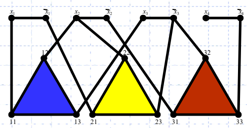
Lecture 15: Is P = NP?



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

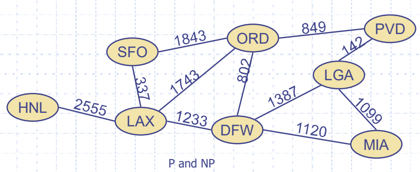
Complexity class NP is fundamental to complexity theory in computer science. Decision problems in the class NP are problems that can be non-deterministically decided in polynomial time. Non-deterministic decision algorithms have two phases, a non-deterministic phase and a deterministic phase. *Science of Consciousness:* 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 (mathematical)
- Strings over an alphabet (language)
- Language acceptors
- Nondeterministic computing
- Decision Problems
- Converting Optimization Problems to a Decision Problem
- Reductions of one decision problem to another

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., the upper bound is a polynomial $O(n^k)$).
  - Exception: any pseudo-polynomial time algorithm



Examples:

- In sorting a Sequence with n elements, the size of the input is $O(n)$
- The size of a graph G is $O(n+m)$
- The size of the set S of benefit-weight pairs is O(n) for the 0-1 Knapsack problem,
  - But what about the size of the knapsack, i.e., the number W?

Example: with numbers it's a little different

- What is the input size of input argument to function Fibonacci(*n*)?
- It is the number of bits needed to represent *n*
- How many bits are needed?
  - **log n** bits are needed to represent *n* in binary so the input size is **log n**

What is the running time of this function?

```
Algorithm FindFactor(n)
Input: integer n
Output: Returns a factor of n
    fact ← 2
    while fact ≤ n/2 do
        if (n mod fact) = 0 then // if fact is a factor of n
            return fact
        fact ← fact + 1
    return n              // if n is prime
```

- the running time is **O(n)**, but note that the input size is **log n**
- Therefore, the running time is exponential in the size of the input
- That is, $O(n) = O(2^{\log n})$
- In public key cryptography we talk about keys that are 256 bits long because then the search space for the factors of a key is 2^{256}

Intractability

- In general, 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
  - In graph theory, TSP is a complete graph
- Determine the order we should visit all of the cities (once), returning to the starting city at the end, while minimizing the total cost
- How many possible simple cycles are there in a complete graph G?

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 (i.e., with a polynomial time upper bound).

   Science of Consciousness: 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
  - The decision problem answers the question of whether or not a solution exists
  - 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?*

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?

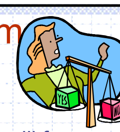
Example Conversions

- Minimum Spanning Tree *Optimization* Problem:
  - Given a Weighted Graph *G*, find a spanning tree of *G* with the minimum total weight?
  - What to do: convert to a decision problem by adding another parameter to the optimization problem, i.e., a *max value* if we are searching for a minimum or a *min value* if we are searching for a maximum.
- Minimum Spanning Tree *Decision* Problem:
  - Given a pair (*G*, *max*), where G is a graph. Does there exist a spanning tree of G whose total weight is at most *max*?

Example Conversions

- 0-1 Knapsack *Optimization* Problem:
  - Given a pair (S, *W*), where S is a set of benefit-weight pairs and *W* is the size of the knapsack. Find the subset of S whose total weight is at most W but whose total benefit is as large as possible.
  - What should we do?
    - Since we are maximizing benefit, we add a minimum total benefit as a parameter to the problem
- 0-1 Knapsack *Decision* Problem:
  - Given a triple (S, *W*, *min*), where S is a set of benefit-weight pairs and *W* is the size of the knapsack. Does there exist a subset of S whose total weight is at most W but whose total benefit is at least *min*?

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") where the problem asks whether or not a solution exists?

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

Exercise: Example Conversion

- Longest Common Subsequence *Optimization* Problem:
  - Given a pair of strings (*S1*, *S2*), find the longest common subsequence of strings S1 and S2.
- Longest Common Subsequence *Decision* Problem:
  - Given a triple (*S1, S2*, *min*), where S1 and S2 are strings, does there exist a common subsequence of S1 and S2 with length at least *min*?

