R-4.18
Consider a sequence of \( n \) distinct numbers and quickselect used for finding
the smallest (i.e., 1st) number that continues to select the largest number as
the pivot throughout its recursive calls.

Complexity:
\[
n + (n - 1) + (n - 2) + \cdots + 2 + 1 = \Omega(n^2)
\]

C-4.4
Two solutions:
1. Slight modification of cuckoo hashing: hash \( A \) into hash table, but before
inserting item \( o \), check the positions \( h_1(o) \) and \( h_2(o) \). If one of them
already contains a copy of \( o \), disregard (do not insert) \( o \).

Analysis: \( O(n) \) extra memory, \( O(n) \) time (expected).
2. Sort \( A \) using quicksort. Remove all items that are the same as their
predecessors.

Analysis: no extra memory, \( O(n \log n) \) time (expected).

R-5.4
a) \( T(n) = \Theta(n) \) (Case 1)
b) \( T(n) = \Theta(n^3) \) (Case 1)
c) \( T(n) = \Theta(n^4 \log^5 n) \) (Case 2)
d) \( T(n) = \Theta(n^{\log_3 7}) \) (Case 1)
e) \( T(n) = \Theta(n^3 \log n) \) (Case 3)

C-5.5
Algorithm: sort \( X \). Process sorted list in increasing order. If \( x_i \) is already
protected, move on. Otherwise, place a guard at position \( x_i + 1 \).

Time: \( O(n \log n) \).

Correctness: assume there is an optimal solution \( OPT \) that differs from ours.
Let \( x \) be the first position of a guard that is in our solution but not in \( OPT \).
Let \( x' \) be the corresponding placement of a guard in \( OPT \). We cannot have
\( x' > x \) (we placed a guard at \( x \) because there was an unguarded painting at
\( x - 1 \); our solution and \( OPT \) are identical to the left of \( x \), so placing guard at
\( x' > x \) would leave the painting at \( x - 1 \) unguarded). Therefore, \( x' < x \). Every
painting to the left of \( x - 1 \) is already guarded, so moving a guard from \( x' \) to
\( x \) does not make any painting unguarded. Therefore, our solution is no worse
that \( OPT \), so it is optimal.
C-5.8

proc UpperEnvelope(([a₀, b₀], h₀), ([a₁, b₁], h₁), ..., ([aₙ, bₙ], hₙ))
if n = 0 then
  return (0, 0), (a₀, h₀), (b₀, 0)
else
  U₁ ← UpperEnvelope(([a₀, b₀], h₀), ..., ([a_{n/2}, b_{n/2}], h_{n/2}))
  U₂ ← UpperEnvelope(([a_{n/2}, b_{n/2}], h_{n/2}), ..., ([aₙ, bₙ], hₙ))
  U ← merge(U₁, U₂)
  return U
end if

Merging upper envelopes (the merge procedure) is realized in a way similar to merging in mergesort. Upper envelopes are ordered by starting points of the intervals, so it is always enough to compare the heights of the first intervals on the two lists to determine the combined upper envelope.

Analysis: \(O(m)\) time for merge called with lists of size \(m\), \(O(n \log n)\) time for UpperEnvelope.