📂 Index

1(A) LINEAR SEACH

C CODE:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

// Linear Search function
int linearSearch(int arr[], int n, int key) {
    for (int i = 0; i < n; i++) {
        if (arr[i] == key)
            return i; // return index if found
    }
    return -1; // not found
}

int main() {
    int n, key, result;
    clock_t start, end;
    double cpu_time_used;

    printf("Enter number of elements (n): ");
    scanf("%d", &n);

    int *arr = (int *)malloc(n * sizeof(int));
    if (arr == NULL) {
        printf("Memory not allocated.\n");
        return 1;
    }


    for (int i = 0; i < n; i++) {
        arr[i] = i + 1;
    }

)
    key = n;

    start = clock();
    result = linearSearch(arr, n, key);
    end = clock();

    cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC;

    if (result != -1)
        printf("Element %d found at index %d\n", key, result);
    else
        printf("Element %d not found\n", key);

    printf("Time taken: %f seconds\n", cpu_time_used);

    free(arr);
    return 0;
```

}

PYTHON SCRIPT:

```
import matplotlib.pyplot as plt


n_values = [1000, 5000, 10000, 20000, 50000, 100000]
time_values = [0.00001, 0.00004, 0.00009, 0.00018, 0.00045, 0.0009]  # in seconds

plt.plot(n_values, time_values, marker='o')
plt.title("Linear Search: Time vs Number of Elements")
plt.xlabel("Number of Elements (n)")
plt.ylabel("Time Taken (seconds)")
plt.grid(True)
plt.show()
```

PSEUDO CODE:

C CODE:

```
Procedure LinearSearch(Array, n, key)
    For i ← 0 to n-1
        If Array[i] = key Then
            Return i   // Found at index i
        End If
    End For
    Return -1
End Procedure


Main Program:
    Input n
    Create Array of size n
    Fill Array with values (e.g., 1 to n)
    key ← n

    Start timer
    result ← LinearSearch(Array, n, key)
    Stop timer

    If result ≠ -1 Then
        Print "Element found at index", result
    Else
        Print "Element not found"
    End If

    Print "Time taken:", (Stop - Start)
End Program
```

PYTHON CODE:

Import matplotlib library

Initialize list n_values = [1000, 5000, 10000, 20000, 50000, 100000]
Initialize empty list time_values

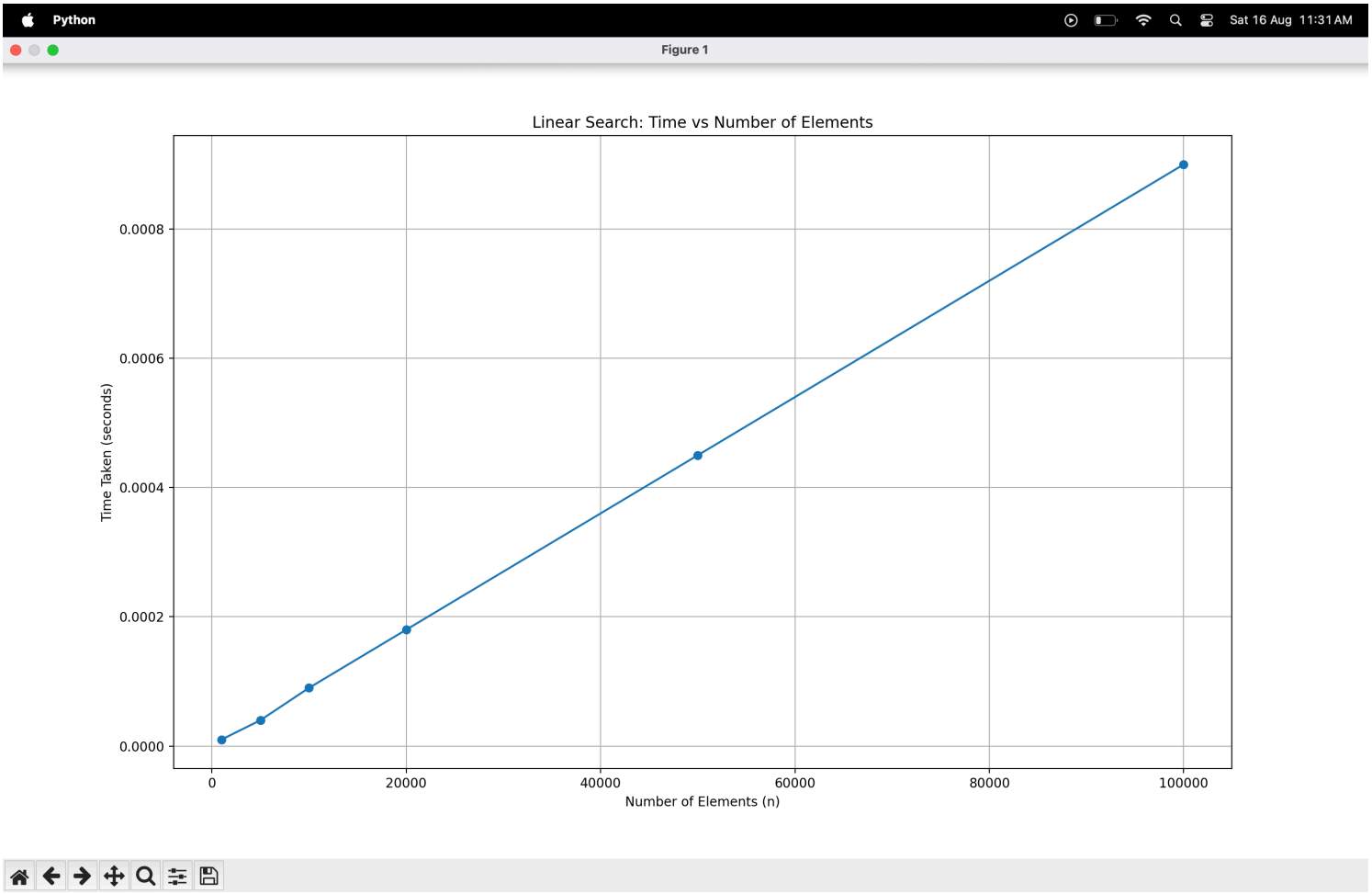
For each n in n_values:
    Create an array of numbers from 1 to n
    Set key = n (worst case)

    Start timer
    Perform Linear Search on array
    Stop timer

    Append (Stop - Start) to time_values

Plot graph with:
    X-axis = n_values
    Y-axis = time_values
    Title = "Linear Search: Time vs Number of Elements"
    Labels = ("Number of Elements", "Time Taken (seconds)")

Display graph



1(B) BINARY SEARCH

C CODE:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

// Binary Search function
int binarySearch(int arr[], int n, int key) {
    int low = 0, high = n - 1;
    while (low <= high) {
        int mid = (low + high) / 2;
        if (arr[mid] == key)
            return mid;
        else if (arr[mid] < key)
            low = mid + 1;
        else
            high = mid - 1;
    }
    return -1;
}

int main() {
    int n, key, result;
    clock_t start, end;
    double cpu_time_used;

    printf("Enter number of elements (n): ");
    scanf("%d", &n);

    int *arr = (int *)malloc(n * sizeof(int));
    if (arr == NULL) {
        printf("Memory not allocated.\n");
        return 1;
    }
    for (int i = 0; i < n; i++) {
        arr[i] = i + 1;
    }
    key = n;
    start = clock();
    result = binarySearch(arr, n, key);
    end = clock();

    cpu_time_used = ((double)(end - start))

    if (result != -1)
        printf("Element %d found at index %d\n", key, result);
    else
        printf("Element %d not found\n", key);

    printf("Time taken: %f seconds\n", cpu_time_used);

    free(arr);
    return 0;
}
```

PYTHON SCRIPT:

```
import matplotlib.pyplot as plt

# Example data (replace with your actual recorded values)
n_values = [1000, 5000, 10000, 20000, 50000, 100000]
time_values = [0.000001, 0.000002, 0.000003, 0.000003, 0.000004, 0.000004]  # in seconds

plt.plot(n_values, time_values, marker='o', label="Binary Search")
plt.title("Binary Search: Time vs Number of Elements")
plt.xlabel("Number of Elements (n)")
plt.ylabel("Time Taken (seconds)")
plt.grid(True)
plt.legend()
plt.show()
```

PSEUDO CODE:

C CODE:

```
Procedure BinarySearch(Array, n, key)
    low ← 0
    high ← n - 1
    While low ≤ high Do
        mid ← (low + high) / 2
        If Array[mid] = key Then
            Return mid        // Element found
        Else If Array[mid] < key Then
            low ← mid + 1
        Else
            high ← mid - 1
        End If
    End While
    Return -1
End Procedure


Main Program:
    Input n
    Create Array of size n
    Fill Array with values (1 to n) in sorted order
    key ← n

    Start timer
    result ← BinarySearch(Array, n, key)
    Stop timer

    If result ≠ -1 Then
        Print "Element found at index", result
    Else
        Print "Element not found"
    End If

    Print "Time taken:", (Stop - Start)
End Program
```

PYTHON CODE:

Import matplotlib library

Initialize list n_values = [1000, 5000, 10000, 20000, 50000, 100000]
Initialize empty list time_values

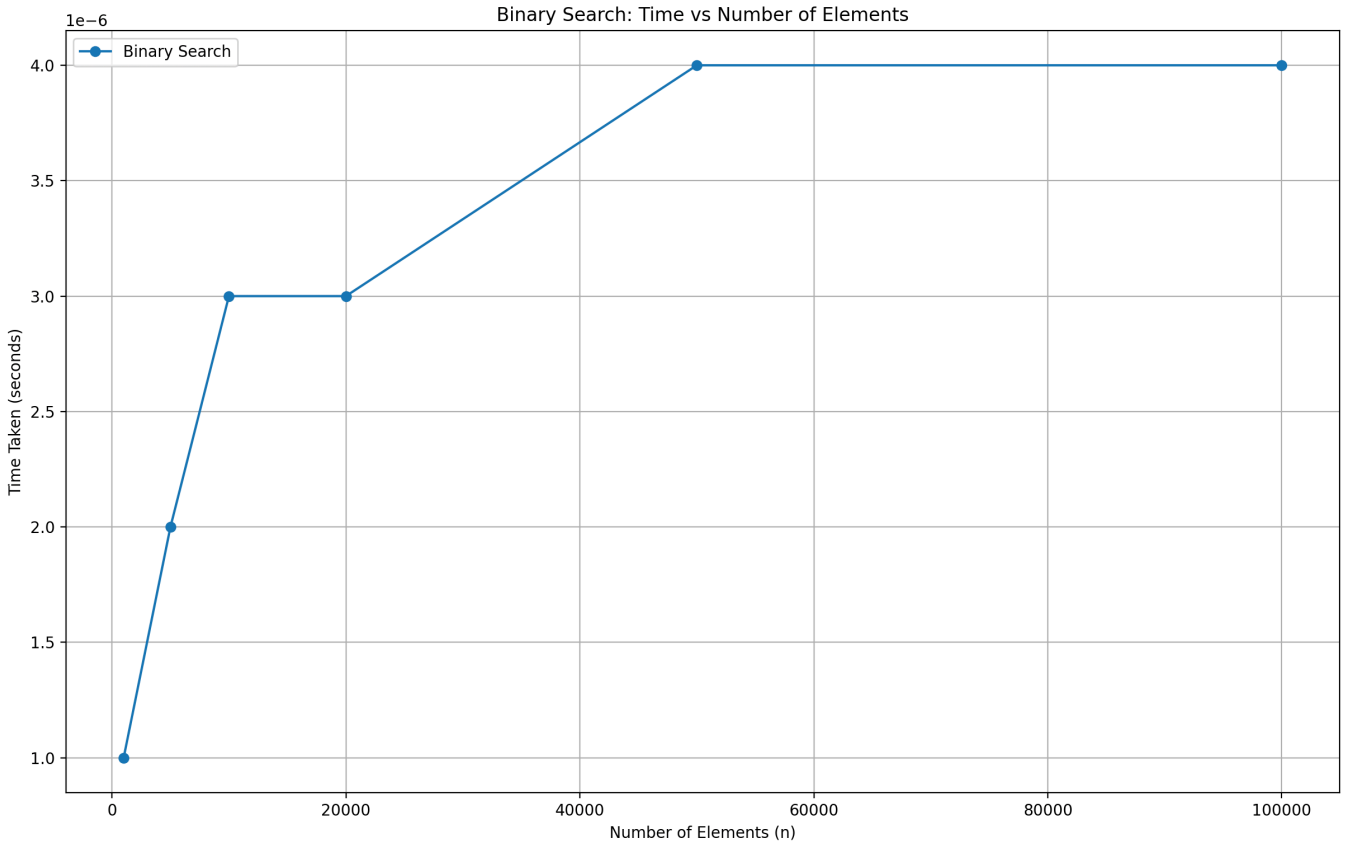
For each n in n_values:
    Create sorted array of numbers from 1 to n
    Set key = n (worst case)

    Start timer
    Perform BinarySearch on array
    Stop timer

    Append (Stop - Start) to time_values

Plot graph with:
    X-axis = n_values
    Y-axis = time_values
    Title = "Binary Search: Time vs Number of Elements"
    Labels = ("Number of Elements", "Time Taken (seconds)")

Display graph



CONCLUSION:

From the two experiments, we clearly see a difference in how **Linear Search** and **Binary Search** perform as the number of elements (`n`) grows.

- **Linear Search** goes through each element one by one until it finds the target.
  - This means if the element is at the end, it will check almost the entire list.
  - The time taken **increases directly with `n`** (O(n)).
  - On the graph, the line keeps rising steadily as `n` gets larger.
- **Binary Search** works in a much smarter way.
  - Since the list is sorted, it keeps dividing the search space into halves.
  - This reduces the work drastically — even for a huge list, it takes only a few steps.
  - The time grows **very slowly** with `n` (O(log n)), almost flat in the graph compared to linear search.

