CS232 - Lab 3 Report

Name - Atishay Jain
Roll Number - 210050026

25th February 2023

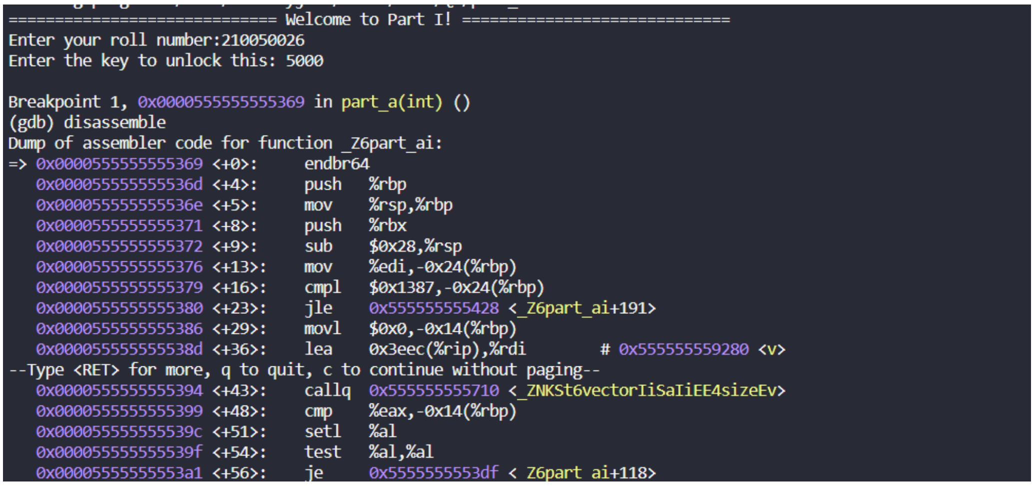
1 Question 1

We are provided with 3 binary executables named as part_a, part_b, part_c. We have to disassemble these executables, and understand them and obtain a valid output from them. I used `objdump` as well as `gdb` to disassemble the file and understand its contents.

1.1 part_a

The program takes in only a integer input [definition: part_a(int)] . A valid input results in a valid output while any other input terminates the program.

Figure 1: part of disassembled code for part_a



```
============================ Welcome to Part I! ============================
Enter your roll number:210050026
Enter the key to unlock this: 5000

Breakpoint 1, 0x0000555555555369 in part_a(int) ()
(gdb) disassemble
Dump of assembler code for function _Z6part_ai:
=> 0x0000555555555369 <+0>:     endbr64
   0x000055555555536d <+4>:     push   %rbp
   0x000055555555536e <+5>:     mov    %rsp,%rbp
   0x0000555555555371 <+8>:     push   %rbx
   0x0000555555555372 <+9>:     sub    $0x28,%rsp
   0x0000555555555376 <+13>:    mov    %edi,-0x24(%rbp)
   0x0000555555555379 <+16>:    cmpl   $0x1387,-0x24(%rbp)
   0x0000555555555380 <+23>:    jle    0x555555555428 <_Z6part_ai+191>
   0x0000555555555386 <+29>:    movl   $0x0,-0x14(%rbp)
   0x000055555555538d <+36>:    lea    0x3eec(%rip),%rdi        # 0x555555559280 <v>
--Type <RET> for more, q to quit, c to continue without paging--
   0x0000555555555394 <+43>:    callq  0x555555555710 <_ZNKSt6vectorIiSaIiEE4sizeEv>
   0x0000555555555399 <+48>:    cmp    %eax,-0x14(%rbp)
   0x000055555555539c <+51>:    setl   %al
   0x000055555555539f <+54>:    test   %al,%al
   0x00005555555553a1 <+56>:    je     0x5555555553df <_Z6part_ai+118>
```

The above part of disassembled code shows the `part_a` function. In this, after pushing the base pointer another pointer is pushed which is the input arguement of the function. Then, it compares $0x1387$ with the input. The decimal form of this hexadecimal number is 4999. Then it uses `jle` instruction and jumps to address of `part_a+191` if $4999 \leq input$. It jumps to another instructions at address 1428 if this comparison is true. So, I got a hint from this part of code that there is some kind of comparison of the input which we type as secret key with the number 4999. So, I tried running the program for some secret keys around 4999 and found the property of secret key as :

