Genome 540 discussion

February 20th, 2025
Joe Min

Genome Sciences
UNIVERSITY OF WASHINGTON

Agenda

Characterizing problem difficulties

Problem reduction

Turing machines

P vs. NP

Characterizing problem difficulties

What does “problem difficulty” mean?

We have seen:

- Time/space complexity of algorithms (“Big O” notation)

Problems that can be solved in similar Big O time can be grouped as having similar difficulties

Familiar problem groups

Constant time: $O(1)$

- Accessing an element in an array

Linear time: $O(n)$

- D-segments; finding max element in an array

Quadratic time: $O(n^2)$

- Brute-force sorting (e.g., insertion sort)

Unfamiliar problem groups

Factorial time: O(n!)

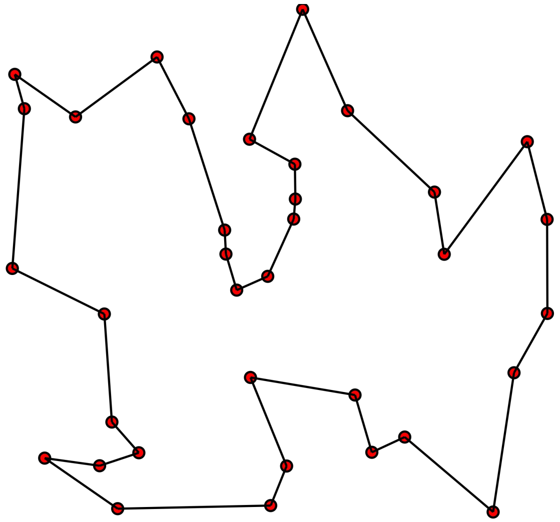
Exponential time: $O(k^n)$; e.g., $O(2^n)$

Examples??

- A classic example is the Traveling Salesman

Traveling salesman problem

“Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?”



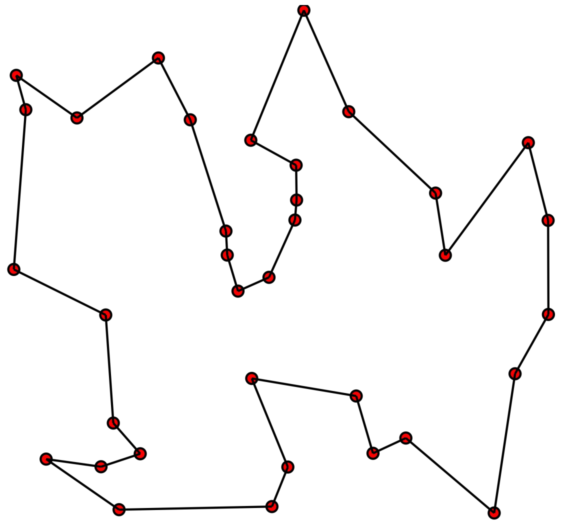
https://en.wikipedia.org/wiki/Travelling_salesman_problem#/media/File:GLPK_solution_of_a_travelling_salesman_problem.svg

Traveling salesman problem: brute force algorithm

For a route of n cities:

- There are n! orderings of the n cities
- Try each ordering of cities (e.g., for cities A, B, C, try all 3! permutations) and keep track of the shortest route

Time complexity: O(n!)

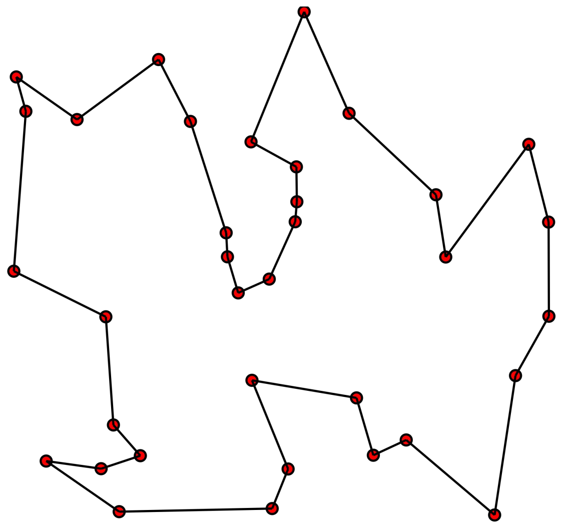


https://en.wikipedia.org/wiki/Travelling_salesman_problem#/media/File:GLPK_solution_of_a_travelling_salesman_problem.svg

Traveling salesman problem: dynamic programming

Dynamic programming can save us!

