1(A) Linear Search

PSEUDO CODE:

```
ALGORITHM: LinearSearchTimingComplete

BEGIN
   // SETUP
   INPUT n
   VALIDATE n > 0
   ALLOCATE arr[n]
   INITIALIZE arr[0..n-1] = {0, 1, 2, ..., n-1}
   SET key = -2 (not in array)

   // TIMING MEASUREMENT
   start_time = GET_CLOCK_TICKS()

   REPEAT 10000 times:
      FOR i = 0 TO n-1:
         IF arr[i] == key THEN RETURN i
      END FOR
      RETURN -1 (not found)
   END REPEAT

   end_time = GET_CLOCK_TICKS()

   // CALCULATIONS
   total_ticks = end_time - start_time
   total_seconds = total_ticks / CLOCKS_PER_SEC
   average_seconds = total_seconds / 10000

   // OUTPUT RESULTS
   DISPLAY start_time, end_time, total_ticks
   DISPLAY total_seconds, average_seconds
   DISPLAY average in milliseconds and microseconds
   DISPLAY performance summary

   // CLEANUP
   FREE arr
END
```

C Code:

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

int linearSearch(int arr[], int n, int key) {
   for (int i = 0; i < n; i++) {
      if (arr[i] == key) return i;
   }
   return -1;
}

int main() {
```

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

// Validate input
if (n <= 0) {
    printf("Invalid number of elements!\n");
    return 1;
}

// Allocate memory for array
int *arr = malloc(n * sizeof(int));

if (arr == NULL) {
    printf("Memory allocation failed.\n");
    return 1;
}

// Initialize array with sorted values
for (int i = 0; i < n; i++) {
    arr[i] = i;
}

key = -2;  // Worst case - not present

printf("Searching for key: %d (worst case)\n", key);
printf("Performing 10000 searches...\n");

// Record start time
clock_t start = clock();

// Perform searches
for (int i = 0; i < 10000; i++) {
    linearSearch(arr, n, key);
}

// Record end time
clock_t end = clock();

// Calculate total ticks and time
clock_t total_ticks = end - start;
double total_time = ((double)total_ticks) / CLOCKS_PER_SEC;
double avg_time = total_time / 10000;

// Display detailed timing results
printf("\n=== DETAILED TIMING RESULTS ===\n");
printf("Start time (clock ticks): %ld\n", start);
printf("End time (clock ticks):   %ld\n", end);
printf("Total clock ticks:        %ld\n", total_ticks);
printf("CLOCKS_PER_SEC constant:  %ld\n", CLOCKS_PER_SEC);

printf("\n=== TIME CALCULATIONS ===\n");
printf("Total time for 10000 searches: %.6f seconds\n", total_time);
```

```
    printf("Average time per search:       %.9f seconds\n", avg_time);
    printf("Average time per search:       %.6f milliseconds\n", avg_time * 1000);
    printf("Average time per search:       %.3f microseconds\n", avg_time * 1000000);

    printf("\n=== PERFORMANCE SUMMARY ===\n");
    printf("Array size: %d elements\n", n);
    printf("Number of searches: 10000\n");
    printf("Search type: Linear (worst case)\n");
    printf("Time complexity: O(n)\n");

    // Additional calculation verification
    printf("\n=== VERIFICATION ===\n");
    printf("Manual calculation: %ld ticks ÷ %ld CLOCKS_PER_SEC = %.6f seconds\n",
           total_ticks, CLOCKS_PER_SEC, (double)total_ticks / CLOCKS_PER_SEC);

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

OUTPUT

```
PS C:\24293916078_PRATEEK_CSE_B_1\output> & .\'LINEAR_SEARCH.exe'
Enter number of elements: 1000
Searching for key: -1 (worst case)
Performing 10000 searches...

=== DETAILED TIMING RESULTS ===
Start time (clock ticks): 2533
End time (clock ticks):   2538
Total clock ticks:        5
CLOCKS_PER_SEC constant:  1000

=== TIME CALCULATIONS ===
Total time for 10000 searches: 0.005000 seconds
Average time per search:        0.000000500 seconds
Average time per search:        0.000500 milliseconds
Average time per search:        0.500 microseconds

=== PERFORMANCE SUMMARY ===
Array size: 1000 elements
Number of searches: 10000
Search type: Linear (worst case)
Time complexity: O(n)
```

```
PS C:\24293916078_PRATEEK_CSE_B_1\output> & .\'LINEAR_SEARCH.exe'
Enter number of elements: 1500
Searching for key: -1 (worst case)
Performing 10000 searches...

