CS232-Lab3 Report

210050115 - Patil Vipul Sudhir

1 Q1 (Capture the Flag!)

1.1 Keys and flag

Ques	Roll No.	key	flag	secret number
part_a	210050115	5001	CS230{	774302307
part_b	210050115	3 4 5	is_easy!!}	864131143
part_c	210050115	abcdcba	R3v3rse_Engine3ring_	1013646892

Table 1: Keys and flags for subparts of Q1

Final FLAG: CS230{R3v3rse_Engine3ring_is_easy!!}

1.2 Explanation

1.2.1 part_a

1. Main takes 2 inputs- roll number and key by calling >> operator 2 times (pc-14ef & 1515) and key is stored into register edi (pc-1529 & 152c)
2. Main calls function part_a(int) which is at pc 1369(pc-1547)
3. In part_a the value of key is moved from edi to [rbp-0x24] register(pc-1376)
4. Value of key is checked whether it is greater than 4999(0x1387) or not.(pc-1379)
5. If value of key value is not greater than 4999 then it jumps to pc-1428 (Which is error message)
6. If key is greater than 4999 then it prints flag and secret number(pc-13d4 & 13ed)

1.2.2 part_b

1. Main takes 4 inputs- roll number and 3 numbers by calling >> operator 4 times (pc-1522, 1548, 155a, 156c) and these 3 inputs are moved to registers edi, edx, esi(pc-158c, 1595, 1597)

2. Main calls function part_b(int, int, int) which is at pc 1369 (pc-1599)

3. Values from registers edi, edx, esi are moved to registers [rbp-0x24],[rbp-0x2c],[rbp-0x28] respectively and stored them in eax register.(pc-1379, 137c, 1379)

4. Used imul to calculate square of [rbp-0x24] and [rbp-0x28] and using add they are added.(pc-1382,138a,138d,138f)

5. Used imul again to calculate square of [rbp-0x2c] and stored it in edx register.(pc-1392)

6. Compared the value of edx and eax registers and if they are not equal it will jump to pc 143f (pc-1395,1397) which gives error message

7. If equal then it prints flag and secret number(pc-13e4 & 1404)

1.2.3 part_c

1. Main takes 2 inputs- roll number and key by calling >> operator 2 times (pc-1700)

2. Main calls function part_c(char*) which is at pc 1409 (pc-174d)

3. Value of parameter of function (key) is moved to register [rbp-0x24] from register eax

4. If the value in [rbp-0x24] is less than or equal to 6 it is jumping to 143f which is an error message.(pc-1429)

5. Similarly it is checking if its length is greater than 10 also it jumps to 143f which is also gives error message. (pc-142f)

6. Then it uses operation 'and' with last bit of length of string if it is 1 (to check number is odd or not) if result is not 1 (string length is not odd) it jumps to pc 1457 which prints error message (pc-1427)

7. If string length is odd then it will go into a for loop which checks last and first letter of the string is same or not if same it checks $2nd$ and last $2nd$ letter and it goes until element is common. (It checks if the given input in palindrome of odd length or not). Here instruction sets the low byte of rax register to 1 if the previous comparison (cmp) resulted in a "less than" condition, and 0 otherwise.(pc-14f1,14f4,14f7)

8. After checking string at last it checks low byte of rax register is zero or not. If zero it jumps to pc 14e5 which prints error message. This instruction tests the al register to see if it is zero. (pc-1535)

9. If not it it prints flag and secret number(pc-153e,1551)

2 Q2 (Find the inverse modulo m)

2.1 Idea

Program takes a and m as inputs using syscall of MIPS. Storing the values of a and m into a register Given a and m I'm finding x, y such that
ax+my=gcd(a,m)
ax+my=1 (as gcd(a,m) is 1)
By taking modulo m on both sides, we get
$ax \cong 1(mod(m))$
so x is inverse of a modulo m. x and y can be found using Extended Euclid Algorithm. After getting value of x using syscall of MIPS it prints it.
Extended Euclid Algorithm:
The extended Euclidean algorithm updates the results of gcd(a, b) using the results calculated by the recursive call gcd(b%a, a). Let values of x and y calculated by the recursive call be x_1 and y_1. x and y are updated using the below expressions.
ax + by = gcd(a, b)
gcd(a, b) = gcd(b%a, a)
gcd(b%a, a) = (b%a)x1 + ay1
ax + by = (b%a)x1 + ay1
ax + by = (b – [b/a] * a)x1 + ay1
ax + by = a(y1 – [b/a] * x1) + bx1
Comparing LHS and RHS,
x = y1 – b/a * x1
y = x1
So with only log(m) iterations we can find inverse of a modulo m.
Time Complexity: O($logm$)

