Principled Programming

Introduction to Coding in Any Imperative Language

Tim Teitelbaum

Emeritus Professor
Department of Computer Science
Cornell University

Enumeration Patterns

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:

		Practitioners
`k = 1` `while True: k += 1`	1-origin	children
`k = 0` `while True: k += 1`	0-origin	older children
`k = start` `while True: k += 1`	*start*-origin	sophisticated children

Counting:

```
k = 1
while True: k += 1
```

```
k = 0
while True: k += 1
```

```
k = start
while True: k += 1
```

Linguistic Confusions

	First value enumerated	Number of increments
1-origin	1	k-1
0-origin	0	k
start-origin	*start*	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:

Python

```
k = 1
while True: k += 1
```

```
k = 0
while True: k += 1
```

```
k = start
while True: k += 1
```

Children learn the concept of infinity from counting. Indeed, these loops run forever, or more precisely until all available computer memory has been exhausted.

Counting:

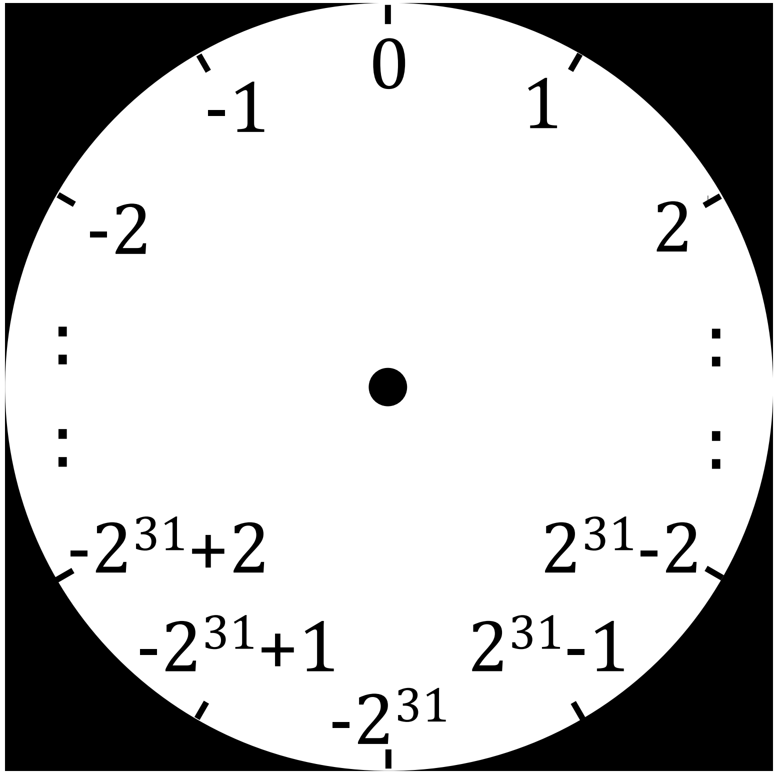
Java

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

```
k = 0;
while (true) k = k+1;
```

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

N.B. Although there is no maximum **int** in Python, other languages do have a maximum (e.g., in Java the maximum is 2^{31}-1). Exceeding the maximum (e.g., by adding 1 to it) is called *arithmetic overflow.* Curiously, the next **int** after 2^{31}-1 is -2^{31}. So, these loops actually run forever, not because the value of k gets arbitrarily large, but because after the arithmetic overflow, counting proceeds "up" to -1, and then around again.



1-D Indeterminate Enumeration:

```
# Enumerate from start until not(condition).
k = start
while condition: k += 1
```

1-D Indeterminate Enumeration:

```
# Enumerate from start until not(condition).
k = start
while condition: k += 1

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.
k = 0
while k < n:
    #.whatever.
    k += 1
```

or

```
# Do whatever n times.
for k in range(0, n):
    #.whatever.
```

1-D Determinate Enumeration: Don't terminate a determinate enumeration prematurely.

```
# Do whatever n times, but stop on condition.
for k in range(0, n):
    #.whatever.
    if condition: k = n    # Don't do this.
```

Rather, do this:

```
# Do whatever n times, but stop on condition.
k = 0
while k < n and not(condition):
     # whatever.
     k += 1
```

N.B. The two versions are not exactly equivalent.