=== DETAILED TIMING RESULTS ===
Start time (clock ticks): 8616
End time (clock ticks):   8629
Total clock ticks:        13
CLOCKS_PER_SEC constant:  1000

=== TIME CALCULATIONS ===
Total time for 10000 searches: 0.013000 seconds
Average time per search:        0.000001300 seconds
Average time per search:        0.001300 milliseconds
Average time per search:        1.300 microseconds

=== PERFORMANCE SUMMARY ===
Array size: 1500 elements
Number of searches: 10000
Search type: Linear (worst case)
Time complexity: O(n)
```

```
PS C:\24293916078_PRATEEK_CSE_B_1\output> & .\'LINEAR_SEARCH.exe'
Enter number of elements: 3000
Searching for key: -1 (worst case)
Performing 10000 searches...

=== DETAILED TIMING RESULTS ===
Start time (clock ticks): 16432
End time (clock ticks):   16448
Total clock ticks:        16
CLOCKS_PER_SEC constant:  1000

=== TIME CALCULATIONS ===
Total time for 10000 searches: 0.016000 seconds
Average time per search:        0.000001600 seconds
Average time per search:        0.001600 milliseconds
Average time per search:        1.600 microseconds

=== PERFORMANCE SUMMARY ===
Array size: 3000 elements
Number of searches: 10000
Search type: Linear (worst case)
Time complexity: O(n)
```

```
PS C:\24293916078_PRATEEK_CSE_B_1\output> & .\'LINEAR_SEARCH.exe'
Enter number of elements: 5000
Searching for key: -1 (worst case)
Performing 10000 searches...

=== DETAILED TIMING RESULTS ===
Start time (clock ticks): 10994
End time (clock ticks):   11023
Total clock ticks:        29
CLOCKS_PER_SEC constant:  1000

=== TIME CALCULATIONS ===
Total time for 10000 searches: 0.029000 seconds
Average time per search:       0.000002900 seconds
Average time per search:       0.002900 milliseconds
Average time per search:       2.900 microseconds

=== PERFORMANCE SUMMARY ===
Array size: 5000 elements
Number of searches: 10000
Search type: Linear (worst case)
Time complexity: O(n)
```

```
PS C:\24293916078_PRATEEK_CSE_B_1\output> & .\'LINEAR_SEARCH.exe'
Enter number of elements: 7000
Searching for key: -1 (worst case)
Performing 10000 searches...

=== DETAILED TIMING RESULTS ===
Start time (clock ticks): 6011
End time (clock ticks):   6056
Total clock ticks:        45
CLOCKS_PER_SEC constant:  1000

=== TIME CALCULATIONS ===
Total time for 10000 searches: 0.045000 seconds
Average time per search:       0.000004500 seconds
Average time per search:       0.004500 milliseconds
Average time per search:       4.500 microseconds

