**HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY**
THE SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY
ICT K67

**IT3283E**

COMPUTER ARCHITECTURE LAB


# COMPUTER ARCHITECTURE LAB

## PROJECT REPORT

**MEMBER LIST**

| | | |
|---|---|---|
| Nguyễn Đình Dương | 20225966 | duong.nd225966@sis.hust.edu.vn |
| Hồ Bảo Thư | 20226003 | thu.hb2260032@sis.hust.edu.vn |

**Teacher: Lê Bá Vui**

**Hà Nội — 2024**

# 1 Infix and Postfix Expressions

## 1.1 Problem Description

Infix notation is the standard way of writing arithmetic expressions, where operators are placed between operands (e.g., $A + B$). However, computers typically prefer postfix notation (also known as Reverse Polish Notation, or RPN), where operators are written after the operands (e.g., $AB+$).

The task involves:

- Reading an infix expression from the user.
- Converting the infix expression to postfix notation.
- Evaluating the postfix expression.
- Handling exceptions such as division by zero and invalid syntax.

## 1.2 Input and Output

**Input:** The input consists of a mathematical expression in infix notation, which may contain the following components:

- Operands: Integer between 0 and 99.
- Operators: $+, -, *, /, \%$ for addition, subtraction, multiplication, division, and division with remainder respectively.
- Parentheses: For grouping expressions.

**Output:** The program outputs the following:

- The input infix expression.
- The converted postfix expression.
- The result of evaluating the postfix expression.
- Error messages for invalid syntax or runtime errors (such as division by zero).

## 1.3 Exception Handling

Several exceptions are handled during the execution of the program:

- **Division by Zero:** If the user attempts to divide by zero, the program displays an error message and prompts the user to enter a new expression.

```
1     div_func:
2     # Kiem tra chia cho 0
3     beqz t5, division_by_zero
4     div t4, t4, t5
5     sw t4, 0(sp)
6     addi sp, sp, 4
7     addi s1, s1, 1
8     j loop_postfix
9
```

```
division_by_zero:
    la a0, msg_error3        # Prompt error
    li a7, 4
    ecall

    la a0, msg_enter
    li a7, 4
    ecall

    j input_infix
```

- **Invalid Parentheses:** If parentheses are misplaced or not properly balanced, the program will display an error message and prompt the user to enter a valid expression.

```
scan_parentheses:
    la t1, infix
    add t1, t1, s11
    lb t0, 0(t1)

    beqz t0, check_final_parentheses
    li t1, '\n'
    beq t0, t1, check_final_parentheses

    li t1, '('
    beq t0, t1, count_open
    li t1, ')'
    beq t0, t1, count_close

    addi s11, s11, 1
    j scan_parentheses

count_open:
    addi t6, t6, 1
    addi s11, s11, 1
    j scan_parentheses

count_close:
    addi t6, t6, -1
    bltz t6, parentheses_error  # If negative, we have too many
       closing parentheses
    addi s11, s11, 1
    j scan_parentheses

check_final_parentheses:
    bnez t6, parentheses_error  # If non-zero, we have unmatched
       parentheses
```

- **Invalid Syntax:** The program checks for invalid characters (such as non-digit characters or incorrect operators) and displays an error message if any are found.

```
    # Load first character
    la t0, infix
    lb t0, 0(t0)
    li t1, '-'
    beq t0, t1, lt0_error

```

```
    li t1, '/'
    beq t0, t1, lt0_error

    li t1, '+'
    beq t0, t1, lt0_error

    li t1, '*'
    beq t0, t1, lt0_error
```

```

# Check if character is digit
    li t1, '0'
    blt t0, t1, lt0_error
    li t1, '9'
    bgt t0, t1, lt0_error
```

- **Number Range Errors:** If a number greater than 99 or less than 0 is entered, the program will notify the user of the invalid input.

```
# Check if character is digit
    li t3, '0'
    bge t2, t3, check_digit
    j not_digit

check_digit:
    li t3, '9'
    ble t2, t3, continue

not_digit:
    li s3, ' '
    la t1, postfix
    add t1, t1, s1
    sb s3, 0(t1)
    addi s1, s1, 1
    j loop_infix

continue:
    addi t3, s0, 1
    la t1, infix
    add t1, t1, t3
    lb t4, 0(t1)

    # Check for numbers > 99
    li t1, '0'
    bge t4, t1, check_gt99
    j store_number

check_gt99:
    li t1, '9'
    ble t4, t1, gt99_error


# Various error handlers
gt99_error:
    la a0, msg_error1
```

```
37      li a7, 4
38      ecall
39      j input_infix
```

