CSCI-SHU 210 Data Structures

Recitation2 Analysis of Algorithms

You have a series of tasks in front of you. Complete them! Everyone should code on their own computer, but you are encouraged to talk to others, and seek help from each other and from the Professor/TA/LAs. Please submit your work in NYU Classes Assignment Section (Recitation2).

Important for this week:

- Determine the tightest big O runtime for a given "iterative" code snippet;
- Code under big O runtime restrictions;
- Know what is space complexity;

Question 1 (Theory) - just making sure you understand the Big-O definition:

1: Prove that running time $T(n) = n^2 + 20n + 1$ is $O(n^2)$

When $n = 20$, $20n = n^2$, so $n^2 + 20n + 1 = 2n^2 + 1 < 3n^2$,

So,for $n >= 20$, $20n <= n^2$, $n^2 + 20n + 1 < 3n^2$, so we could say $T(n)$ is $O(n^2)$

2: Prove that running time $T(n) = n^2 + 20n + 1$ is not $O(n)$
Although when $n=1$, $T(n) = 22n$, but for $n > 1$, $n^2 > n$, $n^2 + 20n + 1 > 22n$.
Thus we could determine $T(n)$ is not $O(n)$.

Question 2 (Code snippet analysis):

Determine the tightest big O runtime for each of the following code fragment:

Fragment1:

```
1. def func1(N):
2.      for i in range(N):
3.          for   in range(N, 0, -2):
4.              print("hi")
```

Fragment2:

```
1. def func2(N):
2.      for i in range(N):
3.          for j in range(N, 0, -2):
4.              print("hi")
5.
6.      x = 0
7.      while x < N:
8.          x += 1
9.          print("hiii")
```

Fragment3:

```
1. def func3(N):
2.      i = 0
3.      while i < N:
4.          j = N
5.          while j > 0:
6.              j //= 2
7.              print("hi")
8.          i += 1
```

Fragment 1 tightest big O:

$O(N^2)$

Fragment 2 tightest big O:
$O(N^2)$

Fragment 3 tightest big O:
O(N*logN)

Question 3 (Concept):

You have an N-floor building and plenty of eggs. Suppose that an egg is broken if it is thrown from floor F or higher, and unhurt otherwise.

1. Describe a strategy to determine the value of F such that the number of throws is at most log N.

First we need to keep a record of the upper bound and the lower bound. Initially, the upper bound is N, the lower bound is 0. Every time we take the middle value inside the bound. If at the middle value the egg is broken, then we adjust the upper bound to (current middle value-1), then we take a second middle value by adding up the new lower and upper bound and divide by two; If the egg does not break at the middle point, we adjust the lower bound to the middle point. We could find the F floor by minimizing the range of the two bound.

2. Find a new strategy to reduce the number of throws to at most 2 log F.

We start by dropping the egg at 1, 2, 4 until 2^i floor, if the egg broke at 2^i floor, we could determine that 2^(i-1)<=F<=2^i, which take logF throws, then we do binary search between 2^(i-1) and 2^i, which takes another logF throws, thus we have 2logF throws.

Question 4 (Prime number):

A number is said to be prime if it is divisible by 1 and itself only, not by any third variable.

1. Divide N by every number from 2 to N - 1, if it is not divisible by any of them hence it

is a prime.

2. Instead of checking until N, we can check until $\sqrt{N}$ because a larger factor of N must be a multiple of smaller factor that has been already checked.

Starting point for question 4 is uploaded on NYU Classes. Implement algorithm 1 and 2 in python.

What is the runtime for algorithm 1?

O(N) What is the runtime for algorithm 2? O(N)

Question 5 (permutation):

Suppose you need to generate a *random* permutation from **0** to **N-1**. For example, **{4, 3, 1, 0, 2}** and **{3, 1, 4, 2, 0}** are legal permutations, but **{0, 4, 1, 2, 1}** is not, because one number **(1)** is duplicated and another **(3)** is missing. This routine is often used in simulation of algorithms. We assume the existence of a random number generator, **r**, with method **randInt(i,j)**, that generates integers between **i** and **j** with equal probability. Here are three algorithms:

1. Create a size **N** empty array. **(array = [None] * N)** Fill the array **a** from **a[0]** to **a[N-1]** as follows: To fill **a[i]**, generate random numbers until you get one that is not already in **a[0], a[1], . . . , a[i-1]**.

2. Same as algorithm (1), but keep an extra array called the **used** array. When a random number, **ran**, is first put in the array **a**, set **used[ran] = true**. This means that when filling **a[i]** with a random number, you can test in one step to see whether the random number has been used, instead of the (possibly) **i** steps in the first algorithm.

3. Fill the array such that **a[i] = i**. Then:

```
for i in range(len(array)):

    swap( a[ i ], a[ randint( 0, i ) ] );
```

Starting point for question 5 is also uploaded on NYU Classes. Implement algorithm 1, 2 and 3 in python.

What is the expected runtime for algorithm 1?
$O(N^2)$
What is the expected runtime for algorithm 2?
O(N)
What is the expected runtime for algorithm 3?
O(N)

4. Plot the runtime of algorithm 1, 2, 3 using the given plot_data( ) function.
What are your observations?