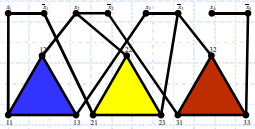
Lecture 15: Is P = NP?



Goals of today's lecture

- Define classes P and NP
- Explain the difference between decision and optimization problems
  - Show how to convert optimization problems to decision problems
- Describe what puts a problem into class NP
- Prove that P is a subset of NP
- Show how to write an algorithm to check a potential solution to an NP problem
- Give examples of how to reduce (convert) one problem into another
  - Importance of reduction (next lecture)

Wholeness Statement

Complexity class NP is fundamental to complexity theory in computer science. Decision problems in the class NP are problems that can be non-deterministically decided in polynomial time. Non-deterministic decision algorithms have two phases, a non-deterministic phase and a deterministic phase. In physics and natural law, the unified field of pure consciousness appears infinitely dynamic, chaotic, and non-deterministic, yet it is the silent source of the order and laws of nature in creation.

Outline and Reading

- P and NP (§13.1)
  - Definition of P
  - Definition of NP
  - Alternate definition of NP (skip)
- Strings over an alphabet (language)
- Language acceptors
- Nondeterministic computing

Can you write a program that decides whether this program ever halts?

A perfect number is an integer that is the sum of its positive factors (divisors), not including the number: 6 = 1 + 2 + 3

```
Algorithm FindOddPerfectNumber()
Input: none
Output: Returns an odd perfect number
  n ← 1
  sum ← 0
  while sum ⌐= n do
    n ← n + 2
    sum ← 0
    for fact ← 1 to n -1 do
      if fact is a factor of n then
        sum ← sum + fact
  return n
```


Theory of Computation

- A *function* is a mapping of elements from a set called the domain to exactly one element of a set called the range.
- What is a computable function?
  - A function for which an algorithm (step by step procedure) can be defined to compute the mapping no matter how long it takes or how much memory it needs
  - For example, sorting, LCS, selection, MST, TSP, Fractional and 0-1 Knapsack, etc.
- What is a definable function?
  - A function for which the mapping can be described with a mathematical formula
  - For example, the halting problem is definable, but not computable
- Are most functions definable or undefinable?

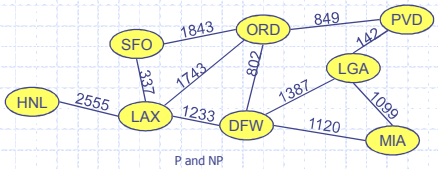
Halting Problem
Alan Turing (1936)

"Given the description of a program and its input, determine whether the program, when executed on this input, ever halts (completes). The alternative is that it runs forever without halting"

- Alan Turing proved that a general algorithm to solve the halting problem for all possible inputs does not exist.

Running Time Revisited

- Input size, ***n***
  - To be exact, let ***n*** denote the number of **bits** in a nonunary encoding of the input
- All the polynomial-time algorithms studied so far in this course run in polynomial time using this definition of input size (i.e., $O(n^k)$).
  - Exception: any pseudo-polynomial time algorithm



Intractability

- A problem is *intractable* if it is not possible to solve it with a polynomial-time algorithm.
- Non-polynomial examples: 2^n, 4^n, n!
- Polynomial-time algorithms are usually faster than non-polynomial time ones, but not always. Why?

Traveling Salesperson Problem (TSP)

- Given a set of cities and a "cost" to travel between each of these cities
- Determine the order we should visit all of the cities (once), returning to the starting city at the end, while minimizing the total cost

TSP Perspective

- With 8 cities, there are 40,320 possible orderings of the cities
- With 10 cities, there are 3,628,800 possible orderings
- If we had a program and computer that could do 100 of these calculations per second, then it would take more than four centuries to look at all possible permutations of 15 cities [McConnell]

TSP

- Does computing all shortest paths solve the TSP problem? Why or why not?
  - Shortest path is only between two cities
  - TSP has to go to all cities and back to the starting city
- What about MST?
  - MST does not compute a simple cycle

Main Point

1. Many important problems such as job scheduling, TSP, the 0-1-Knapsack problem, and Hamiltonian cycles have no known efficient algorithm (with a polynomial time bound).

   When an individual projects his intention from the state of pure awareness, then the algorithms of natural law compute the fulfilment of those intentions with perfect efficiency.

