```
'''
Created on Oct 13, 2010.  Last modified Oct 13, 2014.

@author: anderson

Annotated Heapsort program.  Basic ideas:
1. Start with items to be sorted in positions 1 through n of a list a[].
2. Arrange those items into a max-heap, so that a[1] is the largest element.
3. for i = n downto 2:
   a. exchange a[1] and a[i]
   b. Percolate down a[1] so that a[1] .. a[i-1]  again form  a heap.
'''

import random

# This uses a max heap

swapcount = 0

def swap(a, i, j):
    ' excahange the values of a[i] and a[j]'
    global swapcount
    swapcount += 1
    a[i], a[j] = a[j], a[i]

def percolateDown(a,i, n):
    """Within the n elements of A to be "re-heapified", the two subtrees of A[i]
       are already maxheaps. Repeatedly exchange the element currently in A[i] with
       the largest of its children until the tree whose root is a[i] is a max heap. """
    current = i # root position for subtree we are heapifying
    lastNodeWithChild = n//2  # if a node number is higher than this, it is a leaf.
    while current <= lastNodeWithChild:
        max = current
        if a[max] < a[2*current]:  # if it is larger than its left child.
            max = 2*current
        if 2*current < n and a[max] < a[2*current+1]:  # But if there is a right child,
            max = 2*current + 1                # right child may be larger than either
        if max == current:
            break  # larger than its children, so we are done.
        swap(a, current, max) # otherwise, exchange, move down tree, and check again.
        current = max

def percolateUp(a,n):
    ''' Assume that elements 1 through n-1 are a heap; add element n and "re-heapify".  '''
    # compare to parent and swap until not larger than parent.
    current = n
    while current > 1:  # or until this value is in the root.
        if a[current//2] >= a[current]:
            break
        swap(a, current, current//2)
        current //= 2

# The next two functions do the same thing; each takes an unordered
# array and turns it into a max-heap.  In the homework, you will show
```

```
# that the second is much more efficient than the first.
# So this first one is not actually called in this code.
def heapifyByInsert(a, n):
    """ Repeatedly insert elements into the heap.
        Worst case number of element exchanges:
            sum of depths of nodes."""
    for i in range(2, n+1):
        percolateUp(a, i)

def buildHeap(a, n):
    """ Each time through the loop, each of node i's two
        subtreees is already a heap.
        Find the correct position to move the root down to
        in order to "reheapify."
        Worst case number of element exchanges:
            sum of heights of nodes."""
    for i in range (n//2, 0, -1):
        percolateDown(a, i, n)

def heapSort(a, n):
    heapifyByInsert(a, n)
    for i in range(n, 1, -1):
        swap(a, 1, i)
        percolateDown(a, 1, i-1)

# Some code to test heapSort by ranndomly generating an array and sorting it.
# Counts the number of exchanges.  To compare the two heap-building approaches, try substituting
#   heapifyByInsert for buildHeap in the heapSort code.
n = 500
source = list(range(n))
a = [0]
for i in range(n):
    next = random.choice(source)
    a += [next]
    source.remove(next)
print("unsorted array:", a[1:])

heapSort(a, n)
print("sorted array:  ", a[1:])
print ("number of exchanges: ", swapcount)
```