

Continuation for large data list input:

Final Document with correct code:

This was my final code before- but it only worked for limited data list inputs, lets say it worked for Data-1.txt to Data-6.txt files and it didnt sort the list. So, I started asking LLM again for the correct code which runs for 10M+ input list and produce a sorted list.

Previous code-

#lang racket

```
; Function to read integers from a file
(define (read-integers-from-file filename)
  (with-input-from-file filename
    (lambda ()
      (let loop ([integers '()])
        (let ([num (read)])
          (if (eof-object? num)
              (reverse integers) ; Return the list of integers in the correct order
              (loop (cons num integers)))))) ; Add the number to the list and continue

; Function to create value-frequency pairs
(define (create-frequency-pairs integers)
  (let loop ([lst integers] [freq-map (hash)]) ; Start with an empty hash map
    (if (null? lst)
        (hash->list freq-map) ; Convert the hash map to a list of pairs
        (let ([num (car lst)])
          (loop (cdr lst) (hash-update freq-map num add1 0)))))) ; Update the frequency count

;; Manual sorting function (using insertion sort)
(define (manual-sort pairs)
  (define (insert x sorted)
    (if (null? sorted)
        (list x) ; If the list is empty, just add x
        (let ([current (car sorted)])
          (if (< (car x) (car current)) ; Compare the first element of the pairs
              (cons x sorted) ; Insert x before the current element
              (cons current (insert x (cdr sorted)))))) ; Keep the current element and insert x into the
    rest
```

```
(foldl insert '() pairs)) ; Insert each element into the sorted list

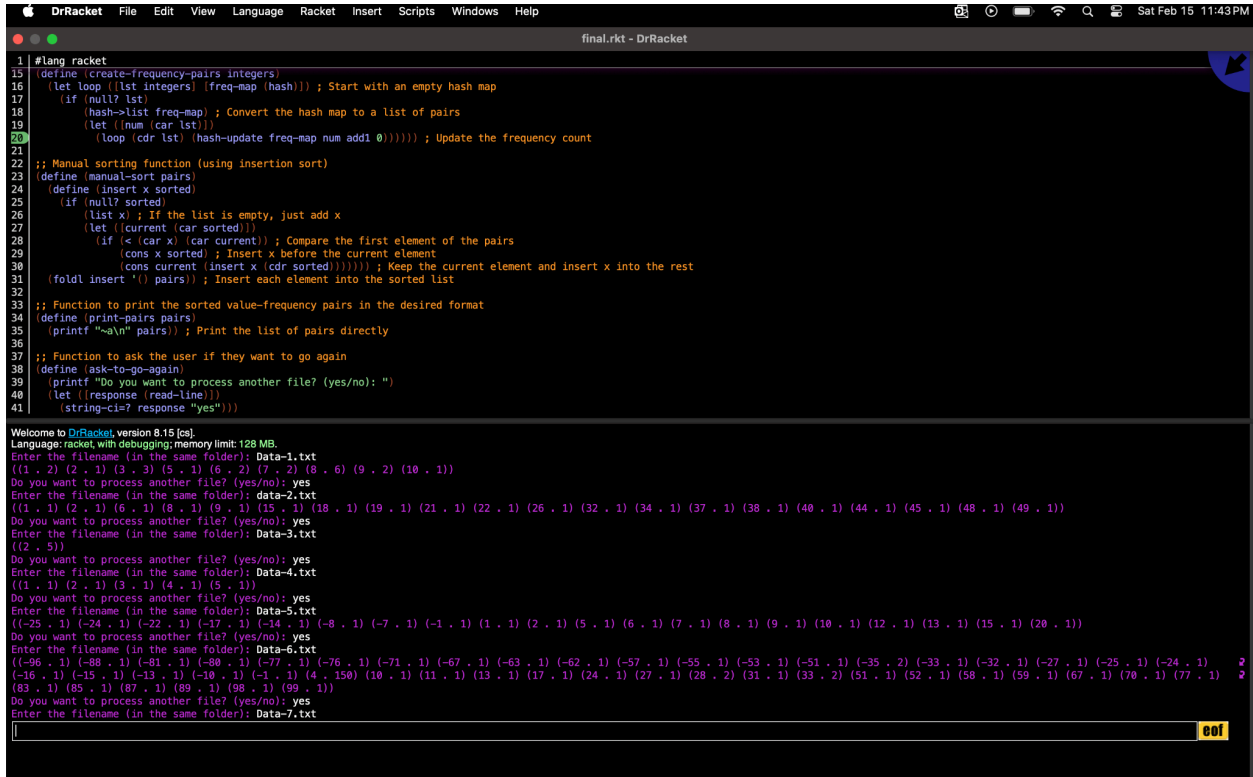
;; Function to print the sorted value-frequency pairs in the desired format
(define (print-pairs pairs)
  (printf "~a\n" pairs)) ; Print the list of pairs directly

;; Function to ask the user if they want to go again
(define (ask-to-go-again)
  (printf "Do you want to process another file? (yes/no): ")
  (let ([response (read-line)])
    (string-ci=? response "yes")))

;; Main function
(define (main)
  (let loop ()
    (printf "Enter the filename (in the same folder): ")
    (let ([filename (read-line)])
      (if (file-exists? filename)
          (let* ([integers (read-integers-from-file filename)]
                 [pairs (create-frequency-pairs integers)]
                 [sorted-pairs (manual-sort pairs)])
            (print-pairs sorted-pairs)) ; Print the pairs in the desired format
          (printf "File not found. Please try again.\n")))
      (if (ask-to-go-again)
          (loop) ; Restart the loop
          (printf "Goodbye!\n")))) ; Exit the program

;; Start the program
(main)
```

Previous Screenshot:



```
1 #lang racket
15 (define (create-frequency-pairs integers)
16   (let loop ([lst integers] [freq-map (hash)]) ; Start with an empty hash map
17     (if (null? lst)
18         (hash->list freq-map) ; Convert the hash map to a list of pairs
19         (let ([num (car lst)])
20           (loop (cdr lst) (hash-update freq-map num add1 0))))) ; Update the frequency count
21
22 ;; Manual sorting function (using insertion sort)
23 (define (manual-sort pairs)
24   (define (insert x sorted)
25     (if (null? sorted)
26         (list x) ; If the list is empty, just add x
27         (let ([current (car sorted)])
28           (if (< (car x) (car current)) ; Compare the first element of the pairs
29               (cons x sorted) ; Insert x before the current element
30               (cons current (insert x (cdr sorted)))))) ; Keep the current element and insert x into the rest
31   (fold insert '() pairs)) ; Insert each element into the sorted list
32
33 ;; Function to print the sorted value-frequency pairs in the desired format
34 (define (print-pairs pairs)
35   (printf "~a\n" pairs)) ; Print the list of pairs directly
36
37 ;; Function to ask the user if they want to go again
38 (define (ask-to-go-again)
39   (printf "Do you want to process another file? (yes/no): ")
40   (let ([response (read-line)])
41     (string-ci=? response "yes")))
42
```

Welcome to DrRacket, version 8.15 [cs].
Language: racket, with debugging; memory limit: 128 MB.
Enter the filename (in the same folder): Data-1.txt
((1 . 2) (2 . 1) (3 . 3) (5 . 1) (6 . 2) (7 . 2) (8 . 6) (9 . 2) (10 . 1))
Do you want to process another file? (yes/no): yes
Enter the filename (in the same folder): data-2.txt
((1 . 1) (2 . 1) (6 . 1) (8 . 1) (9 . 1) (15 . 1) (18 . 1) (19 . 1) (21 . 1) (22 . 1) (26 . 1) (32 . 1) (34 . 1) (37 . 1) (38 . 1) (40 . 1) (44 . 1) (45 . 1) (48 . 1) (49 . 1))
Do you want to process another file? (yes/no): yes
Enter the filename (in the same folder): Data-3.txt
((2 . 5))
Do you want to process another file? (yes/no): yes
Enter the filename (in the same folder): Data-4.txt
((1 . 1) (2 . 1) (3 . 1) (4 . 1) (5 . 1))
Do you want to process another file? (yes/no): yes
Enter the filename (in the same folder): Data-5.txt
((-25 . 1) (-24 . 1) (-22 . 1) (-17 . 1) (-14 . 1) (-8 . 1) (-7 . 1) (-1 . 1) (1 . 1) (2 . 1) (5 . 1) (6 . 1) (7 . 1) (8 . 1) (9 . 1) (10 . 1) (12 . 1) (13 . 1) (15 . 1) (20 . 1))
Do you want to process another file? (yes/no): yes
Enter the filename (in the same folder): Data-6.txt
((-96 . 1) (-80 . 1) (-81 . 1) (-80 . 1) (-77 . 1) (-76 . 1) (-71 . 1) (-67 . 1) (-63 . 1) (-62 . 1) (-57 . 1) (-55 . 1) (-53 . 1) (-51 . 1) (-35 . 2) (-33 . 1) (-32 . 1) (-27 . 1) (-25 . 1) (-24 . 1) 2
(-16 . 1) (-15 . 1) (-13 . 1) (-10 . 1) (-1 . 1) (4 . 150) (10 . 1) (11 . 1) (13 . 1) (17 . 1) (24 . 1) (27 . 1) (28 . 2) (31 . 1) (33 . 2) (51 . 1) (52 . 1) (58 . 1) (59 . 1) (67 . 1) (70 . 1) (77 . 1) 2
(83 . 1) (85 . 1) (87 . 1) (89 . 1) (98 . 1) (99 . 1))
Do you want to process another file? (yes/no): yes
Enter the filename (in the same folder): Data-7.txt

Final approach, final code and final Screenshot of my output:

Prompt -1:

This is my code.(I gave the whole previous code). It doesn't work for large data lists. Modify it so that it can work for at least 10 million+ data lists.

Code Generated:

```
#lang racket
```

(require racket/list) ; Import list utilities

; Function to read integers from a file efficiently

```
(define (read-integers-from-file filename)
```

```
  (define in-port (open-input-file filename))
```

```
  (let loop ([integers '()])
```

```
    (define num (read in-port))
```

```
    (if (eof-object? num)
```

```
        (begin
```

```
          (close-input-port in-port)
```

```
(reverse integers))  
(loop (cons num integers))))))
```

; Function to create value-frequency pairs efficiently

```
(define (create-frequency-pairs integers)  
  (for/fold ([freq-map (hash)]) ([num integers])  
    (hash-update freq-map num add1 0))) ; Efficient frequency count using hash map
```

; Merge Sort function for better sorting performance

```
(define (merge-sort lst)  
  (define (merge left right)  
    (cond  
      [(null? left) right]  
      [(null? right) left]  
      [(<= (car left) (car right)) (cons (car left) (merge (cdr left) right))]  
      [else (cons (car right) (merge left (cdr right)))]))  
  (if (or (null? lst) (null? (cdr lst)))  
      lst  
      (let* ([mid (quotient (length lst) 2)]  
             [left (take lst mid)]  
             [right (drop lst mid)])  
        (merge (merge-sort left) (merge-sort right)))))
```

; Function to sort the frequency pairs by value

```
(define (sort-pairs pairs)  
  (merge-sort pairs)) ; Use merge-sort for better performance on large lists
```

; Function to print the sorted value-frequency pairs

```
(define (print-pairs pairs)  
  (for-each (λ (pair) (printf "~a: ~a\n" (car pair) (cdr pair))) pairs))
```

; Function to ask the user if they want to go again

```
(define (ask-to-go-again)  
  (printf "Do you want to process another file? (yes/no): ")  
  (let ([response (read-line)])  
    (string-ci=? response "yes")))
```

; Main function

```
(define (main)  
  (let loop ()
```

```
(printf "Enter the filename (in the same folder): ")
(let ([filename (read-line)])
  (if (file-exists? filename)
      (let* ([integers (read-integers-from-file filename)]
             [freq-map (create-frequency-pairs integers)]
             [pairs (hash->list freq-map)]
             [sorted-pairs (sort-pairs pairs)])
        (print-pairs sorted-pairs))
      (printf "File not found. Please try again.\n"))))
(if (ask-to-go-again)
    (loop)
    (printf "Goodbye!\n"))))

; Start the program
(main)
```

Prompt -2:

It has this error- module: identifier already defined in: read-integers-from-file . Please try avoiding this error

- It gave the same code with same errors for three times.

Prompt-3:

sort this error , give correct code- car: contract violation expected: pair? given: -999999,also Given the list of value-count pairs, produce a sorted list; that is, from the above you would produce (-1 1 2 2 2 3 3 5) also it should work like this. give my final code which work for both steps. (give value-frequency pairs and sort the list)

#lang racket

(require racket/list) ; Import list utilities

```
; Function to read integers from a file efficiently
(define (read-integers filename)
  (define in-port (open-input-file filename))
  (let loop ([integers '()])
    (define num (read in-port))
    (if (eof-object? num)
        (begin
          (close-input-port in-port)
          (reverse integers))
        (loop (cons num integers)))))
```

```
(loop (cons num integers))))))
```

; Function to create value-frequency pairs efficiently

```
(define (create-frequency-pairs integers)
  (for/fold ([freq-map (hash)]) ([num integers])
    (hash-update freq-map num add1 0))) ; Efficient frequency count using hash map
```

; Merge Sort function for better sorting performance

```
(define (merge-sort lst)
  (define (merge left right)
    (cond
      [(null? left) right]
      [(null? right) left]
      [(<= (car left) (car right)) (cons (car left) (merge (cdr left) right))]
      [else (cons (car right) (merge left (cdr right)))]))
  (if (or (null? lst) (null? (cdr lst)))
      lst
      (let* ([mid (quotient (length lst) 2)]
             [left (take lst mid)]
             [right (drop lst mid)])
        (merge (merge-sort left) (merge-sort right)))))
```

; Function to sort the frequency pairs by value

```
(define (sort-pairs pairs)
  (merge-sort (map car pairs))) ; Use merge-sort to sort by the number
```

; Function to print the sorted value-frequency pairs in the desired format on the same line

```
(define (print-pairs pairs)
  (for-each (λ (pair)
    (printf "(~a ~a) " (car pair) (cdr pair))) ; Print in the format (number No.oftimes
repeated)
    pairs)
  (newline)) ; Move to the next line after printing all pairs
```

; Function to generate a sorted list of numbers based on frequency

```
(define (generate-sorted-list pairs)
  (apply append
    (map (λ (pair)
      (make-list (cdr pair) (car pair))) ; Repeat the number by its frequency
      pairs)))
```

```
; Function to print the sorted list of numbers
(define (print-sorted-list sorted-list)
  (for-each (λ (num) (printf "~a " num)) sorted-list)
  (newline)) ; Move to the next line after printing all numbers

; Function to ask the user if they want to go again
(define (ask-to-go-again)
  (printf "Do you want to process another file? (yes/no): ")
  (let ([response (read-line)])
    (string-ci=? response "yes"))))

; Main function
(define (main)
  (let loop ()
    (printf "Enter the filename (in the same folder): ")
    (let ([filename (read-line)])
      (if (file-exists? filename)
          (let* ([integers (read-integers filename)]
                 [freq-map (create-frequency-pairs integers)]
                 [pairs (hash->list freq-map)]
                 [sorted-keys (sort-pairs pairs)]
                 [sorted-pairs (map (λ (key) (cons key (hash-ref freq-map key))) sorted-keys)]
                 [sorted-list (generate-sorted-list sorted-pairs)])
            (printf "Value-Count Pairs: ")
            (print-pairs sorted-pairs)
            (printf "Sorted List: ")
            (print-sorted-list sorted-list))
          (printf "File not found. Please try again.\n"))))
    (if (ask-to-go-again)
        (loop)
        (printf "Goodbye!\n"))))

; Start the program
(main)
- It finally gave the correct code after this prompt.
However, It was really hard for me to make it run for 10M + list items. Finally, after a huge
practice and patience, I got the correct code which gives both value-frequency pairs and a sorted
list.
```

Final Outputs:

[illegible]


```

1 #lang racket
2
3 (require racket/list) ; Import list utilities
4
5 ; Function to read integers from a file efficiently
6 (define (read-integers filename)
7   (define (open-input-file filename)
8     (let loop ([integers '()])
9       (define num (read-inport))
10      (if eof-object? num
11          begin
12            (close-input-port in-port)
13            (reverse integers)
14            (loop (cons num integers))))
15      ))
16
17 ; Function to create value-frequency pairs efficiently
18 (define (create-frequency-pairs integers)
19   (for/fold ([freq-map (hash)] [num integers])
20     (hash-update (freq-map num add0))) ; Efficient frequency count using hash map
21 )
22
23 ; Merge Sort function for better sorting performance
24 (define (merge-sort lst)
25   (define (merge left right)
26     (cond
27       [(null? left) right]
28       [(null? right) left]
29       [(<= (car left) (car right)) (cons (car left) (merge (cdr left) right))]
30       [else (cons (car right) (merge left (cdr right)))]))
31 )
32
33 (-73499 1) (-73497 1) (-73492 1) (-73490 1) (-73485 2) (-73483 2) (-73476 3) (-73474 2) (-73471 2) (-73469 2) (-73467 1) (-73464 1) (-73463 1) (-73462 1) (-73457 3) (-73456 2) (-73453 2)
34 (-73452 1) (-73450 3) (-73448 1) (-73446 1) (-73445 2) (-73440 1) (-73435 1) (-73434 1) (-73431 3) (-73430 1) (-73426 1) (-73424 1) (-73422 2)
35 (-73418 3) (-73417 1) (-73414 2) (-73409 2) (-73404 1) (-73401 1) (-73400 1) (-73398 1) (-73397 1) (-73396 1) (-73394 1) (-73390 1) (-73387 1) (-73386 1) (-73383 1) (-73382 2) (-73379 2)
36 (-73378 2) (-73377 2) (-73374 3) (-73373 1) (-73369 1) (-73367 1) (-73359 1) (-73356 1) (-73355 3) (-73354 1) (-73352 1) (-73343 1) (-73342 1) (-73340 1) (-73339 1) (-73338 2)
37 (-73334 2) (-73326 1) (-73324 1) (-73323 1) (-73321 1) (-73317 2) (-73315 2) (-73314 1) (-73310 1) (-73309 1) (-73305 2) (-73304 1) (-73298 1) (-73294 1) (-73291 2) (-73290 1)
38 (-73287 1) (-73286 1) (-73280 1) (-73277 1) (-73275 1) (-73274 2) (-73266 1) (-73264 1) (-73262 1) (-73259 1) (-73258 2) (-73257 1) (-73253 1) (-73246 2) (-73243 1) (-73237 1)
39 (-73231 1) (-73230 1) (-73228 1) (-73227 1) (-73225 1) (-73220 1) (-73215 1) (-73214 1) (-73207 1) (-73206 1) (-73205 2) (-73200 1) (-73198 1) (-73192 2) (-73190 1) (-73188 1)
40 (-73186 3) (-73181 2) (-73178 1) (-73177 2) (-73176 1) (-73173 1) (-73167 2) (-73162 1) (-73160 1) (-73158 3) (-73156 1) (-73151 1) (-73149 1) (-73145 2) (-73144 1) (-73142 2)
41 (-73141 1) (-73140 1) (-73139 1) (-73134 1) (-73132 1) (-73131 1) (-73129 1) (-73121 1) (-73118 2) (-73117 1) (-73116 2) (-73115 1) (-73111 1) (-73109 2) (-73105 1) (-73104 1)
42 (-73100 2) (-73095 2) (-73091 2) (-73084 1) (-73082 1) (-73080 1) (-73079 1) (-73078 3) (-73072 1) (-73065 1) (-73064 1) (-73063 1) (-73061 1) (-73060 1) (-73059 1) (-73057 1)
43 (-73055 2) (-73049 1) (-73046 1) (-73046 1) (-73043 1) (-73042 1) (-73040 1) (-73035 1) (-73032 1) (-73029 1) (-73028 2) (-73027 1) (-73026 1) (-73021 1) (-73016 1) (-73015 1)
44 (-73010 1) (-73007 2) (-73002 1) (-73001 1) (-73000 1) (-72998 1) (-72997 1) (-72994 1) (-72993 1) (-72991 1) (-72989 1) (-72988 1) (-72987 1) (-72986 1) (-72985 1) (-72984 2)