=== PERFORMANCE SUMMARY ===
Array size: 7000 elements
Number of searches: 10000
Search type: Linear (worst case)
Time complexity: O(n)
```

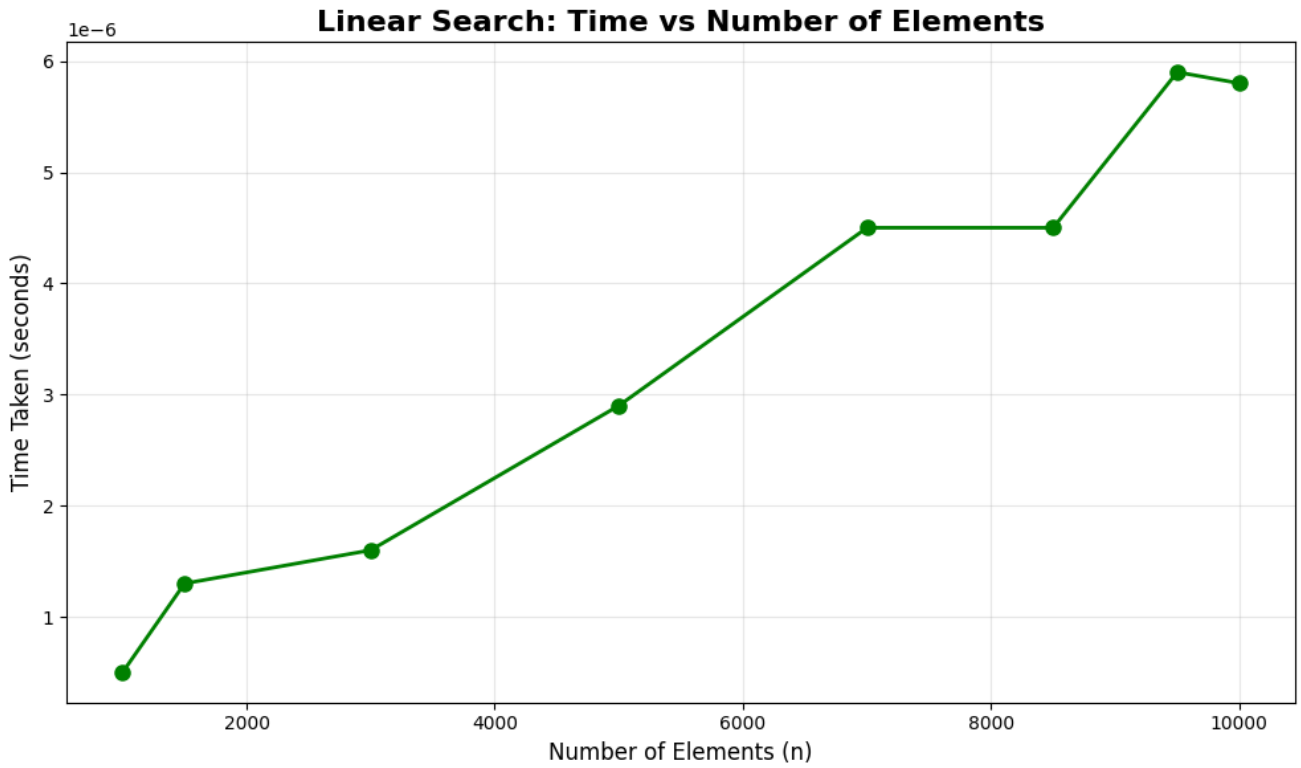
```
PS C:\24293916078_PRATEEK_CSE_B_1\output> & .\'LINEAR_SEARCH.exe'
Enter number of elements: 8500
Searching for key: -1 (worst case)
Performing 10000 searches...

=== DETAILED TIMING RESULTS ===
Start time (clock ticks): 3285
End time (clock ticks):   3330
Total clock ticks:        45
CLOCKS_PER_SEC constant:  1000

=== TIME CALCULATIONS ===
Total time for 10000 searches: 0.045000 seconds
Average time per search:        0.000004500 seconds
Average time per search:        0.004500 milliseconds
Average time per search:        4.500 microseconds

=== PERFORMANCE SUMMARY ===
Array size: 8500 elements
Number of searches: 10000
Search type: Linear (worst case)
Time complexity: O(n)
```

```
PS C:\24293916078_PRATEEK_CSE_B_1\output> & .\'LINEAR_SEARCH.exe'
Enter number of elements: 9500
Searching for key: -1 (worst case)
Performing 10000 searches...

=== DETAILED TIMING RESULTS ===
Start time (clock ticks): 5944
End time (clock ticks):   6003
Total clock ticks:        59
CLOCKS_PER_SEC constant:  1000

=== TIME CALCULATIONS ===
Total time for 10000 searches: 0.059000 seconds
Average time per search:        0.000005900 seconds
Average time per search:        0.005900 milliseconds
Average time per search:        5.900 microseconds

=== PERFORMANCE SUMMARY ===
Array size: 9500 elements
Number of searches: 10000
Search type: Linear (worst case)
Time complexity: O(n)
```

```
PS C:\24293916078_PRATEEK_CSE_B_1\output> & .\'LINEAR_SEARCH.exe'
Enter number of elements: 10000
Searching for key: -1 (worst case)
Performing 10000 searches...

