Lab 2
Isa Dzhumabaev.

Both algorithms are implemented using C lang, compiled using clang and ran on MacOS . Function rand() was used to generate random numbers to be sorted.

Report

MergeSort is considered to be more stable and in theory it should be faster, but in practice it appeared to be slower than QuickSort on random generated numbers.

Statistics on algorithms

	1000	100 000	500 000	1 000 000	2 000 000
QuickSort	0.000164 sec	0.014865 sec	0.066969 sec	0.143248 sec	0.284892 sec
MergeSort	0.000241 sec	0.021833 sec	0.106593 sec	0.218164 sec	N/A

As we can observe QuickSort beats MergeSort on 1000 input already. We can also see that QuickSort was almost 1.5 times faster than MergeSort on 1 million input. It may be assumed that on 2 million input the ratio will also be larger and will continue to grow as input size grows up. Considering that random numbers were used in this comparison conclusion may be made that QuickSort is better adapted for real-life problems than MergeSort.

Source Code

All prints removed so it is more readable. You can find version with prints in the attached .c files. You can also find output reports of both algorithms in .txt files.

QuickSort:

```c
#include "stdio.h"
#include "time.h"
#include "stdlib.h"

int partition (int* arr, int low, int high)
{
    int pivot = arr[high];
    int i = (low - 1);

    for (int j = low; j < high; j++)
    {
        if (arr[j] < pivot)
        {
            i++;
            int t = arr[i];
```

```
            arr[i] = arr[j];
             ;
        }
    }
    int t = arr[i + 1];
    arr[i + 1] = arr[high];
    arr[high] = t;

    return (i + 1);
}

void quickSort(int* arr, int low, int high)
{
    if (low < high)
    {

        int pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

int main(int argc, char** argv) {

    int sz = -1;
    printf("Enter size of array:\n");
    scanf("%d", &sz);
    int arr[sz];

    srand(time(NULL));
    for (int i = 0; i < sz; ++i)
    {
       arr[i] = rand() % (sz * 20);
    }

    clock_t begin = clock();
    quickSort(arr, 0, sz - 1);
    clock_t end = clock();

    printf("Time taken: %f seconds\n", ((double) (end - begin)) /
CLOCKS_PER_SEC);

    return 0;
}
```

MergeSort:

```
#include "stdio.h"
#include "time.h"
#include "stdlib.h"

void merge(int* arr, int low, int mid, int high)
{
    int i, j, k;
    int sz_a = mid - low + 1;
```

```
    int sz_b =  high - mid;

    int a[sz_a], b[sz_b];

    for (i = 0; i < sz_a; i++) {
        a[i] = arr[low + i];
    }
    for (j = 0; j < sz_b; j++) {
        b[j] = arr[mid + j + 1];
    }

    i = j = 0;
    k = low;
    while (i < sz_a && j < sz_b)
    {
        if (a[i] <= b[j])
        {
            arr[k++] = a[i++];
        }
        else
        {
            arr[k++] = b[j++];
        }
    }

    while (i < sz_a)
    {
        arr[k++] = a[i++];
    }

    while (j < sz_b)
    {
        arr[k++] = b[j++];
    }

}

void mergeSort(int* arr, int low, int high)
{
    if (low < high)
    {
        int mid = low + (high - low) / 2;

        mergeSort(arr, low, mid);
        mergeSort(arr, mid + 1, high);
        merge(arr, low, mid, high);
    }
}

int main(int argc, char** argv) {

    int sz = -1;
    printf("Enter size of array:\n");
    scanf("%d", &sz);
    int arr[sz];

    srand(time(NULL));
```

```
    for (int i = 0; i <= sz; ++i)
    {
        arr[i] = rand() % (sz * 20);
    }

    clock_t begin = clock();
    mergeSort(arr, 0, sz - 1);
    clock_t end = clock();

    printf("Total time: %f seconds\n", ((double) (end - begin)) /
CLOCKS_PER_SEC);
    return 0;
}
```