```python
def hamming_distance(x, y):
    difference = x ^ y
    count = 0
    while difference:
        # this removes ones from right to left (least to most significant)
        difference &= difference - 1
        count += 1
    return count


# Wegner method - O(b) where b is number of set bits
def hamming_weight(x):
    if x < 0:
        return None

    count = 0
    while x:
        x &= x - 1
        count += 1

    return count


def pop_count(i):
    i -= ((i >> 1) & 0x55555555)
    i = (i & 0x33333333) + ((i >> 2) & 0x33333333)
    return (((i + (i >> 4) & 0xF0F0F0F) * 0x1010101) & 0xffffffff) >> 24


def is_bit_set(x, pos):
    return (x & (1 << pos)) != 0


def is_even(x):
    return x & 1 == 0


def is_power_of_2(x):
    return (x & x - 1) == 0


# Returns the n bits in bitfield starting at position pos
def get_bits(x, pos, n):
    return (x >> (pos + 1 - n)) & ~(~0 << n)


def set_bit(x, position):
    return x | (1 << position)


def clear_bit(x, position):
    return x & ~(1 << position)


def toggle_bit(x, position):
    return x ^ (1 << position)


def rotate_left(x, n):
    print(bin(-n & 31))
    return (x << n) | (x >> (-n & 31))  # assumes a 32 bit word size


def add(a, b):
    while a:
        c = b & a
        b ^= a
        c <<= 1
        a = c
    return b


def get_sign(x):
    return -(x < 0)
```

```
# has odd number of bits
def has_parity(x):
    parity = False

    while x:
        parity = not parity
        x &= (x - 1)

    return parity


def has_parity_parallel(x):
    x ^= x >> 16
    x ^= x >> 8
    x ^= x >> 4
    x &= 0xf
    return (0x6996 >> x) & 1


def next_power_of_2(x):
    x -= 1
    x |= x >> 1
    x |= x >> 2
    x |= x >> 4
    x |= x >> 8
    x |= x >> 16
    x += 1

    return x


#
# def modulo(x, y):
#     return x & ((1 << y) - 1)

# high_bit_mask will be all 1s if negative or all 0 if positive
# for negative will NOT(x) - (-1) = NOT(x) + 1, which is two's complement conversion
from negative to positive
def myabs(x):
    high_bit_mask = x >> 31
    return (x ^ high_bit_mask) - high_bit_mask


def swap_ints(a, b):
    a ^= b
    b ^= a
    a ^= b
    return [a, b]


def main():
    for x in range(0, 20):
        print(x, bin(x), hex(x))

    print("Hamming distance: 1010111100 and 1001010101: ", hamming_distance(0b1010111100
, 0b1001010101))

    print("Bit 0: 1011 1100 set?", is_bit_set(0b10111100, 0))
    print("Bit 1: 1011 1100 set?", is_bit_set(0b10111100, 1))
    print("Bit 2: 1011 1100 set?", is_bit_set(0b10111100, 2))
    print("Bit 3: 1011 1100 set?", is_bit_set(0b10111100, 3))
    print("Bit 4: 1011 1100 set?", is_bit_set(0b10111100, 4))
    print("Bit 5: 1011 1100 set?", is_bit_set(0b10111100, 5))
    print("Bit 6: 1011 1100 set?", is_bit_set(0b10111100, 6))
    print("Bit 7: 1011 1100 set?", is_bit_set(0b10111100, 7))

    print("Hamming weight: 1010111100:", hamming_weight(0b1010111100))
    print("Hamming weight: 1001000001:", hamming_weight(0b1001000001))
    print("Hamming weight: 0001000000:", hamming_weight(0b0001000000))
    print("Hamming weight: 0000000000:", hamming_weight(0b0000000000))
    print("Hamming weight: 11111111:", hamming_weight(0b11111111))

    print("PopCount: " + str(bin(1231424)) + ":", pop_count(1231424))
    print("PopCount: " + str(bin(123)) + ":", pop_count(123))
    print("PopCount: " + str(bin(8568)) + ":", pop_count(8568))
    print("PopCount: " + str(bin(1)) + ":", pop_count(1))
```

```
    print("PopCount: " + str(bin(-1)) + ":", pop_count(-1))
    print("PopCount: " + str(bin(4)) + ":", pop_count(4))
    print("PopCount: " + str(bin(903523125)) + ":", pop_count(903523125))

    for x in range(0, 5):
        print(str(x) + " is even?", is_even(x))

    for x in range(0, 17):
        print(str(x) + " is power of 2?", is_power_of_2(x))

    print("Get 3 bits starting at position 5 in 1010111100", bin(get_bits(0b1010111100,
5, 3)))
    print("Get 3 bits starting at position 8 in 1010111100", bin(get_bits(0b1010111100,
9, 3)))

    print("Set bit 4 to 0: 01010010", bin(clear_bit(0b01010010, 4)))
    print("Set bit 0 to 1: 01010010", bin(set_bit(0b01010010, 0)))
    print("Toggle bit 1: 01010010", bin(toggle_bit(0b01010010, 1)))
    print("Toggle bit 3: 01010010", bin(toggle_bit(0b01010010, 3)))

    print("Rotate left 5 positions: 01001000100101011001010010011011:",
          bin(rotate_left(0b01001000100101011001010010011011, 5)))

    print("3 + 5 = ", add(3, 5))
    print("33 + 51 = ", add(33, 51))
    print("40 + 90 = ", add(40, 90))
    print("45 + 15 = ", add(45, 15))

    print("Sign of 45", get_sign(45))
    print("Sign of 0", get_sign(0))
    print("Sign of -1", get_sign(-1))
