Napisac program w jezyku assemblera MIPS wypisujacy do konsoli komunikat ”Hello World”, a nastepnie konczacy swoje działanie. Program powinien zawierac: — sekcje danych; — sekcje tekstu; — poprawne zakonczenie.

```
.data
str: .asciiz "Hello World"   # napis do wypisania

.text
main:
    addi $v0, $zero, 4       # v0 = 4 (syscall: print string)
    la   $a0, str            # a0 = adres stringa
    syscall                  # wypisz string

    li $v0, 10               # v0 = 10 (syscall: exit)
    syscall                  # zakończ program
```

Jaki kod w języku asemblera MIPS odpowiada poniższemu wyrażeniu w języku C? Założyć, że zmienne f, g, h, i, j sa przypisane odpowiednio do rejestrów $s0, $s1, $s2, $s3, $s4 . Założyć, że bazowe adresy tablic A i B znajdują się odpowiednio w rejestrach $s6 i $s7.

Kod w języku C:

```
B[8]=A[i-j];
```

```
.data
f:    .word 0
g:    .word 0
h:    .word 0
i:    .word 5
j:    .word 3
A:    .word 1,2,3,4
B:    .word 0,1,2,3,4,5,6,7,8,9

.text
      lw $s0,f
```

```
      lw $s1,g
      lw $s2,h
      lw $s3,i
      lw $s4,j
      la $s6,A
      la $s7,B


      sub $t0, $s3, $s4     #operacja t1 = i - j = 2
      sll $t0, $t0, 2       #przesunięcie w lewo o 8 bajtów czyli (i-
j)*4
      add $t1, $s6, $t0     #ustawienie adresu A[i-j] czyli A[2]
      lw $t2, 0($t1)        #ustawienie wartości z pozycji A[i-j] z
tablicy? t2 = A[2]
      sw $t2, 32($s7)       #zapisz słowo w tablicy B na miejscu [8] o
wartości A[2]

  li $v0, 10
  syscall
```

Przetłumacz następujący kod w języku C na kod w języku asemblera MIPS. Założyć, że zmienne f, g, h, i, j są przypisane odpowiednio do rejestrów $s0, $s1, $s2, $s3, $s4. Założyć, że bazowe adresy tablic A i B znajdują się odpowiednio w rejestrach $s6 i $s7. Założyć, że elementy tablic A i B są 4-bajtowymi słowami.

Kod w języku C:

```
B[8]=A[i]+A[j];
```

```
.data
    f: .word 0
    g: .word 0
    h: .word 0
    i: .word 2
```

```
    j: .word 4
    A: .word 1,2,3,4
    B: .word 1,2,3,4,5,6,7,8,9

.text
    lw $s0, f #wczytujemy f itd:
    lw $s1, g
    lw $s2, h
    lw $s3, i
    lw $s4, j

    la $s6, A
    la $s7, B

    sll $t0, $s3, 2 # I jako 4 bitowe slowo
    sll $t1, $s4, 2 # J jako 4 bitowo slowo
    add $t0, $s6, $t0 # A[i] = A[0] + 4*i SAM ADRES
    add $t1, $s6, $t1 # A[j] = A[0] + 4*j SAM ADRES
    lw $t0, 0($t0) # Zapisujemy wartosc z indeksu
    lw $t1, 0($t1)
    add $t2, $t0, $t1 # t2 = A[i] + A[j]
    sw $t2, 32($s7) #B[8] = t2 Zapisujemy wartosc

    li $v0, 10
    syscall
```

Przetłumacz poniższy kod z języka C na język asemblera MIPS. Użyj minimalnej liczby instrukcji asemblera MIPS. Założyć, że wartości a, b, i, j są przechowywane odpowiednio w rejestrach $s0, $s1, $t0, $t1. Założyć również, że rejestr $s2 przechowuje adres bazowy tablicy D.

