Practical - 2

2 (a) Design and implement C Program to sort a given set of n integer elements using Merge Sort method and compute its time complexity. Run the program for varied values of n, and record the time taken to sort. Plot a graph of the time taken versus n. The elements can be read from a file or can be generated using the random number generator..

Pseudo code -

```
FUNCTION Merge(arr, left, mid, right)
n1 ← mid - left + 1
n2 ← right - mid
L ← new array of size n1
R ← new array of size n2
FOR i FROM 0 TO n1 - 1
L[i] ← arr[left + i]
FOR j FROM 0 TO n2 - 1
R[j] ← arr[mid + 1 + j]
i ← 0, j ← 0, k ← left
WHILE i < n1 AND j < n2
IF L[i] ≤ R[j]
arr[k] ← L[i]
i ← i + 1
ELSE
arr[k] ← R[j]
j ← j + 1
k ← k + 1
WHILE i < n1
arr[k] ← L[i]
i ← i + 1
k ← k + 1
WHILE j < n2
arr[k] ← R[j]
j ← j + 1
k ← k + 1
END FUNCTION
FUNCTION MergeSort(arr, left, right)
IF left < right
mid ← (left + right) / 2
CALL MergeSort(arr, left, mid)
CALL MergeSort(arr, mid + 1, right)
CALL Merge(arr, left, mid, right)
END FUNCTION
FUNCTION GenerateRandomArray(arr, size)
FOR i FROM 0 TO size - 1
arr[i] ← RANDOM INTEGER BETWEEN 0 AND 99END FUNCTION
MAIN PROGRAM
PROMPT "Enter the number of elements: "
READ n
arr ← new array of size n
temp ← new array of size n
IF memory allocation fails
PRINT "Memory allocation failed"
EXIT
CALL GenerateRandomArray(arr, n)
total_time ← 0
FOR run FROM 1 TO 1000
COPY arr INTO temp
start_time ← CURRENT CLOCK TIME
CALL MergeSort(temp, 0, n - 1)
end_time ← CURRENT CLOCK TIME
```

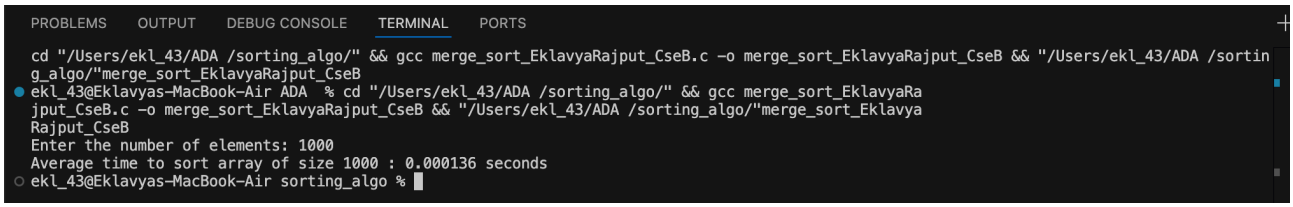
```
elapsed_time ← (end_time - start_time)
total_time ← total_time + elapsed_time
average_time ← total_time / 1000
PRINT "Average time to sort", "whose size is n:", average_time, "seconds"
FREE arr
FREE temp
END PROGRAM
```

