```
# Interactive code for Chapter 1

# The timing code is not included in the book:
"""
>>> from time import *
>>> t0 = time()
>>> count = 10**4 # Change to 10**5 for original (slow) test
>>> nums = []
>>> for i in range(count):
...     nums.append(i)
...
>>> nums.reverse()
>>> t1 = time() - t0
>>> t0 = time()
>>> nums = []
>>> for i in range(count):
...     nums.insert(0, i)
...
>>> t2 = time() - t0
"""

# CHAPTER-2
-----------
"""
>>> hash(42)
42
"""
# The same in 2.6 and 3.1 -- but not in 2.7
# >>> hash("Hello, world!")
# -943387004357456228

def test_1():
    n = 1000
    nums = [0]*n
    nums.append(1)
    nums.insert(0,1)

def dump_linked_list(L):
    res = []
    while L is not None:
        res.append(L.value)
        L = L.next
    return res

def test_loop_asymptotics():
    seq = range(10)

    s = 0
    for x in seq:
        s += x

    assert(s == sum(seq))

    squares = [x**2 for x in seq]

    s = 0
    for x in seq:
        for y in seq:
            s += x*y
        for z in seq:
            for w in seq:
                s += x-w

    seq1 = range(10)
    seq2 = range(5)

    s = 0
    for x in seq1:
        for y in seq2:
            s += x*y

    seq1 = [[0, 1], [2], [3, 4, 5]]
```

```
    s = 0
    for seq2 in seq1:
        for x in seq2:
            s += x

    seq = range(10)
    s = 0
    n = len(seq)
    for i in range(n-1):
        for j in range(i+1, n):
            s += seq[i] * seq[j]

def test_timeit():
    # Has been tested -- ignored here because timing details vary, and the
    # tests are slow
    IGNORE = """
    >>> timeit.timeit("x = 2 + 2")
    0.034976959228515625
    >>> timeit.timeit("x = sum(range(10))")
    0.92387008666992188
    """

def test_linked_list():
    """
    >>> Node = test_linked_list()
    >>> L = Node("a", Node("b", Node("c", Node("d"))))
    >>> L.next.next.value
    'c'
    >>> b = L.next
    >>> c = b.next
    >>> b.next = Node("x", c)
    >>> dump_linked_list(L)
    ['a', 'b', 'x', 'c', 'd']
    """
    class Node:
        def __init__(self, value, next=None):
            self.value = value
            self.next = next
    return Node

def test_listing_2_1():
    """
    >>> N = test_listing_2_1()
    >>> a, b, c, d, e, f, g, h = range(8)
    >>> b in N[a]  # Neighborhood membership
    True
    >>> len(N[f])  # Degree
    3
    """

    a, b, c, d, e, f, g, h = range(8)
    N = [
        [b, c, d, e, f],    # a
        [c, e],             # b
        [d],                # c
        [e],                # d
        [f],                # e
        [c, g, h],          # f
        [f, h],             # g
        [f, g]              # h
    ]

    return N

def test_listing_2_2():
    """
    >>> N = test_listing_2_2()
    >>> a, b, c, d, e, f, g, h = range(8)
    >>> b in N[a]  # Neighborhood membership
    True
    >>> len(N[f])  # Degree
    3
```

```
    """

    a, b, c, d, e, f, g, h = range(8)
    N = [
        {b, c, d, e, f},    # a
        {c, e},             # b
        {d},                # c
        {e},                # d
        {f},                # e
        {c, g, h},          # f
        {f, h},             # g
        {f, g}              # h
    ]

    return N

def test_listing_2_3():
    """
    >>> N = test_listing_2_3()
    >>> a, b, c, d, e, f, g, h = range(8)
    >>> b in N[a]  # Neighborhood membership
    True
    >>> len(N[f])  # Degree
    3
    >>> N[a][b]    # Edge weight for (a, b)
    2
    """

    a, b, c, d, e, f, g, h = range(8)
    N = [
        {b:2, c:1, d:3, e:9, f:4},    # a
        {c:4, e:3},                   # b
        {d:8},                        # c
        {e:7},                        # d
        {f:5},                        # e
        {c:2, g:2, h:2},              # f
        {f:1, h:6},                   # g
        {f:9, g:8}                    # h
    ]

    return N

def test_listing_2_4():
    """
    >>> N = test_listing_2_4()
    >>> 'b' in N['a']  # Neighborhood membership
    True
    >>> len(N['f'])  # Degree
    3
    """

    N = {
        'a': set('bcdef'),
        'b': set('ce'),
        'c': set('d'),
        'd': set('e'),
        'e': set('f'),
        'f': set('cgh'),
        'g': set('fh'),
        'h': set('fg')
    }

    return N

def test_listing_2_5():
    """
    >>> N = test_listing_2_5()
    >>> a, b, c, d, e, f, g, h = range(8)
    >>> N[a][b]    # Neighborhood membership
    1
    >>> sum(N[f])  # Degree
    3
```

```
    """

    a, b, c, d, e, f, g, h = range(8)

    #     a b c d e f g h

    N = [[0,1,1,1,1,1,0,0], # a
         [0,0,1,0,1,0,0,0], # b
         [0,0,0,1,0,0,0,0], # c
         [0,0,0,0,1,0,0,0], # d
         [0,0,0,0,0,1,0,0], # e
         [0,0,1,0,0,0,1,1], # f
         [0,0,0,0,0,1,0,1], # g
         [0,0,0,0,0,1,1,0]] # h

    return N

def test_listing_2_6():
    """
    >>> W = test_listing_2_6()
    >>> a, b, c, d, e, f, g, h = range(8)
    >>> inf = float('inf')
    >>> W[a][b] < inf   # Neighborhood membership
    True
    >>> W[c][e] < inf   # Neighborhood membership
    False
    >>> sum(1 for w in W[a] if w < inf) - 1  # Degree
    5
    """

    a, b, c, d, e, f, g, h = range(8)
    _ = float('inf')

    #     a b c d e f g h

    W = [[0,2,1,3,9,4,_,_], # a
         [_,0,4,_,3,_,_,_], # b
         [_,_,0,8,_,_,_,_], # c
         [_,_,_,0,7,_,_,_], # d
         [_,_,_,_,0,5,_,_], # e
         [_,_,2,_,_,0,2,2], # f
         [_,_,_,_,_,1,0,6], # g
         [_,_,_,_,_,9,8,0]] # h

    return W

def test_list_tree():
    """
    >>> T = [["a", "b"], ["c"], ["d", ["e", "f"]]]
    >>> T[0][1]
    'b'
    >>> T[2][1][0]
    'e'
    """

def test_listing_2_7():
    """
    >>> Tree = test_listing_2_7()
    >>> t = Tree(Tree("a", "b"), Tree("c", "d"))
    >>> t.right.left
    'c'
    """
    class Tree:
        def __init__(self, left, right=None):
            self.left = left
            self.right = right
    return Tree


def test_listing_2_8():
    """
    >>> Tree = test_listing_2_8()
```

```
    >>> t = Tree(Tree("a", Tree("b", Tree("c", Tree("d")))))
    >>> t.kids.next.next.val
    'c'
    """
    class Tree:
        def __init__(self, kids, next=None):
            self.kids = self.val = kids
            self.next = next
    return Tree


def test_bunch():
    """
    >>> Bunch = test_bunch()
    >>> x = Bunch(name="Jayne Cobb", position="Public Relations")
    >>> x.name
    'Jayne Cobb'
    >>> T = Bunch
    >>> t = T(left=T(left="a", right="b"), right=T(left="c"))
    >>> t.left
    {'right': 'b', 'left': 'a'}
    >>> t.left.right
    'b'
    >>> t['left']['right']
    'b'
    >>> "left" in t.right
    True
    >>> "right" in t.right
    False
    """
    class Bunch(dict):
        def __init__(self, *args, **kwds):
            super(Bunch, self).__init__(*args, **kwds)
            self.__dict__ = self
    return Bunch

def test_hidden_squares():
    """
    >>> from random import randrange, seed
    >>> seed(529)
    >>> L = [randrange(10000) for i in range(1000)]
    >>> 42 in L
    False
    >>> S = set(L)
    >>> 42 in S
    False
    >>> input = ["x", "y", "z"]
    >>> s = ""
    >>> for chunk in input:
    ...     s += chunk
    ...
    >>> chunks = []
    >>> for chunk in input:
    ...     chunks.append(chunk)
    ...
    >>> s = ''.join(chunks)
    >>> s = ''.join(input)
    >>> lists = [[42] for i in range(100)]
    >>> res = []
    >>> for lst in lists:
    ...    res.extend(lst)
    >>> res = sum(lists, [])
    """

def test_floats_and_decimals():
    """
    >>> sum(0.1 for i in range(10)) == 1.0
    False
    >>> def almost_equal(x, y, places=7):
    ...     return round(abs(x-y), places) == 0
    >>> almost_equal(sum(0.1 for i in range(10)), 1.0)
    True
```

```
    >>> from decimal import *
    >>> sum(Decimal("0.1") for i in range(10)) == Decimal("1.0")
    True
    >>> from math import sqrt
    >>> x = 8762348761.13
    >>> sqrt(x + 1) - sqrt(x)
    5.341455107554793e-06
    >>> 1.0/(sqrt(x + 1) + sqrt(x))
    5.3414570026237696e-06
    """


# CHAPTER-3
-----------
def test_basic_notation():
    """
    >>> from random import *
    >>> seq = S = [randrange(100) for i in range(1000)]
    >>> x = randrange(100)
    >>> x*sum(S) == sum(x*y for y in S)
    True
    >>> def f(i): return i
    >>> def g(i): return 2*i + 42
    >>> m = 0; n = 10
    >>> dummy = sum(f(i) for i in range(m, n+1))
    >>> s = 0
    >>> for i in range(m, n+1):
    ...     s += f(i)
    ...
    >>> X = sum(f(i) for i in seq) + sum(g(i) for i in seq)
    >>> Y = sum(f(i) + g(i) for i in seq)
    >>> X == Y
    True
    """


def test_particles():
    """
    >>> import random; random.seed(42)
    >>> from random import randrange
    >>> n = 10**90
    >>> p = randrange(n)
    >>> p == 525619275483324350902827558940034848040198424203331
    False
    >>> p < n/2
    True
    >>> from math import log
    >>> log(n, 2) #doctest: +ELLIPSIS
    298.9735...
    """


def test_primes():
    """
    >>> is_prime = test_primes()
    >>> is_prime(4)
    False
    >>> is_prime(100)
    False
    >>> is_prime(37)
    True
    """
    def is_prime(n):
        for i in range(2,n):
            if n % i == 0: return False
        return True

    return is_prime


def test_recursion():
    """
```

```
    >>> from random import *
    >>> seq = [randrange(1000) for i in range(100)]
    >>> S(seq) == sum(seq)
    True
    >>> seq = range(1,101)
    >>> S(seq)
    5050
    """

def S(seq, i=0):
    if i == len(seq): return 0
    return S(seq, i+1) + seq[i]

def test_rec_time():
    """
    >>> from random import *
    >>> seq = [randrange(1000) for i in range(100)]
    >>> T(seq) == len(seq) + 1
    True
    >>> seq = range(1,101)
    >>> T(seq)
    101
    >>> for n in range(100):
    ...    seq = range(n)
    ...    assert T(seq) == n+1
    ...
    >>>
    """

def T(seq, i=0):
    if i == len(seq): return 1
    return T(seq, i+1) + 1

def test_gnomesort():
    """
    >>> from random import *
    >>> for i in range(10):
    ...     seq = [randrange(1000) for i in range(100)]
    ...     seq2 = sorted(seq)
    ...     gnomesort(seq)
    ...     assert seq == seq2
    ...
    >>>
    """

def gnomesort(a):
    i = 0
    while i < len(a):
        if i == 0 or a[i-1] <= a[i]:
            i += 1
        else:
            a[i], a[i-1] = a[i-1], a[i]
            i -= 1

def test_mergesort():
    """
    >>> from random import *
    >>> for i in range(10):
    ...     seq = [randrange(1000) for i in range(100)]
    ...     seq2 = sorted(seq)
    ...     seq = mergesort(seq)
    ...     assert seq == seq2
    ...
