Please send any comments or criticisms to dwharder@alumni.uwaterloo.ca
with the subject ECE 250 Questions 2.4.
Assistances and comments will be acknowledged.

2.4*a* One of your junior engineers-in-training comes to you and indicates that a recent paper from Caltech gives an outline for a *faster* matrix-matrix multiplication routine. What should your first response be?

2.4.1*a* The C++ source code `operators.1.cpp`

```
int main() {
        int c = 7;
        c += 12;
        c *= 1533;

        return 0;
}
```

can be compiled and disassembled using

```
% g++ -o operators.1.cpp
% objdump -d operators.1.o
```

to produce

```
operators.1.o:     file format elf64-x86-64

Disassembly of section .text:

0000000000000000 <main>:
   0:   55                      push   %rbp
   1:   48 89 e5                mov    %rsp,%rbp
   4:   c7 45 fc 07 00 00 00    movl   $0x7,0xfffffffffffffffc(%rbp)
   b:   83 45 fc 0c             addl   $0xc,0xfffffffffffffffc(%rbp)
   f:   8b 45 fc                mov    0xfffffffffffffffc(%rbp),%eax
  12:   69 c0 fd 05 00 00       imul   $0x5fd,%eax,%eax
  18:   89 45 fc                mov    %eax,0xfffffffffffffffc(%rbp)
  1b:   b8 00 00 00 00          mov    $0x0,%eax
  20:   c9                      leaveq
  21:   c3                      retq
```

Identify which line or lines correspond to the four operators `=`, `+=`, `*=` and `return`.

2.4.1*b* Given the C++ source code `operators.2.cpp`:

```
int main() {
        int c = 4134;
        c *= 3;

        return 0;
}
```

when this is compiled into an object file and disassembled, we get the output

```
operators.2.o:     file format elf64-x86-64

Disassembly of section .text:

0000000000000000 <main>:
   0:   55                      push   %rbp
   1:   48 89 e5                mov    %rsp,%rbp
   4:   c7 45 fc 26 10 00 00    movl   $0x1026,0xfffffffffffffffc(%rbp)
   b:   8b 55 fc                mov    0xfffffffffffffffc(%rbp),%edx
   e:   89 d0                   mov    %edx,%eax
  10:   01 c0                   add    %eax,%eax
  12:   01 d0                   add    %edx,%eax
  14:   89 45 fc                mov    %eax,0xfffffffffffffffc(%rbp)
  17:   b8 00 00 00 00          mov    $0x0,%eax
  1c:   c9                      leaveq
  1d:   c3                      retq
```

What has the compiler done to the statement `c *= 3`?

2.4.1*c* Given the result in 2.4.1*b*, what would you suggest has faster absolute speed, integer addition or integer multiplication?

2.4.1*d* The following code is used by many graphic video game engines (*e.g.*, the Quake engine) to approximate $\frac{1}{\sqrt{x}}$ for a floating-point number *x* with an error of less than 0.1 %:

```
// These are "magic" numbers
float const MULTIPLIER = 1.000876311302185f;
int const INV_SQRT_NEWTON_N = 1597463175;

float inv_sqrt_multiplier( float x ) {
        float const mx = 0.5f*MULTIPLIER*x;
        int xi = *reinterpret_cast<int *>( &x );
        xi = INV_SQRT_NEWTON_N - (xi >> 1);
        x = *reinterpret_cast<float *>( &xi );
        return x*(1.5f*MULTIPLIER - mx*x*x);
}
```

What is the run-time of this function?

