Before you turn this problem in, make sure everything runs as expected. First, **restart the kernel** (in the menubar, select Kernel→Restart) and then **run all cells** (in the menubar, select Cell→Run All).

Make sure you fill in any place that says `YOUR CODE HERE` or "YOUR ANSWER HERE", as well as your name and collaborators below:

In [534]:
```
NAME = "Jingren Wang"
COLLABORATORS = "N.A."
```

CS110 Fall 2019 - Assignment 1

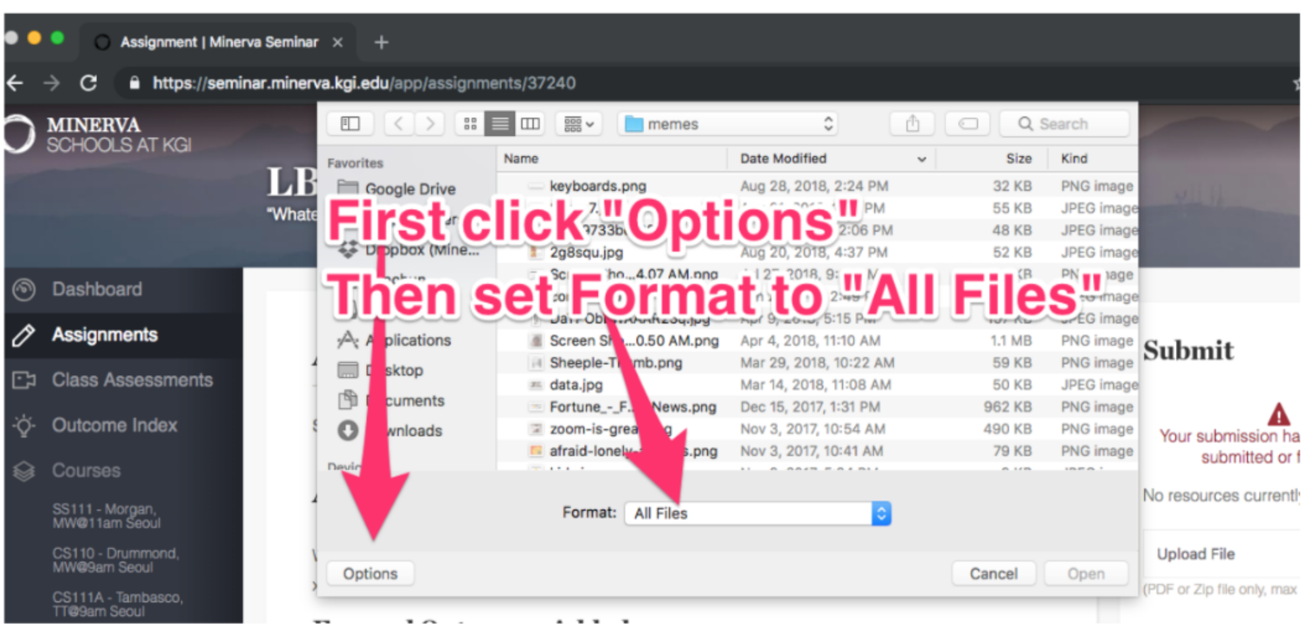
Divide and Conquer Sorting Algorithms

This assignment focuses on the implementation of sorting algorithms and analyzing their performance both mathematically (using theoretical arguments on the asymptotic behavior of algorithms) and experimentally (i.e., running experiments for different input arrays and plotting relevant performance results).

Every CS110 assignment begins with a check-up on your class responsibilities and professional standing, as well as your ability to address one of the course LOs #ComputationalSolutions. Thus to complete the first part of this assignment, you will need to take a screenshot of your CS110 dashboard on Forum where the following is visible: your name. your absences for the course have been set to excused up to session 2.2 (inclusively). This will be evidence that you have submitted acceptable pre-class and make-up work for a CS110 session you may have missed. Check the specific CS110 make-up and pre-class policies in the syllabus of the course.

NOTES:

1. Your assignment submission needs to include the following resources:
    - A PDF file must be the first resource. This file must be generated from the template notebook where you have written all of the answers (check this link for instructions on how to do this). Make sure that the PDF displays properly (all text and code can be seen within the paper margins).
    - Make sure that you submit a neat, clearly presented, and easy-to-read PDF. Please make sure to include page numbers
    - Your second resource must be the template notebook you have downloaded from the gist provided and where you included your answers. Submit this file directly following the directions in this picture:



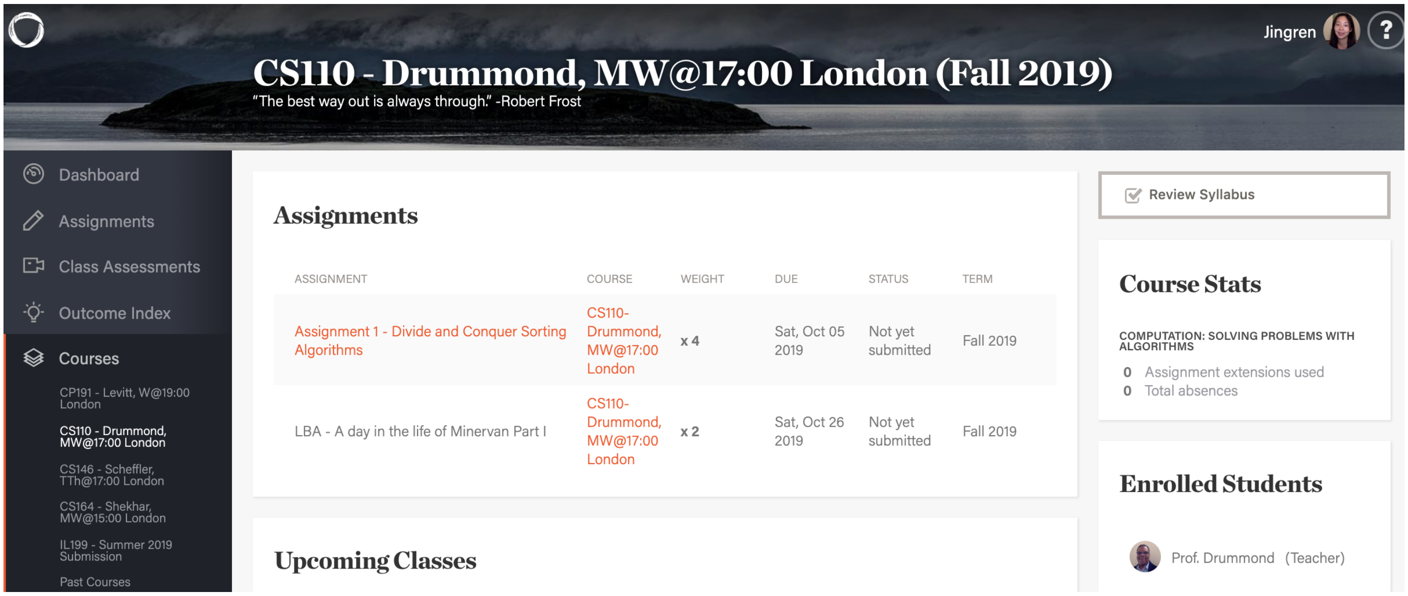
1. Questions (1)-(7) will be graded on the indicated LOs, please make sure to consult their descriptions and rubrics in the course syllabus. You will not be penalized for not attempting the optional challenge.
2. After completing the assignment, evaluate the application of the HCs you have identified prior to and while you were working on this assignment and footnote them (refer to these guidelines (https://docs.google.com/document/d/1s7yOVOtMIaHQdKLeRmZbq1gRqwJKfezBsfru9Q6PcHw/edit) on how to incorporate HCs in your work). Here are some examples of weak applications of some of the relevant HCs:

    - Example 1: “#algorithms: I wrote an implementation of the Bubble sort”.
        - This is an extremely superficial use of the HC in a course on Algorithms, and your reference will be graded accordingly. Instead, consider what constitutes an algorithm (see Cormen et al, sections 1.1 and 1.2). Once you have a good definition of an algorithm, think of how this notion helped you approach the implementation of the algorithm, analyze its complexity and understand why it’s important to write an optimal python implementation of the algorithm.
    - Example 2: “#dataviz: I plotted nice curves showing the execution time of bubble sort, or I plotted beautiful curves with different colors and labels.”
        - Again, these two examples are very superficial uses of the HC #dataviz. Instead consider writing down how do the plots and figures helped you interpret, analyze and write concluding remarks from your experiments. Or write about any insight you included in your work that came from being able to visualize the curves.
    - Example 3: “#professionalism: I wrote a nice paper/article that follows all the directions in this assignment.”
        - By now, you should realize that this is a poor application of the HC #professionalism. Instead, comment on how you actively considered the HC while deciding on the format, length, and style for writing your report.
3. Your code will be tested for similarity using Turnitin, both to other students’ work and examples available online. As such, be sure to cite all references that you used in devising your solution. Any plagiarism attempts will be referred to the ASC.

Complete the following tasks which will be graded in the designated LOs and foregrounded HCs:

Question 1. [HCs #responsibility and #professionalism; #ComputationalSolutions]

Submit a PDF file with a screenshot of your CS110 dashboard with the information described above.



(* please see HCs in the Appendx)

Question 2. [#SortingAlgorithms, #PythonProgramming, #CodeReadability]

Write a Python 3 implementation of the three-way merge sort discussed in class using the code skeleton below. You should also provide at least three test cases (possibly edge cases) that demonstrate the correctness of your code. Your output must be a sorted **Python list**.

In [535]:
```
import numpy as np
import math
import time # for runtime calculation
import matplotlib.pyplot as plt  # for plotting
```

In [536]:
```
global Mrg2_step  # inintialize a global Mrg2_step counter for Q 6 and 7

def merge_two(A1, A2):
    '''
    input must be two lists A1 and A2
    output merged list of length A1 + A2 (no necessarily sorted)
    '''
    global Mrg2_step
    Mrg2_step = 0 # reset global counter to zero

    # initiate empty list of lenth A1+A2 to store sorted values
    n12 = len(A1)+len(A2)
    A12 = [0]*(n12)
    Mrg2_step += 2 # two assignments

    # add sentinels to the end of A1, A2
    A1.append(np.inf)
    A2.append(np.inf)
    Mrg2_step += 2 # two assignments

    i,j = 0,0 # initiate indices
    Mrg2_step += 2 # two assignments

    # element-wise comparison to generate sorted A12
    Mrg2_step += 1 # account for one last 'for' evaluation
    for k in range(n12):
        Mrg2_step += 1
        #print('i=',i)
        if A1[i] <= A2[j]:
            Mrg2_step += 1

            A12[k] = A1[i]
            i += 1
            Mrg2_step += 2

        else:
            Mrg2_step += 1 # 'else' statement

            A12[k] = A2[j]
            #print('  j=',j)
            j += 1
            Mrg2_step += 2 # two assignments

    return A12
```

In [537]:
```
def merge_three(A1, A2, A3):
    '''
    merge three lists by two two-way merges
    '''
    global Mrg3_step # global Mrg3_step counter for Q 6 and 7
    Mrg3_step = 0 # initialized at zero

    A12 = merge_two(A1, A2)
    Mrg3_step += Mrg2_step  # add in steps from merge_two()
    Mrg3_step += 1 # assignment to A12

    A123 = merge_two(A12, A3)
    Mrg3_step += Mrg2_step  # add in steps from merge_two()
    Mrg3_step += 1 # assignment to A12

    return A123
```

In [538]:
```
def threeWayMerge(A):
    """Implements three-way marge sort

    Input:
    A: a Python list OR numpy array (your code should work with both of
 these data types)

    Output: a sorted Python list"""

    # check input validity
    assert(all(isinstance(A[i], (int,float)) for i in range(len(A)))) #
 for test codes

    # if every item in A is a float or integer
    if all(isinstance(A[i], (int,float)) for i in range(len(A))):

        A = list(A) # cast input to a list object
        n = len(A)

        assert(n >= 1)
        if n < 1:
            raise Exception('input length less than 1')

        elif n==1:
            return A

        elif n==2:
            A.sort()
            return A

        # else continum subdivision
        else:

            # DIVIDE a problem into three subproblems
            m = n//3

            # CONQUER subproblems by solving recursively
            A1 = threeWayMerge(A[0:m])
            A2 = threeWayMerge(A[m:m*2])
            A3 = threeWayMerge(A[m*2:n])

            # COMBINE three sublists into single list
            A = merge_three(A1, A2, A3)

            return A

    else:
        raise Exception('input is not a number list or numpy array.')
```

In [539]:
```
A = [1,2,3]
all(isinstance(A[i], (int,float)) for i in range(len(A)))
```

