Merge Sort

More sorting?

Heapsort had complexity O(n log n), and sorted in-place.
Is there anything else worth looking at?

- Are there algorithms with better worst cases complexity?
  - even if they have the same complexity, maybe the lower order terms are better?
- Are there algorithms with better average complexity?
- Do we need to worry about in-place sorting?
- Are there other problem-solving strategies we could look at?

Divide and Conquer

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

Typically implemented using multiple recursion

A general problem solving strategy used throughout computing

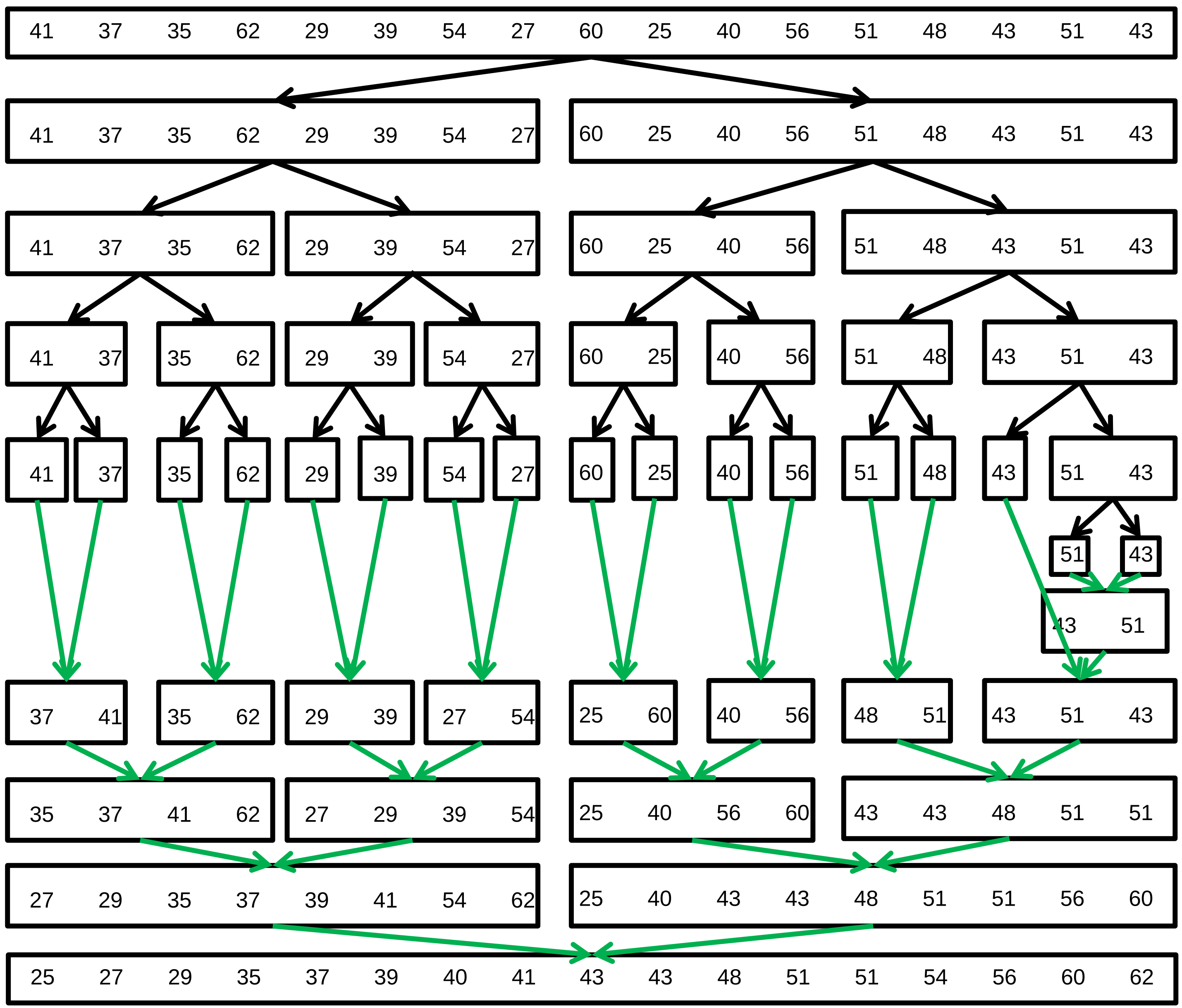
Divide and Conquer: sorting

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

Typically implemented using multiple recursion

Sorting a list:
  If an input list is of size 1 (or 0), do nothing.
  If an input list is of size 2 or more
    split it into two almost equal sublists   *#divide*
    sort the first sublist   *#conquer*
    sort the second sublist
    merge the two sublists into a combined list   *#combine*

