CMSC 351: Algorithm Analysis

Justin Wyss-Gallifent

January 17, 2022

1 Introduction 2
2 Counting Particular Operations 3
3 Time Measurement and Complexity of an Algorithm 6
3.1 Introduction 6
3.2 Examples 8
3.3 Focusing on Big-O and on Worst-Case 10
3.4 Smaller Time Complexity is Not Always Better: 10
3.5 Common O and General Facts 11
4 Space Requirements 11
4.1 Space Complexity 11
4.2 Auxiliary Space 11
4.3 Quirky Ideas 13
5 Thoughts, Problems, Ideas 15

1 Introduction

In this course we can put our goals broadly into two categories:

- Primary Goal: To measure things which are reliant upon the code structure and not upon things like hardware specifics, data management, and so on.
- Seconary Goal: To measure things which are related to things like hardware specifics, data management, and so on.

We will focus mostly on our primary goals and we will look at our secondary goals only under ideal and very specific conditions.

2 Counting Particular Operations

Some particular operations we might count would be the following. This is by no means comprehensive and in fact these are just simple examples to get us thinking.

- Total number of assignments in an algorithm.
- Total number of comparisons in an algorithm.
- Total number of accesses to certain types of memory by an algorithm.

In a more complicated real-world scenario we might be counting database inserts, database record-locking queries, front-end calls to a back-end REST API, updates to the display, and so on.
Personal note: I can't count (pun intended) the number of times I've had a client complain about slow software which can be traced back to a particular operation being repeated far more frequently than the software was designed to sensibly manage or which invoked calls for data on a scale not planned for.
When we use pseudocode we'll need to be very clear about making it code-like enough to accurately measure these things.
We'll usually focus on three measurements and we'll look at each of them in terms of how they change as the input size n changes.

1. **Worst-Case Analysis:** Worst-case analysis might be thought of better as "maximum-case analysis". Here the idea is to see how large a given quantity might be. For example if we're counting assignments in a block of code a worst-case analysis would be asking for the maximum number of assignments that might occur.
2. **Best-Case Analysis:** Worst-case analysis might be thought of better as "minimum-case analysis". Here the idea is to see how small a given quantity might be. For example if we're counting assignments in a block of code a best-case analysis would be asking for the minimum number of assignments that might occur.
3. **Average-Case Analysis:** This is a tricky issue because trying to figure out what an "average-case" looks like involves having an understanding of what all inputs look like and what the probability of each is. This can be situation-dependent. We'll see this more in manageable examples.

In all cases it would be ideal to calculate $\Theta(n)$ to really constrain each case but if that is not possible then we will aim for $\mathcal{O}(n)$ since this will provide an upper bound on the measurement.

Example 2.1. For example consider this simple block of code which sorts a pair of distinct values:

```
\\ PRE: A is an array of two distinct numbers A[0] and A[1].
if A[0] > A[1]
    temp = A[0]
    A[0] = A[1]
    A[1] = temp
end
\\ POST: A is sorted.
```

Let's look at some counts:

- If we're analyzing assignments we can view the code as:

  ```
  if A[0] > A[1]
      temp = A[0]   ⟸
      A[0] = A[1]   ⟸
      A[1] = temp   ⟸
  end
  ```

  In a worst-case scenario `A[0] > A[1]` is true and three assignments are made.

  In a best-case scenario `A[0] < A[1]` is false and no assignments are made.

  In an average-case scenario we must have an idea what the input looks like. Two distinct numbers, great, but are they random? Maybe the input is more likely to have a larger first number, or maybe a larger second number? A simple average-case analysis can be thought of as suggesting that on average half the time the first number is smaller and half the time the first number is larger. Thus half the time we have three assigments and half the time we have none. This gives us an average count of:

  $$0.5(3) + 0.5(0) = 1.5$$

- If we're analyzing comparisons we can view the code as:

  ```
  if A[0] > A[1]    ⟸
      temp = A[0]
      A[0] = A[1]
      A[1] = temp
  end
  ```

  Here there is always one comparison so that's the worst-case, the best-case and the average-case.

