8. Limitation of Algorithm Power

DATA STRUCTRES AND ALGORITHMS [19ECSC201]

PRAKASH HEGADE

SCHOOL OF CSE | KLE Tech

Important Note:

The contents of this notes are tailored from the data available through Wikipedia, textbooks (mentioned in lesson plan and more) and other relevant references. **This notes presents the basic definition notes only. Of-course the chapter has more.**

Syllabus:

Undecidability, Lower Bound Arguments, P and NP Classes, P vs NP, NP-Hard and NP-Complete, Examples.

Undecidability

A problem is decidable if we can construct a Turing machine which will halt in finite amount of time for every input and give answer as 'yes' or 'no'. A decidable problem has an algorithm to determine the answer for a given input.

A problem is undecidable if there is no Turing machine which will always halt in finite amount of time to give answer as 'yes' or 'no'. An undecidable problem has no algorithm to determine the answer for a given input.

R, real numbers is uncountable.

In order to show R is uncountable, we show that no correspondence exists between N and R (A set A is countable if either it is finite or has same size as N, where is N is set of natural numbers.)

Let us assume that a corresponding function f exists which maps R to N. Let f(1) = 8.1234…, f(2) 77.5555…., f(3) = 0.1367… and so on. To prove R is uncountable, we construct an x, that is not mapped to N. To ensure that x is not equal to f(1), we let the first digit of x be anything different from the first fractional digit 1 of f(1) = 8.1234. To ensure that x is not equal to f(2), we let the second digit of x be anything different from the second fractional digit 5 of f(2) = 77.5555… and so on. We thus obtain a x which is not mapped to N yet. With considered idea, we can always find a new x which is not mapped to N yet. Hence, R is uncountable.

Halting Problem

"Given a problem and input to it, determine whether the program will halt on that input or continue working indefinitely on it."

Alan Turing proved in 1936 that a general algorithm to solve the halting problem for all possible program-input pairs cannot exist.

A is an algorithm that solves the halting problem.

A(P, I) = 1, if the program halts

= 0, if program P does not halt

Input P to itself, say Q.
If A(P, P) = 0, Q(P) halts
 = 1, Q(P) does not

Now supply Q(Q). No matter what Q does, it is forced to do opposite, which obviously a contradiction.

Lower Bound Arguments

When we want to ascertain the efficiency of an algorithm with respect to other algorithms for the same problem, it is desirable to know the best possible efficiency any algorithm solving the problem may have. Knowing such lower bound can tell us how much improvement we can hope to achieve in our quest for a better algorithm for the problem in question.

Trivial Lower Bounds
The simplest method of obtaining a lower bound class is based on counting the number of items in the problems input that must be processed and the number of output items that need to be produced.
Example: generating all permutations of n distinct items must be in n! because the size of output is n!

Information-Theoretic Arguments
This method seeks to establish a lower bound based on the amount of information it has to produce.
Example: The well-known game of deducing a positive integer between 1 to n selected by somebody by asking that person questions with yes/no answers. The amount of uncertainty that any algorithm solving this problem has to resolve can be measured by logn, the number of bits needed to specify a particular number among the n possibilities. We can think of each question as yielding at most one bit of information about the algorithms output, the selected number. Consequently, any such algorithm will need at least logn such steps before it can determine its output in worst case.

Adversary Arguments
It is based on pushing the algorithm down the most time-consuming path. A lower bound is obtained by measuring the amount of work needed to shrink a set of potential inputs to a single input among the most time-consuming path.
Example: Number of comparison to merge two lists in worst case in 2n-1.

Problem Reduction
We reduce the problem to problem with well-established lower bound.
Example: the lower bound for searching in a sorted array is logn. The lower bound for sorting is nlogn.

P, NP, NP-Complete and NP-Hard

P

P is a complexity class that represents the set of all decision problems that can be solved in polynomial time. That is, given an instance of the problem, the answer yes or no can be decided in polynomial time.

Example:
Given a connected graph G, can its vertices be coloured using two colours so that no edge is monochromatic?
Algorithm: start with an arbitrary vertex, color it red and all of its neighbours blue and continue. Stop when you run out of vertices or you are forced to make an edge have both of its endpoints be the same color.

