CS 441 Program 1: Code Discussion

Kyle Naluan

Overall Observations + Notes:

- LLM Used: ChatGPT 4o
- LLM performed better with smaller, simpler functions, but slightly more complex functions didn't have many issues either.
- I ran into less errors and issues than I thought I would heading into this program.
- LLM would sometimes make small logic errors & other mistakes
  - LLM would claim that code did one thing, but it didn't when tested.
  - LLM sometimes pointed out errors and generated 'new' code but gave the same exact code as before that was wrong.
- Approach:
  - Build smaller functions first and test, then use them to build the bigger functions, like the median of medians partitioning and quick sort.
  - I thought that doing it this way would be more efficient and organized. Instead of having the LLM build an entire program all at once, I could do it piece by piece and make sure everything is working before moving on to the next step.
  - After initially completing the functionality of the quick sort program, I asked the LLM for various optimizations for my code. I didn't use all of the suggestions, since some would remove key features, like median of medians partitioning, some utility functions, etc.

Utility Code

Find max and find min:

```
(define (find-max lst)
  (if (null? lst)
      (error "The list is empty")
      (foldl max (first lst) (rest lst))))
```

```
(define (find-min lst)
  (if (null? lst)
      (error "The list is empty")
      (foldl min (first lst) (rest lst))))
```

- These functions were very simple and straightforward. They worked fine when tested and there was no need to make any modifications.
- I initially used the find-max function, but ended up replacing it with find-min, which will be discussed under the selection-sort function discussion.

Filter Function

Original Response

```
(define (categorize value lst)
  (define less-than '())
  (define equal-to '())
  (define greater-than '())
  (for-each
    (lambda (x)
      (cond
        [(< x value) (set! less-than (cons x less-than))]
        [(= x value) (set! equal-to (cons x equal-to))]
        [(> x value) (set! greater-than (cons x greater-than))]))
    lst)
  (list (reverse less-than) (reverse equal-to) (reverse greater-than)))
```

New/Modified Response

```
(define (filter-by-comparison comp-fn value lst)
  (filter (lambda (x) (comp-fn x value)) lst))
```

- I originally misinterpreted the purpose of the filter function and asked the LLM the following prompt:
  - Write a function in racket, where given a value and a list, the items will be filtered into 3 categories based on whether they are

        - less than - equal to - greater than
    - the given value.
- After reviewing the assignment instructions, I realized that the point of filtering was to get a single list based on a single chosen comparison, so I had to ask the LLM to write a completely new function.
- This new function was a lot more concise and had no issues when testing, so I didn't have to make any further changes.

Selection Sort

First Version

```
;; Selection sort using find-max and filter-by-comparison
(define (selection-sort lst)
  (define (sort-helper unsorted sorted)
    (if (null? unsorted)
        (reverse sorted)  ;; Return the sorted list in the correct order
        (let ([max-val (find-max unsorted)])
          (sort-helper (filter-by-comparison < max-val unsorted)  ;; Remove the max element
                       (cons max-val sorted)))))  ;; Add max to sorted list
  (sort-helper lst '()))  ;; Initialize the sorted list as empty
```

Second Version

```
;; Selection sort using find-min and filter-by-comparison
(define (selection-sort lst)
  (define (sort-helper unsorted sorted)
    (if (null? unsorted)
        sorted  ;; Return the sorted list directly
        (let ([min-val (find-min unsorted)])
          (sort-helper (filter-by-comparison > min-val unsorted)  ;; Remove the min element
                       (cons min-val sorted)))))  ;; Add min to sorted list
  (reverse (sort-helper lst '())))  ;; Reverse the result to get ascending order
```

Final Optimized Version

```
(define (selection-sort lst)
  (define (sort-helper unsorted sorted)
    (if (null? unsorted)
        (reverse sorted)
        (let ([min-val (find-min unsorted)])
          (sort-helper (filter-by-comparison > min-val unsorted)
                       (cons min-val sorted)))))
  (sort-helper lst '()))
```

- As stated earlier, I originally implemented the find-max function instead of find-min. So, I asked the LLM to write a selection-sort algorithm using the find-max and filter-by-comparison functions it previously wrote.