Example 2.2. For example consider this simple block of code which does a single pass through a list of length n and swaps adjacent values if they are out of order:

```
\\ PRE: A is an list of length n indexed as A[0,...,n-1].
for i = 0 to n-2
    if A[i] > A[i+1]
        swap = A[i]
        A[i] = A[i+1]
        A[i+1] = swap
    end
end
\\ POST: A is sorted.
```

Let's look at some counts:

- If we're analyzing assignments we can view the code as:

  ```
  for i = 0 to n-2
      if A[i] > A[i+1]
          swap = A[i]      ⟸
          A[i] = A[i+1]    ⟸
          A[i+1] = swap    ⟸
      end
  end
  ```

  In a worst-case scenario the conditional will pass $n-1$ times and $3(n-1) = \Theta(n)$ assignments are made.

  In a best-case scenario the conditional will fail every time and 0 assignments are made.

  In an average-case scenario we must have an idea what the input looks like. Perhaps half the time the conditional will pass. This is $(n-1)/2$ times out of $n-1$. Then $3(n-1)/2 = \Theta(n)$ assignments are made.

- If we're analyzing comparisons we can view the code as:

  ```
  for i = 0 to n-2
      if A[i] > A[i+1]     ⟸
          swap = A[i]
           A[i] = A[i+1]
           A[i+1] = swap
      end
  end
  ```

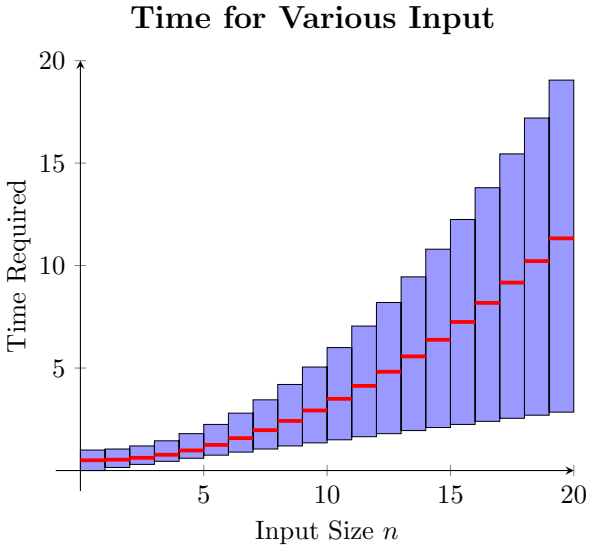
  Here there are always $n-1 = \Theta(n)$ comparisons so that result applies for all of the worst-, best- and average-cases.

3 Time Measurement and Complexity of an Algorithm

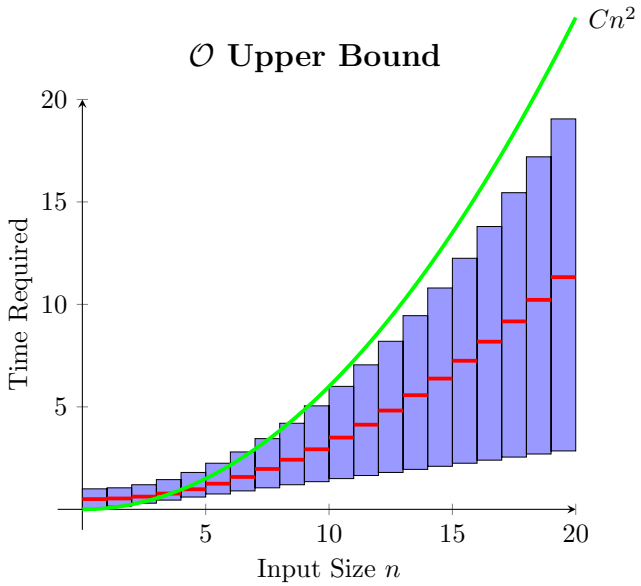
3.1 Introduction

Typically we are less interested in getting an explicit formula for the time complexity (which is good, since the calculations can be messy) but rather in how fast the time complexity grows as n does. In other words we are interested in statements involving $\mathcal{O}$, Θ and Ω.
One key to understanding what's going on is that for each input size n there are many possible inputs, and each input has a time requirement associated to it. For each n there will be some input(s) which take the minimum time, some input(s) which take the maximum time, and then an average time taken over all inputs with their respective probabilities.
We can think of it something like this made-up example:

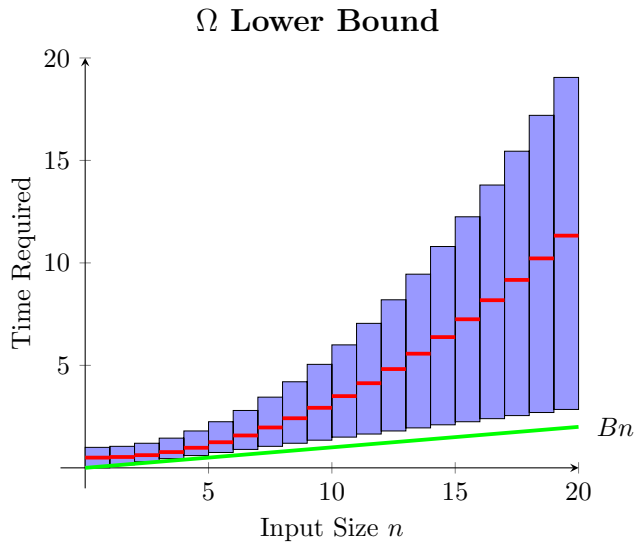
Example 3.1. In this picture each blue rectangle indicates the range of times required for each particular n. The top of each rectangle is each n's worst-case, the bottom of each rectangle is each n's best-case, the red line in each rectangle is each n's average case.
Observe that the average is not necessarily the middle because the distribution of time requirements within each n is not necessarily uniform:



Finding $T(n) = \mathcal{O}(g(n))$ for the worst-case $T(n)$ means finding an eventual bound above the worst-case and hence above everything, such as the following. For example if the green line is $g(n) = Cn^2$ for some $C > 0$ then we can say that the worst-case, best-case and average case are all $\mathcal{O}(n^2)$, meaning it would never get worse than that as $n \to \infty$:



Finding other bounds can be helpful, too. Finding $T(n) = \Omega(g(n))$ for the best-case $T(n)$ means finding an eventual bound below the best-case and hence below everything, such as the following. For example if the green line is $g(n) = Bn$ for some $B > 0$ then we can say that the worst-case, best-case and average case are all $\Omega(n)$, meaning that it would never get better than that as $n \to \infty$.



Other bounds might be less helpful. For example if we find $T(n) = \Omega(g(n))$ for the worst-case $T(n)$ then all we have is an eventual bound below the worst-case.

This only tells us that the worst-case would never get better than this but this is not particularly helpful for a worst-case situation, as it could get a lot worse!

3.2 Examples

Here are a couple of thorough examples.

Example 3.2. Consider the pseudocode code here with time requirements given. It finds the maximum in a list:

```
max = A[0]              c1
for i = 1 to n-1        c2 iterates n − 1 times
    if A[i] > max       c3 iterates n − 1 times
        max = A[i]      c1 iterates when the conditional passes
    end
end
```

In a worst-case scenario the conditional passes for each iteration of the loop and the total time requirement is:

$$T(n) = c_1 + (n-1)(c_2 + c_3 + c_1) = \Theta(n)$$

In a best-case scenario the conditional fails for each iteration of the loop and the total time requirement is:

$$T(n) = c_1 + (n-1)(c_2 + c_3) = \Theta(n)$$

In an average-case scenario we must have an idea what the input looks like. Perhaps half the time the conditional will pass and half the time it will fail. Then the total time requirement is:

$$T(n) = c_1 + \frac{n-1}{2}(c_2 + c_3 + c_1) + \frac{n-1}{2}(c_2 + c_3) = \Theta(n)$$

Note that the time complexity is identical every time but the exact time is different for each.

Example 3.3. Consider the pseudocode code here with time requirements given. It goes through a list once and swaps adjacent elements until it finds two that are not adjacent and then it stops.

```
for i = 0 to n-2          c1 iterates n − 1 times
    if A[i] > A[i+1]      c2 iterates n − 1 times
        swap = A[i]       c3 iterates when the conditional passes
        A[i] = A[i+1]     c3 iterates when the conditional passes
        A[i+1] = swap     c3 iterates when the conditional passes
    else
        break             c4 iterates when the conditional fails
    end
end
```

In a worst-case scenario every iteration has the condition true which then results in a swap and the break never happens. The total time requirement is:

$$T(n) = (n-1)(c_2 + 3c_3) = \Theta(n)$$

In a best-case scenario the first iteration has the condition false and the break occurs immediately. The total time requirement is:

$$T(n) = c_1 + C_2 + c_4 = \Theta(1)$$

In an average-case scenario we must have an idea what the input looks like. Perhaps on average the conditional fails halfway through. Then the total time requirement is:

$$T(n) = \left(\frac{n-1}{2}\right)(c_1 + c_2 + 3c_3) + (c_1 + c_2 + c_4) = \Theta(n)$$

Note that the time complexity is not identical every time.

