G52LAC

P vs. **NP**

Andrew Parkes

http://www.cs.nott.ac.uk/~pszajp/

Overview

Theory of **P** and **NP**

- Definitions and usages
  - DTM and P
  - NDTM and NP
  - Reductions
  - NP-hard vs NP-complete
  - SAT, and other NP-complete problems

- Many papers justify algorithm choices by "*the problem is NP-complete*", and so should know what this means.

Runtimes of Common Algorithms

- Broad groups of time behaviours, (with input size n):
  - "Logarithmic" – O(log n), often "polylogarithmic"
    i.e. polynomial in log n, e.g. O($(\log n)^k$) for some fixed k
  - "Polynomial" – O(n^k) for some fixed k
    (k is not allowed to depend on n)
  - "Exponential" – O(2^{an}) for some fixed a>0
- Problems within these groups are not all "equal"
  - Not all "exponential time problems" are "of the same kind"
    - Even in the same group, problems can be in different complexity classes and so need different kinds of algorithms and heuristics.
  - We can get insights into "problem difficulty" using "reductions"

Introduction to idea of reductions

- Suppose that have a 'library function'
  sort(int[] A)
  that puts the array A into ascending (strictly non-descending) order.
- How can you use it to sort into descending order?

Introduction to idea of reductions

- Two "obvious" options
  - take the negative of all of A, then sort, and take negative again
  - sort into ascending and then reverse the list
  - Both are O(n) extra work
- The problem
  - "sort into descending order"

  can be "quickly" "reduced" to the problem
  - "sort into ascending order"
- If we know "ascending" can be done in O(n log n) time then we know that "descending can also be done in O(n log n)

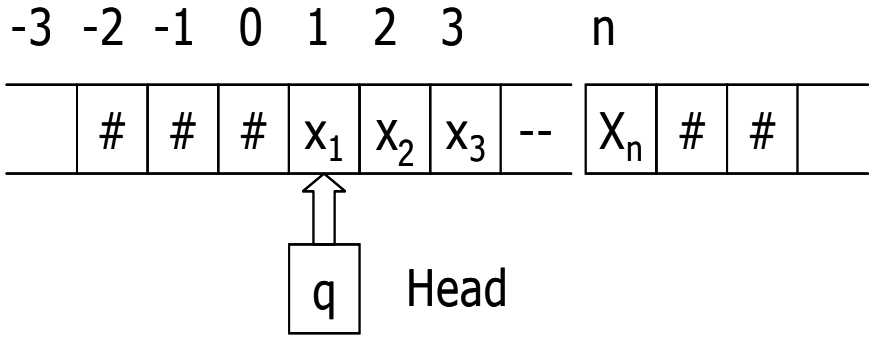
Motivations for P and NP

Observationally (1960s/70s), many practical (O.R.) problems seem to split into two classes:

- "**Efficient**": Exact algorithms with runtimes of with low-order power laws
  - Sorting, Assignment problem, Shortest path, Network flow, (linear programming), etc.
- "**Inefficient**": All complete/exact algorithms had exponential runtime, though incomplete or approximate methods often fast
  - TSP, graph colouring, scheduling, etc.

- Theory of **P** and **NP** was developed to capture this split
  - first need to "define a computer":

Turing Machines (TM)



- P/NP theory is usually phrased using Turing Machines
- A Turing machine is **deterministic** if there is at most one action or 5-tuple for any current state q_k and any current symbol that can occur when the machine is in state q_k.
  - "**At most one choice for the next move**"

Definition of **P**

- **P** is the set of problem classes that can be solved using some Deterministic TM in polynomial time p(n)

Standard Decision Problems

It is usual to work in terms of decision problems.

It is standard to write decision problems in the format

"INSTANCE" the input data

"QUESTION" the simple yes/no question that an algorithm must answer

To help motivate and explain **NP** will use "SUBSET-SUM":

Subset Sum

INSTANCE:

A set S of n positive integers a[i] i=1,…,n

A target integer K

QUESTION:

Is there a subset T of S such that the elements of T sum up to precisely K

sum(i in T, a[i]) = K

Solving SUBSET SUM

INSTANCE:

A set S of n positive integers a[i] i=1,…,n

A target integer K

QUESTION:

Is there a subset T of S such that the elements of T sum up to precisely K

sum(i in T, a[i]) = K

Questions:

1. Give an algorithm to verify a claimed solution, T, to this. What is its complexity?
2. Give an algorithm to solve SUBSET-SUM. What is its complexity?

