During the whole process of making the quicksort work in racket, I made 2 of many transcripts that aided my research on how to make the code working through Chat GPT. I decided to use the code provided from Transcript 2 as it was able to sort out a list of 10M+ in a few seconds unlike the first Transcript. It did take me a long time to get Transcript 2 since I was able to make it through multiple attempts and luck. Also, moving over from transcript 1 to 2 was annoying since I had to do the process all over again, but it was worth it in the end. Transcript 2 was able to transfer some info from Transcript 1 by having their “memory updated”. The most important “memory update” was when I told it to make the quicksort algorithm in racket with median-of-medians algorithm, including some programming notes from the rubric.

```
;; Generate a list of random integers
(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))
```

This initial code was functionally correct and was also provided by the rubric “Program01”. It was easy to start with, and it didn’t change much throughout the end, so I didn't do much with it.

```
;; Function to find the median of a list
(define (median lst)
  (let ([sorted (sort lst <)])
    (list-ref sorted (quotient (length sorted) 2))))
```

The median function was also correctly implemented as is and because of that, I really didn't use ChatGPT to modify this code too much. It was one of many blocks of code that makes up the median-of-medians algorithm when I asked Chat GPT for it, and it wasn’t too difficult to make because of it.

```
;; Function to find the median of medians (pivot selection)
(define (median-of-medians lst)
  (let loop ([remaining lst] [medians '()])
    (cond
      ;; If fewer than 5 items, return the median of the remaining items
      [(not (at-least-5-items? remaining))
       (median remaining)]
      ;; Take the first 5 elements, find the median, and continue processing the rest
      [else
       (loop (drop remaining 5)
             (cons (median (take remaining 5)) medians))])))
```


```
;; Function to find the median of medians (pivot selection)
(define (median-of-medians lst)
  (let loop ([remaining lst] [medians '()])
    (cond
      [(not (at-least-5-items? remaining))
       (median remaining)]
      [else
       (loop (drop remaining 5)
             (cons (median (take remaining 5)) medians))])))
```


At first, it was created along with some other functions that made the algorithm for median-of-medians. However, this function (and qrst) would disappear in the process. I assumed that memory was lost whenever I kept telling Chat GPT to keep adding more other functions to the program, and that it wasn’t being used at the time. I didn’t notice this until the program told me that the qsrt function was missing from the program. The only

noticeable changes were the comments disappearing. It wasn't difficult to make since I asked for it in the beginning, but it was annoying since it disappeared one time in the process.

```
;; Helper function to check if list has at least 5 items
(define (at-least-5-items? lst)
  (and (pair? lst) (pair? (cdr lst)) (pair? (cddr lst))
       (pair? (cadddr lst)) (pair? (cddddr lst))))
```

```
;; Helper function to check if list has at least 5 items
(define (at-least-5-items? lst)
  (define (check lst n)
    (cond
      [(null? lst) #f]      ;; If the list is empty, return #f
      [(= n 0) #t]          ;; If we've seen at least 5 items, return #t
      [else (check (cdr lst) (- n 1))])) ;; Recursively check the rest of the list
  (check lst 5))
```

This was also another function for the median-of-medians algorithm. I noticed that this function wasn't correctly implemented when I was given the "cadddr: contract violation" error twice. According to Chat GPT, this function was attempting to access elements that do not exist in the list. It wasn't too difficult to deal with, as I kept posting the error to the LLM until I got it to fix the function.

```
;; Quicksort using the median-of-medians partitioning method
(define (qsrt lst)
  (if (or (null? lst) (null? (cdr lst)))
      lst  ;; Base case: empty or single-item list is already sorted
      (let* ([pivot (median-of-medians lst)]
             [smaller (filter (lambda (x) (< x pivot)) lst)]
             [equal (filter (lambda (x) (= x pivot)) lst)]
             [greater (filter (lambda (x) (> x pivot)) lst)])
        (append (qsrt smaller) equal (qsrt greater)))))
```

```
;; Quicksort using the median-of-medians partitioning method
(define (qsrt lst)
  (cond
    [(null? lst) '()]       ;; Base case: empty list is already sorted
    [(null? (cdr lst)) lst] ;; Base case: single-item list is already sorted
    [(<= (length lst) 5) (selection-sort lst)] ;; Use selection sort for small lists
    [else
     (let* ([pivot (median-of-medians lst)]
            [smaller (filter (lambda (x) (< x pivot)) lst)]
            [equal (filter (lambda (x) (= x pivot)) lst)]
            [greater (filter (lambda (x) (> x pivot)) lst)])
       (append (qsrt smaller) equal (qsrt greater)))]))
```

Like stated earlier with the median-of-medians function, this chunk of code would disappear in the process whenever I kept asking Chat GPT to keep adding more functions to the code and it may have caused memory loss. It would be brought back to the program, but would change when I asked Chat GPT to implement selection sort for short sublist. The revised code would now be able to handle lists with 5 or fewer elements using selection sort.

```
;; Function to filter a list based on a value and comparison type
(define (filter-by-comparison lst value comparison-type)
  (cond
    [(equal? comparison-type 'less)
     (filter (lambda (x) (< x value)) lst)]
    [(equal? comparison-type 'equal)
     (filter (lambda (x) (= x value)) lst)]
    [(equal? comparison-type 'greater)
     (filter (lambda (x) (> x value)) lst)]
    [else (error "Invalid comparison type. Use 'less, 'equal, or 'greater.")]))
```

