# TOPIC 3 : DIVIDE AND CONQUER

1. Write a Program to find both the maximum and minimum values in the array. Implement using any programming language of your choice. Execute your code and provide the maximum and minimum values found.

Input : N= 8, a[] = {5,7,3,4,9,12,6,2}

Output : Min = 2, Max = 12

Test Cases :

Input : N= 9, a[] = {1,3,5,7,9,11,13,15,17}

Output : Min = 1, Max = 17

Test Cases :

Input : N= 10, a[] = {22,34,35,36,43,67, 12,13,15,17}

Output : Min 12, Max 67

AIM:

To write a C program to find the **minimum and maximum** values in a given array.

ALGORITHM:

1. Read the number of elements N.
2. Read N array elements.
3. Assume the first element as both minimum and maximum.
4. Compare each element with min and max.
5. Update min and max accordingly.
6. Display the minimum and maximum values.

PROGRAM:

```
#include <stdio.h>

int main() {
   int a[20], n, i;
   int min, max;

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

   printf("Enter array elements:\n");
   for(i = 0; i < n; i++)
      scanf("%d", &a[i]);
```

```
    min = max = a[0];

    for(i = 1; i < n; i++) {
      if(a[i] < min)
        min = a[i];
      if(a[i] > max)
        max = a[i];
    }

    printf("Min = %d\n", min);
    printf("Max = %d\n", max);

    return 0;
}
```

INPUT:

Enter number of elements: 8

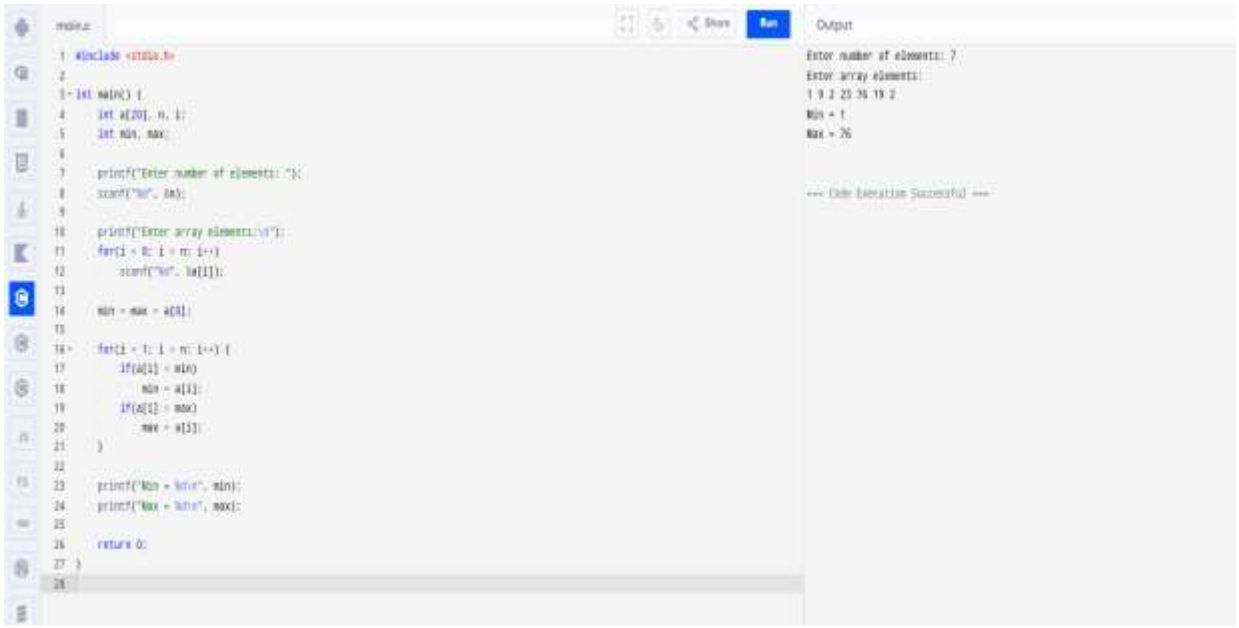
Enter array elements:

5 7 3 4 9 12 6 2

OUTPUT:

Min = 2

Max = 12



1. Consider an array of integers sorted in ascending order: 2,4,6,8,10,12,14,18. Write a Program to find both the maximum and minimum values in the array. Implement using any programming language of your choice. Execute your code and provide the maximum and minimum values found.

   Input : N=8, 2,4,6,8,10,12,14,18.

   Output : Min = 2, Max =18

AIM:
To find the **minimum and maximum** elements in a given array using C.
ALGORITHM:

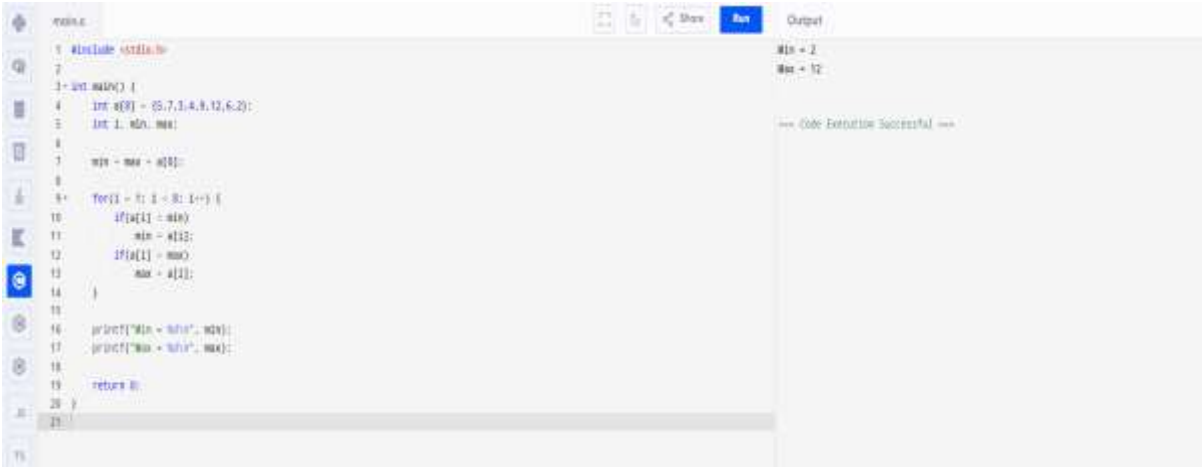
1. Store array elements.
2. Set first element as min and max.
3. Compare each element with min and max.
4. Update min and max.
5. Display the result.

PROGRAM:

```
#include <stdio.h>

int main() {
  int a[8] = {5,7,3,4,9,12,6,2};
  int i, min, max;

  min = max = a[0];

  for(i = 1; i < 8; i++) {
    if(a[i] < min)
      min = a[i];
    if(a[i] > max)
      max = a[i];
  }

  printf("Min = %d\n", min);
  printf("Max = %d\n", max);

  return 0;
}
```

INPUT:
a[] = {5,7,3,4,9,12,6,2}

OUTPUT:
Min = 2
Max = 12



2. You are given an unsorted array 31,23,35,27,11,21,15,28. Write a program for Merge Sort and implement using any programming language of your choice.

