Lesson 14
NP-Complete Problems:

Handling Problems From the Field of All Possibilities

Wholeness of the Lesson

Decision problems that have no known polynomial time solution are considered *hard*, but hard problems can be further classified to determine their degree of hardness. A decision problem belongs to NP if there is a polynomial *p* and an algorithm *A* such that for any instance of the problem of size *n*, a correct solution to the problem can be *verified* using *A* in at most *p*(*n*) steps. In addition, the problem is said to be *NP-complete* if it belongs to NP and every NP problem can be polynomial-reduced to it.

Science of Consciousness: The human intellect can grasp truths within a certain range but is not the only faculty of knowing. The transcendental level of awareness is a field beyond the grasp of the intellect ("beyond even the intellect is he" -- Gita, III.42). And the field of manifest existence, from gross to subtle, is too vast and complex to be grasped by the intellect either ("unfathomable is the course of action" – Gita IV.17).

Overview

In this lesson:

- Review polynomial time decision problems and the class *P*
- Review the class *NP* of decision problems.
- Describe the class of “hard” NP problems – the *NP complete problems.*
- Demonstrating NP-completeness, with examples, and the *P=NP* problem
- Techniques for handling *NP*-completeness in practice, with examples.

Decision Problems and Instances of Problems

- A decision problem is a problem with a 'yes-no' answer.
- Example of a Decision Problem (VertexCover)

  Given a graph G = (V,E) and a positive integer k, is there a vertex cover for G of size at most k?
- Example of an instance of a Decision Problem:

  Is there a vertex cover of size at most 3 for the complete graph on 5 vertices?
- An instance of a Decision Problem is said to *have a solution* or be a *solvable instance* if "true" is the correct answer to the problem.
  - *Example:* "Is there a vertex cover of size at most 3 for the complete graph on 4 vertices?" *has a solution* (so "true" is the correct answer)
  - *Example:* "Is there a vertex cover of size at most 3 for the complete graph on 8 vertices?" *does not* have a solution (so "true" is *not* correct)

Review: Polynomial-time Bounded Algorithms and the class *P*

- If an algorithm runs in $O(n^k)$ for some k, it is said to be a *polynomial-time bounded* algorithm.
- A decision problem is *polynomial-time bounded* (also, a "P Problem") if there is some polynomial-time bounded algorithm that solves (every solvable instance of) the decision problem. The class of all such problems is denoted *P.*
  - *Examples*: The Sorting Problem, the Shortest Path Problem, the MST Problem all belong to *P.*

How Do We Know When a Problem Does Not Belong to *P*?

- Hard to know for sure because even if there is no known polynomial time algorithm today, tomorrow someone may come up with one.
- Modern-day example: The **IsPrime** problem. Before 2002, all known algorithms to solve this problem ran in **superpolynomial** time.

  Note: An algorithm is said to take **superpolynomial** time if $T(n)$ is not bounded above by any polynomial. It is $\omega(n^k)$ time for all constants k, where n is the size of the input parameter.
- **AKS Primality Test** was the first polynomial-time solution. Its fastest known implementation runs in

  $$O(n^6 * \log^k n))$$

  for some k, and n is the size of the input parameter. (AKS stands for Agrawal–Kayal–Saxena)

Review: The class *NP*

- To understand decision problems that may not belong to *P,* one approach is to see how hard it is to *check* whether a given solution is correct. Typically easier to check a solution than to obtain a solution in the first place.
- ***Intuitively speaking***, the class of decision problems with the property that a solution can be verified in polynomial time is denoted NP.
- ***More precisely:*** We say a decision problem belongs to NP if there is an algorithm B(x,y) which runs in O(p(n)) time for some polynomial p(n) such that, whenever x is input of size n for an instance of the problem that has a solution, then correctness of some solution y is verified by B(x,y) in O(p(n)) time and returns "true".

  Note: Since we consider only inputs x for which there is a solution, we are guaranteed that a solution will exist – but we may use any solution we like for the purpose of verification.

