Lab 2 : Sorting Algorithms

Steven Robles *

University of Texas, El Paso, TX

February 26, 2019

This paper aims to demonstrate and compare four separate sorting algorithms by tasking each algorithm to return the medium of a given list with length n. The four types of sorting algorithms are: Bubble Sort, Merge Sort, Quick Sort, and a modified Quick Sort which only sorts the region where the median is known to reside. Each of the sorting algorithms are given the same random generated linked list of length n. To compare the performance of each sorting algorithm, a counter is set to track the number of comparisons done by each algorithm as well as timing their execution. In the outcome, the modified Quick Sort algorithm had the best standing results in terms of time performance and the least amount of comparisons made.

;

Contents

I. Introduction

Within the computer science field, having the best time executions for each program is always the top priority after creating a successful program. This lab focus on comparing four different sorting algorithms through a series of test to determine which algorithm would be the most efficient to implement. The four sorting algorithms are: Bubble Sort, Merge Sort, Quick Sort, and a modified Quick Sort. Instead of testing each sorting algorithm with a native Python list, a single linked list is utilized.

To demonstrate and compare the time efficiency's from within the program, a counter is set set within each algorithm. The counter will track the number of comparisons each function makes while executing. The returned value from the counter will signify the total number of comparisons made and will be plotted as part of the results. The time execution of the algorithm as a whole will also be accounted for and plotted in the results as a graph. Both of these graphs will represent the efficiency's of each algorithm in a physical manner for comparison.

*ID Number: 80678755

II. Design

The sorting algorithms for the labs was modeled after existing algorithms. The only major challenge is constructing the algorithms to be able to handle a singly linked list since most sorting algorithms are built based upon a Python native list. The biggest disadvantage of a linked list is that there is no "back-tracking" which means that a node in a linked list is not able to compare with the it's previous node. This was important to keep in mind throughout the lab.

A. Part 1: Bubble Sort

The Bubble Sort algorithm is the most straight forward algorithm to construct in this lab. The layout of a singly linked list bubble sort is very similar to the layout of a native Python bubble sort algorithm. Even though this sorting method is the most straight forward, it also has the largest expected running time with the worst case being $O(n^2)$. This is because the bubble sort algorithm requires a set of nested for loops, which iterate the list n times each.

B. Part 2: Merge Sort

The main concept of the merge sort algorithm is to "divide and conquer". By dividing and conquering, the program splits the given list until it can't be reduced further and then proceeds to compare each item. Essentially, the algorithym reduces the items by $n/2$ at each iteration.

A design challenge encountered in this section was creating a new list every time the sorting algorithm split the previous list. This means at each iteration, a new list copies the contents of the previous up to or form its middle. The final solution is to create a helper function which is called every time a new copy lists is needed. Merging two copied lists was a simpler task is is just required to connect certain nodes together.

C. Part 3: Quick Sort

Quick Sort also follows the concept of "divide and conquer" as well. Instead of dividing each time by its half, it divides at it's pivot point. The lengths at either side of the pivot may vary which depends how many values are greater or less than the pivot's item. In this version of quick sort both sides of the pivot are sorted, so the position of the pivot against the half of the length is not important, thus increasing its number of comparisons made against the modified quick sort.

The quick sort algorithm was one of the most challenging algorithms to build in this lab. Since the algorithm is designed to work with a single linked list, it has to implement ways to kept track of its previous node without adding more time complexity. Usually, to find a node's previous node, a while iteration is called. At the worst case for the while iteration, it adds $O(n)$ to the time complexity. To prevent this, an additional node is set to the changing pivot and eventually returned to the recursive function. This format will not only prevent from increasing the time complexity but allows the range of which the quick sort will iterate through to be changed.

D. Part 4: Modified Quick Sort

This last section follows the model of the previous section. The main difference between the modified quick sort and quick sort is that the original algorithm sorts both side of the pivot while the modified version only sorts one side. The approach this problem, the location of the pivot on the list must be compared to the predicted location of the median (which is located at half the original length). If the number of values that are less than the pivot is bigger than half of the original length, then the function will continue sorting the left side of the pivot. Otherwise, it will sort the left side. Eventually the program's pivot location will match that of the half of the original length and thus returning the median, only needing to sort once each recursive call.

