Limits of Computation

15-112 (4/25/19)

Big Ideas

- Sometimes, we cannot find a (reasonable) solution to a problem
- Sometimes, we cannot ensure that a program does what we want it to
- But we can often find solutions that are reasonable enough, and prove that they work in specific cases.

For a given problem…

Can we find an efficient algorithm?

What do we think is efficient?

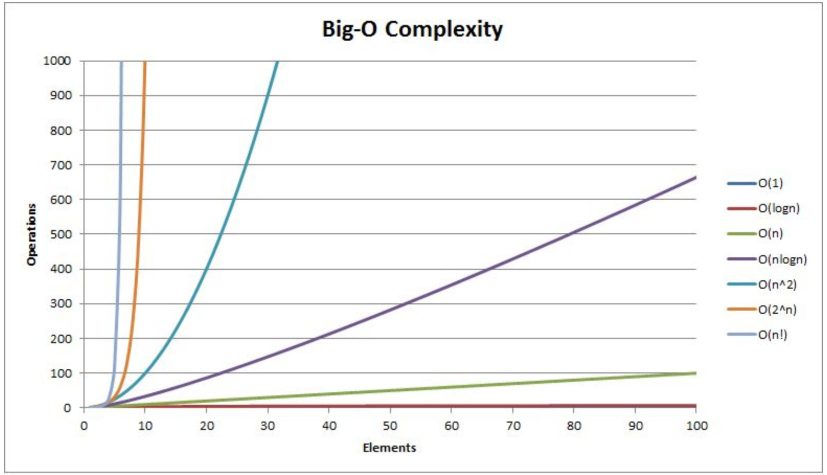
So far, we've talked about function families: $O(1)$, $O(n)$, $O(n\log n)$, $O(n^2)$, $O(2^n)$...

In broader computer science, we categorize these function families into two groups:

Polynomial time: $O(1)$, $O(n)$, $O(n\log n)$, $O(n^k)$

Exponential time: $O(2^n)$, $O(k^n)$, $O(n!)$

Whenever possible, we want our algorithms to run in polynomial time.



subsetSum

Problem: Given a list L of n elements and an integer x, is there a sublist of L that sums to x?

Obvious solution: produce all possible subsets, return the first one that sums to x.

This works, but is slow on even medium-sized inputs! It runs in $O(2^n)$ time (why?), which is exponential (bad)

Can we do better?

Verifying a solution

Assume we have a magic box that can take in a list and produce an answer to `subsetSum` for that list. We want to check if this box is legit.

Discuss: How long does it take to verify that that answer is correct (is a subset of L and sums to x)?

Answer: O(nlogn) for checking the subset, O(n) for checking the sum. **Verifying is polynomial!**

subsetSum([16, 37, 6, 40, 96, 34, 16, 66], 112)

[16, 6, 40, 34, 16]

Complexity Classes NP and P

There is a class of problems that can have solutions **verified** in polynomial time. This class is called NP, short for “non-deterministic polynomial time”. A function like `solveSudoku` is in NP.

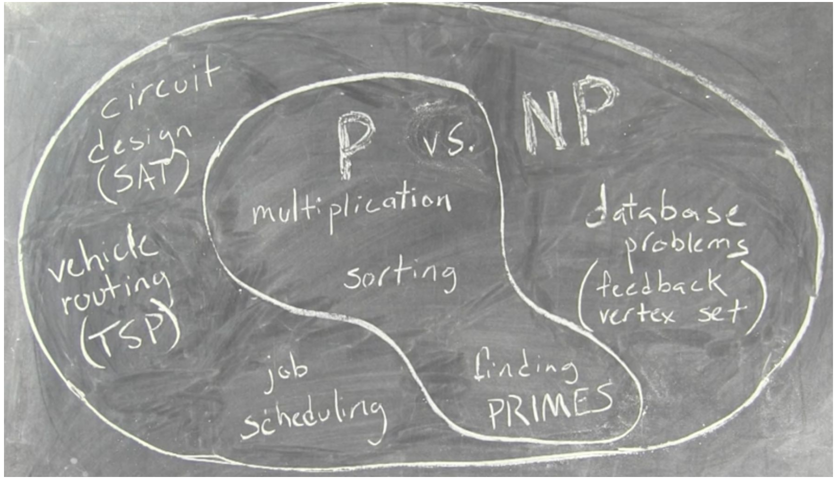
There is another class of problems that can be **solved** in polynomial time. This class is called P, for polynomial. A function like `isPrime` is in P.

So far we’ve established that `subsetSum` is in NP. We don’t yet know whether it’s in P- maybe we could make a faster solution, and then it would be. In general, we know that **P is a subset of NP** (if we can solve in polynomial time, we can verify too!).

Problems in NP

There are lots of common and useful problems in the class NP, problems that we don't have a polynomial-time solution for (yet). These include:

- Subset sum
- Optimized packing of items (loadBalance)
- Route-planning (Travelling Salesman)
- Coloring a graph (solveSudoku)
- Scheduling with constraints (final exams)
- And many more...