2.4.2*a* In own words, explain why we can make the statement that these two statements run in Θ(1) time.

```
{
    ++index;
    prev_modulus = modulus;
    modulus = next_modulus;
    next_modulus = modulus_table[index];
}
```

```
{
    Type tmp = array[i];
    array[i] = array[posn];
    array[posn] = tmp;
}
```

2.4.3*a* What are the run times of the following for loops?

```
for ( int i = 0; i < n; ++i ) {
    sum += i;
}

for ( int i = 0; i < n*n; ++i ) {
    sum += i;
}

for ( int i = 0; i <= n*m; ++i ) {
    sum += i;
}

for ( int i = n - 1; i >= 0; --i ) {
    sum += i;
}

for ( int i = 0; (i <= n) && (j <= m); ++i, ++j ) {
    sum += i;
}

for ( int i = 0; (i <= n) || (j <= m); ++i, ++j ) {
    sum += i;
}
```

2.4.3.1*a* What are the run times of each of the following for-loops?

```
for ( int i = 0; i < n; ++i ) {
    for ( int j = 0; j < m; ++j ) {
        sum += i + j;
    }
}

for ( int i = 0; i < n; ++i ) {
    for ( int j = 0; j < n*n; ++j ) {
        sum += i + j;
    }
}

for ( int i = 0; i < n1; ++i ) {
    for ( int j = 0; j < n2; ++j ) {
        for ( int k = 0; k < n3; ++k ) {
            sum += i + j + k;
        }
    }
}
```

2.4.3.2*a* What are the run times of each of the following for-loops?

```
for ( int i = 1; i <= n; i *= 2 ) {
    sum += i;
}

for ( int i = n; i >= 1; i /= 2 ) {
    sum += i;
}
```

2.4.3.2*b* Why do the following two for-loops have the same run time?

```
for ( int i = 1; i <= n; i *= 2 ) {
    sum += i;
}

for ( int i = 1; i <= n*n; i *= 2 ) {
    sum += i;
}
```

2.4.3.3*a* The run time of this code that iterates through a singly linked list

```
for ( Single_node<int> *ptr = head(); ptr != 0; ptr = ptr->next() ) {
    sum += ptr->retrieve();
}
```

would be said to run in Θ(*n*) time where *n* is defined as the number of entries in the linked list. In the Standard Template Library, it is an absolute requirement that an integrator runs in Θ(*n*) time:

```
for ( std::container<int>::iterator *itr = v.begin(); itr != v.end(); ++itr ) {
    sum += *itr;
}
```

In the STL, it is not required that the following code runs in Θ(*n*) time:

```
for ( int k = 0; k != v.size(); ++k ) {
    sum += v.at( k );
}
```

where `at( int )` acceses the k^{th} entry. What does this say about the run-time requirement for `++itr` versus the run-time requirement for `at( int )`?

2.4.3.4*a* Given the discussion in Question 2.4.3.3*a*, what are the run-times of the following?

```
for ( Single_node<int> *ptr = head(); ptr != 0; ptr = ptr->next() ) {
    sum += ptr->retrieve();
}
```

and

```
bool positive = true;

for ( Single_node<int> *ptr = head(); ptr != 0; ptr = ptr->next() ) {
    if ( ptr->retrieve() < 0 ) {
        positive = false;
        break;
    }
}
```

?

2.4.3.5 The three nexted for-loops:

```
for ( int i = 0; i < n; ++i ) {
    for ( int j = 0; j < i; ++j ) {
        sum += i + j;
    }
}
```

```
for ( int i = 0; i < n; ++i ) {
    for ( int j = 0; j < i*i; ++j ) {
        sum += i + j;
    }
}
```

```
for ( int i = 0; i < n*n; ++i ) {
    for ( int j = 0; j < i; ++j ) {
        sum += i + j;
    }
}
```

require us to calculate

$$\sum_{i=0}^{n-1}\sum_{j=0}^{i-1} 1 = \sum_{i=0}^{n-1} i \qquad \sum_{i=0}^{n-1}\sum_{j=0}^{i^2-1} 1 = \sum_{i=0}^{n-1} i^2 \qquad \sum_{i=0}^{n^2-1}\sum_{j=0}^{i-1} 1 = \sum_{i=0}^{n^2-1} i$$

In each case, we can always use the formula

$$\sum_{i=0}^{n-1} i^k \approx \frac{(n-1)^{k+1}}{k+1} = \Theta\left(n^{k+1}\right).$$

Thus, the run times of these three loops are:

$$\sum_{i=0}^{n-1} i = \Theta\left(n^2\right) \qquad \sum_{i=0}^{n-1} i^2 = \Theta\left(n^3\right) \qquad \sum_{i=0}^{n^2-1} i = \Theta\left(\left(n^2\right)^2\right) = \Theta\left(n^4\right)$$

