TRANSCRIPT

A transcript/log of your LLM interaction. It is not necessary to include interactions with the AI about general algorithms or problem decomposition, but do include any interactions in which you ask the LLM to generate or modify code. Include the LLM's answer, and show any further interactions in which you refine the code, attempt to get it to fix errors, etc.

PROJECT START: LAYING FOUNDATIONS

Me: in racket, without using library sort routines, can you write a function which divides an array into n/5 groups where each group is 5 elements each. Except the last group which may have less than 5 elements.

ChatGPT: Yes! To divide an array into groups of size 5, where the last group may have fewer than 5 elements, you can create a function in Racket that takes an array (or list) and divides it into sublists of up to 5 elements each.

Here's a function that implements this:

```
#lang racket

(define (group-into-fives lst)

  (if (empty? lst)

      '()  ; Base case: if the list is empty, return an empty list

      (cons (take lst 5)  ; Take the first 5 elements (or fewer if less than 5 remain)

            (group-into-fives (drop lst 5)))))  ; Recursively call with the rest of the elements

;; Example usage:

(group-into-fives '(1 2 3 4 5 6 7 8 9 10 11))
```

Me: I keep getting this error. what does it mean: take: contract violation expected: a list with at least 5 elements given: '(11)

ChatGPT: The error you're seeing occurs because the `take` function in Racket expects a list with at least 5 elements, but you're passing a shorter list when there are fewer than 5 elements left. In your case, when the list is down to `'(11)`, the `take` function tries to take 5 elements, which it cannot do because there is only 1 element left.

To fix this, you can modify the function to check the length of the list and ensure that `take` is only called with the appropriate number of elements.

Here's the corrected version of the function:

```
#lang racket
```

```
(define (group-into-fives lst)

  (if (empty? lst)

      '()  ; Base case: if the list is empty, return an empty list

      (let* ((n (min 5 (length lst)))  ; Take the smaller of 5 or the
remaining list length

             (group (take lst n)))     ; Take only up to n elements

        (cons group

              (group-into-fives (drop lst n))))))  ; Drop the first n elements
and recurse



;; Example usage:

(group-into-fives '(1 2 3 4 5 6 7 8 9 10 11))
```

Me: can you create a function, without using library sort routines, where you sort the created n/5 groups and find the median of all groups. Create a different array where you store the medians of all n/5 groups in this median array

Yes, we can create a function in Racket that first divides the list into groups of size 5, sorts each group, and then finds the median of each group. The medians will be stored in a new array (list). This implementation won't rely on any library sort routines but will implement sorting from scratch, possibly using something simple like insertion sort.

Here's the complete implementation:

```
#lang racket

;; Insertion sort implementation
(define (insertion-sort lst)
  (define (insert x sorted)
    (cond
      [(empty? sorted) (list x)]
      [(< x (first sorted)) (cons x sorted)]
      [else (cons (first sorted) (insert x (rest sorted)))]))

  (foldl insert '() lst))
```

```
;; Function to find the median of a sorted list
(define (find-median sorted-group)
  (let ((n (length sorted-group)))
    (list-ref sorted-group (quotient n 2))))

;; Group the list into n/5 groups, sort them, and find medians
(define (find-medians lst)
  (define (process-groups groups)
    (map (lambda (group)
           (find-median (insertion-sort group)))  ; Sort each group and find
its median
         groups))

  ;; Group the list into chunks of 5
  (process-groups (group-into-fives lst)))

;; Group-into-fives function from earlier
(define (group-into-fives lst)
  (if (empty? lst)
      '()  ; Base case: if the list is empty, return an empty list
      (let* ((n (min 5 (length lst)))  ; Take the smaller of 5 or the
remaining list length
             (group (take lst n)))     ; Take only up to n elements
        (cons group
              (group-into-fives (drop lst n))))))  ; Drop the first n elements
and recurse

;; Example usage:
(find-medians '(1 2 3 4 5 6 7 8 9 10 11))
```

Me: can you create a function, without using library sort routines, where you sort the created n/5 groups and find the median of all groups. Create a different array where you store the medians of all n/5 groups in this median array

ChatGPT:Yes, we can create a function in Racket that first divides the list into groups of size 5, sorts each group, and then finds the median of each group. The medians will be stored in a new array (list). This implementation won't rely on any library sort routines but will implement sorting from scratch, possibly using something simple like insertion sort.

Here’s how we can proceed:

1. Create a function to sort each group (I'll use insertion sort here).

2. Use the previous group-into-fives function to divide the array into subgroups.
3. Find the median of each sorted group.
4. Collect all medians in a new array.

Here’s the complete implementation:

```
#lang racket

;; Insertion sort implementation
(define (insertion-sort lst)
  (define (insert x sorted)
    (cond
      [(empty? sorted) (list x)]
      [(< x (first sorted)) (cons x sorted)]
      [else (cons (first sorted) (insert x (rest sorted)))]))

  (foldl insert '() lst))

;; Function to find the median of a sorted list
(define (find-median sorted-group)
  (let ((n (length sorted-group)))
    (list-ref sorted-group (quotient n 2))))

;; Group the list into n/5 groups, sort them, and find medians
(define (find-medians lst)
  (define (process-groups groups)
    (map (lambda (group)
           (find-median (insertion-sort group)))  ; Sort each group and find
its median
         groups))

  ;; Group the list into chunks of 5
  (process-groups (group-into-fives lst)))

;; Group-into-fives function from earlier
(define (group-into-fives lst)
  (if (empty? lst)
      '()  ; Base case: if the list is empty, return an empty list
      (let* ((n (min 5 (length lst)))  ; Take the smaller of 5 or the
remaining list length
             (group (take lst n)))     ; Take only up to n elements
        (cons group
              (group-into-fives (drop lst n))))))  ; Drop the first n elements
and recurse
```

```
;; Example usage:
(find-medians '(1 2 3 4 5 6 7 8 9 10 11))
```

Me: what will happen if we take our array of medians and pass it through to find the median of the medians?

