mergesort: sorting by merging

- given two ***already-sorted*** lists we can efficiently merge them into a single sorted list (merge algorithm to be discussed)
- mergesort uses a divide-and-conquer approach to break an unsorted list into smaller and smaller parts, merging them together to create a sorted whole
- when we study algorithm efficiency we'll see this can be very effective for large lists

merging two sorted lists

- use three indices, initialized to 0:
  - pos1: position in sorted list 1
  - pos2: position in sorted list 2
  - posR: position in list we're creating
- repeat until we hit the end of one sorted list:
  - look at elements in list1,pos1 and list2,pos2, pick smaller
  - copy that to posN in new list
  - increment posN and pos1 or pos2 (whichever we just used)
- then add all remaining elements from the unfinished list

merge example

- showing current positions with *
- merging (*10,20,30) with (*6,8,23,91) into (*)
- compare 6 & 10, copy 6 into result
- L1: (*10,20,30), L2: (6,*8,23,91), R (6, *)
- L1: (*10,20,30), L2: (6,8,*23,91), R (6, 8, *)
- L1: (10,*20,30), L2: (6,8,*23,91), R (6, 8, 10, *)
- L1: (10,20,*30), L2: (6,8,*23,91), R (6, 8, 10, 20, *)
- L1: (10,20,*30), L2: (6,8,23,*91), R (6, 8, 10, 20, 23, *)
- L1: (10,20,30)*, L2: (6,8,23,*91), R (6, 8, 10, 20, 23, 30, *)
- L1: (10,20,30)*, L2: (6,8,23,91)*, R (6, 8, 10, 20, 23, 30, 91)*

mergesort concept

- each call we're given section of array to sort, low position and high position (e.g. 0,size-1 initially)
- if low > high ignore it
- if low == high we're done (a single element, is sorted)
- otherwise
  - pick middle position
  - call mergesort on low .. middle
  - call mergesort on middle+1..high
  - merge the two sorted halves

mergesort example

- mergesort (10, 3, 8, 6)
  - calls mergesort on (10,3)
    - calls mergesort on (10)
    - calls mergesort on (3)
    - calls merge on (10) and (3), giving (3,10)
  - calls mergesort on (8,6)
    - calls mergesort on (8)
    - calls mergesort on (6)
    - calls merge on (8) and (6), giving (6,8)
  - calls merge on (3,10) and (6,8)

mergesort algorithm

```
mergesort(float arr[ ], int lower, int upper)

   if lower >= upper there is nothing left to sort

   otherwise
      midpoint = (lower+upper)/2
      mergesort(arr, lower, midpoint)
      mergesort(arr, midpoint+1, upper)
      merge(arr, lower, midpoint, upper)
```

this assumes a specialized merge function built for mergesort, where
- the two “halves” to be merged are in arr low..midpoint and midpoint+1..upper
- the merged results are going back into the same section of the array

complication: working “in place”

- our original version of merge assumed we had arrays we were reading from and another array we were writing into
- in mergesort it's all just one array
- as we write the merged results into the array, we can wind up overwriting some of the “first half” data before it gets merged
- one solution is to create a temporary array to hold the results in while merging, then copy that back to the original

revised merge algorithm

```
merge(float arr[], int low, int mid, int high)

    N = high + 1 - low                          (number of elements to be merged)
    allocate new array, temp, of N floats

    pos=0                                       (position in  results)
    pos1 = low, pos2 = mid+1                    (positions in the two halves)

    while pos1 <= mid and pos2 <= high          (i.e. we're not done with either half yet)
        temp[pos] = smaller of arr[pos1], arr[pos2]
        increment pos and either pos1 or pos2   (whichever we used)

    (now we've reached the end of one of the two halves,
       put the remaining elements from the other half into temp)

    copy the contents of temp back into arr     (in positions low..high)
    deallocate the temp array
```