III. Results

In order to analyze and compare the efficiency's of the sorting algorithms against one another, the number of comparisons made within each algorithm as well as the time it took for execution were tracked, recorded, and plotted. Figure 1 and Figure 3 below displays results of the sorting algorithm for lists of length 0 and 100.

Figure 1: Number of Comparisons

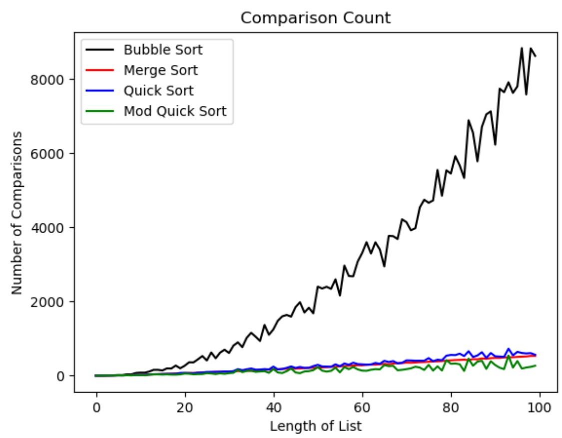
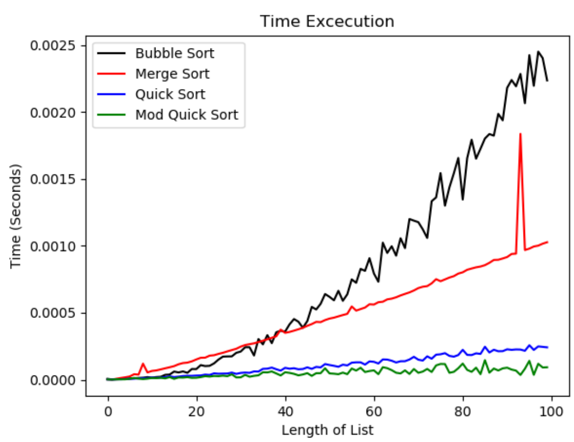


Figure 2: Run Times



Notice that the results are not consistent at times. Even though the number of comparisons made by the merge sort is significantly mush smaller than that of the bubble sort, it may take more time for execution since it is creating new lists at every split. In general, the least efficient method for sorting large amount of items is the bubble sort. It has and follows a $O(n^2)$ for its cases. The second most efficient method for handling large amounts of data is merge sort. Since it iterates through the entire list at least once and essentially halves its range at every recursive call, it's Big O ends up being $O(nlog(n))$. The third most efficient method is quick sort. It follows obtains the same time complexity of $O(nlog(n))$ but executes less comparisons thus being a better algorithm in terms of efficiency. The most effective algorithm of the four sorting algorithms is the modified quick sort with the time complexity of $O(nlog(n))$. Even though it shares the same time complexity as merge sort and quick sort, the modified only sorts the specific half where the median is known to reside. In return, the quick sort algorithm reduces the total amount of comparisons needed to be me made to find the median.

IV. Discussion

Overall, the most efficient method to sort a random generated list of 100 items is the modified quick sort. Figures 1 and 2 support this outcome as they convey as the larger the length becomes, the disparity of comparisons become more larger. Although this might change for smaller lists such as a list that contains 5 items. This may be due to the steps taken within the recursive calls; while it be for creating new lists at each split or simply iterating the entire lists at once.

