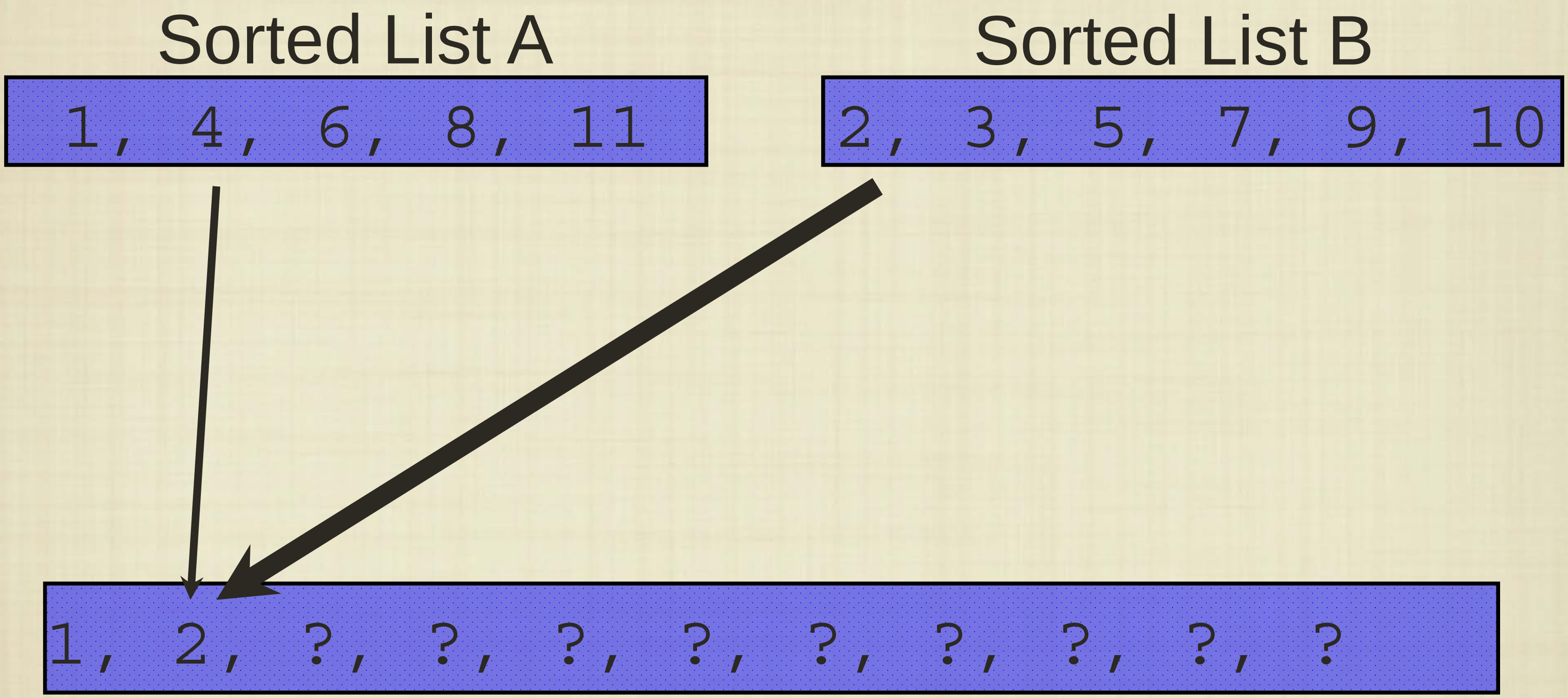
Merging Lists

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



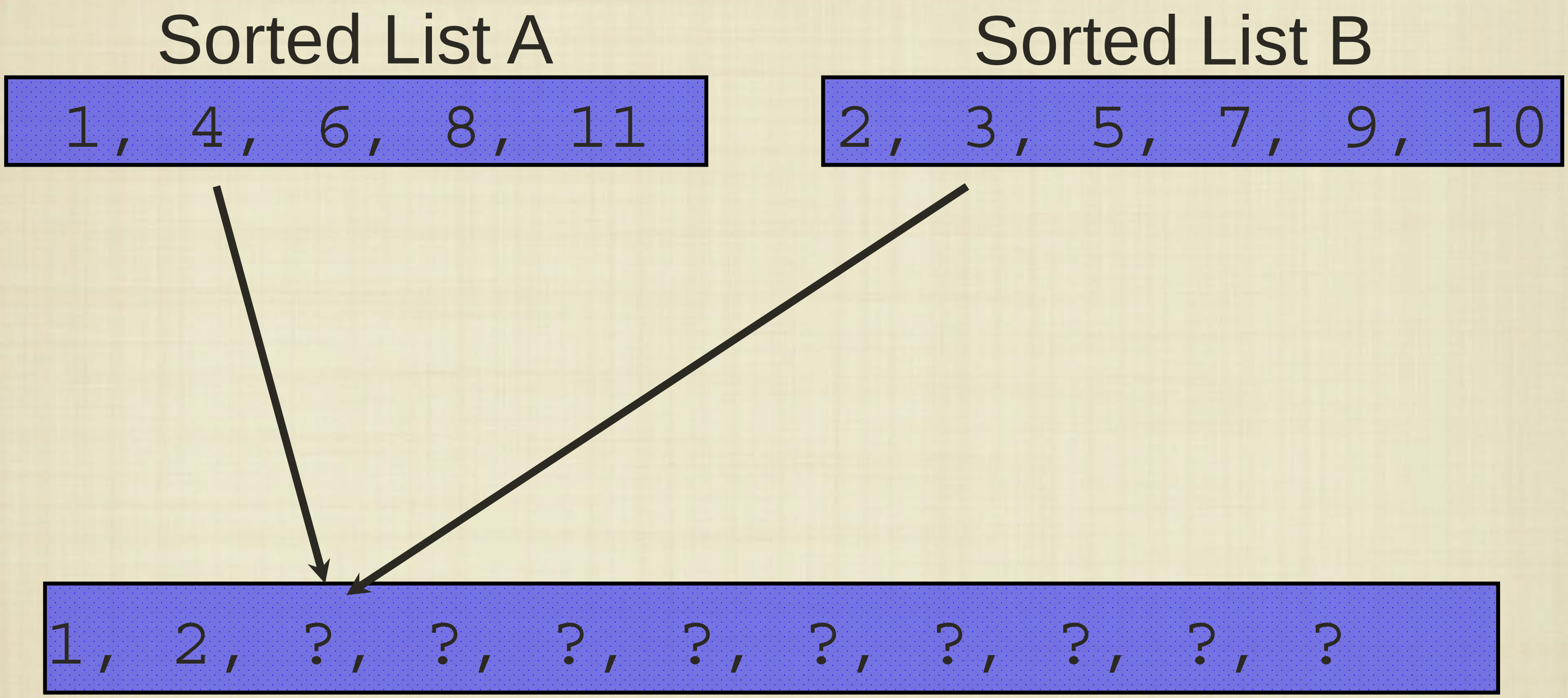
Merging Lists

- 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?