## 1.4 Algorithm Workflow

The algorithm for processing the infix expression involves three main stages:

**Stage 1: Scan parentheses**

In this stage, our program checks the balance of parentheses in the infix expression input by the user by counting the number of opening parentheses "(" and closing parentheses ")" and ensures that they are properly matched. If there are unmatched parentheses or the number of opening and closing parentheses does not match, the program will output an error message and prompt the user to re-enter the expression.

1. Traverse the infix expression character by character
2. For each opening/closing is encountered, the value of t6 is incremented/decremented by 1. Once t6 becomes negative, it means there are more closing than opening ones , and the program will trigger an error
3. After iterating through the entire expression, the program checks the value of t6. If t6 is not equal to 0, it indicates that there are unmatched parentheses, and an error message will be displayed.

**Stage 2: Conversion from Infix to Postfix**

1. Traverse the infix expression character by character
2. For each operand (number), append it to the postfix expression

```
1       la t1, postfix
2       add t1, t1, s1
3       sb t0, 0(t1)
4       addi s1, s1, 1
```

3. For each operator, pop operators from the stack to the postfix expression if their precedence is higher than the current operator. For example:

```
1       consider_plus_minus:
2       li t1, -1
3       beq s2, t1, push_op
4
5       la t1, stack
6       add t1, t1, s2
7       lb t5, 0(t1)     # Changed from t9 to t5
8
9       li t1, '('
10      beq t5, t1, push_op  # Changed from t9 to t5
11
12      la t1, stack
13      add t1, t1, s2
14      lb t1, 0(t1)
15
```

```
16      la t2, postfix
17      add t2, t2, s1
18      sb t1, 0(t2)
19
20      addi s2, s2, -1
21      addi s1, s1, 1
22
23      li s3, ' '
24      la t1, postfix
25      add t1, t1, s1
26      sb s3, 0(t1)
27      addi s1, s1, 1
28
29      j consider_plus_minus
30
31  push_op:
32      addi s2, s2, 1
33      la t1, stack
34      add t1, t1, s2
35      sb t0, 0(t1)
36      addi s0, s0, 1
37      j loop_infix
```

4. Handle parentheses: push '(' onto the stack and pop operators to the postfix expression when encountering ')'.

```
1           consider_rpar1:
2       beqz a3, invalid_parentheses
3       addi a3, a3, -1
4       j consider_rpar2
5
6   consider_rpar2:
7       la t1, stack
8       add t1, t1, s2
9       lb t1, 0(t1)
10
11      li t2, '('
12      beq t1, t2, remove_lpar #remove the parathese
13
14      la t2, postfix
15      add t2, t2, s1
16      sb t1, 0(t2)
17
18      addi s2, s2, -1
19      addi s1, s1, 1
20
21      li s3, ' '
22      la t1, postfix
23      add t1, t1, s1
24      sb s3, 0(t1)
25      addi s1, s1, 1
26
27      j consider_rpar2
```

5. Once the entire infix expression is processed, pop any remaining operators from the

stack to the postfix expression.

**Stage 3: Evaluation of Postfix Expression**

1. Traverse the postfix expression.
2. For each operand, it builds a number.

```
build_number:
# Convert character to number and add to s4
addi t0, t0, -48          # Convert ASCII to number
li t1, 10
mul s4, s4, t1            # Multiply current value by 10
add s4, s4, t0            # Add new digit
addi s1, s1, 1
j loop_postfix
handle_number:
# Convert character to number and push to stack
addi t0, t0, -48
sw t0, 0(sp)
addi sp, sp, 4
addi s1, s1, 1
j loop_postfix
```

3. For each operator, pop the required operands from the stack, perform the operation, and push the result back onto the stack.

```
 check_operator:
# Pop two numbers and perform operation
lw t4, -8(sp)             # First number
lw t5, -4(sp)             # Second number
addi sp, sp, -8
li t1, '+'
beq t0, t1, add_func
li t1, '-'
beq t0, t1, sub_func
li t1, '*'
beq t0, t1, mul_func
li t1, '/'
beq t0, t1, div_func
li t1, '%'
beq t0, t1, mod_func
addi s1, s1, 1
j loop_postfix
```