ChatCPT: If you take the array of medians and pass it through to find the median of the medians, the process would involve the following steps:

Array of medians: After collecting the medians from the n/5 groups, this array is essentially a smaller list compared to the original list.

Finding the median of this array: You would pass this smaller list (the medians) back through the same procedure. This would involve:

Dividing this new list of medians into groups of size 5 (or fewer).
Sorting each group.
Finding the median of each sorted group.
Continuing this recursively until a single median is found.
This process is essentially applying the median of medians algorithm recursively, which is useful in selecting a "good" pivot for algorithms like Quickselect and Quicksort.

Would you like to see the recursive code for calculating the median of medians?

Me: yes please continue with this. Please avoid using library sort routines however.

ChatGPT: Here's the full implementation:

```
#lang racket

;; Insertion sort implementation (same as before)
(define (insertion-sort lst)
  (define (insert x sorted)
    (cond
      [(empty? sorted) (list x)]
      [(< x (first sorted)) (cons x sorted)]
      [else (cons (first sorted) (insert x (rest sorted)))]))

  (foldl insert '() lst))
```

```
;; Function to find the median of a sorted list
(define (find-median sorted-group)
  (let ((n (length sorted-group)))
    (list-ref sorted-group (quotient n 2))))

;; Group the list into n/5 groups
(define (group-into-fives lst)
  (if (empty? lst)
      '()  ; Base case: if the list is empty, return an empty list
      (let* ((n (min 5 (length lst)))  ; Take the smaller of 5 or the
remaining list length
             (group (take lst n)))     ; Take only up to n elements
        (cons group
              (group-into-fives (drop lst n))))))  ; Drop the first n elements
and recurse

;; Function to compute the median of medians recursively
(define (median-of-medians lst)
  (let* ((groups (group-into-fives lst))  ; Divide list into n/5 groups
         (sorted-groups (map insertion-sort groups))  ; Sort each group
         (medians (map find-median sorted-groups)))  ; Find median of each
group
    (if (<= (length medians) 5)  ; If the list of medians is small enough
        (find-median (insertion-sort medians))  ; Sort and return median
directly
        (median-of-medians medians))))  ; Otherwise, recurse on the medians
list

;; Example usage:
(define test-list '(12 4 2 7 10 20 15 1 8 3 19 13 17 11 6 5 18 14 16 9))

;; Finding the median of medians
(median-of-medians test-list)
```

Me: So now that we have the median of medians can you set that as the pivot and perform a quicksort. Please avoid library sorting routines .

ChatGPT: Here's the implementation:

```
#lang racket

;; Function to partition a list based on a pivot
(define (partition lst pivot)
```

```
  (define less-than '())
  (define greater-than '())

  (for-each (lambda (x)
              (cond
                [(< x pivot) (set! less-than (cons x less-than))]
                [(> x pivot) (set! greater-than (cons x greater-than))]))
            lst)
  (list less-than greater-than))

;; Insertion sort implementation (same as before)
(define (insertion-sort lst)
  (define (insert x sorted)
    (cond
      [(empty? sorted) (list x)]
      [(< x (first sorted)) (cons x sorted)]
      [else (cons (first sorted) (insert x (rest sorted)))]))

  (foldl insert '() lst))

;; Median of Medians function (as defined earlier)
(define (find-median sorted-group)
  (let ((n (length sorted-group)))
    (list-ref sorted-group (quotient n 2))))

;; Group the list into n/5 groups
(define (group-into-fives lst)
  (if (empty? lst)
      '()  ; Base case: if the list is empty, return an empty list
      (let* ((n (min 5 (length lst)))  ; Take the smaller of 5 or the
remaining list length
             (group (take lst n)))     ; Take only up to n elements
        (cons group
              (group-into-fives (drop lst n))))))  ; Drop the first n elements
and recurse

;; Function to compute the median of medians recursively
(define (median-of-medians lst)
  (let* ((groups (group-into-fives lst))  ; Divide list into n/5 groups
         (sorted-groups (map insertion-sort groups))  ; Sort each group
         (medians (map find-median sorted-groups)))  ; Find median of each
group
```

```
    (if (<= (length medians) 5)  ; If the list of medians is small enough
        (find-median (insertion-sort medians))  ; Sort and return median
directly
        (median-of-medians medians))))  ; Otherwise, recurse on the medians
list

;; Quicksort function
(define (quicksort lst)
  (cond
    [(empty? lst) '()]  ; If the list is empty, return an empty list
    [(= (length lst) 1) lst]  ; A list of one element is already sorted
    [else
     (let* ((pivot (median-of-medians lst))  ; Use the median of medians as
the pivot
            (partitions (partition lst pivot))  ; Partition the list into two
parts
            (less-than (first partitions))  ; Elements smaller than pivot
            (greater-than (second partitions)))  ; Elements larger than pivot
       (append (quicksort less-than)         ; Recursively sort the smaller
elements
               (list pivot)                 ; Add the pivot in the middle
               (quicksort greater-than))))])  ; Recursively sort the larger
elements

;; Example list
(define test-list '(12 4 2 7 10 20 15 1 8 3 19 13 17 11 6 5 18 14 16 9))

;; Quicksort using median of medians as pivot
(displayln "Sorted list:")
(displayln (quicksort test-list))
```

Me: So good news, it runs. Bad news: it seems we are running out of memory when running our code.