Kod w języku C:

```
for (i=0; i<a; i++)
    for (j=0; j<b; j++)
        D[4*j]=i+j;
```

```
.data
a:    .word 5
b:    .word 2
D:    .word 1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1

.text
      lw $s0,a
      lw $s1,b
      la $s2,D
      #pętla for(i)
      li $t0,0        #t0 = 0
cond_i:    slt $t2, $t0, $s0     #sprawdzenie czy i<a wtedy t2=1 prawda
lub t2=0 fałsz
      beq $t2, $zero, end_i # jesli t2 = 0 to "go to end_i"

      #pętla for(j)
      li $t1,0        #t0 = 0
cond_j: slt $t3, $t1, $s1  #sprawdzenie czy j<b wtedy t3=1 prawda lub
t3=0 fałsz
      beq $t3, $zero, end_j # jesli t3 = 0 to "go to end_j"

      #działanie wewnątrz for(j)
      add $t4, $t0, $t1     #t4=i+j
      sll $t5, $t1, 4       #t5=4*4*j
      add $t5, $s2, $t5     #t5=&D+16*j=&D[4j]
      sw $t4, 0($t5)        #D[4j]=i+j
      addi $t1, $t1, 1#j=j+1
      j cond_j        #go to cond_j
end_j:     addi $t0, $t0, 1#i=i+1
      j cond_i        #go to cond_i
end_i:     li $v0,10
      syscall         #terminate
```

Kod w języku C:

```c
int fib(int n){
    if (n==0)
        return 0;
    else if (n==1)
        return 1;
    else
        return fib(n-1)+fib(n-2);
}
```

```
.globl main
.data

n: .word 5   # Zmienna dla wartości n (w tym przypadku n = 5)

.text
main:
    lw $a0, n          # a0 := n (wczytanie wartości n)
    jal fib            # wywołanie funkcji fib
    move $a0, $v0      # a0 := wynik z fib (w $v0)
    li $v0, 1          # kod systemowy 1 (drukowanie liczby)
    syscall            # wywołanie systemowe (wypisanie liczby)
    li $v0, 10         # kod systemowy 10 (zakończenie programu)
    syscall            # wywołanie systemowe (zakończenie programu)

fib:
    li $t0, 1          # t0 = 1 (warunek zakończenia dla n = 1)
    bne $a0, $zero, test_1  # jeśli a0 != 0, idź do test_1
    li $v0, 0          # jeśli n == 0, fib(n) = 0
    jr $ra             # zakończenie funkcji

test_1:
    bne $a0, $t0, rek  # jeśli a0 != 1, idź do rekursji
    li $v0, 1          # jeśli n == 1, fib(n) = 1
    jr $ra             # zakończenie funkcji
```

```
rek:
    addi $sp, $sp, -12   # rezerwacja miejsca na 3 zmienne na stosie
    sw $a0, 8($sp)       # zapisz wartość $a0 na stosie (n)
    sw $ra, 4($sp)       # zapisz adres powrotu $ra na stosie
    sw $s0, 0($sp)       # zapisz rejestr $s0 na stosie

    addi $a0, $a0, -1    # a0 = n - 1
    jal fib              # wywołaj fib(n-1)
    move $s0, $v0        # $s0 = wynik fib(n-1)

    addi $a0, $a0, -1    # a0 = n - 2
    jal fib              # wywołaj fib(n-2)
    add $v0, $v0, $s0    # $v0 = fib(n-1) + fib(n-2)

    lw $s0, 0($sp)       # przywróć wartość $s0 ze stosu
    lw $ra, 4($sp)       # przywróć wartość $ra ze stosu
    lw $a0, 8($sp)       # przywróć wartość $a0 (n) ze stosu
    addi $sp, $sp, 12    # przywróć wskaźnik stosu

    jr $ra               # powrót do wywołującego
```

Kod w języku C:

```
int potega(int x, int n){
    if (n==0)
        return 1;
    else
        return x*potega(x,n-1);
}
```

```
.globl main

.data
potega_x:    .word 5
potega_n:    .word 3

.text
```

```
main:
    lw $a0, potega_x          # a0 = x
    lw $a1, potega_n          # a1 = n
    jal potega                # wywołanie potega(x, n)
    add $a0, $v0, 0           # przenieś wynik do a0 (do wypisania)
    li $v0, 1                 # syscall 1 = print integer
    syscall                   # wypisz wynik

    li $v0, 10                # syscall 10 = exit
    syscall

potega:
    bne $a1, 0, nzero         # jeśli n != 0 → idź do nzero
    li $v0, 1                 # n == 0 → zwróć 1 (definicja: x^0 = 1)
    jr $ra                    # powrót

nzero:
    addi $sp, $sp, -4         # zrób miejsce na stosie (4 bajty)
    sw $ra, 0($sp)            # zachowaj adres powrotu (ra) na stosie

    sub $a1, $a1, 1           # n = n - 1 → potega(x, n-1)
    jal potega                # wywołanie rekurencyjne

    add $t0, $v0, 0           # t0 = potega(x, n-1)
    mul $v0, $a0, $t0         # v0 = x * potega(x, n-1)

    lw $ra, 0($sp)            # przywróć ra
    addi $sp, $sp, 4          # zwolnij miejsce na stosie

    jr $ra                    # powrót do wywołującego
```

