Bloom filter, Summer 2017

Minoiu George Emilian

June 23, 2017

Teacher: Costin Bădică
Section: Computers with teaching in Romanian Class
Group: C.R. 1.2.
Year of study: I

1 Problem statement

A library for bloom lters . The library should allow inserting and searching for an item.

2 Pseudocode

```


int * quick_sort(struct Array *array){


    int size_array = size_of(array);
    int *aux = copy_array(array, size_array);
    int i;
    int j;
    int aux_number;
    int swap_number;
    begin_clock();

    quickSort(aux, 0, size_array -1);

    end_clock("Quick sort", 4);

    if(G_ACTIVATE_PRINT == 1){
        printf("\nArray after sorting: ");

        for(i = 0; i < size_array; i++)
            printf("%d ", aux[i]);
    }

    return aux;
}

void binary_search(struct Array *array, int number){

    int size_array = size_of(array);
    int *aux = sort_with_best(array, size_array);
    int aux_number;
    int left;
    int right;
    int middle;

    begin_clock();

    left = 0;
    right = size_array - 1;

    while (left <= right){
        middle = left + (right - left)/2;
        if (aux[middle] == number){
            printf("\nNumber %d found at position %d.",
                number, middle);
            break;
        }
        if (aux[middle] < number)
            left = middle + 1;
        else
            right = middle - 1;
    }

    if(left > right)
        printf("\nNumber %d was not found in the
            array.", number);

    end_clock("Binary search", 8);

}
```

2.1 Pseudocode description

The **quick sort** function is made for quick sorting a given array without modifying it. It's one of the methods used for sorting array from this program. The **binary_search** function is sorting an array with the best method of sorting found so far than binary search an item in the sorted array.

3 Application Design

3.1 Main

The **main** of my program contains a **while** loop so the user will be forced to choose a valid option. The user has the option to choose from sixteen different options, including all types of sorting, adding a number to the end of our array, adding a number to the beginning of our array, generate random numbers.
The user will have to insert some values in order for functions to be called.

3.2 Input Data

For my program, input data is "decision". An integer value for choosing an option in from the menu or telling how many numbers i want to be generated in array or telling which is the number i want to search for in current array.

3.3 Output Data

The data outputs resulted from functions processing. The functions include printing unmodified array to the console, printing array after sorting, printing the seconds that have passed for a sorting to be done, printing the seconds that have passed for a search to be done.

3.4 Functions used

queue_pointer new_queue(int queue_size) function is used to allocate dynamically a new queue. It creates a new priority queue with the given size for it's elements. It returns the address of memory where our new priority queue starts.

int * bubble_sort(struct Array *array) function is used for an array to be bubble sorted. The seconds took for this operation are printed to the console.

int * insertion_sort(struct Array *array) function is used for an array to be insertion sorted. The seconds took for this operation are printed to the console.

int * selection_sort(struct Array *array) function is used for an array to be selection sorted. The seconds took for this operation are printed to the console.

int * quick_sort(struct Array *array) function is used for an array to be quick sorted. The seconds took for this operation are printed to the console.

int * merge_sort(struct Array *array) function is used for an array to be merge sorted. The seconds took for this operation are printed to the console.

int * heap_sort(struct Array *array) function is used for an array to be heap sorted. The seconds took for this operation are printed to the console.

int * radix_sort(struct Array *array) function is used for an array to be radix sorted. The seconds took for this operation are printed to the console.

int * radix_sort(struct Array *array) function is used for an array to be radix sorted. The seconds took for this operation are printed to the console.

int * sort_with_best(struct Array *array, int size) function is used for an array to be sorted using the best method found so far. Sorting with all types of sorting will determine the best method.

void binary_search(struct Array *array, int number) function is used to search for a number into the given array. The array is being sorted at first using the best method found so far after the number is being searched.

void linear_search(struct Array *array, int number) function is used to search for a number into the given array. The search will be linear.

begin_clock() function is used to start a clock at the beginning of each function.

void end_clock(char sorting_method[100], unsigned int sorting_number) function is used to end the already started clock and prints result to the console.