ChatGPT:

1. Optimized Check for List Length

Here's how you might modify the group-into-fives function using an efficient check for list size:

racket
Copy code

```
(define (more-than-5-items? L)
```

```
  (let loop ([current L] [count 0])
    (cond [(null? current) #f]
          [(>= count 5) #t]
          [else (loop (cdr current) (+ count 1))])))

(define (group-into-fives lst)
  (if (null? lst)
      '()
      (if (more-than-5-items? lst)
          (let* ((n (min 5 (length lst)))
                 (group (take lst n)))
            (cons group (group-into-fives (drop lst n))))
          (list lst))))  ; If fewer than 5 items, just return the remaining
list
```

2. Carry Length as a Parameter

If you choose to carry the length as a parameter, here's how you can modify functions to use this approach:

```
racket
Copy code
(define (group-into-fives lst length)
  (if (zero? length)
      '()
      (let ((n (min 5 length)))
        (cons (take lst n)
              (group-into-fives (drop lst n) (- length n))))))

;; When calling initially
(group-into-fives your-list (length your-list))
```

This method carries the length throughout, reducing the need to call (length L) repeatedly, which can be computationally expensive for large lists.

Me: I got this error when running the code for that edited function: has-at-least-5-items?: unbound identifier in: has-at-least-5-items?

ChatGPT:

```
#lang racket
```

```
;; Function to check if the list has at least 5 items without counting all
items
(define (has-at-least-5-items? lst)
  (define (check-items remaining count)
    (cond
      [(null? remaining) #f]  ; If the end of the list is reached before count
= 5
      [(>= count 5) #t]       ; If we have counted at least 5 items, return
true
      [else (check-items (cdr remaining) (+ count 1))]))  ; Recursively check
the next item
  (check-items lst 0))  ; Start checking from the first item with a count of 0

;; Modified function to group list into sublists of up to 5 elements using the
new check
(define (group-into-fives lst)
  (if (null? lst)
      '()
      (if (has-at-least-5-items? lst)
          (let* ((group (take lst 5)))  ; Take the first 5 elements
            (cons group (group-into-fives (drop lst 5))))  ; Recur with the
remainder
          (list lst))))  ; If fewer than 5 items left, just return them as a
single group

;; Test the functionality
(define test-list '(1 2 3 4 5 6 7 8 9 10 11))
(displayln (group-into-fives test-list))  ; Expect: '((1 2 3 4 5) (6 7 8 9 10)
(11))

(define short-list '(1 2 3 4))
(displayln (group-into-fives short-list))  ; Expect: '((1 2 3 4))

(define empty-list '())
(displayln (group-into-fives empty-list))  ; Expect: '()
```

Me: This works, thanks

MID-PROJECT: ATTEMPTING TO IMPROVE CODE

Me: I have concerns regarding the runtime. It is very slow. What can you suggest to improve it?

ChatGPT: You have implemented a **median of medians** based Quicksort using lists and tested it with randomly generated integers. Your code seems well-organized.

1. **Avoid Frequent length Calls:**
    - Calling (length lst) repeatedly inside quicksort is inefficient because it traverses the entire list every time. Instead, you could calculate the length once and pass it as an argument to avoid recalculating it multiple times.

Suggested Improvements to quicksort:

```
;; Optimized Quicksort that passes the list length as a parameter
(define (quicksort lst)
  (letrec ((quicksort-helper (lambda (lst len)
                               (if (<= len 1)
                                   lst
                                   (let* ((pivot (median-of-medians lst))
                                          (partitions (partition lst pivot))
                                          (less-than (first partitions))
                                          (greater-than (second partitions)))
                                     (append (quicksort-helper less-than (length
less-than))
                                             (list pivot)
                                             (quicksort-helper greater-than (length
greater-than))))))))
    (quicksort-helper lst (length lst))))
```

This modification avoids repeated length calculations by passing the length of the sublist directly to the helper function.

Me: In this code I'm noticing the output doubles numbers:
Outcome:
Random list: (94 87 46 10 44)
Sorted list: (10 44 44 46 46 87 87 94 94)

ChatGPT:
Cause of the Issue:
When partitioning, your current code compares each element with the pivot. If the element is greater than or equal to the pivot, it is added to the greater-than list. This means that the pivot itself is being included twice: once in the greater-than list and once explicitly when you append the pivot in the recursive quicksort.

Solution:

You need to modify the partition function so that elements equal to the pivot are handled separately. Instead of putting elements equal to the pivot in both the less-than and greater-than lists, we'll add them to a separate list or exclude them from the partition entirely (since the pivot is handled separately).

