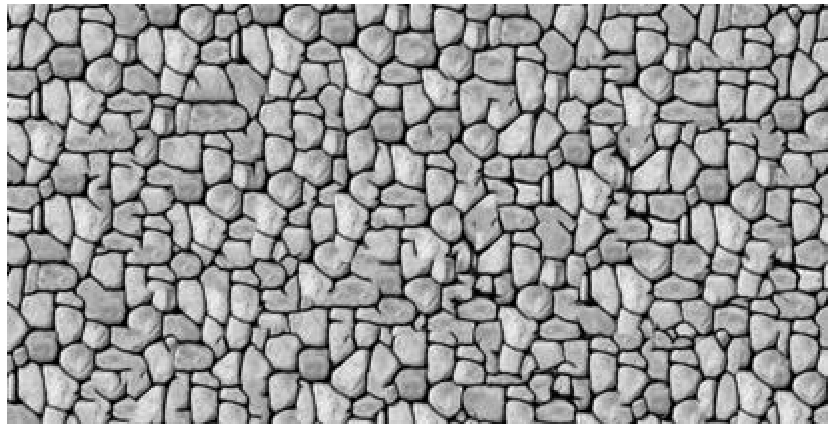


Texture Quilting

In this assignment, you will develop code to stitch together image patches sampled from an input texture in order to synthesize new texture images. You can download the test image used to generate the example above from assignment folder Canvas.

You should start by reading through the whole assignment, looking at the provided code in detail to make sure you understand what it does. The main fucntion ***quilt_demo*** appears at the end. You will need to write several subroutines in order for it to function properly.

Name: Daniel Binoy

SID: 22622651

---

1. Shortest Path [25 pts]

Write a function ***shortest_path*** that takes an 2D array of ***costs***, of shape HxW, as input and finds the *shortest vertical path* from top to bottom through the array. A vertical path is specified by a single horizontal location for each row of the H rows. Locations in successive rows should not differ by more than 1 so that at each step the path either goes straight or moves at most one pixel to the left or right. The cost is the sum of the costs of each entry the path traverses. Your function should return an length H vector that contains the index of the path location (values in the range 0..W-1) for each of the H rows.

You should solve the problem by implementing the dynamic programming algorithm described in class. You will have a for-loop over the rows of the "cost-to-go" array (M in the slides), computing the cost of the shortest path up to that row using the recursive formula that depends on the costs-to-go for the previous row. Once you have get to the last row, you can then find the smallest total cost. To find the path which actually has this smallest cost, you will need to do backtracking. The easiest way to do this is to also store the index of whichever minimum was selected at each location. These indices will also be an HxW array. You can then backtrack through these indices, reading out the path.

Finally, you should create at least three test cases by hand where you know the shortest path and see that the code gives the correct answer.

In your implementation you will need to have a *for-loop* over the rows of the cost matrix since the computation has to be carried out in a squential order. However, the computation for each row can be done in a vectorized manner without an explicit loop (e.g., my implementation used the **numpy** operations **concatenate,stack,min,argmin**). If you get stuck I recommend first implementing a version with nested loops to make sure you get the algorithm correct and then go back and see how to "vectorize" it.

In [3]:
```
#modules used in your code
import numpy as np
import matplotlib.pyplot as plt
import sys
```