Application of 1-D Determinate Enumeration: Print all primes up to n.

2 3 4 5 6 7 8 9 10 11 12 13 14 15

Consider each integer from 2 through n.

Sieve of Eratosthenes

Application of 1-D Determinate Enumeration: Print all primes up to n.

2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	3	**4**	5	**6**	7	**8**	9	**10**	11	**12**	13	**14**	15

Consider each integer from 2 through n.

If it is not marked out, it is prime: Print it, and mark out all its multiples.

Sieve of Eratosthenes

Application of 1-D Determinate Enumeration: Print all primes up to n.

2	3	4	5	6	7	8	9	10	11	12	13	14	15
(2)	3	4	5	6	7	8	9	10	11	12	13	14	15
2	(3)	4	5	**6**	7	8	**9**	10	11	**12**	13	14	**15**

Consider each integer from 2 through n.

If it is not marked out, it is prime: Print it, and mark out all its multiples.

Sieve of Eratosthenes

Application of 1-D Determinate Enumeration: Print all primes up to n.

2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	3	4	5	6	7	8	9	**10**	11	12	13	14	**15**

Consider each integer from 2 through n.

If it is not marked out, it is prime: Print it, and mark out all its multiples.

Sieve of Eratosthenes

Application of 1-D Determinate Enumeration: Print all primes up to n.

2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	3	4	5	6	7	8	9	10	11	12	13	**14**	15

Consider each integer from 2 through n.

If it is not marked out, it is prime: Print it, and mark out all its multiples.

Sieve of Eratosthenes

Application of 1-D Determinate Enumeration: Print all primes up to n.

2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	3	4	5	6	7	8	9	10	11	12	13	14	15

Consider each integer from 2 through n.

If it is not marked out, it is prime: Print it, and mark out all its multiples.

Sieve of Eratosthenes

Application of 1-D Determinate Enumeration: Print all primes up to n.

2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	3	4	5	6	7	8	9	10	11	12	13	14	15

Consider each integer from 2 through n.

If it is not marked out, it is prime: Print it, and mark out all its multiples.

Sieve of Eratosthenes

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 j in range(2, n + 1): _____________

#.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 j in range(2, n + 1): _____________

# Print each prime in sieve, and cross out its multiples.
for j in range(2, n + 1): _____________
```

Application of 1-D Determinate Enumeration: Print all primes up to n.

```
# Print primes up to n.
# ---------------------
# Initialize sieve to all prime.
for j in range(2, n + 1): _____________

# Print each prime in sieve, and cross out its multiples.
for j in range(2, n + 1):
    if ________ :
        print(j)
        for k in range(2 * j, n + 1, j): _____________
```

Optional third argument of range provides increment.

Sieve of Eratosthenes

Application of 1-D Determinate Enumeration: Print all primes up to n.

```
# Print primes up to n.
# ---------------------
# Initialize sieve to all prime.
prime: list[bool] = [True] * ____  # For each k, prime[k] is True iff k is prime.

# Print each prime in sieve, and cross out its multiples.
for j in range(2, n + 1):
    if prime[j]:
        print(j)
        for k in range(2 * j, n + 1, j): prime[k] = False
```

Sieve of Eratosthenes

Application of 1-D Determinate Enumeration: Print all primes up to n.

```
# Print primes up to n.
# ---------------------
# Initialize sieve to all prime.
prime: list[bool] = [True] * (n+1) # For each k, prime[k] is True iff k is prime.

