Programming in Haskell
Aug-Nov 2015

LECTURE 11

SEPTEMBER 10, 2015

S P SURESH
CHENNAI MATHEMATICAL INSTITUTE

Measuring efficiency

Measuring efficiency

- Computation is reduction
    - Application of definitions as rewriting rules

Measuring efficiency

- Computation is reduction
  - Application of definitions as rewriting rules
- Count the number of reduction steps
  - Running time is T(n) for input size n

Example: Complexity of ++

Example: Complexity of ++

```
[] ++ y = y
(x:xs) ++ y = x:(xs++y)
```

Example: Complexity of ++

```
[] ++ y = y
(x:xs) ++ y = x:(xs++y)
```

- `[1,2,3] ++ [4,5,6]` ⇛

Example: Complexity of ++

```
[] ++ y = y
(x:xs) ++ y = x:(xs++y)
```

- `[1,2,3] ++ [4,5,6]` ⇛

  `1:([2,3] ++ [4,5,6])` ⇛

Example: Complexity of ++

```
[] ++ y = y
(x:xs) ++ y = x:(xs++y)
```

- `[1,2,3] ++ [4,5,6]` ⇛

  `1:([2,3] ++ [4,5,6])` ⇛

  `1:(2:([3] ++ [4,5,6]))` ⇛

Example: Complexity of ++

```
[] ++ y = y
(x:xs) ++ y = x:(xs++y)
```

- `[1,2,3] ++ [4,5,6]` ⇛

  `1:([2,3] ++ [4,5,6])` ⇛

  `1:(2:([3] ++ [4,5,6]))` ⇛

  `1:(2:(3:([] ++ [4,5,6])))` ⇛

Example: Complexity of ++

```
[] ++ y = y
(x:xs) ++ y = x:(xs++y)
```

- `[1,2,3] ++ [4,5,6]` ⇛

  `1:([2,3] ++ [4,5,6])` ⇛

  `1:(2:([3] ++ [4,5,6]))` ⇛

  `1:(2:(3:([] ++ [4,5,6])))` ⇛

  `1:(2:(3:([4,5,6])))`

Example: Complexity of ++

```
[] ++ y = y
(x:xs) ++ y = x:(xs++y)
```

- `[1,2,3] ++ [4,5,6]` ⇛

  `1:([2,3] ++ [4,5,6])` ⇛

  `1:(2:([3] ++ [4,5,6]))` ⇛

  `1:(2:(3:([] ++ [4,5,6])))` ⇛

  `1:(2:(3:([4,5,6])))`
- `l1 ++ l2` : use the second rule `length l1` times, first rule once, always

Example: elem

Example: elem

```
elem :: Int -> [Int] -> Bool
elem i [] = False
elem i (x:xs)
  | (i==x) = True
  | otherwise = elem i xs
```

Example: elem

```
elem :: Int -> [Int] -> Bool
elem i [] = False
elem i (x:xs)
  | (i==x) = True
  | otherwise = elem i xs
```

- `elem 3 [4,7,8,9] ⇛ elem 3 [7,8,9] ⇛ elem 3 [8,9] ⇛ elem 3 [9] ⇛ elem 3 [] ⇛ False`

Example: elem

```
elem :: Int -> [Int] -> Bool
elem i [] = False
elem i (x:xs)
  | (i==x) = True
  | otherwise = elem i xs
```

- `elem 3 [4,7,8,9]` ⇛ `elem 3 [7,8,9]` ⇛ `elem 3 [8,9]` ⇛ `elem 3 [9]` ⇛ `elem 3 []` ⇛ `False`

- `elem 3 [3,7,8,9]` ⇛ `True`

Example: elem

```
elem :: Int -> [Int] -> Bool
elem i [] = False
elem i (x:xs)
  | (i==x) = True
  | otherwise = elem i xs
```

- `elem 3 [4,7,8,9]` ⇛ `elem 3 [7,8,9]` ⇛ `elem 3 [8,9]` ⇛ `elem 3 [9]` ⇛ `elem 3 []` ⇛ `False`
- `elem 3 [3,7,8,9]` ⇛ `True`
- Complexity depends on input size and value

Variation across inputs

- Worst case complexity
  - Maximum running time over all inputs of size n
  - Pessimistic: may be rare
- Average case
  - More realistic, but difficult/impossible to compute

Asymptotic complexity

Asymptotic complexity

- Interested in T(n) in terms of orders of magnitude

Asymptotic complexity

- Interested in $T(n)$ in terms of orders of magnitude
- $f(n) = O(g(n))$ if there is a constant k such that $f(n) \leq k\, g(n)$ for all $n > 0$

Asymptotic complexity

- Interested in $T(n)$ in terms of orders of magnitude
- $f(n) = O(g(n))$ if there is a constant k such that $f(n) \leq k\, g(n)$ for all $n > 0$
  - $an^2 + bn + c = O(n^2)$ for all a,b,c (take $k = a+b+c$ if $a,b,c > 0$)

Asymptotic complexity

- Interested in $T(n)$ in terms of orders of magnitude
- $f(n) = O(g(n))$ if there is a constant k such that $f(n) \leq k\, g(n)$ for all $n > 0$
  - $an^2 + bn + c = O(n^2)$ for all a,b,c (take $k = a+b+c$ if $a,b,c > 0$)
- Ignore constant factors, lower order terms

Asymptotic complexity

- Interested in $T(n)$ in terms of orders of magnitude
- $f(n) = O(g(n))$ if there is a constant k such that $f(n) \leq k\, g(n)$ for all $n > 0$
    - $an^2 + bn + c = O(n^2)$ for all a,b,c (take $k = a+b+c$ if $a,b,c > 0$)
- Ignore constant factors, lower order terms
    - $O(n)$, $O(n \log n)$, $O(n^k)$, $O(2^n)$, …

Asymptotic complexity …

- Complexity of ++ is O(n), where n is the length of the first list
- Complexity of elem is O(n)
  - Worst case!

Complexity of reverse

Complexity of reverse

```
myreverse :: [a] -> [a]
myreverse [] = []
myreverse (x:xs) = (myreverse xs) ++ [x]
```

Complexity of reverse

```
myreverse :: [a] -> [a]
myreverse [] = []
myreverse (x:xs) = (myreverse xs) ++ [x]
```

- Analyze directly (like `++`), or write a recurrence for $T(n)$

Complexity of reverse

```
myreverse :: [a] -> [a]
myreverse [] = []
myreverse (x:xs) = (myreverse xs) ++ [x]
```

- Analyze directly (like ++), or write a recurrence for T(n)
  - `T(0) = 1`
    `T(n) = T(n-1) + n`

Complexity of reverse

```
myreverse :: [a] -> [a]
myreverse [] = []
myreverse (x:xs) = (myreverse xs) ++ [x]
```

- Analyze directly (like ++), or write a recurrence for T(n)

  -
    ```
    T(0) = 1
    T(n) = T(n-1) + n
    ```
- Solve by expanding the recurrence

Complexity of `reverse` …

```
T(0) = 1
T(n) = T(n-1) + n
```

Complexity of `reverse` …

- T(n) = T(n-1) + n

```
T(0) = 1
T(n) = T(n-1) + n
```

Complexity of `reverse` ...

- T(n) = T(n-1) + n

  = (T(n-2) + n-1) + n

```
T(0) = 1
T(n) = T(n-1) + n
```

Complexity of `reverse` ...

- T(n) = T(n-1) + n

  = (T(n-2) + n-1) + n

  = (T(n-3) + n-2) + n-1 + n

```
T(0) = 1
T(n) = T(n-1) + n
```

Complexity of `reverse` ...

- T(n) = T(n-1) + n

  = (T(n-2) + n-1) + n

  = (T(n-3) + n-2) + n-1 + n

  ...

```
T(0) = 1
T(n) = T(n-1) + n
```

Complexity of `reverse` ...

- T(n) = T(n-1) + n

  = (T(n-2) + n-1) + n

  = (T(n-3) + n-2) + n-1 + n

  ...

  = T(0) + 1 + 2 + ... + n

```
T(0) = 1
T(n) = T(n-1) + n
```

Complexity of `reverse` …

- T(n) = T(n-1) + n

  = (T(n-2) + n-1) + n

  = (T(n-3) + n-2) + n-1 + n

  …

  = T(0) + 1 + 2 + … + n

  = 1 + 1 + 2 + … + n = 1 + n(n+1)/2

```
T(0) = 1
T(n) = T(n-1) + n
```

Complexity of `reverse` ...

- $T(n) = T(n-1) + n$

  $= (T(n-2) + n-1) + n$

  $= (T(n-3) + n-2) + n-1 + n$

  ...

  $= T(0) + 1 + 2 + \ldots + n$

  $= 1 + 1 + 2 + \ldots + n = 1 + n(n+1)/2$

  $= O(n^2)$

```
T(0) = 1
T(n) = T(n-1) + n
```

Speeding up `reverse`

Speeding up `reverse`

- Can we do better?

Speeding up `reverse`

- Can we do better?
- Imagine we are reversing a stack of heavy stack of books

Speeding up `reverse`

- Can we do better?
- Imagine we are reversing a stack of heavy stack of books
- Transfer to a new stack, top to bottom

Speeding up `reverse`

- Can we do better?
- Imagine we are reversing a stack of heavy stack of books
- Transfer to a new stack, top to bottom
- New stack is in reverse order!

Speeding up `reverse` …

Speeding up `reverse` …

```
transfer :: [a] -> [a] -> [a]
transfer [] l = l
transfer (x:xs) l = transfer xs (x:l)
```

Speeding up `reverse` …

```
transfer :: [a] -> [a] -> [a]
transfer [] l = l
transfer (x:xs) l = transfer xs (x:l)
```

- Input size for `transfer l1 l2` is `length l1`

Speeding up `reverse` …

```
transfer :: [a] -> [a] -> [a]
transfer [] l = l
transfer (x:xs) l = transfer xs (x:l)
```

- Input size for `transfer l1 l2` is `length l1`
- Recurrence

Speeding up `reverse` …

```
transfer :: [a] -> [a] -> [a]
transfer [] l = l
transfer (x:xs) l = transfer xs (x:l)
```

- Input size for `transfer l1 l2` is `length l1`
- Recurrence

```
T(0) = 1
T(n) = T(n-1) + 1
```

Speeding up `reverse` ...

