Зображення, що містить текст

Автоматично згенерований опис

Міністерство освіти і науки України

Національний технічний університет України

“Київський політехнічний інститут імені Ігоря Сікорського”

Факультет інформатики та обчислювальної техніки

Кафедра інформатики та програмної інженерії

**Комп’ютерний практикум №4**

Системне програмне забезпечення

**Тема:** Масиви

Виконав Перевірив:

студент групи ІП-11: Лісовиченко О. І

Панченко С. В. “8” березня 2023 р.

Київ 2023

Зміст

[1 Мета комп’ютерного практикуму 7](#__RefHeading___Toc3319_204808571)

[2 Завдання 8](#__RefHeading___Toc5937_3437555080)

[3 Текст програми 9](#__RefHeading___Toc4957_2404284615)

[3.1 Програма 1 9](#__RefHeading___Toc5455_2579377654)

[3.2 Програма 2 33](#__RefHeading___Toc5457_2579377654)

[4 Схема функціонування програми 58](#__RefHeading___Toc4963_2404284615)

[5 Приклад виконання 70](#__RefHeading___Toc2239_1691502584)

[6 Висновок 75](#__RefHeading___Toc4971_2404284615)

[6.1 Команди організації циклів 75](#__RefHeading___Toc5468_2579377654)

[6.1.1 JMP 75](#__RefHeading___Toc5470_2579377654)

[6.1.2 CMP 75](#__RefHeading___Toc5472_2579377654)

[6.1.3 INC, DEC 75](#__RefHeading___Toc5474_2579377654)

[6.2 Рядкові команди та особливості їх використання. 75](#__RefHeading___Toc5476_2579377654)

[6.2.1 LODS (завантажити рядок) 75](#__RefHeading___Toc5478_2579377654)

[6.2.2 STOS (зберігати рядок) 75](#__RefHeading___Toc5480_2579377654)

[6.2.3 REP (Повторення) 75](#__RefHeading___Toc5482_2579377654)

[6.2.4 REPE або REPZ (Повторити при рівній кількості) 75](#__RefHeading___Toc5484_2579377654)

[6.2.5 REPNE або REPNZ (Повторити, якщо значення не дорівнює) 76](#__RefHeading___Toc5486_2579377654)

[6.2.6 CMPS (рядок порівняння) 76](#__RefHeading___Toc5488_2579377654)

[6.2.7 SCAS (сканування рядка) 76](#__RefHeading___Toc5490_2579377654)

[6.2.8 MOVSX (переміщення із розширенням знака) 76](#__RefHeading___Toc5492_2579377654)

[6.2.9 MOVZX (Переміщення з нульовим розширенням) 76](#__RefHeading___Toc5494_2579377654)

[6.2.10 MOVS (переміщення рядка) 76](#__RefHeading___Toc5496_2579377654)

[6.3 Методи адресації за базою, з індексуванням, з подвійним індексуванням 76](#__RefHeading___Toc5498_2579377654)

[6.3.1 Індексована адресація 76](#__RefHeading___Toc5500_2579377654)

[6.3.2 Подвійна індексована адресація 77](#__RefHeading___Toc5502_2579377654)

[6.3.3 Адресація на основі 77](#__RefHeading___Toc5504_2579377654)

# Мета комп’ютерного практикуму

Навчитись програмувати масиви та проводити операції над ними.

# Завдання

1. Написати програму, яка повинна мати наступний функціонал:

* Можливість введення користувачем розміру одномірного масиву.
* Можливість введення користувачем значень елементів одномірного масиву.
* Можливість знаходження суми елементів одномірного масиву.
* Можливість пошуку максимального (або мінімального) елемента одномірного масиву.
* Можливість сортування одномірного масиву цілих чисел загального вигляду.

1. Написати програму, яка повинна мати наступний функціонал:

* Можливість введення користувачем розміру двомірного масиву.
* Можливість введення користувачем значень елементів двомірного масиву.
* Можливість пошуку координат всіх входжень заданого елемента в двомірному масиві, елементи масиву та пошуковий елемент вводить користувач.
* Програма повинна мати захист від некоректного введення вхідних даних (символи, переповнення і т.і.)

# Текст програми

## Програма 1

bits 64

; list of system calls

; https://filippo.io/linux-syscall-table/

SYS\_READ equ 0

SYS\_WRITE equ 1

SYS\_IOCTL equ 16

; Descriptors

STDIN equ 0

STDOUT equ 1

ICANON: equ 1<<1

TCGETS: equ 0x5401

TCSETS equ 0x5402

; ASCII characters

NULL\_TERMINATOR equ 0

NEW\_LINE\_CHARACTER equ 10

SPACE equ 32

PLUS\_SIGH equ 43

MINUS\_SIGN equ 45

PERIOD equ 46

DIGIT\_ZERO equ 48

DIGIT\_NINE equ 57

Y\_LETTER equ 121

; Other constants

BUFFER\_LENGTH equ 20

MAX\_LENGTH equ 10

MAX\_SIZE equ 10

; !!! SECTION DATA !!!

section .data

; Errors

error\_incorrect\_symbol: db "Incorrect symbol in input", NEW\_LINE\_CHARACTER, 0

error\_incorrect\_symbol\_length: equ $-error\_incorrect\_symbol

error\_sign\_character\_not\_first db "Sign characters must be first", NEW\_LINE\_CHARACTER, 0

error\_sign\_character\_not\_first\_length equ $-error\_sign\_character\_not\_first

error\_not\_positive db "Value is not positive", NEW\_LINE\_CHARACTER, 0

error\_not\_positive\_length equ $-error\_not\_positive

max\_length\_error db "Max length is 10", NEW\_LINE\_CHARACTER, 0

max\_length\_error\_length equ $-max\_length\_error

; Buffers

buffer: times BUFFER\_LENGTH db 0

inputtedLength: dq 0

; Messages

enterArraySize db "Enter array size: ", NEW\_LINE\_CHARACTER, 0

enterArraySizeLength equ $-enterArraySize

enterArray db "Enter array: ", NEW\_LINE\_CHARACTER, 0

enterArrayLength equ $-enterArray

enterArrayElement db "Enter array element ", 0

enterArrayElementLength equ $-enterArrayElement

sumArray db "Sum of array elements: ", NEW\_LINE\_CHARACTER, 0

sumArrayLength equ $-sumArray

maxArrayElement db "Max array element: ", NEW\_LINE\_CHARACTER, 0

maxArrayElementLength equ $-maxArrayElement

minArrayElement db "Min array element: ", NEW\_LINE\_CHARACTER, 0

minArrayElementLength equ $-minArrayElement

sortedArray db "Sorted array: ", NEW\_LINE\_CHARACTER, 0

sortedArrayLength equ $-sortedArray

continue\_msg db "Continue? y - yes, other - no:", NEW\_LINE\_CHARACTER, 0

continue\_msg\_length equ $-continue\_msg

; !!! SECTION TEXT !!!

section .text

global asm\_main

;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

asm\_main:

push rax

push rbx

push rcx

push rdi

push rsi

push r12

.loopDD:

.inputArraySize:

mov rax, enterArraySize

mov rdi, enterArraySizeLength

call WriteToConsole

xor rsi, rsi

mov rax, buffer

mov rdi, BUFFER\_LENGTH

call InputArgumentPositive

; rax - buffer, rdi - bufferLength, rsi - arraySize

.allocMemoryOnStack:

push rax

push rdi

mov rax, enterArray

mov rdi, enterArrayLength

call WriteToConsole

pop rdi

pop rax

; inputted value

xor rbx, rbx

; counter to zero

xor rcx, rcx

; the beggining of the array

mov r12, rsp

sub r12, 8

.loopAllocStack:

cmp rcx, rsi

jge .OnLoopAllocStackEnd

.printEnterArrayElement:

push rax

push rdi

mov rax, enterArrayElement

mov rdi, enterArrayElementLength

call WriteToConsole

pop rdi

pop rax

push rdi

push rsi

mov rsi, rdi

mov rdi, rax

mov rdx, rcx

call PrintInteger

pop rsi

pop rdi

call PrintEndl

push rsi

call InputArgument

mov rbx, rsi

pop rsi

push rbx

inc rcx

; jump to the loop head

jmp .loopAllocStack

.OnLoopAllocStackEnd:

xor rcx, rcx

xor rbx, rbx

.callSortArray:

push rdi

push rsi

push rdx

mov rdi, r12

mov rdx, 8

call SortArray

pop rdx

pop rsi

pop rdi

.callPrintArray

; rax - buffer, rdi - bufferLength, rsi - arraySize,

; r12 - beginning of the array, rcx - 0

; rbx - 0, rdx - 0

push rax

push rdi

mov rax, sortedArray

mov rdi, sortedArrayLength

call WriteToConsole

pop rdi

pop rax

; mov rsi into r8 arrayLength

mov r8, rsi

; mov address of the first element of the array

mov rdx, r12

; mov bufferLength from rdi to rsi

mov rsi, rdi

mov rdi, rax

call PrintArray

; r12 - beginning of the array, r8 - arrayLength, rsi - bufferLength

; rdi - buffer, rax - buffer, rdx - beginning of the array,

; rbx - 0, rcx - 0

.printMinElement:

call PrintEndl

push rax

push rdi

mov rax, minArrayElement

mov rdi, minArrayElementLength

call WriteToConsole

pop rdi

pop rax

mov rdx, qword [r12]

call PrintInteger

call PrintEndl

.printMaxElement:

push rax

push rdi

mov rax, maxArrayElement

mov rdi, maxArrayElementLength

call WriteToConsole

pop rdi

pop rax

mov rbx, 8

mov rcx, r12

push rax

mov rax, r8

dec rax

imul rbx

sub rcx, rax

mov rdx, qword [rcx]

pop rax

call PrintInteger

call PrintEndl

.printSum:

push rax

push rdi

mov rax, sumArray

mov rdi, sumArrayLength

call WriteToConsole

pop rdi

pop rax

push rdi

push rsi

push rax

mov rdi, r12

mov rsi, r8

mov rdx, 8

call ArraySum

mov rdx, rax

pop rax

pop rsi

pop rdi

call PrintInteger

call PrintEndl

.deAddlocateArray:

xor rcx, rcx

.loop:

cmp rcx, r8

jge .end

pop rax

inc rcx

jmp .loop

.end:

mov rax, continue\_msg

mov rdi, continue\_msg\_length

call WriteToConsole

mov rax, buffer

mov rdi, BUFFER\_LENGTH

call ReadIntoBuffer

xor rbx, rbx

mov bl, byte [rax]

cmp bl, Y\_LETTER

je .loopDD

pop r12

pop rsi

pop rdi

pop rcx

pop rbx

pop rax

ret

;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

InputArgumentPositive:

; void (rax-buffer, rdi-bufferLength, rsi- int&\_out number)

.loopWhileNegative:

call InputArgument

cmp rsi, 0

jg .End

push rax

push rdi

mov rax, error\_not\_positive

mov rdi, error\_not\_positive\_length

call WriteToConsole

pop rdi

pop rax

jmp .loopWhileNegative

.End:

ret

;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

ArraySum:

; rax - sum, rdi - array, rsi - arrayLength, rdx - typeSize

push rcx

push rbx

push r8

push r9

xor rax, rax

xor rcx, rcx

; long int size

mov rbx, 8

; mov the array beginning to r8

mov r8, rdi

.loop:

cmp rcx, rsi

jge .end

mov r9, qword [r8]

add rax, r9

.switchToNextElement:

sub r8, rbx

inc rcx

jmp .loop

.end:

pop r9

pop r8

pop rbx

pop rcx

ret

;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

SortArray:

; rdi - array, rsi - arrayLength, rdx - typeSize

push rax

push rbx

push rdi

push rsi

push r8

push rcx

push rdx

push r8

push r9

push r10

push r11

mov rbx, rdx

; counter i to zero

xor rcx, rcx

; bubble sort

.loopSortOne:

; if(i >= arraySize) break

cmp rcx, rsi

jge .onLoopSortOneEnd

; i = j

mov rdx, rcx

.loopSortTwo:

; if(j >= arraySize) break;

cmp rdx, rsi

jge .onLoopSortTwoEnd

push rsi

; rsi = i

; rdx = 8 // sizeof(long int)

push rdx

mov rsi, rcx

mov rdx, rbx

call GetElementAddressByIndex

; r8 = &array[i]

mov r8, rax

pop rdx

push rdx

mov rsi, rdx

mov rdx, rbx

call GetElementAddressByIndex

; r9 = &array[j]

mov r9, rax

pop rdx

; rax = array[i]

; rdi = array[j]

mov r10, qword [r8]

mov r11, qword [r9]

; if(array[i] > array[j])

cmp r10, r11

jg .swap

jmp .notSwap

.swap:

mov qword [r8], r11

mov qword [r9], r10

.notSwap

pop rsi

inc rdx

jmp .loopSortTwo

.onLoopSortTwoEnd:

inc rcx

jmp .loopSortOne

.onLoopSortOneEnd:

pop r11

pop r10

pop r9

pop r8

pop rdx

pop rcx

pop r8

pop rsi

pop rdi

pop rbx

pop rax

ret

;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

PrintArray:

; rax - void (rdi - buffer, rsi - bufferLength, rdx - array, r8 - arrayLength )

push rcx

push rbx

push r9

; long int size

mov rbx, 8

; zero counter

xor rcx, rcx

; mov the array beginning to r9

mov r9, rdx

.loop:

cmp rcx, r8

jge .end

.callPrintInteger:

push rdx

mov rdx, qword [r9]

call PrintInteger

pop rdx

.callPrintSpace:

push rax

mov rax, rdi

call PrintSpace

pop rax

.switchToNextElement:

sub r9, rbx

inc rcx

jmp .loop

.end:

pop r9

pop rbx

pop rcx

ret

;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

PrintInteger:

; rax - void (rdi - buffer, rsi - bufferLength, rdx - number)

push rax

push rdi

push rsi

push rdx

push r8

mov rax, rdi

mov rdi, rsi

mov r8, rdi

xor rsi, rsi

call ClearBuffer

call TryConvertNumberToString

mov rdi, rsi

call WriteToConsole

mov rdi, r8

call ClearBuffer

pop r8

pop rdx

pop rsi

pop rdi

pop rax

ret

;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

GetElementAddressByIndex:

; rax - address (rdi - array, rsi - index, rdx - typeSize)

push rbx

push rdx

push r8

mov r8, rdi

mov rbx, rdx

xor rdx, rdx

mov rax, rsi

imul rbx

sub r8, rax

mov rax, r8

pop r8

pop rdx

pop rbx

ret

;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

InputArgument:

; void (rax-buffer, rdi-bufferLength, rsi- int&\_out number)

push rax

push rdi

push rdx

push r8

push r9

push r10

mov r9, rax

mov r10, rdi

xor rsi, rsi

.loop:

mov rax, r9

mov rdi, r10

call ReadIntoBuffer

call TryConvertStringToInteger

call ClearBuffer

cmp r8, 0

je .loop

call ClearBuffer

mov rsi, rdx

pop r10

pop r9

pop r8

pop rdx

pop rdi

pop rax

ret

;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

procABS:

; rdi Abs(rax);

mov rdi, rax

cmp rdi, 0

jge .End

neg rdi

.End:

ret

;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

PrintSpace:

; Function printing enl

; void PrintEndl(char\* const buffer);

; Params:

; rax: char\* buffer

; Returns:

; void

push rdi

push rbx

mov bl, byte [rax]

mov byte [rax], SPACE

mov rdi, 1

call WriteToConsole

mov byte [rax], bl

pop rbx

pop rdi

ret

;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

PrintEndl:

; Function printing enl

; void PrintEndl(char\* const buffer);

; Params:

; rax: char\* buffer

; Returns:

; void

push rdi

push rbx

mov bl, byte [rax]

mov byte [rax], NEW\_LINE\_CHARACTER

mov rdi, 1

call WriteToConsole

mov byte [rax], bl

pop rbx

pop rdi

ret

;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

PrintPlus:

; Function printing enl

; void PrintEndl(char\* const buffer);

; Params:

; rax: char\* buffer

; Returns:

; void

push rdi

push rbx

mov bl, byte [rax]

mov byte [rax], PLUS\_SIGH

mov rdi, 1

call WriteToConsole

mov byte [rax], bl

pop rbx

pop rdi

ret

PrintMinus:

; Function printing enl

; void PrintEndl(char\* const buffer);

; Params:

; rax: char\* buffer

; Returns:

; void

push rdi

push rbx

mov bl, byte [rax]

mov byte [rax], MINUS\_SIGN

mov rdi, 1

call WriteToConsole

mov byte [rax], bl

pop rbx

pop rdi

ret

;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

ClearBuffer:

; Function clearing buffer

; void ClearBuffer(char\* buffer, int length);

; Params:

; rax: char\* buffer

; rdi: int length

; Returns:

; void

push rcx

xor rcx, rcx

.loop:

cmp rcx, rdi

jbe .End

mov byte [rax + rcx], 0

jmp .loop

.End:

pop rcx

ret

;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

TryConvertNumberToString:

; Function converting integer to string;

; bool TryConvertNumberToString(char\* buffer, int bufferLength, int inputtedLength, int number);

; Params:

; rax: char\* buffer

; rdi: int bufferLength

; rsi: int& inputtedLength

; rdx: int number

; Returns:

; r8: bool if true, no error, else throwed error

;

pushf

push rbx

push rcx

push r8

push r9

xor r9, r9

xor r8, r8

; counter = 0

mov rcx, 0

cmp rdx, 0

jge .ReadingNumbersIntoStack

.CheckForNegative:

mov r9, 1

neg rdx

mov byte [rax], MINUS\_SIGN

.ReadingNumbersIntoStack:

; The idea behind this is to read number into stack

; For example, 123 into stack like "3","2","1"

; and we counted digits. In this, example count = 3

; so we need to do smth like that:

; while(index<count) {

; pop stack into var

; var = var + ZERO\_CODE

; \*buffer[index] = var

; ++index

; }

; copy value to rbx

mov rbx, rdx

cmp rbx, 0

je .zero

.loop:

cmp rbx, 0

jle .ReadingNumbersFromStackToBuffer

.ReadDigit:

push rax

push rdx

; rax\_rbx\_copy = rbx;

mov rax, rbx

; rdx = 0; rbx = 10

xor rdx, rdx

mov rbx, 10

; rax\_rbx\_copy, rdx\_remaindex = rax\_rbx\_copy / rbx

idiv rbx

; r8 = rdx\_remainder; rbx = rax\_rbx\_copy

mov r8, rdx

mov rbx, rax

pop rdx

pop rax

push r8

inc rcx

jmp .loop

jmp .ReadingNumbersFromStackToBuffer

.zero:

push 0

inc rcx

.ReadingNumbersFromStackToBuffer:

xor rbx, rbx

xor r8, r8

cmp r9, 0

je .Preparation

.IncrementIfNegative:

inc r8

inc rcx

.Preparation:

mov rsi, rcx

.loop2:

cmp r8, rcx

jge .NoError

pop rbx

add rbx, DIGIT\_ZERO

mov [rax + r8], bl

inc r8

jmp .loop2

.NoError:

pop r9

pop r8

pop rcx

pop rbx

popf

ret

;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

WriteToConsole:

; Function writing string into STDOUT

; void WriteToConsole(char\* buffer, int bufferLength)

; Params:

; rax: char\* buffer

; rdi: int bufferLength

; Returns:

; void

push rax

push rdi

push rsi

push rdx

mov rsi, rax

mov rdx, rdi

mov rax, SYS\_WRITE

mov rdi, STDOUT

call DoSystemCallNoModify

pop rdx

pop rsi

pop rdi

pop rax

ret

;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

ReadIntoBuffer:

; Function reading string into buffer;

; void ReadIntoBuffer(char\* buffer, int bufferLength, int\* inputtedLength);

; Params:

; rax: char\* buffer

; rdi: int bufferLength

; rsi: int& inputtedLength

; Returns:

; void

push rax

push rdi

push rdx

push r8

push r9

mov r8, rax

mov r9, rdi

mov rsi, 0

.loop:

mov rsi, r8

mov rdx, r9

mov rax, SYS\_READ

mov rdi, STDIN

call DoSystemCallNoModify

cmp rax, MAX\_LENGTH

jle .NoError

mov rax, max\_length\_error

mov rdi, max\_length\_error\_length

call WriteToConsole

jmp .loop

.NoError:

mov rsi, rax

pop r9

pop r8

pop rdx

pop rdi

pop rax

ret

;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

DoSystemCallNoModify:

; Function doing system call without

; modifying rcx and r11 registers after the call.

; type(rax) sys\_call(rax, rdi, rsi, rdx, r8, r9...);

; The reason behind this function is that in x64 NASM

; system call neither stores nor loads any registers

; it just uses and modifies them.

; https://stackoverflow.com/questions/47983371/why-do-x86-64-linux-system-calls-modify-rcx-and-what-does-the-value-mean

; http://www.int80h.org/bsdasm/#system-calls

; https://docs.freebsd.org/en/books/developers-handbook/x86/#x86-system-calls

pushf

push rcx

push r11

syscall

pop r11

pop rcx

popf

ret

;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

TryConvertStringToInteger:

; Function converting string to integer;

; bool TryConvertStringToInteger(char\* buffer, int bufferLength, int inputtedLength, int\* number);

; Params:

; rax: char\* buffer

; rdi: int bufferLength

; rsi: int inputtedLength

; rdx: int& number

; Returns:

; r8: bool if true, no error, else throwed error

;

pushf

push rbx

push rcx

push r9

push r10

xor rdx, rdx

xor r10, r10

xor r9, r9

mov rdx, 0

mov rcx, rsi

xor rbx, rbx

xor r8, r8

dec rcx

cmp byte [rax + rcx], NEW\_LINE\_CHARACTER

jne .loop

dec rcx

.loop:

cmp rcx, 0

jl .NoError

mov bl, byte [rax + r9]

.IsNewLineCharacter:

cmp bl, NEW\_LINE\_CHARACTER

je .NoError

.CheckForSigns:

.IsPlusCharacter:

cmp bl, PLUS\_SIGH

je .CheckForSignBeingFirst

.IsMinusCharacter:

cmp bl, MINUS\_SIGN

je .OnEqualMinus

jmp .CallIsDigit

.OnEqualMinus:

mov r10, 1

.CheckForSignBeingFirst:

cmp r9, 0

jne .ErrorSignNotFirst

jmp .OnIterationEnd

.CallIsDigit:

push rax

push rdi

xor rax, rax

mov al, bl

call IsDigit

mov r8, rdi

pop rdi

pop rax

cmp r8, 0

je .ErrorIncorrectSymbol

sub bl, DIGIT\_ZERO

.CallPow:

push rax

push rdi

push rsi

mov rax, 10

mov rdi, rcx

call Pow

imul rsi, rbx

mov rdi, rdx

add rdi, rsi

mov rdx, rdi

pop rsi

pop rdi

pop rax

; whole\_digit = digit\*(10^counter)

.OnIterationEnd:

dec rcx

inc r9

jmp .loop

.ErrorSignNotFirst:

mov r8, 0

push rax

push rdi

mov rax, error\_sign\_character\_not\_first

mov rdi, error\_sign\_character\_not\_first\_length

call WriteToConsole

pop rdi

pop rax

jmp .End

.ErrorIncorrectSymbol:

; print error message

mov r8, 0

push rax

push rdi

mov rax, error\_incorrect\_symbol

mov rdi, error\_incorrect\_symbol\_length

call WriteToConsole

pop rdi

pop rax

jmp .End

.NoError:

cmp r10, 1

jne .GeneralNoError

push rax

mov rax, rdx

neg rax

mov rdx, rax

pop rax

.GeneralNoError:

mov r8, 1

jmp .End

.End:

pop r10

pop r9

pop rcx

pop rbx

popf

ret

;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

IsDigit:

; Function checking whether byte value

; is in digit codes range

; bool IsDigit(char c);

; Params:

; rax: char c

; Returns:

; rdi: bool if true, then it is digit, else not

pushf

cmp al, DIGIT\_ZERO

jl .Invalid

cmp al, DIGIT\_NINE

jg .Invalid

mov rdi, 1

jmp .End

.Invalid:

mov rdi, 0

jmp .End

.End:

popf

ret

;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

Pow:

; Function powing number to a certain degree.

; Params:

; rax: int number

; rdi: int degree

; Returns:

; rsi: Powed number

pushf

push rcx

mov rsi, 1

xor rcx, rcx

.loop:

cmp rcx, rdi

jge .End

imul rsi, rax

inc rcx

jmp .loop

.End:

pop rcx

popf

ret

## Програма 2

bits 64

; list of system calls

; https://filippo.io/linux-syscall-table/

SYS\_READ equ 0

SYS\_WRITE equ 1

; Descriptors

STDIN equ 0

STDOUT equ 1

; ASCII characters

NULL\_TERMINATOR equ 0

NEW\_LINE\_CHARACTER equ 10

SPACE equ 32

PLUS\_SIGH equ 43

MINUS\_SIGN equ 45

PERIOD equ 46

DIGIT\_ZERO equ 48

DIGIT\_NINE equ 57

N\_LETTER equ 110

Y\_LETTER equ 121

; Other constants

BUFFER\_LENGTH equ 20

MAX\_LENGTH equ 10

MAX\_SIZE equ 10

; !!! SECTION DATA !!!

section .data

; Errors

error\_incorrect\_symbol: db "Incorrect symbol in input", NEW\_LINE\_CHARACTER, 0

error\_incorrect\_symbol\_length: equ $-error\_incorrect\_symbol

error\_sign\_character\_not\_first db "Sign characters must be first", NEW\_LINE\_CHARACTER, 0

error\_sign\_character\_not\_first\_length equ $-error\_sign\_character\_not\_first

error\_not\_positive db "Value is not positive", NEW\_LINE\_CHARACTER, 0

error\_not\_positive\_length equ $-error\_not\_positive

max\_length\_error db "Max length is 10", NEW\_LINE\_CHARACTER, 0

max\_length\_error\_length equ $-max\_length\_error

; Buffers

buffer: times BUFFER\_LENGTH db 0

inputtedLength: dq 0

; Messages

enterMatrixRows db "Enter number of rows: ", NEW\_LINE\_CHARACTER, 0

enterMatrixRowsLength equ $-enterMatrixRows

enterMatrixCols db "Enter number of cols: ", NEW\_LINE\_CHARACTER, 0

enterMatrixColsLength equ $-enterMatrixCols

enterMatrix db "Enter matrix: ", NEW\_LINE\_CHARACTER, 0

enterMatrixLength equ $-enterMatrix

enterMatrixElement db "Enter Matrix element ", 0

enterMatrixElementLength equ $-enterMatrixElement

findElement db "Do you want to find position of any element? y - yes, other - no", NEW\_LINE\_CHARACTER, 0

findElementLength equ $-findElement

enterFindElement db "Enter element: ", NEW\_LINE\_CHARACTER, 0

enterFindElementLength equ $-enterFindElement

positions db "Positions:", NEW\_LINE\_CHARACTER, 0

positionsLength equ $-positions

noPositionsFound db "No positions found", NEW\_LINE\_CHARACTER, 0

noPositionsFoundLength equ $-noPositionsFound

continue\_msg db "Continue? y - yes, other - no:", NEW\_LINE\_CHARACTER, 0

continue\_msg\_length equ $-continue\_msg

; !!! SECTION TEXT !!!

