1. A ***three-dimensional matching*** in an undirected graph G is a collection of vertex-disjoint triangles. A three-dimensional matching is *maximal* if it is not a proper subgraph of a larger three-dimensional matching in the same graph.

   (a) Let M and M' be two arbitrary maximal three-dimensional matchings in the same underlying graph G. ***Prove*** that $|M| \le 3 \cdot |M'|$.

   Solution: Consider an arbitrary triangle $\triangle$ in M. If $\triangle$ does not share a vertex with any triangle in M', then $M' + \triangle$ is a 3D matching, contradicting our assumption that M' is maximal. We conclude that every triangle in M shares at least one vertex with a triangle in M'.

   On the other hand, consider an arbitrary triangle $\triangle'$ in M'. Each vertex of $\triangle'$ is a vertex of *at most* one triangle in M, because the triangles in M are vertex-disjoint. Thus, the number of triangles in M is at most the number of *vertices* in M'. ■

   Rubric: 4 points = 2 for every triangle in M touches a triangle in M' + 2 for each triangle in M touches at most three triangles in M'.

   (b) Finding the *largest* three-dimensional matching in a given graph is NP-hard. Describe and analyze a fast 3-approximation algorithm for this problem.

   Solution: The following brute-force algorithm computes a maximal 3D matching in ***$O(VE)$ time***.

```
DUMB3DMATCHING(G):
  unmark every vertex of G
  M ← ∅
  for all vertices v
      for all edges uv
          for all edges vw
              if u ≠ w and uw ∈ E and u, v, w are all unmarked
                  mark u, v, w
                  add uvw to M
  return M
```

   Let M denote the 3D matching computed by our algorithm, and let OPT denote the largest 3D matching in G. Part (a) immediately implies $|OPT|/|M| \le 3$. ■

   Rubric: 3 points = 2 for algorithm + 1 for time analysis. No penalty for $O(V^3)$. We didn't ask for a proof of the approximation ratio, so no proof is required for full credit.

   (c) Finding the *smallest maximal* three-dimensional matching in a given graph is NP-hard. Describe and analyze a fast 3-approximation algorithm for this problem.

   Solution: Same algorithm as part (b).

   Let M denote the 3D matching computed by our algorithm, and let OPT denote the smallest maximal 3D matching in G. Part (a) immediately implies $|M|/|OPT| \le 3$. ■

   Rubric: 3 points = 2 for algorithm + 1 for time analysis. "See part (b)" is worth exactly the same number of points as earned in part (b). We didn't ask for a proof of the approximation ratio, so no proof is required for full credit.

2. Let $G = (V, E)$ be an arbitrary dag with a unique source s and a unique sink t. Suppose we compute a random walk from s to t, where at each node v (except t), we choose an outgoing edge $v \to w$ uniformly at random to determine the successor of v.

   (a) Describe and analyze an algorithm to compute, for every vertex v, the probability that the random walk visits v.

Solution: For any vertex v, let $Prob(v)$ denote the probability that our random walk visits v. This function satisfies the following recurrence, where $outdeg(u)$ denotes the number of edges leaving u.

$$Prob(v) = \begin{cases} 1 & \text{if } v = s \\ \sum_{u \to v} Prob(u)/outdeg(u) & \text{otherwise} \end{cases}$$

We need to compute this function for every vertex v.

We can memoize this function into the dag itself, adding a field $v.Prob$ to every node v, and we can evaluate the function for all v in *forward* topological order, in ***$O(V+E)$ time***.

```
RandomWalkProbabilities($G$):
  topologically sort $G$
  for all vertices $v$ in topological order
      if $v = s$
          $v.Prob \gets 1$
      else
          $v.Prob \gets 0$
          for all edges $u \to v$
              $v.Prob \gets v.Prob + u.Prob/u.outdeg$
```

Equivalently, we can evaluate this function for all v by by performing a depth-first search of the reversed graph G^R starting at t.

```
RandomWalkProbabilities($G$):
  $t.Prob \gets$ RecRWP($t$)
```

```
RecRWP($v$):
  $v.Prob \gets 0$
  for all edges $u \to v$
      $v.Prob \gets v.Prob +$ RecRWP($u$)$/u.outdeg$
  return $v.Prob$
```

■

Rubric: 5 points: standard dynamic programming rubric.

(b) Describe and analyze an algorithm to compute the expected number of edges in the random walk.

