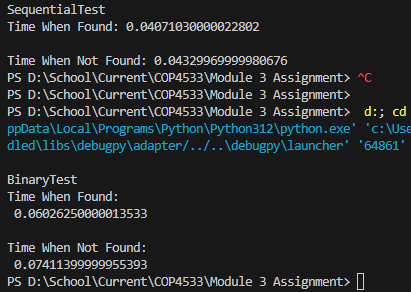
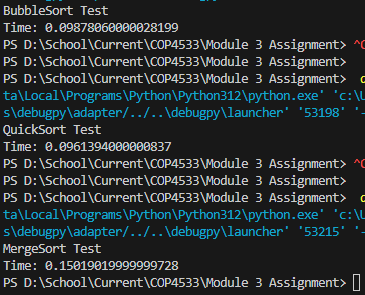
1. Searching Time Analysis

After taking a quick glance at the results, sequential search is faster than binary, but this is dependent on the size of the array. These tests only sort arrays with 20 elements in them, but when the array is much larger, binary sort is better suited. Sequential search has a time complexity of O(n), meaning it may be more effective when the element is in the beginning of the array, however this is not efficient for large data sets. Binary search has a time complexity of O(logn) because it splits the array in half through each iteration, making it a logarithmic algorithm. This is much better for larger data sets compared to sequential search.

1. Sorting Time Analysis

Bubble sort has a time complexity of O(n) because it will iterate once for each element. Bubble sort is best suited for smaller arrays. Quick sort has a time complexity of O(n\*log(n)) and it best used for sorting arrays out of the three studied, as it can scale to large array sizes. The worst-case scenario for quick sort is an array that partitions the sub arrays unevenly each time. Lastly, merge sort has a time complexity of O(n\*log n) because it depends on the size of the overarching array, and may take a long time for small arrays, but much faster for larger arrays.

1. Binary Tree Time Analysis



Binary Tree has a time complexity of O(logn) because it will scale depending on the size of the array. This has a different time complexity that the other three, and this search is much faster than the other three as well.

Source Code

1. Searching

binary\_test.py:

import timeit

class BinaryStringList:

def \_\_init\_\_(self):

self.pylist = []

#add item to end of internal list

def add(self, string):

self.pylist.append(string)

#find method uses binary search

def find(self, string):

low = 0

high = len(self.pylist) - 1

while low <= high:

mid = (low + high) // 2

if self.pylist[mid] < string:

low = mid + 1

elif self.pylist[mid] > string:

high = mid - 1

else:

return self.pylist[mid]

return None

# binarystringlist tests arrays of size 200

test\_list = BinaryStringList()

for i in range(200):

test\_list.add(f"String{i}")

def test\_binary():

return test\_list.find("String17")

def test\_binary\_nf():

return test\_list.find("nothere")

print("BinaryTest\nTime When Found:\n", timeit.timeit(test\_binary, number=100000))

print("\nTime When Not Found:\n", timeit.timeit(test\_binary\_nf, number=100000))

sequential\_test.py:

import timeit

class SequentialStringList:

def \_\_init\_\_(self):

self.pylist = []

#add item to end of internal list

def add(self, string):

self.pylist.append(string)

#find method uses sequential search

def find(self, str):

for i in self.pylist:

if i == str:

return i

return None

# Test for SequentialStringList

sequential\_list = SequentialStringList()

for i in range(20):

sequential\_list.add(f"String{i}")

def test\_sequential():

return sequential\_list.find("String17")

def test\_sequential\_nf():

return sequential\_list.find("notfound")

print("SequentialTest\nTime When Found:", timeit.timeit(test\_sequential, number=100000))

print("\nTime When Not Found:", timeit.timeit(test\_sequential\_nf, number=100000))

1. Sorts

Mergesort\_test.py:

import random

import timeit

class MergeStringList():

def \_\_init\_\_(self):

self.pylist = []

#adds item to internal list

def add(self, str):

self.pylist.append(str)

def merge(arr, l, m, r):

n1 = m - l + 1

n2 = r - m

# create temp arrays

L = [0] \* (n1)

R = [0] \* (n2)

# Copy data to temp arrays L[] and R[]

for i in range(0, n1):

L[i] = arr[l + i]

for j in range(0, n2):

R[j] = arr[m + 1 + j]

# Merge the temp arrays back into arr[l..r]

i = 0 # Initial index of first subarray

j = 0 # Initial index of second subarray

k = l # Initial index of merged subarray

while i < n1 and j < n2:

if L[i] <= R[j]:

arr[k] = L[i]

i += 1

else:

arr[k] = R[j]

j += 1

k += 1

# Copy the remaining elements of L[], if there

# are any

while i < n1:

arr[k] = L[i]

i += 1

k += 1

# Copy the remaining elements of R[], if there

# are any

while j < n2:

arr[k] = R[j]

j += 1

k += 1

# l is for left index and r is right index of the

# sub-array of arr to be sorted

def mergeSort(self, arr):

if len(arr) > 1:

#creates second

mid = len(arr)//2

sub\_array1 = arr[:mid]

sub\_array2 = arr[mid:]

#sort both halves

self.mergeSort(sub\_array1)

self.mergeSort(sub\_array2)

i=j=k=0

#sorts arrays

while i < len(sub\_array1) and j < len(sub\_array2):

if sub\_array1[i] < sub\_array2[j]:

self.pylist[k] = sub\_array1[i]

i += 1

else:

self.pylist[k] = sub\_array2[j]

j += 1

k += 1

#places remaining elements back in

while i < len(sub\_array1):

self.pylist[k] = sub\_array1[i]

i += 1

k += 1

while j < len(sub\_array2):

self.pylist[k] = sub\_array2[j]

j += 1

k += 1

# Test for MergeSort, adds 20 random numbers between 1 and 100

merge\_list = MergeStringList()

for i in range(20):

merge\_list.add(random.randint(0,100))

def test\_mergeSort():

merge\_list.mergeSort(merge\_list.pylist)

print("MergeSort Test\nTime:", timeit.timeit(test\_mergeSort, number=10000))

bubblesort\_test.py:

import random

import timeit

class BubbleStringList():

def \_\_init\_\_(self):

self.pylist = []

#adds item to internal list

def add(self, str):

self.pylist.append(str)

#bubblesorting method

def sort(self):

n = len(self.pylist)

for i in range(n):

for j in range(0, n-i-1):

if self.pylist[j] > self.pylist[j+1] :

self.pylist[j], self.pylist[j+1] = self.pylist[j+1], self.pylist[j]

bubble\_list = BubbleStringList()

for i in range(20):

bubble\_list.add(random.randint(0,100))

def test\_bubbleSort():

bubble\_list.sort()

print("BubbleSort Test\nTime:", timeit.timeit(test\_bubbleSort, number=10000))

quicksort\_test.py:

import random

import timeit

class QuickStringList():

def \_\_init\_\_(self):

self.pylist = []

#adds item to internal list

def add(self, str):

self.pylist.append(str)

#quicksorting method

def quickSort(self, list, low, high):

if low < high:

pi = QuickStringList.partition(list,low,high)

self.quickSort(list, low, pi-1)

self.quickSort(list, pi+1, high)

def partition(list,low,high):

i = ( low-1 )

pivot = list[high]

for j in range(low , high):

if list[j] <= pivot:

i = i+1

list[i],list[j] = list[j],list[i]

list[i+1],list[high] = list[high],list[i+1]

return ( i+1 )

# Test for MergeSort, adds 20 random numbers between 1 and 100

quick\_list = QuickStringList()

for i in range(20):

quick\_list.add(random.randint(0,100))

def test\_quickSort():

n=len(quick\_list.pylist)

quick\_list.quickSort(quick\_list.pylist, 0, n-1)

print("QuickSort Test\nTime:", timeit.timeit(test\_quickSort, number=10000))

binary\_tree.py:

"""Sample Node defined """

import random

import timeit

class BinaryTree:

def \_\_init\_\_(self, val):

self.val = val

self.leftChild = None

self.rightChild = None

def get(self):

return self.val

def set(self, val):

self.val = val

def getChildren(self):

children = []

if(self.leftChild != None):

children.append(self.leftChild)

if(self.rightChild != None):

children.append(self.rightChild)

return children

""" Binary Tree operation performed in this class """

class BinaryTreeStringList:

def \_\_init\_\_(self):

self.root = None

def setRoot(self, val):

self.root = BinaryTree(val)

def insert(self, val):

if(self.root is None):

self.setRoot(val)

else:

self.insertNode(self.root, val)

def insertNode(self, currentNode, val):

if(val <= currentNode.val):

if(currentNode.leftChild):

self.insertNode(currentNode.leftChild, val)

elif val > currentNode.val:

if(currentNode.rightChild):

self.insertNode(currentNode.rightChild, val)

else:

currentNode.rightChild = BinaryTree(val)

else:

currentNode.leftChild = BinaryTree(val)

def find(self, val):

return self.findNode(self.root, val)

def findNode(self, currentNode, val):

if(currentNode is None):

return False

elif(val == currentNode.val):

return True

elif(val < currentNode.val):

return self.findNode(currentNode.leftChild, val)

else:

return self.findNode(currentNode.rightChild, val)

""" Test the functionality using below driver class"""

bst = BinaryTreeStringList()

root = bst.setRoot(12)

tree\_list = BinaryTreeStringList()

for i in range(20):

tree\_list.insert(random.randint(0,100))

def test\_binaryTree():

tree\_list.find(25)

print("BinaryTree Test\nTime:", timeit.timeit(test\_binaryTree, number=10000))