4. Once the entire postfix expression is evaluated, the result will be the final value on the stack.

The program utilizes a stack-based approach to handle operators and operands during both the conversion and evaluation phases. The precedence and associativity of operators are considered when determining the order of operations during conversion.

```
Please enter an infix expression: 90 / ( 2-2 )
Infix expression: 90 / ( 2-2 )
Postfix expression: 90 2 2 - /
ERROR: You enter a divisor that equal 0.

Please enter an infix expression: 90 / ha
Infix expression: 90 / ha
ERROR: You do not enter an appropriate operation. Please try again!
Please enter an infix expression: 1000 + 90
Infix expression: 1000 + 90
You enter a number that greater than 99. Please try again!
Please enter an infix expression: ( 90 / 2
ERROR: You enter a bracket that has wrong position.Please try again!
Please enter an infix expression: 90 + 2 =
Infix expression: 90 + 2 =
ERROR: You do not enter an appropriate operation. Please try again!
Please enter an infix expression: 90 + ( 2*2) /2
Infix expression: 90 + ( 2*2) /2
Postfix expression: 90 2 2 * 2 / +
Result of the expression: 92
```

Figure 1: Infix and postfix expression

### 1.5 Results

### 1.6 Source code

GitHub

## 2 Testing Sorting Algorithms

### 2.1 Problem Description

The task involves testing sorting algorithms with specific requirements:

- Research system calls to open and read data from a text file.
- Text files contain random numbers separated by spaces, with a maximum of 10,000 elements.
- The program allows the user to input a file name, reads the numbers in the file, and saves them to memory.
- The user can choose a sorting algorithm to execute from the options: Bubble Sort, Insertion Sort, or Selection Sort.
- The program runs the selected algorithm and measures its execution time using the TimerTool.
- The program writes the sorted results to an output file.

## 2.2 Input and Output

**Input:**

- A text file containing random integers separated by spaces.
- The file name is entered by the user.
- User selects the sorting algorithm (Bubble Sort, Insertion Sort, or Selection Sort).

**Output:**

- Execution time of the chosen sorting algorithm.
- A result file containing the sorted numbers.

## 2.3 Exception Handling

- **File open error:** If the file cannot be opened (system call returns a negative value), the program jumps to the "file_error" label and prints an error message.

```
1     open_file:
2     li a7, 1024
3     la a0, filename
4     li a1, 0
5     ecall
6     bltz a0, file_error
7     la t1, fd
8     sw a0, 0(t1)
9     jal read_numbers
```

- **Handling negative numbers:** After sorting, the program identify and flag negative numbers using a bitmask. If a number is non-negative, it is skipped in the marking process. Otherwise, Byte and bit position calculations ensure that array indices remain valid
- **Invalid path exception:** When the program attempt to open the file for writing and the operation fails, it will throw an error message to notify the user

```
1     # Open results.txt for writing
2     li a7, 1024       # Open file syscall
3     la a0, outfile    # Filename
4     li a1, 1          # Write-only flag
5     li a2, 0x1ff      # File permissions (777 in octal)
6     ecall
7
8     # Check if file opened successfully
9     bltz a0, write_error
10    sw a0, out_fd, t0 # Save file descriptor
```

```
1     # Check if minus sign
2     li t3, 45          # ASCII for '-'
3     bne t2, t3, not_minus
4     beqz t1, set_negative   # Only set negative if at start of number
5     j read_loop
6
7 set_negative:
```

```
8       li t6, 1            # Set negative flag
9       li t1, 1            # Set in-number flag
10      j read_loop
```

