R-9.2 Describe a radix-sort method for lexicographically sorting a sequence S of triplets (k, l, m) where k, l, m are integers in the range $[0, N-1]$, for some $N \geq 2$. Extend this method to sequences of d-tuples.

Solution. Iterate through the index i of the tuples, starting from the last index $(d-1)$ and going down to the first index 0. At each iteration, bucket sort the tuples of S by their i-th entry using N buckets. The running-time of this algorithm is $O(d(n+N))$ where n is the length of S. •

C-9.4 Let $S_1, S_2, \ldots, S_k$ be k different sequences whose elements have integer keys in the range $[0, N-1]$, for some parameter $N \geq 2$. Describe an algorithm running in $O(n+N)$ time for sorting all the sequences (not as a union), where n denotes the total size of all sequences.

Solution. For each sequence S_i, construct a new sequence $\bar{S}_i$ in which the elements are of the form (i, x) for every x in S_i. We sort by performing two rounds of bucketsort. In the first round, we sort $\bar{S}_1, \ldots \bar{S}_k$ into N buckets, by the second entry of each pair. In the second round, we sort this result into k buckets, by the first entry of each pair. The result is that each of the k buckets will contain all the elements of a single sequence $\bar{S}_i$, sorted by the second entry.

The first round runs in $O(n+N)$ whereas the second round runs in $O(n+k)$. Assuming no sequence is empty, $k \leq n$. So the total run time is $O(3n+N)$, or $O(n+N)$. •

C-9.9 Given an unsorted sequence S of n comparable elements, and an integer k, give an $O(n \log k)$ expected-time algorithm for finding the $O(k)$ elements that have rank $\lceil n/k \rceil$, $2\lceil n/k \rceil$, $3\lceil n/k \rceil$, and so on.

Solution. For simplicity, assume that $k = 2^m$ for some $m \geq 0$. The idea is to first find the median of the array because the call has side effect that the array will be partitioned around the median. This side effect speeds up the subsequence calls to quickselect, because those calls can be made on each half-subarrays rather than the whole array.

Apply quickselect in the following order to the array, in the following range:

- $\lceil n/2 \rceil$ on $[0, n]$
- $\lceil n/4 \rceil$ on $[0, n/2]$, $3\lceil n/4 \rceil$ on $[n/2, n]$,
- $\lceil n/8 \rceil$ on $[0, n/4]$, $3\lceil n/8 \rceil$ on $[n/4, n/2]$, $5\lceil n/8 \rceil$ on $[n/2, 3n/4]$, $7\lceil n/8 \rceil$ on $[3n/4, n]$,
- ...
- $\lceil n/k \rceil, 3\lceil n/k \rceil, 5\lceil n/k \rceil, \ldots (k-1)\lceil n/k \rceil$

At each round of quickselect, the array is modified. We call each round of quickselect on the array modified by the previous round of quickselect. The expected running time is $O(n)$ on the first call, $O(n/2)$ on the next two calls, $O(n/4)$ on the following next 4 calls, etc. So the total expected running time is $O(n \log k)$. •

C-9.10 Suppose you have two sorted lists, A and B, of n elements each, all of which are distinct. Describe a method that runs in $O(\log n)$ time for finding the median in the set defined by the union of A and B.

Solution. The idea is to do a simultaneous binary search on both lists, where we keep the two indices summing to $k-1$. The details are given in the python code below.

```
def k_small(A, B, k):
    n = len(A)
    m = len(B)

    # Handle base cases.
    if n == 0:
        return B[k - 1]
    if m == 0:
        return A[k - 1]
    if n + m == k:
        if A[n - 1] < B[m - 1]:
            return B[m - 1]
        else:
            return A[n - 1]

    # Set starting indices
    i = min((n - 1) / 2, k - 1)
    j = k - 1 - i
    # Cap j if too big
    if j >= m:
        j = m - 1
        i = k - 1 - j

    # Recurse
    if A[i] <= B[j]:
        if j == 0:
            return A[i]
        elif B[j - 1] <= A[i]:
            return A[i]
        else:
            return k_small(A[i:n], B[0:j], k - i)
    else:
        if i == 0:
            return B[j]
        elif A[i - 1] <= B[j]:
            return B[j]
        else:
            return k_small(A[0:i], B[j:m], k - j)
```

•