In [4]:
```
def shortest_path(costs):
    """
    This function takes an array of costs and finds a shortest path from the
    top to the bottom of the array which minimizes the total costs along the
    path. The path should not shift more than 1 location left or right between
    successive rows.

    In other words, the returned path is one that minimizes the total cost:

        total_cost = costs[0,path[0]] + costs[1,path[1]] + costs[2,path[2]] + ...

    subject to the constraint that:

        abs(path[i]-path[i+1])<=1

    Parameters
    ----------
    costs : 2D float array of shape HxW
         An array of cost values with W>=3

    Returns
    -------
    path : 1D array of length H
        indices of a vertical path.  path[i] contains the column index of
        the path for each row i.
    """
    nrows = costs.shape[0]
    ncols = costs.shape[1]
    path = []
    newcosts = np.zeros(nrows*ncols).reshape(nrows, ncols)

    #to get to anywhere in top row, just same value
    newcosts[0] = costs[0]

    #start at second row
    for row in range(1, nrows):
        newcosts[row] = costs[row]+np.array([
            newcosts[row-1],
            np.append(newcosts[row-1][1:], [sys.maxsize]),
            np.append([sys.maxsize],newcosts[row-1][:-1])]).min(axis=0)

    #start at lowest  col in bottom row
```

```
    row = nrows-1
    col = np.argmin(newcosts[row])
    while row>=0:
        path.append(col) #add to path
        row-=1
        left = col-1 if col>0 else col
        center = col
        right = col+1 if col<ncols-1 else col

        if newcosts[row][left]<newcosts[row][center]:
            if newcosts[row][left]<newcosts[row][right]:
                col = left
            else:
                col = right
        else:
            if newcosts[row][right]<newcosts[row][center]:
                col = right
            else:
                col = center


    # to keep the implementation simple, we will refuse to handle
    # the boundary case where the cost array is very narrow.
    assert(ncols>=3)



    return list(reversed(path))
```

In [5]:
```
#
# Your test code goes here.  Come up with at least 3 test cases by manually
# constructing a cost matrix where you know what the shortest path should be.
#

# One of your tests should be a case where a simple greedy forward search
# (i.e., finding the min in the first row and then repeatedly choosing the
# min of the 3 neighbors below it) fails but your dynamic program finds the
# global optimum.
#
def greedy(costs):
    nrows = costs.shape[0]
    ncols = costs.shape[1]
    path = []
    row = 0
    col = np.argmin(costs[row])
    while row<nrows-1:
        path.append(col)
        row+=1
        left = col-1 if col>0 else col
        center = col
        right = col+1 if col<ncols-1 else col

        if costs[row][left]<costs[row][center]:
            if costs[row][left]<costs[row][right]:
                col = left
            else:
                col = right
        else:
            if costs[row][right]<costs[row][center]:
                col = right
            else:
                col = center
    path.append(col)
    return path



costs1 = np.array([[1, 0, 8, 1],
                   [0, 3, 2, 1],
                   [1, 1, 4, 8],
                   [1, 1, 3, 1]])
```

```
path1 = shortest_path(costs1)
path1greedy = greedy(costs1)
print("***************")
print("shortest: ", path1)
print("greedy  : ", path1greedy)

costs2 = np.array([[5, 9, 9, 4],
                   [0, 9, 9, 5],
                   [9, 0, 9, 5],
                   [0, 9, 9, 5]])
path2 = shortest_path(costs2)
path2greedy = greedy(costs2)
print("***************")
print("shortest: ", path2)
print("greedy  : ", path2greedy)


costs3 = np.array([[0, 9, 9, 4],
                   [0, 0, 9, 0],
                   [9, 9, 9, 0],
                   [9, 9, 9, 0]])
path3 = shortest_path(costs3)
path3greedy = greedy(costs3)
print("***************")
print("shortest: ", path3)
print("greedy  : ", path3greedy)

costs4 = np.array([[3, 9, 9, 0],
                   [9, 0, 9, 9],
                   [9, 9, 0, 9],
                   [9, 9, 9, 0]])
path4 = shortest_path(costs4)
path4greedy = greedy(costs4)
print("***************")
print("shortest: ", path4)
print("greedy  : ", path4greedy)
```

```
****************
shortest:  [1, 0, 0, 0]
greedy  :  [1, 0, 0, 0]
****************
shortest:  [0, 0, 1, 0]
```

```
greedy  :  [3, 3, 3, 3]
***************
shortest:  [3, 3, 3, 3]
greedy  :  [0, 0, 0, 0]
***************
shortest:  [0, 1, 2, 3]
greedy  :  [3, 3, 2, 3]
```

2. Image Stitching: [25 pts]

Write a function ***stitch*** that takes two gray-scale images, ***left_image*** and ***right_image*** and a specified ***overlap*** and returns a new output image by stitching them together along a vertical seam where the two images have very similar brightness values. If the input images are of widths ***w1*** and ***w2*** then your stitched result image returned by the function should be of width ***w1+w2-overlap*** and have the same height as the two input images.