Verify: SubsetSum is in NP

- Recall that the input for this problem is a set S = $\{s_0, s_1, . . ., s_{n-1}\}$ and k, The decision problem to solve is, is there a subset of S whose elements add up to k?
- To verify that the problem is NP, we assume we are given input that has a solution, and that T is a solution. We determine the running time required to verify that T is a subset of S and sum of the elements in T is k.

  Verification:

```
verify all elements in T belong to S
sum = 0;
for each j in T
        sum += j
if(sum == k) return true
else  return false
```

- These verification can be performed in O(n) steps. Therefore, SubsetSum belongs to NP.

Verify: HamiltonianCycle is in NP

- Recall that the input for this problem is a graph G =(V,E), where |V|=n. The decision problem to solve is, Does G contain a Hamiltonian cycle?
- To verify that the problem is NP, suppose we are given such a graph that does have a Hamiltonian cycle, and C is a solution. We determine the running time required to verify that C is indeed a Hamiltonian cycle. Here is what must be verified:
  - Check: As a graph, C is a simple cycle.
  - Check: C contains all vertices of G.
  - Check: All edges of C are also edges of G.
- These verification can be performed in O(n) steps. Therefore, HamiltonianCycle belongs to NP.

Finding Problems *Not* in ***NP***

- *PowerSet problem.* Given a set X of size n, a kind of optimization problem concerning the power set of X is to generate all subsets of X. Whatever method is used, just writing out the output requires at least 2^n steps.
- A corresponding decision problem is: Given a set X and a collection P, is P = P(X)?

- Any algorithm that solves PowerSet problem must generate every subset of X, and add the subsets to a return set.(requires at least 2^n steps) So this problem does not belong to P.

  Moreover, verifying correctness requires checking that each set in P is a subset of X, and this has to be done in 2^n steps, so it doesn't belong to NP either.

$P \subseteq NP$

General ideas of the proof: Suppose Q is a decision problem that belongs to P. Therefore, there is an algorithm A(x) that solves size-n instances of Q in O(p(n)) time for some polynomial p(n). To show Q is in NP, we need to define a polynomial time algorithm B(x,y) that verifies, for each size-n solvable instance x of Q, the correctness of <u>some</u> solution y. For this proof, we pick y to be the solution that A gives us when it runs on x. The solution y can be represented as a set of size O(p(n)). Here is what B will do:

1. B will accept inputs x, y.
2. B will run A on input x to produce a solution z, represented as a set of size O(p(n)). (Note that it must be true that y = z since y was obtained in the same way.)
3. B will verify that y = z. Since both y and z are sets of size O(p(n)), it will require no more than O(p(n)) time to check y = z.

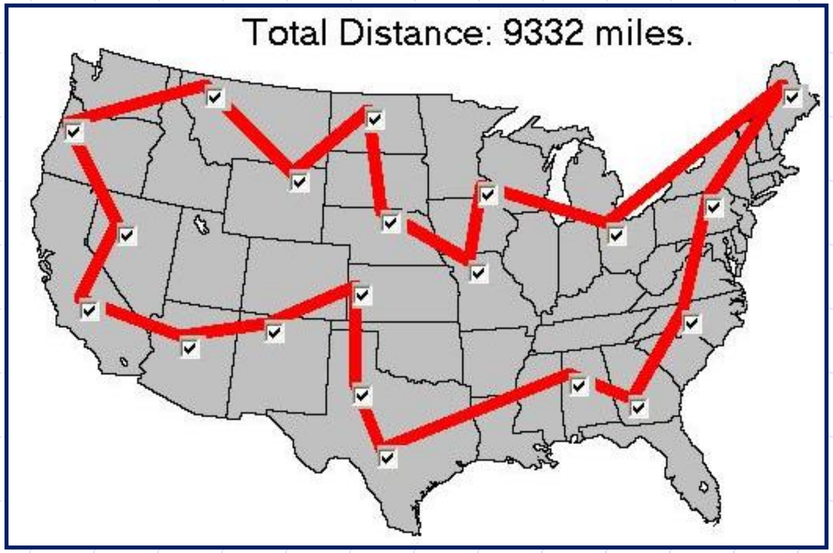
Therefore, B will correctly return true in O(p(n)) time.