Out[539]: True

In [540]:
```
### implement 8 test cases below:
import unittest

class TestThreeWayMerge(unittest.TestCase):

    def test_1(self):
        # test case 1: worst case with max-sorted input
        A = list(range(10,0,-1))  # [10,9,...2,1]
        A_sorted = list(range(1,11))     # [1,2,....9,10]
        self.assertEqual(threeWayMerge(A), A_sorted)

    def test_2(self):
        # test case 2: best case with min-sorted input
        A = list(range(1,11))   # [1,2,....9,10]
        A_sorted = list(range(1,11))   # [1,2,....9,10]
        self.assertEqual(threeWayMerge(A), A_sorted)

    def test_3(self):
        # test case 3: identical input - array of single number
        A = [88]*10
        A_sorted = A  # [88, 88, 88, 88, 88, 88, 88, 88, 88, 88]
        self.assertEqual(threeWayMerge(A), A_sorted)

    def test_4(self):
        # test case 4: input np.array of length 1
        A = np.random.randn(1)
        A_sorted = A
        self.assertEqual(threeWayMerge(A), A_sorted)

    def test_5(self):
        # test case 5: input np.array of length 2
        A = np.random.randn(2)
        A_sorted = list(np.sort(A))
        self.assertEqual(threeWayMerge(A), A_sorted)

    def test_6(self):
        # test case 6: input an np.array of random floats
        A = 3.9 * np.random.randn(10) - 2
        A_sorted = list(np.sort(A))
        self.assertEqual(threeWayMerge(A), A_sorted)

    def test_7(self):
        # test case 7: input an emply list
        try:
            A = []
            threeWayMerge(A)
        except AssertionError as error:
            print('*Error: input length less than 1.')

    def test_8(self):
        # test case 8: input wrong type
        try:
            A = ['hello!', '3', 4, 8]
            threeWayMerge(A)
```

```
        except AssertionError as error:
            print('*Error: input is not a number list or numpy array.')
```

In [541]:
```
# implement test cases and print out test results
# initiate a testcase object
test3wyMrg = TestThreeWayMerge()

for i in range(1,9):
    idx = str(i)
    test_case = 'test_'+ idx
    test = getattr(test3wyMrg, test_case)
    print(f'{test_case}: ')
    if test() == None:
        print('  >>> test passed!')
```

```
test_1:
  >>> test passed!
test_2:
  >>> test passed!
test_3:
  >>> test passed!
test_4:
  >>> test passed!
test_5:
  >>> test passed!
test_6:
  >>> test passed!
test_7:
*Error: input length less than 1.
  >>> test passed!
test_8:
*Error: input is not a number list or numpy array.
  >>> test passed!
```

In [542]:
```
##### Please ignore this cell. This cell is for us to implement the test
s
# to see if your code works properly.
```

Question 3. [(#SortingAlgorithms, #PythonProgramming, #CodeReadability, #ComputationalCritique]

Implement a second version of a three-way merge sort that calls selection sort when sublists are below a certain length (of your choice) rather than continuing the subdivision process. Justify what might be an appropriate threshold for the input array for applying selection sort.

In [543]:
```
global sele_step
sele_step = 0

def selectionSort(A):
    '''
    implement selection sort
        input:  must be a list
        output:  a sorted list
    *function in place
    '''
    global sele_step   # global counter for steps of selection Sort

    # if every item in A is a float or integer
    if all(isinstance(A[i], (int,float)) for i in range(len(A))):

        n = len(A)
        sele_step += 1

        sele_step += 1 # account for last 'for statement' evaluation
        for i in range(n):  # i in 0 to n-1
            sele_step += 1 # if statement
            min_idx = i  # assume the first element is the minimum
            sele_step += 1 # assignment

            sele_step += 1  # account for last 'for statement' evaluatio
n
            for j in range(i+1,n):   # j in i+1 to n
                sele_step += 1
                if A[j] < A[min_idx]:
                    sele_step += 1  # if statement
                    min_idx = j
                    sele_step += 1   # update minimal index

            # swap A[i] with A[min_idx] after comparison
            A[i], A[min_idx] = A[min_idx], A[i]
            sele_step += 3 # python three-step swap using an intermediat
e tuple

    else:
        raise Exception('input must be a number list objbct')

    return A
```

In [544]:
```
def extendedThreeWayMerge(A, k):
    """Implements the second version of a three-way merge sort

    Input:
    A: a Python list OR numpy array (your code should work with both of
 these data types)
    k: choice of stopping length of sublist, from which selectionSort()
 is called

    Output: a sorted Python list
    """

    # check input validity for k
    if not isinstance(k,int) or (k <= 0):
        raise Exception ('k must be a positive integer.')

    # check input validity for A
    assert(all(isinstance(A[i], (int,float)) for i in range(len(A)))) #
 for test codes
    # if every item in A is a float or integer
    if all(isinstance(A[i], (int,float)) for i in range(len(A))):

        A = list(A) # cast input to a list object
        n = len(A)

        if n < 1:
            raise Exception('input length less than 1')

        elif n==1:
            return A
        elif n==2:
            A.sort()
            return A

        # call selection sort when length of sublist below threshold k
        elif n <= k:
            print('>>> length of sublist = ', n)
            print('>>> stop subdivision!')
            return selectionSort(A)

        # else continum subdivision
        else:
            # Divide a problem into three subproblems
            m = n//3

            # CONQUER subproblems by solving recursively
            A1 = extendedThreeWayMerge(A[0:m], k)
            print('A1 =', A1)
            A2 = extendedThreeWayMerge(A[m:m*2], k)
            print('A2 =', A2)
            A3 = extendedThreeWayMerge(A[m*2:n], k)
            print('A3 =', A3)

            # COMBINE: merge three sublists,
            # length of each no shorter than threshold k
```

```
            print('>>> start merge!')
            A = merge_three(A1, A2, A3)


        return A

    else:
        raise Exception('input is not a number list or numpy array.')
```

A case study:

before full-scale runtime plotting, it is good to gain some intuition of the algorithm's base-case performance through a dummy case study. Here, we investigate run time variation with a small input size of len(A) = 20,

```
i.e. A = list(range(20,0,-1))
```

in strict descending order to simulate worst case performance of the sorting algorithm.

- Embedded 'print( )' plug-ins throughout the method codes are switched on to draw out a reader-friendly flow of operations within an otherwise 'blackbox' method call.

In [545]:
```
A = list(range(20,0,-1))

# scenario 0: k <= 2, impose no stopping effect
n = len(A)
k = 2

print(' A =', A)
print(' k =', k)
print('')

# get runtime
start_time = time.clock()
extendedThreeWayMerge(A,k)
print('A =', A)
T_merge_k = (time.clock()-start_time)*1000 #convert to milisecond (ms)
print('')
print(f'run time at k = {k} is %.3f'% T_merge_k, 'ms')

# all sublists break down to bases, with length of 1 and 2, before mergi
ng
# running time relatively large
```

```
 A = [20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3,
2, 1]
 k = 2

A1 = [19, 20]
A2 = [17, 18]
A3 = [15, 16]
>>> start merge!
A1 = [15, 16, 17, 18, 19, 20]
A1 = [13, 14]
A2 = [11, 12]
A3 = [9, 10]
>>> start merge!
A2 = [9, 10, 11, 12, 13, 14]
A1 = [7, 8]
A2 = [5, 6]
A1 = [4]
A2 = [3]
A3 = [1, 2]
>>> start merge!
A3 = [1, 2, 3, 4]
>>> start merge!
A3 = [1, 2, 3, 4, 5, 6, 7, 8]
>>> start merge!
A = [20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3,
2, 1]

run time at k = 2 is 4.622 ms
```

In [546]:
```
# scenario 1: k = 3//n
n = len(A)
k = n//3

print(' A =', A)
print(' k =', k)
print('')

# get runtime
start_time = time.clock()
extendedThreeWayMerge(A,k)
print('A =', A)
T_merge_k = (time.clock()-start_time)*1000 #convert to milisecond (ms)
print('')
print(f'running time at k = {k} is %.3f'% T_merge_k, 'ms')


# first round subdivision gives three sublists of length 6, 6 and 8
# the if statement n <= k(=6) evaluates to false for A1 and A2, stops su
bdivision
# however, A3 of length 8 > 6, thus allowed a further division into 2-2-
4
# since length of 4 is less than k = n//3 = 6, A3 is stopped from furthe
r subdivision
```

```
 A = [20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3,
2, 1]
 k = 6

>>> length of sublist =  6
>>> stop subdivision!
A1 = [15, 16, 17, 18, 19, 20]
>>> length of sublist =  6
>>> stop subdivision!
A2 = [9, 10, 11, 12, 13, 14]
A1 = [7, 8]
A2 = [5, 6]
>>> length of sublist =  4
>>> stop subdivision!
A3 = [1, 2, 3, 4]
>>> start merge!
A3 = [1, 2, 3, 4, 5, 6, 7, 8]
>>> start merge!
A = [20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3,
2, 1]

running time at k = 6 is 2.614 ms
```

In [547]:
```
# scenario 2: k = 3//n+2
n = len(A)
k = n//3+2

print(' A =', A)
print(' k =', k)
print('')

# get runtime
start_time = time.clock()
extendedThreeWayMerge(A,k)
print('A =', A)
T_merge_k = (time.clock()-start_time)*1000 #convert to milisecond (ms)
print('')
print(f'run time at k = {k} is %.3f'% T_merge_k, 'ms')

# at k = n//3+2 = 8,  sublists with length equal to 8 or less must stop
# all three sublists stops subdivision after 1st round
# this is the minimal subdivision case, and
# selection sort does the main job of sorting three times before merging
```

```
 A = [20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3,
2, 1]
 k = 8

>>> length of sublist =  6
>>> stop subdivision!
A1 = [15, 16, 17, 18, 19, 20]
>>> length of sublist =  6
>>> stop subdivision!
A2 = [9, 10, 11, 12, 13, 14]
>>> length of sublist =  8
>>> stop subdivision!
A3 = [1, 2, 3, 4, 5, 6, 7, 8]
>>> start merge!
A = [20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3,
2, 1]

run time at k = 8 is 2.747 ms
```

In [548]:
```
# scenario 3: k = n
n = len(A)
k = n

print(' A =', A)
print(' k =', k)
print('')

# get runtime
start_time = time.clock()
extendedThreeWayMerge(A,k)
T_merge_k = (time.clock()-start_time)*1000 #convert to milisecond (ms)
print('')
print(f'run time at k = {k} is %.3f'% T_merge_k, 'ms')

# no subdivision at all
# a single selection sort call
```

```
 A = [20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3,
2, 1]
 k = 20

>>> length of sublist =  20
>>> stop subdivision!

run time at k = 20 is 0.576 ms
```

Disucssion:

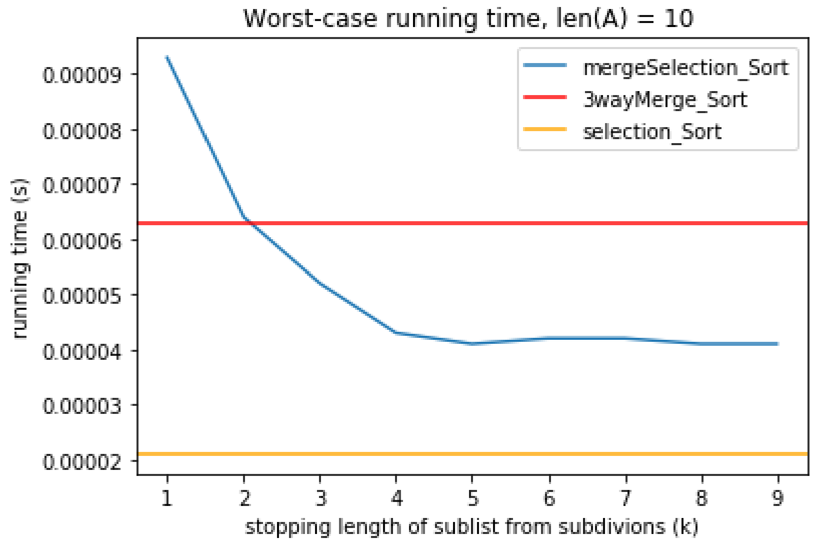
The featured scenarios in the case study above explains the cyclic pattern of runtime performance of extendedThreeWayMerge(A, k) as k grows up, this periodic spike is also illustrated in runtime plots below.

