Lab 3
Part1

Himanshu Gangwal

October 6, 2023

Abstract

Forgive me, the latex won't look as appealing as it should be due to time constrains and also due to the fact that I am incredibly lazy. . .

```
$ objdump -d <executable> -M intel
```

was used to generate the assembly code.

Contents

1 program2

Let's start, I'll directly jump onto main function and let's see what you are trying to do :

$$154011b9 : call 401030 < printf@plt >$$

this is the initial printed message after this, memory equivalent to 8 bytes is allocated on the stack and then $scanf$ is called which places the result of call into the address pointed by rsi,

```
155   4011be:   48 8d 45 f8          lea   rax,[rbp-0x8]
156   4011c2:   48 89 c6             mov   rsi,rax
157   4011c5:   bf 27 20 40 00       mov   edi,0x402027
158   4011ca:   b8 00 00 00 00       mov   eax,0x0
159   4011cf:   e8 6c fe ff ff       call  401040 <__isoc99_scanf@plt>
```

now we move further to calling the function, which seems to place the computed value in the register rax and takes rdi as an input,

```
161   4011d8:   48 89 c7             mov   rdi,rax
162   4011db:   e8 56 ff ff ff       call  401136 <func>
163   4011e0:   48 89 c6             mov   rsi,rax
164   4011e3:   bf 2c 20 40 00       mov   edi,0x40202c
165   4011e8:   b8 00 00 00 00       mov   eax,0x0
166   4011ed:   e8 3e fe ff ff       call  401030 <printf@plt>
```

jumping over to program, firstly we are putting the argument say x at $[rbp - 0x28]$, now on the next line we compare and if the comparison results true, i.e. $x = 0$ we move 1 to the return register eax and return,

```
118   40113f:   48 89 7d d8          mov   QWORD PTR [rbp-0x28],rdi
119   401143:   48 83 7d d8 00       cmp   QWORD PTR [rbp-0x28],0x0
120   401148:   75 07                jne   401151 <func+0x1b>
121   40114a:   b8 01 00 00 00       mov   eax,0x1
122   40114f:   eb 50                jmp   4011a1 <func+0x6b>
```

which essentially is return.

Perhaps I would also like to share a cool discovery of mine, or as most of the people call it, obviousness, the compiler pre-calculates the number of quads that have to be used or say the summation of the sizes of all the

local variables involved and allocates that amount of space onto the stack, then fills the stack rather bottom up so that the current function iteration never messes with the locals of the previous or next iter, but however this optimization is non-trivial at least for me, but I think it is quite central in writing machine code, as there are only so much operations that we can do using 32 registers. This should imply the local variables have a cost equivalent to 0x0 + (0x8)*3, i.e. 3 quads + one argument, as we'll see the *rdi* register cannot store the argument throughout the process due to nested calls.

Now first consider the portion,

```
140   401193:       48 8b 45 e0          mov    rax,QWORD PTR [rbp-0x20]
141   401197:       48 39 45 d8          cmp    QWORD PTR [rbp-0x28],rax
142   40119b:       73 c6                jae    401163 <func+0x2d>
```

here, from what I read just now points that *jae* only jumps if the comparison above or the carry flag is set to 0, i.e. if the comparison is unsuccessful, it'll jump.

Now let's continue reading from the *return*1 or the base case branching, we here allocate two new local variables lets name them a and b corresponding to their quad indexing $rbp - 0x18$ and $rbp - 0x20$ on the local variable allocation stack, which I explained in the previous point. We'll first dry run it, initially the values are $a = 0b = 1$, arghhh.., at this point I really wish I could've been able to draw pointer like I do in chess, anyways, refer to the portion below, at 401161 we jump to 401193 and as it is clear from the above statement the code must jump, at least once as $x > 0$.

Now, in 401163 to 40116e the program essentially calculates f(b-1) and puts it in *rax* and for the next four lines it calculates f(x-b) which is f(x-1) and stores in *rbx* here, then the program correspondingly updates the value of $a+ = rax*rbx$ and increments b. Now notice, the recursion will stop when the result of comparison (x, b) is 1, *jae* won't jump and thus the recursion ends.

```
127   401161:   eb 30              jmp    401193 <func+0x5d>
128   401163:   48 8b 45 e0        mov    rax,QWORD PTR [rbp-0x20]
129   401167:   48 83 e8 01        sub    rax,0x1
130   40116b:   48 89 c7           mov    rdi,rax
131   40116e:   e8 c3 ff ff ff     call   401136 <func>
132   401173:   48 89 c3           mov    rbx,rax
133   401176:   48 8b 45 d8        mov    rax,QWORD PTR [rbp-0x28]
134   40117a:   48 2b 45 e0        sub    rax,QWORD PTR [rbp-0x20]
135   40117e:   48 89 c7           mov    rdi,rax
136   401181:   e8 b0 ff ff ff     call   401136 <func>
137   401186:   48 0f af c3        imul   rax,rbx
138   40118a:   48 01 45 e8        add    QWORD PTR [rbp-0x18],rax
139   40118e:   48 83 45 e0 01     add    QWORD PTR [rbp-0x20],0x1
140   401193:   48 8b 45 e0        mov    rax,QWORD PTR [rbp-0x20]
141   401197:   48 39 45 d8        cmp    QWORD PTR [rbp-0x28],rax
142   40119b:   73 c6              jae    401163 <func+0x2d>
```