Updated Code:
Here's how you can update your partition function and fix the doubling issue:

racket
Copy code

```
#lang racket

;; Function to generate a list of random integers
(define (generate-random-integers count min-value max-value)
  (define (generate n)
    (if (zero? n)
        '()
        (cons (random min-value max-value)
              (generate (- n 1)))))
  (generate count))

;; Function to partition a list based on a pivot
(define (partition lst pivot)
  (define less-than '())
  (define equal-to '())
  (define greater-than '())
  (for-each (lambda (x)
              (cond
                [(< x pivot) (set! less-than (cons x less-than))]
                [(= x pivot) (set! equal-to (cons x equal-to))]  ; Keep track of equal
elements
                [else (set! greater-than (cons x greater-than))]))
            lst)
  (list (reverse less-than) (reverse equal-to) (reverse greater-than)))  ; Reverse to
maintain input order

;; Function to find the median of a sorted list
(define (find-median sorted-group)
  (list-ref sorted-group (quotient (length sorted-group) 2)))

;; Function to check if the list has at least 5 items without counting all items
(define (has-at-least-5-items? lst)
  (define (check-items remaining count)
    (cond
      [(null? remaining) #f]  ; If the end of the list is reached before count = 5
      [(>= count 5) #t]       ; If we have counted at least 5 items, return true
      [else (check-items (cdr remaining) (+ count 1))]))  ; Recursively check the next
item
  (check-items lst 0))  ; Start checking from the first item with a count of 0
```

```
;; Modified function to group list into sublists of up to 5 elements using the new check
(define (group-into-fives lst)
  (if (null? lst)
      '()
      (if (has-at-least-5-items? lst)
          (let* ((group (take lst 5)))  ; Take the first 5 elements
            (cons group (group-into-fives (drop lst 5))))  ; Recur with the remainder
          (list lst))))  ; If fewer than 5 items left, just return them as a single group

;; Insertion sort implementation
(define (insertion-sort lst)
  (define (insert x sorted)
    (cond
      [(empty? sorted) (list x)]
      [(< x (first sorted)) (cons x sorted)]
      [else (cons (first sorted) (insert x (rest sorted)))]))
  (foldl insert '() lst))

;; Function to compute the median of medians recursively
(define (median-of-medians lst)
  (let* ((groups (group-into-fives lst))
         (sorted-groups (map insertion-sort groups))
         (medians (map find-median sorted-groups)))
    (if (<= (length medians) 5)
        (find-median (insertion-sort medians))
        (median-of-medians medians))))

;; Optimized Quicksort that passes the list length as a parameter
(define (quicksort lst)
  (letrec ((quicksort-helper (lambda (lst len)
                               (if (<= len 1)
                                   lst
                                   (let* ((pivot (median-of-medians lst))
                                          (partitions (partition lst pivot))
                                          (less-than (first partitions))
                                          (equal-to (second partitions))
                                          (greater-than (third partitions)))
                                     (append (quicksort-helper less-than (length
less-than))
                                             (reverse equal-to)  ; Append all elements
equal to pivot
                                             (quicksort-helper greater-than (length
greater-than))))))))
    (quicksort-helper lst (length lst))))

;; Test quicksort with random integers
(define test-list (generate-random-integers 5 1 100))
(displayln "Random list:")
```

```
(displayln test-list)
(displayln "Sorted list:")
(displayln (quicksort test-list))
```

Me: Can we optimize the partition function?

ChatGPT:

Full Optimized Code with Partition Improvements:

```
#lang racket

;; Function to generate a list of random integers
(define (generate-random-integers count min-value max-value)
  (define (generate n)
    (if (zero? n)
        '()
        (cons (random min-value max-value)
              (generate (- n 1)))))
  (generate count))

;; Optimized partition function
(define (partition lst pivot)
  (define (loop lst less less-len equal equal-len greater greater-len)
    (cond
      [(null? lst)
       (list (reverse less) less-len (reverse equal) equal-len (reverse greater)
greater-len)]
      [(< (first lst) pivot)
       (loop (rest lst) (cons (first lst) less) (+ less-len 1) equal equal-len greater
greater-len)]
      [(= (first lst) pivot)
       (loop (rest lst) less less-len (cons (first lst) equal) (+ equal-len 1) greater
greater-len)]
      [else
       (loop (rest lst) less less-len equal equal-len (cons (first lst) greater) (+
greater-len 1))]))
  (loop lst '() 0 '() 0 '() 0))

;; Function to find the median of a sorted list
(define (find-median sorted-group)
  (list-ref sorted-group (quotient (length sorted-group) 2)))

;; Function to check if the list has at least 5 items without counting all items
(define (has-at-least-5-items? lst)
  (define (check-items remaining count)
    (cond
      [(null? remaining) #f]  ; If the end of the list is reached before count = 5
```

```
      [(>= count 5) #t]       ; If we have counted at least 5 items, return true
      [else (check-items (cdr remaining) (+ count 1))]))  ; Recursively check the next
item
  (check-items lst 0))

;; Modified function to group list into sublists of up to 5 elements using the new check
(define (group-into-fives lst)
  (if (null? lst)
      '()
      (if (has-at-least-5-items? lst)
          (let* ((group (take lst 5)))  ; Take the first 5 elements
            (cons group (group-into-fives (drop lst 5))))  ; Recur with the remainder
          (list lst))))  ; If fewer than 5 items left, just return them as a single group

;; Insertion sort implementation
(define (insertion-sort lst)
  (define (insert x sorted)
    (cond
      [(empty? sorted) (list x)]
      [(< x (first sorted)) (cons x sorted)]
      [else (cons (first sorted) (insert x (rest sorted)))]))
  (foldl insert '() lst))

;; Function to compute the median of medians recursively
(define (median-of-medians lst)
  (let* ((groups (group-into-fives lst))
         (sorted-groups (map insertion-sort groups))
         (medians (map find-median sorted-groups)))
    (if (<= (length medians) 5)
        (find-median (insertion-sort medians))
        (median-of-medians medians))))

;; Optimized Quicksort that uses the optimized partition function
(define (quicksort lst)
  (letrec ((quicksort-helper (lambda (lst len)
                               (if (<= len 1)
                                   lst
                                   (let* ((pivot (median-of-medians lst))
                                          (partitions (partition lst pivot))
                                          (less-than (first partitions))
                                          (len-less-than (second partitions))
                                          (equal-to (third partitions))
                                          (len-equal-to (fourth partitions))
                                          (greater-than (fifth partitions))
                                          (len-greater-than (sixth partitions)))
                                     (append
                                      (quicksort-helper less-than len-less-than)
```

```
                                    (reverse equal-to)
                                    (quicksort-helper greater-than
len-greater-than)))))))
    (quicksort-helper lst (length lst))))

;; Test quicksort with random integers
(define test-list (generate-random-integers 5 1 100))
(displayln "Random list:")
(displayln test-list)
(displayln "Sorted list:")
(displayln (quicksort test-list))
```