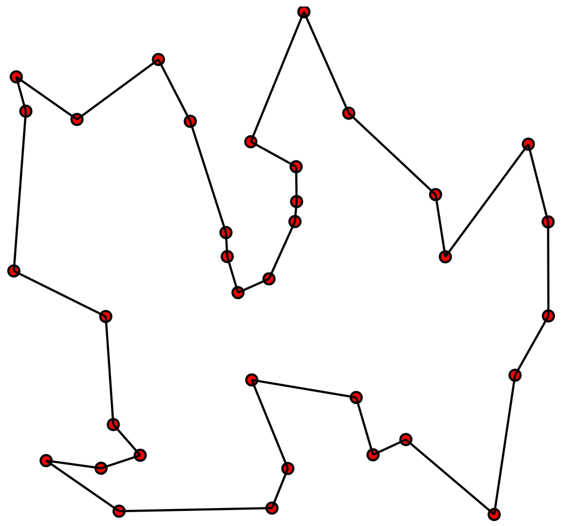
- Split up the set of cities into subsets and save shortest routes
- Each city can be included or not, resulting in 2^n subsets that can each be solved in $O(n^2)$



https://en.wikipedia.org/wiki/Travelling_salesman_problem#/media/File:GLPK_solution_of_a_travelling_salesman_problem.svg

Traveling salesman problem: dynamic programming

Solving subsets



- In each 2^n subsets, S, consider each of the n possible ending cities, j
- To minimize the route to j, we must minimize the route to all n possible ending cities in S - {j}

(2^n subsets)*(n ending cities)*(n ending cities) = $O(2^n*n^2)$

https://en.wikipedia.org/wiki/Travelling_salesman_problem#/media/File:GLPK_solution_of_a_travelling_salesman_problem.svg

Problem reduction

Reducing one problem to another

Let’s say we want to find the median in an unsorted list of integers

A brute force algorithm would be to look at the value, V1, of the first index in the list, compare it to all other values vi. If the number of vi > V1 is the same as vi < V1, then this is the median. If not, start over with the next value, V2

So, we have an upper bound of $O(n^2)$ on time complexity

Reducing one problem to another

But! If we could just have a sorted list, then finding the median is O(1), as we just go to the middle index and retrieve that value

We know sorting can be done in O(n*logn) time, so we can reduce the problem of “finding a median” to the problem “sorting a list”; that is, finding a median is *at least* as hard as sorting a list, giving us a new upper bound of O(n*logn)

Reducing one problem to another

If we could continue to reduce the problem to even easier problems (e.g., that could be solved in O(n) or O(1) time), then we could bring down the upper bound even further

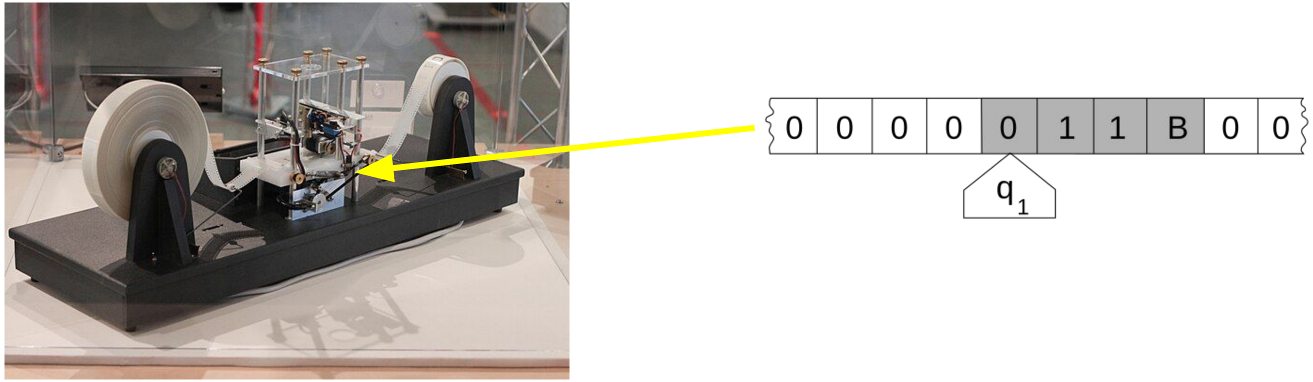
But because we haven’t found a way to reduce the sorting problem any further, O(n*logn) is the best we can do

Turing machines

What is a Turing machine?

