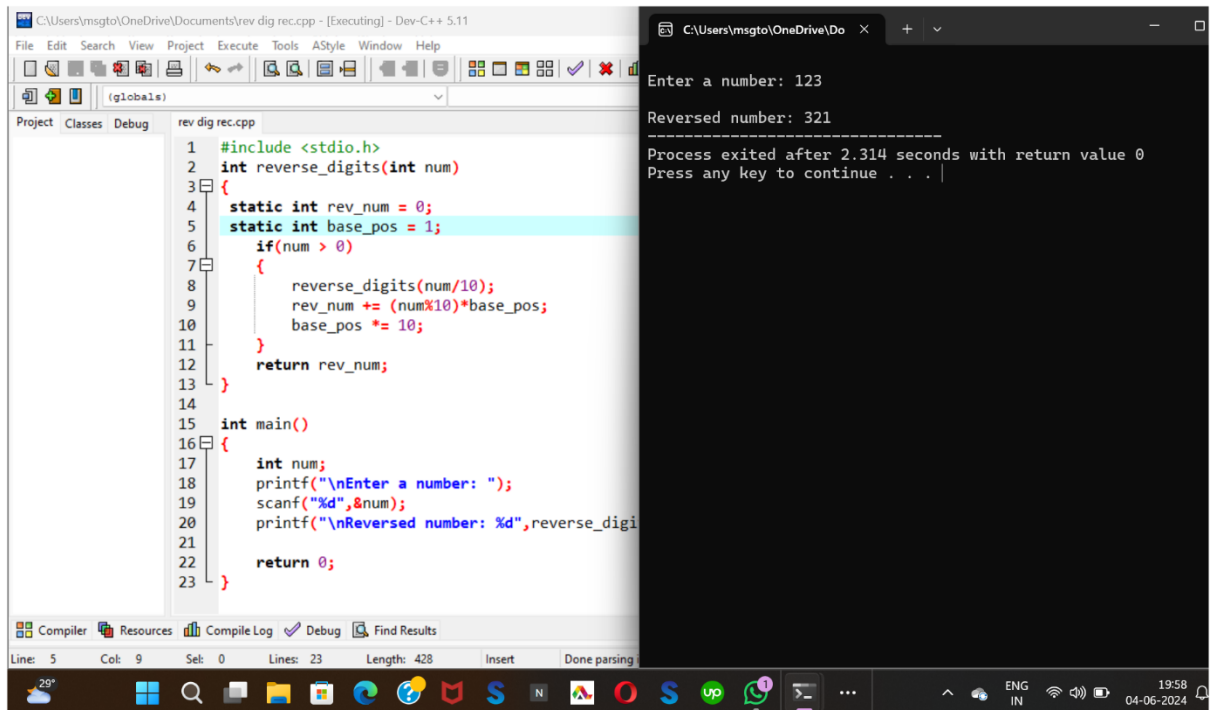


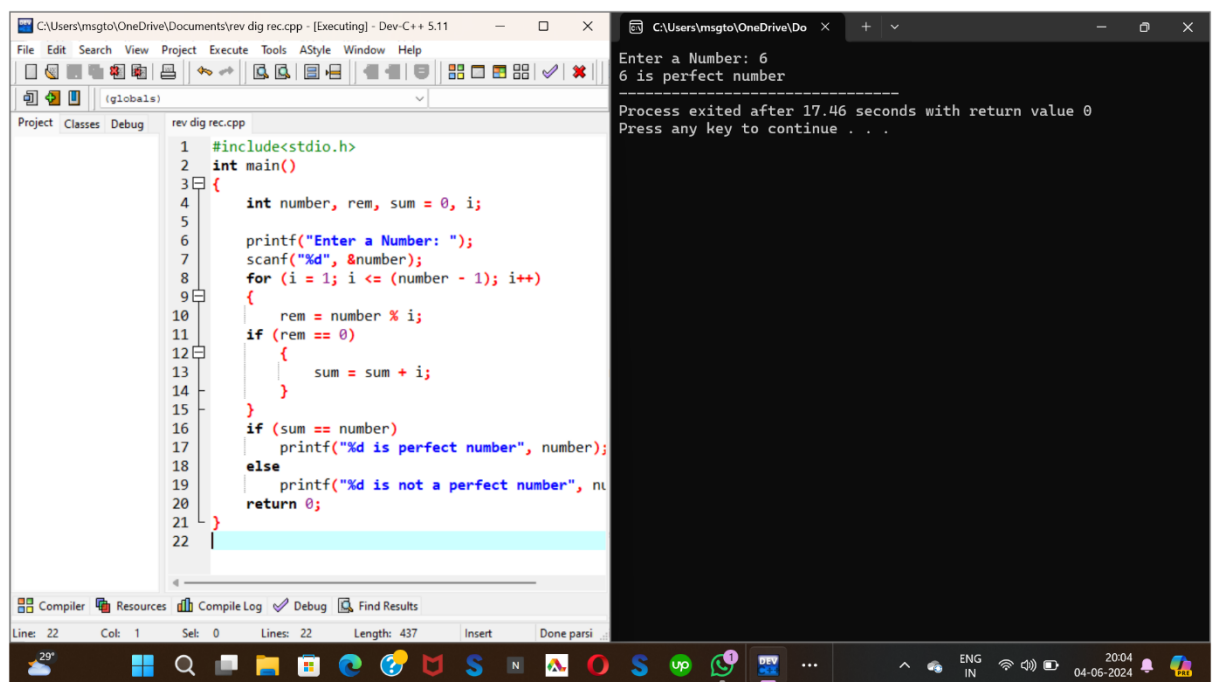
1. Write a program to find the reverse of a given number using recursive.



```
1 #include <stdio.h>
2 int reverse_digits(int num)
3 {
4     static int rev_num = 0;
5     static int base_pos = 1;
6     if(num > 0)
7     {
8         reverse_digits(num/10);
9         rev_num += (num%10)*base_pos;
10        base_pos *= 10;
11    }
12    return rev_num;
13 }
14
15 int main()
16 {
17     int num;
18     printf("\nEnter a number: ");
19     scanf("%d",&num);
20     printf("\nReversed number: %d",reverse_digits(num));
21
22     return 0;
23 }
```

Enter a number: 123  
Reversed number: 321  
-----  
Process exited after 2.314 seconds with return value 0  
Press any key to continue . . .