## 2.4 Algorithm Workflow

**Stage 1: Preprocessing**

1. Open file and read data. Data is read character by character, and ASCII values are converted to integers. Special characters such as space (' ') and newline (") indicate the end of a number. The program handles negative numbers by checking for the "-" character.
2. After successfully reading and converting the data, numbers are stored in the numbers array. The variable count keeps track of the total number of elements stored.
3. Mark negative numbers with a bitmask: Each bit in the bitmask corresponds to one element in the numbers array. If a number is negative, its corresponding bit is set to 1. The byte and bit positions are calculated as follows:
   - **Byte offset**: `index / 8`
   - **Bit position**: `index % 8`

```
1  mark_negatives:
2      # a0 = array address
3      # a1 = size
4      addi sp, sp, -16
5      sw ra, 12(sp)
6      sw s0, 8(sp)
7      sw s1, 4(sp)
8      sw s2, 0(sp)
9
10     mv s0, a0           # s0 = array address
11     mv s1, a1           # s1 = size
12     li s2, 0            # s2 = counter
13
14 mark_loop:
15     bge s2, s1, mark_done
16
17     # Load current number
18     slli t0, s2, 2      # t0 = counter * 4
19     add t0, s0, t0
20     lw t1, 0(t0)       # Load number
21
22     # Skip if positive
23     bgez t1, skip_mark
24
25     # Calculate byte and bit position in bitmask
26     mv t0, s2           # Copy index
27     srai t1, t0, 3      # Byte offset = index / 8
28     andi t2, t0, 0x7    # Bit position = index % 8
29     li t3, 1
30     sll t3, t3, t2      # Shift 1 to correct bit position
31
```

```
    # Set bit in bitmask
    la t4, neg_mask
    add t4, t4, t1      # Add byte offset
    lb t5, 0(t4)        # Load current byte
    or t5, t5, t3       # Set bit
    sb t5, 0(t4)        # Store updated byte

skip_mark:
    addi s2, s2, 1
    j mark_loop

mark_done:
    lw ra, 12(sp)
    lw s0, 8(sp)
    lw s1, 4(sp)
    lw s2, 0(sp)
    addi sp, sp, 16
    ret
```

**Stage 2.1: Quick sort**

1. Loads the array numbers and calculates the total element count. The sorting function is called with the array's base address, left = 0, and right = count - 1.
2. The partition function rearranges the array around the pivot:
    - Elements greater than the pivot are moved to the right.
    - Elements smaller than the pivot are moved to the left.

```
    partition_loop:
bge t1, s2, partition_done

# Load current element
slli t0, t1, 2
add t0, s0, t0
lw t2, 0(t0)      # t2 = arr[j]

# Compare with pivot
bgt t2, s3, skip_swap

# Increment i
addi s4, s4, 1

# Swap elements if i != j
slli t0, s4, 2
add t0, s0, t0    # Address of arr[i]
slli t3, t1, 2
add t3, s0, t3    # Address of arr[j]

lw t4, 0(t0)      # t4 = arr[i]
lw t5, 0(t3)      # t5 = arr[j]
sw t5, 0(t0)      # arr[i] = arr[j]
sw t4, 0(t3)      # arr[j] = arr[i]

```

```
skip_swap:
    addi t1, t1, 1    # j++
    j partition_loop

partition_done:
    # Place pivot in its final position
    addi s4, s4, 1    # i++

    # Swap pivot with element at i
    slli t0, s4, 2
    add t0, s0, t0    # Address of arr[i]
    slli t1, s2, 2
    add t1, s0, t1    # Address of arr[right]

    lw t2, 0(t0)      # t2 = arr[i]
    lw t3, 0(t1)      # t3 = arr[right]
    sw t3, 0(t0)      # arr[i] = arr[right]
    sw t2, 0(t1)      # arr[right] = arr[i]

    # Return pivot index
    mv a0, s4

    lw ra, 20(sp)
    lw s0, 16(sp)
    lw s1, 12(sp)
    lw s2, 8(sp)
    lw s3, 4(sp)
    lw s4, 0(sp)
    addi sp, sp, 24
    ret
```

3. Quick Sort is applied recursively to the left and right subarrays until the base case (left >= right) is reached.
4. After sorting, the program calls "mark_negatives" to set bits in a bitmask for all negative elements in the array.
5. The program calculates the execution time and writes the sorted array to an output file.