Me: Tt seems it can't run 10 million elements. It can run 1 million though. Can we try avoiding repeated length calculation with my current code?

ChatGPT: We will modify your quicksort-helper to track and pass the length of the list through recursive calls. This will avoid the need to recalculate the length of each partition repeatedly, which can save significant time, especially for large lists.

Updated quicksort-helper:
racket
Copy code

```
(define (quicksort lst)
  (letrec ((quicksort-helper (lambda (lst len)
                               (if (<= len 1)
                                   lst
                                   (let* ((pivot (median-of-medians lst))
                                          (partitions (partition lst pivot))
                                          (less-than (first partitions))
                                          (equal-to (second partitions))
                                          (greater-than (third partitions))
                                          ;; Calculate lengths once
                                          (len-less-than (length less-than))
                                          (len-equal-to (length equal-to))
                                          (len-greater-than (length greater-than)))
                                     (append
                                      (quicksort-helper less-than len-less-than)
                                      (reverse equal-to)
                                      (quicksort-helper greater-than
len-greater-than)))))))
    ;; Call the helper with the initial list and its length
    (quicksort-helper lst (length lst))))
```

Me: It still can't run 10 million

ChatGPT:

To handle sorting 10 million integers with the median-of-medians algorithm without running out of memory in Racket, we need to focus on minimizing memory usage and avoiding

excessive recursion. Since recursion depth and memory allocation for lists are likely causing the issue, I'll make a few more adjustments:

Key Changes:
Use an iterative approach for the sorting loop, reducing recursive overhead and memory pressure.
Limit memory copying, especially when dealing with very large lists, to avoid memory exhaustion.
Improved Iterative Quicksort with Median of Medians
The following version uses tail recursion more effectively to avoid memory overflows and optimizes list handling to reduce memory copying:

racket
Copy code

```
#lang racket

;; Function to 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))

;; Optimized partition function (returns three partitions)
(define (partition lst pivot)
  (define (loop lst less equal greater)
    (cond
      [(null? lst) (list (reverse less) (reverse equal) (reverse greater))]
      [(< (first lst) pivot) (loop (rest lst) (cons (first lst) less) equal greater)]
      [(= (first lst) pivot) (loop (rest lst) less (cons (first lst) equal) greater)]
      [else (loop (rest lst) less equal (cons (first lst) greater))]))
  (loop lst '() '() '()))

;; Function to find the median of a sorted list
(define (find-median sorted-group)
  (list-ref sorted-group (quotient (length sorted-group) 2)))

;; Group list into sublists of up to 5 elements
(define (group-into-fives lst)
  (if (null? lst)
      '()
      (let ((group (take lst 5)))
        (cons group (group-into-fives (drop lst 5))))))

;; Insertion sort implementation for small lists
(define (insertion-sort lst)
  (define (insert x sorted)
```

```
    (cond
      [(empty? sorted) (list x)]
      [(< x (first sorted)) (cons x sorted)]
      [else (cons (first sorted) (insert x (rest sorted)))]))
  (foldl insert '() lst))

;; Median of medians function
(define (median-of-medians lst)
  (let* ((groups (group-into-fives lst))
         (sorted-groups (map insertion-sort groups))
         (medians (map find-median sorted-groups)))
    (if (<= (length medians) 5)
        (find-median (insertion-sort medians))
        (median-of-medians medians))))

;; Tail-recursive quicksort implementation using median of medians
(define (quicksort lst)
  (letrec ((quicksort-helper
            (lambda (lst acc)
              (if (<= (length lst) 1)
                  (append acc lst)
                  (let* ((pivot (median-of-medians lst))
                         (partitions (partition lst pivot))
                         (less-than (first partitions))
                         (equal-to (second partitions))
                         (greater-than (third partitions)))
                    (quicksort-helper greater-than
                                      (append (quicksort-helper less-than acc)
                                              equal-to)))))))
    (quicksort-helper lst '())))

;; Test quicksort with random integers
(define test-list (generate-random-integers 10000000 1 100))
(displayln "Sorted list:")
(quicksort test-list)
```

LATE-PROJECT: ADDRESSING THE MEMORY ISSUES

Me: So is there a way that I can add some sort of function which will assist in digesting 10 million or more integers? My code seems to run okay with 1 million but with anything larger.