You will want to first extract the overlapping strips from the two input images and then compute a cost array given by the absolute value of their difference. You can then use your ***shortest_path*** function to find the seam along which to stitch the images where they differ the least in brightness. Finally you need to generate the output image by using pixels from the left image on the left side of the seam and from the right image on the right side of the seam. You may find it easiest to code this by first turning the path into an binary (alpha) mask for each image and then using the standard blending approach we used in the previous assignment.

In [ ]:

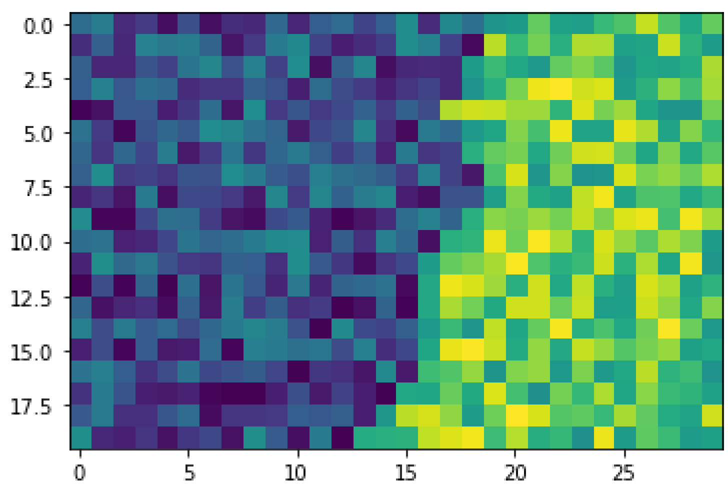
```

```

In [99]:
```
def stitch(left_image, right_image, overlap):
    """
    This function takes a pair of images with a specified overlap and stitches them
    togther by finding a minimal cost seam in the overlap region.

    Parameters
    ----------
    left_image : 2D float array of shape HxW1
        Left image to stitch

    right_image : 2D float array of shape HxW2
        Right image to stitch

    overlap : int
        Width of the overlap zone between left and right image

    Returns
    -------
    stitched : 2D float array of shape Hx(W1+W2-overlap)
        The resulting stitched image
    """

    # inputs should be the same height
    assert(left_image.shape[0]==right_image.shape[0])
    assert(overlap>=3)

    # your code here
    stitched = np.zeros((left_image.shape[0], left_image.shape[1]+right_image.shape[1]-overlap))
    left_strips = left_image[:,-overlap:]
    right_strips = right_image[:,:overlap]

    #double checking size of strips
    assert(left_strips.shape[1] == overlap)
    assert(left_strips.shape[0] == left_image.shape[0])
    assert(right_strips.shape[1] == overlap)
    assert(right_strips.shape[0] == right_image.shape[0])

    #calculate difference between overlapping strips
    difference = np.absolute(left_strips-right_strips)

    #least value path down difference, in absolute coordinates
    path = np.array(shortest_path(difference))+left_image.shape[1]-overlap
```

```
    for r in range(stitched.shape[0]):
        for c in range(stitched.shape[1]):
            stitched[r][c] = left_image[r][c] if c<path[r] else right_image[r][c-left_image.shape[1]+overlap]


    assert(stitched.shape[0]==left_image.shape[0])
    assert(stitched.shape[1]==(left_image.shape[1]+right_image.shape[1]-overlap))

    return stitched
```

In [109]:

```
# a simple test visualization of stitching two random
# tiles which have different overall brightness so we
# can easily see where the seam is

L = np.random.rand(20,20)+1
R = np.random.rand(20,20)+2

S = stitch(L,R,10)
plt.imshow(S)
plt.show()
```