1.1.1 Property of Secret Key

The Secret Key provided should be a number which is greater than 4999. That is any number greater than 4999 works as secret key. Note that 4999 does not works for secret key, however 5000 and onwards work.

$$secret\_key > 4999$$

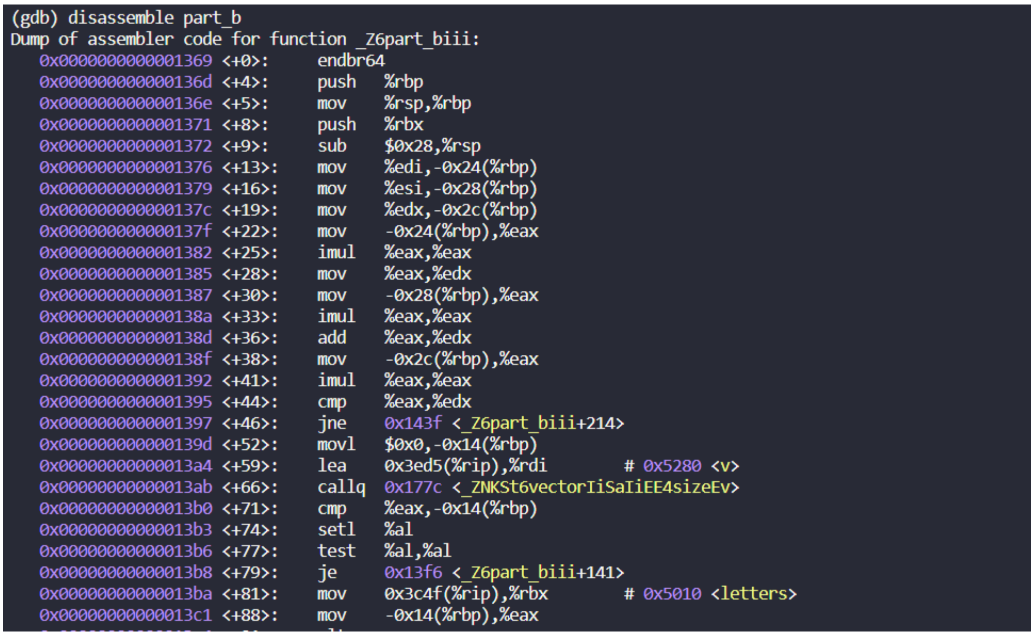
1.1.2 Flag and secret number

Key used : 62525 , as 62525 > 4999
Secret Number obtained : 863113432
Flag : CS230{

1.2 part_b

The program takes in 3 integer inputs [definition: part_b(int,int,int)]. Given a valid triplet you obtain a valid output, while every other branch terminates. Here, the function `part_b`