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

```
def mergesort(mylist):
    n = len(mylist)
    if n > 1:
        list1 = mylist[:n//2]
        list2 = mylist[n//2:]
        mergesort(list1)
        mergesort(list2)
        merge(list1, list2, mylist)
```

Slicing creates a new list each time, so not in-place.
But it is difficult to write an in-place mergesort
without increasing the time complexity

Merge:

35	37	41	62	27	29	39	54

```
def merge(list1, list2, mylist):
    f1 = 0
    f2 = 0
    while f1 + f2 < len(mylist):
        if f1 == len(list1):
            mylist[f1+f2] = list2[f2]
            f2 += 1
        elif f2 == len(list2):
            mylist[f1+f2] = list1[f1]
            f1 += 1
        elif list2[f2] < list1[f1]:
            mylist[f1+f2] = list2[f2]
            f2 += 1
        else:
            mylist[f1+f2] = list1[f1]
            f1 += 1
```

Note: written for clarity. Repeated code is not a good idea, so should rewrite to require only 1 test in loop body

```
def merge(list1, list2, mylist):
    f1 = 0
    f2 = 0
    while f1 + f2 < len(mylist):
        if f1 == len(list1):
            mylist[f1+f2] = list2[f2]
            f2 += 1
        elif f2 == len(list2):
            mylist[f1+f2] = list1[f1]
            f1 += 1
        elif list2[f2] < list1[f1]:
            mylist[f1+f2] = list2[f2]
            f2 += 1
        else:
            mylist[f1+f2] = list1[f1]
            f1 += 1
```

Analysis: (|mylist| = n)

round the loop n times

inside the loop, at most 3 tests, 2 calls to len(.), and 2 assignments

So O(1) inside the loop.

So function has worst case time O(n)

Note: writing the result into the 3rd input list, so we do not occupy any extra space.

```
def mergesort(mylist):
    n = len(mylist)
    if n > 1:
        list1 = mylist[:n//2]
        list2 = mylist[n//2:]
        mergesort(list1)
        mergesort(list2)
        merge(list1, list2, mylist)
```

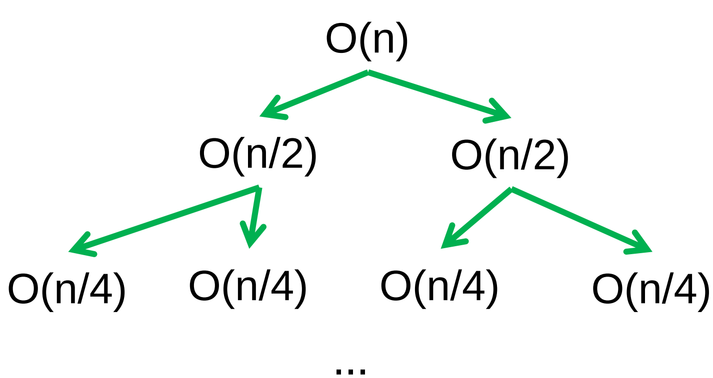
<u>Analysis:</u>

Each call (without recursion) takes:

- n assignments to create the slices
- O(n) for the merge function

So O(n)

Each recursive call is for a list of size n/2, and so takes O(n/2), etc.



So we have O(n) at each level in the call tree.

The depth of the tree is either $\log_2(n)$ or $\log_2(n) + 1$

So O(n log n) in total

Each call creates new smaller lists, so space complexity (of this implementation) is same as time – O(n log n)

Alternative Analysis: recurrence equations

The base case is $O(1)$. Merge is $O(n)$, so we will write as $c*n$.

Time to sort a list of length n, $t(n)$, for $n > 1$, is then:

$t(n) = 2*t(n/2) + c*n.$

But $t(n/2)$ must then be $2*t(n/4) + c*n/2$. So

$t(n) = 2*(2*t(n/4) + c*n/2) + c*n = 4*t(n/4) + 2c*n$

$t(n) = ... = 8*t(n/8) + 3c*n$

So $t(n) = 2^k*t(n/2^k) + kc*n$

This eventually stops when the list is of size 1, which happens when $k = \log_2 n$. But $t(n/2^k) = t(n/2^{\log_2 n}) = O(1)$ since list is length 1. Also, $2^{\log_2 n} = n$.

