NPTEL MOOC

PROGRAMMING, DATA STRUCTURES AND ALGORITHMS IN PYTHON

Week 4, Lecture 1

Madhavan Mukund, Chennai Mathematical Institute
http://www.cmi.ac.in/~madhavan

$O(n^2)$ sorting algorithms

- Selection sort and insertion sort are both $O(n^2)$
- $O(n^2)$ sorting is infeasible for n over 5000

A different strategy?

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

Combining sorted lists

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

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

* Sort `A[0:n//2]`
* Sort `A[n//2:n]`
* Merge sorted halves into `B[0:n]`
* 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

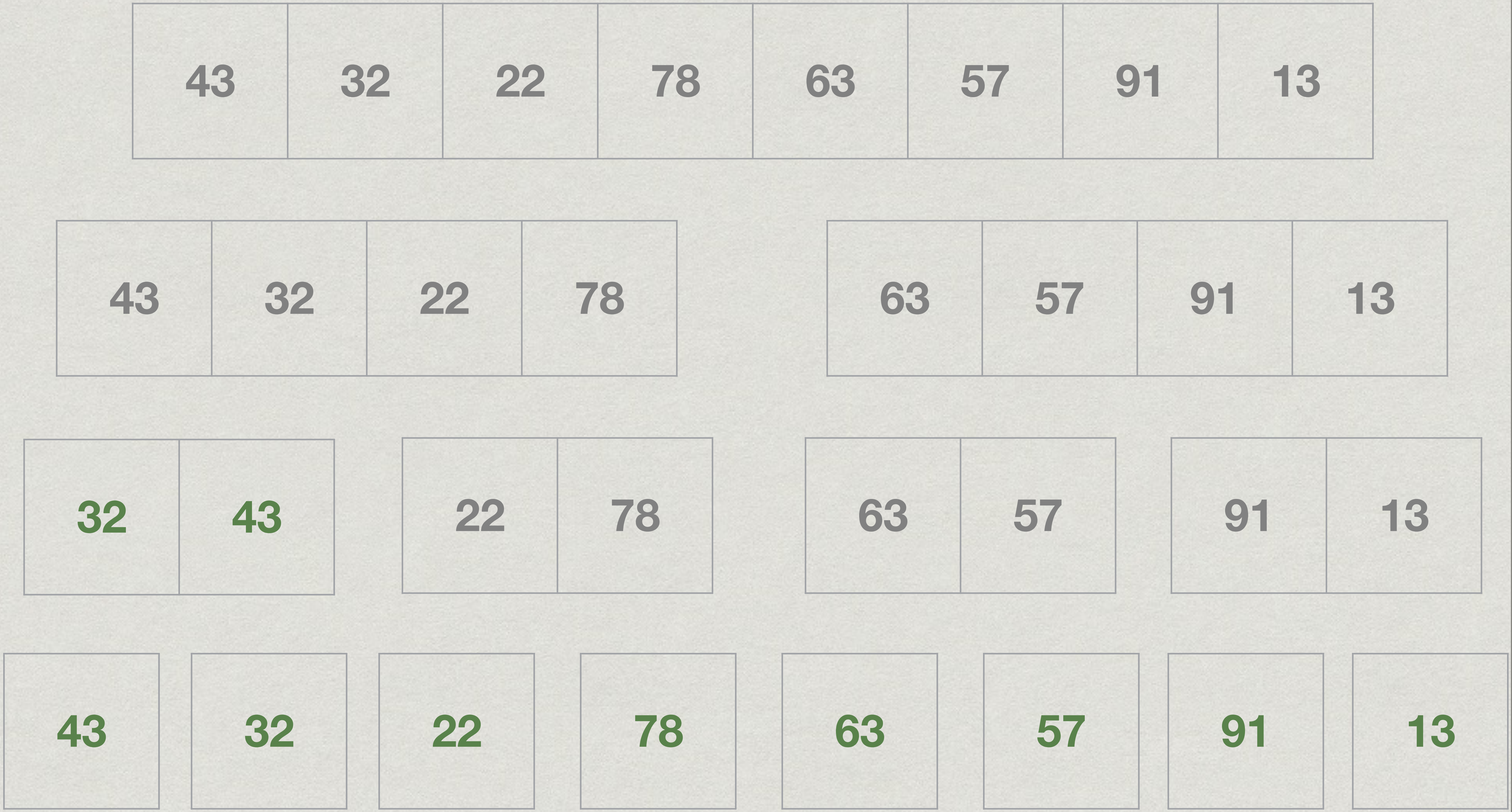
22	78

63	57

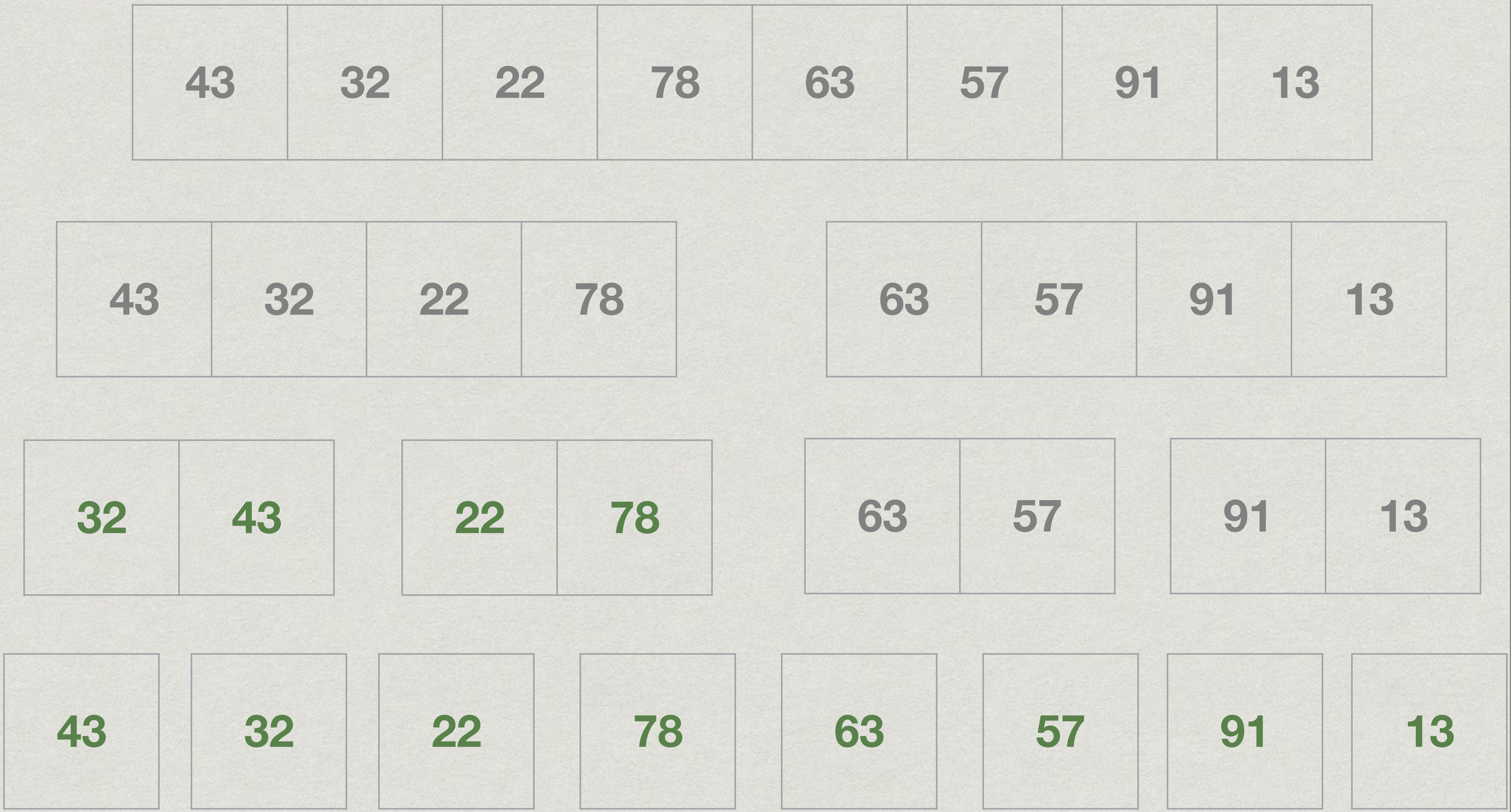
91	13

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

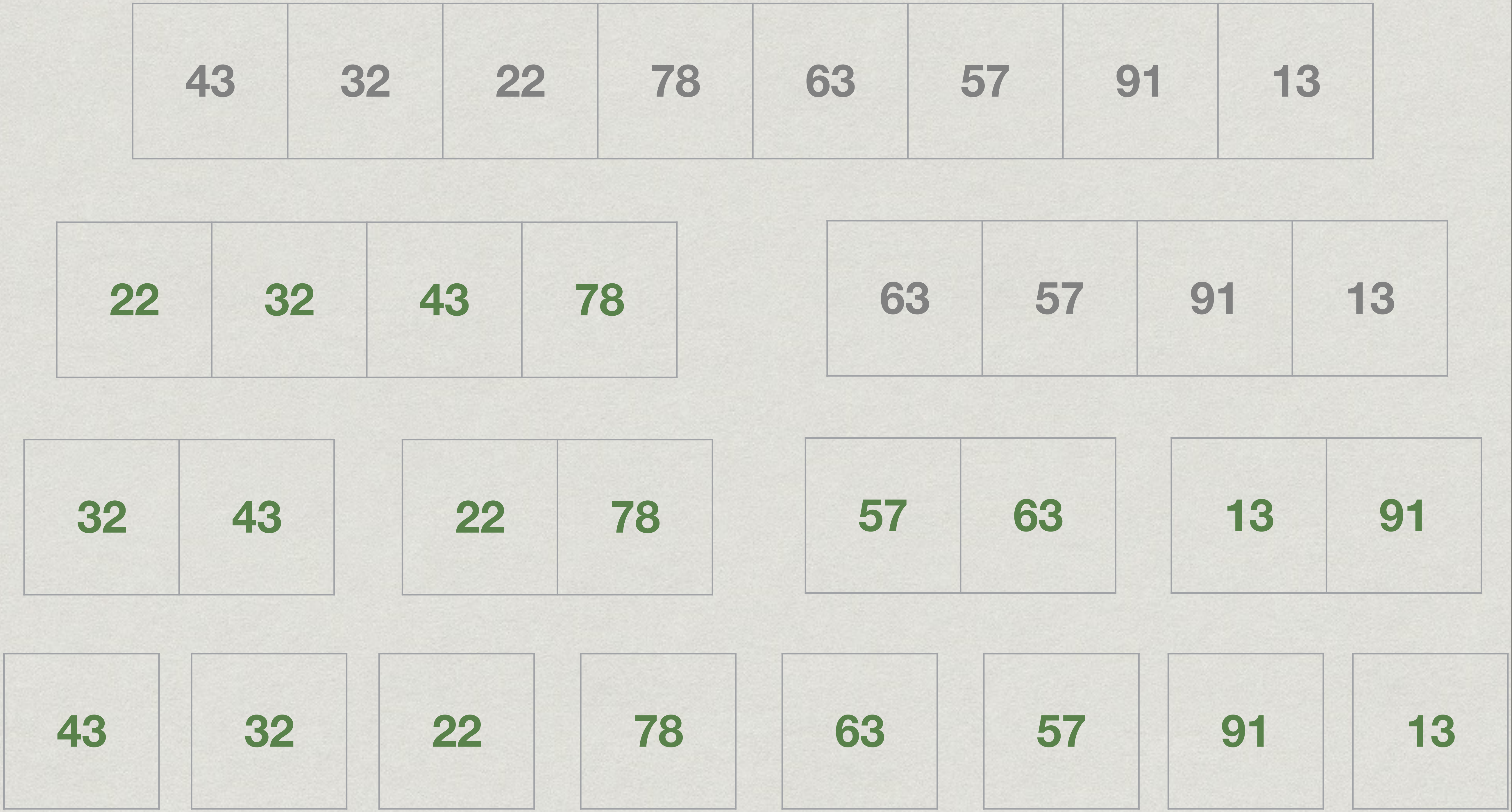
63

57

91

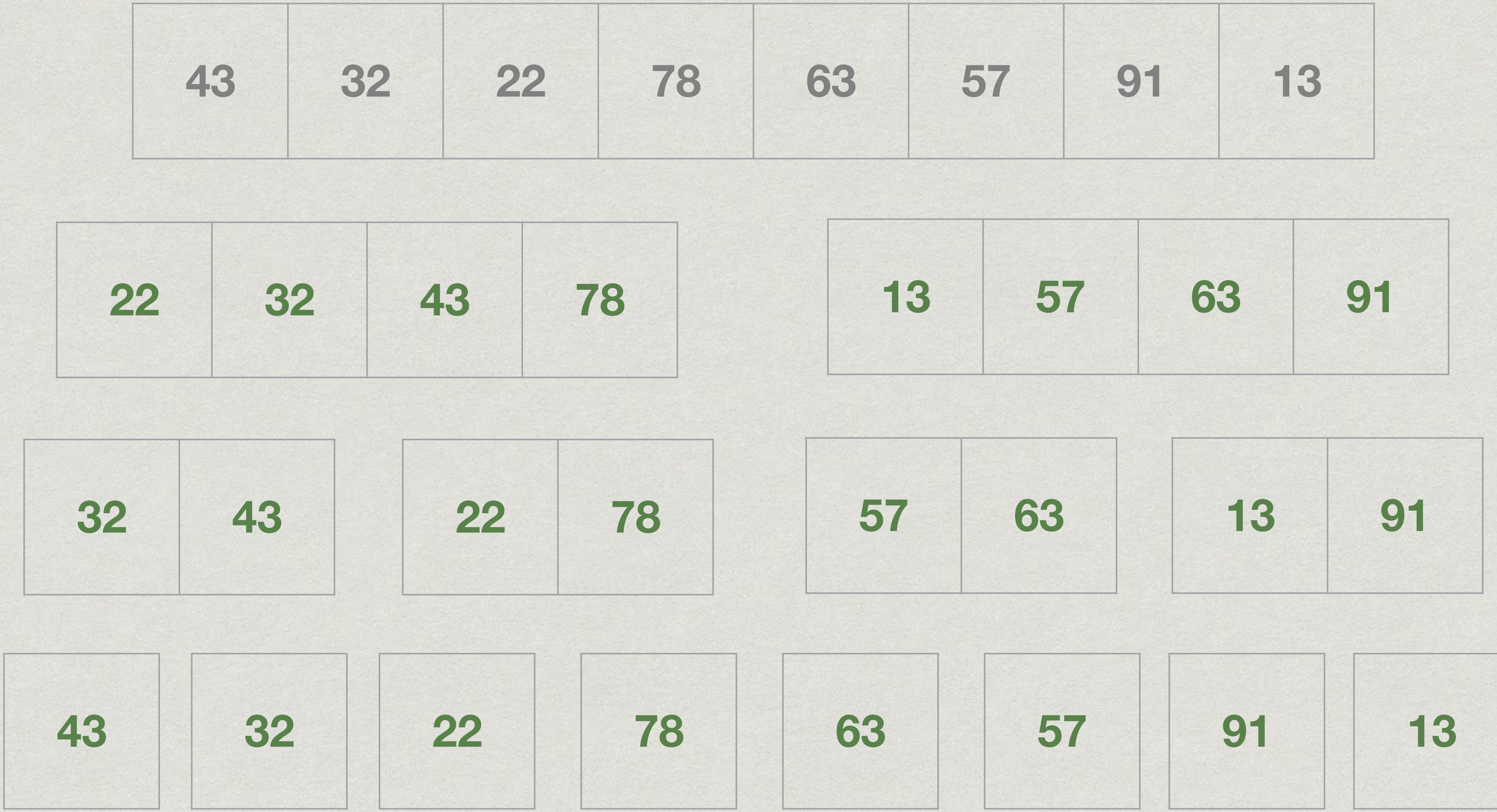
13

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

Divide and conquer

* Break up problem into disjoint parts
* Solve each part separately
* Combine the solutions efficiently

Merging sorted lists

Combine two sorted lists `A` and `B` into `C`

- If `A` is empty, copy `B` into `C`
- If `B` is empty, copy `A` into `C`
- Otherwise, compare first element of `A` and `B` and move the smaller of the two into `C`
- Repeat until all elements in `A` and `B` have been moved

Merging

```
def merge(A,B): # Merge A[0:m],B[0:n]
  (C,m,n) = ([],len(A),len(B))
  (i,j) = (0,0) # Current positions in A,B
  while i+j < m+n: # i+j is number of elements merged so far
    if i == m:  # Case 1: A is empty
      C.append(B[j])
      j = j+1
    elif j == n: # Case 2: B is empty
      C.append(A[i])
      i = i+1
    elif A[i] <= B[j]: # Case 3: Head of A is smaller
      C.append(A[i])
      i = i+1
    elif A[i] > B[j]: # Case 4: Head of B is smaller
      C.append(B[j])
      j = j+1
  return(C)
```

