Analysis of Assignment 5 part 1 Algorithms

Sahil Tara

December 6, 2017

1 largest34

For this problem I wrote the following code:

```
1 def largest_34(a):
2     """(list of numbers) -> number
3         Returns the sum of the 3rd and 4th largest number
4         Preconditions: len(a) >= 4
5     """
6     first = float("-inf")#lowest possible numbers so will get replaced with larger
7     second = float("-inf")
8     third = float("-inf")
9     fourth = float("-inf")
10    for num in a:
11        if num >= first:
12            fourth = third
13            third = second
14            second = first
15            first = num
16        elif num >= second:
17            fourth = third
18            third = second
19            second = num
20        elif num >= third:
21            fourth = third
22            third = num
23        elif num >= fourth:
24            fourth = num
25    return third + fourth
```

Note Line 6 and onward are the only bits of actual code running.
Consider them as Lines 1, 2, ... and 20 in the program to keep things simple
(we will ignore non-code lines in all examples following this too).
The number of operations could be then described with the following equation:
$T_n = T_1 + T_2 + T_3 + T_4 ... T_{20}$
Where T_n is the total number of the operations, and T_1, T_2, $...T_{20}$.
The first 4 lines are assignment operators which all take constant time, therefore $T_1 = 1$, $T_2 = 1$, $T_3 = 1$, $T_4 = 1$.
Line 5's loop check will run about n times where n denotes the length of the list, so $T_5 = n$.
Line 6- 19 execute all execute less than or equal to n times since they are contained within the for loop.

$T_6 \leq n$, $T_7 \leq n$, $T_8 \leq n$, $...T_{19} \leq n$.
Note adding each of the ns to the total number of operations is not exactly correct as not all of them will run n times but doesn't matter really, since the big O notation gets rid of constants and such. Finally, the last line is constant so $T_{20} = 1$.
Giving us the following equation for the number of operations.
$T_n \leq 1+1+1+1+n+n+n+n+n+n+n+n+n+n+n+n+n+n+n+1$
We can write this in big O notation as the following: $\mathcal{O}(n)$
So, the time complexity of 1 a) $\mathcal{O}(n)$. Meaning it runs in linear time.

2 largest_third

For this problem I wrote the following code:

```python
def largest_third(a):
    """(list of numbers) -> number
        Returns the sum of the largest third of numbers
        Preconditions: len(a) >= 3
    """
    tmp = sorted(a, reverse=True)
    tmp = tmp[:len(a)//3]
    summation = 0
    for i in tmp:
        summation += i
    return summation
```

This algorithm's number of operations can be described as:
$T_n = T_1 + T_2 + T_3 + T_4 + T_5 + T_6$
Where T_n is the total number of the operations, and $T_1, T_2, T_3, T_4, T_5, T_6$ are operations on their respective lines.
Now we know Python's inbuilt sort does approximately $nlogn$ operations, so we can say $T_1 \approx nlogn$.
Slicing in python is $\mathcal{O}(k)$ but we are doing a slice of one third of a list of length n $T_2 = \frac{n}{3}$.
Initializing a variable is constant so $T_3 = 1$
The for loop test line executes n times so $T_4 = n$
The summation line also executes n times since it is in the for loop so $T_5 = n$
Returning is constant so $T_6 = 1$
We now have $T_n \approx nlogn + \frac{n}{3} + 1 + n + n + 1$. We can write this in big O notation as the following: $\mathcal{O}(nlogn)$
So, the time complexity of 1 b) is $\mathcal{O}(nlogn)$.

