Suppose we are interested in implementing a special linear algebra operation:

$$y = (A + A^\top)x + b.$$

In Julia, this is quite easy to do. First, we would check that A is square (otherwise, the addition of A to its transpose does not conform.). Then, we set the entries of y by a nested pair of `for` loops given the fact that:

$$(Ax)_i = \sum_{j=1}^{n} A_{i,j}x_j \quad \text{and} \quad (A^\top x)_i = \sum_{j=1}^{n} A_{j,i}x_j.$$

```
my_special_problem2! (generic function with 1 method)
```

```
begin
    function my_special_problem!(y::Vector, A::Matrix, x::Vector, b::Vector)
        m, n = size(A)
        m ≠ n && throw(DimensionMismatch("Matrix A is not square."))
        length(y) == n == length(x) == length(b) || throw(DimensionMismatch("One of these vectors
is behaving like an orangutan."))
        for i = 1:n
            y[i] = b[i]
            for j = 1:n
                y[i] += (A[i,j]+A[j,i])*x[j]
            end
        end
        return y
    end
    function my_special_problem2!(y::Vector, A::Matrix, x::Vector, b::Vector)
        m, n = size(A)
        m ≠ n && throw(DimensionMismatch("Matrix A is not square."))
        length(y) == n == length(x) == length(b) || throw(DimensionMismatch("One of these vectors
is behaving like an orangutan."))
        for i = 1:n
            y[i] = b[i]
        end
        for i = 1:n
            for j = 1:n
                y[i] += A[j,i]*x[j]
                y[j] += A[j,i]*x[i]
            end
        end
        return y
    end
end
```

```
y =  Float64[0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0,
```

```
y = zeros(15)
```

```
A = 15×15 Array{Float64,2}:
 0.8651458050686811   0.4786522390210759   …  0.7676036624409959
 0.6589812034231886   0.8294394961068174      0.7464382407057901
 0.03660790033377448  0.7825869687819169      0.7401378399983083
 0.48428799408086465  0.5619495540046493      0.0014851585726518568
 0.903777243029479    0.3342242386514278      0.20226940046272146
 0.03395180399564812  0.5515801565516458   …  0.37988698860200754
 0.9829393654222167   0.86570946457736        0.40736630989943334
 ⋮                                         ⋱
 0.3362659439540163   0.5779636330924531      0.1225678886446806
 0.00587985097237409  0.6933943991273765   …  0.343210282678055
 0.9582634418709455   0.9459398865546045      0.2067179771964438
 0.8344646712482477   0.9949275055103766      0.12889772102915598
 0.3774474930284324   0.5564209753220111      0.37454538228485723
 0.6606678713181866   0.39398713763755855     0.3164433501797774
```

```
A = rand(15, 15)
```

```
x = Float64[0.997113, 0.519819, 0.0791804, 0.119173, 0.422666, 0.574166, 0.895408
```

```
x = rand(15)
```

```
b = Float64[0.882056, 0.496812, 0.650064, 0.0433487, 0.881431, 0.989835, 0.77313
```

```
b = rand(15)
```

```
Float64[9.32556, 9.99617, 8.15599, 8.51033, 8.67362, 7.41787, 8.5096, 7.0482, 7
```

```
my_special_problem!(y, A, x, b)
```

```
using LinearAlgebra
```

```
4.864753555590494e-15
```

```
norm(y - ((A+A')*x + b))
```