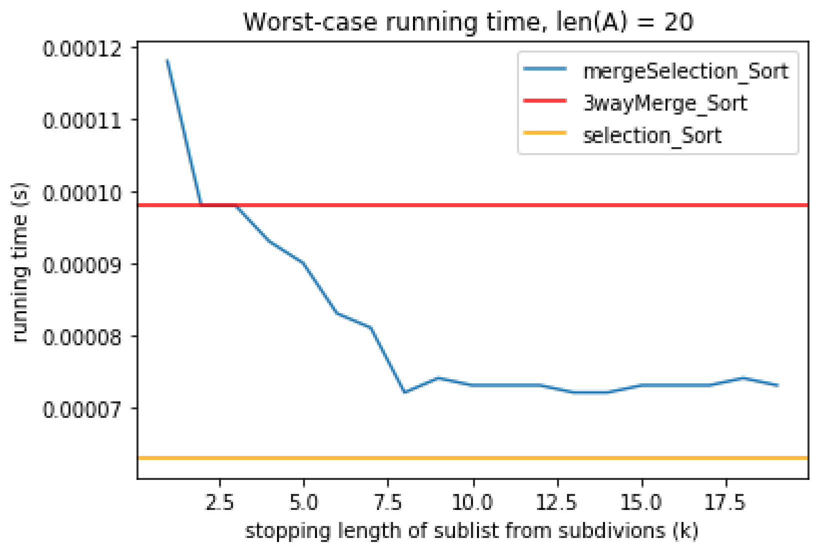
below is a runtime comparison plot illustrating this idea for a large len(A) for runtime accuracy, I reproduce the extendedThreeWayMerge(A, k) without all inserted 'print( )' lines below

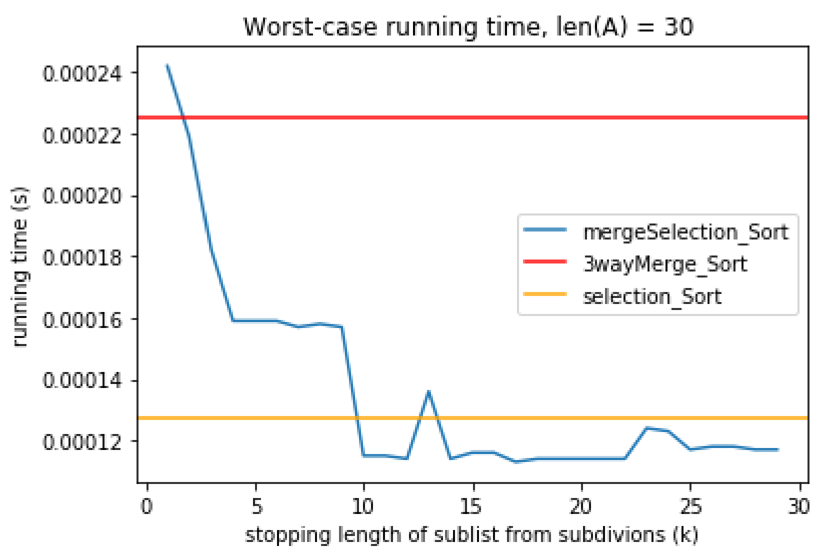
In [549]:
```
def extendedThreeWayMerge(A, k):

    # check input validity
    if not isinstance(k,int) or (k <= 0):
        raise Exception ('k must be a positive integer.')

    # check input validity for A
    assert(all(isinstance(A[i], (int,float)) for i in range(len(A)))) #
 for test codes
    # if every item in A is a float or integer
    if all(isinstance(A[i], (int,float)) for i in range(len(A))):

        A = list(A) # cast input to a list object
        n = len(A)

        if n < 1:
            raise Exception('input length less than 1')

        elif n==1:
            return A
        elif n==2:
            A.sort()
            return A

        # call selection sort when length of sublist below threshold k
        elif n <= k:
            return selectionSort(A)

        # else continum subdivision
        else:

            # Divide a problem into three subproblems
            m = n//3

            # CONQUER subproblems by solving recursively
            A1 = extendedThreeWayMerge(A[0:m], k)
            A2 = extendedThreeWayMerge(A[m:m*2], k)
            A3 = extendedThreeWayMerge(A[m*2:n], k)

            # COMBINE: merge three sublists,
            A = merge_three(A1, A2, A3)

        return A

    else:
        raise Exception('input is not a number list or numpy array.')
```

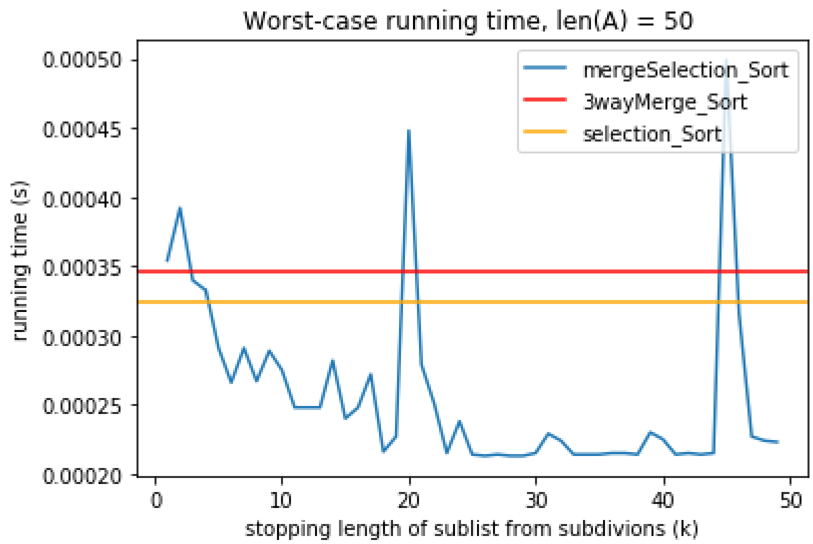
In [550]:
```
# worst-case running time analysis
# assume that input A is in strict descending order
# choose a large len(A)

def runTimePlots(lenA):
    '''
    python plot of worst-case run time comparison among three sorts
    input:
        lenA = chosen length of descending array
    output:
        one python plot of runtime comparison

    '''

    A = list(range(lenA,0,-1))

    # compute runtime for extendedThreeWayMerge(A, k)
    ks = list(range(1,lenA))
    runTimes = []

    for k in ks:
        start_time = time.clock()
        extendedThreeWayMerge(A, k)
        runTime = time.clock()-start_time
        runTimes.append(runTime)

    # compute runtime for threeWayMerge(A)
    start_time = time.clock()
    threeWayMerge(A)
    T_threeWayMerge = time.clock()-start_time

    # compute runtime for selectionSort(A)
    start_time = time.clock()
    selectionSort(A)
    T_selectionSort = time.clock()-start_time

    # plot runtime comparison
    x1 = ks
    y1 = runTimes

    plt.plot(x1,y1, label='mergeSelection_Sort')
    plt.axhline(y=T_threeWayMerge, color='r', linestyle='-', label='3way
Merge_Sort')
    plt.axhline(y=T_selectionSort, color='orange', linestyle='-', label=
'selection_Sort')

    plt.title(f"Worst-case running time, len(A) = {lenA} ")
    plt.xlabel('stopping length of sublist from subdivions (k)')
    plt.ylabel('running time (s)')
    plt.legend()
    plt.show()
```

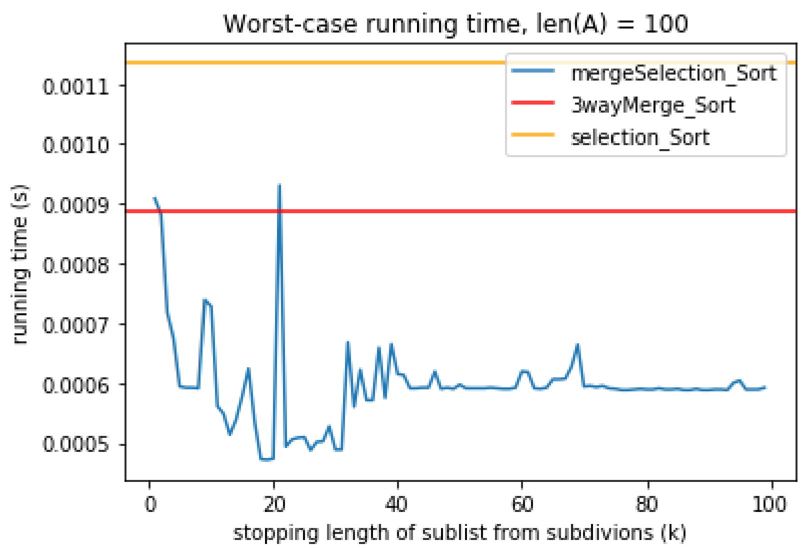
In [551]:
```
# plot runtime against growth of k, at increasing level of input size n
for lenA in [10, 20, 30, 50, 100, 200, 500]:
    runTimePlots(lenA)
```

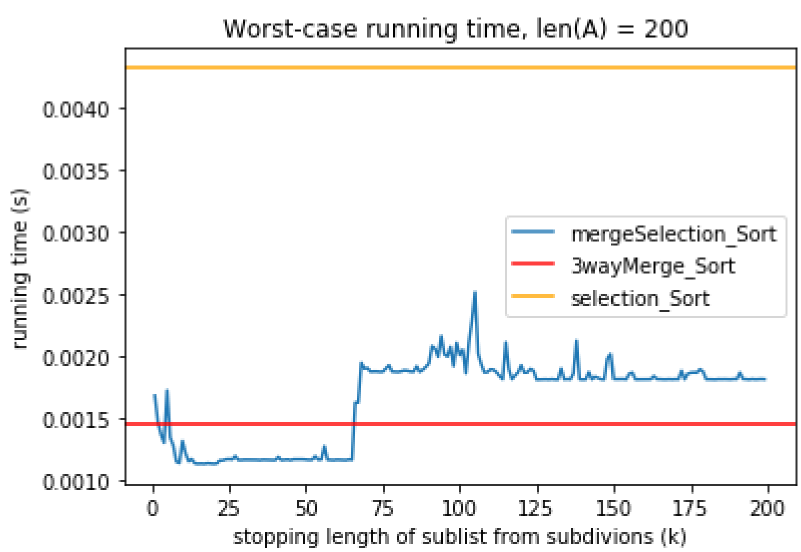

Worst-case running time, len(A) = 10
mergeSelection_Sort
3wayMerge_Sort
selection_Sort
running time (s)
0.00009
0.00008
0.00007
0.00006
0.00005
0.00004
0.00003
0.00002
1 2 3 4 5 6 7 8 9
stopping length of sublist from subdivions (k)

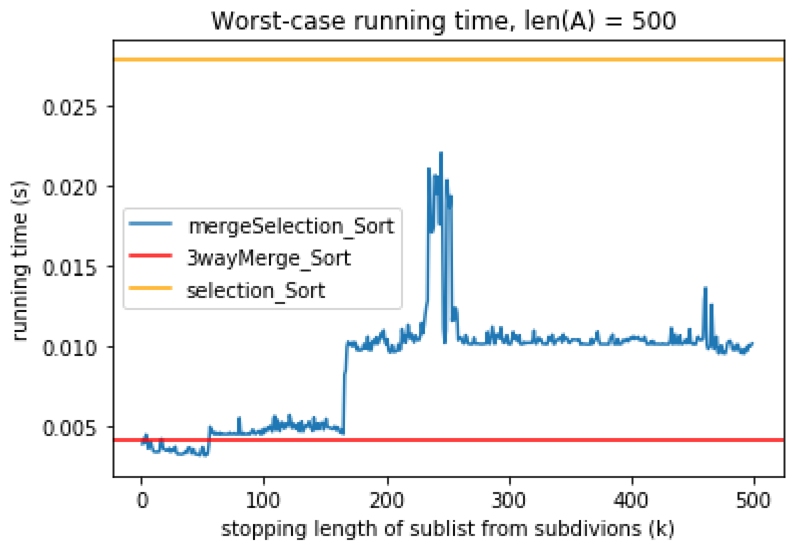

Worst-case running time, len(A) = 20
mergeSelection_Sort
3wayMerge_Sort
selection_Sort
running time (s)
0.00012
0.00011
0.00010
0.00009
0.00008
0.00007
2.5 5.0 7.5 10.0 12.5 15.0 17.5
stopping length of sublist from subdivions (k)


Worst-case running time, len(A) = 30
mergeSelection_Sort
3wayMerge_Sort
selection_Sort
running time (s)
0.00024
0.00022
0.00020
0.00018
0.00016
0.00014
0.00012
0 5 10 15 20 25 30
stopping length of sublist from subdivions (k)


Worst-case running time, len(A) = 50
mergeSelection_Sort
3wayMerge_Sort
selection_Sort
running time (s)
0.00050
0.00045
0.00040
0.00035
0.00030
0.00025
0.00020
0
10
20
30
40
50
stopping length of sublist from subdivions (k)


Worst-case running time, len(A) = 100
mergeSelection_Sort
3wayMerge_Sort
selection_Sort
running time (s)
0.0011
0.0010
0.0009
0.0008
0.0007
0.0006
0.0005
0
20
40
60
80
100
stopping length of sublist from subdivions (k)


Worst-case running time, len(A) = 200
mergeSelection_Sort
3wayMerge_Sort
selection_Sort
running time (s)
0.0040
0.0035
0.0030
0.0025
0.0020
0.0015
0.0010
0
25
50
75
100
125
150
175
200
stopping length of sublist from subdivions (k)



Conclusion:

From the 5 plots above, we might conclude that:

1. when len(A) is small (<= 20), pure selection sort does the best job, so set k to len(A) to convert extendedThreeWayMerge(A,k) to a single selection sort with no subdivisions at all;
2. as len(A) increases from around 30 to 50, threeWayMerge sort gradually outperforms selection sort (yellow line rise up on top of red line), and the hybrid sort outperforms both sorts, this time, best performing k is either n//3+2 or 10 < k < 15, which enables a hybrid of merge and selection;
3. as len(A) --> 200, 300, 500, there is a step-wise runtime growth along k for merge_selection sort after k reaches 60~70, however, runtime at 10 < k < 15 still remainst the lowest regardless of input size

   overall, a rule of thumb for picking a good k to maximize runtime efficiency could be:

```
1) k = len(A) for len(A) < 20 , and
2) 10 < k < 15 for any len(A) > 20
```

In [552]:
```
# Please ignore this cell. This cell is for us to implement the tests
# to see if your code works properly.
```