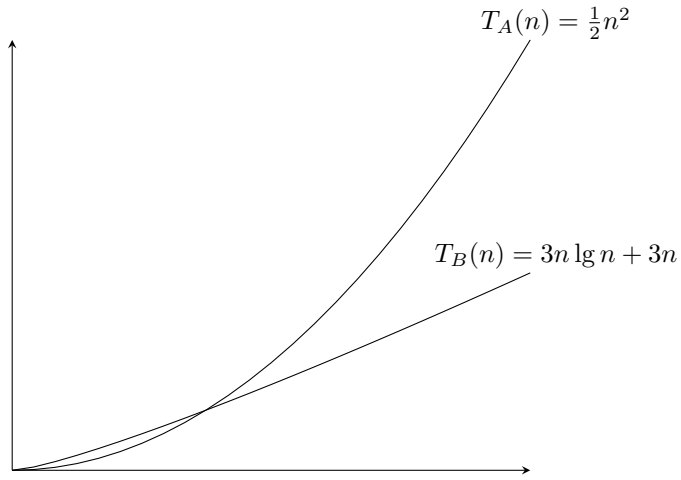
3.3 Focusing on Big-O and on Worst-Case

While finding Θ is ideal, if we can't do this then we'll look for $\mathcal{O}$ as this gives an "it can't be worse than this" result.
While finding all of worst-, best-, and average-cases are ideal, if we can't do all three then we'll look at the worst-case.

3.4 Smaller Time Complexity is Not Always Better:

When it comes to time complexity, smaller functions are preferable, because it means that as the input size n increases the time requirement doesn't increase as much.
However for small values of n it's entirely possible that this isn't the case.
For this reason it's not necessarily good enough to simply know that an algorithm has a certain $\mathcal{O}$ runtime.

Example 3.4. Suppose you have two algorithms A and B and all you know is that $T_A(n) = \mathcal{O}(n^2)$ and $T_B(n) = \mathcal{O}(n \lg n)$. You might automatically think that B is better.
However suppose algorithm A has explicit time complexity $T_A(n) = \frac{1}{2}n^2$ while algorithm B has time complexity $T_B(n) = 3n \lg n + 3n$.
Here is a plot of both, treating n as a real number:



It seems that for n-values before a certain threshold that $T_A(n) < T_B(n)$ and consquently algorithm A is preferable. If we actually test these values for specific values of n we find that $T_A(n) < T_B(n)$ for $n = 0, 1, ..., 37$ and $T_A(n) > T_B(n)$ for $n = 38, 39, ...$ It follows that if our input size is $n = 37$ or smaller then algorithm A is preferable.

When it comes to algorithm design we can imagine that A and B do the same thing but perhaps B is recursive, calling itself on smaller and smaller inputs. We could then suggest that when B is handed an input of size 37 or smaller it might actually invoke A in order to do a quicker job.

3.5 Common O and General Facts

The following arise most frequently:

- $\Theta(1)$: Constant time. The best.
- $\Theta(\lg n)$: Generally the best next largest. Binary search works at this speed. Most things don't.
- $\Theta(n)$: Not bad. Standard linear operations tend to be like this.
- $\Theta(n \lg n)$: Common for things involving trees, recursion, and so on. The best sorting algorithms are in this category.
- $\Theta(n^2)$, $\Theta(n^3)$, ...: Not bad when the input size is small but can be useless for large input size. Sloppy algorithms on arrays tend to land here.
- $\Theta(2^n)$, $\Theta(3^n)$, ...: Pretty useless except for very small input size.
- $\Theta(n!)$, $\Theta(n^n)$: Ditto.

4 Space Requirements

There are several ways in which we can look at the space that an algorithm requires.