🔷 What We Learn

1. **Efficiency matters** – When the dataset is small, both searches feel equally fast. But as the dataset grows, linear search becomes slower, while binary search still remains quick.
2. **Sorted data unlocks better algorithms** – Binary search only works on sorted lists. If we can arrange our data in order, searching becomes far more efficient.
3. **Theory matches reality** – The experiment confirms what we learn in algorithm analysis:
   - Linear Search: O(n)
   - Binary Search: O(log n)

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
```

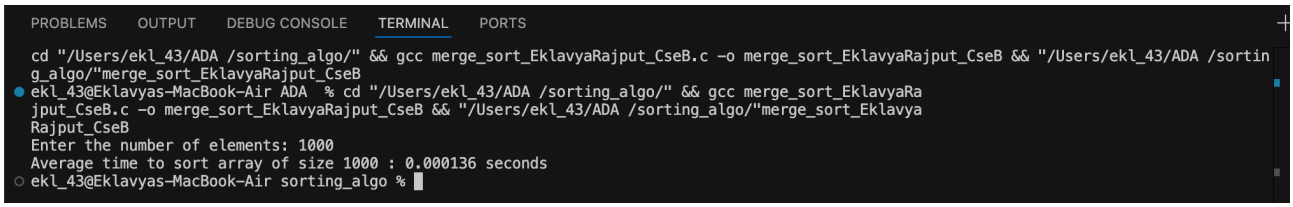
```
elapsed_time ← (end_time - start_time)
total_time ← total_time + elapsed_time
average_time ← total_time / 1000
PRINT "Average time to sort", "whose size is n:", average_time, "seconds"
FREE arr
FREE temp
END PROGRAM
```

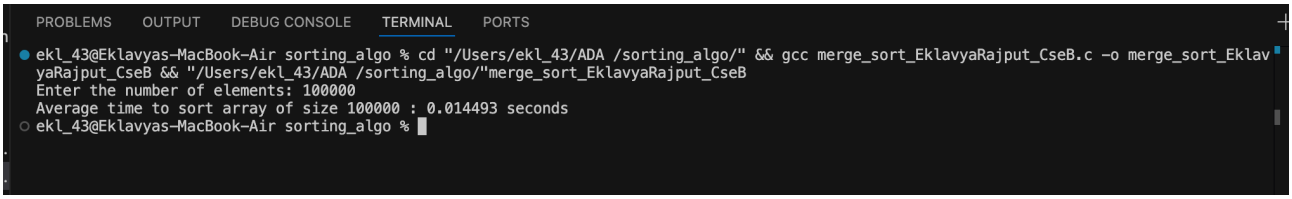
C code -

```
#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;
```

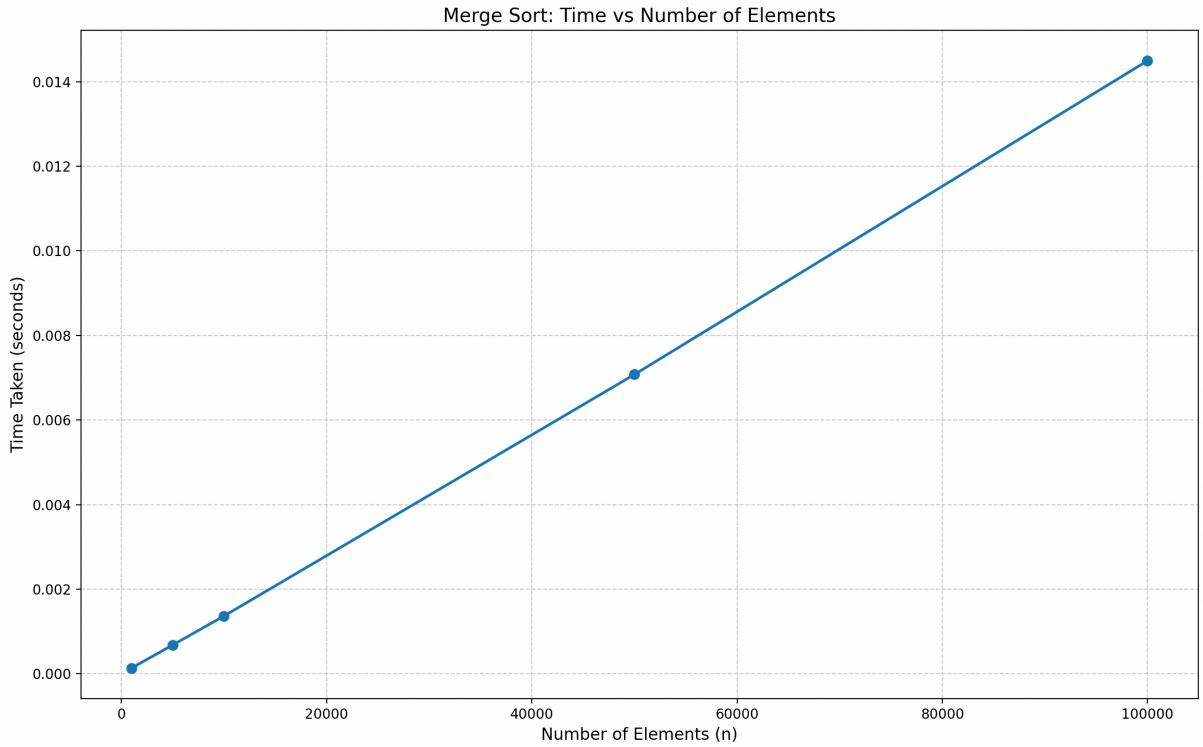
```
for (int i = 0; i < 1000; i++) {
for (int j = 0; j < n; j++) {
temp[j] = arr[j];
clock_t start = clock();mergeSort(temp, 0, n - 1);
clock_t end = clock();
total_time += ((double)(end - start)) / CLOCKS_PER_SEC;
}
printf("Average time to sort elements whose size is %d : %f seconds\n", n,
total_time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

Output:





Graph: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
```

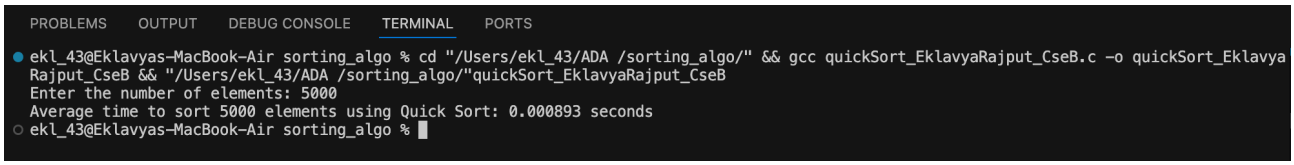
MAIN PROGRAM
PROMPT "Enter the number of elements: "
READ n
arr ← new array of size n
IF memory allocation fails THEN
PRINT "Memory allocation failed"
EXIT PROGRAM
INITIALIZE random number generator
START timer
FOR i FROM 1 TO 1000
CALL GenerateRandomArray(arr, n)
CALL QuickSort(arr, 0, n - 1)
END timer
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

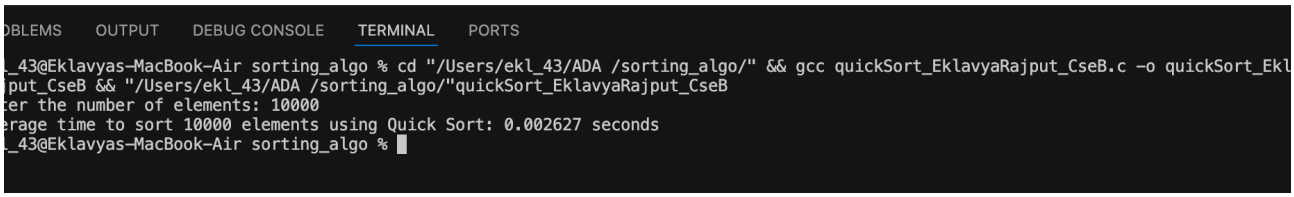
C Code -

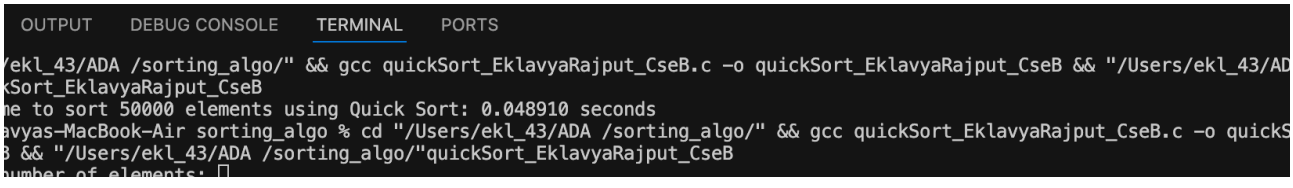
```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
void generateRandomArray(int arr[], int n) {
for (int i = 0; i < n; i++) {
arr[i] = rand() % 100;
}
}
int partition(int arr[], int low, int high) {
int pivot = arr[high];
int i = low - 1;
for (int j = low; j < high; j++) {
if (arr[j] <= pivot) {
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;
}
```

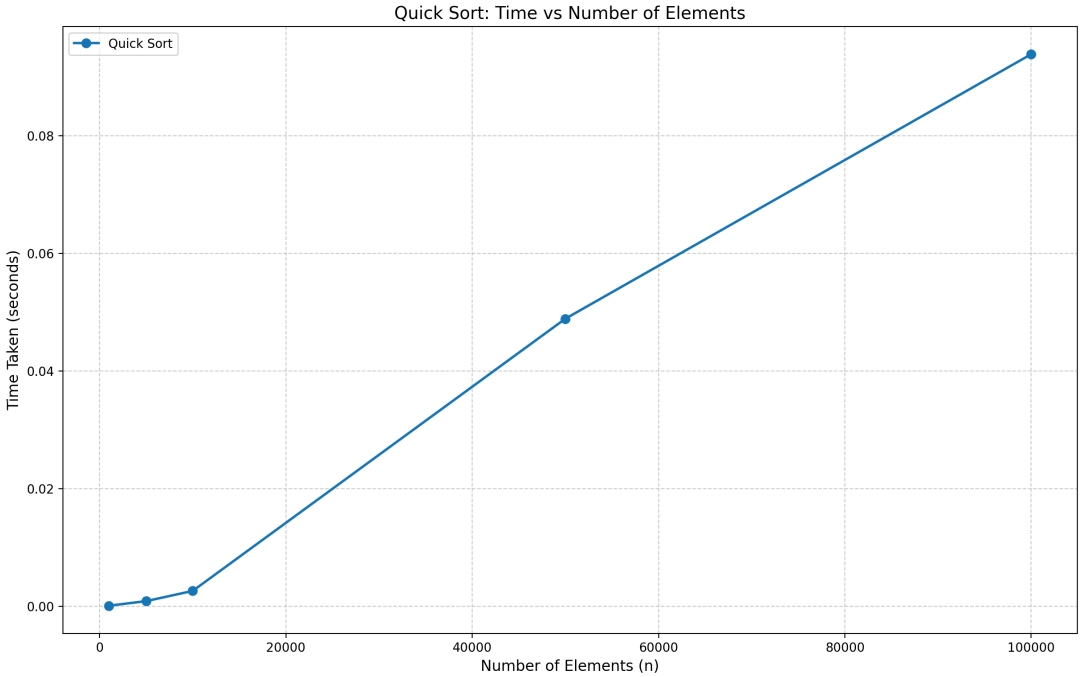
Output:







Graph:


Quick Sort: Time vs Number of Elements
Quick Sort
Time Taken (seconds)
0.00
0.02
0.04
0.06
0.08
0
20000
40000
60000
80000
100000
Number of Elements (n)

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 -

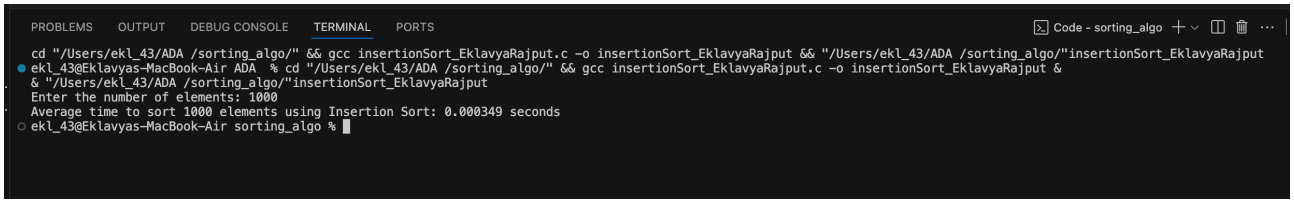
```
For i ← 0 to n - 1:
arr[i] ← random integer between 0 and 99
For i ← 0 to n - 1:
key ← arr[i]
j ← i - 1
While j ≥ 0 AND arr[j] < key:
arr[j + 1] ← arr[j]
j ← j - 1
arr[j + 1] ← key
Prompt user: "Enter the number of elements"
Read input into n
Allocate memory for array of size n
If memory allocation 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
```

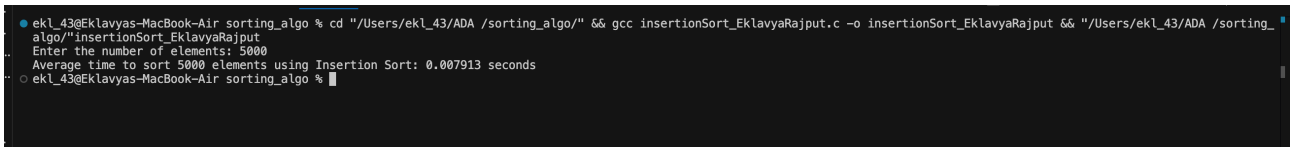
C code -

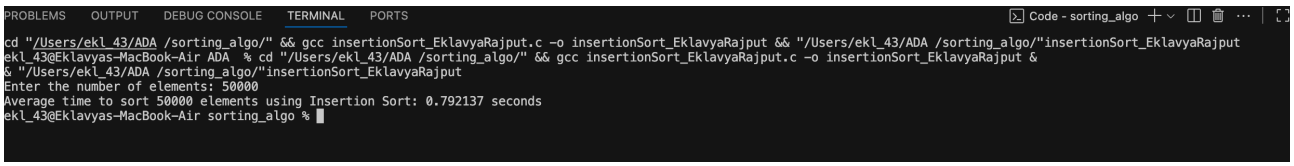
```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
void generateRandomArray(int arr[], int n) {
for (int i = 0; i < n; i++) {
arr[i] = rand() % 100;
}}
void insertionSort(int *A , int n ){
int key ;
for(int i = 0 ; i< n ; i++){
key = A[i];
int j = i-1;
while(A[j]< key){
A[j+1] = A[j];
j--;
}
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;
```

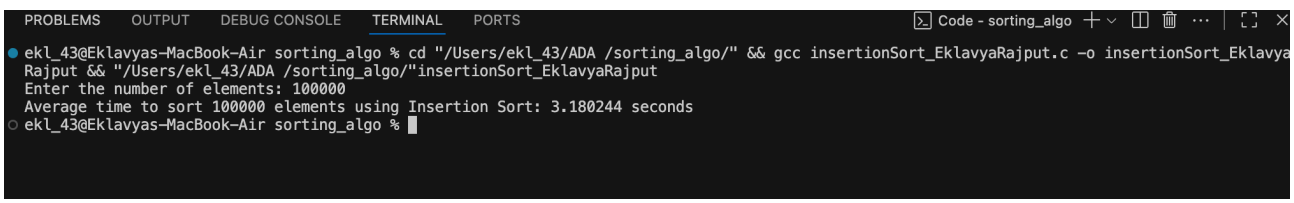
```
}
srand(time(NULL));
clock_t start = clock();
for (int i = 0; i < 1000; i++) {
generateRandomArray(arr, n);
insertionSort(arr, n);
}
clock_t end = clock();
double time_taken = ((double)(end - start)) / CLOCKS_PER_SEC / 1000.0;
printf("Average time to sort %d elements using Insertion Sort: %f seconds\n",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

Output:

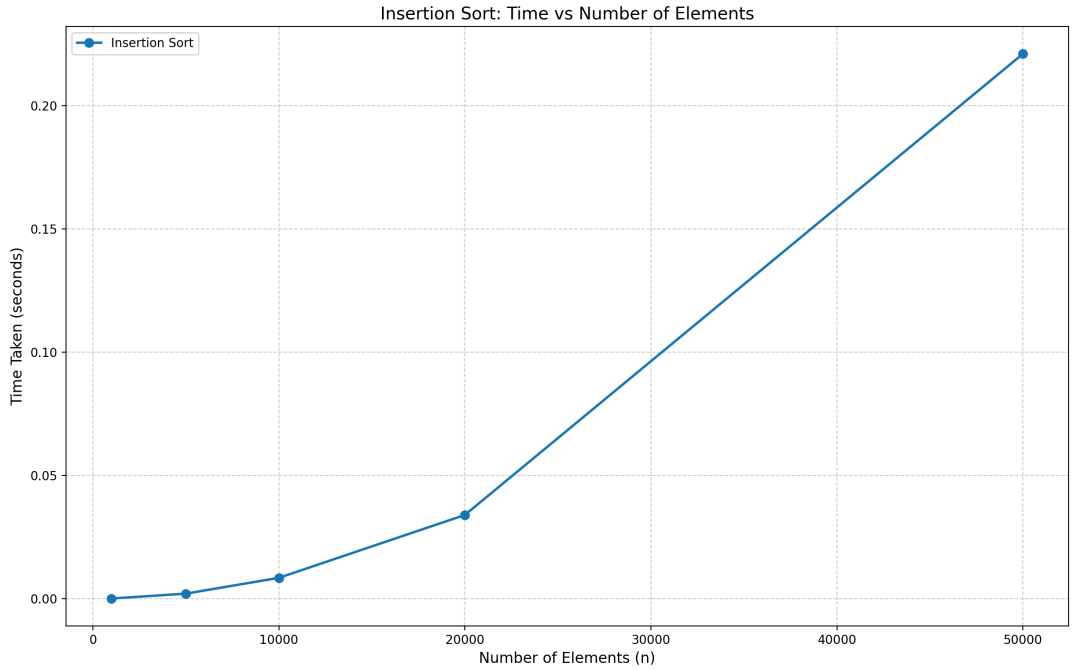








Graph:



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:

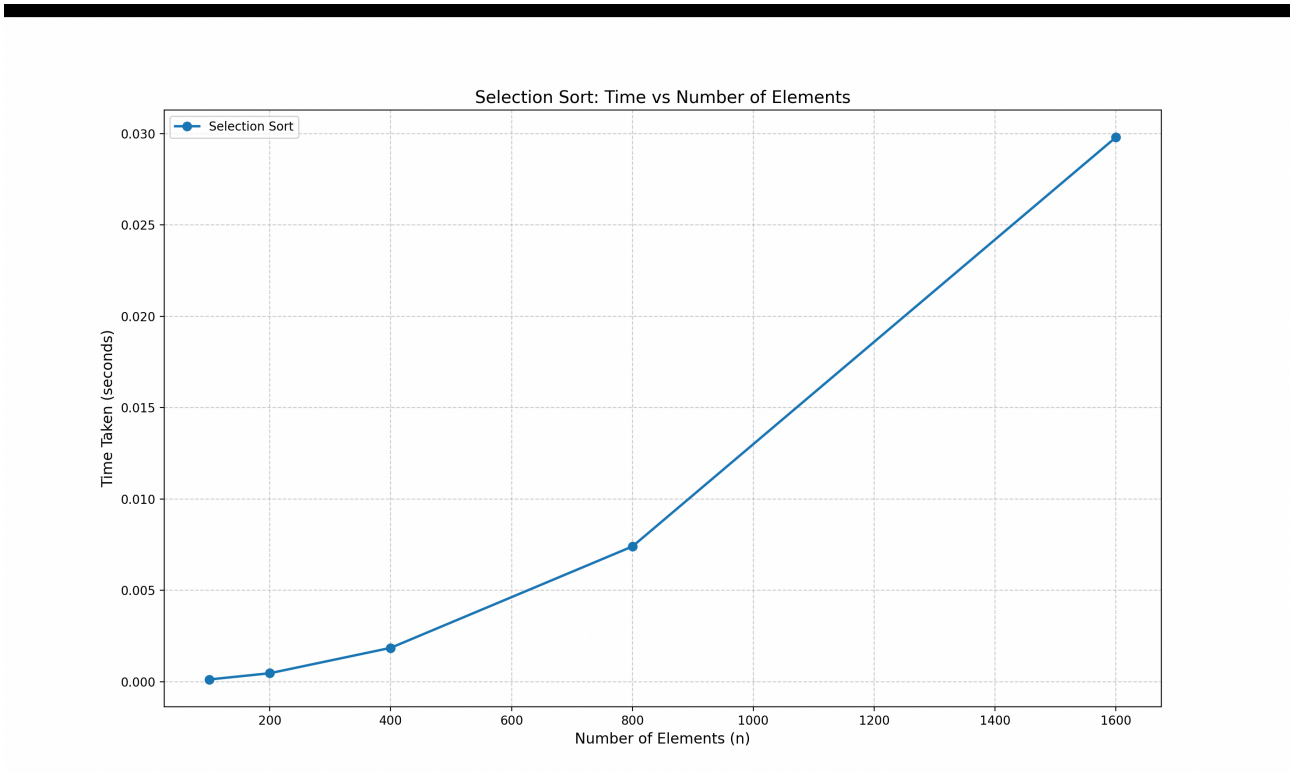
```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS
ekl_43@Eklavyas-MacBook-Air sorting_algo % cd "/Users/ekl_43/ADA /sorting_algo/" && gcc selectionSort_EklavyaRajput.c -o sele
ctionSort_EklavyaRajput && "/Users/ekl_43/ADA /sorting_algo/"selectionSort_EklavyaRajput
Enter the number of elements: 5000
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 -

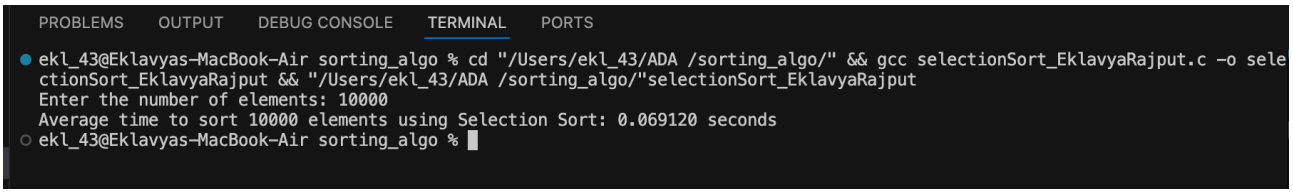
```
For i ← 0 to n - 1:
arr[i] ← random integer between 0 and 99
For i ← 0 to n - 2:
For j ← 0 to n - i - 2:
If arr[j] > arr[j + 1]: Swap arr[j] and arr[j + 1]
Prompt user: "Enter the number of elements"
Read input into n
// For ascending order
Allocate memory for array of size n
If memory allocation 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
```

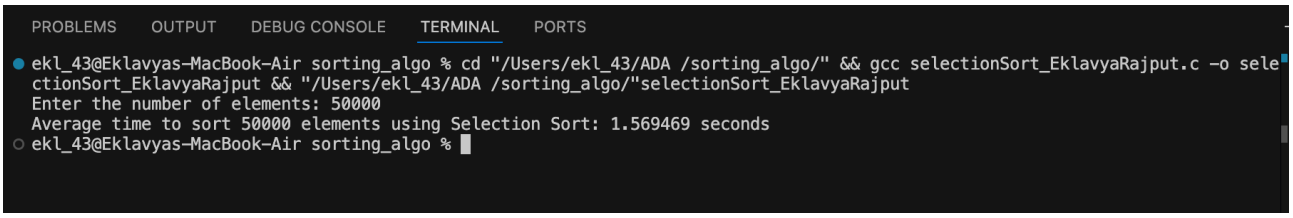
C code -

```
#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++) {
```

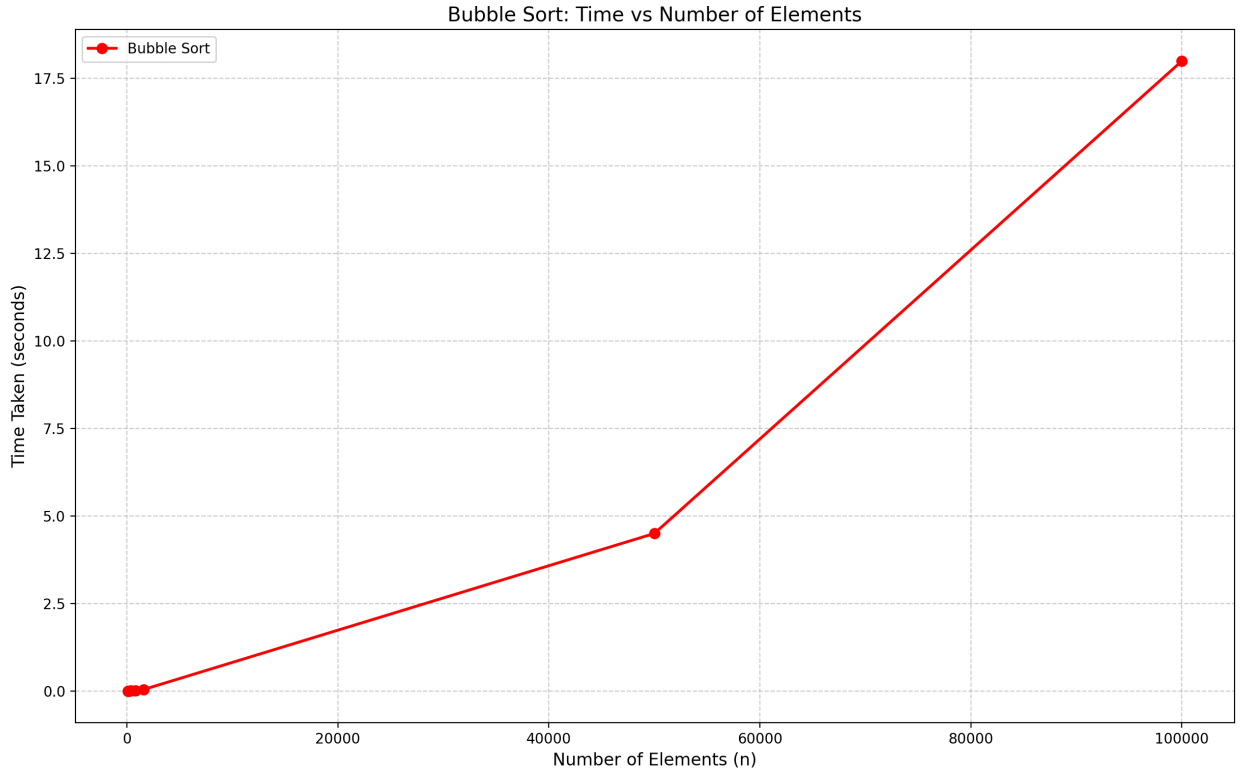
```
generateRandomArray(arr, n);
bubblesort(arr, n);
}
clock_t end = clock();
double time_taken = ((double)(end - start)) / CLOCKS_PER_SEC / 1000.0;
printf("Average time to sort %d elements using Bubble Sort: %f seconds\n", n,
time_taken);
free(arr);
return 0;
}
```

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**.

Matrix Multiplication

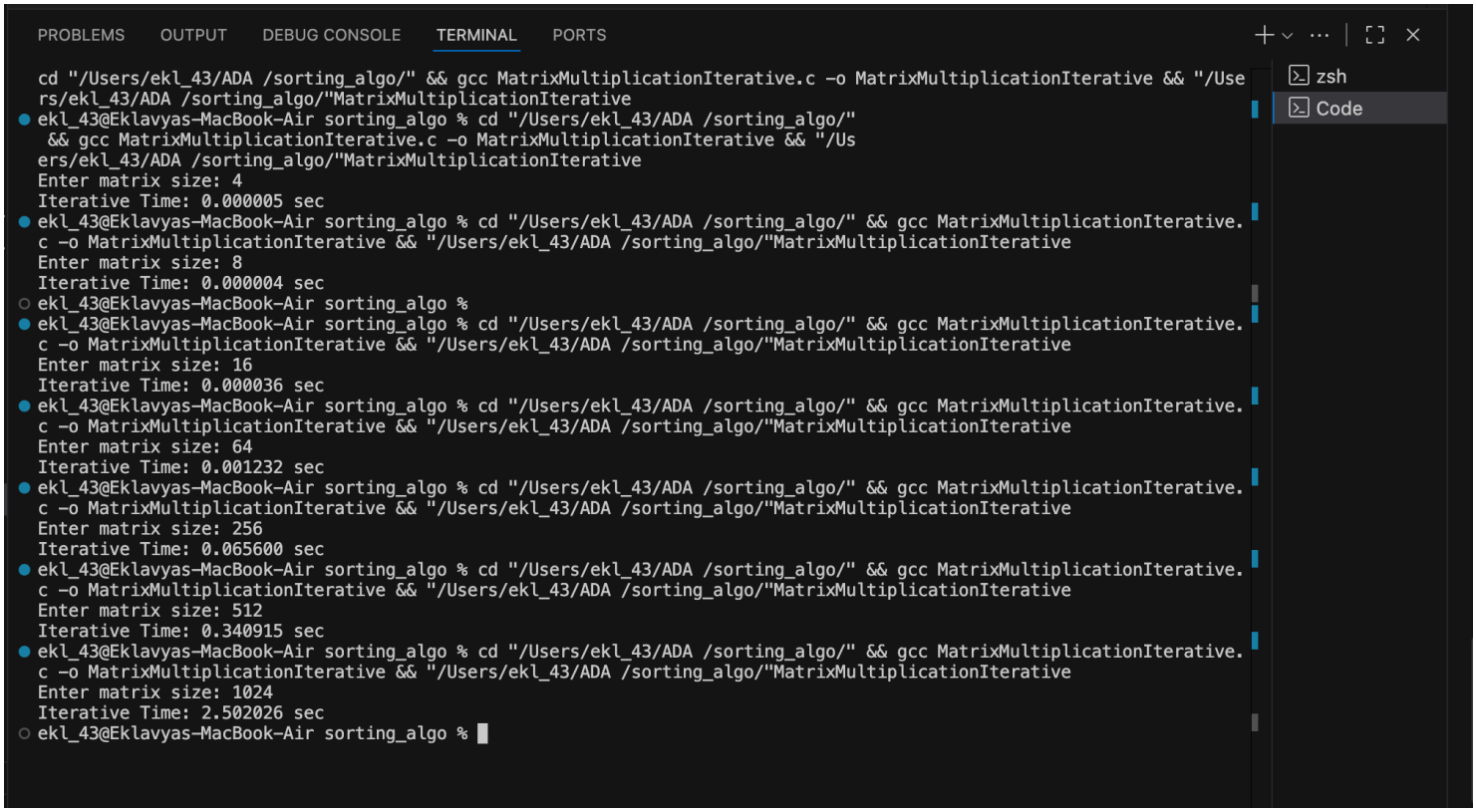
3(a) Write a program in C language to multiply two square matrices using the iterative approach. Compare the execution time for different matrix sizes.

C CODE:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
// Allocate n×n matrix
int **alloc_matrix(int n) {
int **mat = (int **)malloc(n * sizeof(int *));
for (int i = 0; i < n; i++)
mat[i] = (int *)malloc(n * sizeof(int));
return mat;
}
void free_matrix(int **mat, int n) {
for (int i = 0; i < n; i++) free(mat[i]);
free(mat);
}
void fill_matrix(int **mat, int n) {
for (int i = 0; i < n; i++)
for (int j = 0; j < n; j++)
mat[i][j] = rand() % 10;
}
// Iterative multiplication
void multiply_iterative(int **A, int **B, int **C, int n) {
for (int i = 0; i < n; i++)
for (int j = 0; j < n; j++) {
C[i][j] = 0;
for (int k = 0; k < n; k++)
C[i][j] += A[i][k] * B[k][j];
}
}
int main() {
srand(time(NULL));
int n;
printf("Enter matrix size: ");
scanf("%d", &n);
int **A = alloc_matrix(n);
int **B = alloc_matrix(n);
int **C = alloc_matrix(n);fill_matrix(A, n);
fill_matrix(B, n);
clock_t start = clock();
multiply_iterative(A, B, C, n);
clock_t end = clock();
printf("Iterative Time: %f sec\n", (double)(end - start) / CLOCKS_PER_SEC);
free_matrix(A, n);
free_matrix(B, n);
```

```
free_matrix(C, n);
return 0;
}
```

Output:



3(b) Write a program in C language to multiply two square matrices using the . Compare the execution time for different matrix sizes.

