Lecture 27: Theory of Computation

Marvin Zhang
08/08/2016

Announcements

Roadmap

- Introduction
- Functions
- Data
- Mutability
- Objects
- Interpretation
- Paradigms
- Applications

- This week (Applications), the goals are:
  - To go beyond CS 61A and see examples of what comes next
  - To wrap up CS 61A!

Theoretical Computer Science

- The subfield of computer science that focuses on more *abstract* and *mathematical* aspects of computing
- A very broad and diverse subfield that interacts with many other fields in and outside of computer science
- A big part of this subfield is *theory of computation*
- We will look at two topics in theory of computation:
  - *Computability theory*
    - "Can my computer solve this problem?"
  - *Complexity theory*
    - "Can my computer solve this problem efficiently?"
- If today is interesting, consider CS 170 and CS 172

Computability Theory

What can computers do?

The Halting Problem

- Can computers solve any problem we give them?
  - If not, what can't they do?
- One useful problem we would like to solve, called the *halting problem*, is to check if a function runs into an infinite loop, since we would usually like to avoid this
  - Let's focus on functions that take in one argument

```
def whoops(x):
    while True:
        pass
```

```
def okay(x):
    return x + 1
```

```
def whookay(x):
    while x != 0:
        x -= 2
```

- Can we write a function `halts` that takes in a function `func` and an input `x` and returns whether or not `func` halts when given input `x`?

The Halting Problem

```
def halts(func, x):
    # ???
```

- It turns out that we cannot write `halts`! There is no implementation that accomplishes what we want
  - The halting problem is called *undecidable*, which basically means that we can't solve it using a computer
- We can prove that we cannot write `halts` through a *proof by contradiction*:
  1. *Assume* that we can write `halts`
  2. Show that this leads to a logical *contradiction*
  3. *Conclude* that our assumption must be false

The Halting Problem

1. *Assume* that we can write `halts`

- Let's say we have an implementation of `halts`, that works for every function `func` and every input `x`:

```
def halts(func, x):
    """Returns whether or not func ever stops
    when given x as input.
    """
```

2. Show that this leads to a logical *contradiction*

- Let's write another function `very_bad` that takes in a function `func` and does the following:

```
def very_bad(func):
    if halts(func, func):  # check if func(func) halts
        while True:  # loop forever
            pass
    else:
        return  # halt
```

The Halting Problem

2. Show that this leads to a logical *contradiction*

```
def very_bad(func):
    if halts(func, func):  # check if func(func) halts
        while True:  # loop forever
            pass
    else:
        return  # halt
```

- What happens when we call `very_bad(very_bad)`?
  - If `very_bad(very_bad)` halts, then loop forever
  - If `very_bad(very_bad)` does not halt, then halt
- So... does `very_bad(very_bad)` halt or not?
  - It *must* either halt or not halt, there exists no third option

The Halting Problem

2. Show that this leads to a logical *contradiction*

- If `very_bad(very_bad)` halts,
  - Then `very_bad(very_bad)` does not halt
- If `very_bad(very_bad)` does not halt,
  - Then `very_bad(very_bad)` halts
- This is a contradiction! It simply isn't possible

3. *Conclude* that our assumption must be false

- `very_bad` is valid Python, there is nothing wrong there
- So it *must* be the case that our assumption is wrong
- Therefore, there is no way to write `halts`, and the halting problem must be undecidable

Decidability

- Roughly speaking, the *decidability* of a problem is whether a computer can solve the particular problem
  - The halting problem is undecidable, as we have shown
  - All other problems we have studied are decidable, because we have written code for all of them!
- There are other problems that are undecidable, and there are various ways to prove their undecidability
  - One way is proof by contradiction, which we have seen
  - Another way is to *reduce* the problem to the halting problem
- In a reduction, we find a way to solve the halting problem using the solution to another problem
  - "If I can solve this problem, then I can also solve the halting problem" implies:
    - "I can't solve this problem, because I can't solve the halting problem."

Decidability

- As an example, we can't write a function `computes_same` that takes in two functions `f1` and `f2` and returns whether or not `f1(y) == f2(y)` for all inputs `y`

```
def computes_same(f1, f2):
    # ???
```

- "If I can solve `computes_same`, then I can also solve the halting problem"

```
def halts(func, x):
    def f1(y):
        func(x)
        return 0
    def f2(y):
        return 0
    return computes_same(f1, f2)
```

Decidability

```python
def halts(func, x):
    def f1(y):
        func(x)
        return 0
    def f2(y):
        return 0
    return computes_same(f1, f2)
```

- If `f1(y) == f2(y)` for all inputs `y`, then `f1(y) == 0` for all inputs `y`
  - This implies that `func(x)` halts, because otherwise `f1(y)` is undefined for all inputs `y`
- So this successfully solves the halting problem!
  - "I can't solve `computes_same`, because I can't solve the halting problem."

Complexity Theory

What can computers do efficiently?