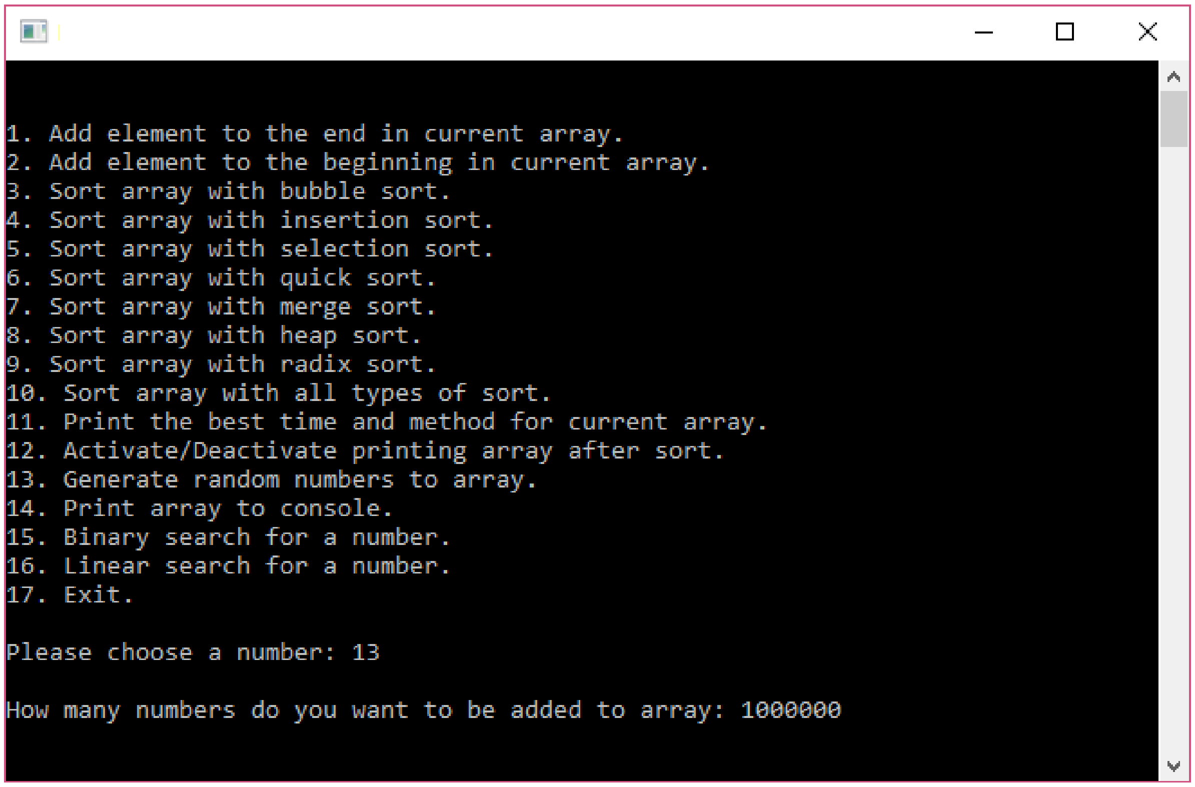
4 Source code

My program is wrote in C99 standard. It has basic libraries and functions

The code is compiled in two different compilers: "GNU GCC Compiler" and "Visual C++".