Solution: Linearity of expectation implies that the expected number of *vertices* in the random walk is exactly $\sum_v \mathit{Prob}(v)$. So the following algorithm computes the expected number of *edges* in the random walk in $\boldsymbol{O(V+E)}$ ***time***.

```
RandomWalkExpectedLength(G):
    RandomWalkProbabilities(G)
    $\ell \leftarrow 0$
    for all vertices $v$
        $\ell \leftarrow \ell + v.\mathit{Prob}$
    return $\ell - 1$
```

■

Rubric: 5 points = 4 for algorithm + 1 for running time. Standard DP rubric for solutions that use DP directly.

3. Consider the following solitaire game. The puzzle consists of an $n \times m$ grid of squares, where each square may be empty, occupied by a red stone, or occupied by a blue stone. The goal of the puzzle is to remove some of the given stones so that the remaining stones satisfy two conditions: (1) every row contains at least one stone, and (2) no column contains stones of both colors. For some initial configurations of stones, reaching this goal is impossible.

   Prove that it is NP-hard to determine, given an initial configuration of red and blue stones, whether the puzzle can be solved.

Solution: We show that this puzzle is NP-hard by reducing from 3SAT.

Let Φ be a 3CNF boolean formula with n variables $x_1, x_2, \ldots, x_n$ and m clauses. We transform this board into a puzzle in polynomial time as follows. The puzzle grid has n columns, one for each variable in Φ, and m rows, one for each clause in Φ. For all indices i and j, we place a stone at position (i, j) as follows:

- If the variable x_j appears in the ith clause of Φ, we place a blue stone at (i, j).
- If the negated variable $\overline{x_j}$ appears in the ith clause of Φ, we place a red stone at (i, j).
- Otherwise, we leave cell (i, j) blank.

We claim that this puzzle has a solution if and only if Φ is satisfiable. This claim immediately implies that solving the puzzle is NP-hard. We prove our claim as follows:

$\Longrightarrow$ First, suppose Φ is satisfiable, and consider an arbitrary satisfying assignment. For each index j, remove stones from column j according to the value assigned to x_j:

- If $x_j =$ TRUE, remove all red stones from column j.
- If $x_j =$ FALSE, remove all blue stones from column j.

In other words, remove precisely the stones that correspond to FALSE literals. Because every variable s either TRUE or FALSE, no column contains stones of both colors. On the other hand, each clause of Φ must contain at least one TRUE literal, and thus each row still contains at least one stone. We conclude that the puzzle is solvable.

$\Longleftarrow$ On the other hand, suppose the puzzle is solvable; consider an arbitrary solution. For each index j, assign a value to x_j depending on the colors of stones left in column j:

- If column j contains blue stones, set $x_j =$ TRUE.
- If column j contains red stones, set $x_j =$ FALSE.
- If column j is empty, set x_j arbitrarily.

In other words, assign values to the variables so that the literals corresponding to the remaining stones are all TRUE. Each row still has at least one stone, so each clause of Φ contains at least one TRUE literal, so this assignment satisfies Φ. We conclude that Φ is satisfiable.

The reduction clearly requires only polynomial time, even if we use brute force. ■

Rubric: 10 points: standard NP-hardness rubric. This solution is more verbose than necessary for full credit.

4. Suppose you are given a bipartite graph $G = (L \sqcup R, E)$ and a maximum matching M in G. Describe and analyze fast algorithms for the following problems:

   (a) Insert(e): Insert a new edge e into G and update the maximum matching. (You can assume that e is not already an edge in G, and that $G + e$ is still bipartite.)

   (b) Delete(e): Delete the existing edge e from G and update the maximum matching. (You can assume that e is in fact an edge in G.)

   Your algorithms should modify M so that it is still a maximum matching, faster than recomputing a maximum matching from scratch.

Solution: First transform G into a directed graph H by directing all edges in M from R to L and all edges not in M from L to R. The resulting directed graph is essentially the residual graph of M. The unmatched vertices in L become sources in H, and unmatched vertices in R become sinks in H.

Both of our algorithms use the following augmentation subroutine: Find a path from any source (in L) to any sink (in R) using whatever-first search. If we find such a path, reverse every edge in that path; then the new matching M consists of all right-to-left edges in H. If we do not find such a path, M is unchanged. The augmentation subroutine requires $O(E)$ time.

We can Insert edge uv into G by adding $u \rightarrow v$ to H, and then calling the augmentation subroutine. Similarly, we can Delete edge uv from G by removing the corresponding edge from H (either $u \rightarrow v$ or $v \rightarrow u$) and then calling the augmentation subroutine. Both algorithms run in $\boldsymbol{O(E)}$ ***time***. ■

Rubric: 10 points = 5 for Insert + 5 for Delete. Standard graph reduction rubric (scaled). This is not the only correct solution.

