- See previous practice problem sets for instructions.
- I repeat one very important instruction: You must explicitly state all *non-trivial* assumptions that you make. When in doubt, over-communicate.

1. Show using elementary arguments[1] that $\log_2 n! = \Theta(n \log_2 n)$ holds. Note that this involves showing that two separate statements involving inequalities hold. You may assume that $\log_2 x > \log_2 y$ holds for all real numbers x, y ; $x > y \geq 2$.

2. The input to the SEARCH problem consists of an array A of n integers, and an integer a. The goal is to find an index i of A such that $A[i] = a$, or to report—correctly—that there is no such index i.

   (a) Use a decision tree argument to show that *any* algorithm that solves the SEARCH problem and is allowed to query the input only using comparisons of the following forms:

   - Is $x < y$?
   - Is $x = y$?

   must make $\Omega(\log_2 n)$ comparisons in the worst case.

   Here, each of x, y could be an indexed array element of the form $A[j]$, or the integer a.

   (b) Use an adversary argument to derive a lower bound of $\Omega(n)$ steps on the running time of any algorithm that solves the SEARCH problem. How do you reconcile this with the running time of Binary Search?

3. Use an adversary argument to derive a lower bound of $\Omega(\log_2 n)$ in the comparison-based model, on the running time of any algorithm that solves the SEARCH problem is a *sorted* array.

4. The input to the EASYDUPLICATESCHECK problem consists of a *sorted* integer array A with n elements, and the question is whether A contains two identical elements (duplicates).

   (a) Provide the complete pseudocode for an algorithm that solves this problem using at most $(n-1)$ pairwise comparisons of elements of A.

   (b) Use an adversary argument to prove that no algorithm that solves EASYDUPLICATESCHECK correctly, can make fewer than $(n-1)$ pairwise integer comparisons—of elements of A—in the worst case.

[1] That is, *without* appealing to heavy machinery such as Stirling's Approximation.

5. Prove that each of the following problems is NP-complete. Note that this consists of showing two properties[2] for each problem. For the purpose of this question the set of "known" NP-complete problems consists of (i) those which were shown to be NP-complete in class, and (ii) those which you prove to be NP-complete.

   (a)

   1-IN-3 SAT

   - Input: A 3-CNF formula ϕ.
   - Question: Is there a way to assign (Boolean) values to the variables of ϕ such that each clause contains *exactly* one true literal? *Hint:* Look for a reduction from 3-SAT.

   (b)

   SET COVER

   - Input: A finite universe U, a collection $\mathcal{F} = S_1, S_2, \ldots, S_m$ of (some, not necessarily all) subsets of U, and an integer k.
   - Question: Does there exist a sub-collection $\mathcal{F}' \subseteq \mathcal{F}$ such that (i) $|\mathcal{F}'| \leq k$, and (ii) every element of U is present in some element of $\mathcal{F}'$?

     (Such a sub-collection $\mathcal{F}'$ is said to be a *set cover* of the universe U.)

   (c)

   HITTING SET

   - Input: A finite universe U, a collection $\mathcal{F} = S_1, S_2, \ldots, S_m$ of subsets of U, and an integer k.
   - Question: Does there exist a subset $X \subseteq U$; $|X| \leq k$ such that each element of $\mathcal{F}$ has a non-empty intersection with X?

     Such a set X is said to be a *hitting set* for the collection $\mathcal{F}$, because it "hits" every set in $\mathcal{F}$.

     Hint: Look for a reduction from VERTEX COVER.

[2] What are they?

(d)

Exact k-Path

- Input: A finite undirected graph G, and an integer k.
- Question: Does G contain a simple path with exactly k vertices?

A *simple path* is one which does not have repeated vertices or edges.
You may assume that the following problem is NP-complete: Hamiltonian Path: Given an undirected graph G, decide if G has a Hamiltonian path. A Hamiltonian path in a graph G is a simple path that visits every vertex of G exactly once.

(e)

k-Path

- Input: A finite undirected graph G, and an integer k.
- Question: Does G contain a simple path with at least k vertices?

