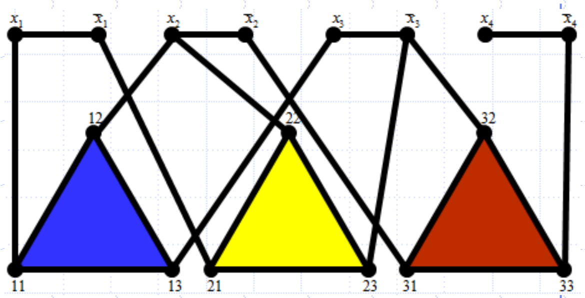
Lesson 15
NP-Complete Problems:
Handling Problems From the Field of All Possibilities

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.

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

- Most of the algorithms we have looked at run in O(n^3) time – usually much better.
  - *Examples*: Sorting algorithms, graph algorithms (if we write in terms of just number of vertices, Dijkstra and Kruskal run in O(n^2log n)), and operations on data structures
- If an algorithm runs in O(n^k) for some k, it is said to be a *polynomial-time bounded* algorithm
- A decision problem (i.e. output is either “true” or “false”) 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 Selection Problem, and so on.

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 deterministic algorithms to solve this problem ran in superpolynomial time.
- **AKS Primality Test** was the first polynomial-time solution. Its fastest known implementation runs in
  $$O(\mathit{length}(n)^6 * \log^k(\mathit{length}(n)))$$
  for some k. (AKS stands for Agrawal–Kayal–Saxena)
- *EXPTIME-Complete problems*, like nxn chess, *do require* superpolynomial time and therefore do not belong to *P*

Group activity: The class *NP*

Solve the equation:

$7x^2 - 12x - 352 = 0$

Group activity: The class *NP*

Solve the equation:

$7x^2 - 12x - 352 = 0$

Verify x = 8 is a solution to the equation

$7x^2 - 12x - 352 = 0$

Compare the time you took to solve to compare the time you took to verify.

Example : A Non-deterministic Algorithm

```
IsPresent(A, x)
// A is an array of items. x is an item.
// Will return true if x is present and false otherwise.
        i <- guess(A, x)  // guess will return the correct index
                          // if x is present in the array A.
        if (A[i] == x)
                return true
        else
                return false
```

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.
- The class of decision problems with the property that a solution can be verified in polynomial time is denoted *NP*.

Decision Problems and Instances of Problems

- 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 4 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)

(continued)

- An instance I of a problem has *input data* (e.g. (V,E,k) for VertexCover); any *candidate solution* for an instance also consists of data (e.g. a subset W of V for VertexCover) called *solution data*.

- Need to know the *size* of input data. For graphs, the convention is that the number of vertices is the size that is used

Definition of P

- **Definition** A decision problem Q is said to *belong to P* if there exists a polynomial p(y) ("polynomial witness") and an algorithm A such that for any instance I of Q that has a solution and that has input data X of size n, when A is run on input X, A outputs "true" in O(p(n)) time.

  We say A, p(y) *witness* that Q belongs to *P.*

Definition of NP

- **Defintion.** A decision problem Q is said to belong to *NP* if there exists a polynomial p(y) ("polynomial witness") and an algorithm B such that for any instance I of Q that has a solution, if X is input data of size n and Y is a solution data (called a *certificate*), then when B is run with input (X, Y), B outputs the message "verified" in O(p(n)) time. (Time includes the effort to read the certificate Y.)

  We say that B, p(y) *witness* that Q belongs to *NP*

P ⊆ *NP*

Proof: Let Q be a decision problem that belongs to P. Let p(y) be a polynomial and A an algorithm so that A, p(y) witness that Q belongs to *P.* We define an algorithm B so that B, p(y) witness that Q belongs to NP. Define B by letting it be the algorithm that does the same steps as A, but outputs "verified" instead of "true" on (solvable) input data it receives. Then if I is any instance of Q having a solution, with input data X of size n, then, when B is run on (X, {}) (i.e., the certificate is empty), B outputs "verified" in O(p(n)) steps (since A outputs "true" on input X in O(p(n)) steps).