**Stage 2.2: Bubble sort**

1. The function passes the array address (numbers) and the array size (count) to "bubble_sort_impl" for sorting.
2. Repeatedly compare adjacent elements in the array. Swap them if they are in the wrong order. This process continues until the array is fully sorted.

```
    outer_loop_bubble:
    bge s2, s1, bubble_done
    li t0, 0            # j = 0

inner_loop_bubble:
    sub t1, s1, s2
    addi t1, t1, -1
    bge t0, t1, inner_done_bubble

```

```
    # Compare adjacent elements
    slli t2, t0, 2
    add t2, s0, t2
    lw t3, 0(t2)        # arr[j]
    lw t4, 4(t2)        # arr[j+1]

    ble t3, t4, no_swap_bubble

    # Swap elements
    sw t4, 0(t2)
    sw t3, 4(t2)

no_swap_bubble:
    addi t0, t0, 1
    j inner_loop_bubble

inner_done_bubble:
    addi s2, s2, 1
    j outer_loop_bubble

bubble_done:
    lw ra, 12(sp)
    lw s0, 8(sp)
    lw s1, 4(sp)
    lw s2, 0(sp)
    addi sp, sp, 16
    ret
```

3. After sorting, the program calls mark_negatives to set bits in a bitmask for all negative elements in the array.
4. The program calculates the execution time and writes the sorted array to an output file.

**Stage 2.3: Insertion sort**

1. Starts with the second element (index 1) and iterates through the array. Treats all elements before the current position as already sorted.
2. The inner loop compares the current element (key) with elements in the sorted portion, move larger elements one position to the right and inserts the key into its correct position within the sorted portion.

```
    outer_loop_insertion:
    bge s2, s1, insertion_done

    # Get current element
    slli t0, s2, 2      # t0 = i * 4
    add t0, s0, t0
    lw t1, 0(t0)        # key = arr[i]
    addi t2, s2, -1     # j = i-1

inner_loop_insertion:
    bltz t2, inner_done_insertion     # if j < 0, break

    # Compare elements
```

```
    slli t3, t2, 2
    add t3, s0, t3
    lw t4, 0(t3)       # arr[j]

    ble t4, t1, inner_done_insertion

    # Move element
    sw t4, 4(t3)       # arr[j+1] = arr[j]

    addi t2, t2, -1    # j--
    j inner_loop_insertion

inner_done_insertion:
    # Place key in correct position
    addi t2, t2, 1
    slli t3, t2, 2
    add t3, s0, t3
    sw t1, 0(t3)

    addi s2, s2, 1     # i++
    j outer_loop_insertion

insertion_done:
    lw ra, 12(sp)
    lw s0, 8(sp)
    lw s1, 4(sp)
    lw s2, 0(sp)
    addi sp, sp, 16
    ret
```

**Stage 2.4: Selection sort**

1. The outer loop iterates through the array, treating each position as the start of the unsorted portion.
2. The inner loop scans the remaining unsorted portion of the array to update the "min_idx" pointer whenever a smaller element is found..After the inner loop completes, the smallest element in the unsorted portion is found.

```
    outer_loop_selection:
    addi t0, s1, -1
    bge s2, t0, selection_done

    mv t1, s2          # min_idx = i
    addi t2, s2, 1     # j = i + 1

inner_loop_selection:
    bge t2, s1, inner_done_selection

    # Compare elements
    slli t3, t2, 2
    add t3, s0, t3
    lw t4, 0(t3)       # arr[j]
```

```
15
16      slli t5, t1, 2
17      add t5, s0, t5
18      lw t6, 0(t5)        # arr[min_idx]
19
20      bge t4, t6, no_update_min
21      mv t1, t2           # Update min_idx
22
23  no_update_min:
24      addi t2, t2, 1
25      j inner_loop_selection
26
27  inner_done_selection:
28      # Swap elements if needed
29      beq t1, s2, no_swap_selection
30
31      slli t2, s2, 2
32      add t2, s0, t2
33      lw t3, 0(t2)        # temp = arr[i]
34
35      slli t4, t1, 2
36      add t4, s0, t4
37      lw t5, 0(t4)        # arr[min_idx]
38
39      sw t5, 0(t2)        # arr[i] = arr[min_idx]
40      sw t3, 0(t4)        # arr[min_idx] = temp
41
42  no_swap_selection:
43      addi s2, s2, 1
44      j outer_loop_selection
45
46  selection_done:
47      lw ra, 12(sp)
48      lw s0, 8(sp)
49      lw s1, 4(sp)
50      lw s2, 0(sp)
51      addi sp, sp, 16
52      ret
```