2. Write a program to find the perfect number.



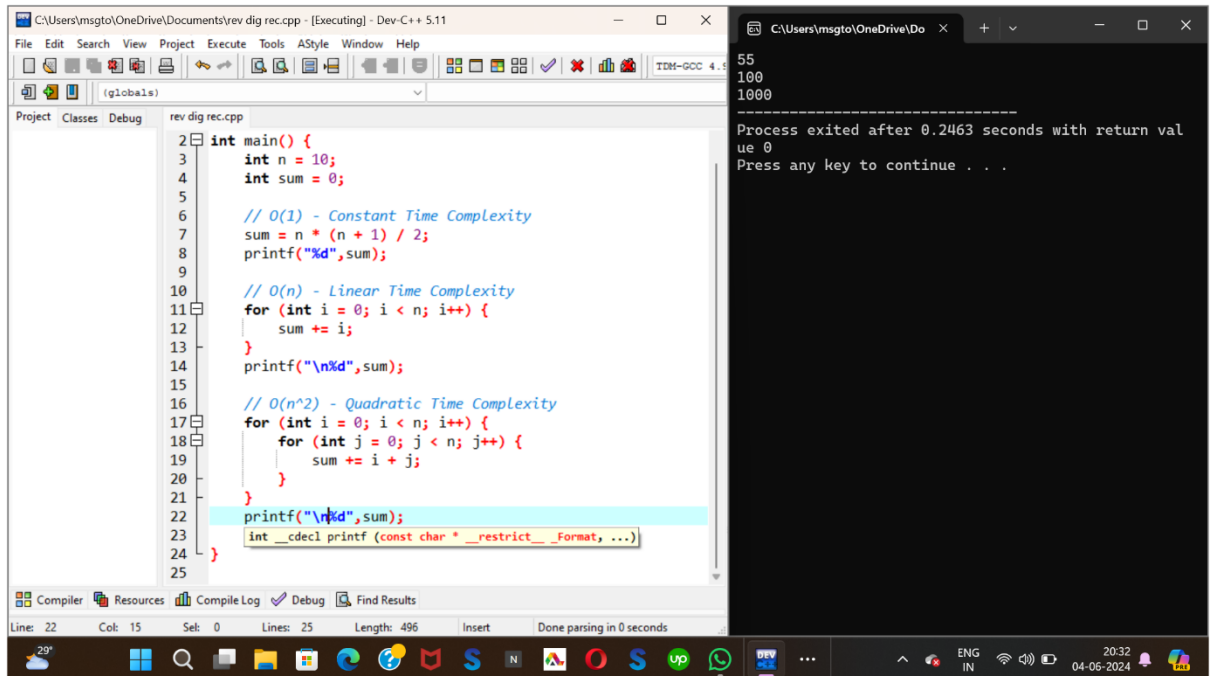
The screenshot shows a C++ program in Dev-C++ that checks if a number is perfect. The program prompts the user to enter a number, calculates the sum of its proper divisors, and prints the result. The output shows that 6 is a perfect number.

```
1 #include<stdio.h>
2 int main()
3 {
4     int number, rem, sum = 0, i;
5
6     printf("Enter a Number: ");
7     scanf("%d", &number);
8     for (i = 1; i <= (number - 1); i++)
9     {
10        rem = number % i;
11        if (rem == 0)
12        {
13            sum = sum + i;
14        }
15    }
16    if (sum == number)
17        printf("%d is perfect number", number);
18    else
19        printf("%d is not a perfect number", number);
20    return 0;
21 }
```

Enter a Number: 6  
6 is perfect number  
-----  
Process exited after 17.46 seconds with return value 0  
