DISCUSSION

A document discussing, for each block of code, how close to correct the initial code was, what changes you needed to make, etc. Basically, walk me through the process of working with the LLM. DO NOT JUST DESCRIBE WHAT THE CODE IS DOING. I can see what the code is doing, and any LLM can generate (badly-written) descriptions of what the code is doing (that might even be correct).

1. Generating Random Integers (generate-random-integers)

Initial:

```
(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))
```

Final:
No changes

This portion of source code came from the assignment text. It was kept the same throughout development. No issues came with it. Because it generates random numbers, it helped me see that my code needed better solutions to handle larger lists of numbers.

2. Partitioning the List (partition)

Initial:

```
(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))
```

Final:

```
(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)))
```

The initial partition function didn't account for duplicates as a result of the random generation of numbers, which caused some elements to be added to both the less-than and greater-than partitions, resulting in incorrect sorting. I asked the LLM to therefore optimize it. It suggested a fix by adding an `equal-to` list to handle elements equal to the pivot. This resolved the issue with duplicates and allowed the pivot to be correctly handled.

The LLM provided a solution that was easy to integrate. After this change, the partition function worked as expected, and it was a smooth fix that didn't affect the rest of the code.

3. Finding the Median of a Sorted Group (find-median)

Initial:

```
(define (find-median sorted-group)
  (list-ref sorted-group (quotient (length sorted-group) 2)))
```

Final:
No changes.

The median-finding function was correct from the start. The LLM used a straightforward approach by finding the middle element of the sorted group using `list-ref`. It didn't need any changes, and the code worked efficiently for the task. This was one of those parts where the LLM's code was clean and easy to understand, making it simple to integrate into the overall sorting process.

4. Grouping the List into Sublists of 5 (group-into-fives)

Initial:

```
(define (group-into-fives lst)
  (if (empty? lst)
      '()
      (cons (take lst 5) (group-into-fives (drop lst 5)))))
```

Final

```
(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))))
```

The original version of this function worked fine until the list had fewer than 5 elements left at the end, which caused an error. The LLM suggested adding a check using `has-at-least-5-items?` to handle this case, which fixed the issue. This was a necessary fix, and the LLM's solution was straightforward. It didn't require major changes and integrated easily into the program without affecting other components.

5. Insertion Sort (insertion-sort)

Initial:

```
(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))
```

Final:
No changes.

The insertion sort implementation provided by the LLM was good from the beginning. It used a simple recursive approach to insert elements into a sorted list, and since it was only used on small sublists of 5 elements, it worked efficiently. There were no changes needed, and it was easy to integrate this into the overall median-of-medians algorithm.

6. Median of Medians Algorithm (median-of-medians)

Initial:

```
(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))))
```

Final:
No changes.

The median-of-medians algorithm was mostly correct from the beginning, though the biggest headache was optimization for performance. The LLM did a good job of handling the base cases and recursion, but after more prompting about efficiency, it refined the recursive calls to make them more efficient, especially for larger datasets. The final version worked well and was easy to integrate with the quicksort implementation.

7. Optimized Quicksort (quicksort)

Initial:

```
(define (quicksort lst)
  (cond
    [(empty? lst) '()]
    [(= (length lst) 1) lst]
    [else
     (let* ((pivot (median-of-medians lst))
            (partitions (partition lst pivot))
            (less-than (first partitions))
            (greater-than (second partitions)))
       (append (quicksort less-than) (list pivot) (quicksort
greater-than)))]))
```

Final:

```
(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))))
```

The original quicksort implementation worked, but performance issues arose when sorting large datasets. After I brought this up, the LLM suggested optimizing the recursive calls by passing the list length as a parameter instead of recalculating it each time. This change improved both performance and memory usage, especially for large datasets. The LLM's suggestion made the quicksort function more efficient, and the updated version was easy to integrate.