    >>>
    """

def mergesort(a):
    n = len(a)
    lft, rgt = a[:n//2], a[n//2:]
    if len(lft) > 1: lft = mergesort(lft)
    if len(rgt) > 1: rgt = mergesort(rgt)
    res = []
```

```
    while lft and rgt:
        if lft[-1] >= rgt[-1]:
            res.append(lft.pop())
        else:
            res.append(rgt.pop())
    res.reverse()
    return (lft or rgt) + res

# CHAPTER-4
-----------
import sys

def test_closest_pair():
    """
    >>> from random import randrange
    >>> from random import seed; seed(2523)
    >>> seq = [randrange(10**5) for i in range(100)]
    >>> dd = float("inf")
    >>> for x in seq:
    ...    for y in seq:
    ...       if x == y: continue
    ...       d = abs(x-y)
    ...       if d < dd:
    ...           xx, yy, dd = x, y, d
    ...
    >>> xx, yy
    (29836, 29825)
    >>> seq.sort()
    >>> dd = float("inf")
    >>> for i in range(len(seq)-1):
    ...     x, y = seq[i], seq[i+1]
    ...     if x == y: continue
    ...     d = abs(x-y)
    ...     if d < dd:
    ...         xx, yy, dd = x, y, d
    ...
    >>> xx, yy
    (29825, 29836)
    """

def test_board():
    """
    >>> board = [[0]*8 for i in range(8)]
    >>> board[7][7] = -1
    >>> cover(board)
    22
    >>> for row in board:
    ...     print((" %2i"*8) % tuple(row))
      3  3  4  4  8  8  9  9
      3  2  2  4  8  7  7  9
      5  2  6  6 10 10  7 11
      5  5  6  1  1 10 11 11
     13 13 14  1 18 18 19 19
     13 12 14 14 18 17 17 19
     15 12 12 16 20 17 21 21
     15 15 16 16 20 20 21 -1
    """

def cover(board, lab=1, top=0, left=0, side=None):
    if side is None: side = len(board)

    # Side length of sub-board:
    s = side // 2

    # Offsets for outer/inner squares of sub-boards:
    offsets = (0, -1), (side-1, 0)

    for dy_outer, dy_inner in offsets:
        for dx_outer, dx_inner in offsets:
            # If the outer corner is not set...
            if not board[top+dy_outer][left+dx_outer]:
                # ... label the inner corner:
```

```
                board[top+s+dy_inner][left+s+dx_inner] = lab

    # Next label:
    lab += 1
    if s > 1:
        for dy in [0, s]:
            for dx in [0, s]:
                # Recursive calls, if s is at least 2:
                lab = cover(board, lab, top+dy, left+dx, s)

    # Return the next available label:
    return lab


def test_trav():
    """
    >>> def trav(seq, i=0):
    ...     if i==len(seq): return
    ...     trav(seq, i+1)
    ...
    >>> trav(range(100))
    """
    # Using range(1000) should give max recursion depth exceeded

def test_recursive_insertion_sort():
    """
    >>> from random import randrange
    >>> seq = [randrange(1000) for i in range(100)]
    >>> seq2 = list(seq)
    >>> ins_sort_rec(seq, len(seq)-1)
    >>> seq == seq2
    False
    >>> seq2.sort()
    >>> seq == seq2
    True
    """

def ins_sort_rec(seq, i):
    if i==0: return                             # Base case -- do nothing
    ins_sort_rec(seq, i-1)                      # Sort 0..i-1
    j = i                                       # Start "walking" down
    while j > 0 and seq[j-1] > seq[j]:          # Look for OK spot
        seq[j-1], seq[j] = seq[j], seq[j-1]     # Keep moving seq[j] down
        j -= 1                                  # Decrement j


def test_insertion_sort():
    """
    >>> from random import *
    >>> seq = [randrange(1000) for i in range(100)]
    >>> seq2 = list(seq)
    >>> ins_sort(seq)
    >>> seq == seq2
    False
    >>> seq2.sort()
    >>> seq == seq2
    True
    """

def ins_sort(seq):
    for i in range(1,len(seq)):                 # 0..i-1 sorted so far
        j = i                                   # Start "walking" down
        while j > 0 and seq[j-1] > seq[j]:      # Look for OK spot
            seq[j-1], seq[j] = seq[j], seq[j-1] # Keep moving seq[j] down
            j -= 1                              # Decrement j

def test_recursive_selection_sort():
    """
    >>> from random import *
    >>> seq = [randrange(1000) for i in range(100)]
    >>> seq2 = list(seq)
    >>> sel_sort_rec(seq, len(seq)-1)
```

```
    >>> seq == seq2
    False
    >>> seq2.sort()
    >>> seq == seq2
    True
    """

def sel_sort_rec(seq, i):
    if i==0: return                             # Base case -- do nothing
    max_j = i                                   # Idx. of largest value so far
    for j in range(i):                          # Look for a larger value
        if seq[j] > seq[max_j]: max_j = j       # Found one? Update max_j
    seq[i], seq[max_j] = seq[max_j], seq[i]     # Switch largest into place
    sel_sort_rec(seq, i-1)                      # Sort 0..i-1

def test_selection_sort():
    """
    >>> from random import *
    >>> seq = [randrange(1000) for i in range(100)]
    >>> seq2 = list(seq)
    >>> sel_sort(seq)
    >>> seq == seq2
    False
    >>> seq2.sort()
    >>> seq == seq2
    True
    """

def sel_sort(seq):
    for i in range(len(seq)-1,0,-1):            # n..i+1 sorted so far
        max_j = i                               # Idx. of largest value so far
        for j in range(i):                      # Look for a larger value
            if seq[j] > seq[max_j]: max_j = j   # Found one? Update max_j
        seq[i], seq[max_j] = seq[max_j], seq[i] # Switch largest into place


def test_naive_perm():
    """
    >>> M = [2, 2, 0, 5, 3, 5, 7, 4]
    >>> M[2] # c is mapped to a
    0
    >>> sorted(naive_max_perm(M))
    [0, 2, 5]
    """

def naive_max_perm(M, A=None):
    if A is None:                               # The elt. set not supplied?
        A = set(range(len(M)))                  # A = {0, 1, ... , n-1}
    if len(A) == 1: return A                    # Base case -- single-elt. A
    B = set(M[i] for i in A)                    # The "pointed to" elements
    C = A - B                                   # "Not pointed to" elements
    if C:                                       # Any useless elements?
        A.remove(C.pop())                       # Remove one of them
        return naive_max_perm(M, A)             # Solve remaining problem
    return A                                    # All useful -- return all


def test_perm():
    """
    >>> M = [2, 2, 0, 5, 3, 5, 7, 4]
    >>> M[2] # c is mapped to a
    0
    >>> sorted(max_perm(M))
    [0, 2, 5]
    """

def max_perm(M):
    n = len(M)                                  # How many elements?
    A = set(range(n))                           # A = {0, 1, ... , n-1}
    count = [0]*n                               # C[i] == 0 for i in A
    for i in M:                                 # All that are "pointed to"
        count[i] += 1                           # Increment "point count"
```

```
    Q = [i for i in A if count[i] == 0]         # Useless elements
    while Q:                                    # While useless elts. left...
        i = Q.pop()                             # Get one
        A.remove(i)                             # Remove it
        j = M[i]                                # Who's it pointing to?
        count[j] -= 1                           # Not anymore...
        if count[j] == 0:                       # Is j useless now?
            Q.append(j)                         # Then deal w/it next
    return A                                    # Return useful elts.

def test_alternate_perm():
    """
    >>> M = [2, 2, 0, 5, 3, 5, 7, 4]
    >>> M[2] # c is mapped to a
    0
    >>> sorted(alternate_max_perm(M))
    [0, 2, 5]
    """

# A test of the tip that says the for loop can be replaced with the use of
# collections.Counter:
def alternate_max_perm(M):
    # Satisfy the Python 2.6 test run:
    if sys.version <= "3.1": return max_perm(M)
    from collections import Counter
    n = len(M)                                  # How many elements?
    A = set(range(n))                           # A = {0, 1, ... , n-1}
    count = [0]*n                               # C[i] == 0 for i in A
    count = Counter(M)
    Q = [i for i in A if count[i] == 0]         # Useless elements
    while Q:                                    # While useless elts. left...
        i = Q.pop()                             # Get one
        A.remove(i)                             # Remove it
        j = M[i]                                # Who's it pointing to?
        count[j] -= 1                           # Not anymore...
        if count[j] == 0:                       # Is j useless now?
            Q.append(j)                         # Then deal w/it next
    return A                                    # Return useful elts.

def test_counting_sort():
    """
    >>> k = 100
    >>> from random import *
    >>> seq = [randrange(k) for i in range(100)]
    >>> seq2 = list(seq)
    >>> seq = counting_sort(seq) # counting_sort(seq, k)
    >>> seq == seq2
    False
    >>> seq2.sort()
    >>> seq == seq2
    True
    """

def old_counting_sort(A, k):                    # Value range = 0..k-1
    n = len(A)
    B, C = [0]*n, [0]*k                         # Output and counts
    for x in A:
        C[x] += 1                               # Count it
    for x in range(1,k):
        C[x] += C[x-1]                          # Make counts cumulative
    for x in reversed(A):
        C[x] -= 1                               # Find position of x
        B[C[x]] = x                             # Insert x at its position
    return B

from collections import defaultdict

def counting_sort(A, key=lambda x: x):
    B, C = [], defaultdict(list)                # Output and "counts"
    for x in A:
        C[key(x)].append(x)                     # "Count" key(x)
    for k in range(min(C), max(C)+1):           # For every key in the range
```

```
        B.extend(C[k])                          # Add values in sorted order
    return B

def test_naive_celeb():
    """
    >>> from random import *
    >>> n = 100
    >>> G = [[randrange(2) for i in range(n)] for i in range(n)]
    >>> c = randrange(n)
    >>> c = 57 # For testing
    >>> for i in range(n):
    ...     G[i][c] = True
    ...     G[c][i] = False
    ...
    >>> naive_celeb(G)
    57
    """

def naive_celeb(G):
    n = len(G)
    for u in range(n):                          # For every candidate...
        for v in range(n):                      # For everyone else...
            if u == v: continue                 # Same person? Skip.
            if G[u][v]: break                   # Candidate knows other
            if not G[v][u]: break               # Other doesn't know candidate
        else:
            return u                            # No breaks? Celebrity!
    return None                                 # Couldn't find anyone

def test_celeb():
    """
    >>> from random import *
    >>> n = 100
    >>> G = [[randrange(2) for i in range(n)] for i in range(n)]
    >>> c = randrange(n)
    >>> c = 57 # For testing
    >>> for i in range(n):
    ...     G[i][c] = True
    ...     G[c][i] = False
    ...
    >>> celeb(G)
    57
    """

def celeb(G):
    n = len(G)
    u, v = 0, 1                                 # The first two
    for c in range(2,n+1):                      # Others to check
        if G[u][v]: u = c                       # u knows v? Replace u
        else:       v = c                       # Otherwise, replace v
    if u == n:      c = v                       # u was replaced last; use v
    else:           c = u                       # Otherwise, u is a candidate
    for v in range(n):                          # For everyone else...
        if c == v: continue                     # Same person? Skip.
        if G[c][v]: break                       # Candidate knows other
        if not G[v][c]: break                   # Other doesn't know candidate
    else:
        return c                                # No breaks? Celebrity!
    return None                                 # Couldn't find anyone

def test_naive_topsort():
    """
    >>> n = 6
    >>> from random import sample, randrange, shuffle
    >>> from random import seed; seed(2365)
    >>> G = dict()
    >>> seq = list(range(n)) # Py 3 range objects aren't sequences
    >>> shuffle(seq)
    >>> rest = set(seq)
    >>> for x in seq[:-1]:
    ...     rest.remove(x)
    ...     m = randrange(1,len(rest)+1)
```

```
    ...     G[x] = set(sample(rest, m))
    ...
