Ministry of Education of Republic of Moldova

Technical University of Moldova

Faculty of Computers, Informatics and Microelectronics

Software Engineering Department

Computer Programming

Laboratory work #2

One-Dimensional Array Operations and Processing

Author:
Andrei Chicu
std. gr. FAF-233

Verified:
Alexandru Furdui

Chișinău 2023

Theory Background

A one-dimensional array, often simply referred to as an "array," is a data structure in computer programming that represents a collection of elements of the same data type stored in a linear sequence. These elements are typically accessed using an index or position within the array. One-dimensional arrays are among the simplest and most commonly used data structures in programming. I have researched the following sorting algorithms in order to implement them, and compare their speed:

1. Bubble sort
2. Selection sort
3. Insertion sort
4. Quick sort

And also, have learned the uses of pointers and their relation with arrays; to allocate[4] memory and copy[2] an array to this memory and have relied on these functions:

1. `malloc()`
2. `memcpy()`
3. `free()`

The Task

2.3 Hard

Implement these sorting algorithms[3]

1. Bubble
2. Selection
3. Insertion
4. Quick Sort

and explain them.

Technical implementation

pseudocode

```
FUNCTION print_arr(arr: Array of Integer, len: Integer, time:
    Double)
    IF time > 0 THEN
        PRINT "Time taken: " + time
    END IF
    FOR i FROM 0 TO len - 1 DO
        PRINT arr[i] + " "
    END FOR
    PRINT NEWLINE
END FUNCTION

FUNCTION swap(xp: Pointer to Integer, yp: Pointer to Integer)
    SET temp = *xp
    *xp = *yp
    *yp = temp
END FUNCTION

FUNCTION quick_sort_part(arr: Array of Integer, low: Integer, high:
    Integer) -> Integer
    SET pivot = arr[high]
    SET i = low - 1

    FOR j FROM low TO high - 1 DO
        IF arr[j] < pivot THEN
            INCREMENT i
            CALL swap(&arr[i], &arr[j])
        END IF
    END FOR

    CALL swap(&arr[i + 1], &arr[high])
    RETURN i + 1
END FUNCTION

FUNCTION copy_arr(arr: Array of Integer, len: Integer) -> Array of
    Integer
    DECLARE b as Array of Integer with length len
    COPY arr TO b
    RETURN b
```

```
END FUNCTION

FUNCTION bubble_sort(a: Array of Integer, len: Integer, cnt:
    Pointer to Double) -> Array of Integer
     SET a_copy = copy_arr(a, len)
     SET t_start = current_time()

     FOR i FROM 0 TO len - 1 DO
          FOR j FROM 0 TO len - 2 DO
               IF a_copy[j] > a_copy[j + 1] THEN
                    CALL swap(&a_copy[j], &a_copy[j + 1])
               END IF
          END FOR
     END FOR

     SET t_end = current_time()
     SET *cnt = (t_end - t_start) / CLOCKS_PER_SEC
     RETURN a_copy
END FUNCTION

FUNCTION selection_sort(a: Array of Integer, len: Integer, cnt:
    Pointer to Double) -> Array of Integer
     SET min_idx

     SET a_copy = copy_arr(a, len)
     SET t_start = current_time()

     FOR i FROM 0 TO len - 2 DO
          SET min_idx = i
          FOR j FROM i + 1 TO len - 1 DO
               IF a_copy[j] < a_copy[min_idx] THEN
                    SET min_idx = j
               END IF
          END FOR

          IF min_idx != i THEN
               CALL swap(&a_copy[min_idx], &a_copy[i])
          END IF
     END FOR

     SET t_end = current_time()
```

```
    SET *cnt = (t_end - t_start) / CLOCKS_PER_SEC
    RETURN a_copy
END FUNCTION

FUNCTION insertion_sort(a: Array of Integer, len: Integer, cnt:
   Pointer to Double) -> Array of Integer
    SET key, j, i

    SET a_copy = copy_arr(a, len)
    SET t_start = current_time()

    FOR i FROM 1 TO len - 1 DO
        SET key = a_copy[i]
        SET j = i - 1

        WHILE j >= 0 AND a_copy[j] > key DO
            SET a_copy[j + 1] = a_copy[j]
            DECREMENT j
        END WHILE

        SET a_copy[j + 1] = key
    END FOR

    SET t_end = current_time()
    SET *cnt = (t_end - t_start) / CLOCKS_PER_SEC
    RETURN a_copy
END FUNCTION

FUNCTION quick_sort(a: Array of Integer, low: Integer, high:
   Integer)
    IF low < high THEN
        SET pi = quick_sort_part(a, low, high)
        CALL quick_sort(a, low, pi - 1)
        CALL quick_sort(a, pi + 1, high)
    END IF
END FUNCTION

FUNCTION main()
    DECLARE len as Integer
    DECLARE cnt as Double
    SET cnt = 0
```

```
    INPUT len
    DECLARE arr as Array of Integer with length len

    FOR i FROM 0 TO len - 1 DO
        INPUT arr[i]
    END FOR

    SET bubble_arr = bubble_sort(arr, len, &cnt)
    PRINT "Bubble sort"
    CALL print_arr(bubble_arr, len, cnt)
    FREE bubble_arr

    SET insertion_arr = insertion_sort(arr, len, &cnt)
    PRINT "Insertion sort"
    CALL print_arr(insertion_arr, len, cnt)
    FREE insertion_arr

    SET a_copy = copy_arr(arr, len)
    SET t_start = current_time()
    CALL quick_sort(a_copy, 0, len - 1)
    SET t_end = current_time()
    SET cnt = (t_end - t_start) / CLOCKS_PER_SEC
    PRINT "Quick sort"
    CALL print_arr(a_copy, len, cnt)
    FREE a_copy

    SET selection_arr = selection_sort(arr, len, &cnt)
    PRINT "Selection sort"
    CALL print_arr(selection_arr, len, cnt)
    FREE selection_arr

    RETURN 0
END FUNCTION
```