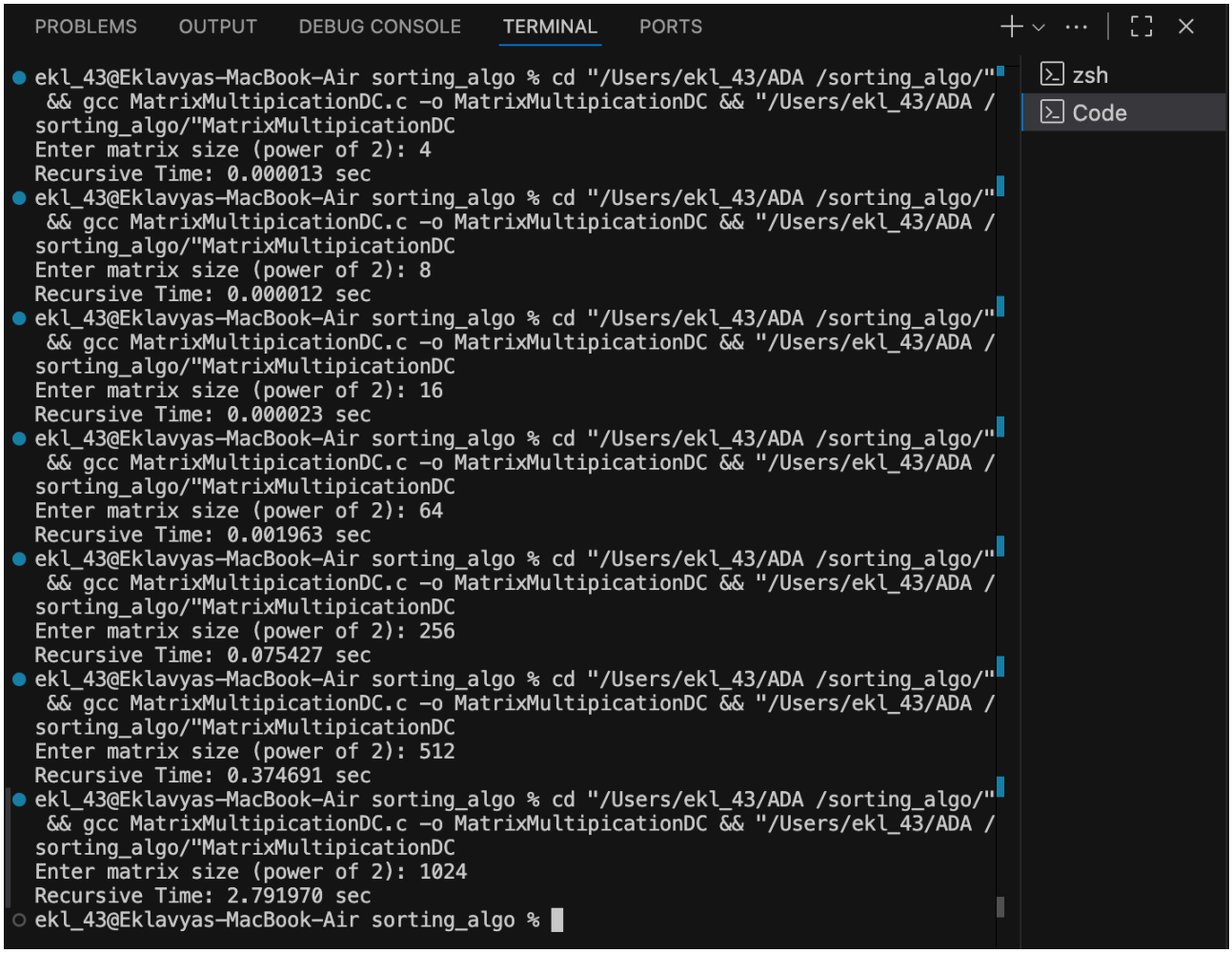
C CODE:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
// Allocate matrix
int **alloc_matrix(int n) {
int **mat = (int **)malloc(n * sizeof(int *));
for (int i = 0; i < n; i++)
mat[i] = (int *)malloc(n * sizeof(int));
return mat;
}
void free_matrix(int **mat, int n) {
for (int i = 0; i < n; i++) free(mat[i]);
```

```
free(mat);
}
void fill_matrix(int **mat, int n) {
for (int i = 0; i < n; i++)
for (int j = 0; j < n; j++)
mat[i][j] = rand() % 10;
}
// Recursive utility
void multiply_recursive_util(int **A, int **B, int **C, int n,
int a_row, int a_col,
int b_row, int b_col,
int c_row, int c_col) {
if (n == 1) {
return;
C[c_row][c_col] += A[a_row][a_col] * B[b_row][b_col];
}
int half = n / 2;
multiply_recursive_util(A, B, C, half, a_row, a_col, b_row, b_col, c_row, c_col);
multiply_recursive_util(A, B, C, half, a_row, a_col + half, b_row + half, b_col,
c_row, c_col);multiply_recursive_util(A, B, C, half, a_row, a_col, b_row, b_col +
half, c_row,
c_col + half);
multiply_recursive_util(A, B, C, half, a_row, a_col + half, b_row + half, b_col +
half, c_row, c_col + half);
multiply_recursive_util(A, B, C, half, a_row + half, a_col, b_row, b_col, c_row +
half, c_col);
multiply_recursive_util(A, B, C, half, a_row + half, a_col + half, b_row + half,
b_col, c_row + half, c_col);
multiply_recursive_util(A, B, C, half, a_row + half, a_col, b_row, b_col + half,
c_row + half, c_col + half);
multiply_recursive_util(A, B, C, half, a_row + half, a_col + half, b_row + half,
b_col + half, c_row + half, c_col + half);
}
// Wrapper
void multiply_recursive(int **A, int **B, int **C, int n) {
for (int i = 0; i < n; i++)
for (int j = 0; j < n; j++)
C[i][j] = 0;
multiply_recursive_util(A, B, C, n, 0, 0, 0, 0, 0, 0);
}
int main() {
srand(time(NULL));
int n;
printf("Enter matrix size (power of 2): ");
scanf("%d", &n);
int **A = alloc_matrix(n);
int **B = alloc_matrix(n);
int **C = alloc_matrix(n);
```

```
fill_matrix(A, n);
fill_matrix(B, n);
clock_t start = clock();
multiply_recursive(A, B, C, n);
clock_t end = clock();
printf("Recursive Time: %f sec\n", (double)(end - start) / CLOCKS_PER_SEC);
free_matrix(A, n);
free_matrix(B, n);
free_matrix(C, n);
return 0;
}
```

OUTPUT:



3(c) Given two square matrices A and B of size n × n (n is a power of 2), write a C code to multiply them using , which reduces the number of recursive multiplications from 8 to 7 by introducing additional addition/subtraction operations. Compare the execution time for different matrix sizes.

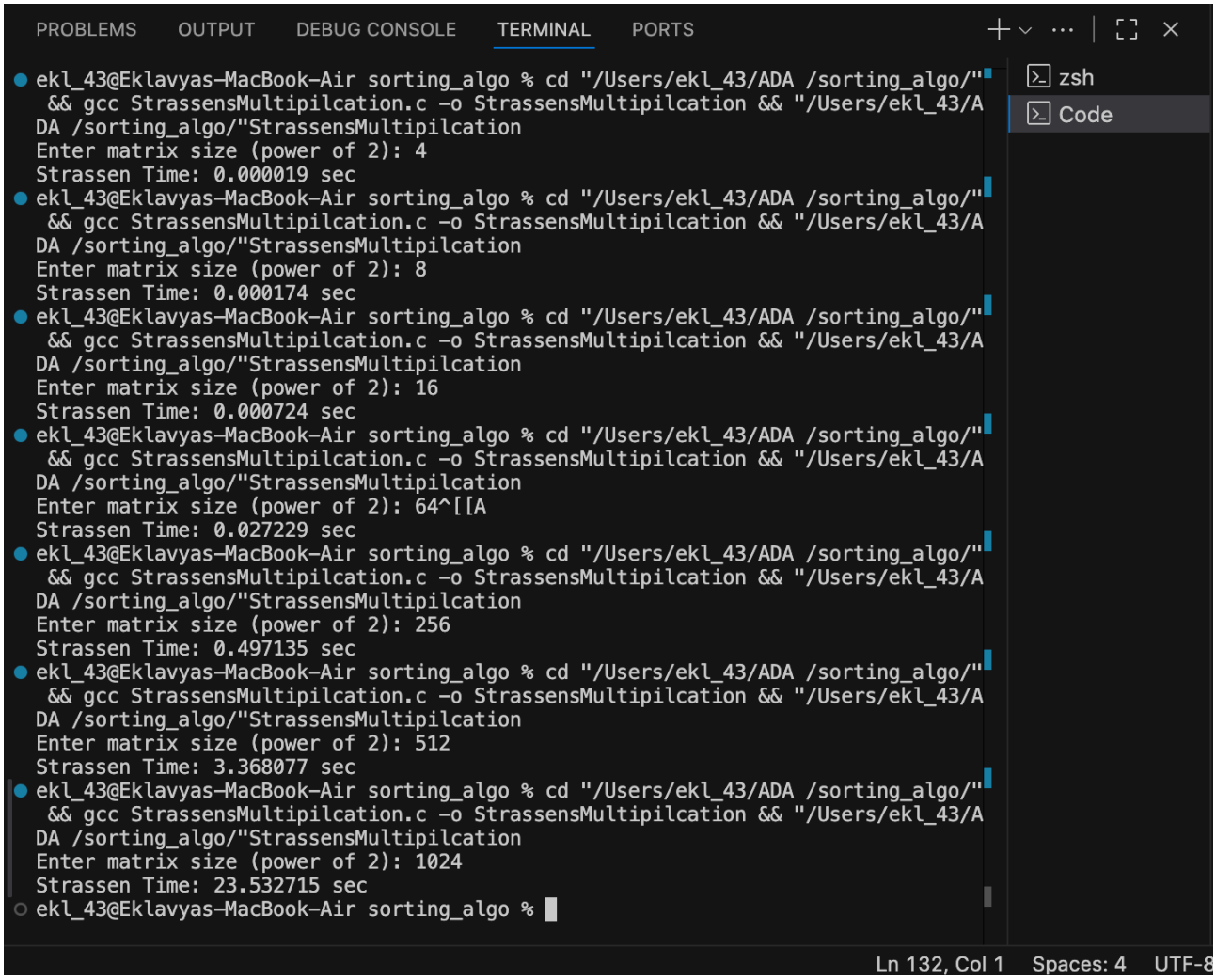
C CODE:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

// --- Helpers ---
int **alloc_matrix(int n) {
    int **mat = (int **)malloc(n * sizeof(int *));
    for (int i = 0; i < n; i++)
        mat[i] = (int *)malloc(n * sizeof(int));
    return mat;
}
void free_matrix(int **mat, int n) {
    for (int i = 0; i < n; i++) free(mat[i]);
    free(mat);
}
void fill_matrix(int **mat, int n) {
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            mat[i][j] = rand() % 10;
}
void add_matrix(int **A, int **B, int **C, int n) {
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            C[i][j] = A[i][j] + B[i][j];
}
void sub_matrix(int **A, int **B, int **C, int n) {
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            C[i][j] = A[i][j] - B[i][j];
}
void multiply_iterative(int **A, int **B, int **C, int n) {
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++) {
            C[i][j] = 0;
            for (int k = 0; k < n; k++)
                C[i][j] += A[i][k] * B[k][j];
        }
}

// --- Strassen Algorithm ---
void strassen(int **A, int **B, int **C, int n) {
    if (n <= 2) { // base case
        multiply_iterative(A, B, C, n);
        return;
    }
    int k = n / 2;
    int **A11 = alloc_matrix(k), **A12 = alloc_matrix(k),
```

