To *enumerate* is to list off, one by one.

We consider:

- Counting
- 1-D Indeterminate Enumeration
- 1-D Determinate Enumeration
- 2-D Enumerations

and these applications:

- Sieve of Eratosthenes
- Ramanujan Cubes
- Enumerations of Rational Numbers
- Magic Squares
Counting:

```c
int k = 1;
while (true) k++;
```

1-origin children

```c
int k = 0;
while (true) k++;
```

0-origin older children

```c
int k = start;
while (true) k++;
```

start-origin sophisticated children
Counting:

<table>
<thead>
<tr>
<th>int k = 1;</th>
<th>Linguistic Confusions</th>
</tr>
</thead>
<tbody>
<tr>
<td>while ( true ) k++;</td>
<td>First value enumerated</td>
</tr>
<tr>
<td>1-origin</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>int k = 0;</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>while ( true ) k++;</td>
<td>0-origin</td>
</tr>
</tbody>
</table>

| int k = start;                                           |                        |
|---------------------------------------------------------| start-origin           | start                 |
| while ( true ) k++;                                      | k-start                |                        |

Off-by-one errors, and their ilk

- Number of integers in a range from first to last, inclusive: last-first+1
- Index of last integer in a range of N integers starting at 0: N-1
Counting:

```
int k = 1;
while ( true ) k++;
```

```
int k = 0;
while ( true ) k++;
```

```
int k = start;
while ( true ) k++;
```

Children learn the concept of infinity from counting. Indeed, these loops run forever, but not because there is no maximum `int`. Rather, because after $2^{31} - 1$, the next `int` is $-2^{31}$. This is called *arithmetic overflow*.

From there, counting proceeds “up” to -1, and then around again.
Children learn the concept of infinity from counting. Indeed, these loops run forever, but not because there is no maximum `int`. Rather, because after $2^{31} - 1$, the next `int` is $-2^{31}$. This is called *arithmetic overflow*.

From there, counting proceeds “up” to -1, and then around again unless `condition` becomes `false` first.
1-D Indeterminate Enumeration:

/* Enumerate from start until !condition. */
int k = start;
while ( condition ) k++;
1-D Indeterminate Enumeration:

/* Enumerate from start until !condition, but no further than maximum. */
int k = start;
    while ( k<=maximum && condition ) k++;
if ( k>maximum ) /* condition was true for all k in [start..maximum]. */
else /* k is smallest in [start..maximum] for which condition is false. */
1-D Determinate Enumeration:

/* Do whatever n times. */
int k = 0;
while ( k<n ) {
    /* whatever */
    k++;
}

or

/* Do whatever n times. */
for ( int k=0; k<n; k++ )
    /* whatever */
1-D Determinate Enumeration: Don’t terminate a determinate enumeration prematurely.

/* Do whatever n times. */
for (int k=0; k<n; k++) {
    /* whatever */
    if (condition ) k = n;  // Don’t do this.
}

Rather, do this:

k = 0;
while ( k<n && !condition ) {
    /* whatever */
    k++;
}

N.B. The two versions are not exactly equivalent.
Application of 1-D Determinate Enumeration: Print all primes up to n.

Consider each integer from 2 through n.
Application of 1-D Determinate Enumeration: Print all primes up to n.

Consider each integer from 2 through n.

If it is not marked out, it is prime: Print it, and mark out all its multiples.
Application of 1-D Determinate Enumeration: Print all primes up to n.

Consider each integer from 2 through \( n \).

If it is not marked out, it is prime: Print it, and mark out all its multiples.
Application of 1-D Determinate Enumeration: Print all primes up to n.

Consider each integer from 2 through n.

If it is not marked out, it is prime: Print it, and mark out all its multiples.
Application of 1-D Determinate Enumeration: Print all primes up to \( n \).

Consider each integer from 2 through \( n \).

If it is not marked out, it is prime: Print it, and mark out all its multiples.
Application of 1-D Determinate Enumeration: Print all primes up to n.

Consider each integer from 2 through n.

If it is not marked out, it is prime: Print it, and mark out all its multiples.
Application of 1-D Determinate Enumeration: Print all primes up to n.

Consider each integer from 2 through n.

If it is not marked out, it is prime: Print it, and mark out all its multiples.
Application of 1-D Determinate Enumeration: Print all primes up to n.

/* Print primes up to n. */
   /* Initialize sieve to all prime. */
   /* Print each prime in sieve, and cross out its multiples. */
Application of 1-D Determinate Enumeration: Print all primes up to n.

/* Print primes up to n. */
/* Initialize sieve to all prime */
    for (int j=2; j<=n; j++) _____________
/* Print each prime in sieve, and cross out its multiples. */
Application of 1-D Determinate Enumeration: Print all primes up to n.

/* Print primes up to n. */
/* Initialize sieve to all prime. */
for (int j=2; j<n; j++) _____________
/* Print each prime in sieve, and cross out its multiples. */
for (int j=2; j<n; j++) _____________
Application of 1-D Determinate Enumeration: Print all primes up to n.

/* Print primes up to n. */
/* Initialize sieve to all prime. */
for (int j=2; j<=n; j++)
/* Print each prime in sieve, and cross out its multiples. */
for (int j=2; j<=n; j++)
  if ( _______ ) {
    System.out.println(j);
    for (int k=2*j; k<=n; k=k+j) _________
  }
Application of 1-D Determinate Enumeration: Print all primes up to n.

/* Print primes up to n. */
boolean prime[] = new boolean[____]; // prime[k] true iff k is prime.
/* Initialize sieve to all prime. */
for (int j=2; j<=n; j++) prime[j] = true;
/* Print each prime in sieve, and cross out its multiples. */
for (int j=2; j<=n; j++)
   if ( prime[j] ) {
      System.out.println(j);
      for (int k=2*j; k<=n; k=k+j) prime[k] = false;
   }
Application of 1-D Determinate Enumeration: Print all primes up to n.

/* Print primes up to n. */
boolean prime[] = new boolean[n+1];
/* Initialize sieve to all prime. */
for (int j=2; j<=n; j++) prime[j] = true;
/* Print each prime in sieve, and cross out its multiples. */
for (int j=2; j<=n; j++)
  if ( prime[j] ) {
    System.out.println(j);
    for (int k=2*j; k<=n; k=k+j) prime[k] = false;
  }
Row-major order, determinate enumeration

/* Enumerate \langle r,c \rangle in [0..height-1][0..width-1] in row-major order. */
for (int r=0; r<height; r++)
    for (int c=0; c<width; c++)
        /* whatever */

or

/* Enumerate \langle r,c \rangle in [1..height][1..width] in row-major order. */
for (int r=1; r<=height; r++)
    for (int c=1; c<=width; c++)
        /* whatever */
Column-major order, determinate enumeration

/* Enumerate \( (r,c) \) in \([0..\text{height}-1][0..\text{width}-1]\) in column-major order. */

\[
\text{for } (\text{int } c=0; c<\text{width}; c++) \\
\quad \text{for } (\text{int } r=0; r<\text{height}; r++) \\
\quad \text{/* whatever */}
\]
Row-major order, indeterminate enumeration

/* Enumerate \( (r,c) \) in \([0..\text{height}-1][0..\text{width}-1]\) in row-major order until condition, and do whatever for each. */

```c
int r = 0; int c = 0;
while ( r<\text{height} \&\& !\text{condition} ) {
    /* whatever */
    if ( c<\text{width}-1 ) c++;
    else { c = 0; r++; }
}
if ( r==\text{height} ) /* fail */ else /* succeed */
```

**INVARIANT:** width height enumerated
Triangular order

/* Enumerate \langle r,c \rangle in a closed lower-triangular region of \[0..\text{size}-1\][0..\text{size}-1] in row-major order.*/
for (int r=0; r<\text{size}; r++)
    for (int c=0; c<=r; c++)
        /* whatever */

/* Enumerate \langle r,c \rangle in an open lower-triangular region of \[0..\text{size}-1\][0..\text{size}-1] in row-major order.*/
for (int r=1; r<\text{size}; r++)
    for (int c=0; c<r; c++)
        /* whatever */

Think of the enumeration as all ways of choosing two distinct values from [0..\text{size}-1].
Diagonal order

/* Unbounded enumeration of ordered ⟨r,c⟩ starting at ⟨0,0⟩ until condition. */
int d = 0;
while ( !condition ) {
    int r = d;
    for (int c=0; c<=d; c++) {
        /* whatever */
        r--;
    }
    d++;
}

Think of d as the index of the diagonal.
Row and column subscripts wrap around, i.e., after the right-most column comes the left-most column, and after the bottom-most row comes the top-most row.
/* n-by-n toroidal diagonal-order enumeration in "magical order". */
int r = 0; int c = n/2;
for (int d=0; d<n; d++) {
  for (int k=0; k<n; k++) {
    /* whatever */
    r = (r+n-1)%n; c = (c+1)%n; // up 1 and right 1.
  }
  r = (r+2)%n; c = (c+n-1)%n; // down 2 and left 1.
}
Application of triangular-order enumeration: We wish to confirm Ramanujan’s claim that 1729 is the smallest number that is the sum of two positive cubes in two different ways.

• The integer part of the cube root of 1729 is 12. Thus, we only need to consider the cubes of positive integers that are no larger than 12.
• Let \( r^3 \) and \( c^3 \) be the two cubes.
Application of triangular-order enumeration:

/* Confirm Ramanujan’s claim that 1729 is the smallest number that is the sum of two positive cubes in two different ways. */
/* Record the values of $r^3+c^3$ that arise for all sets \{r,c\} of distinct positive integers that are no larger than 12. */
/* Confirm that 1729 is the smallest integer that arose twice. */
Application of triangular-order enumeration:

/* Confirm Ramanujan’s claim that 1729 is the smallest number that is the sum of two positive cubes in two different ways. */
/* Record the values of r^3+c^3 that arise for all sets \{r,c\} of distinct positive integers that are no larger than 12. */
    for (int r=2; r<13; r++)
        for (int c=1; c<r; c++)
            /* Keep track of having seen r^3+c^3. */
    /* Confirm that 1729 is the smallest integer that arose twice. */

We complete this code in Chapter 12.
**Application of diagonal-order enumeration:** We wish to enumerate positive rational numbers.

Start with an enumeration of positive fractions.

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1/1</td>
<td>1/2</td>
<td>1/3</td>
<td>1/4</td>
<td>1/5</td>
</tr>
<tr>
<td>2/1</td>
<td>2/2</td>
<td>2/3</td>
<td>2/4</td>
<td>2/5</td>
</tr>
<tr>
<td>3/1</td>
<td>3/2</td>
<td>3/3</td>
<td>3/4</td>
<td>3/5</td>
</tr>
<tr>
<td>4/1</td>
<td>4/2</td>
<td>4/3</td>
<td>4/4</td>
<td>4/5</td>
</tr>
<tr>
<td>5/1</td>
<td>5/2</td>
<td>5/3</td>
<td>5/4</td>
<td>5/5</td>
</tr>
</tbody>
</table>

There are, of course, an infinite number of numerators and denominators, so a row-major-order or column-major-order enumeration won’t do.
Application of diagonal-order enumeration: We wish to enumerate positive rational numbers.

Start with an enumeration of positive fractions.

There are, of course, an infinite number of numerators and denominators, so a row-major-order or column-major-order enumeration won’t do.

A diagonal-order enumeration allows both the numerators and denominators to grow without bound.
Application of diagonal-order enumeration: We wish to enumerate positive rational numbers.

Start with an enumeration of positive fractions.

```java
/* Output positive fractions. */
int d = 0;
while (true) {
    int r = d;
    for (int c=0; c<=d; c++) {
        System.out.println((r+1) + "/" + (c+1));
        r--;
    }
    d++;
}
```
Application of diagonal-order enumeration: We wish to enumerate positive rational numbers.

Start with an enumeration of positive fractions.

```
/* Output positive fractions. */
int d = 0;
while (true) {
    int r = d;
    for (int c=0; c<=d; c++) {
        System.out.println((r+1) + "/" + (c+1));
        r--;
    }
    d++;
}
```

However, this lists each rational more than once.
Application of diagonal-order enumeration: We wish to enumerate positive rational numbers.

```java
/* Output positive fractions. */
int d = 0;
while (true) {
    int r = d;
    for (int c=0; c<=d; c++) {
        System.out.println( (r+1) + "/" + (c+1) );
        r--;
    }
    d++;
}
```

To avoid duplicate listings, we can:
Application of diagonal-order enumeration: We wish to enumerate positive rational numbers.

```java
/* Output positive rationals. */
int d = 0;
/* set reduced = { }; */
while (true) {
    int r = d;
    for (int c=0; c<=d; c++) {
        /* Let z be the reduced form of the fraction (r+1)/(c+1). */
        int g = gcd(r, c+1);
        /* rational z = ((r+1)/g,(c+1)/g); */
        if (/* z is not an element of reduced */) {
            System.out.println( (r+1) + "/" + (c+1) );
            /* reduced = reduced U {z}; */
        }
        r--;
    }
    d++;
}
```

To avoid duplicate listings, we can:
- Maintain the set of reduced fractions already listed.
- Only list a fraction if its reduced form is not in the set.
Application of diagonal-order enumeration: We wish to enumerate positive rational numbers.

/* Output positive rationals. */
int d = 0;
/* set reduced = { }; */
while (true) {
    int r = d;
    for (int c=0; c<=d; c++) {
        /* Let z be the reduced form of the fraction (r+1)/(c+1). */
        int g = gcd(r, c+1);
        /* rational z = ((r+1)/g,(c+1)/g); */
        if ( /* z is not an element of reduced */
            System.out.println( (r+1) + "/" + (c+1) );
            /* reduced = reduced U {z}; */
        }
        r--;
    }
    d++;
}
Application of diagonal-order enumeration: We wish to enumerate positive rational numbers.

/* Output positive rationals. */
int d = 0;
/* set reduced = { }; */
while (true) {
    int r = d;
    for (int c=0; c<=d; c++) {
        /* Let z be the reduced form of the fraction (r+1)/(c+1). */
        int g = gcd(r, c+1);
        /* rational z = ((r+1)/g,(c+1)/g); */
        if (/* z is not an element of reduced */) {
            System.out.println((r+1) + "/" + (c+1));
            /* reduced = reduced ∪ {z}; */
        }
        r--;
    }
    d++;
}
Application of toroidal diagonal-order enumeration: $n$-by-$n$ Magic Squares, for odd $n$.

A square grid of numbers is a Magic Square if all rows, columns, and both diagonals sum to the same value.
Application of toroidal diagonal-order enumeration: \( n \)-by-\( n \) Magic Squares, for odd \( n \).

To make an \( n \)-by-\( n \) Magic Square, for odd \( n \), start with a 1 in the middle of the top row.
Application of toroidal diagonal-order enumeration: $n$-by-$n$ Magic Squares, for odd $n$.

To make an $n$-by-$n$ Magic Square, for odd $n$, start with a 1 in the middle of the top row, and count up as you proceed diagonally up and to the right (on the surface of a torus).
Application of toroidal diagonal-order enumeration: \( n \)-by-\( n \) Magic Squares, for odd \( n \).

To make an \( n \)-by-\( n \) Magic Square, for odd \( n \), start with a 1 in the middle of the top row, and count up as you proceed diagonally up and to the right (on the surface of a torus).
Application of toroidal diagonal-order enumeration: \( n \)-by-\( n \) Magic Squares, for odd \( n \).

To make an \( n \)-by-\( n \) Magic Square, for odd \( n \), start with a 1 in the middle of the top row, and count up as you proceed diagonally up and to the right (on the surface of a torus).
Application of toroidal diagonal-order enumeration: $n$-by-$n$ Magic Squares, for odd $n$.

To make an $n$-by-$n$ Magic Square, for odd $n$, start with a 1 in the middle of the top row, and count up as you proceed diagonally up and to the right (on the surface of a torus).
Application of toroidal diagonal-order enumeration: $n$-by-$n$ Magic Squares, for odd $n$.

