ALGORITHMS LABORATORY
[CS-2098]
Individual Work

Lab. No.- 5 Date. 03/10/2020

Roll Number:	1905083	Branch/Section:	CSE/CS-2
Name in Capital:	AMIL UTKARSH GUPTA		

Program No: 1.1

Program Title:

Write a program to search an element x in an array of n integers using binary search algorithm that uses divide and conquer technique. Find out the best case, worst case and average case time complexities for different values of n and plot a graph of the time taken versus n. The n integers can be generated randomly and x can be choosen randomly, or any element of the array or middle or last element of the array depending on type of time complexity analysis.

Input/Output Screenshots:

RUN-1:

```
PS C:\Algo-Lab\Lab-4> gcc .\q1.c
PS C:\Algo-Lab\Lab-4> ./a
Enter:
        0 to EXIT
        1 to change n
        2 to find best case time complexity
        3 to find worst case time complexity
        4 to find the average case time complexity
1
Enter new value of n
99999

Enter:
        0 to EXIT
        1 to change n
        2 to find best case time complexity
        3 to find worst case time complexity
        4 to find the average case time complexity
2
Best case time taken = 0.000000 ticks
Best case time taken = 0.000000 seconds
Enter:
        0 to EXIT
        1 to change n
        2 to find best case time complexity
        3 to find worst case time complexity
        4 to find the average case time complexity
3
Worst case time taken = 0.000000 ticks
Worst case time taken = 0.000000 seconds
Enter:
        0 to EXIT
        1 to change n
        2 to find best case time complexity
        3 to find worst case time complexity
        4 to find the average case time complexity
4
Average case time taken = 0.000000 ticks
Average case time taken = 0.000000 seconds
Enter:
        0 to EXIT
        1 to change n
        2 to find best case time complexity
        3 to find worst case time complexity
        4 to find the average case time complexity
```

Source code

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

int choose_random(int n)
{
   return (rand()%n);
}

void destroy_prev_allocation(int *a, int n) // Destroys previously (dynamically allocated) memory from the array
{
   int i;
   for(i=0; i<n; i++)
   {
      free(a + i);
   }
}

void insert_rand_array(int *a, int n) // Inserts n random values into the array
{
   int i;
   for(i=0; i<n; i++)
   {
      a[i] = rand();
   }
}

void merge(int *a, int p, int q, int r)
{
   int i = p, j = q+1, next = 0;
   int sorted[r-p+1];
   while( (i<=q) && (j<=r) )
   {
      if(a[i] <= a[j])
      {
         sorted[next++] = a[i++];
      }

      else
      {
         sorted[next++] = a[j++];
      }
   }
   while(i<=q)
   {
```

```
        sorted[next++] = a[i++];
    }
    while(j<=r)
    {
        sorted[next++] = a[j++];
    }
    for(i=p; i<=r; i++) //time complexity of the step remains O(n)
    {
        a[i] = sorted[i-p];
    }
}

void merge_sort_ascending(int *a, int p, int r) // Sorts the array in ascending order using merge sort
{
    int q;
    if(p<r)
    {
        q = (p+r-1)/2;
        merge_sort_ascending(a, p, q);
        merge_sort_ascending(a, q+1, r);
        merge(a, p, q, r);
    }
}

int binary_search(int *a, int e, int l, int h)
{
    int m = (l+h)/2;
    if(a[m] == e)
    {
        return m;
    }
    else if(l == h) //exit condition
    {
        return -1;
    }
    else if(a[m] > e)
    {
        return binary_search(a, e, l, m-1);
    }
    else
    {
        return binary_search(a, e, m+1, h);
    }
}

int main()
{
    int n = 0, *a, i, choice = 1;
```

```
clock_t start, end;
double time;

while(choice)
{
   printf("Enter: \n\t0 to EXIT"
   "\n\t1 to change n"
   "\n\t2 to find best case time complexity"
   "\n\t3 to find worst case time complexity"
   "\n\t4 to find the average case time complexity\n");
   scanf("%d", &choice);
   switch(choice)
   {
      case 0:
         printf("EXITING\n");
         break;
      case 1:
         if(n>0)
         {
            destroy_prev_allocation(a, n);
         }
         printf("Enter new value of n\n");
         scanf("%d", &n);
         a = (int*) malloc(sizeof(int) * n);
         insert_rand_array(a, n);
         merge_sort_ascending(a, 0, n-1);
         // printf("Sorted array :\n");
         // for(i=0; i<n; i++)
         // {
         //    printf("%d\t", a[i]);
         // }
         printf("\n");
         break;
      case 2:
         start = clock();
         binary_search(a, a[(n-1)/2], 0, n-1);
         end=clock();
         time = (double) (end-start);
         printf("Best case time taken = %lf ticks\n", time);
         time = ((double) (end - start) )/CLOCKS_PER_SEC;
         printf("Best case time taken = %lf seconds\n", time);
         break;
      case 3:
         start = clock();
         binary_search(a, a[0], 0, n-1);
         end=clock();
         time = (double) (end-start);
         printf("Worst case time taken = %lf ticks\n", time);
```

