CISC-235 Data Structures W23
Assignment 1

January 14, 2023

General Instructions

Show your solution steps with your findings to the questions in a pdf file named "235-1234-Assn1.pdf", where 1234 stands for the last 4 digits of your student ID. If you cannot save your file as pdf, you may save it and submit it as a Word document and name it "235-1234-Assn1.docx".

Write your own program(s) using Python. Once you complete your assignment, place all Python files in a zip file and name it according to the same method, i.e., "235-1234-Assn1.zip". Unzip this file should get all your Python file(s).

Then upload 235-1234-Assn1.zip and 235-1234-Assn1.pdf into Assignment 1's entry on onQ. You may upload several times if you wish. However, onQ keeps only the last uploaded file. The newly uploaded file will overwrite the old file. Please check your files after uploading. We will check the latest submission you made following the required naming.

You must ensure your code is executable and document your code to help TA mark your solution. We suggest you follow PEP8[1] style to improve the readability of your code.

All data structures involved must be implemented by yourself, except for the built-in data types, i.e., List in Python.

An "I uploaded the wrong file" excuse will result in a mark of zero.

1 Algorithm Complexity Analysis (20 points)

1.1 Count the Number of Operations (10 points)

Analyze the time complexity of the program shown in Figure 1, briefly describe how you calculate the number of operations and provide the final program complexity function.

[1] `https://realpython.com/python-pep8/`

```python
def function(seq):
    d = 0
    n = len(seq)
    for i in range(n-1):
        for j in range(i+1, n):
            d += seq[i] * seq[j]
    return d
```

Figure 1: Python Function for Question 1.1

1.2 Big-Θ Proof (10 points)

Use the definition of big-Θ to prove that the following operation function T(n) $\in \Theta(n^4)$.

$$T(n) = n^4 - 10n^2 + 50$$

2 Binary Search or Linear Search? 50 points

Let us analyze the time complexity of two algorithms, i.e., linear search and binary search, using the experimental method.

Our goal is to compare linear and binary search efficiency for a general search scenario, i.e., search a list of integers from another list of integers. We can implement the search using a function *search(list_target, list_source)* - it searches each of the integers in *list_target* in another list named *list_source*. For instance, search([1,3], [1,5,6]) means searching for 1 in [1,5,6] and searching for 3 in [1,5,6]. If the length of list_target equals 1, it means searching for one integer in a list.

We know that when using linear search, searching for a value from n values will require an average of $n/2$ comparisons, while in the worst case, searching for a value that is not in the list will require n comparisons. Searching for any value will require an average of $logn$ comparisons when using binary search. However, before applying binary search, you should sort the list_source once and then perform several searches, and sorting the list will take $O(nlogn)$ time.

If we are doing a very small number of searches, linear search is preferable. However, if we are doing many searches of the same list, binary search is preferable since the time required to sort the list once is more than offset by the reduced time for the searches. This is what complexity theory tells us.

Your task is to conduct experiments to explore the relationship between the size of the list and the number of searches required to make binary search preferable to linear search. See the detailed requirement below:

1) Implement two algorithms for the general search scenario using Python. You must write your own code for binary search and linear search. For the sorting algorithm, you may choose any sorting algorithm that has

complexity in $O(nlogn)$. If your sorting code is modified from an online resource, you need to add a reference in the comment.

2) For n = 100, 1000, and 10,000, conduct the following experiment:

   - Use Python library *random* to create a list named list_source containing n integers, with seed = 12345. You can call "random.seed(12345)" to control the seed value. We use the seed to make sure the TA can reproduce your results.

   - Choose k target values to form list_target, make sure 50% of the values in list_target are in list_source and the rest 50% are not in list_source. You can round the number up if 50% * k does not result in a integer.

   - Use binary search and linear search separately to search list_target in list_source. **Note, when recording the time for the binary search algorithm, you must include the time for sorting the list_source once.**

   - Design and conduct experiments to determine the approximate smallest value of k for which binary search becomes faster than linear search. This means you should try different k values, starting from a small one, and increase it until you observe that the binary search method is faster than the linear search method.

   - Provide a short description in your written report (the pdf file) on how you generate the list_source and list_target, how you determine the smallest value of k, and what is the smallest value of k you find.

Hint: When generating the list_source, you can use random.sample(range(0, m), n) to generate n random values in the range of 0 to m. When generating list_target, you can randomly pick 50% of the k values from the list_source, and then randomly generate the rest values in a range that does not overlap with 0 to m. For instance, you can generate integers larger than m or smaller than 0. There could be other methods as well. You do not need to follow this hint.

3 Develop a Special Bot Leveraging Stack: 30 points

Let us implement a special bot using a Stack data structure. This bot holds an empty sequence of data named *_data_items* when initialized. It then reads a list of string operations as input and performs the corresponding manipulation on *_data_items*.

The i-th item in the input list represents one operation that the bot needs to perform. The types of the operations are the follows:

1. "A": add a new integer to *_data_items* that is the sum of the previous two integer values in *_data_items*

2. "T": add a new integer to *_data_items* that is the triple of the previous integer value in *_data_items*.

3. "D": Delete the previous integer value from *_data_items*.

4. An integer: Add the integer to *_data_items*.

Your goal is to implement a Stack and use it to implement the special bot described above. The bot should have a function that takes a valid list of strings as input and return the sequence of integers that the bot collected in *_data_items*. Write your own test case to demo how your algorithm works. For simplicity, you can assume that the input list is always valid. However, we encourage you to think about how to handle invalid cases.

Hint: I can give you one test case. Input: operations = ["10","3","D", "T","A"], your bot should output [10, 30, 40]. *_data_items* could be a Stack.