```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-

class IndirectAddressedDictonary(object):
    """
    This class wraps a common python dict object and
    addresses it indirectly by doing an additional
    dereferencing step based on a provided function.
    """
    def __init__(self, func, init=None, default=None):
        self.index_of = func
        self.values = init or {}
        self.default = default

    def __getitem__(self, key):
        key = self.index_of(key)
        return self.values.get(key, self.default) if key != None else self.default

    def __setitem__(self, key, value):
        key = self.index_of(key)
        if key == None: return
        elif value != self.default: self.values[key] = value
        elif key in self.values: del self.values[key]

    def __delitem__(self, key):
        key = self.index_of(key)
        if key != None and key in self.values: del self.values[key]

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

    def __iter__(self):
        return self.values.iteritems()


class RowDict(IndirectAddressedDictonary):
    def __init__(self, indices):
        index_of = lambda i: indices[i]
        super(RowDict, self).__init__(index_of, default=0)


class EdgeDict(IndirectAddressedDictonary):
    def __init__(self, indices):
        def mkkey(i,j):
            if i == j: return None
            elif i > j: return mkkey(j,i)
            else: return indices[i], indices[j]
        index_of = lambda key: mkkey(*key)
        super(EdgeDict, self).__init__(index_of, default=0)


class BasicMatrix(object):
    """
    This is an implementation on an nxn+2 matrix as proposed
    on the exercise sheet. To be performant we do not use
    a 2d array but dictonaries with tupel keys to implement
    weight matrix. This matrix is addionally indexed indirectly
    to allow a more performant delete operation.
    """

    def __init__(self, graph, names=None):
        self.indices = range(len(graph))
        self.edges = EdgeDict(self.indices)
        self.node_sets = RowDict(self.indices)
        self.node_weights = RowDict(self.indices)
        for i in range(len(graph)):
            row_sum = 0
            for j in range(i+1, len(graph)):
                self.edges[i,j] = graph[i][j]
                row_sum += graph[i][j]
            self.node_weights[i] = row_sum
            self.node_sets[i] = [names[i]] if names else [i]
```

```
    def __getitem__(self, key):
        return self.edges[key]

    def __setitem__(self, key, value):
        i, j = sorted(key)
        self.node_weights[i] -= self.edges[i,j]
        self.edges[i,j] = value
        self.node_weights[i] += self.edges[i,j]

    def __delitem__(self, key):
        i, j = sorted(key)
        self.node_weights[i] -= self.edges[i,j]
        del self.edges[i,j]

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


class DebugableMatrix(BasicMatrix):
    def debug(self, msg=None, rjust=3):
        print msg or "Debug Matrix:"
        for i in range(len(self)):
            row = [str(self.indices[i]).rjust(rjust) + ": "]
            row += [str(self[i,j]).rjust(rjust) for j in range(len(self))]
            row.append( "|".rjust(rjust) )
            row.append( str(self.node_weights[i]).rjust(rjust) )
            row.append( "  " + str(self.node_sets[i]) )
            print ' '.join(row)
```

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-

from copy import deepcopy
from random import random
from matrix import DebugableMatrix

def randint(start, stop):
    """
    Generates a uniformliy distributed random
    number in the interval [start, stop).
    """
    # generates a uniformly distributed random float in [0.0,1.0)
    rand = random()
    # lift this to [start,stop)
    rand = start + (stop-start) * rand
    return int(rand)


class ContractableMatrix(DebugableMatrix):
    def contract_nodes(self, x, y):
        """
        This method contracts the nodes x and y.
        This alters also the dimension of the matrix.
        """
        assert x != y
        if x > y: return self.contract(y, x)

        # fuse all edges of y with the edges of x
        # then delete the edges of y
        for k in range(len(self)):
            self[x,k] += self[y,k]
            del self[y,k]
        # update the row sum and the node sets (keep track of changes)
        self.node_sets[x] += self.node_sets[y]
        del self.node_sets[y]
        del self.node_weights[y]
        del self.indices[y] # linear in len(self.indices)

    def contract(self, k=None):
        k = k or len(self)-2
        assert k <= len(self)-2
        while k > 0:
            x, y = self.select_edge()
            self.contract_nodes(x, y)
            k -= 1

    def select_edge(self):
        """
        Subrutine of contract.

        Pick an edge of the simple weighted graph
        representat by this objectsuch that the
        liklihood of picking an edge is proportional
        to its weight. Formally:

            Probability( (i,j) is picked )
            = weight(i,j) / sum_of_all_weights

        Returns the result as tupel (row, col).

        Note that no edge is picked from a row
        where the node_weight(i) = 0.
        """
        total_weight = reduce(lambda x,y: x+y, self.node_weights.values.values())
        rand = randint(0, total_weight)

        # backtrace row of rand
        row = 0
        while rand >= self.node_weights[row]: # no index-out-of-bounds: total_weight never hit
            rand -= self.node_weights[row] # never negative due to while condition
            row += 1
```

```
        # backtrace col of rand
        col = row + 1 # first edge entry in row
        while rand >= self[row,col]:
            rand -= self[row,col]
            col += 1

        return row, col
```

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-

from math import sqrt, ceil
from copy import deepcopy
from contract import ContractableMatrix


class Matrix(ContractableMatrix):
    def copy(self):
        graph = [[self[i,j] for j in range(len(self))] for i in range(len(self))]
        return Matrix(graph)


def fastcut(M):
    """
    Fast-Cut implementation based on contract,
    as presented in the lecture
    """
    if len(M) <= 3:
        # enumerate all solutions for size <= 3
        return min_cut(M)
    else:
        p = int(ceil(len(M)/sqrt(2)))
        k = len(M)-p

        # create two independent contractions
        M1, M2 = M.copy(), M
        M1.contract(k)
        M2.contract(k)

        # return smaller cut
        return min(fastcut(M1), fastcut(M2))


def min_cut(M):
    assert len(M) >= 2 and len(M) <= 3
    if len(M) == 2:
        return M[0][1], M.node_sets[0], M.node_sets[1]
    else: # len(M) == 3
        return min([
            (M[0,1]+M[0,2], M.node_sets[0], M.node_sets[1]+M.node_sets[2]),
            (M[0,1]+M[1,2], M.node_sets[1], M.node_sets[0]+M.node_sets[2]),
            (M[0,2]+M[1,2], M.node_sets[2], M.node_sets[0]+M.node_sets[1]),
        ])
```