```
            time = ((double) (end - start) )/CLOCKS_PER_SEC;
            printf("Worst case time taken = %lf seconds\n", time);
            break;
        case 4:
            start = clock();
            binary_search(a, a[choose_random(n)], 0, n-1);
            end = clock();
            time = (double) (end-start);
            printf("Average case time taken = %lf ticks\n", time);
            time = ((double) (end - start) )/CLOCKS_PER_SEC;
            printf("Average case time taken = %lf seconds\n", time);
            break;
        default:
            printf("INVALID CHOICE, try again\n");
      }
   }
   if(n>0)
   {
      destroy_prev_allocation(a, n);
   }
   return 0;
}
```

Conclusion/Observation

Thus, the best, worst and average case time taken is found for binary search.

Program No: 2.2

Program Title:

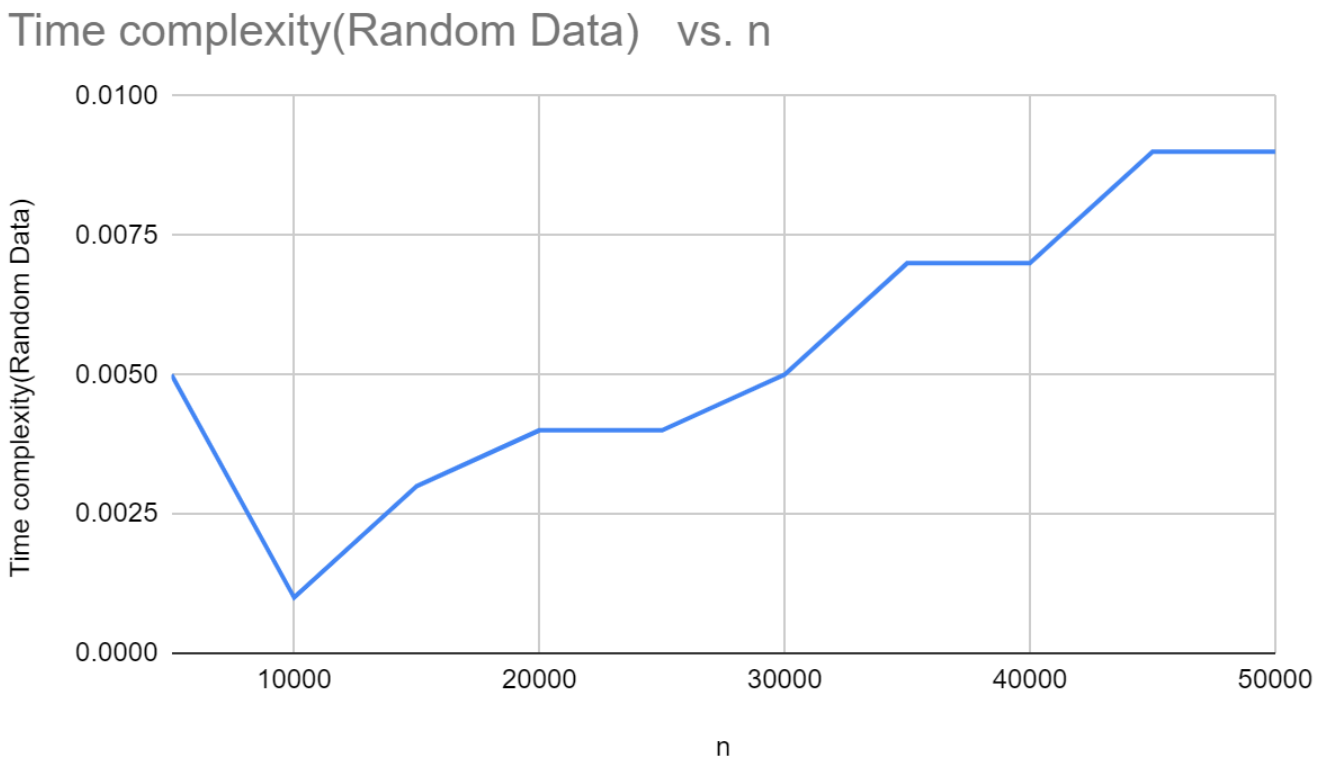
Write a program to sort a list of n elements using the merge sort method and determine the time required to sort the elements. Repeat the experiment for different values of n and different nature of data (random data, sorted data, reversely sorted data) in the list. n is the user input and n integers can be generated randomly. Finally plot a graph of the time taken versus n

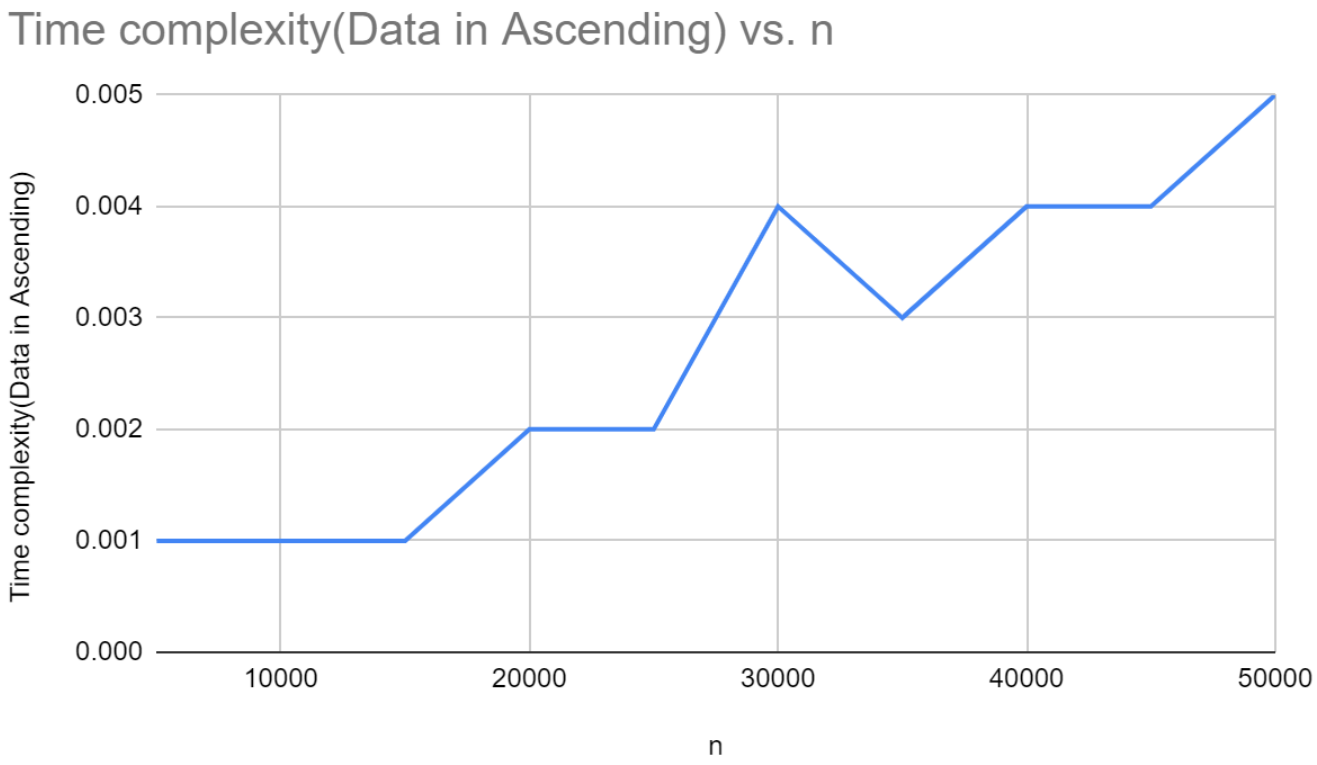
Input/Output Screenshots:

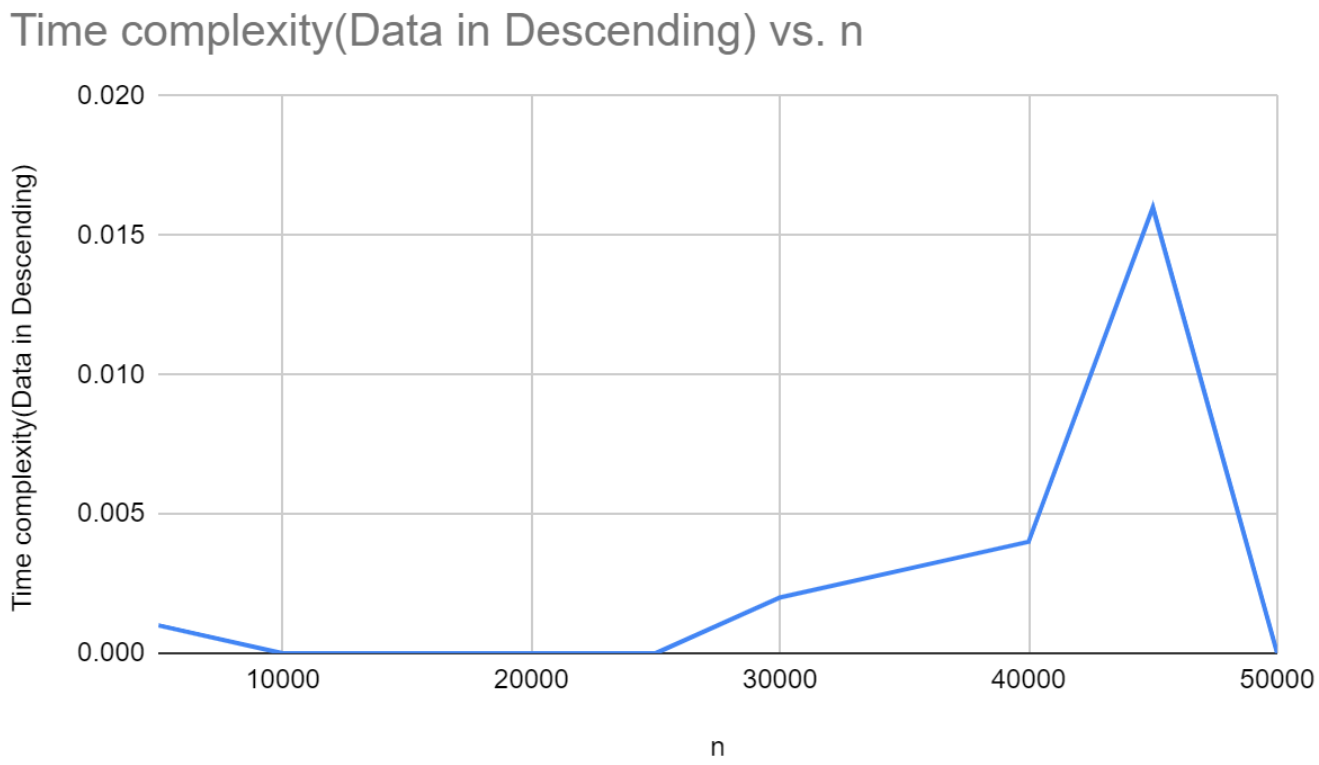
RUN-1:

```
PS C:\Algo-Lab\Lab-4> gcc .\q2.c
PS C:\Algo-Lab\Lab-4> ./a

0. Quit
1. n Random numbers=>Array
2. Display the Array
3. Sort the Array in Ascending Order by using merge sort Algorithm
4. Sort the Array in Descending Order by using any sorting algorithm
5. Time Complexity to sort ascending of random data
6. Time Complexity to sort ascending of data already sorted in ascending order
7. Time Complexity to sort ascending of data already sorted in descending order
8. Time Complexity to sort ascending of data for all Cases (Data Ascending, Data in Descending & Random Data) in Tabular form for values n=5000 to 50000, step=5000
8
SI No.  Value of n      Time complexity(Random Data)    Time complexity(Data in Ascending)      Time complexity(Data in Descending)
1       5000            0.005000                        0.001000                                0.001000
2       10000           0.001000                        0.001000                                0.000000
3       15000           0.003000                        0.001000                                0.000000
4       20000           0.004000                        0.002000                                0.000000
5       25000           0.004000                        0.002000                                0.000000
6       30000           0.005000                        0.004000                                0.002000
7       35000           0.007000                        0.003000                                0.003000
8       40000           0.007000                        0.004000                                0.004000
9       45000           0.009000                        0.004000                                0.016000
10      50000           0.009000                        0.005000                                0.000000

