Data Structures and Algorithms with Python

Workshop 2

In this workshop you'll implement search and sort algorithms in Python. You'll also practice reading Python code written by others, and work on the concept of Big-Oh complexity. A skeleton code for these exercises is provided in the Python file available on the Hub. Open this file in Spyder and you're ready to go. This is the core part of the workshop.

The main workshop is followed by two extra problems — this time, you can explore a dataset of Fortune 500 companies, as well as the merge sort algorithm, which greatly improves sorting speed compared to selection sort. This is intended for those of you who wish to challenge themselves and go further than the required course material. The exercises may be open to different approaches and sometimes involve new concepts — Google may be your friend here.

1 Searching

In the lecture we started to delve into the mechanics of searching for an item in a list. We first looked at how linear search goes through each element x in list A, as defined below.

```
1  def linearSearch(A,x):
2      for elem in A:
3          if elem == x:
4              return True
5      return False
6
7  y = 6
8  linearSearch(A,y)
```

Try it out and make sure you understand how the function works. Notice that the elements of the list do not need to be integers (or indeed numbers).

We saw in the lecture that the worst-case running time of this algorithm is $O(n)$, where n is the length of A: that is, if the element we're looking for happen not to be in the list, we have to go

through all n elements.

If your list is sorted, a better solution is to use **binary search**. We defined the algorithm in the lecture using the following "recipe" to search for item x in list L.

- Pick an index i roughly dividing L in half
- If there is nothing left to search, return False
- If $L[i] == x$, return True
- If not:
  - If $L[i] > x$, recursively search left half of L
  - Otherwise recursively search right half

Complete the skeleton code given in the Python file. Your function should recursively call itself, narrowing down the section of the list that it is searching. It may be helpful to use `print`-statements in your code to see what is happening in the recursion.

```
def binarySearch(A,x,low,high):
    """Assumes A is a list with elements in ascending order.
            x is the element we're searching for.
            low and high are indices of the list in between which we search.
        Returns True if x is in A and False otherwise"""
    if low >= high: # if range is empty: nothing left to search
        if A[low] == x: # if just found x, return True
            return True
        else: # otherwise did not find x
            return False
    midpoint = (low + high) // 2 # divide list roughly in half using integer division
    if A[midpoint] == x:
        return True # found x!
    elif A[midpoint] > x:
        return binarySearch(A,x,low,midpoint-1) # search left side of midpoint recursively
```

```
    else:
        return binarySearch(A,x,?,?) # code: search right side of midpoint recursively
```

Verify that your search works and compare the speed of your implementation with linear search. You can use the `%timeit` command to time the performance of your functions. Note that this command works in the "enhanced" IPython console that Spyder uses, but not in a standard Python console.

```
%timeit linearSearch(A,y)
```

2 Sorting

We saw earlier how searching becomes much faster when we can exploit the fact that a list is sorted. Today we've looked at how sorting algorithms work under the hood. You'll implement the two methods you saw in the lecture, selection sort and merge sort.

The selection sort algorithm uses searching as a subroutine. In each iteration of the main loop, it searches for the smallest element in the unsearched part of the list and moves it to the end of the searched part. The algorithm is as follows:

Selection sort of list L of length n:

- Initialization step: divide the list into a "prefix" P and a "suffix" S with initially P empty and $S = L$
- Main loop:
  - Search for the smallest element of S and move it to the end of P
  - Repeat until S is empty

Complete the skeleton code in the Python file. The idea in this implementation is to keep track of the prefix and a suffix using a single index, `prefixSize`. Each go of the main loop goes adds another element to the prefix, increasing `prefixSize` by one. Your task is to make sure that the algorithm finds the minimum element of the suffix and moves it to this index.

```
def selectionSort(A):
"""Assumes A is a list (whose elements can be compared).
Returns sorted list"""
# see code in the Python file provided
```

Although the selection sort algorithm is intuitive, it is not as fast as we would hope for: its running time is $O(n^2)$ for n items. We can do better by other algorithms, for example using merge sort, which is a recursive "divide-and-conquer" algorithm. It significantly improves the complexity of sorting to $O(n \log n)$. You can explore this algorithm in a challenge problem below.

3 Big-Oh practice

Exercise. How would you order the following functions in increasing order of complexity?

$$T_1(n) = 20n \log n$$
$$T_2(n) = (n-1)^2$$
$$T_3(n) = 2^n$$
$$T_4(n) = \log n^4$$

The following exercises are intended for you to both practice reading Python code, and developing a sense for the big-Oh complexity of different loops. It may be helpful to run the code in Spyder for different inputs.

