Jordan Taranto

Code Source

```
#lang racket

;; QuickSort with Median of Medians Partitioning
;; Main function to sort the list using quicksort with median-of-medians
(define (quicksort-mom lst)
  (cond
    [(< (length lst) 2) lst] ;; base case: lists with less than 2 elements
are already sorted
    [else
     (let* ([pivot (median-of-medians lst)] ;; Find pivot using median-of-
medians
            [less (filter (lambda (x) (< x pivot)) lst)] ;; Elements less
than pivot
            [equal (filter (lambda (x) (= x pivot)) lst)] ;; Elements equal
to pivot
            [greater (filter (lambda (x) (> x pivot)) lst)]) ;; Elements
greater than pivot
       ;; Recursively quicksort the parts and combine
       (append (quicksort-mom less) equal (quicksort-mom greater)))]))

;; Median of Medians Algorithm
;; Function to calculate median-of-medians pivot
(define (median-of-medians lst)
  (cond
    [(<= (length lst) 5) (find-median lst)] ;; If the list has 5 or fewer
elements, return the median
    [else
     (let ([medians (map find-median (partition-into-sublists lst 5))]) ;;
Find medians of sublists
       (median-of-medians medians))])) ;; Recursively find the median of
medians

;; Helper function to partition a list into sublists of size 'n'
(define (partition-into-sublists lst n)
  (if (empty? lst)
```

```
      '()
       (let ([sublist (if (< (length lst) n) lst (take lst n))]) ;; Handle
cases where lst has fewer than n elements
        (cons sublist (partition-into-sublists (drop lst (length sublist))
n))))) ;; Drop only the actual size of sublist

;; Helper function to find the median of a list
;; Assumes the list has 5 or fewer elements
(define (find-median lst)
  (let ([sorted (selection-sort lst)]) ;; Sort the list using selection sort
    (list-ref sorted (quotient (length sorted) 2)))) ;; Return the middle
element

;; Selection Sort (used to sort sublists of size <= 5)
(define (selection-sort lst)
  (if (empty? lst)
      '()
      (let ([min (find-min lst)])
        (cons min (selection-sort (remove-first min lst))))))

;; Helper function to find the minimum element in a list
(define (find-min lst)
  (foldl (lambda (x acc) (if (< x acc) x acc)) (first lst) lst))

;; Helper function to remove the first occurrence of an element from a list
(define (remove-first elem lst)
  (cond
    [(empty? lst) '()]
    [(= (first lst) elem) (rest lst)]
    [else (cons (first lst) (remove-first elem (rest lst)))]))

;; Helper function to check if a list is sorted
(define (is-sorted? lst)
  (cond
    [(or (empty? lst) (empty? (rest lst))) #t]
    [else
     (and (<= (first lst) (second lst))
          (is-sorted? (rest lst)))]))

;; Function to generate random integers (provided in the problem statement)
```

```
(define (generate-random-integers count min-value max-value)
  (define (generate n)
    (if (zero? n)
        '()
        (cons (+ min-value (random (- max-value min-value)))
              (generate (- n 1)))))
  (generate count))

;; TEST CASES

;; 4
(define 4-test (generate-random-integers 4 1 100))
(displayln "Original list: ")
(displayln 4-test)
(displayln "Sorted list using quicksort with median-of-medians: ")
(displayln (quicksort-mom 4-test))

;; 43
(define test-43 (generate-random-integers 43 1 100))
(displayln "Original list: ")
(displayln test-43)
(displayln "Sorted list using quicksort with median-of-medians: ")
(displayln (quicksort-mom test-43))

;; 403
(define 403-test (generate-random-integers 403 1 100))
(displayln "Original list: ")
(displayln 403-test)
(displayln "Sorted list using quicksort with median-of-medians: ")
(displayln (quicksort-mom 403-test))



;; 400,003
(define 400003-test (generate-random-integers 400003 1 100))
(displayln "Original list: ")
(displayln 400003-test)
(displayln "Sorted list using quicksort with median-of-medians: ")
(displayln (quicksort-mom 400003-test))
```