Sorted List A

`1, 4, 6, 8, 11`

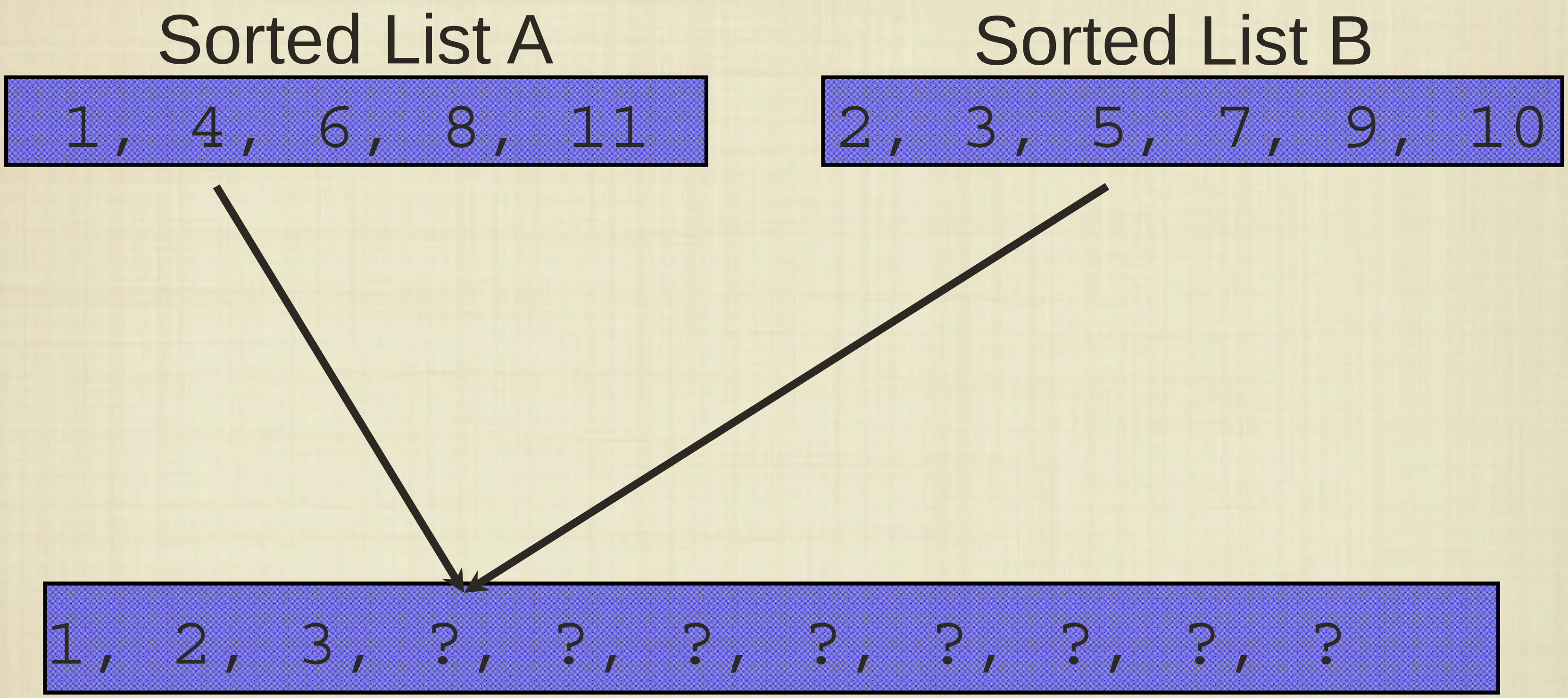
Sorted List B

`2, 3, 5, 7, 9, 10`

`1, 2, 3, ?, ?, ?, ?, ?, ?, ?, ?`

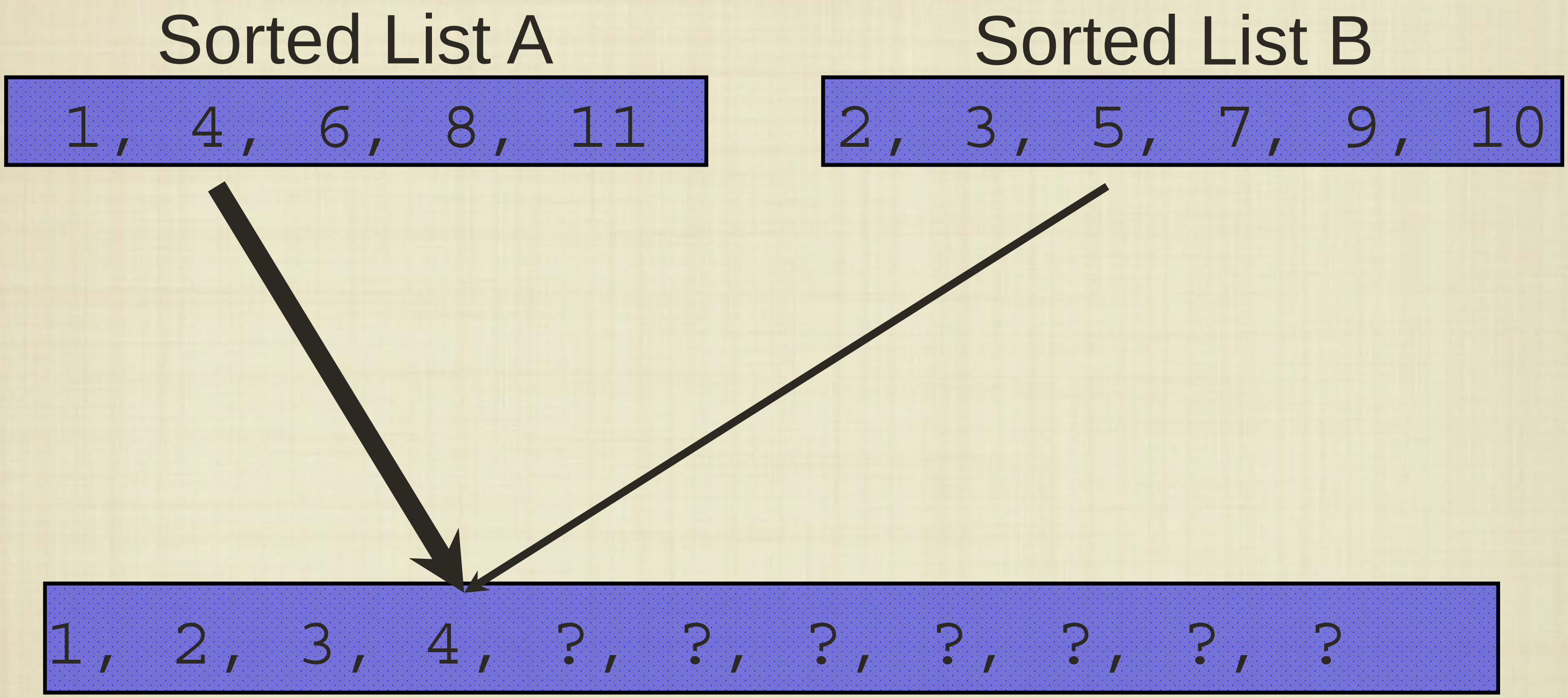
Merging Lists

- 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, 4, 6, 8, 11`

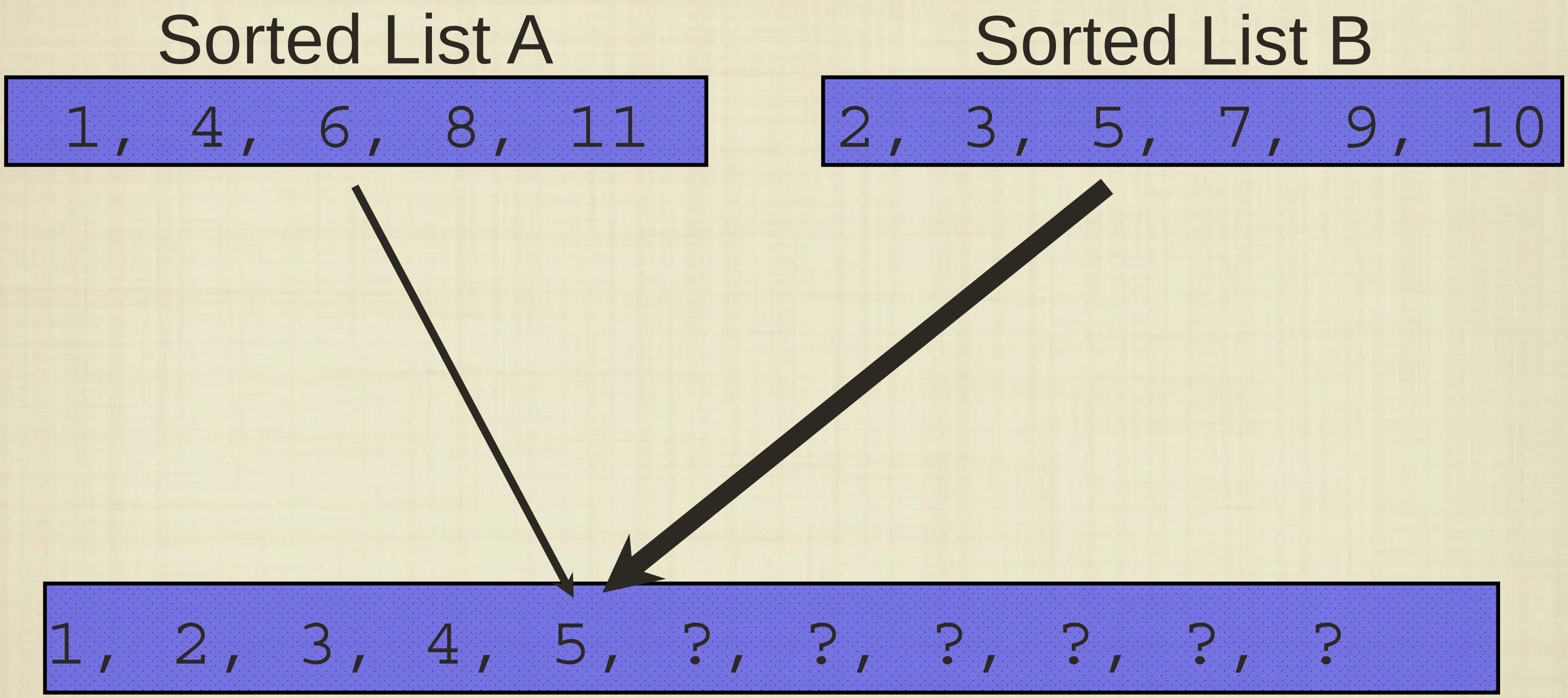
Sorted List B

`2, 3, 5, 7, 9, 10`

`1, 2, 3, 4, ?, ?, ?, ?, ?, ?, ?`

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, 4, 6, 8, 11`

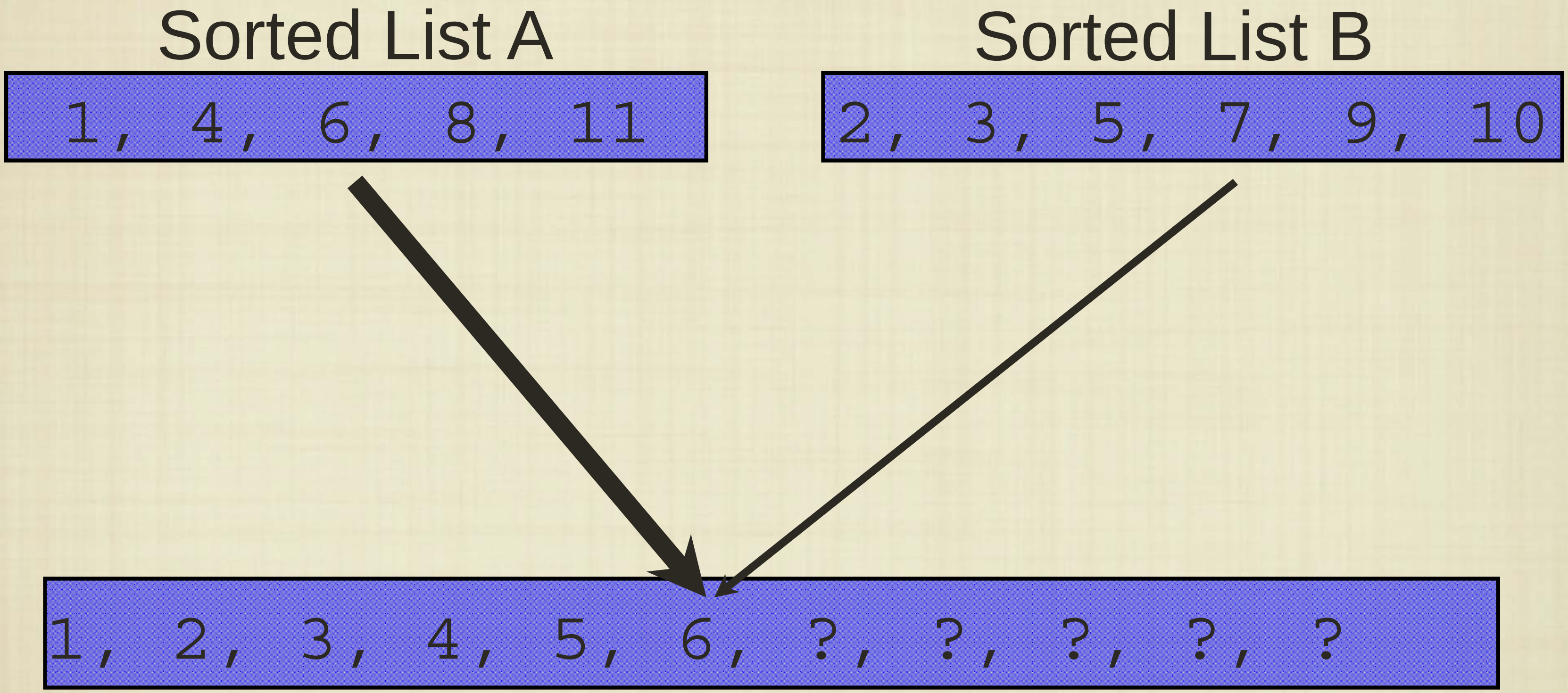
Sorted List B

`2, 3, 5, 7, 9, 10`

`1, 2, 3, 4, 5, ?, ?, ?, ?, ?, ?`

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, 4, 6, 8, 11`

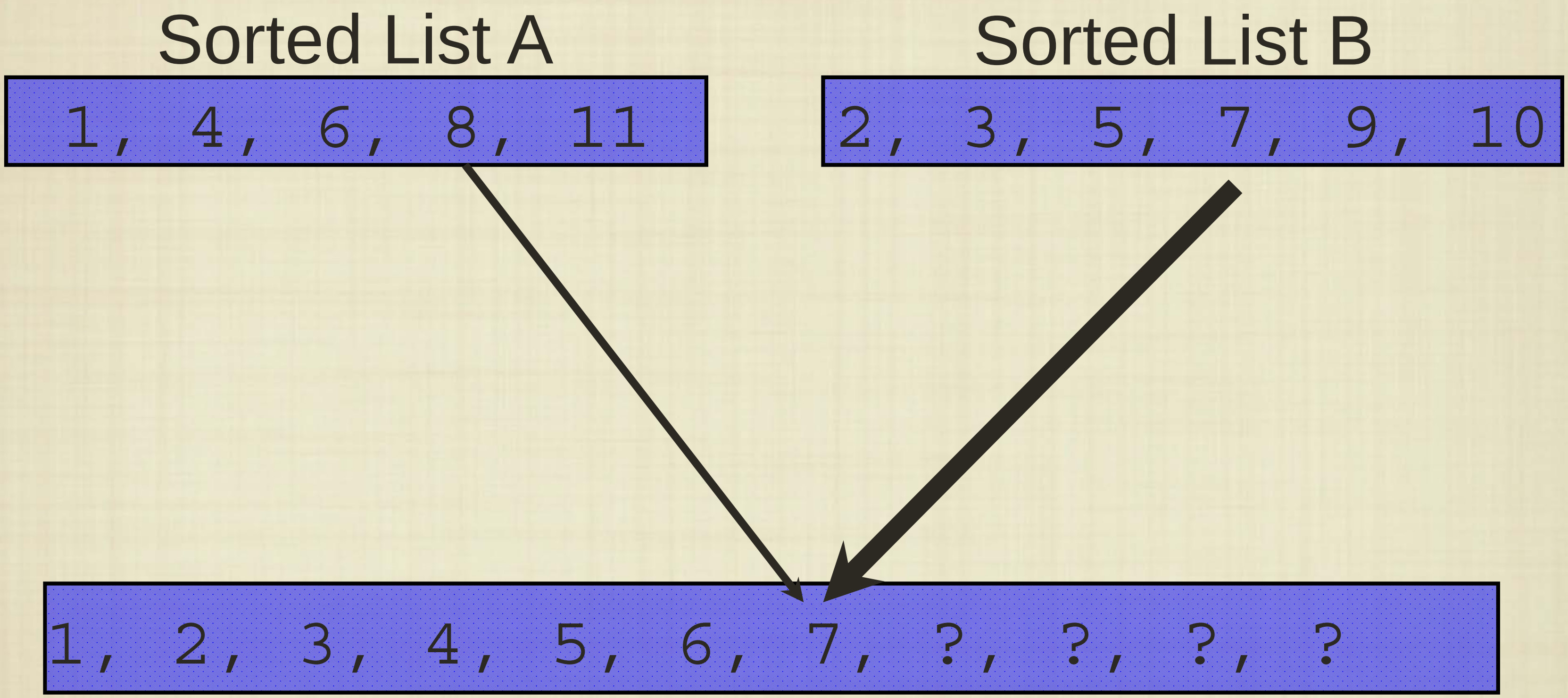
Sorted List B

`2, 3, 5, 7, 9, 10`

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

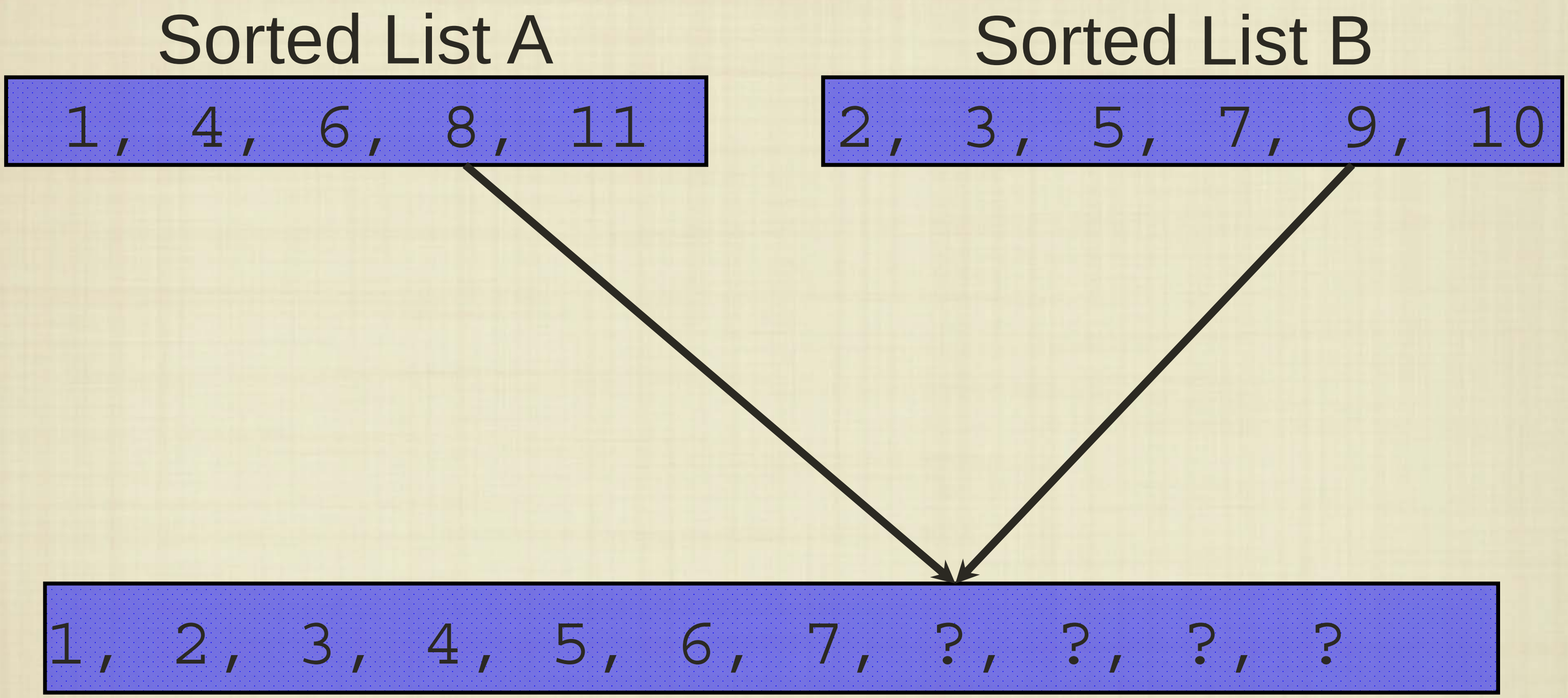
Merging Lists

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



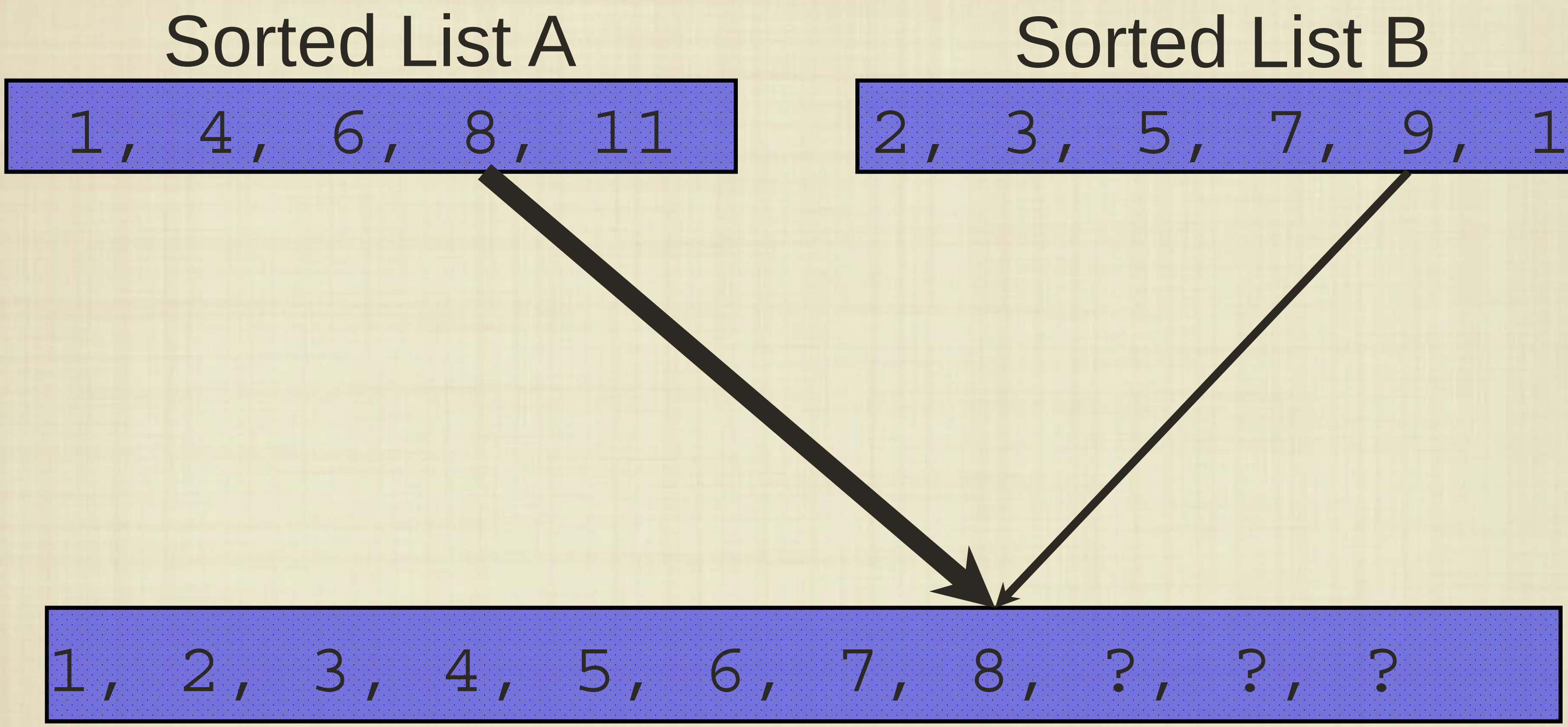
Merging Lists