This function was correctly implemented since it was able to filter the list on the first try. No changes were needed here overall. It wasn't difficult to do since it was already working when I asked for it.

```
;; Function to find the minimum value of a list
(define (find-min lst)
  (foldl min (car lst) (cdr lst)))  ;; Start with the first item and fold with the min function

;; Function to find the maximum value of a list
(define (find-max lst)
  (foldl max (car lst) (cdr lst)))  ;; Start with the first item and fold with the max function
```

Telling Chat GPT to make a function about finding the min and max value in a list was really easy and straightforward. No code was changed during the process.

```
;; Function to check if a list is sorted in nondecreasing order
(define (sorted? lst)
  (define (check lst)
    (cond
      [(or (null? lst) (null? (cdr lst))) #t]
      [(<= (car lst) (cadr lst)) (check (cdr lst))]
      [else #f]))
  (check lst))
```

This chunk of code was easy to implement since it works by recursively comparing each element with the next one. There wasn't any need for modifying or changing the code since it was already working.

```
;; Selection sort for small lists
(define (selection-sort lst)
  (define (select-min lst)
    (if (null? (cdr lst))
        lst
        (let* ([min-val (apply min lst)]
               [min-index (index-of min-val lst)]
               [rest (remove min-val lst)])
          (append (list min-val) (selection-sort rest)))))

  (define (index-of value lst)
    (let loop ([lst lst] [index 0])
      (cond
        [(null? lst) (error "Value not found in the list")]
        [(equal? (car lst) value) index]
        [else (loop (cdr lst) (+ index 1))])))

  (select-min lst))
```

This required 4 attempts until I got this chunk of code working in Transcript 2. Adding this to the code, however, required some modification for the qrst function since the selection sort would be required for a shorter sublist. It wasn't too hard making this code, but it did require few attempts to work do to LLM nature being wrong sometime.

```
;; Function to test sorting and manage printing based on list size
(define (test-sorting size)
  (displayln (format "Testing list of size ~a:" size))
  (define test-list (generate-random-integers size 0 1000))
  (define sorted-list (qsrt test-list))

  ;; Display the sorted list only if the size is less than or equal to 100,000
  (if (<= size 100000)
      (begin
        (displayln "Sorted list:")
        (displayln sorted-list))
      (displayln "Sorting complete for large list.")))
```

```
;; Function to test sorting and manage printing based on list size
(define (test-sorting size)
  (displayln (format "Testing list of size ~a:" size))
  (define test-list (generate-random-integers size 0 10000))
  (define sorted-list (qsrt test-list))

  ;; Display the sorted list if the size is smaller than 100,000
  (if (< size 100000)
      (begin
        (displayln "Sorted list:")
        (displayln sorted-list))
      (displayln "Sorted list not displayed for large list size."))

  ;; Find and display the minimum and maximum values of the list
  (define min-value (find-min test-list))
  (define max-value (find-max test-list))
  (displayln (format "Minimum value: ~a" min-value))
  (displayln (format "Maximum value: ~a" max-value))

  ;; Check if the sorted list is actually sorted
  (displayln (format "Is the list sorted? ~a" (sorted? sorted-list)))

  ;; Filter sorted list items based on comparison type and the value 5000
  (define filtered-less (filter-by-comparison sorted-list 5000 'less))
  (define filtered-equal (filter-by-comparison sorted-list 5000 'equal))
  (define filtered-greater (filter-by-comparison sorted-list 5000 'greater))

  ;; Display the filtered results if the size is smaller than 100,000
  (if (< size 100000)
      (begin
        (displayln "Filtered list (less than 5000):")
        (displayln filtered-less)
        (displayln "Filtered list (equal to 5000):")
        (displayln filtered-equal)
        (displayln "Filtered list (greater than 5000):")
        (displayln filtered-greater))
      (displayln "Filtered results not displayed for large list size."))

  ;; Add two newlines after each test result
  (displayln "")
  (displayln ""))
```

The process of making test cases with lists of sizes 4, 43, 403, and 400,003 were simple to make, but took a long time as I had to make multiple trials and errors for size 10,000,000 to work. It was difficult to work with, but getting a list sorted with size 10,000,000 within a few seconds would tell me that the code is better optimized and that it would be a good chunk of code to start with. It took me Transcript 2 and more deleted chats to get the large size list to work.

I had to keep updating the test-sorting function so that it would include all the utility functions that needed in the rubric. Throughout multiple tests, I discovered that it would take a long time to print out a list that was too large. For example (done in Transcript 1), a list with size 400,003 would take a few minutes to load and print all items. So I added a condition that allowed test cases to stop printing out integers from the list if their size was too

large (100,000 and more). I also added two new lines after each test was conducted, so that the command log would be a lot more readable.
Overall, this chunk of code was the hardest to make since it required many trial and errors, conditions, and functions to get what I wanted.

```
;; Test sorting for lists of size 4, 43, 403, 400,003, and 10,000,000
(test-sorting 4)
(test-sorting 43)
(test-sorting 403)
(test-sorting 400003)
(test-sorting 10000000)
```

This ensures that the sorting algorithm is tested across a range of list varied sizes, which helps me validate performance and correctness much easier and faster. It is easy to implement, as you need to call out the test-sorting function and the size the list you want to have.
