NOTE: I am not yet sure that this algorithm is actually new, yet. I am looking into it.

A well-understood problem in computer science is *selection*: given a collection of size n, find the k largest elements. We typically expect that k is several orders of magnitude smaller than n. There are two commonly used solutions to this problem.

The first, typically called quickselect, is due to [3]. It is implemented by modifying the quicksort algorithm to recurse only once, as follows.

Call : `quick-select` (X, k)
Input : An array X of size n and an integer $0 \leq k \leq n$.
Result: Moves the k largest elements of X to the last k positions in X.
if $k = n$ **or** $k = 0$ **then return**;
pivot $\leftarrow$ `choose-pivot` (X);
newPivotPos $\leftarrow$`partition(`X`,`pivot`)`;
$X_L \leftarrow X[0 \ldots$ newPivotPos $- 1]$; `// The elements of` X_L `are <` pivot
$X_R \leftarrow X[$newPivotPos $+ 1 \ldots]$; `// The elements of` X_R `are >` pivot
if `size(`X_R`)` $> k$ **then**
  `quick-select(`X_R, k`)`;
else if `size(`X_R`)` $< k$ **then**
  `quick-select(`X_L, $k -$ `size(`X_R`))`;
else return; `// size(`X_R`)` $= k$

If we can choose our pivot to have rank between ϵn and $(1 - \epsilon)n$ for some $\epsilon > 0$, this algorithm will take $O(n)$. The most well-known algorithm to accomplish this deterministically is presented in its essentials in [1], and is as follows:

Call : `choose-pivot(`X`)`
Input : An array X of size n.
Output: $x \in X$ such that the rank of x is between $\frac{3}{10}n$ and $\frac{7}{10}n$.
Initialize Y as a new array with length $n = \lfloor n/5 \rfloor$;
for $i = 0$ *to* $\lfloor n/5 \rfloor - 1$ **do**
  $Y[i] \leftarrow$ the median of $X[5i \ldots 5i + 4]$;
end
`quick-select(`Y,$\lfloor n/2 \rfloor$`)`;
return *the minimum element of* $Y[\lceil n/2 \rceil \ldots]$;

For evident reasons, this is sometimes called the median-of-medians pivot procedure.

The second algorithm, which we will refer to as heapselect, is extremely simple.

Input : A collection X of size n and an integer $0 \leq k \leq n$.
Output: An array of the k largest elements of X.
heap $\leftarrow$`new-heap(`$k + 1$`)`; `// heap is initialized with capacity` $k + 1$
foreach $x \in X$ **do**
  `insert(`x,heap`)`;
  if `size(heap)`$> k$ **then**
    `delete-min(heap)`;
  end
end
return `to-array(heap)`;

This takes $O(n \log k)$, but has the advantage that it requires only one pass over the collection. In computations on extremely large data sets, we may not be able to fit the entire collection into memory at once. In these cases, it is extremely useful to have an algorithm that performs only a single pass over the data.

I present an algorithm that performs only a single pass over the data, and runs in $O(n)$ time, but uses only $O(k)$ memory. My algorithm, which I call softselect, makes critical use of an esoteric data structure called *soft heaps*, which were invented by [2]. Soft heaps are priority queues that bypass information-theoretic lower bounds by allowing a certain proportion of elements to become 'corrupted' in a carefully defined way. The easiest way to define an interface for soft heaps is probably as follows. Fix some $0 < \epsilon < 1$. A soft heap supports the following operations:

- Insert an element into the soft heap, in amortized $O(\log(1/\epsilon))$ time.
- Delete an element from the soft heap that is "nearly" the minimum element of the heap, in amortized $O(\log(1/\epsilon))$ time. If the element deleted is x, it is guaranteed that x is less than at least $(1-\epsilon)n$ of the current elements of the soft heap. For brevity, we abuse notation by referring to this operation as delete-min.
- Iterate over all of the elements of the soft heap in some unspecified order, in $O(n)$ time.

An implementation of such a data structure can be found in [4], but we will not concern ourselves with the implementation.

When they were invented in [2], it was noted that soft heaps could be used to deterministically guarantee good pivot selection in $O(n)$ time, for quicksort as well as quickselect, as follows. Set $\epsilon = 1/3$, and add every element of the collection to a soft heap. Delete $n/3$ elements from the soft heap, and let α be the largest of the deleted elements. Since α was deleted, we know that α is less than at least $n/3$ elements which are still in the soft heap. Furthermore, α is greater than each of the other $n/3$ elements that were deleted. Therefore, the rank of α is between $n/3$ and $2n/3$. Partitioning around α in quickselect, we only have to recurse into a subcollection of size at most $2n/3$, so we can narrow the collection down to the k largest elements in time proportional to $n + 2n/3 + (2/3)^2 n + \cdots = n\sum_{i=0}^{\infty}(2/3)^i = O(n)$. However, this algorithm still requires the entire collection to be stored in memory, and performs multiple passes.

Here, then,/ is the advertised algorithm.

Input : A collection X of size n, which can only be iterated over once, and an integer $0 \le k \le n$
Output: The k largest elements of X

```
α ← −∞;
heap ←NewSoftHeap(ϵ = 1/2);
foreach x ∈ X do
    if x > α then
        heap.insert(x);
        if size(heap)> 2k then
            α ← max(α, delete-min(heap));
        end
    end
end
return quick-select(to-array(heap),k);
```

The runtime of this algorithm is $O(n\log(1/\epsilon)) + O(k) = O(n)$, and it clearly uses only $O(k)$ auxiliary memory. We must show its correctness, however.

Lemma 1. *The rank of α in X is always strictly less than $n - k$.*

Proof. When α was first assigned to its current value, the size of the soft heap was $2k + 1$, and α was the element deleted from the soft heap. By the definition of delete-min, α was less than at least $\epsilon \cdot 2k = k$ of the other elements of the heap at the time. □

Theorem 2. *After we have finished iterating over X, the soft heap contains each of the k largest elements of X.*

Proof. By the lemma, we will never skip over any of the k largest elements, since we skip over only elements which have rank less than α. Furthermore, if any of the k largest elements were deleted, they would be assigned to α, which contradicts the lemma. □

Therefore, the pass with the soft heap eliminates all but $2k$ candidates for the k largest elements of X, after which quickselect takes $O(k)$.

References

[1] Manuel Blum, Robert W. Floyd, Vaughan Pratt, Ronald L. Rivest, and Robert E. Tarjan. Time bounds for selection. *J. Comput. Syst. Sci.*, 7:448–461, August 1973.

[2] Bernard Chazelle. The soft heap: an approximate priority queue with optimal error rate. *J. ACM*, 47:1012–1027, November 2000.

[3] C. A. R. Hoare. Algorithm 65: find. *Commun. ACM*, 4:321–322, July 1961.

[4] Haim Kaplan and Uri Zwick. A simpler implementation and analysis of chazelle's soft heaps. In *Proceedings of the twentieth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '09, pages 477–485, Philadelphia, PA, USA, 2009. Society for Industrial and Applied Mathematics.