MID-TERM REPORT

Type A – Integer Number:

14. Enter three positive integers A, B, and C from the keyboard. Check if A, B, and C are the sides of a triangle. If they form a triangle, check if it is a right triangle.

- Understanding the Problem:
  + Read 3 numbers from user input
  + Apply the Triangle Inequality to find out whether the numbers make a triangle
  + If they make a triangle, apply Pythagoras's Theorem to find out whether it's a right triangle
  + Print the result

- Implementation:
  First, the program gets three side lengths from the user. It then checks if they can form a triangle by ensuring that the sum of any two sides is greater than the third. If this rule is not met, it concludes the sides do not form a triangle. If they do form a triangle, the program then checks if it is a right-angled triangle by seeing if the square of one side equals the sum of the squares of the other two. If this Pythagorean condition is true, it is a right triangle; otherwise, it is not.

- Source code:

```
.data
A: .asciz "Enter side A: "
B: .asciz "Enter side B: "
C: .asciz "Enter side C: "
isTriangle: .asciz "A, B, C form a triangle\n"
isNotTriangle: .asciz "A, B, C don't form a triangle.\n"
isRight: .asciz "It's a right triangle.\n"
isNotRight: .asciz "It's not a right triangle.\n"
errorMsg: .asciz "Not a number. Try again: "
buffer: .space 20

.text
main:
        # prompt and read integer A
        li a7, 4
```

```
la a0, A
ecall

jal ra, check
mv s0, a0 # s0 = A

# prompt and read integer B
li a7, 4
la a0, B
ecall

jal ra, check
mv s1, a0 # s1 = B

# prompt and read integer C
li a7, 4
la a0, C
ecall

jal ra, check
mv s2, a0 # s2 = C

# check triangle inequality: A + B > C, A + C > B, B + C > A
add t0, s0, s1  # t0 = A + B
ble t0, s2, not_a_triangle  # if A + B <= C, not a triangle

add t0, s0, s2  # t0 = A + C
ble t0, s1, not_a_triangle  # if A + C <= B, not a triangle

add t0, s1, s2  # t0 = B + C
ble t0, s0, not_a_triangle  # if B + C <= A, not a triangle

# if all checks pass, it is a triangle
li a7, 4
la a0, isTriangle
ecall

# Check for a right triangle
```

```
        mul s3, s0, s0 # s3 = A*A
        mul s4, s1, s1 # s4 = B*B
        mul s5, s2, s2 # s5 = C*C

        add t0, s3, s4  # t0 = A*A + B*B
        beq t0, s5, is_a_right_triangle # if A*A + B*B = C*C

        add t0, s3, s5  # t0 = A*A + C*C
        beq t0, s4, is_a_right_triangle # if A*A + C*C = B*B

        add t0, s4, s5  # t0 = B*B + C*C
        beq t0, s3, is_a_right_triangle # if B*B + C*C = A*A

        # if none of the right triangle conditions are met
        li a7, 4
        la a0, isNotRight
        ecall
        j exit

is_a_right_triangle:
        li a7, 4
        la a0, isRight
        ecall
        j exit

not_a_triangle:
        li a7, 4
        la a0, isNotTriangle
        ecall

exit:
        li a7, 10
        ecall

check:
        # read string into buffer
        li a7, 8
        la a0, buffer
```

```
        li a1, 20
        ecall

        la t1, buffer # t1 = Pointer to current char
        li t2, 0 # t2 = result
        li t3, 0 # t3 = sign flag (0 = pos, 1 = neg)

        lb t4, 0(t1) # load first byte

        # check for negative sign
        li t5, 45                   # t5 = '-'
        bne t4, t5, parse_loop
        li t3, 1                    # set sign flag to negative
        addi t1, t1, 1    # move pointer forward
        lb t4, 0(t1)                # load next char

parse_loop:
        # check for end of string (newline or null)
        li t5, 10 # t5 = '\n'
        beq t4, t5, end_parse
        beqz t4, end_parse  # Null terminator

        # digit must be from '0' to '9'
        li t5, 48
        blt t4, t5, invalid_input
        li t5, 57
        bgt t4, t5, invalid_input

        # convert string to integer
        # result = (result * 10) + (char - 48)
        addi t4, t4, -48          # convert char to integer digit
        li t5, 10
        mul t2, t2, t5    # shift current result left (decimal)
        add t2, t2, t4    # add new digit

        # move to next char
        addi t1, t1, 1
        lb t4, 0(t1)
```

```
        j parse_loop

invalid_input:
        # print error message
        li a7, 4
        la a0, errorMsg
        ecall
        # jump back to start of check to read again
        j check

end_parse:
        # apply negative sign if needed
        beqz t3, return_result
        sub t2, zero, t2        # result = 0 - result

return_result:
        mv a0, t2               # move result to a0 for return
        ret
```