Question 4 [#SortingAlgorithms, #PythonProgramming, #CodeReadability]

Bucket sort (or Bin sort) is an algorithm that takes as inputs an n-element array and the number of buckets, k, to be used during sorting. Then, the algorithm distributes the elements of the input array into k-different buckets and proceeds to sort the individual buckets. Then, merges the sorted buckets to obtained the sorted array. Here is pseudocode for the BucketSort algorithm:

```
BucketSort( A, k)
    mn = min(A)           # Find minimum value in the array
    mx = max(A)           # Find maximum value in the array
    sz = ceiling((max - min)/k) # divide the range of values in A
                                # into k intervals of size sz
    Buckets ←Array of k list   # Create a blank list of buckets
    for i = 1 to A.length      # Distribute elements in k-buckets
        b = GetBucketNum(A[i], mn, mx, sz, k)
        Buckets[b].Append(A[i])# A[i] is place in bucket b
    for i = 1 to k             # sort buckets individually
        Sort(Buckets[i])
                  # Concatenate contents of sorted buckets
    A = Buckets[1]+Buckets[2]+ . .  +Buckets[k]
    return  A                  # returns sorted list
```

The BucketSort above calls the function **GetBucketNum** (see the pseudocode below) to distribute all the elements of array A into k-buckets. Every element in the array is assigned a bucket number based on its value (positive or negative numbers). **GetBucketNum** returns the bucket number that corresponds to element $A[i]$. It takes as inputs the element of the array, $A[i]$, the max and min elements in A, the size of the intervals in every bucket (e.g., if you have numbers with values between 0 and 100 numbers and 5 buckets, every bucket has an interval of size $20 = [100 - 0]/5$). Notice that in pseudocode the indices of the arrays are from 1 to n. Thus, GetBucketNum consistently returns a number between 1 and n (make sure you account for this in your Python program).

```
GetBucketNum( a, mn, mx, sz, k ) # Assigns a bucket number to
    if a = mx                    # every element in A based
       j = k
    elseif a = mn
       j = 1
    else
        j = 1
        while a > mn+(sz*j)
            j = j + 1
      return j
```

Write a Python 3 implementation of BucketSort that uses the selection sort algorithm for sorting the individual buckets in line 10 of the algorithm.

In [553]:
```
global buktNum_step
buktNum_step = 0

def GetBucketNum(A_i, mn, mx, sz, k):
    '''
    distribute all elements of A into k buckets before bucket-wise sorti
ng
    input:
        A_i is the ith element of array A
        mn, mx are the minimum and maximum element of A
        sz is the bucket size, the interval/range of each bucket values
        k is total number of buckets
    output:
        index j, which is the bucket number that A_i goes to
        j starts from 1
    '''
    global buktNum_step

    # assign the maximum element to the kth bucket
    if A_i == mx:
        buktNum_step += 1 # if statement
        j = k # to align with python 0 indexing
        buktNum_step += 1  # 1 assignment

    # assign the minimum element to the 1st bucket
    elif A_i == mn:
        buktNum_step += 1
        j = 1
        buktNum_step += 1

    # else assign A[i] to the bucket that's
    # j time's interval away from the minimum value

    else:
        buktNum_step += 1  # else statement
        j = 1
        buktNum_step += 1 # 1 assignment

        buktNum_step += 1 # count the last while statement evaluation
        while A_i > mn+sz*j:
            buktNum_step += 1  # while statement evaulation
            j = j+1
            buktNum_step += 1  # assignment

    return j
```

In [554]:
```
def bucketSort(A, k):
    """Implements BucketSort

    Input:
    A: a Python list OR numpy array (your code should work with both of
 these data types)
    k: int, length of A

    Output: a sorted Python list"""

    # check input validity
    assert(type(k)== int)
    assert(k > 1)
    if not isinstance(k,int) or (k <= 0):
        raise Exception ('k must be a positive integer.')

    # check input validity for A
    assert(all(isinstance(A[i], (int,float)) for i in range(len(A)))) #
 for test codes
    # if every item in A is a float or integer
    if all(isinstance(A[i], (int,float)) for i in range(len(A))):

        A = list(A) # cast A into a list object (in case)

        assert(len(A)>=1)
        if len(A) < 1:
            raise Exception('input length less than 1')

        mn = min(A)  # compute minimum value in the array
        mx = max(A)  # compute maximum value in the array
        sz = math.ceil((mx - mn)/k)  # chop range of values in A into
                                     # k intervals of equal sizes sz

        # generate a list of k buckets (sublists)
        Buckets = [ [] for bkt in range(k) ]

        # equally distribute all elements into k-buckets
        for i in range (len(A)):
            b = GetBucketNum(A[i], mn, mx, sz, k)-1 # return bucket numb
er (from 0)
            Buckets[b].append(A[i]) # A[i] is in bucket number j

        sorted_A = [] # initiate empty array to combine sorted buckets

        for s in range (k):
            # sort inividual bucket and concatenate to sorted buckets
            #print(f'unsorted Bucket {s+1} =', Buckets[s])
            sorted_A += selectionSort(Buckets[s])

        return sorted_A

    else:
        raise Exception('A must be either a number list or a numpy arra
y.')
```

In [555]:
```
# create test cases
class TestBucketSort(unittest.TestCase):

    def test_1(self):
        # test case 1: descending ordering (worst case)
        A = list(range(20, 0, -1))
        k = 4
        A_sorted = selectionSort(A)
        self.assertEqual(bucketSort(A, k), A_sorted)

    def test_2(self):
        # test case 2: asscending ordering (best case)
        A = list(range(20))
        k = 5
        A_sorted = selectionSort(A)
        self.assertEqual(bucketSort(A, k), A_sorted)

    def test_3(self):
        # test case 3: identical element value, min = max, sz = 0
        A = [22]*10
        k = 3
        A_sorted = A
        self.assertEqual(bucketSort(A, k), A_sorted)

    def test_4(self):
         # test case 4: random floats
        A = 7.6 * np.random.randn(10) - 3.5
        k = 10
        A_sorted = list(selectionSort(A)) # cast np.array to list
        self.assertEqual(bucketSort(A, k),A_sorted)

    def test_5(self):
        # test case 5: input invalid k type
        try:
            A = np.random.randn(8)
            k = 'k'
            bucketSort(A, k)
        except AssertionError as error:
            print('*TypeError: k must be a positive integer.')

    def test_6(self):
        # test case 6: input invalid k range
        try:
            A = np.random.randn(8)
            k = -3
            bucketSort(A, k)
        except AssertionError as error:
            print('*ValueError: k must be a positive integer.')

    def test_7(self):
        # test case 7: input an emply list
        try:
            A = []
            bucketSort(A, k)
        except AssertionError as error:
```

```
            print('*InputError: input length less than 1.')

    def test_8(self):
        # test case 8: input wrong type
        try:
            A = ['fds', 25, 66.276, '33']
            bucketSort(A, k)
        except AssertionError as error:
            print('*InputError: input is not a number list or numpy arra
y.')
```

In [556]:

```
# implement test cases and print out test results
# initiate a testcase object
testBucket = TestBucketSort()

for i in range(1,9):
    idx = str(i)
    test_case = 'test_'+ idx
    test = getattr(testBucket, test_case)
    print(f'{test_case}: ')
    if test() == None:
        print('  >>> test passed!')
```

```
test_1:
  >>> test passed!
test_2:
  >>> test passed!
test_3:
  >>> test passed!
test_4:
  >>> test passed!
test_5:
*TypeError: k must be a positive integer.
  >>> test passed!
test_6:
*ValueError: k must be a positive integer.
  >>> test passed!
test_7:
*InputError: input length less than 1.
  >>> test passed!
test_8:
*InputError: input is not a number list or numpy array.
  >>> test passed!
```

In [557]:

```
# Please ignore this cell. This cell is for us to implement the tests
# to see if your code works properly.
```

Question 5 [#SortingAlgorithms, #PythonProgramming, #CodeReadability]

Implement a second version of the BucketSort algorithm. This time in line 10 of BucketSort use the Bucket sort recursively until the size of the bucket is less than or equal to k, the base case for the recursion.

In [558]:
```
def extendedBucketSort(A, k):
    """Implements the second version of the BucketSort algorithm

    Input:
    A: a Python list OR numpy array (your code should work with both of
 these data types)
    k: int, length of A

    Output: a sorted Python list"""

    # check input validity
    assert(type(k)== int)
    assert(k > 1)
    if not isinstance(k,int) or (k <= 1):
        raise Exception ('k must be a positive integer greater than 1.')

    # check input validity for A
    assert(all(isinstance(A[i], (int,float)) for i in range(len(A)))) #
 for test codes

    # if every item in A is a float or integer
    if all(isinstance(A[i], (int,float)) for i in range(len(A))):

        A = list(A) # cast A into a list object (in case)

        assert(len(A) >= 1)
        if len(A) < 1:
            raise Exception ('input array length less than 1.')

        mn = min(A)  # compute minimum value in the array
        mx = max(A)  # compute maximum value in the array
        sz = math.ceil((mx - mn)/k) # chop range of values in A into
                                    # k intervals of equal sizes sz

        # check if size of bucket is less than or equal to k,
        # the base case of recursion, and return the sublist
        if sz <= k:
            # return sorted base
            return selectionSort(A)

        else:
            Buckets = [ [] for bkt in range(k)] # generate a list of k b
uckets

            # equally distribute all elements into k-buckets
            for i in range (len(A)):
                b = GetBucketNum(A[i], mn, mx, sz, k)-1 # return bucket
 number (from 0)
                Buckets[b].append(A[i]) # A[i] is in bucket number j

            sorted_A = [] # initiate empty array to combine sorted bucke
ts

            # CONQUER: sort each bucket recursively
            for s in range (k):
```

```
                A = Buckets[s]
                Bucket_s = extendedBucketSort(A, k)

                # COMBINE:
                sorted_A += Bucket_s # merge newly sorted bucket with pa
st sorted buckets
            return sorted_A

    else:
        raise Exception('A must be either a list or a numpy array.')
```

In [559]:
```
# create test cases for recursive bucket sort
# make sure all tests cases from regular bucket sort are passed
# next, stack new test cases particular to extended bucket sort
class TestRecursiveBucket(unittest.TestCase):

    def test_9(self):
        # test case 9: k = 1
        try:
            A = np.random.randn(12)
            k = 1
            extendedBucketSort(A, k)
        except AssertionError as error:
            print('*ValueError: k must be an integer greater than 1.')

    def test_1(self):
        # test case 1: descending ordering (worst case)
        A = list(range(20, 0, -1))
        k = 4
        A_sorted = selectionSort(A)
        self.assertEqual(extendedBucketSort(A, k), A_sorted)

    def test_2(self):
        # test case 2: asscending ordering (best case)
        A = list(range(20))
        k = 5
        A_sorted = selectionSort(A)
        self.assertEqual(extendedBucketSort(A, k), A_sorted)

    def test_3(self):
        # test case 3: identical element value, min = max, sz = 0
        A = [22]*10
        k = 3
        A_sorted = A
        self.assertEqual(extendedBucketSort(A, k), A_sorted)

    def test_4(self):
         # test case 4: random floats
        A = 7.6 * np.random.randn(10) - 3.5
        k = 10
        A_sorted = list(selectionSort(A)) # cast np.array to list
        self.assertEqual(extendedBucketSort(A, k),A_sorted)

    def test_5(self):
        # test case 5: input invalid k type
        try:
            A = np.random.randn(8)
            k = 'k'
            extendedBucketSort(A, k)
        except AssertionError as error:
            print('*TypeError: k must be a positive integer.')

    def test_6(self):
        # test case 6: input invalid k range
        try:
            A = np.random.randn(8)
```

```
            k = -3
            extendedBucketSort(A, k)
        except AssertionError as error:
            print('*ValueError: k must be a positive integer.')

    def test_7(self):
        # test case 7: input an emply list
        try:
            A = []
            extendedBucketSort(A, k)
        except AssertionError as error:
            print('*InputError: input length less than 1.')

    def test_8(self):
        # test case 8: input wrong type
        try:
            A = ['hello!']
            extendedBucketSort(A, k)
        except AssertionError as error:
            print('*InputError: input is not a number list or numpy arra
y.')
```

In [560]:
```
# implement test cases and print out test results
# initiate a testcase object
testRecurBucket = TestRecursiveBucket()

for i in range(1,10):
    idx = str(i)
    test_case = 'test_'+ idx
    test = getattr(testRecurBucket, test_case)
    print(f'{test_case}: ')
    if test() == None:
        print('  >>> test passed!')
```

```
test_1:
  >>> test passed!
test_2:
  >>> test passed!
test_3:
  >>> test passed!
test_4:
  >>> test passed!
test_5:
*TypeError: k must be a positive integer.
  >>> test passed!
test_6:
*ValueError: k must be a positive integer.
  >>> test passed!
test_7:
*InputError: input length less than 1.
  >>> test passed!
test_8:
*InputError: input is not a number list or numpy array.
  >>> test passed!
test_9:
*ValueError: k must be an integer greater than 1.
  >>> test passed!
```

In [561]:
```
# Please ignore this cell. This cell is for us to implement the tests
# to see if your code works properly.
```

Question 6 [#ComplexityAnalysis, #ComputationalCritique]

Analyze and compare the practical run times of regular merge sort (i.e., two-way merge sort), three-way merge sort, and the extended merge sort from (3) by producing a plot that illustrates how every running time and number of steps grows with input size. Make sure to:

1. define what each algorithm's complexity is
2. enumerate the explicit assumptions made to assess each run time of the algorithm's run time.
3. and compare your benchmarks with the theoretical result we have discussed in class.