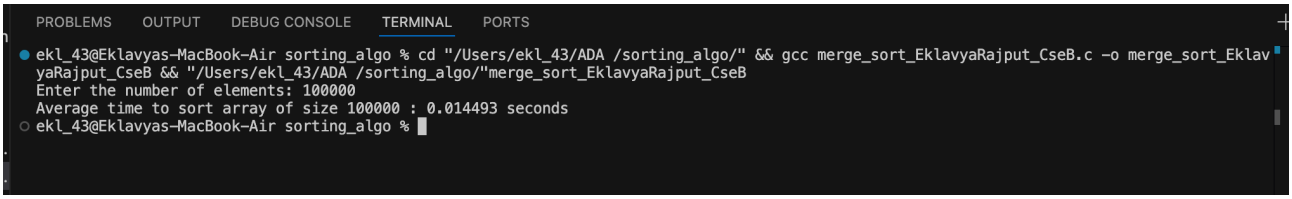
C code -

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
void merge(int arr[], int l, int m, int r) {
int i, j, k;
int n1 = m - l + 1;
int n2 = r - m;
int *L = (int *)malloc(n1 * sizeof(int));
int *R = (int *)malloc(n2 * sizeof(int));
for (i = 0; i < n1; i++)
L[i] = arr[l + i];
for (j = 0; j < n2; j++)
R[j] = arr[m + 1 + j];
i = 0;
j = 0;
k = l;while (i < n1 && j < n2) {
if (L[i] <= R[j]) {
arr[k++] = L[i++];
} else {
arr[k++] = R[j++];
}
while (i < n1) {
arr[k++] = L[i++];
while (j < n2) {
arr[k++] = R[j++];
}
}
}
free(L);
free(R);
}
void mergeSort(int arr[], int left, int right) {
if (left < right) {
int m = left + (right - left) / 2;
mergeSort(arr, left, m);
mergeSort(arr, m + 1, right);
merge(arr, left, m, right);
}
}
void generateRandomArray(int arr[], int n) {
for (int i = 0; i < n; i++) {
arr[i] = rand() % 100;
}
}
int main() {
int n;
printf("Enter the number of elements: ");
scanf("%d", &n);
int *arr = (int *)malloc(n * sizeof(int));
int *temp = (int *)malloc(n * sizeof(int)); // For copying original array
if (arr == NULL || temp == NULL) {
printf("Memory allocation failed\n");
return 1;
}
generateRandomArray(arr, n);
double total_time = 0;
```

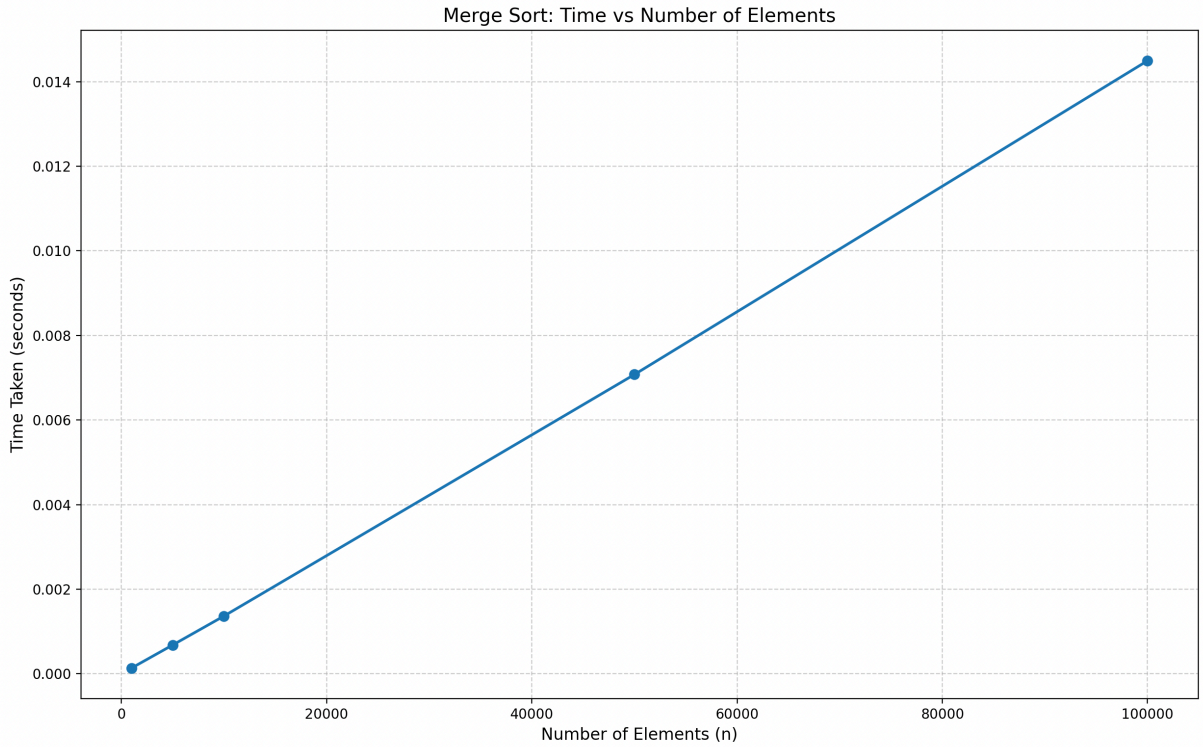
```
for (int i = 0; i < 1000; i++) {
for (int j = 0; j < n; j++) {
temp[j] = arr[j];
clock_t start = clock();mergeSort(temp, 0, n - 1);
clock_t end = clock();
total_time += ((double)(end - start)) / CLOCKS_PER_SEC;
}
printf("Average time to sort elements whose size is %d : %f seconds\n", n,
total_time / 1000.0);
free(arr);
free(temp);
return 0;
}}
```

Output:





Graph:


Merge Sort: Time vs Number of Elements
Time Taken (seconds)
0.014
0.012
0.010
0.008
0.006
0.004
0.002
0.000
0
20000
40000
60000
80000
100000
Number of Elements (n)

2 (b)Design and implement C Program to sort a given set of n integer elements using Quick Sort method and compute its time complexity. Run the program for varied values of n, and record the time taken to sort. Plot a graph of the time taken versus n. The elements can be read from a file or can be generated using the random number generator

Pseudo code

