

Question 1

```

                                add r0, #4, r9                ; initialise inp_int

max:    add r26, r0, r1        ; use r1 for local variable v
        sub r27, r1, r0, {C}   ; b>v
        jle max0
        xor r0, r0, r0        ; nop in delay slot
        add r27, r0, r1

max0:   sub r28, r1, r0, {C}    ; c>v
        jle max1
        xor r0, r0, r0
        add r28, r0, r1        ; v=c

min1:   ret r31, 0
        xor r0, r0, r0

max5:   add r9, r0, r10         ; set up 1st param = inp_int
        add r26, r0, r11
        callr r15, max         ; call max, return address in r15
        add r27, r0, r12       ; set up 3rd param during delay
        add r1, r0, r10        ; set up 1st param
        add r28, r0, r11       ; set up 2nd param
        callr r15, max         ; call max, return address in r15
        add r29, r0, r12       ; set up 3rd param during delay
        ret r31, 0             ; return ( return address in r31 )
        xor r0, r0, r0
```

Question 2

```
fun:      sub r0, r27, r0, {C}          ; b == 0
          jne fun1
          xor r0, r0, r0
          add r0, r0, r1
          ret r31, 0                    ; return 0
          xor r0, r0, r0

fun1:     add r27, r0, r10               ; set up 1st param b
          callr r15, mod                 ; b % 2
          add r0, #2, r11               ; set up 2nd param 2 during delay slot
          sub r0, r1, r0, {C}           ; r1 == 0
          jne fun2
          xor r0, r0, r0
          add r27, r0, r10               ; set up 1st param b
          callr r15, div                 ; set up 2nd param 2 during delay slot
          add r0, #2, r11               ; set up 1st param a+a
          add r26, r26, r10
          callr r15, fun
          add r1, r0, r11               ; set up 2nd param b / 2 during delay slot
          ret r31, 0
          xor r0, r0, r0

fun2:     add r27, r0, r10               ; set up 1st param b
          callr r15, div                 ; set up 2nd param 2 during delay slot
          add r0, #2, r11               ; set up 1st param a+a
          add r26, r26, r10
          callr r15, fun
          add r1, r0, r11               ; set up 2nd param b / 2 during delay slot
          add r1, r26, r1               ; fun(a+a, b/2) + a
          ret r31, 0
          xor r0, r0, r0
```

Question 3

I compiled a C++ file that implemented the compute_pascal function. I wrote a function pascalwithchecks(int row, int position) that basically ran the compute_pascal function but it had code at entry and exit to keep track of all the variables that we are after, such as depth, function calls, overflows and underflows. File = t3.cpp

At entry, the overflowCheck() function is called.

```
void overflowCheck() {
    calls++;
    depth++;
    if (depth > maxDepth) {
        maxDepth = depth;
    }
    if (windowsInUse == nwindows) {
        overflows++;
    }
    else {
        windowsInUse++;
    }
}
```

We can see that there is a variable that is updated every time the function is called. Also we keep track of the maxDepth that is reached. The overflows variable is only incremented when all windows are in use. When we wish to run the code such that overflow occurs when one register window is unused, we can modify this code to do so. Instead of (windowsInUse == nwindows) we would have (windowsInUse == nwindows-1).

At exit, the underflowCheck() function is called.

```
void underflowCheck() {
    depth--;
    if (windowsInUse == 2) {
        underflows++;
    }
    else {
        windowsInUse--;
    }
}
```

Here we see that underflow only occurs when there are only two register windows in use.

6 Register Windows

Computing `compute_pascal(30,20)` with 6 register windows resulted in 40060019 function calls.

The maximum register window depth was 29.

The total number of overflows that occurred was 7051109.

The total number of underflows that occurred was 7051109.

8 Register Windows

Computing `compute_pascal(30,20)` with 8 register windows resulted in 40060019 function calls.

The maximum register window depth was 29.

The total number of overflows that occurred was 2840177.

The total number of underflows that occurred was 2840177.

16 Register Windows

Computing `compute_pascal(30,20)` with 16 register windows resulted in 40060019 function calls.

The maximum register window depth was 29.

The total number of overflows that occurred was 30826.

The total number of underflows that occurred was 30826.

Modifying code such that overflow only occurs when one register iwindow is unused

After modifying the code to do this, the number of function calls and max depth stayed the same.

When ran with 6 register windows, the number of overflows was 10656359 and underflows was 10656359.

When ran with 8 register windows, the number of overflows was 4527434 and underflows was 4527434.

When ran with 16 register windows, the number of overflows was 58650 and underflows was 58650.

We can see here that modifying the code so that overflow only occurred when one register window was unused significantly decreased the amount of overflows when 6 register windows were available. It increased the amount of overflows that occurred when 8 and 26 register windows were available however.

I provided screenshots of my console output so that you can see proof of the results I got.

First set of results

Microsoft Visual Studio Debug Console

```
compute_pascal(30,20) -> number of windows: 6  
Number of Calls: 40060019  
Maximum Register Window Depth: 29  
Overflows: 7051109  
Underflows: 7051109  
  
compute_pascal(30,20) -> number of windows: 8  
Number of Calls: 40060019  
Maximum Register Window Depth: 29  
Overflows: 2840177  
Underflows: 2840177  
  
compute_pascal(30,20) -> number of windows: 16  
Number of Calls: 40060019  
Maximum Register Window Depth: 29  
Overflows: 30826  
Underflows: 30826
```

Second set of results with code modified for different overflow condition

Microsoft Visual Studio Debug Console

```
compute_pascal(30,20) -> number of windows: 6  
Number of Calls: 40060019  
Maximum Register Window Depth: 29  
Overflows: 10656359  
Underflows: 10656359  
  
compute_pascal(30,20) -> number of windows: 8  
Number of Calls: 40060019  
Maximum Register Window Depth: 29  
Overflows: 4527434  
Underflows: 4527434  
  
compute_pascal(30,20) -> number of windows: 16  
Number of Calls: 40060019  
Maximum Register Window Depth: 29  
Overflows: 58650  
Underflows: 58650
```

Question 4

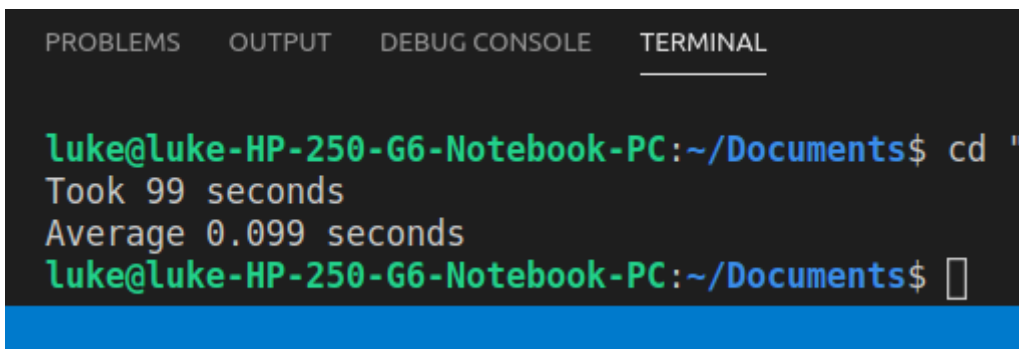
To time the code I created another C++ file with some code that timed how long it took to execute the code for `compute_pascal(30,20)`.

My personal approach was to run this code a large number of times (I ran it 1000 times). I believed that this would be the most accurate way to measure as running it a single time would lack precision in how well the cpp clock implimentation would be on my computer's cpu. Once I had the overall time for 1000 cycles, I computed the average time. Some accuracy might be lost due to incrementing the count in the while loop and the variable comparison, but I still believe that this is a very accurate implementation of timing the function. File = `timed_pascal.cpp`

My computer: Intel Core i7-7500U processor, 2700Mhz, 2 core.

Average time: 0.099 seconds.

Console:



```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

luke@luke-HP-250-G6-Notebook-PC:~/Documents$ cd "
Took 99 seconds
Average 0.099 seconds
luke@luke-HP-250-G6-Notebook-PC:~/Documents$
```

This function took longer to execute than I expected, probably due to the large amount of function calls required.