- 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, 4, 6, 8, 11`

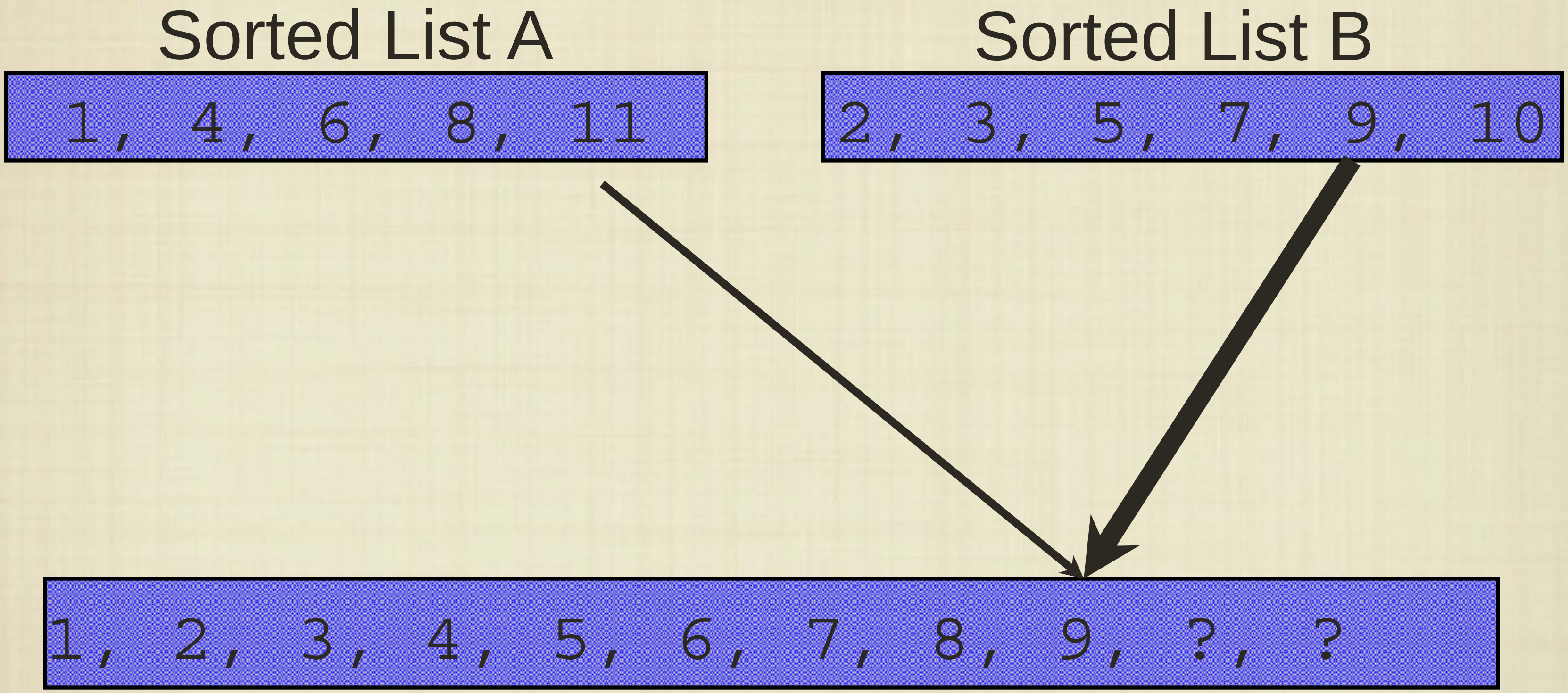
Sorted List B

`2, 3, 5, 7, 9, 10`

`1, 2, 3, 4, 5, 6, 7, 8, ?, ?, ?`

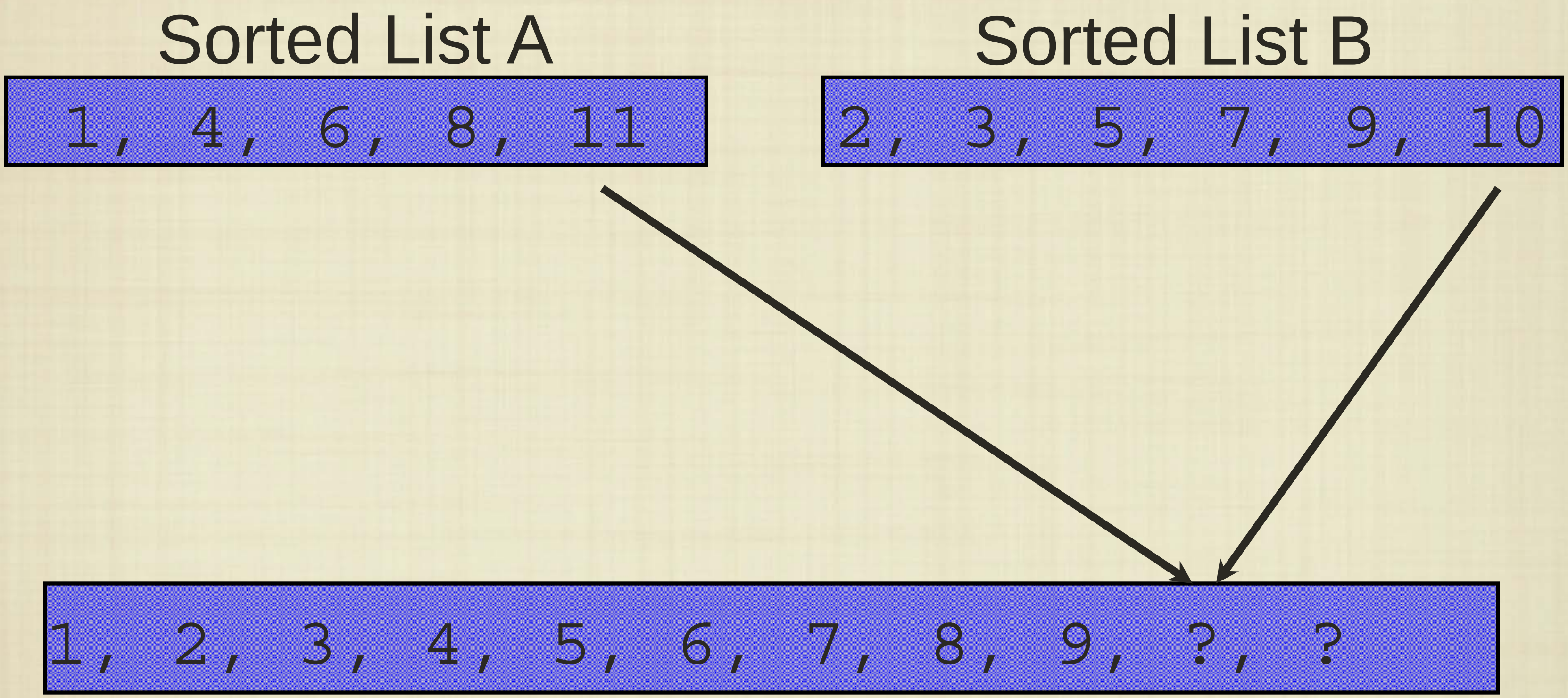
Merging Lists

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



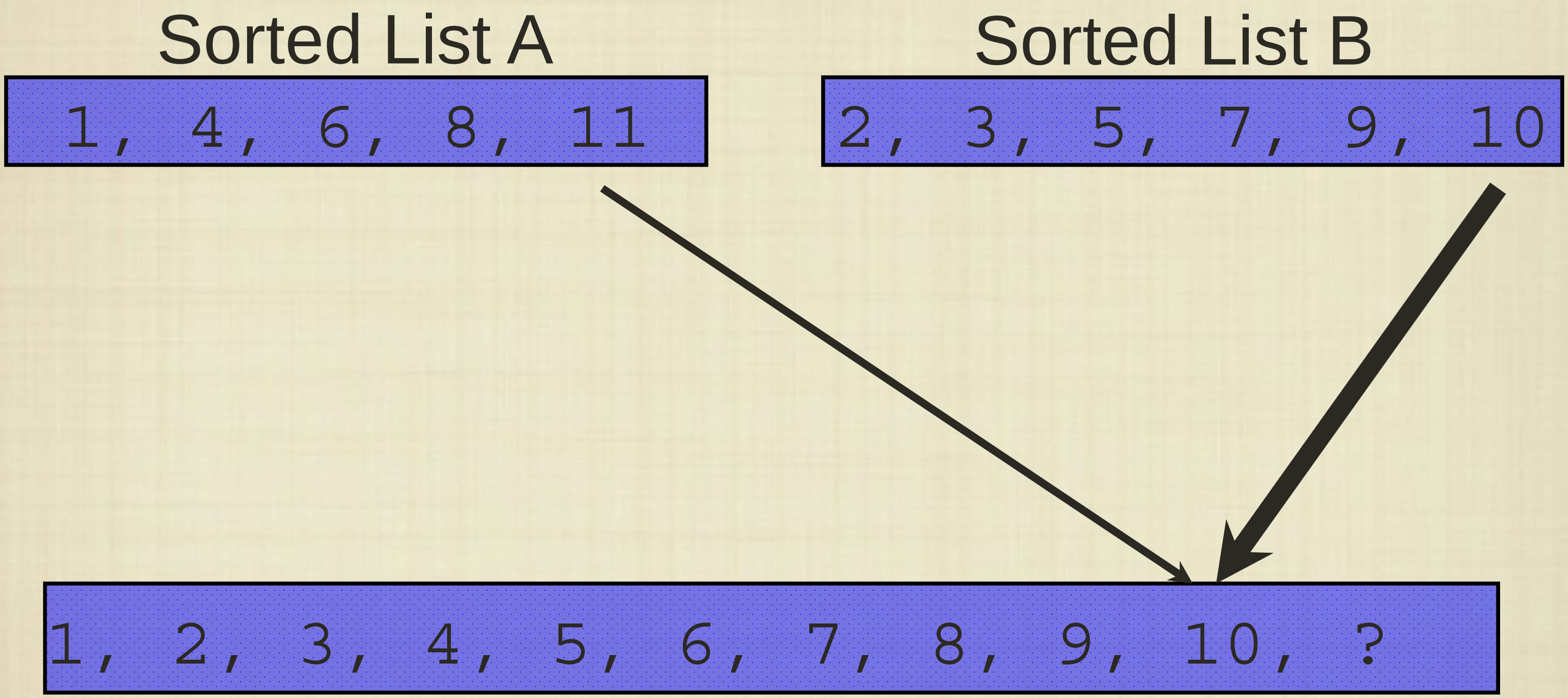
Merging Lists

- 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, 4, 6, 8, 11`

Sorted List B

`2, 3, 5, 7, 9, 10`

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

Merging Lists

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

Sorted List A

`1, 4, 6, 8, 11`

Sorted List B

`2, 3, 5, 7, 9, 10`

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

Merging Lists

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

Sorted List A

`1, 4, 6, 8, 11`

Sorted List B

`2, 3, 5, 7, 9, 10`

`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? 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:

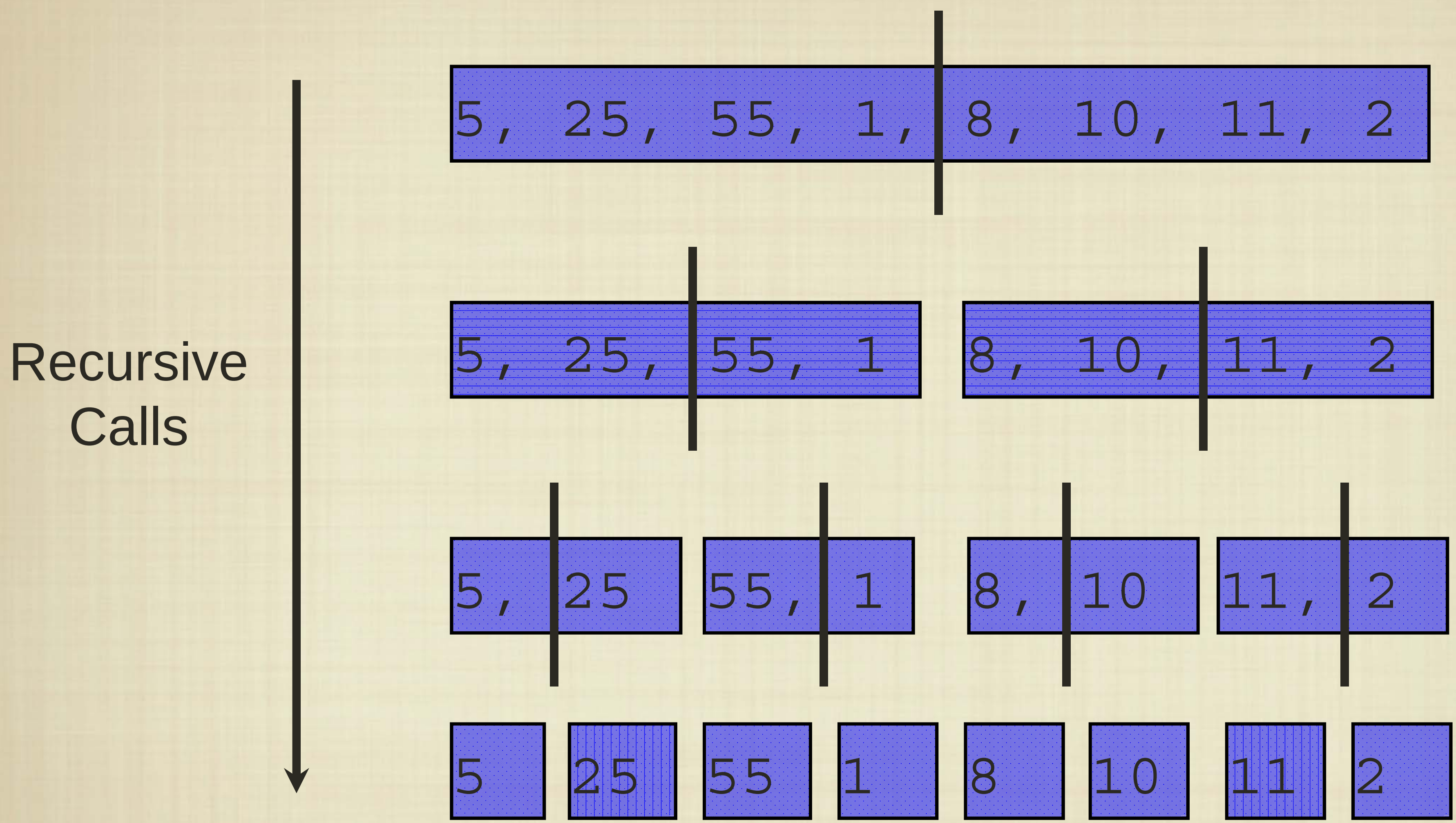
- 1. Split the given list into two equal parts.
- 2. Recursively sort each half.
- 3. Merge the sorted halves and return the result.

Merge Sort

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

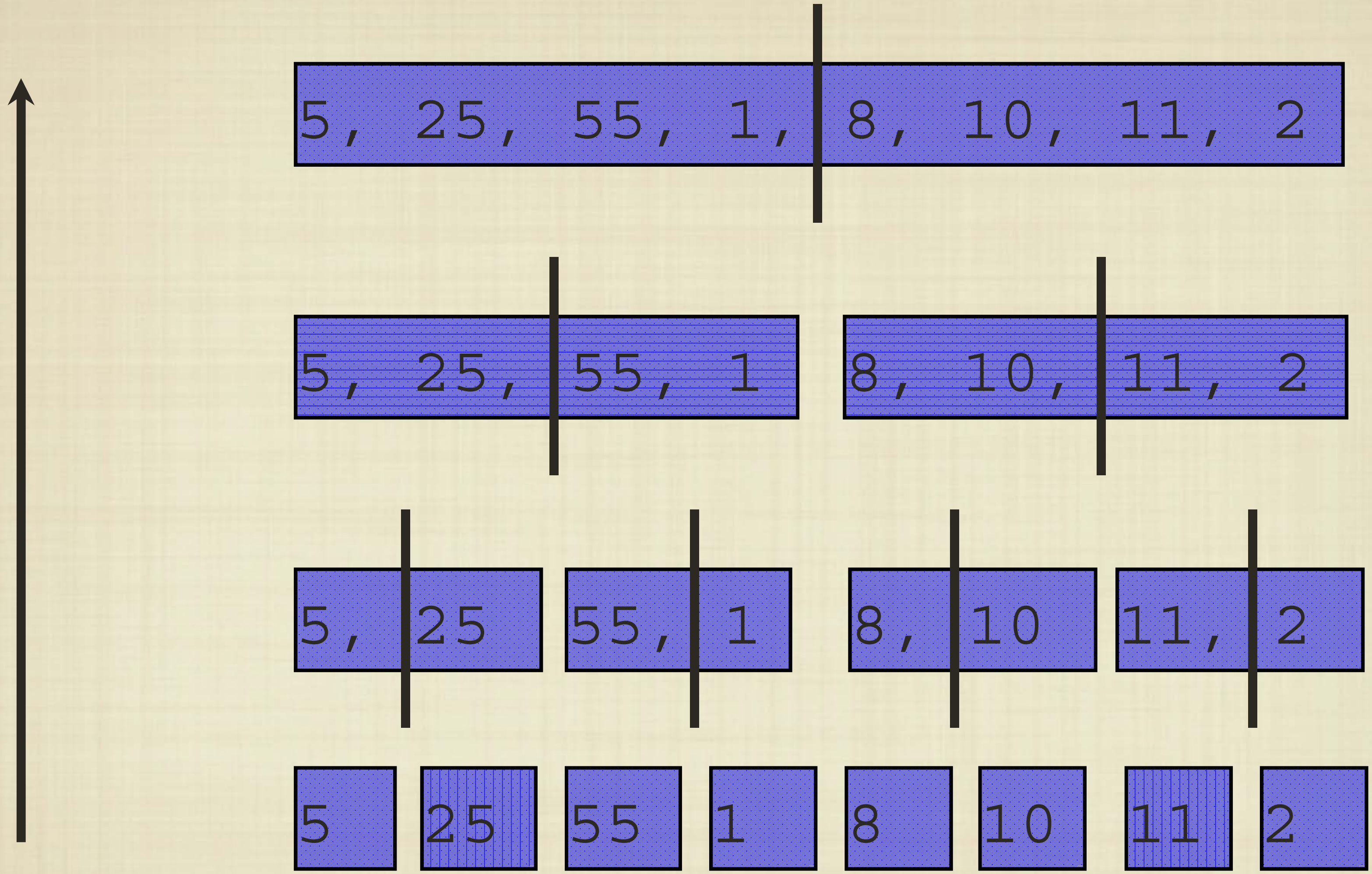
```
def merge_sort (L):
    n = len(L)
    #base case:
    if n<=1:
        return L
    #recursive case: Recursively sort each half
    A = merge_sort(L[:n/2])   # left half, L[0..n/2-1]
    B = merge_sort(L[n/2:])   # right half, L[n/2..n-1]
    # merge sorted halves:
    return merge(A,B)
```

Merge Sort



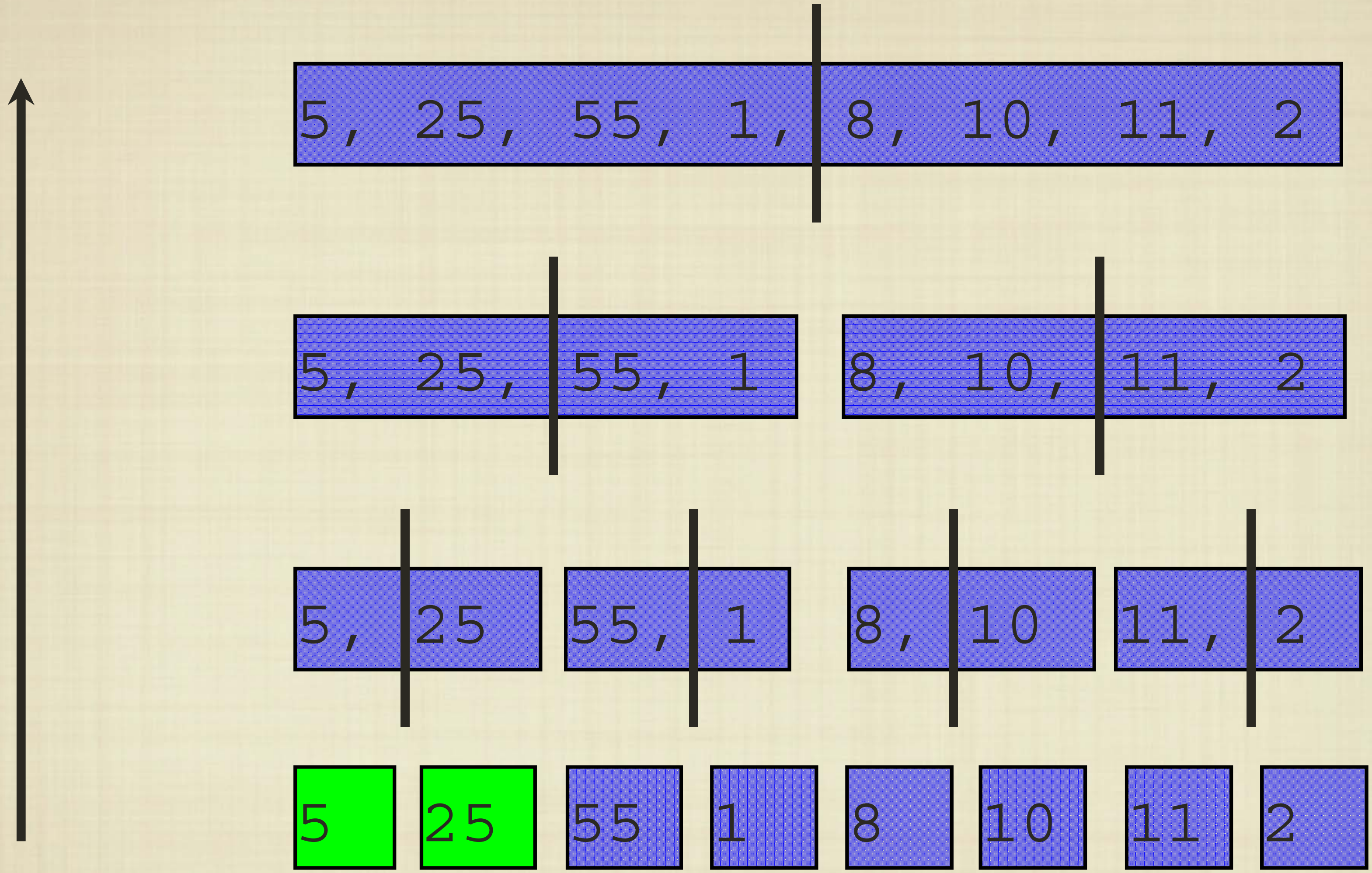
Actually, not a lot is happening in the recursive calls. So where is the sorting happening?

Merge Sort



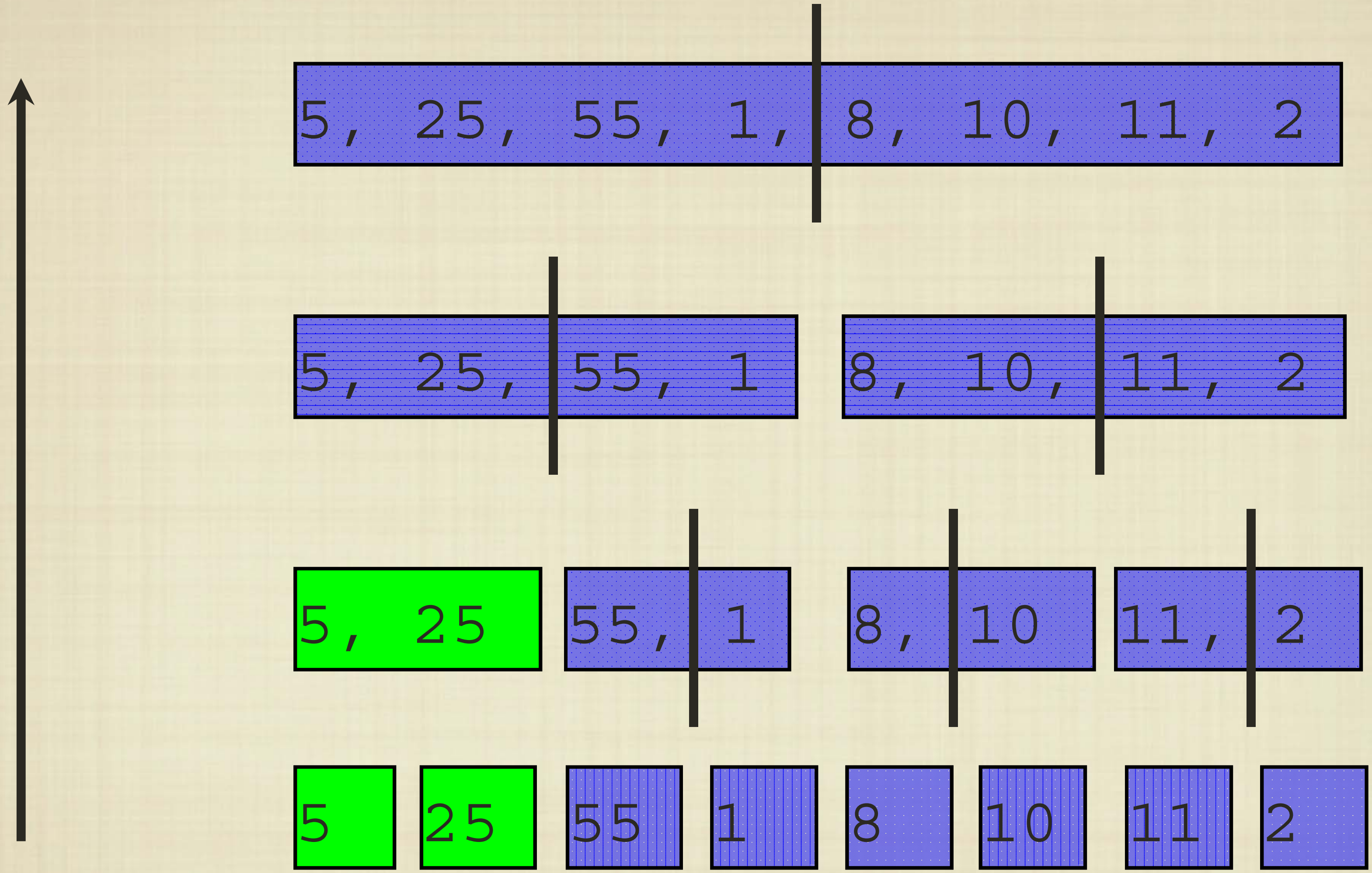
The merge step is actually doing all of the work!

