**1.Files and Structure:**

Problem Statement:

Use the random number generator program written earlier to do the following:

1 . Implement Insertion Sort. Bubble Sort, Selection Sort, Merge Sort and Quick Sort each with the following signature:

void arr[, int l. int r):

where x is one of the above sorting algorithms. and I represents the index of the left-most element and r represents the index Of the right-most element \_

Thus, to Sort int arr15] = (Q, 17. C, -20, using Merge Sort you would call

Merge\_sort(arr,0,4);

2. Generate and store N random numbers in the range [INT\_MIN, INT\_MAX] in a file. storing one number per line.

3. Produce one such file for each of these values of N = {20,21, 22,….,215, 216,217,218 ,219).

3.1 For each of these files, make two additional copies per file. where the randomly generated numbers are sorted in ascending Order, and another in descending order. For example, for a N = 2' 2 = we might have the first randomly generated file as:

19

-7

3

So, the other two files for that would be:

-7

3

19

and

19

3

-7

(This is done so that we can evaluate the performance of each sorting algorithm for both randomly ordered cases and

already sorted cases.)

4.For each file, read the numbers in that file into an array, and sort that array using each of the above mentioned algorithms. recording the number of comparisons taken by each algorithm for that value of N. Be careful not to pass the already sorted array from one sorting function to the next (you may use helper function to take cope of the

array).

An example spreadsheet for the recorded values may lock like:

Algorithm, N, comparisons\_taken

merge, 1024,10311

quick. 1024,

(You may skip the highest few powers of 2 for values of N if the sorting takes more than 15 mins to complete.)

5. Plot the above observations as line graph, taking X-axis log2(N) and Y-axis with one line plot per sorting algorithm, and plot all of them on the same graph with the same scale.

The number of comparisons taken for each algorithm. for each input file, may be recorded using global variable (that is reset between function calls. after recording the value). The comparisons to count are the comparisons used internally in your implementation of the respective sorting algorithm. when comparing the values of elements with

each other for deciding whether to swap. In general, the number of comparisons should correspond to the complexity of the sorting algorithm.

• Input example :

In the Command Plate->

gcc ./main.c

./a.exe 16 "E:\collage\c\sorting\resources\nums2^4.txt" "E:\collage\c\sorting\resources\nums2^4inc.txt" "E:\collage\c\sorting\resources\nums2^4dec.txt" "E:\collage\c\sorting\stats.csv"

• Output example :

select,16,120

select,16,120

select,16,120

bubble,16,110

bubble,16,15

bubble,16,120

insert,16,43

insert,16,15

insert,16,120

merge,16,45

merge,16,32

merge,16,32

quick,16,57

quick,16,120

quick,16,120

Proposed C Code:

/\*-------gen\_random.c--------\*/

(For generating 20 random files)

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

int main()

{

FILE \*fwr;

for (int i = 0; i < 17; i++)

{

char \*str = (char \*)calloc(100, sizeof(char));

sprintf(str, "E:\\collage\\c\\sorting\\resources\\nums2^%d.txt", i);

fwr = fopen(str, "w");

int p = pow(2, i);

for (int j = 0; j < p; j++)

{

int a = rand() % 2;

if (a == 0)

{

fprintf(fwr, "%d\n", rand());

}

else

{

fprintf(fwr, "%d\n", (-1) \* rand());

}

}

fclose(fwr);

}

return 0;

}

/\*-------------------\*/

/\*-------generate\_asc\_dsc.c-------\*/

(For sorting those 20 files in ascending and descending order)

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

void sort(int \*arr, int size)

{

for (int i = 0; i < size; i++)

{

for (int j = i + 1; j < size; j++)

{

if (arr[i] > arr[j])

{

int n = arr[i];

arr[i] = arr[j];

arr[j] = n;

}

}

}

}

int main()

{

for (int i = 0; i < 17; i++)

{

int size = pow(2, i);

int \*arr = (int \*)calloc(size, sizeof(int));

FILE \*fpr, \*fwr1, \*fwr2;

char \*str = (char \*)calloc(100, sizeof(char));

sprintf(str, "E:\\collage\\c\\sorting\\resources\\nums2^%d.txt", i);

fpr = fopen(str, "r");

for (int j = 0; j < size; j++)

{

fscanf(fpr, "%d", &arr[j]);