Strategy of analysis

```
1. I will investigate running time both through step counting and computer r
untime tracking;
2. for step counting, I will reproduce codes for three target Sort functions
below, add a global counter to each sort, i.e
    - 'step_2wyMsrt' for twoWayMerge(A)
    - 'step_3wyMsrt' for threeWayMerge(A)
    - 'step_extMsrt' for extendedThreeWayMerge(A, k)
3. global step counters from auxiliary functions during sorting have been em
bedded in Qn 2 and Qn3, :
    - global Mrg2_step for merge_two(A1, A2)
    - global Mrg3_step for merge_three(A1, A2, A3)
    - global sele_step for selectionSort(A)
*will call for auxiliary steps to sort steps in 2 accordingly during sorting
4. for computer runtime tracking, implement clock from python time module, s
tart clock at 0, run the target algorithm, stop timer and calculate the diff
erent between start and end time.
5. will collect two lists of empirical run time data, steps_list and runTime
_list, for performance plotting.
```

Some assumptions for step counting

1. constant execution is counted (1 step) for 'if', 'raise, and assignment statements;
2. a n-size for loop is counted as n+1 step to account for variable initialization .
3. when an auxiliary function is called within sort function, the total number of steps in the auxiliary fuction is added right after the call statement
4. 'return' is not counted as a step

Definition of complexity

Adopting a pessimist's view, my choice of complexity for analysis here refers to

```
1) runtime complexity rather than space/memory complexity;
2) specifically the worst-case runtime complexity, or the asymptotic upper b
ound denoted by big-Oh of n,
```

i.e. How would the sorting algorithm behave in the worst-case scenario of a strictly descending array as input size grows large?

recap the definition of a big-Oh:

- for a given function g(n), we denote by O(g(n)) the set of functions:

```
O(g(n)) = {f(n): there exist positive constant c and n0 such that 0 <=
f(n) <= cg(n) for all n >= n0}
```

we will use this definition to analyze theoretical cmoplexity of the three targe sorting algorithms.

For clarity of illustration of step counting, I would like to reproduce codes of the target sorting algorithms, add step counters line-by-line with comments, and also remove all step-irrelevant comments to obey princple of parsimony for neater presentation.

In [562]:
```
global step_2wyMsrt  # a global step counter for two-way merge sort
step_2wyMsrt  = 0 # initialized outside of recursive loop

def twoWayMerge(A):

    global step_2wyMsrt
    global Mrg2_step
    Mrg2_step = 0 # reset auxiliary global counter to zero

    if isinstance(A,(list,np.ndarray)):
        step_2wyMsrt  += 1 # 1-step evaluation of if statement

        A = list(A)
        n = len(A)
        step_2wyMsrt += 2 # 2*1-step assignment

        if n < 1:
            step_2wyMsrt  += 1 # if statement
            step_2wyMsrt += 1 # count raise statement before exit
            raise Exception('input length less than 1') # exit takes 0 s
tep

        elif n==1:
            step_2wyMsrt  += 1 # efif statement
            return A

        else:
            m = n//2
            step_2wyMsrt  += 1  # 1 assignment

            A1 = twoWayMerge(A[:m])
            A2 = twoWayMerge(A[m:])
            # global step_2wyMsrt continue grows in recursion
            step_2wyMsrt  += 1*2 # exit recursion, two assignments to A
1, A2

            A = merge_two(A1, A2)
            step_2wyMsrt += Mrg2_step # add inner steps from merge_two()
            step_2wyMsrt += 1 # assignment to A

            return A
    else:
        step_2wyMsrt += 1 # else statement
        step_2wyMsrt += 1 # raise statement
        raise Exception('input is not a list or numpy array.')
```

In [563]:
```
global step_3wyMsrt  # a global step counter for three-way merge sort
step_3wyMsrt  = 0 # initialized outside of recursive loop

def threeWayMerge(A):

    global step_3wyMsrt
    global Mrg3_step
    Mrg3_step = 0 # reset auxiliary global counter to zero

    if isinstance(A,(list,np.ndarray)):
        step_3wyMsrt += 1 # if statement evaluation O(1)

        A = list(A)
        n = len(A)
        step_3wyMsrt += 2 # two assignment statements

        if n < 1:
            step_3wyMsrt += 1 # if statement
            step_3wyMsr += 1 # raise statement
            raise Exception('input length less than 1')

        elif n==1:
            step_3wyMsrt += 1 # if statement
            return A

        elif n==2:
            step_3wyMsrt += 1 # if statement
            A.sort()
            # python built-in TimSort follows O(nlgn)
            step_3wyMsrt += n*np.log2(n)

            return A

        else:
            m = n//3
            step_3wyMsrt += 1 # 1 assignment

            A1 = threeWayMerge(A[0:m])
            A2 = threeWayMerge(A[m:m*2])
            A3 = threeWayMerge(A[m*2:n])
            # global step_2wyMsrt continues to grow in recursion
            step_3wyMsrt += 3 # three assignments to A1, A2, and A3

            A = merge_three(A1, A2, A3)
            step_3wyMsrt += Mrg3_step # add steps from merge_three()

            return A

    else:
        step_3wyMsr += 1 #  'else' statement
        step_3wyMsr += 1 # raise statement
        raise Exception('input is not a list or numpy array.')
```

In [564]:
```
global step_extMsrt # a global step counter for three-way merge sort
step_extMsrt  = 0 # initialized outside of recursive loop

def extendedThreeWayMerge(A, k):

    global step_extMsrt
    global sele_step
    global Mrg3_step
    sele_step, Mrg3_step = 0,0 # reset auxiliary global counters to zero


    if not isinstance(k,int) or (k <= 0):
        step_extMsrt += 1 # if statement evaluation
        step_extMsrt += 1 # raise statement
        raise Exception ('k must be a positive integer.')

    if isinstance(A,(list,np.ndarray)):
        step_extMsrt += 1 # if statement

        A = list(A)
        n = len(A)
        step_extMsrt += 2 # two assignments

        if n < 1:
            step_extMsrt += 1 # if statement
            step_extMsrt += 1 # raise statement
            raise Exception('input length less than 1')

        elif n==1:
            step_extMsrt += 1 # if statement
            return A

        elif n==2:
            step_extMsrt += 1 # if statement
            A.sort()
            # python built-in TimSort follows O(nlgn)
            step_extMsrt += n*np.log2(n) # add steps from built-in TimSo
rt

            return A

        elif n <= k:
            step_extMsrt += 1 # elif statement
            step_extMsrt += sele_step # add steps from selection sort be
fore return
            return selectionSort(A)

        else:
            step_extMsrt += 1 # else statement evaluation
            m = n//3
            step_extMsrt += 1 # 1 assignment

            A1 = extendedThreeWayMerge(A[0:m], k)
            A2 = extendedThreeWayMerge(A[m:m*2], k)
            A3 = extendedThreeWayMerge(A[m*2:n], k)
```

```
            # global step_extMsrt continues to grow in recursion
            step_extMsrt += 3 # 3 assignments to A1, A2 and A3

            A = merge_three(A1, A2, A3)
            step_extMsrt += Mrg3_step # add steps from merge_three()

        return A

    else:
        step_extMsrt += 1 # else statement
        step_extMsrt += 1 # raise statement
        raise Exception('input is not a list or numpy array.')
```

In [565]:
```
def stepList(function_name, step_counter, n, k, A_base):
    '''
    this is a time complexity data generator for number-of-steps analysi
s
    assume worst-case performance: A in strictly descending order

    input
        function_name = twoWayMerge/threeWayMerge
                           /extendedThreeWayMerge
            * pass funtions as args
        step_counter = step_2wyMsrt/step_3wyMsrt/step_extMsrt
        n = number of data points
        k = recursion base length, relevant only in extended merge sort
        A_base = base of growth for length of A
    output
        x, y lists of runtime data w.r.t input function for plotting
    '''

    global step_2wyMsrt, step_3wyMsrt, step_extMsrt

    order_list = [] # input sizes (order of growth)
    steps_list = [] # initiate empty data list

    for i in range(1,n+1): # order of size growth from 10^1 to 10^n

        #A = list(range(1000+10**(i+1), 0, -1)) # descending order
        A = list(range(A_base**i, 0, -1))
        n = len(A)
        order_list.append(n) # append size n

        # reset all global counters to zero
        step_2wyMsrt, step_3wyMsrt, step_extMsrt = 0,0,0

        # choose the right method to count steps
        if function_name == twoWayMerge:
            #print('is twoWayMerge!')
            #print('len(A)', len(A))
            function_name(A)
            steps_list.append(step_2wyMsrt)

        elif function_name == threeWayMerge:
            #print('is threeWayMerge!')
            #print('len(A)', len(A))
            function_name(A)
            steps_list.append(step_3wyMsrt)

        else:
            #print('is extended3Way!')
            #print('len(A)', len(A))
            function_name(A,k)
            steps_list.append(step_extMsrt)

    return [order_list, steps_list]
```

```
def runTimeList(function_name, n, k, A_base):
    '''
    this is a time complexity data generator for computer runtime analys
is (s)
    assume worst-case performance: A in strictly descending order

    input
        function_name = twoWayMerge/threeWayMerge
                            /extendedThreeWayMerge
            * pass funtions as args
        n = number of data points
        k = recursion base length, relevant only in extended merge sort
        A_base = base of growth for length of A
     output
        x, y lists of runtime data w.r.t input function for plotting
    '''
    order_list = [] # input sizes (order of growth)
    runTime_list = [] # initiate empty data list

    for i in range(1,n+1): # order of growth from 10^1 to 10^n
        A = list(range(A_base**i, 0, -1))  #descending order
        n = len(A)
        order_list.append(n) # append size n

        start_time = time.clock()     # reset timer to zero

        # choose the right method to count steps
        if function_name == extendedThreeWayMerge:
            function_name(A,k)
        else:
            function_name(A)

        runTime_list.append(time.clock()-start_time)

    return [order_list, runTime_list]
```

In [566]:
```
def plotSteps(lists_stepList, algo_names):
    '''
    plot a single figure with multiple lines of step growth
        for different target algorithms of analysis
    input:
        lists_stepList is a list of
            stepList = [order_list, steps_list]
        algo_names = a list of string name of target algorithms,
            must have same length and order as lists_stepList
    output:
                    must be in the same order as lists_stepList
        a single, formatted pyplot
    '''
    # check input validity
    if len(lists_stepList) != len(algo_names):
        raise Exception("input lists don't have equal lengths.")

    n = len(lists_stepList)

    # plot individual runtime growth as size increases
    for i in range(n):
        x = lists_stepList[i][0] # get order_list of ith stepList
        y = lists_stepList[i][1] # get steps_list of ith stepList
        plt.plot(x,y, linestyle='-', label= algo_names[i])

    # plot formatting
    plt.title( "Worst-case complexity comparison (in steps)")
    plt.xlabel('input size (n)')
    plt.ylabel('number of steps')
    plt.legend()
    plt.show()


def plotRunTime(lists_runTimeList, algo_names):
    '''
        plot a single figure with multiple lines of runtime growth
        for different target algorithms of analysis
    input:
        lists_runTimeList is a list of
            runTimeList = [order_list, runTime_list]
        algo_names = a list of string name of target algorithms,
            must have same length and order as lists_runTimeList
    output:
        a single, formatted pyplot
    '''

        # check input validity
    if len(lists_runTimeList) != len(algo_names):
        raise Exception("input lists don't have equal lengths.")

    n = len(lists_runTimeList)

    # plot individual runtime growth as size increases
    for i in range(n):
        x = lists_runTimeList[i][0] # get order_list of ith runTimeList
```

```
        y = lists_runTimeList[i][1] # get runTime_list of ith runTimeLis
t
        plt.plot(x,y, linestyle='-', label= algo_names[i])

    # plot formatting
    plt.title( "Worst-case running time comparison")
    plt.xlabel('input size (n)')
    plt.ylabel('running time (s)')
    plt.legend()
    plt.show()
```

In [567]:
```
# define key variables
function_names = [twoWayMerge, threeWayMerge, extendedThreeWayMerge]
algo_names = ['2way Merge', '3way Merge', 'Merge_Selection']
step_counters = [step_2wyMsrt, step_3wyMsrt, step_extMsrt]
# labellers for twoWayMerge(A), threeWayMerge(A) and extendedThreeWayMer
ge(A, k)


step_2wyMsrt, step_3wyMsrt, step_extMsrt = 0,0,0 # reset step counters
n = 5 # order of growth (number of data points calculated)
```