Kod w języku C:

```
int factorial(int n){
    if (n==0)
        return 1;
    else
        return n*factorial(n-1);
}
```

```
    .globl main

    .data
factorial_n:    .word 6

    .text
.text

main:
    lw $a0, factorial_n           # wczytaj wartość 6 do rejestru $a0
    jal factorial                 # wywołaj funkcję factorial(6)

    add $a0, $v0, 0               # przenieś wynik z $v0 do $a0
(przygotowanie do wypisania)
    li $v0, 1                     # syscall 1 = wydrukuj liczbę
całkowitą
    syscall                       # wykonanie syscall, wypisanie
wyniku

    li $v0, 10                    # syscall 10 = zakończenie programu
    syscall

factorial:
    bne $a0, 0, nzero             # jeśli $a0 != 0 idź do nzero
(rekurencja)
    li $v0, 1                     # jeśli $a0 == 0 zwróć 1 (0! = 1)
    jr $ra                        # powrót z funkcji

nzero:
    addi $sp, $sp, -8             # zrób miejsce na stosie (8 bajtów)
```

```
    sw $ra, 4($sp)                  # zapisz adres powrotu na stosie
    sw $a0, 0($sp)                  # zapisz argument $a0 na stosie

    sub $a0, $a0, 1                 # zmniejsz $a0 o 1 (przygotowanie do
rekurencji)
    jal factorial                   # wywołanie rekurencyjne

    lw $a0, 0($sp)                  # odczytaj poprzednie $a0 z stosu
    lw $ra, 4($sp)                  # odczytaj poprzedni adres powrotu
    addi $sp,$sp, 8                 # przywrócenie wskaźnika stosu

    mul $v0, $a0, $v0               # pomnóż n * factorial(n-1)

    jr $ra                          # wróć do wywołującego
```

Kod w języku C:

```
int factorial(int n){
    int temp=1;
    for (i=1; i<n+1; i=i+1) {
        temp=temp*i;
    }
    return temp;
}
```

```
    .globl main

    .data
factorial_n:    .word 6

    .text

main:
    lw $a0, factorial_n             # wczytaj n do $a0
    jal factorial                   # wywołaj funkcję factorial(n)
```

```
    move $a0, $v0                   # przenieś wynik do $a0, wymagane
dla print int
    li $v0, 1                       # syscall 1 = print integer
    syscall                         # wypisz wynik

    li $v0, 10                      # syscall 10 = zakończ program
    syscall

factorial:
    li $t0, 1                       # t0 = 1 (wynik)
    li $t1, 1                       # t1 = 1 (licznik i = 1)

poczatek_i:
    add $t2, $a0, 1                 # t2 = n + 1
    slt $t3, $t1, $t2               # jeśli t1 < (n+1), t3 = 1
    beq $t3, $zero, koniec_i        # jeśli t1 >= n+1, wyjdź z pętli

    mul $t0, $t0, $t1               # t0 = t0 * i
    add $t1, $t1, 1                 # i = i + 1
    j poczatek_i                    # wróć do początku pętli

koniec_i:
    add $v0, $t0, 0                 # wynik do $v0 (return value)
    jr $ra                          # powrót do main
```

Kod w języku C:

```
int potega(int x, int n){
    int temp=1;
    for (i=1; i<n+1; i=i+1) {
        temp=temp*x;
    }
    return temp;
}
```

```
    .globl main

    .data
potega_x:    .word 5
potega_n:    .word 4

    .text

main:
    lw $a0, potega_x              # wczytaj x do $a0
    lw $a1, potega_n              # wczytaj n do $a1
    jal potega                    # wywołaj funkcję potega(x, n)

    move $a0, $v0                 # przenieś wynik do $a0 (print int)
    li $v0, 1                     # syscall 1 = print integer
    syscall                       # wypisz wynik

    li $v0, 10                    # syscall 10 = exit
    syscall

potega:
    li $t0, 1                     # t0 = 1 (wartość potęgi)
    li $t1, 1                     # t1 = 1 (licznik i = 1)
    addi $t2, $a1, 1              # t2 = n + 1 (granica pętli)

poczatek:
    slt $t3, $t1, $t2             # t3 = 1 jeśli t1 < t2
    beq $t3, $zero, potega_exit   # jeśli t1 >= t2 → koniec

    mul $t0, $t0, $a0             # t0 = t0 * x   (mnożenie)
    addi $t1, $t1, 1              # i++

    j poczatek                    # kontynuuj pętlę

potega_exit:
    move $v0, $t0                 # wynik do $v0
    jr $ra                        # powrót
```