}

sort(arr, size);

char \*str1 = (char \*)calloc(100, sizeof(char));

char \*str2 = (char \*)calloc(100, sizeof(char));

sprintf(str1, "E:\\collage\\c\\sorting\\resources\\nums2^%dinc.txt", i);

sprintf(str2, "E:\\collage\\c\\sorting\\resources\\nums2^%ddec.txt", i);

fwr1 = fopen(str1, "w");

fwr2 = fopen(str2, "w");

for (int j = 0; j < size; j++)

{

fprintf(fwr1, "%d\n", arr[j]);

fprintf(fwr2, "%d\n", arr[size - 1 - j]);

}

}

return 0;

}

/\*-------------------------\*/

/\* ------- graph\_sort.c ------- \*/

(this will check sorting algorithm of only one file)

#include <stdio.h>

#include <stdlib.h>

// For debugging purposes

void display(int arr[], int size)

{

for (int i = 0; i < size; i++)

printf("%d, ", arr[i]);

printf("\n");

}

// Returns a memory copy of an array

int \*duplicate\_array(int arr\_orig[], int size)

{

int \*arr\_copy = malloc(sizeof(int) \* size);

if (arr\_copy == NULL)

{

fprintf(stderr, "Failed to allocate duplicate of size %lu bytes.\n", sizeof(int) \* size);

exit(EXIT\_FAILURE);

}

for (int i = 0; i < size; i++)

arr\_copy[i] = arr\_orig[i];

return arr\_copy;

}

void swap(int \*arr, int i, int j)

{

int n = arr[j];

arr[j] = arr[i];

arr[i] = n;

}

// Comparison count for each of the algorithms

// is updated whenever a comparison is made, regardless

// of whether that resulted in a swap.

unsigned long ccount;

// TODO: Implement the following, taking care to

// update `ccount` as appropriate.

void select\_sort(int arr[], int l, int r)

{

ccount = 0;

for (int i = l; i <= r; i++)

{

int max = l;

int last = r - i;

for (int j = l + 1; j <= last; j++)

{

ccount++;

if (arr[max] < arr[j])

{

max = j;

}

}

swap(arr, max, last);

}

}

void bubble\_sort(int arr[], int l, int r)

{

ccount = 0;

int sort = 1;

for (int i = l; i <= r; i++)

{

sort = 1;

for (int j = 1; j <= r - i; j++)

{

if (arr[j] < arr[j - 1])

{

swap(arr, j - 1, j);

sort = 0;

}

ccount++;

}

if (sort == 1)

break;

}

}

void insert\_sort(int arr[], int l, int r)

{

ccount = 0;

for (int i = l + 1; i <= r; i++)

{

int key = arr[i];

int j;

int c = 0;

for (j = i - 1; j >= 0 && arr[j] > key; j--)

{

ccount++;

arr[j + 1] = arr[j];

c = 1;

}

if (c == 0)

{

ccount++;

}

arr[j + 1] = key;

}

}

void merge(int \*arr, int l, int r)

{

int m = l + (r - l) / 2;

if (r > l)

{

merge(arr, l, m);

merge(arr, m + 1, r);

int n1 = m - l + 1;

int n2 = r - m;

int L[n1], R[n2];

for (int i = 0; i < n1; i++)

L[i] = arr[l + i];

for (int j = 0; j < n2; j++)

R[j] = arr[m + 1 + j];

int i = 0, j = 0, k = l;

while (i < n1 && j < n2)

{

ccount++;

if (L[i] <= R[j])

{

arr[k] = L[i];

i++;

}

else

{

arr[k] = R[j];

j++;

}

k++;

}

while (i < n1)

{

arr[k] = L[i];

i++;

k++;

}

while (j < n2)

{

arr[k] = R[j];

j++;

k++;

}

}

}

void merge\_sort(int arr[], int l, int r)

{

ccount = 0;

merge(arr, l, r);

}

void quick(int \*arr, int l, int r)

{

if (l < r)

{

int max = arr[r];

int i = l - 1;

for (int j = l; j <= r - 1; j++)

{

ccount++;

if (arr[j] < max)

{

i++;

swap(arr, i, j);

}

}

swap(arr, i + 1, r);

int pivot = i + 1;

quick(arr, l, pivot - 1);

quick(arr, pivot + 1, r);

}

}

void quick\_sort(int arr[], int l, int r)

{

ccount = 0;

quick(arr, l, r);

}

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

{

if (argc != 4)