    >>> G[seq[-1]] = set()
    >>> sorted = naive_topsort(G)
    >>> rest = set(sorted)
    >>> for u in sorted:
    ...     rest.remove(u)
    ...     assert G[u] <= rest
    ...
    >>> G = {'a': set('bf'), 'b': set('cdf'),
    ... 'c': set('d'), 'd': set('ef'), 'e': set('f'), 'f': set()}
    >>> naive_topsort(G)
    ['a', 'b', 'c', 'd', 'e', 'f']
    """

def naive_topsort(G, S=None):
    if S is None: S = set(G)                    # Default: All nodes
    if len(S) == 1: return list(S)              # Base case, single node
    v = S.pop()                                 # Reduction: Remove a node
    seq = naive_topsort(G, S)                   # Recursion (assumption), n-1
    min_i = 0
    for i, u in enumerate(seq):
        if v in G[u]: min_i = i+1               # After all dependencies
    seq.insert(min_i, v)
    return seq

def test_topsort():
    """
    >>> n = 6
    >>> from random import sample, randrange, shuffle
    >>> from random import seed; seed(2365)
    >>> G = dict()
    >>> seq = list(range(n)) # Py 3 range objects aren't sequences
    >>> shuffle(seq)
    >>> rest = set(seq)
    >>> for x in seq[:-1]:
    ...     rest.remove(x)
    ...     m = randrange(1,len(rest)+1)
    ...     G[x] = set(sample(rest, m))
    ...
    >>> G[seq[-1]] = set()
    >>> sorted = topsort(G)
    >>> rest = set(sorted)
    >>> for u in sorted:
    ...     rest.remove(u)
    ...     assert G[u] <= rest
    ...
    >>> G = {'a': set('bf'), 'b': set('cdf'),
    ... 'c': set('d'), 'd': set('ef'), 'e': set('f'), 'f': set()}
    >>> topsort(G)
    ['a', 'b', 'c', 'd', 'e', 'f']
    """

def topsort(G):
    count = dict((u, 0) for u in G)             # The in-degree for each node
    for u in G:
        for v in G[u]:
            count[v] += 1                       # Count every in-edge
    Q = [u for u in G if count[u] == 0]         # Valid initial nodes
    S = []                                      # The result
    while Q:                                    # While we have start nodes...
        u = Q.pop()                             # Pick one
        S.append(u)                             # Use it as first of the rest
        for v in G[u]:
            count[v] -= 1                       # "Uncount" its out-edges
            if count[v] == 0:                   # New valid start nodes?
                Q.append(v)                     # Deal with them next
    return S

def test_relax():
    """
    >>> n = 100
```

```
    >>> from random import *
    >>> B = dict((i, dict((j, randrange(1000)) for j in range(n)))
    ... for i in range(n))
    >>> for i in range(n):
    ...     B[i][i] = 0
    >>> A = dict((i, randrange(1000)) for i in range(n))
    >>> C = {}
    >>> N = 100
    >>> for v in range(n):
    ...     C[v] = float('inf')
    >>> for i in range(N):
    ...     u, v = randrange(n), randrange(n)
    ...     C[v] = min(C[v], A[u] + B[u][v]) # Relax
    """

# CHAPTER-5
-----------
def some_graph():
    a, b, c, d, e, f, g, h = range(8)
    N = [
        [b, c, d, e, f],    # a
        [c, e],             # b
        [d],                # c
        [e],                # d
        [f],                # e
        [c, g, h],          # f
        [f, h],             # g
        [f, g]              # h
    ]
    return N

def some_tree():
    a, b, c, d, e, f, g, h = range(8)
    N = [
        [b, c],             # a
        [d, e],             # b
        [f, g],             # c
        [],                 # d
        [],                 # e
        [],                 # f
        [h],                # g
        []                  # h
    ]
    return N

class stack(list):
    add = list.append

def test_traverse():
    """
    >>> G = some_graph()
    >>> list(traverse(G, 0))
    [0, 1, 2, 3, 4, 5, 6, 7]
    >>> list(traverse(G, 0, stack))
    [0, 5, 7, 6, 2, 3, 4, 1]
    >>> for i in range(len(G)): G[i] = set(G[i])
    >>> sorted(walk(G, 0))
    [0, 1, 2, 3, 4, 5, 6, 7]
    >>> G = {
    ...    0: set([1, 2]),
    ...    1: set([0, 2]),
    ...    2: set([0, 1]),
    ...    3: set([4, 5]),
    ...    4: set([3, 5]),
    ...    5: set([3, 4])
    ... }
    >>> comp = []
    >>> seen = set()
    >>> for u in G:
    ...     if u in seen: continue
    ...     C = walk(G, u)
    ...     seen.update(C)
```

```
    ...     comp.append(C)
    ...
    >>> [list(sorted(C)) for C in comp]
    [[0, 1, 2], [3, 4, 5]]
    >>> [list(sorted(C)) for C in components(G)]
    [[0, 1, 2], [3, 4, 5]]
    """

def walk(G, s, S=set()):                        # Walk the graph from node s
    P, Q = dict(), set()                        # Predecessors + "to do" queue
    P[s] = None                                 # s has no predecessor
    Q.add(s)                                    # We plan on starting with s
    while Q:                                    # Still nodes to visit
        u = Q.pop()                             # Pick one, arbitrarily
        for v in G[u].difference(P, S):         # New nodes?
            Q.add(v)                            # We plan to visit them!
            P[v] = u                            # Remember where we came from
    return P                                    # The traversal tree

def components(G):                              # The connected components
    comp = []
    seen = set()                                # Nodes we've already seen
    for u in G:                                 # Try every starting point
        if u in seen: continue                  # Seen? Ignore it
        C = walk(G, u)                          # Traverse component
        seen.update(C)                          # Add keys of C to seen
        comp.append(C)                          # Collect the components
    return comp

def traverse(G, s, qtype=set):
    S, Q = set(), qtype()
    Q.add(s)
    while Q:
        u = Q.pop()
        if u in S: continue
        S.add(u)
        for v in G[u]:
            Q.add(v)
        yield u

def test_tree_walk():
    """
    >>> T = some_tree()
    >>> tree_walk(T, 0) # Testing that it doesn't crash
    >>> list(tree_walk_tested(T, 0)) # Get the ordering
    [0, 1, 3, 4, 2, 5, 6, 7]
    """

def tree_walk(T, r):                            # Traverse T from root r
    for u in T[r]:                              # For each child...
        tree_walk(T, u)                         # ... traverse its subtree

def tree_walk_tested(T, r):
    yield r # For testing
    for u in T[r]:
        for v in tree_walk_tested(T, u):
            yield v

def test_dfs():
    """
    >>> G = some_graph()
    >>> for i in range(len(G)): G[i] = set(G[i])
    >>> list(rec_dfs(G, 0))
    [0, 1, 2, 3, 4, 5, 6, 7]
    >>> rec_dfs_tested(G, 0)
    [0, 1, 2, 3, 4, 5, 6, 7]
    >>> list(iter_dfs(G, 0))
    [0, 5, 7, 6, 2, 3, 4, 1]
    >>> d = {}; f = {}
    >>> dfs(G, 0, d, f)
    16
    >>> [d[v] for v in range(len(G))]
```

```
    [0, 1, 2, 3, 4, 5, 6, 7]
    >>> [f[v] for v in range(len(G))]
    [15, 14, 13, 12, 11, 10, 9, 8]
    """

# Important: Can't use "for u in G[s] - S" here, bc S might change
def rec_dfs(G, s, S=None):
    if S is None: S = set()                     # Initialize the history
    S.add(s)                                    # We've visited s
    for u in G[s]:                              # Explore neighbors
        if u in S: continue                     # Already visited: Skip
        rec_dfs(G, u, S)                        # New: Explore recursively
    return S # For testing

def rec_dfs_tested(G, s, S=None):
    if S is None: S = []
    S.append(s)
    for u in G[s]:
        if u in S: continue
        rec_dfs_tested(G, u, S)
    return S

def iter_dfs(G, s):
    S, Q = set(), []                            # Visited-set and queue
    Q.append(s)                                 # We plan on visiting s
    while Q:                                    # Planned nodes left?
        u = Q.pop()                             # Get one
        if u in S: continue                     # Already visited? Skip it
        S.add(u)                                # We've visited it now
        Q.extend(G[u])                          # Schedule all neighbors
        yield u                                 # Report u as visited

def dfs(G, s, d, f, S=None, t=0):
    if S is None: S = set()                     # Initialize the history
    d[s] = t; t += 1                            # Set discover time
    S.add(s)                                    # We've visited s
    for u in G[s]:                              # Explore neighbors
        if u in S: continue                     # Already visited. Skip
        t = dfs(G, u, d, f, S, t)               # Recurse; update timestamp
    f[s] = t; t += 1                            # Set finish time
    return t                                    # Return timestamp

def test_dfs_topsort():
    """
    >>> n = 6
    >>> from random import sample, randrange, shuffle
    >>> from random import seed; seed(2365)
    >>> G = dict()
    >>> seq = list(range(n)) # Py 3 range objects aren't sequences
    >>> shuffle(seq)
    >>> rest = set(seq)
    >>> for x in seq[:-1]:
    ...     rest.remove(x)
    ...     m = randrange(1,len(rest)+1)
    ...     G[x] = set(sample(rest, m))
    ...
    >>> G[seq[-1]] = set()
    >>> sorted = dfs_topsort(G)
    >>> rest = set(sorted)
    >>> for u in sorted:
    ...     rest.remove(u)
    ...     assert G[u] <= rest
    ...
    >>> G = {'a': set('bf'), 'b': set('cdf'),
    ... 'c': set('d'), 'd': set('ef'), 'e': set('f'), 'f': set()}
    >>> dfs_topsort(G)
    ['a', 'b', 'c', 'd', 'e', 'f']
    """

def dfs_topsort(G):
    S, res = set(), []                          # History and result
    def recurse(u):                             # Traversal subroutine
```

```
        if u in S: return                       # Ignore visited nodes
        S.add(u)                                # Otherwise: Add to history
        for v in G[u]:
            recurse(v)                          # Recurse through neighbors
        res.append(u)                           # Finished with u: Append it
    for u in G:
        recurse(u)                              # Cover entire graph
    res.reverse()                               # It's all backward so far
    return res

def test_iddfs_and_bfs():
    """
    >>> G = some_graph()
    >>> list(iddfs(G, 0))
    [0, 1, 2, 3, 4, 5, 6, 7]
    >>> bfs(G, 0)
    {0: None, 1: 0, 2: 0, 3: 0, 4: 0, 5: 0, 6: 5, 7: 5}
    >>> G = [[1, 2], [0, 3], [0, 3], [1, 2]]
    >>> list(iddfs(G, 0))
    [0, 1, 2, 3]
    >>> bfs(G, 0)
    {0: None, 1: 0, 2: 0, 3: 1}
    >>> P = _
    >>> u = 3
    >>> path = [u]
    >>> while P[u] is not None:
    ...     path.append(P[u])
    ...     u = P[u]
    ...
    >>> path.reverse()
    >>> path
    [0, 1, 3]
    """

def iddfs(G, s):
    yielded = set()                             # Visited for the first time
    def recurse(G, s, d, S=None):               # Depth-limited DFS
        if s not in yielded:
            yield s
            yielded.add(s)
        if d == 0: return                       # Max depth zero: Backtrack
        if S is None: S = set()
        S.add(s)
        for u in G[s]:
            if u in S: continue
            for v in recurse(G, u, d-1, S):     # Recurse with depth-1
                yield v
    n = len(G)
    for d in range(n):                          # Try all depths 0..V-1
        if len(yielded) == n: break             # All nodes seen?
        for u in recurse(G, s, d):
            yield u

from collections import deque

def bfs(G, s):
    P, Q = {s: None}, deque([s])                # Parents and FIFO queue
    while Q:
        u = Q.popleft()                         # Constant-time for deque
        for v in G[u]:
            if v in P: continue                 # Already has parent
            P[v] = u                            # Reached from u: u is parent
            Q.append(v)
    return P

from string import ascii_lowercase
def parse_graph(s):
    G = {}
    for u, line in zip(ascii_lowercase, s.split("/")):
        G[u] = set(line)
    return G
```

```
def test_scc():
    """
    >>> G = parse_graph('bc/die/d/ah/f/g/eh/i/h')
    >>> list(map(list, scc(G)))
    [['a', 'c', 'b', 'd'], ['e', 'g', 'f'], ['i', 'h']]
    """

def tr(G):                                      # Transpose (rev. edges of) G
    GT = {}
    for u in G: GT[u] = set()                   # Get all the nodes in there
    for u in G:
        for v in G[u]:
            GT[v].add(u)                        # Add all reverse edges
    return GT

def scc(G):
    GT = tr(G)                                  # Get the transposed graph
    sccs, seen = [], set()
    for u in dfs_topsort(G):                    # DFS starting points
        if u in seen: continue                  # Ignore covered nodes
        C = walk(GT, u, seen)                   # Don't go "backward" (seen)
        seen.update(C)                          # We've now seen C
        sccs.append(C)                          # Another SCC found
    return sccs

# CHAPTER-6
-----------
def test_heap():
    """
    >>> from heapq import heappush, heappop
    >>> from random import randrange
    >>> Q = []
    >>> for i in range(10):
    ...     heappush(Q, randrange(100))
    ...