Solving SUBSET SUM

Verifying a claimed solution is only O(n)

Algorithm to solve subset sum:

```
For all subsets T of S
    if (sum(T) == K ) return true
return false
```

Complexity: $O(2^n)$

Advanced algorithms only improve this to $O(2^{an})$, for some $0 < a < 1$.

Solving SUBSET SUM

Poly-time "Algorithm" to solve subset sum:

```
For each element i:
        either:
                place a[i] in T
        or:
                place a[i] not in T
Check when finished if   sum(T) == K,
if so return "yes", else return "no"
```

- Catch: This is not a proper algorithm as there is no prescription for which "either-or" option to take.
- But: the overall answer to the SUBSET-SUM is "yes" iff some sequence of "either / or" choices will give "yes"

Non-Deterministic Turing Machine

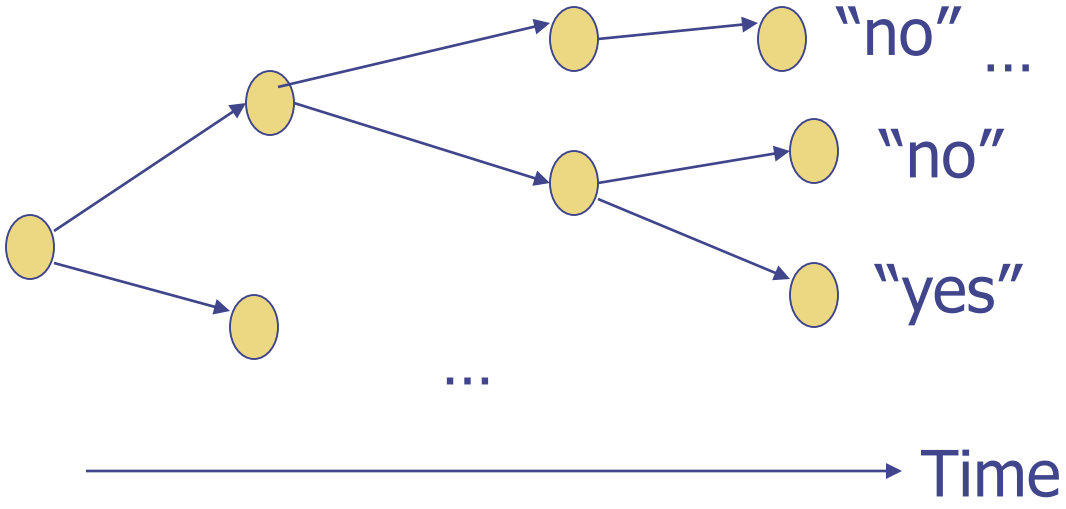
- From given state of the machine, and head position and tape symbol there can be more than one option for the next action
  - One action will be selected
  - but it is not specified which one
- **IMPORTANT: This is not the same as "probabilistic" or "random"**
  - **there is no implication of making the choice "with uniform probability", or any probability distribution at all.**
  - **Closer to being "the opposite of random", e.g. "super-lucky" or "super-intelligent" – "always does the best thing"**

NDTM for Decision Problems

- A NDTM working on a decision problem outputs
  - "yes" iff **some** sequence of non-deterministic decisions gives a "yes"
  - "no" iff **all** sequences give "no"
- Note the very strong asymmetry:
  A single "yes" is enough to outweigh any number of "no"
- It is not directly physically realisable (as far as is known?); but a very convenient device for defining useful complexity classes

Execution tree of NP

- A tree that succeeds:



"Advice Strings" or "oracles"

Poly-time "Algorithm" to solve subset sum:

Suppose have an extra poly-length input string s of {T,F}

```
For each element i:
        if getNext(s)
                place a[i] in T
        else
                place a[i] not in T
Check when finished if   sum(T) == K,
if so return "yes", else return "no"
```

- If the overall answer to the SUBSET-SUM is "yes" then some string s will guide to that solution
  - like an "oracle" or "satnav"
  - 's' guides the execution – but in itself does not solve the problem
- If the instance has no solutions, then all strings s must give 'no'
  - (So is not 'just cheating').

