Pseudo code

1. diagonal_difference_1

Step 1 Use two nested loops to iterate on matrix elements.

Step 2 If counters match, add the value at the array index to first diagonal sum.

If first counter matches the operation below, add the value at the array index to second diagonal sum.

Step 3 Find the difference between first diagonal and the second diagonal.

Step 4 Return the absolute value of the difference.

```
ALGORITHM: diagonal_difference_1(int **a, int size)
  //Computes absolute diagonal difference
  //Input: 2d array of integers, size of the square matrix
  //Output: absolute diagonal difference of the matrix a

  FOR i ← 0 TO size DO
    FOR j ← 0 TO size DO
      IF (i == j)
        first_diagonal ← first_diagonal + a[i][j]
      IF (i == size - j - 1)
        second_diagonal ← second_diagonal + a[i][j]

  difference ← first_diagonal - second_diagonal
  RETURN ABS(difference)
```

2. diagonal_difference_2

Step 1 Use one loop to iterate on the matrix elements.

Step 2 Compute sum using the indices of the first and second diagonal.

Step 3 Find the difference between first diagonal and the second diagonal.

Step 4 Return the absolute value of the difference.

*ALGORITHM: diagonal_difference_2(int **a, int size)*

//Computes absolute diagonal difference

//Input: 2d array of integers, size of the square matrix

//Output: absolute diagonal difference of the matrix a

FOR** i ← 0 **TO** size **DO

first_diagonal ← first_diagonal + a[i][i]

second_diagonal ← second_diagonal + a[size - i - 1][i]

difference ← first_diagonal - second_diagonal

***RETURN ABS**(difference)*

Time complexity

Let matrix dimension (n)

	diagonal_difference_1	**diagonal_difference_2 (*Better*)**
Basic operation	Comparison (i < size)	Comparison (i < size)
Execution times	$\sum_{i=0}^{n}\left(\sum_{j=0}^{n} 2\right)$	$\sum_{i=0}^{n} 2$
Efficiency class	$O(2n^2)$	$O(2n)$
Growth	Quadratic	Linear
Space (Memory)	$O(n^2)$	$O(n^2)$

diagonal_difference_1

The algorithm requires *two nested loops* to complete a single iteration on a matrix of size n ; therefore, it must go through $n * n$ steps to complete the traverse.

diagonal_difference_2

The algorithm takes advantage of the previously known pattern of diagonal indices and uses *one loop* counter to extract the diagonal elements from the matrix; therefore, it must go through n steps to compute the required summation.

Space complexity

Both algorithms depend on the input size of the matrix.

$O(n^2)$

For both algorithms, space grows linearly with the input size of the matrix and constant space is used to initialize diagonal sums.

Implementation

ALGORITHM: *diagonal_difference_1(int **a, int size)*

```
unsigned int diagonal_difference_1(int **a, int size)
{
    int i, j, d1 = 0, d2 = 0, diff;

    for (i = 0; i < size; i++)
    {
        for (j = 0; j < size; j++)
        {
            if (i == j)
                d1 += a[i][j];

            if (i == size - j - 1)
                d2 += a[i][j];
        }
    }

    diff = abs(d1 - d2);
    return (diff);
}
```

ALGORITHM: *diagonal_difference_2(int **a, int size)*

```
unsigned int diagonal_difference_2(int **a, int size)
{
    int i, d1 = 0, d2 = 0, diff;

    for (i = 0; i < size; i++)
    {

        d1 += a[i][i];

        d2 += a[size - i - 1][i];
    }

    diff = abs(d1 - d2);

    return (diff);
}
```

Testing (*main.c*)

```c
#include "main.h"
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int **a, size, i, j;

    printf("Square matrix size: ");
    scanf("%d", &size);

    if (size < 2)
        return (-1);

    a = (int **)malloc(size * sizeof(int *));
    if (a == NULL)
    {
        free(a);
        return (-2);
    }

    for (i = 0; i < size; i++)
    {
        a[i] = (int *)malloc(size * sizeof(int));
        if (a[i] == NULL)
        {
            for(j = i; j >= 0; j--)
                free(a[i]);
            free(a);
            return (-2);
        }
    }

    for (i = 0; i < size; i++)
    {
        for (j = 0; j < size; j++)
        {
            scanf("%d", &a[i][j]);
            if (a[i][j] > 100 || a[i][j] < -100)
                return (-1);
        }
    }
```

```
    printf("Algorithm 1\nAbsolute diagonal difference: %u\n",
diagonal_difference_1(a, size));
    printf("Algorithm 2\nAbsolute diagonal difference: %u\n",
diagonal_difference_2(a, size));

    return (0);
}
```

Case 1:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

PS D:\Learn\ALX SE\Algorithms project\absolute_difference> ./a
Square matrix size: 3
1 2 3
4 5 6
9 8 9
Algorithm 1
Absolute diagonal difference: 2
Algorithm 2
Absolute diagonal difference: 2
PS D:\Learn\ALX SE\Algorithms project\absolute_difference>
```

Case 2:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

PS D:\Learn\ALX SE\Algorithms project\absolute_difference> ./a
Square matrix size: 3
11 2 4
4 5 6
10 8 -12
Algorithm 1
Absolute diagonal difference: 15
Algorithm 2
Absolute diagonal difference: 15
PS D:\Learn\ALX SE\Algorithms project\absolute_difference>
```