3. Texture Quilting: [25 pts]

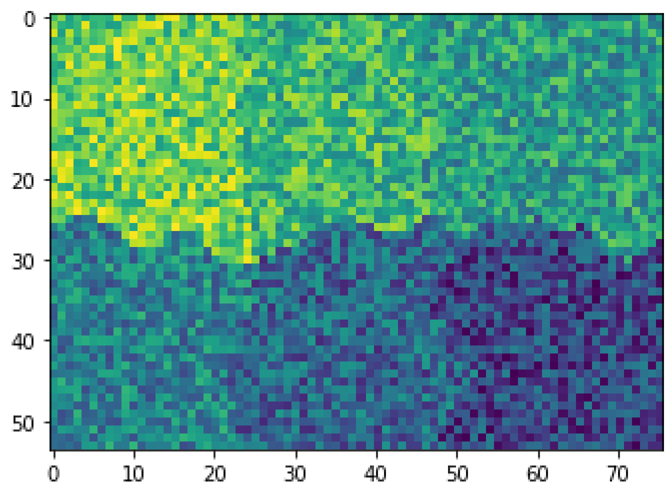
Write a function ***synth_quilt*** that takes as input an array indicating the set of texture tiles to use, an array containing the set of available texture tiles, the ***tilesize*** and ***overlap*** parameters and synthesizes the output texture by stitching together the tiles. ***synth_quilt*** should utilize your stitch function repeatedly. First, for each horizontal row of tiles, construct the stitched row by successively stitching the next tile in the row on to the right side of your row image. Once you have row images for all the rows, you can stitch them together to get the final image. Since your stitch function only works for vertical seams, you will want to transpose the rows, stitch them together, and then transpose the result. You may find it useful to look at the provided code below which simply puts down the tiles with the specified overlap but doesn't do stitching. Your quilting function will return a similar result but with much smoother transitions between the tiles.

In [68]:
```
def synth_quilt(tile_map,tiledb,tilesize,overlap):

    """
    This function takes as input an array indicating the set of texture tiles
    to use at each location, an array containing the database of available texture
    tiles, tilesize and overlap parameters, and synthesizes the output texture by
    stitching together the tiles


    Parameters
    ----------
    tile_map : 2D array of int
        Array storing the indices of which tiles to paste down at each output location

    tiledb : 2D array of int
        Collection of sample tiles to select from. The array is of size ntiles x npixels
        where each tile image is stored in vectorized form as a row of the array.

    tilesize : (int,int)
        Size of a tile in pixels

    overlap : int
        Amount of overlap between tiles

    Returns
    -------
    output : 2D float array
        The resulting synthesized texture of size
    """

    # determine output size based on overlap and tile size
    outh = (tilesize[0]-overlap)*tile_map.shape[0] + overlap
    outw = (tilesize[1]-overlap)*tile_map.shape[1] + overlap
    output = np.zeros((outh,outw))


    output = []
    for i in range(tile_map.shape[0]):
        row = np.reshape(tiledb[tile_map[i,0]], tilesize)
        for j in range(1, tile_map.shape[1]):
            tile = np.reshape(tiledb[tile_map[i, j]], tilesize)
            row = stitch(row, tile, overlap) #stitch row together
```

```
        if len(output) == 0:
            output = row.T #if first row, set equal
        else:
            output = stitch(output, row.T, overlap) #stitch columns together


    return output.T
```

In [47]:

```
tile_map = np.arange(6).reshape(2, 3)
tiledb = np.random.rand(6, 1024)



for i in range(6):
    tiledb[i]-=i/4

tilesize = (32, 32)
overlap = 10

S = synth_quilt(tile_map, tiledb, tilesize, overlap)
plt.imshow(S)
plt.show()
```



4. Texture Synthesis Demo [25pts]

The function provided below ***quilt_demo*** puts together the pieces. It takes a sample texture image and a specified output size and uses the functions you've implemented previously to synthesize a new texture sample.

You should write some additional code in the cells that follow to in order demonstrate the final result and experiment with the algorithm parameters in order to produce a compelling visual result and write explanations of what you discovered.

Test your code on the provided image *rock_wall.jpg*. There are three parameters of the algorithm. The *tilesize*, *overlap* and *K*. In the provided ***texture_demo*** code below, these have been set at some default values. Include in your demo below images of three example texture outputs when you: (1) increase the tile size, (2) decrease the overlap, and (3) increase the value for K. For each result explain how it differs from the default setting of the parameters and why.

Test your code on two other texture source images of your choice. You can use images from the web or take a picture of a texture yourself. You may need to resize or crop your input image to make sure that the ***tiledb*** is not overly large. You will also likely need to modify the ***tilesize*** and ***overlap*** parameters depending on your choice of texture. Once you have found good settings for these parameters, synthesize a nice output texture. Make sure you display both the image of the input sample and the output synthesis for your two other example textures in your submitted pdf.