{

fprintf(stderr, "Usage:\n%s [INPUT SIZE] [INPUT FILE] [OUTPUT STATS CSV]\n", argv[0]);

exit(EXIT\_FAILURE);

}

int size = 0;

if (sscanf(argv[1], "%d", &size) != 1 || size < 0)

{

fprintf(stderr, "Invalid size supplied.\n");

exit(EXIT\_FAILURE);

}

int \*arr = malloc(sizeof(int) \* size);

if (arr == NULL)

{

fprintf(stderr, "Failed to allocate array of size %lu bytes.\n", sizeof(int) \* size);

exit(EXIT\_FAILURE);

}

FILE \*fin = fopen(argv[2], "r");

if (fin == NULL)

{

fprintf(stderr, "Could not open %s\n", argv[2]);

exit(EXIT\_FAILURE);

}

// Input file has each integer on a new line

for (int i = 0; i < size; i++)

{

fscanf(fin, "%d", arr + i);

}

fclose(fin);

// Output file format:

// sort\_algo,input\_size,num\_comparisons

FILE \*fout = fopen(argv[3], "a");

int \*arr\_copy;

// Selection Sort

arr\_copy = duplicate\_array(arr, size);

ccount = 0;

select\_sort(arr\_copy, 0, size - 1);

fprintf(fout, "%s,%d,%lu\n", "select", size, ccount);

free(arr\_copy);

// Bubble Sort

arr\_copy = duplicate\_array(arr, size);

ccount = 0;

bubble\_sort(arr\_copy, 0, size - 1);

fprintf(fout, "%s,%d,%lu\n", "bubble", size, ccount);

free(arr\_copy);

// Insertion Sort

arr\_copy = duplicate\_array(arr, size);

ccount = 0;

insert\_sort(arr\_copy, 0, size - 1);

fprintf(fout, "%s,%d,%lu\n", "insert", size, ccount);

free(arr\_copy);

// Merge Sort

arr\_copy = duplicate\_array(arr, size);

ccount = 0;

merge\_sort(arr\_copy, 0, size - 1);

fprintf(fout, "%s,%d,%lu\n", "merge", size, ccount);

free(arr\_copy);

// Quick Sort

arr\_copy = duplicate\_array(arr, size);

ccount = 0;

quick\_sort(arr\_copy, 0, size - 1);

fprintf(fout, "%s,%d,%lu\n", "quick", size, ccount);

free(arr\_copy);

fclose(fout);

free(arr);

return EXIT\_SUCCESS;

}

/\* ---------------------- \*/

/\*--------graph\_sort\_3.c-------\*/

(this will check sorting algorithm of 3 files at a time(random,ascending &decending))

#include <stdio.h>

#include <stdlib.h>

// For debugging purposes

void display(int arr[], int size)

{

for (int i = 0; i < size; i++)

printf("%d, ", arr[i]);

printf("\n");

}

// Returns a memory copy of an array

int \*duplicate\_array(int arr\_orig[], int size)

{

int \*arr\_copy = malloc(sizeof(int) \* size);

if (arr\_copy == NULL)

{

fprintf(stderr, "Failed to allocate duplicate of size %lu bytes.\n", sizeof(int) \* size);

exit(EXIT\_FAILURE);

}

for (int i = 0; i < size; i++)

arr\_copy[i] = arr\_orig[i];

return arr\_copy;

}

void swap(int \*arr, int i, int j)

{

int n = arr[j];

arr[j] = arr[i];

arr[i] = n;

}

// Comparison count for each of the algorithms

// is updated whenever a comparison is made, regardless

// of whether that resulted in a swap.

unsigned long ccount;

// TODO: Implement the following, taking care to

// update `ccount` as appropriate.

void select\_sort(int arr[], int l, int r)

{

ccount = 0;

for (int i = l; i <= r; i++)

{

int max = l;

int last = r - i;

for (int j = l + 1; j <= last; j++)

{

ccount++;

if (arr[max] < arr[j])

{

max = j;

}

}

swap(arr, max, last);

}

}

void bubble\_sort(int arr[], int l, int r)

{

ccount = 0;

int sort = 1;

for (int i = l; i <= r; i++)

{

sort = 1;

for (int j = 1; j <= r - i; j++)

{

if (arr[j] < arr[j - 1])

{

swap(arr, j - 1, j);

sort = 0;

}

ccount++;

}

if (sort == 1)

break;

}

}

void insert\_sort(int arr[], int l, int r)