19

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

20

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

21

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 *max*
    - 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 *max*

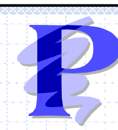
22

From now on All Problems must be 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 input strings to the set {yes, no}

23

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** (yes or no) 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)

24

Polynomial-Time Algorithms

- Are some problems solvable in polynomial time?
  - Yes: every algorithm we've studied provides a polynomial-time solution to some problem (except for the pseudo-polynomial algorithms)
  - Thus the algorithms we've studied so far 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**

P and NP 25

25

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 impractical in practice

P and NP 26

26

So far in the course
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.
   - problems with a lower bound that is not polynomial like generating all permutations or the set of subsets
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, Hamiltonian Cycle
   - Leads us to the theory of NP

P and NP 27

27

Nondeterministic Decision Algorithms

- A problem is solved through a two stage process
  1. Nondeterministic stage (guessing)
     - Generate a proposed solution ***w*** (random guess)
     - E.g., some randomly chosen string of characters, ***w***, is written at some designated place in memory
  2. Deterministic stage (verification/checking)
     - A deterministic algorithm to determine whether or not ***w*** is a solution then begins execution
     - If ***w*** is a solution, then halt with an output of yes otherwise output no (or NOT_A_Solution)
  - If ***w*** is not a solution, then keep repeating steps 1 and 2 until a solution is found, otherwise we keep trying without halting

P and NP 28

28

The Class NP

- **Definition**: NP is the set of all decision problems that can be solved by **N**on-deterministic **P**olynomial-time algorithms.
- NP consists of the problems whose proposed solutions (guesses) 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 a solution

P and NP 29

29

Nondeterministic Algorithms

- The number of steps in a nondeterministic algorithm is the sum of the steps in the two phases
  - Steps to write s (size of the guess)
  - Steps to check s (running time of the verifier)
- If both steps take polynomial time, then the problem is said to be a member of NP
  - We sometimes say that problems in NP are those whose solutions are easy to check
  - Easy means can be done in polynomial time
  - Polynomial time means $O(n^k)$ for $k \geq 0$
- **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"

P and NP 30

30

Non-deterministic Algorithm

- We create a non-deterministic algorithm using verifier V
- We again assume that V returns NOT_A_Solution if the guess is not a **valid** solution

Algorithm isMemberOfL(x)
  result ← NOT_A_Solution
  while result = NOT_A_Solution **do**
1.   w ← randomly guess at a solution from search space
2.   result ← V(x,w) // V must run in polynomial time
  return result // allows returning no from V(x,w) when L ∈ P

In a proof that a language is a member of NP, our verifier has to run in polynomial time and has to be substitutable in place of V above.

Nondeterministic Algorithm (MST)

- Decision Problem: Does graph G have a spanning tree with total weight at most max?

Algorithm (**high level**):

1. Guess (non-deterministic): randomly choose a set of edges from G and place these edges in a Sequence T
2. Verification (deterministic): check whether the edges in T form a spanning tree with weight at most max

How would you implement the verification step 2?

2. Verification (deterministic): check whether the edges in T form a spanning tree with weight at most *max*

Verifiers take an instance of the problem, (G,max) here, and the guess T and determine whether or not T is a solution. The following would be an example of a correct interface:

Algorithm verifyMST(G,max,T)

Verifiability and NP

- Claim: checking whether or not an input guess (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
- ***N**on-deterministic **P**olynomial-time algorithm:*
  - a non-deterministic algorithm whose verification stage can be done in polynomial time

Example proofs of membership in NP

MST and Sorting: To do this we only have to show the existence of a Non-deterministic Polynomial Algorithm

NP Example1a (MST)

- Problem: Does graph G have a spanning tree with total weight at most max?

Algorithm (**high level**):

1. Guess (non-deterministic): randomly choose a set of edges from G and place these edges in Sequence T
2. Verification (deterministic): check whether the edges in T form a spanning tree with weight at most max

- 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 class NP. (Partial Credit)
  - However, need more details in step 2 to be sure the algorithm runs in polynomial time, i.e., how is checking done.

