CMSC 351: Rigorous Time Summary

Justin Wyss-Gallifent

September 7, 2023

1 Introduction

Exactly how to measure the time requirements of an algorithm is often a source of confusion so here's a brief but hopefully comprehensive explanation.

2 Time Complexity

When we discuss time complexity of code in the simplest case the code will depend upon some n which could be the length of a list, the number of times a loop iterates, etc. Our goal is to imagine that we could write down a function $T(n)$ which tells us how much time the code takes for any given n and then find a simple $f(n)$ so that $T(n) = \Theta(f(n))$ when possible, and $\mathcal{O}$ or Ω perhaps when that's all we need or all we can get.

3 Assignments Take Time

Assignments take time.

Formally speaking each assignment takes its own time:

```
a = 0   c1
b = 0   c2
c = 0   c3
```

The total time is $c_1 + c_2 + c_3 = \Theta(1)$.

Of course we could argue that each assignment takes the same amount of time. Whether this is actually true or not might be system/hardware/implementation dependent but for time complexity it doesn't matter.

```
a = 0   c1
b = 0   c1
c = 0   c1
```

The total time is $3c_1 = \Theta(1)$.

Of course in light of that we could just lump it all together and suggest that all three lines together take some constant time:

```
a = 0   }
b = 0   } c1
c = 0   }
```

The total time is $c_1 = \Theta(1)$.

And even more informally we might just say that the actual constant doesn't matter so we'll call it time 1:

```
a = 0   }
b = 0   } 1
c = 0   }
```

The total time is $1 = \Theta(1)$.

Note 3.0.1. We're not saying some are right and some are wrong, it's all a question of what we're measuring and how much detail we desire.

4 For Loops Take Time

Formally speaking everything takes time, including the maintenance associated to loops. Consider this pseudocode, assuming `n` is given:

```
sum = 0          c1
for i = 1 to n   c2 a total of n times (this is the maintenance line)
    // Comment   0 a total of n times
end
```

We have an initial time of c_1 and then technically speaking the `for` loop does n assignments at some time c_2. Note that this is not the body of the `for` statement but rather this is the overhead consisting of the maintenance of the loop; assigning `i`, updating, and so on.

The body of the loop is a comment which takes 0 time.

The total time is $c_1 + c_2 n + (0)n = \Theta(n)$.

Now then, suppose we add something inside the loop:

```
sum = 0             c1
for i = 1 to n      c2 a total of n times
    sum = sum + i   c3 a total of n times
end
```

Now we have time cost $c_1 + c_2 n + c_3 n = \Theta(n)$.

However here is when computer scientists get understandably sloppy. Since the maintenance is just adding constant time c_2 to the body, and since constant time doesn't alter time complexity, generally we will ignore the contribution of the maintenance line.

```
sum = 0             c1
for i = 1 to n      Iterates n times
    sum = sum + i   c3 a total of n times
end
```

The total time is $c_1 + c_3 n = \Theta(n)$.

Of course this is no longer entirely accurate if we're assuming that the maintenance line takes time but when we're analyzing time complexity it's okay. If we're doing explicit time totals and if we wish to factor in that maintenance line time then it's not and we need to keep the c_2 in there.

In addition since the `for` loop will take some time the assignment before it, contributing constant time, could even be ignored:

```
sum = 0              Meh, contributes constant time overall
for i = 1 to n       Iterates n times
    sum = sum + i    c3 a total of n times
end
```

The total time is $c_3 n = \Theta(n)$.

Note 4.0.1. We're not saying some are right and some are wrong, it's all a question of what we're measuring and how much detail we desire.

5 While Loops Take Time

The same is true for `while` loops. Consider this pseudocode:

```
sum = 1          c1
i = 1            c2
while i <= n     c3 a total of n times (this is the maintenance line)
    i = i + 1    c4 a total of n times
end
```

The total time is $c_1 + c_2 + c_3 n + c_4 n = \Theta(n)$.

Again we can be a bit more sloppy, letting the `while` maintenance fold into the body of the loop and assuming all assignments take the same time:

```
sum = 1          c1
i = 1            c1
while i <= n     Iterates n times
    i = i + 1    c1 a total of n times
end
```

The total time is $c_1 + c_1 + n(c_1) = \Theta(n)$.

Or even more sloppy:

```
sum = 1          }
i = 1            } c1
while i <= n     Iterates n times
    i = i + 1    c1 a total of n times
end
```

The total time is $c_1 + n(c_1) = \Theta(n)$.

Or even more:

```
sum = 1          Meh
i = 1            Meh
while i <= n     Iterates n times
    i = i + 1    c1 a total of n times
end
```

The total time is $n(c_1) = \Theta(n)$.

Note 5.0.1. We're not saying some are right and some are wrong, it's all a question of what we're measuring and how much detail we desire.

6 Conditionals Take Time

The comparisons in a conditional also formally take time. Consider this pseucode, where `a` and `b` are assumed to be given.

```
if a<b          c1
    print('hi')  c2
end
```

Formally speaking the time this requires is:

- If `a<b` passes then the time is $c_1 + c_2$.
- If `a<b` fails then the total time is c_1.

Often then we'll just say the whole thing is constant time:

```
if a<b              }
    print('hi')     } c1
end                 }
```

However this doesn't mean we can just ignore the comparison all the time, it depends in delicate situations on the body of the conditional.

```
if a<b        c1
    UNKNOWN   Without understanding this, can't get rid of c1
end
```

In the next section we'll see what happens when we start nesting things.

Note 6.0.1. We're not saying some are right and some are wrong, it's all a question of what we're measuring and how much detail we desire.