45 (-72975 1) (-72972 1) (-72971 1) (-72970 1) (-72967 2) (-72966 2) (-72965 2) (-72962 1) (-72955 1) (-72951 2) (-72950 1) (-72946 1) (-72945 1) (-72939 1) (-72937 1) (-72936 1)
46 (-72934 3) (-72929 2) (-72928 2) (-72927 1) (-72925 1) (-72924 1) (-72917 1) (-72915 1) (-72914 1) (-72912 1) (-72909 1) (-72908 1) (-72904 2) (-72903 1) (-72900 1) (-72899 1)
47 (-72895 2) (-72891 2) (-72888 1) (-72881 1) (-72878 1) (-72873 1) (-72871 1) (-72868 1) (-72863 1) (-72862 1) (-72860 1) (-72855 1) (-72854 1) (-72853 1) (-72848 1) (-72844 1)
48 (-72842 1) (-72838 1) (-72837 1) (-72833 1) (-72832 1) (-72831 1) (-72826 1) (-72823 1) (-72822 1) (-72820 1) (-72816 1) (-72807 1) (-72804 1) (-72800 2) (-72798 1) (-72799 1)
49 (-72798 1) (-72795 2) (-72794 1) (-72793 1) (-72787 1) (-72785 1) (-72782 1) (-72781 3) (-72775 1) (-72772 1) (-72770 1) (-72768 2) (-72767 1) (-72761 1) (-72758 2) (-72756 1)
50 (-72754 1) (-72753 2) (-72751 1) (-72741 1) (-72734 2) (-72732 1) (-72730 2) (-72728 1) (-72728 1) (-72724 1) (-72721 2) (-72715 1) (-72714 1) (-72713 1) (-72712 1) (-72706 2)
51 (-72700 1) (-72693 1) (-72691 1) (-72689 1) (-72686 1) (-72683 1) (-72681 1) (-72678 1) (-72676 2) (-72672 1) (-72670 1) (-72666 3) (-72665 1) (-72664 1) (-72662 1) (-72658 2)
52 (-72657 2) (-72654 1) (-72649 1) (-72645 1) (-72642 1) (-72641 1) (-72636 1) (-72635 1) (-72630 1) (-72627 1) (-72622 1) (-72617 4) (-72615 1) (-72614 1) (-72613 1) (-72612 1)
53 (-72609 1) (-72604 1) (-72601 2) (-72600 1) (-72599 1) (-72595 1) (-72591 1) (-72590 1) (-72586 2) (-72584 1) (-72583 1) (-72580 2) (-72578 1) (-72577 2) (-72572 1) (-72569 1)
54 (-72562 2) (-72561 1) (-72554 1) (-72552 2) (-72550 1) (-72546 1) (-72543 1) (-72538 2) (-72532 1) (-72531 1) (-72528 2) (-72525 1) (-72524 1) (-72522 1) (-72521 1) (-72517 1)
55 (-72515 1) (-72509 1) (-72506 1) (-72505 1) (-72503 1) (-72498 1) (-72497 1) (-72496 2) (-72494 1) (-72487 1) (-72486 1) (-72483 1) (-72480 1) (-72476 2) (-72473 1) (-72470 1)
56 (-72465 1) (-72464 1) (-72462 1) (-72454 2) (-72451 2) (-72447 1) (-72442 1) (-72438 1) (-72438 1) (-72437 1) (-72433 1) (-72432 1) (-72431 1) (-72429 1) (-72427 2) (-72426 1)
57 (-72422 2) (-72418 1) (-72416 1) (-72415 1) (-72417 2) (-72411 3) (-72410 1) (-72408 1) (-72406 1) (-72405 1) (-72400 1) (-72398 1) (-72396 1) (-72393 1) (-72390 1) (-72388 1)
58 (-72387 1) (-72384 1) (-72383 3) (-72382 1) (-72379 1) (-72378 1) (-72377 3) (-72374 1) (-72373 1) (-72371 2) (-72370 1) (-72362 2) (-
```

Conclusion:

I submitted multiple versions of the code because I was still learning and trying to understand the concepts involved. Initially, I was confused about how to implement certain features efficiently, so I experimented with different approaches. Through this iterative process of practice and trial and error, I gradually improved my skills and gained a better understanding of the problem. Each version reflects my evolving knowledge and commitment to finding the best solution. After extensive practice and gaining clarity, I was finally able to develop the optimized and functional version of the code.