Quick Sort

Divide and Conquer inverted?

If a problem is very simple, solve it in a single step.
If a problem is too complex to solve in a single step,
  divide it into multiple pieces
  solve the individual pieces
  combine the pieces together to get a solution

Merge sort divides really quickly, but then does all its work in the *combination* phase.

- for the bottom-up version, divide takes 0 time

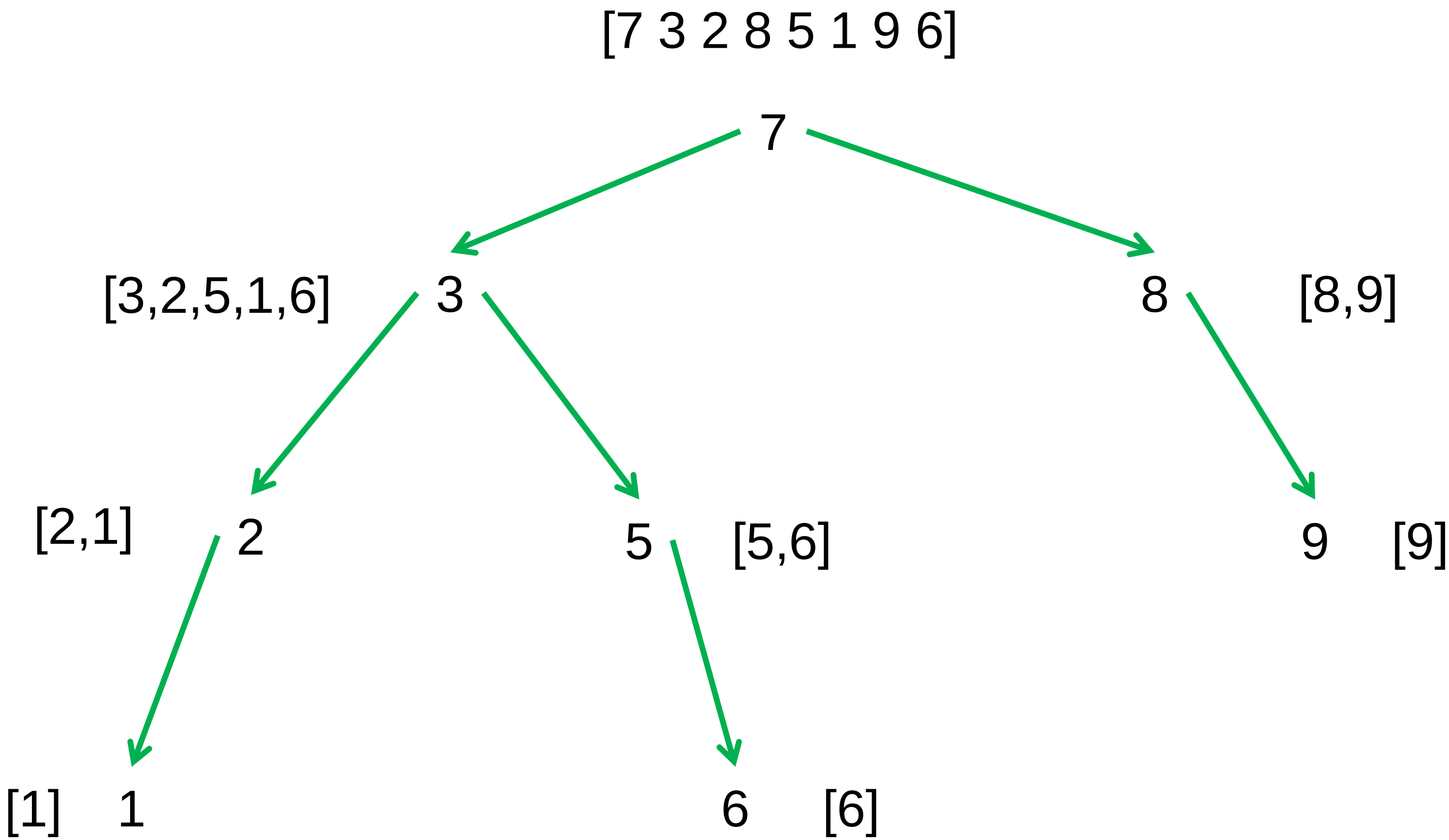
What happens if we put all the work into clever *dividing*, with the aim of having a really fast combination phase?