So $t(n) = n + \log_2 n*c*n$ which is $O(n \log n)$

Alternative Mergesort implementations

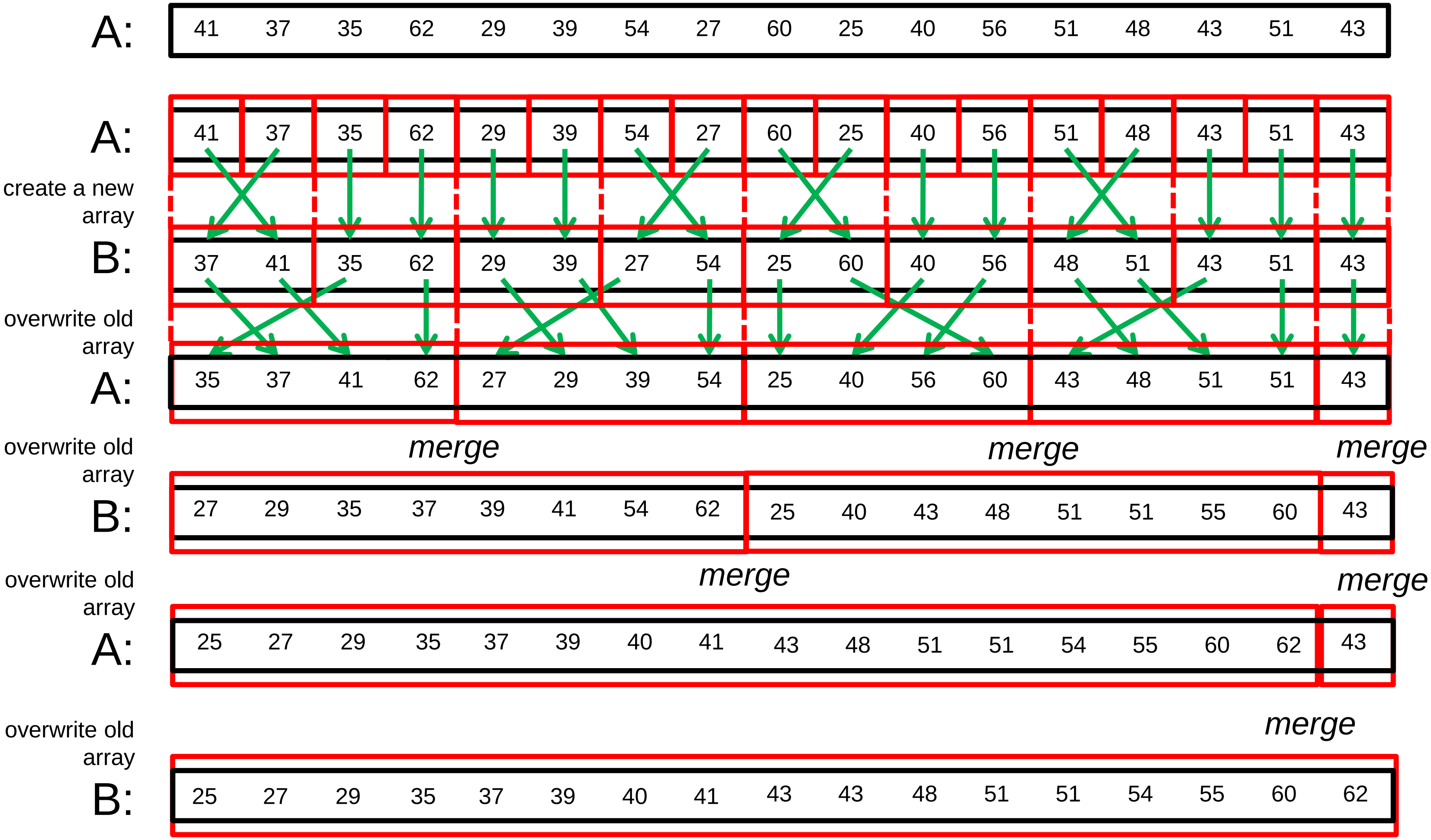
1. implementing mergesort on linked lists is (probably) easier

2. Mergesort on arrays can be implemented bottom-up rather than top down, using just O(n) extra space:

Create a new empty list of size n, called list 2
For each pair of cells in original list
    merge into sorted pair in corresponding cells in list 2
For each successive group of 4 cells in list 2
    merge into sorted group of 4 in corresponding cells in original list
For each successive group of 8 cells ...

    continuing until entire list is sorted.

Treat each cell in the array as a list of size 1



Next Lecture

Quicksort