Results

I have provided the program with a random array with the use of the for cycle of the shell, and the `$RANDOM` global variable.[1]

I have obtained the following time of execution for the implemented algorithms:

1. **Bubble sort** 0.000146 seconds
2. **Selection sort** 0.000063 seconds
3. **Insertion sort** 0.000032 seconds
4. **Quick sort** 0.000029 seconds

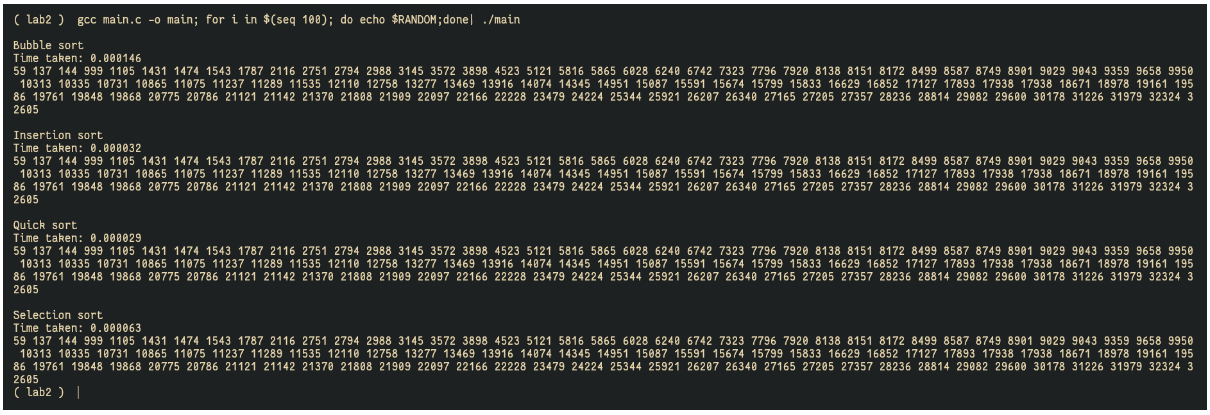

Figure 1: results

Conclusion

In conclusion I can say that *Quick sort* was on average the fastest of the algorithms, and bubble sort was the slowest.

But, interestingly enough, *Insertion sort* was at times faster than *Quick sort*, and the true difference of speed was seen only after several runs of the program on different randomly generated arrays. This could imply that the randomly generated arrays were at times partially sorted, at which *Insertion sort* performs well.[5]

References

[1] *Bash Shell Generate Random Numbers*. URL: `https://www.cyberciti.biz/faq/bash-shell-script-generating-random-numbers`.

[2] *Copying Memory in C*. URL: `https://www.youtube.com/watch?v=NqUTiJPgBn8`.

[3] *Geeks for Geeks*. URL: `https://www.geeksforgeeks.org`.

[4] *How to use malloc*. URL: `https://www.youtube.com/watch?v=yFboyOwk2oM`.

[5] *ProgrammerSought*. URL: `https://www.programmersought.com`.