Lab 3: Part 1

Ved Danait

October 2023

We used the commands:

```
objdump -D -M intel program1
objdump -D -M intel program2
```

This gave us the x86 assembly code for both the programs. Following is the analysis of each of these codes.

1 Program 1

We claim, that the program checks whether or not the sequence is an AP and prove this by analysing the assembly code.

We start off by allocating 48 bytes of memory on the stack. Now we load the memory location corresponding to *”Enter three or more numbers (Terminate with CTRL + D):”* into $edi and then call the print function. Now we initialize some useful variables: [$rbp-0x4] is a variable i, which we claim takes value either 0 or 1, [$rbp-0xc] is our counter c, which stores the number of integers input by the user both being 4 byte variables. [$rbp-0xd] stores a flag which is initially set to 1 and which is changed to 0 if the sequence is not an AP. Now we take an unconditional jump to PC = 4011df.

Here, we load the value of i into $eax, perform a sign extension, and store the now 8 byte value of 4i into $edx using the lae instruction. Now, the address [$rbp-0x1c] is stored into $rax, and the value of $rdx is added. Now we copy this value of $rax which stores the memory location [$rbp-0x1c + 4i] into $rsi. Now we load the corresponding memory location for the *scanf* function and call it (note that before doing so we reset $eax to 0x0). Now we compare $eax with 0x1, which is basically true if we take in a valid input ($eax is 0x0 if we input ctrl+D). If $eax is 0x0, then we compare the value of counter(c) with 2. If it is greater we jump to PC = 401225 where the flag is compared to 0, if equal we print *”NO”* and if not then we print *”NO”* and then the program terminates. Otherwise, if our counter(c) is less than or equal to 2, we print the error message, *”You have not entered enough numbers, try again”*. Now, we analyse the case where the user enters a valid input i.e. $eax = 0x1.

We jump to PC = 401178, where we increment the counter(c) by 1. We compare the counter with 0x1. If it is less than or equal to basically when there is only 1 element we need to take care of this edge case. We jump to PC = 4011c7. Here we encounter this chunk of code :

```
4011c7: mov    eax,DWORD PTR [rbp-0x4]
4011ca: lea    edx,[rax+0x1]
4011cd: mov    eax,edx
4011cf: sar    eax,0x1f
4011d2: shr    eax,0x1f
4011d5: add    edx,eax
4011d7: and    edx,0x1
4011da: sub    edx,eax
4011dc: mov    DWORD PTR [rbp-0x4],edx
4011df: mov    eax,DWORD PTR [rbp-0x4]
```

We can show that this code effectively just alternates between 0 and 1, i.e. if input is 0 it outputs 1 and if input is 1 it outputs 0. Observe, firstly that i is non negative. Then we copy i into $eax and load $\$eax + 1$ into $edx and then copy this value back into $eax. Leaving us with $i + 1$ in both $eax and $edx. Now we apply sar instruction on $eax with shift of 0x1f(31). since, $eax contained a positive value, it effectively becomes cleared out to 0. Now, the subsequent shr also plays no effect and $eax is still 0. add edx,eax also does nothing to the value of edx and subsequently taking and with 0x1 is just like taking the number modulo 2. So ultimately, we are left with $i + 1 modulo 2$ in $edx (basically the not operation), which we will copy into [$rbp-0x4]. The initial value of i is 0 and since this chunk is just a logical not, the value of i keep fluctuating between 0 and 1. We use this value of i and this similar mechanism to store the most recent values input by the user in the memory locations [$rbp+4i-0x1c] where i takes the value 0 or 1. Now, coming back to the PC = 4011e4, we set $rsi to [$rbp+4i-0x1c], and call *scanf*, taking in the second value. Now, we go back to our main iteration and analyse the core logic, with all the edge cases out of the way.

