```python
import random


class Quicksort(object):
    def __init__(self, data):
        self.__data = data

    def sort(self):
        self.quicksort(0, len(self.__data) - 1)

        return self.__data

    def quicksort(self, start, end):
        if end == start:
            return

        data = self.__data

        pivot = data[random.randint(start, end)]

        left = start
        right = end

        while left < right:
            while data[left] < pivot:
                left += 1

            while data[right] > pivot:
                right -= 1

            if left <= right:
                data[left], data[right] = data[right], data[left]
                left += 1
                right -= 1

        if start < right:
            self.quicksort(start, right)

        if left < end:
            self.quicksort(left, end)


def main():
    numbers = [325432, 989, 547510, 3, -93, 189019, 5042, 123,
               597, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 42, 7506,
               184, 184, 2409, 45, 824, 4, -2650, 9, 662, 3928,
               -170, 45358, 395, 842, 7697, 110, 14, 99, 221]

    qs = Quicksort(numbers)
    output = qs.sort()

    print(output)


if __name__ == '__main__':
    main()
```

File - /Users/john/Projects/code-catalog-python/catalog/suggested/sort_select/merge_sort.py

```python
class MergeSort(object):
    def __init__(self, numbers):
        self.values = numbers
        self.count = len(numbers)

    def sort(self):
        self.merge_sort(0, self.count - 1)
        return self.values

    def merge_sort(self, low, high):
        if low < high:
            mid = (low + high) // 2

            self.merge_sort(low, mid)
            self.merge_sort(mid + 1, high)
            self.merge(low, mid, high)

    def merge(self, low, mid, high):
        b = []
        i = low
        j = mid + 1

        while i <= mid and j <= high:
            if self.values[i] <= self.values[j]:
                b.append(self.values[i])
                i += 1
            else:
                b.append(self.values[j])
                j += 1

        while i <= mid:
            b.append(self.values[i])
            i += 1

        while j <= high:
            b.append(self.values[j])
            j += 1

        for index, val in enumerate(b):
            self.values[low + index] = val
```

```python
from ..lists.linked_queue import LinkedQueue


def merge(S1, S2, S):
    """Merge two sorted queue instances S1 and S2 into empty queue S."""
    while not S1.is_empty() and not S2.is_empty():
        if S1.first() < S2.first():
            S.enqueue(S1.dequeue())
        else:
            S.enqueue(S2.dequeue())
    while not S1.is_empty():  # move remaining elements of S1 to S
        S.enqueue(S1.dequeue())
    while not S2.is_empty():  # move remaining elements of S2 to S
        S.enqueue(S2.dequeue())


def merge_sort(S):
    """Sort the elements of queue S using the merge-sort algorithm."""
    n = len(S)
    if n < 2:
        return  # list is already sorted
    # divide
    S1 = LinkedQueue()  # or any other queue implementation
    S2 = LinkedQueue()
    while len(S1) < n // 2:  # move the first n//2 elements to S1
        S1.enqueue(S.dequeue())
    while not S.is_empty():  # move the rest to S2
        S2.enqueue(S.dequeue())
    # conquer (with recursion)
    merge_sort(S1)  # sort first half
    merge_sort(S2)  # sort second half
    # merge results
    merge(S1, S2, S)  # merge sorted halves back into S
```

File - /Users/john/Projects/code-catalog-python/catalog/suggested/sort_select/quick_queue.py

```python
from ..lists.linked_queue import LinkedQueue


def quick_sort(S):
    """Sort the elements of queue S using the quick-sort algorithm."""