Think about building a binary search tree from a list, and then doing an in-order traversal to create the sorted list.

Building the tree:
- best case is O(n log n)
- worst case O(n^2)

In-order traversal is Θ(n)

```
Make the first elt of the list the root of a tree
for all elts in the list
   if they are less than root,
      put them in a left list
  else
      put them in a right list
if the left list is not empty,
   build the root's left sub tree from it
if the right list is not empty,
   build the root's right sub tree from it
```



Rough analysis:
Building each level i in the tree required at most n-1 comparisons, and n-i O(1) operations.
There are at most n levels, so $O(n^2)$ worst case.
At best, there are log n levels, so O(n log n) in best case .
Traversing the final tree is O(n).

Quicksort applies this procedure, but without actually building the tree.

For doubly-linked lists:

```
pseudocode quicksort(start, end):
   if start.next != end and start != end:
      pivot = start
      node = pivot.next
      while node is not the end
          nextnode = node.next
          if node.elt < pivot.elt
              move node in front of pivot
              if first move
                 start = node
          node = nextnode
      quicksort(start, pivot)
      quicksort(pivot.next, end)
```

7-3-2-8-5-1-9-6-N

3-2-5-1-6-7-8-9-N

2-1-3-5-6-7-8-9-N

1-2-3-5-6-7-8-9-N

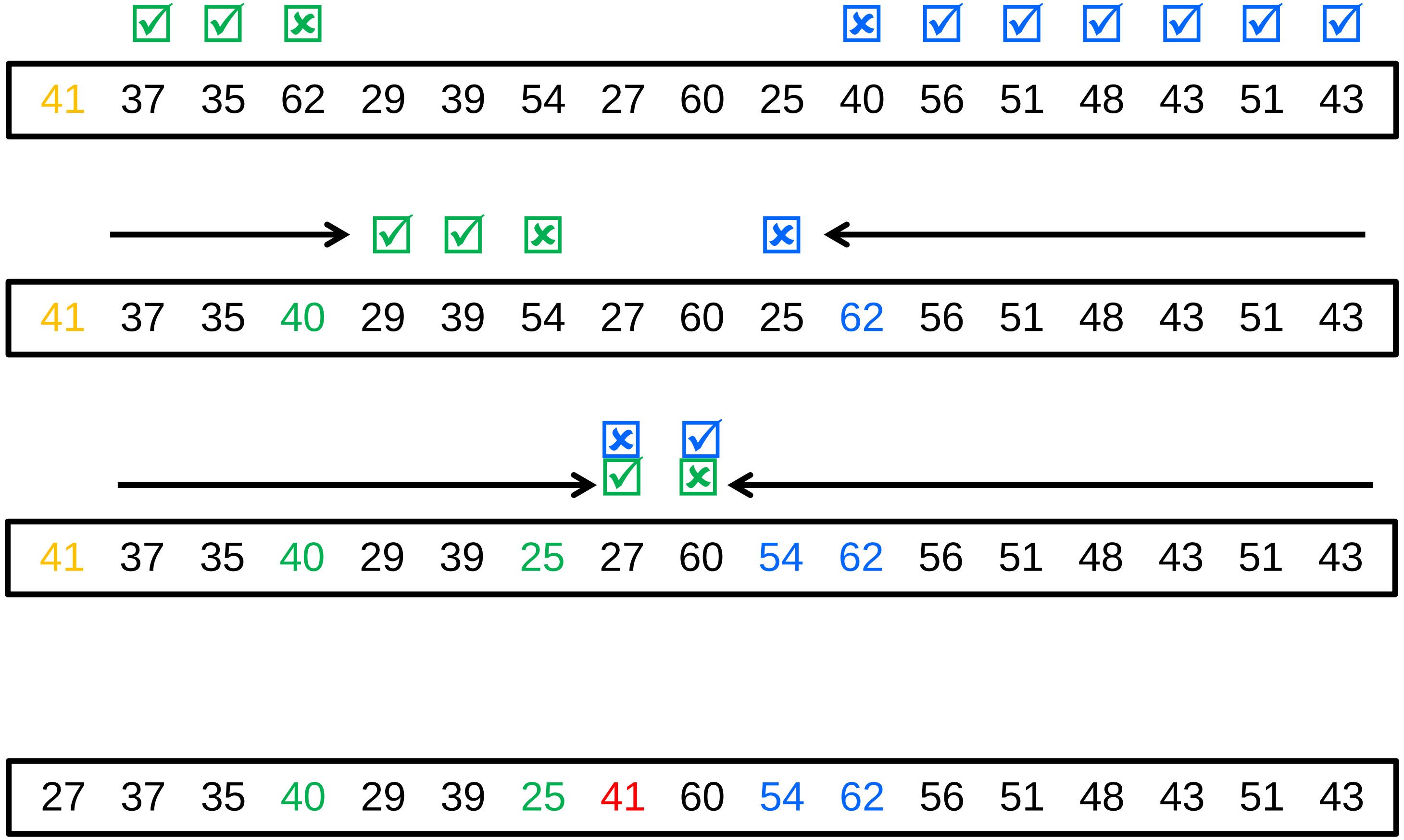
1-2-3-5-6-7-8-9-N

Can we implement quicksort on arrays, in-place?
We need to avoid shuffling elements along the array

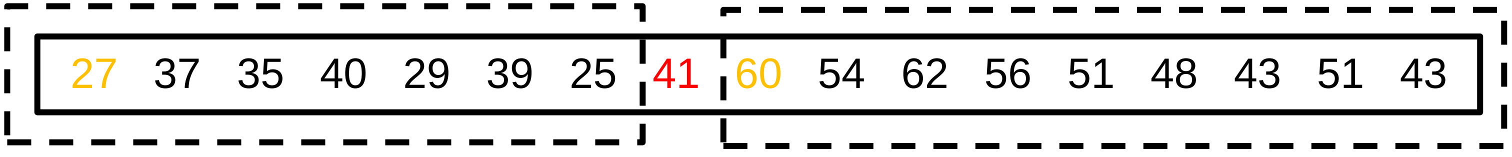
41	37	35	62	29	39	54	27	60	25	40	56	51	48	43	51	43

