Lab 5
Isa Dzhumabaev

1) Compare randomized quicksort to quicksort and quadsort using various input scenarios.

I used Quick Sort algorithm from previous labs and added **rand_quickSort()** function that uses randomized pivot. First, my program generates an array with N random integer numbers, then sends it to standard **norm_quickSort()** and calculates its time, after that it sends the same array to randomized **rand_quickSort()** function and also calculates its time.

As you can see in screenshot below randomized quick sort was faster in all cases.

I used some simple formula to calculate average ratio and we can see that randomized quick sort is about 1.5 times faster then standard quick sort.

$$\frac{\frac{0.000049}{0.000027}+\frac{0.000343}{0.000279}+\frac{0.003447}{0.002271}+\frac{0.026845}{0.015785}+\frac{0.258034}{0.18084}+\frac{0.531923}{0.336194}}{6}=1.54529\ldots$$

2) Record the results.

Screenshots of comparisons:

N = 100

```
$ ./a.out
Enter size of array:
100
Time taken for norm_quickSort: 0.000049 seconds
Time taken for rand_quickSort: 0.000027 seconds
```

N = 1000

```
$ ./a.out
Enter size of array:
1000
Time taken for norm_quickSort: 0.000343 seconds
Time taken for rand_quickSort: 0.000279 seconds
```

N = 10 000

```
$ ./a.out
Enter size of array:
10000
Time taken for norm_quickSort: 0.003447 seconds
Time taken for rand_quickSort: 0.002271 seconds
```

N = 100 000

```
$ ./a.out
Enter size of array:
1000000
Time taken for norm_quickSort: 0.258034 seconds
Time taken for rand_quickSort: 0.180840 seconds
```

N = 1 000 000

```
$ ./a.out
Enter size of array:
100000
Time taken for norm_quickSort: 0.026845 seconds
Time taken for rand_quickSort: 0.015785 seconds
```

N = 2 000 000

```
$ ./a.out
Enter size of array:
2000000
Time taken for norm_quickSort: 0.531923 seconds
Time taken for rand_quickSort: 0.336194 seconds
```

3) Which one is better and why?

Randomized quick sort is better as it lower chances that after partition operation we get one array of n-1 elements and second with 0 elements. This situation is the worst case and leads to O(n*n) in case we get such a partition in every partition operation. By lowering chances for worst case we increase chances for O(n*log(n)) time which is obviously a better option.

Source code:

```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];
            arr[j] = t;
```

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

    return (i + 1);
}


int partition_r(int* arr, int low, int high)
{
    srand(time(NULL));
    int random = low + rand() % (high - low);

    int t = arr[random];
    arr[random] = arr[high];
    arr[high] = t;

    return partition(arr, low, high);
}

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);
    }
}

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

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

        rand_quickSort(arr, low, pi - 1);
        rand_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();
    rand_quickSort(arr, 0, sz - 1);
    clock_t end = clock();
    printf("Time taken for norm_quickSort: %f seconds\n", ((double) (end -
begin)) / CLOCKS_PER_SEC);

    begin = clock();
    rand_quickSort(arr, 0, sz - 1);
    end = clock();
    printf("Time taken for rand_quickSort: %f seconds\n", ((double) (end -
begin)) / CLOCKS_PER_SEC);

    return 0;
}
```