3 third_at_least

For this problem I wrote the following code:

```python
def third_at_least(a):
    """(list of numbers) -> number or None
        Returns the value that occurs len(a)//3+1 times, if none
        exist returns None. If more than one exists returns smaller
        value
        Preconditions: len(a) >= 4
    """
```

```
8
9       tmp = sorted(a)
10      third_of_a = len(a)//3
11      third = None
12      for i in range(len(tmp) - third_of_a):
13          if tmp[i] == tmp[i+third_of_a]:
14              if third == None:
15                  third = tmp[i]
16              elif tmp[i] < third:
17                  third = tmp[i]
18      return third
```

This algorithm's number of operations can be described as:
$T_n = T_1 + T_2 + T_3 + T_4 + T_5 + T_6 + T_7 + T_8 + T_9 + T_{10}$
Where T_n is the total number of the operations, and $T_1, T_2, T_3, T_4, T_5, T_6, T_7, T_8, T_9, T_{10}$ are operations on their respective lines.
Now we know Python's inbuilt sort does approximately $nlogn$ operations, so we can say $T_1 \approx nlogn$.
Line 2 and Line 3 are constant due to len(a) being executed in constant time in python so $T_2 = 1, T_3 = 1$
Line 4 executes $\frac{2n}{3}$ times so $T_4 = \frac{2n}{3}$
All the Lines inside the for loop are constant, but are executed $\frac{2n}{3}$ times due to being inside of the for loop.
So, $T_5 = \frac{2n}{3}, T_6 = \frac{2n}{3}, T_7 = \frac{2n}{3}, T_8 = \frac{2n}{3}, T_9 = \frac{2n}{3}$
Returning is constant so $T_{10} = 1$
We now have $T_n \approx nlogn + 1 + 1 + \frac{2n}{3} + \frac{2n}{3} + \frac{2n}{3} + \frac{2n}{3} + \frac{2n}{3} + \frac{2n}{3} + 1$. We can write this in big O notation as the following: $\mathcal{O}(nlogn)$
So, the time complexity of 1 c) is $\mathcal{O}(nlogn)$.

4 sum_tri

For this problem I wrote the following code:

```
def sum_tri(a, x):
    """(list of numbers, number) -> bool
        Returns True if a value a[i] + a[j] + a[k] = x exists.
        Otherwise returns False.
    """
    tmp = sorted(a)
    for i in range(len(tmp)):
        j = i
        k = len(tmp) -1
        while j <= k:
            test_sum = tmp[i]+tmp[j]+tmp[k]
            if test_sum == x:
                return True
            elif test_sum > x:
                k -= 1
            else:
                j += 1
    return False
```

Finally, this algorithm's number of operations can be described as:
$T_n = T_1 + T_2 + T_3 + T_4 + T_5 + T_6 + T_7 + T_8 + T_9 + T_{10} + T_{11} + T_{12} + T_{13}$

Where T_n is the total number of the operations, and $T_1, T_2, T_3, T_4, T_5, T_6, T_7, T_8, T_9, T_{10}, T_{11}, T_{12}, T_{13}$ are operations on their respective lines.
Now we know Python's inbuilt sort does approximately $nlogn$ operations, so we can say $T_1 \approx nlogn$.
Line 2 executes n times, where $n = len(a)$ so $T_2 = n$
Line 3 and line 4 execute $constant * n$ times so $T_3 = n, T_4 = n$
Now here's where things get expensive.
Line 5 will execute at most $n * n$ times i.e if the sum isn't possible as k will approach j over n elements n times.
Therefore, $T_5 = n^2$
Line 6, 7, 9, 10 and 11, 12 will run n^2 times at most so $T_6 = n^2, T_7 = n^2, T_9 = n^2, T_{10} = n^2, T_{11} = n^2, T_{12} = n^2$. Note this is not exactly correct since we know that there is no way that they will all execute n^2 times, but it will not affect our analysis, since we will turn it into big O anyways.
Line 8 will execute at most once so $T_8 = 1$
Returning is constant so $T_{13} \leq 1$
Either T_{13} or T_8 will execute not both, so I'll exclude T_8 from the calculation considering the worst possible case.
This is allowed since we get rid of constants anyways in big O.
We now have $T_n <\approx nlogn + n + n + n + n^2 + n^2 + n^2 + n^2 + n^2 + n^2 + n^2 + 1$. We can write this in big O notation as the following: $\mathcal{O}(n^2)$
So, the time complexity of 1 d) is $\mathcal{O}(n^2)$.