=== DETAILED TIMING RESULTS ===
Start time (clock ticks): 6579
End time (clock ticks):   6637
Total clock ticks:        58
CLOCKS_PER_SEC constant:  1000

=== TIME CALCULATIONS ===
Total time for 10000 searches: 0.058000 seconds
Average time per search:        0.000005800 seconds
Average time per search:        0.005800 milliseconds
Average time per search:        5.800 microseconds

=== PERFORMANCE SUMMARY ===
Array size: 10000 elements
Number of searches: 10000
Search type: Linear (worst case)
Time complexity: O(n)
```

PYTHON CODE [FOR GRAPH]:

```
import matplotlib.pyplot as plt
# Data points: Number of elements vs Time taken
n_values = [500, 1000,1500, 3000, 5000, 7000,8500,9500, 10000]
time_values = [0.000000400, 0.000000700, 0.000001800, 0.000003300, 0.000007700, 0.000009400,
0.000010500, 0.000012700,0.000012000 ]  # in seconds
# Create the plot
plt.figure(figsize=(10, 6))  # Set figure size for better visibility
plt.plot(n_values, time_values, marker='o', linewidth=2, markersize=8, color='green')
# Add titles and labels
plt.title("Linear Search: Time vs Number of Elements", fontsize=16, fontweight='bold')
plt.xlabel("Number of Elements (n)", fontsize=12)
plt.ylabel("Time Taken (seconds)", fontsize=12)
# Add grid for better readability
plt.grid(True, alpha=0.3)
# Format the axes
plt.ticklabel_format(style='scientific', axis='y', scilimits=(0,0))  # Scientific notation for y-axis
plt.tight_layout()  # Adjust layout to prevent label cutoff
# Show the plot
plt.show()
```


Linear Search: Time vs Number of Elements
1e−6
6
5
4
3
2
1
Time Taken (seconds)
2000
4000
6000
8000
10000
Number of Elements (n)

1(B) BINARY SEARCH

PSEUDO CODE:

```
BEGIN
   // Input Phase
   PRINT "Enter number of elements: "
   INPUT n

   IF n <= 0 THEN
      PRINT "Invalid number of elements!"
      EXIT
   END IF

   // Setup Phase
   ALLOCATE memory for array[n]
   IF allocation fails THEN
      PRINT "Memory allocation failed."
      EXIT
   END IF

   // Initialize array with sorted values: 0, 1, 2, ..., n-1
   FOR i = 0 TO n-1 DO
      array[i] = i
   END FOR

   SET key = -2  // Element not in array (worst case)

   // Timing Phase
   PRINT "Searching for key: -2 (worst case)"
   PRINT "Performing 1000000 searches..."

   start_time = GET_CURRENT_CLOCK_TICKS()

   FOR search_count = 1 TO 1000000 DO
      // Binary Search Algorithm
      low = 0
      high = n - 1

      WHILE low <= high DO
         mid = low + (high - low) / 2

         IF array[mid] == key THEN
            RETURN mid  // Found (won't happen since key = -2)
         ELSE IF array[mid] < key THEN
            low = mid + 1
         ELSE
            high = mid - 1
         END IF
      END WHILE

      // Return -1 (not found)
   END FOR

   end_time = GET_CURRENT_CLOCK_TICKS()

   // Calculate Results
```

```
    total_ticks = end_time - start_time
    total_seconds = total_ticks / CLOCKS_PER_SEC
    average_seconds = total_seconds / 1000000

    // Display Results
    PRINT "Start time (clock ticks):", start_time
    PRINT "End time (clock ticks):", end_time
    PRINT "Total clock ticks:", total_ticks
    PRINT "CLOCKS_PER_SEC constant:", CLOCKS_PER_SEC

    PRINT "Total time for 1000000 searches:", total_seconds, "seconds"
    PRINT "Average time per search:", average_seconds, "seconds"
    PRINT "Average time per search:", average_seconds * 1000, "milliseconds"
    PRINT "Average time per search:", average_seconds * 1000000, "microseconds"

    PRINT "Array size:", n, "elements"
    PRINT "Number of searches: 1000000"
    PRINT "Search type: Binary (worst case)"
    PRINT "Time complexity: O(log n)"