    n = len(S)
    if n < 2:
        return  # list is already sorted
    # divide
    p = S.first()  # using first as arbitrary pivot
    L = LinkedQueue()
    E = LinkedQueue()
    G = LinkedQueue()
    while not S.is_empty():  # divide S into L, E, and G
        if S.first() < p:
            L.enqueue(S.dequeue())
        elif p < S.first():
            G.enqueue(S.dequeue())
        else:  # S.first() must equal pivot
            E.enqueue(S.dequeue())
    # conquer (with recursion)
    quick_sort(L)  # sort elements less than p
    quick_sort(G)  # sort elements greater than p
    # concatenate results
    while not L.is_empty():
        S.enqueue(L.dequeue())
    while not E.is_empty():
        S.enqueue(E.dequeue())
    while not G.is_empty():
        S.enqueue(G.dequeue())
```

```python
import random


def partition(vector, left, right, pivot_index):
    pivot_value = vector[pivot_index]
    # Move pivot to end
    vector[pivot_index], vector[right] = vector[right], vector[pivot_index]
    store_index = left
    for i in range(left, right):
        if vector[i] < pivot_value:
            vector[store_index], vector[i] = vector[i], vector[store_index]
            store_index += 1
    # Move pivot to its final place
    vector[right], vector[store_index] = vector[store_index], vector[right]
    return store_index


def _select(vector, left, right, k):
    """
    Returns the k-th smallest, (k >= 0), element of vector within
    vector[left:right+1] inclusive.
    """
    while True:
        # select pivot_index between left and right
        pivot_index = random.randint(left, right)
        pivot_new_index = partition(vector, left, right, pivot_index)
        pivot_dist = pivot_new_index - left
        if pivot_dist == k:
            return vector[pivot_new_index]
        elif k < pivot_dist:
            right = pivot_new_index - 1
        else:
            k -= pivot_dist + 1
            left = pivot_new_index + 1


def select(vector, k, left=None, right=None):
    """
    Returns the k-th smallest, (k >= 0), element of vector within vector[left:right+1].
    left, right default to (0, len(vector) - 1) if omitted
    """
    if left is None:
        left = 0
    lv1 = len(vector) - 1
    if right is None:
        right = lv1
    assert vector and k >= 0, "Either null vector or k < 0 "
    assert 0 <= left <= lv1, "left is out of range"
    assert left <= right <= lv1, "right is out of range"
    return _select(vector, left, right, k)


if __name__ == '__main__':
    v = [9, 8, 7, 6, 5, 0, 1, 2, 3, 4]
    print([select(v, i) for i in range(10)])
```

```python
import random


def quick_select(S, k):
    """Return the kth smallest element of list S, for k from 1 to len(S)."""
    if len(S) == 1:
        return S[0]
    pivot = random.choice(S)  # pick random pivot element from S
    L = [x for x in S if x < pivot]  # elements less than pivot
    E = [x for x in S if x == pivot]  # elements equal to pivot
    G = [x for x in S if pivot < x]  # elements greater than pivot
    if k <= len(L):
        return quick_select(L, k)  # kth smallest lies in L
    elif k <= len(L) + len(E):
        return pivot  # kth smallest equal to pivot
    else:
        j = k - len(L) - len(E)  # new selection parameter
        return quick_select(G, j)  # kth smallest is jth in G
```

File - /Users/john/Projects/code-catalog-python/catalog/suggested/sort_select/quick_inplace.py

```python
def inplace_quick_sort(S, a, b):
    """Sort the list from S[a] to S[b] inclusive using the quick-sort algorithm."""
    if a >= b: return  # range is trivially sorted
    pivot = S[b]  # last element of range is pivot
    left = a  # will scan rightward
    right = b - 1  # will scan leftward
    while left <= right:
        # scan until reaching value equal or larger than pivot (or right marker)
        while left <= right and S[left] < pivot:
            left += 1
        # scan until reaching value equal or smaller than pivot (or left marker)
        while left <= right and pivot < S[right]:
            right -= 1
        if left <= right:  # scans did not strictly cross
            S[left], S[right] = S[right], S[left]  # swap values
            left, right = left + 1, right - 1  # shrink range

    # put pivot into its final place (currently marked by left index)
    S[left], S[b] = S[b], S[left]
    # make recursive calls
    inplace_quick_sort(S, a, left - 1)
    inplace_quick_sort(S, left + 1, b)
```

```python
def insertion_sort(A):
    """Sort list of comparable elements into non-decreasing order."""