7 Combining

Combining is when we need to be careful and attentive especially when we want to be sloppy and especially with regards to conditionals which contain a body operating at non-constant time.

Consider this pseucode:

```
if a<b                  c1
    sum = 0             c2
    for i = 1 to n      c3 a total of n times
        sum = sum + i   c4 a total of n times
    end
end
```

Formally the conditional check is c_1 time no matter what and the assignment is c_2. But then observe:

- If `a<b` passes then the total time is $c_1 + c_2 + c_3 n + c_4 n = \Theta(n)$.
- If `a<b` fails then the total time is $c_1 = \Theta(1)$.

In this case we can't blindly ignore the c_1 because if we did, then if `a<b` fails, then we'd be suggesting that this pseudocode takes 0 time.

Of course if this conditional had more stuff before (or after) it:

```
print('hi')              c5
if a<b                   c1
    sum = 0              c2
    for i = 1 to n       c3 a total of n times
        sum = sum + i    c4 a total of n times
    end
end
```

The `print` statement is contributing constant time and so removing the c_1 does not result in time 0 when `a<b` fails and therefore does not affect time complexity.

8 Best and Worst-Case

In the previous example:

```
print('hi')              c5
if a<b                   c1
    sum = 0              c2
    for i = 1 to n       c3 a total of n times
        sum = sum + i    c4 a total of n times
    end
end
```

Many resources (all over the internet) would simply say that this is $\mathcal{O}(n)$ but the truth is a bit more nuanced. The reality is:

- In a best-case scenario `a<b` fails and the time is $c_5 + c_1$ which is in fact $\Theta(1)$. It's also $\mathcal{O}(1)$ and $\Omega(1)$. Of course it is also $\mathcal{O}(n)$ but this is being pretty liberal.
- In a worst-case scenario `a<b` passes and the time is $c_5 + c_1 + c_2 + c_3 n + c_4 n$ which is in fact $\Theta(n)$. It's also $\mathcal{O}(n)$ and $\Omega(n)$.
- If anyone says casually that this is $\mathcal{O}(n)$ what they mean is that in the worst case it is $\mathcal{O}(n)$.

9 Summary on Time Consideration

So how far can we actually simplify if we're interested only in time complexity? In other words, what do we need to keep?

- With loops, provided that the body is guaranteed to take nonzero time we can ignore the maintenance line.

Example 9.1. In this example:

```
for i = 1 to n^2    c1 iterates n^2 times
    print(i)        c2 iterates n^2 times
end
```

We can ignore the c_1:

```
for i = 1 to n^2
    print(i)        c2 iterates n^2 times
end
```

The (best, worst, and average) time complexity is $\Theta(n^2)$:

- Any line that takes constant time can be ignored provided there is other adjacent code which takes nonzero time.

Example 9.2. In this example:

```
stuff = 1           c1
for i = 1 to n^2    c2 iterates n^2 times
    print(i)        c3 iterates n^2 times
end
```

We can ignore the c_2 as before and in addition the c_1:

```
stuff = 1
for i = 1 to n^2
    print(i)        c3 iterates n^2 times
end
```

The (best, worst, and average) time complexity is $\Theta(n^2)$:

- In a worst-case scenario a conditional is assumed to be true and the entire conditional can be replaced by the time that the body takes.

Example 9.3. In this example:

```
if a+b<c                c1
    for i = 1 to n      c2 iterates n times
        print('spicy')  c2 iterates n times
    end
end
```

For worst-case time complexity we can ignore the c_2 as noted earlier and in addition we can ignore the c_1:

```
if a+b<c
    for i = 1 to n
        print('spicy')  c2 iterates n times
    end
end
```

The worst-case time complexity is $\Theta(n)$.

- In a best-case scenario if we can guarantee that there is an input for which the conditional fails then the entire conditional is constant time for the conditional check so we can't remove the conditional check carelessly.

  Example 9.4. In this example:

  ```
  if a+b<c                 c1
      for i = 1 to n       c2 iterates n times
          print('spicy')   c3 iterates n times
      end
  end
  ```

  For best-case time complexity if we know for sure that there can be `a`, `b`, and `c` in some input for which `a+b<c` fails then we cannot ignore the c_1 but we can ignore everythign else:

  ```
  if a+b<c                 c1
      for i = 1 to n
          print('spicy')
      end
  end
  ```

  The best-case time complexity is $\Theta(1)$.

  However if another command takes care of that, then we can:

  Example 9.5. In this example:

  ```
  print('Hi')              c4
  if a+b<c                 c1
      for i = 1 to n       c2 iterates n times
          print('spicy')   c3 iterates n times
      end
  end
  ```

  For best-case time complexity we can ignore the c_2 and either the c_1 or the c_4:

  ```
  print('Hi')              c4
  if a+b<c
      for i = 1 to n
          print('spicy')
      end
  end
  ```

  The best-case time complexity is $\Theta(1)$.

- In an average-case scenario it's much more delicate.

  Example 9.6. Consider this example for average-case time complexity:

```
if a+b<c                 c1
    for i = 1 to n       c2 iterates n times
        print('spicy')   c3 iterates n times
    end
end
```

we can ignore the c_2 as noted earlier, thus we can see this as:

```
if a+b<c                 c1
    for i = 1 to n
        print('spicy')   c3 iterates n times
    end
end
```

Suppose the input is such that half the time `a+b<c` passes and half the time it fails. When it fails the total time is c_1 and when it passes the total time is $c_1 + c_3 n$ so the average total time is $c_1 + \frac{1}{2} c_3 n$ which is $\Theta(n)$.