Instances of a Problem

- *What is the difference between a problem and an instance of that problem?*
  - To formalize things, we will express instances of problems as strings
- To simplify things, we will worry only about *decision problems* with a yes/no answer
  - Many problems are *optimization problems*, so we often have to re-cast those as decision problems
- *How can we express an instance of the MST problem as a string?*

Strings and Languages

- A language is a subset of the possible finite strings over a finite alphabet
  - Example: L = {(a|b)* | #a's = #b's}
- We can view a decision problem as an acceptor that accepts just the strings that correctly solve the problem
  - Assumption: if the syntax of the proposed solution is wrong, then the acceptor answers no

Problems and Languages

- A **language** L is a set of strings defined over some alphabet Σ
- Every decision algorithm A defines a language L
  - L is the set consisting of every string x such that A outputs "yes" on input x.
  - We say "A **accepts** x" in this case

    Example:
    - Suppose algorithm A determines whether or not a given graph G has a spanning tree with weight at most *K*
    - The language L is the set of graphs accepted by A
    - A accepts graph G (represented as a string) if it has a spanning tree with weight at most *K*

Transforming the Problem to a Decision Problem

- To simplify the notion of "hardness," we will focus on the following:
  1. Polynomial-time is the cut-off for efficiency/feasibility
  2. Decision problems: output is 1 or 0 ("yes" or "no")

     Examples:
     - Does a text T contain a pattern P?
     - Does an instance of 0/1 Knapsack have a solution with benefit at least K?
     - Does a graph G have an MST with weight at most K?
     - Does a given graph G have an Euler tour (a path/cycle that visits every edge exactly once)?
     - Does a given graph G have an Hamiltonian cycle (a simple cycle that visits every node exactly once)?

Proposed Solutions to a Decision Problem

- Many decision problems are phrased as existence questions:
  - Does there exist a truth assignment that makes a given logical expression true?
- For a given input, a "*solution*" is an object that satisfies the criteria in the problem and hence justifies a yes answer
- A "*proposed solution*" is simply an object of the appropriate kind that may or may not satisfy the criteria
  - A proposed solution may be described by a string of symbols from some finite alphabet, e.g., the set of keyboard symbols

The Complexity Class P

- A **complexity class** is a collection of languages
- P is the complexity class consisting of all languages that are accepted by **polynomial-time** algorithms
  - i.e., decision problems that can be **decided** in polynomial time
- For each language L in P there is a polynomial-time decision algorithm A for L.
  - If n=|x|, for x in L, then A runs in p(n) time on input x. where function p(n) is some polynomial (n^k)

Polynomial-Time Algorithms

- Are some problems solvable in polynomial time?
  - Yes: every algorithm we've studied provides a polynomial-time solution to some problem
  - Thus the algorithms we've studied so far (except for the pseudo-polynomial algorithms) are members of complexity class **P**
- Are all problems solvable in polynomial time?
  - No: Turing's "Halting Problem" is not solvable by any computer, no matter how much time is given
  - Such problems are clearly intractable, not in **P**

The Class P

- The problems in class P are said to be tractable problems
- Not every problem in P has an acceptably efficient algorithm
  - Nonetheless, if not in P, then it will be extremely expensive and probably impossible in practice

3 Categories of Problems

1. Problems for which polynomial-time algorithms have been found.
   - sorting, searching, matrix multiplication, shortest paths, MST, LCS
2. Problems that have been proven to be intractable.
   - "undecidable" problems like Halting.
3. Problems that have not been proven to be intractable but for which polynomial-time algorithms have not been found.
   - 0-1 knapsack, TSP, subset-sum
   - Leads us to the theory of NP

Decision Problems

- A formal definition of NP
  - Only applies to decision problems
  - Uses nondeterministic algorithms
    - not realistic (i.e., we do not run them on a computer)
    - but they are useful for classifying problems
  - A decision problem is, abstractly, some function from a set of inputs to the set {yes, no}

Converting an Optimization Problem to a Decision Problem

- Convenient relationship
  - We can usually cast an optimization problem as a decision problem by imposing a bound on the value to be optimized
- For example, instead of calculating the shortest path, we can cast it as a decision problem as follows:
  - Is there a path between vertices u and v with distance at most K units?

