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
   ```

   Solution:
   $O(N)$ as we have only one for loop that goes through all the values in `my_list`.

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

   Solution:
   $O(1)$ as the loop does not depend on N. Thus, the time complexity is constant when N grows larger.

   (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])]
   ```

   Solution:
   $O(N^3)$ as we have three nested for loop that each go through all the values in $[0, N-1]$.

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

Solution:
$O(N)$ as we have only one for loop that goes through one tenth of the values in `my_list`. Since we are using big-O notation, the 1/10 factor can be ignored.

(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)
```

Solution:
The inner nested for loop uses the index of the outer for loop, thus we will be performing $1+2+3+4+\ldots+N-2 = ((N-2)^2+(N-2))/2 = (N-2)^2/2 + (N-2)/2 = O(N^2)$ computations.

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.

Solution:

average.py

```
def average(mylist):
    """ computes the average of the values contained
    in the input mylist."""

    # sum in here all the values in the list
    my_sum = 0
    for i in mylist:
        my_sum += i

    # eventually divide by the number of elements
    return my_sum/len(mylist)
```

The asymptotic time complexity is $O(N)$ since we loop once through all the elements of the list, one at a time.

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.

Solution:

matrix_multiply.py

```
import numpy as np

def matrix_multiply(matrix1,matrix2):
    """ Implements the matrix multiplication between the
        amtrix inputs matrix1 and matrix2. """

    # obtain the size of the matrices
    (matrix_size,matrix_size) = matrix1.shape

    # initialise the output_matric to zeros
    output_matrix = np.zeros([matrix_size,matrix_size])

    # for all element positions in the output matrix
    for col in range(matrix_size):
        for row in range(matrix_size):

            # compute the element to be placed in the output matrix
            my_sum = 0
            for i in range(matrix_size):
                my_sum += matrix1[row][i] * matrix2[i][col]
            output_matrix[row][col] = my_sum

    return output_matrix
```

The asymptotic time complexity is $O(N^3)$ since we have three nested for loops, each going through N elements.

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.
Solution:

```
>>> def my_cmp(input_str):  return input_str[1:]
>>> sorted(mystr,key=my_cmp)
    ['Jakob', 'Thomas', 'Alex', 'john']
```

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

```
>>> def my_cmp2(input_str):  return len(input_str)
>>> sorted(mystr,key=my_cmp2)
    ['john', 'Alex', 'Jakob', 'Thomas']
```

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.

Solution:

insertion_sort.py

```
def insertion_sort(inlist):
    """ Implements the insertion sort algorithm
    takes as input a (possibly) unsorted list
    and outputs the corresponding sorted list """

    outlist = [inlist[0]]

    # for all the elements in the input list
    for j in range(1,len(inlist)):

        # identify where this element needs to be placed
        # in the current sorted output list
        i=0
        while outlist[i] < inlist[j]:
            i += 1
            # go out of the loop in case you reached the end of the
                list
            if i==len(outlist):
                break
        outlist.insert(i,inlist[j])

    return outlist
```

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 ?

Solution:

simple_search.py

```
def simple_search(mylist,element):
    ''' This function performs a simple search of the input
        element
    over the input list mylist. It returns True if the element
        belongs
    to the list, False otherwise.
    '''

    # for all the entries of the list, we check
    # it is equal to element and return True if so
    for i in mylist:
        if i == element:
            return True

    return False
```

The best case complexity is $O(1)$ (the searched element can be the first one of the list), the worst case complexity is $O(n)$ (the searched element can be the last one of the list). The average case complexity is $O(n)$ as on average you will need $n/2$ comparisons before finding the element.

(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.

Solution:

binary_search.py

```
def binary_search(mylist,element):
    ''' This function performs a binary search of the input
        element
    over the input list mylist. It returns True if the element
        belongs
    to the list, False otherwise.
    '''

    if len(mylist) == 0:
```

```
        return False

    # if the input list has only a single entry,
    # we check if it is equal to element
    if len(mylist) == 1:
        if mylist[0] == element:
            return True
        else:
            return False