What do the following code snippets do? What is their big-Oh complexity in terms of the size of the input (eg a list length)?

```
def lsum(L):
    """ assumes L is list of length n """
    listSum = 0
    for item in L:
```

```
        listSum =  listSum + item
    print(listSum)
```

```
def addOne(L):
    """ assumes L is list of length n """
    for index in range(len(L)):
        L[index] = L[index] + 1
    print(L)
```

```
def mult(L1,L2):
    """ assumes L1,L2 are lists of length n,m """
    s = 0
    for i in L1:
        for j in L2:
            s = s + i*j
    print(s)
```

```
def testSubset(x,y):
    """ assumes x and y are lists of length n,m"""
    for a in x:
        found = False
        for b in y:
            if a == b:
                found = True
                break
        if not found:
            return False
```

```
    return True
```

The following exercises are more challenging...

```
def intToStr(i):
    """ assumes i is integer """
    digits = '0123456789'
    if i==0:
        return '0'
    result = ''
    while i>0:
        result = digits[i%10] + result
        i = i//10
    return result
```

```
def allIntsToStr(A):
    """ assumes A is a list of integers """
    r = []
    for i in A:
        r.append(intToStr(i))
    return r
```

```
def F(n):
    """ assume n is a nonnegative integer """
    y = 0
    i = n
    while  i > 0:
        j = 0
        while j < i:
```

```
            y += 1
            j += 1
        i -= 1
    return y
```

4 Built-in Python functions for searching and sorting

Searching and sorting are common building blocks in more complex programs and their performance is therefore crucial. Whilst designing and writing these algorithms is both fun and useful for developing your algorithmic and coding skills, in practice we usually rely on Python's built-in methods.

Python gives you an easy method of checking whether an item can be found in a list through the keyword `in`.

```
A = [3,8,11,9,4]
4 in A
```

You can sort a list using `list.sort()` and create a sorted copy using `sorted`. If you want to change the order, you can simply reverse the list.

```
A.sort()
B = sorted(A)
B = sorted(A, reverse=True)
B.reverse()
help(list)
```

But how does Python do its sorting? It uses an algorithm called **timsort**, which has worst-case running time $O(n \log n)$ just like your merge sort, but is faster on average. In fact, it turns out that $O(n \log n)$ is a theoretical bound for how fast you can expect to sort in the worst case. Timsort's is faster on average because it exploits the fact that many lists have some order to begin with, and finding such ordered sequences saves time compared to straight-up sorting. However,

the cost is that timsort is conceptually quite a bit more complex than merge sort (just like merge sort is conceptually more complex than selection sort). If you want to understand a state-of-the art sorting algorithm, look up timsort on Wikipedia. While you're at it, there are many other well-known sorting algorithms that we don't go through here: for example insertion sort, bubble sort, quicksort, radix sort, just to name a few.

5 Challenge problem: Fortune 500

The file `fortune500.csv` contains a list of Fortune 500 companies every year from 1955 to 2005 and Fortune 1000 companies from 2006 to 2009.[1] The Fortune 500 list comprises the largest American companies by yearly revenue.

The file is comma-delimited, with the first line containing column titles (Year, Rank, Company, Revenue (in millions), Profit (in millions)). The subsequent lines contain this information for the companies in each year's list.

Use Python to read through the file and explore the questions below. The Python file provides two ways by which you could proceed: by methods you already know, or a different but convenient Python library. That is, you can either read the file into Python lists and work from there (using eg loops), or alternatively use the Python library `pandas`. Code for reading the the file into Python is provided for both options.

`pandas` is a convenient library for dealing with data in Python. We will cover it in more detail in later tutorials. For now, the code in the Python file explains some useful commands in dealing with data — you can find `pandas` help at http://pandas.pydata.org/pandas-docs/version/0.18.1/tutorials.html.

When reading data in Spyder, make sure your data is in your current working directory (visible in the Spyder toolbar).

Questions to explore:

1. What were the largest profits and revenues of a company, both overall and in each decade covered by the data? What were the largest losses?

2. Which companies have held the number one slot? Which companies have been on the list for the longest periods? Are there companies that have stayed on the list during the entire time?

[1] Cleaned data courtesy of Robert Sedgewick, Princeton University.

3. How has the average profit of the companies on the list developed over the years?

4. Visualize the position of different companies on the list in a plot. For example, how has the position of Apple and Kodak changed over the years? We will cover a Python plotting library, `matplotlib`, in a later tutorial. For now, you can find a help on plotting here: http://matplotlib.org/users/pyplot_tutorial.html.

6 Challenge problem: Merge sort

Although the selection sort algorithm is intuitive, it is not as fast as we would hope for: its running time is $O(n^2)$ for n items. We can do better by other algorithms, for example using merge sort, which is a recursive "divide-and-conquer" algorithm. It significantly improves the complexity of sorting to $O(n \log n)$.

The merge sort algorithm is given below. The idea bears some resemblance to binary search. You start with a list you want to sort. You identify a *base case* for which sorting is easy: if the length of a list is no greater than one, the list is obviously sorted. So you recursively divide the list roughly in half as many times as it requires to reach this base case - approximately $\log n$ times. You then combine these results together at each level of recursion and return the result. This combining is called *merging* lists, and the exact procedure is given in the algorithm below. As it turns out, each level of recursion requires $O(n)$ work. So merge sort in total has $O(n \log n)$ complexity.

Merge sort:

- Base case: if list length $n \leq 1$, the list is sorted: return the list itself
- Divide: if list length $n > 1$, split into two lists and recursively sort each
- Combine (merge) the two (sorted) lists:
  - Initialize an empty `result` list
  - Look at first element of each list, add smaller to end of `result`
  - When one of the lists becomes empty, copy the rest of the other list to `result`
- Return the result of merging

The Python file contains a full implementation of the `merge` subroutine, as well as a skeleton code for the `mergesort` routine which you'll have to complete. For an example of how merge sort works, see the extra slides in the lecture folder, or this wonderful video: https://www.youtube.com/watch?v=XaqR3G_NVoo.