## 2.5 Results

**Case 1:Normal**

**Case 2: Negative number**

**Case 3: Maximum number of elements**

**Case 4: File not exist error**

**Case 5: Output path not exist**

```
Enter filename: C:\Users\Admin\OneDrive\BÁCH KHOA\2024-1\Assembly Language and Computer Architecture Lab\CA\FinalProject\sortBasic.txt

Select sorting algorithm:
1. Bubble Sort
2. Insertion Sort
3. Selection Sort
4. Quick Sort
5.Quit
Choice: 1

Execution time (ms): 21
Select sorting algorithm:
1. Bubble Sort
2. Insertion Sort
3. Selection Sort
4. Quick Sort
5.Quit
Choice: 4

Execution time (ms): 3
```

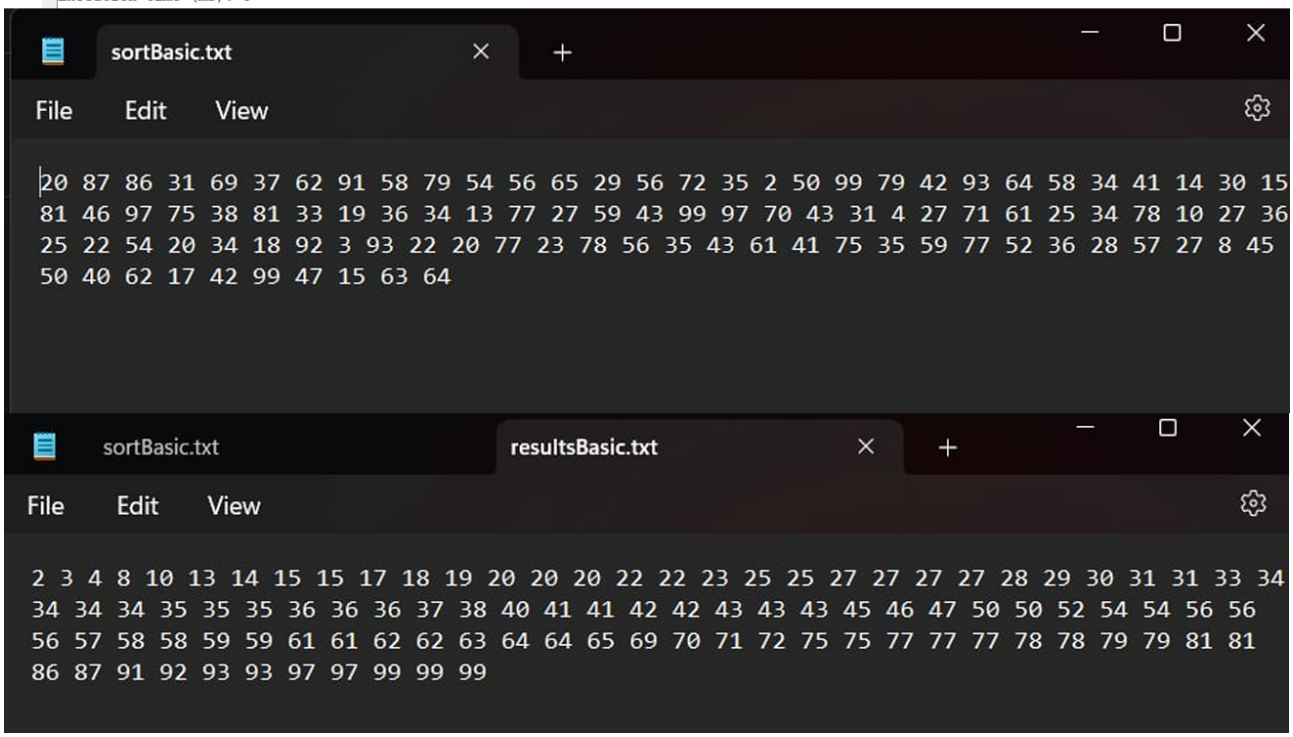


Figure 2: Case 1

```
Enter filename: C:\Users\Admin\OneDrive\BÁCH KHOA\2024-1\Assembly Language and Computer Architecture Lab\CA\FinalProject\sortBasicNev2.txt

Select sorting algorithm:
1. Bubble Sort
2. Insertion Sort
3. Selection Sort
4. Quick Sort
5.Quit
Choice: 1

Execution time (ms): 81
Select sorting algorithm:
1. Bubble Sort
2. Insertion Sort
3. Selection Sort
4. Quick Sort
5.Quit
Choice: 4

Execution time (ms): 9
```