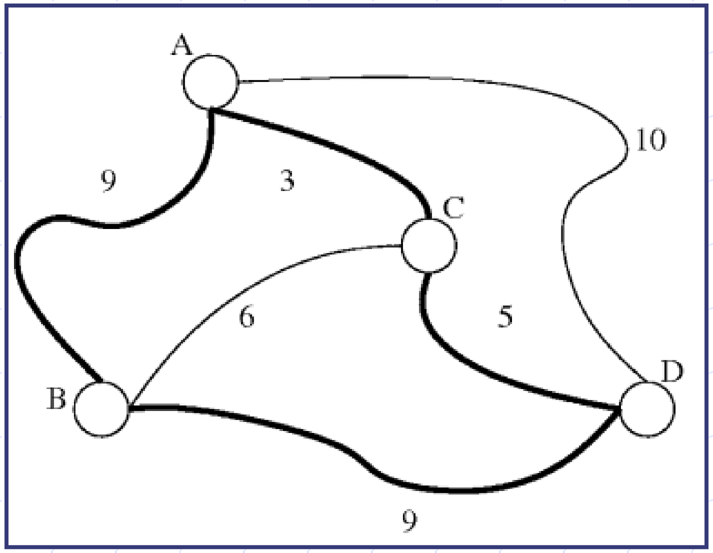
NP Problems

- As in last slide, all the problems that we have been able to solve in polynomial time belong to NP.
- We have already verified that SUBSETSUM, HAMILTONIANCYCLE and VERTEXCOVER(lab) belong to NP.
- Another famous example is the TRAVELINGSALESMAN problem (TSP)

Another famous Example: Traveling Salesman Problem

- *Traveling Salesman Problem* (TSP): Given a complete graph G with cost function c: E $\rightarrow$ N and a positive integer k, is there a Hamiltonian cycle C in G so that the sum of the costs of the edges in C is at most k? <u>Solution data</u>: a subset of E.





Is ***P*** = ***NP***?

- Biggest open question in CS.
- Many thousands of research papers have been written in an attempt to make progress in solving this problem.
- If it were true, then thousands of problems that were believed to have infeasible solutions would suddenly have feasible solutions.

(Note: feasible algorithm: polynomial-time algorithm)

A Remarkable Fact About ***NP***

- Many of the problems that are known to be in NP can also be shown to be *NP-complete.* Intuitively, this means they are the hardest among the NP problems.
- It can be shown that if someone ever figures out a polynomial-time algorithm to solve an NP-complete problem, then *all problems in NP will also have polynomial time solutions.*
- One consequence: If anyone finds a polynomial time solution to an NP-complete problem, then P = NP.
- These points are elaborated in the next slides.

NP-Complete Problems

- A problem Q is ***NP***-*hard* if for *every* problem R in ***NP***, R is *polynomial reducible* to Q.
- A problem Q is ***NP***-*complete* if Q belongs to *NP*, and Q is ***NP***-*hard*.

Reducibility

- *Intuitively*: Q is *polynomial reducible* to R if Q is “no harder than” than R or R is “at least as hard as” Q.
- *More detail:* Q is *polynomial reducible* to R if, in polynomial time, you can transform any solvable instance of type Q into a solvable one of type R so that a solution to one yields a solution to the other.
- *Formally*: A problem Q is *polynomial reducible* to a problem R if there is a polynomial p(y) and an algorithm C so that when C runs on input data X of size O(n) for any instance I_Q of Q, C outputs input data Y of size O(p(n)) for an instance I_R of R in O(p(n)) steps so that

  $$I_Q \textit{ has a solution iff } I_R \textit{ has a solution}$$

  In this case, we say that C, p(y) *witness* that Q is polynomial reducible to R.
- We write $Q \xrightarrow{\text{poly}} R$
- Fact: (Transitivity of Reducibility) If $A \xrightarrow{\text{poly}} B$ and $B \xrightarrow{\text{poly}} C$, then $A \xrightarrow{\text{poly}} C$

Practice with the definition: VertexCover $\xrightarrow{\text{poly}}$ HamiltonianCycle

It can be shown that VertexCover is polynomial reducible to HamiltonianCycle. What does this mean?