Calculate the run times for the following for-loops:

```
for ( int i = 0; i < n*n*n; ++i ) {
    for ( int j = 0; j < i; ++j ) {
        sum += i + j;
    }
}

for ( int i = 0; i < n; ++i ) {
    for ( int j = 0; j < i*i*i; ++j ) {
        sum += i + j;
    }
}

for ( int i = 0; i < n*n; ++i ) {
    for ( int j = 0; j < i*i; ++j ) {
        sum += i + j;
    }
}

for ( int i = 0; i < n*m; ++i ) {
    for ( int j = 0; j < i; ++j ) {
        sum += i + j;
    }
}

for ( int i = 0; i < n; ++i ) {
    for ( int j = 0; j < i*m; ++j ) {
        sum += i + j;
    }
}
```

2.4.4*a* Describe the run time of this function where `M` is a matrix class and `v` and `u` are vector classes.

```
// Calculate v = M*u
if ( M.is_diagonal() ) {
    for ( int i = 0; i < size(); ++i ) {
        v.array[i] = M.array[i][i]*u.array[i];
    }
} else {
    for ( int i = 0; i < size(); ++i ) {
        for ( int j = 0; j < size(); ++j ) {
            v.array[i] += M.array[i][j] * u.array[j];
        }
    }
}
```

2.4.4*b* Describe the run time of this function where `M` is a matrix class and `v` and `u` are vector classes.

```
// Calculate v = M*u
if ( M.is_upper_triangular() ) {
    for ( int i = 0; i < size(); ++i ) {
        for ( int j = i; j < size(); ++j ) {
            v.array[i] += M.array[i][j] * u.array[j];
        }
    }
} else {
    for ( int i = 0; i < size(); ++i ) {
        for ( int j = 0; j < size(); ++j ) {
            v.array[i] += M.array[i][j] * u.array[j];
        }
    }
}
```

2.4.4*c* Describe the run time of this function where `M` is a matrix class and `v` and `u` are vector classes.

```
// Calculate v = M*u
if ( M.is_tridiagonal() ) {
    for ( int i = 0; i < size(); ++i ) {
        for ( int j = std::max(0, i - 1); j <= std::min(i + 1, size() - 1); ++j ) {
            v.array[i] += M.array[i][j] * u.array[j];
        }
    }
} else {
    for ( int i = 0; i < size(); ++i ) {
        for ( int j = 0; j < size(); ++j ) {
            v.array[i] += M.array[i][j] * u.array[j];
        }
    }
}
```

2.4.5*a* What is the run time of the following function statement?

```
{
    double tmp[n];

    for ( int i = 0; i < n; ++i ) {
        tmp[i] = 0;
    }

    for ( int i = 0; i < n; ++i ) {
        for ( int j = 0; j < n; ++j ) {
            tmp[i] = matrix[i][j]*vector[j];
        }
    }

    for ( int i = 0; i < n; ++i ) {
        vector[i] = tmp[i];
    }
}
```

2.4.5*b* What is the run time of the following function that finds the Hamming distance between two strings `str1` and `str2` of lengths *m* and *n*, respectively?

```
int haming_distance( std::string const &str1, std::string const &str2 ) {
    if ( str1.length() > str2.length() ) {
        return distance( str2, str1 );
    }

    int dist = 0;


    for ( int i = 0; i < str1.length(); ++i ) {
        if ( str1[i] != str2[i] ) {
            ++dist;
        }
    }

    return dist + str2.length() – str1.length();
}
```

2.4.5*c* What is the run time of the following function that finds the Levenshtein distance between two strings `str1` and `str2` of lengths *m* and *n*, respectively?

```
int distance( std::string const &s, std::string const &t ) {
    int table[s.length() + 1][t.length() + 1];

    for ( int i = 0; i <= s.length(); ++i ) {
        table[i][0] = i;
    }

    for ( int j = 1; j <= t.length(); ++j ) {
        table[0][j] = j;
    }

    for ( int i = 1; i <= s.length(); ++i ) {
        for ( int j = 1; j <= t.length(); ++j ) {
            int subs = ( s[i - 1] != t[j - 1] ) ? 1 : 0;

            table[i][j] = std::min( table[i - 1][  j  ] + 1, std::min(
                table[i - 1][j - 1] + subs, table[  i  ][j - 1] + 1
            );
        }
    }

    return table[s.length()][t.length()];
}
```

