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

Could we switch it around, and put all the work into clever *dividing*, aiming for a really fast combination phase?

Keep splitting the list in two ...

Our aim will be to rearrange the list into two parts, so that we can treat each part independently. Then we will recurse on each part, and so on. Once we get down to lists of size 1, then we want to finish, with almost zero work to recombine.

What property should we try to enforce for the two parts of the list?

Suppose the initial list is: 5 9 2 4 8 1 3 6

All elements of the 1st part should be less than all elements in the 2nd part?

- how do we decide, efficiently, which items come before the split and which after?
- how do we move items around efficiently?
- can we get the split to be exactly halfway each time?

Partition by pivoting around a chosen value

Modified sub-list property:

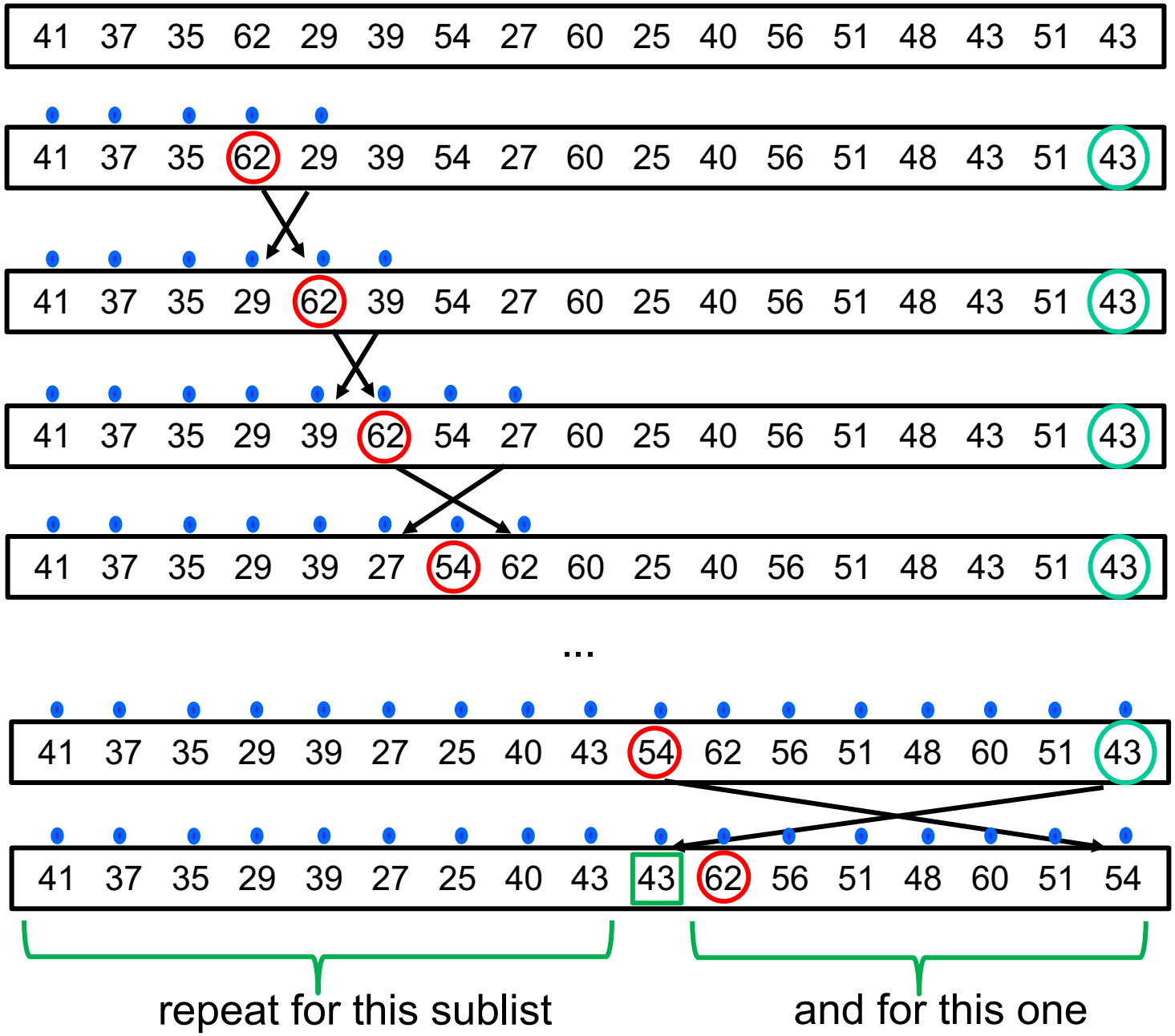
- pick an element of the list to be the *pivot*
- 1st part of list will be elements less than or equal to the pivot; then pivot; then 2nd part is those greater than pivot
- rearrange by swapping items that are in the wrong part ...