NP Example1b (MST)

- Problem: Does graph G=(V, E) have a spanning tree with total weight at most max?

NP Algorithm (**more detail, much better**):

1. Non-deterministically choose a Sequence T of edges from E
2. **Algorithm** verifyMST(G,max,T)
   - 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 max?

- Analysis: More details in the next slide.
  - Step 1 takes O(m) time
  - Step 2a takes O(1) depending how T is represented
  - Step 2b takes O(n+m) time to check whether T is connected (BFS)
  - Step 2c takes O(n) to sum the edge weights
  - Conclusion: Checking takes O(n+m) time, so this algorithm runs in polynomial time; thus this problem is a member of class NP.

NP Example1b (MST) (Details Using BFS Template-v1)

Problem: Does graph G=(V, E) have a spanning tree with total weight at most max?

NP Algorithm (even more details using BFS):

1. Non-deterministically choose a Sequence T of edges from E
2. **Algorithm** verifyMST(G,max,T)

```
if T.size() != G.numVertices()-1 then
    return NOT_A_Solution
for each e in G.edges() do
    setEdgeOfT(e, NO)
total ← 0
for each e in T.elements() do
    setEdgeOfT(e, YES)
    total ← total + weight(e)
isConnected ← BFS(G)
if isConnected /\ total ≤ max then
    return yes
else return NOT_A_Solution
```

```
Algorithm initResult(G)
    components ← 0
Algorithm preComponentVisit(G, v)
    components ← components + 1
Algorithm result(G)
    return (components = 1)
Algorithm preEdgeVisit(G, v, e)
    if getEdgeOfT(e) = NO then
        setLabel(e, SKIP)
```


Template Version of BFS

```
Algorithm BFS(G) {all components}
  Input graph G
  Output labeling of the edges of G as
   discovery edges and cross edges

   initResult(G)
   for all u ∈ G.vertices() do
      setLabel(u, UNEXPLORED)
      postInitVertex(u)
   for all e ∈ G.edges() do
      setLabel(e, UNEXPLORED)
      postInitEdge(e)
   for all v ∈ G.vertices() do
      if isNextComponent(G, v)
         preComponentVisit(G, v)
         BFScomponent(G, v)
         postComponentVisit(G, v)

   return result(G)

Algorithm isNextComponent(G, v)
   return getLabel(v) = UNEXPLORED
```

```
Algorithm BFScomponent(G, s) {1 component}
  setLabel(s, VISITED)
  Q ← new empty Queue
  Q.enqueue(s)
  startBFScomponent(G, s)
  while ¬Q.isEmpty() do
     v ← Q.dequeue()
     preVertexVisit(G, v)
     for all e ∈ G.incidentEdges(v) do
        preEdgeVisit(G, v, e)
        if getLabel(e) = UNEXPLORED
           w ← opposite(v, e)
           edgeVisit(G, v, e, w)
           if getLabel(w) = UNEXPLORED
              preDiscEdgeVisit(G, v, e, w)
              setLabel(e, DISCOVERY)
              setLabel(w, VISITED)
              Q.enqueue(w)
              postDiscEdgeVisit(G, v, e, w)
           else
              setLabel(e, CROSS)
              crossEdgeVisit(G, v, e, w)
     postVertexVisit(G, v)

  finishBFScomponent(G, s)
```


NP Example1b (MST) (Details using BFS Template-v2)

Problem: Does graph G=(V, E) have a spanning tree with total weight at most max?

NP Algorithm (even more details using BFS):

1. Non-deterministically choose a Sequence T of edges from E
2. **Algorithm** verifyMST(G,max,T)

```
if T.size() != G.numVertices()-1 then
    return NOT_A_Solution
for each e in G.edges() do
    setEdgeOfT(e, NO)
total ← 0
for each e in T.elements() do
    setEdgeOfT(e, YES)
    total ← total + weight(e)
hasCycle ← BFS(G)
if ! hasCycle /\ total ≤ max then
    return yes
else return NOT_A_Solution
```

```
Algorithm initResult(G)
    hasCycle ← false
Algorithm crossEdgeVisit(G, v, e, w)
    hasCycle ← true
Algorithm result(G)
    return hasCycle
Algorithm preEdgeVisit(G, v, e)
    if getEdgeOfT(e) = NO then
        setLabel(e, SKIP)
```


Ex. 1c (MST-matching guess with solution)

- 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 Sequence T (could specifically choose n-1 edges)

Phase 2.