```
FUNCTION GenerateRandomArray(arr, size)
FOR i FROM 0 TO size - 1
arr[i] ← RANDOM INTEGER BETWEEN 0 AND 99
END FUNCTION
FUNCTION Partition(arr, low, high)
pivot ← arr[high]
i ← low - 1
FOR j FROM low TO high - 1
IF arr[j] ≤ pivot THEN
i ← i + 1
SWAP arr[i] WITH arr[j]
SWAP arr[i + 1] WITH arr[high]
RETURN i + 1
END FUNCTION
FUNCTION QuickSort(arr, low, high)
IF low < high THEN
pi ← Partition(arr, low, high)
CALL QuickSort(arr, low, pi - 1)
CALL QuickSort(arr, pi + 1, high)
END FUNCTION
MAIN PROGRAM
PROMPT "Enter the number of elements: "
READ n
arr ← new array of size n
IF memory allocation fails THEN
PRINT "Memory allocation failed"
EXIT PROGRAM
INITIALIZE random number generator
START timer
FOR i FROM 1 TO 1000
CALL GenerateRandomArray(arr, n)
CALL QuickSort(arr, 0, n - 1)
END timer
time_taken ← (end_time - start_time) / CLOCKS_PER_SEC / 1000.0PRINT "Average time to sort", n, "elements using
Quick Sort:", time_taken,
"seconds"
FREE arr
END PROGRAM
```

C Code -

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
void generateRandomArray(int arr[], int n) {
for (int i = 0; i < n; i++) {
arr[i] = rand() % 100;
}
}
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 temp = arr[i];
```

```
arr[i] = arr[j];
arr[j] = temp;
}
}
int temp = arr[i + 1];
arr[i + 1] = arr[high];
arr[high] = temp;
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 n;
printf("Enter the number of elements: ");
scanf("%d", &n);
int *arr = (int *)malloc(n * sizeof(int));
if (arr == NULL) {printf("Memory allocation failed\n");
return 1;
}
srand(time(NULL));
clock_t start = clock();
for (int i = 0; i < 1000; i++) {
generateRandomArray(arr, n);
quickSort(arr, 0, n - 1);
}
clock_t end = clock();
double time_taken = ((double)(end - start)) / CLOCKS_PER_SEC / 1000.0;
printf("Average time to sort %d elements using Quick Sort: %f seconds\n", n,
time_taken);
free(arr);
return 0;
}
```

Output:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

ekl_43@Eklavyas-MacBook-Air sorting_algo % cd "/Users/ekl_43/ADA /sorting_algo/" && gcc quickSort_Ek
lavyaRajput_CseB.c -o quickSort_EklavyaRajput_CseB && "/Users/ekl_43/ADA /sorting_algo/"quickSort_Ek
lavyaRajput_CseB
Enter the number of elements: 1000
Average time to sort 1000 elements using Quick Sort: 0.000108 seconds
ekl_43@Eklavyas-MacBook-Air sorting_algo %
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

ekl_43@Eklavyas-MacBook-Air sorting_algo % cd "/Users/ekl_43/ADA /sorting_algo/" && gcc quickSort_EklavyaRajput_CseB.c -o quickSort_Eklavya
Rajput_CseB && "/Users/ekl_43/ADA /sorting_algo/"quickSort_EklavyaRajput_CseB
Enter the number of elements: 5000
Average time to sort 5000 elements using Quick Sort: 0.000893 seconds
ekl_43@Eklavyas-MacBook-Air sorting_algo %
```

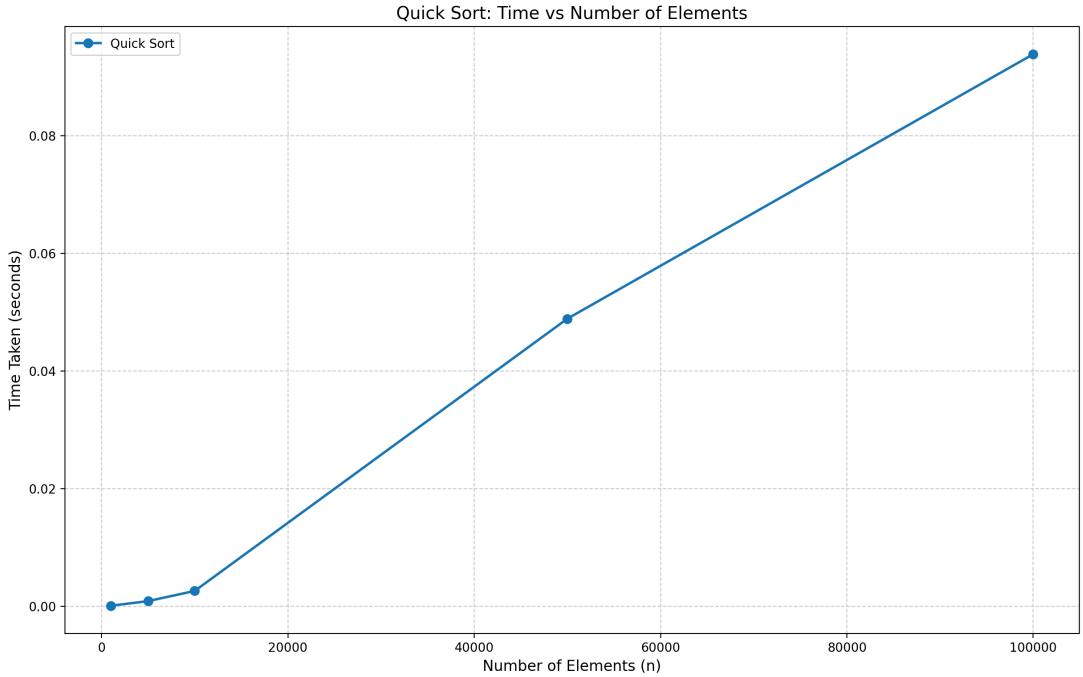
```
OBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