# Print each prime in sieve, and cross out its multiples.
for j in range(2, n + 1):
    if prime[j]:
        print(j)
        for k in range(2 * j, n + 1, j): prime[k] = False
```

2-D Enumerations

Row-major order, determinate enumeration

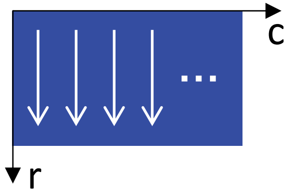
0-origin, e.g., for subscripts

```
# Enumerate ⟨r,c⟩ in [0..height-1][0..width-1] in row-major order & do whatever.
for r in range(0, height):
    for c in range(0, width):
        # whatever.
```

or

1-origin, e.g., for itemization

```
# Enumerate ⟨r,c⟩ in [1..height][1..width] in row-major order & do whatever.
for r in range(1, height + 1):
    for c in range(1, width + 1):
        # whatever.
```

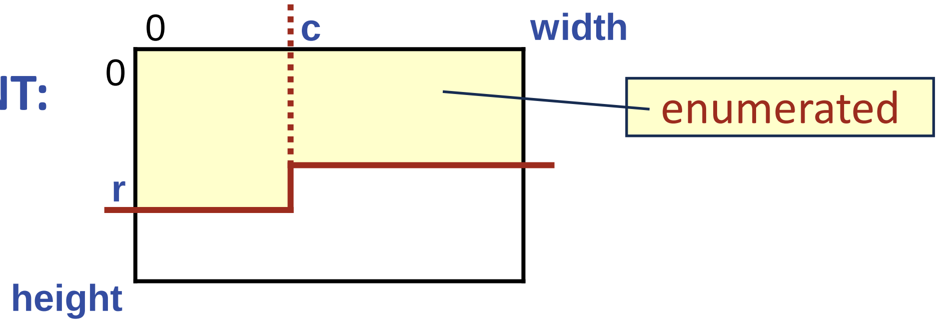


Column-major order, determinate enumeration

0-origin, e.g., for subscripts

```
# Enumerate ⟨r,c⟩ in [0..height-1][0..width-1] in column-major order & do whatever.
for c in range(0, width):
    for r in range(0, height):
        # whatever.
```

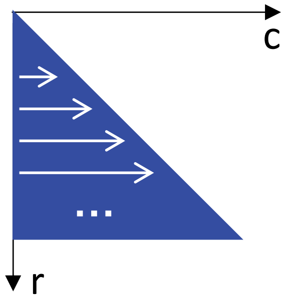
INVARIANT:



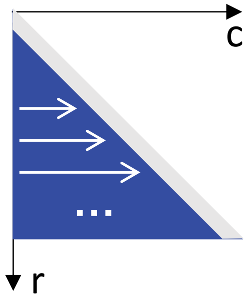
Row-major order, indeterminate enumeration

```
# Enumerate ⟨r,c⟩ in [0..height-1][0..width-1] in row-major order until
#   condition, and do whatever for each.
r = 0;  c = 0
while r < height and not(condition):
    #.whatever.
    if c < width - 1: c += 1   # Not the end of a row; go to next column.
    else:                      # The end of a row; go to start of next row.
        c = 0; r += 1
if r == height: #.fail
else: #.succeed
```

Triangular order

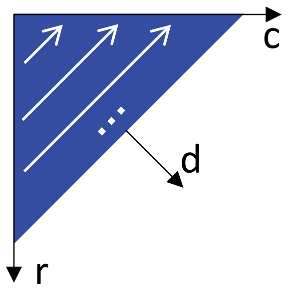



```
# Enumerate ⟨r,c⟩ in a closed lower-triangular region of [0..size-1]
#   [0..size-1] in row-major order, and do whatever for each.
for r in range(0, size):
    for c in range(0, r + 1):
        #.whatever.
```




