Ex 6 & 7:

We first define a datastructure that encapsulates nasty details of array accesses:

```
# This class represents a matrix like in ex 6:
#
# 0    w_12    ...   w_1n  | W_1 | v_1
# 0     0      ...   w_2n  | W_2 | v_2
#              ...         | ... | ...
# 0            ...    0    | W_n | v_n
#
# It is stored in an array. Hereby w_ij and W_i are integers,
# v_i are arrays themselves.
#
# Every read and write access is done in constant time.
class DatMatrix

      attr_reader :n
      attr_accessor :m, :array

      # We store the following
      #   - the one integer 'n'
      #   - the next integer 'm', intended to count the remaining thingsies
      #   - the array 'array' of size O(n²)
      # Hence we need O(n²)
      def initialize n
            size_of_matrix = (n*(n-1) / 2)
            size_of_vectors = 2 * n
            @n = n
            @m = n
            @array = Array.new size_of_matrix + size_of_vectors, 0
            # initialize names
            (1..n).each do |i|
                  @array[v_position(i)] = [i]
            end
      end

      # Produces a copy of the DatMatrix object called
      def copy
            o = DatMatrix.new @n
            o.array = @array.dup
            o
      end

      # gets the weight of the coordinate i, j
      def get_w i, j
            return 0 if i >= j
            @array[w_position(i,j)]
      end

      # sets the weight of entry i, j
      def set_w i, j, w
            return if i >= j
            pos = w_position(i,j)
            before = @array[w_position(i,j)]
            @array[w_position(i,j)] = w
            @array[W_position(i)] += w - before
      end

      # gets the sum in line i
```

```
def get_W i
      @array[W_position(i)]
end

# gets the nodes of index i
def get_v i
      @array[v_position(i)]
end

def join_v i, j
      vi = v_position i
      vj = v_position j
      a = @array[vi]
      b = @array[vj]
      @array[vi] = a + b
      @array[vj] = []
end

# Output to console (max 3 char per entry)
def print
      # lines
      (1..@n).each do |i|
            line = ""
            # rows of w
            (1..@n).each do |j|
                  line << printable(get_w(i,j))
                  line << ' '
            end
            line << '| ' + printable(get_W(i))
            line << '| ' + get_v(i).to_s

            puts line
      end
end

private

# Helper that calculates for coordinates i and j the array index.
def w_position i, j
      (((j - 2)*(j - 1)) / 2) + (i - 1)
end

# Helper that calculates for i the array index of W_i
def W_position i
      ((@n * (@n-1)) / 2) + (i - 1)
end

# Helper that calculates for i the array index of v_i
def v_position i
      ((@n * (@n-1)) / 2) + @n + (i - 1)
end

# Returns a string starting with n and filled up with spaces
# such that it is 3 chars long
def printable n
      if n < 10
            return "#{n}  "
      elsif n < 100
            return "#{n} "
      elsif n < 1000
            return "#{n}"
      end
```

```
      end
end
```

We then define everything needed to contract edges.

```

# Computes contract for input DatMatrix W
def contract w
      # 1) If graph has only two vertices,  then stop
    #    output y and cap(y)
      if w.m == 2
            y = w.get_v 1
            cap = w.get_W 1
            return [y, cap]
      end

      # 2) At random contract an edge where the likelihood of
    #    taking an edge is proportional to the edge weight.
    edge = select_edge w
    contract_edge w, edge

    # 3) Recursively apply CONTRACT to the resulting graph
      w.m -= 1
    contract w
end

# As above but without recursion
def k_contract w, k

      k.times do |i|
            if w.m == 2
                  y = w.get_v 1
                  cap = w.get_W 1
                  return [y, cap]
            end

            edge = select_edge w
            contract_edge w, edge

            w.m -= 1
      end
end



# Returns an array [row, col] with indeces.
# Note: row < col holds!
def select_edge w
      # Sum up the partial weights ~ O(n)
      total_w = 0
      (1..w.n).each do |w_i|
            total_w += w.get_W(w_i)
      end

      # Draw random number between 0 and total_w ~ O(1)
      rnd = Random.rand * total_w

      # search the column where this weight is located ~ O(n)
```

```
      sum_up = w.get_W(1)
      row_index = 1
      while sum_up < rnd
            row_index += 1
            sum_up += w.get_W(row_index)
      end

      # now we have the correct row
      col_index = w.n
      while sum_up > rnd
            sum_up -= w.get_w(row_index, col_index)
            col_index -= 1
      end

      # compensate for -1 too much
      [row_index, col_index + 1]
end

def contract_edge w, edge
      # 1) combine the lines of the selected nodes
      (1..w.n).each do |j|
            a = w.get_w(edge[0], j)
            b = w.get_w(edge[1], j)
            w.set_w(edge[0], j, a + b)
            w.set_w(edge[1], j, b - b)
      end
      w.set_w(edge[0], edge[1], 0)

      # 2) combine edges that point to both nodes of the edge
      #    or just reroute them
      (1..w.n).each do |i|
            a = w.get_w(i, edge[0])
            b = w.get_w(i, edge[1])
            w.set_w(i, edge[0], a + b)
            w.set_w(i, edge[1], b - b)
      end

      # 3) write down the names
      w.join_v(edge[0], edge[1])
end
```