_43@Eklavyas-MacBook-Air sorting_algo % cd "/Users/ekl_43/ADA /sorting_algo/" && gcc quickSort_EklavyaRajput_CseB.c -o quickSort_Ekl
jput_CseB && "/Users/ekl_43/ADA /sorting_algo/"quickSort_EklavyaRajput_CseB
ter the number of elements: 10000
erage time to sort 10000 elements using Quick Sort: 0.002627 seconds
_43@Eklavyas-MacBook-Air sorting_algo %
```

```
OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

/ekl_43/ADA /sorting_algo/" && gcc quickSort_EklavyaRajput_CseB.c -o quickSort_EklavyaRajput_CseB && "/Users/ekl_43/AD
kSort_EklavyaRajput_CseB
me to sort 50000 elements using Quick Sort: 0.048910 seconds
avyas-MacBook-Air sorting_algo % cd "/Users/ekl_43/ADA /sorting_algo/" && gcc quickSort_EklavyaRajput_CseB.c -o quickS
B && "/Users/ekl_43/ADA /sorting_algo/"quickSort_EklavyaRajput_CseB
umber of elements:
```

Graph:



2(c) Design and implement C Program to sort a given set of n integer elements using Insertion Sort method and compute its time complexity. Run the program for varied values of n, and record the time taken to sort. Plot a graph of the time taken versus n. The elements can be read from a file or can be generated using the random number generator

Pseudo code -

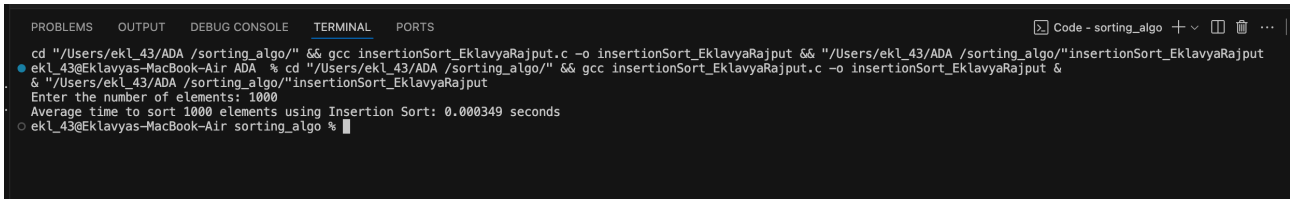
```
For i ← 0 to n - 1:
arr[i] ← random integer between 0 and 99
For i ← 0 to n - 1:
key ← arr[i]
j ← i - 1
While j ≥ 0 AND arr[j] < key:
arr[j + 1] ← arr[j]
j ← j - 1
arr[j + 1] ← key
Prompt user: "Enter the number of elements"
Read input into n
Allocate memory for array of size n
If memory allocation fails:
Print "Memory allocation failed"
Exit program
Seed random number generator with current time
Start timer
Repeat 1000 times:
generateRandomArray(arr, n)
insertionSort(arr, n)
Stop timer
Compute average time:
time_taken ← (end_time - start_time) / CLOCKS_PER_SEC / 1000.0
Print "Average time to sort n elements using Insertion Sort: time_taken seconds"
Free allocated memory
Exit program
```

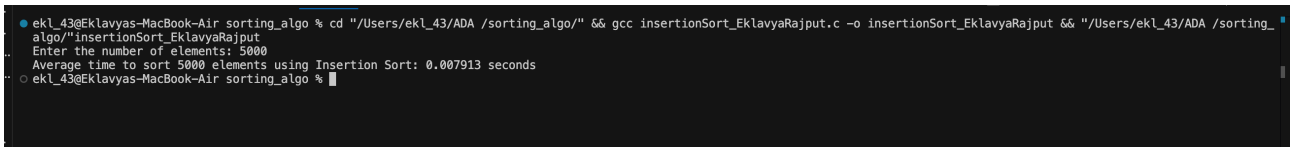
C code -

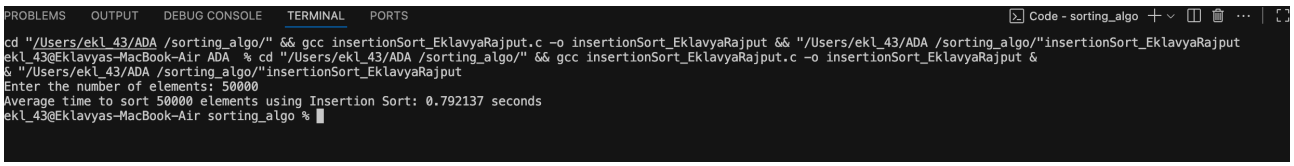
```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
void generateRandomArray(int arr[], int n) {
for (int i = 0; i < n; i++) {
arr[i] = rand() % 100;
}}
void insertionSort(int *A , int n ){
int key ;
for(int i = 0 ; i< n ; i++){
key = A[i];
int j = i-1;
while(A[j]< key){
A[j+1] = A[j];
j--;
}
A[j+1] = key ;
}
}
int main() {
int n;
printf("Enter the number of elements: ");
scanf("%d", &n);
int *arr = (int *)malloc(n * sizeof(int));
if (arr == NULL) {
printf("Memory allocation failed\n");
