Lecture #26: Theory

CS106E, Young

In this lecture we take a look at Computer Science Theory.

We begin by considering Analysis of Algorithms. We consider three different methods for searching for an item – Linear Search, Binary Search, and searching for an item in a Hash Table. As we learn, these each have very different performance characteristics.

We take a look at O-Notation, which is used by Computer Scientists to compare algorithms. While we are most often concerned with how fast an algorithm is, we can also analyze how much space it takes. In addition, we can consider how the algorithm performs in the average case vs. the worst case scenarios.

Not all problems are decidable. We consider the case of the Halting Problem – whether or not a program can be written which can analyze whether or not another program will halt given a specific input. We show that we cannot write such a program.

Finally we take a quick look at Turing Machines, which are used by Computer Scientist to prove theories about computing.

Analysis of Algorithms
When writing a program, there are often many different ways to carry out a task. How do computer scientist decide which approach to take? Some computer theory experts focus on analyzing different algorithms. We'll take a look at searching algorithms and along the way will get a better understanding of what computer scientists look at when comparing algorithms.

Searching Algorithms
Searching for an item within a list or collection of items is a very common task. There are several different ways to carry out this task.

Linear Search

- In a linear search we simply look at each item in our list of items, starting at the first item in the list and stopping either when we've found the item we are looking for or when we reach the end of the list. If we reach the end of our list and still haven't found the item, we know it's not in the list.

- One issue we're often concerned with is how long does it take to carry out an algorithm as the number of items increases. Let's consider this in the case of Linear Search.

    - If on average, it takes our computer 1 minute to find an item in a list of 1000 items, how long will it take to find an item in a list of 2000 items?
    - If we think about how linear search works, we can see on average it will take twice as long to find an item in a list that is twice as long, since we'll have to

search through twice as many items. So if it takes our computer 1 minute to search 1000 items, it will take 2 minutes on average to search through 2000 items.[1]

- In fact, we can work out a formula that looks something like this:

$$averageTimeToFind = \frac{NumberOfItems}{1000} \times 1\ minute$$

- This formula is based on how long it will take a particular computer to find an item our list. What if we get a faster computer? In this case, we'll probably be able to process more than 1000 items per minute. However, the amount of time it takes will still increase directly with the number of items in the list.

- We can write a general formula like this:

$$averageTimeToFind = Constant \times NumberOfItems$$

where the Constant is determined by the speed of our computer. For our first computer, the Constant is 1 minute / 1000 items which works out to 0.06 seconds per item. The constant will change depending on the speed of our computer, but there will always be a linear increase in the amount of time taken as the number of items we search through increases.

Binary Search

- Can we search for items more efficiently than linear search? Yes. ***If the list is sorted***, we can carry out a binary search.
- In a ***binary search*** we start looking for an item by splitting the sorted list in half. We compare the middle item in the list to the item we're looking for, if it matches, we're done. If it doesn't, we consider whether that middle item is bigger or smaller than the item we're looking for.
- If the middle item is smaller than the item we're looking for, then our item must be in the upper-half of the list (since the list is ordered), we look at the upper-half of the list, split it in half, compare our item to the item in the middle of the upper-half of the list and repeat.
- If instead the middle item in the list was larger than the item we're looking for, then our item must be in the lower-half of the list. We discard the upper-half of the list, split the lower-half of the list in half, compare our item to the middle item of the lower-half the list and repeat.

- Because we repeatedly halved the list we're searching for over and over again, the time needed to search a list doesn't increase linearly with the number of items we're searching. Instead it increases logarithmically with the number of items.

    - You may recall that log(10) = 1, log(100) = 2, log(1000) = 3, log(10,000) = 4 and so on.
    - In this case, we're not taking base-10 logs. We're taking base-2 logs, which may be written as $\log_2(x)$ or ln(x).

[1] We do need to emphasize that this comparison is "on average". It's possible the item we're looking for is actually the first item in our list of 2000 in which case we'll take considerably less time than 2 minutes. However, it's equally likely that our search item is the last item in the list, requiring us to search all 2000 items.

    - For base-2 $\log_2(2) = 1$, $\log_2(4) = 2$, $\log_2(8) = 3$.

- For Binary Search, we can write a general formula like this:

$$averageTimeToFind = Constant \times \log_2(NumberOfItems)$$

Hash Tables

- Binary Search is much more efficient than Linear Search.
- You may be surprised to find out, however, that we can do even better than that using something called a Hash Table.
- In a Hash Table, we store items in an array. What makes the Hash Table interesting is how we determine which array slot to store an item in. We use a Hash Function on our item to determine where to store it.
- There are different ways to Hash an item, let's take a look at a simple example.