```
# Enumerate ⟨r,c⟩ in a closed lower-triangular region of [0..size-1]
#   [0..size-1] in row-major order, and do whatever for each.
for r in range(0, size):
    for c in range(0, r):
        #.whatever.
```

Think of the enumeration as all ways of choosing two distinct values from [0..size-1].

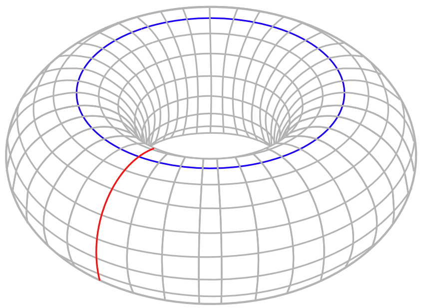


Diagonal order

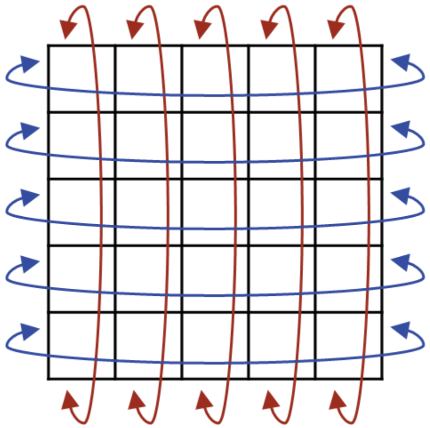
```
# Unbounded enumeration of ordered ⟨r,c⟩ starting at ⟨0,0⟩ until condition.
d = 0
while not(condition):
    r = d
    for c in range(0, d + 1):
        # whatever.
        r -= 1
    d += 1
```

Think of d as the index of the diagonal.

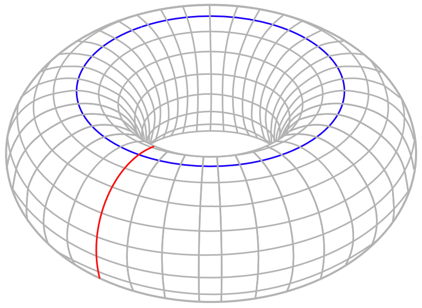
2-D Enumerations



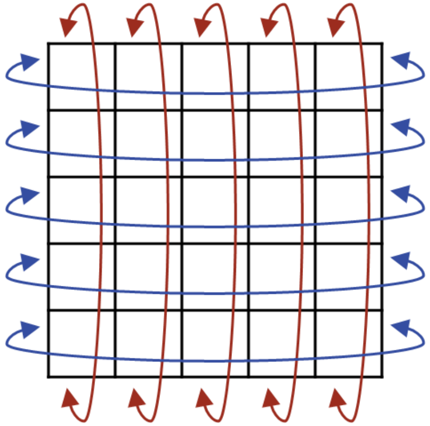
2-D array on a torus

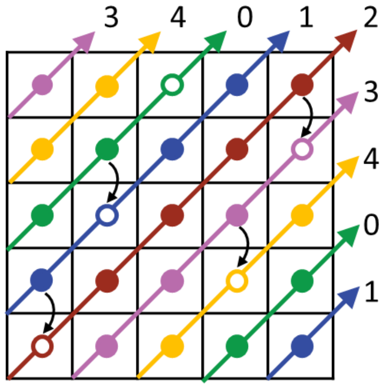


Row and column subscripts wrap around, i.e., after the right-most column comes the left-most column, and after the bottom-most row comes the top-most row.



Toroidal diagonal order






```
# n-by-n toroidal diagonal-order enumeration in "magical order".
r = 0;  c = n // 2
for d in range(0, n):
  for k in range(0, n):
     #.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 r in range(2, 13):
    for c in range(1, r):
        #.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.

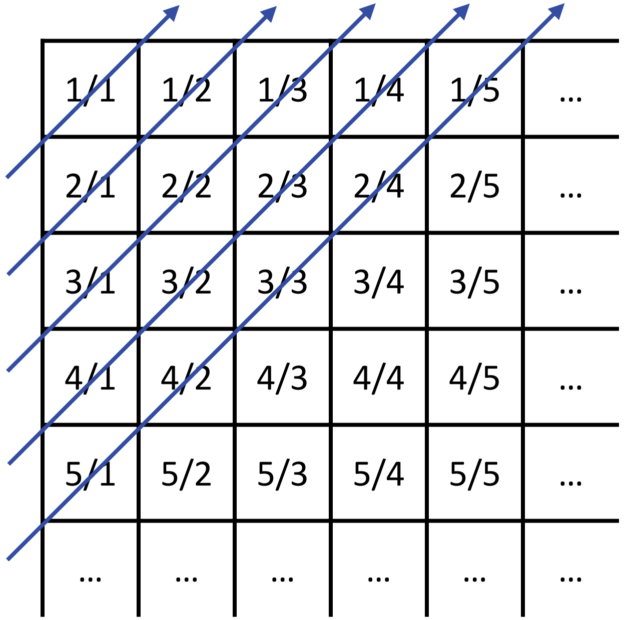
Start with an enumeration of positive fractions.