section .text

global asm\_main

;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

asm\_main:

push rax

push rdi

push rsi

push rdx

push rcx

push rbx

push r8

push r12

.loopDD:

mov rax, buffer

mov rdi, BUFFER\_LENGTH

.inputMatrixRows:

call PrintEnterMatrixRows

call InputArgumentPositive

push rsi

.inputMatrixCols:

call PrintEnterMatrixCols

call InputArgumentPositive

push rsi

.printEnterMatrix:

call PrintEnterMatrix

.inputMatrix:

; number of columns

pop rdx

; number of rows

pop rsi

; inputted value

xor rbx, rbx

; counter to zero

xor rcx, rcx

xor r8, r8

; the beggining of the Matrix

mov r12, rsp

sub r12, 8

.loopAllocateMemoryRow:

cmp rcx, rsi

jge .onEndAllocateMemoryRow

.loopAllocateMemoryColumn:

cmp r8, rdx

jge .onEndAllocateMemoryColumn

call PrintEnterMatrixElement

.printRowNumber:

push rdi

push rsi

push rdx

mov rsi, rdi

mov rdi, rax

mov rdx, rcx

call PrintInteger

pop rdx

pop rsi

pop rdi

call PrintSpace

.printColNumber:

push rdi

push rsi

push rdx

mov rsi, rdi

mov rdi, rax

mov rdx, r8

call PrintInteger

pop rdx

pop rsi

pop rdi

call PrintEndl

push rsi

call InputArgument

mov rbx, rsi

pop rsi

push rbx

inc r8

jmp .loopAllocateMemoryColumn

.onEndAllocateMemoryColumn:

xor r8, r8

inc rcx

; jump to the loop head

jmp .loopAllocateMemoryRow

.onEndAllocateMemoryRow:

xor rcx, rcx

xor r8, r8

xor rbx, rbx

; rax - buffer, rdi - bufferLength, rsi - rows, rdx - cols,

; rbx - 0, rcx - 0, r8 - 0, r12 - matrix

.callPrintMatrix:

push rax

push rdi

push rsi

push rdx

push r8

push r9

mov r9, rdx

mov r8, rsi

mov rdx, r12

mov rsi, rdi

mov rdi, rax

call PrintMatrix

pop r9

pop r8

pop rdx

pop rsi

pop rdi

pop rax

mov r8, r12

call FindElementRoutine

.deAllocateMatrix:

xor rcx, rcx

xor r8, r8

.loopDeAllocateRow:

cmp rcx, rsi

jge .onEndDeAllocateRow

.loopDeAllocateColumn:

cmp r8, rdx

jge .onEndDeallocateColumn

pop rax

inc r8

jmp .loopDeAllocateColumn

.onEndDeallocateColumn:

xor r8, r8

inc rcx

jmp .loopDeAllocateRow

.onEndDeAllocateRow:

mov rax, continue\_msg

mov rdi, continue\_msg\_length

call WriteToConsole

mov rax, buffer

mov rdi, BUFFER\_LENGTH

call ReadIntoBuffer

xor rbx, rbx

mov bl, byte [rax]

cmp bl, Y\_LETTER

je .loopDD

pop r12

pop r8

pop rbx

pop rcx

pop rdx

pop rsi

pop rdi

pop rax

ret

;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

PrintFindElement:

push rax

push rdi

mov rax, findElement

mov rdi, findElementLength

call WriteToConsole

pop rdi

pop rax

ret

;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

FindElementRoutine:

; void (rax - buffer, rdi - bufferLength, rsi - rows, rdx - cols, r8 - matrix)

push rbx

push rcx

push r9

push r10

push r12

push r13

push r14

.loopRoutine:

call PrintFindElement

push rsi

call ReadIntoBuffer

pop rsi

xor rbx, rbx

mov bl, byte [rax]

cmp bl, Y\_LETTER

jne .end

call ClearBuffer

call PrintEnterFindElement

push rsi

call InputArgument

mov rbx, rsi

pop rsi

call PrintPositions

mov r9, 8

mov r11, r8

xor r13, r13 ; is\_any\_found\_flag

xor rcx, rcx

xor r10, r10

.loopFindPositionsRows:

cmp rcx, rsi

jge .onEndFindPositionsRows

.loopFindPositionsCols:

cmp r10, rdx

jge .onEndFindPositionsCols

mov r14, qword [r11]

cmp r14, rbx

jne .switchToNextElement

mov r13, 1

.printRowNumber:

push rdi

push rsi

push rdx

mov rsi, rdi

mov rdi, rax

mov rdx, rcx

call PrintInteger

pop rdx

pop rsi

pop rdi

call PrintSpace

.printColNumber:

push rdi

push rsi

push rdx

mov rsi, rdi

mov rdi, rax

mov rdx, r10

call PrintInteger

pop rdx

pop rsi

pop rdi

call PrintEndl

.switchToNextElement:

sub r11, r9

inc r10

jmp .loopFindPositionsCols

.onEndFindPositionsCols:

xor r10, r10

inc rcx

jmp .loopFindPositionsRows

.onEndFindPositionsRows

cmp r13, 1

je .onEndRoutine

call PrintNoPositionsFound

.onEndRoutine

jmp .loopRoutine

.end:

pop r14

pop r13

pop r12

pop r10

pop r9

pop rcx

pop rbx

ret

;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

PrintNoPositionsFound:

push rax

push rdi

mov rax, noPositionsFound

mov rdi, noPositionsFoundLength

call WriteToConsole

pop rdi

pop rax

ret

;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

PrintPositions:

push rax

push rdi

mov rax, positions

mov rdi, positionsLength

call WriteToConsole

pop rdi

pop rax

ret

;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

PrintEnterFindElement:

push rax

push rdi

mov rax, enterFindElement

mov rdi, enterFindElementLength

call WriteToConsole

pop rdi

pop rax

ret

;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

PrintEnterMatrixElement:

push rax

push rdi

mov rax, enterMatrixElement

mov rdi, enterMatrixElementLength

call WriteToConsole

pop rdi

pop rax

ret

;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

PrintEnterMatrix:

push rax

push rdi

mov rax, enterMatrix

mov rdi, enterMatrixLength

call WriteToConsole

pop rdi

pop rax

ret

;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

PrintEnterMatrixRows:

push rax

push rdi

mov rax, enterMatrixRows

mov rdi, enterMatrixRowsLength

call WriteToConsole

pop rdi

pop rax

ret

;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

PrintEnterMatrixCols:

push rax

push rdi

mov rax, enterMatrixCols

mov rdi, enterMatrixColsLength

call WriteToConsole

pop rdi

pop rax

ret

;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

PrintMatrix:

; rax - void (rdi - buffer, rsi - bufferLength, rdx - matrix, r8 - rows, r9 - cols )

push rcx

push rbx

push r10

; zero counter

xor rcx, rcx

; mov the Matrix beginning to r9

mov r10, rdx

.calcRowSize:

; long int size

mov rbx, 8

push rax

mov rax, r9

imul rbx

mov rbx, rax

pop rax

.loop:

cmp rcx, r8

jge .end

.callPrintArray:

push rdx

push r8

mov rdx, r10

mov r8, r9

call PrintArray

pop r8

pop rdx

.callPrintEndl:

push rax

mov rax, rdi

call PrintEndl

pop rax

.switchToNextElement:

sub r10, rbx

inc rcx

jmp .loop

.end:

pop r10

pop rbx

pop rcx

ret

;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

PrintArray:

; rax - void (rdi - buffer, rsi - bufferLength, rdx - array, r8 - arrayLength )

push rcx

push rbx

push r8

push r9

; long int size

mov rbx, 8

; zero counter

xor rcx, rcx

; mov the Matrix beginning to r9

mov r9, rdx

.loop:

cmp rcx, r8

jge .end

.callPrintInteger:

push rdx

mov rdx, qword [r9]

call PrintInteger

pop rdx

.callPrintSpace:

push rax

mov rax, rdi

call PrintSpace

pop rax

.switchToNextElement:

sub r9, rbx

inc rcx

jmp .loop

.end:

pop r9

pop r8

pop rbx

pop rcx

ret

;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

PrintInteger:

; rax - void (rdi - buffer, rsi - bufferLength, rdx - number)

push rax

push rdi

push rsi

push rdx

push r8

mov rax, rdi

mov rdi, rsi

mov r8, rdi

xor rsi, rsi

call ClearBuffer

call TryConvertNumberToString

mov rdi, rsi

call WriteToConsole

mov rdi, r8

call ClearBuffer

pop r8

pop rdx

pop rsi

pop rdi

pop rax

ret

;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

GetElementAddressByIndex:

; rax - address (rdi - Matrix, rsi - index, rdx - typeSize)

push rbx

push rdx

push r8

mov r8, rdi

mov rbx, rdx

xor rdx, rdx

mov rax, rsi

imul rbx

sub r8, rax

mov rax, r8

pop r8

pop rdx

pop rbx

ret

;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

InputArgumentPositive:

; void (rax-buffer, rdi-bufferLength, rsi- int&\_out number)

.loopWhileNegative:

call InputArgument

cmp rsi, 0

jg .End

push rax

push rdi

mov rax, error\_not\_positive

mov rdi, error\_not\_positive\_length

call WriteToConsole

pop rdi

pop rax

jmp .loopWhileNegative

.End:

ret

InputArgument:

; void (rax-buffer, rdi-bufferLength, rsi- int&\_out number)

push rax

push rdi

push rdx

push r8

push r9

push r10

mov r9, rax

mov r10, rdi

xor rsi, rsi

.loop:

mov rax, r9

mov rdi, r10

call ReadIntoBuffer

call TryConvertStringToInteger

call ClearBuffer

cmp r8, 0

je .loop

call ClearBuffer

mov rsi, rdx

pop r10

pop r9

pop r8

pop rdx

pop rdi

pop rax

ret

PrintSpace:

; Function printing enl

; void PrintEndl(char\* const buffer);

; Params:

; rax: char\* buffer

; Returns:

; void

push rdi

push rbx

mov bl, byte [rax]

mov byte [rax], SPACE

mov rdi, 1

call WriteToConsole

mov byte [rax], bl

pop rbx

pop rdi

ret

;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

PrintEndl:

; Function printing enl

; void PrintEndl(char\* const buffer);

; Params:

; rax: char\* buffer

; Returns:

; void

push rdi

push rbx

mov bl, byte [rax]

mov byte [rax], NEW\_LINE\_CHARACTER

mov rdi, 1

call WriteToConsole

mov byte [rax], bl

pop rbx

pop rdi

ret

;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

PrintPlus:

; Function printing enl

; void PrintEndl(char\* const buffer);

; Params:

; rax: char\* buffer

; Returns:

; void

push rdi

push rbx

mov bl, byte [rax]

mov byte [rax], PLUS\_SIGH

mov rdi, 1

call WriteToConsole

mov byte [rax], bl

pop rbx

pop rdi

ret

PrintMinus:

; Function printing enl

; void PrintEndl(char\* const buffer);

; Params:

; rax: char\* buffer

; Returns:

; void

push rdi

push rbx

mov bl, byte [rax]

mov byte [rax], MINUS\_SIGN

mov rdi, 1

call WriteToConsole

mov byte [rax], bl

pop rbx

pop rdi

ret

ClearBuffer:

; Function clearing buffer

; void ClearBuffer(char\* buffer, int length);

; Params:

; rax: char\* buffer

; rdi: int length

; Returns:

; void

push rcx

xor rcx, rcx

.loop:

cmp rcx, rdi

jbe .End

mov byte [rax + rcx], 0

jmp .loop

.End:

pop rcx

ret

;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

;<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

TryConvertNumberToString:

; Function converting integer to string;

; bool TryConvertNumberToString(char\* buffer, int bufferLength, int inputtedLength, int number);

; Params:

; rax: char\* buffer

; rdi: int bufferLength

; rsi: int& inputtedLength

; rdx: int number

; Returns:

; r8: bool if true, no error, else throwed error

;

pushf

push rbx

push rcx

push r8

push r9

xor r9, r9

xor r8, r8

; counter = 0

mov rcx, 0

cmp rdx, 0

jge .ReadingNumbersIntoStack

.CheckForNegative:

mov r9, 1

neg rdx

mov byte [rax], MINUS\_SIGN

.ReadingNumbersIntoStack:

; The idea behind this is to read number into stack

; For example, 123 into stack like "3","2","1"

; and we counted digits. In this, example count = 3

; so we need to do smth like that:

; while(index<count) {

; pop stack into var

; var = var + ZERO\_CODE

; \*buffer[index] = var

; ++index

; }

; copy value to rbx

mov rbx, rdx

cmp rbx, 0

je .zero

.loop:

cmp rbx, 0

jle .ReadingNumbersFromStackToBuffer

.ReadDigit:

push rax

push rdx

; rax\_rbx\_copy = rbx;

mov rax, rbx

; rdx = 0; rbx = 10

xor rdx, rdx

mov rbx, 10

; rax\_rbx\_copy, rdx\_remaindex = rax\_rbx\_copy / rbx

idiv rbx

; r8 = rdx\_remainder; rbx = rax\_rbx\_copy

mov r8, rdx

mov rbx, rax

pop rdx

pop rax

push r8

inc rcx

jmp .loop

jmp .ReadingNumbersFromStackToBuffer

.zero:

push 0

inc rcx

.ReadingNumbersFromStackToBuffer:

xor rbx, rbx

xor r8, r8

cmp r9, 0

je .Preparation

.IncrementIfNegative:

inc r8

inc rcx

.Preparation:

mov rsi, rcx

.loop2:

cmp r8, rcx

jge .NoError

pop rbx

add rbx, DIGIT\_ZERO

mov [rax + r8], bl

inc r8

jmp .loop2

.NoError:

pop r9

pop r8

pop rcx

pop rbx

popf

ret

WriteToConsole:

; Function writing string into STDOUT

; void WriteToConsole(char\* buffer, int bufferLength)

; Params:

; rax: char\* buffer

; rdi: int bufferLength

; Returns:

; void

push rax

push rdi

push rsi

push rdx

mov rsi, rax

mov rdx, rdi

mov rax, SYS\_WRITE

mov rdi, STDOUT

call DoSystemCallNoModify

pop rdx

pop rsi

pop rdi

pop rax

ret

ReadIntoBuffer:

; Function reading string into buffer;

; void ReadIntoBuffer(char\* buffer, int bufferLength, int\* inputtedLength);

; Params:

; rax: char\* buffer

; rdi: int bufferLength

; rsi: int& inputtedLength

; Returns:

; void

push rax

push rdi

push rdx

push r8

push r9

mov r8, rax

mov r9, rdi

mov rsi, 0

.loop:

mov rsi, r8

mov rdx, r9

mov rax, SYS\_READ

mov rdi, STDIN

call DoSystemCallNoModify

cmp rax, MAX\_LENGTH

jle .NoError

mov rax, max\_length\_error

mov rdi, max\_length\_error\_length

call WriteToConsole

jmp .loop

.NoError:

mov rsi, rax

pop r9

pop r8

pop rdx

pop rdi

pop rax

ret

DoSystemCallNoModify:

; Function doing system call without

; modifying rcx and r11 registers after the call.

; type(rax) sys\_call(rax, rdi, rsi, rdx, r8, r9...);

; The reason behind this function is that in x64 NASM

; system call neither stores nor loads any registers

; it just uses and modifies them.

; https://stackoverflow.com/questions/47983371/why-do-x86-64-linux-system-calls-modify-rcx-and-what-does-the-value-mean

; http://www.int80h.org/bsdasm/#system-calls

; https://docs.freebsd.org/en/books/developers-handbook/x86/#x86-system-calls

pushf

push rcx

push r11

syscall

pop r11

pop rcx

popf

ret

TryConvertStringToInteger:

; Function converting string to integer;

; bool TryConvertStringToInteger(char\* buffer, int bufferLength, int inputtedLength, int\* number);

; Params:

; rax: char\* buffer

; rdi: int bufferLength

; rsi: int inputtedLength

; rdx: int& number

; Returns:

; r8: bool if true, no error, else throwed error

;

pushf

push rbx

push rcx

push r9

push r10

xor rdx, rdx

xor r10, r10

xor r9, r9

mov rdx, 0

mov rcx, rsi

xor rbx, rbx

xor r8, r8

dec rcx

cmp byte [rax + rcx], NEW\_LINE\_CHARACTER

jne .loop

dec rcx

.loop:

cmp rcx, 0

jl .NoError

mov bl, byte [rax + r9]

.IsNewLineCharacter:

cmp bl, NEW\_LINE\_CHARACTER

je .NoError

.CheckForSigns:

.IsPlusCharacter:

cmp bl, PLUS\_SIGH

je .CheckForSignBeingFirst

.IsMinusCharacter:

cmp bl, MINUS\_SIGN

je .OnEqualMinus

jmp .CallIsDigit

.OnEqualMinus:

mov r10, 1

.CheckForSignBeingFirst:

cmp r9, 0

jne .ErrorSignNotFirst

jmp .OnIterationEnd

.CallIsDigit:

push rax

push rdi

xor rax, rax

mov al, bl

call IsDigit

mov r8, rdi

pop rdi

pop rax

cmp r8, 0

je .ErrorIncorrectSymbol

sub bl, DIGIT\_ZERO

.CallPow:

push rax

push rdi

push rsi

mov rax, 10

mov rdi, rcx

call Pow

imul rsi, rbx

mov rdi, rdx

add rdi, rsi

mov rdx, rdi

pop rsi

pop rdi

pop rax

; whole\_digit = digit\*(10^counter)

.OnIterationEnd:

dec rcx

inc r9

jmp .loop

.ErrorSignNotFirst:

mov r8, 0

push rax

push rdi

mov rax, error\_sign\_character\_not\_first

mov rdi, error\_sign\_character\_not\_first\_length

call WriteToConsole

pop rdi

pop rax

jmp .End

.ErrorIncorrectSymbol:

; print error message

mov r8, 0

push rax

push rdi

mov rax, error\_incorrect\_symbol

mov rdi, error\_incorrect\_symbol\_length

call WriteToConsole

pop rdi

pop rax

jmp .End

.NoError:

cmp r10, 1

jne .GeneralNoError

push rax

mov rax, rdx

neg rax

mov rdx, rax

pop rax

.GeneralNoError:

mov r8, 1

jmp .End

.End:

pop r10

pop r9

pop rcx

pop rbx

popf

ret

IsDigit:

; Function checking whether byte value

; is in digit codes range

; bool IsDigit(char c);

; Params:

; rax: char c

; Returns:

; rdi: bool if true, then it is digit, else not

pushf

cmp al, DIGIT\_ZERO

jl .Invalid

cmp al, DIGIT\_NINE

jg .Invalid

mov rdi, 1

jmp .End

.Invalid:

mov rdi, 0

jmp .End

.End:

popf

ret

Pow:

; Function powing number to a certain degree.

; Params:

; rax: int number

; rdi: int degree

; Returns:

; rsi: Powed number

pushf

push rcx

mov rsi, 1

xor rcx, rcx

.loop:

cmp rcx, rdi

jge .End

imul rsi, rax

inc rcx

jmp .loop

.End:

pop rcx

popf

ret

# Схема функціонування програми

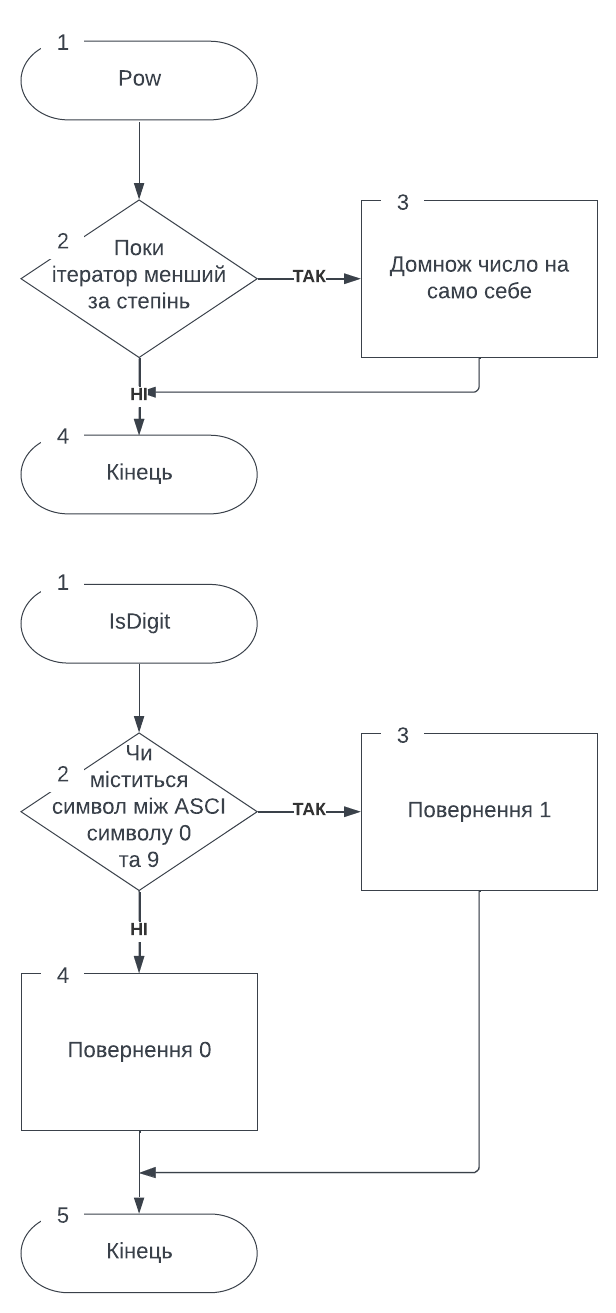


Рисунок 4.1 — схема функцій Pow та IsDigit

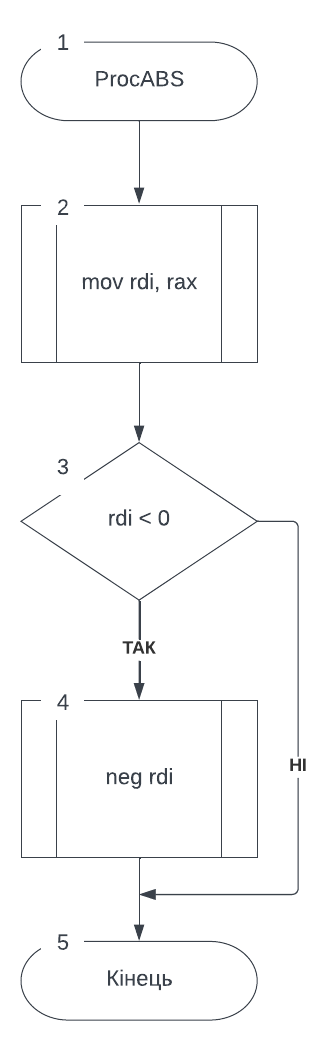


Рисунок 4.2 — схема функції ProcABS

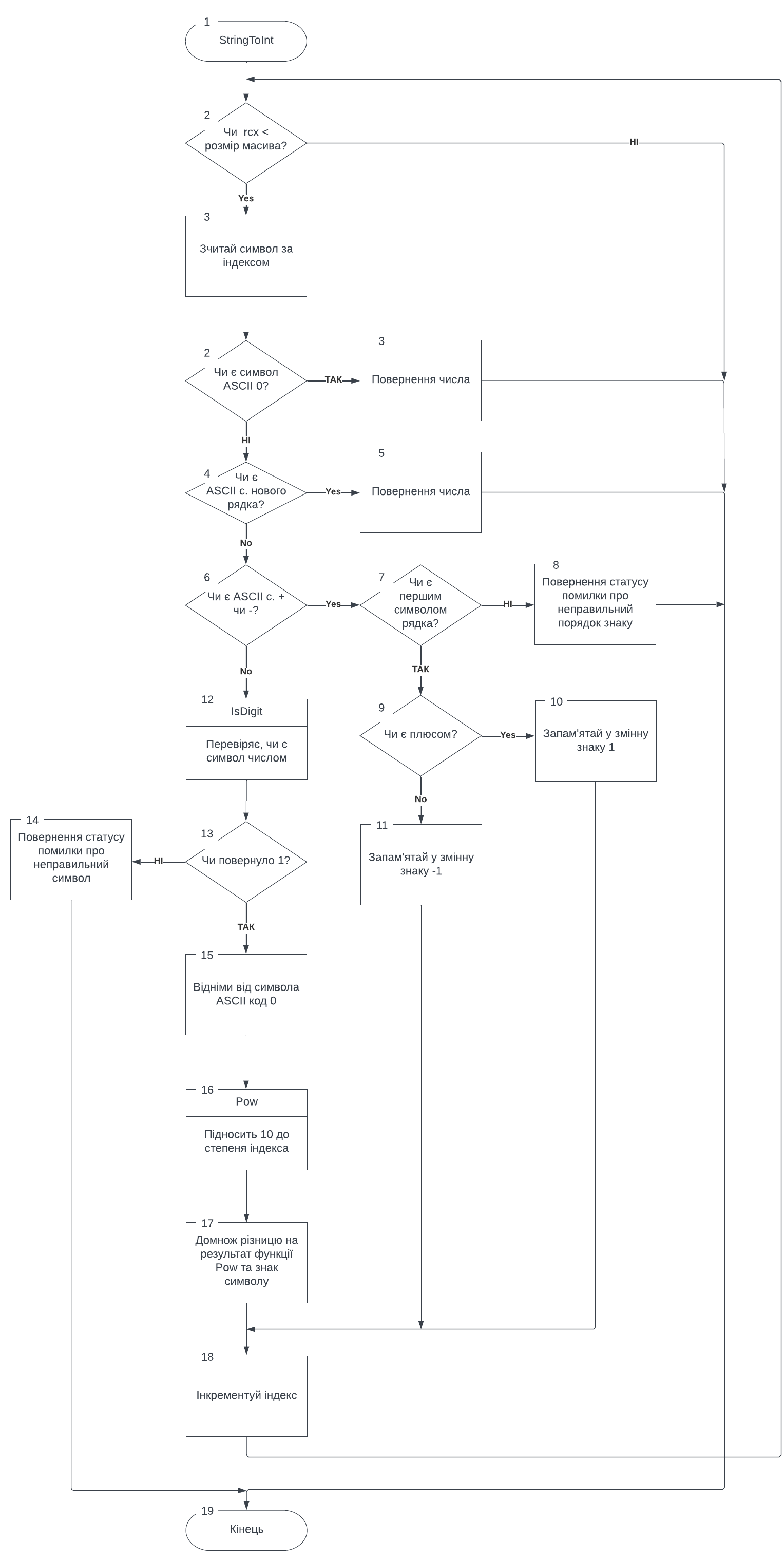


Рисунок 4.3 — схема функції StringToInt

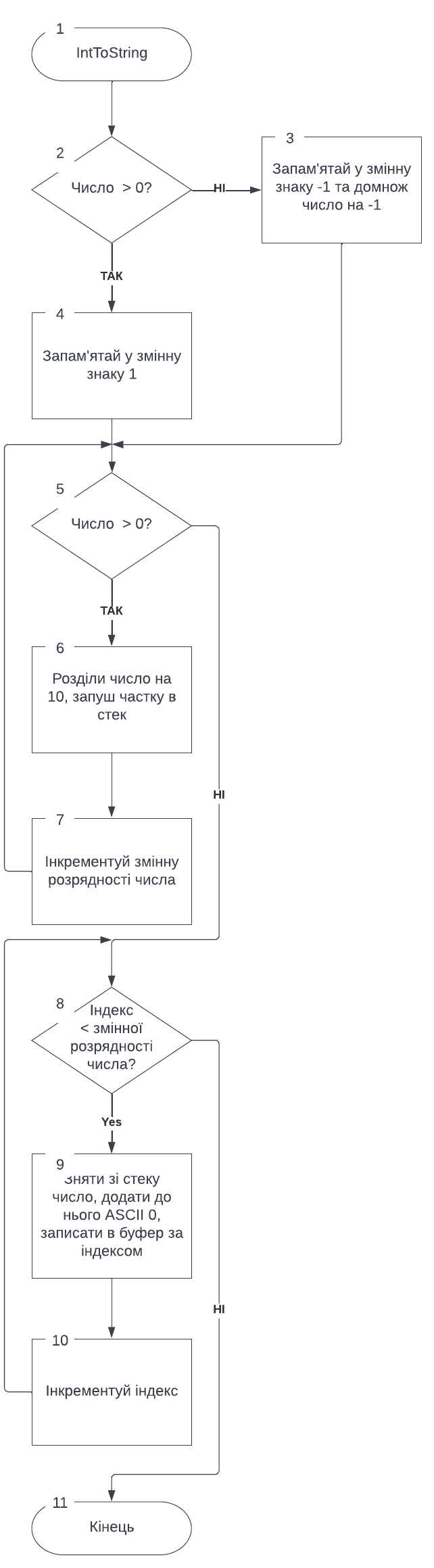


Рисунок 4.4 — схема функції IntToStr

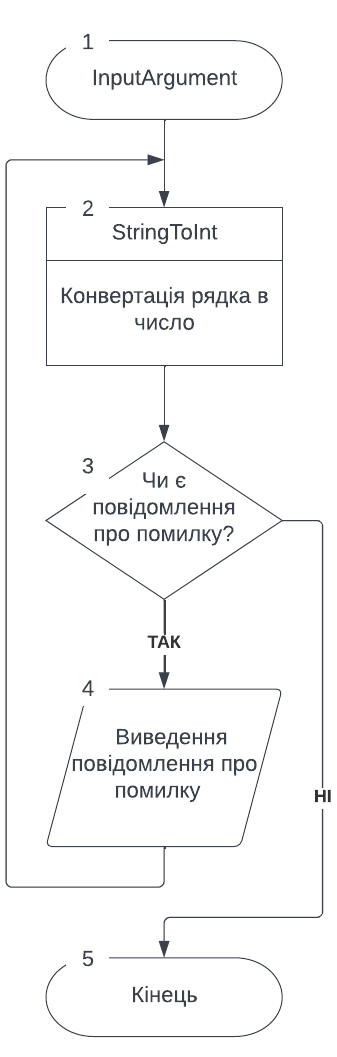


Рисунок 4.5 — схема функції InputArgument

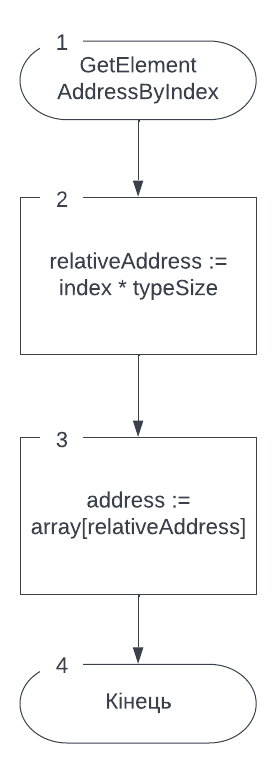


Рисунок 4.6 — схема функції GetElementAddressByIndex

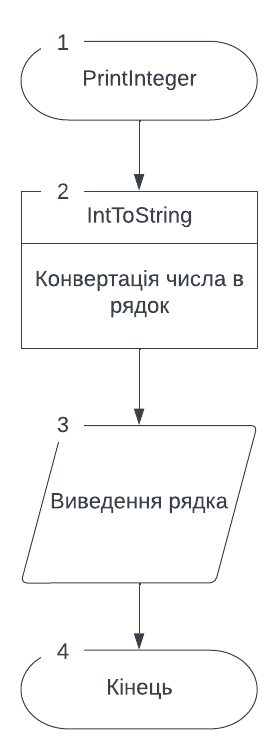


Рисунок 4.7— схема функції PrintInteger