    print("Sign of -23", get_sign(-23))

    for x in range(0, 12):
        print("has parity: " + str(x), has_parity(x), bin(x))

    for x in range(0, 7):
        print("has parity (parallel): " + str(x), has_parity_parallel(x), bin(x))

    print("Next power of 2 after 3", next_power_of_2(3))
    print("Next power of 2 after 4", next_power_of_2(4))
    print("Next power of 2 after 7", next_power_of_2(7))
    print("Next power of 2 after 12", next_power_of_2(12))
    print("Next power of 2 after 45", next_power_of_2(45))

    # print("3 mod 5", modulo(3, 5))
    # print("4 mod 2", modulo(4, 2))
    # print("5 mod 2", modulo(5, 2))
    # print("10 mod 3", modulo(10, 3))
    # print("10 mod 4", modulo(10, 4))
    # print("10 mod 5", modulo(10, 5))

    assert myabs(-1) == 1
    assert myabs(-134) == 134
    assert myabs(99) == 99
    assert myabs(0) == 0
    assert myabs(16) == 16

    print("swap 1, 3", swap_ints(1, 3))
    print("swap 213, 14", swap_ints(213, 14))
    print("swap 872, 992", swap_ints(872, 992))


if __name__ == "__main__":
    main()
```

```python
class Empty(Exception):
    """Exception for requesting data from an empty collection"""
    pass
```

```python
def factors(n):  # generator that computes factors
    k = 1
    while k * k < n:  # while k < sqrt(n)
        if n % k == 0:
            yield k
            yield n // k
        k += 1
    if k * k == n:  # special case if n is perfect square
        yield k
```

```python
def hash_string(word, m):
    """
    31 was set by Kernighan & Ritchie, and creates a good distribution on ASCII
    """
    hash = 0
    for c in word:
        hash = (hash * 31 + ord(c))

    return abs(hash % m)


def hash_integer(number, a, b, p, m):
    """
    p is a prime greater than m, the number of slots your hash table will support.
    a and b are randomly chosen integers modulo p with a != 0
    """
    mod_p = (a * number + b) % p

    return abs(mod_p % m)
```

```python
def multiply(a, b):
    val = 0

    while b > 0:
        val += a
        b -= 1

    return val


def add(a, b):
    sum = 0

    while sum < b:
        sum += 1
        a += 1

    return a


def divide(a, d):
    if d == 0:
        raise ValueError('Cannot divide by 0.')

    q = 0
    r = a

    while r >= d:
        r -= d
        q += 1

    print("{} div by {} = {} remainder {}".format(a, d, q, r))


def power(base, exp):
    val = 1

    for _ in range(exp):
        val *= base

    return val


def power2(base, exp):
    if exp == 0:
        return 1
    else:
        partial = power(x, exp // 2)  # rely on truncated division
        result = partial * partial
        if exp % 2 == 1:  # if n odd, include extra factor of x
            result *= base

        return result


def factorial(n):
    val = 1

    for num in range(n, 0, -1):
        val *= num

    return val


def fib(n):
    a, b = 1, 1

    for _ in range(1, n):
        a, b = b, a + b

    return a


def fibonacci():
    a, b = 0, 1
    while True:
```

```
        yield a
        a, b = b, a + b


def gcd(a, b):
    while a:
        b, a = a, b % a

    return b


def lcm(a, b):
    return (a * b) / gcd(a, b)
```

```python
class UnionFind:
    def __init__(self, n):
        self._id = list(range(n))
        self._sz = [1] * n

    def _root(self, i):
        j = i
        while j != self._id[j]:
            # compressing path
            self._id[j] = self._id[self._id[j]]
            j = self._id[j]
        return j

    def find(self, p, q):
        return self._root(p) == self._root(q)

    def union(self, p, q):
        i = self._root(p)
        j = self._root(q)
        # merge smaller into larger
        if self._sz[i] < self._sz[j]:
            self._id[i] = j
            self._sz[j] += self._sz[i]
        else:
            self._id[j] = i
            self._sz[i] += self._sz[j]

    def get_roots(self):
        for root in self._id:
            yield root


def main():
    uf = UnionFind(10)
    for (p, q) in [(3, 4), (4, 9), (8, 0), (2, 3)]:
        uf.union(p, q)

    assert uf.get_roots() == [0, 1, 3, 3, 3, 5, 6, 7, 0, 3]

    for (p, q) in [(5, 6), (5, 9), (7, 3)]:
        uf.union(p, q)

    assert uf.get_roots() == [0, 1, 3, 3, 3, 3, 5, 3, 0, 3]

    for (p, q) in [(4, 8), (6, 1)]:
        uf.union(p, q)

    assert uf.get_roots() == [8, 3, 3, 3, 3, 3, 3, 3, 3, 3]
    assert uf.find(0, 1) is True
    assert uf.get_roots() == [3, 3, 3, 3, 3, 3, 3, 3, 3, 3]
```

```
class Empty(Exception):
    """Exception for requesting data from an empty collection"""
    pass


class ArrayQueue:
    """FIFO queue implementation using a Python list as underlying storage."""
