Exam 3 – Notes

**NP Theory**

* Polynomial Runtimes
  + O(n^k) for some constant of k (except for Knapsack which is pseudo polynomial)
* Exponential Runtimes
  + O(k^n) for some k >1
* Pseudo Polynomial Runtimes
  + Polynomial to the magnitude of the input, but exponential to the size of the input
  + Any runetime based on log(x) is considered polynomial
* P = Polynomial
  + Runs in poly time
* NP = Nondeterministic Polynomial time
  + Can be verified in poly time
* Dos P = NP
  + If we can verify a problem in poly time, can we also solve it in poly time?
  + Is this true for all problems

**Pseudo Polynomial Runtimes**

* Knapsack is O(nB)
* For-Fulkerson is O(mC)
* N and m are based on the size of the inputs (n – objects, m – edges)
* B and C are magnitudes
  + They are a single number that increases the runtime, w/o needing more space

**Undecidable Problem**

* There exists problems that some inputs cannot be solved
* The problem is considered computationally impossible with unlimited time and space
* Decision problem: a problem that can be posed as a yes/no question of the input values
* Undecidable problem: a decision problem where a yes/no cannot be determined for all inputs
* Different than having no solution

**Halting Problem**

* Given a program P and an input I
* Decide if P(I) will terminate
  + True if it will terminate
  + False if it will not terminate (infinite loop)
* We can create a paradox such that we can’t determine an answer

**NP-Complete Reductions**

* Always reduce from A to B
* Known problem to an Unknown problem
* Writing your NP proof:
  + Demonstrate that problem B is in the class NP
    - Give steps on how to verify a solution for problem B in poly time
    - Give a runtime in O() format, and don’t forget to verify solution size equals the goal size
  + Demonstrate B is at least as hard as your chosen A
    - Show how an instance of A can be coverted to an instance of B in poly time
    - All instances of A must convert to an instance of B
    - Cannot choose the input of A
    - Force a solution to B, if there’s a solution to A
  + Show that B has a solution if and only if (IFF) A has a solution
    - Both directions are required

**SAT Reduction Tips & Tricks**

* NP Proof:
  + Make sure to verify the SAT assignments
    - O(nm) for normal SAT
    - O(m) if there’s a specific clause limit
  + Make sure to count anything
  + Verify any special conditions in the solution. Don’t verify the input
* Input:
  + Maintain CNF form for the boolean formula input
  + Maintain equivalent logic
  + Only add literals and clauses
    - Based on the size of clauses needed for a custom SAT (ex: 4SAT), you need to transform the clauses such that one single clause evaluates to the original clause when passing the modified CNF into the custom SAT algorithm.
  + Never add True or False “clause”
* Output:
  + Remove added literals or variables
  + Not the added clauses b/c clauses are not part of the output
* Correctness:
  + Establish equivalency of the two SAT problems after transformation
  + Then argue IFF

**Graph Reductions Tips & Tricks**

* NP Proof:
  + Verify the graph very carefully
  + If graph contains clique, but not separated, it must be separated first
  + Finding a clique in an arbitrary graph is not polynomial
  + Make sure to count anything or check the size of solutions
  + Avoid degree counting which is usually wrong or is much harder
* Input:
  + Do not try to find what you are looking for before modifying it
  + Force your unknown problem to find the solution for you
  + Remember your budget, goal, etc.
  + Be careful accidentally adding extra candidate vertices or edges
* Output:
  + Isolate the thing you are looking for form the unknown output
  + Just removing added vertices or edges is not enough
* Correctness:
  + Argue why the solution for the unknown can only be found in G’ if the original G has a solution to your known

**Set Reductions Tips & Tricks**

* Ex: Vertex Cover - > Hitting Set
* NP Proof:
  + Verify each set and its contents or intersects
  + Count anything or check the size of the sets
  + Consider runtimes in how large the largest set can be
* Input:
  + Allowed previous set problems (Hitting Set, Set Cover)
  + Can also use a graph or SAT
  + Usually a generalization problem. Create a problem that matches the problem to be solved
  + Consider runtimes for size of all allowed items
* Output:
  + Covert back to problem A
* Correctness:
  + Argue why the problems are equivalent
  + Then state IFF

**Summation Reduction**

* Ex: Subeset-Sum - > Knapsack
* NP Proof:
  + Straightforward to verify
    - Does the solutions set add up to what we’re looking for?
  + Remember adding is O(n log t) where t is the sum of what we’re looking for
* Input:
  + Force sums to represent the solution
  + Consider bit or integer representations
  + Consider forcing inequalities to converge on an equity
* Output:
  + Convert back to problem A
* Correctness:
  + Argue why the summations are equivalent
  + Argue IFF

**Linear Programming**

* Remember:
  + Standard form
  + Calculate duals
  + Feasibility and boundness
  + How to do the math (graphing)