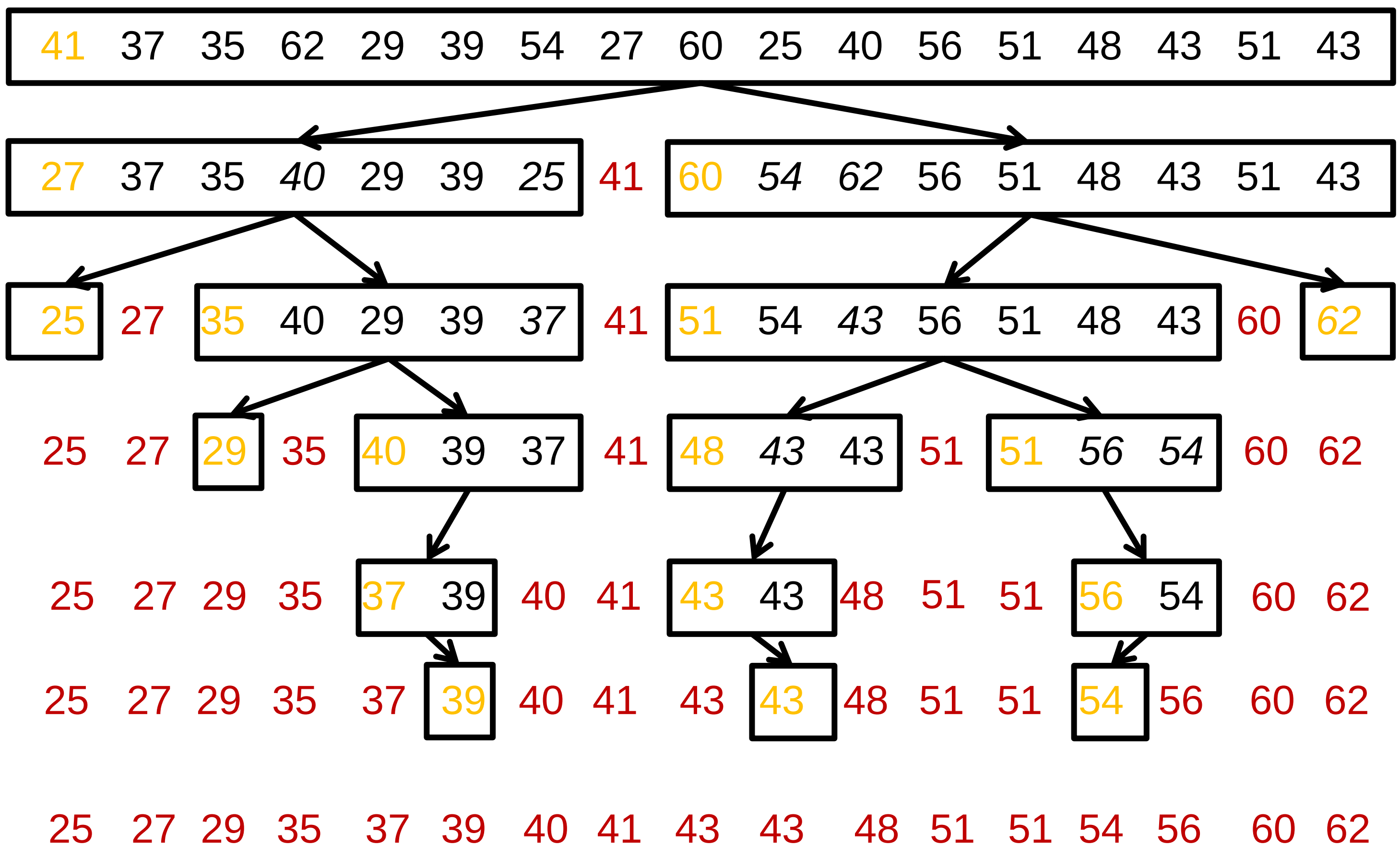
When dividing, search from both ends, and swap positions of any elements that are on the wrong side of where pivot will go.

```
pseudocode sort(list, pivot, end)
  while searches not crossed
     search right from pivot for a bigger item
     search left from end for a smaller item
     If searches not crossed
         swap items
  swap pivot with small item
  sort(list, small, pivot)
  sort(list, pivot+1, end)
```



One iteration done. Now repeat on the sublist before 41, and the sublist after.