Figure 2: part of disassembled code for part_b



```
(gdb) disassemble part_b
Dump of assembler code for function _Z6part_biii:
   0x0000000000001369 <+0>:     endbr64
   0x000000000000136d <+4>:     push   %rbp
   0x000000000000136e <+5>:     mov    %rsp,%rbp
   0x0000000000001371 <+8>:     push   %rbx
   0x0000000000001372 <+9>:     sub    $0x28,%rsp
   0x0000000000001376 <+13>:    mov    %edi,-0x24(%rbp)
   0x0000000000001379 <+16>:    mov    %esi,-0x28(%rbp)
   0x000000000000137c <+19>:    mov    %edx,-0x2c(%rbp)
   0x000000000000137f <+22>:    mov    -0x24(%rbp),%eax
   0x0000000000001382 <+25>:    imul   %eax,%eax
   0x0000000000001385 <+28>:    mov    %eax,%edx
   0x0000000000001387 <+30>:    mov    -0x28(%rbp),%eax
   0x000000000000138a <+33>:    imul   %eax,%eax
   0x000000000000138d <+36>:    add    %eax,%edx
   0x000000000000138f <+38>:    mov    -0x2c(%rbp),%eax
   0x0000000000001392 <+41>:    imul   %eax,%eax
   0x0000000000001395 <+44>:    cmp    %eax,%edx
   0x0000000000001397 <+46>:    jne    0x143f <_Z6part_biii+214>
   0x000000000000139d <+52>:    movl   $0x0,-0x14(%rbp)
   0x00000000000013a4 <+59>:    lea    0x3ed5(%rip),%rdi        # 0x5280 <v>
   0x00000000000013ab <+66>:    callq  0x177c <_ZNKSt6vectorIiSaIiEE4sizeEv>
   0x00000000000013b0 <+71>:    cmp    %eax,-0x14(%rbp)
   0x00000000000013b3 <+74>:    setl   %al
   0x00000000000013b6 <+77>:    test   %al,%al
   0x00000000000013b8 <+79>:    je     0x13f6 <_Z6part_biii+141>
   0x00000000000013ba <+81>:    mov    0x3c4f(%rip),%rbx        # 0x5010 <letters>
   0x00000000000013c1 <+88>:    mov    -0x14(%rbp),%eax
```

takes 3 inputs at addresses -0x24(%rbp), -0x28(%rbp), -0x2c(%rbp) and loads them into some registers. Then, it multiples the value at register `eax` by itself, thus squaring the first input (the one at -0x24) and stores this squared value at `edx`. Then, it squares the second input at -0x28 by again loading it into `eax` and multiplying it by itself and adds this square of second input obtained to the square of first input obtained. Let's say for convenience that the inputs were a, b, c inorder. So, by now it has calculated $a^2 + b^2$ and has stored it in `edx`.
Now it loads the third input (at -0x2c) into `eax`, again multiplies it by itself, thus squaring it. And it compares this squared value of third input (c^2) with the previous value stored at `edx`, which is $a^2 + b^2$. If they are not equal, it jumps by `jne` to the end part of function otherwise continues the procedure. Further on running the executable with taking this comparison and calculation into consideration, the consistion of secret is as :

1.2.1 Property of Secret Key

The inputs must be in a **Pythagorean Triplet**. Lets say the inputs (in order) are a, b, c, then