ChatGPT:

Use of future for Parallelism:

You can implement parallelism in Racket with the future construct, which allows different parts of the computation to run concurrently.

racket

```
Copy code
;; Parallel Quicksort Helper
(define (quicksort-parallel lst)
  (letrec ((quicksort-helper (lambda (lst)
                               (if (<= (length lst) 1)
                                   lst
                                   (let* ((pivot (median-of-medians lst))
                                          (partitions (partition lst pivot))
                                          (less-than (first partitions))
                                          (equal-to (second partitions))
                                          (greater-than (third partitions))
                                          (sorted-less-than (future (lambda ()
(quicksort-helper less-than))))
                                          (sorted-greater-than (future (lambda ()
(quicksort-helper greater-than)))))
                                     (append (touch sorted-less-than)
                                             (reverse equal-to)
                                             (touch sorted-greater-than)))))))
    (quicksort-helper lst)))
```

Me: This sounds more convoluted. Disregard using Future. I have an idea. My code seems to run 1 million items okay. What if we split 10 million up into digestible chunks. So we place integers that range from let's say 1 to 9 then from 10 to 99 and so on. So we sort those out first using our quicksort median of medians approach. However, at the end we just combine them together once the groupings have been all sorted out

ChatGPT:
Let's break down the implementation step by step:

1. Define Buckets
We will first define a function that places integers into their respective ranges/buckets. Each range will be manually defined (e.g., 1-9, 10-99, 100-999, etc.).

2. Sort Each Bucket
Once numbers are placed in buckets, we'll apply your quicksort (which avoids library sorting) to each bucket.

3. Combine Sorted Buckets
After sorting, we concatenate the buckets in order to produce the final sorted list.

Here's how you can implement it:

```
#lang racket

;; Function to 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))

;; Function to partition a list based on a pivot
(define (partition lst pivot)
  (define less-than '())
  (define equal-to '())
  (define greater-than '())
  (for-each (lambda (x)
              (cond
                [(< x pivot) (set! less-than (cons x less-than))]
                [(= x pivot) (set! equal-to (cons x equal-to))]
                [else (set! greater-than (cons x greater-than))]))
            lst)
  (list (reverse less-than) (reverse equal-to) (reverse greater-than)))

;; Function to find the median of a sorted list
(define (find-median sorted-group)
  (list-ref sorted-group (quotient (length sorted-group) 2)))

;; Function to check if the list has at least 5 items without counting all items
(define (has-at-least-5-items? lst)
  (define (check-items remaining count)
    (cond
      [(null? remaining) #f]
      [(>= count 5) #t]
      [else (check-items (cdr remaining) (+ count 1))]))
  (check-items lst 0))

;; Modified function to group list into sublists of up to 5 elements using the new check
(define (group-into-fives lst)
  (if (null? lst)
      '()
      (if (has-at-least-5-items? lst)
          (let* ((group (take lst 5)))
            (cons group (group-into-fives (drop lst 5))))
          (list lst))))

;; Insertion sort implementation
(define (insertion-sort lst)
  (define (insert x sorted)
    (cond
      [(empty? sorted) (list x)]
      [(< x (first sorted)) (cons x sorted)]
      [else (cons (first sorted) (insert x (rest sorted)))]))
  (foldl insert '() lst))

;; Function to compute the median of medians recursively
```

```
(define (median-of-medians lst)
  (let* ((groups (group-into-fives lst))
         (sorted-groups (map insertion-sort groups))
         (medians (map find-median sorted-groups)))
    (if (<= (length medians) 5)
        (find-median (insertion-sort medians))
        (median-of-medians medians))))

;; Optimized Quicksort that passes the list length as a parameter
(define (quicksort lst)
  (letrec ((quicksort-helper (lambda (lst len)
                               (if (<= len 1)
                                   lst
                                   (let* ((pivot (median-of-medians lst))
                                          (partitions (partition lst pivot))
                                          (less-than (first partitions))
                                          (equal-to (second partitions))
                                          (greater-than (third partitions)))
                                     (append (quicksort-helper less-than (length
less-than))
                                             equal-to
                                             (quicksort-helper greater-than (length
greater-than))))))))
    (quicksort-helper lst (length lst))))

;; Function to distribute integers into different buckets based on value ranges
(define (bucketize lst)
  (define bucket1 '())  ; 1-999
  (define bucket2 '())  ; 1000-9999
  (define bucket3 '())  ; 10000-99999
  (define bucket4 '())  ; 100000-999999
  (define bucket5 '())  ; 1000000-9999999
  (define bucket6 '())  ; 10000000 and above

  ;; Iterate through the list and place each integer into the correct bucket
  (for-each (lambda (x)
              (cond
                [(<= x 999) (set! bucket1 (cons x bucket1))]
                [(<= x 9999) (set! bucket2 (cons x bucket2))]
                [(<= x 99999) (set! bucket3 (cons x bucket3))]
                [(<= x 999999) (set! bucket4 (cons x bucket4))]
                [(<= x 9999999) (set! bucket5 (cons x bucket5))]
                [else (set! bucket6 (cons x bucket6))]))
            lst)

  ;; Return all buckets as a list of lists
  (list (reverse bucket1) (reverse bucket2) (reverse bucket3) (reverse bucket4) (reverse
bucket5) (reverse bucket6)))
```

```
;; Function to sort the buckets and combine the results
(define (bucket-sort lst)
  (let* ((buckets (bucketize lst))  ; Split list into buckets
         (sorted-buckets (map quicksort buckets)))  ; Sort each bucket using quicksort
    (apply append sorted-buckets)))  ; Concatenate all sorted buckets

;; Test bucket sort with random integers
(define test-list (generate-random-integers 10000000 1 10000000))
(displayln "Sorting...")
(displayln (bucket-sort test-list))
```

Me: What can be done for the optimization for bucket sizes?