4.1 Space Complexity

Definition 4.1.1. One is *space complexity* which measures how much memeory is used for everything, including the data, extra memory requirements, and so on. We will not examine this.

4.2 Auxiliary Space

Definition 4.2.1. Second is *auxiliary space* which measures how much space is required in addition to of the input data. The input data itself is not taken into account.

Example 4.1. The following code swaps two variables `a` and `b`:

```
\\ PRE: a and b are variables.
temp = a
a = b
b = temp
\\ POST: The values are switched.
```

This code uses a single external value `temp`. If we're looking at auxiliary space then we would say it requires just one value and hence the auxiliary space is $\mathcal{O}(1)$.

Suppose we wanted to reverse a list with an even number of elements. Here is one approach:

```
\\ PRE: A is a list.
B = []
for i = len(A)-1 down to 0
    B.append(A[i])
end
for i = 0 to len(A)-1
    A[i] = B[i]
end
\\ POST: A is reversed.
```

This code creates a new array `B` by reading the elements of `A` in reverse order and growing `B` accordingly. Due to the creation of this new list, if `A` has n elements then our algorithm creates a brand new list also requiring n elements. In addition we have the loop variable `i` and therefore uses $n+1 = \mathcal{O}(n)$ auxiliary space.
We can do better, though:

```
\\ PRE: A is a list.
for i = 0 to len(A)/2-1
    temp = A[i]
    A[i] = A[len(A)-1-i]
    A[len(A)-1-i] = temp
end
\\ POST: A is reversed.
```

In this case we essentially swap elements - the first with the last, the second with the second-to-last, and so on. We require one additional auxiliary variable for the swapping, `temp`, and one for the loop `i`, and so this algorithm uses $1 + 1 = \mathcal{O}(1)$ auxiliary space.

Definition 4.2.2. This algorithm is said to reverse the list *in-place*.

It will often be the case that one algorithm may be slower but in-place while another may be faster and not in-place. In a case like this we may need to decide (using some external reasoning) which algorithm is "better" for our circumstances.

4.3 Quirky Ideas

Sometimes it can be educational (even if not useful) to attempt to write algorithms which are restricted to small amounts of auxiliary space or other quirks with working memory.

Example 4.2. Consider the following BubbleSort pseudocode:

```
\\ PRE: A is a list of length n.
stillgoing = True
while stillgoing
    stillgoing = False
    for i = 0 to n-2
        if A[i] > A[i+1]
            temp = A[i]
            A[i] = A[i+1]
            A[i+1] = temp
            stillgoing = True
        end
    end
end
\\ POST: A is sorted.
```

This requires three auxiliary variables - `stillgoing`, `i` and `temp`. Can we modify this code to remove the required `temp`? This variable arises to do the swap, so can we swap with no third variable? Interestingly yes.
Observe that if x and y are given and if we do the three steps in order:

$$x = x + y$$
$$y = x - y$$
$$x = x - y$$

The result will be that the variables switched. For example if $x = 10$ and $y = 42$ then:

$$x = x + y = 10 + 42 = 52$$
$$y = x - y = 52 - 10 = 42$$
$$x = x - y = 52 - 42 = 10$$

Thus the pseudocode can be modifed according with the conditional becoming:

```
...
        if A[i] > A[i+1]
            A[i] = A[i] + A[i+1]
            A[i+1] = A[i] - A[i+1]
            A[i] = A[i] - A[i+1]
            stillgoing = True
        end
...
```

These sorts of exercises are useful because:

- They force us to consider what space requirements actually are.
- They force us to consider how wasteful we often are.
- If tight memory situations do arise we don't automatically give up.

5 Thoughts, Problems, Ideas

1. Consider the following pseudocode:

```
\\ PRE: A[0,...,n-1] is a list of integers.
positiveproduct = 1
for i = 0 to n-1
    if A[i] > 0
        positiveproduct = positiveproduct * A[i]
    end
end
\\ POST: positiveproduct contains the product of the
\\       positive entries.
```

   Assume for the average cases that the conditional is true half the time.

   (a) In each of the worst-, best-, and average-cases how many conditional checks will take place?

   (b) In each of the worst-, best-, and average-cases how many assignments will take place?

   (c) If time values are assigned as follows:

   - Assignments take time c_1.
   - Loop maintenance takes time c_2.
   - Conditional checks take time c_3.
   - Multiplication takes time c_4.

