Report CS3113 Lab-01

Assumptions

We are assuming that the x32 system we have been working on until this point will no longer be relevant to our current needs and thus have switched over to an x86_64 system. The hardest part, both technically and overall, of the project will most likely implement the sorting algorithms. In our case, we decided on both bubble sort and insertion sort on our list because the bulk of the work in both the stack and registers is done in these algorithms. Most likely, the easiest part of the lab project will be the presentation, since by that point, we will have worked through many other aspects of the project, and now we must only explain our work process and the final product. However, the creation of README.md may also be a relatively simple process as it is mostly a document used to keep track of our schedule, meetings, and reflections. Our interpretation of the specification and subsequent enhancements all relied on the principle of sorting the address rather than the actual list values.

Objectives

The objective of this lab is that we implement bubble sort on a linked list in x86_64 assembly. At a high level, we have fixed data values, and after putting them into an array, we create a second array to hold the addresses of the data items. We can then compare the values from our source data by traversing through the address array. We swap them if the first value is bigger than the second and leave them alone if not. After doing so, it moves onto the next value and address and then loops the process until the end of the linked list is reached, denoted by a -1. Using a register as a boolean flag, we check to see if any swaps are made when passing through the list, and if none are made, then the list is sorted and the function is exited.

Additionally, we did various enhancements to the project, including function compartmentalization, alternative output, implementation of insertion sort. At a high level, all these tasks are simple. In a language like C/C++, function creation and parameter passing are built-in, and while assembly can support functions, more work has to be put into parameter passing. In x86_64, the registers acted as parameters to pass information in. In order to print to the terminal, the values must first be manually converted from integers to ASCII. Then they must be added to a buffer which can then be displayed using a system call.

Insertion sort can easily be done using duel for loops. Like in bubble sort, the algorithm uses various input values and compares neighboring values to see which one is bigger. However, insertion sort has an additional caveat that instead of just swapping with the original neighbor, the smaller value will continue swapping until it finds a value that is smaller than itself. This repeats until the list is sorted.

Another enhancement we added was scripting GDB commands to show the memory locations without actually having input and outputs. We did this by traversing through the GDB, turning pagination off and placing breakpoints at certain functions, and printing out registers at that certain time. We can also tell the system to print all of the results out to a file by calling "set logging file

<filename>." After ending the program, you set the pagination off, and it will save all your outputs to a separate file under the specified name.

Fortunately, a lot of our code was debugged through the GDB, which we are acquainted with due to the sheer amount of tracing we do on most of our programming assignments. Specifically, learning break _start or break _FUNCTIONNAME was super helpful as we didn't need to step through the entire program and could start wherever we wanted. Much of our program relied heavily on both registers and the stack. All available general registers were used: %rax, %rbx, %rcx, %rdx, %rdi, %rsi. The stack was used heavily within many of the major functions

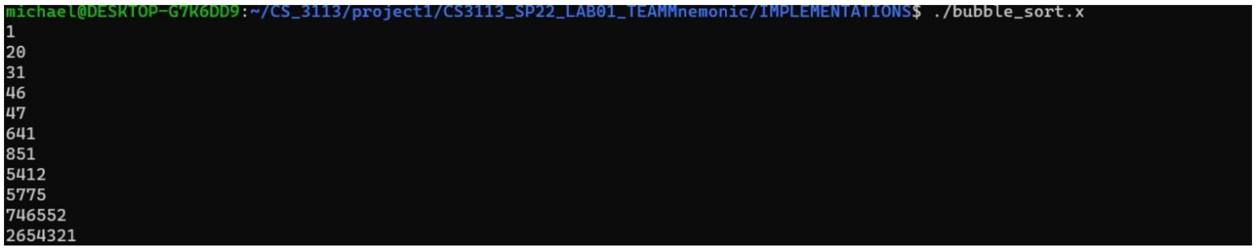
Inputs and Outputs

Inputs

Our program has the input baked into the bubble_sort.s program. As stated previously, input is held within a quad called data_items. Data_items are hardcoded into the .data section of the program and can be adjusted to hold up to 100 quad addresses with no further modification of the code. An even greater number of items could easily be inputted amount of reserved memory for the addresses array was increased. The last element of data_items is -1, which is a flag signaling the end of the list. This flag would not be necessary for a higher-level language as other more user-friendly methods are available. We did attempt the enhancement where the input came from various files containing fixed sized list. The failed file reading for input will be talked about more extensively in the refinement section.

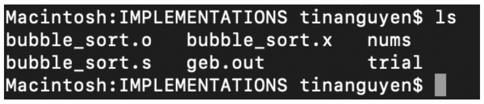
Outputs

In the beginning, it would only print single-digit numbers. However, this issue was eventually fixed. The algorithm that prints out the function are housed within the bubble_sort.s. Below is a picture of a list sorted by our program.



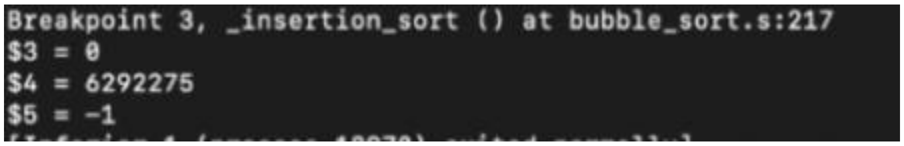
```
michael@DESKTOP-G7K6DD9:~/CS_3113/project1/CS3113_SP22_LAB01_TEAMMnemonic/IMPLEMENTATIONS$ ./bubble_sort.x
1
20
31
46
47
641
851
5412
5775
746552
2654321
```

In our GDB scripting, we were also able to print out values that are held in registers into a separate output file. Below is the file (geb.out) shown when calling ls and a snippet of what's in the directory.



```
Macintosh:IMPLEMENTATIONS tinanguyen$ ls
bubble_sort.o   bubble_sort.x   nums
bubble_sort.s   geb.out         trial
Macintosh:IMPLEMENTATIONS tinanguyen$
```