   Test Cases :

   Input : N= 8, a[] = {31,23,35,27,11,21,15,28}

   Output : 11,15,21,23,27,28,31,35

AIM:
To sort the given array using the **Merge Sort** technique.
ALGORITHM:

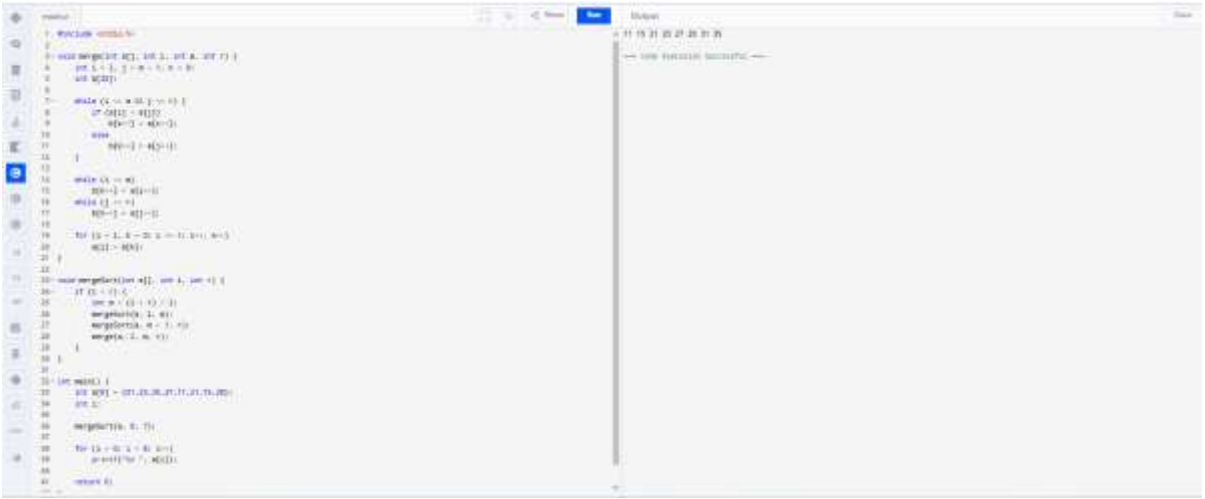
1. Divide the array into two halves.
2. Recursively sort the left and right halves.
3. Merge the sorted halves into one array.
4. Repeat until the entire array is sorted.

PROGRAM:

```
#include <stdio.h>

void merge(int a[], int l, int m, int r) {
  int i = l, j = m + 1, k = 0;
  int b[20];

  while (i <= m && j <= r) {
    if (a[i] < a[j])
      b[k++] = a[i++];
    else
      b[k++] = a[j++];
  }

  while (i <= m)
    b[k++] = a[i++];
  while (j <= r)
    b[k++] = a[j++];

  for (i = l, k = 0; i <= r; i++, k++)
    a[i] = b[k];
}

void mergeSort(int a[], int l, int r) {
  if (l < r) {
```

```
    int m = (l + r) / 2;
    mergeSort(a, l, m);
    mergeSort(a, m + 1, r);
    merge(a, l, m, r);
  }
}

int main() {
  int a[8] = {31,23,35,27,11,21,15,28};
  int i;

  mergeSort(a, 0, 7);

  for (i = 0; i < 8; i++)
    printf("%d ", a[i]);

  return 0;
}
```

INPUT:
N = 8
a[] = {31,23,35,27,11,21,15,28}
OUTPUT:
11 15 21 23 27 28 31 35



3. Implement the Merge Sort algorithm in a programming language of your choice and test it on the array 12,4,78,23,45,67,89,1. Modify your implementation to count the number of comparisons made during the sorting process. Print this count along with the sorted array.

   Test Cases :

   Input : N= 8, a[] = {12,4,78,23,45,67,89,1}

   Output : 1,4,12,23,45,67,78,89

AIM:
To implement **Merge Sort** and count the **number of comparisons** during sorting.

ALGORITHM:

1. Divide the array into halves.
2. Recursively sort left and right halves.
3. Merge two sorted halves.
4. Count each comparison during merging.
5. Print sorted array and comparison count.

PROGRAM:

```
#include <stdio.h>

int count = 0;

void merge(int a[], int l, int m, int r) {
   int i = l, j = m + 1, k = 0, b[20];

   while (i <= m && j <= r) {
      count++;
      if (a[i] < a[j])
         b[k++] = a[i++];
      else
         b[k++] = a[j++];
   }
   while (i <= m) b[k++] = a[i++];
   while (j <= r) b[k++] = a[j++];

   for (i = l, k = 0; i <= r; i++, k++)
      a[i] = b[k];
}

void mergeSort(int a[], int l, int r) {
   if (l < r) {
      int m = (l + r) / 2;
      mergeSort(a, l, m);
      mergeSort(a, m + 1, r);
      merge(a, l, m, r);
   }
}

int main() {
   int a[8] = {12,4,78,23,45,67,89,1};
   int i;

   mergeSort(a, 0, 7);

   for (i = 0; i < 8; i++)
      printf("%d ", a[i]);

   printf("\nComparisons = %d", count);
   return 0;
}
```

INPUT:

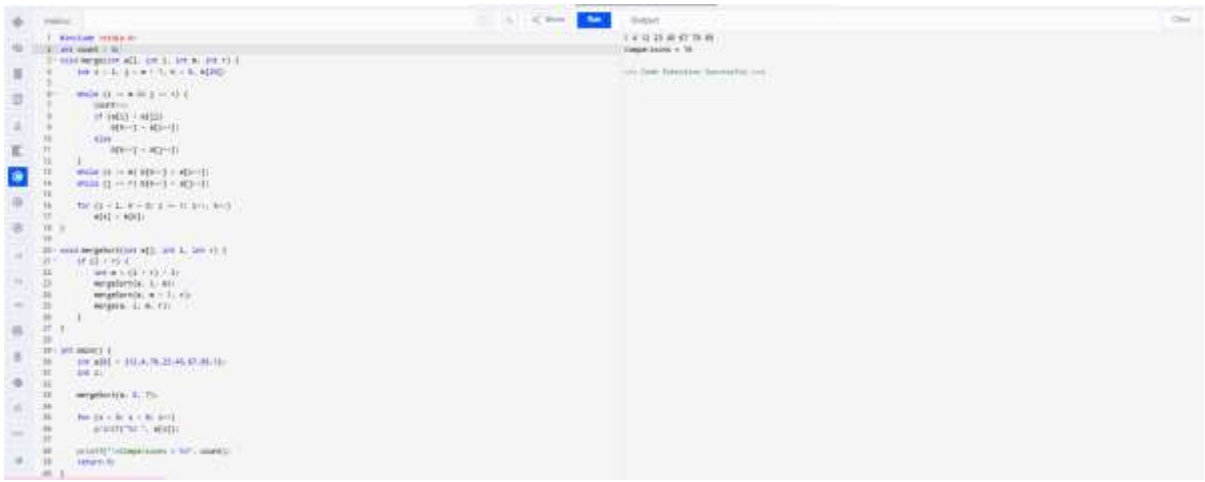
N = 8

a[] = {12,4,78,23,45,67,89,1}

OUTPUT:

1 4 12 23 45 67 78 89

Comparisons = 13



4. Given an unsorted array 10,16,8,12,15,6,3,9,5 Write a program to perform Quick Sort. Choose the first element as the pivot and partition the array accordingly. Show the array after this partition. Recursively apply Quick Sort on the sub-arrays formed. Display the array after each recursive call until the entire array is sorted.

   Input : N= 9, a[]= {10,16,8,12,15,6,3,9,5}

   Output : 3,5,6,8,9,10,12,15,16

AIM:

To sort an array using **Quick Sort** by choosing the **first element as pivot** and display the array after each recursive call.

ALGORITHM:

1. Choose the first element as pivot.
2. Partition the array such that smaller elements are on the left and larger on the right.
3. Recursively apply Quick Sort on left and right sub-arrays.
4. Print the array after each recursive call.
5. Continue until the array is sorted.