5 Experiments and results

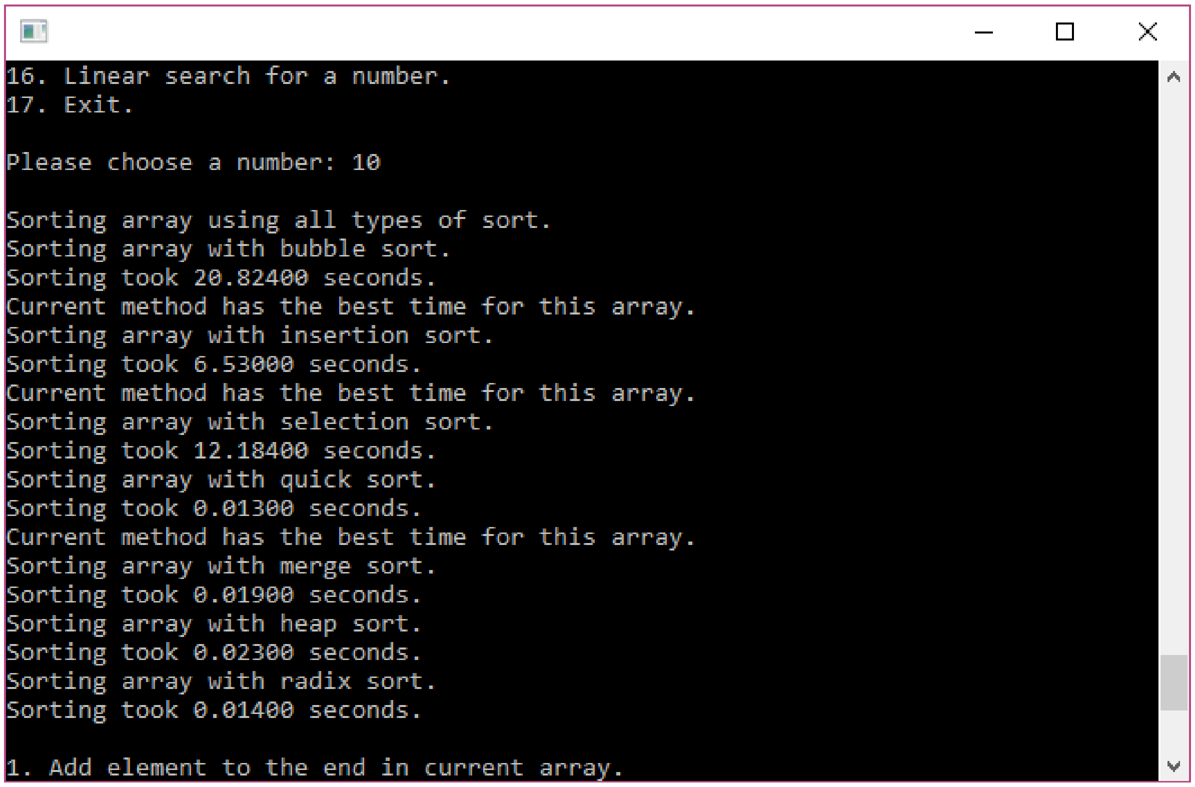
5.1 GCC Compiler



```
1. Add element to the end in current array.
2. Add element to the beginning in current array.
3. Sort array with bubble sort.
4. Sort array with insertion sort.
5. Sort array with selection sort.
6. Sort array with quick sort.
7. Sort array with merge sort.
8. Sort array with heap sort.
9. Sort array with radix sort.
10. Sort array with all types of sort.
11. Print the best time and method for current array.
12. Activate/Deactivate printing array after sort.
13. Generate random numbers to array.
14. Print array to console.
15. Binary search for a number.
16. Linear search for a number.
17. Exit.

Please choose a number: 13

How many numbers do you want to be added to array: 1000000
```

