1 | sources

1.1 | gentle introductions

1.1.1 | https://en.wikipedia.org/wiki/Computational_complexity_theory

1.1.2 | https://complexityzoo.net/Petting_Zoo

2 | overview

2.1 | computational complexity theory studies how ”difficult” a problem is

2.1.1 | importantly, not how ”good” an algorithm is... this field deals with all algorithms that solve a given problem

2.2 | key concepts

2.2.1 | types of problems

2.2.2 | Turing machines

2.2.3 | reducibility

2.2.4 | complexity classes

2.2.5 | hierarchy

2.3 | key problems

2.3.1 | P vs NP

3 | vocab and definitions def

3.1 | DTM, deterministic Turing machine

A Turing machine with one infinite tape and a state function that has exactly one output for each (tape value, machine state) input. (Not a formal definition).

4 | complexity classes

4.1 | P

Problems that can be solved in polynomial time using a deterministic Turing machine (DTM).

In practice, problems with large-degree solutions usually have smaller-degree solutions discovered later, so the division between P and other problems ’has turned out to be somewhat ”natural”’.

4.2 | NP

4.3 | NP-complete

5 | flows

5.1 | Wikipedia computational complexity theory

5.1.1 | computational problems

1. problem instances

   A problem describes the problem. the actual "numbers" that describe a specific problem is called a problem instance. sorting a list is a problem, sorting *this* list is a problem instance.

2. representing problem instances

   formally strings of characters from alphabets. The input size is the length of the string. Different representations can be chosen but it should be trivial (fast) to convert from one to the other.

3. decision problems (most basic type)

   Generally, given an input, the output is either yes (accept) or no (reject). For example, deciding whether a graph is connected or not.

   (a) it can be thought of as a "formal language" toexpand

4. function problems

   Very general: a function problem 'is a computational problem where a single output (of a total function) is expected for every input, but the output is more complex than that of a decision problem'. Basically calculate a non-binary function.

   Examples: traveling salesman, integer factorization.

   However, all function problems can be modeled as decision problems: For some function $f(*args) \rightarrow ans$, it can be modeled as the decision problem of whether $(*args, ans)$ is a valid output.

   (a) but does this really work? how can a decision TM be used to compute the function output efficiently? toexpand

5. size of an instance

   Size is usually the length of the input. The complexity is a function of the input size, usually representing the worst case time or space (or any other complexity measure) required for any input size.

5.1.2 | machine models and complexity measures

1. Turing machine

   standard Turing machine stuff. its very general. Many types of turing machines (probabilistic, nondeterministic, quantum, etc) are used to define different complexity classes.

2. other machine models toexpand

   Other non-standard Turing machines are used, but the idea is that they aren't actually any better, somehow?

3. Complexity Measures

   Usually time or space, but any complexity measure that satisfies Blum's complexity axioms can be used. Examples include: communication complexity, circuit complexity.

   Also constant factors don't really matter. And its usually the worst case.

   Importantly, complexity measures are also a function of the type of Turing machine used, since some Turing machines are better in some scenarios.

   (a) blums complexity axioms toexpand

4. best/worst/average case

   We generally talk about worst case complexity, but some algorithms have good average-case which is good enough (eg. quicksort). Generally, best-case < average-case < amortized analysis < worst-case.

5. upper and lower bounds for problems

   Importantly, this is **not an upper or lower bound for an algorithm**. Instead, for problems in general, it's relatively easy to decide an upper bound (which is just the worst case complexity of any correct algorithm), but a lower bound is difficult (since it must involve algorithms that haven't been discovered yet).

5.1.3 | complexity classes

1. dependencies

   Complexity classes are a function of the following factors

   (a) problem type
   { decision, function, counting, optimization, promise, etc }

   (b) computation model
   { deterministic Turing machine, non-deterministic, Boolean circuits, quantum TM, monotone circuits, etc }

   (c) bounded resources
   { polynomial time, logarithmic space, constant depth }

2. an example definition

   > The set of decision problems solvable by a deterministic Turing machine within time f(n). (This complexity class is known as DTIME(f(n)).)

   However, using a concrete function $f(n)$ is often computational-model-dependent, but the Cobham-Edmonds thesis states that 'the time complexities in any two reasonable general models of computation are polynomial related.'

   This suggests that all if we want to be machine-independent, all polynomial problems are roughly the same and belong in the same class: P (for decision problems) and FP (for function problems).

   (a) why are there different classes if decision and function problems are the same-ish? dunno toexpand

3. important complexity classes

   A nice list here but the complexity petting zoo is more friendly.

4. Hierarchy theorems toexpand

   We would like to establish a strict containment hierarchy within classes (but between different eg. polynomial functions). This does that, apparently?

5. Reduction

   Many problems can be turned into other problems in their class, which provides an upper bound on the difficulty of the problem.

   There are many types of reductions, but the most common type is the polynomial-time reduction which means the reduction takes polynomial time. If you take a non-polynomial reduction to turn a problem into a polynomial problem, then you haven't proven anything.

   (a) hardness and completeness

   A problem X is hard for a class C if every problem in C can be reduced to X. A problem X is complete for C if it is hard for C it is in C. NP-complete problems are the "most difficult problems in NP" because other problems can be reduced to them.

   Being able to reduce a hard problem to another problem shows that that other problem is just as hard, by contradiction. Similarly, being able to reduce a hard problem to a known easy one collapses the hierarchy.

5.1.4 | important open problems

1. P vs NP

   If any NP-complete problem can be reduced (polynomially) to a P problem, then many NP problems would be solvable in polynomial time. There are many NP problems that we would like to solve efficiently, so this would be a big deal.

   In fact, many of the other 'important open problems' are important because they would show that $P \neq NP$.

2. NP-indeterminate problems (in NP but not in P nor NP-complete) toexpand

   some theorem shows that if P neq NP then there are NP-indeterminate problems. If we show that there are none, then that proves P = NP. Some unclassified problems (graph isomorphism problem, integer factorization problem) being NP-complete would 'collapse the polynomial hierarchy.' ?????

3. separations between other complexity classes

   There are many classes that are improper subsets of each other. If any of those relations can be shown to be a proper subset, then classes on either side would be unequal. For example, many such relations exist between P and NP and showing that one of those relations is a proper subset relation would prove that P != NP. Or, proving that two classes (eg. P, PSPACE) are equal would squish all classes in between into one (in this case, showing that P = NP).

5.1.5 | Intractability

Meaning "not handleable". The Cobham-Edmonds thesis suggests that all polynomial problems are tractable. However, in the real world, specific numbers matter (N^{15} is much worse than 0.0001^N)

5.1.6 | continuous complexity theory toexpand

Something about continuous functions or analog logic.

5.1.7 | History

1. Many foundations laid, eg. Turing machine in 1936 which allowed for analysis of various algorithms.
2. First systematic study attributed to Juris Hartmanis and Richard E. Stearns in "On the Computational Complexity of Algorithms" (1965)

3. Edmonds (Cobham-Edmonds thesis) suggests polynomial problems are ”good” (1965)
4. other studies of problems with bounded resources in the previous few years
5. Blum axioms for complexity measures (1967), and the ”speed-up theorem”
6. 1971 Stephen Cook and Leonid Levin proved existance of practically relevant NP-complete problems
7. Richard Karp (1972) showed 21 relevant and NP-complete problems (op)