Merge Sort



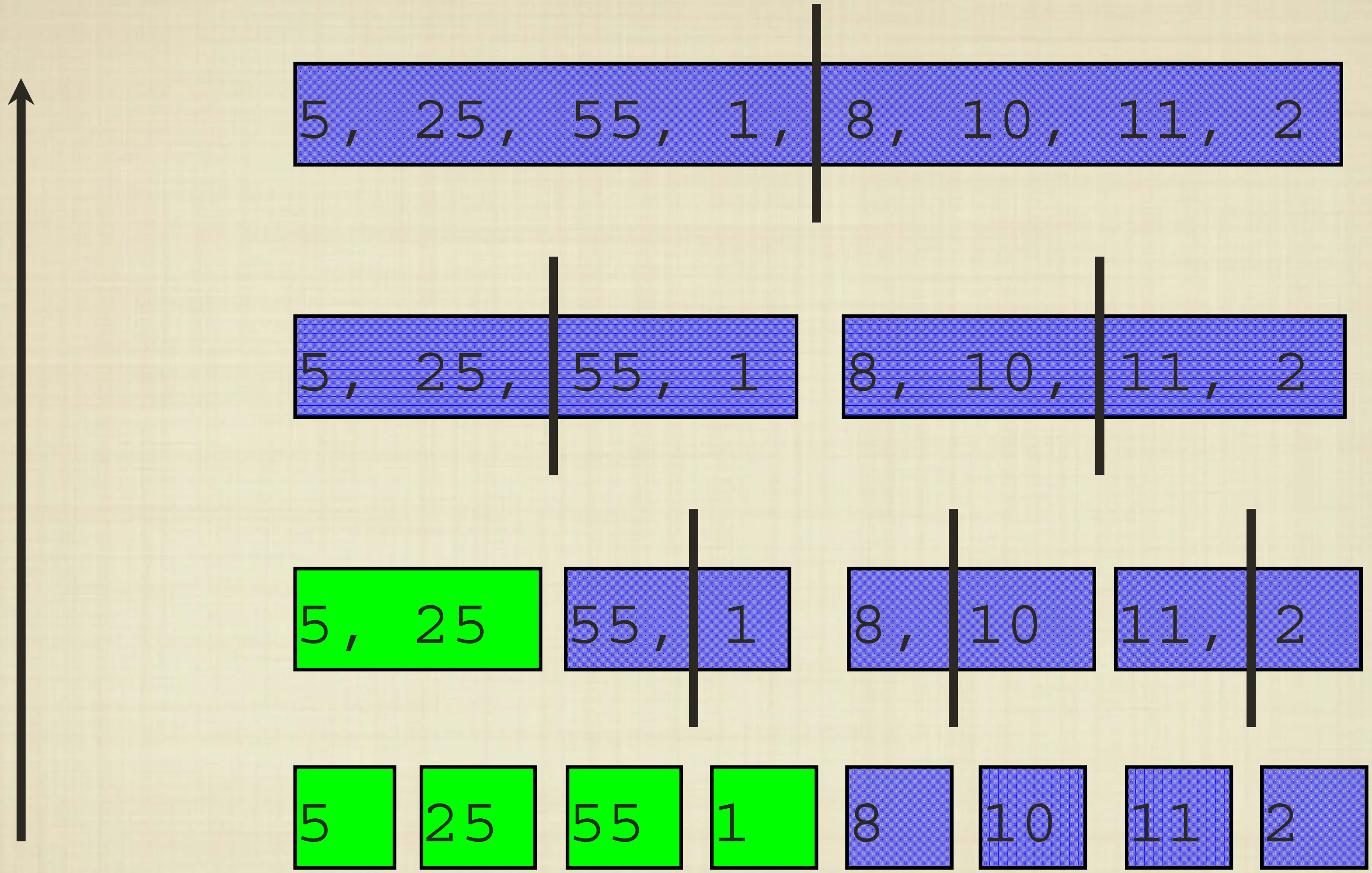
The merge step is actually doing all of the work!

Merge Sort



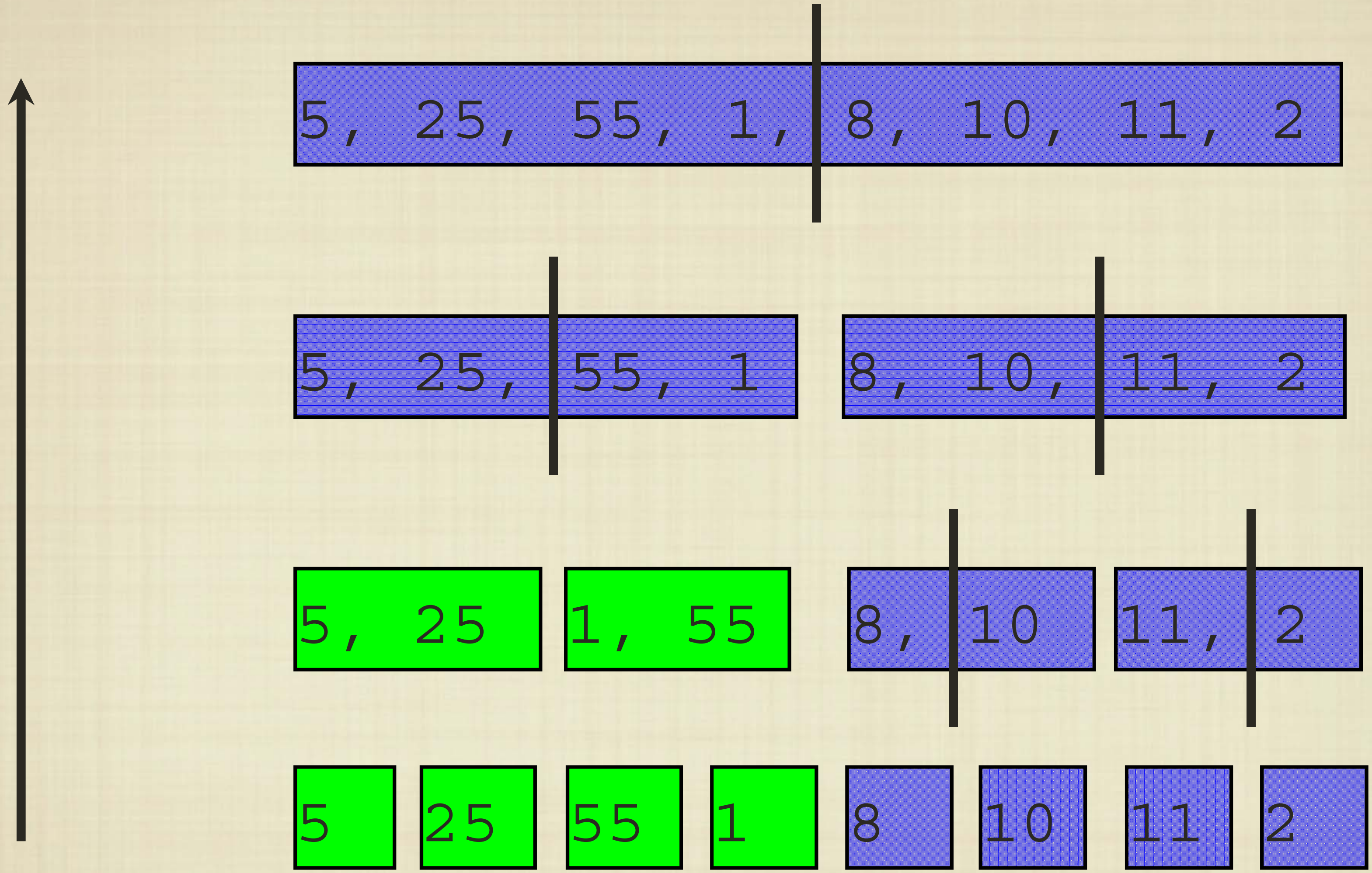
The merge step is actually doing all of the work!

Merge Sort



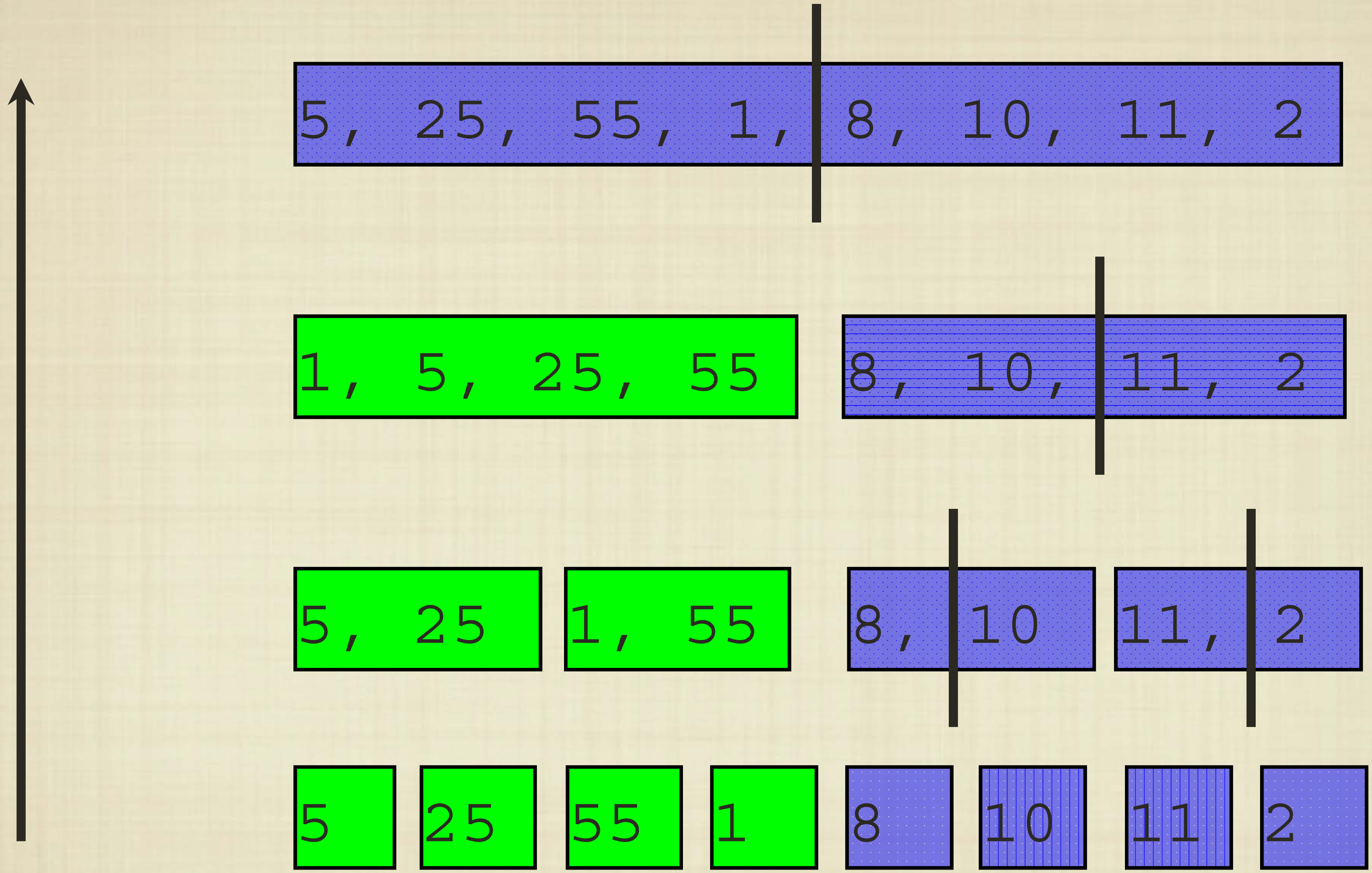
The merge step is actually doing all of the work!