So, finally the expression is,

$$f(x) = f(0)f(x-1) + f(1)f(x-2) + ... + f(x-1)f(0); f(0) = 1$$

now continuing with the usual order in which i conduct my life as writing the closed form is optional, ill choose not to solve for it, or write it in this case, I'll just go and try solving Q2!!.

2 program1

Ufff... , the code seems really lengthy, what i'll do is basically dry run throughout the code starting from the initialization portion that is allocating space on the stack and we'll see what is going on through out the operations,

```
401146:     55                          push   rbp
401147:     48 89 e5                    mov    rbp,rsp
40114a:     48 83 ec 30                 sub    rsp,0x30
40114e:     89 7d dc                    mov    DWORD PTR [rbp-0x24],edi
401151:     48 89 75 d0                 mov    QWORD PTR [rbp-0x30],rsi

401155:     bf 08 20 40 00              mov    edi,0x402008//initial
                                              //print call
40115a:     b8 00 00 00 00              mov    eax,0x0
40115f:     e8 dc fe ff ff              call   401040 <printf@plt>

401164:     c7 45 fc 00 00 00 00        mov    DWORD PTR [rbp-0x4],0x0
40116b:     c7 45 f4 00 00 00 00        mov    DWORD PTR [rbp-0xc],0x0
401172:     c6 45 f3 01                 mov    BYTE PTR [rbp-0xd],0x1 !!
401176:     eb 67                       jmp    4011df <main+0x99>
```

do note the initialization of the character at $rbp - 0xd$, which is actually the yes/ no flag for the program. Then let's jump to the address $4011df$, which is pointed out in the last instruction,

```
4011df:     8b 45 fc                    mov    eax,DWORD PTR [rbp-0x4]
4011e2:     48 98                       cdqe
4011e4:     48 8d 14 85 00 00 00        lea    rdx,[rax*4+0x0]
4011eb:     00
4011ec:     48 8d 45 e4                 lea    rax,[rbp-0x1c]
4011f0:     48 01 d0                    add    rax,rdx
4011f3:     48 89 c6                    mov    rsi,rax

4011f6:     bf 40 20 40 00              mov    edi,0x402040
4011fb:     b8 00 00 00 00              mov    eax,0x0
401200:     e8 4b fe ff ff              call   401050 <__isoc99_scanf@plt>

401205:     83 f8 01                    cmp    eax,0x1
```

```
401208:        0f 84 6a ff ff ff          je     401178 <main+0x32> //first
                                                                    //jump
```

A rookie error that i was making for about an hour was thinking the last jump occurs when the value entered is actually, 1, after looking the $scanf$ function, I git to understand the value of eax rather represents the bool if there's an input or not, now we'll require a bit of bookkeeping for our locals, $0x4 = 0$, $0xc = 0$ and $0xd = 1$. Then, the function reads a value and jumps back, if there's no input the instructions in the next line break to the fact, if the number of values given is greater than 3 or not, if there are less than 3 values the program essentially returns, otherwise it gives and output based on the value of out flag $0xd$.

let' now go back to the address 401178, now $0xc = 1$, now 401180 initiates a jump to $4011c7$,

```
401178:        83 45 f4 01                add    DWORD PTR [rbp-0xc],0x1
40117c:        83 7d f4 01                cmp    DWORD PTR [rbp-0xc],0x1
401180:        7e 45                      jle    4011c7 <main+0x81>
```

this portion is a bit tricky, remember that $0x4 = 0$

```
4011c7:        8b 45 fc                   mov    eax,DWORD PTR [rbp-0x4]
4011ca:        8d 50 01                   lea    edx,[rax+0x1]
4011cd:        89 d0                      mov    eax,edx
4011cf:        c1 f8 1f                   sar    eax,0x1f
4011d2:        c1 e8 1f                   shr    eax,0x1f
4011d5:        01 c2                      add    edx,eax
4011d7:        83 e2 01                   and    edx,0x1
4011da:        29 c2                      sub    edx,eax
4011dc:        89 55 fc                   mov    DWORD PTR [rbp-0x4],edx
```