- In this case we will store Strings in our Hash Table.
- The Hash Function we'll use to store our Strings will go through our String and determine the ordinal number in the alphabet for each individual letter in the String. We will then add each of these numbers together giving us a Hash Value. The Hash Value will be used to determine the storage location.
    - For example, if we're trying to store the string "CAT", "C" is the 3rd letter of the alphabet, "A" is the 1st letter, and "T" is the 20th letter. So our Hash on "CAT" gives us 3 + 1 + 20 = 24.
    - Depending on how big our table is, we may be able to simply store "CAT" in the 24th slot. However, in some cases, the table will be too small, so we can take the Hash value and modulus it by the size of the array.
        - The modulus is the remainder in integer division. So 34 ÷ 10 is 3 with a remainder of 4. So 34 modulus 10 is 4.
        - Similarly 17 ÷ 4 is 4 with a remainder of 1, so 17 modulus 4 is 1.
        - Taking the result of our Hash Function modulus the array size means that the result will always fit somewhere within our array.
        - Let's assume our array is size 10, "CAT" gives us 24 modulus 10, which is 4, so we store "CAT" in slot 4.
    - Let's try a couple more of these.
        - As we've seen, "CAT" stores in slot 4.
        - "DOG" has the 4th, 15th, and 7th letters, so it gives us a total of 26. 26 modulus 10 gives us 6 so we store it in slot 6.
        - "TIP" has the 20th, 9th, and 16th letters. Totaled this gives us 45. We take modulus 10 and this gives us 5, so we store it in slot 5.

- We've been placing items in our table. We can retrieve items the same way. If we want to know if "BOX" is in our table, we add 2 + 15 + 24 giving us 41. We modulus with the table size of 10 giving us 1, we then check the 1st entry.

- How does this compare with Linear and Binary Search?
- The amount of time it takes to determine if an item is in the table is actually independent of the table size. If my table includes 30,000 items, I take the same amount of time as if my table contains 5 items.

    - The time cost is simply a matter of calculating the Hash. This is a fixed time.[2]

$$timeToFind = Constant$$

- For large numbers of items, this is much, much more efficient than either Linear Search or Binary Search.

- If you've thought carefully about our Hash Table, you may have realized that if we have a lot of items, they'll sometimes end up being assigned to the same slot. For example, both "CAT" and "ACT" hash to the same location.
- This is called a ***collision***. There are a variety of ways to handle collisions. One solution is to have the slots in our hash table contain lists instead of individual items. However, if we have a good hashing function and a reasonably large table, collisions should not occur frequently, and our Hash Table will have good performance.

O-Notation

- As we discussed in considering how long a Linear Search might take, the actual amount of time taken to find an item is actually dependent on the speed of the computer, in addition to the number of items.
- We want a way of comparing algorithms independent of the actual computer.
- In Computer Science we can compare an algorithms performance using O-notation. For example, we say that Linear Search is O(n) whereas Binary Search is O(log(n)).

- Formally O-notation says:

$$f(x) = O(g(x)) \text{ if there exists constants c and k such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq k$$

- Let's break this down.
    - You can think of the first constant c as representing how long it takes to perform a particular iteration through our algorithm. Suppose it takes 1 minute to carry out each comparison in our binary search, but only 1 second to carry out a comparison in linear search? It turns out if we're looking through a lot of items, binary search will still be better, because the actual number of comparisons is much smaller.
    - The second constant k tells us that we aren't concerned with any initial startup costs. If it takes 3 minutes to setup our binary search and no time at all to start our linear search. As we try to process larger and larger numbers of items the initial 3 minute startup cost becomes less and less important.
    - O-notation is concerned with the behavior of an algorithm when processing large numbers of items

- Using O-Notation, we can say that:

    Linear Search is O(n)
    - That is to say that the amount of time a linear search takes increases directly with the number of items. This ignores startup costs and is not concerned with the actual amount of time to process a single item. Regardless of the time to process a single item, the total amount of time will increase at a linear rate with the number of items.

[2] Actually if we want to be completely accurate, calculating this particular hash function is dependent on the length of the word we are looking up, since we have to count up each individual letter. However, the length of the word is independent of the number of words we're storing, and generally we're concerned with the total number of items, not the length of individual items.

Binary Search is O(log (n))

- Binary search increases as a logarithm of the number of items. Again ignoring startup costs and not being concerned with the time to process a given item, the total time will increase at a logarithmic rate with the number of items we are working with.

Hash Lookup is O(1)

- Looking up an item in a hash table takes a constant amount of time, regardless of the number of items, so we say it is O(1) (pronounced "Oh of One").

Common Complexities

Common complexities you may see for algorithms include (from best to worse)

O(1)	Constant
O(log(n))	Logarithmic
O(n)	Linear
$O(n \times \log(n))$	
$O(n^2)$	Quadratic
$O(2^n)$	Exponential

Notations Related to O-Notation

- There are a variety of different notations similar to O-Notation.
- O-Notation says that the performance of our algorithm is equal to or better than the given formula. So $O(n^2)$ means our actual performance is n^2 or better.
- Ω-Notation (big omega) is the opposite. It says our actual performance is equal to or worse than a particular formula. So $\Omega(n^2)$ means our actual performance could be much worse than n^2.
- Θ-Notation (big theta) says that our performance is exactly equal to the given formula.