Definition of NP

- **NP** is the set of problems that **can** be solved using some Nondeterministic TM in **P**olynomial time

- Note that this is a positive statement:
  - there exists a program, but it is allowed to use non-determinism

Definition of NP

- A problem is in **NP** if and only if there **is** some **P**olynomial time algorithm for it on some **N**DTM
- The non-deterministic algorithm is essentially a sequence of "good guesses"
  - Just need that verifying a "guessed solution" is in **P**
- Note: NP does not stand for "Non-Polynomial"!!
  - If it did, then P vs. NP would be a nonsense question!
  - **Being "in NP" is a positive statement about an algorithm existing.**

“Obvious” Results

- **P** is a subset of **NP**
- Many common decision problems are “obviously” in **NP**
- Two examples follow
- More examples are listed in the “Appendix”

NUMBER PARTITION

INSTANCE:

set of n positive integers a[i] i=1,…,n

QUESTION:

is there a partition of the numbers into two sets S1 S2, such that each subset sums up to the same value:

sum(i in S1, a[i]) = sum(i in S2, a[i])

"Obviously" in NP, by same reasoning as for SUBSET-SUM

Thought Exercise:

Does this seem (intuitively) easier or harder than SUBSET-SUM?

SAT (boolean satisfiability)

- Variables are atoms, e.g. x1,…,xn
- A literal is an atom or its negation
  - e.g. x1, ¬x1, …
- A clause is a disjunction of literals
- An expression in CNF (clausal normal form) is a conjunction of clauses

E.g. $(x1 \vee \neg x2) \wedge (x3 \vee x2 \vee \neg x1)$

SAT (boolean satisfiability)

INSTANCE:
boolean variables x1,…,xn

a boolean expression G in clausal normal form (that is a conjunction of disjunctions of literals)

QUESTION:

Is there a satisfying assignment for G?

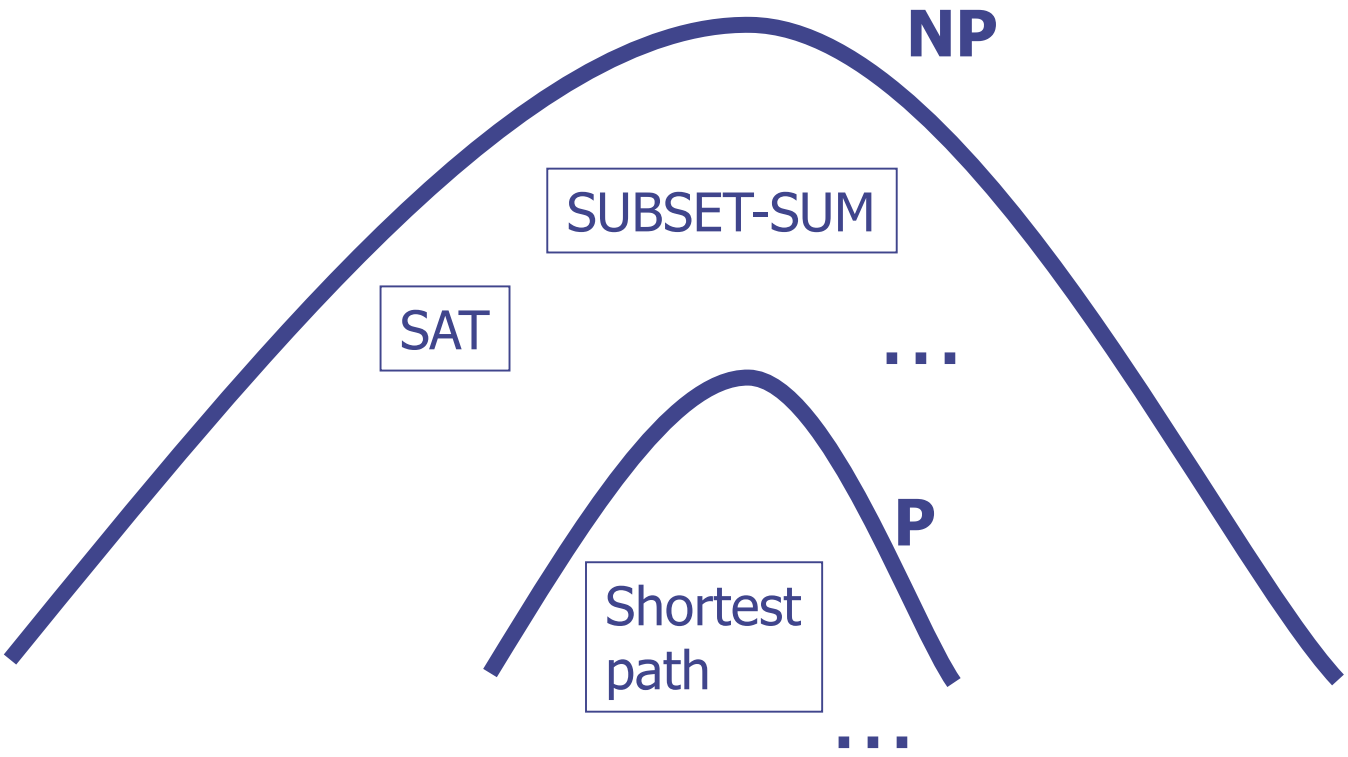
That is, is there truth assignment to the variables such that G is satisfied (evaluates to true)?