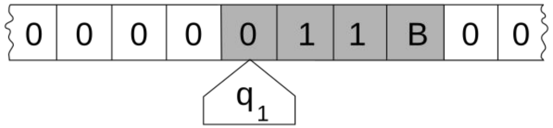
In his 1948 essay, "Intelligent Machinery", Turing wrote that his machine consists of:

> ...an unlimited memory capacity obtained in the form of an infinite tape marked out into squares, on each of which a symbol could be printed. At any moment there is one symbol in the machine; it is called the scanned symbol. The machine can alter the scanned symbol, and its behavior is in part determined by that symbol, but the symbols on the tape elsewhere do not affect the behavior of the machine. However, the tape can be moved back and forth through the machine, this being one of the elementary operations of the machine. Any symbol on the tape may therefore eventually have an innings.[15]
>
> —Turing 1948, p. 3[16]



https://en.wikipedia.org/wiki/Turing_machine

What is a Turing machine?



Similar ideas to an HMM:

- Keeps track of some internal state
- Rules govern transitions to new states

Key differences:

- Machines are used to perform calculations; HMMs are probabilistic representations of sequences

What can Turing machines do?

Given enough tape (memory) and retained states (internal rules of computation), a Turing machine can, almost by definition, compute anything that can be described as a deterministic algorithm

- Given the same input, it will always produce the same output

What *can't* Turing machines do?

There are fundamental limitations to what is computable…

- because it requires infinite memory like calculating the digits of Pi (an infinitely long list of ints)
- or because it requires infinite steps like a simple counter inside a “while True” loop

What *can't* Turing machines do?

A more interesting uncomputable problem is the classic Halting Problem, which basically asks:

Given some algorithm/program **P** that takes input **x**, is there some other program **H(P, x)** that can output whether or not **P** will finish running given x?

- E.g., **H(P, x)** returns True if **P** will finish when given **x**

The halting problem

Why is the halting problem uncomputable? High level proof by contradiction, assuming a good **H(P, x)** exists

Let **P** now do the opposite of what **H** says, e.g.,:

- If **H(P, x)** returns True, **P** loops forever
- Else, **P** halts

Then **H** can not be correct when returning True

The halting problem

This is a contradiction! Thus the assumption that a good **H(P, x)** exists must be false; thus the halting problem cannot be solved as a deterministic algorithm.

This means that the halting problem is *not* computable by a Turing machine! AND there are some problems that are simply uncomputable for all inputs

Turing completeness

Any general programming language that can simulate a Turing machine is considered to be “Turing complete” (TC)

Most modern coding languages (e.g., Python and C++) are Turing complete, but also things like Excel are Turing complete; things like HTML or XML, however, are more like data structures than languages, and are not TC

P vs. NP

So far…

Some algorithms are impossible to implement for all inputs

The algorithms we *can* implement have various different time complexities

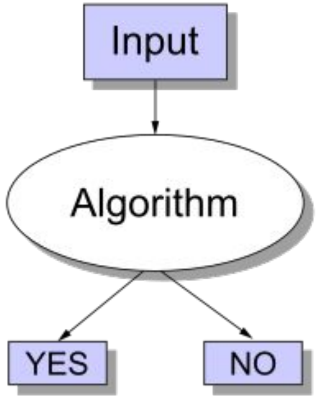
Some of these algorithms can be reduced to easier ones

Now, finally! P vs. NP

What is P?

“P” stands for polynomial, and P represents all decision problems that can be solved by a Turing machine in polynomial time

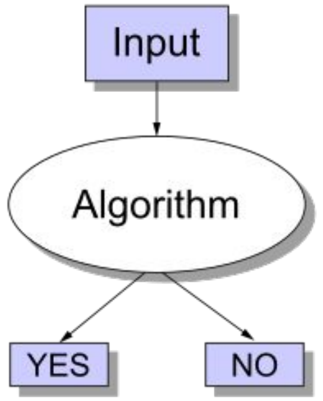
This is the world in which all of our algorithms for class live



https://en.wikipedia.org/wiki/Decision_problem

What is NP?