8. Find Min-Max (find-min-max)

Initial:

This function was added later during the bucketization process

Final:

```
(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)))
```

The `find-min-max` function was added when I needed to dynamically adjust the bucket sizes for better performance during bucketization. The LLM provided a clean implementation using `foldl` to compute the minimum and maximum values of the list. This function worked well and didn't require any modifications. It was easy to integrate into the bucketization process and made the overall sorting system more efficient.

9. Adaptive Bucketize (adaptive-bucketize)

Initial:

```
(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))
```

Final:

```
(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 (floor (/ range num-buckets))))  ; Ensure
bucket-size is an integer
             (buckets (make-vector num-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))))
                      (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)))))
```

This function was one of the most essential portions for improving performance when sorting large datasets. Initially, the bucketization process wasn't dynamic enough. The runtime overall before this implementation was 35 seconds roughly. So I prompted the LLM to come up with a more adaptive approach. The LLM suggested this version, which created buckets based on the range of values in the list. This worked much better for evenly distributing elements across buckets and made the sorting process more scalable. The overall runtime was reduced to 29-30 seconds.

10. Splitting Large Buckets (split-large-buckets)

Initial:

This function didn't exist initially, but became necessary once bucket sorting needed to handle large datasets

Final:

```
(define (split-large-buckets buckets threshold num-buckets)
  (map (lambda (bucket)
         (if (> (length bucket) threshold)
             (apply append (adaptive-bucketize bucket num-buckets))
             bucket))
       buckets))
```

After implementing the bucketization, I noticed that some buckets were still too large. I asked the LLM for a solution, and it suggested further splitting these large buckets using the adaptive bucketization method. This change occurred at the same time as the adaptive buckets above. This function was a helpful addition, and it made sorting large datasets much more efficient. The code fit in well with the existing bucketize function and didn't require any

major changes. I noticed the optimization of code using recursive methods was helpful by the LLM.

11. Sorting the Buckets (bucket-sort)

Initial:

This was introduced later to improve sorting performance for large datasets

Final:

```
(define (bucket-sort lst)
  (let* ((num-buckets 10)
         (threshold 1000)
         (initial-buckets (adaptive-bucketize lst num-buckets))
         (final-buckets (split-large-buckets initial-buckets threshold
num-buckets))
         (sorted-buckets (map quicksort final-buckets)))
    (apply append sorted-buckets)))
```

The `bucket-sort` function was developed later in the process to improve sorting performance for very large datasets. The LLM helped design a bucket-sorting strategy that worked well with the adaptive bucketization and quicksort functions. This approach efficiently handled datasets of up to 10 million elements by breaking the list into smaller, more manageable chunks. Integrating this function into the rest of the program was straightforward, and it greatly improved performance.

Conclusion:

Overall, the LLM produced decent code, but some functions required refinement, especially for handling large datasets and memory usage. In the context of doing all of this in a new language I have never used, I am fairly impressed with the LLM. However, it has its issues.

In retrospect, I believe that in order to achieve the working code I have now, one would have to approach in a stepwise manner and build a solid foundation of basic functions that create a logical, clear, and concise vision of what the user wants to develop. Asking the LLM to develop large chunks of code leads to various logical errors and syntax issues. I did find that asking the LLM questions (like asking a smart friend) assisted greatly. It provided me options on how to proceed when I was stuck, especially when it came to overcoming the memory usage problem for 10 million integers. However, that also had its limitations. At one point it recommended me to use a future

construct which overall made the code hard to follow and unhelpful. Eventually, I had to think outside the generation of the LLM. I found that trying to pry deeper into optimizing the same function over and over, led to unhelpful results. So, I introduced the concept of buckets to the LLM which provided a whole new avenue of options. In the end this helped me produce a working product while keeping the original constraints (no sort library routines, avoiding length, quicksort via median of medians partitioning).

LLMs are an amazing tool for programmers that help speed up the process of creating workable source code. Nonetheless, the introduction of buckets showed me that as programmers, we still need to be creative with what we think to write and cannot rely on AI to generate the entire solution for us.