(f)

Rooted k-Path

- Input: A finite undirected graph G, a vertex r of G, and an integer k.
- Question: Does G contain a simple path that starts at vertex r and contains at least k vertices?

(g)

Tethered k-Path

- Input: A finite undirected graph G, vertices u, v of G, and an integer k.
- Question: Does G contain a simple path that has u, v as its end-vertices, and contains at least k vertices?

(h)

0,1 Integer Programming

- Input: An $m \times n$ integer matrix A and an $m \times 1$ integer vector b.
- Question: Does there exist an $n \times 1$ vector x whose elements are from the set $\{0, 1\}$, such that $Ax = b$?

  Hint: Look for a reduction from 3SAT.

(i)

Clique

- Input: A finite undirected graph G, and an integer k.
- Question: Does there exist a subset C of the vertex set of G such that $|C| = k$, and for every two vertices u, v in C, the edge $\{u, v\}$ is present in G?

  (Such a subset C is called a *clique* of size k in G.)

(j)

Subgraph Isomorphism

- Input: Two finite undirected graphs G_1, G_2.
- Question: Is G_1 a subgraph of G_2? That is, is there a way to delete zero or more vertices and/or edges of G_2 to get a graph H which is, up to renaming of vertices, identical to G_1?

  Hint: Use the result from the previous part.

6. Assume that you are given black-box access to a polynomial-time algorithm 3SatSolver that solves the *decision* problem 3SAT. Describe how you can use this algorithm to find satisfying assignments to—satisfiable!—3SAT instances in polynomial time.

   That is: you can call a function 3SatSolver and pass it a 3SAT formula ϕ as its argument. The call 3SatSolver(ϕ) will take time polynomial in the length of the formula ϕ, and will correctly return Yes or No according to whether formula ϕ has at least one satisfying assignment, or not. You are required to design an algorithm FindSatAssignment which takes a 3SAT formula φ as input, runs in time polynomial in the length of the formula φ, and does the following: If φ is *un*satisfiable, then the call FindSatAssignment(φ) will return No. If φ is *satisfiable*, then the call

FindSatAssignment(φ) will return an assignment of values to the variables of φ that satisfies φ.

7. The problem CLIQUE-3 is the CLIQUE problem, restricted to input graphs in which every vertex has degree at most 3. Consider following argument for CLIQUE-3 being NP-hard. Is the argument correct? If not, why?

   > We know that the CLIQUE problem is NP-complete in general graphs, so it is enough to present a reduction from CLIQUE-3 to CLIQUE. Given a graph G with vertices of degree at most 3 and an integer k as inputs, the reduction leaves the graph and the integer unchanged; clearly the output of the reduction is a valid input instance for the CLIQUE problem. And, trivially, G has a clique of size k if and only if ... G has a clique of size k. This proves the correctness of the reduction and, therefore, the NP-hardness of CLIQUE-3.

8. Recall the definition of the relation $\leq_P$ as defined in class. Let A, B be two decision problems. Show that if $A \leq_P B$ and problem B can be solved in polynomial time, then problem A can also be solved in polynomial time.

9. Show that the relation $\leq_P$ as defined in class, is a transitive relation.

10. Recall the reduction from 3SAT to 3COLOUR that we saw in class. Assuming the construction of gadgets involved in the reduction, formally argue that the reduction is correct, and that it takes polynomial time.

11. What is wrong with the following argument?

    > Recall that 2SAT can be solved in polynomial time. We use a polynomial-time algorithm for 2SAT as a black-box to solve 3SAT in polynomial time, as follows: Let the input 3SAT formula be $\phi(x_1, \ldots, x_n) = C_1 \wedge C_2 \wedge \cdots \wedge C_k$. We look at some clause in ϕ with 3 literals, and split it into two formulas on one of its literals (say x), one where we set x to true, and another where we set x to false. We simplify each of the split formulas so they are smaller. We do this until we get formulas that all have clauses with at most 2 literals, and then solve these formulas using the polynomial-time algorithm for 2SAT. Thus we have solved the 3SAT instance in polynomial time.