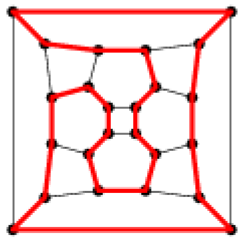
Hamiltonian Cycle And Vertex Covers

- A Hamiltonian cycle in a graph G is a simple cycle that contains every vertex of G. A graph is a Hamiltonian graph if it contains a Hamiltonian cycle.
- Examples. The Herschel graph is not a Hamiltonian graph.
- If G = (V,E) is a graph, a vertex cover for G is a set C $\subseteq$ V such that for every e $\in$ E, at least one end of e lies in C.
- Fact. The known algorithms for determining whether a graph is Hamiltonian, and for computing the smallest size of a vertex cover, run in exponential time.

NP Examples

- The HamiltonianCycle and VertexCover problems have already been shown to belong to *NP.*

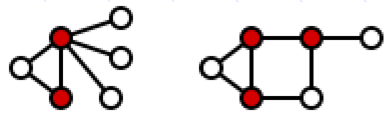
  HamiltonianCycle: Given G=(V,E), does G have a Hamiltonian Cycle? <u>Solution data</u>: a subset of E



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

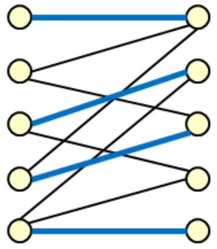
$\deg v + \deg w \geq n$ for every pair of non-adjacent vertices v and w of G

VertexCover: Given G=(V,E) and k, does G have a vertex cover of size at most k? <u>Solution data</u>: subset of V



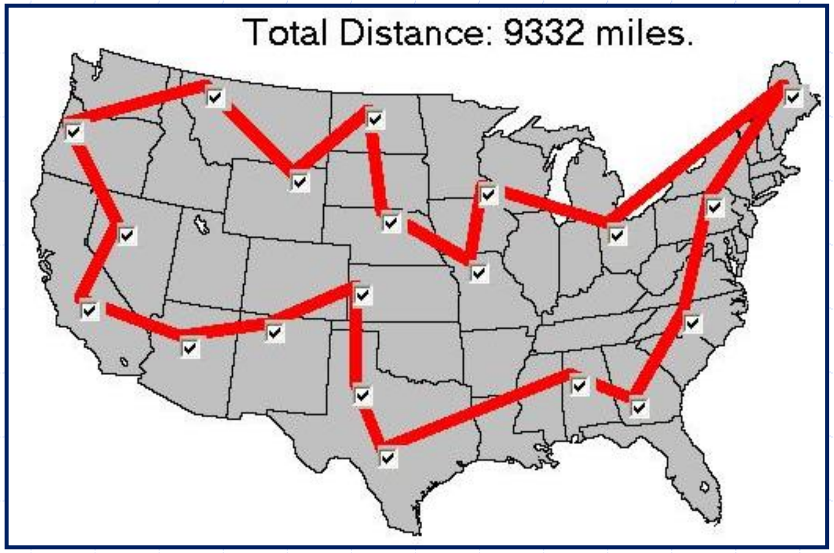
A <u>matching</u> in a graph is a set of edges no two of which share an endpoint.

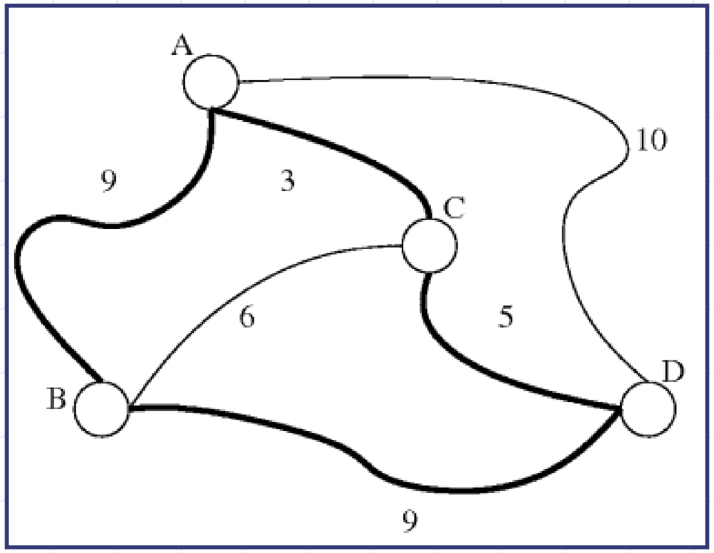
<u>Konig's Theorem</u>. In a bipartite graph, the number of edges in a maximum matching is the same as the number of vertices in a minimum vertex cover.