Merge Sort



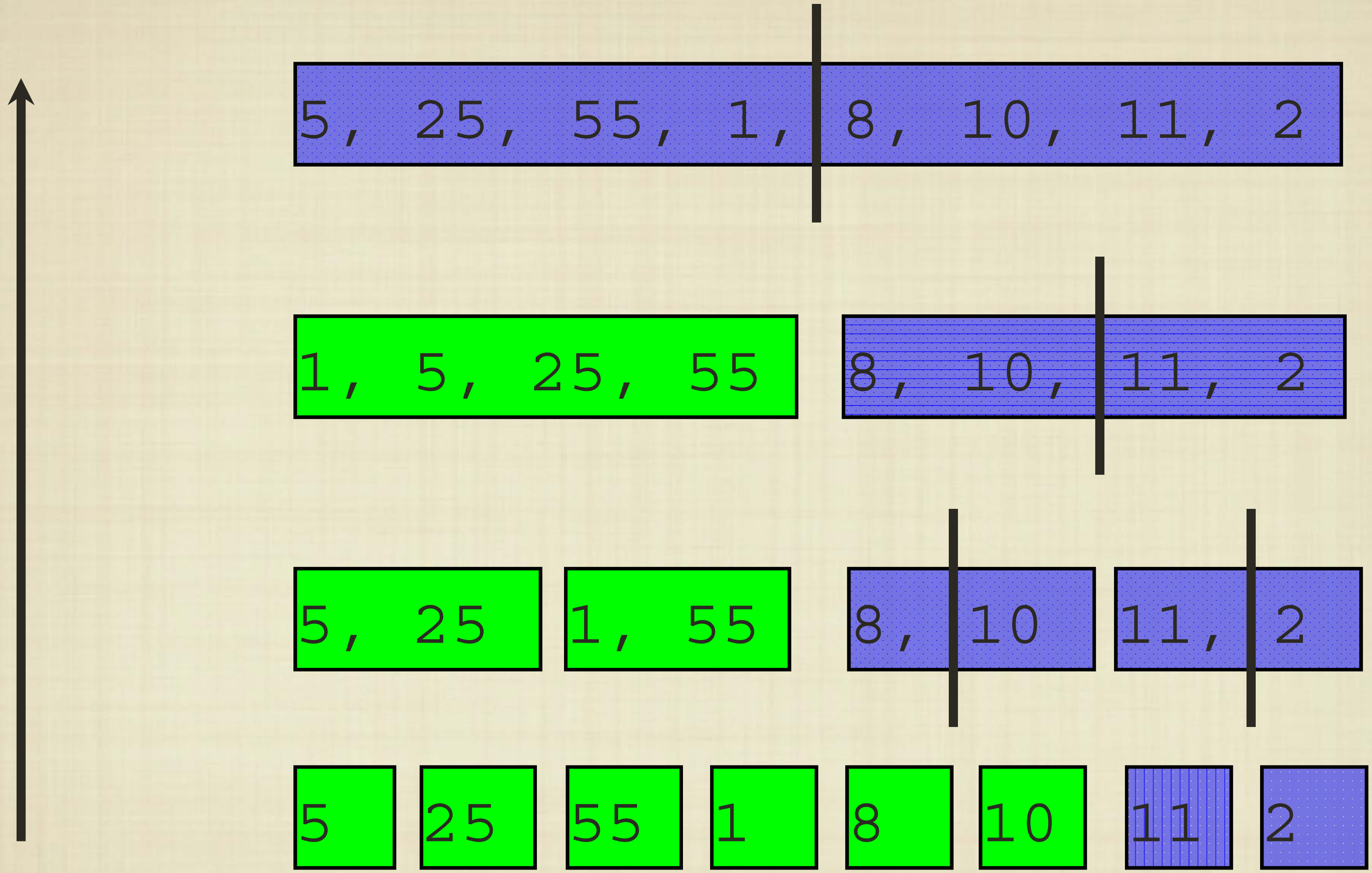
The merge step is actually doing all of the work!

Merge Sort



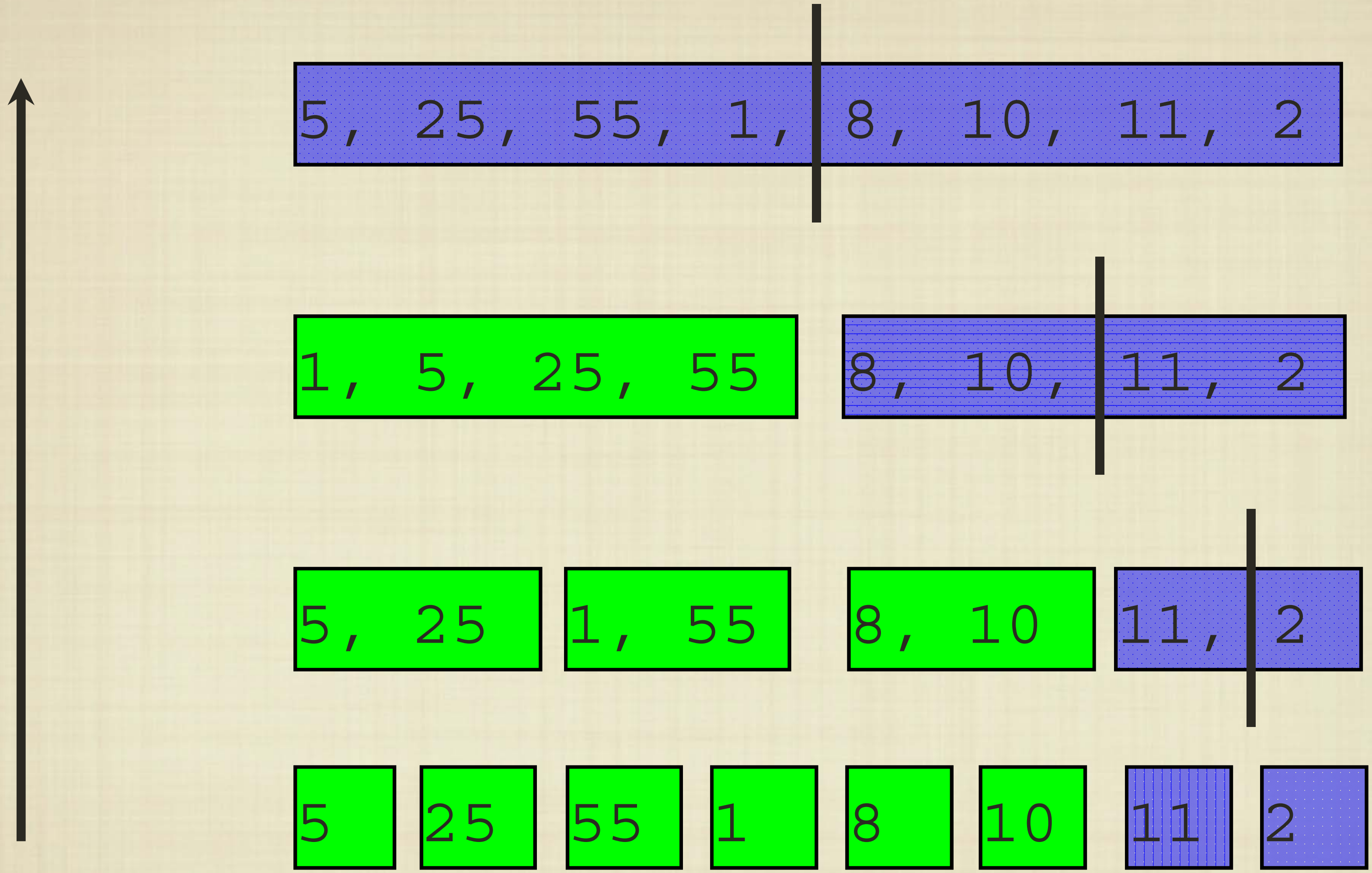
The merge step is actually doing all of the work!

Merge Sort



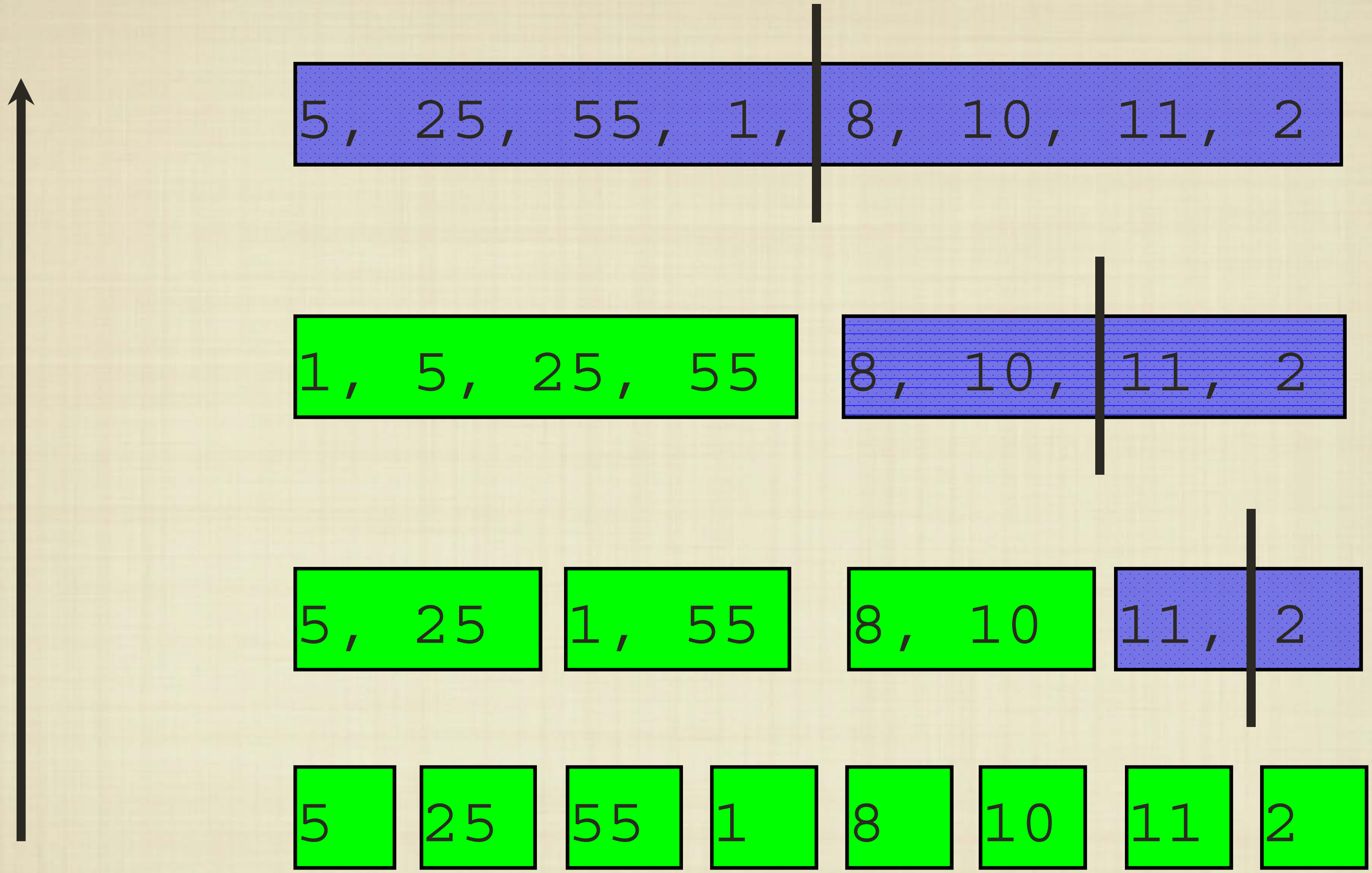
The merge step is actually doing all of the work!

Merge Sort



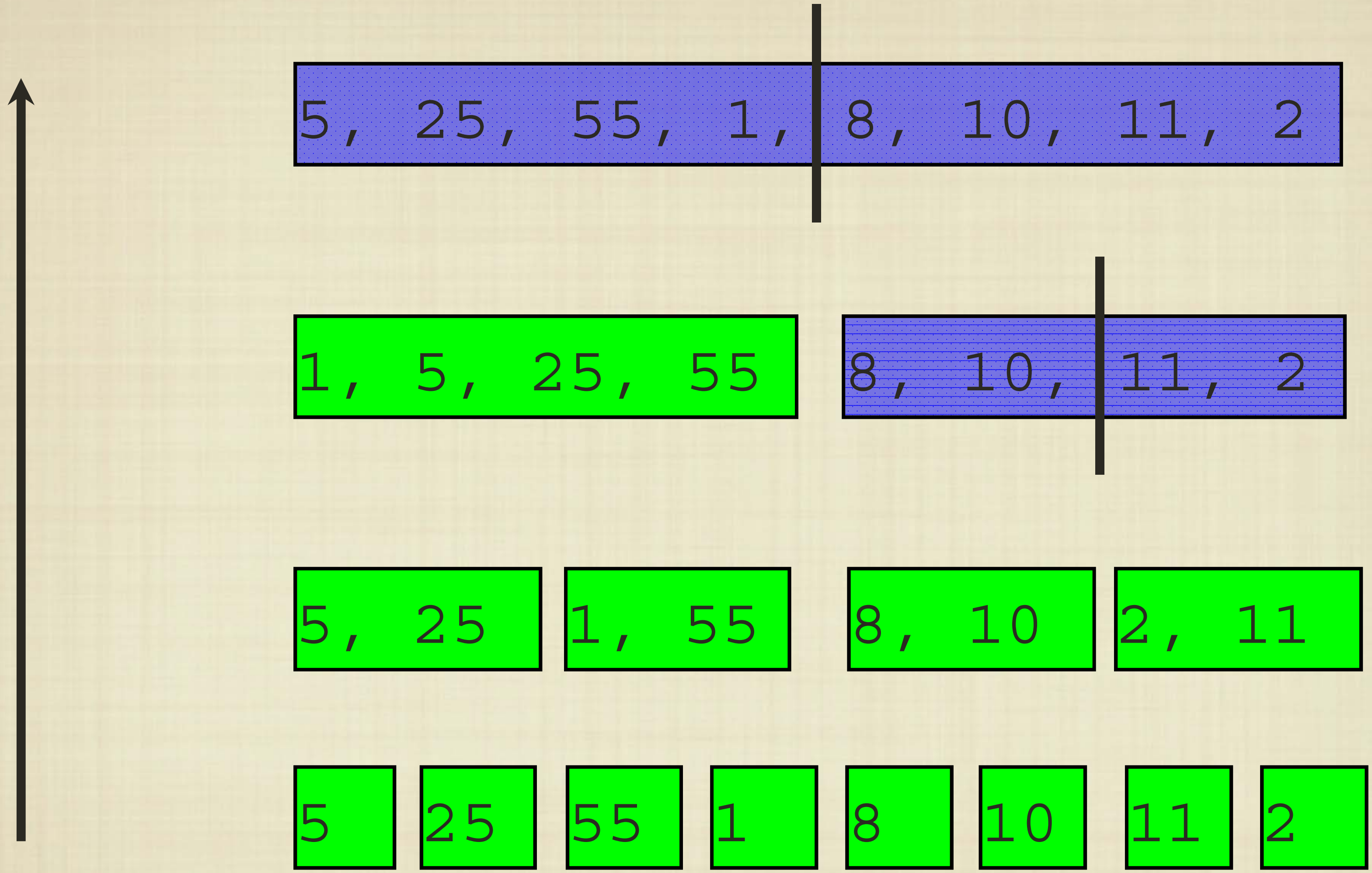
The merge step is actually doing all of the work!