2.4.5*d* What is the run time of this function that calculates backward substitution to solve a system **Av** = **b** where **A** is a sparse matrix and if `row_index[i] <= row_index[i + 1]` for all values of `i` from `0` to `N - 1` and where `row_index[0] == 0` and `row_index[N - 1] == M`.

```
template<int N>
Vector<N> backsubs ( Matrix<N, N> &A, Vector<N> &b, bool identity ) {
       Vector<N> v;

       for ( int i = N - 1; i >= 0; --i ) {
              v.array[i] = b.array[i];

              for ( int j = A.row_index[i]; j < A.row_index[i + 1]; ++j ) {
                     v.array[i] -= A.off_diagonal[j]*v.array[A.column_index[j]];
              }

              if ( !identity ) {
                     v.array[i] /= A.diagonal[i];
              }
       }

       return v;
}
```

2.4.6 For the questions in this section, assume that the run time of a function `fn_name` is $T_{\text{fn\_name}}$ and if the run time depends on a parameter n (e.g., the number of objects being stored in a container), then we will denote the run time as $T_{\text{fn\_name}}(n)$.

2.4.6*a* For this function, assume that the run time of all functions depends on the number of elements, n, that are stored in this disjoint set:

```
void Disjoint_sets::set_union( int elem1, int elem2 ) {
    // Map each to its representative
    elem1 = find( elem1 );
    elem2 = find( elem2 );

    if ( elem1 == elem2 ) {
        return;
    }

    --num_disjoint_sets;

    // Graft the shorter tree onto the root of the larger tree
    if ( tree_height[elem1] >= tree_height[elem2] ) {
        parent[elem2] = elem1;

        // If equal, graft 'elem2' onto 'elem1'
        if ( tree_height[elem1] == tree_height[elem2] ) {
            ++( tree_height[elem1] );
            // Update the maximum height if necessary
            max_height = std::max( max_height, tree_height[elem1] );
        }
    } else {
        parent[elem1] = elem2;
    }
}
```

2.4.6*b* What is the run time of this function that uses the method of steepest descent to solve a system **Av** = **b** where **A** is an $N \times N$ matrix and where:

1. Any matrix-vector multiplication is $\Theta(N^2)$, and
2. Any vector-vector addition, vector-transpose, vector-norm, inner product, or scalar multiplication is $\Theta(N)$.

```
template <int N>
Vector<N> steepest_descent( Matrix<N, N> &A, Vector<N> &b, int K, double epsilon ) {
    Vector<N> x = diagonal_solve( A, b );
    Vector<N> r = b - A*x;
    Vector<N> p = A*r;

    for ( int i = 0; i < K; ++i ) {
        double alpha = (transpose(r)*r) / (transpose(p)*r);
        x = x + alpha*r;

        if ( norm( alpha*r ) < epsilon ) {
            return x;
        }

        r = r - alpha*p;
        p = A*r;
    }

    throw non_convergence( K );
}
```

2.4.6*c* Continuing Question 2.4.6*b*, upon further investigation, you find that the implementation of the diagonal solver is as follows:

```
template<int N>
Vector<N, COLUMN> diagonal_solve( Matrix<N, N> const &A, Vector<N, COLUMN> const &b ) {
    Vector<N, COLUMN> v;

    for ( int i = 0; i < N; ++i ) {
        v.array[i] = b.array[i]/A.array[i][i];
    }

    return v;
}
```

What is the run time of the `steepest_descent` function now?

2.4.6*d* Suppose that most entries in a matrix are zero (this is usually the case in large systems, as most a non-zero values in a matrix indicates direct interaction between two components in the system). In this case, matrix vector multiplication is reduced to Θ($N + m$) where m is the number of non-zero entries. If you are additionally told that $m = \Theta(N)$, what is the run time of the `steepest_descent` function now?