1/1	1/2	1/3	1/4	1/5	…
2/1	2/2	2/3	2/4	2/5	…
3/1	3/2	3/3	3/4	3/5	…
4/1	4/2	4/3	4/4	4/5	…
5/1	5/2	5/3	5/4	5/5	…
…	…	…	…	…	…

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.

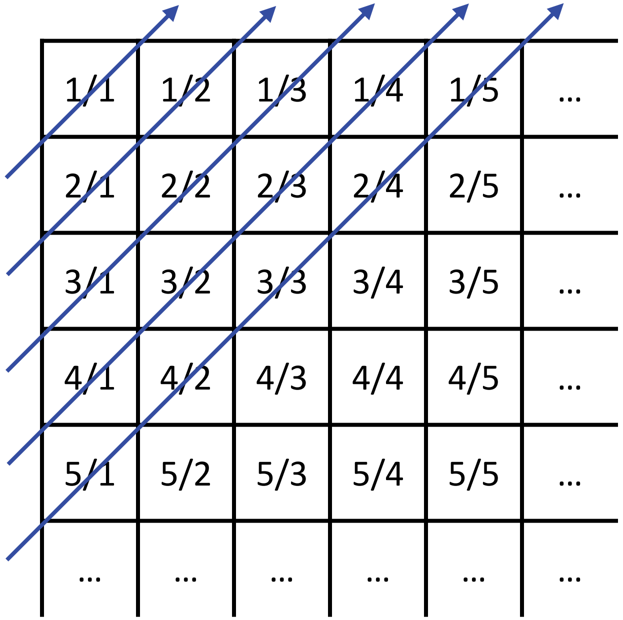


There are, of course, an infinite number of numerators and denominators, so a row-major-order or column-major-order enumeration won't do.

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.

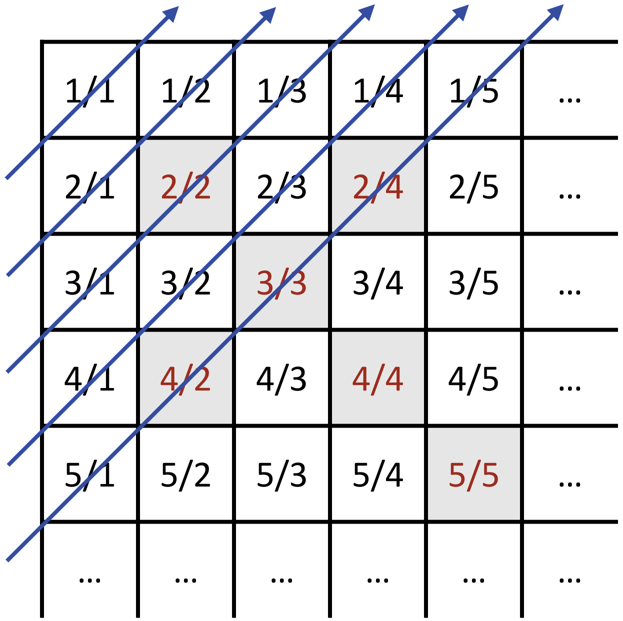
Start with an enumeration of positive fractions.




```
# Output positive fractions.
d = 0
while True:
    r = d
    for c in range(0, d + 1):
        print((r + 1), "/", (c + 1))
        r -= 1
    d += 1
```

Application of diagonal-order enumeration: We wish to enumerate positive rational numbers.

Start with an enumeration of positive fractions.




```
# Output positive fractions, including those equivalent
#   as rationals.
d = 0
while True:
    r = d
    for c in range(0, d + 1):
        print((r + 1), "/", (c + 1))
        r -= 1
    d += 1
```

However, this lists each rational more than once.