5. (a) Suppose you *randomly* choose $n/2$ of your n coins to put on one pan of the Balance, and put the remaining $n/2$ coins on the other pan. What is the probability that the two subsets have equal weight?

Solution: $\frac{n-2}{2n-2} = \frac{1}{2} - \frac{1}{2n-2}$

The pans are balanced if and only if either both fake coins are on the left, or both fake coins are on the right. Suppose we randomly index the coins from 1 to n, and put the coins indexed from 1 to $n/2$ in the left pan, and the coins indexed from $n/2+1$ to n in the right pan. No matter what index the light coin gets, there are $n-1$ equally likely indices for the heavy coin, of which $n/2-1$ make the pans balanced. Thus, the pans are balanced with probability $(n/2-1)/(n-1) = (n-2)/2(n-1)$. ■

Rubric: 2 points. 1½ for $1/2 \pm o(1)$. No proof required.

(b) Describe and analyze a randomized algorithm to identify the two fake coins. What is the expected number of times your algorithm uses the Balance? To simplify the algorithm, you may assume that n is a power of 2.

Solution (random split then binary search): First we repeatedly split the n coins *at random* into two subsets of size $n/2$, until we find a pair of subsets of different size. At this point, each subset has exactly one fake coin, and we know whether that coin is light or heavy. For each subset, we then find the fake coin in $O(\log n)$ tests via binary search.

```
FINDFAKECOINS($X$):
    $L, H \leftarrow$ SPLITFAKECOINS($X$)
    return FINDLIGHTCOIN($L$), FINDHEAVYCOIN($H$)
```

```
SPLITFAKECOINS($X$):
    repeat
        randomly split $X$ into equal-size subsets $L$ and $R$
    until $L$ and $R$ have different total weight
    if $L$ is heavier than $R$
        swap $L \leftrightarrow R$
    return $L, R$
```

Each iteration of SPLITFAKECOINS(X) splits the coins with probability $n/(2n-2)$, so the expected number of times SPLITFAKECOINS uses the Balance is exactly $(2n-2)/n = 2-2/n = O(1)$.

Alternatively, instead of recombining the coins after an unsuccessful split, we can treat each subset "recursively". However, standard recursion performs a *depth*-first search of the recursion tree; to keep the algorithm efficient, we must perform a *breadth*-first search instead.

```
SplitFakeCoins($X$):
  $Q \leftarrow$ empty queue
  push $X$ into $Q$
  repeat forever
      pull $S$ from $Q$
      ***randomly*** split $S$ into equal-size subsets $L$ and $R$
      if $L$ is lighter than $R$
          return $L, R$
      else if $L$ is heavier than $R$
          return $R, L$
      else
          push $L$ into $Q$
          push $R$ into $Q$
```

The algorithm is performing a breadth-first search of a perfect binary tree, where the internal nodes represent tests and the n leaves represent randomly permuted coins. The search stops when we reach the least common ancestor x of the two fake coins. We immediately have

$$\mathrm{E}[\#\text{ tests}] \le \sum_{d=0}^{\lg n} \Pr[\text{depth}(x) \ge d] \cdot (\#\text{nodes with depth } d)$$

Part (a) implies inductively that $\Pr[\text{depth}(x) \ge d] \le 2^{-d}$, and there are trivially exactly 2^d nodes at depth d. We conclude that the expected number of tests performed by this algorithm is exactly $\lg n + 1$.

```
FindLightCoin($X$):
  if $|X| = 1$
      return the only coin in $X$
  split $X$ into equal-size subsets $L$ and $R$
  if $L$ is lighter than $R$
      return FindLightCoin($L$)
  else
      return FindLightCoin($R$)
```

```
FindHeavyCoin($X$):
  if $|X| = 1$
      return the only coin in $X$
  split $X$ into equal-size subsets $L$ and $R$
  if $L$ is lighter than $R$
      return FindHeavyCoin($R$)
  else
      return FindHeavyCoin($L$)
```

Each of these algorithms performs exactly $\log_2|X|$ tests, because both algorithms are equivalent to binary search.