3 Q3 (Merge sort with a twist!)

Taking Inputs:
First program takes a input which is size of array(n). Then it loops n times to get n inputs and store it in memory (defined as array).
Creating Partitions:
As we cannot use recursive merge sort (because we cannot store return pointers for recursive call in stack as it occupies extra spaces) so I used iterative merge sort. I'm merging subarrays in bottom up manner i.e first merging subarrays of size 1 to create a subarray of size 2, then merging subarray of size 2 to create

subarrays of size 4, and so on.
Merge Function:
Normally while merging we create a new array but as per the constraints merge sort should not use extra space and also given that input values are less than or equal to 10000($\leq 2^{14}$) So I can use upper remaining bits (31-14=17 bits) to store my output.
While taking input I'm finding out maximum element from given inputs and defined MAX = (maximum element +1). Now using operations modulus and division I can save 2 numbers in one 32-bit register. Suppose if I want to store arr[j] at index i then I'm storing it as arr[i]= arr[i] + arr[j]*MAX. So if I want to recover original element arr[i] then arr[i]%MAX gives original arr[i] and arr[j]/MAX will give original arr[j].While merging 2 arrays I'm using same trick and storing sorted array in upper 17 bits.
Printing Outputs:
Program will start a loop which iterates n times. While printing the output I'm printing element/MAX which gives the sorted array.
Time complexity: O($nlogn$)
Space complexity: O(1)

4 Q4 (Curious case of Matrix Multiplication)

Memory Allocation:
Used mmap (syscall 9) for memory allocation with appropriate flags (MAP_ANONYMOUS, MAP_PRIVATE). After allocation stored the initial pointer in it's required place.
Matrix multiplication:
Used 3 for loops as per the given pseudocode. Optimized code for variant ikj and variant kij (Loading values from memory is optimized here)
TSC frequency is **2419.200 MHz**

variant	128	256	512	1024	2048
ijk	0.0267868	0.2412948	3.1786687	64.3334112	807.1689615
ikj	0.0265996	0.2163085	1.9914481	16.1969530	144.5876207
jik	0.0260722	0.2745542	3.6488924	63.9209628	763.9057028
jki	0.0361502	0.3691887	6.817219	100.5419893	1397.9444482
kij	0.0270438	0.2398788	2.1150457	16.7841103	155.2048170
kji	0.0353579	0.3593108	6.7898921	77.9555313	1421.3277928