- After giving me the first version of code, ChatGPT stated afterwards that it would output the sorted list in ascending order, but when I tested it, the list output was in descending order.
- I asked ChatGPT to fix this, and it suggested using find-min instead. I'm sure that find-max would still work though, since the only adjustment the second version did was replace max functions/values with min, and add a reverse function to the final list (I tested the new version w/o the reverse function and the output was also in descending order).
- I used the second version of the selection-sort until I completed the entire program's functionality. Then I asked the LLM for any possible optimizations. One of the suggestions was to use tail recursion to limit stack calls, which involved moving the reverse function into the if statement.

Is-sorted Function:

Original

```
(define (is-sorted lst)
  (cond
    [(or (null? lst) (null? (rest lst))) #t]  ;; Empty or single element list is sorted
    [(> (first lst) (second lst)) #f]         ;; If first element is greater than second, not sorted
    [else (is-sorted (rest lst))]))           ;; Recursively check the rest of the list
```

Optimized

```
(define (is-sorted lst)
  (let loop ((lst lst))
    (cond
      [(or (null? lst) (null? (rest lst))) #t]
      [(> (first lst) (second lst)) #f]
      [else (loop (rest lst))])))  ;; Converted to tail-recursive style
```

- This function was very simple, like most of the other utility functions. It appeared to work right away, so I didn't really have to make any adjustments or ask the LLM to make any modifications.
- The only time I changed it was when I asked for possible optimizations at the end, and just like the selection sort, I changed it to tail recursion.

Function to check for 5 or more items

Original

```
(define (more-than-five-items? lst)
  (cond
    [(null? lst) #f]                  ;; List is empty, so it has fewer than 5 items
    [(null? (rest lst)) #f]            ;; 1 item
    [(null? (rest (rest lst))) #f]     ;; 2 items
    [(null? (rest (rest (rest lst)))) #f]  ;; 3 items
    [(null? (rest (rest (rest (rest lst))))) #f]  ;; 4 items
    [(null? (rest (rest (rest (rest (rest lst)))))) #f]  ;; 5 items
    [else #t]))  ;; If we get past the 5th item, return true
```

‘Optimized’ Version

```
(define (more-than-five-items? lst)
  (and (pair? lst)
       (pair? (rest lst))
       (pair? (rest (rest lst)))
       (pair? (rest (rest (rest lst))))
       (pair? (rest (rest (rest (rest lst)))))))
```

Actual Optimized Version

```
(define (more-than-five-items? lst)
  (and (pair? lst)
       (pair? (rest lst))
       (pair? (rest (rest lst)))
       (pair? (rest (rest (rest lst))))
       (pair? (rest (rest (rest (rest lst)))))
       (pair? (rest (rest (rest (rest (rest lst)))))))
  )
```

- I asked the LLM to write this function to avoid using the length function when it wasn’t necessary. The first version worked correctly and I didn’t have to do any debugging, but it wasn’t the most efficient or readable.
- After completing the functionality of the quick sort, this function was also one that the LLM suggested optimizing. The optimized version reduced the amount of null checks and rest calls, and instead checked for pairs. However, when tested, this function failed to return false when tested on a list of exactly 5 elements.
- So, I asked the LLM why this would happen, and it explained that the first optimization only checked if there were up to 5 elements, so the most recent optimization added another pair call to check for a potential 6th element.

Median of Medians Functions

The following functions were all written at the same time by the LLM when I asked it to write a function for median of medians partitioning (sublists of 5), while utilizing the functions described above.

Split into sublists function

- This function performed the operation of splitting a list into sublists of 5 or less.