“NP” stands for nondeterministic polynomial, and NP represents decision problems where, if the answer is “yes”, can be easily verified in polynomial time



https://en.wikipedia.org/wiki/Decision_problem

Example of NP

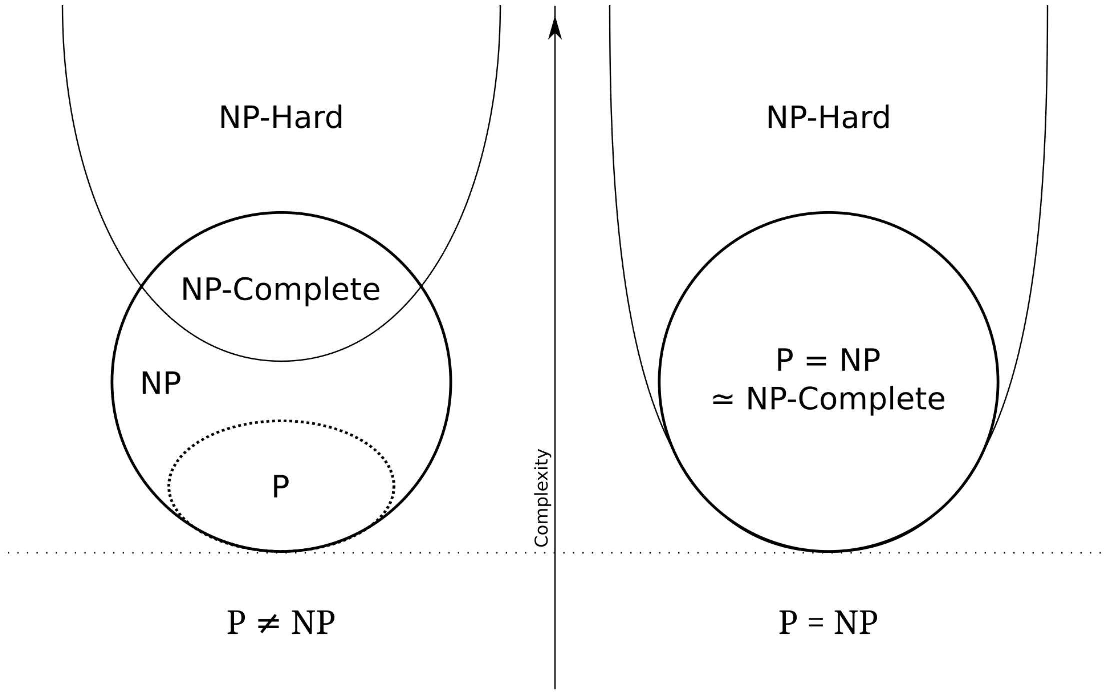
Consider a set of integers {-3, 1, 2, 5}, where we want to know if any subset of this set of integers can sum to 0

- Here, the answer is “yes” (-3 + 1 + 2 = 0)

As the set grows linearly, however, the possible number of subsets grows exponentially (not polynomial!)

- HOWEVER given any single subset, we can easily verify in O(n) time if it actually adds up to 0