- *Intuitively*: the VertxCover problem can be turned into a Hamiltonian Cycle problem in polynomial time, so if you can solve the Hamiltonian Cycle problem, you can solve the VertexCover problem without doing much more work.

- *Formally:* There is a polynomial p(y) and an algorithm C that does the following: Using an instance G, k of the VertexCover problem (where G has n vertices) as input to C, C outputs, in O(p(n)) time, a graph H with O(p(n)) vertices so that:

  G has a vertex cover of size at most k iff

  H has a Hamiltonian cycle

- *Note*: The details for this algorithm C are tricky.

HamiltonianCycle $\xrightarrow{\text{poly}}$ TSP

We show HamiltonianCycle is reducible to TSP

- Given a graph G = (V,E) on n vertices (input for HamiltonianCycle) – notice G is a subgraph of K_n. Obtain an instance H, c, k of TSP as follows: Let H be the complete graph on n vertices (i.e. H is K_n), obtained by adding the missing edges to G. Let $c(e) = 0$ if $e \in E$, else $c(e) = 1$. Let $k = 0$.
- Need to show: G has a Hamiltonian cycle if and only if H, c, k has a Hamiltonian cycle with edge cost $\leq k$
- If G has Hamiltonian cycle C, C is Hamiltonian in H also. Since each edge e of C is in G, $c(e) = 0$. So cost sum $\leq k$. Converse: A solution C for H,c,k implies all edges of C have weight 0; therefore, every edge of C also is an edge in G. Therefore C is an HC in G.

NP-Complete Problems

- A problem Q is ***NP**-hard* if for *every* problem R in ***NP***, R is polynomial reducible to Q.
- A problem Q is ***NP**-complete* if Q belongs to *NP*, and Q is ***NP**-hard*.

The First NP-Complete Problem

- Cook-Levin discovered the first NP-complete problem, known as the SATISFIABILITY Problem (called SAT). The proof was complicated.

- SAT is the following problem: Given an expression using n boolean variables p,q,r,… joined together using connectives AND, OR, NOT, is there a way to assign values "true" or "false" to the variables so that the expression evaluates to true?

  Example: Consider

      p AND NOT (q OR (NOT r))

The First NP-Complete Problem

- Cook-Levin discovered the first NP-complete problem, known as the SATISFIABILITY Problem (called SAT). The proof was complicated.

- SAT is the following problem: Given an expression using *n* boolean variables p,q,r,… joined together using connectives AND, OR, NOT, is there a way to assign values "true" or "false" to the variables so that the expression evaluates to true?

  Example: Consider an *n* = 3 instance:

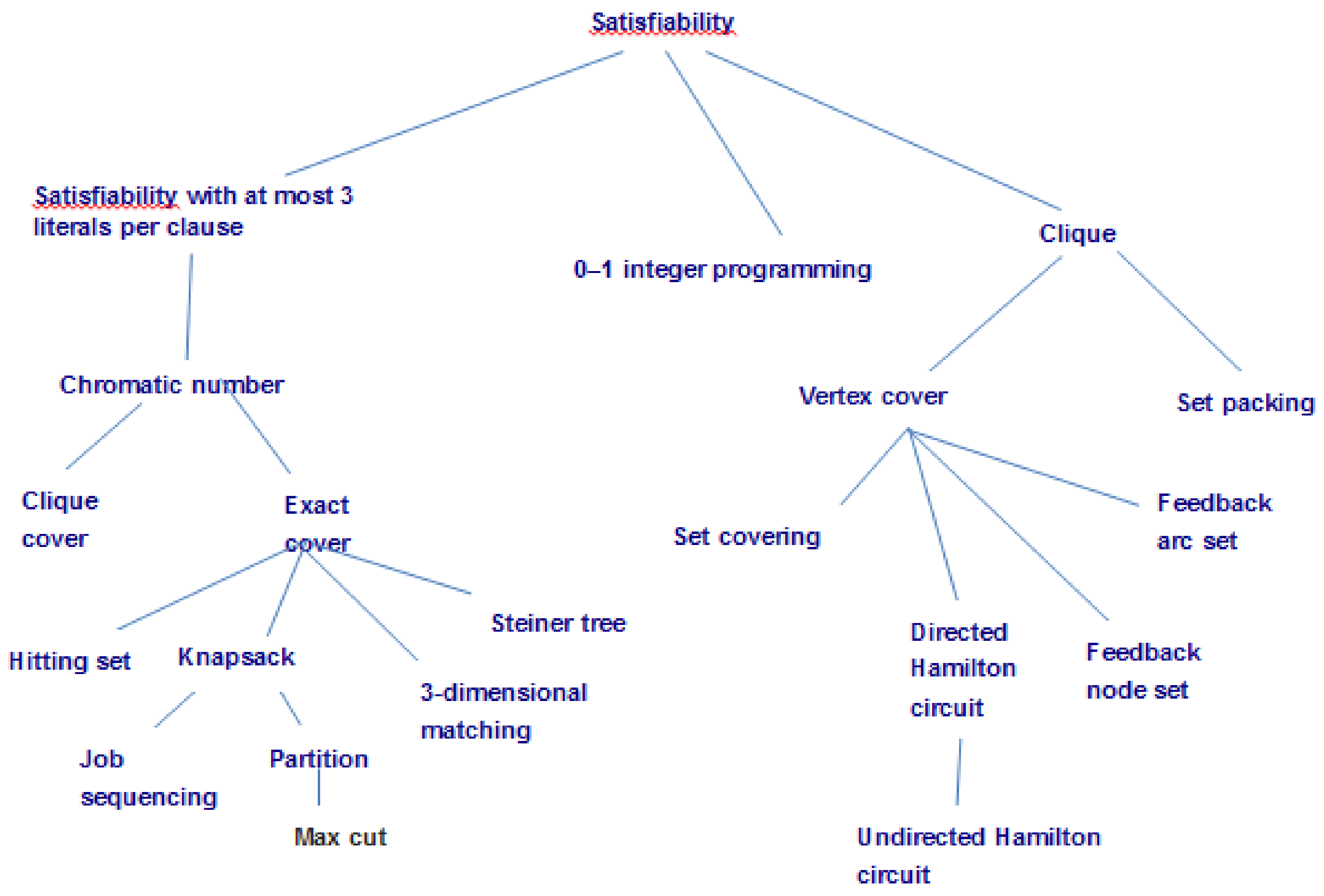
  p AND NOT (q OR (NOT r))

  Assigning T to p and to r and assigning F to q causes the expression to evaluate to T (i.e. "true").

VertexCover is NP-Complete

- Once a single NP-complete problem was discovered, others could be found by using the Cook-Levin result
- *Example*: Show VERTEXCOVER is NP-complete. The main idea is to prove that SAT is polynomial reducible to VertexCover. Then: Given any NP problem Q, to show Q is polynomial reducible to VertexCover, observe:
  - Q is polynomial reducible to Sat
  - SAT is polynomial reducible to VertexCover
  - Therefore, Q is polynomial reducible to VertexCover (By Transitivity of Reducibility)

21 NP-Complete Problems by Richard M. Karp



Solving One NP-Complete Problem Solves Them All

- Proof: Suppose Q is an NP-complete problem and someone finds a polynomial-time algorithm A that solves it in $O(p_1(\text{size of } Q))$ time.
- Let R be any problem in NP. R is polynomial reducible to Q, with witness polynomial $p_2(y)$.
- Polynomial-time algorithm B to solve R: Given an instance I_R of R, create an instance I_Q of Q in $O(p_2(n))$ time (size of input data for Q is $O(p_2(n))$). Solve I_Q in $O(p_1(p_2(n)))$ time. Solution to I_Q yields solution to I_R. Algorithm B runs in $O(p_2(n)) + O(p_1(p_2(n)))$. So we found a polynomial-time algorithm to solve any NP problem, which leads P=NP.