V. Source Code

```python
"""
Author: Steven J. Robles
Class: CS 2302 Data Structures III
Instructor: Olac Fuentes
TA: Anindita Nath And Maliheh Zargaran
Last Modified: 02/22/2019
Discreption: Lab 2:
  The purpose of this program is to find the mediam on a random generated list of length n. It retrieves the medium
  by first sorting the list in four different ways; bubble sort, merge sort, quick sort and a modified quick sort.
  The result, if chosen to display, shows the number of comparisons made and the given time it took for each sorting.

"""
from BuildLists import BuildList1
from BuildLists import Print
from BuildLists import Copy
from BubbleSort import Section1
from MergeSort import Section2
from QuickSort import Section3
from QuickSortMod import Section4
import matplotlib.pyplot as plt
import matplotlib.pyplot as plt2
import timeit
import numpy as np


#asks for suer input
choice = input("Hello! Do you want to proceed with a time trial?\n1. Yes\n2. No\n")
choiceNum = int(choice)
nInput = input("Input the N'th length desired. \n")
n = int (nInput)

if(choice == "Yes" or choiceNum == 1): #user chose to display results
  print(" * Note : The test will time the sorting algorythms to produce the mediam from lengths of 0 to the inputed length of N")

  #sets the size of the array of which the run times are going to be stored
  counts1 = [0] * n
  counts2 = [0] * n
  counts3 = [0] * n
  counts4 = [0] * n
  times1 = [0] * n
  times2 = [0] * n
  times3 = [0] * n
  times4 = [0] * n

  fig, ax = plt.subplots()
  fig, ax = plt2.subplots()

  for i in range(0, n, 1): #creates a lists to record results from length 0 to n

    L = BuildList1(i)
    if (L.head == None):
      n = 0
    L1 = Copy(L, 0, i)
    L2 = Copy(L, 0, i)
    L3 = Copy(L, 0, i)
    L4 = Copy(L, 0, i)


    start = timeit.default_timer() # starts timer
    Mid1, count = Section1(L1, i)
    stop = timeit.default_timer() # ends timer
    times1[i] = stop - start #stores the lapsed time
    counts1[i] = count
```

```
    start = timeit.default_timer()
    Mid2, count = Section2(L2, i)
    stop = timeit.default_timer()
    times2[i] = stop - start
    counts2[i] = count

    start = timeit.default_timer()
    Mid3 , count = Section3(L3, i)
    stop = timeit.default_timer()
    times3[i] = stop - start
    counts3[i] = count

    start = timeit.default_timer()
    Mid4 , count = Section4(L4, i)
    stop = timeit.default_timer()
    times4[i] = stop - start
    counts4[i] = count

    #proceeds to plot the results for the comparison count
  plt.close("all")
  plt.title('Comparison Count')
  plt.xlabel('Length of List')
  plt.ylabel('Number of Comparisons')
  x = np.arange(0, i+1, 1)
  plt.plot(x, counts1, 'k', label='Bubble Sort')
  plt.plot(x, counts2, 'r', label='Merge Sort')
  plt.plot(x, counts3, 'b', label='Quick Sort')
  plt.plot(x, counts4, 'g', label='Mod Quick Sort')
  plt.legend()
  plt.savefig('ComparisonCounts')
  plt.show()

  #proceeds to plot the results for the time
  plt2.close("all")
  plt2.title('Time Excecution')
  plt2.xlabel('Length of List')
  plt2.ylabel('Time (Seconds)')
  plt2.plot(x, times1, 'k', label='Bubble Sort')
  plt2.plot(x, times2, 'r', label='Merge Sort')
  plt2.plot(x, times3, 'b', label='Quick Sort')
  plt2.plot(x, times4, 'g', label='Mod Quick Sort')
  plt2.legend()
  plt2.savefig('Times')
  plt2.show()


else: #only the medium is displayed

  L = BuildList1(n)
  if (L.head == None):
    n = 0
  L1 = Copy(L, 0, n)
  L2 = Copy(L, 0, n)
  L3 = Copy(L, 0, n)
  L4 = Copy(L, 0, n)

  print("The Medium are: ")

  Mid1, count = Section1(L1, n)
  print(Mid1)

  Mid2, count = Section2(L2, n)
  print(Mid2)

  Mid3, count = Section3(L3, n)
  print(Mid3)

  Mid4, count = Section4(L4, n)
  print(Mid4)
```

```python
"""
Author: Steven J. Robles
Class: CS 2302 Data Structures III
Instructor: Olac Fuentes
TA: Anindita Nath And Maliheh Zargaran
Last Modified: 02/22/2019
Discreption: Lab 2:
    The purpose of this program is to serve as basic manipulation of a likned list. Wheter
    it to build it, copy it, or
    find the middle item, it is all done here. Some functions are called from other files.
"""

import random

class Node(object):

    def __init__(self, item, next=None):
        self.item = item
        self.next = next

class List(object):

    def __init__(self):
        self.head = None
        self.tail = None

def IsEmpty(L):
    return L.head == None

def Append(L,x):

    if IsEmpty(L):
        L.head = Node(x)
        L.tail = L.head
    else:
        L.tail.next = Node(x)
        L.tail = L.tail.next

def Print(L):
    temp = L.head
    while temp is not None:
        print(temp.item, end=' ')
        temp = temp.next
    print()  # New line x

def Copy(L, start, end):

    L2 = List()
    temp = L.head
    count = 1

    while temp is not None and count <= end:

        if (count > start):
            Append(L2, temp.item)

        temp = temp.next
        count += 1
    return L2


def GetMidNode(L, count):
    if IsEmpty(L) or count == 0:
        return None
    else:
        temp = L.head
        i = 1
        while (i != count):
            temp = temp.next
            i+=1
```

```
        return temp.item


def BuildList1(n):

    L = List()

    for i in range(n):
        Append(L, random.randint(1,101))
    return L
```

```
"""
Author: Steven J. Robles
Class: CS 2302 Data Structures III
Instructor: Olac Fuentes
TA: Anindita Nath And Maliheh Zargaran
Last Modified: 02/22/2019
Discreption: Lab 2:
  This pogram is desinged to sort the list using bubble sort in order to retrieve the median
"""

from BuildLists import GetMidNode

def IsEmpty(L): #checks if the the head is empty or not
    return L.head == None

def Sort(L): #Bubble sort function
  track = 0
  if IsEmpty(L):
    return True
  else:
    done = False
    while(done == False):
      done = True
      temp = L.head
      while temp.next is not None:
        track += 1  # keep strak of the comparison count
        if (temp.item > temp.next.item):
          tempItem = temp.item
          temp.item = temp.next.item
          temp.next.item = tempItem
          done = False
        temp = temp.next
  return track

#main functino call from the main program
def Section1(L, n):
  Track = Sort(L)
  return GetMidNode(L, n//2), Track  #returns the medium and the comparison count
```

```
"""
Author: Steven J. Robles
Class: CS 2302 Data Structures III
Instructor: Olac Fuentes
TA: Anindita Nath And Maliheh Zargaran
Last Modified: 02/22/2019
Discreption: Lab 2:
  The following program sorts a given lists using the merge sort method. After sorting the
    items in a
  given linked list, the mediam item is return back to the main files
"""


from BuildLists import Copy
from BuildLists import GetMidNode

def mSort(L, length, track): #merge sort functoin

  if length > 1:

```

```
    mid = length//2
    leftHand = Copy(L, 0, mid)  #creates copies of of the split lists back in Build lists
    rightHand = Copy(L, mid, length)

    track = mSort(leftHand, mid, track) #recursive calls for futher splitting if neccessairy
    track = mSort(rightHand, length - mid, track)

    leftTemp = leftHand.head
    rightTemp = rightHand.head
    temp = L.head

    while leftTemp is not None and rightTemp is not None: #loop which does the comparisons

      if leftTemp.item < rightTemp.item:  #updates the value of the passed in lists if it is smaller than the other one
        temp.item = leftTemp.item
        leftTemp = leftTemp.next

      else: # updates the passed lists with the other item
        temp.item = rightTemp.item
        rightTemp =rightTemp.next

      temp = temp.next
      track += 1  #keeps track of the comparison count

    while leftTemp is not None: #adds left over items to the origional list
      temp.item = leftTemp.item
      leftTemp = leftTemp.next
      temp = temp.next

    while rightTemp is not None: #adds lef over items to teh origional lists
      temp.item = rightTemp.item
      rightTemp =rightTemp.next
      temp = temp.next
  return track

def Section2(L, n): #function call from the main program file

  Track = mSort(L, n, 0)
  return GetMidNode(L, n//2), Track #returns median and comparison count
```

```
"""
Author: Steven J. Robles
Class: CS 2302 Data Structures III
Instructor: Olac Fuentes
TA: Anindita Nath And Maliheh Zargaran
Last Modified: 02/22/2019
Discreption: Lab 2:
  The purpose of this program is to sort a given lists with length n using the quick sort method. The list comes as
  a linked lists and returns the median after is is all sorted
"""

from BuildLists import GetMidNode

def partion(L, start, end, track): #quick sort partion call

  pivotNode = start
  pivotPrev = pivotNode
  transverse = start.next

  while transverse != end.next: #comparison are made here

    if(start.item >= transverse.item): #switched items occur if with the comparison of vlues
      pivotPrev = pivotNode
      pivotNode = pivotNode.next
      hold = pivotNode.item
      pivotNode.item = transverse.item
      transverse.item = hold