Press any key to continue . . .

Time complexity –  $O(n)$

3. Write C program that demonstrates the usage of these notations by analyzing the time complexity of some example algorithms.



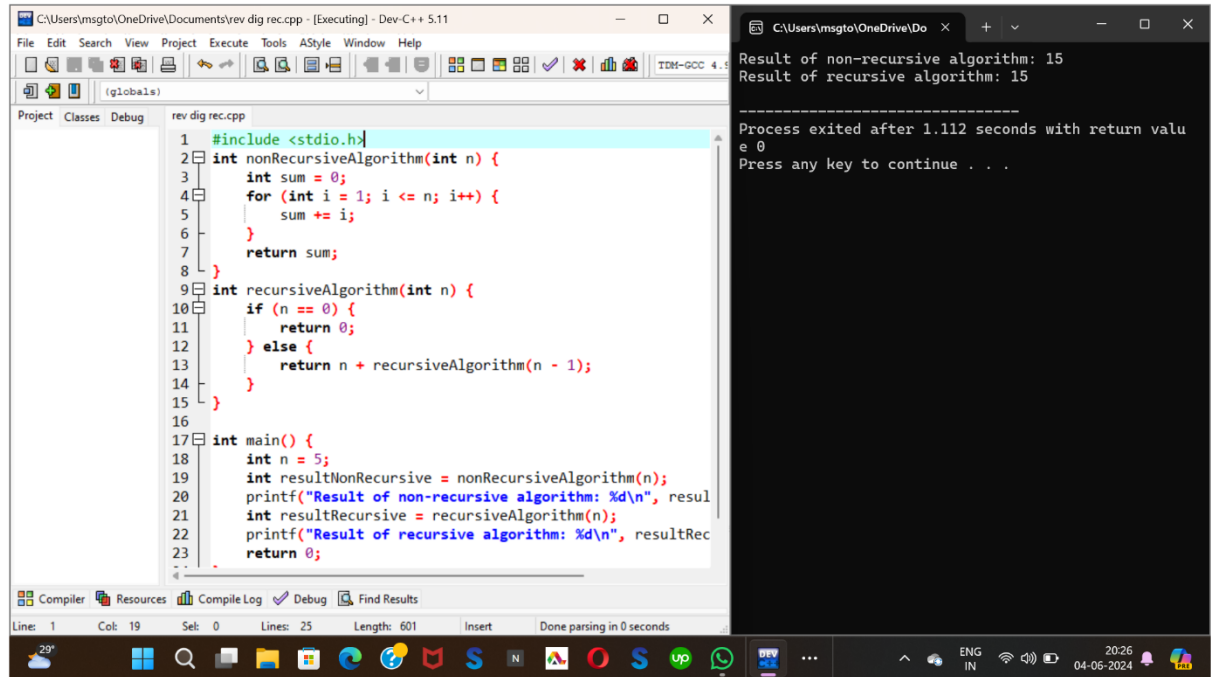
The screenshot shows a C program in Dev-C++ with the following code:

```
2 int main() {  
3     int n = 10;  
4     int sum = 0;  
5  
6     // O(1) - Constant Time Complexity  
7     sum = n * (n + 1) / 2;  
8     printf("%d", sum);  
9  
10  
11     // O(n) - Linear Time Complexity  
12     for (int i = 0; i < n; i++) {  
13         sum += i;  
14     }  
15     printf("\n%d", sum);  
16  
17     // O(n^2) - Quadratic Time Complexity  
18     for (int i = 0; i < n; i++) {  
19         for (int j = 0; j < n; j++) {  
20             sum += i + j;  
21         }  
22     }  
23     printf("\n%d", sum);  
24     int __cdecl printf(const char * __restrict __Format, ...)  
25 }
```

