DESIGN AND ANALYSIS OF ALGORITHMS LAB

NAME: Mohammed Saad Belgi

UID: 2021700005

BATCH: A

BRANCH: CSE DS

EXPT. NO.: 1B

AIM: Experiment on finding the running time of an algorithm.

ALGORITHM:

Selection sort:

SELECTION-SORT(ARR):

```
1. for i = 1 to ARR.LENGTH-1:
2.     min_idx ← i
3.     for j = i + 1 to ARR.LENGTH:
4.         if ARR[j] < ARR[min_idx]:
5.             min_idx = j
6.     if i != min_idx:
7.         temp ← ARR[min_idx]
8.         ARR[min_idx] ← ARR[i]
9.         ARR[i] ← temp
```

Insertion sort:

INSERTION-SORT(ARR):

```
1. for j = 2 to ARR.LENGTH:
2.     key ← ARR[j]
3.     i ← j – 1
4.     while i > 0 and ARR[i] > key:
5.         ARR[i+1] ← ARR[i]
6.         i ← i - 1
7.     ARR[i + 1] ← key
```

THEORY:

Selection and insertion sort algorithms have a worst-case time complexity of $O(n^2)$, n being the size of the input array to be sorted. Both algorithms require n(n-1)/2 comparisons in worst-case scenario. However, selection sort performs n(n-1)/2 comparisons in all cases and thus has quadratic time complexity for best and average cases as well, while insertion sort requires fewer comparisons in most cases. In best-case scenario, insertion sort requires n comparisons, and thus has a linear time complexity. This is why insertion sort outperforms selection sort in most cases.

Both these algorithms are in-place sorting algorithms, i.e., extra data structures are not required to sort an array. Number of temporary variables required is constant (independent of input size) for both these algorithms, and thus they have a constant space complexity (O(1)).

CODE:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>    // required to use clock function if executing program on linux
#ifdef _WIN32        // to check if OS is windows (_WIN32 is a macro defined on every windows based gcc compiler)
#include <windows.h> // required to use windows api for time measurement if executing on windows
#endif

/* clock function of time.h on linux provides precision upto microseconds,
but on windows precision upto only milliseconds is supported by clock function
on windows, the windows api has much better functions for measuring time. */

// prototypes:
void selectionSort(int *, int);
void insertionSort(int *, int);
void deepCopy(int *, int *, int);
void generateRandomNumbers();
void LinuxMain();
#ifdef _WIN32
void WindowsMain();
#endif

unsigned long long compareCounter, assignmentCounter; // global variables to keep track of number of comparisons and assignment operations

int main()
{
    generateRandomNumbers();
#ifdef _WIN32
    WindowsMain();
```

```
#else
    LinuxMain();
#endif
    return 0;
}

void selectionSort(int *arr, int len)
{
    compareCounter = 0;
    assignmentCounter = 0;
    int min_i, temp;
    for (int i = 0; i < len; i++)
    {
        min_i = i;
        for (int j = i + 1; j < len; j++)
        {
            compareCounter++;
            if (arr[j] < arr[min_i])
            {
                min_i = j;
                assignmentCounter++;
            }
        }
        if (i != min_i)
        {
            assignmentCounter += 3;
            temp = arr[min_i];
            arr[min_i] = arr[i];
            arr[i] = temp;
        }
    }
}

void insertionSort(int *arr, int len)
{
    compareCounter = 0;
    assignmentCounter = 0;
    int key, pos;
    for (int i = 1; i < len; i++)
    {
        key = arr[i];
        pos = 0;
```

```
        for (int j = i - 1; j >= 0; j--)
        {
            compareCounter++;
            if (arr[j] > key)
            {
                arr[j + 1] = arr[j];
                assignmentCounter++;
            }
            else
            {
                pos = j + 1;
                assignmentCounter++;
                break;
            }
        }
        arr[pos] = key;
        assignmentCounter++;
    }
}

void deepCopy(int *source, int *dest, int len)
{
    for (int i = 0; i < len; i++)
        dest[i] = source[i];
}

void generateRandomNumbers()
{
    FILE *fptr = fopen("rand_num.txt", "w");
    time_t cur_time;
    srand((unsigned int)time(&cur_time));
    for (int i = 0; i < 100000; i++)
        fprintf(fptr, "%d\n", rand());
    fclose(fptr);
}