    # We check the middle entry of the list, and depending on
    # its value we divide the current list in two halves,
    # so as to only check the appropriate half
    mid = int(len(mylist)/2)
    if element < mylist[mid]:
        return binary_search(mylist[:mid],element)
    elif element > mylist[mid]:
        return binary_search(mylist[mid+1:],element)
    else:
        return True
```

The best case complexity is $O(1)$ (the searched element can be the first middle element picked). The worst and average case complexity is $O(log(n))$ as we will successively divide the list in 2 until we reach a 1-element list. Indeed, if we would like to reach 1-element list with x successive divisions in two, we have $1 = N/2^x$, thus $x = log_2(N) = log(N)/log(2) = O(log(N))$.

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

Solution:

measure.py

```
import time
import random
import simple_search
import binary_search

N = 1000000
my_list_original = [random.randint(0,2*N) for i in range(N)]
my_list_original.sort()
element = random.randint(0,2*N)

start = time.clock()
simple_search.simple_search(my_list_original,element)
print("The simple search time is: ",end='')
print(time.clock() - start)

start = time.clock()
binary_search.binary_search(my_list_original,element)
print("The binary search time is: ",end='')
print(time.clock() - start)
```

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.

Solution:

counting.py

```
def counting(mylist,element):
    ''' This function counts the number of times an element is present
        in the
    input list mylist. It also return a list of the indexes of these
       entries'''

    # if this is empty, then count is 0
    if len(mylist) == 0:
        return 0

    # initialise index values
    start = 0
    end = len(mylist)
    mid = int((end-start)/2)

    # continue searching for element in the list
    while True:
        # if we have to search in the low-half of the list
        if element < mylist[mid]:
            end = mid
            mid = start+int((end-start)/2)
        # if we have to search in the high-half of the list
        elif element > mylist[mid]:
            start = mid
            mid = start+int((end-start)/2)
        # if we have found the element in the list
        else:
            # we count how many times element is repeated
            # by checking adjacent positions in the list
            start = end = mid
            print(start,mid,end)
            while element == mylist[start]:
                start -= 1
                if start < 0: break
            while element == mylist[end]:
                end += 1
                if end >= len(mylist): break
            return end-start-1

        # if the element is not present in the list, we return 0
        if mid-start == 0 and element != mylist[mid]:
            return 0
```

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)

Solution:

merge_sort_iterative.py

```
def merge_sort_iterative(inlist):
    """ Implement the merge sort algorithm without
    using recursive function (aka iterative implementation)
    Takes as input a (possibly) unsorted list
    and outputs the corresponding sorted list """

    outlist = inlist[:]

    # we will test successive powers of two for sublist sizes
    # aka we start from 1, then 2, then 4, then 8, etc.
    # and we continue until the sublist size is bigger than the list
        itself
    sub_list_size = 1
    while sub_list_size < len(inlist):

        # we will test all adjacent sublists of size sub_list_size
        # and merge them together while keeping the property that they
            are sorted
        for start_index in range(0,len(inlist),2*sub_list_size):

            # we extract the two adjacent sublists (left and right)
            left_list = outlist[start_index:start_index+sub_list_size]
            right_list = outlist[start_index+sub_list_size:start_index
                +2*sub_list_size]

            # we merge these two lists together
            # while keeping the property that everythin is property
            my_sorted_list = []
            while left_list != [] or right_list != []:
                if left_list == []:
                    my_sorted_list.append(right_list.pop(0))
                elif right_list == []:
                    my_sorted_list.append(left_list.pop(0))
                elif left_list[0] < right_list[0]:
                    my_sorted_list.append(left_list.pop(0))
                else:
                    my_sorted_list.append(right_list.pop(0))
            outlist[start_index:start_index+2*sub_list_size]=
                my_sorted_list[:2*sub_list_size]

        sub_list_size *= 2
    return outlist
```