The output window shows the following output:

```
55  
100  
1000  
-----  
Process exited after 0.2463 seconds with return value 0  
Press any key to continue . . .
```

4. Write C programs that demonstrate the mathematical analysis of non-recursive and recursive algorithms.

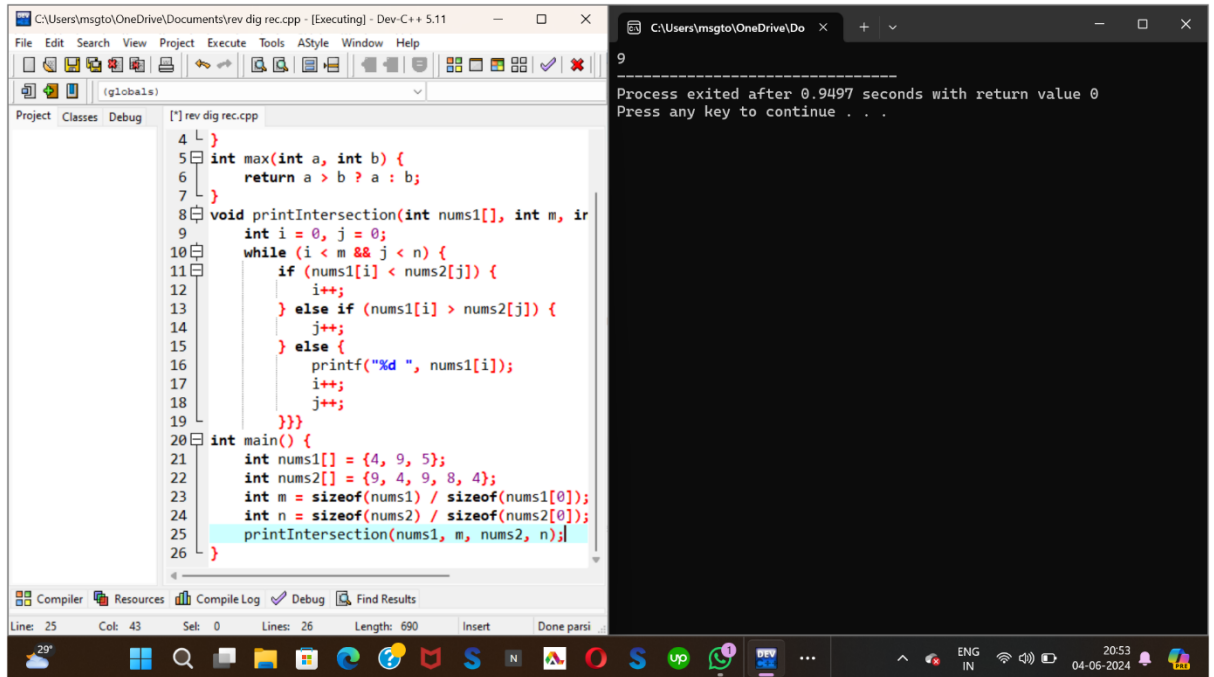


```
1 #include <stdio.h>
2 int nonRecursiveAlgorithm(int n) {
3     int sum = 0;
4     for (int i = 1; i <= n; i++) {
5         sum += i;
6     }
7     return sum;
8 }
9 int recursiveAlgorithm(int n) {
10    if (n == 0) {
11        return 0;
12    } else {
13        return n + recursiveAlgorithm(n - 1);
14    }
15 }
16
17 int main() {
18     int n = 5;
19     int resultNonRecursive = nonRecursiveAlgorithm(n);
20     printf("Result of non-recursive algorithm: %d\n", resultNonRecursive);
21     int resultRecursive = recursiveAlgorithm(n);
22     printf("Result of recursive algorithm: %d\n", resultRecursive);
23     return 0;
24 }
```