Traveling Salesperson Problem

- *Traveling Saleperson 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? Solution data: a subset of E.





TSP Belongs to NP

- Given an instance I of TSP with input data a complete graph G with n vertices, a cost function c: E $\to$ N and a positive integer k, and given a certificate E_0, a subset of E, the algorithm B checks that E_0 is a Hamiltonian cycle for G and then computes the sum of c(e) over the edges e in E_0 to verify that it is at most k.

Example: 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 (“find the largest possible collection of subsets of X without duplicates”). 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 this problem must check every element of P to see if it is a subset of X, so the algorithm requires at least 2^n steps in the worst case. So this problem does not belong to P.

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

- *Finite Halting Problem and N x N Chess.* We discusssed NxN chess in Lesson 1. Finite Halting Problem: Given a program P and positive integers n, k, does P, when running with input n, produce an output after excecuting k or fewer steps? Neither of these problems belong to P, but it is not known whether they belong to NP.

Is *P* = *NP*?

- The answer is not known
- 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

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.

Reducibility (informal 1)

Let Q denote the problem of finding the area of a square.

Let R denote the problem of finding the area of rectangle.

Given an instance I_Q of Q, we can transform into an instance I_R of R.

I_Q has a solution iff I_R has a solution

- We write $Q \xrightarrow{\text{poly}} R$
- Note that R is “harder” than Q

Reducibility (informal 2)

Let Q denote the problem computing the distance between two points in 2D.

Let R denote the problem computing the distance between two points in 3D.

Given an instance I_Q of Q, we can transform into an instance I_R of R.

I_Q has a solution iff I_R has a solution

- We write Q $\overset{\text{poly}}{\longrightarrow}$ R
- Note that R is "harder" than Q

Reducibility

- *Intuitively*: Q is *polynomial reducible* to R if, in polynomial time, you can transform a solvable problem 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 an 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 has a solution iff I_R has a solution

  In this case, we say that C, p(y) *witness* that Q is polynomial reducible to R.

- We write $Q \overset{\text{poly}}{\longrightarrow} R$

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

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

- *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

- *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.

- *Note*: The details for this algorithm C are tricky – not given here, but we make use of this result later.