Przetłumacz funkcję f z języka C na język asemblera MIPS. Jeżeli będzie potrzeba użycia rejestrów tymczasowych, to wykorzystać rejestry w rosnącej kolejności numerów. Założyć, że deklaracja funkcji func jest następująca:

```
int func(int a, int b){
    return a+b};
```

Ponadto kod funkcji f jest następujący:

```
int f(int a, int b, int c, int d){
    return func(func(a,b),c+d);
}
```

```
    .globl main

    .data
f_a:    .word 30
f_b:    .word 30
f_c:    .word 40
f_d:    .word 150

    .text

main:
    lw $a0, f_a                 # a0 = 30
    lw $a1, f_b                 # a1 = 30
    lw $a2, f_c                 # a2 = 40
    lw $a3, f_d                 # a3 = 150
    jal f                       # wywołanie funkcji f()

    move $a0, $v0               # przenieś wynik do wypisania
    li $v0, 1                   # syscall 1 = print integer
    syscall

    li $v0, 10                  # syscall 10 = exit
    syscall


func:                             # func(a0, a1) -> a0 + a1
```

```
    add $v0, $a0, $a1           # v0 = a0 + a1
    jr $ra                      # powrót


f:
    addi $sp, $sp, -4           # miejsce na stosie
    sw $ra, 0($sp)              # zapis adresu powrotu

    jal func                    # wywołanie func(a0, a1)
                                # v0 = a0 + a1 = 30 + 30 = 60

    add $t1, $a2, $a3           # t1 = a2 + a3 = 40 + 150 = 190
    move $a0, $v0               # a0 = 60
    move $a1, $t1               # a1 = 190

    jal func                    # wywołanie func(60, 190)
                                # wynik = 250 → v0 = 250

    lw $ra, 0($sp)              # odtwórz ra
    addi $sp, $sp, 4            # przywróć stos
    jr $ra                      # powrót z f
```

Przetłumacz poniższy kod z języka C na język asemblera MIPS. Założyć, że wartości i, adres bazowy a, y są przechowywane odpowiednio w rejestrach $s0, $s1, $s2. Nie zapomnij napisać kodu procedury `int abs(int x)`.

Kod w języku C:

```
i=3;
y=y+abs(a[i]);
```

```
    .globl main

    .data
y:      .word 125
a:      .word 5,15,25,125,150,500

    .text
```

```
main:
    li $s0, 3                   # s0 = indeks = 3
    la $s1, a                   # s1 = adres początku tablicy a
    lw $s2, y                   # s2 = y = 125

    sll $t0, $s0, 2             # t0 = s0 * 4 (przesunięcie w
bajtach)
    add $t0, $s1, $t0           # t0 = adres elementu a[3]
    lw $a0, 0($t0)              # a0 = a[3] = 125

    jal abs                     # wywołanie funkcji abs(a0)
                                # v0 = |125| = 125

    add $s2, $s2, $v0           # s2 = 125 + 125 = 250

    move $a0, $s2               # przygotowanie wyniku do wypisania
    li  $v0, 1                  # syscall 1 = print integer
    syscall

    li $v0, 10                  # syscall 10 = exit
    syscall

abs:
    abs $v0, $a0                # v0 = |a0| – instrukcja MIPS
    jr $ra                      # powrót
```

Funkcja `sum` oblicza sumę elementów tablicy A pozynając od elementu o numerze p i kończąc na elemencie o numerze k. Założyć, że deklaracja funkcji func jest następująca:

```
int sum(int A[], int p, int k);
```