NP

NP is a complexity class that represents the set of all decision problems for which the instances where the answer is "yes" have proofs that can be verified in polynomial time.

This means that if someone gives us an instance of the problem and a certificate (sometimes called a witness) to the answer being yes, we can check that it is correct in polynomial time.

NP-Complete

NP-Complete is a complexity class which represents the set of all problems X in NP for which it is possible to reduce any other NP problem Y to X in polynomial time.

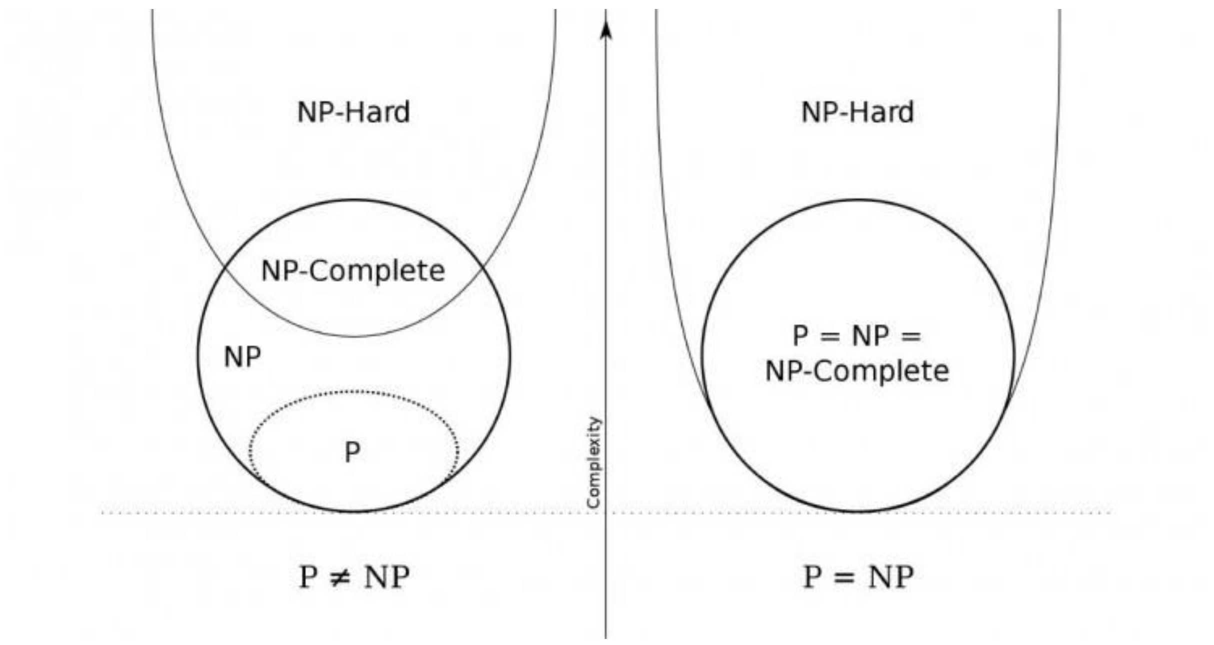
Intuitively this means that we can solve Y quickly if we know how to solve X quickly. Precisely, Y is reducible to X, if there is a polynomial time algorithm f to transform instances y of Y to instances x = f(y) of X in polynomial time, with the property that the answer to y is yes, if and only if the answer to f(y) is yes.

NP-hard

Intuitively, these are the problems that are at least as hard as the NP-complete problems. Note that NP-hard problems do not have to be in NP, and they do not have to be decision problems.
The precise definition here is that a problem X is NP-hard, if there is an NP-complete problem Y, such that Y is reducible to X in polynomial time.

But since any NP-complete problem can be reduced to any other NP-complete problem in polynomial time, all NP-complete problems can be reduced to any NP-hard problem in polynomial time. Then, if there is a solution to one NP-hard problem in polynomial time, there is a solution to all NP problems in polynomial time.

If P = NP,


NP-Hard
NP-Complete
NP
P
Complexity
P ≠ NP
NP-Hard
P = NP =
NP-Complete
P = NP

~*~*~*~*~*~*~