Lab Questions: Lab Session 11

Deadline: 08.11.2017 11:59pm SGT

Complete all assignments below. For the question that is marked with an asterisk *, i.e. **Questions 5** and **7**, create the files as requested. Once you are done with it, submit the file(s) via `iNTU`. Remember to put plenty of comments for all assignments, as this helps us to better understand your program (which might give you higher marks).

Important!!! Make sure your scripts work properly, as we give 0 marks otherwise. Please name the files according to the requirements, and upload each file separately and not in a Zip file or similar. Check that your files properly have been uploaded to `iNTU`. The submission system closes at the deadline. Hence after that, you will get no marks for your solution.

1. Assume that you have access to a list of floats called `my_list`. Estimate the asymptotic time complexity of the following algorithms according to the number N of elements in `my_list`, using the big-O notation:

(a)
```
N = len(my_list)
for i in range(N):
    my_list[i] += 10
```

(b)
```
import random
N = len(my_list)
for i in range(100):
    my_list[N-1] += random.random()
```

(c)
```
import random
import numpy as np
N = len(my_list)
my_3D_list = np.zeros([N,N,N])
for i in range(N):
    my_list[0] += random.random()
    for j in range(N):
        for k in range(N):
            my_3D_list[i][j][k] = my_list[min([i,j,k])]
```

(d)
```
N = len(my_list)
for i in range(int(N/10)):
    my_list[i] += 10
```

(e)
```
my_list = [5, 2, 8, 7]
import random
N = len(my_list)
for i in range(N):
    for j in range(i):
        my_list[i] += random.randint(0,j)
```

2. Write a function that takes as input a list of floats `mylist` and that returns the average of all the floats of the list (do not use the built-in `sum` function). Estimate the asymptotic time complexity of your function relative to the number of list elements.

3. Write a function that takes as input two $(N \times N)$ square matrices M_1 and M_2 (implemented as two 2-dimensional NumPy arrays) and that returns another $(N \times N)$ square matrix M obtained by multiplying the two input matrices $M = M_1 * M_2$ (do not use the NumPy `dot` function of course, except for testing that your program indeed performs properly the matrix multiplication). Estimate the asymptotic time complexity of your function relative to the size N of the matrix.

4. The `sorted` function allows a more advanced sorting by providing an extra input to the function: one can specify what criterion should be used for comparison. This customization is done with the syntax "`key=my_function`" where `my_function` is a function that transforms each element of the list before comparison (the sorting will thus be done on the transformed data).

   For example, assume that we have a list of strings `mystr` that contains words written in upper-case or lower-case. We would like to sort these strings alphabetically, but without taking the upper-case/lower-case into account. Using the `sorted` function leads to a wrong ordering:

```
>>> mystr = ['Thomas', 'john', 'Jakob', 'Alex']
>>> sorted(mystr)
    ['Alex', 'Jakob', 'Thomas', 'john']
```

   However, if we use the built-in string method `str.lower` as key, we can pre-transform the strings in `mystr` into a lower-case form only, so that the comparison is properly done:

```
>>> sorted(mystr,key=str.lower)
    ['Alex', 'Jakob', 'john', 'Thomas']
```

   For the two following cases, write your own comparison function and use it as key in `sorted` (test on some `mystr` you would have generated beforehand):

   (a) sort the strings of the list alphabetically, but ignoring their first letter.

   (b) sort strings of the list according to their length.

5. * Write a function `insertion_sort` in a file `insertion_sort.py`, that will implement the insertion sorting algorithm. The principle of this sorting algorithm is simple: starting from a float list `inlist` to be sorted, the elements of `inlist` will be extracted one at a time, and placed into a new list `outlist` (originally empty) such that `outlist` always remain a sorted list. For example, using `inlist = [5 36 14 7.2]`, the algorithm would first extract 5 from `inlist` and place it in vector `outlist = [5]`. Then it would extract 36 and place it in `outlist = [5 36]`. Then it would extract 14 and place it in `outlist = [5 14 36]`. Finally, it would extract 7.2 and place it in `outlist = [5 7.2 14 36]`. The algorithm stops when all elements of `inlist` have been extracted.

6. Assume that you are given a list of floats `my_list` that is already sorted. We are interested in programming a function that searches if a certain element belongs to the list (of course, we assume that you can't use the built-in `in` operator).

   (a) Implement a simple function `search_element` that takes as inputs a sorted list and a float, and that will return a boolean value `True` if the float belongs to the list, `False` otherwise. The function will simply scan through each of the elements one at a time. What is the best/average/worst case asymptotic time complexity of this algorithm, relative to the input list size n ?

   (b) Implement the same functionality, but this time using the binary search strategy: in order to look for an element x in a sorted list L, just pick the entry located in the middle of L and compare it with x. If it is equal, you found x. If it is greater than x, then there is no chance that x can be in the upper part of L, so we only need to repeat the search in the lower part of L only. If it is smaller than x, then there is no chance that x can be in the lower part of L, so we only need to repeat the search in the upper part of L only.

       After implementing the binary search, analyse what is the best/average/worst case asymptotic time complexity of this algorithm, relative to the input list size n.

   (c) Measure and compare the efficiency of both functions using big lists as input.

7. * In a file `counting.py`, implement a function `counting` that takes as inputs a sorted list of integers and an integer, and that will return the number of times this integer is present in the list (of course do not use the build-in `count` method). The algorithm must have an average asymptotic time complexity of $O(log(n))$, where n represents the input list size (we assume that an element can only be present at maximum 10 times in the list). Hint: solve question 6 first.

8. In a file `merge_sort_iterative.py`, rewrite the merge sorting algorithm seen during the lecture, but in an iterative way (that is without using a recursive function). (hint: first consider all adjacent 1-element sublists and merge them together in a sorted 2-element sublist, then consider all adjacent 2-element sublists and merge them together in a sorted 4-element sublist, then consider all adjacent 4-element sublists and merge them together in a sorted 8-element sublist, ... until your sublist size reaches the original list size)