Napisać program w języku asemblera MIPS obliczający średnią arytmetyczną elementów tablicy A rozmiaru N. Założyć, że adres bazowy tablicy A jest przekazywany w rejestrze $a0,

a rozmiar N w rejestrze $a1. Nie zapomnij napisać kodu funkcji `sum`.

```
.data
A:    .word 1,2,3,4,5,6,7,8,9
N:    .word 9

.text
main:
      la $a0, A          # a0 = adres tablicy A
      lw $a1, N          # a1 = liczba elementów
      jal mean           # wywołanie funkcji mean
      move $a0, $v0           # wynik średniej do a0
      li $v0, 1          # print_int
      syscall
      li $v0, 10         # exit
      syscall

sum:
      li $v0, 0       # v0 = suma = 0
      sll $t1, $a1, 2       # t1 = przesunięcie 4*i (rozmiar słowa)
      add $t1, $a0, $t1     # t1 = &A[i]
      move $t2, $a1         # t2 = i (bieżący indeks)
      addi $t3, $a2, 1# t3 = k+1 (warunek końca)

sum_t_i:
      slt $t4, $t2, $t3     # jeśli i < k+1 → t4 = 1
      beq $t4, $zero, sum_e # jeśli nie (i >= k+1) → koniec sumowania
      lw $t5, 0($t1)        # t5 = A[i]
      add $v0, $v0, $t5     # suma += A[i]
      addi $t1, $t1, 4# przejście do A[i+1]
      addi $t2, $t2, 1# i++
      j sum_t_i       # powrót do pętli

sum_e:
      jr $ra                # powrót z sum

mean:
      addi $sp, $sp, -8
      sw $a1, 4($sp)        # zachowaj n
      sw $ra, 0($sp)        # zachowaj adres powrotu

      addi $a2, $a1, -1     # a2 = n - 1 (ostatni indeks)
      li $a1, 0       # a1 = 0 (pierwszy indeks)

      jal sum         # v0 = suma elementów

      lw $ra, 0($sp)        # przywróć ra
      lw $a1, 4($sp)        # przywróć n
      addi $sp, $sp, 8

      div $v0, $a1          # LO = suma / n
      mflo $v0        # wynik średniej do v0
```

```
    jr $ra
```

Napisz program w języku asemblera MIPS, który będzie konwertował łańcuch znaków ASCII zawierający dodatnia liczbę całkowitą na wartość tej liczby całkowitej. Założyć, że rejestr $a0 przechowuje adres bazowy łańcucha znaków ASCI zakończonego znakiem „null”. Po obliczeniu całkowitoliczbowego odpowiednika tego łańcucha program powinien umieścić wynik w rejestrze $v0. Jeżeli w łańcuchu pojawia się znak nie będący cyfrą dziesiętną, to program powinien się zakończyć z wynikiem -1 w rejestrze $v0.

```
.globl main

.data
num:    .asciiz "25"

.text

main:
    la $a0, num          # a0 = adres stringa "25"
    jal stoi             # konwersja string → liczba
    move $a0, $v0        # przenieś wynik do a0
    li $v0, 1            # print_int
    syscall
    li $v0, 10           # exit
    syscall

stoi:
    li $v0, 0            # v0 = wynik = 0
    move $t0, $a0        # t0 = aktualny wskaźnik na znak

stoi_next:
    lb  $t1, 0($t0)      # wczytaj kolejny znak
    beq $t1, 0, stoi_exit # koniec stringa → wyjście
```

```
    bgt $t1, 57, stoi_err # jeśli znak > '9' → błąd
    blt $t1, 48, stoi_err # jeśli znak < '0' → błąd

    mul $v0, $v0, 10     # wynik = wynik * 10
    sub $t1, $t1, 48     # zamiana ASCII → cyfra
    add $v0, $v0, $t1    # dodaj cyfrę
    addi $t0, $t0, 1     # przejdź do następnego znaku
    j stoi_next

stoi_exit:
    jr $ra               # wyjście

stoi_err:
    li $v0, -1           # kod błędu
    jr $ra
```

Przetłumacz poniższy kod z języka C na język asemblera MIPS. Założyć, że wartości x jest przechowywana w rejestrze $a0.

Kod w języku C:

```
int bitcount(unsigned x){
    int bit;
    if (x==0) return 0;
    bit = x & 0x1;
    return bit + bitcount(x>>1);
}
```

```
.globl main

.data
x_val:  .word 255          # przykładowa liczba do policzenia bitów

.text
main:
        lw   $a0, x_val    # wczytaj x do a0
        jal  bitcount      # wywołaj funkcję liczącą bity
```

```
        move $a0, $v0      # przygotuj wynik do wypisania
        li $v0, 1          # syscall: print integer
        syscall
        li $v0, 10         # syscall: exit
        syscall

bitcount:
        bne $a0, $zero, b_nonzero   # jeśli a0 != 0 → przejdź do
b_nonzero
        li   $v0, 0                 # jeśli a0 == 0 → zwróć 0
        jr   $ra                    # powrót

b_nonzero:
        addi $sp, $sp, -4           # rezerwuj miejsce na stosie
        sw   $ra, 0($sp)            # zapisz adres powrotu

        andi $t0, $a0, 1            # t0 = najmłodszy bit (0 albo 1)
        srl  $a0, $a0, 1            # a0 = a0 >> 1 (przesuń w prawo)

        jal  bitcount               # rekurencyjne wywołanie

        add  $v0, $v0, $t0          # dodaj bit do wyniku

        lw $ra, 0($sp)              # przywróć ra
        addi $sp,$sp, 4             # przywróć stos

        jr   $ra                    # powrót z funkcji
```