5 9 2 4 8 1 3 6

but we still have to work out the position for the pivot (so that we can identify items in the wrong part ...)

Partitioning the list (Lomuto)

- choose the last item of the (sub-) list as the pivot
- step through the list one cell at a time, remembering the earliest position of the items greater than the pivot
  - at all times, everything before that ‘earliest’ cell must be ≤ pivot;
- whenever we find an item ≤ than the pivot, swap it with the ‘earliest’ item, and advance the index for earliest cell by 1
  - at all times, everything before that ‘earliest’ cell must be ≤ pivot; everything between earliest cell and the current cell must be bigger than the pivot; items after current cell have not yet been checked
- after we process the last item, the partitioning is complete
  - the last item was the pivot, so it is swapped with ‘earliest’; everything before it is ≤, everything after it is >
- (and now we can recurse on the two sub-lists)


41 37 35 62 29 39 54 27 60 25 40 56 51 48 43 51 43
41 37 35 62 29 39 54 27 60 25 40 56 51 48 43 51 43
41 37 35 29 62 39 54 27 60 25 40 56 51 48 43 51 43
41 37 35 29 39 62 54 27 60 25 40 56 51 48 43 51 43
41 37 35 29 39 27 54 62 60 25 40 56 51 48 43 51 43
...
41 37 35 29 39 27 25 40 43 54 62 56 51 48 60 51 43
41 37 35 29 39 27 25 40 43 43 62 56 51 48 60 51 54
repeat for this sublist
and for this one

5 9 2 4 8 1 3 6

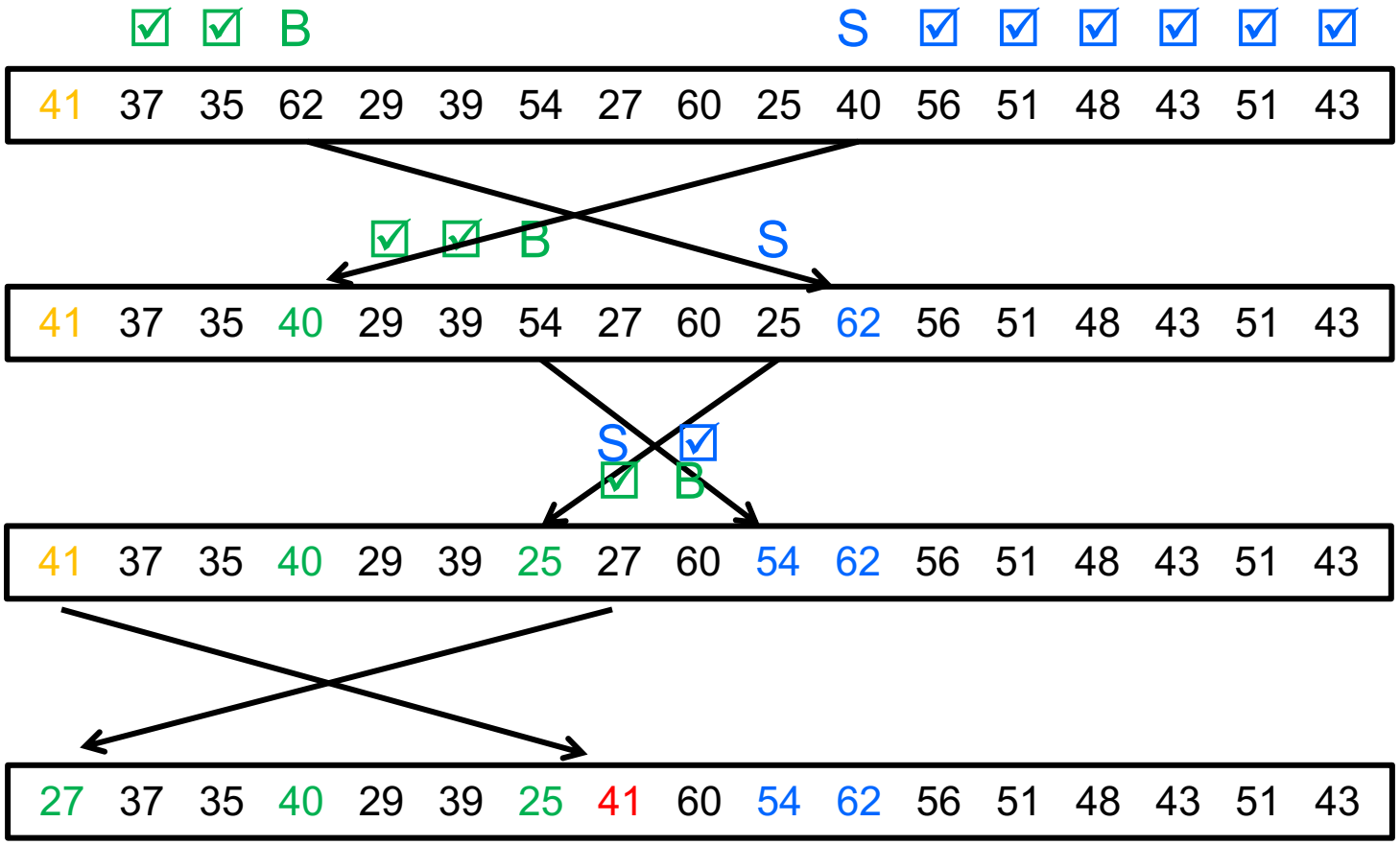
Partitioning the list (Hoare)