P vs. NP

Big Idea: wouldn't it be nice if all problems in NP were also in P? Then we could have fast solutions to lots of problems! (Though this would also break most encryption methods...) In other words, **could P = NP?**

Alternatively, if we can't have this nice thing, wouldn't it be great to prove that it's impossible for some problem like `subsetSum` to have a polynomial-time solution, so we can stop wasting time trying to find one? In other words, **can we show P != NP?**

This question of whether or not P = NP is one of the most important problems in computer science. It's also one of the seven Millenium Prize problems.

How can we show P = NP?

If you want to demonstrate that P=NP, you need to show that **all problems in NP are also in P**.

To make this easier, computer scientists have identified a set of problems called **NP-Complete** that can be mapped to each other.

If you find a solution to an NP-Complete problem (like `subsetSum`), you can use it to generate a solution to any other NP problem in **polynomial time**. If you find a polynomial solution to `subsetSum`, it works for all NP problems!

How can we show P != NP?

If you want to demonstrate that P != NP, you need to prove that at least one NP problem **cannot be solved in polynomial time**.

How do we prove that it's impossible to find a better solution? You need to consider all possible situations so you don't miss an unusual, clever algorithm. Writing proofs like this is a large part of theoretical computer science.

Most computer scientists think that P != NP, but proving this is very tricky.

For now, whether P = NP or not remains a mystery...

For a given algorithm…

Can we ensure that it is correct?

Recent Boeing 737 Crashes

Two Boeing 737 jets have crashed in the past six months, both seemingly due to technical errors in the automation system.

Why did this happen? Shouldn't we be able to write code that we can be **100% sure** will work correctly?

The Perfect Test Function

Goal: we want to write the ideal test function, one that will verify whether a given program returns the correct result on **all possible inputs**. Let's call it `testAll(f)`.

To test all possible inputs, we first need to make sure we don't get infinite loops/infinite recursion on any input to our function. If we don't, `testAll(f)` will take forever to run!

```
def testAll(f):
    if not alwaysHalts(f):
        return False
    ...
```

The Perfect Halting Function

New goal: write the program `alwaysHalts(f)`, which returns True if f 'halts' (stops and returns a value) on all possible inputs to f.

To solve this, we must write the program `halts(f, inp)`, which returns True if f halts on the given function and input, and False otherwise.

```
def alwaysHalts(f):
    for inp in allPossibleInputs(f):
        if not halts(f, inp):
            return False
    return True
```

The Halting Problem: can we write a program to do this?

No.

Let's use a Proof by Contradiction to show why.

Proof by Contradiction

To show that the program `halts()` cannot exist, we only need to find one program f and one input inp such that it is **impossible** for `halts(f, inp)` to return the correct result.

To do this, let's design a program, `breakHalts(f)`, which **uses `halts` to break itself**.

Here's the big question: **does `breakHalts` halt when given itself as an input, or not?**

```
def breakHalts(f):
    inp = f
    if halts(f, inp):
        print('Running forever!')
        while True: pass
    else:
        print('Halting!')
        return
```

Case one: breakHalts halts

Assume that `breakHalts(breakHalts)` **does** halt.

Therefore, `halts(breakHalts, breakHalts)` should return True, and we enter the if case.

Then we enter an infinite while loop… and the program never halts.

CONTRADICTION!

```python
def breakHalts(f):
    inp = f
    if halts(f, inp):
        print('Running forever!')
        while True: pass
    else:
        print('Halting!')
        return
```

Case two: breakHalts loops forever

Assume that `breakHalts(breakHalts)` will not halt, and will instead loop forever.

Therefore, `halts(breakHalts, breakHalts)` should return False, and we enter the else case.

But then we immediately return, which means the program halts!

CONTRADICTION!

```
def breakHalts(f):
    inp = f
    if halts(f, inp):
        print('Running forever!')
        while True: pass
    else:
        print('Halting!')
        return
```

Some Functions are Uncomputable

We just showed that it is impossible to write the program `breakHalts` and call it on itself. But the program we used only had one unusual bit of code- the call to `halts()`. Therefore, **it is impossible to write the program `halts()`**.

Since we can't write `halts()`, we can't write `alwaysHalts()`, and since we can't write `alwaysHalts()`, we can't write `testAll()`. These problems are **uncomputable**- we cannot write a program to compute them, no matter how clever we are.

Takeaway: there are some programs that are simply impossible to write!

We still need to write programs.

Can we make them fast and correct?

How do you solve a problem in NP?

Option 1: only run your function on small inputs. (Then bad efficiency doesn’t matter)

Option 2: use heuristics to find a ‘good-enough’ solution

Example: scheduling final exams at CMU

Big Idea: if you can identify when your algorithm is non-polynomial, you can find workarounds to deal with it!

How do we verify that our code works?

We can’t write a universal test function for code.

But we can **prove** that certain functions will behave as expected on certain classes of inputs.

We can also use **contracts** to ensure that functions only accept certain types of input and only return certain types of output.

Big Idea: test your code well and often and it will be robust, if not perfect.