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

We show HamiltonianCycle is reducible to TSP

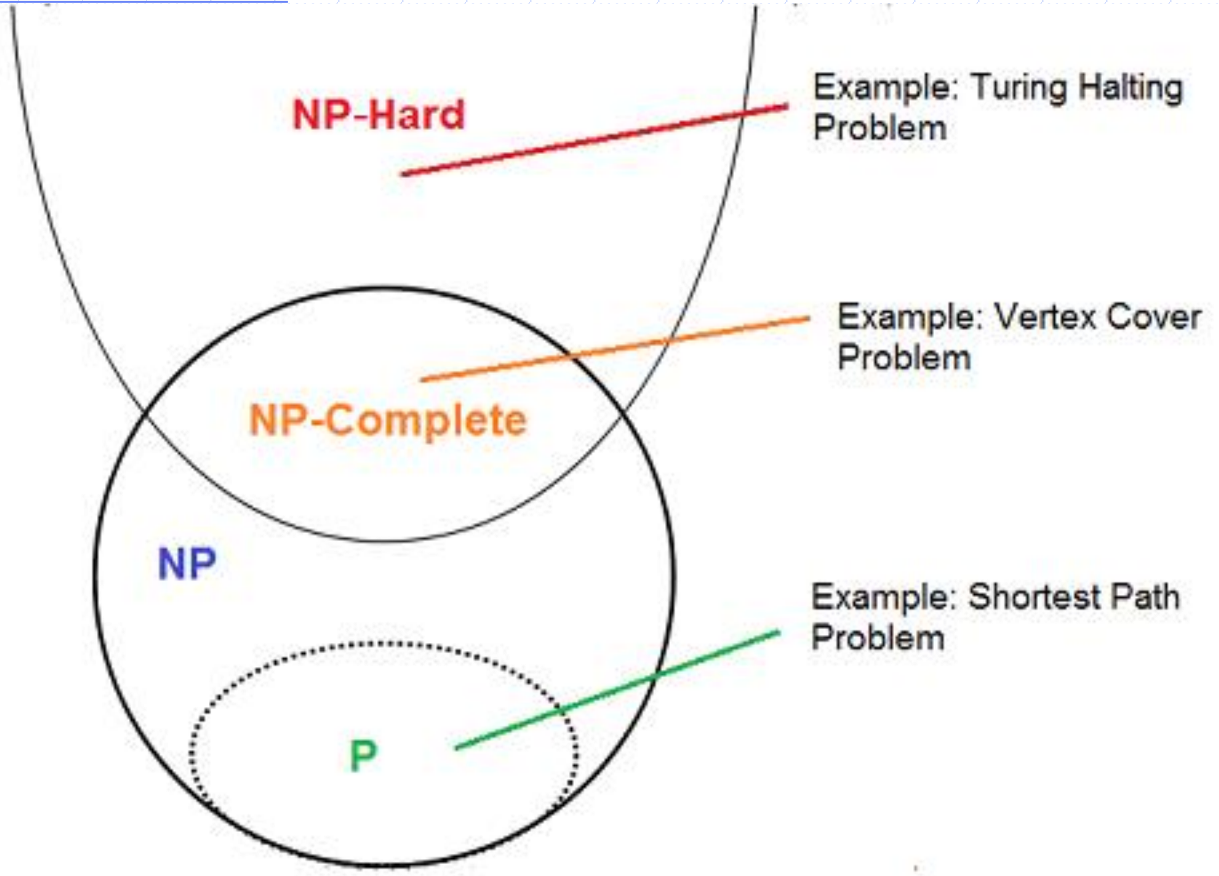
- Given a graph $G = (V,E)$ on n vertices (input for HamiltonianCycle) – notice G is a subgraph of K_n. Obtain an instance H, c, k of TSP as follows: Let H be the complete graph on n vertices (i.e. H is K_n), obtained by adding the missing edges to G. Let $c(e) = 0$ if $e \in E$, else $c(e) = 1$. Let $k = 0$.
- 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.

$$R \xrightarrow{\text{poly}} Q$$

NP-Complete Problems


NP-Hard
Example: Turing Halting Problem
Example: Vertex Cover Problem
NP-Complete
NP
Example: Shortest Path Problem
P

This diagram assumes that P != NP

NP-Complete Problems

A problem Q is ***NP**-complete* if Q belongs to *NP*, and Q is NP-hard.

TSPMIN: not in NP; but in NP-hard

Traveling Saleperson Problem MIN(TSPMIN): Given a complete graph G with cost function c: E $\rightarrow$ N and find a Hamiltonian cycle C in G so that the sum of the costs of the edges in C is minimum among all Hamiltonian cycles in G.

- TSPMIN is NP-hard
- TSPMIN is not in NP.
- Thus TSPMIN is not in NP-complete but in 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 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))

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

Other NP-Complete Problems

- Once SAT is shown to be NP-complete, others can be shown to be NP-complete much easily.
- Suppose we want to show VC is NP-complete. Till now, we must show for all Q in NP, Q is polynomial reducible to VC and VC is is NP.
- **All we have to do is SAT is polynomial reducible to VC and VC is in NP.**
- Reason: Given any NP problem Q we have
  - Q is polynomial reducible to SAT
  - SAT is polynomial reducible to VC
  - Therefore, Q is polynomial reducible to VC
  - VC is is NP.

Other NP-Complete Problems

- There is nothing special about SAT. It can be replaced any known NP-complete Problem

- **Pick a known NP-complete problem (say NPCP)**
- **All we have to do is NPCP is polynomial reducible to VC and VC is in NP.**

- Reason: Given any NP problem Q we have
  - Q is polynomial reducible to NPCP
  - NPCP is polynomial reducible to VC
  - Therefore, Q is polynomial reducible to VC
  - VC is in NP

HamiltonianCycle is NP-Complete

This is an outline of a proof that HamiltonianCycle is NP-Complete under the assumption that VertexCover is NP-complete:

- Show HC is in NP.
- Pick VC as the known NP-complete Problem
- Show VC is polynomial reducible to HC (see Slide 20)

Summary: To show Y is NP-Complete

- Show Y is in NP.
- Pick X. A known NP-complete Problem
- Show X is polynomial reducible to Y

Solving One NP-Complete Problem Solves Them All

- Suppose Q is an NP-complete problem and someone finds a polynomial-time algorithm A that solves it in O(p(n)) time.
- Let R be any problem in NP. R is polynomial reducible to Q, with witness polynomial q(y).
- Polynomial-time algorithm to solve R: Given an instance I_R of R, create an instance I_Q of Q (in O(q(n)) time) that has a solution iff I_R does. Solve I_Q in O(p(q(n))) time. Output solution to I_R . The algorithm runs in O(q(n) + q(p(n))).