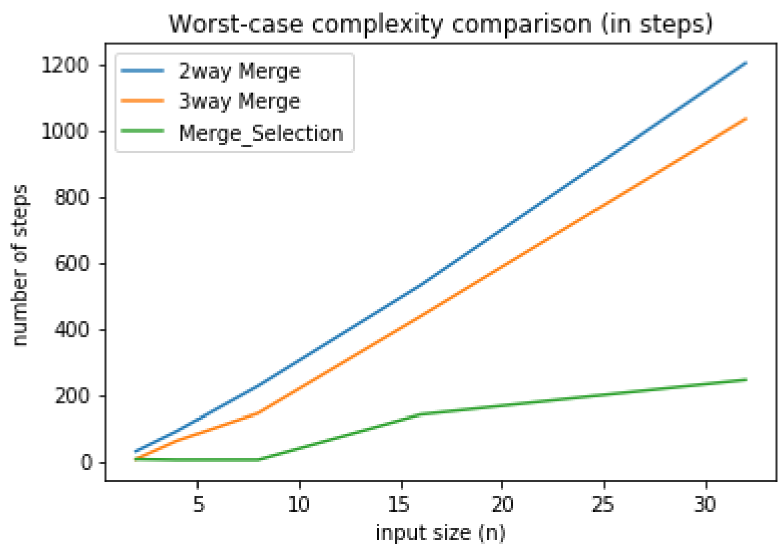
Start of complexity analysis

Experimental round:

In [568]:
```
# At small input size <br>
A_base = 2
k = 14 # intuition gained from Qn 4, 10 < k < 15 for most efficiency
```

In [569]:

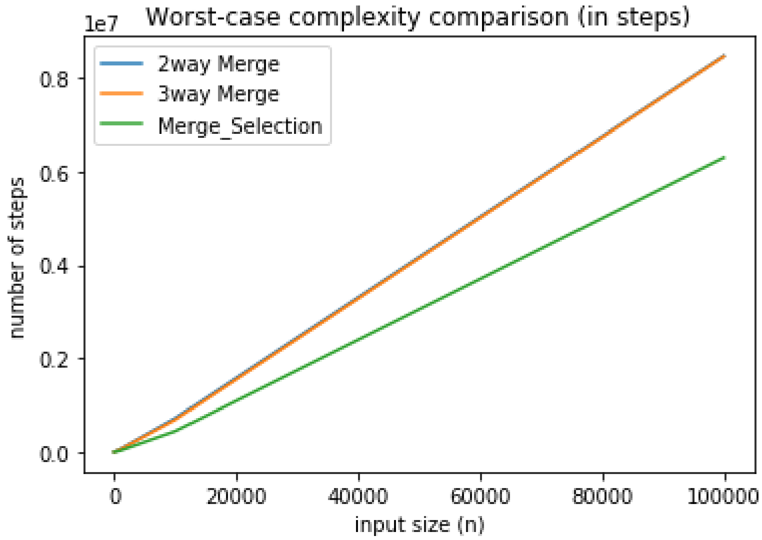
```
# plot steps of growth at small input size [2, 1024]
# generate lists_stepList
lists_stepList = []
for i in range(len(algo_names)):
    function_name = function_names[i]
    step_counter = int(step_counters[i]) # cast float to int
    x_y = stepList(function_name, step_counter, n,k, A_base )
    lists_stepList.append(x_y)

# plot growth of steps
plotSteps(lists_stepList, algo_names)
```



Comment 1:

We can see that all three merge sorts exhibit tiny gain of exponential at the very start of input size n < 10, however, this effect is quickly dominated by a linear growth for both 2-way and 3-way merge sort. Among three,the hybrid merge-selection sort performs the best at small values, exhibiting a mixture of linear and quadratic growth.

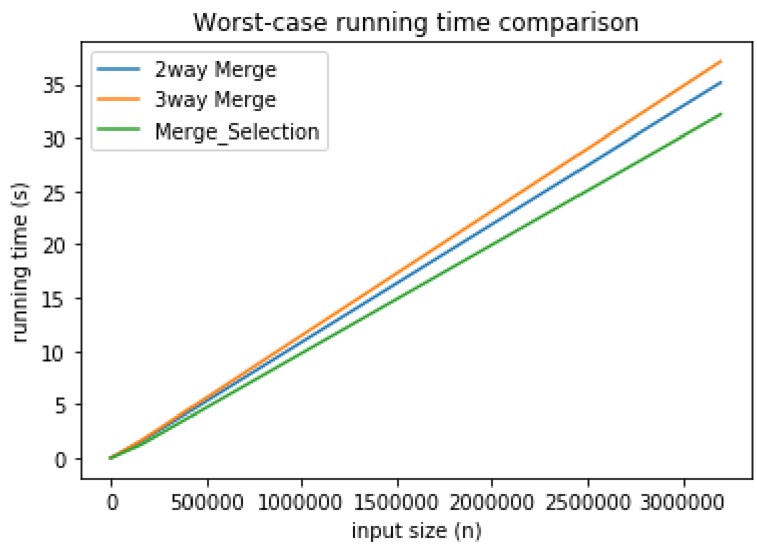
In [570]:
```
# plot steps of growth at large n [may run for 30s]
A_base = 10

# generate lists_stepList
lists_stepList = []
for i in range(len(algo_names)):
    function_name = function_names[i]
    step_counter = int(step_counters[i]) # cast float to int
    x_y = stepList(function_name, step_counter, n,k, A_base )
    lists_stepList.append(x_y)

# plot growth of steps
plotSteps(lists_stepList, algo_names)
```



Comment 2:

At very large n size (up to 10^5), all three sorts look highly linear. both 2-way and 3-way merge sorts have equal perofrmance, and the hybrid sort with k = 14 constantly outperforms both types of pure merge sorts.

In [571]:
```
# plot runtime growth at even larger n [may run for up to 1 min]

A_base = 20

# generate lists_runTimeList
lists_runTimeList = []
for i in range(len(algo_names)):
    function_name = function_names[i]
    step_counter = int(step_counters[i]) # cast float to int
    x_y = runTimeList(function_name, n, k, A_base)
    lists_runTimeList.append(x_y)

# plot growth of steps
plotRunTime(lists_runTimeList, algo_names)
```



Comment 3:

At even larger n size (up to 3x10^6), linearity is preserved for all three sorts.
A similar conclusion from Comment 2 can be drawn: pure merge sorts underperforms merge-selection hybrid sort.
(*runtime growth plot is chosen over step growth plot to reduce computer execution time, the block above will run for 1~2 min)

Discussion

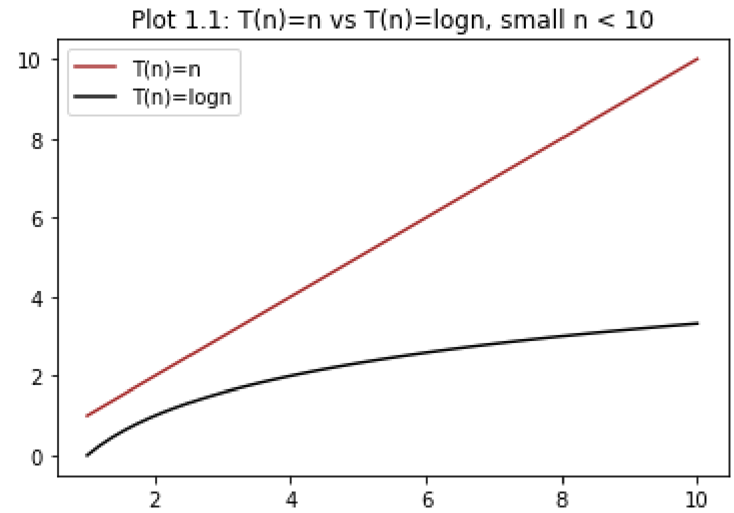
The above experimental results make analytically sense:

- This upper bound is calculated by solving the recursion tree of base: in the worst case, every element is compared at leaves (base level, such that the width of base of the recursion tree must be n, and the height of tree is logn, thus, the 'area' of tree, i.e. the total step would be $n * logn = nlogn$
- In the worst case of a strictly descending array, a selection sort have to iterate a for loop of size n, and within each iteration , compare n-i times (i is the past number of iterations), thus, total comparison is $T(n) = n!$, which is asymptotically $O(n^2)$
- For a hybrid of selection and merge sort, the integrated complexity would depends on the dominant part of $O(n^2)$ and $O(nlogn)$. Given our intuition for a 'best' k value between 10 to 15, I chose to set k=14 here to exhibit a most efficient version of merge-selection sort. This means that selection sort works on a scale of array length less than 14. However, as n grows to be very large, the quadratic effect of selection sort operation deminishes to be insignificant, and the merge-selection sort is expected to behave like a merge sort at $O(nlogn)$, albeit at a different slope (different constant c for g(n)).

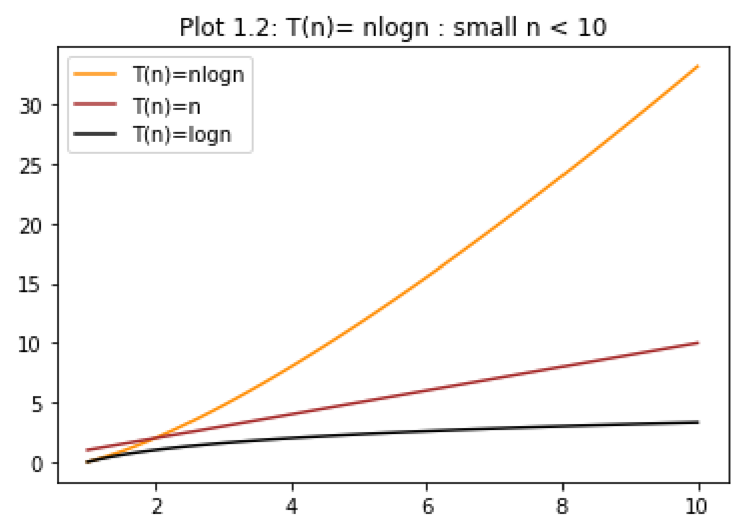
Therefore, the worst-case running time complexity for all three sorts at very large n would be $O(nlogn)$

Analytical round:

In [572]:
```
# analytical plot  T(n) = nlog(n)

x = np.linspace(1,10)
#plt.plot(x, x*np.log2(x), '-',color ='darkorange', label='T(n)=nlogn')
plt.plot(x, x, '-',color ='brown', label='T(n)=n' )
plt.plot(x, np.log2(x), '-',color ='black', label='T(n)=logn')
plt.title('Plot 1.1: T(n)=n vs T(n)=logn, small n < 10')
plt.legend()
plt.show()
```



In [573]:
```
x = np.linspace(1,10)
plt.plot(x, x*np.log2(x), '-',color ='darkorange', label='T(n)=nlogn')
plt.plot(x, x, '-',color ='brown', label='T(n)=n' )
plt.plot(x, np.log2(x), '-',color ='black', label='T(n)=logn')
plt.title('Plot 1.2: T(n)= nlogn : small n < 10')
plt.legend()
plt.show()
```

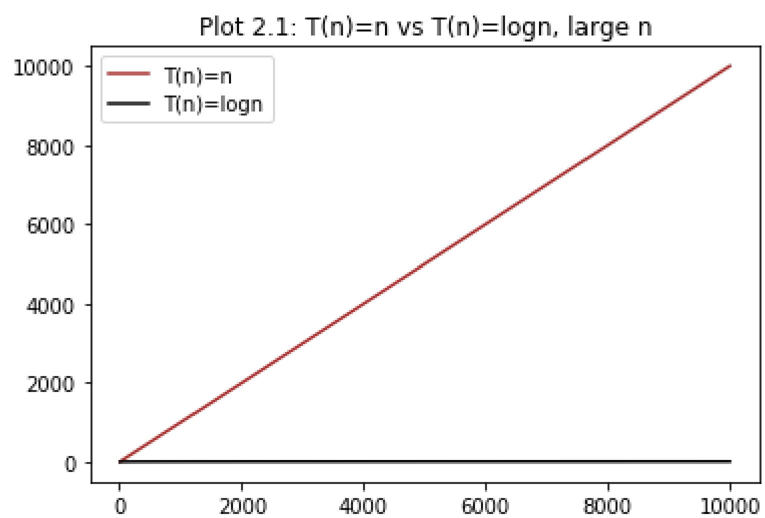


Comment 4:

At very small input size, there is no order of magnitude difference between $T(n)$ and $T(n) = logn$, although n is a strict upper bound of logn, as shown in plot1.1

> This means that we would reasonably expect $T(n) = nlogn$ to exhibit some positive logarithmic growth on top of a linear growth for very small input size, so the curve of $T(n) = nlogn$ starts to increases at tiny increasing rate from the very beginning. However, the logarithmic scaling effect quickly diminishes as n passes 5, and dominated by a linear growth a n becomes relatively bigger, as illustrated through the yellow line in plot1.2,
> This property was also confirmed experimentally in the step and runtime plots above.