Algorithm checkMST(G , max, T) // check if T contains same edges as in MST

```
if T.numEdges() ≠ n-1 then return NOT_A_Solution
Prim-Jarnik-MST(G) // recall that the edges in the MST are saved at the vertices
H ← new Dictionary(HT)
for all e ∈ T.edges() do
     H.insertItem(e, e)
W ← 0
for all v ∈ G.vertices() do // compare edges in T to edges in MST
     e ← getParent(v)
     if e ≠ null then
          W ← W + weight(e) {add up weights of edges in MST}
     if H.findElement(e) = NO_SUCH_ELEM then return NOT_A_Solution
if W ≤ max
    then return yes
    else return NOT_A_Solution
```


NP Example1d (MST ignoring guess)

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

Phase 1. Non-deterministically choose a set of edges from E and insert them into a Sequence T

Phase 2.

Algorithm checkMST(G, max, T) // can only be done like this if MST(G) is $O(n^k)$

```
Baruvka-MST(G) // ignore T; okay since MST runs in O(m log n)
W ← 0
for all e ∈ G.edges() do
     If getMSTLabel(e) = IN_MST then // Baruvka labels edges in MST
          W ← W + weight(e)    // add up weights of edges in MST
if W ≤ max
    then return yes
    else return no       // we definitively return a no answer
```

- The **only** time a verifier can return no is when the problem is a member of P, i.e., a solution, R, can be generated in polynomial time

Non-deterministic Algorithm

- We create a non-deterministic algorithm using verifier V
- We again assume that V returns NOT_A_Solution if the guess is not a **valid** solution

Algorithm isMemberOfL(x)
 result ← NOT_A_Solution
 while result = NOT_A_Solution **do**
1. w ← randomly guess at a solution from search space
2. result ← V(x,w) // must run in polynomial time
 return result // allows returning no from V(x,w) when L ∈ P

In a proof that a language is a member of NP, our verifier has to run in polynomial time and has to be substitutable in place of V above.

Deterministic implementation of the mathematical definition

- We assume that V(x,w) returns NOT_A_Solution if w is not a **valid** solution to help clarify what we are doing

Algorithm isMemberOfL(x)
 solutionSpace ← generate all the possible solutions & put in an iterator
  // note that the solution space could be exponential (power set)
  // or factorial (permutations), etc. cannot be done if infinite size
 result ← NOT_A_Solution
 while result = NOT_A_Solution Λ solutionSpace.hasNext() **do**
  w ← solutionSpace.nextObject()
  result ← V(x,w) // L ∈ NP, only requires that V(x,w) run in $O(n^k)$ time

 if result = NOT_A_Solution **then**
  return no
 else
  return result // this allows us to return no from V(x,w) when L ∈ P

When is L a member of NP?

- Membership of L in NP only requires that verifier V(x,w) run in $O(n^k)$ time when w is a **valid** solution
  - It does not matter when w is not a valid solution
  - However, if we will limit the structure of w to possible solutions, then V(x,w) will run in polynomial time on all input w as we would like
- This is why we say that L is a member of NP if it's easy (takes polynomial time) to verify (or recognize) when w is a **valid** solution

NP Example2 (Sorting)

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

NP Example2 (Sorting)

- Problem: Decide if a sequence of objects in S can be rearranged into non-decreasing order using comparator C
- Algorithm:
  1. Non-deterministically insert all objects in S into sequence T
  2a. Check that all elements of T are comparable using C
  2b. Test that C.lessOrEqual(T_i ,T_{i+1}) for all i ← 0 to n-2
- Conclusion: Testing takes O(n) time, so the checking algorithm runs in polynomial time and the sorting problem is a member of NP. (Partial credit since we need to specify that no is returned on 2a and NOT_A_Solution on 2b and does not show the interface for the verifier)

NP Example2 (version 2)

- Problem: Decide if a sequence of objects in S can be rearranged into non-decreasing order using comparator C

Algorithm:
 1) Non-deterministically insert the integers from S into a sequence T
 2) **Algorithm** verifySort(S, C, T)
  S ← Sort(S, C) { O(n log n) }
  for i ← 0 to S.size()-1 do { verify S matches guess T}
   if S.elemAtRank(i) ≠ T.elemAtRank(i)
    then return NOT_A_Solution
        {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.
- (Partial credit) Almost but somethings missing, what if elements in S cannot be sorted, i.e., cannot be compared using C?