The original partitioning scheme

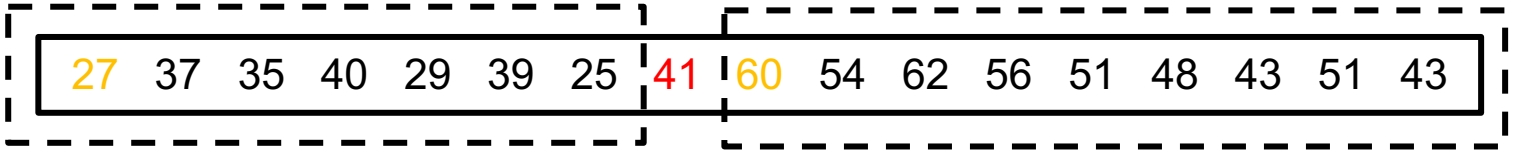
- Choose the *first* item of the (sub-) list as the pivot
- repeat
  - step through the list from the *left*, looking for the 1st item bigger than the pivot – record this cell as *bigger*
    - all to left of this are ≤ pivot, so pivot must end up in previous cell or later, but if it does go later, this item will have to move right
  - step through the list from the *right*, looking for the 1st item less than or equal to the pivot - record this cell as *smaller*
    - all to right of this are > pivot, so pivot must end up here or earlier, but if it does go earlier, this item will have to move left
  - if these two positions have not crossed (ie *bigger* is still before *smaller*), swap the two items