Main Point

The hardest NP problems are *NP-complete.* These require the highest degree of creativity to solve. However, if a polynomial-time algorithm is found for any one of them, then all NP problems will automatically be solved in polynomial time. This phenomenon illustrates the fact that the field of pure consciousness, the source of creativity, is itself a field of *infinite correlation* – "an impulse anywhere is an impulse everywhere".

What To Do with NP-Complete Problems?

There are many thousands of NP-complete problems, and a large percentage of these would be very useful if they could be implemented in a feasible way. But the known algorithms are too slow. What can be done?

Handling Hard Problems

1. Directly find a more efficient algorithm
   - The IsPrime breakthrough of 2002
2. Improve running time in special cases
   - The technique of dynamic programming – examples: SUBSETSUM, KNAPSACK
   - Application of mathematical results – examples: HAMILTONIANCYCLE
3. Modify the problem slightly – example: FRACTIONALKNAPSACK
4. Approximation algorithms – example: VERTEXCOVERAPPROXIMATION
5. Probabilistic algorithms – example: Solovay-Strassen algorithm
6. Use hard problems as an advantage

Option #1: Find a Better Algorithm

In the case of NP problems not known to be NP-complete, this is at least a reasonable goal.

Example: a polynomial time solution to the IsPrime problem was discovered in 2002; all known (deterministic) algorithms before 2002 were exponential

Option #2: Improve Performance in Special Cases -- Dynamic Programming

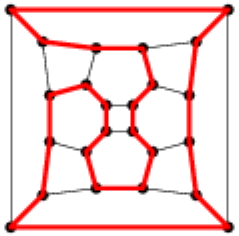
SUBSETSUM and KNAPSACK

- *Brute-force algorithm* is exponential, and in practice extremely slow
- *Dynamic programming solution* makes use of the fact that an optimal solution is built from optimal solutions of overlapping subproblems – solutions to these subproblems can be stored in a table and accessed as necessary
    - SUBSETSUM. The memoization table for SubsetSum has O(n * k) elements (where n is number of input integers, k the target value). When k is O(p(n)) for some polynomial p (for instance, when k is $O(n^3)$), we get polynomial running time (namely, $O(n^4)$). Here we compute running time entirely in terms of n (rather than in terms of n and ||k||).
    - KNAPSACK. The memoization table for Knapsack has O(n * W) elements (where n is number of items, W the maximum weight). When W is O(p(n)) for some polynomial p (for instance, when ||W|| is $O(n^3)$), we get polynomial running time (namely, $O(n^4)$).

Option #2: Improve Performance in Special Cases – Using Mathematical Results

Recall HAMILTONIANCYCLE and VERTEXCOVER are NP-complete. We indicate here how mathematical results can give us a way to solve these problems efficiently in special cases.

- **HAMILTONIANCYCLE***:* Given G=(V,E), does G have a Hamiltonian Cycle? <u>Solution data</u>: a subset of E



<u>Ore's Theorem</u>: If G is a finite simple graph with at least 3 vertices, G is Hamiltonian if:

$\deg v + \deg w \geq n$ for every pair of non-adjacent vertices v and w of G

Therefore, by performing an $O(n^2)$ preprocessing step, we can, in many cases, solve the HamiltonianCycle problem in polynomial time.

Option #3: Modify the Problem

Example: KNAPSACK -> FRACTIONALKNAPSACK

- Knapsack is NP-complete – all known solutions are exponential.
- By allowing *fractions* of items to be used (rather than requiring *whole* items every time), as is done in FractionalKnapsack, running time is improved to O(n log n)

Option #4: Approximation Algorithms

- There are many examples of NP-hard problems that have been “approximately solved” using a very fast algorithm. Sometimes an approximate solution is good enough.
- Criteria for a good approximation algorithm:
  1. Algorithm should have a fast (at least polynomial) running time
  2. There should be a (provable) upper bound on how far the algorithm’s output deviates from the optimal output.

An Approximation Algorithm for VertexCover

- Recall the VERTEXCOVER problem: Given an undirected graph G = (V,E) and a positive integer k, decide whether there is a vertex cover (a subset U of V such that every edge in E has at least one endpoint in U) of size at most k.
- The idea behind VertexCoverApprox is this: Loop through all the edges of G. For each edge e = (u,v), include u, v in the cover and then delete all edges that are incident to u or v.