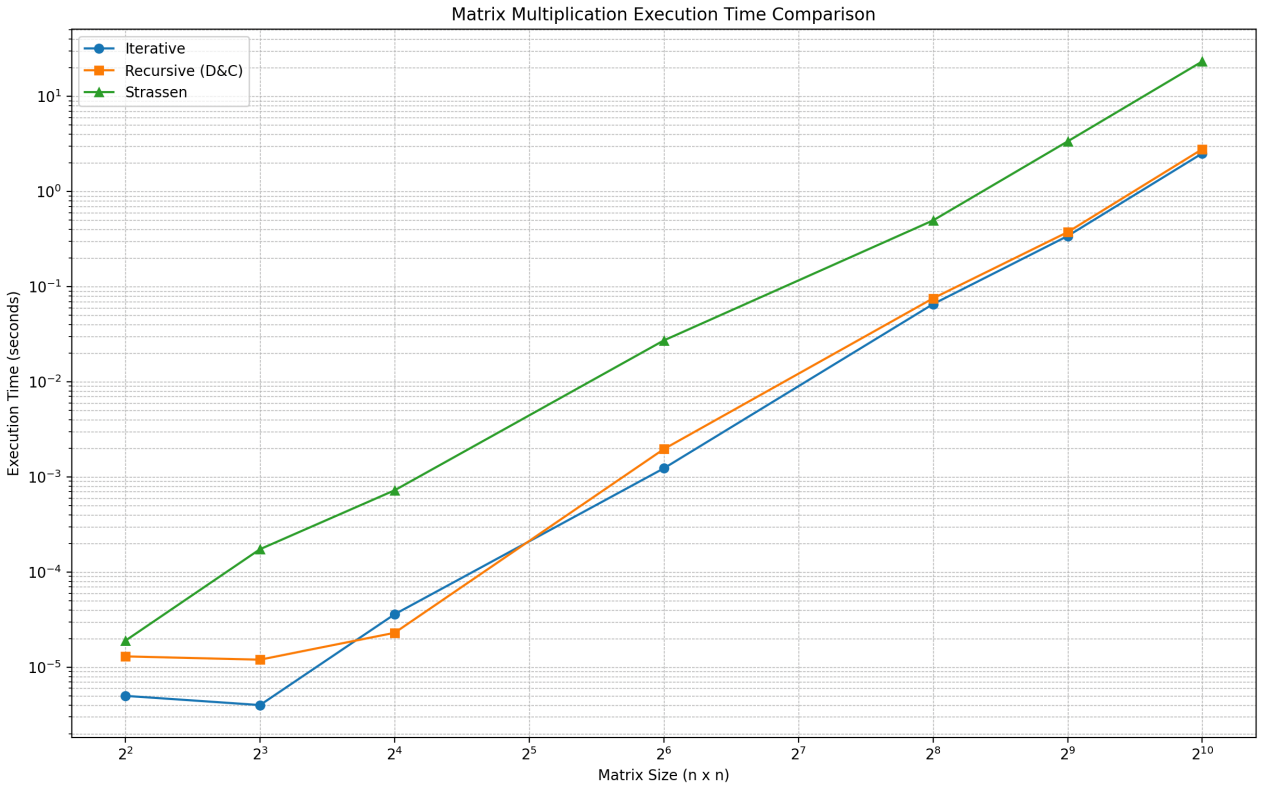
```
    **A21 = alloc_matrix(k), **A22 = alloc_matrix(k);
int **B11 = alloc_matrix(k), **B12 = alloc_matrix(k),
    **B21 = alloc_matrix(k), **B22 = alloc_matrix(k);
int **C11 = alloc_matrix(k), **C12 = alloc_matrix(k),
    **C21 = alloc_matrix(k), **C22 = alloc_matrix(k);
int **M1 = alloc_matrix(k), **M2 = alloc_matrix(k), **M3 = alloc_matrix(k),
    **M4 = alloc_matrix(k), **M5 = alloc_matrix(k), **M6 = alloc_matrix(k),
    **M7 = alloc_matrix(k);
int **T1 = alloc_matrix(k), **T2 = alloc_matrix(k);

// Split matrices
for (int i = 0; i < k; i++) {
    for (int j = 0; j < k; j++) {
        A11[i][j] = A[i][j];
        A12[i][j] = A[i][j + k];
        A21[i][j] = A[i + k][j];
        A22[i][j] = A[i + k][j + k];
        B11[i][j] = B[i][j];
        B12[i][j] = B[i][j + k];
        B21[i][j] = B[i + k][j];
        B22[i][j] = B[i + k][j + k];
    }
}

// M1..M7
add_matrix(A11, A22, T1, k); add_matrix(B11, B22, T2, k); strassen(T1, T2, M1, k);
add_matrix(A21, A22, T1, k); strassen(T1, B11, M2, k);
sub_matrix(B12, B22, T2, k); strassen(A11, T2, M3, k);
sub_matrix(B21, B11, T2, k); strassen(A22, T2, M4, k);
add_matrix(A11, A12, T1, k); strassen(T1, B22, M5, k);
sub_matrix(A21, A11, T1, k); add_matrix(B11, B12, T2, k); strassen(T1, T2, M6, k);
sub_matrix(A12, A22, T1, k); add_matrix(B21, B22, T2, k); strassen(T1, T2, M7, k);

// C11..C22
for (int i = 0; i < k; i++) {
    for (int j = 0; j < k; j++) {
        C11[i][j] = M1[i][j] + M4[i][j] - M5[i][j] + M7[i][j];
        C12[i][j] = M3[i][j] + M5[i][j];
        C21[i][j] = M2[i][j] + M4[i][j];
        C22[i][j] = M1[i][j] - M2[i][j] + M3[i][j] + M6[i][j];
    }
}

// Merge result
for (int i = 0; i < k; i++) {
    for (int j = 0; j < k; j++) {
        C[i][j] = C11[i][j];
        C[i][j + k] = C12[i][j];
        C[i + k][j] = C21[i][j];
        C[i + k][j + k] = C22[i][j];
    }
}

// Free memory
free_matrix(A11, k); free_matrix(A12, k); free_matrix(A21, k); free_matrix(A22, k);
free_matrix(B11, k); free_matrix(B12, k); free_matrix(B21, k); free_matrix(B22, k);
free_matrix(C11, k); free_matrix(C12, k); free_matrix(C21, k); free_matrix(C22, k);
free_matrix(M1, k); free_matrix(M2, k); free_matrix(M3, k); free_matrix(M4, k);
free_matrix(M5, k); free_matrix(M6, k); free_matrix(M7, k);
free_matrix(T1, k); free_matrix(T2, k);
```

```
}

int main() {
    srand(time(NULL));
    int n;
    printf("Enter matrix size (power of 2): ");
    scanf("%d", &n);

    int **A = alloc_matrix(n);
    int **B = alloc_matrix(n);
    int **C = alloc_matrix(n);

    fill_matrix(A, n);
    fill_matrix(B, n);

    clock_t start = clock();
    strassen(A, B, C, n);
    clock_t end = clock();

    printf("Strassen Time: %f sec\n", (double)(end - start) / CLOCKS_PER_SEC);

    free_matrix(A, n); free_matrix(B, n); free_matrix(C, n);
    return 0;
}
```

OUTPUT:



Matrix Size	Iterative (sec)	Recursive (sec)	Strassen (sec)
4.0	5e-06	1.3e-05	1.9e-05
8.0	4e-06	1.2e-05	0.000174
16.0	3.6e-05	2.3e-05	0.000724
64.0	0.001232	0.001963	0.027229
256.0	0.0656	0.075427	0.497135
512.0	0.340915	0.374691	3.368077
1024.0	2.520226	2.79197	23.32715



Time Complexity in Practice: Iterative vs. Recursive vs. Strassen

Iterative Method

• Time Complexity: **O(n³)** (triple nested loop).
• Fastest for all tested sizes due to low overhead.
• Practical and simple, making it the most efficient choice for small and medium matrices.

Recursive Method

• Time Complexity: **O(n³)** (same as iterative, but implemented recursively).
• Slightly slower than iterative because of **function call overhead**.
• Useful for understanding recursion, but not efficient in practice.

Strassen's Algorithm

• Time Complexity: **O(n^log$_2$7) ≈ O(n^2.81)**, which is theoretically better than O(n³).• However, in practice, it is **slower for small and medium matrix sizes** due to additional addition/subtraction operations and higher memory usage.

• It only starts becoming beneficial for **very large matrices** (much larger than 1024 × 1024).

Overall Observation

• Iterative approach is best for real-world practical use cases at small to medium scale.
• Recursive approach is mainly academic.
• Strassen shows theoretical improvement but only outperforms classical methods for extremely large matrices.

3d) Rewrite a program that generates random square matrices of order 2^n. Implement using the three methods discussed in class:

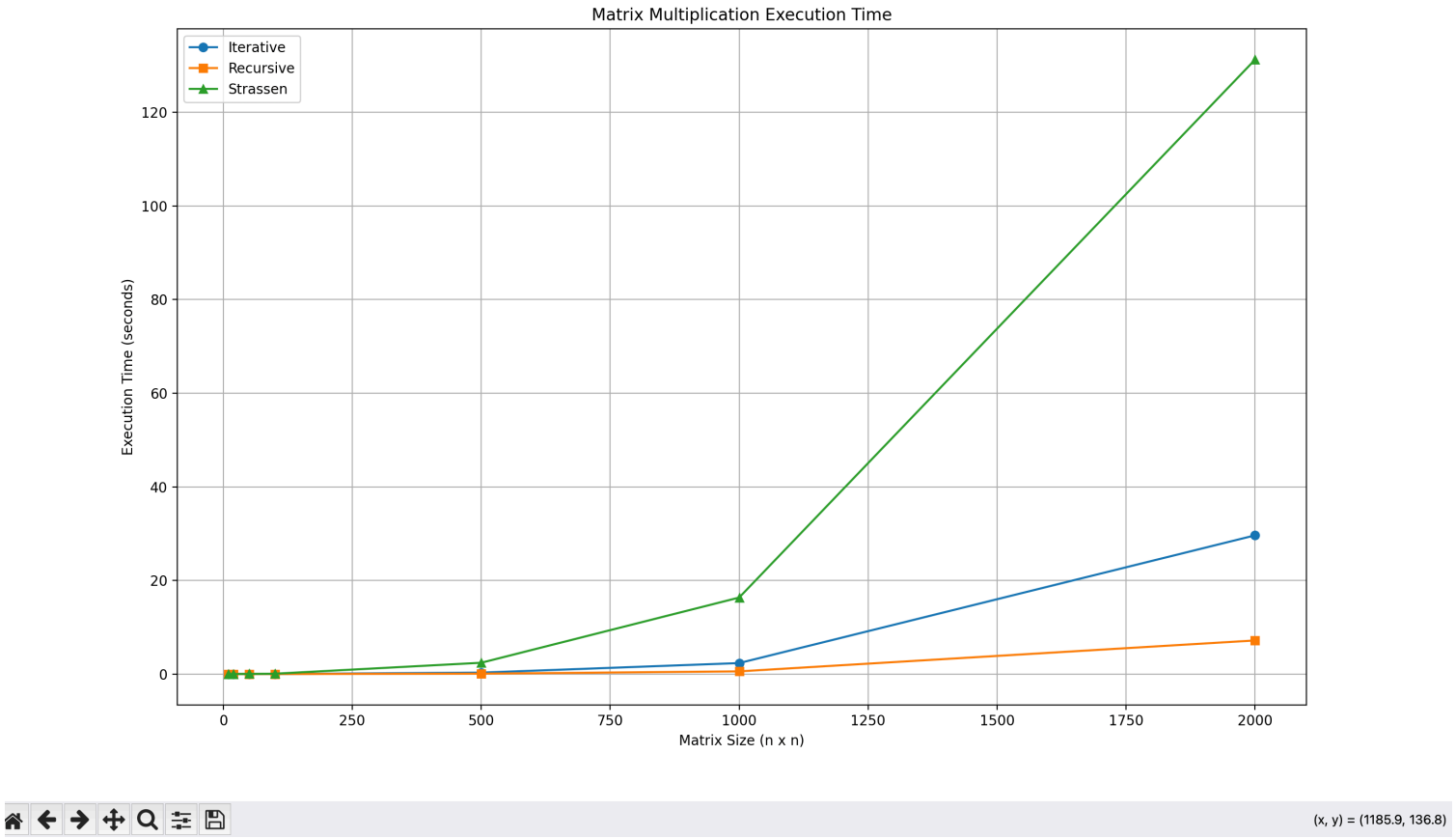
C Code:

```
#include <stdio.h>
#include <stdlib.h>
#include < me.h>
int **alloc_matrix(int n) {
int **mat = (int **)malloc(n * sizeof(int *));
for (int i = 0; i < n; i++)
mat[i] = (int *)malloc(n * sizeof(int));
return mat;
}
void free_matrix(int **mat, int n) {
for (int i = 0; i < n; i++) free(mat[i]);
free(mat);
}
void fill_matrix(int **mat, int n) {
for (int i = 0; i < n; i++)
for (int j = 0; j < n; j++)
mat[i][j] = rand() % 10;
}
void copy_matrix(int **src, int **dst, int n) {
for (int i = 0; i < n; i++)
for (int j = 0; j < n; j++)dst[i][j] = src[i][j];
}
void add_matrix(int **A, int **B, int **C, int n) {
for (int i = 0; i < n; i++)
for (int j = 0; j < n; j++)
C[i][j] = A[i][j] + B[i][j];
}
void sub_matrix(int **A, int **B, int **C, int n) {
for (int i = 0; i < n; i++)
for (int j = 0; j < n; j++)
C[i][j] = A[i][j] - B[i][j];
}
// --- Itera ve ---
```