#ifdef _WIN32
void WindowsMain()
{
    FILE *rand_num = fopen("rand_num.txt", "r");
    FILE *dest = fopen("output.txt", "w");
```

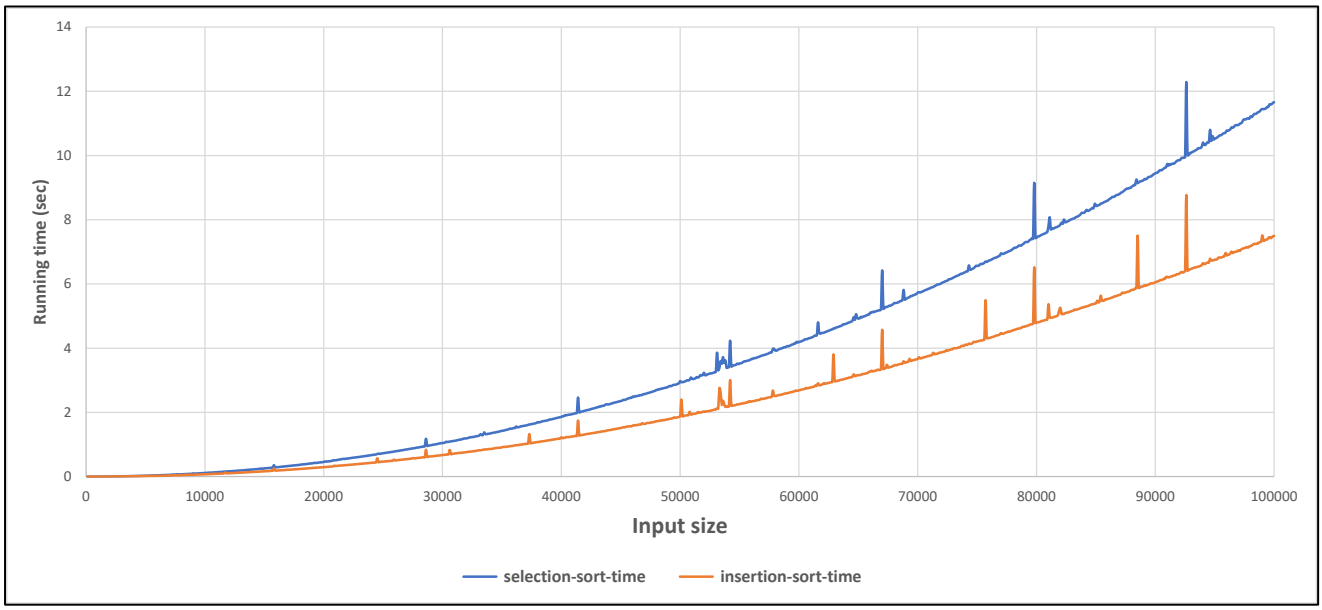
```
    fprintf(dest, "size   | selection-sort-time |
insertion-sort-time | selection-sort-comparisons |
insertion-sort-comparisons | selection-sort-assignments |
insertion-sort-assignments\n");
    double time1, time2;
    unsigned long long selectionSortComparisons,
insertionSortComparisons, selectionSortAssignments,
insertionSortAssignments;
    LARGE_INTEGER clock_freq, start, end;
    QueryPerformanceFrequency(&clock_freq);
    for (int size = 52000; size <= 52000; size += 100)
    {
        int arr1[size];
        int arr2[size];
        for (int j = 0; j < size; j++)
            fscanf(rand_num, "%d", &arr1[j]);
        fseek(rand_num, 0, SEEK_SET);
        deepCopy(arr1, arr2, size);