ChatGPT:
Below is your updated code with adaptive bucketization and recursive splitting for large buckets:

```
#lang racket

;; Function to 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 (+ 1 (- max-value min-value))))
              (generate (- n 1)))))
  (generate count))

;; Function to partition a list based on a pivot
(define (partition lst pivot)
  (define less-than '())
  (define equal-to '())
  (define greater-than '())
  (for-each (lambda (x)
              (cond
                [(< x pivot) (set! less-than (cons x less-than))]
                [(= x pivot) (set! equal-to (cons x equal-to))]
                [else (set! greater-than (cons x greater-than))]))
            lst)
  (list (reverse less-than) (reverse equal-to) (reverse greater-than)))

;; Function to find the median of a sorted list
(define (find-median sorted-group)
  (list-ref sorted-group (quotient (length sorted-group) 2)))

;; Function to check if the list has at least 5 items without counting all items
(define (has-at-least-5-items? lst)
  (define (check-items remaining count)
    (cond
```

```
      [(null? remaining) #f]
      [(>= count 5) #t]
      [else (check-items (cdr remaining) (+ count 1))]))
  (check-items lst 0))

;; Modified function to group list into sublists of up to 5 elements using the new check
(define (group-into-fives lst)
  (if (null? lst)
      '()
      (if (has-at-least-5-items? lst)
          (let* ((group (take lst 5)))
            (cons group (group-into-fives (drop lst 5))))
          (list lst))))

;; Insertion sort implementation
(define (insertion-sort lst)
  (define (insert x sorted)
    (cond
      [(empty? sorted) (list x)]
      [(< x (first sorted)) (cons x sorted)]
      [else (cons (first sorted) (insert x (rest sorted)))]))
  (foldl insert '() lst))

;; Function to compute the median of medians recursively
(define (median-of-medians lst)
  (let* ((groups (group-into-fives lst))
         (sorted-groups (map insertion-sort groups))
         (medians (map find-median sorted-groups)))
    (if (<= (length medians) 5)
        (find-median (insertion-sort medians))
        (median-of-medians medians))))

;; Optimized Quicksort that passes the list length as a parameter
(define (quicksort lst)
  (letrec ((quicksort-helper (lambda (lst len)
                               (if (<= len 1)
                                   lst
                                   (let* ((pivot (median-of-medians lst))
                                          (partitions (partition lst pivot))
                                          (less-than (first partitions))
                                          (equal-to (second partitions))
                                          (greater-than (third partitions)))
                                     (append (quicksort-helper less-than (length
less-than))
                                             equal-to
                                             (quicksort-helper greater-than (length
greater-than)))))))))
    (quicksort-helper lst (length lst))))
```

```
;; Function to compute min and max values of a list
(define (find-min-max lst)
  (foldl (lambda (x acc)
           (list (min (first acc) x) (max (second acc) x)))
         (list (first lst) (first lst))
         (rest lst)))

;; Adaptive bucketize function based on data range
(define (adaptive-bucketize lst num-buckets)
  (if (null? lst)
      '()
      (let* ((min-max (find-min-max lst))
             (min-val (first min-max))
             (max-val (second min-max))
             (range (+ 1 (- max-val min-val)))  ; Calculate the total range
             (bucket-size (max 1 (/ range num-buckets)))  ; Determine the size of each
bucket
             (buckets (make-vector num-buckets '()))  ; Create an empty vector for
buckets

        ;; Distribute each element into the appropriate bucket
        (for-each (lambda (x)
                    (let* ((bucket-index (min (quotient (- x min-val) bucket-size)
                                              (- num-buckets 1))))  ; Ensure index stays
within bounds
                      (vector-set! buckets bucket-index
                                   (cons x (vector-ref buckets bucket-index)))))
                  lst)
        (vector->list (map reverse (vector->list buckets))))))  ; Return buckets as a list
of lists

;; Function to recursively split large buckets
(define (split-large-buckets buckets threshold num-buckets)
  (map (lambda (bucket)
         (if (> (length bucket) threshold)
             (apply append (adaptive-bucketize bucket num-buckets))  ; Further split large
bucket
             bucket))  ; Keep small buckets as they are
       buckets))

;; Function to sort the buckets and combine the results using adaptive bucketing
(define (bucket-sort lst)
  (let* ((num-buckets 10)  ; Define the number of initial buckets
         (threshold 1000)   ; Define a threshold for splitting large buckets
         (initial-buckets (adaptive-bucketize lst num-buckets))  ; First pass of
bucketization
         (final-buckets (split-large-buckets initial-buckets threshold num-buckets))  ;
Split large buckets
```

```
        (sorted-buckets (map quicksort final-buckets)))  ; Sort each bucket using
quicksort
    (apply append sorted-buckets)))  ; Concatenate all sorted buckets

;; Test bucket sort with random integers and add timer
(define test-list (generate-random-integers 400003 1 1000000))

(define start-time (current-inexact-milliseconds))  ; Record the start time

(displayln "Sorting...")
(define result (bucket-sort test-list))  ; Run the sorting algorithm

(define end-time (current-inexact-milliseconds))  ; Record the end time
(define elapsed-time (/ (- end-time start-time) 1000))  ; Calculate elapsed time in
seconds

(displayln (string-append "Elapsed time: " (number->string elapsed-time) " seconds")) ;
Display elapsed time
```

Me: This error keeps occurring: vector->list: contract violation expected: vector?