sortBasicNev2.txt

File Edit View

```
-396 2915 1838 -1556 3715 2870 1739 1544 -1805 -3908 -3885 -3834 2775 -574 -1775 121
2187 -2266 -2187 -4137 27 -129 4145 -2171 -1660 2473 -2615 219 -3435 -1423 504 -4364
3981 137 -4181 4310 4667 4559 -4721 905 -3446 -1167 1643 -2745 441 -3620 4804 3529 -3657
3448 4468 4577 -1613 -1371 -1994 2873 3718 2046 -1105 2642 572 -736 574 -4352 4044 -4837
4130 1706 -2625 1871 -4307 -4048 -1437 863 4676 3075 -1578 -864 157 -176 3330 2379 -1313
941 -1908 719 4813 621 -489 1040 -2912 -550 2305 -510 -4656 4142 -3002 635 -688
2869 -2749 -2455 -4840 -2186 -2863 289 -2033 1724 1314 4502 2322
2195 -3992 -3987 -242 -3172 2273 -3120 -1835 -3145 1000 -549 -4963 443 1203 217 -2893
3867 -418 -3136 -2079 353 1192 2211 -2139 -3505 -2463 -1028 -1515 -482 1218 -1249
4395 -2213 2885 -2228 -189 -3057 -3406 -2152 -1211 -2641 -4329 -1057 -2149 -1505 -2180
468 -2420 -3235 1413 2010 -2482 4057 1676 -893 3984 -1689 -3616 159 -1545 2983 754 2399
4198 -945 4753 -2581 2313 154 3577 -3725 3029 2781 2692 4792 -1445 3246 -3596 4609
3848 -2595 2733 -3845 -4878 3196 3996 1413 2397 4994
```

sortBasicNev2.txt | resultsBasicNev2.txt

File Edit View

```
-4963 -4878 -4840 -4837 -4721 -4656 -4364 -4352 -4329 -4307 -4181 -4137 -4048 -3992 -398
7 -3908 -3885 -3845 -3834 -3725 -3657 -3620 -3616 -3596 -3505 -3446 -3435 -3406 -3235 -3
172 -3145 -3136 -3120 -3057 -3002 -2912 -2893 -2863 -2749 -2745 -2641 -2625 -2615 -2595
-2581 -2482 -2463 -2455 -2420 -2266 -2228 -2213 -2187 -2186 -2180 -2171 -2152 -2149 -213
9 -2079 -2033 -1994 -1908 -1835 -1805 -1775 -1689 -1660 -1613 -1578 -1556 -1545 -1515 -1
505 -1445 -1437 -1423 -1371 -1313 -1249 -1211 -1167 -1105 -1057 -1028 -945 -893 -864 -73
6 -688 -574 -550 -549 -510 -489 -482 -418 -396 -242 -189 -176 -129 27 121 137 154 157
159 217 219 289 353 441 443 468 504 572 574 621 635 719 754 863 905 941 1000 1040 1192
1203 1218 1314 1413 1413 1544 1643 1676 1706 1724 1739 1838 1871 2010 2046 2187 2195
2211 2273 2305 2313 2322 2379 2397 2399 2473 2642 2692 2733 2775 2781 2869 2870 2873
2885 2915 2983 3029 3075 3196 3246 3330 3448 3529 3577 3715 3718 3848 3867 3981 3984
3996 4044 4057 4130 4142 4145 4198 4310 4395 4468 4502 4559 4577 4609 4667 4676 4753
4792 4804 4813 4994
```

Figure 3: Case 2

```
Enter filename: C:\Users\Admin\OneDrive\BÁCH KHOA\2024-1\Assembly Language and Computer Architecture Lab\CA\FinalProject\sortBig.txt

Select sorting algorithm:
1. Bubble Sort
2. Insertion Sort
3. Selection Sort
4. Quick Sort
5.Quit
Choice: 4

Execution time (ms): 87
Select sorting algorithm:
1. Bubble Sort
2. Insertion Sort
3. Selection Sort
4. Quick Sort
5.Quit
Choice: 1

Execution time (ms): 2460
```

