To help make grading easier, please start each numbered problem on a new sheet of paper, make sure your name appears on each page of your solutions. and staple everything together. Groups of up to three people can turn in a single solution. You are strongly encouraged, but not required, to typeset your solutions using LaTeX. You may use books, journal articles, notes, the web, other students, faculty, flying monkeys, or the Oracle at Delphi to help solve these problems, but you *must* cite *any* source that you use!

*Stars indicate problems whose answers I don't know. The larger the star, the less sure I am of the answer. A star does not necessarily imply that the problem is open, or even difficult—try it anyway! On the other hand, some unstarred problems may be unfairly hard—try them anyway!

1. Blum, Floyd, Pratt, Rivest, and Tarjan discovered the following deterministic algorithm for selecting the kth largest element in an unsorted array in linear time. The algorithm actually returns the *index* of the selected element.

```
SELECT(A[1..n], k):
  for i ← 1 to ⌈n/5⌉
      B[i] ← MEDIAN(A[5i − 4 .. max{5i, n}])
  p ← SELECT(B, ⌈n/10⌉)     ⟨⟨Recurse!⟩⟩
  q ← PARTITION(A, B[p])
  if k < q
      return SELECT(A[1..q − 1], k)
  else if k > q
      return SELECT(A[q + 1..n], k − q)
  else
      return q
```

   The algorithm uses two subroutines. MEDIAN computes the median element in an array with up to five elements, obviously in constant time. PARTITION is directly from Quicksort; it partitions the array into two subarrays, one containing all the elements smaller than the pivot value, the other containing all the elements larger than the pivot value, and then returns the index of the pivot value itself in the partitioned array. Since the pivot is smaller than at least three of the elements in half of the 5-element chunks, and larger than at least three elements in the other half, we can bound the running time by the recurrence $T(n) \leq O(n) + T(n/5) + T(7n/10)$, whose solution is $T(n) = O(n)$.

   Describe and analyze an external version of this algorithm that can select the kth largest element in an array of N elements (on disk) in $O(n)$ time. For simplicity, assume all the input elements are distinct.

2. Design and analyze an external memory algorithm to remove all duplicate elements from an unsorted array. The running time of your algorithm should be

$$O\left(n + \sum_{i=1}^{K} n_i \log_m \frac{n}{N_i}\right)$$

   I/Os, where K is the number of distinct input elements (or equivalently, the size of the output), and N_i is the multiplicity of ith largest element in the output. In particular, $\sum_{i=1}^{K} N_i = N$, so if $N_i = 1$ for all i, the running time simplifies to the sorting bound $O(n \log_M n)$.

   [Hint: Modify mergesort to remove duplicates as soon as they are found. How many of the N_i copies of element i can you still have after j merge passes? Try it in internal memory first!]

⋆3. The previous algorithm can be used to reduce the worst-case running time of quicksort to $O(n \log n)$, by choosing the median of the array as the pivot at every level of recursion. However, we can achieve the same effect with high probability by choosing a *random* pivot.[1]

Find the expected running time (in I/Os) of the following randomized external sorting algorithm, the obvious external analogue of randomized quicksort. The subroutine DISTRIBUTE partitions the array A into m chunks using the elements of B as pivots, where elements in earlier chunks are smaller than elements of later chunks, and returns the indices of the pivots in an array L. For convenience, we set $L[0] = 0$ and $L[m+1] = N$.

```
RANDOMDISTRIBUTIONSORT(A[1..N]):
  if N ≤ M
      sort A in internal memory     ⟨⟨O(n) time⟩⟩
  else
      for i ← 1 to m − 1            ⟨⟨O(m) = o(n) time⟩⟩
          B[i] ← A[RANDOM(1, N)]
      L ← DISTRIBUTE(A, B)          ⟨⟨O(n) time⟩⟩
      for i ← 0 to m
          RANDOMDISTRIBUTIONSORT(A[L[i] + 1 .. L[i + 1]])
```

4. In the first lecture, we saw the classical proof that any comparison-based sorting algorithm requires $\Omega(n \log n)$ time in the worst case. This lower bound actually holds in *much* more powerful models of computation.[2] Unfortunately, no such generalizations are known—or at least, none have ever been published—for the $\Omega(n \log_m n)$ external-memory lower bound. Until now!

   (a) Let W be a *constant-size* set of real numbers. A *weighted comparison tree* is a binary decision tree, where decisions have the form "if $\omega \cdot A[i] < A[j]$" for some weight $\omega \in W$ and some array indices i and j. For example, a standard comparison tree uses the set $W = \{1\}$.

   Define an external-memory version of the weighted comparison tree model, and prove an $\Omega(n \log_m n)$ I/O lower bound for sorting in this model.

   ★(b) Let d be a *constant* and let W be a *constant-size* set of vectors in $\mathbb{R}^d$. In a *restricted d-linear decision tree*, every decision has the form "if $\sum_{k=1}^{d} \omega_k \cdot A[i_k] > 0$" for some weight vector $(\omega_1, \omega_2, \ldots, \omega_d) \in W$ and some array indices $i_1, i_2, \ldots, i_d$. For example, a restricted 2-linear decision tree is just a weighted comparison tree.

   Define an external-memory version of the restricted d-linear decision tree model, and prove an $\Omega(n \log_m n)$ I/O lower bound for sorting in this model.

   ★(c) Generalize!

   [Students who took CS 497 from me last semester will have a *slightly* easier time with this problem, but only slightly.]

[1] You can find four different ways to analyze randomized quicksort in my algorithms lecture notes: http://www.cs.uiuc.edu/~jeffe/teaching/373/.

[2] See my lecture notes at http://www.cs.uiuc.edu/~jeffe/teaching/473/ for details.