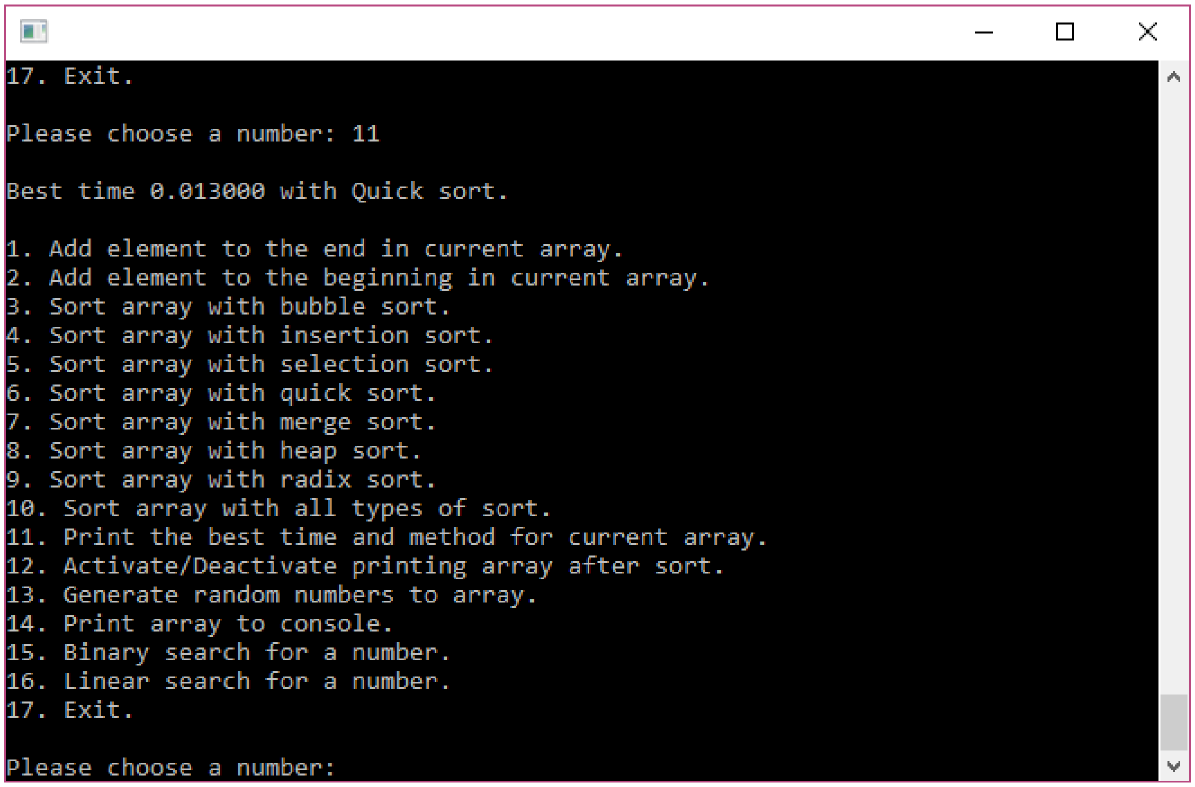


```
16. Linear search for a number.
17. Exit.

Please choose a number: 10

Sorting array using all types of sort.
Sorting array with bubble sort.
Sorting took 20.82400 seconds.
Current method has the best time for this array.
Sorting array with insertion sort.
Sorting took 6.53000 seconds.
Current method has the best time for this array.
Sorting array with selection sort.
Sorting took 12.18400 seconds.
Sorting array with quick sort.
Sorting took 0.01300 seconds.
Current method has the best time for this array.
Sorting array with merge sort.
Sorting took 0.01900 seconds.
Sorting array with heap sort.
Sorting took 0.02300 seconds.
Sorting array with radix sort.
Sorting took 0.01400 seconds.

1. Add element to the end in current array.
```