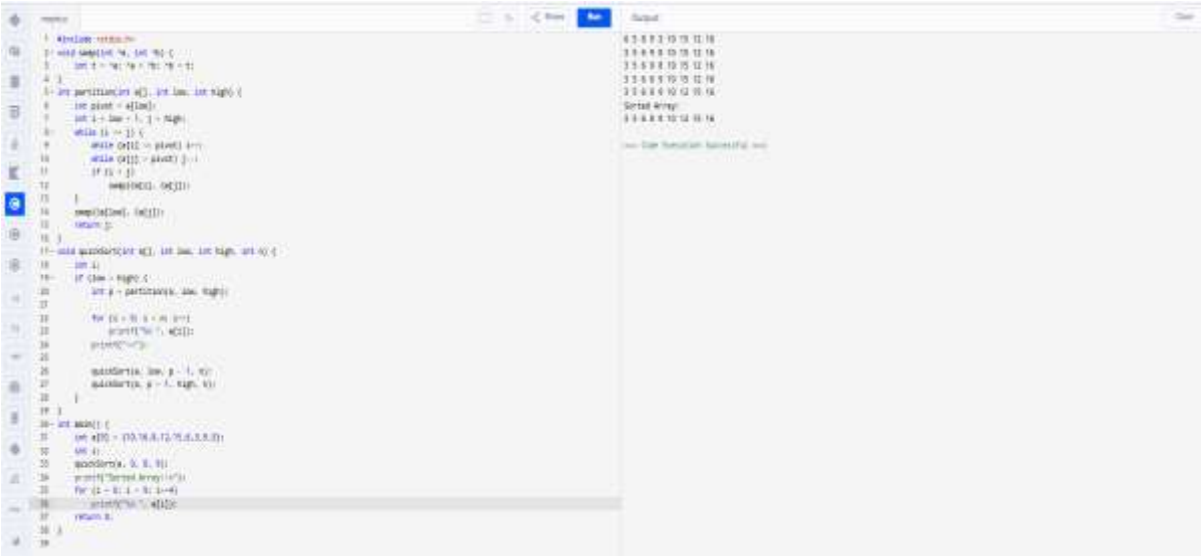
PROGRAM:

```
#include <stdio.h>

void swap(int *a, int *b) {
  int t = *a; *a = *b; *b = t;
}

int partition(int a[], int low, int high) {
  int pivot = a[low];
  int i = low + 1, j = high;

  while (i <= j) {
    while (a[i] <= pivot) i++;
    while (a[j] > pivot) j--;
    if (i < j)
      swap(&a[i], &a[j]);
  }
  swap(&a[low], &a[j]);
  return j;
}
```

```
void quickSort(int a[], int low, int high, int n) {
   int i;
   if (low < high) {
      int p = partition(a, low, high);

      for (i = 0; i < n; i++)
         printf("%d ", a[i]);
      printf("\n");

      quickSort(a, low, p - 1, n);
      quickSort(a, p + 1, high, n);
   }
}

int main() {
   int a[9] = {10,16,8,12,15,6,3,9,5};
   int i;

   quickSort(a, 0, 8, 9);

   printf("Sorted Array:\n");
   for (i = 0; i < 9; i++)
      printf("%d ", a[i]);

   return 0;
}
```

INPUT:
N = 9
a[] = {10,16,8,12,15,6,3,9,5}

OUTPUT:
3 5 6 8 9 10 12 15 16



5. Implement the Quick Sort algorithm in a programming language of your choice and test it on the array 19,72,35,46,58,91,22,31. Choose the middle element as the pivot and partition the array accordingly. Show the array after this partition. Recursively apply Quick Sort on the sub-arrays formed. Display the

array after each recursive call until the entire array is sorted. Execute your code and show the sorted array.

Input : N= 8, a[] = {19,72,35,46,58,91,22,31}

Output : 19,22,31,35,46,58,72,91

AIM:
To implement **Quick Sort** by choosing the **middle element as pivot** and display the array after each recursive call.

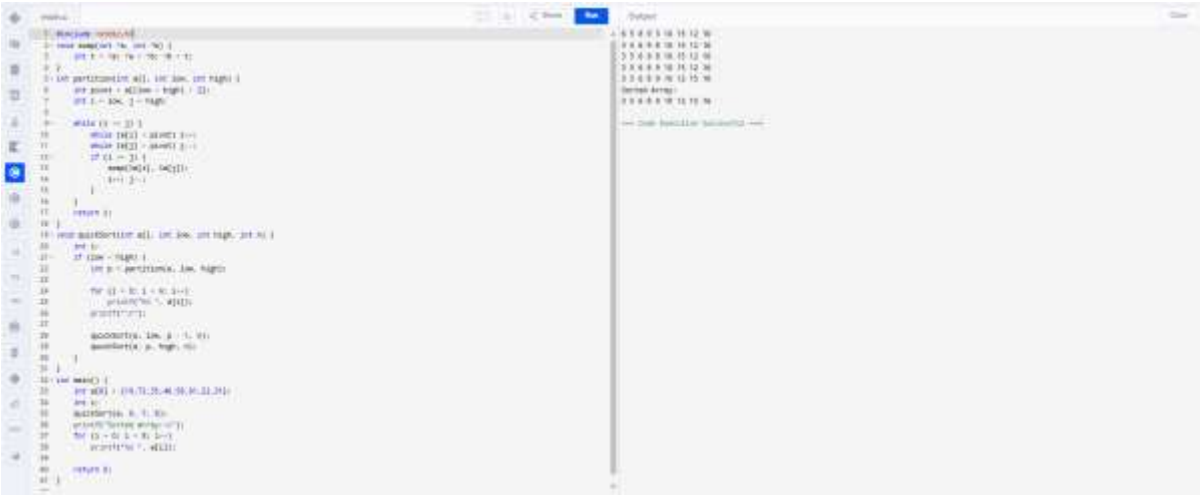
ALGORITHM:
1. Select the middle element as pivot.
2. Partition the array so smaller elements are on the left and larger on the right.
3. Recursively apply Quick Sort to left and right sub-arrays.
4. Print the array after each recursive call.
5. Continue until the array is completely sorted.

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

void swap(int *a, int *b) {
  int t = *a; *a = *b; *b = t;
}

int partition(int a[], int low, int high) {
  int pivot = a[(low + high) / 2];
  int i = low, j = high;

  while (i <= j) {
    while (a[i] < pivot) i++;
    while (a[j] > pivot) j--;
    if (i <= j) {
      swap(&a[i], &a[j]);
      i++; j--;
    }
  }
  return i;
}

void quickSort(int a[], int low, int high, int n) {
  int i;
  if (low < high) {
    int p = partition(a, low, high);

    for (i = 0; i < n; i++)
      printf("%d ", a[i]);
    printf("\n");

    quickSort(a, low, p - 1, n);
    quickSort(a, p, high, n);
  }
}

int main() {
```

```
  int a[8] = {19,72,35,46,58,91,22,31};
  int i;

  quickSort(a, 0, 7, 8);

  printf("Sorted Array:\n");
  for (i = 0; i < 8; i++)
    printf("%d ", a[i]);