Kod w języku C:

```
int tau(int n, int m){
    return (2^n)*(2*m+1)-1;
}
```

```
.globl main

.data
x:  .word 1        # wartość x
n:  .word 1        # wartość n

.text
main:
```

```
        lw   $a0, x         # a0 = x
        lw   $a1, n         # a1 = n
        jal  tau            # wywołanie funkcji tau(x, n)
        move $a0, $v0       # przygotowanie wyniku do wypisania
        li   $v0, 1         # syscall: print integer
        syscall
        li   $v0, 10        # syscall: exit
        syscall
tau:
        li   $t0, 1             # t0 = 1
        sllv $t0, $t0, $a0      # t0 = 1 << x = 2^x
        sll  $t1, $a1, 1        # t1 = n << 1 = 2n
        addi $t1, $t1, 1        # t1 = 2n + 1
        mul  $t2, $t0, $t1      # t2 = (2^x)*(2n+1)
        addi $v0, $t2, -1       # v0 = t2 - 1
        jr   $ra                # powrót
```

Drugi Wariant by Francuz

```
.data
n: 1
m: 1

.text
     lw $a0, n              # a0 = n
     lw $a1, m              # a1 = m
     jal tau                # wywołanie funkcji tau(n, m)
     move $a0, $v0          # przygotowanie wyniku do wypisania
     li $v0, 1              # syscall: print integer
     syscall
     li $v0, 10             # syscall: exit
     syscall

tau:
     li $t0, 1              # t0 = 1 (początek obliczania 2^n)
     move $t1, $a0          # t1 = n (licznik w pętli)

potega:
     beq $t1, $zero, pot_end # jeśli n = 0 → przejście do końca
     sll $t0, $t0, 1        # t0 = t0 * 2 (czyli przesunięcie bitowe – 
szybka potęga)
     addi $t1, $t1, -1      # zmniejsz licznik
     j potega               # kontynuacja pętli

pot_end:
     mul $t2, $a1, 2        # t2 = 2 * m
     addi $t2, $t2, 1       # t2 = 2*m + 1
     mul $t3, $t0, $t2      # t3 = (2^n) * (2*m + 1)
```

```
    addi $v0, $t3, -1       # v0 = t3 - 1 = (2^n)*(2*m+1) - 1
    jr $ra                  # powrót
```

Kod w języku C:

```
int rm(int x, int y){
    if (y==0)
        return 0;
    elseif (rm(x,y)+1==x)
        return 0;
    else
        return rm(x,y)+1;
}
```

Oby tego nie było :(

```
.globl main

.data
x:  .word 1
y:  .word 1

.text
main:
        lw   $a0, x
        lw   $a1, y
        jal rm              # wywołanie funkcji rm(x, y)
        move $a0, $v0       # przygotowanie wyniku do wypisania
        li   $v0, 1         # syscall: print integer
```

```
        syscall
        li   $v0, 10       # syscall: exit
        syscall

rm:
    bne $a1, $zero, y_nonzero   # jeśli y != 0 → przejdź dalej
    li   $v0, 0                # jeśli y == 0 → zwróć 0
    jr   $ra

y_nonzero:
    addi $sp, $sp, -4           # miejsce na stosie
    sw   $ra, 0($sp)            # zachowaj adres powrotu

    jal rm                      # rekurencja: rm(x, y-1)
    move $t0, $v0               # t0 = poprzedni wynik

    addi $t1, $t0, 1            # t1 = wynik + 1
    bne $t1, $a0, dalej         # jeśli (wynik+1 != x) → idź dalej

    li $v0, 0                   # jeśli (wynik+1 == x): zwróć 0
    lw $ra, 0($sp)
    addi $sp, $sp, 4
    jr $ra

dalej:
    addi $v0, $t0, 1            # v0 = wynik + 1
    lw $ra, 0($sp)              # przywróć ra
    addi $sp, $sp, 4            # usuń dane ze stosu
    jr $ra                      # powrót
```