Original

```
(define (split-into-sublists lst)
  (define (split-helper lst sublists)
    (if (null? lst)
        (reverse sublists)
        (let ([chunk (take lst 5)])
          (split-helper (drop lst 5) (cons chunk sublists)))))
  (split-helper lst '()))
```

Fixed Version

```
(define (split-into-sublists lst)
  (define (split-helper lst sublists)
    (if (null? lst)
        (reverse sublists)  ;; When list is empty, return the accumulated sublists
        (let ([chunk (if (more-than-five-items? lst) (take lst 5) lst)])  ;; Take 5 or fewer items
          (split-helper (drop lst (min 5 (length lst)))  ;; Drop 5 or remaining length
                        (cons chunk sublists)))))
  (split-helper lst '()))
```

Optimized Version

```
(define (split-into-sublists lst)
  (define (split-helper lst sublists)
    (if (null? lst)                          ;; Base case: If the list is empty
        (reverse sublists)                   ;; Return the reversed list of sublists
        (let* ([has-more-than-five (more-than-five-items? lst)]  ;; Check if there are more than 5 items left
               [chunk (if has-more-than-five (take lst 5) lst)]  ;; Take 5 or remaining items
               [remaining (if has-more-than-five (drop lst 5) '())])  ;; Drop 5 or stop if fewer
          (split-helper remaining (cons chunk sublists)))))  ;; Recurse with remaining items
  (split-helper lst '()))  ;; Initial call with an empty sublist accumulator
```

- When the LLM gave me an initial draft, I tested it out and was met with an error. Since I was testing with a list that wasn't an exact multiple of 5 (3 extra items), there was an error because the take function was expecting a list of exactly 5 numbers.
- I knew which line was wrong, but I wanted to see if the LLM could properly identify where the code was faulty, so I asked it where the logic error was that didn't allow lists of fewer than 5.

- It was able to identify the error in the code on its first attempt, and then proceeded to write ‘corrected’ code, and explained the differences between the take and drop functions.
- However, the code it gave was the exact same as the one with the error, so I had to point out that nothing had changed.
- When it rewrote the code the second time, it successfully made a change to allow sublists of 5 or fewer, by including an if statement to check if there were more than 5 elements left in the list (using the more-than-five-items? function).
- After using the first fixed version for a while, I noticed the line
    - (split-helper (drop lst (min 5 (length lst)))
- I didn’t want to use the length function if it wasn’t necessary, since that could lead to heavy computing time.
- I asked the LLM to write an alternative that avoided using the length function, which is what led to the final optimized version, which I found to work correctly after a bit of testing.
- The LLM offered more optimization options when I asked it to look over my entire program, and the suggestion for this function reintroduced the length function.
- I tested that version just in case, but found the quick sort to take much longer for similar list sizes, so I just stuck with my initial optimizations.

Find Median function

Original

```
(define (median lst)
  (let ([n (length lst)])
    (list-ref lst (quotient n 2))))  ;; Return the middle element of a sorted list
```

Fixed

```
(define (median lst)
  (let ([n (length lst)])
    (list-ref lst (quotient (- n 1) 2))))
```

- This function was simple, so I didn’t think anything would be wrong with it.
- When I tested it though, I noticed that the index for the median wasn’t the floor of (last index/2).
    - For example, a list of ‘(1 2 3 4) would give a median of 3.
- I realized that it was because it took n (the # of items) instead of n-1 (index of last item).
- This was an easy fix, but I had the LLM try to fix it to see if it would make any mistakes.
- It acknowledged that Racket used 0-based indexing and gave the fixed version above.

Median of Medians function

Original

```
(define (median-of-medians lst)
  ;; Step 1: Split the list into sublists of 5 elements
  (let ([sublists (split-into-sublists lst)])
    ;; Step 2: Sort each sublist and find its median
    (define medians
      (map (lambda (sublist) (median (selection-sort sublist))) sublists))
    ;; Step 3: If the list of medians has more than 5 items, recursively apply median-of-medians
    (if (more-than-five-items? medians)
        (median-of-medians medians)
        ;; Otherwise, return the median of the medians
        (median (selection-sort medians)))))
```

- This is one of the functions that worked correctly the entire time I used it, and I didn't have to change at all.
- Any problems that I might have encountered stemmed from the smaller functions that it was made up of (such as split-into-sublists and median).
- When I asked for optimizations, none of the suggestions actually worked or fit the assignment criteria.
    - One suggested that the last statement only needed to be (median medians) and that the medians list didn't need to be sorted if there were fewer than five elements. This was wrong, since even a list of 2 elements could be unsorted, and I proved it through testing.
    - Another suggestion involved the use of a built in sorting method.
- Because of this, the median of medians function is actually one that I left untouched.

