REPORT

Part 2

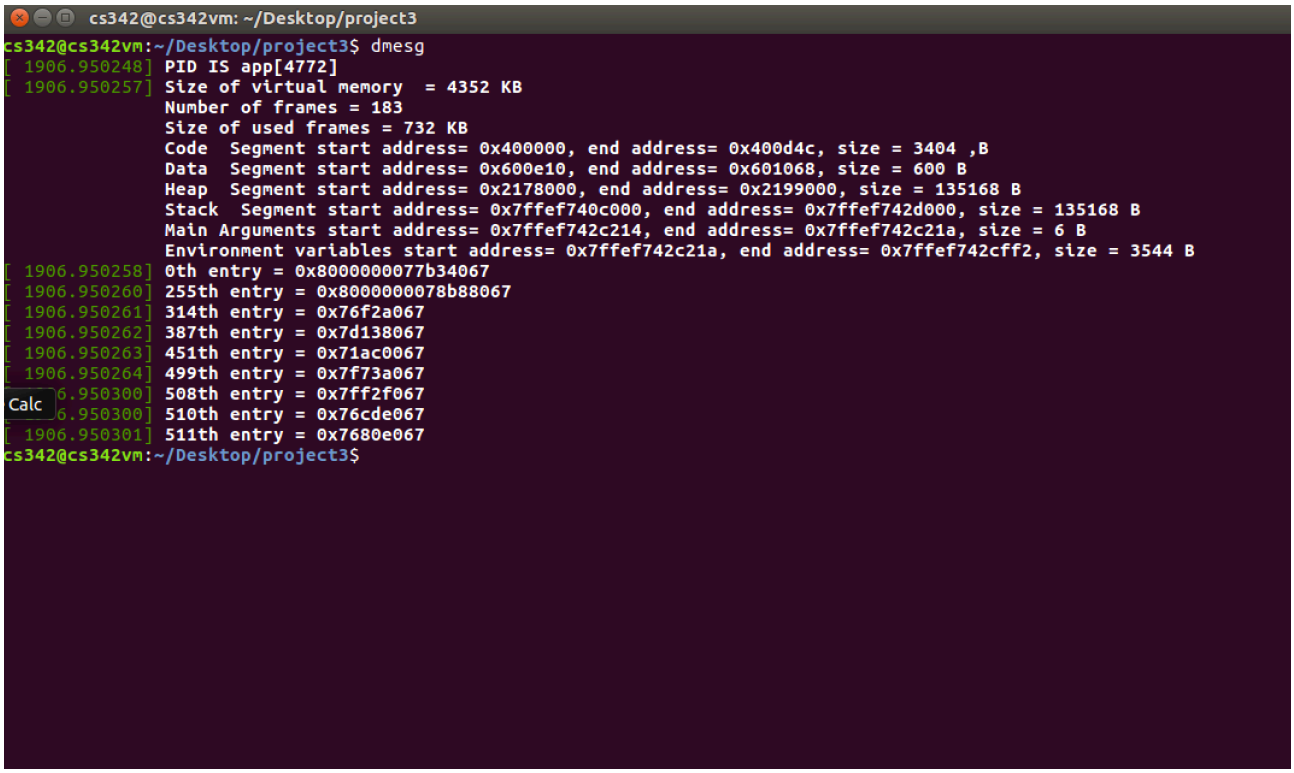
mymodule.c is a module that will be inserted to the kernel in order to display memory management of a specific process.

After make, by typing “sudo insmod ./mymodule.ko processid=K” the module is inserted. K is the process id whose memory management will be displayed.

Firstly, it will output the name and id of the process in the form of name_of_process[pid_of_process].
Secondly, start, end, and size of the code’s, data’s, stack’s, heap’s, main arguments’, environment variables’ virtual addresses, number of frames used by the process and its size, total virtual memory used by the process. Addresses are printed as hexadecimal, sizes are printed as decimal numbers.
Thirdly, top level page’s entry number and entry.

In order to print virtual addresses of memory blocks of the process, attributes in mm_struct belonging to the input process are used. Only for stack, its virtual memory ares’s start and end addresses are used. For top page table entry, pgd_t pgd attribute of mm_struct is used. 512 entries of pdg are parsed and then the valid ones are printed.

An example output of this module will be given below.



Part 3

app.c is designed in a way that it asks the user to choose an option repeatedly, until he chooses to exit. These are:

- Option 0: Exit
- Option 1: Call a recursive function, in other terms, use the stack
- Option 2: Allocate memory dynamically, in other terms, use the heap

During these steps, we look the status of the stack and heap by checking with the command "cat /proc/pid/maps". We get the process id of app.c by getpid() which is at the beginning of the main fuction. Also, we check virtual memory size and number of frames by the command "ps aux" at each step.

Stack

main function calls a recursive function f which has only one integer parameter, x. At the beginning, this value is 1. f(x) calls f(x+1) repeatedly. Whenever x mod 10000 is equal to 0, it asks to continue or not. Instead of putting sleep(sec), we checked the stack in this way. At the beginnng, size of the stack is 135168 B where its start address is 0x7ffd59d1a000 and its end address is 0x7ffd59d3b000, and VM size is 4352 KB, and number of frames is equal to 680. We observed that when x = 100000, its size becomes 4812800 B and its start address changes to 0x7ffd598a4000, and VM size is 8920 KB, and number of frames is equal to 5824.

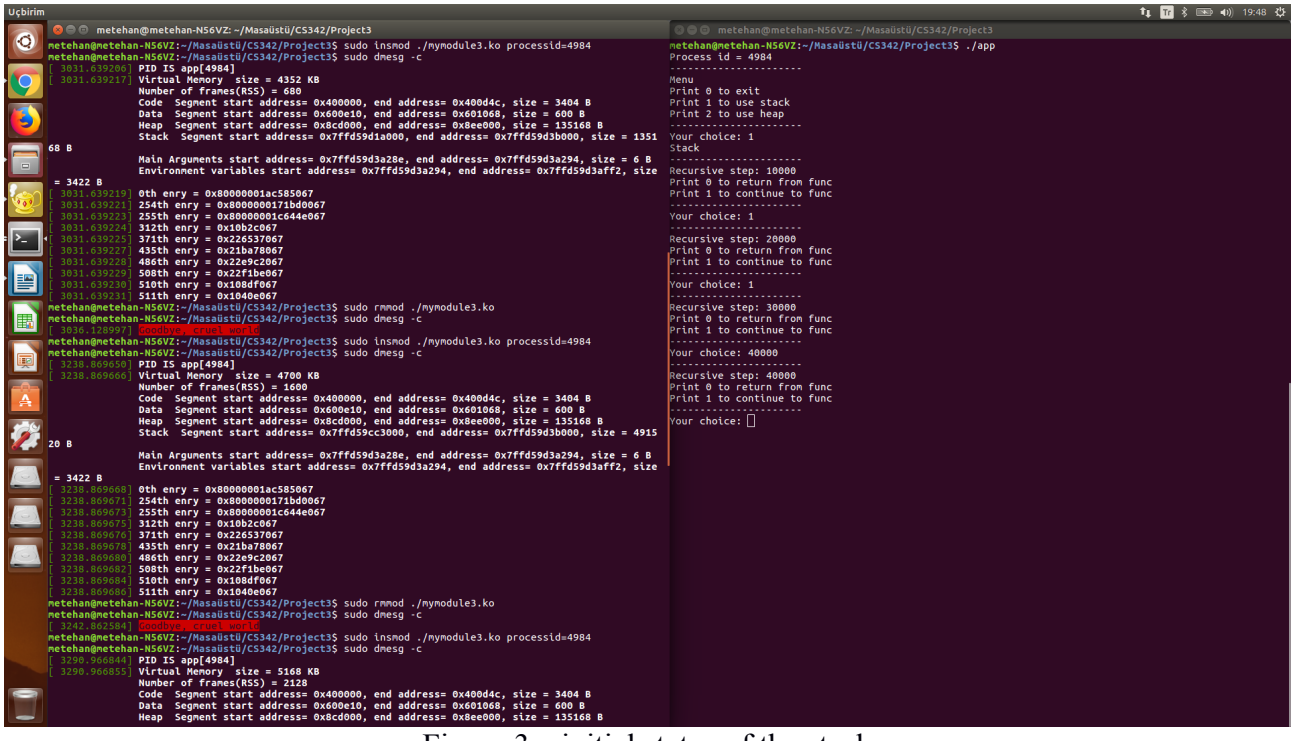


Figure 3a: initial status of the stack

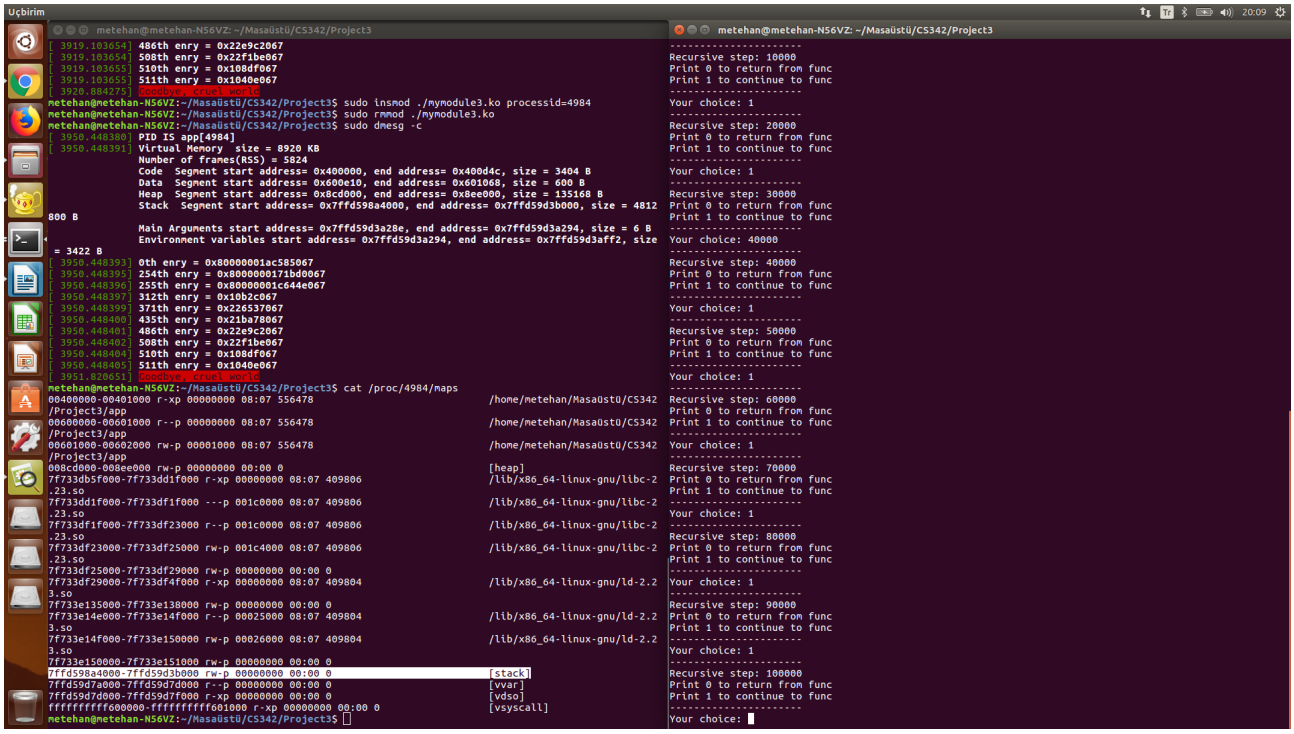


Figure 3b: final status of the stack (with cat /proc/pid/maps command)

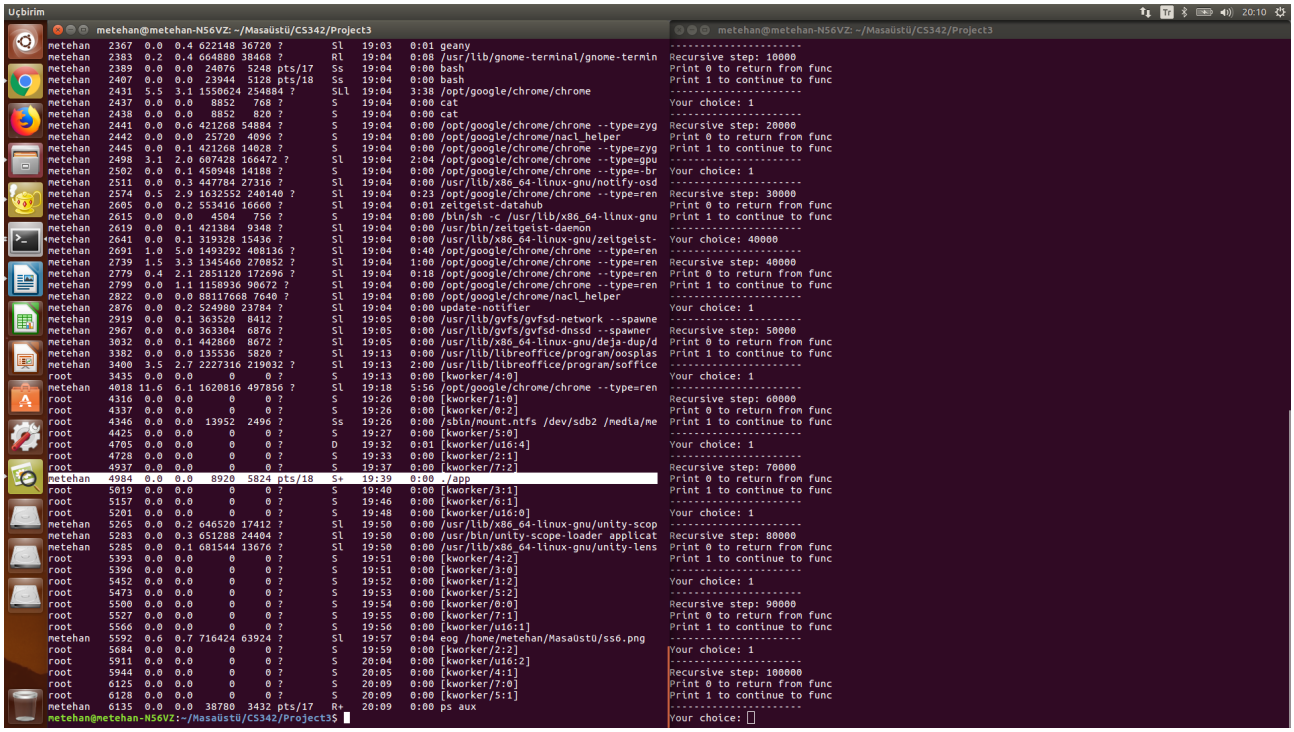


Figure 3c: final status of the stack (with ps aux command)

Heap

In main function, there is a loop that allocates 10000 integers at each step. After it allocates memory dynamically, asks for user's choice, to continue or not. At the beginning, size of the heap is 135168 B where its start address is 0xc6e000 and its end address is 0xc8f000, and VM size is 4352 KB, and number of frames is equal to 161. After 100000 integers are allocated, heap's end address

becomes 0xcdd000, and its size becomes 454656 B, and VM size becomes 4664 KB, and number of frames is equal to 161 which did not change. What we observed that, whenever stack size increases, VM size increases too.

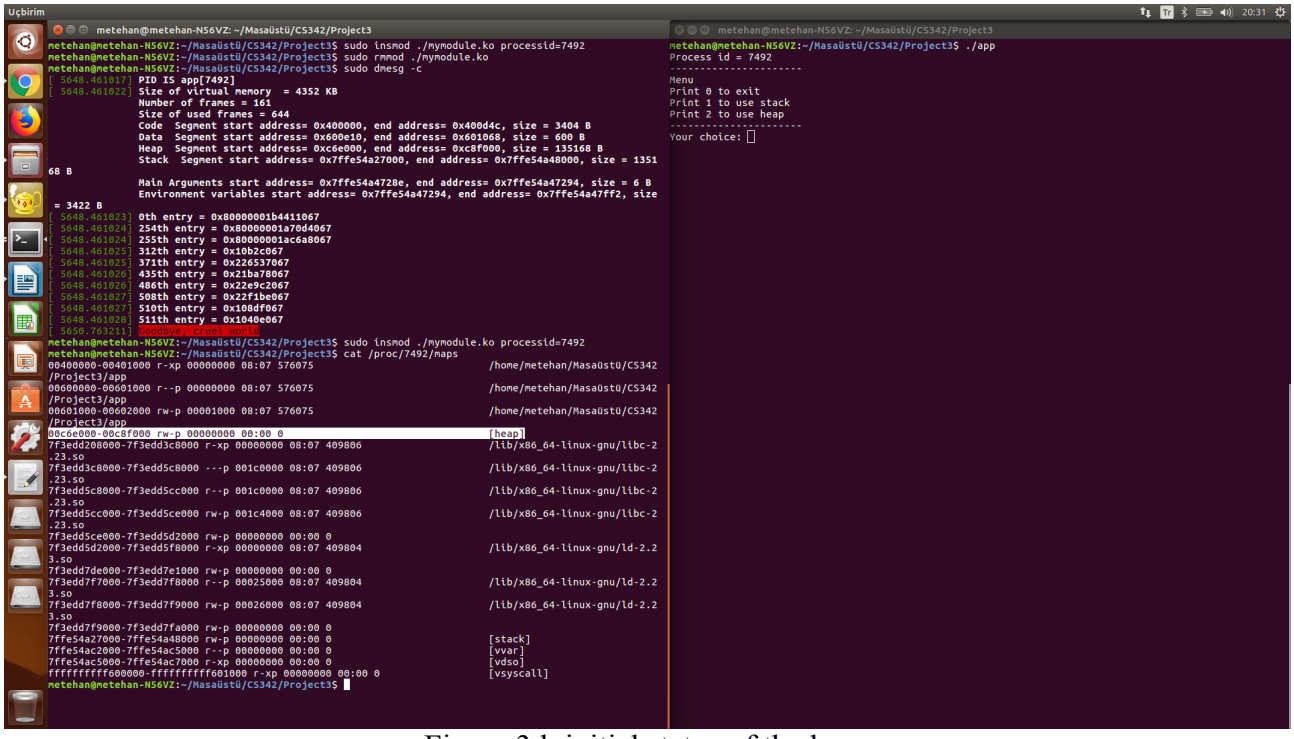


Figure 3d: initial status of the heap

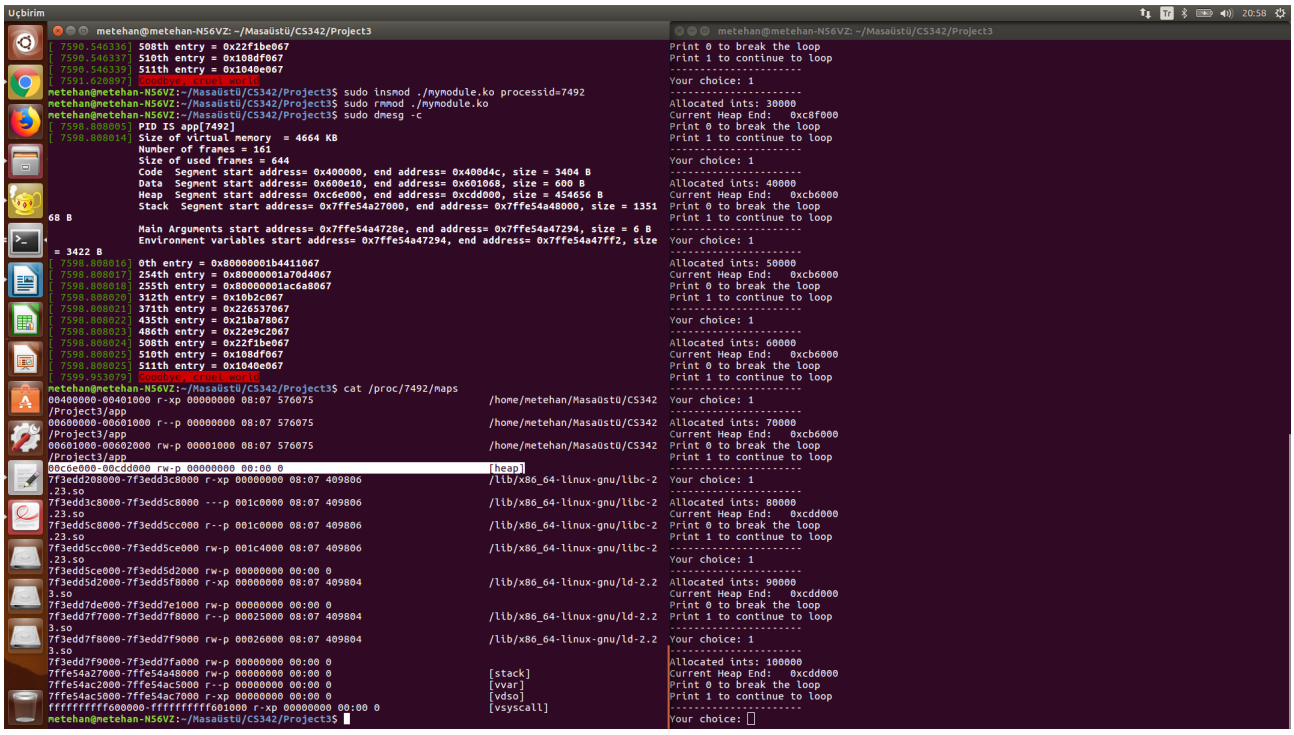
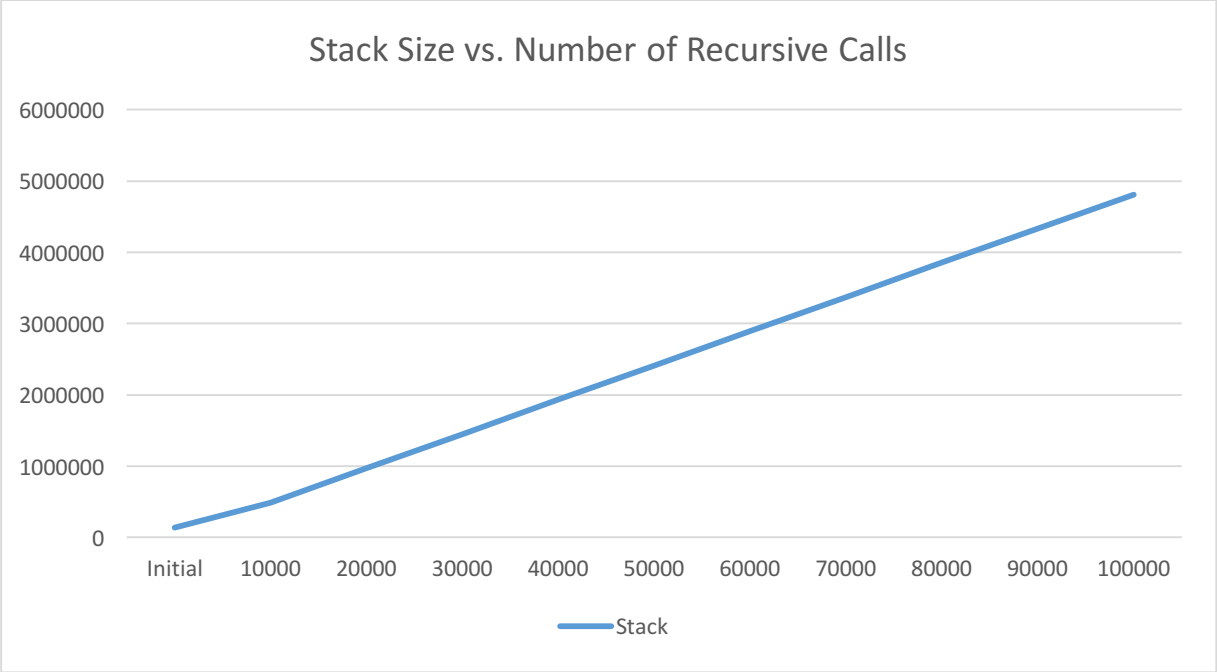