    >>> Q
    [15, 20, 56, 21, 62, 87, 67, 74, 50, 74]
    >>> [heappop(Q) for i in range(10)]
    [15, 20, 21, 50, 56, 62, 67, 74, 74, 87]
    """

# Pseudocode(ish)
def divide_and_conquer(S, divide, combine):
    if len(S) == 1: return S
    L, R = divide(S)
    A = divide_and_conquer(L, divide, combine)
    B = divide_and_conquer(R, divide, combine)
    return combine(A, B)

def test_bisect():
    """
    >>> from bisect import bisect
    >>> a = [0, 2, 3, 5, 6, 8, 8, 9]
    >>> bisect(a, 5)
    4
    >>> from bisect import bisect_left
    >>> bisect_left(a, 5)
    3
    """

# From the Python library, Python 2.3. License issues?
def bisect_right(a, x, lo=0, hi=None):
    if hi is None:                              # Searching to the end
        hi = len(a)
    while lo < hi:                              # More than one possibility
        mid = (lo+hi)//2                        # Bisect (find midpoint)
        if x < a[mid]: hi = mid                 # Value < middle? Go left
        else: lo = mid+1                        # Otherwise: Go right
    return lo

# From the Python library, Python 2.3. License issues?
# Renamed from _siftdown
```

```
def sift_up(heap, startpos, pos):
    newitem = heap[pos]                         # The item we're sifting up
    while pos > startpos:                       # Don't go beyond the root
        parentpos = (pos - 1) >> 1              # The same as (pos - 1) // 2
        parent = heap[parentpos]                # Who's your daddy?
        if parent <= newitem: break             # Valid parent found
        heap[pos] = parent                      # Otherwise: Copy parent down
        pos = parentpos                         # Next candidate position
    heap[pos] = newitem                         # Place the item in its spot


# Note: Duplicates are overwritten
def test_binary_tree():
    """
    >>> tree = Tree()
    >>> tree["a"] = 42
    >>> tree["a"]
    42
    >>> "b" in tree
    False
    >>> tree = Tree()
    >>> keys = [4,2,6,1,3,5,7]
    >>> for key in keys:
    ...     tree[key] = str(key)
    ...
    >>> print(bin_tree_str(tree.root))
    4:'4'{2:'2'{1:'1'{*,*},3:'3'{*,*}},6:'6'{5:'5'{*,*},7:'7'{*,*}}}
    >>> tree[6] = "?"
    >>> print(bin_tree_str(tree.root))
    4:'4'{2:'2'{1:'1'{*,*},3:'3'{*,*}},6:'?'{5:'5'{*,*},7:'7'{*,*}}}
    >>> tree[3]
    '3'
    >>> tree[6]
    '?'
    >>> 5 in tree
    True
    >>> 19 in tree
    False
    >>> tree[19]
    Traceback (most recent call last):
        ...
    KeyError
    """


def bin_tree_str(root, chunks=None):
    if chunks is None: chunks = []
    if root is None: chunks.append("*")
    else:
        chunks.append(repr(root.key))
        chunks.append(":")
        chunks.append(repr(root.val))
        chunks.append("{")
        bin_tree_str(root.lft, chunks)
        chunks.append(",")
        bin_tree_str(root.rgt, chunks)
        chunks.append("}")
    return "".join(chunks)

class Node:
    lft = None
    rgt = None
    def __init__(self, key, val):
        self.key = key
        self.val = val

def insert(node, key, val):
    if node is None: return Node(key, val)      # Empty leaf: Add node here
    if node.key == key: node.val = val          # Found key: Replace val
    elif key < node.key:                        # Less than the key?
        node.lft = insert(node.lft, key, val)   # Go left
    else:                                       # Otherwise...
```

```
        node.rgt = insert(node.rgt, key, val)   # Go right
    return node

def search(node, key):
    if node is None: raise KeyError             # Empty leaf: It's not here
    if node.key == key: return node.val         # Found key: Return val
    elif key < node.key:                        # Less than the key?
        return search(node.lft, key)            # Go left
    else:                                       # Otherwise...
        return search(node.rgt, key)            # Go right

class Tree:                                     # Simple wrapper
    root = None
    def __setitem__(self, key, val):
        self.root = insert(self.root, key, val)
    def __getitem__(self, key):
        return search(self.root, key)
    def __contains__(self, key):
        try: search(self.root, key)
        except KeyError: return False
        return True

# ----------------------------------------------------------------------------

def aa_tree_str(root, chunks=None):
    if chunks is None: chunks = []
    if root is None: chunks.append("*")
    else:
        chunks.append(repr(root.key))
        chunks.append(repr(root.val))
        chunks.append("@")
        chunks.append(repr(root.lvl))
        chunks.append("{")
        aa_tree_str(root.lft, chunks)
        chunks.append(",")
        aa_tree_str(root.rgt, chunks)
        chunks.append("}")
    return "".join(chunks)

def test_aa_tree():
    """
    >>> root = None
    >>> for key in range(7):
    ...     root = aa_insert(root, key, str(key))
    ...
    >>> print(aa_tree_str(root))
    3'3'@3{1'1'@2{0'0'@1{*,*},2'2'@1{*,*}},5'5'@2{4'4'@1{*,*},6'6'@1{*,*}}}
    """

class AANode:
    lft = None
    rgt = None
    lvl = 1                                     # We've added a level...
    def __init__(self, key, val):
        self.key = key
        self.val = val

def skew(node):                                 # Basically a right rotation
    if None in [node, node.lft]: return node    # No need for a skew
    if node.lft.lvl != node.lvl: return node    # Still no need
    lft = node.lft                              # The 3 steps of the rotation
    node.lft = lft.rgt
    lft.rgt = node
    return lft                                  # Switch pointer from parent

def split(node):                                # Left rotation & level incr.
    if None in [node, node.rgt, node.rgt.rgt]: return node
    if node.rgt.rgt.lvl != node.lvl: return node
    rgt = node.rgt
    node.rgt = rgt.lft
    rgt.lft = node
    rgt.lvl += 1                                # This has moved up
```

```
    return rgt                                   # This should be pointed to

def aa_insert(node, key, val):
    if node is None: return AANode(key, val)
    if node.key == key: node.val = val
    elif key < node.key:
        node.lft = aa_insert(node.lft, key, val)
    else:
        node.rgt = aa_insert(node.rgt, key, val)
    node = skew(node)                            # In case it's backward
    node = split(node)                           # In case it's overfull
    return node

# ----------------------------------------------------------------------------

def test_partition_and_select():
    """
    >>> seq = [3, 4, 1, 6, 3, 7, 9, 13, 93, 0, 100, 1, 2, 2, 3, 3, 2]
    >>> partition(seq)
    ([1, 3, 0, 1, 2, 2, 3, 3, 2], 3, [4, 6, 7, 9, 13, 93, 100])
    >>> select([5, 3, 2, 7, 1], 3)
    5
    >>> select([5, 3, 2, 7, 1], 4)
    7
    >>> ans = [select(seq, k) for k in range(len(seq))]
    >>> seq.sort()
    >>> ans == seq
    True
    """

def partition(seq):
    pi, seq = seq[0], seq[1:]                    # Pick and remove the pivot
    lo = [x for x in seq if x <= pi]             # All the small elements
    hi = [x for x in seq if x > pi]              # All the large ones
    return lo, pi, hi                            # pi is "in the right place"

def select(seq, k):
    lo, pi, hi = partition(seq)                  # [<= pi], pi, [> pi]
    m = len(lo)
    if m == k: return pi                         # We found the kth smallest
    elif m < k:                                  # Too far to the left
        return select(hi, k-m-1)                 # Remember to adjust k
    else:                                        # Too far to the right
        return select(lo, k)                     # Just use original k here

def test_quicksort():
    """
    >>> seq = [7, 5, 0, 6, 3, 4, 1, 9, 8, 2]
    >>> quicksort(seq)
    [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
    """

def quicksort(seq):
    if len(seq) <= 1: return seq                 # Base case
    lo, pi, hi = partition(seq)                  # pi is in its place
    return quicksort(lo) + [pi] + quicksort(hi) # Sort lo and hi separately

def test_mergesort():
    """
    >>> seq = [7, 5, 0, 6, 3, 4, 1, 9, 8, 2]
    >>> mergesort(seq)
    [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
    """

# Mergesort, repeated from Chapter 3 (with some modifications)
def mergesort(seq):
    mid = len(seq)//2                            # Midpoint for division
    lft, rgt = seq[:mid], seq[mid:]
    if len(lft) > 1: lft = mergesort(lft)        # Sort by halves
    if len(rgt) > 1: rgt = mergesort(rgt)
    res = []
    while lft and rgt:                           # Neither half is empty
```

```
        if lft[-1] >= rgt[-1]:                  # lft has greatest last value
            res.append(lft.pop())               # Append it
        else:                                   # rgt has greatest last value
            res.append(rgt.pop())               # Append it
    res.reverse()                               # Result is backward
    return (lft or rgt) + res                   # Also add the remainder

def test_slice():
    """
    >>> A = [-1, 2, 1, 0, 4, -3, -6, 1]
    >>> n = len(A)
    >>> max((A[i:j] for i in range(n) for j in range(i+1,n+1)), key=sum)
    [2, 1, 0, 4]
    >>> best = A[0] # A valid solution
    >>> for size in range(1,n+1):
    ...     cur = sum(A[:size])
    ...     for i in range(n-size):
    ...         cur += A[i+size] - A[i]
    ...         best = max(best, cur)
    ...
    >>> best
    7
    """

def test_dsu_bisect():
    """
    >>> from bisect import bisect_left
    >>> seq = "I aim to misbehave".split()
    >>> dec = sorted((len(x), x) for x in seq)
    >>> dec[bisect_left(dec, (3, ""))][1]
    'aim'
    """

def old_test_dsu_bisect():
    """
    >>> from bisect import bisect_left
    >>> seq = "I aim to misbehave".split()
    >>> dec = sorted((len(x), x) for x in seq)
    >>> keys = [k for (k, v) in dec]
    >>> vals = [v for (k, v) in dec]
    >>> vals[bisect_left(keys, 3)]
    'aim'
    >>> vals[bisect_left(keys, 2)]
    'to'
    """

# CHAPTER-7
-----------

def greedy(E, S, w):
    T = []                                      # Emtpy, partial solution
    for e in sorted(E, key=w):                  # Greedily consider elements
        TT = T + [e]                            # Tentative solution
        if TT in S: T = TT                      # Is it valid? Use it!
    return T

def test_making_change():
    """
    >>> denom = [10000, 5000, 2000, 1000, 500, 200, 100, 50, 25, 10, 5, 1]
    >>> owed = 56329
    >>> payed = []
    >>> for d in denom:
    ...     while owed >= d:
    ...         owed -= d
    ...         payed.append(d)
    ...