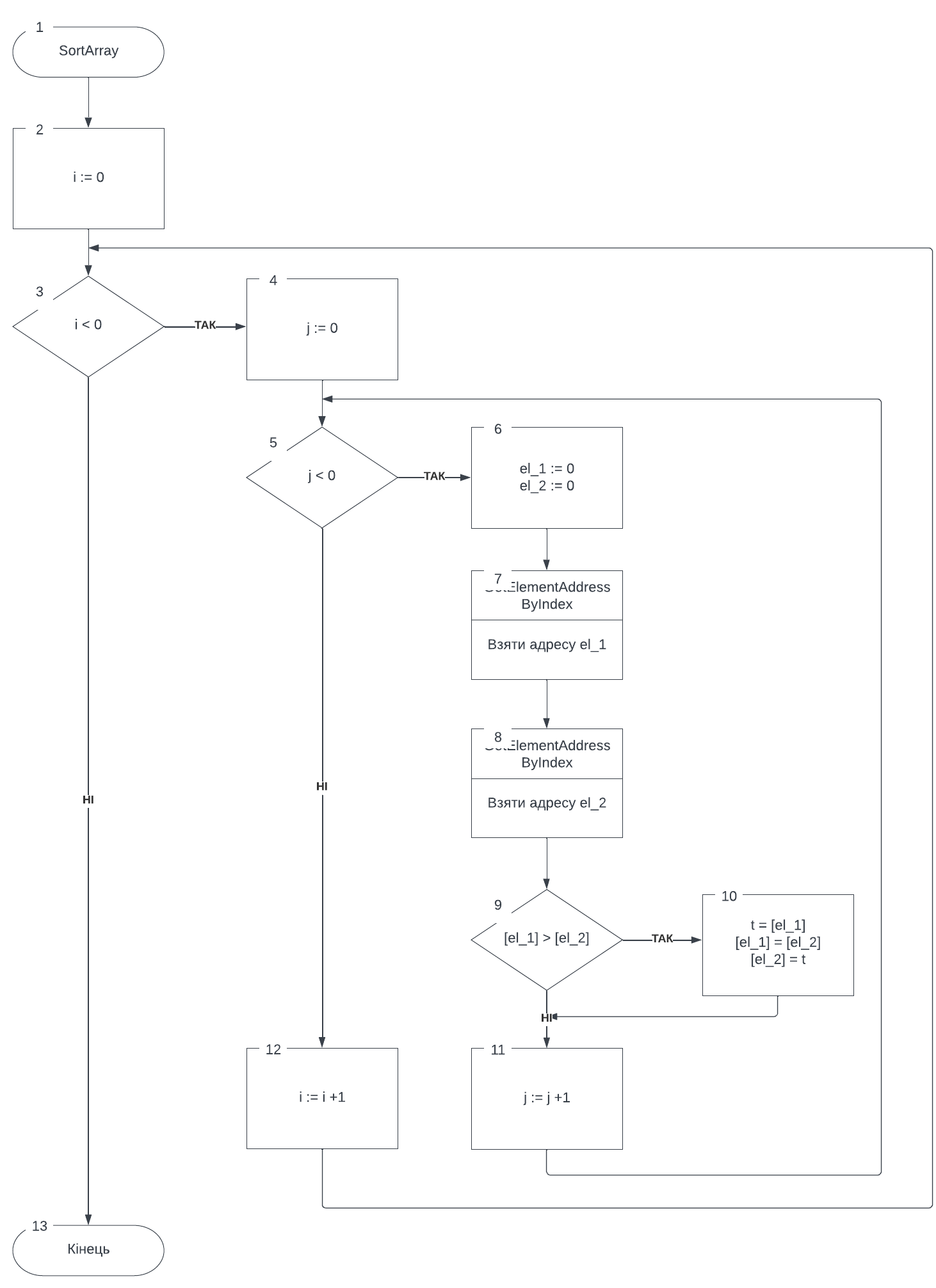


Рисунок 4.8— схема функції SortArray

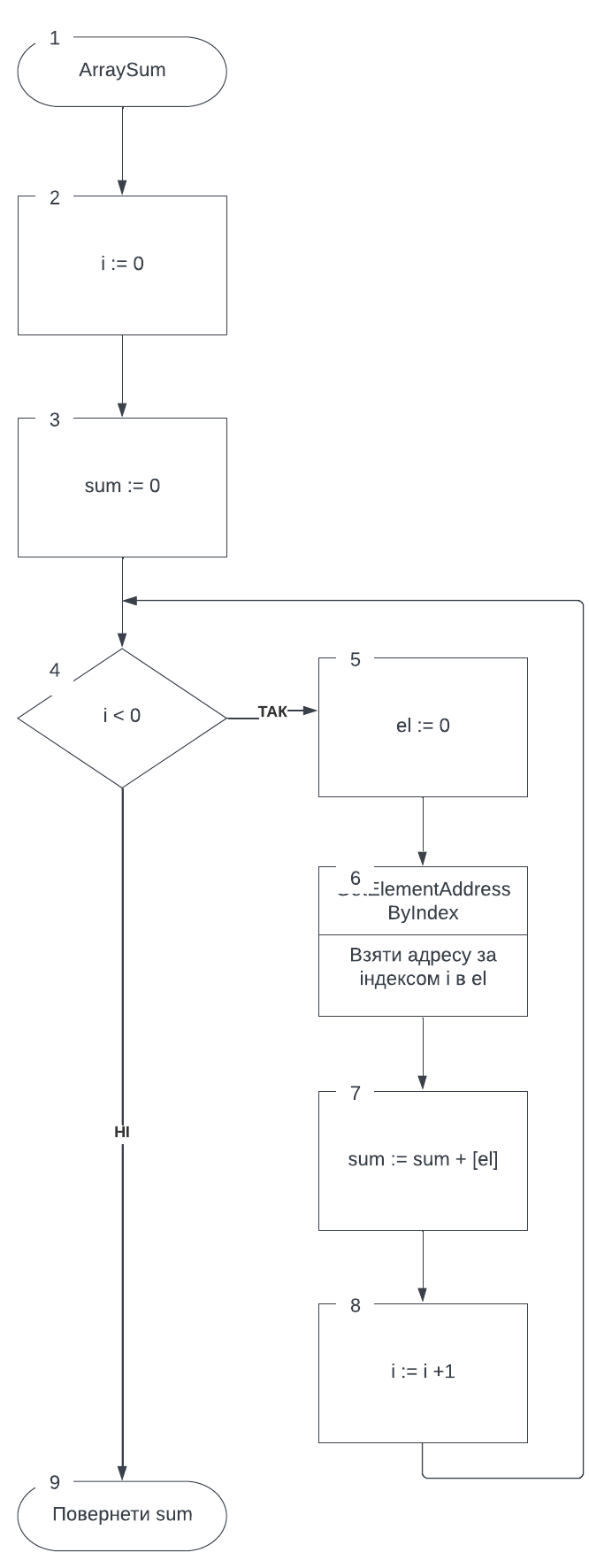


Рисунок 4.8— схема функції ArraySum

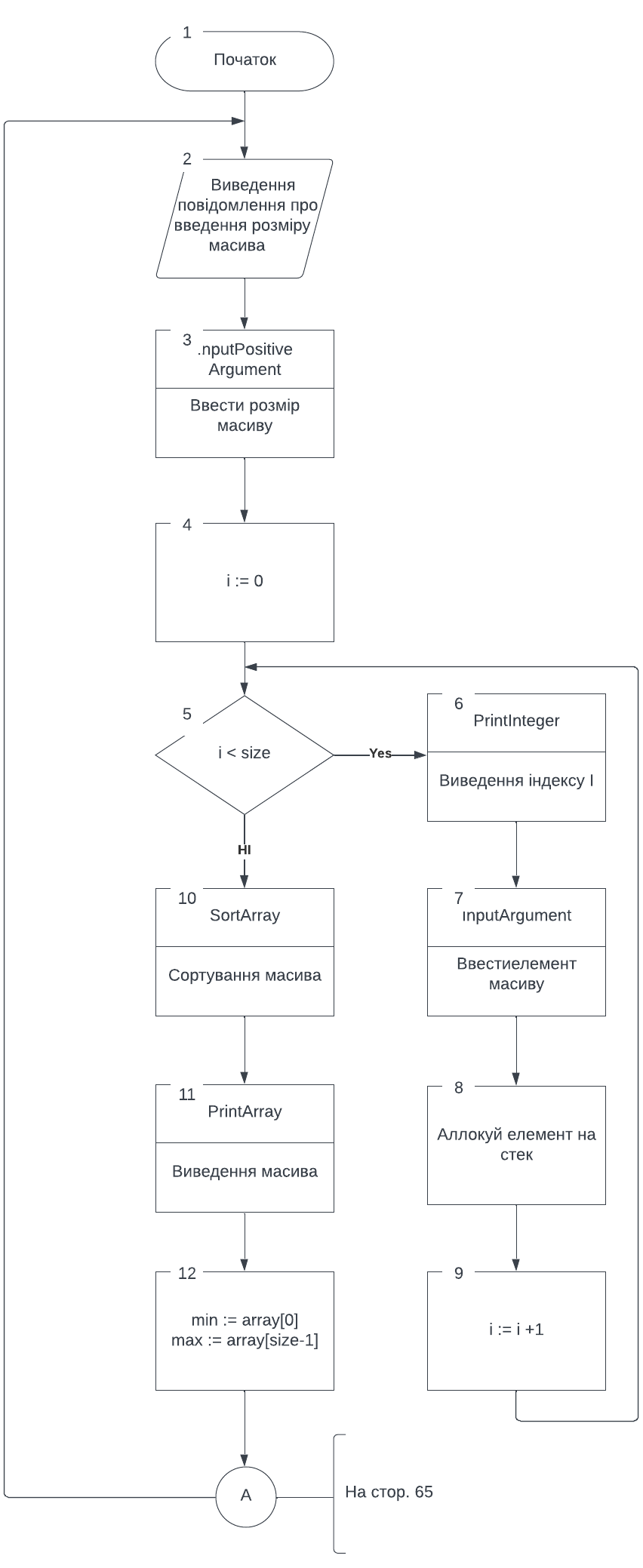


Рисунок 4.9— схема функції Програми 1 Main, частина 1

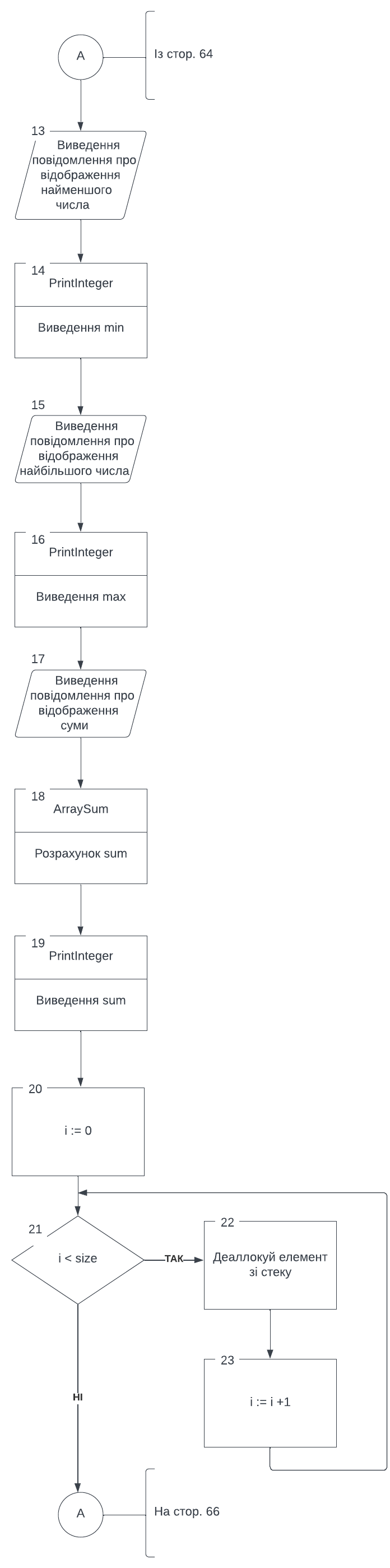


Рисунок 4.10— схема функції Програми 1 Main, частина 2

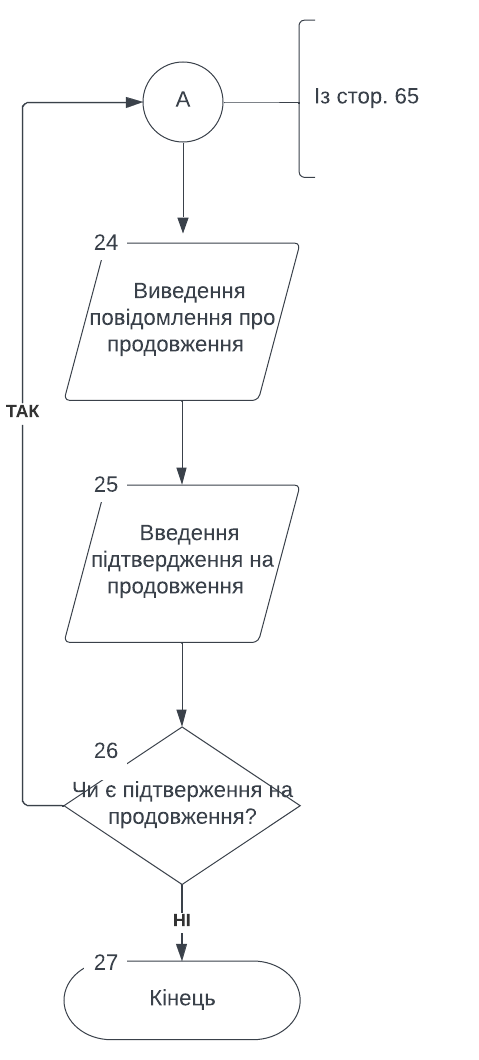


Рисунок 4.11— схема функції Програми 1 Main, частина 3

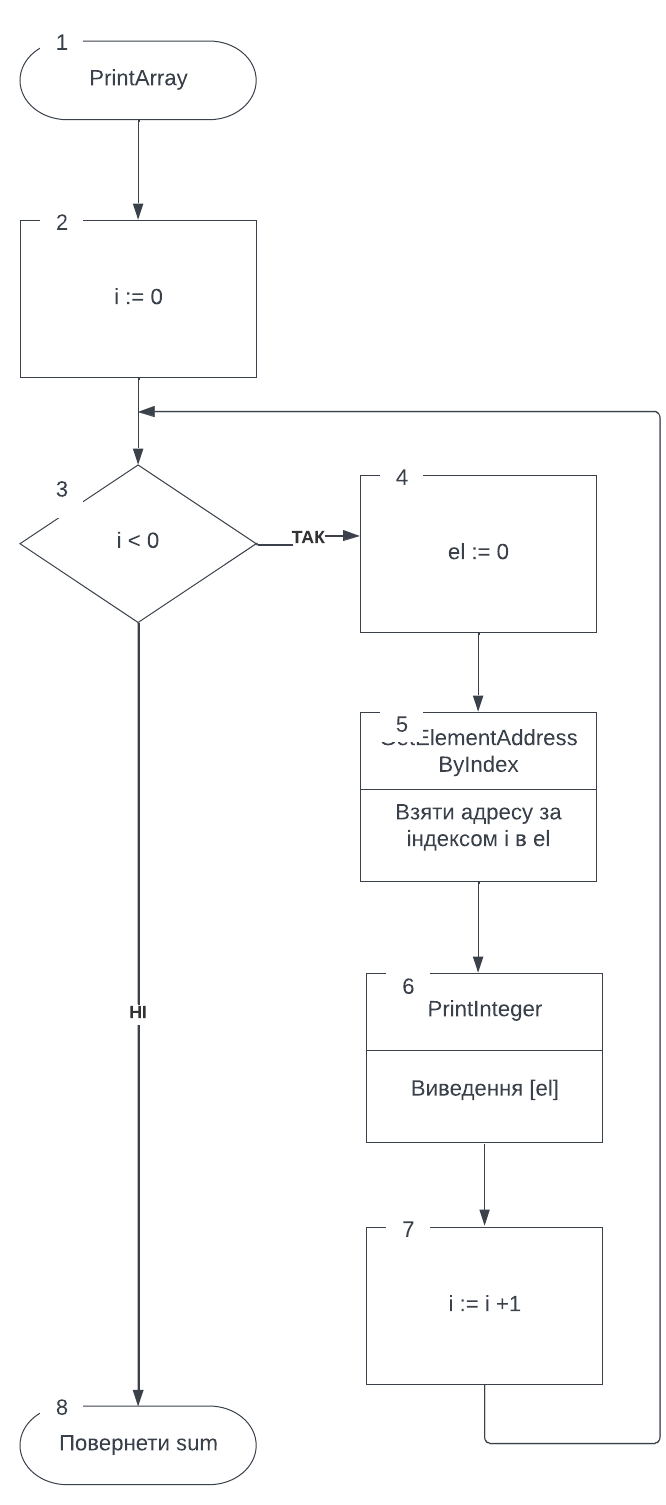


Рисунок 4.12— схема функції PrintArray



Рисунок 4.13— схема функції InputPositiveArgument

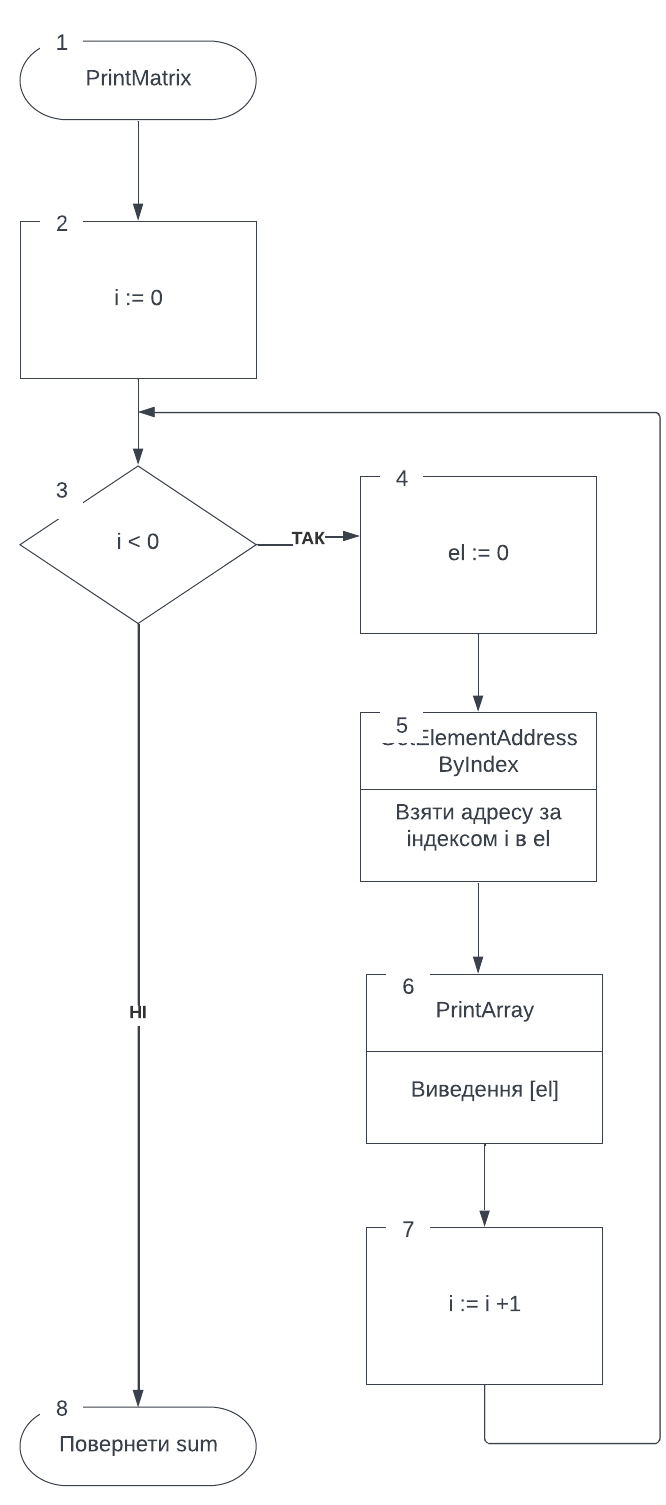


Рисунок 4.14— схема функції PrintMatrix