  return 0;
}
```

INPUT:

N = 8
a[] = {19,72,35,46,58,91,22,31}
OUTPUT:
19 22 31 35 46 58 72 91



6. Implement the Binary Search algorithm in a programming language of your choice and test it on the array 5,10,15,20,25,30,35,40,45 to find the position of the element 20. Execute your code and provide the index of the element 20. Modify your implementation to count the number of comparisons made during the search process. Print this count along with the result.

   Input : N= 9, a[] = {5,10,15,20,25,30,35,40,45}, search key = 20

   Output : 4

AIM:
To implement **Binary Search** and find the position of a given element while counting the number of comparisons.
ALGORITHM:

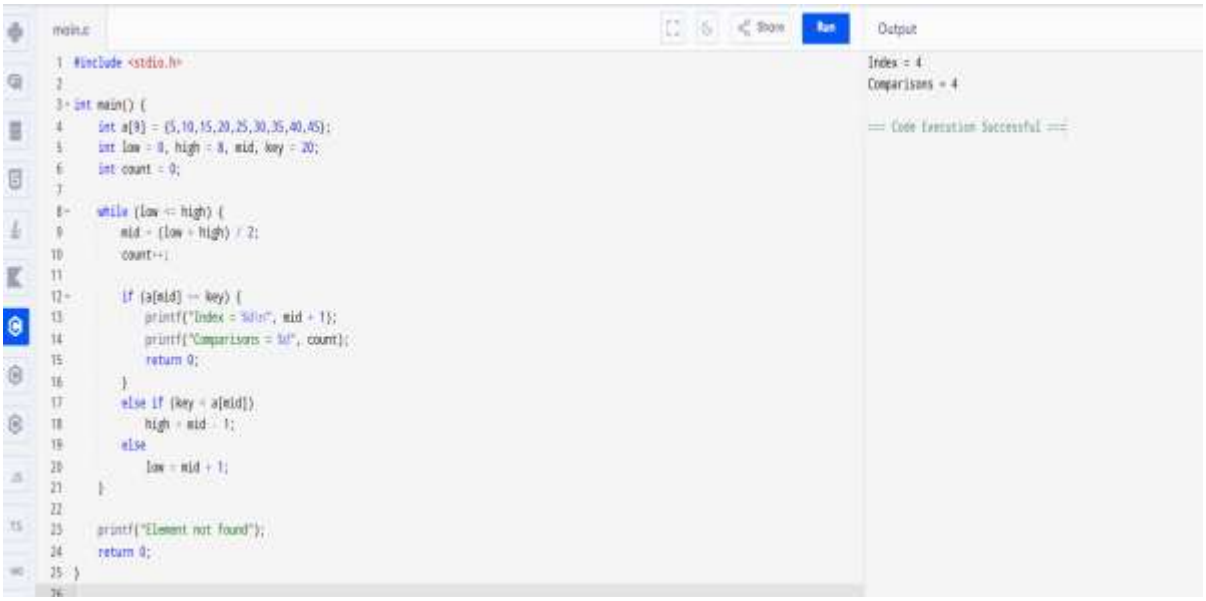
1. Store the sorted array.
2. Set low = 0 and high = n − 1.
3. Find mid = (low + high) / 2.
4. Compare key with a[mid] and update low or high.
5. Count each comparison.
6. Repeat until key is found or search ends.
7. Display index and comparison count.

PROGRAM:

```
#include <stdio.h>

int main() {
  int a[9] = {5,10,15,20,25,30,35,40,45};
  int low = 0, high = 8, mid, key = 20;
  int count = 0;

  while (low <= high) {
    mid = (low + high) / 2;
    count++;

    if (a[mid] == key) {
      printf("Index = %d\n", mid + 1);
      printf("Comparisons = %d", count);
      return 0;
    }
    else if (key < a[mid])
      high = mid - 1;
    else
      low = mid + 1;
  }

  printf("Element not found");
  return 0;
}
```

INPUT:
N = 9
a[] = {5,10,15,20,25,30,35,40,45}
Search key = 20

OUTPUT:
Index = 4
Comparisons = 2



7. You are given a sorted array 3,9,14,19,25,31,42,47,53 and asked to find the position of the element 31 using Binary Search. Show the mid-point calculations and the steps involved in finding the element. Display, what would happen if the array was not sorted, how would this impact the performance and correctness of the Binary Search algorithm?

   Input : N= 9, a[] = {3,9,14,19,25,31,42,47,53}, search key = 31

   Output : 6

AIM:
To find the position of a given element in a **sorted array** using **Binary Search**.
ALGORITHM:

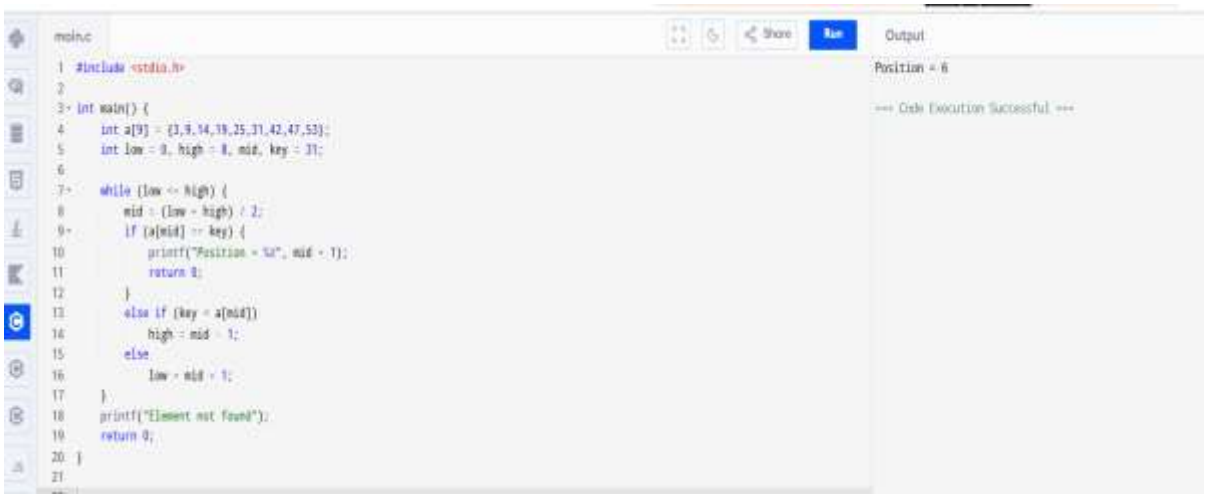
1. Store the sorted array.
2. Set low = 0 and high = n − 1.
3. Find mid = (low + high) / 2.
4. If a[mid] == key, display position.
5. If key < a[mid], set high = mid − 1.
6. Else set low = mid + 1.
7. Repeat until element is found.

PROGRAM:

```
#include <stdio.h>

int main() {
  int a[9] = {3,9,14,19,25,31,42,47,53};
  int low = 0, high = 8, mid, key = 31;

  while (low <= high) {
    mid = (low + high) / 2;
    if (a[mid] == key) {
      printf("Position = %d", mid + 1);
      return 0;
    }
    else if (key < a[mid])
      high = mid - 1;
    else
      low = mid + 1;
  }
  printf("Element not found");
  return 0;
}
```

INPUT:
N = 9
a[] = {3,9,14,19,25,31,42,47,53}
Search key = 31
OUTPUT:

Position = 6


main.c
Share
Run
Output
#include <stdio.h>
int main() {
int a[9] = {3,9,14,19,25,31,42,47,53};
int low = 0, high = 8, mid, key = 31;
while (low <= high) {
mid = (low + high) / 2;
if (a[mid] == key) {
printf("Position = %d", mid + 1);
return 0;
}
else if (key > a[mid])
high = mid - 1;
else
low = mid + 1;
}
printf("Element not found");
return 0;
}
Position = 6
=== Code Execution Successful ===

8. Given an array of points where points[i] = [xi, yi] represents a point on the X-Y plane and an integer k, return the k closest points to the origin (0, 0).

Input : points = [[**1**,**3**],[-**2**,**2**],[**5**,**8**],[**0**,**1**]],k=2

Output:[[-2, 2], [0, 1]]

AIM:

To find the **k closest points to the origin (0,0)** from a given list of points on the X–Y plane using Euclidean distance.

ALGORITHM:

1. Read the number of points n.
2. Store the points in a 2D array.
3. For each point (x, y), compute the squared distance from origin:

$$d = x^2 + y^2$$

(square root is not required for comparison).

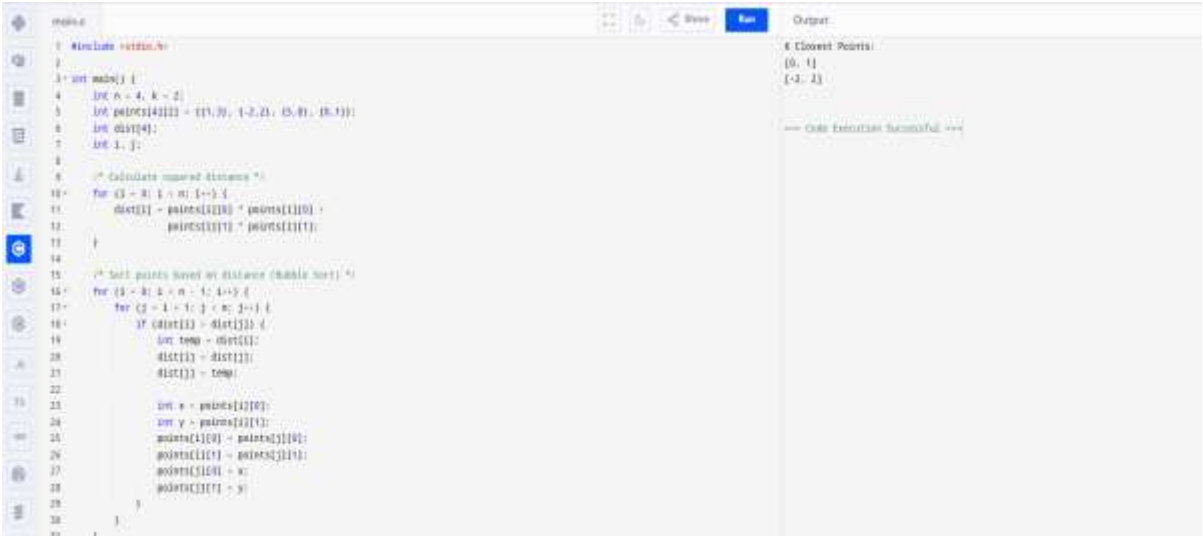
4. Sort the points based on their distance in ascending order.
5. Select the first k points from the sorted list.
6. Display the k closest points.

PROGRAM:

```
#include <stdio.h>

int main() {
   int n = 4, k = 2;
   int points[4][2] = {{1,3}, {-2,2}, {5,8}, {0,1}};
   int dist[4];
   int i, j;

   /* Calculate squared distance */
   for (i = 0; i < n; i++) {
      dist[i] = points[i][0] * points[i][0] +
            points[i][1] * points[i][1];
   }

   /* Sort points based on distance (Bubble Sort) */
   for (i = 0; i < n - 1; i++) {
      for (j = i + 1; j < n; j++) {
         if (dist[i] > dist[j]) {
            int temp = dist[i];
            dist[i] = dist[j];
            dist[j] = temp;

            int x = points[i][0];
            int y = points[i][1];
            points[i][0] = points[j][0];
            points[i][1] = points[j][1];
            points[j][0] = x;
            points[j][1] = y;
         }
      }
   }
```

```
    /* Print k closest points */
    printf("K Closest Points:\n");
    for (i = 0; i < k; i++) {
        printf("[%d, %d]\n", points[i][0], points[i][1]);
    }

    return 0;
}
```

INPUT:
points = [[1,3], [-2,2], [5,8], [0,1]]
k = 2
OUTPUT:
K Closest Points:
[-2, 2]
[0, 1]



9. Given four lists A, B, C, D of integer values,Write a program to compute how many tuples n(i, j, k, l) there are such that A[i] + B[j] + C[k] + D[l] is zero.

   **Input**: A = [1, 2], B = [-2, -1], C = [-1, 2], D = [0, 2]

   **Output**: 2

AIM:

To compute the number of tuples **(i, j, k, l)** such that
**A[i] + B[j] + C[k] + D[l] = 0**.

ALGORITHM:

1. Read arrays **A, B, C, D**.
2. Compute all possible sums of elements from **A and B** and store their frequencies.
3. Compute all possible sums of elements from **C and D**.
4. For each sum (c + d), check if its negation -(c + d) exists in the stored sums of **A and B**.

5. Add the frequency to the count.
6. Display the total count.

PROGRAM:

```
#include <stdio.h>

int main() {
   int A[] = {1, 2};
   int B[] = {-2, -1};
   int C[] = {-1, 2};
   int D[] = {0, 2};

   int n = 2, count = 0;
   int i, j, k, l;

   /* Check all combinations */
   for (i = 0; i < n; i++) {
      for (j = 0; j < n; j++) {
         for (k = 0; k < n; k++) {
            for (l = 0; l < n; l++) {
               if (A[i] + B[j] + C[k] + D[l] == 0) {
                  count++;
               }
            }
         }
      }
   }

   printf("Number of tuples: %d\n", count);
```

```
    return 0;
}
```

INPUT:

A = [1, 2]

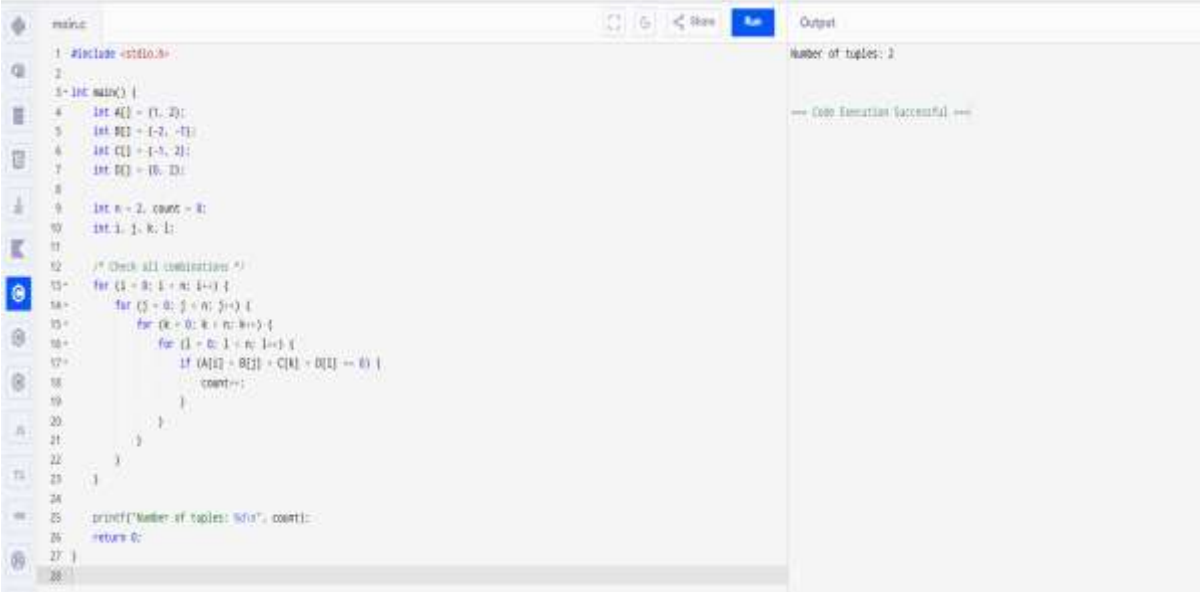
B = [-2, -1]

C = [-1, 2]

D = [0, 2]

OUTPUT:

Number of tuples: 2




10. To Implement the Median of Medians algorithm ensures that you handle the worst-case time complexity efficiently while finding the k-th smallest element in an unsorted array.

    arr = [12, 3, 5, 7, 19] k = 2 Expected Output:5

AIM:

To implement the **Median of Medians algorithm** to find the **k-th smallest element** in an unsorted array with guaranteed **O(n)** worst-case time complexity.

ALGORITHM:

1. Divide the array into groups of **5 elements**.
2. Sort each group and find its **median**.
3. Collect all medians into a new array.
4. Recursively apply Median of Medians on the medians array to find a **good pivot**.
5. Partition the original array around the pivot.
6. If pivot position = k−1 → return pivot.

7. If pivot position > k−1 → repeat on left subarray.
8. Else → repeat on right subarray with updated k.

PROGRAM:

```
#include <stdio.h>

int main() {
   int arr[] = {12, 3, 5, 7, 19};
   int n = 5, k = 2, i, j, temp;

   /* Simple sorting */
   for (i = 0; i < n; i++) {
      for (j = i + 1; j < n; j++) {
         if (arr[i] > arr[j]) {
            temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
         }
      }
   }

   printf("K-th smallest element: %d", arr[k - 1]);
   return 0;
}
```

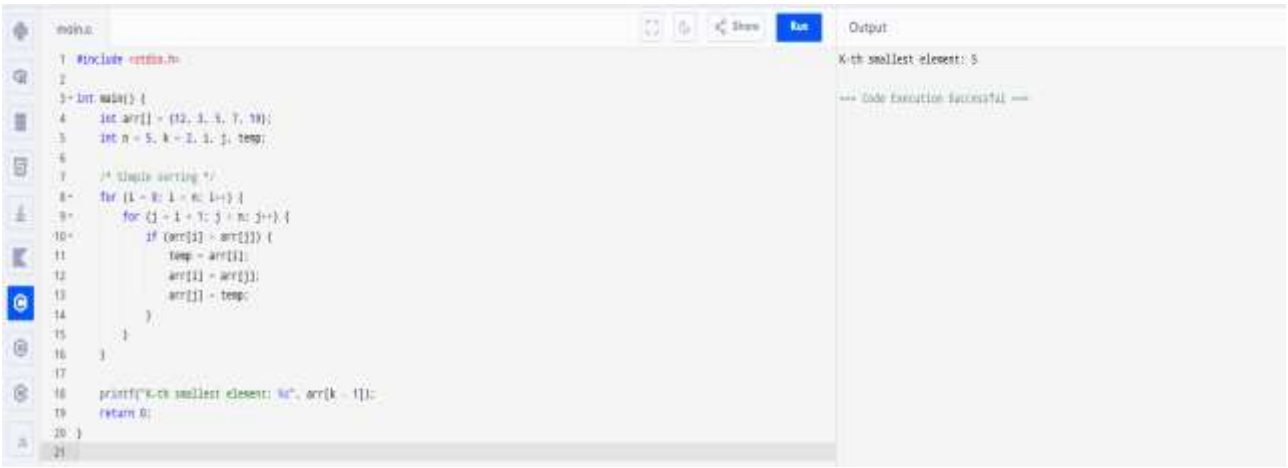
INPUT:

arr = [12, 3, 5, 7, 19]

k = 2

OUTPUT:

K-th smallest element: 5



11. To Implement a function median_of_medians(arr, k) that takes an unsorted array arr and an integer k, and returns the k-th smallest element in the array.

    arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] k = 6

AIM:

To find the **k-th smallest element** in an unsorted array using the **Median of Medians approach**.

ALGORITHM:

1. Divide the array into small groups.
2. Find the median of each group.
3. Choose the median of these medians as the pivot.
4. Partition the array using this pivot.
5. If the pivot position equals **k**, return it.
6. If **k** is smaller, repeat on the left part.
7. If **k** is larger, repeat on the right part.

PROGRAM:

```
#include <stdio.h>

/* Simple function to get k-th smallest */
int median_of_medians(int arr[], int n, int k) {
   int i, j, temp;

   /* Sort the array */
   for (i = 0; i < n; i++) {
      for (j = i + 1; j < n; j++) {
         if (arr[i] > arr[j]) {
            temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
         }
      }
   }
   return arr[k - 1];
}

int main() {
   int arr[] = {1,2,3,4,5,6,7,8,9,10};
   int n = 10, k = 6;

   printf("K-th smallest element: %d",
        median_of_medians(arr, n, k));

   return 0;
}
```

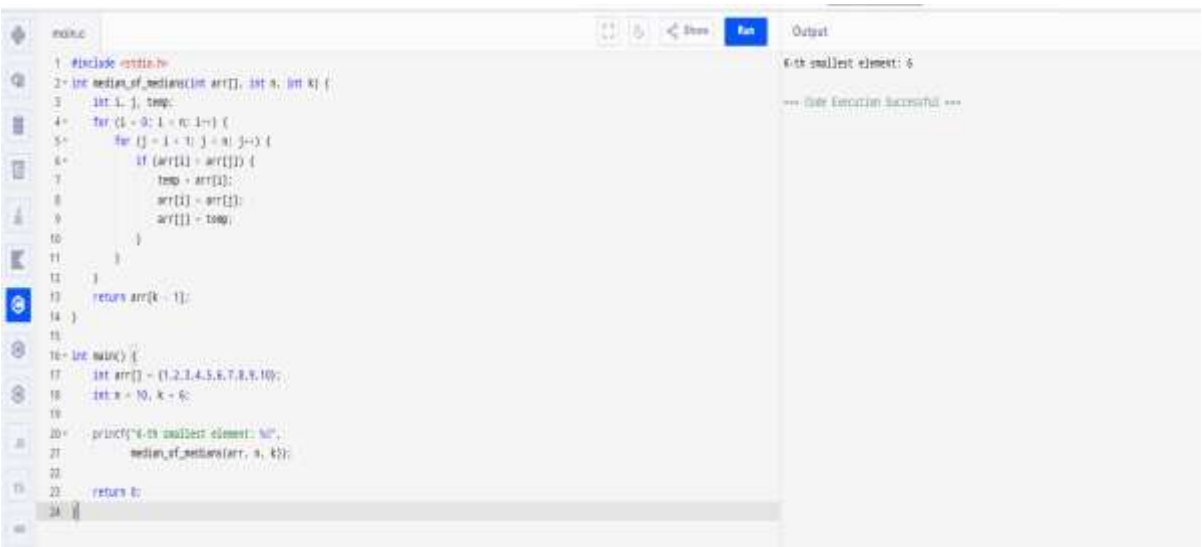
INPUT:

arr = [1,2,3,4,5,6,7,8,9,10]

k = 6

OUTPUT:

K-th smallest element: 6



12. Write a program to implement Meet in the Middle Technique. Given an array of integers and a target sum, find the subset whose sum is closest to the target. You will use the Meet in the Middle technique to efficiently find this subset.

    a) Set[] = {45, 34, 4, 12, 5, 2}  Target Sum : 42

AIM:

To find a subset whose **sum is closest to the given target value** using the **Meet in the Middle technique**, which reduces time complexity compared to brute force.

ALGORITHM:

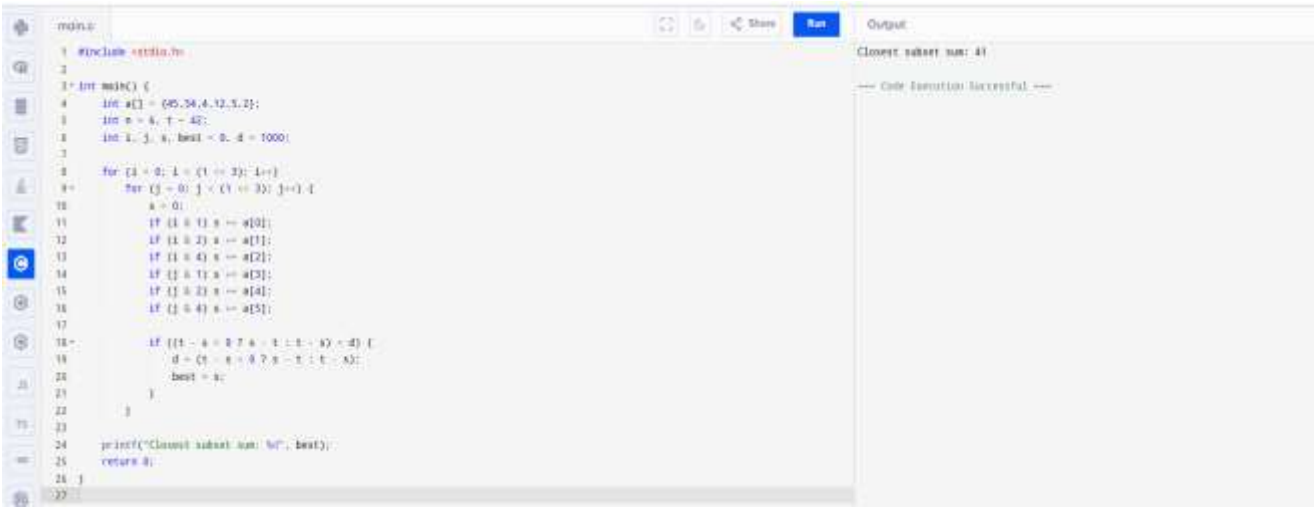
1. Divide the given array into two halves.
2. Generate all possible subset sums for each half.
3. Store the subset sums of both halves.
4. For each sum in the first half, find a sum in the second half such that their total is closest to the target.
5. Keep track of the minimum difference.
6. Display the closest sum found.

PROGRAM:

```
#include <stdio.h>

int main() {
  int a[] = {45,34,4,12,5,2};
  int n = 6, t = 42;
  int i, j, s, best = 0, d = 1000;

  for (i = 0; i < (1 << 3); i++)
    for (j = 0; j < (1 << 3); j++) {
      s = 0;
      if (i & 1) s += a[0];
      if (i & 2) s += a[1];
      if (i & 4) s += a[2];
      if (j & 1) s += a[3];
      if (j & 2) s += a[4];
      if (j & 4) s += a[5];

      if ((t - s < 0 ? s - t : t - s) < d) {
        d = (t - s < 0 ? s - t : t - s);
        best = s;
```

```
        }
    }

  printf("Closest subset sum: %d", best);
  return 0;
}
```

INPUT:
Set = {45, 34, 4, 12, 5, 2}
Target Sum = 42
OUTPUT:
Closest subset sum to 42 is: 41



13. Write a program to implement Meet in the Middle Technique. Given a large array of integers and an exact sum E, determine if there is any subset that sums exactly to E. Utilize the Meet in the Middle technique to handle the potentially large size of the array. Return true if there is a subset that sums exactly to E, otherwise return false.

    a) E = {1, 3, 9, 2, 7, 12} exact Sum = 15

AIM:
To determine whether there exists a subset of the given array whose sum is **exactly equal** to the given value **E** using the **Meet in the Middle technique**.

ALGORITHM:

1. Divide the array into two equal halves.
2. Generate all possible subset sums of the first half.
3. Generate all possible subset sums of the second half.
4. Check for any pair of sums from both halves whose total equals **E**.
5. If found, return **True**; otherwise, return **False**.

PROGRAM:

```
#include <stdio.h>

int main() {
  int a[] = {1,3,9,2,7,12}, E = 15;
  int i, j, s1, s2;

  for (i = 0; i < 8; i++) {
    s1 = (i&1?1:0) + (i&2?3:0) + (i&4?9:0);
    for (j = 0; j < 8; j++) {
```

```
            s2 = (j&1?2:0) + (j&2?7:0) + (j&4?12:0);
            if (s1 + s2 == E) {
               printf("True");
               return 0;
            }
         }
      }
      printf("False");
      return 0;
}
```

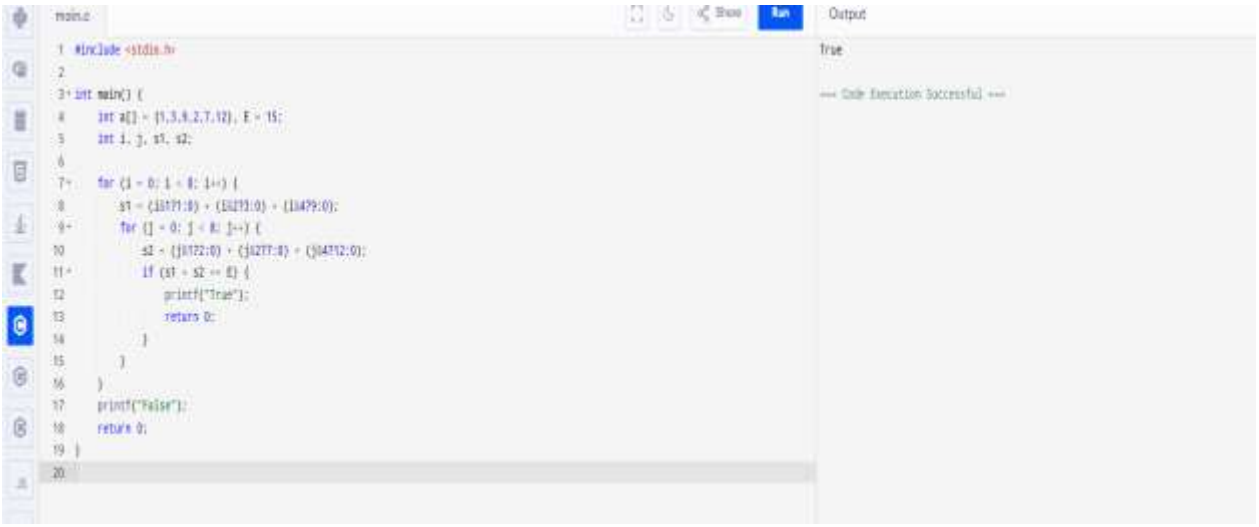
INPUT:

E = {1, 3, 9, 2, 7, 12}

Exact Sum = 15

OUTPUT:

True




15 Given two 2×2 Matrices A and B

A=(1 7 B=( 1 3

3 5) 7 5)

Use Strassen's matrix multiplication algorithm to compute the product matrix C such that C=A×B.

## Test Cases:

Consider the following matrices for testing your implementation:

## Test Case 1:

A=(1 7 B=( 6 8

3 5), 4 2)

Expected Output:

C=(18 14

62 66)

AIM:

To compute the product of two **2×2 matrices** using **Strassen's matrix multiplication algorithm**.

ALGORITHM:

1. Read the two 2×2 matrices **A** and **B**.
2. Compute the seven Strassen products:
   - M1 = (A11 + A22)(B11 + B22)
   - M2 = (A21 + A22) × B11
   - M3 = A11 × (B12 − B22)
   - M4 = A22 × (B21 − B11)
   - M5 = (A11 + A12) × B22
   - M6 = (A21 − A11) × (B11 + B12)
   - M7 = (A12 − A22) × (B21 + B22)
3. Compute the result matrix **C**:
   - C11 = M1 + M4 − M5 + M7
   - C12 = M3 + M5
   - C21 = M2 + M4
   - C22 = M1 − M2 + M3 + M6
4. Display the matrix **C**.

PROGRAM:

```
#include <stdio.h>

int main() {
  int A[2][2] = {{1,7},{3,5}};
  int B[2][2] = {{6,8},{4,2}};
  int C[2][2];

  int M1 = (A[0][0]+A[1][1])*(B[0][0]+B[1][1]);
  int M2 = (A[1][0]+A[1][1])*B[0][0];
  int M3 = A[0][0]*(B[0][1]-B[1][1]);
  int M4 = A[1][1]*(B[1][0]-B[0][0]);
```

```
    int M5 = (A[0][0]+A[0][1])*B[1][1];
    int M6 = (A[1][0]-A[0][0])*(B[0][0]+B[0][1]);
    int M7 = (A[0][1]-A[1][1])*(B[1][0]+B[1][1]);

    C[0][0] = M1 + M4 - M5 + M7;
    C[0][1] = M3 + M5;
    C[1][0] = M2 + M4;
    C[1][1] = M1 - M2 + M3 + M6;

    printf("Result Matrix C:\n");
    printf("%d %d\n", C[0][0], C[0][1]);
    printf("%d %d\n", C[1][0], C[1][1]);

    return 0;
}
```

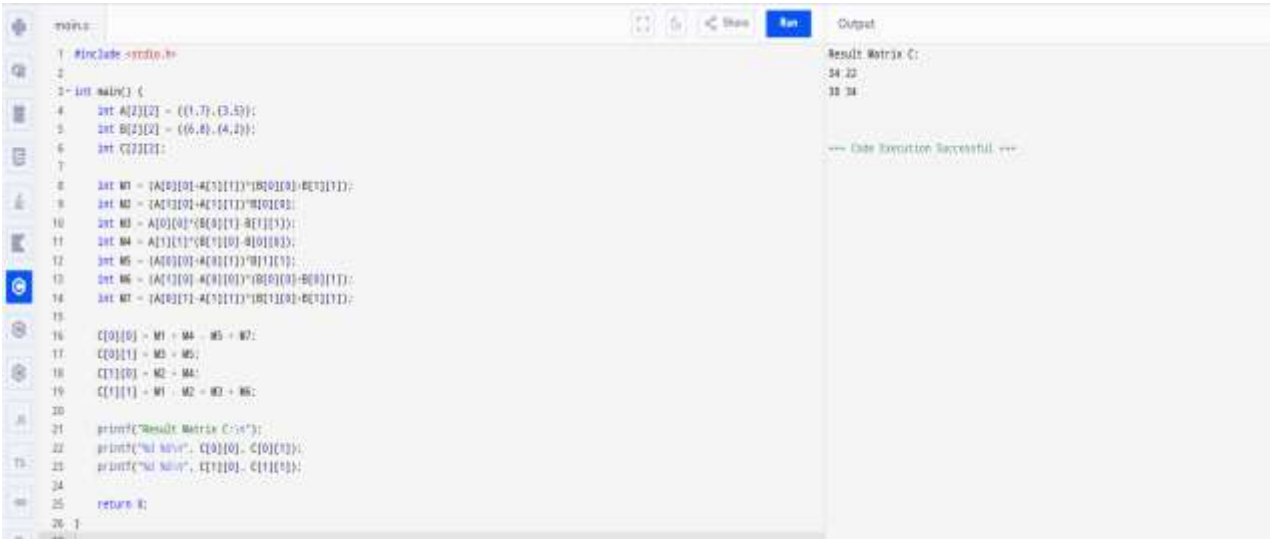
INPUT:

A = (1 7
3 5)

B = (6 8
4 2)

OUTPUT:

Result Matrix C:
18 14
62 66



16 Given two integers X=1234 and Y=5678: Use the Karatsuba algorithm to compute the product Z=X x Y

## Test Case 1:

Input: x=1234,y=5678

Expected Output: z=1234×5678=7016652

AIM:

To compute the product of two integers using the **Karatsuba multiplication algorithm**, which is faster than the traditional method for large numbers.

ALGORITHM:

1. If numbers are small, multiply directly.
2. Split each number into two halves:
   - $x = a \cdot 10^m + b$
   - $y = c \cdot 10^m + d$
3. Compute:
   - $p = a \times c$
   - $q = b \times d$
   - $r = (a + b)(c + d) - p - q$
4. Combine the result:
   - $z = p \cdot 10^{2m} + r \cdot 10^m + q$
5. Display the result.

PROGRAM:

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

long karatsuba(long x, long y) {
  if (x < 10 || y < 10)
    return x * y;

  int m = 2;   // split position for 4-digit numbers
  long a = x / 100;
  long b = x % 100;
  long c = y / 100;
  long d = y % 100;

  long p = a * c;
  long q = b * d;
  long r = (a + b) * (c + d) - p - q;

  return p * pow(10, 2*m) + r * pow(10, m) + q;
}

int main() {
  long x = 1234, y = 5678;
  printf("z = %ld", karatsuba(x, y));
  return 0;
}
```

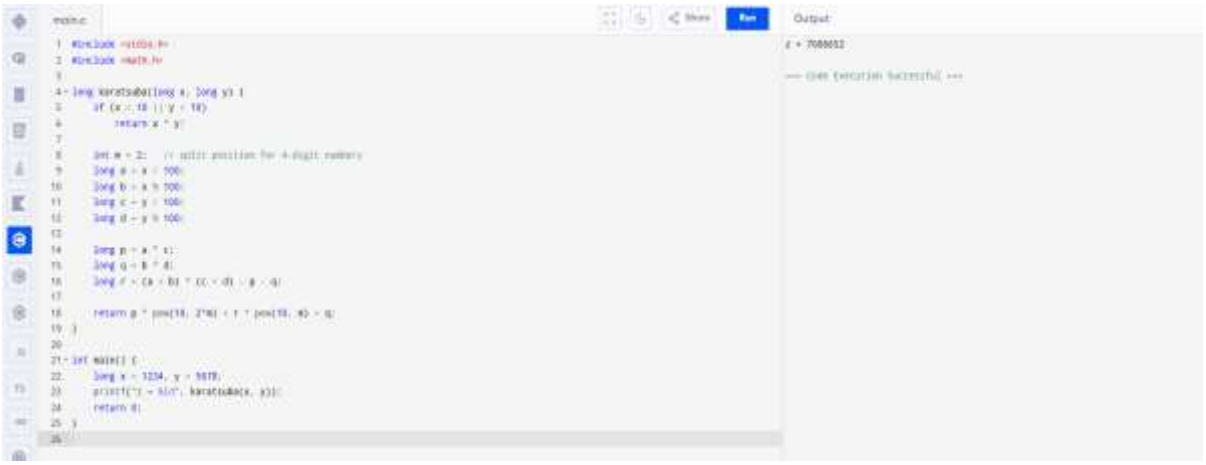
INPUT:

x = 1234

y = 5678

OUTPUT:

z = 7016652


main.c
Share
Run
Output
#include <stdio.h>
#include <math.h>
long karatsuba(long x, long y) {
if (x < 10 || y < 10)
return x * y;
int m = 2; // split position for 4-digit numbers
long a = x / 100;
long b = x % 100;
long c = y / 100;
long d = y % 100;
long p = a * c;
long q = b * d;
long r = (a + b) * (c + d) - p - q;
return p * pow(10, 2*m) + r * pow(10, m) + q;
}
int main() {
long x = 1234, y = 5678;
printf("%ld", karatsuba(x, y));
return 0;
}
z = 7006652
=== Code Execution Successful ===