    >>> sum(payed)
    56329
    >>> len(payed)
    14
    """
```

```
def test_huffman():
    """
    >>> seq = "abcdefghi"
    >>> frq = [4, 5, 6, 9, 11, 12, 15, 16, 20]
    >>> huffman(seq, frq)
    [['i', [['a', 'b'], 'e']], [['f', 'g'], [['c', 'd'], 'h']]]
    >>> C = dict(codes(_))
    >>> C['i'], C['a'], C['c']
    ('00', '0100', '1100')
    """

def codes(tree, prefix=""):
    if len(tree) == 1:
        yield (tree, prefix)                    # A leaf with its code
        return
    for bit, child in zip("01", tree):          # Left (0) and right (1)
        for pair in codes(child, prefix + bit): # Get codes recursively
            yield pair

from heapq import heapify, heappush, heappop
from itertools import count

def huffman(seq, frq):
    num = count()
    trees = list(zip(frq, num, seq))            # num ensures valid ordering
    heapify(trees)                              # A min-heap based on freq
    while len(trees) > 1:                       # Until all are combined
        fa, _, a = heappop(trees)               # Get the two smallest trees
        fb, _, b = heappop(trees)
        n = next(num)
        heappush(trees, (fa+fb, n, [a, b]))     # Combine and re-add them
    return trees[0][-1]

def test_naive_kruskal():
    """
    >>> G = {
    ...   0: {1:1, 2:3, 3:4},
    ...   1: {2:5},
    ...   2: {3:2},
    ...   3: set()
    ... }
    >>> list(naive_kruskal(G))
    [(0, 1), (2, 3), (0, 2)]
    """

def naive_find(C, u):                           # Find component rep.
    while C[u] != u:                            # Rep. would point to itself
        u = C[u]
    return u

def naive_union(C, u, v):
    u = naive_find(C, u)                        # Find both reps
    v = naive_find(C, v)
    C[u] = v                                    # Make one refer to the other

def naive_kruskal(G):
    E = [(G[u][v],u,v) for u in G for v in G[u]]
    T = set()                                   # Empty partial solution
    C = {u:u for u in G}                        # Component reps
    for _, u, v in sorted(E):                   # Edges, sorted by weight
        if naive_find(C, u) != naive_find(C, v):
            T.add((u, v))                       # Different reps? Use it!
            naive_union(C, u, v)                # Combine components
    return T

def test_kruskal():
    """
    >>> G = {
    ...   0: {1:1, 2:3, 3:4},
    ...   1: {2:5},
    ...   2: {3:2},
    ...   3: set()
```

```
    ... }
    >>> list(kruskal(G))
    [(0, 1), (2, 3), (0, 2)]
    """

def find(C, u):
    if C[u] != u:
        C[u] = find(C, C[u])                    # Path compression
    return C[u]

def union(C, R, u, v):
    u, v = find(C, u), find(C, v)
    if R[u] > R[v]:                             # Union by rank
        C[v] = u
    else:
        C[u] = v
    if R[u] == R[v]:                            # A tie: Move v up a level
        R[v] += 1

def kruskal(G):
    E = [(G[u][v],u,v) for u in G for v in G[u]]
    T = set()
    C, R = {u:u for u in G}, {u:0 for u in G}   # Comp. reps and ranks
    for _, u, v in sorted(E):
        if find(C, u) != find(C, v):
            T.add((u, v))
            union(C, R, u, v)
    return T

def test_prim():
    """
    >>> G = {
    ...   0: {1:1, 2:3, 3:4},
    ...   1: {0:1, 2:5},
    ...   2: {0:3, 1:5, 3:2},
    ...   3: {2:2, 0:4}
    ... }
    >>> prim(G, 0)
    {0: None, 1: 0, 2: 0, 3: 2}
    """

from heapq import heappop, heappush

def prim(G, s):
    P, Q = {}, [(0, None, s)]
    while Q:
        _, p, u = heappop(Q)
        if u in P: continue
        P[u] = p
        for v, w in G[u].items():
            heappush(Q, (w, u, v))
    return P

# CHAPTER-8
-----------
__meta__ = type

# ---------------------------------------------------------------------------

from functools import wraps

def memo(func):
    cache = {}                                  # Stored subproblem solutions
    @wraps(func)                                # Make wrap look like func
    def wrap(*args):                            # The memoized wrapper
        if args not in cache:                   # Not already computed?
            cache[args] = func(*args)           # Compute & cache the solution
        return cache[args]                      # Return the cached solution
    return wrap                                 # Return the wrapper

def test_memo():
    """
```

```
    >>> @memo
    ... def fib(i):
    ...     if i < 2: return 1
    ...     return fib(i-1) + fib(i-2)
    ...
    >>> fib(10)
    89
    >>> #fib = memo(fib)
    >>> print(fib(100)) # Avoid the L suffix in 2.7
    573147844013817084101
    >>> @memo
    ... def two_pow(i):
    ...     if i == 0: return 1
    ...     return two_pow(i-1) + two_pow(i-1)
    ...
    >>> two_pow(10)
    1024
    >>> print(two_pow(100))
    1267650600228229401496703205376
    >>> def two_pow(i):
    ...     if i == 0: return 1
    ...     return 2*two_pow(i-1)
    ...
    >>> two_pow(10)
    1024
    >>> print(two_pow(100))
    1267650600228229401496703205376
    """

# ----------------------------------------------------------------------------

from itertools import combinations

def naive_lis(seq):
    for length in range(len(seq), 0, -1):       # n, n-1, ... , 1
        for sub in combinations(seq, length):   # Subsequences of given length
            if list(sub) == sorted(sub):        # An increasing subsequence?
                return sub                      # Return it!

def test_lis():
    """
    >>> seq = [3, 1, 0, 2, 4]
    >>> naive_lis(seq)
    (1, 2, 4)
    >>> rec_lis(seq)
    3
    >>> basic_lis(seq)
    3
    >>> lis(seq)
    3
    >>> naive_lis([1, 0, 7, 2, 8, 3, 4, 9, 5, 6])
    (1, 2, 3, 4, 5, 6)
    >>> from random import *
    >>> seqs = [[randrange(100) for i in range(5+i)] for i in range(10)]
    >>> seqs.append([1, 1, 1, 1, 2, 2, 2, 2, 0, 0, 0, 4, 3, 3, 3, 4, 4, 4])
    >>> for seq in seqs:
    ...     res = naive_lis(seq)
    ...     for f in [basic_lis, rec_lis, lis]:
    ...         res2 = f(seq)
    ...         assert res2 == len(res), (res, seq, res2, f)
    """

def rec_lis(seq):                               # Longest increasing subseq.
    @memo
    def L(cur):                                 # Longest ending at seq[cur]
        res = 1                                 # Length is at least 1
        for pre in range(cur):                  # Potential predecessors
            if seq[pre] <= seq[cur]:            # A valid (smaller) predec.
                res = max(res, 1 + L(pre))      # Can we improve the solution?
        return res
    return max(L(i) for i in range(len(seq)))   # The longest of them all
```

```
def basic_lis(seq):
    L = [1] * len(seq)
    for cur, val in enumerate(seq):
        for pre in range(cur):
            if seq[pre] <= val:
                L[cur] = max(L[cur], 1 + L[pre])
    return max(L)

from bisect import bisect

def lis(seq):                               # Longest increasing subseq.
    end = []                                # End-values for all lengths
    for val in seq:                         # Try every value, in order
        idx = bisect(end, val)              # Can we build on an end val?
        if idx == len(end): end.append(val) # Longest seq. extended
        else: end[idx] = val                # Prev. endpoint reduced
    return len(end)                         # The longest we found

DAG = {
    'a': {'b':0},
    'b': {'c':4, 'd':6},
    'c': {'g':2, 'h':-6},
    'd': {'f':3, 'e':5},
    'e': {'g':0, 'h':-6},
    'f': {'i':-1},
    'g': {'h':4},
    'h': {'i':7},
    'i': {}
}

def test_dag_sp():
    """
    >>> rec_dag_sp(DAG, 'a', 'i')
    5
    >>> dag_sp(DAG, 'a', 'i')
    5
    """

def rec_dag_sp(W, s, t):                    # Shortest path from s to t
    @memo                                   # Memoize f
    def d(u):                               # Distance from u to t
        if u == t: return 0                 # We're there!
        return min(W[u][v]+d(v) for v in W[u])  # Best of every first step
    return d(s)                             # Apply f to actual start node

# From Chapter 4:
def topsort(G):
    count = dict((u, 0) for u in G)         # The in-degree for each node
    for u in G:
        for v in G[u]:
            count[v] += 1                   # Count every in-edge
    Q = [u for u in G if count[u] == 0]     # Valid initial nodes
    S = []                                  # The result
    while Q:                                # While we have start nodes...
        u = Q.pop()                         # Pick one
        S.append(u)                         # Use it as first of the rest
        for v in G[u]:
            count[v] -= 1                   # "Uncount" its out-edges
            if count[v] == 0:               # New valid start nodes?
                Q.append(v)                 # Deal with them next
    return S

def dag_sp(W, s, t):                        # Shortest path from s to t
    d = {u:float('inf') for u in W}         # Distance estimates
    d[s] = 0                                # Start node: Zero distance
    for u in topsort(W):                    # In top-sorted order...
        if u == t: break                    # Have we arrived?
        for v in W[u]:                      # For each out-edge ...
            d[v] = min(d[v], d[u] + W[u][v])    # Relax the edge
    return d[t]                             # Distance to t (from s)

# ----------------------------------------------------------------------------
```

```
def test_c():
    """
    >>> @memo
    ... def C(n,k):
    ...     if k == 0: return 1
    ...     if n == 0: return 0
    ...     return C(n-1,k-1) + C(n-1,k)
    >>> C(4,2)
    6
    >>> print(C(100,50))
    100891344545564193334812497256
    >>> C(10,7)
    120
    >>> C(4, 4)
    1
    >>> C(4, 5)
    0
    """

def test_c2():
    """
    >>> from collections import defaultdict
    >>> n, k = 10, 7
    >>> C = defaultdict(int)
    >>> for row in range(n+1):
    ...     C[row,0] = 1
    ...     for col in range(1,k+1):
    ...         C[row,col] = C[row-1,col-1] + C[row-1,col]
    ...
    >>> C[n,k]
    120
    """

# ---------------------------------------------------------------------------

def test_lcs():
    """
    >>> rec_lcs("spock", "asoka")
    3
    >>> rec_lcs("AGCGA", "CAGATAGAG")
    4
    >>> rec_lcs("Starbuck", "Starwalker")
    5
    >>> lcs("spock", "asoka")
    3
    >>> lcs("AGCGA", "CAGATAGAG")
    4
    >>> lcs("Starbuck", "Starwalker")
    5
    """

def rec_lcs(a,b):                               # Longest common subsequence
    @memo                                       # L is memoized
    def L(i,j):                                 # Prefixes a[:i] and b[:j]
        if min(i,j) < 0: return 0               # One prefix is empty
        if a[i] == b[j]: return 1 + L(i-1,j-1)  # Match! Move diagonally
        return max(L(i-1,j), L(i,j-1))          # Chop off either a[i] or b[j]
    return L(len(a)-1,len(b)-1)                 # Run L on entire sequences

def lcs(a,b):
    n, m = len(a), len(b)
    pre, cur = [0]*(n+1), [0]*(n+1)             # Previous/current row
    for j in range(1,m+1):                      # Iterate over b
        pre, cur = cur, pre                     # Keep prev., overwrite cur.
        for i in range(1,n+1):                  # Iterate over a
            if a[i-1] == b[j-1]:                # Last elts. of pref. equal?
                cur[i] = pre[i-1] + 1           # L(i,j) = L(i-1,j-1) + 1
            else:                               # Otherwise...
                cur[i] = max(pre[i], cur[i-1])  # max(L(i,j-1),L(i-1,j))
    return cur[n]                               # L(n,m)
```

```
# -----------------------------------------------------------------------------

def test_knapsack():
    """
    >>> funcs = [brutish_knapsack, old_rec_knapsack, rec_knapsack,
    ... knapsack]
    >>> cases = [
    ...    #[[2, 4, 3, 6, 5], [2, 4, 3, 6, 6], 12, -1],
    ...    [[2, 3, 4, 5], [3, 4, 5, 6], 5, 7]
    ... ]
    >>> from random import *
    >>> for i in range(20):
    ...     n = randrange(10)
    ...     w = [randrange(100) for i in range(n)]
    ...     v = [randrange(100) for i in range(n)]
    ...     W = randrange(sum(w)+1)
    ...     cases.append([w, v, W, -1])
    >>> for w, v, W, e in cases:
    ...     sols = set(f(w, v, W) for f in funcs)
    ...     assert len(sols) == 1, (w, v, W, e, sols)
    ...     if e >= 0: assert sols.pop() == e
    ...