Table 2: Time taken by each variant in seconds

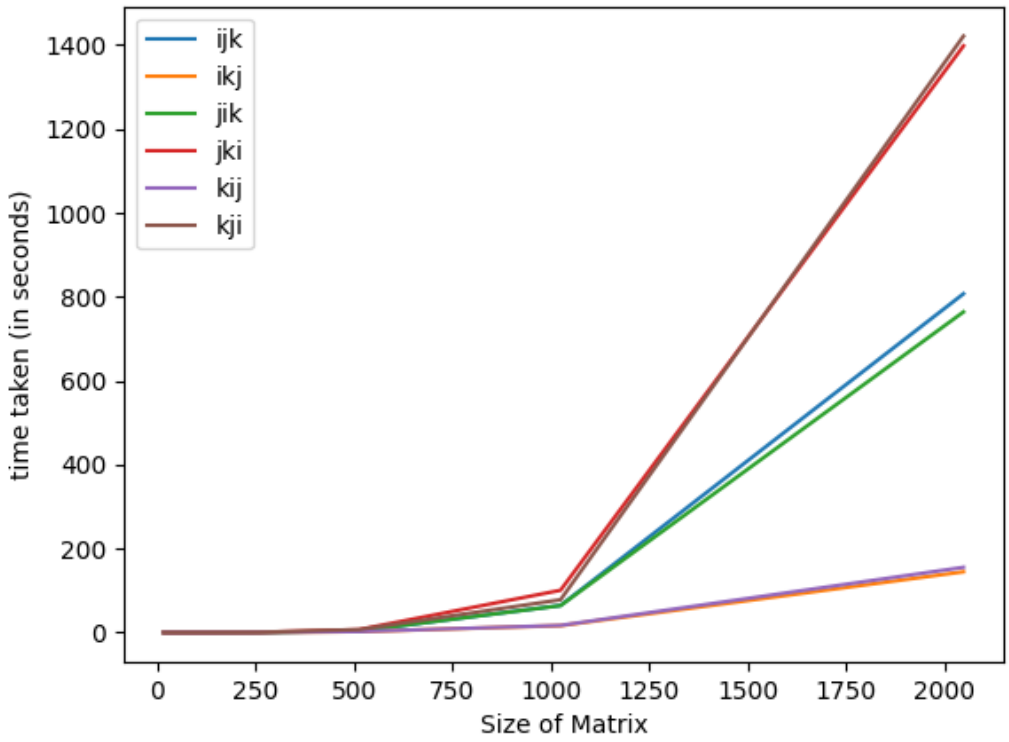

Figure 1: Plot of time (in seconds) vs size of matrix for each variant

variant	128	256	512	1024	2048
ijk	5569610	55339964	756824098	15516737998	195076234423
ikj	5642185	49335792	469188669	3867941056	34765418505
jik	5517203	63410238	870616594	15410563961	184552834416
jki	7908415	86145079	1636428034	24268044325	337968402103
kij	5681719	54832426	497185696	4005791969	37345707337
kji	7726901	83759888	1628490157	18795656712	343656284862

Table 3: Number of cycles for each variant

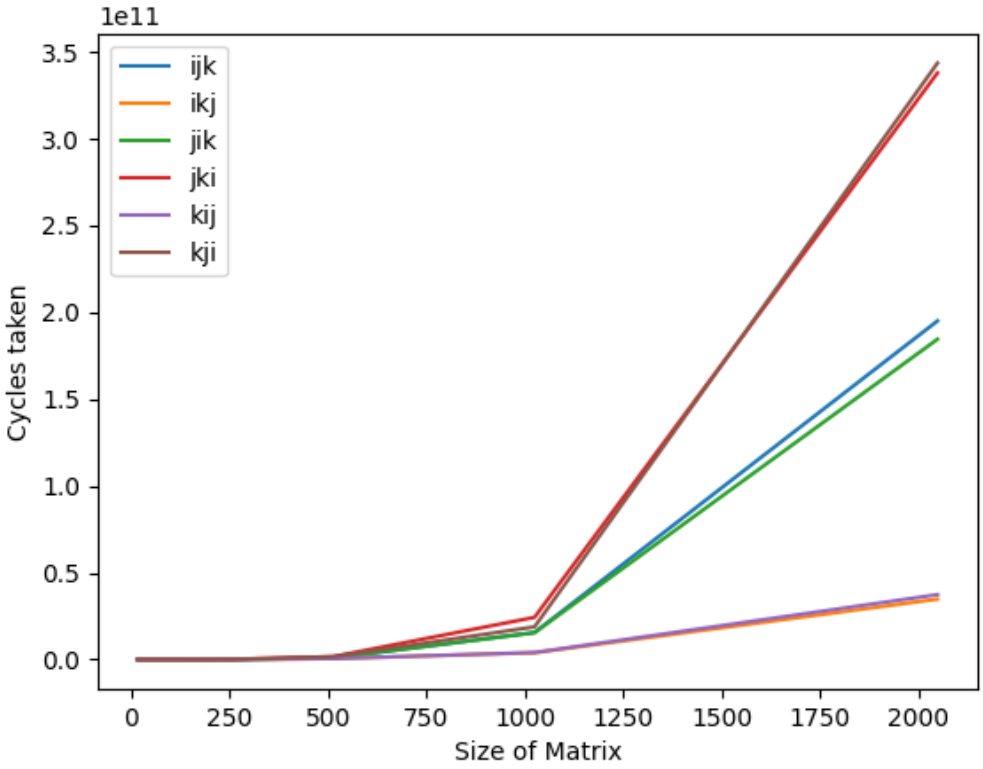

Figure 2: Plot of number of cycles vs size of matrix for each variant

5 References

1. mmap memory allocation in X86
2. c++ code for inverse modulo in log(m) time
3. Iterative version to make partitions in merge sort
4. Merge function in O(1) space