$$a^2 + b^2 = c^2$$

That is the third input's square should be the sum of squares of other 2 inputs

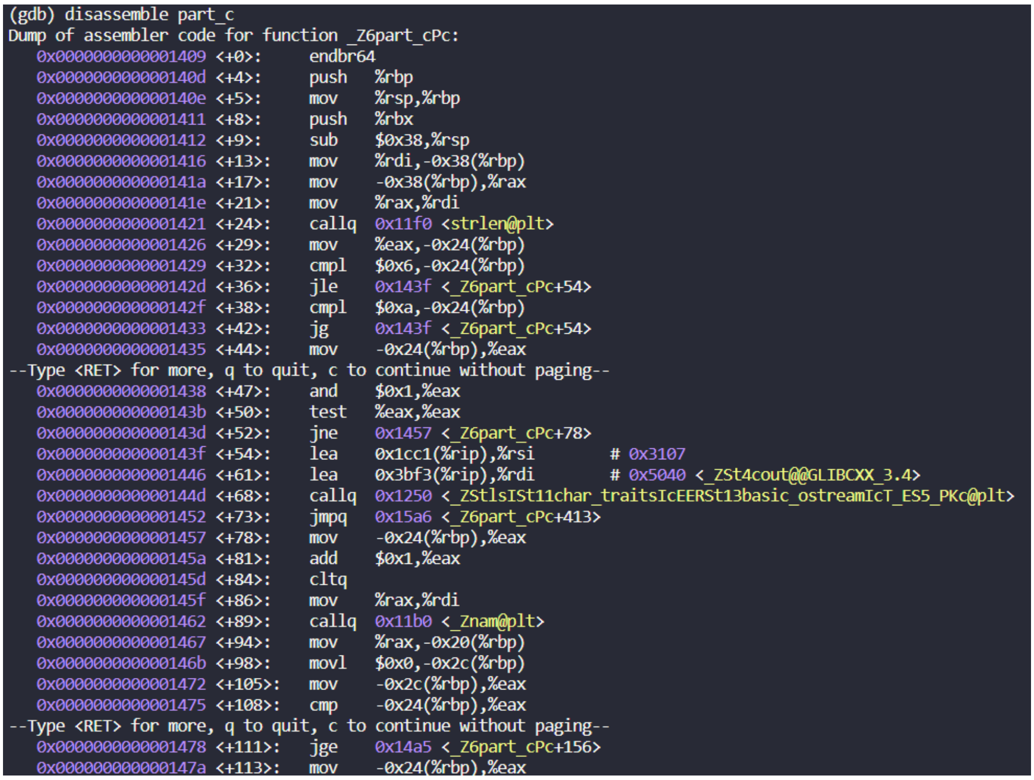
1.2.2 Flag and secret number

Key used : 15817 , as $15^2 + 8^2 = 17^2$
Secret Number obtained : 93144717
Flag : is_easy!!}

1.3 part_c

The program takes in a string input [definition: part_c(char*)]. A specific null-terminated string results in a valid output, in all other cases the program terminates It is first calculating

Figure 3: part of disassembled code for part_b



```
(gdb) disassemble part_c
Dump of assembler code for function _Z6part_cPc:
   0x0000000000001409 <+0>:     endbr64
   0x000000000000140d <+4>:     push   %rbp
   0x000000000000140e <+5>:     mov    %rsp,%rbp
   0x0000000000001411 <+8>:     push   %rbx
   0x0000000000001412 <+9>:     sub    $0x38,%rsp
   0x0000000000001416 <+13>:    mov    %rdi,-0x38(%rbp)
   0x000000000000141a <+17>:    mov    -0x38(%rbp),%rax
   0x000000000000141e <+21>:    mov    %rax,%rdi
   0x0000000000001421 <+24>:    callq  0x11f0 <strlen@plt>
   0x0000000000001426 <+29>:    mov    %eax,-0x24(%rbp)
   0x0000000000001429 <+32>:    cmpl   $0x6,-0x24(%rbp)
   0x000000000000142d <+36>:    jle    0x143f <_Z6part_cPc+54>
   0x000000000000142f <+38>:    cmpl   $0xa,-0x24(%rbp)
   0x0000000000001433 <+42>:    jg     0x143f <_Z6part_cPc+54>
   0x0000000000001435 <+44>:    mov    -0x24(%rbp),%eax
--Type <RET> for more, q to quit, c to continue without paging--
   0x0000000000001438 <+47>:    and    $0x1,%eax
   0x000000000000143b <+50>:    test   %eax,%eax
   0x000000000000143d <+52>:    jne    0x1457 <_Z6part_cPc+78>
   0x000000000000143f <+54>:    lea    0x1cc1(%rip),%rsi        # 0x3107
   0x0000000000001446 <+61>:    lea    0x3bf3(%rip),%rdi        # 0x5040 <_ZSt4cout@@GLIBCXX_3.4>
   0x000000000000144d <+68>:    callq  0x1250 <_ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc@plt>
   0x0000000000001452 <+73>:    jmpq   0x15a6 <_Z6part_cPc+413>
   0x0000000000001457 <+78>:    mov    -0x24(%rbp),%eax
   0x000000000000145a <+81>:    add    $0x1,%eax
   0x000000000000145d <+84>:    cltq
   0x000000000000145f <+86>:    mov    %rax,%rdi
   0x0000000000001462 <+89>:    callq  0x11b0 <_Znam@plt>
   0x0000000000001467 <+94>:    mov    %rax,-0x20(%rbp)
   0x000000000000146b <+98>:    movl   $0x0,-0x2c(%rbp)
   0x0000000000001472 <+105>:   mov    -0x2c(%rbp),%eax
   0x0000000000001475 <+108>:   cmp    -0x24(%rbp),%eax
--Type <RET> for more, q to quit, c to continue without paging--
   0x0000000000001478 <+111>:   jge    0x14a5 <_Z6part_cPc+156>
   0x000000000000147a <+113>:   mov    -0x24(%rbp),%eax
```

the length of input string and checking if it between 6 to 10. Then, it performs a binary operation between length of the input as 0x1, which is one. This shows that the length of the input should be odd. Now, it reverses the string by a loop and adds a null character at its end to make it null terminated and compares it with input. So its reverse should be same as input, therefore it should be a palindrome, otherwise it jumps to end.