In [574]:
```
# analytical plot  T(n) = nlog(n)

x = np.linspace(10,10000, 100)
#plt.plot(x, x*np.log2(x), '-',color ='darkorange', label='T(n)=nlogn')
plt.plot(x, x, '-',color ='brown', label='T(n)=n' )
plt.plot(x, np.log2(x), '-',color ='black', label='T(n)=logn')
plt.title('Plot 2.1: T(n)=n vs T(n)=logn, large n')
plt.legend()
plt.show()
```

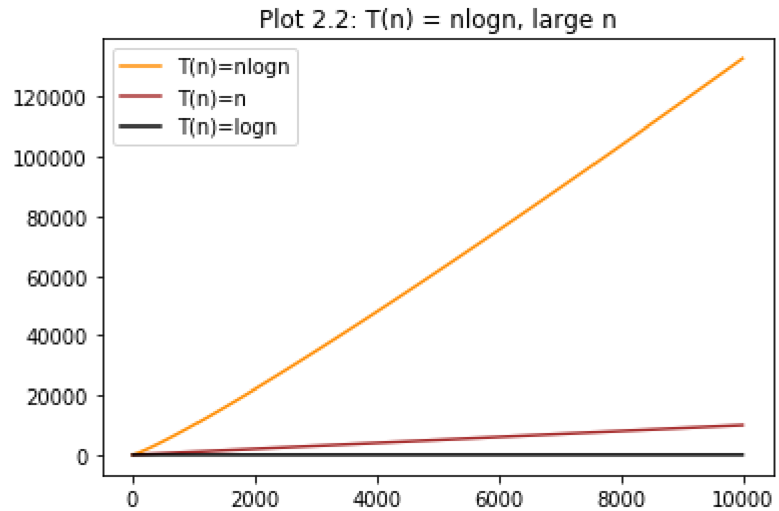


Comment 5:

As we can see from Plot 2.1, the magnitude of $T(n) = n$, brown line, is orders of magintude larger than that of $T(n) = logn$, black line, i.e. the rato between two is almost equal to n itself. This renders the multipilication effect of a $logn$ onto n insignificant, and the long-run behavior of $T(n) = nlogn$ would approximate a linear growth of $T(n) = cn$.

The dominant effect of linear growth is illustrated in Plot 2.2 when $T(n) = nlogn$ is added back to the graphs:

In [575]:
```
# analytical plot  T(n) = nlog(n)
x = np.linspace(10,10000, 100)
plt.plot(x, x*np.log2(x), '-',color ='darkorange', label='T(n)=nlogn')
plt.plot(x, x, '-',color ='brown', label='T(n)=n' )
plt.plot(x, np.log2(x), '-',color ='black', label='T(n)=logn')
plt.title('Plot 2.2: T(n) = nlogn, large n')
plt.legend()
plt.show()
```



Conclusion:

both of my experimental plots with step and computer runtime confirms the analytical results obtained above.

long-run behavior for each of the three sorts is $nlogn$, which approximates a linear growth of $T(n) = cn$ as input size tends to infinity. so we may conclude that the worst-case performance of all three sorts are $O(nlogn)$

Question 7. [#ComplexityAnalysis, #ComputationalCritique]

Analyze and compare the practical run times of regular merge sort (i.e., two-way merge sort), Bucket sort and recursive sort from (5) by producing a plot that illustrates how each running time grows with input size. Make sure to:

1. define what each algorithm's complexity is
2. enumerate the explicit assumptions made to assess each run time of the algorithm's run time.
3. and compare your benchmarks with the theoretical result we have discussed in class.

Strategy of analysis & Assumptions

overall strategy of complexity analysis stays the same as Qn 6, will reuse codes from Qn 6 to generate runtime data estimates and plot complexity graphs.

same assumptions from Qn 6 apply here for step calculation, additional assumptions include:

1. python min( ),max( ) evaluates every element once, O(n);
2. python arithmetic operations, such as (mx-mn)/k take constant time O(1);
3. math.ceil() takes constant time
4. auxiliary step counters include: buktNum, step

Define complexity

Similar to Qn 6, I am most interested in the worst case performance among three sorts, so complexity refers to analysis of runtime asymptotic upper bound.

1. will reuse code and step counter for regular merge sort
2. To better illustrate my thought process, I would like to reproduce codes of the target sorting algorithms, add step counters line-by-line with comments, and remove all other step-irrelevant comments to obey princple of parsimony.

In [576]:
```
global bukt_step
bukt_step = 0  # initiate at zero outside of bucket sort method
           # to avoid over counting

def bucketSort(A, k):

    # add a global bucket step counter
    global bukt_step

    # reset auxiliary counters
    global buktNum_step, sele_step
    buktNum_step, sele_step = 0,0

    if not isinstance(k,int) or (k <= 0):
        bukt_step += 1 # 'if' statement evaluation
        bukt_step += 1 # 'raise' statement evaluation before exit
        raise Exception ('k must be a positive integer.')

    if isinstance(A,(list,np.ndarray)):
        bukt_step += 1 # 'if' statement evaluation

        A = list(A)
        bukt_step += 1 # assignment evaluation

        n = len(A)
        if n < 1:
            bukt_step += 1 # 'if' statement evaluation
            bukt_step += 1 # 'raise' statement evaluation before exit
            raise Exception('input length less than 1')

        mn = min(A)
        mx = max(A)
        bukt_step += n*2 # python min(),max() evaluates every element on
ce, O(n)

        sz = math.ceil((mx - mn)/k)
        bukt_step += 4 # 1 arithmetic operation + 3 assignments

        Buckets = [ [] for bkt in range(k) ]
        bukt_step += k+1  # k+1 times for loop

        bukt_step += 1  # for loop initialization
        for i in range (n):
            b = GetBucketNum(A[i], mn, mx, sz, k)-1
            bukt_step += buktNum_step # add steps from bucketNum
            #print('buktNum_step =',buktNum_step)
            bukt_step += 1 # 1 assignment to b
            Buckets[b].append(A[i])
            bukt_step += 1 # list.append takes constant time

        sorted_A = []
        bukt_step += 1 # 1 assignment

        bukt_step += 1 # for loop initialization
        for s in range (k):
```

```
            bukt_step += 1 # for statement evaluation
            #print('sele_step =',sele_step)
            sorted_A += selectionSort(Buckets[s])
            bukt_step += sele_step + 1  # add steps in selection sort +
 1 assignment

        return sorted_A

    else:
        bukt_step += 2 # else statement + raise statement
        raise Exception('A must be either a list or a numpy array.')
```

In [578]:
```
global Rcurbkt_step
Rcurbkt_step = 0  # initiate at zero outside of bucket sort method
             # to avoid over counting

def extendedBucketSort(A, k):

    # add a global recurisve-bucket step counter
    global Rcurbkt_step

    # reset auxiliary counters
    global buktNum_step, sele_step
    buktNum_step, sele_step = 0,0

    if not isinstance(k,int) or (k <= 1):
        Rcurbkt_step += 2 # if statement + raise statement
        raise Exception ('k must be a positive integer greater than 1.')

    if isinstance(A,(list,np.ndarray)):
        Rcurbkt_step += 1 # if statement

        A = list(A)
        Rcurbkt_step += 1 # 1 assignment

        if len(A) < 1:
            Rcurbkt_step += 2 # if + raise statements
            raise Exception ('input array length less than 1.')

        mn = min(A)
        mx = max(A)
        # python min(),max() evaluates every element once, O(n)
        Rcurbkt_step += n*2

        sz = math.ceil((mx - mn)/k)
        Rcurbkt_step += 4 # 1 arithmetic operation + 3 assignments above

        if sz <= k:
            Rcurbkt_step += 1 # if statement
            Rcurbkt_step += sele_step # add steps from selection sort be
fore exit
            return selectionSort(A)

        else:
            Rcurbkt_step += 1  # if statement

            Buckets = [ [] for bkt in range(k) ]
            Rcurbkt_step += k+1  # k+1 times for loop

            Rcurbkt_step += 1  # for loop initialization
            for i in range (len(A)):
                Rcurbkt_step += 1  # for loop evaluation
                b = GetBucketNum(A[i], mn, mx, sz, k)-1
                Rcurbkt_step += buktNum_step  # add steps from getBucktN
um()
                Rcurbkt_step += 1 # assignment to b
```

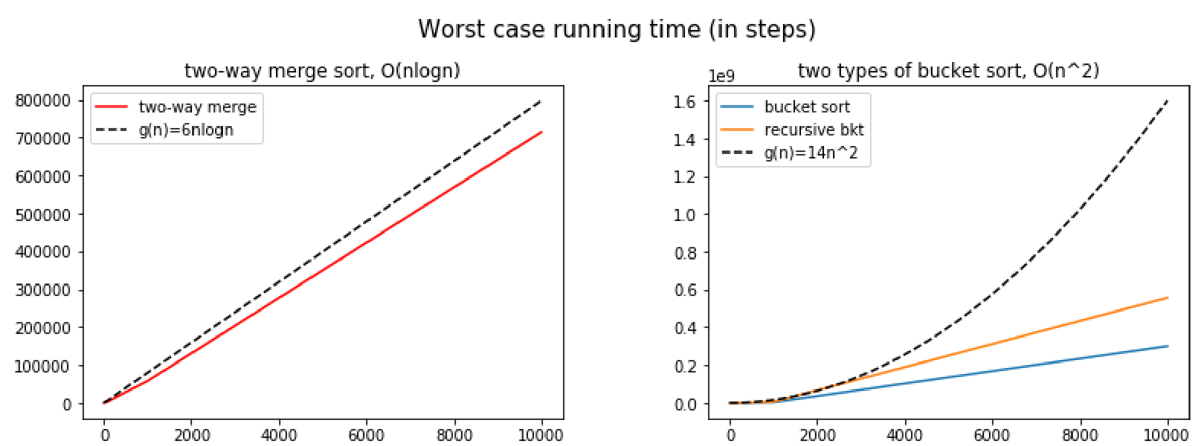
```
                Buckets[b].append(A[i])
                Rcurbkt_step += 1 # list.append takes 1 step

            sorted_A = []
            Rcurbkt_step += 1 # constant time, 1 step

            Rcurbkt_step += 1  # for loop initialization
            for s in range (k):
                Rcurbkt_step += 1  # for loop evaluation
                A = Buckets[s]
                Rcurbkt_step += 1  # 1 assignment
                Bucket_s = extendedBucketSort(A, k)
                # Rcurbkt_step will continue to grow within recursive ca
lls
                Rcurbkt_step += 1  # exit recursion, add 1 assignment

                sorted_A += Bucket_s
                Rcurbkt_step += 1 # 1 assignment

            return sorted_A

    else:
        Rcurbkt_step += 2 # else statement + raise statement
        raise Exception('A must be either a list or a numpy array.')
```

In [579]:

```
# initialize key variables
function_names = [twoWayMerge, bucketSort, extendedBucketSort]
step_counters = [step_2wyMsrt, bukt_step, Rcurbkt_step ]
algo_names = ['2-way merge', 'bucket sort', 'recursive bucket'] # labell
er
```

In [580]:
```
## plot individual algorithm (worst case)

# prepare step data
g = 5 # number of data points
A_base = 10 # base of growth of input size

# choice of k (number of buckets)
# assuming same number of buckets used for both bucket and extended buck
et sort
# start with the extreme case of binary buckets
k = 2

order_list = []

step_mergeSrt = []
step_bucketSrt = []
step_rcurBucketSrt = []


for i in range (g):
    A = list(range(A_base**i,0, -1))
    n = len(A) # input size
    order_list.append(n)

    step_2wyMsrt = 0
    twoWayMerge(A)
    step_mergeSrt.append(step_2wyMsrt)

    bukt_step =  0
    bucketSort(A, k)
    step_bucketSrt.append(bukt_step)

    Rcurbkt_step = 0
    extendedBucketSort(A, k)
    step_rcurBucketSrt.append(Rcurbkt_step)
```

In [581]:
```
# plot 1: Worst case running time (in steps)
fig, (ax1, ax2) = plt.subplots(1,2,figsize=(13,4))
fig.suptitle('Worst case running time (in steps)', va='bottom', fontsize
='15' )

x = np.linspace(1, 10000, 100)

ax1.plot(order_list, step_mergeSrt, color = 'red', label= 'two-way merg
e')
ax1.set_title('two-way merge sort, O(nlogn)')
ax1.plot(x, 6*(x*np.log2(n)), '--',color='black', label='g(n)=6nlogn')
ax1.legend()

ax2.plot(order_list, step_bucketSrt, label= 'bucket sort')
ax2.plot(order_list, step_rcurBucketSrt, label= 'recursive bkt')
ax2.plot(x, 16*x**2, '--',color='black', label='g(n)=14n^2')
ax2.set_title('two types of bucket sort, O(n^2)')
plt.subplots_adjust( wspace = 0.3, hspace = 0.5 )
ax2.legend()

plt.show()
```