    >>>
    """

# Not used -- just for testing:
def brutish_knapsack(w, v, W):
    items = list(range(len(w)))
    vals = [0]
    for r in range(1,len(items)+1):
        for subset in combinations(items, r):
            wt = sum(w[x] for x in subset)
            if wt <= W: vals.append(sum(v[x] for x in subset))
    return max(vals)

def rec_knapsack(w, v, c):                      # Weights, values and capacity
    @memo                                       # m is memoized
    def m(k, r):                                # Max val., k objs and cap r
        if k == 0 or r == 0: return 0           # No objects/no capacity
        i = k-1                                 # Object under consideration
        drop = m(k-1, r)                        # What if we drop the object?
        if w[i] > r: return drop                # Too heavy: Must drop it
        return max(drop, v[i] + m(k-1, r-w[i])) # Include it? Max of in/out
    return m(len(w), c)                         # All objects, all capacity

def old_rec_knapsack(w, v, c):                  # Weights, values and capacity
    @memo                                       # m is memoized
    def m(i, r):                                # Max val., obj 0..i and cap r
        if i == -1 or r == 0: return 0          # No objects/no capacity
        drop = m(i-1, r)                        # What if we drop object i?
        if w[i] > r: return drop                # Too heavy: Must drop it
        return max(drop, v[i] + m(i-1, r-w[i])) # Include it? Max of in/out
    return m(len(w)-1, c)                       # All objects, all capacity

def knapsack_old(w, v, c):
    n = len(w)
    m = [[0]*(c+1) for i in range(n+1)]
    for k in range(1,n+1):
        i = k-1
        for r in range(1,c+1):
            m[k][r] = drop = m[k-1][r]
            if w[i] <= r:
                m[k][r] = max(drop, v[i] + m[k-1][r-w[i]])
    return m[n][c]

def knapsack_wrap(w, v, c):
    return knapsack_inner(w, v, c)[0][len(w)][c]

def test_knapsack_items():
    """
    >>> knapsack = knapsack_inner
    >>> w, v, c = [2, 3, 4, 5], [3, 4, 5, 6], 5
```

```
    >>> m, P = knapsack(w, v, c)
    >>> k, r, items = len(w), c, set()
    >>> while k > 0 and r > 0:
    ...     i = k-1
    ...     if P[k][r]:
    ...         items.add(i)
    ...         r -= w[i]
    ...     k -= 1
    ...
    >>> sorted(items)
    [0, 1]
    """

def knapsack(w, v, c):                          # Returns solution matrices
    n = len(w)                                  # Number of available items
    m = [[0]*(c+1) for i in range(n+1)]         # Empty max-value matrix
    P = [[False]*(c+1) for i in range(n+1)]     # Empty keep/drop matrix
    for k in range(1,n+1):                      # We can use k first objects
        i = k-1                                 # Object under consideration
        for r in range(1,c+1):                  # Every positive capacity
            m[k][r] = drop = m[k-1][r]          # By default: drop the object
            if w[i] > r: continue               # Too heavy? Ignore it
            keep = v[i] + m[k-1][r-w[i]]        # Value of keeping it
            m[k][r] = max(drop, keep)           # Best of dropping and keeping
            P[k][r] = keep > drop               # Did we keep it?
    return m, P                                 # Return full results

knapsack_inner = knapsack
knapsack = knapsack_wrap

def test_unbounded_knapsack():
    """
    >>> funcs = [rec_unbounded_knapsack, unbounded_knapsack]
    >>> w, v = [1, 2], [2, 5]
    >>> [f(w, v, 5) for f in funcs]
    [12, 12]
    >>> w, v = [3, 2, 4], [5, 4, 2]
    >>> [f(w, v, 7) for f in funcs]
    [13, 13]
    """

def rec_unbounded_knapsack(w, v, c):            # Weights, values and capacity
    @memo                                       # m is memoized
    def m(r):                                   # Max val. w/remaining cap. r
        if r == 0: return 0                     # No capacity? No value
        val = m(r-1)                            # Ignore the last cap. unit?
        for i, wi in enumerate(w):              # Try every object
            if wi > r: continue                 # Too heavy? Ignore it
            val = max(val, v[i] + m(r-wi))      # Add value, remove weight
        return val                              # Max over all last objects
    return m(c)                                 # Full capacity available

def unbounded_knapsack(w, v, c):
    m = [0]
    for r in range(1,c+1):
        val = m[r-1]
        for i, wi in enumerate(w):
            if wi > r: continue
            val = max(val, v[i] + m[r-wi])
        m.append(val)
    return m[c]

# ----------------------------------------------------------------------------

def test_opt_tree():
    """
    >>> w = [0.25, 0.2, 0.05, 0.2, 0.3]
    >>> rec_opt_tree(w)
    2.1
    >>> opt_tree(w)
    2.1
    >>> from random import *
```

```
    >>> ws = [[random() for i in range(randrange(4,9))] for j in range(20)]
    >>> for w in ws:
    ...     assert rec_opt_tree(w) == opt_tree(w)
    """

def rec_opt_tree(p):
    @memo
    def s(i,j):
        if i == j: return 0
        return s(i,j-1) + p[j-1]
    @memo
    def e(i,j):
        if i == j: return 0
        sub = min(e(i,r) + e(r+1,j) for r in range(i,j))
        return sub + s(i,j)
    return e(0,len(p))

from collections import defaultdict

def opt_tree(p):
    n = len(p)
    s, e = defaultdict(int), defaultdict(int)
    for l in range(1,n+1):
        for i in range(n-l+1):
            j = i + l
            s[i,j] = s[i,j-1] + p[j-1]
            e[i,j] = min(e[i,r] + e[r+1,j] for r in range(i,j))
            e[i,j] += s[i,j]
    return e[0,n]

# CHAPTER-9
-----------
def test_relax():
    """
    >>> u = 0; v = 1
    >>> D, W, P = {}, {u:{v:3}}, {}
    >>> D[u] = 7
    >>> D[v] = 13
    >>> D[u]
    7
    >>> D[v]
    13
    >>> W[u][v]
    3
    >>> relax(W, u, v, D, P)
    True
    >>> D[v]
    10
    >>> D[v] = 8
    >>> relax(W, u, v, D, P)
    >>> D[v]
    8
    """

inf = float('inf')
def relax(W, u, v, D, P):
    d = D.get(u,inf) + W[u][v]                  # Possible shortcut estimate
    if d < D.get(v,inf):                        # Is it really a shortcut?
        D[v], P[v] = d, u                       # Update estimate and parent
        return True                             # There was a change!

def test_bellman_ford():
    """
    >>> s, t, x, y, z = range(5)
    >>> W = {
    ...     s: {t:6, y:7},
    ...     t: {x:5, y:8, z:-4},
    ...     x: {t:-2},
    ...     y: {x:-3, z:9},
    ...     z: {s:2, x:7}
    ... }
    >>> D, P = bellman_ford(W, s)
```

```
    >>> [D[v] for v in [s, t, x, y, z]]
    [0, 2, 4, 7, -2]
    >>> s not in P
    True
    >>> [P[v] for v in [t, x, y, z]] == [x, y, s, t]
    True
    >>> W[s][t] = -100
    >>> bellman_ford(W, s)
    Traceback (most recent call last):
            ...
    ValueError: negative cycle
    """

def bellman_ford(G, s):
    D, P = {s:0}, {}                            # Zero-dist to s; no parents
    for rnd in G:                               # n = len(G) rounds
        changed = False                         # No changes in round so far
        for u in G:                             # For every from-node...
            for v in G[u]:                      # ... and its to-nodes...
                if relax(G, u, v, D, P):        # Shortcut to v from u?
                    changed = True              # Yes! So something changed
        if not changed: break                   # No change in round: Done
    else:                                       # Not done before round n?
        raise ValueError('negative cycle')      # Negative cycle detected
    return D, P                                 # Otherwise: D and P correct


def test_dijkstra():
    """
    >>> s, t, x, y, z = range(5)
    >>> W = {
    ...     s: {t:10, y:5},
    ...     t: {x:1, y:2},
    ...     x: {z:4},
    ...     y: {t:3, x:9, z:2},
    ...     z: {x:6, s:7}
    ... }
    >>> D, P = dijkstra(W, s)
    >>> [D[v] for v in [s, t, x, y, z]]
    [0, 8, 9, 5, 7]
    >>> s not in P
    True
    >>> [P[v] for v in [t, x, y, z]] == [y, t, s, y]
    True
    """

from heapq import heappush, heappop

def dijkstra(G, s):
    D, P, Q, S = {s:0}, {}, [(0,s)], set()      # Est., tree, queue, visited
    while Q:                                    # Still unprocessed nodes?
        _, u = heappop(Q)                       # Node with lowest estimate
        if u in S: continue                     # Already visited? Skip it
        S.add(u)                                # We've visited it now
        for v in G[u]:                          # Go through all its neighbors
            relax(G, u, v, D, P)                # Relax the out-edge
            heappush(Q, (D[v], v))              # Add to queue, w/est. as pri
    return D, P                                 # Final D and P returned

def test_johnson():
    """
    >>> a, b, c, d, e = range(5)
    >>> W = {
    ...     a: {c:1, d:7},
    ...     b: {a:4},
    ...     c: {b:-5, e:2},
    ...     d: {c:6},
    ...     e: {a:3, b:8, d:-4}
    ... }
    >>> D, P = johnson(W)
    >>> [D[a][v] for v in [a, b, c, d, e]]
    [0, -4, 1, -1, 3]
```

```
    >>> [D[b][v] for v in [a, b, c, d, e]]
    [4, 0, 5, 3, 7]
    >>> [D[c][v] for v in [a, b, c, d, e]]
    [-1, -5, 0, -2, 2]
    >>> [D[d][v] for v in [a, b, c, d, e]]
    [5, 1, 6, 0, 8]
    >>> [D[e][v] for v in [a, b, c, d, e]]
    [1, -3, 2, -4, 0]
    """

from copy import deepcopy

def johnson(G):                                 # All pairs shortest paths
    G = deepcopy(G)                             # Don't want to break original
    s = object()                                # Guaranteed unique node
    G[s] = {v:0 for v in G}                     # Edges from s have zero wgt
    h, _ = bellman_ford(G, s)                   # h[v]: Shortest dist from s
    del G[s]                                    # No more need for s
    for u in G:                                 # The weight from u...
        for v in G[u]:                          # ... to v...
            G[u][v] += h[u] - h[v]              # ... is adjusted (nonneg.)
    D, P = {}, {}                               # D[u][v] and P[u][v]
    for u in G:                                 # From every u...
        D[u], P[u] = dijkstra(G, u)             # ... find the shortest paths
        for v in G:                             # For each destination...
