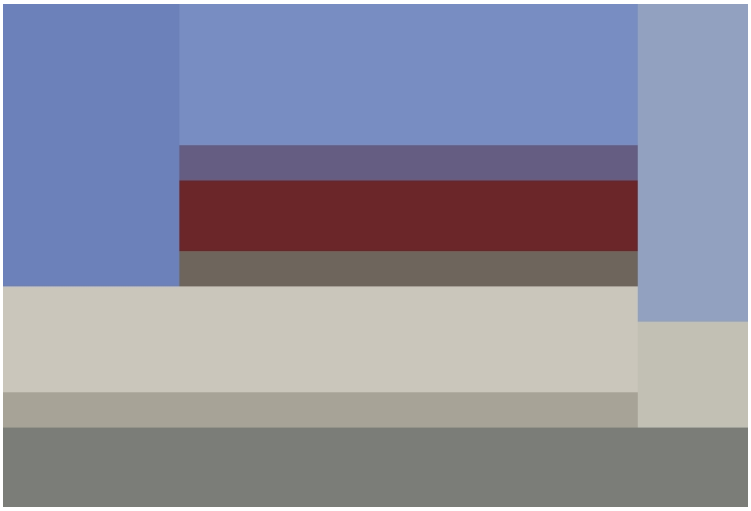
```
# compress.py
# Ronald L. Rivest
# October 9, 2007

# Routines to compress a photo using dynamic programming
# Inputs:
#    image (read from file)
#    k     (positive integer)
#    t     (positive integer)
# Output:
#    image (written to file)
# Procedure:
#    Consider image as array of kxk pixel "blocks"
#    Find "best" representation of image as a set of t
#    non-overlapping constant-rgb rectangles, where each rectangle is
#    a union of blocks, and solution has "hierarchical"
#    property: it can be derived by making top-level
#    cut all the way across image, and doing same recursively
#    on two parts.  Best solution is one that minimizes
#    sum of squares between actual pixel values and ones
#    derived from solution.

import Image                    # Python Imaging Library

def get_image(filename):
    """
    Read image from file and return pixel array of RGB values.
    Upper left is pixels[0][0].
    In general, pixel[row][col] is (r,g,b) tuple at given position.
    """
    im = Image.open(filename)
    w,h = im.size
    data = im.getdata()
    pixels = [ [(0,0,0)]*w for i in range(h) ]
    for i in range(h):
        for j in range(w):
            pixels[i][j] = data[i*w+j]
    return pixels

def save_image(pixels, filename):
    """
    Save image (given as pixel array) as a file with given filename.
    """
    w,h = image_size = (len(pixels[0]),len(pixels))
    im = Image.new('RGB',image_size)
    data = [(0,0,0)]*(w*h)
    for i in range(h):
        for j in range(w):
            data[i*w+j] = pixels[i][j]
    im.putdata(data)
    im.save(filename)

def one_part_subproblem(ak,bk,ck,dk):
    """
    Find value of one-part solution to subproblem (ak,bk,ck,dk);
    save this value in subproblem_value, and save solution itself
    (i.e. rgb values best for this rectangle) in subproblem_soln.
```

```
    (Value is sum of squares of differences between actual pixel
    values and average pixel values for that rectangle.)
    """
    stats = [0,0,0,0,0,0,0]
    for ik in range(ak,bk):
        for jk in range(ck,dk):
            for i in range(len(stats)):
                stats[i] += block_stats[(ik,jk)][i]
    n,rss,rs,gss,gs,bss,bs = stats
    v = rss - (rs**2)//n + gss - (gs**2)//n + bss - (bs**2)//n
    subproblem_value[(ak,bk,ck,dk,1)] = v
    subproblem_soln[(ak,bk,ck,dk,1)] = [(ak,bk,ck,dk,rs/n,gs/n,bs/n)]

def multipart_subproblem(ak,bk,ck,dk,tt):
    """
    Solve subproblem with blocks (ik,jk)
    where:
        0 <= ak <= ik < bk <= hk
        0 <= ck <= jk < dk <= wk
    by dividing it into exactly tt parts
    Save solution value in subproblem_value, and save in
    subproblem_soln either:
        (ak,bk,ck,dk,r,g,b)            [for t == 1]
        list of two subproblems to use [for t>1]
    """
    global block_stats,subproblem_value, subproblem_soln
    if (bk-ak)*(dk-ck)>tt:
        subproblem_value[(ak,bk,ck,dk,tt)] = 1e100
    best_v = 1e100
    best_soln = []
    # horizontal cuts:
    for ek in range(ak+1,bk):
        for t1 in range(1,tt):
            v1 = subproblem_value[(ak,ek,ck,dk,t1)]
            v2 = subproblem_value[(ek,bk,ck,dk,tt-t1)]
            if v1+v2<best_v:
                best_v = v1+v2
                best_soln = [(ak,ek,ck,dk,t1),(ek,bk,ck,dk,tt-t1)]
    # vertical cuts
    for fk in range(ck+1,dk):
        for t1 in range(1,tt-1):
            v1 = subproblem_value[(ak,bk,ck,fk,t1)]
            v2 = subproblem_value[(ak,bk,fk,dk,tt-t1)]
            if v1+v2<best_v:
                best_v = v1+v2
                best_soln = [(ak,bk,ck,fk,t1),(ak,bk,fk,dk,tt-t1)]
    subproblem_value[(ak,bk,ck,dk,tt)] = best_v
    subproblem_soln[(ak,bk,ck,dk,tt)] = best_soln

def generate_and_solve_subproblems(pixels,k,t):
    """
    Consider given h x w image of pixels
    as divided into k x k blocks.
    Number of k x k blocks is:
        hk = (h-1)/k (height) by
        wk = (w-1)/k (width)