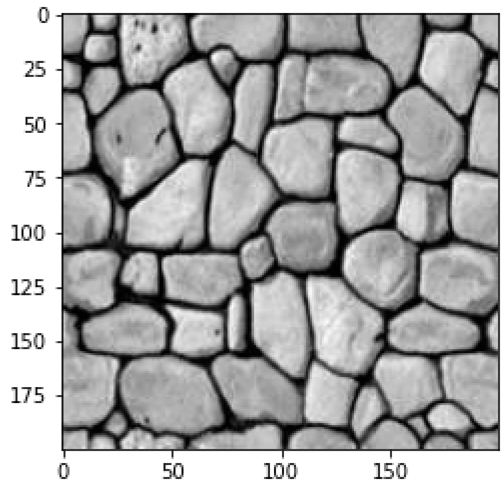
In [48]:
```
#skimage is only needed for sample tiles code provided below
#you should not use it elsewhere in your own code
import skimage as ski

def sample_tiles(image,tilesize,randomize=True):
    """
    This function generates a library of tiles of a specified size from a given source image

    Parameters
    ----------
    image : float array of shape HxW
        Input image

    tilesize : (int,int)
        Dimensions of the tiles in pixels


    Returns
    -------
    tiles : float array of shape  numtiles x numpixels
        The library of tiles stored as vectors where npixels is the
        product of the tile height and width
    """

    tiles = ski.util.view_as_windows(image,tilesize)
    ntiles = tiles.shape[0]*tiles.shape[1]
    npix = tiles.shape[2]*tiles.shape[3]
    assert(npix==tilesize[0]*tilesize[1])

    print("library has ntiles = ",ntiles,"each with npix = ",npix)

    tiles = tiles.reshape((ntiles,npix))

    # randomize tile order
    if randomize:
        tiles = tiles[np.random.permutation(ntiles),:]

    return tiles


def topkmatch(tilestrip,dbstrips,k):
    """
```

```
    This function finds the top k candidate matches in dbstrips that
    are most similar to the provided tile strip.

    Parameters
    ----------
    tilestrip : 1D float array of length npixels
        Grayscale values of the query strip

    dbstrips : 2D float array of size npixels x numtiles
        Array containing brightness values of numtiles strips in the database
        to match to the npixels brightness values in tilestrip

    k : int
        Number of top candidate matches to sample from

    Returns
    -------
    matches : list of ints of length k
        The indices of the k top matching tiles
    """
    assert(k>0)
    assert(dbstrips.shape[0]>k)
    error = (dbstrips-tilestrip)
    ssd = np.sum(error*error,axis=1)
    ind = np.argsort(ssd)
    matches = ind[0:k]
    return matches


def quilt_demo(sample_image, ntilesout=(10,20), tilesize=(30,30), overlap=5, k=5):
    """
    This function takes an image and quilting parameters and synthesizes a
    new texture image by stitching together sampled tiles from the source image.


    Parameters
    ----------
    sample_image : 2D float array
        Grayscale image containing sample texture

    ntilesout : list of int
        Dimensions of output in tiles,  e.g. (3,4)
```

```
    tilesize : int
        Size of the square tile in pixels

    overlap : int
        Amount of overlap between tiles

    k : int
        Number of top candidate matches to sample from

    Returns
    -------
    img : list of int of length K
        The resulting synthesized texture of size
    """

    # generate database of tiles from sample
    tiledb = sample_tiles(sample_image,tilesize)
    # number of tiles in the database
    nsampletiles = tiledb.shape[0]

    if (nsampletiles<k):
        print("Error: tile database is not big enough!")

    # generate indices of the different tile strips so we can easily
    # extract the left, right, top or bottom overlap strip from a tile
    i,j = np.mgrid[0:tilesize[0],0:tilesize[1]]
    top_ind = np.ravel_multi_index(np.where(i<overlap),tilesize)
    bottom_ind = np.ravel_multi_index(np.where(i>=tilesize[0]-overlap),tilesize)
    left_ind = np.ravel_multi_index(np.where(j<overlap),tilesize)
    right_ind = np.ravel_multi_index(np.where(j>=tilesize[1]-overlap),tilesize)

    # initialize an array to store which tile will be placed
    # in each location in the output image
    tile_map = np.zeros(ntilesout,'int')

    print('row:')
    for i in range(ntilesout[0]):
        print(i)
        for j in range(ntilesout[1]):

            if (i==0)&(j==0):                  # first row first tile
                matches = np.zeros(k) #range(nsampletiles)
```