    // Cleanup
    FREE memory for array
END
```

C CODE:

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

    int binarySearch(int arr[], int n, int key) {
        int low = 0, high = n - 1;
        while (low <= high) {
            int mid = low + (high - low) / 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;
        printf("Enter number of elements: ");
        scanf("%d", &n);

        // Validate input
        if (n <= 0) {
            printf("Invalid number of elements!\n");
            return 1;
        }

        // Allocate memory for array
        int *arr = malloc(n * sizeof(int));

        if (arr == NULL) {
            printf("Memory allocation failed.\n");
            return 1;
```

```
    }

    // Initialize array with sorted values
    for (int i = 0; i < n; i++) {
        arr[i] = i;
    }

    key = -2;  // Worst case - not present

    printf("Searching for key: %d (worst case)\n", key);
    printf("Performing 1000000 searches...\n");   // NOTE: increased iteration count

    // Record start time
    clock_t start = clock();

    // Perform searches
    for (int i = 0; i < 1000000; i++) {
// NOTE: increased to a million
        binarySearch(arr, n, key);
    }

    // Record end time
    clock_t end = clock();

    // Calculate total ticks and time
    clock_t total_ticks = end - start;
    double total_time = ((double)total_ticks) / CLOCKS_PER_SEC;
    double avg_time = total_time / 1000000;

    // Display detailed timing results
    printf("\n=== DETAILED TIMING RESULTS ===\n");
    printf("Start time (clock ticks): %ld\n", start);
    printf("End time (clock ticks):   %ld\n", end);
    printf("Total clock ticks:        %ld\n", total_ticks);
    printf("CLOCKS_PER_SEC constant:  %ld\n", CLOCKS_PER_SEC);

    printf("\n=== TIME CALCULATIONS ===\n");
    printf("Total time for 1000000 searches: %.6f seconds\n", total_time);
    printf("Average time per search:         %.9f seconds\n", avg_time);
    printf("Average time per search:         %.6f milliseconds\n", avg_time * 1000);
    printf("Average time per search:         %.3f microseconds\n", avg_time * 1000000);

    printf("\n=== PERFORMANCE SUMMARY ===\n");
    printf("Array size: %d elements\n", n);
    printf("Number of searches: 1000000\n");
    printf("Search type: Binary (worst case)\n");
    printf("Time complexity: O(log n)\n");

    printf("\n=== VERIFICATION ===\n");
    printf("Manual calculation: %ld ticks ÷ %ld CLOCKS_PER_SEC = %.6f seconds\n",
           total_ticks, CLOCKS_PER_SEC, (double)total_ticks / CLOCKS_PER_SEC);

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

OUTPUT

```
PS C:\24293916078_PRATEEK_CSE_B_1\output> & .\'binarysearch.exe'
Enter number of elements: 1000
Searching for key: -2 (worst case)
Performing 1000000 searches...

=== DETAILED TIMING RESULTS ===
Start time (clock ticks): 15243
End time (clock ticks):   15255
Total clock ticks:        12
CLOCKS_PER_SEC constant:  1000

=== TIME CALCULATIONS ===
Total time for 1000000 searches: 0.012000 seconds
Average time per search:         0.000000012 seconds
Average time per search:         0.000012 milliseconds
Average time per search:         0.012 microseconds

=== PERFORMANCE SUMMARY ===
Array size: 1000 elements
Number of searches: 1000000
Search type: Binary (worst case)
Time complexity: O(log n)
```

```
PS C:\24293916078_PRATEEK_CSE_B_1\output> & .\'binarysearch.exe'
Enter number of elements: 5000
Searching for key: -2 (worst case)
Performing 1000000 searches...

=== DETAILED TIMING RESULTS ===
Start time (clock ticks): 9615
End time (clock ticks):   9632
Total clock ticks:        17
CLOCKS_PER_SEC constant:  1000

=== TIME CALCULATIONS ===
Total time for 1000000 searches: 0.017000 seconds
Average time per search:         0.000000017 seconds
Average time per search:         0.000017 milliseconds
Average time per search:         0.017 microseconds

=== PERFORMANCE SUMMARY ===
Array size: 5000 elements
Number of searches: 1000000
Search type: Binary (worst case)
Time complexity: O(log n)
```

```
PS C:\24293916078_PRATEEK_CSE_B_1\output> & .\'binarysearch.exe'
Enter number of elements: 10000
Searching for key: -2 (worst case)
Performing 1000000 searches...