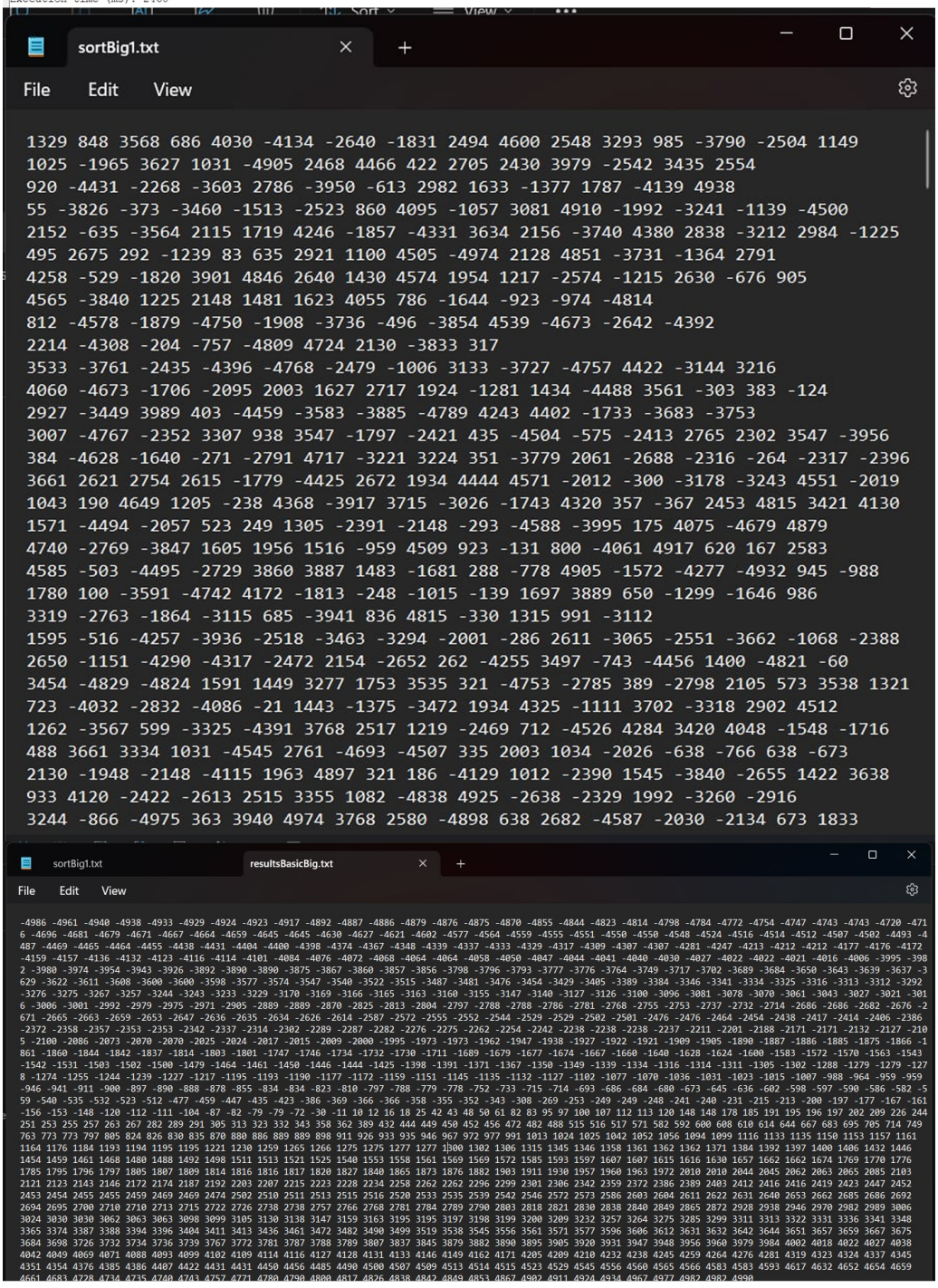


Figure 4: Case 3

## 2.6 Source code

GitHub

```
Enter filename: filekhongtontai.txt

Error opening file

-- program is finished running (0) --
```

Figure 5: Case 4

```
Enter filename: C:\Users\Admin\OneDrive\BÁCH KHOA\2024-1\Assembly Language and Computer Architecture Lab\CA\FinalProject\sortBig.txt

Select sorting algorithm:
1. Bubble Sort
2. Insertion Sort
3. Selection Sort
4. Quick Sort
5.Quit
Choice: 1

Execution time (ms): 2083
Error writing to output file
```

Figure 6: Case 5