Figure 3e: final status of the heap

Plots and Tables

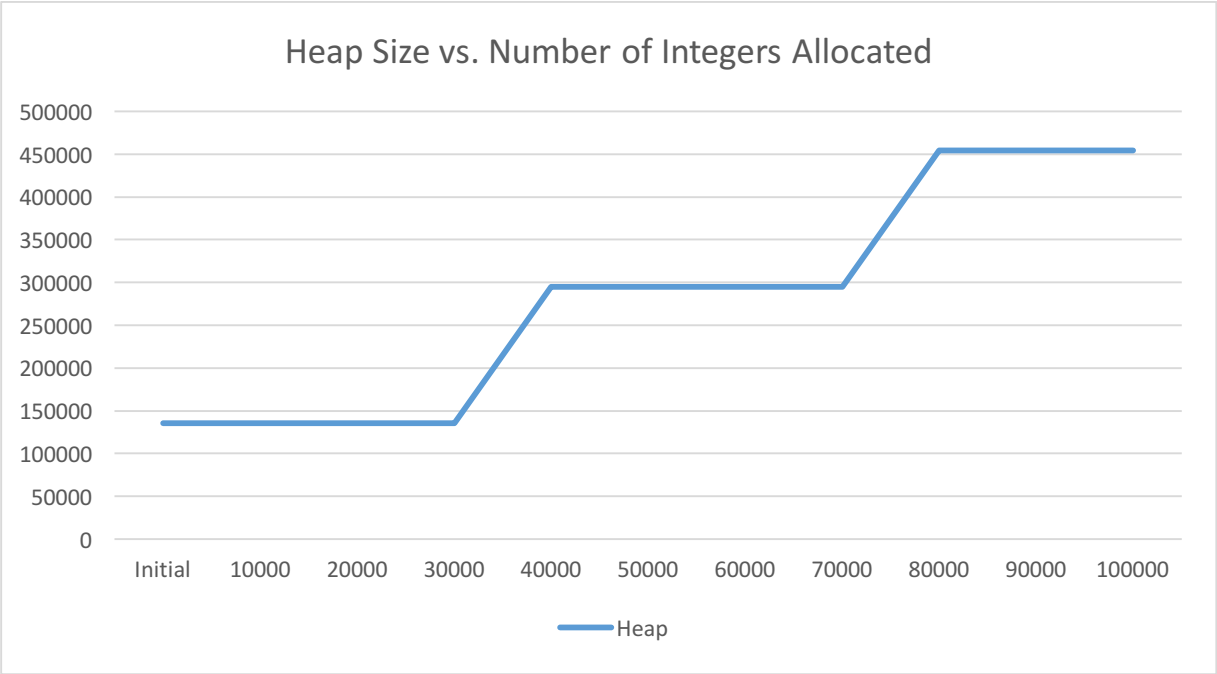
# of recursive calls (1 int for each)	Stack size (B)
Initial	135168
10000	491520
20000	970752
30000	1454080
40000	1933312
50000	2412544
60000	2891776
70000	3371008
80000	3854336
90000	4333568
100000	4812800

Table 3a: Stack



# of integers allocated	Heap size (B)
Initial	135168
10000	135168
20000	135168
30000	135168
40000	294912
50000	294912
60000	294912
70000	294912
80000	454656
90000	454656
100000	454656

Table 3b: Heap



Heap space with 135168 B was sufficient to store up to 30000 integers, similarly 294912 B was sufficient to store from 30000 to 70000 integers. Therefore, when we want to allocate 40000 and 80000 integers on the heap, its size grows immediately.

Stack grows much faster than heap because when we call a function, it will allocate more space than the same number of integers that are allocated in the heap. This is because function wants to allocate spae for its parameters, code, and some other data in the stack.