Complexity

- So, there are some problems that computers can't solve
- For all the problems that can be solved, can we solve them efficiently? This is a much more practical concern

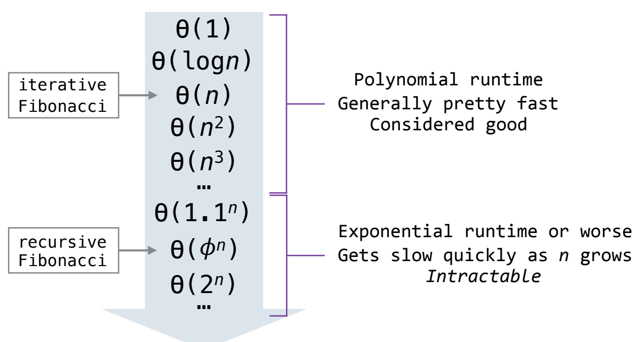
```python
def fib(n):
    if n == 1:
        return 0
    elif n == 2:
        return 1
    return fib(n-1) + fib(n-2)
```

$\Theta(\phi^n)$
exponential runtime
(very bad!)

```python
def fib(n):
    curr, next = 0, 1
    while n > 0:
        curr, next = next, curr + next
        n -= 1
    return curr
```

$\Theta(n)$
linear runtime
(much better!)

Orders of Growth



Complexity Classes

- We often make the distinction between polynomial runtime and exponential runtime, and ignore the differences between different polynomials or different exponentials
- Roughly speaking, solutions with polynomial runtime are usually "good enough", whereas exponential runtime is usually too bad to be useful
- Practically, there is certainly a difference between solutions with, e.g., $\Theta(n)$ runtime and $\Theta(n^3)$ runtime
  - But this is a smaller difference than solutions with, e.g., $\Theta(n^3)$ runtime and $\Theta(2^n)$ runtime
  - It is also generally easier to reduce polynomials than to reduce exponential runtime to polynomial runtime
- Ignoring the smaller differences allows us to develop more rigorous theory involving *complexity classes*

Disclaimer

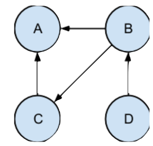
- The rest of this lecture is less formal, because we have to skip some of the more complicated details
- So, don't quote what I say or write, because I will get in trouble
  - Instead, just try to understand the *main ideas*
- If you want all of the details, I refer you to:
  - CS 170 (Efficient Algorithms and Intractable Problems)
  - CS 172 (Computability and Complexity)
  - Or the equivalent courses at other institutions

Complexity Classes

- The two most famous complexity classes are called *P* and *NP*
- The class P contains problems that have solutions with polynomial runtime
  - Fibonacci is in this class, since the iterative solution has linear runtime
  - Most problems we have seen so far are in P
- The class NP contains problems where the answer can be *verified* in polynomial time
  - If I tell you: “The n^{th} Fibonacci number is k”
  - Can you verify that this is correct in polynomial time?
- In this example, the answer is yes, because you can just run the iterative solution to check, so Fibonacci is also in NP

Example: Hamiltonian Path (demo)

- Given a graph, is there a path through the graph that visits each vertex exactly once?



- Is this problem in NP? Yes!
  - If I am given a graph and a proposed Hamiltonian path, I can easily verify whether or not the path is correct
  - I just have to trace the path through the graph and make sure it visits every vertex
- Is this problem in P? We don’t know
  - We have seen two exponential runtime solutions for this problem, one in Logic and a similar one in Python
  - But there could be another solution with polynomial runtime, we can’t be sure

P and NP

- Is every problem in P also in NP? Yes!
  - If a problem is in P, then it has a solution with polynomial runtime
  - So if I want to verify an answer for an instance of the problem, I can just run the solution and compare
  - This takes polynomial time, so the problem is in NP
- Is every problem in NP also in P?
  - In other words, if I can verify an answer for a problem in polynomial time, can I also compute that answer myself in polynomial time?
  - *No one knows*
  - But most people think it’s unlikely

P = NP (?)

- So, we know that P is a subset of NP, but we still don’t know whether or not they are equal
- Most people think they’re not equal, because you could do a lot of crazy things if they are
  - Automatically generate mathematical proofs
  - Optimally play Candy Crush, Pokémon, and Super Mario Bros
  - Break many types of security encryption
    - Verifying a password is very easy, just type it in and see if it works
    - Imagine if *figuring out* a password was just as easy
- The P = NP problem is one of the seven Millennium prizes
- If I just proved that P = NP, how do I take over the world?

Summary

- Computability theory studies what problems computers can and cannot solve
  - The halting problem cannot be solved by a computer
  - Reducing other problems to the halting problem shows that they cannot be solved either
  - This is not really a practical concern for most people
- Complexity theory studies what problems computers can and cannot solve efficiently
  - This is a practical concern for basically everyone
  - There are still many unanswered questions, for example, whether or not P = NP
- CS 170 and CS 172 go into more detail on this material