```

```
    transverse = transverse.next
    track += 1 #keeps track in the number of comparions

  hold = start.item
  start.item = pivotNode.item
  pivotNode.item = hold

  return pivotPrev, track

def quickSort(L, start, end, track): #quick sort main recursive call function
  if start != end and end.next is not start:

    previousNode , track = partion(L, start, end, track) #partions the lists
    track = quickSort(L, start, previousNode, track)  #the follwing are recursive calles for either side of the pivot
    track = quickSort(L, previousNode.next.next, end, track)
  return track


def Section3(L, n): #main functoin called by the main program file

  Track = quickSort(L, L.head, L.tail, 0)
  return GetMidNode(L, n//2), Track #returns the median and the comparison count
```

```
"""
Author: Steven J. Robles
Class: CS 2302 Data Structures III
Instructor: Olac Fuentes
TA: Anindita Nath And Maliheh Zargaran
Last Modified: 02/22/2019
Discreption: Lab 2:
  The purpose of this program is to sort a given lists with length n using a modified quick sort method. It is
  modified in the aspedt of soritng only the correct side fo the pivot of which the median is known to resdie.
  The list comes as a linked lists and returns the median after is is all sorted.
"""
from BuildLists import GetMidNode

def partion(L, start, end, pivotCount, track): #partion call

  pivotNode = start
  pivotPrev = pivotNode
  transverse = start.next

  while transverse != end.next: # loop which does the comparisons

    if(start.item >= transverse.item): #items switched in the lists according the the values
      pivotCount +=1
      pivotPrev = pivotNode
      pivotNode = pivotNode.next
      hold = pivotNode.item
      pivotNode.item = transverse.item
      transverse.item = hold

    transverse = transverse.next
    track +=1 #keeps track of the number of comparisons made

  hold = start.item
  start.item = pivotNode.item
  pivotNode.item = hold

  return pivotPrev, pivotCount, track

def quickSort(L, start, end, length, pivotLocation, track): #main recusive sort calls

  if start != end and end.next is not start:

    previousNode, pivotCount, track = partion(L, start, end, pivotLocation, track) #calls the partion
```

```

    if pivotCount > length//2: #if mediam resides in the left side, it calls to sort the left
      track = quickSort(L, start, previousNode, length, pivotLocation, track)

    elif pivotCount < length//2: #if median resides in teh right sides, it calls to sort the right
      track = quickSort(L, previousNode.next.next, end, length, pivotCount+1, track)

    else: #returns the lists
      return track
  return track


def Section4(L, n): #function called from main program file

  Track = quickSort(L, L.head, L.tail, n, 0, 0)
  return GetMidNode(L, n//2), Track #returns the comparison count and the median item
```