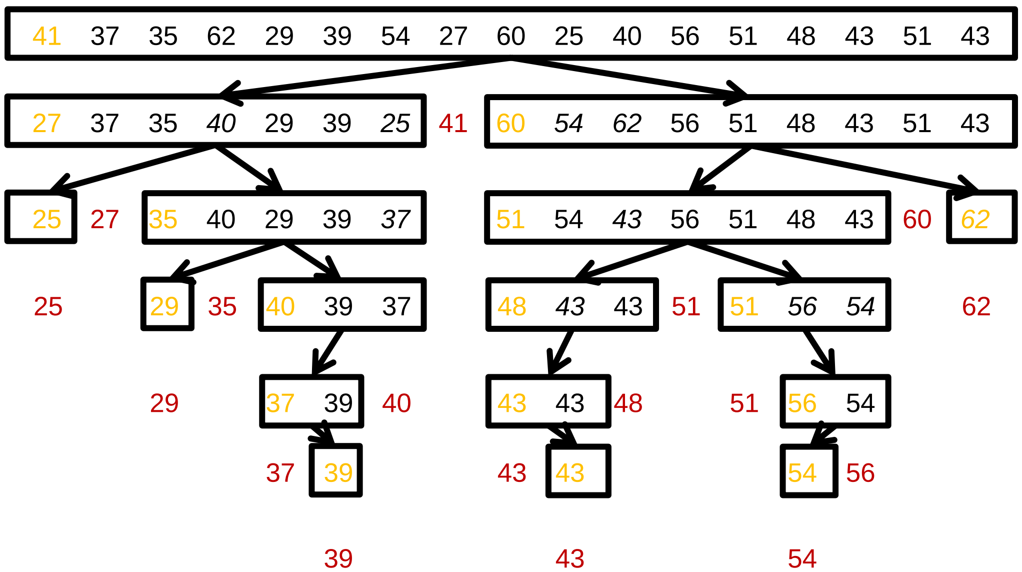



41 37 35 62 29 39 54 27 60 25 40 56 51 48 43 51 43
27 37 35 40 29 39 25 41 60 54 62 56 51 48 43 51 43
25 27 35 40 29 39 37 41 51 54 43 56 51 48 43 60 62
25 27 29 35 40 39 37 41 48 43 43 51 51 56 54 60 62
25 27 29 35 37 39 40 41 43 43 48 51 51 56 54 60 62
25 27 29 35 37 39 40 41 43 43 48 51 51 54 56 60 62
25 27 29 35 37 39 40 41 43 43 48 51 51 54 56 60 62

```
def _quicksort(mylist, first, last):
    #sort elements of mylist from first up to last
     if last > first:
         pivot =  mylist[first]
         f = first + 1
         b = last
         while f <= b:
             while f <= b and mylist[f] <= pivot:
                 f += 1
             while f <= b and mylist[b] >= pivot:
                 b -= 1
             if f < b:
                 mylist[f], mylist[b] = mylist[b], mylist[f]
                 f += 1
                 b -= 1
         mylist[b], mylist[first] = mylist[first], mylist[b]
         _quicksort(mylist, first, b-1)
         _quicksort(mylist, b+1, last)
```

Quicksort:

12 21 16 9 18 5 10 7



Analysis:

Each level of the tree takes at most *n* comparisons, and at most *n*/2 swaps.

What happens with

2 3 5 6 8 9 10 ?

Worst case depth of the tree is *n*
Then the comparisons would be
n+(n-1)+(n-2)+(n-3)+...+1
which means quicksort has
worst case $O(n^2)$ time complexity

We know that a balanced tree has depth log n, when each node has equal numbers of descendants on left and right.

So if we could choose a pivot each time that splits its sublist into two equal parts, our quicksort tree would also be log n depth, and the runtime would be O(n log n).

The value we are looking for is the *median*. Could we search for it in each sublist before sorting?

There is an algorithm ('median of medians") to find the median of an unsorted list in time O(n), but it is complex.

What if we chose a random value from the list as the pivot each time?

- swap that value with the first value, then continue as in the existing algorithm ...

It can be shown that the *expected* running time for a random pivot selection is O(n log n).

The worst case will still be $O(n^2)$, since it is always possible that the random choice chooses values close to the min or max.

A simpler solution is to create a random shuffle of the top level list, and then call the existing algorithm.

```
def quicksort(mylist):
    n = len(mylist)
    for i in range(len(mylist)):
        j = random.randint(0, n-1)
        mylist[i], mylist[j] = mylist[j], mylist[i]
    _quicksort(mylist, 0, n-1)
```

Adds *n* calls to randint, and *n* swaps before we start.
This doesn't change the *expected* runtime of O(n log n), or the worst case of O(n^2)

In practice, even with this random shuffle, quicksort runs a little faster than mergesort or heap sort.

Next Lecture

Other sort methods