Time vs. Space Complexity

- While our discussion of complexity has focused on the amount of time a particular algorithm takes, we may also be concerned with the amount of space a particular algorithm needs to carry out its task.
- Often time and space are traded off, with one algorithm providing speed but requiring a lot of space and another algorithm using less space, but not performing as quickly.
  - For example, the merge sort algorithm provides very good sorting time performance, but cannot sort an array in the original space provided, it needs extra storage space to carry out its task. In contrast, most sorting algorithms sort in place, needing no space other than the original array storing the items to be sorted.

Average-Case, Worst-Case Performance

- An algorithm's average performance isn't necessarily the same as its worst-case performance. For example, quicksort typically gives O(n log(n)) performance but gives $O(n^2)$ performance in the worst case.

Other Considerations

- There are other issues that we might consider when choosing an algorithm. For example, some algorithms are more amenable to splitting the work across multiple processors.

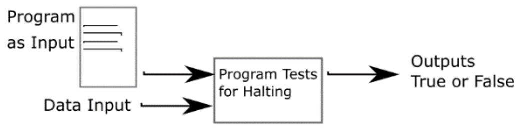
Undecidable Problems

- Is there a limit to what our computer programs can compute? Yes. Some problems are undecidable and we cannot write a program to solve them.
- One famous undecidable problem is the halting problem.

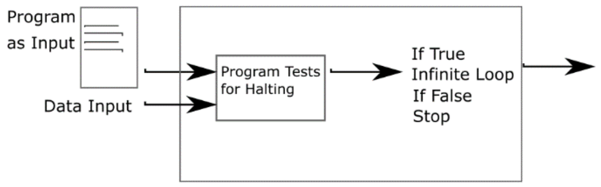
Halting Problem

- We want to write a program that analyses other programs. Can we write a program that can determine whether or not a program passed in for analysis will actually halt for a given input?
    - We can prove by contradiction that such a program is impossible.

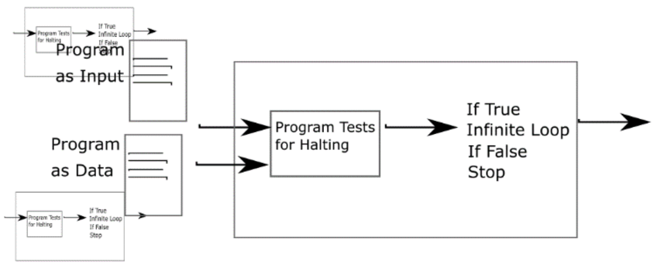
- We begin by assuming that we can in fact write such a program. Here I've drawn it out, showing it taking another program's code as one input and the data that we would execute that program's code on as the other input. It then either returns true or false, depending on whether or not that particular data input should cause the program being analyzed to halt.



- Now, let's extend our program by tacking on some additional lines of code. If the original halting analysis program returns true, we'll extend it forcing the program to go into an infinite loop. If the original halting analysis program returns false, we'll let our extended program stop. Our extended program includes two parts, the original halting program, which I'll refer to as the first stage, and the add-on which may infinite loop, which I'll refer to as the second stage.



- Finally we take the program code for this new extended program (the halting analysis program with our potentially infinite looping code attached to it) and we feed this code in as the program we want to analyze. We then ask whether or not this new program would halt, if its own code was fed into it for analysis.



- The question is what happens?
    - If our original halting analysis program in the first stage concludes that our extended program would halt when analyzing its own code, then that first stage returns true. However, if the first stage returns true, then the program doesn't actually halt – it goes into the infinite loop in stage two.
    - Similarly if the original halting analysis program in stage one concludes that our extended program would not halt, when analyzing its own code, then the first stage returns false. However, if the first stage returns false, then our extended program halts.
    - In both cases, the stage one halting analysis program that we postulated should exist fails to correctly analyze the program – determining the program would halt, when in fact it

goes into an infinite loop, and determining that it won't halt, when in fact the program does.
    - No such analysis program can exist, because it would lead to this contradiction.

Turing Machines

- If you start reading articles on computers, you may run into a reference to a Turing Machine.
- A Turing Machine is a machine that can read and write symbols onto an infinitely long paper tape.
- The Turing Machine includes a control program that tells it when it sees a particular symbol on the tape, what to do.
    - The Turing Machine can write a symbol onto the tape.
    - It can move the tape to the left or the right.
    - It can halt execution.

- What is a Turing Machine and why does it exist?
    - The Turing Machine provides a relatively simple model of computing.
    - This allows Computer Scientists to theorize what computers can or can't do while performing their analysis on the clearly defined Turing Machine, rather than the more complex messier environment of a real-world computer.
    - It has been proved that most programming languages are Turing Complete[3], which means anything you can do with a Turing Machine you can do with a Turing Complete language. So statements that are true of Turing Machines are also true of programs in general.

Other Models of Computing

- There are a variety of other models of computing that Computer Theory experts use for their proofs.
- There are several variants of Turing Machines, including one designed to simulate Quantum Computers.
- There are other models designed to facilitate proofs about parallel computing.

[3] Actually this is almost, but not quite correct. Most programming languages would be Turing Complete, if they had an infinite amount of computer memory to work with.