    for i in range(1, len(A)):  # from 1 to n-1
        cur = A[i]  # current element to be inserted
        j = i  # find correct index j for current
        while j > 0 and A[j - 1] > cur:  # element A[j-1] must be after current
            A[j] = A[j - 1]
            j -= 1
        A[j] = cur  # cur is now in the right place


"""
Tests
"""

from random import shuffle

ex1 = [-5, -2.3, 0, 1, 1, 5, 6, 6.5, 7, 12]
shuffle(ex1)
insertion_sort(ex1)
assert ex1 == [-5, -2.3, 0, 1, 1, 5, 6, 6.5, 7, 12]
```

```python
def merge(S1, S2, S):
    """Merge two sorted Python lists S1 and S2 into properly sized list S."""
    i = j = 0
    while i + j < len(S):
        if j == len(S2) or (i < len(S1) and S1[i] < S2[j]):
            S[i + j] = S1[i]  # copy ith element of S1 as next item of S
            i += 1
        else:
            S[i + j] = S2[j]  # copy jth element of S2 as next item of S
            j += 1


def merge_sort(S):
    """Sort the elements of Python list S using the merge-sort algorithm."""
    n = len(S)
    if n < 2:
        return  # list is already sorted
    # divide
    mid = n // 2
    S1 = S[0:mid]  # copy of first half
    S2 = S[mid:n]  # copy of second half
    # conquer (with recursion)
    merge_sort(S1)  # sort copy of first half
    merge_sort(S2)  # sort copy of second half
    # merge results
    merge(S1, S2, S)  # merge sorted halves back into S
```

```python
import math


def merge(src, result, start, inc):
    """Merge src[start:start+inc] and src[start+inc:start+2*inc] into result."""
    end1 = start + inc  # boundary for run 1
    end2 = min(start + 2 * inc, len(src))  # boundary for run 2
    x, y, z = start, start + inc, start  # index into run 1, run 2, result
    while x < end1 and y < end2:
        if src[x] < src[y]:
            result[z] = src[x]
            x += 1
        else:
            result[z] = src[y]
            y += 1
        z += 1  # increment z to reflect new result
    if x < end1:
        result[z:end2] = src[x:end1]  # copy remainder of run 1 to output
    elif y < end2:
        result[z:end2] = src[y:end2]  # copy remainder of run 2 to output


def merge_sort(S):
    """Sort the elements of Python list S using the merge-sort algorithm."""
    n = len(S)
    logn = math.ceil(math.log(n, 2))
    src, dest = S, [None] * n  # make temporary storage for dest
    for i in (2 ** k for k in range(logn)):  # pass i creates all runs of length 2i
        for j in range(0, n, 2 * i):  # each pass merges two length i runs
            merge(src, dest, j, i)
        src, dest = dest, src  # reverse roles of lists
    if S is not src:
        S[0:n] = src[0:n]  # additional copy to get results to S
```

```python
from .merge_sort_recur import merge_sort


class _Item:
    """Lightweight composite to store decorated value for sorting."""
    __slots__ = '_key', '_value'

    def __init__(self, k, v):
        self._key = k
        self._value = v

    def __lt__(self, other):
        return self._key < other._key  # compare items based on their keys


def decorated_merge_sort(data, key=None):
    """Demonstration of the decorate-sort-undecorate pattern."""
    if key is not None:
        for j in range(len(data)):
            data[j] = _Item(key(data[j]), data[j])  # decorate each element
    merge_sort(data)  # sort with existing algorithm
    if key is not None:
        for j in range(len(data)):
            data[j] = data[j]._value  # undecorate each element
```