The screenshot below is from geb.out. Breakpoint 3 was printing out three registers: the index register, the register used to hold memory, and the end of the list, which was –1.



```
Breakpoint 3, _insertion_sort () at bubble_sort.s:217
$3 = 0
$4 = 6292275
$5 = -1
```

Refinement

There was some significant refinement during the lab, and several different ideas had to be radically changed or dropped altogether. Originally, our creation of the linked list relied on a back-to-back method where the value is immediately followed by the address of the following link. However, hardcoding records took extensive time and was hard to adjust, and creating dynamically sized records proved to be too challenging. To combat the issues caused by record-based linked lists, we decided to use a different kind of storing method, parallel lists. One list, in our case, we used a quad to store information, stored the data values. Memory was then reserved for a second array that would hold the addresses of the data values. This list did not have initial values and, as such, could be stored in the .bss section of the code. It would later be filled in using the load effective address command to get the virtual memory address of each data element. Since the address list contained pointers to the next item, the first item in the list had nothing pointing directly to it. To solve this, a separate head location was created that would hold an address pointing specifically to the front of the data list.

As mentioned earlier, we did attempt to read to create a section of the program which would read in various files that contained a fixed-length list. However, despite trying to replicate what the Programming from the Ground Up succeeded in doing in chapter five, we were never able to manage to get it to work due to the lack of registers available for use; after spending quite a bit of time on various attempts, we eventually gave up and focused on trying other speciation instead. There were two main types of versions: one that ran inside the original bubble_sort.s file and linked two files, bubble_sort.s and trial together to run. We tried connecting them through global variables but got stuck on how to link them both together in the end.

Another enhancement that was completely dropped was the doubly linked list. Unfortunately, due to how we created the sorting algorithms and other various functions, problems arose when trying to sort in a backward manner. Issues came in a variety of forms, from trying to find the stopping point of the news list, modifying our sorting algorithms to accept the new stopping point, the head address being overwritten, etc.; filling the list backward list was a nightmare. Sorting the backward address list ended up breaking our previously functioning bubble and insertion sort functions. The doubly linked list was eventually dropped to pursue other enhancements, such as the implementation of the alternate output, as fixing was taking more time than we had.

The sorting algorithms and function compartmentalization went through major refinement as well. We started working on bubble sort and managed to originally get in working without ever using the stack, but it took almost all the registers–baring %rdi and %rsi. Lack of registers became a major obstacle in the creation of functions as register would contain important data when passed but be lost within the function when used; however, the lack of open registers became a much bigger problem with the creation of insertion sort. Insertion sort was an enhancement itself that was made almost simultaneously as function compartmentalization. It became clear that just using registers was not a viable solution, which led to the use of the stack. The stack enabled us to temporarily store the value of registers, which in turn freed them up to be used at other parts of the algorithm. The values could then be safely returned to the registers by popping from the stack when needed.

One of the last enhancements we added was scripting GBD commands. Throughout the lab, before inputs were hardcoded in, there was no way to check if our code was working properly, but with the use of the GBD commands, we were able to look at memory locations being passed around, and when they were compared, we were able to see if our code was properly working. This became very useful because the outputs would save out into a separate file, and we could reference back to it at any time.

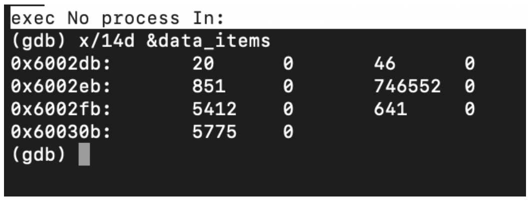
Pseudo-Code

Due to the sheer amount of code we went through to create our final functioning output, the pseudo-code section will only contain the code on the completed section of our project.

```
data_items = {20,46,851,746552,5412,641,5775,47,31,2654321,1,-1};
var buff = {items used for printing};
array next_address = {holds up to 100 addresses};
var head = {holds the head of the next_address array};
-start
rcx = address of data_items [0];
head = rcx;
rdx = address of next_address [0] <- empty at the start;
_fill_in_addresses (); <- fills in next addresses so that it holds the neighbor value's address (acts as a pointer)
rcx = address of next_ address [0]; <- no longer empty
rdx = head;
_insertion_sort (); <- bubble sort could also be called/ sorts the elements of next_address
_print_loop (); (Cycles through the address array and calls print_value on the value being pointed at <- using the buffer section to convert from and address to a number to a ASCII value) <- prints out the elements of next_address
_exit_x86_64bit (); <- exits the program
```

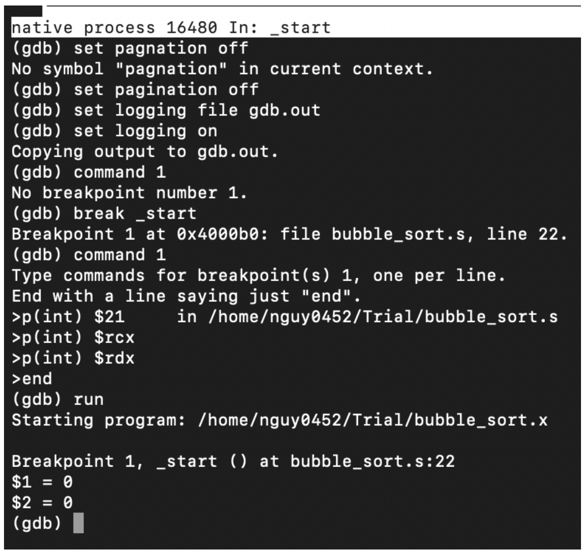
GDB Scripting

Using GDB scripting commands were particularly useful for checking if our code was working without checking inputs and outputs and only looking at memory locations and seeing if they were being transferred properly.



