UPPSALA UNIVERSITY
Artur Kulmizev

5LN715 Assignment package 2 – search, sort, and linked lists

The soft deadline for this assignment package is February 19, 2021.

This lab contains three tasks. One each on searching, linked structures and sorting. In addition to writing code, you should also show that you can relate the theory on time complexity to actual measurements of how long time it takes to run different algorithms. You will try two ways to measure time, by measuring the actual time it takes to run some code, and by counting the number of operations performed by your code.

You are provided with some files, both some code files, and some text files to use for testing your code. You can copy all files from:

`/local/kurs/advprog/module2/`

1 Searching

In this task you are asked to implement different data structures and compare the performance of searching on them in Python, using different methods. You will be given files with English words in alphabetical order, to search for and among. You will time your code and discuss the actual time your codes takes to run compared to the theoretical expectations.

Provided files

- `sortedWordList.txt` – A file containing English words ordered in alphabetical order. These are the words you should search among.
- `toSearchFor.txt` – A file containing unsorted English words, which you should search for.
- `keys.txt` – A file with pairs of words and indices. Each index corresponds to the position of the word in `sortedWordList.txt` (starting indexation at 0) if it is in there, or is -1 otherwise. This file should be used for testing that your search code works correctly.

You should read in all words in `sortedWordList.txt` and store them in different python data structures, and then later search for the words in `toSearchFor.txt`. Some of the words in `toSearchFor.txt` exist in `sortedWordList.txt`, and some of them don't.

There are several options for timing your code. One option is:

```
from timeit import default_timer as timer

start = timer()
# run your code
end = timer()
print(end - start)
```

Note that it is important that you only time your actual calls to search and do not measure the time to read in the files or to print the results, as reading and printing might influence the timing. The testing of your code should not be included in the time measurements. Also note that this method of timing is far from perfect, since other processes on your computer will influence it. If you run the timing several times your results will likely vary. Be aware of these issues in your discussion! If you find that the time differences between the algorithms are small, you may repeat the search multiple times.

Your tasks:

1. Search in Python lists
    - Read in the words in `sortedWordList.txt` into a Python list.
    - Write code that implements the search methods sequential search (two variants) and binary search for lists. In the course book and on the slides, sequential search is implemented using a while loop. You should use some variant of a for-loop in your implementation. You should implement two variants of sequential search, one that uses indexing (e.g. `myList[i]`), and one that use some kind of iterator (e.g. by using `enumerate`). The input should be a sorted list of words (from `sortedWordList.txt`), and a word that you search for. The output should be the index of the word in the list, or -1 if the word is not in the list. Check that your code is correct, by seeing that it gives the correct answer for the words in `keys.txt`.
    - Run the code for each search algorithm searching for each word in `toSearchFor.txt`. Time this, and take note of how long time it took for each of the two algorithms.
2. Search in Python deque
    - Repeat subtask 1, i.e. implement and time sequential (two variants) and binary search, but use deque instead of list. In most cases you should be able to reuse your code from subtask 1, since it is likely to also work on deque.
3. Search in Python dictionaries
    - Read in `sortedWordList.txt`, and store each word in a Python dictionary, with its position in the file as value, starting with 0 (i.e. the first word in `sortedWordList.txt` should have the value 0, the second word value 1, and so on).
    - Implement a simple search function on your dictionary, that returns the value, i.e. the index of the word, for words in the dictionary, and -1 for words not in the dictionary. Test this on `keys.txt`.

- Apply your search function on all the words in `toSearchFor.txt`, and time it.

4. Write a short report (around 1/2 A4), where you discuss your results, and compare them to the theoretical time complexity for the different methods of search for the three data structures. (Binary search for deque will probably be faster than your expectations, since deque uses blocking and indexation is faster than for a standard double ended queue).

2 Linked structures

In this part you will implement a linked list from scratch, which maps tokens to integers (frequencies). You are provided a file, `text.txt`, containing some tokenized English text, for testing your code. The list should be sorted in alphabetical order.

Provided files

- `text.txt` – a file containing lower-cased and tokenized English text (taken from Europarl)
- `linkedTest.py` – Python code for testing the implementation of your code. It is vital that you use exactly the names defined in this text for writing your own code in order for this code to work. You should not modify this file (but you may print extra things while developing, in order to make sure things work correctly)

Your task is to implement a singly-linked list. You should have a node class containing a reference to the next node, as well as a word, and an integer representing the frequency of a word. You should also have a list class, FreqLinkedList, containing a reference to the first node in the list, and the three methods below.

