

Lab-X

Hasan Amin

(374866)

CS-250

Data Structures and Algorithms

School of Natural Sciences

Contents

[Task 1: 3](#_Toc151832947)

[Task 2: 7](#_Toc151832948)

[Task 3: 7](#_Toc151832949)

# Task 1:

|  |
| --- |
| **Code** |
| import matplotlib.pyplot as plt  import numpy as np  import timeit  def selection(mylist):      n=len(mylist)      for i in range(0,n-1):          min=i          for j in range(i+1,n):              if mylist[j]<mylist[min]:                  min=j          mylist[i],mylist[min]=mylist[min],mylist[i]      return mylist    def insertion(mylist):      n=len(mylist)      for i in range(1,n):          key=mylist[i]          j=i-1          while j>=0 and mylist[j]>key:              mylist[j+1]=mylist[j]              j-=1          mylist[j+1]=key      return mylist  def bubble(mylist):      n=len(mylist)      for i in range(n-1):          flag=False          for j in range(n-i-1):              if mylist[j]>mylist[j+1]:                  mylist[j],mylist[j+1]=mylist[j+1],mylist[j]                  flag=True          if not flag:              return mylist      return mylist  def time\_sorting\_algorithm(algorithm, arr):      # setup\_code = f"from \_\_main\_\_ import {algorithm}, arr"      setup\_code=f"import numpy as np; from \_\_main\_\_ import {algorithm}"      stmt = f"{algorithm}({list(arr)})"      # print(stmt)      execution\_time = timeit.timeit(stmt, setup=setup\_code, number=int(np.log10(len(list(arr)))))      return execution\_time  rng = np.random.default\_rng()  # Arrays to test (using np.random.uniform for floats)  arr1 = rng.uniform(low=0.0, high=1000.0, size=100)  arr2 = rng.uniform(low=0.0, high=1000.0, size=1000)  arr3 = rng.uniform(low=0.0, high=1000.0, size=5000)  # List of sorting algorithms  sorting\_algorithms = ["selection","insertion","bubble"]    ###############################    def run\_trials(algorithms, arrays, num\_trials):      results = {alg: {len(arr): [] for arr in arrays} for alg in algorithms}      for \_ in range(num\_trials):          for arr in arrays:              for algorithm in algorithms:                  time\_taken = time\_sorting\_algorithm(algorithm, arr)                  results[algorithm][len(arr)].append(time\_taken)      avg\_results = {alg: {size: np.mean(times) for size, times in data.items()} for alg, data in results.items()}      return avg\_results  # Number of trials to run  num\_trials = 5  # Run trials for sorting algorithms on arrays and get average times  average\_results = run\_trials(sorting\_algorithms, [arr1, arr2, arr3], num\_trials)  ascending\_average\_results=run\_trials(sorting\_algorithms, [np.sort(arr1),np.sort(arr2),np.sort(arr3)],num\_trials)  descending\_average\_results=run\_trials(sorting\_algorithms, [np.sort(arr1)[::-1],np.sort(arr2)[::-1],np.sort(arr3)[::-1]],num\_trials)  # Plotting average results  num\_scenarios = 3  # For Average, Ascending, Descending  fig, axs = plt.subplots(num\_scenarios, 1, figsize=(10, 8\*num\_scenarios))  scenarios = ['Average', 'Ascending', 'Descending']  for i, scenario in enumerate(scenarios):      for algorithm in sorting\_algorithms:          sizes = []          avg\_times = []          if scenario == 'Average':              data = average\_results[algorithm]          elif scenario == 'Ascending':              data = ascending\_average\_results[algorithm]          else:              data = descending\_average\_results[algorithm]            sizes = list(data.keys())          avg\_times = list(data.values())          axs[i].plot(sizes, avg\_times, marker='o', label=f"{algorithm}")      axs[i].set\_xlabel('Sizes')      axs[i].set\_ylabel('Average Times')      axs[i].set\_title(f'Comparison for {scenario} Scenarios')      axs[i].legend()  plt.tight\_layout()  plt.show() |

|  |
| --- |
| **Output** |
|  |

# Task 2:

1. a. You have already implemented a function that prints all elements of a list of size n, where n>=0. What is the Big-O complexity of that operation?
2. Ans.O(n)
3. b. Suppose you have an **array-based list** of size **n**. Implement a function takes a position number **pos** as input from the user, and returns the value stored at that position. What is the Big-O time complexity of this function?
4. Ans.O(1)
5. c. Suppose you have a **singly linked list** of size **n**. Implement a function takes a position number **pos** as input from the user, and returns the value stored at that position. What is the Big-O time complexity of this function? What is its best-case time complexity?

def locate\_element(self, index):

        currnode = self.head

        for i in range(index):

            currnode = currnode.nextnode

        return currnode.data

# O(index) time complexity,O(c) time complexity for best case where c=3

# Task 3:

|  |  |
| --- | --- |
| **Operation** | **Big-O Complexity** |
| Insert an element at the front of a singly linked list of size n | O(1) |
| Insert an element at the tail end of a singly linked list of size n. | O(n) |
| Delete the last node of a singly linked list of size n. | O(n) |
| Insertion at the front of an array list of size n |  |
| Insertion at the tail end of an array list of size n |  |
| Enqueue in a queue of length n. | O(1) for array based |
| Dequeue in a queue of length n. | O(1) for array based |
| Converting an expression of length n from infix to postfix form using stack | O(n) |
| Finding an element via Binary Search algorithm in a sorted array-list of size n. | Log(n) |
| Finding an element via Binary Search algorithm in an **unsorted** array-list of size n. Think about it! | The worst case will result in failure to locate the element. Another way is to first sort using say bubble sort with O(n2) and the binary search. The whole operation will be of **O(n2) time complexity** |