{

ccount = 0;

for (int i = l + 1; i <= r; i++)

{

int key = arr[i];

int j;

int c = 0;

for (j = i - 1; j >= 0 && arr[j] > key; j--)

{

ccount++;

arr[j + 1] = arr[j];

c = 1;

}

if (c == 0)

{

ccount++;

}

arr[j + 1] = key;

}

}

void merge(int \*arr, int l, int r)

{

int m = l + (r - l) / 2;

if (r > l)

{

merge(arr, l, m);

merge(arr, m + 1, r);

int n1 = m - l + 1;

int n2 = r - m;

int L[n1], R[n2];

for (int i = 0; i < n1; i++)

L[i] = arr[l + i];

for (int j = 0; j < n2; j++)

R[j] = arr[m + 1 + j];

int i = 0, j = 0, k = l;

while (i < n1 && j < n2)

{

ccount++;

if (L[i] <= R[j])

{

arr[k] = L[i];

i++;

}

else

{

arr[k] = R[j];

j++;

}

k++;

}

while (i < n1)

{

arr[k] = L[i];

i++;

k++;

}

while (j < n2)

{

arr[k] = R[j];

j++;

k++;

}

}

}

void merge\_sort(int arr[], int l, int r)

{

ccount = 0;

merge(arr, l, r);

}

void quick(int \*arr, int l, int r)

{

if (l < r)

{

int max = arr[r];

int i = l - 1;

for (int j = l; j <= r - 1; j++)

{

ccount++;

if (arr[j] < max)

{

i++;

swap(arr, i, j);

}

}

swap(arr, i + 1, r);

int pivot = i + 1;

quick(arr, l, pivot - 1);

quick(arr, pivot + 1, r);

}

}

void quick\_sort(int arr[], int l, int r)

{

ccount = 0;

quick(arr, l, r);

}

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

{

if (argc != 6)

{

fprintf(stderr, "Usage:\n%s [INPUT SIZE] [INPUT FILE] [ASCENDING INPUT FILE] [DESCENDING INPUT FILE] [OUTPUT STATS CSV]\n", argv[0]);

exit(EXIT\_FAILURE);

}

int size = 0;

if (sscanf(argv[1], "%d", &size) != 1 || size < 0)

{

fprintf(stderr, "Invalid size supplied.\n");

exit(EXIT\_FAILURE);

}

int \*arr = malloc(sizeof(int) \* size);

int \*arrasc = malloc(sizeof(int) \* size);

int \*arrdes = malloc(sizeof(int) \* size);

if (arr == NULL || arrasc == NULL || arrdes == NULL)

{

fprintf(stderr, "Failed to allocate array of size %lu bytes.\n", sizeof(int) \* size);

exit(EXIT\_FAILURE);

}

FILE \*fin = fopen(argv[2], "r");

FILE \*finasc = fopen(argv[3], "r");

FILE \*findes = fopen(argv[4], "r");

if (fin == NULL || finasc == NULL || findes == NULL)

{

fprintf(stderr, "Could not open %s\n", argv[2]);

exit(EXIT\_FAILURE);

}

// Input file has each integer on a new line

for (int i = 0; i < size; i++)

{

fscanf(fin, "%d", arr + i);

fscanf(finasc, "%d", arrasc + i);

fscanf(findes, "%d", arrdes + i);

}

fclose(fin);

fclose(finasc);

fclose(findes);

// Output file format:

// sort\_algo,input\_size,num\_comparisons

FILE \*fout = fopen(argv[5], "a");

int \*arr\_copy;

int \*arrasc\_copy;

int \*arrdes\_copy;

// Selection Sort

arr\_copy = duplicate\_array(arr, size);

arrasc\_copy = duplicate\_array(arrasc, size);

arrdes\_copy = duplicate\_array(arrdes, size);

ccount = 0;

select\_sort(arr\_copy, 0, size - 1);

fprintf(fout, "%s,%d,%lu\n", "select", size, ccount);

select\_sort(arrasc\_copy, 0, size - 1);

fprintf(fout, "%s,%d,%lu\n", "select", size, ccount);

select\_sort(arrdes\_copy, 0, size - 1);

fprintf(fout, "%s,%d,%lu\n", "select", size, ccount);

free(arr\_copy);

free(arrasc\_copy);

free(arrdes\_copy);