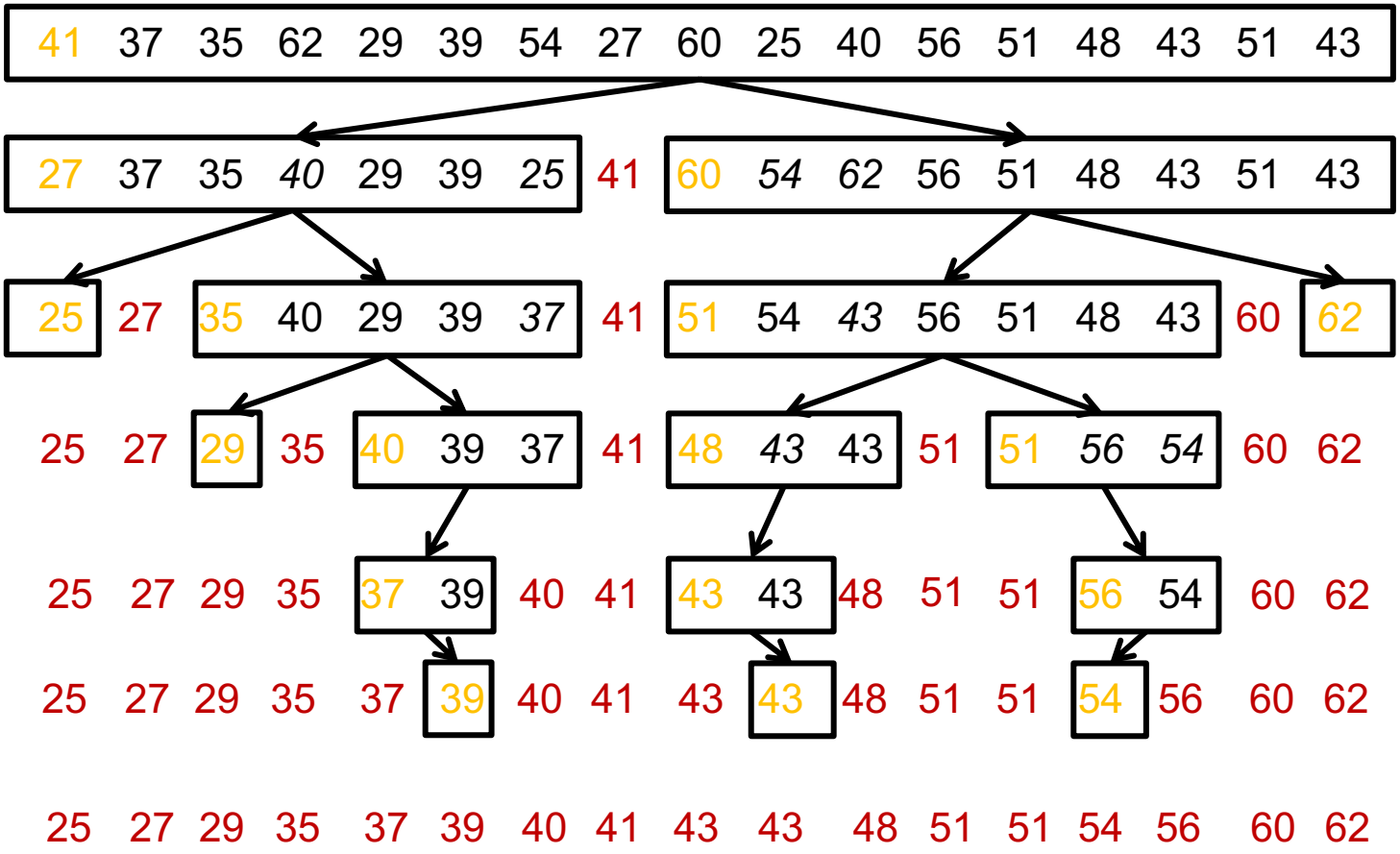
  until *bigger* index is greater than or equal to *smaller* index
- swap pivot with item in *smaller*
- now recurse on the two sublists

```
pseudocode sort(list, pivot, end)
   while searches not crossed
      search to right from pivot for a bigger item
      search to left from end for a smaller item
      if searches not crossed
          swap items
   swap pivot with most recently found 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.

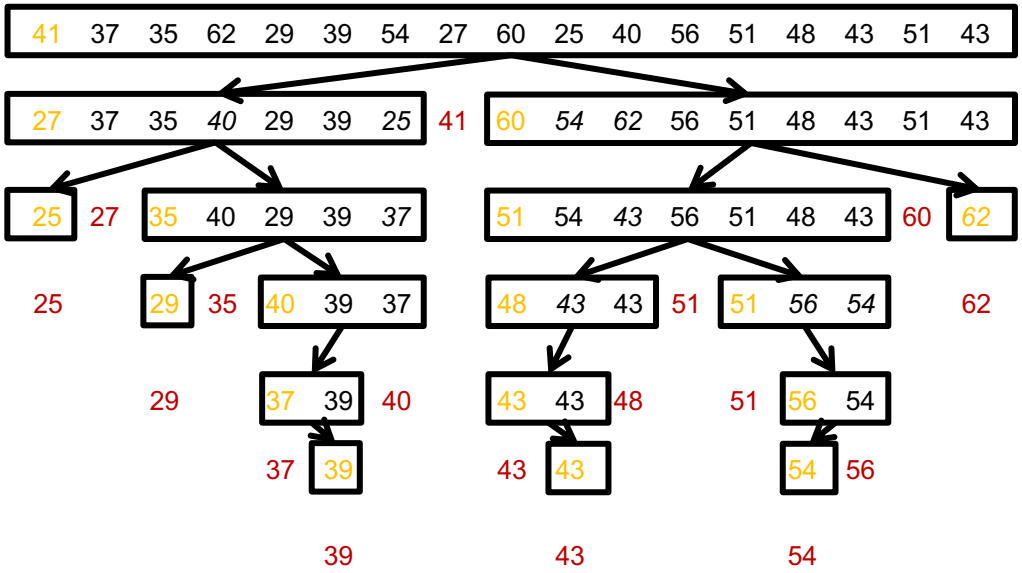



41 37 35 62 29 39 54 27 60 25 40 56 51 48 43 51 43
27 37 35 40 29 39 25 41 60 54 62 56 51 48 43 51 43
25 27 35 40 29 39 37 41 51 54 43 56 51 48 43 60 62
25 27 29 35 40 39 37 41 48 43 43 51 51 56 54 60 62
25 27 29 35 37 39 40 41 43 43 48 51 51 56 54 60 62
25 27 29 35 37 39 40 41 43 43 48 51 51 54 56 60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

Quicksort (Hoare):

12 21 16 9 18 5 10 7

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



Analysis:

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

Exercise: how many comparisons and swaps per level for Lomuto?

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

Exercise: what is the worst case for Lomuto? For what pattern in the lists?

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 (approximate) median of an unsorted list in time O(n), but it is complicated.

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 quicksort with 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.

In practice, randomising the pivot selection tends to be faster than using the median of medians 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 usually runs a little faster than mergesort or heapsort.

How could quicksort be applied to DLLs?

```
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

Next Lecture

The fundamental limits of comparison sorting

Other sort methods