Merge Sort



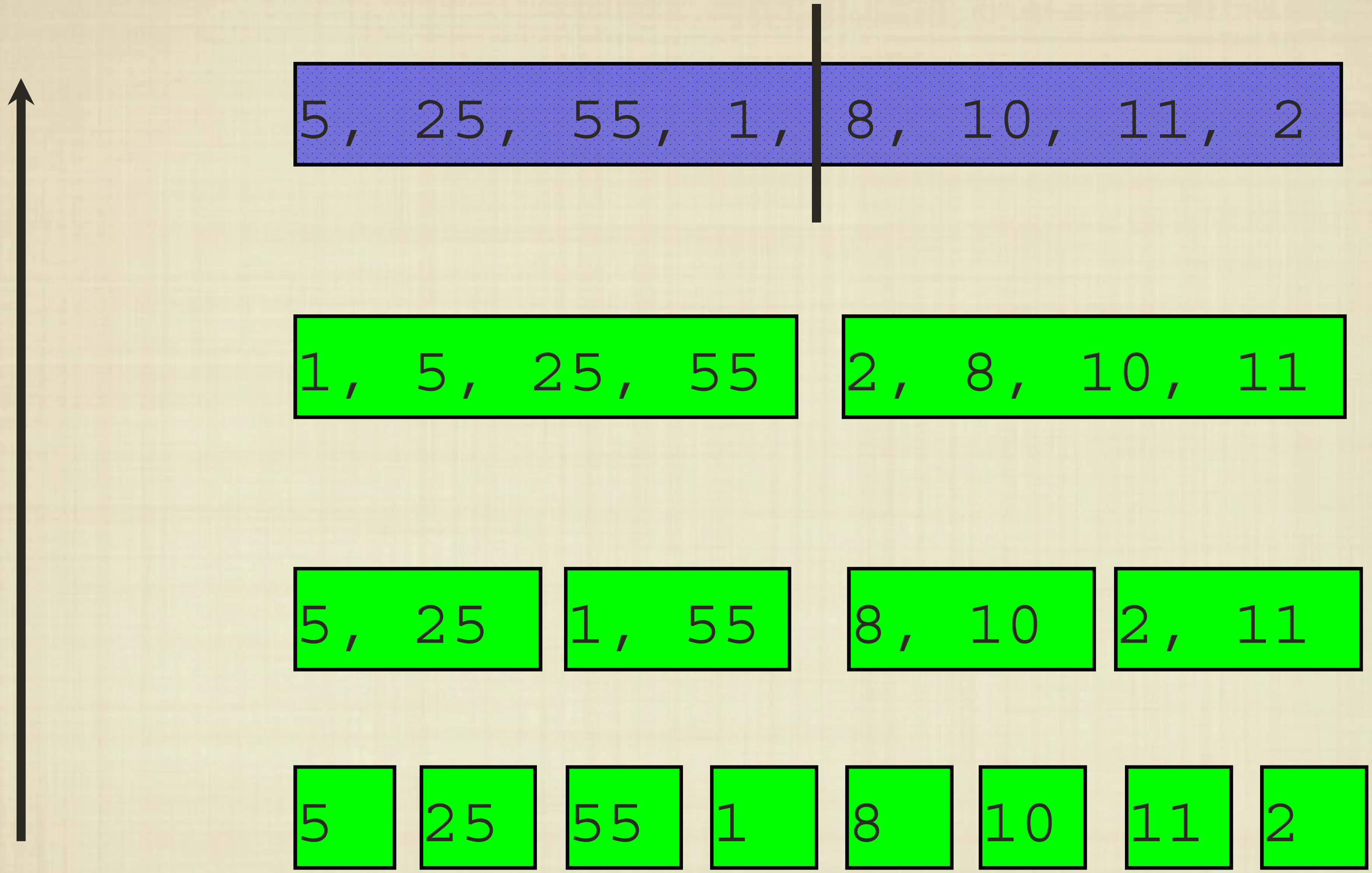
The merge step is actually doing all of the work!

Merge Sort



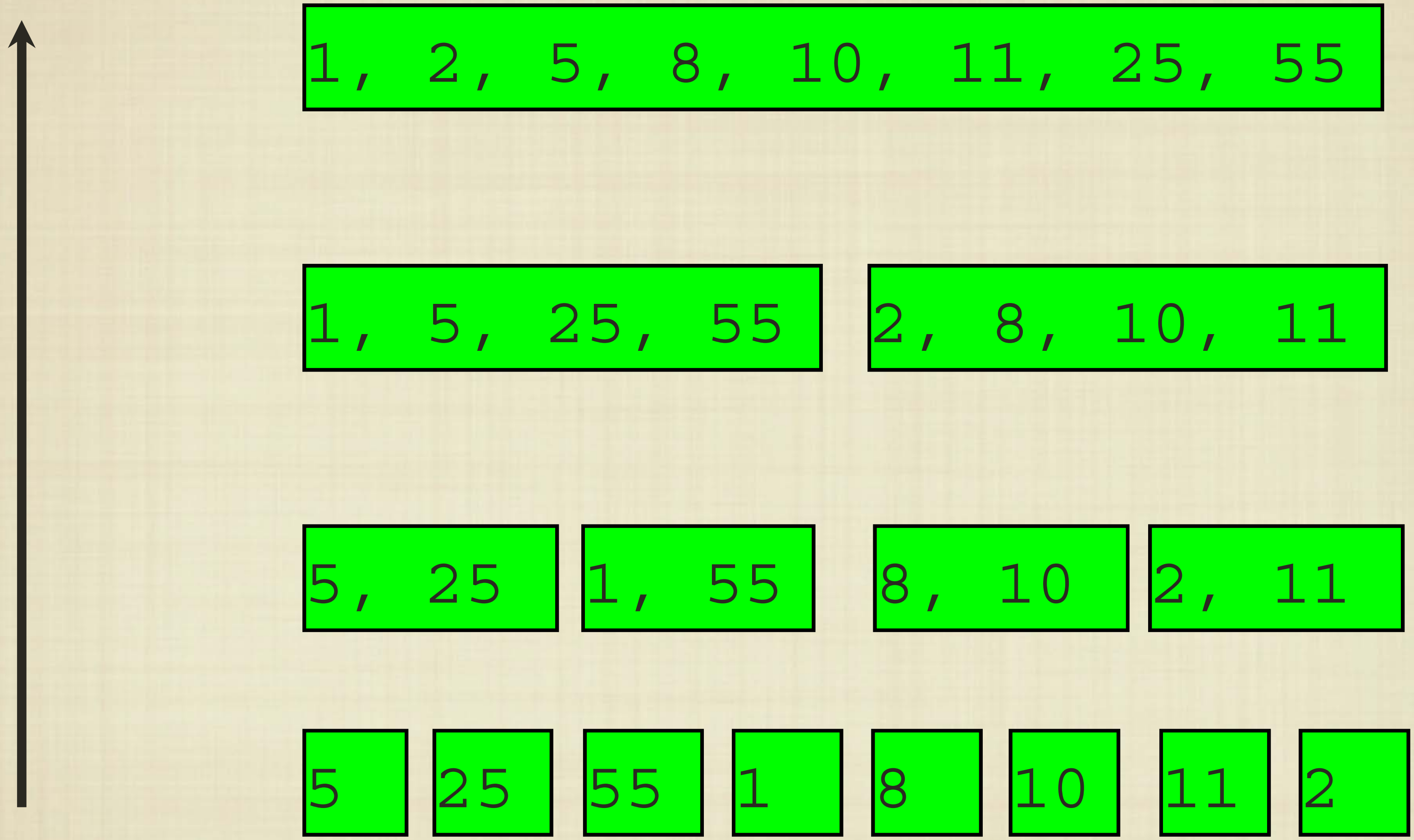
The merge step is actually doing all of the work!

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 Runtime Analysis

Runtime	
$T(n)$	`def merge_sort (L):`
	`    n = len(L)`
c	`    #base case:`
	`    if n<=1:`
	`        return L`
	`    #recursive case: Recursively sort each half`
$T(n/2)$	`    A = merge_sort(L[:n/2])   # left half, L[0..n/2-1]`
$T(n/2)$	`    B = merge_sort(L[n/2:])   # right half, L[n/2..n-1]`
	`    # merge sorted halves:`
dn	`    return merge(A,B)`

Runtime Recurrence for Merge Sort

$$T(n) = \begin{cases} c \text{ if } n = 1; \\ 2T(n/2) + dn \text{ if } n > 1. \end{cases}$$

- But what does $T(n)$ solve to? I.e., is it $O(n)$ or $O(n^2)$ or $O(n^3)$ or …?

Recursion Tree

Solve $T(n) = 2T(n/2) + dn$, where $d > 0$ is constant.

$$T(n)$$

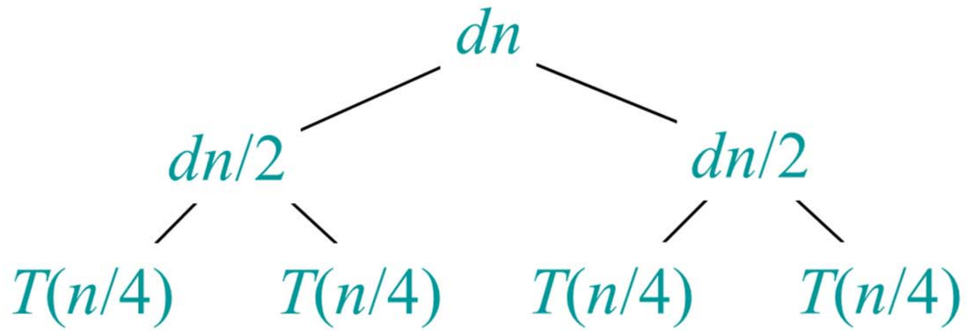
Recursion Tree

Solve $T(n) = 2T(n/2) + dn$, where $d > 0$ is constant.

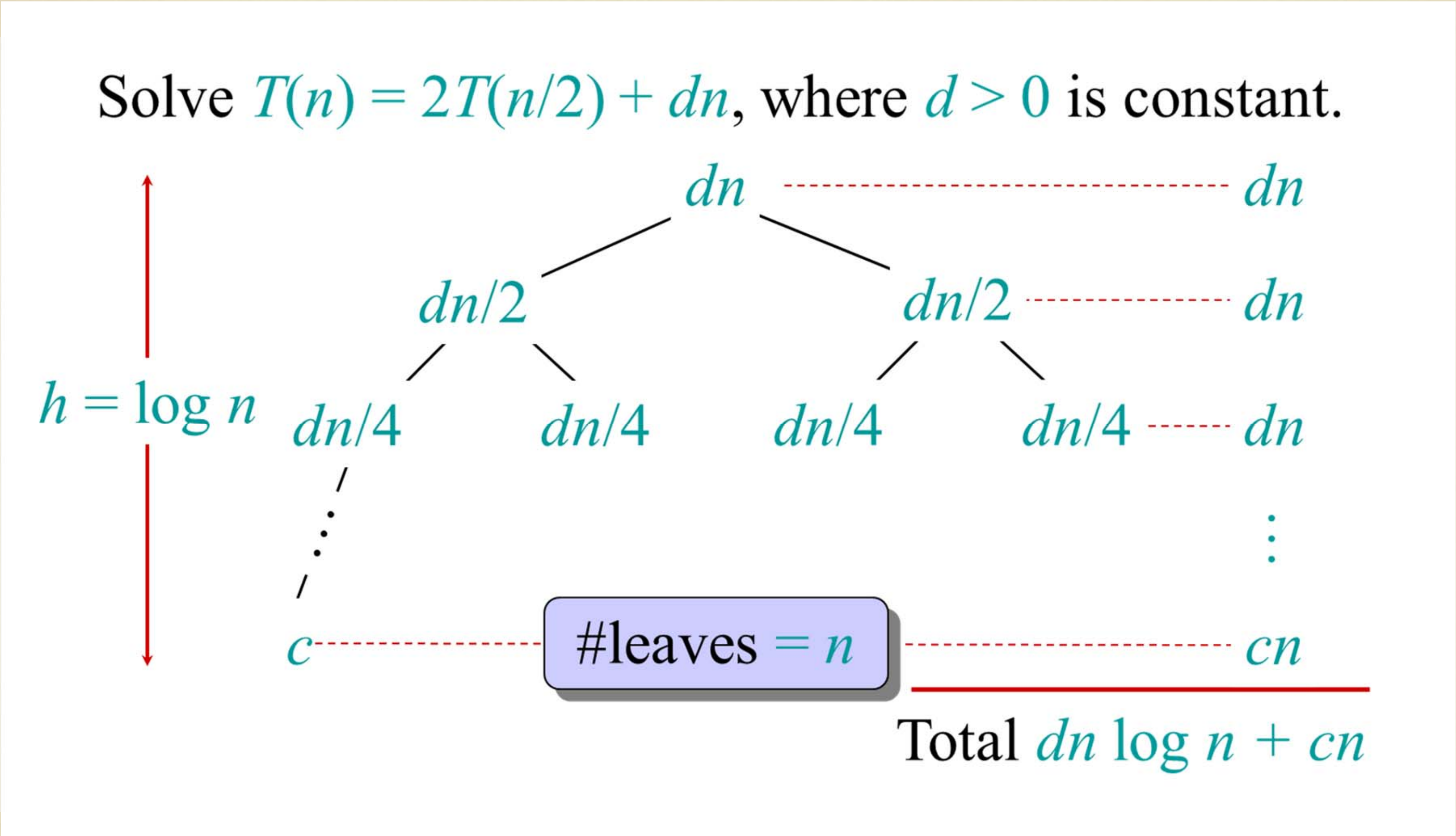
$$
\begin{array}{ccc}
 & dn & \\
\diagup & & \diagdown \\
T(n/2) & & T(n/2)
\end{array}
$$

Recursion Tree

Solve $T(n) = 2T(n/2) + dn$, where $d > 0$ is constant.