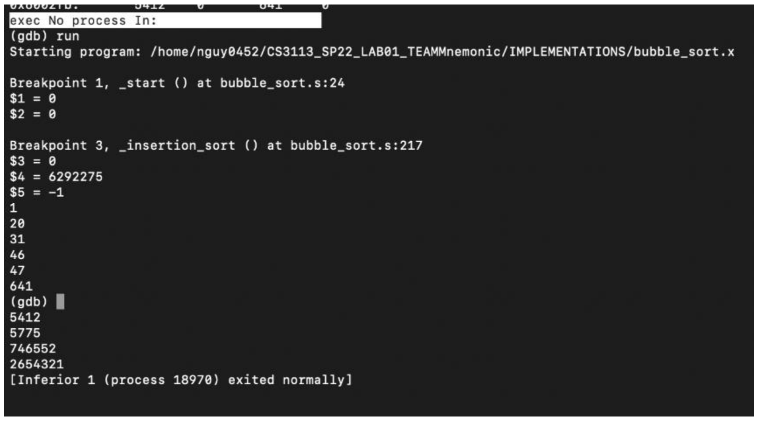
```
exec No process In:
(gdb) x/14d &data_items
0x6002db:       20      0       46      0
0x6002eb:       851     0       746552  0
0x6002fb:       5412    0       641     0
0x60030b:       5775    0
(gdb)
```

This screenshot shows the values of data_items being held. Typing this while the gdb is running will print it out into the output file at the very end.



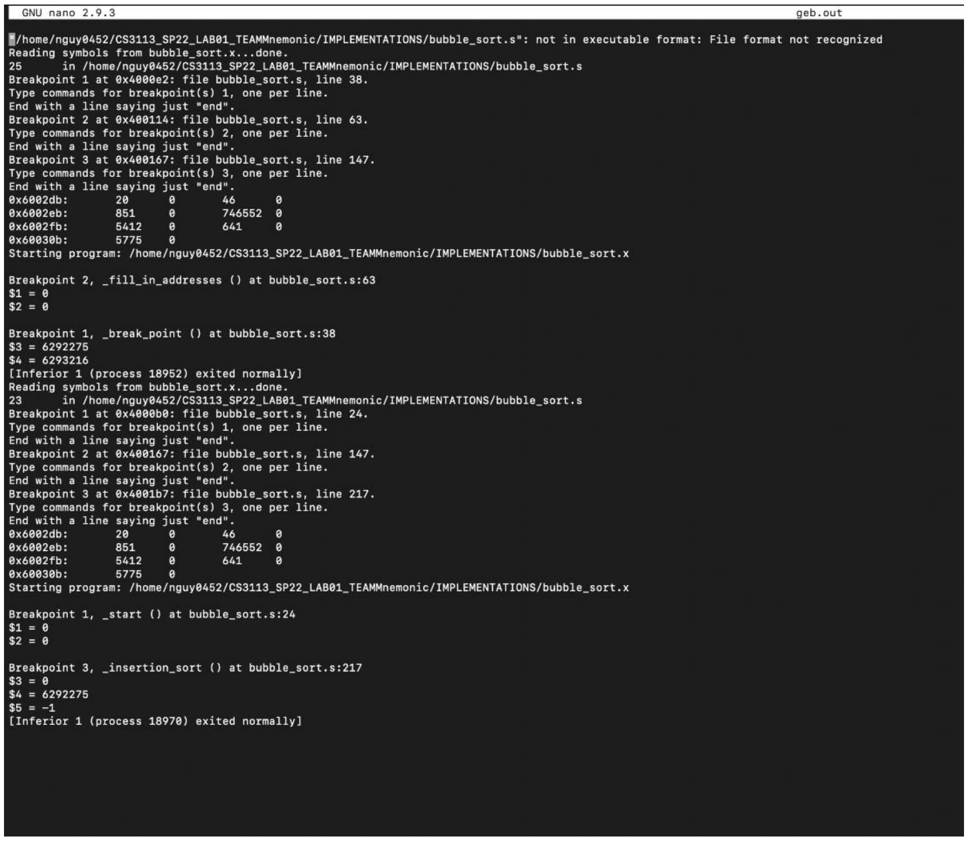
```
native process 16480 In: _start
(gdb) set pagnation off
No symbol "pagnation" in current context.
(gdb) set pagination off
(gdb) set logging file gdb.out
(gdb) set logging on
Copying output to gdb.out.
(gdb) command 1
No breakpoint number 1.
(gdb) break _start
Breakpoint 1 at 0x4000b0: file bubble_sort.s, line 22.
(gdb) command 1
Type commands for breakpoint(s) 1, one per line.
End with a line saying just "end".
>p(int) $21     in /home/nguy0452/Trial/bubble_sort.s
>p(int) $rcx
>p(int) $rdx
>end
(gdb) run
Starting program: /home/nguy0452/Trial/bubble_sort.x

Breakpoint 1, _start () at bubble_sort.s:22
$1 = 0
$2 = 0
(gdb)
```

These are the commands used to set up the gdb scripting. The commands p(int) $whatever will print out whatever is held in the registers cast as an int, and one can specify to start at any specific function.



```
exec No process In:
(gdb) run
Starting program: /home/nguy0452/CS3113_SP22_LAB01_TEAMMnemonic/IMPLEMENTATIONS/bubble_sort.x

Breakpoint 1, _start () at bubble_sort.s:24
$1 = 0
$2 = 0

Breakpoint 3, _insertion_sort () at bubble_sort.s:217
$3 = 0
$4 = 6292275
$5 = -1
1
20
31
46
47
641
(gdb) 
5412
5775
746552
2654321
[Inferior 1 (process 18970) exited normally]
```