```
transfer :: [a] -> [a] -> [a]
transfer [] l = l
transfer (x:xs) l = transfer xs (x:l)
```

- Input size for `transfer l1 l2` is `length l1`
- Recurrence

```
T(0) = 1
T(n) = T(n-1) + 1
```

- Expanding: T(n) = 1 + 1 + ... + 1 = O(n)

Speeding up `reverse` ...

```
fastreverse :: [a] -> [a]
fastreverse l = transfer l []
```

- Complexity is O(n)
- Need to understand the computational model to achieve efficiency

Summary

- Measure complexity in Haskell in terms of reduction steps
- Account for input size and values
  - Usually worst-case complexity
- Asymptotic complexity
  - Ignore constants, lower order terms
  - $T(n) = O(f(n))$

Sorting

Sorting

- Goal is to arrange a list in ascending order

Sorting

- Goal is to arrange a list in ascending order
- How would we start a pack of cards?

Sorting

- Goal is to arrange a list in ascending order
- How would we start a pack of cards?
    - A single card is sorted

Sorting

- Goal is to arrange a list in ascending order
- How would we start a pack of cards?
    - A single card is sorted
    - Put second card before/after first

Sorting

- Goal is to arrange a list in ascending order
- How would we start a pack of cards?
  - A single card is sorted
  - Put second card before/after first
  - “Insert” third, fourth,… card in correct place

Sorting

- Goal is to arrange a list in ascending order
- How would we start a pack of cards?
    - A single card is sorted
    - Put second card before/after first
    - “Insert” third, fourth,… card in correct place
- Insertion sort

Insertion sort : insert

Insertion sort : insert

- Insert an element in a sorted list

Insertion sort : insert

- Insert an element in a sorted list

```
insert :: Int -> [Int] -> [Int]
insert x [] = [x]
insert x (y:ys)
  | (x <= y) = x:y:ys
  | otherwise y:(insert x ys)
```

Insertion sort : insert

- Insert an element in a sorted list

```
insert :: Int -> [Int] -> [Int]
insert x [] = [x]
insert x (y:ys)
  | (x <= y) = x:y:ys
  | otherwise y:(insert x ys)
```

- Clearly T(n) = O(n)

Insertion sort : isort

Insertion sort : isort

```
isort :: [Int] -> [Int]
isort [] = []
isort (x:xs) = insert x (isort xs)
```

Insertion sort : isort

```
isort :: [Int] -> [Int]
isort [] = []
isort (x:xs) = insert x (isort xs)
```

- Alternatively

Insertion sort : isort

```
isort :: [Int] -> [Int]
isort [] = []
isort (x:xs) = insert x (isort xs)
```

- Alternatively

```
isort = foldr insert []
```

Insertion sort : isort

```
isort :: [Int] -> [Int]
isort [] = []
isort (x:xs) = insert x (isort xs)
```

- Alternatively

```
isort = foldr insert []
```

- Recurrence

Insertion sort : isort

```
isort :: [Int] -> [Int]
isort [] = []
isort (x:xs) = insert x (isort xs)
```

- Alternatively

```
isort = foldr insert []
```

- Recurrence

```
T(0) = 1
T(n) = T(n-1) + O(n)
```

Insertion sort : isort

```
isort :: [Int] -> [Int]
isort [] = []
isort (x:xs) = insert x (isort xs)
```

- Alternatively

```
isort = foldr insert []
```

- Recurrence

```
  T(0) = 1
  T(n) = T(n-1) + O(n)
```

- Complexity: $T(n) = O(n^2)$

A better strategy?

- Divide list in two equal parts
- Separately sort left and right half
- Combine the two sorted halves to get the full list sorted

Combining sorted lists

- Given two sorted lists `l1` and `l2`, combine into a sorted list `l3`
  - Compare first element of `l1` and `l2`
  - Move it into `l3`
  - Repeat until all elements in `l1` and `l2` are over
- Merging `l1` and `l2`

Merging two sorted lists

32 74 89

21 55 64

Merging two sorted lists

32 74 89

~~21~~ 55 64

21

Merging two sorted lists

~~32~~ 74 89

~~21~~ 55 64

21 32

Merging two sorted lists

~~32~~ 74 89

~~21~~ ~~55~~ 64

21 32 55

Merging two sorted lists

~~32~~ 74 89

~~21~~ ~~55~~ ~~64~~

21 32 55 64

Merging two sorted lists

~~32~~ ~~74~~ 89

~~21~~ ~~55~~ ~~64~~

21 32 55 64 74

Merging two sorted lists

~~32~~ ~~74~~ ~~89~~

~~21~~ ~~55~~ ~~64~~

21 32 55 64 74 89

Merge Sort

Merge Sort

- Sort `l!!0` to `l!!(n/2-1)`

Merge Sort

- Sort `l!!0` to `l!!(n/2-1)`
- Sort `l!!(n/2)` to `l!!(n-1)`

Merge Sort

- Sort `l!!0` to `l!!(n/2-1)`
- Sort `l!!(n/2)` to `l!!(n-1)`
- Merge sorted halves into `l'`

Merge Sort

- Sort `l!!0` to `l!!(n/2-1)`
- Sort `l!!(n/2)` to `l!!(n-1)`
- Merge sorted halves into `l'`
- How do we sort the halves?

Merge Sort

- Sort `l!!0` to `l!!(n/2-1)`
- Sort `l!!(n/2)` to `l!!(n-1)`
- Merge sorted halves into `l'`
- How do we sort the halves?
  - Recursively, using the same strategy!

Merge Sort

43	32	22	78	63	57	91	13

Merge Sort

43	32	22	78	63	57	91	13

43	32	22	78

Merge Sort

43	32	22	78	63	57	91	13

43	32	22	78

63	57	91	13

Merge Sort

43	32	22	78	63	57	91	13

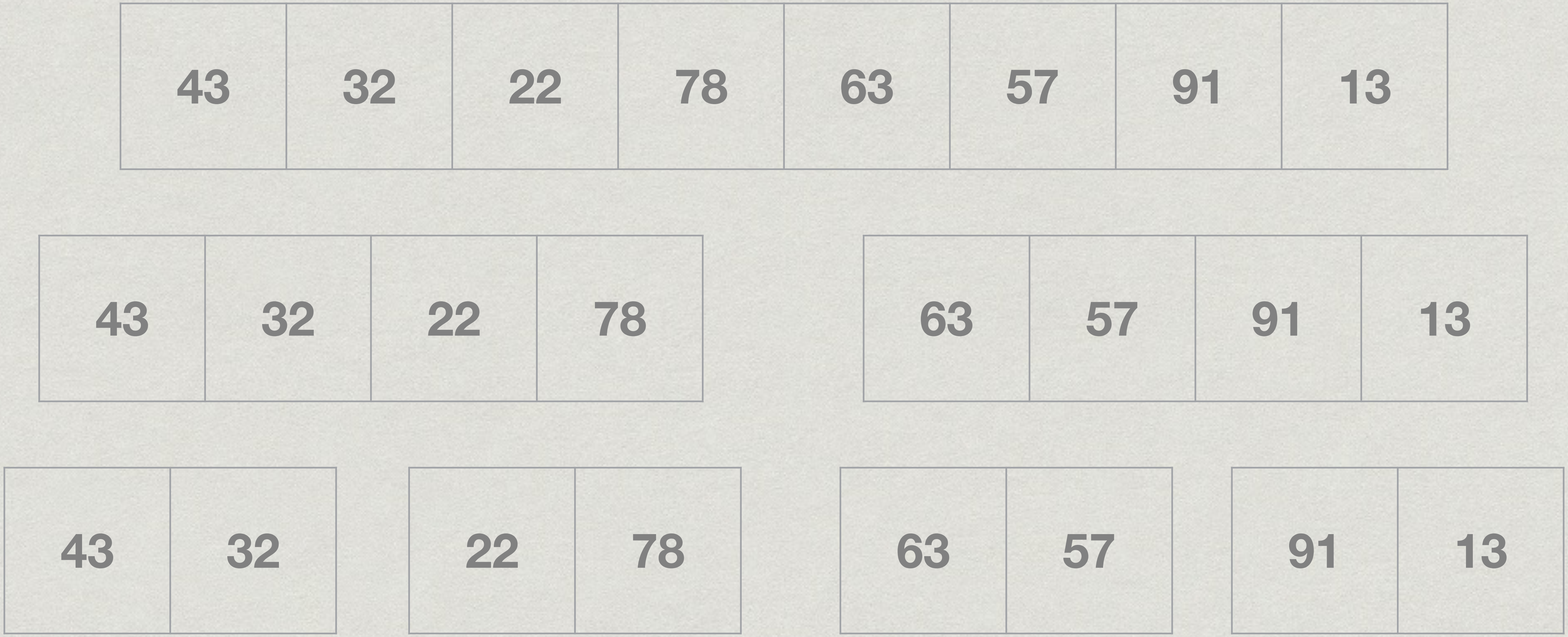
43	32	22	78

63	57	91	13

43	32

22	78

Merge Sort

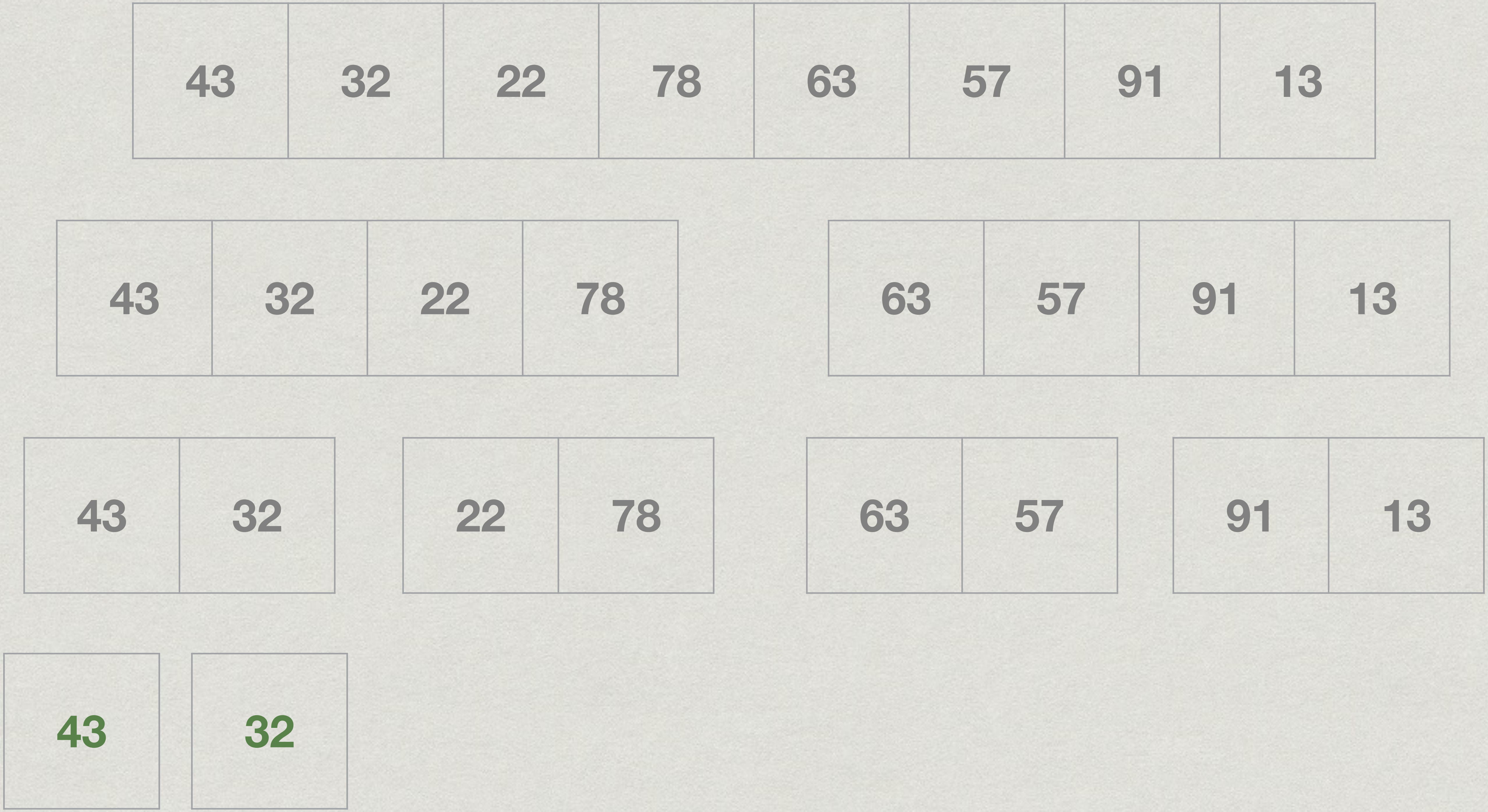


43 | 32 | 22 | 78 | 63 | 57 | 91 | 13

43 | 32 | 22 | 78   63 | 57 | 91 | 13

43 | 32   22 | 78   63 | 57   91 | 13

Merge Sort



43	32	22	78	63	57	91	13

43	32	22	78

63	57	91	13

43	32

22	78

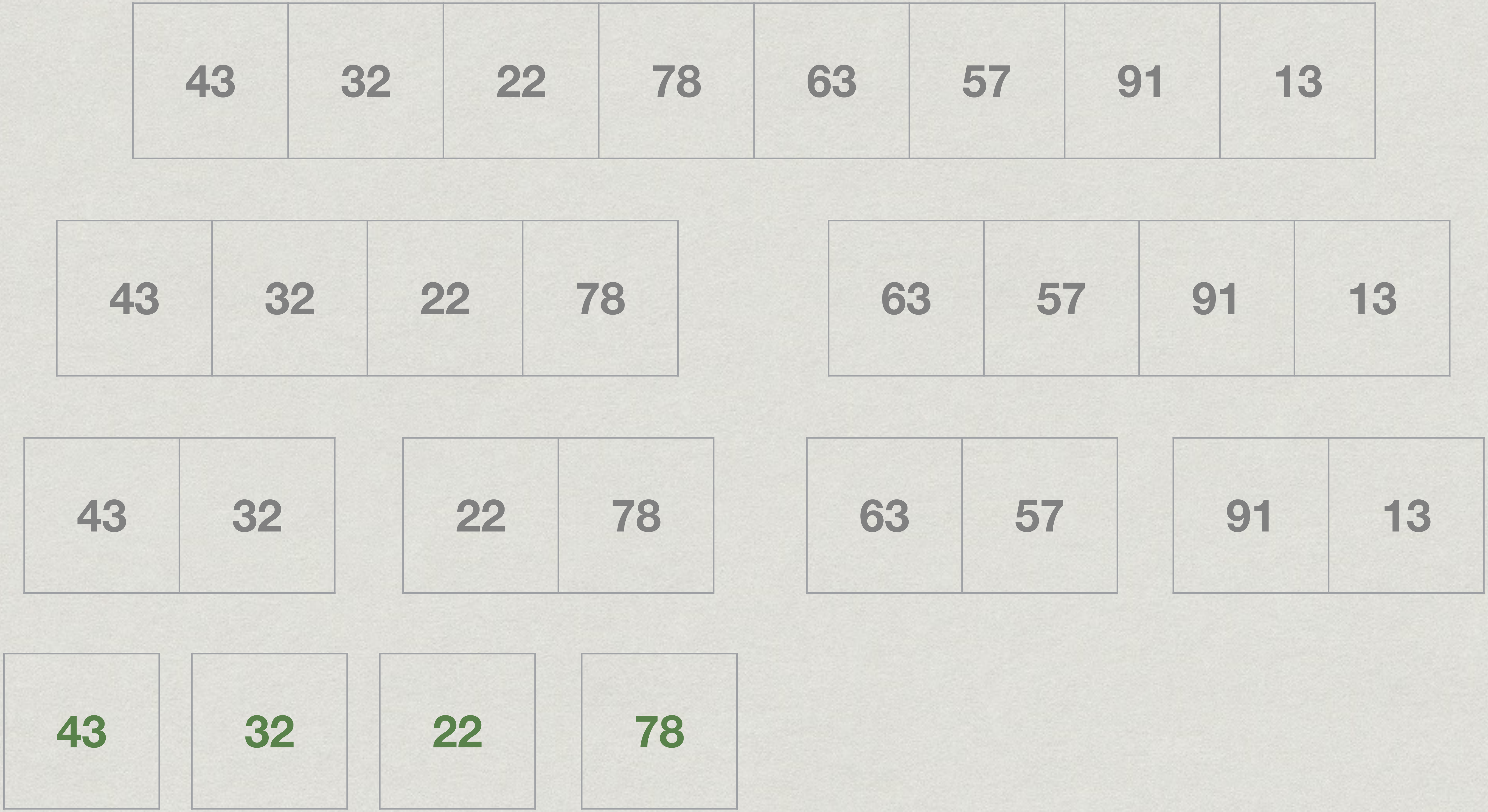
63	57

91	13

43

32

Merge Sort



43 32 22 78 63 57 91 13

43 32 22 78

63 57 91 13

43 32

22 78

63 57

91 13

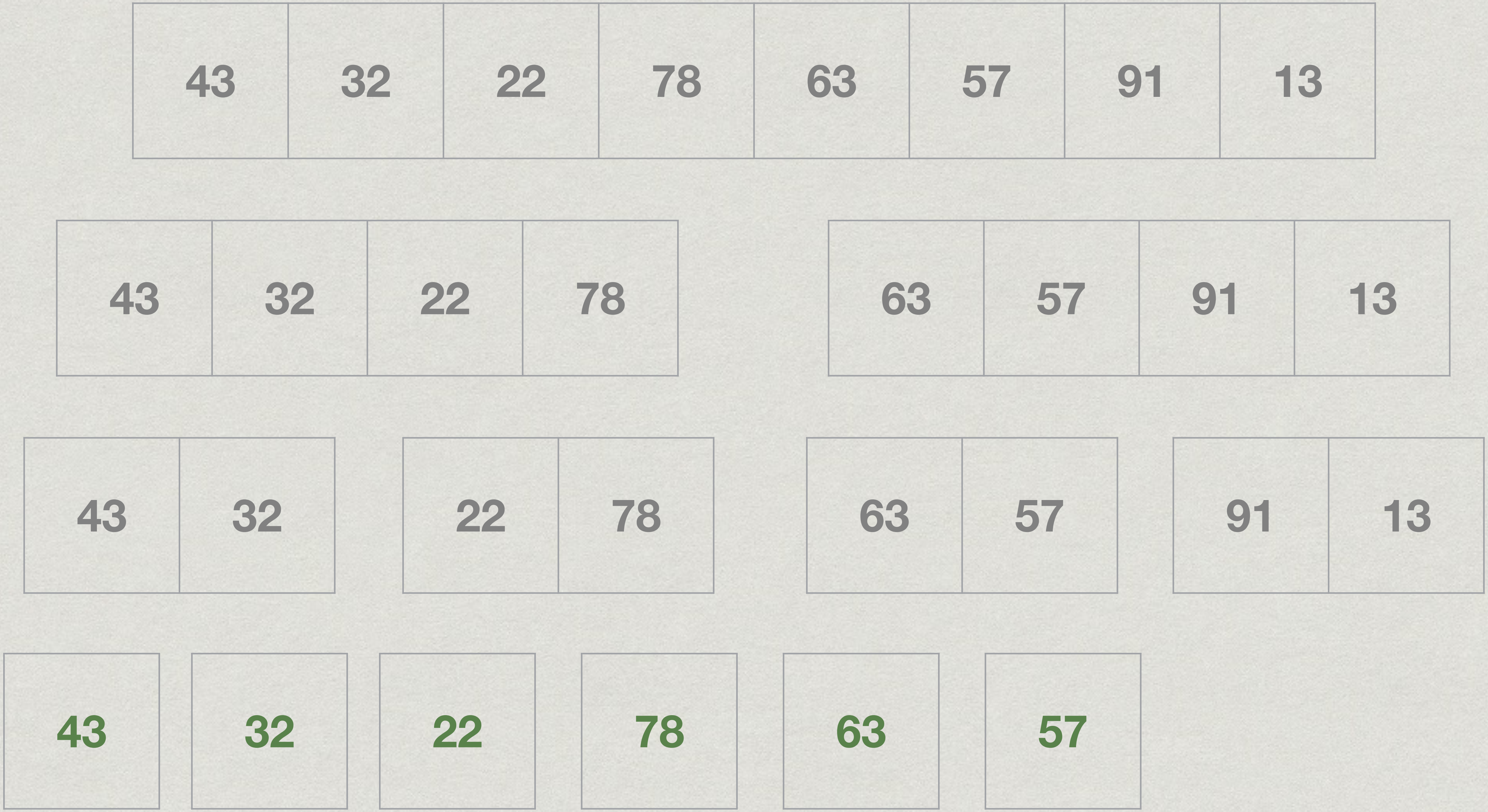
43

32

22

78

Merge Sort



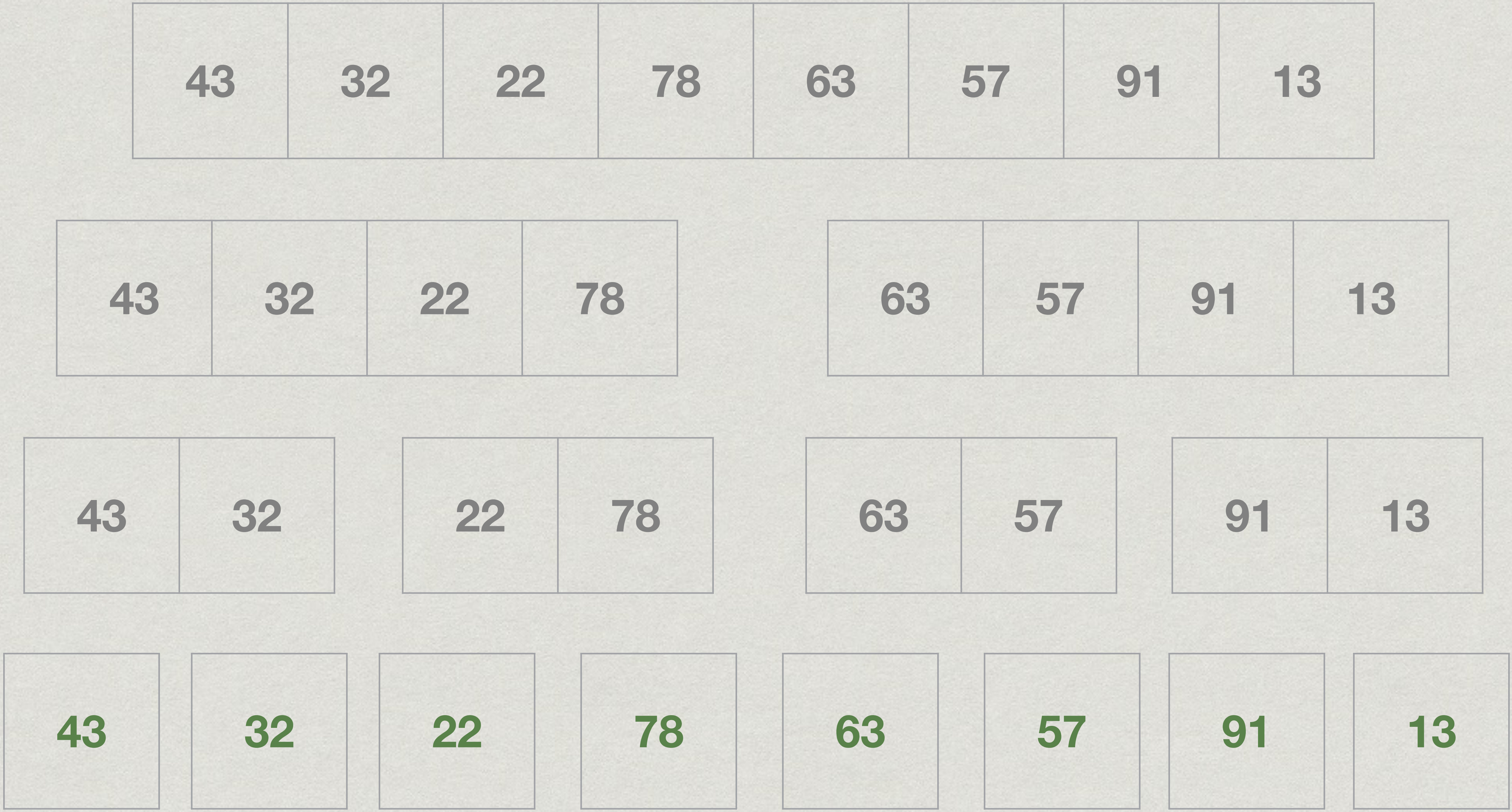
43 32 22 78 63 57 91 13

43 32 22 78   63 57 91 13

43 32   22 78   63 57   91 13

43   32   22   78   63   57

Merge Sort



43 32 22 78 63 57 91 13

43 32 22 78

63 57 91 13

43 32

22 78

63 57

91 13

43

32

22

78

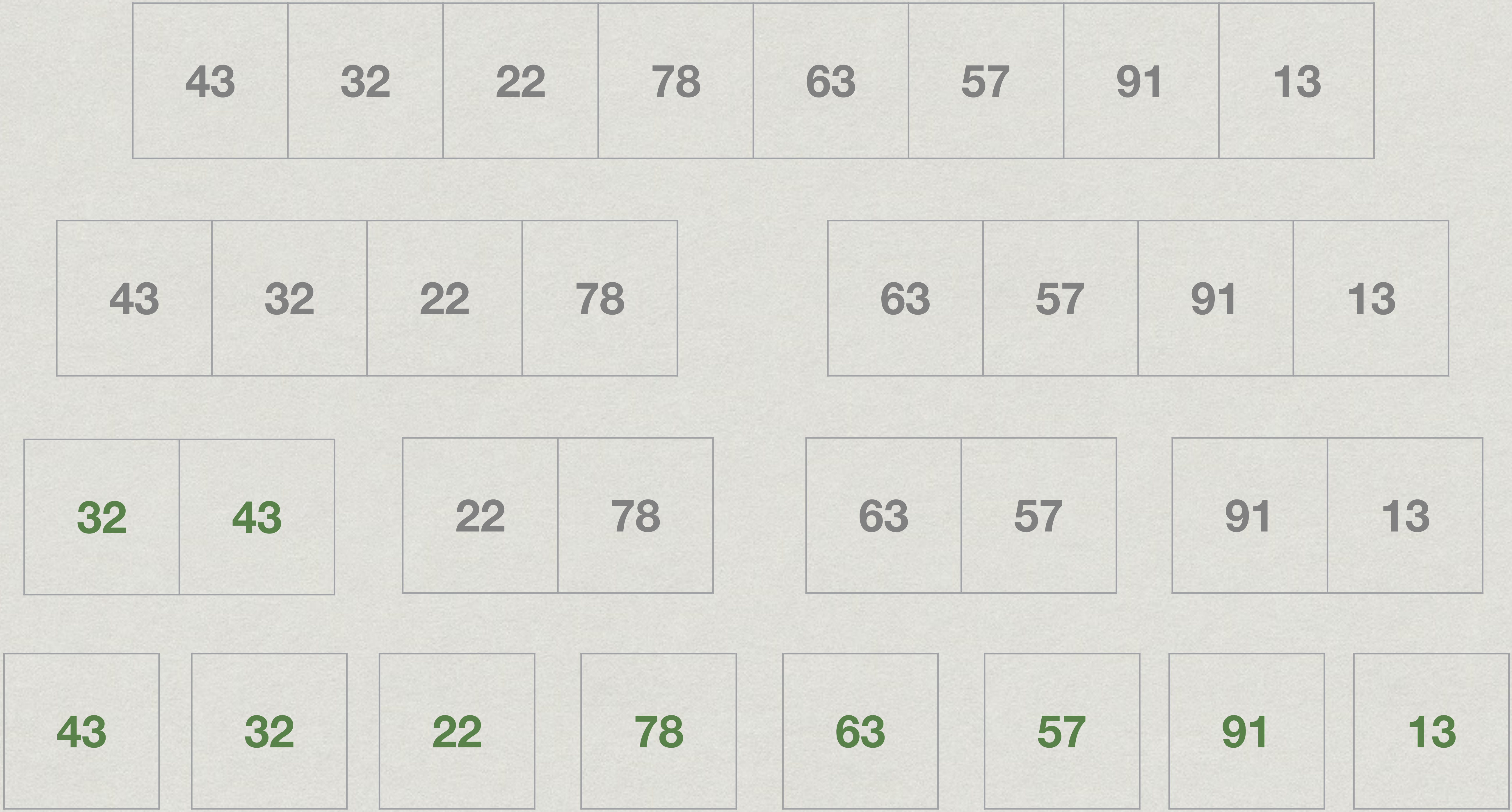
63

57

91

13

Merge Sort



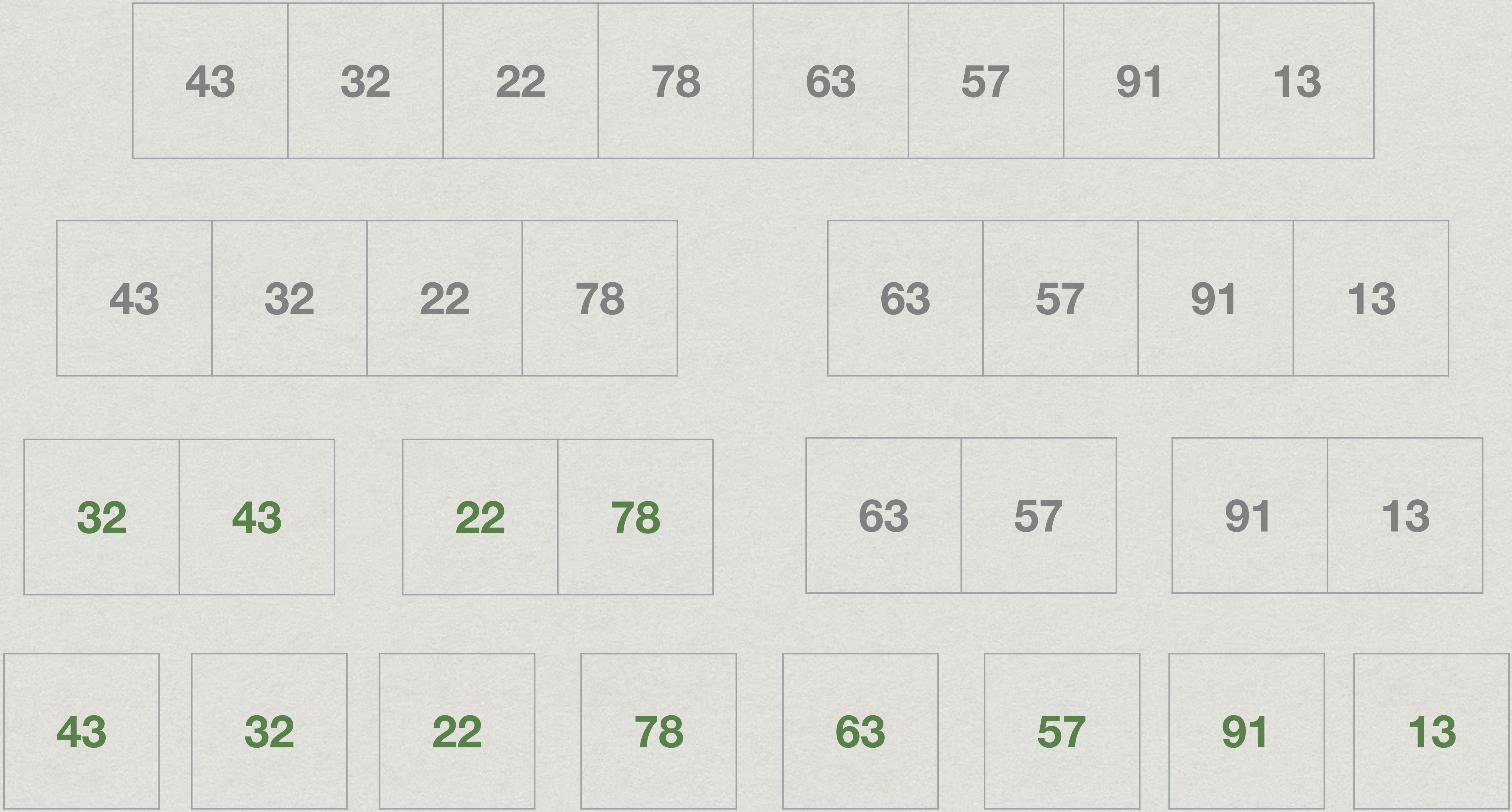
43 32 22 78 63 57 91 13

43 32 22 78   63 57 91 13

32 43   22 78   63 57   91 13

43   32   22   78   63   57   91   13

Merge Sort



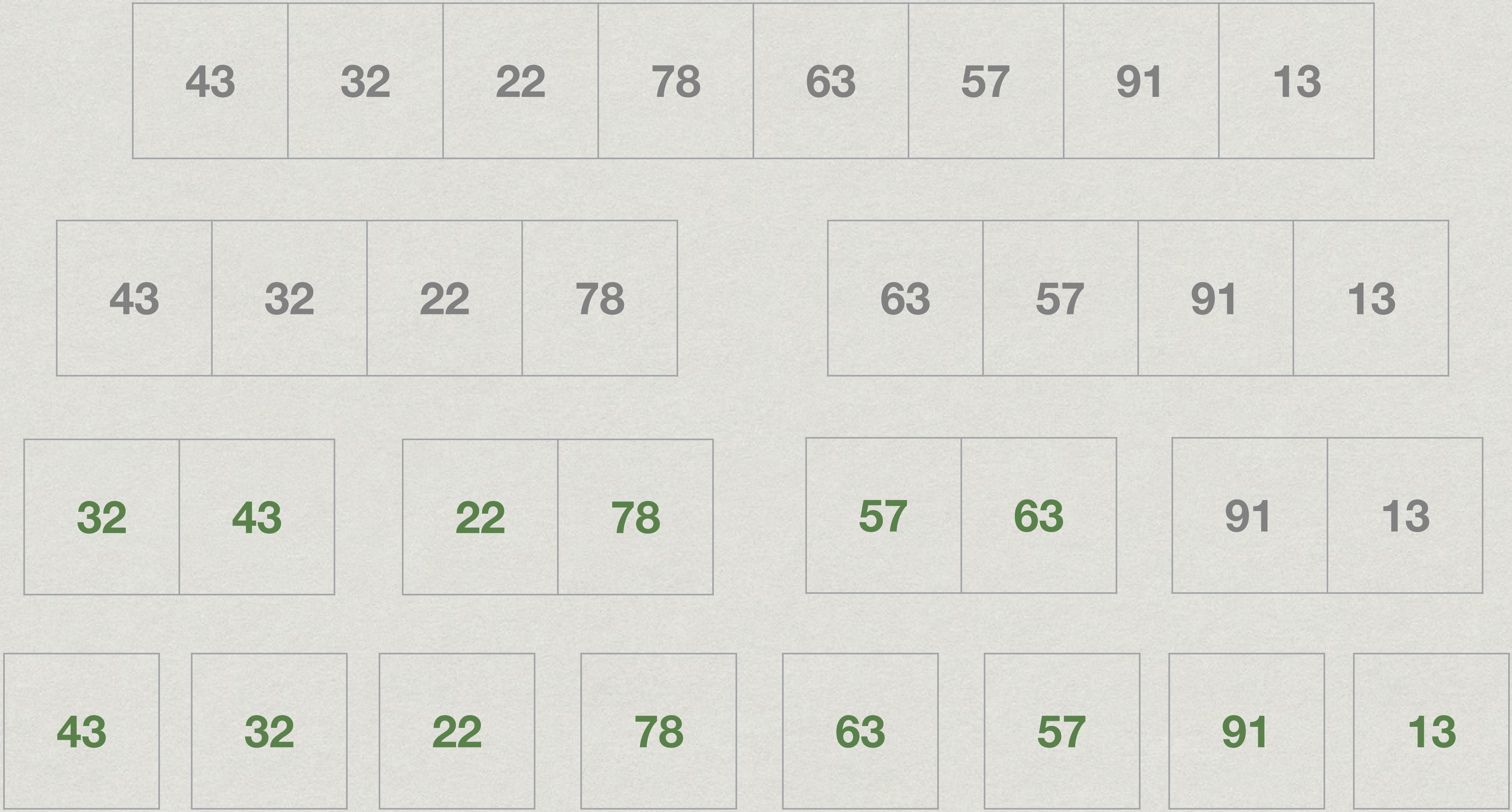
43 32 22 78 63 57 91 13

43 32 22 78   63 57 91 13

32 43   22 78   63 57   91 13

43   32   22   78   63   57   91   13

Merge Sort



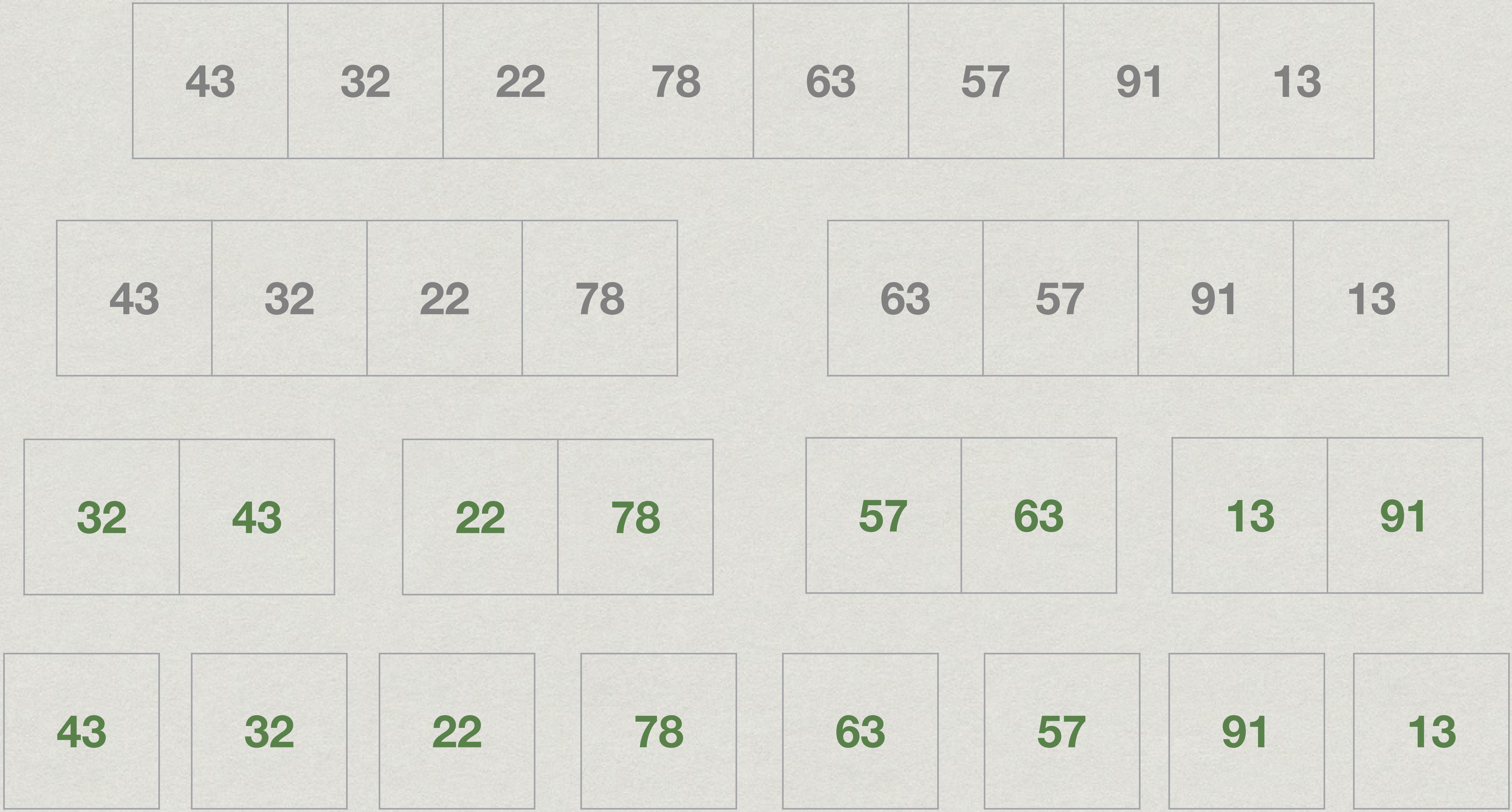
43 32 22 78 63 57 91 13

43 32 22 78   63 57 91 13

32 43   22 78   57 63   91 13

43   32   22   78   63   57   91   13

Merge Sort



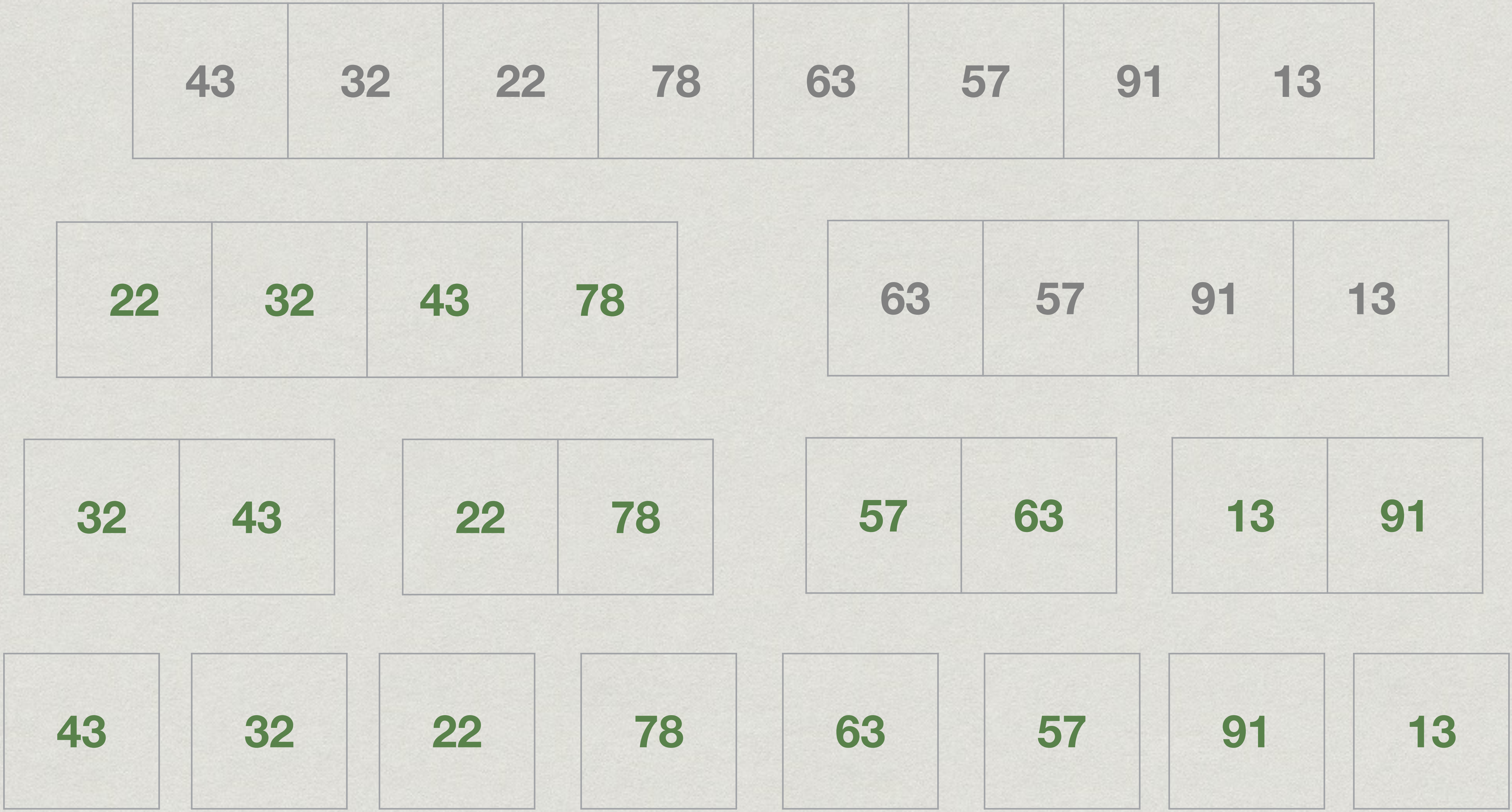
43 32 22 78 63 57 91 13

43 32 22 78 | 63 57 91 13

32 43 | 22 78 | 57 63 | 13 91

43 | 32 | 22 | 78 | 63 | 57 | 91 | 13

Merge Sort



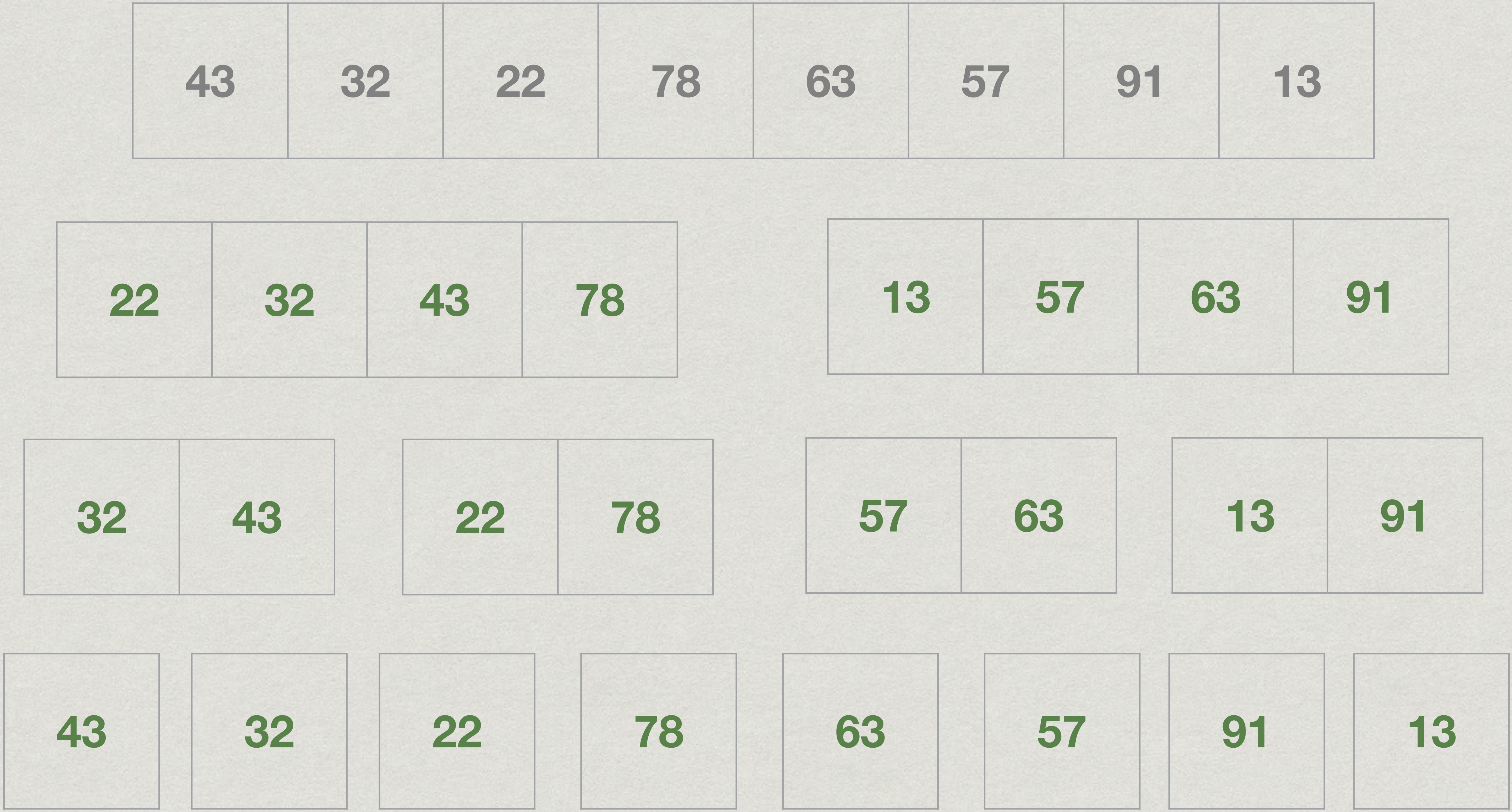
| 43 | 32 | 22 | 78 | 63 | 57 | 91 | 13 |

| 22 | 32 | 43 | 78 | | 63 | 57 | 91 | 13 |

| 32 | 43 | | 22 | 78 | | 57 | 63 | | 13 | 91 |

| 43 | | 32 | | 22 | | 78 | | 63 | | 57 | | 91 | | 13 |

Merge Sort



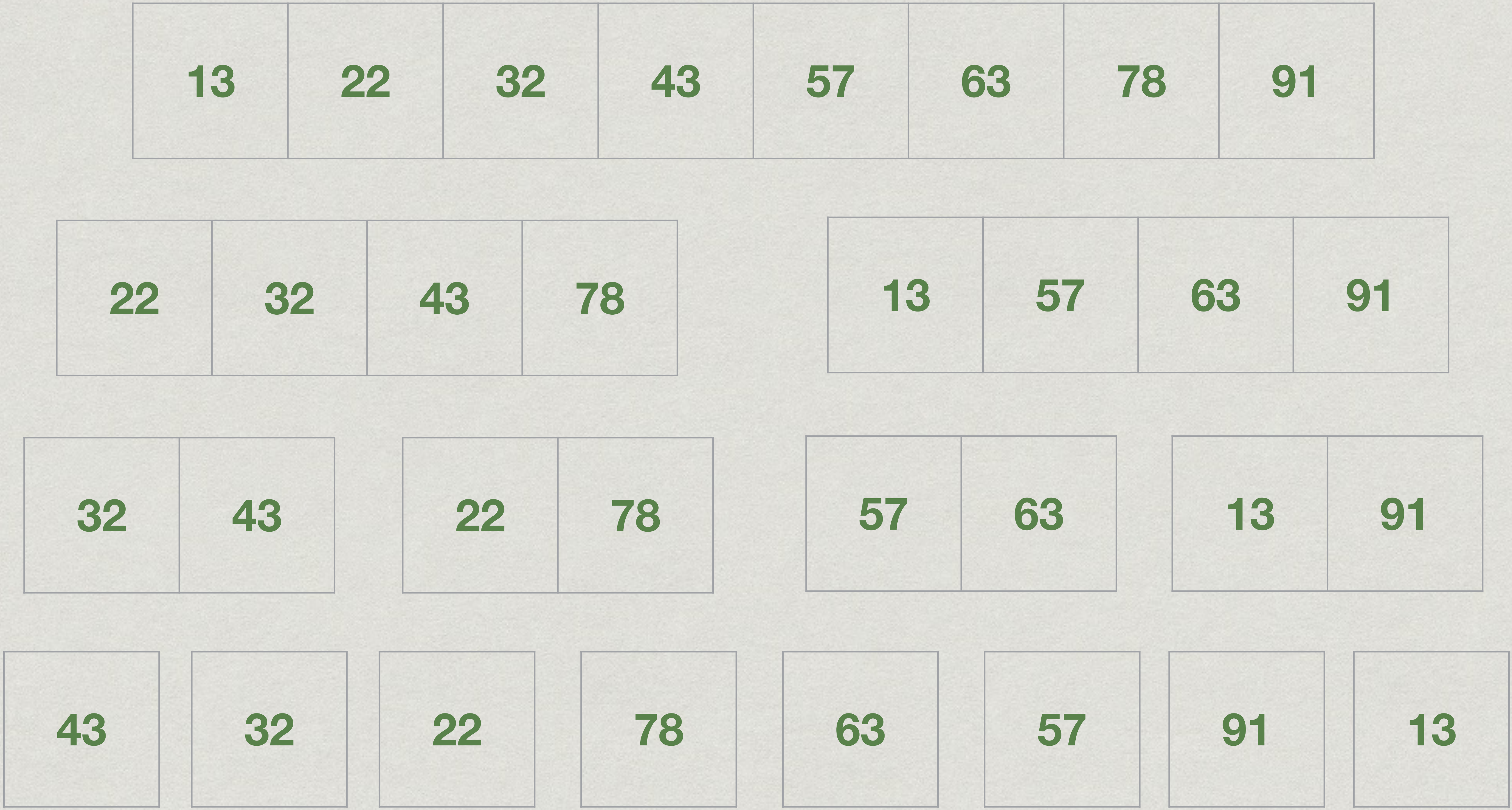
43 32 22 78 63 57 91 13

22 32 43 78 | 13 57 63 91

32 43 | 22 78 | 57 63 | 13 91

43 | 32 | 22 | 78 | 63 | 57 | 91 | 13

Merge Sort



13 22 32 43 57 63 78 91

22 32 43 78   13 57 63 91

32 43   22 78   57 63   13 91

43   32   22   78   63   57   91   13

Merge sort : merge

Merge sort : merge

```
merge :: [Int] -> [Int] -> [Int}
merge [] ys = ys
merge xs [] = xs

merge (x:xs) (y:ys)
  | x <= y    = x:(merge xs (y:ys))
  | otherwise = y:(merge (x:xs) ys)
```

Merge sort : merge

```
merge :: [Int] -> [Int] -> [Int}
merge [] ys = ys
merge xs [] = xs

merge (x:xs) (y:ys)
  | x <= y    = x:(merge xs (y:ys))
  | otherwise = y:(merge (x:xs) ys)
```

- Each comparison adds one element to output

Merge sort : merge

```
merge :: [Int] -> [Int] -> [Int}
merge [] ys = ys
merge xs [] = xs

merge (x:xs) (y:ys)
  | x <= y     = x:(merge xs (y:ys))
  | otherwise = y:(merge (x:xs) ys)
```

- Each comparison adds one element to output
- $T(n) = O(n)$, where n is sum of lengths of input lists

Merge sort

```
mergesort :: [Int] -> [Int]
mergesort [] = []
mergesort [x] = [x]
mergesort l = merge (mergesort (front l))
                    (mergesort (back l))
  where
  front l = take ((length l) `div` 2) l
  back l = drop ((length l) `div` 2) l
```

Analysis of Merge Sort

- $T(n)$: time taken by Merge Sort on input of size n
  - Assume, for simplicity, that $n = 2^k$
  - $T(n) = 2T(n/2) + 2n$
  - Two subproblems of size $n/2$
  - Splitting the list into front and back takes n steps
  - Merging solutions requires time $O(n/2+n/2) = O(n)$
- Solve the recurrence by unwinding

Analysis of Merge Sort ...

Analysis of Merge Sort …

- $T(1) = 1$

Analysis of Merge Sort …

- $T(1) = 1$
- $T(n) = 2T(n/2) + 2n$

Analysis of Merge Sort …

- $T(1) = 1$
- $T(n) = 2T(n/2) + 2n$

  $= 2\ [\ 2T(n/4) + n] + 2n = 2^2\, T(n/2^2) + 4n$

Analysis of Merge Sort ...

- $T(1) = 1$
- $T(n) = 2T(n/2) + 2n$

  $= 2 [2T(n/4) + n] + 2n = 2^2 T(n/2^2) + 4n$

  $= 2^2 [2T(n/2^3) + 2n/2^2] + 4n = 2^3 T(n/2^3) + 6n$

  ...

Analysis of Merge Sort …

- $T(1) = 1$
- $T(n) = 2T(n/2) + 2n$

$= 2 [2T(n/4) + n] + 2n = 2^2 T(n/2^2) + 4n$

$= 2^2 [2T(n/2^3) + 2n/2^2] + 4n = 2^3 T(n/2^3) + 6n$

…

$= 2^j T(n/2^j) + 2jn$

Analysis of Merge Sort ...

- $T(1) = 1$
- $T(n) = 2T(n/2) + 2n$

  $= 2 [2T(n/4) + n] + 2n = 2^2 T(n/2^2) + 4n$

  $= 2^2 [2T(n/2^3) + 2n/2^2] + 4n = 2^3 T(n/2^3) + 6n$

  ...

  $= 2^j T(n/2^j) + 2jn$

- When $j = \log n$, $n/2^j = 1$, so $T(n/2^j) = 1$

Analysis of Merge Sort ...

- $T(1) = 1$
- $T(n) = 2T(n/2) + 2n$

  $= 2 [\ 2T(n/4) + n] + 2n = 2^2\, T(n/2^2) + 4n$

  $= 2^2 [\ 2T(n/2^3) + 2n/2^2] + 4n = 2^3\, T(n/2^3) + 6n$

  ...

  $= 2^j\, T(n/2^j) + 2jn$

- When $j = \log n$, $n/2^j = 1$, so $T(n/2^j) = 1$
- $T(n) = 2^j\, T(n/2^j) + 2jn = 2^{\log n} + 2(\log n)\, n =$ $n + 2n \log n = O(n \log n)$

Avoid merging

Avoid merging

- Some elements in left half move right and vice versa

Avoid merging

- Some elements in left half move right and vice versa
- Can we ensure that everything to the left is smaller than everything to the right?

Avoid merging

- Some elements in left half move right and vice versa
- Can we ensure that everything to the left is smaller than everything to the right?
- Suppose the median value in list is m

Avoid merging

- Some elements in left half move right and vice versa
- Can we ensure that everything to the left is smaller than everything to the right?
- Suppose the median value in list is m
    - Move all values ≤ m to left half of list

Avoid merging

- Some elements in left half move right and vice versa
- Can we ensure that everything to the left is smaller than everything to the right?
- Suppose the median value in list is m
    - Move all values $\leq m$ to left half of list
    - Right half has values $> m$

Avoid merging

- Some elements in left half move right and vice versa
- Can we ensure that everything to the left is smaller than everything to the right?
- Suppose the median value in list is m
  - Move all values $\leq m$ to left half of list
  - Right half has values $> m$
- Recursively sort left and right halves

Avoid merging

- Some elements in left half move right and vice versa
- Can we ensure that everything to the left is smaller than everything to the right?
- Suppose the median value in list is m
    - Move all values ≤ m to left half of list
    - Right half has values > m
- Recursively sort left and right halves
- List is now sorted! No need to merge

Avoid merging …

Avoid merging …

- How do we find the median?

Avoid merging …

- How do we find the median?
    - Sort and pick up middle element

Avoid merging …

- How do we find the median?
  - Sort and pick up middle element
  - But our aim is to sort!

Avoid merging …

- How do we find the median?
  - Sort and pick up middle element
  - But our aim is to sort!
- Instead, pick up some value in list — **pivot**

Avoid merging …

- How do we find the median?
  - Sort and pick up middle element
  - But our aim is to sort!
- Instead, pick up some value in list — **pivot**
  - Split list with respect to this pivot element

Quicksort

Quicksort

- Choose a pivot element

Quicksort

- Choose a pivot element
    - Typically the first value in the list

Quicksort

- Choose a pivot element
    - Typically the first value in the list
- Partition list into lower and upper parts with respect to pivot

Quicksort

- Choose a pivot element
    - Typically the first value in the list
- Partition list into lower and upper parts with respect to pivot
- Move pivot between lower and upper partition

Quicksort

- Choose a pivot element
  - Typically the first value in the list
- Partition list into lower and upper parts with respect to pivot
- Move pivot between lower and upper partition
- Recursively sort the two partitions

Quicksort

- High level view

Quicksort

- High level view

43	32	22	78	63	57	91	13

Quicksort

- High level view

43	32	22	78	63	57	91	13

Quicksort

- High level view

43	32	22	78	63	57	91	13

Quicksort

- High level view

13	32	22	43	63	57	91	78

Quicksort

- High level view

13	22	32	43	57	63	78	91

Quicksort

```
quicksort :: [Int] -> [Int]
quicksort [] = []
quicksort (x:xs) = (quicksort lower) ++
                   [splitter] ++
                   (quicksort upper)
   where
   splitter = x
   lower    = [ y | y <- xs, y <= x ]
   upper    = [ y | y <- xs, y > x ]
```

Analysis of Quicksort

Worst case

- Pivot is maximum or minimum
  - One partition is empty
  - Other is size $n-1$
  - $T(n) = T(n-1) + n = T(n-2) + (n-1) + n$
    $= \ldots = 1 + 2 + \ldots + n = O(n^2)$
- Already sorted array is worst case input!

Analysis of Quicksort

But …

- Average case is O(n log n)
  - Sorting is a rare example where average case can be computed
- What does average case mean?

Quicksort: Average case

- Assume input is a permutation of {1,2,…,n}
  - Actual values not important
  - Only relative order matters
  - Each input is equally likely (uniform probability)
- Calculate running time across all inputs
- **Expected** running time can be shown O(n log n)

Summary

- Sorting is an important starting point for many functions on lists
- Insertion sort is a natural inductive sort whose complexity is $O(n^2)$
- Merge sort has complexity $O(n \log n)$
- Quicksort has worst-case complexity $O(n^2)$ but average-case complexity $O(n \log n)$