(continued)

```
Algorithm VertexCover Approx(G)
  Input: A graph G
  Output: A small vertex cover C for G

  C ← new Set
  while G still has edges do
      select an edge e = (v,w) of G
      add vertices v and w to C
      for each edge f incident to v or w do
          remove f from G
  return C
```

(continued)

- *C is a vertex cover.* Notice that the empty set covers any isolated vertex. Every edge was either *used* or *discarded.* Suppose e = (v,w) was a used edge. Then both v and w are in C. Suppose e = (v,w) was a discarded edge. Then one of v and w is in C, by the criterion for discarding.
- *C is at worst twice the size of an optimal vertex cover.* Let r be the number of edges that are *used* in VertexCoverApprox. The vertex cover C obtained from the algorithm will therefore have size 2r. At least one endpoint of each of these r edges must occur in a minimal vertex cover, and no endpoint is ever shared between two such edges. Therefore, a minimal vertex cover U must have at least r vertices:

$$r \leq |U| \leq 2r = |C|$$

- Running time of VertexCoverApprox is clearly $O(m^2)$.

Option #5: Probabilistic Algorithms

- Sometimes, randomization can be used in the construction of an algorithm to guarantee correct outputs with high probability, and also to guarantee fast running times.
- *Example*: The IsPrime problem again.

Solovay-Strassen Solution to IsPrime

Fact: There is a function f, which runs in O(log n) (that is, O(length(n))), such that for any odd positive integer n and any a chosen randomly in [1, n-1], if f(a,n) = 1, then n is composite, but if f(a,n) = 0, n is “probably” prime, but is in fact composite with probability < ½.

Such a function f with range {0,1} is defined by:

$$f(a,n) = 0 \text{ iff } a^{\frac{n-1}{2}} \equiv \left(\frac{a}{n}\right) \bmod n \text{ and } \left(\frac{a}{n}\right) \neq 0$$

When f(a,n) = 1, n must be composite. When f(a,n) = 0, n is “probably prime” but is composite with probability < ½.

(continued)

The **Solovay-Strassen** algorithm for determining whether an input natural number is prime is the following:

To conclude n is prime with probability > $1 - (1/2)^k$,

- Perform **Single-Round IsPrime** k times and store outputs in a list L
- If at least one value in L is FALSE, return FALSE.
- Otherwise (if all values in L are TRUE), return TRUE

Algorithm **Single-Round IsPrime**:

```
Input: A positive integer n
Ouptut: TRUE if n is probably prime, FALSE if n is composite
    if n = 2 return TRUE
    if n % 2 = 0 return FALSE
    a ← a random number in [1, n-1]
    if f(a,n) = 1
        return FALSE
    return TRUE
```

Option #6*:* Use hardness as an advantage

Using NP-hardness as sentry to protect resources. Cryptosystems sometimes base their encryption algorithm on an NP-hard problem. When successful, a hacker would have to, in essence, solve the NP-hard problem in order to crack a code.

- Example: Merkle-Hellman attempted to base a cryptosystem on the (hardness of the) Knapsack problem. However, this cryptosystem was eventually hacked.
- These days, the problem of factoring large numbers is used as the hard problem hackers have to solve to crack cryptosystems.

Connecting The Parts of Knowledge With The Wholeness of Knowledge

1. There are many natural decision problems in Computer Science for which feasible solutions are needed, but which are NP-complete. Therefore, there is little hope of finding such solutions.
2. The hardness of certain NP-complete problems is being used to ensure the security of certain cryptographic systems.

3. *Transcendental Consciousness* is a field of all possibilities and infinite creativity.
4. *Impulses Within the Transcendental Field.* Pure consciousness as it prepares to manifest is a "wide angle lens" making use of every possibility for creative ends
5. *Wholeness Moving Within Itself.* In Unity Consciousness, awareness does not get stuck in problems; problems are seen as steps of progress in the unfoldment of the dynamics of consciousness.