// Bubble Sort

arr\_copy = duplicate\_array(arr, size);

arrasc\_copy = duplicate\_array(arrasc, size);

arrdes\_copy = duplicate\_array(arrdes, size);

ccount = 0;

bubble\_sort(arr\_copy, 0, size - 1);

fprintf(fout, "%s,%d,%lu\n", "bubble", size, ccount);

bubble\_sort(arrasc\_copy, 0, size - 1);

fprintf(fout, "%s,%d,%lu\n", "bubble", size, ccount);

bubble\_sort(arrdes\_copy, 0, size - 1);

fprintf(fout, "%s,%d,%lu\n", "bubble", size, ccount);

free(arr\_copy);

free(arrasc\_copy);

free(arrdes\_copy);

// Insertion Sort

arr\_copy = duplicate\_array(arr, size);

arrasc\_copy = duplicate\_array(arrasc, size);

arrdes\_copy = duplicate\_array(arrdes, size);

ccount = 0;

insert\_sort(arr\_copy, 0, size - 1);

fprintf(fout, "%s,%d,%lu\n", "insert", size, ccount);

insert\_sort(arrasc\_copy, 0, size - 1);

fprintf(fout, "%s,%d,%lu\n", "insert", size, ccount);

insert\_sort(arrdes\_copy, 0, size - 1);

fprintf(fout, "%s,%d,%lu\n", "insert", size, ccount);

free(arr\_copy);

free(arrasc\_copy);

free(arrdes\_copy);

// Merge Sort

arr\_copy = duplicate\_array(arr, size);

arrasc\_copy = duplicate\_array(arrasc, size);

arrdes\_copy = duplicate\_array(arrdes, size);

ccount = 0;

merge\_sort(arr\_copy, 0, size - 1);

fprintf(fout, "%s,%d,%lu\n", "merge", size, ccount);

merge\_sort(arrasc\_copy, 0, size - 1);

fprintf(fout, "%s,%d,%lu\n", "merge", size, ccount);

merge\_sort(arrdes\_copy, 0, size - 1);

fprintf(fout, "%s,%d,%lu\n", "merge", size, ccount);

free(arr\_copy);

free(arrasc\_copy);

free(arrdes\_copy);

// Quick Sort

arr\_copy = duplicate\_array(arr, size);

arrasc\_copy = duplicate\_array(arrasc, size);

arrdes\_copy = duplicate\_array(arrdes, size);

ccount = 0;

quick\_sort(arr\_copy, 0, size - 1);

fprintf(fout, "%s,%d,%lu\n", "quick", size, ccount);

quick\_sort(arrasc\_copy, 0, size - 1);

fprintf(fout, "%s,%d,%lu\n", "quick", size, ccount);

quick\_sort(arrdes\_copy, 0, size - 1);

fprintf(fout, "%s,%d,%lu\n", "quick", size, ccount);

free(arr\_copy);

free(arrasc\_copy);

free(arrdes\_copy);

fclose(fout);

free(arr);

free(arrasc);

free(arrdes);

return EXIT\_SUCCESS;

}

/\*------------------\*/

/\*-------stats.py--------\*/

(For generating the graph)

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

stats = pd.read\_csv("stats.csv")

stats["mean\_num\_comparisons"] = 0.0

for algo in stats["sort\_algo"].unique():

stats\_algo = stats[stats["sort\_algo"] == algo]

for N in stats\_algo["input\_size"].unique():

stats\_algo\_N = stats\_algo[stats\_algo["input\_size"] == N]

stats.loc[(stats["sort\_algo"] == algo) & (stats["input\_size"] == N),

("mean\_num\_comparisons")] = stats\_algo\_N["num\_comparisons"].mean()

stats\_mean = stats.drop("num\_comparisons", axis=1, inplace=False)

stats\_mean.drop\_duplicates(inplace=True)

stats\_mean["mean\_num\_comparisons\_log"] = np.log2(

stats\_mean["mean\_num\_comparisons"])

stats\_mean["input\_size\_log"] = np.log2(stats\_mean["input\_size"])