To make an $n$-by-$n$ Magic Square, for odd $n$, start with a 1 in the middle of the top row, and count up as you proceed diagonally up and to the right (on the surface of a torus). When you encounter an already-filled cell, move to the row below (in the same column).
Application of toroidal diagonal-order enumeration: $n$-by-$n$ Magic Squares, for odd $n$.

To make an $n$-by-$n$ Magic Square, for odd $n$, start with a 1 in the middle of the top row, and count up as you proceed diagonally up and to the right (on the surface of a torus). When you encounter an already-filled cell, move to the row below (in the same column).
Application of toroidal diagonal-order enumeration: $n$-by-$n$ Magic Squares, for odd $n$.

To make an $n$-by-$n$ Magic Square, for odd $n$, start with a 1 in the middle of the top row, and count up as you proceed diagonally up and to the right (on the surface of a torus). When you encounter an already-filled cell, move to the row below (in the same column).
Application of toroidal diagonal-order enumeration: $n$-by-$n$ Magic Squares, for odd $n$.

To make an $n$-by-$n$ Magic Square, for odd $n$, start with a 1 in the middle of the top row, and count up as you proceed diagonally up and to the right (on the surface of a torus). When you encounter an already-filled cell, move to the row below (in the same column).
Application of toroidal diagonal-order enumeration: $n$-by-$n$ Magic Squares, for odd $n$.

To make an $n$-by-$n$ Magic Square, for odd $n$, start with a 1 in the middle of the top row, and count up as you proceed diagonally up and to the right (on the surface of a torus). When you encounter an already-filled cell, move to the row below (in the same column).
Application of toroidal diagonal-order enumeration: $n$-by-$n$ Magic Squares, for odd $n$.

To make an $n$-by-$n$ Magic Square, for odd $n$, start with a 1 in the middle of the top row, and count up as you proceed diagonally up and to the right (on the surface of a torus). When you encounter an already-filled cell, move to the row below (in the same column).
Application of toroidal diagonal-order enumeration: \( n \)-by-\( n \) Magic Squares, for odd \( n \).

To make an \( n \)-by-\( n \) Magic Square, for odd \( n \), start with a 1 in the middle of the top row, and count up as you proceed diagonally up and to the right (on the surface of a torus). When you encounter an already-filled cell, move to the row below (in the same column).
Application of toroidal diagonal-order enumeration: \( n \)-by-\( n \) Magic Squares, for odd \( n \).

To make an \( n \)-by-\( n \) Magic Square, for odd \( n \), start with a 1 in the middle of the top row, and count up as you proceed diagonally up and to the right (on the surface of a torus). When you encounter an already-filled cell, move to the row below (in the same column).
Application of toroidal diagonal-order enumeration: \(n\times n\) Magic Squares, for odd \(n\).

To make an \(n\times n\) Magic Square, for odd \(n\), start with a 1 in the middle of the top row, and count up as you proceed diagonally up and to the right (on the surface of a torus). When you encounter an already-filled cell, move to the row below (in the same column).
Application of toroidal diagonal-order enumeration: $n$-by-$n$ Magic Squares, for odd $n$.

/* Let $M$ be an $N$-by-$N$ Magic Square, for odd $N \geq 1$. */
int $M[][] = new$ int$[N][N]$; // Initialized to zeros.
int $r = 0$; int $c = N/2$;
for (int $k=1; k<=N*N; k++$) {
    $M[r][c] = k$;
    /* Advance $(r,c)$ in toroidal diagonal order. */
}
Application of toroidal diagonal-order enumeration: $n$-by-$n$ Magic Squares, for odd $n$.

/* Let $M$ be an $N$-by-$N$ Magic Square, for odd $N \geq 1$. */

int $M[][] = \text{new int}[N][N]$; // Initialized to zeros.
int $r = 0$; int $c = N/2$;
for (int $k=1$; $k<=N*N$; $k++$) {
    $M[r][c] = k$;
    /* Advance $(r,c)$ in toroidal diagonal order. */
    if ( $M[(r+N-1)\%N][(c+1)\%N]!=$0 ) $r = (r+1)\%N$;
    else { $r = (r+N-1)\%N$; $c = (c+1)\%N$; }
}