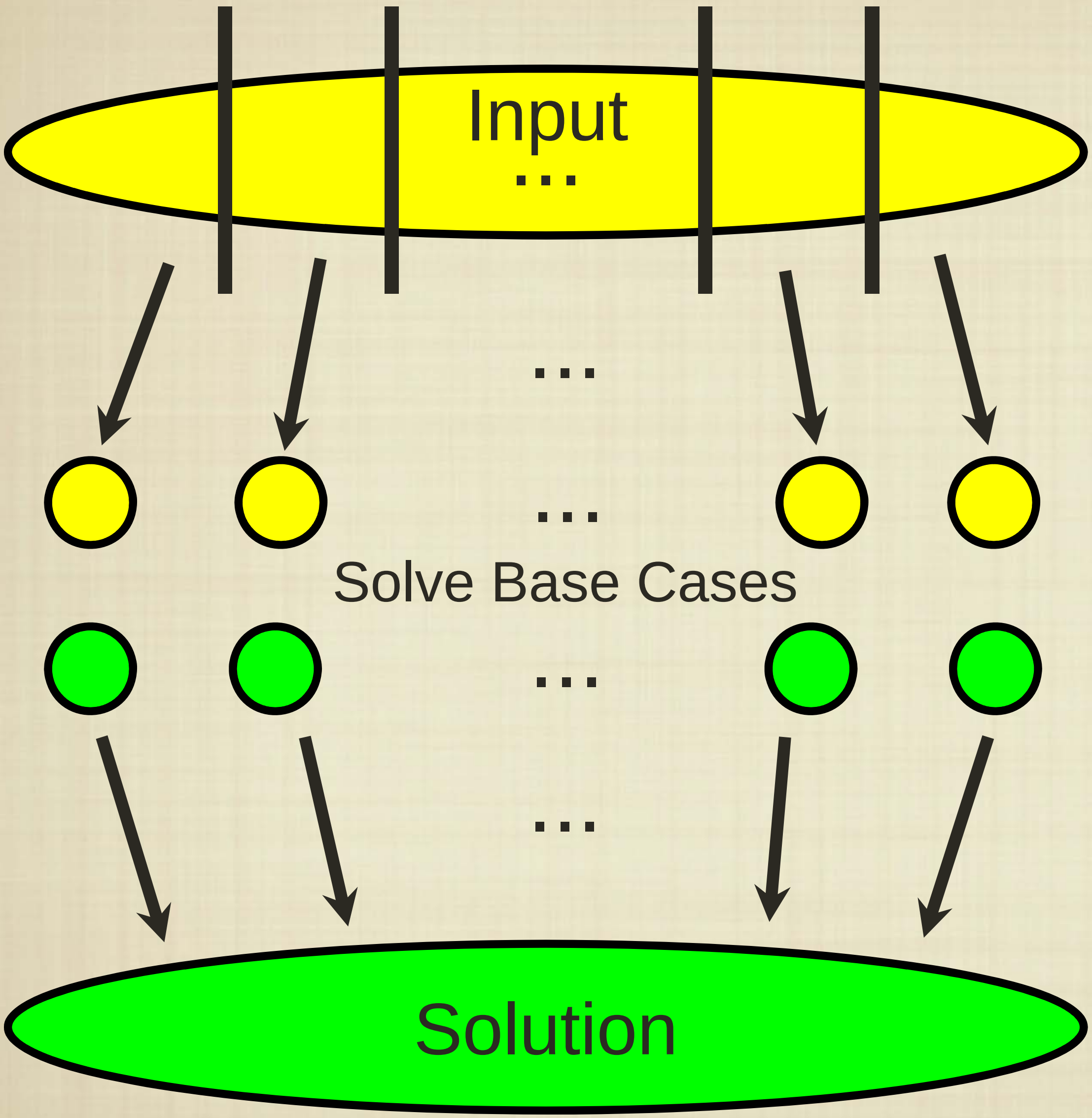
Recursion Tree



So, Merge Sort has runtime O(*n* log *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?

<u>Generalized Divide-and-Conquer Recurrence</u>

$$T(1) = c \qquad\qquad T(n) = \underbrace{f(n)}_{\text{work to merge}} + \underbrace{a}_{\text{number of recursive calls}} \cdot T(n/\underbrace{b}_{\text{size of each split}})$$

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

Real-World Sorting

Size: n	n^2	Selection Sort: seconds	Merge Sort: seconds	Tim Sort: seconds
10	100	0.000505	0.000556	0.000013
100	10000	0.002175	0.001619	0.000096
1,000	1,000,000	0.178361	0.020270	0.001035
10,000	100,000,000	17.010634	0.258054	0.015473
100,000	10,000,000,000	2524.767636	2.753175	0.182799

Selection Sort does not 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 on many lists, although its theoretical worst-case performance is identical to Merge Sort.

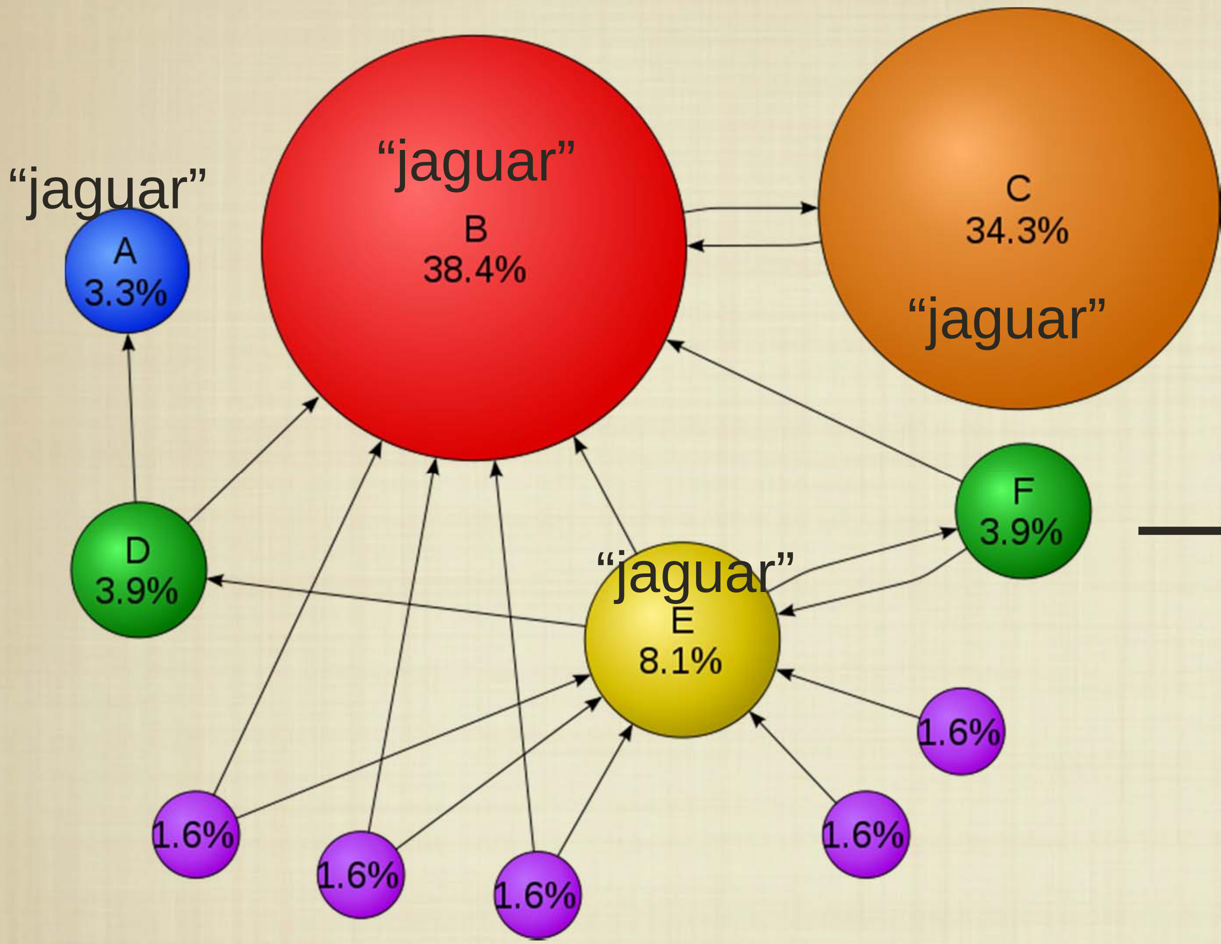
Real-World Sorting

Size: n	n^2	Selection Sort: seconds	Merge Sort: seconds	Tim Sort: seconds
10	100	0.000505	0.000556	0.000013
100	10000	0.002175	0.001619	0.000096
1,000	1,000,000	0.178361	0.020270	0.001035
10,000	100,000,000	17.010634	0.258054	0.015473
100,000	10,000,000,000	2524.767636	2.753175	0.182799

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.

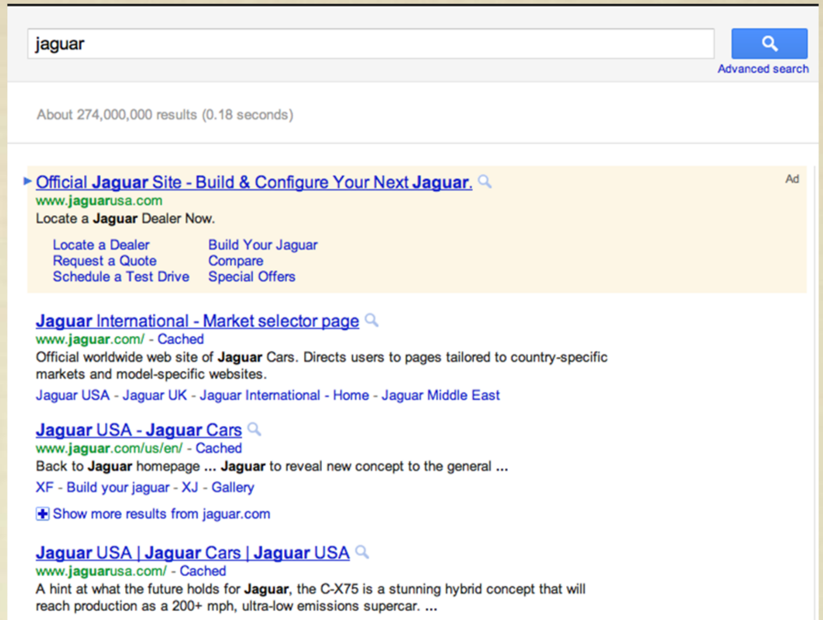
Google in a Nutshell



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.

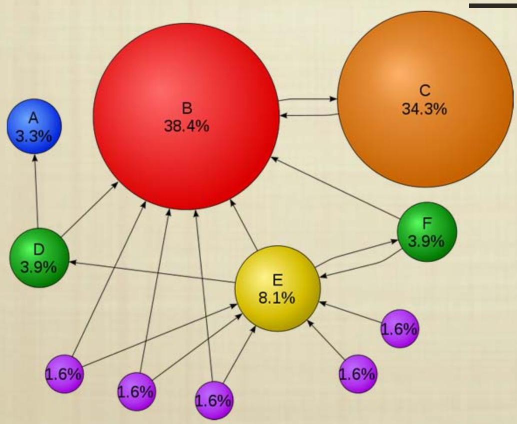
Google in a Nutshell



Query

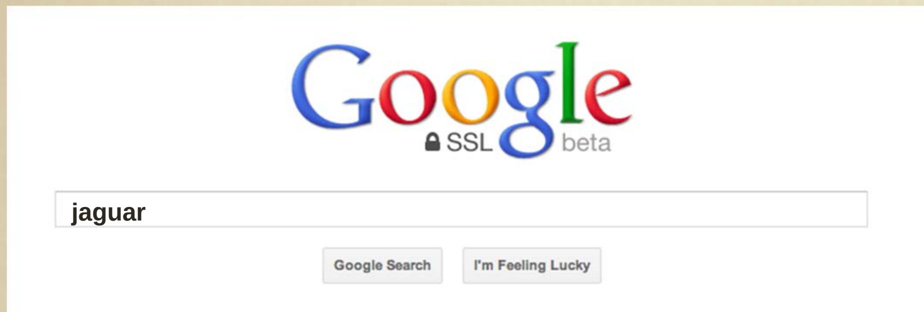
Result

Google Data Center

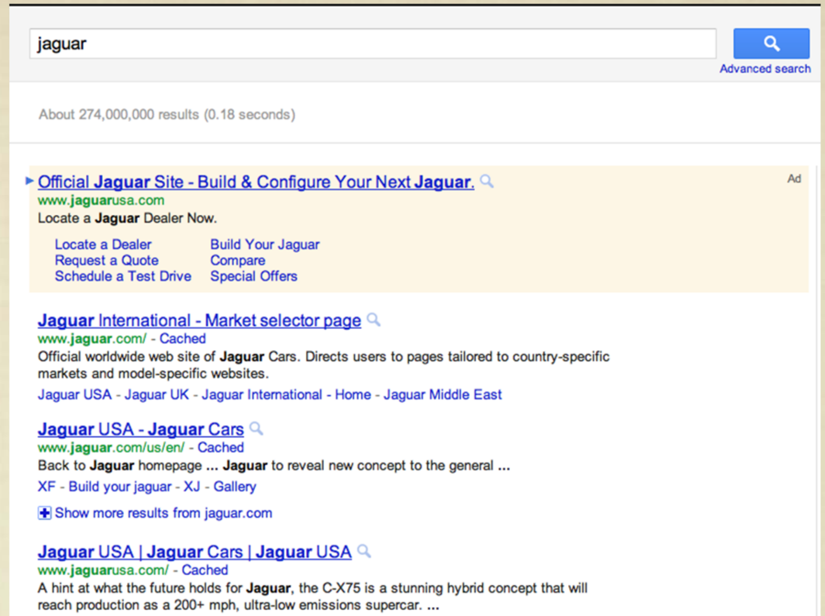


1. Search for query keywords in mined pages.
2. Select a set of “matching” pages and ads.
3. Sort pages by PageRank and return results.

Google in a Nutshell

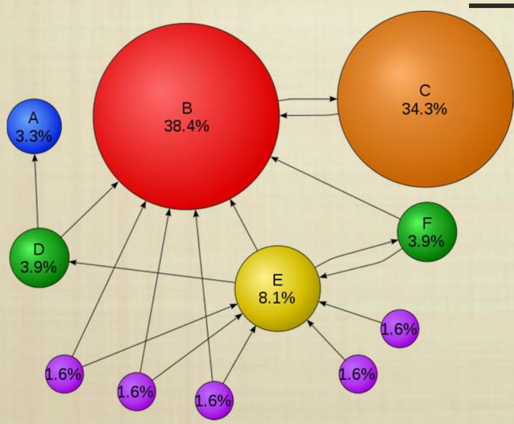


Query



Result

Google Data Center



1. Search for query keywords in mined pages.
2. Select a set of "matching" pages and <u>ads</u>.
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.”