Application of diagonal-order enumeration: We wish to enumerate positive rational numbers.

```
# Output positive fractions, including those equivalent as rationals.
d = 0
while True:
    r = d
    for c in range(0, d + 1):
        print((r + 1), "/", (c + 1))
        r -= 1
    d += 1
```

To avoid duplicate listings, we can:

Application of diagonal-order enumeration: We wish to enumerate positive rational numbers.

```
# Output reduced positive fractions, i.e., each positive rationals once.
d = 0
#.reduced = { }
while True:
    r = d
    for c in range(0, d + 1):
        # Let z be the reduced form of the fraction (r+1)/(c+1).
            g: int = gcd(r, c + 1)
            #.z = ⟨(r+1)/g, (c+1)/g⟩
        if z-is-not-an-element-of-reduced:
            print((r + 1), "/", (c + 1))
            #.reduced = reduced ∪ {z}
        r -= 1
    d += 1
```

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 reduced positive fractions, i.e., each positive rationals once.
d = 0
#.reduced = { }
while True:
    r = d
    for c in range(0, d + 1):
        # Let z be the reduced form of the fraction (r+1)/(c+1).
            g: int = gcd(r, c + 1)
            #.z = ⟨(r+1)/g, (c+1)/g⟩
        if z-is-not-an-element-of-reduced:
            print((r + 1), "/", (c + 1))
            #.reduced = reduced ∪ {z}
        r -= 1
    d += 1
```

Introduces two key ideas:

- User-defined types, e.g., `rational` types (for z)
- User-defined types that are collections, e.g., `set-of-rational` (for reduced).

Application of diagonal-order enumeration: We wish to enumerate positive rational numbers.

