Assignment 2. Sorting and Hash Tables

Data Structures & Algorithms 2022-2023, BKI, UvA

Deadline for this assignment: Friday March 3rd, 5pm

Readability: In these assignments, we require that your code satisfy the Python style standards, as measured by the `flake8` tool used in CodeGrade. One point of your grade is awarded for passing the style check with zero warnings, please ensure this. You may install `black` and use `black -l 80 myfile.py` to automatically reformat your code and get rid of most warnings, but please do check your files before submitting to ensure they are readable, and check the CodeGrade output for style warnings. (5 pts)

Introduction

Sorting lists by some criterion is a common step in processing data, with applications ranging from straight-forward ones (sorting through a web store by price) to more subtle ones (speeding up search operations.) In this assignment, you will get familiar with implementation and behavior of a few of the most important sorting algorithms.

Order. Sort in ascending order, i.e., `[1, 3, 8]` is sorted, `[8, 3, 1]` is not.

Stability. Insertion and merge sort are stable. The quicksort variants in this assignment are not.

In-place vs out-of-place. Sometimes, you want to keep the original list untouched and produce a new sorted copy. Other times it's better to reuse the memory.

Space complexity. In class, we learned big-$\mathcal{O}$ notation for talking about the growth rate of functions, and used this notation to talk about the *number of operations* performed by a program (i.e., *time complexity*). The same concept can also be used to talk about the amount of storage required by a program, and this is called *space complexity*. We will reference this concept in this assignment.

New Python concepts. We introduce some new Python features to make code nicer. Refer to the end of this document for details.

1 Sorting a Python List (Array Data Structure)

(See provided file `sort_array.py`)

This material was covered in Lectures 4 and 5.

A store's inventory is organized as a Python list, with each element being an instance of the following Product structure.

```
@dataclass
class Product:
    name: str
    price: float
```

Implement the following functions.

1. `sort_insertion(products: List[Product]) -> List[Product]`.

   (The insertion sort algorithm, in-place, as shown in class.) (3 pts)

2. `sort_merge(products: List[Product]) -> List[Product]`.

   (The merge sort algorithm, recursive. You will also have to implement the auxilliary function `merge(a: List[Product], b: List[Product]) -> List[Product]`, which takes two sorted lists `a, b` and creates a new sorted list `c` by interleaving their elements.) (3 pts)

3. `sort_quick_lomuto(products: List[Product]) -> List[Product]`.

   (The quicksort algorithm, with the Lomuto partitioning scheme, in-place, as in the lecture. Use the internal function `_quicksort_recurse_lomuto(lst: List[Product], p: int, r: int)` for the main recursion on sub-arrays, and implement the `_partition_lomuto(lst: List[Product], p: int, r: int) -> int` function as in the lecture, using the last element as pivot.) (2 pts)

4. `sort_quick_hoare(products: List[Product]) -> List[Product]`.

   (The quicksort algorithm using the Hoare partition scheme, implemented in-place. This slightly more efficient variant has the same basic principle, but a slightly different implementation of the partition function. As in the previous point, you must implement the `_quicksort_recurse_hoare` and `_partition_hoare` functions. The main recursion is almost identical to the previous point, except that the left split must include the pivot index. The pseudocode of Hoare's partition scheme is given at the end of this document and uses the middle element as pivot, which avoids some worst-case situations. You may read more at the Wikipedia page on Quicksort.

(2 pts)

Implementation notes. Python lists are implemented as arrays, so you can safely use the syntax `products[i]` to retrieve (GET) or assign to (SET) the ith element in constant ($O(1)$) time. You can also use `len(products)` to get the total length. You should not use the built-in sorting functions in Python (other than for your own testinge.)

2 Sorting a Linked List

(See provided file `sort_linked_list.py`)

This material was covered in Lecture 4.

We are back in the weather station environment from the previous assignment. Temperature measurements are stored a linked list (an instance of the class `TemperatureList`, with nodes structured as

```
@dataclass
class Node:
    degrees: float
    link: Optional[Node] = None
```

You are provided a bare-bones `TemperatureList` implementation, including the method `TemperatureList.sort_merge`. However, this method invokes two functions that are not yet implemented: `TemperatureList.split_half` and `merge(a, b)`. You will implement them, as specified next.

1. `TemperatureList.split_half(self) \`
   `-> tuple(TemperatureList, TemperatureList)`
   This function finds the middle two nodes of the current list (say $n_1 \to n_2$, severs the link between them, and create a new `TemperatureList` with n_2 as the head. This should take $\Theta(N)$ steps and you should not need to create any new Nodes. (4 pts)

2. `merge(a: TemperatureList, b: TemperatureList) \`
   `-> TemperatureList`
   This function takes two linked lists, assumed sorted, and merges them **in place**. This means you should not create copies of the existing nodes, but instead change the links between existing ones. (6 pts)

The resulting merge sort should operate entirely in place, without new `Nodes` as copies of old `Nodes`; this makes it a particularly nice choice for sorting linked lists. Make sure all returned lists have the correct `_length` attributes. You are allowed to create only a constant number of helper `Nodes`.

Implementation notes. In Python, *double underscore* methods are used to make user-defined classes play nice with built-in operators. For instance, if a class implements the `Class.__len__(self)` method, Python will automatically call that method whenever you invoke the builtin `len(x)` for an object `x` of type `Class`. In the provided implementation, we use this strategy to give access to the length of an `TemperatureList`. Similarly, `Class.__repr__(self)` is called every time Python is asked to display an object, e.g., using `print(x)`. (Check out how this works at line 119 of the provided file.)