    DEFAULT_CAPACITY = 10  # moderate capacity for all new queues

    def __init__(self):
        self._data = [None] * ArrayQueue.DEFAULT_CAPACITY
        self._size = 0
        self._front = 0

    def __len__(self):
        return self._size

    def is_empty(self):
        return self._size == 0

    def first(self):
        """Return (but do not remove) the element at the front of the queue.

        Raise Empty exception if the queue is empty.
        """
        if self.is_empty():
            raise Empty('Queue is empty')
        return self._data[self._front]

    def dequeue(self):
        """Remove and return the first element of the queue (i.e., FIFO).

        Raise Empty exception if the queue is empty.
        """
        if self.is_empty():
            raise Empty('Queue is empty')
        answer = self._data[self._front]
        self._data[self._front] = None  # help garbage collection
        self._front = (self._front + 1) % len(self._data)
        self._size -= 1
        return answer

    def enqueue(self, e):
        """Add an element to the back of queue."""
        if self._size == len(self._data):
            self._resize(2 * len(self.data))  # double the array size
        avail = (self._front + self._size) % len(self._data)
        self._data[avail] = e
        self._size += 1

    def _resize(self, cap):  # we assume cap >= len(self)
        """Resize to a new list of capacity >= len(self)."""
        old = self._data  # keep track of existing list
        self._data = [None] * cap  # allocate list with new capacity
        walk = self._front
        for k in range(self._size):  # only consider existing elements
            self._data[k] = old[walk]  # intentionally shift indices
            walk = (1 + walk) % len(old)  # use old size as modulus
        self._front = 0  # front has been realigned
```

```python
"""Basic example of an adapter class to provide a stack interface to Python's list."""


class Empty(Exception):
    """Exception for requesting data from an empty collection"""
    pass


class ArrayStack:
    """LIFO Stack implementation using a Python list as underlying storage."""

    def __init__(self):
        self._data = []  # nonpublic list instance

    def __len__(self):
        return len(self._data)

    def is_empty(self):
        return len(self._data) == 0

    def push(self, e):
        """Add element e to the top of the stack."""
        self._data.append(e)  # new item stored at end of list

    def top(self):
        """Return (but do not remove) the element at the top of the stack.

        Raise Empty exception if the stack is empty.
        """
        if self.is_empty():
            raise Empty('Stack is empty')
        return self._data[-1]  # the last item in the list

    def pop(self):
        """Remove and return the element from the top of the stack (i.e., LIFO).

        Raise Empty exception if the stack is empty.
        """
        if self.is_empty():
            raise Empty('Stack is empty')
        return self._data.pop()  # remove last item from list


if __name__ == '__main__':
    S = ArrayStack()        # contents: [ ]
    S.push(5)               # contents: [5]
    S.push(3)               # contents: [5, 3]
    print(len(S))           # contents: [5, 3];    outputs 2
    print(S.pop())          # contents: [5];       outputs 3
    print(S.is_empty())     # contents: [5];       outputs False
    print(S.pop())          # contents: [ ];       outputs 5
    print(S.is_empty())     # contents: [ ];       outputs True
    S.push(7)               # contents: [7]
    S.push(9)               # contents: [7, 9]
    print(S.top())          # contents: [7, 9];    outputs 9
    S.push(4)               # contents: [7, 9, 4]
    print(len(S))           # contents: [7, 9, 4]; outputs 3
    print(S.pop())          # contents: [7, 9];    outputs 4
    S.push(6)               # contents: [7, 9, 6]
    S.push(8)               # contents: [7, 9, 6, 8]
    print(S.pop())          # contents: [7, 9, 6]; outputs 8
```

```
from collections import namedtuple
import random

Point = namedtuple('Point', 'x y')


class ConvexHull(object):
    _points = []
    _hull_points = []

    def __init__(self):
        pass

    def add(self, point):
        self._points.append(point)

    def _get_orientation(self, origin, p1, p2):
        '''
        Returns the orientation of the Point p1 with regards to Point p2 using origin.