2.4.6*e* What is the run time of this function?

```
double f( double *array1, int n1, double *array2, int n2 ) {
    if ( n2 > n1 ) {
        return f( array2, n2, array1, n1 );
    }

    double result = 0.0;

    for ( int i = 0; i < n1; ++i ) {
        result += array1[i]*array2[i];
    }

    return result;
}
```

2.4.7*a* Write down the run time of the following recursive function as a recurrence relation:

```
int f1( int n ) {
    if ( n <= 1 ) {
        return 1;
    }

    return f( n – 1 );
}
```

2.4.7*b* Write down the run time of the following recursive functions as a recurrence relation:

```
int f2( int n ) {
    if ( n <= 1 ) {
        return 1;
    }

    return f( n – 1 ) + f( n – 1 );
}

int f3( int n ) {
    if ( n <= 1 ) {
        return 1;
    }

    return 2*f( n – 1 );
}
```

Will both of these functions run in the same time?

2.4.7*c* Write down the run times of these functions as a recurrence relation:

```
double f4( int n ) {
    if ( n == 0 ) {
        return 0;
    }

    int result = 0;

    for ( int i = 0; i < n; ++i ) {
        result += i;
    }

    return result + f4( n - 1);
}


double f5( int n ) {
    if ( n == 0 ) {
        return 0;
    }

    int result = 0;

    for ( int i = 0; i < n; ++i ) {
        result += f(i);
    }

    return result;
}
```

2.4.8*a* The following function attempts to generate random numbers that appear to be normally distributed with a standard normal function. Every second time this function is called, it calculates two random numbers and stores one while returning the other. The next time the function is called, it simply returns the stored value. In finding two random numbers, it continues finding random points (x, y) in the unit square [–1, 1] × [–1, 1] (with an area of 4) and continues only after it has chosen a point that also falls within the unit circle (with an area of π) .

```
double Marsaglia_polar::randn() {
    // n is a static member variable incremented
    // each time this function is called, starting with n = 1;
    ++n;


    // If 'n' is odd, return a stored value.

    if ( n & 1 ) {
        return stored_number;
    }

    double u1, u2, s;

    do {
        x = 2.0*drand48() - 1.0;
        y = 2.0*drand48() - 1.0;
```

```
                s = x*x + y*y;
            } while ( s >= 1 );

            double c = std::sqrt( -2.0*log(s)/s );

            stored_number = y*c;

            return x*c;
        }
```

What are the best-case and worst-case run times?

2.4.8*b* In question 2.4.8*a*, the worst-case scenario is unlikely to happen. Instead, we can consider following argument:

1. There is a $\pi/4 \approx 11/14$ chance that the first point (x, y) will be in the circle—we are done, otherwise we must keep going,
2. In the $(1 - \pi/4) \approx 3/14$ chance that we didn't get a point in the circle, we try again. In this case, we still have a $\pi/4$ of finishing, but if we don't, we must continue,
3. We've now found two numbers outside the circle—a probability of $(1 - \pi/4)^2 \approx 9/196$, and this time, again, we have a $\pi/4$ chance of getting a point in the circle, *etc*.

Thus, we must calculate the sum

$$\sum_{k=1}^{\infty} k\left(\frac{\pi}{4}\right)\left(1-\frac{\pi}{4}\right)^{k-1}$$

This includes:

1. k: the number of times we must generate a random point in $[-1, 1] \times [-1, 1]$,
2. $\pi/4$: the probability that we are successful on the k^{th} attempt, and
3. $(1 - \pi/4)^{k-1}$: the probability that we were unsuccessful on the previous $k - 1$ attempts.

If we ask Maple for this answer, we get:

```
> sum( k*(Pi/4)*(1 - Pi/4)^(k - 1), k = 1..infinity );
```

$$\frac{4}{\pi}$$

where $\frac{4}{\pi} \approx 1.27$. What is the average run time of the function `Marsaglia_polar::randn()`?