return 1;
```

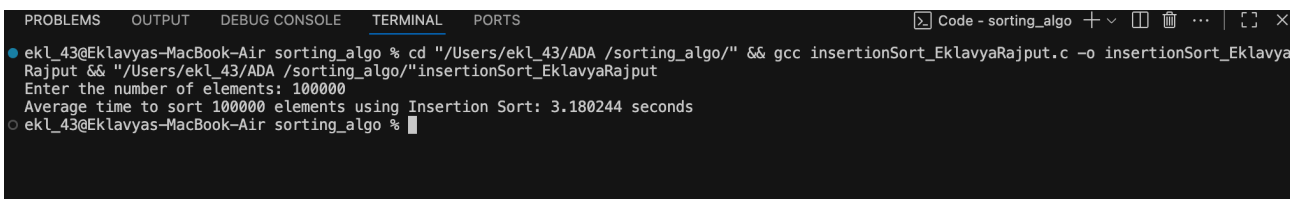
```
}
srand(time(NULL));
clock_t start = clock();
for (int i = 0; i < 1000; i++) {
generateRandomArray(arr, n);
insertionSort(arr, n);
}
clock_t end = clock();
double time_taken = ((double)(end - start)) / CLOCKS_PER_SEC / 1000.0;
printf("Average time to sort %d elements using Insertion Sort: %f seconds\n", n,
time_taken);
free(arr);
return 0;
```

Output:

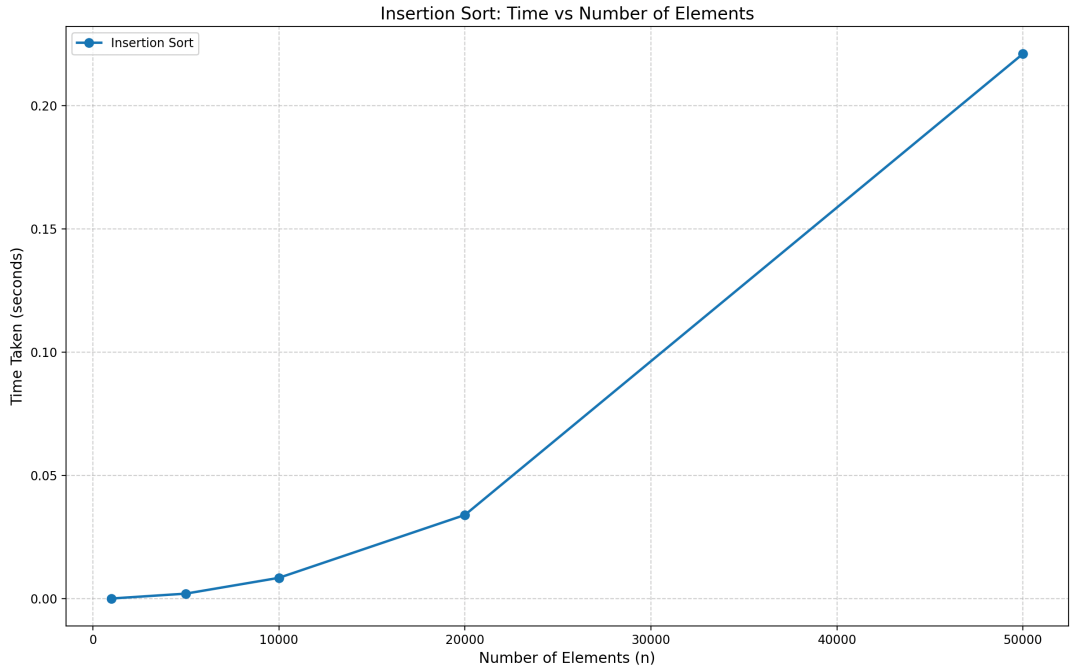








Graph:



2(d) Design and implement C Program to sort a given set of n integer elements using Selection Sort method and compute its time complexity. Run the program for varied values of n, and record the time taken to sort. Plot a graph of the time taken versus n. The elements can be read from a file or can be generated using the random number generator

Pseudo Code

For i ← 0 to n - 1:
arr[i] ← random integer between 0 and 99
For i ← 0 to n - 1:
indexOfMin ← i
For j ← 0 to n - 1:
If arr[j] < arr[indexOfMin]:
indexOfMin ← j
Swap arr[i] and arr[indexOfMin]
Prompt user: "Enter the number of elements"
Read input into n
Allocate memory for array of size n
If memory allocation fails:
Print "Memory allocation failed"
Exit program
Seed random number generator with current time
Start timer
Repeat 1000 times:
generateRandomArray(arr, n)

selectionSort(arr, n)
Stop timer
Compute average time:
timeTaken ← (endTime - startTime) / CLOCKS_PER_SEC / 1000.0
Print "Average time to sort n elements using Selection Sort: timeTaken seconds"
Free allocated memory
Exit program

C - code

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
void generateRandomArray(int arr[], int n) {
for (int i = 0; i < n; i++) {
arr[i] = rand() % 100;}
}
void selectionsort(int*A, int n){
int indexofmin ;
for(int i = 0 ; i< n ; i++){
indexofmin = i;
for(int j = 0 ; j< n; j++){
if(A[indexofmin]> A[j])
indexofmin = j;
}
// swap index of index of min with i
int temp = A[indexofmin];
A[indexofmin] = A[i];
A[i] = temp ;
}
}
int main() {
int n;
printf("Enter the number of elements: ");
scanf("%d", &n);
int *arr = (int *)malloc(n * sizeof(int));
if (arr == NULL) {
printf("Memory allocation failed\n");
return 1;
}
srand(time(NULL));
clock_t start = clock();
for (int i = 0; i < 1000; i++) {
generateRandomArray(arr, n);
selectionsort(arr, n);
}
clock_t end = clock();
double time_taken = ((double)(end - start)) / CLOCKS_PER_SEC / 1000.0;
printf("Average time to sort %d elements using Selection Sort: %f seconds\n", n,
time_taken);
free(arr);
return 0;
}
```