- `addWord(String word)` – should add the word to the list in alphabetical order, with frequency 1 if the word is not already in the list, or increase the frequency by one if the word is already in the list.
- `printList()` – should print each word and frequency in the list
- `filterWords(int N)` – should remove all words with a frequency less than N from the list.

There are different ways to implement this list. You can either let the reference to the first node be None when the list is empty, or have a "dummy node" when the list is empty. The latter option can make the implementation of some methods easier.

You are provided a test program that reads in all words in the file `text.txt`, adds them to your linked list, filters away all words less frequent than 50 and finally prints the remaining list. The output of your program should be:

```
,     194
.     187
a     57
and     57
i     78
in     81
is     63
of     128
that     59
the     248
this     74
to     127
```

Note that this task is only intended as an exercise in linked structures. If you really want to use a sorted frequency list in your programs, there are more efficient methods, for instance using a dictionary and an efficient sorting method or a binary search tree.

3 Sorting

In this part you will implement two sorting algorithms in a set framework and compare how the theoretical time complexity of the algorithms compare to the numbers of swaps and comparisons performed in the different algorithms during sorting.

Provided files

- `sortarray.py` – This file contains a class SortArray, which wraps a list of integers that you are asked to sort. The class provides methods for initializing the list in different ways, and methods that are needed for sorting and printing. You can only use these methods when solving this task. You are NOT allowed to directly modify the list (i.e. you cannot use `sa._data` if your instance of the class is called `sa`). The code counts the number of swaps and comparisons as well, allowing you to estimate the efficiency of the different algorithms. The three methods you will need for the sorting algorithms are:
  - `swap(i,j)` – swaps the items at indeces i and j.
  - `cmp(i,j)` – compares the items at indeces i and j. It returns a digit:

    `<0 if _data[i]<_data[j]`

    `0 if _data[i]==_data[j]`

    `>0 if _data[i]>_data[j]`
  - `getSize()` – returns the size of the list

  You can decide on your own which size the list should have, and how it should be scrambled. There are three options for scrambling, that the list is randomly scrambled (`shuffle`), that it is randomly scrambled but with minimal scrambling (`miniShuffle`), so it is relatively close to sorted, and that the list is reversed (`reverse`). Different

sorting algorithms have different strengths and some work better or worse in some of these conditions. The code in this file should not be modified by you!

- `selectionSort.py` – Code that implements the selection sort algorithm using the SortArray class, and prints info about sorting with different initializations and sizes of the list. This can serve as a starting point for your own implementations.

You will need to look at the code of both classes in order to understand it and complete your own assignments.

Your task in this assignment is to implement the following sorting algorithms in the given framework:

- One sorting algorithm with complexity $O(n^2)$, choose between:
  - **Insertion sort** (Note that you cannot use the (slightly more efficient) variant that shifts elements, only swap is available.)
  - **Bubble sort** (The improved version from the slides, where sorting stops if no swap was made during an iteration.)
- One more complex sorting algorithm, choosing from:
  - **Quick sort** (Note that you need to implement an in-place version of quick sort. You are free to choose the method for picking the pivot (e.g. the first item, a random item, the median of three items, ...)
  - **Shell sort**. (In shell sort there are different methods for choosing the gap sequence. You may use any method, except dividing by 2. Finding the next gap by dividing by 2 is typically inefficient, but it is the gap described in the course book. You can read up on your own to find another gap sequence, or you could use the method to divide the gap by 2.2 and round down, which tends to have good performance. Think carefully about what your first gap should be!)

Remember that you are not allowed to directly access the list, or any other instance variables in the SortArray class. You can only use the methods provided in the given code.

Once you have implemented your two algorithms, make sure that they work correctly. Then you should compare them, and selection sort, for some different sizes and configurations of lists. The code in `selectionSort.py` is a good starting point for how to do this.

In addition you should write a short report (about 1/2 A4) about the theoretical complexity of the four algorithms, and compare the theoretical expectations to your own measurements. Note that the performance will vary between runs for some of the algorithms! To account for that, either run it several times, or average over multiple runs in the code! Also state which variant of quick sort or shell sort you implemented (i.e. how you choose pivot or gaps). Your report should mention the concept "adaptive".

What to hand in

You should hand in all your own code in a `.zip` archive. If you used your own tests and print statements during development (recommended), these should be removed or commented out before handing the final code in. You should also hand in a short report discussing the time complexity for task 1 and 3, searching and sorting. Hand in your files through Studium. The soft deadline is **Friday, February 19, 2021 at 20:00 CET**, after which you will receive feedback within 14 days. The hard deadline is **March 19th at 20:00 CET**. You will receive no feedback if you submit for this deadline.