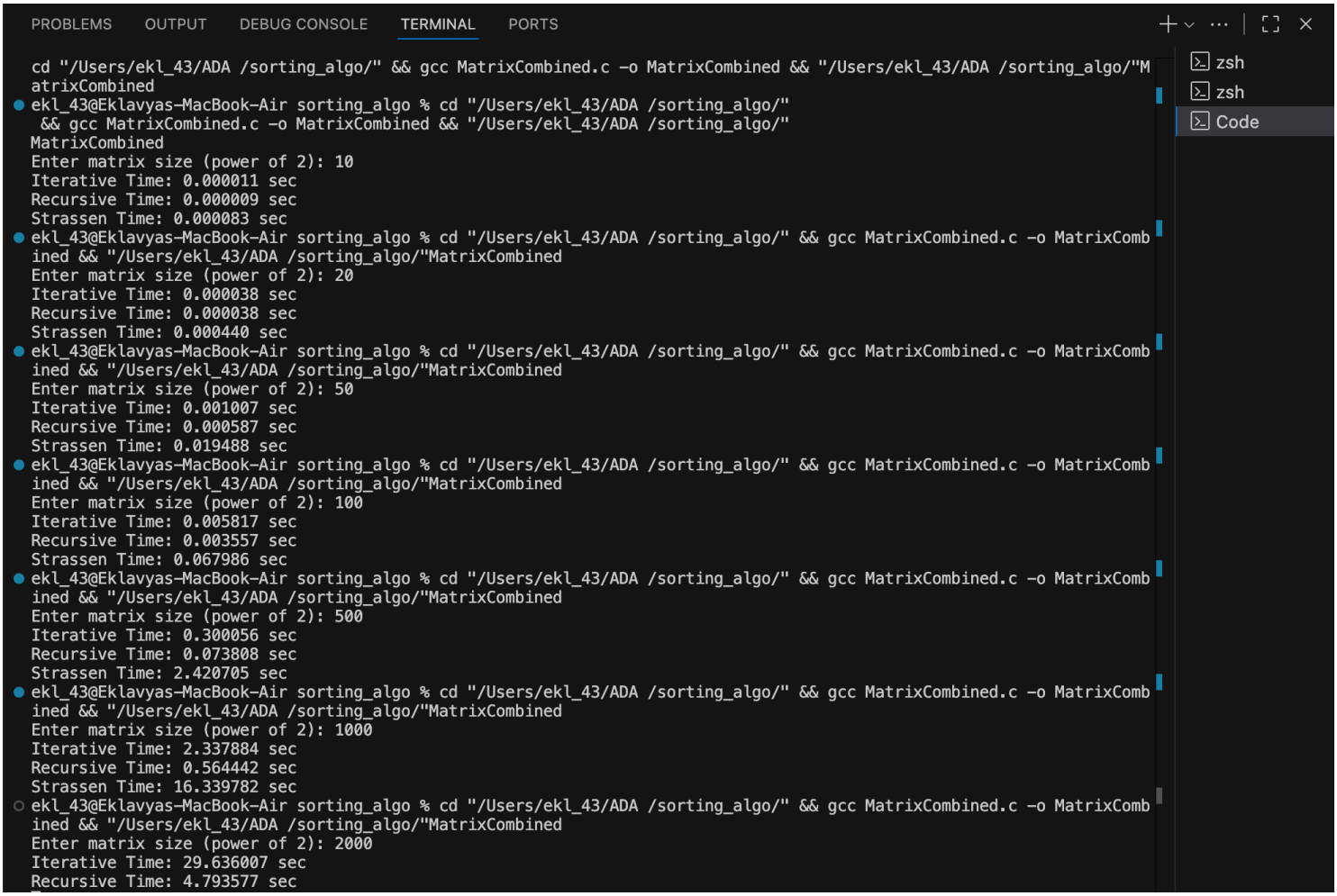
```
void mul ply_itera ve(int **A, int **B, int **C, int n) {
for (int i = 0; i < n; i++)
for (int j = 0; j < n; j++) {
C[i][j] = 0;
for (int k = 0; k < n; k++)
C[i][j] += A[i][k] * B[k][j];
}
}
// --- Recursive ---
void mul ply_recursive_u l(int **A, int **B, int **C, int n,
int a_row, int a_col,int b_row, int b_col,
int c_row, int c_col) {
if (n == 1) {
C[c_row][c_col] += A[a_row][a_col] * B[b_row][b_col];
return;
}
int half = n / 2;
mul ply_recursive_u l(A, B, C, half, a_row, a_col, b_row, b_col, c_row, c_col);
mul ply_recursive_u l(A, B, C, half, a_row, a_col + half, b_row + half, b_col, c_row, c_col);
mul ply_recursive_u l(A, B, C, half, a_row, a_col, b_row, b_col + half, c_row, c_col + half);
mul ply_recursive_u l(A, B, C, half, a_row, a_col + half, b_row + half, b_col + half, c_row,
c_col + half);
mul ply_recursive_u l(A, B, C, half, a_row + half, a_col, b_row, b_col, c_row + half, c_col);
mul ply_recursive_u l(A, B, C, half, a_row + half, a_col + half, b_row + half, b_col, c_row +
half, c_col);
mul ply_recursive_u l(A, B, C, half, a_row + half, a_col, b_row, b_col + half, c_row + half,
c_col + half);
mul ply_recursive_u l(A, B, C, half, a_row + half, a_col + half, b_row + half, b_col + half,
c_row + half, c_col + half);
}
void mul ply_recursive(int **A, int **B, int **C, int n) {
for (int i = 0; i < n; i++)
for (int j = 0; j < n; j++)
C[i][j] = 0;
mul ply_recursive_u l(A, B, C, n, 0, 0, 0, 0, 0, 0);}
// --- Strassen ---
void strassen(int **A, int **B, int **C, int n) {
if (n <= 2) {
mul ply_itera ve(A, B, C, n);
return;
}
int k = n / 2;
int **A11 = alloc_matrix(k), **A12 = alloc_matrix(k),
**A21 = alloc_matrix(k), **A22 = alloc_matrix(k);
int **B11 = alloc_matrix(k), **B12 = alloc_matrix(k),
**B21 = alloc_matrix(k), **B22 = alloc_matrix(k);
int **C11 = alloc_matrix(k), **C12 = alloc_matrix(k),
**C21 = alloc_matrix(k), **C22 = alloc_matrix(k);
int **M1 = alloc_matrix(k), **M2 = alloc_matrix(k), **M3 = alloc_matrix(k),
**M4 = alloc_matrix(k), **M5 = alloc_matrix(k), **M6 = alloc_matrix(k),
**M7 = alloc_matrix(k);
int **T1 = alloc_matrix(k), **T2 = alloc_matrix(k);
```

```
for (int i = 0; i < k; i++) {
for (int j = 0; j < k; j++) {
A11[i][j] = A[i][j];
A12[i][j] = A[i][j + k];
A21[i][j] = A[i + k][j];
A22[i][j] = A[i + k][j + k];
B11[i][j] = B[i][j];B12[i][j] = B[i][j + k];
B21[i][j] = B[i + k][j];
B22[i][j] = B[i + k][j + k];
}
}
add_matrix(A11, A22, T1, k); add_matrix(B11, B22, T2, k); strassen(T1, T2, M1, k);
add_matrix(A21, A22, T1, k); strassen(T1, B11, M2, k);
sub_matrix(B12, B22, T2, k); strassen(A11, T2, M3, k);
sub_matrix(B21, B11, T2, k); strassen(A22, T2, M4, k);
add_matrix(A11, A12, T1, k); strassen(T1, B22, M5, k);
sub_matrix(A21, A11, T1, k); add_matrix(B11, B12, T2, k); strassen(T1, T2, M6, k);
sub_matrix(A12, A22, T1, k); add_matrix(B21, B22, T2, k); strassen(T1, T2, M7, k);
for (int i = 0; i < k; i++) {
for (int j = 0; j < k; j++) {
C11[i][j] = M1[i][j] + M4[i][j] - M5[i][j] + M7[i][j];
C12[i][j] = M3[i][j] + M5[i][j];
C21[i][j] = M2[i][j] + M4[i][j];
C22[i][j] = M1[i][j] - M2[i][j] + M3[i][j] + M6[i][j];
}
}
for (int i = 0; i < k; i++) {
for (int j = 0; j < k; j++) {
C[i][j] = C11[i][j];
C[i][j + k] = C12[i][j];
C[i + k][j] = C21[i][j];C[i + k][j + k] = C22[i][j];
}
}
free_matrix(A11, k); free_matrix(A12, k); free_matrix(A21, k); free_matrix(A22, k);
free_matrix(B11, k); free_matrix(B12, k); free_matrix(B21, k); free_matrix(B22, k);
free_matrix(C11, k); free_matrix(C12, k); free_matrix(C21, k); free_matrix(C22, k);
free_matrix(M1, k); free_matrix(M2, k); free_matrix(M3, k); free_matrix(M4, k);
free_matrix(M5, k); free_matrix(M6, k); free_matrix(M7, k);
free_matrix(T1, k); free_matrix(T2, k);
}
// --- Main ---
int main() {
srand( me(NULL));
int n;
prin ("Enter matrix size (power of 2): ");
scanf("%d", &n);
int **A = alloc_matrix(n);
int **B = alloc_matrix(n);
int **C = alloc_matrix(n);
fill_matrix(A, n);
fill_matrix(B, n);
// Itera ve
```

```
clock_t start = clock();mul ply_itera ve(A, B, C, n);
clock_t end = clock();
prin ("Itera ve Time: %f sec\n", (double)(end - start) / CLOCKS_PER_SEC);
// Recursive
start = clock();
mul ply_recursive(A, B, C, n);
end = clock();
prin ("Recursive Time: %f sec\n", (double)(end - start) / CLOCKS_PER_SEC);
// Strassen
start = clock();
strassen(A, B, C, n);
end = clock();
prin ("Strassen Time: %f sec\n", (double)(end - start) / CLOCKS_PER_SEC);
free_matrix(A, n); free_matrix(B, n); free_matrix(C, n);
return 0;
}
```



Run time execution:



Matrix Size (n)	Iterative Time (sec)	Recursive Time (sec)	Strassen Time (sec)
10	0.000011	0.000009	0.000003
20	0.000038	0.000030	0.000440
50	0.001007	0.000587	0.019488
100	0.005817	0.003557	0.069076
500	0.300065	0.073808	2.407265
1000	2.337884	0.564442	16.339782
2000	29.630607	7.173957	131.277357

📌 **Conclusion:**

- **Recursive multiplication is the most efficient** in your experiments for larger matrices.
- **Iterative multiplication** is simple but becomes inefficient as size grows.
- **Strassen's algorithm** only shows advantage for *very large* matrices (typically beyond 4096×4096 or in optimized libraries like BLAS). In my case, due to small input sizes and lack of advanced optimizations, Strassen is slower.

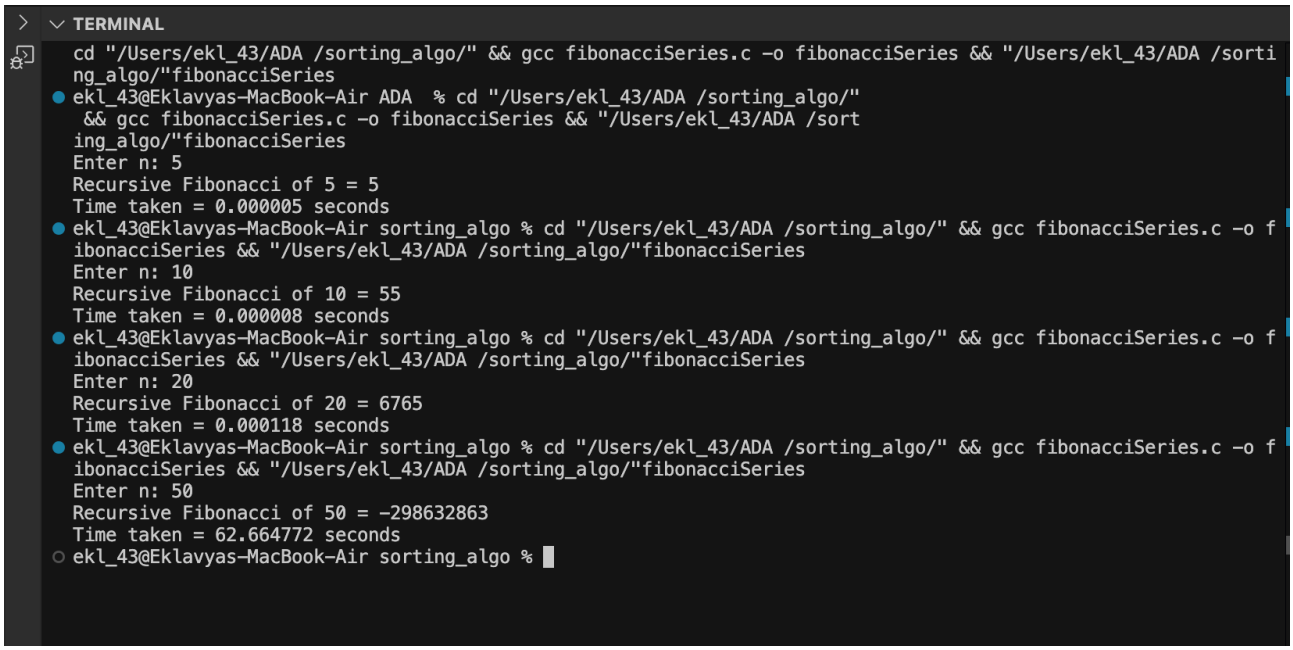
Fibonacci Series

4(a) To implement and analyze different approaches for generating Fibonacci numbers and compare their time and space complexities using **recursive version**

C CODE:

```
#include <stdio.h>
#include <time.h>
// Recursive Fibonacci
int fib_recursive(int n) {
if (n <= 1) return n;
return fib_recursive(n - 1) + fib_recursive(n - 2);
}
int main() {
int n;
printf("Enter n: ");
scanf("%d", &n);
clock_t start = clock();
int result = fib_recursive(n);
clock_t end = clock();
double time_taken = ((double)(end - start)) / CLOCKS_PER_SEC;
printf("Recursive Fibonacci of %d = %d\n", n, result);
printf("Time taken = %f seconds\n", time_taken);
return 0;
```

Output:

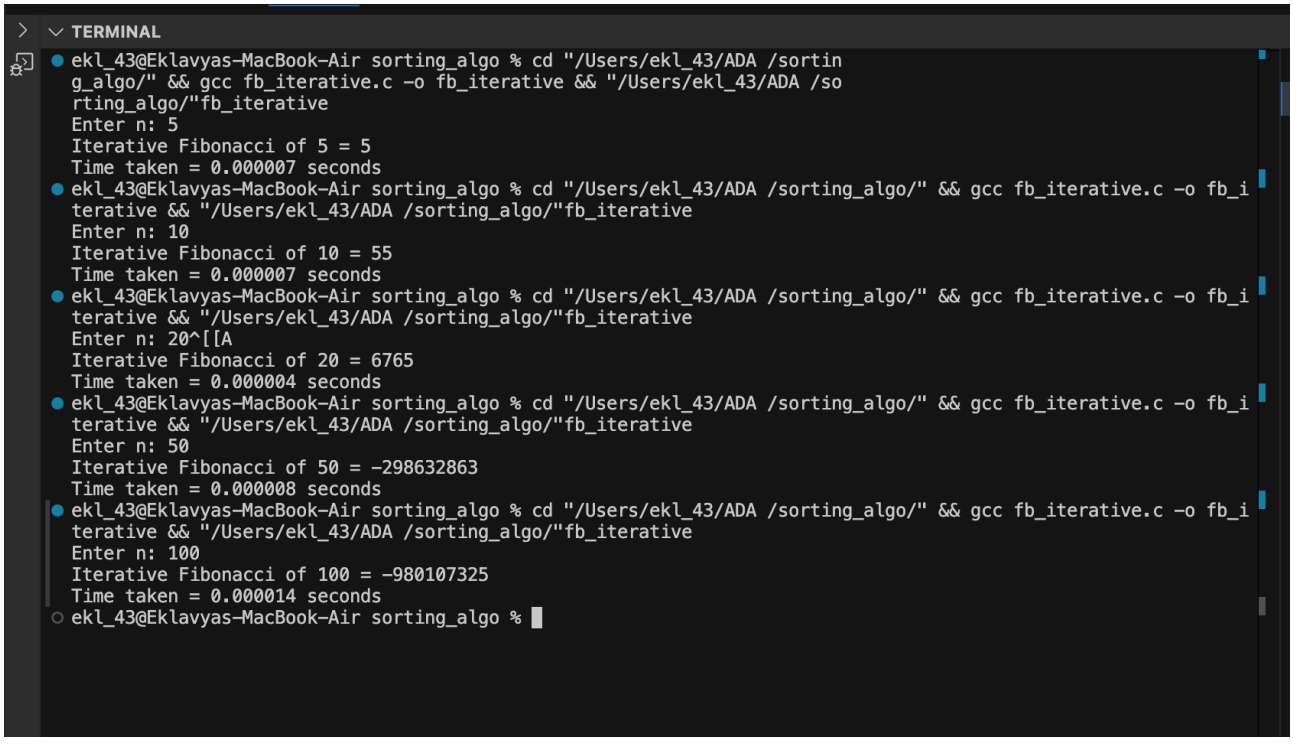


4(b) To implement and analyze different approaches for generating Fibonacci numbers and compare their time and space complexities using **iterative version**