Example Conversions

- Subset Sum *Optimization* Problem:
  - Given a pair (S, *max*), where S is a set of positive sizes and *max* is a positive number. What is the subset of S whose sum is as large as possible, but no larger than *max*?
- Subset Sum *Decision* Problem:
  - Given a triple (S, *min*, *max*), where S is a set of positive sizes and *min* and *max* are positive numbers. Is there a subset of S whose sum is at least *min*, but no larger than *max*?
- **Exercise**: State the 0-1 Knapsack Problem, then convert it to a decision problem

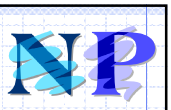
Nondeterministic Algorithms

- A problem is solved through a two stage process
  1. Nondeterministic stage (guessing)
     - Generates a proposed solution (random guess)
     - E.g., some completely arbitrary string of characters, s, is written at some designated place in memory
  2. Deterministic stage (verification/checking)
     - A deterministic algorithm, then begins execution (may read or ignore the input s)
     - It eventually halts with an output of yes or no or may go into an infinite loop

Nondeterministic Algorithm (MST)

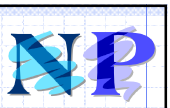
- Decision Problem: Does graph G have a spanning tree with total weight at most K?
- Algorithm (**high level**):
  1. Guess (non-deterministic): randomly choose a set of edges from G and call the subgraph formed by these edges T
  2. Verification (deterministic): check whether T forms a spanning tree with weight at most K

Verifiability and NP



- Claim: checking whether or not an input string is a solution to a problem is not harder than computing a solution
  - So a deterministic solution is at least as hard to compute as the corresponding non-deterministic decision algorithm
- *Polynomial-time non-deterministic algorithm:*
  - a non-deterministic algorithm whose verification stage can be done in polynomial time

The Class NP



- **Definition**: NP is the set of all decision problems that can be solved by non-deterministic polynomial-time algorithms.
  - Consists of the problems whose proposed solutions can be "verified" (stage two) in polynomial time
  - A problem in the class NP is characterized by the extremely large number of possibilities one might have to try before finding an answer
    - We could imagine running a polynomial-time non-deterministic algorithm repeatedly and gradually improving our proposed solution (for optimization problems)
    - (but never being sure it's optimal without trying all possibilities)

Nondeterministic Algorithms

- The number of steps in a nondeterministic algorithm is the sum of the steps in the two phases
  - Steps to write s (the guess)
  - Steps to check s
- If both steps take polynomial time, then the problem is said to be a member of NP
  - One could say that problems in NP are those whose solutions are easy to check
- **Note**:
  - We don't know how many times this algorithm will have to be repeated before a solution is generated and verified
    - May need to repeat it exponential or factorial number of times
  - May arise that there is no natural interpretation for "solutions" and "proposed solutions"

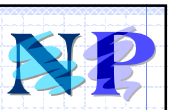
Tractable vs. Intractable for non-deterministic algorithms

- All problems (solvable and unsolvable) are simplified to a corresponding decision problem
- The problem then becomes a decision about whether or not a guess is a valid solution
  - **Tractable (feasible) problems**:
    - a valid guess can be deterministically generated in polynomial time, i.e., the problems in complexity class P
  - **Undecidable problems**:
    - there can be no algorithm to validate a guess
    - (must be proven mathematically, e.g., the halting problem)
  - **Intractable (infeasible) problems**:
    - no polynomial time algorithm to deterministically generate a valid guess has yet been found
    - NP-Complete and NP-Hard problems are considered intractable, but we are not sure
    - Includes problems in NP and others not in NP

Examples

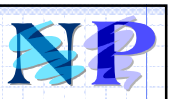
MST and Sorting

NP Example1 (MST)



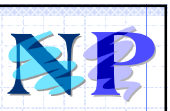
- Problem: Does graph G have a spanning tree with total weight at most K?
- Algorithm (**high level**):
  1. Guess (non-deterministic): randomly choose a set of edges from G and call the subgraph formed by these edges T
  2. Verification (deterministic): check whether T forms a spanning tree with weight at most K
- To show that MST ∈ NP:
  - Need to show that the algorithm that verifies the guess runs in polynomial time? If yes, then this problem is a member of the class NP.
  - However, need more details in step 2 to be sure the algorithm runs in polynomial time, i.e., how is checking done.

