uc3m | Universidad Carlos III de Madrid

Data Structures and Algorithms.
Author: Isabel Segura Bedmar

Unit 3 – Analysis of Algorithms

Part A - Empirical Analysis

1. Write a function randomList(n,a,b) that returns a list of n random integers between a and b. The function can consider that the default values for a and b are 0 and 25, respectively. This functions will be used later.

Solution:

```
import random
def randomList(n,a=0,b=25):
  """Returns a list with n elements. The elements
  are randomly created and a list of size n with random integers"""
  l=[]
  for i in range(n):
    #creates a random number N between 0 and 50
    l.append(random.randint(a,b))
  return l
```

```
A=randomList(10)

print(A)
```

2. Write a function, called sumList(l), which takes a list of integers as parameter and returns the sum of all its elements. You must empirically study the time complexity of this function. To do this, you should include instructions to measure the running time. Then, you should run it for different size of lists (such as 10,100,1000,10000,10000,etc). Finally, plot its time complexity on list of different sizes. What does the running time depend on?

Notes:

- You should use the randomList function to generate lists of different sizes.
- When you plot the results, you should choose the logarithmic scale for the axes.

Solution:

```
def sumList(l):
  "Returns the sum of all the elements of the list"
  total=0
  for e in l:
    total +=e
  return e

print(A,sumList(A))

import time
start = time.time()
print(A,sumList(A))
end = time.time()
total=end-start
print('{} seconds'.format(total))

for e in range(1,7):
  n=10 ** e
  l=randomList(n)
  start = time.time()
  sumTotal=sumList(l)
  end=time.time()
  #Instead of showing seconds, we use nano seconds (1 sg = 10 ** 9 nano
seconds)
  total=int((end-start)*(10**9))
```

```
  #print('size={},time (ms)={}'.format(n,total))
  print('{},{}'.format(n,total))

"""**What does the running time depend on?**

You can see that the time complexity depends on the size of the list. The
more elements the list has, the longer the function takes.
```

3. Write a function, named search, which takes a Python list of integers and a number as parameters, and returns true if the element exists and false otherwise.

Solution:

```
def contains(l,x):
  for e in l:
    if e==x:
      return True
  return False

A=randomList(20,0,10)
print(A)
print(contains(A,5))

A.sort()
print(A)
```

4. Now suppose that the input list is always sorted in ascending order. Write a new version of the search function, named binary_search, which takes a sorted list and a number as parameters, and checks if the number exists into the list. In this function, you should take advantage of the list is sorted.

Solution:

```
def binary_search(data,value):
    "The list must be sorted"
    left= 0
    right = len(data)-1

    while left <= right:
        #mid = low + (high - low) // 2
        mid = (left + right)//2
        mid_value = l[mid]
        if mid_value == value:
            return True
        elif mid_value < value:
            left = mid + 1
```

```
        else:
            right = mid - 1
    return False


A.sort()
print(A)
print(binary_search(A,5))
```

5. Study (empirically) and compare the time complexity of the two previous functions (assume that the lists are sorted). To simplify, you can assume that the number to search is always 5. You can generate the lists with the function randomList and sort them using the Python function of lists, sort (for example, l.sort()). What function is more efficient?

Solution:

```
for e in range(1,8):
  n=10 ** e
  l=randomList(n)
  l.sort()

  start = time.time()
  contains(l,5)
  end=time.time()
  totalContains=int((end-start)*(10**9))

  start = time.time()
  binary_search(l,5)
  end=time.time()
  totalBinarySearch=int((end-start)*(10**9))

  #print('size={},time (ms)={}'.format(n,total))
  print('{},{},{}'.format(n,totalContains,totalBinarySearch))

"""** What function is more efficient?**

When the size rises, the running time grows much faster in the contains
method than in the binary search method. Therefore,  the binary search method
is more efficient than the contains method.
```

Part B - Theoretical Analysis

6. Calculate the running time function T(n) for the sumList function.

Solution:

```
T(n)=2n+2.
To obtain it **Big-O** fuction, you must consider:


1.   Find the fastest grow term: 2n
2.   Take out the coefficient: n

Therefore, the Big-O function for the sumList function is O(n) (linear time
or linear complexity)
"""

def sumList(l):
  total=0      #1
  for e in l:  #n
    total = total + e #2n
  return e     #1

#T(n) = 1 + 3*n + 1= 3n+2
```

7. Write a function, called *sumPair0*, which takes a Python list of integers, *data*, as a parameter and returns the number of pairs (i,j) such as *i!=j and data[i]+data[j]=0*. Calculate the running time function T(n).

Solution:

```
T(n)=3n^2+n+2. To obtain it Big-O fuction, you must consider:

1.   Find the fastest grow term: 3n^2
2.   Take out the coefficient: n^2


Therefore, the Big-O function for the **sumPair** function is O(n^2)
(quadratic time or quadratic complexity)
"""

def sumPair(data):
  numPairs=0                      #1

  for i in range(len(data)):    #n

    for j in range(len(data)):  #n

      if i!=j and data[i]+data[j]==0:  #1
        numPairs = numPairs+1           #1
```

```
  return numPairs                              #1

#T(n)=3n^2+n+2
```

8. Write a function, called *sumTriple0*, that takes a Python list of integers, data, as a parameter and returns the number of triples (i,j,k) such as *i!=j and i!=k and j!=k and* data[i]+data[j]]+data[k]=0. Calculate the running time function T(n).

Solution:

```
T(n)=3n^3+n^2+n+2. To obtain it Big-O function, you must consider:


1.   Find the fastest grow term: 3n^3
2.   Take out the coefficient: n^3


Therefore, the Big-O function for the sumPair function is O(n^3) (cubic time
or cubic complexity)
"""

def sumTriple(vector):
  numtriple=0                       #1
  for i in range(len(vector)):     #n

    #Tj(n)=n+n*Tk(n)=n+3n^2
    for j in range(len(vector)):  #n

      #Tk(n)=n+n*2=3n
      for k in range(len(vector)):  #n
        if i!=j!=k and i!=k and j!=k and vector[i]+vector[j]+vector[k]==0:
#1
          print(i,j,k)                           #1
          numtriple = numtriple+1                #1

  return numtriple                             #1

#T(n)= 2+ 3n*n*n + n*n + n= 3n^3+n^2+n+2
```