ChatGPT: The error you're seeing suggests that vector->list is being passed something other than a vector. In the adaptive-bucketize function, we create a vector of buckets using make-vector, which should indeed be a vector. However, the issue might arise from how we're passing the argument to vector->list.

Here's a revised version of the adaptive-bucketize function, ensuring that the conversion from vector to list is handled correctly:

Revised adaptive-bucketize Function:

```
racket
Copy code
(define (adaptive-bucketize lst num-buckets)
  (if (null? lst)
      '()
      (let* ((min-max (find-min-max lst))
             (min-val (first min-max))
             (max-val (second min-max))
             (range (+ 1 (- max-val min-val)))  ; Calculate the total range
             (bucket-size (max 1 (/ range num-buckets)))  ; Determine the size of each
bucket
             (buckets (make-vector num-buckets '())))  ; Create an empty vector for
buckets

        ;; Distribute each element into the appropriate bucket
        (for-each (lambda (x)
                    (let* ((bucket-index (min (quotient (- x min-val) bucket-size)
```

```
                                          (- num-buckets 1))))  ; Ensure index stays
within bounds
                    (vector-set! buckets bucket-index
                                 (cons x (vector-ref buckets bucket-index)))))
                lst)

        ;; Convert each bucket in the vector to a list and return a list of lists
        (map reverse (vector->list buckets)))))  ; Apply reverse and convert vector of
buckets to a list
```

TESTING

Me: can you create small/quick tests that are centered around testing for small sublists, edge cases, and duplicates?

ChatGPT:

Here are test cases designed to verify the behavior of your code, specifically focused on testing small sublists, edge cases (empty lists, single-element lists, lists with duplicates), and ensuring that everything works as expected. These tests will help ensure your implementation can handle these scenarios properly.

Test Cases for Small Sublists, Edge Cases, and Duplicates

```
#lang racket

;; Test case for small sublists
(define (test-small-sublists)
  (let* ((small-list '(1 2 3 4)) ; A list with fewer than 5 elements
         (result (group-into-fives small-list)))
    (displayln "Test Small Sublists:")
    (displayln (string-append "Result: " (format "~a" result)))
    (displayln (string-append (if (equal? result '((1 2 3 4))) "Test passed" "Test
failed")
                              " - Expected result: '((1 2 3 4))"))))

;; Test case for empty list
(define (test-empty-list)
  (let* ((empty-list '()) ; An empty list
         (result (quicksort empty-list)))
    (displayln "Test Empty List:")
    (displayln (string-append "Result: " (format "~a" result)))
    (displayln (string-append (if (equal? result '()) "Test passed" "Test failed")
                              " - Expected result: '()"))))

;; Test case for single-element list
(define (test-single-element-list)
  (let* ((single-element-list '(42)) ; A list with one element
```

```
         (result (quicksort single-element-list)))
    (displayln "Test Single Element List:")
    (displayln (string-append "Result: " (format "~a" result)))
    (displayln (string-append (if (equal? result '(42)) "Test passed" "Test failed")
                              " - Expected result: '(42)"))))

;; Test case for duplicates in the list
(define (test-duplicates)
  (let* ((duplicate-list '(5 3 5 7 5 2 5 1 5)) ; A list with many duplicate values
         (result (quicksort duplicate-list)))
    (displayln "Test Duplicates List:")
    (displayln (string-append "Result: " (format "~a" result)))
    (displayln (string-append (if (equal? result '(1 2 3 5 5 5 5 5 7)) "Test passed" "Test
failed")
                              " - Expected result: '(1 2 3 5 5 5 5 5 7)"))))

;; Test case for multiple sublists (more than 5 elements)
(define (test-multiple-sublists)
  (let* ((large-list '(10 9 8 7 6 5 4 3 2 1)) ; A list with more than 5 elements
         (grouped-result (group-into-fives large-list)))
    (displayln "Test Multiple Sublists (Group into Fives):")
    (displayln (string-append "Result: " (format "~a" grouped-result)))
    (displayln (string-append (if (equal? grouped-result '((10 9 8 7 6) (5 4 3 2 1)))
                                  "Test passed"
                                  "Test failed")
                              " - Expected result: '((10 9 8 7 6) (5 4 3 2 1))"))))

;; Test case for handling of list with fewer than 5 medians
(define (test-fewer-than-five-medians)
  (let* ((small-list '(3 1 2 5 4))
         (result (median-of-medians small-list)))
    (displayln "Test Median of Medians (Fewer than Five Medians):")
    (displayln (string-append "Result: " (format "~a" result)))
    (displayln (string-append (if (equal? result 3) "Test passed" "Test failed")
                              " - Expected result: 3"))))

;; Run the tests
(test-small-sublists)
(test-empty-list)
(test-single-element-list)
(test-duplicates)
(test-multiple-sublists)
(test-fewer-than-five-medians)
```