Output:

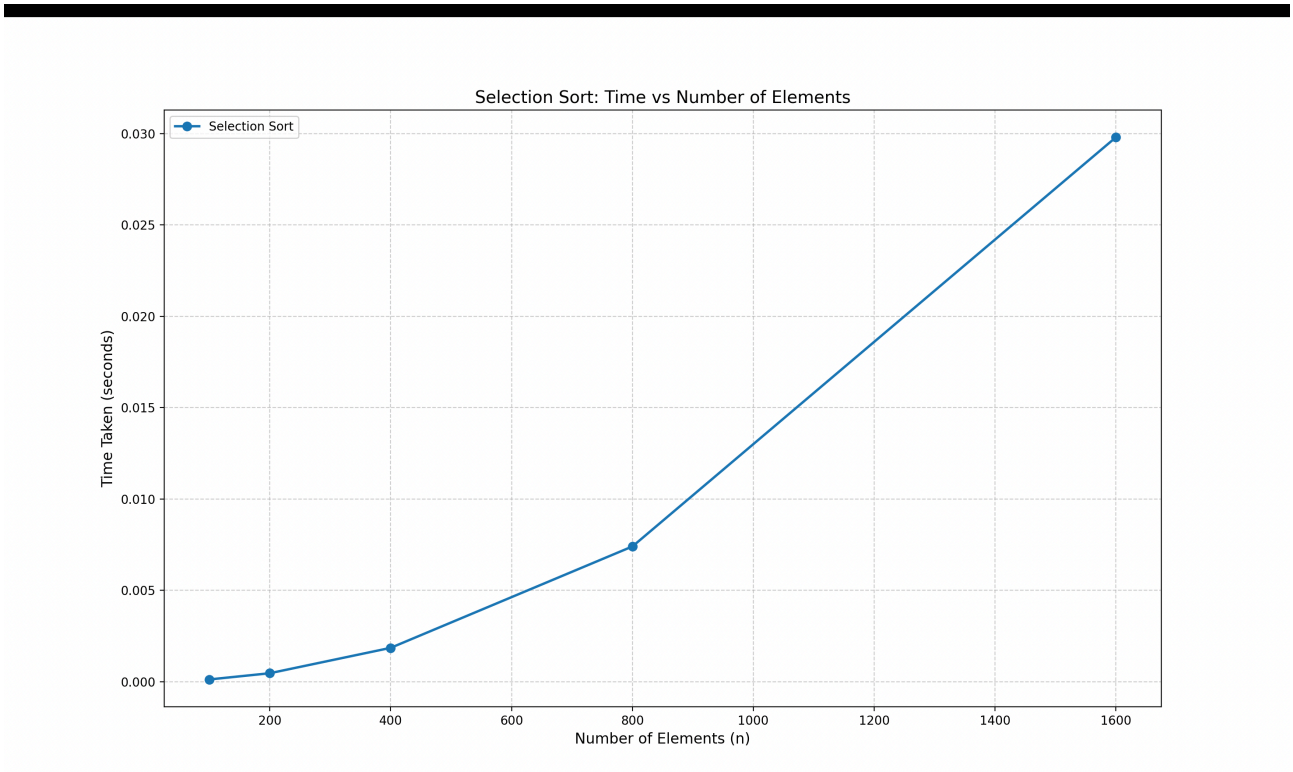
```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS
ekl_43@Eklavyas-MacBook-Air sorting_algo % cd "/Users/ekl_43/ADA /sorting_algo/" && gcc selectionSort_EklavyaRajput.c -o sele
ctionSort_EklavyaRajput && "/Users/ekl_43/ADA /sorting_algo/"selectionSort_EklavyaRajput
Enter the number of elements: 5000
Average time to sort 5000 elements using Selection Sort: 0.016839 seconds
ekl_43@Eklavyas-MacBook-Air sorting_algo %
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS
ekl_43@Eklavyas-MacBook-Air sorting_algo % cd "/Users/ekl_43/ADA /sorting_algo/" && gcc selectionSort_EklavyaRajput.c -o sele
ctionSort_EklavyaRajput && "/Users/ekl_43/ADA /sorting_algo/"selectionSort_EklavyaRajput
Enter the number of elements: 10000
Average time to sort 10000 elements using Selection Sort: 0.069120 seconds
ekl_43@Eklavyas-MacBook-Air sorting_algo %
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS
ekl_43@Eklavyas-MacBook-Air sorting_algo % cd "/Users/ekl_43/ADA /sorting_algo/" && gcc selectionSort_EklavyaRajput.c -o sele
ctionSort_EklavyaRajput && "/Users/ekl_43/ADA /sorting_algo/"selectionSort_EklavyaRajput
Enter the number of elements: 50000
Average time to sort 50000 elements using Selection Sort: 1.569469 seconds
ekl_43@Eklavyas-MacBook-Air sorting_algo %
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS
ekl_43@Eklavyas-MacBook-Air sorting_algo % cd "/Users/ekl_43/ADA /sorting_algo/" && gcc selectionSort_EklavyaRajput.c -o sele
ctionSort_EklavyaRajput && "/Users/ekl_43/ADA /sorting_algo/"selectionSort_EklavyaRajput
Enter the number of elements: 100000
Average time to sort 100000 elements using Selection Sort: 6.274439 seconds
ekl_43@Eklavyas-MacBook-Air sorting_algo %
```