"Obviously":

- solvable – just enumerate all 2^n possible assignments
- "in NP" – just use nondeterminism to select the assignment if it exists, then check it

Relations between **P** and **NP**

- Situation so far seems to be



P vs. NP

- It is not known if P = NP
  - $1m prize for solving it – see millennium problems
  - http://www.claymath.org/millennium-problems/p-vs-np-problem

- If P=NP, then any problem in NP that we can solve in polytime on an NDTM can also be solved in polytime on a DTM.

Impact

A major CS result in 1970's, and the theory of "NP-completeness", was that:

many thousands of common but different-looking problems are provably "equally hard", even though we cannot (yet?) prove any of them are "actually hard".

“Reductions”: Overview

- It is common in computability to try to relate together the hardness of two different problem classes
- Suppose have two problem classes X and Y then one might try to make statements such as (roughly)
  - “X requires as much time as Y”
  - “If Y can be solved quickly then so can X”
  - etc

Reductions

- We talked of reducing one problem to another:
  - Given an instance x of problem class X
  - convert to an instance y(x) of problem class y
  - Solve x by solving y(x)
    - "x=yes" if and only "y(x)=yes"
- Reductions give information about relative hardness of problems even in cases that the true hardness is not known
- Finding reductions can be 'tricky'
  - (If doing new problems then it may be important to develop the needed skills.)

Unrestricted "Conversions"?

- If we allow the conversion to be unrestricted then the concept of reduction is not useful for **P** and **NP**
  - Can reduce everything to trivial problems, because the conversion itself can essentially solve the problem
  - We want to be able to reason about what can or cannot be done in polynomial time
- Hence, for questions about **P** and **NP** we decide to restrict ourselves to reductions that can be done in polynomial time

Polynomial Time Reductions

Given **any** instance x of a problem class X

then compute an instance f(x) in problem class Y

- With the conversion done in polynomial time
- Such that the answer is preserved:

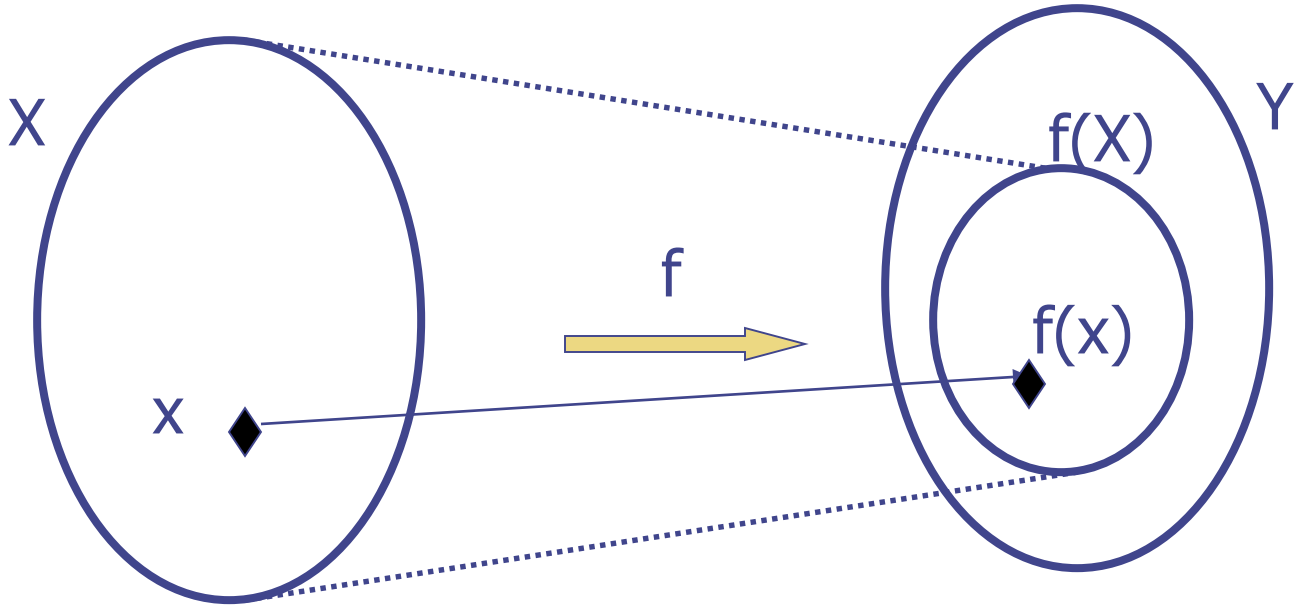
  ans(x)=yes if and only if ans(f(x))=yes

```
boolean solveX( instanceX x ) {
    instanceY y = convert_X_to_Y(x); // f(x)
    return solveY( y );
}
```

Function `convert_X_to_Y' must run in polytime

Remarks

- The mapping is “into” but does not need to be “onto”
- Many instances of class Y might not arise from any instance x of class X



Definition: **NP-hard**

- Suppose that Z is some problem class
- If we can (in polynomial time) reduce **ANY** problem in **NP** to Z then we say that Z is **NP**-hard
- It means that 'Z' is 'as bad as it gets'
  - If 'Z' turns out to be in **P**, then all problems in **NP** are also in **P**, i.e. **P**=**NP**
- We can convert **any** problem within **NP** to 'Z'
  - Note: it does **NOT** need to be the case that Z itself in NP
    - many problems are NP-hard because they are "far more powerful than NP"

Next Steps?

- Using reduction we aim link the hardness of different problems:

  'X is **NP**-hard' therefore 'Y is **NP**-hard'

- 'X' can solve all problems in **NP**
  therefore, also
  'Y' can solve all problems in **NP**

NP-complete

- A problem class X (such as SUBSET-SUM) is said to be **NP-complete**, if and only if both of the following are satisfied:
  - X is **in NP**,
    - all instances of X can be solved in nondeterministic polynomial time
    - membership of NP is usually demonstrated directly (and fairly easily)
  - X is **NP-hard**
    - all problems within NP (not just X) can be solved by converting them (in polytime!) to some instance of X
    - this is usually done by showing (polytime) reduction from some known NP-complete problem to X

Caution:

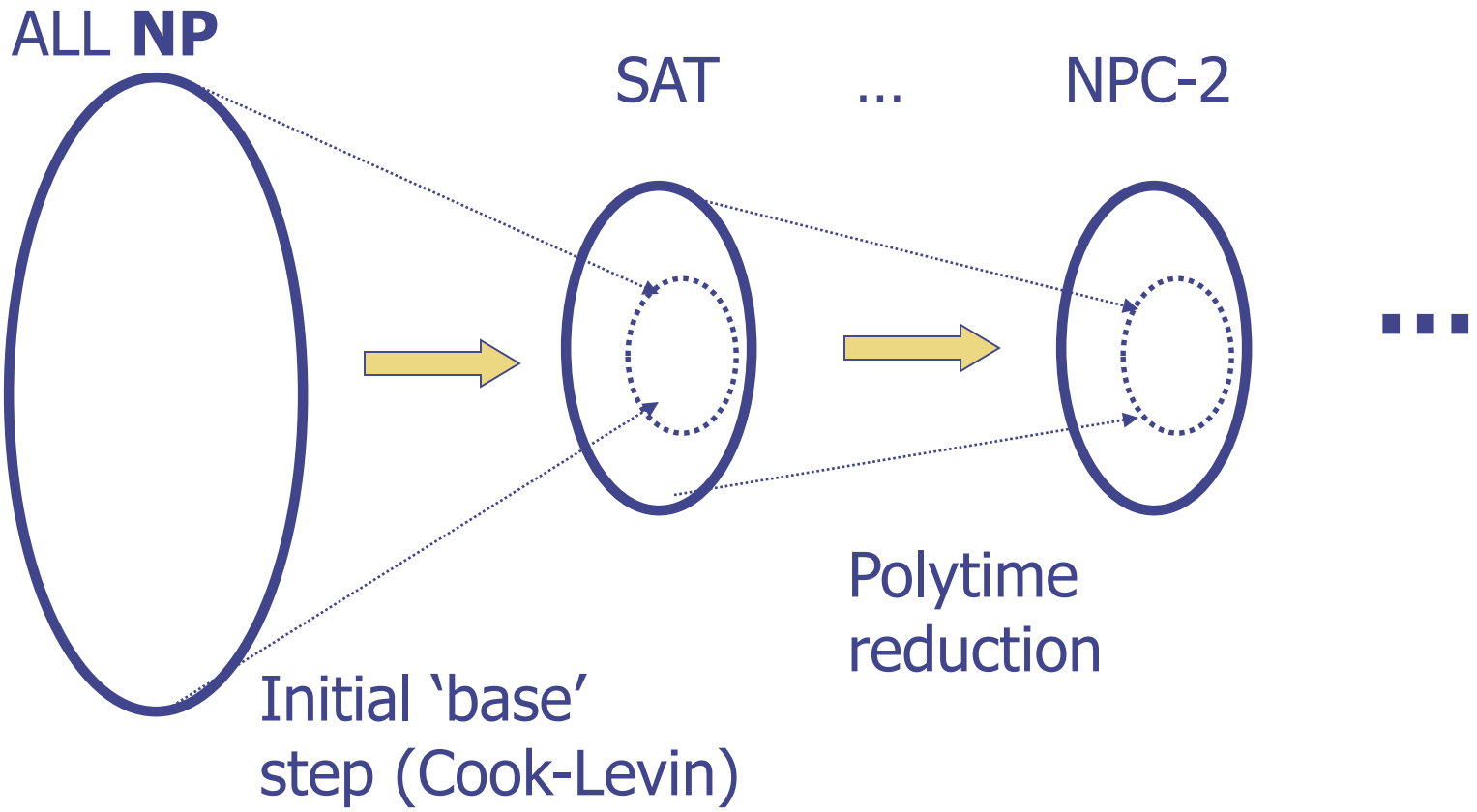
- Do not confuse "NP-hard" and "NP-complete"
  - **A problem being NP-hard does not automatically mean it is NP-complete**
- For most problem classes and results you see then it will be also complete, but it is not always true.

Starting chains of reductions

- The reductions needs a starting point!
- Cook & Levin (1971) proved that SAT is NP-hard
  - SAT is clearly in NP
    - Since then, people prove NP completeness by reduction chains starting from SAT
  - The proof works by showing that
    - any polytime Non-deterministic (Turing) machine running in polytime converts to some polysize SAT formula
    - existence of an accepting execution path of the NDTM corresponds to a satisfying assignment of the SAT formula

Chaining between problems

- Consider NP-complete (NPC) problems



Chain reasoning

Given:

- Any problem in NP can be solved by conversion (in polytime) to some instance of NPC-1
- Any problem in NPC-1 can be solved by conversion (in polytime) to some instance of NPC-2