1.3.1 Property of Secret Key

The secret key should be a string with the following properties:

- Its length should between 6 to 10
- It should be palindrome
- The length of the string should be odd, so only of length 7 or 9 are allowed

1.3.2 Flag and secret number

Key used : `qwerewq`
Secret Number obtained : 165192890
Flag : R3v3rse_Engine3ring_

`qwerewq` is a correct key as it's reverse is same as this (palindrome) and it is a string of length 7 (should be 7 or 9)

1.4 Final FLAG

Final flag = part 1's output + part 3's output + part 2's output
Final flag = CS230{R3v3rse_Engine3ring_is_easy!!}

2 Question 2

We have to write an assembly program (`inverse.s`) using **MIPS32** ISA to find the inverse of a number a modulo m. The inverse of a modulo m is the number x ($0 < x < m$) such that

$$ax = 1 \quad (mod\ m)$$

Since it was given in the constraints that a and m are **coprime** (i.e. $gcd(a, m) = 1$) , I used the **Extended Euclidean algorithm** for finding the modular inverse x.
The Extended Euclidean algorithm takes the 2 integers 'a' and 'b', then finds their gcd as well as two numbers x and y such that

$$ax + by = gcd(a, b)$$

For our problem of finding multiplicative inverse of a modulo m, we put $b = m$ in the above equation. As we already know that a and m are relatively prime, the value of $gcd(a, m)$ is 1 and we can use it here

$$ax + my = 1$$

Taking modulo m on both sides, we get

$$ax + my \simeq 1 \quad (mod\ m)$$

Note that my $(mod\ m)$ would be $= 0$ for an integer y, we get

$$ax \simeq 1 \quad (mod\ m)$$

Hence, the x which we got using the Extended Euclid Algorithm is the multiplicative inverse of a modulo m. Below is a reference code that illustrates this idea :

Code 1: C++
Reference code for finding modulo inverse

```cpp
// Iterative C++ program to find modular
// inverse using extended Euclid algorithm

#include <bits/stdc++.h>
using namespace std;

// Returns modulo inverse of a with respect to m using extended
// Euclid Algorithm with Assumption: a and m are coprimes
// i.e., gcd(A, M) = 1
int modInverse(int A, int M){
        int m0 = M;
        int y = 0, x = 1;
        if (M == 1) return 0;
        while (A > 1) {
                int q = A / M;      // q is quotient
                int t = M;          // m is remainder now, process
                M = A % M;          // same as Euclid's algo
                A = t;
                t = y;
                y = x - q * y;      // Update y and x
                x = t;
        }
        if (x < 0){                 // Make x positive
            x += m0;
        }
        return x;
}

// Driver Code
int main(){
        int A = 3, M = 11;
        cout << modInverse(A, M) << endl;
}
```

Time Complexity: $\mathcal{O}(\log m)$
Auxiliary Space: $\mathcal{O}(1)$

I used the above C++ version of the idea of calculating inverse modulo m and implemented this C++ code in **MIPS** Assembly.
I did not accounted for $m = 1$ in my code as it was given that $a > 0$ and $m > a$, thus $m > 1$. Also, I have not implemented the Bonus part.

3 Question 3

We have to write an assembly program(`inplacemergesort.s`) using **MIPS32** ISA to sort an array of numbers using **In-place Merge Sort** (merge sort without using extra additional space).