NP Example2 (version 3)

- Problem: Decide if a sequence of objects in S can be rearranged into non-decreasing order using comparator C
- Algorithm:

1) Non-deterministically insert all objects one by one from S and put into a sequence T
2) **Algorithm** checkSort(S, C, T)

```
for i ← 0 to S.size()-1 do     { make sure elements in T can be sorted }
    if ¬C.isComparable(T.elemAtRank(i)) then
        return no              { definite no answer, loop runs in O(n) time }
S ← Sort(S, C)                          { O(n log n) }
for i ← 0 to S.size()-1 do     { verify S after sorting matches guess T}
    if S.elemAtRank(i) ≠ T.elemAtRank(i) then
        return NOT_A_Solution
  {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. (Full credit)

49

NP Example2 (version 4)

- Problem: Decide if a sequence of objects in S can be rearranged into non-decreasing order using comparator C
- Algorithm:

1) Non-deterministically insert all objects from S and put into a sequence T
2) **Algorithm** checkSort(S, C, T)

```
for i ← 0 to S.size()-1 do     { make sure elements in T can be sorted }
    if ¬C.isComparable(T.elemAtRank(i)) then
        return no              { loop runs in O(n) time }
                                       {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. (Full credit)

50

Prove: $P \subseteq NP$

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 ($O(n^k)$ time)
3. Check whether the proposed solution matches the correct solution in polynomial time (always p(n)=size of w time, why?)
4. Verify that the generated solution satisfies all decision criteria

51

Example: Power Set

- Consider the previous homework problem to output the set of all subsets, the power set
- What is the corresponding decision problem?
  - Verifier V(S, T) determines whether T the powerset of S?
- Is this decision problem a member of NP?

52

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/2 do
        if (n mod fact) = 0 then // if fact is a factor of n, add to sum
            sum ← sum + fact
return n
```


53

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.

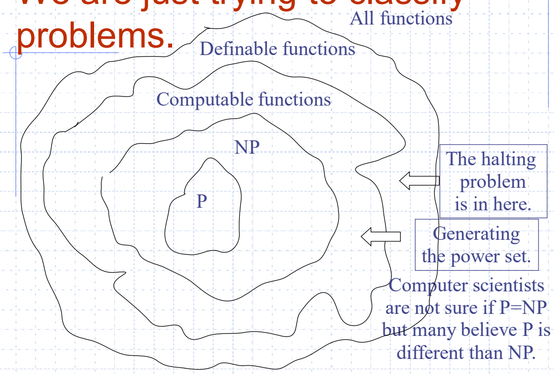
54

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, power set, set of permutations, etc.
- What is a definable function?
  - A function for which the mapping can be described with a mathematical formula or mathematical description
  - For example, the halting problem is definable, but not computable
- Are most functions definable or undefinable?

55

We are just trying to classify problems.



56

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 or a solution exists?
  - **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

57

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.

   Science of Consciousness: Natural law always computes all possible paths to the goal and chooses the one with the least action and maximum positive benefit.

58

Problem Reductions

The reason we need to convert to a decision problem!

59

Problem Reductions

- A problem A reduces to problem B if we can efficiently (easily) 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)

60

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 (P, *W*, *minB*), where P is a set of (benefit, weight) pairs, *W* is a positive weight, and *minB* is a positive benefit. Is there a subset of P such that the total weight is at most *W* with total benefit at least *minB*?

61

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}
```


62

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

63

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.

64

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?*
- *Or does it matter which approach is used?*

65

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 whose correct solutions can be recognized by 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 algorithm solution, but no polynomial-time algorithm has yet been found.

66

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.