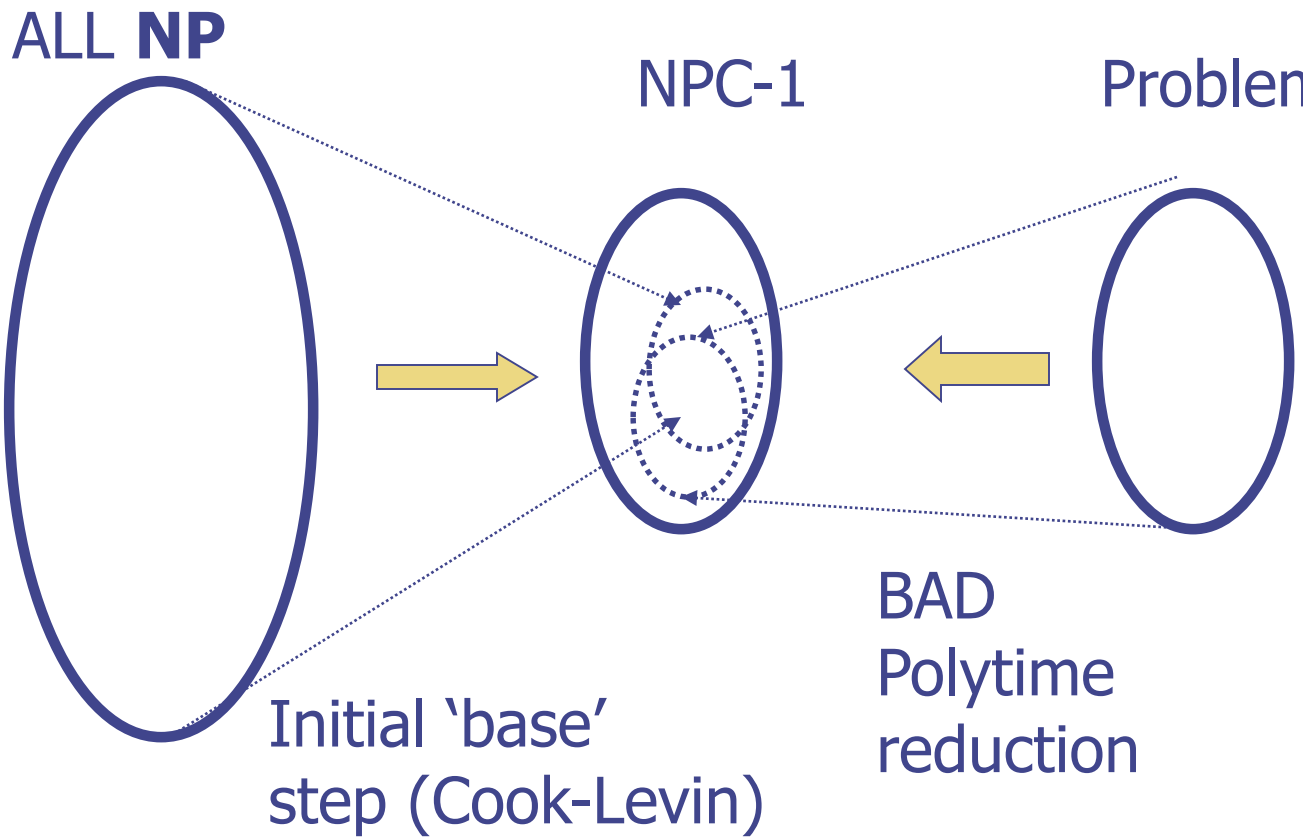
Then:

- Any problem in NP can be solved by conversion (in polytime) to some instance of NPC-2

The chains allow to reduce any NP-complete problem to any other NP-complete problem.

CAUION: BAD ORDERING

- Suppose the reduction is 'back-to-front'



We cannot deduce that Problem 2 solve all of **NP**

Example: Reducing from PART to SUBSET-SUM

- This means that we need to give a method to convert **any** instance of PART into **some** instance of SUBSET-SUM (SSUM for short)
- The "**any**" and "**some**" are vital
- (Of course, it is implicit that the answer by using SUBSET-SUM should used to give the correct answer to PART.)