For making the merging operation "inplace", I used tricks of division and modulus, i.e. I stored 2 elements values at 1 index and used division and modulus for extracting them. The approach was -

3.1 Inplace Merging

- First, we have to find the a value which is greater than all the elements of array. I did this by finding out the maximum element of the array (let it be `maxele`) and taking the desired value to be $maxelement + 1$. Finding out `maxele` doesn't took extra time as I did it along with taking input of array and maintaining a variable for tracking max element.
- Now, I stored the original value an element of array as modulus and its second value as division modulo `maxele`
- For example, lets say I want to store two values $arr[i]$ and $arr[j]$ both at index i (i.e. in $arr[i]$). I will store them as

  $$arr[i] = arr[i] + arr[j] * maxele$$

  Now, I can access original value of $arr[i]$ by $arr[i]\%maxele$ and $arr[j]$ by $arr[i]/maxele$.
- The basic idea is based on Euclidean Division, that is

  $$dividend = divisor * quotient + remainder$$

  - divisor = `maxele`, which I took as maximum element of array + 1
  - quotient = $min(first, second)$
  - remainder = original value of element
  - Example : 7/4 = Q:1 R:2, applying euclidean: $4 * 1 + 3 = 5$ (dividend)
  - Now, first = $arr[i]\%divisor$
  - second = $arr[j]\%divisor$
  - encoded element = remainder + quotient*divisor
- Note that this works for non-negative elements of array and thus it works here as according to constraints $0 <= a_i <= 10000$

In usual normal mergesort, in this merging operation we create two temporary arrays for copying the subarrays and then merging them. However, here I am not using any extra additional space. I am just using 3 variables and some iterative loops which use the technique described above, thus the auxiliary space used by this algorithm is $\mathcal{O}(1)$.
Also, the parameters for which the iterations are happening are subarrays of original array, whose indices are sent here by `mergesort()` function by dividing original array, leading to $\mathcal{O}(n \log n)$ time complexity. Below is the C++ code demonstrating this, I converted this code into MIPS Assembly.

Code 2: C++
Inplace Merging

```
// C++ program to sort an array using merge sort such
// that merge operation takes O(1) extra space
void merge(int arr[], int beg, int mid, int end, int maxele){
    int i = beg;
    int j = mid+1;
    int k = beg;
    while (i <= mid && j <= end) {
        if (arr[i] \% maxele <= arr[j] \% maxele) {
            arr[k] = arr[k] + (arr[i] % maxele) * maxele;
            k++;
            i++;
        }
        else {
            arr[k] = arr[k] + (arr[j] % maxele) * maxele;
            k++;
            j++;
        }
    }
    while (i <= mid) {
        arr[k] = arr[k] + (arr[i] % maxele) * maxele;
        k++;
        i++;
    }
    while (j <= end) {
        arr[k] = arr[k] + (arr[j] % maxele) * maxele;
        k++;
        j++;
    }
    for (int i = beg; i <= end; i++){  // Obtaining actual values
        arr[i] = arr[i] / maxele;
    }
}
```

3.2 Mergesort function with no recursive stack

Code 3: C++

Iterative calls for mergesort

```
void mergeSort(int arr[], int n, int max_elem){
   int curr_size;  // current size of subarrays to be merged
   int left_start; // picking start index of left subarray
                   // to be merged

   // Merge subarrays in bottom up manner.  First merge subarrays of
   // size 1 to create sorted subarrays of size 2, then merge
      subarrays
```

```
    // of size 2 to create sorted subarrays of size 4, and so on.
    for (curr_size=1; curr_size<=n-1; curr_size = 2*curr_size){
        // Pick starting point of different subarrays of current size
        for (left_start=0; left_start<n-1; left_start += 2*curr_size)
            {
            // Find ending point of left subarray. mid+1 is starting
            // point of right
            int mid = min(left_start + curr_size - 1, n-1);
            int right_end = min(left_start + 2*curr_size - 1, n-1);
            // Merge Subarrays arr[left_start...mid] & arr[mid+1...
               right_end]
            merge(arr, left_start, mid, right_end, max_elem, arr_cp);
        }
    }
}
```

4 Question 4

We had to perform matrix multiplication in **x86** assembly in this question.

4.1 Memory Management

After analyzing the code given and understanding the running instructions & constraints, the first task inorder to step into this question was to allocate memory for the matrices.

- For allocating memory, I decided to use heap. This was because when I used stack, it gave segmentation fault due to overflow of stack memory when testing the multiplication at higher values of order of matrices.
- For implementing heap, I got to know about 2 ways, one of them was using `malloc()` function, but it was **not** allowed as we were not allowed to use any C/C++ functionality. Therefore I used the other option of using `mmap` by the `sys_mmap` linux system call for x86. It took me a lot of time to figure out how this thing works.
- I noticed that before every *TODO*, the value of r_i and c_i, that is the number of rows & number of columns of the matrix for which we have to allocate memory was being stored in registers `rcx` and `rax` respectively.
- The area of memory that needs to be allocated for a matrix of order r_ixc_i would be $(r_i \times c_i) * 8$. This is because there are r_ixc_i elements in that matrix and as each is 64-bit, each requires 8 bytes of memory
- So, I multiplied `rax` and `rcx` and again multiplied then used `shl` command with 3 as parameter and stored this size in *rcx*
- Now, I had to use `mmap`. For this, it requires some parameters, which I set as -
  - `rax` as 9, because 9 is the system call number for `mmap`

- `rsi` as `rcx`, as I stored the size to be allocated in `rcx`
- `rdx` as $0x3$, which denotes the property of new memory region to be writable
- `r8` as -1, which denotes the file descriptor
- `rdi` as 0, which ensures that operating system will choose mapping destination
- `r9` as 0, denoting offset
- `r10` as 34, which denotes map_anonymus + map_private

- And after these many parameters, a final `syscall`. After which the register `rax` now contains the initial pointer of that memory allocated
- So, I stored the value of `rax` into the desired parameter a_i of the bss section

In this way, I allocated the memory for all the 3 matrices. For testing the memory allocation, I ran `./memtest.o < mem_test.inp` and its other values (except first one) matched with the given correct output and gave no segfault, denoting that the memory is successfully allocated.

4.2 Matrix Multiplication

There were 6 different assembly files for matrix multiplication, which differed only in the order of the iterative loops in them. Coding the first one took time, then rest were just manipulating the iterative terms and other values. The way I did this is -

- The values of $r1, c1, a1, a2, a3$ were loaded in the files with `push` command as the arguements to the multiplication function.
- There were 3 other registers $r11$, $r12$, $r13$ which I used in my files as i, j, k respectively
- For looping the 3 loops, i used these registers and looped over them comparing with r1, c1, c2
- Then I accessed the matrix value as instructed and multiplied it in each program

4.3 Time Taken Calculation and Plot

I made a python script that runs all the 6 files for the given different values of $N = 128, 256, 512, 1024, 2048$. As the first number at the output of the files denoted th number of cycles taken, I found the TSC ferquency of my device and divided the number of cycles for each of the program by TSC frequency to obtain the time taken by it in seconds.
For finding the TSC frequency, I used -

```
cat /proc/cpuinfo | grep "cpu MHz"
```

It provided the TSC frequency. However, if I run it on different times, the TSC frequency was different depending on CPU usage.
Suppose the clocks taken by a program were c_i, so I calculated the times taken by it (t_i) as

$$t_i = \frac{c_i}{TSC\_FREQUENCY}$$

The below plot shows the time taken by each program for each value of input size. The TSC frequency used for this plot was 2419753418 Hz

Figure 4: Number of cycles (c_i) for each N for each program

N	ijk	ikj	jik	jki	kij	kji
128	9361929	7436462	9450210	8873081	7368530	8915911
256	91347518	58234371	107152072	83239234	60570989	80678375
512	1062066252	504889498	1093144204	1283751333	514718139	1255139217
1024	20888242616	3977899166	15884859568	29704166199	4285410693	31054893451
2048	220486943385	31108105388	183285415210	472027807679	40962387398	457912462353

Figure 5: Matrix multiplication time for different values of N