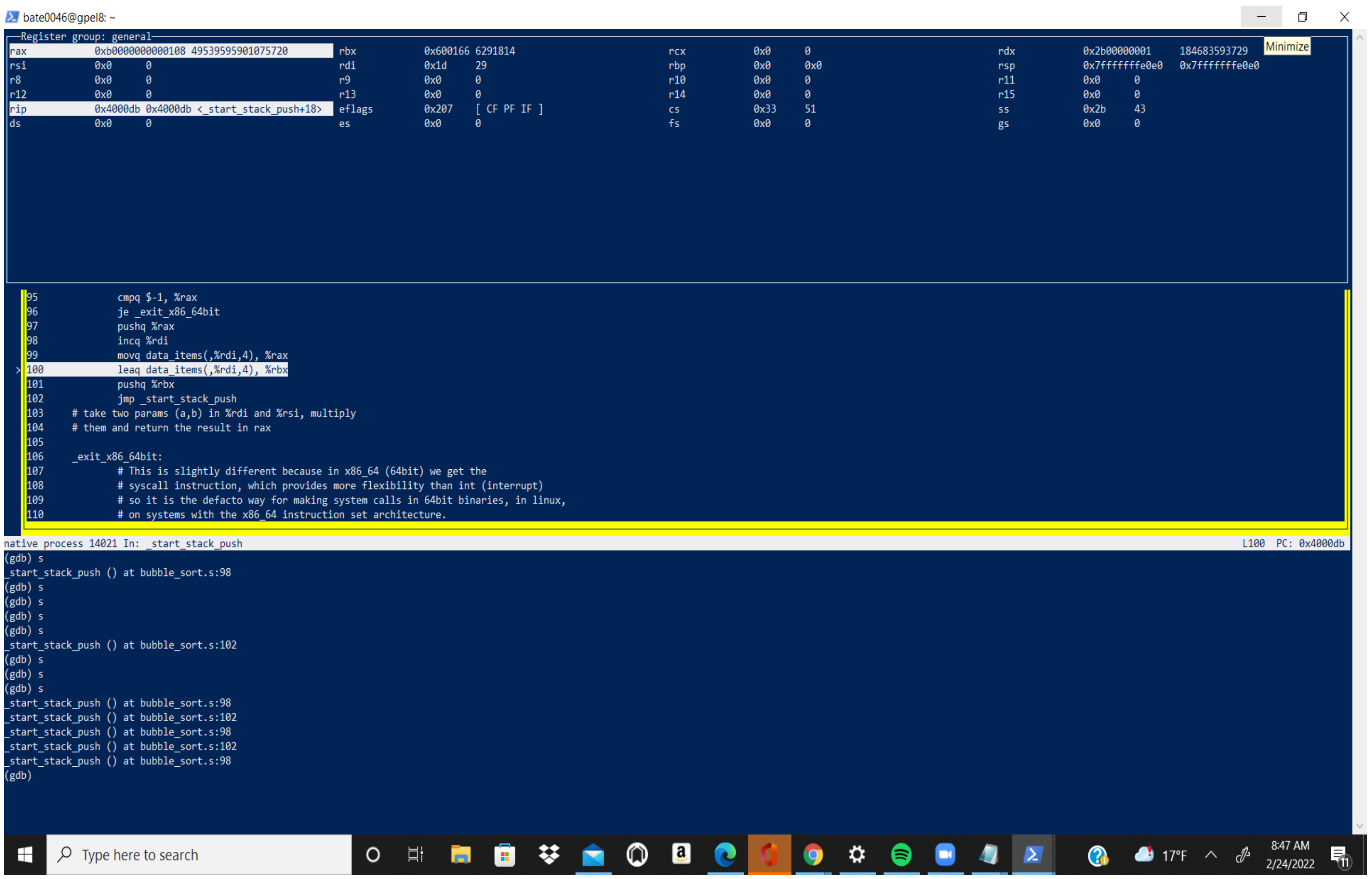
This screenshot depicts more examples of what is contained in the output file. The values printed such as 1, 20, 31, 46 are values held in a register when line 217 was running.



```
  GNU nano 2.9.3                                                                                geb.out

"/home/nguy0452/CS3113_SP22_LAB01_TEAMMnemonic/IMPLEMENTATIONS/bubble_sort.s": not in executable format: File format not recognized
Reading symbols from bubble_sort.x...done.
25      in /home/nguy0452/CS3113_SP22_LAB01_TEAMMnemonic/IMPLEMENTATIONS/bubble_sort.s
Breakpoint 1 at 0x4000e2: file bubble_sort.s, line 38.
Type commands for breakpoint(s) 1, one per line.
End with a line saying just "end".
Breakpoint 2 at 0x400114: file bubble_sort.s, line 63.
Type commands for breakpoint(s) 2, one per line.
End with a line saying just "end".
Breakpoint 3 at 0x400167: file bubble_sort.s, line 147.
Type commands for breakpoint(s) 3, one per line.
End with a line saying just "end".
0x6002db:       20      0       46      0
0x6002eb:       851     0       746552  0
0x6002fb:       5412    0       641     0
0x60030b:       5775    0
Starting program: /home/nguy0452/CS3113_SP22_LAB01_TEAMMnemonic/IMPLEMENTATIONS/bubble_sort.x

Breakpoint 2, _fill_in_addresses () at bubble_sort.s:63
$1 = 0
$2 = 0

Breakpoint 1, _break_point () at bubble_sort.s:38
$3 = 6292275
$4 = 6293216
[Inferior 1 (process 18952) exited normally]
Reading symbols from bubble_sort.x...done.
23      in /home/nguy0452/CS3113_SP22_LAB01_TEAMMnemonic/IMPLEMENTATIONS/bubble_sort.s
Breakpoint 1 at 0x4000b0: file bubble_sort.s, line 24.
Type commands for breakpoint(s) 1, one per line.
End with a line saying just "end".
Breakpoint 2 at 0x400167: file bubble_sort.s, line 147.
Type commands for breakpoint(s) 2, one per line.
End with a line saying just "end".
Breakpoint 3 at 0x4001b7: file bubble_sort.s, line 217.
Type commands for breakpoint(s) 3, one per line.
End with a line saying just "end".
0x6002db:       20      0       46      0
0x6002eb:       851     0       746552  0
0x6002fb:       5412    0       641     0
0x60030b:       5775    0
Starting program: /home/nguy0452/CS3113_SP22_LAB01_TEAMMnemonic/IMPLEMENTATIONS/bubble_sort.x

Breakpoint 1, _start () at bubble_sort.s:24
$1 = 0
$2 = 0

Breakpoint 3, _insertion_sort () at bubble_sort.s:217
$3 = 0
$4 = 6292275
$5 = -1
[Inferior 1 (process 18970) exited normally]
```

This is the geb.out file that was created after typing all the script commands. It gives either the value at the given register or the memory addresses that are stored inside.

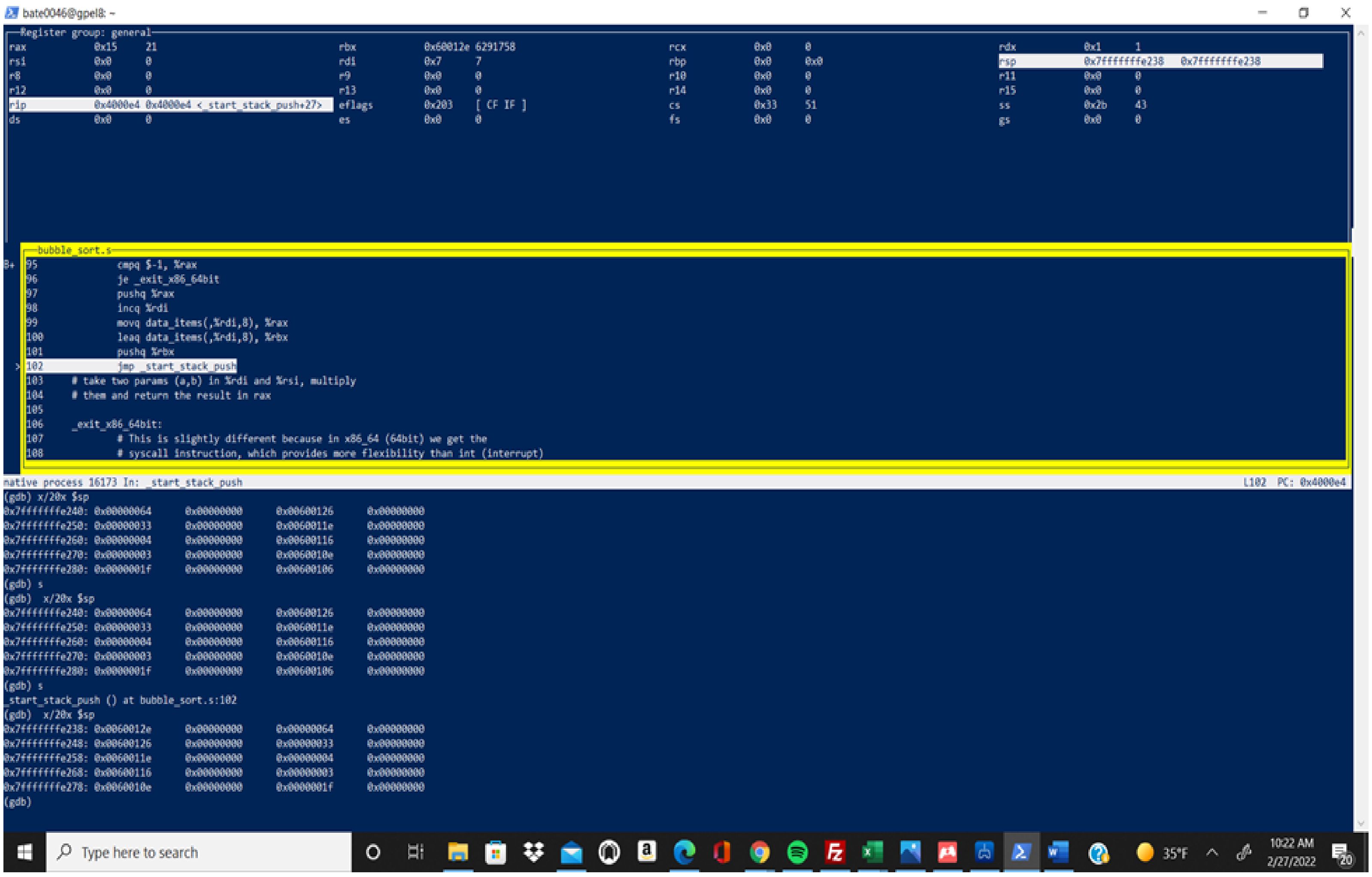
GDB Tracing



```
┌─Register group: general──────────────────────────────────────────────────────────────────────────────────────────────────
│rax            0xb000000000000108 495395959010 75720   rbx            0x600166 6291814      rcx            0x0      0                     rdx            0x2b00000001     184683593729
│rsi            0x0      0                             rdi            0x1d     29           rbp            0x0      0x0                   rsp            0x7fffffffe0e0   0x7fffffffe0e0
│r8             0x0      0                             r9             0x0      0            r10            0x0      0                     r11            0x0      0
│r12            0x0      0                             r13            0x0      0            r14            0x0      0                     r15            0x0      0
│rip            0x4000db 0x4000db <_start_stack_push+18> eflags       0x207    [ CF PF IF ] cs             0x33     51                    ss             0x2b     43
│ds             0x0      0                             es             0x0      0            fs             0x0      0                     gs             0x0      0
```

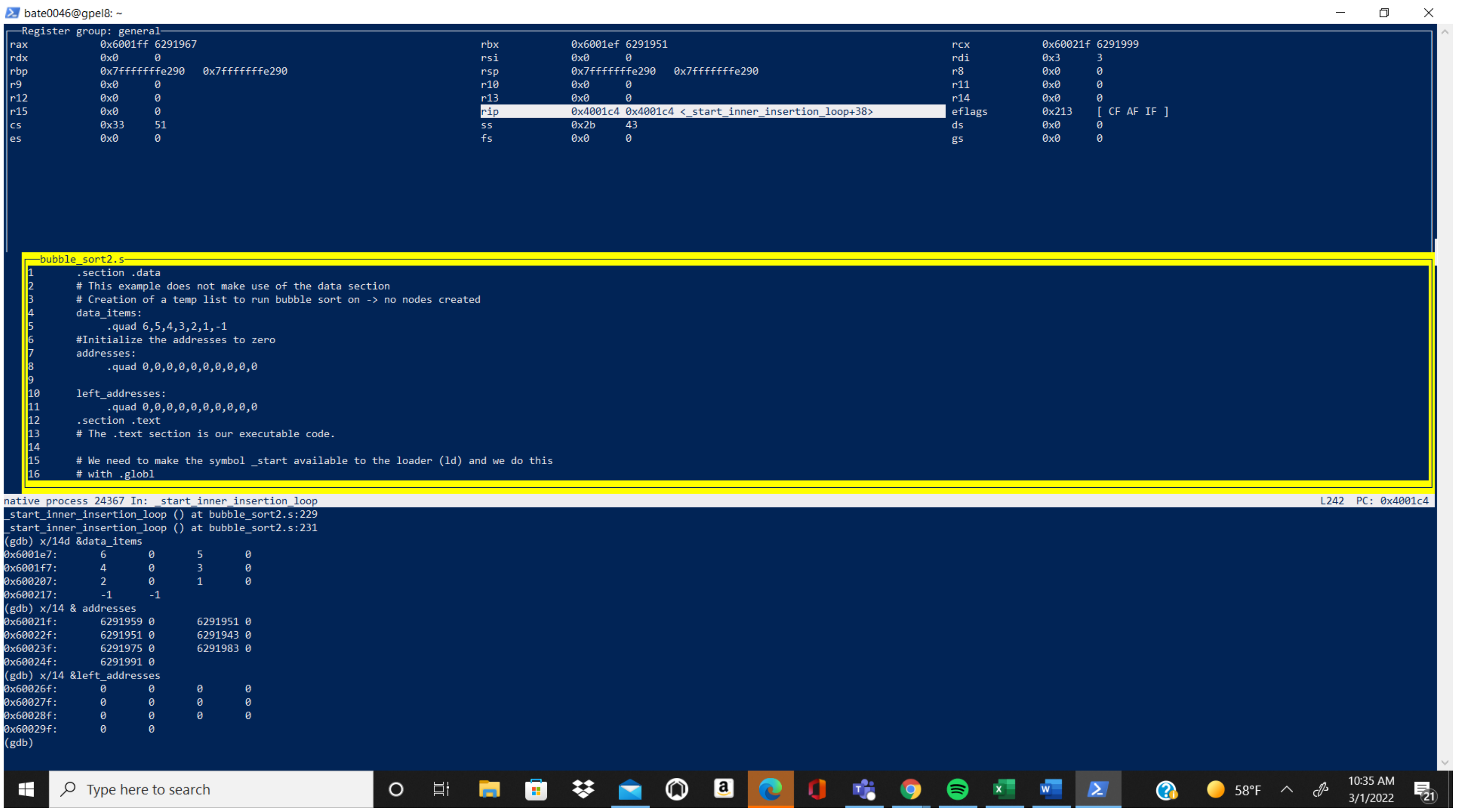
```
   95              cmpq $-1, %rax
   96              je _exit_x86_64bit
   97              pushq %rax
   98              incq %rdi
   99              movq data_items(,%rdi,4), %rax
 > 100             leaq data_items(,%rdi,4), %rbx
   101             pushq %rbx
   102             jmp _start_stack_push
   103     # take two params (a,b) in %rdi and %rsi, multiply
   104     # them and return the result in rax
   105
   106     _exit_x86_64bit:
   107             # This is slightly different because in x86_64 (64bit) we get the
   108             # syscall instruction, which provides more flexibility than int (interrupt)
   109             # so it is the defacto way for making system calls in 64bit binaries, in linux,
   110             # on systems with the x86_64 instruction set architecture.
native process 14021 In: _start_stack_push                                                                   L100  PC: 0x4000db
(gdb) s
_start_stack_push () at bubble_sort.s:98
(gdb) s
(gdb) s
(gdb) s
(gdb) s
_start_stack_push () at bubble_sort.s:102
(gdb) s
(gdb) s
(gdb) s
_start_stack_push () at bubble_sort.s:98
_start_stack_push () at bubble_sort.s:102
_start_stack_push () at bubble_sort.s:98
_start_stack_push () at bubble_sort.s:102
_start_stack_push () at bubble_sort.s:98
(gdb)
```

This gdb tracing shows one of the first attempts at a side-by-side addressing method. However, there are some significant issues within the code. Number one, a quad was not used instead of a long, which did work with how the rest of the code was set up to manipulate information from a quad. Number two piggybacks of the issue seen in number one; the register is stepping midway through a long address, when a quad should have been used, instead of accommodating for the larger size of addresses used by x86_64. These issues caused the register to hold values widely different from the actual list values and thus never end due to the ending flag never appearing.



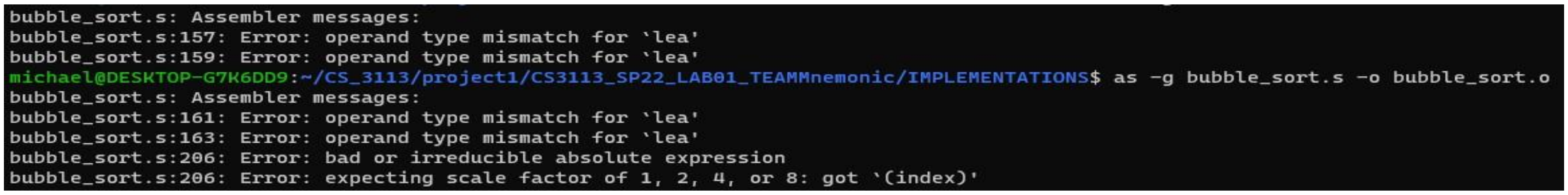
```
bate0046@gpel8: ~
┌─Register group: general─
│rax            0x15     21                      rbx            0x60012e 6291758                rcx            0x0      0                      rdx            0x1      1
│rsi            0x0      0                       rdi            0x7      7                      rbp            0x0      0x0                    rsp            0x7fffffffe238    0x7fffffffe238
│r8             0x0      0                       r9             0x0      0                      r10            0x0      0                      r11            0x0      0
│r12            0x0      0                       r13            0x0      0                      r14            0x0      0                      r15            0x0      0
│rip            0x4000e4 0x4000e4 <_start_stack_push+27>  eflags  0x203    [ CF IF ]            cs             0x33     51                     ss             0x2b     43
│ds             0x0      0                       es             0x0      0                      fs             0x0      0                      gs             0x0      0

  ┌─bubble_sort.s─
B+│95               cmpq $-1, %rax
  │96               je _exit_x86_64bit
  │97               pushq %rax
  │98               incq %rdi
  │99               movq data_items(,%rdi,8), %rax
  │100              leaq data_items(,%rdi,8), %rbx
  │101              pushq %rbx
 >│102              jmp _start_stack_push
  │103      # take two params (a,b) in %rdi and %rsi, multiply
  │104      # them and return the result in rax
  │105
  │106      _exit_x86_64bit:
  │107              # This is slightly different because in x86_64 (64bit) we get the
  │108              # syscall instruction, which provides more flexibility than int (interrupt)
native process 16173 In: _start_stack_push                                                                       L102  PC: 0x4000e4
(gdb) x/20x $sp
0x7fffffffe240: 0x00000064      0x00000000      0x00600126      0x00000000
0x7fffffffe250: 0x00000033      0x00000000      0x0060011e      0x00000000
0x7fffffffe260: 0x00000004      0x00000000      0x00600116      0x00000000
0x7fffffffe270: 0x00000003      0x00000000      0x0060010e      0x00000000
0x7fffffffe280: 0x0000001f      0x00000000      0x00600106      0x00000000
(gdb) s
(gdb)  x/20x $sp
0x7fffffffe240: 0x00000064      0x00000000      0x00600126      0x00000000
0x7fffffffe250: 0x00000033      0x00000000      0x0060011e      0x00000000
0x7fffffffe260: 0x00000004      0x00000000      0x00600116      0x00000000
0x7fffffffe270: 0x00000003      0x00000000      0x0060010e      0x00000000
0x7fffffffe280: 0x0000001f      0x00000000      0x00600106      0x00000000
(gdb) s
_start_stack_push () at bubble_sort.s:102
(gdb)  x/20x $sp
0x7fffffffe238: 0x0060012e      0x00000000      0x00000064      0x00000000
0x7fffffffe248: 0x00600126      0x00000000      0x00000033      0x00000000
0x7fffffffe258: 0x0060011e      0x00000000      0x00000004      0x00000000
0x7fffffffe268: 0x00600116      0x00000000      0x00000003      0x00000000
0x7fffffffe278: 0x0060010e      0x00000000      0x0000001f      0x00000000
(gdb)
```

The gdb tracing above shows the progress in some of our earlier attempts to use the stack after the addition of the head. While the attempt to push to the stack was successful, the head element ran into some errors as it was causing problems when trying to add to the next_address list due to it no longer being purely sequential in memory.



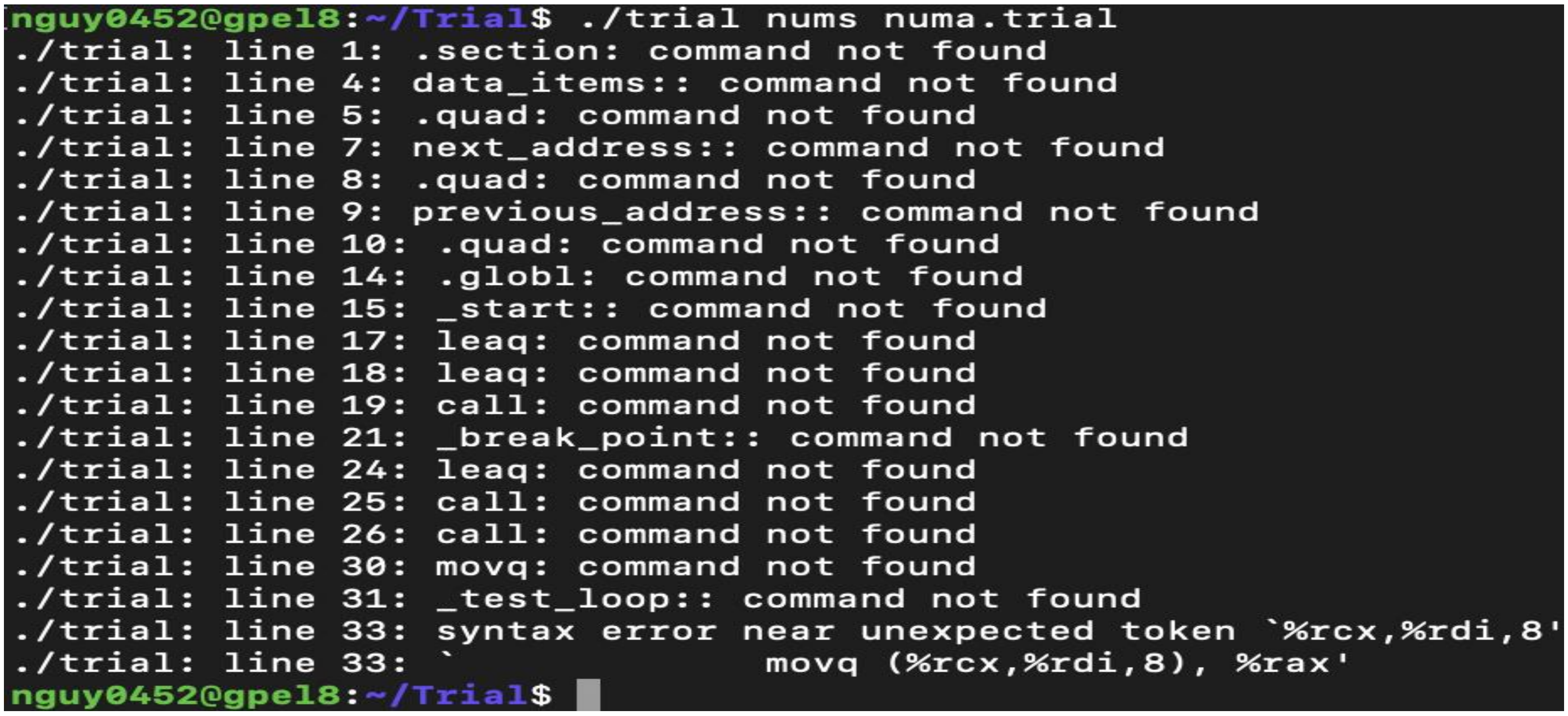
```
bate0046@gpel8: ~
Register group: general
rax            0x6001ff 6291967                    rbx            0x6001ef 6291951                    rcx            0x60021f 6291999
rdx            0x0      0                          rsi            0x0      0                          rdi            0x3      3
rbp            0x7fffffffe290   0x7fffffffe290     rsp            0x7fffffffe290   0x7fffffffe290     r8             0x0      0
r9             0x0      0                          r10            0x0      0                          r11            0x0      0
r12            0x0      0                          r13            0x0      0                          r14            0x0      0
r15            0x0      0                          rip            0x4001c4 0x4001c4 <_start_inner_insertion_loop+38>   eflags   0x213   [ CF AF IF ]
cs             0x33     51                         ss             0x2b     43                         ds             0x0      0
es             0x0      0                          fs             0x0      0                          gs             0x0      0

bubble_sort2.s
        .section .data
        # This example does not make use of the data section
        # Creation of a temp list to run bubble sort on -> no nodes created
        data_items:
             .quad 6,5,4,3,2,1,-1
        #Initialize the addresses to zero
        addresses:
             .quad 0,0,0,0,0,0,0,0,0,0

        left_addresses:
             .quad 0,0,0,0,0,0,0,0,0,0
        .section .text
        # The .text section is our executable code.

        # We need to make the symbol _start available to the loader (ld) and we do this
        # with .globl
native process 24367 In: _start_inner_insertion_loop                                   L242  PC: 0x4001c4
_start_inner_insertion_loop () at bubble_sort2.s:229
_start_inner_insertion_loop () at bubble_sort2.s:231
(gdb) x/14d &data_items
0x6001e7:       6       0       5       0
0x6001f7:       4       0       3       0
0x600207:       2       0       1       0
0x600217:       -1      -1
(gdb) x/14 & addresses
0x60021f:       6291959 0       6291951 0
0x60022f:       6291951 0       6291943 0
0x60023f:       6291975 0       6291983 0
0x60024f:       6291991 0
(gdb) x/14 &left_addresses
0x60026f:       0       0       0       0
0x60027f:       0       0       0       0
0x60028f:       0       0       0       0
0x60029f:       0       0
(gdb)
```

This is the gdb tracing that relates back to the problems we ran into regarding the doubly linked list. When sorting the values, we were unable to fulfill the double link due to the fact that the list would break the code, causing an infinite loop due to not having an ending flag. As we later found out, placing the flag at the beginning of the list would cause it to exit out of the address filling function.



```
bubble_sort.s: Assembler messages:
bubble_sort.s:157: Error: operand type mismatch for `lea'
bubble_sort.s:159: Error: operand type mismatch for `lea'
michael@DESKTOP-G7K6DD9:~/CS_3113/project1/CS3113_SP22_LAB01_TEAMMnemonic/IMPLEMENTATIONS$ as -g bubble_sort.s -o bubble_sort.o
bubble_sort.s: Assembler messages:
bubble_sort.s:161: Error: operand type mismatch for `lea'
bubble_sort.s:163: Error: operand type mismatch for `lea'
bubble_sort.s:206: Error: bad or irreducible absolute expression
bubble_sort.s:206: Error: expecting scale factor of 1, 2, 4, or 8: got `(index)'
```

In the gdb above, operand type mismatch type is caused by misusing lea. This was toward the beginning of our coding project before we tried using the stack. We were trying out various methods that might allow us to use the least amount of registers as possible, in our experimentation ended up causing this error.



```
nguy0452@gpel8:~/Trial$ ./trial nums numa.trial
./trial: line 1: .section: command not found
./trial: line 4: data_items:: command not found
./trial: line 5: .quad: command not found
./trial: line 7: next_address:: command not found
./trial: line 8: .quad: command not found
./trial: line 9: previous_address:: command not found
./trial: line 10: .quad: command not found
./trial: line 14: .globl: command not found
./trial: line 15: _start:: command not found
./trial: line 17: leaq: command not found
./trial: line 18: leaq: command not found
./trial: line 19: call: command not found
./trial: line 21: _break_point:: command not found
./trial: line 24: leaq: command not found
./trial: line 25: call: command not found
./trial: line 26: call: command not found
./trial: line 30: movq: command not found
./trial: line 31: _test_loop:: command not found
./trial: line 33: syntax error near unexpected token `%rcx,%rdi,8'
./trial: line 33: `             movq (%rcx,%rdi,8), %rax'
nguy0452@gpel8:~/Trial$
```

The gdb tracing above shows that a minor error we were running into in the beginning attempts to create a function. Inside, the bubble_sort.s contained multiple section.txt leading to a long error list as everything started to break. Unfortunately, it did take a while to realize the issue as to what could be causing this error.