mov eax,DWORD PTR [$rbp-0x8], basically copied the current difference for the previous iteration into eax and mov DWORD PTR [rbp-0x14],eax copies it subsequently into [$rbp-0x14]. Hence, in each iteration we have the previous difference stored in [$rbp-0x14] and the current difference stored in [$rbp-0x8]. Now, we copy i into $eax and also the value stored in [$rbp+4i-0x1c] into $ecx (This is our most recently input number). Now we move i into $eax and apply the same chunk of code we explained above to get its logical not. We then perform move eax,DWORD PTR [rbp+rax*4-0x1c], which basically copies the second most recently input number into $eax. Now we take the difference of ecx and eax, which is just our common difference and copy this value to $edx. Now we store this value in $edx into [$rbp-0x8] which basically stores our current difference. We again compare the counter with 2, if it is less than or equal to we don't do anything and proceed to input the next number since no comparison can be made with a previous common difference. More importantly, when our counter is greater than 2, we load the value of the previous common difference stored in [$rbp-0x14] into $eax and compare this value with [$rbp-0x8]. If these values are equal that means it is a valid AP upto this point and we can move onto the next iteration at PC = 4011c7 and input the new value. Otherwise, we set the flag to 0 and then fall through into the new input section. This will result in the final output being *"NO"* since now the flag can never revert back to 1.

2 Program 2

2.1 Explanation of main

The first three lines of the main function basically set the stack frame and allocate 16 bytes of memory on the stack. Now, the lines print the statement *"Enter a non-negative integer : "*. Now, the subsequent 2 lines load the address [$rbp - 0x8] into the register $rax (This is where the value input by the user will be stored), and then the mov instruction copies the content of $rax into $rsi. Now, the program runs the *scanf* command, and the value entered by the user is stored into the address pointed to by the $rsi register. Then, the program copies this value of "n" into the $rax register and subsequently copies it into the $rsi register which generally stores the first argument for function calls. Now, the function executes and stores the value into the $rax register, the working of which we analyze in the next subsection. The output value in $rax is then copied into the $rsi register and we then proceed to the print statement corresponding to the address in $edi, 0x40202c which prints the statement *"Output : %d"*. And %d gets its value from the $rsi register. After this the program terminates. Now let us analyse the function <func> called at PC = 4011db.

2.2 Explanation of func

As always the first 3 lines setup the stack frame and this time we allocate 40 bytes of memory on the stack. mov QWORD PTR [rbp-0x28],rdi copies the 8 byte value from $rdi into the memory location [$rbp-0x28] on the stack. This stores our value of **n**.

We first compare the value of n with 0x0, this acts as the base case. If n = 0, we load the value 0x1 into the $eax register and jump to 4011a1 which executes mov rbx, QWORD PTR [rbp-0x8] and then leaves the function, returning the value stored in $eax which is 1.

Now if n is not equal to 1, we jump to PC = 401151. This is where the main logic of the program is stored. We first initialize the stack location [$rbp-0x18] to 0x0, occupying 8 bytes on the stack : This is where we store our final result.

Now we initialize [$rbp-0x20] to 1, again occupying 8 bytes, we call this variable **i**. This completes our setup.

Now, we take an unconditional jump to PC = 401193, where we copy the value of i into $rax register. Now, we compare value stored in [$rbp - 0x28] which is n with $rax which is i. If $n \geq i$, we jump to PC = 401163, continuing to the next iteration. Note that if this is false then we are done, we copy the value stored in [$rbp-0x18] into $rax register (which is our final result that we want to return) and [$rbp-0x8] in $rbx and then exit the function returning the value in $rax.

Now, the iteration logic at PC = 401163 : We copy the value of i into $rax, decrement value in $rax(note we initially stored 1 and the program wants the values to start from 0, so this decrement achieves exactly that), and we store it into $rdi. Now we execute recursive call, yielding the value f(i) in the register $rax, which we store into $rbx temporarily. Now the next three lines compute the value of $n - i$ in the $rax register, copy this value into $rdi which is the argument for function call, and make the recursive call yielding the value f(n-i). Now, imul rax,rbx multiplies f(n-i) with f(i) and stores it in the $rax register. (Recall that we had stored f(i) in $rbx). Now we add $rax to value in [$rbp-0x18], which is our final result. We, then increment i, on the stack, copy the value to $rax and perform our loop's main comparison checking whether $n \geq i$. If this is true we continue to iterate, else as we saw before, we copy our result stored in [rbp-0x18] to $rax and exit the function.
Hence, we effectively get the recurrence relation,

$$C(n) = \sum_{i=0}^{n-1} C(i)C(n-i) \quad C(0) = 1$$

Which means the function effectively outputs the n^{th} **Catalan Number**