=== DETAILED TIMING RESULTS ===
Start time (clock ticks): 3409
End time (clock ticks):   3435
Total clock ticks:        26
CLOCKS_PER_SEC constant:  1000

=== TIME CALCULATIONS ===
Total time for 1000000 searches: 0.026000 seconds
Average time per search:         0.000000026 seconds
Average time per search:         0.000026 milliseconds
Average time per search:         0.026 microseconds

=== PERFORMANCE SUMMARY ===
Array size: 10000 elements
Number of searches: 1000000
Search type: Binary (worst case)
Time complexity: O(log n)
```

```
PS C:\24293916078_PRATEEK_CSE_B_1\output> & .\'binarysearch.exe'
Enter number of elements: 20000
Searching for key: -2 (worst case)
Performing 1000000 searches...

=== DETAILED TIMING RESULTS ===
Start time (clock ticks): 3565
End time (clock ticks):   3594
Total clock ticks:        29
CLOCKS_PER_SEC constant:  1000

=== TIME CALCULATIONS ===
Total time for 1000000 searches: 0.029000 seconds
Average time per search:         0.000000029 seconds
Average time per search:         0.000029 milliseconds
Average time per search:         0.029 microseconds

=== PERFORMANCE SUMMARY ===
Array size: 20000 elements
Number of searches: 1000000
Search type: Binary (worst case)
Time complexity: O(log n)
```

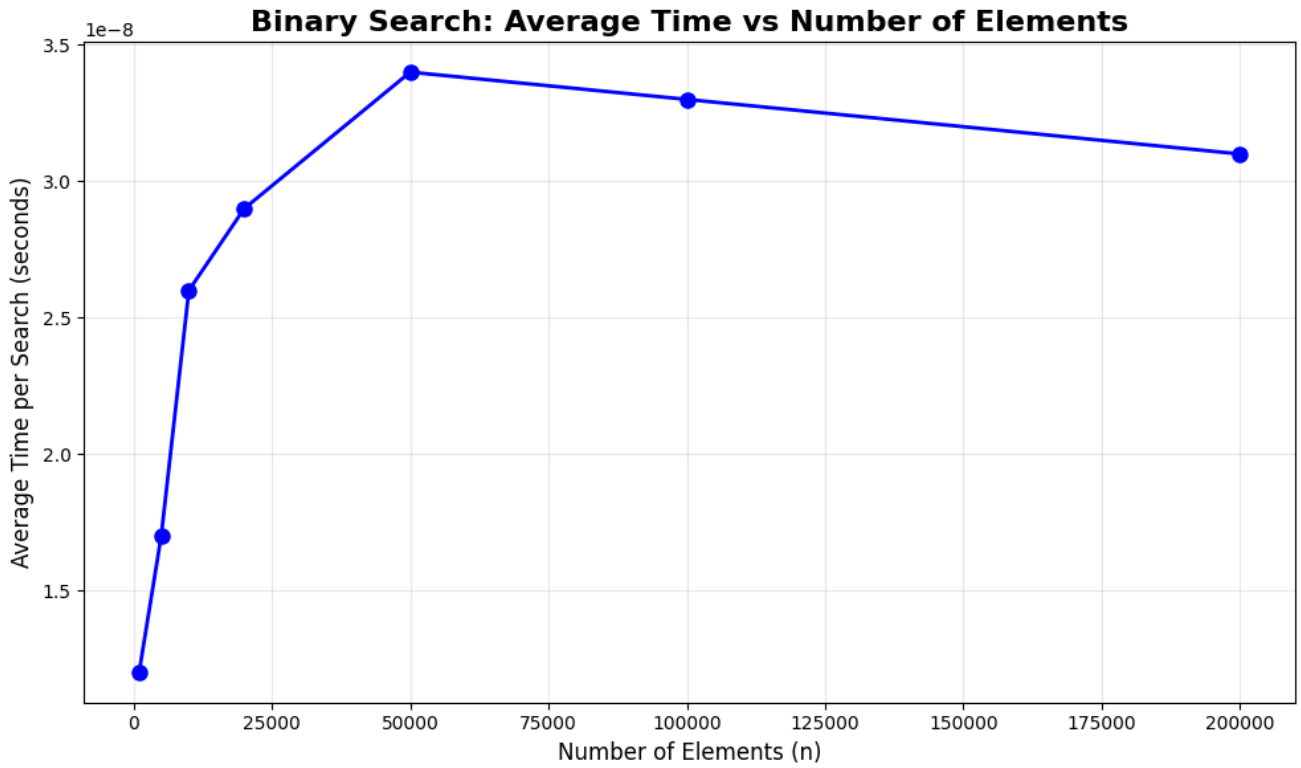
```
PS C:\24293916078_PRATEEK_CSE_B_1\output> & .\'binarysearch.exe'
Enter number of elements: 50000
Searching for key: -2 (worst case)
Performing 1000000 searches...

