ITI 1120

Lab # 9

SEARCHING AND SORTING
(and a bit of exceptions handling):

Algorithms to solve them, their analysis/efficiency

Applications of searching and sorting

Starting Lab 9

- Open a browser and log into Brightspace.
- On the left hand side under Labs tab, find lab6 material contained in lab9-students.zip file.
- Download that file to the Desktop and unzip it.

Before starting, always make sure you are running Python 3

This slide is applicable to all labs, exercises, assignments … etc

ALWAYS MAKE SURE FIRST that you are running Python 3.4 (3.5 or 3.6 is fine too)

That is, when you click on IDLE (or start python any other way) look at the first line that the Python shell displays. It should say Python 3.4 or 3.5 or 3.6 (and then some extra digits)

If you do not know how to do this, read the material provided with Lab 1. It explains it step by step

Do all the exercises labeled as Task in your head i.e. on a paper

Later on if you wish, you can type them into a computer (or copy/paste from the solutions once I poste them)

Catching and Handling Exceptions

First, learn about catching and handling exceptions

- by reading the provided `exceptions-handling.pdf` file which contains the material following from one of your recommended textbooks (the one by Ljubomir Perkovic) and/or
- and/or by watching this short video: https://www.youtube.com/watch?v=evu5qFjpNJk&index=3&list=PLO9y7hOkmmSEqzlgHLJ0XpJTIBNceFQb
- and or by optionally reading: http://interactivepython.org/runestone/static/pythonds/Introduction/ExceptionHandling.html
- Finally to see an example of how to used this in bigger context and learn more about reading files, populating 2D list and making a bigger program, such as making a trivia quiz, some may find the following lecture I taught last year useful: https://youtu.be/wI9dEAyeAto (For those interested, the files from that lecture are also included in this lab, quiz.csv and quizgame.py) The first file contains a collection of trivia quiz questions and the second a program we developed in class last year that make a trivia game)

Programming exercise 0 (useful for your assignment 4):

Write a function called `get_year()` that has no input parameter and it returns an integer. The function prompts the user to enter 4 digit integer for a year. If the user enters 4 digit integer for a year, then the function returns that number as integer. Otherwise the function keeps on asking the user to reenter. Note that your function cannot crash if the user enters something like `"aha"`. Instead it should print a meaningful message, like "Please give a four digit integer for the year".

Test your function in python shell by calling it:

```
>>> get_year()
```

ANALASYS OF ALGORITHMS:

For the rest of this semester running time of a program/algorithm/function will mean the same thing as the number of operations of a program/algorithm/function. These two terms mean the same thing!! I will use them interchangeably. You textbook uses "running time"

Big O notation hides the constants and slower going terms of a function. For example:

$4n^2$ is $O(n^2)$, (and so is $n^2/100 + n$ for example)

$20n - 70$ is $O(n)$

$10\log_2 n + 100$ is $O(\log_2 n)$

Task 1:

For each of the following 7 functions answer the question about how many operations the functions does, roughly, as n grows. For example the answer is roughly linear in n for foo3, i.e O(n)

```
def foo1(n):
    i=1
    while i<n:
        i=i*2
        count=count+1

def foo2(n):
    i=n
    while i>=1:
        i=i//2

def foo3(n):
    for i in range(n):
        print("*")

def foo4(n):
    for i in range(-n,n,3):
        print("*")
```

```
def foo5(n):
    for i in range(n):
        for j in range(i+1,n):
            print("*")

def foo6(n):
    for i in range(n):
        for j in range(n):
            for k in range(n):
                print("*")
def foo7(n):
    for i in range(n):
        j=n
        while j>=1:
            j=j//2
```

Task 2: answer the 4 questions

```
def annona(L):
    count=0
    for i in range(len(L)):
        flag=True
        for j in range(len(L)):
            if L[i] == L[j] and i!=j:
                flag = False
        if flag:
            count=count+1
    return count

lst=[1, 0, 2, 0, 3]
result = annona(lst)
print(result)
```

1. What does the program on the left print?

2. Write one sentence that describes abstractly (in plain English) what the function annona does? (something a person with no programming knowledge can understand)

3. Suppose the function annona is called on an list that has 1000 integers. How many times would L[i]==L[j] test be executed?

4. How many operations (roughly) does function L do on a list of size n?

a) less than 20
b) Between 1000 and 9,000 times
c) Between 10,000 and 100, 000 times
d) Between 200, 000 and 2 million times
e) between 10 million and 200 million times
f) None of the above

PART 1: SEARCHING

Python's search

Python's "in" operator can be used to determine if a given element is in a given list. For example

```
>>> 10 in [1,20,-1,10,-5, 10]
True
```

Python's .index method can be used to determine if a given element is in a given list and if it is it returns its position, and otherwise -1

```
>>> L=[1,20,-1,10,-5,10]
>>> L.index(10)
3
>>> A=['d', 'a', 'b', 'a']
>>> A.index('a')
1
>>> A.index('c')
-1
```

Both in and .index methods perform linear search and thus take linear number of operations in the size of the list

Study: overview of Linear Search

Linear search starts at index 0 and looks at each item one by one. At each index, we ask this question: Is the value we are looking for at the current index?

IMPORTANT: Linear search is used to find an item in an UNSORTED list! It takes, roughly, linear number of operations (in the worst case, i.e. O(n)), where n is the size of the list.

Python's "in" and .index built-in functions perform linear search and thus they too take do O(n) operations.

Task 3: Linear Search

Open the file called linear_search-3-versions.py and study the 3 implementations of linear search. They are all correct and all do roughly n operations (i.e. O(n)) on lists of size n.