Result of non-recursive algorithm: 15  
Result of recursive algorithm: 15

Process exited after 1.112 seconds with return value 0  
Press any key to continue . . .

5. Given two integer arrays nums1 and nums2, return an array of their intersection. Each element in the result must appear as many times as it shows in both arrays and you may return the result in any order.

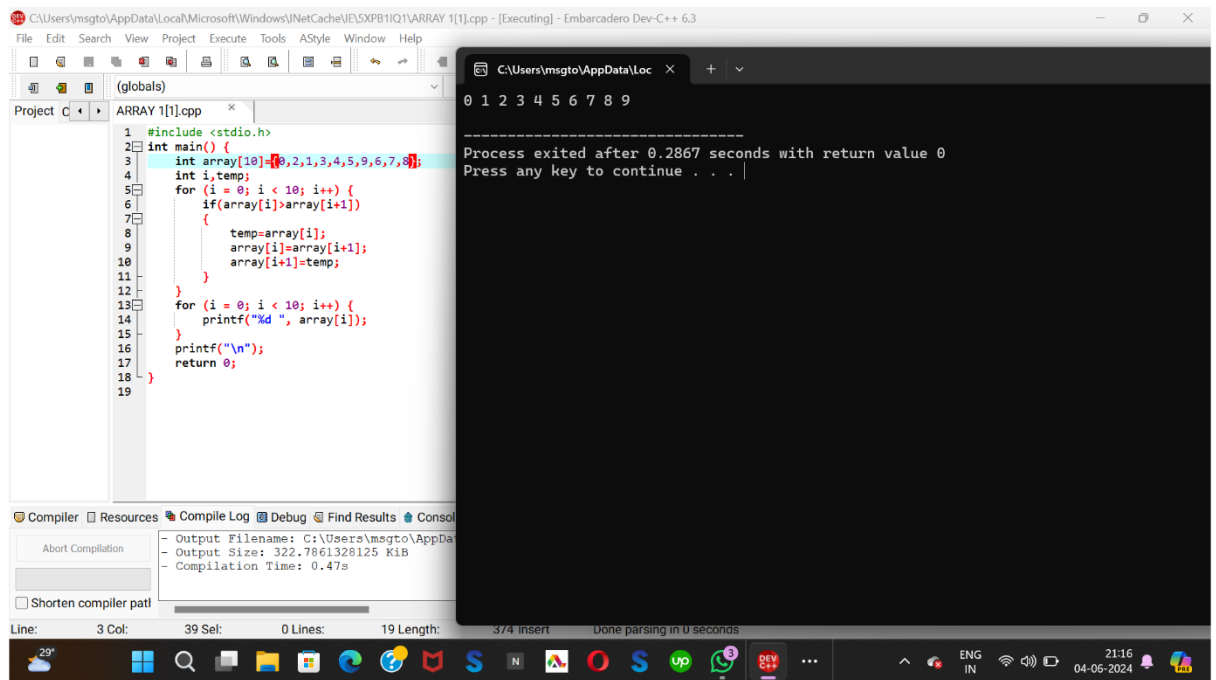


The screenshot shows a C++ IDE with a source code editor on the left and a console window on the right. The source code defines a function `max` to find the maximum of two integers, a function `printIntersection` to print the intersection of two arrays, and a `main` function that initializes two arrays, `nums1` and `nums2`, and calls `printIntersection` with their sizes and indices. The console window shows the output of the program, which is an empty array, indicating that the intersection of the two arrays is empty.

```
4 }
5 int max(int a, int b) {
6     return a > b ? a : b;
7 }
8 void printIntersection(int nums1[], int m, int nums2[], int n) {
9     int i = 0, j = 0;
10    while (i < m && j < n) {
11        if (nums1[i] < nums2[j]) {
12            i++;
13        } else if (nums1[i] > nums2[j]) {
14            j++;
15        } else {
16            printf("%d ", nums1[i]);
17            i++;
18            j++;
19        }
20    }
21    int main() {
22        int nums1[] = {4, 9, 5};
23        int nums2[] = {9, 4, 9, 8, 4};
24        int m = sizeof(nums1) / sizeof(nums1[0]);