In fact, we can speed up this algorithm slightly by recursively splitting into three subsets instead of two. For simplicity, assume n is twice a power of 3.

```
FindLightCoin($X$):
  if $|X| = 1$
      return the only coin in $X$
  split $X$ into equal-size subsets $L, M, R$
  if $L$ is lighter than $R$
      return FindLightCoin($L$)
  else if $L$ is heavier than $R$
      return FindLightCoin($R$)
  else
      return FindLightCoin($M$)
```

```
FindHeavyCoin($X$):
  if $|X| = 1$
      return the only coin in $X$
  split $X$ into equal-size subsets $L, M, R$
  if $L$ is lighter than $R$
      return FindHeavyCoin($R$)
  else if $L$ is heavier than $R$
      return FindHeavyCoin($L$)
  else
      return FindLightCoin($M$)
```

Each of these algorithms performs exactly $\log_3|X| < 0.6309 \log_2|X|$ tests.

The overall expected number of tests is $O(\log n)$, using either pair of search algorithms. ■

Rubric: 8 points = 4 for splitting the fake coins + 4 for binary search (including 1 point each for time analysis). These are not the only correct algorithms that perform $O(\log n)$ expected tests. These solutions are more verbose than necessary for full credit.

6. Suppose you are given a set L of n line segments in the plane, where each segment has one endpoint on the vertical line $x = 0$ and one endpoint on the vertical line $x = 1$, and all $2n$ endpoints are distinct. Describe and analyze an algorithm to compute the largest subset of L in which no pair of segments intersects.

Solution (dynamic programming): Start by sorting the left endpoints of L and separately sorting the right endpoints of L. For each index i, let $\ell[i]$ denote the index of the left endpoint that matches the ith lowest right endpoint, and let $r[i]$ denote the index of the right endpoint that matches the ith lowest left endpoints. Thus, $\ell[r[i]] = r[\ell[i]] = i$.

Let *MaxDisjoint*(i, j) denote the maximum number of disjoint segments in L connecting the lowest i left endpoints with the lowest j right endpoints. This function obeys the following recurrence.

$$\mathit{MaxDisjoint}(i,j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ \mathit{MaxDisjoint}(i-1,j) & \text{if } r[i] > j \\ \max\left\{\begin{array}{c} \mathit{MaxDisjoint}(i-1,j) \\ 1 + \mathit{MaxDisjoint}(i-1, r[i]-1) \end{array}\right\} & \text{otherwise} \end{cases}$$

We need to compute *MaxDisjoint*(n, n). To memoize this recurrence, we can fill an array *MaxDisjoint*$[1..n, 1..n]$ in standard row-major (or column-major) order in $\boldsymbol{O(n^2)}$ ***time***. ■

Solution (graph reduction): We construct a directed acyclic graph G as follows:

- G contains a vertex for each input segment.
- G contains an edge from u to v if and only if each endpoint of segment u is above the corresponding endpoint of segment v.

We can construct G in $O(n^2)$ time by brute force. Every path in G corresponds with a subset of the input segments with no intersections, and vice versa. Thus, to compute the largest subset of disjoint segments, it suffices to compute the ***longest path*** in G, in $O(V + E) = \boldsymbol{O(n^2)}$ ***time***, using the dynamic programming algorithm described in class and in the notes. ■

Solution (smartass reduction): Suppose the input segments are specified by two arrays $L[1..n]$ and $R[1..n]$, where $(0, L[i])$ and $(1, R[i])$ are the endpoints of the ith segment. Without loss of generality, we can assume the array L is sorted in increasing order. Then any subset of *disjoint* input segments corresponds to an *increasing* subsequence of the array R and vice versa. Thus, to compute the largest subset of disjoint segments, it suffices to compute the ***longest increasing subsequence*** of R in $\boldsymbol{O(n^2)}$ ***time***, using the dynamic programming algorithm described in class and in the notes.[1] ■

Rubric: 10 points: standard dynamic programming and/or reduction rubric. These are not the only correct solutions. Yes, the last solution is worth full credit. +5 extra credit for $O(n \log n)$ time.

[1]In fact, the notes describe an $O(n \log n)$-time algorithm!

Some Useful Inequalities

Let $X = \sum_{i=1}^{n} X_i$, where each X_i is a 0/1 random variable, and let $\mu = \mathrm{E}[X]$.