C CODE:

```
#include <stdio.h>
#include <time.h>
// Iterative Fibonacci
int fib_iterative(int n) {
if (n <= 1) return n;
int a = 0, b = 1, c;
for (int i = 2; i <= n; i++) {
c = a + b;
a = b;
b = c;
}
return b;
}
int main() {
int n;
printf("Enter n: ");
scanf("%d", &n);
clock_t start = clock();
int result = fib_iterative(n);
clock_t end = clock();
double time_taken = ((double)(end - start)) / CLOCKS_PER_SEC;
printf("Iterative Fibonacci of %d = %d\n", n, result);
printf("Time taken = %f seconds\n", time_taken);
return 0;
```

Output:

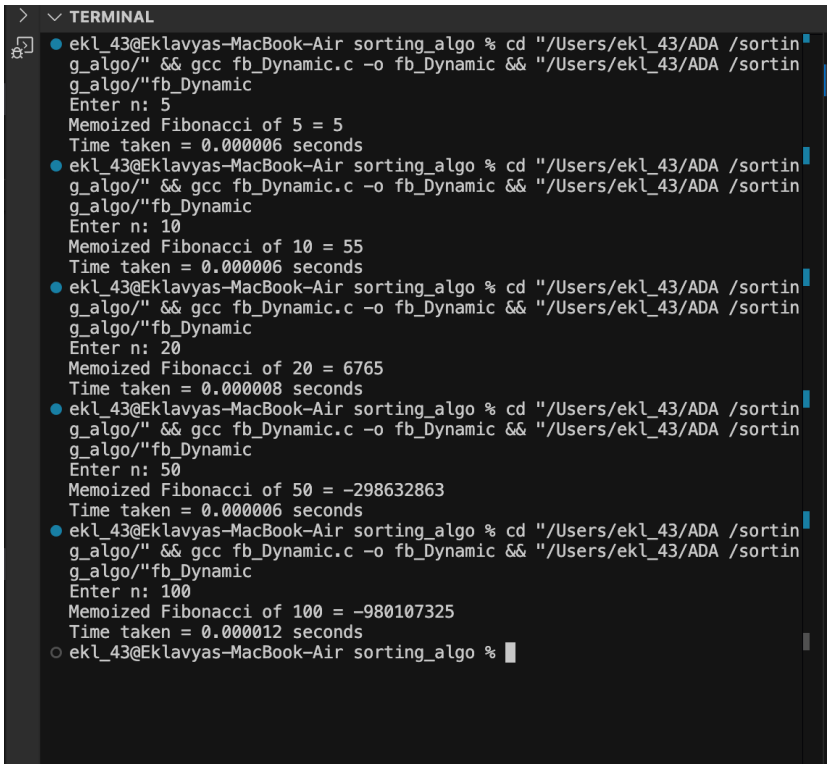


4(c) To implement and analyze different approaches for generating Fibonacci numbers and compare their time and space complexities using **memoization approach(Dynamic programming)**

C CODE:

```
#include <stdio.h>
#include <time.h>
#define MAX 10000
int memo[MAX];
// Memoization (Top-Down DP)
int fib_memo(int n) {
if (memo[n] != -1) return memo[n];
if (n <= 1) memo[n] = n;
else memo[n] = fib_memo(n-1) + fib_memo(n-2);
return memo[n];
}
int main() {
int n;printf("Enter n: ");
scanf("%d", &n);
for (int i = 0; i < MAX; i++) memo[i] = -1; // init
clock_t start = clock();
int result = fib_memo(n);
clock_t end = clock();
double time_taken = ((double)(end - start)) / CLOCKS_PER_SEC;
printf("Memoized Fibonacci of %d = %d\n", n, result);
printf("Time taken = %f seconds\n", time_taken);
return 0;
}
```

Output:

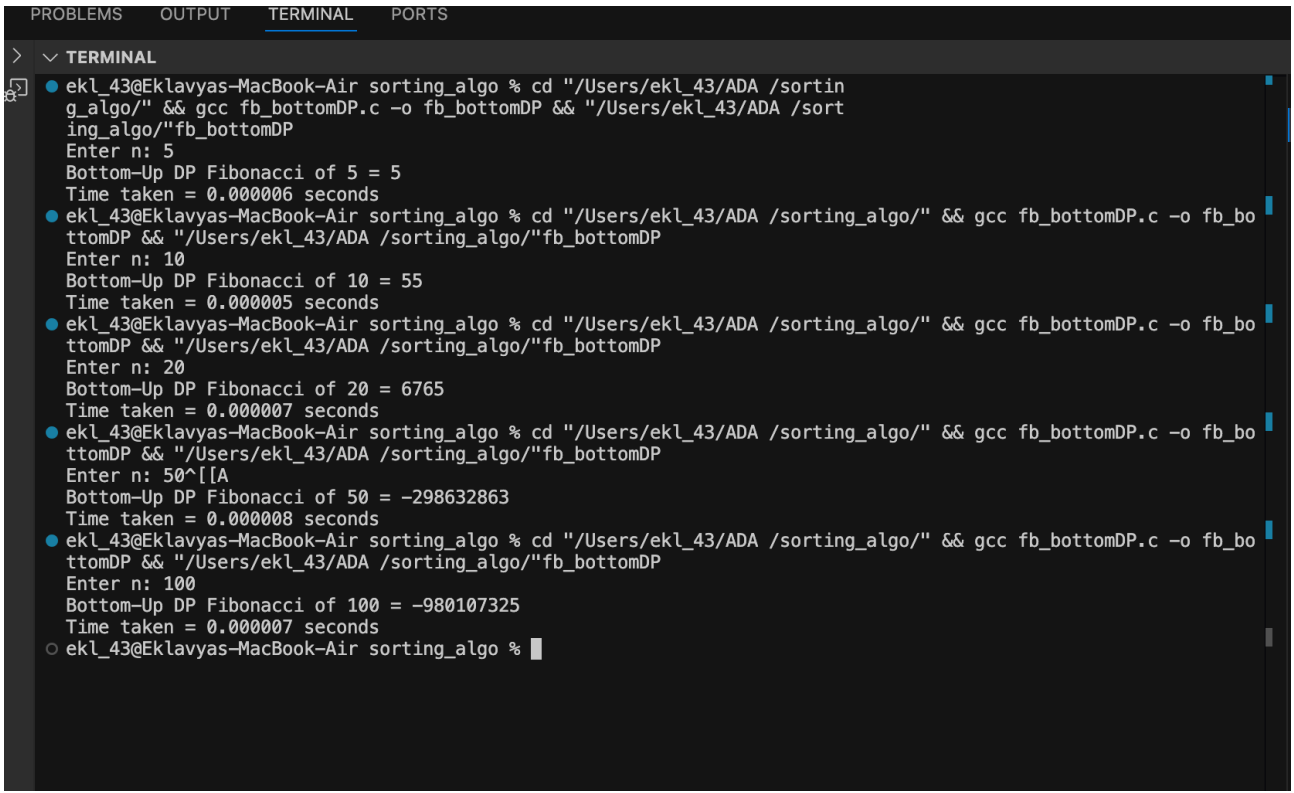


4(d) To implement and analyze different approaches for generating Fibonacci numbers and compare their time and space complexities using **Bottom Up Approach (Dynamic Programming)**

C CODE:

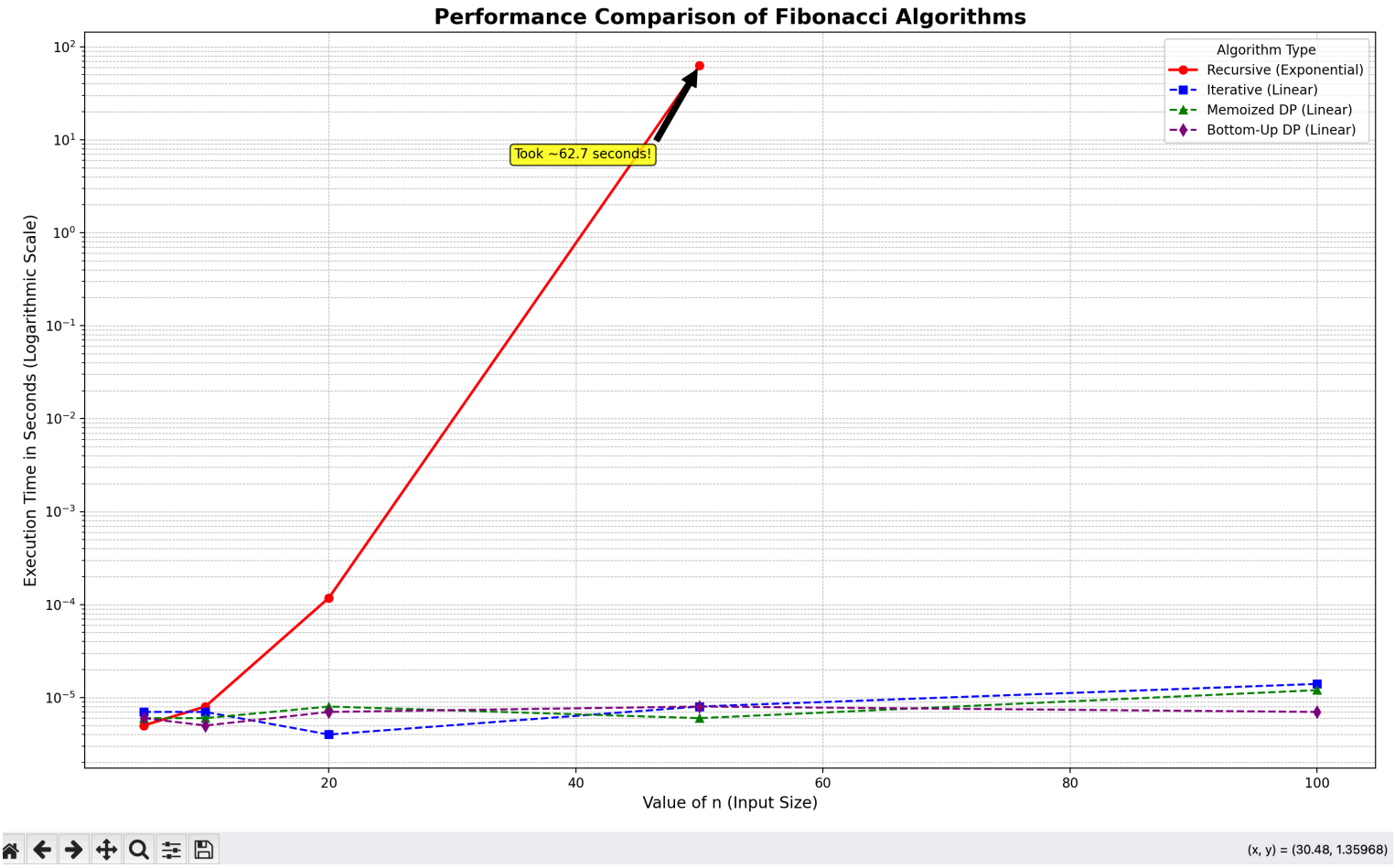
```
#include <stdio.h>
#include <time.h>
// Bottom-Up DP
int fib_bottomup(int n) {
if (n <= 1) return n;
int dp[n+1];
dp[0] = 0; dp[1] = 1;
for (int i = 2; i <= n; i++)
dp[i] = dp[i-1] + dp[i-2];
return dp[n];
}
int main() {
int n;
printf("Enter n: ");
scanf("%d", &n);
clock_t start = clock();
int result = fib_bottomup(n);
clock_t end = clock();
double time_taken = ((double)(end - start)) / CLOCKS_PER_SEC;
printf("Bottom-Up DP Fibonacci of %d = %d\n", n, result);
printf("Time taken = %f seconds\n", time_taken);
return 0;
}
```

Output:



Fibonacci Execution Time Comparison

Value of n	Recursive Method (seconds)	Iterative Method (seconds)	Memoized DP (Top-Down) (seconds)	Bottom-Up DP (Tabulation) (seconds)
5	0.000005	0.000007	0.000006	0.000006
10	0.000008	0.000007	0.000006	0.000005
20	0.000118	0.000004	0.000008	0.000007
50	62.664772	0.000008	0.000006	0.000008
100	Not run	0.000014	0.000012	0.000007



Space Complexity of Fibonacci Algorithms

Algorithm	Space Complexity	Explanation
Recursive (Naive)	$O(n)$	The space is dominated by the maximum depth of the recursion stack, which can be up to `n` levels deep.
Iterative	$O(1)$	This method uses a fixed number of variables, so its memory usage is constant regardless of `n`.
Memoized DP (Top-Down)	$O(n)$	Requires space for both the recursion stack ($O(n)$) and a memoization table ($O(n)$) to store results.
Bottom-Up DP (Tabulation)	$O(n)$	Uses an array of size `n+1` to store the Fibonacci numbers, leading to linear space usage.

CONCLUSION:

Recursive Approach:
• Simple and elegant, but **very slow for large n** (**O(2^n)**).
• **High space usage (O(n))** due to recursion stack.
• Practical only for small values of n.

Iterative Approach:
• Fast (O(n) time) and memory-efficient (O(1) space).
• Ideal for computing a **single Fibonacci number**.

• Memoization (Top-Down DP):
○ Reduces time complexity to O(n) using memoization.
○ **Space complexity O(n)** for memo array + recursion stack.
○ Useful when reusing previously computed Fibonacci numbers.

• Bottom-Up DP:
○ O(n) time and O(n) space, can be optimized to O(1) space.
○ Avoids recursion stack, **best for large n**.

Key Takeaways:
• Recursive method is impractical for large inputs.
• Iterative and optimized Bottom-Up DP are **fastest and most memory-efficient**.
• Memoization is a good balance if you want **recursion with efficiency**.