    Consider each (ak,bk,ck,dk) subimage
```

```
    consisting of blocks (ik,jk) where
        0 <= ak <= ik < bk <= hk
        0 <= ck <= jk < dk <= wk
    ("k" suffix means we are talking about blocks, not pixels.)
    Problems are considered in order of increasing size;
    so that when a problem is done, all of its smaller
    component problems have already been solved.
    """
    w,h = (len(pixels[0]),len(pixels))
    hk = (h-1)//k + 1
    wk = (w-1)//k + 1
    for uk in range(1,hk+1):
        for ak in range(0,hk-uk+1):
            bk = ak + uk
            for vk in range(1,wk+1):
                for ck in range(0,wk-vk+1):
                    dk = ck + vk
                    one_part_subproblem(ak,bk,ck,dk)
                    for tt in range(2,t+1):
                        multipart_subproblem(ak,bk,ck,dk,tt)

def compute_block_stats(pixels,k):
    """
    Compute statistics for each kxk block in picture.
    Stats are a 7-tuple:
        n, rss, rs, gss,gs,bss,bs
    where
        n = number of pixels in block
            (note blocks on right and bottom may be partial)
        rss = red sum of squares of pixel values
        rs = red sum of pixel values
        gss = green sum of squares of pixel values
        gs = green sum of pixel values
        bss = blue sum of squares of pixel values
        bs = blue sum of pixel values
    Store stats in block_stats.
    """
    global block_stats
    w,h = (len(pixels[0]),len(pixels))
    hk = (h-1)//k + 1
    wk = (w-1)//k + 1
    block_stats = {}
    for i in range(h):
        ik = i//k
        for j in range(w):
            jk = j//k
            if not block_stats.has_key((ik,jk)):
                block_stats[(ik,jk)] = (0,0,0,0,0,0,0)
            r,g,b = pixels[i][j]
            n,rss,rs,gss,gs,bss,bs = block_stats[(ik,jk)]
            n += 1
            rss += r*r
            rs += r
            gss += g*g
            gs += g
            bss += b*b
            bs += b
```

```
            block_stats[(ik,jk)] = (n,rss,rs,gss,gs,bss,bs)

def generate_solution_parts(subproblem):
    """
    Return list that contains all one-part
    solution parts for this subproblem, by
    recursively unwinding solutions with more parts.
    """
    global subproblem_soln
    ak,bk,ck,dk,t = subproblem
    if t==1:
        return list(subproblem_soln[subproblem])
    else:
        L = []
        for soln_part in subproblem_soln[subproblem]:
            L.extend(generate_solution_parts(soln_part))
    return L

def main():
    """
    Demonstrate dynamic programming image compression.
    """
    global block_stats, subproblem_value, subproblem_soln
    filename_list = ["eye",
                     "kilimanjaro",
                     "pumpkin",
                     "firetruck"]
    for f in filename_list:
        input_filename = f+".jpg"
        pixels = get_image(input_filename)
        w,h = (len(pixels[0]),len(pixels))
        # try different k x k block sizes
        for k in [256,128,64,32,16,8]:
            if w//k > 50: continue
            hk = (h-1)//k + 1
            wk = (w-1)//k + 1
            # print "hk = ",hk,"wk = ",wk
            compute_block_stats(pixels,k)
            # try different number t of parts wanted
            for t in [10,20,30,40,50,60,70,80,90,100]:
                if t>(hk*wk):
                    break
                subproblem_value = {}
                subproblem_soln = {}
                generate_and_solve_subproblems(pixels,k,t)
                pixels_out = [[0]*w for i in range(h)]
                for soln in generate_solution_parts((0,hk,0,wk,t)):
                    ak,bk,ck,dk,r,g,b = soln
                    for i in range(ak*k,min(bk*k,h)):
                        for j in range(ck*k,min(dk*k,w)):
                            pixels_out[i][j] = (r,g,b)
                output_filename = "%s_%03d_%03d.jpg"%(f,k,t)
                save_image(pixels_out,output_filename)
                print output_filename
```