```
# Output reduced positive fractions, i.e., each positive rationals once.
d = 0
#.reduced = { }
while True:
    r = d
    for c in range(0, d + 1):
        # Let z be the reduced form of the fraction (r+1)/(c+1).
            g: int = gcd(r, c + 1)
            #.z = ⟨(r+1)/g, (c+1)/g⟩
        if z-is-not-an-element-of-reduced:
            print((r + 1), "/", (c + 1))
            #.reduced = reduced ∪ {z}
        r -= 1
    d += 1
```

There are better ways to have proceeded, which we will ignore for pedagogical purposes until Chapter 18.

Application of toroidal diagonal-order enumeration: *n*-by-*n* Magic Squares, for odd *n*.

			15
8	1	6	15
3	5	7	15
4	9	2	15
15	15	15	15

A square grid of numbers is a Magic Square if all rows, columns, and both diagonals sum to the same value.

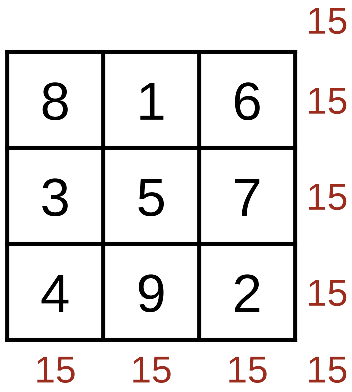
Application of toroidal diagonal-order enumeration: *n*-by-*n* Magic Squares, for odd *n*.

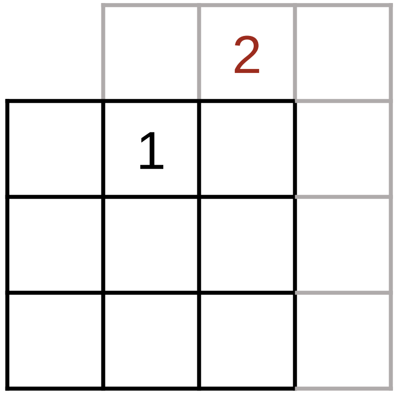
			15
8	1	6	15
3	5	7	15
4	9	2	15
15	15	15	15

	1	

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*.

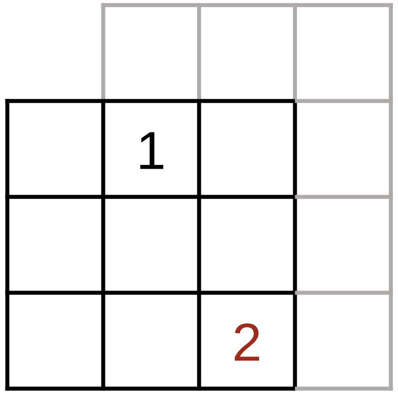




To make an *n*-by-*n* Magic Square, for odd *n*, start with a 1 in the middle of the top row, and count up as you proceed diagonally up and to the right (on the surface of a torus).

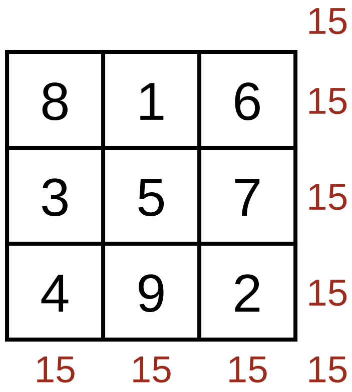
Application of toroidal diagonal-order enumeration: *n*-by-*n* Magic Squares, for odd *n*.

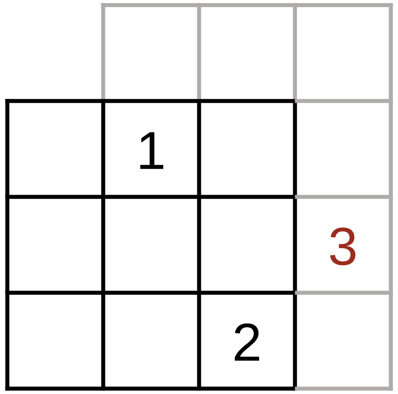
			15
8	1	6	15
3	5	7	15
4	9	2	15
15	15	15	15



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*.

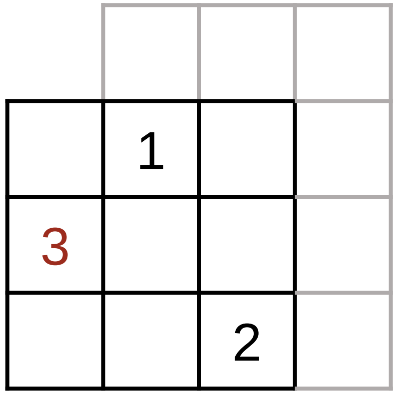




To make an *n*-by-*n* Magic Square, for odd *n*, start with a 1 in the middle of the top row, and count up as you proceed diagonally up and to the right (on the surface of a torus).

Application of toroidal diagonal-order enumeration: *n*-by-*n* Magic Squares, for odd *n*.