NP Example1 (MST)



- Problem: Does graph G=(V, E) have a spanning tree with total weight at most K?
- Algorithm (**more detail**):
  1. Non-deterministically choose a set T of edges from E
  2a. Is the number of edges in T equal to n-1?
  2b. Is the subgraph formed by T connected?
  2c. Is the total weight of the edges in T at most K?
- Analysis:
  - Step 1 takes O(m) time
  - Step 2a takes O(1) depending how T is represented
  - Step 2b takes O(n) time to check whether T is connected (BFS)
  - Step 2c takes O(n) to sum the edge weights
  - Conclusion: Checking takes O(n) time, so this algorithm runs in polynomial time; thus this problem is a member of class NP.

NP Example1 (MST)

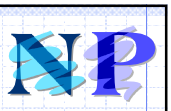


- Problem: Does graph G=(V, E) have a spanning tree of weight at most K?
- Non-deterministic Algorithm (**full detail**):

Phase 1. Non-deterministically choose a set of edges from E and insert them into a graph T (could specifically choose n-1 edges)
Phase 2.

```
Algorithm checkMST(G, T)
1  if T.numEdges() ≠ n-1 then return no
2  H ← new empty hash table
3  W ← 0
4  for all e ∈ T.edges()  do
5       W ← W + weight(e)  {add up the edge weights}
6       (u,v) ← e.endVertices()
7       H.insertItem(u, u)
8       H.insertItem(v, v)
9  if  W > K  ∨  H.size() ≠ n {Does this determine if G is not connected?}
10      then return no
11      else return yes
```


NP Example2 (Sorting)



- Problem: Decide if a sequence of integers S can be rearranged into non-decreasing order using comparator ≤
- **Exercise**:
  Prove that this decision problem is a member of complexity class NP?

NP Example2 (Sorting)

- Problem: Decide if a sequence of numbers S can be rearranged into non-decreasing order using comparator $\leq$
- Algorithm:
  1) Non-deterministically insert all integers in S into a sequence T
  2) Test that $T_i \leq T_{i+1}$ for i ← 0 to n-2
- Conclusion: Testing takes O(n) time, so the checking algorithm runs in polynomial time and the sorting problem is in NP.

NP Example2 (version 2)

- Problem: Decide if a sequence of numbers S can be rearranged into non-decreasing order using comparator $\leq$
- Algorithm:
  1) Non-deterministically insert all numbers in S into a sequence T
  2) **Algorithm** checkSort(T, $\leq$)

```
if T.size() = 0 or 1 then
        return yes
    else
        for i ← 0 to n-2
            if T.elemAtRank(i) ¬≤ T.elemAtRank(i+1)
                then return no

        return yes
```

- Conclusion: Checking takes O(n) time, so the checking algorithm runs in polynomial time and the sorting problem is in NP.

NP Example2 (version 3)

- Problem: Decide if a sequence of numbers S can be rearranged into non-decreasing order using comparator $\leq$
- Algorithm:
  1) Non-deterministically insert the integers from S into a sequence T
  2) **Algorithm** checkSort(T, $\leq$)

```
Q ← HeapSort(S, ≤)                          { O(n log n) }
for i ← 0 to S.size()-1  do     { verify Q matches guess T}
    if Q.elemAtRank(i) ¬= T.elemAtRank(i)
        then return no
                                            {loop runs in O(n) time}
return yes                                  { O(1) }
```

- Conclusion: Checking takes O(n log n) time, so the checking algorithm runs in polynomial time and the sorting problem is in NP.

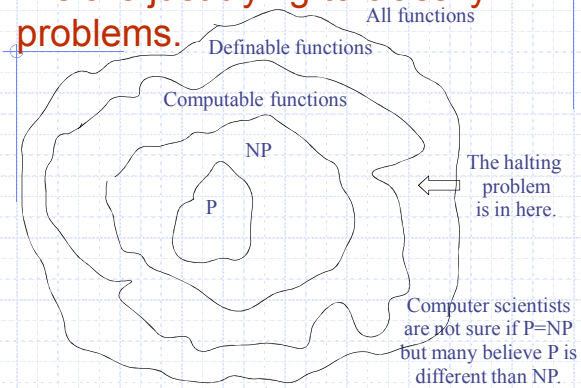
Prove: P ⊆ NP

Claim:
Any problem that can be solved in polynomial time is a member of NP

Proof:
Non-deterministic Polynomial Algorithm:
1. Non-deterministically output a proposed solution (a guess)
2. Compute the correct solution in polynomial time (p(n) time)
3. Check whether the proposed solution matches the correct solution in polynomial time (always O(n) time, why?)