Graph:



2 (e) Design and implement C Program to sort a given set of n integer elements using Bubble Sort method and compute its time complexity. Run the program for varied values of n, and record the time taken to sort. Plot a graph of the time taken versus n. The elements can be read from a file or can be generated using the random number generator

Pseudo code -

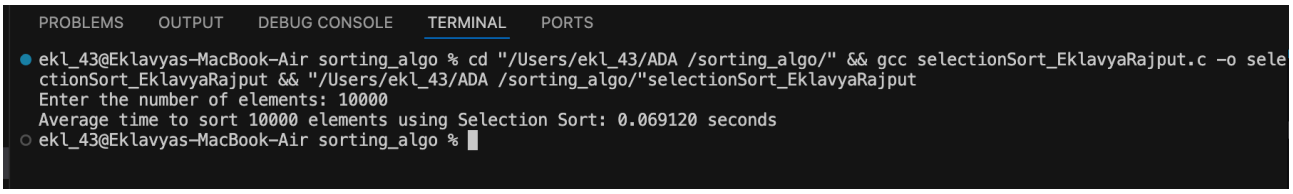
```
For i ← 0 to n - 1:
arr[i] ← random integer between 0 and 99
For i ← 0 to n - 2:
For j ← 0 to n - i - 2:
If arr[j] > arr[j + 1]: Swap arr[j] and arr[j + 1]
Prompt user: "Enter the number of elements"
Read input into n
// For ascending order
Allocate memory for array of size n
If memory allocation fails:
Print "Memory allocation failed"
Exit program
Seed random number generator with current time
Start timer
Repeat 1000 times:
generateRandomArray(arr, n)
bubbleSort(arr, n)
Stop timer
Compute average time:
timeTaken ← (endTime - startTime) / CLOCKS_PER_SEC / 1000.0
Print "Average time to sort n elements using Bubble Sort: timeTaken seconds"
Free allocated memory
Exit program
```

