CMSC 351: Rigorous Time Summary

Justin Wyss-Gallifent

May 25, 2023

1 Introduction ................................................. 2
2 Assignments Take Time ................................. 2
3 For Loops Take Time ...................................... 3
4 While Loops Take Time ................................. 5
5 Conditionals Take Time ................................. 6
6 Combining .................................................. 7
7 Best and Worst-Case ................................. 8
8 Summary on Time Consideration ..................... 9
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 Assignments Take Time

Assignments take time.

Formally speaking each assignment takes its own time:

\[
\begin{align*}
    a &= 0 \quad c_1 \\
    b &= 0 \quad c_2 \\
    c &= 0 \quad c_3
\end{align*}
\]

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.

\[
\begin{align*}
    a &= 0 \quad c_1 \\
    b &= 0 \quad c_1 \\
    c &= 0 \quad c_1
\end{align*}
\]

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:

\[
\begin{align*}
    a &= 0 \\
    b &= 0 \\
    c &= 0
\end{align*}
\]

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:

\[
\begin{align*}
    a &= 0 \\
    b &= 0 \\
    c &= 0
\end{align*}
\]

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

Note 2.0.1. 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.
3 For Loops Take Time

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

\[
\begin{align*}
\text{sum} &= 0 \quad c_1 \\
\text{for } i &= 1 \text{ to } n \quad c_2 \text{ a total of } n \text{ times (this is the maintenance line)} \\
&\quad // \text{ Comment } 0 \text{ a total of } n \text{ times} \\
\end{align*}
\]

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:

\[
\begin{align*}
\text{sum} &= 0 \quad c_1 \\
\text{for } i &= 1 \text{ to } n \quad c_2 \text{ a total of } n \text{ times} \\
&\quad \text{sum} = \text{sum} + i \quad c_3 \text{ a total of } n \text{ times} \\
\end{align*}
\]

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.

\[
\begin{align*}
\text{sum} &= 0 \quad c_1 \\
\text{for } i &= 1 \text{ to } n \quad \text{Iterates } n \text{ times} \\
&\quad \text{sum} = \text{sum} + i \quad c_3 \text{ a total of } n \text{ times} \\
\end{align*}
\]

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:

\[
\begin{align*}
\text{sum} &= 0 \quad \text{Meh, contributes constant time overall} \\
\text{for } i &= 1 \text{ to } n \quad \text{Iterates } n \text{ times} \\
&\quad \text{sum} = \text{sum} + i \quad c_3 \text{ a total of } n \text{ times} \\
\end{align*}
\]

The total time is \( c_3 n = \Theta(n) \).
**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 While Loops Take Time

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

\begin{verbatim}
sum = 1  \hspace{1cm} c_1
i = 1  \hspace{1cm} c_2
while i <= n  \hspace{1cm} c_3 \text{ a total of } n \text{ times (this is the maintenance line)}
    i = i + 1  \hspace{1cm} c_4 \text{ a total of } n \text{ times}
end
\end{verbatim}

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:

\begin{verbatim}
sum = 1  \hspace{1cm} c_1
i = 1  \hspace{1cm} c_1
while i <= n  \hspace{1cm} \text{Iterates } n \text{ times}
    i = i + 1  \hspace{1cm} c_1 \text{ a total of } n \text{ times}
end
\end{verbatim}

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

Or even more sloppy:

\begin{verbatim}
sum = 1  \hspace{1cm} \{ c_1
i = 1
while i <= n  \hspace{1cm} \text{Iterates } n \text{ times}
    i = i + 1  \hspace{1cm} c_1 \text{ a total of } n \text{ times}
end
\end{verbatim}

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

Or even more:

\begin{verbatim}
sum = 1  \hspace{1cm} \text{Meh}
i = 1  \hspace{1cm} \text{Meh}
while i <= n  \hspace{1cm} \text{Iterates } n \text{ times}
    i = i + 1  \hspace{1cm} c_1 \text{ a total of } n \text{ times}
end
\end{verbatim}

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

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 Conditionals Take Time

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

```plaintext
if a<b  
  c_1  
  print('hi')  c_2  
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:

```plaintext
if a<b 
  print('hi') 
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.

```plaintext
if a<b 
  UNKNOWN c_1 
  Without understanding this, can’t get rid of c_1 
end
```

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

**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 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 pseudocode:

```plaintext
if a<b
    sum = 0
    for i = 1 to n  a total of n times
        sum = sum + i  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 \(c_1 + c_2 + c_3n + c_4n = \Theta(n)\).
- 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:

```plaintext
print('hi')
if a<b
    sum = 0
    for i = 1 to n  a total of n times
        sum = sum + i  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.
7 Best and Worst-Case

In the previous example:

```python
print('hi')
if a < b
    sum = 0
    for i = 1 to n
        sum = sum + i
    end
end
```

Many resources (all over the internet) would simply say that this is $O(n)$ 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 $O(1)$ and $\Omega(1)$. Of course it is also $O(n)$ 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 $O(n)$ and $\Omega(n)$.

- If anyone says casually that this is $O(n)$ what they mean is that in the worst case it is $O(n)$. 
8 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 8.1.** In this example:

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

We can ignore the c1:

```plaintext
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 8.2.** In this example:

```plaintext
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 c2 as before and in addition the c1:

```plaintext
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 8.3.** In this example:

```plaintext
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 c2 as noted earlier
and in addition we can ignore the $c_1$:

```python
if a+b<c
    for i = 1 to n
        print('spicy')  # $c_2$ 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 8.4.** In this example:

```python
if a+b<c
    for i = 1 to n
        print('spicy')  # $c_2$ iterates $n$ times
    end  # $c_3$ iterates $n$ times
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 everything else:

```python
if a+b<c
    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 8.5.** In this example:

```python
print('Hi')  # $c_4$
if a+b<c
    for i = 1 to n
        print('spicy')  # $c_2$ iterates $n$ times
    end  # $c_3$ iterates $n$ times
end
```

For best-case time complexity we can ignore the $c_2$ and either the $c_1$ or the $c_4$:
print('Hi')
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 8.6.** Consider this example for average-case time complexity:

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

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

```
if a+b<c
    for i = 1 to n
        print('spicy')
    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 $\Theta(n)$. 