25        int n = sizeof(nums2) / sizeof(nums2[0]);
26        printIntersection(nums1, m, nums2, n);
27    }
```

```
9
-----
Process exited after 0.9497 seconds with return value 0
Press any key to continue . . .
```

6. Given an array of integers `nums`, sort the array in ascending order and return it. You must solve the problem without using any built-in functions in  $O(n \log(n))$  time complexity and with the smallest space complexity possible.



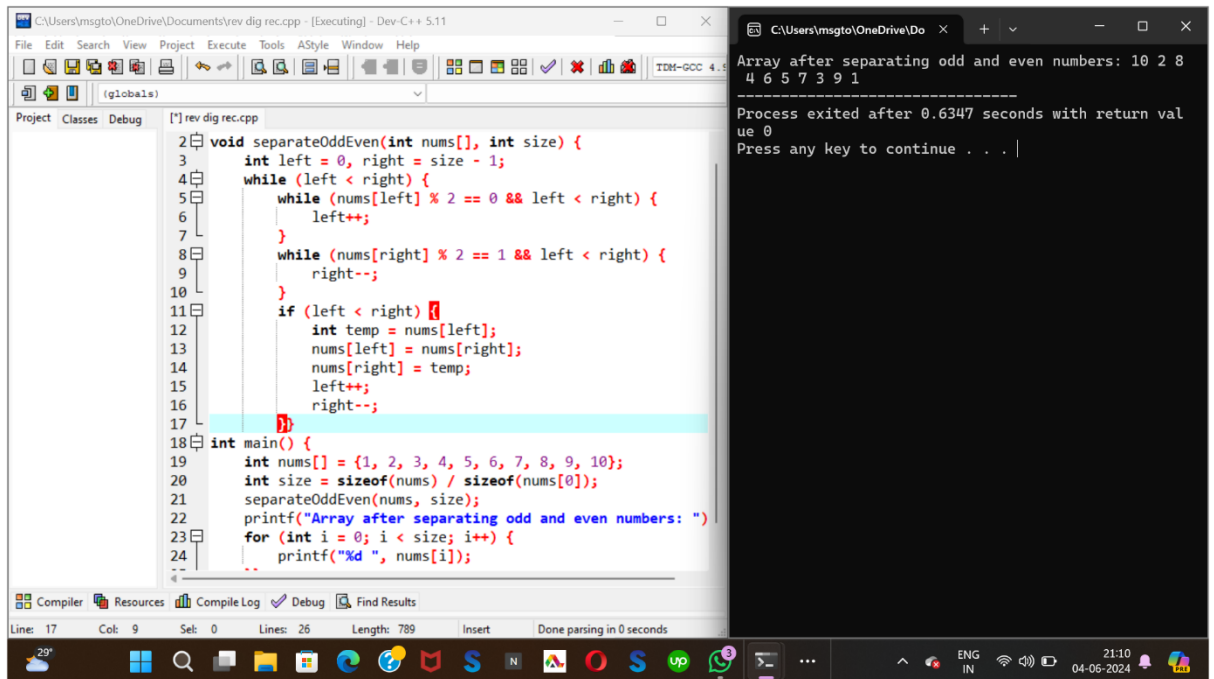
The screenshot shows a C++ IDE with a project named "ARRAY 1[1].cpp". The code implements a bubble sort algorithm. The array is initialized with the values [0, 2, 1, 3, 4, 5, 9, 6, 7, 8]. The algorithm uses two nested loops: an outer loop for  $i$  from 0 to 9, and an inner loop for  $j$  from  $i$  to 9. In each iteration of the inner loop, if `array[i] > array[i+1]`, the elements are swapped. After the sorting loop, the array is printed using `printf("%d ", array[i]);` for each element. The output window shows the sorted array: 0 1 2 3 4 5 6 7 8 9. The process exited after 0.2867 seconds with a return value of 0.

```
1 #include <stdio.h>
2 int main() {
3     int array[10] = {0, 2, 1, 3, 4, 5, 9, 6, 7, 8};
4     int i, temp;
5     for (i = 0; i < 10; i++) {
6         if (array[i] > array[i+1]) {
7             temp = array[i];
8             array[i] = array[i+1];
9             array[i+1] = temp;
10        }
11    }
12    for (i = 0; i < 10; i++) {
13        printf("%d ", array[i]);
14    }
15    printf("\n");
16    return 0;
17 }
```