Reducing PART to SUBSET-SUM

- Given

  boolean SSUM(int[] arr, int k)

- Implement

  boolean PART(int[] arr)

Reducing PART to SUBSET-SUM

```
boolean PART(int[] arr) {
   int sum=0;
   for( int i = 0 ; i < arr.length ; i++ ) {
     sum += arr[i];
   }
   if ( sum %2 == 1 ) return false;
   return SSUM(arr, sum/2);
}
```

Reducing PART to SUBSET-SUM

```
boolean PART(int[] arr) {
   int sum=0;
    for( int i = 0 ; i < arr.length ; i++ ) {
     sum += arr[i];
    }
   if ( sum%2 == 1 ) return false;
    return SSUM(arr, sum/2);
}
```

What can we conclude?

Reduction from PART to SSUM

shows

If SSUM is easy, then so is PART

Reducing from PART to SUBSET-SUM

```
boolean PART(int[] arr) {
   int sum=0;
    for( int i = 0 ; i < arr.length ; i++ ) {
     sum += arr[i];
    }
   if ( k%2 == 1 ) return false;
    return SSUM(arr, sum/2);
}
```

What can we conclude?

Reduction from PART to SSUM

shows

If PART is NP-hard then SSUM is NP-hard

Essence of NP-complete Problems

- Naïve and "sound and complete" algorithms are exponential time
- They are "guess and verify"
  - Verification is fast, i.e. in **P**
- Theory:
  - "Guess" is encoded using "Non-determinism"
- Practice:
  - "Guess" is encoded using "Heuristics"
- "Non-determinism" can be thought of as a "perfect heuristic" obtained from a ("magic") "oracle"

Expectations include:

- Know definitions of P and NP
  - That, trivially, P is a subset of NP
- Understand the P = NP question:
  - That it is currently unsolved
  - Why it is important
- Definitions of NP-hard and NP-complete
  - Reductions, in general
    - And how they relate "the difficulty of problems"
  - That can, in poly-time
    - Reduce "all NP" to SAT
    - Reduce any problem in NP, to SAT, or any other NP-complete problem

“Appendix”

- Some more NP-complete problems
- For each one you should check that it is in NP

Graph Colouring (GC)

INSTANCE:

an undirected graph G = (V,E) with vertices V and edges E

an integer k

QUESTION

Is there a legal colouring of G with k colours. I.e. can we assign a colour c(v) to every vertex v in V, such that for every edge, the nodes at the ends are different colours?

Usage: many usages. E.g. node is an exam, edge is that the two exams have a student in common, colour is the timeslot it is assigned to. Colouring corresponds to a legal timetable with no-one in two exams a the same time

INDEPENDENT SET

INSTANCE:

an undirected graph G

an integer k

QUESTION

is there an independent vertex set of size k?
I.e. is there a set of k vertices such that all no pair within the set have an edge between them?

Usage: (exercise)

CLIQUE

INSTANCE:

an undirected graph G

an integer k

QUESTION

is there an clique of size k?

I.e. is there a set of k vertices such that all pairs within the set have an edge between them?

Usage: Many uses. Finding a closely connected set of friends in a social network. Timetabling (exercise: why?)

VERTEX COVER

INSTANCE:

an undirected graph G

an integer k

QUESTION

is there set of k vertices such that they cover all the edges of the graph? I.e. can we pick a set of vertices such that for all edges, at least one of the end vertices is in the set.

Many usages: E.g. facility location:

- so everyone is close to a fire station
- so CCTV can see every corridor,
- etc

TSP(D)

Decision version of the TSP (Travelling Salesperson Problem)

INSTANCE:
A graph G with distances on the edges
A target distance d

QUESTION:
Is there a tour of all the vertices with total sum of the edge lengths being at most d?

Usage: vehicle routing problems, delivery and collection systems

Note: Optimisation vs. Decision

- Need to distinguish
  - Decision Problem:
    - Simple yes/no answer
  - Optimisation Problem
    - Find a min or max value for some objective
- Complexity theory generally works in terms of Decision Problems
- Conversion is simple:
  - Minimisation of f, becomes picking an maximum allowed value "k" and using the decision version "is there a solution with f < k"
  - Binary search on k can then do the minimisation

INTEGER PROGRAMMING

INSTANCE:

A set of nonnegative integer variables, x[i]
A set of linear inequalities on the variables

QUESTION

Is there an assignment of values such that all the inequalities are satisfied

Note:

This is different: Why is it not "obvious" that the problem is in NP? Instead a difficult theorem is needed (Exercise. Hint: What is the size of an "answer"?)