0. Quit
1. n Random numbers=>Array
2. Display the Array
3. Sort the Array in Ascending Order by using merge sort Algorithm
4. Sort the Array in Descending Order by using any sorting algorithm
5. Time Complexity to sort ascending of random data
6. Time Complexity to sort ascending of data already sorted in ascending order
7. Time Complexity to sort ascending of data already sorted in descending order
8. Time Complexity to sort ascending of data for all Cases (Data Ascending, Data in Descending & Random Data) in Tabular form for values n=5000 to 50000, step=5000
0
Exiting
```

The corresponding graphs are:


Time complexity(Random Data) vs. n
0.0100
0.0075
0.0050
0.0025
0.0000
Time complexity(Random Data)
10000
20000
30000
40000
50000
n


Time complexity(Data in Ascending) vs. n
0.005
0.004
0.003
0.002
0.001
0.000
Time complexity(Data in Ascending)
10000
20000
30000
40000
50000
n



Source code

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

void destroy_prev_allocation(int *a, int n) // Destroys previously (dynamically allocated) memory from the array
{
   int i;
   for(i=0; i<n; i++)
   {
      free(a + i);
   }
}

void insert_rand_array(int *a, int n) // Inserts n random values into the array
{
   int i;
   for(i=0; i<n; i++)
   {
      a[i] = rand();
   }
}
```

```
void display(int *a, int n) // Displays array elements
{
   int i;
   for(i=0; i<n; i++)
   {
      printf("%d\t",a[i]);
   }
   printf("\n");
}

void merge(int *a, int p, int q, int r)
{
   int i = p, j = q+1, next = 0;
   int sorted[r-p+1];
   while( (i<=q) && (j<=r) )
   {
      if(a[i] <= a[j])
      {
         sorted[next++] = a[i++];
      }

      else
      {
         sorted[next++] = a[j++];
      }
   }
   while(i<=q)
   {
      sorted[next++] = a[i++];
   }
   while(j<=r)
   {
      sorted[next++] = a[j++];
   }
   for(i=p; i<=r; i++) //time complexity of the step remains O(n)
   {
      a[i] = sorted[i-p];
   }
}

void merge_sort_ascending(int *a, int p, int r) // Sorts the array in ascending order using merge sort
{
   int q;
   if(p<r)
   {
      q = (p+r-1)/2;
      merge_sort_ascending(a, p, q);
      merge_sort_ascending(a, q+1, r);
```

```
        merge(a, p, q, r);
    }
}

void selection_sort_descending(int *a, int n) // Sorts the array in descending order using selection sort
{
    int i, j, max, temp;
    for(i=0; i<n; i++)
    {
        max = i;
        for(j=i+1; j<n; j++)
        {
            if(a[j] > a[max])
            {
                max = j;
            }
        }

        temp = a[i];
        a[i] = a[max];
        a[max] = temp;
    }
}

void row_display(int *a, int n)
{
    clock_t start, end;
    double time1, time2, time3;
    insert_rand_array(a, n);

    // Random
    start = clock();
    merge_sort_ascending(a, 0, n-1);
    end = clock();
    time1 = ((double) (end - start) )/CLOCKS_PER_SEC;

    // Ascending
    start = clock();
    merge_sort_ascending(a, 0, n-1); //Array is already sorted in ascending order
    end = clock();
    time2 = ((double) (end - start) )/CLOCKS_PER_SEC;

    // Descending
    selection_sort_descending(a, n);
    start = clock();
    merge_sort_ascending(a, 0, n-1);
    end = clock();
    time3 = ((double) (end - start) )/CLOCKS_PER_SEC;
```

```
    printf("%d\t\t%lf\t\t\t%lf\t\t\t\t%lf\n", n, time1, time2, time3);
}

int main()
{
    int choice=1, *a, n=0;
    clock_t start, end;
    double time;

    while(choice)
    {
        printf("\n0. Quit"
        "\n1. n Random numbers=>Array"
        "\n2. Display the Array"
        "\n3. Sort the Array in Ascending Order by using merge sort Algorithm"
        "\n4. Sort the Array in Descending Order by using any sorting algorithm"
        "\n5. Time Complexity to sort ascending of random data"
        "\n6. Time Complexity to sort ascending of data already sorted in ascending order"
        "\n7. Time Complexity to sort ascending of data already sorted in descending order"
        "\n8. Time Complexity to sort ascending of data for all Cases (Data Ascending, Data in Descending
& Random Data) in Tabular form for values n=5000 to 50000, step=5000\n"
        );

        scanf("%d",&choice);

        switch(choice)
        {
            case 0:
            printf("Exiting\n");
            break;

            case 1:
            destroy_prev_allocation(a, n);
            printf("Enter n\n");
            scanf("%d", &n);
            a = (int*) malloc( sizeof(int) * n );
            insert_rand_array(a, n);
            break;

            case 2:
            display(a, n);
            break;

            case 3:
            merge_sort_ascending(a, 0, n-1);
            break;
```

```
case 4:
selection_sort_descending(a, n);
break;

case 5:
start = clock();
merge_sort_ascending(a, 0, n-1);
end = clock();
time = ((double) (end - start) )/CLOCKS_PER_SEC;
printf("Time taken : %lf seconds\n", time);
break;

case 6:
merge_sort_ascending(a, 0, n-1);
start = clock();
merge_sort_ascending(a, 0, n-1); // The array is already sorted in ascending order
end = clock();
time = ((double) (end - start) )/CLOCKS_PER_SEC;
printf("Time taken : %lf seconds\n", time);
break;

case 7:
selection_sort_descending(a, n);
start = clock();
merge_sort_ascending(a, 0, n-1); // The array is already sorted in descending order
end = clock();
time = ((double) (end - start) )/CLOCKS_PER_SEC;
printf("Time taken : %lf seconds\n", time);
break;

case 8:
printf("SI No."
"\tValue of n"
"\tTime complexity(Random Data)"
"\tTime complexity(Data in Ascending)"
"\tTime complexity(Data in Descending)\n"
);
for(n=5000; n<=50000; n += 5000)
{
   a = (int*) malloc( sizeof(int) * n );
   printf("%d\t", (n/5000));
   row_display(a, n);
   destroy_prev_allocation(a, n);
}
break;

default:
printf("\t\tINVALID CHOICE\n");
```

```
        }
    }
    if(n>0)
    {
        destroy_prev_allocation(a, n);
    }
    return 0;
}
```

Conclusion/Observation

Thus, the best, worst and average case execution time is found for merge sort.

Program No: 2.3

Program Title:

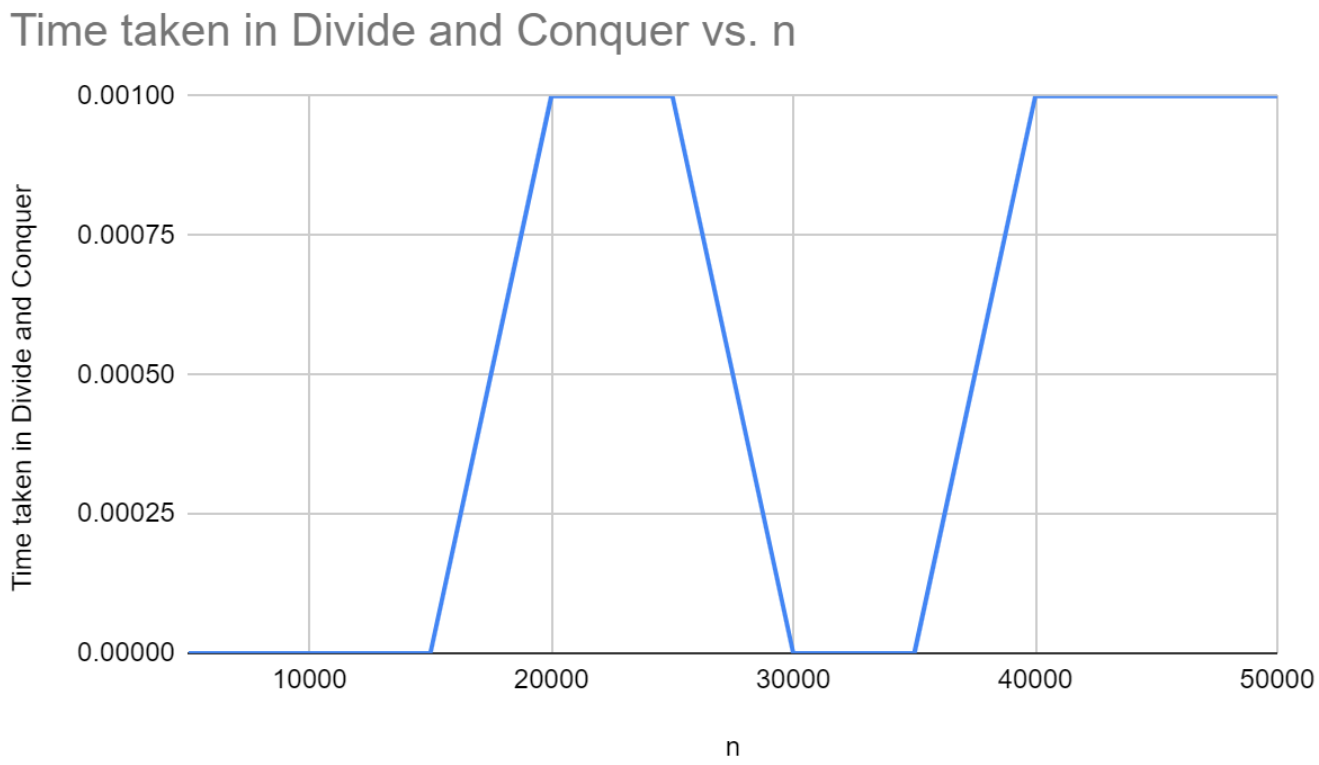
Write a program to use divide and conquer method to determine the time required to find the maximum and minimum element in a list of n elements. The data for the list can be generated randomly. Compare this time with the time taken by straight forward algorithm or brute force algorithm for finding the maximum and minimum element for the same list of n elements. Show the comparison by plotting a required graph for this problem.

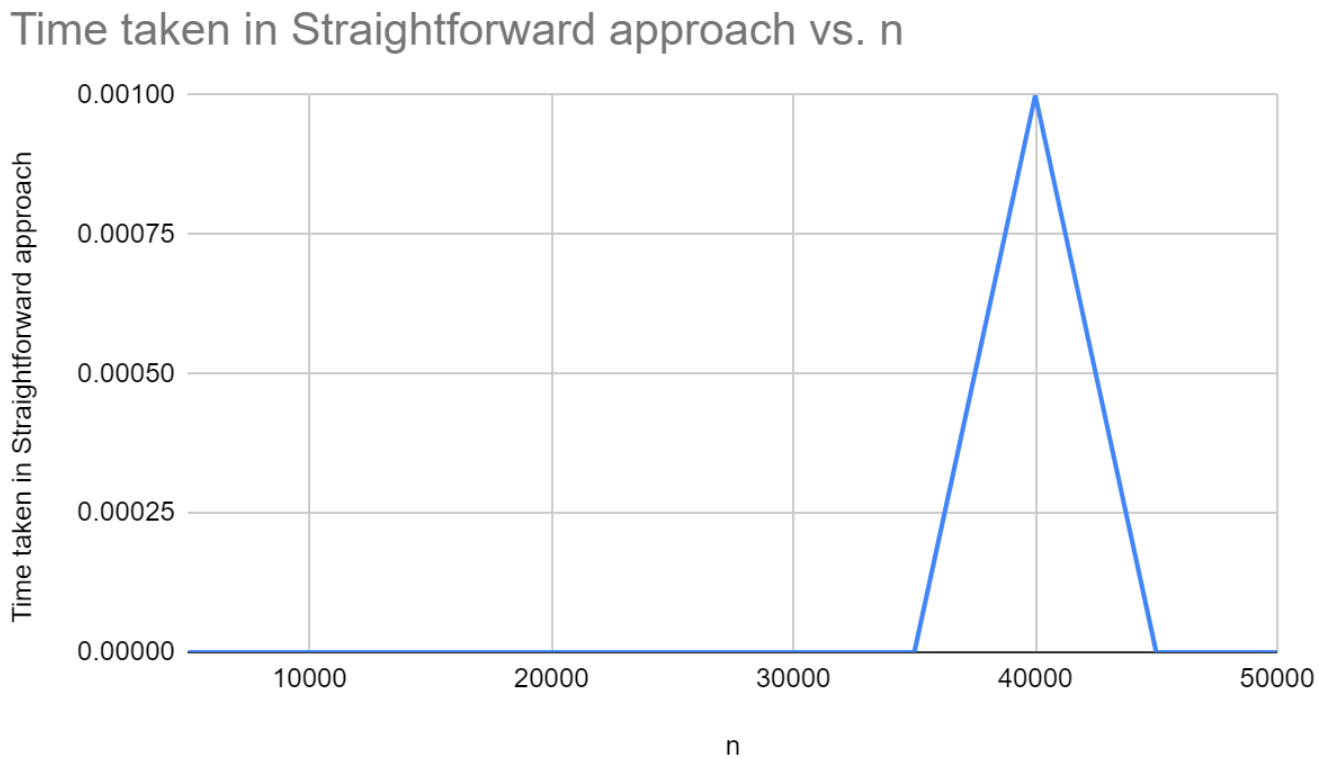
Input/Output Screenshots:

RUN-1:

```
PS C:\Algo-Lab\Lab-4> gcc .\q3.c
PS C:\Algo-Lab\Lab-4> ./a
Enter 0 to EXIT, 1 for manual testing, or anything else for auto-testing
9
Array Size(n)   Time taken in D&C       Time taken in SF
5000            0.000000                0.000000
10000           0.000000                0.000000
15000           0.000000                0.000000
20000           0.001000                0.000000
25000           0.001000                0.000000
30000           0.000000                0.000000
35000           0.000000                0.000000
40000           0.001000                0.001000
45000           0.001000                0.000000
50000           0.001000                0.000000
```

The corresponding graphs are:





Source code

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

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

void destroy_prev_allocation(int *a, int n) // Destroys previously (dynamically allocated) memory from the array
{
   int i;
   for(i=0; i<n; i++)
   {
      free(a + i);
   }
}

void insert_rand_array(int *a, int n) // Inserts n random values into the array
{
   int i;
   for(i=0; i<n; i++)
   {
      a[i] = rand();
   }
}

void find_max_min_dc(int *a, int l, int h, int *max, int *min) //Find maximum & minimum elements by divide and conquer
{
   int lmax, hmax, lmin, hmin, m = (l+h)/2;

   if(h-l < 0) //If array is empty, we return maximum and minimum values as 0
   {
      *max = *min = 0;
   }

   else if(h-l <= 1) //0 means we have 1 element, 1 means we have 2 elements
   {
      if(a[l] >= a[h])
      {
         *max = a[l];
         *min = a[h];
      }
      else
      {
         *max = a[h];
         *min = a[l];
      }
   }

   else
```

```
    {
        find_max_min_dc(a, l, m, &lmax, &lmin);
        find_max_min_dc(a, m+1, h, &hmax, &hmin);

        if(lmax >= hmax)
        {
            *max = lmax;
        }
        else
        {
            *max = hmax;
        }

        if(lmin <= hmin)
        {
            *min = lmin;
        }
        else
        {
            *min = hmin;
        }
    }
}

void find_max_min_sf(int *a, int n, int *max, int *min) //Find maximum and minimum elements by
straightforward method
{
    int i;
    if(n > 0)
    {
        *max = a[0];
        *min = a[0];
        for(i=1; i<n; i++)
        {
            if(a[i] > *max)
            {
                *max = a[i];
            }
            if(a[i] < *min)
            {
                *min = a[i];
            }
        }
    }

    else
    {
        *max = *min = 0; //If array has no elements, we take maximum and minimum values to be 0
```

```
    }
}

int main()
{
    int *a, n, i, choice, max, min;
    clock_t start, end;
    double time;

    printf("Enter 0 to EXIT, 1 for manual testing, or anything else for auto-testing\n");
    scanf("%d", &choice);

    //EXIT
    if(choice == 0)
    {
        printf("EXITING\n");
        return 0;
    }

    //Manual Testing
    else if(choice == 1)
    {
        printf("Enter size of array\n");
        scanf("%d", &n);
        a = (int*) malloc(sizeof(int) * n);
        insert_rand_array(a, n);


        printf("Display array?(1=YES, 0=NO)\n"); //So that we can avoid this when using larger values of n
        scanf("%d", &choice);
        if(choice)
        {
            printf("Array is:\n");
            for(i=0; i<n; i++)
            {
                printf("%d\t", a[i]);
            }
            printf("\n");
        }

        //Divide and Conquer
        start = clock();
        find_max_min_dc(a, 0, n-1, &max, &min);
        end = clock();
        time = ((double) (end - start) )/CLOCKS_PER_SEC;
        printf("Maximum element by divide and conquer method = %d\n", max);
        printf("Minimum element by divide and conquer method = %d\n", min);
        printf("Time taken by divide and conquer method = %lf seconds\n\n", time);
```

```
        //Straightforward
        start = clock();
        find_max_min_sf(a, n, &max, &min);
        end = clock();
        time = ((double) (end - start) )/CLOCKS_PER_SEC;
        printf("Maximum element by straightforward method = %d\n", max);
        printf("Minimum element by straightforward method = %d\n", min);
        printf("Time taken by straightforward method = %lf seconds\n\n", time);

        destroy_prev_allocation(a, n);

        return 0;
    }

    //Auto-testing
    printf("Array Size(n)\tTime taken in D&C\tTime taken in SF\n");
    for(n=5000; n<=50000; n+=5000)
    {
        a = (int*) malloc(sizeof(int) * n);
        insert_rand_array(a, n);

        //Divide and Conquer
        start = clock();
        find_max_min_dc(a, 0, n-1, &max, &min);
        end = clock();
        time = ((double) (end - start) )/CLOCKS_PER_SEC;
        printf("%d\t\t%lf\t\t", n, time);

        //Straightforward
        start = clock();
        find_max_min_sf(a, n, &max, &min);
        end = clock();
        time = ((double) (end - start) )/CLOCKS_PER_SEC;
        printf("%lf\n", time);

        destroy_prev_allocation(a, n);
    }

    return 0;
}
```

Conclusion/Observation

Thus, it is observed that the straightforward approach generally takes less time as compared to divide and conquer technique in this particular case due to lower number of comparisons performed because maximum and minimum can be found simultaneously.

Program No: 2.4

Program Title:

Write a program that uses a divide-and-conquer algorithm/user defined function for the exponentiation problem of computing a_n where a > 0 and n is a positive integer. How does this algorithm compare with the brute-force algorithm in terms of number of multiplications made by both algorithms.

Input/Output Screenshots:

RUN-1:

```
PS C:\Algo-Lab\Lab-4> gcc .\q4.c
PS C:\Algo-Lab\Lab-4> ./a
Enter a<space>n
2 10
a^n = 1024
```

Source code

```
//Number of multiplications made in divide and conquer algorithm will be less than that in brute force algorithm
//This is because in D&C, we will multiply powers of a to get to the next power (approx. log n multiplications)
//But in brute force method, we multiply with 'a' every time (n-1 multiplications)

#include<stdio.h>

int expo_dc(int a, int n) //Calculates a^n by divide and conquer method
{
   int half_pow = n/2;

   if(n==1) //As n is a positive integer, its minimum value is 1. It will not take lower value during recursion either
   {
      return a;
   }
   else
   {
      return (expo_dc(a, half_pow) * expo_dc(a, (n - half_pow)));
   }
}

// int expo_bf(int a, int n) //Calculates a^n by brute force method (written for reference)
// {
//    int i, pow = a; //Minimum value of n=1, so minimum value of pow=a
//    for(i=1; i<n; i++)
//    {
//       pow *= a;
//    }
```

```
//     return pow;
// }

int main()
{
   int a, n;
   printf("Enter a<space>n\n");
   scanf("%d%d", &a, &n);
   printf("a^n = %d\n", expo_dc(a, n));

   return 0;
}
```

Conclusion/Observation

Thus, we observe that the divide and conquer approach takes less time as compared to the brute force approach in this case due to the lower number of multiplications that need to be performed.