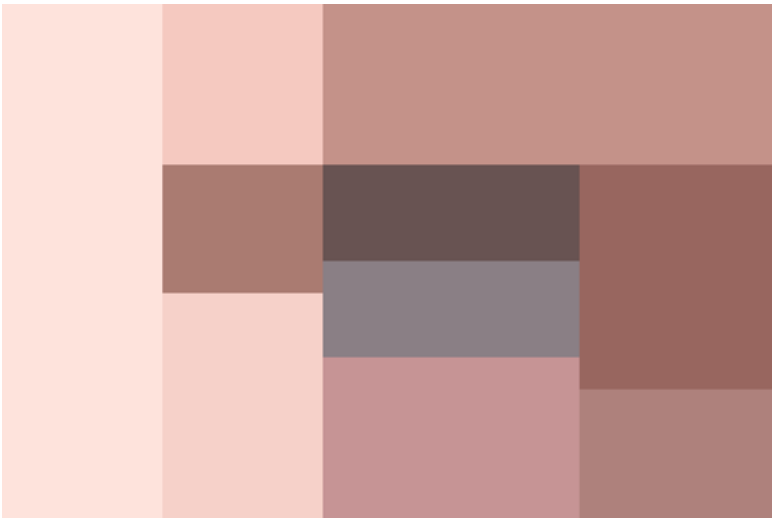
Original



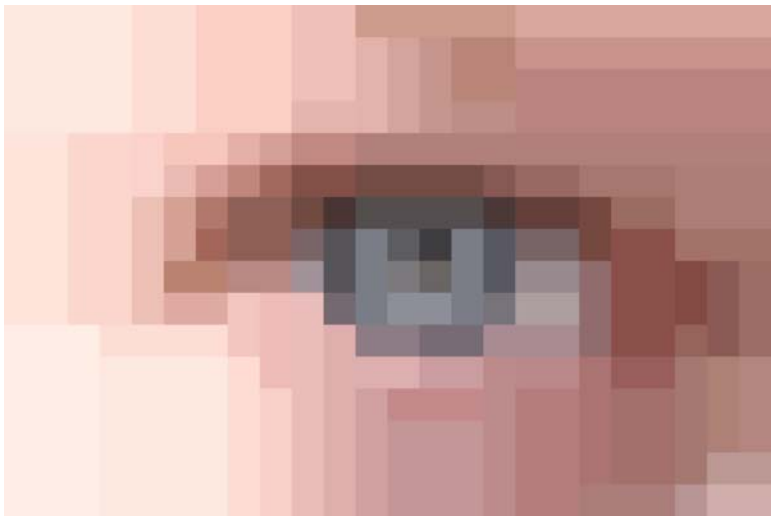
10 rectangles

100 rectangles

Original



10 rectangles



100 rectangles