Finally we can implement fastcut with the lines coded before:

```
def n_fastcut w
      smallest = -1
      smallest_w = nil

      no = Math.log2(w.n)**2
      no.ceil.times do |n|
            cur_w = w.copy
            cur = fastcut cur_w
            if cur < smallest || smallest == -1
                  smallest_w = cur_w
                  smallest = cur
            end
      end
      [smallest, smallest_w]
end

def fastcut w
```

```
    if w.m <= 3
        return enum_edge w
    end

    w1 = w
    w2 = w.copy

    p = (w.m / Math.sqrt(2)).ceil

    k_contract w1, p
    k_contract w2, p

    r1 = fastcut w1
    r2 = fastcut w2

    return r1 < r2 ? r1 : r2
end


def enum_edge w
    # we call the remaining nodes a, b and c
    # by construction we know, that one of them has index 1
    a = 1
    b = 0
    c = 0

    (1..w.n).each do |j|
        if b == 0 and w.get_w(1,j) > 0
            b = j
            next
        elsif c == 0 and w.get_w(1,j) > 0
            c = j
            break
        end
    end

    # if we have no c yet, the graph might look like
    #       b
    #     /   \
    #   a       c
    # and we have to look at node b
    (1..w.n).each do |j|
        if w.get_w(b,j) > 0
            c = j
            break
        end
    end

    # if still no c is found, the graph looks like
    #   a - b
    # and therefore there's just one split
    if c == 0
        return w.get_w(a,b)
    end

    ab = w.get_w(a,b)
    bc = w.get_w(b,c)
    ac = w.get_w(a,c)

    puts "----"
    puts "Intermediate we have a: #{a}, b: #{b}, c: #{c}"
```

```
      # shall we cut of a ?
      if ab + ac <= ab + bc and ab + ac <= ac + bc
            puts "cut of a"
            return ab + ac
      # shall we cut of b ?
      elsif ab + bc <= ab + ac and ab + bc <= ac + bc
            puts "cut of b"
            return ab + bc
      # shall we cut of c ?
      elsif ac + bc <= ab + bc and ac + bc <= ab + ac
            puts "cut of c"
            return ac + bc
      end

end
```

In order to execute this, we have to load the files from above and define a DatMatrix object. The results can be extracted from that object, see this example script:

```
load 'datMatrix.rb'
load 'contract.rb'
load 'fastcut.rb'

m = DatMatrix.new 5

# Setup the following matrix:
#   0 5 0 3 0
#   0 0 3 0 1
#   0 0 0 0 2
#   0 0 0 0 0
#   0 0 0 0 0
m.set_w 1, 2, 5
m.set_w 1, 4, 3
m.set_w 2, 3, 3
m.set_w 2, 5, 1
m.set_w 3, 5, 2

puts "============="
puts "Before:"
m.print


r = n_fastcut m

puts "============="
puts "After:"
puts " capacity: #{r[0]}"
puts " v_0: #{r[1].get_v 1}"
puts " v_1 is the rest :)"
```