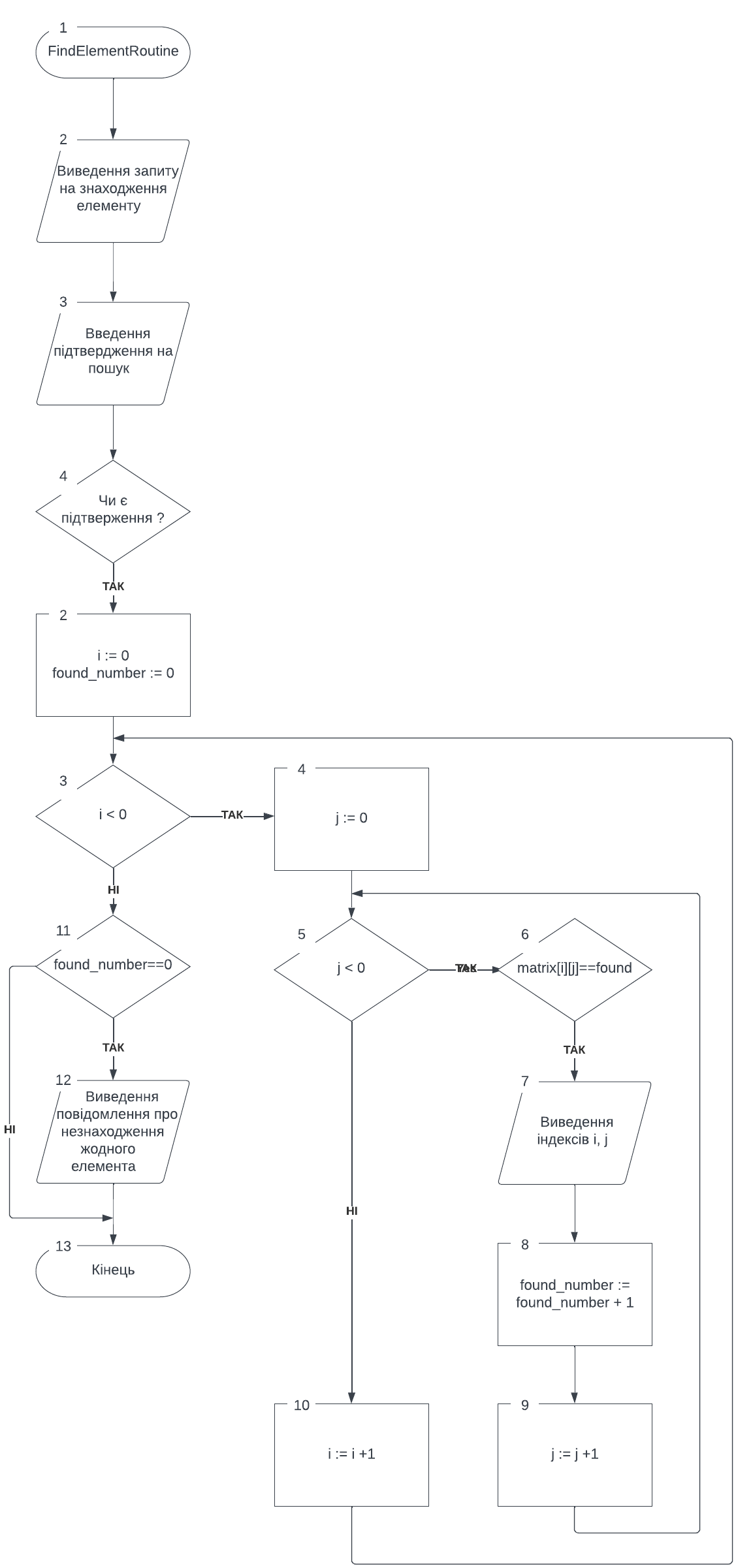


Рисунок 4.15— схема функції FindElementRoutine

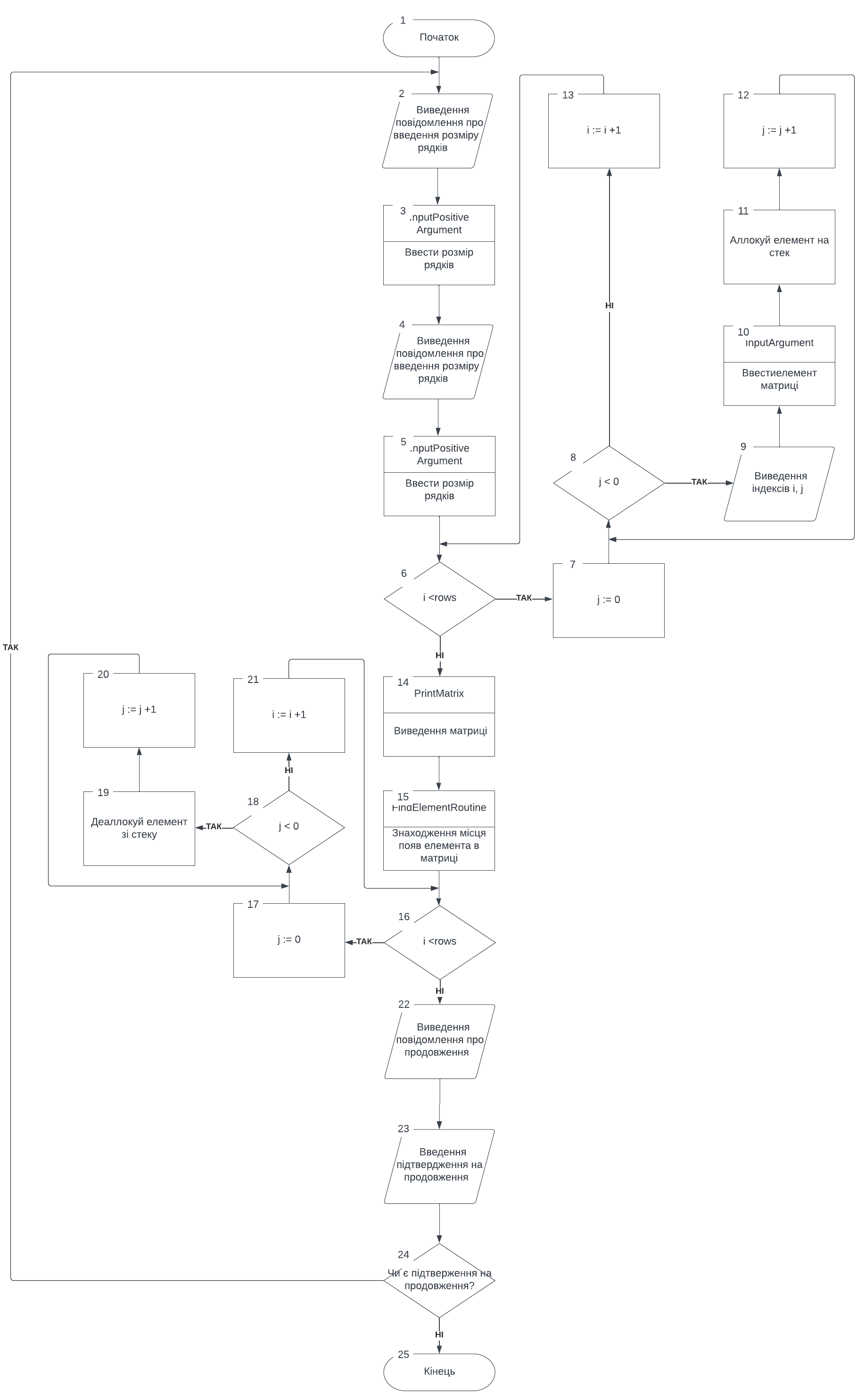


Рисунок 4.11— схема функції Програми 2 Main

# Приклад виконання

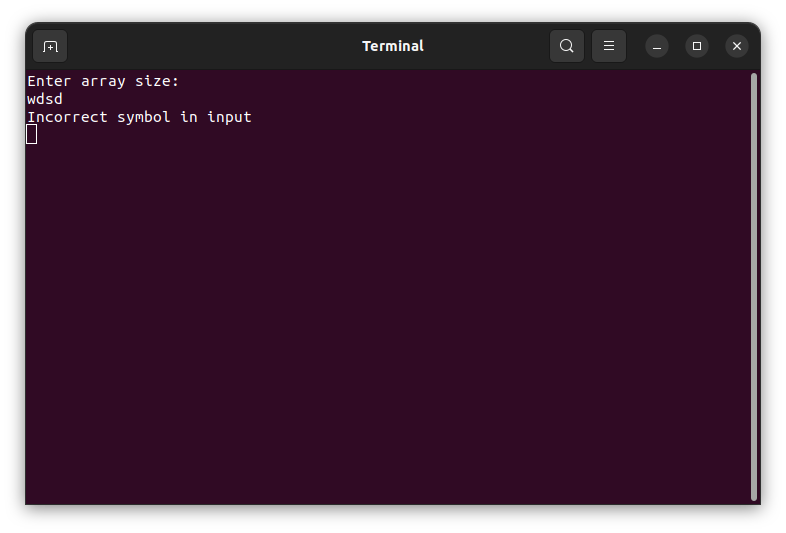


Рисунок 5.1— Програма 1: виведення повідомлення про неправильний символ

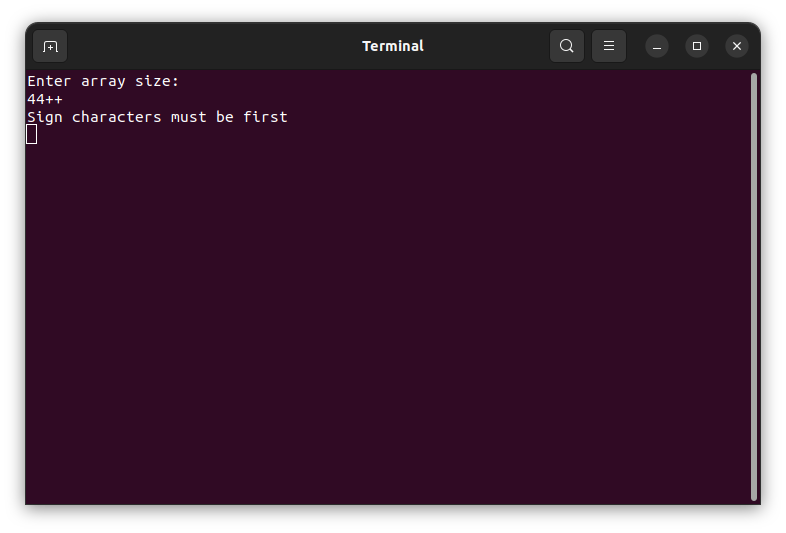


Рисунок 5.2— Програма 1: виведення повідомлення про неправильну позицію знака символу

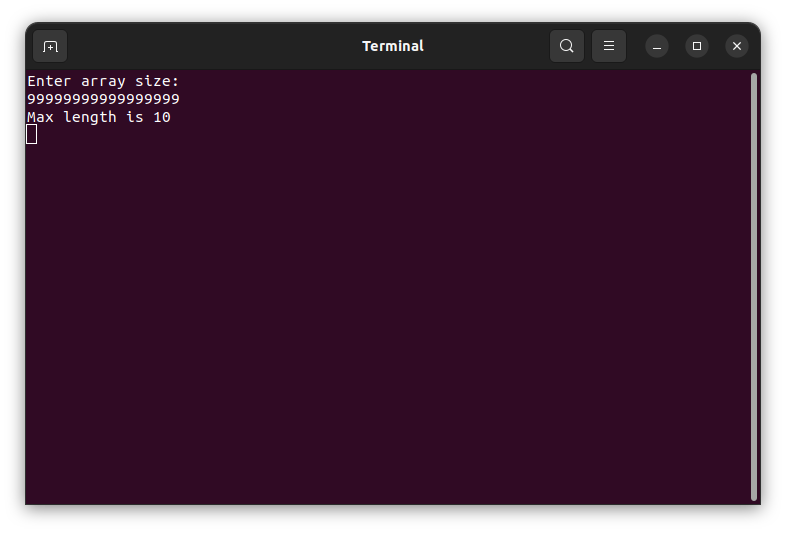


Рисунок 5.3— Програма 1: виведення повідомлення переповнення буфера

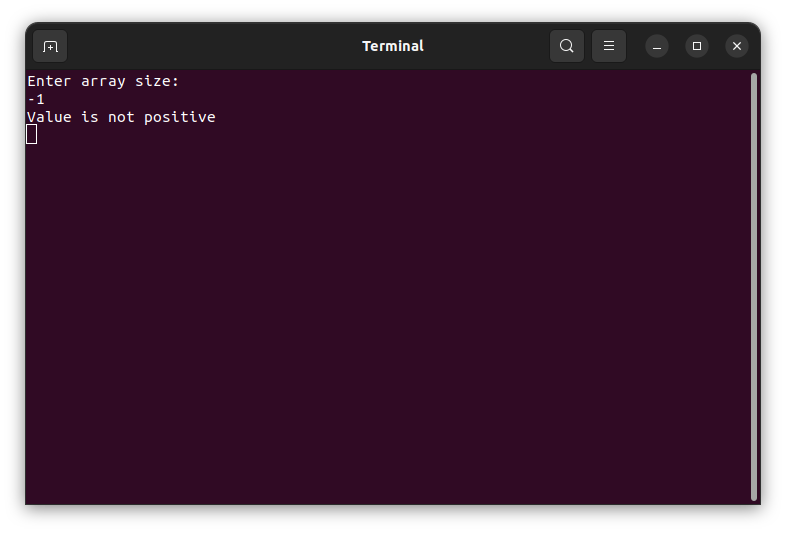


Рисунок 5.4— Програма 1: виведення повідомлення непозитивне введення розміру масиву

