This is a real-life example of a data structure (“pixel sum”) that we use all the time in various computer vision and graphics work. The code is highly performance critical: it is called millions of times per image, often with very large kernels (thousands of pixels).

At the end of the text there is a header that shows the interface that needs to be implemented.

In its constructor, the class receives as its input an 8-bit two-dimensional pixel buffer (we tend to work with grayscale data or one channel at a time). At this stage the constructor can take its sweet time to set up its data structures. The input buffers are guaranteed to be < 4096 x 4096 pixels. Any changes made into the input buffer later *should not* affect the results of future queries, i.e., the constructor should take its own copy of the input.

The input buffer pixels are organized linearly with (0,0) at the beginning, e.g., accessing pixel (x = 5, y = 9) can be done fetching the memory address: buffer[5 + 9 * xWidth];

Once you have the relevant data structures in place, the class should be able to provide efficient implementations of the following operations:

```
unsigned int getPixelSum (int x0, int y0, int x1, int y1) const;
double getPixelAverage (int x0, int y0, int x1, int y1) const;
```

In other words, it should return the sum and average of the window containing pixels (x0,y0) - (x1,y1) where all coordinates are **inclusive**. This also means that the window (x0,y0) - (x0,y0) contains a single pixel.

For example: if the pixels inside the window have the values [0, 4, 0, 2, 1, 0], their sum is 0+4+0+2+1+0 = 7, and their average is 7/6 = 1.166...

The code should gracefully handle cases x0 < x1, and y0 < y1: swap the values if necessary, i.e., (5,4) - (3,6) should be treated as (3,4) - (5,6).

If the search window extends outside the buffer, the code **should treat the non-existent pixels as if they had a value of 0.** This has several implications: first, if the search window is completely outside the buffer, the return value of all the functions should be 0. Secondly, the area of the *specified* window should be respected. For example, if we have a 2x2 buffer with values [1,1,1,1] and we request the average of pixels (1,1)-(3,3) the resulting value should be 1/9 (=0.111…), as the window encompasses 8 non-existing pixels. Similarly, for the same buffer, the average of the window (0,0)-(3,3) would be 4 / 16 (0.25).

One can make a naive implementation of this in less than five minutes. Implementing the naive version first is a good idea, as it can be used as a reference in your tester code. However, you should keep in mind that typical calls to these functions have windows containing tens or hundreds of thousands of pixels: a naive implementation would be dead slow. Well-known O(1) solutions for the problem exist, and should be utilized in the implementation.

Feel free to use Google, StackOverflow or any other resources you would normally consult!

Once you have the basic functionality in place, we'd like to have two more functions.

```
int getNonZeroCount (int x0, int y0, int x1, int y1) const;
double getNonZeroAverage (int x0, int y0, int x1, int y1) const;
```

The first function tells how many pixels within window (x0,y0) - (x1,y1) have a non-zero value. The latter function computes *the average of **those pixels** that have non-zero values*.

For example: if the window contains pixels [0, 4, 0, 2, 1, 0], its pixel sum is (4+2+1) = 7, its average is (7/6) = 1,166..., its nonZeroCount is 3 (three non-zero pixels), and its nonZeroAverage = (4+2+1) / 3 = 2.333…

If the search window does not contain any non-zero pixels, the return value for getNonZeroAverage() should be 0 (instead of NaN).

Please provide a C++ implementation of the class (any variant of C++ is fine), as well as any tester code you have used. Do not use any external libraries (except the C++ Standard Library).

What are we looking for?

When looking at your code, we emphasize the following qualities:

1. The code should produce correct and exact output for all valid input data.
2. There should be sufficient testing code to demonstrate the robustness and handling of obvious edge cases for all of the functions.
3. The code should not crash or leak memory (assuming valid input is provided) under *any circumstances.*
4. All non-obvious parts of your code are properly commented.
5. The algorithm chosen is O(1) and obvious code optimizations have been made. It should be noted here that there are **tons of tricks** to speed up this particular algorithm. We don't expect you to implement all of them.
6. The general style and readability of the code.

Essay questions

After you're done with the coding part, here's a couple of additional quick questions (no coding needed):

a) Typical buffers in our use cases are extremely sparse matrices, i.e., the vast majority of the pixels are zero. How would you exploit this to optimize the implementation?

b) What changes to the code and the API need to be made if buffer dimensions can be >= 4096 x 4096? Also, what would happen if instead of 8 bits per pixel we have 16 bits per pixel?

c) What would change if we needed to find the **maximum value** inside the search window (instead of the sum or average)? You don't need to code this, but it would be interesting to know how you would approach this problem in general, and what is your initial estimate of its O-complexity.

d) How would multi-threading change your implementation? What kinds of performance benefits would you expect?

e) Outline shortly how the class would be implemented on a GPU (choose any relevant technology, e.g. CUDA, OpenCL). No need to code anything, just a brief overview is sufficient.

Interface

```
//--------------------------------------------------------------------------
// Class for providing fast region queries from an 8-bit pixel buffer.
// Note: all coordinates are *inclusive* and clamped internally to the borders
// of the buffer by the implementation.
//
```

```
// For example: getPixelSum(4,8,7,10) gets the sum of a 4x3 region where top left
// corner is located at (4,8) and bottom right at (7,10). In other words
// all coordinates are _inclusive_.
//
// If the resulting region after clamping is empty, the return value for all
// functions should be 0.
//
// The width and height of the buffer dimensions < 4096 x 4096.
//--------------------------------------------------------------------------

class PixelSum
{
public:
                    PixelSum            (const unsigned char* buffer, int xWidth, int
yHeight);
                    ~PixelSum           (void);
                    PixelSum            (const PixelSum&);
    PixelSum&       operator=           (const PixelSum&);

    unsigned int    getPixelSum         (int x0, int y0, int x1, int y1) const;
    double          getPixelAverage     (int x0, int y0, int x1, int y1) const;

    int             getNonZeroCount     (int x0, int y0, int x1, int y1) const;
    double          getNonZeroAverage   (int x0, int y0, int x1, int y1) const;
};
```