## MergeSort

**Question 1**

**Merging with smaller auxiliary array.** Suppose that the subarray a[0] to a[N-1] is sorted and the subarray a[N] to a[2\*N-1] is sorted. How can you merge the two subarrays so that a[0] to a[2\*N-1] is sorted using an auxiliary array of size *N* (instead of 2*N*)?

Hint: copy only the left half into the auxiliary array.

**Question 2**

**Counting inversions.** An *inversion* in an array *a*[] is a pair of entries *a*[*i*] and *a*[*j*] such that *i*<*j* but *a*[*i*]>*a*[*j*]. Given an array, design a linearithmic algorithm to count the number of inversions.

Hint: count while mergesorting.

<http://stackoverflow.com/questions/337664/counting-inversions-in-an-array>

**Question 3**

**Shuffling a linked list.** Given a singly-linked list containing *N* items, rearrange the items uniformly at random. Your algorithm should consume a logarithmic (or constant) amount of extra memory and run in time proportional to *N*log*N* in the worst case.

Hint: design a linear-time subroutine that can take two uniformly shuffled linked lists of sizes *N*1 and *N*2 and combined them into a uniformly shuffled linked lists of size *N*1+*N*2.

<http://stackoverflow.com/questions/12167630/algorithm-for-shuffling-a-linked-list-in-n-log-n-time>

What about the following? Perform the same procedure as merge sort. When merging, instead of selecting an element (one-by-one) from the two lists in sorted order, flip a coin. Choose whether to pick an element from the first or from the second list based on the result of the coin flip.

shuffle(list):

if list contains a single element

return list

list1,list2 = [],[]

while list not empty:

move front element from list to list1

if list not empty: move front element from list to list2

shuffle(list1)

shuffle(list2)

if length(list2) < length(list1):

i = pick a number uniformly at random in [0..length(list2)]

insert a dummy node into list2 at location i

# merge

while list1 and list2 are not empty:

if coin flip is Heads:

move front element from list1 to list

else:

move front element from list2 to list

if list1 not empty: append list1 to list

if list2 not empty: append list2 to list

remove the dummy node from list

## Quicksort

**Question 1**

**Nuts and bolts.** A disorganized carpenter has a mixed pile of *N* nuts and *N* bolts. The goal is to find the corresponding pairs of nuts and bolts. Each nut fits exactly one bolt and each bolt fits exactly one nut. By fitting a nut and a bolt together, the carpenter can see which one is bigger (but the carpenter cannot compare two nuts or two bolts directly). Design an algorithm for the problem that uses *N*log*N* compares (probabilistically).

<http://www.wisdom.weizmann.ac.il/~naor/PUZZLES/nuts_solution.html>

## Matching Nuts and Bolts - Solution

Suppose that there are n nuts and bolts. A simple modification of Quicksort shows that there are randomized algorithms whose expected number of comparisons (and running time) are O(n log n): pick a random bolt, compare it to all the nuts, find its matching nut and compare it to all the bolts, thus splitting the problem into two problems, one consisting of the nuts and bolts smaller than the matched pair and one consisting of the larger ones. Repeating in this manner yields an algorithm whose expected running time can be analyzed by imitating the known analysis for Quicksort (see, e.g., the book by Coreman, Leiserson and Rivest, **Algorithms,** MIT Press, 1990.) showing that it is O(n log n).

Is this the best possible? There are n! possibilities for matching the nuts and bolts a priori. Every attempted matching between a nut and a bolt has three possible outcomes (they match, the nut is larger, the nut is small). Therefore the *information theoretic lower bound* shows that any bounded degree decision tree that solves the problem has depth at least log (n!)= Theta (n log n). This is a lower bound for the expected number of comparisons in any *randomized* algorithm for the problem as well.

The nuts and bolts matching problem was invented by G. J. E. Rawlins who gives it as an exercise in his book **Compared to what? an introduction to the analysis of algorithms,** Computer Science Press, 1991, on page 293. He gives the above outlined solution.

What about deterministic algorithms? They seem more difficult to find. In fact, even obtaining an o(n^2) algorithm appears to be a non-trivial task. Here is a paper ([Postscript](http://www.wisdom.weizmann.ac.il/%7Enaor/PAPERS/nuts_bolts.ps), [gzipped Postscipt](http://www.wisdom.weizmann.ac.il/%7Enaor/PAPERS/nuts_bolts.ps.gz)) by Noga Alon, Manuel Blum, Amos Fiat, Sampath Kannan, Moni Naor and Rafi Ostrovsky (appeared in SODA 94) that describes an almost asymptotically optimal deterministic algorithm. Recently, Janos Komlos, Yuan Ma and Endre Szemeredi found a deterministic O(n log n) algorithm (appeared in [SODA 96](http://www.siam.org/meetings/general/da96info.htm) ).

### Question 2

**Selection in two sorted arrays.** Given two sorted arrays *a*[] and *b*[], of sizes *N*1 and *N*2, respectively, design an algorithm to find the *kth* largest key. The order of growth of the worst case running time of your algorithm should be log*N*, where *N*=*N*1+*N*2.

* Version 1: *N*1=*N*2 and *k*=*N*/2
* Version 2: *k*=*N*/2
* Version 3: no restrictions

**Question Explanation***Hints*: there are two basic approaches.

* Approach A: Compute the median in *a*[] and the median in *b*[]. Recur in a subproblem of roughly half the size.
* Approach B: Design a constant-time algorithm to determine whether *a*[*i*] is the *kth* largest key. Use this subroutine and binary search.

Dealing with corner cases can be tricky.

**Question 3**

**Decimal dominants.** Given an array with *N* keys, design an algorithm to find all values that occur more than *N*/10 times. The expected running time of your algorithm should be linear.

**Question Explanation***Hint*: determine the (*N*/10)*th* largest key using quickselect and check if it occurs more than *N*/10 times.

*Alternate solution hint:* use 9 counters.