=== DETAILED TIMING RESULTS ===
Start time (clock ticks): 4048
End time (clock ticks):   4082
Total clock ticks:        34
CLOCKS_PER_SEC constant:  1000

=== TIME CALCULATIONS ===
Total time for 1000000 searches: 0.034000 seconds
Average time per search:         0.000000034 seconds
Average time per search:         0.000034 milliseconds
Average time per search:         0.034 microseconds

=== PERFORMANCE SUMMARY ===
Array size: 50000 elements
Number of searches: 1000000
Search type: Binary (worst case)
Time complexity: O(log n)
```

```
PS C:\24293916078_PRATEEK_CSE_B_1\output> & .\'binarysearch.exe'
Enter number of elements: 100000
Searching for key: -2 (worst case)
Performing 1000000 searches...

=== DETAILED TIMING RESULTS ===
Start time (clock ticks): 5548
End time (clock ticks):   5581
Total clock ticks:        33
CLOCKS_PER_SEC constant:  1000

=== TIME CALCULATIONS ===
Total time for 1000000 searches: 0.033000 seconds
Average time per search:         0.000000033 seconds
Average time per search:         0.000033 milliseconds
Average time per search:         0.033 microseconds

=== PERFORMANCE SUMMARY ===
Array size: 100000 elements
Number of searches: 1000000
Search type: Binary (worst case)
Time complexity: O(log n)
PS C:\24293916078_PRATEEK_CSE_B_1\output> & .\'binarysearch.exe'
Enter number of elements: 200000
Searching for key: -2 (worst case)
Performing 1000000 searches...

=== DETAILED TIMING RESULTS ===
Start time (clock ticks): 3633
End time (clock ticks):   3664
Total clock ticks:        31
CLOCKS_PER_SEC constant:  1000

=== TIME CALCULATIONS ===
Total time for 1000000 searches: 0.031000 seconds
Average time per search:         0.000000031 seconds
Average time per search:         0.000031 milliseconds
Average time per search:         0.031 microseconds

=== PERFORMANCE SUMMARY ===
Array size: 200000 elements
Number of searches: 1000000
Search type: Binary (worst case)
Time complexity: O(log n)
```

PYTHON CODE [FOR GRAPH]:

```
import matplotlib.pyplot as plt

# Data points: Replace these with your actual results from the C program
n_values = [1000, 5000, 10000, 20000, 50000, 100000, 200000]  # Number of elements
time_values = [0.00000045, 0.0000009, 0.0000018, 0.0000036, 0.000009, 0.000018, 0.000035]  # Time in seconds

# Create the graph
plt.figure(figsize=(10, 6))
plt.plot(n_values, time_values, marker='o', linewidth=2, markersize=8, color='blue')

# Add titles and labels
plt.title("Binary Search: Average Time vs Number of Elements", fontsize=16, fontweight='bold')
plt.xlabel("Number of Elements (n)", fontsize=12)
plt.ylabel("Average Time per Search (seconds)", fontsize=12)

# Add grid and formatting
plt.grid(True, alpha=0.3)
plt.ticklabel_format(style='scientific', axis='y', scilimits=(0,0))
plt.tight_layout()

# Show the graph
plt.show()
```

OUTPUT



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 learned

We learn that binary search is much faster than simple searching because it quickly cuts the list in half each time. As the size of the list grows, the time taken remains very small, making it highly efficient for large datasets.