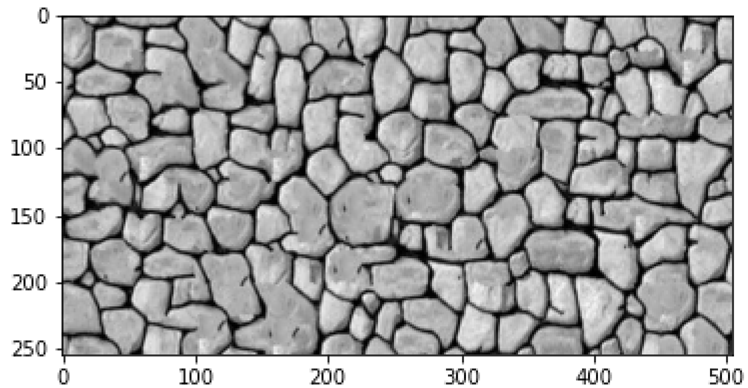
```
        elif (i==0):                          # first row (but not first tile)
            left_tile = tile_map[i,j-1]
            tilestrip = tiledb[left_tile,right_ind]
            dbstrips = tiledb[:,left_ind]
            matches = topkmatch(tilestrip,dbstrips,k)

        elif (j==0):                          # first column (but not first row)
            above_tile = tile_map[i-1,j]
            tilestrip = tiledb[above_tile,bottom_ind]
            dbstrips = tiledb[:,top_ind]
            matches = topkmatch(tilestrip,dbstrips,k)

        else:                                 # neigbors above and to the left
            left_tile = tile_map[i,j-1]
            tilestrip_1 = tiledb[left_tile,right_ind]
            dbstrips_1 = tiledb[:,left_ind]
            above_tile = tile_map[i-1,j]
            tilestrip_2 = tiledb[above_tile,bottom_ind]
            dbstrips_2 = tiledb[:,top_ind]
            # concatenate the two strips
            tilestrip = np.concatenate((tilestrip_1,tilestrip_2))
            dbstrips = np.concatenate((dbstrips_1,dbstrips_2),axis=1)
            matches = topkmatch(tilestrip,dbstrips,k)

        #choose one of the k matches at random
        tile_map[i,j] = matches[np.random.randint(0,k)]

    output = synth_quilt(tile_map,tiledb,tilesize,overlap)

    return output
```

In [112]:
```
# load in rock_wall.jpg
I = plt.imread('rock_wall.jpg')
I = np.mean(I, axis = 2)
plt.imshow(I,cmap=plt.cm.gray)
plt.show()
```



In [113]:
```
# (0) default parameters
print("default parameters")
out = quilt_demo(I)
plt.imshow(out,cmap=plt.cm.gray)
plt.show()
```

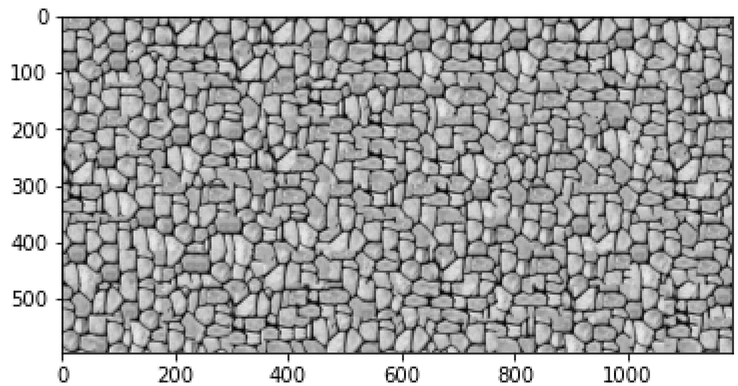
```
default parameters
library has ntiles =  29241 each with npix =  900
row:
0
1
2
3
4
5
6
7
8
9
```