sns.lineplot(x="input\_size\_log", y="mean\_num\_comparisons\_log",

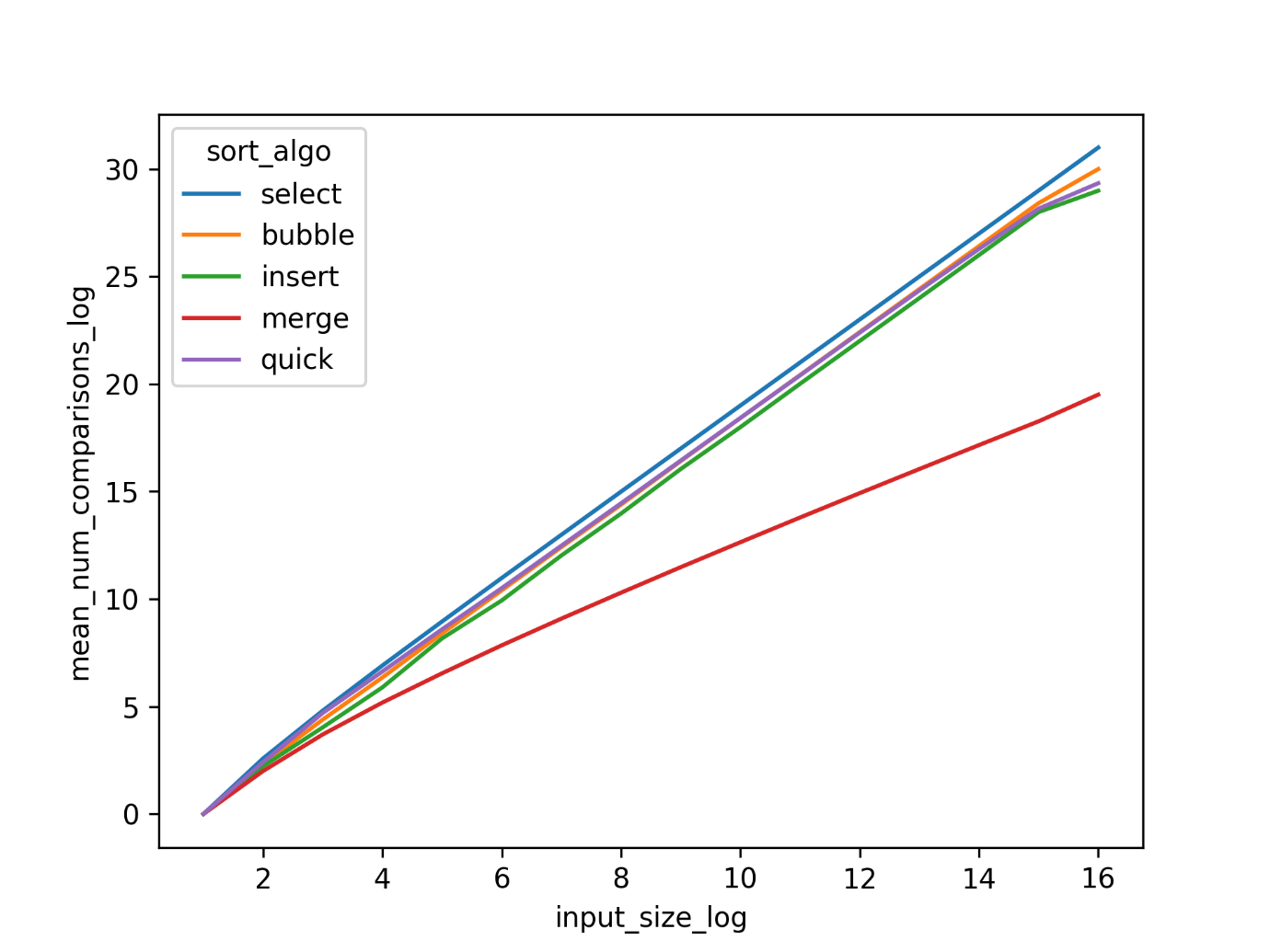
hue="sort\_algo", data=stats\_mean)

plt.savefig("stats.png", dpi=300)

plt.show()

/\*-------------------------\*/

Comparison Graph:



Conclusion:

The bubble sort ,selection sort, insertion sort and quick sort all takes O(n2) time in the worst case, merge sort take O(nlogn) time in worst case, where n is the number of elements in the input array.

Limitations and assumptions for this algorithm include:

1. Till 215 elements you can use graph\_sort\_3 but for greater no of elements you have to use graph\_sort.
2. In command line we have to write:

gcc filename // for compilation

For graph\_sort:

./execution\_file\_name random\_file\_location writing\_file \_location

For graph\_sort\_3:

./execution\_file\_name random\_file\_location ascending\_file\_location descending\_file\_location writing\_file \_location.