Quick Sort Function

Original

```
(define (quicksort lst)
  (if (null? lst)  ;; Base case: an empty list is already sorted
      '()
      (let* ([pivot (median-of-medians lst)]  ;; Step 1: Choose pivot using median of medians
             [less (filter (lambda (x) (< x pivot)) lst)]  ;; Step 2: Partition list
             [equal (filter (lambda (x) (= x pivot)) lst)]
             [greater (filter (lambda (x) (> x pivot)) lst)])
        ;; Step 3: Recursively sort the "less" and "greater" partitions and concatenate results
        (append (quicksort less) equal (quicksort greater)))))
```

Suggested Optimization (not used)

```
(define (quicksort lst)
  (if (null? lst)
      '()
      (let* ([pivot (median-of-medians lst)]
             [partitions (foldl
                          (lambda (x acc)
                            (let* ([less (first acc)]
                                   [equal (second acc)]
                                   [greater (third acc)])
                              (cond
                                [(< x pivot) (list (cons x less) equal greater)]
                                [(= x pivot) (list less (cons x equal) greater)]
                                [(> x pivot) (list less equal (cons x greater))])))
                          (list '() '() '())  ;; Initialize less, equal, greater as empty lists
                          lst)])
        ;; Destructure the partitions and recursively sort
        (let ([less (first partitions)]
              [equal (second partitions)]
              [greater (third partitions)])
          (append (quicksort less) (reverse equal) (quicksort greater))))))
```

- The original version of quick sort worked fine and I found that it actually had no issues running.
- However, I wanted to see if I could achieve faster runtimes with lists of larger numbers, so I asked the LLM for possible optimizations for this function.
- I didn't use some suggestions for the following reasons:
  - Changing method of getting pivot
  - Using different data structures (vectors)
  - Using a hybrid quicksort
  - Etc.
- I did end up choosing one optimization to try. The LLM explained that the original function made multiple passes over the list for the less, equal, and greater partitions. It claimed to optimize the

function by traversing the list only once and partitioning in a single pass, using foldl and let-values.

  - This optimization originally had an error with lambda & multiple return values, but the LLM fixed it by accumulating the results in lists.

- I tried this optimization, but found through testing that the original variation consistently outperformed the 'optimized' version, so I stuck with the original.
  - The original sorted a list of 10+ million elements around 3 seconds faster than the 'optimized' function on average.
- The quick sort function was another that I didn't find myself changing.

Testing Notes

- Tests were done on each utility function, covering multiple cases.
- I also tested the median & median of medians functions on small data sets, and compared them to my own calculations.
- For the quick sort function, I tested it multiple times using the random number generator on lists of size:
    - 4
    - 43
    - 403
    - 400,003
    - 10,000,003
- These lists included numbers ranging from 1-100
- For the first couple list sizes [4, 43] and another hand typed list, I printed out the sorted lists to show that the sorting algorithm worked.
- For the higher-sized lists, I printed the result of (is-sorted (quicksort (number generation))) since there were too many numbers to display.
- For [4 - 400,003] the computing times were pretty much instant (400,003 took slightly longer).
- For the list of 10,000,003 items, I found that the total computation time was around 30-40 seconds, but this included the time it took to generate the numbers, as well as run the is-sorted function over the list. I'm approximating that the actual time to run the quicksort algorithm is around half of this time, since it took a while for me to just generate 10 million numbers when testing the generation function.