- **Markov's Inequality:** $\Pr[X \ge x] \le \mu/x$ for all $x > 0$.
- **Chebyshev's Inequality:** If $X_1, X_2, \ldots, X_n$ are pairwise independent, then for all $\delta > 0$:

$$\Pr[X \ge (1+\delta)\mu] < \frac{1}{\delta^2\mu} \quad \text{and} \quad \Pr[X \le (1-\delta)\mu] < \frac{1}{\delta^2\mu}$$

- **Chernoff Bounds:** If $X_1, X_2, \ldots, X_n$ are fully independent, then for all $0 < \delta \le 1$:

$$\Pr[X \ge (1+\delta)\mu] \le \exp\left(-\delta^2\mu/3\right) \quad \text{and} \quad \Pr[X \le (1-\delta)\mu] \le \exp\left(-\delta^2\mu/2\right)$$

Some Useful Algorithms

- **Random(k):** Returns an element of $\{1, 2, \ldots, k\}$, chosen independently and uniformly at random, in $O(1)$ time. For example, Random(2) can be used for a fair coin flip.
- **Ford and Fulkerson's maximum flow algorithm:** Returns a maximum (s,t)-flow f^* in a given flow network in ***$O(E \cdot |f^*|)$ time***. If all input capacities are integers, then all output flow values are also integers.
- **Orlin's maximum flow algorithm:** Returns a maximum (s,t)-flow in a given flow network in ***$O(VE)$ time***. If all input capacities are integers, then all output flow values are also integers.
- **Orlin's minimum-cost flow algorithm:** Returns a minimum-cost flow in a given flow network in ***$O(E^2 \log^2 V)$ time***. If all input capacities, costs, and balances are integers, then all output flow values are also integers.

Some Useful NP-hard Problems:

3Sat: Given a boolean formula in conjunctive normal form, with exactly three distinct literals per clause, does the formula have a satisfying assignment?

MaxIndependentSet: Given an undirected graph G, what is the size of the largest subset of vertices in G that have no edges among them?

MaxClique: Given an undirected graph G, what is the size of the largest complete subgraph of G?

MinVertexCover: Given an undirected graph G, what is the size of the smallest subset of vertices that touch every edge in G?

MinSetCover: Given a collection of subsets $S_1, S_2, \ldots, S_m$ of a set S, what is the size of the smallest subcollection whose union is S?

MinHittingSet: Given a collection of subsets $S_1, S_2, \ldots, S_m$ of a set S, what is the size of the smallest subset of S that intersects every subset S_i?

3Color: Given an undirected graph G, can its vertices be colored with three colors, so that every edge touches vertices with two different colors?

HamiltonianCycle: Given a graph G (either directed or undirected), is there a cycle in G that visits every vertex exactly once?

FeasibleILP: Given a matrix $A \in \mathbb{Z}^{n \times d}$ and a vector $b \in \mathbb{Z}^n$, determine whether the set of feasible integer points $\max\{x \in \mathbb{Z}^d \mid Ax \le b, x \ge 0\}$ is empty.

HydraulicPress: And here ve go!

Common Grading Rubrics

(For problems out of 10 points)

General Principles:

- Faster algorithms are worth more points, and slower algorithms are worth fewer points, typically by 2 or 3 points (out of 10) for each factor of n. Partial credit is scaled to the new maximum score, and all points above 10 are recorded as extra credit.
- A clear, correct, and correctly analyzed algorithm, no matter how slow, is always worth more than "I don't know". An incorrect algorithm, no matter how fast, may be worth nothing.
- Proofs of correctness are required on exams if and only if we explicitly ask for them.

Dynamic Programming:

- 3 points for a **clear English specification** of the underlying recursive function = 2 for describing the function itself + 1 for describing how to call the function to get your final answer. We want an English description of the underlying recursive *problem*, not just the algorithm/recurrence. In particular, your description should specify precisely the role of each input parameter. **No credit for the rest of the problem if the English description is is missing; this is a Deadly Sin.**
- 4 points for correct recurrence = 1 for base case(s) + 3 for recursive case(s). **No credit for iterative details if the recursive case(s) are incorrect.**
- 3 points for iterative details = 1 for memoization structure + 1 for evaluation order + 1 for time analysis. Complete iterative pseudocode is *not* required for full credit.

Graph Reductions:

- 4 points for a complete description of the relevant graph, including vertices, edges (including whether directed or undirected), numerical data (weights, lengths, capacities, costs, balances, and the like), source and target vertices, and so on. If the graph is part of the original input, just say that.
- 4 points for other details of the reduction, including how to build the graph from the original input, the precise problem to be solved on the graph, the precise algorithm used to solve that problem, and how to extract your final answer from the output of that algorithm.
- 2 points for running time of the overall algorithm, expressed as a function of the original input parameters, *not* just the number of vertices and edges in the graph.

NP-hardness Proofs:

- 3 points for a complete description of the reduction, including an appropriate NP-hard problem to reduce from, how to transform the input, and how to transform the output.
- 6 points for the proof of correctness = 3 for the "if" part + 3 for the "only if" part.
- 1 points for "polynomial time".