CMSC 351: Algorithm Time Complexity

Justin Wyss-Gallifent

July 22, 2023

1 Introduction 2
2 Problems, Algorithms, Time Complexity 2
3 Complications 3
3.1 Storage Assumptions 3
3.2 Unclear Algorithm Descriptions 5
3.3 What Goes Inside? 5

1 Introduction

It's very common to hear statements such as "Bubble sort is $\mathcal{O}(n^2)$" but what does this mean exactly?

2 Problems, Algorithms, Time Complexity

Typically in computer science we have a problem we wish to solve (or a calculation we wish to perform) and we create an algorithm to do so.

$$\text{Problem} \longrightarrow \text{Algorithm} \longrightarrow \text{(Pseudo)Code}$$

That algorithm then has an implementation in (pseudo)code and that implementation has a time complexity.

Definition 2.0.1. Technically speaking an *algorithm* is a finite sequence of instructions, rigorously given, which solves a problem or performs a computation.

Most of this time the instructions in the algorithm are clear enough that it is obvious what the corresponding (pseudo)code would look like and so the time complexity becomes clear.

Example 2.1. Consider the problem of determining if a list is sorted. This description of the problem is not an algorithm. An algorithm might be something like:

Algorithm 1: Check if every element is no larger than the next one and return True if this is the case and False otherwise.

We could implement Algorithm 1 as follows:

```
function isSorted1(A):
  n = length(A)
  for i = 0 to n-2:
    if A[i] > A[i+1]:
      return False
    end if
  end for
  return True
end function
```

We can easily see that this pseudcode has time complexity $\Theta(n)$ and so we say that Algorithm 1 has time complexity $\Theta(n)$ where n is the length of the list.

Of course this is not the only algorithm which determines if a list is sorted.

Algorithm 2: Check if every element is no larger than all the following elements and return True if this is the case and False otherwise.

We could implement Algorithm 2 as follows:

```
function isSorted2(A):
  n = length(A)
  for i = 0 to n-2:
    for j = i+1 to n-1:
      if A[i] > A[j]:
        return False
      end if
    end for
  end for
  return True
end function
```

We can easily see that this pseudcode has time complexity $\Theta(n^2)$ and so we say that Algorithm 2 has time complexity $\Theta(n^2)$ where n is the length of the list.

What is critical then is that when we make a statement such as:

Determining if a list is sorted is $\Theta(n)$.

What we really mean is:

The fastest possible known algorithm for determining if a list is sorted has an implementation which has time complexity $\Theta(n)$.

In the above example it would be Algorithm 1.

Of course you might protest - perhaps there is a faster algorithm. And perhaps you're right, and if you found such an algorithm you could then disagree with the above statement and say something like "No, determining if a list is sorted is actually $\Theta(\lg n)$!" and you could present your algorithm for doing this.

Note 2.0.1. In reality this is partly the reason why we might avoid using Θ and stick with $\mathcal{O}$. If we have shown a problem is $\mathcal{O}(n)$ using a $\mathcal{O}(n)$ algorithm then even if an updated algorithm is $\mathcal{O}(\lg n)$, meaning the problem is $\mathcal{O}(\lg n)$, it is still $\mathcal{O}(n)$ as well, whereas this is not the case if we've used Θ.

If this is a bit confusing don't forget that if $f(n) = \mathcal{O}(\lg n)$ then $f(n) = \mathcal{O}(n)$ as well but if $f(n) = \mathcal{O}(n)$ then we cannot tell if $f(n) = \mathcal{O}(\lg n)$.

3 Complications

There are several points to consider, however:

3.1 Storage Assumptions

In our list example above we are assuming that our list is stored in some "nice" way such that we can access list elements in constant time. There are of course

other ways to store a list of numbers such as in a linked list, in a stack, in a queue, and so on, and if we were to alter our assumption of how the list is stored this may or may not change the result.

Example 3.1. Suppose a stack A represents a list of numbers and we wish to determine if the list is sorted. Assume when we pop from an empty stack we get NULL.

An algorithm might be something like:

Algorithm: Pop elements off the stack and check if each element is greater than or equal to the next one popped. Return True if this is the case and False otherwise.

We could implement this as follows:

```
function issorted(A):
  x = A.pop
  if x == NULL:
    return True
  y = A.pop
  if y != NULL:
    if x < y:
      return False
    x = y
    y = A.pop
  end if
  return True
end function
```

We can easily see that if n is the number of elements on the stack then this pseudcode has time complexity $\Theta(n)$ and so we say that our algorithm has time complexity $\Theta(n)$.

While the storage may vary one critical item to note is that the way we are storing the data must be agnostic to the problem. What this means, essentially, is that we can't choose to store the data in a way that makes the problem obvious.

Example 3.2. Suppose we said that we will store a list in the entries $A[1], A[2], ...$ and require that $A[0]$ contains 0 if the list is unsorted and 1 if the list is sorted. Then our algorithm will be:

Algorithm: Check if $A[0]$ is 0 or 1.

With pseudocode:

```
function issorted(A)
  if A[0] == 1:
```

```
        return True
      end if
      return False
    end function
```

Now we claim that determining if a list is sorted is $\Theta(1)$.

Clearly we're being dishonest here in the sense that we're storing our list in a way which is not agnostic to the problem we're trying to solve, and this is a no-no.

3.2 Unclear Algorithm Descriptions

It's not uncommon for the description of an algorithm to be unclear in some specifics.

Example 3.3. Kruskal's algorithm (for graph) involves detecting if adding an edge to a graph forms a cycle. In the description and in the pseudocode we will often simply see some fragment like the following:

```
    if adding this edge forms a cycle:
      blah blah
```

However it is not entirely clear how we would know if adding an edge would form a cycle or not. In such a case we should be more precise when discussing time complexity.

3.3 What Goes Inside?

When we are working with an object such as a list it's fairly clear that when we say that some algorithm is $\Theta(n)$ the n is referring to the number of elements in the list. However if the data we are working with is something different then we have to be more specific.

Some examples that we will see as we progress will be things like:

- When working with $n \times n$ arrays then it makes sense to use n inside.
- When working with $n \times m$ arrays then it makes sense to use both n and m inside.
- When working with graphs with V vertices and E edges then it makes sense to use both V and E inside.