after going through the code the value will change to $0x4 = 1$, this piece of code is just a complicated way of writing a not gate, because at the end the bit value will only decide the position at which the next input is to be stored, here the value in $0x4$, essentially acts as an toggle between two positions. Then on the code moves again to $4011df$, which now puts the new value in a different location(for the first iter it is $0x08$ which is $0xc$-$rax * 4$, and rax alternates between 1 and 0), pointed by the above logic. At anytime, the program has to essentially use two values for further calculation. Now we move back to 401178, and re-run with $0x4 = 1$, the value in $0xc$ is now 2, so no jump at the address 401180, and we move onto 401182,

```
401182:        8b 45 f8                    mov    eax,DWORD PTR [rbp-0x8]
401185:        89 45 ec                    mov    DWORD PTR [rbp-0x14],eax
401188:        8b 45 fc                    mov    eax,DWORD PTR [rbp-0x4]
40118b:        48 98                       cdqe
40118d:        8b 4c 85 e4         mov    ecx,DWORD PTR [rbp+rax*4-0x1c]
401191:        8b 45 fc                    mov    eax,DWORD PTR [rbp-0x4]
401194:        8d 50 01                    lea    edx,[rax+0x1]
401197:        89 d0                       mov    eax,edx
401199:        c1 f8 1f                    sar    eax,0x1f
40119c:        c1 e8 1f                    shr    eax,0x1f
40119f:        01 c2                       add    edx,eax
4011a1:        83 e2 01                    and    edx,0x1
4011a4:        29 c2                       sub    edx,eax
4011a6:        89 d0                       mov    eax,edx
4011a8:        48 98                       cdqe
4011aa:        8b 44 85 e4         mov    eax,DWORD PTR [rbp+rax*4-0x1c]
4011ae:        29 c1                       sub    ecx,eax
4011b0:        89 ca                       mov    edx,ecx
4011b2:        89 55 f8                    mov    DWORD PTR [rbp-0x8],edx
4011b5:        83 7d f4 02                 cmp    DWORD PTR [rbp-0xc],0x2
4011b9:        7e 0c                       jle    4011c7 <main+0x81>
4011bb:        8b 45 ec                    mov    eax,DWORD PTR [rbp-0x14]
4011be:        3b 45 f8                    cmp    eax,DWORD PTR [rbp-0x8]
4011c1:        74 04                       je     4011c7 <main+0x81>
```

now at line 40118b, the value of rax becomes 1, $ecx = 0x8$ and $0x14 = 0x8 = enteredVal$, now moving on to the inner logic see that it calculates the value for the rax that has to be used for 4011aa and thus, as explained in the alternating above the shift and and operations, again not the value of rax used, such that for $ecx = 0xc$, which was the previous value. Now furthermore, at the address 4011$b2$, ecx, stores the difference between the two values, also the reason what the code look so skewed, when it tries to alternate the values at multiple portions in the code is, the compiler , notes the effect that a toggle has and minimizes the local variable stack space in regarding to the final observed effect that a swap operation essentially has, so rather than swapping the new and previous values, it toggles the position at which the newer value has to be put.

```
4011aa:     8b 44 85 e4          mov    eax,DWORD PTR [rbp+rax*4-0x1c]
4011ae:     29 c1                  sub    ecx,eax
4011b0:     89 ca                  mov    edx,ecx
4011b2:     89 55 f8               mov    DWORD PTR [rbp-0x8],edx
4011b5:     83 7d f4 02            cmp    DWORD PTR [rbp-0xc],0x2
4011b9:     7e 0c                  jle    4011c7 <main+0x81>
4011bb:     8b 45 ec               mov    eax,DWORD PTR [rbp-0x14]
4011be:     3b 45 f8               cmp    eax,DWORD PTR [rbp-0x8]
4011c1:     74 04                  je     4011c7 <main+0x81>
4011c3:     c6 45 f3 00          mov    BYTE PTR [rbp-0xd],0x0 !!
```

Now we are towards the end of the program, as you can see the program first check is a comparison valid, i.e. are there enough elements to compare, and when there are, it compares the values of previous and the calculated difference, if the values are same the program jumps to $4011c7$ and if not it sets the value of flag $0xd = 0$, which you can see will not change throughout the complete control flow, and at the end when there are no input elements, the program proceeds through the $scanf$ call, now the number of elements is also greater than 2 hence, it resorts to put output corresponding to the flag value.

Final Result The output depends on the flag, as deduced from above, i.e. are the values equally spaced, and when we put provide it with such a sequence it returns YES and otherwise NO. Hence, the program check and gives a positive output if the given sequence is an arithmetic progression.