“Which one you prefer is largely a matter of taste: some programmers dislike returning in the middle of a loop, so they won’t like the second version. Others dislike modifying parameters in any way, so they won’t like the third version. Still others will dislike that extra check that happens in the first version. “

Interlude: some useful list methods to know

Here is some useful list methods that modify the given list lst:

`lst.append(item)` adds item to the end of `lst`
`lst.pop()` removes the last element from `lst`
`lst.pop(i)` removes `item` in position `i` in the `lst` and returns that `item`
`lst.insert(i, item)` inserts v before index `i` in `lst`

For more run in python shell:

```
help(list.append)
help(list.pop)
help(list.insert)
```

Programming exercise 1 and Task 4:

Prog ex 1:

- All three versions of linear search in linear_search-3-versions.py start at index 0. Rewrite all three to search from the end of the list instead of from the beginning. Make sure you test them.

Task 4:

- For the new versions of linear search: if you are looking for value v and it appears in the list more than once, which position will your modified linear searches find?

Programming ex 2: min or max and Task 5

Prog Ex 2:

•Write a function named min_or_max_index that has two parameters: one of type list and another type bool. If the Boolean parameter refers to True, the function returns a tuple containing the minimum and its index; and if it refers to False, it returns a tuple containing the maximum and its index.

(do not use python's min and max functions – roll your own implementation)

Task 5:

•On a list of size n, what is roughly the number of operations your program does in the worst case? (constant, linear in n, quadratic in n, log n ….?)

Study: overview of Binary Search and Task 6

IMPORTANT:

Binary search is used to find an item in an SORTED list!

It does, roughly, $\log_2 n$ of operations (in the worst case, i.e. $O(\log_2 n)$), where n the size of the list.

Task 6:

Open the file called binary_search.py. It contains the binary search version we developed in class and study again how it works.

Question: Binary search is significantly faster than the linear search but requires that the list is sorted. As you know, the number of operations for the best sorting algorithm is on the order of $n \log_2 n$, where n is the size of the list. If we search a lot of times on the same list of data, it makes sense to sort it once before doing the searching. Roughly how many times do we need to search in order to make sorting and then searching as fast or faster than linear search?

Programming exercise 3 (a bit of a challenge):

1. Write a function called first_one(L) that takes as input a list where each element of L is either a number 0 or 1. In addition all 0s appear before all 1s. Function first_one(L) should return the position in L of the first 1. If there is no 1s in the list, return -1.

Unless the list is very small (less than 3 elements) your solution should not access all the elements in the list. In particular if the list has 1000 elements you solution should access roughly 10 of them, if it has 100,000 elements it should access not more than 20. Roll your own implementation (only use loops and if statements). Basically it should run in O(log n) operations.

```
>>> first_one( [0,0,0,0,0,0,1,1] )
6
>>> first_one( [1,1] )
0
>>> first_one( [0,0,0] )
-1
```

2. Repeat the exercise except this time write a function called last_zero(L) that returns the position of the last 0. If helpful, you can make a call to your own function first_one(L)

Study Refined Binary Search

The version of binary search we developed in class returns SOME POSITION where value v is found in the given list L (and -1 if v is not in the list).

Open the file called binary_search_refined.py. It contains a refined binary search version that returns THE FIRST position the value v is found in L (and -1 if v is not in L)

1. Study and understand the solution
2. Think of how just one call to binary_search_refined.py would resolve the previous first_one problem (and different variants of such problem)

PART 2: SORTING

Task 7: Selection Sort

See file selection_sort.py to recall the selection sort algorithm we studied and developed in class.

- Given the unsorted list [6, 5, 4, 3, 7, 1, 2] write down what the contents of the list would be after each iteration (of the outer loop) as the list is sorted using the selection sort

6 Programming Exercises

Do first 3 questions (not including questions X again)

Recall from the analysis we did in class that selection sort does roughly n^2 operations (more precisely, O(n^2)) on a list of size n. As contrast, merge sort does at most $O(n \log_2 n)$ operations. You can think of python's sort as also doing at most $O(n \log_2 n)$ operations (although it is not exactly true).

Open file applications.py and program (and test) the 6 functions described there. All your solutions should perform

at most O(n log n) operations.

The only Python function you can use is ".sort" or "sorted" (since you know something about the number of operations they take and how they work roughly). You can also use .append and assume one append call takes constant number of operations.

DO NOT USE DICTIONARIES. DICTIONARIES are black boxes, for now. Their running time analysis you will only study in the 2nd year.

BONUS TASK 8: CHALLENGE ANALYSIS:

The function below sorts the given list L. Can you figure out how many operations it takes in the worst case? In other words is it better of worse or the same as as selection sort (selec. sort does $O(n^2)$ operations). What about merge sort? Think of L being a list where elements are ordered from biggest to smallest. That is the worst case for the below algorithm. You can assume that .append and .pop(0) take constant number of operations. min takes linear on a list of size n. You can find rough running times of Python's operations here: https://wiki.python.org/moin/TimeComplexity

```
def some_sort(L):
    '''(list)->list
    Returns a sorted version of the given list L
    '''
    sorted_list = []
    while L!=[]:
        if L[0] == min(L):
            sorted_list.append(L[0])
            L.pop(0)
        else:
            L.append(L[0])
            L.pop(0)
    return sorted_list
```

AFTERWARDS, AT HOME:

--- Study the insertion sort from the (Practical Programming) textbook. It is yet another algorithm that takes quadratic number of operations, i.e. $O(n^2)$, to sort. Try to figure out why in the worst case it takes quadratic number of operations?

--- Implement bubble sort

https://en.wikipedia.org/wiki/Bubble_sort

Bubble sort is also quadratic algorithm in the worst case