- Example input/output:

```
Enter side A: 3
Enter side B: 4
Enter side C: 5
A, B, C form a triangle
It's a right triangle.
```

```
Enter side A: 3
Enter side B: 5
Enter side C: 6
A, B, C form a triangle
It's not a right triangle.
```

```
Enter side A: 1
Enter side B: 2
Enter side C: 3
A, B, C don't form a triangle.
```

Type B - Arrays:

3. Enter an array of integers from the keyboard. Print out the pair of adjacent elements with the largest product. For instance, enter the array [3, 6, -2, -5, 7, 3], the adjacent pair with the largest product is 7 and 3.

- Understanding the Problem:
  - + Reading the first number to start.
  - + Looping through the rest of the numbers one by one.
  - + In each step, it multiplies the **current number** by the **previous number** to get an "adjacent product."
  - + It compares this new product to the largest product it has found so far.
  - + If the new product is larger, it **saves that product** and the **two numbers** that created it.
  - + After checking all numbers, it prints the saved pair.

- Implementation:

  The program accepts a user-input array of integers by first asking the user for the array size, then taking input for each element one at a time using a loop. Inside the loop, it multiplies the current number with the previously read number, compares the product to the largest one found so far, and updates the maximum product and the corresponding pair if a new high is found. After

checking all pairs, it prints the two numbers that formed the largest adjacent product.

- Source code:

```
.data
        msg1: .asciz "Input N: "
        msg2: .asciz "Output: "
        msg_error: .asciz "Error (requires at least 2 elements)\n"
        endl: .asciz "\n"
        space: .asciz " "
        msg_error2: .asciz "Not a number. Try again: "
        buffer:  .space 32   # Buffer to store input text

.text
# Registers:
# t0 = number of elements
# t1 = largest product found so far
# t2 = first element of the pair with the largest product
# t3 = second element of the pair with the largest product
# t4 = previous element read
# t5 = current element read
# t6 = loop counter

main:
        # prompt and read for the number of elements
        li a7, 4
        la a0, msg1
        ecall

        jal ra, check
        mv t0, a0

        # check if there are at least 2 elements for a pair
        li a7, 2
        blt t0, a7, error

        # initialize largest product as INT_MIN
```

```
        li t1, -2147483648

        # read the first element
        jal ra, check
        mv t4, a0 # t4 holds the previous element

        # initialize loop counter to 1
        li t6, 1

loop:
        bge t6, t0, res # if counter >= number of elements, go to result

        # read the next (current) element
        jal ra, check
        mv t5, a0 # t5 holds the current element

        # calculate product of previous (t4) and current (t5)
        mul a0, t4, t5

        # check if the new product is larger than the max product so far
        ble a0, t1, no_update # if new_product <= max_product, skip the update

update:
        mv t1, a0 # update largest product
        mv t2, t4 # update first element of the pair
        mv t3, t5 # update second element of the pair
        j no_update # continue to the next step

no_update:
        # the current element now becomes the previous one for the next
iteration
        add t4, t5, zero
        j next

next:
        addi t6, t6, 1          # increment index counter
        j loop                          # repeat the loop
```

```
res:
        # print the output message
        li a7, 4
        la a0, msg2
        ecall

        # print the first element of the pair
        li a7, 1
        mv a0, t2
        ecall

        # print " "
        li a7, 4
        la a0, space
        ecall

        # print the second element of the pair
        li a7, 1
        add a0, t3, zero
        ecall

    # Exit cleanly
        li a7, 10
        ecall

error:
        # print the message for insufficient elements
        li a7, 4
        la a0, msg_error
        ecall
        j main

check:
    # read string from user
    li a7, 8
    la a0, buffer
    li a1, 30
    ecall
```

```
    # initialize parsing variables
    la s1, buffer # s1 = pointer to current char
    li s3, 0 # s3 = result
    li s4, 0 # s4 = sign flag (0 = +, 1 = -)

    lb s2, 0(s1)        # Load first byte

    # check for negative sign
    li s5, 45           # s5 = for '-'
    bne s2, s5, parse_loop
    li s4, 1            # set sign flag to negative
    addi s1, s1, 1      # move pointer forward
    lb s2, 0(s1)        # load next char

parse_loop:
    # Check for end of string (newline or null)
    li s5, 10           # s5 = '\n'
    beq s2, s5, end_parse
    beqz s2, end_parse  # Null terminator

    # digit must be from '0' to '9'
    li s5, 48           # s5 = '0'
    blt s2, s5, invalid_input
    li s5, 57           # s5 = '9'
    bgt s2, s5, invalid_input

    # result = (result * 10) + (char - 48)
    addi s2, s2, -48    # convert ASCII char to integer digit
    li s5, 10
    mul s3, s3, s5       # shift current result left (decimal)
    add s3, s3, s2      # add new digit

    # next char
    addi s1, s1, 1
    lb s2, 0(s1)
    j parse_loop
```

```
invalid_input:
    # print error message
    li a7, 4
    la a0, msg_error2
    ecall
    # jump back to start of check to read again
    j check

end_parse:
    # apply negative sign if needed
    beqz s4, return_result
    sub s3, zero, s3     # result = 0 - result

return_result:
    mv a0, s3           # move result to a0 for return
    ret
```