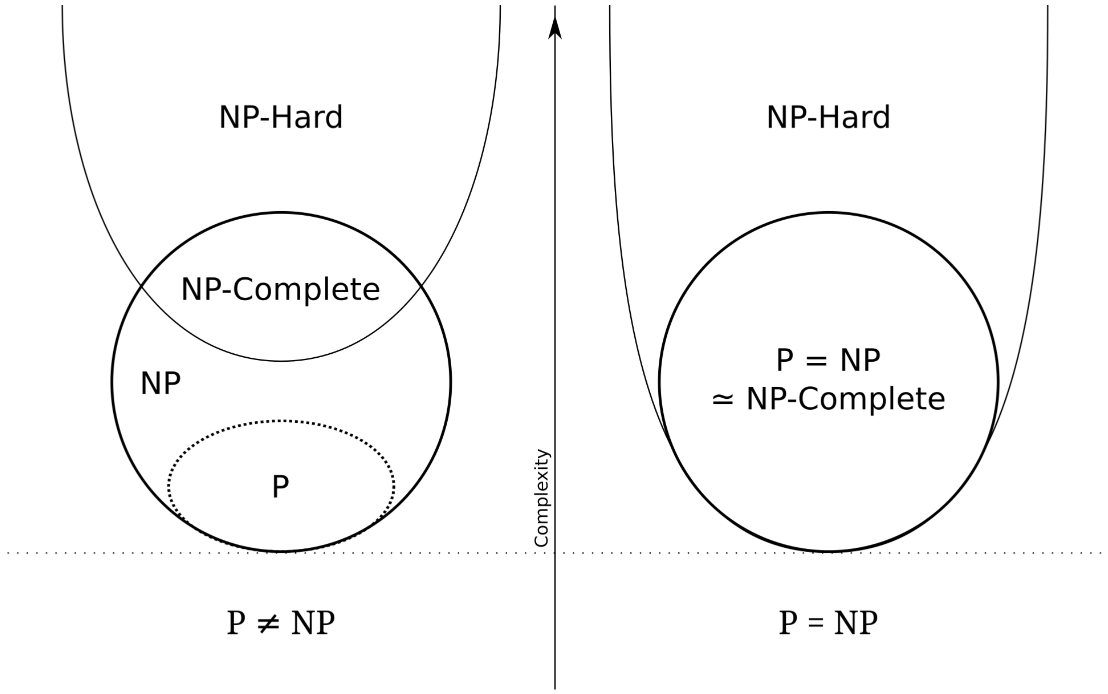
The space of decision problems



NP-Hard means that these problems are *at least* as difficult as the hardest NP problems

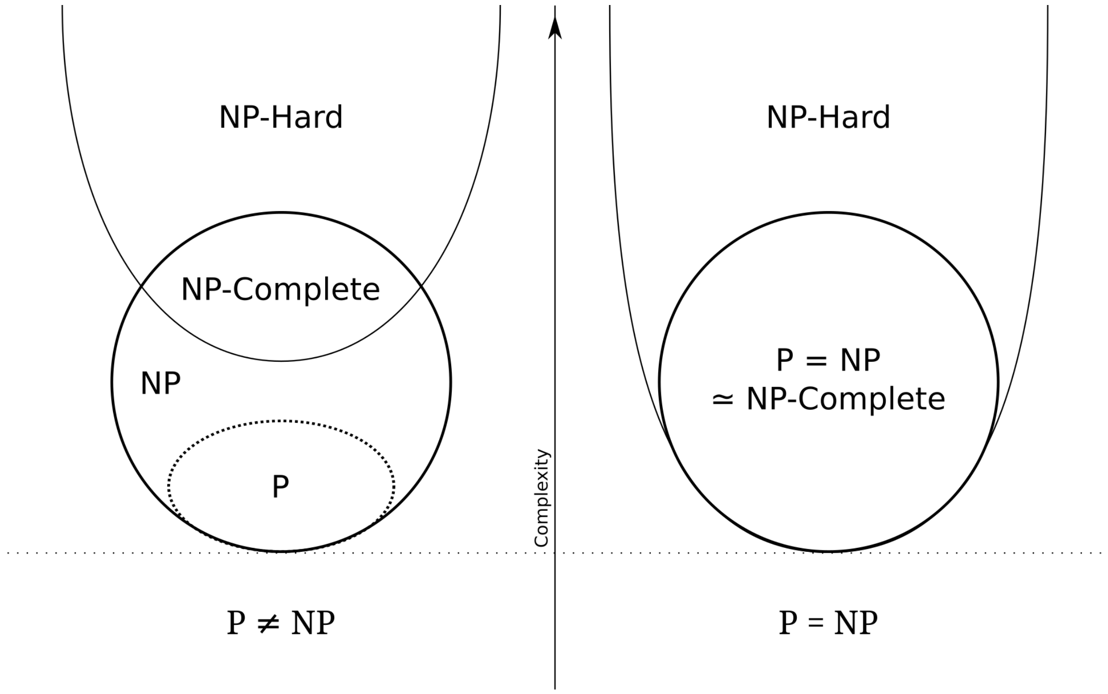
NP-complete means it’s the hardest NP problems AND is NP

The space of decision problems



E.g., the halting problem is NP-hard because it’s harder (impossible?) than the hardest NP problem, and is not NP

The space of decision problems



Many (most?) people think that P != NP

…but what if it was??

If P = NP

Then any NP problem can be reduced to be solved in polynomial time

- E.g., the traveling salesman problem (at least, the decision version where we ask “is this the shortest route?”) could be solved in polynomial time
- Would break (some) cryptography, which basically relies on problems being too hard to solve

Office hours

Reminder:

Homework 6 is due Sunday, February 23rd at 11:59pm!