Last Time

• Dynamic Programming
  – Solving a problem recursively by divide-and-conquer, then converting the algorithm to a bottom-up, iterative, array-based algorithm.
    • Fibonacci
    • Longest Common Subsequence problem
    • Subset Sum problem
  – Solving a problem by stepping through a recursive tree constructed by either taking or not taking some choice at each step.
    • Subset Sum problem
Reductions

• **Reduction** – Given problems A and B, we say that “A is reducible to B” (written as $A \leq B$) if there exists an algorithm to solve A that lacks only code for B.

  – We solve problem A by:
    1. Reading the input for A
    2. Converting it (if necessary) to the input for B
    3. Solving problem B
    4. Converting the output (if necessary) to suit problem A
    5. Wrap up problem A
Element Uniqueness Problem

• Input: Numbers $x_1, x_2, ..., x_n$
• Output: 0 if $\exists i, j$ such that $x_i = x_j$ and $i \neq j$
  1 otherwise

• Example Input: 54, 23, 42, -16, 17, 23, 95
• Example Output: 0
Element Uniqueness Problem

• Solution 1: Compare every element to every other element, and break if any two (different) elements are equal.
  – Complexity: $O(n^2)$
Element Uniqueness Problem

• Solution 2: Sort the list, then compare each element to the element which follows.
  – Complexity: $O(n \times \log(n))$

Element Uniqueness  Sorting

Read input $O(n)$  Sort input $O(n \times \log(n))$

Run comparisons $O(n)$
Reductions are Easy to Get Backwards

• NewProblem ≤ TravelingSalesman
  – Efficient work
  – Efficient work
  – Efficient work
  – <Call TravelingSalesman, O(n!)>
Reductions are Easy to Get Backwards

• TravellingSalesman ≤ NewProblem
  – Efficient work
  – Efficient work
  – Efficient work
  – <Call NewProblem, O(n)> 
  – Efficient work
  – Efficient work

• Now we have an (imaginary) efficient TS solution!
Matrix Multiplication

• Fact: Despite a lot of effort, no one knows a \( O(n^2) \) time algorithm for matrix multiplication.

• Problem 1: Matrix Multiplication
  – **Input:** \( n \times n \) matrices A, B
  – **Output:** A*B

• Problem 2: Matrix Squaring
  – **Input:** Matrix C
  – **Output:** \( C^2 = C*C \)
Matrix Multiplication

• To get an algorithm for squaring, reduce squaring to multiplication.

Problem SQUARE
read C
D = MULT(C,C)
output D

• Issue: Could there be an easier solution for squaring, one that doesn’t make use of multiplication?
Matrix Multiplication

• To show that squaring is at least as hard as multiplication, reduce multiplication to squaring.

Problem MULT
read A, B

construct C = \begin{bmatrix} B & 0 \\ A & 0 \end{bmatrix}

D = SQUARE(C)

output D[1][0]

D = \begin{bmatrix} B^2 & 0 \\ AB & 0 \end{bmatrix}
Matrix Multiplication

• To show that squaring is at least as hard as multiplication, reduce multiplication to squaring.

\[
\begin{align*}
\text{MULT} & \quad \text{SQUARE} \\
\text{Read A, B} & \quad \text{Create C} \\
\text{Output D}[1][0] & \quad \text{Resulting computation D}
\end{align*}
\]

\[O(n^2)\]
Some Time Complexity Classes

- **Class P** – The class of languages that are solvable in polynomial time.

- **Class NP** – The class of languages that have polynomial time verifiers, and are not solvable in polynomial time without introducing nondeterminism.

- **Verifier** – An algorithm $V$ for language $A$, where $A = \{ w \mid V$ accepts $\langle w, c \rangle \text{ for some string } c \}$. 
Is P = NP?

• Probably not, but we haven’t proven it yet.
• If you prove it, you’ll be set for life.
• (Don’t waste your time trying to prove it.)
The PATH Problem

\[ \text{PATH} = \{ (G, s, t) | G \text{ is a directed graph that has a directed path from } s \text{ to } t \} \]

- **Theorem:** \( \text{PATH} \in P \)
- **Proof:** We prove this theorem by presenting a polynomial time algorithm that decides \( \text{PATH} \).

\[ M = \text{"On input } (G, s, t), \text{ where } G \text{ is a directed graph with nodes } s \text{ and } t:\]

1. Place a mark on node \( s \).
2. Repeat the following until no additional nodes are marked:
   3. Scan all the edges of \( G \). If an edge \((a,b)\) is found going from a marked node \( a \) to an unmarked node \( b \), mark node \( b \).
   4. If \( t \) is marked, **accept**. Otherwise, **reject**.

- Steps 1 and 4 are only run once. Step 3 runs at most \( m \) times, where graph \( G \) has \( m \) nodes. Thus, the total number of steps is \( 1+1+m = O(m) \), which is polynomial time. ■
The HAMPATH Problem

\[ HAMPATH = \{ \langle G, s, t \rangle \mid G \text{ is a directed graph with a Hamiltonian path from } s \text{ to } t \} \]

- **Theorem:** \( HAMPATH \in NP \)
- **Proof:** We prove this theorem by presenting a polynomial time algorithm that decides \( HAMPATH \) nondeterministically.
  
  \( N = \) “On input \( \langle G, s, t \rangle \), where \( G \) is a directed graph with nodes \( s \) and \( t \):
  
  1. Write a list of \( m \) numbers, \( p_1, \ldots, p_m \), where \( m \) is the number of nodes in \( G \). Each number in the list is nondeterministically selected to be between 1 and \( m \).
  
  2. Check for repetitions in the list. If any are found, reject.
  
  3. Check whether \( s = p_1 \) and \( t = p_m \). If either fail, reject.
  
  4. For each \( i \) between 1 and \( m-1 \), check whether \( (p_i, p_{i+1}) \) is an edge of \( G \). If any are not, reject. Otherwise, accept.”

- All steps run in nondeterministic polytime. Now we need to show that a verifier exists that runs in deterministic polytime:
  
  \( V = \) “On input \( \langle w, c \rangle \), where \( w \) and \( c \) are strings:
  
  1. Simulate input \( w \) on NTM \( N \) and do same.”
NP-Completeness

- **NP-Completeness** – There exist certain problems in NP whose individual complexity is related to that of the entire class.
  - (In other words, each problem in this NP-Complete class is reducible to each other problem in this NP-Complete class.)
  - If you find an algorithm to solve Traveling Salesman, you have a method to break RSA cryptography!
  - SubsetSum is actually NP-Complete!
Any Questions?