        QueryPerformanceCounter(&start);
        selectionSort(arr1, size);
        QueryPerformanceCounter(&end);
        time1 = (double)(end.QuadPart - start.QuadPart) *
1.0 / clock_freq.QuadPart;
        selectionSortComparisons = compareCounter;
        selectionSortAssignments = assignmentCounter;

        QueryPerformanceCounter(&start);
        insertionSort(arr2, size);
        QueryPerformanceCounter(&end);
        time2 = (double)(end.QuadPart - start.QuadPart) *
1.0 / clock_freq.QuadPart;
        insertionSortComparisons = compareCounter;
        insertionSortAssignments = assignmentCounter;

        fprintf(dest, "%6d | %19f | %19f | %26llu | %26llu
| %26llu | %26llu\n", size, time1, time2,
selectionSortComparisons, insertionSortComparisons,
selectionSortAssignments, insertionSortAssignments);
        printf("Size %d done!\n", size);
    }
    fclose(rand_num);
```

```
    fclose(dest);
}
#endif

void LinuxMain()
{
    FILE *rand_num = fopen("rand_num.txt", "r");
    FILE *dest = fopen("output.txt", "w");
    fprintf(dest, "size   | selection-sort-time |
insertion-sort-time | selection-sort-comparisons |
insertion-sort-comparisons | selection-sort-assignments |
insertion-sort-assignments\n");
    double time1, time2;
    clock_t start, end;
    unsigned long long selectionSortComparisons,
insertionSortComparisons, selectionSortAssignments,
insertionSortAssignments;
    for (int size = 100; size <= 100000; size += 100)
    {
        int arr1[size];
        int arr2[size];
        for (int j = 0; j < size; j++)
            fscanf(rand_num, "%d", &arr1[j]);
        fseek(rand_num, 0, SEEK_SET);
        deepCopy(arr1, arr2, size);

        start = clock();
        selectionSort(arr1, size);
        end = clock();
        time1 = (double)(end - start) * 1.0 /
CLOCKS_PER_SEC;
        selectionSortComparisons = compareCounter;
        selectionSortAssignments = assignmentCounter;