- Consider the previous homework problem to output the permutations of a sequence of integers
- What is the corresponding decision problem?
- Is this decision problem a member of NP?

We are just trying to classify problems.



Main Point

2. A problem is in NP (nondeterministic polynomial) if there is a polynomial time algorithm for checking whether or not a proposed solution (guess) is a correct solution.
   Natural law always computes all possible paths to the goal and chooses the one with the least action and maximum positive benefit.

Problem Reductions

The reason we need to convert to a decision problem!

Problem Reductions

- A problem A reduces to problem B if we can efficiently transform instances of A into instances of B such that solving the transformed instance of B yields the answer to the original instance of A
  - The key is that the transformation (reduction) must preserve the correctness of the answer to A
- More specifically
  Let a be an arbitrary instance of A.
  Let *R*(a) produce an instance of problem B.
  Let *f* be an algorithm that correctly solves instances of A.
  Let *g* be an algorithm that correctly solves instances of B.
  R is a valid reduction of instances of A to instances of B, if for all a ∈ A, *g*(*R*(a)) produces the correct answer to the original problem a, i.e., *g*(*R*(a)) = *f*(a)

Example reduction

- Consider the following decision problems:
  Subset Sum: Given a triple (S, *min*, *max*), where S is a set of positive sizes and *min* and *max* are positive numbers. Is there a subset of S whose sum is at least *min*, but no larger than *max*?
  0-1 Knapsack: Given a triple (S, *W*, *b*), where S is a set of (benefit, weight) pairs, *W* is a positive weight, and *b* is a positive benefit. Is there a subset of S such that the total weight is at most *W* with total benefit at least *b*?

Reduction of Subset Sum to 0-1 Knapsack

Let the (S, min, max) be an instance of Subset Sum. The transformation would use the following algorithm:

```
Algorithm reduceSSto0-1K(S, min, max)
 Input: a Sequence S of numbers and the limits min and max from Subset Sum
 Output: a Sequence P of pairs (representing benefit and weight) and the values of w and b for 0-1 Knapsack
   P ← new empty Sequence
   for i ← 0 to S.size()-1 do
       val ← S.elemAtRank(i)
       P.insertLast( (val, val) )
   return (P, max, min)  {pairs, maximum weight, minimum benefit}
```


Implications of Problem Reductions

- Reducing problem A to problem B means:
  - An algorithm to solve B can be used to solve A as follows:
    - Take input to A and transform it into input to B
    - Use algorithm to solve B to produce the answer for B which is the answer to A
  - Typically, instances of A are reduced to a small subset of the instances of B
  - If the transformation (reduction) takes polynomial time, then a polynomial solution to B implies that A can be done in polynomial time

Main Point

3. If a problem A can be reduced to another problem B, then a solution to B would also be a solution to A. Furthermore, if the reduction can be done in polynomial time, then A must be easier or of the same difficulty as B.
   Individual and collective problems are hard to solve on the surface level of the problem. However, if we go to the root, the source of creativity and intelligence in individual and collective life, we can enliven and enrich positivity on all levels of life.

Take home quiz

- *What is the relationship between memoization and dynamic programming?*
- *What are the differences?*
- *When might memoization be more efficient?*
- *When might dynamic programming be more efficient?*

Connecting the Parts of Knowledge with the Wholeness of Knowledge

1. All problems for which reasonably efficient (tractable) algorithms are known are grouped into the class P (polynomial-bounded). The class NP consists of problems that can be solved by non-deterministic polynomial-time algorithms.
2. Algorithms in class P can easily be shown to be members of class NP. Undecidable problems (such as halting) cannot be members of NP, since they cannot have an algorithm to verify a guess. Intractable problems are those that have an algorithmic solution, but no polynomial-time algorithm has yet been found.

3. **Transcendental Consciousness** is the field of all solutions, a taste of life free from problems.
4. **Impulses within Transcendental Consciousness**: The natural laws within this unbounded field are the algorithms of nature that efficiently solve all problems of the universe.
5. **Wholeness moving within itself:** In Unity Consciousness, one realizes the full dignity of cosmic life in the individual. We have the vision of possibilities – transcend to remove stress in the individual physiology and live our full potential free of problems.