Rather than having to compute the length every time by traversing the list, which takes $\Theta(N)$ steps, our provided implementations **keeps track** of the length of every list inside a secret `_length` attribute. Because of this, inside the `insert` method, we also update the `_length` attribute: otherwise the object might end up in an *inconsistent state*, which could break everything. The reason `_length` is considered secret (private) is because we do not want users to modify it directly, only to read it by calling `len(mylist)`. As mentioned in class, underscore methods and attributes are considered *internal* in Python: users can modify them but should not.

3 Build A Dictionary For A Dictionary

(See provided files `dictionary_demo.py` and `dictionary.py`)

This material is covered in Lecture 6.

In this problem you will build a Dutch-to-English translation dictionary, by implementing your own Dictionary ADT from scratch, using a hash table.

1. First, let's see how a finished dictionary implementation can be used. Modify `dictionary_demo.py` to populate a builtin Python dictionary with the word definitions. Check that running the file prints the correct translations for the selected example words. Feel free to try some other words! (1 pts)

2. Now, let's implement our own dictionary, in the form of a hash table using the chaining strategy:

   (a) Implement the `Bucket` data structure as a singly-linked list, filling in the `__init__`, `lookup`, `insert`, `delete`, and `__len__` methods. (5 pts)

   (b) Implement the `Dictionary.lookup` method. (1 pts)

   (c) Implement the `Dictionary.update` method. (1 pts)

   (d) Implement the `Dictionary.delete` method. (1 pts)

(e) Implement `Dictionary.load_factor` method to report the load factor α. (1 pts)

Check your implementation using the `dictionary_demo.py` file. All lookups should give the same answer as they did for the Python dictionary. (1 pts)

3. Implement the following hash functions.

    - `_hash_first`: the hash of a word is the character value of the first character in the string, modulo the number of buckets. (0 if empty string.) *(already implemented)*
    - `_hash_length`: length of the string (modulo n. buckets) (2 pts)
    - `_hash_sum`: sum of the char. values in the string (modulo n. buckets) (2 pts)

    To test and switch out the function used by the hash table, change line 76 from `_hash = _hash_first` to, e.g., `_hash = _hash_length`. This line effectively creates an ”alias” method `_hash` that calls the specified method. It doesn’t matter which hash function is in use when handing in the assignment as long as it’s correct.

4. Now we are going to do a small case study on the efficiency of every hash function. This part should be handed in a pdf.

    Plot the distribution of buckets for the three different hash functions using three different bucket sizes of your choice (nine plots in total). The plot should be a **bar plot**, with a bar for each bucket, the height of each bar being the number of entries assigned in that bucket. The buckets should be labeled $0, 1, 2, \ldots$. *(Hint: In matplotlib, use `plt.bar` to make the plot and `plt.xticks` to set the tick labels.)*

    Based on the plots, answer the following questions: do the three hash functions spread the words uniformly? Which hash functions are better and which are worse? Use the provided `Dictionary.bucket_sizes` method. You don’t have to submit the code that creates the plots. (4 pts)

5. Finally we want to create a dictionary that can be used to translate English-to-Dutch as well. For this purpose we need to interchange (invert) all keys and values. Implement `Dictionary.invert`, a method that creates a new empty dictionary and inserts “inverted” forms of every entry in the current dictionary. The current dictionary should be left unchanged.

    The problem is that some Dutch words could have been translated to the same English words: “wildernis” → “desert”, “woesteij” → “desert”, hence only one such pair will remain after inversion. You can choose which value from the initial dictionary will remain as a key in the new dictionary (any strategy will pass). However, you should through a warning message, indicating the number of lost pairs. *(Hint: You will need you will need some private methods or attributes of the current dictionary, but just the . To calculate the number of pairs lost you can use the `len` function.)*

(4 pts)

Additional notes

Hoare partitioning pseudocode. (Following the notation in the Quicksort lecture slides.)

As in the lecture, the convention is that `QuicksortHoare(A, p, r)` sorts the sublist `A[p:r]`, i.e., including index p but not index r.

```
function QuicksortHoare(A, p, r)
  if r - p > 1 then
    q = PartitionHoare(A, p, r)
    QuicksortHoare(A, p, q + 1)
    QuicksortHoare(A, q + 1, r)
  end if
end function

function PartitionHoare(A, p, r)
  x = A[ floor((p + r - 1) / 2) ]  # middle element
  i = p - 1  # index right before start of list
  j = r      # index right after end of list

  loop:
    # find next smallest i with A[i] >= x
    do
      increment i
    while A[i] < x

    # find next largest  j with A[j] <= x
    do
      decrement j
    while A[j] > x

    if i >= j then  # it means x is in its desired position!
       return j     # we are done
    end if

    # otherwise, swap, putting i and j in order relative to x.
    exchange(A[i], A[j])

  end loop
end function
```

New Python features used this week. As the assignments are getting more realistic, the provided code will make use of more advanced Python features that can help readability or code organization.

The provided code uses python dataclasses, a recent feature that allows us to define simple record types with little code. See `https://docs.python.org/3/library/dataclasses.html` for more info.

Additional Python tips. Python supports multiple assignments at the same time, so a convenient way to exchange (swap) two variables a, b is `a, b = b, a`. Integer division in Python 3 is denoted by // and always returns the floor if the result has a remainder, for instance `7 // 2` gives back 3.