Merging, wrong

```
def mergewrong(A,B): # Merge A[0:m],B[0:n]
  (C,m,n) = ([],len(A),len(B))
  (i,j) = (0,0) # Current positions in A,B
  while i+j < m+n:
  # i+j is number of elements merged so far
    # Combine Case 1, Case 4
    if i == m or A[i] > B[j]:
      C.append(B[j])
      j = j+1
    # Combine Case 2, Case 3:
    elif j == n or A[i] <= B[j}:
      C.append(A[i])
      i = i+1
  return(C)
```

Merge Sort

To sort `A[0:n]` into `B[0:n]`

- If n is 1, nothing to be done
- Otherwise
  - Sort `A[0:n//2]` into `L` (left)
  - Sort `A[n//2:n]` into `R` (right)
  - Merge `L` and `R` into `B`

Merge Sort

```
def mergesort(A,left,right):
  # Sort the slice A[left:right]

  if right - left <= 1: # Base case
    return(A[left:right])

  if right - left > 1:  # Recursive call

    mid = (left+right)//2

    L = mergesort(A,left,mid)
    R = mergesort(A,mid,right)

    return(merge(L,R))
```

NPTEL MOOC

PROGRAMMING, DATA STRUCTURES AND ALGORITHMS IN PYTHON

Week 4, Lecture 2

Madhavan Mukund, Chennai Mathematical Institute
http://www.cmi.ac.in/~madhavan

Merge sorted lists

* Given two sorted lists `A` and `B`, combine into a sorted list `C`
  * Compare first element of `A` and `B`
  * Move it into `C`
  * Repeat until all elements in `A` and `B` are over
* Merging `A` and `B`

Analysis of Merge

How much time does Merge take?

- Merge `A` of size m, `B` of size n into `C`
- In each iteration, we add one element to `C`
  - Size of `C` is $m+n$
  - $m+n \le 2 \max(m,n)$
- Hence $O(\max(m,n)) = O(n)$ if $m \approx n$

Merge Sort

To sort `A[0:n]` into `B[0:n]`

- If n is 1, nothing to be done
- Otherwise
  - Sort `A[0:n//2]` into `L` (left)
  - Sort `A[n//2:n]` into `R` (right)
  - Merge `L` and `R` into `B`

Analysis of Merge Sort …

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

Analysis of Merge Sort …

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

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

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

  …

  $= 2^j T(n/2^j) + jn$
- When $j = \log n$, $n/2^j = 1$, so $T(n/2^j) = 1$
  - $\log n$ means $\log_2 n$ unless otherwise specified!
- $T(n) = 2^j T(n/2^j) + jn = 2^{\log n} + (\log n) n = n + n \log n = O(n \log n)$

Variations on merge

- Union of two sorted lists (discard duplicates)
  - While `A[i] == B[j]`, increment `j`
  - Append A[i] to C and increment `i`
- Intersection of two sorted lists
  - If `A[i] < B[j]`, increment `i`
  - If `B[j] < A[i]`, increment `j`
  - If `A[i] == B[j]`
    - While `A[i] == B[j]`, increment `j`
    - Append A[i] to C and increment `i`
- Exercise: List difference: elements in A but not in `B`

Merge Sort: Shortcomings

- Merging A and B creates a new array C
  - No obvious way to efficiently merge in place
- Extra storage can be costly
- Inherently recursive
  - Recursive call and return are expensive

NPTEL MOOC

PROGRAMMING, DATA STRUCTURES AND ALGORITHMS IN PYTHON

Week 4, Lecture 3

Madhavan Mukund, Chennai Mathematical Institute
http://www.cmi.ac.in/~madhavan

Merge Sort: Shortcomings

- Merging `A` and `B` creates a new array `C`
  - No obvious way to efficiently merge in place
- Extra storage can be costly
- Inherently recursive
  - Recursive call and return are expensive

Alternative approach

- Extra space is required to merge
- Merging happens because elements in left half must move right and vice versa
- Can we divide so that everything to the left is smaller than everything to the right?
  - No need to merge!

Divide and conquer without merging

- Suppose the median value in A is m
- Move all values $\leq m$ to left half of A
  - Right half has values $> m$
  - This shifting can be done in place, in time $O(n)$
- Recursively sort left and right halves
- A is now sorted! No need to merge
  - $T(n) = 2T(n/2) + n = O(n \log n)$

Divide and conquer without 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 `A` — pivot
  * Split `A` with respect to this pivot element

Quicksort

- Choose a pivot element
  - Typically the first value in the array
- Partition `A` 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

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: Partitioning

43	32	22	78	63	57	91	13

Quicksort: Partitioning

43	32	22	78	63	57	91	13

Quicksort: Partitioning

43	32	22	78	63	57	91	13

Quicksort: Partitioning

43	32	22	78	63	57	91	13

Quicksort: Partitioning

43	32	22	78	63	57	91	13

Quicksort: Partitioning

43	32	22	78	63	57	91	13

Quicksort: Partitioning

43	32	22	78	63	57	91	13

Quicksort: Partitioning

43	32	22	78	63	57	91	13

Quicksort: Partitioning

43	32	22	78	63	57	91	13

Quicksort: Partitioning

43	32	22	78	63	57	91	13

Quicksort: Partitioning

43	32	22	13	63	57	91	78

Quicksort: Partitioning

13	32	22	43	63	57	91	78

Quicksort in Python

```
def Quicksort(A,l,r): # Sort A[l:r]
  if r - l <= 1:  # Base case
     return ()

  # Partition with respect to pivot, a[l]
  yellow = l+1

  for green in range(l+1,r):
     if A[green] <= A[l]:
       (A[yellow],A[green]) = (A[green],A[yellow])
       yellow = yellow + 1

  # Move pivot into place
  (A[l],A[yellow-1]) = (A[yellow-1],A[l])

  Quicksort(A,l,yellow-1)  # Recursive calls
  Quicksort(A,yellow,r)
```

NPTEL MOOC

PROGRAMMING, DATA STRUCTURES AND ALGORITHMS IN PYTHON

Week 4, Lecture 4

Madhavan Mukund, Chennai Mathematical Institute
http://www.cmi.ac.in/~madhavan

Quicksort

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

Quicksort in Python

```
def Quicksort(A,l,r): # Sort A[l:r]
  if r - l <= 1:  # Base case
    return ()

  # Partition with respect to pivot, a[l]
  yellow = l+1

  for green in range(l+1,r):
    if A[green] <= A[l]:
      (A[yellow],A[green]) = (A[green],A[yellow])
      yellow = yellow + 1

  # Move pivot into place
  (A[l],A[yellow-1]) = (A[yellow-1],A[l])

  Quicksort(A,l,yellow-1)  # Recursive calls
  Quicksort(A,yellow,r)
```

Analysis of Quicksort

Worst case

- Pivot is either maximum or minimum
  - One partition is empty
  - Other has 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)
    - All permutations of n values, each equally likely
    - Average running time across all permutations
- Sorting is a rare example where average case can be computed

Quicksort: randomization

- Worst case arises because of fixed choice of pivot
  - We chose the first element
  - For any fixed strategy (last element, midpoint), can work backwards to construct $O(n^2)$ worst case
- Instead, choose pivot randomly
  - Pick any index in `range(0,n)` with uniform probability
- Expected running time is again $O(n \log n)$

Quicksort in practice

- In practice, Quicksort is very fast
  - Typically the default algorithm for in-built sort functions
    - Spreadsheets
    - Built in sort function in programming languages

Stable sorting

- Sorting on multiple criteria
- Assume students are listed in alphabetical order
- Now sort students by marks
  - After sorting, are students with equal marks still in alphabetical order?
- Stability is crucial in applications like spreadsheets
  - Sorting column B should not disturb previous sort on column A

Stable sorting …

- Quicksort, as described, is not stable
  - Swap operation during partitioning disturbs original order
- Merge sort is stable if we merge carefully
  - Do not allow elements from right to overtake elements from left
  - Favour left list when breaking ties

NPTEL MOOC

PROGRAMMING, DATA STRUCTURES AND ALGORITHMS IN PYTHON

Week 4, Lecture 5

Madhavan Mukund, Chennai Mathematical Institute
http://www.cmi.ac.in/~madhavan

Tuples

- Simultaneous assignments

  `(age,name,primes) = (23,"Kamal",[2,3,5])`

- Can assign a “tuple” of values to a name

  ```
  point = (3.5,4.8)
  date = (16,7,2013)
  ```

- Extract positions, slices

  ```
  xcoordinate = point[0]
  monthyear = date[1:]
  ```

- Tuples are immutable

  `date[1] = 8` is an error

Generalizing lists

- `l = [13, 46, 0, 25, 72]`
- View `l` as a function, associating values to positions
  - `l : {0,1,..,4} ⟶ integers`
  - `l(0) = 13, l(4) = 72`
- `0,1,..,4` are keys
- `l[0],l[1],..,l[4]` are corresponding values

Dictionaries

- Allow keys other than `range(0,n)`
- Key could be a string

```
test1["Dhawan"] = 84
test1["Pujara"] = 16
test1["Kohli"] = 200
```

- Python dictionary
  - Any immutable value can be a key
  - Can update dictionaries in place —mutable, like lists

Dictionaries

* Empty dictionary is `{}`, not `[]`
  * Initialization: `test1 = {}`
  * Note: `test1 = []` is empty list, `test1 = ()` is empty tuple
* Keys can be any immutable values
  * `int, float, bool, string, tuple`
  * But not lists, or dictionaries

Dictionaries

- Can nest dictionaries

```
score["Test1"]["Dhawan"] = 84
score["Test1"]["Kohli"] = 200
score["Test2"]["Dhawan"] = 27
```

- Directly assign values to a dictionary

```
score = {"Dhawan":84, "Kohli":200}
score = {"Test1":{"Dhawan":84,
  "Kohli":200}, "Test2":{"Dhawan":50}}
```

Operating on dictionaries

- `d.keys()` returns sequence of keys of dictionary `d`

```
for k in d.keys():
  # Process d[k]
```

- `d.keys()` is not in any predictable order

```
for k in sorted(d.keys()):
  # Process d[k]
```

- `sorted(l)` returns sorted copy of `l`, `l.sort()` sorts `l` in place

- `d.keys()` is not a list — use `list(d.keys())`

Operating on dictionaries

- Similarly, `d.values()` is sequence of values in `d`

```
total = 0
for s in test1.values():
  total = total + test1
```

- Test for key using `in`, like list membership

```
for n in ["Dhawan","Kohli"]:
  total[n] = 0
  for match in score.keys():
    if n in score[match].keys():
      total[n] = total[n] + score[match][n]
```

Dictionaries vs lists

- Assigning to an unknown key inserts an entry

```
d = {}
d[0] = 7  # No problem, d == {0:7}
```

- … unlike a list

```
l = []
l[0] = 7  # IndexError!
```

Summary

* Dictionaries allow a flexible association of values to keys
  * Keys must be immutable values
* Structure of dictionary is internally optimized for key-based lookup
  * Use `sorted(d.keys())` to retrieve keys in predictable order
* Extremely useful for manipulating information from text files, tables … — use column headings as keys

NPTEL MOOC

PROGRAMMING, DATA STRUCTURES AND ALGORITHMS IN PYTHON

Week 4, Lecture 6

Madhavan Mukund, Chennai Mathematical Institute
http://www.cmi.ac.in/~madhavan

Passing values to functions

- Argument value is substituted for name

```
def power(x,n):
  ans = 1
  for i in range(0,n):
    ans = ans*x
  return(ans)
```

```
power(3,5)

x = 3
n = 5
ans = 1
for i in range..
```

- Like an implicit assignment statement

Pass arguments by name

```
def power(x,n):
  ans = 1
  for i in range(0,n):
    ans = ans*x
  return(ans)
```

- Call `power(n=5,x=4)`

Default arguments

- Recall `int(s)` that converts string to integer
  - `int("76")` is `76`
  - `int("A5")` generates an error
- Actually `int(s,b)` takes two arguments, string `s` and base `b`
  - `b` has default value `10`
  - `int("A5",16)` is `165` (10 x 16 + 5)

Default arguments

```
def int(s,b=10):
  . . .
```

- Default value is provided in function definition
- If parameter is omitted, default value is used
  - Default value must be available at definition time
  - `def Quicksort(A,l=0,r=len(A)):` does not work

Default arguments

```
def f(a,b,c=14,d=22):
    . . .
```

- `f(13,12)` is interpreted as `f(13,12,14,22)`
- `f(13,12,16)` is interpreted as `f(13,12,16,22)`
- Default values are identified by position, must come at the end
  - Order is important

Function definitions

- `def` associates a function body with a name
- Flexible, like other value assignments to name
- Definition can be conditional

```
if condition:
  def f(a,b,c):
    . . .
else:
  def f(a,b,c):
    . . .
```

Function definitions

- Can assign a function to a new name

```
def f(a,b,c):
    . . .

g = f
```

- Now `g` is another name for `f`

Can pass functions

- Apply `f` to `x` `n` times

```
def apply(f,x,n):
  res = x
  for i in range(n):
    res = f(res)
  return(res)
```

```
def square(x):
  return(x*x)

apply(square,5,2)

square(square(5))

625
```

Passing functions

- Useful for customizing functions such as sort
- Define `cmp(x,y)` that returns `-1` if `x < y`, `0` if `x == y` and `1` if `x > y`
  - `cmp("aab","ab")` is `-1` in dictionary order
  - `cmp("aab","ab”)` is `1` if we compare by length
- `def sortfunction(l,cmpfn=defaultcmpfn):`

Summary

- Function definitions behave like other assignments of values to names
- Can reassign a new definition, define conditionally …
- Can pass function names to other functions

NPTEL MOOC

PROGRAMMING, DATA STRUCTURES AND ALGORITHMS IN PYTHON

Week 4, Lecture 7

Madhavan Mukund, Chennai Mathematical Institute
http://www.cmi.ac.in/~madhavan

Operating on lists

- Update an entire list

```
for x in l:
  x = f(x)
```

- Define a function to do this in general

```
def applylist(f,l):
  for x in l:
    x = f(x)
```

Built in function `map()`

- `map(f,l)` applies `f` to each element of `l`
- Output of `map(f,l)` is not a list!
  - Use `list(map(f,l))` to get a list
  - Can be used directly in a `for` loop

    ```
    for i in map(f,l):
    ```

  - Like `range(i,j)`, `d.keys()`

Selecting a sublist

- Extract list of primes from list `numberlist`

```
primelist = []
for i in numberlist:
  if isprime(i):
    primelist.append(i)
return(primelist)
```

Selecting a sublist

- In general

```
def select(property,l):
  sublist = []
  for x in l:
    if property(x):
      sublist.append(x)
  return(sublist)
```

- Note that `property` is a function that returns `True` or `False` for each element

Built in function `filter()`

- `filter(p,l)` checks `p` for each element of `l`
- Output is sublist of values that satisfy `p`

Combining map and filter

- Squares of even numbers from 0 to 99

```
list(map(square,filter(iseven,range(100)))

def square(x):
  return(x*x)

def iseven(x):
  return(x%2 == 0)
```

List comprehension

- Pythagorean triple: $x^2 + y^2 = z^2$
- All Pythagorean triples (x,y,z) with values below n

$$\{ (x,y,z) \mid 1 \leq x,y,z \leq n,\ x^2 + y^2 = z^2 \}$$

- In set theory, this is called set comprehension
    - Building a new set from existing sets
- Extend to lists

List comprehension

- Squares of even numbers below 100

```
[square(x) for i in range(100) if iseven(x)]
```

map generator filter

Multiple generators

- Pythagorean triples with x,y,z below 100

```
[(x,y,z) for x in range(100)
          for y in range(100)
           for z in range(100)
            if x*x + y*y == z*z]
```

- Order of x,y,z is like nested for loop

```
for x in range(100):
  for y in range(100):
    for z in range(100):
```

Multiple generators

- Later generators can depend on earlier ones
- Pythagorean triples with x,y,z below 100, no duplicates

```
[(x,y,z) for x in range(100)
          for y in range(x,100)
           for z in range(y,100)
            if x*x + y*y == z*z]
```

Useful for initialising lists

- Initialise a 4 x 3 matrix
  - 4 rows, 3 columns
  - Stored row-wise

```
l = [ [ 0 for i in range(3) ]
         for j in range(4)]
```

Warning

- What's happening here?

```
>>> zerolist = [ 0 for i in range(3) ]
>>> l = [ zerolist for j in range(4) ]

>>> l[1][1] = 7

>>> l
[[0,7,0],[0,7,0],[0,7,0],[0,7,0]]
```

- Each row in `l` points to same list `zerolist`

Summary

- `map` and `filter` are useful functions to manipulate lists
- List comprehension provides a useful notation for combining `map` and `filter`