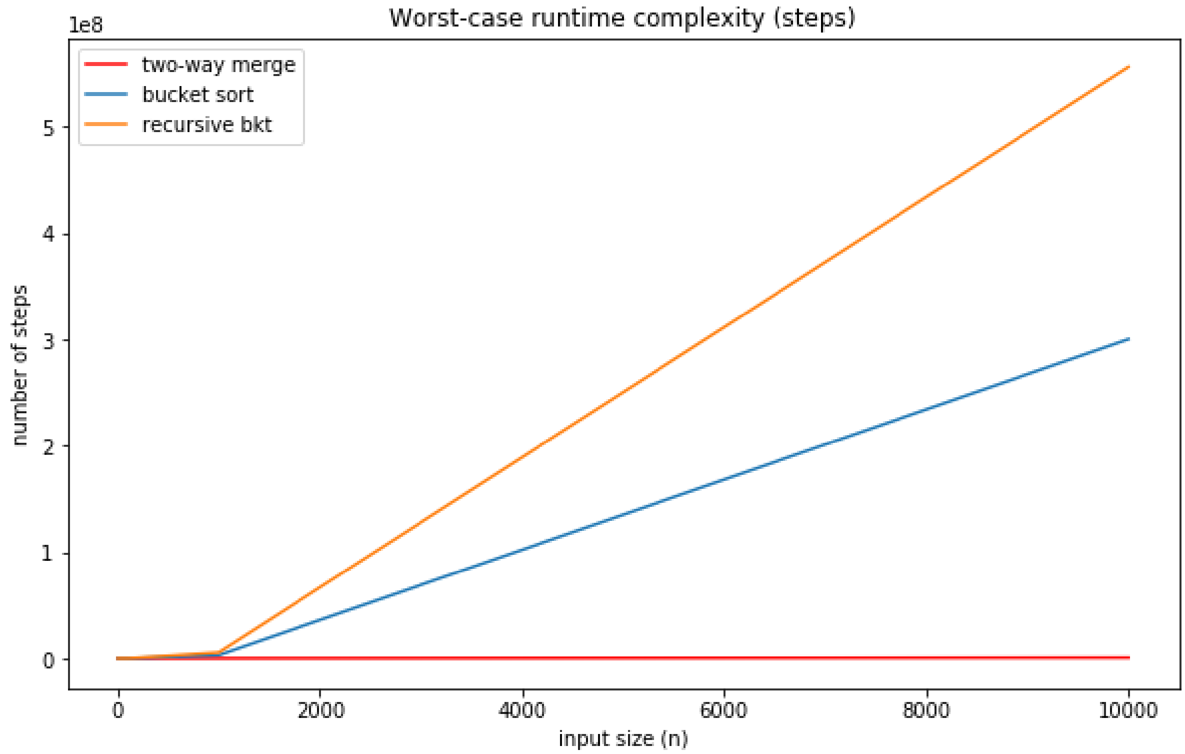


Analysis 1:

The above 2-panel graph compares runtime complexity behavior between regular merge sort and the bucket sort family. Due to limited graphing space and range of n values, all three growth plots look linear. However, theoretically, we know that:

```
1. regular merge sort has a worst-case runtime complexity of $T(n) = O(nlog
n)$, which is illustrated in the left panel above: the black dashed line rep
rensents an instance of g(n) = cnlogn, c a constant, such that 0 <= T(n) <=
 6nlogn for some small n0;

2. in the worst case, regular bucket sort with k buckets would call for sele
ction sort k times, and each input subarray to selection sort is in a strict
ly descending array, so the asymptotic runtime for selection sort is O(n^2),
rendering the worst-caes runtime of bucket sort to be kn^2 ~ O(n^2). This is
illustrated through the dashed quadratic line in the right panel, which is s
trictly above bucket sort (blue line) after some small positive n;

3. the recursive bucket sort recursively break down buckets until size < k
 (length of base bucket), end up having log_base_k(n) number of base bucket
s, each bucket strictly descending, so there would be log_base_k(n)*O(n^2) c
alls for selectionSort(), depending on value of k, the recursive bucket sort
may or maynot outperform regular bucket sort, however it's runtime is still
 bounded by O(n^2) from above.

4. Here, the graph illustrates a caes of binary buckets, which means that, f
or same input size, log_base_2(n) is the largest compare to other log_base_k
(n), this is probably why recursive bucket appears suboptimal compared to re
gular bucket sort.
```

In [582]:
```
# plot 2: combined plot
plt.figure(figsize=(10,6))
plt.plot(order_list, step_mergeSrt, color = 'red', label= 'two-way merg
e')
plt.plot(order_list, step_bucketSrt, label= 'bucket sort')
plt.plot(order_list, step_rcurBucketSrt, label= 'recursive bkt')

plt.xlabel('input size (n)')
plt.ylabel('number of steps')
plt.title('Worst-case runtime complexity (steps)')
plt.legend()

plt.show()
```



Analysis 2:

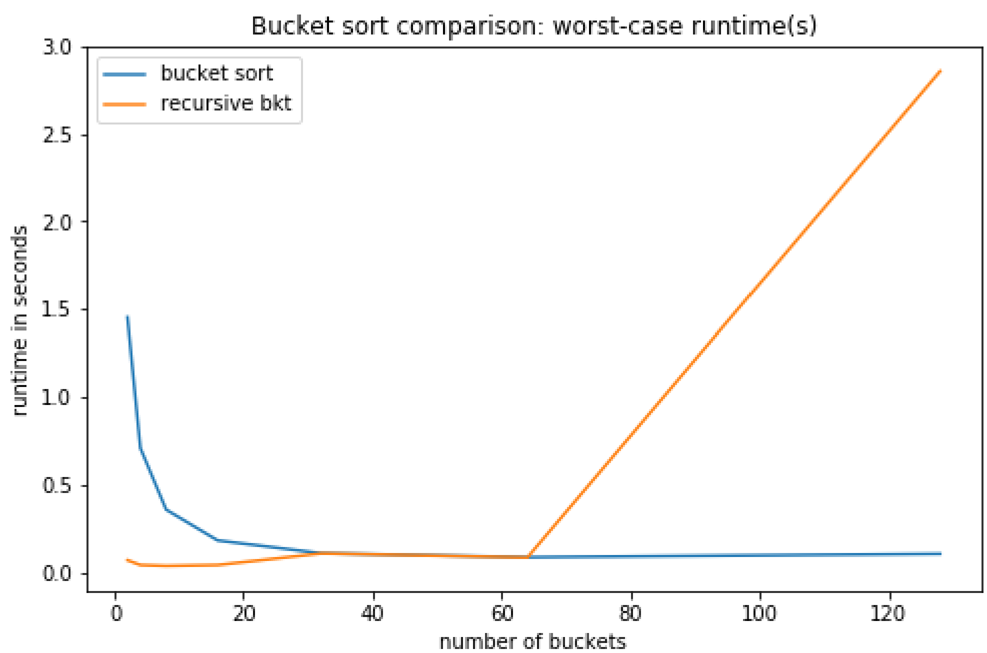
Combine two panels from plot 1 together we get a direct comparison of runtime steps among all three steps at large n (from n = 2000 up to 10^4). Previously when individual algorithm is plotted, they all exhibit linear-like growht pattern, however at different order of magnitude. When compiled together, the runtime growth rate for merge sort (max at 8*10^4) is orders of magnitude lower than bucket sort, and the recursive bucket sort doubles runtime of recursive bucket sort.

Next, we are interested in finding out, in the worst case, the critical k value that makes bucket sort and recursive bucket sort equal at large input size n > 2000.

In [583]:
```
A = list(range(5000, 0, -1)) # fix input array
```

In [584]:
```
# examine using runtime
k_list = []
time_mergeSrt = []
time_bucketSrt = []
time_rcurBucketSrt = []

for i in range (1, 8):
    k = 2**i
    k_list.append(k)

    start_time = time.clock()
    twoWayMerge(A)
    time_mergeSrt.append(time.clock()-start_time)

    start_time = time.clock()
    bucketSort(A, k)
    time_bucketSrt.append(time.clock()-start_time)

    start_time = time.clock()
    extendedBucketSort(A, k)
    time_rcurBucketSrt.append(time.clock()-start_time)
```

In [585]:
```
plt.figure(figsize=(8,5))
plt.title('Bucket sort comparison: worst-case runtime(s)', fontsize='12'
)
plt.plot(k_list, time_bucketSrt, label= 'bucket sort')

plt.plot(k_list, time_rcurBucketSrt, label= 'recursive bkt')
plt.xlabel('number of buckets')
plt.ylabel('runtime in seconds')
plt.legend()
plt.show()
```



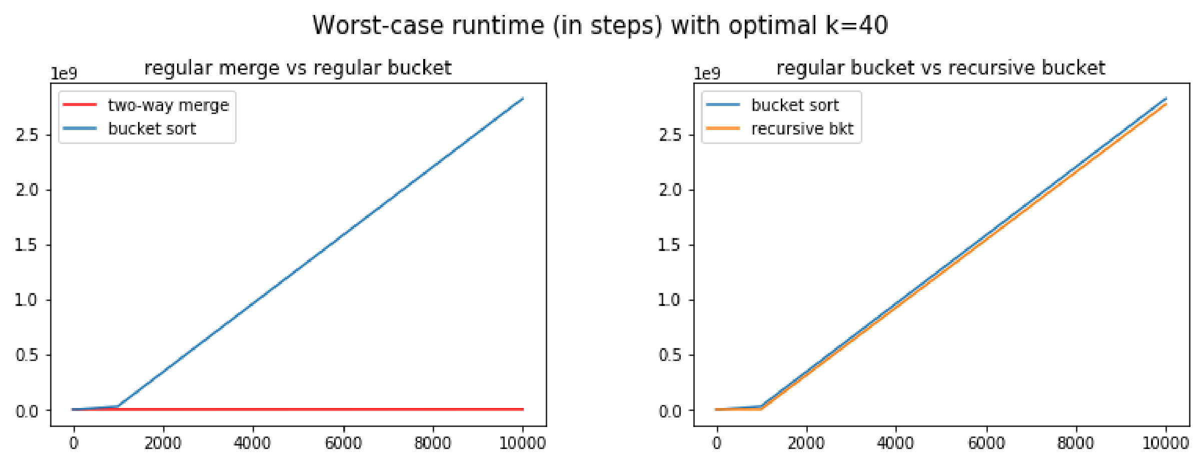
Analysis 3:

According to runtime plot above, as the number of buckets grow, pure bucket sort performance at lower time, while the runtime for recursive sort surges up irreversibly after a flat start. due to the opposing direction of growth between two sorts, we are confident that the 'best' k value shall locate in the range between 35 to 50, during which the two lines cross.

Let's pick k = 40 as the estimated best number of buckets for most efficient, and replot Plot 2 (three algo comparison) at large n:

In [586]:
```
# recollect runtime data given k = 40
# g = 5 (number of growth data), A_base = 10
k = 40
order_list = []

step_mergeSrt = []
step_bucketSrt = []
step_rcurBucketSrt = []


for i in range (g):
    A = list(range(A_base**i,0, -1))
    n = len(A) # input size
    order_list.append(n)

    step_2wyMsrt = 0
    twoWayMerge(A)
    step_mergeSrt.append(step_2wyMsrt)

    bukt_step =  0
    bucketSort(A, k)
    step_bucketSrt.append(bukt_step)

    Rcurbkt_step = 0
    extendedBucketSort(A, k)
    step_rcurBucketSrt.append(Rcurbkt_step)
```

In [587]:
```
fig, (ax1, ax2) = plt.subplots(1,2,figsize=(13,4))
fig.suptitle('Worst-case runtime (in steps) with optimal k=40', va='bott
om', fontsize='15' )

ax1.plot(order_list, step_mergeSrt, color = 'red', label= 'two-way merg
e')
ax1.plot(order_list, step_bucketSrt, label= 'bucket sort')
ax1.set_title('regular merge vs regular bucket')
ax1.legend()

ax2.plot(order_list, step_bucketSrt, label= 'bucket sort')
ax2.plot(order_list, step_rcurBucketSrt, label= 'recursive bkt')
ax2.set_title('regular bucket vs recursive bucket')
ax2.legend()

plt.subplots_adjust( wspace = 0.3, hspace = 0.5 )
plt.show()
```



Conclusion:

From the panel above, we may conclude that, in the worst-case scenario of a strictly descending input array, and given a optimal bucket number for most efficient bucket sort performance (right panel), the bucket sort family (regular and recursive) fails to beat a pure recursive-base sorting algorithm, the regular merge sort,as it requires a runtime at order of maginitude greater than that of merge sort for large input size, as illustrated in the left panel.

[Optional challenge] Question 8 (#SortingAlgorithm and/or #ComputationalCritique)

Implement k-way merge sort, where the user specifies k. Develop and run experiments to support a hypothesis about the “best” value of k.

In [588]:
```
# wish I had time to complete this... will try taking the callenge in my next assignment.
```

Appendix

HCs used :

algorithm:
to turn a black box of a particular python function into a 'white' one, I creatively implemented indicative 'print()' plug-ins between codelines (show in the Case Study section of Qn 3, this way I could understand the machine's step-by-step solution by reading the printed manuscript, and assess if the method I have developed so far have satisfied all constraints to produce a desired outcome.

designthinking:
I applied strategy of design thinking throughout my developing process by iterative feedback generation and incremental improvement. e.g. my first round of test-code implementation was simply adding code blocks of 3-5 line test conditions, each with lengthy system output, what's worse was that the program stops running when I tested for a successful raise of Exception. From here I went on to search python libaries for better test code packages, and redesign my test codes into an object of class test_case.

organization:
I intentionally organized my assignment in a prose fashion, especially in Q6,7, such that I could clearly illustrate my thought process in a reader-friendly manner to facilitate assessment; also I could insert markdowns, with clear sub-section titles, whenever I would like to comment on the code blocks above, saving myself time of scrolling through entire assignment to find a code block of interest.