            D[u][v] += h[v] - h[u]              # ... readjust the distance
    return D, P                                 # These are two-dimensional

from ch_08 import memo

def test_rec_floyd_warshall():
    """
    >>> a, b, c, d, e = range(1,6) # One-based
    >>> W = {
    ...     a: {c:1, d:7},
    ...     b: {a:4},
    ...     c: {b:-5, e:2},
    ...     d: {c:6},
    ...     e: {a:3, b:8, d:-4}
    ... }
    >>> for u in W:
    ...     for v in W:
    ...         if u == v: W[u][v] = 0
    ...         if v not in W[u]: W[u][v] = inf
    >>> D = rec_floyd_warshall(W)
    >>> [D[a,v] for v in [a, b, c, d, e]]
    [0, -4, 1, -1, 3]
    >>> [D[b,v] for v in [a, b, c, d, e]]
    [4, 0, 5, 3, 7]
    >>> [D[c,v] for v in [a, b, c, d, e]]
    [-1, -5, 0, -2, 2]
    >>> [D[d,v] for v in [a, b, c, d, e]]
    [5, 1, 6, 0, 8]
    >>> [D[e,v] for v in [a, b, c, d, e]]
    [1, -3, 2, -4, 0]
    """

def rec_floyd_warshall(G):                                # All shortest paths
    @memo                                                 # Store subsolutions
    def d(u,v,k):                                         # u to v via 1..k
        if k==0: return G[u][v]                           # Assumes v in G[u]
        return min(d(u,v,k-1), d(u,k,k-1) + d(k,v,k-1))   # Use k or not?
    return {(u,v): d(u,v,len(G)) for u in G for v in G}   # D[u,v] = d(u,v,n)

def test_floyd_warshall1():
    """
    >>> a, b, c, d, e = range(1,6) # One-based
    >>> W = {
    ...     a: {c:1, d:7},
    ...     b: {a:4},
    ...     c: {b:-5, e:2},
    ...     d: {c:6},
```

```
    ...     e: {a:3, b:8, d:-4}
    ... }
    >>> for u in W:
    ...     for v in W:
    ...         if u == v: W[u][v] = 0
    ...         if v not in W[u]: W[u][v] = inf
    >>> D = floyd_warshall1(W)
    >>> [D[a][v] for v in [a, b, c, d, e]]
    [0, -4, 1, -1, 3]
    >>> [D[b][v] for v in [a, b, c, d, e]]
    [4, 0, 5, 3, 7]
    >>> [D[c][v] for v in [a, b, c, d, e]]
    [-1, -5, 0, -2, 2]
    >>> [D[d][v] for v in [a, b, c, d, e]]
    [5, 1, 6, 0, 8]
    >>> [D[e][v] for v in [a, b, c, d, e]]
    [1, -3, 2, -4, 0]
    """

def floyd_warshall1(G):
    D = deepcopy(G)                             # No intermediates yet
    for k in G:                                 # Look for shortcuts with k
        for u in G:
            for v in G:
                D[u][v] = min(D[u][v], D[u][k] + D[k][v])
    return D

def test_floyd_warshall():
    """
    >>> a, b, c, d, e = range(5)
    >>> W = {
    ...     a: {c:1, d:7},
    ...     b: {a:4},
    ...     c: {b:-5, e:2},
    ...     d: {c:6},
    ...     e: {a:3, b:8, d:-4}
    ... }
    >>> for u in W:
    ...     for v in W:
    ...         if u == v: W[u][v] = 0
    ...         if v not in W[u]: W[u][v] = inf
    >>> D, P = floyd_warshall(W)
    >>> [D[a][v] for v in [a, b, c, d, e]]
    [0, -4, 1, -1, 3]
    >>> [D[b][v] for v in [a, b, c, d, e]]
    [4, 0, 5, 3, 7]
    >>> [D[c][v] for v in [a, b, c, d, e]]
    [-1, -5, 0, -2, 2]
    >>> [D[d][v] for v in [a, b, c, d, e]]
    [5, 1, 6, 0, 8]
    >>> [D[e][v] for v in [a, b, c, d, e]]
    [1, -3, 2, -4, 0]
    >>> [P[a,v] for v in [a, b, c, d, e]]
    [None, 2, 0, 4, 2]
    >>> [P[b,v] for v in [a, b, c, d, e]]
    [1, None, 0, 4, 2]
    >>> [P[c,v] for v in [a, b, c, d, e]]
    [1, 2, None, 4, 2]
    >>> [P[d,v] for v in [a, b, c, d, e]]
    [1, 2, 3, None, 2]
    >>> [P[e,v] for v in [a, b, c, d, e]]
    [1, 2, 3, 4, None]
    """

def floyd_warshall(G):
    D, P = deepcopy(G), {}
    for u in G:
        for v in G:
            if u == v or G[u][v] == inf:
                P[u,v] = None
            else:
                P[u,v] = u
```

```
    for k in G:
        for u in G:
            for v in G:
                shortcut = D[u][k] + D[k][v]
                if shortcut < D[u][v]:
                    D[u][v] = shortcut
                    P[u,v] = P[k,v]
    return D, P

def test_idijkstra():
    """
    >>> s, t, x, y, z = range(5)
    >>> W = {
    ...     s: {t:10, y:5},
    ...     t: {x:1, y:2},
    ...     x: {z:4},
    ...     y: {t:3, x:9, z:2},
    ...     z: {x:6, s:7}
    ... }
    >>> D = dict(idijkstra(W, s))
    >>> [D[v] for v in [s, t, x, y, z]]
    [0, 8, 9, 5, 7]
    """

def idijkstra(G, s):
    Q, S = [(0,s)], set()                   # Queue w/dists, visited
    while Q:                                # Still unprocessed nodes?
        d, u = heappop(Q)                   # Node with lowest estimate
        if u in S: continue                 # Already visited? Skip it
        S.add(u)                            # We've visited it now
        yield u, d                          # Yield a subsolution/node
        for v in G[u]:                      # Go through all its neighbors
            heappush(Q, (d+G[u][v], v))     # Add to queue, w/est. as pri

def test_bidir_dijkstra_et_al():
    """
    >>> W = {
    ...     'hnl': {'lax':2555},
    ...     'lax': {'sfo':337, 'ord':1743, 'dfw': 1233},
    ...     'sfo': {'ord':1843},
    ...     'dfw': {'ord':802, 'lga':1387, 'mia':1120},
    ...     'ord': {'pvd':849},
    ...     'lga': {'pvd':142},
    ...     'mia': {'lga':1099, 'pvd':1205}
    ... }
    >>> nodes = list(W)
    >>> for u in nodes:
    ...     for v in W[u]:
    ...         if not v in W: W[v] = {}
    ...         W[v][u] = W[u][v]
    ...
    >>> for u in W:
    ...     W[u][u] = 0
    ...
    >>> for u in W:
    ...     for v in W[u]:
    ...         assert W[u][v] == W[v][u]
    ...
    >>> for u in W:
    ...     Dd, _ = dijkstra(W, u)
    ...     Db, _ = bellman_ford(W, u)
    ...     for v in W:
    ...         d = bidir_dijkstra(W, u, v)
    ...         assert d == Dd[v], (d, Dd[v])
    ...         assert d == Db[v], (d, Db[v])
    ...         a = a_star_wrap(W, u, v, lambda v: 0)
    ...         assert a == d
    ...
    >>> G = {0:{0:0}, 1:{1:0}}
    >>> bidir_dijkstra(G, 0, 1)
    inf
    >>> bidir_dijkstra(G, 0, 0)
```

```
    0
    >>> G = {0:{1:7}, 1:{0:7}}
    >>> bidir_dijkstra(G, 0, 1)
    7
    >>> bidir_dijkstra(G, 0, 1)
    7
    >>> D, P = dijkstra(W, 'hnl')
    >>> P['pvd'], P['ord'], P['lax']
    ('ord', 'lax', 'hnl')
    >>> D['pvd'] == W['hnl']['lax'] + W['lax']['ord'] + W['ord']['pvd']
    True
    >>> D['pvd']
    5147
    >>> bidir_dijkstra(W, 'hnl', 'pvd')
    5147
    >>> bidir_dijkstra(W, 'pvd', 'sfo')
    2692
    """

from itertools import cycle

def bidir_dijkstra(G, s, t):
    Ds, Dt = {}, {}                             # D from s and t, respectively
    forw, back = idijkstra(G,s), idijkstra(G,t) # The "two Dijkstras"
    dirs = (Ds, Dt, forw), (Dt, Ds, back)       # Alternating situations
    try:                                        # Until one of forw/back ends
        for D, other, step in cycle(dirs):      # Switch between the two
            v, d = next(step)                   # Next node/distance for one
            D[v] = d                            # Demember the distance
            if v in other: break                # Also visite by the other?
    except StopIteration: return inf            # One ran out before they met
    m = inf                                     # They met; now find the path
    for u in Ds:                                # For every visited forw-node
        for v in G[u]:                          # ... go through its neighbors
            if not v in Dt: continue            # Is it also back-visited?
            m = min(m, Ds[u] + G[u][v] + Dt[v]) # Is this path better?
    return m                                    # Return the best path

def a_star(G, s, t, h):
    P, Q = {}, [(h(s), None, s)]                # Pred and queue w/heuristic
    while Q:                                    # Still unprocessed nodes?
        d, p, u = heappop(Q)                    # Node with lowest heuristic
        if u in P: continue                     # Already visited? Skip it
        P[u] = p                                # Set path predecessor
        if u == t: return d - h(t), P           # Arrived! Ret. dist and preds
        for v in G[u]:                          # Go through all neighbors
            w = G[u][v] - h(u) + h(v)           # Modify weight wrt heuristic
            heappush(Q, (d + w, u, v))          # Add to queue, w/heur as pri
    return inf, None                            # Didn't get to t

def a_star_wrap(G, s, t, h):
    return a_star(G, s, t, h)[0]

from string import ascii_lowercase as chars

class WordSpace:                                # An implicit graph w/utils

    def __init__(self, words):                  # Create graph over the words
        self.words = words
        self.M = M = dict()                     # Reachable words

    def variants(self, wd, words):              # Yield all word variants
        wasl = list(wd)                         # The word as a list
        for i, c in enumerate(wasl):            # Each position and character
            for oc in chars:                    # Every possible character
                if c == oc: continue            # Don't replace with the same
                wasl[i] = oc                    # Replace the character
                ow = ''.join(wasl)              # Make a string of the word
                if ow in words:                 # Is it a valid word?
                    yield ow                    # Then we yield it
            wasl[i] = c                         # Reset the character
```

```
    def __getitem__(self, wd):                  # The adjacency map interface
        if wd not in self.M:                    # Cache the neighbors
            self.M[wd] = dict.fromkeys(self.variants(wd, self.words), 1)
        return self.M[wd]

    def heuristic(self, u, v):                  # The default heuristic
        return sum(a!=b for a, b in zip(u, v))  # How many characters differ?

    def ladder(self, s, t, h=None):             # Utility wrapper for a_star
        if h is None:                           # Allows other heuristics
            def h(v):
                return self.heuristic(v, t)
        _, P = a_star(self, s, t, h)            # Get the predecessor map
        if P is None:
            return [s, None, t]                 # When no path exists
        u, p = t, []
        while u is not None:                    # Walk backward from t
            p.append(u)                         # Append every predecessor
            u = P[u]                            # Take another step
        p.reverse()                             # The path is backward
        return p

# Some test code you could use:
'''
FORBIDDEN = set("""
dal alod dol aloed algedo elod lod gol geodal dola dogal
""".split())

# This assumes that you have a dictionary in this location, of course:
wds = [line.strip().lower() for line in open("/usr/share/dict/words")]
wds = [wd for wd in wds if wd not in FORBIDDEN]

G = WordSpace(wds)
t0 = time()
print G.ladder(s, t)
print time()-t0

# Should be a lot slower:

G = WordSpace(wds)
t0 = time()
print G.ladder(s, t, h=lambda v: 0)
print time()-t0
'''

# CHAPTER-10
-----------
from ch_05 import tr

def test_match():
    """
    >>> G = {
    ... 0: {2, 3},
    ... 1: {3},
    ... 2: set(),
    ... 3: set()
    ... }
    >>> M = match(G, {0, 1}, {2, 3})
    >>> sorted(M)
    [(0, 2), (1, 3)]
    >>> G = {
    ... 0: {3, 4},
    ... 1: {3, 4},
    ... 2: {4},
    ... 3: set(),
    ... 4: set(),
    ... 5: set()
    ... }
    >>> M = match(G, {0, 1, 2}, {3, 4, 5})
    >>> sorted(M)
    [(1, 3), (2, 4)]
    """
```

```
from collections import defaultdict
from itertools import chain

def match(G, X, Y):                               # Maximum bipartite matching
    H = tr(G)                                     # The transposed graph
    S, T, M = set(X), set(Y), set()               # Unmatched left/right + match
    while S:                                      # Still unmatched on the left?
        s = S.pop()                               # Get one
        Q, P = {s}, {}                            # Start a traversal from it
        while Q:                                  # Discovered, unvisited
            u = Q.pop()                           # Visit one
            if u in T:                            # Finished augmenting path?