In [114]:

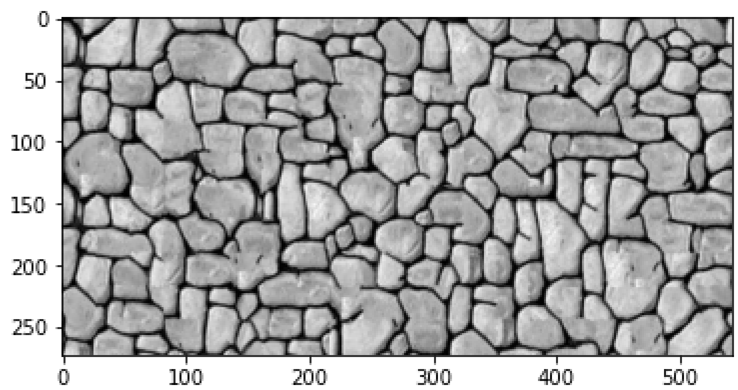
```
# (1) increased tile size
tilesize = (64, 64)
print("increased tilesize to ", tilesize)
out = quilt_demo(I, tilesize = tilesize)
plt.imshow(out,cmap=plt.cm.gray)
plt.show()
```

```
increased tilesize to  (64, 64)
library has ntiles =  18769 each with npix =  4096
row:
0
1
2
3
4
5
6
7
8
9
```



In [119]:
```
# (2) decrease the overlap
overlap = 3
print("decrease overlap to ", overlap)
out = quilt_demo(I, overlap = overlap)
plt.imshow(out,cmap=plt.cm.gray)
plt.show()
```

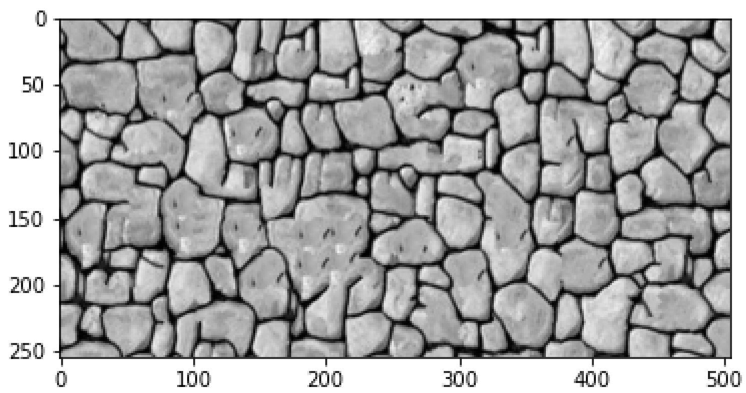
```
decrease overlap to  3
library has ntiles =  29241 each with npix =  900
row:
0
1
2
3
4
5
6
7
8
9
```



In [116]:

```
# (3) increase the value for K.
k = 10
print("increase k value to ", k)
out = quilt_demo(I, k = k)
plt.imshow(out, cmap=plt.cm.gray)
plt.show()
```

```
increase k value to  10
library has ntiles =  29241 each with npix =  900
row:
0
1
2
3
4
5
6
7
8
9
```

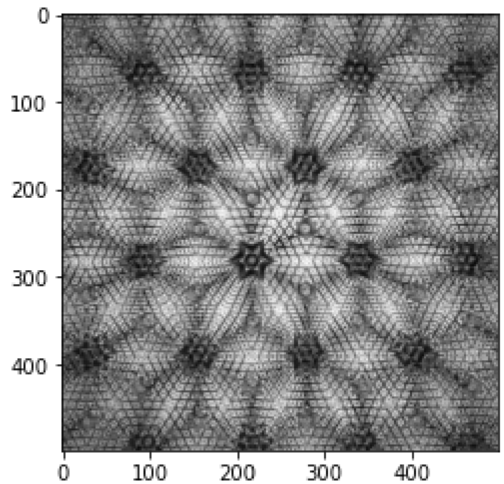


For each result shown, explain here how it differs visually from the default setting of the parameters and explain why: .