- Example input/output:

```
Input N: 5
2
3
4
5
6
Output: 5 6
```

```
Input N: 4
3
-5
-6
24
Output: -5 -6
```

Type C - Strings:

14. Enter two strings A and B, print out the uppercase letters that appear in both A and B.

- Understanding the problem:

  + Read two strings S and character C from user and store the inputs in memory
  + Check each letter from string S compare it with character C, regardless of case
  + If matches, increase the result
  + Print the result

- Implementation:

After the user reads string S and character C. For easier checking, we normalize every character in string S and character C to lowercase. First, we convert character C to lowercase if it's an uppercase character. Then, we iterate through every character in string S and do the same thing. At each character, we check if it's the same as character C. If so, we update the counter. Print the counter as the result.

- Source code:

```
.data
        S: .space 100
        C: .space 4

        msg1: .asciz "Enter string S: "
        msg2: .asciz "Enter character C: "
        msg3: .asciz "Occurrence count: "

.text
        # prompt and read string S
        li a7, 4
```

```
        la a0, msg1
        ecall
        li a7, 8
        la a0, S
        li a1, 100
        ecall

        # prompt and read character C
        li a7, 4
        la a0, msg2
        ecall
        li a7, 8
        la a0, C
        li a1, 4
        ecall

        # print result message
        li a7, 4
        la a0, msg3
        ecall

        # initialize registers
        la t0, S
        la t3, C
        lb t1, 0(t3) #t1 holds the character we are searching for
        li t6, 0 # t6 will be our counter, initialized to 0

        # character bounds for case conversion
        li t4, 'A'
        li t5, 'Z'


normalize_search_char:
        blt t1, t4, search_loop  # if char < 'A', it's not uppercase, so skip
        bgt t1, t5, search_loop  # if char > 'Z', it's not uppercase, so skip

        addi t1, t1, 32          # if uppercase, so convert to lowercase
```

```
search_loop:
        lb t2, 0(t0)
        beq t2, zero, res

        add t3, t2, zero          # copy t2 to t3
        blt t2, t4, search       # if char < 'A', not uppercase, skip conversion
        bgt t2, t5, search       # if char > 'Z', not uppercase, skip conversion
        addi t3, t3, 32          #if uppercase, convert to lowercase

search:

        bne t1, t3, next_char     # if they don't match, move to the next char

        # if they match, increment the counter
        addi t6, t6, 1

next_char:
        addi t0, t0, 1           # move pointer to the next character in the string
        j search_loop            # jump back to the top of the loop


res:
        li a7, 1
        add a0, t6, zero
        ecall
        li a7, 10
        ecall
```

- Example input/output:

```
Enter string S: ABcAAaa
Enter character C: a
Occurrence count: 5
```

```
Enter string S: AbcB
Enter character C: B
Occurrence count: 2
```