Brief Time Complexity Review

Cpt S 321

Washington State University

What is the (worst case) Time Complexity?

```
int CountUnique(int[] nums)
{
    List<int> unique = new List<int>();
    foreach (int num in nums)
    {
            if(!unique.Contains(num))
            {
                unique.Add(num);
            }
    }
    return unique.Count;
}
```

Remember the Time Complexity Analysis Procedure

1. Go through each individual line of code in the algorithm and label its time complexity

2. Go through each line and count the number of times it's executed

3. Sum up (time_taken * times_executed) for all lines and you have a function of n that will tell you your time complexity.

Step 1: Label each line with the time complexity needed to execute it once

```
int CountUnique(int[] nums) // n = nums.Length
{
    List<int> unique = new List<int>(); // O(1)
    foreach (int num in nums) // O(1) to execute this line once
    {
            if(!unique.Contains(num)) // O(m), where m=unique.Count
            {
                unique.Add(num); // O(1)
            }
    }
    return unique.Count; // O(1)
}
```

Before we get to step 2...

- If we need to label each line with the number of times it gets executed, then we may run into a problem if a line can be executed a variable number of times depending on some condition

Before we get to step 2...

```
int CountUnique(int[] nums)
{
    List<int> unique = new List<int>();
    foreach (int num in nums)
    {
            if(!unique.Contains(num))
            {
                unique.Add(num); // Executes a variable number of times
            }
    }
    return unique.Count;
}
```

Best/Average/Worst Case Scenarios

- For such conditions you need to choose a particular scenario where you can determine the exact number of times the line will execute.

- For this case we were already told that we want worst case scenario

What is the Worst Case Scenario for this Algorithm?

```
int CountUnique(int[] nums)
{
    List<int> unique = new List<int>();
    foreach (int num in nums)
    {
            if(!unique.Contains(num))
            {
                unique.Add(num); // Executes a variable number of times
            }
    }
    return unique.Count;
}
```

What is the Worst Case Scenario for this Algorithm?

The worst case scenario is that the condition:

unique.Contains(num)

is always false, causing it to execute the line:

unique.Add(num);

for every single iteration in the loop (n times). This happens when the array of numbers has no duplicate values. With this knowledge we can finally complete step 2 and label each line with the number of times it is executed.

Step 2: Label each line with the number of times it is executed

```
int CountUnique(int[] nums) // n = nums.Length
{
    List<int> unique = new List<int>(); // 1 time
    foreach (int num in nums) // n times
    {
            if(!unique.Contains(num)) // n times
            {
                unique.Add(num); // n times in worst case scenario
            }
    }
    return unique.Count; // 1 time
}
```

Step 2.5: Combine the Two

```
int CountUnique(int[] nums) // n = nums.Length
{
    List<int> unique = new List<int>(); // 1 time @ O(1) each time
    foreach (int num in nums) // n times @ O(1) each time
    {
        if(!unique.Contains(num)) // n times @ O(m) each time
        {
            unique.Add(num); // n times @ O(1) each time
        }
    }
    return unique.Count; // 1 time @ O(1) each time
}
```

Step 3

// 1 time @ O(1) each time

// n times @ O(1) each time

// n times @ O(m) each time

// n times @ O(1) each time

// 1 time @ O(1) each time

1*1 + n*1 + n*m + n*1 + 1*1 =

1+ n + n*m + n + 1 =

n*m + 2n + 2

But we need everything in terms of n, so how do we describe the n*m part in terms of n?

n*m

We've noted that m refers to the number of items in the unique list. We've also stated that the array of numbers has no duplicates, meaning the size of the unique list increases by 1 for each of the n loop iterations.

On the first iteration, m=0

On the second, m=1

On the third, m=2

…

On the nth, m=n-1

n*m

This means we can identify this sequence as:

0 + 1 + 2 + … + n-2 + n-1

This sums to approximately:

n*(n-1)/2

or equivalently

$(1/2)*(n^2-n)$

Thus, it's an $O(n^2)$ operation

Finalizing Step 3

// 1 time @ O(1) each time

// n times @ O(1) each time

// n times @ O(m) each time = (1/2)*(n^2-n)

// n times @ O(1) each time

// 1 time @ O(1) each time

1*1 + n*1 + (1/2)*(n^2-n) + n*1 + 1*1 =

1+ n + (1/2)*(n^2-n) + n + 1 =

(1/2)*(n^2-n) + 2n + 2 =

(1/2)*n^2 + 1.5*n + 2

Therefore, the time complexity of the algorithm is O(n^2)