        Negative if p1 is clockwise of p2.
        '''
        difference = (
            ((p2.x - origin.x) * (p1.y - origin.y))
            - ((p1.x - origin.x) * (p2.y - origin.y))
        )

        return difference

    def compute_hull(self):
        points = self._points

        # get leftmost point
        start = points[0]
        min_x = start.x
        for p in points[1:]:
            if p.x < min_x:
                min_x = p.x
                start = p

        point = start
        self._hull_points.append(start)

        far_point = None
        while far_point is not start:

            # get the first point (initial max) to use to compare with others
            p1 = None
            for p in points:
                if p is point:
                    continue
                else:
                    p1 = p
                    break

            far_point = p1

            for p2 in points:
                # ensure we aren't comparing to self or pivot point
                if p2 is point or p2 is p1:
                    continue
                else:
                    direction = self._get_orientation(point, far_point, p2)
                    if direction > 0:
                        far_point = p2

            self._hull_points.append(far_point)
            point = far_point

    def get_hull_points(self):
        if self._points and not self._hull_points:
            self.compute_hull()

        return self._hull_points
```

```python
"""
Generate m random, non-duplicate numbers between 1 and n
Credit to Knuth
"""

import random


def generate_random(count, max_val):
    selected = []

    for i in range(max_val):
        if ((random.random() * max_val) % (max_val - i)) < count:
            selected.append(i + 1)
            count -= 1

    return selected


def main():
    m = 20
    n = 100

    rands = generate_random(m, n)

    print(rands)

    assert len(rands) == m


if __name__ == '__main__':
    main()
```

```python
import numpy as np


def main():
    A = np.array([[3, -9], [2, 4]])
    b = np.array([-42, 2])

    z = np.linalg.solve(A, b)
    print(z)

    M = np.array([[1, -2, -1], [2, 2, -1], [-1, -1, 2]])
    c = np.array([6, 1, 1])

    y = np.linalg.solve(M, c)
    print(y)

    F = np.array([[30, 20], [50, 60]])
    e = np.array([6000, 12000])

    r = np.linalg.solve(F, e)
    print(r)


if __name__ == '__main__':
    main()
```

```python
"""
Binary search, iteratively
"""


def binary_search(nums, target):
    low = 0
    high = len(nums) - 1

    while low <= high:
        mid = low + (high - low) // 2
        if nums[mid] == target:
            return mid
        elif nums[mid] < target:
            low = mid + 1
        else:
            high = mid - 1

    return False


def main():
    nums = [-1024, -681, -73, -24, 6, 7, 16, 22, 22,
            23, 25, 35, 56, 234, 235, 262, 897, 3463,
            9999, 9999, 10000]

    assert binary_search(nums, 16) == 6
    assert binary_search(nums, -1024) == 0
    assert binary_search(nums, 3463) == 17
    assert binary_search(nums, 56) == 12
    assert binary_search(nums, 498) is False
    assert binary_search(nums, 10001) is False
    assert binary_search(nums, 10000) == 20
    assert binary_search(nums, 9999) == 18


if __name__ == '__main__':
    main()
```

```python
"""
Binary search, recursive
"""


def binary_search(data, target):
    return binary_search_recur(data, target, 0, len(data) - 1)


def binary_search_recur(data, target, low, high):
    """Return position if target is found in indicated portion of a Python list.

    The search only considers the portion from data[low] to data[high] inclusive.
    """
    if low > high:
        return False  # interval is empty; no match
    else:
        mid = (low + high) // 2
        if target == data[mid]:  # found a match
            return mid
        elif target < data[mid]:
            # recur on the portion left of the middle
            return binary_search_recur(data, target, low, mid - 1)
        else:
            # recur on the portion right of the middle
            return binary_search_recur(data, target, mid + 1, high)


def main():
    nums = [-1024, -681, -73, -24, 6, 7, 16, 22, 22,
            23, 25, 35, 56, 234, 235, 262, 897, 3463,
            9999, 9999, 10000]

    assert binary_search(nums, 16) == 6
    assert binary_search(nums, -1024) == 0
    assert binary_search(nums, 3463) == 17
    assert binary_search(nums, 56) == 12
    assert binary_search(nums, 498) is False
    assert binary_search(nums, 10001) is False
    assert binary_search(nums, 10000) == 20
    assert binary_search(nums, 9999) == 18


if __name__ == '__main__':
    main()
```