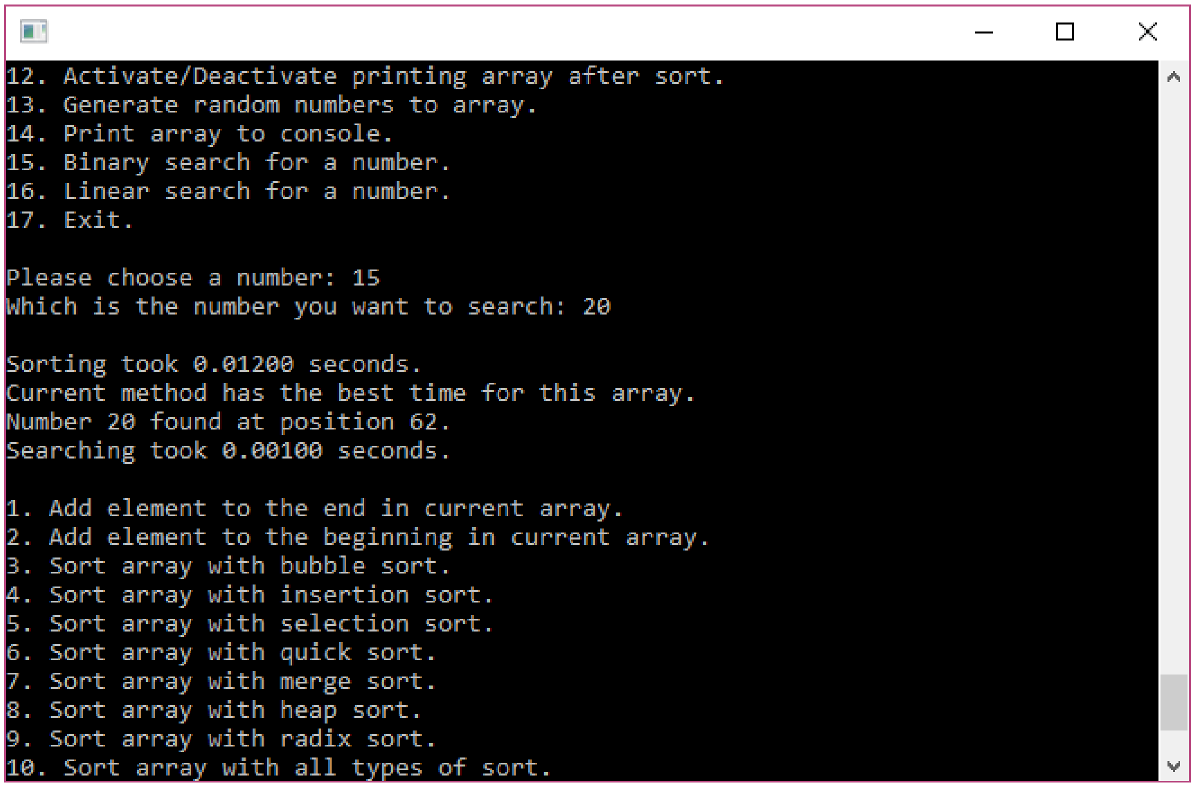
```
17. Exit.

Please choose a number: 11

Best time 0.013000 with Quick sort.

1. Add element to the end in current array.
2. Add element to the beginning in current array.
3. Sort array with bubble sort.
4. Sort array with insertion sort.
5. Sort array with selection sort.
6. Sort array with quick sort.
7. Sort array with merge sort.
8. Sort array with heap sort.
9. Sort array with radix sort.
10. Sort array with all types of sort.
11. Print the best time and method for current array.
12. Activate/Deactivate printing array after sort.
13. Generate random numbers to array.
14. Print array to console.
15. Binary search for a number.
16. Linear search for a number.
17. Exit.

Please choose a number:
```