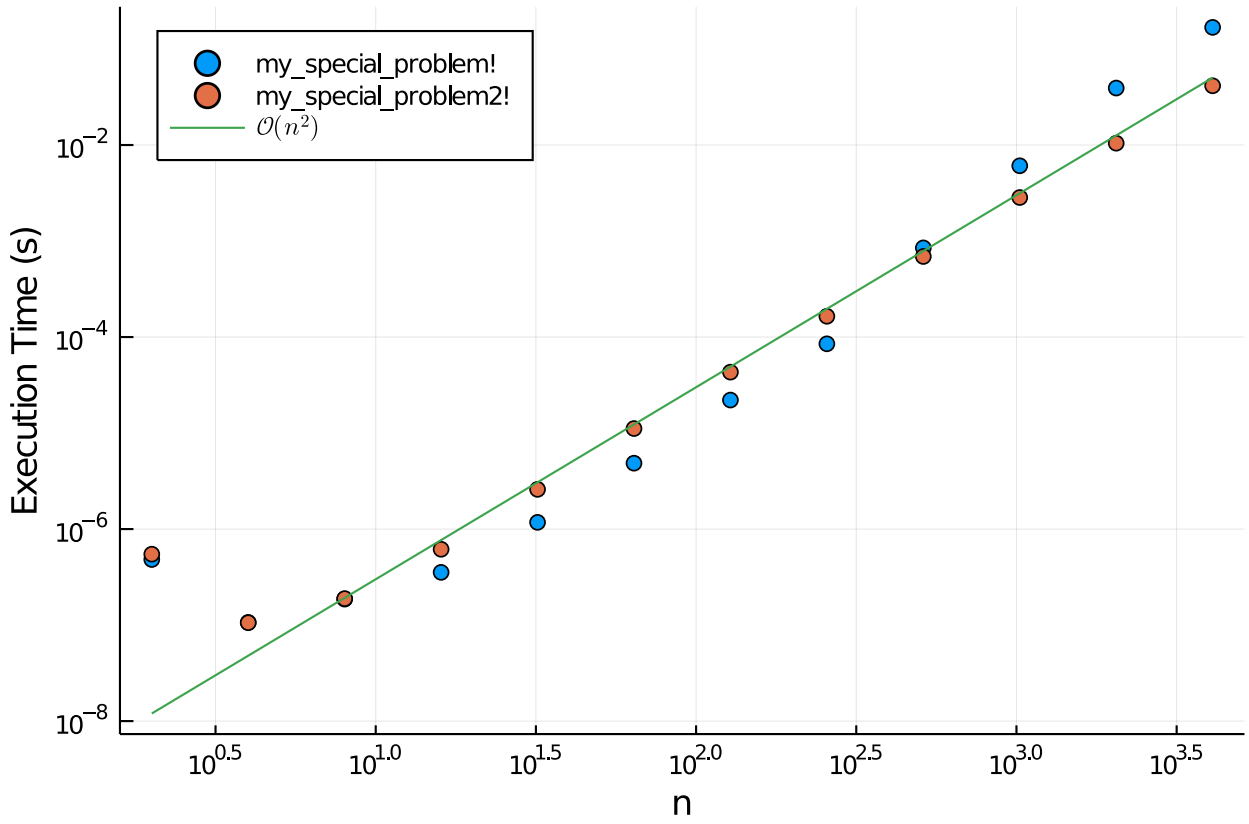
When writing a special linear algebra problem such as the one above, it is important to keep track of the number of flops (floating-point operations), because this plays a basic role in prediciting the computational time. For any n, we can see by the nested `for` loops that after `y` is set to `b`, it takes $2n$ additions and n multiplications for each y_i. In total, the function takes $3n^2$ flops. The second variant in fact costs $4n^2$ flops but is more friendly vis-à-vis the contiguous column-major storage of `A`.

When n is small, the constants 3 and 4 may be helpful, but as n gets large, what we really want to know is "if I double n, how much longer must I wait?" That question can be answered just with the exponent, 2: a problem twice the size takes four times as long.

These arguments justify the new notation we are learning so we can summarize computational complexity more tersely and thus with more relevance: $3n^2 = \mathcal{O}(n^2)$ as $n \to \infty$.

```
begin
    j = 1
    n = 2 .^ (1:12)
    t = zeros(length(n))
    r = zeros(length(n))
    for n in n
        y = zeros(n)
        A = rand(n, n)
        x = rand(n)
        b = rand(n)
        t[j] = @elapsed for i in 1:10 my_special_problem!(y, A, x, b) end
        t[j] /= 10
        r[j] = @elapsed for i in 1:10 my_special_problem2!(y, A, x, b) end
        r[j] /= 10
        global j += 1
    end
end
```

```
using Plots
```



```
begin
    scatter(n, t; xscale=:log10, yscale=:log10, label="my_special_problem!", legend=:topleft)
    scatter!(n, r; xscale=:log10, yscale=:log10, label="my_special_problem2!", legend=:topleft)
    plot!(n, 3e-9n.^2, label="\$\\mathcal{O}(n^2)\$")
    xlabel!("n")
    ylabel!("Execution Time (s)")
end
```

In the early stages, the first implementation is a bit faster than the second. But as soon as memory complexity begins to play a role, respecting the contiguous ordering of `A` improves the timings by about a factor of four.

```
4.05864085012903
```

```
t[end]/r[end]
```