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 NP-Hard Problems

- Find a more efficient algorithm somehow
- The IsPrime breakthrough of 2002
- The technique of dynamic programming – examples: SubsetSum (Knapsack also)
- Sometimes exponential running time is good enough
- Approximation algorithms
- Probabilistic algorithms
- Use hard problems as an advantage - cryptosystems

Options: 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

Better Algorithms: Dynamic Programming

- *Dynamic programming* is a technique that has been used to find more efficient solutions to NP-hard problems, though often even these solutions are still exponential.
- The idea: Sometimes problems can be broken down into subproblems, which can be solved, and whose solutions can be combined in some way to obtain a solution to the main problem. Solutions to subproblems are stored and combined stage by stage to produce a solution to the main problem.

(continued)

- When such a problem exhibits the following characteristics, it can in many cases be tackled using dynamic programming:
  - *Overlapping subproblems* – the subproblems "overlap" – the recursion tends to solve the same subproblems over and over (example: recursive fibonacci)
  - *Optimal substructure* – an optimal solution is composed of a combination of optimal solutions to subproblems (can't have an optimal solution to main problem that would result in suboptimal solutions to the subproblems)

Dynamic Programming Example: Subset Sum Problem

- The Subset Sum optimization problem says: We have set $S = \{s_0, s_1, \ldots, s_{n-1}\}$ of n positive integers and a non-negative integer k. Find a subset T of S so that the sum of the s_r in T is k.

$$\sum_{s_r \in T} s_r = k$$

Note. A solution to the optimization problem gives a solution to the decision problem

Main Observation

- Given an instance of SubsetSum: $S = \{s_0, s_1, \ldots, s_{n-1}\}$ and k. If T is a solution, then s_{n-1} is either in T or not.
- If s_{n-1} not in T, then T is a solution for $\{s_0, s_1, \ldots, s_{n-2}\}$, k.
- If s_{n-1} is in T, then $T - \{s_{n-1}\}$ is a solution for $\{s_0, s_1, \ldots, s_{n-2}\}$, $k - s_{n-1}$.

Applying Dynamic Programing to Solve SubsetSum

- There are only (k+1) * n problems to solve, namely:

For $0 \le i \le n-1, 0 \le j \le k$,
find a subset $T \subseteq \{s_0, s_1, \ldots, s_i\}$ so that
$\sum_{s_r \in T} s_r = k$.

Build a solution for bigger values of i and j using stored solutions for smaller values of i and j.

The Goal

Obtain a 2-dimensional array (a matrix) A so that

$$A[i,j] = \begin{cases} T & \text{where } T \subseteq \{s_0, s_1, \ldots, s_i\}, \sum_{s_r \in T} s_r = j \\ \text{NULL} & \text{if such a } T \text{ does not exist} \end{cases}$$

- If S contains values > k, we ignore them since they don't contribute to the solution (computations for which j is too big are skipped – see the implementation in code)
- Fill row i = 0 first, then fill later rows based on values of earlier rows.
- Each cell requires O(1) time, so computation of A[n-1, k] requires O(kn) time.

Details

Row 0:

$$\begin{aligned} A[0,0] &= \emptyset \quad \text{and} \quad A[0,s_0] = \{s_0\} \\ A[0,e] &= \text{NULL whenever } e \neq 0 \text{ and } e \neq s_0 \end{aligned}$$

Note: $\sum_{s_r \in \emptyset} s_r = 0$ and $\sum_{s_r \in \{s_0\}} s_r = s_0$.

Row i:

$$A[i,j] = \begin{cases} T = A[i-1,j] & \text{if } \sum_{s_r \in T} s_r = j \\ T = A[i-1,j-s_i] \cup \{s_i\} & \text{if } \sum_{s_r \in T} s_r = j \end{cases}$$

Note: In computation of $A[i,j]$, a value of NULL in both $A[i-1,j]$ and $A[i-1,j-s_i]$ means that $A[i,j] = \text{NULL}$.

Pseudo-polynomial time

The dynamic programming solution to SubsetSum runs in $O(kn)$. However, k may be much bigger than n, and even if k is $\Theta(n)$, the true running time is based on the number of bits in k, not on the value of k. So even this algorithm runs in exponential time.

Optional: Dynamic Programming: Knapsack Problem

The Problem. Given a set S = $\{s_0, s_1, \ldots, s_{n-1}\}$ of items, weights $\{w_0, w_1, \ldots, w_{n-1}\}$ and values $\{v_0, v_1, \ldots, v_{n-1}\}$ and a max weight W, find a subset T of S whose total value is maximal subject to constraint that total weight is at most W.

Note. A solution to the optimization problem gives a solution to the decision problem.

Observation. If T is a solution, either s_{n-1} belongs to T or it does not.

1. If it does not, then T is a solution to the Knapsack problem with items $\{s_0, s_1, \ldots, s_{n-2}\}$ weights $\{w_0, w_1, \ldots, w_{n-2}\}$ values $\{v_0, v_1, \ldots, v_{n-2}\}$ and max weight W.
2. If s_{n-1} does belong to T, then T – $\{s_{n-1}\}$ is a solution to the Knapsack problem with items $\{s_0, s_1, \ldots, s_{n-2}\}$ weights $\{w_0, w_1, \ldots, w_{n-2}\}$ values $\{v_0, v_1, \ldots, v_{n-2}\}$ and max weight W – w_{n-1}.

This shows that T is built up from solutions to subproblems.

Optional: Knapsack Solution

Solution. For $0 \le i \le n-1$ and $0 \le j \le W$, we obtain an $(n-1) \times (W+1)$ matrix A of subsets of the original set S of items, and an auxiliary matrix B of integers so that for each i, j, $B[i,j]$ is the sum of the values of the items in $A[i,j]$. Precisely:

$$A[i,j] = T \subseteq \{s_0, s_1, \ldots, s_i\} \text{ where } \sum_{t \in T} w_t \le j \text{ and } \sum_{t \in T} v_t \text{ is maximal.}$$

$$B[i,j] = \text{maximum total value of a subset of } \{s_0, s_1, \ldots, s_i\} \text{ whose total weight is } \le j.$$

Procedure to populate the matrices.

Row 0:

$$A[0,t] = \begin{cases} \emptyset & \text{if } t < w_0 \\ \{s_0\} & \text{if } t \ge w_0 \end{cases}$$

$$B[0,t] = \begin{cases} 0 & \text{if } t < w_0 \\ v_0 & \text{if } t \ge w_0 \end{cases}$$

Row i:

$$T_a = A[i-1,j], T_b = A[i-1,j-w_i] \cup \{s_i\}, B_a = B[i-1,j], B_b = v_i + B[i-1,j-w_i]$$

$$B[i,j] = \max\{B_a, B_b\},$$

$$A[i,j] = \begin{cases} T_a & \text{if } B_a \ge B_b \\ T_b & \text{otherwise.} \end{cases}$$

Optional: Knapsack

- The set stored in A[n-1, W] will be the solution.
- Running time is O(nW) in terms of values, but, in terms of input size, it's $O(n * 2^{\text{length}(W)})$, which is typically exponential in n (whenever $n \leq W$)

Options: Sometimes Exponential Time is Good Enough

- Sometimes the context in which an exponential algorithm is to be used does not require very large inputs. In such cases, exponential algorithms don’t present difficulties.

Options: Approximation Algorithms

Use an approximation algorithm. 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.

An Approximation Algorithm for VertexCover

- An approximation algorithm outputs values that are within a specified tolerance of the correct outputs, but typically runs much faster than an algorithm that can produce completely correct outputs.
- 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 $O(m^2)$; with attention to implementation details, this can be improved to $O(m+n)$.

Options: Probabilistic Algorithms

- *Use a probabilistic (or "randomized") algorithm.* 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.

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

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

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
```

Options: Use hardness as an advantage

- 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.

Optional: Public-Key Cryptography

- Alice wants to send message M to Bob and wants assurance that
  - No one else will be able to view M (confidentiality)
  - The message Bob gets is the one sent by Alice (authorship)
- Idea:
  - Each person has a public key P and a secret key S.
  - P is available to anyone but only the owner of S knows S
  - P and S are inverses: $P(S(M)) = M = S(P(M))$

(cont)

- Usage:
  - Alice can send a message to Bob by encrypting with Bob's public key: send $P_B(M)$. Then Bob can decrypt with his secret key $S_B(P_B(M)) = M$. This guarantees confidentiality but not authorship
  - To ensure authorship, Alice can also digitally sign $P_B(M)$ using her secret key: $S_A(P_B(M))$. Then Bob can decrypt using P_A followed by S_B: $M = S_B(P_A(S_A(P_B(M))))$. If the result is "garbage" it means that the author is not Alice. Both confidentiality and authorship are ensured (though in practice more efficient protocols are followed)

RSA Cryptosystem

The Requirements

- P(S(M)) = M
- Computation of P(M) and S(M) is efficient (when P and S are known)
- Infeasible to compute S from P (can therefore make P public)
- S(P(M)) = M (S, which is private, serves to *digitally sign* the message)

Making It Infeasible to Derive D from E

- Arrange it so that computation of S from P is an NP-complete (or at least *hard*) problem.
- Merkle-Hellman tried to do this using hardness of Knapsack problem, but relevant instances of Knapsack were not hard to solve
- RSA ties difficulty of factoring large numbers to problem of computing S from P.

RSA Setup

1. Pick at random two several hundred bit (~512 these days) primes p, q, and let $n = pq$.
2. Pick a small prime e relatively prime to $\phi(n) = (p-1)(q-1)$. Often e = 3, 7, 65,537 are used.
3. Compute d as modular inverse of e mod $\phi(n)$; that is, $ed \equiv 1 \bmod \phi(n)$
4. Public key is (e,n). Private key is (d,n).
5. Encryption of message M:
   $$P(M) = M^e \bmod n$$
6. Decryption of message M:
   $$S(M) = M^d \bmod n$$

Things To Verify

1. Encryption P(M) and decryption S(M) are fast when P and S are known [fast modular exponentiation]
2. P(S(M)) = S(P(M)) = M [Euler's Theorem]
3. Setup portion is reasonably fast
   - Obtain two large primes p, q [probable primes]
   - Verify e is relatively prime to $\phi(n)$ [gcd algorithm]
   - Compute d from e given $\phi(n)$ [variation of gcd algorithm]
4. Argue that computing d from e without knowledge of $\phi(n)$ is hard. [If only e and n are known, need to find factors p, q of n. Best known algorithm is exponential. For 512 bit primes, infeasible.]

Complete RSA Example

- Setup:
  - $p = 5$, $q = 11$
  - $n = 5 \cdot 11 = 55$
  - $\phi(n) = 4 \cdot 10 = 40$
  - $e = 3$
  - $d = 27$ $(3 \cdot 27 = 81 = 2 \cdot 40 + 1)$
- Encryption
  - $C = M^3 \bmod 55$
- Decryption
  - $M = C^{27} \bmod 55$

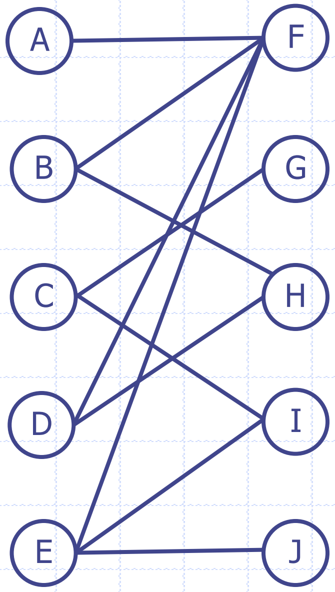
M	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
C	1	8	27	9	15	51	13	17	14	10	11	23	52	49	20	26	18	2
M	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36
C	39	25	21	33	12	19	5	31	48	7	24	50	36	43	22	34	30	16
M	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54
C	53	37	29	35	6	3	32	44	45	41	38	42	4	40	46	28	47	54

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.

Appendix: A Bipartite Graph


A
B
C
D
E
F
G
H
I
J