Objective: To implement and analyze GREEDY AND DYNAMIC approaches for solving Knapsack problem and compare their time and space complexities (through graph)

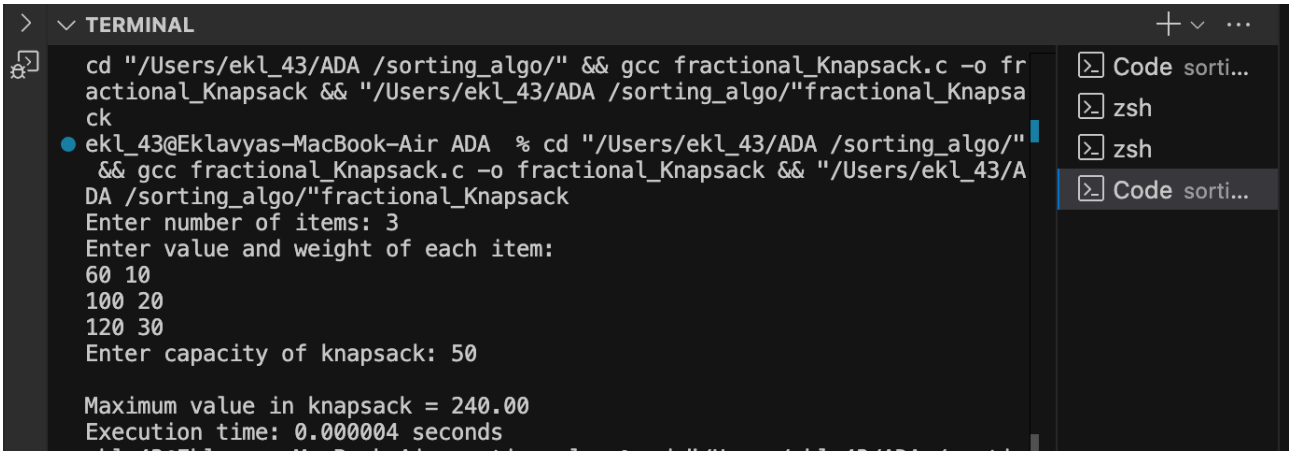
5(a) Fractional Knapsack

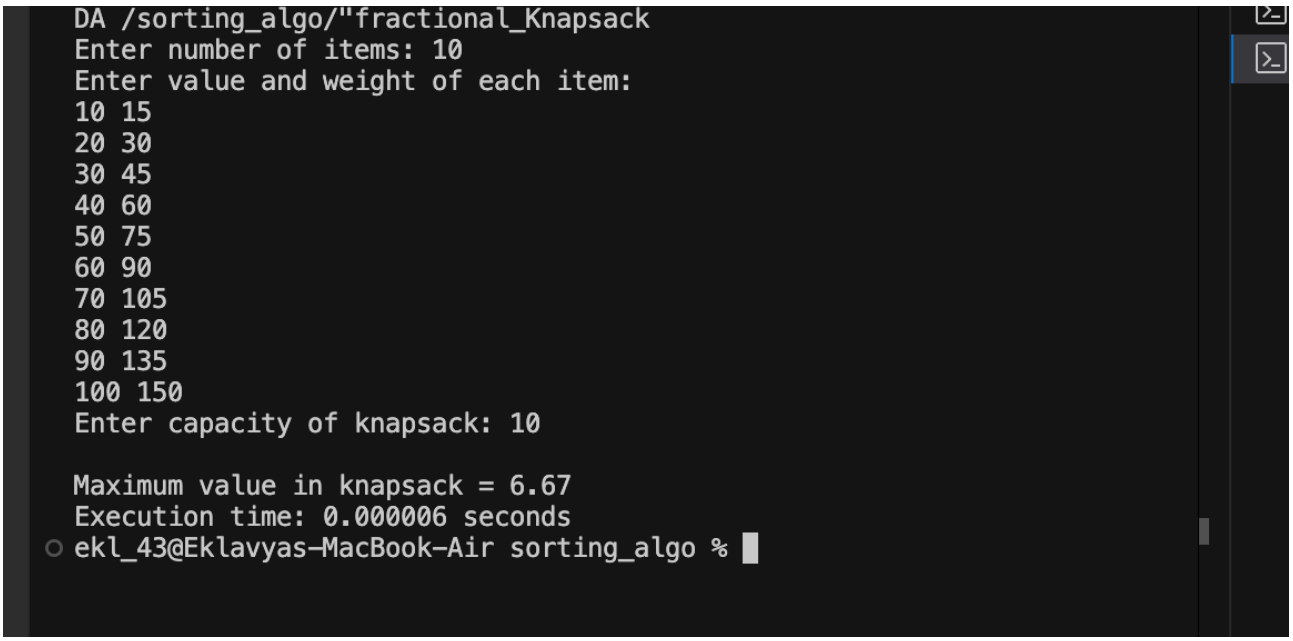
Code:

```
#include <stdio.h>
#include <time.h>

struct Item {
    int value, weight;
    float ratio;
};

// Function to sort items in decreasing order of ratio
void sortItems(struct Item items[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (items[j].ratio < items[j + 1].ratio) {
                struct Item temp = items[j];
                items[j] = items[j + 1];
                items[j + 1] = temp;
            }
        }
    }
}

int main() {
    int n;
    float capacity;
    printf("Enter number of items: ");
    scanf("%d", &n);

    struct Item items[n];
    printf("Enter value and weight of each item:\n");
    for (int i = 0; i < n; i++) {
        scanf("%d %d", &items[i].value, &items[i].weight);
        items[i].ratio = (float)items[i].value / items[i].weight;
    }

    printf("Enter capacity of knapsack: ");
    scanf("%f", &capacity);

    clock_t start, end;
    start = clock(); // start timer

    sortItems(items, n);

    float totalValue = 0.0;
    for (int i = 0; i < n; i++) {
        if (items[i].weight <= capacity) {
            totalValue += items[i].value;
            capacity -= items[i].weight;
```

```
        } else {
            totalValue += items[i].value * (capacity / items[i].weight);
            break;
        }
    }

    end = clock(); // end timer double time_taken = ((double)(end - start)) / CLOCKS_PER_SEC;

    printf("\nMaximum value in knapsack = %.2f", totalValue);
    printf("\nExecution time: %.6f seconds\n", time_taken);

    return 0;
}
```

Output:





Algorithm:

Step 1: Input the number of items n, knapsack capacity W,
weights w[i] and values v[i] for all items.

Step 2: Calculate value/weight ratio for each item.

Step 3: Sort all items in decreasing order of value/weight ratio.

Step 4: Initialize total_value = 0 and remaining_capacity = W.

Step 5: For each item i in sorted order:
    if (w[i] ≤ remaining_capacity)
        take the whole item
        total_value += v[i]
        remaining_capacity -= w[i]
    else
        take fractional part:
        total_value += v[i] * (remaining_capacity / w[i])
        remaining_capacity = 0
        break

Step 6: Print total_value as maximum profit.

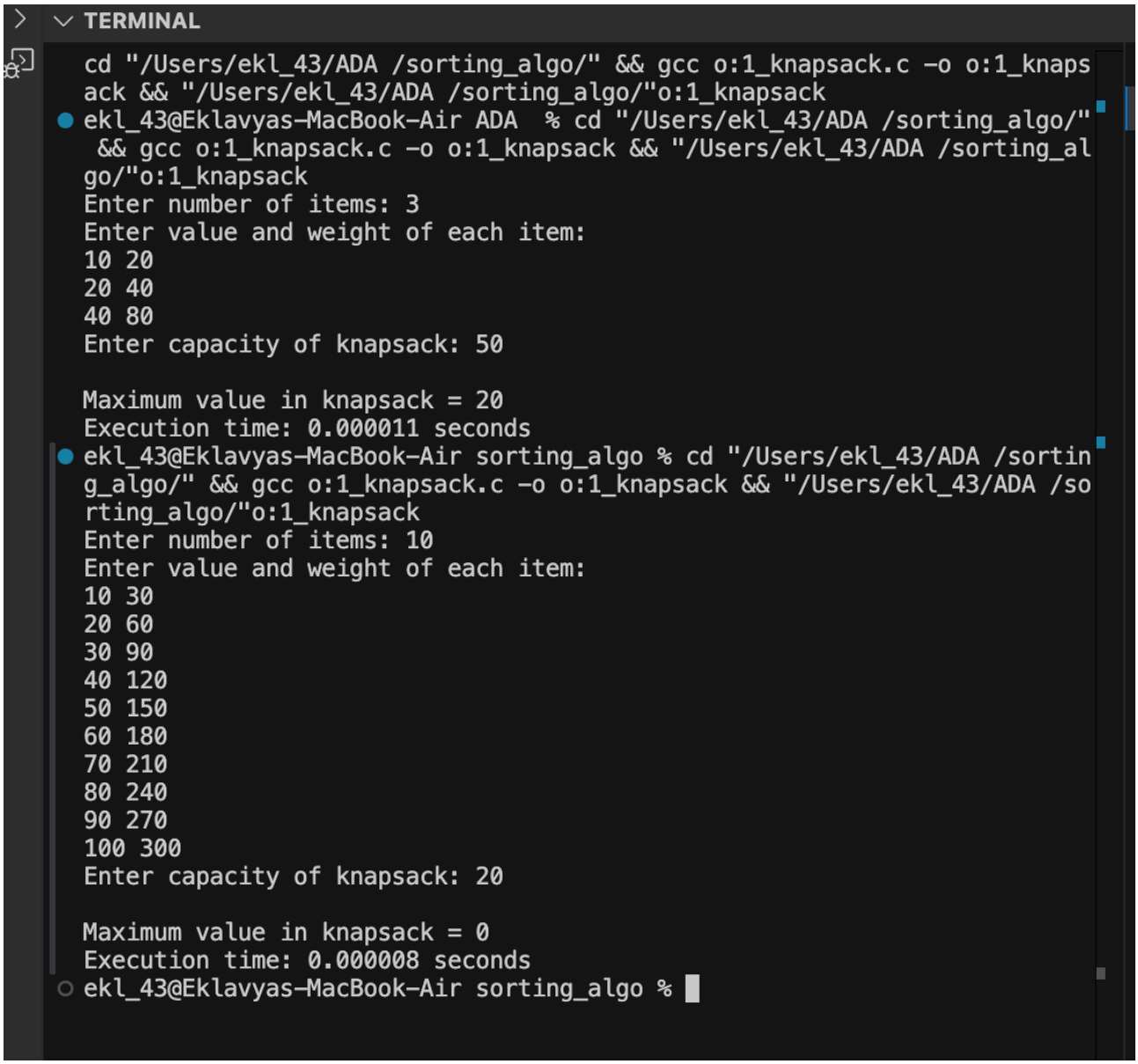
5(b) 0/1 Knapsack

Code:

```
#include <stdio.h>
#include <time.h>

int max(int a, int b) {
    return (a > b) ? a : b;
}

int knapsack(int W, int wt[], int val[], int n) {
    int dp[n + 1][W + 1];
    for (int i = 0; i <= n; i++) {
        for (int w = 0; w <= W; w++) {
            if (i == 0 || w == 0)
                dp[i][w] = 0;
            else if (wt[i - 1] <= w)
                dp[i][w] = max(val[i - 1] + dp[i - 1][w - wt[i - 1]], dp[i - 1][w]);
            else
                dp[i][w] = dp[i - 1][w];
        }
    }
    return dp[n][W];
}

int main() {
    int n, W;
    printf("Enter number of items: ");
    scanf("%d", &n);

    int val[n], wt[n];
    printf("Enter value and weight of each item:\n");
    for (int i = 0; i < n; i++)
```

```
        scanf("%d %d", &val[i], &wt[i]);

    printf("Enter capacity of knapsack: ");
    scanf("%d", &W);

    clock_t start, end;
    start = clock(); // start timer

    int result = knapsack(W, wt, val, n);

    end = clock(); // end timer
    double time_taken = ((double)(end - start)) / CLOCKS_PER_SEC;

    printf("\nMaximum value in knapsack = %d", result);
    printf("\nExecution time: %.6f seconds\n", time_taken);

    return 0;
}
```

Output:



Algorithm:

Step 1: Input number of items n, knapsack capacity W,
weights w[i] and values v[i].

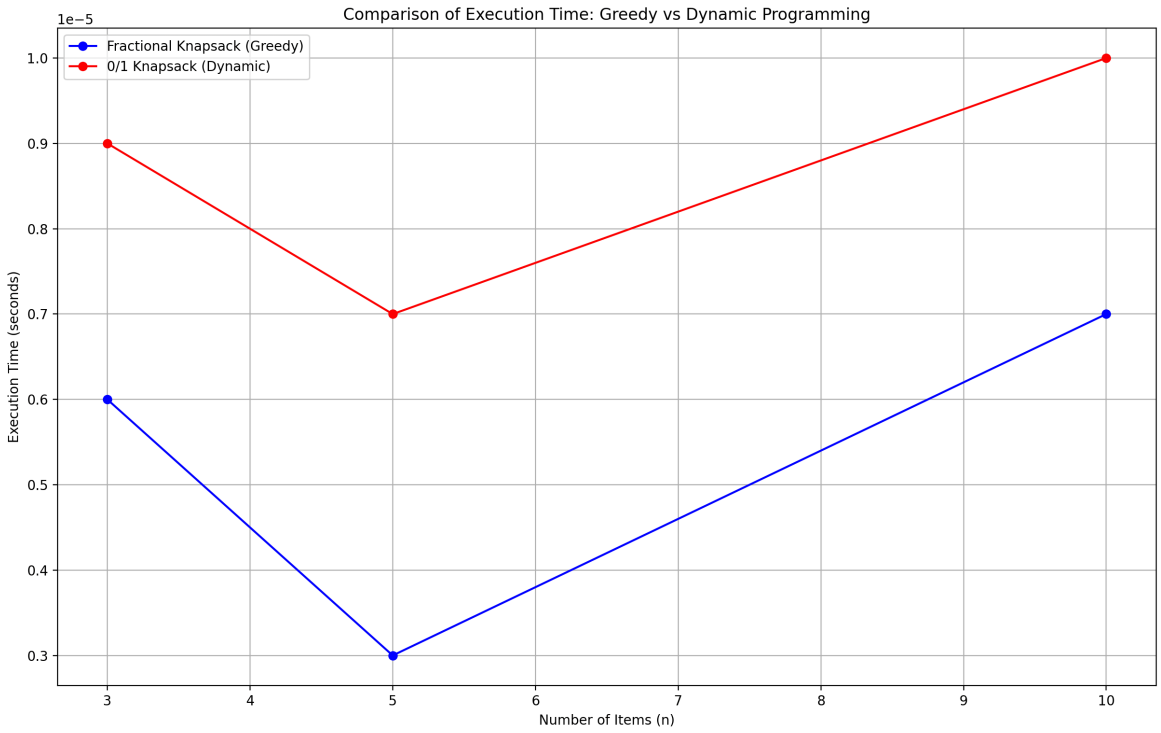
Step 2: Create a 2D array dp[n+1][W+1] where
dp[i][j] represents max value with i items and capacity j.

Step 3: Initialize dp[0][j] = 0 and dp[i][0] = 0.

Step 4: For i = 1 to n:
For j = 1 to W:
if (w[i-1] <= j)
dp[i][j] = max(v[i-1] + dp[i-1][j - w[i-1]], dp[i-1][j])
else
dp[i][j] = dp[i-1][j]

Step 5: Result = dp[n][W]

Step 6: Print dp[n][W] as maximum profit.



Conclusion:

Both Fractional Knapsack (Greedy Approach) and 0/1 Knapsack (Dynamic Programming Approach) were successfully implemented and executed.
The Fractional Knapsack algorithm selects items based on the highest value-to-weight ratio, allowing fractional inclusion of items. It provides an optimal solution for problems where items can be divided. Its time complexity is O(n log n) due to sorting, and it performs efficiently even for large input sizes.

The 0/1 Knapsack algorithm, implemented using Dynamic Programming, provides an exact optimal solution for cases where items cannot be divided. However, it requires more computation and memory, with time and space complexity O(nW), where W is the knapsack capacity.
The experimental comparison and graph show that the Greedy approach executes faster than the Dynamic Programming approach, but the Dynamic method guarantees exact optimality for 0/1 cases.

Hence, both algorithms were implemented successfully, tested for various inputs, and their time and space complexities were analyzed and compared.