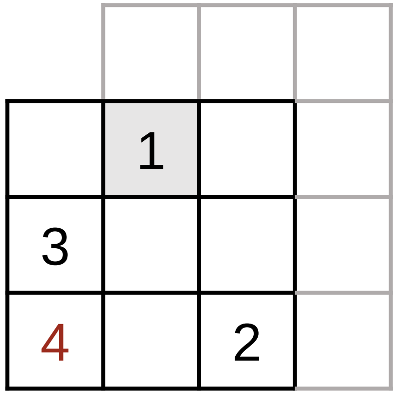
			15
8	1	6	15
3	5	7	15
4	9	2	15
15	15	15	15



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*.

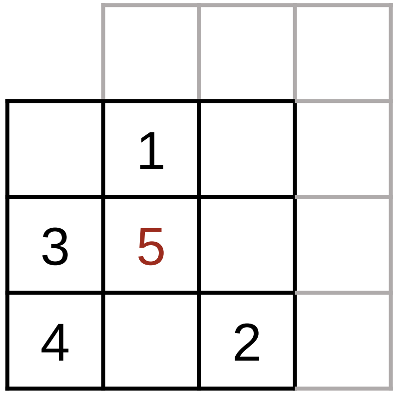
			15
8	1	6	15
3	5	7	15
4	9	2	15
15	15	15	15



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*.

			15
8	1	6	15
3	5	7	15
4	9	2	15
15	15	15	15



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*.

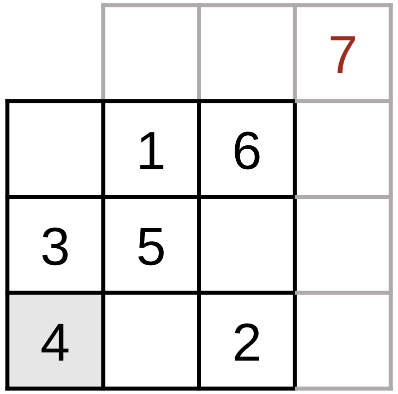
			15
8	1	6	15
3	5	7	15
4	9	2	15
15	15	15	15

	1	6
3	5	
4		2

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*.

			15
8	1	6	15
3	5	7	15
4	9	2	15
15	15	15	15



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*.

			15
8	1	6	15
3	5	7	15
4	9	2	15
15	15	15	15

	1	6	
3	5	7	
4		2	

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*.

			15
8	1	6	15
3	5	7	15
4	9	2	15
15	15	15	15

	1	6	8
3	5	7	
4		2	

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).

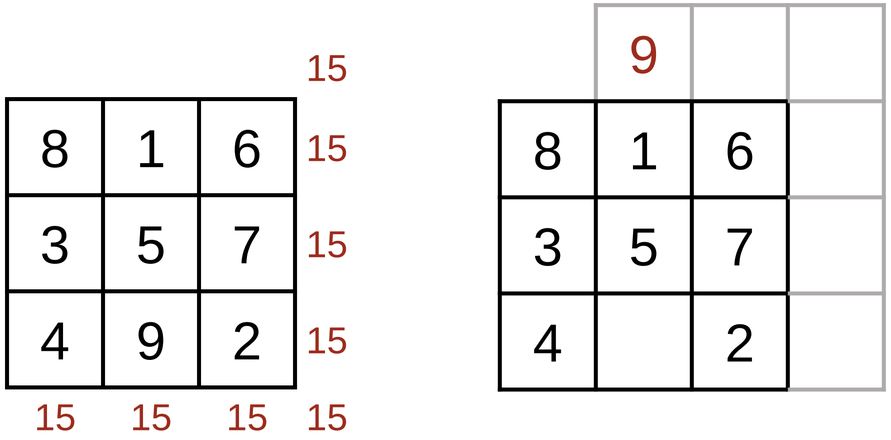
Application of toroidal diagonal-order enumeration: *n*-by-*n* Magic Squares, for odd *n*.

			15
8	1	6	15
3	5	7	15
4	9	2	15
15	15	15	15

8	1	6	
3	5	7	
4		2	

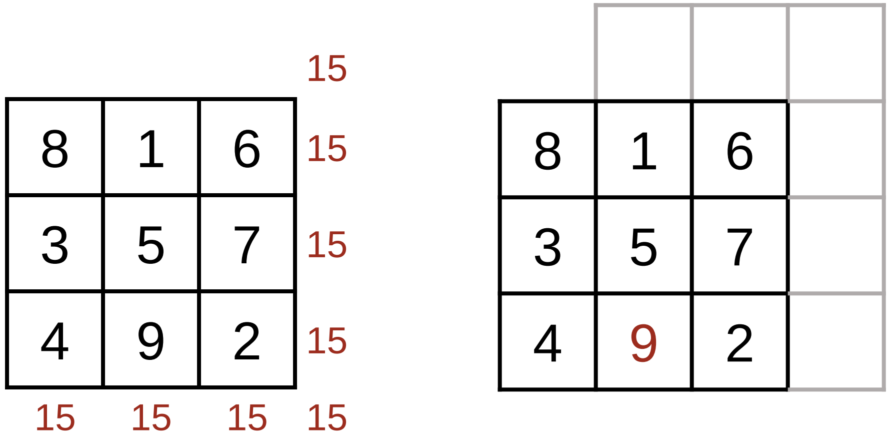
To make an *n*-by-*n* Magic Square, for odd *n*, start with a 1 in the middle of the top row, and count up as you proceed diagonally up and to the right (on the surface of a torus). When you encounter an already-filled cell, move to the row below (in the same column).

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*.

```
# Let M be an N-by-N Magic Square, for odd N≥1.
M: list[list[int]] = [[0 for _ in range(N)] for _ in range(N)]
r = 0; c = N // 2
for k in range(1, (N * N) + 1):
    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≥1.
M: list[list[int]] = [[0 for _ in range(N)] for _ in range(N)]
r = 0; c = N // 2
for k in range(1, (N * N) + 1):
    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
```