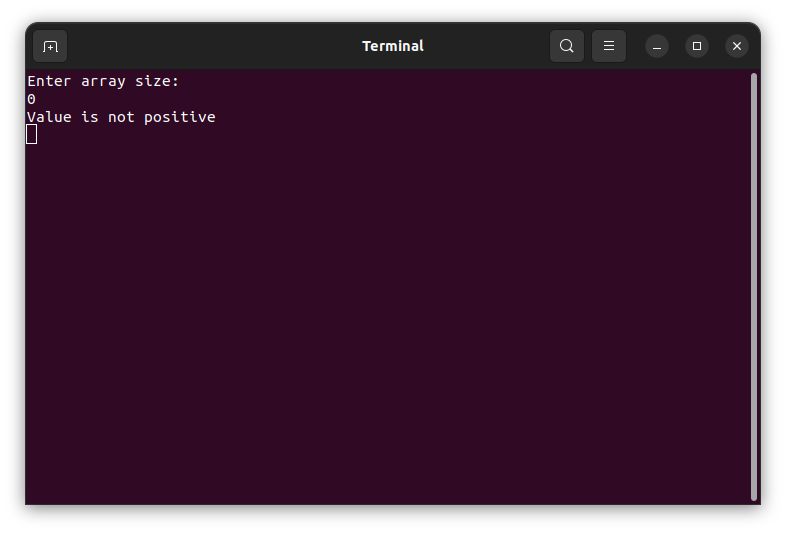


Рисунок 5.5— Програма 1: виведення повідомлення непозитивне введення розміру масиву

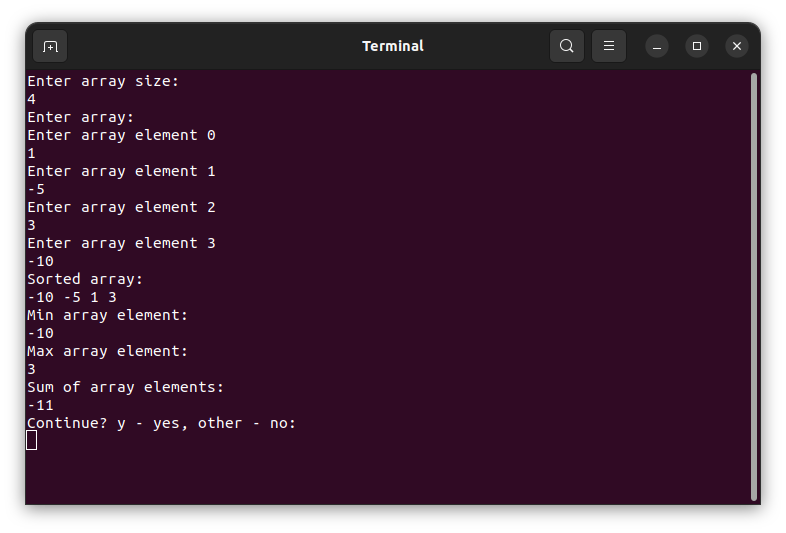


Рисунок 5.6— Програма 1: виведення сортованого масиву, максимального і мінімального значення, суми елементів

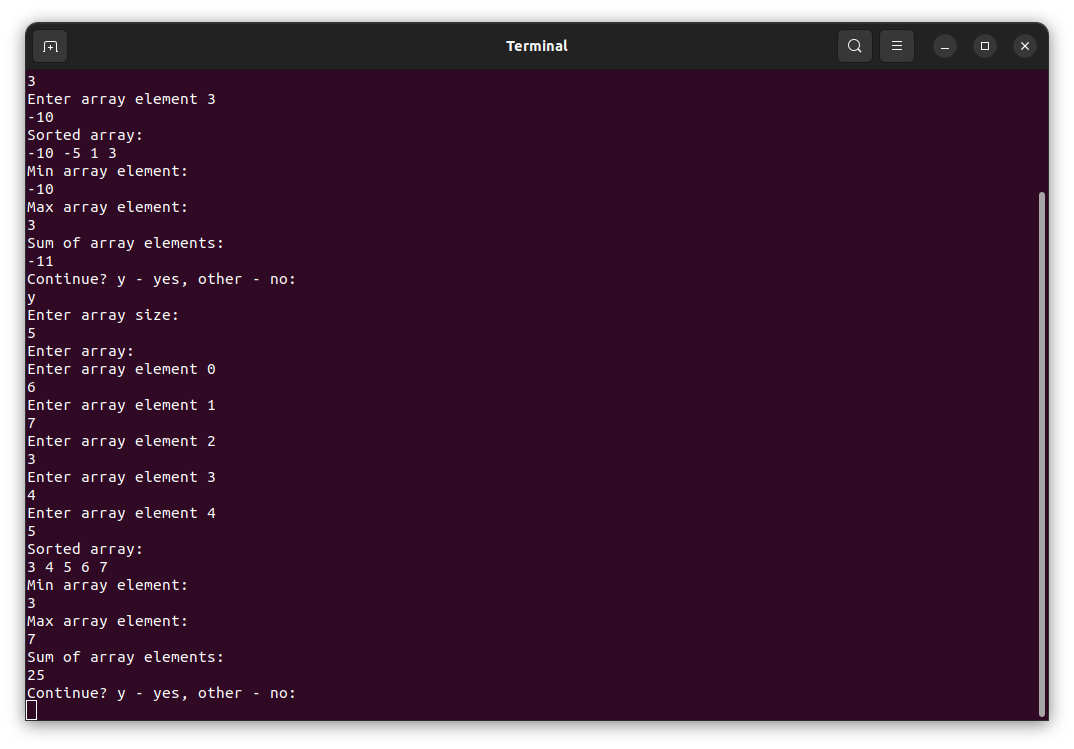


Рисунок 5.7— Програма 1: виведення повідомлення про запит на продовження роботи з програмою

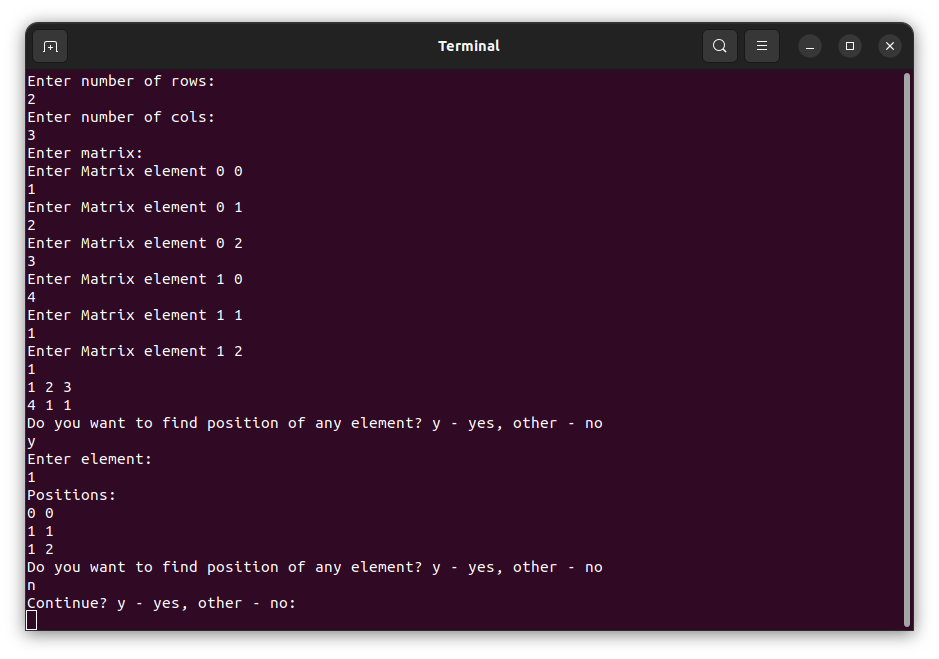


Рисунок 5.8— Програма 2: виведення масиву, знаходження елементу

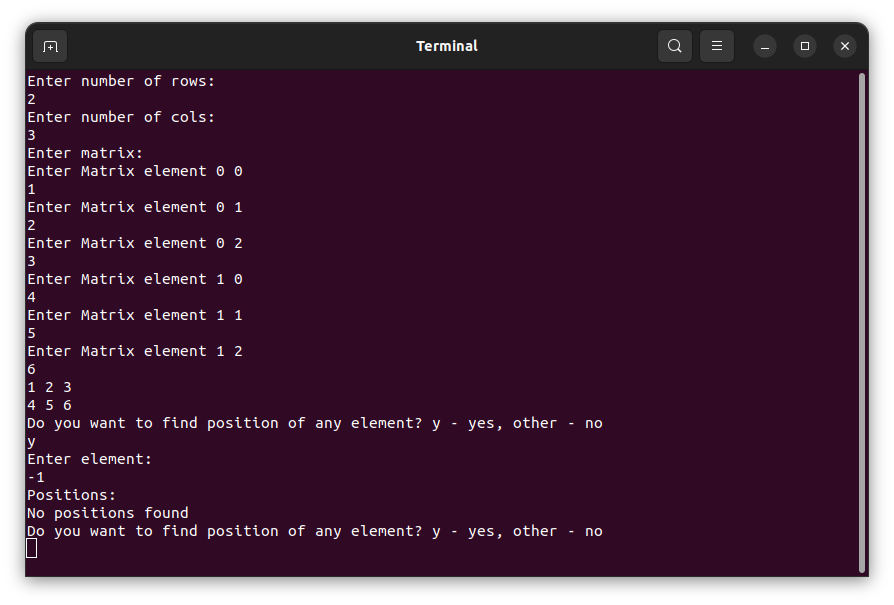


Рисунок 5.9— Програма 2: виведення повідомлення про відсутність елементу у двовимірному масиві

# Висновок

## Команди організації циклів

### JMP

Команда jmp означає перехід і використовується для переходу до іншої частини коду. Цю команду можна використовувати для створення циклів, повертаючись до попереднього пункту коду.

### CMP

Команда cmp використовується для порівняння двох значень. Його часто використовують у поєднанні з умовними переходами: JE, JNE, JZ, JNZ, JA, JAE, JB, JBE, JS, JNS, JO, JNO

### INC, DEC

Використовується для організації лічильників, які використовуються для обмеження кількості ітерацій

## Рядкові команди та особливості їх використання.

### LODS (завантажити рядок)

Ця інструкція завантажує байт, слово або подвійне слово з місця пам’яті, на яке вказує покажчик джерела, в акумулятор (AL, AX або EAX) і відповідно оновлює покажчик.

### STOS (зберігати рядок)

Ця інструкція зберігає байт, слово або подвійне слово з накопичувача (AL, AX або EAX) у місці пам’яті, на яке вказує вказівник призначення, і відповідно оновлює вказівник.

### REP (Повторення)

Ця префіксна інструкція використовується з рядковими інструкціями для повторення інструкції задану кількість разів.

### REPE або REPZ (Повторити при рівній кількості)

Ця префіксна інструкція використовується з рядковими інструкціями для повторення інструкції, коли встановлено нульовий прапор (ZF).

### REPNE або REPNZ (Повторити, якщо значення не дорівнює)

Ця префіксна інструкція використовується разом із рядковими інструкціями для повторення інструкції, коли прапор нуля (ZF) очищений.

### CMPS (рядок порівняння)

Ця інструкція порівнює байт, слово або подвійне слово з вихідного розташування з байтом, словом або подвійним словом з місця призначення та встановлює відповідні позначки.

### SCAS (сканування рядка)

Ця інструкція шукає в рядку вказане значення та встановлює відповідні позначки на основі результату.

### MOVSX (переміщення із розширенням знака)

Ця інструкція переміщує байт або слово з вихідного розташування до місця призначення та розширює значення за знаком до подвійно- або чотирислівного відповідно.

### MOVZX (Переміщення з нульовим розширенням)

Ця інструкція переміщує байт або слово з вихідного розташування до місця призначення та нульовим розширенням значення до подвіно- або чотирислівного відповідно.

### MOVS (переміщення рядка)

Ця інструкція переміщує байт, слово або подвійне слово з вихідного розташування до місця призначення та відповідно оновлює вказівники.

## Методи адресації за базою, з індексуванням, з подвійним індексуванням

### Індексована адресація

У цьому режимі регістр використовується як базова адреса, а зсув додається до базової адреси для обчислення адреси пам’яті операнда. Наприклад, MOV EAX, [EBX + 4] завантажує значення з місця пам’яті за адресою EBX + 4 у регістр EAX за допомогою індексованої адресації.

### Подвійна індексована адресація

Це поєднання режимів індексованої адресації та непрямої адресації. При подвійній індексованій адресації два регістри використовуються для обчислення адреси пам'яті операнда. Один регістр служить базовою адресою, а інший регістр служить індексом. Базову адресу та індекс додають разом, щоб обчислити адресу операнда.

### Адресація на основі

У цьому режимі сегментний регістр використовується як базова адреса, а зсув додається до базової адреси для обчислення адреси пам’яті операнда. Цей режим адресації зазвичай використовується в реальному режимі. Наприклад, MOV AX, [DS:0x1234] завантажує значення з місця пам’яті за адресою DS:0x1234 у регістр AX за допомогою адресації на основі.