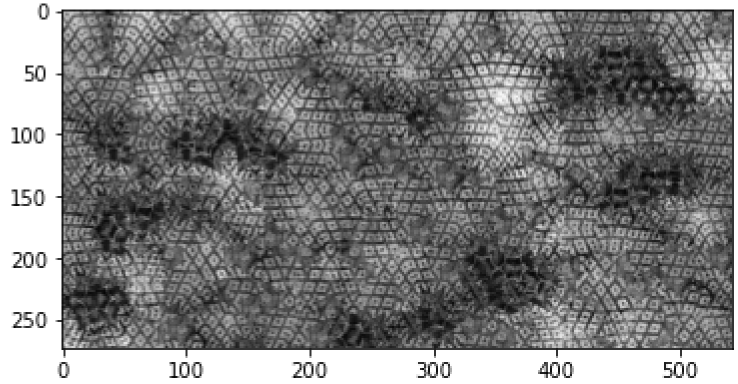
1. increased tile size: The whole result doubled in size when the tile size was doubled because each tile got bigger.

2. decreased overlap: There are less areas where rocks seem joined because the seams have less total variation
3. increased k The pattern seems more random and repeats less because it uses more samples from more parts of the input image.

In [128]:
```
#
# load in yourimage1.jpg
#
# call quilt_demo, experiment with parameters as needed to get a good result
#
# display your source image and the resulting synthesized texture
#
I1 = plt.imread('yourimage1.jpg')
I1 = np.mean(I1, axis = 2)
plt.imshow(I1,cmap=plt.cm.gray)
plt.show()
out1 = quilt_demo(I1, overlap = 3, k = 4)
plt.imshow(out1,cmap=plt.cm.gray)
plt.show()
```

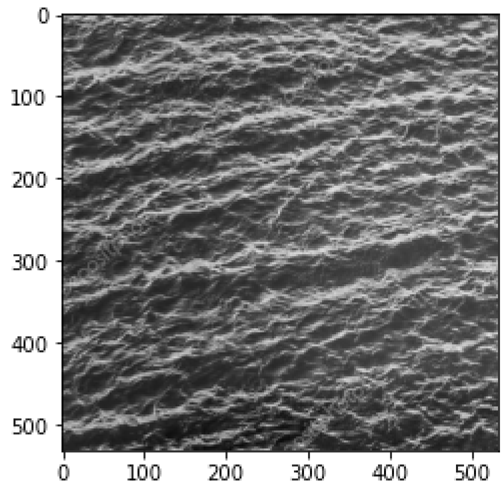



```
library has ntiles =  220900 each with npix =  900
row:
0
1
2
3
4
5
6
7
8
9
```

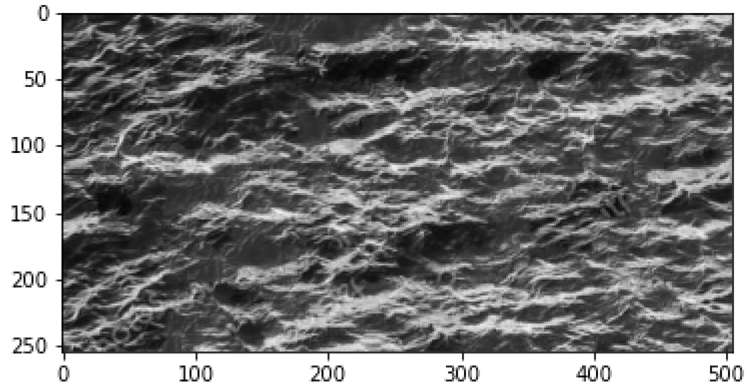


In [ ]:
```
#
# load in yourimage2.jpg
#
# call quilt_demo, experiment with parameters as needed to get a good result
#
# display your source image and the resulting synthesized texture
#
```

In [127]:
```
I2 = plt.imread('yourimage2.jpg')
I2 = np.mean(I2, axis = 2)
plt.imshow(I2,cmap=plt.cm.gray)
plt.show()
out2 = quilt_demo(I2)
plt.imshow(out2,cmap=plt.cm.gray)
plt.show()
```




```
library has ntiles =  253009 each with npix =  900
row:
0
1
2
3
4
5
6
7
8
9
```



In [ ]: