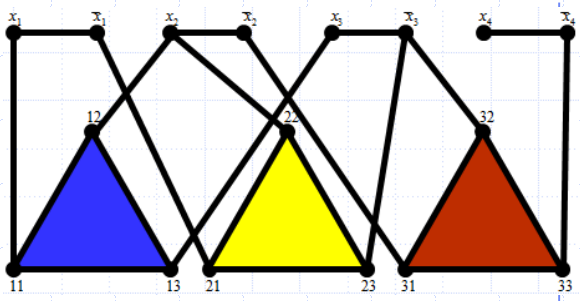
Lesson 12
NP-Complete Problems:
Handling Problems From the Field of All Possibilities

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:

- Polynomial time decision problems and the class *P*
- 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.

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 a problem that has a yes or no answer.

- 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, Searching Problem, the Shortest Path Problem, the MST Problem, all belong to *P.*
  - So far we have *not* found polynomial time bounded algorithms to solve any of the following problems:
    SUBSETSUM, POWERSET, VERTEXCOVER ("SMALLESTBASE"), ISPRIME
    Except for ISPRIME, the only known solutions run in exponential time (or worse).
  - *Note:* In this class, we have seen an exponential algorithm to solve SUBSETSUM. Later in this lesson we will see a faster algorithm, but it will still be exponential.

Decision Problems and Instances of Problems

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

  Given a set S of positive integers and a nonnegative integer k, is there a subset of S whose sum is k?
- Example of an instance of a Decision Problem:

  Given the set S = {3, 5, 8, 11} and the number k = 22, is there a a subset of S whose sum is 22?

  The *data* for this instance is S = {3, 5, 8, 11} and k = 22.

  The *solution data* for a solution is a subset T of S whose sum is 22.

- 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:* The SubsetSum problem {3, 5, 8, 11} and k = 22 *has a solution* (so "true" is the correct answer)
  - *Example:* The SubsetSum problem {3, 5, 8, 11} and k = 23 *has no solution* (so "false" is the correct answer)

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 exponential time. (See next slide.)
- Some problems, like PowerSet, *do require* exponential time and therefore do not belong to *P* – there will never be a feasible solution to any of these problems.

More on IsPrime

- In terms of its integer inputs, the running time of the usual IsPrime algorithm is $\Theta(\sqrt{n})$. However, running times in the study of algorithms are computed in terms of the *size of inputs*.
- The size of a natural number n is the number of bits it contains: # bits in n = $1+\lfloor \log(n) \rfloor$. Note that n is exponentially larger than the number of bits it contains. We write *length*(n) for #bits in n.
- If we think of input to the IsPrime problem as the number b of bits in a given natural number n, then since n is $\Theta(2^b)$, a running time of $\Theta(\sqrt{n})$ in terms of the *value* of input n becomes $\Theta(\sqrt{2^b}) = \Theta((\sqrt{2})^b)$ in terms of input *size* (input size = #bits in input) which is exponential.
- **AKS Primality Test** was the first polynomial-time deterministic solution, published in 2002. Its fastest known implementation runs in
  $O(length(n)^6 * \log^k(length(n)))$
  for some k. (AKS stands for Agrawal–Kayal–Saxena)

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 Q belongs to NP if there are an algorithm B(x,y) and a polynomial p(n) such that, whenever x is data of size n for a solvable instance of Q, then correctness of *some* solution y (called a *certificate*) is verified by B(x,y) in O(p(n)) time (and B(x,y) finally returns "verified"). ["NP" is short for "nondeterministic polynomial".]

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.

Compare with a precise definition of the class P:

A decision problem Q belongs to P if there are an algorithm A(x) and a polynomial p(n) so that, whenever x is data of size n for a solvable instance of Q, A when run on x returns "true" in O(p(n)) time.

SubsetSum is in NP

- Recall that the input for this problem is a set S = $\{s_0, s_1, \ldots, s_{n-1}\}$ and a non-negative integer 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 data that has a solution, and that T is a solution (a *certificate*). We determine the running time required to verify that T is a subset of S and that the 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 verified;
  else, return not verified;
  ```

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

HamiltonianCycle is in NP

- Given a graph G = (V,E), a Hamiltonian cycle in G is a spanning simple cycle in G. the HAMILTONIANCYCLE decision problem is:

  Given G, does G contain a Hamiltonian cycle?

- The input for this problem is a graph G =(V,E), where |V|=n. A candidate solution C is a collection of edges in G (which needs to be checked to be a spanning 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 (certificate). 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 verifications can be performed in O(n) steps. Therefore, HamiltonianCycle belongs to NP.

$P \subseteq NP$

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 some 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. (Store the elements of z as keys in a HashMap and perform containsKey on each element of y.)
4. B returns "verified".

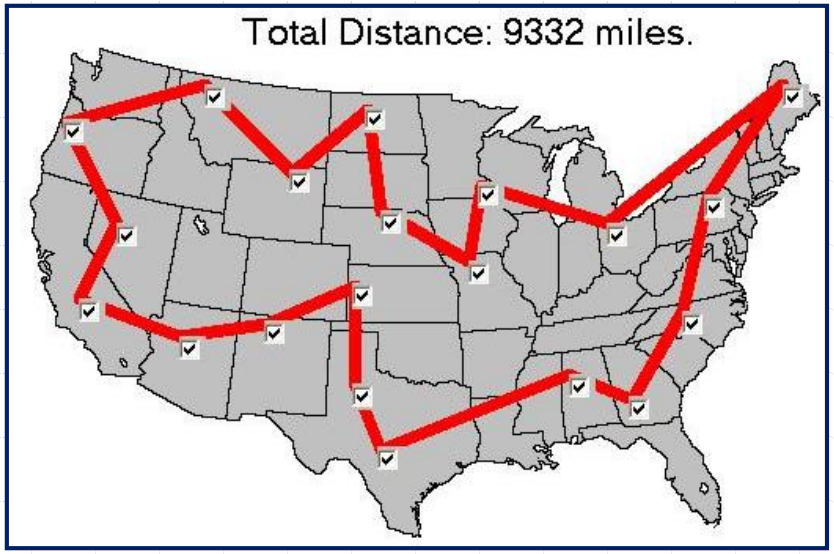
Note that B will correctly return "verified" in O(p(n)) time.

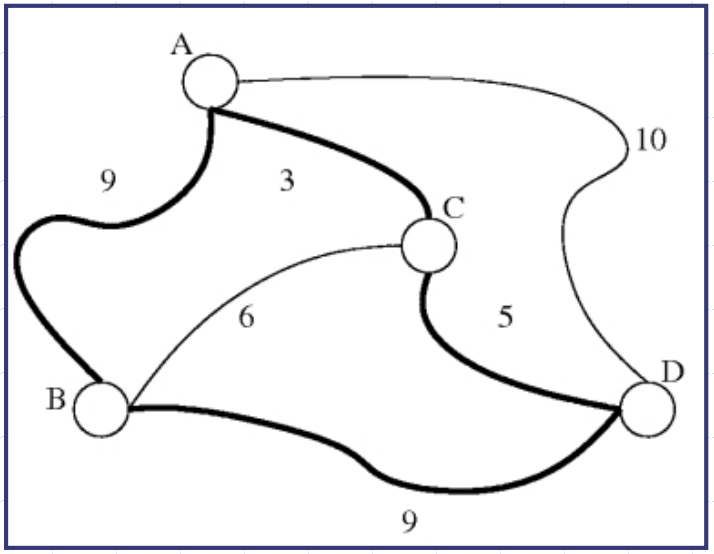
NP Problems

- By the result in the last slide, all the problems that we have been able to solve in polynomial time belong to NP (trivially)
  - *Warning!* Sometimes students tend to believe that NP has something to do with exponential algorithms. The last slide shows that this is NOT TRUE! Every problem that has a fast algorithm solution (and some others that don't) belongs to NP.
- We have already verified that HAMILTONIANCYCLE and VERTEXCOVER belong to NP.
- Another famous example is the TRAVELINGSALESMAN problem (TSP)

TRAVELINGSALESMAN 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? 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 $\rightarrow$ 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.

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 ("print out 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 it true that P contains every subset of X? Any algorithm that solves this problem must check every subset of X to see if it belongs to P, 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.
- Most problems that appear to be hard to solve *cannot* be proven to lie outside of NP. PowerSet is one of a handful of well-known examples.

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 (to be explained shortly)

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 even one 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

Reducibility

- *Informally*: 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. Intuitively, Q is no harder to solve than R.

- *Formally*: A problem Q is *polynomial reducible* to a problem R if there are 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, in O(p(n)) steps, input data Y of size O(p(n)) for an instance I_R of R, so that

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

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

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

- Fact: (Transitivity of Reducibility) If $A \overset{poly}{\longrightarrow} B$ and $B \overset{poly}{\longrightarrow} C$, then $A \overset{poly}{\longrightarrow} C$

Practice with the Definition: VertexCover $\overset{poly}{\longrightarrow}$ HamiltonianCycle

Recall the VertexCover problem ("Smallest Base" problem in labs): Given G and nonnegative integer k, is there a subset U of the vertices so that every edge of G has an endpoint in U and $|U| \leq k$?

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

- *Intuitively*: the VertexCover problem can be turned into a Hamiltonian Cycle problem in polynomial time, so that if you can solve the Hamiltonian Cycle problem efficiently, 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 – not given here, but we make use of this result later.

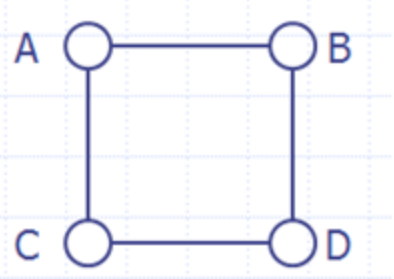
HamiltonianCycle $\overset{poly}{\rightarrow}$ TSP

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.

Illustration of the Proof with an Example

- The proof that HamiltonianCycle $\overset{poly}{\rightarrow}$ TSP can be illustrated with concrete examples. Here is one such example.
- We start with input for an HC problem, which is any connected graph G= (V,E)

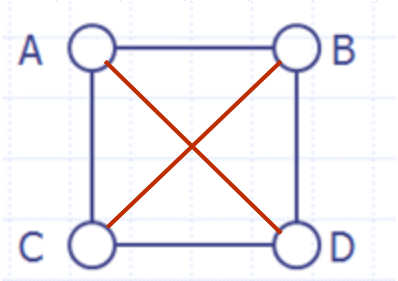


G = (V,E)

- We must transform this input G into input for a TSP problem – having the form (H, c, k) where H is a complete graph, k is a nonnegative integer and c is a cost function.
- We must transform in a way that guarantees there is a solution to the original HC problem if and only if there is a solution to the new TSP problem

Illustration (continued)

- To define the complete graph H we fill in the missing edges of G

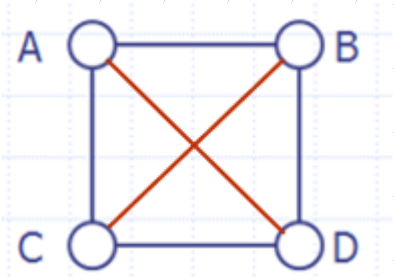


$H = (V, E_H)$

- We define k to be the integer 0.
- We define a cost function c on the edge set E_H of H:

$$c(e) = \begin{cases} 0 & \text{if } e \text{ belongs to } E \\ 1 & \text{if } e \text{ does not belong to } E \end{cases}$$

Illustration (continued)



- The definition of c tells us that

  $c(AB) = c(AC) = c(BD) = c(CD) = 0$ and $c(BC) = c(AD) = 1$

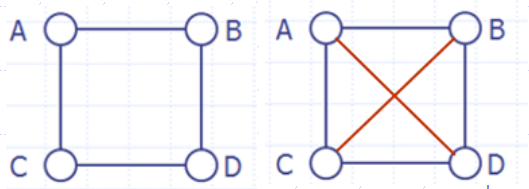
- We note that defining (H,c,k) from G can be done efficiently (polynomial time in the general case)

- Now that input for a TSP problem has been defined from G, we must verify two things:

  (1) If G has a Hamiltonian cycle then H also has a Hamiltonian cycle whose edge weights sum to a number $\leq k$

  (2) If H has a Hamiltonian cycle whose edge weights sum to a number $\leq k$, then G also has a Hamiltonian cycle.

Illustration (continued)



- Verification of (1): We observe that G *does* have a Hamiltonian cycle K (namely, K = A – B – D – C - A). We show that K is also a Hamiltonian cycle for H and that the sum of the edge weights of H is ≤ k

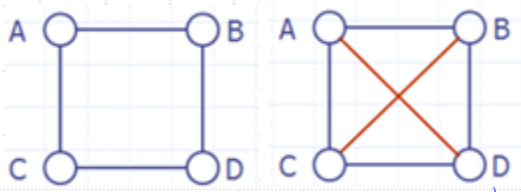
- *K is a Hamiltonian cycle in H*: This is true because every vertex of H is in K and K has no cycles (same as when K is considered a cycle in G).

- *Sum of edge weights of K is ≤ k :* Notice that every edge in K belongs to E, the set of edges of G. By definition, for each e in E, c(e) = 0. Therefore

$$\sum_{e \in K} c(e) = 0 \leq k$$

- We have shown that a solution to the HC problem having input G= (V,E) gives rise to a solution to the TSP problem we defined from G (namely, the problem with input data (H,c,k)).

Illustration (continued)



- Verification of (2): We observe that H *does* have a Hamiltonian cycle whose edge weights sum to a number ≤ k: Notice that the cycle K' = A – B – D – C – A is a Hamiltonian cycle in H and the sum of its edge weights is 0 (which is the value of k).
- We must show that this solution K' to the TSP problem gives rise to a solution to the original HC problem. We do this by showing K' itself is a solution to the original HC problem.
- To show K' is a Hamiltonian cycle in G, we must verify that all edges of K' belong to E. Notice that each edge in K' has weight 0 (since we know that the sum of *all* the edge weights of K' must be equal to 0).
- By the definition of c, if c(e) = 0, then it must be true that e belongs to E. Therefore, every edge in K belongs to E. We have shown K is a Hamiltonian cycle in G.
- We have shown that a solution to the TSP problem having input (H,c,k) gives rise to a solution to the original HC problem G.
- We have therefore shown HamiltonianCycle $\overset{\text{poly}}{\longrightarrow}$ TSP

$$c(e) = \begin{cases} 0 & \text{if } e \text{ belongs to } E \\ 1 & \text{if } e \text{ does not belong to } E \end{cases}$$

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 an n = 3 instance:

  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. 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").

Other NP-Complete Problems (SAT $\xrightarrow{poly}$ VC)

- Once a single NP-complete problem was discovered, others could be found by using the Cook-Levin result
- *Example*: VERTEXCOVER can be shown to be 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

  (the last step uses transitivity of reducibility)

VERTEXCOVER is NP-complete

HamiltonianCycle is NP-Complete (SAT $\xrightarrow{poly}$ VC $\xrightarrow{poly}$ HC)

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

- Recall that VertexCover is polynomial reducible to HamiltonianCycle (discussed in previous slide but not proven!).
- Therefore, since VertexCover is NP-complete, given any NP problem Q, Q is polynomial reducible to VertexCover
- Therefore, any such Q is polynomial reducible to HamiltonianCycle as well.

HAMILTONIANCYCLE is NP-complete

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 B, 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) + p(q(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, for a large percentage of these, polynomial-time solutions would be very useful. But the known algorithms are too slow. What can be done?

Handling NP-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 – example: SUBSETSUM
   - Application of mathematical results – examples: HAMILTONIANCYCLE
3. Approximation algorithms – example: VERTEXCOVERAPPROXIMATION
4. Probabilistic algorithms – example: Solovay-Strassen algorithm
5. Use hard problems as an advantage
   - Hard problems at the basis of cryptosystem design.

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

- *Brute-force algorithm* is exponential ($\Omega(n*2^n)$), and in practice extremely slow
- *Recursive algorithm* is also exponential ($O(2^n)$), but performs better than the brute-force solution
- *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. These run in pseudo-polynomial time.

The Subset Sum Problem

The Subset Sum Problem is the following: We have set $S = \{s_0, s_1, \ldots, s_{n-1}\}$ of n positive integers and a non-negative integer k. Is there a subset T of S so that the sum of the elements in T is k?

$$s_0 + s_1 + \ldots + s_{n-1} = k$$

Solving SubsetSum Using Subproblems

- We have seen that brute force solution for SubsetSum loops through all subsets of the given set S and tests whether any of these sum to the given number k.
- The key to a faster solution using recursion is based on the observation that there is a solution to the original problem $(\{s_0, s_1, \ldots, s_{n-1}\},k)$ if and only if one of two subproblems of the form $(\{s_0, s_1, \ldots, s_{n-2}\},k)$ and $(\{s_0, s_1, \ldots, s_{n-2}\},k - s_{n-1})$ has a solution.
- Example: Suppose S = {1,3,6,8}, k are inputs to the problem. If this problem has a solution, either:
  - {1,3,6},k has a solution (example: k = 4), or
  - {1,3,6}, k-8 has a solution (example: k =11)
- In the first case, T = {1,3} is a solution to the subproblem and also to the original problem. In the second case, T={3} is a solution to the subproblem and T ∪ {8} = {3,8} is a solution to the original problem

SubsetSum: Recursive Solution

A recursive solution is based on the following observation:

We are seeking a $T \subseteq S = \{s_0, s_1, \ldots, s_{n-2}, s_{n-1}\}$ whose sum is k. Such a T can be found if and only if one of the following is true:

(1) A subset T_1 of $\{s_0, s_1, \ldots, s_{n-2}\}$ can be found whose sum is k, OR

(2) A subset T_2 of $\{s_0, s_1, \ldots, s_{n-2}\}$ can be found whose sum is $k - s_{n-1}$

If (1) holds, then the desired set T is T_1. If (2) holds, the desired set T is $T_2 \cup \{s_{n-1}\}$.

The recursion proceeds by considering progressively smaller subsetsum problems, as the input set evolves from $\{s_0, \ldots, s_{n-1}\}$ to $\{s_0, \ldots, s_{n-2}\}$ to $\{s_0, \ldots, s_{n-3}\}$ to ... to $\{s_0\}$ to $\{\ \}$.

A Recursive Algorithm for SubsetSum

Algorithm *RecSubsetSum*(*S*, len, *k*)

Input: $S = \{s_0, s_1, \ldots, s_{n-1}\}$ positive integers,
k nonnegative integer, len = S.length

Output: *true* if for some $T \subseteq S$ we have
$sum(T) = k$, else *false*

//base case

if *S.size*() = 0 **then**
  if *k* = 0 **then return** *true*
  else return *false*

[lastIndex, last] ← *S.removeLast*()

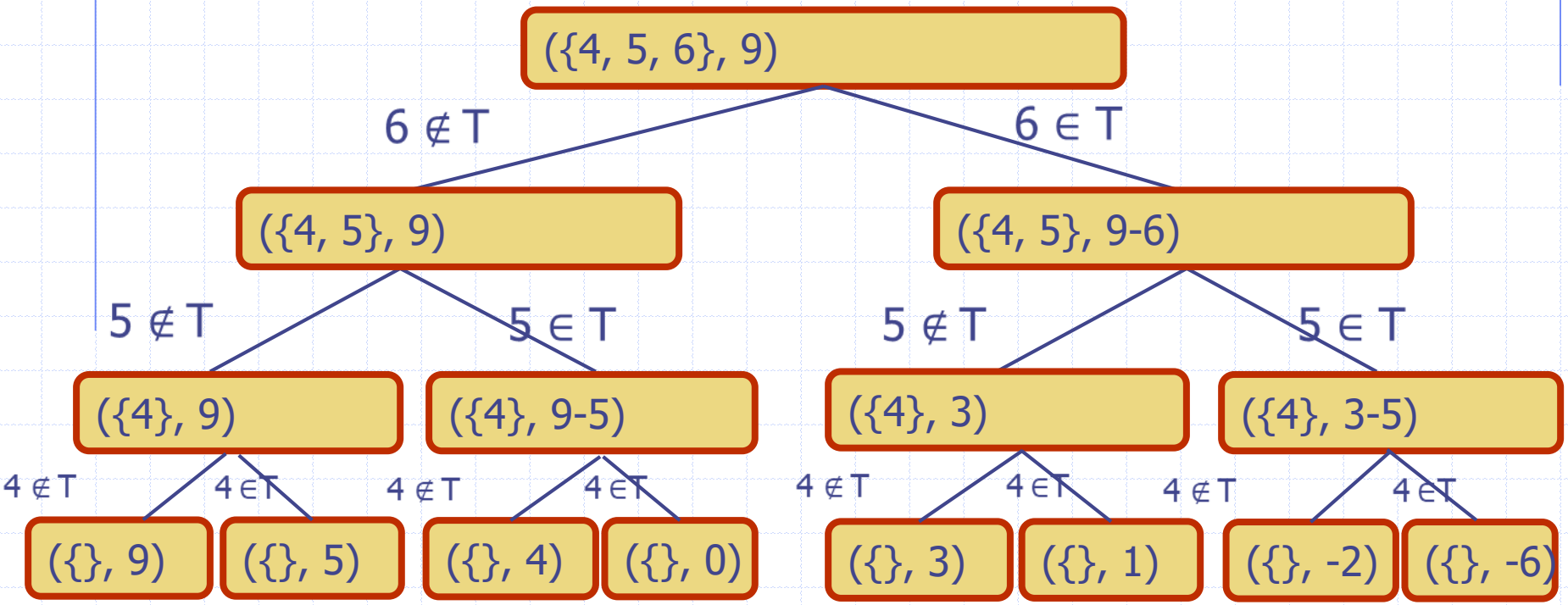
result1 ← *RecSubsetSum*(*S*, lastIndex, *k*)

if *result1* **then**
  return *result1*

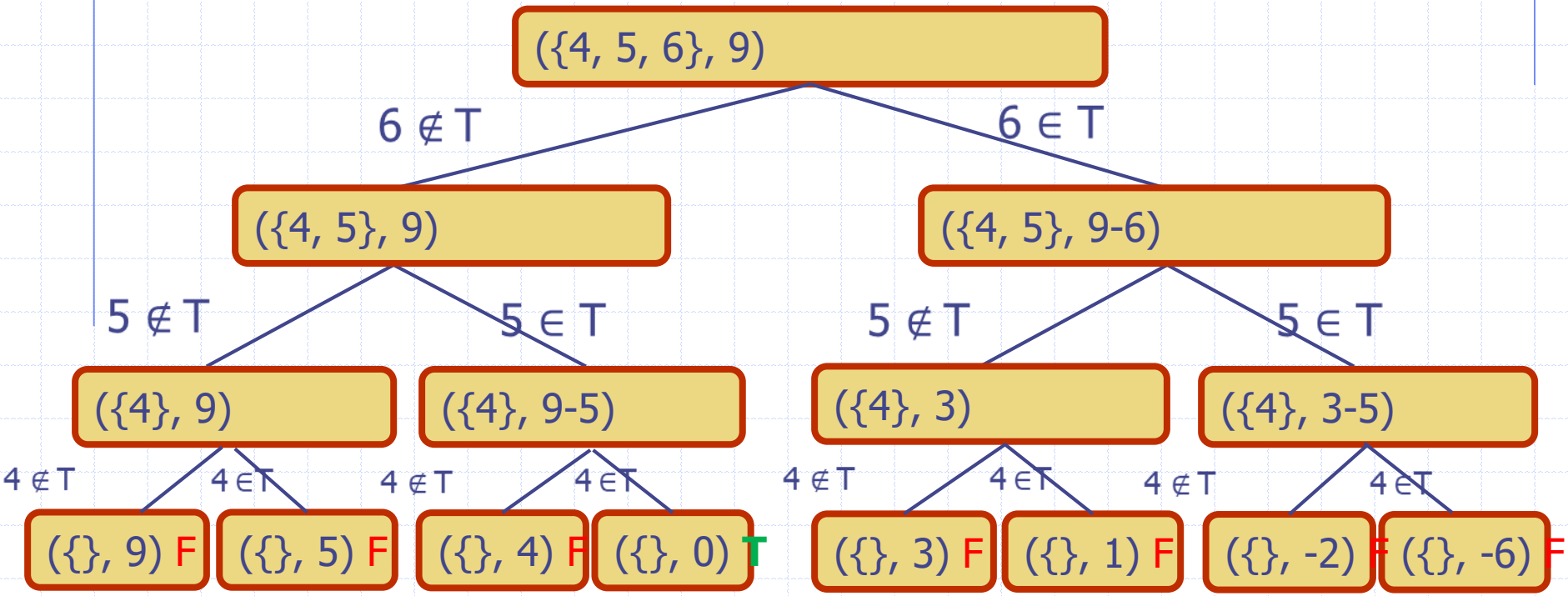
result2 ← *RecSubsetSum*(*S*, lastIndex, *k* – *last*)

return *result2*

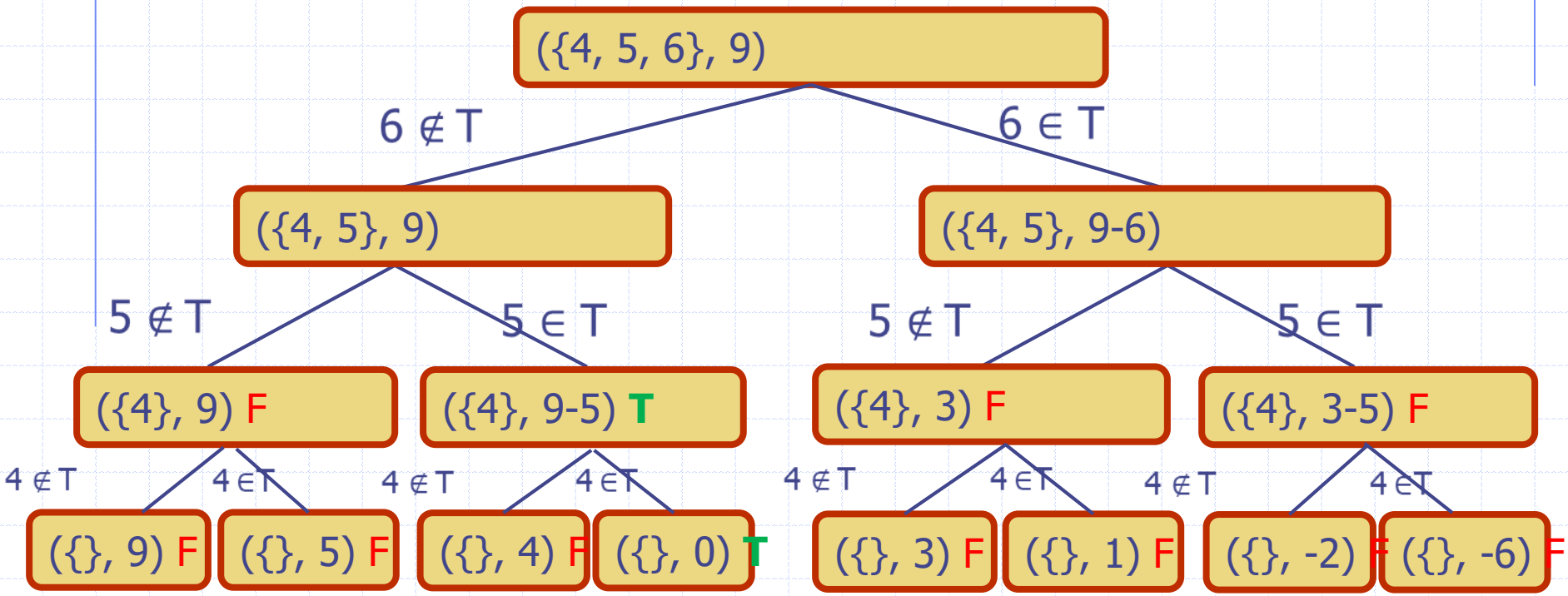
Execution Example



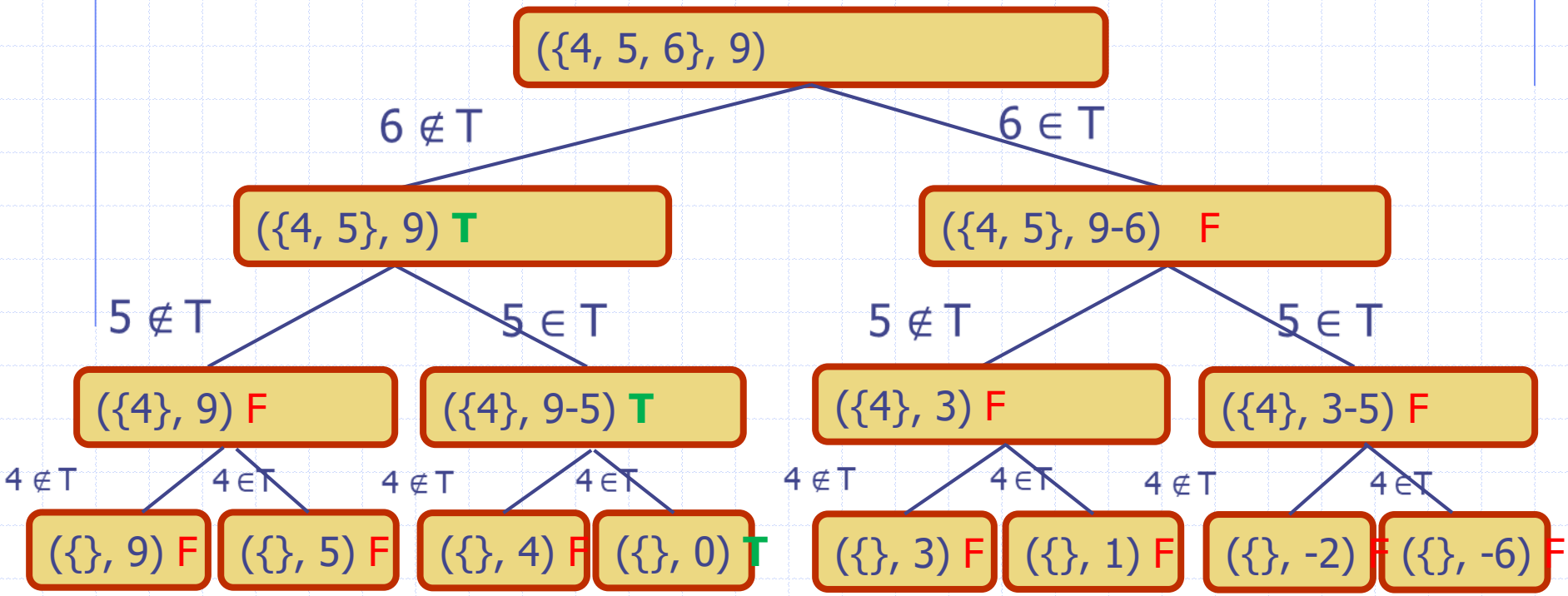
Execution Example



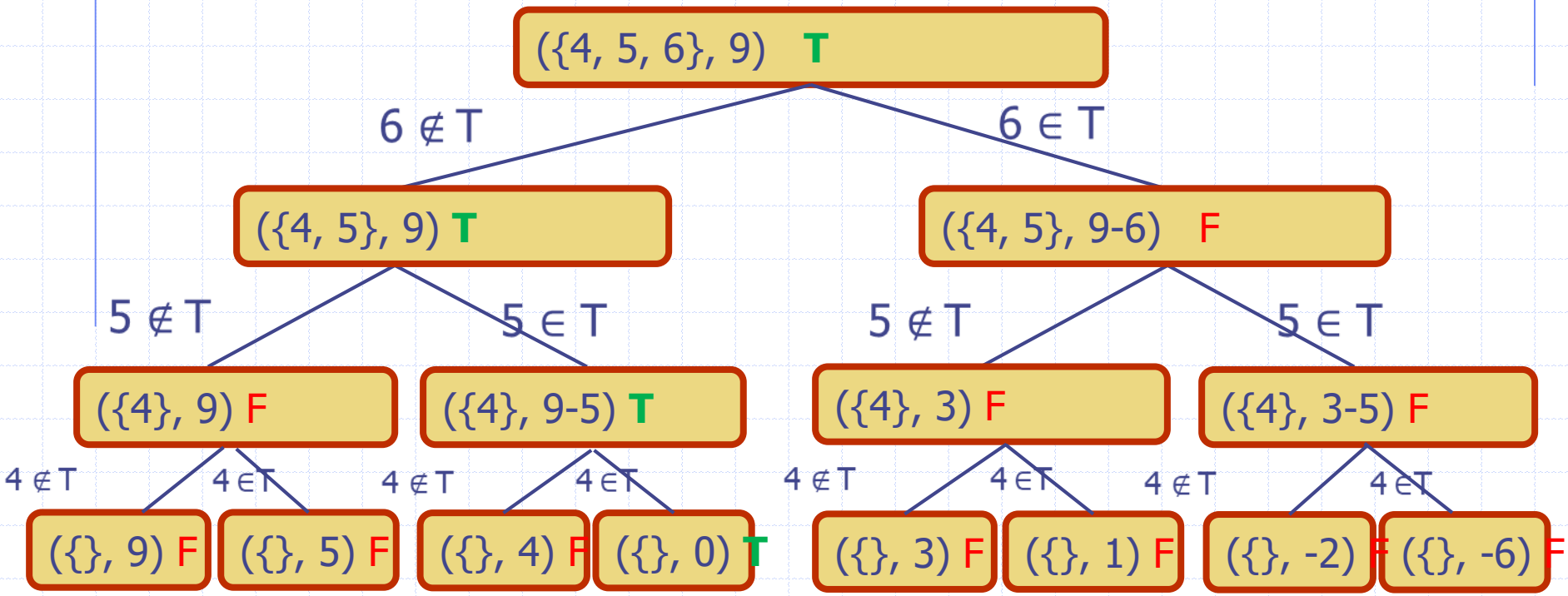
Execution Example



Execution Example



Execution Example



Running time

- The height of this recursion tree is n and in the worst case, every node stores a value. So the number of nodes in the recursion tree is $\Theta(2^n)$. Therefore, the recursive solution to SubsetSum runs in $\Omega(2^n)$.
- When the recursion tree is large, the reason nearly all nodes contain values is that certain computations are done over and over.
- If the results of computations are stored and then reused (reducing computation time in each case to O(1)), performance can be improved considerably. See SubsetSum demo.

Optimized Running Time

- When computations are stored and reused (this is called the *dynamic programming solution)*, the SubsetSum problem ($\{s_0, s_1, \ldots, s_{n-1}\}$,k) can be solved in
  $$O(n * (k+1)) \text{ time}$$
- However, this running time is not based on input *size.* If b is the number of bits in the number k, running time becomes
  $$O(n * 2^b)$$
  which is exponential in the input.
- Therefore, although the dynamic programming solution is much faster than the un-optimized recursive solution, it is still an exponential algorithm. Nevertheless in practice this solution is efficient in typical cases.

Option #3: Modify the Problem

Example: SUBSETSUM $\Rightarrow$ SUMOFTWO

- SUBSETSUM is NP-complete – all known solutions are exponential.
- SUMOFTWO is a special case of SUBSETSUM.

  SUMOFTWO: Given a list L of n positive integers and a positive integer z, do there exist x and y in L so that x + y = z?

- One could also consider similar problems SUMOFTHREE, SUMOFFOUR etc. These easier problems may be good enough in some contexts, and they are polynomial-time solvable.

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 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 an 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:

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

- Running time of VertexCoverApprox is clearly $O(m^2)$; with attention to implementation details, this can be improved to O(m+n).

Option #5*:* 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.
- RSA Cryptosystems (basis of public-key cryptography) obtain two distinct primes and multiply them together to obtain a large integer n. Using n, a public key and a private key are derived. To obtain the private key, hacker needs to be able to find the two prime factors of n.

Breaking RSA

- Interesting Fact: Using a *quantum computer*, it is possible to break RSA by solving the problem of factoring large numbers in feasible time ($O(n^3)$). Factorization of large numbers in this case is accomplished by Shor's *quantum algorithm* for factoring.

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.