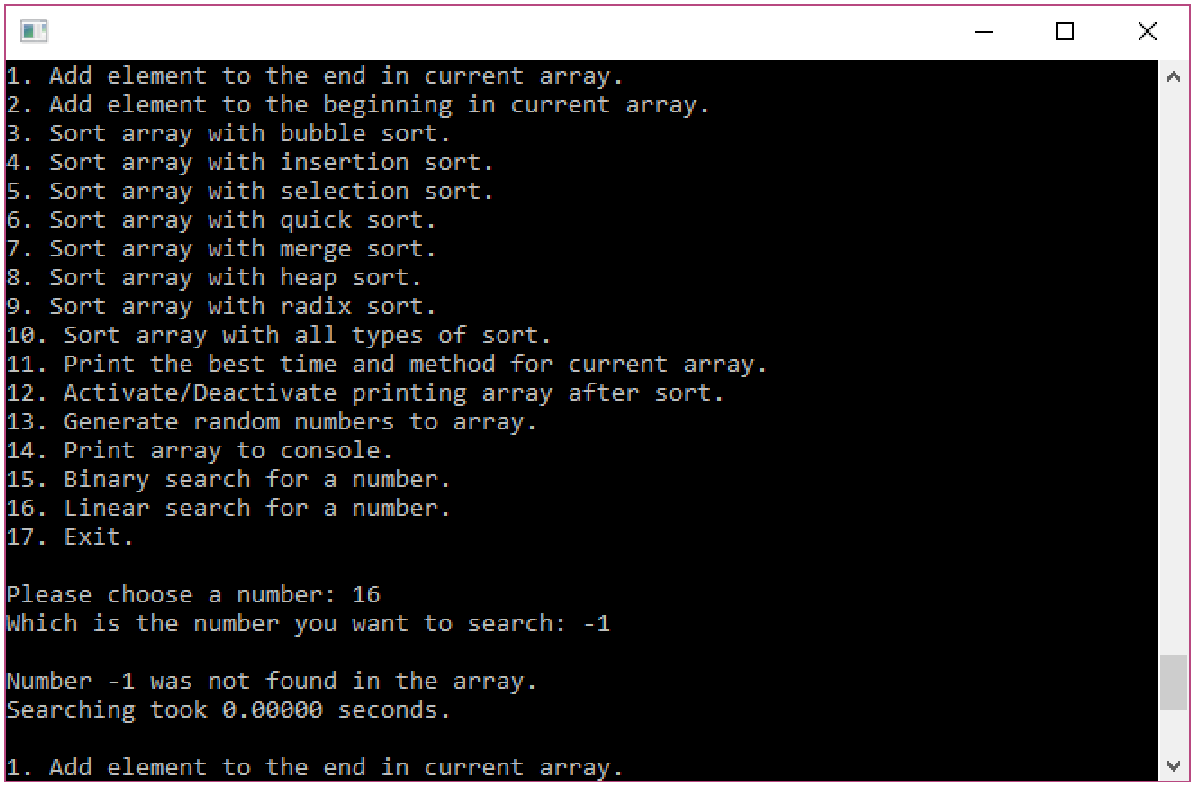


```
12. Activate/Deactivate printing array after sort.
13. Generate random numbers to array.
14. Print array to console.
15. Binary search for a number.
16. Linear search for a number.
17. Exit.

Please choose a number: 15
Which is the number you want to search: 20

Sorting took 0.01200 seconds.
Current method has the best time for this array.
Number 20 found at position 62.
Searching took 0.00100 seconds.

1. Add element to the end in current array.
2. Add element to the beginning in current array.
3. Sort array with bubble sort.
4. Sort array with insertion sort.
5. Sort array with selection sort.
6. Sort array with quick sort.
7. Sort array with merge sort.
8. Sort array with heap sort.
9. Sort array with radix sort.
10. Sort array with all types of sort.
```



```
1. Add element to the end in current array.
2. Add element to the beginning in current array.
3. Sort array with bubble sort.
4. Sort array with insertion sort.
5. Sort array with selection sort.
6. Sort array with quick sort.
7. Sort array with merge sort.
8. Sort array with heap sort.
9. Sort array with radix sort.
10. Sort array with all types of sort.
11. Print the best time and method for current array.
12. Activate/Deactivate printing array after sort.
13. Generate random numbers to array.
14. Print array to console.
15. Binary search for a number.
16. Linear search for a number.
17. Exit.

Please choose a number: 16
Which is the number you want to search: -1

Number -1 was not found in the array.
Searching took 0.00000 seconds.

1. Add element to the end in current array.
```

6 Conclusions

Dynamic allocation of memory is helping me store and generate numbers till my memory fills up.

References

[1] **Links**
Alexander S. Kulikov, Singly linked lists https://www.coursera.org/learn/data-structures/lecture/kHhgK/singly-linked-lists
Neil Rhodes,University of California, San Diego https://www.coursera.org/learn/data-structures/lecture/OsBSF/arrays