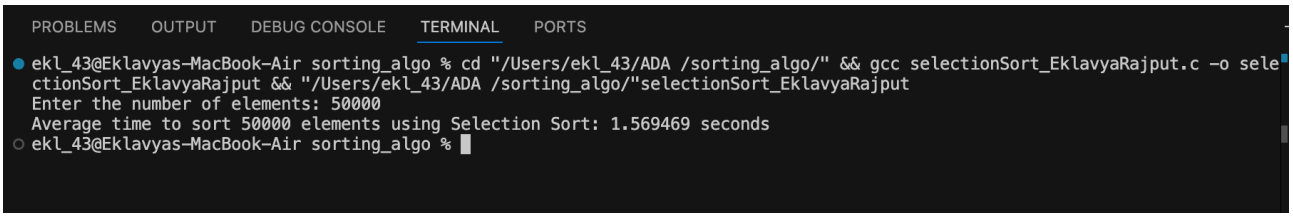
C code -

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
void generateRandomArray(int arr[], int n) {
for (int i = 0; i < n; i++) {
arr[i] = rand() % 100;
}
}
void bubblesort(int *A, int n){
int temp ;
for(int i = 0 ; i < n-1 ; i++ ){
for(int j = 0 ; j< n-i-1 ; j++){if(A[j]<A[j]+1){
temp = A[i];
A[i] = A[j+1];
A[j+1] = temp ;
}
}
}
}
int main() {
int n;
printf("Enter the number of elements: ");
scanf("%d", &n);
int *arr = (int *)malloc(n * sizeof(int));
if (arr == NULL) {
printf("Memory allocation failed\n");
return 1;
}
srand(time(NULL));
clock_t start = clock();
for (int i = 0; i < 1000; i++) {
```

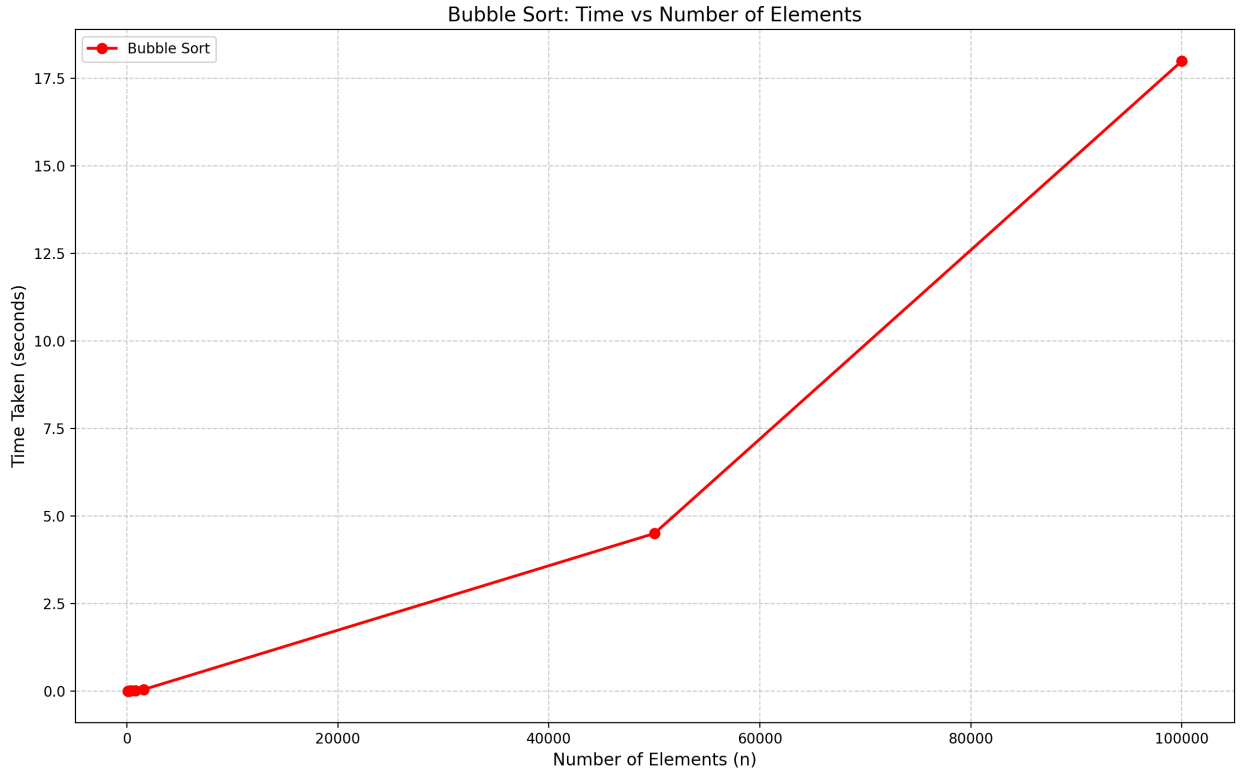
```
generateRandomArray(arr, n);
bubblesort(arr, n);
}
clock_t end = clock();
double time_taken = ((double)(end - start)) / CLOCKS_PER_SEC / 1000.0;
printf("Average time to sort %d elements using Bubble Sort: %f seconds\n", n,
time_taken);
free(arr);
return 0;
}
```

Output:





Graph:



Conclusion:

-
- **Bubble Sort, Selection Sort, and Insertion Sort** are all O (n^2)
  O(n^2) algorithms, which makes them inefficient for large input sizes.
- For **small arrays**, they work fine and are easy to implement.
- **Bubble Sort** is the slowest as it does many unnecessary swaps.
- **Selection Sort** reduces swaps but still makes n^2 n^2 comparisons.
- **Insertion Sort** performs best among the three for nearly sorted or small datasets because it minimizes unnecessary work.
- For larger datasets (e.g., 50,000 or 100,000 elements), all three take a long time (several seconds to minutes), so **more advanced algorithms like Merge Sort, Quick Sort, or built-in library sorts should be used**.