Output: 0 1 2 3 4 5 6 7 8 9

Process exited after 0.2867 seconds with return value 0  
Press any key to continue . . .

7. Given an array of integers nums, half of the integers in nums are odd, and the other half are even.



The screenshot shows a C++ IDE with a file named 'rev dig rec.cpp'. The code implements a function 'separateOddEven' that uses a two-pointer technique to swap odd and even numbers in an array. The 'main' function initializes an array of 10 numbers (1 to 10) and prints the array after the function is called. The output window shows the array after separation: 10 2 8 4 6 5 7 3 9 1.

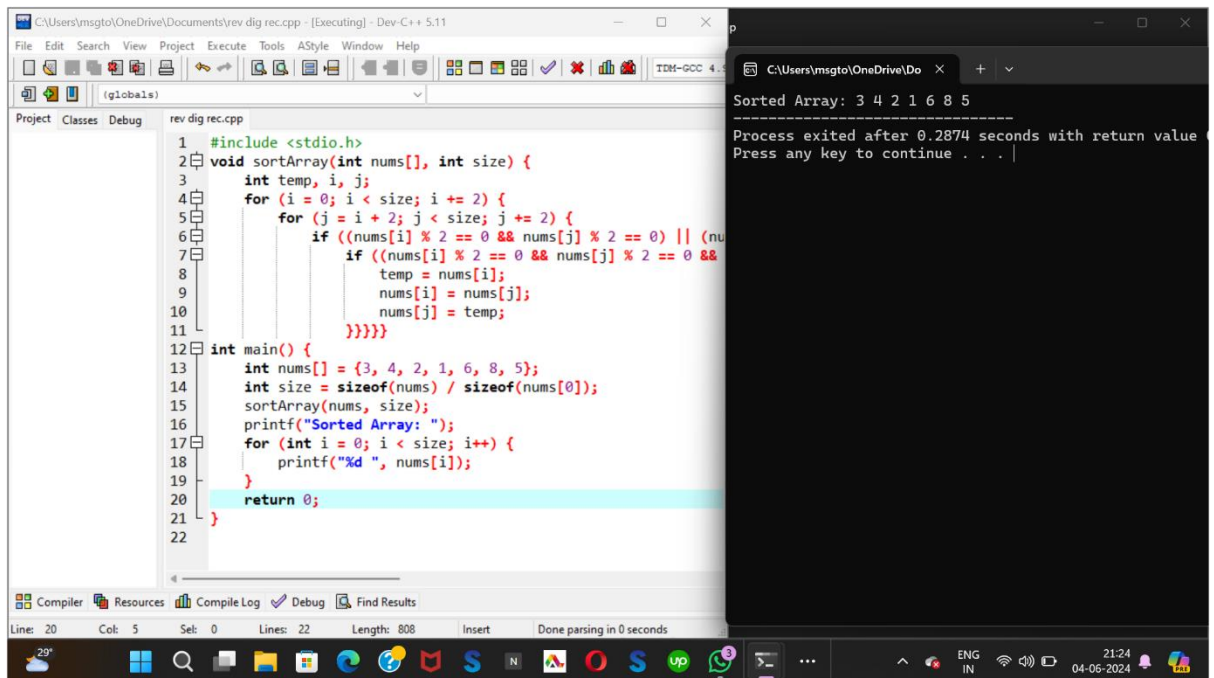
```
2 void separateOddEven(int nums[], int size) {
3     int left = 0, right = size - 1;
4     while (left < right) {
5         while (nums[left] % 2 == 0 && left < right) {
6             left++;
7         }
8         while (nums[right] % 2 == 1 && left < right) {
9             right--;
10        }
11        if (left < right) {
12            int temp = nums[left];
13            nums[left] = nums[right];
14            nums[right] = temp;
15            left++;
16            right--;
17        }
18    }
19    int main() {
20        int nums[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
21        int size = sizeof(nums) / sizeof(nums[0]);
22        separateOddEven(nums, size);
23        printf("Array after separating odd and even numbers: ");
24        for (int i = 0; i < size; i++) {
25            printf("%d ", nums[i]);
26        }
27    }
```

Array after separating odd and even numbers: 10 2 8 4 6 5 7 3 9 1

Process exited after 0.6347 seconds with return value 0

Press any key to continue . . .

8. Sort the array so that whenever  $\text{nums}[i]$  is odd,  $i$  is odd, and whenever  $\text{nums}[i]$  is even,  $i$  is even. Return any answer array that satisfies this condition.



The image shows a screenshot of a C++ IDE (Dev-C++) with a project named 'rev dig rec.cpp'. The code implements a sorting algorithm that rearranges an array of integers such that even numbers are at even indices and odd numbers are at odd indices. The initial array is {3, 4, 2, 1, 6, 8, 5}. The output shows the sorted array as 3 4 2 1 6 8 5, which is the same as the input array. The process exited after 0.2874 seconds with a return value of 0.

```
1 #include <stdio.h>
2 void sortArray(int nums[], int size) {
3     int temp, i, j;
4     for (i = 0; i < size; i += 2) {
5         for (j = i + 2; j < size; j += 2) {
6             if ((nums[i] % 2 == 0 && nums[j] % 2 == 0) || (nums[i] % 2 == 1 && nums[j] % 2 == 1)) {
7                 if ((nums[i] % 2 == 0 && nums[j] % 2 == 0 && nums[i] < nums[j]) || (nums[i] % 2 == 1 && nums[j] % 2 == 1 && nums[i] > nums[j])) {
8                     temp = nums[i];
9                     nums[i] = nums[j];
10                    nums[j] = temp;
11                }
12            }
13        }
14    }
15}
16 int main() {
17     int nums[] = {3, 4, 2, 1, 6, 8, 5};
18     int size = sizeof(nums) / sizeof(nums[0]);
19     sortArray(nums, size);
20     printf("Sorted Array: ");
21     for (int i = 0; i < size; i++) {
22         printf("%d ", nums[i]);
23     }
24     return 0;
25 }
```

Sorted Array: 3 4 2 1 6 8 5  
Process exited after 0.2874 seconds with return value 0  
Press any key to continue . . .