        start = clock();
        insertionSort(arr2, size);
        end = clock();
        time2 = (double)(end - start) * 1.0 /
CLOCKS_PER_SEC;
        insertionSortComparisons = compareCounter;
        insertionSortAssignments = assignmentCounter;
```

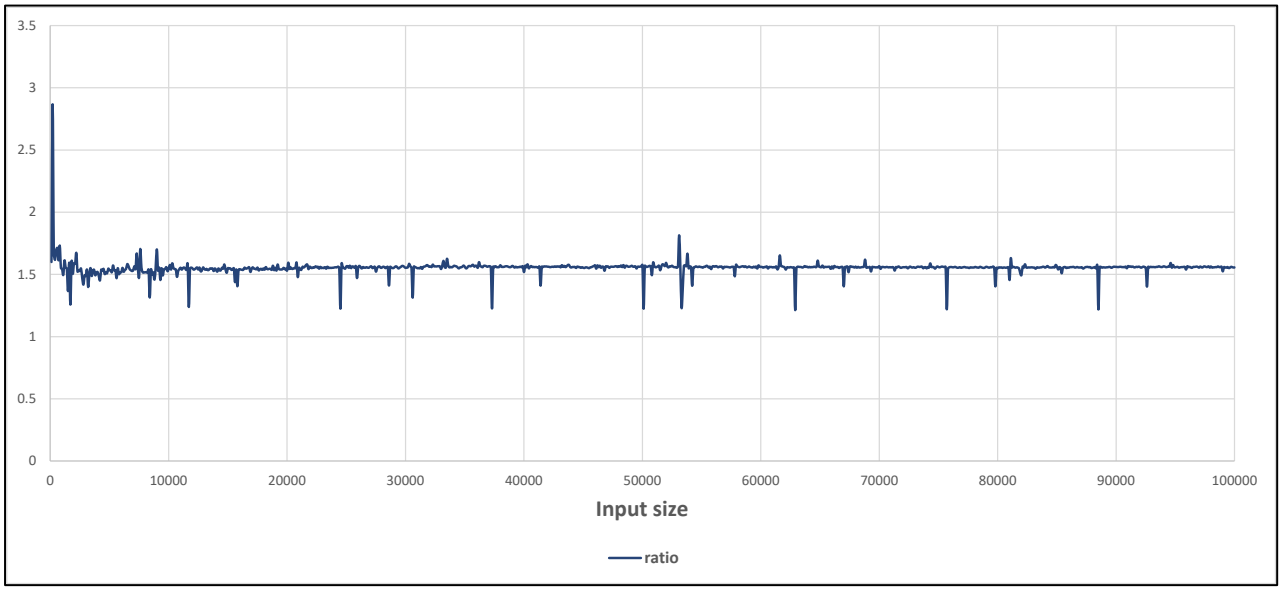
```
        fprintf(dest, "%6d | %19f | %19f | %26llu | %26llu
| %26llu | %26llu\n", size, time1, time2,
selectionSortComparisons, insertionSortComparisons,
selectionSortAssignments, insertionSortAssignments);
        printf("Size %d done!\n", size);
    }
    fclose(rand_num);
    fclose(dest);
}
```

PLOTTING THE DATA OBTAINED AFTER EXECUTION IN EXCEL:

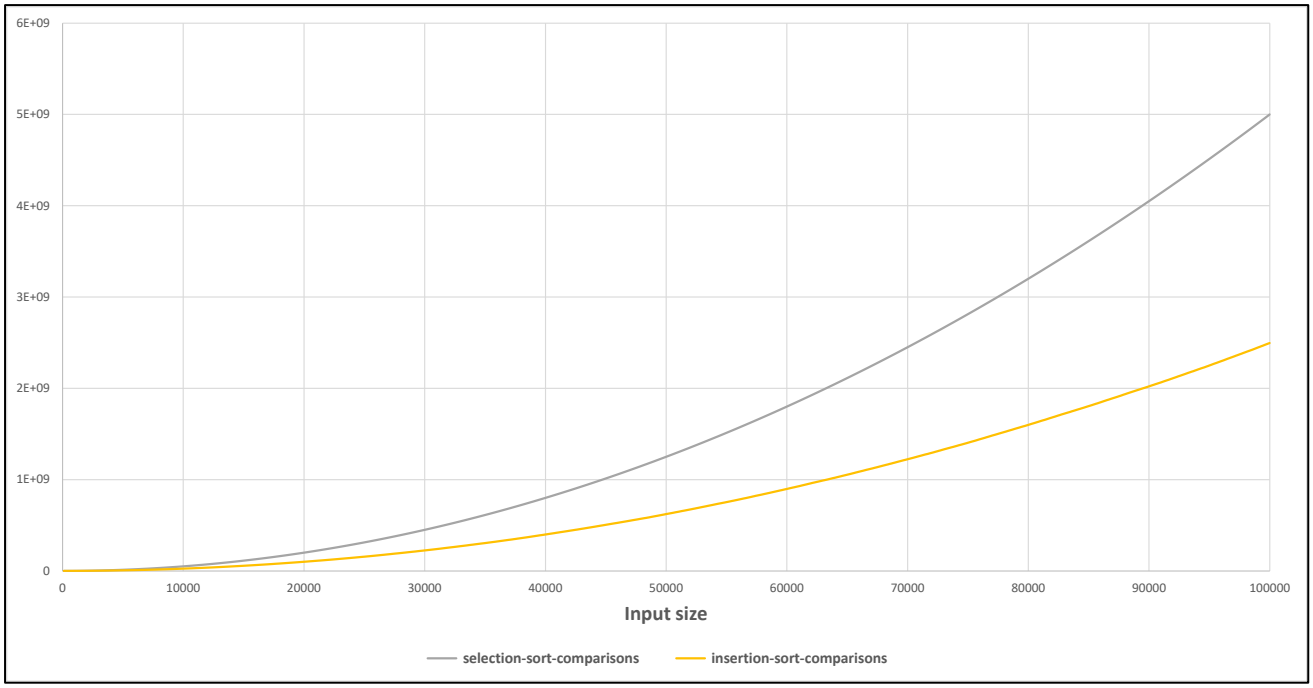
Graph of running time of selection sort and insertion sort vs input size:



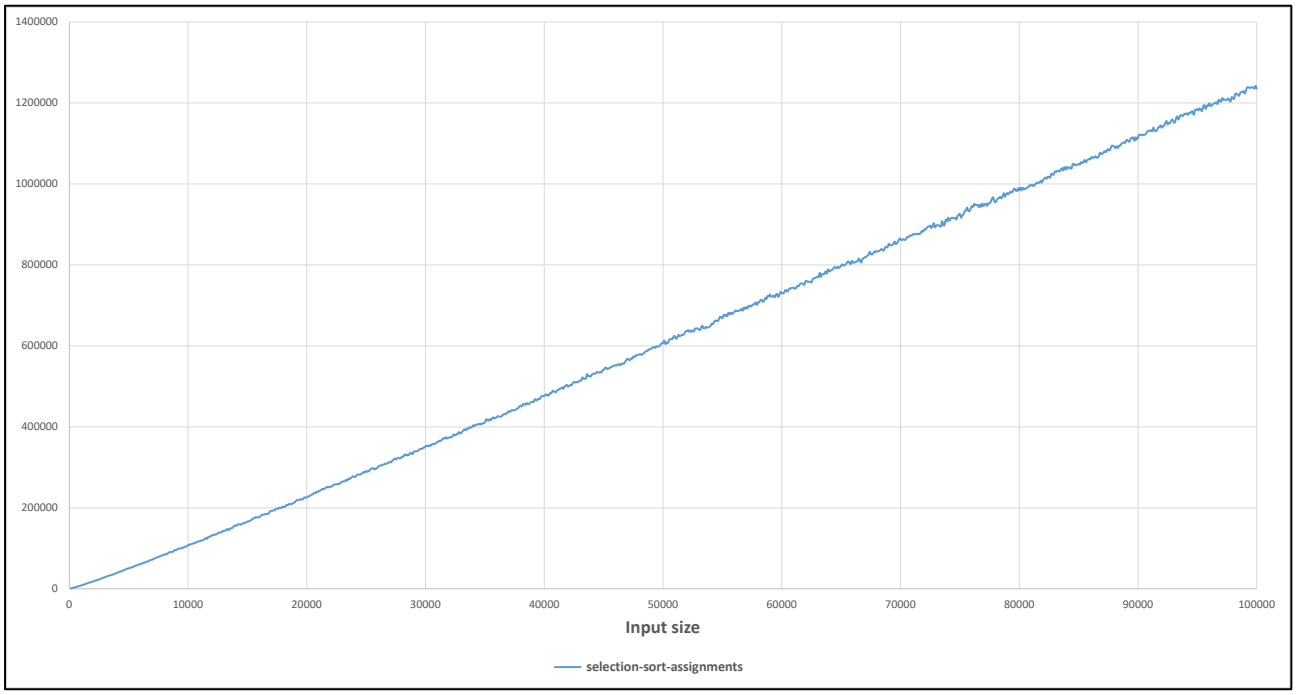
Graph of ratio of running time of selection sort to that of insertion sort vs input size:

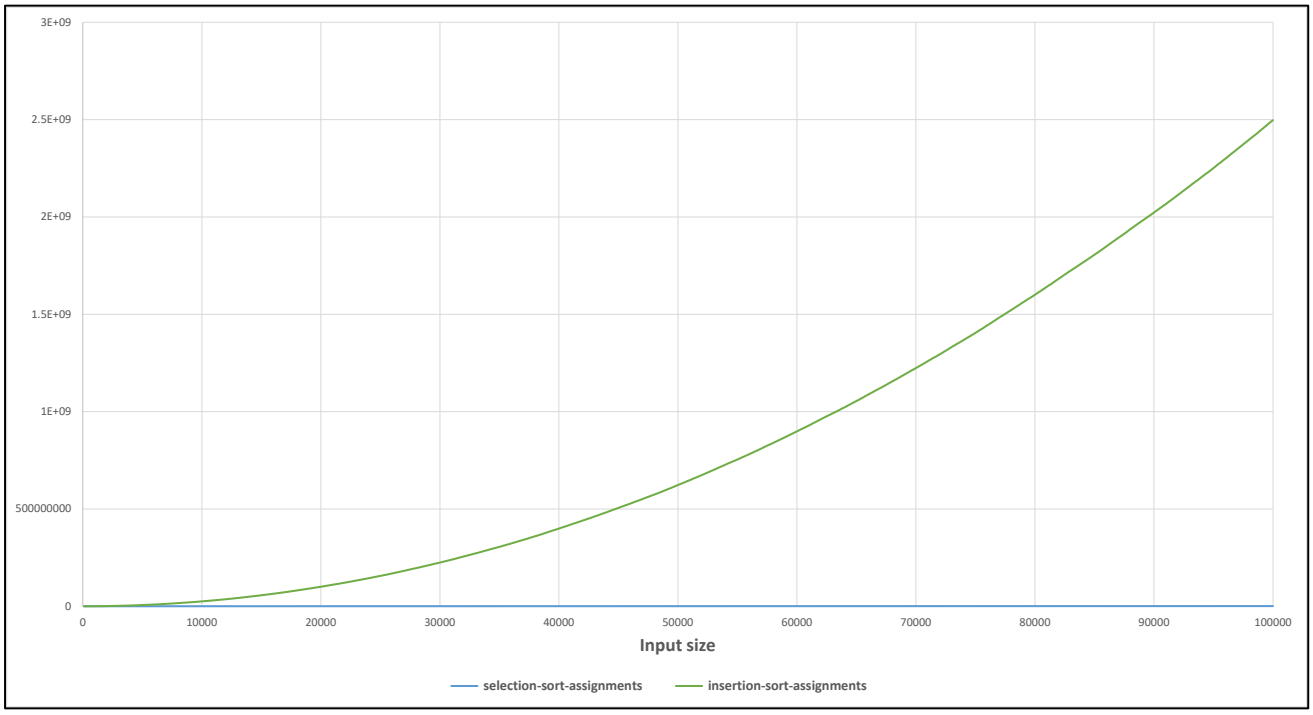


Graph of number of comparison operations of insertion and selection sorts vs input size:



Graph of number of assignment operations required for selection and insertion sorts vs input size:





EMPIRICAL OBSERVATIONS AND RESULT ANALYSIS:

Insertion sort beats selection sort for every input size.

There are large fluctuations in the ratio of running time of the two algorithms (selection/insertion sort) for very small values of input size (n < 10000). For n>10000, this ratio stabilises around an approximate value of 1.55. From empirical observation, running

time of selection sort was found to be roughly 1.5 times of that of insertion sort for the same input.

Number of comparison operations is proportional to n^2 for both algorithms. However, insertion sort performs fewer comparisons in most cases than selection sort. Selection sort performs the same number of comparisons regardless of the degree to which array is already sorted. Insertion sort on the other hand, performs fewer than worst case comparisons (when array is sorted in opposite order) for most cases.

Selection sort requires far fewer assignment operations than insertion sort. This is because this number is linearly proportional to the input size in case of selection sort. For an array of size n, selection sort will perform at most n-1 swapping operations. For insertion sort, number of assignment has quadratic proportionality with input size, as array might have to shifted to the right each time a new element has to be inserted.

CONCLUSION:

Insertion sort outperforms selection sort for same input most of the times, despite having the same worst-case time complexity of $O(n^2)$.