   In each of the worst-, best-, and average-cases how much time will be required? Calculate and give the time complexity of each.

2. Consider the following pseudocode:

```
\\PRE: A[0,...,n-1] is a list of integers.
sum = 0
for i = 0 to n-1
    if A[i] < 0
        break
    else
        sum = sum + A[i]
    end
end
\\ POST: sum is the sum up until the first negative entry.
```

   Assume for the average cases that the conditional is true half the time.

   (a) In each of the worst-, best-, and average-cases how many conditional checks will take place?

(b) In each of the worst-, best-, and average-cases how many assignments will take place?

(c) In each of the worst-, best-, and average-cases how many `break` statements will take place?

(d) If time values are assigned as follows:

- Assignments take time c_1.
- Loop maintenance takes time c_2.
- Conditional checks take time c_3.
- Addition takes time c_4.
- Break statements takes time c_5.

In each of the worst-, best-, and average-cases how much time will be required? Calculate and give the time complexity of each.

3. Consider the following pseudocode:

```
\\PRE: A[0,...,n-1] is a list of positive integers
\\     and M is a positive integer.
sum = 0
for i = 0 to n-1
    sum = sum + A[i]
    if sum > M
        break
    end
end
\\ POST: sum is the sum up until it is more than M.
```

Assume for the average cases that the average value in each array entry is $2M/n$.

(a) In each of the worst-, best-, and average-cases how many conditional checks will take place?

(b) In each of the worst-, best-, and average-cases how many assignments will take place?

(c) If time values are assigned as follows:

- Assignments take time c_1.
- Loop maintenance takes time c_2.
- Conditional checks take time c_3.
- Addition takes time c_4.
- Break statements takes time c_5.

In each of the worst-, best-, and average-cases how much time will be required? Calculate and give the time complexity of each.

4. Suppose Algorithm A has running time $T_A(n) = 0.01n$ while Algorithm B has running time $T_B(n) = 12 \lg n$. For which n is algorithm A better and for which n is algorithm B better?

5. Suppose Algorithm A has running time $T_A(n) = 100 + 100n^{3/4} \lg n$ while Algorithm B has running time $T_B(n) = n^2$. Find the smallest n_0 such that for all $n \geq n_0$ we know that Algorithm B is faster.

6. Consider the following pseudocode:

```
\\ PRE: A is a list of length n
\\ with each entry between 1 and n inclusive.
B = [1]
for i = 0 to n-2
    if A[i] > A[i+1]
        B.append[1]
    end
end
\\ POST: What a mystery!
```

   What are the best- and worst-case auxiliary space requirements?

7. Consider the following pseudocode:

```
\\ PRE: A is a list of length n
\\ with each entry between 1 and n inclusive.
B = []
for i = 0 to n-1
    for j = 1 to A[i]
        B.append(A[i])
    end
end
\\ POST: What a mystery!
```

   What are the best- and worst-case auxiliary space requirements?

8. Suppose you have a list A of length n and a command `swap(i,j)` that swaps $A[i]$ and $A[j]$. Specifics of pseudocode aside what is the maximum number of swaps that could possibly be necessary to put the list in order?

9. Modify the array-reversing pseudocode so that it will work with lists containing an odd number of elements as well, while still only requiring $\mathcal{O}(1)$ auxiliary space.

10. Consider the following pseudocode:

```
\\ PRE: A is a list of length n.
count = 0
for i = 0 to n-1
    for j = 0 to n-1
        if i < j and A[i] > A[j]
            count = count + 1
        end
        if i > j and A[i] < A[j]
            count = count + 1
        end
    end
end
\\ POST: What a mystery!
```

Suppose you know that:

- Each of the four basic arithmetic operations takes time 3.
- Each comparison takes time 2.
- Each assignment (including each iteration assignment in a loop) takes time 1.

(a) Find the value of `count` for the inputs `A=[1,2,3,4,5]`, `A=[5,4,3,2,1]` and `[1,4,3,5,2]`.

(b) In general what value does `count` give?

(c) Replace both `if` statements by a single `if` statement with a few simple arithmetic operations and a single comparison which achieves the same goal.

(d) Which out of the original pseudocode or your modified pseudocode take would take the least time in a worst-case scenario?