                T.remove(u)                       # u is now matched
                break                             # and our traversal is done
            forw = (v for v in G[u] if (u,v) not in M)  # Possible new edges
            back = (v for v in H[u] if (v,u) in M)      # Cancellations
            for v in chain(forw, back):           # Along out- and in-edges
                if v in P: continue               # Already visited? Ignore
                P[v] = u                          # Traversal predecessor
                Q.add(v)                          # New node discovered
        while u != s:                             # Augment: Backtrack to s
            u, v = P[u], u                        # Shift one step
            if v in G[u]:                         # Forward edge?
                M.add((u,v))                      # New edge
            else:                                 # Backward edge?
                M.remove((v,u))                   # Cancellation
    return M                                      # Matching -- a set of edges

FF_01_SIMPLE_GRAPH = {
    's': {'u': 1, 'x': 1},
    'u': {'v': 1},
    'v': {'x': 1, 't': 1},
    'x': {'y': 1},
    'y': {'t': 1},
    't': {}
}

def test_01_flow():
    """
    >>> G = FF_01_SIMPLE_GRAPH
    >>> paths(G, 's', 't')
    2
    """

def paths(G, s, t):                               # Edge-disjoint path count
    H, M, count = tr(G), set(), 0                 # Transpose, matching, result
    while True:                                   # Until the function returns
        Q, P = {s}, {}                            # Traversal queue + tree
        while Q:                                  # Discovered, unvisited
            u = Q.pop()                           # Get one
            if u == t:                            # Augmenting path!
                count += 1                        # That means one more path
                break                             # End the traversal
            forw = (v for v in G[u] if (u,v) not in M)  # Possible new edges
            back = (v for v in H[u] if (v,u) in M)      # Cancellations
            for v in chain(forw, back):           # Along out- and in-edges
                if v in P: continue               # Already visited? Ignore
                P[v] = u                          # Traversal predecessor
                Q.add(v)                          # New node discovered
        else:                                     # Didn't reach t?
            return count                          # We're done
        while u != s:                             # Augment: Backtrack to s
            u, v = P[u], u                        # Shift one step
            if v in G[u]:                         # Forward edge?
                M.add((u,v))                      # New edge
            else:                                 # Backward edge?
                M.remove((v,u))                   # Cancellation

FF_SIMPLE_GRAPH = {
    's': {'u': 2, 'x': 2},
    'u': {'v': 1},
```

```
    'v': {'x': 1, 't': 2},
    'x': {'y': 1},
    'y': {'t': 2},
    't': {}
}

def path_from_P(P, s, t):
    res = [t]
    u = t
    while u != s:
        u, v = P[u], u
        res.append(u)
    res.reverse()
    return res

def test_bfs_aug():
    """
    >>> G = {0: {1:1}, 1:{}}
    >>> H = tr(G)
    >>> f = defaultdict(int)
    >>> P, c = bfs_aug(G, H, 0, 1, f)
    >>> path_from_P(P, 0, 1)
    [0, 1]
    >>> c
    1
    >>> G = FF_SIMPLE_GRAPH
    >>> H = tr(G)
    >>> f = defaultdict(int)
    >>> f['s','u'] = 1
    >>> f['u','v'] = 1
    >>> f['v','x'] = 1
    >>> f['x','y'] = 1
    >>> f['y','t'] = 1
    >>> P, c = bfs_aug(G, H, 's', 't', f)
    >>> path_from_P(P, 's', 't')
    ['s', 'x', 'v', 't']
    >>> c
    1
    """

from collections import deque
inf = float('inf')

def bfs_aug(G, H, s, t, f):
    P, Q, F = {s: None}, deque([s]), {s: inf}   # Tree, queue, flow label
    def label(inc):                             # Flow increase at v from u?
        if v in P or inc <= 0: return           # Seen? Unreachable? Ignore
        F[v], P[v] = min(F[u], inc), u          # Max flow here? From where?
        Q.append(v)                             # Discovered -- visit later
    while Q:                                    # Discovered, unvisited
        u = Q.popleft()                         # Get one (FIFO)
        if u == t: return P, F[t]               # Reached t? Augmenting path!
        for v in G[u]: label(G[u][v]-f[u,v])    # Label along out-edges
        for v in H[u]: label(f[v,u])            # Label along in-edges
    return None, 0                              # No augmenting path found

def test_ford_fulkerson():
    """
    >>> G = FF_SIMPLE_GRAPH
    >>> f = ford_fulkerson(G, 's', 't')
    >>> sorted(f.items()) ==  [(('s', 'u'), 1), (('s', 'x'), 1),
    ... (('u', 'v'), 1), (('v', 't'), 1), (('v', 'x'), 0), (('x', 'y'), 1),
    ... (('y', 't'), 1)]
    True
    """

def ford_fulkerson(G, s, t, aug=bfs_aug):       # Max flow from s to t
    H, f = tr(G), defaultdict(int)              # Transpose and flow
    while True:                                 # While we can improve things
        P, c = aug(G, H, s, t, f)               # Aug. path and capacity/slack
        if c == 0: return f                     # No augm. path found? Done!
        u = t                                   # Start augmentation
```

```
        while u != s:                         # Backtrack to s
            u, v = P[u], u                    # Shift one step
            if v in G[u]: f[u,v] += c         # Forward edge? Add slack
            else:         f[v,u] -= c         # Backward edge? Cancel slack

def busacker_gowen(G, W, s, t):               # Min-cost max-flow
    def sp_aug(G, H, s, t, f):                # Shortest path (Bellman-Ford)
        D, P, F = {s:0}, {s:None}, {s:inf,t:0}  # Dist, preds and flow
        def label(inc, cst):                  # Label + relax, really
            if inc <= 0: return False         # No flow increase? Skip it
            d = D.get(u,inf) + cst            # New possible aug. distance
            if d >= D.get(v,inf): return False  # No improvement? Skip it
            D[v], P[v] = d, u                 # Update dist and pred
            F[v] = min(F[u], inc)             # Update flow label
            return True                       # We changed things!
        for rnd in G:                         # n = len(G) rounds
            changed = False                   # No changes in round so far
            for u in G:                       # Every from-node
                for v in G[u]:                # Every forward to-node
                    changed |= label(G[u][v]-f[u,v], W[u,v])
                for v in H[u]:                # Every backward to-node
                    changed |= label(f[v,u], -W[v,u])
            if not changed: break             # No change in round: Done
        else:                                 # Not done before round n?
            raise ValueError('negative cycle')  # Negative cycle detected
        return P, F[t]                        # Preds and flow reaching t
    return ford_fulkerson(G, s, t, sp_aug)    # Max-flow with Bellman-Ford

def test_busacker_gowen():
    """
    >>> G = {
    ...     0: {1:3, 2:3},
    ...     1: {3:2, 4:2},
    ...     2: {3:1, 4:2},
    ...     3: {5:2},
    ...     4: {5:2},
    ...     5: {}
    ... }
    >>> W = {
    ... (0,1): 3,
    ... (0,2): 1,
    ... (1,3): 1,
    ... (1,4): 1,
    ... (2,3): 4,
    ... (2,4): 2,
    ... (3,5): 2,
    ... (4,5): 1
    ... }
    >>> f1 = ford_fulkerson(G, 0, 5)
    >>> for u, v in W: assert f1[u,v] <= G[u][v]
    >>> f1[3,5] + f1[4,5]
    4
    >>> f1[0,1] + f1[0,2]
    4
    >>> f2 = busacker_gowen(G, W, 0, 5)
    >>> for u, v in W: assert f2[u,v] <= G[u][v]
    >>> sum(f2[key]*W[key] for key in W)
    20
    >>> fs = [f2[key] for key in sorted(W)]
    >>> fs
    [2, 2, 2, 0, 0, 2, 2, 2]
    """


# CHAPTER-11
-----------
from __future__ import division
from ch_07 import prim

from math import sqrt
from collections import defaultdict
```

```
def euc(a, b):
    return sqrt((a[0]-b[0])**2 + (a[1]-b[1])**2)

def euc_graph(pts):
    G = defaultdict(dict)
    for i, p in enumerate(pts):
        for j, q in enumerate(pts):
            if i == j: continue
            G[i][j] = euc(p,q)
    return G

def test_mtsp():
    """
    >>> G = euc_graph([
    ... (1,4), (1,2), (0,1), (3,4), (4,3), (3,2), (5,2), (2,0)
    ... ])
    >>> names = "abcdefgh"
    >>> [names[i] for i in mtsp(G, 0)] # Example from Cormen:
    ['a', 'b', 'c', 'h', 'd', 'e', 'f', 'g']
    """

from collections import defaultdict

def mtsp(G, r):                                 # 2-approx for metric TSP
    T, C = defaultdict(list), []                # Tree and cycle
    for c, p in prim(G, r).items():             # Build a traversable MSP
        T[p].append(c)                          # Child is parent's neighbor
    def walk(r):                                # Recursive DFS
        C.append(r)                             # Preorder node collection
        for v in T[r]: walk(v)                  # Visit subtrees recursively
    walk(r)                                     # Traverse from the root
    return C                                    # At least half-optimal cycle

from ch_08 import brutish_knapsack, rec_knapsack, knapsack

def test_knapsack():
    """
    >>> funcs = [brutish_knapsack, rec_knapsack, knapsack, bb_knapsack]
    >>> cases = [
    ...    #[[2, 4, 3, 6, 5], [2, 4, 3, 6, 6], 12, -1],
    ...    [[2, 3, 4, 5], [3, 4, 5, 6], 5, 7],
    ...    [[5, 1], [10, 75], 3, 75]
    ... ]
    >>> from random import *
    >>> for i in range(20):
    ...     n = randrange(10)
    ...     w = [randrange(1,100) for i in range(n)]
    ...     v = [randrange(1,100) for i in range(n)]
    ...     W = randrange(sum(w)+1)
    ...     cases.append([w, v, W, -1])
    >>> for w, v, W, e in cases:
    ...     sols = set(f(w, v, W) for f in funcs)
    ...     assert len(sols) == 1, (w, v, W, e, sols)
    ...     if e >= 0: assert sols.pop() == e
    ...
    >>>
    """

# Modified to run with 2.x (for the unit tests -- the 3.x version has also
# been tested).

#from __future__ import division
from heapq import heappush, heappop
from itertools import count

def bb_knapsack(w, v, c):
    sol = [0]                                   # Solution so far
    n = len(w)                                  # Item count

    idxs = list(range(n))
    idxs.sort(key=lambda i: v[i]/w[i],          # Sort by descending unit cost
              reverse=True)
```

```
    def bound(sw, sv, m):                        # Greedy knapsack bound
        if m == n: return sv                     # No more items?
        objs = ((v[i], w[i]) for i in idxs[m:]) # Descending unit cost order
        for av, aw in objs:                      # Added value and weight
            if sw + aw > c: break                # Still room?
            sw += aw                             # Add wt to sum of wts
            sv += av                             # Add val to sum of vals
        return sv + (av/aw)*(c-sw)               # Add fraction of last item

    def node(sw, sv, m):                         # A node (generates children)
        #nonlocal sol                            # "Global" inside bb_knapsack
        if sw > c: return                        # Weight sum too large? Done
        sol[0] = max(sol[0], sv)                 # Otherwise: Update solution
        if m == n: return                        # No more objects? Return
        i = idxs[m]                              # Get the right index
        ch = [(sw, sv), (sw+w[i], sv+v[i])]      # Children: without/with m
        for sw, sv in ch:                        # Try both possibilities
            b = bound(sw, sv, m+1)               # Bound for m+1 items
            if b > sol[0]:                       # Is the branch promising?
                yield b, node(sw, sv, m+1)       # Yield child w/bound

    num = count()                                # Helps avoid heap collisions
    Q = [(0, next(num), node(0, 0, 0))]          # Start with just the root
    while Q:                                     # Any nodes left?
        _, _, r = heappop(Q)                     # Get one
        for b, u in r:                           # Expand it...
            heappush(Q, (b, next(num), u))       # ... and push the children

    return sol[0]                                # Return the solution
```